

大道至简

— C++ STL

(标准模板库)精解

The Greatest Truth is The Simplest —

Precision Expounding on C++ STL (Standard Template Library)

闫常友 编著

一杯清茶，一卷书，开卷有益。

—— C++ STL程序员的灯塔

A Cup of Tea, A Volume of Book.
Reading This Book Enriches Our Minds.
——The Beacon for C++ STL Programmer

 机械工业出版社
CHINA MACHINE PRESS



闫常友 博士后，高级工程师，九三学社社员，电力系统及其自动化专业，IEEE计算机协会会员，热爱计算机语言和软件开发，尤其对C/C++系列有独到的见解和深刻认识。对计算机语言的长期使用和实践，积累了些许的体会，抛砖引玉，写出来供大家斟酌。希望对C++程序员提升开发能力有所帮助。已出版书籍：《跟我学Visual C++ 6.0》，《不要重复发明轮子——C++STL标准程序库开发指南》。



大道至简

——C++ STL（标准模板库）精解

闫常友 编著



机械工业出版社

众所周知，C++ 是在 C 的基础上发展起来的。它进一步扩充和完善了 C 语言，是一种面向对象的程序设计语言。经过几十年的发展，C++ 现已支持多种编程规范，其相应的 C++ 国际标准也在不断更新。C++ STL 现在是 C++ 标准程序库的一部分，其作用是对常用数据结构和算法进行封装——标准化组件，以便让程序员可以直接使用现成的算法和数据结构，这样既避免了重复开发，又提高了效率。

本书按照 C++ STL 的内容结构由浅入深地讲解了其应用开发技术。主要内容共计 17 章。第 1 章主要介绍了与 C++ STL 相关的基本概念和基础知识，并简要介绍了本书后面会用到的一些模板类型。第 2~17 章依次介绍了字符串类模板、容器、算法库、迭代器、数值计算模板、输入/输出类模板、异常处理类模板、通用工具类模板、语言支持类模板、检测类模板、国际化类模板、仿函数、配置器、原子操作类、线性控制类模板以及正则表达式模板等内容。由于 C 标准函数库的相关内容在程序开发中的使用频率较高，因此本书的附录部分给出了几乎所有传统 C 库函数中的数学计算类函数和数值转换类函数，以供读者使用。

本书适合已有一定 C/C++ 基础的读者阅读，也适合有意于深度钻研 C++ STL 技术的朋友们阅读。

图书在版编目 (CIP) 数据

大道至简：C++ STL (标准模板库) 精解/闫常友编著. —北京：机械工业出版社，2015. 5

ISBN 978-7-111-51399-5

I. ①大… II. ①闫… III. ①C 语言—程序设计 IV. ①TP312

中国版本图书馆 CIP 数据核字 (2015) 第 206435 号

机械工业出版社 (北京市百万庄大街 22 号 邮政编码 100037)

策划编辑：申永刚 责任编辑：申永刚 吴晋瑜

版式设计：霍永明 责任校对：张晓蓉 刘怡丹

封面设计：马精明 责任印制：乔宇

保定市中国画美凯印刷有限公司印刷

2015 年 11 月第 1 版第 1 次印刷

184mm × 260mm · 46.75 印张 · 1 插页 · 1158 千字

0001—3000 册

标准书号：ISBN 978-7-111-51399-5

定价：125.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

电话服务

网络服务

服务咨询热线：010-88361066

机工官网：www.cmpbook.com

读者购书热线：010-68326294

机工官博：weibo.com/cmp1952

010-88379203

金书网：www.golden-book.com

编辑热线：010-88379425

封面无防伪标均为盗版

教育服务网：www.cmpedu.com

前 言

本书详细介绍了C++标准模板库（C++ Standard Template Library, C++ STL）的所有内容。本书主要是参照ISO/IEC 14882 Second edition的内容编写的，同时也参考了最新的C++ 11国际标准（ISO/IEC 14882: 2011）。

C++经历了多年的发展，其新增特性较多，而C++ STL是众多成员辛勤劳动的结晶，其中大量的可重复代码为广大程序员提供了巨大的方便，节省了大量时间和人力。C++ STL的开发主要是 Alexander Stepanov、David Musser 以及 MengLee 三位大师，其中 Alexander Stepanov 被誉为“STL之父”。1994年7月，美国国家标准学会（ANSI）通过投票决定，将STL纳入C++标准，使之成为C++库的重要组成部分。

C++ STL作为C++的一部分，历经多次修改，经过多个程序员团队的“精加工”，已经不同于最初的版本。STL为编程人员提供了诸多方便和好处，以前用传统C++编写的复杂代码，现在通过使用STL，仅仅几句话就可以实现。在STL中，模板使用得可谓淋漓尽致。通过使用模板，用户可获得优质并且高效的代码。STL的优越性使其迅速流行起来，并且发展劲头强劲。近些年，国内的STL热也迅速升温，但相关的资料并不丰富，较好的资料更是少之又少。对程序员来说，掌握STL编程技术，并精通C++高级编程技术，是非常有必要的。

本书以广大程序员的角度，详细介绍了C++ STL标准库，对其中的模板技术更是进行了细致深入的讲解。书中通过大量例题（均由作者亲自编写或摘自MSDN）对各知识点进行实例讲解，希望读者认真阅读。

鉴于作者水平有限，书中难免存在不足之处，敬请广大读者批评指正。

目 录

前言

第 1 章 预备知识及简介 1

- 1.1 基本概念 1
 - 1.1.1 何谓“命名空间” 1
 - 1.1.2 头文件 2
 - 1.1.3 面向对象的程序设计 3
 - 1.1.4 C++ 中的声明和定义 6
 - 1.1.5 最简单的C++ 程序 8
 - 1.1.6 指针 10
 - 1.1.7 函数 12
 - 1.1.8 文件 16
 - 1.1.9 编译和链接 19
 - 1.1.10 程序启动和终止 20
 - 1.1.11 异常处理 20
 - 1.1.12 预处理命令 21
 - 1.1.13 宏 29
- 1.2 类模板定义 31
 - 1.2.1 类模板实例化 32
 - 1.2.2 类模板的成员函数 34
 - 1.2.3 类模板的静态成员 34
- 1.3 成员模板 36
- 1.4 友元模板 37
- 1.5 函数模板 38
- 1.6 类模板的参数 42
- 1.7 STL 简介 45
 - 1.7.1 STL 历史 45
 - 1.7.2 STL 组件 46
 - 1.7.3 STL 基本结构 46
 - 1.7.4 STL 编程概述 49
- 1.8 小结 51

第 2 章 字符串 52

- 2.1 字符串类库简述 52
- 2.2 字符串的特点 54
- 2.3 字符串类模板 (basic_string) 54
- 2.4 字符串通用操作 55

- 2.4.1 构造器和析构器 56
- 2.4.2 大小和容量 58
- 2.4.3 元素存取 (访问) 60
- 2.4.4 字符串比较 61
- 2.4.5 字符串内容的修改和替换 64
- 2.4.6 字符串联接 71
- 2.4.7 字符串 I/O 操作 71
- 2.4.8 字符串查找 72
- 2.4.9 字符串对迭代器的支持 76
- 2.4.10 字符串对配置器的支持 78
- 2.5 小结 79

第 3 章 容器——对象储存器 80

- 3.1 容器概念 80
- 3.2 序列式容器 83
 - 3.2.1 vector (向量) 类模板 83
 - 3.2.2 list (列表) 类模板 103
 - 3.2.3 deque (双端队列) 类模板 124
- 3.3 关联式容器 133
 - 3.3.1 set/multiset (集合) 类模板 133
 - 3.3.2 map/multimap (图) 类模板 145
- 3.4 特殊容器用法 163
 - 3.4.1 bitset (位集合) 类模板 163
 - 3.4.2 stack (栈) 类模板 166
 - 3.4.3 queue (队列) 类模板 169
 - 3.4.4 priority_queues (优先队列) 类模板 172
- 3.5 小结 175

第 4 章 STL 算法 176

- 4.1 算法库简介 176
- 4.2 非修改性算法 177
 - 4.2.1 for_each() 算法 177
 - 4.2.2 元素计数算法 181
 - 4.2.3 最小值和最大值算法 183
 - 4.2.4 搜索算法 184
 - 4.2.5 比较算法 193

4.3 修改性算法	197	6.1.4 复数类运算	251
4.3.1 复制	198	6.1.5 复数的超越函数运算	254
4.3.2 转换	200	6.2 数组(向量)运算	256
4.3.3 互换	204	6.2.1 类 valarray	257
4.3.4 赋值	205	6.2.2 数组子集类——类 slice 和类 模板 slice_ array	264
4.3.5 替换	207	6.2.3 类 gsllice 和类模板 gsllice_ array	266
4.3.6 逆转	208	6.2.4 类 mask_array	270
4.3.7 旋转	210	6.2.5 类 indirect_array	271
4.3.8 排列	211	6.3 通用数值计算	273
4.4 排序及相关操作算法	215	6.3.1 求和算法 accumulate()	273
4.4.1 全部元素排序	215	6.3.2 内积算法 inner_product()	274
4.4.2 局部排序	217	6.3.3 部分和算法 partial_sum ()	276
4.4.3 根据某个元素排序	219	6.3.4 序列相邻差算法 adjacent_ difference ()	277
4.4.4 堆(Heap)操作算法	221	6.4 全局性数学函数	279
4.4.5 容器合并、交集和差集算法	223	6.5 小结	281
4.4.6 搜索算法	227	第 7 章 输入/输出类模板	282
4.5 删除算法	229	7.1 IOSTream 简介	282
4.6 小结	232	7.1.1 stream 对象	282
第 5 章 迭代器——访问容器的接口	233	7.1.2 stream 类别	283
5.1 迭代器及其特性	233	7.1.3 stream 操作符	284
5.2 头文件 <iterator>	234	7.1.4 操控器	284
5.3 迭代器类型详述	234	7.2 IOSTream 基本类和标准 IOSTream 对象	285
5.3.1 输入型迭代器	234	7.2.1 头文件	285
5.3.2 输出型迭代器	235	7.2.2 标准 stream 操作符	285
5.3.3 前向型迭代器	235	7.2.3 stream 状态	289
5.3.4 双向型迭代器	236	7.2.4 标准输入和输出函数	293
5.3.5 随机访问型迭代器	236	7.3 格式化	298
5.3.6 vector 迭代器的递增和递减	237	7.3.1 格式标识	298
5.4 迭代器配接器	237	7.3.2 bool 类型数据的格式控制	300
5.4.1 逆向型迭代器	237	7.3.3 详解“字段宽度、填充字符和 位置调整”	300
5.4.2 插入型迭代器	239	7.3.4 正记号与大写字符	303
5.4.3 流型迭代器	241	7.3.5 数值进制	304
5.5 迭代器辅助函数	244	7.3.6 浮点数输出	305
5.5.1 前进 advance() 函数	244	7.3.7 一般性格式定义	306
5.5.2 距离 distance() 函数	245	7.4 类 streambuf	307
5.5.3 交换两个迭代器所指内容 iter_swap() 函数	246	7.4.1 流缓冲区	307
5.6 小结	248	7.4.2 缓冲区迭代器	309
第 6 章 数值计算类模板	249	7.4.3 自定义缓冲区	311
6.1 复数运算	249		
6.1.1 一个复数运算例题	249		
6.1.2 复数类成员函数	250		
6.1.3 复数类运算符	251		

7.5	基于字符串的流	318	9.2.4	特定算法	388
7.5.1	streambuf 类	318	9.2.5	C 函数库中的内存管理函数	389
7.5.2	类模板 basic_istream	319	9.3	堆的内存分配	389
7.5.3	类模板 basic_ostringstream	320	9.3.1	new 和 delete 运算符	389
7.5.4	类模板 basic_stringstream	321	9.3.2	分配固定维数的数组	390
7.6	基于文件的流	321	9.3.3	分配动态内存数组	390
7.6.1	文件标识及其使用	322	9.3.4	处理堆耗尽	391
7.6.2	随机访问	332	9.4	辅助功能	391
7.6.3	4 个类模板	336	9.4.1	数值极限	392
7.6.4	C 库中的文件存取功能概述	338	9.4.2	较大/较小值 (最大/最小值)	395
7.7	小结	340	9.4.3	两值交换	396
第 8 章	异常处理类模板	341	9.4.4	辅助性比较	398
8.1	异常的概念和基本思想	341	9.4.5	头文件 cstdlib 和 cstdint 简介	399
8.1.1	异常的概念	341	9.5	日期和时间	400
8.1.2	异常的分类	342	9.5.1	3 个类型	400
8.1.3	异常的捕捉和处理	344	9.5.2	结构体 (tm)	400
8.1.4	资源管理	346	9.5.3	相关时间函数	401
8.1.5	异常和效率	348	9.5.4	时间示例	403
8.1.6	异常的描述	349	9.6	模板类 auto_ptr	405
8.1.7	未捕捉的异常	352	9.6.1	auto_ptr 类构造函数	406
8.2	异常类及几个重要问题	353	9.6.2	类 auto_ptr 的成员及转换	407
8.2.1	类 exception	353	9.6.3	使用类 auto_ptr	407
8.2.2	调用 abort ()	360	9.7	小结	411
8.2.3	堆栈解退	362	第 10 章	语言支持类模板	412
8.2.4	错误代码	362	10.1	类型	412
8.2.5	异常的迷失	363	10.2	执行属性	412
8.2.6	异常处理的局限性	367	10.2.1	类模板 numeric_limits 及其成员	412
8.3	处理异常详述	369	10.2.2	float_round_style 和 float_denorm_style	416
8.4	异常的特殊处理函数	372	10.2.3	数值极限的特殊化	417
8.5	小结	373	10.2.4	C 库函数	418
第 9 章	通用工具类模板 (Utility)	374	10.2.5	应用举例	418
9.1	通用工具库简介	374	10.3	程序的启动和终止	426
9.1.1	相等比较	374	10.4	动态内存管理	427
9.1.2	小于比较	374	10.4.1	内存的分配和释放	428
9.1.3	复制构造	377	10.4.2	内存分配错误	431
9.1.4	配置器要求	377	10.4.3	应用举例	431
9.1.5	运算符	379	10.5	类型标识符	434
9.1.6	对组 (Pairs)	379	10.5.1	类 type_info	434
9.2	动态内存管理	385	10.5.2	类 bad_cast	435
9.2.1	默认配置器	385	10.5.3	类 bad_typeid	436
9.2.2	raw_storage_iterator	387	10.5.4	操作符 typeid	436
9.2.3	临时缓冲区 (Temporary Buffers)	387			

10.5.5	操作符 <code>dynamic_cast</code>	437	12.4.4	模板类 <code>collate</code>	506
10.5.6	应用举例	437	12.4.5	类 <code>time</code>	508
10.6	异常处理	439	12.4.6	模板类 <code>monetary</code>	515
10.6.1	类 <code>exception</code>	439	12.4.7	类 <code>message retrieval</code>	523
10.6.2	特殊异常处理	440	12.4.8	<code>Program-defined facets</code>	526
10.6.3	异常终止	441	12.4.9	C 库 <code>locale</code>	527
10.6.4	未捕获异常 (<code>uncaught_</code> <code>exception</code>)	441	12.5	细述使用刻面	528
10.6.5	应用举例	442	12.5.1	数值的格式化	528
10.7	其他运行支持	444	12.5.2	时间/日期的格式化	530
10.7.1	概述	444	12.5.3	货币符号的格式化	533
10.7.2	应用举例	445	12.5.4	字符的分类和转换	536
10.8	小结	448	12.5.5	字符串校勘	541
第 11 章	检测类模板详解	449	12.5.6	信息国际化	542
11.1	异常类	449	12.6	小结	543
11.1.1	类 <code>logic_error</code>	449	第 13 章	仿函数	544
11.1.2	类 <code>domain_error</code>	450	13.1	仿函数的概念	544
11.1.3	类 <code>invalid_argument</code>	451	13.1.1	仿函数的概念	544
11.1.4	类 <code>length_error</code>	452	13.1.2	仿函数的作用	545
11.1.5	类 <code>out_of_range</code>	454	13.2	预定义仿函数	553
11.1.6	类 <code>runtime_error</code>	455	13.3	辅助用仿函数	554
11.1.7	类 <code>range_error</code>	456	13.3.1	一元组合函数配接器	554
11.1.8	类 <code>overflow_error</code>	457	13.3.2	二元组合函数配接器	557
11.1.9	类 <code>underflow_error</code>	458	13.4	关系仿函数	558
11.2	断言	459	13.4.1	等于 (<code>equal_to</code> <code>< type > ()</code>)	558
11.3	错误编码	461	13.4.2	不等于 (<code>not_equal_to</code> <code>< type > ()</code>)	559
11.4	小结	462	13.4.3	小于 (<code>less < type > ()</code>)	560
第 12 章	国际化库详解	463	13.4.4	大于 (<code>greater < type > ()</code>)	561
12.1	国际化元素	464	13.4.5	大于等于 (<code>greater_equal</code> 和小于等于 (<code>less_equal</code>)	561
12.2	多种字符编码	464	13.5	逻辑仿函数	562
12.2.1	宽字符和多字节文本	464	13.5.1	谓词	562
12.2.2	字符特性	465	13.5.2	逻辑仿函数	563
12.2.3	特殊字符国际化	467	13.6	算术仿函数	567
12.3	类 <code>locale</code>	467	13.6.1	加、减、乘、除仿函数	568
12.3.1	类 <code>locale</code> 概述	467	13.6.2	求余仿函数和求反仿函数	569
12.3.2	类 <code>locale</code> 的 <code>facet</code>	470	13.7	其他类型的仿函数	571
12.3.3	区域表示和混合区域表示	473	13.7.1	证和映射	571
12.3.4	流和区域	477	13.7.2	仿函数 <code>hash</code> 和 <code>subtractive_</code> <code>rng</code>	574
12.3.5	刻面的处理	477	13.8	适配器	574
12.4	标准 <code>locale</code> 的分类	479	13.8.1	成员函数适配器	575
12.4.1	类 <code>ctype</code>	480			
12.4.2	数值类的类 <code>locale</code>	496			
12.4.3	刻面 <code>numeric_punctuation</code>	503			

13.8.2 其他适配器	581	16.3.2 lock 模板类	639
13.9 小结	590	16.3.3 call_once	646
第 14 章 配置器	591	16.4 条件变量	647
14.1 使用配置器	591	16.4.1 类 condition_variable	648
14.2 C++ STL 默认的配置器 (标准配置器)	593	16.4.2 类 condition_variable_any	655
14.3 自定义配置器	594	16.5 模板类 future	659
14.4 配置类的详细讨论	595	16.5.1 模板类 future_error、future_errc 和 future_category 以及共享 状态	659
14.4.1 型别	595	16.5.2 模板类 promise	662
14.4.2 配置类的成员函数	595	16.5.3 模板类 future	666
14.4.3 广义配置器	596	16.5.4 模板类 shared_future	670
14.4.4 动态存储	597	16.5.5 仿函数 async	671
14.4.5 C 风格的分配	597	16.5.6 模板类 packaged_task	673
14.5 未初始化的内存	598	16.6 小结	677
14.6 配置器示例	600	第 17 章 正则表达式	678
第 15 章 原子运行库模板	602	17.1 定义及要求	678
15.1 头文件 <atomic> 简介	602	17.2 类模板 basic_regex	679
15.1.1 无锁属性	602	17.2.1 类模板 basic_regex 的声明	679
15.1.2 3 个模板	602	17.2.2 名称空间 std:: regex_constants	685
15.1.3 原子模板的常规操作	603	17.2.3 类 regex_error	687
15.1.4 头文件中的模板函数及算术 运算函数	605	17.2.4 类模板 regex_traits	688
15.1.5 原子类型 atomic_flag	609	17.2.5 类 basic_regex 的使用	692
15.2 顺序及一致性	610	17.3 类模板 sub_match 和 match_results	696
15.3 原子类型	611	17.3.1 类模板 sub_match	696
15.3.1 模板类 atomic	614	17.3.2 类模板 match_results	698
15.3.2 针对整型数据的特殊化模板	616	17.4 正则表达式相关的 3 种算法	704
15.3.3 针对指针的特殊化模板	619	17.4.1 正则匹配算法 regex_match	704
15.4 小结	621	17.4.2 正则搜索算法 regex_search	707
第 16 章 线程控制类模板	622	17.4.3 正则替换算法 regex_replace	708
16.1 要求和性能	622	17.5 正则表达式的迭代器	711
16.1.1 异常	622	17.5.1 迭代器 regex_iterator	711
16.1.2 本地句柄	622	17.5.2 迭代器 regex_token_iterator	713
16.1.3 时序规定	622	17.6 小结	719
16.1.4 可锁定类型	623	附录 部分 C 函数库详解	720
16.2 线程类	623	附录 A 数学函数	720
16.2.1 线程类成员变量 id	624	A.1 数学函数库中的宏	720
16.2.2 线程类成员函数	624	A.2 浮点计算减法协议开关	721
16.2.3 命名空间 this_thread	625	A.3 数学库中的宏函数	721
16.2.4 线程示例	626	A.4 三角函数和反三角函数	722
16.3 互斥	629	A.5 指数和对数函数	724
16.3.1 mutex 模板类	630	A.6 幂函数和绝对值函数	726

A. 7 误差和 gamma 函数	727	B. 1 字符转整数函数 (atoi ()	
A. 8 近似取整函数	728	和 atol ())	731
A. 9 求余函数	729	B. 2 字符型转换浮点型函数 (atof ()	
A. 10 操作处理函数	730	和 atol ())	732
A. 11 最大值、最小值和正差函数	730	B. 3 整型数转字符串函数 (itoa (), ltoa ()	
A. 12 浮点乘法函数	730	和 ultoa ())	732
A. 13 比较函数 (宏)	731	B. 4 浮点数转换字符串函数	734
附录 B 数据类型转换	731		

第 1 章

预备知识及简介

模板是C++的一个新特性。在现今的C++标准模板库中，几乎所有东西都被设计为模板形式。如果不支持模板，就无法使用标准模板库。模板是实现代码重用的一种工具，即针对一个或多个尚未明确的类型，编写一套函数或类型，以供用户使用。通过使用模板，C++允许推迟对某些类型的选择，直到需要对模板进行处理时才使用。使用模板，还可以使程序员针对不同的相似特性开发出更加大众化的代码。

本章首先介绍了C/C++语言中常用的一些基本概念；其次介绍了类模板定义、成员模板、函数模板、友元模板及类模板的参数等内容；最后介绍了模板库的相关知识。

1.1 基本概念

本节主要介绍C/C++语言中常用的一些基本概念。通过学习本节内容，读者应能正确使用这些概念，以便更加方便、高效地开发代码。

1.1.1 何谓“命名空间”

命名空间（Namespace）是指标识符的可见范围或者有效范围。定义命名空间的目的是为了防止出现名称冲突现象。命名空间是C++中一个较新的特性。为了将多个程序员开发的代码便捷、高效地组合起来，防止出现重复的函数名、类名等，命名空间将不同的代码封装在自己的有效范围内。

命名空间需要使用using来声明，并且每个命名均需要使用using。标准模板库（Standard Template Library, STL）采用命名空间技术解决了大规模软件开发的难题。C++ STL内的所有标识符都被定义为一个名为std的命名空间中，因而可以直接使用指定的标志符号std。例如：

```
std::cout << 2.0 << std::endl;
```

如果在上述语句之前添加以下声明：

```
using std::cout;
```

```
using std::endl;
```

那么上述语句可以写为：

```
cout << 2.0 << endl;
```

如果嫌麻烦，需要逐个声明名称，可以直接声明、使用命名空间std，例如：

```
using namespace std;  
...  
cout << 2.0 << endl;
```

显而易见,使用C++标准库的命名空间 `std` 之后,可以任意使用标准库中的函数和变量。参见例 1-1。

例 1-1

```
#include <iostream>
#include <stdio.h>
using namespace std;
void main()
{
    cout << "欢迎您开始步入C++ 标准模板库!" << endl;           //输出“欢迎您开始步入C++ 标准模板库!”
    getchar();                                                   //等待输入任意键,之后程序退出
    return;
}
```



提示 在例 1-1 中,使用了命名空间 `std`,即在 `main()` 函数前添加如下语句:

```
using namespace std;
```

之后在 `main()` 函数中,可以直接使用 `cout` 函数和 `endl` 函数。语句 `using namespace std` 可以放在源代码的任意位置,不同的位置代表 `std` 不同的有效范围。



总结 本小节主要讲述了命名空间的概念及其最简单的使用方法,并通过例 1-1 使读者对命名空间有一个直接的认识。

1.1.2 头文件

在上一节的例 1-1 中,有 `#include <iostream>` 这样的语句,其中 `iostream` 就是被 `include` 语句包含的头文件。在原来的 C 语言中,头文件都是以 `.h` 的形式存在的;而在 C++ 中,头文件只有名字,没有扩展名,例如 `iostream`。这部分头文件均在原有名字前添加字符 `c` 来加以标志,例如 `cmath`、`cstring` 等。这类头文件的使用方法如下所示:

```
#include <cmath>
#include <cstring>
```

值得注意的是:在书写头文件名时,一定要认真,不要添加其他字符,例如空格。如果添加了其他字符,有些编译程序会提示无法找到该头文件(Visual C++ 6.0 编译程序可以避免类似的问题)。

头文件的功能主要是将原程序片段收集到一起,形成一个提供给编译程序的文件。一般情况下,头文件中只包含各种声明、常量定义、预编译、注释、类型定义、模板定义等。常规的函数定义、数据定义、导出的模板定义等,不能出现在头文件中。

C++ 标准模板库头文件是标准模板库的外在表现形式。使用标准模板库的唯一途径就是包含相应的头文件。标准模板库头文件是没有后缀的。



提示 请关注头文件 `cmath` 的使用,并注意例 1-2 中的中文注释。

例 1-2

```
#include <iostream>
#include <cmath> //使用头文件 cmath
#include <stdio.h>
using namespace std; //使用命名空间 std
void main()
{
    float pi=3.1415916;
    float radius=0;
    float area=0.0;
    float girth=0.0;
    cout << "请输入圆的半径:" << endl;
        cin >> radius;
    area=pi* pow(radius,2); //使用幂函数 pow();乘方
    girth=2* pi* radius;
    cout << "area = " << area << ";  girth = " << girth << endl;

    getchar(); //等待程序退出
    return;
}
```



提示 在例 1-2 中，使用了 C 语言幂函数 `pow (double x, double y)`。这个函数有两个参数 `x` 和 `y`，函数的功能是求解 `x` 的 `y` 次方。

标准模板库提供了大量的头文件，使用头文件主要是为了提供类、函数、变量的声明，以供用户方便地使用这些类、函数、变量等。

1.1.3 面向对象的程序设计

面向对象 (Object Oriented, OO) 是 C/C++ 语言解决问题的一种方法。之前的程序开发是面向过程的。面向过程的程序设计主要考虑解决问题的先后顺序、措施安排等，具有典型的过程性。而面向对象的程序设计主要是建立在各种对象基础上的软件开发，每一个具有独立特性和相应功能的个体均可以作为一个对象来加以处理。

人类社会历史充满了“抽象”的概念，例如天、地、道等。

在软件开发和程序设计领域，同样存在诸多“抽象”的对象。根据软件开发的需要，每个对象被赋予不同的属性和方法 (函数)。当该对象被定义时，用户同样可以方便地使用该对象的属性和方法。大多数程序员都曾学过 C++ 中类的概念。类的对象一般有构造函数、析构函数等。类的对象可以被复制、被传递或被作为其他类的成员，甚至有些类的成员还可以直接参加数学计算。

面向对象是当前计算机界关心的重点，它是 20 世纪 90 年代软件开发方法的主流。在对对象有了简单认识之后，广大程序员开始就面向对象的程序设计进行各种尝试和研究，历经多年的发展和变革，才有了今天面向对象软件开发技术的现状。如今，面向对象的概念和应

用已超越了程序设计和软件开发，并扩展到了各个领域。

C++ 类是一种将抽象类型转换为用户自定义类型的方法。该方法将数据表示和操纵数据的方法组合成一个整齐的包。对于类的实现，一般包括两方面：类的声明和类方法的定义。类的声明是指以数据成员的方式描述数据部分，以成员函数的形式描述公有接口。类方法的定义是指描述如何实现类的成员函数。通俗来说，类的声明提供了类的宏观概貌，而类的定义则提供了细节。下面举一个最简单的例子。



提示 请注意例 1-3 中的中文注释。类 Calculator 作为一个简单的计算器，用于实现一些最简单的数学计算功能。

例 1-3

```
#include <iostream> //包含输入/输出流功能的头文件
#include <cmath> //包含数学函数库的头文件
#include <stdio.h> //包含标准输入/输出的头文件
using namespace std; //使用命名空间 std
class Calculator //声明类 Calculator
{
public: //将下述定义为公共成员
    Calculator(); //构造函数
    ~ Calculator(); //析构函数
    float x; //成员变量 x
    float y; //成员变量 y
    float Adder(float x, float y); //加法运算
    float Substration(float x, float y); //减法运算
    float Multiplication(float x, float y); //乘法运算
    float Division(float x, float y); //除法运算
    float CPow(float x, float y); //乘方运算
    float CSqrt(float x); //开方运算
}; //类声明结束
Calculator::Calculator() //定义构造函数
{
    x=0;y=0;
}
Calculator::~~Calculator() //定义析构函数
{
}
float Calculator::Adder(float x, float y) //定义实现加法运算的成员函数
{
    float He=0;
    He=x+y;
    return He;
}
float Calculator::Substration(float x, float y) //定义实现减法运算的成员函数
{
    float Cha=0;
```

```
        Cha = x - y;
        return Cha;
    }
    float Calculator::Multiplication(float x, float y)           //定义实现乘法运算的成员函数
    {
        float Ji = 0;
        Ji = x * y;
        return Ji;
    }
    float Calculator::Division(float x, float y)               //定义实现除法运算的成员函数
    {
        float Shang = 0;
        if (y == 0.0)
            return -1;
        Shang = x / y;
        return Shang;
    }
    float Calculator::CPow(float x, float y)                  //定义实现乘方运算的成员函数
    {
        float ChengFang = 0;
        ChengFang = pow(x, y);
        return ChengFang;
    }
    float Calculator::CSqrt(float x)                          //定义实现开平方运算的成员函数
    {
        float sqrtC = 0.0;
        sqrtC = sqrt(x);
        return sqrtC;
    }
    void main()                                               //程序入口,main()函数
    {
        float x = 0;
        float y = 0;
        cout << "请输入 x:" ;
        cin >> x;                                             //输入数据 x
        cout << endl;
        cout << "请输入 y:" ;
        cin >> y;                                             //输入数据 y
        cout << endl;
        Calculator my;                                        //定义类的对象
        float He = my.Adder(x, y);
        cout << "两数之和:" << He << endl;
        float Cha = my.Substraction(x, y);
        cout << "两数之差:" << Cha << endl;
        float Ji = my.Multiplication(x, y);
        cout << "两数之积:" << Ji << endl;
        float Shang = my.Division(x, y);
        cout << "两数之商:" << Shang << endl;
        float Pow = my.CPow(x, y);
        cout << "x 的 y 次方:" << Pow << endl;
    }
```

```
float KaiFang=my.CSqrt(x);  
cout<<"x的平方根:"<<KaiFang<<endl;  
cout<<"任意键程序退出";  
getchar();  
}
```

程序执行后的效果如图 1-1 所示。

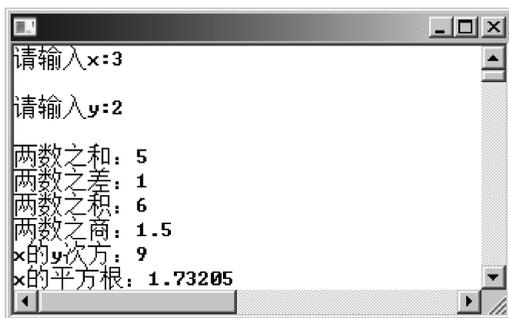


图 1-1 例 1-3 执行效果图

总结 在例 1-3 中，通过声明、定义一个类 Calculator，说明了如何使用类来进行面向对象的编程。通过定义类，形成一个具有一定功能的通用类型或通用模板；使用这个类的对象时，可以调用模板中的所有成员函数。面向对象技术要高于面向过程技术。面向对象的程序开发是给每个对象具备了一定的特殊“能力”，而这些“能力”一般也是通过面向过程的技术实现的。通过本小节的学习，主要理解“面向对象”的概念就可以了。

1.1.4 C++ 中的声明和定义

在程序开发领域，声明是指当一个计算机程序需要调用内存空间时，对内存发出的“占位”指令。定义则是指将声明的变量、函数、类等呈现或描述出来，为之提供一个意义相当的表达，并表明其与众不同之处，最终展现其内涵。

在软件开发过程中，常见的声明和定义语句列举如下（注意中文注释语句）。

例如：

```
int a; //声明一个整型变量 a  
float b; //声明一个浮点型变量 b  
struct S //声明一个结构 s  
{  
    int a;  
    float b;  
};  
extern int x; //声明一个外部变量 x  
void count(bool YN, int& counter); //声明一个函数 count
```

上述几行语句均是典型的声明语句。

```
void count (bool YN, int& counter )      //函数 count 的定义
{
    if (YN)                             //如果 YN 是 true
    {
        counter ++;                     //计数器加 1
    }
}
```

上述几行代码对函数 `count (bool YN, int& counter)` 进行定义，用以实现函数的功能。下面通过例 1-4 来验证上述函数的定义。

例 1-4

```
#include <iostream>
using namespace std;
void count (bool YN, int& counter);      //函数声明
void count (bool YN, int& counter )     //函数定义
{
    if (YN)
    {
        counter ++;
    }
}
void main ()                            //main() 主函数
{
    int counter = 0;                    //计数器
    bool YN = 1;                       //逻辑变量,用于当条件满足时,退出 while 循环
    while (YN)
    {
        count (YN, counter);           //计算循环次数
        cout << counter << endl;       //输出次数至屏幕
        if (counter > 5)                //判断循环次数是否大于 5 次
        {
            YN = 0;                    //如果循环次数超过 5 次,将逻辑变量 YN 置 0
        }
    }
}
```

例 1-4 的执行效果如图 1-2 所示。

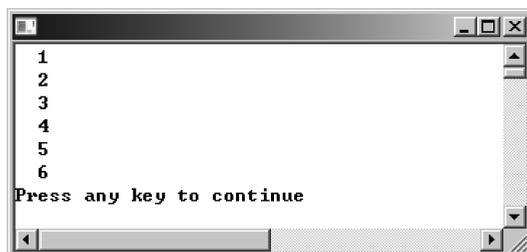


图 1-2 例 1-4 的执行效果

通过例 1-4 可知, 在 C++ 中, 声明和定义仍然是程序开发所必需的, 这沿袭了 C 语言的特点。一个名字 (标识符) 在程序中使用之前必须首先声明, 即在一个标识符使用之前, 必须表述清楚其类型, 便于编译器识别。在 C++ 程序中, 每个标识符必须有一个定义。通俗来讲, 定义就是给标识符确定了一个值。对于常量的标识符, 其值是不变的; 而对于非常量的标识符, 其值是可以被修改的。有时, 声明和定义可以同时进行, 见下面两行语句。

```
double pi = 3.1415026;  
char * ch = "Here. ";
```

下面简单介绍 typedef 的使用。typedef 是为每个类型声明了一个新的名字 (标识符), 而不是一个类型的对象。例如:

```
typedef int INT_32;  
typedef short INT_16;  
typedef unsigned char UCHAR;
```



总结 本小节使用通俗易懂的语句和范例, 简明扼要地讲述了 C++ 语言中的声明和定义。这部分内容是程序开发最基础的部分, 希望读者能反复阅读, 彻底理解这一部分内容。

1.1.5 最简单的 C++ 程序

每个 C++ 程序必须有一个 main() 函数。该函数中的每条语句分别实现各自的功能, 序列组成 main() 函数的主体。例如:

```
#include <iostream>  
using namespace std;  
void main()  
{  
    cin.get();  
    return;  
}
```

若上述代码被执行, 我们只需按下键盘的任意键, 即可使之退出运行。下面从多个角度尽可能地阐释各方面知识, 并再举一例, 以说明如何使用 C++ STL 创建一个最简单的 C++ 程序。



提示 在例 1-5 中, 首先包含了两个头文件 <iostream> 和 <list>, 使用命名空间 std。头文件 <iostream> 的使用, 使得程序可以使用 C++ STL 中的流输入和流输出。在本例题中, 具体来说, 就是可以使用 cin 和 cout 实现屏幕输入和屏幕输出。头文件 <list> 使得程序可以使用 C++ STL 中的容器 list。在访问容器中的元素时, 使用了 list 模板的迭代器。

读者应尽量读懂这段程序代码, 如果有些内容实在无法理解 (例如迭代器), 应该先记住这些内容, 随着本书章节的延伸, 逐渐明白其中的意义。

例 1-5

```
#include <iostream>  
#include <list>  
using namespace std;  
struct PERSON{ //定义结构 PERSON  
    int id, sex;
```

```
double core;
void clear()
{
    id=0;
    sex=0;
    core=0;
}
};
void main()
{
    PERSON temp;
    list<PERSON> C1;           //声明一个 list 容器的对象 C1,容器中元素是 PERSON 类型的对象
    int id_temp, sex_temp, size;
    double core_temp;
    C1.clear();              //清空容器
    int counter=0;
    cout<<" This is a simplest C++ Example! \n " <<endl;
    cout<<" 任意键开始..... ";
    cin.get();               //按<Enter>键等待
    while (counter<5)       //输入数据
    {
        cout<<" 请输入 ID:";
        cin>>id_temp;
        cout<<" 请输入性别:";
        cin>>sex_temp;
        cout<<" 请输入分数:";
        cin>>core_temp;
        temp.id=id_temp;
        temp.sex=sex_temp;
        temp.core=core_temp;
        C1.push_back (temp); //将结构 PERSON 类型的变量 temp 装入容器 C1
        memset (&temp, 0, sizeof (PERSON)); //将变量 temp 清零
        counter++;           //计数器加 1
    }
    cout<<" 按<Enter>键继续..... ";
    cin.get();              //输入等待
    size=C1.size();
    cout<<endl;
    list<PERSON>::iterator Iter; //声明 list 容器的迭代器类型变量 Iter
    for (Iter=C1.begin(); Iter!=C1.end(); Iter++) //遍历输出容器中的每个元素
    {
        temp.clear();
        temp=* Iter;
        cout<<" ID: " <<temp.id<<" , SEX: " <<temp.sex<<" , Core: " <<temp.core<<endl;
    }
}
```

```
cout << "任意键退出程序..... "; // << endl;  
cin.get();  
return;  
}
```

在例 1-5 中, 读者应该重点理解程序的总体架构和模板的使用, 不要执着于个别生僻的知识点。在今后的学习中, 随着编程知识的逐渐积累, 逐渐做到信手拈来的程度。再者, 读者应该重点理解 C++ 程序中 list 容器的使用方法; 如何使用 list 模板, 如何使用模板的迭代器, 如何遍历和访问容器中的元素等知识。例 1-5 的执行效果如图 1-3 所示。

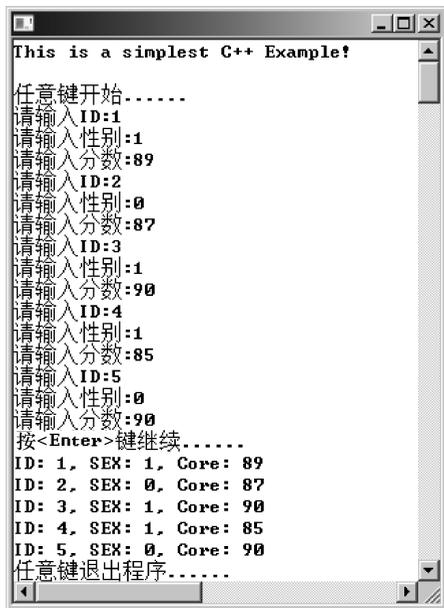


图 1-3 例 1-5 的执行效果

1.1.6 指 针

指针是 C/C++ 语言中一个重要的概念, 同时也是 C/C++ 语言的重要特色。正确而灵活地使用指针, 可以有效地表示复杂的数据结构、动态分配内存、便捷地使用字符串、方便地使用数组 (甚至超大数组)、直接处理内存、函数可以返回多个数值等。

在内存中, 变量存储在特定的地址中, 通过地址可以找到所需的变量。该变量在内存中的地址“指向”该变量单元。一个变量的地址称为该变量的“指针”。若一个变量被用来存储另一变量的地址, 则它称为“指针变量”。

在学习过程中, 读者要注意区分变量的指针和指针变量。变量的指针是指该变量在内存中的存储地址; 而指针变量是指该变量的值是一个指向其他变量的指针值。在学习指针的过程中, 读者首先要学习如下两个运算符:

&——取地址运算符;

*——指针运算符。

下面举个简单的示例, 以帮助读者理解什么是指针、什么是指针变量。

例 1-6

```
#include <iostream>
using namespace std;
void main()
{
    long Aa = 11; //给变量 Aa 赋值
    long *B_Pointer = NULL; //声明指针变量 B_Pointer, 并将其赋值为空
    B_Pointer = &Aa; //将变量 Aa 的内存地址保存在 B_Pointer 中
    cout << "Aa 的值: " << Aa << " ,Aa 的内存地址: " << B_Pointer << endl;; //输出
    cin.get(); //等待程序退出
}
```

下面继续指针的学习。

在 C++ 语言中, 指针和数组基本等价的原因在于指针算术和 C++ 内部处理数组的方式。指针变量增加 1 或减去 1 之后, 增加或减少的量等于指针指向的类型的字节数, 例如, 指向 double 型变量的指针加 1 之后, 指针数值将增加 8。

而 C++ 将数组名解释为数组第 1 个元素的地址, 例如, 数组 wages [20] 存储 20 个 double 型数据, 即

```
double *pw = wages;
```

其实 wages = &wages [0]。现在指针 pw 指向了 wages [0], 而 * (pw + 1) 将显示 wages [1] 的数值, 说明 (pw + 1) 指向了数组的第 2 个元素。

指针既可以动态创建, 也可以静态创建。静态创建指的是在声明时, 即给定数组的长度。例如,

```
Int array[10];
```

动态创建指针需要使用 new 操作符, 即数组的长度可以是未知的, 在程序运行时为数组分配内存空间。这类指针或数组名必须使用 delete [] 指令释放其占用的内存。例如,

```
cin >> size;
int* pi = new int[size];
...
delete [] pi;
```

指针和数组的关系还可以扩展到字符串。



提示 1 动态分配内存会导致内存泄露。如果使用 new (在自由存储空间或堆中) 操作符创建变量之后, 没有使用 delete 将其释放, 会导致内存泄露。即使包含指针的内存由于作用域规则和对象生命周期的原因而被释放, 在自由存储空间上动态分配的变量或结构将继续存在。但实际上将会无法访问自由存储空间中的结构, 因为指向该内存的指针无效。这部分内存存在程序的整个生命周期都不可使用, 即这些内存被分配出去, 而无法收回, 造成内存浪费。一旦内存泄露严重, 会导致程序的可用内存被耗尽, 导致程序崩溃。

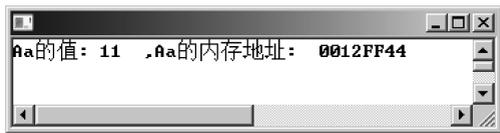


图 1-4 例 1-6 的执行效果

提示
2

指针的目的是希望能直接映射机器上的地址机制，即实现可以对字节寻址，甚至可以从机器字中取出字节。

1.1.7 函 数

1. 函数的概念

函数的英文单词是 `function` (起……作用)。单词 `function` 既可以是动词，也可以是名词。作为名词时，其意义为：功能，作用，应变量，函数，职务，重大聚会；作为动词时，其意义为：有作用，起作用，行使职责。显然，在程序设计过程中，函数既有名词的意义，也有动词的意义。通常在程序中用函数来表示子例程。

C/C++ 的源程序是由函数组成的。读者在学习 C 语言阶段，对函数已有了一定程度的理解。本节复习函数的概念，以帮助读者打下学习模板库的基础。

C/C++ 程序中至少需要（必须）一个 `main()` 函数。程序通常由多个函数组成的。函数是 C++ 源程序的基本模块。通过调用相应的函数可实现特定的功能。

从函数定义的角度划分，函数可分为库函数和用户自定义函数两种。C/C++ 语言提供了强大的、极为丰富的库函数。库函数使用时只需包含相应的头文件即可直接调用。用户自定义函数是按用户需求编写的函数。用户或程序开发者也可以把自己的思想或算法编写成一个个相对独立的函数模块。在使用时，不但要在程序中定义函数本身，而且要在 `main()` 函数模块中进行函数类型说明。

函数还可以分为有返回值函数和无返回值函数。有返回值函数被调用后将向调用者返回一个执行结果（函数返回值）。有返回值的函数必须在函数声明和定义中明确函数返回值的类型。无返回值函数用于完成某项特定的处理任务，执行完成后不必向调用者返回函数值。在定义此类型函数时，函数返回值的类型为 `void`。

函数还可以具有参数。从这个角度划分，函数可以分为无参函数和有参函数。无参函数是指函数定义、函数声明及函数调用过程中均不包含参数；有参函数是指在函数定义及函数声明时均包含参数。有参函数的参数通常称为形式参数。函数调用时必须给出参数（实际参数）。函数调用时，主函数会把实际参数的值传送给形式参数，供被调用的函数使用。

对于函数的声明和定义，举例说明如下：

```
#define pi =3.1415926
double area(float Radius)           //函数声明,函数为有返回值函数,并且拥有一个参数
double area(float Radius)           //函数定义,函数返回值为 double 类型
{
    double A=0;
    A=pi*Radius* Radius;
    return A;
}
```

2. 函数的调用

函数是如何被调用的呢？

C 语言中，函数调用的一般形式为：

函数名(实际参数表)

参数表中的参数可以是常数、变量、其他类型的数据以及表达式。各参数之间用逗号分隔。若函数没有参数，则调用时不用填写参数表。形式如下：

函数名 ()

说明：函数定义时填写的参数表是形式参数，简称形参；函数被调用时，填入的参数是实际参数，简称实参。

函数被调用时，将实际参数传递给形参表可以有两种形式：①赋值调用；②引用调用。赋值调用是将各个参数的值传递给函数的形参表。此时函数中形参的值发生变化，不会影响函数调用时使用的变量（实际参数）。引用调用是将实际参数的地址复制给形式参数。当函数调用时，这个地址用来访问所使用的实际参数。这意味着在函数调用过程中，形参的数值变化会直接反映到实际参数的值。下面举例说明这两种调用形式的区别。



提示 例 1-7 中包含两个自定义函数 `pow2 (float x)` 和 `pow3 (float * y)`，函数 `pow2()` 是赋值调用形式；函数 `pow3()` 是引用调用形式，但在函数 `pow3()` 中参数 `y` 并没有发生变化，因此在输出变量 `B` 时，`B` 的值没有变化。

例 1-7

```
#include <iostream>
#include <cmath>
using namespace std;
double pow2(float x);           //函数 pow2 () 的声明
double pow3(float y);          //函数 pow3 () 的声明
double pow2(float x)           //函数 pow2 () 的定义
{
    double z=0;
    z = x*x;
    return z;
}
double pow3(float* y)          //函数 pow3 () 的定义
{
    double z=0;
    z = (*y)* (*y)* (*y);
    return z;
}
void main()
{
    double A =10;
    float B =20;
    double C =0;
    double D =0;
    C =pow2 (A);
    cout << "A:  " << A << "  ; Pow2 (A):  " << C << endl;
    D =pow3 (&B);
    cout << "B:  " << B << "  ; Pow3 (B):  " << D << endl;
    cin.get ();
}
```

上述代码的执行效果如图 1-5 所示。

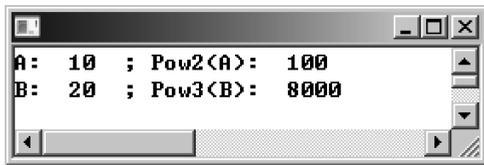


图 1-5 例 1-7 的执行效果



总结 本小节主要学习如何使用函数，如何调用函数，理解函数的形参和实参，理解被调用函数的返回值。

3. 函数的递归调用形式

函数可以实现自我调用。若在函数内部调用了函数自身，则称这个函数为“递归”。例如，计算阶乘的函数代码即是使用了递归形式。3 的阶乘等于 $1 \times 2 \times 3$ ，即 6。一般计算阶乘的代码如例 1-8 所示：

例 1-8

```
#include <iostream>
using namespace std;
int Factor(int n)
{
    int answer=0;
    if(n ==1) //递归结束条件
        return (1);
    answer =Factor(n -1) * n; //函数的递归调用
    return answer;
}
void main()
{
    int n=5;
    int result=0;
    result =Factor(n); //求整数 5 的阶乘
    cout << "5! = 5x4x3x2x1 = " << result << " " << endl; //输出计算结果
    cin.get(); //等待程序退出
}
```

例 1-8 的执行效果如图 1-6 所示。

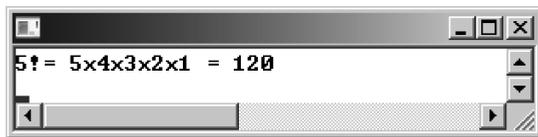


图 1-6 例 1-8 的执行效果



总结 在实现函数的递归调用形式时，最关键的是将“递归结束条件”放在递归调用的前面，并且“递归结束条件”一定要书写正确，否则会带来不必要的麻烦。

4. 函数的通用性和效率

函数的通用性一般用来指某些通用性较强的函数，通用函数可被许多程序员使用，并且通用函数需要的变量和数据必须用函数参数的形式传递。使用参数传递数据，除有助于函数可用在多种情况下外，还可提高函数代码的可读性。

C 语言是基于函数的语言，故函数是必不可少的。但函数调用的效率较低，对于特别简单的功能模块，或程序对执行速度要求较高时，可考虑使用内嵌代码的形式。如例 1-9 所示：

例 1-9

```
#include <iostream>
using namespace std;
void main()
{
    int x;
    for(x=1;x<11;++x)
        printf("%d ",x*x);
    printf("\n");
    cin.get();
}
```

例 1-9 中，代码语句 `for (x = 1; x < 11; ++x) printf (“%d ”, x * x)` 可以单独编写成一个函数，但是函数调用的效率较低，此处体现了内嵌代码的优越性。

5. 函数中的变量

1) 局部变量和全局变量

局部变量也称为内部变量，它是在函数内实现定义说明的。其作用域仅限于函数内，离开该函数后再使用这种变量是非法的。

全局变量也称为外部变量，它是在函数外部定义的变量。全局变量不属于哪一个函数，而属于一个源程序文件。其作用域是整个源程序。在函数中使用全局变量，一般应进行全局变量说明。全局变量的说明符号为 `extern`。但在某函数之前定义的全局变量，在该函数内使用可不再加以说明。

2) 变量的存储类别

使用函数会涉及不同的存储方式：静态存储方式；动态存储方式。全局变量使用静态存储方式；函数的形参和普通局部变量使用动态存储方式。

在函数中还可以声明或定义 4 种类型的变量。这 4 种类型分别是 `auto`，`static`，`register` 和 `extern`。

- `auto` 类型是默认类型，可以省略不写。
- `static` 类型属于静态存储类型，当希望函数中局部变量的值在函数调用结束后不消失并保留原值时，此时可用关键字 `static` 声明该局部变量。
- C/C++ 语言允许将局部变量的值放在 CPU 的寄存器中，这种变量叫做“寄存器变量”，用关键字 `register` 进行声明。
- 如果外部变量不在文件的开头定义，其有效的作用范围只限于定义处到文件末尾。

若函数需要引用该外部变量，则应该在引用之前用关键字 `extern` 对该变量进行“外部变量声明”，表示该变量是一个已经定义的外部变量。有此声明，即可从“声明”处起合法地使用该外部变量。

下面举例说明 4 种变量的存储类别。

首先定义一个文件 `my.cpp`，在文件中添加以下代码：

```
#pragma once
int A=1;
int B=2;
```

创建例 1-10 的源文件如下：

例 1-10

```
#include <iostream>
#include <cstring>
#include <string>
using namespace std;
extern int A; //外部变量
extern int B; //外部变量
void main()
{
    auto int x=0; //auto 类型整数变量
    register int y=0; //register 类型整数变量
    static int count=0; //静态存储类型整型变量
    char * ch=new char();
    sprintf(ch,"%d,%d,%d,%d,%d ",x,y,count,A,B);
    cout << ch << endl; //输出
}
```

将 `my.cpp` 文件和 `ex10.cpp` 放在一起，或者将 `my.cpp` 文件加入到 `ex10.cpp` 所属工程中。例 1-10 中包含了 4 种变量的存储类型，虽然没有给出各种变量的详细说明，但最重要的是给出了外部变量的使用方法。例 1-10 的执行效果如图 1-7 所示。



图 1-7 例 1-10 的执行效果



总结 在阅读此小节时，读者要认真体会函数中各种变量的使用方法。

1.1.8 文 件

C/C++ 中的文件是非常重要的概念。所谓“文件”，一般是指存储在外部介质上数据的

集合，并且是一组相关数据的有序集合。当被使用时，文件被调入内存中。

文件可分为两种：一种是上述所说的，就是驻留在磁盘或其他外部介质上的一个有序数据集，可以是数据文件、可执行程序等；另一种是设备文件，是指和主机相连的各种设备，如显示器、键盘、打印机等。外部设备一般可看作一个文件来进行管理。它们的输入、输出等同于对磁盘文件的读和写。

在 C 语言中，文件的类型可分为两种：缓冲型和非缓冲型。文件系统的读写也因此分为两种方法：缓冲文件系统一般用来处理文本文件；非缓冲文件系统可用来处理二进制文件。C 语言函数库中包含了大量的文件处理函数。当需要使用这批函数时，需要在源程序中包含头文件“stdio.h”。

在 C++ 语言中，关于文件的处理功能更加高级——新封装了部分模板和分类。常用的类如下：

- ifstream 和 wifstream 用来读取文件；其中 wifstream 是字读取操作。
- ofstream 和 wofstream 用来将数据写入文件；其中 wofstream 是字读取操作。
- fstream 和 wfstream 用于读写文件；其中 wfstream 是字读取操作。
- filebuf 和 wfilebuf 用于进行实际的字符读写；其中 wfilebuf 是字读写操作类。

值得注意的是，使用上述几个类时，需要包含头文件 <fstream>。下面介绍在 C++ STL (模板库) 中如何使用文件操作。

1. 文件标识

在 C 语言中打开文件时，一般需要使用标识符来表明文件的使用方式。同样，在 C++ 语言中打开文件时，同样需要使用文件标识符（文件模式常量）来说明文件的使用方式。常用的文件模式常量如下：

ios_base::in	打开文件，用于读取
ios_base::out	打开文件，用于改写或写入
ios_base::ate	打开文件，用于将文件指针移到文件末尾
ios_base::app	打开文件，写入时始终添加入尾端
ios_base::trunc	打开文件，将先前的内容移除
ios_base::binary	以二进制形式打开文件

2. 文件的读写

本小节主要讲述如何使用 ifstream 类和 ofstream 类。这两个类均包含了一批各自的成员函数。这些成员函数主要用于打开和关闭文件、设置文件缓冲区、读取和写入文件、文件指针定位等。下面举例说明文件读写操作。例 1-11 的功能是将源文件中的内容读取出来，之后将这些内容再写入目标文件。

例 1-11

```
#include <iostream>
#include <fstream>
using namespace std;
void main()
{
    ifstream f1;
    ofstream f2;
```

```

char filename1[256];
char filename2[256];
char content[256];
cout << "请输入文件名(源):"; //输入源文件名
cin >> filename1;
cout << "请输入文件名(目的):"; //输入目标文件名
cin >> filename2;
f1.open(filename1,ios::in); //打开源文件,用于读取
f2.open(filename2,ios::out); //打开目标文件,用于写入
while(! f1.eof())
{
    f1.getline(content,128); //读取源文件中的数据
    f2 << content << endl; //将数据写入目标文件
}
f1.close();
f2.close();
}

```

3. 文件流状态检查

C++ 文件流类从 `ios_base` 类那里继承了一个流状态成员。这个流状态成员指出诸如一切顺利、已达文件尾、I/O 操作失败等流状态信息。例如, `fail()`; `is_open()`; `eof()`。

4. 使用临时文件

在软件开发过程中,程序员经常需要使用临时文件。临时文件的存在是短暂的,且必须受程序控制。在C++中,创建临时文件、复制另一个文件的内容并删除文件其实都很简单。首先,需要为临时文件制订一个命名方案,确保每个文件都被指定独一无二的文件名。`cstdio` 库函数中声明的 `tmpnam()` 标准函数可以满足这一要求。函数原型如下:

```
char * tmpnam ( char * pszname )
```

`tmpnam()` 函数创建一个临时文件名 (值得注意: 仅仅产生一个文件名), 将它放在 `pszname` 指向的 C - 风格字符串中。常量 `L_tmpnam` 和 `TMP_MAX` 限制了文件名包含的字符数以及在确保当前目录中不生成重复文件名的情况下, `tmpnam()` 可被调用的最多次数。临时文件的使用方法见例 1-12。



提示 在例 1-12 的代码中, 头文件 `cstdio` 其实是原 C 语言中的头文件 `stdio. h`。在本例中, 变量 `TMP_MAX` 的值为 32767, 表明临时文件名字符长度的变量 `L_tmpnam` 的值为 14。文件名是随机产生的, 所有临时文件输出到屏幕上。

例 1-12

```

#include <iostream>
#include <cstdio>
void main()
{
    using namespace std;
    cout << "This system can generate " << TMP_MAX << " temporary names of up to " << L_tmpnam << "
characters. \n";
}

```

```
char pszName[L_tmpnam] = {'\0'};
cout << "Here are ten names: \n";
for(int i=0;i<10;i++)
{
    tmpnam(pszName); //生成临时文件名
    cout << pszName << endl;
}
return;
}
```



总结 读者若能在 C 语言中熟练进行文件操作，再学本小节是比较轻松的。唯一的不同之处在于：在 C++ 中使用了模板库和常用的 IO 类。读者可以执行附例中的可执行程序，或者重新编译源文件之后执行，体验一下产生临时文件名的过程。

1.1.9 编译和链接

程序员使用任一编辑软件将编写好的 C++ 程序输入计算机，并以文本文件形式保存在计算机的硬盘上。编辑的结果是建立 C++ 源程序文件。C++ 程序一般使用小写英文字母，常量和用途的符号可用大写字母。C++ 编译器对大写字母和小写字母是有区别的。C++ 语言的关键字必须是小写。

1. 程序编译

程序编译是指将编辑好的源文件翻译成二进制目标代码的过程。编译过程是使用 C 语言提供的编译程序完成的。在不同操作系统下，各种编译器的使用命令不完全相同，使用时应注意计算机环境。在编译过程中，编译器首先要检查源程序中的每一条语句是否存在语法错误，一旦发现错误，会提示错误的位置和错误类型信息。程序员需要再次调用编辑器进行查错修改，之后再编译，直至排除所有语法和语义错误。正确的源程序文件经过编译后在硬盘上生成目标文件。

C/C++ 程序需要经过多个步骤才能生成。除了需要定义问题需求、设计并在计算机上输入代码外，必须通过计算机生成一个可执行文件。源代码包含了指挥计算机运行的 C/C++ 语句。

编译器是一个计算机程序，读入代码后，如果源代码“语法正确”，将把该程序生成机器码。用户将一个源文件提交给编译器后，首先进行的是该文件的预处理，即完成宏处理，并按照 #include 指令引进所有头文件。预处理之后的结果被称为被编译单位。这种编译单位才是编译器真正的工作对象，即 C++ 语言规则所描述的对象。

2. 程序的链接

程序经过编译产生的目标文件是可重定位的程序模块，不能直接运行。

链接程序把目标文件和其他分别进行编译生成的目标程序模块及系统提供的标准库函数链接在一起，生成可以运行的可执行文件。

链接过程使用 C++ 语言提供的链接程序完成，链接器把机器码和函数库代码连接起来，生成一个可执行文件。生成的可执行文件存放在计算机硬盘中。

若程序运行在个人计算机上，则必须在个人计算机上进行编译和链接。但符合 ISO C++

标准的源代码可以在不同机器上编译和链接, 这种情况称为代码可移植。标准化的语言应该是可移植语言。

在所有编译单位中, 对所有函数、类、模板、变量、命名空间、枚举和枚举符的名称使用都必须保持一致, 除非被显式地描述为局部的东西。所有名称空间、类、函数等都应该在它们出现的各编译单位中有适当的声明, 且声明都应该一致地引用同一个实体。

如果一个名字可以在与其定义所在的编译单位不同的地方使用, 即被称为具有外部链接; 如果某个名字只能在其定义所在的编译单位内部使用, 即被称为具有内部链接。

1.1.10 程序启动和终止

对于任何 C/C++ 程序, 程序执行时总是从 `main()` 函数开始的。执行函数 `main()` 之后, 依次或者顺序执行 `main()` 函数中的程序代码, 实现其中各个函数的功能。

这使人联想到在生活和工作中, 工作者总是将“大功能”分解成“小功能”, 以便于实现。在 C/C++ 中, 主函数 `main()` 就是所谓的“大功能”。例如, 一个做菜程序, 总的过程就是 `main()` 函数, 在主函数中根据情况, 调用“买菜”“切菜”“烧油”“放葱花”等一系列步骤 (子函数)。一般在可执行程序中, 必须以主函数 `main()` 作为程序运行的入口。其余函数均由 `main()` 或其他一般函数调用。

一个 C/C++ 的源程序经过编译、链接之后, 会生成扩展名为 `exe` (或 `com`) 的可执行文件。扩展名为 `exe` 的程序可以在操作系统下直接运行, 即可以由系统启动。`main()` 函数则可以在程序运行时传递参数。

对于非 Windows 程序, 程序的终止一般通过调用 C++ 函数库的 `return()` 函数来实现。`return()` 函数既可以返回一个数值, 也可以不返回数值。程序的终止还可以使用 C 函数库中的 `exit()` 函数、`abort()` 函数等。

调用 `exit()` 函数时, 程序正常退出, 退出之前程序会完成一切需要完成的工作, 例如释放分配的内存单元等。`abort()` 函数是“放弃”的意思, 执行该函数时, 程序不做“善后”处理, 直接退出运行。

C/C++ 还提供其他进程控制函数, 例如 `terminate()`、`system()` 等。一般情况下, 若程序处理异常, 则一般调用 `terminate()` 函数; 若在程序中需要调用其他命令或者可执行文件, 则可调用 `system()` 函数。

1.1.11 异常处理

后续章节会详细地介绍异常处理。本小节只简单介绍异常处理的概念。

程序运行时会遇到某种错误, 导致程序无法正常运行。C++ 的异常类为处理这种情况提供一种功能强大而灵活的工具。异常处理机制是 C++ 的新功能。异常处理机制可以看作编译时的类型检查和歧义性控制机制在运行中的对应物。错误处理是一件非常困难的工作。而 C++ 的异常处理机制为程序员提供了一种处理错误的方式。在给定的系统结构中, 以最自然的方式去处理这些错误。异常机制使处理错误的复杂性更加清晰可见。

当 C++ 程序中出现异常时, 检测到异常的程序段可以通过产生或抛出异常, 使主进程知晓“异常已经发生”。在 C++ 中, 用于进行异常处理的是 `catch` 子句。当异常被 `try` 块中的语句抛出时, 系统通过查看 `try` 块后的 `catch` 子句列表, 来查找能够处理该异常的 `catch` 子句。

C++ STL 中定义了标准库异常类 `exception`，类的声明包含在头文件 `<exception>` 中，但是异常类 `exception` 不能捕获所有异常，它仅仅为人们提供了一种更为便捷的异常处理方法。

1.1.12 预处理命令

预处理语句是由一系列和预处理相关的命令符组成的。预处理语句以“#”作为起始标志，其后紧跟预处理命令关键字，之后是空格，空格之后是预处理命令的内容。C 语言提供了多种预处理功能，如宏定义、文件包含、条件编译等。合理使用预处理功能编写的程序模块有助于阅读、修改、移植和调试，且有助于模块化程序设计。C 语言预处理程序的作用是根据源代码中的预处理指令修改源代码。预处理指令是一种命令语句（如 `#define`），用以指示预处理程序如何修改源代码。在对程序进行通常的编译处理之前，编译程序会自动运行预处理程序，对程序进行编译预处理，这部分工作对程序员来说是不可见的。

宏是预处理命令的一种形式，之所以单独介绍它，是因为其使用非常普遍。宏的完整名称是“宏替换”，在 ISO/IEC9899 国际标准“Programming Language—C”中和 ISO/IEC14882 国际标准“Programming Language—C++”中，均对宏进行了详细的描述。

1. “#”和“##”

(1) “#”符号的使用

“#”是预处理命令的标志符号。示例如下：

```
#define PI 3.1415926
```

“#”符号还有一个功能，就是将跟在其后的参数转化成一个字符串。例如，

```
#define PASTE(n) "adhfkj"#n
main()
{
    printf("%s", PASTE(15));
}
```

上述程序段代码的执行结果为：`adhfkj15`。

请读者体会宏定义语句中包含的“#n”的含义。

(2) “##”符号的使用

“##”也是预处理命令的标志符号，但是不允许出现在宏定义语句的开始和末尾。“##”字符串可以将两个独立的字符串连接成一个字符串。例如，

```
#include <stdio.h>
#define NUM(a,b,c) a##b##c
#define STR(a,b,c) a##b##c
main()
{
    printf("%d ",NUM(1,2,3));
    printf("%s ",STR("aa","bb","cc"));
}
```

上述程序段代码的执行结果为：

aabbcc



提示

“#”就是特殊语句（预处理语句）的标志符号；“##”是字符连接标识。

2. 常见预处理命令

(1) 定义变量和取消定义变量

预处理程序段是通过 `define` 和 `undef` 命令实现定义变量和取消定义变量的。`define` 其实是宏定义命令，应该放在后面的 1.13 节讲解，但是鉴于它在预处理语句中的重要性，这里先对其进行简单介绍。

取消定义变量（取消宏定义）的命令是 `undef`。其作用是取消该命令前面的程序段中使用 `define` 定义的宏变量。在例 1-13 的 Example 1 中，使用 `#define` 命令定义宏变量 `PI` 为 3.14159，程序最后使用 `undef` 取消宏变量 `PI` 的定义。在 Example 2 中，在 `main()` 函数中，先定义宏变量 `MAX` 等于 200，并将其输出到屏幕上，之后取消前面的定义；重新定义宏变量 `MAX` 等于 150，重新输出宏变量 `MAX`。Example 2 的执行结果如图 1-8 所示。

例 1-13

Example 1.

```
#define PI 3.14159
main()
{
...
}
#undef PI
```

Example 2.

```
#include "stdio.h"
int main( void )
{ #define MAX 200
printf("MAX = %d\n",MAX);
#undef MAX
#define MAX 150
printf("MAX = %d\n",MAX);
getchar(); //这一行代码的作用:上述程序执行完以后,待用户按任意键,程序才退出运行。
return 0;
}
```



图 1-8 例 1-13 的程序执行结果

(2) 条件预处理语句

一般情况下，程序的每一行源代码都是要编译的。特殊情况下，只有在满足一定条件时

才编译某部分内容。条件不同，编译的程序部分不同，即产生不同的功能，即条件编译。常用条件编译的关键字主要有 #if、#ifdef、#ifndef、#else、#elif 和 #endif。这些关键字的常见组合一般如下：

第一种形式：

```
#if 表达式 ( expression )
    程序段 1 ( program code1 )
#else
    程序段 2 ( program code2 )
#endif
```

以上形式的功能是当表达式的值为“真”（非零）时，编译程序段 1；否则，编译程序段 2。

例 1-14

```
#define MAX 10
main ()
{
    #if MAX > 99
        printf("1234\n");
    #else
        printf("abcd\n");
    #endif
}
```

第二种形式：

```
#ifdef 标识符 ( Identifier )
    程序段 1
#else
    程序段 2
#endif
```

以上形式的功能是：当所指定的标识符已经被 #define 定义时，仅编译程序段 1；否则编译程序段 2。

例 1-15

```
#define DEBUG
main()
{
    ...
    #ifdef DEBUG
        printf("x=%d, y=%d\n", x, y);
    #endif
    ...
}
```

第三种形式：

```
#ifndef 标识符 (Identifier)
```

```
    程序段 1
```

```
#else
```

```
    程序段 2
```

```
#endif
```

以上形式的功能和第二种形式相反，即表示当标识符没有被#define 语句定义时，仅编译程序段 1；否则，仅编译程序段 2。

例 1-16

```
#ifndef _FILENAME_H
    #define _FILENAME_H
#endif
#if _WINDOWS_
    #define OS_VER    " WINDOWS"
#else
    #define OS_VER    " UNIX"
#endif
```

(3) 包含头文件命令

关键字 include 是最常见的预处理命令之一。其功能是将被包含的文件中的源代码“放进”源文件中，从而实现代码重用，避免编程者大量的工作。最常见的形式为：

```
#include“文件名.h”
```

例如，

```
#include“stdio.h”    //在默认的系统目录、安装目录、当前目录中寻找头文件
```

或

```
#include <stdio.h>    //尽在当前目录中寻找头文件 stdio.h
```

上述语句的意思是将标准库文件 stdio.h 包含进源文件中。stdio.h 文件内主要是声明了一些和标准输入输出相关的变量和函数。

在使用文件包含命令时，主要注意避免的是重复包含同一个头文件。为实现此目的，程序员可以利用条件预处理语句和 include 语句相结合的方法。例如，

```
#include VERSION == 1
    #define INCFILE“vers1.h”
#elif VERSION == 2
    #define INCFILE“vers2.h”
#else
    #define INCFILE“versN.h”
#endif
#include INCFILE
```

上述代码的功能是：若 VERSION 等于 1，则定义宏变量 INCFILE 为“vers1.h”；若 VERSION 等于 2，则定义宏变量 INCFILE 为“vers2.h”；若 VERSION 等于其他值，则定义宏变量 INCFILE 为“versN.h”。最后使用包含语句“#include INCFILE”将头文件 INCFILE

包含进源文件中。

(4) #line 预处理命令

“#line”属于 C/C++ 语言的预处理命令。其作用是修改代码行的行号。

在正常情况下，程序源代码的行号是逐次增加的，从开始直到程序结尾。如果程序的源代码总共包含 100 行，在正常情况下程序的行号应该是 1~100，并且是顺序递增的。“#line”预处理命令可以改变程序源代码的行号以及编译源文件的名称。其主要用法有以下两种：

```
#line 8          //改变当前行的编号
#line 8"a.c"     //改变当前行的编号,改变当前编译源文件的名称
```



提示 在 C/C++ 的编译系统中，`_LINE_` 和 `_FILE_` 是预先定义的两个宏变量，代表了被编译源文件的当前行和当前文件名，`#line` 命令就是改变这两个宏的值。

常见的预定义宏还包括 `_LINE_`、`_FILE_`、`_DATE_`、`_TIME_` 和 `_STDC_`。例如，

```
#include "stdio.h"
#line 1 "Y1.c"          //修改宏 _LINE_ 和 _FILE_ 的值
void main()
{
    printf (" ling ID: %d, FILENAME = %s \n", __LINE__, __FILE__);
    #line 3 "Y3.cpp"     //修改宏 _LINE_ 的值为 3; 修改 _FILE_ 的值为 "Y3.cpp"
    printf (" ling ID: %d, FILENAME = %s \n", __LINE__, __FILE__);
    #line 5             //修改宏 _LINE_ 的值为 5;
    printf (" ling ID: %d, FILENAME = %s \n", __LINE__, __FILE__);
    getchar();
}
```

上段程序代码执行结果为：

```
ling ID:1, FILENAME = Y1.c
ling ID: 3, FILENAME = Y3.cpp
ling ID: 5, FILENAME = Y3.cpp
```

(5) 特殊命令预处理（“error”“pragam”和“NULL”）

本小节讲述预处理命令“error”“pragam”和“NULL”。

1) 预处理命令“error”。error 预处理命令可以强制编译程序停止编译，并给出提示信息。如果执行该 `#error` 语句，程序不再向下执行，更不用说链接产生可执行文件了。

例 1-17

```
#include "stdio.h"
#ifdef __DOS
#error DOS OS is required.
#endif
void main()
{
    printf (" The OS is DOS! \n");
    getchar();
}
```

以上程序段的功能说明：如果没有定义宏“__DOS”，#error 语句提示“DOS OS is required.”，即程序编译无法成功。如果定义了宏“__DOS”，程序将顺利编译成功。

2) 预处理命令“#pragma”。#pragma 命令可以指定特殊的指令给编译器。

每个编译程序都可以通过#pragma 命令激活或终止该编译程序所支持的一些编译功能。#pragma 预处理命令的使用方法如下：

```
#pragma parameter
```

参数 parameter 是命令的具体参数，具有多种形式。parameter 常见的使用形式如下：

- message。
- argsused。
- exit/startup。
- inline。
- once。
- warn。
- code_seg 和 data_seg。
- resource。
- saveregs。
- hdrstop/hdrfile。

下面分别详细讲解。

① #pragma message。

说明：#pragma message 命令适用于提示一些有用信息，于程序编译过程中，在输出编译信息窗口的同时输出这些信息。

例 1-18

```
#include "stdio.h"
#pragma message("Hello, The OS is Windows 2000!")
int main(int argc, char* argv[])
{
    printf("The program is started! \n");
    printf("Beijing is the Capital. \n");
    printf("The program will end! \n");
    getchar();
    return 0;
}
```

② #pragma argsused。

说明：#pragma argsused 命令仅允许出现在函数定义之间，且仅影响下一个函数，使警告信息被禁止或无效。如果函数的某一个参数在该函数内没有被使用，编译系统可能会有相应的提示信息。如果使用了#pragma argsused 预处理命令，这种警告信息就不会出现了。

③ #pragma exit 和 #pragma startup。

说明：#pragma startup 命令可以实现设置程序启动之前需要执行的函数；#pragma exit 命令可以实现设置程序退出之前需要执行的函数。

例 1-19

```
#include "stdio.h"
void fun_start()
{
    printf (" fun_start has been done! \n");
}
void fun_end()
{
    printf (" fun_end has been done! \n");
}
#pragma startup fun_start
#pragma exit fun_end
void main()
{
    printf (" The main program has been done! \n");
}
```

上述程序代码的执行结果为：

```
fun_start has been done!
The main program has been done!
fun_end has been done!
```

由此可见：程序执行时，先执行了 `fun_start()` 函数，之后才执行了 `main()` 函数，当程序即将退出时，执行了 `fun_end()` 函数。

④ `#pragma inline`。

说明：在 Turbo C++ 中，使用参数 `inline`，而在 Visual C++ 中是使用参数 `auto_line`、`inline_depth` 和 `inline_recursion`。`#pragma inline` 可以指定一个 C/C++ 函数是否要被内联，并且要指定被内联或者被不内联的函数。被调用的内联函数的代码会直接嵌入调用函数的源代码中。

⑤ `#pragma once`。

说明：使用 `once` 参数可以实现仅编译一次该头文件。一般 `#pragma once` 放在头文件的最开始。

⑥ `#pragma warning`。

说明：参数 `warning`（有的 C 版本为 `warn`）可以实现设定提示信息的显示与否以及如何显示。例如，

a) `#pragma warning (disable: 4507 34)`。

功能：不显示 4507 和 34 号警示信息。

b) `#pragma warning (once: 4385)`。

功能：仅显示一次 4385 号信息。

c) `#pragma warning (error: 164)`。

功能：将 164 号警告信息作为一个错误信息显示。

d) `#pragma warning (push)`。

功能：保存所有警告信息的现有警告状态。

e) `#pragma warning (pop)`。

功能：从栈中弹出最后一个警告信息，在入栈和出栈之间所做的一切改动取消。

⑦ `#pragma code_seg` 和 `#pragma data_seg`。

说明：`code_seg` 用于指定函数要存放的代码段。`data_seg` 用于指定数据存放的代码段。

a) 完整的 `code_seg` 使用方法。其形式如下：

```
#pragma code_seg ( [ [ {push | pop},] [identifier,] [" segment-name" [, " segment-class"]])
```

在 .obj 文件中默认的存放节为 .text 节，如果 `code_seg` 没有带参数使用，函数就存放在 .text 节中。push (可选参数) 将一个记录放到内部编译器的堆栈中，可选参数可以为一个标识符或者节名，pop (可选参数) 将一个记录从堆栈顶端弹出，该记录可以为一个标识符或者节名。当使用 push 指令时，identifier (可选参数) 为压入堆栈的记录指派的一个标识符，当该标识符被删除时，与其相关的堆栈中的记录将被弹出堆栈，“segment-name” (可选参数) 表示函数存放的节名。例如，

```
void fun1 ()                                //函数默认放在“.text”节中
{
...
}
#pragma code_seg ( “.mydata”)
void fun2 ()                                //函数放在 “.mydata” 节中
{
...
}
#pragma code_seg (push, C1, “.mycode”)      //给 “.mycode” 节指定标识符
//C1, 将 C1 压//入堆栈中。
void fun3 ()                                //函数放在 “.mycode” 节中
{
...
}
main()
{
...
}
```

b) 完整的 `data_seg` 使用方法。

`#pragma data_seg` 预处理命令主要用于数据共享，尤其是在 Windows 操作系统中，多个进程之间共享数据，可以使用 `#pragma data_seg` 预处理命令实现。共享的数据既可以是单个的数据，也可以是数组、向量等。必须注意的是，共享数据在数据段中必须初始化，否则可能引起错误。该预处理命令常应用于 DLL 动态链接库中。其使用方法如下：

```
#pragma data_seg ( “publicDATA”)           //共享数据段开始，名称为 publicDATA
int data = 0;                               //定义共享数据，初始化公用数据 data，赋值为 0
#pragma data_seg ()                          //共享数据段结束
#pragma comment (linker, " /section: .SharedDataName, rws") //本语句实现共享数据在 DLL 的所有实例间共享
```

⑧ #pragma resource。

说明：将制订的资源文件加入工程中。例如，

```
#pragma resource “.dfm” //表示把*.dfm文件中的资源加入工程。*.dfm中包括窗体外观的定义
```

⑨ #pragma saveregs。

说明：`#pragma saveregs` 预处理命令保证调用 `huge()` 函数时不会改变任何寄存器的值。如果在 C 语言中可能要嵌入汇编语言代码，就可能用到本命令。该命令应该放在某个函数定义之前，并且仅适用于这个函数。

⑩ #pragma hdrstop。

说明：`#pragma hdrstop` 预处理命令用于结束预编译头文件列表，表示预编译头文件到此为止，后面的头文件不再进行预编译。通过使用该命令来减少预编译头文件所占用的磁盘空间。

⑪ #pragma hdrfile。

说明：`#pragma hdrfile` 预处理命令用来指定保存预编译头文件的文件的名称。在 Turbo C++ 中默认为 `TCDEF.SYM`。如果不使用预编译头文件，本命令无效。例如，

```
#pragma hdrfile “filename.sym”
```

1.1.13 宏

在 C++ 源程序中，允许用一个标识符来表示一个字符串，该标识符被称为“宏”。被定义为“宏”的标识符称为“宏名”。在编译预处理时，对程序中所有出现的“宏名”，都用宏定义中的字符串去代换，这称为“宏代换”“宏替换”或“宏展开”。在本节中，使用 `#define` 命令定义的变量或函数即是典型的宏定义方式。

宏定义是由源程序中的宏定义命令完成的。宏替换是由预处理程序自动完成的。“宏”简单地可分为有参数和无参数两种。源程序被编译或执行时，一旦遇到宏名，将视为对宏的调用。用宏体的副本替换宏名。若宏定义包含参数，则用宏名后面的参数替换宏体中的正式参数。

1. 宏的范围

宏名的有效范围是从定义宏名开始，一直到该源文件结束。例如，若 `#define` 命令写在文件开头、函数之前，则该“宏”作为文件的一部分，在此文件范围内有效。

可以使用 `#undef` 命令终止宏定义的作用域。

2. 无参宏定义

无参数宏的定义格式：`#define 标识符 字符串`

其中标识符就是所谓的符号常量，即“宏”名称。`#define` 和标识符之间用空格隔开，标识符和字符串之间也用空格隔开。

当源程序被预处理（预编译）时，宏名称将被替换为字符串。掌握“宏”的概念，关键是要抓住其“换”的本质，即在对相关命令或语句的含义和功能做具体分析之前要“换”。例如，

```
#define PI 3.1415926
```

上述语句一旦写在源代码中，程序中所有出现的“PI”将全部被替换为“3.1415926”。

一般情况下，宏名需要大写。使用宏可以提高程序的通用性和易读性，减少不一致性，减少错误输入和便于修改；宏定义语句末尾不适用分号；宏可以嵌套；宏定义不分配内存。

3. 有参宏定义

使用宏时，除了一般的字符串替换，还可以进行参数替换。类似于函数调用形式，其格式为：

```
#define S(A,B) A*B
```

#define 和 S (A, B) 之间用空格隔开，S (A, B) 和 “A * B” 之间也用空格隔开。参数 A 和 B 之间用逗号隔开。

需要注意的问题：

- 1) 在进行宏定义时，如果需要使用括号，就一定要使用括号。
- 2) 宏替换是在编译时实现的，不占用程序的执行时间。
- 3) 函数一般只有 1 个返回值，宏可以有多个返回值。

例 1-20

```
#include "stdio.h"
#define C(A,B) A*B
void main()
{
    int AC=0;
    AC=C(3,4);
    printf("C=%d\n",AC);
    getchar();
}
```

4. 嵌套宏

宏还可以实现嵌套，即使用已经定义过的“宏”来定义新的宏。例如，

```
#define PI 3.1415926
#define R 5
#define S PI*R*R
```

此处的“嵌套”是指宏 S 定义时，使用了已经定义过的宏 PI 和 R。

例 1-21

```
#include "stdio.h"
#define PI 3.1415926
#define R 5
#define S PI*R*R
#define C(A,B) A*B
void main()
{
    int AC=0;
    printf("S=%5.3f\n",S);
    AC=C(3,4);
    printf("C=%d\n",AC);
    getchar();
}
```

5. 常见的预定义宏

每一种版本的C++语言都定义了一些全局标识符。预定义宏至少需要和包含7种，另外还包括一些具有附加条件定义的宏。在C语言中，没有定义`__cplusplus`宏。每个预定义宏的名称以两个下画线字符开头和结尾，这些宏不能被程序员取消（`#undef`）或重新定义。下面依次介绍。

(1) 基本预定义宏

`__cplusplus` 在C++编译系统对该宏进行编译时，宏定义本身被赋值为201103L。

`__DATE__` 本宏提供预处理程序开始处理当前源文件的日期。在文件中，每一个`__DATE__`都给出同一值。日期格式表达为：`mmm dd yyyy`，这里`mmm`表示月份（如Feb, Apr）；`dd`表示日期（如1, 2, 10），若小于10，则`dd`的第一个字符为空；`yyyy`表示年份（如1999, 2009）。

`__FILE__` 本宏提供当前正在处理源文件的文件名，表示为字符串型常量。在程序执行过程中，当处理的文件发生变化时，本宏的值均会发生变化。

`__LINE__` 本宏提供当前正在处理源文件的当前行号。通常源文件的第一行定义为1，通过`#line`命令可改变它。

`__STDC__` 本宏定义为整数1。

`__STDC_HOSTED__` 若作为宿主形式使用，本宏定义为1；否则，本宏定义为0。

`__STDC_VERSION__` 在C89（该版本的C语言常被称作“ANSIC”）中，本宏定义为199409L；在C99中，本宏定义为199901L，否则数值是未定义。

`__TIME__` 本宏提供开始处理当前源文件的时间。它的格式为：“`hh:mm:ss`”。

(2) 其他预定义宏

`__STDC_IEC_559__` 在IEC 60559浮点数算法中，本宏定义为整数1。

`__STDC_IEC_559_COMPLEX__` 在IEC 60559复数算法中，本宏定义为1。

`__STDC_ISO_10646__` 在C99中，本宏定义为长整型变量，格式为：`yyyymmL`。其中的10646表示该宏符合ISO10646标准。

`__STDC_MB_MIGHT_NEQ_WC__` 宏定义的值为整型常量1。

`__STDCPP_THREADS__` 其值为整型常量1。当且仅当程序多于一个线程时，其值为1。

后续章节将会逐渐详细讲述这些宏的意义及其使用方法。

1.2 类模板定义

本书将正式开始讲述C++ STL（标准模板库）的内容。STL的一个重要特点是数据结构和算法的分离。这种分离增强了STL的通用性。

STL的另一个重要特点是它不是面向对象的。STL主要依赖于模板。这使得STL的组件具有广泛通用性的底层特征。由于STL基于模板，内联函数的使用使得生成的代码短小高效。STL包含了诸多在计算机科学领域中常用的基本数据结构和基本算法，为广大C++程序员提供了一个可扩展的应用框架，高度体现了软件的可复用性。

从实现层次看，整个 STL 是以模板作为基石的。

STL 背后蕴含着泛型化程序设计 (GP) 的思想。在这种思想中，大部分基本算法被抽象、被泛化，独立于对应的数据结构，用于以相同或相近的方式处理各种不同的情形。

Alexander Stepanov 是 STL 的创建者。20 世纪 70 年代，他开始研究将算法从诸多具体应用中抽象出来的可能性。这是泛型思想的最早雏形。后来，他和纽约州立大学教授 Deepak Kapur 以及伦塞里尔技术学院教授 David Musser 共同开发了一种叫作 Tecton 的语言。但 Tecton 语言最终没有取得实用性的成果。之后，Alexander Stepanov 又和他人合伙建立了一些大型程序库。由于当时的面向对象程序设计思想存在一些问题，例如抽象数据类型概念的缺陷，他希望通过软件各部分分类，逐渐形成软件设计的概念性框架。

1987 年左右，Alexander Stepanov 在贝尔实验室工作期间，开始采用 C++ 语言进行泛型化软件库研究，并开发了庞大的算法库。1988 年，Alexander Stepanov 进入惠普实验室工作，并继续进行泛型化算法的研究，最终开发出了包含大量数据结构和算法部件的庞大运行库。这便是现今 STL 的雏形，即 HP STL。1993 年 9 月，Alexander Stepanov 为 ANSI/ISO C++ 标准委员会做了相关演讲。1994 年 3 月，Alexander Stepanov 向 ANSI 递交建议书，希望 STL 成为 C++ 标准库的一部分，但最终未被采纳。之后，Stepanov 对原有的 STL 进行了改进，增加了封装内存模式信息的新模块 (allocator)，使 STL 的大部分功能独立于具体的内存模式，并独立于具体的操作系统平台。在 1994 年 7 月的滑铁卢会议上，ANSI 最终通过了 Alexander Stepanov 的提案，决定将 STL 正式纳入 C++ 标准化进程之中。至此，STL 终于成为 C++ 家族的一员。

1998 年，ANSI/ISO C++ 标准正式定案之后，STL 即被纳入整个 C++ 标准库中。

那么，类模板的定义到底是什么呢？

在 ISO/IEC 14882: 2003 (E) 中，类模板的英文定义是：“A class template defines the layout and operation for an unbounded set of related types.”，较恰当的中文翻译应该是：“对于一些非紧密相关的数据类型，类模板定义了它们共同的设计 (类的结构) 和操作 (子函数)”。

1.2.1 类模板实例化

类模板仅仅是模板。“如何使用模板”涉及类模板的实例化问题。模板实例化一般指使用模板类和模板参数生成一个类声明的过程。



提示 从上述对模板实例化的概念可知，使用模板和模板参数可以产生一个类的声明。同样，使用其他模板和相关参数也可以生成一个其他的 C/C++ 实例化。例如使用函数模板和相关的模板参数生成 (实例化) 相应的函数。

模板还可以使用较短的源代码定义生成代码，是非常强有力的程序开发方法。在使用时，要避免使用大量几乎相同的函数定义。

模板的具体形式是：`template <class Type > .`

后面章节会逐渐介绍很多容器类，例如 list, stack, map 等。

在实例化过程中，需要声明一个类型为模板类的对象，即使用所需的具体类型替换通用类型名。例如，

```
list<int>mylist;
stack<int>mystack;
```



提示 类模板实例化其实就是怎样使用类模板生成类。这样才能使用生成的类定义对象，并加以使用。本节没有大篇幅地讲述类模板的抽象知识，而是通过简单的讲述形式，对类模板的功能和意译加以阐述，之后直接给出实例形式，以免让读者开始学习就困于抽象的模板理论中。

下面举例说明类模板的使用方法。提示：注意代码中的黑体字。

例 1-22

```
#include <iostream>
#include <list>
#include <ctime>
using namespace std; //必须
voidmysleep(int second) //自己编写的延时函数
{
    clock_t st;
    st = clock();
    while (clock() - st < second*CLOCKS_PER_SEC);
}
void main()
{
    int count = 5;
    float number = 0.0;
    list<int> mylist; //对 list 模板类进行实例化，list<int>
    cout << " 请任意输入 5 个数字:" << endl;
    while (count --)
    {
        cin >> number;
        mylist.push_back (number); //将输入数据压入列表 list
    }
    list<int>:: iterator iter;
    for (iter=mylist.begin(); iter! =mylist.end(); iter++) //输出列表 list 的元素
        cout << *iter << " , ";
    cout << endl;
    return;
}
```

1.2.2 类模板的成员函数

类模板的成员函数可以被类模板实例化产生的类所拥有。每个类模板都有自己相应的成员函数，并且这些函数可以被模板的实例调用。

若类模板的成员函数也是用模板实现的，当使用时必须将其实例化。具体形式如下：

- 1) 必须以关键字 `template` 开头。
- 2) 必须指出是哪个类的成员。
- 3) 类名必须包含模板形参。

例如，

```
template <class T> ret-type Queue::<T>::member - name
```

对于 `Queue` 类的 `destroy()` 函数，其定义源代码为（摘自《C++ Primer 中文版》第 4 版 814 页）：

```
template <class Type> void Queue<Type>::destroy()
{
while(! empty())
    pop();           //依次取出元素
}
```

对于 `vector` 的 `insert()` 函数，其定义源代码为（摘自《STL 源码剖析》第 124 页）：

```
template <class T, class Alloc>
void vector<T,Alloc>::insert(iterator position,size_type n, const T&x)
{
...
}
```



提示 这里主要是明白其中的道理。本书重点讲授 STL 的使用。对于 STL 内部深奥的理论和逻辑，读者了解即可。

1.2.3 类模板的静态成员

在 C++ 中，类的成员变量被声明为 `static` 型就意味着它为该类的所有实例所共享，即当某个类的实例修改了该静态成员变量时，其修改值为该类的其他所有实例所见。下面详细说明静态成员的使用方法。

类模板的成员可以被声明为 `static` 型，例如，

```
template <class T>           //声明下面是一个类模板
class BClass
{
public:
    static int count;       //静态成员变量
    static long size;      //静态成员变量
public:
```

```
static long GetSize() {                //静态成员函数
    return size;
}
static int GetCount() {                //静态成员函数
    return count;
}
};
```

在上述代码中，BClass 作为类模板的名称，count 和 size 是类模板的成员变量，GetSize() 和 GetCount() 是类模板的成员函数。

如要在程序中需要使用类模板的成员变量，必须在程序的类外部（或程序中）对该 static 静态成员进行定义。例如，

```
template <class T> int BClass<T>::count = 0;
template <class T> long BClass<T>::size = 2;
```

如果在程序中需要使用类模板的成员函数，必须在程序中使用类模板创建对象，进而使用对象访问该成员函数；或者使用类模板的作用域操作符直接访问成员函数。例如，

```
BClass<int> myO;
cnt = myO.GetCount();
nsize = myO.GetSize();
```

完整的使用静态成员的示例如下。

例 1-23

```
#include <iostream>
using namespace std;
template <class T>
class BClass
{
public:
    static int count;
    static long size;
public:
    static long GetSize() {
return size;
}
    static int GetCount() {
return count;
}
};
template <class T> int BClass<T>::count = 0; //在类模板外部定义类模板的静态成员变量 count
template <class T> long BClass<T>::size = 2; //在类模板外部定义类模板的静态成员变量 size
void main()
{
    int cnt;
    long nsize;
```

```

BClass <int > my0;
    my0.count = 2;
    my0.size = 3;
cnt = my0.GetCount();
nsize = my0.GetSize();
cout << cnt << ", " << nsize << endl;
}

```

例 1-23 的执行效果如图 1-9 所示。



图 1-9 例 1-23 的执行效果

总结 本小节主要讲述如何在类模板中使用静态成员的方法。虽然略有些难度，但经过上述的讲解，读者应已基本掌握了这一内容。类模板的静态成员函数可以通过类模板定义对象或作用域操作符的形式调用；而类模板的静态成员变量必须在程序中（类外部）声明。

1.3 成员模板

C++ STL 支持“成员模板”新特性，即模板可用作结构、类或模板类的成员。在完全实现 STL 设计的过程中，这项特性是必须使用的。成员模板的定义一般为：任意类（可以是类模板，也可以不是类模板）可以拥有类模板或函数模板作为其成员。在 ISO/IEC14882:2003 (E) 中，关于成员模板的说明是：“一个模板可以在一个类或类模板中声明，这样的类模板称为成员模板；成员模板的定义既可以在类（或类模板）定义的内部，也可以在类（或类模板）定义的外部；当类模板的成员模板在类模板定义的外部定义时，应该完整地指定类模板参数和成员模板的参数。例如，

```
template <class T>template <class I > void queue::assign(I beg, I end);
```

上述代码中的 `template <class T >` 是类模板，作为第一个模板形参表；`template <class I >` 是成员模板，作为第二个模板形参表。例如，

```

template <class T>class string //类模板 string
{
public:
    template <class T2 >int compare(const T2&);
    ...
}

template <class T>template <class T2 >int string <T >::compare(const T2& s) //在类外部定义成员函数
{
    ...
}

```

上述代码在类模板 `string` 中，使用了另一个类模板类型的成员函数。再如，

```
template <typename T> class bC
{
public:
    template <typename V> class HC //类模板 HC 作为模板 bC 的成员
    {
        public:
            V m;
            V n;
        public:
            HC(){};
            void show(){cout <<m <<endl;};
    }
    HC <T> A;
    HC <int> B;
public:
    bC(){};
    template <typename U>U show_m(){}; //类模板定义成员函数
}
```

上述代码举例说明了使用类模板作为另一类模板的成员。另外，成员模板具有如下一些使用规则。

- 成员模板遵循常规访问控制。
- 成员模板不能为虚（`virtue`）。
- 局部类不能拥有成员模板。
- 析构器不能是模板类型。
- 成员函数模板不能重载基类的虚函数。
- 成员模板具有复杂的转换功能。



总结 本小节的重点是对成员模板的理解。最简单的理解为：在类模板中包含了类模板，被包含的类模板作为其成员。

1.4 友元模板

友元机制允许一个类对其非公有成员的访问权授予指定的函数或类。一般友元声明以关键字 `friend` 开始。友元只能出现在类定义的内部。而友元的声明则可以出现在类中的任何地方。一般友元声明被成批地放在类定义的开始或结尾。最简单的使用友元机制的例子如下：

```
class Screen{
    friend class Windows;
    ...
}
```

在上述代码中，类 `Windows` 被声明为类 `Screen` 的友元类，之后在类 `Windows` 中可以使

用类 Screen 的非公有成员, 包括其私有成员。友元可以是非成员函数, 也可以是其他类的成员函数, 或整个类。

在 ISO/IEC14882: 2003 (E) 中, 类或类模板的友元可以是函数模板或者类模板。友元模板可以在类 (或类模板) 内部被声明。友元函数可以在类 (或类模板) 中定义, 而友元类模板不可以在类 (或类模板) 中被定义。

类模板中友元定义主要分为以下三类。

- 1) 非模板函数、类作为实例类的友元。
- 2) 模板函数、模板类作为同类型实例类的友元。
- 3) 模板函数、类作为不同类型实例类的友元。

下面分别举例说明。

- 1) 非模板函数、类作为实例类的友元。例如,

```
class A{
void AF();
};
template <class Type >
class Queue
{
friend class B;           //类 B 不需要事先声明
friend void A();         //函数 A ()
friend void A::AF();     //类 A 必须事先声明
}
```

- 2) 模板函数、模板类作为同类型实例类的友元。例如,

```
template <class Type > class A{...};
template <class Type > void D() (B <Type >);
template <class Type > class C{ void Cf();...};
template <class Type > class B
{
friend class A <Type >;    //模板类 A 需要先定义或声明
friend void D(B <Type >); //模板函数 D ()需要先定义或声明
friend void C <Type >::Cf(); //模板类 C 必须先定义
};
```

- 3) 模板函数、类作为不同类型实例类的友元。例如,

```
template <class T > class B
{
friend <class Type > friend class A;
template <class Type > friend void D(B <Type >);
template <class Type > friend void C::Cf();
}
```

1.5 函数模板

学习和使用 STL 的程序员或者研究人员可能更需要快速地掌握和使用 STL 提供的各种

容器。但是，函数模板的作用极其强大，也是不可忽略的。

函数模板定义了一个无限的相关函数集合。当函数除了数据类型不一致外，其余的处理全部相同。此时函数模板随之诞生。这是创建函数模板的根本原因。

函数模板可以定义参数化的非成员函数，使程序员能够用不同类型的参数调用相同的函数，由编译器决定该采用哪种类型，从函数模板中生成相应的代码。



提示 在例 1-24 的源代码中，`template <typename T> void print (const T& var)` 是函数的定义，其中 T 代表数据类型或者类，函数 `print` 的参数是通用数据类型 T。

例 1-24

```
#include <iostream>
#include <string>
using namespace std;
template <typename T> void print (const T& var)    //也可以使用 template <class T> 声明函数模板
{
    cout << var << endl;
}
void main()
{
    string str("Hello Beijing!");
    print(str);
    cin.get();
}
```

例 1-24 的执行效果如图 1-10 所示。



图 1-10 例 1-24 的执行效果



提示 可以使用例 1-25 中的 `template <typename T>` 定义函数模板，也可以使用 `template <class T>` 定义函数模板，两者没有区别。目前程序员多用后一种方法，主要是因为关键字 `typename` 的诞生时间较短，多数人不知晓或者不习惯使用。

例 1-24 介绍了函数模板的使用方法。如果函数模板需要定义多个参数，可以对这些参数随意命名。每个参数必须以关键词开头，彼此之间使用逗号分隔。这样，函数模板的便捷性取代了多态函数的繁杂性，其意义可谓深远。

例 1-25

```
#include <iostream>
#include <string>
using namespace std;
```

```
template <class T> void print(T& ii, T& jj)           //函数模板
{
    cout << ii << endl;
    cout << jj << endl;
}
void main()
{
    string strA("Hello Beijing!");
    string strB("I am a programmer!");
    print(strA, strB);
    cin.get();
}
```

例 1-25 的执行效果如图 1-11 所示。

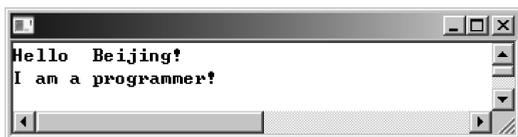


图 1-11 例 1-25 执行效果

在例 1-25 中，函数模板的两个参数是同一类型，还可以使用多种类型的多个参数。



提示 请注意下面源代码中的黑体字。

例 1-26

```
#include <iostream>
#include <string>
using namespace std;
template <class T1, class T2> void print(T1& ii, T2& jj)           //函数模板
{
    cout << ii << endl;
    cout << jj << endl;
}
void main()
{
    stringstrA("Hello Beijing!");           //定义 string 对象
    int B=50;
    print(strA, B);           //输出
    cin.get();
}
```

例 1-26 的执行效果如图 1-12 所示。



图 1-12 例 1-26 执行效果



总结 使用函数模板实现多个参数时，函数模板的参数既可以是同类型数据，也可以不是同类型数据。

函数模板的功能极其强大，但也存在个别情况。当待比较的函数模板没有提供正确的操作符或者方法时，程序不会对此进行编译。为避免此类错误，程序员可以使用函数模板与同名的非模板函数重载，即函数的定制。本书不再赘述。

下面提供一个函数模板的实例，供读者参考。例 1-27 中函数模板的功能是将任何数据类型转换为字符串，还可以将字符串转换为其他数据类型的数据。

首先，创建模板函数 `toString()` 和 `fromString()`，并将其放在头文件 `Ex27.h` 中。

例 1-27

```
#ifndef Ex27_H
#define Ex27_H
#include <iostream>
#include <string>
#include <sstream>
template <typename T> T fromString (const std:: string& s) //模板函数，将字符串转化成其他数据类型
{
    std:: istringstream is (s) ;
    T t;
    is >>t;
    return t;
}
template <typename T> std::  string toString (const T& t) //模板函数，将其他数据类型转换成字符串
{
    std::  ostringstream s;
    s <<t;
    return s. str ();
}
#endif
```

其次，创建主程序 `Ex27.cpp`。

```
#include "Ex27.h"
#include <iostream>
#include <complex>
using namespace std;
void main()
{
    int i =1234;
    float j =567. 34;
    complex <float > c (2. 5,4. 1) ;
    cout << "i == \"\" << toString(i) << \"\" << endl; //将数字转换为字符串,并输出
    cout << "i == \"\" << toString(j) << \"\" << endl; //将数字转换为字符串,并输出
    cout << "i == \"\" << toString(c) << \"\" << endl; //将数字转换为字符串,并输出
    //...
```

```

i = fromString<int>(string("1234"));           //将字符串转换为数字
j = fromString<float>(string("567.34"));      //将字符串转换为数字
c = fromString<complex<float>>(string("(2.5,4.1)")); //将字符串转换为数字
cout << "i == \"" << i << "\"" << endl;
cout << "j == \"" << j << "\"" << endl;
cout << "i == \"" << c << "\"" << endl;
}

```

例 1-27 的执行效果如图 1-13 所示。

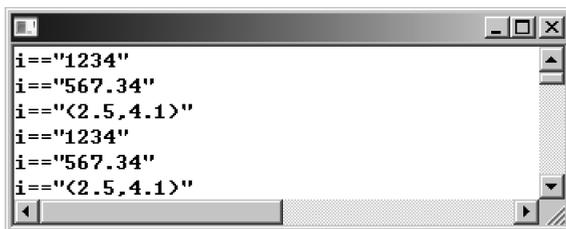


图 1-13 例 1-27 的执行效果



总结 本小节主要讲述了函数模板的一些特性和使用方法。通过 4 个例题详细地演示函数模板的使用。希望读者在阅读文字内容的同时，认真阅读本节的 4 个例题。

1.6 类模板的参数

前面几节的知识讲解已经涉及了模板的参数。本小节将详细地对模板参数进行阐述。模板参数可以是类型参数，也可以是常规类型参数，并且一个模板可以有多个参数。

如果是类型参数，类型参数可以采用 `typename` 或 `class` 关键字指明。

```

template <class T1, class T2, class T3> class classname;
template <typename T1, typename T2, typename T3> class classname;

```

如果是常规类型参数，即采用通常的参数定义。

```

template <class Type, int size> class Queue;

```

对于上述代码，实例化时可采用如下形式：

```

Queue <int, 100> iq;

```

如果在声明类模板时，已经明确了模板参数：

```

template <class Type = int, int size = 1024> class Queue;

```

则实例化时可采用以下几种形式：

```

Queue <int, 100> iq1;
Queue <int> iq2;
Queue <> iq3;

```

整数参数可以用来提供大小或界限。

1. typename 关键字的使用

typename 关键字告诉编译器，其后的名称为一个类型，并且可以用其来创建实例。例如，

```
template <class T> class Y
{
    typedef typename T::A TA;
    TA*   abc;
    ...
}
```

在上述代码中，如果没有 typename 关键字，编译程序则无法理解 T::A 是什么。

由此可知，typename 关键字主要用于解决由类模板机制带来的语义歧义，使模板声明更加直观。

类模板及其成员函数使用 typename 关键字来限定所有对参数化类型的定义类型的引用。编译器一旦遇到 typename 关键字，就知道其后紧跟的是数据类型，并且在分析代码时假定类型的名字通过实例来填充。根据 C++ 标准的要求，如果类型在模板化的类中定义，在模板中引用该类型时，都要使用 typename 限定符。下面举例说明 typename 的用法。

例 1-28

```
#include <iostream>
template <class T> class Table{           //模板类 Table
    typename T::iter t;                 //typename 关键字说明 T 代表类型
public:
    explicit Table(const typename T::iter& ti):t(ti) //构造器
    {
    }
    void show()                               //输出
    {
        typename T::iter* y;
        y=&t;
        std::cout << (* y) ->p << std::endl;
    }
};
class card{
public:
    typedef card* iter;
    int p;
    card(int pos):p(pos)                 //构造器
    { }
};
void main()
{
    card pc(3);
    Table<card> S_Table (&pc);           //使用类 card 作为类模板 Table 的参数
    S_Table.show();                     //输出
    return;
}
```

例 1-28 的执行效果如图 1-14 所示。



图 1-14 例 1-28 的执行效果



总结 请读者认真琢磨例 1-28，体会其中的意思。

2. typename 与 class

在引入 typename 之前，class 关键字早已在模板声明中被予以使用。C++ 标准委员会把 typename 关键字引入标准 C++ 中，并用其替代 class 来声明参数化类型的模板中。例如，

```
template <class T1, class T2 >
```

在引入 typename 关键字之后，标准 C++ 允许在模板声明中用 typename 关键字替代 class，例如，

```
template <typename T1, typename T2 > //
```

例 1-29

```
#include <iostream>
using namespace std;
template <typename T> T doubleV(T Val) //模板函数 double(T Val)
{
    return Val*2;
};
template<typename T1, typename T2> class TwoThing{ //模板类
    T1 th1;
    T2 th2;
    T1 th3;
public:
    TwoThing(T1 tt1, T2 tt2)
    {
        th1 = tt1;
        th2 = tt2;
        th3 = doubleV<T1>(tt1);
    }
    void show() //输出
    {
        cout << th1 << " , " << th2 << " , " << th3 << endl;
    }
};
void main()
{
```

```
int m=123;
double n=456.789;
TwoThing<int, double> my(m,n);           //定义模板对象
my.show();
return;
}
```

例 1-29 的执行效果如图 1-15 所示。



图 1-15 例 1-29 的执行效果



总结 本小节先讲述了模板参数的概念和意义；之后详细讲述了 `typename` 关键字的使用以及它与 `class` 的关系。读者应重点学习本节内容。

1.7 STL 简介

本节主要讲述 STL 历史、STL 组件、STL 基本结构以及 STL 编程概述。STL 历史可以追溯到 1972 年 C 语言在 UNIX 计算机上的首次使用。直到 1994 年，STL 才被正式纳入 C++ 标准中。STL 组件主要包括容器，迭代器、算法和仿函数。STL 基本结构和 STL 组件对应。STL 主要由迭代器、算法、容器、仿函数、内存配置器和配接器六部分组成，可帮助程序员完成许多功能完善、形式多样的程序。

1.7.1 STL 历史

C 语言是 1972 年由美国的 Dennis Ritchie 设计发明的，并首次在 UNIX 操作系统的计算机上使用。C 语言由早期的汇编语言 BCPL 发展演变而来。随着微型计算机的日益普及，C 语言出现了许多其他版本。由于没有统一的标准，各版本之间出现了不一致之处。ANSI 因此为 C 语言制定一套 ANSI 标准，后来成为现行的 C 语言标准。

早期的 C 语言主要用于 UNIX 系统。C 语言因其强大的功能和各方面的优点逐渐被人们认识。20 世纪 80 年代，C 语言开始应用于其他操作系统，并很快在各种计算机上得到广泛应用，成为当代最优秀的程序设计语言之一。

C 语言的表现能力和处理能力极强。它不仅具有丰富的运算符和数据类型，便于实现各类复杂的数据结构，还可以直接访问内存物理地址，甚至进行位操作。此外，C 语言还可实现对硬件的编程操作，十分便捷方便。

1983 年，贝尔实验室的 Bjarne Stroustrup 推出了 C++。C++ 进一步扩充和完善了 C 语言，成为面向对象的程序设计语言。最初 C++ 主要用于小型计算机系统。1988 年，出现了第一个用于 PC 的 ZORTECH C++ 2.0 编译系统；1989 年，出现了 Turbo C++ 2.0 编译器。

从1991年开始, Borland公司陆续推出了 Borland C++ 2.0/3.0/4.0系统。而微软公司直到1992年,才推出基于DOS的MS C/C++ 7.0系统。1993年,微软推出了面向Windows的Visual C++1.0系统,并于1998年推出了Visual C++6.0。

C语言提供了具有可适应性的、强大的抽象机制,用于对问题进行抽象。这种语言结构允许程序员创建和使用新的类型,而这些新的类型则可以与实际应用中所包含的概念相适应。在C++的最新发展过程中,C++新增了模板新特性。通过使用模板,程序具备更好的代码重用性能。1994年7月,美国国家标准与技术研究院通过投票决定,将STL纳入C++标准,使之成为C++库的重要组成部分。1997年,C++标准完成了最近一次的修改,官方名称为ISO/IEC 14882。

STL从根本上讲是“容器”的集合,也是组件的集合。容器包括list、vector、set、map等;组件包括迭代器等。STL的目的是标准化组件,与Visual C++中的ATL相似。STL是C++的一部分,不用额外安装,被内建在支持C++的编译器中。STL的算法是标准算法。STL实现了将已经定义好的算法应用在容器的对象上。

1.7.2 STL 组件

STL是C++标准程序库的核心。STL内的所有组件都由模板构成,其元素可以是任意型别。程序员通过选用恰当的群集类别调用其成员函数和算法中的数据即可,但代价是STL晦涩难懂。

本节将介绍STL的各种组件。STL组件主要包括容器,迭代器、算法和仿函数。

容器即用来存储并管理某类对象的集合。例如鱼缸是用来盛放金鱼的容器。每一种容器都有其优点和缺点。为满足程序的各种需求,STL准备了多种容器类型。容器可以是arrays或是linked lists,或者每个元素有特别的键值。

迭代器用于在一个对象群集的元素上进行遍历动作。对象群集可能是容器,也可能是容器的一部分。迭代器的主要用途是为容器提供一组很小的公共接口。利用这个接口,某项操作可以行进至群集内的下一个元素。每种容器都提供了各自的迭代器。迭代器了解该容器的内部结构,所以能够正确行进。迭代器的接口和一般指针类似。

算法用来处理群集内的元素,可以出于不同目的搜寻、排序、修改、使用那些元素。所有容器的迭代器都提供一致的接口,通过迭代器的协助,算法程序可以用于任意容器。

STL的一个特性是将数据和操作分离。数据由容器类别加以管理,操作则由可定制的算法定义。迭代器在两者之间充当“粘合剂”,以使算法可以和容器交互运作。

STL的另一个特性即组件可以针对任意型别运作。“标准模板库”这一名称即表示“可接受任意型别”的模板,并且这些型别均可执行必要操作。

在STL中,容器又分为序列式容器和关联式容器两大类,而迭代器的功能主要是遍历容器内全部或部分元素的对象。迭代器可划分为5种类属。这5种类属归属两种类型:双向迭代器和随机存取迭代器。STL中提供的算法包括搜寻、排序、复制、重新排序、修改、数值运算等。STL中大量运用了仿函数。仿函数具有泛型编程强大的威力,是纯粹抽象概念的例证。

1.7.3 STL 基本结构

STL是C++通用库,由迭代器、算法、容器、仿函数、配接器和配置器(即内存配置

器) 组成。

1. 容器

STL 包含诸多容器类。容器类是可以包含其他对象的类, 就像数组和队列堆栈等数据结构包含整数、小数、类等数据成员一样。STL 可以包含常见的向量类、链表类、双向队列类、集合类、图类等。每个类都是一种模板。这些模板可以包含各种类型的对象。下述代码是常用的 `vector` 赋值方法:

```
vector<int> l;  
for(int i=0;i<100;i++)  
    l.push_back(i);
```

下述代码采用 `map` 容器进行二维元素的管理:

```
map<string, string, less<string>> cap;           //按从小到大排序  
cap["Ark"] = "Little Rock";  
cap["New"] = "Albany";
```

`map` 类似于二维数组, 但比二维数组灵活得多。

目前, STL 中已经提供的容器主要如下。

- `vector<T>`。一种向量。
- `list<T>`。一个双向链表容器, 完成了标准C++ 数据结构中链表的所有功能。
- `queue<T>`。一种队列容器, 完成了标准C++ 数据结构中队列的所有功能。
- `stack<T>`。一种栈容器, 完成了标准C++ 数据结构中栈的所有功能。
- `deque<T>`。双端队列容器, 完成了标准C++ 数据结构中栈的所有功能。
- `priority_queue<T>`。一种按值排序的队列容器。
- `set<T>`。一种集合容器。
- `multiset<T>`。一种允许出现重复元素的集合容器。
- `map<key, val>`。一种关联数组容器。
- `multimap<key, val>`。一种允许出现重复 `key` 值的关联数组容器。

以上容器设计高效, 还提供了接口。程序员可以在任何适当的地方使用它们。

容器可以分为序列式容器和关联式容器两大类。序列式容器主要有 `vector`、`list` 和 `deque`; 关联式容器包括 `set`、`map`、`multiset` 和 `multimap` 等容器模板类。

2. 算法

STL 提供了非常多的数据结构算法。这些算法在命名空间 `std` 的范围内定义, 通过包含头文件 `<algorithm>` 来获得使用权。

常见的部分算法如下。

```
for_each()  
find()  
find_if()  
count()  
count_if()  
replace()  
replace_if()
```

```
copy()
unique_copy()
sort()
equal_range()
merge()
```

STL 中的所有算法都是基于模板实现的。

3. 迭代器

通俗来讲, 迭代器就是指示器。迭代器技术能够使程序非常快捷地实现对 STL 容器中内容的反复访问。反复访问意味着一次可以访问一个或多个元素。迭代器为访问容器提供了通用的方法, 类似于 C++ 的指针。当参数化类型是 C++ 内部类型时, 迭代器即 C++ 指针。STL 定义了 5 种类型的指示器, 并根据其使用方法予以命名。每种容器都支持某种类别的迭代器。常见的迭代器包括输入、输出、前向、双向和随机接入等类别。

输入迭代器主要用于为程序中需要的数据源提供输入接口, 此处的数据源一般指容器、数据流等。输入迭代器只能从一个序列中读取数值。该迭代器可以被修改和被引用。

输出迭代器主要用于输出程序中已经得到的数据结果 (容器, 数据流)。输出迭代器只能向一个序列写入数据。该迭代器也可以被修改和被引用。

双向迭代器既可以用来读又可以用来写, 它与前向迭代器相类似。双向迭代器可以同时进行前向和后向元素操作。所有 STL 容器都提供了双向迭代器功能, 这既有利于数据的写入和读出, 又有利于提供更加灵活的数据操作。

有的容器甚至提供了随机接入迭代器。随机接入迭代器可以通过跳跃的方式访问容器中的任意数据, 使数据的访问非常灵活。随机访问迭代器具有双向迭代器的所有功能, 是功能最强大的迭代器类型。

迭代器的诞生使算法和容器分离成为可能。算法是模板, 其类型依赖于迭代器, 不会局限于单一容器。不同的 STL 算法需要不同类型的迭代器来实现相应的功能。因为不同类型的 STL 容器支持不同类型的迭代器, 所以不能对所有容器使用相同的算法。

4. 仿函数

理解了算法、迭代器、容器之后, 读者已经熟悉了 STL 的大部分内容。STL 还有两个最重要的内容: 仿函数和内存配置器。本小节简单介绍仿函数, 下一小节介绍内存分配器。

STL 包含了大量仿函数。仿函数可以理解为函数的一般形式。对于编程来说, 仿函数非常重要, 并有几种约束。在 C++ 标准中, 函数调用一般使用指针, 当需要调用函数时, 只需要提供函数的地址即可。例如,

```
static int cmp(int*I, int*j){return (*i-*j);}
```

上述代码定义了一个 `cmp()` 函数, 当需要调用此函数时, 只需提供函数的地址即可。例如,

```
qsort(a,10,sizeof(int), cmp);
```

此方法的最大缺陷是效率低。为提高效率, STL 定义了仿函数这种概念。下述代码定义了一个仿函数, 其特征是使用 `operator` 实现定义。

```
struct three_mul
```

```

{
    bool operator() (int& v)
    {
        return (v%3 ==0)
    }
}

```

通过运算符定义显著提高效率。例如，

```
for_each (myvector.begin(), myvector.end(), three_mul);
```

5. 内存配置器和配接器

STL 包括底层的内存分配和释放。内存配置器非常少见，在此可以忽略，在后面章节专门介绍。配接器可以实现不同类之间的数据转换。最常用的配接器有 `istream_iterator`，它提供了函数复制的接口。配接器对于 STL 技术来说非常重要。STL 提供了 3 种容器配接器：`stack < Container >`，`queue < Container >`，和 `deque < Container >`。

例 1-30

```

#include <iostream>
#include <stack>
using namespace std;
void main()
{
    stack<int> st; //定义堆栈对象
    for(int i=0;i<10;i++)
        st.push(i); //将数据压入堆栈
    while(! st.empty())
    {
        cout << st.top() << " "; //弹出堆栈的第一个元素,并输出
        st.pop(); //弹出堆栈元素
    }
    cout << endl;
    cin.get(); //任意键退出
}

```

例 1-30 执行结果如图 1-16 所示。



图 1-16 例 1-30 的执行效果

1.7.4 STL 编程概述

STL 作为 C++ 通用库，主要由迭代器、算法、容器、仿函数、内存配置器和配接器等六大部分组成。程序员使用 STL 容器能够实现多种标准类型且操作便捷的容器。对于编程人

员, 标准化组件意味着直接使用现成的组件, 不用重复开发。使用 STL 最重要的是掌握基本理论和编程方法, 了解 STL 编程技术, 必须深刻掌握 STL 容器技术和 STL 迭代器技术。

STL 提供了一组表示容器、迭代器、仿函数和算法的模板。容器是类似数组的单元, 可存储若干个值, 且 STL 容器是同质的, 即存储的值类型相同; 算法是完成特定任务的处方; 迭代器能够用来遍历容器的对象, 与能够遍历数组的指针类似, 是广义指针; 仿函数是类似于函数的对象, 可以是类对象或函数指针。STL 使程序员能够构造各种容器和执行各种操作。

下面以矢量为例, 简要讲述矢量模板的使用。在数学计算和 STL 模板中, `vector` 对应数组, 提供与 `valarray` 和 `ArrayTP` 类似的操作。而 STL 为使 `vector` 矢量具备通用性, 在头文件 `<vector>` 中定义了 `vector` 模板。具体方法为: 创建 `vector` 模板对象, 使用通常的 `<type>` 表示法指出要使用的类型; 然后使用初始化参数决定矢量的大小, 并定义矢量动态内存。例如,

```
#include <vector>
using namespace std;                //使用命名空间 std
vector <int > ratings(5);           //定义矢量对象
int n;
cin >> n;                          //输入矢量大小
vector <double > scores(n);        //定义矢量动态内存
```

内存分配器是用来管理对象内存的。在 STL 容器模板中, 一般都有一个可选的模板参数。例如,

```
template <class T, class Allocator = allocator<T>>    //矢量模板
class vector
{
    ...
}
```

若省略该模板参数的值, 则容器模板将默认使用 `allocator<T>` 类。类 `allocator` 以标准形式使用 `new` 和 `delete` 内存管理方式。下面举例说明。例 1-31 中创建了两个 `vector` 对象: 一个是 `int` 规范; 另一个是 `string` 规范。



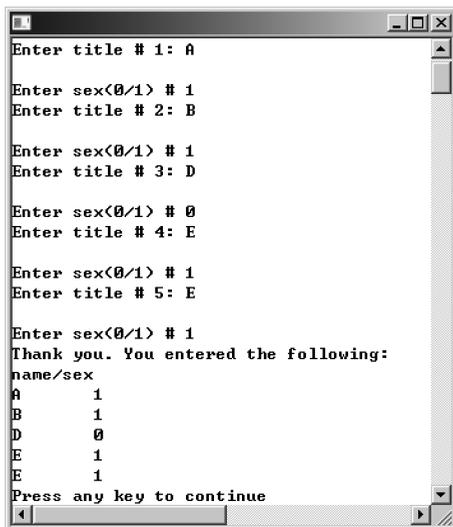
提示 请注意代码中的黑体字。

例 1-31

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;
const int NUM=5;
void main()
{ vector<string>names(NUM);           //定义矢量对象
  vector<int>sexs(NUM);              //同上
  cout << "Please Do Exactly As Told You Will enter \n" << NUM << " Personal Name and Their Sex. \n";
  int i=0;
```

```
for(i=0;i<NUM;i++) //输入信息
{   cout << "Enter title # " << i+1 << ": ";
    getline(cin,names[i]); //获取输入信息
    cout << "Enter sex(0/1) #";
    cin >> sexes[i]; //获取输入信息
    cin.get(); //等待
}
cout << "Thank you. You entered the following: \n" << "name/sex" << endl;
for(i=0;i<NUM;i++) //输出信息
{   cout << names[i] << "\t" << sexes[i] << endl;
}
return;
}
```

例 1-31 的执行效果如图 1-17 所示。



```
Enter title # 1: A
Enter sex<0/1> # 1
Enter title # 2: B
Enter sex<0/1> # 1
Enter title # 3: D
Enter sex<0/1> # 0
Enter title # 4: E
Enter sex<0/1> # 1
Enter title # 5: E
Enter sex<0/1> # 1
Thank you. You entered the following:
name/sex
A      1
B      1
D      0
E      1
E      1
Press any key to continue
```

图 1-17 例 1-31 的执行效果



总结 本小节简要介绍了 STL 模板的一些特性，并以 vector 矢量为例，简要讲述其使用方法。

1.8 小结

本章主要介绍了 C++ STL 的基本概念，介绍了类模板定义、成员模板、友元模板、函数模板以及类模板的参数等内容。

其中，1.7 节主要就 STL 历史、STL 组件、STL 基本结构及 STL 编程概述等四部分内容进行了简要概述。

读者应该重点学习“STL 组件”和“STL 基本结构”两部分内容，还应认真学习“STL 编程概述”的内容。大量的关于编程的内容在后面章节会逐渐深入介绍。

第 2 章

字 符 串

本章讲述C++ STL中的字符串类模板 `string`。在最初的C语言中，头文件 `string.h` 提供了一系列字符串函数。早期的C++也为处理字符串提供了类。`string`类由头文件 `<string>` 支持，该类包含了大量方法及若干构造函数，用于将字符串赋给变量、合并字符串、比较字符串和访问各个元素的重载操作符、查找字符和子字符串的方法等。到目前为止，`string`已被用户广泛接受及使用。

C++从C继承的字符串概念仍然是以‘\0’为结束符的 `char` 数组。C++标准库中的 `string class` 可以将 `string` 作为一个型别，可以实现复制、赋值和比较，不必担心内存大小及占用内存实际长度等具体问题。现今，数据处理大部分是字符串处理，相较于早期的C语言和Fortran语言，这是非常重要的进步。在这些语言中，字符串的处理是非常复杂的。

本章将详细讲述字符串类库简述、字符的特点、字符串类模板 (`basic_string`)，字符串通用函数、字符串联接、字符串IO操作、搜索和查找、字符串对迭代器的支持；字符串对配置器的支持等内容。

2.1 字符串类库简述

本节主要介绍字符串类和智能指针 `auto_ptr`。字符串的表现形式多种多样，如 `TCHAR`，`std::string`、`BSTR`，等。字符串类均起源于C语言的字符串，而最初C语言的字符串是字符的数组。单字节字符串顺序存放各个字符串，并用‘\0’表示字符串的结束。在C语言中，已存在部分字符串处理函数，例如 `strcpy()`，`sprintf()`，`stoi()`等，只能用于单字节字符串。在标准库中，还有仅用于Unicode字符串的函数，如：`wcscpy()`，`swprintf()`，`_wtoi()`等。

多数人都惯于使用指针 `++` 和 `--` 操作符来遍历字符串。使用数组处理字符串中的字符也非常方便。无论ASCII码字符串还是unicode字符串，使用指针均能够正确无误地返回要寻求的字符位置。

STL中只有一个字符串类，即 `basic_string`。类 `basic_string` 实现管理以‘\0’结尾的字符数组。字符类型由模板参数决定。通常，`basic_string` 被处理为不透明的对象，靠获得只读指针来访问缓冲区，写操作是由 `basic_string` 的成员函数实现的。STL的C++标准程序库中的 `string` 类，使用时不必担心内存是否充足、字符串长度等问题。`string` 作为类出现，其集成的操作函数足以完成多数情况下的需要。可以使用“=”进行赋值，使用“==”进行等值比较，使用“+”做串联。

要使用 `string` 类，必须包含头文件 `<string>`。在STL库中，`basic_string` 有两个预定义类

型：包含 char 的 string 类型和包含 wchar 的 wstring 类型。

string 类的 string::npos 可同时定义字符串的最大长度，通常设置为无符号 int 的最大值。String 类包含了 6 个构造函数。string 类支持 cin 方式和 getline() 方式两种输入方式。简单示例如下：

```
string stuff;
cin >> stuff;
getline(cin, stuff);
```

上述三行代码，第一行是声明 string 类的对象 stuff，第二行是从屏幕读入输入的字符串；第三行同样实现第二行代码的功能。

string 库提供了许多其他功能，如删除字符串的部分或全部；用一个字符的部分或全部替换另一个字符串的部分或全部；插入、删除字符串中数据；比较、提取、复制、交换等。

STL 还提供了另一个模板类：auto_ptr 类。该类主要用于管理动态内存分配。如果使用 new() 函数分配堆中的内存，而又不记得回收这部分内存，会导致内存泄漏。因此必须使用 delete 语句释放该内存块。即使在函数末端添加了 delete 语句释放内存，还需要在任何跳出该函数的语句（如抛出异常）之前添加释放内存的处理，例如 goto 语句和 throw 语句。auto_ptr 模板定义了类似指针的对象，将 new 获得的地址赋给该对象。当 auto_ptr 对象过期时，析构函数将使用 delete 来释放内存。如果将 new 返回的地址赋值给 auto_ptr 对象，无须记住还需要释放这些内存。在 auto_ptr 对象过期时，内存将自动被释放。在 C++ 语言中，要使用 STL 中的 auto_ptr 对象，必须包含头文件 <memory>。该文件包括 auto_ptr 模板。使用通常的模板句法来实例化所需类型的指针。auto_ptr 构造函数是显式的，不存在从指针到 auto_ptr 对象的隐式类型转换。

```
auto_ptr <double> pd;
double *p_reg = new double;
pd = p_reg; //不允许
pd = auto_ptr <double> (p_reg); //允许
auto_ptr <double> pauto = p_reg; //不允许
auto_ptr <double> pauto (p_reg); //允许
```

模板可以通过构造函数将 auto_ptr 对象初始化为一个常规指针。auto_ptr 是一个智能指针，但其特性远比指针要多。值得注意的是，在使用 auto_ptr 时，只能配对使用 new 和 delete。



提示 只能对 new 分配的内存使用 auto_ptr 对象，不要对由 new () 分配的或通过声明变量分配的内存使用它。



总结 总结：C++ 库中的 auto_ptr 对象是一种智能指针。智能指针是一种类，即其特征类似于指针。智能指针存储 new 分配的内存地址，也可以被解除引用。智能指针是一个类对象，可以修改和扩充简单指针的行为。智能指针可以建立引用计数，这使得多个对象可共享由智能指针跟踪的同一个值。当使用该值的对象数为 0 时，智能指针将删除该值。智能指针可以提高内存的使用效率，帮助防止内存泄露。

2.2 字符的特点

“字符”本身是个有趣的抽象概念。例如，在纸上或者屏幕上，字符“C”仅仅是一段曲线而已。在计算机中，用一个 8Byte 存储该字符，并赋值 67；字符“C”还是拉丁字母的第三个字母；在化学专业，字符“C”是原子碳的缩写形式；在计算机学科中，字符“C”又被用来表示一种程序设计语言的名字。目前，在计算机领域，字符集合是在字符与整数值之间的一种映射关系。

C++ 程序员通常假定能够使用美国字符集 (ASCII)，但 C++ 允许程序员缺少某些字符的可能性。如果在程序开发过程中，源代码含有 ASCII 里所没有的字符，这是比较麻烦的。许多语言 (例如中文、丹麦文、法文、冰岛文、日文) 无法用 ASCII 中的字符正常写出来。即使扩充到 16 位字符集，也无法将人类所知的所有字符放在同一字符集中。据说已经出现的 32 位字符集能保存每一个字符，但因字符数量庞大，不便于使用，尚未得到推广。

C++ 语言允许程序员使用任何字符集作为字符串的字符，也允许程序员使用扩充字符集或可移植的数值编码。从原则上讲，字符串能以任何 (带有正确的复制操作) 类型作为其字符类型。标准字符串类 `string` 要求其中的字符不能包含用户自定义复制操作，有助于字符串 I/O 的简化与高效率。

字符类型的性质是由字符特征类 (`char_traits`) 定义的。字符特征类是下述模板的特例。

```
template <class Ch> struct char_traits { }
```

所有字符特征类均定义在名称空间 `std` 中，标准的字符特征类由头文件 `<string>` 给出。通用字符串特征类 `char_traits` 本身不具有任何属性，只有针对特定字符类型的专门 `char_traits` 才具有属性。

标准字符串模板的实例类 `basic_string` 依赖于诸多类型和函数。若一个类型作为 `basic_string` 字符类型，必须提供支持上述功能的字符特征类 (`char_traits`)。

C++ 的字符串模板实例化类 `basic_string` 中，还集合了大量的字符串处理函数。这些函数的使用方法将在 2.3 节介绍。

2.3 字符串类模板 (`basic_string`)

标准库字符串功能的基础是 `basic_string`，该类模板提供了许多成员和函数，与标准容器类似。该类模板的声明如下：

```
template <class Ch, class Tr=char_traits<Ch>, class A=allocator<Ch>> class std:: basic_string
{
public:
...
}
```

在上述模板声明中，第一个参数 (`class Ch`) 是说明单个字符 (`Ch`) 所属型别 (`class`)；

第二个参数（`class Tr = char_traits < Ch >`）是特性类别，用以提供字符串类别中的所有字符核心操作。该特性类别规定了“复制字符”或“比较字符”的做法；如果不指定该特性类别，系统会根据现有的字符型别采用默认的特性类别。第三个参数带有默认值（`class A = allocator < Ch >`），用以定义字符串类别所采用的内存模式，通常设定为“默认内存模型 `allocator`”。该模板及其相关功能都定义在名称空间 `std` 中，由头文件 `<string>` 给出。其中包含了两个定义类型，可以为最常用的串类型提供便于使用的名称，即 C++ STL 提供了两个 `basic_string <>` 实例化版本：

```
typedef basic_string <char> string;
typedef basic_string <wchar> wstring;
```

其中，`wstring` 类是为了便于使用宽字符集，例如 `unicode` 或某些欧洲字符集。但所有字符串类型均使用相同接口，其用法和问题是相同的。在本书中，仍以 `string` 表示任何字符串型别。

`basic_string` 和 `vector` 类似，而 `basic_string` 还提供典型的字符串操作，例如子串检索。`basic_string` 没有提供一组完整的操作函数。通常 `string` 不能直接使用数组或者 `vector`，为了更好地支持 `string` 的常见应用，程序员在实现过程中需要尽量减少复制。尤其对于较短的字符串，不应使用自由存储空间，但允许对较长的字符串进行简单修改。

`basic_string <T>` 没有虚函数，这点和其他标准库类型一致。当需要设计更复杂的文字处理类时，可考虑用它加以实现。

与其他标准容器相似，`basic_string` 提供了一组成员类型名，程序员能使用这些与串相关的类型。

例如，

```
typedef Tr traits_type;
typedef typename Tr:: char_type value_type
typedef A allocator_type
...
```

`basic_string` 除支持最简单的 `basic_string <char>` 之外，还支持许多不同种类的字符串，例如，

```
typedef basic_string <unsigned char> Ustring;
typedef basic_string <Jchar> Jstring;           //日文字符串
```

无论如何定义字符串，模板 `basic_string` 的大量函数均可便捷地使用。模板 `basic_string <Ch>` 能够存放集合 `Ch` 中的任何字符，特别是 `string` 中的 ‘0’。“字符类型” `Ch` 的行为必须像字符，但它不能包含用户确定的复制构造函数、析构函数和复制赋值。

虽然字符串类 `string` 包含了诸多的成员和函数，但个别功能没能能够实现，例如正则表达式和较复杂的文本处理功能。

总体而言，`string` 类似的字符串操作逐渐变得简单了。程序员可以定义 `string` 类型的对象、`string` 类的重载操作符和成员函数，这使字符串操作变得非常容易。

2.4 字符串通用操作

在定义 `string` 类对象时，`string` 类自身可以管理内存，程序员不必关注内存的分配细节。

String 类提供的各种操作函数大致分为八类：构造器和析构器、大小和容量、元素存取、字符串比较、字符串修改、字符串接合、I/O 操作以及搜索和查找。本节还将介绍静态数据类型 npos、string 类的迭代器以及配置器等内容。表 2-1 列出了 string 类的所有成员函数。

表 2-1 string 类的所有成员函数

函数名称	效果
构造函数	产生或复制字符串
析构函数	销毁字符串
=, assign	赋以新值
Swap	交换两个字符串的内容
+ =, append(), push_back()	添加字符
insert()	插入字符
erase()	删除字符
clear()	移除全部字符
resize()	改变字符数量
replace()	替换字符
+	串联字符串
==, !=, <, <=, >, >=, compare()	比较字符串内容
size(), length()	返回字符数量
max_size()	返回字符的最大可能个数
empty()	判断字符串是否为空
capacity()	返回重新分配之前的字符容量
reserve()	保留内存以存储一定数量的字符
[], at()	存取单一字符
>>, getline()	从 stream 中读取某值
<<	将值写入 stream
copy()	将内容复制为一个 C - string
c_str()	将内容以 C - string 形式返回
data()	将内容以字符数组形式返回
substr()	返回子字符串
find()	搜寻某子字符串或字符
begin(), end()	提供正向迭代器支持
rbegin(), rend()	提供逆向迭代器支持
get_allocator()	返回配置器

2.4.1 构造器和析构器

构造器函数有四个参数，其中三个具有默认值。要初始化一个 string 类，可以使用 C 风格字符串或 string 类型对象；也可以使用 C 风格字符串的部分或 string 类型对象的部分或序

列。注意：不能使用字符或者整数去初始化字符串。

常见的 string 类构造函数有以下几种形式：

```
string strs //生成空字符串
string s(str) //生成字符串 str 的复制品
string s(str, stridx) //将字符串 str 中始于 stridx 的部分作为构造函数的初值
string s(str, strbegin, strlen) //将字符串 str 中始于 strbegin、长度为 strlen 的部分作为字符串初值
string s(cstr) //以 C_string 类型 cstr 作为字符串 s 的初值
string s (cstr, char_len) //以 C_string 类型 cstr 的前 char_len 个字符串作为字符串 s 的初值
string s (num, c) //生成一个字符串，包含 num 个 c 字符
string s (strs, beg, end) //以区间 [beg, end] 内的字符作为字符串 s 的初值
```

析构函数的形式如下：

```
~string() //销毁所有内存,释放内存
```

如果字符串只包含一个字符，使用构造函数对其初始化时，使用以下两种形式比较合理：

```
std::string s('x'); //错误
std::string s(1, 'x'); //正确
```

或

```
std::string s("x"); //正确
```

 **提示** 上述内容涉及 C_string。这里做简要介绍。C_string 一般被认为是常规的 C++ 字符串。目前，在 C++ 中确实存在一个从 const char * 到 string 的隐式类型转换，却不存在从 string 对象到 C_string 的自动类型转换。对于 string 类型的字符串，可以通过 c_str() 函数返回该 string 类对象对应的 C_string。

通常，程序员在整个程序中应坚持使用 string 类对象，直到必须将内容转化为 char * 时才将其转换为 C_string。

注意：请读者关注中文注释。

例 2-1

```
#include <iostream>
#include <string>
using namespace std;
void main()
{
    string str("12345678");
    char ch[] = "abcdefgh";
    string a; //定义一个空字符串
    string str_1 (str); //构造函数，全部复制
    string str_2 (str, 2, 5); //构造函数，从字符串 str 的第 2 个元素开始，复制 5 个元素，赋值给 str_2
    string str_3 (ch, 5); //将字符串 ch 的前 5 个元素赋值给 str_3
    string str_4 (5, 'X'); //将 5 个 'X' 组成的字符串 "XXXXX" 赋值给 str_4
    string str_5 (str.begin(), str.end()); //复制字符串 str 的所有元素，并赋值给 str_5
```

```
cout << str << endl;  
cout << a << endl;  
cout << str_1 << endl;  
cout << str_2 << endl;  
cout << str_3 << endl;  
cout << str_4 << endl;  
cout << str_5 << endl;  
}
```

例 2-1 的执行效果如图 2-1 所示。



图 2-1 例 2-1 的执行效果



提示 使用 cout 输出 string 类型对象 a 时，输出为空。这是因为没有给 string 类型对象 a 赋值。

通过上述内容的学习，读者应对 string 类的构造函数和析构函数有了初步了解，并能够使用构造函数创建 string 类型对象。

2.4.2 大小和容量

String 类型对象包括三种求解大小的函数：size() 和 length()；maxsize()；capacity()。

1) size() 和 length()。这两个函数会返回 string 类型对象中的字符个数，且它们的执行效果相同。

2) max_size()。max_size() 函数返回 string 类型对象最多包含的字符数。一旦程序使用长度超过 max_size() 的 string 操作，编译器会抛出 length_error 异常。

3) capacity()。该函数返回在重新分配内存之前，string 类型对象所能包含的最大字符数。

string 类型对象还包括一个 reserve() 函数。调用该函数可以为 string 类型对象重新分配内存。重新分配的大小由其参数决定。reserve() 的默认参数为 0。

下面以例 2-2 说明上述几个函数的使用方法。

例 2-2

```
#include <iostream>  
#include <string>  
using namespace std;
```

```
void main()
{
    int size=0;
    int length=0;
    unsigned long maxsize=0;
    int capacity=0;
    string str("12345678");
    string str_custom;
    str_custom.reserve(5);
    str_custom=str;
    size=str_custom.size();
    length=str_custom.length();
    maxsize=str_custom.max_size();
    capacity=str_custom.capacity();
    cout<<" size = " <<size<<endl;
    cout<<" length = " <<length<<endl;
    cout<<" maxsize = " <<maxsize<<endl;
    cout<<" capacity = " <<capacity<<endl;
}
```

例 2-2 的执行效果如图 2-2 所示。



图 2-2 例 2-2 的执行效果

由例 2-2 可知，string 类型对象 str_custom 调用 reserve() 函数时，似乎并没有起到重新分配内存的目的（笔者所用编译器为 Visual C++ 6.0）。

修改上述代码，删除语句 str_custom.reserve(5)，在代码 str_custom=str 之后如下添加代码：

```
str_custom.resize(5);
```

修改后程序的执行效果如图 2-3 所示。



图 2-3 例 2-2 修改后程序的执行效果

重新设置 string 类型对象 str_custom 的大小之后，重新求解 str_custom 的大小，由图 2-3 可知，其执行效果与设置的数值一致（均为 5）。

2.4.3 元素存取 (访问)

在通常情况下, string 是C++ 中的字符串。字符串是一种特殊类型的容器, 专门用来操作字符序列。字符串中元素的访问是允许的。一般可使用两种方法访问字符串中的单一字符: 下标操作符 [] 和成员函数 at()。两者均返回指定的下标位置的字符。第 1 个字符索引 (下标) 为 0, 最后的字符索引为 length() - 1。需要注意的是, 这两种访问方法是有区别的。

提示 ① 下标操作符 [] 在使用时不检查索引的有效性。如果下标超出字符串的长度范围, 会导致未定义行为; 对于常量字符串, 使用下标操作符时, 字符串的最后字符 (即 ‘\0’) 是有效的。对应 string 类型对象 (常量型) 最后一个字符的下标是有效的, 调用返回字符 ‘\0’。

② 函数 at () 在使用时会检查下标是否有效。如果给定的下标超出字符串的长度范围, 系统会抛出 out_of_range 异常。

例 2-3

```
#include <iostream>
#include <string>
void main()
{
    const std::string cS("conststring");
    std::string s("abcde");
    char temp=0;
    char temp_1=0;
    char temp_2=0;
    char temp_3=0;
    char temp_4=0;
    char temp_5=0;
    temp=s[2]; //获取字符 'c'
    temp_1=s.at(2); //获取字符 'c'
    temp_2=s[s.length()]; //未定义行为, 返回字符 '\0', 但 Visual C++2012 执行时未报错
    temp_3=cS[cS.length()]; //指向字符 '\0'
    temp_4=s.at(s.length()); //程序异常
    temp_5=cS.at(cS.length()); //程序异常
    std::cout<<temp<<temp_1<<temp_2<<temp_3<<temp_4<<temp_5<<std::endl;
}
```

通过对上述代码的分析可知, 要理解字符串的存取需要多实践、多尝试, 并且要牢记基础知识和基本规则。

为修改 string 字符串的内容, 下标操作符 [] 和函数 at() 均返回字符的“引用”。但当字符串的内存被重新分配以后, 可能会发生执行错误。

例 2-4

```
#include <iostream>
#include <string>
```

```

void main()
{
    std::string s("abcde");
    std::cout << s << std::endl;
    char& r = s[2];           //建立引用关系
    char* p = &s[3];         //建立引用关系
    r = 'X';                  //修改内容
    *p = 'Y';                 //修改内容
    std::cout << s << std::endl; //输出
    s = "12345678";          //重新赋值
    r = 'X';                  //修改内容
    *p = 'Y';                 //修改内容
    std::cout << s << std::endl; //输出
}

```

例 2-4 的执行效果如图 2-4 所示。



图 2-4 例 2-4 的执行效果

在例 2-4 中，使用 Visual C++ 2012 编译器编译，字符串被重新赋值后，修改其中某位置字符的值，仍然成功。这与前面所述的“可能会发生执行错误”其实并不矛盾。因为，从意义上讲，字符串被重新赋值后，只是其原来的引用关系已经没有意义了。

2.4.4 字符串比较

字符串可以和类型相同的字符串相比较，也可以和具有同样字符类型的数组比较。Basic_string 类模板既提供了 >、<、==、>=、<=、!= 等比较运算符；还提供了 compare() 函数。其中 compare() 函数支持多参数处理，支持用索引值和长度定位子串进行比较。该函数返回一个整数来表示比较结果。如果相比较的两个子串相同，compare() 函数返回 0，否则返回非零值。

1) compare() 函数。类 basic_string 的成员函数 compare() 的原型如下：

```

int compare (const basic_string& s) const;
int compare (const Ch* p) const;
int compare (size_type pos, size_type n, const basic_string& s) const;
int compare (size_type pos, size_type n, const basic_string& s, size_type pos2, size_type n2) const;
int compare (size_type pos, size_type n, const Ch* p, size_type n2 = npos) const;

```

如果在使用 compare() 函数时，参数中出现了位置和大小，比较时只能用指定的子串。例如：

```
s.compare(pos, n, s2);
```

若参与比较的两个串值相同, 则函数返回 0; 若字符串 *s* 按字典顺序要先于 *s2*, 则返回负值; 反之, 则返回正值。下面举例说明如何使用 `string` 类的 `compare()` 函数。请注意代码中的中文注释语句。

例 2-5

```
#include <iostream>
#include <string>
using namespace std;
void main()
{
    string A("aBcdef");
    string B("AbcdEf");
    string C("123456");
    string D("123dfg");
    //下面是各种比较方法
    int m=A.compare(B);           //完整的 A 和 B 的比较
    int n=A.compare(1,5,B);       //"Bcdef"和"AbcdEf"比较
    int p=A.compare(1,5,B,4,2);   //"Bcdef"和"Ef"比较
    int q=C.compare(0,3,D,0,3);   //"123"和"123"比较
    cout << "m=" <<m << ", n=" <<n << ", p=" <<p << ", q=" <<q << endl;
    cin.get();
    return;
}
```

例 2-5 的执行效果如图 2-5 所示。

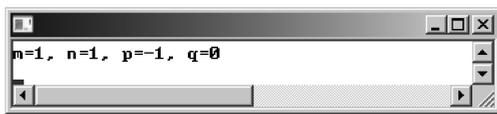


图 2-5 例 2-5 的执行效果



总结 由例 2-5 可知, `string` 类的比较 `compare()` 函数使用非常方便, 而且能区分字母的大小写。建议读者多使用此函数。而比较运算符使用起来更加方便, 在后面介绍。

2) 比较运算符。`String` 类的常见运算符包括 `>`、`<`、`==`、`>=`、`<=`、`!=`。其意义分别为“大于”“小于”“等于”“大于等于”“小于等于”“不等于”。

比较运算符使用起来非常方便, 此处不再介绍其函数原型, 读者直接使用即可。下面以例 2-6 进行说明。

例 2-6

```
#include <iostream>
#include <string>
using namespace std;
void TrueOrFalse(int x)
```

```
{
    cout << (x?"True":"False") << endl;
}
void main()
{
    string S1 = "DEF";
    string CP1 = "ABC";
    string CP2 = "DEF";
    string CP3 = "DEFG";
    string CP4 = "def";
    cout << "S1 = " << S1 << endl;
    cout << "CP1 = " << CP1 << endl;
    cout << "CP2 = " << CP2 << endl;
    cout << "CP3 = " << CP3 << endl;
    cout << "CP4 = " << CP4 << endl;
    cout << "S1 <= CP1 returned ";
    TrueOrFalse(S1 <= CP1);
    cout << "S1 <= CP2 returned ";
    TrueOrFalse(S1 <= CP2);
    cout << "S1 <= CP3 returned ";
    TrueOrFalse(S1 <= CP3);
    cout << "CP1 <= S1 returned ";
    TrueOrFalse(CP1 <= S1);
    cout << "CP2 <= S1 returned ";
    TrueOrFalse(CP2 <= S1);
    cout << "CP4 <= S1 returned ";
    TrueOrFalse(CP4 <= S1);
    cin.get();
}
```

例 2-6 的执行结果为:

S1 = DEF

CP1 = ABC

CP2 = DEF

CP3 = DEFG

CP4 = def

S1 <= CP1 returned False

S1 <= CP2 returned True

S1 <= CP3 returned True

CP1 <= S1 returned True

CP2 <= S1 returned True

CP4 <= S1 returned False



总结 由前述内容可知,使用比较运算符可以非常容易地实现字符串的大小比较。在使用时比较运算符时,读者应注意:对于参加比较的两个字符串,任一个字符串均不能为 NULL,否则程序会异常退出。

2.4.5 字符串内容的修改和替换

字符串内容的变化包括修改和替换两种。本节将分别讲解字符串内容的修改和字符串内容的替换。

1. 字符串内容的修改

可以通过使用多个函数修改字符串的值。例如 `assign()`, `operator =`, `erase()`, 交换 (`swap`), 插入 (`insert`) 等。另外,还可通过 `append()` 函数添加字符。下面逐一介绍各成员函数的使用方法。

(1) `assign()` 函数

使用 `assign()` 函数可以直接给字符串赋值。该函数既可以将整个字符串赋值给新串,也可以将字符串的子串赋值给新串。其在 `basic_string` 中的原型为:

```
basic_string& assign (const E *s); //直接使用字符串赋值
basic_string& assign (const E *s, size_type n);
basic_string& assign (const basic_string& str, size_type pos, size_type n); //将 str 的子串赋值给调用串
basic_string& assign (const basic_string& str); //使用字符串的“引用”赋值
basic_string& assign (size_type n, E c); //使用 n 个重复字符赋值
basic_string& assign (const_iterator first, const_iterator last); //使用迭代器赋值
```

以上几种方法在例 2-7 中均有所体现。请读者参考下述代码。

例 2-7

```
#include <iostream>
#include <string>
using namespace std;
void main()
{
    string str1("123456");
    string str;
    str.assign(str1); //直接赋值
    cout << str << endl;
    str.assign(str1,3,3); //赋值给子串
    cout << str << endl;
    str.assign(str1,2,str1.npos); //赋值给从位置 2 至末尾的子串
    cout << str << endl;
    str.assign(5,'X'); //重复 5 个 'X' 字符
    cout << str << endl;
    string::iterator itB;
    string::iterator itE;
```

```

itB = str1.begin();
itE = str1.end();
str.assign(itB, (- - itE)); //从第 1 个至倒数第 2 个元素,赋值给字符串 str
cout << str << endl;
...
}

```

(2) operator = 函数

operator = 的功能就是赋值。

(3) erase() 函数

erase() 函数的原型为:

```

iterator erase(iterator first, iterator last);
iterator erase(iterator it);
basic_string& erase (size_type p0 = 0, size_type n = npos);

```

erase() 函数的使用方法为:

```

str.erase(str.begin(), str.end());
或 str.erase(3);

```

(4) swap() 函数

swap() 函数的原型为:

```

void swap(basic_string& str);

```

swap() 函数的使用方法为:

```

string str2("abcdefghijklmn");
str.swap(str2);

```

(5) insert() 函数

insert() 函数的原型为:

```

basic_string& insert (size_type p0, const E *s); //插入 1 个字符至字符串 s 前面
basic_string& insert (size_type p0, const E *s, size_type n); //将 s 的前 3 个字符插入 p0 位置
basic_string& insert (size_type p0, const basic_string& str);
basic_string& insert (size_type p0, const basic_string& str,
size_type pos, size_type n); //选取 str 的子串
basic_string& insert (size_type p0, size_type n, E c); //在下标 p0 位置插入 n 个字符 c
iterator insert (iterator it, E c); //在 it 位置插入字符 c
void insert (iterator it, const_iterator first, const_iterator last); //在字符串前插入字符
void insert (iterator it, size_type n, E c); //在 it 位置重复插入 n 个字符 c

```

insert() 函数的使用方法为:

```

string A("ello");
string B("H");
B.insert(1,A);
cout << B << endl;
A = "ello";
B = "H";

```

```

B.insert(1, "yanchy ", 3);
cout << B << endl;
A = "ello";
B = "H";
B.insert(1, A, 2, 2);
cout << B << endl;
A = "ello";
B = "H";
B.insert(1, 5, 'C');
cout << B << endl;
A = "ello";
B = "H";
string::iterator it = B.begin() + 1;
const string::iterator itF = A.begin();
const string::iterator itG = A.end();
B.insert(it, itF, itG);
cout << B << endl;

```

(6) append() 函数

append() 函数的原型为:

```

basic_string& append (const E *s); //在原始字符串后面追加字符串 s
basic_string& append (const E *s, size_type n); //在原始字符串后面追加字符串 s 的前 n 个字符
basic_string& append (const basic_string& str, size_type pos, size_type n);
//在原始字符串后面追加字符//串 s 的子串 s [pos, ..., pos + n - 1]
basic_string& append (const basic_string& str);
basic_string& append (size_type n, E c); //追加 n 个重复字符
basic_string& append (const_iterator first, const_iterator last); //使用迭代器追加

```

append() 函数的使用方法为:

```

A = "ello";
B = "H";
cout << A << endl;
cout << B << endl;
B.append(A);
cout << B << endl;
A = "ello";
B = "H";
cout << A << endl;
cout << B << endl;
B.append("12345", 2);
cout << B << endl;
A = "ello";
B = "H";
cout << A << endl;
cout << B << endl;

```

```
B.append("12345",2,3);
cout << B << endl;
A = "ello";
B = "H";
cout << A << endl;
cout << B << endl;
B.append(10, 'a');
cout << B << endl;
A = "ello";
B = "H";
cout << A << endl;
cout << B << endl;
B.append(A.begin(), A.end());
cout << B << endl;
```

下面通过例 2-8 介绍这些函数的使用。完整代码如下：

例 2-8

```
#include <iostream>
#include <string>
using namespace std;
void main()
{
    string str1("123456");
    string str2("abcdefghijklmn");
    string str;
    str.assign(str1);
    cout << str << endl;
    str.assign(str1,3,3);
    cout << str << endl;
    str.assign(str1,2, str1.npos);
    cout << str << endl;
    str.assign(5, 'X');
    cout << str << endl;
    string::iterator itB;
    string::iterator itE;
    itB = str1.begin();
    itE = str1.end();
    str.assign(itB, ( -- itE));
    cout << str << endl;
    str = str1;
    cout << str << endl;
    str.erase(3);
    cout << str << endl;
    str.erase(str.begin(), str.end());
    cout << "." << str << "." << endl;
```

```
str.swap(str2);
cout << str << endl;
string A("ello");
string B("H");
B.insert(1,A);
cout << B << endl;
A="ello";
B="H";
B.insert(1,"yanchy ",3);
cout << "插入: " << B << endl;
A="ello";
B="H";
B.insert(1,A,2,2);
cout << "插入: " << B << endl;
A="ello";
B="H";
B.insert(1,5,'C');
cout << "插入: " << B << endl;
A="ello";
B="H";
string::iterator it=B.begin()+1;
const string::iterator itF=A.begin();
const string::iterator itG=A.end();
B.insert(it,itF,itG);
cout << "插入:" << B << endl;
A="ello";
B="H";
cout << "A= " << A << ", B= " << B << endl;
B.append(A);
cout << "追加:" << B << endl;
A="ello";
B="H";
cout << "A= " << A << ", B= " << B << endl;
B.append("12345",2);
cout << "追加:" << B << endl;
A="ello";
B="H";
cout << "A= " << A << ", B= " << B << endl;
B.append("12345",2,3);
cout << "追加:" << B << endl;
A="ello";
B="H";
cout << "A= " << A << ", B= " << B << endl;
B.append(10,'a');
cout << "追加:" << B << endl;
A="ello";
```

```

B = "H";
cout << "A = " << A << ", B = " << B << endl;
B.append(A.begin(), A.end());
cout << "追加:" << B << endl;
cin.get();
}

```

例 2-8 执行效果如图 2-6 所示。读者可根据源代码逐项对照。

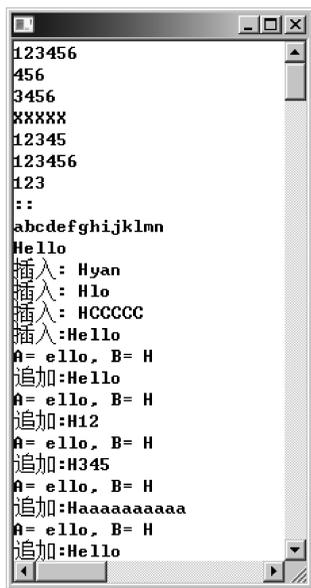


图 2-6 例 2-8 的执行效果

2. 字符串内容的替换

如果在一个字符串中标识出具体位置，便可以通过下标操作修改指定位置字符的值，或者替换某个子串。完成此项操作需要使用 `string` 类的成员函数 `replace()`。该函数的原型如下：

```

1) basic_string& replace (size_type p0, size_type n0, const E *s);
// 使用字符串 s 中的 n 个字符，从源串的位置 p0 处开始替换
2) basic_string& replace (size_type p0, size_type n0, const E *s, size_type n);
// 使用字符串 s 中的 n 个字符，从源串的位置 p0 处开始替换 1 个字符
3) basic_string& replace (size_type p0, size_type n0, const basic_string& str);
// 同 1)
4) basic_string& replace (size_type p0, size_type n0, const basic_string& str, size_type pos,
size_type n);
// 使用 str 的子串 str [pos, pos+n-1] 替换源串中的内容，从位置 p0 处开始替换，替换字符 n0 个
5) basic_string& replace (size_type p0, size_type n0, size_type n, E c);
// 使用 n 个字符 'c' 替换源串中位置 p0 处开始的 n0 个字符
6) basic_string& replace (iterator first0, iterator last0, const E *s);
// 使用迭代器替换，和 1) 用法类似

```

```

7) basic_string& replace (iterator first0, iterator last0, const E*s, size_type n);
// 和 2) 类似
8) basic_string& replace (iterator first0, iterator last0, const basic_string& str);
// 和 3) 类似
9) basic_string& replace (iterator first0, iterator last0, size_type n, E c);
// 和 5) 类似
10) basic_string& replace (iterator first0, iterator last0, const_iterator first, const_iterator last);
// 使用迭代器替换

```

该函数的使用方法参见例 2-9。

例 2-9

```

#include <iostream>
#include <string>
using namespace std;
void main()
{
    string var("abcdefghijklmn");
    const string dest("1234");
    string dest2("567891234");
    var.replace(3,3,dest);
    cout << "1: " << var << endl;
    var = "abcdefghijklmn";
    var.replace(3,1,dest.c_str(), 1, 3);
    cout << " 2: " << var << endl;
    var = " abcdefghijklmn";
    var.replace(3, 1, 5, 'x');
    cout << " 3: " << var << endl;
    string::iterator itA, itB;
    string::iterator itC, itD;
    itA = var.begin();
    itB = var.end();
    var = " abcdefghijklmn";
    var.replace(itA, itB, dest);
    cout << " 4: " << var << endl;
    itA = var.begin();
    itB = var.end();
    itC = dest2.begin() + 1;
    itD = dest2.end();
    var = " abcdefghijklmn";
    var.replace(itA, itB, itC, itD);
    cout << " 5: " << var << endl;
    var = " abcdefghijklmn";
    var.replace(3, 1, dest.c_str(), 4); //这种方式会限定字符串替换的最大长度
    cout << " 6: " << var << endl;
}

```

例 2-9 的执行效果如图 2-7 所示。



图 2-7 例 2-9 的执行效果

总结 本节讲述了诸多可进行字符串内容的修改和替换的函数及其使用方法，并给出了例题。由于每个函数可能有多个原型，希望读者根据自己的情况，掌握其中的一种或两种，以满足自己使用的需要。同时，希望读者能够对照例题的执行效果，认真阅读本章节中的源代码，彻底掌握本节内容。

2.4.6 字符串联接

字符串拼接即将两个子串联接起来，将其中一个串放在另一个串之后，并形成一个新串。在 C++ STL 中，可以使用“+”将两个字符串拼接起来。例如：

```
string str1 ("Inter");
string str2 ("national");
string str3 = str1 + str2;
cout << str3 << endl;
```

上述代码的输出结果为：

```
International
```

2.4.7 字符串 I/O 操作

“<<”和“>>”提供了 C++ 语言的字符串输入和字符串输出功能。“<<”可以将字符读入一个流中（例如 ostream）；“>>”可以实现将以空格或回车为“结束符”的字符序列读入到对应的字符串中，并且开头和结尾的空白字符不包括进字符串中。

还有一个常用的 getline() 函数，该函数的原型包括两种形式：

```
template <class CharType, class Traits, class Allocator > basic_istream<CharType, Traits> &
getline (basic_istream<CharType, Traits> & _Istr, basic_string<CharType, Traits, Allocator>
& _Str);
//上述原型包含 2 个参数：第 1 个参数是输入流；第 2 个参数是保存输入内容的字符串
template <class CharType, class Traits, class Allocator > basic_istream<CharType, Traits> &
getline (basic_istream<CharType, Traits> & _Istr, basic_string<CharType, Traits, Allocator> &
_Str,
CharType _Delim);
//上述原型包含 3 个参数：第 1 个参数是输入流，第 2 个参数保存输入的字符串，第 3 个参数指定分界符。
```

该函数可将整行的所有字符读到字符串中。在读取字符时，遇到文件结束符、分界符、

回车符时，将终止读入操作，且文件结束符、分界符、回车符在字符串中不会保存；当已读入的字符数目超过字符串所能容纳的最大字符数时，将会终止读入操作。下面分别按上述两种函数原型举例说明，参见例 2-10。

例 2-10

```
#include <iostream>
#include <string>
using namespace std;
void main()
{
    string s1,s2;
    getline(cin,s1);
    getline(cin, s2, ' ');
    cout << "You inputed chars are: " << s1 << endl;
    cout << "You inputed chars are: " << s2 << endl;
}
```

例 2-10 的执行效果如图 2-8 所示。

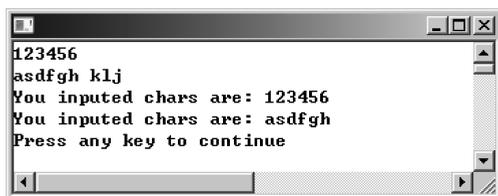


图 2-8 例 2-10 的执行效果



提示 如图 2-8 所示，输入的第二行字符中间包含空格字符，而空格之后的字符没有被存储到字符串 s2 中。

2.4.8 字符串查找

在 C 语言和 C++ 语言中，可用于实现字符串查找功能的函数非常多。在 STL 中，字符串的查找功能可以实现搜索单个字符、搜索子串；可以实现前向搜索、后向搜索；可以分别实现搜索第一个和最后一个满足条件的字符（或子串）实现搜索第一个和最后一个与给定参考值不相等的字符（或子串）。若查找 find() 函数和其他函数没有搜索到期望的字符（或子串），则返回 npos；若搜索成功，则返回搜索到的第 1 个字符或子串的位置。



提示 npos 是一个无符号整数值，初始值为 -1。当搜索失败时，npos 表示“没有找到 (not found)”或“所有剩余字符”。

值得注意的是，所有查找 find() 函数的返回值均是 size_type 类型，即无符号整数类型。该返回值用于表明字符串中元素的个数或者字符在字符串中的位置。

下面分别介绍和字符查找相关的函数。

1. find() 函数和 rfind()

find() 函数的原型主要有以下 4 种：

```
size_type find (value_type _Ch, size_type _Off = 0) const;
//find()函数的第1个参数是被搜索的字符,第2个参数是在源串中开始搜索的下标位置
size_type find (const value_type* _Ptr, size_type _Off = 0) const;
//find()函数的第1个参数是被搜索的字符串,第2个参数是在源串中开始搜索的下标位置
size_type find (const value_type* _Ptr, size_type _Off = 0, size_type _Count) const;
//第1个参数是被搜索的字符串,第2个参数是源串中开始搜索的下标,第3个参数是关于第1个
//参数的字符个数,可能是_Ptr的所有字符数,也可能是_Ptr的子串字符个数
size_type find (const basic_string& _Str, size_type _Off = 0) const;
//第1个参数是被搜索的字符串,第2个参数是在源串中开始搜索的下标位置
```

rfind()函数的原型和find()函数的原型类似,参数情况也类似。只不过 rfind()函数适用于实现逆向查找。

find()函数和 rfind()函数的使用方法参见例 2-11。

例 2-11

```
#include <iostream>
#include <string>
using namespace std;
void main()
{
    string str_ch (" for");
    string str (" Hi, Peter, I'm sick. Please bought some drugs for me.");
    string::size_type m= str.find ('P', 5);
    string::size_type rm= str.rfind ('P', 5);
    cout<<" Example - find(): The position (forward) of 'P' is: " <<(int) m<<endl;
    cout<<" Example - rfind(): The position (reverse) of 'P' is: " <<(int) rm<<endl;
    string::size_type n= str.find (" some", 0);
    string::size_type rn= str.rfind (" some", 0);
    cout<<" Example - find(): The position (forward) of 'some' is: " <<(int) n<<endl;
    cout<<" Example - rfind(): The position (reverse) of 'some' is: " <<(int) rn<<endl;
    string::size_type mo= str.find (" drugs", 0, 5);
    string::size_type rmo= str.rfind (" drugs", 0, 5);
    cout<<" Example - find(): The position (forward) of 'drugs' is: " <<(int) mo<<endl;
    cout<<" Example - rfind(): The position (reverse) of 'drugs' is: " <<(int) rmo<<endl;
    string::size_type no= str.find (str_ch, 0);
    string::size_type rno= str.rfind (str_ch, 0);
    cout<<" Example - find(): The position of 'for' is: " <<(int) no<<endl;
    cout<<" Example - rfind(): The position of 'for' is: " <<(int) rno<<endl;
    cin.get();
}
```

例 2-11 的执行效果如图 2-9 所示。

2. find_first_of()函数和 find_last_of()函数

find_first_of()函数可实现在源串中搜索某字符串的功能,该函数的返回值是被搜索字符串的第 1 个字符第 1 次出现的下标(位置)。若查找失败,则返回 npos。

```

Example-rfind(): The position(reverse) of 'P' is: 4
Example-find(): The position(forward) of 'some' is: 35
Example-rfind(): The position(reverse) of 'some' is: -1
Example-find(): The position(forward) of 'drugs' is: 40
Example-rfind(): The position(reverse) of 'drugs' is: -1
Example-find(): The position of 'for' is: 46
Example-rfind(): The position of 'for' is: -1

```

图 2-9 例 2-11 的执行效果

`find_last_of()` 函数同样可实现在源串中搜索某字符串的功能。与 `find_first_of()` 函数所不同的是, 该函数的返回值是被搜索字符串的最后 1 个字符的下标 (位置)。若查找失败, 则返回 `npos`。

上述两个函数的原型分别为:

```

size_type find_first_not_of (value_type_Ch, size_type_Off = 0) const;
size_type find_first_of (const value_type* _Ptr, size_type_Off = 0) const;
size_type find_first_of (const value_type* _Ptr, size_type_Off, size_type_Count) const;
size_type find_first_of (const basic_string& _Str, size_type_Off = 0) const;

```

```

size_type find_last_of (value_type_Ch, size_type_Off = npos) const;
size_type find_last_of (const value_type* _Ptr, size_type_Off = npos) const;
size_type find_last_of (const value_type* _Ptr, size_type_Off, size_type_Count) const;
size_type find_last_of (const basic_string& _Str, size_type_Off = npos) const;

```

例 2-12 详细阐述了 `find_first_of()` 函数和 `find_last_of()` 函数的使用方法。这两个函数和 `find()` 函数及 `rfind()` 函数的使用方法相同, 具体参数的意义亦相同。

例 2-12

```

#include <iostream>
#include <string>
using namespace std;
void main()
{
    string str_ch (" for");
    string str (" Hi, Peter, I'm sick. Please bought some drugs for me. ");
    int length = str.length();
    string::size_type m = str.find_first_of ('P', 0);
    string::size_type rm = str.find_last_of ('P', (length-1));
    cout << " Example - find_first_of(): The position (forward) of 'P' is: " << (int) m << endl;
    cout << " Example - find_last_of(): The position (reverse) of 'P' is: " << (int) rm << endl;
    string::size_type n = str.find_first_of (" some", 0);
    string::size_type rn = str.find_last_of (" some", (length-1));
    cout << " Example - find_first_of(): The position (forward) of 'some' is: " << (int) n << endl;
    cout << " Example - find_last_of(): The position (reverse) of 'some' is: " << (int) rn << endl;
    string::size_type mo = str.find_first_of (" drugs", 0, 5);

```

```

string::size_type rmo = str.find_last_of (" drugs", (length-1), 5);
cout << " Example - find_first_of(): The position (forward) of 'drugs' is: " << (int) mo <<
endl;
cout << " Example - find_last_of(): The position (reverse) of 'drugs' is: " << (int) rmo <<
endl;
string:: size_type no = str.find_first_of (str_ch, 0);
string:: size_type rno = str.find_last_of (str_ch, (length-1));
cout << " Example - find_first_of(): The position of 'for' is: " << (int) no << endl;
cout << " Example - find_last_of(): The position of 'for' is: " << (int) rno << endl;
cin.get();
}

```

例 2-12 的执行效果如图 2-10 所示。

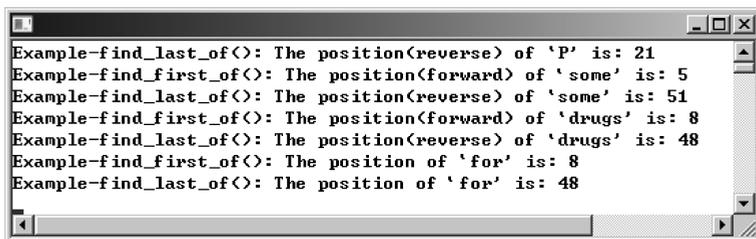


图 2-10 例 2-12 的执行效果

3. find_first_not_of () 函数和 find_last_not_of () 函数

find_first_not_of () 函数的函数原型为:

```

size_type find_first_not_of (value_type _Ch, size_type _Off = 0) const;
size_type find_first_not_of (const value_type* _Ptr, size_type _Off = 0) const;
size_type find_first_not_of (const value_type* _Ptr, size_type _Off, size_type _Count) const;
size_type find_first_not_of (const basic_string& _Str, size_type _Off = 0) const;

```

find_first_not_of () 函数可实现在源字符串中搜索与指定字符 (串) 不相等的第 1 个字符; find_last_not_of () 函数可实现在源字符串中搜索与指定字符 (串) 不相等的最后 1 个字符。这两个函数的参数意义和前面几个函数相同, 它们的使用方法和前面几个函数也基本相同。详见例 2-13。

例 2-13

```

#include <iostream>
#include <string>
using namespace std;
void main()
{
    string str_ch (" for");
    string str (" Hi, Peter, I'm sick. Please bought some drugs for me. ");
    int length = str.length();
    string:: size_type m = str.find_first_not_of ( 'P', 0);

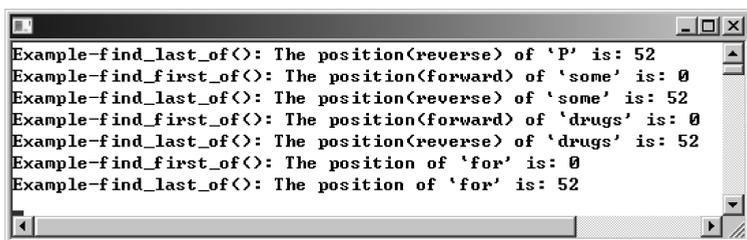
```

```

string::size_type rm = str.find_last_not_of ('P', (length-1));
cout << " Example -find_first_of(): The position (forward) of 'P' is: " << (int) m << endl;
cout << " Example -find_last_of(): The position (reverse) of 'P' is: " << (int) rm << endl;
string:: size_type n = str.find_first_not_of (" some", 0);
string:: size_type rn = str.find_last_not_of("some", (length-1));
cout << "Example -find_first_of(): The position (forward) of 'some' is: " << (int) n << endl;
cout << " Example -find_last_of(): The position (reverse) of 'some' is: " << (int) rn << endl;
string:: size_type mo = str.find_first_not_of (" drugs", 0, 5);
string:: size_type rmo = str.find_last_not_of (" drugs", (length-1), 5);
cout << " Example -find_first_of(): The position (forward) of 'drugs' is: " << (int) mo << endl;
cout << " Example -find_last_of(): The position (reverse) of 'drugs' is: " << (int) rmo << endl;
string:: size_type no = str.find_first_not_of (str_ch, 0);
string:: size_type rno = str.find_last_not_of (str_ch, (length-1));
cout << " Example -find_first_of(): The position of 'for' is: " << (int) no << endl;
cout << " Example -find_last_of(): The position of 'for' is: " << (int) rno << endl;
cin.get();
}

```

例 2-13 的执行效果如图 2-11 所示。



```

Example-find_last_of(): The position(reverse) of 'P' is: 52
Example-find_first_of(): The position(forward) of 'some' is: 0
Example-find_last_of(): The position(reverse) of 'some' is: 52
Example-find_first_of(): The position(forward) of 'drugs' is: 0
Example-find_last_of(): The position(reverse) of 'drugs' is: 52
Example-find_first_of(): The position of 'for' is: 0
Example-find_last_of(): The position of 'for' is: 52

```

图 2-11 例 2-13 的执行效果

总结 本小节主要讲述 C++ STL 中的字符串查找函数。对于所述的 6 个查找函数，它们的使用形式大致相同，对于每个函数均配备了实例作为参考。请读者能认真对照例题，深刻理解这 6 个函数的使用方法，仔细体会函数每个参数的意义。

2.4.9 字符串对迭代器的支持

提示 这是本书首次介绍迭代器。后面章节还会专门讲述迭代器的相关内容。通过学习本章的内容，读者应对迭代器有个大致了解。

理解迭代器是理解 STL 的关键所在。模板使得算法独立于存储的数据类型，而迭代器使得算法独立于使用的容器类型。STL 定义了 5 种迭代器，根据所需的迭代器类型对算法进行描述。这 5 种迭代器分别是：输入迭代器、输出迭代器、正向迭代器、双向迭代器和随机访问迭代器。对于这 5 种迭代器不仅可以执行解除引用操作（* 操作符），还可进行比较。本节主要讲述 `basic_string`（或 `string` 类）中迭代器的使用。

`basic_string` 和 `string` 类均提供了常规的迭代器和反向迭代器。`string` 是字符的有序群集。

C++ 标准库为 `string` 提供了相应接口，便于将字符串作为 STL 容器使用。可以使用迭代器遍历 `string` 内的所有字符；也可以使用其他算法遍历 `string` 内的所有字符。而且能够访问字符串中的每个字符，并对这些字符进行排序、逆向重排、找出最大值等操作。

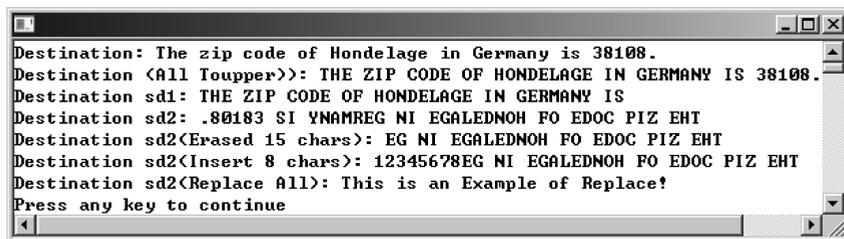
`string` 类的迭代器是随机存取迭代器，即支持随机存取。任何一个 STL 算法都可与其搭配使用。通常 `string` 的“迭代器型别”由 `string` class 本身定义，通常可以被简单地定义为一般指针。对迭代器而言，如果发生重新分配，或其所指的值发生某些变化时，迭代器会失效。

`string` 类中和使用迭代器相关的成员函数是很多的，主要包括 `begin()`、`end()`、`rbegin()`、`rend()`、`append()`、`assign()`、`insert()`、`erase()`、`replace()` 等。下面通过例 2-14 说明迭代器在 `string` 类中的使用方法。

例 2-14

```
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;
void main()
{
    string s("The zip code of Hondelage in Germany is 38108. ");
    cout << "Original: " << s << endl;
    string sd(s.begin(), s.end()); //构造函数中使用迭代器
    cout << "Destination: " << sd << endl;
    transform(sd.begin(), sd.end(), sd.begin(), toupper); //算法中使用迭代器(仿函数)
    cout << "Destination (All Toupper): " << sd << endl;
    string sd1;
    sd1.append(sd.begin(), (sd.end() - 7)); //append()函数中使用迭代器
    cout << "Destination sd1: " << sd1 << endl;
    string sd2;
    string::reverse_iterator iterA;
    string temp = "0";
    for (iterA = sd.rbegin(); iterA != sd.rend(); iterA++) //reverse_iterator
    {
        temp = *iterA;
        sd2.append(temp);
    }
    cout << " Destination sd2: " << sd2 << endl;
    sd2.erase(0, 15); //erase()函数中使用迭代器
    cout << " Destination sd2 (Erased 15 chars): " << sd2 << endl;
    string::iterator iterB = sd2.begin();
    string sd3 = string("12345678");
    sd2.insert(sd2.begin(), sd3.begin(), sd3.end()); //insert()函数中使用迭代器
    cout << " Destination sd2 (Insert 8 chars): " << sd2 << endl;
    sd2.replace(sd2.begin(), sd2.end(), " This is an Example of Replace!"); //Replace
    cout << " Destination sd2 (Replace All): " << sd2 << endl; //replace()函数中使用迭代器
}
```

例 2-14 的执行效果如图 2-12 所示。



```

Destination: The zip code of Hondelage in Germany is 38108.
Destination <All Toupper>: THE ZIP CODE OF HONDELAGE IN GERMANY IS 38108.
Destination sd1: THE ZIP CODE OF HONDELAGE IN GERMANY IS
Destination sd2: .80183 SI YNAMREG NI EGALEDNOH FO EDOC PIZ EHT
Destination sd2(Erased 15 chars): EG NI EGALEDNOH FO EDOC PIZ EHT
Destination sd2(Insert 8 chars): 12345678EG NI EGALEDNOH FO EDOC PIZ EHT
Destination sd2(Replace All): This is an Example of Replace!
Press any key to continue
  
```

图 2-12 例 2-14 的执行效果



总结 上述内容对迭代器在 string 类及其相关成员函数中的使用做了详细的描述。希望读者认真体会。

2.4.10 字符串对配置器的支持

配置器也是 STL 的重要内容。使用 STL 必然会涉及容器，而容器中存储了大量的数值，必然需要分配内存空间。配置器的作用就是为容器分配内存。

配置器最早是为将内存模型抽象化而提出的。所以使用内存配置器分配内存时，是按对象的个数进行的，而不是按字节数。这有别于原来的 new [] 和 new 操作符。配置器最大的优点在于：配置器实现了将算法、容器与物理存储细节分隔。配置器可以提供一套分配与释放内存的标准方式，并提供用作指针类型和引用类型的标准名称。目前而言，配置器仅是一种纯粹的抽象。行为上类似分配器的类型都可看作配置器。

C++ STL 提供了标准分配器，目的是为用户提供更多的服务。basic_string 模板以及 string 类均提供了对常见配置器的相关支持。basic_string 类模板中包含 1 个配置器类型的成员 allocator_type。对于 string 对象，allocator_type 可以作为配置器类的对象使用；对 string 类而言，allocator_type 等价于 allocator < char >，即分配数据类型为 char 的内存，便于 string 类的对象存储 char 型字符。

同时 basic_string 类模板的第 3 个参数也是配置器模板参数。basic_string 类模板的形式如下：

```
template <class CharType, class Traits = char_traits < CharType >, class Allocator = allocator
< CharType >> class basic_string
```

而 string 类的声明形式如下：

```
typedef basic_string < char > string;
```

对于 basic_string 类模板，其第 1 个参数是 CharType，第 2 个参数和第 3 个参数的默认值和 CharType 均相关。在声明 string 类时，参数 char 取代模板中的 CharType，string 即成为模板的实例，同时模板中的第 3 个参数成为“class Allocator = allocator < char >”，其意义为 string 中对象的内存类型为 char 型。

string 类还提供了 1 个和配置器相关的函数 get_allocator()，其函数原型为：

```
allocator_type string:: get_allocator() const
```

函数返回 `string` 类的内存模型对象，可以用于构造新的字符串。下面以例 2-15 为例介绍该函数的使用方法。

例 2-15

```
#include <iostream>
#include <string>
#include <memory>
using namespace std;
void main()
{
    string s("abcd");
    basic_string<char> s1 (s.get_allocator());
    basic_string<char>:: allocator_type aT=s1.get_allocator();
    string:: size_type se=s1.size();
    cout << se << endl;
}
```

由于在 `string` 类中，`allocator` 是保护成员，难以直接调用对内存的直接配置。本节仅以简单的知识简要介绍配置器的概念和使用方法。关于配置器的详细使用见后面相应的章节。



总结 对于配置器，一般情况下都是使用默认配置器。对于本节内容，读者了解即可。

2.5 小结

本章首先对字符串类进行了详细的介绍，之后介绍字符串的特点。2.3 节对字符串类模板基础知识做了简要介绍。2.4 节对字符串模板类进行了深入研究，并对其中包含的绝大部分内容进行了详细讲解。

字符串是任何程序员必须大量使用的数据类型。对于绝大多数程序员来说，掌握字符串类的基础知识及其使用方法是必需的。

第 3 章

容器——对象储存器

容器类是容纳、包含一组对象或对象集的对象。通过容器类提供的成员函数可实现对序列中元素的各种操作。C++ STL 中的部分算法可以用于容器序列的控制（关于算法的内容在第 4 章介绍）。容器类库中包括七种基本容器：向量（vector）、列表（list）、双向队列（deque）、集合（set）、多重集合（multiset）、映射（map）和多重映射（multimap）。通常，向量（vector）可以认为是包含 1 个或 N 个更多元素的数组；列表（list）是由节点组成的双向链表，每个节点包含 1 个元素；双向队列（deque）是包含 N 个连续的、指向不同元素的指针组成的数组；集合（set）是由节点组成的，每个节点包含 1 个元素，节点之间以某种谓词排序；多重集合（multiset）是允许存在两个数值（或次序）相等的元素集合；映射（map）是由“{键, 值} 对”组成的集合，同样以某种谓词排序；多重映射（multimap）是允许“键对”包含相等值（或次序）的映射。

STL 在实现诸多容器类的同时，还实现了部分序列式容器的适配器（adapter）。容器的适配器是对原有基本容器不足的补充，是对原有基本容器功能的补充。所有适配器均不提供迭代器，元素访问是通过专有接口函数实现的。

本章重点介绍各种容器的定义及其使用方法。



提示 本章在介绍基础知识之后，给每个知识点均配备了较详实的例题，便于读者在阅读的同时，能有机会动手尝试。

3.1 容器概念

1. 容器概念

在汉语中，容器是指用来包装或装载物品的存储器皿（例如箱、罐、坛），或者成形或柔软不成形的包覆材料。在 C++ 语言中，STL 提供了大量的容器类。容器类的对象可以认为是“实在”的容器。在编程过程中，容器可以认为是“用来存储和组织其他对象的对象”。严格来说，容器适配器并不是容器，而是使用容器的对象，是在容器的基础上发展起来的。

2. 容器成员和函数

STL 提供的每种容器均提供了许多操作行为和成员。作为容器的成员，必须满足三个条件：

1) 元素必须是可复制的。所有容器均会产生 1 份元素副本，不会发生 alias 现象；所有容器操作行为传回的均是其元素的副本。这导致复制构造函数的执行非常频繁。

2) 元素必须是可指派（assign）的。容器的成员函数以及 STL 的各种算法均可利用

assign() 函数为元素设定新值。

3) 元素必须是可释放的 (经过析构函数释放内存)。使用者从容器中删除元素时, 容器必须释放该元素所占的内存。按这种需求, 析构函数不能被设置为 private 类型。

作为容器的成员函数 (操作), 需要具备一些共同的能力。其中最重要的能力有三个: ①容器均能提供 value, 而非 reference (引用)。涉及元素操作时, 元素满足上述的三个条件。②所有元素自动形成顺序, 即按此顺序可多次遍历所有元素。这些容器均包含可返回迭代器的函数, 使用这些迭代器可遍历元素。这些迭代器是算法和容器的接口。③函数使用者必须确保传递的参数符合要求。否则, 可能会导致未定义的行为 (通常 STL 是不会抛出异常的)。以上均为容器中成员函数的核心竞争力。所有容器均包含部分共有的成员函数, 主要包括初始化、容器大小、比较以及赋值和交换等。

3. 容器的种类和数据结构

STL 容器通常分为三种: 序列式容器、关联性容器和容器配接器。

1) 序列式容器 (sequence 容器)。此类容器中的元素是有序的, 但未排序。包括 vector (动态数组), deque (双向队列), list (双向串)。

2) 关联性容器, 容器中的元素都经过排序。包括 set, multiset, map, multimap, 和 hash (散列) table。

3) 容器配接器, 是以某种 STL 容器作为底, 修改其接口, 具备各自的特点。包括 stack (栈), queue (队列), priority_queue (优先队列)。

STL 容器的数据结构也包括三种: string (字符串)、bitset (位) 和 valarray (数组)。

1) string 字符串。这种数据结构中保存的是字符。string 并不是真正的类, 而是 1 个 basic_string 类的类型定义。其定义为:

```
template <class charT, class traits = char_traits<charT>, class Allocator = allocator<charT>>
class basic_string;
typedef basic_string<char> string;
```

2) bitset。这种数据结构中保存的是 bits 的结构体。每个 bit 表示 1 个标志 (flag)。其长度固定。长度便是模板的自变量。详见例 3-1。

例 3-1

```
#include <iostream>
#include <bitset>
#include <string>
using namespace std;
void main()
{
    bitset<10> bs1(7); //bitset 中长度为 10 个 bit;初始化为十进制数 7
    bitset<10> bs2(string("1000101011")); //初始化为 10 个字符长度的字符串
    cout << bs1 << endl; //输出
    cout << bs2 << endl; //输出
    cin.get(); //任意键退出
}
```

例 3-1 的执行效果如图 3-1 所示。



图 3-1 例 3-1 的执行效果

图 3-1 中，第 1 行字符正好是十进制数 7 的二进制表现形式；第 2 行字符正好是字符串“1000101011”。

3) valarray。这种数据结构是数学中的线性数列概念的呈现。其优点为：如同处理单一数值，对整个 valarray 中的每个元素实现运算。详见例 3-2。

例 3-2

```
#include <iostream>
#include <valarray>
using namespace std;
template <typename T> void printValarray(const valarray<T>& va) //体验一下模板函数的用法
{
    for(int i=0;i<va.size();i++)
        cout<<va[i]<<" ";
    cout<<endl;
}
void main()
{
    valarray<int> val(4); //包含 4 个元素,但未指定元素的值
    printValarray(val); //输出
    valarray<int> va2(3,4); //包含 4 个元素,其数值均为 3;
    printValarray(va2);
    int ia[] = {1,2,3,4,5,6}; //定义整数型数组
    valarray<int> va3(ia,sizeof(ia)/sizeof(ia[0])); //动态数组大小和数组 ia 的元素个数相同
    printValarray(va3);
    valarray<int> va4(ia+1,4); //4 个元素,数值分别为数组 ia 的前 4 个元素加 1
    printValarray(va4);
    val = (va2 + va4) * va4; //给 val 赋值
    printValarray(val);
}
```

例 3-2 的执行效果如图 3-2 所示。

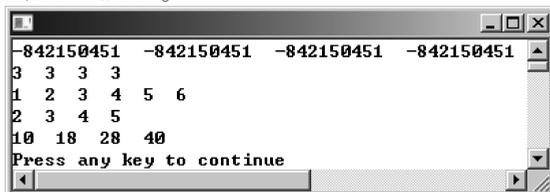


图 3-2 例 3-2 的执行效果



提示 请读者注意例 3-2 中 `valarray` 类型对象的使用方法。

3.2 序列式容器

对于基本容器，程序员可以通过增加条件来对其进行改进。序列就是一种重要的改进。`deque`、`list`、`queue`、`priority_queue`、`stack` 和 `vector` 这六种容器均为序列式容器。序列最重要的特点是可以在首端删除元素，在尾端添加元素。尤其是双向序列，它允许在两端添加和删除元素。序列中应该包含至少一种迭代器，从而保证元素按特定的顺序排列，而不会在两次迭代间发生变化。

数组和链表均是序列，但分支结构不是序列。序列中的元素具有确定顺序，使用时可以执行将元素插入至特定位置、删除特定元素、删除某范围内的所有元素等操作。C++ STL 提供的基本序列式容器包括 `vector`、`list`、`deque`；同时还包括三个适配器 `stack`、`queue` 和 `priority_queue`。

`vector`、`list` 和 `deque` 三种序列不可能既互相实现，又不损失效率。另一方面，`stack` 和 `queue` 都可以在这三种基本序列式容器的基础上高效实现。3.4 节将详述适配器的相关内容。

在使用 `vector`、`list`、`deque` 这三种容器时，需要分别包含相应的头文件。前面讲过，容器都是类模板，要定义或实现某种特定的容器对象，必须在容器名后加 1 对尖括号，括号中提供容器中所存放元素的数据类型。

```
#include <vector >
#include <list >
#include <deque >
```

```
vector <string > svec;
list <int > ilist;
deque <long > litem;
```

下面分小节逐一阐述这三种序列式容器。

3.2.1 `vector`（向量）类模板

`vector` 是定义于名称空间（`namespace`）`std` 内的模板，其定义在头文件 `<vector >` 中。

其原型为：

```
template <class T, class Allocator = allocator <T >> class vector;
```

`vector` 中的元素可以是任意型别 `T`，必须具备可设置和可复制两个属性。模板的第 2 个参数是关于空间配置器设置的，用于定义内存模型，其默认内存模型是 C++ STL 提供的 `allocator`，对于一般的程序员来说，可有可无。

`vector` 是最简单的序列式容器，支持随机访问元素。这一属性使 `vector` 有时显得效率低一些。`vector` 的“大小”与容量之间有重要差别。容量肯定是大于或等于其“大小”的。`vector` 就像一个动态数组，是典型的“将元素置于动态数组中加以管理”的抽象概念。然

而, C++ 标准并未规定必须以动态数组来实现 `vector`, 仅规定了相应的条件和操作复杂度。在程序开发过程中, 程序员使用 `vector` 作为动态数组是非常方便的。**vector 的迭代器是随机存取迭代器, 对任何一个 STL 算法均奏效。**尤其在容器末端或删除元素时, `vector` 性能相当好。

`vector` 可以实现数据结构中的队列、数组和堆栈的所有功能。一旦 `vector` 定义了类对象, 就可以“装”东西了。

下面介绍 `vector` 类模板的使用。

1. `vector` 类基础

在使用 `vector` 类模板, 程序员需要定义该模板的对象, 并对其初始化。`vector` 容器对象的大小和容量也是使用的重要前提, 就像使用数组一样, 数组的大小决定了数组能否正常使用。

(1) `vector` 对象定义

定义 `vector` 类对象, 主要是使用 `std` 名称空间的 `vector` 类模板。详见例 3-3。

例 3-3

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;
void main()
{
    vector<string>myvt;           //定义类模板对象
    cout << "OK!" << endl;
}
```

在例 3-3 中, 语句 `vector<string>myvt` 使用了类模板 `vector` 定义类模板的对象 `myvt`, 容器 `myvt` 中所存储元素的数据类型是 `string`。当然, 也可以存储其他数据类型的元素, 例如 `int`、`long`、`double` 等。命名空间 `std` 是类的包容器 (或者叫“更大的容器”)。上述程序在编译时, 会出现 4 个警告信息 (warning), 警告信息的编号为 `c4786`。为避免出现警告信息, 程序员可以在代码中添加语句“`#pragma warning (disable:4786)`”。修改后的代码如下:

```
#pragma warning(disable:4786)
#include <iostream>
#include <string>
#include <vector>
using namespace std;
void main()
{
    vector<string>myvt;
    cout << "OK!" << endl;
    cout << "成功消除编译警告信息!" << endl;
}
```

第一行代码的作用是消除警告信息 `c4786`。



提示

注意上例中 `#pragma` 语句的用法。

(2) vector 类对象初始化

要对 vector 类对象实现初始化操作可以使用 push_back() 函数。下面在例 3-3 中添加部分代码, 以实现 vector 类对象 myvt 的初始化。

```
#pragma warning(disable:4786)
#include <iostream>
#include <string>
#include <vector>
using namespace std;
void main()
{
    vector<string>myvt;
    myvt.push_back (" 1. Beijing City. ");
    myvt.push_back (" 2. Tianjin City. ");
    myvt.push_back (" 3. Shanghai City. ");
    myvt.push_back (" 4. Chongqing City. ");
    cout << " OK!" <<endl;
    cout << " 成功消除编译警告信息!" << endl;
    vector<string>:: iterator it;
    for (it=myvt.begin(); it! =myvt.end(); it++)
        cout <<* it <<endl;
    cin.get();
}
```

上述代码执行之后, 容器 myvt 中将包含 4 个字符串。push_back() 函数将对象放入容器中, 有时会错误地把信息添加到容器中, 此时可以使用 pop_back() 函数将其弹出。

既然 vector 类模板实例化的对象是容器, 那么按照“容器”的物理涵义, 对于既定的容器, 必然存在容器的容量。对于 vector 的对象, 程序员可以使用 reserve() 函数预先设置容器的容量。例如,

```
#pragma warning(disable:4786)
#include <iostream>
#include <string>
#include <vector>
using namespace std;
void main()
{
    vector<string>myvt;
    myvt.reserve(4);
    myvt.push_back (" 1. Beijing City. ");
    myvt.push_back (" 2. Tianjin City. ");
    myvt.push_back (" 3. Shanghai City. ");
    myvt.push_back (" 4. Chongqing City. ");
    cout << " OK!" <<endl;
    cout << " 成功消除编译警告信息!" << endl;
    vector<string>:: iterator it;
    for (it=myvt.begin(); it! =myvt.end(); it++)
        cout <<* it <<endl;
    cin.get();
}
```

例 3-3 的执行效果如图 3-3 所示。



图 3-3 例 3-3 的执行效果

`myvt.reserve(4)` 语句可以实现设置容器的容量为 4 个字符串。一般情况下，定义 `vector` 类的对象后，如果没有预先设置容器的容量，是不允许直接给容器中的元素赋值的，例如，若执行下面一段代码，程序会发生异常。

```
vector<string> myvt;  
myvt[0] = 1;  
myvt[1] = 2;  
myvt.reserve(4);
```

(3) 容器的大小和容量

`vector` 类模板定义了 `size()` 和 `capacity()` 两个函数，用以实现统计容器元素的目的；还定义了 `resize()` 和 `reserve()` 两个函数，用以实现设置容器大小的目的。

`size()` 函数和 `capacity()` 函数可以统计容器中元素的数量。`size()` 函数返回容器中现有的元素数；`capacity()` 函数返回容器中实际能够容纳的元素数。`max_size()` 函数可以返回容器所能容纳的最大元素数。一般情况下，当元素数量超越 `capacity()` 函数返回的数值时，`vector` 有必要重新配置内部存储器。

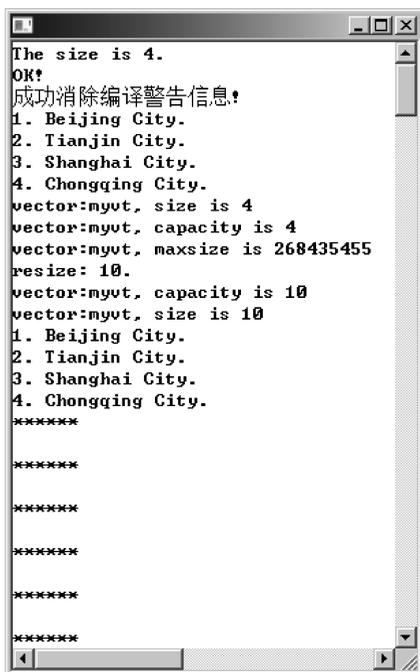
`reserve()` 函数可以预先设置容器的容量；`resize()` 函数可以修改容器的大小。`reserve()` 函数可以保留适当的容量，避免重新配置内存。当然，在定义容器对象时，通过传递函数的形式，也可以实现设置 `vector` 的起始大小。

以上知识点，可参考完整的例 3-3。

```
#pragma warning(disable:4786)  
#include <iostream>  
#include <string>  
#include <vector>  
using namespace std;  
void main()  
{  
    vector<string> myvt;  
    myvt.reserve(4);  
    cout << "The size is 4. " << endl;  
    myvt.push_back (" 1. Beijing City. ");  
    myvt.push_back (" 2. Tianjin City. ");  
    myvt.push_back (" 3. Shanghai City. ");  
    myvt.push_back (" 4. Chongqing City. ");  
    cout << " OK!" << endl;  
}
```

```
cout << "成功消除编译警告信息!" << endl;
vector<string>::iterator it;
for(it=myvt.begin();it!=myvt.end();it++)
    cout << *it << endl;
int m=myvt.size();
int n=myvt.capacity();
int m1=myvt.max_size();
cout << " vector: myvt, size is " << m << endl;
cout << " vector: myvt, capacity is " << n << endl;
cout << " vector: myvt, maxsize is " << m1 << endl;
myvt.resize(10);
cout << " resize: 10. " << endl;
int n1=myvt.capacity();
int n2=myvt.size();
cout << " vector: myvt, capacity is " << n1 << endl;
cout << " vector: myvt, size is " << n2 << endl;
for (it=myvt.begin(); it!=myvt.end(); it++)
{
    if (*it=="")
        cout << " * * * * * " << endl;
    cout << *it << endl;
}
cin.get();
}
```

完整的例 3-3 的执行效果如图 3-4 所示。



```
The size is 4.
OK!
成功消除编译警告信息:
1. Beijing City.
2. Tianjin City.
3. Shanghai City.
4. Chongqing City.
vector:myvt, size is 4
vector:myvt, capacity is 4
vector:myvt, maxsize is 268435455
resize: 10.
vector:myvt, capacity is 10
vector:myvt, size is 10
1. Beijing City.
2. Tianjin City.
3. Shanghai City.
4. Chongqing City.
*****
*****
*****
*****
*****
```

图 3-4 完整的例 3-3 的执行效果



提示 请关注例 3-3 中 `size()`、`capacity()`、`reserve()` 和 `resize()` 函数的用法。

2. vector 类的基本应用函数

vector 最基本的应用包括判断向量是否为空、遍历向量元素以及使用算法等。下面逐一介绍。

(1) 判断向量是否为空

向量模板 `vector<T>` 提供了一个 `empty()` 函数。此函数可以判断向量容器中元素是否为 0，如果向量为空，函数返回真；如果向量不为空，函数返回非真。`empty()` 函数的函数原型为：

```
template<class _TYPE, class _A> bool vector::empty() const;
```

下面举例说明 `empty()` 函数的使用方法。在例 3-4 中，首先定义了结构体类型 `ST`，其次定义了初始化 `Origin()` 函数。`Origin()` 函数可以根据指定的参数，实现对向量的初始化，为向量添加内容。

例 3-4

```
#include <iostream>
#include <vector>
using namespace std;
struct ST{                               //定义结构体类型
    int id;
    double db;
};
void Origin(int num, vector<ST>& vt)     //定义初始化函数
{ int m=num;
  ST temp;
  for(int i=0;i<m;i++)
  { temp.id=i+1;
    temp.db=(i+1)*10;
    vt.push_back(temp);                 //初始化向量
  }
}
void main()
{ ST tmp;
  vector<ST>myvt;                        //定义向量
  Origin(5, myvt);                       //初始化向量
  int size=myvt.size();                  //统计向量中元素个数
  cout<<" size: " <<size<<endl;         //输出向量个数
  while (! myvt.empty())                //判断向量是否为空
  { tmp=myvt.back();                    //如果非空，输出其末尾元素，并弹出
    cout<<" id " <<tmp.id<<" , db: " <<tmp.db<<endl;
    myvt.pop_back();
  }
}
```

例 3-4 的执行效果如图 3-5 所示。

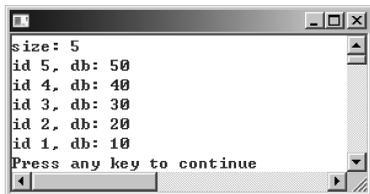


图 3-5 例 3-4 的执行效果

请读者认真对照程序输出结果和源代码，以体会该函数的用法。

上面简单介绍了 `empty()` 函数，这里顺便讲一下 `clear()` 函数。`clear()` 函数可以将容器中的所有元素移出，将容器清空。



提示 请关注例 3-4 中 `empty()` 函数在 `while` 循环中的用法。

(2) 遍历 `vector` 型容器

要遍历 `vector` 中的元素，必须使用循环语句 `for` 或者 `while`。可以使用迭代器实现遍历，也可以通过使用 `at()` 函数和循环语句实现。下面举例说明。在例 3-4 中添加 `Iter_for()` 和 `at_for()` 两个函数。

```
void Iter_for(vector<ST>& vt) //使用迭代器遍历 vector 型容器
{
    ST temp;
    vector<ST>::iterator iter;
    for(iter = vt.begin(); iter != vt.end(); iter++)
    {
        temp = *iter;
        cout << "id: " << temp.id << ", db: " << temp.db << endl;
    }
}

void at_for (vector<ST>& vt) //使用 at() 函数遍历 vector 型容器
{
    ST temp;
    int i = 0;
    int m = vt.size();
    for (i = 0; i < m; i++)
    {
        temp = vt.at (i);
        cout << "id: " << temp.id << ", db: " << temp.db << endl;
    }
}
```

修改 `main()` 函数如下：

```
#include <iostream>
#include <vector>
using namespace std;
struct ST{
    int id;
```

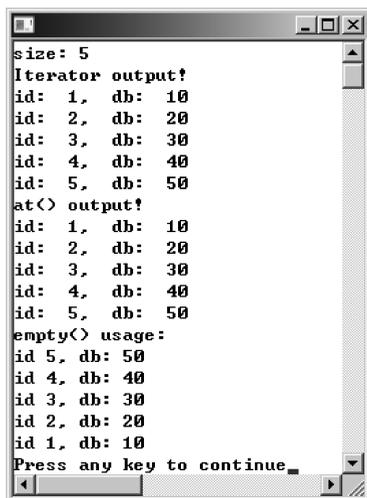
```
double db;
};
void Origin(int num, vector<ST>& vt)
{ int m=num;
  ST temp;
  for(int i=0;i<m;i++)
  { temp.id=i+1;
    temp.db=(i+1)*10;
    vt.push_back(temp);
  }
}
void Iter_for (vector<ST>& vt)
{ ST temp;
  vector<ST>::iterator iter;
  for (iter=vt.begin(); iter!=vt.end(); iter++)
  { temp=*iter;
    cout<<" id: " <<temp.id<<" , db: " <<temp.db<<endl;
  }
}
void at_for (vector<ST>& vt)
{ ST temp;
  int i=0;
  int m=vt.size();
  for (i=0; i<m; i++)
  { temp=vt.at(i);
    cout<<" id: " <<temp.id<<" , db: " <<temp.db<<endl;
  }
}
void main()
{ ST tmp;
  vector<ST>myvt;
  Origin(5, myvt);
  int size=myvt.size();
  cout<<" size: " <<size<<endl;
  cout<<" Iterator output!" <<endl;
  Iter_for(myvt);
  cout<<" at() output!" <<endl;
  at_for(myvt);
  cout<<" empty() usage:" <<endl;
  while (!myvt.empty())
  { tmp=myvt.back();
    cout<<" id " <<tmp.id<<" , db: " <<tmp.db<<endl;
    myvt.pop_back();
  }
}
```

上述两种方法均较好地实现了对元素的遍历。



提示 请关注例 3-4 中如何遍历 vector 型容器中的元素。

程序修改后的执行效果如图 3-6 所示。



```
size: 5
Iterator output!
id: 1, db: 10
id: 2, db: 20
id: 3, db: 30
id: 4, db: 40
id: 5, db: 50
at() output!
id: 1, db: 10
id: 2, db: 20
id: 3, db: 30
id: 4, db: 40
id: 5, db: 50
empty() usage:
id 5, db: 50
id 4, db: 40
id 3, db: 30
id 2, db: 20
id 1, db: 10
Press any key to continue
```

图 3-6 遍历 vector 型容器

(3) 使用算法

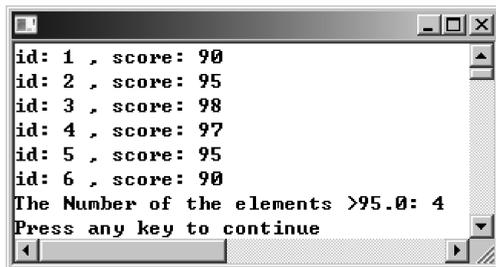
使用 vector 类模板，还可以通过使用部分算法，实现对 vector 的操作。下面以 `for_each()` 和 `count()` 两个函数为例进行说明。在例 3-5 中，定义了一个模板函数 `Original()` 和两个全局函数 `out()` 和 `greater95()`。通过例 3-5，既复习了函数模板的用法，也提前了解算法的使用。

例 3-5

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
struct Student{
    int id;
    double score;
};
template <class T> void Original(T&myvt)
{ Student temp;
  temp.id=1;
  temp.score=90;
  myvt.push_back (temp);
  temp.id=2;
```

```
temp.score = 95;
myvt.push_back (temp);
temp.id = 3;
temp.score = 98;
myvt.push_back (temp);
temp.id = 4;
temp.score = 97;
myvt.push_back (temp);
temp.id = 5;
temp.score = 95;
myvt.push_back (temp);
temp.id = 6;
temp.score = 90;
myvt.push_back (temp);
}
void out (Student& stu)
{ cout << " id: " << stu.id << ", score: " << stu.score << endl; //输出信息
}
bool greater95 (Student& stu) //如果大于 95, 就返回 "true"
{ if (stu.score >= 95.0)
    return 1;
  else
    return 0;
}
void main()
{ vector<Student> myvt; //声明向量容器
  vector<Student>::iterator iter; //声明迭代器
  int countV = 0;
  Original (myvt); //初始化
  for_each (myvt.begin(), myvt.end(), out); //for_each 算法
  countV = count_if (myvt.begin(), myvt.end(), greater95); //统计 score > 95.0 的元素个数
  cout << " The Number of the elements >95.0: " << countV << endl; //输出符合条件的元素个数
}
```

例 3-5 的执行效果如图 3-7 所示。



```
id: 1 , score: 90
id: 2 , score: 95
id: 3 , score: 98
id: 4 , score: 97
id: 5 , score: 95
id: 6 , score: 90
The Number of the elements >95.0: 4
Press any key to continue
```

图 3-7 例 3-5 的执行效果

 **总结** 请关注函数模板的使用以及 `for_each()` 函数的使用。最重要的是关注如何定义 `for_each()` 的子进程。对于算法的子进程，后面还会有更多的介绍。读者不必急于掌握其原理，应先“照葫芦画瓢”学会其使用方法。随着使用次数的增加，逐渐加深印象，进而深刻理解其内涵。

3. vector 高级编程

下面主要介绍 `vector` 容器相关的元素访问方法、迭代器相关函数、元素查找和搜索、字符串处理、元素排序、插入元素、删除元素、交换元素等内容。

(1) 元素访问方法

按照 C 和 C++ 的惯例，第一个元素的下标为 0，最后一个元素的下标为 `size() - 1`，即第 `n` 个元素的下标为 `n - 1`。可以直接访问 `vector` 型容器中元素的操作方法主要包括 `at()`、`[]`、`front()` 和 `back()`。例如，

```
vector<typename T> c;
```

```
c.at(index);  
c[index];  
c.front();  
c.back();
```

其中，`c.at(index)` 函数的返回值是引用类型。该函数既可以取出元素值，也可以对元素赋值。

`c[index]` 的返回值是引用类型。该函数既可以取出元素，也可以对元素赋值，但必须确定下标是有效的。

`c.front()` 函数用于返回第一个元素。

`c.back()` 函数用于返回最后一个元素。

(2) 迭代器相关函数

`vector` 类模板提供部分常规函数来获取迭代器。迭代器是随机访问迭代器，它类似于一个指向 `vector` 中元素的指针通过迭代器甚至可以操作所有算法。和迭代器相关的函数主要包括 `begin()`、`end()`、`rbegin()` 和 `rend()`。其中 `begin()` 函数指向第一个元素；`end()` 函数指向最后元素的下一个位置；`rbegin()` 函数指向逆向迭代的第一个元素；`rend()` 函数指向逆向迭代的最后元素的下一位置。`begin()` 和 `end()` 函数的返回值均为迭代器类型；`rbegin()` 和 `rend()` 函数的返回值均为逆向迭代器类型。

(3) 元素查找和搜索

若要查找和搜索 `vector` 型容器中的元素，可以使用 STL 的通用算法 `find()` 函数。若要有条件地搜索相关元素，可以使用 `find_if()` 算法函数。这两个算法函数均可以使用迭代器，两个迭代器参数决定了查找和搜索的范围。这两个函数的返回值均为迭代器类型。

`find()` 函数的原型为：

```
template<class InputIterator, class T> inline  
InputIterator find(InputIterator first, InputIterator last, const T& value)
```

`find_if()` 函数的原型为：

```
template < class InputIterator, class T, class Predicate > inline  
InputIterator find_if ( InputIterator first, InputIterator last, Predicate predicate)
```

下面先举例说明如何实现在 vector 型容器中的元素查找和搜索。

例 3-6

```
#include < iostream >  
#include < vector >  
#include < algorithm >  
#include < functional >  
using namespace std;  
void print (const int& temp)  
{ cout << temp << endl;  
}  
void main()  
{ const int ARRAY_SIZE = 8 ;  
  int IntArray [ARRAY_SIZE] = { 1, 2, 3, 4, 4, 5, 6, 7 } ;  
  vector < int > myvt ;  
  int* location_index = NULL ;  
  for ( int i = 0 ; i < 8 ; i ++ )  
    myvt.push_back ( IntArray [ i ] ) ;  
  for_each ( myvt.begin(), myvt.end(), print ) ;  
  location_index = find ( myvt.begin(), myvt.end(), 2 ) ; //查找和搜索  
  cout << " 数字 2 的下标是 : " << ( location_index - myvt.begin() ) << endl ;  
  location_index = find_if ( myvt.begin(), myvt.end(), bind2nd ( greater < int > (), 5 ) ) ;  
  cout << " 第一个大于 5 的数字的下标是 : " << ( location_index - myvt.begin() ) << endl ;  
}
```

例 3-6 的执行效果如图 3-8 所示。

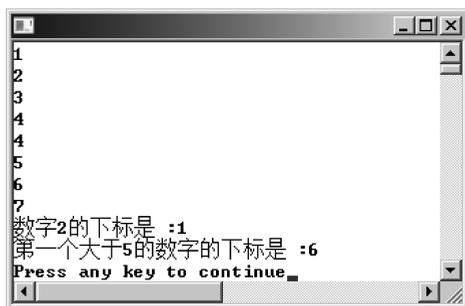


图 3-8 例 3-6 的执行效果



提示 读者应掌握 find() 函数和 find_if() 函数的使用，尤其注意 find_if() 函数中条件表达式的使用。

(4) 字符串处理

使用 vector 管理字符串，可以处理字符串中的每个字符，并且可以自动管理内存。将字符串中的每一个字符作为 vector 型容器中的元素，使用容器的所有成员函数就可以便捷地完

成字符串的操作读者可参考本书第2章中关于字符串的内容，此处不再赘述。

(5) 元素排序

若要实现对 `vector` 型容器中元素的排序，需要使用算法函数 `sort()` 或其他排序算法。有些容器支持对特殊算法的实现，而使用算法排序有助于提高计算性能。但 `vector` 类模板没有提供具有排序算法的成员函数。下面简要介绍使用 `sort()` 算法函数实现对 `vector` 型容器中元素排序的方法。

例 3-7

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;
class student{
public:
    student(const string &a,double b):name(a),score(b){}
    string name;
    double score;
    bool operator < (const student& m) const
    {   return score < m.score;
    }
};
bool name_sort_less (const student& m, const student& n)           //定义子进程
{   return m.name < n.name;
}
bool name_sort_greater (const student& m, const student& n)       //定义子进程
{   return m.name > n.name;
}
bool score_sort (const student& m, const student& n)              //定义子进程
{   return m.score > n.score;
}
void print (student& S)                                           //定义子进程
{   cout << S.name << "          " << S.score << endl;
}
void Original (vector < student > & V)                            //初始化
{   student st1 (" Tom", 74);
    V.push_back (st1);
    st1.name = " Jimy";
    st1.score = 56;
    V.push_back (st1);
    st1.name = " Mary";
    st1.score = 92;
    V.push_back (st1);
    st1.name = " Jessy";
    st1.score = 85;
```

```
V.push_back (st1);
st1.name = " Jone";
st1.score = 56;
V.push_back (st1);
st1.name = " Bush";
st1.score = 52;
V.push_back (st1);
st1.name = " Winter";
st1.score = 77;
V.push_back (st1);
st1.name = " Ander";
st1.score = 63;
V.push_back (st1);
st1.name = " Lily";
st1.score = 76;
V.push_back (st1);
st1.name = " Maryia";
st1.score = 89;
V.push_back (st1);
}
void main ()
{
    vector < student > vect;
    Original (vect);
    cout << " - - - -Before sorted. - - - -" << endl;
    for_each (vect.begin(), vect.end(), print);           //输出容器中的元素
    sort (vect.begin(), vect.end());                     //按 score 从小到大排序
    cout << " - - - -After sorted by score. - - - -" << endl;
    for_each (vect.begin(), vect.end(), print);         //输出排序结果
    sort (vect.begin(), vect.end(), name_sort_less);    //按 name 从小到大排序
    cout << " - - - -After sorted by name. - - - -" << endl;
    for_each (vect.begin(), vect.end(), print);         //输出排序结果
    sort (vect.begin(), vect.end(), score_sort);       //按 score 从大到小排序
    cout << " - - - -After sorted by score. - - - -" << endl;
    for_each (vect.begin(), vect.end(), print);         //输出容器中的元素
    sort (vect.begin(), vect.end(), name_sort_greater); //按 name 从大到小排序
    cout << " - - - -After sorted by name. - - - -" << endl;
    for_each (vect.begin(), vect.end(), print);         //输出排序结果
}
```

例 3-7 重点讲述了如何使用 `sort()` 算法函数实现对 `vector` 型容器中的元素进行排序。在使用 `sort()` 算法函数时, 针对元素的特点实现了四种排序, 即分别对 `name` 和 `score` 进行了从大到小和从小到大排序。

程序执行结果如下:

```

- - - -Before sorted. - - - -
Tom      74
Jimmy   56
Mary    92
Jessy   85
Jones   56
Bush    52
Winter  77
Ander   63
Lily    76
Maryia  89
- - - -After sorted by score. - - - -
Bush    52
Jimmy   56
Jones   56
Ander   63
Tom      74
Lily    76
Winter  77
Jessy   85
Maryia  89
Mary    92
- - - -After sorted by name. - - - -
Ander   63
Bush    52
Jessy   85
Jimmy   56
Jones   56
Lily    76
Mary    92
Maryia  89
Tom      74
Winter  77
- - - -After sorted by score. - - - -
Mary    92
Maryia  89
Jessy   85
Winter  77
Lily    76
Tom      74
Ander   63
Jimmy   56
Jones   56
Bush    52
- - - -After sorted by name. - - - -

```

Winter	77
Tom	74
Maryia	89
Mary	92
Lily	76
Jone	56
Jimy	56
Jessy	85
Bush	52
Ander	63



提示

读者应掌握 `sort()` 算法函数的使用，并学会定义其子进程。

(6) 插入元素

若要向现有的 `vector` 型容器中插入一个新的元素，可以使用 `push_back()` 函数将元素加入至 `vector` 型对象（容器）的末尾；可以使用 `insert()` 函数将元素插入至 `vector` 型容器中的任意位置，即 `insert()` 函数可以实现在迭代器指向位置插入元素，其中 `insert()` 函数返回值为迭代器类型，指向刚插入的元素；可以在迭代器指定位置插入多个连续的不同元素；可以插入多个值相同的元素。`push_back()` 函数和 `insert()` 函数的原型为：

```
template<class _TYPE, class _A>void vector:: push_back (const _TYPE& X);
iterator insert (iterator it, const T& x = T());
void insert (iterator it, size_type n, const T& x);
void insert (iterator it, const_iterator first, const_iterator last);
```

下面通过例 3-8 说明 `push_back()` 和 `insert()` 函数的使用方法。

例 3-8

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
void OutToScreen (int& Ele) //输出元素至屏幕
{cout << Ele << ", ";
}
void main()
{vector<int> myvt; //定义vector 容器型对象
  for(int i=0;i<10;i++)
    myvt.push_back (i); //初始化容器的元素
  myvt.insert (myvt.begin(), -1); //在向量起始位置之前插入元素 -1
  for_each (myvt.begin(), myvt.end(), OutToScreen); //将向量中的元素输出
  cout << endl;
  myvt.insert (myvt.end(), -2); //在向量末尾之前插入 -2
  for_each (myvt.begin(), myvt.end(), OutToScreen);
```

```
cout << endl;
vector<int> vt2; //定义新的数值序列
vt2.push_back(-8);
vt2.push_back(-9);
myvt.insert(myvt.end(), vt2.begin(), vt2.end()); //插入多个数值
for_each(myvt.begin(), myvt.end(), OutToScreen);
cout << endl;
myvt.insert(myvt.begin(), 3, 0); //插入多个相同的数值
for_each(myvt.begin(), myvt.end(), OutToScreen);
cout << endl;
}
```

例 3-8 的执行效果如图 3-9 所示。请读者对照源代码和执行效果深刻理解插入元素的方法。

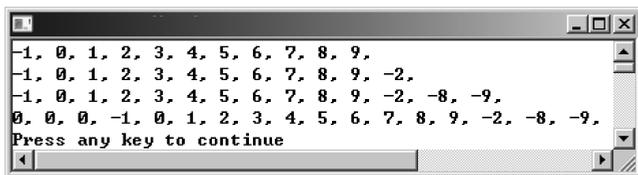


图 3-9 例 3-8 的执行效果

(7) 删除元素

在定义 vector 型容器时，需要初始化其元素，并且需要同时使用其他 STL 容器中的区间数据来进行初始化操作。该 STL 容器不必是 vector 型，只要元素的类型相同即可。例如，

```
vector<int> Harry;
Harry.push_back(1);
Harry.push_back(2);
vector<int> Bill(Harry.begin(), Harry.end());
```

若要删除容器中的元素，可以使用三个成员函数：pop_back()、erase()和 clear()。还可以使用算法库的 remove()算法函数来实现。

pop_back()函数可以删除最后 1 个元素；erase()函数可以删除由迭代器指定的元素，也可删除区间范围的元素。clear()函数可以删除 vector 型容器中的所有元素，相当于 erase(begin(), end())。这三个函数的使用方法详见例 3-9。

例 3-9

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
void OutToScreen(int& Ele) //输出元素至屏幕
{cout << Ele << ", ";
}
void main()
```

```

s{vector<int> myvt;
  for(int i=0;i<10;i++)
    myvt.push_back(i);
  for_each(myvt.begin(), myvt.end(), OutToScreen);      //输出 vector 型容器中的元素
  cout<<endl;
  cout<<" - - - - -" <<endl;
  while(!myvt.empty())                                //判断向量是否为空
  {
    myvt.pop_back();                                  //弹出向量
    for_each(myvt.begin(), myvt.end(), OutToScreen);  //输出向量中的元素
    cout<<endl;
  }
  myvt.clear();                                       //清空 vector 型容器
  for(int j=0; j<10; j++)
    myvt.push_back(j);
  for_each(myvt.begin(), myvt.end(), OutToScreen);    //输出
  cout<<endl;
  cout<<" - - - - -" <<endl;
  while(!myvt.empty())                                //判断 vector 型容器是否为空
  {
    myvt.erase(myvt.begin());                        //删除开始第一个元素
    for_each(myvt.begin(), myvt.end(), OutToScreen); //输出 vector 型容器中的元素
    cout<<endl;
  }
}

```

例 3-9 的执行效果如图 3-10 所示。

```

0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
-----
0, 1, 2, 3, 4, 5, 6, 7, 8,
0, 1, 2, 3, 4, 5, 6, 7,
0, 1, 2, 3, 4, 5, 6,
0, 1, 2, 3, 4, 5,
0, 1, 2, 3, 4,
0, 1, 2, 3,
0, 1, 2,
0, 1,
0,
0,
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
-----
1, 2, 3, 4, 5, 6, 7, 8, 9,
2, 3, 4, 5, 6, 7, 8, 9,
3, 4, 5, 6, 7, 8, 9,
4, 5, 6, 7, 8, 9,
5, 6, 7, 8, 9,
6, 7, 8, 9,
7, 8, 9,
8, 9,
9,
Press any key to continue

```

图 3-10 例 3-9 的执行效果

(8) 对象交换

vector 类模板还提供了成员 swap() 函数，以实现两个 vector 型容器之间的元素互换。如果两个参与交换的 vector 类型相同，对象交换会瞬间完成；如果两个参与交换的 vector 对象中元素类型不同，在实现对象交换的过程中，需要执行非常复杂的操作。swap() 函数的原型为：

```
template <class T, class A> void swap (const vector <T,A>&v1, const vector <T,A>&v2;
```

例 3-10

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
void OutToScreenI (int& Ele)
{cout << Ele << ", ";
}
void main()
{vector<int> ci,cd;
for(int i=0;i<10;i++)
{ ci.push_back (i);
cd.push_back (i* 3);
}
cout << " vector - ci -below:" << endl;
for_each (ci.begin(), ci.end(), OutToScreenI);
cout << endl;
cout << " vector - cd -below:" << endl;
for_each (cd.begin(), cd.end(), OutToScreenI);
cout << endl;
cout << " - - - - - swap - - - - -" << endl;
ci.swap (cd);
cout << " vector - ci -below:" << endl;
for_each (ci.begin(), ci.end(), OutToScreenI);
cout << endl;
cout << " vector - cd -below:" << endl;
for_each (cd.begin(), cd.end(), OutToScreenI);
cout << endl;
}
```

例 3-10 的执行效果如图 3-11 所示。

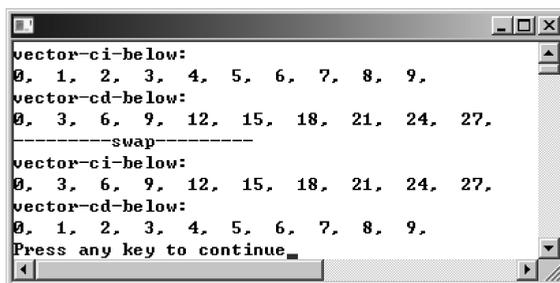


图 3-11 例 3-10 的执行效果

(9) vector<bool> 类

C++ STL 专门针对元素型别为 bool 的 vector 设计了特殊版本, 目的是获取优化的 vector。其所占用的内存空间远小于一般的 vector 模板实例化的 bool 型向量 (容器)。普通的 vector 类模板会分配 1 个 Byte 空间, 而 vector<bool> 只占用 1 个 bit 存储单个元素, 所占内存空间是空间的 1/8。而 C++ 的最小寻址通常以 Byte 为单位, 在类 vector<bool> 中需针对 reference 和 iterator 进行特殊考虑。类 vector<bool> 的操作要比普通的 vector 慢很多, 所有元素操作必须转换为 bit 操作。后面章节还会介绍 bitset 的使用, 程序员应该优先使用 bitset, 而不是 vector<bool>。

vector<bool> 类不仅仅是向量型的容器类, 还提供位操作——非常方便的操作“位”和“标志”。所有用于元素存取的函数, 返回值均为引用型 (reference)。

vector<bool> 类提供了位取反函数 flip(), 可以实现对容器中的所有元素取反, 还可以对某“位”取反。下面以例 3-11 进行说明。

例 3-11

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
void print (bool&Ele)
{
    cout << Ele;
}
void main()
{
    int X[] = {1,0,1,0,0,0,1,1,1};
    vector<bool> vt;
    vector<bool>::iterator it;
    int i = 0;
    for (i = 0; i < 9; i++)
    {
        vt.push_back (bool (X [i]));
    }
    for_each (vt.begin(), vt.end(), print);
    cout << endl;
    vt [3] = bool (1);
    for_each (vt.begin(), vt.end(), print);
    cout << endl;
}
```

例 3-11 的执行效果如图 3-12 所示。



图 3-12 例 3-11 的执行效果



提示 本小节主要讲述了 vector 模板类、vector 模板类（容器）的定义和容量、基本成员函数使用方法以及 vector 容器的高级编程。在学习使用 vector 容器的同时，本小节还讲述了部分算法的学习。

3.2.2 list（列表）类模板

同样，list 模板类也是一个容器。list 是由双向链表来实现的，每个节点存储 1 个元素。list 支持前后两种移动方向。list 和 vector 类似，没有提供对元素的随机访问。list 的优势在于任何位置执行插入和删除动作都非常迅速，因为改变的仅仅是链接而已，所以在 list 中移动元素要比在 vector 和 deque 中快得多。list 模板类是定义于命名空间（name space）std 中的，该类模板的声明形式为：

```
template <class T, class Allocator = allocator<T>> class list;
```

使用 list 需要包含头文件 <list>。前面已经讲过，任意型别 T 只要具备可设置性和可复制性，即可作为 list 元素。模板的第 2 个参数可有可无，适用于指定内存模型。在模板中，第 2 个参数被指定了默认的内存模型为 allocator。list 的内部结构和 vector 不同，存在较明显的区别。

- 1) list 不支持随机存取。
- 2) 在 list 的任何位置执行元素的插入和移除都非常快，可以迅速实现。插入和删除动作不会影响指向其他元素的指针、引用、迭代器，不会造成失效。
- 3) list 不提供下标操作符 [] 和 at() 函数。
- 4) list 没有提供容量、空间重新分配等操作函数，每个元素都有自己的内存。
- 5) list 也提供了特殊成员函数，专门用于移动元素。和同名的算法相比，使用这些函数速度更快。

list 模板类实现了标准 C++ 数据结构中链表的所有功能。一旦 list 定义了类对象，就可以完成链表操作。

下面介绍 list 类模板的使用方法。

1. list 的定义和容量

(1) list 的定义和构造函数

list 模板类描述的对象控制是一个元素类型为 T 的可变长度序列。该序列以双向链表的方式存储。链表中的每个元素都包含一个类型为 T 的成员。list 对象是通过存储于其内部的一个类 A 的分配器对象进行存储空间的分配和释放。该分配器对象必须拥有和模板类 allocator 同样的外部接口。

头文件 list 中定义了四种构造函数。

```
explicit list(const A& _A1 = A()): allocator(_A1), _Head(_Buynode()), _Size(0)
{
}

explicit list(size_type _N, const Ty& _V = Ty(), const A& _A1 = A()): allocator(_A1),
_Head(_Buynode()), _Size(0)
```

```

    {
        insert(begin(), _N, _V);
    }
    list(const_Myt& _X): allocator(_X.allocator), _Head(_Buynode()), _Size(0)
    {
        insert(begin(), _X.begin(), _X.end());
    }
    list(const_Ty* _F, const_Ty* _L, const_A& _Al = _A()): allocator(_Al), _Head(_Buynode()),
    _Size(0)
    {
        insert(begin(), _F, _L);
    }

```

以上四种构造函数中，第 1 个是默认的构造函数，表示要创建空 list 对象。根据以上构造函数，list 对象定义时一般有以下几种方法：

```

list<A>listname;
list<A>listname(size);
list<A>listname(size,value);
list<A>listname(elselist)
list<A>listname(first, last)

```

以上几种定义 list 类型对象的方法中，第一种形式最简单。第一种示例可以创建 1 个空的 list 对象，其中可以容纳类型为 A 的元素，其名称为 listname。例如，

```
list<int>mylist
```

而第二种示例可以创建初始大小为 size 的 list 对象；第三种示例可以创建初始大小为 size，每个元素初始值为 value 的 list 对象；第四种示例用复制构造函数从现有的 list 中创建新的 list 对象；第五种方法创建 1 个 list 对象，并从其他 list 对象中复制由迭代器指定范围的多个元素。

(2) 元素的赋值

list 模板类提供了两个成员函数 `push_front()` 和 `push_back()`，用来把新的元素插入到 list 对象中。`push_front()` 函数用来在容器中的头部插入新元素，由于 `vector` 在头部插入新元素的效率非常低，因此仅在 list 中存在该函数。`push_back()` 函数用于在容器的底部插入新元素。还有另外两个函数 `pop_front()` 和 `pop_back()` 分别用于获取容器的“头”元素和容器的“尾”元素。这 4 个函数的原型分别为：

```

void push_front (const T& x);
void push_back (const T& x);
void pop_front ();
void pop_back ();

```

与 `vector` 类模板不同的是，`vector` 模板类只具有 `push_back()` 和 `pop_back()`；而 list 模板类具有 4 个相应的函数，充分说明 list 类型容器是双向链表。

(3) 容器的容量

list 对象作为容器，和其容量相关的成员函数包括 `resize()`、`size()` 和 `max_size()`。其函

数原型分别为：

```
size_type size() const;
size_type max_size() const;
void resize (size_type n, T x = T());
```

`size()` 和 `max_size()` 函数的返回类型均为 `size_type` 型，即 `unsigned int` 类型。`size()` 函数用于返回 `list` 对象（容器）中元素的个数；`max_size()` 函数用于返回 `list` 对象的最大允许容量（一般是一个非常大整数）。`resize()` 函数用于重新调整 `list` 对象的大小。

（4）迭代器相关

`list` 模板类所包含的和迭代器相关的函数主要有 `begin()`、`front()`、`rbegin()`、`end()`、`back()` 和 `rend()`。这几个函数的原型分别为：

```
const_iterator begin() const;
iterator begin();
reference front();
const_reference front() const;
const_reverse_iterator rbegin() const;
reverse_iterator rbegin();
const_iterator end() const;
iterator end();
reference back();
const_reference back() const;
const_reverse_iterator rend() const;
reverse_iterator rend();
```

下面以例 3-12 说明以上 4 个知识点的使用方法。通过实例学习，希望读者对 `list` 类型容器的最基本知识有所了解，理解 `list` 的定义、容量、赋值方法以及该容器中和迭代器相关的成员函数。

例 3-12

```
#pragma warning(disable:4786)
#include <iostream>
#include <list>
#include <algorithm>
#include <string>
//#include <iomanip.h>
using namespace std;
template <class T> void print(const T&Ele)
{ cout << " " << Ele << "; " << endl;
}
void Print_D (double&Ele) //格式化输出
{ cout.width(5); //格式化输出，宽度为 5 个字符
  cout.precision(1); //保留 1 位小数点
  cout << std::fixed << Ele << ", "; //以定点数形式输出
}
```

```
void Print_I (int&Ele)
{   cout << Ele << ",   ";
}
```

注意：请关注第 1 行和 3 个不同的 print() 函数的使用方法。

```
void main()
{   list<string>mylist_string;
    list<double>mylist_double (6);
    //上述是两种 list 型容器的定义形式
    // - - - - - 初始化 mylist_string
    mylist_string.push_front (" 1:   Jack");
    mylist_string.push_front (" 2:   Tom");
    mylist_string.push_front (" 3:   Mike");
    // - - - - - 初始化 mylist_double
    mylist_double.push_front (10.0);
    mylist_double.push_front (20.0);
    mylist_double.push_front (30.0);
    mylist_double.push_front (40.0);
    mylist_double.push_front (50.0);
    //下述是 3 种容器的定义形式
    list<int>mylist_int (6, 0);
    list<double>mylist_double2 (6, 0.0);
    list<int>elselist (mylist_int);
    list<double>Iterlist (mylist_double.begin(), mylist_double.end());
    // - - - 输出各个容器中的元素
    cout << " the string list: mylist_string is below:" << endl;
    list<string>:: iterator iter_String;
    for (iter_String=mylist_string.begin(); iter_String! =mylist_string.end(); iter_String
++)
    {   string temp=* iter_String;
        print (temp);
    }
    cout << " the double list - mylist_double is below:" << endl;
    for_each (mylist_double.begin(), mylist_double.end(), Print_D);
    cout << endl;
    cout << " the double list - mylist_double2 is below:" << endl;
    for_each (mylist_double2.begin(), mylist_double2.end(), Print_D);
    cout << endl;
    cout << " the double list - Iterlist is below:" << endl;
    for_each (Iterlist.begin(), Iterlist.end(), Print_D);
    cout << endl;
    cout << " the int list - mylist_int is below:" << endl;
    for_each (mylist_int.begin(), mylist_int.end(), Print_I);
    cout << endl;
    cout << " the int list - elselist is below:" << endl;
    for_each (elselist.begin(), elselist.end(), Print_I);
```

```
cout << endl;
//各容器的容量
int size = mylist_string.size();
int maxsize = mylist_string.max_size();
mylist_string.resize(5);
size = mylist_double.size();
maxsize = mylist_double.max_size();
mylist_double.resize(5);
size = mylist_double2.size();
maxsize = mylist_double2.max_size();
mylist_double2.resize(5);
size = Iterlist.size();
maxsize = Iterlist.max_size();
Iterlist.resize(5);
size = mylist_int.size();
maxsize = mylist_int.max_size();
mylist_int.resize(5);
size = elselist.size();
maxsize = elselist.max_size();
elselist.resize(5);
//再次输出各个容器中的元素
cout << " the string list: mylist_string is below:" << endl;
for (iter_String = mylist_string.begin(); iter_String != mylist_string.end(); iter_String
++)
{
    string temp = *iter_String;
    print(temp);
}
cout << " the double list - mylist_double is below:" << endl;
for_each(mylist_double.begin(), mylist_double.end(), Print_D);
cout << endl;
cout << " the double list - mylist_double2 is below:" << endl;
for_each(mylist_double2.begin(), mylist_double2.end(), Print_D);
cout << endl;
cout << " the double list - Iterlist is below:" << endl;
for_each(Iterlist.begin(), Iterlist.end(), Print_D);
cout << endl;
cout << " the int list - mylist_int is below:" << endl;
for_each(mylist_int.begin(), mylist_int.end(), Print_I);
cout << endl;
cout << " the int list - elselist is below:" << endl;
for_each(elselist.begin(), elselist.end(), Print_I);
cout << endl;
//使用迭代器相关的函数
list<double>::iterator Iter_D;
list<double>::reverse_iterator Iter_rD;
```

```

double tmp = 0.0;
Iter_D = mylist_double.begin();
tmp = *Iter_D;
cout << " The beginning element of the mylist_double:" << endl;
cout << tmp << endl;
Iter_rD = mylist_double.rbegin();
tmp = *Iter_rD;
cout << " The reverse beginning element of the mylist_double:" << endl;
cout << tmp << endl;
Iter_D = mylist_double.end();
tmp = *Iter_D;
cout << " The ending element of the mylist_double:" << endl;
cout << cout.scientific << tmp << endl;
Iter_rD = mylist_double.rend(); //
tmp = *Iter_rD;
cout << " The reverse ending element of the mylist_double:" << endl;
cout << tmp << endl; //
tmp = mylist_double.front();
cout << " The front element of the mylist_double:" << endl;
cout << tmp << endl; //
tmp = mylist_double.back();
cout << " The back element of the mylist_double:" << endl;
cout << tmp << endl;
}

```

请关注例 3-12 源代码中的中文注释。例 3-12 还使用了模板函数和仿函数以及 `for_each()` 算法函数，还对 `list` 容器的大小进行了重新设置。为防止程序编译时显示过多的警告信息，在程序起始位置添加了“`#pragma warning (disable: 4786)`”语句。读者应对照源代码和程序输出结果，认真理解其用法。



提示 ① 例 3-12 中，由于 `end()` 函数所指向的是容器中最后一个元素的后面的位置，因此程序中输出 `end()` 返回的数值时，并不是最后一个元素的数值。`rend()` 函数同理。由此想到，在使用迭代器进行 `for` 循环时，一般循环终止条件是 `(iterator! = end())`，即循环至 `end()` 返回值指向的位置时（恰好是容器中最后一个元素的后面），停止循环。

② `Print_D (double& Ele)` 函数中，使用了 `cout` 的格式化输出。

例 3-12 的执行效果如图 3-13 所示。

2. list 容器的基本成员函数

`list` 容器最基本的应用包括判断 `list` 内容是否为空、元素的存取和访问、元素重置、交换两个 `list` 型容器的内容、元素的插入和删除等。

(1) 判断容器是否为空

若 `list` 型容器中为空，则成员函数 `empty()` 返回 `true`。`empty()` 函数的原型为：

```
bool empty() const;
```


list 型容器不提供成员函数 at() 和操作符 [], 这对容器中的元素访问无疑是不便的。值得庆幸的是, 还可以使用迭代器进行元素的访问。例如,

例 3-14

```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;
void print(double&Ele)
{ cout << Ele << " , ";
}
void main()
{ list<double>mylist;
  mylist.push_back(11.1);
  mylist.push_back(21.5);
  mylist.push_back(31.6);
  mylist.push_back(41.7);
  int count=mylist.size();
  for_each(mylist.begin(), mylist.end(), print);
  cout << endl;
  list<double>::iterator Iter_S;
  Iter_S=mylist.begin();
  cout << " The third element is " << *(++ (++ (++ Iter_S))) << endl;
}
```

例 3-14 的执行效果如图 3-14 所示。

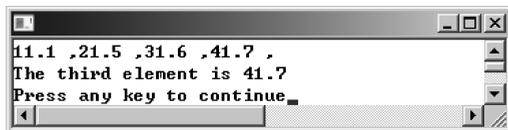


图 3-14 例 3-14 的执行效果

(3) 元素重置

list 型容器提供了可重置元素值的成员函数 assign()。使用 assign() 函数可以修改容器中任意元素的数值, 甚至可以修改多个连续元素的数值。assign() 函数的原型为:

```
void assign(const_iterator first, const_iterator last);
void assign (size_type n, const T& x = T());
```

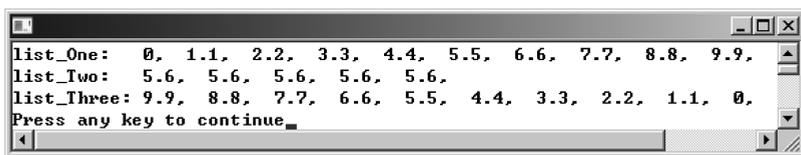
下面举例说明其使用方法。

例 3-15

```
#include <iostream>
#include <list>
using namespace std;
template<class T> void print(list<T>& mylist)
{
```

```
list<T>::iterator Iter;
mylist.reverse();
for(Iter=mylist.begin();Iter!=mylist.end();Iter++)
{ cout<<*Iter<<" ";
}
cout<<endl;
}
void main()
{
list<double> list_One, list_Two, list_Three;
double Ele=0.0;
for (int i=0; i<10; i++)
{ Ele=i+i/10.0;
list_One.push_front (Ele);
}
cout<<" list_One: ";
print (list_One);
list_Two.assign (5, 5.6);
cout<<" list_Two: ";
print (list_Two);
list_Three.assign (list_One.begin(), list_One.end());
cout<<" list_Three: ";
print (list_Three);
}
```

例 3-15 涉及了 `assign()` 函数的两种用法。在需要初始化 `list` 型容器或者复制 `list` 型容器中的元素时，使用 `assign()` 函数是非常方便的。例 3-15 的执行效果如图 3-15 所示。



```
list_One: 0. 1.1. 2.2. 3.3. 4.4. 5.5. 6.6. 7.7. 8.8. 9.9.
list_Two: 5.6. 5.6. 5.6. 5.6. 5.6.
list_Three: 9.9. 8.8. 7.7. 6.6. 5.5. 4.4. 3.3. 2.2. 1.1. 0.
Press any key to continue
```

图 3-15 例 3-15 的执行效果



提示 学习 `assign()` 函数的使用方法时，请关注函数模板 `print()`。关注一下即可，后面会专门讲解。

(4) 交换两个 `list` 型容器的内容

`list` 模板类提供了成员 `swap()` 函数，可用以实现两个 `list` 型容器（对象）的内容交换。通过 `swap()` 函数完成两个 `list` 型容器中内容交换的同时，两个参与交换的容器大小也发生变化。STL 的算法库也提供了 `swap()` 函数。作为通用算法，`swap()` 函数同样可以实现两个 `list` 型容器中的内容交换。如例 3-16 所示，在例 3-15 的基础上进行适当的修改，添加了用于容器内容交换的代码。

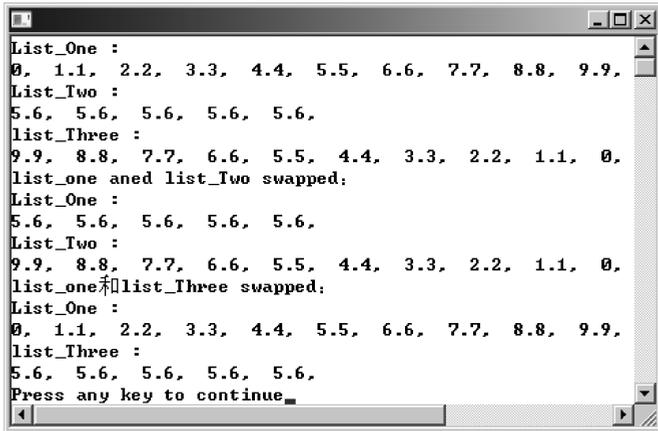
例 3-16

```
#include <iostream>
#include <list>
using namespace std;
template <class T> void print(list <T> &mylist)
{
    list<T>::iterator Iter;
    mylist.reverse();
    for(Iter=mylist.begin();Iter!=mylist.end();Iter++)
    {
        cout << *Iter << ", ";
    }
    cout << endl;
}
void main()
{
    list<double> list_One, list_Two, list_Three;
    double Ele=0.0;
    for (int i=0; i<10; i++)
    {Ele=i+i/10.0;
        list_One.push_front (Ele);
    }
    cout << " List_One : " << endl;
    print (list_One);
    list_Two.assign (5, 5.6);
    cout << " List_Two : " << endl;
    print (list_Two);
    list_Three.assign (list_One.begin(), list_One.end());
    cout << " list_Three : " << endl;
    print (list_Three);
    list_One.swap (list_Two);
    cout << " list_one and list_Two swapped:" << endl;
    cout << " List_One : " << endl;
    print (list_One);
    cout << " List_Two : " << endl;
    print (list_Two);
    swap (list_One, list_Three);
    cout << " list_one 和 list_Three swapped:" << endl;
    cout << " List_One : " << endl;
    print (list_One);
    cout << " list_Three : " << endl;
    print (list_Three);
}
```

例 3-16 的执行效果如图 3-16 所示。

(5) 元素的插入和删除

list 模板类和其他型容器一样，提供了丰富而灵活的插入和删除元素操作函数。list 型容器甚至可以在序列的开头和队尾灵活地插入和删除元素。涉及的成员函数主要包括 push_



```
List_One :
0, 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9,
List_Two :
5.6, 5.6, 5.6, 5.6, 5.6,
list_Three :
9.9, 8.8, 7.7, 6.6, 5.5, 4.4, 3.3, 2.2, 1.1, 0,
list_one and list_Two swapped:
List_One :
5.6, 5.6, 5.6, 5.6, 5.6,
List_Two :
9.9, 8.8, 7.7, 6.6, 5.5, 4.4, 3.3, 2.2, 1.1, 0,
list_one和list_Three swapped:
List_One :
0, 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9,
list_Three :
5.6, 5.6, 5.6, 5.6, 5.6,
Press any key to continue_
```

图 3-16 例 3-16 的执行效果

`back()`、`push_front()`、`pop_back()`、`pop_front()`、`insert()`、`erase()`和`clear()`。`push_front()`和`push_back()`均可以轻松地元素插入至序列中，`pop_back()`和`pop_front()`亦可以轻松地首端或尾端将元素从序列中移除。这4个函数在前面章节已有涉及，此处不再赘述。本小节重点介绍`insert()`、`erase()`和`clear()`函数。

成员函数`insert()`的原型为如下3种型式：

```
iterator insert(iterator it, const T& x = T ());
void insert(iterator it, size_type n, const T& x) ;
void insert (iterator it, const_iterator first, const_iterator last);
```

第一种形式的作用是把某个元素插入到指定位置；第二种形式的作用是把某个具体值的多个备份插入到`list`中迭代器所指的起始位置；第三种形式的作用是把指定范围的多个元素插入到`list`中迭代器所指的范围中。该成员函数的使用方法和`vector`型容器的同名成员函数类似。

成员函数`erase()`的原型为如下两种型式：

```
iterator erase(iterator it);
iterator erase(iterator first, iterator last);
```

同样，第一种形式的作用是删除迭代器指向的元素；第二种形式的作用是删除迭代器所定义的范围。该函数的使用方法和`vector`型容器的同名成员函数近似。

成员函数`clear()`相当于使用`erase()`函数删除序列中的所有元素，即`erase (begin(), end())`。修改例3-9，使之适用于`list`型容器。

例 3-17

```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;
void OutToScreen (int& Ele)
```

```

{   cout << Ele << ", ";
}

void main()
{   list<int> mylt;                               //定义容器型对象
    for(int i=0;i<10;i++)
        mylt.push_back(i);                       //添加元素
    for_each(mylt.begin(),mylt.end(),OutToScreen); //输出容器中的元素
    cout<<endl;
    cout<<"-----" <<endl;
    while(!mylt.empty())                          //每次从尾部取出一个元素,之后输出,直至为空。
        {mylt.pop_back();
        for_each(mylt.begin(),mylt.end(),OutToScreen);
        cout<<endl;
        }
    mylt.clear();                                 //清空容器
    for(int j=0;j<10;j++)                         //重新添加元素
        mylt.push_back(j);
    for_each(mylt.begin(),mylt.end(),OutToScreen); //输出容器中的所有元素
    cout<<endl;
    cout<<"-----" <<endl;
    while(!mylt.empty())                          //删除容器中的第一个元素,输出剩余元素,直至为空
        {mylt.erase(mylt.begin());
        for_each(mylt.begin(),mylt.end(),OutToScreen);
        cout<<endl;
        }
}

```

例 3-17 的执行效果如图 3-17 所示。

```

0. 1. 2. 3. 4. 5. 6. 7. 8. 9.
-----
0. 1. 2. 3. 4. 5. 6. 7. 8.
0. 1. 2. 3. 4. 5. 6. 7.
0. 1. 2. 3. 4. 5. 6.
0. 1. 2. 3. 4. 5.
0. 1. 2. 3. 4.
0. 1. 2. 3.
0. 1. 2.
0. 1.
0.
-----
0. 1. 2. 3. 4. 5. 6. 7. 8. 9.
-----
1. 2. 3. 4. 5. 6. 7. 8. 9.
2. 3. 4. 5. 6. 7. 8. 9.
3. 4. 5. 6. 7. 8. 9.
4. 5. 6. 7. 8. 9.
5. 6. 7. 8. 9.
6. 7. 8. 9.
7. 8. 9.
8. 9.
9.

```

图 3-17 例 3-17 的执行效果

 **总结** 例 3-17 是在例 3-9 的基础上修改而来的（仅将头文件 `<vector>` 修改为 `<list>`，将容器的定义语句 `vector<int> myvt` 修改为 `list<int> mylt`）。请读者关注上述代码中的黑体字，并重点体会 `pop_front()`、`pop_back()`、`erase()` 和 `clear()` 的使用方法。

3. 运算符函数

同样，`list` 型容器也提供了大量的函数模板，即提供了大量运算符函数。常见的运算符函数包括 `operator ==`、`operator <`、`operator !=`、`operator <=`、`operator >`、`operator >=` 等。

(1) `operator ==`

`operator []` 主要用于读取 `list` 中的元素；而 `operator ==` 则是判断语句，用于判断两个 `list` 型容器是否相等。如果相同，返回 `true`；否则，返回 `false`。其原型为：

```
bool operator == (const list<Type, Allocator>& _Left, const list<Type, Allocator>& _Right);
```

上述代码表明，参与比较的两个 `list` 对象的格式应该完全一样，不能使用两个类型不同的容器进行比较。

(2) `operator <`

`operator <` 用于判断两个 `list` 型容器是否“前者小于后者”。如果“前者小于后者”，返回 `true`；否则，返回 `false`。其原型为：

```
bool operator < (const list<Type, Allocator>& _Left, const list<Type, Allocator>& _Right);
```

(3) `operator !=`

`operator !=` 用于判断两个 `list` 型容器是否“不相等”。如果两者不同，返回 `true`；否则，返回 `false`。其原型为：

```
bool operator != (const list<Type, Allocator>& _Left, const list<Type, Allocator>& _Right);
```

(4) `operator <=`

`operator <=` 用于判断两个 `list` 型容器是否“前者小于或等于后者”。如果“前者小于或等于后者”，返回 `true`；否则，返回 `false`。其原型为：

```
bool operator <= (const list<Type, Allocator>& _Left, const list<Type, Allocator>& _Right);
```

(5) `operator >`

`operator >` 用于判断两个 `list` 型容器是否“前者大于后者”，如果“前者大于后者”，返回 `true`；否则，返回 `false`。其原型为：

```
bool operator > (const list<Type, Allocator>& _Left, const list<Type, Allocator>& _Right);
```

(6) `operator >=`

`operator >=` 用于判断两个 `list` 型容器是否“前者大于或等于后者”。如果“前者大于后者”，运算符返回 `true`；否则，返回 `false`。其原型为：

```
bool operator >= (const list<Type, Allocator>& _Left, const list<Type, Allocator>& _Right);
```

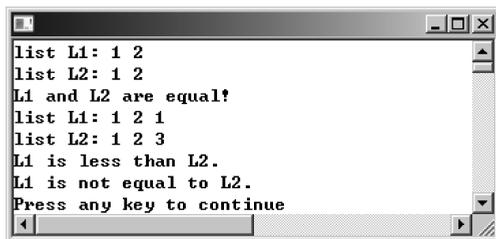
下面使用例 3-18 说明以上 6 个运算符函数的使用方法。

例 3-18

```
#include <iostream>
#include <list>
```

```
#include <algorithm>
using namespace std;
void print(int&Ele)
{   cout <<Ele <<" ";
}
void main()
{   list<int> L1,L2;
    L1.push_back (1);
    L1.push_back (2);
    L2.assign (L1.begin(), L1.end());
    cout<<" list L1: ";
    for_each (L1.begin(), L1.end(), print);
    cout<<endl;
    cout<<" list L2: ";
    for_each (L2.begin(), L2.end(), print);
    cout<<endl;
    if (L1 ==L2)
        cout <<" L1 and L2 are equal!" <<endl;
    L2.push_back (3);
    L1.push_back (1);
    cout<<" list L1: ";
    for_each (L1.begin(), L1.end(), print);
    cout<<endl;
    cout<<" list L2: ";
    for_each (L2.begin(), L2.end(), print);
    cout<<endl;
    if (L1 <L2)
        cout <<" L1 is less than L2. " <<endl;
    else if (L1 >L2)
        cout <<" L1 is greater than L2. " <<endl;
    if (L1 !=L2)
        cout <<" L1 is not equal to L2. " <<endl;
}
```

例 3-18 的执行效果如图 3-18 所示。



```
list L1: 1 2
list L2: 1 2
L1 and L2 are equal!
list L1: 1 2 1
list L2: 1 2 3
L1 is less than L2.
L1 is not equal to L2.
Press any key to continue
```

图 3-18 例 3-18 的执行效果

4. 其他重要成员函数

list 型容器具有一些特殊函数, 如 merge()、remove()、remove_if()、sort()、splice()

和 `unique()`。

(1) `merge()` 函数和 `sort()` 函数

`merge()` 函数可以将两个 `list` 型对象合并成一个 `list` 对象。该函数的功能在于把原型中的 `list` 型容器对象作为函数参数，插入到目标 `list`（即函数的调用者）中。合并之后的容器中元素是按从小到大升序排列的。其原型为：

```
void merge(list& x);  
void merge(list& x, greater<T> pr);
```

`list` 型容器还提供了具有排序功能的成员函数 `sort()`，用于对 `list` 型容器中的元素进行排序。`sort()` 函数默认的排序方式是从小到大。其原型为：

```
void sort() //从小到大排序  
void sort(greater<T> pr) //从大到小排序
```

以上两个函数的具体使用方法参见例 3-19。

例 3-19

```
#include <iostream>  
#include <list>  
#include <algorithm>  
using namespace std;  
void print(int&Ele)  
{ cout << Ele << " ";  
}  
void main()  
{ list<int> L1,L2,L3;  
list<int>::iterator I1,I2,I3;  
L1.push_back(1);  
L1.push_back(5);  
L2.push_back(2);  
L2.push_back(3);  
L3.push_back(7);  
L3.push_back(8);  
cout << " L1 : ";  
for_each(L1.begin(), L1.end(), print); //输出 L1 中的内容  
cout << endl;  
cout << " L2 : ";  
for_each(L2.begin(), L2.end(), print); //输出 L2 中的内容  
cout << endl;  
cout << " L3 : ";  
for_each(L3.begin(), L3.end(), print); //输出 L3 中的内容  
cout << endl;  
cout << " L1 merges L2 and L3 :";  
L1.merge(L2); //合并 L1 和 L2  
L1.merge(L3); //合并 L1 和 L3  
for_each(L1.begin(), L1.end(), print); //可知，在 list 合并之后，所有元素自动按从小到大排序
```

```

    cout << endl;
    L1.sort( greater<int>() ); //降序排序
    cout << "L1 : ";
    for_each( L1.begin(), L1.end(), print ); //所有元素输出 (降序)
    cout << endl;
    L1.sort(); //默认按从小到大排序
    cout << " L1 : ";
    for_each( L1.begin(), L1.end(), print ); //所有元素自动按从大到小排序
    cout << endl;
}

```

在上述代码中，两个 list 型容器合并之后，新容器中的元素是自动排序的。

```
L1.sort( greater<int>() )
```

上述语句实现了对容器中的元素降序排列。函数的参数“greater<int>()”是 STL 中的预定义仿函数。List 型容器的成员函数 sort() 默认的排序方式是升序。

例 3-19 的执行效果如图 3-19 所示。



```

L1 : 1 5
L2 : 2 3
L3 : 7 8
L1 合并 L2 和 L3 : 1 2 3 5 7 8
L1 <从大到小排序>: 8 7 5 3 2 1
L1 <从小到大排序>: 1 2 3 5 7 8
Press any key to continue

```

图 3-19 例 3-19 的执行效果

(2) remove() 函数和 remove_if() 函数

删除 list 中的对象可以使用 pop_back()、pop_front()、erase()、clear() 等常见函数。List 型容器还提供了 remove() 函数，其原型为：

```
void remove(const Type& _Val);
```

该函数可以删除 list 型容器中某个具体的元素。使用 remove() 函数不需要指定具体位置，只要告诉需要删除的元素的值，就可以直接删除所有相应的元素。这是和其他删除函数不同的地方。值得注意的是，使用 remove() 函数并不改变容器中元素的顺序。同时，由于在使用 remove() 函数时，会删除掉所有等于参数值（指定数值_Val）的元素，程序员要谨慎使用。

remove_if() 函数是有条件地删除所有等于参数值的 list 型容器中的元素。其原型为：

```
template <class Pred> void remove_if (Pred pr)
```

remove_if() 函数仅删除满足条件“Pred pr”的相应元素。“Pred pr”参数是一元谓词。在 Visual C++ 6.0 中，Pred 的定义如下：

```
typedef binder2nd <not_equal_to<_Ty>> Pred;
```

只有当 Pred pr 为 true 时, remove_if() 函数才能正确执行删除操作。在 visual C++ 6.0 编译环境下, 该函数的执行效果是删除和指定参数 pr 不相等的元素。请读者参考例 3-20, 仔细体会这两个删除函数的用法。

例 3-20

```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;
void print (int&Ele)
{ cout << Ele << ", ";
}
bool is_Even (int &Ele)
{ return (Ele%2 ==1);
}
void Origin (list<int>& L, int num)
{ int temp;
  L.clear();
  for (int i=0; i<num; i++)
  { temp=i+1;
    L.push_back (temp);
  }
  for_each (L.begin(), L.end(), print);
  cout << endl;
}
void main()
{ list<int> L1;
  Origin (L1, 9);
  int temp;
  temp=9;
  L1.push_back (temp);
  temp=8;
  L1.push_back (temp);
  cout << " Ouput the list \'L1 \': " << endl;
  for_each (L1.begin(), L1.end(), print);
  cout << endl;
  L1.remove (9);
  for_each (L1.begin(), L1.end(), print);
  cout << endl;
  L1.remove_if (bind2nd (not_equal_to<int> (), 1));
  for_each (L1.begin(), L1.end(), print);
  cout << endl;
}
```

例 3-20 的执行效果如图 3-20 所示。

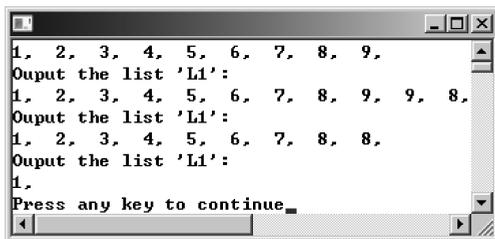


图 3-20 例 3-20 的执行效果



提示 学习 `remove()` 函数和 `remove_if()` 函数的同时, 请读者详读这两个函数的功能。最重要的是关注成员函数 `remove_if()` 的使用。这不同于算法 `remove_if()` 的使用, 算法 `remove_if()` 在使用时, 要求更宽泛一些。

(3) `splice()` 函数

前面讲述了容器合并成员函数 `merge()`。该函数使用起来并不是非常灵活, 反而具有一定的局限性。list 型容器提供了另一个函数 `splice()`。其原型为:

```
void splice(iterator it, list& x);
void splice(iterator it, list& x, iterator first);
void splice(iterator it, list& x, iterator first, iterator last);
```

第一种形式既可以把 list 型对象 `x` 插入在迭代器指针 `it` 指定的位置后面; 第二种形式可以将 `x` 中的某个元素 (`first` 指向的元素) 插入 `it` 的后面; 第三种形式可以将 `x` 中某个范围 (`first`, `last`) 内的元素插入在 `it` 后面。值得注意的是, 一旦合并完成, 参数 `x` 中会减少相应数目的元素。因为这些元素已经转移走了。尤其是第一种形式, 一旦合并函数 `splice()` 执行成功之后, 参数 `x` 中将不包含任何元素——空容器。在例 3-19 的基础进行修改, 使其可以体现 `splice()` 函数的用法。

例 3-21

```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;
void print(int&Ele)
{ cout << Ele << " ";
}
void main()
{ list<int> L1,L2,L3,L0;
  L1.push_back(1);
  L1.push_back(5);
  L2.push_back(2);
  L2.push_back(3);
  L3.push_back(7);
  L3.push_back(8);
```

```
L0.push_back (9);
L0.push_back (-1);
cout << " L1 : ";
for_each (L1.begin(), L1.end(), print);
cout << endl;
cout << " L2 : ";
for_each (L2.begin(), L2.end(), print);
cout << endl;
cout << " L3 : ";
for_each (L3.begin(), L3.end(), print);
cout << endl;
cout << " L0 : ";
for_each (L0.begin(), L0.end(), print);
cout << endl;
cout << " L1 合并 L2:";
L1.splice (L1.end(), L2);
for_each (L1.begin(), L1.end(), print);
cout << endl;
cout << " L2 : ";
for_each (L2.begin(), L2.end(), print);
cout << endl;
cout << " L1 合并 L0 :";
L1.splice (L1.end(), L0, (++L0.begin()));
for_each (L1.begin(), L1.end(), print);
cout << endl;
cout << " L0 : ";
for_each (L0.begin(), L0.end(), print);
cout << endl;
cout << " L1 合并 L3 :";
L1.splice (L1.end(), L3, L3.begin(), L3.end());
for_each (L1.begin(), L1.end(), print); //可知, 在合并之后, 所有元素自动按从小到大排序
cout << endl;
cout << " L3 : ";
for_each (L3.begin(), L3.end(), print);
cout << endl;
L1.sort (greater <int > ()); //所有元素自动按从大到小排序
cout << " L1 (从大到小排序): ";
for_each (L1.begin(), L1.end(), print);
cout << endl;
L1.sort (); //默认按从小到大排序
cout << " L1 (从小到大排序): ";
for_each (L1.begin(), L1.end(), print); //所有元素自动按从大到小排序
cout << endl;
}
```

例 3-21 的执行效果如图 3-21 所示。

```

L1 : 1 5
L2 : 2 3
L3 : 7 8
L0 : 9 -1
L1 合并 L2:1 5 2 3
L2 :
L1 合并 L0 :1 5 2 3 -1
L0 : 9
L1 合并 L3 :1 5 2 3 -1 7 8
L3 :
L1 <从大到小排序>: 8 7 5 3 2 1 -1
L1 <从小到大排序>: -1 1 2 3 5 7 8
Press any key to continue

```

图 3-21 例 3-21 的执行效果



提示 学习 list 型容器的成员函数 splice() 时, 要关注合并以后, 参数 x 表示的容器中的元素数目。同时应该注意对于不同的编译系统, 对于 STL 的容器模板可能略有差别。

(4) unique() 函数

对于 list 型容器中存储的元素, 可能存在连续的多个元素数值相等的情况, 使用 unique() 函数会移除重复元素, 仅留下 1 个。其原型有如下两种形式:

```
void unique();
template < class BinaryPredicate > void unique( BinaryPredicate _Pred );
```

splice() 函数假定容器中元素是已排序的, 因此所有相同的元素都是相邻的。不相邻的重复元素不能被移除。对于第二种函数形式, 只有当元素满足条件表达式 _Pred 时, 该元素才被删除。由以上可知, unique() 函数并不能保证序列中元素值的唯一性, 而仅仅是将相邻的重复元素保留一个。该函数的第二种原型: template < class BinaryPredicate > void unique (BinaryPredicate _Pred), 其中参数 BinaryPredicate _Pred 在 Visual C++ 6.0 中其具体形式如下:

```
not_equal_to < _Ty > _Pr2
```

所以, 第二种形式所实现的功能是仅保留和第一个元素相等的元素, 参数仅提供“函数”, 即仅提供谓词。此时参数的实例化问题较突出。和前面的 remove_if() 函数有相似之处。

例 3-22

```

#include <iostream>
#include <list>
#include <algorithm>
using namespace std;
void Print(int&Ele)
{ cout << Ele << " ";
}
void main()
{ list <int> L1, L2;

```

```

not_equal_to <int > Pred;
L1.push_back (1);
L1.push_back (2);
L1.push_back (3);
L1.push_back (1);
L1.push_back (2);
L1.push_back (3);
L1.push_back (3);
L1.push_back (5);
L1.push_back (7);
L2.assign (L1.begin(), L1.end());
for_each (L1.begin(), L1.end(), Print);
cout << endl;
for_each (L2.begin(), L2.end(), Print);
cout << endl;
L1.sort ();
L1.unique ();
for_each (L1.begin(), L1.end(), Print);
cout << endl;
L2.sort ();
L2.unique (Pred);
for_each (L2.begin(), L2.end(), Print);
cout << endl;
}

```

例 3-22 的执行效果如图 3-22 所示。

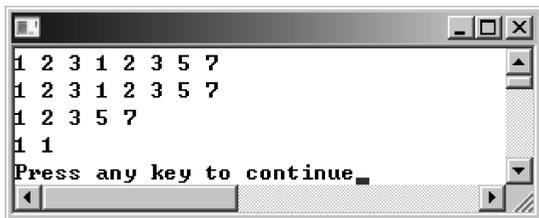


图 3-22 例 3-22 的执行效果



提示

- ① 使用 list 型容器的成员函数 unique() 时，尽量先使用 sort() 函数对容器中的元素进行排序；
- ② 使用 unique() 函数的第二种形式时，参数仅是“谓词”，而不提供具体的数值！

(5) reverse() 函数

reverse() 函数可将容器中的所有元素与原来相反的顺序排列。其原型为：

```
void reverse ();
```

在使用 reverse() 函数时，不需要任何参数，仅调用即可。容器中的所有元素会按与原来相反的顺序排列。

总结 这里再次对 list 模板类做简要说明。list 型容器是非常重要的—种模板类。遗憾的是，它没有提供操作符 [] 和成员函数 at()。list 型容器提供了很多可实现序列操作功能的函数。在学习以上内容的时候，读者应注意反复多读几遍，以充分掌握这些知识点。

3.2.3 deque (双端队列) 类模板

“deque”是简写形式，其原意为“double-ended queue”。

deque 模板类提供了对序列随机访问的功能，可以实现在序列两端快速插入和删除操作的功能，在需要时修改自身大小。deque 型容器是典型的双端队列，可以完成标准 C++ 数据结构中队列的所有功能。

deque 型容器采用动态数组来管理序列中的元素，提供随机存取，和 vector 具有几乎类似的接口。模板类 deque 最重要的特征是在 deque 两端高效地放置元素和删除元素，其原因在于 deque 型序列开放了序列的两端，即头尾均开放，可以快速地在序列两端进行快速插入和删除操作。当需要向序列两端频繁地插入或删除数据元素时，最佳的容器就是 deque。

采用动态数组，deque 型容器具有其优势的同时，必然存在其局限性：当在 deque 型序列中间插入元素时，是非常费时费力的，必须移动其他元素。

deque 型容器的数据结构逻辑图如图 3-23 所示。



图 3-23 deque 型容器的数据结构逻辑图

1. deque 型容器和 vector 型容器的对比

deque 型容器和 vector 型容器相比，具有如下优越之处。

- 1) deque 可以在两端迅速插入和删除元素，而 vector 只提供了成员函数 push_back() 和 pop_back()。
- 2) 存取元素时，deque 型容器会稍慢一些。
- 3) deque 型容器的迭代器是智能型指针。
- 4) 在内存区块受限制的系统中，deque 型容器可以包含更多元素，因为它使用了多块内存。
- 5) deque 型容器不支持对容器和内存重分配时机的控制。
- 6) deque 的内存区块不使用时，会被释放。
- 7) 在序列中间插入和删除元素时，deque 的速度很慢，需要移动相关的所有元素。
- 8) deque 型容器的迭代器属于随机存取迭代器。

2. deque 型容器的定义和容量

(1) deque 定义

deque 是动态数组，可以向两端发展。在 STL 库的头文件 <deque> 中定义了 4 种构造函数：

```
explicit deque(const A& a1 = A());
explicit deque(size_type n, const T& v = T(), const A& a1 = A());
deque(const deque& x);
deque(const_iterator first, const_iterator last, const A& a1 = A());
```

第一种形式构造器定义了一种空的可控序列；第二种形式构造器定义了 n 个相同的元素 x ；第三种形式构造器通过另一个 deque 型容器定义，实现从另一个 deque 型容器复制到新建容器；第四种形式构造器通过另一个容器的某范围内元素，创建新容器。

根据以上 4 种构造函数，可以有以下型式用于定义 deque 对象：

```
deque < typename T > name;
deque < typename T > name (size);
deque < typename T > name (size, value);
deque < typename T > name (elsedeque);
deque < typename T > name (elsedeque. first(), elsedeque. end());
```

上述 5 种型式，第 1 种方法创建容纳类型 T 的空 deque 容器对象；第 2 种方法创建初始大小为 $size$ 的 deque 对象；第 3 种方法创建初始大小为 $size$ ，每个元素初始值为 $value$ 的 deque 容器对象；第 4 种方法用复制构造函数从现有的 deque 型容器 $elsedeque$ 中创建新的 deque 容器对象；第 5 种方法使用迭代器创建 deque 容器对象。

(2) 容量

deque 为度量数据成员数量和容器的大小，提供了以下几种方法。

`resize (size_type n, T x = T())`：把 deque 容器对象的大小重新调整为 n ，并把对象中的新元素初始化为 x 。

`max_size()`：返回 deque 容器对象中的最大允许值。

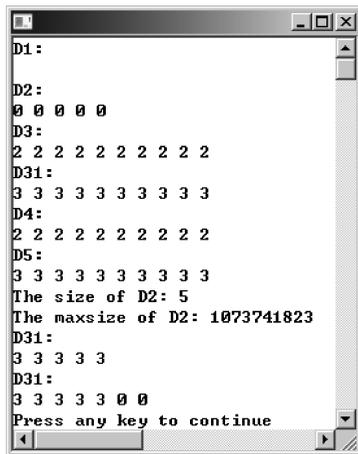
`size()`：返回 deque 容器对象的大小，即容器中元素的个数。

例 3-23

```
#include < iostream >
#include < deque >
#include < algorithm >
using namespace std;
void Print (int&Ele)
{ cout << Ele << " ";
}
void main ()
{ deque < int > D1;
  deque < int > D2 (5);
  deque < int > D3 (10, 2), D31 (10, 3);
  deque < int > D4 (D3);
  deque < int > D5 (D31. begin(), D31. end());
  cout << "D1:" << endl;
  for_each (D1. begin(), D1. end(), Print);
  cout << endl;
  cout << " D2:" << endl;
  for_each (D2. begin(), D2. end(), Print);
  cout << endl;
  cout << " D3:" << endl;
  for_each (D3. begin(), D3. end(), Print);
  cout << endl;
```

```
cout << "D31:" << endl;
for_each (D31.begin(), D31.end(), Print);
cout << endl;
cout << " D4:" << endl;
for_each (D4.begin(), D4.end(), Print);
cout << endl;
cout << " D5:" << endl;
for_each (D5.begin(), D5.end(), Print);
cout << endl;
int size = D2.size();
cout << " The size of D2: " << size << endl;
intMsize = D2.max_size();
cout << " The maxsize of D2: " << Msize << endl;
D31.resize (5, 'A');
cout << " D31:" << endl;
for_each (D31.begin(), D31.end(), Print);
cout << endl;
D31.resize (7, 0);
cout << " D31:" << endl;
for_each (D31.begin(), D31.end(), Print);
cout << endl;
}
```

例 3-23 的执行效果如图 3-24 所示。



```
D1:
D2:
0 0 0 0 0
D3:
2 2 2 2 2 2 2 2
D31:
3 3 3 3 3 3 3 3
D4:
2 2 2 2 2 2 2 2
D5:
3 3 3 3 3 3 3 3
The size of D2: 5
The maxsize of D2: 1073741823
D31:
3 3 3 3
D31:
3 3 3 3 0 0
Press any key to continue
```

图 3-24 例 3-23 的执行效果



总结

通过例 3-23, 读者应掌握 5 种 deque 容器对象的定义形式。

3. deque 型容器的基本成员函数

(1) 赋值

deque 型容器提供了两个数据成员函数 `push_front()` 和 `push_back()`, 以实现把新元素插

入到 deque 对象中的功能。push_front() 函数用于在容器的头部插入新元素；push_back() 函数用于在容器的尾部插入新元素。同理，pop_front() 函数和 pop_back() 函数分别用于获取容器的头、尾两个元素。

注：在 vector 型容器中，在序列最前端插入新元素的效率非常低，所以 vector 型容器中不包括在序列头部插入新元素的成员函数。

deque 容器还提供了 operator []。其原型为：

```
const_reference operator [] (size_type pos) const;
reference operator [] (size_type pos);
```

通过赋值运算符“=”，也可以实现对容器中元素的赋值。

例 3-24

```
#include <iostream>
#include <deque>
#include <algorithm>
using namespace std;
void Print(int&ele)
{ cout << ele << " ";
}
void main()
{ deque<int> D1;
  D1.push_front(0);
  D1.push_front(1);
  D1.push_front(2);
  D1.push_front(3);
  D1.push_front(4);
  D1.push_back(1);
  D1.push_back(2);
  D1.push_back(3);
  D1.push_back(4);
  cout << " D1:" << endl;
  for_each(D1.begin(), D1.end(), Print); //注意观察元素在序列中的排列位置
  cout << endl;
  D1[4] = 9; //修改序列中的中间一个元素
  cout << " D1:" << endl;
  for_each(D1.begin(), D1.end(), Print); //注意观察元素在序列中的排列位置
  cout << endl;
}
```

例 3-24 的执行效果如图 3-25 所示。

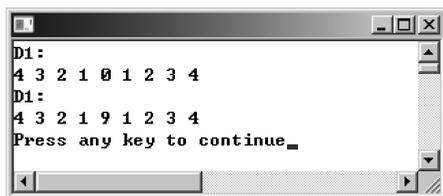


图 3-25 例 3-24 的执行效果

(2) 迭代器相关函数

deque 型容器提供了返回 deque 对象的迭代器或引用的成员函数，具体分类如下：

- 1) begin()、rbegin()、end() 和 rend()。
- 2) back() 和 front()。

1) 类中的 4 个函数返回值均为迭代器/逆迭代器型；2) 类中的两个函数的返回值为引用类型。

上述函数的原型为：

```
const_iterator begin() const;
iterator begin();
const_reverse_iterator rbegin() const;
reverse_iterator rbegin();
const_iterator end() const;
iterator end();
const_reverse_iterator rend() const;
reverse_iterator rend();
reference back();
const_reference back() const;
reference front();
const_reference front() const;
```

(3) 判断迭代器是否为空

deque 型容器提供了成员函数 empty()。如果容器为空，empty() 函数返回值为真。其原型为：

```
bool empty() const
```

(4) 元素的访问

deque 型容器提供了用于随机访问容器中元素的成员函数 at()，at() 函数的原型为：

```
const_reference at (size_type pos) const;
reference at (size_type pos);
```

(5) deque 序列的元素值重置技术

deque 型容器提供了 assign() 函数。该函数可以便捷地重置 deque 序列中某个元素的数值。其原型为：

```
void assign(const_iterator first, const_iterator last);
void assign (size_type n, const T& x = T());
```

下面利用例 3-25，对上述 (2)，(3)，(4)，(5) 的内容进行说明，以帮助读者加深印象。

例 3-25

```
#include <iostream>
#include <deque>
#include <algorithm>
using namespace std;
void Out(double&Ele)
```

```
{    cout.width(5);
    cout.precision(1);
    cout << std::fixed << Ele << " ";
}

void main()
{    deque <double> ::iterator Iter;
    deque <double> ::reverse_iterator rIter;
    deque <double> D1, D2, D3;
    for (int i=0; i<10; i++)
        { D1.push_front (90 + i/10.0);
        }

    cout << " All the elements of 双端序列 D1:" << endl;
    for_each (D1.begin(), D1.end(), Out);
    cout << endl;
    Iter = D1.begin();
    double begin = *Iter;
    Iter = D1.end();
    double end = * (--Iter);
    ////////////////
    rIter = D1.rbegin();
    double rbegin = *(rIter);
    rIter = D1.rend();
    double rend = * (--rIter);          //注意是 "--"
    double front = D1.front();
    double back = D1.back();
    cout << " begin : " << begin << ", " << " end : " << end << ", " << " front : " << front << ", " << "
back : " << back << endl;
    cout << " reverse begin : " << rbegin << " reverse end : " << rend << ", " << endl;
    if (D1.empty())
        {    cout << " 双端序列为空 ." << endl;
        }
    else
        {    int size = D1.size();
            cout << " 双短序列中包含 " << size << " 个元素 ." << endl;
        }

    double five = D1.at (5);
    cout << " 第5个元素是 " << five << " ." << endl;
    D2.assign (6, 0);
    D3.assign (D1.begin(), D1.end());
    cout << " All the elements of 双端序列 D2:" << endl;
    for_each (D2.begin(), D2.end(), Out);
    cout << endl;
    cout << " All the elements of 双端序列 D3:" << endl;
    for_each (D3.begin(), D3.end(), Out);
    cout << endl;
}
```

例 3-25 的执行效果如图 3-26 所示。

```

All the element of 双端序列 D1:
90.9 90.8 90.7 90.6 90.5 90.4 90.3 90.2 90.1 90.0
begin :90.9, end :90.0, front :90.9, back :90.0
reverse begin :90.0 reverse end :90.9,
双短序列中包含10个元素.
第5个元素是90.4.
All the element of 双端序列 D2:
0.0 0.0 0.0 0.0 0.0 0.0
All the element of 双端序列 D3:
90.9 90.8 90.7 90.6 90.5 90.4 90.3 90.2 90.1 90.0
Press any key to continue
  
```

图 3-26 例 3-25 的执行效果

4. deque 型容器的高级编程

(1) 容器元素交换

deque 型容器提供了成员函数 `swap()`，用以进行两个 deque 变量的交换操作。通过 `swap()` 函数可以完成两个 deque 变量的内容交换。其原型为：

```

Template < class T, class Allocator > void swap (deque < T, Allocator > & x, deque < T, Allocator >
&y);
  
```

`swap()` 函数即可以作为 deque 的成员函数来使用，还可以作为通用算法使用，两者功能是一致的。

(2) 插入和删除

deque 型容器提供了可实现丰富而又灵活的插入和删除元素操作功能的函数。除了 deque 型容器提供的 `push_front()`、`push_back()`、`pop_front()` 和 `pop_back()` 之外，`insert()` 函数既可以把某个元素插入到指定的位置，也可以把指定范围的多个元素插入到 deque 型容器中迭代器所指的位置，还可以把某个具体数值的多个副本重复插入到 deque 型容器中迭代器所指的位置。`erase()` 函数主要用来删除 deque 型容器中的元素。该函数既可以删除具体位置的某个元素，也可以删除指定范围内的多个元素。`clear()` 函数用于删除 deque 型容器中的所有元素。

1) insert() 函数。其原型为：

```

iterator insert(iterator it, const T& x = T());
void insert(iterator it, size_type n, const T& x);
void insert (iterator it, const_iterator first, const_iterator last);
  
```

第一种原型的作用在于把某个元素插入到指定位置，返回值为被插入元素的位置；第二种原型的作用是把某个值的多个备份插入到 deque 容器中迭代器所指位置。

2) `erase()` 和 `clear()`。在进行 deque 操作时，经常需要进行元素的删除操作，包括在 deque 的头部、尾部和中间任何位置进行该操作。由于 deque 是双端队列，因此在队列的头部和尾部删除操作最简单、最高效。deque 型容器提供了 `pop_front()` 和 `pop_back()` 两个函数，用以实现从队列的头部和尾部删除元素。成员函数 `erase()` 比这两个函数更为灵活实用，它可以删除迭代器指向的任意单个元素或相联的多个元素。

`clear()` 函数可以无条件地删除容器中的所有元素，可以被 `erase (begin, end)` 所代替。

3) 查找。deque 型容器没有提供“查找”和“搜索”相关的函数。但使用头文件

< algorithm > 中声明的算法 find() 函数, 同样可以实现对 deque 型容器的查找。详见例 3-26 的中文注释。

例 3-26

```
#include <iostream>
#include <deque>
#include <algorithm>
using namespace std;
void Out(double&Ele)
{   cout.width(5);
    cout.precision(1);
    cout<<std::fixed<<Ele<<" ";
}
void main()
{   deque<double> D1,D2;
    for(int i=0;i<10;i++) //初始化容器
    {   D1.push_front(90+i/10.0);
    }
    cout<<" All the element of 双端序列 D1:" <<endl;
    for_each(D1.begin(), D1.end(), Out); //输出容器 D1 中的内容
    cout<<endl;
    for(i=0; i<10; i++)
    {   D2.push_front(80+i/10.0);
    }
    cout<<" All the element of 双端序列 D2:" <<endl;
    for_each(D2.begin(), D2.end(), Out); //输出容器 D2 中的内容
    cout<<endl;
    D1.swap(D2); //交换容器 D1 和 D2 中的内容
    cout<<" D1 swap D2 : " <<endl;
    cout<<" All the element of 双端序列 D1:" <<endl;
    for_each(D1.begin(), D1.end(), Out); //输出 D1 中的所有元素
    cout<<endl;
    cout<<" All the element of 双端序列 D2:" <<endl;
    for_each(D2.begin(), D2.end(), Out); //输出 D2 中的所有元素
    cout<<endl;
    D1.insert(D1.begin(), -1.0); //插入 3 个元素
    D1.insert(D1.end(), -1.0);
    D1.insert(D1.begin()+6, -0.0);
    cout<<" All the element of 双端序列 D1:" <<endl;
    for_each(D1.begin(), D1.end(), Out);
    cout<<endl;
    D1.erase(D1.begin()+1); //删除两个元素
    D1.erase(D1.begin()+10);
    cout<<" All the element of 双端序列 D1:" <<endl;
    for_each(D1.begin(), D1.end(), Out);
    cout<<endl;
```

```

D2.clear();
cout << "D2 has been already cleared!" << endl;
cout << "All the element of 双端序列 D2:" << endl;
for_each (D2.begin(), D2.end(), Out);
cout << endl;
deque <double >:: iterator it = find (D1.begin(), D1.end(), 80.5); //使用 find()算法函数
int step = (it - D1.begin()); //求出所找元素的位置 (下标)
cout << " find 80.5. in D1 , its Index: " << step << ". " << endl;
}

```

例 3-26 的执行效果如图 3-27 所示。

```

All the element of 双端序列 D1:
90.9 90.8 90.7 90.6 90.5 90.4 90.3 90.2 90.1 90.0
All the element of 双端序列 D2:
80.9 80.8 80.7 80.6 80.5 80.4 80.3 80.2 80.1 80.0
D1 swap D2 :
All the element of 双端序列 D1:
80.9 80.8 80.7 80.6 80.5 80.4 80.3 80.2 80.1 80.0
All the element of 双端序列 D2:
90.9 90.8 90.7 90.6 90.5 90.4 90.3 90.2 90.1 90.0
All the element of 双端序列 D1:
-1.0 80.9 80.8 80.7 80.6 80.5 0.0 80.4 80.3 80.2 8
All the element of 双端序列 D1:
-1.0 80.8 80.7 80.6 80.5 0.0 80.4 80.3 80.2 80.1 -
D2 has been already cleared!
All the element of 双端序列 D2:

find 80.5. in D1 , its Index: 4 .
Press any key to continue_

```

图 3-27 例 3-26 的执行效果

5. deque 的函数模板

deque 模板类提供了大量函数模板，这使得 deque 型容器功能异常强大，同时也使编程变得异常便捷。deque 容器既包括运算符函数，也包括 swap() 函数（前面已介绍），此处主要介绍运算符函数。以函数模板形式实现的运算符主要包括 []、==、<、!=、>、>= 和 <=。

其中，“[]”用于访问容器中任意元素。

“==”用于判断两个容器是否相等。

“<”判断两个容器是否相同，如果前者小于后者，返回 true；否则，返回 false。

“!=”判断两个容器是否不相等，如果不相等，返回 true；否则，返回 false。

“<=”用于判断两个容器是否相等，如果前者不大于后者，返回 true；否则返回 false。

“>”用于判断两个容器是否相同，如果前者大于后者，返回 true；否则，返回 false。

“>=”用于判断两个容器是否相等，如果前者不小于后者，返回 true；否则，返回 false。

上述函数模板不受数据类型限制，顺序型容器全部提供了这些成员函数，且其使用方法也相同。

3.3 关联式容器

关联式容器是最常见的、也是最有用的容器。关联式容器其实是关联数组概念的扩展。

关联式容器依据特定的排序准则，自动为其元素排序。关联式容器中的元素都经过排序，是有序的。所有关联式容器都有一个可供选择的 `template` 参数。这个参数用以指明排序准则。排序准则是以函数形式体现的，用于比较元素值或键值。默认情况下以 “`operator <`” 进行比较排序。程序员也可以提供自定义的比较函数，从而定义出不同的排序准则。

通常关联式容器由二叉树数据结构实现的。在二叉树中，每个元素都有一个父节点和两个子节点；左子树的所有元素都比该元素的值小，右子树的所有元素都比该元素的值大。关联式容器的差别在于元素的类型以及处理重复元素的方式。关联式容器还能提供对元素的快速访问，但不能实现任意位置的操作。

主要的关联式容器包括 `set`（集合）、`multisets`、`maps`（映射）和 `multimaps`。其中 `set` 可视为一种特殊的 `map`，其元素的值即键值。前两种容器是在 `<set>` 头文件中声明和定义，后两种容器是在 `<map>` 头文件中声明和定义的。

`set` 是支持随机存取的容器，其键值和实值是同一个值。`set` 型容器中的所有元素必须具有唯一值，即不能包含重复的元素。`set` 型容器中的元素可以实现按照次序来存储一组数值，即在一个集合中元素既作为被存储的数据又作为数据的键值。`multiset` 是另一种类型的容器，其键值和数据元素是同样的值。与 `set` 不同的是，它可以包含重复的元素。`multiset` 对象也可以实现按照次序来存储一组数值。

`map` 是一种包含成对数据的关联数组容器。成对数据中的一个值是实值，另一个值是用来寻找数据的键值。一个特定的键值只能与一个元素相联系。`map` 是排序结构体，键值是独一无二的。事实上，`map` 的内部结构和 `set` 是一样的。`set` 可以看作一种特殊的 `map`，其键值和实值是同一个。`multimap` 是一种允许出现重复键值的关联数组容器。与 `map` 对象不同，一个键值可以和多个元素相联系，`multimap` 而且允许键值重复。

下面将主要介绍上述 4 种关联式容器。

3.3.1 set/multiset（集合）类模板

`set` 型容器能顺序存储一组数值，这些数值既充当存储的数据，又充当数据的键值。该集合更像一个有序链表，其中的元素以升序顺序存储。

1. set 的定义

`set` 类模板的声明如下：

```
template <class Key, class Traits = less <Key >, class Allocator = allocator <Key >> class set
```

其中，参数 `Key` 是存储在集合中元素的数据类型；参数 `Traits` 是用于实现集合内部排序的仿函数，默认值为 `less <Key >`；参数 `Allocator` 代表集合的内存配置器，负责内存的分配和销毁。使用 `set` 类模板时，需要包含头文件 `<set>`。`set` 类模板的构造函数包含以下几种：

```
explicit set ( const Traits& _Comp );  
explicit set ( const Traits& _Comp, const Allocator& _Al );  
set ( const _set& _Right );
```

```

template < class InputIterator > set ( InputIterator _First, InputIterator _Last );
template < class InputIterator > set ( InputIterator _First,
                                     InputIterator _Last,
                                     const Traits& _Comp );
template < class InputIterator > set ( InputIterator _First,
                                     InputIterator _Last,
                                     const Traits& _Comp,
                                     const Allocator& _Al );

```

上述是 set 类模板中构造函数的 6 种形式。第一种形式最简单，定义一个空对象（容器），参数包含了一个谓词，用于容器内部排序用；第二种形式除包含一个谓词之外，还加入了内存配置器；第三种形式用于使用已存在的 set 定义新的 set；第四种形式包含了使用迭代器指定固定范围，用于定义新的 set；第五种形式在第四种形式的基础增加了参数 _Comp，从而可以自定义排序规则；第六种形式在第五种形式的基础上增加了内存配置器的内容。

在本节之前，本书已经大量使用了“排序”。在此对排序准则进行简要描述。所谓排序准则，应具备如下特征：

1) 必须是“反对称的”。对操作符“operator <”而言，若 $x < y$ 为真，则 $y < x$ 为假。对判断式 predicate op() 而言，若 op (x, y) 为真 (1)，则 op (y, x) 为假 (0)。

2) 必须是“可传递的”。对“operator <”而言，若 $x < y$ 为真且 $y < z$ 为真，则 $x < z$ 为真。对于判断式 op() 而言，如果 op (x, y) 为真且 op (y, x) 为真，则 op (x, z) 为真。

3) 必须是“非自反的”。对“operator <”而言， $x < x$ 永远为假是不可能的。对判断式 predicate op() 而言，op (x, x) 永远为假。



总结 以上内容是 STL 学术理论的一部分。STL 先建立起一个抽象概念阶层体系，继而形成一个软件组件分类学，最后再以实际工具（template）将各个概念实现。《Generic Programming and the STL》一书对这些理论做了诸多描述，其中文译本为《泛型编程与 STL》。

要具体实现排序准则，可采用两种形式：①在类模板中以参数形式实现；②以构造函数参数定义之。具体形式如下：

```

std::set < int, std::greater < int >> s1;           //第一种形式实现排序规则
set < int > s2 (less < int > ());                 //第二种形式实现排序规则

```

set 虽然具有排序功能（multiset 同样），但也有其局限性。一旦元素被输入至容器中，就无法再对其进行“确定的”访问，因为元素的位置改变了。原有的输入顺序被自动排序功能打乱了。要改变元素的值，需要先删除原有的元素，再重新插入新元素。所以，set (和 multiset) 没有提供任何直接存取元素的成员函数。虽然 STL 提供了多种 set 定义方法，但 Visual C++ 6.0 并没有提供所有定义方法，而 Linux 环境下的 C++ 编译系统提供了所有定义方法。通过例 3-27 对以上内容加以详细说明。

例 3-27

```

#include < iostream >
#include < set >

```

```
using namespace std;
void OutPut (set < int > &s)
{
    set < int > ::iterator it;
    for(it = s.begin(); it != s.end(); it++)
        cout << " " << *it << ", ";
    cout << endl;
}
void OutPutM (multiset < int > &s)
{
    multiset < int > ::iterator it;
    for(it = s.begin(); it != s.end(); it++)
        cout << " " << *it << ", ";
    cout << endl;
}
void main ()
{
    set < int > s1;
    s1.insert (10);
    s1.insert (15);
    s1.insert (25);
    s1.insert (20);
    s1.insert (30);
    s1.insert (33);
    s1.insert (5);
    s1.insert (20);
    OutPut (s1);
    set < int > s2 (less < int > ());
    s2.insert (10);
    s2.insert (13);
    s2.insert (11);
    s2.insert (19);
    s2.insert (17);
    s2.insert (21);
    OutPut (s2);
    set < int > ::allocator_type s1_Alloc;
    s1_Alloc = s1.get_allocator ();
    set < int > s3 (less < int > (), s1_Alloc);
    s3.insert (1);
    s3.insert (5);
    s3.insert (2);
    OutPut (s3);
    set < int > s4 (s1);
    OutPut (s4);
    multiset < int > sm1;
    sm1.insert (10);
    sm1.insert (15);
    sm1.insert (25);
    sm1.insert (20);
```

```

sm1.insert(30);
sm1.insert(33);
sm1.insert(5);
sm1.insert(20);
OutPutM(sm1);
}

```

例 3-27 的执行效果如图 3-28 所示。

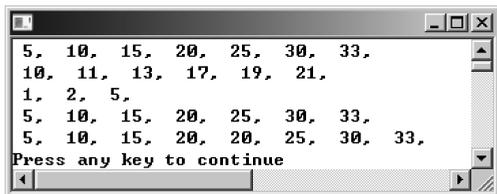


图 3-28 例 3-27 的执行效果

例 3-27 中的几种定义形式，在 Visual C++ 6.0 中调试通过。但在模板中使用多个参数的定义形式没有调试通过。另外，除了允许元素重复之外，multiset 的定义形式和 set 是相同的。



提示 在 main() 函数的第 9 行，输入了一个重复元素“20”，但程序输出时，并没有这个元素。说明了 set 中不允许有重复的元素。

2. set 和 multiset 的容量，搜寻及统计

和其他容器一样，set 型容器提供了获取容器容量的 size() 函数，判断容器是否为空的 empty() 函数以及返回容器可容纳最大元素数的成员函数 max_size()。

set (multiset) 型容器还提供了元素个数统计 count() 函数。

set (multiset) 型容器提供了查找 find() 函数，还提供了 low_bound()、upper_bound() 和 equal_range() 函数。

以上几个函数的原型为：

```

size_type size() const;
bool empty() const;
size_type max_size() const;
size_type count ( const Key& _Key) const;
iterator find ( const Key& _Key) const;
const_iterator find ( const Key& _Key) const;
const_iterator lower_bound (const Key& _Key) const;
iterator lower_bound (const Key& _Key) const;
const_iterator upper_bound (const Key& _Key) const;
iterator upper_bound (const Key& _Key) const;
pair <const_iterator, const_iterator> equal_range (const Key& _Key) const;
pair <iterator, iterator> equal_range (const Key& _Key) const;

```

find (Key) 函数的功能是返回键值为 Key 的元素的位置，其返回值是迭代器类型。count (Key) 函数的功能是统计键值为 Key 的元素个数。

`low_bound (const Key& _Key)` 函数的返回值是迭代器，该迭代器指向集合中键值大于并等于参数 `Key` 的第一个元素。

`upper_bound (const Key& _Key)` 函数的返回值是迭代器，该迭代器指向集合中键值大于参数 `Key` 的第一个元素。

`equal_range (const Key& _Key)` 函数的返回值是迭代器对 (`pair`)，该迭代器对 (`pair`) 的两个迭代器分别指向集合中键值大于并等于参数 `Key` 的第一个元素和集合中键值大于参数 `Key` 的第一个元素。

下面使用例 3-28 来说明以上几个函数的使用方法。

例 3-28

```
#pragma warning(disable:4786)
#include <iostream>
#include <set>
#include <algorithm>
using namespace std;
void Print(int&Ele)
{ cout << Ele << ", ";
}
void main()
{ typedef set<int> SET;
  typedef multiset<int> MSET;
  SET s1;
  MSET s2;
  SET::iterator it;
  MSET::iterator Mit;
  pair<SET::iterator, SET::iterator> pl;
  pair<MSET::iterator, MSET::iterator> Mpl;
  s1.insert(10);
  s1.insert(15);
  s1.insert(21);
  s1.insert(17);
  s2.insert(11);
  s2.insert(16);
  s2.insert(20);
  s2.insert(18);
  s2.insert(20); //以上是初始化
  cout << "The Set S1 is below:" << endl;
  for_each(s1.begin(), s1.end(), Print);
  cout << endl;
  cout << " The Multiset S2 is below:" << endl;
  for_each(s2.begin(), s2.end(), Print);
  cout << endl;
  int size = s1.size(); //计算容量
  int Msize = s2.size();
  cout << " The size of the set s1: " << size << endl;
```

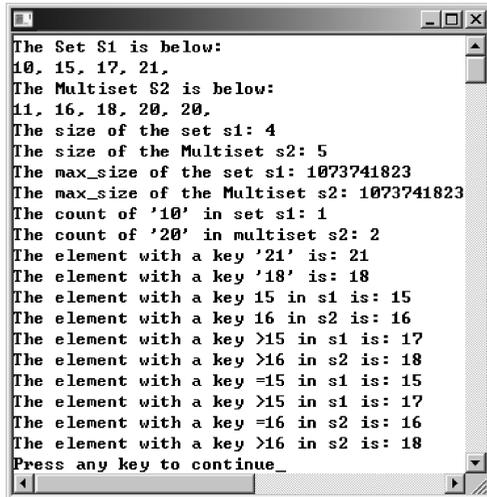
```
cout << "The size of the Multiset s2: " << Msize << endl;
int max_size = s1.max_size(); //最大容量
int max_Msize = s2.max_size();
cout << " The max_size of the set s1: " << max_size << endl;
cout << " The max_size of the Multiset s2: " << max_Msize << endl;
int cnt = s1.count (10); //指定元素的个数
int Mcnt = s2.count (20);
cout << " The count of \'10 \' in set s1: " << cnt << endl;
cout << " The count of \'20 \' in multiset s2: " << Mcnt << endl;
it = s1.find (21); //查找元素
Mit = s2.find (18);
cout << " The element with a key '21' is: " << *it << endl;
cout << " The element with a key '18' is: " << *Mit << endl;
it = s1.lower_bound (15); //查找元素
Mit = s2.lower_bound (16);
if (it == s1.end())
{ cout << " The element with a key 15 in s1 doesn't exist. " << endl;
}
else
{ cout << " The element with a key 15 in s1 is: " << *it << endl;
}
if (Mit == s2.end())
{ cout << " The element with a key 16 in s2 doesn't exist. " << endl;
}
else
{cout << " The element with a key 16 in s2 is: " << *Mit << endl;
}
it = s1.upper_bound (15); //查找元素
Mit = s2.upper_bound (16);
if (it == s1.end())
{ cout << " The element with a key >15 in s1 doesn't exist. " << endl;
}
else
{cout << " The element with a key >15 in s1 is: " << *it << endl;
}
if (Mit == s2.end())
{ cout << " The element with a key >16 in s2 doesn't exist. " << endl;
}
else
{ cout << " The element with a key >16 in s2 is: " << *Mit << endl;
}
p1 = s1.equal_range (15); //查找元素
Mp1 = s2.equal_range (16);
if (it == s1.end())
{ cout << " The element with a key >=15 in s1 doesn't exist. " << endl;
}
}
```

```

else
{
    cout << "The element with a key =15 in s1 is: " << *p1.first << endl;
    cout << "The element with a key >15 in s1 is: " << *p1.second << endl;
}
if(Mit == s2.end())
{
    cout << "The element with a key >16 in s2 doesn't exist. " << endl;
}
else
{
    cout << "The element with a key =16 in s2 is: " << *Mpl.first << endl;
    cout << "The element with a key >16 in s2 is: " << *Mpl.second << endl;
}
}
}

```

例 3-28 的执行效果如图 3-29 所示。



```

The Set S1 is below:
10, 15, 17, 21.
The Multiset S2 is below:
11, 16, 18, 20, 20.
The size of the set s1: 4
The size of the Multiset s2: 5
The max_size of the set s1: 1073741823
The max_size of the Multiset s2: 1073741823
The count of '10' in set s1: 1
The count of '20' in multiset s2: 2
The element with a key '21' is: 21
The element with a key '18' is: 18
The element with a key 15 in s1 is: 15
The element with a key 16 in s2 is: 16
The element with a key >15 in s1 is: 17
The element with a key >16 in s2 is: 18
The element with a key =15 in s1 is: 15
The element with a key >15 in s1 is: 17
The element with a key =16 in s2 is: 16
The element with a key >16 in s2 is: 18
Press any key to continue_

```

图 3-29 例 3-28 的执行效果

3. set 和 multiset 的迭代器相关函数及赋值函数

set 和 multiset 提供了所有容器均拥有的基本赋值操作：“=”，swap 函数。赋值操作的两端容器必须具有相同型别。

set 和 multiset 类模板中，和迭代器相关的成员函数主要包括 begin()、end()、rbegin() 和 rend()。其中，begin() 和 end() 函数是双向迭代器，rbegin() 和 rend() 函数是逆向迭代器。对于迭代器，所有元素都被视为常数，确保不会人为改变元素值或打乱既定顺序。set 和 multiset 中的元素是无法调用任何修改性算法的。set 和 multiset 是不能使用 remove() 算法函数的，如果要移除 set 和 multiset 中的元素，必须使用它们本身提供的成员函数。

swap() 函数的原型为：

```
void swap(set& str);
```

begin() 和 end() 函数的原型为：

```
const_iterator begin() const
const_iterator end() const;
```

`rbegin()` 和 `rend()` 函数的原型为:

```
const_reverse_iterator rbegin() const;
const_reverse_iterator rend() const;
```

例 3-29

```
#pragma warning(disable:4786)
#include <iostream>
#include <set>
#include <algorithm>
using namespace std;
void print(double&Ele)
{ cout << Ele << " ";
}
void main()
{ set <double> s1,s2;
  s1.insert(11);
  s1.insert(21);
  s1.insert(17);
  s1.insert(19);
  s1.insert(9);
  s1.insert(13); //初始化
  cout << "s1: " << endl;
  for_each(s1.begin(), s1.end(), print); //输出 s1
  cout << endl;
  s2 = s1;
  cout << " s2: " << endl;
  for_each(s2.begin(), s2.end(), print); //输出 s2
  cout << endl;
  s2.insert(31);
  s2.insert(24);
  cout << " s2: " << endl;
  for_each(s2.begin(), s2.end(), print); //输出 s2
  cout << endl;
  s1.swap(s2);
  cout << " s1: " << endl;
  for_each(s1.begin(), s1.end(), print); //交换
  cout << endl;
  cout << " s2: " << endl;
  for_each(s2.begin(), s2.end(), print); //输出
  cout << endl;
  set <double>::iterator its;
  set <double>::reverse_iterator rits;
  its = s1.begin(); //第一个元素
  cout << " The first Element of sequence \'s1 \': " << *its << endl;
  its = s1.end(); //最后一个元素
```

```

cout << "The last Element of sequence \'s1\':" <<* (-- its) << endl;
rits = s1.rbegin(); //逆向第一个元素
cout << "The first Element of sequence in reverse direction. \'s1\':" <<* rits << endl;
rits = s1.rend(); //逆向最后一个元素
cout << "The last Element of sequence in reverse direction. \'s1\':" <<* (-- rits) << endl;
}

```

例 3-29 的执行效果如图 3-30 所示。

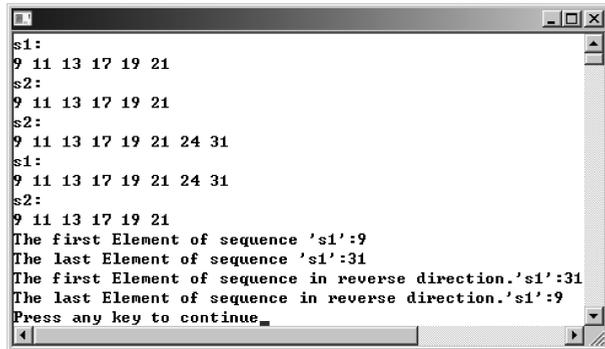


图 3-30 例 3-29 的执行效果

4. set 和 multiset 的插入和移除

set 和 multiset 中用于实现插入元素和删除元素的函数主要包括 insert()、erase() 和 clear()。迭代器需要指向有效位置，但序列起点不能位于终点之后，并且不能从空容器中删除元素。因此当删除元素时，首先要判断容器的容量。

insert() 函数的原型为：

```

pair< iterator, bool > insert(const value_type& x);
iterator insert (iterator it, const value_type& x);
void insert (const value_type *first, const value_type *last);

```

上述第一种模型，insert() 函数的返回值是 pair< iterator, bool > 类型。pair 型返回值的第 1 个参数是迭代器，代表插入元素的位置；pair 型返回值的第 2 个参数是 bool 类型，代表是否插入成功？

insert() 函数可以向容器中加入一个对象、一个对象的若干副本或者某个范围内的多个对象。其插入位置是使用迭代器指定的。

erase() 函数可以删除一个由迭代器指定的元素，也可以删除某个范围内的多个元素。

erase() 函数原型为：

```

iterator erase(iterator it);
iterator erase(iterator first, iterator last);
size_type erase (const Key& key);

```

第一种形式的功能是将迭代器所指向的元素删除，第二种形式的功能是将迭代器所限定范围内的元素全部删除，第三种形式的功能是将元素 Key 删除。

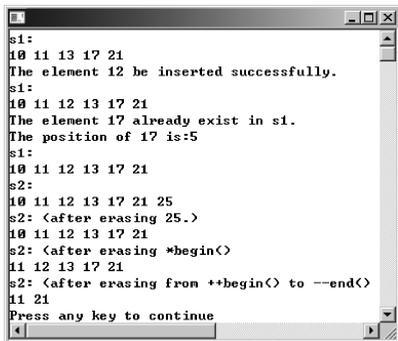
下面请参考例 3-30。

例 3-30

```
#pragma warning(disable:4786)
#include <iostream>
#include <set>
#include <algorithm>
using namespace std;
void print(int&Ele)
{ cout<<Ele<<" ";
}
template <typename T>void printSet(set<T>&s)
{ for_each(s.begin(), s.end(), print);
  cout<<endl;
}
void main()
{ set<int> s1, s2;
  pair<set<int>::iterator, bool> p1;
  s1.insert(10);
  s1.insert(11);
  s1.insert(13);
  s1.insert(21);
  s1.insert(17);
  cout<<" s1: " <<endl;
  printSet(s1);
  p1=s1.insert(12);
  if(p1.second==true)
  { cout<<" The element 12 be inserted successfully. " <<endl;
  }
  else
  { cout<<" The element 12 already exist in s1. " <<endl;
  }
  cout<<" s1: " <<endl;
  printSet(s1);
  p1=s1.insert(17);
  if(p1.second==true)
  { cout<<" The element 17 be inserted successfully. " <<endl;
    cout<<" The position of 17 is:" <<distance(s1.begin(), p1.first) +1 <<endl;
  }
  else
  { cout<<" The element 17 already exist in s1. " <<endl;
    cout<<" The position of 17 is:" <<distance(s1.begin(), p1.first) +1 <<endl;
  }
  cout<<" s1: " <<endl;
  printSet(s1);
  s2=s1;
  s2.insert(25);
```

```
cout << "s2: " << endl;
printSet(s2);
s2.erase(25);
cout << "s2: (after erasing 25.) " << endl;
printSet(s2);
s2.erase(s2.begin());
cout << "s2: (after erasing * begin() " << endl;
printSet(s2);
s2.erase(++s2.begin(), --s2.end());
cout << "s2: (after erasing from ++begin() to --end()" << endl;
printSet(s2);
}
```

例 3-30 的执行效果如图 3-31 所示。



```
s1:
10 11 13 17 21
The element 12 be inserted successfully.
s1:
10 11 12 13 17 21
The element 17 already exist in s1.
The position of 17 is:5
s1:
10 11 12 13 17 21
s2:
10 11 12 13 17 21 25
s2: <after erasing 25.>
10 11 12 13 17 21
s2: <after erasing *begin<>
11 12 13 17 21
s2: <after erasing from ++begin<> to --end<>
11 21
Press any key to continue
```

图 3-31 例 3-30 的执行效果

提示 请读者注意，在例 3-30 中，第一次使用了 `distance()` 算法函数，用于计算容器中两个元素的距离。本例较详细地使用了 `pair` 类型的两个成员 `first` 和 `second`，这两个成员分别代表 `pair` 对象中的两个元素。本例仅使用了 `set` 型容器。

5. `set` 和 `multiset` 的比较运算符

容器之间的比较只能用于相同型别。因此，若要排序，就必须是相同型别的元素，否则编译时会产生错误。ISO/IEC 14882 提供了关于运算符的声明和定义。运算符的使用和序列式容器是一致的。

Visual C++ 6.0 没有提供比较运算符的成员函数，但提供了另外两个函数：`key_comp()` 和 `value_comp()`。其中 `key_comp()` 函数用于键值的比较，`value_comp()` 函数用于实值比较。这两个函数的原型为：

```
key_compare key_comp() const;
value_compare value_comp() const;
```

`key_comp()` 函数的返回值是 `key_compare` 类型，`key_compare` 决定了集合中元素的排列顺序。通过对返回值的测试，可以确定序列排序形式。`value_comp()` 函数的返回值是 `value_compare` 类型。其意义和 `key_compare` 是一致的。

(1) 键值比较

下面举例说明 `key_comp()` 函数的使用方法。请读者认真阅读下面的源代码。

例 3-31

```
#pragma warning(disable:4786)
#include <iostream>
#include <set>
#include <algorithm>
using namespace std;
void Print(double&Ele)
{ cout<<Ele<<" ";
}
template <typename T>void PrintS(set<T>& s)
{ for_each(s.begin(), s.end(), Print);
  cout<<endl;
}
template <typename T>void PrintM(multiset<T, greater<T>>& s)
{ for_each(s.begin(), s.end(), Print);
  cout<<endl;
}
using namespace std;
void main()
{ set<double, less<double>> s1;
  s1.insert(1);
  s1.insert(5);
  s1.insert(7);
  s1.insert(4);
  multiset<double, greater<double>> s2;
  s2.insert(7);
  s2.insert(9);
  s2.insert(6);
  s2.insert(4);
  set<double, less<double>>::key_compare kc1 = s1.key_comp();
  multiset<double, greater<double>>::key_compare kc2 = s2.key_comp();
  cout<<" s1:" <<endl;
  PrintS(s1);
  cout<<" s2:" <<endl;
  PrintM(s2);
  if(kc1(2, 3) == true)
  { cout<<" The function object of s1 return true. " <<endl;
  }
  else
  { cout<<" The function object of s1 return false. " <<endl;
  }
  if(kc2(2, 3) == true)
  { cout<<" The function object of s2 return true. " <<endl;
```

```

}
else
{ cout << "The function object of s2 return false." << endl;
}
}
}

```

例 3-31 的执行效果如图 3-32 所示。

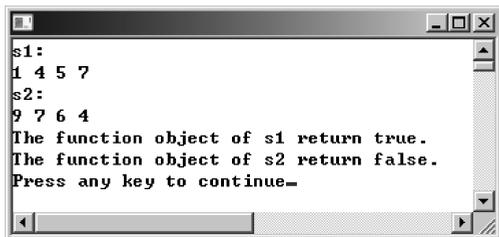


图 3-32 例 3-31 的执行效果



总结

在例 3-31 中，关于 set 和 multiset 两个类型，使用了较复杂的定义对象形式（见源代码第 19 行和第 24 行）。在使用时，注意“less < double >”和其后面的“>”之间一定要有一个空格，否则编译时会出错。

(2) 实值比较

value_comp() 函数的使用方法和 key_comp() 的使用方法相同。只不过 value_comp() 函数的返回类型为 value_compare。此处就不再重新举例，在例 3-31 稍作改动即可。

3.3.2 map/multimap (图) 类模板

map 型容器是（键-值）对的集合。map 型容器通常可理解为关联数组（associative array），可使用键作为下标来获取对应的值，类似于内置数组类型。关联的本质在于元素的值与某个特定的键相联系，而不是通过在数组中的位置来实现关联的。

总而言之，map 是由许多对的键值组成的排序结构体，且键值是独一无二的。

multimap 型容器和 map 型容器基本是一致的。只是前者允许重复元素，而后者不允许。

map 型容器和 multimap 型容器的示意图如图 3-33 所示。

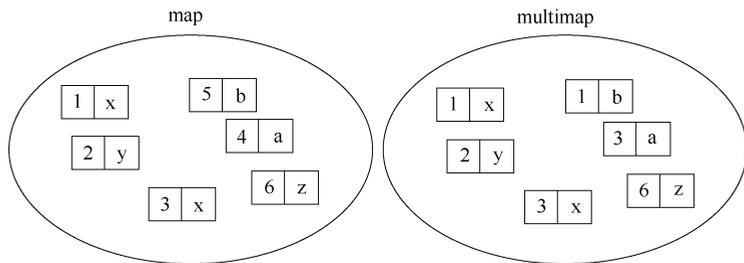


图 3-33 map 型容器和 multimap 型容器的示意图

在 STL 中，这两种型别的模板为：

```
template <class Key,class T,class Compare = less <Key >, class Allocator = allocator <pair <const Key,T >>> class map;
```

和

```
template <class Key, class T, class Compare = less <Key >, class Allocator = allocator <pair <const Key,T >>> class multimap;
```

第一个 `template` 参数被当作容器中元素的 `key` (键值), 第二个 `template` 参数被当作元素的 `value` (实值)。前两个参数键值和实值必须具备可赋值和可复制的性质。并且由于容器内部要排序, 因此键值必须是可比较的。第三个参数可有可无, 用于定义排序准则。元素的顺序仅由键值决定, 和实值是无关系的。默认排序准则是 “`less <>`”。

由以上内容可知, `map` 对象可以使程序按照次序存储一组数值。每个元素均由一个键实现排序和检索。在上一节介绍的 `set/multiset` 型容器中, 元素既作为键值, 又作为实值。而在 `map/multimap` 型容器中, 键值和实值是以 “数据对” (`pair`) 的形式出现的。

1. `map` 和 `multimap` 基础

使用 `map` 和 `multimap` 型容器时, 必须包含头文件 `<map >`。在内存中, `map` 和 `multimap` 的对象均是以 “数据对” 的形式存放在二叉树中的。

(1) `map` 和 `multimap` 的定义和初始化

`map` 型容器和 `multimap` 型容器的构造函数见表 3-1。

表 3-1 `map` 型容器和 `multimap` 型容器的构造函数

构造函数	备注
<code>map c7 / multimap c</code>	产生一个空的 <code>map</code> 或 <code>multimap</code> , 不包含任何因素
<code>map c (op) / multimap c (op)</code>	以 <code>op</code> 为排序准则, 产生一个空的 <code>map/multimap</code>
<code>map c1 (c2) / multimap c1 (c2)</code>	产生某个 <code>map/multimap</code> 对象的副本, 所有元素均被复制
<code>map c (beg, end) / multimap c (beg, end)</code>	以区间 (<code>beg, end</code>) 内的元素产生一个新的 <code>map/multimap</code>
<code>map c (beg, end, op) / multimap c (beg, end, op)</code>	以区间 (<code>beg, end</code>) 内的元素产生一个新的 <code>map/multimap</code> , 排序准则为 <code>op</code>

`map` 型容器和 `multimap` 型容器的形式见表 3-2。

表 3-2 `map` 型容器和 `multimap` 型容器的形式

<code>map</code>	效果
<code>map <key, Elem ></code>	以 <code>less <></code> 为排序准则
<code>map <key, Elem, op ></code>	以 <code>op</code> 为排序准则
<code>multimap <key, Elem ></code>	以 <code>less <></code> 为排序准则
<code>multimap <key, Elem, op ></code>	以 <code>op</code> 为排序准则

通过学习上述内容, 读者基本可以实现对 `map` 的定义。例如,

```
map <key,Elem> c;
map <key,Elem> c(op);
map <key,Elem> c(beg, end);
...
```

通常定义排序准则有两种方法：以模板参数定义；以构造函数参数定义。这些在表 3-1 和表 3-2 中已经有所体现，下面进行简单的阐述。

1) 使用模板参数定义排序准则。

```
map<float, string, greater<float>> m1;
```

此时，排序准则是型别的一部分。而实际的排序准则是容器产生的函数对象（即仿函数）。

2) 以构造函数参数定义排序准则。以构造函数参数定义排序准则时，只有当构造函数被执行时，排序准则才有效，即执行时才能获得排序准则，并且程序还可应用到多种的排序准则。

下面以例 3-32 来阐述以上内容。

例 3-32

```
#pragma warning(disable:4786)
#include <iostream>
#include <map>
using namespace std;
typedef pair<int, double> CustomPair;
void Print(map<int, double>& m)
{
    map<int, double>::iterator it;
    for(it = m.begin(); it != m.end(); it++)
    {
        CustomPair p1 = (pair<int, double>)(*it);
        cout << p1.first << ", ";
        cout << std::fixed << cout.precision(2) << p1.second << "; " << endl;
    }
}
void PrintM(multimap<int, double>& m)
{
    multimap<int, double>::iterator it;
    for(it = m.begin(); it != m.end(); it++)
    {
        CustomPair p1 = (pair<int, double>)(*it);
        cout << p1.first << ", ";
        cout << std::fixed << cout.precision(2) << p1.second << "; " << endl;
    }
}
void PrintG(map<int, double, greater<int>>& m)
{
    map<int, double, greater<int>>::iterator it;
    for(it = m.begin(); it != m.end(); it++)
    {
        CustomPair p1 = (pair<int, double>)(*it);
        cout << p1.first << ", ";
        cout << std::fixed << cout.precision(2) << p1.second << "; " << endl;
    }
}
void PrintG_M(multimap<int, double, greater<int>>& m)
{
    multimap<int, double, greater<int>>::iterator it;
    for(it = m.begin(); it != m.end(); it++)
    {
        CustomPair p1 = (pair<int, double>)(*it);
```

```
        cout << p1.first << ", ";
        cout << std::fixed << cout.precision(2) << p1.second << "; " << endl;
    }
}

void PrintL_M (map<int, double, less<int>>& m)
{
    map<int, double, less<int>>::iterator it;
    for (it=m.begin(); it!=m.end(); it++)
    {
        CustomPair p1 = (pair<int, double>) (* it);
        cout << p1.first << ", ";
        cout << std::fixed << cout.precision(2) << p1.second << "; " << endl;
    }
}

void main()
{
    map<int, double>::iterator itm;
    map<int, double, greater<int>>::iterator itmG;
    map<int, double, less<int>>::iterator itmL;
    map<int, double> m1;
    map<int, double, greater<int>>m2;
    multimap<int, double> m3;
    multimap<int, double, greater<int>>m4;
    m1.insert (CustomPair (1, 2.0));
    m1.insert (CustomPair (2, 5.0));
    m1.insert (CustomPair (3, 7.0));
    m1.insert (CustomPair (4, 8.0));
    m1.insert (CustomPair (5, 11.0));
    m1.insert (CustomPair (6, 6.0));
    cout << " m1: " << endl;
    Print (m1);
    m2.insert (CustomPair (1, 2.0));
    m2.insert (CustomPair (2, 5.0));
    m2.insert (CustomPair (3, 7.0));
    m2.insert (CustomPair (4, 8.0));
    m2.insert (CustomPair (5, 11.0));
    m2.insert (CustomPair (6, 6.0));
    cout << " m2 (greater<int>:)" << endl;
    PrintG (m2);
    m3.insert (CustomPair (1, 2.0));
    m3.insert (CustomPair (2, 5.0));
    m3.insert (CustomPair (3, 7.0));
    m3.insert (CustomPair (4, 8.0));
    m3.insert (CustomPair (5, 11.0));
    m3.insert (CustomPair (6, 6.0));
    cout << " m3:" << endl;
    PrintM (m3);
    m4.insert (CustomPair (1, 2.0));
```

```
m4.insert(CustomPair(2,5.0));
m4.insert(CustomPair(3,7.0));
m4.insert(CustomPair(4,8.0));
m4.insert(CustomPair(5,11.0));
m4.insert(CustomPair(6,6.0));
cout<<"m4 (greater<int>:)"<<endl;
PrintG_M(m4);
map<int, double>::allocator_type ma;
ma=m2.get_allocator();
map<int, double> m5 (less<int>(), ma);
m5.insert (CustomPair (16, 1.0));
m5.insert (CustomPair (15, 7.0));
m5.insert (CustomPair (24, 9.0));
m5.insert (CustomPair (23, 13.0));
m5.insert (CustomPair (32, 21.0));
m5.insert (CustomPair (11, 27.0));
cout<<" m5 (less<int>:)"<<endl;
PrintL_M(m5);
}
```

例 3-32 的执行效果如图 3-34 所示。

```
m1:
1. 62.00;
2. 25.00;
3. 27.00;
4. 28.00;
5. 211.00;
6. 26.00;
m2(greater<int>:)
6. 26.00;
5. 211.00;
4. 28.00;
3. 27.00;
2. 25.00;
1. 22.00;
m3:
1. 22.00;
2. 25.00;
3. 27.00;
4. 28.00;
5. 211.00;
6. 26.00;
m4(greater<int>:)
6. 26.00;
5. 211.00;
4. 28.00;
3. 27.00;
2. 25.00;
1. 22.00;
m5(less<int>:)
11. 227.00;
15. 27.00;
16. 21.00;
23. 213.00;
24. 29.00;
32. 221.00;
Press any key to continue
```

图 3-34 例 3-32 的执行效果



总结 例 3-22 对 map/multimap 的多种定义形式均进行了尝试。在输出函数中, 还用了格式化输出的方法。

(2) map 的容量

map 型容器和 multimap 型容器定义了两个成员函数 size() 和 max_size(), 用来确定 map 型容器和 multimap 型容器中数据成员的数量。其原型为:

```
size_type size() const;  
size_type max_size() const;
```

上述两个函数的使用方法见例 3-33。

例 3-33

```
#pragma warning(disable:4786)  
#include <iostream>  
#include <map>  
using namespace std;  
void main()  
{ map<int,double,greater<int>>m1;  
  multimap<int,double,greater<int>>m2;  
  typedef pair<int,double>mypair;  
  m1.insert(mypair(1,5.0));  
  m1.insert(mypair(2,7.0));  
  m1.insert(mypair(3,7.0));  
  m1.insert(mypair(4,17.0));  
  m1.insert(mypair(5,11.0));  
  m2.insert(mypair(1,4.0));  
  m2.insert(mypair(2,8.0));  
  m2.insert(mypair(3,9.0));  
  m2.insert(mypair(4,12.0));  
  m2.insert(mypair(5,19.0));  
  int size=m1.size(); //计算 size  
  int max_size=m1.max_size(); //计算 max_size  
  cout<<" the size of map m1 : " <<size<<" , the max_size of map m1 : " <<max_size<<endl;  
  int m_size=m2.size();  
  int m_max_size=m2.max_size();  
  cout<<" the size of multimap m2 : " <<m_size<<" , the max_size of multimap m2 : " <<m_max_size<<endl;  
}
```

例 3-33 的执行效果如图 3-35 所示。

```
the size of map m1 : 5, the max_size of map m1 : 536870911  
the size of multimap m2 : 5, the max_size of multimap m2 : 536870911  
Press any key to continue
```

图 3-35 例 3-33 的执行效果

2. map 和 multimap 型容器的基本成员函数

(1) 判断容器是否为空

判断一个 map/multimap 型容器是否为空是很重要的。如果 map/multimap 型容器为空，成员函数 empty() 返回 true。其原型为：

```
bool empty() const;
```

(2) 遍历容器

map 和 multimap 型容器不支持元素直接存取。元素的存取需要经过迭代器实现，并且 map 和 multimap 的迭代器均是双向迭代器。此类成员函数包括 begin()、end()、rbegin() 和 rend()。其函数原型为：

```
const_iterator begin() const;  
iterator begin();  
const_iterator end() const;  
iterator end();  
const_reverse_iterator rbegin() const;  
reverse_iterator rbegin();  
const_reverse_iterator rend() const;  
reverse_iterator rend();
```

下面以例 3-34 为例，说明上述 4 个函数的使用。

例 3-34

```
#pragma warning(disable:4786)  
#include <iostream>  
#include <map>  
using namespace std;  
typedef pair<int,double>mypair;  
void print ( map<int,double,greater<int>>&m)  
{ map<int,double,greater<int>>::iterator it;  
mypair tmp;  
int size=m.size();  
if (size<=0)  
cout<<"* * * * *"<<endl;  
else  
{for(it=m.begin();it!=m.end();it++)  
{tmp=(mypair)*it;  
cout<<tmp.first<<" "; cout<<tmp.second<<endl;  
}  
cout<<endl;  
}  
}  
void main()  
{ map<int,double,greater<int>>m1;  
map<int,double,greater<int>>m3;  
multimap<int,double,greater<int>>m2;
```

```
multimap<int,double,greater<int>>::iterator itmA,itmB;
m1.insert(mypair(1,5.0));
m1.insert(mypair(2,10.5));
m1.insert(mypair(3,11.5));
m1.insert(mypair(4,13.5));
m1.insert(mypair(5,15.1));
cout<<"m1: " <<endl;
print(m1);
m2.insert(mypair(1,9.5));
m2.insert(mypair(2,12.5));
m2.insert(mypair(3,9.2));
m3=m1;
int size=m1.size();
cout<<"The size of map m1: " <<size <<endl;
size=m2.size();
cout<<"The size of map m2: " <<size <<endl;
m3.erase(m3.begin());
cout<<"m3: " <<endl;
print(m3);
    m3.erase(m3.begin(),--m3.end());
cout<<"m3: " <<endl;
print(m3);
m1.erase(2);
cout<<"m1: " <<endl;
print(m1);
m3.clear();
    cout<<"m3: " <<endl;
print(m3);
m1.swap(m3);
    cout<<"m1: " <<endl;
print(m1);
cout<<"m3: " <<endl;
    print(m3);
}
```

例 3-34 的执行效果如图 3-36 所示。



总结 例 3-34 主要讲述了 4 种和迭代器相关函数的使用方法。请读者关注数据类型 pair 的使用方法。

3. map 和 multimap 的高级编程

map/multimap 型容器还提供各种功能相应的成员函数，例如插入、删除、交换、统计元素个数统计、查找元素、元素的随机访问、元素大小比较以及如何获取内存配置器。

(1) map/multimap 的插入和删除

map/multimap 型容器的成员函数 insert() 可用以插入一个对象、一个对象的若干副本或

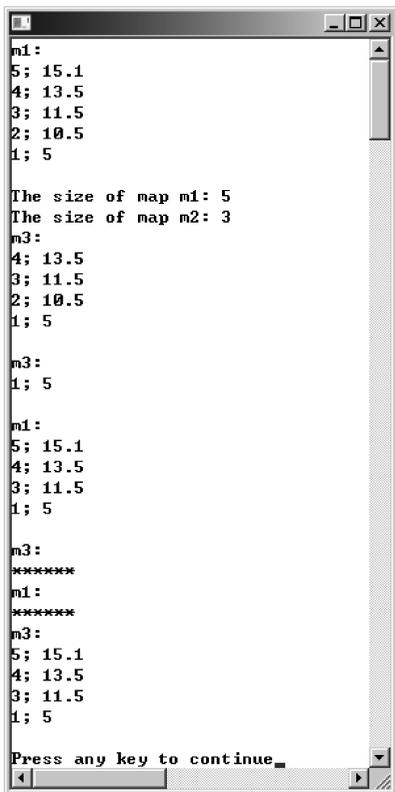


图 3-36 例 3-34 的执行效果

者某个范围内的对象。insert()函数可把一个或多个元素插入到迭代器指示的位置。所插入的元素将出现在迭代器指示的位置之前。其原型为：

```
pair<iterator, bool> insert(const value_type& x);
iterator insert (iterator it, const value_type& x);
void insert (const value_type *first, const value_type *last);
```

第一种形式的功能是将元素 x 插入到 map 的末尾，第二种形式的功能是将元素 x 插入到迭代器 it 之前，第三种形式的功能是将迭代器 (first, last) 指定范围的元素插入到 map 中。

erase()函数可以用来删除由一个迭代器或键值指定的元素，也可以删除某个范围的元素。

其原型为：

```
iterator erase(iterator it);
iterator erase(iterator first, iterator last);
size_type erase (const Key& key);
```

第一种形式的功能是将迭代器 it 所指向的元素删除；第二种形式的功能是将迭代器 (first, end) 所指范围中的元素全部删除；第三种形式的功能是将元素 key 删除。

此外，map 型容器也提供了成员函数 clear()。其原型为：

```
void clear() const;
```

clear() 函数可以删除容器中的所有元素, 并且比 erase() 函数功能强大且更加灵活。clear() 函数等效于:

```
erase(map::begin(), map::end())
```

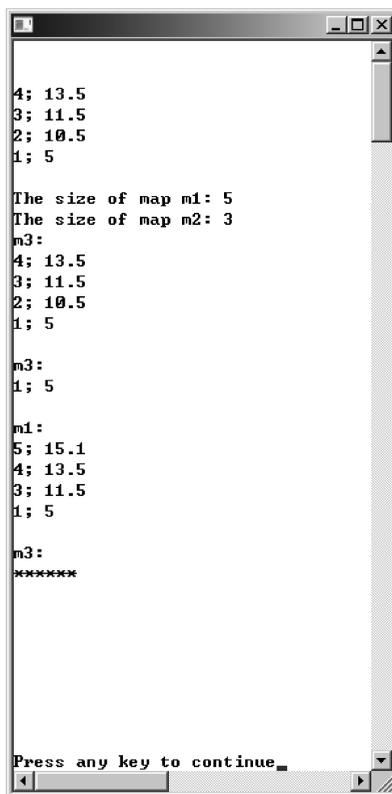
下面使用例 3-35 来详细讲解对 map/multimap 型容器的插入和删除操作。

例 3-35

```
#pragma warning(disable:4786)
#include <iostream>
#include <map>
using namespace std;
typedef pair<int,double> mypair;
void print ( map<int,double,greater<int>>&m)
{ map<int,double,greater<int>>::iterator it;
  mypair tmp;
  int size=m.size();
  if (size<=0)
      cout<<"* * * * *" <<endl;
  else
  {   for(it=m.begin();it!=m.end();it++)
      {
        tmp=(mypair)*it;
        cout<<tmp.first<<" "; <<tmp.second<<endl;
      }
      cout<<endl;
  }
}
void main()
{map<int,double,greater<int>>m1;
  map<int,double,greater<int>>m3;
  multimap<int,double,greater<int>>m2;
  multimap<int,double,greater<int>>::iterator itmA,itmB;
  m1.insert(mypair(1,5.0));
  m1.insert(mypair(2,10.5));
  m1.insert(mypair(3,11.5));
  m1.insert(mypair(4,13.5));
  m1.insert(mypair(5,15.1));
  cout<<"m1: " <<endl;
      print(m1);
  m2.insert(mypair(1,9.5));
  m2.insert(mypair(2,12.5));
  m2.insert(mypair(3,9.2));
  m3=m1;
  int size=m1.size();
  cout<<"The size of map m1: " <<size<<endl;
  size=m2.size();
```

```
cout << "The size of map m2: " << size << endl;
    m3.erase(m3.begin());
cout << "m3: " << endl;
    print(m3);
m3.erase(m3.begin(), --m3.end());
cout << "m3: " << endl;
    print(m3);
m1.erase(2);
cout << "m1: " << endl;
    print(m1);
m3.clear();
cout << "m3: " << endl;
print(m3);
}
```

例 3-35 的执行效果如图 3-37 所示。



```
4; 13.5
3; 11.5
2; 10.5
1; 5

The size of map m1: 5
The size of map m2: 3
m3:
4; 13.5
3; 11.5
2; 10.5
1; 5

m3:
1; 5

m1:
5; 15.1
4; 13.5
3; 11.5
1; 5

m3:
*****

Press any key to continue
```

图 3-37 例 3-35 的执行效果

(2) 元素交换

map 和 multimap 型容器还提供了 swap() 函数, 用以实现元素交换。参与交换的两个对象必须类型一致。swap() 函数的原型为:

```
void swap(map& str);
```

无论是在 `map` 型容器中还是在 `multimap` 型容器中, `swap()` 函数的使用方法是一致的。在实际编程中, 需要把两个变量进行交换。`swap()` 函数一般包括两种形式: `swap(m1, m2)` 和 `m1.swap(m2)`。值得注意的是, 第二种形式只是容器 `m1` 的内容发生改变, 容器 `m2` 中的元素不变。在例 3-35 的基础上进行修改, 即得到例 3-36。

例 3-36

```
m1.swap(m3);
cout << "m1: " << endl;
print(m1);
cout << "m3: " << endl;
print(m3);
```

之后, 即可实现容器 `m1` 和容器 `m2` 之间的内容交换。

(3) 元素个数统计、查找元素和元素的随机访问

实现元素个数统计功能的函数是 `count()`。`count()` 函数的功能是返回元素在 `map` / `multimap` 型容器中重复出现的次数。`map` 型容器中的元素具有唯一性, 不能包含重复的元素。并且 `count()` 函数的返回值只有两个: “0” 或者 “1”。对于 `multimap` 型容器, `count()` 函数的功能和 `map` 型容器中是一致的, 两者的区别在于: `multimap` 型容器中允许出现重复元素, `count()` 函数的返回值不仅局限于 0 和 1, 还会返回元素的重复个数。`count()` 函数的原型为:

```
size_type count (const Key& key) const;
```

实现查找元素功能的函数是 `find()`。其功能是返回指向 `key` 的迭代器, 如果该迭代器定义的是常迭代器, 返回的类型是 `const_iterator`。若和键值 `key` 对应的元素不存在, 则函数返回迭代器 `end()`。还可以使用 STL 的 `find()` 和 `find_if()` 函数。这两个函数可以使用迭代器。通常第一个迭代器指明开始处理的位置, 第二个迭代器指明停止处理的位置。并且第二个迭代器指向的元素不被处理。对于 `multimap` 型容器, 其中允许存在多个相同的元素。在使用 `find()` 函数时, 返回的是所查找到第一个出现的元素迭代器。其原型为:

```
iterator find(const Key& key);
const_iterator find (const Key& key) const;
```

实现元素的随机访问功能的函数包括 `upper_bound()`、`lower_bound()` 以及 `equal_range()`。

`equal_range()` 函数实现的功能是返回容器中元素上下限的两个迭代器, 由于迭代器是常迭代器, 因此 `equal_range()` 函数可以返回 `map` / `multimap` 型容器元素中上下限的两个常迭代器, 若元素没找到, 则返回最后一个元素, 函数返回的是迭代器 `end()`。函数返回的两个迭代器中, 第一个迭代器和 `lower_bound()` 函数的返回值相同; 第二个返回值和 `upper_bound()` 函数的返回值一样。

`upper_bound(Key key)` 函数返回值是返回指向大于 `key` 元素的迭代器; `lower_bound()` 函数的返回值是指向 `key` 前面元素的迭代器, 即 `key` 是“界限”。

上述 3 个函数的原型为:

```
iterator lower_bound (const Key& key);
const_iterator lower_bound (const Key& key) const;
```

```
pair<iterator, iterator> equal_range (const Key& key);  
pair<const_iterator, const_iterator> equal_range (const Key& key) const;  
iterator upper_bound (const Key& key);  
const_iterator upper_bound (const Key& key) const;
```

下面以例 3-37 讲解上述几个函数的使用方法。

例 3-37

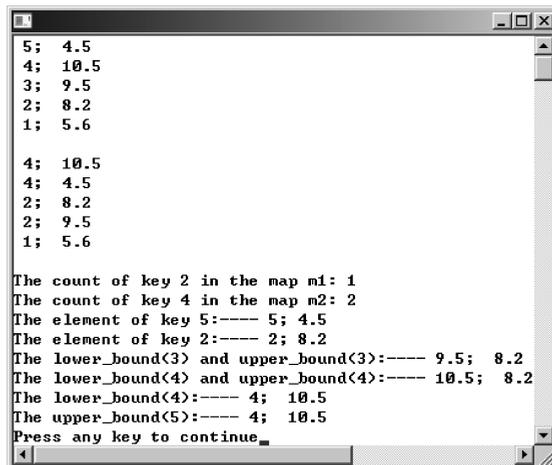
```
#pragma warning(disable:4786)  
#include <iostream>  
#include <map>  
#include <algorithm>  
using namespace std;  
typedef pair<int, double> mypair;  
map<int, double, greater<int>>::iterator it1;  
multimap<int, double, greater<int>>::iterator it2;  
void print (map<int, double, greater<int>>& m)  
{ mypair tmp;  
  for(it1=m.begin(); it1!=m.end(); it1++)  
  { tmp=(mypair)*it1;  
    cout<<" "<<tmp.first<<" "; cout<<tmp.second<<endl;  
  }  
  cout<<endl;  
}  
void printM(multimap<int, double, greater<int>>& m)  
{ mypair tmp;  
  for(it2=m.begin(); it2!=m.end(); it2++)  
  { tmp=(mypair)*it2;  
    cout<<" "<<tmp.first<<" "; cout<<tmp.second<<endl;  
  }  
  cout<<endl;  
}  
void main()  
{  
  typedef map<int, double, greater<int>> Map;  
  typedef multimap<int, double, greater<int>> MMap;  
  Map m1;  
  MMap m2;  
  mypair temp;  
  pair<Map::iterator, Map::iterator> p1;  
  pair<MMap::iterator, MMap::iterator> p2;  
  m1.insert(mypair(1, 5.6));  
  m1.insert(mypair(2, 8.2));  
  m1.insert(mypair(3, 9.5));  
  m1.insert(mypair(4, 10.5));  
  m1.insert(mypair(5, 4.5));
```

```

print(m1);
m2.insert(mypair(1,5.6));
m2.insert(mypair(2,8.2));
m2.insert(mypair(2,9.5));
m2.insert(mypair(4,10.5));
m2.insert(mypair(4,4.5));
printM(m2);
int size=m1.count(2);
int Msize=m2.count(4);
cout<<"The count of key 2 in the map m1: "<<size<<endl;
cout<<"The count of key 4 in the map m2: "<<Msize<<endl;
it1=m1.find(5);
temp=*it1;
cout<<"The element of key 5: - - - - "<<temp.first<<" ";<<temp.second<<endl;
it2=m2.find(2);
temp=*it2;
cout<<"The element of key 2: - - - - "<<temp.first<<" ";<<temp.second<<endl;
p1=m1.equal_range(3);
cout<<" The lower_bound(3) and upper_bound(3): - - - - " <<p1.first->second<<" ";
<<p1.second->second<<endl;
p2=m2.equal_range(4);
cout<<" The lower_bound(4) and upper_bound(4): - - - - " <<p2.first->second<<" ";
<<p2.second->second<<endl;
it1=m1.lower_bound(4);
temp=*it1;
cout<<" The lower_bound(4): - - - - " <<temp.first<<" "; <<temp.second<<endl;
it2=m2.upper_bound(5);
temp=*it2;
cout<<" The upper_bound(5): - - - - " <<temp.first<<" "; <<temp.second<<endl;
}

```

例 3-37 的执行效果如图 3-38 所示。



```

5; 4.5
4; 10.5
3; 9.5
2; 8.2
1; 5.6

4; 10.5
4; 4.5
2; 8.2
2; 9.5
1; 5.6

The count of key 2 in the map m1: 1
The count of key 4 in the map m2: 2
The element of key 5:---- 5; 4.5
The element of key 2:---- 2; 8.2
The lower_bound(3) and upper_bound(3):---- 9.5; 8.2
The lower_bound(4) and upper_bound(4):---- 10.5; 8.2
The lower_bound(4):---- 4; 10.5
The upper_bound(5):---- 4; 10.5
Press any key to continue

```

图 3-38 例 3-37 的执行效果

(4) 元素大小比较

在序列式容器中，可采用 <、> 等算数操作符进行容器大小比较。map/multimap 型容器同样提供了一系列算数或逻辑运算符：==，<，!=，>，>=，<=，还提供了特殊的函数，主要包括 key_comp() 和 value_comp()。

1) 键值比较。key_comp() 函数可用于进行键值的比较。其返回值为 key_comp 类型。key_comp 决定了 map 中元素的排列顺序。key_comp() 函数的原型为：

```
key_compare key_comp() const;
```

下面使用例 3-38 对 key_comp() 函数的使用方法进行阐释。

例 3-38

```
#pragma warning(disable:4786)
#include <iostream>
#include <map>
using namespace std;
void main()
{ typedef map<int,double,less<int>> MAP;
  typedef multimap<int,double,greater<int>> M_MAP;
  MAP m1;
  M_MAP m2;
  MAP::key_compare kc1=m1.key_comp();
  M_MAP::key_compare kc2=m2.key_comp();
  bool r=kc1(2,3);
  bool M_r=kc2(3,4);
  if(r)
  { cout<<" kc1 (2,3) 返回值为 true." <<endl;
  }
  else
  { cout<<" kc1 (2,3) 返回值为 false." <<endl;
  }
  if(M_r)
  { cout<<" kc2 (2,3) 返回值为 true." <<endl;
  }
  else
  { cout<<" kc2 (2,3) 返回值为 false." <<endl;
  }
}
```

在例 3-38 中，由于容器 m1 是按 less 规则排列，容器 m2 是按 greater 规则排列，因此 kc1() 函数的返回值为 true，kc2() 函数的返回值为 false。



图 3-39 例 3-38 的执行效果

2) 实值比较。value_comp()函数可用于进行实值的比较。其返回值为 value_compare 类型。value_compare 决定了 map 中元素的排列顺序。value_compare()函数的原型为:

```
value_compare value_comp() const;
```

其用法和 key_comp()函数类似,即先定义键值类型 vc1 和 vc2,然后通过 vc1 和 vc2 进行排列顺序的测试。对于按 less 规则排序的 map/multimap 型容器,按 greater 规则排序的 map/multimap 型容器,该函数将分别产生不同的返回值。

例 3-39

```
#pragma warning(disable:4786)
#include <iostream>
#include <map>
using namespace std;
typedef map<int,double,less<int>> MAP;
typedef multimap<int,double,greater<int>> M_MAP;
typedef pair<MAP::iterator,bool> mypair;
typedef pair<int,double> myp;
void print (MAP& m)
{
    MAP::iterator it;
    myp temp_p;
    for(it=m.begin(); it!=m.end(); it++)
    {
        temp_p=(*it);
        cout<<temp_p.first<<" "<<temp_p.second<<endl;
    }
    cout<<endl;
}
void print_M (M_MAP& m)
{
    M_MAP::iterator it;
    myp temp_p;
    for (it=m.begin(); it!=m.end(); it++)
    {
        temp_p=(*it);
        cout<<temp_p.first<<" "<<temp_p.second<<endl;
    }
    cout<<endl;
}
void main()
{
    MAP m1;
    M_MAP m2;
    M_MAP::iterator itA, itB;
    mypair p1, p2;
    MAP::value_compare vc1=m1.value_comp();
    M_MAP::value_compare vc2=m2.value_comp();
    p1=m1.insert (map<int,double>::value_type (1,10));
    p2=m1.insert (map<int,double>::value_type (2,5));
    cout<<" 输出容器 m1: " <<endl;
    print (m1);
}
```

```

if(vc1(*p1.first,*p2.first)==true)
{ cout<<"元素(1,10)在元素(2,5)之前"<<endl;
}
else
{ cout<<"元素(1,10)在元素(2,5)之后"<<endl;
}
itA=m2.insert(multimap<int,double>::value_type(1,20));
itB=m2.insert(multimap<int,double>::value_type(2,50));
cout<<"输出容器m2:"<<endl;
print_M(m2);
if(vc2(*itA,*itB)==true)
{ cout<<"元素(1,20)在元素(2,50)之前"<<endl;
}
else
{ cout<<"元素(1,20)在元素(2,50)之后"<<endl;
}
}

```

例 3-39 的执行效果如图 3-40 所示。

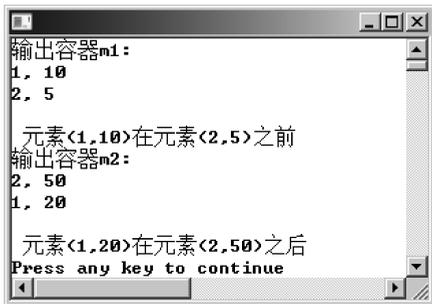


图 3-40 例 3-39 的执行效果



总结 例 3-38 和例 3-39 主要讲述了和排序相关的两个函数 `key_comp()` 和 `value_comp()`。这两个函数的使用方法大体类似，其主要作用是判断容器中元素的排序规则。

(5) 获取内存配置器

`get_allocator()` 函数用于返回 `map` 型容器或 `multimap` 型容器的内存配置器。内存配置器类似于指针的首地址，用于指明对象的初始存储位置。因此，获取容器的内存配置器至关重要。`get_allocator()` 函数的原型为：

```
Allocator get_allocator() const;
```

该函数只能读取内存配置器，不能改变内存配置器，即不能改变容器变量所存储的位置。下面使用例 3-40 对内存配置器进行简单介绍。

例 3-40

```
#pragma warning(disable:4786)
#include <iostream>
```

```
#include <map>
using namespace std;
void main()
{typedef map<int,double> MAP;
MAP::allocator_type m1_alloc;
MAP:: allocator_type m2_alloc;
MAP:: allocator_typem3_alloc;
MAP:: allocator_type m4_alloc;
MAP m1, m2, m3;
m1_alloc =m1.get_allocator();
m2_alloc =m2.get_allocator();
m3_alloc =m3.get_allocator();
MAP m4 (less<int> (), m1_alloc);
m4_alloc =m4.get_allocator();
if (m1_alloc ==m4_alloc)
{ cout << " 配置器相同!" <<endl;
}
else
{ cout << " 配置器不同." <<endl;
}
}
```

例 3-40 的执行效果如图 3-41 所示。



图 3-41 例 3-40 的执行效果

(6) map 作为关联式数组

通常关联式容器不提供元素的直接存取，必须依靠迭代器。但 map 型容器提供下标操作符，支持元素的直接存取。下标操作符的索引值并非元素整数位置，而是元素的键值 key。使用键值索引有时会带来诸多问题。例如，容器中不存在指定键值的元素，会导致自动插入该元素，并且新元素的默认 value 由构造函数提供。通常，所有基本数据类型都提供了默认的构造函数，并都以 0 为初值。

(7) 通常数组可作为 STL 容器

大家都知道 string 可以被视为容器，其内元素均为字符。通常的数组不是类，也不可能提供成员函数。但是可以通过普通指针，使用 STL 的算法实现对普通数组的操作。

例 3-41

```
#include <iostream>
#include <map>
#include <algorithm>
```

```
using namespace std;
void main()
{int col[] = {1,5,7,2,8};
sort(col,col+5);
copy(col,col+5,ostream_iterator<int>(cout," "));
cout<<endl;
}
```

例 3-41 的执行效果如图 3-42 所示。



图 3-42 例 3-41 的执行效果

除了上述方法，还可以通过自定义类或者自定义类模板的形式，实现对数组的访问和操作。



总结 本节主要讲述了 4 种关联式容器：set、multiset、map 和 multimap，并详细讲述了这 4 种容器的各种成员与函数及其使用方法。此外，本节针对每个容器均详细讲述了其构造函数、容量大小，遍历容器，元素查找，插入和删除，交换等。在讲述知识点的同时，也适当补充了部分其他知识。例如 for_each() 函数的使用、sort() 函数的使用等。

3.4 特殊容器用法

3.4.1 bitset（位集合）类模板

例 3-1 中简单介绍了 bitset 类。本节专门讲解 bitset 类的使用。bitset 类模板创造了 1 个内含位或布尔值且大小固定的数组。当需要管理各种标识，并需要以标识的任意组合表现变量时，即可使用 bitset 类模板。在 C 语言或者 C++ 语言中，通常使用型别 long 来作为 bits array，再通过位操作符（&、!、~等）实现操作各个“位”。类 bitset 的优点在于它可容纳任意个数的位，并能提供各项操作。bitset 型容器可视为由 0 和 1 组成的序列，进而实现各种读写操作。

对于既定的 bitset 型容器，其中位的个数是不能改变的。bitset 型容器中变量或对象的位的个数是由模板参数决定的。如果需要使用可变长度的位容器，可考虑使用模板 vector<bool>。

注意：模板参数并不是一个型别，而是一个无符号型整数。

如果在定义对象时，使用的模板参数不同，会导致模板的型别不同。因此，在对 bitset 型对象进行比较和组合时，两个对象或多个对象的位的个数必须相同。类模板 bitset 的构造函数包括以下形式：

```
bitset();
bitset(unsigned long val);
explicitbitset(const string& str, size_t pos = 0, size_t n = -1);
```

第一种形式的构造函数将复位 `bitset` 型对象中的所有位（例如对 `bitset` 型对象进行全部清零）。第二种形式的构造函数的参数为整数，该数值的二进制形式对应的位为 1。第三种形式的构造函数使用字符串对 `bitset` 对象实现初始化。

对于第三种形式的构造函数，参数 `str` 是用来初始化对象的初始字符串；参数 `pos` 的含义表明在字符串 `str` 中，从 `pos` 位置开始的子字符串“才是赋值给类 `bitset` 模板对象的字符串”，即赋值时采用的是 `str` 的某个子串；参数 `n` 说明是从 `pos` 位置开始的，长度为 `n` 的子字符串，该子字符串用来给 `bitset` 类对象赋值。

`bitset` 类模板的成员函数见表 3-3。

表 3-3 `bitset` 类模板的成员函数

函数名称	功能描述
<code>size()</code>	返回位的个数
<code>count()</code>	返回位值为“1”的位的个数
<code>any()</code>	判断是否有任何位为“1”
<code>none()</code>	判断是否所有值为“0”
<code>test (size_t idx)</code>	判断位“idx”是否为“1”
<code>set()</code>	设置所有位为“1”
<code>set (size_t idx, in value)</code>	设定位置“idx”上的值为 value
<code>reset()</code>	复位所有位为“0”
<code>reset (size_t idx)</code>	将位置 idx 上的位设置为“0”
<code>flip()</code>	反转所有位的值
<code>flip (size_t idx)</code>	反转“idx”位置上的位
<code>to_ulong()</code>	返回所有位代表的整数
<code>==</code>	判断两个 <code>bitset</code> 型对象是否“相等”
<code>!=</code>	判断两个 <code>bitset</code> 型对象是否“不相等”
<code>^=</code>	
<code> =</code>	
<code>&=</code>	
<code><<=</code>	所有位向左移动 n 个位置
<code>>>=</code>	所有位向右移动 n 个位置
<code>[size_t idx]</code>	返回 idx 位置上的位值
<code>=</code>	赋值
<code>~</code>	返回该位的补码（反转值）
<code>~()</code>	反转原对象的位，作为产生新对象的初值
<code><<()</code>	向左移动原对象 n 个位置，作为产生新对象的初值
<code>>>()</code>	向右移动原对象 n 个位置，作为产生新对象的初值
<code>&()</code>	两个 <code>bitset</code> 对象逐一“与”操作
<code> ()</code>	两个 <code>bitset</code> 对象逐一“或”操作
<code>^()</code>	两个 <code>bitset</code> 对象逐一“异或”操作
<code>to_string()</code>	以字符串形式表示 <code>bitset</code> 对象中的二进制

下面用例 3-42 阐释 bitset 类模板的使用方法。

例 3-42

```
#include <iostream>
#include <bitset>
#include <string>
using namespace std;
void print(bitset<16>&b)
{   int i=0;
    intbsize=b.size();
    for(i=0;i<bsize;i++)
    {   cout<<b[i];
        }
    cout<<" : "<<"The size of bitset : "<<bsize<<endl;
}
void main()
{   string str="00111111111111111110000";
    bitset<16> b1;
    bitset<16> b2(25);
    bitset<16> b3(str,2,16);
    print(b1);
    print(b2);
    print(b3);
    int c1=b1.count();
    int c2=b2.count();
    int c3=b3.count();
    cout<<"b1 's count : "<<c1<<" , "<<"b2 's count : "<<c2<<" , "<<"b3 's count : "<<c3<<" , "
<<endl;
    bool l1=b1.any();
    bool l2=b2.any();
    bool l3=b3.any();
    cout<<"b1 's any() : "<<l1<<" , "<<"b2 's any() : "<<l2<<" , "<<"b3 's any() : "<<l3<<" , "
<<endl;
    bool n1=b1.none();
    bool n2=b2.none();
    bool n3=b3.none();
    cout<<"b1 's none() : "<<n1<<" , "<<"b2 's none() : "<<n2<<" , "<<"b3 's none() : "<<n3
<<" , "<<endl;
    bool t1=b1.test(2);
    bool t2=b2.test(2);
    bool t3=b3.test(2);
    cout<<"b1 's test() : "<<t1<<" , "<<"b2 's test() : "<<t2<<" , "<<"b3 's test() : "<<t3
<<" , "<<endl;
    cout<<"set(5) : "<<endl;
    b1.set(5,1);
    b2.set(5,1);
```

```
b3.set(5,1);
print(b1);
print(b2);
print(b3);
cout << "reset (5): " << endl;
b1.reset(5);
b2.reset(5);
b3.reset(5);
print(b1);
print(b2);
print(b3);
b1.flip();
b2.flip();
b3.flip();
cout << "flip: " << endl;
print(b1);
print(b2);
print(b3);
unsigned long ul1 = b1.to_ulong();
unsigned long ul2 = b2.to_ulong();
unsigned long ul3 = b3.to_ulong();
cout << " to_ulong : " << endl;
cout << ul1 << endl;
cout << ul2 << endl;
cout << ul3 << endl;
string s1 = b1.to_string();
string s2 = b2.to_string();
string s3 = b3.to_string();
cout << " to_string" << endl;
cout << s1.c_str() << endl;
cout << s2.c_str() << endl;
cout << s3.c_str() << endl;
}
```

例 3-42 的执行效果如图 3-43 所示。



总结 bitset 类模板有些类似于 C 语言中的“位段”，只不过 bitset 是以类模板的形式存在，可以实例化为 bitset 型类或者 bitset 型容器。STL 为 bitset 提供了一系列的成员函数，便于对该型容器的操作和访问。读者应该重点关注 bitset 型容器的用途，并掌握 bitset 型容器的具体使用方法。

3.4.2 stack (栈) 类模板

1. stack 概述

stack 类模板 (栈) 可以实例化一个栈容器 (后进先出, LIFO) 或栈对象。使用 push() 函

```

0000000000000000 :The size of bitset :16
1001100000000000 :The size of bitset :16
1111111111111111 :The size of bitset :16
b1 's count : 0, b2 's count : 3, b3 's count : 16,
b1 's any() : 0, b2 's any() : 1, b3 's any() : 1,
b1 's none() : 1, b2 's none() : 0, b3 's none() : 0,
b1 's test() : 0, b2 's test() : 0, b3 's test() : 1,
set(5):
0000010000000000 :The size of bitset :16
1001110000000000 :The size of bitset :16
1111111111111111 :The size of bitset :16
reset(5):
0000000000000000 :The size of bitset :16
1001100000000000 :The size of bitset :16
1111101111111111 :The size of bitset :16
flip:
1111111111111111 :The size of bitset :16
0110011111111111 :The size of bitset :16
0000010000000000 :The size of bitset :16
to_ulong :
65535
65510
32
to_string
1111111111111111
111111111100110
000000000100000
Press any key to continue_

```

图 3-43 例 3-42 的执行效果

数可以将任意数量的元素置入 stack（栈）中，使用 pop() 函数可以将元素依次反序地从栈中移除。在 STL 中，要使用 stack，则必须要包含头文件 < stack >。在头文件中，stack 类模板的定义如下：

```
namespace std{template <class T, class Container = deque<T>>class stack; }
```

类模板声明中的第 1 个参数代表元素型别；第 2 个参数用来定义 stack 内部存放元素所用的实际容器，其默认值为 deque。选择 deque（没有选择 vector）的目的主要是因为从 deque 类型容器中移除元素时会释放内存，并不必在重新分配内存时复制所有其他元素。例如，

```
stack<int> s1;
```

stack 型容器可以将各项操作转化为内部的对应调用。实际上可以使用任何序列式容器（vector，list）支持 stack，该容器仅需要支持 back()、push_back()、pop_back() 等操作即可。

```
std::stack<int, std::vector<int>>sv;
```

2. 核心成员函数

stack 的核心接口是 3 个成员函数：push()、top() 和 pop()。push() 函数将元素压入 stack 中，pop() 函数从 stack 中移除元素，top() 函数返回栈内的下一个元素。

push() 函数和 pop() 函数并无返回值，top() 函数却返回栈内的下一个元素，但不移除该元素。top() 函数和 pop() 存在一个问题：当栈中为空时，调用该两个函数会导致失败，前提是 stack 必须非空。

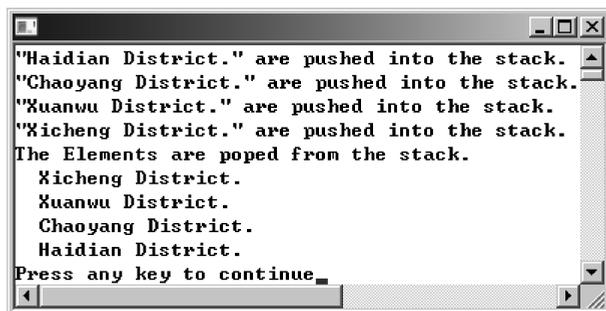
stack 类模板为便于操作和访问，还提供了其余几个函数：size()、empty()、get_allocator() 等。

3. 实例

例 3-43

```
#pragma warning(disable: 4786)
#include <iostream>
#include <stack>
#include <string>
#include <list>
#include <assert.h>
using namespace std;
void main()
{ stack<string, list<string>> s1;
  s1.push("Haidian District. ");
  cout << "\nHaidian District. \n are pushed into the stack.  " << endl;
  s1.push("Chaoyang District. ");
  cout << "\nChaoyang District. \n are pushed into the stack.  " << endl;
  s1.push("Xuanwu District. ");
  cout << "\nXuanwu District. \n are pushed into the stack.  " << endl;
  s1.push("Xicheng District. ");
  cout << "\nXicheng District. \n are pushed into the stack.  " << endl;
  assert(s1.size() == 4);
  assert(s1.top() == "Xicheng District. ");
  cout << "The Elements are popped from the stack. " << endl;
  while(s1.size())
  {   cout << "  " << s1.top() << endl;
      s1.pop();
  }
}
```

例 3-43 的执行效果如图 3-44 所示。



```
"Haidian District." are pushed into the stack.
"Chaoyang District." are pushed into the stack.
"Xuanwu District." are pushed into the stack.
"Xicheng District." are pushed into the stack.
The Elements are popped from the stack.
  Xicheng District.
  Xuanwu District.
  Chaoyang District.
  Haidian District.
Press any key to continue
```

图 3-44 例 3-43 的执行效果



总结 除使用 STL 库定义的类模板之外，程序员还可以根据实际情况创建自定义的 stack 类模板。

3.4.3 queue (队列) 类模板

1. queue 概述

queue 类模板可以实例化 queue，即 FIFO（先进先出）型队列。queue 和 stack 的不同之处在于：queue 是双端口的，元素压入时是从一端，元素移除时是从另一端；而 stack 是单端口的容器，元素的压入和移除均是从同一端口。通俗来讲，queue 是典型的数据缓冲区结构。

若要使用 queue，必须包含头文件 <queue>。在头文件 <queue> 中，queue 类模板的定义如下：

```
namespace std{template <class T, class Container = deque<T>> class queue;}
```

和 stack 一样，类模板声明中的第 1 个参数表示元素的型别；第 2 个参数用来定义队列内部用于存放元素的容器类别，其默认值为 deque。最简单的 queue 实例化容器的例子如下：

```
std::queue<std::string> buffer;
```

实际上，queue 型容器是单纯地把各项操作转化为内部容器的对应调用。可以使用任何序列式容器来支持 queue，前提是该序列式容器要包括成员与函数：front()、back()、push_back()、pop_front() 等即可。假定使用序列式 list 型容器来容纳元素，queue 的声明形式可如下所示：

```
std::queue<std::string, std::list<std::string>>buffer;
```

2. 核心成员函数

queue 的核心接口主要有成员函数 push()、front()、back() 及 pop() 构成。push() 函数会将元素压入 queue 中；front() 返回 queue 中的第 1 个元素；back() 会返回 queue 中最后一个元素，pop() 函数会从 queue 中移除一个元素。

和 stack 一样，queue 型容器的成员函数 pop() 在移除元素时，并不返回该元素。front() 和 end() 返回队列中对应的元素，但并不移除或移动该元素。

和 stack 一样，当 queue 型容器中的元素个数为 0 时，使用 front()、back() 和 pop() 函数会导致调用失败。在使用这些函数之前，最好先调用 size() 或 empty() 函数，判断容器是否为空。

3. 使用 queue 型容器

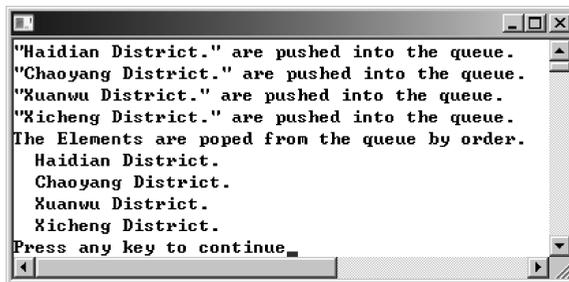
下面主要讲述如何使用 queue 型容器，并使用两个例题进行详细的说明：一个例题用于阐述 STL 中 queue 的使用；另一个例题用于阐述如何使用自定义的队列。

例 3-44

```
#pragma warning(disable:4786)
#include <iostream>
#include <queue>
#include <string>
#include <list>
#include <assert.h>
using namespace std;
void main()
{ queue<string, list<string>> q1; //使用 list 型容器
```

```
q1.push("Haidian District. ");
cout << "\nHaidian District. \n" are pushed into the queue.  " << endl;
q1.push("Chaoyang District. ");
cout << "\nChaoyang District. \n" are pushed into the queue.  " << endl;
q1.push("Xuanwu District. ");
cout << "\nXuanwu District. \n" are pushed into the queue.  " << endl;
q1.push("Xicheng District. ");
cout << "\nXicheng District. \n" are pushed into the queue.  " << endl;
assert(q1.size() ==4);                               //断言
assert(q1.front() == "Haidian District. ");          //断言
cout << "The Elements are popped from the stack by order. " << endl;
while(q1.size())
{   cout << "  " << q1.front() << endl;
    q1.pop();                                         //取出缓冲区中的元素
}
}
```

例 3-44 的执行效果如图 3-45 所示。



```
"Haidian District." are pushed into the queue.
"Chaoyang District." are pushed into the queue.
"Xuanwu District." are pushed into the queue.
"Xicheng District." are pushed into the queue.
The Elements are popped from the queue by order.
  Haidian District.
  Chaoyang District.
  Xuanwu District.
  Xicheng District.
Press any key to continue
```

图 3-45 例 3-44 的执行效果

例 3-45 用来说明自定义队列的使用。当然，STL 中标准的 queue 型队列通常运行速度较快，在使用时，方便性和安全性均较高。但是程序员仍然可以根据自己的需求编写自定义式的队列。例 3-45 选自《C++ 标准程序库自修教程与参考手册》。

例 3-45

```
#pragma warning(disable:4786)
#include <iostream>
#include "queue. cpp"
#include <list>
#include <string>
#include <assert. h>
using namespace std;
void main()
{   try{
Queue <string> q1;
q1.push("Haidian District. ");
```

```

cout << "\"Haidian District. \" are pushed into the queue.  " << endl;
q1.push("Chaoyang District. ");
cout << "\"Chaoyang District. \" are pushed into the queue.  " << endl;
q1.push("Xuanwu District. ");
cout << "\"Xuanwu District. \" are pushed into the queue.  " << endl;
q1.push("Xicheng District. ");
cout << "\"Xicheng District. \" are pushed into the queue.  " << endl;
assert(q1.size() ==4);
assert(q1.front() == "Haidian District. ");
cout << "The Elements are popped from the queue by order. " << endl;
while(q1.size())
{
    cout << "  " << q1.front() << endl;
    q1.pop();
}
}
catch(const exception& e)
{
    cerr << "Exception: " << e.what() << endl;
}
}

```

例 3-45 的执行效果如图 3-46 所示。

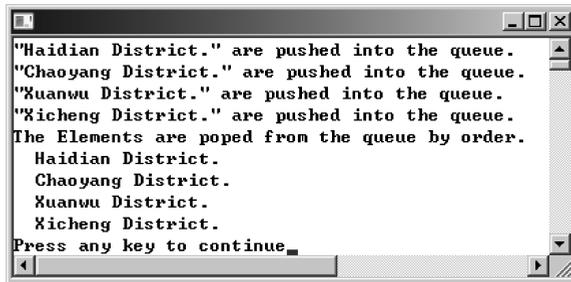


图 3-46 例 3-45 的执行效果

其中自定义模板类 queue 在头文件 queue.cpp 中。queue.cpp 的源代码如下：

```

//queue.cpp
#ifndef QUEUE_CPP
#define QUEUE_CPP
#include <list>
#include <exception>
template <class T>
class Queue {
protected:
    std::list<T> c;
public:
    class ReadEmptyQueue: public std::exception {
public:

```

```
        virtual const char* what() const throw()
        { return "Read Empty Queue. ";
        }
};

typename std::list<T>::size_type size() const {
    return c.size();
}

bool empty() const
{ return c.empty();
}

void push (const T&elem)
{ c.push_back (elem);
}

T pop ()
{ if (c.empty())
    { throw ReadEmptyQueue ();
    }
  T elem (c.front ());
  c.pop_front ();
  return elem;
}

T& front ()
{ if (c.empty())
    { throw (ReadEmptyQueue ());
    }
  return c.front ();
}
};

#endif
```



总结 本小节主要讲述了 queue 类模板的特性及其使用方法，最后讲述自定义队列的使用方法。

3.4.4 priority_queues (优先队列) 类模板

1. priority_queue 概述

priority_queue 类模板可实例化 queue 型容器，其中的元素根据优先级被读取。该类模板的接口和 queue 非常接近。在此类队列中，元素并不是按顺序存储在容器中的，而是按优先级顺序存储在容器中，即在此类队列中，被压入的元素已经按优先级进行了自动排序。默认排序准则是“<”。和常规队列同样，使用 priority_queue 类模板时需要包含头文件 <queue>。例如，

```
#include <queue >
```

priority_queue 类模板的定义如下：

```
namespace std{template <class T, class Container = vector<T>, class Compare = less<typename
Container::value_type>>class priority_queue;}
```

该类模板声明中的第1个参数表示元素的型别；第2个参数定义了内部用来存放元素的容器，其默认容器是 `vector`；第3个元素定义了排序准则，默认情况是以“`operator <`”作为比较准则。例如，

```
std::priority_queue<float> buffer;
```

和前述几种特殊容器相同，`priority_queue` 型容器也是把各项操作转化为容器内部的对应调用。程序开发人员可以使用任何序列类型的容器来支持 `priority_queue`，前提是这些序列式容器要支持 `front()`、`push_back()`、`pop_back()` 等操作即可。

值得一提的是，在 `priority_queue` 型容器中需要用到 STL 中的“堆”（heap）算法，因此其内部容器必须支持随机存取迭代器。比如 `deque` 型容器。

如果程序员需要定义自己的排序准则，必须传递一个函数（或仿函数）作为二元判断式，用于比较两个元素，从而作为排序准则。

2. 核心成员函数

`priority_queue` 的核心接口主要由成员函数 `push()`、`top()` 和 `pop()` 组成。这3个函数的使用方法和前述的几种特殊容器相同。当容器为空时，若调用成员函数 `top()` 和 `pop()`，会调用失败。

和前面介绍的容器不同之处在于：`priority_queue` 提供了多种形式的构造函数。在 C++ STL 中，`stack`、`bitset` 和 `queue` 在各自的模板中，均提供了一种构造函数；而 `priority_queue` 提供了两种构造函数。

1) Visual C++ 中提供的构造函数。其形式如下：

```
explicit priority_queue (constPred& pr = Pred(), const allocator_type& al = allocator_type());
priority_queue (const value_type *first, const value_type *last, constPred& pr = Pred(), const
allocator_type& al = allocator_type());
```

2) STL 提供的构造函数。其形式如下。

```
explicit priority_queue (const Compare& x=Compare(), const Container& =Container());
template <class InputIterator > priority_queue (InputIterator first, InputIterator last, const
Compare& x=Compare(), const Container& = Container());
```

此外，`priority_queue` 提供了两个操作符“`==`”和“`<`”，并提供了 `size()` 函数和 `empty()` 函数。

3. 实例

例 3-46 用于说明 `priority_queue` 的使用方法。

例 3-46

```
#include <iostream>
#include <queue>
#include <vector>
#include <list>
#include <queue>
using namespace std;
void print (double&Ele)
```

```
{cout << Ele << ", ";
}
template <class T>void Out(priority_queue <T, deque<T>, less<T>>& p)
{ while (! p.empty())
  { cout << p.top() << ", ";
    p.pop();
  }
  cout << endl;
}
template <class T>void OutG (priority_queue <T, deque<T>, greater<T>>& pg)
{ while (! pg.empty())
  { cout << pg.top() << ", ";
    pg.pop();
  }
  cout << endl;
}
void main()
{ priority_queue <double, deque<double>, less<double>> p1, p2;
  p1.push (11.5);
  p1.push (22.5);
  p1.push (32.5);
  p1.push (21.1);
  p1.push (15.6);
  p1.push (8.9);
  p1.push (55.0);
  p2 = p1;
  Out (p1);
  p1 = p2;
  priority_queue <double, deque<double>, greater<double>> p3;
  while (p2.size())
  {
    p3.push (p2.top());
    p2.pop();
  }
  OutG (p3);
}
```

例 3-46 的执行效果如图 3-47 所示。

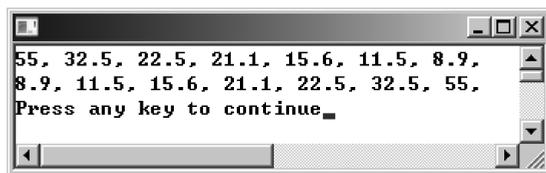


图 3-47 例 3-46 的执行效果

 **总结** 本节主要讲述如何使用 `priority_queue` 类模板，`queue` 与 `priority_queue` 的不同之处在于：后者实现了内部自动排序。程序员可根据实际情况自定义排序准则，也可以自己编写函数或仿函数用于内部优先级的确定。在 Visual C++ 6.0 开发环境中，`priority_queue` 型容器可使用 `vector` 和 `deque` 两种序列式容器；而在使用 `list` 型容器时，出现了编译错误。例 3-46 对两种排序准则均进行了尝试。读者应该认真阅读，以达到触类旁通、举一反三的效果。

3.5 小结

本章主要讲述了 5 部分内容。第一部分介绍了容器的概念，详细介绍容器的基本概念和基本原理；第二部分介绍了序列式容器，包括 `vector`、`list` 和 `deque`；第三部分内容主要涉及这些容器的使用及各自的特殊功能；第四部分重点介绍了 4 种关联式容器，是本章的重点内容，其中 `map` 又较 `set` 更复杂些；第五部分讲述了部分较特殊的容器，如 `bitset`、`stack`、`queue` 和 `priority_queue`。

至此，C++ STL 中的所有容器就全部介绍完了。本章是全书的核心。其中所涉及的大部分例题均是作者参照 ISO/IEC 14882 C++ 国际标准自行编写的。请读者仔细研读，及时掌握这部分内容。

第 4 章

STL 算法

以有限的步骤解决逻辑或数学问题的方法被称为算法。具体的算法是指对解题方案的准确而完整的描述，是一系列解决问题的清晰指令。

算法是用系统的方法描述解决问题的策略机制。

算法能够实现对于一定规范的输入，在有限时间内获得所要求的输出。

C++ STL 也提供了一系列的算法。算法一般用来处理迭代器区间。迭代器是一个“可遍历 STL 容器内全部或部分元素”的对象。迭代器的当前值指向容器中的特定位置。因此，算法可以理解为使用迭代器处理容器中元素的方法。

一般情况下，算法大致分为基本算法、数据结构的算法、数论与代数算法、几何算法、图论算法、动态规划及数值分析、加密算法、排序算法、检索算法、随机化算法、并行算法等。

4.1 算法库简介

STL 算法一般采用“覆盖 (Overwrite)”模式而不是“插入 (Insert)”模式。调用算法时，必须保证目标区间拥有足够的元素空间。当然也可以使用“插入 (Insert)”型迭代器访问容器。

为了增强灵活性和功效，STL 算法允许使用者传递自定义的操作 (函数)，以便于由其调用。这些操作 (函数) 既可以是一般函数，也可以是仿函数。若返回值是 bool 类型，则称之为条件判断式。

算法设计有两个主要的通用部分。首先，这些函数都使用模板来提供通用类型；其次，这些函数都使用迭代器来提供访问容器中数据的通用表示。所以，程序员可以将数值存储于数组中，或者存储在链表中；还可以将对象存储在树结构中。另外，由于指针是一种特殊的迭代器，因此使用时更要注意其各种用法，以防发生失误。

STL 中的算法库包括 4 种算法：非修改性算法、修改性算法、排序和相关操作算法以及删除算法。

非修改性算法不移动容器中元素的次序，也不修改元素的值。这些算法一般通过输入型迭代器和前向型迭代器完成工作，可用于所有标准容器。

修改性算法一般不直接改变容器中元素的值，或者在复制到另一区间的过程中改变元素值。修改性算法还包括了移除性质或删除性质的算法。移除一般只是在逻辑上“移除”元素，不改变容器的大小和容器中的元素个数。“移除”和“删除”是不同的算法。

排序和相关操作算法包括多个排序函数和其他各种函数，包括集合操作等。C++ STL 涵

盖了所有变序性算法。排序一般是指通过对容器中元素的赋值和交换改变元素顺序。排序算法的复杂度通常低于线性算法，要用到随机存取迭代器。变序性算法不能以关联式容器作为目标，这是因为关联式容器的元素都被视为常数，不能变更。

4.2 非修改性算法

非修改性算法不需要使用循环，就能从序列中找出某个元素。此类算法不会修改容器中元素的值，也不会修改元素的次序。下面依次介绍如下算法：

- for_each() 算法
- 元素计数算法
- 最小值和最大值算法
- 搜索算法
- 比较算法

4.2.1 for_each() 算法

for_each() 算法非常灵活，可以非常方便地处理容器中的每一个元素。其函数的定义形式为为：

```
for_each ( Iterator begin Iterator end, proc op )
```

说明：for_each 算法用以实现对区间 [begin, end] 中每个元素均调用进程（或算法）op。下面用 3 个例题来讲解 for_each() 算法的使用方法：

- 1) for_each() 算法的最普通用法。
- 2) 使用 for_each() 算法时使用仿函数。
- 3) 介绍 for_each() 算法的返回值。

例 4-1 的功能是实现将向量中的每个元素的值输出到屏幕上。请读者关注代码中的中文注释。

例 4-1

```
#include <iostream> //包含头文件 iostream
#include <vector> //包含头文件 vector
#include <algorithm> //包含头文件 algorithm
using namespace std; //使用命名空间 std
template <class T> //使用类模板定义类 T
void FillValue(T& vect, int first, int last) //定义 FillValue() 函数,该函数的功能是给对象
{ //vect 的每个元素赋值
    if(last >= first)
    {
        for(int i=first;i<=last;++i) //使用 for 循环,向序列中插入元素
            vect.insert(vect.end(),i);
    }
    else
    {
```

```

        cout << " The indexes is error: last < first. " << endl;
    }
}
void print(intelem) //在屏幕上输出函数的参数 elem
{
    cout << elem << " ";
}
void main() //程序入口 main() 函数
{
    vector <int > myvector; //使用向量模板定义向量 myvector
    FillValue(myvector, 1,10); //给向量赋值
    for_each (myvector.begin(), myvector.end(), print); //输出向量中的每个元素的值
    cout << endl; //换行
}

```

例 4-1 的执行效果如图 4-1 所示。

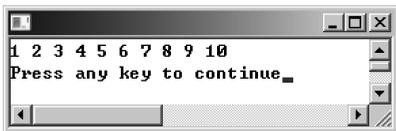


图 4-1 例 4-1 的执行效果

例 4-2 的功能是实现利用仿函数处理容器中每个元素的值。所谓仿函数即使一个类的使用看上去像一个函数。其具体实现是在类中包含 `operator()` 函数，之后这个类就有了类似函数的行为，即所谓的仿函数类。当使用类似函数调用形式 `classname()` 时，`operator()` 函数被自动执行。

提示：例 4-2 的倒数第三行代码：

```
for_each (myvector.begin(), myvector.end(), Multiple <int > (2) );
```

在调用上述代码时，首先产生一个 `Multiple <int >` 型的临时对象，对象的初始赋值为 2，代码实现对容器 `myvector` 中的每个元素值均乘以 2 这一功能。

提示 在实现仿函数时，该类中需要包含 `operator()` 函数，之后这个类就具备了类似函数的行为，即所谓的仿函数类。当使用类似函数调用形式 `classname()` 时，类中的 `operator()` 函数将被自动执行。

仿函数，又称为函数对象，是 STL 六大组件（容器、配置器、迭代器、算法、配接器和仿函数）之一。仿函数虽然小，但却极大地拓展了算法的功能。几乎所有算法都有仿函数版本。

例 4-2

```

#include <iostream >
#include <vector >
#include <algorithm >
using namespace std;

```

```
template <class T >
void FillValue(T& vect, int first, int last)
{
    if(last >= first)
    {
        for(int i=first;i <=last; ++i)
            vect.insert(vect.end(),i);
    }
    else
    {
        cout << " The indexes is error: last < first. " << endl;
    }
}

void print (intelem)
{
    cout << elem << " ";
}

template <class T >
class Multiple{ //定义类 Multiple
private:
    T theValue;
public:
    Multiple(const T& v):theValue (v)
    {
    }

    void operator() (T&elem) const //添加成员函数
    {
        elem* = theValue;
    }
} ;

void main()
{
    vector <int >myvector;
    FillValue(myvector, 1,10);
    for_each (myvector.begin(), myvector.end(), print);
    cout << endl;
    for_each (myvector.begin(), myvector.end(), Multiple<int > (2) ); //参见程序前的解释
    for_each (myvector.begin(), myvector.end(), print);
    cout << endl;
}
```

例 4-2 的执行效果如图 4-2 所示。

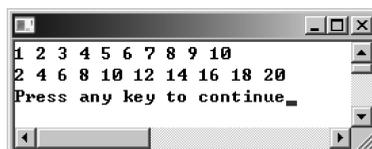


图 4-2 例 4-2 的执行效果

例 4-3 用来说明如何使用 `for_each()` 算法的返回值。具体过程为：首先，定义类 `SUM`；其次，在类 `SUM` 中定义 `operator()` 函数 (`int elem`)，用于计算容器中元素的值的和；最后，用 `operator double()` 函数返回计算求得的总和。



提示 请读者关注例 4-3 源代码中的中文注释。

例 4-3

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
template <class T>
void FillValue(T& vect, int first, int last)
{
    if(last >= first)
    { for(int i=first;i<=last;++i)
        vect.insert(vect.end(),i);
    }
    else
    { cout<<" The indexes is error: last < first."<<endl;
    }
}
void print(intelem)
{
    cout<<elem<<" ";
}
class SUM{ //定义 SUM类
private:
    long sum_D;
public:
    SUM():sum_D(0) {
    }
    void operator() (intelem) //实现求和功能
    { sum_D+=elem;
    }
    operator double() { //返回求得的总和
        return static_cast<double>(sum_D);
    }
};
void main()
{
    vector<int>myvector;
    FillValue(myvector,1,10); //给向量 myvector 赋值
    for_each(myvector.begin(),myvector.end(),print); //输出向量 myvector 中的元素
```

```
cout << endl;
double sum = for_each (myvector.begin(), myvector.end(), SUM()); //求和, 返回结果为 double 类
                                                                型数值
cout << " The Sum: " << sum << endl;
}
```

例 4-3 的执行效果如图 4-3 所示。



图 4-3 例 4-3 的执行效果

总结 本小节重点讲述了 `for_each()` 算法。通过 3 个知识点和 3 个例题，阐释了 `for_each()` 算法的使用方法以及仿函数的相关知识和使用技巧。读者应掌握 `for_each()` 算法的使用以及 `for_each()` 算法的第三个参数采用仿函数的形式、仿函数的调用和仿函数的返回值。

4.2.2 元素计数算法

STL 算法库提供了用于元素计数功能的算法，即求得容器中元素总个数的算法。该功能由 `count()` 算法和 `count_if()` 条件记数算法来实现。其原型为：

```
count( Iterator begin, Iterator end, const T& value)
count(Iterator begin, Iterator end, UnaryPredicate op)
```

说明：`count (Iterator begin, Iterator end)` 算法将统计在迭代器 `[begin, end]` 之内等于 `value` 的元素个数；`count (Iterator begin, Iterator end, UnaryPredicateop)` 算法只有当 `op` 参数为“真”时，才统计元素的个数。

为了说明 `count()` 算法的使用方法，下面使用例 4-4 进行说明。



提示

值得注意的是，例 4-4 使用了 `bind2nd (greater <int > (), 2)`，其意义是数值大于 2 的条件表达式。这种使用形式记住就可以了，随着使用次数的增多，印象会逐渐加深。

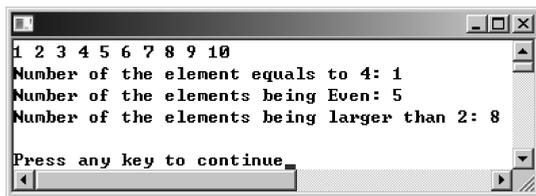
例 4-4

```
#include <functional >
#include <iostream >
#include <vector >
#include <algorithm >
using namespace std;
template <class T >
void FillValue(T& vect, int first, int last) //赋值
```

```
{
    if(last >= first)
    {
        for(int i=first;i <= last;++i)
            vect.insert(vect.end(),i);
    }
    else
    {
        cout << " The indexes is error: last < first. " << endl;
    }
}
void print (intelem) //输出到屏幕
{
    cout << elem << " ";
}
bool isEven (int elem) //是否是偶数
{
    return elem% 2 ==0;
}
void main()
{
    vector <int >myvector; //定义向量
    FillValue(myvector, 1,10); //给向量赋值
    for_each (myvector.begin(), myvector.end(), print); //输出到屏幕
    cout << endl;
    int ct = count (myvector.begin(), myvector.end(), 4); //统计等于 4 的元素个数
    int ctif = count_if (myvector.begin(), myvector.end(), isEven); //统计偶数的个数
    int ctg = count_if (myvector.begin(), myvector.end(), bind2nd (greater <int >(), 2)); //统计大于 2 的元素个数

    cout << " Number of the element equals to 4: " << ct << endl;
    cout << " Number of the elements being Even: " << ctif << endl;
    cout << " Number of the elements being larger than 2: " << ctg << endl;
    cout << endl;}
```

例 4-4 的执行效果如图 4-4 所示。



```
1 2 3 4 5 6 7 8 9 10
Number of the element equals to 4: 1
Number of the elements being Even: 5
Number of the elements being larger than 2: 8
Press any key to continue
```

图 4-4 例 4-4 的执行效果



总结

通过例 4-4 可知，count() 算法既可以统计任何容器（本例是 vector 型容器）的元素个数，也可以通过条件表达式统计满足相应条件的元素个数。读者应注意各种条件的表达形式和使用方法。

4.2.3 最小值和最大值算法

STL 算法库中包含了最大值算法和最小值算法，并且还提供了较复杂的比较形式。

最小值算法的原型为：

```
Iterator min_element (Iterator beg, Iterator end)
Iterator min_element (Iterator begin, iterator end, compFunc op)
```

最大值算法的原型为：

```
Iterator max_element (Iterator beg, Iterator end)
Iterator max_element (Iterator begin, iterator end, compFunc op)
```

上述两种算法原型中的第一种形式是以 “operator <” 进行元素比较；第二种形式是以参数 op 比较两个元素 op (elem1, elem2)，如果第一个元素小于第二个元素，应当返回 true。如果存在多个最小值或最大值，算法返回找到的第一个最小或最大值。这 4 个函数的使用说法详见例 4-5。

例 4-5

```
#include <functional >
#include <iostream >
#include <vector >
#include <algorithm >
using namespace std;
template <class T > void FillValue(T& vect, int first, int last)
{
    if(last >= first)
    {
        for(int i = first; i <= last; ++i)
            vect.insert(vect.end(), i);
    }
    else
    {
        cout << " The indexes is error: last < first. " << endl;
    }
}
void print (intelem)
{
    cout << elem << " ";
}
bool AbsLess (int elem1, int elem2) //使用绝对值进行比较
{
    return abs (elem1) < abs (elem2);
}
void main ()
{ vector <int > myvector;
```

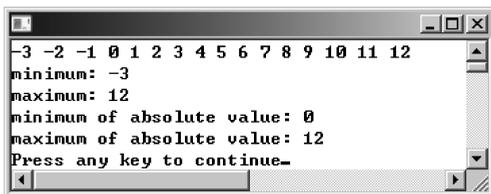
```

FillValue(myvector, -3,12); //给向量赋值
for_each (myvector.begin(), myvector.end(), print); //输出向量型容器中的元素
cout << endl;
cout << " minimum: " << *min_element (myvector.begin(), myvector.end()) << endl;
//获得最小值

cout << " maximum: " << *max_element (myvector.begin(), myvector.end()) << endl; //获得最大值
cout << " minimum of absolute value: " << *min_element (myvector.begin(), myvector.end(), Abs-
sLess)
<< endl; //获得向量型容器中绝对值最小的元素
cout << " maximum of absolute value: " << *max_element (myvector.begin(), myvector.end(), Abs-
sLess)
<< endl; //获得向量型容器中绝对值最大的元素
}

```

例 4-5 的执行效果如图 4-5 所示。



```

-3 -2 -1 0 1 2 3 4 5 6 7 8 9 10 11 12
minimum: -3
maximum: 12
minimum of absolute value: 0
maximum of absolute value: 12
Press any key to continue-

```

图 4-5 例 4-5 的执行效果



总结 通过例 4-5 说明了 `min_element()` 算法和 `max_element()` 算法的使用方法, 实现了无条件的最小值、最大值的求取, 还实现了利用仿函数的有条件的最大值、最小值的求取。其中仿函数的作用是利用元素绝对值进行比较。

4.2.4 搜索算法

STL 算法库提供了一系列搜索算法, 用以获得第一个和给定值相同的元素的位置, 同时也提供了条件搜索算法。

第一组搜索算法 (搜索第一个匹配元素) 的原型为:

```

Iterator find (Iterator begin , Iterator end , const T& value);
Iterator find_if (Iterator begin , Iterator end, UnaryPredicate op);

```

第一种形式用于返回 `[begin, end]` 中第一个“元素值等于 `value`”的元素的位置; 第二种形式用于返回在 `[begin, end]` 中满足条件 `op (elem)` 的第一个元素。如果搜索失败, 返回值为 `end`。

上述搜索算法的使用见例 4-6。

例 4-6

```

#include <functional>
#include <iostream>

```

```
#include <vector>
#include <algorithm>
using namespace std;
template <class T> void FillValue(T& vect, int first, int last)
{
    if(last >= first)
    {
        for(int i=first;i <=last;++i)
            vect.insert(vect.end(),i);
    }
    else
    {
        cout << " The indexes is error: last < first. " << endl;
    }
}
void print (intelem)
{
    cout << elem << " ";
}
void main()
{
    vector <int> myvector;
    FillValue(myvector, -3,12);
    for_each (myvector.begin(), myvector.end(), print);
    cout << endl;
    vector <int>:: iterator pos1;
    pos1 = find (myvector.begin(), myvector.end(), 5);
    vector <int>:: iterator pos2;
    pos2 = find_if (myvector.begin(), myvector.end(), bind2nd (greater <int> (), 3));
    cout << " 第一个等于 5 的元素的位置 : " << distance (myvector.begin(), pos1) + 1 << endl;
    cout << " 第一个大于 3 的元素的位置 : " << distance (myvector.begin(), pos2) + 1 << endl;
}
```

例 4-6 执行效果如图 4-6 所示。



```
-3 -2 -1 0 1 2 3 4 5 6 7 8 9 10 11 12
第一个等于5的元素的位置 :9
第一个大于3的元素的位置 :8
Press any key to continue
```

图 4-6 例 4-6 的执行效果

第二组搜索算法（搜索前 n 个连续匹配值）的原型为：

```
Iterator search_n (Iterator begin , Iterator end , Size count , const T& value );
Iterator search_n (Iterator begin , Iterator end , Size count , const T& value , BinaryPredicate op
);
```

第一种形式用以返回 `[begin, end]` 中的第一组满足条件“连续 `count` 个元素值全等于 `value`”的元素的位置；第二种形式用以返回 `[begin, end]` 中第一组满足条件“`op` 为‘真’”的“连续 `count` 个元素”的起始位置。如果搜索失败，返回值为 `end`。上述搜索算法的使用方法见例 4-7。



提示 关注例 4-7 中“`greater <int > ()`”这个参数的使用形式，这也是类模板的使用方法之一。

例 4-7

```
#include <functional >
#include <iostream >
#include <vector >
#include <algorithm >
using namespace std;
template <class T > void FillValue (T& vect, int first, int last)
{ if(last >= first)
  {
    for(int i=first;i <=last; ++i)
      vect.insert(vect.end(),i);
  }
  else
  {
    cout << " The indexes is error: last < first. " << endl;
  }
}
void print (intelem)
{ cout << elem << " ";
}
void main()
{ vector <int >myvector;
  FillValue(myvector, -3,12);
  for_each (myvector.begin(), myvector.end(), print);
  cout << endl;
  vector <int >:: iterator pos1;
  pos1 = search_n (myvector.begin(), myvector.end(), 4, 3);
  if (pos1 !=myvector.end())
  {
    cout << " 4个连续等于3的元素的起始位置:" << distance (myvector.begin(), pos1) +1 << endl;
  }
  else
  {
    cout << " 没有4个连续等于3的元素" << endl;
  }
  vector <int >:: iterator pos2;
  pos2 = search_n(myvector.begin(),myvector.end(),4,3,greater <int > ());
```

```
if (pos2 != myvector.end())
{
    cout << "4 个连续大于 3 的元素的起始位置 : " << distance(myvector.begin(), pos2) + 1 << endl;
}
else
{
    cout << "第一个大于 3 的元素位置 : " << distance(myvector.begin(), pos2) + 1 << endl;
}
}
```

例 4-7 的执行效果如图 4-7 所示。



图 4-7 例 4-7 的执行效果

第三组搜索算法（搜索第一个子区间）的原型为：

```
Iterator search(Iterator1 begin, Iterator1 end, Iterator2 searchbegin, Iterator2 searchend);
Iterator search(Iterator1 begin, Iterator1 end, Iterator2 searchbegin, Iterator2 searchend, BinaryPredicate op);
```

上述搜索算法都返回和 [searchbegin, searchend] 完全吻合的第一个子区间的第一个元素位置。第二种形式只有在 op 为“真”时才被执行。如果搜索失败，返回值均为 end。这一组算法的使用方法比较简单，详见例 4-8。请读者要仔细体会 checkEven() 函数的定义和调用过程。

提示 在调用搜索算法的过程中，checkEven() 函数的两个参数分别从向量和数组 checkEvenArg 中获得。checkEven() 函数的第一个参数 elem 是向量 myvector 中的逐个元素，第二个参数 even 是从数组 checkEvenArg 中的元素，按顺序逐一带入。

例 4-8

```
#include <functional>
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
template <class T> void FillValue(T& vect, int first, int last)
{
    if (last >= first)
    {
        for (int i = first; i <= last; ++i)
```

```
        vect.insert(vect.end(),i);
    }
    else
    {
        cout << " The indexes is error: last < first. " << endl;
    }
}
void print(intelem)
{
    cout << elem << " ";
}
boolcheckEven(int elem, bool even)
{
    if(even)
    {
        return elem%2 ==0;
    }
    else
    {
        return elem%2 ==1;
    }
}
void main()
{
    vector <int >myvector;
    vector <int >subvector;
    boolcheckEvenArg[3] = {true,false,true};
    FillValue(myvector, -3,12);
    FillValue(subvector, -1,3);
    for_each (myvector.begin(), myvector.end(), print);
    cout << endl;
    for_each (subvector.begin(), subvector.end(), print);
    cout << endl;
    vector<int >:: iterator pos1;
    pos1 = search (myvector.begin(), myvector.end(), subvector.begin(), subvector.end());
    if (pos1 != myvector.end())
    { cout << " 子串在原串中的位置" << distance (myvector.begin(), pos1) +1 << endl;
    }
    else
    { cout << " 没有搜索到需要的子串" << endl;
    }
    vector<int >:: iterator pos2;
    pos2 = search (myvector.begin(), myvector.end(), checkEvenArg, checkEvenArg +3, checkEven);
    cout << " 满足'偶, 奇, 偶'顺序的子串起始位置:" << distance (myvector.begin(), pos2) +1 << endl;
}
```

例 4-8 的执行效果如图 4-8 所示。



图 4-8 例 4-8 的执行效果

第四组搜索算法（搜索最后一个子区间）的原型为：

```
Iterator find_end (Iterator begin , Iterator end , Iterator2 searchbegin, Iterator2 searchend,);  
Iterator find_end (Iterator1 begin , Iterator1 end, Iterator2 searchbegin, Iterator2 searchend,  
BinaryPredicate op );
```

上述搜索算法都返回 $[begin, end]$ 之中“和 $[searchbegin, searchend]$ 完全吻合”的最后一个子区间内的第一个元素位置。其中，第二种形式只有在条件表达式 op 为“真”时才有意义。如果搜索失败，返回值为 end 。这两种算法的使用和前面的使用方法大相径庭，详见例 4-9。

例 4-9

```
#include <functional>  
#include <iostream>  
#include <vector>  
#include <algorithm>  
using namespace std;  
template <class T> void FillValue (T& vect, int first, int last)  
{  
    if (last >= first)  
    {  
        for (int i = first; i <= last; ++i)  
            vect.insert (vect.end(), i);  
    }  
    else  
    {  
        cout << " The indexes is error: last < first. " << endl;  
    }  
}  
void print (int elem)  
{  
    cout << elem << " ";  
}  
void main ()  
{  
    vector <int> myvector;  
    vector <int> subvector;  
    FillValue (myvector, -3, 12);
```

```

FillValue(myvector, -3,6);
FillValue(subvector, -1,3);
for_each (myvector.begin(), myvector.end(), print);
cout << endl;
for_each (subvector.begin(), subvector.end(), print);
cout << endl;
vector<int>:: iterator posl;
posl = find_end (myvector.begin(), myvector.end(), subvector.begin(), subvector.end());
if (posl != myvector.end())
{
    cout << " 最后一个子串在原串中的位置: " << distance (myvector.begin(), posl) + 1 << endl;
}
else
{
    cout << " 没有搜索到需要的子串. " << endl;
}
}

```

例 4-9 的执行效果如图 4-9 所示。



图 4-9 例 4-9 的执行效果

第五组搜索算法（搜索某些元素的第一次出现位置）的原型为：

```

Iterator find_first_of (Iterator begin, Iterator end , Iterator2 searchbegin, Iterator2 search
end);
Iterator find_first_of (Iterator begin, Iterator end , Iterator2 searchbegin, Iterator2 search
end, BinaryPredicate op);

```

上述搜索算法均返回“子串 [searchbegin, searchend]”在 [begin, end] 中第一次出现的位置；第二种形式只有在满足条件表达式 op 为“真”时才有意义。

例 4-9 可以实现两个功能：在原容器（myvector）中搜索和目标容器（subvector）完全吻合的子向量第一次出现的位置；从后面反向搜索和目标容器（subvector）完全吻合的子向量第一次出现的位置。



提示

请读者关注 `vector<int>:: reverse_iterator rpos` 这条语句。

例 4-10

```

#include <functional>
#include <iostream>

```

```
#include <vector>
#include <algorithm>
using namespace std;
template <class T>
void FillValue(T& vect, int first, int last)
{
    if(last >= first)
    {
        for(int i=first;i <=last; ++i)
            vect.insert(vect.end(),i);
    }
    else
    {
        cout << " The indexes is error: last < first. " << endl;
    }
}
void print(intelem)
{
    cout << elem << " ";
}
void main()
{
    vector <int >myvector;
    vector <int >subvector;
    FillValue(myvector, -3,12);
    FillValue(myvector, -3,6);
    FillValue(subvector, -1,3);
    for_each (myvector.begin(), myvector.end(), print);
    cout << endl;
    for_each (subvector.begin(), subvector.end(), print);
    cout << endl;
    vector<int >:: iterator pos1;
    pos1=find_first_of (myvector.begin(), myvector.end(), subvector.begin(), subvector.end());
    if (pos1!=myvector.end())
    {
        cout << " 第一个子串在原串中的位置: " << distance (myvector.begin(), pos1) +1 << endl;
    }
    else
    {
        cout << " 没有搜索到需要的子串. " << endl;
    }
    vector<int >:: reverse_iterator rpos;
    rpos=find_first_of (myvector.rbegin(), myvector.rend(), subvector.begin(), subvector.end());
};
cout << " 原串中最后一个字串的位置: " << distance (myvector.begin(), rpos.base()) << endl;
}
```

例 4-10 的执行效果如图 4-10 所示。

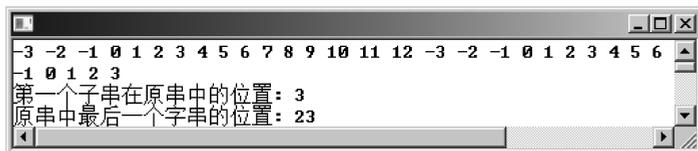


图 4-10 例 4-10 的执行效果

第六组搜索算法（搜索两个连续相等的元素）

使用 `adjacent_find()` 可实现搜寻序列中两个连续相等元素的功能。其原型为：

```
template <class FwdIt> FwdIt adjacent_find (FwdIt first, FwdIt last);
template <class FwdIt, class Pred> FwdIt adjacent_find (FwdIt first, FwdIt last, Pred pr);
```

第一种形式用于返回在 `[first, last]` 中第一对“连续相等两个元素”之中的第一元素位置；第二种形式用于返回 `[first, last]` 中第一对“连续两个元素均使二元判断式 `pr` 为 `true`”的其中第一个元素的位置。第二种形式的本质还是有条件搜索，即搜索容器中符合条件 `pr` 的两个连续元素，并返回第一个元素的位置。

上述搜索算法既可以用于普通的 C 语言数组，也可以用于类似 `vector` 的容器，详见例 4-11。

例 4-11

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
void print(int&Ele)
{   cout << Ele << ", ";
}
bool doubleS(int ele1, int ele2)
{   return ele1*2 == ele2;
}
void main()
{   vector<int> v1;
    vector<int>::iterator it;
    v1.push_back(1);
    v1.push_back(3);
    v1.push_back(3);
    v1.push_back(2);
    v1.push_back(4);
    v1.push_back(5);
    v1.push_back(5);
    v1.push_back(0);
    for_each(v1.begin(), v1.end(), print);
    cout << endl;
    it = adjacent_find(v1.begin(), v1.end());
```

```
if(it! =v1.end())
{ cout << "The position is :" << distance(v1.begin(),it) +1 << endl;
}
it=adjacent_find (v1.begin(), v1.end(), doubleS);
if (it! =v1.end())
{ cout << " The position is : " << distance (v1.begin(), it) +1 << endl;
}
}
```

例 4-11 的执行效果如图 4-11 所示。

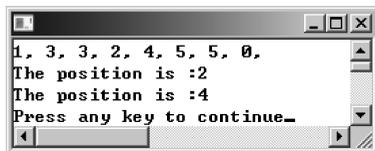


图 4-11 例 4-11 的执行效果

4.2.5 比较算法

STL 算法库提供了 3 个比较算法：`equal()`、`mismatch()`和 `lexicographical_compare()`。

`equal()`算法可以实现两个容器对象的比较，若两个对象相等，返回值 `true`。该算法只能比较属于同一类型的不同变量。如果需要判断两个对象是否相等，要使用 `equal()`算法。若没有找到相异点，返回值是一个 `pair`，以第一个对象的 `end` 元素和第二个对象的对应元素组成；若找到相异点，则以该相异点的对应元素组成一个 `pair`，并被 `equal()` 算法返回。

`mismatch()`算法用于寻找两个容器对象之间两两相异的对应元素，若没有找到相异点，也并不意味着这两个对象完全相同，因为这两个对象的容量还可能不相同。

`lexicographical_compare()`算法属于字典式比较。所谓字典式比较是指一一比较，两个容器中的元素，直到发生以下情况：①若两个元素不相等，则两个元素的比较结果即两个序列的比较结果；②若两个容器中的元素数量不相等，则元素较少的序列小于另一个序列，即若第一序列元素数量较少，则比较结果为 `true`；③如果两序列均没有更多的元素以供比较，则两序列相等，比较结果为 `false`。

上述 3 种算法的原型分别为：

```
template < class InIt1, class InIt2 > bool equal(InIt1 first, InIt1 last, InIt2 x);
template < class InIt1, class InIt2, class Pred > bool equal(InIt1 first, InIt1 last, InIt2 x, Pred pr);
template < class InIt1, class InIt2 > pair < InIt1, InIt2 > mismatch(InIt1 first, InIt1 last, InIt2 x);
template < class InIt1, class InIt2, class Pred > pair < InIt1, InIt2 > mismatch(InIt1 first, InIt1 last, InIt2 x, Pred pr);
template < class InIt1, class InIt2 > bool lexicographical_compare (InIt1 first1, InIt1 last1, InIt2 first2, InIt2 last2);
template < class InIt1, class InIt2, class Pred > bool lexicographical_compare (InIt1 first1, InIt1 last1, InIt2 first2, InIt2 last2, Pred pr);
```

下面以例 4-12 ~ 例 4-14 对上述 3 种算法的使用方法进行说明。

例 4-12

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
void Origin(vector<int>& v)
{   v.push_back (1);
    v.push_back (2);
    v.push_back (3);
    v.push_back (4);
    v.push_back (5);
    v.push_back (6);
    v.push_back (7);
}
void Origin2 (vector<int>& v)
{   v.push_back (3);
    v.push_back (4);
    v.push_back (5);
    v.push_back (6);
    v.push_back (7);
    v.push_back (8);
    v.push_back (9);
}
bool relationship (int e1, int e2)
{   int tmp=e1+2;
    return e2==tmp;
}
void Print (vector<int>& v)
{   vector<int>::iterator it;
    for (it=v.begin(); it!=v.end(); it++)
        cout<<*it<<" ";
    cout<<endl;
}
void main()
{   vector<int> v1, v2;
    Origin (v1);
    Origin2 (v2);
    cout<<" vector v1: " <<endl;
    Print (v1);
    cout<<" vector v2: " <<endl;
    Print (v2);
    bool eq=equal (v1.begin(), v1.end(), v2.begin());
    if (eq)
    {   cout<<" v1 == v2" <<endl;
```

```
    }  
    else  
    {   cout << "v1 ! = to v2" << endl;  
    }  
    bool eq2 = equal(v1.begin(), v1.end(), v2.begin(), relationship);  
    if (eq2)  
    {   cout << "v2[i] == v1[i] + 2" << endl;  
    }  
    else  
    {   cout << "v2[i] ! = v1[i] + 2" << endl;  
    }  
}
```

例 4-12 的执行效果如图 4-12 所示。

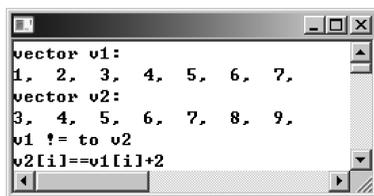


图 4-12 例 4-12 的执行效果

例 4-13 用于说明 mismatch() 算法的使用方法。

例 4-13

```
#include <iostream>  
#include <algorithm>  
#include <list>  
#include <vector>  
using namespace std;  
void Print(int&Ele)  
{   cout << Ele << ", ";  
}  
void main()  
{list <int> l1;  
    vector <int> l2;  
    pair <list <int> ::iterator, vector <int> ::iterator> p1;  
    l1.push_back (1);  
    l1.push_back (2);  
    l1.push_back (3);  
    l1.push_back (4);  
    l1.push_back (7);  
    l1.push_back (5);  
    l2.push_back (1);  
    l2.push_back (2);  
    l2.push_back (3);
```

```
l2.push_back (5);
l2.push_back (6);
l2.push_back (4);
for_each (l1.begin(), l1.end(), Print);
cout << endl;
for_each (l2.begin(), l2.end(), Print);
cout << endl;
p1 = mismatch (l1.begin(), l1.end(), l2.begin());
if (p1.first == l1.end())
{   cout << " no mismatch. " << endl;
}
else
{   cout << " The first mismatch: ";
    cout << *p1.first << ", " << *p1.second << endl;
}
p1 = mismatch (l1.begin(), l1.end(), l2.begin(), less_equal < int > ());
if (p1.first == l1.end())
{   cout << " no mismatch. " << endl;
}
else
{   cout << " The first mismatch: ";
    cout << *p1.first << ", " << *p1.second << endl;
}
}
```

例 4-13 执行效果如图 4-13 所示。

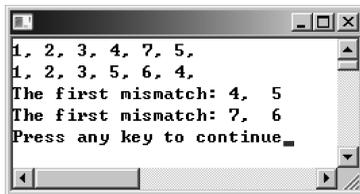


图 4-13 例 4-13 的执行效果

例 4-14 用于说明 lexicographical_compare() 算法的使用方法。

例 4-14

```
#include <iostream>
#include <algorithm>
#include <list>
#include <vector>
using namespace std;
void Print(int&Ele)
{   cout << Ele << ", ";
}
void main()
```

```
{list<int> l1;
    vector<int> l2;
    l1.push_back(1);
    l1.push_back(2);
    l1.push_back(3);
    l1.push_back(4);
    l1.push_back(5);
    l2.push_back(1);
    l2.push_back(2);
    l2.push_back(3);
    l2.push_back(5);
    l2.push_back(4);
    cout<<" l1: " <<endl;
    for_each(l1.begin(), l1.end(), Print);
    cout<<endl;
    cout<<" l2: " <<endl;
    for_each(l2.begin(), l2.end(), Print);
    cout<<endl;
    bool b=lexicographical_compare(l1.begin(), l1.end(), l2.begin(), l2.end());
    if(b)
        { cout<<" l1 < l2." <<endl;
        }
    else
        { cout<<" l1 >= l2." <<endl;
        }
}
```

例 4-14 的执行效果如图 4-14 所示。

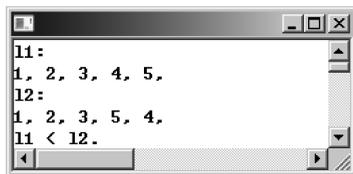


图 4-14 例 4-14 的执行效果

提示 以上三种算法除其各自的一般形式外，均提供了自定义比较规则。使用自定义比较规则时，其实现的功能已经超越了其名称的意义。equal() 和 lexicographical_compare() 算法的返回值均为 bool 类型，而 mismatch() 算法的返回值为 pair 类型。读者应理解这三种算法的意义。

4.3 修改性算法

前面讲述的非修改性算法对其所作用的容器不做任何修改。在实际编程中，程序员经常

需要对所操作的容器进行修改和写操作。STL 提供了诸多功能完善的算法，能够对所作用的容器进行修改的算法被称为修改性算法（或修正序列算法）。此类算法一般通过以下两种方法改变容器中元素的值：

- 1) 在使用迭代器遍历序列时直接改变元素的值。
- 2) 在元素复制过程中改变元素的值。

修改性算法的功能主要包括复制、转换、互换、赋值、替换、逆转、旋转和排列。



提示 在实现本节的上述算法时，其目标容器或目标区间必须不能是关联式容器，因为关联式容器是自动内部排序的。关联式容器的元素均被视为常数，也是不允许作为变量方式使用的，否则无法实现其内部自动排序。

4.3.1 复制

容器复制是指在两个容器之间进行元素传递。STL 提供了复制算法用以实现元素复制。复制算法包括两种形式：`copy()`和`copy_backward()`。其原型为：

```
OutputIterator copy(InputIterator_First, InputIterator_Last, OutputIterator_DestBeg);  
BidirectionalIterator2 copy_backward (BidirectionalIterator1_First, BidirectionalIterator1_Last BidirectionalIterator2_DestEnd);
```

这两种形式具有较大区别，其参数和返回值类型也不相同。

`copy()`算法实现将（`_First`, `_Last`）指定范围的所有元素复制到另一个容器中由`_DestBeg`指定的起始位置。

`copy_backward()`算法的输入参数均是双向迭代器，其返回值也是双向迭代器，唯一不同的是：该算法从指定范围的最后一个元素开始复制，从后向前直到第一个元素。

由上述内容可知，`copy()`算法是正向遍历序列，`copy_backward()`算法是逆向遍历序列。这两种算法的使用方法不同，给程序员也带来些问题。例如，当源区间和目标区间存在重复区域时，如果要把一个区间复制到前端，应使用`copy()`，此时目标位置应在`_First`之前；如果要把一个区间复制到后端，应使用`copy_backward()`。目标位置`_DestEnd`应该在`sourceEnd`之后。总之，如果`_DestEnd`在（`_First`, `_Last`）范围内时，程序员要慎重考虑。

需要注意如下事项。

1) STL 并没有提供`copy_if()`算法。如果希望实现有条件地复制容器中的元素，可以使用`remove_copy_if()`算法。

2) 如果希望在复制过程中逆转元素的顺序，可以使用`reverse_copy()`算法。

3) 程序员要确保目标区间有足够空间，否则需要使用`insert`迭代器。

4) 若要实现两个容器间所有元素的复制，可以不使用`copy()`和`copy_backward()`算法，而使用`assign()`算法。

5) 如果希望在复制过程中删除部分元素，可使用`remove_copy()`和`remove_copy_if()`算法。

6) 如果在复制过程中改变元素的数值，需要使用`transform()`算法和`replace_copy()`算法。

7) 如果目标容器是空容器, 需要使用插入型迭代器 (Insert Iterator)。

例 4-15

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <list>
using namespace std;
void print(int&ele)
{   cout << ele << ", ";
}
void main()
{   int dim[] = {1,3,5,7,9,11,32,54,65};
    vector<int> v1;
    v1.assign(dim,dim+9);
    cout << "vector v1: " << endl;
    for_each (v1.begin(), v1.end(), print);
    cout << endl;
    list<int> l1, l2;
    copy (v1.begin(), v1.end(), back_inserter (l1));
    cout << " list l1: " << endl;
    for_each (l1.begin(), l1.end(), print);
    cout << endl;
    //输出 l2
    l2 = l1;
    cout << " list l2: " << endl;
    copy (l2.begin(), l2.end(), ostream_iterator<int> (cout, " "));
    cout << endl;
    copy_backward (v1.begin() + 2, v1.begin() + 7, l2.end());
    cout << " list l2 (modified) : " << endl;
    copy (l2.begin(), l2.end(), ostream_iterator<int> (cout, " "));
    cout << endl;
}
```

例 4-15 的执行效果如图 4-15 所示。

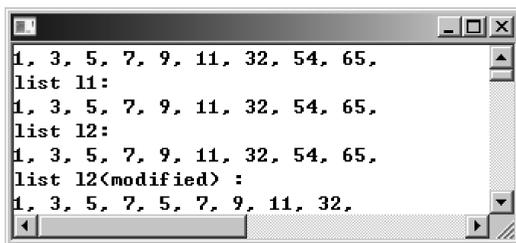


图 4-15 例 4-15 的执行效果

 **提示** ① 在使用 `copy_backward(_first, _end, _dest)` 算法时，源数据是源容器中 `[_first, _end]` 内的元素；在目标容器中，被复制的元素应该保留在目标指针 `_dest` 之前，并且在复制过程中是将 `[_first, _end]` 内的元素从倒数第一个至正数第一个顺序复制，存在指针 `_dest` 前面的内存空间。因此，在指定指针 `_dest` 时，程序员要充分留出足够的存储空间。

② 关注 `back_inserter()` 的使用。

③ 关注如何将数据复制至 `cout` 中。

4.3.2 转换

STL 提供了 `transform()` 算法。该算法既实现将源容器（源区间）的元素复制到目标区间，复制和修改元素一气呵成；也可以实现两个区间的元素合并，最终将结果写入目标区间。其原型为：

```
template<class InIt, class OutIt, class Unop> OutIt transform(InIt first, InIt last, OutIt x, Unop uop);  
template<class InIt1, class InIt2, class OutIt, class Binop> OutIt transform(InIt1 first1, InIt1 last1, InIt2 first2, OutIt x, Binop bop);
```

第一种形式实现的功能：针对 `[sourceBeg, sourceEnd]` 中的每个元素均调用 `uop (ele)` 函数，并将结果写入到以 `destBeg` 为起始的目标区间内。其返回值是“最后一个被转换元素”的下一个位置。在使用该形式时，必须确保具有足够的空间存放这些元素，否则需要使用插入型迭代器。`transform()` 算法可以修改序列中每个元素，这方面和 `for_each()` 算法相似。

第二种形式实现的功能：针对 `[first1, last1]` 中的每个元素，与从 `first2` 开始的第二个源区间的对应元素，调用函数或规则 `bop (ele1, ele2)`，并将结果写入以 `x` 为起始的目标区间。其返回值和第一种形式相同。在使用时，程序员需要确定第二源区间具有足够的容器，至少具有和第一源区间同样的容量。在使用过程中，程序员要确保具有足够的空间（容量），否则需要使用插入型迭代器。值得一提的是，两个源区间可以相同，目标区间也可以和源区间相同。

下面以例 4-16 说明 `transform()` 算法的使用方法。

例 4-16

```
#include <iostream>  
#include <vector>  
#include <list>  
#include <algorithm>  
using namespace std;  
void print(int&Ele)  
{    cout << Ele << ", ";  
}  
void main()
```

```
{
int dim[] = {1,2,3,4,5,6,7,8,9};
    vector<int> v1;
    list<int> l2,l3;
    v1.assign(dim,dim+9);
    cout<<"vector v1: ";
    for_each (v1.begin(),v1.end(),print);
    cout<<endl;
    //l2 中的元素分别乘以 -1
    transform(v1.begin(),v1.end(),back_inserter (l2),negate<int> ());
    cout<<"list l2: ";
    for_each (l2.begin(), l2.end(), print);
    cout<<endl;
    //l2 中的元素分别乘以 10
    transform (l2.begin(),l2.end(),l2.begin(),bind2nd(multiplies<int> (),10));
    cout<<"list l2 (*10): ";
    for_each(l2.begin(),l2.end(),print);
    cout<<endl;
    cout<<"list l2 (-): ";
    //l2 中的元素没有变化
    transform(l2.rbegin(),l2.rend(),ostream_iterator<int> (cout," ", negate<int> ());
    cout<<endl;
    cout<<" list l2 (/2, reverse_direction): ";
    //l2 中的元素没有变化
    transform(l2.rbegin(),l2.rend(),ostream_iterator<int> (cout," ", bind2nd(divides<int>
> (),2));
    cout<<endl;
    //以上是第一种形式的用法
    cout<<"list l2: ";
    for_each (l2.begin(),l2.end(),print);
    cout<<endl;
    cout<<"list l2 (* v1): ";
    transform(v1.begin(),v1.end(),l2.begin(),l2.begin(),multiplies<int> ());
    copy(l2.begin(),l2.end(),ostream_iterator<int> (cout," "));
    cout<<endl;
    cout<<"list l2 (squared): ";
    //l2 中的元素没有变化
    transform (l2.begin(), l2.end(), l2.begin(), ostream_iterator<int> (cout," ", multi-
plies<int> ());
    cout<<endl;
    cout<<"list l2: ";
    for_each (l2.begin(), l2.end(), print);
    cout<<endl;
    cout<<" l3 ( v1 + l2 ): ";
```

```

transform(v1.begin(), v1.end(), l2.begin(), back_inserter(l3), plus<int>());
for_each(l3.begin(), l3.end(), print);
cout << endl;
cout << "cout (l2 - l3) : ";
//l2、l3 中的元素没有变化
transform(l2.begin(), l2.end(), l3.begin(), ostream_iterator<int>(cout, " "), minus<int>());
cout << endl;
}

```

例 4-16 的执行效果如图 4-16 所示。

```

vector v1: 1, 2, 3, 4, 5, 6, 7, 8, 9,
list l2: -1, -2, -3, -4, -5, -6, -7, -8, -9,
list l2(*10): -10, -20, -30, -40, -50, -60, -70, -80, -90,
list l2(-): 90, 80, 70, 60, 50, 40, 30, 20, 10,
list l2(</2, reverse_direction>): -45, -40, -35, -30, -25, -20, -15, -10, -5,
list l2: -10, -20, -30, -40, -50, -60, -70, -80, -90,
list l2(*v1): -10, -40, -90, -160, -250, -360, -490, -640, -810,
list l2(squared): 100, 1600, 8100, 25600, 62500, 129600, 240100, 409600, 656100,

list l2: -10, -40, -90, -160, -250, -360, -490, -640, -810,
l3< v1+l2 >: -9, -38, -87, -156, -245, -354, -483, -632, -801,
cout (l2-l3) : -1, -2, -3, -4, -5, -6, -7, -8, -9,
Press any key to continue_

```

图 4-16 例 4-16 的执行效果

请读者参照程序的执行结果，对照源代码和注释，仔细体会 transform() 算法的用途和使用方法。transform() 算法既可以使用 STL 既定的函数或仿函数作为规则，也可以使用自定义的函数或仿函数作为自定义规则。自定义规则参见例 4-17。例 4-17 使用仿函数的形式实现了 transform() 算法的自定义规则传递。自定义类作为仿函数使用，即可以编写任意算法，实现预想的 transform() 功能，从而实现程序员的预想目的。

例 4-17

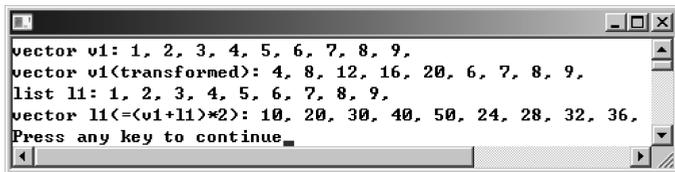
```

#include <iostream>
#include <vector>
#include <list>
#include <algorithm>
#include <functional>
using namespace std;
void print(int&Ele)
{ cout << Ele << " , ";
}
template <class T> class customFun{
public:
    T Parm;
public:
    customFun(const T& _Val): Parm (_Val)

```

```
{  
int operator() (T&elem) const  
{   return elem* Parm;  
}  
};  
template <class T> class customFun2 {  
public:  
    T Parm;  
public:  
    customFun2(const T& _Val): Parm (_Val)  
    {  
        int operator() (T&elem, T& elem2) const  
        {   return (elem+elem2) *Parm;  
        }  
    };  
  
void main()  
{   int dim [] = {1, 2, 3, 4, 5, 6, 7, 8, 9};  
    vector<int> v1, v2;  
    list<int> l1;  
    v1.assign (dim, dim+9);  
    copy (v1.begin(), v1.end(), back_inserter (l1));  
    cout << " vector v1: ";  
    for_each (v1.begin(), v1.end(), print);  
    cout << endl;  
    transform (v1.begin(), v1.begin() +5, v1.begin(), customFun<int> (4));  
    cout << " vector v1 (transformed): ";  
    for_each (v1.begin(), v1.end(), print);  
    cout << endl;  
    cout << " list l1: ";  
    for_each (l1.begin(), l1.end(), print);  
    cout << endl;  
    transform (v1.begin(), v1.end(), l1.begin(), l1.begin(), customFun2<int> (2));  
    cout << " vector l1 (= (v1+l1) *2): ";  
    for_each (l1.begin(), l1.end(), print);  
    cout << endl;  
    // transform (v1.begin(), v1.end(), l1.begin(), ostream_iterator<int> (cout, ", "), custom-  
Fun2<int> (2));  
    // cout << endl;  
    // transform (v1.begin(), v1.end(), l1.begin(), back_inserter (v2), customFun2<int> (2));  
    // cout << " vector l1 (= (v1+l1) *2): ";  
    // copy (v2.begin(), v2.end(), ostream_iterator<int> (cout, ", "));  
    // cout << endl;  
}
```

例 4-17 的执行效果如图 4-17 所示。



```
vector v1: 1, 2, 3, 4, 5, 6, 7, 8, 9,
vector v1(transformed): 4, 8, 12, 16, 20, 24, 28, 32, 36,
list l1: 1, 2, 3, 4, 5, 6, 7, 8, 9,
vector l1(<=v1+l1)*2): 10, 20, 30, 40, 50, 60, 70, 80, 90.
Press any key to continue
```

图 4-17 例 4-17 的执行效果



总结 1) 例 4-17 中定义了两个仿函数类。请读者认真阅读第二个仿函数类，此仿函数是按照 `transform()` 算法的第二种形式编写的。

2) `main()` 函数的末尾包含了 6 行注释语句，前两行是一组，后 4 行是一组，可以使用该组代码替换 `main()` 函数中第二种形式的 `transform()` 算法的使用，即 `main()` 函数中代码第 16 行 ~ 第 19 行。

3) 例 4-17 较完整地阐释了 `transform()` 算法的自定义规则，同时在本书中第一次给出了多参数的仿函数例子，有利于拓开读者的思路和视野，为读者今后编写出更好的仿函数打下基础。

4.3.3 互换

STL 的大部分容器均提供了成员函数 `swap()`，用于实现两个不同容器之间的元素交换。STL 同时还提供了 `swap()` 算法。其原型为：

```
template <class T> void swap(T& a, T& b);
```

和

```
template<classFwdIt1, class FwdIt2> FwdIt2 swap_ranges (FwdIt1 first, FwdIt1 last, FwdIt2 x);
```

第一种形式可用于交换两个类型相同的容器对象，即交换两个容器对象的所有元素。

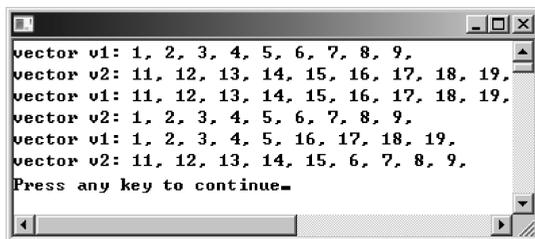
第二种形式可用于交换两个容器中的部分元素。`swap_ranges()` 函数的第一个 `first` 和第二个参数 `last` 用来指定第一个用于交换的容器 1 中的元素的范围；参数 `x` 表示在第二个容器中，以迭代器 `x` 为开始位置，并和容器 1 指定范围相等元素个数的容器 2 的元素范围。下面以例 4-18 为例阐释 `swap()` 算法的使用方法。

例 4-18

```
#include <iostream>
#include <vector>
#include <list>
#include <algorithm>
#include <functional>
using namespace std;
void main()
{
    int dim[] = {1,2,3,4,5,6,7,8,9};
    int dim2[] = {11,12,13,14,15,16,17,18,19};
    vector<int> v1;
```

```
vector<int> v2;
v1.assign(dim,dim+9);
cout<<"vector v1: ";
copy(v1.begin(),v1.end(),ostream_iterator<int> (cout," "));
cout<<endl;
copy (dim2, dim2+9, back_inserter (v2));
cout<<" vector v2: ";
copy (v2.begin(), v2.end(), ostream_iterator<int> (cout," "));
cout<<endl;
swap (v1, v2);
cout<<" vector v1: ";
copy (v1.begin(), v1.end(), ostream_iterator<int> (cout," "));
cout<<endl;
cout<<" vector v2: ";
copy (v2.begin(), v2.end(), ostream_iterator<int> (cout," "));
cout<<endl;
swap_ranges (v1.begin(), v1.begin()+5, v2.begin());
cout<<" vector v1: ";
copy (v1.begin(), v1.end(), ostream_iterator<int> (cout," "));
cout<<endl;
cout<<" vector v2: ";
copy (v2.begin(), v2.end(), ostream_iterator<int> (cout," "));
cout<<endl;
}
```

例 4-18 的执行效果如图 4-18 所示。



```
vector v1: 1. 2. 3. 4. 5. 6. 7. 8. 9.
vector v2: 11. 12. 13. 14. 15. 16. 17. 18. 19.
vector v1: 11. 12. 13. 14. 15. 16. 17. 18. 19.
vector v2: 1. 2. 3. 4. 5. 6. 7. 8. 9.
vector v1: 1. 2. 3. 4. 5. 16. 17. 18. 19.
vector v2: 11. 12. 13. 14. 15. 6. 7. 8. 9.
Press any key to continue
```

图 4-18 例 4-18 的执行效果

4.3.4 赋值

STL 提供了 4 种算法，用于给容器赋值。这 4 种算法分别是 `fill()`、`fill_n()`、`generate()` 和 `generate_n()`。

`fill_n()` 算法和 `generate_n()` 算法只给指定区间的前 `n` 个数值赋值。`fill()` 算法和 `fill_n()` 算法用于给每个元素赋予相同的数值；`generate()` 算法和 `generate_n()` 算法在执行时会调用函数的子进程或仿函数 (`g`)，产生新值，并赋值给容器中的元素。其原型如下：

```
template<classFwdIt, class T> void fill(FwdIt first, FwdIt last, const T& x);
template<classOutIt, class Size, class T> void fill_n (OutIt first, Size n, const T& x);
template<classFwdIt, class Gen> void generate (FwdIt first, FwdIt last, Gen g);
```

```
template <class OutIt, class Pred, class Gen > void generate_n (OutIt first, Dist n, Gen g);
```

值得一提的是, `fill_n()` 算法和 `generate_n()` 算法均指定了赋值区间的起始位置和赋值元素的个数。详见例 4-19。

例 4-19

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int Fibonacci (void)
{
    static int r;
    static int f1 = 0;
    static int f2 = 1;
    r = f1 + f2;
    f1 = f2;
    f2 = r;
    return f1;
}
void main ()
{
    vector<int> v1;
    vector<int> v2(5,0),v3(6,0);
    int dim[]={1,2,3,4,5,6,7,8,9};
    v1.assign(dim,dim+9);
    copy(v1.begin(),v1.end(),ostream_iterator<int>(cout,""));
    cout<<endl;
    fill(v1.begin(),v1.begin()+4,9);
    copy(v1.begin(),v1.end(),ostream_iterator<int>(cout,""));
    cout<<endl;
    fill_n(v1.begin(),5,20);
    copy(v1.begin(),v1.end(),ostream_iterator<int>(cout,""));
    cout<<endl;
    /* - - - */
    generate(v2.begin(),v2.end(),rand);
    copy(v2.begin(),v2.end(),ostream_iterator<int>(cout,""));
    cout<<endl;
    /* - - - */
    generate_n(v2.begin(),3,rand);
    copy(v2.begin(),v2.end(),ostream_iterator<int>(cout,""));
    cout<<endl;
    /* - - 自定义序列 - - */
    generate_n(v3.begin(),6,Fibonacci);
    copy(v3.begin(),v3.end(),ostream_iterator<int>(cout,""));
    cout<<endl;
}
```

例 4-19 的执行效果如图 4-19 所示。

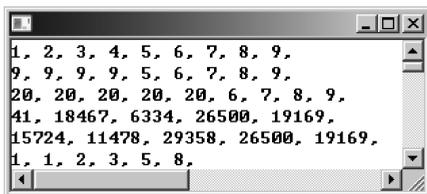


图 4-19 例 4-19 的执行效果

总结 generate()和 generate_n()算法除了可以使用 STL 提供的函数作为子进程或仿函数（用于产生元素），还可以使用自定义的函数产生新的元素值，甚至可以使用特定的算法产生数值序列。例如，在例 4-19 中，main()函数最后即产生 1 个斐波那契数列。

4.3.5 替换

STL 提供了 replace()算法，用以实现代替容器中需要替换的元素。其原型为：

```
template < class FwdIt, class T > void replace (FwdIt first, FwdIt last, const T& vold, const T& vnew);
```

和

```
template < class FwdIt, class Pred, class T > void replace_if (FwdIt first, FwdIt last, Pred pr, const T& val);
```

上述两种形式均实现条件性的替换。第一种形式的功能是替换容器（序列）的 [first, last] 内和 void 值相等的元素，将这些元素的值替换为 vnew。第二种形式的功能是替换容器的 [first, last] 中能够使规则 pr（一元判断式）为 true 的元素，并将这些元素的值替换为 val。下面使用例 4-20 来阐释 replace()算法的使用方法。

例 4-20

```

#include <iostream>
#include <vector>
#include <list>
#include <algorithm>
using namespace std;
void main()
{
    vector<int> v1;
    int dim[] = {1,2,3,4,5,6,7,8,9};
    v1.assign(dim,dim+9);
    copy(v1.begin(),v1.end(),ostream_iterator<int> (cout," "));
    cout<<endl;
    replace(v1.begin(),v1.end(),7,99);
    copy(v1.begin(),v1.end(),ostream_iterator<int> (cout," "));
    cout<<endl;
    replace_if(v1.begin(),v1.end(),bind2nd(less<int>(),5),11);
    copy(v1.begin(),v1.end(),ostream_iterator<int> (cout," "));
    cout<<endl;
}

```

例 4-20 的执行效果如图 4-20 所示。

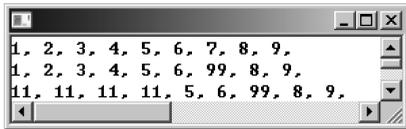


图 4-20 例 4-20 的执行效果

4.3.6 逆转

逆转算法用于将容器或序列中的元素按逆向顺序反转，即第 1 个元素变成倒数第一个元素，第 2 个元素变成倒数第二个元素，依次反转。STL 提供了两种可以实现逆转功能的算法：`reverse()` 和 `reverse_copy()`。

这两种算法均使用双向迭代器，其原型为：

```
template <class BidirectionalIterator> void reverse (BidirectionalIterator first, BidirectionalIterator last)
template <class BidirectionalIterator, class OutputIterator> OutputIterator reverse_copy (BidirectionalIterator first, BidirectionalIterator last, OutputIterator result);
```

`reverse()` 算法会将 `[beg, end]` 中的元素全部逆序；`reverse_copy()` 算法会将 `[sourceBeg, sourceEnd]` 内的元素复制到“以 `destBeg` 起始的目标区间”，并在复制过程中颠倒元素次序。`reverse_copy()` 算法返回区间内最后一个被复制元素的下一位置，即第一个未被覆盖的元素。

在使用过程中，程序员必须确保目标区间足够大，否则需要使用插入型迭代器。

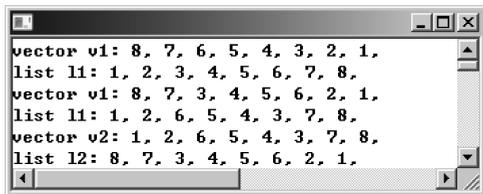
值得注意的是，上述两种算法的参数均是迭代器。通过迭代器可以指定需要逆转的元素的范围，也就是说，既可以逆转整个容器中的所有元素，也可以逆转容器中的部分元素。

例 4-21

```
#include <iostream>
#include <list>
#include <vector>
#include <algorithm>
using namespace std;
void print (int&ele)
{   cout << ele << ", ";
}
void main ()
{int dim[] = {1,2,3,4,5,6,7,8};
  vector<int> v1,v2;
  list<int> l1,l2;
  v1.assign(dim,dim+8);
  reverse(v1.begin(),v1.end());
  cout << "vector v1: ";
```

```
for_each (v1.begin(), v1.end(), print);
cout << endl;
copy (v1.rbegin(), v1.rend(), back_inserter (l1));
cout << " list l1: ";
for_each (l1.begin(), l1.end(), print);
cout << endl;
reverse (v1.begin() + 2, v1.end() - 2);
cout << " vector v1: ";
for_each (v1.begin(), v1.end(), print);
cout << endl;
list<int>:: iterator posf = l1.begin();
list<int>:: iterator pose = l1.end();
advance (posf, 2);
advance (pose, -2);
reverse (posf, pose);
cout << " list l1: ";
for_each (l1.begin(), l1.end(), print);
cout << endl;
reverse_copy (v1.begin(), v1.end(), back_inserter (v2));
cout << " vector v2: ";
for_each (v2.begin(), v2.end(), print);
cout << endl;
cout << " list l2: ";
reverse_copy (l1.begin(), l1.end(), ostream_iterator<int> (cout, ", "));
cout << endl;
}
```

例 4-21 的执行效果如图 4-21 所示。



```
vector v1: 8, 7, 6, 5, 4, 3, 2, 1,
list l1: 1, 2, 3, 4, 5, 6, 7, 8,
vector v1: 8, 7, 3, 4, 5, 6, 2, 1,
list l1: 1, 2, 6, 5, 4, 3, 7, 8,
vector v2: 1, 2, 6, 5, 4, 3, 7, 8,
list l2: 8, 7, 3, 4, 5, 6, 2, 1,
```

图 4-21 例 4-21 的执行效果



总结 通过本节和前面几节的学习，读者应该意识到几个问题：

为什么没有使用 `assign()` 直接赋值一个常量数组到 `list` 型的容器？

`List` 型容器的迭代器在移动时，其使用方法和 `vector` 型容器是不同的。`vector` 型迭代器在移动时可以使用加法运算（如例 4-21 中所示），而 `list` 型迭代器需要使用 `advance()` 函数。本小节在此前介绍了 `advance()`。`advance()` 可以方便地实现容器的迭代器的前进和后退。

只有随机访问型迭代器才可以使用加、减法运算，直接将偏移量加到迭代器上。

4.3.7 旋转

STL 提供了对于容器或序列的旋转算法 `rotate()` 和 `rotate_copy()`。旋转运算是使序列中的元素按照一个环的方式旋转, 而不是简单的左移。`rotate()` 算法将容器中的元素或序列看作一个环, 旋转这些元素直至原来 `middle` 处的元素到达 `first` 位置。`rotate_copy()` 算法会产生一个旋转后的副本, 即在旋转之后会复制参加旋转的元素。其原型为:

```
template < classForwardIterator > void rotate (ForwardIterator first, ForwardIterator middle,
ForwardIterator last)
```

和

```
template < classForwardIterator, class OutputIterator > OutputIterator rotate (
ForwardIterator first, ForwardIterator middle, ForwardIterator last, OutputIterator result)
```

值得指出的是, 对于 `rotate_copy()` 算法的源区间和目标区间是不允许重复的。下面以例 4-22 来阐释上述两种算法的使用方法。

例 4-22

```
#include < iostream >
#include < vector >
#include < algorithm >
using namespace std;
void main()
{
    int dim[] = {1,2,3,4,5,6,7,8,9};
    vector<int> v1,v2;
    v1.assign(dim,dim+9);
    v2.assign(v1.begin(),v1.end());
    cout << "vector v1: ";
    copy(v1.begin(),v1.end(),ostream_iterator<int> (cout," "));
    cout << endl;
    cout << " vector v1 (first rotation): ";
    rotate (v1.begin(), v1.begin() +4, v1.end());
    copy (v1.begin(), v1.end(), ostream_iterator<int> (cout," "));
    cout << endl;
    cout << " vector v1 (second rotation): ";
    rotate_copy (v1.begin(), v1.begin() +3, v1.end(), ostream_iterator<int> (cout," "));
    cout << endl;
    cout << " vector v1 (third rotation): ";
    rotate (v2.begin(), v2.begin() +2, v2.end() -3);
    copy (v2.begin(), v2.end(), ostream_iterator<int> (cout," "));
    cout << endl;
    cout << " vector v1 (fourth rotation): ";
    rotate_copy (v2.begin(), v2.begin() +3, v2.end(), ostream_iterator<int> (cout," "));
    cout << endl;
}
```

例 4-22 的执行效果如图 4-22 所示。

```
vector v1: 1, 2, 3, 4, 5, 6, 7, 8, 9,
vector v1(first rotation): 5, 6, 7, 8, 9, 1, 2, 3, 4,
vector v1(second rotation): 8, 9, 1, 2, 3, 4, 5, 6, 7,
vector v1(third rotation): 3, 4, 5, 6, 1, 2, 7, 8, 9,
vector v1(fourth rotation): 6, 1, 2, 7, 8, 9, 3, 4, 5,
```

图 4-22 例 4-22 的执行效果

4.3.8 排列

本小节主要介绍容器（序列）中元素的“排列元素”“重排元素”和“前向搬移”三种操作。“排列元素”会改变容器（序列）中的元素次序，排序方式为字典式“正规”排序。“重排元素”是指对容器中的所有元素进行随机排序，一般有两种形式：符合均匀分布随机的排序；按指定规则打乱容器（序列）中的元素次序。“前向搬移”是指按指定的一元判断式向前搬移元素，当一元判断式的值为 true 时，算法会向前移动符合条件的元素，其返回值是使一元判断式为“false”的第一个元素位置。

排列元素的原型为：

```
template < class BidirectionalIterator >
bool next_permutation (BidirectionalIterator first, BidirectionalIterator last);
//升序

template < class BidirectionalIterator, class Compare >
bool next_permutation (BidirectionalIterator first, BidirectionalIterator last, Compare comp);
//降序
```

重排元素的原型为：

```
template < class RandomAccessIterator > void random_shuffle (RandomAccessIterator first,
RandomAccessIterator last); //均匀分布

template < class RandomAccessIterator, class RandomNumberGenerator > void
random_shuffle (RandomAccessIterator first, RandomAccessIterator last, RandomNumberGenera-
tor& rand);
//按指定规则
```

前向搬移元素的原型为：

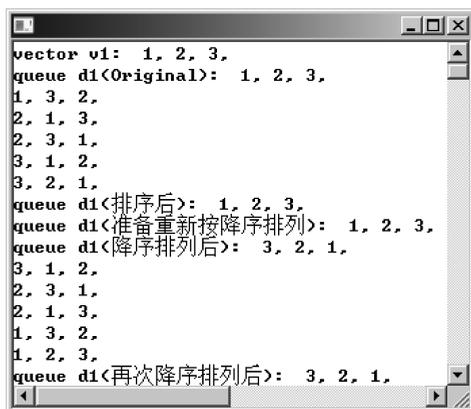
```
template < class BidirectionalIterator, class Predicate >
BidirectionalIterator partition (BidirectionalIterator first, BidirectionalIterator last,
Predicate pred); //不排序

template < class BidirectionalIterator, class Predicate >
BidirectionalIterator stable_partition ( BidirectionalIterator first, BidirectionalIterator
last, Predicate pred);
//会保留一个相对的次序（排序）
```

例 4-23

```
#include <iostream>
#include <vector>
#include <deque>
#include <algorithm>
using namespace std;
void main()
{deque <int> dl;
  vector<int> v1;
  int dim[] = {1,2,3};
  v1.assign(dim, dim+3);
  cout << "vector v1: ";
  copy(v1.begin(), v1.end(), ostream_iterator<int> (cout, ", "));
  cout << endl;
  cout << " queue dl (Original): ";
  copy(v1.begin(), v1.end(), back_inserter(dl));
  copy(dl.begin(), dl.end(), ostream_iterator<int> (cout, ", "));
  cout << endl;
  while (next_permutation(dl.begin(), dl.end()))
  { copy(dl.begin(), dl.end(), ostream_iterator<int> (cout, ", "));
    cout << endl;
  }
  cout << " queue dl (排序后): ";
  copy(dl.begin(), dl.end(), ostream_iterator<int> (cout, ", "));
  cout << endl;
  cout << " queue dl (准备重新按降序排列): ";
  copy(dl.begin(), dl.end(), ostream_iterator<int> (cout, ", "));
  cout << endl;
  while (prev_permutation(dl.begin(), dl.end()))
  { copy(dl.begin(), dl.end(), ostream_iterator<int> (cout, ", "));
    cout << endl;
  }
  cout << " queue dl (降序排列后): ";
  copy(dl.begin(), dl.end(), ostream_iterator<int> (cout, ", "));
  cout << endl;
  while (prev_permutation(dl.begin(), dl.end()))
  { copy(dl.begin(), dl.end(), ostream_iterator<int> (cout, ", "));
    cout << endl;
  }
  cout << " queue dl (再次降序排列后): ";
  copy(dl.begin(), dl.end(), ostream_iterator<int> (cout, ", "));
  cout << endl;
}
```

例 4-23 的执行效果如图 4-23 所示。



```
vector v1: 1, 2, 3,
queue d1(Original): 1, 2, 3,
1, 3, 2,
2, 1, 3,
2, 3, 1,
3, 1, 2,
3, 2, 1,
queue d1<排序后>: 1, 2, 3,
queue d1<准备重新按降序排列>: 1, 2, 3,
queue d1<降序排列后>: 3, 2, 1,
3, 1, 2,
2, 3, 1,
2, 1, 3,
1, 3, 2,
1, 2, 3,
queue d1<再次降序排列后>: 3, 2, 1,
```

图 4-23 例 4-23 的执行效果

例 4-24

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
using namespace std;
classmyRandom //创建一个可产生随机数的仿函数
{public:
    ptrdiff_t operator() (ptrdiff_t max)
    {
        double tmp;
        tmp = static_cast<double> (rand()) /static_cast<double> (RAND_MAX);
        return static_cast<ptrdiff_t> (tmp* max);
    }
};
void main()
{
    vector<int> v1;
    int dim [] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
    v1.assign (dim, dim+9);
    cout<<" vector v1: ";
    copy (v1.begin(), v1.end(), ostream_iterator<int> (cout, ", "));
    //输出最初的 vector 型容器中的元素

    cout<<endl;
    random_shuffle (v1.begin(), v1.end());
    cout<<" vector v1 (random): ";
    copy (v1.begin(), v1.end(), ostream_iterator<int> (cout, ", "));
    cout<<endl;
    //输出打乱顺序后的 vector 型容器中的元素

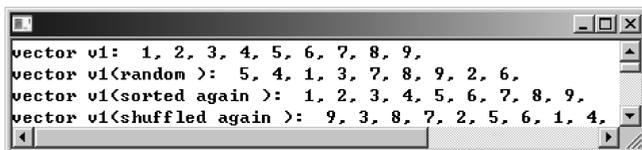
    sort (v1.begin(), v1.end());
    cout<<" vector v1 (sorted again): ";
```

```

copy(v1.begin(),v1.end(),ostream_iterator<int>(cout,""));
cout<<endl; //输出排序后的 vector 型容器中的元素
myRandom mrd;
random_shuffle(v1.begin(),v1.end(),mrd);
cout<<" vector v1 (shuffled again): ";
copy(v1.begin(),v1.end(),ostream_iterator<int>(cout,""));
cout<<endl; //输出第二次打乱顺序后的 vector 型容器中的元素
}

```

例 4-24 的执行效果如图 4-24 所示。



```

vector v1: 1, 2, 3, 4, 5, 6, 7, 8, 9,
vector v1(random ): 5, 4, 1, 3, 7, 8, 9, 2, 6,
vector v1(sorted again ): 1, 2, 3, 4, 5, 6, 7, 8, 9,
vector v1(shuffled again ): 9, 3, 8, 7, 2, 5, 6, 1, 4,

```

图 4-24 例 4-24 的执行效果

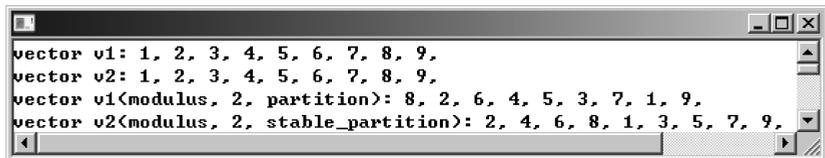
例 4-25 (引自《C++ 标准程序库》)

```

#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
using namespace std;
void main()
{
    int dim[] = {1,2,3,4,5,6,7,8,9};
    vector<int> v1;
    vector<int> v2;
    v1.assign(dim,dim+9);
    v2=v1;
    cout<<"vector v1: ";
    copy(v1.begin(),v1.end(),ostream_iterator<int>(cout,""));
    cout<<endl;
    cout<<" vector v2: ";
    copy(v2.begin(),v2.end(),ostream_iterator<int>(cout,""));
    cout<<endl;
    vector<int>::iterator pos1, pos2;
    pos1=partition(v1.begin(),v1.end(),not1(bind2nd(modulus<int>(),2)));
    pos2=stable_partition(v2.begin(),v2.end(),not1(bind2nd(modulus<int>(),2)));
    cout<<" vector v1 (modulus, 2, partition): ";
    copy(v1.begin(),v1.end(),ostream_iterator<int>(cout,""));
    cout<<endl;
    cout<<" vector v2 (modulus, 2, stable_partition): ";
    copy(v2.begin(),v2.end(),ostream_iterator<int>(cout,""));
    cout<<endl;
}

```

例 4-25 的执行效果如图 4-25 所示。



```
vector v1: 1, 2, 3, 4, 5, 6, 7, 8, 9,
vector v2: 1, 2, 3, 4, 5, 6, 7, 8, 9,
vector v1(modulus, 2, partition): 8, 2, 6, 4, 5, 3, 7, 1, 9,
vector v2(modulus, 2, stable_partition): 2, 4, 6, 8, 1, 3, 5, 7, 9,
```

图 4-25 例 4-25 的执行效果

上述 3 个例题分别讲述了 6 种和排序有关的算法，其功能各不相同。但它们均是通过不同的方式来实现对容器（序列）中元素的排序。看似用途不大，但是各种妙处希望读者多理解，多体会。

4.4 排序及相关操作算法

STL 提供了多种算法用来对容器（序列）中的全部或部分元素进行排序。对于程序员来说，如果局部排序能够满足需要，应尽量使用局部排序。对全体元素进行一次性排序要比时刻维护它们保持有序状态要高效得多。当然，关联式容器可以实现自动排序。

对于一个序列，排序是最简单也是最有力的算法之一。因此也不必为 STL 提供那么多的排序算法而惊讶。STL 提供的排序算法均是功能较强大、机制较完善的。这些算法甚至多达 25 个，有些在前面章节已经介绍了，本章逐步介绍剩余的排序算法。

4.4.1 全部元素排序

`sort()` 算法和 `stable_sort()` 算法支持对容器（序列）中所有元素的排序。`sort()` 算法和 `stable_sort()` 算法需要访问随机访问型迭代器，只能适用于 `vector` 型和 `deque` 型容器。由于 `list` 型容器不支持随机访问型迭代器，所以不能使用这里两个算法，但是 `list` 型容器提供了 `sort()` 成员函数，可用于自身元素的排序。

`sort()` 算法和 `stable_sort()` 的原型为：

```
template <class RanIt> void sort(RanIt first, RanIt last);
template <class RanIt, class Pred> void sort(RanIt first, RanIt last, Pred pr);
```

和

```
template <class RanIt> void stable_sort (RanIt first, RanIt last);
template <class RanIt, class Pred> void stable_sort (RanIt first, RanIt last, Pred pr);
```

`stable_sort()` 算法能够保持序列或容器中的元素相对顺序不改变，而 `sort()` 则不能。上述两种算法均具有两种形式的原型。其第二种形式均以二元判断式 `pr (elem1, elem2)` 作为排序准则。顾名思义，`stable_sort()` 会比 `sort()` 更加稳定、可靠。上述原型中 `RanIt` 代表（随机访问迭代器）。

上述两种算法的默认排序准则是：利用“`operator <`”实现升序排序。可以利用规则表达式实现降序排序，也可以使用自定义的规则表达式，详见例 4-26。

例 4-26

```

#include <iostream>
#include <vector>
#include <deque>
#include <algorithm>
#include <functional>
using namespace std;
void main()
{
int dim[] = {1,2,3,4,5,6,7,8,9};
    deque<int> dl;
    copy(dim,dim+9,back_inserter (dl));
    //copy (dim, dim+9, back_inserter (dl));
    cout<<" deque dl (Original): ";
    copy (dl.begin(), dl.end(), ostream_iterator<int> (cout," "));
    cout<<endl;
    //sort (dl.begin(), dl.end());
    stable_sort (dl.begin(), dl.end());
    cout<<" deque dl (sorted by '<')": ";
    copy (dl.begin(), dl.end(), ostream_iterator<int> (cout," "));
    cout<<endl;
    //sort (dl.begin(), dl.end(), greater<int> ());
    stable_sort (dl.begin(), dl.end(), greater<int> ());
    cout<<" deque dl (sorted by '>')": ";
    copy (dl.begin(), dl.end(), ostream_iterator<int> (cout," "));
    cout<<endl;
}

```

例 4-26 的执行效果如图 4-26 所示。

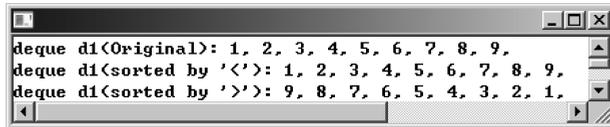


图 4-26 例 4-26 的执行效果

用户可以自定义二元判断式算法。算法的定义如下：

```

bool CustomDefineFun (T elem1, T elem2)
{
    bool res=0;
    ... //使用 elem1 和 elem2 进行运算，并对 res 的值进行修改
    return res; //一定要返回 bool 型变量值
}

```

上述省略部分在此仅举两例：

- 1) 若 T 为 int 型，则省略部分可为：res = (elem1 > elem2)。
- 2) 若 T 为 string 型，则省略部分可为：res = (elem1. length() > elem2. length())。

4.4.2 局部排序

STL 还提供了局部排序算法：`partial_sort()`和 `partial_sort_copy()`。其原型为：

```
void partial_sort ( RandomAccessIterator_First, RandomAccessIterator_SortEnd, RandomAccessIterator_Last );  
  
void partial_sort ( RandomAccessIterator_First, RandomAccessIterator_SortEnd, RandomAccessIterator_Last,  
                  BinaryPredicate_Comp );  
  
RandomAccessIterator partial_sort_copy ( InputIterator_First1, InputIterator_Last1, RandomAccessIterator_First2,  
                                       RandomAccessIterator_Last2 );  
  
RandomAccessIterator partial_sort_copy ( InputIterator_First1, InputIterator_Last1, RandomAccessIterator_First2,  
                                       RandomAccessIterator_Last2, BinaryPredicate_Comp );
```

1. `partial_sort()` 算法

`partial_sort()` 算法的功能是实现对 `[_First, _Last]` 内的元素进行排序，排序后 `[_First, _SortEnd]` 内的元素处于有序状态。`partial_sort()` 算法还可以使用二元判断式 `_Comp (elem1, elem2)`，排序后 `[_First, _SortEnd]` 内的元素是有序的。`partial_sort()` 算法仅仅是对容器（序列）中的部分元素进行排序，在需要的情况下，不仅可以节约时间，还不会对剩余元素进行不必要的排序。若参数 `_SortEnd` 和参数 `_Last` 相等，则可实现对全部元素的排序。

经过 `partial_sort()` 算法排序之后的容器（序列）中，有序的元素被放在容器（序列）的前面，非有序的部分放在容器的后面。

`partial_sort()` 算法的默认排序规则是 “operator <”。

2. `partial_sort_copy()` 算法

`partial_sort_copy()` 算法将容器（序列）的元素从 `[_First1, _Last1]` 复制到目标 `[_First2, _Last2]`，同时对这部分元素进行排序。其返回值是“最后一个被复制元素”的下一位置。被排序和复制的元素数量是源区间 `[_First1, _Last1]` 和目标区间 `[_First2, _Last2]` 两者所含元素数量中较小的那个数量。

若源区间 `[_First1, _Last1]` 的元素数量小于目标区间 `[_First2, _Last2]` 的元素数量，则容器（序列）中所有元素均参与排序。

`partial_sort_copy()` 算法还可使用二元判断式来指定排序规则。

通过以上阐释可知，对于 `partial_sort_copy()` 算法，其功能和 `partial_sort()` 算法基本相似，只不过需要指定一个独立的容器去接受复制。目标独立容器中的元素是有序的。



提示 上述几种适用于排序的算法均使用随机访问型迭代器。随机访问型迭代器可以适用于 `deque` 型和 `vector` 型容器，还适用于普通的 C 语言数组。

例 4-27

```
#include <iostream>  
#include <deque>
```

```

#include <vector>
#include <algorithm>
#include <functional>
using namespace std;
void main()
{
int dim[] = {1,2,4,6,5,8,7,9,3};           //初始数组,用来初始化 vector 型和 deque 型容器
    vector<int> v1,v2,v3(9);
    deque<int> d1,d2,d3(9);
    v1.assign(dim,dim+9);                 //初始化 v1
    copy(v1.begin(),v1.end(),back_inserter(d1)); //初始化 d1
    cout<<" Dim (Original) is below : " <<endl;
    copy(dim, dim+9, ostream_iterator<int> (cout," ")); //输出数组 dim
    cout<<endl;
    cout<<" vector v1 (Original) is below : " <<endl;
    copy(v1.begin(), v1.end(), ostream_iterator<int> (cout," ")); //输出 v1
    cout<<endl;
    cout<<" deque d1 (Original) is below : " <<endl;
    copy(d1.begin(), d1.end(), ostream_iterator<int> (cout," ")); //输出 d1
    cout<<endl;
    v2=v1;
    d2=d1;
    partial_sort(v1.begin(), v1.begin()+5, v1.end()); //局部排序 v1 的前 5 个元素
    cout<<" vector v1 is sorted by the partial_sort() algorithm. : " <<endl;
    copy(v1.begin(), v1.end(), ostream_iterator<int> (cout," ")); //输出局部排序之后的 v1 的结果
    cout<<endl;
    partial_sort(d1.begin(), d1.begin()+5, d1.end()); //局部排序 d1
    cout<<" deque d1 is sorted by the partial_sort() algorithm. : " <<endl;
    copy(d1.begin(), d1.end(), ostream_iterator<int> (cout," ")); //输出局部排序之后的 d1
    cout<<endl;
    cout<<" - - - - -" <<endl;
    cout<<" vector v2 is as same as the original v1 : " <<endl;
    copy(v2.begin(), v2.end(), ostream_iterator<int> (cout," ")); //输出 v2 和 v1 的初始内容相似
    cout<<endl;
    cout<<" deque d2 is as same as the original d1: " <<endl; //输出 d2 的内容和 d1 的初始内容相似
    copy(d2.begin(), d2.end(), ostream_iterator<int> (cout," "));
    cout<<endl;

    partial_sort(v2.begin(), v2.begin()+5, v2.end(), greater<int>()); //对 v2 的前 5 个元素进行降序排序
    cout<<" vector v2 is sorted by the partial_sort() algorithm (降序) . : " <<endl;
    copy(v2.begin(), v2.end(), ostream_iterator<int> (cout," ")); //输出降序排序之后的 v2

```

```
cout << endl;
partial_sort (d2.begin(), d2.begin() + 5, d2.end(), greater<int> ());
//降序排序 d2 的前 5 个元素

cout << " deque d2 is sorted by the partial_sort() algorithm ( 降序 ) . : " << endl;
copy (d2.begin(), d2.end(), ostream_iterator<int> (cout, ", "));
//输出排序后的 d2

cout << endl;

partial_sort_copy (dim, dim + 9, v3.begin(), v3.end());
//对初始数组 dim 排序, 并复制到 v3 中

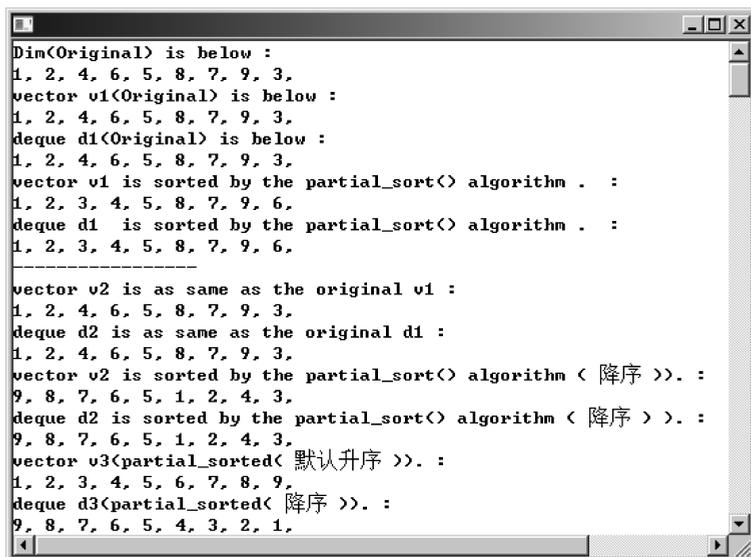
cout << " vector v3 (partial_sorted ( 默认升序 ) ) . : " << endl;
copy (v3.begin(), v3.end(), ostream_iterator<int> (cout, ", "));
//输出 v3

cout << endl;
partial_sort_copy (v3.begin(), v3.end(), d3.begin(), d3.end(), greater<int> ()); //对 v3
进行降序排序, 并复制到 d3 中

cout << " deque d3 (partial_sorted ( 降序 ) ) . : " << endl;
copy (d3.begin(), d3.end(), ostream_iterator<int> (cout, ", "));
//输出 d3

cout << endl;
}
```

例 4-27 的执行效果如图 4-27 所示。



```
Dim(Original) is below :
1, 2, 4, 6, 5, 8, 7, 9, 3,
vector v1(Original) is below :
1, 2, 4, 6, 5, 8, 7, 9, 3,
deque d1(Original) is below :
1, 2, 4, 6, 5, 8, 7, 9, 3,
vector v1 is sorted by the partial_sort() algorithm . :
1, 2, 3, 4, 5, 8, 7, 9, 6,
deque d1 is sorted by the partial_sort() algorithm . :
1, 2, 3, 4, 5, 8, 7, 9, 6,
-----
vector v2 is as same as the original v1 :
1, 2, 4, 6, 5, 8, 7, 9, 3,
deque d2 is as same as the original d1 :
1, 2, 4, 6, 5, 8, 7, 9, 3,
vector v2 is sorted by the partial_sort() algorithm ( 降序 ) . :
9, 8, 7, 6, 5, 1, 2, 4, 3,
deque d2 is sorted by the partial_sort() algorithm ( 降序 ) . :
9, 8, 7, 6, 5, 1, 2, 4, 3,
vector v3(partial_sorted( 默认升序 ) ) . :
1, 2, 3, 4, 5, 6, 7, 8, 9,
deque d3(partial_sorted( 降序 ) ) . :
9, 8, 7, 6, 5, 4, 3, 2, 1,
```

图 4-27 例 4-27 的执行效果

4.4.3 根据某个元素排序

STL 提供了 `nth_element()` 算法。该算法可以对指定区间内的元素进行排序, 并使第 n 个

位置上的元素就位，即所有在位置 n 之前的元素都小于等于它，所有在位置 n 上的元素都大于等于它，由此可以得到根据位置 n 上的元素分割开来的两个子序列。第一子序列的元素全部小于第二子序列的元素。如果程序开发人员只需要 n 个最大或最小元素，但不要求这些元素是已排序的，此时本算法会提供很大帮助。

由上述内容可知，在使用 `nth_element()` 算法之后的序列中，元素是已排序的，并且所有小于第 n 个元素的元素出现在该元素的前面，而大于第 n 个元素的元素出现在该元素的后面。其原型为：

```
template <class RandomAccessIterator > void nth_element ( RandomAccessIterator first, RandomAccessIterator nth,
RandomAccessIterator last);
template <class RandomAccessIterator, class Compare > void nth_element (RandomAccessIterator first,
RandomAccessIterator nth, RandomAccessIterator last, Compare comp);
```

在上述两种形式的原型中，迭代器属于随机访问型迭代器。第一种形式属于常规形式，第二种形式包含二元判断式，并使用二元判断式 `comp (elem1, elem2)` 作为排序规则。

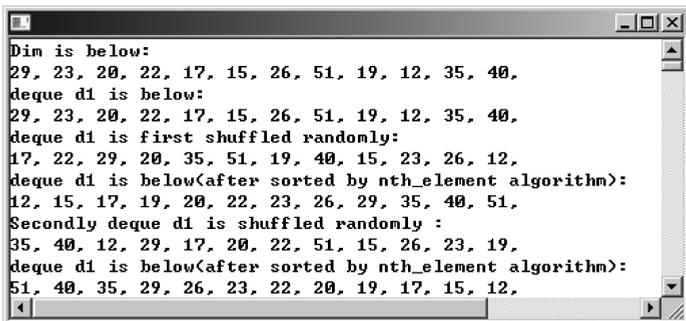
`nth_element()` 算法和 `partial_sort()` 算法有相似之处。其区别在于：`partial_sort()` 算法在实现局部排序时，指定位置作为划分整个区间的界线；而 `nth_element()` 算法在实现排序时，是指定区间中的某个位置（元素）作为划分整个区间的界线。

例 4-28

```
#include <iostream>
#include <deque>
#include <algorithm>
#include <functional>
using namespace std;
void main()
{int dim[] = {29,23,20,22,17,15,26,51,19,12,35,40};
  deque<int> d1;
  copy(dim, dim+12, back_inserter (d1));
  cout << " Dim is below: " << endl;
  copy (dim, dim+12, ostream_iterator<int> (cout, " "));
  cout << endl;
  cout << " deque d1 is below: " << endl;
  copy (d1.begin(), d1.end(), ostream_iterator<int> (cout, " "));
  cout << endl;
  random_shuffle (d1.begin(), d1.end());
  cout << " deque d1 is first shuffled randomly: " << endl;
  copy (d1.begin(), d1.end(), ostream_iterator<int> (cout, " "));
  cout << endl;
  nth_element (d1.begin(), d1.begin()+4, d1.end());
  cout << " deque d1 is below (after sorted by nth_element algorithm): " << endl;
  copy (d1.begin(), d1.end(), ostream_iterator<int> (cout, " "));
  cout << endl;
```

```
random_shuffle (dl.begin(), dl.end());  
cout << " Secondly deque dl is shuffled randomly : " << endl;  
copy (dl.begin(), dl.end(), ostream_iterator<int> (cout, " "));  
cout << endl;  
nth_element (dl.begin(), dl.begin() + 6, dl.end(), greater<int> ());  
cout << " deque dl is below (after sorted by nth_element algorithm): " << endl;  
copy (dl.begin(), dl.end(), ostream_iterator<int> (cout, " "));  
cout << endl;  
}
```

例 4-28 的执行效果如图 4-28 所示。



```
Dim is below:  
29, 23, 20, 22, 17, 15, 26, 51, 19, 12, 35, 40.  
deque dl is below:  
29, 23, 20, 22, 17, 15, 26, 51, 19, 12, 35, 40.  
deque dl is first shuffled randomly:  
17, 22, 29, 20, 35, 51, 19, 40, 15, 23, 26, 12.  
deque dl is below(after sorted by nth_element algorithm):  
12, 15, 17, 19, 20, 22, 23, 26, 29, 35, 40, 51.  
Secondly deque dl is shuffled randomly :  
35, 40, 12, 29, 17, 20, 22, 51, 15, 26, 23, 19,  
deque dl is below(after sorted by nth_element algorithm):  
51, 40, 35, 29, 26, 23, 22, 20, 19, 17, 15, 12.
```

图 4-28 例 4-28 的执行效果



提示

程序的执行效果可能和读者想象的不一樣。請仔細閱讀、認真思考。

4.4.4 堆 (Heap) 操作算法

在计算机算法领域，“堆” (Heap) 通常是指一种组织序列元素的方式。“堆”的第一个元素通常是具有最大值的元素。STL 的算法库中提供了部分堆操作算法：push_heap()、pop_heap()、sort_heap()、make_heap() 等。

就其应用于排序而言，“堆”是一种特别的元素组织方式，即以序列式群集形式存在的二叉树。“堆”具有两大性质：堆中的第一个元素的值总是最大；能够在对数时间内增加或删除一个元素。

堆的默认排序规则为“operator <”。程序开发人员还可以使用自定义的排序准则。下面分别讲述各种“堆”相关的算法。

1. make_heap() 算法

make_heap() 算法的原型为：

```
void make_heap (RandomAccessIterator_First, RandomAccessIterator_Last);  
void make_heap (RandomAccessIterator_First, RandomAccessIterator_Last, BinaryPredicate_Comp);
```

上述两种形式均可实现将 [_First, _last] 中的元素转化为“堆”。参数_Comp 是二元判断式，是排序准则。使用“堆”处理的条件是：元素多于 1 个。如果仅有 1 个元素，没必

要使用“堆”。也可以认为单个元素本身就是“堆”。

2. push_heap() 算法

push_heap() 算法可实现在其参数 (指定区间) 中插入一个元素, 新插入的元素放在堆栈末尾, 即尾指针处。使用 push_heap() 算法时, 必须保证区间原有的元素已是既定的“堆”。其原型为:

```
void push_heap ( RandomAccessIterator_First, RandomAccessIterator_Last );
void push_heap ( RandomAccessIterator_First, RandomAccessIterator_Last, BinaryPredicate_Comp );
```

3. pop_heap() 算法

pop_heap() 算法的作用是删除在迭代器 [_First, _Last] 指定范围 (“堆”) 内的元素序列中的最大元素 (第一个元素)。剩余的其余元素成为一个新的“堆”。其原型为:

```
void pop_heap ( RandomAccessIterator_First, RandomAccessIterator_Last );
void pop_heap ( RandomAccessIterator_First, RandomAccessIterator_Last, BinaryPredicate_Comp );
```

4. sort_heap() 算法

sort_heap() 算法的作用是对其参数迭代器指定的范围内元素进行排序。其原型为:

```
void sort_heap ( RandomAccessIterator_First, RandomAccessIterator_Last );
void sort_heap ( RandomAccessIterator_First, RandomAccessIterator_Last, BinaryPredicate_Comp );
```

sort_heap() 算法可以使堆 [beg, end] 转化为一个已序的序列。算法执行之后, 该区间及其区间内的元素, 就不再是“堆”了。



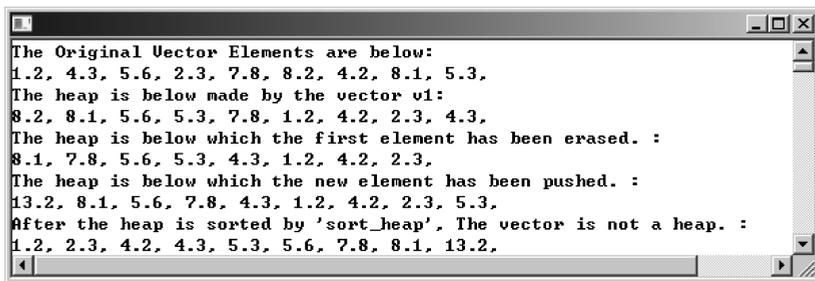
提示 堆的内部数据结构为二叉树。输出的堆数据有时看上去有些乱。其实并不乱, 堆数据在内存中是以二叉树的形式存在的。

例 4-29

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
void main()
{ //序列中元素发生变化之后,二叉树全部重新排列
    vector<double> v1;
    double dim[]={1.2, 4.3, 5.6, 2.3, 7.8, 8.2, 4.2, 8.1, 5.3};
    int cnt = sizeof(dim)/sizeof(double);
    v1.assign(dim, dim + cnt);
    cout << "The Original Vector Elements are below: " << endl;
    copy(v1.begin(), v1.end(), ostream_iterator<double> (cout, " "));
    cout << endl;
    //make_heap
    make_heap (v1.begin(), v1.end());
    cout << " The heap is below made by the vector v1: " << endl;
    copy (v1.begin(), v1.end(), ostream_iterator<double> (cout, " "));
    cout << endl;
```

```
pop_heap (v1.begin(), v1.end());
v1.pop_back();           //移除最后一个元素
cout << " The heap is below which the first element has been erased : " << endl;
copy (v1.begin(), v1.end(), ostream_iterator<double> (cout, ", "));
cout << endl;
//push
v1.push_back (13.2);
push_heap (v1.begin(), v1.end());
cout << " The heap is below which the new element has been pushed : " << endl;
copy (v1.begin(), v1.end(), ostream_iterator<double> (cout, ", "));
cout << endl;
//sort_heap; after sort_heap, the vector will not be a heap.
sort_heap (v1.begin(), v1.end());
cout << " After the heap is sorted by \'sort_heap \', The vector is not a heap. : " << endl;
copy (v1.begin(), v1.end(), ostream_iterator<double> (cout, ", "));
cout << endl;
}
```

例 4-29 的执行效果如图 4-29 所示。



```
The Original Vector Elements are below:
1.2, 4.3, 5.6, 2.3, 7.8, 8.2, 4.2, 8.1, 5.3.
The heap is below made by the vector v1:
8.2, 8.1, 5.6, 5.3, 7.8, 1.2, 4.2, 2.3, 4.3.
The heap is below which the first element has been erased. :
8.1, 7.8, 5.6, 5.3, 4.3, 1.2, 4.2, 2.3,
The heap is below which the new element has been pushed. :
13.2, 8.1, 5.6, 7.8, 4.3, 1.2, 4.2, 2.3, 5.3.
After the heap is sorted by 'sort_heap', The vector is not a heap. :
1.2, 2.3, 4.2, 4.3, 5.3, 5.6, 7.8, 8.1, 13.2.
```

图 4-29 例 4-29 的执行效果

4.4.5 容器合并、交集和差集算法

本节介绍的算法用来将两个容器或区间的元素合并、交集和差集。其中合并运算主要包括总和、并集、交集等。

1. 两个有序集合的总和

merge() 算法可用来处理两个有序集合的总和。其原型为：

```
OutputIterator merge (InputIterator first1, InputIterator last1, InputIterator first2, InputIterator last2, OutputIterator x);
OutputIterator merge (InputIterator first1, InputIterator last1, InputIterator first2, InputIterator last2, OutputIterator x,
Compare comp);
```

上述两种形式均是将两个区间 [first1, last1] 和 [first2, last2] 内的元素合并，并将合并后的所有元素保存在以 x 为起始的目标区间内。目标区间内的所有元素都将按顺序排列。其返回值为目标区间的最后一个被复制元素的下一位置。二元判断式 comp 是可有可无的，

用户可以自定义二元判断式以作为序列的排序准则。merge()算法对两个源区间没有任何变化。

源区间(源容器)中的元素应该是有序的。在多数情况下,merge()算法可以将两个无序的容器(源区间)合并到一个无序的目标区间中。通常,使用copy()函数会更方便些。在使用merge()算法时,要保证目标区间足够大,否则需要使用插入型迭代器。

一般情况下,目标区间和源区间不允许重复。但是,merge()算法没有保证新生成的区间(容器)中元素的唯一性。

list型容器提供了一个特殊成员函数merge(),用于合并两个list型容器中的元素。

2. 两个已排序集合的并集

set_union()算法用于实现将有序的两个区间内的元素合并,获得以指定起始位置开始的目标区间。新生成区间内的元素来自于两个源区间,但是重复的元素被唯一化。其原型为:

```
OutputIterator set_union(InputIterator1 _First1, InputIterator1 _Last1, InputIterator2 _First2,
InputIterator2 _Last2, OutputIterator _Result);
OutputIterator set_union (InputIterator1 _First1, InputIterator1 _Last1, InputIterator2 _
First2, InputIterator2 _Last2, OutputIterator _Result, BinaryPredicate _Comp);
```

其中[_First1, _Last1]和[_First2, _Last2]表示两个源区间,参数_Result代表新生成的目标区间的起始位置。

当源区间内已经含有重复的元素时,使用set_union()算法会保留这些重复的元素。此时元素的重复个数是两个源区间中内的重复个数较大的那个数值。

_Comp是一个二元判断式,可作为排序准则。

3. 两个有序集合的交集

set_intersection()算法用于实现将有序的两个区间的元素合并。新生成区间内的元素应该是那些同时属于两个源区间(容器)的元素。如果两个源区间内原来存在重复元素,新生成的目标区间内也包含重复元素,重复个数为两个源区间中较小的那个重复个数。其原型为:

```
OutputIterator set_intersection (InputIterator1 _First1, InputIterator1 _Last1, InputIterator2
_First2, InputIterator2 _Last2, OutputIterator _Result);
OutputIterator set_intersection (InputIterator1 _First1, InputIterator1 _Last1, InputIterator2
_First2, InputIterator2 _Last2, OutputIterator _result, BinaryPredicate _Comp);
```

其返回值是输出型迭代器,迭代器指向目标区间的最后一个被“合并”元素的下一个位置。目标区间内的所有元素均按顺序排列。

4. 两个有序集合的差集

set_difference()算法实现将实现两个源区间之间的差集,新生成区间内的元素是仅只包含在第一个源区间中的元素。其原型为:

```
OutputIterator set_difference (InputIterator1 _First1, InputIterator1 _Last1, InputIterator2
_First2, InputIterator2 _Last2, OutputIterator _result);
OutputIterator set_difference (InputIterator1 _First1, InputIterator1 _Last1, InputIterator2
_First2, InputIterator2 _Last2, OutputIterator _Result, BinaryPredicate _Comp);
```

第一个源区间中所有存在于第二个源区间中的元素将不存在于新生成区间中。和前面的合并算法一样,新生成区间内的元素均是有序的。

若源区间内有重复元素,则目标区间内相应的也会包含重复元素。重复元素的个数是第

一源区间内的重复个数减去第二源区间内的相应重复个数；若第二个源区间内的重复个数大于第一源区间内的相应重复个数，则目标区间内的对应重复个数将等于 0。

算法对源区间不做任何修改，并且两个源区间都是有序的。目标区间和源区间不允许重叠。

另外，STL 中还存在另一种算法 `set_symmetric_difference()`。该算法可以实现两个源区间的合并，并生成新区间。新生成区间中不包含两个源区间中共同存在的元素。和前面的合并算法一样，新生成的目标区间的元素均是有序的。其原型为：

```
OutputIterator set_symmetric_difference ( InputIterator1 _First1, InputIterator1 _Last1, InputIterator2 _First2,
                                         InputIterator2 _Last2, OutputIterator _Result);
OutputIterator set_symmetric_difference ( InputIterator1 _First1, InputIterator1 _Last1, InputIterator2 _First2,
                                         InputIterator2 _Last2, OutputIterator _Result, BinaryPredicate _Comp);
```

5. 连贯的有序区间的合并

`inplace_merge()` 算法可以将两个有序区间 `[beg1, end1]` 和 `[beg2, end2]` 的元素合并，使区间 `[beg1, end2]` 成为两者之和。其原型为：

```
void inplace_merge ( BidirectionalIterator _First, BidirectionalIterator _Middle, BidirectionalIterator _Last);
void inplace_merge ( BidirectionalIterator _First, BidirectionalIterator _Middle, BidirectionalIterator _Last,
                   BinaryPredicate _Comp);
```

下面以例 4-30 来阐释上述几种算法的使用方法。

例 4-30

```
#pragma warning(disable:4786)
#include <iostream>
#include <list>
#include <set>
#include <algorithm>
#include <functional>
using namespace std;
/*
merge, set_union, set_difference, set_intersection,
*/
void print (int&elem)
{
    cout << elem << ", ";
}
void main()
{
    int dim [] = {1, 2, 3, 4, 5, 6, 7, 8};
    int dim2 [] = {3, 4, 5, 6, 7, 8, 9, 10};
    list<int> l1;
    set<int> s1;
    for (int i=0; i<sizeof (dim) /sizeof (int); i++)
```

```

    l1.push_back (dim [i]);
for (i=0; i<sizeof (dim2) /sizeof (int); i++)
    s1.insert (dim2 [i]);
cout << " list l1: " <<endl;
copy (l1.begin(), l1.end(), ostream_iterator<int> (cout," "));
cout <<endl;
cout << " list s1: " <<endl;
copy (s1.begin(), s1.end(), ostream_iterator<int> (cout," "));
cout <<endl;
cout << " l1 and s1 are merged: " <<endl;
merge (l1.begin(), l1.end(), s1.begin(), s1.end(), ostream_iterator<int> (cout," "));
cout <<endl;
cout << " l1 and s1 are merged by algorithm set_union: " <<endl;
set_union (l1.begin(), l1.end(), s1.begin(), s1.end(), ostream_iterator<int> (cout," "));
cout <<endl;
cout << " l1 and s1 are merged by algorithm set_difference: " <<endl;
set_difference (l1.begin(), l1.end(), s1.begin(), s1.end(), ostream_iterator<int>
(cout," "));
cout <<endl;
cout << " l1 and s1 are merged by algorithm set_intersection: " <<endl;
set_intersection (l1.begin(), l1.end(), s1.begin(), s1.end(), ostream_iterator<int>
(cout," "));
cout <<endl;
cout << " l1 and s1 are merged by algorithm inplace_merge: " <<endl;
copy (s1.begin(), s1.end(), back_inserter (l1));
list<int>::iterator pos;
pos=find (l1.begin(), l1.end(), 3);
pos=find (++pos, l1.end(), 3);
pos++;
//int d=distance (l1.begin(), pos);
inplace_merge (l1.begin(), pos, l1.end());    //inplace_merge 没有实现排序的效果
l1.sort();
for_each (l1.begin(), l1.end(), print);
cout <<endl;
}

```

例 4-30 的执行效果如图 4-30 所示。

```

list l1:
1, 2, 3, 4, 5, 6, 7, 8.
list s1:
3, 4, 5, 6, 7, 8, 9, 10.
l1 and s1 are merged:
1, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7, 8, 8, 9, 10.
l1 and s1 are merged by algorithm set_union:
1, 2, 3, 4, 5, 6, 7, 8, 9, 10.
l1 and s1 are merged by algorithm set_difference:
1, 2.
l1 and s1 are merged by algorithm set_intersection:
3, 4, 5, 6, 7, 8.
l1 and s1 are merged by algorithm inplace_merge:
1, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7, 8, 8, 9, 10.

```

图 4-30 例 4-30 的执行效果

4.4.6 搜索算法

1. `binary_search()` 算法

STL 针对有序区间提供了搜索算法 `binary_search()`，可用以实现在有序区间中搜寻指定元素。其原型为：

```
bool binary_search (ForwardIterator _First, ForwardIterator _Last, const Type& _Val);  
bool binary_search (ForwardIterator _First, ForwardIterator _Last, const Type& _Val, BinaryPred-  
icate _Comp);
```

以上两种算法形式均用来判断有序区间 `[_First, _Last]` 中是否包含值为 `_Val` 的元素。`_Comp` 是一个可有可无的二元判断式，用来作为排序准则。如果搜寻被查找元素的位置，应使用 `lower_bound()`、`upper_bound()` 和 `equal_range()`。

使用 `binary_search()` 算法之前，程序员要确保被搜索的区间必须是有序的。

2. `includes()` 算法

`includes()` 算法可用于在指定源区间中检查若干值是否存在。Visual C++ 6.0 的 msdn 文档中提供的该算法的原型为：

```
inline bool includes (InputIterator1 First1, InputIterator1 Last1, InputIterator2 First2, InputIt-  
erator2 Last2)
```

而 C++ 语言的 ISO/IEC14482 标准中提供了两种函数形式，其中一种与上述形式相同，另一种形式如下所示：

```
Bool includes (InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIt-  
erator2 last2, Compare comp)
```

3. 搜索第一个或（和）最后一个可能位置

STL 还提供了 `lower_bound()`、`equal_range()` 和 `upper_bound()` 算法。`lower_bound()` 算法返回第一个“大于等于 `value`”的元素位置。这是被查找元素的第一个位置。`equal_range()` 算法和 `lower_bound()` 算法一样，要求被搜索的源区间是已序的。`equal_range()` 算法的返回值其实是 `lower_bound()` 和 `upper_bound()` 的共同返回值。

注意：使用 `lower_bound()` 和 `upper_bound()` 算法时，如果未找到指定的元素，其返回值 `end()` 指向位置。

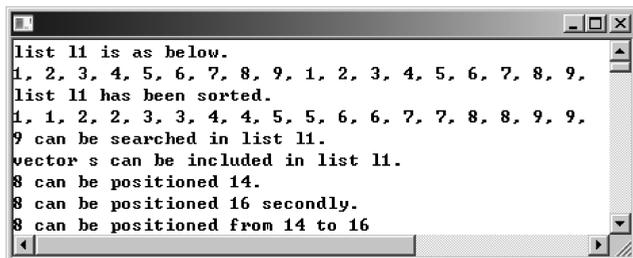
以上 3 个算法的使用方法见例 4-31。

例 4-31

```
#include <iostream>  
#include <algorithm>  
#include <list>  
#include <vector>  
#include <functional>  
using namespace std;
```

```
void main()
{
    int dim[] = {1,2,3,4,5,6,7,8,9};
    list<int> l1;
    list<int>::iterator posL,posU;
    pair<list<int>::iterator,list<int>::iterator> range_Iterator;
    copy (dim, dim+9, back_inserter (l1));
    copy (dim, dim+9, back_inserter (l1));
    cout<<" list l1 is as below. " <<endl;
    copy (l1.begin(), l1.end(), ostream_iterator<int> (cout," "));
    cout<<endl;
    l1.sort();
    cout<<" list l1 has been sorted. " <<endl;
    copy (l1.begin(), l1.end(), ostream_iterator<int> (cout," "));
    cout<<endl;
    bool sh=binary_search (l1.begin(), l1.end(), 9);
    if (sh)
    {
        cout<<" 9 can be searched in list l1. " <<endl;
    }
    else
    {
        cout<<" 9 can't be searched in list l1. " <<endl;
    }
    vector<int> s;
    int dim2 [] = {3, 4, 5, 6};
    s.assign (dim2, dim2+4);
    sh=includes (l1.begin(), l1.end(), s.begin(), s.end());
    if (sh)
    {
        cout<<" vector s can be included in list l1. " <<endl;
    }
    else
    {
        cout<<" vector s can't be included in list l1. " <<endl;
    }
    posL=lower_bound (l1.begin(), l1.end(), 8);
    posU=upper_bound (l1.begin(), l1.end(), 8);
    range_Iterator=equal_range (l1.begin(), l1.end(), 8);
    if (posL!=l1.end())
        cout<<" 8 can be positioned " <<distance (l1.begin(), posL) <<" ." <<endl;
    if (posU!=l1.end())
        cout<<" 8 can be positioned " <<distance (l1.begin(), posU) <<" secondly." <<endl;
    cout<<" 8 can be positioned from " <<distance (l1.begin(), range_Iterator.first) <<" to " <<
distance (l1.begin(),
range_Iterator.second) <<endl;
}
```

例 4-31 的执行效果如图 4-31 所示。



```
list l1 is as below.
1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 2, 3, 4, 5, 6, 7, 8, 9.
list l1 has been sorted.
1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7, 8, 8, 9, 9.
9 can be searched in list l1.
vector s can be included in list l1.
8 can be positioned 14.
8 can be positioned 16 secondly.
8 can be positioned from 14 to 16
```

图 4-31 例 4-31 的执行效果

4.5 删除算法

1. remove() 算法

查找并删除容器（区间）中的元素是最基本的容器操作之一。STL 提供了 `remove()` 算法，用以直接进行元素删除，其原型为：

```
ForwardIterator remove ( ForwardIterator _First, ForwardIterator _Last, const Type& _Val);
```

`remove()` 算法的功能是实现在迭代器 `[_First, _Last]` 指定范围内删除所有值为 `_Val` 的元素。

注意：其返回值是删除元素之后容器（或区间）的新末尾位置，此返回值是非常重要的。

2. remove_if() 算法

STL 还提供了条件删除算法 `remove_if()`，用于移除容器（区间）中每个“使其条件判断式为 `true`”的元素。

被删除元素之后的元素会顺序向前移动。由于删除算法会导致元素变动，因此不能应用于关联式容器，只能应用于顺序式容器。对于那些关联式容器，需要使用 `erase()` 算法来实现删除容器中元素的目的。对于 `list` 型容器，在其类模板中提供了一个成员函数 `remove()`，具有更加性能。`remove_if()` 算法的原型为：

```
ForwardIterator remove_if (ForwardIterator First, ForwardIterator Last, Predicate Pred)
```

3. remove_copy() 和 remove_copy_if() 算法

`remove_copy()` 和 `remove_copy_if()` 算法用于在复制过程中移除相关的元素。如果使用条件删除语句，只有当一元判断式为“`true`”时，才正确执行 `remove_if()` 算法。其原型为：

```
OutputIterator remove_copy ( ForwardIterator First, ForwardIterator Last, OutputIterator Result, const T& Value)
OutputIterator remove_copy_if (InputIterator _First, InputIterator _Last, OutputIterator _Result, Predicate _Pred);
```

其中，`remove_copy_if()` 算法是 `copy()` 和 `remove_if()` 的组合。该算法将源区间 `[_First, _Last]` 内的元素复制到以 `_Result` 为起始的目标区间中。在复制过程中，删除所有使一元判断式 `_Pred` 为“`true`”的元素。在使用这两种算法时，如果目标容器（区间）的容量不足够大，

必须使用插入型迭代器。其返回值为目标容器（区间）的最后一个元素的位置。

4. 移除重复元素

在vector、list、deque、multiset和multimap等类型的容器中，允许元素重复出现，但是有时需要删除重复的元素。STL提供了unique()算法，可以实现移除容器中的重复元素。

```
ForwardIterator unique(ForwardIterator _First, ForwardIterator _Last);  
ForwardIterator unique (ForwardIterator _First, ForwardIterator _Last, BinaryPredicate _Comp);
```

上述两种形式的区别在于增加了比较函数_Comp()。其返回值是修正后的容器中的最后一个元素位置。

5. 复制过程中移除重复元素

STL提供了两种形式的unique_copy()算法。其原型为：

```
OutputIterator unique_copy (InputIterator _First, InputIterator _Last, OutputIterator _Result);  
OutputIterator unique_copy (InputIterator _First, InputIterator _Last, OutputIterator _Result,  
BinaryPredicate _Comp);
```

unique_copy()算法可以实现复制容器（区间）[_First, _Last]中的元素到以_Result为起始位置的目标容器或目标区间，并移除其中的重复元素。移除重复元素的前提是源区间必须是已序的。

其返回值是最后一个被复制的元素的下一位置。同样，在使用时必须确保目标区间有足够的容量，否则需要使用插入型迭代器。当使用的判断式_Comp为true时，该元素将会被删除。此判断式并非用来将元素和其原来的前一元素比较，而是将它和未被移除的前一元素比较。

下面以例4-32来说明上述几种删除算法的使用方法。

例 4-32

```
#include <iostream>  
#include <vector>  
#include <algorithm>  
#include <functional>  
using namespace std;  
bool mylessthan(int ele1, int ele2)  
{ if (ele2 > 3 || ele1 > 3)  
    return true;  
    else  
    return false;  
}  
void main()  
{ int source[] = {1, 4, 4, 6, 1, 2, 2, 3, 1, 6, 6, 6, 5, 7, 5, 4, 4};  
  int num = sizeof(source) / sizeof(int);  
  vector<int> v1, v2(num, 0);
```

```
vector<int> l2(num,0), l3(num,0);
vector<int>::iterator it1;
l1.assign(source, source + num);
cout << "The Original vector l1 is as below. " << endl;
copy(l1.begin(), l1.end(), ostream_iterator<int> (cout, ", "));
cout << endl;
it1 = remove (l1.begin(), l1.end(), 1);
l1.erase (it1, l1.end());
cout << " The Original vector l1 is as below after removing 1. " << endl;
copy (l1.begin(), l1.end(), ostream_iterator<int> (cout, ", "));
cout << endl;
it1 = remove_if (l1.begin(), l1.end(), bind2nd (greater<int> (), 6));
//条件表达式 bind2nd (greater<int> (), 6)

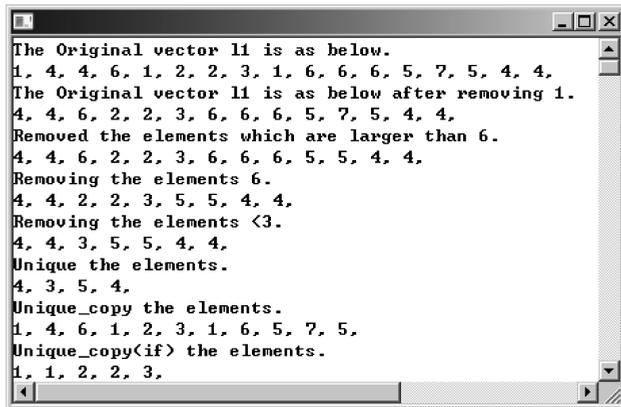
l1.erase (it1, l1.end());
cout << " Removed the elements which are larger than 6. " << endl;
copy (l1.begin(), l1.end(), ostream_iterator<int> (cout, ", "));
cout << endl;
it1 = remove_copy (l1.begin(), l1.end(), l2.begin(), 6); //返回目标容器的终止位置
cout << " Removing the elements 6. " << endl;
copy (l2.begin(), it1, ostream_iterator<int> (cout, ", "));
cout << endl;
it1 = remove_copy_if (l2.begin(), l2.end(), l3.begin(), bind2nd (less<int> (), 3));
//返回目标容器的终止位置

cout << " Removing the elements <3. " << endl;
copy (l3.begin(), it1, ostream_iterator<int> (cout, ", "));
cout << endl;
cout << " Unique the elements. " << endl;
it1 = unique (l3.begin(), l3.end());
copy (l3.begin(), -- it1, ostream_iterator<int> (cout, ", "));
cout << endl;
cout << " Unique_copy the elements. " << endl;
l1.assign (source, source + num);
it1 = unique_copy (l1.begin(), l1.end(), l11.begin());
copy (l11.begin(), -- it1, ostream_iterator<int> (cout, ", "));
cout << endl;
cout << " Unique_copy (if) the elements. " << endl;
l1.assign (source, source + num);
it1 = unique_copy (l1.begin(), l1.end(), l11.begin(), mylessthan);
copy (l11.begin(), -- it1, ostream_iterator<int> (cout, ", "));
cout << endl;
}
```

例 4-32 的执行效果如图 4-32 所示。



提示 本小节讲述了 5 种和删除操作有关的算法。这 5 种算法各有特色，值得读者认真研究。



```
The Original vector l1 is as below.
1, 4, 4, 6, 1, 2, 2, 3, 1, 6, 6, 6, 6, 5, 7, 5, 4, 4.
The Original vector l1 is as below after removing 1.
4, 4, 6, 2, 2, 3, 6, 6, 6, 6, 5, 7, 5, 4, 4.
Removed the elements which are larger than 6.
4, 4, 6, 2, 2, 3, 6, 6, 6, 5, 5, 4, 4.
Removing the elements 6.
4, 4, 2, 2, 3, 5, 5, 4, 4.
Removing the elements <3.
4, 4, 3, 5, 5, 4, 4.
Unique the elements.
4, 3, 5, 4.
Unique_copy the elements.
1, 4, 6, 1, 2, 3, 1, 6, 5, 7, 5.
Unique_copy(if) the elements.
1, 1, 2, 2, 3.
```

图 4-32 例 4-32 的执行效果

4.6 小结

本章重点介绍了C++ STL 中的各种算法。STL 算法库主要包括非修改性算法、修改性算法、排序及相关操作算法、删除算法等。通过本章的学习，读者应该较全面地掌握 STL 的各种算法，尤其是较常用的算法，例如 `for_each()`、`copy()`、`max()`、`min()`、`compare()`、`swap()`、`transform()`、`reverse()` 以及各种排序算法、排列算法、`remove()` 等。

第 5 章

迭代器——访问容器的接口

迭代器是容器和算法的纽带。

迭代器以对象序列作为它所支持的数据访问模型，将具有数组和字节形式的低级数据模型映射到高级的对象模型。

迭代器提供了一个数据访问的标准模型，缓解了要求容器提供一组更广泛的访问操作的压力。迭代器为数据提供了一种抽象的观点，使写算法的人不必顾忌多种多样的数据结构和具体细节。

5.1 迭代器及其特性

迭代器本身是一种抽象概念。即使不是迭代器，若其行为类似迭代器的东西，也可认为是迭代器。迭代器不是通用指针，而是指向数组的指针的概念抽象。通俗来讲，迭代器是一个指示器。迭代器技术能够使程序反复地对 STL 容器内容进行访问。当参数化类型为 C++ 内部类型时，迭代器是 C++ 的指针。STL 定义了 5 种类型的迭代器，根据使用方式不同而分别命名。每种容器类型支持某种类型的迭代器。通常，迭代器分为输入型迭代器、输出型迭代器、前向型迭代器、双向型迭代器和随机访问型迭代器。

输入型迭代器主要用于为程序中需要的数据源提供输入接口，此处的数据源可以是容器、数据流等。输入迭代器只能从一个序列中读取数据。迭代器可以被修改、引用并进行比较。

输出型迭代器主要用于输出程序中已经得到的数据结果，此处的数据结果指的是容器、数据流等。输出迭代器只能向一个序列写入数据，此类迭代器可以被修改和引用。

前向型迭代器可以随意访问序列中的元素，许多 STL 算法需要提供前向迭代器。前向迭代器可以用来读也可以用来写，结合了输入型迭代器和输出型迭代器的功能，并能够保存迭代器的值，以便从其原先位置开始重新遍历。

双向型迭代器既可以用来读也可以用来写，可以被增值和减值，还可以同时进行前向和后向元素的操作。所有 STL 容器都提供了双向型迭代器功能，以便于数据的写入和读出。

随机访问型迭代器可以通过跳跃的方式访问容器中的任意数据，从而使数据的访问非常灵活。随机访问型迭代器作为功能最强大的迭代器类型，具有双向迭代器的所有功能，能够使用算法和所有的迭代器比较功能。

通俗来讲，迭代器就是指向序列元素的指针，其关键的属性如下。

- 1) 当前被指向的元素，用 “*” 或 “->” 表示。
- 2) 指向下一个元素，迭代器的增量使用运算符 “++”。

- 3) 相等, 使用运算符 “==”。
- 4) 只有随机访问型迭代器可以通过加减整数, 取得相对地址。
- 5) 除了输出型迭代器之外, 其他类型的迭代器均可获得任意两个迭代器之间的位置 (使用 `distance()` 函数)。

5.2 头文件 < iterator >

所有容器都定义了其各自的迭代器型别。使用某种容器的迭代器并不需要包含专门的头文件。但由于逆向迭代器 (Reverse Iterator) 被定义于头文件 < iterator > 中, 因此在使用逆向型迭代器时, 需要包含头文件 < iterator >。

5.3 迭代器类型详述

前面已讲过, 迭代器是一种“能够遍历某个容器 (或序列) 内的所有元素”的对象。迭代器通常是利用与一般指针一致的接口来完成自己的工作。迭代器奉行纯抽象概念, 即“任何东西, 只要其行为类似迭代器, 那么它就是一种迭代器”。不同的迭代器具有不同的“能力” (行进和存取能力)。而某些算法需要特定的迭代器能力。在迭代器中, 能力是很重要的概念。

各种迭代器的关系如图 5-1 所示:

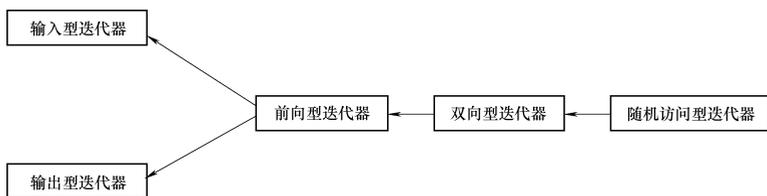


图 5-1 各种迭代器的关系

一方面, 不同的算法需要以不同的迭代器作为参数; 另一方面, 同一种算法对于不同的迭代器可能有不同效率的实现。基于迭代器类别, C++ STL 提供了 5 个类, 分别代表 5 个迭代器类别——输入型迭代器、输出型迭代器、前向型迭代器、双向型迭代器和随机访问型迭代器。

5.3.1 输入型迭代器

输入型迭代器 (Input Iterator) 用于向前读取; 输出型迭代器用于向前写入; 前向型迭代器用于读取和写入; 双向型迭代器可用于向前和向后读取及写入; 随机访问型迭代器可用于读取也可用于写入。

输入型迭代器: 只能一次一个地向前读取元素, 并按此顺序传回元素值。如果复制输入型迭代器, 原输入型迭代器和新产生的副本都向前读取, 会遍历到不同的值。所有迭代器都具备输入型迭代器的能力, 纯粹输入型迭代器的典型例子是“从标准输入装置读取数据”的迭代器。同一个值不会被读取两次, 一旦从输入流读入一个字后, 下次读取时就会传回另

一个字。

若两个输入型迭代器占用同一个位置，则两者相同。输入型迭代器的操作符如下。

- 1) *。该操作符用于从迭代器中读取元素的实际值。
- 2) ->。该操作符读取实际元素的成员。
- 3) ++。该操作符无论是放在迭代器的前边还是后边，均代表向前步进。
- 4) ==。该操作符判断两个迭代器是否相等。
- 5) !=。该操作符判断两个迭代器是否不相等。
- 6) TYPE (iter)。该操作符用于复制迭代器。

使用递增迭代器 (“++”) 时，程序员应尽量使用前置式递增操作符 (“++ iter”)，而不是后置式递增操作符 (“iter ++”)，因为前置式递增运算符性能更好。

5.3.2 输出型迭代器

输出型迭代器 (Output Iterator) 和输入型迭代器相反，其作用是将元素值逐个写入，即只能逐个对元素进行赋值，不能对同一序列进行两次遍历。输出型迭代器可以实现 “*” “++” 和复制操作。一般迭代器可以读取和写入元素值，几乎所有迭代器都具有输出型迭代器的功能。

输出型迭代器的示例如下：

- 1) “将元素写至标准输出装置” 的迭代器。如果采用两个输出型迭代器将元素写至屏幕，第二个字将跟在第一个字后面，而不是覆盖第一个字。
- 2) 另一个典型例子是插入器 (Inserter)。所谓插入器是用来将元素值插入容器内的一种迭代器。

5.3.3 前向型迭代器

前向型迭代器 (Forward Iterator) 是输入型迭代器和输出型迭代器结合的产物，具有输入型迭代器的全部功能和输出型迭代器的大部分功能。前向型迭代器的各项操作见表 5-1。

表 5-1 前向型迭代器的各项操作

表 达 式	效 果
* Iter	存取实际元素
Iter -> number	存取实际元素成员
++ Iter	向前步进
Iter ++	向前步进
Iter1 == iter2	判断两个迭代器相等
Iter1 != iter2	判断两个迭代器不相等
TYPE ()	产生迭代器 (默认构造函数)
TYPE (iter)	复制迭代器
Iter1 = iter2	赋值

前向型迭代器能多次指向同一集合的同一元素，并能多次处理同一元素。

- 在使用输出型迭代器时，无需检查是否抵达序列尾端，可直接写入数据。输出型迭代器不提供比较操作，不能将它和尾端迭代器相比较。
- 在使用前向型迭代器之前，需要确定迭代器是否有效。因此，无论使用哪种迭代器，在循环时尽量使用 `begin()` 和 `end()` 作为循环的起止。

5.3.4 双向型迭代器

双向型迭代器 (Bidirectional Iterator) 在前向型迭代器的基础上增加了回头遍历的功能，既可以支持递减运算符，也可以实现步退操作。双向型迭代器的递减操作见表 5-2。

表 5-2 双向型迭代器的递减操作

表 达 式	效 果
<code>-- iter</code>	步退 (传回新位置)
<code>Iter --</code>	步退 (传回老位置)

5.3.5 随机访问型迭代器

随机访问型迭代器在双向型迭代器的基础上增加了随机访问功能。因此，必须增加 (提供) 迭代器算术运算，即可以实现迭代器加减某个偏移量功能，能处理距离问题，并可运用诸如 “<” 和 “>” 等操作符实现比较操作。

可以支持随机访问型迭代器的容器对象或数据类型包括 `vector`、`deque`、`strings` (`string`、`wstring`) 以及普通数组 `array`。

随机访问型迭代器的各项操作见表 5-3。

表 5-3 随机访问型迭代器的各项操作

表 达 式	效 果
<code>Iter [n]</code>	存取索引位置为 n 的元素
<code>Iter += n</code>	前进 n 个元素
<code>Iter -= n</code>	后退 n 个元素
<code>Iter + n</code>	传回 iter 之后的第 n 个元素
<code>n + iter</code>	传回 iter 之后的第 n 个元素
<code>Iter - n</code>	传回 iter 之前的第 n 个元素
<code>Iter1 - iter2</code>	传回 iter1 和 iter2 的距离
<code>Iter1 < iter2</code>	判断 iter1 是否在 iter2 之前
<code>Iter1 > iter2</code>	判断 iter1 是否在 iter2 之后
<code>Iter1 < = iter2</code>	判断 iter1 是否在 iter2 之后
<code>Iter1 > = iter2</code>	判断 iter1 是否不在 iter2 之前

随机访问型迭代器提供了上述丰富的运算功能，同时也带来诸多麻烦。例如：

```

vect. end() - 1; //迭代器指针可能指向 begin()之前
pos + = 2; //迭代器指针可能超越 end()位置
for(pos = vect. begin(); pos! = vect. end(); pos + = 2) //迭代器可能超越容器末尾
{
...
}

```



提示

随机访问型迭代器对 list、sets 和 maps 是无效的。

5.3.6 vector 迭代器的递增和递减

迭代器的递增问题较复杂，程序员可以递增或递减暂时性迭代器，对于 vector 和 string 两种容器类型则不同。（下面代码摘自《C++ 标准程序库》）

```
std::vector<int> v;  
if(v.size() > 1)  
{  
    sort(++v.begin(), v.end())  
}
```

上述代码在编译时，通常会出错。但如果将容器类型换作 deque，就可以通过编译。对于 vector 的其他操作，有时也可以通过编译。

上述现象的原因一般为：vector 迭代器通常被作为一般指针。C++ 不允许修改任何基本型别的暂时值，但对 struct 和 class 则可以。若迭代器被实例化为一般指针，则编译会失败；若迭代器被实例化为 class，则编译可以成功。对于 deque、lists、sets 和 maps 编译总是能通过，因为这些容器对象的迭代器不可能被实例化为一般指针。

5.4 迭代器配接器

迭代器配接器可以使算法能够以逆向模式和插入模式进行工作，还可以和流搭配。本节将讲述使用迭代器进行逆向遍历、插入元素、实现流操作等内容。

5.4.1 逆向型迭代器

逆向型迭代器（Reverse Iterator）是一种配接器，用于重新定义递增运算和递减运算，使其行为正好倒置。程序执行时，算法以逆序次序来处理元素。所有标准容器都允许使用逆向型迭代器来遍历元素。下面举例说明。

例 5-1

```
#include <iostream>  
#include <list>  
#include <algorithm>  
using namespace std;  
void print(intele)  
{  
    cout << ele << ", ";  
}  
void main()  
{  
    list<int> l1;
```

```

for(int i=1;i<=9;i++)
    l1.push_back(i);
for_each(l1.begin(),l1.end(),print);
cout<<endl;
for_each(l1.rbegin(),l1.rend(),print);
cout<<endl;
}

```

例 5-1 的执行效果如图 5-2 所示。

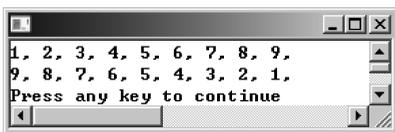


图 5-2 例 5-1 的执行效果

在例 5-1 中,容器的成员函数 `rbegin()` 和 `rend()` 会分别传回一个逆向迭代器,就像 `begin()` 和 `end()` 的返回值一样,共同定义出一个半开区间。其中 `rbegin()` 函数传回逆向遍历第一元素的位置,即实际最后一个元素的位置; `rend()` 函数传回逆向遍历时最后一个元素的下一个位置,即第一个元素的前一个位置(已经不属于本容器了)。

通常的迭代器若具备双向移动的功能,则可以转化成逆向迭代器。例如,

```

vector<int>::iterator pos;
vector<int> v1;
pos=find(v1.begin(),v1.end(),5);
vector<int>::reverse_iterator rpos(pos); //转换
cout<<" *rpos" <<*rpos<<endl;

```

上述代码实现将正常迭代器转化为逆向迭代器,并输出其值。

例 5-2

```

#include <iostream>
#include <list>
#include <algorithm>
using namespace std;
void print(intele)
{
    cout<<ele<<" ";
}
void main()
{
    list<int> l1;
    for(int i=1;i<=9;i++)
        l1.push_back(i);
    cout<<" vector : " <<endl;
    for_each(l1.begin(),l1.end(),print);
}

```

```

cout << endl;
list<int>::iterator it;
it = find(ll.begin(), ll.end(), 5);
cout << "pos: " << *it << endl;
list<int>::reverse_iterator itR (it);
cout << " rpos: " << *itR << endl;
}

```

例 5-2 的执行效果如图 5-3 所示。

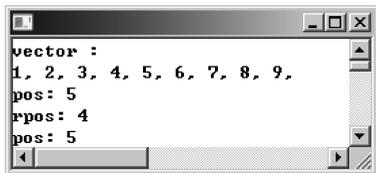


图 5-3 例 5-2 的执行效果

逆向型迭代器类模板提供了一个 `base()` 成员函数。`base()` 成员函数可以实现将逆向迭代器类型的迭代器转化为正常迭代器。

5.4.2 插入型迭代器

插入型迭代器 (Insert Iterator) 又称为插入器 (Inserter)。用来将“赋值新值”操作转换为“插入新值”操作。通过插入型迭代器，算法可以执行插入行为而非覆盖行为。所有插入型迭代器都隶属于输出型迭代器类型，只提供“赋予新值”的功能。

插入型迭代器可以把上述的赋值操作转化为插入操作。实际上这里面有两个操作：运算符“*”传回迭代器的当前位置，然后由“operator =”赋值新值。插入型迭代器通常使用以下两个技巧：

- 1) “operator*”被作为一个无实际动作的动作，简单传回 *this。对插入型迭代器来说，*pos 与 pos 等价。
- 2) 赋值操作被转化为插入操作。事实上插入型迭代器会调用容器的 `push_back()`、`push_front()` 或 `insert()` 成员函数。

对于一个插入型迭代器，插入新值可以采用两种形式：`pos = value` 和 `*pos = value`。但正确的表达式应该是 `*pos = value`。

插入型迭代器分 3 种类型：后插入型迭代器 (BackInserter)、前插入型迭代器 (FrontInserter) 和产生型迭代器 (Generallnserter)。它们之间的区别在于插入位置的不同。在迭代器初始化时，一定要清楚自己所属的容器是哪一种。每一种插入迭代器均由一个对应的便捷函数对其加以生成和初始化 (见表 5-4)。

表 5-4 插入型迭代器的分类

名 称	类 型	其所调用的函数	生成函数
后插入型迭代器	back_insert_iterator	push_back	back_inserter (cont)
前插入型迭代器	front_insert_iterator	push_front	front_inserter (cont)
产生型迭代器	insert_iterator	Insert (pos, value)	Inserter (cont, pos)

容器本身必须支持插入型迭代器所调用的函数, 否则该插入型迭代器不可用。例如, 后插入型迭代器只能用在 `vector`、`deque`、`list`、`string` 等类型的容器上, 前插入型迭代器只能用在 `deque` 和 `list` 型容器上。原因很简单, 相应型别的容器必须具备相应型别的迭代器所对应的可调用成员函数。

后插入型迭代器生成时必须根据其所属容器进行初始化。`back_inserter()` 函数为此提供了捷径。前面章节已经大量使用了 `back_inserter()` 函数。`back_inserter()` 在插入元素时, 会造成指向该 `vector` 的其他迭代器失效。`string` 也提供了 STL 容器接口, 包括成员函数 `push_back()`。同时可以使用 `back_inserter()` 函数为 `string` 型容器追加字符串。

前插入型迭代器生成时必须根据其所属容器进行初始化。`front_inserter()` 函数为此提供了捷径。前插入型迭代器透过成员函数 `push_front()` 将一个元素值加在容器头部。而 `push_back()` 仅在 `deque` 和 `list` 中有所实现。而 C++ STL 中就只有这两个类型的容器支持前插入型迭代器。值得一提的是, 在插入多个元素时, 前插入型迭代器以逆序方式插入, 因为它总是将后一个元素插于前一个元素的前面。

产生型迭代器根据两个参数初始化: ①容器; ②待插入位置。迭代器内部以“待插入位置”为参数, 调用成员函数 `insert()`。`inserter()` 函数则可以提供更方便的手段产生产生型迭代器, 并将其初始化。本类迭代器对所有标准容器均适用, 因为所有容器都有 `insert()` 成员函数。然而对关联式容器而言, 插入位置仅是标识, 元素的真正位置要根据其实值或键值而定。插入操作完成后, 产生型迭代器获得被插入元素的位置。在 `deque`、`vector` 和 `string` 型容器中, 为确保该迭代器的位置始终有效, 该产生型迭代器本身会失效。其实, 每一次插入操作都会使指向容器的所有迭代器失效。

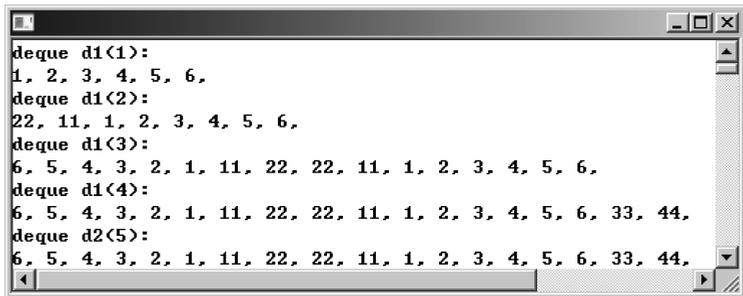
关联式容器的定制型插入型迭代器: 对于关联式容器, 产生型迭代器的“位置参数”仅仅是个提示, 用于加快速度。如果产生型迭代器使用不当, 反而可能导致性能下降。如果逆序插入, 会较缓慢, 甚至导致插入错误。这也是 STL 的缺陷之一。

例 5-3

```
#include <iostream>
#include <deque>
#include <algorithm>
using namespace std;
void main()
{
    int dim[] = {1,2,3,4,5,6};
    deque<int> dl;
    cout << "deque dl (1): " << endl;
    copy(dim, dim + 6, back_inserter (dl));
    copy (dl.begin(), dl.end(), ostream_iterator<int> (cout, " "));
    cout << endl;
    cout << " deque dl (2): " << endl;
        front_inserter (dl) = 11;
    front_inserter (dl) = 22;
    copy (dl.begin(), dl.end(), ostream_iterator<int> (cout, " "));
```

```
cout << endl;
cout << "deque d1 (3): " << endl;
copy(d1.begin(), d1.end(), front_inserter(d1));
copy(d1.begin(), d1.end(), ostream_iterator<int>(cout, " "));
cout << endl;
inserter(d1, d1.end()) = 33;
inserter(d1, d1.end()) = 44;
cout << " deque d1 (4): " << endl;
copy(d1.begin(), d1.end(), ostream_iterator<int>(cout, " "));
cout << endl;
deque<int> d2;
copy(d1.begin(), d1.end(), inserter(d2, d2.begin()));
cout << " deque d2 (5): " << endl;
copy(d2.begin(), d2.end(), ostream_iterator<int>(cout, " "));
cout << endl;
}
```

例 5-3 的执行效果如图 5-4 所示。



```
deque d1(1):
1. 2. 3. 4. 5. 6.
deque d1(2):
22. 11. 1. 2. 3. 4. 5. 6.
deque d1(3):
6. 5. 4. 3. 2. 1. 11. 22. 22. 11. 1. 2. 3. 4. 5. 6.
deque d1(4):
6. 5. 4. 3. 2. 1. 11. 22. 22. 11. 1. 2. 3. 4. 5. 6. 33. 44.
deque d2(5):
6. 5. 4. 3. 2. 1. 11. 22. 22. 11. 1. 2. 3. 4. 5. 6. 33. 44.
```

图 5-4 例 5-3 的执行效果

5.4.3 流型迭代器

流型迭代器（Stream Iterator）是一种迭代器适配器。程序员可以把流型迭代器当成算法的原点和终点。流型迭代器是特殊用途的输入型和输出型迭代器，程序通过流型迭代器能管理与 I/O 流相关的数据。插入型迭代器和逆向型迭代器均由迭代器适配器形成。一个输入流型（Istream）迭代器可用来从输入流中读取元素，而输出流型（Ostream）迭代器可以用来对输出流写入元素。流型迭代器的特殊形式是所谓的流缓冲区迭代器，用来对流缓冲区进行直接读取和写入操作。

（1）输出流型迭代器

输出流型迭代器可以将被赋予的值写入输出流中。其各项操作见表 5-5。输出流型迭代器将赋值操作转化为运算符 `operator <<`。算法就可以使用一般的迭代器接口直接对流执行写动作。

表 5-5 输出流型迭代器的各项操作

算 式	效 果
<code>ostream_iterator <T> (ostream)</code>	为输出流产生一个输出流型迭代器
<code>ostream_iterator <T> (ostream, delim)</code>	为输出流产生一个输出流型迭代器, 各元素间以 <code>delim</code> 为分隔符
<code>* iter</code>	无实际操作
<code>Iter = value</code>	将 <code>value</code> 写到 <code>ostream</code> , 如 <code>ostream << value</code>
<code>++ iter</code>	无实际操作
<code>Iter ++</code>	无实际操作

注: `delim` 为分隔符。

生成迭代器时, 必须提供一个输出流作为参数, 迭代器将会把元素写至该输出流身上。另一个参数可有可无, 是个字符串, 作为每个元素值之间的分隔符。分隔符的型别是 `const char *`。如果使用 `string` 的对象 (变量), 在使用时一定要使用成员函数 `c_str()` 以获得该对象的内容。例如,

```
string delim;
ostream_iterator<int> cout, delim c_str());
```

流型迭代器的模板为:

```
template < classType , class CharType = char, class Traits = char_traits<CharType> >>
```

其中第一个参数代表被插入至流的数据类型; 第二个参数的默认值是 `char`; 第三个参数是可选的, 其默认值是 `char_traits<CharType>`。

输出流型迭代器必须满足输出型迭代器的所有要求, 在算法中使用输出流型迭代器时, 可以直接访问输出流。

例 5-4

```
#include <iostream>
#include <iterator>
#include <algorithm>
#include <vector>
using namespace std;
void main()
{
    int dim[] = {1,2,3,4,5,6,7,8,9};
    vector<int> v1;
    ostream_iterator<int> iter (cout, " \n");
    v1.assign (dim, dim+9);
    copy (v1.begin(), v1.end(), ostream_iterator<int> (cout));
    cout << endl;
    copy (v1.begin(), v1.end(), ostream_iterator<int> (cout, " < "));
    cout << endl;
    string delim (" ");
    copy (v1.begin(), v1.end(), ostream_iterator<int> (cout, delim c_str()));
    cout << endl;
    *iter = 22;
    iter ++;
```

```
*iter = 33;
iter ++;
*iter = 44;
}
```

例 5-4 的执行效果如图 5-5 所示。

(2) 输入流型迭代器

输入流型迭代器是输出流型迭代器的“伙伴”，用于从输入流中读取元素。通过输入流型迭代器，算法可以从流中直接读取元素。然而，输入流型迭代器较输出流型迭代器稍微复杂一些。

产生输入流型迭代器时，必须提供一个输入流作为参数，迭代器将从中读取数据（一般经由输入型迭代器的通用接口，利用 `operator >>` 读取元素）。读取动作可能会失败，此外算法需要知道区间是否到达终点。为了解决这些问题，可使用 `end-of-stream` 迭代器，可利用输入流型迭代器的默认构造函数生成。只要读取一次失败，所有输入流型迭代器都会变成 `end-of-stream` 迭代器。所以，每次读取结束后，应该将输入流型迭代器和 `end-of-stream` 迭代器比较，观察迭代器是否合理合法。

输入流型迭代器的构造函数会将流打开，并读取第一个元素。在确实需要用到一个输入流型迭代器之前，不要过早定义它。上述做法是必要的，否则一旦 `operator *` 被调用，将无法传回第一个元素。这和软件版本有关系，某些版本会延缓第一次读取操作，直到第一次 `operator *` 被调用。输入流型迭代器的各项操作见表 5-6。

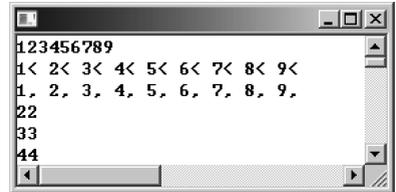


图 5-5 例 5-4 的执行效果

表 5-6 输入流型迭代器的各项操作

算 式	效 果
<code>Istream_iterator <T> ()</code>	产生一个 <code>end-of-stream</code> 迭代器
<code>Istream_iterator <T> (istream)</code>	为输入流型迭代器产生一个迭代器（可能立刻读取一个元素）
<code>* iter</code>	传回先前读取的值（若构造函数未读取第一个元素值，则本式执行读取任务）
<code>Iter -> member</code>	传回先前读取的元素成员
<code>++ iter</code>	读取下一个元素，并传回其位置
<code>Iter ++</code>	读取下一个元素，传回迭代器指向前一个元素
<code>Iter1 == iter2</code>	检验 <code>iter1</code> 和 <code>iter2</code> 是否相等
<code>Iter1 != iter2</code>	检查 <code>iter1</code> 和 <code>iter2</code> 是否不相等

输入流型迭代器类模板如下：

```
template < class T, class charT = char, class traits = char_traits < charT >, class Distance =
ptrdiff_t > class istream_iterator
```

其中第一个参数是数据类型；第二个参数和第三个参数确定流的型别，第四个参数用来指定迭代器距离的表示型别。

两个输入流型迭代器相等的条件如下：

- 1) 两者都是 `end-of-stream` 迭代器。
- 2) 两者均可进行读取操作，并指向相同的流。

例 5-5

```
#include <iostream>
#include <vector>
#include <iterator>
#include <algorithm>
using namespace std;
void main()
{
    vector <int> v1;
    istream_iterator<int> inputCin (cin);
    istream_iterator<int> IEOF;
    while (inputCin != IEOF)
    {
        *inputCin;
        v1.push_back (*inputCin);
        ++inputCin;
    }
    copy (v1.begin(), v1.end(), ostream_iterator<int> (cout, ", "));
    cout << endl;
}
```

例 5-5 的执行效果如图 5-6 所示。



图 5-6 例 5-5 的执行效果

在例 5-5 中，所定义的流型迭代器的输入数据类型为 `int`，一旦输入非 `int` 型数据时，输入即面临结束，`while` 循环退出。

5.5 迭代器辅助函数

C++ STL 为迭代器提供了 3 个辅助函数：`advance()`、`distance()` 和 `iter_swap()`。前两个函数为所有迭代器提供前进和后退的功能；第三个函数允许程序员交换两个迭代器的数值。前面章节已经讲过，只有随机访问型迭代器才可以自由地前进和后退，方便地使用偏移量。`advance()` 函数使所有类型的迭代器均可以前进和后退；`distance()` 函数可以计算同一容器中两个迭代器值之间的距离；`iter_swap()` 函数可以方便地交换两个迭代器所指向元素的值。

5.5.1 前进 `advance()` 函数

迭代器辅助 `advance()` 函数可使相关的迭代器前进和后退一个或多个元素，增加的速度由参数决定。其原型为：

```
template < class InputIterator, class Distance > void advance ( InputIterator& _InIt, Distance  
Off);
```

`advance()` 函数的功能：实现将输入型迭代器 `_Init` 前进或后退 `_Off` 个元素。对于参数 `_Off`，前进为正数，后退为负数。只有当迭代器类型为双向型迭代器和随机访问型迭代器时，参数 `_Off` 为负值，表示后退。原型中的 `Distance` 是模板型别，通常为整数型别。`advance()` 函数在移动迭代器指针时，不会检查是否超过序列的 `end()`。因此可能会导致迭代器已到达尾部（首部），而还会继续增加迭代器的数值。

`advance()` 函数也有其局限性：

1) 只有随机访问型迭代器才能方便地移动迭代器的位置；非随机访问型迭代器需要使用 `advance()` 函数移动迭代器指向的位置，其执行性能并不佳。

2) `advance()` 函数没有返回值。

但这些并不会掩盖 `advance()` 函数的优点，正是 `advance()` 使迭代器的移动变得容易了。

例 5-6

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
void main()
{
    int dim[] = {1,2,3,4,5,6,7,8,9};
    vector<int> v1;
    vector<int>::iterator it;
    v1.assign(dim,dim+9);
    copy(v1.begin(),v1.end(),ostream_iterator<int>(cout," "));
    cout<<endl;
    it=v1.begin();
    advance(it,5);
    cout<<" The sixth element: " <<*it<<endl;
    advance(it,-3);
    cout<<" The third element: " <<*it<<endl;
}
```

例 5-6 的执行效果如图 5-7 所示。

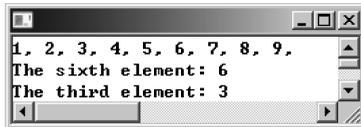


图 5-7 例 5-6 的执行效果

5.5.2 距离 `distance()` 函数

`distance()` 函数用来处理两个迭代器之间的距离。使用该函数时，需要包含头文件 `<iterator>`。其原型为：

```
ptrdiff_t distance (InIt first, InIt last);
```

`distance()` 函数的返回值 `ptrdiff_t` 是整型, `first` 和 `last` 分别代表两个迭代器, 这两个迭代器必须是同一容器的迭代器。如果不是随机访问型迭代器, 从 `first` 开始往前走必须能到达 `last`, 即 `last` 必须和 `first` 相同或在其后。

`distance()` 函数能够根据两个迭代器传回它们之间的距离。对于随机访问型迭代器, 距离是常数; 对于其他型迭代器, 距离具有线性复杂度; 对于非随机访问型迭代器, 使用 `distance()` 函数时, 其性能和效果并不是很好, 程序员应尽力避免使用。

例 5-7

```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;
void main()
{
    int dim[] = {1,2,3,4,5,6,7,8,9};
    list<int> l1;
    copy(dim, dim+9, back_inserter (l1));
    copy (l1.begin(), l1.end(), ostream_iterator<int> (cout, " "));
    cout << endl;
    list<int>:: iterator pos;
    pos = find (l1.begin(), l1.end(), 5);
    if (pos != l1.end())
    {
        cout << " The distance between beginning and 5: " << distance (l1.begin(), pos) << endl;
    }
    else
    {
        cout << " 5 not found. " << endl;
    }
}
```

例 5-7 的执行效果如图 5-8 所示。



图 5-8 例 5-7 的执行效果

5.5.3 交换两个迭代器所指内容 `iter_swap()` 函数

`iter_swap()` 函数用来交换两个迭代器所指向的元素值。其原型为:

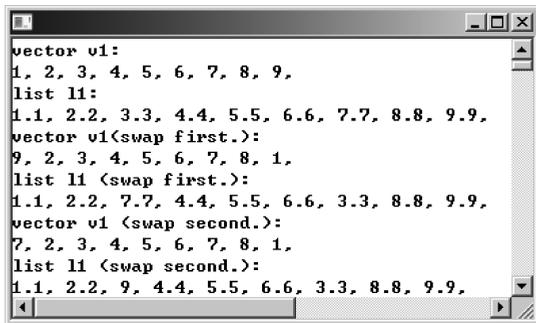
```
void iter_swap ( ForwardIterator1 First, ForwardIterator2 Second)
```

上述代码中的 `iter_swap()` 函数用以交换迭代器 `First` 和 `Second` 所指向的元素值。参与交换的两个迭代器的型别不一定相同, 但所指向的两个元素必须可以互相赋值。

例 5-8

```
#include <iostream>
#include <vector>
#include <list>
#include <algorithm>
#include <iterator>
using namespace std;
void main()
{
    int dim[] = {1,2,3,4,5,6,7,8,9};
    double Vdim[] = {1.1,2.2,3.3,4.4,5.5,6.6,7.7,8.8,9.9};
    vector<int> v1;
    list<double> l1;
    v1.assign(dim,dim+9);
    copy(Vdim,Vdim+9,back_inserter (l1));
    cout << " vector v1: " <<endl;
    copy (v1.begin(), v1.end(), ostream_iterator<int> (cout," "));
    cout <<endl;
    cout << " list l1: " <<endl;
    copy (l1.begin(), l1.end(), ostream_iterator<double> (cout," "));
    cout <<endl;
    vector<int>::iterator itV;
    itV=v1.end();
    itV--;
    iter_swap (v1.begin(), itV);
    cout << " vector v1 (swap first.): " <<endl;
    copy (v1.begin(), v1.end(), ostream_iterator<int> (cout," "));
    cout <<endl;
    list<double>::iterator itLs, itLe;
    itLs=l1.begin();
    advance (itLs, 2);
    itLe=l1.end();
    advance (itLe, -3);
    iter_swap (itLs, itLe);
    cout << " list l1 (swap first.): " <<endl;
    copy (l1.begin(), l1.end(), ostream_iterator<double> (cout," "));
    cout <<endl;
    itV=v1.begin();
    iter_swap (itLs, itV);
    cout << " vector v1 (swap second.): " <<endl;
    copy (v1.begin(), v1.end(), ostream_iterator<int> (cout," "));
    cout <<endl;
    cout << " list l1 (swap second.): " <<endl;
    copy (l1.begin(), l1.end(), ostream_iterator<double> (cout," "));
    cout <<endl;
}
```

例 5-8 的执行效果如图 5-9 所示。



```
vector v1:  
1, 2, 3, 4, 5, 6, 7, 8, 9,  
list l1:  
1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9,  
vector v1<swap first.>:  
9, 2, 3, 4, 5, 6, 7, 8, 1,  
list l1 <swap first.>:  
1.1, 2.2, 7.7, 4.4, 5.5, 6.6, 3.3, 8.8, 9.9,  
vector v1 <swap second.>:  
7, 2, 3, 4, 5, 6, 7, 8, 1,  
list l1 <swap second.>:  
1.1, 2.2, 9, 4.4, 5.5, 6.6, 3.3, 8.8, 9.9,
```

图 5-9 例 5-8 的执行效果

5.6 小结

本章重点讲述了 5 部分内容：第一部分简要地讲述了迭代器的概念及其特性；第二部分讲述了使用迭代器及相关函数需要包含的头文件；第三部分是本章的重点，详细讲述了迭代器的类型——输入型、输出型、前向型、双向型和随机访问型以及 vector 迭代器如何实现递增和递减；第四部分同样是本章的重点，讲述了迭代器配接器，主要讲述了三种类别的迭代器——插入型、逆向型和流型迭代器；第五部分讲述了迭代器辅助函数，主要包括 advance()，distance()和 iter_swap()，这 3 个函数用于实现迭代器的前进和后退、计算同一容器中两个迭代器之间的距离以及交换两个迭代器所指向的元素值。

第 6 章

数值计算类模板

本章讲述C++ STL的数值计算相关内容，其中包括复数（Complex）、数值数组（Value Arrays）以及从C标准程序库继承而来的全局数值计算算法函数。C++在数值方法上的优越性使其被广泛应用于涉及复杂数值的科学与工程计算，也使得C++语言支持各种计算的技术逐渐出现。本章围绕STL中的内容，主要讲述以下4个方面的内容：

- 复数运算
- 数组（向量）运算
- 通用数值运算
- 全局性数学函数

6.1 复数运算

C++ STL中提供了一个template class `complex < >`，用于实现复数操作。复数是由实部和虚部组成的数值。虚部的特点是“其平方值为负数”，即复数虚部带着 i ， i 是 -1 的平方根。类`complex`定义于头文件`<complex >`中。使用复数运算的相关函数时，需要包含头文件`<complex >`。

```
#include <complex >
```

类`complex`的定义：

```
template <class Type > class complex
```

其中`template`参数`Type`被用来作为复数的实部和虚部的标量型别。STL还提供了针对标量型别`float`、`double`和`long double`的特殊版本。

6.1.1 一个复数运算例题

本小节介绍一个最简单的复数运算例题。

例 6-1

```
#include <iostream >
#include <complex >
using namespace std;
void main()
{complex <double > c1(3.0,4.0); //real, image
complex <float > c2(polar(5.0,1.0)); //magnitude, phase
cout << "c1: " << c1 << endl;
cout << "c2: " << c2 << endl;
```

```

    cout << "c1: magnitude: " << abs(c1) << "(squared magnitude: " << norm(c1) << ")" << "phase angle: "
    << arg(c1) << endl;
    cout << "c2: magnitude: " << abs(c2) << "(squared magnitude: " << norm(c2) << ")" << "phase angle: "
    << arg(c2) << endl;
    cout << "c1 conjugated: " << conj(c1) << endl;
    cout << "c2 conjugated: " << conj(c2) << endl;
    cout << "c1* 2.0 = : " << (c1* 2.0) << endl;
    cout << "c1 + c2 = : " << (c1 + complex<double>(c2.real(), c2.imag())) << endl;
}

```

例 6-1 的执行效果如图 6-1 所示。

```

c1: <3,4>
c2: <2.70151,4.20736>
c1: magnitude: 5(squared magnitude: 25)phase angle: 0.927295
c2: magnitude: 5(squared magnitude: 25)phase angle: 1
c1 conjugated: <3,-4>
c2 conjugated: <2.70151,-4.20736>
c1*2.0=: <6,8>
c1+c2=: <5.70151,8.20736>

```

图 6-1 例 6-1 的执行效果

6.1.2 复数类成员函数

复数模板类 `complex` 的成员函数包括构造函数、实部函数、虚部函数和运算符函数。本节主要讲述两个构造函数、实部函数和虚部函数。

1. 构造函数

复数模板类 `complex` 提供了两个构造函数，其形式略有不同。其原型为：

```

complex(const T& re=0, const T& im=0);
complex(const complex& x);

```

上述两个构造函数的形式均较简单，第一种形式是利用参数 `re` 和 `im` 确定需要构造的复数对象；第二种形式是利用原有的复数对象构造新的复数对象。详见例 6-2 的部分代码。

```

complex<float> c1(1.5,2.5); //构造复数对象 c1
complex<float> c2(c1); //使用复数对象 c1 来构造复数对象 c2

```

2. 实部函数 (real) 和虚部函数 (imag)

复数模板类 `complex` 还提供了专门针对复数对象实部和虚部的函数。其原型为：

```

T real() const;
T imag() const;

```

C++ STL 的复数库中仅指定了上述形式的函数声明。上述函数可以仅返回（获取）复数对象的实部或虚部。在 Visual C++ 6.0 中，程序员还可以使用 `real()` 和 `imag()` 函数修改复数对象的实部数值和虚部数值。例如，

```
complex<float> c1(1.5,2.5);
complex<float> c2(c1);
float re=3.5;
float im=5.2;
c2.real(re); //修改复数对象 c2 的实部为 3.5
c2.imag(im); //修改复数对象 c2 的虚部为 5.2
```

6.1.3 复数类运算符

复数和实数一样，需要参与各种数学运算。STL 提供了“+”“-”“*”“/”“=”等复数运算符，还提供了其他一些较复杂的运算符，例如“+=”“-=”“*=”和“/=”。除此之外，STL 还提供了部分逻辑运算符，例如“==”和“!=”。

6.1.4 复数类运算

复数类运算除了包括在上一节中提到的简单数学运算，还包括部分较复杂的运算。本小节除对上一节涉及的简单数学运算进行举例说明之外，还讲述绝对值（或“模”）函数、绝对值平方、复数相位、输入与输出、共轭函数、复数极坐标形式、比较等较复杂的运算。

1. 算术运算

下面以例 6-2 对 6.1.3 节中所提到复数运算符的使用方法均进行了说明。复数运算比较复杂——既可以和复数进行运算，也可以和实数进行运算。

例 6-2

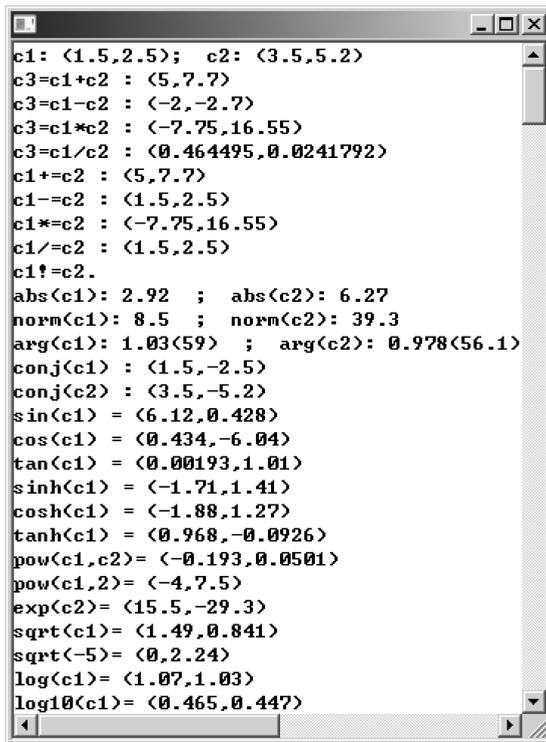
```
#include <iostream>
#include <complex>
using namespace std;
void main()
{complex<float> c1(1.5,2.5);
complex<float> c2(c1);
complex<float> c3;
float re=3.5;
float im=5.2;
c2.real(re);
c2.imag(im);
cout << "c1: " << c1 << "; c2: " << c2 << endl;
c3=c1+c2;
cout << "c3=c1+c2: " << c3 << endl;
c3=c1-c2;
cout << "c3=c1-c2: " << c3 << endl;
c3=c1*c2;
cout << "c3=c1*c2: " << c3 << endl;
c3=c1/c2;
cout << "c3=c1/c2: " << c3 << endl;
c1+=c2;
cout << "c1+=c2: " << c1 << endl;
c1-=c2;
```

```

cout << "c1 - = c2 : " << c1 << endl;
c1* = c2;
cout << "c1* = c2 : " << c1 << endl;
c1/ = c2;
cout << "c1/ = c2 : " << c1 << endl;
bool cmp = (c1 == c2);
if (cmp)
    cout << "c1 == c2. " << endl;
else
    cout << "c1! = c2. " << endl;
}

```

例 6-2 的执行效果如图 6-2 所示。



```

c1: (1.5,2.5); c2: (3.5,5.2)
c3=c1+c2 : (5,7.7)
c3=c1-c2 : (-2,-2.7)
c3=c1*c2 : (-7.75,16.55)
c3=c1/c2 : (0.464495,0.0241792)
c1+=c2 : (5,7.7)
c1-=c2 : (1.5,2.5)
c1*=c2 : (-7.75,16.55)
c1/=c2 : (1.5,2.5)
c1!=c2.
abs(c1): 2.92 ; abs(c2): 6.27
norm(c1): 8.5 ; norm(c2): 39.3
arg(c1): 1.03(59) ; arg(c2): 0.978(56.1)
conj(c1) : (1.5,-2.5)
conj(c2) : (3.5,-5.2)
sin(c1) = (6.12,0.428)
cos(c1) = (0.434,-6.04)
tan(c1) = (0.00193,1.01)
sinh(c1) = (-1.71,1.41)
cosh(c1) = (-1.88,1.27)
tanh(c1) = (0.968,-0.0926)
pow(c1,c2)= (-0.193,0.0501)
pow(c1,2)= (-4,7.5)
exp(c2)= (15.5,-29.3)
sqrt(c1)= (1.49,0.841)
sqrt(-5)= (0,2.24)
log(c1)= (1.07,1.03)
log10(c1)= (0.465,0.447)

```

图 6-2 例 6-2 的执行效果

2. 其他运算

复数还可以进行其他一些简单运算，例如绝对值、绝对值平方、复数相位、输入与输出、共轭、复数极坐标形式等。

1) 绝对值 `abs()` 函数。其原型为：

```
T abs(const complex<T>& x);
```

复数的绝对值即复数的模，绝对值的计算公式为

$$\sqrt{\text{cmp. real}()^2 + \text{cmp. imag}()^2}$$

2) 绝对值平方 `norm()` 函数。其原型为:

```
T norm(const complex<T>& x);
```

复数绝对值平方的求取需要使用 `norm()` 函数。复数绝对值平方的计算公式为

$$\text{cmplx. real}()^2 + \text{cmplx. imag}()^2$$

3) 复数相位 `arg()` 函数。其原型为:

```
T arg(const complex<T>& x);
```

`arg()` 函数返回复数的相位, 其返回值的单位是弧度, 是复数极坐标形式的相位角。相位角的计算公式为

$$\text{atan2}(\text{cmplx. imag}() + \text{cmplx. real}())$$

4) 输入与输出。复数的输入与输出, 可以使用 “>>” 和 “<<”。

5) 共轭 `conj()` 函数。其原型为:

```
complex<T> conjg(const complex<T>& x);
```

共轭函数可以归入超越函数的范围。这里在此进行介绍。

共轭 `conj()` 函数产生并返回一个复数。共轭复数是指新生成复数的虚部与原复数的虚部互为反相。

6) 复数极坐标形式 `polar()` 函数。其原型为:

```
complex<T> polar(const T& rho, const T& theta=0);
```

`polar()` 函数的返回值是复数类型, 可用于使用极坐标和相位的形式创建一个复数对象。例如在 6.1.1 节中的代码:

```
complex<float> c2(polar(5.0,1.0)); //magnitude, phase
```

下面对上述几个函数进行举例, 说明它们各自的使用方法。

```
float a1 = abs(c1);
float a2 = abs(c2);
cout.precision(3);
cout << "abs(c1): " << a1 << " "; << " abs(c2): " << a2 << endl;
a1 = norm(c1);
a2 = norm(c2);
cout.precision(3);
cout << "norm(c1): " << a1 << " "; << " norm(c2): " << a2 << endl;
a1 = arg(c1);
a2 = arg(c2);
cout.precision(3);
cout << "arg(c1): " << a1 << "(" << (a1*180/3.1415926) << ")" << " "; << " arg(c2): " << a2 <<
    " (" << (a2*180/3.1415926) << ")" << endl;
complex<float> c11;
complex<float> c12;
c11 = conj(c1);
c12 = conj(c2);
cout << "conj(c1): " << c11 << endl;
cout << "conj(c2): " << c12 << endl;
```

上述代码的执行结果如下:

```
abs(c1): 2.92 ; abs(c2): 6.27
norm(c1): 8.5 ; norm(c2): 39.3
arg(c1): 1.03(59) ; arg(c2): 0.978(56.1) //括号内数字为角度
conj(c1): (1.5, -2.5)
conj(c2): (3.5, -5.2)
```



总结 6.1 节主要讲述复数运算, 涉及复数类对象 (复数) 的各种操作、各种运算以及部分超越函数的使用。

6.1.5 复数的超越函数运算

复数相关的超越函数一般包括三角函数和指数函数。三角函数主要是指 \sin 、 \cos 、 \tan 、 \sinh 、 \cosh 和 \tanh 等。指数函数主要是指幂函数、以 e 为底的幂函数、平方根函数、自然对数函数及以 10 为底的对数。

本小节将依次介绍上述超越函数。

1. 三角函数

C 语言提供的三角函数只有 6 种: \sin 、 \cos 、 \tan 、 \sinh 、 \cosh 和 \tanh 。其原型分别为:

```
complex<T> sin(const complex<T>& x);
complex<T> cos(const complex<T>& x);
complex<T> tan(const complex<T>& x);
complex<T> sinh(const complex<T>& x);
complex<T> cosh(const complex<T>& x);
complex<T> tanh(const complex<T>& x);
```

复数的三角函数计算是依托自然指数实现的。假定复数 $z = x + j^*y$, 则 $\sin(z)$ 的计算式为

$$\sin z = \sin(x + j^*y) = \sin x \left(\frac{e^y + e^{-y}}{2} \right) + j \cos x \left(\frac{e^y - e^{-y}}{2} \right)$$

$$\cos z = \cos(x + j^*y) = \cos x \left(\frac{e^y + e^{-y}}{2} \right) - j \sin x \left(\frac{e^y - e^{-y}}{2} \right)$$

其余的三角函数均可由正弦函数、余弦函数和自然指数获得。



提示 Visual C++ 6.0 没有提供复数的正切函数。

例如,

```
c11 = sin(c1);
cout << "sin(c1) = " << c11 << endl;
c11 = cos(c1);
cout << "cos(c1) = " << c11 << endl;
c11 = sin(c1)/cos(c1);
cout << "tan(c1) = " << c11 << endl;
```

```

c11 = sinh(c1);
cout << "sinh(c1) = " << c11 << endl;
c11 = cosh(c1);
cout << "cosh(c1) = " << c11 << endl;
c11 = sinh(c1)/cosh(c1);
cout << "tanh(c1) = " << c11 << endl;

```

上述程序的执行结果为：

```

sin(c1) = (6.12, 0.428)
cos(c1) = (0.434, -6.04)
tan(c1) = (0.00193, 1.01)
sinh(c1) = (-1.71, 1.41)
cosh(c1) = (-1.88, 1.27)
tanh(c1) = (0.968, -0.0926)

```

2. 其他超越函数

其他超越函数主要包括幂函数、以 e 为底的幂函数、平方根函数、自然对数函数及以 10 为底的对数。

1) 幂 `pow()` 函数。复数可以进行指数运算，即幂运算。STL 为复数提供了幂 `pow()` 函数。其原型为：

```

template < class T >
    complex < T > pow(const complex < T > & x, int y);
    complex < T > pow(const complex < T > & x, const T & y);
    complex < T > pow(const complex < T > & x, const complex < T > & y);
    complex < T > pow(const T & x, const complex < T > & y);

```

上述几种形式中，参数 x 为底，参数 y 为指数。

```

c3 = pow(c1, c2);
cout << "pow(c1, c2) = " << c3 << endl;
c3 = pow(c1, 2);
cout << "pow(c1, 2) = " << c3 << endl;
// c3 = pow(3.0, c1); //在 Visual C++ 6.0 环境中编译没有通过
// cout << "pow(3.0, c1) = " << c3 << endl; //在 Visual C++ 6.0 环境中编译没有通过

```

上述代码的执行结果为：

```

pow(c1, c2) = (-0.193, 0.0501)
pow(c1, 2) = (-4, 7.5)

```

2) 以 e 为底的幂 `exp()` 函数。以 e 为底的幂 `exp()` 函数的原型为：

```

template < class T >    complex < T > exp(const complex < T > & x);

```

`exp()` 函数的输入参数和返回值均为复数。

例如，

```

c3 = exp(c2);
cout << "exp(c2) = " << c3 << endl;

```

上述代码的执行结果为：

```
exp(c2) = (15.5, -29.3)
```

3) 平方根 `sqrt()` 函数。复数平方根 `sqrt()` 函数的原型为：

```
template <class T> complex<T> sqrt(const complex<T>& x);
```

`sqrt()` 函数的输入参数和输出参数均为复数类型。

例如，

```
c3 = sqrt(c1);
cout << "sqrt(c1) = " << c3 << endl;
c3 = sqrt(complex<float>(-5,0));
cout << "sqrt(-5) = " << c3 << endl;
```

上述代码的执行结果为：

```
sqrt(c1) = (1.49, 0.841)
sqrt(-5) = (0, 2.24)
```

4) 对数 `log()` 函数。对数函数可分为“以 10 为底的对数函数”和“自然对数函数”。其原型为：

```
template <class T> complex<T> log(const complex<T>& x);
template <class T> complex<T> log10(const complex<T>& x);
```

例如，

```
c3 = log(c1);
cout << "log(c1) = " << c3 << endl;
c3 = log10(c1);
cout << "log10(c1) = " << c3 << endl;
```

上述代码的执行结果为：

```
log(c1) = (1.07, 1.03)
log10(c1) = (0.465, 0.447)
```



总 结 6.1.5 节讲述了复数的超越函数。超越函数是最常见、最常用的数学函数，这些函数的使用方法较简单，读者了解即可。复数类模板最主要的成员函数包括 `real()`、`image()`、`abs()`、`norm()`、`arg()`、`<<`、`>>`、`conj()`、`polar()` 等。读者应熟练掌握，尤其是参与计算工作较多的程序开发人员，更是应该认真阅读本章的内容。

6.2 数组（向量）运算

STL 提供了一个数组类 `valarray`。类 `valarray` 用于实现数值数组的运算。`valarray` 代表一个数学概念：数值线性序列。该序列是一维的，但程序员可以通过运用特殊技巧得到多维序列。所谓的特殊技巧，即使用“索引”能力和“威力强大”的“子集”能力。因此，`valarray` 是向量和矩阵运算的基础。数值计算多依赖于浮点数值的一维向量，下标从零开始。类 `valarray` 的设计目的就是加速常用数值向量的计算。

类 `valarray` 的设计原则不是通用性和易用性，而是高效地利用计算机性能，尤其是高性能计算机。若程序员编写的程序要满足灵活性和通用性，则使用前面讲述的容器即可，没必要使用类 `valarray`。

6.2.1 类 `valarray`

通俗来讲，类 `valarray` 是经过优化的向量。类 `valarray` 是由 4 个用于描述 `valarray` 中各个部分的辅助类支持的。这 4 个类分别是：`slice_array`、`gslice_array`、`mask_array` 和 `indirect_array`。

其中 `slice_array`、`gslice_array`、`mask_array` 和 `indirect_array` 均用于存放临时数值或数据。

若要使用类 `valarray`，则必须包含头文件 `<valarray>`。本节主要讲述类 `valarray` 的构造函数、成员函数、超越函数、下标和赋值、二元运算和数学运算以及子集操作。其中，子集操作将在后面章节介绍。

1. 类 `valarray` 的构造函数

类 `valarray` 的构造函数一般有以下几种形式：

```
valarray();
explicit valarray(size_t n);
valarray(const T& val, size_t n);
valarray(const T* p, size_t n);
valarray(const slice_array<T>& sa);
valarray(const gslice_array<T>& ga);
valarray(const mask_array<T>& ma);
valarray(const indirect_array<T>& ia);
```

构造类 `valarray` 的对象时，最简单的形式是：将元素的数量作为输入参数，即上述构造函数的第二种形式和第三种形式。第二种形式的参数 `size_t n` 用于指定数组的大小；第三种形式的参数 `n` 用于指定数组的大小，参数 `val` 用作所有元素的初始默认值。这和 STL 中容器的构造函数不同。容器构造函数的第一个参数多数用于指定容器的大小，第二个参数用于指定容器中元素的初始默认值。

利用构造函数的第三种形式可以使用已知数组对 `valarray` 对象进行初始化。

例如，

```
int dim[] = {1,2,3,4,6,7,8,9};
valarray<int> val1(10);
valarray<int> val2(0,10);
valarray<int> val3(dim, sizeof(dim)/sizeof(int));
valarray<int> val4(val2);
cout << "valarray val1: " << endl;
print(val1);
cout << "valarray val2: " << endl;
print(val2);
cout << "valarray val3: " << endl;
print(val3);
cout << "valarray val4: " << endl;
print(val4);
```

上述代码的执行结果为：

```
valarray val1:
-842150451, -842150451, -842150451, -842150451, -842150451, -842150451, -8421504
51, -842150451, -842150451, -842150451,
valarray val2:
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
valarray val3:
1, 2, 3, 4, 6, 7, 8, 9,
valarray val4:
0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

2. 下标、赋值以及数学运算

类 `valarray` 还提供了一系列的下标操作`[]`、数学运算等。下标操作符`[]`使程序员可以方便地访问数组中的每个元素以及获取数值子集。类 `valarray` 还定义了所有普通数学运算，例如 `+`、`-`、`*`、模数、反相、位操作、位比较、逻辑操作、赋值操作等。数组的乘法运算是对应元素相乘，即

```
val = va2 * va3
```

等同于

```
val[0] = va2[0] * va3[0];
```

```
val[1] = va2[1] * va3[1];
```

...

若参与计算的数组容量不一致，则运算结果未有定义，即不进行计算。

3. 成员函数

类 `valarray` 提供了一系列的成员函数 `size()`、`sum()`、`max()`、`min()`、`resize()`、`shift()`、`cshift()`、`apply()`和 `free()`。

- 1) `size()` 函数。该函数用于获取数组的大小。
- 2) `max()` 函数。该函数用于获取数组中的最大值。
- 3) `min()` 函数。该函数用于获取数组中的最小值。
- 4) `sum()` 函数。该函数用于获取数组中所有元素的总和。
- 5) `resize()` 函数。该函数用于实现重新设置数组的大小。
- 6) `shift()`和 `cshift()` 函数。`shift()` 函数用于实现逻辑移位，`cshift()` 函数用于实现循环移位。移位即指将数组中的元素位置顺次移动。
- 7) `apply()` 函数。其原型为：

```
valarray<T> apply(T fn(T)) const;
valarray<T> apply(T fn(const T&)) const;
```

其中 `fn(T)`是指定的函数。`apply()` 函数的功能是：使用 `fn()` 函数处理数组中的每个元素。

- 8) `free()` 函数。其原型为：

```
void free();
```

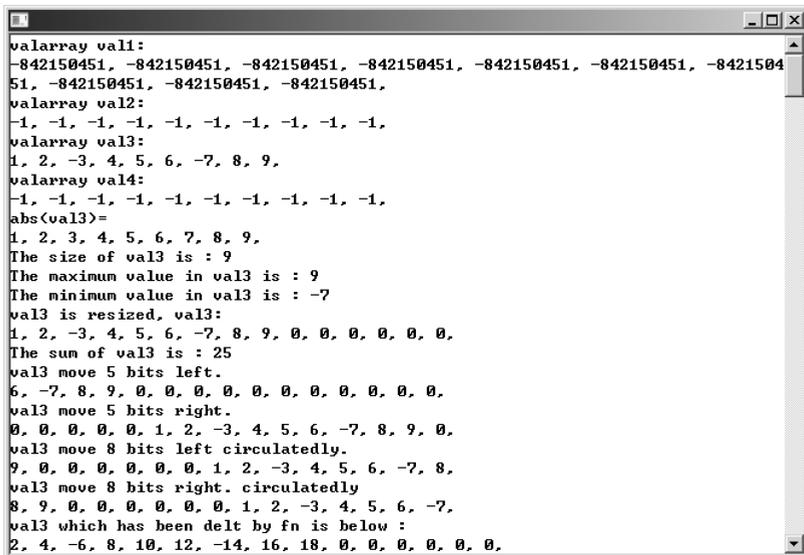
`free()` 函数用于删除数组的所有元素，并将数组长度设置为 0。

例 6-3

```
#include <iostream>
#include <valarray>
//#include <>
using namespace std;
void print (valarray<int> & array )
{   int size=array.size();
for(int i=0;i<size;i++)
    cout<<array[i]<<" ";
cout<<endl;
}
int fn(int ele)
{   int r=0;
    r=ele*2;
    return r;
}
void main()
{   int dim[] = {1,2, -3,4,5,6, -7,8,9};
    valarray<int> val1(10);
    valarray<int> val2(-1,10);
    valarray<int> val3(dim,sizeof(dim)/sizeof(int));
    valarray<int> val4(val2);
    valarray<int> val5;
    cout<<"valarray val1: " <<endl;
    print(val1);
    cout<<"valarray val2: " <<endl;
    print(val2);
    cout<<"valarray val3: " <<endl;
    print(val3);
    cout<<"valarray val4: " <<endl;
    print(val4);
    val5=abs(val3);
    cout<<"abs(val3) = " <<endl;
print(val5);
    int s=val3.size();
    int ma=val3.max();
    int mi=val3.min();
    cout<<"The size of val3 is : " <<s <<endl;
    cout<<"The maximum value in val3 is : " <<ma <<endl;
    cout<<"The minimum value in val3 is : " <<mi <<endl;
    val3.resize(15);
    cout<<"val3 is resized, val3: " <<endl;
    print(val3);
    int He=val3.sum();
    cout<<"The sum of val3 is : " <<He <<endl;
    val5=val3.shift(5);
```

```
cout << "val3 move 5 bits left. " << endl;
print(val5);
val5 = val3.shift(-5);
cout << "val3 move 5 bits right. " << endl;
print(val5);
val5 = val3.cshift(8);
cout << "val3 move 8 bits left circulatedly. " << endl;
print(val5);
val5 = val3.cshift(-8);
cout << "val3 move 8 bits right. circulatedly" << endl;
print(val5);
val5 = val3.apply(&fn);
cout << "val3 which has been delt by fn is below : " << endl;
print(val5);
val1.free();
val2.free();
val3.free();
val4.free();
val5.free();
}
```

例 6-3 的执行效果如图 6-3 所示。



```
valarray val1:
-842150451, -842150451, -842150451, -842150451, -842150451, -842150451, -8421504
51, -842150451, -842150451, -842150451,
valarray val2:
-1, -1, -1, -1, -1, -1, -1, -1, -1,
valarray val3:
1, 2, -3, 4, 5, 6, -7, 8, 9,
valarray val4:
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
abs(val3)=
1, 2, 3, 4, 5, 6, 7, 8, 9,
The size of val3 is : 9
The maximum value in val3 is : 9
The minimum value in val3 is : -7
val3 is resized, val3:
1, 2, -3, 4, 5, 6, -7, 8, 9, 0, 0, 0, 0, 0, 0, 0,
The sum of val3 is : 25
val3 move 5 bits left.
6, -7, 8, 9, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
val3 move 5 bits right.
0, 0, 0, 0, 0, 1, 2, -3, 4, 5, 6, -7, 8, 9, 0,
val3 move 8 bits left circulatedly.
9, 0, 0, 0, 0, 0, 0, 1, 2, -3, 4, 5, 6, -7, 8,
val3 move 8 bits right. circulatedly
8, 9, 0, 0, 0, 0, 0, 0, 1, 2, -3, 4, 5, 6, -7,
val3 which has been delt by fn is below :
2, 4, -6, 8, 10, 12, -14, 16, 18, 0, 0, 0, 0, 0, 0,
```

图 6-3 例 6-3 的执行效果



提示

请读者认真读例题。例题中包含了作者精心安排的各种编程细节。

4. 超越函数

类 `valarray` 可允许的超越函数包括 `abs()`、`pow()`、`exp()`、`sqrt()`、`log()`、`log10()`、

$\sin()$ 、 $\cos()$ 、 $\tan()$ 、 $\sinh()$ 、 $\cosh()$ 、 $\tanh()$ 、 $\operatorname{asin}()$ 、 $\operatorname{acos}()$ 、 $\operatorname{atan}()$ 和 $\operatorname{atan2}()$ 。以上函数均返回一个新的 `valarray` 型数组，元素个数和该函数输入的数组大小相同。上述函数的功能是针对数组中的每个元素分别进行超越函数计算。其使用方法详见例 6-4。

1) `abs()` 函数。其原型为：

```
template < class T > valarray < T > abs (const valarray < T > & x);
```

`abs()` 函数的功能是对数组中的每个元素进行绝对值计算，获取每个元素的绝对值。

2) `pow()` 函数。其原型为：

```
template < class T > valarray < T > pow (const valarray < T > & x, const valarray < T > & y);  
template < class T > valarray < T > pow (const valarray < T > x, const T & y);  
template < class T > valarray < T > pow (const T & x, const valarray < T > & y);
```

若要使用 `pow()` 函数，需要包含数学库头文件 `<cmath>`。对于类 `valarray` 的对象（数组），`pow()` 函数主要有上述三种形式。在 Visual C++ 6.0 环境中，本书作者仅编译通过了 `pow()` 函数的第一种形式。

3) `exp()` 函数。其原型为：

```
template < class T > valarray < T > exp (const valarray < T > & x);
```

若要使用 `exp()` 函数，同样需要包含头文件 `<cmath>`。

4) `sqrt()` 函数。其原型为：

```
template < class T > valarray < T > sqrt (const valarray < T > & x);
```

若要使用 `sqrt()` 函数，同样需要包含头文件 `<cmath>`。

5) `log()` 和 `log10()` 函数。其原型为：

```
template < class T > valarray < T > log (const valarray < T > & x);  
template < class T > valarray < T > log10 (const valarray < T > & x);
```

若要使用 `log()` 和 `log10()` 函数，同样需要包含头文件 `<cmath>`。

6) 三角函数。其原型为：

```
template < class T > valarray < T > sin (const valarray < T > & x);  
template < class T > inline valarray < T > cos (const valarray < T > & x);  
template < class T > inline valarray < T > tan (const valarray < T > & x);  
template < class T > inline valarray < T > sinh (const valarray < T > & x);  
template < class T > inline valarray < T > cosh (const valarray < T > & x);  
template < class T > inline valarray < T > tanh (const valarray < T > & x);  
template < class T > inline valarray < T > asin (const valarray < T > & x);  
template < class T > inline valarray < T > acos (const valarray < T > & x);  
template < class T > inline valarray < T > atan (const valarray < T > & x);  
template < class T > inline valarray < T > atan2 (const valarray < T > & x, const valarray < T > & y);  
template < class T > inline valarray < T > atan2 (const valarray < T > x, const T & y);  
template < class T > inline valarray < T > atan2 (const T & x, const valarray < T > & y);
```

若要使用三角函数，同样需要包含头文件 `<cmath>`。

上述三角函数中，`atan2()` 函数有三种形式。本文作者在 Visual C++ 6.0 环境中，仅调试通过了第一种形式（**黑体**）。

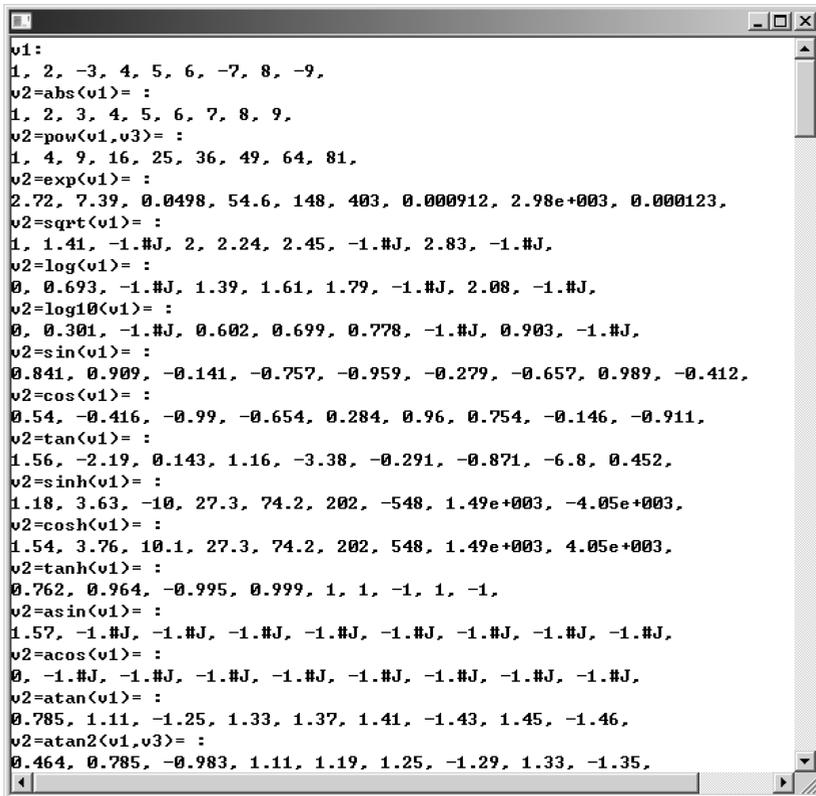
上述超越函数的使用方法详见例 6-4。

例 6-4

```
#include <iostream>
#include <valarray>
#include <cmath>
using namespace std;
void print (valarray<double>& array )
{int size=array.size();
cout.precision(3);
for(int i=0;i<size;i++)
    cout<<array[i]<<" ";
cout<<endl;
}
void main()
{double dim[] = {1,2, -3,4,5,6, -7,8, -9};
valarray<double> v1 (dim,sizeof(dim)/sizeof(double));
valarray<double> v2;
valarray<double> v3(2,9);
cout<<"v1: " <<endl;
print(v1);
v2 = abs(v1);
cout<<"v2 = abs(v1) = : " <<endl;
print(v2);
v2 = pow(v1,v3);
cout<<"v2 = pow(v1,v3) = : " <<endl;
print(v2);
v2 = exp(v1);
cout<<"v2 = exp(v1) = : " <<endl;
print(v2);
v2 = sqrt(v1); //如果元素为负值,输出时仅表明结果为虚数
cout<<"v2 = sqrt(v1) = : " <<endl;
print(v2);
v2 = log(v1);
cout<<"v2 = log(v1) = : " <<endl;
print(v2);
v2 = log10(v1);
cout<<"v2 = log10(v1) = : " <<endl;
print(v2);
v2 = sin(v1);
cout<<"v2 = sin(v1) = : " <<endl;
print(v2);
v2 = cos(v1);
cout<<"v2 = cos(v1) = : " <<endl;
print(v2);
v2 = tan(v1);
cout<<"v2 = tan(v1) = : " <<endl;
print(v2);
v2 = sinh(v1);
```

```
cout << "v2 = sinh(v1) = :" << endl;
print(v2);
v2 = cosh(v1);
cout << "v2 = cosh(v1) = :" << endl;
print(v2);
v2 = tanh(v1);
cout << "v2 = tanh(v1) = :" << endl;
print(v2);
v2 = asin(v1);
cout << "v2 = asin(v1) = :" << endl;
print(v2);
v2 = acos(v1);
cout << "v2 = acos(v1) = :" << endl;
print(v2);
v2 = atan(v1);
cout << "v2 = atan(v1) = :" << endl;
print(v2);
v2 = atan2(v1, v3);
cout << "v2 = atan2(v1, v3) = :" << endl;
print(v2);
}
```

例 6-4 的执行效果如图 6-4 所示。



```
v1:
1, 2, -3, 4, 5, 6, -7, 8, -9,
v2=abs(v1)= :
1, 2, 3, 4, 5, 6, 7, 8, 9,
v2=pow(v1,v3)= :
1, 4, 9, 16, 25, 36, 49, 64, 81,
v2=exp(v1)= :
2.72, 7.39, 0.0498, 54.6, 148, 403, 0.000912, 2.98e+003, 0.000123,
v2=sqrt(v1)= :
1, 1.41, -1.#J, 2, 2.24, 2.45, -1.#J, 2.83, -1.#J,
v2=log(v1)= :
0, 0.693, -1.#J, 1.39, 1.61, 1.79, -1.#J, 2.08, -1.#J,
v2=log10(v1)= :
0, 0.301, -1.#J, 0.602, 0.699, 0.778, -1.#J, 0.903, -1.#J,
v2=sin(v1)= :
0.841, 0.909, -0.141, -0.757, -0.959, -0.279, -0.657, 0.989, -0.412,
v2=cos(v1)= :
0.54, -0.416, -0.99, -0.654, 0.284, 0.96, 0.754, -0.146, -0.911,
v2=tan(v1)= :
1.56, -2.19, 0.143, 1.16, -3.38, -0.291, -0.871, -6.8, 0.452,
v2=sinh(v1)= :
1.18, 3.63, -10, 27.3, 74.2, 202, -548, 1.49e+003, -4.05e+003,
v2=cosh(v1)= :
1.54, 3.76, 10.1, 27.3, 74.2, 202, 548, 1.49e+003, 4.05e+003,
v2=tanh(v1)= :
0.762, 0.964, -0.995, 0.999, 1, 1, -1, 1, -1,
v2=asin(v1)= :
1.57, -1.#J, -1.#J, -1.#J, -1.#J, -1.#J, -1.#J, -1.#J, -1.#J,
v2=acos(v1)= :
0, -1.#J, -1.#J, -1.#J, -1.#J, -1.#J, -1.#J, -1.#J, -1.#J,
v2=atan(v1)= :
0.785, 1.11, -1.25, 1.33, 1.37, 1.41, -1.43, 1.45, -1.46,
v2=atan2(v1,v3)= :
0.464, 0.785, -0.983, 1.11, 1.19, 1.25, -1.29, 1.33, -1.35,
```

图 6-4 例 6-4 的执行效果



总结 本节讲述了数组类 `valarray` 的内容，主要涉及的内容包括数组的最基本使用、访问、基本操作以及超越函数。希望读者对照给出的源代码，认真阅读源代码，以熟练掌握类 `valarray`。

6.2.2 数组子集类——类 `slice` 和类模板 `slice_array`

`slice` 是切割的意思，即将一个向量作为任意维数组去操作。“切割”是指在一个 `valarray` 中，间距为 `n` 的多个元素。`slice` 的跨步是切割中两个元素的距离（间隔的下标个数）。每次切割均具备以下 3 个属性：

- 1) 起始下标。
- 2) 元素个数。
- 3) 元素的间距。

切割可以从一个数组中提出部分元素，组成一个新数组（`valarray`）。

从一个 `valarray` 和一个 `slice`，可以构造出感觉上类似 `valarray` 而实际上是一种由切割描述的数组子集。使用 `slice` 方式可以创建数组的各种各样的子集。STL 没有提供矩阵类，使用 `slice` 方式构造出由不同需要而优化的各种矩阵。因此矩阵的表示即是一个 `valarray`，通过切割（`slice`）的方式，使 `valarray` 类型数组拥有了维数。

为方便运算和操作，程序员需要将数组的子集转换成类。`slice_array` 类由此应运而生。`slice_array` 是为切割（`slice`）提供内部辅助类别。对用户而言，`slice_array` 类内部是完全透明的，但类模板 `slice_array` 的所有构造函数和赋值操作符均是私有类型（`private`）。

`slice_array` 类的操作均有定义：

- 1) 将同一值赋予每一个元素。
- 2) 赋值另一个 `valarray`。
- 3) 调用任何一个赋值复合运算。如果希望完成其他操作，必须先将子集转换成 `valarray` 类的对象才可以。

同一个 `valarray` 的不同 `slice` 方式，可以组合成诸多子集，自己结果可以存入 `valarray` 类型的另一个子集中。在此基础上，`valarray` 型数组可作为二维矩阵看待。

对于 `valarray`，用户不会直接创建一个 `valarray` 类的对象。实际情况是，根据程序员描述 `valarray` 类对象的下标，为给定切割（`slice`）建立一个 `slice_array` 类型的数组。值得一提的是，`slice_array` 是禁止复制的，但 `slice` 是允许复制的。

下面列举两个例题作为本节学习的范例。

例 6-5

```
#include <valarray>
#include <iostream>
#include <cmath>
using namespace std;
void myprint(valarray<double>& v)
{int size=v.size();
 for(int i=0;i<size;i++)
  cout<<v[i]<<" ";
```

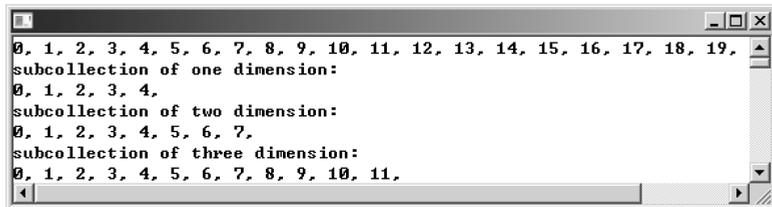
```

    cout << endl;
}
void main()
{valarray < double > v1 (12),v2;
    for(int i=0;i<12;i++)
    {
        v1[i]=2* (i+1);
    }
    cout << "v1 (Original) : " << endl;
    myprint (v1);
    valarray<double> tv1 =valarray<double> (v1[ slice (0,4,3)]); //slice 三个参数:始下标,数目,跨度

    valarray<double> tv2 =valarray<double> (v1[ slice (2,4,3)]);
    v1[ slice (0,4,3)] =pow (tv1,tv2);
    cout << "v1 (Calculated) : " << endl;
    myprint (v1);
    valarray<double> v3 (v1[ slice (0,4,3)]);
    cout << "v3: " << endl;
    myprint (v3);
}

```

例 6-5 的执行效果如图 6-5 所示。



```

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
subcollection of one dimension:
0, 1, 2, 3, 4,
subcollection of two dimension:
0, 1, 2, 3, 4, 5, 6, 7,
subcollection of three dimension:
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,

```

图 6-5 例 6-5 的执行效果

例 6-6

```

#include < iostream >
#include < valarray >
using namespace std;
void myprint (valarray<double> & v)
{int size=v.size();
for(int i=0;i<size;i++)
    cout << v[i] << ", ";
cout << endl;
}
void main()
{valarray<double> v1 (12),v2;
for(int i=0;i<12;i++)
    v1[i]=2* (i+1);
cout << "v1 (Original) : " << endl;

```

```

myprint(v1);
slice myslice(0,4,3);
v2 = v1[myslice];
cout << "v2 (v1[slice(0,4,3)]) : " << endl;
myprint(v2);
}

```

例 6-6 的执行效果如图 6-6 所示。

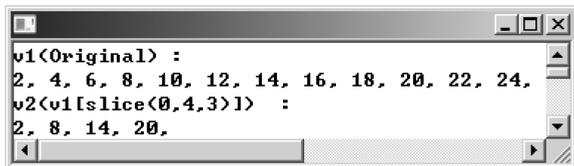


图 6-6 例 6-6 的执行效果



总结

读者要认真阅读上述两个例题，体会 slice 类和类模板 slice_array 的用法。

6.2.3 类 gslice 和类模板 gslice_array

gslice 是 general slice 的简称。类 gslice 和类 slice 相似，同样具有 3 个属性：起始索引、元素数量和元素间距。gslice 的元素数量和间距也是数组，其中的元素个数和其维度相同。gslice 既可以处理一维数组和二维数组，也可以处理三维数组。

类 gslice 实现的是一个广义切割，会包含 n 个切割的所有信息。同样，类模板 gslice_array 提供了与 slice_array 同样的一组成员。类模板 gslice_array 不能直接由用户构造或复制。客观来讲，类模板 gslice_array 即使用 gslice 作为一个 valarray 的下标的结果。

由以上内容可知，类 gslice 和类 slice 的区别在于类 gslice 能够运用数组来定义大小和间距，除此之外两者相同。即，

1) 当需要定义 valarray 数组的具体子集时，可以将类 gslice 作为参数传递给 valarray 的下标操作符。

2) 若 valarray 数组是常量，则子集表达式将导致一个新的 valarray 数组。

3) 若 valarray 数组不是常量，则子集表达式将导致一个 gslice_array。

4) 类模板 gslice_array 提供赋值操作符和复合赋值操作符，用以修改子集内的元素。

5) 通过型别转换，可以将 gslice_array 和其他 valarray 以及 valarray 子集组合起来。

下面讲述如何使用类 gslice 实现处理一维数组和多维数组。

1) 对于一维 valarray 类型数组，使用类 gslice 产生一维子集的方法如下：

```
valarray v1[15] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}
```

若 gslice 为 gslice(2, 4, 3)，则返回的子集为 {2, 5, 8, 11}。

分析：从起始下标 2 开始，共包含 4 个数，数之间的间隔为 3。

2) 对于一维 valarray 类型数组，使用类 gslice 产生二维子集的方法如下：

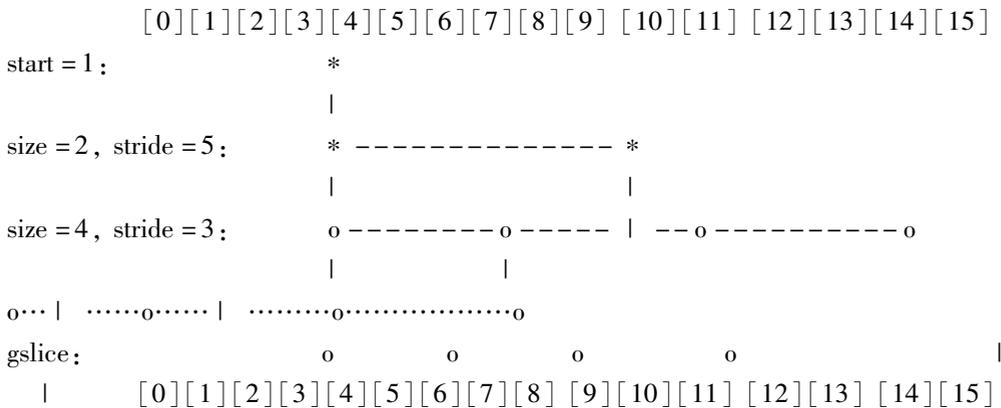
```
valarray<int> v1[15] = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
int length[] = {2,4};
int stride[] = {5,3};
```

若 `gslice` 为 `gslice (1, length, stride)`，则返回子集为：

```
{1, 4,7,10,
6, 9,12,15 }
```

可以将执行 `gslice (1, length, stride)` 后所得的新数据组合成两行四列的二维数组，每行 4 个元素，共有两行。

下面分析二维数组的情况：



最终获取的元素为：

```
{1, 4,7,10 6, 9,12,15 }
```

3) 对于一维 `valarray` 类型数组，使用类 `gslice` 产生三维子集的方法如下：

```
valarray<int> v1[20] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20};
int length[] = {3,2,4};
int stride[] = {30,10,3};
```

若 `gslice` 为 `gslice (1, length, stride)`，则返回子集为：

```
{2 , 5, 8, 11, 12, 18, 3452816845, 4198900, 0, 134272245, 4655572, 131, 51, 46
92280, 4692376, 4692728, 4692824, 0, 0, 0, 0, 0, 0 }
```

分析：由于数组 `stride[]` 的内容过大，导致所提供的下标超出了数组 `v1` 的范围，因此得到的数据是乱数据，不是数组 `v1` 中的数据。

若数组 `v1` 容量定义为 100，数组元素为从 1 至 100 的整数值：

```
valarray<int> v1[100] = {1,2,3,4,5,...100};
int length[] = {3,2,4};
int stride[] = {30,10,3};
```

同样，若执行语句：

```
gslice g1(1,length,stride);
```

则返回的子集为：

```
{ 2, 5, 8, 11, 12, 15, 18, 21, 32, 35, 38, 41, 42, 45, 48, 51, 62, 65, 68, 71, 72,75, 78, 81 }
```

分析：可以将执行 `g1 (1, length, stride)` 之后所得数组作为三行两列四层的三维数组。起始下标为 1；

当 $size = 3(\text{length}[0])$ 时, 对应的 $stride[0] = 30$, 获取的元素为: 2, 32, 62。

当 $size = 2(\text{length}[1])$ 时, 对应的 $stride[1] = 10$, 获取的元素依次为: {2, 12}, {32, 42}, {62, 72}。

当 $size = 4(\text{length}[2])$ 时, 对应的 $stride[2] = 3$, 获取的元素应该为:

{2, 5, 8, 11},
 {12, 15, 18, 21},
 {32, 35, 38, 41}
 {42, 45, 48, 51}
 {62, 65, 68, 71}
 {72, 75, 78, 81}

其实在内存中, 三维数组的排列是分层的。本例中, 可以写成 4 个二维数组, 分别代表 1, 2, 3, 4 层二维数组。

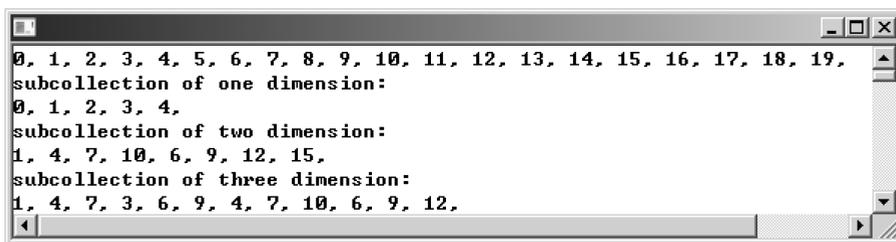
第一层: 2, 12 32, 42 62, 72	第二层: 5, 15 35, 45 65, 75	第三层: 8, 18 38, 48 68, 78	第四层: 11, 21 41, 51 71, 81
---	---	---	---

例 6-7

```
#include <iostream>
#include <valarray>
using namespace std;
void main()
{
    valarray<size_t> v1(20),v2,v3,v11;
    for(size_t i=0;i<20;i++)
    {
        v1[i]=i;
        cout<<i<<" ";
    }
    cout<<endl;
    valarray<size_t> len(5,1);
    valarray<size_t> stri(3,1);
    gslice g1(1,len,stri);
    v11=v1[g1];
    cout<<"subcollection of one dimension: "<<endl;
    int size=v11.size();
    for(size_t j=0;j<size;j++)
    {
        //v1[j]=j;
        cout<<v1[j]<<" ";
    }
    cout<<endl;
```

```
//以上是一维数组
size_t lengthv[] = {2,4};
size_t stridev[] = {5,3};
valarray< size_t > length(lengthv,2);           //构造赋值
valarray< size_t > stride(stridev,2);          //构造赋值
gslice g(1,length, stride);
v2 = v1[g];
cout << "subcollection of two dimension: " << endl;
size = v2.size();
for(j=0;j < size;j++)
{
    cout << v2[j] << ", ";
}
cout << endl;
//以上是二维子集
size_t lengthv2[] = {2,2,3};
size_t stridev2[] = {3,2,3};
valarray< size_t > length2(lengthv2,3);        //构造赋值
valarray< size_t > stride2(stridev2,3);        //构造赋值
gslice g2(1,length2, stride2);
v3 = v1[g2];
cout << "subcollection of three dimension: " << endl;
size = v3.size();
for(j=0;j < size;j++)
{
    cout << v3[j] << ", ";
}
cout << endl;
//以上是三维子集
}
```

例 6-7 的执行效果如图 6-7 所示。



```
0. 1. 2. 3. 4. 5. 6. 7. 8. 9. 10. 11. 12. 13. 14. 15. 16. 17. 18. 19.
subcollection of one dimension:
0. 1. 2. 3. 4.
subcollection of two dimension:
1. 4. 7. 10. 6. 9. 12. 15.
subcollection of three dimension:
1. 4. 7. 3. 6. 9. 4. 7. 10. 6. 9. 12.
```

图 6-7 例 6-7 的执行效果



总结

请读者认真阅读上述例题，仔细体会类 `gslice` 和类模板 `gslice_array` 的使用方法。

6.2.4 类 mask_array

数组类 `valarray` 还提供了另一种描述数组子集的方式：屏蔽子集。`valarray` 类的对象经过“屏蔽”处理后，其返回结果为 `valarray < bool >` 型。将“屏蔽”作为数组（`valarray`）的下标，值为 `true` 的位表明对应的 `valarray` 型数组中的元素将作为结果的一部分。在此基础上，程序员同样可以在 `valarray` 的特定子集上操作。和 `slice_array` 相同，`mask_array` 不能由程序开发者构造或复制。`mask_array` 是将一个 `valarray < bool >` 型对象用作 `valarray` 数组下标的结果。屏蔽的 `valarray` 的元素个数不能多于以它作为下标的那个 `valarray` 的元素个数。

由上述说明可知，类 `mask_array` 的功能就是使用布尔表达式屏蔽相应元素。具体地说，`mask_array` 和所有 `valarray subsets` 一样，具有以下属性：

1) 若要定义某 `valarray` 型数组的具体子集，可以将布尔型的 `valarray` 作为参数传给 `valarray` 的下标操作符。

2) 如果 `valarray` 是常量，子集表达式将导致新的 `valarray`。

3) 若 `valarray` 型数组不是常量，子集表达式将导致一个 `mask_array`，后者以 `reference` 语义来表现 `valarray` 的相应元素。

4) `mask_array` 提供赋值操作符和复合赋值操作符，用来修改子集中的元素。

5) 通过型别转换，将 `mask_array` 与其他 `valarray` 和 `valarray` 的子集组合起来。

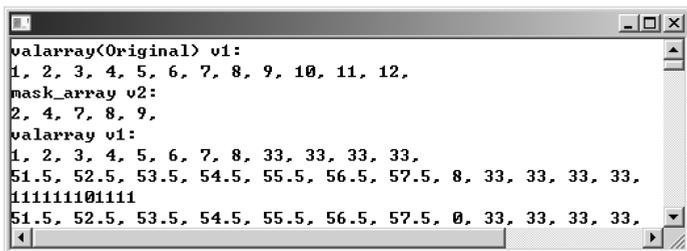
`mask_array` 的使用方法详见例 6-8。

例 6-8

```
#include <iostream>
#include <valarray>
#include <cmath>
using namespace std;
void myprint(valarray<double>& v)
{   int size=v.size();
    for(int i=0;i<size;i++)
        cout<<v[i]<<" ";
    cout<<endl;
}
void main()
{   double dim[12] = {1,2,3,4,5,6,7,8,9,10,11,12};
    valarray<double> v1(dim,sizeof(dim)/sizeof(double),v2,v3;
    cout<<"valarray(Original) v1: " <<endl;
    myprint(v1);
    bool B[] = {0,1,0,1,0,0,1,1,1};
    valarray<bool> mask_array(B,9);
    v2 = v1[mask_array];
    cout<<"mask_array v2:" <<endl;
    myprint(v2);
    v1[v1 > 8.0] = 33.0;
    cout<<"valarray v1: " <<endl;
    myprint(v1);
```

```
v1[v1 < 8.0] = valarray<double>(v1[v1 < 8.0]) + 50.5;
myprint(v1);
bool B2[12];
for(int i=0;i<12;i++)
{ B2[i] = (bool) fmod(v1[i],2);
}
for(i=0;i<12;i++)
{ cout << B2[i];
}
cout << endl;
valarray<bool> vB2(B2,12);
v1[! vB2] = 0;
myprint(v1);
}
```

例 6-8 的执行效果如图 6-8 所示。



```
valarray<Original> v1:
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12.
mask_array v2:
2, 4, 7, 8, 9.
valarray v1:
1, 2, 3, 4, 5, 6, 7, 8, 33, 33, 33, 33.
51.5, 52.5, 53.5, 54.5, 55.5, 56.5, 57.5, 8, 33, 33, 33, 33.
111111101111
51.5, 52.5, 53.5, 54.5, 55.5, 56.5, 57.5, 0, 33, 33, 33, 33.
```

图 6-8 例 6-8 的执行效果



总结 请读者要认真阅读上述例题，仔细体会类 `mask_array` 的功能和使用方法。

6.2.5 类 `indirect_array`

间接数组子集也是创建数组子集的一种方式，还可以任意排列元素。这是定义数组子集的第四种方法。同样，通过传递一个索引数组，即可定义一个 `valarray` 的子集。间接数组子集可以使用类 `indirect_array` 来表示。但是类 `indirect_array` 不能直接由用户构造或复制，和 `mask_array` 的用法一样，`indirect_array` 是将 `valarray<size_t>` 用做 `valarray` 的下标而产生的子集。作为下标的 `valarray` 的元素个数必须小于原始 `valarray` 数组。

另外，类 `indirect_array` 和所有 `valarray` 的子集一样，具有以下属性。

- 1) 若要定义某个 `valarray` 的子集，可以使用元素型别为 `size_t` 的 `valarray` 作为参数传递给 `valarray` 的下标操作符。
- 2) 若 `valarray` 是常量，则子集表达式将导致新的 `valarray`。
- 3) 若 `valarray` 不是常量，子集表达式将导致一个 `indirect_array` 型数组，后者以 `reference` 语意来表现 `valarray` 的相应元素。
- 4) `indirect_array` 型数组提供赋值操作符和赋值复合操作符来修改子集中的元素。

5) 通过型别转换, 可以将 `indirect_array` 型数组和其他 `valarray` 以及 `valarrays` 子集组合起来。



用来指定子集的索引无需排序, 并且可以重复出现。

例 6-9

```
#include <iostream>
#include <valarray>
#include <cmath>
using namespace std;
template <class T> void myprint (valarray <T> & v)
{
    int size=v.size();
    for(int i=0;i<size;i++)
    {
        cout << v[i] << ", ";
    }
    cout << endl;
}
void main()
{
    double dim[] = {1,2,3,4,5,6,7,8,9,10,11,12};
    valarray <double> v(dim,12),v2,v3;
    cout << "valarray v(Original) : " << endl;
    myprint (v);
    valarray <size_t> vi(4);
    vi[0]=4;
    vi[1]=6;
    vi[2]=7;
    vi[3]=1;
    v2=v[vi];
    cout << "valarray v2(indirect_array) : " << endl;
    myprint (v2);
    v3=pow(v,v2);
    cout << "v3=pow(v,v2) : " << endl;
    myprint (v3);
}
```

例 6-9 的执行效果如图 6-9 所示。

```
valarray v(Original) :
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12.
valarray v2(indirect_array) :
5, 7, 8, 2.
v3=pow(v,v2) :
1, 128, 6561, 16, 0, 0, 1, 1, 1, 1, 0, 12.
```

图 6-9 例 6-9 的执行效果

6.3 通用数值计算

STL 按照 `<algorithm>` 中非数值算法的风格, 提供了 4 种通用的数值算法: 求和、内积、部分和以及相邻差。这 4 种算法的声明均包含在头文件 `<numeric>` 中。这 4 种算法函数分别为: `accumulate()`、`inner_product()`、`partial_sum()` 和 `adjacement_difference()`。

6.3.1 求和算法 `accumulate()`

`accumulate()` 算法的原型为:

```
template <class InputIterator, class Type > Type accumulate( InputIterator _First, InputIterator
_Last, Type _Val );
template <class InputIterator, class Type, class Fn2 > Type accumulate( InputIterator _First, In
putIterator _Last,
Type _Val, BinaryOperation _Binary_op );
```

上述算法的第一种形式是计算序列 `[_First, _Last]` 的数值总和, 并将该数值总和添加至数值 `_Val` 上。第二种形式对序列 `[_First, _Last]` 的每个元素均使用谓词 `_op` 进行处理, 处理之后的结果依次累加, 并将所得总和再添加至数值 `_Val` 上。

谓词 `op` 的原型为:

```
_Binary_op (_Val, * Iter)
```

其中, `* Iter` 代表序列中的每个元素。`_Val` 即是算法 `accumulate` 中的参数 `_Val`。

例 6-10

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>
#include <functional>
using namespace std;
void myprint(int& ele)
{ cout << ele << ", ";
}
int myop(int orig, int ele)
{ return ele * 2 + orig;
}
void main()
{ vector<int> vt;
int dim[] = {1,2,3,4,5,6,7,8,9,10,11,12};
vt.assign(dim, dim+12);
for_each(vt.begin(), vt.end(), myprint);
cout << endl;
int sum = accumulate(vt.begin(), vt.end(), 0);
cout << "The sum of vector vt is : " << sum << endl;
sum = accumulate(vt.begin(), vt.end(), 0, myop);
cout << "The sum2 of vector vt is : " << sum << endl;
}
```

例 6-10 的执行效果如图 6-10 所示。

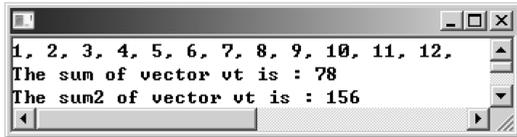


图 6-10 例 6-10 的执行效果

6.3.2 内积算法 inner_product ()

内积是通过把第一个序列中的元素与第二个序列中相同位置的元素进行相乘，然后把计算的结果相加而得到。内积算法的原型为：

```
template<class InIt1, class InIt2, class T> T product (InIt1 first1, InIt1 last1, InIt2 first2, T val);
template<class InIt1, class InIt2, class T, class Pred1, class Pred2 > T product (InIt1 first1, InIt1 last1, InIt2 first2, T val, Pred1 pr1, Pred2 pr2);
```

第一种形式用于计算并返回 [beg, end] 和 “以 beg2 为起始的区间” 的对应元素组的内积。具体来说，即针对 “两区间内的每一组对应元素” 调用以下表达式：

```
initValue = initValue + elem1 * elem2
```

第二种形式对区间 [beg, end] 和 “以 beg2 为起始区间” 内的对应元素进行 op2 运算，然后再和 initValue 进行 op1 运算，并将结果返回。具体来说，即针对 “两区间内的每一组对应元素” 调用以下表达式：

```
initValue = op1 (initValue, op2 (elem1, elem2))
```

由上述内容可知，对于两个数值序列：

```
a1, a2, a3, a4, ...
b1, b2, b3, b4, ...
```

使用上述两种算法分别计算并返回：

```
initValue + (a1 * b1) + (a2 * b2) + (a3 * b3) + ...
```

和

```
initValue op1 (a1 op2 b1) op1 (a2 op2 b2) op1 (a3 op2 b3) op1 ...
```

需要注意以下几点：

- 1) 若第一个区间为空，则两者返回 initValue。
- 2) 程序员必须确保 “以 beg2 为起始的区间” 内含足够的元素空间。
- 3) 谓词（或函数）op1 和 op2 都不允许修改传入其内的参数。
- 4) op1 和 op2 两个函数或两个谓词，均是两个参数。

下面以例 6-11 为例说明内积算法的使用方法。

例 6-11

```
#include <iostream>
#include <list>
#include <numeric>
#include <algorithm>
using namespace std;
void myprint(int ele)
{ cout << ele << ", ";
}
int op1(int initV,int ele3)
{ return initV+ele3;
}
int op2(int ele1,int ele2)
{ return ele1* 2 +ele2* 3;
}
void main()
{ int dim[] = {1,2,3,4,5,6,7,8,9,10,11,12};
  int dim2[] = {3,4,5,6,7,8,9,10,11,12,13,14};
  list<int> l1,l2;
  copy(dim,dim+12,back_inserter(l1));
  for_each(l1.begin(),l1.end(),myprint);
  cout << endl;
  copy(dim2,dim2+12,back_inserter(l2));
  for_each(l2.begin(),l2.end(),myprint);
  cout << endl;
  int prod = inner_product(l1.begin(),l1.end(),l2.begin(),0);
  cout << "inner_product of l1 and l2 is : " << prod << endl;
  prod = inner_product(l1.begin(),l1.end(),l2.begin(),0,op1,op2);
  cout << "inner_product of l1 and l2 with function op1 and op2 is : " << prod << endl;
  int sum = 0;
  for(int j = 0; j < 12; j++)
  { sum += (dim[j]* 2 + dim2[j]* 3);
  }
  cout << "inner_product of l1 and l2 with function op1 and op2 is : " << sum << endl;
  //由此证明定义的两个函数 op1
  // 和 op2 是正确的。
}
```

 **总结** 多数内积算法的例题中，op1 和 op2 采用的是 multiplies <int>() 和 plus <int>()。这样定义 op1 和 op2，读者并不能确切地理解 op1 和 op2 到底是如何进行运算的。本例定义了两个自定义谓词 op1 和 op2，代码清楚，功能简单，便于读者理解。

例 6-11 的执行效果如图 6-11 所示。

```

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12.
3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14.
inner_product of l1 and l2 is : 806
inner_product of l1 and l2 with function op1 and op2 is : 462
inner_product of l1 and l2 with function op1 and op2 is : 462

```

图 6-11 例 6-11 的执行效果

6.3.3 部分和算法 partial_sum()

partial_sum()算法的功能是对迭代器[_First, _Last]确定范围内的元素进行部分求和。其计算结果保存在以_Result为起始的序列中。部分求和的计算步骤：首先，把原序列的第1个元素赋值给新序列的第1个元素；其次，把原序列的前两个元素相加，把结果赋值给新序列的第2个元素；再次，把原序列的第2和第3个元素相加，把结果赋值给新序列的第3个元素；以此类推，直到计算完原序列的全部元素为止。

partial_sum()算法的原型为：

```

template<class InputIterator, class OutIt> OutputIterator partial_sum( InputIterator _First,
InputIterator _Last,
OutputIterator _Result );
template<class InputIterator, class OutIt, class Fn2> OutputIterator partial_sum(InputIterator
_First, InputIterator _Last,
OutputIterator _Result, BinaryOperation _Binary_op);

```

其中，第一种形式用于计算源区间[_First, _Last]中每个元素的部分和，然后将结果写入以result为起点的目标区间；第二种形式包含了一个谓词_Binary_op，对源区间中的每个元素进行_Binary_op处理。

通过以上描述可知，对于序列a, b, c, d, partial_sum使用()算法可产生序列a, a + b, a + b + c, a + b + c + d。

需要注意的问题如：

- 1) 两种形式都返回目标区间内“最后一个被写入的值”的下一位置。
- 2) 源区间和目标区间可以相同。
- 3) 程序员必须确保目标区间足够大，否则需要使用插入型迭代器。
- 4) 谓词_Binary_op不允许修改传入其内的参数。

目前参考资料给出的谓词多为multiplies<T>()等类型。

下面以例6-12说明partial_sum()算法的使用方法。其中给出的谓词为自定义函数。

例 6-12

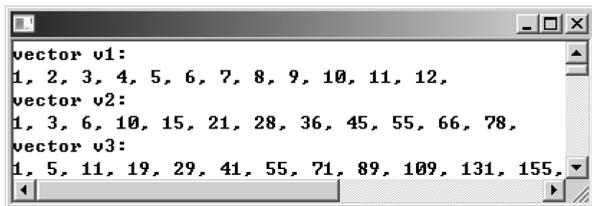
```

#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>
#include <functional>
using namespace std;

```

```
void myprint (int& ele)
{   cout << ele << ", ";
}
int op (int pre_ele, int ele)
{   return pre_ele + ele* 2;
}
void main ()
{   vector< int > v1, v2, v3;
    int dim[] = {1,2,3,4,5,6,7,8,9,10,11,12};
    v1.assign (dim, dim + 12);
    cout << "vector v1: " << endl;
    for_each (v1.begin (), v1.end (), myprint);
    cout << endl;
    partial_sum (v1.begin (), v1.end (), back_inserter (v2));
    cout << "vector v2: " << endl;
    for_each (v2.begin (), v2.end (), myprint);
    cout << endl;
    partial_sum (v1.begin (), v1.end (), back_inserter (v3), op);
    cout << "vector v3: " << endl;
    for_each (v3.begin (), v3.end (), myprint);
    cout << endl;
}
```

例 6-12 的执行效果如图 6-12 所示。



```
vector v1:
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
vector v2:
1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78,
vector v3:
1, 5, 11, 19, 29, 41, 55, 71, 89, 109, 131, 155,
```

图 6-12 例 6-12 的执行效果



总结

请读者认真阅读例题，尤其注意谓词函数的定义。

6.3.4 序列相邻差算法 adjacent_difference ()

adjacent_difference () 算法用于计算序列的相邻差，即计算由迭代器指定的源区间 [_First, _Last] 内相邻元素的差值，并将结果赋至以 _Result 为起始的序列中。序列相邻差的计算步骤为：首先，把源序列的第一个元素赋值给新序列的第一个元素；其次，把源序列的第 2 个元素减去第 1 个元素，并将其结果赋值给新序列的第 2 个元素；再次，把源序列的第 3 个元素减去第 2 个元素，并将结果赋值给新序列的第 3 个元素；以此类推，直至计算完源序列的全部元素为止。同样，adjacent_difference () 算法也提供了带有谓词的形式。

`adjacent_difference()` 算法的原型为:

```
template <class InputIterator, class OutIterator > OutputIterator adjacent_difference (InputIterator
 _First, InputIterator
 _Last, OutputIterator _Result );
template <class InputIterator, class OutIterator, class BinaryOperation > OutputIterator adja
 cent_difference (InputIterator _First, InputIterator _Last, OutputIterator _Result, BinaryOperati
 on _Binary_op );
```

其中, 第一种形式用于计算源区间[`_First`, `_Last`]中元素的相邻差, 并将计算结果保存至以`_Result`为起始的新序列中; 第二种形式用于针对源区间[`_First`, `_Last`]中的每个元素和其紧邻的前一个元素调用谓词处理, 并将计算结果写入以`_Result`为起始的目标序列中。源序列的第一个元素只是被单纯地复制。

需要注意的问题如下:

- 1) 该算法的返回值为目标区间内“最后一个被写入的值”的下一位置。
- 2) 该算法的第一种形式相当于将一个绝对值序列转换为一个相对值序列。
- 3) 源区间和目标区间可以相同。
- 4) 程序员需要确保目标区间足够大, 否则需要使用插入型迭代器。
- 5) 谓词函数`_Binary_op`不能修改传入其内的参数。

由内容可知, 对于序列 `a, b, c, d, ...`, 使用 `adjacent_difference()` 算法将产生新序列: `a, b - a, c - b, d - c, ...`。



提示 `partial_sum()` 算法和 `adjacent_difference()` 算法的效果是完全相对的, 两种算法之间是逆向操作的关系。

例 6-13

```
#include <vector>
#include <list>
#include <numeric>
#include <functional>
#include <iostream>
#include <algorithm>
using namespace std;
void myprint (int& ele)
{ cout << ele << ", ";
}
int op (int ele2, int ele1)
{ return ele2 * 5 - ele1 * 3;
}
void main ()
{ int dim[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
vector<int> v1;
v1.assign (dim, dim + 12);
for_each (v1.begin (), v1.end (), myprint);
cout << endl;
```

```

vector<int> v2(v1.size());
vector<int>::iterator it;
it = adjacent_difference(v1.begin(), v1.end(), v2.begin());
cout << "v1 is dealt by the algorithm adjacent_difference : " << endl;
for_each(v2.begin(), v2.end(), myprint);
cout << endl;
adjacent_difference(v1.begin(), v1.end(), v2.begin(), op);
cout << "v1 is dealt by the algorithm adjacent_difference with pred op: " << endl;
for_each(v2.begin(), v2.end(), myprint);
cout << endl;
}

```

例 6-13 的执行效果如图 6-13 所示。

图 6-13 例 6-13 的执行效果



总结 请读者认真阅读例题，对照前面讲述的内容，体会 adjacent_difference() 算法的执行效果。

6.4 全局性数学函数

头文件 `<cmath>` 和 `<cstdlib>` 提供了从 C 语言继承下来的全局性数学函数。这些函数包括 `pow()`、`exp()`、`sqrt()`、`log()`、`log10()`、`log()`、`sin()`、`cos()`、`tan()`、`sinh()`、`cosh()`、`tanh()`、`asin()`、`acos()`、`atan()`、`atan2()`、`ceil()`、`floor()`、`fabs()`、`fmod()`、`frexp()`、`ldexp()` 和 `modf()`。

在 C 语言中，这些函数均在头文件 `<math.h>` 和 `<stdlib.h>` 中声明。在 C++ 中，这些函数均改在头文件 `<cmath>` 和 `<cstdlib>` 中声明。

和 C 不同的是，C++ 将某些操作函数针对不同型别进行了重载，从而导致部分 C 函数失去了价值。例如 C 语言提供的 `abs()`、`labs()` 和 `fabs()`，分别计算 `int`、`long`、`double` 类型变量或常数的绝对值；而 C++ 提供的 `abs()` 完成了重载，具备了上述 3 个函数的全部功能。

头文件 `<cstdlib>` 内定义的数学函数包括 `abs()`、`labs()`、`div()`、`ldiv()`、`srand()`、`rand()`。确切地说，所有浮点数的数学函数均可对 `float`、`double`、`long double` 3 个型别实现重载，但同时也带来了新的问题，当传入的数值类型为整数时，会引发模棱两可的情况。例如，

```
sqrt(7); //不合理
```

应该是：

```
sqrt(7.0);
```

或

```
float x=7;    //OK  
sqrt(x);     //OK
```

程序员在处理这一问题时，还没有做到尽善尽美。有些提供了重载，有些遵循标准规格，有些针对所有数值类型加以重载，有些借助预处理器在各种策略间切换。这使得在应用程序的开发过程中，模棱两可的情况时有发生。可移植性高的代码，要求参数型别有精确的匹配。

下面给出上述数学函数的声明形式：

```
float abs (float)  
float acos (float)  
float asin (float)  
float atan (float)  
float atan2 (float)  
float ceil (float)  
float cos (float)  
float cosh (float)  
float exp (float)  
float fabs (float)  
float floor (float)  
float fmod (float)  
float frexp (float)  
float ldexp (float)  
float log (float)  
float log10 (float)  
float modf (float)  
float pow (float, float* )  
float pow (float, float)  
float sin (float)  
float sinh (float)  
float sqrt (float)  
float tan (float)  
float tanh (float)  
double abs (double)  
double pow (double, int)  
long double abs (long double)  
long double acos (long double)  
long double asin (long double)  
long double atan (long double)  
long double atan2 (long double)  
long double ceil (long double)  
long double cos (long double)  
long double cosh (long double)  
long double exp (long double)
```

```
long double fabs (long double)
long double floor (long double)
long double fmod (long double, long double)
long double frexp (long double, int* )
long double ldexp (long double, int)
long double log (long double)
long double log10 (long double)
long double modf (long double)
long double pow (long double)
long double pow (long double)
long double sin (long double)
long double sinh (long double)
long double sqrt (long double)
long double tan (long double)
long double tanh (long double)
```



总结 本小节仅简单介绍了部分全局性数学函数，这些函数的使用已经非常普遍。希望读者在学习和今后的程序编写过程中，尽量多使用这些全局性数学函数。

6.5 小结

本章主要有 4 个重点：复数运算、数组（向量）运算、通用数值计算和全局性数学函数。

通用数值计算只有在数学或者和数学紧密相关的场合才会使用。

全局性数学函数的使用极为普遍。无论是在图形编程、工程计算、理论研究，还是在其他更宽泛的范围内，全局性数学函数均是非常重要的。希望读者认真学习并加以使用。

复数运算对于在校的研究生或高年级本科生尤为重要。学习这部分内容有助于读者使用 C++ 以及 STL 进行课题研究。

数组（向量）运算也是数学中的重要内容。在线性代数或矩阵论中均有涉及，学习这部分内容，对读者学习这部分数学课程大有帮助。

读者可通过自己编写程序加深对这部分知识的理解。

第 7 章

输入/输出类模板

每个程序必须包含输入与输出，即使没有输入，也必须包含输出，否则程序员将无法观察程序的执行效果。C++ 使用了很多较为高级的语言特性来实现输入与输出，包括类、派生类、函数重载、虚函数、模板和多重继承等。要真正理解 C++ 是如何实现 I/O 功能的，必须深入了解 C++ 的诸多内容。

IOStream 程序库提供了一系列 I/O classes。用于 I/O（输入/输出）的各个类（classes），是 C++ STL 的重要组成部分。I/O classes 不仅局限于文件、屏幕或键盘，而且形成了一套富有弹性的框架，可用于任意数据的格式化及处理（存取）任意外部表述。为满足 STL 的一致性和某些新的需求，I/O classes 做了多次修改，但其基本原则始终如一。最近几年，输入与输出（I/O）被国际标准化，并且 IOStream classes 均也被泛型化（或标准化），可支持不同的字符表述。在使用过程中，需要包含头文件 `<iosfwd>`。在 STL 中，和其余部分相同，IOStream 的所有符号均定义于命名空间 `std` 内。

`std::IOStream` 类库从标准输入中读取输入，并在标准输出上显示结果。而 IOStream 类库具有的能力远不止读写系统控制台，还包括更复杂的数据格式化和文件 I/O 流等内容。流被应用于在不同的机器之间传递对象，用于加密消息流、数据压缩和对象的持续性存储，甚至包括大量其他工作。

本章将介绍使用 `istream` 类对象 `cin` 和 `ostream` 类对象 `cout` 实现输入和输出的基本方法、使用 `ifstream` 和 `ofstream` 对象实现文件输入和输出的基本方法，还将介绍 C++ 的输入类和输出类。使用文件输入和输出的 C++ 工具均是基于 `cin` 和 `cout` 的基本类定义。

7.1 IOStream 简介

IOStream 类具有处理格式化显示的技术。本节介绍 `stream` 类对象、`stream` 类别以及 `stream` 类的操作符。

7.1.1 `stream` 对象

1. 输入流和输出流对象

C++ I/O 由 `stream` 完成。所谓 `stream`，即一条数据“流”，字符序列在其中“川流不息”。按面向对象原则，`stream` 是由某个类别定义的具有特定性质的对象。C++ 流被实现为类模板。`std::cout` 和 `std::cin` 对象是 `std::ios` 的派生类的全局实例对象，尽管后面将使用 `std::ios` 定义的常量和函数——包括 `std::ios::beg` 和 `std::ios::setprecision(int)`。程序主要处理的对象类型均派生自 `std::ios` 类。这些类对象调用的成员函数均是在 `ios` 类中定义的。

stream 主要包括输出流和输入流。输出操作被解读为“数据流入 stream”，输入操作则被解读为“数据流出 stream”。stream 还包括一些为标准 I/O 通道而定义的全局对象。

C++ 的 iostream 类库管理了诸多细节。例如，若程序中包含 iostream 文件，则将自动创建 8 个 stream 对象（4 个用于窄字符，4 个用于宽字符）。

1) cin 对象与标准输入 stream 相对应。默认情况下，stream 被关联到标准输入设备（键盘）。Wcin 对象与此类似，处理的是 wchar_t 类型。

2) cout 对象与标准输出 stream 相对应。默认情况下，这个 stream 被关联到标准输出设备（显示器）。wcout 对象与此类似，处理的是 wchar_t 类型。

3) cerr 对象与标准错误 stream 相对应，可用于显示错误信息。默认情况下，此 stream 被关联到标准输出设备（显示器）。这个 stream 没有缓冲，意味着信息将被直接发送至屏幕，不会等到缓冲区填满或新的换行符。wcerr 对象与此类似，处理的是 wchar_t 类型。

4) clog 对象同样对应标准错误 stream。默认情况下，这个 stream 被关联到标准输出设备（通常为显示器）。但此 stream 有缓冲。wclog 对象与此类似，但处理的是 wchar_t 类型。

当 iostream 文件为程序声明一个 cout 对象时，该对象将包含存储了与输出有关信息的数据成员，如显示数据时使用的字段宽度、小数位数、显示整数时采用的计数方法以及描述用来处理输出流的缓冲区的 streambuf 对象的地址。cout 对象凭借 streambuf 对象的帮助管理流中的字节流。

2. 文件流对象

文件流对象是用于文件操作过程中。程序需要实例化文件流类的对象，文件流类包括 ofstream、ifstream 和 fstream。这 3 个类分别支持文件输入、输出以及兼具输入和输出。程序需要在流类的构造函数中包括实参，或者实例化没有初始化的文件流对象。

7.1.2 stream 类别

对于不同种类的 I/O（输入、输出、文件存取），其中最重要的是：

- class istream，该类用于定义输入流，可用来读取数据。
- class ostream，该类用于定义输出流，可用来写出数据。
- class ofstream，该类用于既可以实现文件输入，也可以实现文件输出。

1. istream 类和 ostream 类

两者的具体实现为 template class basic_istream < > 和 basic_ostream < >，以 char 作为字符型别。事实上整个 IOStream 程序库均不依赖任何特定的字符型别，而以一个 template 参数替代之。此种参数化在 string 类中也存在。

IOStream 程序库定义了整个型别为 istream 和 ostream 的全局对象，与标准的 I/O 通道相对应。前面已讲过：

cin 是供使用者输入用的标准输入通道，对应于 cstdin。操作系统通常将它和键盘连接。

cout 是供使用者输出用的标准输出通道，对应于 cstdout。操作系统通常将它和屏幕连接。

cerr 是所有错误信息所使用的标准错误输出通道，对应于 cstderr。操作系统通常将它和监视器连接。缺省情况下，cerr 无缓冲装置。

`clog` 是标准日志通道, 没有对应物。默认情况下, 操作系统将它连接入 `cerr` 所连接的装置。`clog` 设有缓冲装置。

“正常输出”和“错误信息的输出”需要分离, 需要让程序以不同的方式对待两种输出。例如, 可以将正常输出重新定向至某个文件, 而同时仍然令错误信息显示于控制台。前提是操作系统必须支持标准 I/O 通道的重定向功能。这种分离方式起源于 UNIX 的 I/O 重定向概念。

`ostream` 类派生自 `std::ios` 类, 用来处理流的输出。`std::cout` 对象是 `std::ostream` 类的外部对象。`cout` 对象在库中声明, `cout` 的 `extern` 声明出现在 `<iostream>` 中, 因此凡是包含头文件 `<iostream>` 的程序均可使用 `cout` 对象。程序通过使用重载 “`<<`” 插入符来写 `std::ostream` 对象, `std::ostream` 类具有的 “`<<`” 插入运算符足以支持把大多数标准 C++ 数据类型写到输出流中。

`std::istream` 类管理流输入的方式和 `std::ostream` 类管理流输出的方式相同。在头文件 `<iostream>` 中声明, `std::cin` 对象从标准输入设备中读取数据。`std::istream` 类使用重载 “`>>`” 抽取运算符来读取输入。重载 “`>>`” 抽取运算符足以支持读取标准 C++ 数据类型的数据。用户自定义的类可以通过重载 “`>>`” 运算符, 实现从 `std::istream` 对象中读取数据。

2. ofstream 类

C++ I/O 类软件包处理文件输入和输出的方式与处理标准输入和输出的方式非常相似。要写入文件, 需要创建一个 `ofstream` 对象, 并使用 `ostream` 方法。例如 “`<<`” 插入符或 `write()`。当读取文件时, 需要创建一个 `ifstream` 对象, 并使用 `istream` 方法。例如 “`>>`” 抽取操作符或 `get()`。文件管理是非常复杂的, 在使用 `ofstream` 时必须将文件和流关联, 并且还分多种操作模式 (只读、只写和读写); 写文件时, 还需要创建新文件、取代旧文件或添加到旧文件中, 还可能在文件中来回移动文件指针。为实现对文件的复杂操作, STL 增加了用于文件输入类 (`ifstream`)、用于文件输出的类 (`ofstream`) 以及用于同步文件输入/输出类 (`fstream`)。

7.1.3 stream 操作符

“`operator >>`” 和 “`operator <<`” 被相应的 `stream classes` 重载, 分别用于输入和输出。C++ 移位操作符变成了 I/O 操作符。例如,

```
int a,b;
std::cin >>a >> b;           //输入整型变量 a, b 的值
std::cout <<"a: " <<a <<"b: " <<b << std::endl; //输出整型变量 a,b 的值
```

7.1.4 操控器

大部分输出语句的最后会写一个称为操控器的东西, 即 `std::endl`。操控器是专门用来操控 `stream` 的对象, 但改变了输入或格式化输出的方式, 例如数值进制 `dec` (10 进位)、`hex` (16 进位) 和 `oct` (8 进位)。用于 `ostream` 的操控器并不凭空输出数据, 同样用于 `istream` 的操控器不会忽略输入数据。部分操控器会发生即时操作, 例如, 用于 “刷新 `output` 缓冲区” 或 “跳过 `input` 缓冲区空格” 的那些操控符。

操控器 `endl` 的功能是终止一行代码, 主要用于完成两项任务。

1) 输出换行符号 '\ n'。

2) 刷新 output 缓冲区。

IOStream 程序库中最重要的一些操控器包括 endl、ends、flush 和 ws。

1) endl 属于 ostream 类，输出 '\ n'，并刷新 output 缓冲区。

2) ends 属于 ostream 类，输出 '\ 0'。

3) flush 属于 ostream 类，刷新 output 缓冲区。

4) ws 属于 istream 类，读入并忽略空格。



总结 本节介绍和 IOStream 类有关的基本概念：stream 对象、stream 类别、stream 操作符、操控器等。对于上述内容，读者应认真阅读，并熟练掌握这些概念，便于后续章节的学习。

7.2 IOStream 基本类和标准 IOStream 对象

本小节主要简述和 IOStream 类相关的头文件、标准 stream 操作符、stream 状态以及标准输入和输出函数。

7.2.1 头文件

关于 stream classes 的诸多定义包含在头文件 < iosfwd >、< streambuf >、< istream >、< ostream > 和 < iostream > 中。大部分头文件主要用于 C++ STL 的内部组织。IOStream 程序使用者只需包含拥有各个 stream classes 声明式的头文件 < iosfwd > 即可。在运用输入或输出功能时，需要分别包含头文件 < istream > 或头文件 < ostream >。如果不需要使用标准 stream 对象，那么不需要包含头文件 < iostream >。

对于特殊的 stream 特性，例如参数化的操控器、file streams 以及 string stream，需要包含其他头文件。

7.2.2 标准 stream 操作符

C/C++ 的运算符 “>>” 和 “<<”，分别用于“位左移”和“位右移”。类 basic_istream < > 和类 basic_ostream < > 重载了它们，使其成为标准的 I/O 操作符。本节主要介绍 output 操作符、input 操作符、特殊型别的 IO、操作符的格式化等内容。

1. output 操作符

output 操作符的声明形式有以下 4 种：

```
basic_ostream<charT, traits>& operator << (basic_ostream<charT, traits>& (* pf) (basic_ostream<charT, traits>&))
basic_ostream<charT, traits>& operator << (basic_ios<charT, traits>& (* pf) (basic_ios<charT, traits>&))
basic_ostream<charT, traits>& operator << (ios_base& (* pf) (ios_base&))
basic_ostream<charT, traits>& operator << basic_streambuf<charT, traits>* sb);
```

basic_ostream (ostream 和 wostream 均可) 将 “<<” 定义为 output 操作符，并对所有基

本数据类型型别均进行了重载, 包括 `char*`、`void*`、`bool` 等。stream 将其 output 操作符定义为: 把第二参数发送到相应的 stream 中。操作符 “<<” 会被重载, 第二参数可以是任意型别。C++ STL 使用相同的机制提供 `string`、`bitset` 及 `complex` 等类型的 output 操作符。前面章节已经大量使用了 “<<” 符号用于输出, 此处不再赘述。输出机制的可扩展性使程序员的自定义型别能够天衣无缝地融入输入/输出系统中, “<<” 和 C 语言的 `printf()` 输出机制相比, 是跨越式进步。程序员在输出数据时, 不仅仅局限于标准型别, 还包括自定义型别。

操作符 “<<” 不仅可以输出单个对象, 还可以输出多个对象, 即使用 “<<” 可以将需要输出的数据编写成一串, 由 “<<” 运算符按由左到右的次序依次输出。

2. input 操作符

input 操作符的声明形式如下:

```
basic_istream<charT,traits>& operator >> (basic_istream<charT,traits>& (* pf) (basic_istream<charT,traits>&))
basic_istream<charT,traits>&operator >> (basic_ios<charT,traits>& (* pf) (basic_ios<charT,traits>&));
basic_istream<charT,traits>& operator >> (ios_base& (* pf) (ios_base&))
```

`basic_istream` 将 “>>” 定义为 input 操作符。和 `basic_ostream` 类似, `basic_istream` 对基本的数据类型型别 (`char*`、`void*` 和 `bool`) 重载了操作符 “>>”。stream input 操作符被定义为: 将读入的数值存储在第二参数。参数的传递方向是箭头方向。

和 “<<” 同样, 程序员可以对任意型别重载 input 操作符, 并串联使用它们。

3. 特别型别 I/O

标准 I/O 操作符还定义了 `bool`、`char*` 和 `void*` 型别的输入和输出。“>>” 和 “<<” 还可以运用于自定义类型对象的输出和输入。

默认情况下, 布尔值的读取和输出均以数字的形式出现。在 C 语言中, `true` 代表 “真” 或 “1”, `false` 代表 “假” 或 “0”。如果读入的数值既非 1 也非 0, 即被认为是错误的。此时 `ios` 类可能会抛出相应的异常。

最重要的是设立 stream 的格式化选项, 以字符串形式对布尔量进行 IO 操作。这目前仍然是国际化的议题, 多数情况下采用字符串 “`true`” 和 “`false`”。并且在不同的国家, 不同的语系, 存在不同的形式。特殊的 locale objects 会使用相应的字符串。

对于 `char` 和 `wchar_t`, 经由操作符 “>>” 读取一个 `char` 或 `wchar_t` 字符时, 通常开头的空格会被去掉, 如果需要保留读入的所有字符 (包括空格), 需要清除 `skipws` 标志或利用成员函数 `get()`。

对于 `char*` 型的 C 字符串, 在读入过程中, 起始的空格会被跳过, 一直到非空格或文件结束为止。如果需要保留起始的空格, 可由标志 `skipws` 控制。以上阐述说明, 使用 C 语言读入的字符串最大长度远超过 80 个字符。因此, C++ 语言的 `string` 型对象 (或容器) 可容纳足够的字符。使用 `string` 代替 `char*` 可使程序更轻松、更安全; 另外, `string` 还提供更便捷的函数 (`getline()`), 用以实现逐行读入数据。程序员应尽量使用 `string`, 以提高编程效率。

对于 `void*`, 操作符 “<<” 和 “>>” 为指针的打印和读入提供了方便。当数据类型型别为 `void*` 的参数被传递至 output 操作符 (“<<”) 时, 其地址将被输出, 而不是输出指针指向的内容。

例 7-1

```
#include <iostream>
using namespace std;
void main()
{
    char* cstring = "Hello";          //输出地址时需要使用 void*
    cout << "string \" \" << \" \" << cstring << \" \" << "is located at address: " <<
        static_cast <void* > (cstring) << endl;
    cout << "string \" \" << \" \" << cstring << \" \" << "is located at address: " <<
        (void* )cstring << endl;
}
```

例 7-1 输出结果为:

```
string "Hello " is located at address: 0046E04C
string "Hello " is located at address: 0046E04C
```

借助 input 操作符（“>>”）读入地址，需要注意的是，地址是临时性的，同一对象的地址在程序每次执行时可能会不同。地址的输出和输入的可能应用：交换对象地址或共享内存。

对于 stream Buffers（串流缓冲区），操作符“>>”和“<<”可以直接用于读写或改写串流缓冲区，运用 C++ 类 IOStream 实现文件复制的最快途径。

自定义型别在使用时是比较方便的。但由于需要考虑所有可能的格式和错误条件，因此需要付出更多努力。对于自定义型别扩展标准 I/O 机制，后面会有更复杂的讨论。

4. 操作符的格式化

前面的章节已经涉及了在输出时需要使用 cout 的内容。使用 cout 时，可以使用很多种格式显示数字。C++ 的 IOStream 库使用操作符、标志和成员函数支持类似的格式化显示。对于简单的数值列表显示，cout 对象会工作得非常好。还可以使用标准 IOStream 系统所提供的格式化，以多种方式显示输出。

5. 带参数的操作符

为便于程序员可以不直接以标志位的方式处理流的状态，STL 还提供了部分函数。这些函数用于操控类似状态，其实质是在读写对象之间插入一个修改状态的操作。

例如，flush 函数可以显式地刷新输出缓冲区。

```
cout << x << flush << y << flush;
```

例如，noskipws 操控符可以在输入时不省略前面的空格。

```
cin >> noskipws >> x;          //保留空格
```

在上述语句中，noskipws 可以代替 cin.unsetf (ios_base::skipws) 函数。

下面主要讲述带参数的操控符。

带参数的操控符非常有用。例如 cout.precision (int np)。该函数的执行结果是使用 4 位精度输出随后输出的变量或其他类型对象。

```
double d = 20.123456;
cout.precision(4);          //精度保留 4 位
cout << d << endl;          //输出 d
```

上述代码的输出结果为:

20.12

6. 标准 I/O 操作符

STL 针对各种各样的格式状态和状态变化提供了一批操控符。标准 I/O 操控符定义在名称空间 `std` 中。针对 `ios_base` 的操控符由 `<ios>` 给出。针对 `istream` 和 `ostream` 的操控符分别在 `<istream>` 和 `<ostream>` 中给出,并在 `<iostream>` 中也同时给出。其他更复杂的标准操控符则由头文件 `<iomanip>` 给出。

例 7-2

```
boolalpha()           //用符号形式表示真和假
nboolalpha()         //unsetf(ios_base::boolalpha)
showbase()           //输出八进制和十六进制时,分别加各自的前缀
noshowbase()         //unsetf(ios_base::showbase)
showpoint()          //unsetf(ios_base::showpoint)
noshowpoint()        //unsetf(ios_base::showpoint)
showpos()            //unsetf(ios_base::showpos)
noshowpos()          //unsetf(ios_base::showpos)
skipws()             //跳过字符串起始的空格
noskipws()           //unsetf(ios_base::skipws)保留字符串起始的空格
uppercase()          //使用 X 和 E 而不是 x 和 e
nouppercase()        //使用 x 和 e,而不是 X 和 E
internal()           //调整
left()               //值后填充
right()              //值前填充
dec()                //整数基数
hex()                //整数基数 16
oct()                //整数基数 8
fixed()              //浮点格式
scientific()         //科学格式
endl()              //按 <Enter> 键换行,并刷新
ends()              //输出 '\0'
flush()             //刷新流
ws()                //去掉“空白”字符
resetiosflags()     //清楚标志
setiosflags()       //设置标志
setbase()           //基于 b 输出整数
setfill()           //用 c 作为填充字符
setprecision()      //n 位数字
setw()              //下个域宽为 n 个字符
```

如果使用带参数的操控符,需要加括号。如果要使用带参数的标准操控符,应包含头文件 `<iomanip>`。

例如,

```
double d1 = 30.3245;
cout << setprecision(4) << d << endl;           //设置输出精度为 4 位
```

上述代码的输出结果为：

30.32

7. 用户自定义操作符

程序员按照标准的风格使用各自的操控符。precision 对所有输出操作具有持续性的作用，而 width() 对下一个数值操作起作用。程序员往往希望按某种预定格式输出浮点数时，事情变得非常简单，而又不影响在这个流上随后的输出操作。最基本的想法是定义一个表示格式的类，用另一个类表示一个格式再加一个需要格式化的值，之后让运算符“<<”按照格式向 ostream 输出该值。例如，

```
#include <iostream>
#include <iomanip>
using namespace std;
void Of(double x)
{
    cout.precision(6);
    cout << cout.scientific << x << endl;           //使用科学计数法形式输出
}
void main()
{
    ...
    double d2 = 12345678.987634568;
    Of(d2);
    ...
}
```

程序员可以通过编写自定义的格式化输出函数，实现将数据按各自需要的格式输出。

7.2.3 stream 状态

stream 维持一种状态，用于标志 IO 是否成功，并能指出不成功的原因。每个流都包含相关联的状态。若状态是 good，则说明操作是成功的；此时下一个输入操作可能成功，否则会失败。在读取变量过程中，操作状态如果不是 good，该数据流不会产生任何作用。流状态为 fail 或 bad 之间存在着微妙的差异：若状态为 fail，则可以假设流未被破坏，并且没有字符丢失；若状态为 bad，就意味着彻底失败。

1. stream 状态常数

stream 定义了一些型别为 iostate 的常数，用以反映 stream 的状态。iostate 是类 ios_base 的成员，其具体型别由实例版本决定，即 iostate 未被限定是列举型别和整数型别。iostate 型别的常量主要包括

goodbit、eofbit、failbit 和 badbit。goodbit 代表一切都好；eofbit 代表文件末尾标志；failbit 代表错误，即某个 IO 操作未成功；badbit 代表毁灭性错误，不确定状态。failbit 和 badbit 相比，后者是更严重的错误。二者的区别如下。

failbit：某项操作未能完成，但 stream 大体处于好的状态，即设立这个位；通常是由于读入格式错误，例如，程序需要读入整数，却遇到字符。

badbit: 若 stream 由于不明原因而损坏或丢失数据, 即设立这个位。例如, 某个 stream 被定为指向“文件起点”的更前方。

eofbit 经常和 failbit 同时出现, 在 eof 之后试图读取数据时, 会检测出 end-of-file 状态。在最后一个字符被读取时, eofbit 并未出现, 再次试图读取字符时, 往往 eofbit 和 failbit 同时被设置, 因为读取操作失败了。

上述常数并非全局性, 而仅定义于类 ios_base 中, 需要加作用域操作符。例如,

```
std::ios_base::eofbit
```

使用 ios_base 类的派生类也是可以的。以上标识能够在 basic_istream 和 basic_ostream 的所有对象中使用。stream buffer 没有状态标识, 单个 stream buffer 可被多个 stream object 共享。

上述标识仅能反映最后一次操作的 stream 状态, 主要原因是这些标识可能被以前的某操作设置。

2. stream 状态相关成员函数

stream 当前状态的一些标志可以由一些成员函数来访问。这些成员函数主要包括 good()、eof()、fail()、bad()、rdstate()、clear()、clear (state) 和 setstate (state)。下面一一介绍。

good(): 若 stream 正常无误, 返回 true。表示 goodbit 已设置。该函数的返回值为 bool 类型, 当 basic_ios 的 rdstate 等于 goodbit 时, 返回值为 1; 否则, 返回 0。

bad(): 发生毁灭性错误时, 返回 true。此时 badbit 被设置。该函数的返回值为 bool 类型, 若 basic_ios 的 rdstate&badbit 非零, 返回值为 1; 否则返回 0。

例如,

```
#include <iostream>
void main( void )
{
    using namespace std;
    bool b = cout.bad();           //返回 badbit 的值
    cout << b << endl;
    b = cout.good();              //返回 goodbit 的值
    cout << b << endl;
}
```

上述代码的输出结果为:

0
1

eof(): 若程序在读取文件时, 遇到流末尾 end-of-file 时, 返回 true。表示 eofbit 已设置。

fail(): 若发生错误, 返回 true。表示 failbit 或 badbit 被设置为 1。

例如,

```
fstream fs;
int n=1;
fs.open(".\\eof.txt");           //打开文件
cout << fs.eof() << endl;        //是否处于文件末尾
fs >> n;                          //输入整型变量 n
cout << fs.eof() << endl;        //是否处于文件末尾
```

上述代码的输出结果为：

0
1

`rdstate()`：返回当前已设置的所有标志；其返回值为 `iosstate` 类型。

`clear()`：清除当前所有标志；其返回值为空。

`clear (state)`：清除所有标志后，设置标记 `state`；其返回值为空。

`setstate (state)`：设置 `state` 标志。其返回值为空。其实质是调用 `clear (_State | rdstate)` 函数。参数 `state` 代表需要设置的标志位。

需要说明的是，被设置的位仅能反映过去曾发生的事情，如果某次操作后，发现某个或若干位被设置，无法确定究竟是该次或先前操作导致的该结果。如果想通过标志了解或获取错误信息，在操作前应先设置 `goodbit()`。对于每种标志的访问，必须使用正确的访问方法。

例如，

```
fstream fx;
fx.open(".", "\\test.txt", ios::out);           //打开文件
fx.clear();                                     //清除所有标志位
cout << "badbit: " << (fx.rdstate() & ios::badbit) << " failbit: " << (fx.rdstate() & ios::failbit) <
< " eofbit: "
    << (fx.rdstate() & ios::eofbit) << " badbit: " << (fx.rdstate() & ios::badbit) << endl;
    //显示各标志位
fx.clear( ios::badbit | ios::failbit | ios::eofbit ); //清除标志位
cout << "badbit: " << (fx.rdstate() & ios::badbit) << " failbit: " << (fx.rdstate() & ios::failbit) <
< " eofbit: "
    << (fx.rdstate() & ios::eofbit) << " badbit: " << (fx.rdstate() & ios::badbit) << endl;
    //显示各标志位
fx.close();
```

上述代码的输出结果为：

```
badbit: 0 failbit: 0 eofbit: 0 badbit: 0
badbit: 4 failbit: 2 eofbit: 1 badbit: 4
```

3. stream 状态和异常

(1) stream 状态

`stream` 定义了两个用于 `bool` 表达式的 `operator void* ()` 和 `! ()` 函数。C++ 的异常处理机制可以用于处理错误和异常。多数情况下，`stream` 不会抛出异常。标准化之后的 `stream` 允许对任何状态标识进行定义，此状态标识被设置时，会引发异常。异常处理机制包含了 `exception()` 函数。下面主要介绍上述 3 个函数的使用方法。

`operator void* ()` 和 `! ()` 函数，并不是显式地调用该函数，而是隐式地调用。例如，

```
while(std::cin)
{
...
}
```

结构中的 `bool` 条件并不一定要转化为 `bool`，只要能够转化成某个整数型别或指针型别即可。转换为 `void*` 是为了在同一表达式中读入对象并测试是否成功。例如，

```
If (std::cin >> x)
{
}
```

此时的 `cin` 用于条件判断，`cin` 会调用 `operator void*`，返回“stream 是否发生错误”。

通常，在 `while` 循环中，有时使用 `eof` 作为循环终止的条件，此时一般 `failbit` 和 `badbit` 均被设置。使用“>>”操作符时，默认情况下起头空格会被跳过。但如果从流中输入的是型别是 `char`，空格是有意义的。流 `stream` 的成员 `put()` 函数和 `get()`，可以实现 IO 过滤，或者使用流缓冲区迭代器 `streambuf_iterator` 也可实现 IO filter。

使用 `operator!` 进行反相测试，会返回“stream 是否发生错误”，此时标志位 `failbit` 和 `badbit` 被设置为 `true`。

```
If (! std::cin >> x)
{
}
```

注意：上述 `cin` 是 `stream` 的对象，而在使用“!`cin`”时，是描述 `cin` 状态的布尔值。

(2) 处理异常

调用 `exception()` 函数，可获取目前的异常标识。调用带参数的 `exception()` 时，如果参数代表的标志被设置，也会引发相应异常。`exception()` 函数的原型为：

```
exceptions (flags)           //设定“会引发异常”的标志
exception ()                 //返回引发异常的标志
```

对于该函数的第二种形式，如果其返回值是 `goodbit`，表示没有任何异常被抛出；该函数的第一种形式是设置相应的标志位，并引发相应异常。若输入的参数是 `goodbit` 或者是 0，则不会引发异常。

抛出异常是“程序调用 `clear()` 或 `setstate()` 之后”，设立某标志，如果该标志已被设置并且未被清除，同样也会抛出异常。抛出的异常是 `std::ios_base::failure` 对象，该对象是派生自类 `exception`。

需要说明的是，获取错误信息的唯一可移植方式是借助 `what()` 函数。

`what()` 函数的操作具有移植性。但 `what()` 的返回值不具有移植性，需要根据实际情况判断函数的返回结果。

`stream` 的异常处理能力主要是在读取“格式化数据”时尽显身手，但在使用异常过程中仍然存在诸多问题。当读取数据至文件末尾时，会产生因 `end-of-file` 导致的异常。通过检查流状态可以分清异常的具体原因——是数据错误还是达到文件末尾。

下面以例 7-3 来说明 `exception()` 函数的使用方法。

例 7-3

```
#include <exception >
#include <iostream >
#include <istream >
#include <ostream >
using namespace std;
```

```
void main ()
{ ios::iostate olde = cin.exceptions();           //返回 cin 的异常标志位
  cout << "old exceptions: " << olde << endl;     //输出该标志位
  int x=0;
  try{
    cin.exceptions(ios::eofbit |ios::failbit |ios::badbit); //设置引发异常的标志位
    //cin.exceptions(ios::goodbit);
    cin >> x;
  }
  catch(exception& e)
  {   cout << "exception: " << e.what() << endl;   //输出异常类别
  }
}
```

执行上述程序时，读者应输入一个整型数据（例如一个阿拉伯数字）。上述代码的执行结果如下：

```
old exceptions: 0
n
exception: ios::failbit set
```

7.2.4 标准输入和输出函数

能够取代标准流操作符“>>”和“<<”的部分成员函数一般用于读写无格式数据。通常，在读取数据时，不跳过起始空格，对异常的处理方式也不同于格式化 I/O 函数；发生意外（异常）后，badbit 通常会设立。若异常掩码中包含 badbit，则会重新抛出该异常。和格式化函数一样，无格式相关函数会产生一个 sentry（岗哨）对象。本小节主要讲述类 istream 的成员函数和类 ostream 的成员函数。

1. 输入函数

istream 是用于输入的一个流类。此外，类 wistream 和 basic_istream 模板类也适用于输入操作。C++ STL 中用于读取字符序列的成员函数主要包括 get(s, num)、get(s, num, t)、getline(s, num)、getline(s, num, t)、read(s, num) 和 readsome(s, num)。

输入运算符“>>”在使用时，所输入的字符串前面的空格通常会自动跳过。当在屏幕的某行输入字符时，只有那些非空字符才能进入接收字符的变量中。尤其在实现输入数值类型的数据时，如果在非起始位置出现空格字符，输入将停止。此空格之后的文字或字符会被读取到下一个存储流的对象中，并且中间的空格会自动消失。

对于命名空间 std 中的 ios::skipws 标志，是不能对“>>”运算符起作用的。如果要保留输入序列中的空格，需要使用 get() 函数或 getline() 函数。

命名空间 istream 类的成员 read() 函数主要用于实现输入功能，通常会用于文件的读写操作中。输入 read() 函数是和输出 write() 函数对应的。输入的二进制数据是存入缓冲区中的，并且会包括数据中的空格。

get() 函数的原型为：

```
int get();&
istream& get(char* pch, int nCount, char delim = '\n');
```

```
istream& get( unsigned char* puch, int nCount, char delim = '\n' );
istream& get( signed char* psch, int nCount, char delim = '\n' );
istream& get( char& rch );
istream& get( unsigned char& ruch );
istream& get( signed char& rsch );
istream& get( streambuf& rsb, char delim = '\n' );
```

第一种形式返回从流中读入的字符序列；第二种形式返回流中读入的字符序列，读入的数据存储在字符型数组 pch 中，函数结束的条件是：已经输入了 nCount - 1 个字符，遇见回车换行符 '\n'；或者遇见文件尾标志 end-of-file。第三种形式和第二种形式近似，只不过输入的数据流存储在无符号型字符数组 puch 中。第四种形式和第二种形式近似，只不过输入的数据流存储在有符号型字符数组 psch 中。第五种形式表示从输入流中提取单个字符，并存储在 rch 中；第六种形式和第五种形式近似，表示从输入流中提取单个字符，并存储在无符号字符型变量 ruch 中；第七种形式表示从输入流中抽取单个字符，存储在有符号字符变量 rsch 中；第八种形式表示从输入流中抽取字符序列，并存储在 streambuf 型对象中，抽取时遇 '\n' 或遇见文件结束标志 eof 时，抽取过程终止。

getline() 函数的原型为：

```
istream& getline( char* pch, int nCount, char delim = '\n' );
istream& getline( unsigned char* puch, int nCount, char delim = '\n' );
istream& getline( signed char* psch, int nCount, char delim = '\n' );
```

第一种形式实现从输入流中抽取字符序列，并保存在字符型数组 pch 中。抽取过程的终止条件是：已经输入了 nCount - 1 个字符，遇见回车换行符 '\n' 或者遇见文件尾标志 end-of-file。

第二种形式和第三种形式均与第一种形式近似。

read() 函数的原型为：

```
istream& read( char* pch, int nCount );
istream& read( unsigned char* puch, int nCount );
istream& read( signed char* psch, int nCount );
```

以上给出三种型式的 read() 函数原型。第一种形式实现读取 nCount 个字符，并存储在字符串 pch 中，返回数据流，数据流的状态可表明读取工作是否成功。

需要注意：在读入过程中，字符串 pch 中的字符不会由于字符终止符号结束。必须确保字符串 pch 的容量大于 nCount 个字符。读取过程中，如果遇见 eof 标识会出现错误，failbit 和 eofbit 会被设置。

第二种形式、第三种形式和第一种形式近似。

readsome() 函数的原型为：

```
streamsize istream::readsome( char * str, streamsize count );
```

上述函数最多可读入 count 个字符，并存储在字符串 str 中。其返回值是读取的字符个数。str 内的字符串不会自动以字符串终止符号结束。使用时，程序员需要确保字符串 str 中留有足够的空间，能够保存 count 个字符。

和 read() 函数不同，readsome() 函数会读取 stream buffer 内的所有有效字符，遇到文件

末尾标志 eof 时，不会报错，并且不设置标志位 eofbit 和 failbit。

另外输入函数还包括 gcount()、ignore()、peek()、unget()、putback()、tellg() 和 seekg() 函数。

gcount() 函数返回上次“非格式化读取操作”所读入的字符个数。

ignore() 有 3 种形式，所有形式均可实现提取字符，并将这些字符舍弃不用。其函数原型为：

```
basic_istream<charT, traits>& ignore(streamsize n=1, int_type delim
=traits::eof())
```

在使用过程中，程序员可使用如下 3 种形式：

```
ignore();
ignore(streamsize count);
ignore(streamsize count, int delim);
```

peek() 函数用于返回输入序列中下一个被读入的字符，但并不读取该字符，若输入序列中没有数据，则返回 eof。其原型为：

```
int_type peek();
```

unget() 函数将上一次读取的字符重新放回 stream 中，下一次从输入流中读取时，便可以将这些字符读取出来。

tellg() 和 seekg() 函数主要用于文件输入/输出。后面会有所描述。

2. 输出函数

命名空间 std 中包含输出函数，可以实现把字符和存储块写入至输出流对象中，同时也可以选择使用重载“<<”插入运算符把字符和存储块写至输出流对象中。put() 函数把单个字符写至输出流中，还有另外两个 write() 函数和 flush()。

put() 函数的原型为：

```
ostream& ostream::put(char c)
```

put() 函数用于实现将参数 c 写入流中。其返回值的状态可表明写入是否成功。

write() 函数的原型为：

```
ostream& write(const char* pch, int nCount);
ostream& write(const unsigned char* puch, int nCount);
ostream& write(const signed char* psch, int nCount);
```

上述 3 种形式均可将字符串 pch（或 puch，psch）中的 nCount 个字符写入输出流中。

其返回值类型为输出流类型，返回流的状态字可说明该写入操作是否成功。值得注意的是，字符串终止符号不会终止写入操作。且程序员必须确保字符串中至少包含 nCount 个字符，否则会导致不可预期的行为。

flush() 函数的原型为：

```
ostream& ostream::flush();
```

函数 flush() 用于刷新 stream 的缓冲区，将所有缓冲区数据强制写入其所属的设备或 I/O 通道。另外，tellg() 和 seekg() 函数用以改变写入位置，主要与文件 I/O 有关。

例 7-4

```

#include <iostream>
#include <fstream>
using namespace std;
void main()
{ char chdim[7] = {0,0,0,0,0,0,0};
  cout << "int cin.get(): " << endl;
  cin.get(chdim,6, '\n'); //获取输入
  cout.write(chdim,6); //输出
  cout << endl;
  cin.get(); //获取输入
  cout << "cin.get(char ch): " << endl;
  char ch;
  cin.get(ch);
  cout.put(ch); //输出该字符
  cout << endl;
  cin.get();
  cout << "cin.getline(char* ch,nCount, \n)" << endl;
  cin.getline(chdim,6, '\n'); //获取该行数据
  cout.write(chdim,6); //输出该行数据
  cout << endl;
  char dim[20];
  fstream fio; //流
  fio.open(" \\test.txt", ios::in); //打开文件
  fio.read(dim,20); //从文件中获取
  cout.write(dim,20); //输出获取的数据
  fio.close(); //关闭数据流
  cout << endl;
}

```

例 7-4 的执行效果如图 7-1 所示。

```

int cin.get():
asdfg
asdfg
cin.get(char ch) :
k
k
cin.getline(char* ch,nCount, \n)
eweew
eweew
asdfasfhdsfh
dgdfgh

```

图 7-1 例 7-4 的执行效果

例 7-5

```

#include <iostream>
#include <fstream>

```

```
#include <istream>
#include <string>
using namespace std;
void main()
{
    char chardim[30];
    char ch[6];
    fstream fio;
    fio.open(" \\test5.txt",ios::in|ios::out);           //打开文件
    int cnt=10;
    int gcnt=0;
    istream i0=fio.read(chardim,cnt);                   //从流中读取 cnt 个字符
    if(i0.good())
    {
        cout<<"Read success!"<<endl;
    }
    gcnt=fio.gcount();                                   //统计个数
    string buf;
    buf.assign(chardim,10);                              //存入字符串
    cout<<"The input chars from file test5.txt are:"<<buf<<endl;
    cout<<"The count of the last read function:"<<gcnt<<endl;
    istream is=fio.ignore(3,EOF);                        //忽略
    buf.erase(buf.begin(),buf.end());                   //清除
    i0=fio.read(ch,5);                                   //读字符
    if(i0.good())
    {
        cout<<"Read success!"<<endl;
    }
    buf.assign(ch,5);
    cout<<"Secondly The input chars from file test5.txt are:"<<buf<<endl;
    char chc=fio.peek();
    if(chc==EOF)
    {
        cout<<"file eof reach."<<endl;
    }
    else
    {
        cout<<"some char has existed in file."<<endl;
    }
    i0=fio.unget();
    i0=fio.read(ch,5);
    if(i0.good())
    {
        cout<<"Read success!"<<endl;
    }
    buf.assign(ch,5);
}
```

```
        cout << "Thirdly The input chars from file test5. txt are: " << buf << endl;  
        fio. flush();  
        fio. close();  
    }
```

例 7-5 的执行效果如图 7-2 所示。

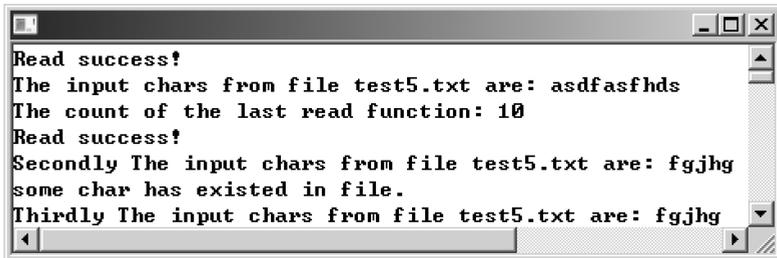


图 7-2 例 7-5 的执行效果

7.3 格式化

前面各节涉及的输出，多数是按照默认规则将对象转为字符序列，但程序员更需要更为细节化的控制。例如，需要控制一个输出操作所用的空格数或数值输出的格式，有时可能更需要以多种格式显示数字。在C++中，IOStream库利用操作符、标志和成员函数支持格式化显示。

对于格式化输入/输出，最重要的是格式标识，可定义诸如数字精度、填充字符、数字进制等。另一个重要概念是对特定国家和地区的不同习惯，实现国际化的格式调整。本节主要介绍格式化的问题，国际格式化的问题在后面章节专门介绍。

7.3.1 格式标识

1. 格式标识

std::ios类定义了部分格式标识。这些标识具有互斥的位值，通过“或”(OR)操作，形成位掩码。setiosflag()函数和函数resetiosflag操作符，成员函数setf()和unsetf()，可接受这些标识和通过“或”(OR操作)形成的掩码做实参。格式标识通常有如下16种：

```
std::ios::boolalpha  
std::ios::hex  
std::ios::internal  
std::ios::left  
std::ios::oct  
std::ios::right  
std::ios::scientific  
std::ios::showbase  
std::ios::showpoint  
std::ios::showpos  
std::ios::skipws
```

```
std::ios::stdio
std::ios::unitbuf
std::ios::uppercase
std::ios::dec
std::ios::fixed
```

2. 格式操控符

类 `ios_base` 提供数个成员函数，可用于定义各种 I/O 格式，进而实现访问格式标志的功能。这些函数包括 `setf()`、`unsetf()`、`flags()`、`flags (flags)` 和 `copyfmt (stream)`。这些函数可以处理所有格式定义。`setf()` 函数和 `unsetf()` 函数用于分别设置和清除一个或多个标识；二元操作符“OR”可将多个标志合并，从而一次操控多个标识。`setf()` 函数以第二参数为掩码，清除该掩码所标识的所有标识，之后设置第一参数所代表的标识。若 `setf()` 输入一个参数，则稍有差异。使用 `flags()` 函数可以一次操控所有格式标志。调用无参数的 `flags()` 会返回当前标识。若传给 `flags()` 一个参数，则以该参数作为新格式标识状态，返回先前状态。`flags()` 函数是非常有用的，还可用于储存当前标识状态，以便在适当时机恢复。

而 `copyfmt()` 函数用于从 `stream` 中复制所有格式定义。

`setiosflags()` 函数和 `resetiosflags()` 函数可以在改写语句或输入语句中设定或清理标识。使用这两个函数需要包含头文件 `<iomanip>`。

下面详细介绍 `setbase()`、`setfill()`、`setprecision()` 和 `setw()` 函数。

`setbase (int base)` 用于实现基于 `base` 输出整数。

`setfill()` 用于设置作为填充字符的字符。

`setprecision()` 用于设置浮点数的精度（即数字的个数或位数）。

`setw()` 用于设置显示字段的宽度。

使用“<<”符号将操作符插入输出流中，也可以使用“>>”运算符插入到输入流中。

典型的格式操作符包括 `std::endl`、`std::ends`、`std::dec`、`std::flush`、`std::hex`、`std::oct` 和 `std::ws`。其含义分别如下：

`std::endl`——在输出流中插入‘\n’，刷新流。

`std::ends`——插入空字符‘\0’。

`std::dec`——按十进制显示整数。

`std::flush`——刷新流。

`std::hex`——按十六进制显示整数。

`std::oct`——按八进制显示整数。

`std::ws`——跳过输入流中的整数。

`std::fixed`——浮点数输出。

3. 格式化函数详解

类 `std::ios` 包含很多控制流格式的函数，通过该类的对象可以调用这些函数。下面介绍部分格式化函数。

`fill()` 用于返回当前填充字符。

`fill (int ch)` 用于设置填充字符，返回先前填充字符。

precision()用于返回当前浮点精度。

precision (int p) 用于设置精度 p, 返回先前的精度。

setf (int m) 用于设置掩码 m 的 OR 表达式中的标识, 返回先前的标识。

setf (int m1, int m2) 用于设置掩码 m1, 关闭掩码 m2。

unsetf (int m) 用于关闭掩码 m 的 OR 表达式中的标识。

width()用于放回当前的宽度。

width (int w) 用于设置宽度为 w, 返回先前的宽度。

7.3.2 bool 类型数据的格式控制

STL 中对于 bool 类型数据的格式控制有明确的使用规定。

标识 boolalpha 定义了布尔值的读写格式: 数字表示或文字表示。

如果 boolalpha 被设置, 输出时便以文字表示; 否则, 输出时以数字表示。使用数字表示时, false 始终是 0, true 始终是 1。在输入时, 如果遇到非“0 和 1”的数值, 程序会报错, 并设置 failbit。

如果此标识被设置, 布尔值会以文字形式表示。读入的字符串必须为“true”或“false”。实际表述法还和 stream locale object 有关系。标准的“C” locale object 使用字符串“true”和“false”表示布尔值。STL 定义了两个操控器: boolalpha() 和 noboolalpha()。boolalpha() 强制使用文字表示法, 并设立标识 ios::boolalpha; noboolalpha() 强制使用数字表示法, 并清除标识 ios::boolalpha。例如,

```
bool b=0;
std::cout << std::noboolalpha << b << " == " << std::boolalpha << b << std::endl;
//输出逻辑变量
```

7.3.3 详解“字段宽度、填充字符和位置调整”

7.3.1 节讲述了各种格式标识和操作符。本小节将详细讲解如何使用格式化中的字段宽度、填充字符和位置调整。

1. 字段宽度

width() 函数用来定义字段宽度。其原型包括两种: 有参数和无参数。

```
int width( int nw );
int width() const;
```

对于输出而言, width() 函数定义了最小字段, 此设置用于下一次格式化输出。调用该函数时, 若无参数, 该函数返回当前字段宽度; 若传入一个整数, 则会改变字段宽度为该整数宽度, 并返回当前宽度。字段宽度的最小默认值是 0。该函数的作用仅限于“下一次”格式化输出。

setw() 函数同样可以为输出的数值设置字段宽度, 其功能相当于 width()。其原型为:

```
setw( int nw );
```

setw() 函数还可以将设置字段宽度命令插入输出流和输入流中, 比单独调用格式化函数更加方便。当需要显示的数值比设置的字段宽度大时, 将显示整个数据值, 这有可能会影响后面的数据输出或者后面输出的数据格式。

值得一提的是，当在输入流中设置字段宽度时，若使用字符数组和字符型指针，尤其是字符型指针，则不能用函数 `sizeof(char*)` 来计算该字符型数组的字符数目。例如，正确：

```
char chdim[81];
cin >> setw(sizeof(chdim)) >> chdim;
```

错误：

```
char* s;
cin >> setw(sizeof(s)) >> s;
```

若使用 `string` 类型，则不会有以上麻烦。例如，

正确：

```
string buf;
cin >> buffer;
```

2. 填充字段

`fill()` 函数定义用来填充“格式化表述”和最小字段之间的填充字符。默认的填充字符是空格符。

```
char fill(char cFill);
char fill() const;
```

`setfill()` 函数同样可以定义填充字符，其作用相当于 `fill()` 函数。例如，

```
setfill(int nFill);
```

再如，

```
cout << setw(8) << setfill('_') << -3.14 << " " << 42 << endl;
```

3. 格式化中的位置

输入和输出中的字段位置一般是指字段的左对齐、右对齐和“符号靠左对齐，数值靠右对齐”。标识 **left** 代表靠左对齐；标识 **right** 代表靠右对齐；标识 **internal** 代表符号靠左对齐，数值靠右对齐；**none** 是默认的靠右对齐。使用字段位置时，如果输出数值或字符所占宽度小于设置的字段宽度，其余位置需要由特定字符填充。

单一字符的对齐方式在标准化过程中发生变化。标准化之前，面对单个字符，忽略字段宽度，直至下一次多字符格式化输出时才使用。在输入和输出时，设置字段位置有两种方法：`setf()` 和 `unsetf()`；或者 `setiosflag()` 和 `resetiosflags()`。

`setf()` 和 `unsetf()` 函数的原型为：

```
long setf(long lFlags);
long setf(long lFlags, long lMask);
long unsetf(long lFlags);
```

或者

```
smanip(long) resetiosflags(long lFlags);
smanip(long) setiosflags(long lFlags);
```

以上函数均包含在头文件 `<iomanip>` 和 `<iostream>` 中。

例 7-6

```

#include <iostream>
#include <iomanip>
using namespace std;
void main()
{
    bool varb=1;
    cout << std::noboolalpha << varb << "==" << std::boolalpha << varb << endl;
    float varf=123.4512;
    int wid=cout.width(); //返回原有的字段宽度
    cout.width(6); //设置输出宽度
    cout << varf << endl;
    cout << setw(7) << varf << endl; //设置输出字段宽度
    cout.width(6);
    cout << cout.fill('_') << varf << endl; //使用字符“_”填充
    cout << setw(9) << setfill('* ') << varf << " " << 45 << endl; //使用字符“* ”填充
    cout << setw(9) << setfill(' ') << setiosflags(ios::right) << varf << "; : " << setw(9) << 45.234
    << endl;
    double varf1=123.4512;
    double varf2=456.892;
    cout << setw(10) << setfill(' ') << setiosflags(ios::right) << varf1 << ";" << resetiosflags
    (ios::left) << setw(10)
    << setfill(' ') << setiosflags(ios::right) << varf2 << endl; //使用空格填充,右对齐输出 varf1,
    varf2
    resetiosflags(ios::left); //恢复左对齐
    cout << varf1 << ";" << varf2 << endl;
    //注意输出没有将格式控制直接插入流输出中,这是因为下述格式控制调用时均有返回值;
    //程序运行时,会将这些函数的返回值输出,带来不必要的麻烦
    cout.width(12);
    cout.fill('* ');
    cout.setf(ios_base::right,ios_base::adjustfield); //右对齐
    cout << varf1 << ";" << endl;
    cout << varf2 << ";" << endl;
    cout.width(12);
    cout.fill('* ');
    cout.setf(ios_base::right,ios_base::adjustfield);
    cout << varf2 << ";" << endl;
    cout.width(12);
    cout.fill('* ');
    cout.setf(ios_base::left,ios_base::adjustfield);
    cout << varf1 << ";" << endl;
    cout.width(12);
    cout.fill('* ');
    cout.setf(ios_base::left,ios_base::adjustfield);
    cout << varf2 << ";" << endl;
}

```

例 7-6 的执行效果如图 7-3 所示。

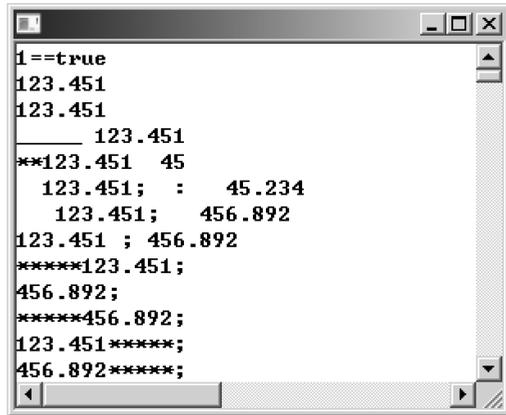


图 7-3 例 7-6 的执行效果

7.3.4 正记号与大写字符

格式标识 `showpos` 和 `uppercase` 用于改变数值的一般表述。这两个标识会影响数值的正负号和字母大小写。标识 `showpos` 使用时会在正数前加上正记号；标识 `uppercase` 使用时会实现使用大写字符的功能，并且还可用于十六进制格式表述的整数以及“科学计数法表述”的浮点数。通常在输出时，字母均为小写，并省略正记号。虽然这是一个小问题，但使用时会便于读者阅读。标识的具体含义见表 7-1。

表 7-1 操作标识符的含义

操作标识符	含 义	操作标识符	含 义
<code>showpos</code>	强制输出正数前的正记号 (<code>ios::showpos</code>)	<code>nouppercase</code>	强制字母小写 (<code>ios::nouppercase</code>)
<code>noshowpos</code>	强制忽略正数前的正记号 (<code>ios::noshowpos</code>)	<code>showbase</code>	显示数值进制 (<code>ios::showbase</code>)
<code>uppercase</code>	强制字母大写 (<code>ios::uppercase</code>)	<code>noshowbase</code>	不显示数值进制 (清除 <code>ios::showbase</code>)

例如，

```

double var1 = 198534.98236;
cout.setf(ios::showpos | ios::uppercase);
cout << scientific << setprecision(4) << setw(16) << setfill(' ') << setiosflags(ios_base::right)
<< var1 << endl;
cout.unsetf(ios::uppercase); //取消大写
cout << scientific << setprecision(4) << setw(16) << setfill(' ') << setiosflags(ios_base::right)
<< var1 << endl;
double var2 = -198534.98236;
cout.setf(ios::showpos | ios::uppercase);
cout << scientific << setprecision(4) << setw(16) << setfill(' ') << setiosflags(ios_base::right)
<< var2 << endl;
cout.unsetf(ios::uppercase); //取消大写

```

```
cout << scientific << setprecision(4) << setw(16) << setfill(' ') << setiosflags( ios_base::right)
<< var2 << endl;
```

上述代码的输出结果为:

```
+1.9853E+005
+1.9853e+005
-1.9853E+005
-1.9853e+005
```



提示 虽然提供了标识符 `ios::noshowpos` 和 `ios::nouppercase`, 但在 Visual C++ 6.0 环境中无法使用 `setf()` 函数设置标识符 `noshowpos` 和标识符 `nouppercase`, 所以在上面的示例中使用了 `unsetf()` 函数。

7.3.5 数值进制

常见的数值进制包括八进制、十进制和十六进制。这 3 种进制分别用 3 种标识: `oct`、`dec` 和 `hex`。标识 `oct` 代表是以八进制形式进行读写; 标识 `dec` 代表是以十进制形式进行读写; 标识 `hex` 代表是以十六进制形式进行读写。若使用参数 `none`, 则代表以十进制形式输出, 在读入时要根据起始字符的实际情况而定。

一旦进制被改变, 除非重新设置相关标识, 否则会持续应用于后继的整数处理过程。默认情况下使用十进制。如果没有使用进制标识或使用了多个进制标识, 输出时会采用十进制。输入流和输出流 (IOStream) 不支持二进制, 使用二进制进行读写时, 可使用类 `bitset`, 实现以二进制形式读写整数值。

在输入时, 进制标识有时也会产生影响。上述的进制标识被设置后, 读取的数字会按该进制处理。如果没有设置进制标识, 起始字符将决定进制。例如, 以 `0x` 或 `0X` 起始的为十六进制数, 以 `0` 起始的为八进制数, 其余被视为十进制数。

标识 `showbase` 会以 C/C++ 惯例处理数值进制。标识 `showbase` 一旦被设置, 在输出数值时, 会自动显示出相应的数字进制。八进制会自动以 `0` 开头, 十六进制会以 `0x` 或 `0X` 开头。整数进制的操作标识符及其意义见表 7-2。

表 7-2 整数进制

操作标识符	意 义	操作标识符	意 义
<code>oct</code>	以八进制进行读写	<code>none</code>	以十进制输出, 读取时视起始字符而定
<code>dec</code>	以十进制进行读写	<code>showbase</code>	显示数值进制 (设置标识 <code>ios::showbase</code>)
<code>hex</code>	以十六进制进行读写	<code>noshowbase</code>	不显示数值进制 (清除标识 <code>ios::showbase</code>)

使用上述数值标识一般有两种方法: 使用 `setf()` 函数直接设置需要指定的标识; 使用 `unsetf()` 函数直接设置需要指定的标识。在设置标识时, 同时自动清理同组的其他标识。`setf()` 函数和 `unsetf()` 函数的原型为:

```
void setf( fmtflags _Mask);
fmtflags setf( fmtflags _Mask,fmtflags _Unset);
```

和

```
void unsetf( fmtflags _Mask);
```

其中 `setf()` 函数的第二种形式的第一个参数 `_Mask` 是需要打开的标识符，第二个参数 `_Unset` 是需要关闭的标识符。该形式的返回值是原有的标识符。例如，

```
#include <iostream>
using namespace std;
void main()
{
    int vari = 123458;
    cout << showbase << uppercase << " vari : " << vari << " vari (oct): " << oct << vari << " vari (hex): " <<
hex << vari << endl;
    cout << endl;
    //以下需要分别调用各函数,而不是将函数插入至输入流和输出流中
    cout.setf( ios::dec | ios::uppercase | ios::showbase );
    cout << " vari : " << vari;
    cout.setf( ios::oct, ios::basefield );
    cout << " vari (oct): " << vari;
    cout.setf( ios::hex, ios::basefield );           //十六进制输出时,x大写
    cout << " vari (hex): " << vari << endl;
}
```

上述代码的输出结果为：

```
vari : 123458 vari (oct): 0361102 vari (hex): 0X1E242
vari : 123458 vari (oct): 0361102 vari (hex): 0X1E242
```

7.3.6 浮点数输出

浮点数的输出通过格式和精度控制。通常有 3 种形式的输出：**一般格式**，让具体实现为输出选一种表现形式，使之能在可用空间内以最佳形式维持这个值，精度表述了数字的最大位数，此种格式对应于 `printf()` 的 `%g` 输出格式；**科学格式**（scientific），通常用小数点前一位数字以及一个指数部分的方式表现数值，精度表述的是小数点之后的最大允许位数，和 `printf()` 的参数 `%e` 相关；**定格式**（fixed），将值表示为一个整数部分，其后面跟小数点及一个小数部分，精度表述的是小数点之后的最大允许位数，相当于 `printf()` 的 `%f` 输出格式。通过状态操控函数来控制浮点数输出的格式。

需要指出的是，精度控制对于浮点数是实现舍入操作，而不是简单截断操作。精度控制 `precision()` 函数不影响整数的输出。

通常涉及的 `stream` 标识以及相关的流（stream）成员函数，可以用来实现浮点数输出的控制。和前面的讲述一致，多种标识分别为 `ios::fixed`、`ios::scientific` 和 `ios::none`。

`ios::fixed`——使用小数计数法。

`ios::scientific`——使用科学计数法。

`ios::none`——使用上述两者中最合适的。

默认情况下，参数 `ios::fixed` 和 `ios::scientific` 未被设置，此时输出的形式取决于实际情

况。还可以通过使用标识符 `ios::showpoint` 来强制书写小数点, 并补 0, 直至足够的精度为止。前面已讲过, 标识符 `ios::showpos` 可用来输出正记号, 标识符 `ios::uppercase` 可用来指定科学计数法中字符 `e` 的大写和小写。

流成员函数 `precision()` 用于定义精度, 并返回当前的浮点数精度。调用函数 `precision(val)` 时, `val` 为新设置的浮点数精度, 并返回原来的设置值。一旦使用科学计数法, `precision()` 可定义小数位数, 其余数是通过四舍五入舍弃, 而不是被截断地舍弃。另外, `setprecision()` 函数也可以设置数值输出的精度, 使用 `setprecision()` 函数时, 需要包含头文件 `<iomanip>`。而且 `setprecision()` 可以直接插入输出流中, 而 `precision()` 却不能直接插入输入流和输出流中。

科学计数法和定点数的输出控制方法也包括两种: ①使用 `setiosflags()` 函数; ②直接将关键字 (`fixed`) 插入输出流 (例 `cout`) 中; ③使用 `setf()` 函数设置浮点类型, 或者将标识符 `scientific` 直接插入输出流中。例如,

```
#include <iostream>
#include <iomanip>
using namespace std;
void main()
{
    double vard = 15.432;
    cout << setprecision(4) << vard << endl;
    cout.precision(6);
    cout << vard << endl;
    cout << " ios::fixed usage: " << endl;           //浮点数输出
    cout << setiosflags(ios::fixed) << vard << endl;
    cout << " ios::scientific: " << endl;
    cout << " fixed usage: " << endl;
    cout << fixed << vard << endl;                 //浮点数输出
    cout << " scientific usage: " << endl;
    cout << scientific << vard << endl;
}
```

上述代码的输出结果为:

```
15.43
15.432
ios::fixed usage:
15.432000
ios::scientific:
fixed usage:
15.432000
scientific usage:
1.543200e+001
```

7.3.7 一般性格式定义

通常的格式标识一般包括 `skipws` 和 `unitbuf`。一旦 `skipws` 格式标识被调用, 读取数值时,

会自动跳过起始空格；`nounitbuf` 格式标识每次输出后，会清空 `output` 缓冲区。默认情况下，设置 `ios::skipws` 意味着读取数值时会跳过起始空格，因为数字之间的空格是无意义的，不需要读取。这也导致了使用“>>”时，无法读取空字符和空格字符（被忽略了）。

标识符 `ios::unitbuf` 用于控制 `output` 缓冲区。若 `ios::unitbuf` 被设置，即不使用缓冲装置，每次输出时均清空（flush）缓冲区。默认情况下，没有设立此标识，不过 `cerr` 和 `wcerr` 已预先设置它。

上述两个标识符均有其相反的形式，比如 `noskipws`，`nounitbuf`。值得注意的是，要区别它们的不同含义。`skipws` 代表着省略起始空格；`noskipws` 代表着读取输入流中的空格，即不跳过起始空格，而是将其读出；`unitbuf` 的默认状态字是 `nounitbuf`，状态字 `unitbuf` 一旦被设置，缓冲区非空时会被处理（清除）；而使用状态字 `nounitbuf` 时，每次输出结束后，并不清除输出缓冲区。

以上 4 种标识符的对应标识分别为 `ios::skipws` 和 `ios::unitbuf`。设置和清除时使用相同的标识，可以使用操作符 `setiosflags()` 和 `resetiosflags()`，还可以使用流 `cout` 的 `setf()` 函数和 `unsetf()` 函数。例如，

```
resetiosflags(ios::skipws);
char strtest[10];
cin >> strtest;
cout << strtest << endl;
cout.unsetf(ios::unitbuf);    //取消格式
cin >> strtest;
cout << strtest << endl;
```

上述代码的输出结果为：

```
sdfghe
sdfghe
12345asdf
12345asdf
```

7.4 类 streambuf

类 `streambuf` 是比较重要的缓冲区类。本节将详细介绍流缓冲区、缓冲区迭代器和自定义缓冲区等内容。

7.4.1 流缓冲区

通常，`stream` 并不负责实际读写操作，而是委托给流缓冲区实现。输出流将某些字符放入缓冲区。之后于某个时刻，这些字符被写至输出设备上。此类缓冲区被称为流缓冲区。类 `streambuf` 为缓冲区管理内存，并提供用于填充的缓冲区、访问缓冲区内容、刷新缓冲区和管理缓冲区内存的类方法。

在使用 `cout` 输出字符时，刷新输出缓冲区具有一定的意义。由于 `ostream` 类对 `cout` 对象处理的输出进行缓冲，输出不会立即发送到目标地址，而被存在缓冲区中，直至缓冲区被填满。之后，程序通过刷新缓冲区，将其中的内容发送出去，并清空缓冲区，以存储新的数

据。通常，缓冲区的大小为 512Byte 或其整数倍。当标准输出连接的是硬盘上的文件时，缓冲可以节省大量时间。程序员不希望程序为发送 512 Byte 而存取磁盘 512 次。将 512 个 Byte 收集到缓冲区中，之后一次性将其写入硬盘的效率，这才是他们所期望的。

对于屏幕输出，首先填充缓冲区的重要性要低得多。幸运的是，屏幕输出时，程序没必要等待缓冲区被填充。例如，换行符发送至缓冲区之后，缓冲区即被刷新。而多数的 C++ 语言，是在输入即将发生时，刷新缓冲区。控制符 flush 和 endl 均可实现刷新缓冲区，而控制符 endl 在刷新缓冲区时，还将插入一个换行符。

对于程序员来说，类 basic_streambuf 仅仅是发送 (sent) 或提取 (extracted) 字符的地方。通常有两个公共函数用于写入字符。它们是 sputc() 和 sputn()。sputc() 函数调用发生错误时，会返回 traits_type::eof()，之后 traits_type 是类中的型别定义。sputn() 函数将写入由第二参数指定的字符数，除非 stream 缓冲区无法使用该字符。这两个函数在使用过程中，是不考虑字符串终止符号的。其返回值是实际写出的字符数。

访问流缓冲区的接口非常复杂。对于输入而言，必须时刻监视未耗用的字符。解析时，字符最好能被送回 stream 缓冲区。为此，类 streambuf 特别提供了相应的成员函数，详见表 7-3。

表 7-3 类 streambuf 的成员函数 (输入)

输入成员函数	含 义	输入成员函数	含 义
in_avail()	返回有效字符的下界	sgetn (b, n)	读取 n 个字符，并将其存储到缓冲区中
sgetc()	返回当前字符，并不耗用它	sputback (c)	将指定字符 c 返回 stream 缓冲区中
sbumpc()	返回当前字符，并耗用它	sungetc()	退回至前一字符
snxetc()	耗用当前字符并返回下一个字符		

in_vail() 函数用于确定缓冲区中至少有多少个有效字符。用于确定从键盘读取数据时不会发生阻塞。值得注意的是，缓冲区实际上可能包含更多的有效字符。

函数 sgetc() 不必移动至下一字符即可获得当前字符。sbumpc() 函数读取当前字符并移动至下一个字符，并使之成为当前字符。snxetc() 函数将下一个字符视为当前字符，之后读取之。这 3 个函数如果调用失败，均会返回 traits_type::eof()。sgetn() 函数读取字符序列并发送至缓冲区中，其参数可以代表欲读取的字符数，返回值是实际读取的字符数目。

sputback() 和 sungetc() 函数被用于后退一步，并使前一个字符成为当前字符。sputback() 函数可将前一字符替换为其他字符。使用这两个函数，只能回退一个字符。还有部分函数用于存取局部对象，改变位置或影响缓冲区，详见表 7-4。

表 7-4 难以分类的类 streambuf 的公用函数

输入成员函数	含 义
pubimbue (loc)	为流缓冲区安装 locale loc
getloc()	返回当前的 locale
pubseekpos (pos)	将当前位置重新设定为某绝对位置
pubseekpos (pos, which)	将当前位置重新设定为某绝对值，并可指定 I/O 方向
pubseekoff (offset, rpos)	将当前位置重新设定为另一位置的相对位置
pubseekoff (offset, rpos, which)	将当前位置重新设定为另一位置的相对位置，并可指定 I/O 方向
pubsetbuf (b, n)	影响缓冲行为

`pubimbue()` 和 `getloc()` 用于国际化议题。`pubimbue()` 在 `stream` 缓冲区中安装一个新的 `locale` 对象，并返回前一个被安装的 `locale` 对象；`getloc()` 返回当前的 `locale` 对象。

`pubsetbuf()` 函数试图对缓冲区的缓冲策略进行控制，是否有效则取决于具体的类 `streambuf`。例如，对于字符串流缓冲区运用 `pubsetbuf()` 将毫无意义。即使用于文件流缓冲区，也只能在第一个 IO 操作之后以 `pubsetbuf(0, 0)` 方式调用才起作用。如果调用出错，函数返回 0；否则，返回该 `stream` 缓冲区。

`pubseekoff()` 函数和 `pubseekpos()` 函数控制读写操作的当前位置，无论控制读或写，都取决于最后一个参数（型别为 `ios_base::openmode`），如果没有特别指定，参数默认值为 `ios_base::in | ios_base::out`。一旦设置 `ios_base::in`，读取位置会跟着改变；一旦设置 `ios_base::out`，改写位置会跟着变化。`pubseekpos()` 函数会把 `stream` 当前位置移动至其第一个参数指示的绝对位置上；`pubseekoff()` 函数会把 `stream` 当前位置移动至相对位置，偏移量由第一个参数决定，起始位置由第二参数决定，标识符一般是 `ios_base::cur/ios_base::beg/ios_base::end`。这两个函数均返回 `stream` 所在位置或无效位置，将函数结果拿来和对象 `pos_type` 比较，检查出无效的 `stream` 位置。如果要获取 `stream` 的当前位置，需要使用 `pubseekoff()` 函数。例如，

```
streambuffer sbuf;
sbuf.pubseekoff(0, std::ios::cur)
```

7.4.2 缓冲区迭代器

流缓冲区和其他任意的存储空间一样，可以使用迭代器进行访问。使用流缓冲区的迭代器类，其实也是使用 `stream` 成员函数的一种形式。类 `streambuf` 提供的迭代器有两类：①符合输入型迭代器和输出型迭代器的规格和要求；②从流缓冲区读取或写入单个字符时。流缓冲区中的迭代器将字符层面的输入与输出，归入 C++ STL 的算法管辖范围内。

模板类 `istreambuf_iterator` 和 `ostreambuf_iterator` 用于从型别为 `basic_streambuf` 的对象中读取或写入单个字符。使用上述迭代器均须包含头文件 `<iterator>`。流缓冲区迭代器是流迭代器的特殊形式，唯一区别在于元素是字符。

1. 输出流缓冲区迭代器

例如，

```
std::ostreambuf_iterator <char> bufwrite(std::cout);
std::string hello("hello, world\n");
std::copy(hello.begin(), hello.end(), bufwriter);
```

第一行代码根据 `cout` 构造一个输出型迭代器，型别为 `ostreambuf_iterator`。除了传递 `output stream` 之外，可以直接传递指针，指向 `stream` 缓冲区。第二行和第三行代码分别实现定义一个字符串和将之复制到输出缓冲区的功能，`copy()` 函数执行之后，字符串 `string` 随之被输出到屏幕上。

输出流缓冲区迭代器的操作函数和输出流的迭代器近似。其成员函数的各项功能见表 7-5。使用一个缓冲区将迭代器初始化，调用 `fail()` 函数检查迭代器是否用于输出。如果任何一个字符的预写入操作失败，`fail()` 函数会返回 `true`。运算符 `operator` 进行的任何改写操作均无效。

表 7-5 输出流缓冲区迭代器成员函数的各项功能

输入成员函数	含 义
<code>ostreambuf_iterator < char > (ostream)</code>	为流 <code>ostream</code> 产生一个输出流缓冲区迭代器
<code>ostreambuf_iterator < char > (buffer_ptr)</code>	为 <code>buffer_ptr</code> 所指的缓冲区产生一个输出流缓冲区迭代器
<code>* iter</code>	无操作, 返回 <code>iter</code>
<code>iter = c</code>	调用 <code>sputc (c)</code> , 对缓冲区写入字符 <code>c</code>
<code>++ iter</code>	无操作, 返回 <code>iter</code>
<code>iter++</code>	无操作, 返回 <code>iter</code>
<code>failed ()</code>	判断输出流迭代器是否能执行改写操作

2. 输入流缓冲区迭代器

输入流缓冲区迭代器的所有操作函数和类 `istream` 迭代器的所有操作函数近似。其成员函数的各项功能见表 7-6。通常, 成员函数 `equal ()` 用于判断两个输入流缓冲区迭代器是否相等。当两个流缓冲区迭代器都是或都不是 `end-of-stream` 迭代器时, 两者均视为相等。

表 7-6 输入流缓冲区迭代器成员函数的各项功能

输入成员函数	含 义
<code>istreambuf_iterator < char > ()</code>	产生一个 <code>end-of-stream</code> 迭代器
<code>istreambuf_iterator < char > (istream)</code>	为 <code>istream</code> 建立一个输入流缓冲区迭代器, 并可能调用 <code>sgetc ()</code> 函数读取第一个字符
<code>Istreambuf_iterator < char > (buffer_ptr)</code>	为 <code>buffer_ptr</code> 所指向的输入流产生一个输入流缓冲区迭代器, 并调用 <code>sgetc ()</code> 函数读取第一个字符
<code>* iter</code>	返回当前字符, 即是先前调用 <code>sgetc ()</code> 函数读取的字符 (如果构造函数未执行读取操作, 在此执行)
<code>++ iter</code>	调用 <code>sbumpc ()</code> 函数读取下一个字符, 并返回其位置
<code>iter++</code>	调用 <code>sbumpc ()</code> 函数读取下一个字符, 返回一个迭代器
<code>equal ()</code>	判断两个迭代器是否相等
<code>==</code>	判断两个迭代器是否相等
<code>!=</code>	判断两个迭代器是否不相等

需要注意的有以下两点:

1) 从流当前位置到流尾部之间的范围用两个迭代器定义出来: `istreambuf_iterator < charT, traits > (stream)` 和 `istreambuf_iterator < charT, traits > ()`, 其中 `stream` 的型别是 `basic_istream < charT, traits >` 或 `basic_strrdbuf < charT, traits >`。

2) 不可能以 `istreambuf_iterators` 建立出一个子序列。

下面给出一个缓冲区迭代器的实例。其实现的功能为: 使用流缓冲区迭代器简单的输出所有读取到的字符。

```
#include <iostream>
#include <algorithm>
using namespace std;
void main ()
```

```
{ ostreambuf_iterator<char> bufwrite(cout);
  string hello("Hello, world! . \n");
  copy(hello.begin(),hello.end(),bufwrite); //复制字符串至缓冲区
  istreambuf_iterator<char> inpos(cin);
  istreambuf_iterator<char> endpos;
  ostreambuf_iterator<char> outpos(cout);
  while(inpos!=endpos)
  {
    * outpos=* inpos;
    ++inpos;
    ++outpos;
  }
}
```

上述代码的输出结果为：

```
Hello, world! .
asdfghjk,lmn
asdfghjk,lmn
```

7.4.3 自定义缓冲区

1. 流缓冲区的简述

流缓冲区是一种 I/O 缓冲区，其接口由类 `basic_streambuf` 定义。针对字符型别 `char` 和 `wchar`，C++ STL 分别提供了预先定义好的流缓冲区（`streambuf`）和宽字符流缓冲区（`wstreambuf`）。尤其在特殊通道上通信时，各类可以作为基类。要实现这一点，必须对流缓冲区的操作有所了解。缓冲区的主要接口由 3 个指针构成。`eback()`、`gptr()`和 `egptr()`函数返回的指针构成了 `read (input)` 缓冲区的界面。`pbase()`、`pptr()`和 `epptr()`函数返回的指针构成了 `write (output)` 缓冲区的接口。这些指针分别由 I/O 操作操控，后者会导致相关 I/O 通道上的相关响应。精确操作将分为读取和写入。

对于程序开发者而言，实现自定义输出流缓冲区尤为重要。

输出流缓冲区一般由 3 个指针维护。这 3 个指针分别由 `pbase()`、`pptr()`和 `epptr()`函数获得。它们表示的意义为：

- 1) `pbase()`是指输出流缓冲区的起始位置。
- 2) `pptr()`是当前写入位置。
- 3) `epptr()`是输出流缓冲区的结尾，指向“得被缓冲之最后一个字符”的下一个位置。

`pbase()` ~ `pptr()`的序列字符已被写至相应的输出通道，但尚未清空。

成员函数 `sputc()`可以写入一个字符。如果当时有个空的改写位置，字符就会被复制到该位置上。之后，指向当前改写位置的那个指针会加 1。如果缓冲区是空的，就调用虚 `overflow()`函数将 `output` 缓冲区的内容发送至对应的输出通道中。这个函数能有效地将字符送至某种“外部表述”。基类 `basic_streambuf` 所实例化的 `overflow()`函数只返回 `end-of-file()`，表示没有更多字符被写入。

成员函数 `sputn()`可用来一次写入多个字符。该函数把实际任务委派给虚 `xspun()`函数。

后者可针对“多个字符”做出更有效的操作。类 `basic_streambuf` 中的 `xsputn()` 对每个字符调用 `sputn()`。改写 `xsputn()` 不是必要的。通常，“同时写入多个字符”会比“一次写入一个字符”效率高得多，多用 `sputn()` 优化对字符序列的处理。

同时，广大程序员也应该认识到，对一个流缓冲区写入数据时不一定要采取缓冲行为，而可以令字符即刻发送。此时，默认构造函数会自动维护“输出缓冲区”的指针设置为 0 或 `NULL`。

2. 虚 `overflow()` 函数详解

通过上述描述，给出以下实例。例中并未采取缓冲行为，而是对每个字符都调用 `overflow()`。

简要介绍一下虚 `overflow()` 函数。Visual C++ 软件开发环境的类 `basic_streambuf` 类的保护成员虚 `overflow()` 函数的说明中提到：`overflow()` 是保护型虚成员函数，每当有新的字符被插入至空的缓冲区内时，`overflow()` 即被调用。该函数的默认返回值是 `traits_type::eof`。如果函数调用失败，将返回 `traits_type::eof` 或抛出一个异常；否则，该函数的返回值为 `traits_type::not_eof()`。

其原型为：

```
virtual int_type overflow(int_type _Meta = traits_type::eof());
```

参数 `_Meta` 并不和 `traits_type::eof` 相等，在该函数被调用时，将努力插入字符型变量 `_Meta`，因为参数是 `int_type` 型，执行过程中被插入到输出缓冲区中的数值是进行了数值类型转换的，即 `traits_type::to_char_type()`。以上功能的实现是通过多种方式实现的：

- 1) 如果“写”的位置有效，元素被存储至写的位置，并增加输出缓冲区的下一个指针。
- 2) 通过分配新的或额定的存储空间至输出缓冲区，以确保每个“写”的位置均有效。
- 3) 通过“写出”至一些外部目标地址，部分或所有元素均保证在输出流缓冲区的起始位置和下一个指针之间。

虚 `overflow()` 和 `sync()`、`underflow()` 函数，定义了输出流缓冲区类的特性。每个类会采用不同的方式执行 `overflow()` 函数，但调用流类的接口是相同的。`overflow()` 函数是最频繁调用的函数之一，通常被流缓冲区类的函数调用，例如 `sputc()` 和 `sputn()`。其他流类也可随时调用 `overflow()` 函数。

`overflow()` 函数耗用输出区域中指针 `pbase` 和指针 `pptr` 之间的字符，并重新初始化这些区域。`overflow()` 函数必须耗用字符 `nCh`，或它可能选择将字符放置入新的输出区域，以便于在下一次调用时被耗费。

在不同的派生类中，耗用的定义是多种多样的。例如，文件缓冲区类 `filebuf` “写字符”到文件时，类 `streambuf` 保持这些字符在缓冲区中，并响应 `overflow()` 的调用扩张缓冲区。通过释放旧的缓冲区，并使用新的缓冲区，更大的缓冲区取代旧的缓冲区，实现扩张内存的目的。同时，指针的调整也是必需的。

鉴于以上阐述，在实现自定义流缓冲区类时，一定要派生出保护类型虚成员函数 `overflow()`。

3. 自定义输出缓冲区最简单的实例

下面实现一个最简单的自定义缓冲区实例。此例题是摘自《C++ 标准程序库》一书。

例 7-7

```
#include <iostream>
#include <streambuf>
#include <locale>
#include <cstdio>
using namespace std;
class outbuf: public std::streambuf
{
protected:
    virtual int_type overflow(int_type c)
    {
        if(c! = EOF)
        {
            c = std::toupper(c, getloc());
            if(putchar(c) == EOF)
            {
                return EOF;
            }
        }
        return c;
    }
};
void main()
{
    outbuf ob;
    std::ostream out (&ob); //自定义输出缓冲区
    int num = 56;
    out << "56 十六进制数值: " << std::hex << std::showbase << num << endl;
}
```

例 7-7 的输出结果为:

```
56 十六进制数值: 0X38
```



提示

getloc() 函数是获取基本流缓冲区类对象的场合 (或场景)。

对于较复杂的自定义输出以及其他任意目标的写入操作, 例如文件、socket 连接名或流缓冲区, 若向目标端改写数据, 只需实例相应的 overflow() 即可。若有必要, 也需要实例化 xsputn() 函数, 有助于提升效率。为方便构造流缓冲区, 实例化特殊类, 用于构造函数参数传递给相应的流缓冲区。例 7-13 中使用 fprintf() 函数写入文件, 使用 ostream 类的继承类, 用以维护流缓冲区。此外, 如果需要具备缓冲能力的流缓冲区, “写” 缓冲区需要使用 setp() 函数初始化, 并使用 sync() 函数实现同步。

setp() 函数的原型为:

```
void setp(char_type * _Pbeg, char_type * _Pend);
```

其作用是保存参数_Pbeg 和_Pend 限定的输出缓冲区内容。

sync() 函数的原型为:

```
int sync()
```

其作用是实现控制流和外部关联流 (设备) 之间的同步。

例 7-8

```
#include <iostream>
#include <io.h>
#include <streambuf>
#include <cstdio>
using namespace std;
static const int bufferSize = 10;
class outbuf:public std::streambuf{
protected:
    char buffer[bufferSize];
public:
    outbuf()
    {
        setp(buffer,buffer + (bufferSize - 1));           //输出
    }
    virtual ~outbuf()
    {
        sync();
    }
protected:
    int flushBuffer()
    {
        int num = pptr() - pbase();
        if(write(1,buffer,num) != num)
        {
            return EOF;
        }
        pbump(-num);
        return num;
    }
    virtual int_type overflow(int_type c)
    {
        if(c != EOF){
            * pptr() = c;
            pbump(1);
        }
        if(flushBuffer() == EOF)
        {
            return EOF;
        }
    }
};
```

```
        return c;
    }
    virtual int sync()
    {
        if (flushBuffer() == EOF)
        {
            return -1;
        }
        return 0;
    }
};

class fdostream: public std::ostream{
protected:
    outbuf buf;
public:
    fdostream(int fd):ostream(0)
    {
        rdbuf(&buf);
    }
};

void main()
{
    fdostream out(1); //设置输出缓冲区
    out << "51 hexadecimal: " << std::hex << std::showbase << 51 << endl;
}
```

例 7-8 的执行结果为:

```
51 hexadecimal: 0x33
```

4. 自定义输入缓冲区

输入机制和输出机制是基本相同的。对输入而言,可能不进行最后的读取操作。sungetc()函数或 sputback()函数可用于存储流缓冲区最后一次读取前的状态(可能需要读取下一字符但并不移动读取位置)。实例化“从 stream 缓冲区中读取”操作和实例化“向 stream 缓冲区写入数据”的操作相比,必须改写(重载)更多的函数。

同样,流缓冲区以 3 个指针维护一个 read 缓冲区,这些指针通过成员函数 eback()、gptr()和 egptr()获取。

1) eback()指的是 input 缓冲区的起始位置,或者回退区的尾端。如果不采取特殊措施,字符最多只能被回退到这个位置。

2) gptr()是当前的“读取位置”。

3) egptr()是 input 缓冲区的尾端。

读取位置和结束位置之间的字符已经从外部表述装置被传至程序内存,但仍等待着程序的处理。sgetc()函数或 sbumpc()函数可以读取单一字符。两者的差别在于后者会令“读取指针”前进,而前者不会。如果缓冲区读取完毕,就不再有可用字符了。缓冲区必须重新获得补给。这项工作可由虚函数 underflow()完成,underflow()函数负责读取数据。如果没

有可用字符, `sbumpc()` 函数会调用虚函数 `uflow()`, 而 `uflow()` 的默认行为即调用 `underflow()`, 移动(前进)“读取指针”。基类 `basic_streambuf` 中对 `underflow()` 的默认做法是令它返回 EOF, 这意味着不可能以此默认版本读取字符。

`sgetn()` 函数用于一次读取多个字符。这个函数把任务委派给虚函数 `xsgetn()`, 后者的默认做法是简单地对每个字符调用 `sbumpc()`。像重载函数 `xspn()` 一样, 通过改进 `xsgetn()` 优化多个字符的读取过程。

输入和输出还有不同, 对于输入而言, 仅仅改写一个函数是不够的, 必须建立缓冲区, 至少实例化函数 `underflow()` 和 `uflow()`。因为 `underflow()` 不会将“读取指针”移动到当前字符之后, 只能通过调用 `sgetc()` 移至下一字符, 并需以缓冲区操作函数或 `uflow()` 完成。任何一个具备字符读取功能的流缓冲区必须实例出 `underflow()`。若 `underflow()` 和 `uflow()` 均被实例化了, 就没必要建立缓冲区了。

成员函数 `setg()` 可以建立一个 `read()` 缓冲区。该函数有三个参数, 依次如下:

- 1) 一个指针指向缓冲区头部 (`eback()`)。
- 2) 一个指针指向当前读取位置 (`gptr()`)。
- 3) 一个指针指向缓冲区尾部 (`egptr()`)。

和 `setp()` 不同的是, `setg()` 包括三个参数, 这是必需的。它们能定义出用来储存“将被回退给 stream”的字符空间。因此, 一旦指向 `read` 缓冲区的指针已经被设定好, 会有部分字符被读取但仍存放在缓冲区内。运用 `sputbackc()` 和 `sungetc()` 可将字符回退到 `read` 缓冲区中, `sputbackc()` 是以回退字符作为参数, 并确保该字符确实是被读取的字符。如果可能, 这两个函数会将读取指针退回一步。只有当读取指针不指向 `read` 缓冲区头部时, 才能实现。如果到达缓冲区头部, 试图退一字符, 虚函数 `pbackfail()` 会被调用。通过改写函数可以实例出“即使在这种情况下也能恢复原读取位置”的机制。基类 `basic_streambuf` 并未定义相应操作, 实际上不可能回退任意字符。如果不使用缓冲区的 streams, `pbackfail()` 函数应该被实例化出来, 这些 streams 通常假设至少有一个字符可以被回退。若新缓冲区只用于读取, 会产生新的问题, 即缓冲区没有将旧数据保存下来, 连一个字符也无法回退。实例化 `undereflow()` 时经常把当前缓冲区的最后若干个字符移到头部, 之后再添加新读取的字符, 且允许在调用 `pbackfail()` 之前回退一些字符。

例 7-9

```
#include <iostream>
#include <streambuf>
#include <cstring>
#include <io.h>
using namespace std;
static const int bufferSize = 10;
class inbuf:public std::streambuf
{
protected:
    char buffer[bufferSize];
public:
    inbuf()
    {
```

```
        setg(buffer + 4,buffer + 4,buffer + 4);
    }
protected:
    virtual int_type underflow() {
        if(gptr() < egptr()) //未至文件末尾
        {
            return * gptr();
        }
        int numputback;
        numputback = gptr() - eback(); //当前位置至文件开头的距离
        if(numputback > 4) {
            numputback = 4;
        }
        memcpy(buffer + (4 - numputback), gptr() - numputback, numputback);
        int num;
        num = read(0,buffer + 4,bufferSize - 4);
        if(num < = 0) {
            return EOF;
        }
        setg(buffer + (4 - numputback),buffer + 4,buffer + 4 + num);
        return * gptr();
    }
};

void main()
{
    inbuf ib;
    std::istream in(&ib); //自定义输入缓冲区
    char c;
    for(int i = 1; i < = 20; i++)
    {
        in.get(c);
        cout << c << flush;
        if(i == 8)
        {
            in.unget();
            in.unget();
        }
    }
    cout << endl;
}
```

例 7-9 的输出结果为:

```
asdfghj
asdfghj
j
```



总结 请读者认真阅读上述几个例题，体会如何使用自定义输出缓冲区和自定义输入缓冲区取代 cout 和 cin，并认真体会各函数的用法。

7.5 基于字符串的流

在 STL 的头文件 `<sstream>` 中定义了 4 个类模板和 6 个类型。它们均是和 `basic_string` 相关的流缓冲区类型。其功能主要是实现将流附着在字符串上，即通过流所提供的格式化功能，从字符串中读取或写入流。同时，这样的流也被命名为基于字符串的流（string stream）。其在头文件中的定义为：

```
template <class Elem, class Tr = char_traits<Elem>, class Alloc = allocator<Elem> > class basic_stringstream :
    public basic_istream<Elem, Tr>
```

7.5.1 stringstream 类

类模板 `basic_stringbuf` 的定义形式为：

```
template <class Elem, class Tr = char_traits<Elem>, class Alloc = allocator<Elem> > class basic_stringbuf :
    public basic_streambuf<Elem, Tr>
```

其中包含 5 个公用类型定义、两个构造函数、一个 `get()` 函数和一个 `set()` 函数，并重载了 6 个虚函数。

```
public:
    typedef      charT          char_type;
    typedef      typename      traits::int_type      int_type;
    typedef      typename      traits::pos_type      pos_type;
    typedef      typename      traits::off_type      off_type;
    typedef      traits          traits_type;
```

两个构造函数：

```
explicit basic_stringbuf(ios_base::openmode which = ios_base::in | ios_base::out);
explicit basic_stringbuf(const basic_string<charT, traits, Allocator> &str, ios_base::openmode
    which = ios_base::in | ios_base::out);
```

`get()` 函数和 `set()` 函数的形式为：

```
basic_string<charT, traits, Allocator> str() const;
void str(const basic_string<charT, traits, Allocator> &s);
```

6 个虚函数的形式：

```
virtual int_type underflow();
virtual int_type pbackfail(int_type c = traits::eof());
virtual int_type overflow(int_type c = traits::eof());
```

```
virtual basic_streambuf<charT,traits>* setbuf(charT* , streamsize);
virtual pos_type seekoff(off_type off, ios_base::seekdir way, ios_base::openmode which = ios_base::in
|ios_base::out);
virtual pos_type seekpos(pos_type sp, ios_base::openmode which = ios_base::in |ios_base::out);
```

下面对其各成员函数和构造函数进行详细说明。

两个构造函数的主要功能是构造基类 `basic_streambuf` 的对象，并初始化基类，使用参数 `which` 初始化模式。

```
explicit basic_stringbuf(ios_base::openmode which = ios_base::in |ios_base::out);
explicit basic_stringbuf(const basic_string<charT, traits, Allocator>& str, ios_base::openmode
which = ios_base::in |ios_base::out);
```

成员函数 `str()` 的功能是获取字符串缓冲区中的内容或初始化输入输出序列。

虚 `underflow()` 函数将参数指定的字符放入输入缓冲区，如果可能，需要 3 种方法：

1) 如果输入序列的当前位置有效，并且 `traits::eq_int_type(c, traits::eof())` 返回 `false`、`traits::eq(to_char_type(c), gptr()[-1])` 返回 `true` 以及 `assigns(gptr()-1 to gptr())`，上述条件均满足时，函数返回字符 `c`。

2) 如果 `traits::eq_int_type(c, traits::eof())` 返回 `false`，并且假设输入序列的当前位置有效、模式为输出 (`ios::out`) 时，指定字符 `c` 给 `*(--gp())`，函数返回 `traits::not_eof(c)`。

说明：`traits::eof()` 函数表明函数调用失败。

成员函数 `overflow()` 的功能是将参数指定的字符 `c` 送入输出缓冲区，如果可能，有两种可能的办法：

1) 当 `traits::eq_int_type(c, traits::eof())` 返回 `false`，并输出缓冲区当前位置有效时，函数会调用 `sputc(c)` 函数，如果调用成功，函数返回字符 `c`。

2) 如果 `traits::eq_int_type(c, traits::eof())` 返回 `true`，将没有字符被添加 (`append`)。函数调用成功时，返回一个数值；调用失败时，返回 `traits::eof()`。

成员函数 `seekoff()` 的功能是在一个可控序列中改变流的位置。`seekpos()` 函数的功能和 `seekoff()` 函数近似。

7.5.2 类模板 `basic_istream`

类模板 `basic_istream` 支持读取 `basic_string` 类型的对象。它使用 `basic_stringbuf` 类型的对象控制相关的存储区域。

类模板 `basic_istream` 的定义形式为：

```
template<class charT, class traits = char_traits<charT>, class Allocator = allocator<charT>>
class basic_istream: public basic_istream<charT, traits>
```

其中包括 5 个公用类型定义、两个构造函数和 3 个成员函数。

5 个公用类型的定义形式为：

```

typedef      charT      char_type;
typedef      typename   traits::int_type   int_type;
typedef      typename   traits::pos_type   pos_type;
typedef      typename   traits::off_type   off_type;
typedef      traits      traits_type;

```

两个构造函数的形式为:

```

explicit basic_istringstream (ios_base::openmode which = ios_base::in);
explicit basic_istringstream(const basic_string< charT, traits, Allocator >& str, ios_base::
openmode which = ios_base::in);

```

3 个成员函数的形式为:

```

basic_stringbuf(charT, traits, Allocator >)* rdbuf() const;
basic_string< charT, traits, Allocator > str() const;
void str(const basic_string< charT, traits, Allocator >&s);

```

两个构造函数主要用于构造类的对象及其初始化缓冲区。

其中成员函数 `rdbuf()` 用以获取缓冲区内的内存 (数据) 指针, 成员函数 `str()` 用以获取或指定缓冲区的内容。

7.5.3 类模板 `basic_ostringstream`

类模板 `basic_ostringstream` 支持写入 `basic_string` 类型的对象。它使用 `basic_stringbuf` 类型的对象控制相关的内存区域。为方便下面描述, 假定标志 `sb` 代表字符串缓冲区对象 (`stringbuf` object)。类模板 `basic_ostringstream` 的定义形式为:

```

template<class charT, class traits = char_traits< charT >, class Allocator = allocator< charT
>>
class basic_ostringstream : public basic_ostream< charT, traits >

```

其中包括 5 个公用类型、两个构造函数和 3 个成员函数。

5 个公用类型的定义形式为:

```

typedef      charT      char_type;
typedef      typename   traits::int_type   int_type;
typedef      typename   traits::pos_type   pos_type;
typedef      typename   traits::off_type   off_type;
typedef      traits      traits_type;

```

两个构造函数的定义形式为:

```

explicit basic_ostringstream(ios_base::openmode which = ios_base::out);
explicit basic_ostringstream (const basic_string< charT, traits, Allocator > & str, ios_
base::openmode
    which = ios_base::out);

```

3 个成员函数的定义形式为:

```

basic_stringbuf< charT, traits, Allocator >* rdbuf() const;
basic_string< charT, traits, Allocator > str() const;
void str(const basic_string< charT, traits, Allocator >&s);

```

构造函数用以构造类的对象，并初始化缓冲区。成员函数 `rdbuf()` 获取缓冲区指针。成员函数 `str()` 既可以获取缓冲区中内容，也可以设置缓冲区内存的内容。

7.5.4 类模板 `basic_stringstream`

类模板 `basic_stringstream` 既可以支持读取 `basic_string` 类对象，也可以支持写入 `basic_string` 类对象。同样，该类使用 `basic_stringbuf` 类的对象控制相关联的序列。类模板 `basic_stringstream` 的定义形式为：

```
template<class charT, class traits = char_traits<charT>, class Allocator = allocator<charT>>
class basic_
    stringstream : public basic_ostream, traits >
```

该类模板包括 5 个公用类型、两个构造函数。

5 个公用类型的定义形式如下：

```
typedef      charT      char_type;
typedef      typename   traits::int_type   int_type;
typedef      typename   traits::pos_type   pos_type;
typedef      typename   traits::off_type   off_type;
typedef      traits      traits_type;
```

两个构造函数的定义形式如下：

```
explicit basic_stringstream(ios_base::openmode which = ios_base::out | ios_base::in);
explicit basic_stringstream(const basic_string<charT, traits, Allocator> &str, ios_base::
openmode which
    = ios_base::out | ios_base::in);
```

两个构造函数实现构造类的对象并初始化缓冲区。

成员函数 `rdbuf()` 获取缓冲区的指针；成员函数 `str()` 获取缓冲区中的内容或设置缓冲区中的内容。

7.6 基于文件的流

类 `stream` 可用来存取文件。C++ STL 提供 4 个模板类，用以预先定义 4 个标准特化版本。这 4 个标准特化版本分别是：

- 1) `template class basic_ifstream<>` 及其特化版本 `ifstream` 和 `wifstream`。该版本用于读取文件。
- 2) `template class basic_ofstream<>` 及其特化版本 `ofstream` 和 `wofstream`。该版本用于将数据写入文件。

- 3) `template class basic_fstream<>` 及其特化版本 `fstream` 和 `wfstream`。该版本用于读写文件。

- 4) `template class basic_filebuf<>` 及其特化版本 `filebuf` 和 `wfilebuf`。该版本被其它文件流类用于进行实际字符的读写工作。

以上这些类的使用，均须包含头文件 `<fstream>`。和 C 语言的文件存取机制相比，C++ 文件流类的最大好处是实现文件的自动管理。文件在构造时会自动打开，析构时自动关闭。

这是因为文件流类使用了非常好的构造函数和析构函数。

对于既可读取亦可改写的流,不允许在读写操作之间任意转换其读写属性。但有时需要在开始读写过程之间转换流属性,甚至需要进行一个 seek 操作,则到达当前位置后,再转换读写属性。如果已经到达文件末尾,可立即接着写入字符。

文件流对象以某个 C-string 为构造函数参数,会自动打开该字符所代表的文件,用于读和写。读或写操作是否成功,会体现在流的状态位中。文件中的内容既可以单个字符的读取和输出,也可以实用文件缓冲区的指针,一次性输出其文件的内容。例如,

```
while( file.get(c)
      cout.put(c);
```

或

```
cout << file.rdbuf();
```

7.6.1 文件标识及其使用

文件描述符是代表文件的标识。每个流会被对应一个开启的 I/O 通道。文件描述符可以将文件流初始化。

文件描述符是一个整数,用于识别被开启的 I/O 通道。尤其在 UNIX 系统中,文件描述符是属于底层接口的。通常,C++ 系统预先定义了 3 个文件描述符:①0 (零)代表标准输入通道;②1 代表标准输出通道;③2 代表标准错误信息通道。

这些通道可能连接文件、控制台、其他进程或 I/O 设备。

目前,C++ STL 中不存在“运用文件描述符将流附着在 I/O 通道”的可能性。通常,这部分功能不具备移植性,目前多数操作系统也不存在这样的标准接口。

1. 文件简介

为了准确控制文件处理模式,类 ios_base 定义了一组标识,其型别均为 openmode,类似于 fstream 的位掩码型别,这些标识的意义见表 7-7。

表 7-7 文件标识的意义

标识	意义	标识	意义
in	打开,用于读写 (ifstream 的默认模式)	ate	打开文件之后令读写位置移至文件尾端
out	打开,用于改写 (ofstream 的默认模式)	trunc	将先前的文件内容移除
app	写入时始终添加于尾端	binary	不要替换特殊字符

标识 binary 使得 stream 能够阻止特殊字符或字符序列的转换。对于以前的 MS-DOS 操作系统的文字文件,每一行结束时均是以两个字符 (“回车 CR”和“换行 LF”)表示的。正常模式下,将以上两个字符替换 newline (换行)字符。对于二进制模式,则不会进行上述转换。当文件是二进制时,需要使用标识 binary。当复制文件时,将源文件字符逐一读出,不做任何修改地写入目标文件。如果文件是文本类型,不需要使用 binary。若此时处理换行符号,则改为两个字符。

部分 C++ 版本还提供了 nocreate 和 noreplace,但这两个关键标识不是标准的内容。

多个标识可使用操作符 “|” 组合起来,其最终结果作为构造函数的第二参数。例如,

```
std::ofstream file("a.txt",std::ios::out|std::ios::app)
```

标识之间并不是可以任意进行“或”组合。常见的“C++ 标识组合”和“C 的文件打开 fopen() 函数所使用之接口字符串”间的关联见表 7-8。

表 7-8 ios_base 标识的意义

ios_base 标识	意 义	C 模式
in	读取（文件必须存在）	“r”
out	清空之后改写（有必要才产生）	“w”
out trunc	清空之后改写（有必要才产生）	“w”
out app	添加（有必要才产生）	“a”
in out	读和写：最初位置在起始点（文件必须存在）	“r+”
in out trunc	先清空，再读写（有必要才产生）	“w+”

需要指出的是，无论是读文件还是写文件，在打开文件时，真正执行打开文件操作的是缓冲区，对文件的实际操作才是由文件流实现的。

文件流类型的文件，是可以显式地被开启和关闭的。C++ STL 定义了 3 个函数。这 3 个函数分别是：

```
open(name)
open(name, flags)
close()
isopen()
```

C++ 程序员需要养成的一个好习惯：文件打开之后，完成必要的操作；使用完毕之后，在关闭文件之前，一定要使用 `clear()` 函数将设置于文件尾端的状态标识清除。因为在 C++ 标准中，文件是可以被多个用户所共享的。

2. 简单的文件 I/O

要使用程序实现文件的写入，需要以下 3 步骤：

- 1) 创建一个 `ofstream` 对象，管理输出流。
- 2) 该对象与特定的文件进行关联。
- 3) 以使用 `cout` 方式使用此对象——输出将进入文件，而不是屏幕。

使用类 `Ofstream` 时需要包含头文件 `<fstream>`。对于大多数情况而言，包含该文件便会自动包括 `iostream` 文件，因此可以不必显式包含 `<iostream>`。类 `Ofstream` 在创建对象时，会同时分配缓冲区空间。使用缓冲空间实现文件操作，可以大大提高文件传输数据的速度。

同样，程序在实现文件读取时，通常需要以下 3 个步骤：

- 1) 创建一个 `ifstream` 对象，实现管理输入流。
- 2) 此对象会与特定的文件关联起来。
- 3) 以使用 `cin` 方式使用此对象。

读文件的步骤和写文件的步骤相似，同样需要包含头文件 `<fstream>`，之后声明一个类 `Ifstream` 的对象，并与文件名关联。之后可以像使用 `cin` 一样，使用该流处理文件了。

需要指出的是，当输入流和输出流对象过期时，该流至文件的连接会自动关闭。当然也可以使用 `close()` 方式显式地关闭文件和流之间的联系。当调用 `close()` 函数时，并不会真正地删除流，仅仅是断开流到文件的连接而已，该流仍然存在，该流的缓冲区仍然存在，并且该流还可以重新连接到同一个文件或另一个文件。

例 7-10

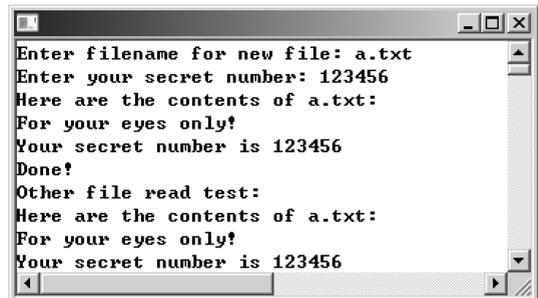
```

#include <iostream> //need
#include <fstream>
#include <string>
using namespace std;
void main()
{
    string filename;
    cout << "Enter filename for new file: ";
    cin >> filename;
    ofstream fout(filename.c_str()); //将流与文件关联
    fout << "For your eyes only! \n"; //输出至文件
    cout << "Enter your secret number: ";
    float secret;
    cin >> secret;
    fout << "Your secret number is " << secret << endl; //将变量 secret 输出至文件
    fout.clear(); //清除文件状态标识
    fout.close();
    ifstream fin(filename.c_str());
    cout << "Here are the contents of " << filename.c_str() << ": \n";
    char ch;
    while(fin.get(ch)) //读取文件的一种形式
    {
        cout << ch;
    }
    cout << "Done! \n";
    fin.clear();
    fin.close();
    cout << "Other file read test:\n";
    ifstream fin2(filename.c_str());
    cout << "Here are the contents of " << filename.c_str() << ": \n";
    cout << fin2.rdbuf(); //读文件的另一种形式
    fin2.clear();
    fin2.close();
}

```

上述代码中，ofstream fout (filename.c_str()) 实现将文件流和数据文件关联，fout << "For your eyes only! \n" 实现将字符串 "For your eyes only! \n" 写入文件中。fout << "Your secret number is " << secret << endl 实现将变量输出至文件中，fout.clear() 实现清楚文件状态标识，get(ch) 是读取文件的一种形式，而 fin2.rdbuf() 是读取文件的另一种形式。

例 7-10 的执行效果如图 7-4 所示。



```

Enter filename for new file: a.txt
Enter your secret number: 123456
Here are the contents of a.txt:
For your eyes only!
Your secret number is 123456
Done!
Other file read test:
Here are the contents of a.txt:
For your eyes only!
Your secret number is 123456

```

图 7-4 例 7-10 的执行效果

3. 文件的打开和文件模式

前面已经讲述了文件描述符和文件模式等内容。本小节主要讲述文件的打开和文件模式。前面的例题在使用文件时，在流定义时将流和文件进行关联，同时打开了该文件。类 `Ofstream` 还定义了 `open()` 函数和 `is_open()` 函数，用于实现文件的打开和是否打开的判断。

类 `Ifstream` 的 `open()` 函数的声明形式为：

```
void open(const char * s, ios_base::openmode mode = ios_base::in);
```

`is_open()` 的函数的声明形式为：

```
bool is_open();
```

类 `Ofstream` 的 `open()` 函数的声明形式为：

```
void open(const char * s, ios_base::openmode mode = ios_base::out | ios_base::trunc);
```

`is_open()` 函数的声明形式为：

```
bool is_open();
```

C++ 文件流从类 `Ios_base` 处继承了一个流状态成员。该成员存储了标明流状态的信息，如一切顺利、已达文件尾、I/O 操作失败等。通常，流状态为零。和其他状态一样，流状态也是通过将特定位设置为 1 而实现标识的。对于文件流，这些状态字还包括了判断“打开文件”是否成功。例如，当要打开的文件未找到或不存在时，`failbit` 会被设置为 1。例如，

```
istream fin;
fin.open(filename);
if(fin.fail())
{
}
```

`is_open()` 函数用于检查“打开文件”是否顺利。类 `Ifstream` 和 `Ofstream` 均提供了 `is_open()` 函数，可用以判断文件是否被正确打开。例如，

```
if(fin.is_open())
{
...
}
```

前面已经介绍过文件模式的概念，此处再补充一些关于文件模式的内容。

文件模式描述的是文件被如何使用，如读、写、追加等。流和文件关联时，可以提供指定文件模式的第二个参数。例如，

```
ifstream fin("abc.txt", mode);
```

或

```
ofstream fout;
fout.open("abc.txt", mode);
```

对于 C++ 语言，类 `Ios_base` 定义了一个 `openmode` 类型，用于表示文件模式；与 `fmtflags` 和 `iostate` 类型是一样的，是 `bitmask` 数值类型。选择类 `Ios_base` 中定义的多个常量来指定

模式。

无论是类 `Istream` 还是 `Ofstream` 类，其构造函数和 `open()` 函数均能提供两个参数，其第二个参数即是用来表示文件模式的。位操作符 OR (“|”) 用于将两个位值合并成一个可用于设置两个位的值。模式必须显式提供，类 `Fstream` 没有提供默认的模式值。在创建类对象时，类必须显式地提供模式。

`ios_base::trunc` 标识意味着打开已有的现存文件，用于接收程序输出。文件打开之后，以前的内容将被删除。此类行为极大地降低了耗尽磁盘空间的危险。当 C++ 提供其他选择时，例如在文件末尾追加新内容，即可使用 `ios_base::app` 模式。

C++ 标准根据 ANSI C 标准 I/O 定义了部分文件 I/O。

```
ifstream fin(filename, C++mode)
```

实现上面的 C++ 语句时，其中 C++ 模式是一个 `openmode` 类型值，例如 `ios_base::in`。在 C++ 语言中，模式是相应的 C 模式字符串，例如 “r”。在前面的内容中，C++ 模式和 C 模式的对应关系已经列出。值得注意的是，`ios_base::ate` 和 `ios_base::app` 均会将文件指针指向将要打开的文件末尾。但是，`ios_base::app` 模式仅允许将数据添加至文件末尾，而 `ios_base::ate` 模式会将指针放到文件尾。

(1) 给文件追加内容

在文件末尾追加数据时，程序会维护一个存储清单的文件。该程序首先显示文件当前的内容。在文件被打开之后，使用 `is_open()` 函数来检查该文件是否存在。之后，程序以 `ios_base::app` 模式打开文件，进行输出。程序请求用户从键盘输入时，会将其添加到文件中。之后，程序显示修订后的文件内容。

下面给出一个关于给文件追加内容的例题。

例 7-11

```
#include <iostream>
#include <fstream>
#include <string>
#include <cstdlib>
using namespace std;
const string filename = "test.txt";
void main()
{
    ifstream fin;
    fin.open(filename.c_str(), ios_base::in);           //打开文件
    if(fin.is_open())
    {
        cout << "Here are the current contents of the \"" << filename << "\" : " << endl;
        cout << fin.rdbuf();                          //输出文件内容
        cout << endl;
    }
    fin.clear();
}
```

```
    fin.close();
    ofstream fout;
    fout.open(filename.c_str(),ios_base::out|ios_base::app);
    if(! fout.is_open())
    {
        cerr <<"Can't open " << filename <<" file for output. \n";
        exit(EXIT_FAILURE);
    }
    cout <<"Enter new file contents (to be append) : \n";
    string buffer;
    while(getline(cin,buffer)&&buffer.size() >0)           //输入整行数据
    {
        fout <<buffer <<endl;
    }
    fout.clear();
    fout.close();           //关闭文件
    fin.open(filename.c_str(),ios_base::in);
    if(fin.is_open())
    {
        cout <<"Here are the current contents of the '\" << filename <<" '\" :\" <<endl;
        cout <<fin.rdbuf();
        cout <<endl;
    }
    fin.clear();
    fin.close();
    return;
}
```

例 7-11 的输出结果为:

```
Here are the current contents of the 'test.txt' :
这里是北京. abcdefg
lkjhgf
这是最后一行了。
abefjus
qwertyuiop
再添加一行吧。

Enter new file contents (to be append) :
再加一行。

Here are the current contents of the 'test.txt' :
这里是北京. abcdefg
lkjhgf
这是最后一行了。
```

```
abefjus
qwertyuiop
再添加一行吧。
再加一行。

Press any key to continue
```

(2) 二进制文件

标识 `std::ios::binary` 代表二进制打开模式，并且实参指定文件被以二进制文件模式打开，而不是以文本文件模式打开。值得提醒的是，在 UNIX 系统中，二进制文件和文本文件是没有区别的；在 Windows 系统中，实参 `std::ios::binary` 是有意义的。文本文件和二进制文件之间可以使用以下方法进行转换：

- 1) 在程序向二进制文件写入新行符“\n”时，文件系统将写入单个新行符；多数平台上，新行符与换行符 (0x0a) 相同。
- 2) 在程序向文本文件写入新行符时，文件系统将写入两个字符：回车符 (0x0d) 和换行符 (0x0a)。
- 3) 程序从二进制文件中读取一个新行符时，文件系统将把单个新行符读取到存储器中。
- 4) 在程序从文本文件中读取一对回车/换行符时，文件系统将把一对字符转换成存储器中一个新行符。
- 5) 在程序从文本文件中读取单个新行符时，换行符前面是没有回车符，文件系统把一个新行符插入存储器。

由以上内容可知，上述方法不仅涉及新行符表示的转换，而且涉及文件位置的操作—`seeking` 和 `telling`。文本文件在存储器中的数据表示与磁盘上数据表示的长度不同，是存储器中单个新行符与磁盘之间回车符和换行符之间双向转换造成的。在文本文件和二进制文件有区别的平台，文本文件中的 `seeking` 和 `telling` 是不可靠动作。程序需要在文件内查找和告知当前文件的位置，程序应该使用二进制模式打开文件。无论在读文件时，还是写文件时，均须完整处理和换行有关的符号“\r\n”，而不是仅仅处理符号“\n”或 `std::endl`。

将数据存储于文件中时，二进制格式具体地说是计算机内部表示。计算机不是存储字符，而是存储这个值的 64 位 `double` 表示。对于字符而言，二进制表示与文本表示是一样的，即字符的 ASCII 码的二进制表示。对于数字而言，二进制表示与文本表示有很大差别。文本格式便于读取，使用编辑器或字处理器来读取或编辑文本文件，可以很方便地将文本文件在计算机系统之间实现传输。对于数字而言，二进制格式比较精确。不会有转换误差或舍入误差。以二进制格式保存数据的速度非常快，因为另一个系统使用另一种内部表示，无法将数据传输给该系统。同一系统上不同的编译器可能使用不同的内部结构表示。必须编写一个将数据转换成另一种数据的程序。

使用二进制格式，更为有利的是，可以实现整块地写入和读取数据。通常在实现文件的读取和写入时，多数使用的是 `write()` 和 `read()` 函数。下面引用 C++ Primer Plus (第五版) 中的例题，来说明二进制文件的读取和写入。

例 7-12

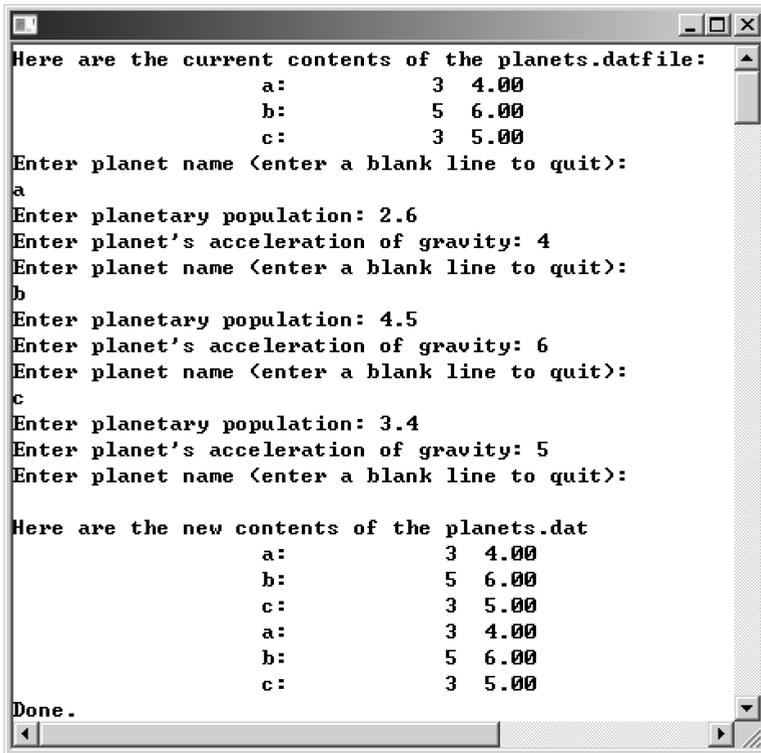
```

#include <iostream>
#include <fstream>
#include <iomanip>
#include <cstdlib>
#include <string>
using namespace std;
const string filename = "planets.dat";
inline void eatline() {while(std::cin.get() != '\n') continue;}
struct planet
{
    char name[20];
    double population;
    double g;
};
void main()
{
    planet pl;
    cout << fixed << right; //定点输出,右靠齐
    ifstream fin;
    fin.open(filename.c_str(),ios_base::in|ios_base::binary);
    if(fin.is_open())
    {
        cout << "Here are the current contents of the " << filename << "file: \n";
        while(fin.read((char* )&pl,sizeof(pl))) //读取
        {
            cout << setw(20) << pl.name << ":" << setprecision(0) << setw(12) << pl.population << setprecision
(2)
                << setw(6) << pl.g << endl;
        }
        fin.clear(); //清除所有标识位
        fin.close(); //关闭文件
    }
    ofstream fout(filename.c_str(),ios_base::out|ios_base::app|ios_base::binary);
    if(! fout.is_open())
    {
        cerr << "Can't open" << filename << " file for output: \n";
        exit(EXIT_FAILURE);
    }
    cout << "Enter planet name (enter a blank line to quit): \n";
    cin.get(pl.name,20);
    while(pl.name[0] != '\0')
    {
        eatline();
        cout << "Enter planetary population: ";
        cin >> pl.population;
        cout << "Enter planet's acceleration of gravity: ";
        cin >> pl.g;
        eatline();
        fout.write((char* )&pl,sizeof(pl));
    }
}

```

```
    cout << "Enter planet name (enter a blank line to quit): \n";
    cin.get(pl.name,20);
}
fout.close();
fin.clear(); //打开文件前,清除所有标识位
fin.open(filename.c_str(),ios_base::in|ios_base::binary);
if(fin.is_open())
{
    cout << "Here are the new contents of the " << filename << "\n";
    while(fin.read((char* )&pl,sizeof(pl)))
    {
        cout << setw(20) << pl.name << ":" << setprecision(0) << setw(12) << pl.population << setprecision(2)
            << setw(6) << pl.g << endl;
    }
    fin.close();
}
cout << "Done. \n";
return;
}
```

例 7-12 的执行效果如图 7-5 所示。



```
Here are the current contents of the planets.datfile:
                a:          3  4.00
                b:          5  6.00
                c:          3  5.00
Enter planet name (enter a blank line to quit):
a
Enter planetary population: 2.6
Enter planet's acceleration of gravity: 4
Enter planet name (enter a blank line to quit):
b
Enter planetary population: 4.5
Enter planet's acceleration of gravity: 6
Enter planet name (enter a blank line to quit):
c
Enter planetary population: 3.4
Enter planet's acceleration of gravity: 5
Enter planet name (enter a blank line to quit):

Here are the new contents of the planets.dat
                a:          3  4.00
                b:          5  6.00
                c:          3  5.00
                a:          3  4.00
                b:          5  6.00
                c:          3  5.00
Done.
```

图 7-5 例 7-12 的执行效果

4. 命令行处理技术

文件处理程序通常使用命令行参数来指定文件。尤其在原有的 MS-DOS 和 UNIX 系统，或者现有的 Windows 操作系统的 cmd 运行环境中。命令行参数是用户在使用程序时即时输入的，即在命令行中输入的参数。例如，

在 UNIX 操作系统中使用命令 `wc`，用于统计文件信息：

```
wc a.dat b.dat c.dat d.dat e.dat
```

此时，`wc` 是程序名，`a.dat`、`b.dat`、`c.dat`、`d.dat` 和 `e.dat` 是命令行参数传递给程序的文件名。对于 C/C++，在命令环境中运行的程序能够访问命令行参数。其使用方法是使用 `main()` 函数的参数。对于一个普通的 `main()` 函数，其形式多为：

```
int main(int argc, char * argv[])
```

其中参数 `argc` 是命令行中的参数个数，其中包括命令名本身。在上例中，所谓命令名本身即是指命令 `wc` 本身。参数 `argv` 变量为一个指针，指向 `char` 型指针。将参数 `argv` 看作指针数组，其中的指针指向命令行参数，`argv [0]` 是一个指针，指向存储第一个命令行参数的字符串的第一个字符，以此类推。`argv [0]` 是命令行中的第一个字符串。例如，在上例中，`argv [1]` 代表文件名 `a.dat`，以此类推。对于上例，

```
wc a.dat b.dat c.dat d.dat e.dat
```

`main()` 函数的各参数分别为 `argc` 等于 6，`argv [0]` 为 “`wc`”，`argv [1]` 为 “`a.dat`”，`argv [2]` 为 “`b.dat`”，`argv [3]` 为 “`c.dat`”，`argv [4]` 为 “`d.dat`”，`argv [5]` 为 “`e.dat`”。

通过在 `main()` 函数的起始位置填写部分语句，将 `main()` 函数的各参数进行输出。当然，命令行参数与命令行操作系统紧密相关。其他程序可能允许使用命令行参数。

例 7-13

```
#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;
void main(int argc, char* argv[])
{
    if(argc==1)
    {   cerr << "Usage: " << argv[0] << " filename[s] \n";           //输出程序的名称
        exit(EXIT_FAILURE);
    }
    else
    {   int count = argc;
        for(int i=0;i < count;i++)
        {   cout << "Paramter: " << argv[i] << endl;                //输出程序的参数
            }
        }
    }
```

例 7-13 的执行效果如图 7-6 所示。



```
管理员: 命令提示符
Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。

C:\Users\yanchy>f:

F:\>ex7-18 a.dat b.dat c.dat d.dat
Parameter: ex7-18
Parameter: a.dat
Parameter: b.dat
Parameter: c.dat
Parameter: d.dat

F:\>
```

图 7-6 例 7-13 的执行效果

7.6.2 随机访问

随机访问指的是直接移动到文件的任何位置，并直接访问文件任何位置的内容。随机访问常被用于数据库文件，程序维护一个独立的索引文件，该文件指出在主数据文件中的位置。程序可以直接跳到这个位置，读取其中的数据。如果文件由长度相同的记录组成，这种方法实现最简单。每条记录表示一组相关的数据。

对于输入流，和 stream 读写位置相关的成员函数如下：

```
tellg()、seekg(pos) 和 seekg(offset, rpos);
```

对于输出流，和 stream 读写位置相关的成员函数包括：

```
tellp()、seekp(pos) 和 seekp(offset, rpos);
```

上述函数分别以特殊字符 g 或 p 结尾，g 代表“get”，p 代表“put”。用于读写的位置函数定义于类 Basic_istream，用于改写的位置函数定义于类 Basic_ostream。然而，并不是所有输入流和输出流均能够支持读写定位。例如，cin、cout 和 cerr 就不支持读写定位。

当程序读写文件时，它将协调地移动输入缓冲区中的输入指针和输出缓冲区中的输出指针。在文件中，移动文件指针也需要一种方式。类 Fstream 继承了两个函数：seekg() 和 seekp()。前者将输入指针移到指定位置，后者将输出指针转移到指定的文件位置。将 seekg() 用于类 Ifstream 的对象，将 seekp() 用于类 Ofstream 的对象。

函数 seekg() 和 seekp() 可以接受一个绝对位置或一个相对位置。对于绝对位置，必须采用 tellg() 和 tellp()，会返回一个绝对位置，型别为 pos_type。在文件中，逻辑位置 and 实际位置可能并不相同，这和字节表示法有关系，是非常复杂的问题。

对于相对位置，偏移值可以和 3 个位置相关，通常这 3 个参数被定义于类 ios_base 中，型别均为 seekdir。这 3 个参数是 ios_base::beg、ios_base::cur 和 ios_base::end。

对于文件位置的偏移量，其类型都属于 off_type 型别，是 streamoff 的间接定义。由于 streamoff 是带正负号的整数型别，使用整数作为 stream 的偏移值。

无论是文件位置的绝对值，还是偏移值，其位置均在文件长度中有效，如果该位置超越了文件开头或文件结尾，均会导致未定义的行为。

`seekg()` 函数和 `seekp()` 的原型分别为：

```
basic_istream& seekg( pos_type _Pos);
basic_ostream& seekp( pos_type _Pos);
basic_istream& seekg( off_type _Off, ios_base::seekdir _Way);
basic_ostream& seekp( off_type _Off, ios_base::seekdir _Way);
```

第一种形式仅包含一个参数，参数代表距离文件开头特定距离的位置；第二种形式包含两个参数，第一个参数的位置描述是相对于第二个参数的，第二个参数可以使用 3 个值：

`ios_base::beg`、`ios_base::end` 和 `ios_base::cur`。

输入流和输出流还分别提供了 `tellg()` 函数和 `tellp()`，用于检查文件指针的当前位置，其返回值均为表示当前位置的 `streampos` 类型值。创建类 `Fstream` 的对象时，输入指针和输出指针将一前一后地移动，是分别独立的。对于同一个文件，如果使用类 `Ifstream` 对象来管理输入流，使用类 `Ofstream` 来使用输出流，则输入指针和输出指针将彼此独立地移动。此时，`tellp()` 和 `tellg()` 的返回值是不同的，是独立的。

例 7-14

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
void printfile(string filename)
{
    ifstream fin(filename.c_str());
    cout << fin.rdbuf(); //输出缓冲区内容
    fin.seekg(0); //文件指针放至文件头
    cout << fin.rdbuf(); //输出文件内容
}
void main(int argc, char* argv[])
{
    for(int i=1;i < argc;i++)
    {
        cout << "The contents of file " << argv[i] << " is below. \n" << endl;
        printfile(string(argv[i]));
    }
}
```

例 7-14 执行效果如图 7-7 所示。

```

管理员: 命令提示符
F:\Example\第7章\7-19>7-19 a.txt b.txt c.txt d.txt
The contents of file a.txt is below.
a.txt is the first file.
b.txt is the second file.
c.txt is the third file.
d.txt is the fourth file.
e.txt is the first file.
f.txt is the second file.
g.txt is the third file.
h.txt is the fourth file. a.txt is the first file.
b.txt is the second file.
c.txt is the third file.
d.txt is the fourth file.
e.txt is the first file.
f.txt is the second file.
g.txt is the third file.
h.txt is the fourth file. The contents of file b.txt is below.
b.txt is the second file.
c.txt is the third file.
d.txt is the fourth file. b.txt is the second file.
c.txt is the third file.
d.txt is the fourth file. The contents of file c.txt is below.
c.txt is the third file.
d.txt is the fourth file. c.txt is the third file.
d.txt is the fourth file. The contents of file d.txt is below.
d.txt is the fourth file. d.txt is the fourth file.

```

图 7-7 例 7-14 的执行效果

例 7-15

```

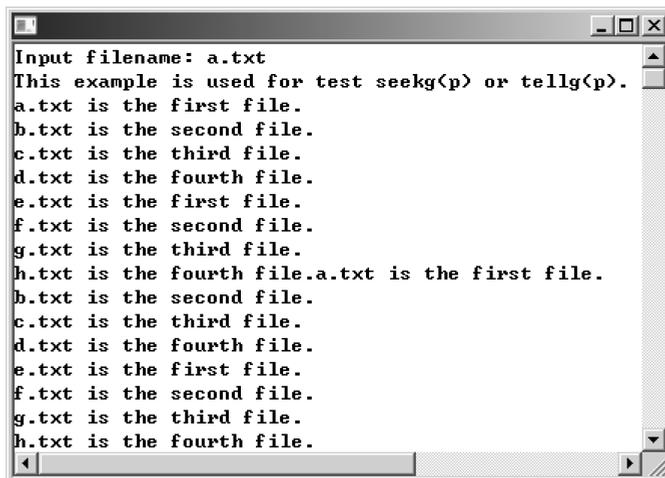
#include <iostream>
#include <fstream>
#include <iomanip>
#include <cstdlib>
using namespace std;
string filename;
void main(int argc, char* argv[])
{
    char file[20];
    cout << "Input filename: ";

    cin >> file;
    filename = string(file);
    if(filename.empty())
    {
        exit(EXIT_FAILURE);
    }
    cout << "This example is used for test seekg(p) or tellg(p). " << endl;
    fstream finout, fout;
    //ofstreambuf
    finout.open(filename.c_str(), ios_base::in | ios_base::out);
    if(finout.is_open())

```

```
{
    cout << finout.rdbuf(); //输出文件内容
    finout.seekg(0);
    cout << finout.rdbuf();
}
finout.seekg(0,ios_base::beg);
fout.open("b.txt",ios_base::out);
if(fout.is_open())
{
    fout << finout.rdbuf();
    fout.close();
}
cout << endl;
fout.open("c.txt",ios_base::out);
if(fout.is_open())
{
    finout.seekp(25,ios_base::beg);
    fout << finout.rdbuf();
    fout.close();
}
cout << endl;
fout.open("d.txt",ios_base::out);
if(fout.is_open())
{
    finout.seekp(51,ios_base::beg); //重新定位文件指针
    fout << finout.rdbuf();
    fout.close();
}
cout << endl;
}
```

例 7-15 的执行效果如图 7-8 所示。



```
Input filename: a.txt
This example is used for test seekg(p) or tellg(p).
a.txt is the first file.
b.txt is the second file.
c.txt is the third file.
d.txt is the fourth file.
e.txt is the first file.
f.txt is the second file.
g.txt is the third file.
h.txt is the fourth file.a.txt is the first file.
b.txt is the second file.
c.txt is the third file.
d.txt is the fourth file.
e.txt is the first file.
f.txt is the second file.
g.txt is the third file.
h.txt is the fourth file.
```

图 7-8 例 7-15 的执行效果



总结 本节使用两个例题说明了文件随机访问的方法。在两个例题中，文件均为文本文件，这样便于读者校对源代码和程序的输出结果。本节借这两个例题主要讲述了 `seekg()` 和 `seekp()` 函数的使用方法。因 `tellg()` 和 `tellp()` 的使用较为简单，故没有在例题中体现。

7.6.3 4 个类模板

本小节简要介绍 4 个类模板。前面已经使用过这 4 个类模板，但并没有对它们的全部功能和全部成员函数进行透彻的讲解和说明。

1. 类模板 `Basic_filebuf`

类模板 `Basic_filebuf` 是从类 `Basic_streambuf` 派生而来，属于名称空间 `std` 内。此类包含了 5 个类型声明：

```
typedef    charT char_type;
typedef    typename traits::int_type    int_type;
typedef    typename traits::pos_type    pos_type;
typedef    typename traits::off_type    off_type;
typedef    traits traits_type;
```

此类模板还包含了一个构造函数和一个析构函数：

```
basic_filebuf();
virtual ~basic_filebuf();
```

此类模板的成员函数有 3 个：`is_open()`、`open()` 和 `close()`。其原型分别为：

```
bool is_open();
basic_filebuf * open(const char * _Filename, ios_base::openmode _Mode);
basic_filebuf * close();
```

其中 `open()` 函数的返回值是指针类型。

此类还包括几个重载函数：`showmanyc()`、`underflow()`、`uflow()`、`pbackfail()`、`overflow()`、`setbuf()`、`seekoff()`、`seekpos()`、`sync()` 和 `imbue()`。

类 `Basic_filebuf` 使输入流和输出流同各自的文件关联起来。

`is_open()` 函数用于判断文件是否被打开。如果文件打开成功，函数返回 `true`；否则，函数返回 `false`。

`open()` 函数和 `close()` 实现文件或流的打开和关闭。

`showmanyc()` 函数可以实现判断出输入流中有多少字符可以被读取。

`underflow()` 函数在前面已经讲过。

`uflow()` 函数从输入流中读取当前字符，并返回为整数值形式。

`pbackfail()` 函数将字符放回输入流中。

`overflow()` 函数前面已经讲过。

`seekpos()` 函数使用绝对位置，随机访问文件。

`seekoff()` 函数使用相对位置，随机访问文件。

`sync()` 函数实现缓冲区和文件的同步。

`imbue()` 函数默认是不做任何事情。

2. 类模板 `Basic_ifstream`

类模板 `Basic_ifstream` 是从类 `Basic_istream` 派生而来的。同样，此类模板包括 5 个类型声明、两个构造函数和 5 个成员函数。它们的声明形式及功能如下：

```
typedef charT char_type;
typedef typename traits::int_type int_type;
typedef typename traits::pos_type pos_type;
typedef typename traits::off_type off_type;
typedef traits traits_type;
```

两个构造函数的声明形式为：

```
basic_ifstream();
explicit basic_ifstream(const char* s, ios_base::openmode mode = ios_
base::in)
```

5 个成员函数的作用为：

`rdbuf()` 函数是获取流缓冲区的指针。

`is_open()` 函数用于判断流是否被正确打开。

`open()` 函数用于打开文件。

`close()` 函数用于关闭文件和输入流之间的关联。

3. 类模板 `Basic_ofstream`

类模板 `Basic_ofstream` 是从类 `Basic_ostream` 派生而来的。同样，此类模板包括 5 个类型声明、两个构造函数和 4 个成员函数。它们的声明形式及功能如下：

```
typedef charT char_type;
typedef typename traits::int_type int_type;
typedef typename traits::pos_type pos_type;
typedef typename traits::off_type off_type;
typedef traits traits_type;
```

两个构造函数的声明形式为：

```
basic_ofstream()
explicit basic_ofstream(const char* s, ios_base::openmode mode = ios_
base::out)
```

4 个成员函数的作用为：

`rdbuf()` 函数是获取流缓冲区的指针。

`is_open()` 函数用于判断是否正确打开文件。

`open()` 函数用于打开文件，如果打开成功，将建立文件和流之间的关联。

`close()` 函数用于关闭流和文件之间的关联。

4. 类模板 `Basic_fstream`

类模板 `Basic_fstream` 是从类 `Basic_iostream` 派生而来的。同样，此类模板包括 5 个类型声明、两个构造函数和 4 个成员函数。它们的声明形式及功能如下：

```

typedef    charT char_type;
typedef    typename traits::int_type    int_type;
typedef    typename traits::pos_type    pos_type;
typedef    typename traits::off_type    off_type;
typedef    traits traits_type;

```

两个构造函数的声明形式为:

```

basic_fstream()           //可用于仅仅创建一个流对象
explicit basic_fstream(const char * s, ios_base::openmode mode = ios_
base::in | ios_base::out) //创建流对
//象的同时打开文件

```

4 个成员函数的作用为:

rdbuf() 函数用于获取流缓冲区的指针。

is_open() 函数用于判断是否正确打开文件。

open() 函数用于打开文件, 如果打开成功, 将建立该文件和流之间的关联。

close() 函数用于关闭流和文件之间的关联。

7.6.4 C 库中的文件存取功能概述

本小节将简单介绍 C 语言中关于文件存取的部分函数和宏。头文件 `<stdio.h>` 中涉及文件存取的宏如下:

BUFSIZ	用户自定义缓冲区大小
FOPEN_MAX	可以同时打开的最大文件数目
SEEK_CUR	文件指针的当前位置
TMP_MAX	tmpnam() 函数可以产生的最大文件数目
_IONBF	缓冲区类型
stdout	标准输出
EOF	文件结束标识
L_tmpnam	tmpnam() 函数产生的临时文件的文件名的长度
SEEK_END	文件末尾
_IOFBF	缓冲区类型
stderr	标准错误
FILENAME_MAX	文件名的最大允许长度
NULL	空
SEEK_SET	文件开始位置
_IOLBF	缓冲区类型
stdin	标准输入

头文件 `<stdio.h>` 中包含的数据类型如下:

FILE	文件结构
fpos_t	长整型数据类型
size_t	无符号整型

头文件 <stdio.h> 中声明的函数如下:

```
clearerr(FILE* stream) //复位流的错误标识
fgets() //从流中获取一串字符
fscanf //从流中按格式读取
gets //从标准输入流中获取一行数据
rename //重新命名文件
tmpfile //创建临时文件
fclose //关闭文件
fopen //打开文件
fseek //随机访问文件
perror //打印错误信息
rewind //重新定义文件指针到文件起始位置
tmpnam //创建临时文件名
feof //判断是否在文件结尾
fprintf //向文件格式输出
fsetpos //设置流位置标识
printf //格式化输出
scanf //格式化读取
ungetc //将字符放回流缓冲区中
ferror //检查流中的错误
fputc //向文件中输出字符
ftell //判断文件指针位置
putc //输出字符
setbuf //将指定的存储区域和流缓冲区位置关联
vfprintf //向文件中写入
fflush //清空缓冲区
fputs //向文件中输出一串字符
fwrite //向文件中写入数据
putchar //输出字符
setvbuf //控制流缓冲区和缓冲区的大小
vprintf //格式化输出
fgetc //从文件中读取一个字符
fread //从文件中读数据
getc //从标准输入中获取一个字符
puts //向标准输出中输出一行数据
sprintf //格式化字符串,输出数据至字符串
vsprintf //格式化输出字符串
fgetpos //获取流的文件位置标识
freopen //重设文件指针
```

```
getchar           // 获取字符  
remove           // 删除  
sscanf           // 从字符串中格式化输入
```

7.7 小结

本章主要讲述输入流和输出流的相关内容。其中讲述了 `IOStream` 流的基本内容和流中的格式化方法，并对流缓冲区进行了详细的讲解。此外，本章还介绍了基于字符串的流和基于文件的流，并重点讲述了基于文件的流的使用方法。本章是本书的学习重点之一，读者应反复阅读，以熟练掌握此部分内容。

第 8 章

异常处理类模板

本书第 1 章简要提到了异常处理的内容。

异常处理是 C++ 的一种特性。异常处理会使程序按照有序、有组织和一致的方式截取和处理异常条件——error。异常处理允许使用一段程序来检测和分析错误条件，使用另一段程序来处理错误。检测错误的代码可能不知道错误处理的策略，代码可以是库中的类和函数，也可以是包含其他种类的代码。通常错误来源于程序的隐含错误或致命错误（无效等）。

函数库不可能知晓所有能检测出的可能发生的异常。函数将错误汇报给正在使用的程序，需要一定的前提。总而言之，异常处理是一个较复杂的过程。

8.1 异常的概念和基本思想

8.1.1 异常的概念

异常处理（Exceptional Handling），又称为错误处理。异常处理分离了接收和处理错误代码，这样既帮助程序员理清了思绪，又增强了代码的可读性，便于维护者的阅读和理解。

异常处理功能提供了处理程序运行时出现的任何意外或异常情况的方法。异常处理使用 try、catch 和 finally 等关键字来尝试可能未成功的操作、处理失败以及在事后清理资源。

异常处理通常是为了防止未知错误产生所采取的处理措施。异常处理使程序员不必绞尽脑汁地去考虑各种错误，为处理多种错误提供了有效的方法，使编程效率大大提高。

C++ 语言中，异常通常由 throw 关键字的应用程序代码抛出。

异常处理通常有两种理论模型：一种称为“终止模型”（Java 与 C++ 所支持的模型）。在这种模型中，所有错误被假设是非常关键的，错误发生时，程序通常无法返回到异常发生之处继续执行。异常被抛出后，说明产生的错误已经无法挽回。另一种是“恢复模型”。异常处理的功能是修正程序产生的错误，并重新尝试调出问题的方法（或函数），且假定二次调用会成功。

对于“恢复模型”，通常异常处理完毕之后能继续执行程序。抛出异常更像是对方法（或函数）的调用。如果把 try 块放在 while 循环中，那么就可以不断地进入 try 块，直到截获可能的异常发生或者获得程序员满意的结果。

由以上描述可知，“恢复模型”更能吸引广大程序员。多数操作系统是支持恢复模型的异常处理的。但是完美的处理不一定是最佳选择，多数程序员更喜欢“终止模型”。因为对于“恢复模型”而言，程序必须关注异常抛出的地点，这会导致无法包含或编写依赖于抛出位置的非通用性代码，从而增加代码编写和维护的难度。

上述对异常的描述可归结为最简单的一句话：异常会使对某个函数的调用动作中止，而

该函数正是异常发生之处。异常处理机制有能力将某个对象作为参数，回传给函数调用者，但并不是使用“回调函数”（回调函数属于信号处理机制范畴）。异常处理和信号处理是不同的概念。

8.1.2 异常的分类

通过异常处理，C++ 标准程序库可以在“不污染”函数接口的情况下处理异常。一旦遭遇异常，通过“抛出异常”停止正常的处理过程。语言本身支持的或标准程序库所抛出的所有异常均派生自基类 `exception`。

类 `exception` 的声明：

```
class exception {
public:
    exception() throw();
    exception(const exception& right) throw();
    exception& operator = (const exception& right) throw();
    virtual ~exception() throw();
    virtual const char * what() const throw();
};
```

其中包含了两个构造函数，这两个构造函数全部包括异常描述（后面讲述），还包括了一个右侧赋值操作符、一个虚析构函数和一个 `what()` 虚函数。

类 `Exception` 是多个标准异常类的基类，与其他标准异常类共同构成类的体系。标准异常类通常分为以下 3 种：

- 1) 语言本身支持的异常。
- 2) C++ STL 所抛出的异常。
- 3) 程序作用域之外发出的异常。

第一种异常是语言本身支持的异常。此类异常多用于支撑某些语言的特性。此类异常处理不是 STL 的一部分，而是核心语言的内容。一旦操作失败，会抛出异常。例如，全局操作符 `new` 一旦操作失败，会抛出 `bad_alloc` 异常。在较复杂的程序中，尤其在大量使用操作符 `new` 的程序中，此类异常可能会随时发生，是一种非常重要的异常。程序执行时，当加诸于 `reference` 身上的“动态型别转换操作”失败时，动态 `dynamic_cast` 通常会抛出 `bad_cast` 异常。对于型别辨识过程，赋予 `typeid` 的参数一旦是零或空指针，`typeid` 会抛出 `bad_typeid` 异常。值得一提的是，对于发生非预期的异常，`bad_exception` 异常会即时处理。此时通常 `bad_exception` 会调用 `unexpected()` 函数。

类 `bad_alloc` 的声明形式如下：

```
class bad_alloc : public exception
{
};
```

类 `bad_cast` 的声明形式如下：

```
class _CRTIMP bad_cast : public exception {
public:
    bad_cast(const __exString& what_arg) : exception(what_arg) {}
};
```

第二种异常是C++标准程序库所发生的异常。C++标准程序库异常总是派生自类 `logic_error`。从理论上讲，能够通过一些手段使程序避免逻辑错误。逻辑错误通常指的是包括违背逻辑前提或违反类的不变性。C++标准程序库提供 4 个逻辑错误类别，分别是：`invalid_argument`、`length_error`、`out_of_range` 和 `domain_error`。

类 `logic_error` 的声明形式如下：

```
class logic_error : public exception {
public:
    logic_error(const string& message);
};
```

上述类 `logic_error` 的构造器的参数 `string& message` 用于描述异常的信息，以便于在使用 `what()` 函数时，返回该 `message`。下同。

类 `invalid_argument` 通常表示无效参数，例如使用 `char` 类型对 `bitset` 初始化。该类的声明形式如下：

```
class invalid_argument : public logic_error {
public:
    invalid_argument(const string& message);
};
```

类 `length_error` 通常指某个行为“可能超越了最大极限”，例如字符串中字符数目过多。该类的声明形式如下：

```
class length_error : public logic_error {
public:
    length_error(const string& message);
};
```

类 `out_of_range` 通常指参数值“不在预期范围内”，例如在 `array` 型容器中或字符串中采用错误的索引。该类的声明形式如下：

```
class out_of_range : public logic_error {
public:
    out_of_range(const string& message);
};
```

类 `domain_error` 通常指专业领域内的错误。该类的声明形式如下：

```
class domain_error : public logic_error {
public:
    domain_error(const string& message);
};
```

STL 的 I/O 部分提供了名为 `ios_base::failure` 的特殊异常。数据流因错误或到达文件末尾而发生状态改变时，可能会抛出异常。

由于标准程序库会用到语言特性及客户所写的程序代码，可能间接抛出任何异常。尤其是何时分配存储空间，都有可能抛出 `bad_alloc` 异常。

标准程序库任何具体的实例化类及其对象，都可能提供额外的异常类别。使用非标准类

别将导致程序无法移植。必要时需要非常痛苦地修改源程序，因此使用标准异常是非常必要的。

第三种异常是程序作用域之外发生的异常。派生自类 `runtime_error` 的异常，用来指出“不在程序范围内，且不容易回避”的事件。C++ 标准程序库对“执行期错误”提供了 3 个类：

类 `range_error` 用以实现指出内部计算时发生区间错误。该类的声明形式如下：

```
class range_error : public runtime_error {
public:
    range_error(const string& message);
};
```

类 `overflow_error` 用以实现指出算术运算发生上溢位。该类的声明形式如下：

```
class overflow_error : public runtime_error {
public:
    overflow_error(const string& message);
};
```

类 `underflow_error` 用以实现指出算术运算发生下溢位。该类的声明形式如下：

```
class underflow_error : public runtime_error {
public:
    underflow_error(const string& message);
};
```

类 `runtime_error` 的声明形式如下：

```
class runtime_error : public exception {
public:
    runtime_error(const string& message);
};
```

使用标准异常类时，需要包含头文件 `<exception>`、`<new>`、`<typeinfo>`、`<ios>` 和 `<stdexcept>`。头文件中 `<exception>` 中包含了类 `exception` 和类 `bad_exception` 的定义声明。类 `bad_alloc` 在头文件 `<new>` 中声明，类 `bad_cast` 和类 `bad_typeid` 在头文件 `<typeinfo>` 中声明。`ios_base::failure` 声明于头文件 `<ios>` 中。其他多数异常声明于头文件 `<stdexcept>` 中。

8.1.3 异常的捕捉和处理

在 C 语言中，通常采用两种方法处理异常：①对每个函数调用错误测试；②调用 `setjmp()` 和 `std::longjmp()` 函数来截取错误条件。

第一种方法是用类似全局宏 `errno` 的变量、零或错误函数的返回值来报告错误种类。这种方法可靠但非常烦琐。程序员无法全部考虑各种错误的可能性。其中 `errno` 包含在头文件 `<cerrno>` 中，其声明形式为：

```
namespace std {
    using ::errno;
}
```

第二种方法是用 `setjmp()` 和 `std::longjmp()` 函数。这两个函数的使用更能接近于 C++ 异常处理所能达到的效果，能够依次、自动地把堆栈还原至函数调用层次结构中较高层位置所记录的状态。`setjmp()` 和 `std::longjmp()` 函数会截取和处理不需要瞬时终止程序的错误条件。程序使用 `setjmp()` 指定返回的位置，并使用 `std::longjmp()` 返回到该位置。

上述两种异常处理办法均不能完好地解决异常处理问题。

在 C++ 中，通常使用 `try` 块和 `catch` 块进行异常的捕捉和处理。`try` 块中可以执行一些 C++ 函数，能检测出执行过程中的异常，并能在出错之后恢复。`try` 块的声明形式如下：

```
try{
    //C++ statements
}
```

在 `try` 块之外执行的代码不能检测出或处理异常情况，假如在某个 `try` 块外面的代码发生异常，程序一般会束手无策，无法正确处理。`try` 块还可以嵌套其他 `try` 块，一旦发生意外，`try` 块会调用其他能够检测和处理异常的函数。

`catch` 块用于处理异常。理论上 `catch` 块出现在 `try` 块之后，并且 `catch` 块包含形参。`catch` 块的声明形式如下：

```
catch(type argument)
{
    // error - handling code
}
```

`catch` 块可以“捕捉”到的由 `try` 块中某个 `throw` 语句抛出的异常。`try` 块之后可以列出多个具有不同形参列表的处理函数。

例如，

```
try{
}
catch(int err){
}
catch(char* msg){
}
```

不同的 `catch()` 块通过形参列表中的类型区分，因此不必给 `catch` 形参列表中的形参命名。如果形参被命名，即声明了具有名字的对象，异常检测代码会传递形参值；如果形参没有被命名，将丢弃异常检测代码中的 `throw` 语句的实参。

`throw` 语句是用来抛出异常的。为了检测出异常并跳转到某个 `catch` 处理函数，C++ 函数发出 `throw` 语句，语句中包含的数据类型与适当的 `catch` 处理函数的形参列表相匹配。例如，

```
throw "An error has occurred";
```

该 `throw` 语句跳转到具有 `char *` 形参列表的 `catch` 异常处理函数。`throw` 语句还会还原堆栈，通过调用对象的析构函数清除在 `try` 块内声明的所有对象。之后，`throw` 会调用匹配的 `catch` 处理函数，传递形参对象。



提示

若 `catch()` 的形参为省略号（即 `catch (...)`），则 `catch` 处理函数能捕捉到所有未曾被捕捉的异常。通常 `catch (...)` 块放在所有 `catch` 块的最后。

下面用一个示例讲述异常处理的问题。

```
try{
    throw "error information".           //抛出异常信息
    throw errorcode;                    //抛出错误代码
}
catch(...)
{
    ...
}
```

8.1.4 资源管理

当函数申请了其所占有的资源时，为保证系统的运行，必须正确地释放此类资源，即通常所说的“正确释放”，即要求申请资源的函数在返回其调用者之前完成资源的释放。例如，一旦在调用 `fopen()` 函数之后，调用 `fclose()` 函数之前，程序运行时出了问题，会出现异常退出，但此时却没有来得及调用 `fclose()` 函数。即使是常规的 `return` 语句实现问题，导致 `use_file()` 退出而没有关闭该文件。

在使用时，程序员可以将所有出现异常的语句都包裹在一个 `try` 块内，在关闭文件时重新抛出这个异常。此方法过于烦琐，且代价高昂，容易诱发各类错误，让程序开发人员深感厌恶。所幸还有一种更好的解决办法。

类的对象通常由构造函数创建，并由析构函数销毁。只要适当地利用带有构造函数和析构函数的类对象，即可以处理此种资源的申请和释放问题。例如，在使用文件时，程序员可以定义文件类，其中包含其构造函数和析构函数。一旦该类的对象在其作用域结束时将被销毁，析构函数会关闭相应的文件（当然，需要在析构函数中添加相应的代码）。无论函数是正常结束，还是非正常退出，析构函数总是会被调用的。异常处理机制使程序员可以从主要算法中删去处理错误的代码，从而使得代码更为简单。

1. 构造函数和析构函数的使用

利用局部对象管理资源的技术通常被称为“资源申请即初始化”。该技术依赖于构造函数和析构函数的性质以及这两个函数与异常处理的相互关系。

当某个对象的构造函数执行完毕时，此对象才被认为已经明确建立。之后，堆栈回退时才为该对象调用析构函数。对于由子对象组成对象的构造函数，将持续到所有子对象的构造函数均完成，其构造函数才执行完毕。对于数组的构造函数，将持续到数组所有元素均构造完成，其构造函数才执行完毕。

对象的构造函数努力去保证对象能够完整、准确地创建起来。如果目标无法实现，书写良好的构造函数会将系统的状态恢复到对象构造之前的情况。理想状况是：按朴素的方式写出构造函数能达到的可能情况，不使它们处理的对象处于某种“片面构造”的状态。使用“资源申请即初始化”技术，可以实现完全构造类对象。

在资源申请简单模型的支持下，编写构造函数的人不必去专门写显式的异常处理代码。这样，程序员就不必去追踪其他情况，但这也带来了很多问题。例如，大量的初始化工作需要放在构造函数中才更加保险，才不会导致剧烈的异常给整个程序或操作系统带来致命的

冲击。

2. STL 的 auto_ptr 类

类 auto_ptr 的对象可以使用指针去初始化，且能够以指针同样的方式间接访问。在 auto_ptr 类对象退出作用域时，所指的对象将被隐式地自动删除。类 auto_ptr 型指针具有与常规指针不一样的复制意义：在将一个 auto_ptr 型指针复制给另一个指针之后，原来的 auto_ptr 不再指向任何东西。复制 auto_ptr 将导致对其自身的修改，但 const auto_ptr 不可以被复制。

类 auto_ptr 的声明形式为：

```
template <class Type> class auto_ptr
```

使用类 auto_ptr 时，需要包含头文件 <memory>。多个 auto_ptr 拥有同一个对象的效果是无定义的。最可能的情况是使该对象无法被彻底删除。

任何程序都应具备从破坏中恢复的能力。不可能所有程序都需要付诸很多的努力——使用“资源申请即初始化”技术、auto_ptr 和 catch 块等技术去保护该程序。对于一些简单程序，一旦发生异常，最有效的办法是使该进程夭折。此时，程序会释放所申请的所有资源，重新运行程序。

3. 构造函数中的异常和 new()

在类的构造函数中，如果大量使用 new() 函数分配存储空间，一旦发生异常，能够完整地释放这些存储空间吗？回答是肯定的。

但如果在程序中使用 new() 函数，那么在释放该内存空间时，即要使用 delete() 函数。

异常是从构造函数报告的问题中提供解决方案。由于构造函数不能顺利返回独立的值供调用程序检查，通常的解决办法有以下 4 种：

- 1) 返回处于错误状态的对象，相信用户有办法检查其状态。
- 2) 设置非局部变量指出创建失败，相信用户能检查该变量。
- 3) 在构造函数中不做初始化，依赖用户第一次使用对象之前调用某个初始化函数。
- 4) 将对象标记为“未初始化”，让该对象调用的成员函数完成实际的初始化工作，并允许该函数在初始化失败时报告错误。

异常处理机制使构造失败信息能从构造函数内部传递出来。程序员可以安排一些 throw()，例如，某类的对象遭遇超量存储时，可以在程序中使用判断语句，一旦遭遇此类情况，即可将其抛出。尤其是处理那些多种资源的构造函数，“资源申请即初始化”技术是最安全最优秀的方法。从根本上讲，该技术把处理多种资源的问题归结为一种反复应用处理单一资源技术的问题。

如果类的成员在初始化时抛出异常，按照默认的约定，该异常将传至类的构造函数位置。构造函数本身通常会将完整的函数体包括在 try 块内，函数本身是无法捕捉这些异常的。

另外，构造函数是无法复制的。复制构造函数可以通过抛出异常的方式发出失败信号。在此种情况下，实际上没有创建对象。例如，对于某容器类，在复制构造函数时需要分配存储并复制元素，可能导致异常的抛出。抛出异常之前，复制构造函数需要释放已经申请到的所有资源。尤其是复制赋值操作，会申请资源，也可能通过抛出异常而结束。但在抛出异常之前必须保证每个操作对象均保持在某种合法状态。

4. 析构函数中的异常

从异常处理的角度来讲，析构函数中也可能发生异常。通常此时析构函数会在两种情况

下被调用:

1) 正常调用。作为某个作用域正常退出的结果, 或者作为 delete 操作的结果。

2) 在异常处理中被调用。在堆栈回退过程中, 异常处理机制退出作用域时, 其中包含析构函数的对象。

对于第二种情况, 析构函数抛出异常是不被允许的。如果真的在析构函数中发生了异常的抛出, 即是异常处理机制的重大失败。在析构函数中抛出异常而导致析构函数退出程序, 这违背了 STL 的基本原则。

为防止上述情况的发生, 程序员可以在析构函数中使用 try 块和 catch 块。例如, 对于类 C, 其析构函数可以如下编写:

```
X::~X()
{
    try{
    }
    catch(...) {
    }
}
```

如果有异常被抛出之后未被捕捉, STL 中的 `uncaught_exception()` 函数会返回 true。此函数多用于析构函数中, 调用时可获取合适的返回值, 从而确定对象是被正常销毁还是作为堆栈回退的一部分。

5. 资源耗尽

申请资源一旦失败, 会导致很大的问题。例如, 程序在打开文件时文件不存在, 或者耗尽了所有自由存储空间。通常会有两种解决方式:

1) 唤醒机制。请求调用某个程序, 纠正问题, 之后继续执行。

2) 终止策略。结束当前计算, 并返回某个调用程序。

对于第一种方式, 预先需要准备一个调用程序来帮助处理某段未知的代码中出现的资源申请问题。对于第二种方式, 调用程序必须准备好应付某个资源申请失败的情况。第二种形式更为简单一些, 也更为实用。第二种方式能够使系统保持抽象层次之间的较好的隔离。采取终止策略时, 终止的不是整个程序, 而是其中个别的计算。“终止”是传统的描述策略的术语, 表示从“失败”的计算返回到与某个调用过程相关的错误处理器, 而不试图去修复坏的状况, 之后从检查出问题的那个位置继续。

在C++中, 唤醒模型通常由函数调用机制支持, 终止模型由异常处理机制支持。要抛出异常, 就必须存在一个被抛出的对象。每个C++实现时都要求保留足够的存储空间, 在存储空间耗尽时, 需要抛出 `bad_alloc()`。由于抛出其他对象而导致存储耗尽的情况也是会发生的。

8.1.5 异常和效率

在没有抛出异常之前, 异常处理的实现在运行时没有产生任何额外开销。抛出异常并不比调用函数的开销大。在完成这些的同时, 要维持与C语言在调用序列、排错规范等方面的兼容性, 而又不引起显著的额外存储开销, 这是非常困难的。简单使用那些能代替异常的东西是需要代价的。在多数的大规模程序中或系统软件中, 更多的代码(甚至超过一半)

实际上是用于错误处理和异常处理的。对于下述形式的代码：

```
void g(int );
void f()
{
    string s;
    g(1);
    g(2);
}
```

如果 `g()` 函数会抛出异常，那么 `f()` 函数必须包含部分代码，使异常发生时能够得到即时处理。例如，确保字符串变量 `s` 被正确销毁。如果 `g()` 函数不采用抛出异常的方式，也会得到使用另一种方式报告发生的错误。与之相较，使用常规方式写出的处理错误的代码需要包含大量的 `if` 判断和 `return` 语句。就像排雷一样，走一步就要判断一步。

异常描述可能非常有助于改进所生成的代码。但是，传统的 C 函数不会抛出异常的。大部分程序中，每个 C 函数均可用“空抛出”描述 `throw()` 声明，即

```
void function () throw()
```

上述函数定义中 `throw()` 的参数表为空，说明此函数不能抛出任何异常。对于标准 C 库函数只有很少的几个函数可以抛出异常，例如 `atexit()` 和 `qsort()`。

```
int atexit ( void ( __cdecl * func ) ( void ) );
```

`atexit()` 函数指定 `func()` 函数在程序终止时被调用。如果调用成功，函数返回 0；如果调用失败，函数返回非零。进程函数 `func()` 必须返回 `void`，并且不允许包含参数。

`qsort()` 函数在前面讲述算法的章节已有介绍。

利用这些函数可以生成更优质的代码。值得注意的是，在为某个 C 函数空异常描述 `throw()` 之前，要思考一下该函数是否有必要抛出异常。

8.1.6 异常的描述

1. 异常描述

抛出或捕捉异常对某个函数与其他函数的关系产生影响。抛出异常集合作为函数声明的部分就具有非常重要的价值（前面已经简单提过空抛出的问题）。例如，

```
void function(int a) throw (x2, x3)
```

其中，函数 `function (int a)` 可能抛出两个异常类 `x2` 和 `x3` 的对象以及从这两个异常类型派生的异常，不会抛出其他类型异常。若函数描述了可能抛出的异常，即有效地为其调用者提供了一种保证。在执行过程中，函数试图废止所做的保证，此意图会被转换成一个对 `std::unexpected()` 的调用。`unexpected()` 的默认意义是 `std::terminate()`，一般会转而调用 `abort()` 函数。

在功能方面：

```
void function throw (x2, x3)
{
    //部分代码
}
```

上述代码等价于：

```
void function()
try
{
    //代码
}
catch(x2)
{
    throw;
}
catch(x3)
{
    throw;
}
catch(...)
{
    std::unexpected();
}
```

上述代码的优点在于函数的声明属于界面（即接口），而界面是函数的调用者明显可以看到的。函数的定义通常不是可用的内容。即使有机会访问所有库的源代码，程序员也不希望经常去查看库函数的源代码。另外，带有异常描述的函数要比使用 try 块和 catch 块简洁得多。

如果函数声明中不包含异常描述，假定该函数可能抛出任何其他异常，但又不希望抛出某些异常，可以使用空抛出声明形式（前面已提到）。例如，

```
int function() //可能抛出异常
int function() throw() //不会抛出任何异常
```

2. 异常描述检查

编译过程可能会遗漏违反界面描述的情况。编译时的检查可以完成大部分工作。异常描述的方式应该是假定函数将要抛出或可能要抛出的所有异常。如果函数的声明中包含了异常描述，该函数的每个声明（或定义）都必须包含完全一致的异常类型集合的异常描述。

若在函数声明时包含了 throw()，而在函数的定义时没有包含 throw()，则是错误的。例如，

```
int function() throw(std::bad_alloc)
...
int function()
{
    ...
}
```

以上代码就是错误的。

上述对异常描述进行了较详细的讲解，读者也没必要为此恐慌。当某处发生异常时，程序员不必强制性地相关的异常描述做完全的更新，不必强制性地要求重新编译受影响的代

码。大多数系统应该能够在一种部分更新的状态中依靠对未预期异常的动态检查照常工作。对于大规模系统的维护是必不可少的，但大规模更新的代价极其昂贵。

尤其，当需要覆盖一个虚函数时，函数所带的异常描述必须至少是和那个虚函数的异常描述一样受限的。

总而言之，异常处理是非常复杂的过程。读者只要掌握最简捷的使用形式即可，确保在使用过程中万无一失即可，没有必要去追求使用形式的多样性和复杂性。

3. 未预期的异常

异常描述可能导致调用 `unexpected()` 函数。如果不希望出现此种调用的情况，可在程序调试期间，通过细心地组织异常和界面描述，尽力避免此种调用，即尽力拦截对 `unexpected()` 的调用，并使之无害化。

`unexpected()` 函数的声明形式如下：

```
void unexpected();
```

C++ 标准要求表明：如果被抛出的异常不包括在 `throw` 列表中，`unexpected()` 会被调用。当前的实现不支持这种情况，例如直接调用 `unexpected()` 时，是通过调用该函数的句柄。如果 `unexpected()` 函数在程序中被直接调用时，`unexpected()` 的句柄可以通过 `set_unexpected()` 设置。其使用方法可参考例 8-4。`unexpected()` 处理器不可能返回至其调用者，它可能通过以下条件终止执行：

1) 异常描述中的类型列表中类型的对象，或任意类型的对象被抛出时，`unexpected()` 处理器可以在程序中直接调用。

2) 抛出 `bad_exception` 类型的对象。

3) 在调用 `terminate()`、`abort()` 或者 `exit()` 过程中。

`set_`函数 `unexpected()` 的声明形式如下：

```
unexpected_handler set_unexpected( unexpected_handler Pnew ) throw();
```

参数 `Pnew` 是指定的被调用的函数，函数返回值是原有的被调用的函数。

通常，设计良好的子系统 `Y` 的所有异常均从类 `Yerr` 中派生。例如，

```
class Some_Yerr: public Yerr{/* ..... */}
```

包含如下声明的函数：

```
void function() throw (Xerr, Yerr, exception);
```

当把所有 `Yerr` 传递给它的调用者。特别是 `function()` 可以通过将 `Some_Yerr` 传给其调用者的方式去处理它。`function()` 函数中的 `Yerr` 不会触发 `unexpected()`。由标准库中抛出的所有异常均是由 `exception` 派生出的。

4. 异常映射

若某个异常一旦发生即终止程序，此种策略无疑是过于严厉的。此时，需要将 `unexpected()` 的行为修改为其他易于接受的方式。

最简便的方法是将标准库异常 `std::bad_exception` 加入某个异常描述中。此时，`unexpected()` 会抛出一个 `bad_exception`，而不去调用某个难以处理的函数。例如，

```
class X{};
class Y{};
void function() throw(X, std::bad_exception)
{
    throw Y();          //抛出“bad”异常
}
```

异常描述将捕捉未预期的异常 Y，之后再抛出一个 `bad_exception` 类型的异常。

类 `bad_exception` 仅仅是提供一种机制，并没有调用 `terminate()` 那样“冷酷”，但丢失了引起问题的那个异常的所有信息。

8.1.7 未捕捉的异常

前面已经讲过，如果某个异常未被捕捉，会调用 `std::terminate()` 函数。当异常处理机制发现堆栈损坏时，或者在由某个异常抛出而导致的堆栈回退过程中，被调用的析构函数企图通过抛出异常而退出时，都会调用 `std::terminate()` 函数。通俗地讲，未捕捉的异常是没有为其准备 `catch()` 处理函数的异常，或是 `throw()` 所执行的析构函数抛出的异常。此异常将造成 `std::terminate()` 函数被调用，该函数随即调用 `std::abort()` 以终止程序。

未预期的异常由 `set_unexpected()` 确定的 `_unexpected_handler` 处理。类似情况是，对未捕捉的异常响应是由 `_uncaught_handler` 确定，而它可由 `<exception>` 中的 `std::set_terminate()` 设置：

```
typedef void(* terminate_handler)();
terminate_handler set_terminate (terminate_handler);
```

返回值给出的是通过 `set_terminate()` 设置的前一个函数。

提出 `terminate()` 的原因是：在少数情况下，必须能用较粗糙的错误处理技术终止异常处理过程。例如，`terminate()` 可能被用于结束某个进程，甚至用于重新初始化一个系统。采用 `terminate()` 的意图是作为一种严格限制，当由异常处理机制所实现的错误恢复策略失败之后，应该进入另一个容错策略层次，即可应用之。

通常，按默认的规定，`terminate()` 会调用 `abort()` 函数。对大部分用户而言，这是一种正确的选择。特别是在排除错误阶段。

`_uncaught_handler` 被假定是不会返回其调用者的，否则 `terminate()` 将调用 `abort()`。`abort()` 函数表示从程序中非正常退出。若使用 `exit()` 函数终止程序，其返回值可以向外围系统表明程序是正常结束还是非正常结束。

在程序因为未捕捉的异常而终止时，是否调用有关的析构函数将由具体实现决定。不调用析构函数是至关重要的，这将使程序能够从排错系统中重新唤醒。在其他系统中，系统的结构决定在检索异常处理器的过程中，不调用析构函数是不可能的。

如果要在发生未捕捉异常时保证进行彻底的清理工作，可以在所有真正需要关注的异常处理器之外，为 `main()` 函数添加捕捉一切的处理程序。例如，

```
int main()
try{
    // code
```

```
}  
catch(std::range_error)  
{  
    // code  
    cerr << "range error : not agan! \n";  
}  
catch(std::bad_alloc)  
{  
    cerr << "new ran out of memory \n";  
}  
catch(...)  
{  
    //...  
}
```

上述形式的代码即可捕捉程序可能发生的所有异常，除在全局变量的构造和析构中抛出的异常之外，没有办法捕捉全局变量初始化期间抛出的异常。对于非局部静态对象初始化中出现的 throw，获取控制的唯一办法是通过 set_unexpected()，即应该尽可能地避免全局变量的其他原因。

一旦异常被捕捉，通常无法确定其抛出位置，这也意味着信息的丢失。在有些 C++ 开发环境中，对某些程序和某些开发人员而言，信息的丢失会导致无法捕捉“程序设计本身无法恢复的异常”。



总结 8.1 节主要讲述了异常及其相关的部分概念和异常处理的基本原理。请读者认真阅读，对异常处理能有较清楚的认识即可。任何知识的学习均是循序渐进的过程，开始最重要的是了解和理解其基本概念和基本原理，随着今后实践机会和实践经验的逐渐增加，对异常处理的认识会更加深刻。同样，对于 C++ STL 的其他内容也是如此，都需要经历循序渐进的过程。

8.2 异常类及几个重要问题

8.2.1 类 exception

在 STL 中，类 exception 除了构造函数和析构函数之外，仅有一个虚成员 what() 函数。该函数用以获取“型别本身以外的附加信息”，其返回值是一个以 null 结束的字符串。类 exception 的声明形式如下：

```
class exception {  
public:  
    exception() throw();  
    exception(const exception& right) throw();  
    exception& operator = (const exception& right) throw();
```

```
virtual ~exception() throw();
virtual const char * what() const throw();
};
```

在上述类声明中，其 3 个构造函数和 1 个析构函数均被设置了空抛出（`throw()`）。其虚成员 `what()` 函数也被设置了空抛出。

`what()` 函数返回的字符串可以被转换为 `wstring`，以便于被显示。`what()` 返回的 C-string 在所属的异常对象被摧毁后即不再有效。`what()` 函数是唯一被用来描述异常种类的虚函数。唯一可能实现的另一种异常评估手段是根据异常的精确型别，自己得出结论。最简单的例子如下：

```
try{
//...
}
catch(const std::exception& error){
std::cerr << error.what() << std::endl;
}
```

前面已经说过，C++ 异常的主要目的是为设计容错程序提供语言级支持，即异常使得在程序设计中包含错误处理功能更简便，避免事后采取“冷酷”的错误处理方式。异常灵活性和相对方便性可以使程序员在条件允许的情况下，在程序中加入错误处理功能。总而言之，异常是一种特性，类似于类，可以改变程序员的编程方式。

头文件 `<exception>` 中定义了异常类，C++ 可以将其作为其他异常类的基类。程序会引发异常（`exception`），但由于 `what()` 是虚函数，类 `exception` 派生的自定义类可以重新定义该函数。例如，

```
#include <exception>
class bad_cl: public std::exception
{
public:
const char* what(){return "bad argument to bad_cl "};
...
};
```

如果使用类 `std::exception` 处理派生的异常，可以在同一个基类处理程序中捕获此类信息。

```
try{
//...
}
catch(std::exception& e)
{
cout << e.what() << endl;
}
```

当然也可以分别捕获这些异常（前面已有介绍）。

STL 定义了很多基于类 `exception` 的异常类型。头文件 `<exception>` 提供了类 `bad_excep-`

tion, 以供 `unexpected()` 函数使用。

1. 异常类 `stdexcept`

头文件 `<stdexcept>` 还定义了其他几个异常类。首先, 该文件定义了类 `logic_error` 和类 `runtime_error`, 这两个类均是以公有方式从 `exception` 类派生而来的。类 `logic_error` 的声明形式如下:

```
class logic_error : public exception {
public:
    logic_error(const string& message);
};
```

该类的构造函数接受一个 `string` 型对象作为参数, 该参数为 `what()` 函数提供了返回值 (`message.data`)。

类 `runtime_error` 的声明形式如下:

```
class runtime_error : public exception {
public:
    runtime_error(const string& message);
};
```

同样, 类 `runtime_error` 的构造函数也接受一个 `string` 型对象作为参数。该参数为 `what()` 函数提供了返回值 (`message.data`)。

除上述两个类之外, 还有类 `domain_error`、类 `invalid_argument`、类 `length_error` 和类 `out_of_range`。这些类和类 `logic_error` 的情况非常类似。

对于类 `domain_error`, 数学函数通常有定义域和值域。定义域由参数的可能值组成, 值域由函数可能的返回值组成。例如, 如果程序员在开发程序时, 将不合理的参数传递给 `std::sin()` 函数, 会导致程序抛出 `domain_error` 类异常。

异常类 `invalid_argument` 用于提示“给函数传递一个意外的值”。例如, 如果函数希望接受一个字符串, 其中每个字符要么是 0 要么是 1, 那么当传递的字符串中包含其他字符时, 该函数会引发类 `invalid_argument` 异常。

异常类 `length_error` 用于提示没有足够的空间来执行所需的操作。例如, 类 `string` 的 `append()` 函数在合并获取的字符串长度超过最大允许长度时, 会引发 `length_error` 异常。

异常类 `out_of_range` 通常用于指示索引错误, 即索引的数值超出了正常容量的上下限。例如, 对于某数组, 使用 `operator() []` 进行访问时, 若使用的索引无效, 则会引发 `out_of_range` 异常。

异常类 `runtime_error` 描述了可能在运行期间难以预计和防范的错误。

每个类的名称均能显示出其报告的错误类型。例如,

```
range_error
overflow_error
underflow_error
```

每个类均有自己的构造函数。构造函数的参数用于虚 `what()` 函数返回的字符串。

下溢 (Underflow) 错误会出现在浮点数计算中。通常, 存在浮点型可以表示的最小非零值, 若计算结果比该值小, 会导致下溢错误。整型和浮点型都可能发生上溢错误, 当计算

结果超出某种类型能够表示的最大数量级时,会导致上溢错误。当计算结果可能不在函数允许的范围内,但又没有发生上溢或下溢错误时,将引发 `range_error` 类型的异常。

通常,类 `logic_error` 的异常对象表明存在可以通过编程修复的问题;而类 `runtime_error` 的异常对象表明存在无法避免的问题。这些错误具有相同的常规特征,其区别在于:不同的类名能够让程序员分别处理每种异常。

捕获异常的形式是多种多样的。例如,

```
try{
//...
}
catch(out_of_bounds? & oe)
{
//....
}
catch(logic_error& oe)
{
//....
}
catch(exception& oe)
{
//....
}
```

如果上述形式还不能满足需要,程序员需要从类 `logic_error` 或类 `runtime_error` 派生新类,以确保异常类被纳入同一个继承层次结构中。

2. `bad_alloc` 异常和 `new()`

对于使用 `new()` 时可能出现的内存分配问题, C++ 提供了两种可供选择的处理方式: ①使 `new()` 在无法满足内存请求时返回空指针; ②使 `new()` 引发 `bad_alloc` 类型异常。头文件 `<new>` 中包含了 `bad_alloc` 类型声明,它是从异常类 `exception` 派生而来的。在具体实现过程中,读者可根据实际情况选取一种方式,也可以使用编译器开关或其他方法。下面给出一个最简单的例题,说明异常被捕捉后,程序将显示 `what()` 函数返回的消息,之后终止运行。如果没有捕捉到异常,程序将继续执行,并查看返回值是否为空指针。

3. 示例

针对上述内容,下面给出一个关于处理异常的简单例题。尽管在 C 语言中不建议使用 `goto` 语句,但为了便于测试各种异常类别,本例还是使用了两次 `goto` 语句。

例 8-1

```
#include <iostream>
#include <exception>
using namespace std;
void main()
{
    logic_error a("Exception: logic_error."); //定义错误对象
    runtime_error b("Exception: runtime_error.");
    domain_error c("Exception: domain_error.");
```

```
invalid_argument d("Exception: invalid_argument. ");
length_error e("Exception: length_error. ");
range_error f("Exception: range_error. ");
overflow_error g("Exception: overflow_error. ");
underflow_error h("Exception: underflow_error. ");
out_of_range i("Exception: out_of_range. ");
int switch_K;
cout << "1. Exception: logic_error. " << endl;           //输出提示信息
cout << "2. Exception: runtime_error. " << endl;
cout << "3. Exception: domain_error. " << endl;
cout << "4. Exception: invalid_argument. " << endl;
cout << "5. Exception: length_error. " << endl;
cout << "6. Exception: range_error. " << endl;
cout << "7. Exception: overflow_error. " << endl;
cout << "8. Exception: underflow_error. " << endl;
cout << "9. Exception: Out_of_Range. " << endl;
cout << "0. Exit. " << endl;
loop:  cout << ".... Please Input 1 - 9: .... " << endl;
try{
    cin >> switch_K;                                     //输入选项
    if(switch_K==0)
    {
        goto loop2;
    }
    switch(switch_K)
    {
    case 1:
        throw a;                                       //抛出异常(错误)
        break;
    case 2:
        throw b;
        break;
    case 3:
        throw c;
        break;
    case 4:
        throw d;
        break;
    case 5:
        throw e;
        break;
    case 6:
        throw f;
        break;
    case 7:
        throw g;
```

```
        break;
    case 8:
        throw h;
        break;
    case 9:
        throw i;
        break;
    }
}
catch(logic_error& e) //捕获错误或异常
{
    cout << e.what() << endl;
}
catch(runtime_error& e)
{
    cout << e.what() << endl;
}
catch(domain_error& e)
{
    cout << e.what() << endl;
}
catch(invalid_argument& e)
{
    cout << e.what() << endl;
}
catch(length_error& e)
{
    cout << e.what() << endl;
}
catch(range_error& e)
{
    cout << e.what() << endl;
}
catch(overflow_error& e)
{
    cout << e.what() << endl;
}
catch(underflow_error& e)
{
    cout << e.what() << endl;
}
catch(out_of_range& e)
{
    cout << e.what() << endl;
}
goto loop;
```

```
loop2: return;
}
```

例 8-1 的执行结果为:

```
1. Exception: logic_error.
2. Exception: runtime_error.
3. Exception: domain_error.
4. Exception: invalid_argument..
5. Exception: length_error.
6. Exception: range_error.
7. Exception: overflow_error.
8. Exception: underflow_error.
9. Exception: Out_of_Range.
0. Exit.
.....Please Input 1 -9:.....
1
Exception: logic_error.
.....Please Input 1 -9: .....
2
Exception: runtime_error.
.....Please Input 1 -9: .....
3
Exception: domain_error.
.....Please Input 1 -9: .....
4
Exception: invalid_argument.
.....Please Input 1 -9: .....
5
Exception: length_error.
.....Please Input 1 -9: .....
6
Exception: range_error.
.....Please Input 1 -9: .....
7
Exception: overflow_error.
.....Please Input 1 -9: .....
8
Exception: underflow_error.
.....Please Input 1 -9: .....
9
Exception: out_of_range.
.....Please Input 1 -9: .....
0
Press any key to continue
```



总结 例 8-1 测试了 STL 中所包含的各种异常派生类。通过反复地调用 `throw` 和 `catch`，尝试了各种错误类别的抛出和捕获。请读者认真体会，理解它们的用法。

8.2.2 调用 `abort()`

异常（例如除数为 0 导致的异常）发生时，处理方式之一是调用 `abort()` 函数。`abort()` 函数的原型位于头文件 `cstdlib` 中，典型实现是向标准错误流发送消息 `abormal program termination`，然后终止程序。之后函数的返回值会通知操作系统处理失败。`abort()` 函数是否要刷新文件缓冲区取决于具体的实现过程。在需要的情况下，也可以使用 `exit()` 函数终止程序运行。`exit()` 函数可以完成刷新缓冲区的功能，但不能显示消息。

`abort()` 函数的原型为：

```
void abort( void );
```

`exit()` 函数的原型为：

```
void exit( int status );
```

`abort()` 函数和 `exit()` 函数一旦被执行，程序即终止。在 Visual C++ 6.0 环境中，使用 `abort()` 函数时会弹出“异常退出”对话框；而使用 `exit()` 函数则不会弹出提示信息，程序将直接退出。

例 8-2

```
#include <iostream>
#include <cmath>
using namespace std;
double fun(double x, double y)
{
    double result = 0.0;
    if(fabs(y) <= 1e-4)
    {
        cout << "y may equal to 0.0. " << endl;
        abort();
    }
    else
        result = x/y;
    return result;
}
void main()
{
    double x, y, z;
    cout << "Input number: x and y. " << endl;
    while(cin >> x >> y)
```

```
{  
    if (x == 'q' || y == 'q')  
        break;  
    z = fun(x, y);  
    cout << "fun(x, y) is : " << z << endl;  
    cout << "Please input next set of numbers : " << endl;  
}  
cout << "Bye. " << endl;  
return;  
}
```

程序执行结果:

```
Input number: x and y.  
1  
2  
fun(x, y) is : 0.5  
Please input next set of numbers :  
3  
5  
fun(x, y) is : 0.6  
Please input next set of numbers :  
7  
5  
fun(x, y) is : 1.4  
Please input next set of numbers :  
8  
1  
fun(x, y) is : 8  
Please input next set of numbers :  
9  
0  
y may equal to 0.0.
```

之后弹出如图 8-1 所示的对话框。



图 8-1 例 8-2 执行时弹出的异常对话框

将例 8-2 中的 `abort()` 函数替换为 `exit()`，则程序不会弹出上述的异常对话框。类似例 8-2 中除数为 0 的错误，依靠程序员编写程序来避免是不现实的。

8.2.3 堆栈解退

假如 `try` 块没有直接调用引发异常的函数，而是调用了“调用引发异常的函数”的函数，则程序流程将从引发异常的函数跳转到包含 `try` 块和处理程序的函数。这涉及堆栈解退。

C++ 程序通常是通过函数调用来实现的。C++ 通过将信息放在堆栈中来处理函数调用。程序将调用函数的指令地址放到堆栈中。被调用函数执行完毕之后，程序将使用该地址来确定从何处开始。函数调用会将函数的参数放到堆栈中。在堆栈中，函数参数被视为自动变量（`auto`）。若被调用的函数创建了新的自动变量，则这些变量也会被填入堆栈中。如果被调用的函数调用了另一个函数，后者的信息也将被添加至堆栈中，以此类推。当函数结束时，程序流程将转至该函数被调用时的存储地址处，堆栈顶端的元素将被释放。

函数通常都返回到调用它的函数，同时每个函数都在结束时释放其自动变量。若自动变量是类对象，则类的析构函数将被调用。

若函数出现异常而终止，则程序将释放堆栈中的内存，但不会释放堆栈内的第一个返回地址，而是继续释放堆栈，直至寻找到 `try` 块中返回的地址。之后，控制权将转到块尾的异常处理程序，而不是函数调用后面的第一条语句。这一过程称为堆栈解退。

引发该机制的一个重要特性是：和函数返回一样，对于堆栈中的自动类对象，类的析构函数将被调用。函数返回仅仅处理该函数放在堆栈中的对象，而 `throw` 语句则处理 `try` 块和 `throw` 块之间整个函数调用序列中放在堆栈中的所有对象。如果没有堆栈解退的特性，引发异常后对于中间函数调用放在堆栈中的自动类对象，析构函数不会被调用。

8.2.4 错误代码

通常还有一种办法比异常更加灵活——使用函数的返回值来指出问题。例如，类 `ostream` 的 `get()` 成员函数通常返回下一个输入字符的 ASCII 码，到达文件尾时，将返回特殊值 `EOF`。这种方法通过告知调用程序该调用是否成功，可使得程序采取“终止模型”之外的其他措施。依靠程序可以检查函数的返回值这一方法虽然灵活，但广大程序员并不经常这样做。例如，为了保证程序的短小精悍，有些程序不会检查 `new()` 产生的指针是否为空，也不会检查 `cout` 是否顺利地处理了输出。

在某处存储返回条件的方法是使用全局变量。在出现问题时，函数可以将该全局变量设置为特定的值，调用该函数的程序就可以检查变量的数值。传统的 C 语言数学库使用的错误处理方法即是全局变量，其名称为 `errno`。当然，需要确保其他函数没有将该全局变量用于其他目的。

8.2.5 异常的迷失

异常被引发之后，通常在两种情况下会导致问题：①若异常是在包含异常规范的函数中引发的，则必须与规范列表中的某种异常匹配，否则称为意外异常。在默认情况下，这将导致程序异常终止。②若异常不在函数中引发的，则该异常必须被捕获。若没有被捕获，则异常被称为未捕获异常。默认情况下，这会导致程序异常终止。对于意外异常和未捕获异常的反应，程序员可以根据情况自行设定。

1. 未捕获异常

未捕获异常不会导致程序立刻异常终止。相反，可以通过调用 `terminate()` 函数终止程序。默认情况下，`terminate()` 会调用 `abort()` 函数。因此，程序员可以通过指定 `terminate()` 应调用的函数来实现修改 `terminate()` 的行为。下面介绍 `set_terminate()` 和 `terminate()` 函数的使用方法。这两个函数通常在头文件 `<exception>` 中声明。

```
typedef void (* terminate_handler) ();
terminate_handler set_terminate(terminate_handler f) throw()
void terminate();
```

其中，`typedef` 使 `terminate_handler` 成为一种类型的名称——指向没有参数和返回值的函数指针。`set_terminate()` 函数的参数必须是不带任何参数且返回类型为 `void` 的函数名称或函数地址。执行时，其返回值为参数的地址。若调用 `set_terminate()` 函数很多次，则 `terminate()` 将调用最后一次 `set_terminate()` 设置的函数。

假定希望未捕获的异常会使程序打印一条消息，之后调用 `exit()` 函数，将退出状态值设置为 5。在使用时，必须包含头文件 `<exception>`。

设计一个上述功能的 `myQuit()` 函数。

```
void myQuit ()
{
    cout << "Terminating test because of uncaught exception. " << endl;
    exit(5);
}
```

在 `main()` 函数的起始部分，设置 `set_terminate (myQuit)`。一旦程序中某些异常被触发，并没有被捕获，程序会调用 `terminate()`，而该函数会触发调用 `myQuit()` 函数。

例 8-3

```
#include <exception>
#include <iostream>
using namespace std;
void myQuit ()
{
    cout << "Terminating test because of uncaught exception. " << endl;
    exit(5);
}
void main ()
```

```
{
terminate_handler oldHandler = set_terminate(myQuit);
int switch_K;
logic_error a("Exception: logic_error. ");
runtime_error b("Exception: runtime_error. ");
cout << "..... Please Input 0 for exit.          ....." << endl;
loop:  cout << "..... Please Input 1 or else number: ....." << endl;
try{
    cin >> switch_K;
    if (switch_K == 0)
    {
        goto loop2;
    }
    switch (switch_K)
    {
    case 1:
        throw a;
        break;
    default:
        throw b;
        break;
    }
}
catch (logic_error& e)
{
    cout << e.what() << endl;
}
catch (...) //注意: catch() 函数的参数为“...”
{
    terminate();
}
goto loop;
loop2: return;
}
```

程序执行结果:

```
..... Please Input 0 for exit.          .....
..... Please Input 1 or else number: .....
1
Exception: logic_error.
```

```
.....Please Input 1 or else number: .....
2
Terminating test because of uncaught exception.
```

2. unexpected() 异常处理

程序员应该明确哪些异常是必须要捕获的。因为在默认情况下，未捕获的异常将导致程序异常终止。通常，从原则上讲，异常规范需要包含函数调用的各种可能引发的异常。例如，若 A() 函数调用 B() 函数，而函数 B 会触发某个对象异常，则异常规范中应包含该对象。

如果程序引发了异常规范中没有的异常（即意外异常），此行为与未捕获的异常极其相似。如果发生意外异常，程序会调用 unexpected() 函数。unexpected() 函数会调用 terminate() 函数，而 terminate() 函数默认情况下会调用 abort() 函数。和 set_terminate() 相对应，STL 提供了 set_unexpected() 函数用于修改 unexpected() 函数的行为。unexpected() 函数和 set_unexpected() 一样，也是包含在头文件 <exception> 中的。

```
typedef void (* unexpected_handler)();
unexpected_handler set_unexpected(unexpected_handler _pnew) throw();
void unexpected();
```

和 set_terminate() 函数不同，set_unexpected() 函数的参数是严格受限制的。通常，unexpected_handler 函数可以有以下行为：

- 1) 调用 terminate()、abort() 或 exit()，从而终止程序。
- 2) 触发异常。

就第二种行为而言，引发异常的结果取决于 unexpected_handler 所引发的异常及引发异常函数的异常规范。一般包括以下几种情况：

① 若新引发的异常与原来的异常规范匹配，则程序会开始进行正常处理，即寻找与新引发的异常匹配的 catch 块。基本上，此方法将用于预期的异常取代意外异常。

② 若新引发的异常与原有的异常规范不匹配，且异常规范中不包括 std::bad_exception 类型，则程序还是会调用 terminate() 函数。类 bad_exception 是从 exception 派生而来的，其类的声明在头文件 <exception> 中。

③ 新引发的异常与原有异常规范不匹配，原来的异常规范中包含 std::bad_exception 类型，不匹配的异常会被 std::bad_exception 异常取代。

简而言之，程序设计的目的是要捕获所有异常。通常可以采取如下处理方法：

- 1) 必须包含头文件 <exception>。
- 2) 设计一个替代函数，将意外异常转换为 bad_exception 异常。该函数的原型如下：

```
void myunexpected()
{
    throw std::bad_exception();
}
```

使用 `throw` 时, 若不指定异常, 将导致重新引发的 (意外) 异常。若异常规范中包含此类型, 该异常将被 `bad_exception` 对象所取代。

在 `main()` 函数的开始位置, 将“意外异常”诱发的操作设定为调用自定义的 `myunexpected()`。最后, 将 `bad_exception` 类型包括在异常规范中, 并添加以下 `catch` 块:

```
catch(bad_exception& e)
{
...
}
```

例 8-4

```
#include <exception>
#include <iostream>
using namespace std;
void myUnexpected()
{
    cout << "myUnexpected() is done. " << endl;
    throw bad_exception();
    exit(5); //不再执行
}
void main()
{
    terminate_handler oldHandler = set_unexpected(myUnexpected);
    int switch_K;
    runtime_error a("Exception: runtime_error.");
    bad_exception b("Exception: bad_exception.");
    cout << ".... Please Input 0 for exit. ...." << endl;
    loop:cout << ".... Please Input 1 or 2: ...." << endl;
    try{
        cin >> switch_K;
        if(! switch_K)
        {
            goto loop2;
        }
        switch(switch_K)
        {
            case 1:
                throw a;
                break;
            case 2:
                unexpected();
                break;
        }
    }
    catch(runtime_error& e)
```

```

{
    cout << "catch: " << e.what() << endl;
}
catch (bad_exception& e)
{
    cout << "catch: " << e.what() << endl;
    exit(0);
}
goto loop;
loop2: return;
}

```

例 8-4 的执行结果为:

```

.....Please Input 0 for exit. ....
.....Please Input 1 or 2: .....
1
catch: Exception: runtime_error.
.....Please Input 1 or 2: .....
2
myUnexpected() is done.
catch: bad exception

```

8.2.6 异常处理的局限性

前面讲述了异常处理的重要性和有效性。其实异常处理也有其局限性。前面的讨论显然说明,应该在程序设计时即加入异常处理功能。但这样做会增加程序代码,降低程序的运行速度。异常规范不适用于模板,模板函数引发的异常可能随特定的具体化而异。异常和动态内存分配总是不能协调工作的。例如,

```

void test (int n)
{
    string msg("I was trapped in endless loop. ");
    ...
    if (oh_no)
    {
        throw exception();
    }
    ...
    return;
}

```

在上面的函数中,一旦执行了 `throw` 函数,即终止了 `test()` 函数。`string` 型变量仍然使其析构函数被调用,这得益于堆栈解退,即内存被正确地管理。例如,

```
void test2(int n)
{
    double * ar = new double[n];
    ...
    if(oh_no)
    {
        throw exception();
    }
    ...
    delete[] ar;
    return;
}
```

上述代码中，解退堆栈时，需要删除堆栈中的变量 `ar`。函数提前终止意味着函数末尾的 `delete[]` 语句无法执行。指针消失了，但该指针指向的内存块未被释放，并且不能被访问。简而言之，这块内存泄漏了。

这种简单的内存泄漏是可以避免的。程序员可以在引发异常的函数中捕获该异常，在 `catch` 块中包含一些清理代码，然后重新引发异常。例如，

```
void test2_i(int n)
{
    double * ar = new double[n];
    ...
    try{
        if(oh_no)
        {
            throw exception();
        }
    }
    catch(exception & e)
    {
        delete [] ar;
        throw;
    }
    ...
    delete[] ar;
    return;
}
```

异常处理对于某些项目极为重要，但同时也增加了编程的工作量，增大了程序、降低了速度。由于编译器对异常的支持以及用户的经验还没达到成熟的程度，因此程序员应有节制地使用该特性。



提示 在 STL 中，异常处理的复杂程度越来越高。其主要原因在于文档没有对异常处理例程进行解释或解释蹩脚。任何熟练使用现代操作系统的人都遇到过“未处理异常”导致的错误和问题。这些错误是程序员通常面临的异常艰难的战役。读者应通过不断了解库的复杂性，熟悉各种异常的引发原因及处理措施。

尤其对于程序员新手而言，理解库中的异常处理比学习 C 语言更难。要开发出优质的软件，必须花时间了解库中的复杂内容，就像必须花时间学习 C++ 一样。通过学习库文档和源代码，了解到的异常和错误处理细节会使程序员受益。

8.3 处理异常详述

异常处理的基本思想是简化程序的错误代码，为程序鲁棒性提供一个标准检测机制。前面章节已经讲述：C++ 通过使用 `throw` 关键字产生异常，使用 `try` 关键字定义需要检测的程序块，使用 `catch` 关键字来捕获异常以及填写异常处理的代码；异常通常是由一个确定类或衍生类的对象产生；C++ 能释放堆栈，并可清除堆栈中的所有对象。

本节将详细讲述异常的处理、异常的声明以及异常的访问。

对于可预料但不可避免的程序错误，不能任其发生并无所作为，要将消极的等待变为积极预防，还要将预防的处理内容归纳整理，分门别类地做成类。不能仅凭编程经验，而是要凭借人人均掌握的规范，实现圆满的处理。通常，规范是指在函数处理过程中设置陷阱，异常一旦触发，必然会被异常处理所收纳，最终实现统一归口处理。

前面已经讲过，异常处理有 3 种方式：

- 1) 在出现异常时，直接调用 `abort()` 或者 `exit()` 函数。
- 2) 通过函数的返回值来判断异常；如果函数包含有多个返回值，会浪费不必要的时间开销。
- 3) 通过 `try()-catch()` 函数的结构化异常处理而完成。

1. 异常处理的实现机制

异常处理的实现离不开 3 个关键字：`throw`、`try` 和 `catch`。抛掷异常是在特定的函数中，`try` 块应该包含该函数，`catch` 块根据不同的情况进行相应的处理。其形式大致如下：

```
function()
{
    throw 表达式;
}
try{
    ...
    function();
    ...
}
catch (异常类型声明)
{
    ...
}
```

```
catch (异常类型声明)
{
    ...
}
```

前面章节给出了很多较详细的例子。此处再提供一例。

例 8-5

```
#include <iostream>
#include <exception>
using namespace std;
float mydiv(float x, float y)
{
    if (y == 0.0)
        throw y;
    return x/y;
}
void main()
{
    try{
        cout << "5/2 = " << mydiv(5.0, 2.0) << endl;
        cout << "8/0 = " << mydiv(8.0, 0.0) << endl;
        cout << "7/1 = " << mydiv(7.0, 1.0) << endl;
    }
    catch(float)
    {
        cout << "exception of dividing zero. \n" << endl;
    }
    cout << "test ok!" << endl;
}
```

例 8-5 的执行结果为：

```
5/2 = 2.5
```

```
exception of dividing zero.
```

```
test ok!
```

2. 异常处理语句的语法

throw、try 和 catch 3 个关键字分工明确，各司其职。throw 负责发现异常，并抛出异常对象；如果将其放在函数声明中，又被称为异常接口声明。try 负责设置一个侦错范围（又叫保护段）。try 的作用其实是划定一个跳跃的边界。catch 负责处理捕获的异常，所抛出的异常对象并非建立在函数栈上，而是建在专用的异常栈上，可以跨越函数而传递给上层。读者请注意以下条目：

- 每个 catch() 相当于一段函数代码。
- 每个 throw 则相当于一个函数调用。
- 每个 try 块至少跟一个 catch()。

- 每个程序可设置个数不定的 try、throw 和 catch。只有逻辑上的呼应，而无数量上的对应关系，且不受所在函数模块限制。
- 异常抛出点往往距异常捕获点很远，可以不在同层模块中。
- 甚至有的 throw 不明确，可能来自于系统函数或标准库中。
- try 块可以并列，可以嵌套。

3. 异常处理不唤醒

在 try 保护段代码执行期间，一旦发生异常，则立即抛出，并由相应的 catch 子句捕获处理。“抛出”和“捕获”之间的代码被越过，不会被执行。程序从 try 块之后跟随的最后一个 catch 块子句后面的语句继续执行。

通俗地讲，异常处理是将检测与处理分离，以便各司其职，灵活搭配工作。检测与处理之间的联系依靠“异常类型”。异常通过 throw 创建一个异常对象并抛出。可能抛出异常的程序段嵌在 try 块中，按正常的顺序执行到达 try 语句时，会执行 try 块内的保护段。如果在保护段执行期间没有引起异常，那么在 try 块之后的 catch 子句将不被执行。程序会从 try 块后最后一个 catch 子句后面的语句继续执行。catch 子句按其在 try 块后出现的顺序被匹配之后，catch 块将其捕获并处理异常。

如果没有找到匹配的处理器，terminate() 函数会被调用，其默认功能是调用 abort() 函数并终止程序。

4. 函数声明

关键字 throw 还可以用于函数声明。前面章节已经有过介绍。例如，

```
void fun() throw(int);  
void fun() throw();  
void fun()
```

在函数声明时，若带有 throw 关键字，则在函数定义时必须同样出现。例如，

```
void fun() throw(int)  
{  
...  
}
```

5. 使用异常

“什么时候使用异常？”这是很难回答的问题。

首先，不能使用异常完全代替返回值。因为返回值的含义不一定只是成功或失败，可能会是一个选择的状态。

其次，在特定情况下，异常能否发挥它的优点（该优点恰好是不能使用其他技术实现的）。

异常还有如下一些规则：

- 1) 异常机制只能用于处理错误。
- 2) 通常不要使用 goto 语句或 switch 语句跳转至 try 块或 catch 块内。
- 3) 不要显式地抛出 NULL。
- 4) 若在函数声明时指定了具体的异常类型，则只能抛出该指定类型的异常。
- 5) 若在函数声明时设定了异常类型，则在编译单元中该函数的声明必须有同样的

设定。

- 6) 异常只能在初始化之后, 程序结束之前抛出。
- 7) `throw` 语句本身不允许引发新的异常。
- 8) 空的 `throw` 语句只能出现在 `catch` 语句块中。
- 9) 所有流程中显示的抛出异常应该有一个类型兼容的处理程序。
- 10) 至少需要有一个程序处理所有其他针对处理的异常。
- 11) 若 `try-catch` 语句块中包含多个处理程序, 或派生类和部分或全部基类的 `function-try-block` 块有多个处理程序, 则处理程序的顺序应该是先派生类后基类。
- 12) 如果 `try-catch` 语句块或者 `function-try-block` 块有多个处理程序, `catch (...)` 处理程序 (捕获所有其他异常) 应该放在最后。
- 13) 若异常对象为类的对象, 则应通过引用来捕获。
- 14) 若构造函数和析构函数是 `function-try-block` 结构, 则在 `catch` 处理程序中不能引用该类或其基类的非静态成员。
- 15) 析构函数退出之后, 不允许还有未处理的异常。

8.4 异常的特殊处理函数

前面章节已经简述了一些和异常处理相关的 STL 函数。本节将详细讲述 `terminate()`、`unexpected()` 和 `uncaught_exception()` 函数的使用方法。

异常处理机制主要依赖于两个函数: `terminate()` 和 `unexpected()`。而处理错误主要依赖于异常处理机制本身。

1. `terminate()` 函数

在下述情况下, 程序员必须放弃异常处理, 而采用较弱的错误处理方法:

- 1) 当异常处理机制调用的用户函数包括一个 `uncaught_exception` 时, 在抛出完整的信息之后, 异常被捕获之前的时间段内, 程序员应采用较弱的错误处理方法。
- 2) 当异常处理机制不能发现抛出的异常句柄时, 堆栈中退出对象的析构函数存在一个异常时。
- 3) 当函数被执行时, 并且此函数使用 `atexit()` 注册过。
- 4) 当一个抛出的表达式尝试重新抛出异常, 并还没有异常被处理时。
- 5) 当 `unexpected()` 函数抛出一个异常, 而该异常不被前面的异常说明所允许, 并且 `std::bad_exception` 不包括在异常描述中; 类 `bad_exception` 的声明如下:

```
class bad_exception : public exception
{
...
};
```

- 6) 当默认的 `unexpected_handler` 被调用时。

当满足上述条件之一时, `terminate()` 函数可以被执行。若没有匹配的句柄被发现, 在函数被调用之前, 不确定堆栈中的对象是否退出。在所有其他条件下, 堆栈中的对象在 `terminate()` 函数调用之前, 不应该从堆栈中退出。当确定退栈过程最终会引起调用 `terminate()`

时，制订措施不允许过早地完成堆栈的退出。

2. unexpected() 函数

若一个函数带有一个异常描述，而抛出异常后，该异常描述没有被列表，则 `void unexpected()` 函数会被迅速调用。`unexpected()` 函数不应该返回，但它能抛出一个异常。如果它抛出新的异常，异常描述是必要的。

当抛出或重新抛出一个异常时，异常描述不允许以下情况发生：如果异常描述不包括类 `bad_exception`，`terminate()` 函数会被调用，否则被抛出的异常是可以被自定义异常类对象替代的。

异常描述仅仅保证抛出被明确列出的异常。如果异常描述包括类 `std::bad_exception` 的异常对象，其他没有被列出的异常，会在 `unexpected()` 函数中被 `std::bad_exception` 替代。

类 `std::bad_exception` 的声明形式如下：

```
class bad_exception : public exception
{
...
};
```

3. uncaught_exception() 函数

函数 `uncaught_exception()` 的声明形式如下：

```
bool uncaught_exception();
```

前面章节已经讲过，此函数可以放在析构函数中。

当函数 `uncaught_exception()` `throw()` 返回 `true` 之后，被抛出的对象被解析评估之后，与其相匹配的句柄完成异常声明的初始化以及“退栈”过程。若异常被重新抛出，则在重新抛出异常之处，`uncaught_exception()` 函数返回 `true`，直到重新抛出的异常被再次捕获。

8.5 小结

本章主要讲述了 4 部分内容：第一部分讲述了异常的概念和基本思想；第二部分对于异常类和几个重要问题做了简要阐述；第三部分对如何处理异常进行了详细阐述；第四部分阐述了异常处理的特殊函数。



提示 本章的学习重点应该是关于异常处理的概念和异常处理的方法、异常类的学习以及对重要问题的阐述。通过对这些内容的学习，读者应逐步掌握异常处理的几个关键函数，再经过长时间的实践锻炼，最终必能熟练掌握异常处理的内容和方法。

第 9 章

通用工具类模板 (Utility)

C++ 标准库的通用工具由短小精悍的诸多类和函数构成，通常执行最一般化的功能。头文件 `<utility>` 是非常小的头文件。本章对于通用工具库主要涉及比较、对组、动态内存管理、日期和时间、通用型别、数值极值等各方面的功能模块和函数。这部分内容在 ISO/IEC—14882—2003 C++ 标准的第 20 款 (clause) 中有所描述。还有部分辅助函数在头文件 `<algorithm>` 中声明，按 STL 的要求，这些函数不能称之为算法。

9.1 通用工具库简介

本节主要描述模板参数的要求，主要讲述模板中类型的使用以及内存配置器的性能和要求。

9.1.1 相等比较

许多模板类均定义了 “==” 运算符。

在表 9-1 中，T 是一种类型，该类型是由 C++ 模板提供的，其中的 a, b, c 是类型 T 的数值。

表 9-1 相等比较运算符的说明

表达式	返回类型	性能和要求
T a, b 且 a==b	可转换为布尔类型	“==” 是一种相等关系，即它满足以下性质： 1. 对任意数值 a, a==a 2. If a==b, then b==a 3. If a==b and b==c, then a==c

9.1.2 小于比较

在表 9-2 中，T 是由 C++ 提供的一种模板类型，其中 a 和 b 是类型 T 的值。

表 9-2 小于比较运算符说明

表达式	返回类型	性能要求
a < b	可转换为布尔类型	< 是一种严格的弱序关系

和“小于”操作相关的还有“!=”“>”“<=”“>=”等 4 个比较运算符。小于比较是利用运算符“==”和“<”完成的。这几个函数均定义在头文件 `<utility>` 中。虽然运算是多个，但仅需定义“<”和“==”即可使用这些运算符。在使用时还需要添加 u-

sing namespace std::rel_ops。这些运算符均定义在 std 的次命名空间 rel_ops 中。这样安排主要是为了防止用户定义的全局命名空间中的同类型操作待发生冲突。使用命名空间 std::rel_ops 之后,新的比较运算符就轻松到手了,无需使用复杂的搜寻规则来引用它们。

运算符“!”“>”“<=”和“>=”的声明形式如下:

```
namespace std{
    namespace rel_ops{
        template <class T> bool operator ! = (const T& , const T&);
        template <class T> bool operator > (const T& , const T&);
        template <class T> bool operator < = (const T& , const T&);
        template <class T> bool operator > = (const T& , const T&);
    }
}
```

C++ STL 还可以使用对组 (pair) 进行比较运算,其声明形式如下:

```
template <class Type1, class Type2 > bool operator ! = (
    const pair <Type1, Type2 > & _left , const pair <Type1, Type2 > & _right
);
```

下面通过例 9-1 对“!”运算符进行举例说明。其余运算符仅需将程序中的运算符符号取代即可。

例 9-1

```
#include <utility>
#include <iomanip>
#include <iostream>
int main()
{
    using namespace std;
    pair <int, double> p1, p2, p3;
    p1 = make_pair ( 10, 1.11e-1 );
    p2 = make_pair ( 1000, 1.11e-3 );
    p3 = make_pair ( 10, 1.11e-1 );
    cout.precision ( 3 );
    cout << "The pair p1 is: ( " << p1.first << ", "
        << p1.second << ")." << endl;
    cout << "The pair p2 is: ( " << p2.first << ", "
        << p2.second << ")." << endl;
    cout << "The pair p3 is: ( " << p3.first << ", "
        << p3.second << ")." << endl << endl;
    if ( p1 ! = p2 )
        cout << "The pairs p1 and p2 are not equal." << endl;
    else
        cout << "The pairs p1 and p2 are equal." << endl;
    if ( p1 ! = p3 )
        cout << "The pairs p1 and p3 are not equal." << endl;
    else
```

```

        cout << "The pairs p1 and p3 are equal. " << endl;
    }

```

例 9-1 的运行结果为:

```

The pair p1 is: (10, 0.111 ).
The pair p2 is: (1000, 0.0011 ).
The pair p3 is: (100, 0.0111 ).
The pairs p1 and p2 are not equal.
The pairs p1 and p3 are not equal.

```

部分版本采用两个不同的参数型别来定义上述模板 (template)。例如,

```

namespace std{
    template <class T1, class T2 >
        inline bool operator! = (const T1& x, const T2& y)
        {
            return ! (x == y);
        }
    ...
}

```

此时, 虽然两个操作数的型别不同, 但它们之间仍可以比较, 而这不是C++ 标准库所支持的。

例 9-2

```

#include <iostream>
#include <utility>
using namespace std;
void main()
{
    int x=0, y=0;
    float z=0,w=0;
    x=1;
    if(x > y)
    {
        cout << "x is bigger than y. " << endl;
    }
    else
    {
        cout << "x is not bigger than y. " << endl;
    }
    y=2;
    if((x, y) != (y, z))
    {
        cout << "(x, y) != (y, z). " << endl;
    }
    else
    {

```

```

        cout << "(x,y) == (y,z). " << endl;
    }
}

```

例 9-2 的输出结果为:

```

x is bigger than y.
(x, y)! = (y, z).

```

9.1.3 复制构造

在表 9-3 中, T 是 C++ 程序提供的一种模板类型, 其中 t 是类型 T 的值, u 是常量类型 T 的值。

表 9-3 可复制构造的性能

表 达 式	返 回 类 型	性 能 要 求	表 达 式	返 回 类 型	性 能 要 求
T (t)		t 等效于 T (t)	&t	T*	表明 t 的地址
T (u)		u 等效于 T (u)	&u	const T*	表明 u 的地址
t ~T					

9.1.4 配置器要求

STL 库描述了内存配置器要求的标准集, 其中包含了内存配置模型的所有对象。该信息包括指针类型知识、对象大小类型、区别的类型、内存配置和存储单元分配等。就内存配置器而言, 所有容器均被参数化。表 9-4 描述了通过内存配置实现类型控制的要求。表 9-5 描述了内存配置器类型的性能和要求。

表 9-4 描述性可变量定义

变 量	定 义
T, U	任何类型 (any type)
X	一个配置器类 (和数据类型 T 相关)
Y	相应的配置器类 (和数据类型 U 相关)
t	常量 T& 型的值
a, a1, a2	类型 X& 的值
b	类型 Y 的值
p	类型 X::pointer 的值, 包含在可调用 a1.allocate 中, 此处 a1 == a
q	X::const_pointer 类型的值, 从数据类型 q 转换而来
r	类型 X::reference 从 *p 转换而来
s	类型 X::const_reference 的值, 从表达式 *q 获取或从值 r 转换而来
u	类型值 Y::const_pointer 通过调用 Y::allocate 获得, 或为 0
n	类型 X::size_type 的值

表 9-5 内存配置器类型的性能和要求

表达式	返回类型	断言/提示预/后-条件处理
<code>X::pointer</code>	Pointer to T	
<code>X::const_pointer</code>	Pointer to const T	
<code>X::reference</code>	T&	
<code>X::const_reference</code>	T const&	
<code>X::value_type</code>	Identical to T	
<code>X::size_type</code>	unsigned integral type	一个类型, 在内存配置模型中可以代表最大对象的大小
<code>X::difference_type</code>	signed integral type	在内存配置模型中, 该类型可以代表任意两个指针之间的差
<code>typename X::template rebind<U>::other</code>	Y	对于所有 U (包括 T), <code>Y::template rebind<T>::other</code> 是 X
<code>a.address (r)</code>	<code>X::pointer</code>	
<code>a.address (s)</code>	<code>X::const_pointer</code>	
<code>a.allocate (n)</code>	<code>X::pointer</code>	内存可配置 n 个类型为 T 的对象, 但对象不会被构造
<code>a.allocate (n, u)</code>		
<code>a.deallocate (p, n)</code>	not used	调用之前, 由 p 确定的区域中, 所有 n 个 T 型对象指出, 应该会被破坏。n 应该匹配由 allocate 配置获取的内存大小。并且不抛出异常
<code>a.max_size ()</code>	<code>X::size_type</code>	由 <code>X::allocate ()</code> 可配置的最大内存值
<code>a1 == a2</code>	bool	如果每个对象分配的内存可以使用其他对象被重新分配, 返回 true
<code>a1 != a2</code>	bool	等同于 <code>!(a1 == a2)</code>
<code>X ()</code>		创建默认的实例。提示: 析构器被假定
<code>Xa (b)</code>		post: <code>Y (a) == b</code>
<code>a.construct (p, t)</code>	not used	效果: <code>new ((void *) p) T (t)</code>
<code>a.destroy (p)</code>	not used	效果: <code>((T *) p) -> ~T ()</code>

说明: 1) 表 9-5 中的成员类模板 `rebind` 是有效的类定义模板: 如果名称 `Allocator` 和某配置器 (`SomeAllocator<T>`) 模板绑定, 配置类的成员函数 `rebind<U>::other` 和 `SomeAllocator<U>` 是同一类型;

2) 在国际标准中描述的容器, 被允许假设, 除表 9-5 之外, 配置器模板参数满足下面的两个额外要求。

① 给定配置器类型的所有实例被要求互相变化, 并且总是相互相等的。

② 类型定义 `members_pointer`、`const_pointer`、`size_type` 和 `difference_type` 均是被要求称为 `T*`、`T const*`、`size_t` 和 `ptrdiff_t` 等。

开发人员 (实现者) 鼓励提供库。该库可以接受内存配置器。该内存配置器可以包括更普通的内存模型, 并支持“不等”实例。配置器实例进行“非等于”比较时, 容器的语义和算法是可自定义实现的。

9.1.5 运算符

为提供运算符 (! =) 定义, 库提供以下运算符的声明:

```
template <class T> bool operator! = (const T&x , const T& y );
```

要求: 类型 T 是可以平等比较的。

返回值:!(x == y)

```
template <class T> bool operator < (const T&x , const T& y );
```

要求: 类型 T 是小于可比较的。

返回值: y < x

```
template <class T> bool operator < = (const T&x , const T& y );
```

要求: 类型 T 是小于等于可比较的。

返回值:!(y < x)

```
template <class T> bool operator > = (const T&x , const T& y );
```

要求: 类型 T 是大于等于可比较的。

返回值:!(x < y)

9.1.6 对组 (Pairs)

1. 对组的概念

通俗地讲, 对组 (Pair) 适用于需要将两个数视为一个单元数值的场合。C++ 标准程序库内多处使用该类。尤其是 map 和 multimap 型容器, 它们均是使用对组实现管理其“键值和实值”这两个成对出现的元素。任何函数若总是需要返回两个数值, 也可以使用对组。在头文件 <utility> 中, 对组的声明形式如下:

```
template<class Type1, class Type2> struct pair
{
    typedef Type1 first_type;
    typedef Type2 second_type
    Type1 first;
    Type2 second;
    pair();
    pair(const Type1& __Val1, const Type2& __Val2);
    template<class Other1, class Other2> pair(const pair<Other1, Other2>& __Right);
};
```

为便于使用 pair 类型, 程序员可以构造明确的模板类 pair<T, U> 对象, 可以通过让默认的构造函数提供默认初始值来完成该工作。如果需要获取 pair 对象 (例 x) 的第一个成员, 可以使用 first; 如果需要获取第二个成员, 需要使用 second。使用成员模板构造函数, 可以用另一个模板类 pair<V, W> 的对象构造模板类 pair<T, U> 的对象。使用成员模板构造函数, 可以用另一个模板类 pair<V, W> 的对象来构造模板类 pair<T, U> 的对象。

相应成员提供初始值。

对组被定义成结构体 (struct), 而不是定义成类 (class)。其所有成员都是公用型 (public), 可以直接存取 pair 中的单一值。结构体中默认的构造函数会生成一个 pair(), 以两个“被该 default 构造函数分别初始化”的值作为初始值。根据语言规则, 基本型别的默认构造函数可以适当的初始化。例如,

```
std::pair<int, float> p; //结构体 pair 的第一个和第二个元素被初始化为 0
```

在结构体 pair 的对象 p 初始化时, 会调用类型 int() 和 float() 来初始化 p。该两个构造函数均返回零值。如果 pair 对象被复制, 调用的是由系统隐式生成的 copy 构造函数。之所以使用模版类的 copy 类型构造函数, 是因为构造过程中可能需要隐式型别转换。

pair 对象之间还可以实现比较。为比较两个 pair 对象, C++ 标准程序库提供常用的操作符。若两个 pair 对象内的所有元素均相等, 则两个 pair 对象即视为相等。pair 对象 “==” 运算符的声明形式如下:

```
name std{
    template <class T1, class T2 >
    bool operator == (const pair<T1, T2 >& x, const pair<T1, T2 >& y)
    {
        return x.first == y.first && x.second == y.second;
    }
}
```

两个 pair 对象互相比时, 第一元素具有较高的优先级。如果两个 pairs 的第一元素不相等, 其比较结果就成为比较行为的结果。若第一元素相等, 才继续比较第二元素, 并将比较结果作为整体比较结果。

```
namespace std{
    template <class T1, class T2 >
    bool operator < (const pair<T1, T2 >& x, const pair<T1, T2 >& y)
    {
        return x.first < y.first || (! (y.first < x.first) && x.second < y.second);
    }
}
```

其他比较运算符和运算符 “<” 的声明方法是相同的。

2. 便捷 make_pair() 函数

模板 (template) make_pair() 函数使程序员无需写出型别, 即可生成一个 pair 对象。make_pair() 函数的声明形式如下:

```
namespace std{
    template <class T1, class T2 >
    pair<T1, T2 > make_pair(const T1& x, const T2& y)
    {
        return pair<T1, T2 > (x, y);
    }
}
```

使用 `make_pair()` 函数即可以迅速地创建一个 `pair` 对象，从而不一定非要使用 `pair` 结构创建该结构的对象，例如，

```
std::make_pair(42, '@');
std::pair<int, char>(42, '@');
```

尤其当必须接受 `pair` 型对象作为函数参数时，使用 `make_pair()` 尤为重要。例如，
//函数声明

```
void func(std::pair<int, const char* >);
```

//函数定义

```
void main()
{
...
    func(make_pair(42, "hello"));
    func(make_pair(41, "bye"));
...
}
```

由以上内容可知，`make_pair()` 函数使得“将两个值作为一个 `pair` 参数来传递”，其形式更简单、更容易。即使两个值的型别不能准确地符合要求，也在 `template` 构造函数提供的支持下顺利工作。尤其使用 `map` 和 `multimap` 时，`pair` 对象更是具有绝对的明确的型别。例如，

```
std::pair<int, float>(42, 2.22);
```

上式产生的效果和下述代码是不一致的

```
std::make_pair(42, 2.22)
```

后者所产生的对组的，第二个元素的型别是 `double`。使用重载函数或类模板 `template`，更确切的型别非常重要。为提高效率，程序员可能会同时提供针对 `float` 和 `double` 的函数或类模板 `template`。

诸如上述情况，确切的数据类型型别尤显重要。

3. 对组简例

C++ 标准程序库大量运用对组结构对象。前面章节介绍的 `map` 型容器和 `multimap` 型容器的元素型别均是使用 `pair` 型别，即键值 (`key/value`)。C++ 标准程序库中凡是“必须返回两个值”的函数，均可利用对组对象。下面给出最简单的例题，说明 `pair` 类型的使用方法。

注意：若 `map` 型容器的元素类型是对组，则使用 `map` 型容器排序有多种方法：①在 `map` 声明时，即定义排序规则；②将 `map` 中的元素转换至 `vector` 型容器中，使用算法 `sort` 排序。第一种方法是自动排序，数据输入之后，不再保留原有的输入顺序；第二种方法在数据输入完毕之后，能够保持原有的输入顺序。在此基础上进行排序看似更加合理一些。

例 9-3

```
#include <iostream>
#include <map>
#include <utility>
```

```
#include <fstream>
#include <cstdio>
#include <iomanip>
#include <algorithm>
#include <functional>
#include <vector>
using namespace std;
typedef pair<int, float> mypair;
void out(mypair& p)
{
    cout << "Index: " << right << p.first << ", ";
    cout << std::fixed << setprecision(2) << setw(8) << right << p.second << " ";
}
void print(map<int, float> &m)
{
    map<int, float>::iterator it;
    for(it = m.begin(); it != m.end(); it++)
    {
        mypair p = (mypair)(* it);
        out(p);
    }
}
void printV(vector<mypair> &v)
{
    vector<mypair>::iterator it;
    for(it = v.begin(); it != v.end(); it++)
    {
        mypair p = (mypair)(* it);
        out(p);
    }
}
void In_Original(string filename, map<int, float> &m)
{
    if(filename.empty())
    {
        cout << "filename is error, please input again. " << endl;
        exit(0);
    }
    m.clear();
    ifstream ifn;
    ifn.open(filename.c_str(), ios_base::in);
    if(ifn.fail())
    {
        cout << "file can't be opened. " << endl;
        exit(0);
    }
    else
```

```
{
    while(! ifn.eof())
    {
        //mypair temp_pair;
        char temp[128];
        ifn.getline(temp,127);
        int id;float val;
        sscanf(temp,"%d,%f",&id,&val);
        //temp_pair.first=id;
        //temp_pair.second=val;
        //m.insert(temp_pair);
        m.insert(make_pair(id,val));
    }
}

bool mysort(mypair &m, mypair &n)
{
    return m.second < n.second;
}

bool mysortGreater(mypair &m, mypair &n)
{
    return m.second > n.second;
}

void main()
{
    map<int, float> m,m1;
    m.clear();
    m1.clear();
    map<int,float>::iterator it;
    vector<mypair> v;
    vector<mypair>::iterator itV;
    In_Original("data.txt",m);
    print(m);
    for(it=m.begin();it!=m.end();it++)
    {
        mypair temp=(mypair)(*it);
        v.push_back(make_pair(temp.first,(float)temp.second));
    }
    cout<<"Output vector: "<<endl;
    printV(v);
    sort(v.begin(),v.end(),mysort); //mysort;
    cout<<"Output vector sorted by less. "<<endl;
    printV(v);
    //暂存在 vector 中
    sort(v.begin(),v.end(),mysortGreater);
    cout<<"Output vector sorted by greater. "<<endl;
```

```
printV(v);  
//暂存在 vector 中  
}
```

例 9-3 的执行结果为:

```
Index: 1 , 98.50 ;  
Index: 2 , 99.00 ;  
Index: 3 , 100.00 ;  
Index: 4 , 87.00 ;  
Index: 5 , 95.00 ;  
Index: 6 , 94.00 ;  
Index: 7 , 96.00 ;  
Index: 8 , 84.50 ;  
Output vector:  
Index: 1 , 98.50 ;  
Index: 2 , 99.00 ;  
Index: 3 , 100.00 ;  
Index: 4 , 87.00 ;  
Index: 5 , 95.00 ;  
Index: 6 , 94.00 ;  
Index: 7 , 96.00 ;  
Index: 8 , 84.50 ;  
Output vector sorted by less.  
Index: 8 , 84.50 ;  
Index: 4 , 87.00 ;  
Index: 6 , 94.00 ;  
Index: 5 , 95.00 ;  
Index: 7 , 96.00 ;  
Index: 1 , 98.50 ;  
Index: 2 , 99.00 ;  
Index: 3 , 100.00 ;  
Output vector sorted by greater.  
Index: 3 , 100.00 ;  
Index: 2 , 99.00 ;  
Index: 1 , 98.50 ;  
Index: 7 , 96.00 ;  
Index: 5 , 95.00 ;  
Index: 6 , 94.00 ;  
Index: 4 , 87.00 ;  
Index: 8 , 84.50 ;  
Press any key to continue
```



提示 本节讲述了相等比较、小于比较、复制构造、默认构造、配置器要求、运算符和对组等 7 部分内容。前面的章节已经涉及了对组 (pair) 的内容, 本章的第 9.1.7 节又详细讲解了对组的概念以及常用对组的使用方法。

9.2 动态内存管理

对于广大程序员而言，内存处理是最令人头疼的。在大学计算机语言教学课程中大多不会详细阐述。在 STL 中，头文件 `<memory>` 以非常杰出的方式为容器中的元素分配存储空间，同时也为某些算法执行期间产生的临时对象提供机制。C++ 标准中提供了内存配置器 (allocator)，从此以前常用的指针被内存配置器取代。头文件 `<memory>` 的主要部分是模板类，由它产生容器中所有默认的内存配置器。该模板类的多数版本由更新的版本替代。最近几年，为支持内存配置器而需要的语言特性开始得到广泛应用，但实际上需要使用内存配置器的经验却非常有限。

9.2.1 默认配置器

名称空间 `std` 中声明了一系列和内存管理相关的模板类、模板函数、缓冲区模板类、迭代器和算法。

```
namespace std{
    template <class T> class allocator;           //配置器类
    template <>class allocator<void>;          //配置器类
    template<class T, class U> bool operator == (const allocator<T>&, const allocator<U>
&) throw(); //运算符 ==
    template<class T, class U>bool operator! = (const allocator<T>&, const allocator<U>
&) throw(); //运算符! =
    template<class OutputIterator, class T> class raw_storage_iterator;
                                                //类 raw_storage_iterator
    template<class T> pair<T*, ptrdiff_t> get_temporary_buffer(ptrdiff_t n);
                                                //模板函数
    template<class T> void return_temporary_buffer(T* p);
                                                //模板函数
    template<class InputIterator, class ForwardIterator> ForwardIterator
uninitialized_copy(InputIterator first, InputIterator last, ForawrdIterator result);
                                                //模板函数 uninitialized_copy
    template<class ForwardIterator, class T>
void uninitialized_fill (ForwardIterator first, ForwardIterator last, const T& x);
                                                // 模板函数 uninitialized_fill
    template<class ForwardIterator, class Size, class T>
void uninitialized_fill_n(ForwardIterator first, Size n, const T& x);
                                                //模板函数 uninitialized_fill_n
    template<class X> class auto_ptr;
}
```

上述源代码包含了和内存管理相关的各种模板类和模板函数。其中 `allocator` 是模板类，运算符 `operator == ()` 和 `operator! = ()` 是模板函数。类 `raw_storage_iterator` 是内存管理的迭代器，模板 `get_temporary_buffer()` 函数用于返回临时内存的指针。模板函数 `return_temporary_buffer()`

是用于返回临时缓冲区指针。模板 `uninitialized_copy()` 函数用于从指定的源区域拷贝对象至未初始化的目标范围内; `uninitialized_fill()` 是模板函数, 用于根据数据类型对数据进行数值填充; `uninitialized_fill_n()` 是模板函数, 和模板函数 `uninitialized_fill()` 相比较, 二者的差别仅仅在于填充数据的个数。

头文件 `<memory>` 中, 模板类 `allocator` 的声明形式如下:

```
template <class T> class allocator
{
public:
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;
    typedef T* pointer;
    typedef const T* const_pointer;
    typedef T& reference;
    typedef const T& const_reference;
}
```

模板类 `allocator` 的成员函数主要包括:

- 1) `pointer address (reference x) const`。此函数的返回值为 `x` 的引用。
- 2) `const_pointer address (const_reference x) const`。此函数的返回值为 `x` 的引用。

3) `pointer allocate (size_type n, allocator<void>::const_pointer hint = 0)`。此函数使用运算符 `new (size_t)`。 `hint` 或者等于 0, 或者从之前成员函数 `allocate` 获取并没有传递给 `deallocate`。值 `hint` 可能被用于改进实现过程的性能。此函数的返回值是指向数组初始元素的指针, 数组大小为 `size n * sizeof (T)`, `T` 为对象的类型。存储空间是可以通过调用函数 `new()` 获取, 但函数调用时是不确定的, 使用 `hint` 也是不确定的。如果存储空间不能够获取, 函数会抛出异常 `bad_alloc`。

4) `void deallocate (pointer p, size_type n)`。指针 `p` 应该是一个指针, 从函数 `allocate()` 获得的。 `n` 应该等于第一个参数指针 `p` 之后的元素的截取个数。此函数的功能是释放 `n` 个对象的内存, 释放内存的起始位置由指针 `p` 决定。函数 `delete()` 也可以释放指针指向的内存块, 但在调用此函数时, 是不确定的。

5) `size_type max_size() const throw()`。此函数的返回值是 `allocate()` 函数开辟内存时设置的内存大小 (即 `allocate()` 函数的第一个参数)。

6) `void construct (pointer p, const_reference val)`。此函数的功能是在特定地址构造特定类型的对象, 并使用特定的数值进行初始化。

7) `void destroy (pointer p)`。此函数的返回值是 `((T*) p) -> ~T()`。

下面介绍两个全局的运算符 “==” 和 “!=”。

运算符 “==” 的声明形式如下:

```
template <class T1, class T2> bool operator == (const allocator<T1> &
, const allocator<T2> &) throw();
```

运算符 “!=” 的声明形式如下:

```
template <class T1, class T2 > bool operator! = (const allocator <T1 > &,
const allocator <T2 > &) throw ();
```

9.2.2 raw_storage_iterator

`raw_storage_iterator` 用于使算法可以存储其结果至非初始化的内存。标准的模板参数 `OutputIterator` 是必需的，其指针 `operator*` 必须返回一个对象。其目的是运算符 `operator&` 被定义并返回一个 `T` 型指针，并且也被要求满足一个输出型迭代器的性能。`raw_storage_iterator` 型的类声明形式如下：

```
namespace std{
    template <class OutputIterator, class T >
    class raw_storage_iterator:public iterator<output_iterator_tag, void, void, void, void>{
    public:
        explicit raw_storage_iterator(OutputIterator x);
        raw_storage_iterator<OutputIterator, T>& operator* ();
        raw_storage_iterator<OutputIterator, T>& operator = (const T& element);
        raw_storage_iterator<OutputIterator, T>& operator++ (const T& element);
        raw_storage_iterator<OutputIterator, T>& operator++ (int);
    };
}
raw_storage_iterator (OutputIterator x);
```

使用参数 `x` 初始化迭代器的值。

```
raw_storage_iterator <OutputIterator, T>& operator* ();
```

返回：`* this` 指针。

```
raw_storage_iterator < OutputIterator, T > & operator = (const T& element);
```

运算符赋值符号 “=” 使用参数 `element` 构造一个值，并返回迭代器的引用。

```
raw_storage_iterator <OutputIterator, T>& operator++ ();
```

功能：使迭代器前进，并返回迭代器新值的引用。

```
raw_storage_iterator <OutputIterator, T>& operator++ (int);
```

功能：和上一个运算符函数不同，此函数的功能使迭代器前进并返回迭代器原来的值。

9.2.3 临时缓冲区 (Temporary Buffers)

```
template <class T > pair<T* , ptrdiff_t > get_temporary_buffer(ptrdiff_t n);
```

功能描述：获取存储器指针，该内存必须能够存储 `n` 个相邻的 `T` 型对象。

返回值：包含缓冲区地址的“对”和“容量”，如果存储器不能获取，函数会返回一对零值。

```
template <class T> void return_temporary_buffer(T* p);
```

功能描述：释放指针 `p` 指向的内存。

要求：缓冲区应该是最初由 `get_temporary_buffer()` 函数分配的内存。

9.2.4 特定算法

所有迭代器均要求使它们的运算符 `operator*` 返回一个对象，运算符 `operator&` 在被定义时，也需要返回一个 `T` 型指针。在下列算法中，迭代器可用于作为标准的模板参数。在 `uninitialized_copy()` 算法中，标准的模板参数 `InputIterator` 要求满足输入迭代器的性能。所有下述算法中，标准的模板参数 `ForwardIterator` 用于满足前向迭代器的性能需要，也满足可变速迭代器的性能要求，并且要求以下具有性质：在增加、指定、比较、废除有效的迭代器过程中，不需要抛出异常。在下述算法中，如果抛出异常，也是不会产生作用的。

未初始化复制函数 (`uninitialized_copy()`)

此函数的声明形式为：

```
template <class InputIterator, class ForwardIterator >
    ForwardIterator
    uninitialized_copy(InputIterator first, InputIterator last, ForwardIterator result);
```

功能描述：`uninitialized_copy()` 函数相当于以下代码。

```
for(; first != last; ++result, ++first)
    new(static_cast<void*> (&* result))
        typename iterator_traits<ForwardIterator>::value_type(* first);
```

返回值：`result`。

未初始化填充函数 (`uninitialized_fill()`)

此函数的声明形式为：

```
template <class ForwardIterator, class T>
    void uninitialized_fill(ForwardIterator first, ForwardIterator last, const T& x);
```

功能描述：`uninitialized_fill()` 函数相当于以下代码。

```
for(; first != last; ++first)
    new(static_cast<void*> (&* first))
        typename iterator_traits<ForwardIterator>::value_type(x);
```

未初始化填充 `n` 个对象函数 (`uninitialized_fill_n()`)

此函数的声明形式为：

```
template <class ForwardIterator, class Size, class T>
    void uninitialized_fill_n(ForwardIterator first, Size n, const T& x);
```

功能描述：`uninitialized_fill_n()` 函数相当于以下代码。

```
for(; n-- > 0; ++first)
    new(static_cast<void*> (&* first))
        typename iterator_traits<ForwardIterator>::value_type(x);
```

9.2.5 C 函数库中的内存管理函数

前面讲述了 C++ STL 中的内存管理。本小节主要讲述 C 语言原有的内存管理函数。C 语言原有的内存管理函数在 C++ 中同样适用，这些内存函数在头文件 `<cstdlib>` 中声明。这些内存函数包括 `calloc()`、`malloc()`、`free()` 和 `realloc()`。头文件 `<cstdlib>` 和 C 语言中的头文件 `stdlib.h` 基本是一致的，不同之处在于：`calloc()` 函数、`malloc()` 和 `realloc()` 不是通过调用 `new()` 函数尝试分配内存空间。`free()` 函数不是通过调用 `delete()` 释放内存空间。

头文件 `<cstring>` 中包含了几个内存管理有关的函数：`memchr()`、`memcmp()`、`memcpy()`、`memmove()` 和 `memset()`。该头文件中还声明了数据类型 `size_t` 和宏 `NULL`。除了 `memchr()` 函数之外，其他函数的功能没有发生变化。`memchr()` 函数的声明形式为：

```
const void* memchr(const void* s, int c, size_t n)
void* memchr(void* s, int c, size_t n);
```

如果调用成功，此函数返回字符串 `s` 中第一个字符 `c` 的位置指针；如果调用失败，此函数返回 `NULL`。

9.3 堆的内存分配

C++ 程序在存储器的全局存储区分配和释放动态存储块；全局存储区有时称为自由存储区 (Free Store)，更通用的称呼是堆 (Heap)。运用运算符 `new` 从堆中分配存储器，用运算符 `delete` 把存储器返回给堆。

9.3.1 new 和 delete 运算符

`new` 运算符和数据类型、类、结构或数组的名字一起使用，会为新建项目分配存储器并返回存储器位置。程序可以把返回的地址赋予指针。`delete` 把之前分配的存储器返回给堆，操作数必须是以前分配的存储器地址。存储器返回之后可以被 `new` 运算符重新分配。

例 9-4

```
#include <memory>
#include <iostream>
using namespace std;
struct Date{
    int month;
    int day;
    int year;
};
void main ()
{
    Date* birthday = new Date;
    birthday -> month = 6;
    birthday -> day = 24;
    birthday -> year = 2012;
```

```
    cout << "This day is : " << birthday -> month << "/" << birthday -> day << "/" << birthday -> year;  
    cout << endl;  
    delete birthday;  
}
```

例 9-4 的执行结果为:

```
This day is : 6/24/2012
```

上述程序使用 `new` 运算符分配内存空间, 之后进行初始化, 然后在执行显示相关内容的操作之后, 使用 `delete` 释放该内存空间。

9.3.2 分配固定维数的数组

使用 `new` 和 `delete` 运算符为数组获取和释放内存空间, 详见例 9-5。

例 9-5

```
#include <iostream>  
using namespace std;  
void main()  
{  
    int* birthday = new int[3];  
    birthday[0] = 8;  
    birthday[1] = 19;  
    birthday[2] = 2012;  
    cout << "Today is : " << birthday[0] << "/" << birthday[1] << "/" << birthday[2] << endl;  
    delete[] birthday;  
    return;  
}
```

例 9-5 的执行结果为:

```
Today is : 8/19/2012
```

在例 9-5 中, `delete` 运算符之后还添加了 “[]”。这表示将要删除的存储空间是数组。程序员在删除所有动态分配的数组时都必须使用此种表示方法。尤其当数组中的元素属于自定义类型对象时, 此种表示方法尤为重要。

9.3.3 分配动态内存数组

在例 9-5 中, `new` 运算符接受数据类型和数组维数。程序中还可以提供一个变化的维数, `new` 运算符将根据变量值精确分配存储器。例如, 数组的大小可以由屏幕输入, 以实现由用户自定义。详见例 9-6。

例 9-6

```
#include <iostream>  
#include <cstdlib>  
void main()  
{
```

```
std::cout << "Enter the array size: ";
int size;
std::cin >> size;
int* array = new int[size];
for(int i=0;i < size;i++)
{
    array[i] = rand();
}
for(i=0;i < size;i++)
{
    std::cout << "\n" << array[i];
}
std::cout << std::endl;
delete[] array;
return;
}
```

例 9-6 的执行结果为:

```
Enter the array size: 5
```

```
41
```

```
18467
```

```
6334
```

```
26500
```

```
19169
```

在例 9-6 中, 运行程序时首先需要输入数组的大小。new 运算符使用输入值计算所要分配的存储缓冲区的大小。new 运算符将 size 值与数组类型的大小相乘, 确定从堆中分配多少存储器。程序使用 new 运算符创建数组, 在数组中填入随机数, 显示数组的每个元素, 最后使用 delete 运算符删除数组。

9.3.4 处理堆耗尽

使用 new 运算符时, 要充分考虑对存储器的耗尽问题。在进行程序开发时, 通常假设堆不会耗尽。如果程序对堆的要求非常大, 将可能出现堆耗尽。当系统提供的存储器不能满足需要时, C++ 的 new 运算符会抛出运行异常。一旦程序未能捕获异常, 系统便会异常中止程序。多数情况下, 用户可能并不知道中止的原因。另外, 若“堆耗尽”是由程序的错误导致的, 而程序员又没有事先采取措施来阻止程序异常中止, 则将无法知道程序中止的详细信息。大多数程序都未就此问题采取措施。

在实际应用中, 程序员可将 try 块和 catch 块放在 main() 函数中, 这样至少可以实现捕获异常。

9.4 辅助功能

通用工具库还包含了大量诸如数值极限、最大最小值、两值互换、比较操作等辅助

功能。

9.4.1 数值极限

通常，数值型别的极值是与平台相关的特性。C++ 标准程序库通过 `template numeric_limits` 提供极值，取代了传统 C 语言所采用的预处理器常数。整数常数定义于 `<climits>` 和 `<limit.h>` 中，浮点常数定义于 `<cfloat>` 和 `<float.h>` 中。极值的概念有两个优点：①提供了较好的型别安全性；②程序员可借此自定义模板来核定极值。

C++ 标准中规定了各种型别必须保证的最小精度。如果能够注意并运用该极值，较容易写出与平台无关的程序。各种数据类型的最小长度见表 9-6。

表 9-6 各种类型的内存长度

型 别	最小字节长度	型 别	最小字节长度
char	1 byte	float	4 byte
short int	2 byte	double	8 byte
int	2 byte	long double	8 byte
long int	4 byte		

STL 提供了类模板 `class numeric_limits < >`。使用模板通常是为了对所有型别给出通用的解决方案。此外，在必要时使用模板 `template` 为每个型别提供共同的接口。STL 不仅提供通用的模板 `template`，还提供其特化 (Specialization) 版本。类模板 `numeric_limits` 即是该技术的典型例子。

1) 通用性的模板 (`template`) 为型别提供默认极值。

```
namespace std{
template <class T> class numeric_limits{
public:
static const bool is_specialized=false;
...
};
}
```

通用性 `template` 将成员 `is_specialized` 设置为 `false`。对型别 `T` 而言，无所谓极值的存在。

2) 各具体型别的极值由特化版本 (Specialization) 提供。

```
namespace std{
template <> class numeric_limits<int>{
public:
static const bool is_specialized = true;
static T min() throw(){
return -2147483648;
}
static T max() throw(){
return 2147483647;
}
}
```

```

static const int digits = 31;
...
}
}

```

通常 `is_specialized` 设置为 `true`，所有其他成员都根据特定型别的具体极值加以设定。通用性的类模板 `numeric_limits` 及其特化版本均放在头文件 `<limits>` 中。C++ STL 的特化版本涵盖了所有数据基本型别：

```

bool
char
signed char
unsigned char
wchar_t
short
unsigned short
int
unsigned int
long
unsigned long
float
double
long double

```

模板类 `numeric_limits` 的所有成员及其意义见表 9-7。其中第三列对应相应的 C 常数，这些常数包含在头文件 `<climits>`、`<limits.h>`、`<cfloat>` 和 `<float.h>` 中。

表 9-7 模板类 `numeric_limits` 的成员说明

成 员	意 义	对应的 C 常数
<code>is_specialized</code>	用于判断类型型别在类模板 <code>numeric_limits</code> 中是否被明确定义	
<code>is_signed</code>	用于判断该类型型别是否有符号表达式	
<code>is_integer</code>	判断该类型型别是否为整型	
<code>is_exact</code>	判断该计算过程中是否该型别是否会产生舍入误差	
<code>is_bounded</code>	用于判断集合中元素个数是否有界的	
<code>is_modulo</code>	测试是否该类型具有模表述	
<code>is_iec559</code>	测试数据类型是否遵从 IEC559	
<code>min()</code>	最小值。对于浮点数而言，是标准化的值，只有当 (<code>is_bounded</code> 且 <code>! is_signed</code>) 为真时才有意义	<code>INT_MIN</code> , <code>FLT_MIN</code> , <code>CHAR_MIN</code> , ...
<code>max()</code>	最大值。当 <code>is_bounded</code> 成立时才有意义	<code>INT_MAX</code> , <code>FLT_MAX</code> , ...
<code>digits</code>	字符和整数：不带正负号之位个数； 浮点数：尾数中之 <code>radix</code> 位个数	<code>CHAR_BIT</code> , <code>FLT_MANT_DIG</code> , ...
<code>digits10</code>	十进制数个数（只有当 <code>is_bound</code> 成立时才有意义）	<code>FLT_DIG</code>

(续)

成 员	意 义	对应的 C 常数
radix	获取数据类型的底数。(整数:表示式的底数(base)(几乎总是2);浮点数:指数表示式的底数(base))	
min_exponent	底数 radix 的最小负整数指数	FLT_MIN_EXP, ...
max_exponent	底数 radix 的最大正整数指数	FLT_MAX_EXP, ...
min_exponent10	底数 10 的最小负整数指数	FLT_MIN_10_EXP, ...
max_exponent10	底数 10 的最大正整数指数	FLT_MAX_10_EXP, ...
epsilon()	1 与最接近 1 (大于 1 的) 数值之间的差距, 即精确度	FLT_EPSILON
round_style	浮点数舍入成整数的风格	
round_error	该类型的最大舍入误差 (根据 ISO/IEC 10967-1 标准)	
has_infinity	测试有否具有“正无穷大”表述	
infinity()	“正无穷大”表述	
has_quiet_NaN	测试某类型是否具有非数值, 非信号表述	
quiet_NaN()	返回某类型的非数值表述	
has_signaling_NaN	测试该型别具有发出信号的“非数值”表述式	
signaling_NaN	如果可以, 返回某信号的表述, 而不是“数值”	
has_denorm()	测试某型别是否允许非标准化数值	
has_denorm_loss	测试是否能检测出精度的缺失, 作为一个非标准化缺失而不是一个不精确的结果	
denorm_min()	返回最小的非零非标准化正值	
traps	用于测试是否捕获算法异常的报告	
tinyness_before	在舍入处理之前, 作为标准化的数值, 测试某类型是否可以决断: 数值由于太小而不能表述	

注: 表格中“意义”一栏中的诸多解释可能会有不合理之处。著者能力有限, 敬请广大读者批评指正。

下面介绍一个简单的实例 (摘自《C++ 标准程序库》)。

例 9-7

```
#include <iostream>
#include <limits>
#include <string>
using namespace std;
void main()
{
    cout << boolalpha;
    cout << "max(short): " << numeric_limits<short>::max() << endl;
    cout << "max(int): " << numeric_limits<int>::max() << endl;
    cout << "max(long): " << numeric_limits<long>::max() << endl;
    cout << endl;
}
```

```
cout << "max(float): " << numeric_limits<float>::max() << endl;
cout << "max(double): " << numeric_limits<double>::max() << endl;
cout << "max(long double): " << numeric_limits<long double>::max() << endl;
cout << endl;

cout << "is_signed(char): " << numeric_limits<char>::is_signed << endl;
cout << "is_specialized(string): " << numeric_limits<string>::is_specialized << endl;
cout << endl;
}
```

例 9-7 的执行结果为:

```
max(short): 32767
max(int): 2147483647
max(long): 2147483647

max(float): 3.40282e+038
max(double): 1.79769e+308
max(long double): 1.79769e+308

is_signed(char): 1
is_specialized(string): 0
```

程序执行结果的最后一行表示型别 `string` 没有定义数值极限, 因为 `string` 并非数值型别。对任何型别可以进行询问: 是否定义了极值。

9.4.2 较大/较小值 (最大/最小值)

STL 算法库中包含了 3 个辅助函数。一个用于挑选两者之中较大者, 另一个用于在两者中的较小者, 还有一个用于交换两个数值。本小节讲述最大值和最小值算法。后面章节讲述两值交换算法。算法 `min()` 和 `max()` 声明于头文件 `<algorithm>` 中。其声明形式如下:

```
template <class Type >
    const Type& min( const Type& _Left, const Type& _Right );
template <class Type, class Pr >
    const Type& min( const Type& _Left, const Type& _Right, BinaryPredicate_Comp );
template <class Type >
    const Type& max( const Type& _Left, const Type& _Right );
template <class Type, class Pr >
    const Type& max( const Type& _Left, const Type& _Right, BinaryPredicate_Comp );
```

当两值相等时, 通常会返回第一值。上述函数的声明中, 每个函数均有两个版本, 即包含“比较准则”。作为“比较准则”的那个参数应该是个函数或仿函数, 接受两个参数并进行比较。在某个指定规则下, 判断第一参数是否小于第二次参数, 并返回判断结果。

9.4.3 两值交换

算法 `swap()` 用于交换两个对象的值。其泛型化版本定义于头文件 `<algorithm>` 中。

```
namespace std{
    template <class T> inline void swap (T& a , T& b)
    {
        T tmp(a);
        a = b;
        b = tmp;
    }
}
```

运用该函数可以交换任意两个变量。

```
std::swap(x, y);
```

只有当 `swap()` 函数所依赖的 `copy()` 构造操作和 `assignment` 操作行为存在时，调用该算法才有效。`swap()` 的最大优势在于：透过模板特化或函数重载，为更复杂的型别提供特殊的实作版本。这样既可以交换对象内部成员，又不必反复赋值，大大节约了时间。标准程序库中所有容器及 `strings` 均使用了这项技术。

前面讲述算法的章节已经讲述了 `swap()` 的使用，调用算法 `swap()` 不必反复赋值即可交换两容器的值，从而提高了效率。对于自定义型别，`swap()` 算法也确实能够提高效率。此时需要义不容辞地提供 `swap` 的特化版本，即自定义的交换算法，详见例 9-8。

例 9-8

```
#include <iostream>
#include <algorithm>
#include <memory>
#include <exception>
//using namespace std;
class MyContrainer{ //自定义容器类
public:
    int* elem; //容器的存储空间(数组)
    int num_elem; //容器中存储元素的个数
public:
    MyContrainer() //构造函数
    {
        try{
            elem = new int[128]; //开辟部分空间
        }
        catch(exception& e) //捕获异常
        {
            std::cout << "exception. " << e.what() << std::endl;
            exit(0);
        }
    }
}
```

```

void myswap(MyContrainer& mc) //自定义交换算法
{
    std::swap(elem,mc.elem);
    std::swap(num_elem,mc.num_elem);
}
void myassign(int* mc_elem,int num) //自定义 assign 算法
{
    memcpy(elem,mc_elem,num* sizeof(int));
    num_elem = num;
}
~MyContrainer() //析构函数
{
    delete[] elem;
}
};
inline void myswap(MyContrainer& c1, MyContrainer& c2) //全局函数声明
{
    c1.myswap(c2);
}
void print (MyContrainer& c1) //输出容器对象
{
    int num = c1.num_elem;
    for(int i = 0; i < num; i++)
    {
        std::cout << " " << * (c1.elem + i);
    }
    std::cout << std::endl;
}
void main()
{
    int array_one[10] = {1,2,3,4,5,6,7,8,9,0}; //数组 1
    int array_two[10] = {11,12,13,14,15,16,17,18,19,20}; //数组 2
    MyContrainer mc1,mc2;
    mc1.myassign(array_one,10); //初始化容器对象 mc1
    std::cout << "the original mc1: " << std::endl;
    print(mc1); //输出容器的对象
    mc2.myassign(array_two,10); //初始化容器对象 mc2
    std::cout << "the original mc2: " << std::endl;
    print(mc2); //输出容器的对象
    mc1.myswap(mc2); //交换算法
    std::cout << "the original mc1 (swaped): " << std::endl;
    print(mc1); //输出容器的对象
    std::cout << "the original mc2 (swaped): " << std::endl;
    print(mc2); //输出容器的对象
    myswap(mc1,mc2); //交换算法
}

```

```

std::cout << "the original mc1 (swaped twice. ): " << std::endl;
print(mc1); //输出容器的对象
std::cout << "the original mc2 (swaped twice. ): " << std::endl;
print(mc2); //输出容器的对象
}

```

例 9-8 的执行结果为:

```

the original mc1:
1 2 3 4 5 6 7 8 9 0
the original mc2:
11 12 13 14 15 16 17 18 19 20
the original mc1 (swaped):
11 12 13 14 15 16 17 18 19 20
the original mc2 (swaped):

1 2 3 4 5 6 7 8 9 0
the original mc1 (swaped twice. ):
1 2 3 4 5 6 7 8 9 0
the original mc2 (swaped twice. ):
11 12 13 14 15 16 17 18 19 20

```



提示 在阅读本例题时，读者应注意指针的用法和自定义 `myassign()` 函数。本例题主要是通过自定义容器和自定义交换算法来实现两个对象的交换的。

9.4.4 辅助性比较

前面讲述了“小于比较”运算符。本小节重新讲述操作符“`!=`”“`>`”“`<=`”和“`>=`”。这4个操作符均是利用操作符“`==`”和“`<`”完成的且均声明于头文件 `<utility>` 中。

```

namespace std{
    namespace rel_ops{
        template <class T> inline bool operator != (const T& x, const T& y)
        {
            return !(x==y);
        }
        template <class T> inline bool operator > (const T& x, const T& y)
        {
            return y<x;
        }
        template <class T> inline bool operator <= (const T& x, const T& y)
        {
            return !(y<x);
        }
        template <class T> inline bool operator >= (const T& x, const T& y)
        {

```

```
        return ! (x<y) ;
    }
}
}
```

其实，仅需定义“<”和“==”操作符，即可使用上述操作符。使用时需要添加语句 `using namespace std::rel_ops`。上述 4 个比较操作符即自动获得了定义。命名空间 `rel_ops` 属于 `std` 的次命名空间。为防止和用户定义的全局命名空间中的同类型操作符发生冲突，使用名称空间 `rel_ops` 时，必须使用语句 `using namespace std::rel_ops`。某些实例化版本采用两个不同的参数型别来定义上述模板 `template`，即

```
namespace std{
    template <class T1, class T2 > inline bool operator! = (const T1& x, const T2&y)
    {
        return ! (x==y);
    }
    ...
}
```

9.4.5 头文件 `cstdlib` 和 `cstddef` 简介

头文件 `<cstddef>` 和 `<cstdlib>` 和其原有的 C 版本对应，但在 C++ 程序中也经常用到。其在 C 语言中的对应头文件分别为 `<stddef.h>` 和 `<stdlib.h>`。

1. `<stddef>` 中的定义

头文件 `<stddef>` 的定义项主要包括 `NULL`、`size_t`、`ptrdiff_t` 和 `offsetof`。`NULL` 通常用于表明一个不指向任何对象的指针，其实就是 0。C 语言中的 `NULL` 通常定义为 `(void*) 0`。在 C++ 中，`NULL` 的型别必须是整数型别，否则无法将 `NULL` 赋值给指针。而且在 C++ 中没有定义从 `void*` 到任何其他型别的自动转换。`NULL` 同时定义于头文件 `<cstdio>`、`<cstdlib>`、`<cstring>`、`<ctime>`、`<wchar>` 和 `<locale>` 内。

- `size_t` 是一种无正负号的型别，用于表示大小（例如元素个数）。
- `ptrdiff_t` 是一种带有正负号的型别，用于表示指针之间的距离。
- `offsetof` 表示一个成员在结构体（Struct）或联合体（Union）中的偏移量。

2. `<cstdlib>` 中的定义

头文件 `<cstdlib>` 中包含了最重要的一些定义，主要包括 `exit()`、`EXIT_SUCCESS`、`EXIT_FAILURE`、`abort()` 和 `atexit (void (* function) ())`。常数 `EXIT_SUCCESS` 和 `EXIT_FAILURE` 均用于 `exit()` 函数的参数，也可以用于 `main()` 的返回值。

经由 `atexit()` 设定的函数，在程序正常退出时会依据设定的相反次序被一一调用起来。无论是通过 `exit()` 还是 `main()` 退出，均不传递任何参数。

`exit()` 函数和 `abort()` 可用于在任意点终止程序的运行，无需返回 `main()`。`exit()` 函数会销毁所有的 `static` 对象，并将所有缓冲区清空（flush）。关闭所有 I/O 通道，之后终止程序。如果 `atexit()` 函数设定的函数抛出异常，会调用 `terminate()`。`abort()` 函数会立刻终止函数，但不做任何清理工作。

这两个函数不会销毁局部对象，因为堆栈 `unwinding` 不被执行。为确保所有局部对象的析构函数均获得调用，应该运用异常或正常返回机制，之后再由 `main()` 离开。

9.5 日期和时间

在学习本节之前，首先提醒读者时间是很复杂的概念。本节涉及的时间概念主要包括 GMT (格林威治时间)、UTC (世界标准时) 以及日历。读些应能清楚区分这些概念，才能学好本节内容。

STL 库中还包含一些和日期、时间相关联的函数及算法。这些函数包含在头文件 `<ctime>` 中。头文件 `<ctime>` 中包含一个宏 (`NULL`)、3 个类型 (`size_t`、`clock_t` 和 `time_t`)、一个结构体 (`tm`) 以及 10 个 (`asctime()` 函数、`clock()`、`difftime()`、`localtime()`、`strftime()`、`ctime()`、`gmtime()`、`mktime()` 和 `time()`)。本小节主要讲述这些内容的使用方法。

9.5.1 3 个类型

3 个类型是指 `size_t`、`clock_t` 和 `time_t`。

`size_t` 应该是 `unsigned int`。前面已经讲过，`size_t` 是与机器相关的 `unsigned` 类型，其大小足以保证存储内存中对象的大小。引入 `size_t` 主要是满足跨平台的需求，以增强程序在平台上的可移植性。类型 `size_t` 在头文件 `<time.h>` 中的声明形式为：

```
#ifndef _SIZE_T_DEFINED
typedef unsigned int size_t;
#define _SIZE_T_DEFINED
#endif
```

`clock_t` 通常代表长整型。在头文件 `<time.h>` 中，其声明形式为：

```
#ifndef _CLOCK_T_DEFINED
typedef long clock_t;
#define _CLOCK_T_DEFINED
#endif
```

`time_t` 通常也是长整型。在头文件 `<time.h>` 中，其声明形式为：

```
#ifndef _TIME_T_DEFINED
typedef long time_t;          /* time value */
#define _TIME_T_DEFINED     /* avoid multiple def's of time_t */
#endif
```

9.5.2 结构体 (tm)

结构体 (`tm`) 在头文件 `<time.h>` 中的声明形式为：

```
#ifndef _TM_DEFINED
struct tm {
    int tm_sec;      /* seconds after the minute - [0,59] */
    int tm_min;     /* minutes after the hour - [0,59] */
    int tm_hour;    /* hours since midnight - [0,23] */
    int tm_mday;    /* day of the month - [1,31] */
    int tm_mon;     /* months since January - [0,11] */
    int tm_year;    /* years since 1900
```

```
int tm_hour; /* hours since midnight - [0,23] */
int tm_mday; /* day of the month - [1,31] */
int tm_mon; /* months since January - [0,11] */
int tm_year; /* years since 1900 */
int tm_wday; /* days since Sunday - [0,6] */
int tm_yday; /* days since January 1 - [0,365] */
int tm_isdst; /* daylight savings time flag */
};

#define _TM_DEFINED
#endif
```

结构体 (tm) 用于存储时刻或者计时器。其中成员 tm_sec 用于存储时刻中的“秒”，成员 tm_min 用于存储时刻中的“分钟”，成员 tm_hour 用于存储时刻中的“小时”。成员 tm_mday 用于存储时间中的“日期”，成员 tm_mon 用于存储时间中的“月份”，成员 tm_year 用于存储时间中的“年份”。成员 tm_wday 用于存储时间中的“星期几”，成员 tm_yday 用于存储时间中的日期 (1~365)，成员 tm_isdst 用于存储时间标志 (白天或黑夜)。

9.5.3 相关时间函数

本小节将详细讲述和时间有关的 9 个函数及其使用方法。

```
char * __cdecl asctime(const struct tm *);
char * __cdecl ctime(const time_t *);
clock_t __cdecl clock(void);
double __cdecl difftime(time_t, time_t);
struct tm * __cdecl gmtime(const time_t *);
struct tm * __cdecl localtime(const time_t *);
time_t __cdecl mktime(struct tm *);
time_t __cdecl time(time_t *);
size_t __cdecl strftime(char *, size_t, const char *, const struct tm *);
```

- 1) asctime() 函数。此函数用于将指定的时间以字符串 (英文简写) 形式输出。
- 2) ctime() 函数。此函数用于将指定时间以字符串形式输出，并遵从本地时区设置。
- 3) clock() 函数。此函数的返回值是硬件嘀嗒数，需要换算成 s 或者 ms，通常需要除以 CLK_TCK 或者 CLOCKS_PER_SEC。例如，在 Visual C++ 6.0 环境下，这两个宏的值均为 1000，表示硬件嘀嗒 1000 次所耗时间是 1s。因此，当需要计算进程所耗时间时，可以连续调用两次 clock()，所获得两个时间之差即为所耗时间。如果需要折算成秒，需要用硬件嘀嗒数除以 1000 即可，如果不满 1s，可以计算出更精确地时间 (ms)：十分之几秒、百分之几秒、千分之几秒等。
- 4) difftime() 函数。此函数返回两个 time_t 型参数 (时刻) 之间的时间差。
- 5) gmtime() 函数。gmtime (const time_t *) 函数用于将参数传递的时刻转换为格林威治时间。返回值保存在 tm 结构体中。结构体 tm 中的成员定义在上文中已经阐述。格林威治标准时间指的是位于英国伦敦郊区的皇家格林威治天文台的标准时间，因为“本初子午线”被定义为通过该处的经线。自 1924 年 2 月 5 日开始，格林威治天文台每隔 1h 会向全世界发放调时信息。1999 年 12 月 28 日，一种新的时间系统——格林威治电子时间正式诞生，从而为全球电子商务提

供了一个时间标准。原有的格林威治时间 (GMT) 仍予以保留, 作为 21 世纪的世界标准时间。

6) `localtime()` 函数。此函数的功能是把从 1970 年 1 月 1 日零时零分到当前时间系统所偏移的秒数时间转换为日历时间 (本地时区时间)。其返回值是 `time_t` 结构体。此结构体中包含了本地时区的时间。

7) `mktime()` 函数。此函数的功能是将当前的日历时间转换为从 1970 年 1 月 1 日零时零分起至今的 UTC 时间经过的秒数。UTC 时间的中文名称为“协调世界时”, 又称“世界统一时间”“世界标准时间”或“国际协调时间”, 简称 UTC。其英文全称为“Coordinated Universal Time”。

8) `time()` 函数。此函数的功能是获取当前系统 (计算机) 时间, 其返回的结果保存在 `time_t` 结构体中。此结果其实是一个长整数, 其数值表示从 1970 年 1 月 1 日零时零分至当前时间的秒数。可使用函数 `localtime()` 将该时间转换为本地 (本时区) 时间, 并转成 `tm` 结构体, 该类型的数据成员分别表示年、月、日、时、分和秒。

9) `strftime()` 函数。此函数的功能是根据区域设置格式化本地时间/日期, 即将时间格式化, 或者说格式化一个时间字符串。`strftime()` 函数的操作类似于 `sprintf()`, 即识别以 % 开始的格式命令集合, 格式化输出结果并保存在一个字符串中。格式化命令说明串 `strDest` 中各种日期和时间信息的确切表示方法。格式串中的其他字符按原样放进串中。格式命令包含很多种, 并且是区分大小写的, 详见表 9-8。

表 9-8 格式命令及其说明

格式命令	说 明	格式命令	说 明
% a	星期几的简写	% n	新行符
% A	星期几的全称	% p	本地的 AM 或 PM 的等价显示
% b	月份的简写	% r	12 小时的时间
% B	月份的全称	% R	显示小时和分钟 (hh: mm)
% c	标准的日期的时间串	% S	十进制的秒数
% C	年份的后两位数字	% t	水平制表符
% d	十进制表示的每月的第几天	% T	显示时分秒 (hh: mm: ss)
% D	月/天/年	% u	每周的第几天, 星期一为第一天 (值 0 ~ 6, 星期一为 0)
% e	在两字符域中, 十进制表示的每月的第几天	% U	第年的第几周, 把星期日作为第一天 (值从 0 到 53)
% F	年-月-日	% V	每年的第几周, 使用基于周的年
% g	年份的后两位数字, 使用基于周的年	% w	十进制表示的星期几 (值 0 ~ 6, 星期天为 0)
% G	年份, 使用基于周的年	% W	每年的第几周, 把星期一做为第一天 (值从 0 ~ 53)
% h	简写的月份名	% x	标准的日期串
% H	24 小时制的小时	% X	标准的时间串
% I	12 小时制的小时	% y	不带世纪的十进制年份 (值从 0 ~ 99)
% j	十进制表示的每年的第几天	% Y	带世纪部分的十进制年份
% m	十进制表示的月份	% z, % Z	时区名称, 若不能得到时区名称, 则返回空字符
% M	十时制表示的分钟数	% %	百分号

函数参数说明:

```
size_t __cdecl strftime(char * , size_t, const char * , const struct tm * );
```

此函数的第一个参数是目标缓冲区, 第二个参数是缓冲区大小, 第三个参数输出格式, 第四个参数是时间参数。

9.5.4 时间示例

下面以例 9-9 来对上述的函数及其使用方法进行说明。

例 9-9

```
#include <ctime>
#include <iostream>
using namespace std;
void main()
{
    char tempbuf[128];
    struct tm* mytime;
    time_t myclock;
    clock_t start, finish;
    time_t tstart,tfinish,tcurr;
    double duration;
    long count = 600000000L;

    time(&myclock);
    mytime = localtime(&myclock);
    cout << "Usage of asctime() : " << endl;
    cout << "asctime(): Current Date and time : " << asctime(mytime) << endl;

    cout << "Usage of ctime() : " << endl;
    cout << "ctime(): Current Date and time : " << ctime(&myclock) << endl;
    cout << endl;

    cout << "Usage of clock() : " << endl;
    start = clock();
    while(count --);
    finish = clock();
    duration = (double) (finish - start) /CLOCKS_PER_SEC;
    cout << "clock(): Consume " << duration << " second. " << endl;
    cout << endl;

    cout << "Usage of difftime() : " << endl;
    count = 600000000L;
    time(&tstart);
    while(count --);
    time(&tfinish);
```

```

    duration = (tfinish - tstart);
    cout << "time() :    Consume " << duration << " second. " << endl;
    cout << endl;

    cout << "Usage of gmtime() (GMT) : " << endl;
    time(&tcurr);
    mytime = gmtime(&tcurr);
    cout << "The current time : " << (mytime ->tm_year+1900) << " , " << (mytime ->tm_mon+1) << " , " << my-
time ->tm_mday << " , ";
        cout << mytime ->tm_hour << " : " << mytime ->tm_min << " : " << mytime ->tm_sec << " , ";
    cout << (mytime ->tm_isdst?"Night. ":"Daylight. ") << " , " << mytime ->tm_wday << " , " << mytime ->tm_
yday << " th. " << endl;
        cout << endl;
    cout << "Usage of localtime() : " << endl;
    time(&tcurr);
    mytime = localtime(&tcurr);
    cout << "The current time : " << (mytime ->tm_year+1900) << " , " << (mytime ->tm_mon+1) << " , " << my-
time ->tm_mday << " , ";
        cout << mytime ->tm_hour << " : " << mytime ->tm_min << " : " << mytime ->tm_sec << " , ";
    cout << (mytime ->tm_isdst?"Night. ":"Daylight. ") << " , " << mytime ->tm_wday << " , " << mytime ->tm_
yday << " th. " << endl;
        cout << endl;
    cout << "Usage of mktime() : " << endl;
    tcurr = mktime(mytime);
    cout << "The seconds from 1970/1/1/0:0:0. : " << tcurr << " seconds. " << " " << endl;

    cout << "Usage of time() : " << endl;
    time(&tcurr);
    cout << "time transformed by localtime() : " << asctime(localtime(&tcurr)) << endl;
    cout << "time transformed by gmtime() : " << asctime(gmtime(&tcurr)) << endl;

    cout << "Usage of strftime() : " << endl;
    mytime = localtime(&tcurr);
    strftime(tempbuf,128,"Today is %A, day %d of %B in the year %Y. \n",mytime);
    cout << tempbuf << endl;
}

```

例 9-9 的执行结果为:

```

Usage of asctime() :
asctime() :    Current Date and time : Tue Aug 21 14:14:05 2012

Usage of ctime() :
ctime() :    Current Date and time : Tue Aug 21 14:14:05 2012

Usage of clock() :

```

```
clock():    Consume 1.796 second.
Usage of difftime():
time():    Consume 1 second.
Usage of gmtime() (GMT):
The current time: 2012,8,21,6:14:8, Daylight, 2, 233 th.
Usage of localtime():
The current time: 2012,8,21,14:14:8, Daylight, 2, 233 th.
Usage of mktime():
The seconds from 1970/1/1/0:0:0: 1345529648 seconds.
Usage of time():
time transformed by localtime(): Tue Aug 21 14:14:08 2012

time transformed by gmtime(): Tue Aug 21 06:14:08 2012

Usage of strftime():
Today is Tuesday, day 21 of August in the year 2012.
```



提示 无论哪种计算机语言, 无论什么计算机系统, 时间概念基本是相似的。要掌握本节的
一系列时间函数, 首先要弄清楚各种时间标准。读者应对各函数的输入参数和输出参
数需要有清楚的认识, 才能熟练地掌握上述时间函数, 才能准确使用计算机系统提供的时间。

9.6 模板类 auto_ptr

使用模板类 auto_ptr, 需要包含头文件 <memory>。模板类 auto_ptr 使用 new() 函数将指针存储在一个对象中, 当需要破坏此指针时, 使用 delete() 函数。模板 auto_ptr_ref 保持了对类 auto_ptr 的一个引用。该引用可用于允许 auto_ptr 类型对象的传递和从函数中将值传递出来。请阅读以下代码:

```
namespace std{
template < class Y > struct auto_ptr_ref{};
template < class X > class auto_ptr{
public:
    typedef X element_type;
    explicit auto_ptr(X* p=0) throw();
    auto_ptr(auto_ptr&) throw();
    template < class Y > auto_ptr(auto_ptr<Y>&) throw();
    auto_ptr& operator = (auto_ptr&) throw();
    template < class Y > auto_ptr& operator = (auto_ptr<Y>&) throw();
    auto_ptr& operator = (auto_ptr_ref<X> r) throw();
    ~auto_ptr() throw();
    X& operator* () const throw();
```

```

X* operator - > () const throw();
X* get() const throw();
X* release() throw();
void reset(X* p=0) throw();

auto_ptr(auto_ptr_ref<X>) throw();
template<class Y> operator auto_ptr_ref<Y> () throw();
template<class Y> operator auto_ptr<Y> () throw();
};
}

```

类 `auto_ptr` 提供了一种语义学意义上的严格的所有权——类 `auto_ptr` 拥有的对象仅仅存储了一个指针。复制一个类 `auto_ptr` 的备份时，指针和所有权均被复制到目的地。如果多个 `auto_ptr` 类在同一时间拥有同一个对象，程序执行时，结果可能是不确定的。



提示 类 `auto_ptr` 的使用包括提供临时异常安全，用于动态分配内存，传递动态内存的所有权给函数，并返回之前为函数动态分配的内存。类 `auto_ptr` 不能满足标准库容器中元素要求的“构造复制性”和“可设置性”，而且在不确定的情况下，不能使用 `auto_ptr` 类型结果初始化标准库容器。

9.6.1 auto_ptr 类构造函数

```
explicit auto_ptr(X* p=0) throw()
```

在函数声明中，*this 保留指针 p。

```
auto_ptr(auto_ptr& a) throw();
```

功能描述：会调用 `a.release()`。*this 代表从 `a.release()` 返回的指针。

```
template<class Y> auto_ptr(auto_ptr<Y>& a) throw();
```

要求：`Y*` 可以被隐式地转换为 `X*`。其作用是调用 `a.release()`，*this 代表从 `a.release()` 返回的指针。

```
auto_ptr& operator = (auto_ptr& a) throw();
```

要求：表达式 `delete get()` 可以被很好地执行。执行该语句的同时会执行 `reset (a.release())`。函数返回值是 *this。

```
template<class Y> auto_ptr& operator = (auto_ptr<Y>& a) throw();
```

要求：函数 `Y*` 可以被隐式转换为 `X*`，同样，表达式 `delete get()` 可以被很好地执行。执行该语句的同时会隐式执行 `reset (a.release())`。函数返回值为 *this。

```
~auto_ptr() throw()
```

要求：表达式 `delete get()` 可以被很好地执行。其作用是删除 `get()` 函数返回的指针，即 `delete get()`。

9.6.2 类 auto_ptr 的成员及转换

1. 成员

```
X& operator* () const throw();
```

要求 `get() != 0`，返回值为函数 `get()` 返回的指针。

```
X* operator -> () const throw();
```

返回值为函数 `get()` 返回的指针。

```
X* get() const throw();
```

函数返回 `*this` 指针。

```
X* release() throw();
```

返回值为 `get()` 函数的返回值，之后 `*this` 指针即变成空指针。

```
void reset(X* p=0) throw();
```

功能：如果 `get()` 函数的返回值不等于指针 `p`，那么需要删除 `get()` 函数的返回值，相应的等效代码为：

```
if (get() != p)
    delete get();
```

函数执行之后，指针 `*this` 保存了指针 `p`。

2. 转换

```
auto_ptr(auto_ptr_ref<X> r) throw();
```

函数的执行效果：调用 `p.release()`，其中 `p` 是 `r` 内保存的指针。

```
template<class Y> operator auto_ptr_ref<Y> () throw();
```

函数返回值是类型为 `auto_ptr_ref` 变量，变量的值为 `*this`。

```
template<class Y> operator auto_ptr<Y> () throw();
```

函数的作用是调用 `release()` 函数。函数返回值是一个 `auto_ptr<Y>`，其中包含 `release()` 函数返回的指针。

```
auto_ptr& operator = (auto_ptr_ref<X> r) throw();
```

函数的作用是调用 `reset(p.release())`，`p` 为 `r` 中包含的引用。函数返回值为指针 `this`。

9.6.3 使用类 auto_ptr

类 `auto_ptr` 是一个模板类，用于管理动态内存分配的用法。例如，

```
void remodel(string & str)
{
    string* ps = new string(str);
    ...
    str = ps;
}
```

```

return;
}

```

读者认真阅读就会发现其中的缺陷。当调用时，该函数将分配堆中的内存，但不会收回该内存，导致内存泄漏。解决的办法是在 `return` 语句之前添加 `delete ps`。一旦忘记删除该内存，会导致意想不到的结果。编程者可能在不经意间删除或注释掉了某些代码，导致出现很多问题。例如，

```

void remodel(string & str)
{
    string* ps = new string(str);
    ...
    If(weird_thing())
        throw exception();
    str = * ps;
    delete ps;
    return;
}

```

上述代码中，一旦发生异常，`delete` 函数将不被执行，同样会发生内存泄漏的问题。异常抛出后，`throw()`后面的代码不会被执行。

内存泄漏的原因：当 `remodel()` 函数执行中止时，函数中的局部变量会被从堆中删除，指针 `ps` 占用的内存会被释放，但 `ps` 指向的内存不被释放。尤其对于基本类型，由于没有提供额外的服务，程序会自动释放 `ps` 指向的内存。对于类，可以通过析构函数机制销毁对象、释放内存、以便解决内存泄漏的问题。上述代码中 `ps` 的问题在于：其仅仅是常规指针，不是类对象。如果是类对象，在对象过期时，其析构函数会删除其指向的内存。

类 `auto_ptr` 恰恰满足了上述的需求！

模板类 `auto_ptr` 定义了类似指针的对象，可以将 `new` 获得（直接或间接）的地址赋给该对象。当 `auto_ptr` 对象过期时，其析构函数将使用 `delete` 来释放内存。若将 `new` 返回的地址赋给 `auto_ptr` 对象时，则不必记住释放的内存。在 `auto_ptr` 对象过期时，内存将自动释放。若需要创建 `auto_ptr` 对象，则必须包含头文件 `<memory>`（该文件中包括了 `auto_ptr` 模板）。使用通常的模板句法实例化所需类型的指针。类模板 `auto_ptr` 的声明形式如下：

```

template <class Type> class auto_ptr {
public:
    typedef Type element_type;
    explicit auto_ptr(Type * _Ptr = 0) throw(); //不抛出异常
    auto_ptr(auto_ptr <Type> & _Right) throw(); //不抛出异常
    template <class Other> operator auto_ptr <Other> () throw(); //不抛出异常
    template <class Other> auto_ptr <Type> & operator = (auto_ptr <Other> & _Right) throw(); //不抛出异常
    template <class Other> auto_ptr(auto_ptr <Other> & _Right);
    auto_ptr <Type> & operator = (auto_ptr <Type> & _Right);
};

```

```
    ~auto_ptr();  
    Type& operator* () const throw();  
    Type * operator -> () const throw();  
    Type * get() const throw();  
    Type * release() throw();  
    void reset(Type * _Ptr = 0);  
};
```

上述代码中的 `throw()` 意味着构造函数不引发异常。在请求 X 类型的 `auto_ptr` 对象会获得指向 X 类型的 `auto_ptr` 对象。

```
auto_ptr <double > pd (new double);  
auto_ptr <string > ps (new string);
```

`new double` 是 `new` 返回的指针，指向新分配的内存块。它是 `auto_ptr <double >` 构造函数的参数，即原型中形参 `p` 的实参。同样，`new string` 也是构造函数的实参。因此，要转换 `remodel()` 函数，需要经过 3 个步骤：

- 1) 包含头文件 `<memory >`。
- 2) 将指向 `string` 的指针替换为指向 `string` 的 `auto_ptr` 对象。
- 3) 删除 `delete` 语句。

前述的函数 `remodel()` 现在可以修改为以下代码：

```
#include <memory >  
void remodel(string & str)  
{  
    auto_ptr <string > ps (new string (str));  
    ...  
    if(weird_thing())  
        throw exception();  
    str = * ps;  
    //delete ps: No LONGER NEEDED  
    return;  
}
```

注意：`auto_ptr` 的构造函数是显式的，这意味着不存在从指针到 `auto_ptr` 对象的隐式类型转换。

```
auto_ptr <double > pd;  
double * p_reg = new double;  
pd = p_reg;  
pd = auto_ptr <double > (p_reg);  
auto_ptr <double > pauto = p_reg;  
auto_ptr <double > pauto (p_reg);
```

模板使程序员能将 `auto_ptr` 类型对象初始化为常规指针。

`auto_ptr` 是智能指针，仅仅是类似于指针。`auto_ptr` 作为类，其特性比指针更丰富。类 `auto_ptr` 被定义为在多方面与常规指针类似，并且 `auto_ptr` 类型的指针可以直接赋值给同类型的 `auto_ptr`。但这会引起一个问题：

值得记住的是, 虽然 STL 中定义了 `auto_ptr`, 但 `auto_ptr` 不是万能的。

```
auto_ptr<int> pi(new int [200]);
```

若使用 `new` 和 `new[]`, 则必须相应地使用 `delete` 和 `delete[]`。`auto_ptr` 类模板使用的是 `delete()` 函数, 不是 `delete[]`, 因此只能和 `new` 一起配对使用, 不能和 `new[]` 配对使用也不适用于动态数组的 `auto_ptr` 等同物。复制头文件 `<memory>` 中的 `auto_ptr` 模板, 将其重命名为 `auto_arr_ptr`, 然后将其修改, 使之使用 `delete[]`, 而不是使用 `delete()`, 以期添加对 `[]` 操作符的支持。

```
string vacation ("I wandered lonely as a cloud ");
auto_ptr<string> pvac (&vacation);
```



注意 只能对 `new` 分配的内存使用 `auto_ptr` 对象, 不要对由 `new[]` 分配的或通过声明变量分配的内存使用它。

还有一个常见的问题:

```
auto_ptr<string> ps (new string ("I reigned lonely as a cloud "));
auto_ptr<string> vocation;
vocation = ps;
```

若 `ps` 和 `vocation` 是常规指针, 则这两个指针将指向同一个 `string` 对象。其中一个另一个的副本。当 `ps` 和 `vocation` 均要过期时, 程序会试图删除同一个对象两次。这个问题应通过以下方法予以避免。

- 定义赋值操作符, 使之执行深复制。两个指针会指向不同的对象, 其中的一个对象是另一个对象的副本。
- 建立所有权概念, 即对于特定的对象, 只能有一个智能指针拥有它。智能指针的构造函数只能删除该智能指针拥有的对象, 并使赋值操作转让所有权。
- 创建智能最高的指针, 跟踪引用特定对象的智能指针数目, 这称为引用计数。例如, 赋值时, 计数将加 1; 指针过期时, 计数将减 1。仅当最后一个指针过期时, `delete` 才被调用。同样的方法也适用于复制构造函数。

详见例 9-10。

例 9-10

```
#include <memory>
#include <iostream>
#include <string>
using namespace std;
void main ()
{
    auto_ptr<string> str(new string ("Goose eggs. "));
    auto_ptr<string> str2 (str);
    auto_ptr<string> str3 (new string ("Chicken runs. "));

    cout << str.get () << endl;
    cout << str2.get () << endl;
```

```
cout << str3.get () << endl;

cout << "str: " << * str << endl;
cout << "str2: " << * str2 << endl;
cout << "str3: " << * str3 << endl;
}
```

例 9-10 的执行结果为:

```
00382D38
00382D38
00382F68
str: Goose eggs.
str2: Goose eggs.
str3: Chicken runs.
```

9.7 小结

本章的第一节简要介绍了通用工具库的情况，重点介绍了“对组”的概念；第二节讲述动态内存管理，重点讲述了特定算法和内存管理函数；第三节讲述堆的内存管理；第四节讲述部分辅助功能，例如数值极限、数值比较、两值交换等；第五节讲述日期和时间的相关内容，较详尽地讲述了时间相关的各种知识，并对各个时间函数进行举例说明；第六节讲述模板类 `auto_ptr`，详细讲述了模板类 `auto_ptr` 的优点。

通过对本章的学习，读者应对通用工具库以及 STL 的 `utility` 库有所了解，也应基本掌握对组、时间、模板类 `auto_ptr` 等对广大程序员帮助较大的知识。

第 10 章

语言支持类模板

本章将介绍在C++程序执行过程中的隐式签名功能和隐含（Implicit）产生的各种类型对象，同时介绍“这些隐含签名”和“定义相关类型”的头文件。随后各节将介绍通常类型的诸多定义，这些定义贯穿整个 STL，并被程序员广泛使用；还将介绍预定义类型的特性以及函数如何支持C++程序的启动和终止、如何支持动态内存管理、如何支持动态类型识别、如何支持异常处理、如何支持其他运行库等内容，详见表 10-1。

表 10-1 语言支持库汇总

名称	头文件	名称	头文件
Types	< cstdint >	Dynamic memory management	< new >
Implementation properties	< limits >、< climits >、< cfloat >	Exception handling	< exception >
Start and termination	< cstdlib >	Other runtime support	< cstdarg >、< csetjmp >、< ctype >、< csignal >、< cstdlib >

10.1 类型

在头文件 < cstdint > 中，类型定义主要包括 NULL、offsetof、ptrdiff_t 和 size_t。头文件中的内容和标准 C 库头文件 < stddef. h > 的内容大致相同，二者的差别在于：在 ISO/IEC 14882 中，宏 NULL 是一个 C++ 空指针常量。同样在国际标准中，宏 offsetof 接受一系列严格的类型参数。类型 type 应该是一个 POD 结构或一个 POD 联合体（Union）。使用宏 offsetof 的后果是导致静态数据成员或静态函数成员不能准确定义。

10.2 执行属性

头文件 < limits >、< climits > 和 < cfloat > 提供了依赖于基本类型的执行特性。数值限制组件提供了一个 C++ 程序，其中包含许多基本类型的执行属性。特殊化或专门化应该提供给每一个基本类型，无论是浮点型还是整型（包括布尔类型）。对于所有数值限制的特殊化，成员函数 is_specialized() 应取 true。

10.2.1 类模板 numeric_limits 及其成员

在数值限制模板 numeric_limits 中，对于所有声明的静态常量成员，在特殊化时应该定

义它们的值，以确保在整型常量表达式中，它们是可以使用的。非基本类型，如复数类型 `complex <T>`，不应该拥有特殊化。在头文件 `<limits>` 中，数值限制类的声明形式如下：

```
namespace std{
    template <class T> class numeric_limits;
    enum float_round style;
    enum float_denorm style;
    template <> class numeric_limits<bool>;
    template <> class numeric_limits<char>;
    template <> class numeric_limits<signed char>;
    template <> class numeric_limits<unsigned char>;
    template <> class numeric_limits<wchar_t>;
    template <> class numeric_limits<short>;
    template <> class numeric_limits<int>;
    template <> class numeric_limits<long>;
    template <> class numeric_limits<unsigned short>;
    template <> class numeric_limits<unsigned int>;
    template <> class numeric_limits<unsigned long>;
    template <> class numeric_limits<float>;
    template <class> numeric_limits<double>;
    template <class> numeric_limits<long double>;
}
```

模板类 `numeric_limits` 的声明形式如下：

```
namespace std{
    template <class T> class numeric_limits{
    public:
        static const bool is_specialized = false;
        static T min() throw();
        static T max() throw();
        static const int digits=0;
        static const int digits10=0;
        static const bool is_signed=false;
        static const bool is_integer=false;
        static const bool is_exact=false;
        static const int radix=0;
        static T epsilon() throw();
        static T round_error() throw();
        static const int min_exponent=0;
        static const int min_exponent10=0;
        static const int max_exponent=0;
        static const int max_exponent10=0;
        static const bool has_infinity=false;
        static const bool has_quiet_NaN=false;
        static const bool has_signaling_NaN=false;
        static const float_denorm_style has_denorm=denorm_absent;
```

```

        static const bool has_denorm_loss = false;
        static T infinity() throw();
        static T quiet_NaN() throw();
        static T signaling NaN() throw();
        static T denorm_min() throw();
        static const bool is_iec559 = false;
        static const bool is_bounded = false;
        static const bool is_modulo = false;
        static const bool traps = false;
        static const bool tininess_before = false;
        static const float_round_style round_style = round_toward_zero;
    }
}

```

类模板中的成员 `is_specialized` (特殊化类型) 用于区分基本类型 (具有特殊化性质) 和非数量 (非标量) 类型。除了 0 或 `false` 之外, 默认的数值极限模板类 `numeric_limits<T>` 应该包含所有成员。

下面逐一讲解数值限制类的各成员。

```
static T min() throw();
```

功能: 求解最小有限值。对于非标准化的浮点类型, 返回的是规范化的最小正值。使用此函数时, 该类型对象的成员 `is_bounded! = false`, 或 `is_bounded == false` 并且 `is_signed == false`。

```
static T max() throw();
```

功能: 求解最大有限值。对于所有特殊化对象, 只有当其成员 `is_bounded! = false` 时, 此函数才有意义。

```
static const int digits;
```

功能: 成员 `digits` 代表能够表示的基数位数。对于整数类型, `digits` 代表非负位的个数; 对于浮点类型, 基数的个数包含在尾数中。

```
static const int digits10;
```

功能: 表示不会丢失精度的十进制数的位数。只有当对象的成员 `is_bounded! = false` 时, 此函数才有意义。

```
static const bool is_signed;
```

功能: 如果是有符号数, 值为 `true`; 对于所有特殊化对象, 此函数均是有意义的。

```
static const bool is_integer;
```

功能: 如果类型是整数, 值为 `true`; 对于所有特殊化对象, 此函数均是有意义的。

```
static const bool is_exact;
```

功能: 如果类型使用一个精确的表示, 值为 `true`。所有类型均是严格的, 但并不是所有严格类型均为整数类型。例如, 合理的定点数表示也是严格准确的, 但并不是整数。

```
static const int radix;
```

功能: 对于浮点类型, 此函数代表指数表达式的基数或底; 对于整数类型, 此函数代表

表达式的底；对于所有特殊化对象，此函数均是有意义的。

```
static T epsilon() throw();
```

功能：epsilon() 函数返回值代表 1 和可表示的大于 1 的最小值之间的误差。对于所有浮点数类型，此函数均是有意义的。

```
static T round_error() throw();
```

功能：round_error() 函数可获取最大的舍入误差。

```
static const int min_exponent;
```

功能：min_exponent 函数返回一个最小负整数，幂的基数是成员 radix，但该整数小于最小的规范化的浮点数。对于所有浮点类型，此函数均是有意义的。

```
static const int min_exponent10;
```

功能：代表最小的负整数，10 是幂的底数或基数，幂的值在规范化的浮点数范围内。对于所有浮点类型，此函数均是有意义的。

```
static const int max_exponent;
```

功能：代表最大的正整数指数。该正整数的基数或底是 radix。

```
static const int max_exponent10;
```

功能：代表最大的正指数（幂）整数。该整数的底或基数为 10，幂系数是可表达的有限浮点数。

```
static const bool has_infinity;
```

功能：如果表达式是正的无穷，值为 true。对于所有浮点数，此函数均是有意义的。

```
static const bool has_quiet_NaN;
```

功能：如果类型是一个关于 quiet 的（非信号量，不是成员）表达式，值为 true。对于所有浮点类型，此函数是有意义的。对于所有特殊化对象，如果其成员 is_iec559 不等于 false，has_quiet_NaN 的值应该是 true。

```
static const bool has_signaling_NaN;
```

功能：如果类型是一个表达式，该表达式可以发出信号 “Not a Number.”。成员 has_signaling_NaN 为真(true)。此函数对于所有浮点类型均是有意义的。对于所有特殊化对象，当其成员 is_iec559 不等于 false 时，has_signaling_NaN 的值为 true。

```
static const float_denorm_style has_denorm;
```

功能：如果是类型允许可表示的数值，成员 has_denorm 的值为 denorm_present；如果是类型不允许可表示的值，成员 has_denorm 的值是 denorm_absent；在编译程序时，如果不能确定该类型是否是可表示的数值，成员 has_denorm 的值为 denorm_indeterminate。对于所有浮点类型，has_denorm 均是有意义的。

```
static const bool has_denorm_loss;
```

功能：如果作为一个非规范化的 loss，loss 的准确度可以被检测到，而不是一个非精确的结果，此时 has_denorm_loss 的值为真。

```
static T infinity() throw();
```

功能：如果函数调用成功，且返回一个有效的数值，该数值为正的无穷大。对于所有特殊化对象，当其成员 `has_infinity!` = `false` 时，此函数均是有意义的；当其成员 `is_iec559!` = `false` 时，该特殊化也是必需的。

```
static T quiet_NaN throw();
```

功能：如果函数正确调用话，函数返回值是一个非信号量，即“Not a Numer”。对于所有特殊化对象，当其成员 `has_quiet_NaN` 不等于 `false` 时，此函数均是有意义的，函数返回值为 `true`；当其成员 `is_iec559!` = `false` 时，特殊化对象也是必需的。

```
static T signaling_NaN() throw();
```

功能：如果函数调用成功，函数返回值是一个非信号量的表达式。只有当对象的成员 `has_signaling_NaN` 不等于 `false` 时，该特殊化对象才有意义，函数返回值为 `true`；当其成员 `is_iec559!` = `false` 时，特殊化对象才是必需的。

```
static T denorm_min() throw();
```

功能：函数返回正的最小规格化值。对于所有浮点类型，此函数均是有意义的。当特殊化对象的成员 `has_denorm` 等于 `false` 时，函数返回正的最小规格化值。

```
static const bool is_iec559;
```

功能：当且仅当数值类型遵从 IEC 559 标准时，其成员 `is_iec559` 为 `true`。对于所有浮点类型，此函数均是有意义的。

```
static const bool is_bounded;
```

功能：如果数值集合是可以表达的，并且类型是有限的，其成员 `is_bounded` 为 `true`。所有嵌入式类型均是有界的，成员是 `false` 对于那些任意的准确度类型。对于所有特殊化对象，此函数均是有意义的。

```
static const bool is_modulo;
```

功能：如果类型是“模”表达式，其成员 `is_modulo` 为 `true`；否则，为 `false`。通常，对于浮点类型，其值为 `false`；对于无符号整数，其值为 `true`；对于大多数机器，有符号整数的值也是 `true`。对于所有特殊化对象，此函数均是有意义的。

```
static const bool traps;
```

功能：对于该类型，如果捕获被执行，`traps` 的值为 `true`。对于所有特殊化对象，此函数均是有意义的。

```
static const bool tinyness_before;
```

功能：如果在舍入之前，“极小”被检测，`tinyness_before` 的值为 `true`。对于所有浮点类型，此函数均是有意义的。

```
static const float_round_style round_style;
```

功能：成员 `round_style` 代表舍入类型。对于整数类型的数值，其成员 `round_style` 的值为 `round_towards_zero`。

10.2.2 float_round_style 和 float_denorm_style

`float_round_style` 是一个枚举类型。其声明形式如下：

```
namespace std{
enum float_round_style{
    round_indeterminate = -1;
    round_toward_zero = 0;
    round_to_nearest = 1;
    round_toward_infinity = 2;
    round_toward_neg_infinity = 3;
};
}
```

根据数值的具体情况，浮点数算法的舍入模式归纳情况如下：

- `round_indeterminate` 如果舍入类型是不确定的
- `round_toward_zero` 如果舍入类型为 0
- `round_to_nearest` 如果舍入类型是最能够被接近的可描述的数值
- `round_toward_neg_infinity` 如果舍入类型是趋向于负无穷的情况

`float_denorm_style` 也是一个枚举类型，其声明形式如下：

```
namespace std{
enum float_denorm_style{
    denorm_indeterminate = -1;
    denorm_absent = 0;
    denorm_present = 1;
};
}
```

存在或缺乏非规范化可以归纳为以下几个方面：

- `denorm_indeterminate` 不能确定类型是否允许非规范化的取值
- `denorm_absent` 如果类型不允许非规格化的数值
- `denorm_present` 如果类型允许非规格化的取值

10.2.3 数值极限的特殊化

当数值限制类被特殊化时，提供所有模板类成员是必需的。然而，许多值在某种条件下才具有意义。任何没有意义的值全部被设置为 0 或 `false`。例如，

```
namespace std{
template < > class numeric_limits < float > {
public:
    static const bool is_specialized = true;
    inline static float min() throw() {return 1.1754;}
    inline static float max() throw() {return 3.40282;}
    static const int digits = 24;
    static const int digits10 = 6;
    static const bool is_signed = true;
    static const bool is_integer = false;
    static const bool is_exact = false;
    static const int radix = 2;
};
}
```

```

static const float epsilon() throw(){return 1.192;}
static const float round_error() throw(){return 0.5F;}
static const int min_exponent = -125;
static const int min_exponent10 = -37;
static const int max_exponent = 128;
static const int max_exponent10 = 38;
static const bool has_infinity = true;
static const bool has_quiet_NaN = true;
static const bool has_signaling_NaN = true;
static const float_denorm_style has_denorm = denorm_absent;
static const bool has_denorm_loss = false;
inline static float infinity() throw() {return ...;}
inline static float quiet_NaN() throw() {return ...;}
inline static float signaling_NaN() throw() {return ...;}
inline static float denorm_min() throw() {return ...;}
static const bool is_iec559 = true;
static const bool is_bounded = true;
static const bool is_modulo = false;
static const bool traps = true;
static const bool tinyness_before = true;
static const float_round_style round_style = round_to_nearest;
}
}

```

10.2.4 C 库函数

头文件 <climits> 中的内容和标准 C 库头文件 <limit.h> 是一致的。在头文件 <climits> 中，其内容主要包括的宏定义有 CHAR_BIT、INT_MAX、LONG_MIN、SCHAR_MIN、UCHAR_MAX、USHRT_MAX、CHAR_MAX、INT_MIN、MB_LEN_MAX、SHRT_MAX、UINT_MAX、CHAR_MIN、LONG_MAX、SCHAR_MAX、SHRT_MIN 和 ULONG_MAX。

在 C++ 语言或 STL 中的头文件 <cfloat> 中，宏定义主要包括 DLB_DIG、DBL_EPSILON、DBL_MANT_DIG、DBL_MAX、DBL_MAX_10_EXP、DBL_MAX_EXP、DBL_MIN、DBL_MIN_10_EXP、DBL_MIN_EXP、FLT_DIG、FLT_EPSILON、FLT_MANT_DIG、FLT_MAX、FLT_MAX_10_EXP、FLT_MAX_EXP、FLT_MIN、FLT_MIN_10_EXP、FLT_MIN_EXP、FLT_RADIX、FLT_ROUNDS、LDBL_DIG、LDBL_EPSILON、LDBL_MANT_DIG、LDBL_MAX、LDBL_MAX_10_EXP、LDBL_MAX_EXP、LDBL_MIN、LDBL_MIN_10_EXP 和 LDBL_MIN_EXP。

头文件 <cfloat> 中的内容和标准 C 库头文件 <float.h> 中的内容也是一致的。

10.2.5 应用举例

例 10-1

```

#include <iostream>
#include <limits>
using namespace std;

```

```
void main ()
{
//denorm_min ()
cout << "The smallest nonzero denormalized value (float). " << "object is: " << numeric_limits < float > ::denorm_min () << endl;
cout << "The smallest nonzero denormalized value (float). " << "object is: " << numeric_limits < int > ::denorm_min () << endl;
cout << "The smallest nonzero denormalized value (float). " << "object is: << numeric_limits < double > ::denorm_min () << endl;
//舍入为零
cout << "The smallest nonzero denormalized value (float). " << "object is: << numeric_limits < float > ::denorm_min () / 2 << endl;
cout << "The smallest nonzero denormalized value (float). " << "object is: " << numeric_limits < int > ::denorm_min () / 2 << endl;
cout << "The smallest nonzero denormalized value (float). " << "object is: << numeric_limits < double > ::denorm_min () / 2
    << endl;
//digits
    cout << "digits (float): " << numeric_limits < float > ::digits << endl;
    cout << "digits (double): " << numeric_limits < double > ::digits << endl;
    cout << "digits (int): " << numeric_limits < int > ::digits << endl;
    cout << "digits (_int64): " << numeric_limits < _int64 > ::digits << endl;
//epsilon
    cout << "epsilon (float): " << numeric_limits < float > ::epsilon () << endl;
    cout << "epsilon (double): " << numeric_limits < double > ::epsilon () << endl;
    cout << "epsilon (int): " << numeric_limits < int > ::epsilon () << endl;
    cout << "epsilon (long double): " << numeric_limits < long double > ::epsilon () << endl;
//has_denorm
    cout << "has_denorm (float): " << numeric_limits < float > ::has_denorm << endl;
    cout << "has_denorm (double): " << numeric_limits < double > ::has_denorm << endl;
    cout << "has_denorm (int): " << numeric_limits < long int > ::has_denorm << endl;
//has_denorm_loss
    cout << "has_denorm_loss (float): " << numeric_limits < float > ::has_denorm_loss << endl;
    cout << "has_denorm_loss (double): " << numeric_limits < double > ::has_denorm_loss << endl;
    cout << "has_denorm_loss (int): " << numeric_limits < long int > ::has_denorm_loss << endl;
//has_infinity
    cout << "Whether float objects have infinity: " << numeric_limits < float > ::has_infinity
    << endl;
    cout << "Whether double objects have infinity: " << numeric_limits < double > ::has_infinity
    << endl;
    cout << "Whether long int objects have infinity: " << numeric_limits < long int > ::has_infinity
    << endl;
```

```
//has_quiet_NaN
    cout << "Whether float objects have quiet_NaN: " << numeric_limits<float>::has_quiet_NaN << endl;
    cout << "Whether double objects have quiet_NaN: " << numeric_limits<double>::has_quiet_NaN << endl;
    cout << "Whether long int objects have quiet_NaN: " << numeric_limits<long int>::has_quiet_NaN << endl;
//has_signaling_NaN
    cout << "Whether float objects have a signaling_NaN: " << numeric_limits<float>::has_signaling_NaN << endl;
    cout << "Whether double objects have a signaling_NaN: " << numeric_limits<double>::has_signaling_NaN << endl;
    cout << "Whether long int objects have a signaling_NaN: " << numeric_limits<long int>::has_signaling_NaN << endl;
//infinity()
    cout << "The infinity for type float is: " << numeric_limits<float>::infinity() << endl;
    cout << "The infinity for type double is: " << numeric_limits<double>::infinity() << endl;
    cout << "The infinity for type long double is: " << numeric_limits<long double>::infinity() << endl;
//is_bounded
    cout << "Whether float objects have bounded set (是否有界的): " << numeric_limits<float>::is_bounded << endl;
    cout << "Whether double objects have bounded set (是否有界的): " << numeric_limits<double>::is_bounded << endl;
    cout << "Whether long int objects have bounded set (是否有界的): " << numeric_limits<long int>::is_bounded << endl;
    cout << "Whether unsigned char objects have bounded set (是否有界的): " << numeric_limits<unsigned char>::is_bounded << endl;
//is_exact
    cout << "Whether free of rounding errors(float): " << numeric_limits<float>::is_exact << endl;
    cout << "Whether free of rounding errors(double): " << numeric_limits<double>::is_exact << endl;
    cout << "Whether free of rounding errors(long int): " << numeric_limits<long int>::is_exact << endl;
    cout << "Whether free of rounding errors(unsigned char): " << numeric_limits<unsigned char>::is_exact << endl;
//is_iec559
    cout << "Whether conform to iec559 standards (float) (是否遵从 IEC559 标准.): " << numeric_limits<float>::is_iec559 << endl;
    cout << "Whether conform to iec559 standards (double) (是否遵从 IEC559 标准.): " << numeric_limits<double>::is_iec559 << endl;
```

```
        ::is_iec559 << endl;
    cout << "Whether conform to iec559 standards (int) (是否遵从 IEC559 标准.): " << numeric_limits <
float > ::is_iec559
        << endl;
    cout << "Whether conform to iec559 standards (unsigned char) (是否遵从 IEC559 标准.): " << numeric_
limits < float >
        ::is_iec559 << endl;
//is_integer
    cout << "Whether has an integral representation: " << numeric_limits < float > ::is_integer << endl;
    cout << "Whether has an integral representation: " << numeric_limits < double > ::is_integer <<
endl;
    cout << "Whether has an integral representation: " << numeric_limits < int > ::is_integer << endl;
    cout << "Whether has an integral representation: " << numeric_limits < unsigned char > ::is_integer
<< endl;
//is_modulo
    cout << "Whether has a modulo representation: " << numeric_limits < float > ::is_modulo << endl;
    cout << "Whether has a modulo representation: " << numeric_limits < double > ::is_modulo << endl;
    cout << "Whether has a modulo representation: " << numeric_limits < signed char > ::is_modulo <<
endl;
    cout << "Whether has a modulo representation: " << numeric_limits < unsigned char > ::is_modulo <<
endl;
//is_signed
    cout << "Whether be a signed representation: " << numeric_limits < float > ::is_signed << endl;
    cout << "Whether be a signed representation: " << numeric_limits < double > ::is_signed << endl;
    cout << "Whether be a signed representation: " << numeric_limits < signed char > ::is_signed <<
endl;
    cout << "Whether be a signed representation: " << numeric_limits < unsigned char > ::is_signed <<
endl;
//is_specialized
    cout << "Whether be an explicit " << "specialization in the class: " << numeric_limits < float > ::is_
specialized
        << endl;
    cout << "Whether be an explicit " << "specialization in the class: " << numeric_limits < float* > ::
is_specialized
        << endl;
    cout << "Whether be an explicit " << "specialization in the class: " << numeric_limits < int > ::is_
specialized << endl;
    cout << "Whether be an explicit " << "specialization in the class: " << numeric_limits < int* > ::is_
specialized << endl;
```

```
//max
    cout << "The maximum value for type float is: " << numeric_limits<float>::max() << endl;
    cout << "The maximum value for type double is: " << numeric_limits<double>::max() << endl;
    cout << "The maximum value for type float is: " << numeric_limits<int>::max() << endl;
    cout << "The maximum value for type short int is: " << numeric_limits<short int>::max() <<
endl;
//max_exponent
    cout << "The default radix is : " << numeric_limits<int>::radix << endl;
    cout << "The maximum radix - based exponent for type float is: " << numeric_limits<float>::
max_exponent << endl;
    cout << "The maximum radix - based exponent for type double is: " << numeric_limits<double
>::max_exponent
    << endl;
    cout << "The maximum radix - based exponent for type long double is: " << numeric_limits<
long double >
    ::max_exponent << endl;
    cout << "The maximum radix - based exponent for type int is: " << numeric_limits<int>::max
_exponent << endl;
//max_exponent10
    cout << "The default radix is : " << numeric_limits<int>::radix << endl;
    cout << "The maximum base 10 exponent for type float is: " << numeric_limits<float>::max_expo
nent10 << endl;
    cout << "The maximum base 10 exponent for type double is: " << numeric_limits<double>::max_expo
nent10 \
    << endl;
    cout << "The maximum base 10 exponent for type long double is: " << numeric_limits<longdouble>::
max_exponent10 << endl;
//min
cout << "The minimum value for type float is: " << numeric_limits<float>::min() << endl;
cout << "The minimum value for type double is: " << numeric_limits<double>::min() << endl;
cout << "The minimum value for type float is: " << numeric_limits<int>::min() << endl;
cout << "The minimum value for type short int is: " << numeric_limits<short int>::min() << endl;
//min_exponent
cout << "The minimum radix - based exponent for type float is: " << numeric_limits<int>::min_expo
nent << endl;
cout << "The minimum radix - based exponent for type float is: " << numeric_limits<float>::min_ex
ponent << endl;
cout << "The minimum radix - based exponent for type double is: " << numeric_limits<double>::min
_exponent << endl;
cout << "The minimum radix - based exponent for type long double is: " << numeric_limits<long double
>::min_exponent
    << endl;
```

```
//min_exponent10
    cout << "The minimum base 10 exponent for type float is: " << numeric_limits<int>::min_exponent10
<< endl;
    cout << "The minimum base 10 exponent for type float is: " << numeric_limits<float>::min_expo-
nent10 << endl;
    cout << "The minimum base 10 exponent for type double is: " << numeric_limits<double>::min_expo-
nent10 << endl;
    cout << "The minimum base 10 exponent for type long double is: " << numeric_limits<long double>::
min_exponent10
        << endl;
//quiet_NaN
    cout << "The quiet NaN for type float is: " << numeric_limits<float>::quiet_NaN( ) <<
endl;
    cout << "The quiet NaN for type int is: " << numeric_limits<int>::quiet_NaN( ) << endl;
    cout << "The quiet NaN for type long double is: " << numeric_limits<double>::quiet_NaN(
) << endl;
//radix
    cout << "The base for type float is: " << numeric_limits<float>::radix << endl;
    cout << "The base for type int is: " << numeric_limits<int>::radix << endl;
    cout << "The base for type long double is: " << numeric_limits<long double>::radix <<
endl;
//round_error
    cout << "The maximum rounding error for type float is: " << numeric_limits<float>::round
_error( ) << endl;
    cout << "The maximum rounding error for type int is: " << numeric_limits<int>::round_er-
ror( ) << endl;
    cout << "The maximum rounding error for type double is: " << numeric_limits<double>::round_er-
ror( ) << endl;
//round_style
    cout << "The rounding style for a double type is: " << numeric_limits<double>::round_
style << endl;
    cout << "The rounding style for a double type is now: " << numeric_limits<float>::round
_style << endl;
    cout << "The rounding style for an int type is: " << numeric_limits<int>::round_style <
< endl;
//signaling_NaN
    cout << "The signaling NaN for type float is: " << numeric_limits<float>::signaling_NaN
( ) << endl;
    cout << "The signaling NaN for type int is: " << numeric_limits<int>::signaling_NaN( ) <
< endl;
    cout << "The signaling NaN for type double is: " << numeric_limits<double>::signaling_
NaN( ) << endl;
//tinyness_before
    cout << "Whether float types can detect tinyness before rounding: " << numeric_limits<float>::ti-
nyness_before << endl;
```

```

    cout << "Whether double types can detect tinyness before rounding: " << numeric_limits<double>::
tinyness_before
        << endl;
    cout << "Whether long int types can detect tinyness before rounding: " << numeric_limits<long int>
::tinyness_before
        << endl;
    cout << "Whether unsigned char types can detect tinyness before rounding: " << numeric_limits<un-
signed char>::
        tinyness_before << endl;
//traps
    cout << "Whether float types have implemented trapping: " << numeric_limits<float>::traps <
< endl;
    cout << "Whether double types have implemented trapping: " << numeric_limits<double>::
traps << endl;
    cout << "Whether long int types have implemented trapping: " << numeric_limits<long int>::
traps << endl;
cout << "Whether unsigned char types have implemented trapping: " << numeric_limits<unsigned char>
::traps << endl;
}

```

例 10-1 的执行结果如下：

```

The smallest nonzero denormalized value(float). object is: 1.4013e-045
The smallest nonzero denormalized value(float). object is: 0
The smallest nonzero denormalized value(float). object is: 4.94066e-324
The smallest nonzero denormalized value(float). object is: 0
The smallest nonzero denormalized value(float). object is: 0
The smallest nonzero denormalized value(float). object is: 0
digits(float): 24
digits(double): 53
digits(int): 31
digits(_int64): 0
epsilon(float): 1.19209e-007
epsilon(double): 2.22045e-016
epsilon(int): 0
epsilon(long double): 2.22045e-016
has_denorm(float): 1
has_denorm(double): 1
has_denorm(int): 0
has_denorm_loss(float): 1
has_denorm_loss(double): 1
has_denorm_loss(int): 0
Whether float objects have infinity: 1
Whether double objects have infinity: 1
Whether long int objects have infinity: 0
Whether float objects have quiet_NaN: 1
Whether double objects have quiet_NaN: 1

```

```

Whether long int objects have quiet_NaN: 0
Whether float objects have a signaling_NaN: 1
Whether double objects have a signaling_NaN: 1
Whether long int objects have a signaling_NaN: 0
The infinity for type float is: 1.# INF
The infinity for type double is: 1.# INF
The infinity for type long double is: 1.# INF
Whether float objects have bounded set (是否有界的): 1
Whether double objects have bounded set (是否有界的): 1
Whether long int objects have bounded set (是否有界的): 1
Whether unsigned char objects have bounded set (是否有界的): 1
Whether free of rounding errors(float): 0
Whether free of rounding errors(float): 0
Whether free of rounding errors(float): 1
Whether free of rounding errors(float): 1
Whether conform to iec559 standards (float) (是否遵从 IEC559 标准.): 1
Whether conform to iec559 standards (double) (是否遵从 IEC559 标准.): 1
Whether conform to iec559 standards (int) (是否遵从 IEC559 标准.): 1
Whether conform to iec559 standards (unsigned char) (是否遵从 IEC559 标准.): 1
Whether has an integral representation: 0
Whether has an integral representation: 0
Whether has an integral representation: 1
Whether has an integral representation: 1
Whether has a modulo representation: 0
Whether has a modulo representation: 0
Whether has a modulo representation: 1
Whether has a modulo representation: 1
Whether be a signed representation: 0
Whether be an explicit specialization in the class: 1
Whether be an explicit specialization in the class: 0
Whether be an explicit specialization in the class: 1
Whether be an explicit specialization in the class: 0
The maximum value for type float is: 3.40282e+038
The maximum value for type double is: 1.79769e+308
The maximum value for type float is: 2147483647
The maximum value for type short int is: 32767
The default radix is : 2
The maximum radix-based exponent for type float is: 128
The maximum radix-based exponent for type double is: 1024
The maximum radix-based exponent for type long double is: 1024
The maximum radix-based exponent for type int is: 0
The default radix is : 2
The maximum base 10 exponent for type float is: 38

```

```
The maximum base 10 exponent for type double is: 308
The maximum base 10 exponent for type long double is: 308
The minimum value for type float is: 1.17549e-038
The minimum value for type double is: 2.22507e-308
The minimum value for type float is: -2147483648
The minimum value for type short int is: -32768
The minimum radix-based exponent for type float is: 0
The minimum radix-based exponent for type float is: -125
The minimum radix-based exponent for type double is: -1021
The minimum radix-based exponent for type long double is: -1021
The minimum base 10 exponent for type float is: 0
The minimum base 10 exponent for type float is: -37
The minimum base 10 exponent for type double is: -307
The minimum base 10 exponent for type long double is: -307
The quiet NaN for type float is: -1.#IND
The quiet NaN for type int is: 0
The quiet NaN for type long double is: -1.#IND
The base for type float is: 2
The base for type int is: 2
The base for type long double is: 2
The maximum rounding error for type float is: 0.5
The maximum rounding error for type int is: 0
The maximum rounding error for type double is: 0.5
The rounding style for a double type is: 1
The rounding style for a double type is now: 1
The rounding style for an int type is: 0
The signaling NaN for type float is: -1.#INF
The signaling NaN for type int is: 0
The signaling NaN for type double is: -1.#INF
Whether float types can detect tinyness before rounding: 1
Whether double types can detect tinyness before rounding: 1
Whether long int types can detect tinyness before rounding: 0
Whether unsigned char types can detect tinyness before rounding: 0
Whether float types have implemented trapping: 1
Whether double types have implemented trapping: 1
Whether long int types have implemented trapping: 0
Whether unsigned char types have implemented trapping: 0
```

10.3 程序的启动和终止

前面已经简要介绍了程序的启动和终止。头文件 `<cstdlib>` 包含两个宏：`EXIT_FAILURE` 和 `EXIT_SUCCESS`；另外还包含 3 个函数：`abort()`、`atexit()` 和 `exit()`。这 3 个函数在第 8 章已经有所涉及，结合前面的内容，本节将继续讲述它们的使用方法。

头文件 `<stdlib.h>` 中的内容和头文件 `<cstdlib>` 大致相似。

1. abort() 和 atexit() 函数

在 C++ 标准中, abort() 函数的作用是终止一个进程, 可以具有额外的功能。对于自动静态存储对象, 执行过程中不调用对象的析构器, 并且不调用 atexit() 函数。

atexit() 函数的声明形式如下:

```
extern "C" int atexit (void(* f)(void));  
extern "C" int atexit (void(* f)(void));
```

功能: atexit() 函数将注册 (或设置) 指针 f 指向的函数, 并且指针 f 指向的函数不带参数。当执行被 atexit() 函数注册的函数时, 由于该指针 f 指向的函数不提供句柄, 并且无法抛出异常, 一旦指针 f 指向的函数不能被有效控制, terminate() 函数将被调用。

局限性: 在使用 atexit() 时, 代表函数的指针 f 至少可以支持注册 32 种函数。

返回值: 如果指针 f 指向的函数注册成功, atexit() 函数返回 0; 否则, atexit() 函数返回非零值。

2. exit (int status) 函数

在 C++ 标准中, exit() 函数也可具有额外的功能。

首先, 具有静态存储类型的对象被破坏, 通过 atexit() 函数注册的函数被调用。具有静态存储类型的非局部类型对象被破坏, 破坏顺序是和其构造器被执行顺序的逆序执行的。当调用 exit() 函数时, 自动存储类型的对象不会被破坏。被 atexit() 函数注册的函数是按其最初注册顺序的逆序执行的。除此之外, 任何之前注册的函数调用之后, 其他函数才会被调用。在静态存储类型的非局部型对象被初始化之前, 使用 atexit() 函数注册的函数是不会被调用的, 直到该对象的析构过程结束。在非局部类型的静态存储类型对象初始化之后, 该对象的析构过程开始之前, 使用 atexit() 函数注册的任意函数, 才会被调用。任意函数调用局部的静态类型对象的析构器时, 只有该析构器是在该对象的构造器的末尾被 atexit() 函数注册的, 该局部静态类型对象才被破坏。

其次, 具有不可写入的缓冲区数据被刷新时, 所有开放 C 流被关闭, 并且所有使用 tmpfile() 函数创建的临时文件均被删除。

最后, 函数将返回至调用该函数的环境。如果参数 status 是 0 或宏 EXIT_SUCCESS, 函数返回的状态为“成功终止程序”; 如果参数是 EXIT_FAILURE, 函数返回的状态为“未成功终止”。除此之外, 返回的状态是由实际环境决定的。并且 exit() 函数绝不会返回到它的调用者, 而是直接终止程序。

10.4 动态内存管理

头文件 <new> 定义了几种函数。这些函数用于负责程序中动态内存的分配。同时, 头文件也定义了负责报告内存管理错误的部件。头文件 <new> 所包含的内容如下:

```
namespace std {  
    class bad_alloc;  
    struct nothrow_t{};
```

```
extern const nothrow_t nothrow;
typedef void(* new_handler)();
new_handler set_new_handler(new_handler new_p) throw();
}

void* operator new(std::size_t size) throw (std::bad_alloc);
void* operator new(std::size_t size, const std::nothrow_t& ) throw();
void* operator delete(void* ptr) throw();
void* operator delete(void* ptr, const std::nothrow_t&) throw();
void* operator new[](std::size_t size) throw (std::bad_alloc);
void* operator new[](std::size_t size, const std::nothrow_t&) throw();
void* operator delete[](void* ptr) throw();
void* operator delete[](void* ptr, const std::nothrow_t&) throw();
void* operator new(std::size_t size, void* ptr) throw();
void* operator new[](std::size_t size, void* ptr) throw();
void* operator delete(void* ptr, void* ) throw();
void* operator delete[](void* ptr, void* ) throw();
```

10.4.1 内存的分配和释放

首先介绍操作符 `new` 和 `delete`。

1. 单个对象的形式

操作符 `new` 的声明形式如下：

```
void* operator new(std::size_t size) throw (std::bad_alloc);
```

使用 `new` 表达式实现分配内存空间，参数 `size` 代表分配内存的字节大小。该内存代表了对象的大小。

在C++ 程序中，可以定义任意函数，甚至该函数的命名标识可以取代C++ 标准库中定义的默认版本。如果分配内存成功，`new` 操作符可以返回一个非空指针；否则，会抛出 `bad_alloc` 类型的异常。

操作符 `new` 也会执行一些默认的行为法则。

1) 执行循环。在循环中，函数通常首先尝试分配必要的内存。而是否尝试调用函数 `malloc()`，也不是特定的。

2) 如果尝试成果，会返回指针，该指针指向分配的内存。否则，如果函数 `set_handler()` 的最后一个参数是一个空指针，会抛出异常 `bad_alloc`。

3) 要不然，函数会调用当前的句柄(`new_handler`)。如果被调用的函数返回，循环会重复执行。

4) 如果分配需要的内存成功或者被调用的函数句柄(`new_handler`)没有正常返回，循环会中止。

```
void* operator new(std::size_t size, const std::nothrow_t&) throw();
```

上述代码和第一种 `new` 的用法相似。唯一不同的是：当上述形式的 `new` 被调用时，C++ 程序会提供一个空指针作为错误标识，而不是抛出 `bad_alloc` 类型的异常。

同样，任意C++ 程序均可定义一个函数，该函数可以替换C++ 标准库中的默认版本。

当使用 `new()` 分配适当的内存空间时，返回的是一个非空指针；否则，返回的是一个空指针。操作符 `new` 的 `nothrow` 版本返回的指针和常规版本返回的指针是一样的。函数的替代版本同样需要满足这个要求。

同样，第二种 `new` 操作符也会执行一些默认的行为法则。

1) 执行一个循环。在循环执行过程中，函数首先尝试分配必要的内存。该尝试涉及调用标准 C 库 `malloc()` 函数。

2) 如果尝试成功，返回指向所分配内存的指针。否则，如果 `set_new_handler()` 函数的最后一个参数是空指针，`new` 函数会返回空指针。

3) 要不然，函数调用当前的 `new_handler`。如果被调用的函数返回，循环会继续重复执行。

4) 当尝试分配必要的内存成功时，或者当被调用的 `new_handler` 代表的函数没有返回时，循环会中止。如果通过抛出异常的形式，被调用的 `new_handler` 中止，`new()` 函数会返回一个空指针。例如，

```
T* p1 = new T; //如果分配内存失败,会抛出 bad_alloc 类型的异常
T* p2 = new(nothrow) T; //如果分配内存失败,会返回 0
```

操作符 `delete` 的两种使用形式：

```
void operator delete(void* ptr) throw();
void operator delete(void* ptr, const std::nothrow_t&) throw();
```

功能描述：释放内存的功能是通过 `delete` 表达式被调用的。目的是使指针 `ptr` 的值无效。任一C++ 程序可能会定义一个函数，该函数命名可以取代C++ 标准库中的默认版本。之前调用函数 `new` 时，指针 `ptr` 获取到相应的内存指针或空指针。同样，`delete` 函数也存在以下默认属性。

1) 对于空指针，不采取任何措施。

2) 当 `ptr` 指针是非空数值时，其数值应该是 `new` 函数被调用时赋予的值，通过对其调用 `delete()` 函数，使该指针无效。对于这样的非空指针，`delete()` 函数执行之后，会收回之前使用默认 `new()` 函数开辟的内存。



提示 对于 `new` 函数或其他（`calloc()`、`malloc()`、`realloc()` 等）函数开辟的内存，并不能确定在什么条件下才能回收这些内存。

2. 数组形式 (array forms)

```
void* operator new[](std::size_t size) throw(std::bad_alloc);
```

功能描述：使用 `new` 可以开辟数组形式的内存空间。内存大小是 `new` 的参数 `size`，数组的大小应该小于或等于 `new` 的参数 `size`。同样，在C++ 程序中可以定义函数，该函数的签名可以替换C++ 标准库中的默认版本。函数返回值是指针类型。

```
void* operator new[](std::size_t size, const std::nothrow_t&) throw();
```

功能描述：和上述内容相同，除了在调用 `new` 函数失败时，返回空指针之外，并不抛

出 `bad_alloc` 类型的异常。而 `new[]` 函数调用失败之后，会抛出 `bad_alloc` 类型的异常。任一 C++ 程序可以定义一个函数，该函数的命名同样可以替换 C++ 标准库中的默认版本(同名)。

两种 `new[]` 函数的使用方法是相似的，函数返回值类型是一致的。

```
void operator delete[](void* ptr) throw();  
void operator delete[](void* ptr, const std::nothrow_t&) throw();
```

功能描述：数组形式的内存释放(`delete[]`)函数在被调用时，主要目的是使该内存指针无效。

对于 C++ 程序，在程序中定义的函数可以替换 C++ 标准库中的同名默认版本。`delete[]` 函数的参数指针 `ptr` 允许的值可能是空指针，或者 `ptr` 的值是之前调用 `new[]` 函数获得的内存指针。同样，在使用 `delete[]` 函数时，需要注意的是：

1) 如果指针 `ptr` 的值是空指针，`delete[]` 函数不执行任何措施。

2) 如果指针 `ptr` 的值是其他数值，其值应该是之前调用 `new[]` 函数时获取的内存指针。对于一个非空值的指针 `ptr`，其之前被分配的内存将被回收。

3) 头文件 `<cstdlib>` 中声明的内存分配函数主要包括 `calloc()`、`malloc()` 或 `realloc()`。调用这些函数会分配相应的内存，并且无论在什么条件下，使用这些函数可以重新分配 `delete[]` 回收的内存。

3. Placement forms

在 C++ 库中，有部分函数是保留函数，是不能被自定义的同名函数取代的。这些函数也不可能代替保留的 `new` 和 `delete` 的特定形式。

形式一：

```
void* operator new (std::size_t size, void* ptr) noexcept;
```

函数返回指针 `ptr`。其目的是执行其他行为。例如，

```
void* place = operator new (sizeof(Something));
```

```
Something* p = new (place) Something();
```

形式二：

```
void* operator new[] (std::size_t size, void* ptr) noexcept;
```

函数返回值为 `ptr`。其目的是执行其他行为。

形式三：

```
void* operator delete (void* ptr, void* ) noexcept;
```

其目的是不执行任何行为。通过采用抛出异常的形式，在 `new` 表达式初始化时终止被调用的默认函数。

形式四：

```
void* operator delete[] ( void* ptr , void* ) noexcept;
```

不执行任何行为。通过采用抛出异常的形式，在 `new` 表达式初始化时终止被调用的默认函数。

10.4.2 内存分配错误

类 `bad_alloc` 的声明形式为：

```
namespace std{
    class bad_alloc: public exception{
public:
    bad_alloc() throw();
    bad_alloc(const bad_alloc&) throw();
    bad_alloc& operator = (const bad_alloc&) throw();
    virtual ~bad_alloc() throw();
    virtual const char* what() const throw();
    }
}
```

类 `bad_alloc` 定义了诸多异常对象的类型。这些异常对象是在分配内存失败时所抛出的异常类型对象。下面分别介绍类中的各个函数。

1. `bad_alloc() throw()`

使用类 `bad_alloc` 构造一个对象，`bad_alloc()` 函数恰好是类的构造器。

调用 `what()` 函数时，会导致定义所构造的新对象。

2. `bad_alloc(const bad_alloc&) throw()`

```
bad_alloc(const bad_alloc&) throw();
```

```
bad_alloc& operator = (const bad_alloc&) throw();
```

这两个函数的功能是实现完成一个类 `bad_alloc` 的备份。

3. `virtual const char* what() const throw()`

函数返回值是一个 NTBS。

4. `typedef new_handler`

`new_handler()` 函数产生一个新的句柄。新产生的 `new_handler` 类型的句柄应该执行下述的行为之一：

- 1) 尽量开辟更多的内存，并返回该内存。
- 2) 抛出 `bad_alloc` 类型的异常，或者抛出类 `bad_alloc` 的派生类的异常。
- 3) 调用 `abort()` 函数或者 `exit()` 函数。

5. `set_new_handler`

其声明形式为：

```
new_handler set_new_handler(new_handler new_p) throw();
```

功能：设置参数 `new_p` 指定的函数作为当前的 `new_handler`。

返回值：第一次调用时，函数返回 0；通常函数返回值是之前设置的 `new_handler` 类型的句柄。

10.4.3 应用举例

针对 10.4.1 节和 10.4.2 节的内容，下面使用例 10-2 和例 10-3 来具体说明上述相关内

容的使用方法。

例 10-2

```
#include <iostream>
#include <new>
using namespace std;
class MyClass{
public:
    int imember;
public:
    MyClass()
    {
        cout << "Construction MyClass. " << this << endl;
    }
    ~MyClass()
    {
        imember = 0;
        cout << "Destructing MyClass. " << this << endl;
    }
};
void __cdecl myfun()
{
    cout << "handler test. " << endl;
    throw bad_alloc();
    return;
}
void main()
{
//1
    MyClass* fpM = new MyClass;
    delete fpM;
    char x[sizeof(Myclass)];
    MyClass* fpM2 = new (&x[0]) MyClass;
    fpM2 -> ~Myclass();
    cout << "The address of x[0] is : " << (void* )&x[0] << endl;
    MyClass* fpM3 = new MyClass;
    delete fpM3;
//2
    char* int_p;
    try{
        //int* int_p = new int[99999];
        int_p = new char[ ~size_t(0)/2];
        delete[] int_p;
    }
    catch(bad_alloc &e)
    {
```

```
        cout << e.what() << endl;
    }
//3
    bad_alloc ba;
    cout << ba.what() << endl;
}
```

例 10-2 的执行结果为:

```
Construction MyClass. 00393710
Destructing MyClass. 00393710
Construction MyClass. 0012FF48
Destructing MyClass. 0012FF48
The address of x [0] is: 0012FF48
Construction MyClass. 00393710
Destructing MyClass. 00393710
bad allocation
bad allocation
```

例 10-3

```
#include <new >
#include <iostream >
using namespace std;
void __cdecl newhandler ( )
{
    cout << "The new_handler is called:" << endl;
    throw bad_alloc ( );
    return;
}
int main ( )
{
    set_new_handler (newhandler);
    try
    {
        while ( 1 )
        {
            new int[5000000];
            cout << "Allocating 5000000 ints." << endl;
        }
    }
    catch ( exception e )
    {
        cout << e.what ( ) << " xxx" << endl;
    }
    cin.get ( );
}
```

例 10-3 的执行结果为:

```
Allocating 5000000 ints.
...
Allocating 5000000 ints.
Allocating 5000000 ints.
The new_handler is called:
bad allocation xxx
```

10.5 类型标识符

本节主要讲述 3 个类: `type_info`、`bad_cast` 和 `bad_typeid`。其中, 类 `type_info` 用以描述类型信息; 类 `bad_cast` 用于定义异常对象的类型, 并抛出该异常对象; 类 `bad_typeid` 同样用于定义异常对象的类型。头文件 `<typeinfo>` 中定义了和类型信息相关的一种类型, 同时也定义了用于报告动态类型识别错误的两种类型。

10.5.1 类 `type_info`

类 `type_info` 的声明形式为:

```
namespace std{
    class type_info{
    public:
        virtual ~type_info();
        bool operator == (const type_info& rhs) const;
        bool operator != (const type_info& rhs) const;
        bool before(const type_info& rhs) const;
        const char* name() const;
    private:
        type_info(const type_info& rhs);
        type_info& operator = (const type_info& rhs);
    };
}
```

类 `type_info` 用以描述类型信息。类的对象有效地存储了一个类型的指针以及适合两个类型比较 (或排序) 的运算符。类型的名称、编码规则和排序顺序均是不确定的, 并且程序之间也存在差异。

```
bool operator == (const type_info& rhs) const
```

功能：使用 rhs 和当前对象相比较。如果两个数值是同一数据类型，函数返回 true。

```
bool operator! = (const type_info& rhs) const;
```

返回值：! (* this == rhs)。

```
bool before (const type_info& rhs) const;
```

功能：使用参数 rhs 和当前对象进行比较。

返回值：如果当前对象在排序序列中位于 rhs 之前，函数返回 true。

```
const char* name() const;
```

返回值：一个已定义的 NTBS。



信息可能是以空 (NULL) 为终止符的多字节字符串，适合作为一个宽字符串 (wstring) 转换和显示。

```
type_info(const type_info& rhs);
type_info& operator = (const type_info& rhs);
```

此函数的功能是实现类 type_info 对象的备份。



由于类 type_info 的复制构造器和可赋值的操作符是私有成员，因此该类型的对象是不能被复制的。

使用类 type_info 编写程序是比较麻烦的。因为其构造函数是私有成员。

10.5.2 类 bad_cast

类 bad_cast 的声明形式为：

```
namespace std{
    class bad_cast : public exception{
    public:
        bad_cast() throw();
        bad_cast(const bad_cast&) throw();
        bad_cast& operator = (const bad_cast&) throw();
        virtual ~bad_cast() throw();
        virtual const char* what() const throw();
    }
}
```

类 bad_cast 是用于定义异常对象的类型，这类异常对象在实现过程中会被抛出，用于汇报一个无效的 dynamic-cast 表达式的执行错误。

bad_cast() throw() 函数是该类的构造函数。构造器可以构造一个该类的对象。在使用新构造对象调用 what() 函数时，其结果是预先定义的。

构造函数还有一种形式：

```
bad_cast(const bad_cast& ) throw();
bad_cast& operator = (const bad_cast& ) throw();
```

功能：创建类 bad_cast 的对象或者该类对象的备份。

```
virtual const char* what() const throw();
```

函数返回值是已定义的 NTBS。



提示 返回值的提示信息是以空 (NULL) 为结束标识符的多字节字符串, 也适用于宽字节字符串 `wstring` 的转换和显示。

10.5.3 类 `bad_typeid`

类 `bad_typeid` 的声明形式为:

```
namespace std{
    class bad_typeid : public exception{
    public:
        bad_typeid() throw();
        bad_typeid(const bad_typeid&) throw();
        bad_typeid& operator = (const bad_typeid&) throw();
        virtual ~bad_typeid() throw();
        virtual const char* what() const throw();
    }
}
```

类 `bad_typeid` 定义了异常对象的类型。在程序执行过程中, 如果在 `typeid` 类型的表达式中发生空指针现象, 这类异常对象会被以异常的形式抛出。

`bad_typeid() throw()` 构造函数的功能是构造一个 `bad_typeid` 类型的对象。



提示 对于新生成的对象, 调用 `what()` 函数的结果是预先定义的。

```
bad_typeid(const bad_typeid&) throw();
bad_typeid& operator = (const bad_typeid&) throw();
```

功能: 实现类 `bad_typeid` 的对象的备份。

```
virtual const char* what() const throw();
```

函数返回值是已定义的 NTBS。



提示 返回值信息是一个以空 (NULL) 为结束标识符的多字节字符串, 也适合作为宽字节字符串的转换和显示。

10.5.4 操作符 `typeid`

`typeid` 操作符使程序能够询问某表达式的类型。

`typeid` 操作符的表达式为:

```
typeid(e);
```

其中 `e` 是任意表达式或者是类型名。

若表达式的类型是“类”类型, 并且该类包含一个或多个虚函数, 则表达式的动态类型可能不同于它的静态编译类型。例如, 若表达式是对基类指针引用, 则该表达式的静态编译类型是基类类型; 若指针实际指向派生类对象, 则 `typeid` 操作符将指出表达式的类型是派生类型。

typeid 操作符可以与任何类型的表达式一起使用。内置类型的表达式以及常量都可以用作 typeid 操作符的操作数。若操作数不是“类”类型或者是没有虚函数的类，则 typeid 操作符指出操作数的静态类型；若操作数是定义至少一个虚函数的“类”类型，则在运行时计算类型。

typeid 操作符的结果为 type_info 的标准库类型的对象引用。要使用类 type_info，必须包含库头文件 <typeinfo>。

typeid 最常见的用途是比较两个表达式的类型，或者将表达式的类型与特定类型相比较。

10.5.5 操作符 dynamic_cast

使用 dynamic_cast 操作符将基类对象的引用或指针转换为同一继承层次中其他类型的引用或指针。与 dynamic_cast 一起使用的指针必须是有效的（它必须是 0 或指向一个对象）。

与其他强制类型转换不同，dynamic_cast 涉及运行过程中的类型检查。若绑定到引用或指针的对象不是目标类型的对象，则 dynamic_cast 失败；若转换到指针类型的 dynamic_cast 失败，则 dynamic_cast 的结果是 0 值。若转换到引用类型的 dynamic_cast 失败，则抛出一个 bad_cast 类型的异常。

dynamic_cast 操作符一次执行两个操作，即先验证被请求的转换是否有效，只有转换有效，操作符才实际进行转换。一般而言，引用或指针所绑定对象的类型在编译时是未知的，基类指针可以赋值为指向派生类对象。同样基类的引用可以用派生类对象初始化，因此 dynamic_cast 操作符执行的验证必须在运行时进行。

使用 dynamic_cast 将基类指针转换为派生类指针，可以使用 dynamic_cast 将基类引用转换为派生类引用，这种 dynamic_cast 操作形式如下：

```
dynamic_cast <Type& > (val)
```

Type 是转换的目标类型，而 val 是基类类型的对象。仅当 val 实际引用一个 Type 类型对象或者 val 是一个 Type 派生类的对象时，dynamic_cast 操作才将操作数 val 转换为希望的 Type& 类型。如果不存在空引用，不可能对引用使用指针强制类型转换的检查策略；相反，当转换失败时，会抛出一个 std::bad_cast 异常。该异常是在 STL 库的头文件 <typeinfo> 中定义的。

10.5.6 应用举例

下面使用例 10-4 来阐述 10.5 节相关内容的使用方法。

例 10-4

```
#include <iostream>
#include <typeinfo>
using namespace std;
class Circle {
public:
    virtual void myfunc() const {}
};
class Ellipse: public Circle {
public:
    virtual void myEllipse() const
```

```
{
};
};
void main()
{
//typeid
int ti_obj=20;
string tstr="I am male.";
double td=32.56;
float tfl=21.075;
const type_info& ti = typeid(ti_obj);
const type_info& tif = typeid(tstr);
const type_info& tid = typeid(td);
const type_info& tifl = typeid(tfl);
cout << ti.name() << " "; << ti.raw_name() << endl;
cout << tif.name() << endl;
cout << tid.name() << endl;
cout << tifl.name() << endl;
//dynamic_cast bad_cast
Circle instance;
Circle& ref_shape = instance;
try {
    Ellipse& ref_circle = dynamic_cast<Ellipse&>(ref_shape);
}
catch (bad_cast) {
    cout << "Caught: bad_cast exception. A Shape is a Circle. \n";
}
//bad_typeid
Circle* p=NULL;
try{
    cout << typeid(* p).name() << endl;
}
catch(bad_typeid)
{
    cout << "Object is NULL. " << endl;
}
cout << "//... " << endl;
Circle* pp = new Circle();
try{
    cout << typeid(* pp).name() << endl;
}
catch(bad_typeid)
{
    cout << "Object is NULL. " << endl;
}
}
```

例 10-4 的执行结果为:

```
int ;: H
class std::basic_string< char, struct std::char_traits< char >, class std::allocator<
char >>
double
float
Caught: bad_cast exception. A Shape is a Circle.
Object is NULL.
//...
class Circle
```

10.6 异常处理

前面已经非常详细地讲述了异常处理的相关内容。本节将详细讲述和语言支持产生的异常处理相关的部分内容。在 C++ 程序中, 需要包含头文件 `<exception>`。在头文件 `<exception>` 中, 定义了几个类型和函数。这些类型和函数均是和异常处理相关的。头文件 `<exception>` 中所包含的内容如下:

```
namespace std{
    class exception;
    class bad_exception;
    typedef void (* unexpected_handler) ();
    unexpected_handler set_unexpected(unexpected_handler f) throw();
    void unexpected();
    typedef void (* terminate_handler) ();
    terminate_handler set_terminate(terminate_handler f) throw();
    void terminate();
    bool uncaught_exception() throw();
}
```

10.6.1 类 exception

类 `exception` 的声明形式如下:

```
namespace std {
    class exception{
    public:
        exception() throw();
        exception(const char * const &message);
        exception(const char * const &message, int);
        exception(const exception&) throw();
        exception& operator = (const exception&) throw();
        virtual ~exception() throw();
        virtual const char* what() const throw();
    }
}
```

类 `exception` 定义了异常对象类型的基类。在程序执行过程中，如果发生并汇报检测到的相关错误，这些异常对象会被作为异常抛出。其构造函数为：

构造函数为：

```
exception() throw();
```

构造 `exception()` 函数可以用于实现构造类 `exception` 的对象，但不能抛出任何异常。

```
exception ( const exception& ) throw();
exception& operator = (const exception& ) throw();
```

上述两个函数用于实现对类 `exception` 类型对象的备份。提示：尤其是赋值之后，调用 `what()` 的返回结果是预先定义的。

```
virtual ~exception() throw();
```

功能：破坏类 `exception` 的对象，释放内存。提示：不抛出异常。

```
virtual const char* what() const throw();
```

函数返回值是 NTBS。提示：返回的信息是以空（NULL）为结束标识符的多字节字符串。此函数也适用于宽字节字符的转换和显示。函数的返回值保持有效，直到捕获的异常对象被破坏或者异常对象的非常量成员函数被调用。

10.6.2 特殊异常处理

1. 类 `bad_exception`

类 `bad_exception` 的声明形式为：

```
namespace std{
    class bad_exception: public exception{
    public:
        bad_exception() throw();
        bad_exception(const bad_exception& ) throw();
        bad_exception& operator = (const bad_exception& ) throw();
        virtual ~bad_exception() throw();
        virtual const char* what() const throw();
    }
}
```

类 `bad_exception` 用于定义被抛出异常对象的类型。其构造函数为：

```
bad_exception() throw();
```

功能：构造类 `bad_exception` 的对象。提示：对于新构造的对象，调用 `what()` 函数时，结果是预先定义的。

```
bad_exception(const bad_exception& ) throw();
bad_exception& operator = (const bad_exception& ) throw();
```

上述两个函数用以实现类 `bad_exception` 的对象的备份，即构造并产生新的对象。

```
virtual const char* what() const throw();
```

函数返回值是已定义的 NTBS。返回的信息是以空（NULL）为结束标识符的多字节字

符串。此函数也适用于宽字节的转换和显示。

2. unexpected_handler

```
typedef void (* unexpected_handler) ();
```

当函数尝试抛出一个异常时（该异常并未在异常说明中列出），句柄函数会通过 `unexpected()` 函数调用。

一个非期望的句柄(`unexpected_handler`)不会被返回。默认的 `unexpected_handler` 调用会终止。

3. set_unexpected

```
unexpected_handler set_unexpected (unexpected_handler f) throw();
```

功能：创建使用指针 `f` 指定的函数，作为当前的 `unexpected_handler` 句柄。值得注意的是，`f` 不应该是空指针，函数返回值是之前的 `unexpected_handler` 型句柄。

4. unexpected

`unexpected` 函数的声明形式为：

```
void unexpected();
```

当函数存在时，是通过异常实现调用的；也可以在程序中直接调用。在 ISO/IEC 14882 C++ 标准中，此段英文原文应该是：“Called by the implementation when a function exits via an exception not allowed by its exception specification. May also be called directly by the program.”

评估 `throw` 表达式之后，即可快速、有效地调用 `unexpected_handler` 类型函数。通常此函数在执行过程中被调用，或者调用当前 `unexpected_handler`；也可以在程序中直接调用。

10.6.3 异常终止

1. terminate_handler

```
typedef void (* terminate_handler) ();
```

该句柄型函数的调用是通过调用 `terminate()` 函数实现的，并同时终止异常处理的进程。任意 `terminate_handler` 均可终止程序的执行，且并不返回函数的调用者。程序执行默认的 `terminate_handler` 时，会调用 `abort()` 函数。

2. set_terminate

```
terminate_handler set_terminate(terminate_handler f) throw();
```

功能：设置参数 `f` 标识的函数作为当前句柄函数，用于终止异常处理进程。值得一提的是，`f` 不能是空指针。函数返回值是原有旧的 `terminate_handler` 型句柄。

3. terminate

```
void terminate();
```

在执行函数过程中，异常句柄由于其他原因必须被抛弃时，此函数会被调用；也可以在程序中直接调用。评估 `throw` 表达式之后，即可有效、快速地调用 `terminate_handler` 类型函数。另外，可以使用程序直接调用 `terminate_handler` 类型函数。

10.6.4 未捕获异常(`uncaught_exception`)

```
bool uncaught_exception() throw();
```

或

```
bool uncaught_exception() noexcept;
```

`uncaught_exception` 函数的返回值为 `true` 的条件是：当某异常句柄被激活之后，当前线程会初始化一个异常对象，之后 `uncaught_exception()` 函数会返回 `true`。



提示 当 `uncaught_exception()` 函数的返回值为 `true` 时，抛出一个异常即可导致调用 `terminate()` 函数。

10.6.5 应用举例

下面使用例 10-5 阐述 10.6 节相关内容的使用方法。

例 10-5

```
#include <iostream>
#include <exception>
using namespace std;
void Tfunction()
{
    cout << "1. I'll be back." << endl;
}
void termfunction()
{
    cout << "2. I'll be back." << endl;
    exit(1);
}
class Test
{
public:
    Test(std::string msg) : m_msg(msg)
    {
        std::cout << "In Test::Test()" << std::endl;
    }
    ~Test()
    {
        std::cout << "In Test::~Test()" << " std::uncaught_exception() = " << std::uncaught_exception()
        << std::endl;
    }
private:
    std::string m_msg;
};
void main()
{
    exception ep("overflow.");
    cout << ep.what() << " 1." << endl;
    try{
        throw ep;
```

```
    }  
    catch(exception& e)  
    {  
        cout << e.what() << " 2. " << endl;  
    }  
    int id = cin.get();  
    if (id == 49)  
    {  
        unexpected_handler oldHand = set_unexpected( Tfunction );  
        unexpected();  
    }  
    else if (id == 50)  
    {  
        terminate_handler oldHand_t = set_terminate( termfunction );  
        throw bad_alloc();  
        //terminate();  
    }  
    else if (id == 51)  
    {  
        Test t1( "outside try block" );  
        try  
        {  
            Test t2( "inside try block" );  
            throw 1;  
        }  
        catch (...)  
        {  
        }  
    }  
}
```

例 10-5 的执行结果如下。

结果 1:

overflow. 1.

overflow. 2.

1

1. I'll be back.

abnormal program termination

Press any key to continue

结果 2:

overflow. 1.

overflow. 2.

2

2. I'll be back.

Press any key to continue

结果 3:

```
overflow.1.
overflow.2.
3
In Test::Test()
In Test::Test()
In Test::~Test()    std::uncaught_exception() = 0
In Test::~Test()    std::uncaught_exception() = 0
```

10.7 其他运行支持

10.7.1 概述

本节主要讲述头文件 `<stdarg>` (variable argument)、`<setjmp>` (nonlocal jumps)、`<ctime>` (system clock clock(), time()), `<csignal>` (signal handling) 和 `<cstdlib>` (runtime environment getenv(), system()) 等的相关知识。

各头文件的摘要详见表 10-2 ~ 表 10-6。表 10-2 给出的宏和函数可用于访问 list 列表。

表 10-2 头文件 `<stdarg>` 概要

类 型	名 称	类 型	名 称
宏	va_arg, va_end, va_start	类	va_list

表 10-3 中给出的函数和宏主要用于在程序中实现跳转功能。

表 10-3 头文件 `<setjmp>` 概要

类 型	名 称	类 型	名 称	类 型	名 称
宏	setjmp	类	jmp_buf	Function	longjmp

表 10-4 给出的宏和函数主要用于处理时间。

表 10-4 头文件 `<ctime>` 概要

类 型	名 称	类 型	名 称	类 型	名 称
宏	CLOCKS_PER_SEC	类	clock_t	Function	clock

表 10-5 给出的宏和函数主要用于信号处理。

表 10-5 头文件 `<csignal>` 概要

类 型	名 称
宏:	SIGABRT, SIGILL, SIGSEGV, SIG_DFL, SIG_IGN, SIGFPE, SIGINT, SIGTERM, SIG_ERR
类:	sig_atomic_t
Function:	raise, signal

表 10-6 给出的是两个环境处理函数。

表 10-6 头文件 <cstdlib> 概要

类 型	名 称
Functions:	getenv , system

上述头文件和标准 C 库的头文件 <stdarg.h>、<setjmp.h>、<time.h>、<signal.h> 和 <stdlib.h> 基本上是一致的，仅包括以下几点差异：

1) va_start、va_end 和 va_arg 这 3 个函数主要用于访问列表 (list) 的变量参数。va_arg() 函数用于返回当前的参数。va_list 函数用于创建一个参数类型 (type) 指针 (ap)；va_start() 用于初始化指针 ap，第二个变量必须是“...”之前的变量名；va_arg(last, ap) 函数第一次被调用时，会返回 last 之后的变长参数表中的第一个参数，并修改 ap 指向下一个元素，当再一次调用 va_arg() 时，函数会返回一个参数，可以将 ap 向前移动一步。va_arg 除了需要 ap 参数指针之外，还需要数据类型 type，用于决定下一步的步长应该迈多大。va_end() 函数是和 va_start() 函数相对应的。

ISO C 为宏 va_start() 函数设置第二个参数，该宏函数在头文件 <stdarg.h> 中声明。严格地说，这和 C++ 国际标准是不同的。假定在函数定义的变量参数表中，参数 parmN 是最右边参数标识。如果参数 parmN 被声明，在传递参数过程时，如果所传递的参数类型不合适，那么导致的行为将是不确定的。

2) setjmp() 和 longjmp() 函数分别担任局部标号和 goto 的功能。在 C++ 国际标准中，函数 longjmp(jmp_buf, int val) 具有更受限制的行为。longjmp() 函数的作用是使进程返回至 setjmp() 处执行，其第二个参数表示下一次调用 setjmp() 时的返回值。有时通过抛出异常转移控制到程序中的另一个目标点，如果任何自动类型的对象会被破坏，然后调用 longjmp(jbuf, val) 函数在抛出点，在该抛出点会转移控制到同一个点具有不确定的行为。

3) clock() 函数在前面的章节已经有所介绍，此处仅在例题中举例说明。

4) 信号处理 raise() 和 signal() 函数可以在 Windows 环境中是使用。raise() 函数用于抛出一个信号，signal() 函数是用于设置中断信号的处理函数 (句柄函数)。

5) getenv() 函数用于获取当前环境的值；system() 函数用于执行系统命令。

10.7.2 应用举例

下面使用例 10-6 阐述 10.7 节内容的使用方法。

例 10-6

```
#include <iostream>
#include <cstdlib>
#include <csetjmp>
#include <ctime>
#include <csignal>
#include <cstdliblib>
#include <string>
```

```
using namespace std;
//va_list char*
const int N=5;
void va_test(int i1,...)
{
    va_list ptr;
    va_start(ptr,i1);
    cout << i1 << " ";
    int n_i = N;
    //逐一取出元素
    while(n_i --)
    {
        cout << va_arg(ptr,int) << " ";           //依次取出一个整数
    }
    cout << endl;
    va_end(ptr);
}
void jmp_test()
{
    int num = -1;
    jmp_buf env;
    int state = setjmp(env); //只有第一次调用 setjmp 时,才返回
    if(state == 0)
    {
        cout << "输入数字:";
        cin >> num;
        if(num != 0)
        {
            longjmp(env,1);                       //下一次调用 setjmp 时,setjmp 的返回值为
        }
    }
    else
    {
        cout << "Error. " << endl;
    }
}
void mysleep(int sec_wait)
{
    clock_t goal;
    long int wait = sec_wait * CLOCKS_PER_SEC;
    goal = wait + clock();
    while(goal > clock())
    {
    };
    cout << "Testing wait " << sec_wait << " second. " << endl;
}
```

```
void time_test ()
{
    clock_t ct;
    ct = clock();
}
void S_Handler(int signal)
{
    printf("Signal dealing... \n");
}
void test_Signal ()
{
    typedef void (* SignalHandlerPointer) (int);
    SignalHandlerPointer previousHandler;
    previousHandler = signal(SIGABRT, S_Handler);
    raise(SIGABRT);
}
void test_getenv ()
{
    cout << "Lib information: " << endl;
    char * libvar;
    libvar = getenv("LIB"); // C4996
    if( libvar != NULL )
        //printf( "Original LIB variable is: % s\n", libvar );
    cout << libvar << endl;
    char* oldlib;
    _putenv("LIB=c:\\d:\\");
    oldlib = getenv("LIB");
    cout << oldlib << endl;
    string s1 = "LIB = ";
    string s2 = string(libvar);
    string s3 = s1 + s2;
    ///s1 = s1 + s2;
    _putenv(s3.data());
    oldlib = getenv("LIB");
    cout << oldlib << endl;
}
void test_system()
{
    cout << endl;
    cout << "system dir e: \\ " << endl;
    system("dir e: \\");
}
void main ()
{
    va_test(12,34,56,31,76,98);
    jmp_test();
}
```

```
mysleep(5);
try{
test_signal();
}
catch(...)
{
    cout << "signal test. " << endl;
}
test_getenv();
test_system();
}
```

例 10-6 的执行结果为:

```
12 ;34 ;56 ;31 ;76 ;98 ;
```

```
输入数字: 1
```

```
Error.
```

```
Testing wait 5 second.
```

```
Signal dealing..
```

```
Lib information:
```

```
C:\Program Files\Microsoft Visual Studio\VC98\mfc\lib;C:\Program Files\Microsoft  
Visual Studio\VC98\lib
```

```
c:\d:\
```

```
C:\Program Files\Microsoft Visual Studio\VC98\mfc\lib;C:\Program Files\Microsoft  
Visual Studio\VC98\lib
```

10.8 小结

本章主要讲述和语言支持相关的内容, 主要包括类型、执行属性、程序的启动和终止、动态内存管理、类型标识符、异常处理和其他运行库支持, 且针对每部分内容均配以例题进行举例说明。



提示 虽然本章涉及的内容大多并不常用, 但读者还是应该认真阅读本章的例题。只有结合基本内容, 对每个例题做到彻底理解, 才能掌握本章内容。

第 11 章

检测类模板详解

本章主要讲述C++ 程序中非常重要的内容——检测和报告错误。本章将详细介绍如何报告几种异常条件、文档程序的断言以及全局变量用于错误编码等内容。涉及的头文件主要有 `<stdexcept>`、`<cassert>` 和 `<cerrno>`。

11.1 异常类

前面已经多次提到了异常类（Exception）。本节将详细讲述异常类中和检测相关的内容。C++ 标准程序库提供了诸多用于在 C++ 程序中报告某些确定的错误的类。这些错误的模型反映（折射）了检测错误类。“错误”被分成两个大类：逻辑错误和运行错误。逻辑错误是由程序的内部逻辑错误所导致的。理论上讲，这些错误是可以预防的；而运行错误则是由一些超出程序范围的事件所导致的，这些错误是难以提前预知的。头文件 `<stdexcept>` 中定义了几种类型的预定义的异常类，以用于报告C++ 程序中的错误。这些异常通常和类的继承相关。前面已经讲过，头文件 `<stdexcept>` 的概要如下：

```
namespace std{
    class logic_error;
    class domain_error;
    class invalid_argument;
    class length_error;
    class out_of_range;
    class runtime_error;
    class range_error;
    class overflow_error;
    class underflow_error;
}
```

11.1.1 类 `logic_error`

类 `logic_error` 的声明形式为：

```
class logic_error : public exception {
public:
    explicit logic_error(const string& message);
    virtual const char * what();
};
```

类 `logic_error` 定义了异常对象的类型，在程序执行过程中，这种异常对象被抛出，可用

于报告程序执行过程中发生的错误。该类仅包括了一个构造函数和一个 `what()` 函数。构造函数的参数是该对象成员 `what()` 函数的返回值。其关系为：`what()` 的返回值应该等于 `message.data()`，或者等于 `message.c_str()`。

例 11-1

```
#include <iostream>
#include <stdexcept>
#include <string>
using namespace std;
void main()
{
    try
    {
        throw logic_error( "logic error: 2012-9-11. " );
    }
    catch ( exception & ep )
    {
        cerr << "Caught: " << ep.what() << endl;
        cerr << "Type name: " << typeid( ep ).name() << endl;
    };
}
```

例 11-1 的执行结果为：

```
Caught: logic error: 2012-9-11.
```

```
Type name: class std::logic_error
```

11.1.2 类 `domain_error`

类 `domain_error` 的声明形式为：

```
namespace std{
class domain_error: public logic_error{
    public:
        explicit domain_error(const string& what_arg );
}
}
```

类 `domain_error` 是类 `logic_error` 的派生类，同时也继承了类 `logic_error` 的 `what()` 函数。类 `domain_error` 的作用是定义了异常对象的类型，在程序执行过程中，这种异常被抛出，可用于在程序执行时报告“域”错误。其构造函数为：

```
domain_error(const string& what_arg);
```

该构造函数的作用是构造一个类 `domain_error` 的对象。构造函数的参数 `what_arg` 的内容和 `what()` 函数的返回值是一致的。

例 11-2

```
#include <iostream>
// #include <stdexcept>
using namespace std;
```

```
void main()
{
    try
    {
        throw domain_error( "Your domain is in error!" );
    }
    catch (domain_error & e)
    {
        cerr << "Caught: " << e.what() << endl;
        cerr << "Type name: " << typeid(e).name() << endl;
    };
};
}
```

在 Visual C++ 6.0 环境中执行时，上述程序在执行过程中被异常终止，并弹出异常对话框。在 Visual C++ 2008 环境中执行时，上述程序能够正常运行。其执行结果为：

```
Caught: Your domain is in error!
Type name: class std::domain_error
```

11.1.3 类 `invalid_argument`

类 `invalid_argument` 也是类 `logic_error` 的派生类。其声明形式为：

```
namespace std{
    class invalid_argument:public logic_error{
    public:
        explicit invalid_argument(const string& what_arg);
    };
}
```

类 `invalid_argument` 的作用是定义异常对象的一种类型。在程序执行过程中，这种异常对象被抛出，以用于汇报一个无效参数错误。其构造函数为：

```
invalid_argument(const string& what_arg);
```

该构造函数的作用是构造一个类 `invalid_argument` 的对象。构造函数的参数 `what_arg` 的内容是该对象成员函数 `what()` 的返回值。

例 11-3

```
# include <bitset >
# include <iostream >

using namespace std;

int main()
{
    try
    {
        bitset< 32 > bitset( string( "11001010101100001b100101010110000" ) );
    }
}
```

```

catch ( exception & e )
{
    cerr << "Caught " << e.what() << endl;
    cerr << "Type name: " << typeid(e).name() << endl;
};
}

```

例 11-3 的执行结果为:

```

Caught invalid bitset <N> char
Type name: class std::invalid_argument

```

上述例题之所以出现异常, 主要是因为 bitset 结构通常使用 long int 作为 bit array, 一般不会超越 16 位。而在程序中定义 bitset 类对象时超过了 16 位。访问位段时, 一旦超越了这个范围, 程序就会出错。第 3 章讲述位段时所给例题的位段长均为 16 位。这是读者需要注意的一个细节。

11.1.4 类 length_error

类 length_error 的声明形式为:

```

class length_error : public logic_error {
public:
    explicit length_error(const string& message);
};

```

类 length_error 定义了异常对象的类型。在程序执行过程中, 这种异常对象被抛出, 以用于报告长度错误。所谓长度错误是指对象的长度超越了其最大允许的大小。其构造函数为:

```
length_error(const string& what_arg)
```

该构造函数的作用是构造一个类 length_error 的对象。构造函数的参数 what_arg 的内容是该对象成员函数 what() 的返回值, 即 strcmp(what(), what_arg.c_str()) == 0。

例 11-4

```

#include <iostream>
#include <vector>
using namespace std;
void print(vector<int> v)
{
    vector<int>::iterator it;
    for(it=v.begin();it!=v.end();it++)
        cout<<" "<<*it<<" ";
    cout<<endl;
}
void test_one()
{
    int a[]={1,2};
    vector<int> vi;
    vi.assign(a,a+2);
    try{

```

```
vi.resize(vi.max_size() + 1);
}
catch(length_error& le)
{
cout << le.what() << endl;
cout << typeid(le).name() << endl;
}
}
void test_two()
{
try{
length_error le("my custom error. ");
throw le;
}
catch(length_error& le)
{
cout << le.what() << endl;
cout << typeid(le).name() << endl;
}
}
void main()
{
int ind=0;
for(;;)
{
cout << "Please Input number (1-3, 3:exit):" << endl;
cin >> ind;
switch (ind)
{
case 1:
test_one();
break;
case 2:
test_two();
break;
case 3:
exit(0);
}
}
}
```

例 11-4 的执行结果为:

```
Please Input number (1-3, 3:exit):
1
vector<T> too long
```

```

class std::length_error
Please Input number (1-3, 3:exit):
2
my custom error.
class std::length_error
Please Input number (1-3, 3:exit):
3
please press any key to continue...

```

11.1.5 类 out_of_range

类 out_of_range 的声明形式为:

```

class out_of_range : public logic_error {
public:
    explicit out_of_range(const string& message);
};

```

类 out_of_range 定义了该种异常对象的类型。同样,在程序执行过程中,这种异常对象被抛出,以用于报告参数值的错误(该参数值不在期望的范围内)。其构造函数为:

```

out_of_range(const string& what_arg);

```

该构造函数的作用是构造一个类 out_of_range 的对象。构造函数的参数 what_arg 的内容是该对象成员函数 what() 的返回值。

例 11-5

```

#include <iostream>
#include <vector>
#include <stdexcept>
using namespace std;
void print(vector<int>& v)
{
    vector<int>::iterator it;
    for(it=v.begin();it!=v.end();it++)
        cout<<*it<<" ";
    cout<<endl;
}
void main()
{
    vector<int> vi;
    int array[]={1,2,3,4,5};
    vi.assign(array,array+5);           //初始化 vector 向量
    print(vi);                          //输出向量
    try{
        int var=vi.at(-1);              //获取某个元素
        int var2=vi.at(0);
    }
}

```

```
catch(out_of_range& oe)           //捕获异常
{
    cout << oe.what() << endl;      //输出异常的信息
    cout << typeid(oe).name() << endl; //获取异常类的名称
}
}
```

例 11-5 的执行结果为:

```
1;2;3;4;5;
invalid vector<T> subscript
class std::out_of_range
```

11.1.6 类 runtime_error

类 runtime_error 的声明形式为:

```
class runtime_error : public exception {
public:
    explicit runtime_error(const string& message);
    virtual const char * what();
};
```

类 runtime_error 定义了该种异常对象的类型。这种异常对象在程序执行过程中被抛出,以用于报告检测到的错误。其构造函数是:

```
runtime_error(const string& what_arg);
```

该构造函数的功能是构造一个类 runtime_error 的对象。构造函数的参数 what_arg 和该对象成员函数 what() 的返回值是一致的。通常发生类 runtime_error 的类型错误时,计算机会发生较明显的变化,例明显变慢。当信息栏被关闭之后,程序一般会自动关闭或失去响应,甚至导致计算机重启。类 runtime_error 的类型错误通常是由内存导致的;也有可能是由病毒或其他恶意软件导致的。

常见的运行错误代码:

- 1). Illegal function call Program error.
- 2). Overflow Program error.
- 3). Out of memory.
- 4). Subscript out of range Program error.
- 5). Duplicate definition Program error.
- 6). Division by zero Problem with a math formula in the program or the programs code.
- 7). Type Mismatch.
- 8) Out of string space Program error.
- 9). No Resume Program error.
- 10). Resume without error Program error.
- 11). Out of stack space
- 12). Sub or Function not defined Program error.
- 13). Error in loading DLL.
- 14) Bad file name or number Program error.

```
15). File not found File required by the program to run is not found.
16). Bad file mode Program error.
17). File already open Program or file associated with program is being used and program does
not have
access to use it
18). File already exists Program error.
19). Disk full The disk, for example.
20). Input past end of file Program error.
...
```

类 `runtime_error` 的使用详见例 11-6。

例 11-6

```
# include <iostream>
# include <string>
# include <stdexcept>
# include <cstdlib>
using namespace std;
int main()
{
    try{
        float in;
        runtime_error re("Input float error.");
        cin >> in;
        if(! cin)
            throw re;
    }
    catch (runtime_error& re)
    {
        cout << "Error type:" << re.what() << endl;
        cout << "Class type:" << typeid(re).name() << endl;
    }
}
```

例 11-6 的执行结果为：

```
1
Error type:Input float error.
Class type:class std::runtime_error
```

11.1.7 类 `range_error`

类 `range_error` 的声明形式为：

```
namespace std{
    class range_error: public runtime_error{
    public:
        explicit range_error(const string& what_arg);
    };
}
```

类 `range_error` 定义了异常对象的类型。在程序执行时，尤其是在内部计算过程中，这种异常对象会被抛出，可用于报告“范围”类型的错误。其构造函数为：

```
range_error(const string& what_arg);
```

该构造函数的功能是构造一个类 `range_error` 的对象。构造函数的参数 `what_arg` 是该对象成员函数 `what()` 的返回值。

例 11-7

```
#include <iostream>
#include <stdexcept>
using namespace std;
void main()
{
    range_error re("Error type: range_error.");
    try
    {
        locale loc("test");
    }
    catch(...)
    {
        cout << re.what() << endl;
        cout << typeid(re).name() << endl;
    }
}
```

例 11-7 的执行结果为：

```
Error type: range_error.
class std::range_error
```

11.1.8 类 `overflow_error`

类 `overflow_error` 的声明形式为：

```
namespace std{
class overflow_error: public runtime_error{
public:
    explicit overflow_error(const string& what_arg);
};
}
```

类 `overflow_error` 定义了异常对象的类型。这种异常对象会在发生算术运算溢出错误时被抛出。其构造函数为：

```
overflow_error(const string& what_arg);
```

构造函数的参数 `what_arg` 和该对象成员函数 `what()` 的返回值是一致的。

例 11-8 (摘自 MSDN 2008, 该示例在 Visual C++ 2008 中调试通过)

```
#include <iostream>
#include <stdexcept>
#include <bitset>
```



```
void main()
{
    try{
        underflow_error ue("under_flow_error.");
        throw ue;
        //int vb = vi.at(-1);
        //cout << va << endl;
    }
    catch(exception& e)
    {
        cout << e.what() << endl;
        cout << typeid(e).name() << endl;
    }
}
```

例 11-9 的执行结果为:

```
under_flow_error.
class std::underflow_error
```

11.2 断言

断言宏常用于 C++ 程序中。宏 `assert` 主要用于调试程序的过程中。在程序运行时, 该宏会计算括号内的表达式, 若表达式为 `false`, 程序将报告错误, 并终止执行; 若表达式不是 `false`, 则继续执行该语句后面的语句。宏 `assertion` 主要用于判断程序中是否出现明显非法数据, 若出现了非法数据, 即迅速终止程序, 以免导致严重后果, 这样还便于查找错误。使用断言 `assert()` 函数时, 程序必须包含头文件 `<assert.h>`。`assert()` 函数用于评估其参数的值 (非零或零), 当参数 `expression` 为 `false` 时, 程序会打印出检测到的信息, 并终止 (`abort`) 程序的运行。例如,

```
#include <assert.h>
void assert(int expression);
```

使用 `assert()` 的缺点是会影响程序的性能, 增加额外的开销。`assert()` 函数并不受宏 `DEBUG` 的限制。无论是程序的 `debug` 版本还是 `release` 版本, `assert()` 函数均起作用, `assert()` 函数能够及时查出编码的错误。很多错误的发生不都是在编译时发生的, 而是运行时发生的, 因此不能仅仅依赖 `assert()`, 更主要的是使用 `exception` 机制来检测运行时的错误, 并采取相应的处理措施。

使用 `assert()` 函数时, 程序员需要注意以下几个问题:

- 1) 使用断言捕捉不应发生的非法情况。不要混淆非法情况与错误情况之间的区别, 后者是必然存在的, 并且一定要采取处理措施。
- 2) 使用断言时, 程序员需要对函数的参数进行确认。
- 3) 编写函数时, 程序员要反复考查, 对设定的假定需要使用断言进行检查。
- 4) 通常程序员都熟悉部分防错性的程序设计, 注意: 此风格会隐瞒错误。当进行防错

性编程时, 若不可能发生的事情发生了, 需要使用断言进行报警。

- 5) 每个断言需要详细、清楚。
- 6) 使用不同的算法对程序结果进行确认。
- 7) 程序员应在错误发生之前使用初始检查程序。

在 Visual Studio MSDN 中指出, 宏 `assert()` 函数典型的应用是识别逻辑错误。在程序开发过程中, 当程序运行不正确时, 参数 `expression` 会转变为错误 (`false`)。当程序调试结束时, 并不需要调整源文件, 仅使用宏 `NDEBUG`, 断言检测可以无效。`NDEBUG` 可以使用命令行选项 `/D` 定义或使用语句 `#define`。如果 `NDEBUG` 使用 `#define` 形式定义之后, `#define` 语句必须出现在头文件 `<assert.h>` 之前。当 `assert()` 函数的参数 `expression` 为 `false` 时, `assert()` 函数会打印出检测信息并调用 `abort()` 函数来终止程序的执行。若函数的参数 `expression` 为 `true`, 则不会采取任何措施。检测信息包括错误的表达式、源文件的名称、`assert()` 函数所在语句的行号等。

在 Visual C++ 2005 版本中, 检测信息是使用宽字符打印的。`assert()` 函数可以按期望的情况工作, 即使表达式是 `unicode characters` 的形式。

检测信息的目的仅仅在于检测应用程序的类型, 该应用会调用进程。控制台程序总是接收来自于 `stderr` 的信息。在基于窗口的应用程序中, `assert()` 函数会调用窗口 `MessageBox()`, 其目的是创建一个信息框来显示伴随【OK】按钮的信息对话框。当用户单击【OK】按钮时, 程序迅速调用 `abort()` 函数, 退出运行。

当应用程序链接 `debug` 版本的运行库时, 宏断言 `assert()` 函数会使用 3 个按钮创建一个消息框: `Abort`、`Retry` 和 `Ignore`。如果用户单击【Abort】按钮, 程序会迅速退出。如果用户单击【Retry】按钮, 调试器会被调用, 并且如果 `just-in-time` 调试有效, 用户可以调试程序。如果用户单击【Ignore】按钮, 断言 `assert()` 函数继续其正常执行——单击【OK】按钮创建信息框。提示: 单击【Ignore】按钮时, 如果错误条件存在, 会导致不确定的行为。

未获取调试过程中的更多信息参见 C 运行库 CRT 调试技术的文档。下面举例说明 `assert()` 函数的使用方法。

例 11-10

```
#include <iostream>
#include <assert.h>
#include <string>
using namespace std;
void ana_string(char* str)
{
    assert(str != NULL);
    assert(str != "\0");
    assert(strlen(str) > 2);
}
void main()
{
    char test1[] = "abc";
    char * test2 = NULL;
    char test3[] = "";
```

```
printf("string: % s \n",test1);  
fflush(stdout);  
ana_string(test1);  
printf("string: % s \n",test2);  
fflush(stdout);  
ana_string(test2);  
printf("string: % s \n",test3);  
fflush(stdout);  
ana_string(test3);  
}
```

例 11-10 的执行结果如图 11-1 所示。

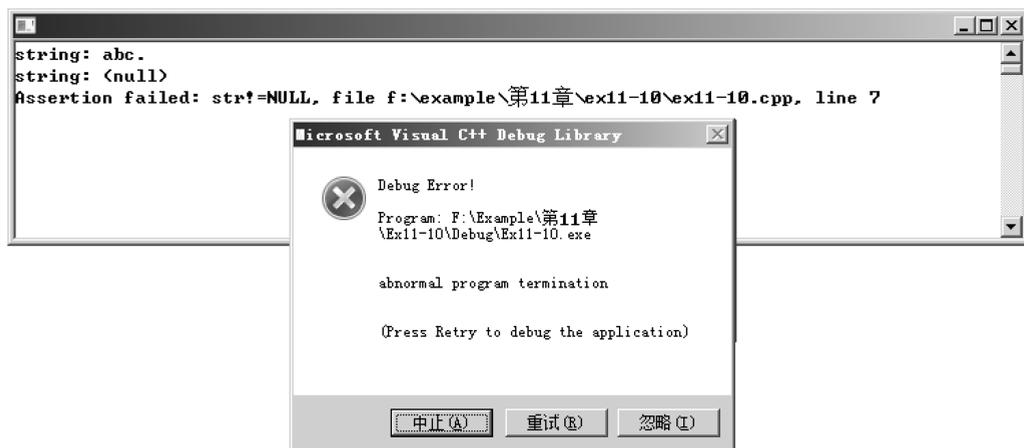


图 11-1 例 11-10 的执行效果

11.3 错误编码

头文件 `<cerrno>` 包含了诸多错误编码。下面逐一介绍 C++ STL 中关于错误编码的内容。变量 `errno` 的值是常量。在不同的错误条件下或事件中，诸多错误代码被赋值给变量 `errno`。头文件 `<errno.h>` 包含了诸多变量 `errno` 值的定义。然而，在 32 位 Windows 操作系统中，并不是所有定义都是在头文件 `<errno.h>` 中给出的。头文件 `<errno.h>` 中的值同时也保持了同 UNIX 家族操作系统的兼容性。

在 32 位 Windows 操作系统中，变量 `errno` 的值是 XENIX 系统中 `errno` 值的子集。因此，`errno` 的值并不是必须和实际的错误代码保持一致。错误代码是通过系统调用从 Windows 操作系统获取的。为实现访问实际的操作系统错误代码，可使用 `doserrno` 变量。该变量中包含了这些值。以下错误代码值是通常被支持的。

- ECHILD——没有大量的处理过程。
- EAGAIN——没有进一步的处理。尝试创建一个新的过程，如果尝试失败主要是因为没有进一步的进程缝隙，或没有足够的内存，或达到了最大嵌套级别。
- E2BIG ——参数列表太长。

- EACCES——不被认可。文件设置不被允许特殊的访问。错误意味着任何方式访问文件的尝试是不和文件的属性矛盾的。
- EBADF——坏的文件代码。两个可能的原因：①特定的文件描述符是无效值；②没有指向已打开的文件。
- EDEADLOCK——资源的死锁会发生。对于一个数学函数的参数，此错误代码是指没有在合适的范围内。
- EDOM——数学参数。
- EEXIST——文件存在。当尝试创建一个文件时，发现该文件已经存在。例如，当实现 `open` 调用时，`_O_CREAT` 和 `_O_EXCL` 标识是特定的，但命名的文件已经存在。
- EILSEQ——字节的非法序列（例如在一个 MBCS 字符串中）。
- EINVAL——无效参数。函数的参数之一是无效值。例如，当定位文件时，最初的给定值处于文件头之前。
- EMFILE——太多被打开的文件。不能创建更多的文件描述符，也不能再打开更多的文件。
- ENOENT——没有该文件和路径。指定的文件或路径不存在或不能被发现。当指定的文件不存在，或文件路径不能说明一个已存在的路径时，该信息会发生。
- ENOEXEC——可执行文件格式错误。尝试执行一个文件时，该文件是不可执行的，或具有无效可执行文件格式。
- ENOMEM——没有足够的内存。当尝试操作符时，设备没有足够的内存。例如，当无足够的内存用于执行子进程时，此信息会发生；或要求调用 `getcwd` 却不能得到满足时。
- ENOSPC——设备没有剩余空间。设备没有足够的空间用于写操作。例如磁盘已满。
- ERANGE——结果值太大。数学函数的参数值太大，导致部分或全部失去意义。当函数的参数远大于其期望值时，此错误也可以发生在其他类型的函数中。
- EXDEV——交叉设备链接。当尝试移动文件到其他存储设备时会发生此类错误提示。
- STRUNCATE——字符串备份或其他相关事物导致一个截取的字符串。

11.4 小结

本章主要包括三部分内容：第一节介绍了各种异常类，并给出了相应的例题；第二节讲述了断言的相关内容；第三节讲述了基于 Windows 操作系统的错误代码。

第 12 章

国际化库详解

随着C++语言在全球的推广和普及，国际化日渐成为软件开发的重要议题。为适应国际化潮流，C++标准程序库提供了编写国际化程序代码支持机制。支持机制主要影响I/O的使用和字符串的处理。这是本章的重点内容。Dietmar Kuhl是C++标准程序库有关I/O和国际化方面的专家，国际化的内容大部分是由其负责的。

C++标准程序库面对“区域”的转换问题，除着眼于某些特定转换外，还提供了一个普通方案。例如，为支持16位亚洲字符，该普通方案不限定字符型别。对于程序国际化而言，以下两个方面非常重要：

- 1) 不同的字符集具有不同的性质。面对字符数多于256个的字符集，char型别显然是不够用的。

- 2) 使用者希望程序能够针对国家和文化传统的差异进行必要转化（例如日期、格式、货币书写格式、布尔值表示法等）。

对于上述两个问题，C++标准程序库均提供了相关解决方案。

其实，所谓国际化问题最根本的莫过于各种问题的本地化。顾名思义，区域表示通常是在全世界不同区域具有不同的信息表达方式，对于不同文化背景的计算机用户，需要使用不同的信息显示方式。例如各个国家的日期显示格式是不同的。即使在使用同一种国际语言的国家之间，信息的表达方式仍然存在较大差异。程序开发者如果试图在全球范围推广其软件产品，必须意识到不同区域的差异。

国际化问题的主要思路是以locale对象代表可扩充的facet集合，以此实现地区转换工作。Locale在C语言中已经有所应用，在C++标准中国际化问题被更加泛型化（通用化），其设计形式更有弹性。事实上，C++的locale机制可以根据用户环境或偏好进行各种专用自定义设制。例如，可以对其进行扩展，使其处理度量衡系统、时区、纸张规格等问题。

大部分国际化相关机制仅需程序员付出最低程度的努力，根本不需要任何额外付出。程序员使用locale对象直接进行格式化、校勘、字符分类等工作。但C++标准程序库所支持的某些方面并没为其自身所用，程序员必须手工调用相应函数才能运用之。C++标准程序库中没有任何stream函数可以用来格式化时间、日期或货币表达式。最突出的代表就是strings和streams使用了国际化机制的另一个概念：字符特性。二者均定义出能够把不同字符集区分开来的某些基本特性和操作。国际化机制的相关类是最近几年才引入C++标准中的。尽管整体方案极其灵活，但要真正达到完善还需要做一些工作。例如字符串校勘函数直接使用型别const char*的迭代器。另外，charT代表某种字符型别，尽管basic_string<charT>通常会使用此型别作为迭代器型别，但未进行任何的明文保证。

12.1 国际化元素

当软件产品推广至国际市场时，最重要的问题是字符串输出。软件产品通常包括以下 8 种信息的输出。

1) 字符。使用简单 ASCII 码表示一种语言。ASCII 码字符集仅能表示最多 256 个字符，但其他某些语言需要更大的字符集。其他语言的表示问题导致 3 种类型字符编码的产生：单字节 ASCII 字符、多字节字符和宽字符。

2) 字符排序。在字符进行排序时，此过程称为排序。不同语言应用不同的排序规则。例如，在西班牙文中，双符号字母“ll”被看作单个字符；而在英语中，双符号字母“ll”被看作两个“l”字符。因此，西班牙文的排序方法与英语是不能相同的。

3) 字符分类。每种语言均含有字母、数字、标点和其他类型符号的字符编码。字符所属的组就是该字符的分类。

4) 数字。不同国家表示数字的方法各不相同。例如，美国风格的数字 34785000.75，如果按德国风格应写为 34.785.000,75。

5) 货币。不同的国家使用不同类型的货币，货币符号各不相同。美国货币符号是美元符号“\$”，德国的货币符号是“DM”。另外，货币符号在货币值中的位置也不相同。美国货币符号放在数量值前，如 \$ 45.98；而德国货币符号放在数量值后，如 12.45 DM。

6) 时间和日期。不同国家显示时间和日期的方式不同。

7) 大小写。不同语言在文本大小写转换规则上有所不同。

8) 语言。如果英语、西班牙语、法语或其他任何一种语言是地球上唯一通用的语言，编写国际化的应用程序就会容易得多。在国际化程序中，文本应用一种适当的语言显示。通常需要把文本放在资源文件中，与执行文件分离，可以根据需要翻译文本。

C++ STL 提供了管理国际化元素的基本方法。但对于每种区域的本地语言，在大多数情况下，程序员必须手工翻译。

12.2 多种字符编码

国际化或本地化所关注的主要问题是如何处理不同的字符编码。尤其是亚洲地区，甚至同一字符集还存在不同的编码。例如，汉字存在多种编码方式（BIG5 码和 GB 码等），甚至存在单一字符编码超过 8 个位的情况。面对上述这些问题，需要使用新的方法和新的函数进行文本处理。

12.2.1 宽字符和多字节文本

面对多于 256 个字符的字符集，有两种不同的解决方案；多字节表示法和宽字符表示法。

1) 多字节表示法。字符所用的字节（Byte）个数是可变的。单个字节字符后面可以跟随 3Byte，例日本的表意文字或中国的象形文字。

2) 宽字符表示法。字符所用的字节数目是恒定的，与所表示的字符无关。典型的个数是 2 或 4Byte。其实，这和那些只使用 1Byte 的表示法没有区别。

多字节表示法比宽字节表示法更紧凑，前者通常用于在程序外部储存数据，后者由于比较容易处理固定大小的字符，通常程序内部被使用。

与 ISO C 相似，ISO C++ 使用 `wchar_t` 表示宽字符。并且，在 C++ 中，`wchar_t` 被确定为关键词，而不仅仅是一个型别定义，而且可以使用该型别对所有函数进行重载。

在多字节字符串中，当某单个字节代表一个字符时，它也有可能是字符的一部分。例如对于 GB 汉字的表示，1 个字符仅表示半个汉字字符。尤其在遍历多字节字符串时，每个字节的意义均由当时的“转换”状态决定。根据转换状态，一个字节仅表示一个字符。通常多字节字符串会具有某些预定的初始转换状态。例如，在初始转换状态下，可能每个字节均代表 ISO 拉丁字符，直到遭遇“Escape”字符为止。紧跟在“Escape”字符之后的字符会用于指示新的转换状态，这意味着随后的字节会被理解为阿拉伯字符，直到遇到下一个“Escape”字符为止。

类模板 `codecvt <>` 用于在多种字符编码方案之间转换。该类主要为模板类 `basic_filebuf <>` 提供服务。在数据内部表述和外部表述时，实现转换。C++ 标准没有对多字节字符编码做任何的假定，仅仅支持“转换状态”的概念。类模板 `codecvt <>` 的成员可以运用一个参数来储存字符串的任意状态。类模板 `codecvt <>` 还支持把某个字符序列将转换状态还原为初始状态。

12.2.2 字符特性

对于多种字符集，多种表述方式给 `string` 和 I/O 的处理带来了多样性。字符串 `string` 和类 `stream` 可以根据内部型别完成对象的初始化。内部型别的接口不能够改变，需要引进一个独立的类别，即特性类别 (Trait Class)。“如何处理和表述相关的各个侧面”的相关细节被统统放入该类别中，这是特性类别的概念表述。`string` 和 `stream classes` 都将 `traits class` 作为 `template` 参数；此参数默认为 `char_traits`，并以 `string` 或 `stream` 获取模板的参数作为参数。

```
namespace std{
    template <class charT , class traits = char_traits <charT > ,
    class Allocator = allocator <charT >>
    class basic_string;
}
namespace std{
    template <class charT, class traits = char_traits <charT >>
        class basic_istream;
    template <class charT, class traits = char_traits <charT >>
        class basic_ostream;
    ...
}
```

字符特性 (Character Traits) 的型别为 `char_traits <>`。该型别在 `<string >` 中定义，以字符型别为参数：

```
namespace std{
    template <class charT >
    struct char_traits{
        ...
    }
}
```

特性类定义了字符型别的所有基本特性，并以静态组件的形式定义出类 `strings` 和类

stream 中不可或缺的相关操作。

字符串和字符序列的处理函数, 其存在价值完全是为了效率优化, 其实完全可以运用“只处理单个字符”的函数实例出来。例如, copy() 可运用 assign() 实例出来, 而在处理字符串时, copy() 可能会采用更有效率的实例化技术。



注意 函数中使用的计数是确切的计数, 不是最大计数, 即所用序列的终止字符将被忽略。

最后一组函数专门对 end-of-file 字符进行特殊处理。该字符通过人工字符扩充字符集, 用以指示某种特殊的处理。对某些表述而言, 字符型别不足以容纳该特殊字符, 因为 EOF 字符必须使用字符集中一般字符之外的值。C 语言为此建立了一种转换模式: 令字符函数返回一个 int, 而不是一个 char。该技巧被 C++ 语言继承并发扬。字符特性型别定义了 char_type 型别用于代表所有字符型别, 定义了 int_type 用以代表“所有字符, 外加 EOF”的型别。函数 to_char_type()、to_int_type()、not_eof()、eq_int_type() 定义了相应的转换和比较操作。对于某些特殊字符, char_type 和 int_type 可能是相同的。char_type 中并非所有数值都必须拿来表示字符, 因此一定会有多余的值拿来代表 EOF。

pos_type 和 off_type 用于定义文件位置和偏移距离。类模板 char_trait 的成员见表 12-1。

表 12-1 字符特性类 char_trait

类 成 员	意 义
char_type	字符型别 (即 char_traits 的模板 (template) 参数)
int_type	足以容纳“附加之 end-of-file 值”的型别
pos_type	此型别用以表现“stream 内的位置”
off_type	此型别用以表现“stream 内的两个位置之间的间距”
state_type	此型别用以表现“multibytes stream”的当前状态
assign (c1, c2)	将字符 c2 赋值给 c1
eq (c1, c2)	判断字符 c1 和 c2 是否相等
lt (c1, c2)	判断字符 c1 是否小于字符 c2
length (s)	返回字符串 s 的长度
compare (s1, s2, n)	比较字符串 s1 和 s2 的前 n 个字符
copy (s1, s2, n)	将字符串 s2 的前 n 个字符复制到 s1
move (s1, s2, n)	将字符串 s2 的前 n 个字符复制到 s1, s1 和 s2 可重叠
assign (s, n, c)	将字符 c 赋值给字符串 s 的前 n 个字符
find (s, n, c)	在字符串 s 中搜寻第一个与 c 相等的字符, 返回一个指针指向它。如果 s 的前 n 个字符中没有找到字符 c, 函数返回 0
eof()	返回 end-of-file 值
to_int_type (c)	将字符 c 转换成“型别为 int_type”的相应值
to_char_type (i)	将型别为 int_type 的 i 转换为字符 (如果转换 EOF, 会导致未定义行为)
not_eof (i)	如果 i 不是 EOF, 返回 i; 如果 i 是 EOF, 返回一个由实作版本所定义的、不同于 EOF 的值
eq_int_type (i1, i2)	检查字符 i1 和 i2 的 int_type 对应值是否相等 (字符可以是 EOF)

C++ 标准程序将 `char_traits < >` 型别针对 `char` 和 `wchar_t` 进行了特化设计:

```
namespace std{
    template < > struct char_traits < char >;
    template < > struct char_traits < wchar_t >;
}
```

针对 `char` 而设计的特化版本, 通常在实作过程中会运用定义于 `<cstring >` 或 `<string.h >` 内的 C 全局函数。

12.2.3 特殊字符国际化

处理特殊字符也是一个比较难的问题。例如, 换行符号或字符串终止符号的国际化, `class basic_ios` 的成员函数 `widen()` 和 `narrow()` 可用于解决这个问题。例如, 对于流类型 `stream`, 可以使用换行符编码如下:

```
stream strm;
strm.widen(' \n');
```

同样, 字符终止符号可以编码如下:

```
strm.widen(' \0');
```

`widen()` 和 `narrow()` 函数在应用时使用了一个 `locale` 对象, 准确地说是该对象的 `ctype` facet。该 facet 用于对所有字符“在 `char` 和其他表现形式之间”进行转换。`locale` 类型对象可以将 `char` 型别的字符 `c` 转换为一个 `char_type` 型别的对象。例如,

```
std::use_facet < std::ctype < char_type >> loc.widen(c);
```

`locales` 及其 `facets` 的使用细节之后会讲述。

12.3 类 locale

在 C++ 中, `std::locale` 类的对象代表一个区域的表示。`locale` 对象是一种容器, 容器中包含若干称为“刻画”的对象, 刻画代表前面讲述的国际化元素。类 `locale` 对象含有若干个处理国际化元素的刻画, 例如日期时间格式、货币符号和数字格式等。刻画不仅提供区域表示的国际化元素信息, 还为国际服务提供接口。

12.3.1 类 locale 概述

类 `locale` 实现一种类型安全的、多态的“刻画”集合, 可使用刻画类型检索。换言之, 一种刻画具有双重角色: 某种意义上, 刻画正好是类的接口, 同时也是 `locale` 刻画集合的索引。通过两个函数模板: `use_facet < >` 和 `has_facet < >` 访问 `locale` 类型刻画。

```
template < class chart , class traits > basic_ostream < chart , traits > &
    operator << ( basic_ostream < chart , traits > & s , Date d )
{
    typename basic_ostream < chart , traits > ::sentry cerberos ( s );
    if ( cerberos )
    {
```

```
ios_base::iostate err=0;
tm tmbuf;
d.extract(tmbuf);
use_facet<time_put<charT, ostreambuf_iterator<charT, traits>>>(
s.getloc()).put(s,s,s.fill(),err,&tmbuf,'x');
s.setstate(err);
}
return s;
}
```

在调用 `use_facet<Facet>(loc)` 时，类型参数选择一种刻面，要使所有名称类型的成员均有效。在 `locale` 中，若 `facet` (刻面) 不存在，该刻面会抛出标准异常 `bad_cast`。一个 C++ 程序可以检查出是否类 `locale` 使用函数模板 `has_facet<Facet>` 实现一个特殊的刻面。类 `locale` 对象中，用户定义的刻面可能被装备并使用，就像许多标准刻面一样。

类库中还提供了一种成员操作符模板 `operator()` (`basic_string<C, T, A>&`, `basic_string<C, T, A>&`)。对于标准集合和校核字符串，此类模板适用于使用类 `locale` 的对象作为一个可预测参数。常规的全局接口界面主要用于传统的 `ctype` 函数，类似于 `isdigit()` 和 `isspace()`。尤其对于给定的 `locale` 对象，一个 C++ 程序可以调用其成员函数 `isspace(c, loc)`。这使升级现存的抽出符变得很容易了。

一旦一种刻面 (或其引用) 从一个 `locale` 对象获取，即可通过调用 `use_facet<>`，才能使用该刻面的功能。既然一些 `locale` 对象涉及了刻面，其成员函数的结果会被期望，并可重复使用。

在连续调用一个 `locale` 类的 `facet` 成员函数的过程中，输入/输出流的插入器或抽出器或流缓冲区成员函数，其返回的结果应该是一致的。类 `locale` 对象在构建时，其名称字符串可能是 (“POSIX”)；或两个被命名的 `locale` 对象，拥有同一个名称，其他是不可能的。类 `locale` 的名称可能用于“相等”比较，无命名的 `locale` 仅能等于它本身 (即其备份)。对于一个无命名的 `locale` 对象，成员函数 `name()` 返回类型为字符串类型。

对于实现编程的国际化，仅仅翻译“文字所带信息”通常是不够的。数值、货币、日期……均具有不同的各种不同的规格，都是需要必须遵守的。之外，用于操作字母的函数，应根据字符进行编码，以确保正确处理特定语言中所有字母的字符。根据 POSIX 和 X/Open 标准，C 程序可使用函数设定一个 `locale` 的各种属性。改变 `locale` 会对 `isupper()` 和 `toupper()` 之类的字符分类/操作函数以及 `printf()` 之类的 I/O 函数产生影响。

然而 C 的解决方案毕竟包括了诸多限制。由于 `locale` 是全局属性，同时使用多个 `locale` 是比较麻烦的。`locale` 不能扩展，仅能提供“由编译器选择供应”的设施。若某个必须遵守的国家协议未被 C `locale` 支持，则只有改弦更张。程序员不可能为支持某种特殊文化而定义新的 `locales` 类。

C++ 标准程序库利用面向对象方式解决了上述问题。首先，`locale` 相关的细节被封装在型别为 `locale` 的对象中。这样在同一时刻运用多个 `locale` 即可解决诸多问题。与 `locale` 相关联的各种操作，会运用相应的 `locale` 对象。将每一个 `locale` 对象关联至到每个流中，流的各成员函数会利用该对象迎合相应规格。经典的 C `locale` 可以直接赋值到标准输入通道中。所谓经典的 C `locale`，即使用最初的 C 形式来进行格式化数字、日期、字符分类等工作。

`std::locale::classic()` 用于获取对应的经典 locale 对象。

表达式 `std::locale("C")` 和 `std::locale::classic()` 的效果相同。

该表达式根据给定的名字产生一个 locale 对象。“C”是特殊名称，实际上是各个 C++ 实作版本唯一必须支持的名称。C++ 标准并未强制要求支持其他 locale，通常各个 C++ 实作版本都会支持其他 locales。表达式 `cout.imbue(locale("de_DE"))` 可以实现将名称为 `de_DE` 的 locale 赋值到标准输出通道中。唯有当系统支持该 locale，该动作才会成功。若所希望构造的 locale 名字不能被 C++ 实作版本识别出来，将会抛出一个 `runtime_error` 异常。

若一切顺利，输入时将按照经典 C 规格，输出时会按照德国规格。这样就可以按英文格式读取浮点数（德文是使用逗号作为小数点的）。

通常，除非需要按照某个固定格式来读写数据，否则程序不会预先定义一个特定的类 locale。而利用环境变量 `LANG` 确定相应的 locale。另一种可能是读取一个 locale 名称时，需要运用它。

例 12-1

```
#include <iostream>
#include <locale>
#include <string>
using namespace std;
int main()
{
    locale loc1("german");
    locale loc2 = locale::global(loc1);
    cout << loc1.name() << ". " << endl;
    cout << loc2.name() << ". " << endl;
    locale loc3 = locale::global(std::locale(""));
    cout << loc3.name() << endl;
}
```

例 12-1 的执行结果为：

```
German_Germany.1252.
```

```
C.
```

```
German_Germany.1252
```

如果程序遵循各地的习惯，应该使用相应的 locale 对象。类 `locale` 的静态成员 `global()` 可以安装一个全局的 locale 对象。该对象可用来作为某函数的 locale 对象参数的默认参数。如果 `global()` 函数所设定的 locale 对象是有名称的，而 C 的 locale 相关函数也会做出相应的响应；若该对象没有名称，则 C 函数的执行结果由实作版本决定。

对于后继被执行的 C 函数会做出相应的注册操作，即那些 C 函数所受到的影响和以下调用操作所带来的影响相同。

```
std::setlocale(LC_ALL, "");
```

通过设定全局 locale，不能替换已储存于对象内的 locales。只能改变由默认构造函数所产生的 locale 对象。stream 对象存储的 locale 对象不会被 `locale::global()` 替换。若希望已经存在的 stream 使用某个特定的 locale，必须使用 `imbue()` 函数通知该流 (stream)。

当默认构造一个 locale 对象时, 全局 locale 就会发挥作用。产生新的 locale 即是全局 locale 的副本。下述代码是 3 个标准的 streams 安装默认的 locale 类对象的实例。

```
std::cin.imbue(std::locale());
std::cout.imbue(std::locale());
std::cerr.imbue(std::locale());
```

在C++ STL中使用 locale, 读者牢记C++ locale 机制和 C locale 机制之间仅有松散的耦合关系。若一个具名的C++ locale 对象被设为全局 locale, 则全局 C locale 会很被动。通常, 不能假设 C 和C++ 函数所操作的 locales 保持一致。

12.3.2 类 locale 的 facet

国内约定俗称的具体项目被划分为数个不同的刻面, 分别由相应的对象处理。处理“国际化议题中的某一特定刻面”的对象, 即称为 facet。locale 对象是容纳 facet 的容器。若存取 locale 的某个刻面, 可以相应的 facet 型别作为索引。如果 facet 作为模板参数, 使用模板的 use_facet() 函数, 可取使用特定的 facet。

```
std::use_facet<std::num_punct<char>>(loc);
```

上述代码用于获取 locale loc 所管辖的“字符型别为 char”的 facet num_punct。每个 facet 均由“定义特定服务”的类别定义。若 facet num_punct 为格式化数值和布尔值提供服务, 下述表达式将返回用以表达 true 的字符串。

```
std::use_facet<std::num_punct<char>>(loc).truename();
```

表 12-2 描述了C++ 标准程序库中预先定义的 facet。每个 facet 都和类别相关联。这些类别在 locale 的某些构造函数中用来组合其他 locale, 并产生新的 locale。

表 12-2 C++ 标准程序库预先定义的 facet 型别

分 类	facet 型别	用 于
numeric	num_get<>()	数值输入
	num_put<>()	数值输出
	num_punct<>()	数值 I/O 中用到的符号
Time	time_get<>()	时间和日期的输入
	time_put<>()	时间和日期输出
monetary	money_get<>()	货币输入
	money_put<>()	货币输出
	money_punct<>()	货币 I/O 中用到的符号
Ctype	ctype<>()	字符信息 (toupper(), isupper())
	codecvt<>()	在不同字符编码之间进行转换
Collate	collate<>()	字符串校勘 (collation)
messages	messages<>()	获取字符信息

1. 标准刻面

程序员可以创建刻面。C++ 定义了如下 7 种标准刻面。

- 1) 代码转换。此类刻面可以处理不同字符表示间的转换。例如从多字节字符转换成宽字符。
- 2) 排序。此类刻面可以处理字符排序，用于字符串排序。
- 3) 分类。这类刻面可以处理字符的分类。若设置字符的大小写，或判断某个字符是否属于语言的标点符号组。
- 4) 数字。此类刻面可以处理数字的格式化。
- 5) 货币。此类刻面可以处理货币的格式化。
- 6) 时间和日期。此类刻面可以处理时间和日期的格式化。
- 7) 消息。此类刻面可以处理多种消息类别，使应用程序能够翻译简单的响应，例如“是”和“否”。

2. 默认和全局表示

前文曾多次使用区域表示。每个C++ 程序均在默认的区域表示下运行，即美国的 ASCII 码表示。C++ 库创建了该区域的表示对象，即 `std::locale::classic`。`std::locale::classic` 会像其他区域表示一样，决定库函数的显示信息方式。

全局区域表示是当前有效的区域表示。若没有在程序中修改区域表示，则默认和全局区域表示是一样的。全局区域表示可以修改，但默认区域表示不能修改。

3. 细述类 locale

C++ locale 是不变的 facet 容器，定义于头文件 `<locale>` 内。

```
namespace std{
class locale{
public:
    static const locale& classic();
    static locale global(const locale& );
    class facet;
    class id;
    typedef int category;
    static const category none, numeric, time, monetary, ctype, collate, message, all;
    locale() throw();
    explicit locale(const char* name);
    locale(const locale& loc) throw();
    locale(const locale& loc, const char* name, category);
    template <class Facet > locale(const locale& loc, Facet* fp);
    locale(const locale& loc, const locale& loc2, category);
    const locale& operator = (const locale& loc) throw();
    template <class Facet > locale combine(const locale& loc );
    ~locale() throw();
    basic_string<char > name() const;
    bool operator == (const locale& loc) const;
    bool operator != (const locale& loc) const;
    template <class chart, class Traits, class Allocator > bool operator ()
    {
        const basic_string<charT, traits, Allocator >& s1;
```

```

const basic_string<charT, Traits, Allocator>& s2) const;
}

template <class Facet > const Facet& use_facet(const locale& );
template <class Facet > bool has_facet(const locale& ) throw();
}
}

```

对于类 locale，其存取方式是比较特殊的。locale 内含 facet，是以 facet 型别为索引进行存取的。由于每个 facet 均有不同的界面，且用于不同的目的，所以期望 locale 的存取函数返回对应索引的型别。以 facet 型别作为索引的好处是得以拥有型别安全的接口。

locale 是不可变的，即存储于 locale 之内的 facet 不能被改变。现有的 locale 和 facet 组合起来产生新的 locale 变体。locale 类的构造函数存在多种形式，详见表 12-3。

表 12-3 locale 类的构造

表 达 式	功 能
locale()	产生一个当前全局的 locale 副本
locale (name)	根据名称产生出一个 locale
locale (loc)	产生 locale loc 的副本
locale (loc1, loc2, cat)	产生 locale loc1 的副本，类别 cat 中所有 facet 将被 locale loc2 的 facets 替换
locale (loc, name, cat)	此动作等同于 locale (loc, locale (name), cat)
locale (loc, fp)	产生 locale loc 的副本，并安装 fp 所指的 facet
loc1 = loc2	将 loc2 赋值给 loc1
loc1. template combine <F> (loc2)	产生 locale loc1 的副本，并将 loc2 中型别为 F 的 facet 装入

几乎所有构造函数均产生某个副本。备份 locale 是很廉价的操作，基本上只是设定一个指针并将引用计数累加 1 而已。但产生一个变化过的 locale 的成本相当大，必须调整 locale 内每个 facet 的引用计数值。虽然 C++ 标准没有对此类操作的效率作出任何保证，但是所有实作版本大概都优先照顾 locale 的备份效率。

表 12-3 的两个构造函数需要 locales 名称。除了“C”这个名称，其他名称都没有标准规定。不过 C++ 标准要求每个 C++ 标准库附带的说明文件必须列出可接受的名称。基本上，可以假设大部分实作版本均会接收前面章节所列名称。

成员函数 combine() 服从最新的编译器特性：一个“带有显式指定之 template 参数”的成员函数模板。该模板 template 参数并非由另一个参数隐式推导而来。两个函数用于存取 locale 对象中的 facet，也采用相同的技术。不同之处在于：这两个函数是全局的 template 函数，不必加模板关键词 template。

use_facet() 函数会返回指向 facet 的一个引用 (reference)。其型别是显式传递的模板 template 参数型别。若传入的 locale 不包含相应的 facet，则函数抛出 bad_cast 异常。has_facet() 函数可用于检查某个 locale 中是否包含特定的 facet。详见表 12-4。

表 12-4 facet 的存取

表 达 式	功 能
has_facet <F> (loc)	若 locale loc 中有一个型别为 F 的 facet，则返回 true
Use_facet <F> (loc)	若返回一个 reference，指向 locale loc 中型别为 F 的 facet

类 locales 的其他操作列表见表 12-5。若某个类 locale 对象是从某个名称或若干个具名的 locale 构造而来的，名称将被保存起来。若参与构造的 locale 包含两个以上，则 C++ 标准并未构造出的名称给予保证。若一个 locale 是另一个 locale 的副本，或两个 locale 名字相同，则两个 locale 被认为是相同的。locale 的名称是被用来创建“具名 facet”的名称。运用该体制，两个 locale 若由具体的相同名称 facet 参与构造，生成的新名称也相同。C++ 标准本质上要求“由相同名称的 facet 组合而成”的 locale 均相同，才可能精心选出支持上述“相等性”思维。

表 12-5 locale 的操作函数

表 达 式	功 能
loc. name	返回 locale loc 的名称字符串
loc1 == loc2	若 loc1 和 loc2 是相同的 locale，返回 true
loc1 != loc2	若 loc1 和 loc2 是不同的 locale，返回 true
loc (str1, str2)	返回布尔值，显示字符串 str1 和 str2 的比较效果
locale::classic ()	返回 locale ("C")
locale::global (loc)	将 loc 安装为全局 locale，并返回上一个全局 locale

小括号操作符使程序员得以运用 locale 对象作为字符串比较工具。此操作符运用 collate facet，按顺序比较参数所传递的字符串，即在判断 locale 对象掌控时第一个字符串是否小于第二个字符串。这是仿函数的行为。采用 locale 对象作为 STL 算法所需的字符串排序规则。

```
std::vector<std::string> v;
std::sort(v.begin(), v.end(), locale("de_DE"));
```

12.3.3 区域表示和混合区域表示

由于默认区域表示不能修改，因此要根据需要创建区域的表示对象。std::locale 类提供了如下 4 种创建区域表示对象的方法：

```
std::locale localeName();
std::locale localeName(localeCode);
std::locale localeName(locale1, locale2, cat);
std::locale localeName(locale1, localeCode, cat);
```

第一种根据全局区域表示创建一个 locale 对象。第二种根据 localeCode 指明的区域表示代码字符串来创建 locale 对象。第三种根据名为 locale1 的 locale 对象创建一个副本，并用 locale2 对象中 cat 指明的 facet 代替 locale1 的 facet。第四种与第三种类似，不同之处在于它的第二个参数是区域表示代码字符串 localeCode，指明 facet 的所属区域。区域表示代码详见表 12-6。

表 12-6 区域表示代码

语 言	名 称
中文	"chinese"
中文 (简体)	"Chinese-simplified" 或 "chs"
中文 (繁体)	"Chinese-traditional" 或 "cht"

(续)

语 言	名 称
捷克语	“czech”或“csy”
丹麦语	“danish”或“dan”
荷兰语	“dutch”或“nld”
荷兰语 (比利时)	“belgian”, “dutch-belgian”或“nlb”
英语 (默认)	“english”
英语 (澳大利亚)	“australian”、“ena”或“english-aus”
英语 (加拿大)	“canadian”、“enc”或“English-can”
英语 (新西兰)	“english-nz”或“enz”
英语 (英国)	“english-uk”、“eng”, 或“uk”
英语 (美国)	“american”、“american english”、“american-english”、“english-american”、“english-us”、“english-usa”、“enu”、“us”或“usa”
芬兰语	“finish”或“fin”
法语 (默认)	“french”或“fra”
法语 (比利时)	“french-belgian”或“frb”
法语 (加拿大)	“french-canadian”或“fre”
法语 (瑞士)	“french-swiss”或“frs”
德语 (默认)	“german”或“deu”
德语 (澳大利亚)	“german-austrian”或“dea”
德语 (瑞士)	“german-swiss”, “des”, 或“swiss”
希腊语	“greek”或“ell”
匈牙利语	“hungarian”或“hun”
冰岛语	“icelandic”或“isl”
意大利语 (默认)	“italian”或“ita”
意大利语 (瑞士)	“italian-swiss”或“its”
日语	“japanese”或“jpn”
朝鲜语	“kor”或“korean”
挪威语 (博克马尔语)	“norwegian-bokmal”或“nor”
波兰语	“polish”或“plk”
葡萄牙语 (默认)	“portuguese”或“ptg”
葡萄牙语 (巴西)	“portuguese-brazilian”或“ptb”
俄语 (默认)	“russian”或“rus”
斯洛伐克语	“slovak”或“sky”
西班牙语 (默认)	“spanish”或“esp”
西班牙语 (墨西哥)	“Spanish-mexican”或“esm”
西班牙语 (现代)	“Spanish-modern”或“esn”
瑞典语	“Swedish”或“sve”
土耳其语	“turkish”或“trk”

例 12-2 给出了创建和使用区域表示的过程，目的是按英语、法语、德语的格式显示日期。此例的功能是：当程序调用仅有一个空字符串实参的 locale 构造函数时，返回的 locale 对象根据系统的用户设置被设置为本地区域表示。用户通常可以通过设置系统变量 (LANG) 来设置本地区域表示。

例 12-2

```
# include <iostream>
# include <locale>
# include <time.h>
using namespace std;
void main()
{
    char dataStr[81];
    time_t curtime;
    struct tm* tmTime;
    time(& curtime);           //获取当前时间
    tmTime = gmtime(& curtime);
    strftime(dataStr,80,"% # x",tmTime); //转换时间至 tm 结构
    std::locale native("");
    std::locale::global(native);       //转换为字符串格式
    std::cout << "Native Date: " << std::endl;
    std::cout << dataStr << std::endl << std::endl;
    std::locale german("german");
    std::locale::global(german);
    strftime(dataStr,80,"% # x",tmTime);
    std::cout << "German Date: " << std::endl;
    std::cout << dataStr << std::endl << std::endl;

    std::locale chinese("chinese");
    std::locale::global(chinese);
    strftime(dataStr,80,"% # x",tmTime);
    std::cout << "Chinese Date: " << std::endl;
    std::cout << dataStr << std::endl << std::endl;

    std::locale french("French");
    std::locale::global(french);
    strftime(dataStr,80,"% # x",tmTime);
    std::cout << "French Date: " << std::endl;
    std::cout << dataStr << std::endl << std::endl;
}
```

例 12-2 的执行结果为：

```
Native Date:
Friday, September 21, 2012
German Date:
```

```

Freitag, 21. September 2012
Chinese Date:
2012 年 09 月 21 日
French Date:
vendredi 21 septembre 2012
French Date:
vendredi 21 septembre 2012

```

在上述代码中，代码行“`std::locale native ("");`”是值得仔细体会的。

通过把一个区域表示的 facet 修改成另一个区域表示的相应 facet，可以创建混合区域表示。

例 12-3

```

#include <iostream>
#include <locale>
#include <clocale>
#include <ctime>
int main()
{
    char dataStr[81];
    time_t curtime;
    struct tm* tmtime;
    time(& curtime);
    tmtime = gmtime (& curtime);
    strftime(dataStr,80, "% # x", tmtime);
    std::locale french(std::locale("french"), std::locale("american"), LC_TIME);
    std::locale::global(french);
    strftime(dataStr,80, "% # x", tmtime);
    std::cout << "French Date: " << std::endl;
    std::cout << dataStr << std::endl;
}

```

例 12-3 的执行结果为：

```

French Date:
Friday, September 21, 2012

```

本例在调用 locale 类的构造函数时，新建的 locale 对象是法语区域表示，但该对象的时间 facet 被替换为美国版。导致该区域表示的日期具有美国英语格式，其他 facet 仍支持法语格式。LC_TIME 是变量。区域表示的种类值见表 12-7。

表 12-7 区域表示的种类值

值	说 明
LC_ALL	设置所有种类
LC_COLLATE	设置与排序函数有关的区域表示种类
LC_CTYPE	设置与字符分类函数有关的区域表示种类

(续)

值	说 明
LC_MONETARY	设置影响货币格式的区域表示种类
LC_NUMERIC	设置影响数字格式的区域表示种类
LC_TIME	设置影响时间和日期格式的区域表示种类

12.3.4 流和区域

多数情况下，应用程序可以使用流简化区域表示，尤其在应用程序同时支持多种区域表示时。由于可以把区域表示灌输进一个流，此后这个流可根据区域表示的刻面格式化数据。可以为程序需要支持的所有区域表示创建和灌输多个流。通过调用流对象的成员函数 `imbue()`，可以实现把某种区域表示灌输到流中。本章多次提到成员函数 `imbue()`。现举例说明，请参考例 12-4。

例 12-4

```
#include <iostream>
#include <locale>
using namespace std;
void main()
{
    locale native("");
    cout.imbue(native);
    cout << "native Number: " << std::endl;
    cout << 10999.82 << endl;
    locale dutch("dutch");
    cout.imbue(dutch);
    cout << "Dutch Number: " << endl;
    cout << 10999.82 << endl;
    locale french("french");
    cout.imbue(french);
    cout << "French Number: " << endl;
    cout << 10999.82 << endl;
}
```

例 12-4 的执行结果为：

```
native Number:
10999.8
Dutch Number:
10999,8
French Number:
10999,8
```

12.3.5 刻面的处理

由于 `locale` 对象的 `facet` 提供用于本地化软件的服务，因此需要一种访问和使用该服务的方法。提供重要支持的模板函数包括两个：`has_facet` 和 `use_facet`。在给定的区域表示中存

在指明的刻面，模板 `has_facet` 函数返回 `true` 值。例如，若判断德语区域表示是否支持 `ctype` facet，可编写以下代码：

```
std::locale german("german");
bool OK = std::has_facet<std::ctype<char>>(german);
```

每种类型的 facet 均定义了函数，程序可以调用函数处理表示所用数据。通常使用 `use_facet` 调用 facet 对象的成员函数。假定需要调用 `ctype` 类型刻面的 `toupper()` 函数，在德语区域表示规则把字符串转化成大写形式。例如，

```
string test = "abcdefghijklmnopqrstuvwxyz";
char* first = test.begin();
char* last = test.end();
std::use_facet<std::ctype<char>>(german).toupper(first, last);
```

此处的 `std::has_facet` 和 `std::use_facet` 的调用某些实际参数的使用方法，在此之前并未出现。模板函数 `has_facet` 的使用方法如下：

```
template<class Facet> bool has_facet(const locale& _Loc);
```

参数 `Loc` 是现存 facet 的 `locale` 对象。如果 `locale` 类的对象具有被测试过的刻面，函数返回值为 `true`；否则，将返回 `false`。



说明 模板函数是非常有用的，尤其在一个 `locale` 中，检查非强制的 facet 是否被列入。在模板函数 `use_facet()` 被调用之前，避免抛出异常。

模板函数 `use_facet` 的使用方法：

```
template<class Facet> const Facet& use_facet(const locale& _Loc);
```

参数 `_Loc` 是 `locale` 类型的常量，包含被引用的 facet 类型。函数返回值是参数 `_Loc` 的引用。函数使用说明：只要保留的任一版本 `locale` 存在，通过模板函数返回的刻面引用保留了有效性。如果没有这样的 facet 对象被列入在参数 `locale` 中，函数会抛出一个 `bad_cast` 类型的异常。

例 12-5

```
#include <iostream>
#include <locale>
#include <string>
using namespace std;
void main()
{
    locale german("german");
    locale::global(german);
    bool OK = std::has_facet<std::ctype<char>>(german);
    if(! OK)
    {
        cout << "Can't perform the conversion. ";
    }
    else
    {
```

```
        cout << "Can perform the conversion. \n";
    }
    char test[] = "abcdefghijk";
    char* first = test;
    char* last = first + sizeof(test);
    cout << "Original String: " << endl;
    cout << first << endl;
    use_facet < ctype < char >> (german). toupper (first, last);
    cout << "Converted String: " << endl;
    cout << first << endl;
    locale loc1 ("German_Germany"), loc2 ("English_Australia");
    bool r1 = use_facet < ctype < char >> (loc1). is (ctype_base::alpha, 'a');
    bool r2 = use_facet < ctype < char >> (loc2). is (ctype_base::alpha, '! ');
    if (r1)
    {
        cout << "is alphabetic. " << endl;
    }
    else
    {
        cout << "Is not alphabetic. " << endl;
    }
    if (r2)
    {
        cout << "Is alphabetic. " << endl;
    }
    else
    {
        cout << "Is not alphabetic. " << endl;
    }
}
```

例 12-5 的执行结果为：

```
Can perform the conversion.
Original String:
abcdefghijk
Converted String:
ABCDEFGHIJK
is alphabetic.
Is not alphabetic.
```

12.4 标准 locale 的分类

每一个标准种类包括：刻画（facets）族。数据实例的格式化或分解，对于标准的使用或用户的流操作符“<<”和“>>”，可以作为成员 put() 和 get()。每一个这样的成员函数均采用一个 ios_base& 参数，其成员 flags()、precision() 和 width() 用于规格化相应的数据。那些

需要使用其他刻面的函数在调用其成员 `getloc()`，以返回通道性质的 `locale` 对象。格式化刻面使用字符参数 “fill” 以完成特定的必需的宽度。成员函数 `put()` 不提供错误报告（任何 `Output_iterator` 类型参数必须从已返回的迭代器中被抽取）。成员 `get()` 函数采用一个 `ios_base::iostate&` 的参数，该参数的值可以忽略，但为防止描述错误要设置失败位（`ios_base::failbit`）。

12.4.1 类 ctype

类 `ctype_base` 的声明形式为：

```
namespace std{
class ctype_base{
public:
enum mask{
space=1<<0, print=1<<1, cntrl=1<<2, upper=1<<3, lower=1<<4, alpha=1<<5, digit=1<<6,
punct=1<<7, xdigit=1<<8, alnum=alpha|digit, graph=alnum|punct
};
};
}
```

其中类型 `mask` 是位掩码类型。

1. 模板类 ctype

模板类 `ctype` 的声明形式为：

```
template <class charT> class ctype: public locale::facet, public ctype_base{
public:
typedef charT char_type;
explicit ctype(size_t refs=0);
bool is(mask m, charT c) const;
const charT* is(const charT* low, const charT* high, mask* vec) const;
const charT* scan_is(mask m, const charT* low, const charT* high) const;
const charT* scan_not(mask m, const charT* low, const charT* high) const;
charT toupper(charT c) const;
const charT* toupper(charT* low, const charT* high) const;
charT tolower(charT c) const;
const charT* tolower(charT* low, const charT* high) const;
charT widen(char c) const;
const char* widen(const char* low, const char* high, charT* to) const;
char narrow(charT c, char default) const;
const charT* narrow(const charT* low, const charT* , char default, char* to)
const;
static locale::id id;
protected:
~ctype();
virtual bool do_is(mask m, charT c) const;
virtual const charT* do_is(const charT* low, const charT* high, mask* vec) const;
```

```
virtual const charT* do_scan_is(mask m, const charT* low, const charT* high) const;
virtual const charT* do_scan_not(mask m, const charT* low, const charT* high) const;
virtual charT do_toupper(charT) const;
virtual const charT* do_toupper(charT* low, const charT* high) const;
virtual charT do_tolower(charT) const;
virtual const charT* do_tolower(charT* low, const charT* high) const;
virtual charT do_widen(char) const;
virtual const char* do_widen(const char* low, const char* high, charT* dest) const;
virtual char do_narrow(charT, char default) const;
virtual const charT* do_narrow(const charT* low,
                               const charT* high, char default, char* dest) const;
};
```

类 `ctype` 包装了 C 库头文件 `<cctype>` 的特性，流 `istream` 成员被要求使用 `ctype<>`，在输入分析时用于字符的分类。在表 12-2 中，即 `ctype<char>` 和 `ctype<wchar_t>` 的实例化过程中，字符分类适应本地字符集的实例过程。

下面逐一介绍类 `ctype` 的成员函数。

2. 类 `ctype` 的成员函数

(1) `is()`

```
bool is(mask m, charT c) const;
const charT* is(const charT* low, const charT* high, mask* vec) const;
```

函数返回值：`do_is(m, c)` 或 `do_is(low, high, vec)`。这两个函数用于测试“单个字符是否具有特殊的属性或对每个字符进行分类，并将该字符存储在一个数组之中”。参数 `m` 是为字符设置的掩码；参数 `c` 是需要测试属性的字符；参数 `low` 和 `high` 指定一个范围，用于判断被测试字符是否在这个范围内；参数 `vec` 指向存储字符掩码的数组的起始位置，这些掩码值描述了存储的每个字符的属性。第一个成员函数返回一个布尔量的值，如果被测试的字符具有掩码描述的属性，返回 `true`；如果不具备掩码规定的属性，返回 `false`。第二个成员函数返回一个指针，指向划定属性范围的最后一个字符。

掩码值用于划分字符的分类，使用 STL 提供的类 `ctype`。类 `ctype` 由 `ctype` 派生而来。第一个 `is()` 函数可以接受掩码规定的表达式。该表达式的第一个字符必须满足掩码的要求，掩码值是通过逻辑位操作实现的。

(2) `scan_is()`

```
const charT* scan_is(mask* m, const charT* low, const charT* high) const;
```

函数返回值：`do_scan_is(m, low, high)`。`scan_is()` 函数用于定位和掩码匹配的第一个字符。参数 `m`、`low` 和 `high` 的意义和 `is()` 函数相同。函数的返回值是一个指针，在指定范围内，指向和掩码匹配的第一个字符；如果该值不存在，函数返回 `high`。

(3) `scan_not()`

```
const charT* scan_not(mask m, const charT* low, const charT* high) const;
```

函数返回值: `do_scan_not(m, low, high)`。`scan_not()` 函数的功能和 `scan_is()` 函数相反, 用于定位和掩码不匹配的字符。

(4) `toupper()`

```
charT toupper(charT c) const;
const charT* toupper(charT* low, const charT* high) const;
```

函数返回值: `do_toupper(c)` 或 `do_toupper(low, high)`。`toupper()` 函数的功能是将单个字符或某个范围的字符转换成大写形式。第一个函数的参数代表需要被转换的字符; 第二个函数的参数 `low` 和 `high` 用于限定需要转换字符的范围 `[low, high]`。

第一种形式的 `toupper()` 函数返回其参数的大写形式; 如果没有大写形式存在, 将返回参数本身。第二种形式的 `toupper()` 函数返回一个常量指针, 该指针指向被转换的字符范围内的最后一个字符。

(5) `tolower()`

```
charT tolower(charT c) const;
const charT* tolower(charT* low, const charT* high) const;
```

函数返回值: `do_tolower(c)` 或 `do_tolower(low, high)`。其中参数 `c` 是需要被转换成小写形式的字符, 参数 `low` 和 `high` 用于划定需要转换成小写形式的字符范围 `[low, high]`。第一种形式函数返回的是参数 `c` 的小写形式, 如果参数 `c` 没有小写形式存在, 函数返回参数 `c` 本身; 第二种形式函数返回一个常量指针, 该指针指向范围 `[low, high]` 内的最后一个字符。

(6) `widen()`

```
charT widen(char c) const;
const char* widen(const char* low, const char* high, charT* to) const;
```

函数返回值: `do_widen(c)` 或 `do_widen(low, high, to)`。第一种形式的函数返回本地类型字符 `char` 对应的 `CharType` 类型的字符; 第二种形式的函数返回一个指向目标范围的指针, 该指针类型为 `CharType`, 该类型是通过 `locale` 将本地字符类型 `char` 转换而来的。

`widen()` 函数用于将本地字符集 `char` 类型的字符, 转换为相应的 `CharType` 类型字符。`CharType` 类型在 `locale` 类中使用。此方法是不安全的, 因为其前提是调用者判断原有的值是正确的。参数 `c` 是本地字符集中的字符, 该字符会被转换; 参数 `low` 是一个指针, 该指针指向需要被转换的字符范围的第一个字符; 参数 `high` 也是一个指针, 该指针指向规定范围 (`[low, high]`) 内的最后一个字符后面的字符。参数 `to` 是一个指针, 该指针指向目标范围中 `CharType` 类型的第一个字符, 该范围中存储了需要被转换的字符。



提示 `CharType` 用于在类 `locale` 中表示字符类型。

(7) `narrow()`

```
char narrow(charT c, char default) const;
const charT* narrow(const charT* low, const charT* high, char default, char* to) const;
```

函数返回值: `do_narrow(c, default)` 或 `do_narrow(low, high, default, to)`。

`narrow()` 函数用于将类 `locale` 中的 `CharType` 类型字符转换成相应的本地字符集的 `char` 类

型的字符。同样，此方法也是不安全的，因为其前提是调用者判断其旧值是正确的。参数 *c* 是类 *locale* 中的 *CharType* 类型的数据，是用来被转换的；参数 *dfault* 参数默认为 ‘\0’，是该函数的默认值；参数 *low* 和 *high* 用于限定一个字符集的范围，该范围的字符会进行 narrow 转换；参数 *to* 是指向存储被转换字符的目标范围第一个字符的指针。

函数的返回值：返回本地 *char* 类型的字符，该字符是和 *CharType* 参数 *dfault* 相对应的。第二种形式的函数返回一个指向本地字符集的目标范围指针，本地字符集是 *CharType* 类型的字符转换而来的。

第一种形式的函数返回 `do_narrow(c, dfault)`；第二种形式的函数返回 `do_narrow(low, high, dfault, to)`。仅仅基础的字符集被保证拥有独一无二的反向映像。对于最基本的源字符，表达式是成立的。

```
narrow(widen(c), 0) == c
```



提示 任何 *ctype* 类的优先于 `narrow()` 的方法同样也优先于 *ctype* 类的 `narrow_s` 方法

3. 类 *ctype* 的虚函数成员

(1) `do_is()`

```
bool do_is(mask m, charT c) const;
const charT* do_is(const charT* low, const charT* high, mask* vec) const;
```

上述两个函数用于分类一个字符或字符序列。对于每个参数，要识别掩码参数 *vec* 的值。第二种形式用以识别范围 [*low*, *high*] 中每个字符的掩码的值，并将其放入向量 [*p-low*] 中。第一种形式的函数返回值返回表达式 $(M \& m) \neq 0$ 的结果；如果字符具备指定的特性，函数返回 *true*，第二种形式函数返回 *high*。此函数被调用时，其功能是判断单个字符是否具有特定的属性；或对某范围内或数组内的每个字符进行分类。

如果被测试的字符具有特定的（由 *mask* 确定）属性，函数返回 *true*；否则，函数返回 *false*。第二种形式的函数返回一个数组，该数组中容纳指定范围内字符的掩码特性。

(2) `do_scan_is()`

```
const charT* do_scan_is(mask m, const charT* low, const charT* high) const;
```

`do_scan_is()` 函数用于在一个缓冲区中定位一个字符，该字符满足分类（掩码）特性 *m*。函数的返回值是范围 [*low*, *high*] 中满足特定掩码特性的第一个指针。若满足这一条件，`is(m, *p)` 函数会返回 *true*，否则，返回 *high*。如果没有相应的值存在，函数返回 *high*。

(3) `do_scan_not()`

```
const charT* do_scan_not(mask m, const charT* low, const charT* high) const;
```

`do_scan_not()` 函数用于定位缓冲区中不满足条件的第一个字符。如果没有相应的值存在，函数返回 *high*。保护成员函数返回范围 [*low*, *high*] 中最小的指针，该指针满足函数 `do_is()` 的返回值为 *false*。

(4) `do_toupper()`

```
charT do_toupper(charT c) const;
const charT* do_toupper(charT* low, const charT* high) const;
```

`do_toupper()` 函数用于转换单个字符或多个字符, 以获取其大写形式。如果字符相应的大写字母存在, 函数的第二种形式会取代指定范围 `[low, high]` 中的每个字符; 否则, 返回该字符本身。如果参数 `c` 的大写字母形式存在, 第一种形式的函数会返回参数 `c` 相应的大写字母; 如果参数的大写形式不存在, 返回字符本身。第二种形式会返回指针 `high`。

(5) `do_tolower()`

```
charT do_tolower(charT c) const;
const charT* do_tolower(charT* low, const charT* high) const;
```

上述两种形式的函数用于将单个字符或多个字符转换成小写形式。第二种形式会转换范围 `[low, high]` 中的每个字符成为其小写形式, 如果该字符不存在小写形式, 会返回字符本身。第一种形式的函数, 如果参数 `c` 的小写形式存在, 会返回相应的小写形式; 如果不存在, 返回参数本身。第二种形式的函数会返回参数 `high`。

(6) `do_widen()`

```
charT do_widen(char c) const;
const char* do_widen(const char* low, const char* high, charT* dest) const;
```

上述两种形式的函数用于将本地字符集中的 `char` 类型字符转换为相应的类 `locale` 中的 `CharType` 类型。本方法存在潜在的不安全性, 它依赖于调用者判断该值是正确的。对于任何具名的 `ctype` 类刻面和有效的 `ctype_base::mask` 的值 (`M (is (M, c) || ! ctw.is (M, do_widen (c)))`) 为真。对于字符的第二种形式的转换, 其结果保存在缓冲区 `dest` 中。函数返回值是变化后的值, 第二种形式返回 `high`。

(7) `do_narrow()`

```
char do_narrow(charT c, char dfault) const;
const charT* do_narrow(const charT* low, const charT* high, char dfault, char* dest) const;
```

`do_narrow()` 函数用于将 `charT` 类型的单个字符或多个字符转换为相应的 `char` 类型值。对于任何基本源字符集中的 `char` 类型字符 `c`, 其变换具有以下性质:

```
do_widen(do_narrow(c,0)) == c
```

对于任何具名的 `ctype` 类型, 其刻面和掩码 `ctype_base::mask` 的值 `M` 之间表达式 (下述) 的值为真:

```
(is(M,c) || ! ctc.is(M, do_narrow(c, dfault)))
```

除非 `do_narrow` 返回 `dfault`。另外, 对于任何数字式字符, 表达式 (`do_narrow (c, dfault) - '0'`) 评估为该字符的数字值。第二种形式转换范围 `[low, high]` 中每个字符, 并将转换之后的结果放至缓冲区 `dest` 中。第一种形式返回转换之后的值, 如果没有匹配的值会返回默认值 `dfault`; 第二种形式返回 `high`。

4. 举例

下面使用例 12-6 来说明类 `ctype` 的具体使用方法。

例 12-6

```
#include <iostream>
#include <locale>
using namespace std;
```

```
void main()
{
//widen
    locale loc("English");
    char* str="Hello everyone!";
    wchar_t s2[16];
    bool r = (use_facet<ctype<wchar_t>>(loc).widen(str, str + strlen(str), &s2[0]) != 0);
// C4996
    s2[strlen(str)] = '\0';
    cout << str << endl;
    wcout << &s2[0] << endl;
    ctype<wchar_t>::char_type charT;
    charT = use_facet<ctype<char>>(loc).widen('a');
    cout << charT << " : " << (char)charT << endl;
//do_is or is
    locale loc1("German_Germany"), loc2("English_Australia");
    if(use_facet<ctype<char>>(loc1).is(ctype_base::alpha, 'a'))
    {
        cout << "The character 'a' in locale loc1 is alphabetic. " << endl;
    }
    else
    {
        cout << "The character 'a' in locale loc1 is not alphabetic. " << endl;
    }
    if(use_facet<ctype<char>>(loc2).is(ctype_base::alpha, '!'))
    {
        cout << "the character '! ' in locale loc2 is alphabetic. " << endl;
    }
    else
    {
        cout << "The character '! ' in loc2 is not alphabetic. " << endl;
    }
    char* str2="Hello, my name is John!";
    ctype<char>::mask maskarray[30];
    use_facet<ctype<char>>(loc2).is(str2, str2 + strlen(str2), maskarray);
    for (int i = 0; i < strlen(str2); i++)
    {
        cout << str2[i] << " : " << maskarray[i] << " " << (maskarray[i] & ctype_base::alpha ? "alpha"
: "not alpha") << endl;
    };
    cout << endl;
//do_narrow and narrow
    locale loc3 ("english");
    wchar_t * str3 = L"\x0392fhello everyone";
    char str4 [16];
    bool r3 = (use_facet<ctype<wchar_t>>(loc3).narrow(str3, str3 + wcslen(str3), 'X',
```

```

& str4[0] ) != 0); str4[wcslen(str3)] = '\0';
    wcout << str3 << endl;
    cout << & str4[0] << endl;
//scan_is and scan_not
    locale loc4 ( "German_Germany" );
    char * str5 = "Hello, my name is John!";
const char* i = use_facet<ctype<char>> ( loc4 ).scan_is( ctype_base::punct, & str5[0], & str5
[strlen(& str5[0]) -1] );
    cout << "The first punctuation is \"" << * i << "\" at position: "
        << i - str5 << endl;

    i = use_facet<ctype<char>> ( loc4 ).scan_not( ctype_base::alpha, & str5[0], & str5[strlen
(& str5[0]) -1] );
    cout << "First nonalpha character is \"" << * i << "\" at position: "
        << i - str5 << endl;
//tolower
    locale loc5 ( "German_Germany" );
    char str6[] = "HELLO, MY NAME IS John!";
    use_facet<ctype<char>> ( loc5 ).tolower( & str6[0], & str6[strlen(& str6[0]) -1] );
    cout << "The lowercase string is: " << str6 << endl;
//toupper
    locale loc6 ( "German_Germany" );
    char str7[] = "Hello, my name is John!";
    use_facet<ctype<char>> ( loc6 ).toupper( & str7[0], & str7[strlen(& str7[0]) -1] );
    cout << "The uppercase string is: " << str7 << endl;
}

```

例 12-6 的执行结果为:

```

Hello everyone!
Hello everyone!
97 : a
The character 'a' in locale loc1 is alphabetic.
The character '! ' in loc2 is not alphabetic.
H: 769 alpha
e: 898 alpha
l: 770 alpha
l: 770 alpha
o: 770 alpha
,: 528 not alpha
: 584 not alpha
m: 770 alpha
y: 770 alpha
: 584 not alpha
n: 770 alpha
a: 898 alpha
m: 770 alpha

```

```
e: 898 alpha
: 584 not alpha
i: 770 alpha
s: 770 alpha
: 584 not alpha
J: 769 alpha
o: 770 alpha
h: 770 alpha
n: 770 alpha
!: 528 not alpha
```

```
Xhello everyone
```

```
The first punctuation is "," at position: 5
```

```
First nonalpha character is "," at position: 5
```

```
The lowercase string is: hello, my name is john!
```

```
The uppercase string is: HELLO, MY NAME IS JOHN!
```

5. 类 ctype_byname

类 `ctype_byname` 的声明形式如下（注意第二行代码中的加粗字）：

```
namespace std{
    template<class charT> class ctype_byname : public ctype<charT> {
    public:
        typedef ctype<charT>::mask mask;
        explicit ctype_byname(const char* , size_t refs=0);
    protected:
        ~ctype_byname();
        virtual bool do_is(mask m, charT c) const;
        virtual const charT* do_is(const charT* low, const charT* high, mask* vec) const;
        virtual const charT* do_scan_is(mask m, const charT* low, const charT* high) const;
        virtual const charT* do_scan_not(mask m, const charT* low, const charT* high) const;
        virtual charT* do_toupper(charT) const;
        virtual const charT* do_toupper(charT* low, const charT* high) const;
        virtual charT* do_tolower(charT) const;
        virtual const charT* do_tolower(charT* low, const charT* high) const;
        virtual charT do_widen(char) const;
        virtual const char* do_widen(const char* low, const char* high, charT* dest) const;
        virtual char do_narrow(charT, char default) const;
        virtual const charT* do_narrow(const charT* low, const charT* high, char default, char* dest) const;
    };
}
```

6. 类 ctype 的实例化类举例

类 `ctype` 的实例化（主要实例化 `char` 类型）在本小节描述。例如，

```

namespace std{
    template < > class ctype < char > : public locale::facet, public ctype_base{
public:
    typedef char char_type;
    explicit ctype(const mask* tab=0, bool del=false, size_t refs=0);
    bool is(mask m, char c) const;
    const char* is(const char* low, const char* high, mask* vec) const;
    const char* scan_is(mask m, const char* low, const char* high) const;
    const char* scan_not(mask m, const char* low, const char* high) const;
    char toupper(char c) const;
    const char* toupper(char* low, const char* high) const;
    char tolower(char c) const;
    const char* tolower(char* low, const char* high) const;
    char widen(char c) const;
    const char* widen(const char* low, const char* high, char* to) const;
    char narrow(char c) const;
    const char* narrow(const char* low, const char* high, char* to) const;
    static locale::id id;
    static const size_t table_size = IMPLEMENTATION_DEFINED;
protected:
    const mask* table() const throw();
    static const mask* classic_table() throw();
    ~ctype();
    virtual char do_toupper(char c) const;
    virtual const char* do_toupper(char* low, const char* high) const;
    virtual char do_tolower(char c) const;
    virtual const char* do_tolower(char* low, const char* high) const;
    virtual char do_widen(char c) const;
    virtual const char* do_widen(const char* low, const char* high, char* to) const;
    virtual char do_narrow(char c) const;
    virtual const char* do_narrow(const char* low, const char* high, char* to) const;
};
}

```

一个类 `ctype` 的实例化特例使得类型 `char` 的成员函数可以被在线实施。可实施的成员 `table_size` 的值至少等于 256。

(1) `ctype < char >` 析构器

```
~ctype();
```

如果构造器的第一个参数是非零的，并且构造器的第二个参数是“true”，析构器的作用等同于执行以下语句：

```
delete[] table();
```

(2) `ctype < char >` 成员函数

下面介绍 `ctype < char >` 成员函数。对于非符号字符型的值 `v`，此处 ($v \geq \text{table_size}$)，

`table()` [v] 是被假定拥有一个可实施的值（对于每个这样的值，存在可能的差异），且没有执行数组搜索。

```
explicit ctype(const mask* tbl=0, bool del=false, size_t refs=0);
```

前提条件：`tbl` 或者为 0，或者是 `table_size` 的数组，该数组至少包含 `table_size` 个元素。该成员函数的作用是传递参数 `refs` 给其基类的构造器。

```
bool is(mask m, char c) const;
const char* is(const char* low, const char* high, mask* vec) const;
```

函数的作用：第一种形式会返回 `table()` [(unsigned char) c] & m，第二种形式返回参数 `high`。

```
const char* scan_is(mask m, const char* low, const char* high) const;
```

函数返回值是指定范围 [low, high] 中的最小元素，以使 `table()` [unsigned char * p] & m 的值是 true。

```
const char* scan_not(mask m, const char* low, const char* high) const;
```

函数返回值是指定范围 [low, high] 中最小的值，以使 `table()` [unsigned char * p] & m 的值为 false。

```
char toupper(char c) const;
const char* toupper(char* low, const char* high) const;
```

函数返回值是 `do_toupper(c)` 或 `do_toupper(low, high)`。

```
char tolower(char c) const;
const char* tolower(char* low, const char* high) const;
```

函数返回值是 `do_tolower(c)` 或 `do_tolower(low, high)`。

```
char widen(char c) const;
const char* widen(const char* low, const char* high, char* to) const;
```

函数返回值是 `do_widen(c)` 或 `do_widen(low, high, to)`。

```
char narrow(char c, char /* default */) const;
const char* narrow(const char* low, const char* high, char /* default */, char* to) const;
```

函数返回值是 `do_narrow(c)` 或 `do_narrow(low, high, to)`。

```
const mask* table() const throw()
```

函数返回值：如果构造器的第一个参数是非零，函数返回值是该参数；否则，返回 `classic_table()`。

(3) ctype < char > 的静态成员

```
static const mask* classic_table() throw();
```

函数返回值是数组初始元素的指针，数组的大小为 `table_size`，该指针代表了类 `locale` 的字符类别。

(4) ctype < char > 的虚函数

```

char do_toupper(char) const;
const char* do_toupper(char* low, const char* high) const;
char do_tolower(char) const;
const char* do_tolower(char* low, const char* high) const;
virtual char do_widen(char c) const;
virtual const char* do_widen(const char* low, const char* high, char* to) const;
virtual char do_narrow(char c, char ddefault) const;
virtual const char* do_narrow(const char* low, const char* high, char ddefault, char*
to) const;

```

上述这些函数和 ctype 模板中与其同名的函数具有同样的功能。

7. 类 ctype_byname < char >

类 ctype_byname < char > 的声明形式如下:

```

namespace std{
template < > class ctype_byname < char > : public ctype < char > {
public:
    explicit ctype_byname(const char* , size_t refs=0);
protected:
    ~ctype_byname();
    virtual char do_toupper(char c) const;
    virtual const char* do_toupper(char* low, const char* high) const;
    virtual char do_tolower(char c) const;
    virtual const char* do_tolower(char* low, const char* high) const;
    virtual char do_widen(char c) const;
    virtual const char* do_widen(const char* low, const char* high, char* to);
    virtual char do_narrow(char c, char ddefault) const;
    virtual const char* do_narrow(const char* low, const char* high, char* to);
}
}

```

8. 模板类 codecvt

模板类 codecvt 的声明形式如下:

```

namespace std{
    class codecvt_base{
    public:
        enum result {ok, partial, error, noconv};
    };
    template <class internT, class externT, class stateT > class codecvt: public locale::facet, public
    codecvt_base{
    public:
        typedef internT intern_type;
        typedef externT extern_type;
        typedef stateT state_type;
        explicit codecvt(size_t refs=0);
        result out(stateT& stste, const internT* from, const internT* from_end,

```

```

const internT* & from_next, internT* to, internT* to_limit, internT* & to_next) const;
    int encoding() const throw();
    bool always_noconv() const throw();
    int length(stateT& , const externT* from, const externT* end, size_t max) const;
    int max_length() const throw();
    static locale::id id;
protected:
    ~codecvt();
    virtual result do_out(stateT& state, const internT* from, const internT* from_end, const internT* &
from_next, externT* to, externT* to_limit, externT* to_next) const;
    virtual result do_in(stateT& state, const extent* from, const externT* from_end, const externT
* & from_next,
        internT* to, internT* to_limit, internT* & to_next) const;
    virtual result do_unshift(stateT& state, externT* to, externT* to_limit, externT* & to_next)
const;
    virtual int do_encoding() const throw();
    virtual bool do_always_noconv() const throw();
    virtual int do_length(stateT& , const externT* from, const externT* end, size_t max) const;
    virtual int do_max_length() const throw();
};
}

```

类 `codecvt < internT, externT, stateT >` 是用来转换代码集的。例如，从宽字符到单字节字符或者宽字符编码之间的转换（例从 unicode 码到 EUC 码的转换）。

`stateT` 类型参数是中间类型，是两种类型在相互转换时需要的中间状态类型，需要选择成对的代码集，并且两个代码集之间互相匹配。

类的实例化通常有 `codecvt < wchar_t, char, mbstate_t >` 和 `codecvt < char, char, mbstate_t >`，用于转换本地字符集。实例化类 `codecvt < char, char, misstate_t >` 实施一种退化的转变，它不实行根本的变换。实例化类 `codecvt < wchar_t, char, mbstate_t >` 实现在本地字符集的窄字符集和宽字符集之间进行转换。`mbstate` 的实例化实现在两种编码之间的转换。另一种编码可以使用自定义类型 `stateT` 实现转换。`stateT` 类型的对象可以包含任何状态，这些状态有利于在特定的 `do_in` 或 `do_out` 成员之间通信。

(1) `codecvt` 类成员

```

result out(stateT& state, const internT* from, const internT* from_end, const internT* & from_
next,
externT* to, externT* to_limit, externT* & to_next) const;

```

函数返回值是 `do_out (state, from, from_end, from_next, to, to_limit, to_next)`。转换内部类型 `CharType` 的序列为外部类型 `Byte` 的序列。

- 参数 `state`。该参数代表一种转换状态，是在调用成员函数之间的中间状态。
- 参数 `from`。该指针指向被转换序列的开始。
- 参数 `from_end`。该指针指向被转换序列的尾部。

- 参数 `from_next`。前一个 `CharType` 类型被转换之后，指向其后第一个没有转换的 `CharType` 类型数据。
- 参数 `to`。该指针指向已转换序列的开始位置。
- 参数 `to_limit`。该指针指向已转换序列的尾部。
- 参数 `to_next`。该指针指向已转换数据之后的第一个未转换的 `Byte` 类型数据。

```
result unshift(stateT& state, externT* to, externT* to_limit, externT* & to_next) const;
```

函数返回值是 `do_unshift (state, to, to_limit, to_next)`。在转换过程中，为了完成序列中最后一个 `Byte` 类型字符的转换，提供必需的 `Byte` 类型数据。参数 `state` 是中间过渡状态数据类型；参数 `to` 是指针，该指针指向目的范围的第一个位置；参数 `to_limit` 也是指针，该指针指向目的范围的最后一个位置；参数 `to_next` 是指针，该指针指向目的序列中第一个未转换的元素。

```
result in(stateT& state, const externT* from, const externT* from_end, const externT* & from_next,
internT* to, internT* to_limit, internT* & to_next) const;
```

函数返回值是 `do_in (state, from, from_end, from_next, to, to_limit, to_next)`。函数 `result in()` 用于将一种外部类型 `Byte` 的序列转换成一种内部的 `CharType` 类型的序列。参数 `state` 是中间过渡状态；参数 `from` 是指针，指向被转换序列的开始位置；参数 `from_end` 是指针，该指针指向被转换序列的尾部；参数 `from_next` 是指针，指向已转换序列尾部之后第一个未转换字符；参数 `to` 是指针，该指针指向已转换序列的起始位置；参数 `to_limit` 是指针，该指针指向已转换序列的尾部；参数 `to_next` 是指针，该指针指向目标序列中已转换的 `CharType` 序列的第一个未被转换的字符。

```
int encoding() const throw();
```

函数返回 `do_encoding()`。函数的作用：如果字节流的编码方式是状态独立的，函数判断 `Bytes` 和 `CharType` 之间的转换比率是否是常量，如果是常量，决定比率参数 `ratio` 的数值。函数的返回值如果是正值，该值是 `Byte` 类型字符的常量数量，这些 `Byte` 类型字符用于产生 `CharType` 类型的字符。函数返回值通常有 3 种形式（见表 12-8）：

表 12-8 函数 `encoding` 的返回值

返回值	条件
-1	如果 <code>extern_type</code> 类型序列的编码是状态独立的
0	如果编码涉及的序列长度是可变的
N	如果编码涉及的序列长度为 N

```
bool always_noconv() const throw();
```

函数返回值是 `do_always_noconv()`。`always_noconv()` 函数用于判断是否没有转换需要执行。函数返回值是逻辑量，如果没有转换被执行，函数返回 `true`；如果至少有一个转换被执行，函数返回 `false`。

```
int length(stateT& state, const externT* from, const externT* from_end, size_t max) const;
```

函数返回值是 do_length (state, from, from_end, max)。length() 函数用于判断有多少 Byte 类型的字节数，并返回该字节的数目。参数 state 是调用函数时要保持的中间转换状态；参数 from 是指针，指向外部序列的起始位置；参数 from_end 是指针，指向外部序列的尾部；参数 max 代表函数可返回的 Byte 类型的最大数值。

(2) 类的虚函数

```
result do_out(stateT& state, const internT* from, const internT* from_end, const internT* &
from_next,
    externT* to, externT* to_limit, externT* & to_next) const;
result do_in(stateT& state, const externT* from, const externT* from_end, const externT* &
from_next,
    internT* to, internT* to_limit, internT* & to_next) const;
```

前提条件：(from <= from_end && to < to_end) 被预先定义，且其值为 true。state 是预先初始化的，它在序列的起始位置，否则等于序列中转换字符的结果。

函数的作用：转变范围 [from, from_end] 中的字符，将其放置于目标序列 to 中。转换不少于 from_end-from 个的元素，并将存储不少于 to_limit-to 个目标元素。当面对一个字符不能转换的情况时，函数将中止；函数总是保留指针 from_next 和 to_next，指向已成功转换的最后一个元素。函数将返回 Noconv，internT 和 externT 是同一类型，并且被转换的序列和输入序列 [from, from_next) 是一致的。指针 to_next 被设定和指针 to 相等，state 的值是不变的，并且范围 [to, to_limit] 中的值是不变的。



提示 当参数 state 未指定时，函数才能运行。

函数返回值有以下几种选项（见表 12-9）：

表 12-9 函数 do_out() 和函数 do_in() 的返回值

值	意 义
ok	完成转换
partial	部分源字符被转换
error	在范围 [from, from_end] 中存在 1 个字符不能转换
noconv	internT 和 externT 是同一类型，输入序列和被转换序列是一致的

返回值 partial 说明目标序列不能吸收所有目标元素，或者另一个目标元素产生之前，附加的源元素是必需的。

```
result do_unshift(stateT& state, externT* to, externT* to_limit, externT* & to_next) const;
```

函数的作用：在当前状态 stateT 等于 state 时，将起始位置为 to 的字符放置入序列中。存储不少于 (to_limit-to) 个目标元素。函数总是保留 to_next 指针，该指针指向被存储的最后一个元素的前面。函数的返回值也是一个枚举变量，其枚举值见表 12-10。

表 12-10 函数 `do_unshift()` 的返回值

值	意 义
ok	完成序列
partial	更多的字符需要用于完成终止
error	state 具有一个无效的值
noconv	对于 state_type 类型, 不需要无终止

`codecvt < char, char, mbstate_t >` 返回 `noconv`。

```
int do_encoding() const throw();
```

若 `externT` 类型序列编码是状态独立的, 函数返回 `-1`; 否则, 常量数目的 `externT` 字符需要产生一个内部字符; 如果数目不是常量, 返回 `0`。对于所有有效参数, 如果 `do_in()` 和 `do_out()` 返回 `noconv`, 函数返回值为真; `codecvt < char, char, mbstate_t >` 返回 `true`。

```
int do_length(stateT& state, const externT* from, const externT* from_end, size_t max) const;
```

前提条件: (`from <= from_end`) 被预先定义, 其值为 `true`。state 被初始化之后, 序列的起始或许等于序列中被转换字符的结果。当参数 `state` 不确定时, 函数调用 `do_in()`, `do_in` 的参数 `to` 指向包含至少 `max` 个元素的缓冲区。

若 `from_next` 是范围 `[from, from_end]` 中的最大值, 则范围 `[from, from_next]` 中的值表示 `internT` 类型的最大的或第二大的有效完整字符。

```
int do_max_length() const throw();
```

函数返回 `do_length` 返回的最大值, 并且 `stateT` 的值为 `state`。

实例化函数 `codecvt < char, char, mbstate_t >::do_max_length()` 会返回 `1`。

下面使用例 12-7 说明类模板 `codecvt` 的使用方法。

例 12-7

```
# define _INTL
# include < locale >
# include < iostream >
# include < wchar.h >
# define LEN 90
using namespace std;
void main()
{
    char pszExt[LEN+1];
    wchar_t * pwszInt = L"This is the wchar_t string to be converted.";
    memset(& pszExt[0], 0, ( sizeof( char ) ) * ( LEN + 1 ) );
    char* pszNext;
    const wchar_t* pwszNext;
    mbstate_t state;
    locale loc("C");//English_Britain");//German_Germany
    int res = use_facet < codecvt < wchar_t, char, mbstate_t >> (loc).out( state, pwszInt, & pwszInt
[wcslen(pwszInt)], pszNext, pszExt, & pszExt[wcslen(pwszInt)], pszNext);
    pszExt[wcslen(pwszInt)] = 0;
```

```

cout << pszExt << endl;
char* pszExt2 = "This is the string to be converted!";
wchar_t pwszInt2 [LEN + 1];
memset(&pwszInt2[0], 0, (sizeof(wchar_t))* (LEN + 1));
const char* pszNext2;
wchar_t* pwszNext2;
mbstate_t state2 = {0};

res = use_facet < codecvt < wchar_t, char, mbstate_t >> (loc).in ( state, pszExt2, & pszExt2
[strlen(pszExt2)], pszNext2, pwszInt2, & pwszInt2[ strlen(pszExt2)], pwszNext2 );
pwszInt2[ strlen(pszExt2) ] = 0;
wcout << pwszInt2 << endl;
locale loc2 ( "German_Germany" );
res = use_facet < codecvt < char, char, mbstate_t >> (loc2).encoding();
cout << "Ratio: " << res << endl;
locale loc3 ( "German_Germany" );
bool resB = use_facet < codecvt < char, char, mbstate_t >> (loc).always_noconv();
if (resB)
    cout << "No conversion is needed " << endl;
else
    cout << "At least one conversion is required " << endl;
char* pszExt3 = "This is the string whose length is to be measured!";
mbstate_t state3 = {0};
locale loc4 ("C");//English_Britain");//German_Germany
res = use_facet < codecvt < wchar_t, char, mbstate_t >> (loc).length (state3, pszExt3, & pszExt3
[ strlen(pszExt3)], LEN );
wcout << "The length of the string is: " << res << "." << endl;
}

```

例 12-7 的执行结果为:

```

This is the wchar_t string to be converted.
This is the string to be converted!
Ratio: 1
No conversion is needed.
The length of the string is: 50.

```

9. 模板类 codecvt_byname

```

namespace std{
    template <class internT, class externT, class stateT>
    class codecvt_byname: public codecvt < internT, externT, stateT > {
    public:
        explicit codecvt_byname (const char* , size_t refs = 0);
    protected:
        ~codecvt_byname ();
        virtual result do_out (stateT& state, const internT* from, internT* from_end, const in-
ternT* & from_next,

```

```

externT* to, externT* to_limit, externT* & to_next) const;
virtual result do_in(stateT& state, const externT* from, externT* from_end, const externT*
& from_next,
internT* to, internT* to_limit, internT* & to_next) const;
virtual result do_unshift(stateT& state, externT* to, externT* to_limit, externT* & to_
next) const;
virtual int do_encoding() const throw();
virtual bool do_always_noconv() const throw();
virtual int do_length(stateT& , const externT* from, const externT* end, size_t max) const;
virtual result do_unshift(stateT& state, externT* to, externT* to_limit, externT* to_next)
const;
virtual int do_max_length() const throw();
};
}

```

12.4.2 数值类的类 locale

类 `num_get < >` 和类 `num_put < >` 处理数值格式和数值分解。虚函数主要用于几种数值类型，实施过程中会将较小类型处理为较大的类型。所有特例化的成员函数仅仅应用于实例化类。这些实例化类可以是 `num_get < char >`、`num_get < wchar_t >`、`num_get < C, InputIterator >`、`num_put < char >`、`num_put < wchar_t >` 和 `num_put < C, OutputIterator >`。这些实例化类涉及了 `ios_base&` 类型的参数，可用于格式化特例和与它相应的 `locale` 类。刻面 `numpunct < >` 用于识别所有数值类标点优先，同时刻面 `ctype < >` 用于字符分类。

对于标准流的抽出器和插入器成员，使用 `num_get < >` 和 `num_put < >` 成员函数格式化或分解数值。

1. 模板类 num_get

```

namespace std{
template <class charT, class InputIterator = istreambuf_iterator < charT >>
class num_get: public locale::facet {
public:
typedef charT char_type;
typedef InputIterator iter_type;
explicit num_get (size_t refs=0);
iter_type get (iter_type in, iter_type end, ios_base&, ios_base::iostate& err,
bool& v);
iter_type get (iter_type in, iter_type end, ios_base&, ios_base::iostate& err,
long& v);
iter_type get (iter_type in, iter_type end, ios_base&, ios_base::iostate& err,
unsigned short & v);
iter_type get (iter_type in, iter_type end, ios_base&, ios_base::iostate& err,
unsigned int & v);
iter_type get (iter_type in, iter_type end, ios_base&, ios_base::iostate& err,
unsigned long & v);

```

```
    iter_type get(iter_type in, iter_type end, ios_base&, ios_base::iostate& err, float &
v);
    iter_type get(iter_type in, iter_type end, ios_base&, ios_base::iostate& err, double &
v);
    iter_type get(iter_type in, iter_type end, ios_base&, ios_base::iostate& err, long
double & v);
    iter_type get(iter_type in, iter_type end, ios_base&, ios_base::iostate& err, void* &
v);
    static locale::id id;
protected:
    ~num_get();
    virtual iter_type do_get(iter_type, iter_type, ios_base&, ios_base::iostate* err, bool& v)
const;
    virtual iter_type do_get(iter_type, iter_type, ios_base&, ios_base::iostate* err, long& v)
const;
    virtual iter_type do_get(iter_type, iter_type, ios_base&, ios_base::iostate* err, unsigned
short& v) const;
    virtual iter_type do_get(iter_type, iter_type, ios_base&, ios_base::iostate* err, unsigned
int& v) const;
    virtual iter_type do_get(iter_type, iter_type, ios_base&, ios_base::iostate* err, unsigned
long& v) const;
    virtual iter_type do_get(iter_type, iter_type, ios_base&, ios_base::iostate* err, float& v)
const;
    virtual iter_type do_get(iter_type, iter_type, ios_base&, ios_base::iostate* err, double& v)
const;
    virtual iter_type do_get(iter_type, iter_type, ios_base&, ios_base::iostate* err, long doub-
le& v) const;
    virtual iter_type do_get(iter_type, iter_type, ios_base&, ios_base::iostate* err, void* & v)
const;
};
}
```

剖面 `num_get < >` 用于分解输入序列中的数值数据。

模板类 `num_get < >` 包含了部分普通成员 `get()` 函数，还包含了一部分虚 `do_get()` 函数。虚 `do_get()` 函数可实现从输入流 `in` 中读取字符，并按照 `str.flag()` 阐释这些字符，使用剖面 `use_facet < ctype < charT >> (loc)` 和 `use_facet < numpunct < charT >> (loc)`。此处 `loc` 是 `str.getloc()` 的返回值，如果产生错误，`val` 是不变的，否则其将被设置为结果数值。参数的运行细节如下。

阶段 1：判断一个特殊的转换。

阶段 2：从输入流 `in` 中读取字符，并判断相应的字符值。

阶段 3：存储结果。

各阶段的细节如下。

阶段 1：初始化本地变量。

```
fmtflags flags = str.flags();
fmtflags basefield = (flags & ios_base::basefield);
fmtflags uppercase = (flags & ios_base::uppercase);
fmtflags boolalpha = (flags & ios_base::boolalpha);
```

对于转换整数类型 `inter`，函数决定整形转换（见表 12-11）。

表 12-11 整形转换

类 型	Stdio equivalent	类 型	Stdio equivalent
<code>basefield == oct</code>	<code>%o</code>	signed integral type	<code>%d</code>
<code>basefield == hex</code>	<code>%X</code>	Unsigned integral type	<code>%u</code>
<code>basefield == 0</code>	<code>%i</code>		

若转换浮点类型，则使用标识符“`%g`”；若转换空指针（`void*`），则需要使用标识符“`%p`”。如果需要，还可以使用长度调整标识符（见表 12-12）。

表 12-12 长度调整标识符

类 型	长度调整标识符	类 型	长度调整标识符
<code>short</code>	<code>h</code>	<code>unsigned long</code>	<code>l</code>
<code>unsigned short</code>	<code>h</code>	<code>double</code>	<code>l</code>
<code>long</code>	<code>l</code>	<code>long double</code>	<code>l</code>

阶段 2：如果 `in == end`，第二阶段将终止。否则 `charT` 类型会从输入流 `in` 中抽取，并初始化给本地变量。

```
char_type ct = * in;
char c = src[find(atoms, atoms + sizeof(src) - 1, ct) - atoms]
if(ct == use_facet < num_punct < charT >> (loc).thousands_sep() && use_facet < num_punct < charT >>
(loc).grouping()
    .length() != 0)
```

值 `src` 和 `atoms` 被定义：

```
static const char src[] = "0123456789abcdefABCDEF + - ";
char_type atoms[sizeof(src)];
use_facet < ctype < charT >> (loc).widen(src, src + sizeof(src), atoms);
```

若参数 `discard` 为 `true`，则字符的位置需要被记忆；否则，字符可以被忽略。若参数 `discard` 为 `false`，需要检查字符 `c` 是否被允许作为阶段 1 中符号转换的输入流 `in` 中的下一个字符。如果为真，该字符被收集。

如果字符被抛弃或被收集，输入流使用“`++`”可以前进，并进行处理，例如 `++in`。

阶段 3：阶段 2 的处理结果可能是以下两种。

1) 在阶段 2 中，一个字符序列被收集，并被转换成该类型的值。该值被存储在 `val` 中，并且 `ios_base::goodbit` 被存储在变量 `err` 中。

2) 阶段 2 收集的字符序列会引起扫描，并汇报输入错误，之后 `ios_base::failbit` 被设置给变

量 `err`。数字分组被检查，即抛弃分隔符的位置需要被检查，并和 `use_facet < numpunct < charT >> (loc).grouping()` 保持一致。如果它们不能保持一致，`ios_base::failbit` 的值将被设置给 `err` 变量。

无论任何情况，如果阶段 2 的处理过程被条件 `(in == end)` 被终止，`err | = ios_base::eofbit` 会随即被执行。

```
iter_type do_get(iter_type in, iter_type end, ios_base& str, ios_base::iostate& err, bool& val) const;
```

如果 `(str.flag() && ios_base::boolalpha) == 0`，输入操作将继续，输入的值被存入布尔变量 `val` 中。接着需要对 `val` 的值进行判断：若被存储的值是 0，则 `val` 的值为 `false`；若被存储的值是 1，则 `val` 的值为 `true`。否则，语句 `err | = ios_base::failbit` 被执行，并不存储任何值。

一旦目标序列被判断，似乎是通过调用刻面的 `false_name()` 函数和 `true_name()` 函数。输入型迭代器 `in` 同 `end` 是可以相等的，此时 `in == end`。当且仅当目标序列被单独匹配时，`val` 的值会被设置给相应的值。`[in, end]` 中的连续字符可以获取，在目标序列中并不按位置匹配。迭代器 `in` 总是指向成功匹配的最后一个元素之前的某个位置。若 `val` 被设置，则 `err` 被设置为 `str.goodbit` 的值；或被设置为 `str.eofbit` 的值，当寻找另一个匹配的字符时，可以发现 `(in == end)`。如果 `val` 没有被设置，然后 `err` 被设置为 `str.failbit` 的值。如果函数执行失败的原因是 `(in == end)`，数值 `val` 没有被设置数值，`str.failbit` 会被设置为 1，或者将 `(str.failbit | str.eofbit)` 的值设置给 `err`。

函数返回指针 `in`。

2. 模板类 `num_put`

模板类 `num_put` 的声明形式如下：

```
namespace std{
template <class charT, class OutputIterator = ostreambuf_iterator<charT >>
class num_put: public locale::facet{
public:
    typedef charT char_type;
    typedef OutputIterator iter_type;
    explicit num_put(size_t refs=0);
    iter_type put(iter_type s, ios_base& f, char_type fill, bool v) const;
    iter_type put(iter_type s, ios_base& f, char_type fill, long v) const;
    iter_type put(iter_type s, ios_base& f, char_type fill, unsigned long v) const;
    iter_type put(iter_type s, ios_base& f, char_type fill, double v) const;
    iter_type put(iter_type s, ios_base& f, char_type fill, long double v) const;
    iter_type put(iter_type s, ios_base& f, char_type fill, const void* v) const;
    static locale::id id;
protected:
    ~num_put();
    virtual iter_type do_put(iter_type, ios_base&, char_type fill, bool v) const;
    virtual iter_type do_put(iter_type, ios_base&, char_type fill, long v) const;
    virtual iter_type do_put(iter_type, ios_base&, char_type fill, unsigned long v) const;
```

```

virtual iter_type do_put(iter_type, ios_base&, char_type fill, double v) const;
virtual iter_type do_put(iter_type, ios_base&, char_type fill, long double v) const;
virtual iter_type do_put(iter_type, ios_base&, char_type fill, const void* v) const;
};
}

```

刻面 `num_put` 用于将数值数据格式化为一个字符串序列（像一个输出流）。虚 `do_put()` 函数用于将字符写至序列 `out` 中，并格式化 `val`。下述语句用以实现本地类对象变量的初始化：

```
locale loc = str.getloc();
```

函数的运行细节发生时一般包括以下几个阶段。

阶段 1：判断一个 `printf` 的转换符标识 `spec`，并决定字符的输出。假定当前的 `locale` 对象属于“C”类型 `locale`。

阶段 2：调整表示方式。转换阶段 1 决定的每一个字符被称为宽字符型式。其值被 `getloc()` 函数返回。

```
use_facet <numput< charT >> (str.getloc());
```

阶段 3：判断何处是无用的。

阶段 4：插入序列至流 `out` 中。

下面对上述阶段进行详细描述。

阶段 1：阶段 1 的第一个行为是判断转换符标识。描述该判断的表使用下述本地变量。

```

fmtflags flags = str.flags();
fmtflags basefield = (flags & (ios_base::basefield));
fmtflags uppercase = (flags & (ios_base::uppercase));
fmtflags floatfield = (flags & (ios_base::floatfield));
fmtflags showpos = (flags & (ios_base::showpos));
fmtflags showbase = (flags & (ios_base::showbase));

```

在阶段 1 中，所有使用的表均是有序的，即只有第一行的条件为 `true`，后面的语句才能执行。对于从整型到字符型的转换，函数会判断整型转换符标识。详见表 12-13。

表 12-13 整型数值转换字符型

状 态	stdio 等效
<code>basefield == ios::base::oct</code>	<code>%o</code>
<code>(basefield == ios::base::hex) && ! uppercase</code>	<code>%x</code>
<code>(basefield == ios::base::hex)</code>	<code>%X</code>
for a signed integral type	<code>%d</code>
For an unsigned integral type	<code>%u</code>

对于浮点类型的转换，函数会判断浮点转换符标识，见表 12-14。

表 12-14 浮点类型转换字符型

状 态	stdio 等效
floatfield == ios::base::fixed	% f
(floatfield == ios::scientific) &&! uppercase	% e
(floatfield == ios::base::scientific)	% E
! uppercase	% g
otherwise	% G

对于转换过程中添加数据长度的标志，函数会判断长度修饰符，见表 12-15。

表 12-15 格式中的长度修饰符

类 型	长度修饰符	类 型	长度修饰符
long	l	long double	L
unsigned long	l	otherwise	none

转换符具有以下可选择的预先附属资格。详见表 12-16。

表 12-16 数值转换

类 型	状 态	stdio 等效
an integral type	flags&showpos	+
	flags&showbase	#
an floating-point type	flags&showpos	+
	flags&showpoint	#

为转换一个浮点类型，如果 (flags&fixed)! = 0 或 str.precision() > 0，那么 str.precision() 是指定的数值。对于空指针的转换，可以使用转换符%p。

阶段 1 最后的表达式包括 char 's。通过调用 printf (s, val) 可以打印该表达式。此处，s 是有上述决定的转换符。

阶段 2：除了十进制小数点 (.) 之外，使用刻画 use_facet < ctype < charT >> (loc).widen (c)，任何字符 c 均可被转换为一个 charT。通过 num_punct < charT > punct = use_facet < num_punct < charT >> (str.getloc ()), 本地变量 punct 可以被初始化。对于整型类型，punct.thousands_sep() 字符可以被插入至序列中，其值由 punct.do_grouping() 函数获取。十进制小数点符号 (.) 可以使用 punct.decimal_point() 函数替换。

阶段 3：本地变量可以被初始化。例如，

```
fmtflags adjustfield = (flags & (ios_base::adjustfield));
```

任何位置的垫衬可以由表 12-17 确定。

表 12-17 填充垫衬

状 态	Location
adjustfield == ios_base::left	pad after
adjustfield == ios_base::right	pad before
adjustfield == internal and a sign occurs in the representation	pad after the sign
adjust field == internal and representation after stage1 began with 0x or 0X	pad after x or X
otherwise	pad before

如果 `str.width()` 非零, 并且阶段 2 之后的序列中 `charT` 的数目小于 `str.width()`, 程序会自动添加填充字符至序列中, 使序列的长度自动增至规定长度 (`str.width()`)。

阶段 4: 阶段 3 末尾形成的 `charT` 类型的序列通过以下代码输出其内容。

```
* out++ = c;
```

函数 `put` 的使用方法:

```
iter_type put(iter_type out, ios_base& str, char_type fill, bool val) const;
```

函数的作用: 若 `(str.flags() & ios_base::boolalpha) == 0`, 则以下语句被执行:

```
out = do_out(out, str, fill, (int)val);
```

否则, 以下语句被执行, 并且将诸多 `s` 中的字符插入至流 `out.out` 中。

```
const num_punct<charT> & np = use_facet<num_punct<charT>>(loc);
```

```
string_type s = val ? np.true_name() : np.false_name();
```

下面使用例 12-8 来说明数值类的使用方法。

例 12-8

```
#include <locale>
#include <iostream>
#include <sstream>
using namespace std;
void main()
{
    locale loc("german_germany");
    basic_stringstream<char> psz, psz2;
    psz << "-1000,56";
    cout << psz.str() << endl;
    ios_base::iostate st = 0;
    long double fVal;
    cout << use_facet<num_punct<char>>(loc).thousands_sep() << endl;
    psz.imbue(loc);
    use_facet<num_get<char>>(loc).get(basic_istream<char>::_Iter(
    char>::_Iter(0), psz, st, fVal);
    if (st & ios_base::failbit)
        cout << "money_get() FAILED" << endl;
    else
        cout << "money_get() = " << fVal << endl;
    st = 0;
    cout << "The thousands separator is:" << use_facet<num_punct<char>>(loc).thousands_sep()
    << endl;
    psz2.imbue(loc);
    use_facet<num_put<char>>(loc).put(basic_ostream<char>::_Iter(
    ' , fVal = 1000.67);
    if (st & ios_base::failbit)
        cout << "num_put() FAILED" << endl;
    else
```

```
    cout << "num_put ( ) = " << psz2.rdbuf() ->str() << endl;  
}
```

例 12-8 的执行结果为:

```
-1000,56  
.  
money_get() = -1000.56  
The thousands separator is: .  
num_put() = 1.000, 67
```

下面使用例 12-9 来说明 `imbue()` 函数的使用方法。

例 12-9

```
#include <iostream>  
#include <locale>  
int main()  
{  
    using namespace std;  
    cout.imbue(locale("french_france"));  
    double x = 123.123456;  
    cout << x << endl;  
    locale native("");  
    cout.imbue(native);  
    double y=25.68;  
    cout << y << endl;  
}
```

例 12-9 的执行结果为:

```
123,123  
25.68
```

12.4.3 刻画 numeric_punctuation

1. 模板类 numpunct

```
namespace std{  
template<class charT> class numpunct : public locale::facet{  
public:  
    typedef charT;  
    typedef basic_string<charT> string_type;  
    explicit numpunct(size_t refs=0);  
    char_type decimal_point() const;  
    char_type thousands_sep() const;  
    string grouping() const;  
    string_type truename() const;  
    string_type falsename() const;  
    static locale::id id;  
protected:
```

```

~numpunct();

virtual char_type do_decimal_point() const;
virtual char_type do_thousands_sep() const;
virtual string do_grouping() const;
virtual string_type do_truename() const;
virtual string_type do_false() const;

};
}

```

模板 `numpunct` 指定了数值的标点符号。实例化类 `numpunct < wchar_t >` 和 `numpunct < char >` 可提供典型的“C”数值格式，即它们包含的信息等效于包含在类 `locale` 和那些使用 `widen()` 获取的宽字符。

数值格式的语法如下所示：`digit` 代表参数 `fmtflags` 确定的 `radix` 集合；使用刻画 `ctype < charT >` 确定的空格（`whitespace`）；并且 `thousands-sep` 和 `decimal-point` 是相应的 `numpunct < charT >` 成员的相应结果。整型值具有以下格式：

```

integer ::= [sign]units
sign ::= plusminus [whitespace]
plusminus ::= '+' | '-'
units ::= digits[thousands - sep units]
digit ::= digit[digits]

```

浮点类型的值具有特点：

```

floatval ::= [sign] units [decimal - point [digits]] [e[sign]digits] |[sign]decimal - point digits
[e[sign]digits]
e ::= 'e' | 'E'

```

`thousands - seps` 决定数字的数目是由 `do_grouping()` 决定的。对于数值分解，如果数字部分不包含 `thousands - separators`，那么没有分组常量被应用。

下面介绍模板类 `numpunct` 的成员函数。

```
char_type decimal_point() const;
```

函数返回值是 `do_decimal_point()`。`decimal_point()` 函数返回一个元素。该元素用作十进制小数点。

```
char_type thousands_sep() const;
```

函数返回值是一个实例化 `locale` 类元素。该元素用作一个千位分隔符。

```
string_type truename() const;
string_type falsename() const;
```

函数返回值是一个字符串。该元素可作为一个其值为 `true` 的文本表达式。

```
char_type do_decimal_point() const;
```

函数返回值是一个字符。该元素可用作十进制基数分隔符。必要的实例化返回 ‘.’ 或 L ‘.’。

```
char_type do_thousands_sep() const;
```

函数返回值是一个字符。该字符串用作数字组分隔符。必要的实例化返回 ‘,’ 或 L ‘,’。

```
string do_grouping()
```

函数返回值是一个字符串（`basic_string<char>vec`）。该字符串可用作一个整型值的向量。向量的每个元素代表数字数目。规则是：判断任何十进制小数点左边有多少个数字被划分为组。起始位置 0 是最右边的组。如果 `vec.size() <= i`，数字是同一组；如果 `(i < 0) || vec[i] <= 0 || vec[i] == CHAR_MAX`，数字组的大小是无限制的。

必要的实例化返回空字符串，表示没有分组。

```
string_type do_truename() const;
string_type do_falsename() const;
```

函数返回值是字符串表达式，其内容为“true”和“false”两种形式，或 L“true”和 L“false”。

2. 类模板 `num_punct_byname`

```
namespace std{
template <class charT > class num_punct_byname: public num_punct < charT > {
public:
    typedef charT char_type;
    typedef basic_string<charT> string_type;
    explicit num_punct_byname(const char* , size_t refs=0);
protected:
    ~num_punct_byname();
    virtual char_type do_decimal_point() const;
    virtual char_type do_thousands_sep() const;
    virtual string do_grouping() const;
    virtual string_type do_truename() const;
    virtual string_type do_falsename() const;
};
}
```

下面以例 12-10 来说明 `numeric_punctuation` 的使用方法。

例 12-10

```
#include <locale >
#include <iostream >
#include <sstream >
using namespace std;
int main()
{
    locale loc( "English" );
    const num_punct < char > & npunct = use_facet < num_punct < char >> (loc);
    cout << loc.name() << " truename() is called. " << npunct.truename() << endl;
    cout << loc.name() << " falsename() is called. " << npunct.falsename() << endl;
    const num_punct < char > & npunct2 = use_facet < num_punct < char >> (loc);
    //cout << npunct.grouping() << endl;
    for (unsigned int i=0; i < npunct.grouping().length(); i++)
    {
        cout << loc.name() << " international grouping:\n the " << i << "th group to the left
```

```

of the radix character "
    << "is of size " << (int)(npunct.grouping()[i]) << endl;
}
cout << loc.name() << " decimal point " << npunct.decimal_point() << endl;
cout << loc.name() << " thousands separator " << npunct.thousands_sep() << endl;
}

```

例 12-10 的执行结果为:

```

English_United States.1252 truename() is called. true
English_United States.1252 falsename() is called. false
English_United States.1252 international grouping:
the 0th group to the left of the radix character is of size 3
English_United States.1252 decimal point.
English_United States.1252 thousands separator,

```

12.4.4 模板类 collate

模板类 collate 的声明形式为:

```

namespace std{
    template <class charT> class collate: public locale::facet{
    public:
        typedef charT char_type;
        typedef basic_string<charT> string_type;
        explicit collate(size_t refs=0);
        int compare(const charT* low1, const charT* high1, const charT* low2, const charT* high2)
const;

        string_type transform(const charT* low, const charT* high) const;
        long hash(const charT* low, const charT* high) const;
        static locale::id id;
    protected:
        ~collate();
        virtual int do_compare(const charT* low1, const charT* high1,
const charT* low2, const charT* high2) const;
        virtual string_type do_transform(const charT* low, const charT* high) const;
        virtual long do_hash(const charT* low, const charT* high) const;
    };
}

```

类 collate < charT > 提供使用 collation 和 hashing 处理字符串的特性。本地成员函数模板 operator() 使用 collate 刻面, 目的是允许一个 locale 对象直接作为处理字符串的标准算法和容器的预测参数。实例化类 collate < char > 和 collate < wchar_t > 可以应用于字典式 (lexicographic) 排序。

1. 类 collate 的成员

```

int compare(const charT* low1, const charT* high1, const charT* low2, const charT*
high2) const;

```

函数的作用是：按照各自刻面定义的规则比较两个字符串是否相等。参数 (low1, high1) 是参与比较的第一个序列；参数 (low2, high2) 是参与比较的第二个序列。函数返回值一般有 3 种可能：-1、0 和 1。如果第一个序列小于第二个序列，函数返回 -1；如果第一个序列等于第二个序列，函数返回 0；如果第一个序列大于第二个序列，函数返回 1。

```
string_type transform(const charT* low, const charT* high) const;
```

函数的作用：一个字符序列从 locale 类对象到字符串 string 的转换可能被用于字典式和相同类 locale 的另一字符序列的比较。参数 low 和参数 high 代表已转换序列的起始字符。函数返回值包含了被转换的字符序列。

```
long hash(const charT* low, const charT* high) const;
```

hash() 函数会按照刻面的规则，判断序列的散列值。参数 low 和参数 high 代表输入序列的起始位置和结束位置。

2. 类 collate 的虚函数

```
int do_compare(const charT* low1, const charT* high1, const charT* low2, const charT* high2) const;
```

如果第一个字符串大于第二个字符串，函数返回 1；如果第一个字符串小于第二个字符串，函数返回 -1；如果第一个字符串相等第二个字符串，函数返回 0。实例化类 collate <char> 和 collate <wchar_t> 可以实施字典式排序。

```
string_type do_transform(const charT* low, const charT* high) const;
```

函数返回值类型是 basic_string <charT>。比较方式是字典式比较。

```
long do_hash(const charT* low, const charT* high) const;
```

函数返回值是整型值，其值等于调用 hash() 函数的结果。

3. 模板类 collate_byname

```
namespace std{
template <class charT>class collate_byname:public collate<charT>{
public:
    typedef basic_string<charT>string_type;
    explicit collate_byname(const char* , size_t refs=0);
protected:
    ~collate_byname();
    virtual int do_compare(const charT* low1, const charT* high1, const charT* low2, const charT* high2) const;
    virtual string_type do_transform(const charT* low, const charT* high) const;
    virtual long do_hash(const charT* low, const charT* high) const;
};
}
```

例 12-11

```
#include <locale>
#include <iostream>
#include <tchar.h>
using namespace std;
```

```

void main() {
    locale loc ( "German_germany" );
    _TCHAR * s1 = _T("Das ist wei\x00dfzz. ");
    _TCHAR * s2 = _T("Das ist weizzz. ");
    int result1 = use_facet<collate<_TCHAR>>(loc).compare(s1, & s1[_tcslen(s1) - 1 ], s2, &
s2[_tcslen(s2) - 1]);
    cout << result1 << endl;
    locale loc2 ("C");
    int result2 = use_facet<collate<_TCHAR>>(loc2).compare(s1, & s1[_tcslen(s1) - 1 ],s2,&
s2[_tcslen(s2) - 1]);
    cout << result2 << endl;
    long r1 = use_facet<collate<_TCHAR>>(loc).hash(s1, & s1[_tcslen(s1) - 1]);
    long r2 = use_facet<collate<_TCHAR>>(loc).hash(s2, & s2[_tcslen(s2) - 1]);
    cout << r1 << " " << r2 << endl;
    s2 = _T("zzz abc. ");
    collate<_TCHAR>::string_type r11; // OK for typedef?
    r11 = use_facet<collate<_TCHAR>>(loc).transform(s2, & s2[_tcslen(s2) - 1]);
    cout << r11 << endl;
    basic_string<_TCHAR> r22 = use_facet<collate<_TCHAR>>(loc).transform(s2, & s2[_tc-
slen(s2) - 1]);
    cout << r22 << endl; }

```

例 12-11 的执行效果如图 12-1 所示。



图 12-1 例 12-1 的执行效果

12.4.5 类 time

模板 `time_get<charT, InputIterator>` 和 `time_put<charT, OutputIterator>` 提供了日期和时间的格式和分解方法。所有类 `time_get` 和类 `time_put` 的成员函数均可应用于实例化。它们的成员使用 `ios_base&`, `ios_base::iostate&` 和 `fill` 参数, 使用类 `ctype` 决定格式的细节。

1. 类模板 `time_get<>`

```

namespace std{
public:
    enum dateorder{no_order, dmy, mdy, ymd, ydm};
    template <class charT, class inputIterator = istreambuf_iterator<charT>>
        class time_get:public locale::facet, public time_base{
public:

```

```

typedef charT char_type;
typedef InputIterator iter_type;
explicit time_get(size_t refs=0);
dateorder date_order() const{return do_date_order();}
iter_type get_time(iter_type s, iter_type end, ios_base& f, ios_base::iostate& err,
tm* t) const;
iter_type get_date(iter_type s, iter_type end, ios_base& f, ios_base::iostate& err,
tm* t) const;
iter_type get_weekday(iter_type s, iter_type end, ios_base& f, ios_base::iostate& err, tm
* t) const;
iter_type get_monthname(iter_type s, iter_type end, ios_base& f, ios_base::iostate& err,
tm* t) const;
iter_type get_year(iter_type s, iter_type end, ios_base& f, ios_base::iostate& err, tm* t)
const;

static locale::id id;
protected:
~time_get();
virtual dateorder do_date_order() const;
virtual iter_type do_get_time(iter_type s, iter_type end, ios_base& , ios_base::iostate& err, tm
* t) const;
virtual iter_type do_get_date(iter_type s, iter_type end, ios_base& , ios_base::iostate& err, tm
* t) const;
virtual iter_type do_get_weekday(iter_type s, iter_type end, ios_base& , ios_base::iostate& err,
tm* t) const;
virtual iter_type do_get_monthname(iter_type s, iter_type end, ios_base& , ios_base::iostate&
err, tm* t) const;
virtual iter_type do_get_year(iter_type s, iter_type end, ios_base& , ios_base::iostate& err, tm
* t) const;
}
}

```

`time_get()` 用于分解字符序列，从中抽出时间和日期的部件，并放入 `tm` 结构中。每一个 `get` 成员函数分解一种格式，作为相应的格式标识符，用于 `time_put < >::put()` 函数中。如果被分解的序列匹配正确的格式，相应的 `tm` 结构成员参数被设置成用于产生序列的值；否则，或者汇报一个错误，或者设置成一个不确定的值。

(1) `time_get < >` 的成员

```
dateorder date_order() const;
```

函数的作用是使用刻度返回日期的排序方式。

```
iter_type get_time(iter_type s, iter_type end, ios_base& str, ios_base::iostate& err, tm*
t) const;
```

函数的作用是将表示时间的字符串放入 `tm` 结构中。

```
iter_type get_date(iter_type s, iter_type end, ios_base& str, ios_base::iostate& err, tm*
t) const;
```

函数的作用是将表示日期的字符串放入 `tm` 结构中。

```
iter_type get_weekday(iter_type s, iter_type end, ios_base& str, ios_base::iostate& err,
tm* t) const;
```

函数的作用是将表示日期的字符串放入 `tm` 结构中。

```
iter_type get_year(iter_type s, iter_type end, ios_base& str, ios_base::iostate& err, tm*
t) const;
```

函数的作用是将表示年的字符串放入 `tm` 结构中。

(2) `time_get < >` 的虚函数

```
dateorder do_date_order(); const;
```

函数返回一个枚举值，该枚举值表明表达日期的组件的优先顺序。该日期格式主要包括日期、月和年。

```
iter_type do_get_time(iter_type s, iter_type end, ios_base* str, ios_base::iostate& err, tm* t)
const;
```

函数的作用是从 `[s, end]` 内获取字符串，并转换至 `tm` 结构中。函数返回迭代器指针。该指针指向最后一个字符前面。

```
iter_type do_get_date(iter_type s, iter_type end, ios_base& str, ios_base::iostate& err, tm*
t) const;
```

函数的作用是从指定的缓冲区抽取日期字符串，并转换至 `tm` 结构中。

```
iter_type do_get_weekday(iter_type s, iter_type end, ios_base& str, ios_base::iostate& err, tm
* t) const;
iter_type do_get_monthname(iter_type s, iter_type end, ios_base& str, ios_base::iostate& err,
tm* t) const;
```

函数的作用是从缓冲区中抽取合适的日期字符串，并转换至 `tm` 结构中。

```
iter_type do_get_year(iter_type s, iter_type end, ios_base& str, ios_base::iostate& err, tm*
t) const;
```

函数的作用是从缓冲区抽取合适的字符，并转换至 `tm` 结构中。

2. 类模板 `time_get_byname`

模板类 `time_get_byname` 的声明形式如下：

```
namespace std{
    template <class charT, class InputIterator = istreambuf_iterator < charT >>
        class time_get_byname: public time_get < charT, inputIterator > {
    public:
        typedef time_base::dateorder dateorder;
        typedef InputIterator iter_type;
        explicit time_get_byname(const char* str, size_t refs=0);
    protected:
        ~time_get_byname();
        virtual dateorder do_date_order() const;
        virtual iter_type do_get_time(iter_type s, iter_type end, ios_base& , ios_base::iostate&
err, tm* t) const;
        virtual iter_type do_get_date(iter_type s, iter_type end, ios_base& , ios_base::iost-
```

```

ate& err, tm* t) const;
    virtual iter_type do_get_weekday(iter_type s, iter_type end, ios_base&, ios_base::iostate& err,
tm* t) const;
    virtual iter_type do_get_monthname(iter_type s, iter_type end, ios_base&, ios_base::iostate& err,
tm* t) const;
    virtual iter_type do_get_year(iter_type s, iter_type end, ios_base&, ios_base::iostate& err, tm* t) const;
    }
}

```

3. 模板类 time_put

```

namespace std{
template<class charT, class OutputIterator = ostreambuf_iterator<charT>>
class time_put: public locale::facet{
public:
    typedef charT char_type;
    typedef OutputIterator iter_type;
    explicit time_put(size_t refs=0);
    iter_type put(iter_type s, ios_base& f, char_type fill, const tm* tmb, const charT* pattern,
const charT*
    pat_end) const;
    static locale::id id;
protected:
    ~time_put();
    virtual iter_type do_put(iter_type s, ios_base&, char_type fill, const tm* t, char format, char mod-
ifier) const;
};
}

```

(1) time_put 类成员

```

iter_type put(iter_type s, ios_base& str, char_type fill, const tm* t, const charT* pattern,
const charT* pat_end) const;

```

函数针对 [pattern, pat_end] 确定的字符序列，识别格式字符。格式字符序列的每一个字符被写入到序列 s (s 是迭代器) 中，并且识别出的每一个格式序列均会导致调用 do_put()。格式化元素和其他字符在顺序输出时均是交叉存取的。按格式显示序列时，会转换字符 c 为 char 类型的值，类似调用 narrow()。此处 ct 是一个对于 ctype <charT> 的引用，此引用是通过 str.getloc() 获取的。每个序列的第一个字符等于 “%”，并且格式标识符是通过 strftime() 函数定义的。如果没有可变字符输出，mod 是 0。对于每个已识别的有效格式序列，需要调用 do_put (s, str, fill, t, spec, mod)。

(2) time_put 虚函数

```

iter_type do_put(iter_type s, ios_base&, char_type fill, const tm* t, char format, char mod-
ifier) const;

```

函数的作用是将参数 `t` 的内容按格式化参数 `format` 处理, 并将之放在输出序列 `s` 中。格式是通过 `format` 和 `modifier` 来控制的。函数返回的迭代器指向最后一个字符之后。

4. 模板类 `time_put_byname`

模板类 `time_put_byname` 的声明形式为:

```
namespace std{
template <class charT, class OutputIterator = ostreambuf_iterator<charT>>
    class time_put_byname:public time_put<charT, OutputIterator>
{
public:
    typedef charT char_type;
    typedef OutputIterator iter_type;
    explicit time_put_byname(const char* , size_t refs=0);
protected:
    ~time_put_byname();
    virtual iter_type do_put(iter_type s, ios_base& , char_type, const tm* t, char format, char
modifier) const;
};
}
```

下面用例 12-12 来说明模板类 `time` 中 `time_base`、`time_get` 和 `time_put` 的使用方法。

例 12-12

```
#include <locale>
#include <iostream>
#include <sstream>
#include <time.h>
using namespace std;
void po(char * p)
{
    locale loc(p);
    time_get<char>::dateorder order = use_facet<time_get<char>>(loc).date_order();
    cout << loc.name();
    switch (order){
        case time_base::dmy: cout << "(day, month, year)" << endl;
            break;
        case time_base::mdy: cout << "(month, day, year)" << endl;
            break;
        case time_base::ydm: cout << "(year, day, month)" << endl;
            break;
        case time_base::ymd: cout << "(year, month, day)" << endl;
            break;
        case time_base::no_order: cout << "(no_order)" << endl;
            break;
    }
}
int main()
```

```
{
    locale loc;
    basic_stringstream< char > pszGetF, pszPutF;
    ios_base::iostate st = 0;
    struct tm t;
    memset(&t, 0, sizeof(struct tm));
    pszGetF << "July 4, 2000";
    pszGetF.imbue( loc );
    basic_istream<char>::_Iter i = use_facet<time_get<char>>
        (loc).get_date(basic_istream<char>::_Iter( pszGetF.rdbuf() ),
            basic_istream<char>::_Iter(0), pszGetF, st, &t);
    if ( st & ios_base::failbit )
        cout << "time_get(" << pszGetF.rdbuf() ->str() << ") FAILED on char: " << * i << endl;
    else
        cout << "time_get(" << pszGetF.rdbuf() ->str() << ") = "
            << "\ntm_sec: " << t.tm_sec
            << "\ntm_min: " << t.tm_min
            << "\ntm_hour: " << t.tm_hour
            << "\ntm_mday: " << t.tm_mday
            << "\ntm_mon: " << (t.tm_mon+1)
            << "\ntm_year: " << t.tm_year
            << "\ntm_wday: " << t.tm_wday
            << "\ntm_yday: " << t.tm_yday
            << "\ntm_isdst: " << t.tm_isdst
            << endl;
        pszGetF.clear();
        pszGetF << "2000";
    i = use_facet<time_get<char>>(loc).get_year(basic_istream<char>::_Iter( pszGetF.rdbuf() ), basic_istream<char>::_Iter(0), pszGetF, st, &t);
    if ( st & ios_base::failbit )
        cout << "time_get::get_year(" << pszGetF.rdbuf() ->str() << ") FAILED on char: " << * i
            << endl;
    else
        cout << "time_get::get_year(" << pszGetF.rdbuf() ->str() << ") = " << "\ntm_year: " <<
            t.tm_year << endl;
        pszGetF << "July";
        pszGetF.imbue( loc );
    i = use_facet<time_get<char>>(loc).get_monthname(basic_istream<char>::_Iter( pszGetF.rdbuf() ), basic_istream
        <char>::_Iter(0), pszGetF, st, &t);
    if ( st & ios_base::failbit )
        cout << "time_get(" << pszGetF.rdbuf() ->str() << ") FAILED on char: " << * i << endl;
    else
        cout << (t.tm_mon+1) << "月" << endl;
}
```

```

//time
    basic_stringstream<char> pszGetF2;
    st = 0;//非常重要的参数
    pszGetF << "11:13:20";
    pszGetF.imbue( loc );
i = use_facet<time_get<char>>(loc).get_time(basic_istream<char>::_Iter( pszGetF.rdbuf() ), basic_istream<char>::_Iter(0), pszGetF2, st, &t);
    if (st & ios_base::failbit)
        cout << "time_get::get_time(" << pszGetF2.rdbuf() ->str() << ") FAILED on char: " << *i << endl;
    else
        cout << "time_get::get_time(" << pszGetF2.rdbuf() ->str() << ") = " << "\ntm_sec: " << t.tm_sec << "\ntm_min: " << t.tm_min << "\ntm_hour: " << t.tm_hour << endl;
    basic_stringstream<char> pszGetF3;
    pszGetF3 << "Tuesday";
    pszGetF3.imbue( loc );
i = use_facet<time_get<char>>(loc).get_weekday(basic_istream<char>::_Iter( pszGetF3.rdbuf() ), basic_istream<char>::_Iter(0), pszGetF3, st, &t);
    if (st & ios_base::failbit)
        cout << "time_get::get_time(" << pszGetF3.rdbuf() ->str() << ") FAILED on char: " << *i << endl;
    else
        cout << "time_get::get_time(" << pszGetF3.rdbuf() ->str() << ") = " << "\ntm_weekday: " << t.tm_wday << endl;
    po( "C" );
    po( "german" );
    po( "English_Britain" );
}

```

例 12-12 的执行结果为:

```

time_get (July 4, 2000) =
tm_sec: 0
tm_min: 0
tm_hour: 0
tm_mday: 4
tm_mon: 7
tm_year: 100
tm_wday: 0
tm_yday: 0
tm_isdst: 0
time_get::get_year (July 4, 20002000) =
tm_year: 100

```

```

7月
time_get::get_time() =
tm_sec: 20
tm_min: 13
tm_hour: 11
time_get::get_time(Tuesday) =
tm_weekday: 2
C(year, month, day)
German_Germany.1252(day, month, year)
English_United Kingdom.1252(day, month, year)

```

12.4.6 模板类 monetary

模板类 `monetary` 主要用于钱币的格式处理。参数模板表明本地或国际钱币格式是否会被使用。其成员函数应用于类 `money_put` 和类 `money_get` 的实际使用过程中。其成员使用 `ios_base&`、`ios_base::iostate&` 和 `fill` 参数以及 `money_punct <>` 和 `ctype <>` 刻面判断格式细节。

1. 模板类 money_get

```

namespace std{
    template <class charT, class InputIterator = istreambuf_iterator <charT >> class money_get: public
locale::facet
    {
    public:
        typedef charT char_type;
        typedef InputIterator iter_type;
        typedef basic_string <charT > string_type;
        explicit money_get(size_t refs = 0);
        iter_type get(iter_type s, iter_type end, bool intl, ios_base& f, ios_base::iostate& err,
long double& units) const;
        iter_type get(iter_type s, iter_type end, bool intl, ios_base& f, ios_base::iostate& err,
long double& units) const;
        static locale::id id;
    protected:
        ~money_get();
        virtual iter_type do_get(iter_type, iter_type, bool ios_base&, ios_base::iostate& err, long
double& units) const;
        virtual iter_type do_get(iter_type, iter_type, bool ios_base&, ios_base::iostate& err, string
_type& digits) const;
    }
}

```

类 `money_get` 的成员函数:

```

iter_type get(iter_type s, iter_type end, bool intl, ios_base& f, ios_base::iostate& err, long
double& quant) const;
iter_type get(iter_type s, iter_type end, bool intl, ios_base& f, ios_base::iostate& err, string_
type& quant) const;

```

函数的作用是从表示钱币的字符串序列中抽取数值。

类 `money_get` 的虚函数:

```

iter_type do_get(iter_type s, iter_type end, bool intl, ios_base& str, ios_base::iostate& err,
long double& units) const;
iter_type do_get(iter_type s, iter_type end, bool intl, ios_base& str, ios_base::iostate& err,
string_type& digits) const;

```

函数的作用是从表示钱币的字符串序列中抽取数值。函数返回值是一个输入型迭代器。该迭代器表示钱币输入字段的第一个元素。

函数说明: 从序列 `s` 中读取字符, 并按照指定的格式, 分解并构造一个钱币值。

2. 模板类 `money_put`

模板类 `money_put` 的声明形式如下:

```

namespace std{
    template <class charT, class OutputIterator = ostreambuf_iterator<charT>>
        class money_put: public locale::facet{
    public:
        typedef charT char_type;
        typedef OutputIterator iter_type;
        typedef basic_string<charT> string_type;
        explicit money_put(size_t refs=0);
        iter_type put(iter_type s, bool intl, ios_base& f, char_type fill, long double units) const;
        iter_type put(iter_type s, bool intl, ios_base& f, char_type fill, const string_type& dig-
its) const;
        static locale::id id;
    protected:
        ~money_put();
        virtual iter_type do_put(iter_type, bool, ios_base& , char_type fill, long double units)
const;
        virtual iter_type do_put(iter_type, bool, ios_base& , char_type fill, const string_type& dig-
its) const;
    };
}

```

(1) 成员函数 `put`

```

iter_type put(iter_type s, bool intl, ios_base& f, char_type fill, long double quant) const;
iter_type put(iter_type s, bool intl, ios_base& f, char_type fill, const string_type& quant)
const;

```

函数的作用是将数字或字符串转换为字符序列。该字符序列用于表示钱币。

(2) 类 money_put 的虚函数

```

iter_type do_put(iter_type s, bool intl, ios_base& str, char_type fill, long double units)
const;
iter_type do_put(iter_type s, bool intl, ios_base& str, char_type fill, const string_type&
digits) const;

```

函数的作用是将按照使用刻画 money_punct < charT, intl > 指定格式，然后把字符写到 s 中。刻画 ctype < charT > 指定的字符映射，可从 locale 获取。参数单元被转换成宽字符序列。

```

ctype < charT > ct;
ct.widen(buf1, buf1 + sprint(buf1, "%.01f", units), buf2);

```

对于字符缓冲区 buf1 和 buf2，若数字中的第一个字符或缓冲区 buf2 等于 ct.widen(' - ')，则对应于某格式的可用模式是调用 neg_format() 函数的结果；否则，对应于某格式的可用模式是调用 pos_format() 的结果。数字字符被写入，其余的分隔符和十进制小数点相交错，按显示顺序逐一输出。在数字中，仅仅可选的前导减号和随后的数字字符被使用。任何尾部字符可以被忽略。



提示 当且仅当表达式 (str.flags() & str.showbase) 结果是非零时，货币标识可以被产生。
示 如果产生的字符数目小于 str.width() 函数的返回值，参数 fill 的副本被自动设置指定的宽度，作为垫衬。对于值 af 等于 (str.flags() & str.adjustfield)，如果 (af == str.internal) 是 true，那么参数 fill 代表的填充字符被放入 none 或 space 显示的格式模式。如果 (af == str.left) 是 true，那么参数 fill 代表的填充字符被放入其他字符之后；反之，它们被放入其他字符之前。

函数返回的迭代器指向最后一个字符之后。

3. 模板类 money_punct

模板类 money_punct 的声明形式为：

```

namespace std{
    class money_base{
        enum part {none, space, symbol, sign, value};
        struct pattern{char field[4];};
    };
    template <class charT, bool International > =false > class money_punct:public locale::facet, public money_base{
    public:
        typedef charT char_type;
        typedef basic_string<charT> string_type;
        explicit money_punct(size_t refs=0);
        charT decimal_point() const;
        charT thousands_sep() const;
        string grouping() const;
        string_type curr_symbol() const;
        string_type positive_sign() const;
        string_type negative_sign() const;

```

```

int frac_digits() const;
    pattern pos_format() const;
    pattern neg_format() const;
    static locale::id id;
    static const bool intl = International;
protected:
    ~moneypunct();
    virtual charT do_decimal_point() const;
    virtual charT do_thousands_sep() const;
    virtual string do_grouping() const;
    virtual string_type do_curr_symbol() const;
    virtual string_type do_positive_sign() const;
    virtual int do_frac_digits() const;
    virtual pattern do_pos_format() const;
    virtual pattern do_neg_format() const;
}
}

```

刻面 `moneypunct <>` 使用 `money_get <>` 和 `money_put <>` 定义了钱币的格式参数。钱币格式是 4 个组件的序列，由模式 `pattern` 确定。`static_cast <part >` 判断格式的第 `i` 个组件。在模式对象的域中，每个值或者空格或者 `none` 会正确显示一次。在 `none` 或空格显示位置，空格是被允许的。空格表示至少需要一个空格。标识 `symbol` 显示的地方，通过 `curr_symbol()` 返回的字符序列是允许的，也是必需的。在符号出现的位置，字符序列中的第一个符号是通过 `positive_sign()` 函数和 `negative_sign()` 返回的。关于符号序列的任何保留字符也是必需的。在值 `value` 显示的地方，绝对的钱币值是必需的。

钱币的数值是一个十进制数。

```
value:: = units[decimal - point [digits]] | decimal - point digits
```

如果 `frac_digits()` 函数返回一个正值或 `value:: = units`。否则，标识被定义为以下形式：

```
units:: = digits [thousands - sep units]
digits:: = adigit [digits]
```

标识 `adigit` 是 `ct.widen (c)` 任意值，其中 `c` 在 `0~9` 内。`ct` 是一个 `const ctype <charT > &` 类型的引用。标识 `thousands-sep` 是函数 `thousands_sep()` 返回的字符。使用的空格字符是 `t.widen (' ')` 的值。白空格 (White Space) 字符 `c` 是可以使函数 `is (space, c)` 返回 `true` 的字符。十进制小数点后面的数字数目是通过 `frac_digits()` 函数返回的精确值。

`thousands-separator` (分隔符) 的位置是由 `grouping()` 函数返回值决定的。

1) 类 `moneypunct` 的成员函数

```
charT decimal_point() const;
```

函数返回一个 `locale` 类对象元素。该元素用于表示十进制小数点。

```
charT thousands_sep() const;
```

函数返回一个 `locale` 类对象元素。该元素用作分隔符。

```
string grouping() const;
```

函数返回一个 locale 类对象元素。该元素用于判断数字是如何被分组的。

```
string_type curr_symbol() const;
```

函数返回一个 locale 类对象的元素。该元素用于表示钱币的标识。

```
string_type positive_sign() const;
```

函数返回一个 locale 对象的元素。该元素用于表示正号（“+”）。

```
string_type negative_sign() const;
```

函数返回一个 locale 类对象的元素。该元素用于表示负号（“-”）。

```
int frac_digits() const;
```

函数返回一个 locale 类对象的元素。该元素用于表示小数点右边显示的数字数目。

```
pattern pos_format() const;
```

函数返回指定的 locale 类对象规则。该规则用于格式化输出正的数量。

```
pattern neg_format() const;
```

函数返回指定的 locale 类对象规则。该规则用于格式化输出负的数量。
每个函数返回相应调用该虚函数的结果。

2) 类 money_punct 的虚函数

```
charT do_decimal_point() const;
```

函数返回值是基数（根）分隔符。该分隔符用于防止 do_frac_digits() 大于 0。

```
charT do_thousands_sep() const;
```

函数返回值是数字分组的分隔符。该分隔符用于以 do_grouping() 确定一个数字分组模式。

```
string do_grouping() const;
```

函数用于确定十进制小数点左边的数字的规则。

```
string_type do_curr_symbol() const;
```

函数返回用于表示钱币的元素序列。

```
string_type do_positive_sign() const;
```

```
string_type do_negative_sign() const;
```

函数返回表示“正号”和“负号”标识的元素序列。

```
int do_frac_digits() const;
```

函数返回表示小数点右边的数字数量。

```
pattern do_pos_format() const;
```

```
pattern do_neg_format() const;
```

函数返回一个用于格式化输出正数量或负数量的规则。

4. 模板类 money_punct_byname

模板类 money_punct_byname 的声明形式如下：

```
namespace std{
    template <class charT, bool intl = false > class money_punct_byname: public money_punct < charT,
intl >{
```

```

public:
    typedef money_base::pattern pattern;
    typedef basic_string<charT> string_type;
    explicit moneypunct_byname(const char* , size_t refs=0);
protected:
    ~moneypunct_byname();
    virtual charT do_decimal_point() const;
    virtual charT do_thousands_sep() const;
    virtual string do_grouping() const;
    virtual string_type do_curr_symbol() const;
    virtual string_type do_positive_sign() const;
    virtual string_type do_negative_sign() const;
    virtual int do_frac_digits() const;
    virtual pattern do_pos_format() const;
    virtual pattern do_neg_format() const;
};
}

```

例 12-13

```

#include <locale>
#include <iostream>
#include <sstream>
using namespace std;
int main()
{
    locale loc( "german_germany" );
    basic_stringstream< char > psz;
    psz << use_facet<moneypunct<char, 1>>(loc).curr_symbol() << "-1.000,56";
    basic_stringstream< char > psz2;
    psz2 << "-100056" << use_facet<moneypunct<char, 1>>(loc).curr_symbol();
    ios_base::iostate st = 0;
    long double fVal;
    psz.flags( psz.flags() | ios_base::showbase );
    psz.imbue(loc);
    use_facet< money_get< char >>( loc ).get( basic_istream<char>::_Iter( psz.rdbuf() ),
        basic_istream<char>::_Iter( 0 ), true, psz, st, fVal );
    if ( st & ios_base::failbit )
        cout << "money_get(" << psz.str() << ", intl = 1) FAILED" << endl;
    else
        cout << "money_get(" << psz.str() << ", intl = 1) = " << fVal/100.0 << endl;
    use_facet< money_get< char >>( loc ).
    get( basic_istream< char >::_Iter( psz2.rdbuf() ), basic_istream< char >::_Iter( 0 ), false, psz2,
    st, fVal );
    if ( st & ios_base::failbit )
        cout << "money_get(" << psz2.str() << ", intl = 0) FAILED" << endl;
    else

```

```

    cout << "money_get(" << psz2.str() << ", intl = 0) = " << fVal/100.0 << endl;
    locale loc2( "english_canada" );
    // basic_stringstream<char> psz2;
    st = 0;
    psz2.imbue( loc );
    psz2.flags( psz2.flags() | ios_base::showbase ); // force the printing of the currency
symbol
    use_facet < money_put < char >> (loc).put(basic_ostream<char>::_Iter( psz2.rdbuf() ), true,
psz2, st, 100012);
    if (st & ios_base::failbit)
        cout << "money_put() FAILED" << endl;
    else
        cout << "money_put() = \"" << psz2.rdbuf() ->str() << "\"" << endl;
    st=0;
};

```

例 12-13 的执行结果为：

```

money_get (EUR-1.000, 56, intl = 1) = -1000.56
money_get (-100056EUR, intl = 0) = -1000.56
money_put() = " -100056EUREUR1.000, 12"

```

下面用例 12-14 来说明类 `money_punct` 的使用方法。

例 12-14

```

#include <locale>
#include <iostream>
#include <sstream>
using namespace std;
int main()
{
    locale loc( "german_germany" );
    const money_punct < char, true > & mpunct = use_facet < money_punct < char, true >>
(loc);
    cout << loc.name() << " international currency symbol " << mpunct.curr_symbol() << endl;
    const money_punct < char, false > & mpunct2 = use_facet < money_punct < char, false >>
(loc);
    cout << loc.name() << " domestic currency symbol " << mpunct2.curr_symbol() << endl;
    const money_punct < char, true > & mpunct3 = use_facet < money_punct < char, true >>
(loc);
    cout << loc.name() << " international decimal point " << mpunct3.decimal_point() << endl;
    const money_punct < char, true > & mpunct4 = use_facet < money_punct < char, false >> (loc);
    cout << loc.name() << " domestic decimal point " << mpunct4.decimal_point() << endl;
    for (unsigned int i = 0; i < mpunct.grouping().length(); i++)
    {

```

```

    cout << loc.name() << " international grouping:\n the " << i << "th group to the left of the radix character "
        << "is of size " << (int)(mpunct.grouping() [i]) << endl;
}
cout << loc.name() << " international frac_digits\n to the right"
    << " of the radix character: " << mpunct.frac_digits() << endl;

const moneypunct <char, false> & mpunct5 = use_facet <moneypunct <char, false>> (loc);
for (unsigned int i = 0; i < mpunct5.grouping().length(); i++)
{
    cout << loc.name() << " domestic grouping:\n the " << i << "th group to the left of the radix character "
        << "is of size " << (int)(mpunct5.grouping() [i]) << endl;
}

cout << loc.name() << " domestic frac_digits\n to the right"
    << " of the radix character: " << mpunct5.frac_digits() << endl << endl;

const moneypunct <char, true> & mpunct6 = use_facet <moneypunct <char, true>> (loc);
cout << loc.name() << " international negative number format: " << mpunct6.neg_format().field
[0]
<< mpunct6.neg_format().field[1]
<< mpunct6.neg_format().field[2]
<< mpunct6.neg_format().field[3] << endl;
const moneypunct <char, false> & mpunct7 = use_facet <moneypunct <char, false>> (loc);
cout << loc.name() << " domestic negative number format: "
<< mpunct7.neg_format().field[0]
<< mpunct7.neg_format().field[1]
<< mpunct7.neg_format().field[2]
<< mpunct7.neg_format().field[3] << endl;

cout << loc.name() << " international negative sign: " << mpunct7.negative_sign() << endl;

    const moneypunct <char, true> & mpunct8 = use_facet <moneypunct <char, true>> (loc);
    cout << loc.name() << " international positive number format: "
<< mpunct8.pos_format().field[0]
<< mpunct8.pos_format().field[1]
<< mpunct8.pos_format().field[2]
<< mpunct8.pos_format().field[3] << endl;

const moneypunct <char, false> & mpunct9 = use_facet <moneypunct <char, false>> (loc);
cout << loc.name() << " domestic positive number format: "
<< mpunct9.pos_format().field[0]

```

```

<< mpunct9.pos_format().field[1]
<< mpunct9.pos_format().field[2]
<< mpunct9.pos_format().field[3] << endl;
cout << loc.name() << " international positive sign:"
    << mpunct9.positive_sign() << endl;

cout << loc.name() << " international thousands separator: "
    << mpunct9.thousands_sep() << endl;
}

```

例 12-14 的执行结果为：

```

German_Germany.1252 international currency symbol EUR
German_Germany.1252 domestic currency symbol €
German_Germany.1252 international decimal point ,
German_Germany.1252 international grouping:
    the 0th group to the left of the radix character is of size 3
German_Germany.1252 international frac_digits
    to the right of the radix character: 2
German_Germany.1252 domestic grouping:
    the 0th group to the left of the radix character is of size 3
German_Germany.1252 domestic frac_digits
    to the right of the radix character: 2
German_Germany.1252 international negative number format: $ +xv
German_Germany.1252 domestic negative number format: +v $
German_Germany.1252 international negative sign: -
German_Germany.1252 international positive number format: $ +xv
German_Germany.1252 domestic positive number format: +v $
German_Germany.1252 international positive sign:
German_Germany.1252 international thousands separator:.

```

12.4.7 类 message retrieval

类 `message < charT >` 用于实现从类 `message` 中返回字符串。模板类 `messages` 用于描述一个对象。该对象服务于一个 `locale` 剖面，用以从给定的具有国际信息 `locale` 对象中返回 `localized` 信息。当前，当 `message` 类被实现时，并没有信息。

1. 类模版 `messages`

类 `message` 的声明形式为：

```

namespace std{
    class message_base{
        typedef int catalog;
    }
}

```

```

template <class charT> class message: public locale::facet , public message_base{
public:
    typedef charT char_type;
    typedef basic_string<charT> string_type;
    explicit message(size_t refs=0);
    catalog open(const basic_string<char>& fn, const locale& ) const;
    string_type get(catalog c int set, int msgid, const string_type& default) const;
    void close(catalog c) const;
    static locale::id id;
protected:
    ~messages();
    virtual catalog do_open(const basic_string<char>& , const locale& ) const;
    virtual string_type do_get(catalog, int set, int msgid, const string_type& default) const;
    virtual void do_close(catalog) const;
};
}

```

类型 `message_base::catalog` 的值是可用作成员 `get` 和 `close` 的参数，可以通过调用成员 `open()` 获取。

(1) 成员函数

```
catalog open(const basic_string<char>& name, const locale& loc) const;
```

函数返回 `do_open` (`name`, `loc`)。函数的作用是打开 `message catalog`。参数 `name` 表示被搜索的 `catalog` 的名称；参数 `loc` 代表在 `catalog` 中正在被搜索的 `locale`。

```
string_type get(catalog cat, int set, int msgid, const string_type& default) const;
```

函数返回 `message catalog`。

```
void close(catalog cat) const;
```

函数的作用是关闭 `message catalog`。

(2) 类 `message` 的虚函数

```
catalog do_open(const basic_string<char>& name ,const locale& loc ) const;
```

函数返回值是 1 个值。该值可能会传递给 `get()` 函数，从而可以返回该值。按照已定义的映射从 `message catalog` 中识别字符串名称。函数执行的结果可以被使用，直到将其传递给 `close()` 函数。`close()` 函数会将其关闭。

若没有 `catalog` 被打开，则返回值会小于 0。

当返回信息时，参数 `loc` 被用于字符集编码转换。

```
string_type do_get(catalog cat, int set, int msgid, const string_type& default) const;
```

`catalog` 类对象 `cat` 是从 `open()` 获取的，并且该 `catalog` 是没有关闭的。

通过参数 `set` 识别的信息，`msgid`，和 `default`，按照已定义的可实施映射。如果没有找到该信息，返回 `default`。

```
void do_close(catalog cat) const;
```

catalog 类对象 cat 可以通过调用 open() 函数获取, 并且该 catalog 是没有关闭的。其作用主要是释放和 cat 相关的不确定的资源。

2. 类模板 message_byname

类模板 message_byname 的声明形式如下:

```
namespace std{
    template <class charT> class message_byname: public message <charT> {
        public:
            typedef message_base::catalog catalog;
            typedef basic_string<charT> string_type;
            explicit message_byname(const char* , size_t refs=0);
        protected:
            ~message_byname();
            virtual catalog do_open(const basic_string<char>&, const locale&)const;
            virtual string_type do_get(catalog , int set, int magid, const string_type& default)
const;
            virtual void do_close(catalog) const;
        }
    }
```

派生而来的模板类 message_byname 用于描述一个对象。该对象服务于给定 locale 类的 message 剖面, 并返回本地信息。

例 12-15

```
#include <locale>
#include <iostream>
#include <sstream>
#include <string>
using namespace std;
void main()
{
    basic_stringstream<char> pszGetF;
    ios_base::iostate st = 0;
    locale loc("german_germany");
    const messages<char> &m = use_facet<messages<char>>(loc);
    int pm = m.open("abcd", loc);
    if(pm < 0)
    {
        cout << "can't open the message. " << endl;
        exit(1);
    }
    cout << "pm = " << pm << endl;
    int _Set = 1;
    int _Message = 1;
    basic_string<char> df("error");
    string dr;
```

```

dr = m.get(pm, _Set, _Message, df);
    cout << dr << endl;
    cout << df << endl;
    cin >> pm;
}

```

例 12-15 的执行结果为:

can't open the message.

出现上述结果的原因在于如下代码:

```

catalog __CLR_OR_THIS_CALL open(const string& _Catname, const locale& _Loc) const
{ // open catalog
    return (do_open(_Catname, _Loc));
}

virtual catalog __CLR_OR_THIS_CALL do_open(const string& , const locale& ) const
{ // open catalog (do nothing)
    return (-1);
}

```

在 Windows XP 环境中安装的 Visual Studio 2008 开发环境中, 没有对 `open()` 函数和 `do_open()` 进行定义, 而仅仅在 `do_open()` 函数中直接返回 `-1`。

12.4.8 Program-defined facets

一个 C++ 程序可能定义诸多刻面。这些刻面附加于 `locale` 类对象, 作为嵌入式刻面使用。若创建一个刻面新接口, C++ 程序需要从类 `locale::facet` 派生一个新的接口类。该类包含静态成员 `static locale::id id`。

类 `locale` 的成员函数的模板会比较它的类型和存储类别。传统的全局本地代是非常简单的, 详见例 12-16。

例 12-16

```

#include <iostream>
#include <locale>
using namespace std;
void main()
{
    cin.imbue(locale("German_Germany"));
    cout.imbue(locale::classic());
    double f;
    cin >> f; //输入按德国形式
    cout << f << endl; //输出按经典(本地)形式
    cout << "cin.fail() : " << cin.failbit << endl;
    return;
}

```

例 12-16 的执行结果为:

```
76.82
cin.fail():0
```

例 12-17

```
#include <iostream>
#include <locale>
#include <string>
using namespace std;
namespace My{
    using namespace std;
    typedef numpunct_byname<char> cnumpunct;
    class BoolNames: public cnumpunct{
    protected:
        string do_truename() const{return "Oui Oui!";}
        string do_falsename() const{return "Mais Non!";}
        ~BoolNames(){};
    public:
        BoolNames(const char* name):cnumpunct(name){}
    };
}
int main(int argc, char* * argv)
{
    locale loc(locale("chinese"),new My::BoolNames(" "));
    cout.imbue(loc);
    cout<<boolalpha<<"today? "<<"Monday."<<endl;
    return 0;
}
```

例 12-17 的执行结果为：
today? Monday.

12.4.9 C 库 locale

头文件 <locale> 中包含了以下内容。这些内容和头文件 <locale.h> 中的内容是一致的，详见表 12-18。

表 12-18 头文件 <locale> 概要

类 型	Name (s)		
宏	LC_ALL	LC_COLLATE	LC_CTYPE
	LC_MONETARY	LC_NUMERIC	LC_TIME
	NULL		
结构	lconv		
函数	localeconv	setlocale	

12.5 细述使用刻面

类 `locale` 包含了诸多刻面 (facet)。所有类 `locale` 必须包容某些标准刻面。C++ 标准程序库实例版本均在 `locale` 中附加了部分其他刻面。此外, 用户还可以自己定义刻面或替换标准刻面。

对于任何类别, 如要作为刻面, 需要满足以下两个条件:

1) 类别 `F` 以 `public` 形式派生于 `locale::facet` 类别。基类主要定义机制, 这些机制是针对 `locale` 对象内部的引用计数器运用。对于 `copy` 构造函数和 `assignment()` 操作符声明为 `private`, 禁止外界复制 `facet`, 同时禁止对刻面的赋值。

2) 类别 `F` 必须拥有一个型别为 `locale::id` 的公共静态成员, 名为 `id`。此成员用于“以某个 `facet` 型别为索引, 搜寻 `locale` 内的一个 `facet`”。之所以要以型别作为索引, 主要是要保证型别的安全。内部是普通容器, 以整数索引管理 `facet`。

标准刻面不仅需要遵循上述要求, 还要遵循部分特殊实例化原则。

此外, 标准刻面还应遵循以下原则:

1) 所有成员函数均声明为 `const` 类型。因为 `use_facet()` 返回一个引用指向 (`const facet`)。在程序中, 对于某个成员函数不能声明为 `const`, 否则会失去被调用的可能。

2) 对于 `public` 函数, 不是虚函数。将调用操作委托给一个“保护型 (protected)”虚函数, 后者的名称类似对应的公用函数, 仅仅在前面加上 `do_`。`num_punct::truename()` 会调用 `num_punct::do_truename()`。

对于标准刻面的描述仅限于公用函数 (`public ()`)。如果需要改变某个 `facet`, 应该修改其对应的保护型成员函数。大部分标准刻面均定义了“`_byname`”版本。该版本均是派生于标准刻面, 被用于“根据相应 `locale` 名字”生成相应的 `facet` 实例。类 `num_punct_byname` 多用于针对一个具体的 `locale` 产生一个新的 `num_punct` 刻面。

```
std::num_punct_byname ("de_DE");
```

诸多“`_byname`”类别在类 `locale` 的“以名称为参数”的构造函数中被作为内部运用。每个标准刻面均有一个对应的名称, 相应的 `_byname` 类别可用于构造该刻面的实例。

12.5.1 数值的格式化

数值的格式化是指数值格式在数值的内部表述和文本表述之间进行转化。`IOstream` 操作符将实际转换工作委托给 `locale::numeric` 类型中的诸多刻面完成。通常是由以下 3 个刻面组成的。

- 1) `num_punct`。该刻面用于处理数值格式化及数值解析相关的标点符号。
- 2) `num_put`。该刻面用于处理数值格式化。
- 3) `num_get`。该刻面用于处理数值解析。

下面详细讲述“以 `num_put` 进行数值格式化”和“以 `num_get` 进行相应字符串解析”的相关知识。`num_punct` 刻面提供了 `stream` 接口未能直接支持的部分灵活性。

1. 数值点符号

`num_punct` 刻面用于控制小数点、可有可无的千位分隔符和布尔值的文本表示符号。其

成员见表 12-19。

表 12-19 num_punct 刻面的成员

表 达 式	功 能
np. decimal_point()	返回一个字符，用于表示小数点
np. thousands_sep()	返回一个字符，用于表示千位分隔符
np. grouping()	返回一个 string，描述千位分隔符的设置位置
np. truenamename()	返回 true 的文本表示
np. falsenamename()	返回 false 的文本表示

num_punct 刻面是以字符型别 charT 作为模板参数的。decimal_point() 和 thousands_sep() 返回的字符型别均是 charT；truenamename() 和 falsenamename() 返回一个 basic_string < charT >。C++ 标准要求刻面 num_punct < char > 和 num_punct < char_t > 的具体实例化必须存在。

如果没有分隔符，长数字很难阅读。数值格式化和数值解析的标准刻面也要支持千位分隔符。通常情况下，阿拉伯数字是 3 个一组，组与组之间用逗号分隔。

例如，我们通常将“一百万”写作 1, 000, 000。

但世界上很多地区不会这样写。例如，在德国不使用逗号 (,)，而是使用“点”号 (.)。例如，“一百万”在德国写作 1.000.000。而且在有些国家，数字不是以 3 个一组的方式分组的。例如，“一百万”在尼泊尔写作 10.00.000

每组阿拉伯数字的个数均不尽相同。差异是必须由 thousands_sep() 决定的。函数 grouping() 返回的字符串即在此处。索引 i 处的数值代表第 i 组的数字个数；从右边数起，索引计数从 0 开始。若字符中的字符数量少于每个群组中的字符数量，最后确定的那组字符的数量将重复。若需要创造无限大组，应当使用 numeric_limit < char > ::max() 的返回值。若不分组，则会给出一个空字符串。详见表 12-20。

表 12-20 一百万的各种表示方法

字 符 串	结 果	字 符 串	结 果
{0} 或 “ ”	1000000	{3, 2, 3, 0} 或 “、3、2、3”	10, 00, 000
{3, 0} 或 “\3”	1, 000, 000	{2, CHAR_MAX, 0}	10000, 00

如果直接使用数字可能是不正确的。例如，字符串“2”指定的是一个 50 个阿拉伯数字的数字分组，因为在 ASCII 编码中，字符“2”即是整数 50。

2. 数值的格式化

num_put 刻面用于对数值的“表述文本”进行格式化。它是一个模板类，包含两个模板参数：字符型别 charT 和 output 迭代器型别 OutIt。产生的字符会写入迭代器中所指的位置。output 迭代器的默认型别是 ostreambuf_iterator < charT >。num_put 刻面提供多参数的 put() 函数，期间只有最后一个参数是不同的。可以使用的刻面主要包括以下一些：

```
std::locale loc;
OutIt      co = ...;
std::ios_base&  fmt = ...;
charT      fill = ...;
```

```
T          value = ...;
const std::num_put<charT, OutIt >& np = std::use_facet<std::num_put<charT, OutIt >(loc);
np.put(to, fmt, fill, value);
```

上述语句是利用型别为 `charT` 的字符，在 `output` 迭代器 `to` 所指位置处填写 `value` 的文本表述式。确切格式由 `fmt` 的格式化标识确认。`fill` 被用于作为填充字符，`put()` 函数返回一个迭代器。该迭代器指向最后一个被写出的字符的下一位置。

`num_put` 剖面提供一系列成员函数，分别接受型别为 `bool`、`long`、`unsigned long`、`double`、`long double` 及 `void*` 的对象作为最后一个参数。`short` 和 `int` 等内建型别的数值可以自动晋升为相应型别，即没有提供针对 `short` 和 `int` 的成员函数。

C++ 标准要求每个类 `locale` 内必须有 `num_put<char>` 和 `num_put<wchar_t>` 两个具体实例化对象。另外，C++ 标准程序库还支持具备以下特性的所有实例化对象：以字符型别作为第一个模板参数；以输出型迭代器型别作为第二个模板参数。标准规格不要求每个 `locale` 均储存具体的实例化对象，会导致无穷的剖面。

3. 数值解析

`num_get` 剖面用于解析数值的文本表述。与 `num_put` 剖面相比，`num_get()` 剖面具有两个模板参数：字符型别 `charT` 和输入型迭代器型别 `InIt`。`num_get()` 剖面提供一系列的 `get()` 函数，期间只有最后一个参数是不同的。数值的格式多数是由 `fmt` 确定的。如果失败，`err` 会被设置为 `ios_base::failbit`；反之，`err` 会被设置为 `ios_base::goodbit`。所有数值将被放入 `value` 中。`value` 的原有数值在解析后才被修改。若全部字符序列解析完毕，`get()` 函数会返回第二个参数；若全部字符序列无法解析完毕，则会返回一个迭代器，指向“无法被解析为数值的那一部分”的第一个字符。

`num_get` 剖面支持“用于读取型别为 `bool`、`long`、`unsigned short`、`unsigned int`、`unsigned long`、`float`、`double`、`long double` 和 `void*` 等对象”的函数。某些型别在 `num_put` 中没有对应的函数，写一个 `unsigned short` 值和写一个 `unsigned int` 值需要提升转型为 `unsigned long` 是相同的。读取一个 `unsigned long` 型别的值再将其转换为 `unsigned short` 型别，和直接读取 `unsigned short` 的结果是不相同的。

C++ 标准要求每个类 `locale` 必须有 `num_get<char>` 和 `num_get<wchar_t>` 两个具体实例化对象。C++ 标准程序库还支持具备以下特性的所有实例化对象：以字符型别作为第一个模板参数，以输入型迭代器型别作为第二个模板参数。与 `num_put` 剖面相同，不是所有被支持的具体实例化对象均必须储存在 `locale` 对象中。

12.5.2 时间/日期的格式化

时间/日期的解析和格式化工作是由 `time` 类型中的两个剖面完成的。这两个剖面分别是 `time_get` 和 `time_put`。其成员函数凭借操作型别为 `tm` 的对象完成该项工作。型别 `tm` 定义于头文件 `<ctime>` 中。`tm` 对象是不会被直接传递的，而是以其地址为参数进行传递的。

`time` 类型中两个剖面的行为均和 `strftime()` 函数有关系。该函数会根据一个含有转换规格的字符串，从一个 `tm` 对象中生成一个字符串。转换规则同样适用于 `time_put` 剖面。转换规则详见表 12-21。

表 12-21 函数 `strptime()` 的转换规则

规格符号	意 义	实 例
<code>%a</code>	星期缩写	Mon
<code>%A</code>	星期全名	Monday
<code>%b</code>	月份缩写	Jul
<code>%B</code>	月份全名	July
<code>%c</code>	locale 首选的日期和时间格式	Jul 12 21: 53: 22 1998
<code>%d</code>	日	12
<code>%H</code>	24 时制下的小时	21
<code>%I</code>	12 时制下的小时	9
<code>%j</code>	某年的第几天	193
<code>%m</code>	月份以数字表示	7
<code>%M</code>	分钟	53
<code>%p</code>	上午或下午	pm
<code>%S</code>	秒	22
<code>%U</code>	从第一个星期日算起的周数	28
<code>%W</code>	从第一个星期一算起的周数	28
<code>%w</code>	星期，以数字表示（星期天是 0）	0
<code>%x</code>	locale 首选的日期格式	Jul 12 1998
<code>%X</code>	locale 首选的时间格式	21: 53: 22
<code>%y</code>	两位数年份	98
<code>%Y</code>	完整年份	1998
<code>%Z</code>	时区	MEST
<code>%%</code>	字符“%”	%

当然，`strptime()` 函数产生的具体字符串要根据“C” locale 来确定。

1. 时间和日期的解析

`time_get` 刻画是一个模板，以一个字符型别 `charT` 和一个输入型迭代器型别作为模板参数。该输入型迭代器的默认型别为 `istreambuf_iterator<charT>`。除了 `date_order()` 之外，所有成员均对字符串进行解析，并将结果存储于参数 `t` 所指的 `tm` 对象中。若字符串不能被正确解析，就不汇报错误；反之，就在 `tm` 对象中存储一个未定值。程序产生的时间会被正确地解析，而用户输入的时间就未必了。通过参数 `fmt` 可以决定解析过程中使用哪些刻画。`fmt` 的其他标识对解析过程的影响目前没有明确规定。

全部函数均返回一个迭代器。该迭代器指向最后一个读取字符的下一位置。当解析结束或发生错误时，解析动作需要立刻停止。读取星期和月份时，函数可以使用缩写，也可使用全名。若缩写后面跟着字母，且恰好是全名中的某个有效字母，则函数试图读取全名。若读取失败，将无视“缩写名称曾解析成功过”的事实，声明解析失败。

`date_order()` 返回日期字符串中的年月日顺序。对日期来说，这是有必要的。因为仅从字符串表示式来看，很难搞清楚顺序。例如，2003 年 2 月的第一天可以写成 `3/2/1`，也可写成 `1/2/3`。`time_get` 刻面的基类 `time_base` 针对所有可能的年月日次序定义了枚举型别 `dateorder`。

C++ 标准要求每个类 locale 必须有 `time_get<char>` 和 `time_get<wchar_t>` 两个具体实例化

对象。此外，C++ 标准程序库要求支持以下形式的具体实体：以 `char` 或 `wchar_t` 作为第一个模板参数，以相应的输入型迭代器作为第二个模板参数。`time_get` 刻面的成员函数详见表 12-22。

表 12-22 `time_get` 刻面的成员函数

表 达 式	意 义
<code>get_time (beg, end, fmt, err, t)</code>	解析 <code>beg</code> 和 <code>end</code> 之间的字符串，其格式与 <code>strftime ()</code> 以 <code>%X</code> 产生者一致
<code>get_date (beg, end, fmt, err, t)</code>	解析 <code>beg</code> 和 <code>end</code> 之间的字符串，其格式与 <code>strftime()</code> 以 <code>%x</code> 产生者一致
<code>get_weekday (beg, end, fmt, err, t)</code>	解析 <code>beg</code> 和 <code>end</code> 之间用以表示星期的字符串
<code>get_monthname (beg, end, fmt, err, t)</code>	解析 <code>beg</code> 和 <code>end</code> 之间用以表示月份的字符串
<code>get_year (beg, end, fmt, err, t)</code>	解析 <code>beg</code> 和 <code>end</code> 之间用以表示年份的字符串
<code>date_order()</code>	返回 <code>facet</code> 所用的年月日次序

`dateorder` 的枚举值详见表 12-23。

表 12-23 `dateorder` 的枚举值

值	意 义	值	意 义
<code>no_order</code>	无特别次序	<code>Ymd</code>	次序为年、月、日
<code>Dmy</code>	次序为日、月、年	<code>Ydm</code>	次序为年、日、月
<code>Mdy</code>	次序为月、日、年		

2. 时间和日期的格式化

`time_get` 刻面用于格式化时间和日期。它是一个模板，接受一个字符型别 `charT` 作为模板参数，另一个可有可无的模板参数是输出型迭代器型别，即 `Outit`。后者默认为 `ostreambuf_iterator`。`time_put` 刻面定义了两个 `put ()` 函数，均被用于将“储存于 `tm` 对象中的日期信息”转化为一个字符序列，并写至输出型迭代器中。表 12-24 列出了 `time_put` 刻面的两个成员函数。

表 12-24 `time_put` 刻面的成员函数

表 达 式	意 义
<code>put (to, fmt, fill, t, cbegin, cend)</code>	根据字符串 <code>[cbegin; cend]</code> 进行转换
<code>put (to, fmt, fill, t, cvt, mod)</code>	运用转换规则 <code>cvt</code> 进行转换

这两个函数的结果均可写至输出型迭代器中。该输出型迭代器返回最后一个被写出的字符的下一位置。参数 `fmt` 的型别是 `ios_base`，用于存取其他刻面及可能有的附加信息。若需要填充字符，则可使用 `fill`。参数 `t` 指向一个 `tm` 对象，其中存放即将被格式化的日期。

表 12-24 中的第二个 `put 函数()` 的后两个参数是字符，此版本将 `t` 所指向的 `tm` 对象格式化，并将参数 `cvt` 视为转换指示符传给 `strftime()`。`put()` 函数仅能执行一次转换，即由字符 `cvt` 指示的转换。当有转换指示符被发现时，函数被另一个 `put()` 函数调用。例如，以“X”作为转换指示符，会导致储存于 `*t` 内的时间信息被写至输出型迭代器。C++ 标准没有对参数 `mod` 的意义进行强制定义。其主要目的是作为修饰符，即针对 `strftime()` 函数的多个实作版本之间的转换进行修饰。

第一种形式的 `put()` 函数接受由 `[cbeg, cend]` 定义的字符串，其转换为行为和 `strftime()` 非常相似，即扫描字符串，并将任何“不隶属转换指示符”的字符直接写至输出型迭代器。若遇到由“%”引导的转换指示符，则会从中提炼修饰符和转换指示符。函数将转换指示符和修饰符作为最后两个参数，并调用另一版本的 `put()` 函数，直到处理完一个转换指示符后，`put()` 继续扫描字符串。

此刻面还提供了一个非虚拟成员函数——“以一个字符串作为转换指示符”的 `put()` 函数。`time_put` 派生类别无法对此函数进行改写，只能改写其他 `put()` 函数。

C++ 标准要求每个类 `locale` 中都必须有两个实例化对象：`time_put < char >` 和 `time_put < wchar_t >`。另外，C++ 标准程序库还支持所有“第一个模板参数为 `char` 或 `wchar_t`，第二个模板参数为相应之输入型迭代器”的 `time_put < >` 实例化对象。C++ 标准不能够保证支持以“`char` 和 `wchar_t` 以外的字符型别”作为第一个模板参数的实例化对象，也不能保证 `locale` 对象在默认情况下包含 `time_put < char >` 和 `time_put < wchar_t >` 之外的任何实例化对象。

12.5.3 货币符号的格式化

类 `monetary` 包含 `moneypunct`、`money_get` 和 `money_put3` 个刻面。其中 `moneypunct` 用于定义货币格式，另外两个刻面使用该格式信息解析货币值。

1. 货币符号

货币值在不同的场合中以不同的格式被打印出来。不同的文化社群所习惯的格式之间存在非常大的差异。例如货币符号的布置、负值或正值的记法、国家或国际货币符号的使用、千位分隔符等。为了具有必要的灵活性，格式细节被存放在 `moneypunct` 刻面中。

`moneypunct` 刻面是一个模板，接受一个字符型别的 `charT` 和一个默认值为 `false` 的布尔值。该布尔值表示将运用本地的或国际的货币符号。表 12-25 列出了 `moneypunct` 刻面的成员函数。

表 12-25 `moneypunct facet` 刻面的成员函数

表 达 式	意 义
<code>decimal_point()</code>	返回小数点表示字符
<code>thousands_sep()</code>	返回千位表示字符
<code>grouping()</code>	返回一个字符串，用于指定千位分隔符的布置格式
<code>curr_symbol()</code>	返回货币符号的字符串
<code>positive_sign()</code>	返回正号表示字符
<code>negative_sign()</code>	返回负号表示字符
<code>frac_digits</code>	返回小数的位数
<code>pos_format()</code>	返回非负数的格式
<code>neg_format()</code>	返回负数的格式

`moneypunct` 派生自类 `money_base`，此基类定义了一个名为 `part` 的枚举量，用以形成货币形式。此基类还定义了 `pattern` 型别，用来存储型别为 `part` 的 4 个值，进而形成一个样式，用以描述货币布局方式。表 12-26 列出了可以放进样式中的 5 种可能的 `parts`。

表 12-26 货币布局样式的局部内容

值	意 义	值	意 义
none	在此位置上可以出现空白, 但不强制要求	symbol	在此位置上需要一个货币符号
space	至少需要一个空白	value	在此位置上需要一个值
sign	在此位置上需要一个正负符号		

money_punct 刻面定义了两个函数, 用于返回样式: `neg_format()` 专门针对负值, `pos_format()` 专门针对非负值。对于一个样式, `sign`、`symbol` 和 `value` 是必要的。`none` 和 `space` 二者必须出现其一, 此时并不意味着会打印一个正号或负号。`parts` 所指之处要打印什么, 要根据刻面其他成员的返回值以及格式化函数传送过来的格式化标志决定。

在输出打印货币值时, 其格式是由 `value` 在样式中出现的位置决定的。货币值小数的位数由 `frac_digits()` 指定, 货币值的小数点表示字符由 `decimal_point()` 确定。

读取货币, 允许分隔千位符存在, 但不会强制要求。若确实存在, 需要根据 `grouping()` 检查“分组布局”的正确性。若 `grouping()` 为空, 则不允许出现千位分隔符。千位分隔符字符可由 `thousands_sep()` 返回。千位分隔符的布局规则和数值格式化的情形是一样的。打印货币值时, 要根据 `grouping()` 返回的字符串插入和打印千位分隔符字符。一旦所有其他解析工作完成, 便可检查出千位分隔符布局是否正确。

`space` 和 `none` 主要用于控制空白的布置。`space` 用于至少需要一个空白的位置。格式化时, 若格式标识中指定 `ios_base::internal`, 则填充字符会被插入到 `space` 或 `none` 部分。当然, 只有在指定的最小宽度没有使用其他字符时, 填充才会产生。“被用于作为空格符”的填充字符应该以参数形式传给格式化函数。若格式化的货币值不包含空白, `none` 可以被放在最后一个位置。`space` 和 `none` 不能出现在样式的第一部分。`space` 不能出现在样式的最后一部分。

货币值的正负符号可由一个以上的字符表示。例如, 某些场合用括号括起的一个值可表示为负值。样式中 `sign` 是正负号第一个字符出现的位置, 其余所有正负号相关字符均出现在其他组件之后。若正负号字符串为空, 则可不以任何符号指示正负。`positive_sign()` 和 `negative_sign()` 这两个函数用于决定正负号字符, `positive_sign()` 对应正号, `negative_sign()` 对应负号。

`symbol` 位置上会出现货币符号。在格式化过程和解析过程中 `ios_base::showbase` 标识被设立的情况下, 此符号才有效。货币符号由 `curr_symbol()` 函数返回的字符串确定, 此为当地符号——若第二个模板参数为 `false`, 便以它来表示货币; 否则, 将使用国际货币符号。

C++ 标准要求每个类 `locale` 中都必须有两个实例化对象: `money_punct < char >` 和 `money_punct < wchar_t >`。除此之外, C++ 标准程序库不支持其他任何形式的实例化对象。

2. 货币的格式化

`money_punct` 刻面可以用于格式化货币值。它是一个模板类, 接受字符型别 `charT` 作为第一个模板参数, 接受输出型迭代器型别 `OutIt` 作为第二个模板参数。输出型迭代器的默认类型为 `ostreambuf_iterator < charT >`。货币样式的例子见表 12-27。

成员 `put()` 函数有两种形式, 用以根据 `money_punct` 刻面指定的格式产生一个字符序列。被格式化的数值或以 `long double` 传递, 或以 `basic_string < charT >` 传递。

表 12-27 货币样式的例子

样 式	正 负 号	结 果
symbol none sign value		\$ 1234.56
symbol none sign value	-	\$ 1234.56
symbol space sign value	-	\$ - 1234.56
symbol space sign value	()	\$ (1234.56)
sign symbol space value	()	(\$ 1234.56)
sign value space symbol	()	(1234.56 \$)
symbol space value sign	-	\$ 1234.56 -
sign value space symbol	-	- 1234.56 \$
sign value none symbol	-	- 1234.56 \$

下面是简例：

```
const std::money_put<chart, OutIt> & mp = std::use_facet<std::money_put<charT, OutIt>>
(loc);
mp.put(to, intl, fmt, fill, value);
```

其中参数 `to` 是一个型别为 `OutIt` 的输出型迭代器（格式化后的字符串会写到该处）。`put()` 函数返回该型对象，指向“生成的最后一个字符”的下一位置。参数 `intl` 表示使用当地货币符号或国际货币符号。`fmt` 用于确定格式化标识，例如宽度和 `money_punct` facet 刻面。`fill` 为填充字符。参数 `value` 的型别是 `long double` 或 `basic_string<charT>`，其内容即为待被格式化的值。若传入的是字符串，此字符串可以全部由小数组成，前面可以带上负号。若字符串的第一个字符是个负号，则该值将依负值形式加以格式化。若确定其值为负值，则负号即可被丢弃。字符串中的小数个数可以由 `money_punct` facet 的成员函数 `frac_digits()` 确定。

C++ 标准要求每个类 `locale` 中必须有两个实例化对象：`money_put<char>` 和 `money_put<wchar_t>`。此外，C++ 标准程序库还支持“第一个模板参数为 `char` 或 `wchar_t`，第二个模板参数为相应的 `output` 迭代器”的所有 `money_put<>` 实例化对象，但不要求每个类 `locale` 必须拥有它们。

3. 货币解析

`facet money_get` 刻面用于解析货币值。它是一个模板类，接受字符型别 `charT` 作为第一个模板参数，接受输入型迭代器型别 `InIt` 作为第二个模板参数。此类别定义了两个 `get()` 成员函数，用于解析一个字符序列。若解析成功，则结果会被储存为 `long double` 型别或 `basic_string<charT>` 型别。例如，

```
const std::money_get<charT, InIt> & mg = std::use_facet<std::money_get<charT, InIt>>(loc);
mg.get(beg, end, intl, fmt, err, val);
```

被解析的是 `beg` 和 `end` 之间的字符序列。当所有元素均可被解析完毕，或是解析过程中遇到错误时，解析动作即可停止。若发生错误，`err` 将设立 `ios_base::failbit`，而 `val` 中将不会有任何东西。若解析成功，其结果放在参数 `val` 中，型别为 `long double` 或 `basic_string`。

参数 `intl` 是布尔量值，用于选定当地货币字符串或国际货币字符串。通过“参数 `fmt` 渗入”的 `locale` 对象，获取一个 `moneypunct` 剖面。该剖面用于定义出待解析的格式。对于货币的解析而言，`moneypunct` 剖面的成员函数 `neg_format()` 所返回的样式始终会被使用。

在 `none` 或 `space` 位置上，货币解析函数会吞噬所有有效空白，除非 `none` 是样式的最后一部分。尾部空白不会被忽略。`get()` 函数会返回一个迭代器。该迭代器指向最后一个被吞噬字符的下一位置。

C++ 标准要求每个类 `locale` 内均必须有两个实例化对象：`money_get < char >` 和 `money_get < wchar_t >`。此外，C++ 标准程序库要求支持所有“第一个模板参数为 `char` 或 `wchar`，第二个模板参数为相应的输入型迭代器”的 `money_get < >` 实例化对象，但不要求每个类 `locale` 必须拥有它们。

12.5.4 字符的分类和转换

C++ 标准程序库包含两个用于处理字符的剖面：`ctype` 和 `codecvt`。这两个剖面均属于 `locale::ctype` 类型。`ctype` 主要用于字符分类，此外还提供字母的大小写转换功能，以及在“型别 `char`”和“该 `facet` 用以实例化对象的字符型别”之间的转换方法。`codecvt` 用于表示在不同编码间进行字符转换，并主要用于 `basic_filebuf` 中的外部表述和内部表述之间的转换。

1. 字符分类

`ctype` 剖面是一个模板类，用以针对字符型别实现参数化。其定义的成员和功能见表 12-28。`ctype < charT >` 包含以下 3 个函数：

- 1) `char` 和 `charT` 之间的转换函数。
- 2) 字符分类函数。
- 3) 字母的大小写转换函数。

表 12-28 剖面 `ctype` 定义的成员和功能

表 达 式	功 能
<code>is (m, c)</code>	测试字符 <code>c</code> 是否匹配 <code>mask</code> (掩码) <code>m</code>
<code>is (beg, end, vec)</code>	为 <code>[beg, end]</code> 区间内的每个字符放置一个 <code>mask</code> ，后者必须匹配 <code>vec</code> 中对应位置上的字符
<code>scan_is (m, beg, end)</code>	返回一个指针，指向 <code>[beg, end]</code> 区间内第一个匹配的 <code>mask m</code> 的字符；若没有找到，返回 <code>end</code>
<code>scan_not (m, beg, end)</code>	返回一个指针，指向 <code>[beg, end]</code> 区间内第一个不匹配的 <code>mask m</code> 字符；若全部匹配，返回 <code>end</code>
<code>toupper()</code>	返回字符 <code>c</code> 的大写字符；若没有，则返回 <code>c</code>
<code>toupper()</code>	将 <code>[beg, end]</code> 区间内的每个字符均以 <code>toupper()</code> 的转换结果取代
<code>tolower()</code>	返回 <code>c</code> 的小写字符。如果没有，则返回 <code>c</code>
<code>tolower (beg, end)</code>	将 <code>[beg, end]</code> 区间内的每个字符均以 <code>tolower()</code> 的转换结果取代
<code>widen()</code>	将字符转换为 <code>charT</code> ，并将结果返回
<code>widen (beg, end, dest)</code>	将 <code>[beg, end]</code> 区间内的每个字符均以 <code>widen()</code> 转换，并将结果存放于 <code>dest</code> 的相应位置
<code>narrow (c, default)</code>	将字符转换为 <code>charT</code> ，并将结果返回。若没有合适的对应字符，即返回 <code>default</code>
<code>narrow (beg, end, default, dest)</code>	将 <code>[beg, end]</code> 区间内的每个字符均以 <code>narrow()</code> 转换，并将结果存放于 <code>dest</code> 的相应位置

函数 `is (beg, end, vec)` 用于在一个数组中存储一系列掩码 (mask)。对于 `beg` 和 `end` 之间的每个字符, 在 `vec` 所指数组中均有一个对应的 mask 以及相应的属性。很多字符需要被分类, 以避免“字符分类”的虚函数调用操作。

`widen()` 函数用于将一个本地 (native) 字符集内“型别为 `char` 的字符转换为‘locale’所用字符集”内的对应字符, 即此函数用于拓宽一个字符。

结果是 `char` 型别也无妨。相反, `narrow()` 函数用来将“locale 所用的字符集”内的字符转换为 native 字符集内对应的字符。下面的例子将数字字符由 `char` 型别转换为 `wchar_t` 型别:

```
std::locale loc;
char narrow [] = "0123456789";
wchar_t wide [10];

std::use_facet<std::ctype<wchar_t>>(loc).widen(narrow, narrow+10, wide);
```

类 `ctype` 派生自类 `ctype_base`, 此类别仅用于定义一个枚举, 名为 `mask`。`mask` 定义一系列值, 可用于合成一个 `bitmask`, 用以检测字符属性。`ctype_base` 使用的各种字符掩码见表 12-29。字符分类的所有函数均需要一个 `bitmask` 参数, 后者是由 `ctype_base` 内的值组合而成。为了获得需要的位掩码 (bitmask), 要使用各种位操作符 (`|`、`&`、`^`和 `~`)。若字符涵盖某个 `mask` 所规范的字符内, 则该字符必定匹配那个 `mask`。

表 12-29 ctype 使用的各种字符掩码

值	意 义	值	意 义
<code>ctype_base::alnum</code>	测试是否为字母或数字	<code>ctype_base::print</code>	测试是否为可打印字符
<code>ctype_base::alpha</code>	测试是否为字母	<code>ctype_base::punct</code>	测试是否为标点符号
<code>ctype_base::cntrl</code>	测试是否为控制字符	<code>ctype_base::space</code>	测试是否为空白
<code>ctype_base::digit</code>	测试是否为数字	<code>ctype_base::upper</code>	测试是否为大写字母
<code>ctype_base::graph</code>	测试是否为标点符号、字母或数字	<code>ctype_base::xdigit</code>	测试是否是十六进制数字
<code>ctype_base::lower</code>	测试是否为小写字母		

2. 针对 char 而做的 ctype 特化版本

为使字符分类函数获取更佳性能, `ctype` 针对字符型别 `char` 有一个特化版本。此特化版本并未将字符分类 (`is()`, `scan_is()`, `scan_not()`) 函数委托给相应的虚函数处理, 而是通过查表动作, 以 `inline` 方式直接实例而得。为此, `ctype < char >` 提供了一些额外的成员函数, 详见表 12-30。

表 12-30 ctype < char > 的额外成员函数及其功能

表 达 式	功 能	表 达 式	功 能
<code>ctype < char >::table_size</code>	返回表格大小 (> = 256)	<code>ctype < char > (table, del = false)</code>	以 <code>table</code> 为表格, 产生一个刻面
<code>ctype < char >::classic_table()</code>	返回经典的“C” locale 的表格	<code>table()</code>	返回刻面 <code>facet</code> 的当前格式

特化版本针对特定的 locales, 操控上述函数的行为。其做法仅仅是将某个 `mask` 表格传

递给 ctype 构造函数作为参数。

静态成员 table_size 是一个常量,用于指示表格大小,由 C++ 标准程序库实例版本自行定义。其值至少为 256。ctype < char > 构造函数的第二个参数用于指示“当刻面被销毁时,表格是否应被删除”。若此参数值为 true,则当刻面被销毁时,传入构造函数的那个表格会被 delete[] 释放掉。

保护型成员 table() 函数返回的是构造函数的第一参数。静态保护成员函数 classic_table() 返回的是经典的“C”locale 中用于进行字符分类的表格。

3. 用于字符分类的全局辅助函数

C++ 标准程序库定义了一些全局函数,可以协助程序员方便地运用 ctype 刻面。表 12-31 列出了所有用于字符分类的全局辅助函数。

表 12-31 用于字符分类的全局辅助函数

函 数	功 能	函 数	功 能
isalnum()	判断 c 是否为字母或数字	ispunct()	测试 c 是否为标点符号
isalpha()	测试 c 是否为字母	isspace()	测试 c 是否为空格
isctrl()	测试 c 是否为控制字符	isupper()	测试 c 是否为大写字母
isdigit()	测试 c 是否为数字	isxdigit()	测试字符 c 是否为十六进制
isgraph()	测试 c 是否为标点符号	tolower()	将字符 c 从大写转换为小写字母
islower()	测试 c 是否为小写字母	toupper()	将字符 c 从小写转换为大写
isprint()	测试 c 是否为可打印字符		

例如,对于表达式确定 locale loc 中的一个字符 c 是不是小写字母:

```
std::islower(c, loc)
```

函数会返回一个 bool 值。

如果 c 是 locale loc 中的小写字母,以下表达式会返回对应的大写字母:

```
std::toupper(c, loc);
```

若 c 不是小写字母,则第一个参数会原封不动地返回。

```
std::islower(c, loc);
```

上述表达式等同于以下表达式:

```
std::use_facet<std::ctype<char>>(loc).is(std::ctype_base::lower,c)
```

此表达式调用 ctype < char > 刻面的成员 is() 函数,用以判断字符 c 是否符合第一个参数传来的位掩码所指定的字符属性。位掩码实值定义于 ctype_base 内。

这些字符分类全局函数和“同名但只有一个参数”的 C 函数相对应。这些定义于 < ctype > 和 < type. h > 的 C 函数总是采用当前的全局 locale,用法简单直接。

```
if(std::isdigit(c))
{
...
}
```

若使用上述辅助函数,在同一个程序内使用不同的 locale,则将无法运用使用者自定义

的 ctype 刻面。

如果考虑字符转换效率，应尽力避免使用 C++ 函数，而应从 locale 中获取对应的刻面对象，然后直接使用该对象的成员函数。很多字符需要根据同一个 locale 进行分类。函数 is() 可用于判断典型字符的掩码。函数对 [beg, end] 区间内的每个字符确定一个掩码，用以描述该字符的属性。这些掩码被存储在 vec 中，其位置对应于字符位置。快速搜寻时，可使用向量 vector。

4. 字符编码转换

codecvt 刻面用于在字符内部编码和外部编码之间进行转换。例如，只要某个 C++ 标准程序库实例版本支持相应的刻面，即可使用 codecvt 在 Unicode 编码和 EUC (extended UNIX Code) 编码之间进行转换。

该刻面在类 basic_filebuf 中被用于在“内部表述”和“文件表述”之间进行转换。类 basic_filebuf <charT, traits > 使用 codecvt <charT, char, typename traits::state_type > 的具体实例对象来完成此项任务：所使用的刻面是从储存于 basic_filebuf 的 locale 中抽取出来的。这是 codecvt 的主要用途——刻面很少直接使用。

为了解 codecvt，读者应该明白字符编码有两种方案：一种是对每个字符以固定个数的字节表示；另一种是对每个字符以不同个数的字节表示。

多字节表示法为了使字符的空间储存率更高，使用了所谓的转换状态。只有当前位置的转换状态正确，才可能正确翻译字节的含义。只有遍历整个多字节字符序列之后，才能正确理解其含义。

codecvt 刻面包含 3 个函数：

- 1) 字符型别 internT 用于指定内部表述方式。
- 2) 型别 externT 用于指定外部表述方式。
- 3) 型别 stateT 表示转换过程中的中间状态。

中间状态可能由不完全的宽字符或当前的转换状态组成。C++ 标准程序库对于该 state 对象中具体存储什么东西没有强制要求。

内部表述始终采用“每个字符的字节个数固定”的表述方案。在大部分情况下，程序使用的是 char 和 wchar_t 两个型别。对于外部表述，程序可能使用固定的字节表示法，也可能使用多字节表示法。若采用后者，第二个模板参数用于表示多字节表示法。若采用前者，第二个模板参数会用于表示多字节编码中的基本单位的型别，即每个多字节字符均存放在一个或多个该型对象中。

第三个模板参数用于表示当前转换状态，某些时候会显示出其必要性。处理单个字符时，可能会因“源端缓冲区”已空或“目的端缓冲区”已满而导致多字节字符的处理中断。若出现这种情况，则可将当前状态储存于该型对象中。

和其他刻面相同，C++ 标准仅仅强制要求支持少数转换。C++ 标准程序库仅支持以下两个具体实例化对象：

- 1) codecvt <char, char, mbstate_t >。该实例化对象将 native 字符集转换为其自身。
- 2) codecvt <wchar_t, char, mbstate_t >。该实例化对象在 native 窄字符集中与 native 字符集之间进行转换。

C++ 标准程序库没有指定第二个转换的确切语意。自然的做法是在 wchar_t 转换为 char

时, 每个 `wchar_t` 可以切割为 `sizeof(wchar_t)` 个 `char` 对象。反向转换时, 可将 `sizeof(wchar_t)` 个 `chars` 组装成一个 `wchar_t`。转换动作与 `ctype` 刻面的成员函数 `widen()` 和 `narrow()` 很不相同: `codecvt` 函数使用多个单字节字符构成一个宽字符 (`wchar_t`); `ctype` 函数则是将某种编码下的某个字符转换为另一种编码下的对应字符。

和 `ctype facet` 一样, `codecvt` 同样派生一个基类 `codecvt_base`, 该基类中也定义了一个枚举型别 `result`, 其枚举值用于指定 `codecvt` 成员函数的结果, 其确切意义视具体的成员函数而定。

`in()` 函数将一个外部表述转换为内部表述。参数 `s` 是一个 `reference`, 指向 `stateT`。这个“代表转移状态”的参数在转换开始时派上用场; 结束时, 最后一个转移状态也记录于其中。若待转换的输入缓冲区不是第一个被转换的缓冲区, 则传入的转移状态可能和初始状态有所不同。参数 `fb` 和 `fe` 的型别都是 `const internT*`, 分别代表输入缓冲区的起点和终点。参数 `tb` 和 `te` 的型别是 `externT*`, 分别代表输出缓冲区的起点和终点。参数 `fn` 和 `tn` 分别用于表示输入和输出缓冲区中经过转换的序列的终点。函数返回型别是 `codecvt_base::result`。`codecvt` 刻面的成员函数见表 12-32。

表 12-32 `codecvt` 刻面的成员函数

表 达 式	意 义
<code>in</code>	将外部表述转换为内部表述
<code>out</code>	将内部表述转换为外部表述
<code>unshift</code>	撰写脱序序列以切换最初的转移状态
<code>encoding</code>	返回外部编码相关信息
<code>always_noconv</code>	若转换不成功, 返回 <code>true</code>
<code>length()</code>	从序列返回 <code>externTs</code> 的数量, 以便产生 <code>max</code> 个内部字符
<code>max_length()</code>	返回“生成一个 <code>internT</code> ”所需的最大数量的 <code>externTs</code>

转换函数的返回值见表 12-33。

表 12-33 转换函数的返回值

返 回 值	意 义
<code>ok</code>	所有字符均成功转换
<code>partial</code>	①并非所有字符被成功转换; ②需要更多字符以生成目标字符
<code>error</code>	遇到一个不能转换的源字符
<code>noconv</code>	无需转换

返回值 `ok` 表示函数已取得部分进展。若保持 `fn == fe`, 则代表整个输入缓冲区均被处理, 并且 `tb` 和 `tn` 间的序列内含转换结果, 其中的字符表示输入序列中的字符, 并有一个上次转换留下的结束字符。若 `in()` 函数的参数 `s` 不是初始状态, 则其中储存上次转换未完成的那个“局部字符”。

返回值 `partial` 包含两层含义: 其一表示输入缓冲区中的内容尚未耗尽之前输出缓冲区已经满溢; 其二表示字符处理完毕之前输入缓冲区的内容已经耗尽。此时, `tb` 和 `tn` 之间的序列包含所有转换完毕的字符, 但输入缓冲区的尾端有一个尚未被完全转换的“部分字符”。下次转换时, 为正确表达转换这个“部分字符”, 需要一些信息, 该信息储存在转移状态 `s`

中。若 $fe! = fn$ ，则输入缓冲区尚未清空，必定存在 $te == tn$ ，即输出缓冲区满溢。下次转换会从 fn 开始。

返回值 `noconv` 用于只是一个特殊情况：若外部表述转换为内部表述，则无需进行任何转换。此时 fn 和 fb 相同， tn 和 tb 相同。目标序列中不包含任何东西，输入缓冲区的内容已经足够使用。

返回值 `error` 意味着遇到不可转换的字符。“标的字符集”中没有合适的表述可以对应字符，输入序列结束时的转移状态是非法的。C++ 标准没有进一步规划任何函数而调查错误原因。

函数 `out()` 和 `in()` 是相似的，只有其转换方向相反，即内部表述转化为外部表述。参数和返回值的意义和 `in()` 同样，参数型别调换之后， tb 和 te 是 `internT *`， fb 和 fe 是 `const externT *`。同样的情况也发生在 fn 和 tn 上。

当前的转换状态被当作 `unshift()` 函数的参数 s ，该函数会安插必要的字符以形成一个序列。这通常意味着转换状态被转换为移动状态，即外部表述到达尾部。参数 tb 和 tf 的型别均是 `externT *`， tn 的型别是 `externT& *`。 tb 和 te 之间的序列是输出缓冲区，其中存储着转换之后的字符。成果序列的终点储存于 tn 之中。`unshift()` 函数的返回值见表 12-34。

表 12-34 `unshift()` 的返回值

返回值	意义	返回值	意义
<code>ok</code>	序列成功完成	<code>error</code>	错误（无效）状态
<code>partial</code>	需要存进更多字符，才能完成这个序列	<code>noconv</code>	完成这个序列无需任何字符

`encoding()` 函数返回外部表述的编码信息。若返回 -1 ，则转换动作将由状态决定。若返回 0 ，则产生内部字符所需的 `externTs` 数量不是常数。若返回 -1 和 0 之外的其他数值，该值表示生成内部字符所需的 `externTs` 数量。该信息可用于粗略判断缓冲区的大小。

若函数 `in()` 和 `out()` 永远不能进行转换，则 `always_noconv()` 函数返回 `true`。

`length()` 函数返回“生成 max 个内部字符”所需的必要的 `externTs` 个数。若 $fb \sim fe$ 序列中完整的 `internT` 字符数目少于 max ，此函数会返回“能够产生序列内最多 `internTs`”的 `externTs` 的数量。

12.5.5 字符串校勘

`collate` 剖面用于处理字符串排序时各种不同约定间的差异。针对同一字符序列，不同的语言进行排序时，规则也不相同。`collate` 剖面可以提供用户熟悉的字符串排序。表 12-35 中列出了部分剖面 `collate` 的部分成员函数。

表 12-35 `collate` 剖面的部分成员函数

表达式	意义
<code>compare()</code>	返回 1 ——若第一字符串大于第二字符串 返回 0 ——若两字符串相等 返回 -1 ——若第一字符串小于第二字符串
<code>transform()</code>	返回一个字符串，被用于和其他改造的字符串作比较
<code>hash()</code>	返回字符串的一个散列

collate 刻面是模板类，以字符型别 charT 作为模板参数。传给 collate 成员函数的字符串，是型别为 const charT* 的迭代器指针。不能保证 basic_string < charT > 中的迭代器是指针。很有必要为指针型别和数量不限的迭代器型别配备 collation 刻面。locale 中特殊的便捷函数可以进行字符串比较。例如，

```
int res = loc (s1, s2);
```

仅仅是对 compare 有效。C++ 标准程序库没有针对 collate 的另外两个成员函数定义对应的便捷函数。

transform() 函数返回型别为 basic_string < charT > 的对象。字符串的字典顺序和使用 collate() 的原始字符串相同。若字符串有必要和其他众多字符串比较，该顺序可协助提高性能。决定字符串字典顺序的动作远比 collate() 好得多。地域化的排序规则可能相对复杂。

C++ 标准程序库仅仅强制要求支持 collate < char > 和 collate < wchar_t > 两个实例化对象。对于其他字符型别，用户需要自定义特化版本。

12.5.6 信息国际化

message 刻面用于从一个信息索引中获取被国际化的信息。该刻面主要用于提供类似于 perror() 的功能。perror() 在 POSIX 系统中根据存储于全局变量 errno 中的错误编号，印出系统错误信息。messages 提供的功能更加灵活，其定义也非常精确。

message 刻面是模板类，以字符型别 charT 为其模板参数。该刻面返回的字符串，型别为 basic_string < charT >。刻面的基本用途是：打开一个信息名册，从中获取信息，然后关闭该名册。类 messages 派生于类 messages_base。该基类用以定义新的 catalog 型别（其实就是一个 int 型别）。此型别的对象用于标识一个名册，messages 的成员函数即针对此名册进行操作。表 12-36 列出了 messages 的成员函数。

表 12-36 messages 的成员函数

表 达 式	意 义
open()	打开一个信息名册，返回对应的 ID
get()	从 catalog 型别的对象中返回 msgid 的对应信息
close()	关闭名册

成员函数 open() 可接受 name 参数标识的名册，相应信息字符串存储于其中。它也可以是文件名称。参数 loc 标识出一个 locale 对象，通过它可以存取 ctype 刻面。运用该刻面可以将信息转换成期望的字符型别。

成员 get() 函数的确切语义没有定义。例如，POSIX 系统中的实例化版本会返回一个对应的 msgid 的错误信息字符串。但 C++ 标准没有进行强制规定。参数 set 用于在信息中建立一个子结构。例如，可用于区别系统错误和 C++ 标准程序库的错误。

若不需要信息名册，则使用 close() 将其关闭。虽然函数 open() 和 close() 的接口名称暗

示信息来自某个文件，但C++标准对此并没有硬性要求。更多的情况是：使用 `open()` 从文件中读取信息，之后将其储存于内存中，最后调用的 `close()` 释放相应的内存。

需要记住的，C++标准要求每个类 `locale` 内必须有两个实例化对象：`messages < char >` 和 `messages < wchar_t >`。除此之外C++标准程序库不支持其他的任何实例体。

12.6 小结

本章内容比较多。第一节简单描述了国际化元素；第二节讲述了多种字符编码；第三节讲述了类 `locale`，并深入分析了 `locale` 的剖面、区域等概念；第四节主要讲述标准 `locale` 类的分类，其中涉及了类 `ctype`、`numeric`、`collate`、`time`、`money`、`messages` 等内容。第五节又针对上述提到的内容详细讲述了各种剖面的使用方法。

第 13 章

仿 函 数

前面章节讲述算法时，读者会注意到很多 STL 算法使用了仿函数（函数对象）。仿函数也叫函数符。函数符是以函数方式与括号（）结合使用的任意对象。在 C++ 标准中，仿函数的英文名称是 Function Objects，即函数对象。更通俗地讲，仿函数是将函数作为参数传递的使用方式，例如逻辑谓词、算术运算、抽取信息等。

13.1 仿函数的概念

13.1.1 仿函数的概念

通常可以这样理解，仿函数是一个定义了 `operator()` 的对象，可以将其视为一般函数。仿函数与一般函数的不同之处在于：仿函数的功能是在其成员 `operator()` 函数中实现的。虽然仿函数的定义形式比较复杂，但其同样具有一定的优点。仿函数的优点表现在以下 3 个方面：

1) 仿函数比一般函数更灵活，主要是因为仿函数拥有状态。仿函数可以拥有两个状态不同的实体。而通常的普通函数是不具备该特点的。

2) 每个仿函数都有其型别。仿函数的型别可以作为模板参数，用以实现指定某种行为的目的。容器型别不会和具体仿函数有关，而仅仅是该型别的仿函数均可使用。

3) 仿函数要比函数指针的执行速度快得多。在 C++ 标准中，调用函数通常使用指针，即当需要调用函数时，只需调用函数的地址（名称）即可。地址调用方法的缺陷是效率非常低。为提高效率，仿函数的形式被引入。定义的仿函数是通过使用运算符 `operator()` 被调用的。通过自定义运算符能显著提高效率。

仿函数均是以类似函数的形式被调用的。仿函数均包含成员 `operator()` 函数，这是仿函数的共同点。仿函数的基类是：

```
template < class _A, class _R > struct unary_function
{ typedef _A Argument_Type;           //单元函数的参数类型定义
  typedef _R Result_Type;
};
```

和

```
template < class Arg1, class Arg2, class Result > struct binary_function
{ typedef Arg1 first_argument_type;   //二元函数的参数 1 类型定义
  typedef Arg2 second_argument_type;  //二元函数的参数 2 类型定义
  typedef Result result_type;
};
```

这两个基类用于为参数提供标准的名字及规定返回值类型，从而使程序员可以从这两个基类派生出相应的自己的类。

类 `unary_function` 是空的基类，类没有任何的成员函数和成员变量，仅包含型别信息。该基类是为了让自适应单元函数模型（Adaptable Unary Function Models）的定义更方便。自适应单元函数模型必须包含嵌套型别声明。继承 `unary_function` 是获得嵌套型别的“不二法门”。

同样，类 `binary_function` 也是一个空类，仅包含成员函数和成员变量以及型别信息。该类的存在使自适应二元函数模型（Adaptable Binary Function models）的定义更方便。同样，自适应二元函数模型也必须包含嵌套型别声明。继承基类 `binary_function` 是获得嵌套型别的“不二法门”。

以上两个基类均声明于头文件 `<functional>` 中。

学习仿函数，首先要了解 3 个概念：生成器、一元函数和二元函数。

- 生成器是不用参数就可以调用的仿函数，即无参数仿函数。
- 一元函数是用一个参数就可以调用的仿函数。
- 二元函数是用两个参数才可以调用的仿函数。

例如，对于算法 `for_each()`，其调用的仿函数应该是一元函数。

上述概念还有其他说法：

- 返回 `bool` 值的一元函数是 `Predicate`（一元谓词）。
- 返回 `bool` 值的二元函数是 `binary predicate`（二元谓词）。

部分 STL 的函数需要一元参数或二元参数。例如，算法 `sort()` 需要使用二元函数作为其调用的仿函数。

此外，仿函数还有型别，即仿函数的返回值。无参数的仿函数具有一个返回型别；一元函数具有两个返回型别；二元函数具有 3 个返回型别。仿函数被定义为“类”，可以拥有嵌套型别。`unary Function` 和 `binary Function` 两个类中仅仅具有一些类型声明。

STL 还把仿函数和自适应仿函数区分开来。通常，一个仿函数会有一个参数型别与一个返回型别，程序并不知晓那些型别的具体名称。自适应仿函数会指明参数与返回型别为何会内含嵌套的类型定义，程序中指定并使用那些型别。若型别 `F1` 是自适应一元仿函数的模型，那么需要定义 `F1::argument_type` 和 `F1::result_type`；若型别 `F2` 是自适应二元仿函数，那么需要定义 `F2::first_argument_type`、`F2::second_argument_type` 和 `F2::result_type`。STL 的两个基类 `unary_function`（一元函数）和 `binary_function`（二元函数），可以使定义自适应的一元和二元仿函数简便许多。

13.1.2 仿函数的作用

程序员通常以下面 4 种形式使用仿函数：

- 1) 作为排序规则。
- 2) 拥有内部状态。
- 3) 算法 `for_each()` 的返回值。
- 4) 作为判断式。

1. 仿函数作为排序规则

仿函数可以将已序的数据放入容器中。使用通常的运算符“`operator <`”有时不能对这

些元素排序，因此需要自定义特别的规则来实现排序的目的，而仿函数可以作为排序规则，详见例 13-1。

例 13-1

```
#include <iostream>
#include <string>
#include <set>
#include <algorithm>
#include <fstream>
using namespace std;
class person //对象类
{
public:
    string firstname()
    {
        return fname;
    };
    string lastname()
    {
        return lname;
    };
public:
    string fname;
    string lname;
};
class person_sort_rule //排序规则类
{
public:
    bool operator () (person p1, person p2)
    {
        return (p1.lastname() < p2.lastname());
    };
};
string getfirstname(string buffer) //从缓冲区获取 firstname
{
    string r;
    size_t index = buffer.find(',');
    if (index != -1)
        r = buffer.substr(0, index);
    else
        r = "-1";
    return r;
}
string getlastname(string buffer) //从缓冲区获取 lastname
{
    string r;
```

```
size_t index = buffer.find(',');
if (index != -1)
    r = buffer.substr(index + 1, buffer.length() - 1);
else
    r = "-1";
return r;
}
void main()
{
    typedef set<person, person_sort_rule> personSet;
    personSet myset;
    myset.clear();
    string myfname, mylname;
    ifstream myfile;
    myfile.clear(); // 删除该文件流的所有标志
    string filename;
    string buffer;
    cout << "Please input the filename: " << endl;
    cin >> filename;
    cout << "Inputed filename : " << filename << endl;
    myfile.open(filename.c_str(), ios_base::in); // 打开文件
    if (myfile.is_open())
    {
        while (getline(myfile, buffer) && buffer.size() > 0)
        {
            myfname = getfirstname(buffer);
            mylname = getlastname(buffer);
            cout << myfname << ", " << mylname << endl;
            if (myfname != "-1" && mylname != "-1") // 将数据添加至 set 中
            {
                person temp;
                temp.fname = myfname;
                temp.lname = mylname;
                myset.insert(temp);
            }
        }
    }
    myfile.clear();
    myfile.close();
    personSet::iterator it;
    for (it = myset.begin(); it != myset.end(); it++) // 输出数据
    {
        cout << (*it).fname << ", " << (*it).lname << endl;
    }
}
```

例 13-1 的执行结果为:

```
Please input the filename:
test.txt
Inputed filename : test.txt
Abra, tom
kelinton, gulan
Crack, pack
grat, jack
koom, wert
grat, jack
Abra, tom
koom, wert
kelinton, gulan
Crack, pack
```

例 13-1 使用了仿函数类 `person_sort_rule`。该类定义了 `operator()` 函数。该函数对输入的数据的 `lastname` 字段进行比较。所有元素以此作为排序规则进行排序。若以一般的函数作为排序规则，则比较困难。由于 `set` 具有自动排序性，这是不能避免的，但程序员可以设计自己的排序规则。

2. 仿函数可以拥有内部状态

仿函数还可以使程序在同一时刻拥有多个状态。函数传递数值的方式也有两种：通过值传递；通过引用传递。仿函数是采用第一种方式：值传递。算法并不会改变随参数而来的仿函数的状态。仿函数参数采用“传递值”方式的好处是可以传递常量或临时表达式；缺陷是无法改变仿函数的状态。算法可以改变仿函数的状态，但无法存取或改变其最终状态，改变的仅仅是仿函数的副本。有时需要存取最终状态，其关键是怎样从算法中获取结果。

从使用仿函数的算法中获取结果或反馈的方法通常有两种：

1) 以引用的方式传递仿函数。若希望能够以引用的方式传递仿函数，则需要在调用算法时明确表示仿函数型别是引用型别。

2) 使用 `for_each()` 算法的返回值。

下面以例 13-2 来说明上述知识点。

例 13-2

```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;
void print(list<int> & lt)
{
    list<int>::iterator it;
    for(it = lt.begin(); it != lt.end(); it++)
    {
        cout << *it << " ";
    }
    cout << endl;
```

```
}  
class sequence  
{  
private:  
    int value;  
public:  
    sequence(int ivalue):value(ivalue)  
    {  
    }  
    int operator() ()  
    {  
        return value ++;  
    }  
};  
void main()  
{  
    list<int> col;  
    generate_n(back_inserter(col),9,sequence(1));  
    print(col);  
    generate(++col.begin(),--col.end(),sequence(42));  
    print(col);  
}
```

例 13-2 的执行结果为:

```
1, 2, 3, 4, 5, 6, 7, 8, 9,  
1, 42, 43, 44, 45, 46, 47, 48, 9,
```

本例是以仿函数产生一个整数序列。当仿函数的运算符 `operator()` 被调用时，函数会返回整数值并累加 1。可通过构造函数的参数指定初始值。例 13-2 还使用了 `generate()` 和 `generate_n()` 算法。这两个算法的作用是：产生数值用以写入群集内。在使用仿函数 `sequence()` 时，均使用了初始参数，该初始参数可作为产生序列的初始值。算法 `generator_n()` 会连续 `n` 次改写元素值，产生 `n` 个元素。例如，

```
sequence(42);
```

上述语句会产生以 42 为起始值的整数序列。

通过修改 `operator()` 可以产生更加复杂的序列。例如，

```
int operator() ()  
{  
    return ((value++) * 2 + 1);  
}
```

上述程序的执行结果为:

```
3, 5, 7, 9, 11, 13, 15, 17, 19,  
3, 85, 87, 89, 91, 93, 95, 97, 19,
```

若要实现以引用的方式传递函数，读者可将例 13-2 修改为例 13-3。

例 13-3

```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;
void print(list<int> & lt)
{
    list<int>::iterator it;
    for(it = lt.begin(); it != lt.end(); it++)
    {
        cout << *it << " , ";
    }
    cout << endl;
}
class sequence
{
private:
    int value;
public:
    sequence(int ival):value(ival)
    {
    }
    int operator() ()
    {
        return value++;
    }
};
void main()
{
    list<int> col;
    sequence seq(1);
    generate_n(back_inserter(col),4,seq);
    print(col);
    generate_n(back_inserter(col),4,sequence(42));
    print(col);
    generate_n(back_inserter(col),4,seq);
    print(col);
    generate_n(back_inserter(col),4,seq);
    print(col);
}
```

例 13-3 的执行结果为:

```
1 , 2 , 3 , 4 ,
1 , 2 , 3 , 4 , 42 , 43 , 44 , 45 ,
1 , 2 , 3 , 4 , 42 , 43 , 44 , 45 , 1 , 2 , 3 , 4 ,
1 , 2 , 3 , 4 , 42 , 43 , 44 , 45 , 1 , 2 , 3 , 4 , 1 , 2 , 3 , 4 ,
```

3. 算法 for_each() 的返回值

若使用算法 for_each(), 则不必实例化仿函数的“引用计数版本”来存取最终状态。算法 for_each() 有其独门绝技, 这是其他算法所没有的, 即可以返回其仿函数。通过算法 for_each() 的返回值可以获取仿函数的状态, 详见例 13-4。

例 13-4

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
class MeanV{
private:
    long num;
    long sum;
public:
    MeanV():num(0),sum(0)
    {

    }
    void operator()(int elem)
    {
        num++;
        sum += elem;
    }
    double value()
    {
        return static_cast<double>(sum * 1.0/num);
    }
};
void myprint(vector<int> &v)
{
    vector<int>::iterator it;
    for(it=v.begin();it!=v.end();it++)
    {
        cout << * it << ", ";
    }
    cout << endl;
}
void main()
{
    vector<int> col;
    int array[8] = {1,2,3,4,5,6,7,8};
    col.assign(array,array+8);
    myprint(col);
}
```

```
MeanV mv = for_each(col.begin(), col.end(), MeanV());  
cout << "Mean Value: " << mv.value() << endl;  
}
```

例 13-4 的执行结果为:

```
1, 2, 3, 4, 5, 6, 7, 8,  
Mean Value: 4.5
```

在语句 `for_each (col.begin(), col.end(), MeanV())` 中, `MeanV()` 产生一个反函数用于记录元素数量, 并计算所有元素的总和。此仿函数传递给算法 `for_each()`, 后者会针对容器内每个元素调用仿函数。而返回的仿函数被赋值给 `mv`。调用仿函数的成员函数 `value()`, 即可获取相应的平均值了。

4. 判断式可返回仿函数的状态

所谓判断式, 即返回布尔值的一个函数或仿函数。对于 STL 来说, 并不是所有返回布尔值的函数是合法的判断式 (谓词)。这样会导致很多意想不到的结果。

例 13-5

```
#include <iostream>  
#include <list>  
#include <algorithm>  
using namespace std;  
class Nth{  
private:  
    int nth;  
    int count;  
public:  
    Nth(int n):nth(n),count(0)  
    {  
    }  
bool operator () (int)  
    {  
    return ++count == nth;  
    }  
};  
void myout (list<int> & lt)  
{  
    list<int>::iterator it;  
    for(it = lt.begin(); it != lt.end(); it++)  
    {  
        cout << * it << " , ";  
    }  
    cout << endl;  
}  
void main()  
{  
    list<int> col;
```

```
int array[9] = {1,2,3,4,5,6,7,8,9};
for(int i=1;i<9;i++)
    col.push_back(i);
//col.assign(array,array+9);
myout(col);
list<int>::iterator pos;
pos = remove_if(col.begin(),col.end(),Nth(3));
col.erase(pos,col.end());
myout(col);
}
```

例 13-5 的执行结果为:

```
1, 2, 3, 4, 5, 6, 7, 8,
1, 2, 4, 5, 7, 8,
```

例 13-5 中使用了一个仿 Nth() 函数, 当被重复调用时, 此仿函数会返回 true。若将其传递给算法 remove_if() 时, 此算法会将所有符合条件的元素删除。

13.2 预定义仿函数

STL 定义了许多基本仿函数。例如, 其执行时会两个值相加、比较两个值等操作。提供这些函数对象是支持将函数作为参数的 STL 函数。常见的预定义仿函数见表 13-1。

表 13-1 预定义仿函数

仿函数	效果	仿函数	效果
negate < type > ()	- param	less < type > ()	param1 < param2
plus < type > ()	param1 + param2	greater < type > ()	param1 > param2
minus < type > ()	param1 - param2	less_equal < type > ()	param1 <= param2
multiplies < type >	param1 * param2	greater_equal < type > ()	param1 >= param2
divide < type >	param1 / param2	logical_not < type > ()	! param
modulus < type > ()	param1 % param2	logical_and < type > ()	Param1 && param2
equal_to < type > ()	param1 == param2	logical_or < type > ()	Param1 param2
not_equal_to < type > ()	param1 != param2		

当使用仿函数对对象进行排序或比较时, 通常都以 less <> 作为预设的准则。预设的排序操作经常按升序排序 (element < nextElement)。

使用这些预定义的仿函数, 必须包含头文件 < functional >。为实现对“国际化字符串”的比较, C++ 标准程序库还提供了可作为字符串排序准则的仿函数。

表 13-1 中的这些预定义仿函数均可在程序编写时任意使用。其中的仿函数可分为算术运算、关系运算和逻辑运算三大类。每类的仿函数既可以作为有名的仿函数使用, 也可作为无名的仿函数传递给其他函数。

13.3 辅助用仿函数

仿函数的组合能力很重要，即可以从一些软件组件构造出另一些组件。最简单的仿函数可以构造出非常复杂的仿函数。一般而言，所有函数行为均可由仿函数的组合而实现。C++ 标准程序库不能提供足够的配接器来到达此境界。从理论上讲，组合型配接器的功能会更强大，其用途会更加广泛。

(1) $f(g(\text{elem}))$

这是一元组合函数最一般的形式。一元判断式被嵌套调用， $g()$ 的执行结果作为 $f()$ 的参数。整个表达式的操作类似于一个一元判断式。

(2) $f(g(\text{elem1}, \text{elem2}))$

两个元素 elem1 和 elem2 作为参数传递给二元判断式 $g()$ 。其结果作为参数传给一元判断式 $f()$ 。整个表达式类似于一个二元判断式。

(3) $f(g(\text{elem}), h(\text{elem}))$

参数 elem 作为参数被传递给两个不同的一元判断式 $g()$ 和 $h()$ ，两者的结果由二元判断式 $f()$ 处理。此形式是以某种方法将单一参数“注射”至组合函数中，整个表达式类似于一个一元判断式。

(4) $f(g(\text{elem1}), h(\text{elem2}))$

此处参数 elem1 和 elem2 作为唯一参数传递给两个不同的一元判断式 $g()$ 和 $h()$ ，两个结果共同被二元判断式 $f()$ 处理。整个表达式类似于一个二元判断式。

上述形式的配接器并没有被写入标准中，也没有标准的名称。SGI STL 的实作版本中定义了两个名称，目前 C++ 社群正在寻找上述类型所有配接器的通用表达方式。组合型仿函数配接器的可能表现方式见表 13-2。

表 13-2 组合型仿函数配接器的可能表现方式

功 能	本书名称	SGI STL 的名称
$f(g(\text{elem}))$	<code>compose_f_gx</code>	<code>compose1</code>
$f(g(\text{elem1}, \text{elem2}))$	<code>compose_f_gxy</code>	
$f(g(\text{elem}), h(\text{elem}))$	<code>compose_f_gx_hx</code>	<code>compose2</code>
$f(g(\text{elem1}), h(\text{elem2}))$	<code>compose_f_gx_hy</code>	

13.3.1 一元组合函数配接器

最基本的组合型函数配接器是 SGI STL 实例版本的一部分。本小节首先以 `compose_f_gx` 进行嵌套计算来演示组合型仿函数的使用方法。最简单也是最基本的组合型仿函数配接器是将某一元运算结果作为另一个一元运算的输入，仅仅是嵌套调用两个一元仿函数，详见例 13-6。

例 13-6

```
#include <iostream>
#include <functional>
#include <vector>
```

```
#include <algorithm>
#include <iterator>
using namespace std;
template <class op1, class op2 > class compose_f_gx_t: public std::unary_function<typename
    op2::argument_type, typename op1::result_type >
{
private:
    op1 myop1;
    op2 myop2;
public:
    compose_f_gx_t(const op1 & o1, const op2 & o2):myop1(o1),myop2(o2)
    {
    }
    typename op1::result_type operator()(const typename op2::argument_type & x) const
    {
        return myop1(myop2(x));
    }
};
template <class op1, class op2 > inline compose_f_gx_t<op1,op2 > compose_f_gx(const op1 & o1,
const op2 & o2)
{
    return compose_f_gx_t<op1,op2 >(o1,o2);
}
void myprint(vector<int > & vt)
{
    vector<int >::iterator it;
    for(it=vt.begin();it!=vt.end();it++)
        cout << *it << " ";
    cout << endl;
}
void main()
{
    vector<int > vt;
    int array[9] = {1,2,3,4,5,6,7,8,9};
    vt.assign(array,array+9);
    myprint(vt);
    transform(vt.begin(),vt.end(),ostream_iterator<int >(cout," "),
        compose_f_gx(bind2nd(multiplies<int >(),5),bind2nd(plus<int >(),10)));
    cout << endl;
}
```

例 13-6 的执行结果为:

```
1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 ,
55 60 65 70 75 80 85 90 95
```

在例 13-6 中, 当执行语句 `transform()` 时, 参数 `compose_f_gx` 的第二个参数优先被执行,

即对于向量序列 vt, 其中的每个元素均先增加 10, 之后“和”再乘以 5。例如, 对于第一个元素“1”, 运算规则是: $(1 + 10) \times 5 = 55$ 。

下面以 `compose_f_gx_hx` 组合两个“运作规则”为例, 使用例 13-7 来说明该规则的使用方法。这可能是最重要的一个组合型函数配接器, 它允许将两个准则加以逻辑组合, 形成单一准则。在表 13-2 中, SGI STL 实作版本称之为 `compose2`, 详见例 13-7。

例 13-7

```
#include <functional>
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
template <class OP1, class OP2, class OP3> class compose_f_gx_hx_t
    : public std::unary_function<typename OP2::argument_type,
        typename OP1::result_type>
{
private:
    OP1 myop1;
    OP2 myop2;
    OP3 myop3;
public:
    compose_f_gx_hx_t(const OP1 & o1, const OP2 & o2, const OP3 & o3)
        : myop1(o1), myop2(o2), myop3(o3)
    {
    }
    typename OP1::result_type operator()(const typename OP2::argument_type & x)
    {
        return myop1(myop2(x), myop3(x));
    }
};
template <class OP1, class OP2, class OP3> inline compose_f_gx_hx_t<OP1, OP2, OP3>
    compose_f_gx_hx(const OP1 & o1, const OP2 & o2, const OP3 & o3)
{
    return compose_f_gx_hx_t<OP1, OP2, OP3>(o1, o2, o3);
}
void myprint(vector<int> & v)
{
    vector<int>::iterator it;
    for(it = v.begin(); it != v.end(); it++)
        cout << *it << " ";
    cout << endl;
}
void main()
{
```

```
vector<int> vt;
int array[9] = {1,2,3,4,5,6,7,8,9};
vt.assign(array,array+9);
myprint(vt);
vector<int>::iterator pos;
pos = remove_if(vt.begin(),vt.end(),
compose_f_gx_hx(logical_and<bool>(),bind2nd(greater<int>(),4),bind2nd(less<int>(),
7)));
vt.erase(pos,vt.end());
myprint(vt);
}
```

例 13-7 的执行结果为:

```
1, 2, 3, 4, 5, 6, 7, 8, 9,
1, 2, 3, 4, 7, 8, 9,
```

其中, 算法 `remove_if()` 中的第三个参数是由仿 `compose_f_gx_hx()` 函数来实现的。此仿 `compose_f_gx_hx()` 函数包含 3 个参数。对于例 13-7, 该仿函数确定的条件是: 大于 4 并且小于 7 的元素。所以, 程序执行完毕之后, 输出向量 `vt`, 仅剩余 7 个元素。向量中的元素“5”和“6”被删除了。

13.3.2 二元组合函数配接器

二元组合函数配接器可以将两个一元运算的结果加以处理。例 13-8 是一种较简单的实例。

例 13-8

```
#include <functional>
#include <iostream>
#include <algorithm>
#include <string>
#include <cctype>
using namespace std;
template < class OP1, class OP2, class OP3 > class compose_f_gx_hx_t:public std::binary_
function<
    typename OP2::argument_type,
    typename OP3::argument_type,
    typename OP1::result_type >
{
private:
    OP1 myop1;
    OP2 myop2;
    OP3 myop3;
public:
    compose_f_gx_hx_t(const OP1 & o1, const OP2 & o2, const OP3 & o3):
        myop1(o1),myop2(o2),myop3(o3)
    {
    }
}
```

```

    typename OP1::result_type operator() (const typename OP2::argument_type & x,
        const typename OP3::argument_type & y) const
    {
        return myop1 (myop2 (x), myop3 (y));
    }
};
template <class OP1, class OP2, class OP3 > inline compose_f_gx_hx_t <OP1, OP2, OP3 >
    compose_f_gx_hy (const OP1 & o1, const OP2 & o2, const OP3 & o3)
{
    return compose_f_gx_hx_t <OP1, OP2, OP3 > (o1, o2, o3);
}
void main()
{
    string s ("Internationalization");

```

```

    string sub (" Nation");

```

```

    string::iterator pos;
    pos = search (s.begin(), s.end(), sub.begin(), sub.end(), compose_f_gx_hy (equal_to <int > (),
        ptr_fun (::toupper), ptr_fun (::toupper)));
    if (pos != s.end())
    {
        cout << "\"" << sub << "\" is part of \"" << s << "\"" << endl;
    }
}

```

例 13-8 的执行结果为:

```
"Nation" is part of "Internationalization"
```

上述例题完成了在给定字符串中搜索子串的功能。

13.4 关系仿函数

预定义的关系仿函数对象包括等于、不等于、大于、大于等于、小于和小于等于。

13.4.1 等于 (equal_to <type > ())

仿函数类 `equal_to <type >` 是一种自适应一元谓词。该仿函数可用于验证某条件的真伪。若 `f` 是类 `equal_to <T >` 的对象, 并且 `x` 与 `y` 为型别 `T` 的值, 仅在 `x == y` 时, `f (x, y)` 才会返回 `true`。

使用仿函数 `equal_to` 时, 必须包含头文件 `<functional >`。参数 `type` 是仿函数参数的型别。其基类为:

```
binary_function <T, T, bool >
```

仿函数类 `equal_to` 包含了 5 个成员函数:

1) `equal_to::first_argument_type`。该成员函数代表第一个参数的型别 `T`。

- 2) `equal_to::second_argument_type`。该成员函数代表第二个参数的型别 `T`。
 - 3) `equal_to::result_type`。该成员代表执行结果的型别 `bool`。
 - 4) `bool equal_to::operator() (const T& x, const T& y) const`。该成员函数代表函数调用操作符。其返回值为 `x == y`。
 - 5) `equal_to::equal_to()`。该成员代表该仿函数类的默认构造函数。
- 使用示例:

```
equal_to<string>stringEqual;  
sres = stringEqual(sval1, sval2);  
ires = count_if(svec.begin(), svec.end(), equal_to<string>(), sval1);
```

下面给出例 13-9, 用以说明仿函数 `equal_to<type>()` 的使用方法。

例 13-9

```
#include <iostream>  
#include <functional>  
#include <algorithm>  
using namespace std;  
void main()  
{  
    const int N=10;  
    int A[N] = {1,3,0,2,5,9,0,0,6,0};  
    partition(A,A+N,bind2nd(equal_to<int>(),0)); //  
    copy(A,A+N,ostream_iterator<int>(cout," "));  
    cout << endl;  
}
```

例 13-9 的执行结果为:

```
0 0 0 0 5 9 2 3 6 1
```

例 13-9 的作用是将序列中等于零的元素放置在序列的前面。

13.4.2 不等于 (`not_equal_to<type>()`)

仿函数类 `not_equal_to<T>` 也是一种自适应二元谓词, 此仿函数可用于验证某条件之真伪。若 `f` 是类 `not_eaual_to<T>` 的一个对象, 并且 `x` 和 `y` 均为型别 `T` 的值, 仅在 `x! = y` 时, `f(x, y)` 才会返回 `true`。

仿函数 `equal_to<T>` 的参数型别和基类与仿函数 `equal_to` 近似。该类同样包含了 5 个成员函数。这 5 个成员函数分别是:

- 1) `not_equal_to::first_argument_type`。该成员函数代表第一个参数的型别。
- 2) `not_equal_to::second_argument_type`。该成员函数代表第二个参数的型别。
- 3) `not_equal_to::result_type`。该成员代表结果的型别 `bool`。
- 4) `bool not_equal_to::operator() (const T& x, const T& y) const`。该成员函数代表函数调用的操作符。其返回值为 `x! = y`。
- 5) `not_equal_to::not_equal_to()`。该成员函数代表该仿函数类的默认构造函数。

例 13-10

```

#include <iostream>
#include <functional>
#include <algorithm>
#include <list>
using namespace std;
void main()
{
    const int N=9;
    int A[N] = {0,0,0,0,1,2,4,8,0};
    list<int> L(A,A+N);
    list<int>::iterator i = find_if(L.begin(),L.end(),bind2nd(not_equal_to<int>(),0));
    cout << "Elements after initial zeros (if any): ";
    copy(i,L.end(),ostream_iterator<int>(cout," "));
    cout << endl;
}

```

例 13-10 的执行结果为:

```
Elements after initial zeros (if any): 1 2 4 8 0
```

13.4.3 小于 (less < type > ())

仿函数类 less < type > 也是一种自适应二元谓词, 此仿函数可用以验证某条件的真伪。若 f 是仿函数的一个对象, 并且 x 和 y 均是型别 type 的值, 仅在 $x < y$ 时, $f(x, y)$ 才可以返回 true。

STL 的许多仿函数和算法均需要比较函数。例如 sort、set 和 map。less 是典型的默认值。此仿函数的基类是 binary_function < type, type, bool >。该仿函数类包含了 5 个成员函数:

- 1) less::first_argument_type。该成员函数代表第一个参数的型别 type。
- 2) less::second_argument_type。该成员函数代表第二个参数的型别: type。
- 3) less::result_type。该成员代表结果的型别 bool。
- 4) bool less::operator() (const T& x, const T& y) const。该成员函数代表函数调用的操作符, 其返回值为 $x < y$ 。
- 5) less::less()。该成员函数代表该仿函数类的默认构造函数。

例 13-11

```

#include <iostream>
#include <functional>
#include <algorithm>
using namespace std;
void main()
{
    const int N=10;
    int A[N] = {1, -3, -7, 2, 5, -9, -2, 1, 6, -8};
    partition(A,A+N,bind2nd(less<int>(),0));
}

```

```
copy(A,A+N,ostream_iterator<int>(cout," "));
cout<<endl;
}
```

例 13-11 的执行结果为:

```
-8 -3 -7 -2 -9 5 2 1 6 1
```

13.4.4 大于 (greater < type > ())

仿函数类 `greater < type >` 也是一种自适应二元谓词, 可用于验证其条件之真伪。若 `f` 是仿函数的对象, 并且 `x` 与 `y` 均是型别 `type` 的值, 仅当 `x > y` 时, `f(x, y)` 才能返回 `true`。此仿函数类同样包含 5 个成员函数:

- 1) `greater::first_argument_type`。该成员函数代表第一个参数的型别 `type`。
- 2) `greater::second_argument_type`。该成员函数代表第二个参数的型别 `type`。
- 3) `greater::result_type`。该成员代表执行结果的型别 `bool`。
- 4) `bool greater::operator() (const T& x, const T& y) const`。该成员函数代表函数调用的操作符。其返回值为 `x > y`。
- 5) `greater::greater()`。该成员函数代表该仿函数类的默认构造函数。

13.4.5 大于等于 (greater_equal) 和小于等于 (less_equal)

仿函数类 `greater_equal < type > ()` 和 `less_equal < type > ()` 和以上的仿函数基本相似。此处不再赘述。

例 13-12

```
#include <iostream>
#include <functional>
#include <algorithm>
#include <vector>
using namespace std;
void main()
{
    const int N=10;
    int A[N] = {1, -3, -7, 2, 5, -9, -2, 1, 6, -8};
    vector<int> V(A,A+N);
    vector<int> v2;
    sort(V.begin(),V.end(),greater<int>()); //排序
    copy(V.begin(),V.end(),ostream_iterator<int>(cout," "));
    cout<<endl;
    remove_copy_if(V.begin(),V.end(),back_inserter(v2),bind2nd(less_equal<int>(),-7));
    //大于等于
    cout<<"Element remained : ";
    copy(v2.begin(),v2.end(),ostream_iterator<int>(cout," "));
    cout<<endl;
    vector<int>::iterator i=find_if(V.begin(),V.end(),bind2nd(greater_equal<int>(),6));
```

```
cout << * i << endl;
}
```

例 13-12 的执行结果为:

```
6 5 2 1 1 -2 -3 -7 -8 -9
Element remained : 6 5 2 1 1 -2 -3
6
```

13.5 逻辑仿函数

在讲述逻辑仿函数之前, 下面首先介绍“谓词”的概念。

13.5.1 谓词

返回 bool 类型的仿函数称为谓词。前面讲述的关系仿函数和本节讲述的逻辑仿函数都属于谓词。头文件 <functional> 包含以下两个仿函数:

```
template <class T> struct logical_not: public unary_function<T, bool> {
    bool operator() (const T & x) const
    {
        return ! x;
    }
}
template <class T> struct less: public binary_function<T, T, bool> {
    bool operator() (const T& x, const T& y) const
    {
        return x < y;
    }
}
```

一元谓词和二元谓词在算法中是非常有用的。尤其在比较两个序列时, 谓词可用于寻找某个序列中的第一个不小于在另一个序列里的对应元素的元素。

```
void f(vector<int> & vi, list<int> & li)
{
    typedef list<int>::iterator LI;
    typedef vector<int>::iterator VI;
    pair<VI, LI> p1 = mismatch(vi.begin(), vi.end(), li.begin(), less<int>());
    ...
}
```

算法 mismatch() 会反复地将其二元谓词作用于各对应元素, 直到比较失败为止。之所以使用 less<int>() 而不是 less<int>, 是因为此处需要的是一个对象, 而不是型别。若需要寻找的不是第一个不小于另一个序列中的对应元素的元素, 而是第一个比对应元素小的元素, 则可以设法寻找第一个使与之相反的比较谓词失败的对。

C++ 标准程序库提供的常用谓词见表 13-3。

表 13-3 常用谓词

谓 词			谓 词		
equal_to	二元	arg1 == arg2	ess_equal	二元	arg1 <= arg2
not_equal_to	二元	arg1 != arg2	llogical_and	二元	arg1 && arg2
greater	二元	arg1 > arg2	logical_or	二元	arg1 arg2
less	二元	arg1 < arg2	logical_not	一元	! arg
greater_equal	二元	arg1 >= arg2			

除了标准库中的谓词之外，程序员还可以自己编写自定义谓词。自定义谓词对于使用标准库和标准算法是非常有利的。尤其在使用标准库算法时，对于自定义的数据结构类型标准算法有时无法识别，此时只好使用自定义的仿函数解决问题。

13.5.2 逻辑仿函数

逻辑仿函数支持逻辑与、逻辑或和逻辑非运算。逻辑运算和算术运算非常相似，均执行“或”和“与”等操作。逻辑操作本身并不重要，其主要用途在于和仿函数适配器相结合，以便在使用仿函数时可以执行逻辑操作。

1. 逻辑与

```
logical_and<T>
```

仿函数类 `logical_and<T>` 是一种自适应二元谓词，可用于验证某条件的真伪。若 `f` 是仿函数类 `logical_and<T>` 的对象，并且 `x` 与 `y` 均是型别 `T` 的数值，而 `T` 可转换为 `bool`，仅当 `x` 和 `y` 皆为 `true` 时，`f(x, y)` 才会返回 `true`。

仿函数逻辑“与”在使用时需要包含头文件 `<functional>`。型别 `T` 还可以是 `bool` 类型。该仿函数的基类是 `binary_function<T, T, bool>`。

该仿函数类包含 5 个成员函数：

- 1) `logical_and::first_argument_type`。该成员函数代表第一个参数的型别 `T`。
- 2) `logical_and::second_argument_type`。该成员函数代表第二个参数的型别 `T`。
- 3) `logical_and::result_type`。该成员代表执行结果的型别 `bool`。
- 4) `bool logical_and::operator()(const T& x, const T& y) const`。该成员函数代表函数调用的操作符，其返回值为 `x&& y`。
- 5) `logical_and::logical_and()`。该成员函数代表该仿函数类默认的构造函数。

例 13-13

```
#include <iostream>
#include <algorithm>
#include <functional>
#include <list>
using namespace std;
template <class OP1, class OP2, class OP3> class compose_f_gx_hx_t
: public std::unary_function<typename OP2::argument_type,
typename OP1::result_type>
```

```

{
private:
    OP1 myop1;
    OP2 myop2;
    OP3 myop3;
public:
    compose_f_gx_hx_t(const OP1 & o1, const OP2 & o2, const OP3 & o3)
        :myop1(o1),myop2(o2),myop3(o3)
    {

    }
    typename OP1::result_type operator() (const typename OP2::argument_type & x)
    {
        return myop1(myop2(x),myop3(x));
    }
};
template <class OP1, class OP2, class OP3 > inline compose_f_gx_hx_t<OP1,OP2,OP3 >
compose_f_gx_hx(const OP1 & o1, const OP2 & o2, const OP3 & o3)
{
    return compose_f_gx_hx_t<OP1,OP2,OP3 >(o1,o2,o3);
}
void myprint(list<int> & l)
{
    list<int>::iterator it;
    for(it=l.begin();it!=l.end();it++)
        cout<<*it<<" ";
    cout<<endl;
}
}

void main()
{
    list<int> L;
    generate_n(back_inserter(L),1000,rand);
    //myprint(L);
    list<int>::iterator i=find_if(L.begin(),L.end(),compose_f_gx_hx(logical_and<bool>
(),
                                                                    bind2nd(greater_equal<int>(),1),
                                                                    bind2nd(less_equal<int>(),100)));
    assert(i!=L.end()||(*i>1 &&*i<=100));
    cout<<*i<<endl;
}
}

```

例 13-13 的执行结果为:

41

2. 逻辑或

logica_or<T>

仿函数逻辑“或”是一种自适应二元谓词，可用于验证某条件的真伪。若 f 是类 `logical_or<T>` 的对象，并且 x 和 y 均是型别 T 的数值，型别 T 还可以是 `bool` 类型，仅当 x 或 y 两者之中至少有一个为 `true` 时， $f(x, y)$ 才会返回 `true`。使用仿函数“逻辑或”时，同样需要包含头文件 `<functional>`。该仿函数的基类是 `binary_function<T, T, bool>`。该仿函数类包含 5 个成员函数：

- 1) `logical_or::first_argument_type`。该成员函数代表第一参数的型别 T 。
- 2) `logical_or::second_argument_type`。该成员函数代表第二参数的型别 T 。
- 3) `logical_or::result_type`。该成员代表执行结果的型别 `bool`。
- 4) `bool logical_or::operator() (const T&x, const T&y) const`。该成员函数代表函数调用的操作符。其返回值为 `x || y`。
- 5) `logical_or::logical_or()`。该成员函数代表该仿函数类的默认构造函数。

例 13-14

```
#include <iostream>
#include <algorithm>
#include <functional>
#include <list>
#include <assert.h>
using namespace std;
template <class OP1, class OP2, class OP3 > class compose_f_gx_hx_t
: public std::unary_function<typename OP2::argument_type,
typename OP1::result_type>
{
private:
    OP1 myop1;
    OP2 myop2;
    OP3 myop3;
public:
    compose_f_gx_hx_t(const OP1 &o1, const OP2 &o2, const OP3 &o3)
        : myop1(o1), myop2(o2), myop3(o3)
    {
    }
    typename OP1::result_type operator() (const typename OP2::argument_type &x)
    {
        return myop1(myop2(x), myop3(x));
    }
};
template <class OP1, class OP2, class OP3 > inline compose_f_gx_hx_t<OP1, OP2, OP3 >
compose_f_gx_hx(const OP1 &o1, const OP2 &o2, const OP3 &o3)
{
    return compose_f_gx_hx_t<OP1, OP2, OP3 >(o1, o2, o3);
}
void myprint(list<int> &l)
```

```

{
    list<int>::iterator it;
    for(it=l.begin();it!=l.end();it++)
        cout<<*it<<" ";
    cout<<endl;
}
void main()
{
    char str[]="The first line\nThe second line";
    int len=strlen(str);
    const char * wptr=find_if(str,str+len,compose_f_gx_hx(logical_or<bool>(),
                                                         bind2nd(equal_to<char>(),' '),
                                                         bind2nd(equal_to<char>(),' \n')));
    assert(wptr==str+len||*wptr==' '||*wptr==' \n');
    cout<<wptr<<endl;
}

```

例 13-14 的执行结果为:

```

first line
The second line

```

3. 逻辑非

仿函数逻辑非:

```
logical_not<T>
```

仿函数类 `logical_not<T>` 是一种自适应谓词, 可用于验证某条件的真伪。仿函数“逻辑非”仅仅需要一个参数。若 `f` 是仿函数“逻辑非” (`logical_not<T>`) 的对象, 并且 `x` 是型别 `T` 的数值, 型别 `T` 还可以是 `bool` 类型, 仅当 `x` 为 `false` 时, `f(x)` 才会返回 `true`。

仿函数“逻辑非”的基类是: `unary_function<T, bool>`

仿函数类“逻辑非”包含 4 个成员函数:

- 1) `logic_not::argument_type`。该成员函数代表参数的型别 `T`。
- 2) `logic_not::result_type`。该成员函数代表结果的型别 `bool`。
- 3) `bool logic_not::operator()(const T& x) const`。该成员函数代表函数调用的操作符, 其返回值为 `!x`。
- 4) `logical_not::logical_not()`。该成员函数代表该仿函数类的默认构造函数。

例 13-15

```

#include <iostream>
#include <algorithm>
#include <vector>
#include <functional>
#include <assert.h>
using namespace std;
void myprint(vector<bool> & vb)
{

```

```

vector<bool>::iterator it;
for (it=vb.begin (); it!=vb.end (); it++)
    cout<< * it<<" , ";
cout<<endl;
}
void main (int argc, char * argv [])
{
    const int N=10;
    vector<bool> v1;
    for (int i=0; i<N; ++i)
    {
        v1.push_back (rand () > (RAND_MAX/2));
    }
    cout<<" vector<bool> v1 : " <<endl;
    myprint (v1);
    vector<bool> v2;
    transform (v1.begin (), v1.end (), back_inserter (v2), logical_not<bool> ());
    cout<<" vector<bool> v2 : " <<endl;
    myprint (v2);
    for (int i=0; i<N; ++i)
        assert (v1 [i] ==! v2 [i]);
}

```

例 13-15 的执行结果为：

```

vector<bool> v1 :
0 , 1 , 0 , 1 , 1 , 0 , 0 , 1 , 1 , 1 ,
vector<bool> v2 :
1 , 0 , 1 , 0 , 0 , 1 , 1 , 0 , 0 , 0 ,

```

13.6 算术仿函数

预定义的算术仿函数对象包括加 (plus)、减 (minus)、乘 (multiplies)、除 (divide)、求余 (modulus) 和取反 (negate)，详见表 13-4。调用的操作符是和 type 相关联的实例。对于一个 class 类型，若提供该操作符的重载实例，即可调用该实例。尤其在处理数值类时，把标准的算数函数作为函数使用是非常有用的。

表 13-4 算术仿函数

算术仿函数			算术仿函数		
Plus	二元	arg1 + arg2	divides	二元	arg1/arg2
minus	二元	arg1 - arg2	modulus	二元	arg1 % arg2
multiplies	二元	arg1 * arg2	negate	一元	- arg

13.6.1 加、减、乘、除仿函数

仿函数类 `plus <T >` 是自适应二元仿函数。若 `f` 是仿函数的对象，并且 `x` 和 `y` 均为型别 `T` 的值，则 `f (x, y)` 会返回 `x + y`。使用该仿函数时，需要包含头文件 `<functional >`，其基类是 `binary_function <T, T, T >`。

仿函数类 `plus <T >` 包含 5 个成员函数：

- 1) `plus::first_argument_type`。该成员函数代表仿函数第一个参数的型别 `T`。
- 2) `plus::second_argument_type`。该成员函数代表仿函数第二参数的型别 `T`。
- 3) `plus::result_type`。该成员函数代表仿函数的执行结果型别 `T`。
- 4) `plus::operator()` (`const T& x, const T& y`) `const`。该成员函数代表函数调用的操作符。其返回值为 `x + y`。
- 5) `plus::plus()`。该成员函数代表该仿函数类的默认构造函数。

减法仿函数类 (`minus <T >()`)、乘法仿函数类 (`multiplies <T >()`)、除法仿函数类 (`divides <T >()`) 和加法仿函数类相似。

例 13-16

```
#include <iostream>
#include <functional>
#include <algorithm>
#include <vector>
#include <numeric>

using namespace std;
void myprint (vector<double> & vd)
{
    vector<double>::iterator it;
    for(it = vd.begin(); it != vd.end(); it++)
        cout << *it << ", ";
    cout << endl;
}
void main(int argc, char * argv[])
{
    int N=5;
    vector<double> v1(N);
    vector<double> v2(N);
    vector<double> v3(N);
    generate(v1.begin(), v1.end(), rand);
    fill(v2.begin(), v2.end(), -RAND_MAX/2.0);
    cout << "vector v1: " << endl;
    myprint(v1);
    cout << "vector v2:" << endl;
    myprint(v2);
    transform(v1.begin(), v1.end(), v2.begin(), v3.begin(), plus<double>());
```

```
cout << "vector v3 (plus) :" << endl;
myprint (v3);
transform (v1. begin (), v1. end (), v2. begin (), v3. begin (), minus < double > ());
cout << "vector v3 (minus) :" << endl;
myprint (v3);
N = 8;
vector < double > v4 (N);
for (int i = 0; i < N; ++i)
    v4 [i] = i + 1;
partial_sum (v4. begin (), v4. end (), v4. begin (), multiplies < double > ());
copy (v4. begin (), v4. end (), ostream_iterator < double > (cout, "\n"));
cout << endl;
cout << "vector v4 (multiplies) :" << endl;
myprint (v4);
N = 6;
vector < double > v5 (N);
generate (v5. begin (), v5. end (), rand);
transform (v5. begin (), v5. end (), v5. begin (), bind2nd (divides < double > (), 155.0));
cout << "vector v5 (divides) :" << endl;
myprint (v5);
}
```

例 13-16 的执行效果如图 13-1 所示。

```
vector v1:
41 , 18467 , 6334 , 26500 , 19169 ,
vector v2:
-16383.5 , -16383.5 , -16383.5 , -16383.5 , -16383.5 ,
vector v3(plus):
-16342.5 , 2083.5 , -10049.5 , 10116.5 , 2785.5 ,
vector v3(minus):
16424.5 , 34850.5 , 22717.5 , 42883.5 , 35552.5 ,
1
2
6
24
120
720
5040
40320
vector v4(multiplies):
1 , 2 , 6 , 24 , 120 , 720 , 5040 , 40320 ,
vector v5(divides):
101.445 , 74.0516 , 189.406 , 173.948 , 157.832 , 36.8065 ,
```

图 13-1 例 13-16 的执行效果

13.6.2 求余仿函数和求反仿函数

1. 求余仿函数 (modulus < T >)

仿函数类 modulus < T > 是一种自适应二元仿函数。若 f 是仿函数的一个对象，并且 x 和 y 均属于型别 T，则 f (x , y) 会返回 x%y。求余仿函数类包含 5 个成员函数：

1) modulus::first_argument_type。该成员函数代表仿函数第一参数的型别 T。

- 2) `modulus::second_argument_type`。该成员函数代表仿函数第二参数的型别 `T`。
 - 3) `modulus::result_type`。该成员函数代表执行结果的型别 `T`。
 - 4) `modulus::operator()` (`const T& x, const T& y`) `const`。该成员函数代表函数调用的操作符。其返回值为 `x%y`。
 - 5) `modulus::modulus()`。该成员函数代表此仿函数类的默认构造函数。
2. 求反仿函数 (`negate <T>`)
- 仿函数类 `negate <T>` 是自适应二元仿函数，是单参数的仿函数。若 `f` 是仿函数的对象，并且 `x` 属于型别 `T`，则 `f(x)` 会返回 `(-x)`。求反仿函数的基类是 `unary_function <T, T>`。该仿函数类包括 4 个成员函数：
- 1) `negate::argument_type`。该成员函数代表参数的型别 `T`。
 - 2) `negate::result_type`。该成员函数代表执行结果的型别 `T`。
 - 3) `negate::operator()` (`const T& x`) `const`。该成员函数代表函数调用时的操作符。其执行结果是 `(-x)`。
 - 4) `negate::negate()`。该成员函数代表该仿函数类的默认构造函数。

例 13-17

```
#include <iostream>
#include <functional>
#include <algorithm>
#include <vector>
using namespace std;
void myprint(vector<int> &vd)
{
    vector<int>::iterator it;
    for(it=vd.begin();it!=vd.end();it++)
        cout << *it << ", ";
    cout << endl;
}
void main(int argc, char* argv[])
{
    const int N=6;
    vector<int> V(N),V2(N),V3(N);
    generate(V.begin(),V.end(),rand);
    cout << "vector v: " << endl;
    myprint(V);
    transform(V.begin(),V.end(),V2.begin(),bind2nd(modulus<int>(),10));
    cout << "vector v2: " << endl;
    myprint(V2);
    transform(V.begin(),V.end(),V3.begin(),negate<int>());
    cout << "vector v3: " << endl;
    myprint(V3);
}
```

例 13-17 的执行结果为：

```
vector v:  
41, 18467, 6334, 26500, 19169, 15724,  
vector v2:  
1, 7, 4, 0, 9, 4,  
vector v3:  
-41, -18467, -6334, -26500, -19169, -15724,
```

13.7 其他类型的仿函数

除了上述最常见、最简单的诸多仿函数之外，C++ STL 还提供了一些比较特殊的仿函数。这些仿函数主要有 7 个。它们分别是：

- `identity <T>`
- `project1st <Arg1, Arg2>`
- `project2st <Arg1, Arg2>`
- `select1st <pair>`
- `select2nd <pair>`
- `hash <T>`
- `subtractive_rng <T>`

13.7.1 证和映射

基本的“证”仿函数是 `identity`，其他均为一般化形式。C++ 标准没有涵盖 `identity` 和映射两种操作行为，但由于它们使用得非常普遍，多数 C++ 版本中添加了它们的定义。

1. 仿函数 `identity`

```
identity<T>
```

类 `identity` 是一种一元仿函数，用于表示“证”仿函数，使用时需要一个参数，返回的是未经任何变化的原参数。

例 13-18

```
#include <iostream>  
#include <functional>  
#include <assert.h>  
using namespace std;  
void main(int argc, char * argv[])  
{  
    int x=137;  
    identity<int> id;  
    assert(x == id(x));  
    cout << id(x) << endl;  
}
```

例 13-18 的执行结果为：



提示 在 Visual Studio 2008 环境下, 为确保程序能够顺利通过, 需要将头文件 `<yvals.h>` 中的宏 `HAS_TRADITIONAL_STL` 的值修改为 1, 编译程序才能顺利通过。13.7 节和 13.8 节中的内容均做相同处理。

2. 仿函数 `project1st` 和 `project2nd`

仿函数 `project1st` 的声明形式为:

```
project1st <Arg1, Arg2 >
```

仿函数 `project2nd` 的声明形式为:

```
project2nd <Arg1, Arg2 >
```

这两个仿函数均接受两个参数。`project1st` 返回第一个参数并忽略第二参数; `project2nd` 返回第二个参数并忽略第一个参数。这两个仿函数的基类均为 `binary_function <Arg1, Arg2, Arg2 >`。这两个仿函数类均包括 5 个成员函数:

- `first_argument_type`
- `second_argument_type`
- `result_type`
- `Arg1 operator () (const Arg1 & x, const Arg2 & y) const / Arg2 operator () (const Arg1 & x, const Arg2 & y) const`
- `project1st () / project2nd ()`

在使用上述两个仿函数时, 需要包含头文件 `<functional >`。

例 13-19

```
#include <iostream>
#include <vector>
#include <functional>
#include <algorithm>
#include <assert.h>
using namespace std;
void myprint(vector<int> &vi)
{
    vector<int>::iterator it;
    for(it=vi.begin();it!=vi.end();it++)
    {
        cout<<*it<<" ";
    }
    cout<<endl;
}
void main(int argc, char * argv[])
{
    vector<int> v1(5,137);
    vector<char*> v2(5,(char*)0);
    vector<int> result(5);
    transform(v1.begin(),v1.end(),v2.begin(),result.begin(),project1st<int, char*>());
```

```

assert (equal (v1.begin(), v1.end(), result.begin()));
myprint (result);
transform (v2.begin(), v2.end(), v1.begin(), result.begin(), project2nd < char * , int > ());
assert (equal (v1.begin(), v1.end(), result.begin()));
myprint (result);
}

```

例 13-19 的执行结果为:

```

137 , 137 , 137 , 137 , 137 ,
137 , 137 , 137 , 137 , 137 ,

```

3. 仿函数 select1st 和 select2nd

仿函数 `select1st < pair >` 和 `select2nd < pair >` 可以接受单一参数, 参数是 `pair` 类型。仿函数 `select1st` 返回该 `pair` 的第一个元素; 仿函数 `select2nd` 返回该 `pair` 的第二个元素。这两个仿函数也定义于头文件 `< functional >` 中。这两个仿函数的基类是: `unary_function < pair, typename pair::first_type >`

这两个仿函数类均包括 4 个成员函数:

- `argument_type`
- `result_type`
- `const typename pair::first_type&select1st::operator() (const pair& p) const`
- `select1st()/select2nd()`

例 13-20

```

#include <iostream>
#include <functional>
#include <algorithm>
#include <map>
using namespace std;
void main(int argc, char * argv[])
{
    map<int, double> M;
    M[1] = 0.3;
    M[47] = 0.8;
    M[33] = 0.1;
    cout << "抽取键值:";
    transform (M.begin(), M.end(), ostream_iterator<int> (cout, " "),
               select1st<map<int, double>::value_type> ());
    cout << endl;
    cout << " 抽取实值:";
    transform (M.begin(), M.end(), ostream_iterator<double> (cout, " "),
               select2nd<map<int, double>::value_type> ());
    cout << endl;
}

```

例 13-20 的执行结果为:

抽取键值: 1 33 47

抽取实值: 0.3 0.1 0.8

13.7.2 仿函数 hash 和 subtractive_rng

1. 仿函数 hash

仿函数类 `hash < T >` 是一种散列函数。STL 中的所有散列关联容器 (Hashed Associative Containers) 均使用其作为默认的散列函数。

仿函数模板 `template hash < T >` 仅针对 `template` 参数型别为 `char *`、`const char *`、`string` 和 `integer` 而定义。若搭配不同的参数型别, 需要提供自定义的 `template specialization`, 或提供新的 `hash` 函数类。

使用仿 `hash()` 函数需要包含头文件 `< hash_set >` 和 `< hash_map >`。

仿函数没有基类。

该类的成员函数为 `size_t hash::operator() (const T& x) const`

运算符函数会返回参数 `x` 的散列值。

2. 仿函数 subtractive_rng

仿函数类 `subtractive_rng < T >` 是一种随机数发生器 (Random number Generator)。使用减去法可以产生拟真乱数。仿函数是一种一元函数。通常参数可以是无符号整型数, 函数可以返回一个小于该数值的无正负号整数。若连续调用同一个 `subtractive_rng` 对象, 会产生一个拟真乱数序列。

《泛型编程与 STL》一书指出, `subtractive_rng` 产生的数列完全是可决定的, 由两个不同的 `subtractive_rng` 对象所产生的数列则互不相干。`subtractive_rng` 产生的数值取决于其种子以及之前被调用的次数。

仿函数的基类是 `unary_function < unsigned int , unsigned int >`

该仿函数类包含 6 个成员函数:

1) `argument_type`。该成员函数代表参数型别。

2) `result_type`。该成员函数代表执行结果的型别 `unsigned int`。

3) `subtractive_rng (unsigned int seed)`。该成员函数代表仿函数类的构造器。

4) `subtractive_rng()`。该成员函数代表默认构造器。

5) `unsigned int operator() (unsigned int seed)`。该成员函数用于重新设定乱数产生器, 使其内部状态恰好与“以 `seed` 值构造”的方式相同。

13.8 适配器

STL 标准库提供了一组函数适配器, 用来特殊化或扩展一元和二元函数对象。适配器是特殊的类, 可以被分成以下 4 种类型:

1. 绑定器

绑定器通过把二元函数对象的一个实参绑定到一个特殊值上, 将其转换成一元函数对象。C++ 标准程序库提供了两种预定义的绑定器适配器: `bind1st` 和 `bind2nd`。`bind1st` 把值绑定到二元函数对象的第一个实参上; `bind2nd` 把值绑定在第二个实参上。绑定器非常有用,

其使用也非常灵活。

2. 取反器

取反器是将函数对象的值反转的函数适配器。C++ 标准程序库提供了两个预定义的取反适配器：not1 和 not2。not1 翻转一元预定义函数对象的真值；not2 翻转二元谓词函数的真值。取反器的定义和实现遵循成员函数适配器的模式。

3. 成员函数适配器

成员函数适配器使成员函数可以被用作算法的参数。当算法需要调用一个标准的操作或自定义操作时，成员函数的使用可能是非常简便的。例如，

```
void draw_all(list<Shape * > &c)
{
    for_each(c.begin(), c.end(), &Shape::draw);
}
```

4. 函数指针适配器

函数指针适配器使函数指针可以被用作算法的参数。头文件 <functional> 提供了两个适配器，使函数指针可以和标准算法一起使用。

本节将上述 4 种类型分成两组：成员函数适配器和其他适配器。

13.8.1 成员函数适配器

所谓成员函数适配器是一些小型的仿函数类。它们能够将成员函数作为仿函数来调用。每个适配器均需要一个型别为 T* 或 T& 的参数。通过该参数调用 T 类的成员函数。由此可知，成员函数是面向对象编程与泛型编程之间的桥梁。

无论是标准库，还是程序员自定义类的成员函数，其数量是巨大的。通常构造仿函数适配器是通过辅助函数，而不是通过构造函数。最普遍使用的两个辅助函数是：mem_fun() 和 mem_fun_ref()。

1. mem_fun_t() 和 const_mem_fun_t() 函数

类 mem_fun_t 是一种成员函数适配器，和所有仿函数具有同样的特点：拥有一个 operator() 函数，使得 mem_fun_t() 可用一般的函数调用语法被调用。通常情况下，类 mem_fun_t 的 operator() 函数接受型别 T* 的参数。

和其他适配器一样，直接使用类 mem_fun_t 的构造函数是不方便的，但可以使用辅助 mem_fun() 函数来代替之。

使用仿函数 mem_fun_t 时，需要包含头文件 <functional>。仿函数的基类为：unary_function<T*, R>

仿函数 const_mem_fun_t 和仿函数 mem_fun_t 大体相似。

例 13-21

```
#include <iostream>
#include <functional>
#include <vector>
#include <algorithm>
using namespace std;
```

```

struct B{
    virtual void print () =0;
};
struct D1:public B{
    void print () {
        cout << "I'm a D1. " << endl;
    }
};
struct D2:public B{
    void print () {
        cout << "I'm a D2. " << endl;
    }
};
void main(int argc, char* argv[])
{
    vector<B* > v;
    v.push_back(new D1);
    v.push_back(new D2);
    v.push_back(new D2);
    v.push_back(new D1);
    for_each(v.begin(),v.end(),mem_fun(&B::print));
    vector<vector<int>* > v2;
    v2.push_back(new vector<int>(5));
    v2.push_back(new vector<int>(3));
    v2.push_back(new vector<int>(4));
    transform(v2.begin(),v2.end(),ostream_iterator<int>(cout," "),mem_fun(&vector<int>::size));
    cout << endl;
}

```

例 13-21 的执行结果为:

```

I'm a D1.
I'm a D2.
I'm a D2.
I'm a D1.
5 3 4

```

2. 仿函数 mem_fun_ref_t 和 const_mem_fun_ref_t

仿函数 `mem_fun_ref_t<R, X>` 也是一种成员函数适配器。若 `X` 是类, 且具有成员函数 `RX::f()`, 那么仿函数 `mem_fun_ref_t<R, X>` 是一个仿函数适配器, 就可以方便地以一般函数的形式来调用 `f()`。在使用该仿函数时, 同样需要包含头文件 `<functional>`, 在实际使用过程中, 通常会以辅助 `mem_fun_ref()` 函数代替它。

`const_mem_fun_ref_t` 和 `mem_fun_ref_t` 大体近似。

例 13-22

```
#include <iostream>
```

```
#include <functional>
#include <vector>
#include <algorithm>
using namespace std;
struct B{
    virtual void print () =0;
};
struct D1:public B{
    void print () {
        cout << "I'm a D1. " << endl;
    }
};
struct D2:public B{
    void print () {
        cout << "I'm a D2. " << endl;
    }
};
void main(int argc, char* argv[])
{
    vector <D1 > v;
    v.push_back(D1());
    v.push_back(D1());
    for_each(v.begin(),v.end(),mem_fun_ref(&B::print));
    vector <vector <int >> v2;
    v2.push_back(vector <int > (2));
    v2.push_back(vector <int > (7));
    v2.push_back(vector <int > (3));
    transform(v2.begin(),v2.end(),ostream_iterator <int > (cout," "),mem_fun_ref(&vector <int >::size));
    cout << endl;
}
```

例 13-22 的执行结果为:

```
I'm a D1.
I'm a D1.
2 7 3
```

3. 仿 mem_fun1_t 函数和 const_mem_fun1_t

仿 mem_fun1_t 函数 <R, X, A> 也是一种成员函数适配器。若 X 是类, 且具有成员函数 RX::f(A), 那么 mem_fun1_t <R, X, A> 即是一种仿函数适配器, 从而可以像一般函数那样来调用 f() 函数。仿 mem_fun1_t 函数需要两个参数: 第一个参数型别为 X*, 第二个参数型别为 A。由于仿 mem_fun1_t 函数的构造器不便于使用, 通常以辅助 mem_fun() 函数代替之。

const_mem_fun1_t 和 mem_fun1_t 大体相似。

例 13-23

```
#include <iostream>
```

```
#include <functional>
#include <vector>
#include <algorithm>
using namespace std;
struct operation{
    virtual double eval(double) =0;
};
struct square:public operation{
    double eval(double x){
        return x * x;
    };
};
struct Negate:public operation{
    double eval(double x)
    {
        return -x;
    }
};
struct B{
    virtual int f(int x) const =0;
};
struct D:public B{
    int val;
    D(int x):val(x)
    {

    };
    int f(int x) const
    {
        return val + x;
    };
};
void main(int argc, char * argv[])
{
    vector<operation * > operations;
    vector<double> operands;
    operations.push_back(new square);
    operations.push_back(new square);
    operations.push_back(new square);
    operations.push_back(new square);
    operations.push_back(new square);
    operands.push_back(1);
    operands.push_back(2);
    operands.push_back(3);
    operands.push_back(4);
    operands.push_back(5);
```

```
transform(operations.begin(), operations.end(), operands.begin(),
         ostream_iterator<double>(cout, "\n"),
         mem_fun(&operation::eval));
cout << endl;
vector<B * > v;
v.push_back(new D(3));
v.push_back(new D(4));
v.push_back(new D(5));
int A[3] = {7, 8, 9};
transform(v.begin(), v.end(), A, ostream_iterator<int>(cout, " "), mem_fun(&B::f));
cout << endl;
}
```

例 13-23 的执行效果如图 13-2 所示。

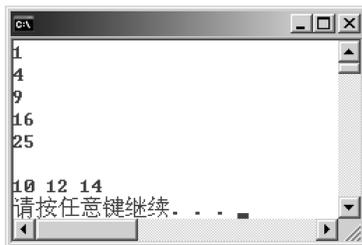


图 13-2 例 13-23 的执行效果

4. 仿函数 mem_fun1_ref_t 和 const_mem_fun1_ref_t

仿 mem_fun1_ref_t<R, X, A> 函数也是一种成员函数适配器。若 X 是类，且具有成员函数 RX::f(A)；那么 mem_fun1_ref_t<R, X, A> 即是一个仿函数适配器，从而可以一般函数的形式来调用 f()。作为仿函数，mem_fun1_ref_t 需要两个参数：第一个参数型别为 X&，第二个参数的型别为 A。通常使用辅助 mem_fun_ref() 函数代替使用其构造器的方式。

仿函数 const_mem_fun1_ref_t 和 mem_fun1_ref_t 大体相似。

例 13-24

```
#include <iostream>
#include <functional>
#include <algorithm>
#include <vector>
using namespace std;
void myprint(vector<vector<int>> &vv)
{
    vector<vector<int>>::iterator it;
    vector<int>::iterator itv;
    for(it = vv.begin(); it != vv.end(); it++)
    {
        for(itv = it->begin(); itv != it->end(); itv++)
```

```

        cout << * itv << " , ";
        cout << endl;
    }
    cout << endl;
}
void main(int argc, char * argv[])
{
    int A1 [5] = {1,2,3,4,5};
    int A2 [5] = {1,1,2,3,5};
    int A3 [5] = {1,4,1,5,9};
    int A4 [5] = {1,5,7,11,13};
    vector <vector <int >> v;
    v.push_back(vector <int > (A1,A1 +5));
    v.push_back(vector <int > (A2,A2 +5));
    v.push_back(vector <int > (A3,A3 +5));
    v.push_back(vector <int > (A4,A4 +5));
    myprint(v);
    int indices[4] = {0,2,4,4};
    int & (vector <int > :: * extract) (vector <int > ::size_type);
    extract = &vector <int > ::operator[];
    transform(v.begin(), v.end(), indices, ostream_iterator <int > (cout, " "), mem_fun_ref(ex-
tract));
    cout << endl;
}

```

例 13-24 的执行结果为:

```

1 , 2 , 3 , 4 , 5 ,
1 , 1 , 2 , 3 , 5 ,
1 , 4 , 1 , 5 , 9 ,
1 , 5 , 7 , 11 , 13 ,

```

```

1 2 9 13

```

例 13-25

```

#include <iostream>
#include <functional>
#include <algorithm>
#include <vector>
using namespace std;
struct B{
    virtual int f(int x) const =0;
};
struct D :public B{
    int val;
    D(int x) : val(x)
    {

```

```

    }
    int f(int x) const
    {
        return val + x;
    }
};
void main(int argc, char * argv[])
{
    vector<D> v;
    v.push_back(D(3));
    v.push_back(D(4));
    v.push_back(D(5));
    int A[3] = {7,8,9};
    transform(v.begin(), v.end(), A, ostream_iterator<int>(cout, " "), mem_fun_ref(&B::f));
    cout << endl;
}

```

例 13-25 执行结果为:

10 12 14

13.8.2 其他适配器

本小节主要讲述 8 种适配器: binder1st、binder2nd、pointer_to_unary_function、unary_negate、pointer_to_binary_function、binary_negate、unary_compose 和 binary_compose。以上 8 种适配器可以被分成 4 组: binder1st 和 binder2nd、pointer_to_unary_function 和 pointer_to_binary_function、unary_negate 和 binary_negate、unary_compose 和 binary_compose。

1. 仿函数 binder1st 和 binder2nd

仿 binder1st() 函数是一种仿函数适配器, 可用于将自适应二元函数转换成自适应一元函数。若 f 是仿函数 binder1st<BinaryFun> 的对象, 则 f(x) 返回 F(c, x), 其中 F 是 BinaryFun 对象, c 是常量。F 和 c 都被当成参数传递给 binder1st() 的构造函数。

通常使用仿 binder1st() 函数, 不是使用其构造函数, 而是使用其辅助 binder1st() 函数。通过将双参函数的第一个参数设置成某个常量, 形成单参函数。

使用仿函数 binder1st 时, 需要包含头文件 <functional>。该仿函数的基类是

```
unary_function<typename BinaryFun::second_argument_type, typename BinaryFun::result_type>
```

仿函数 binder1st 的声明形式为:

```

template<class Operation> class binder1st : public unary_function
    < typename Operation::second_argument_type, typename Operation::result_type >
{
public:
    typedef typename Operation::argument_type argument_type;
    typedef typename Operation::result_type result_type;

```

```

        binder1st( const Operation & _Func, const typename Operation::first_argument_type &
Left );

        result_type operator() ( const argument_type & _Right ) const;
        result_type operator() ( const argument_type & _Right ) const;
protected:
        Operation op;
        typename Operation::first_argument_type value;
};

```

仿函数类 `binder1st` 具有 5 个成员函数:

1) `argument_type`。该成员函数代表参数型别 `BinaryFun::second_argument_type`。

2) `result_type`。该成员函数代表参数型别 `BinaryFun::result_type`。

3) `result_type operator() (const argument_type& x) const`。该成员函数代表函数调用的操作符。其返回值为 `F (c, x)`，其中 `F` 和 `C` 是 `binder1st` 对象构造时的参数。

4) `binder1st (const BinaryFun& F, typename BinaryFun::first_argument_type c)`。该成员函数代表辅助函数，用于产生一个 `binder1st` 对象。若 `F` 的型别是 `BinaryFun`，则 `binder1st (F, c)` 不但等价于 `binder1st < BinaryFun > (F, c)`，并且更方便。型别 `T` 必须转换为 `BinaryFun` 的第一参数型别。

同样，仿 `binder2nd()` 函数也是一种仿函数适配器，可用于将自适应二元函数转换成自适应一元函数。若 `f` 是仿函数类 `binder2nd < BinaryFun >` 的对象，则 `f (x)` 会返回 `F (x, c)`，其中 `F` 为 `BinaryFun` 对象，`c` 是常量。`F` 和 `c` 均被作为参数传给 `binder2nd` 的构造器。直观上来讲，通过将双参函数的第二参数设定为常量，形成单参函数。

使用仿函数 `binder2nd` 时，也需要包含头文件 `<functional >`。

该仿函数的基类是

```

unary_function < typename > BinaryFun::first_argument_type, typename Bi-
naryFun::result_type >

```

仿函数 `binder2nd` 的声明形式为:

```

template < class Operation > class binder2nd : public unary_function
    < typename Operation::first_argument_type, typename Operation::result_type >
{
public:
    typedef typename Operation::argument_type argument_type;
    typedef typename Operation::result_type result_type;
    binder2nd( const Operation & _Func, const typename Operation::second_argument_type & _Right
);

    result_type operator() ( const argument_type & _Left ) const;
    result_type operator() ( argument_type & _Left ) const;
protected:
    Operation op;
    typename Operation::second_argument_type value;
};

```

类 `binder2nd` 和类 `binder1st()` 相似，也包含 5 个成员函数。

由以上内容可知，这两个函数的区别在于：函数的第二个参数是不同的。而函数 `binder1st` 的第二个参数是“**_Left**”；而函数 `binder2nd` 的第二个参数是“**_Right**”。

在处理谓词时，这两个函数也非常有意义——可以将二元谓词转变成一元谓词，故常用于某范围内的数值与特定值之间的比较。

例 13-26

```
#include <iostream>
#include <functional>
#include <algorithm>
#include <list>
using namespace std;
void myprint(list<int> lt)
{
    list<int>::iterator it;
    for(it = lt.begin(); it != lt.end(); it++)
        cout << * it << " , ";
    cout << endl;
}
void main(int argc, char * argv[])
{
    list<int> L;
    int i = 0;
    for (i = 0; i < 10; i++)
    {
        L.push_back(rand() % 3);
    }
    myprint(L);
    list<int>::iterator first_nonzero = find_if(L.begin(), L.end(), binder1st(not_equal_to<int>
    (), 0));
    cout << * first_nonzero << endl;
    L.clear();
    for(i = 0; i < 10; i++)
    {
        L.push_back(rand() % 4 - 3);
    }
    myprint(L);
    list<int>::iterator two_pos = find_if(L.begin(), L.end(), binder2nd(greater<int>(), -1));
    cout << * two_pos << endl;
    //return 0;
}
```

例 13-26 的执行结果为：

```
2 , 2 , 1 , 1 , 2 , 1 , 0 , 0 , 1 , 2 ,
2
```

```
-2, -2, -2, 0, -2, 0, 0, -1, 0, -3,
0
```

例 13-27

```
#include <iostream>
#include <algorithm>
#include <functional>
#include <list>
using namespace std;

void main(int argc, char * argv[])
{
    int iarray[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    list<int> aList(iarray, iarray + 10);
    int k = 0;
    k = count_if(aList.begin(), aList.end(), bind1st(greater<int>(), 8));
    cout << "bind1st test: " << k << endl;
    k = count_if(aList.begin(), aList.end(), bind2nd(greater<int>(), 8));
    cout << "bind2nd test: " << k << endl;
}
```

例 13-27 的执行结果为：

```
bind1st test: 7
```

```
bind2nd test: 2
```

上述例题中，两个仿函数的作用是不同的。bind1st 用于统计小于 8 的数值的个数 ($q < 8$)；而 bind2nd 用于统计大于 8 的数值个数。

通俗地讲，对于某范围内的每个元素用字符 q 表，存在比较表达式 $a > b$ ，程序中给定的值为 val ，如果使用函数 **bind1st**，内部表达式是 $val > q$ ；如果使用函数 **bind2nd**，内部表达式是 $q > val$ 。

2. 仿函数 pointer_to_unary_function 和仿函数 pointer_to_binary_function

仿函数 pointer_to_unary_function 的声明形式如下：

```
template<class Arg, class Result> class pointer_to_unary_function : public unary_function<Arg,
Result>
{
public:
    explicit pointer_to_unary_function( Result (* _pfunc)(Arg) );
    Result operator()( Arg Left ) const;
};
```

仿函数 pointer_to_unary_function 是一种仿函数适配器，允许将函数指针 $result (* f)(Arg)$ 视为一个自适应一元函数。值得说明的是，表示某函数的函数指针应该是一个相当不错的一元仿函数，可以传入任何的 STL 算法中。多数情况下，不会直接使用仿函数的构造器，而是使用其辅助 ptr_fun() 函数。例如，

```
transform(first, last, first, fabs);
```

上述语句可以将 [first, last] 中的元素求取绝对值，并使用该绝对值序列将原序列取代。

仿函数 `pointer_to_binary_function` 也是一种仿函数适配器，允许将函数指针 `Result (*f)(Arg1, Arg2)` 视为一个自适应二元函数。若 `F` 是一个仿函数 `pointer_to_binary_function <Arg1, Arg2, result>` 对象，以型别 `Result (*) (Arg1, Arg2)` 的函数指针 `f` 作为初值，则 `F(x, y)` 会调用函数 `f(x, y)`。例如，

```
list<char * <::iterator item;
item=find_if(L.begin(), L.end(), not1(binder2nd(ptr_fun(strcmp),"OK")));
```

对于仿函数 `pointer_to_binary_function`，最常使用的也是其辅助函数 `ptr_fun()`。

例 13-28

```
#include <vector>
#include <algorithm>
#include <functional>
#include <cstring>
#include <iostream>

int main()
{
    using namespace std;
    vector<char*> v1;
    vector<char*>::iterator Iter1, RIter;

    v1.push_back("Open");
    v1.push_back("up");
    v1.push_back("the");
    v1.push_back("pearly");
    v1.push_back("gates");

    cout << "Original sequence contains: ";
    for(Iter1 = v1.begin(); Iter1 != v1.end(); Iter1++)
        cout << *Iter1 << " ";
    cout << endl;

    // To search the sequence for "pearly"
    // use a pointer_to_function conversion
    RIter = find_if(v1.begin(), v1.end(), not1(bind2nd(ptr_fun(strcmp), "pearly")));

    if(RIter != v1.end())
    {
        cout << "The search for 'pearly' was successful. \n";
        cout << "The next character string is: "
            << *++RIter << ". " << endl;
    }
}
```

例 13-28 的执行结果为:

```
Original sequence contains: Open up the pearly gates
The search for 'pearly' was successful.
The next character string is: gates.
```

3. 仿函数 unary_negate 和 binary_negate

无论是 not1() 还是 not2(), 其返回结果均为取“非”运算。

仿函数 unary_negate 是自适应谓词, 用于表示其他自适应谓词的逻辑负值。若 f 是一个该仿函数的对象, 构造时以 pred 作为其底部的仿函数, 则 f(x) 会返回! pred(x)。通常使用辅助函数 not1() 实现该仿函数的功能, 例 not1(pred)。使用该辅助函数时, 需要包含头文件 <functional>。

仿函数 binary_negate 也是自适应谓词, 用于表示其他自适应二元谓词的逻辑负值。若 f 是二元负 <Predicate> 对象, 构造时以 pred 作为其底层的仿函数, 则 f(x, y) 会返回! pred(x, y)。此仿函数的辅助函数是 not2()。使用该辅助函数时, 同样需要包含头文件 <functional>。



提示 not1() 可实现对单个参数表达式逻辑值的取反; not2() 可实现对两个参数表达式逻辑值的取反。

例 13-29

```
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>
using namespace std;
void myprint(vector<int> &v)
{
    vector<int>::iterator it;
    for(it=v.begin();it!=v.end();it++)
    {
        cout<<*it<<" ";
    }
    cout<<endl;
}
void main(int argc, char * argv[])
{
    vector<int> v1;
    vector<int>::iterator iter;
    int i;
    for(i=0;i<=6;i++)
    {
        v1.push_back(5*i);
```

```
    }  
    cout << "vector v1:" << endl;  
    myprint(v1);  
    vector<int>::iterator::difference_type res1;  
    res1 = count_if(v1.begin(), v1.end(), bind2nd(greater<int>(), 10));  
    cout << "v1 中大于的元素个数: " << res1 << ". " << endl;  
    vector<int>::iterator::difference_type res2;  
    res2 = count_if(v1.begin(), v1.end(), not1(bind2nd(greater<int>(), 10)));  
    cout << " v1 中'不'大于的元素个数: " << res2 << ". " << endl;  
}
```

例 13-29 的执行结果为:

```
vector v1:  
0, 5, 10, 15, 20, 25, 30,  
v1 中大于 10 的元素个数: 4.  
v1 中'不'大于 10 的元素个数: 3.
```

例 13-30

```
#include <vector>  
#include <algorithm>  
#include <functional>  
#include <cstdlib>  
#include <iostream>  
using namespace std;  
void myprint(vector<int> &v)  
{  
    vector<int>::iterator it;  
    for(it=v.begin();it!=v.end();it++)  
    {  
        cout << *it << ", ";  
    }  
    cout << endl;  
}  
int main()  
{  
    vector<int> v1;  
    vector<int>::iterator Iter1;  
  
    int i;  
    v1.push_back(6262);  
    v1.push_back(6262); //程序会发出异常  
    for(i=0;i<4;i++)  
    {  
        v1.push_back(rand());  
    }  
}
```

```

cout << "Original vector v1:" << endl ;
myprint(v1);
sort(v1.begin(), v1.end());
cout << "Sorted vector v1:" ;
myprint(v1);
sort(v1.begin(), v1.end(), not2(less_equal<int>()));
cout << "Resorted vector v1:" ;
myprint(v1);
}

```

例 13-30 的执行结果为:

```

Original vector v1:
6262 , 41 , 18467 , 6334 , 26500 ,
Sorted vector v1:41 , 6262 , 6334 , 18467 , 26500 ,
Resorted vector v1:26500 , 18467 , 6334 , 6262 , 41 ,

```

4. 仿函数 unary_compose 和 binary_compose

在 13.3 节中讲述辅助用仿函数时, 表 13-2 中列出了两个辅助用 compose1() 和 compose2() 函数。此处, 再略加介绍。

仿函数 unary_compose 的基类是: unary_compose<Fun1, Fun2>

仿函数类 unary_compose 是一种仿函数适配器。若 f 和 g 均是自适应一元函数, 并且 g 的返回型别可转换为 f 的参数型别, 则 unary_compose 可用于产生一个仿函数对象 h, 从而 $h(x) = f(g(x))$ (此种行为称为函数合成。函数合成是代数学的重要概念, 即可以使用单纯的仿函数任意构造出复杂的仿函数)。该仿函数的辅助函数是 compose1()。

仿函数 binary_compose 也是一种仿函数适配器。若 f 是自适应二元函数, g1() 和 g2() 均为自适应一元函数, 并且 g1() 和 g2() 的返回型别均可转换为 f() 的参数型别, 仿函数 binary_compose 可用于产生一个仿函数 h, 并且 $h(x) = f(g1(x), g2(x))$ 。该仿函数的辅助函数是 compose2()。

使用上述两个仿函数时, 需要包含头文件 <functional>。

例 13-31

```

#include <vector>
#include <functional>
#include <algorithm>
#include <list>
#include <iostream>
#include <assert.h>
#include <cmath>
using namespace std;
template<typename T> void myprint(vector<T> & vd)
{
    vector<T>::iterator it;
    for(it = vd.begin(); it != vd.end(); it++)
    {

```

```
        cout << * it << " , ";
    }
    cout << endl;
}
template <typename T > void myprintL(list <T > & l)
{
    list <T > ::iterator it;
    for(it = l.begin(); it! = l.end(); it++)
    {
        cout << * it << " , ";
    }
    cout << endl;
}
void main(int argc, char * argv[])
{
    cout << "Test compose1 () and compose2 () :" << endl;
    double angleA[5] = {60,45,90, -30, -180};
    double sinesS[5] = {0,0,0,0,0};
    vector <double > angle;
    vector <double > sines;
    sines.assign(sinesS,sinesS + 5);
    angle.assign(angleA,angleA + 5);
    cout << "Original Vector sines: " << endl;
    myprint(sines);
    cout << "Original Vector angle: " << endl;
    myprint(angle);
    const double pi = 3.1415926;
    assert(sines.size() >= angle.size());
    transform(angle.begin(), angle.end(), sines.begin(),
        compose1(negate <double > (),
        compose1(ptr_fun(sin),
        bind2nd(multiplies <double > (),pi/180.0)))));
    cout << " Vector sines: " << endl;
    myprint(sines);
    int larray[6] = {1,2,6,15,10,3};
    double la[6] = {0.1,0.1,0.1,0.1,0.1,0.1};
    list <int > li(larray,larray + 6);
    list <double > li2(la,la + 6);
    cout << "Original list li :" << endl;
    myprintL(li);
    cout << "Original list li2 :" << endl;
    myprintL(li2);
    list <int > ::iterator it1;
    it1 = find_if(li.begin(), li.end(), compose2(logical_and <bool > (),
        bind2nd(greater_equal <int > (),1),
        bind2nd(less_equal <int > (),10)));
```

```

assert(it1 == li.end() || (* it1 >= * it1 <= 10));
cout << "[1,10]之内的第一个元素:" << endl;
cout << * it1 << endl;
transform(li.begin(), li.end(), li2.begin(),
          compose2(divides < double > (),
                  ptr_fun(sin),
                  bind2nd(plus < double > (), 0))); //计算 sin(x)/(x+5)
cout << "list li2 :" << endl;
myprintL(li2);
}

```

例 13-31 的执行结果为:

```

Test compose1() and compose2():
Original Vector sines:
0, 0, 0, 0, 0,
Original Vector angle:
60, 45, 90, -30, -180,
Vector sines:
-0.866025, -0.707107, -1, 0.5, 5.35898e-008,
Original list li:
1, 2, 6, 15, 10, 3,
Original list li2:
0.1, 0.1, 0.1, 0.1, 0.1, 0.1,
[1,10]之内的第一个元素:
1
list li2:
0.841471, 0.454649, -0.0465692, 0.0433525, -0.0544021, 0.04704,

```

13.9 小结

本章全面介绍了C++ STL的仿函数。本章是全书的精华部分，读者应认真学习，仔细体会。本章阐述了仿函数的概念，按用途对仿函数做了简要地概述。之后按具体功能的不同，对STL中的预定义仿函数做了全面介绍，本章主要涉及关系仿函数、逻辑仿函数、算术仿函数、证和映射仿函数以及hash和subtractive_rng仿函数。13.8节重点讲述了STL的各种适配器。

第 14 章

配 置 器

内存配置器（Allocator）简称为配置器。它代表一种特定的内存模型，并提供一种抽象概念，便于将内存的申请转变为对内存的直接调用。配置器主要用于将算法和容器与物理存储细节相隔离。每个配置器均提供了一套分配与释放存储的标准方式以及一套用于指针类型和引用类型的标准名字。配置器是一种纯粹的抽象概念。C++ STL 提供了一个标准配置器，旨在为程序员提供更好的服务。程序员还可以根据自己的需要设计和提供自定义的分配器。

标准容器和算法均通过配置器提供的功能获取和访问内存。通过提供新的配置器可为标准容器提供新的、不同的存储使用方式。

14.1 使用配置器

对于程序员而言，使用不同的配置器应该没有任何困难。在使用时，程序员仅需将配置器作为一个模板参数而已。例如，

使用特殊的配置器 `myAlloc`，用于产生特殊的容器和 strings：

```
std::vector<int, myAlloc<int>> v;  
std::map<int, float, less<int>, myAlloc<std::pair<const int, float>>> m;  
std::basic_string<char, std::char_traits<char>, myAlloc<char>> s;
```

还可以自定义一些配置器，例如，

```
typedef std::basic_string<char, std::char_traits<char>, myAlloc<char>> xstring;  
typedef std::map<xstring, xstring, less<xstring>, myAlloc<std::pair<const xstring, xstring>>> xmap;  
xmap map;
```

使用非标准的配置器分配的对象表面上没有任何的问题，但是，由于不同的配置器分配的元素是不能混淆的，因此可能会造成未定义行为。运算符“`operator ==`”可以用来比较两个配置器是否使用了相同的内存模型。如果返回 `true`，表示一个内存配置器的储存空间可以由另一个配置器收回。“以配置器为模板参数”的型别会提供一个 `get_allocator()` 函数，由此可以获得对应的配置器。例如，

```
if(mymap.get_allocator() == s.get_allocator())  
{  
    ...  
}
```

对于广大程序员而言，使用配置器可以实例化出容器和其他组件。例如实例化运用共享

内存的配置器，或允许 map 的元素储存在持久性数据库中。配置器提供一个接口，用于分配、生成、销毁和回收对象。通过配置器，容器和算法的元素存储方式才得以被参数化。基本的空间分配操作见表 14-1。

表 14-1 基本的空间分配操作

表 达 式	效 果
allocate (int num)	为 num 个元素分配内存
construct (p)	初始化 p 所指的元素
destroy	销毁 p 所指的元素
deallocate (p, num)	回收 p 所指的容纳 num 个元素的内存空间

例如，对于最简单的 vector 实例方案，配置器被作为模板参数或构造器的函数传递给 vector 型容器，并保存在其内部。vector 型容器的声明形式为（注意代码中的加黑字体）：

```
template < class Type, class Allocator = allocator <Type> > class vector
```

在 vector 型容器中，存在配置器类型的相关声明。其构造函数也包含了配置器的内容。

```
typedef typename Allocator::reference    reference;
typedef typename Allocator::const_reference const_reference;
typedef Allocator allocator_type;
typedef typename Allocator::pointer     pointer;
typedef typename Allocator::const_pointer const_pointer;
...
explicit vector(const Allocator &=Allocator());
explicit vector(size_type n, const T &value =T(), const Allocator &=allocator());
...
```

上述第二个构造函数中的参数 n 代表元素个数，参数 value 代表数值。配置器是由函数 allocator() 产生的。

对于一些未初始化的内存，STL 也提供了 3 个函数：uninitialized_fill()、uninitialized_fill_n() 和 uninitialized_copy()。这 3 个函数的使用较为方便。

- 1) uninitialized_fill (beg, end, val)。该函数以数值 val 初始化 [beg, end] 中的元素。
- 2) uninitialized_fill_n (beg, num, val)。该函数以数值 val 初始化 [beg, end] 中的 num 个元素。
- 3) uninitialized_copy (beg, end, mem)。该函数以 [beg, end] 中的各个元素初始化 mem 为起始地址的各个元素。

例如，可以在构造器中添加以下代码：

```
int size = n;
elems = alloc.allocate(num);           //elems 代表某种类型的指针
uninitialized_fill_n(elems, n, val);
...
```

此外，对于原始存储区，其迭代器也是比较复杂的问题。C++ STL 提供了 raw_storage_iterator 类，用于实现在未初始化的内存中访问并初始化。这样，使用该类定义的指针，即

可使用任何算法。例如，

```
copy(x.begin(), x.end(), raw_storage_iterator<T*, T>(elems));
```

模板 `raw_storage_iterator` 的第一个参数 `T*` 应是一个对应元素型别的输出型迭代器；第二个参数必须是元素型别。

对于临时缓冲区，程序开发过程中可能经常使用以下两个函数：

```
get_temporary_buffer()
return_temporary_buffer()
```

这两个函数可用来处理部分未初始化的内存，以便满足函数的短暂需求。

`get_temporary_buffer()` 函数返回的内存容量可能会比预期的少一些，其返回值是 `pair` 类型，内含所获的内存地址和内存容量。例如，

```
pair<mytype*, std::ptrdiff_t> p = get_temporary_buffer<mytype>(num);
```

如果 `pair` 的 `second` 值为 0，那么表示分配内存失败，即分配的容量为 0；如果 `pair` 的 `second` 小于 `num`，那么表示分配的内存不足够。`pair` 的 `first` 包含被分配的内存地址，如果不为 0，那么返回 `first` 的值，即返回了被分配的内存地址。

`get_temporary_buffer()` 和 `return_temporary_buffer()` 函数现已使用得较少了，故此处不再赘述。

14.2 C++ STL 默认的配置器（标准配置器）

在 C++ STL 的头文件 `<memory>` 中，标准配置器的声明形式如下：

```
namespace std{
    template <class T> class allocator{
    public:
        typedef size_t size_type;
        typedef ptrdiff_t difference_type;
        typedef T* pointer;
        typedef const T* const_pointer;
        typedef T& reference;
        typedef const T& const_reference;
        typedef T value_type;
        template <class U> struct rebind{
            typedef allocator<U> other;
        }
        pointer address(reference value) const;
        const_pointer address(const_reference value) const;
        allocator() throw();
        allocator(const allocator&) throw();
        template <class U> allocator(const allocator<U> &) throw();
        ~allocator() throw();
        size_type max_size() const throw();
        pointer allocator(size_type num, allocator<void>::const_pointer hint = 0);
```

```

void construct(pointer p, const T & value);
void destroy(pointer p);
void deallocator(pointer p, size_type num);
};
}

```

标准配置器使用全局 `operator new()` 和 `operator delete()` 函数分配和回收内存。`allocate()` 函数有可能会抛出 `bad_alloc` 类型的异常。标准配置器是可以优化的，但 `new()` 函数和 `delete()` 函数被调用的确切时机很难预测。内存配置器包含一个较特殊的模板结构定义 `rebind`。这种模板结构内存使配置器可间接为其他型别分配空间。例如，

```
Allocator::rebind<T2>::other
```

如果需要实例一个容器，必须为“非元素型别”的对象分配空间，此时 `rebind` 模板结构可以派上大用场。通常，需要使用配置为数组分配空间，元素型别为指针。

函数 `allocate (num)` 操作可以为 `n` 个对象分配空间，该空间可以由对应的函数 `deallocate (p, n)` 释放。`deallocate()` 函数以元素的个数 `num` 作为参数，在维持最少量的有关被分配存储的信息条件下，尽可能地优化配置器。同时，内存配置器可以要求程序员在调用 `deallocate()` 时能提供正确的 `num` 值。与运算符 `delete()` 不同的是，`deallocate()` 函数的参数必须“非 0”。

默认的配置使用 `operator new (size_t)` 获取内存，使用 `delete (void*)` 操作释放内存。这意味着内存耗尽时，若调用 `new_handler()`，会抛出 `std::bad_alloc` 的异常。

`allocate()` 函数不需要每次都调用低级的分配系统，更好的做法是让配置器管理关于内存的自由表，以期在最小的时间开销下分配存储空间。`allocate()` 的可选参数 `hint` 完全依赖于实现。尤其是在局部性特别重要的系统中，该函数为配置器提供分配内存的功能。

`allocator <void> ::pointer` 类型被作为通用指针类型，在标准容器中都使用 `void*`。用户在调用 `allocator()` 时，存在两种合理选择：

- 1) 没有提示。
- 2) 用一个指针对象的指针作为提示，该对象经常与新对象一起使用。

之所以使用配置器，是为了在分配容器的内存时不必直接和原始存储打交道。

需要说明的是，`allocator` 的操作均是基于 `pointer` 和 `reference` 两个类型定义表述的，这使用户可能提供其他替代类型来访问内存。在 C++ 语言中，不可能定义出完美的引用类型，因为语言和库的实现需要使用大量的类型定义来支持那些常规的基本数据类型。

每个程序可以有一个配置器，用以提供对持续性存储器的访问。另外，配置器可以像“长”指针一样，访问超出常规指针的寻址范围的主存。

常规用户为配置器提供了一种不寻常的指针类型，用于服务特殊的用途，但对引用无法做与此等价的事情。配置器的设计很容易处理使用模板参数描述的类型对象。多数容器的实现需要其他类型的对象。`rebind()` 类型可使一个配置器能分配任意类型的对象。

14.3 自定义配置器

实现一个配置器最主要的是：分配和回收存储空间。要达到这样的效果，程序员只需在

原有标准配置器的基础上修改配置器的几个成员函数即可，例如 `max_size()`、`allocator()` 和 `deallocator()`。程序员可以将自己在内存分配方面的策略体现在这 3 个函数内，例如重新运用内存、使用共享内存、将内存映射到面向对象的数据库内等。

容器的实现往往需要多次分配和释放对象。对于 `allocator()` 的朴素实现，意味着对运算符 `new` 的大量使用。但运算符 `new` 的效率并不是很高。若使用固定大小存储块的存储池，则可以使配置器的分配效率比常规的运算符 `new` 更高。

标准程序库对于标准容器所使用的配置器有限制：允许标准容器的实现将分配器类型的对象认为是等价的。尤其是在访问两个序列的操作时，更是需要检查被处理的对象是否拥有相同的分配器。

14.4 配置类的详细讨论

根据 C++ 国际标准规定，配置器需要提供以下型别定义和操作函数。若配置器被标准容器使用，则还需要一些特殊要求；否则，要求会少一些。

14.4.1 型别

`allocator::value_type`：元素型别，相等于类模板中的 `T`。例如 `allocator<T>`。

`allocator::size_type`：一个无正负号的整数型别，表示应用程序模型中最大对象的大小。

`allocator::different_type`：一个有正负号的整数型别，表示分配模型中两个指针的差距。如果配合标准容器使用，此型别必须等于 `ptrdiff_t`。

`allocator::pointer`：一个指向元素的指针型别。为配合标准容器的使用，此型别要等同于 `allocator<T>` 中的 `T*`。

`allocator::const_pointer`：一个指向元素的常数指针型别。若配合标准容器使用，此型别必须等同于 `allocator<T>` 中的 `const T*`。

`allocator::reference`：一个指向元素的 `reference` 型别。此型别要等同于 `allocator<T>` 中的 `T&`。

`allocator::const_reference`：一个指向元素的 `const reference` 型别。此型别要等同于 `allocator<T>` 中的 `const T&`。

`allocator::rebind`：这是一个模板结构体，可使配置器间接为其他型别分配空间。

14.4.2 配置类的成员函数

构造函数 `allocator::allocator()`：此函数是默认的构造函数，用于产生一个配置器对象。

构造函数 `allocator::allocator(const allocator& a)`：复制构造函数，产生一个配置器副本，使原有配置器分配的空间可通过另一个配置器回收。

析构函数 `~allocator::allocator()`：此函数用于销毁配置器对象。

`pointer allocator::address(reference value)`

`const_pointer allocator::address(const_reference value)`

第一个函数会返回一个非常数指针，指向一个非常数值；第二个函数会返回一个常数指

针，指向一个常数值。

`size_type allocator::max_size()`：此函数返回“对 `allocate()` 有意义的、用于分配空间”的最大许可值。

`pointer allocator::allocate (size_type num)`,

`pointer allocator::allocate (size_type num, allocator <void >::const_pointer hint)`;

上述两个函数均返回一块内存空间，可容纳 `num` 个型别为 `T` 的元素。元素不会被构造或初始化。第二个参数是可有可无的，真实意义由实例化的版本具体决定，可用于辅助提升效能。

`void allocator::deallocate (pointer p, size_type num)`：此函数用于释放空间，指针 `p` 所指的空间是由同一个配置器以 `allocate()` 分配的。`p` 不能是 `NULL` 或 `0`，且块内的元素必须已经被析构。

`void allocator::construct (pointer p, const T& value)`：此函数以参数 `value` 作为 `p` 所指的那个元素的初值，其作用相当于 `new ((void*) p) T (value)`。

`void allocator::destroy (pointer p)`：此函数用于销毁 `p` 所指的对象，但不回收空间，仅仅调用对象的析构函数。其作用相当于 `((T*) p) -> ~T()`。

`bool operator == (const allocator& a1, const allocator& a2)`：若配置器 `a1` 和 `a2` 是可以互换的，则此函数会返回 `true`。若某个配置器分配的空间由另一个配置器收回，则称这两者可以互换。若配合标准容器使用，则对于不同型别产生的配置器必须可以互换，此时函数应当返回 `true`。

`bool operator != (const allocator& a1, const allocator& a2)`：若配置器 `a1` 和 `a2` 不是可以互换的，则此函数返回 `true`。其作用相当于 `!(a1 == a2)`。若配合标准容器使用，则对相同型别产生的配置器彼此必须可互换，此时函数应当始终返回 `false`。

14.4.3 广义配置器

配置器是通过模板参数向容器传递信息的一种简化变形。容器中的每个元素均通过容器的分配器进行分配。若允许两个同类型的 `list` 采用不同的分配器，`splice()` 无法通过重新链接的方式实现。将 `splice()` 定义成基于元素复制，以防出现偶发情况。若允许配置器完全通用，则一个配置器能为任意类型的分配元素的 `rebind` 机制做得更加精细一些。标准配置器被假定为不在每个对象里存放数据，标准的实现可以利用该情况。

反对配置器在对象内存放信息的限制实际不是很严重。大部分配置器不需要在对象内存放数据。配置器可以保存有关配置器类型的数据。若需要各种数据，可以使用各种配置器类型。

配置器不允许在每个对象里存放数据的限制是强制性的。由于标准库在运行时间和空间效率上要求比较严格，因此当标准库在效率方面的限制不是很重要时，可以采用配置器技术。配置器还可以携带某一类信息，该信息从公共基类继承而来的。

配置器可以为容器提供一种控制，使容器的行为类似持续性存储器工作的高速缓存存储器，或者提供容器与其他对象之间的关联。

按照此方式，任何服务可能以常规容器操作透明的形式提供。最好的做法是将有关数据

存储的问题和数据使用的问题分开，后面不属于某个广义的配置器，但可通过单独的模板参数提供。

14.4.4 动态存储

```
class bad_alloc: public exception{ /* ... */
struct nothrow_t{};
extern const nothrow_t nothrow;
typedef void ( * new_handler )();
new_handler set_new_handle(new_handler new_p) throw();
void * operator new(size_t) throw(bad_alloc);
void operator delete(void * ) throw();
void * operator new(size_t , const nothrow_t&) throw();
void operator delete(void * , const nothrow_t &) throw();
void * operator new[] (size_t) throw(bad_alloc);
void operator delete[] (void * ) throw();
void * operator new[] (size_t , const nothrow_t &) throw();
void operator delete[] (void * , const nothrow_t &) throw();
void * operator new(size_t, void * p) throw(){return p};
void operator delete(void * p , void * ) throw(){};
void * operator new[] (size_t, void * p) throw(){return p};
void operator delete[] (void * p , void * ) throw(){};
```

动态存储用于实现 `new` 和 `delete` 运算符的功能。若头文件 `<new>` 中包含空异常描述的 `operator new()` 或 `operator new [] ()` 不能通过抛出 `std::bad_alloc` 异常发出存储耗尽的信号，则函数在分配失败时会返回 0。`new` 表达式将检测由带有空异常描述的分配函数返回的值。若返回值是 0，则不会调用构造函数并返回值 0，带有 `nothrow` 的配置器将返回 0 指明分配失败，而不抛出 `bad_alloc`。例如，

```
int * p=new int[1000]; //如果分配错误,会抛出异常
int * q= new(nothrow) int [1000]; //不会抛出异常
如果 q= =0,那么代表分配内存失败。
```

14.4.5 C 风格的分配

在头文件 `<cstdlib>` 中，读者可以找到以下语句：

```
void * malloc(size_t s); //分配 s 个字节
void * calloc (size_t n, size_t s); //分配 n 乘 s 个字节，初始化为 0
void free (void * p); //释放由 malloc () 或 calloc () 分配的空间
void * realloc (void * p, size_t s); //将 p 所指向的数组的大小变为 s，不能实现就分配 s 个字节，将
//p 所指数组复制过去并释放 p
```

这些函数现在已经很少使用。程序员应尽量使用 `new`、`delete` 和标准容器。上述函数处理的都是未初始化的存储。尤其是 `free()`，它不会对自己所释放的存储调用析构函数。实现 `new` 和 `delete` 有可能会使用这些函数，但不保证这样做。若使用 `realloc()`，并依赖于标准容器，通常会更简单有效。

C++ STL 还提供了一些函数，其目的在于高效地完成字节操作。由于 C 语言是通过 `char*` 指针访问无类型的字节，因此这些函数均存储在头文件 `<cstring>` 中。所有 `void*` 指针在这些函数中作为 `char*` 指针处理：

```
void* memcpy(void* p, const void* q, size_t n);
void* memmove(void* p, const void* q, size_t n);
```

和 `strcpy()` 函数一致，`memcpy()` 函数从 `q` 向 `p` 复制 `n` 个字节并返回 `p`。使用 `memmove()` 复制的区域可以有重叠；而 `memcpy()` 是假定区域，不能有重叠。

14.5 未初始化的内存

本节主要讲述 3 个辅助函数，即 `uninitialized_fill` (`ForwardIterator beg`, `ForwardIterator end`, `const T& value`)、`uninitialized_fill_n`() 和 `uninitialized_copy`()。这 3 个辅助函数均应用于未初始化内存之上。

1. `uninitialized_fill` (`ForwardIterator beg`, `ForwardIterator end`, `const T& value`)

此函数以数值 `value` 初始化范围 `[beg, end]` 内的元素。此函数要么执行成功，要么没有影响，通常实例化为：

```
namespace std{
template <class ForwIter, class T>
void uninitialized_fill(ForwIter beg, ForwIter end, const T & value)
{
typedef typename iterator_traits<ForwIter>::value_type VT;
ForwIter save(beg);
try{
for(; beg! =end; ++beg)
{
new(static_cast<void* > (&* beg))VT(value);
}
}
catch(...)
{
for(; save! =beg; ++save)
{
save -> ~VT();
}
throw();
}
}
}
```

2. `uninitialized_fill_n`()

此函数的原型为：

```
void uninitialized_fill_n(ForwardIterator beg, size num, const T & value);
```

此函数以数值 `value` 初始化从 `beg` 开始的 `num` 个元素。同样，此函数要么执行成功，要

么没有影响，通常实例化为：

```
namespace std{
    template <class ForwIter, class Size, class T >
    void uninitialized_fill_n(ForwIter beg, Size num, const T & value)
    {
        typedef typename iterator_traits<ForwIter>::values_type VT;
        ForwIter save(beg);
        try{
            for(; num - -; ++beg)
                {
                    save - > ~VT();
                }
            throw();
        }
    }
}
```

3. uninitialized_copy()

此函数的原型为：

```
ForwardIterator uninitialized_copy(InputIterator sourcebeg, InputIterator sourceend, ForwardIterator destbeg)
```

此函数以 [sourcebeg, sourceend] 内的元素为根据，对从 destbeg 起始的元素加以初始化。同样，函数要么执行成功，要么没有影响，通常实例化为：

```
namespace std{
    template <class InputIter, class ForwIter >
    ForwIter uninitialized_copy(InputIter beg, InputIter end, ForwIter dest)
    {
        typedef typename iterator_traits<ForwIter>::value_type VT;
        ForwIter save(dest);
        try{
            for(; beg! = end; ++beg, ++dest)
                {
                    new(static_cast<void * > (&* dest)VT(* beg));
                }
            return dest;
        }
        catch(...)
        {
            for(; save! = dest; ++save)
                {
                    save - > ~VT();
                }
        }
    }
}
```

14.6 配置器示例

下面使用例 14-1 来说明配置器的使用方法。

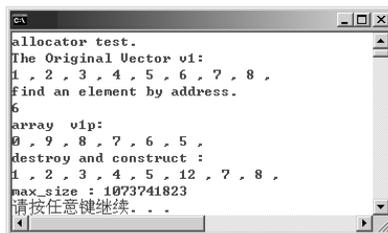
例 14-1

```
#include <iostream>
#include <memory>
#include <vector>
#include <algorithm>
using namespace std;
template <typename T> void myprint (vector<T> & vv) //输出 vector
{
    vector<T>::iterator it;
    for(it = vv.begin(); it != vv.end(); it++)
    {
        cout << * it << " , ";
    }
    cout << endl;
}
void myprintA(allocator<int>::pointer vv, size_t num) //输出配置器
{
    int i;
    for(i = 0; i < num; i++)
    {
        cout << vv[i] << " , ";
    }
    cout << endl;
}
void main(int argc, char * argv[])
{
    cout << "allocator test. " << endl;
    vector<int> v1;
    vector<int>::iterator v1It;
    vector<int>::allocator_type v1A;
    allocator<int>::const_pointer cv1P;
    allocator<int>::pointer v1P;
    allocator<int> AL;

    int i;
    for(i = 1; i <= 8; i++)
    {
        v1.push_back(i); //初始化 vector
    }
    cout << "The Original Vector v1: " << endl;
```

```
myprint(v1);
int kk=6;
cv1P=v1A.address(*find(v1.begin(),v1.end(),kk)); //使用 address
cout<<"find an element by address."<<endl;
cout<<*cv1P<<endl;
v1p=AL.allocate(10); //分配内存
int ar[6]={0,9,8,7,6,5};
for(i=0;i<6;i++)
    v1p[i]=ar[i];
cout<<"array v1p:"<<endl;
    myprintA(v1p,6);
AL.deallocate(v1p,10); //释放内存
kk=6;
int kb=12;
v1p=v1A.address(*find(v1.begin(),v1.end(),kk));
v1A.destroy(v1p);
v1A.construct(v1p,kb); //修改数据
cout<<"destroy and construct:"<<endl;
myprint(v1);
allocator<int>::size_type size;
size=v1A.max_size(); //max_size
cout<<"max_size:"<<size<<endl;
}
```

例 14-1 的执行效果如图 14-1 所示。



```
allocator test.
The Original Vector v1:
1, 2, 3, 4, 5, 6, 7, 8,
Find an element by address.
6
array v1p:
0, 9, 8, 7, 6, 5,
destroy and construct:
1, 2, 3, 4, 5, 12, 7, 8,
max_size: 1073741823
请按任意键继续...
```

图 14-1 例 14-1 的执行效果

第 15 章

原子运行库模板

C++ 语言引进原子类操作主要目的是引进了多线程机制，并坚持实行顺序一致性。尽管效率不高，但也有其优点。原子类和原子类模板保证了某操作的原子性（即不可分割性和不可拆分性）：某操作一旦被执行，将一直执行下去，直至结束。

15.1 头文件 <atomic> 简介

本节介绍头文件 <atomic>。此头文件主要包括无锁属性、常规操作、模板函数、算术运算、关于指针的部分特殊操作和标志操作等内容。

15.1.1 无锁属性

头文件 <atomic> 包含了 11 个宏，但是没有明确定义其值。这几个宏主要用于原子操作的无锁属性。

```
#define ATOMIC_BOOL_LOCK_FREE unspecified
#define ATOMIC_CHAR_LOCK_FREE unspecified
#define ATOMIC_CHAR16_T_LOCK_FREE unspecified
#define ATOMIC_CHAR32_T_LOCK_FREE unspecified
#define ATOMIC_WCHAR_T_LOCK_FREE unspecified
#define ATOMIC_SHORT_LOCK_FREE unspecified
#define ATOMIC_INT_LOCK_FREE unspecified
#define ATOMIC_LONG_LOCK_FREE unspecified
#define ATOMIC_LLONG_LOCK_FREE unspecified
#define ATOMIC_LLONG_LOCK_FREE unspecified
#define ATOMIC_POINTER_LOCK_FREE unspecified
```

由于上述宏没有明确定义其值，因此本书暂不详细讲述。

15.1.2 3 个模板

- `template < class T > struct atomic;`
- `template < > struct atomic < integral >;`
- `template < class T > struct atomic < T* >;`

第一个模板声明了模板类 `atomic`。其意义为：一个模板类型为 `T` 的原子对象中封装了一个类型为 `T` 的值。原子类型对象的主要特点就是从不同线程访问不会导致数据竞争（Data Race），因此从不同线程访问某个原子对象是“良性”（Well-defined）行为；而通常对于非

原子类型对象而言，并发访问某个对象（如果不做任何同步操作）会导致未定义（Undefined）行为发生。

第二个模板是关于整形数据类型的特化实现。其定义式中的符号 `integral` 可以代表以下数据类型：`char`、`signed char`、`unsigned char`、`short`、`unsigned short`、`int`、`unsigned int`、`long`、`unsigned long`、`long long`、`unsigned long long`、`char16_t`、`char32_t` 和 `wchar_t`。

第三个模板是针对指针类型的特化实现。

15.1.3 原子模板的常规操作

头文件 `<atomic>` 提供了 10 个常规操作函数。其原型及意义分别如下：

```
bool atomic_is_lock_free(const volatile atomic-type * ) noexcept;
bool atomic_is_lock_free(const atomic-type * ) noexcept;
```

该函数用于判断 `std::atomic` 对象是否具备 lock-free 特性。如果某个对象满足 lock-free 特性，那么在多个线程访问该对象时不会导致线程阻塞。

```
void atomic_init(volatile atomic-type * ,T)noexcept;
void atomic_init(atomic-type * ,T) noexcept;
```

该函数用于初始化原子对象。`val` 指定原子对象的初始值。若对一个已初始化的原子对象再次调用 `atomic_init()`，则会导致未定义行为。若想修改原子对象的值，则应使用 `std::atomic_store()`。

```
void atomic_store(volatile atomic-type * ,T)noexcept;
void atomic_store(atomic-type * ,T)noexcept;
```

该函数用于修改原子对象的值。该函数相当于 `std::atomic` 对象的 `store` 或者 `operator =` () 成员函数，若需要显式指定内存序，则应使用 `atomic_store_explicit`。

```
void atomic_store_explicit(volatile atomic-type * ,T,memory_order)noexcept;
void atomic_store_explicit(atomic-type * ,T,memory_order)noexcept;
```

该函数用于修改原子对象的值。该函数相当于 `std::atomic` 对象的 `store` 或者 `operator =` () 成员函数，`memory_order` 指定了内存序，其可取的参数为：

Memory Order 值	Memory Order 类型
<code>memory_order_relaxed</code>	Relaxed
<code>memory_order_release</code>	Release
<code>memory_order_seq_cst</code>	Sequentially consistent

```
T atomic_load(const volatile atomic-type * ) noexcept;
T atomic_load(const atomic-type * ) noexcept;
```

该函数用于读取被封装的值，默认的内存序为 `memory_order_seq_cst`。该函数与 `std::atomic` 对象的 `atomic::load()` 成员函数等效。

```
T atomic_load_explicit(const volatile atomic-type * , memory_order) noexcept;
T atomic_load_explicit(const atomic-type * , memory_order) noexcept;
```

该函数用于读取被封装的值。参数 `sync` 设置了内存序，可能的取值如下：

Memory Order 值	Memory Order 类型
memory_order_relaxed	Relaxed
memory_order_consume	Consume
memory_order_acquire	Acquire
memory_order_seq_cst	Sequentially consistent

```
T atomic_exchange(volatile atomic-type *, T) noexcept;
T atomic_exchange(atomic-type *, T) noexcept;
```

该函数用于读取并修改被封装的值，exchange 会用 val 指定的值替换掉之前该原子对象封装的值，并返回之前该原子对象封装的值，整个过程具有原子性（因此 exchange 操作也被称为 read-modify-write 操作）。

```
T atomic_exchange_explicit(volatile atomic-type *, T, memory_order) noexcept;
T atomic_exchange_explicit(atomic-type *, T, memory_order) noexcept;
```

该函数用于读取并修改被封装的值，exchange 会用 val 指定的值替换掉之前该原子对象封装的值，并返回之前该原子对象封装的值，整个过程具有原子性（因此 exchange 操作也被称为 read-modify-write 操作）。

```
bool atomic_compare_exchange_weak(volatile atomic-type *, T *, T) noexcept;
bool atomic_compare_exchange_weak(atomic-type *, T *, T) noexcept;
```

该函数用于比较并交换被封装的值与参数第二个参数所指定的值是否相等，若相等，则用 val 替换原子对象的旧值；若不相等，则用原子对象的旧值替换第二个参数，因此调用该函数之后，如果被该原子对象封装的值与第二个参数所指定的值不相等，第二个参数中的内容就是原子对象的旧值。

该函数通常用于读取原子对象封装的值，若比较结果为 true（即原子对象的值等于第二个参数），则替换原子对象的旧值，但整个操作是原子性的，即在某个线程读取和修改该原子对象时，其他线程不能读取和修改该原子对象。



注意 该函数直接比较原子对象所封装的值与第二个参数的内容。某些情况下，对象的比较操作在使用 operator == () 判断时相等，atomic_compare_exchange_weak 判断时却可能失败，因为对象底层的物理内容中可能存在“位对齐”或其他逻辑表示“相同”但表示不同的值（比如 true 和 2 或 3，它们在逻辑上都表示“真”，但其表示并不相同）。

```
bool atomic_compare_exchange_strong(volatile atomic-type *, T *, T) noexcept;
bool atomic_compare_exchange_strong(atomic-type *, T *, T) noexcept;
```

该函数用于比较并交换被封装的值（strong）与第二个参数所指定的值是否相等，若相等，则用 val 替换原子对象的旧值；若不相等，则用原子对象的旧值替换第二个参数，因此调用该函数之后，如果被该原子对象封装的值与第二个参数所指定的值不相等，第二个参数中的内容就是原子对象的旧值。

该函数通常会读取原子对象封装的值，若比较为结果 true（即原子对象的值等于第二个

参数值), 则替换原子对象的旧值, 但整个操作是原子性的。在某个线程读取和修改该原子对象时, 另外的线程不能读取和修改该原子对象。

注意: 该函数直接比较原子对象所封装的值与第二个参数的物理内容, 所以在某些情况下, 对象的比较操作在使用 `operator ==()` 判断时相等, 但 `compare_exchange_weak` 判断时却可能失败, 因为对象底层的物理内容中可能存在位对齐或其他逻辑表示相同但是物理表示不同的值 (比如 `true` 和 `2` 或 `3`, 它们在逻辑上都表示“真”, 但在物理上两者的表示并不相同)。

与 `atomic_compare_exchange_weak` 不同, `strong` 版本的 `compare-and-exchange` 操作不允许返回 `false`, 即原子对象所封装的值与参数 `expected` 的物理内容相同, 比较操作一定会为 `true`。不过在某些平台下, 若算法本身需要循环操作来做检查, 则使用 `atomic_compare_exchange_weak` 会更好。对于某些不需要采用循环操作的算法而言, 通常采用 `atomic_compare_exchange_strong` 更好。

```
bool atomic_compare_exchange_weak_explicit(volatile atomic-type * ,T * ,T,memory_order, memory_order) noexcept;
bool atomic_compare_exchange_weak_explicit(atomic-type * ,T * ,T, memory_order, memory_order) noexcept;
```

该函数和 `atomic_compare_exchange_weak()` 的不同之处在于指定了两个 `memory_order` 类型参数。其功能还是比较并替换其包含的值。第二个 `memory_order` 类型参数不能是 `memory_order_release`, 也不能是 `memory_order_acq_rel`, 并且其值要小于第一个 `memory_order` 类型参数的值。

```
bool atomic_compare_exchange_strong_explicit (volatile atomic-type * ,T * ,T, memory_order, memory_order) noexcept;
bool atomic_compare_exchange_strong_explicit (atomic-type * ,T * ,T, memory_order, memory_order) noexcept;
```

该函数和 `atomic_compare_exchange_weak_explicit()` 类似。其功能与 `atomic_compare_exchange_strong` 相同。二者的不同之处在于增加了两个 `memory_order` 类型的参数。其两个新增加参数的意义和使用方法与 `atomic_compare_exchange_weak_explicit()` 函数相同。

15.1.4 头文件中的模板函数及算术运算函数

```
template <class T> T atomic_fetch_add(volatile atomic<T> * , T)noexcept;
template <class T> T atomic_fetch_add(atomic<T> * , T)noexcept;
```

该函数用于将原子的封装值增加 `T` 类型 (第二个参数) 参数的具体数值。默认内存序是 `memory_order_seq_cst`。该函数等价于 `std::atomic` 对象的 `atomic::fetch_add` 和 `atomic::operator +=` 成员函数。

```
template <class T> T atomic_fetch_add_explicit(volatile atomic<T> * , T,memory_order)
noexcept;
template <class T> T atomic_fetch_add_explicit(atomic<T> * , T,memory_order)noexcept;
```

该函数的功能和 `atomic_fetch_add()` 相似，仅仅是增加了可以设定内存序的参数。通常的内存序参数如下：

Memory Order 值	Memory Order 类型
<code>memory_order_relaxed</code>	Relaxed
<code>memory_order_consume</code>	Consume
<code>memory_order_acquire</code>	Acquire
<code>memory_order_release</code>	Release
<code>memory_order_acq_rel</code>	Acquire/Release
<code>memory_order_seq_cst</code>	Sequentially consistent

关于上述几种内存序的英文原版说明如下：

——`memory_order_relaxed`: no operation orders memory.

——`memory_order_release`, `memory_order_acq_rel`, and `memory_order_seq_cst`: a store operation performs a release operation on the affected memory location.

——`memory_order_consume`: a load operation performs a consume operation on the affected memory location.

——`memory_order_acquire`, `memory_order_acq_rel`, and `memory_order_seq_cst`: a load operation performs an acquire operation on the affected memory location.

本书将其翻译为：

——`memory_order_relaxed`: 没有内存操作顺序，可以任意排序。

——`memory_order_release`、`memory_order_acq_rel` 和 `memory_order_seq_cst`: 在指定内存位置执行 release 版本的存储（写）操作、即最终的存储（写）操作。

——`memory_order_consume`: 读取操作在指定内存区域是耗费内存的。

——`memory_order_acquire`、`memory_order_acq_rel` 和 `memory_order_seq_cst`: 读取操作时，在指定内存区是获取内存操作。

下面的函数和上述两个函数的用法相同，此处仅列出其名称，不再详细讲述。

```
template <class T> T atomic_fetch_sub(volatile atomic<T> *, T)noexcept;
template <class T> T atomic_fetch_sub(atomic<T> *, T)noexcept;
template <class T> T atomic_fetch_sub_explicit(volatile atomic<T> *, T, memory_order)noexcept;
template <class T> T atomic_fetch_sub_explicit(atomic<T> *, T, memory_order)noexcept;
```

```
template <class T> T atomic_fetch_and(volatile atomic<T> *, T)noexcept;
template <class T> T atomic_fetch_and(atomic<T> *, T)noexcept;
template <class T> T atomic_fetch_and_explicit(volatile atomic<T> *, T, memory_order)noexcept;
template <class T> T atomic_fetch_and_explicit(atomic<T> *, T, memory_order)noexcept;
```

```
template <class T> T atomic_fetch_or(volatile atomic<T> * , T) noexcept;
template <class T> T atomic_fetch_or(atomic<T> * , T) noexcept;
template <class T> T atomic_fetch_or_explicit(volatile atomic<T> * , T, memory_order) noexcept;
template <class T> T atomic_fetch_or_explicit(atomic<T> * , T, memory_order) noexcept;
```

```
template <class T> T atomic_fetch_xor(volatile atomic<T> * , T) noexcept;
template <class T> T atomic_fetch_xor(atomic<T> * , T) noexcept;
template <class T> T atomic_fetch_xor_explicit(volatile atomic<T> * , T, memory_order) noexcept;
template <class T> T atomic_fetch_xor_explicit(atomic<T> * , T, memory_order) noexcept;
```

上述函数的整型实例化形式为：

```
integral atomic_fetch_add(volatile atomic - integral * , integral) noexcept;
integral atomic_fetch_add(atomic - integral * , integral) noexcept;
integral atomic_fetch_add_explicit(volatile atomic - integral * , integral, memory_order) noexcept;
integral atomic_fetch_add_explicit(atomic - integral * , integral, memory_order) noexcept;
integral atomic_fetch_sub(volatile atomic - integral * , integral) noexcept;
integral atomic_fetch_sub(atomic - integral * , integral) noexcept;
integral atomic_fetch_sub_explicit(volatile atomic - integral * , integral, memory_order) noexcept;
integral atomic_fetch_sub_explicit(atomic - integral * , integral, memory_order) noexcept;
integral atomic_fetch_and(volatile atomic - integral * , integral) noexcept;
integral atomic_fetch_and(atomic - integral * , integral) noexcept;
integral atomic_fetch_and_explicit(volatile atomic - integral * , integral, memory_order) noexcept;
integral atomic_fetch_and_explicit(atomic - integral * , integral, memory_order) noexcept;
integral atomic_fetch_or(volatile atomic - integral * , integral) noexcept;
integral atomic_fetch_or(atomic - integral * , integral) noexcept;
integral atomic_fetch_or_explicit(volatile atomic - integral * , integral, memory_order) noexcept;
integral atomic_fetch_or_explicit(atomic - integral * , integral, memory_order) noexcept;
integral atomic_fetch_xor(volatile atomic - integral * , integral) noexcept;
integral atomic_fetch_xor(atomic - integral * , integral) noexcept;
integral atomic_fetch_xor_explicit(volatile atomic - integral * , integral, memory_order) noexcept;
integral atomic_fetch_xor_explicit(atomic - integral * , integral, memory_order) noexcept;
```

上述模板函数的指针形式为：

```
template <class T> T * atomic_fetch_add(volatile atomic<T * > * , ptrdiff_t) noexcept;
```

```

template <class T>T* atomic_fetch_add(atomic<T* > * , ptrdiff_t) noexcept;
template <class T>T* atomic_fetch_add_explicit(volatile atomic<T* > * , ptrdiff_t, memory_order) noexcept;
template <class T>T* atomic_fetch_add_explicit(atomic<T* > * , ptrdiff_t, memory_order) noexcept;
template <class T>T* atomic_fetch_sub(volatile atomic<T* > * , ptrdiff_t) noexcept;
template <class T>T* atomic_fetch_sub(atomic<T* > * , ptrdiff_t) noexcept;
template <class T>T* atomic_fetch_sub_explicit(volatile atomic<T* > * , ptrdiff_t, memory_order) noexcept;
template <class T>T* atomic_fetch_sub_explicit(atomic<T* > * , ptrdiff_t, memory_order) noexcept;

```

例 15-1

```

#include <iostream>
#include <atomic>
using namespace std;
int main(int argc, char * argv[])
{
    cout << "Atomic operation Example : " << endl;
    atomic<int> aa(0);
    cout << "原子类型对象 aa 数值 (构造赋值): " << aa.load() << endl;
    atomic_init(&aa,50);
    cout << "原子类型对象 aa 数值 (init): " << aa.load() << endl;
    aa.store(100);
    cout << "原子类型对象 aa 数值 (store): " << aa.load() << endl;
    atomic_fetch_add(&aa,150);
    cout << "原子类型对象 aa 数值 (+150): " << aa.load() << endl;
    bool y=atomic_is_lock_free(&aa);
    cout << "aa is lock free: " << y << endl;
    cin.get();
    atomic<int> bb(1),cc;
    cout << "原子类型对象 bb 数值 (构造赋值): " << bb.load() << endl;
    cc.store(aa.load());
    atomic_exchange(&aa,bb.load());
    atomic_exchange(&bb,cc.load());
    cout << "原子类型对象 aa ,bb 数值 (交换之后): " << aa.load() << ", " << bb.load() << endl;
    cin.get();
    int dd=90;
    bool res=0;
    res=atomic_compare_exchange_weak(&aa, &dd,5);
    cout << "The first : compare_exchange_weak(aa,dd): " << res << endl << endl;
    cout << "原子类型对象 aa : " << aa.load() << endl;
    cout << "原子类型对象 dd : " << dd << endl;
    res=atomic_compare_exchange_weak(&aa, &dd,5);
    cout << "The second : compare_exchange_weak(aa,dd): " << res << endl << endl;

```

```

cout << "原子类型对象 aa : " << aa.load() << endl;
cout << "原子类型对象 dd : " << dd << endl;
cin.get();
return 0;
}

```

例 15-1 的执行效果如图 15-1 所示。



图 15-1 例 15-1 的执行效果

15.1.5 原子类型 atomic_flag

```
struct atomic_flag;
```

atomic_flag 是原子运行库最基本的类型之一。atomic_flag 是一种简单的原子布尔类型。该类型仅支持两种类型的操作：test_and_set 和 clear。atomic_flag 的构造函数不允许被复制，因此不能使用一个 atomic_flag 对象构造另一个对象。

宏 ATOMIC_FLAG_INIT：如果某个 std::atomic_flag 对象使用该宏初始化，那么可以保证该 std::atomic_flag 对象在创建时处于 clear 状态。

```

bool atomic_flag_test_and_set(volatile atomic_flag * ) noexcept;
bool atomic_flag_test_and_set(atomic_flag * ) noexcept;
bool atomic_flag_test_and_set_explicit(volatile atomic_flag * , memory_order) noexcept;
bool atomic_flag_test_and_set_explicit(atomic_flag * , memory_order) noexcept;

```

atomic_test_and_set() 函数用于检查 std::atomic_flag 标识，若 std::atomic_flag 之前没有被设置过，则设置 std::atomic_flag 的标识，并返回先前该 std::atomic_flag 对象是否被设置过。若之前 std::atomic_flag 对象已被设置，则返回 true；否则，返回 false。

atomic_test_and_set() 的操作具有原子性，并且也可以指定“内存序”参数。

```

void atomic_flag_clear(volatile atomic_flag * ) noexcept;
void atomic_flag_clear(atomic_flag * ) noexcept;
void atomic_flag_clear_explicit(volatile atomic_flag * , memory_order) noexcept;
void atomic_flag_clear_explicit(atomic_flag * , memory_order) noexcept;

```

上述函数用于清除 std::atomic_flag 对象的标识位，即设置 atomic_flag 的值为 false。清除 std::atomic_flag 标识使得下一次调用 std::atomic_flag::test_and_set，可返回 false。该函数

也可以指定内存序参数。

例 15-2

```
#include <iostream> // std::cout
#include <atomic> // std::atomic_flag
#include <thread> // std:: thread
#include <vector> // std:: vector
#include <sstream> // std:: stringstream
using namespace std;
int main ()
{
    atomic_flag lock = ATOMIC_FLAG_INIT;
    bool set = atomic_flag_test_and_set (&lock);
    cout << " lock whether is set before: " << set << endl;
    cout << " lock's value: " << lock._My_flag << endl;
    atomic_flag_clear (&lock);
    cout << " lock's value (cleared): " << lock._My_flag << endl;
    cin.get ();
    return 0;
}
```

例 15-2 的执行效果如图 15-2 所示。



图 15-2 例 15-2 的执行效果

15.2 顺序及一致性

枚举类型变量 `memory_order` 指明了详细的、标准化的内存同步顺序，也提供了运行的次序。其各个枚举值及其意义前面已经有所讲述。需要特别注意的是，即使是采用 `memory_order_relaxed` 的序形式，也要保证原子类型数据访问的不可拆分性。

该枚举类型为：

```
typedef enum memory_order {
    memory_order_relaxed, /* 无序 */
    memory_order_consume, /* 耗费 */
    memory_order_acquire, /* 获取 */
    memory_order_release, /* 释放 */
    memory_order_acq_rel, /* 既获取又释放 */
    memory_order_seq_cst /* 默认规则 */
} memory_order;
```

原子操作的 `release` 类型操作是和 `acquire` 类型操作相对应的。对于原子类型对象 `M`，当原子操作 `A` 是 `release` 类型时，其同步的原子操作 `B` 肯定是在执行 `acquire` 类型操作。

如果在多个线程中对原子类型或其相关类型的共享资源进行操作，编译器将保证这些操作都是原子性的，也就是说，确保任意时刻只有一个线程对这个资源进行访问，编译器将保证多个线程访问这个共享资源的正确性，从而避免了锁的使用，提高了效率。

使用内存序规则时，默认规则都是 `std::memory_order_seq_cst`。此外，`atomic` 还有一些标识类型和测试操作，比较类似操作系统里的原子操作 `std::atomic_flag`。其他规则用来放宽顺序一致性以从无锁算法中获得更好的性能。



提示 `volatile` 是一个类型修饰符。`volatile` 被设计用来修饰被不同线程访问和修改的变量。在编写多线程程序时，如果不使用 `volatile`，可能会导致无法编写多线程程序或者会导致编译器失去大量优化的机会。

15.3 原子类型

前面已经讲过，头文件 `<atomic>` 包含 3 个原子类型：

- `template <class T> struct atomic;`
- `template <> struct atomic <integral>;`
- `template <class T> struct atomic <T* >;`

这些模板类型分别提供了多种针对原子类型对象的操作，例如常规操作、模板函数操作、算术操作、指针类型操作等。

头文件还提供了诸多的原子类型操作接口，而原子类型也提供了相应功能的成员函数。这 3 个类型的声明形式分别为：

```
template <class T> struct atomic {
    bool is_lock_free() const volatile noexcept;
    bool is_lock_free() const noexcept;
    void store(T, memory_order = memory_order_seq_cst) volatile noexcept;
    void store(T, memory_order = memory_order_seq_cst) noexcept;
    T load(memory_order = memory_order_seq_cst) const volatile noexcept;
    T load(memory_order = memory_order_seq_cst) const noexcept;
    operator T() const volatile noexcept;
    operator T() const noexcept;
    T exchange(T, memory_order = memory_order_seq_cst) volatile noexcept;
    T exchange(T, memory_order = memory_order_seq_cst) noexcept;
    bool compare_exchange_weak(T &, T, memory_order, memory_order) volatile noexcept;
    bool compare_exchange_weak(T &, T, memory_order, memory_order) noexcept;
    bool compare_exchange_strong(T &, T, memory_order, memory_order) volatile noexcept;
    bool compare_exchange_strong(T &, T, memory_order, memory_order) noexcept;
    bool compare_exchange_weak(T &, T, memory_order = memory_order_seq_cst) volatile noexcept;
    bool compare_exchange_weak(T &, T, memory_order = memory_order_seq_cst) noexcept;
};
```

```

    bool compare_exchange_strong(T &, T, memory_order = memory_order_seq_cst) volatile noexcept;
};

bool compare_exchange_strong(T &, T, memory_order = memory_order_seq_cst) noexcept;
atomic() noexcept = default;
constexpr atomic(T) noexcept;
atomic(const atomic &) = delete;
atomic & operator=(const atomic &) = delete;
atomic & operator=(const atomic &) volatile = delete;
T operator=(T) volatile noexcept;
T operator=(T) noexcept;
};

template < > struct atomic<integral > {
    bool is_lock_free() const volatile noexcept;
    bool is_lock_free() const noexcept;
    void store(integral, memory_order = memory_order_seq_cst) volatile noexcept;
    void store(integral, memory_order = memory_order_seq_cst) noexcept;
    integral load(memory_order = memory_order_seq_cst) const volatile noexcept;
    integral load(memory_order = memory_order_seq_cst) const noexcept;
    operator integral() const volatile noexcept;
    operator integral() const noexcept;
    integral exchange(integral, memory_order = memory_order_seq_cst) volatile noexcept;
    integral exchange(integral, memory_order = memory_order_seq_cst) noexcept;
    bool compare_exchange_weak(integral &, integral, memory_order, memory_order) volatile
noexcept;
    bool compare_exchange_weak(integral &, integral, memory_order, memory_order) noexcept;
    bool compare_exchange_strong(integral &, integral, memory_order, memory_order) volatile
noexcept;
    bool compare_exchange_strong(integral &, integral, memory_order, memory_order) noexcept;
    bool compare_exchange_weak(integral &, integral, memory_order = memory_order_seq_cst)
volatile noexcept;
    bool compare_exchange_weak(integral &, integral, memory_order = memory_order_seq_cst)
noexcept;
    bool compare_exchange_strong(integral &, integral, memory_order = memory_order_seq_cst)
volatile noexcept;
    bool compare_exchange_strong(integral &, integral, memory_order = memory_order_seq_cst)
noexcept;
    integral fetch_add(integral, memory_order = memory_order_seq_cst) volatile noexcept;
    integral fetch_add(integral, memory_order = memory_order_seq_cst) noexcept;
    integral fetch_sub(integral, memory_order = memory_order_seq_cst) volatile noexcept;
    integral fetch_sub(integral, memory_order = memory_order_seq_cst) noexcept;
};

```

```
integral fetch_and(integral, memory_order = memory_order_seq_cst) volatile noexcept;
integral fetch_and(integral, memory_order = memory_order_seq_cst) noexcept;
integral fetch_or(integral, memory_order = memory_order_seq_cst) volatile noexcept;
integral fetch_or(integral, memory_order = memory_order_seq_cst) noexcept;
integral fetch_xor(integral, memory_order = memory_order_seq_cst) volatile noexcept;
integral fetch_xor(integral, memory_order = memory_order_seq_cst) noexcept;
atomic() noexcept = default;
constexpr atomic(integral) noexcept;
atomic(const atomic &) = delete;
atomic & operator = (const atomic &) = delete;
atomic & operator = (const atomic &) volatile = delete;
integral operator = (integral) volatile noexcept;
integral operator = (integral) noexcept;
integral operator ++ (int) volatile noexcept;
integral operator ++ (int) noexcept;
integral operator -- (int) volatile noexcept;
integral operator -- (int) noexcept;
integral operator ++ () volatile noexcept;
integral operator ++ () noexcept;
integral operator -- () volatile noexcept;
integral operator -- () noexcept;
integral operator += (integral) volatile noexcept;
integral operator += (integral) noexcept;
integral operator -= (integral) volatile noexcept;
integral operator -= (integral) noexcept;
integral operator &= (integral) volatile noexcept;
integral operator &= (integral) noexcept;
integral operator |= (integral) volatile noexcept;
integral operator |= (integral) noexcept;
integral operator ^= (integral) volatile noexcept;
integral operator ^= (integral) noexcept;
};
template <class T> struct atomic<T* > {
    bool is_lock_free() const volatile noexcept;
    bool is_lock_free() const noexcept;
    void store(T* , memory_order = memory_order_seq_cst) volatile noexcept;
    void store(T* , memory_order = memory_order_seq_cst) noexcept;
    T* load(memory_order = memory_order_seq_cst) const volatile noexcept;
    T* load(memory_order = memory_order_seq_cst) const noexcept;
    operator T* () const volatile noexcept;
    operator T* () const noexcept;
    T* exchange(T* , memory_order = memory_order_seq_cst) volatile noexcept;
    T* exchange(T* , memory_order = memory_order_seq_cst) noexcept;
};
```

```

bool compare_exchange_weak(T* &, T*, memory_order, memory_order) volatile noexcept;
bool compare_exchange_weak(T* &, T*, memory_order, memory_order) noexcept;
bool compare_exchange_strong(T* &, T*, memory_order, memory_order) volatile noexcept;
bool compare_exchange_strong(T* &, T*, memory_order, memory_order) noexcept;
bool compare_exchange_weak(T* &, T*, memory_order = memory_order_seq_cst) volatile
noexcept;
bool compare_exchange_weak(T* &, T*, memory_order = memory_order_seq_cst) noexcept;
bool compare_exchange_strong(T* &, T*, memory_order = memory_order_seq_cst) volatile
noexcept;
bool compare_exchange_strong(T* &, T*, memory_order = memory_order_seq_cst) noex-
cept;

T* fetch_add(ptrdiff_t, memory_order = memory_order_seq_cst) volatile noexcept;
T* fetch_add(ptrdiff_t, memory_order = memory_order_seq_cst) noexcept;
T* fetch_sub(ptrdiff_t, memory_order = memory_order_seq_cst) volatile noexcept;
T* fetch_sub(ptrdiff_t, memory_order = memory_order_seq_cst) noexcept;
atomic() noexcept = default;
constexpr atomic(T*) noexcept;
atomic(const atomic&) = delete;
atomic& operator=(const atomic&) = delete;
atomic& operator=(const atomic&) volatile = delete;
T* operator=(T*) volatile noexcept;
T* operator=(T*) noexcept;
T* operator++(int) volatile noexcept;
T* operator++(int) noexcept;
T* operator--(int) volatile noexcept;
T* operator--(int) noexcept;
T* operator++() volatile noexcept;
T* operator++() noexcept;
T* operator--() volatile noexcept;
T* operator--() noexcept;
T* operator+=(ptrdiff_t) volatile noexcept;
T* operator+=(ptrdiff_t) noexcept;
T* operator-=(ptrdiff_t) volatile noexcept;
T* operator-=(ptrdiff_t) noexcept;
};

```

上述3个类型基本上均定义了构造器、store()、load()、无锁属性判断、交换、加法、减法以及运算操作符和逻辑运算符等。

15.3.1 模板类 atomic

std::atomic 是模板类。

```
template <class T> struct atomic;
```

该模板类的成员函数包括构造函数、赋值符号、无锁属性判断、存储、读取、operator T ()、读取并修改、compare_exchange_weak (比较并交换) 和 compare_exchange_strong()。

其中构造函数有如下 3 种形式:

```
atomic() noexcept = default;           //默认值
constexpr atomic(T) noexcept;         //直接赋值
atomic(const atomic &) = delete;       //禁止使用
```

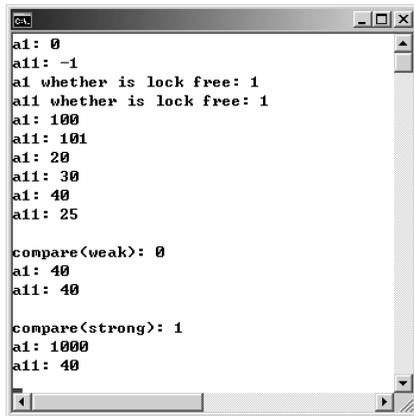
针对该模板类及其各成员函数的功能, 下面举例说明其用法。

例 15-3

```
#include <iostream>
#include <atomic>
using namespace std;
int main(int argc, char * argv[])
{
    atomic<int> a11(-1);
    atomic<int> a1;
    cout << "a1: " << a1.load(memory_order_relaxed) << endl;
    cout << "a11: " << a11.load(memory_order_relaxed) << endl;
    bool y, y1;
    y = a1.is_lock_free();
    y1 = a11.is_lock_free();
    cout << "a1 whether is lock free: " << y << endl;
    cout << "a11 whether is lock free: " << y1 << endl;
    a1.store(100, memory_order_relaxed);
    a11.store(101, memory_order_relaxed);
    cout << "a1: " << a1.load(memory_order_relaxed) << endl;
    cout << "a11: " << a11.load(memory_order_relaxed) << endl;
    a1.exchange(20);
    a11.exchange(30);
    cout << "a1: " << a1.load(memory_order_relaxed) << endl;
    cout << "a11: " << a11.load(memory_order_relaxed) << endl;
    a1.fetch_add(20);
    a11.fetch_sub(5);
    cout << "a1: " << a1.load(memory_order_relaxed) << endl;
    cout << "a11: " << a11.load(memory_order_relaxed) << endl;
    cin.get();
    y1 = a1.compare_exchange_weak((int&)a11, 1000);
    cout << "compare(weak): " << y1 << endl;
    cout << "a1: " << a1.load(memory_order_relaxed) << endl;
    cout << "a11: " << a11.load(memory_order_relaxed) << endl;
    cin.get();
    y1 = a1.compare_exchange_strong((int &)a11, 1000);
    cout << "compare(strong): " << y1 << endl;
    cout << "a1: " << a1.load(memory_order_relaxed) << endl;
    cout << "a11: " << a11.load(memory_order_relaxed) << endl;
```

```
cin.get();  
return 0;  
}
```

例 15-3 的执行效果如图 15-3 所示。



```
GM  
a1: 0  
a11: -1  
a1 whether is lock free: 1  
a11 whether is lock free: 1  
a1: 100  
a11: 101  
a1: 20  
a11: 30  
a1: 40  
a11: 25  
  
compare<weak>: 0  
a1: 40  
a11: 40  
  
compare<strong>: 1  
a1: 1000  
a11: 40
```

图 15-3 例 15-3 的执行效果

15.3.2 针对整型数据的特殊化模板

本小节将讲述原子类型针对整型数据的特殊化模板。

该模板的声明形式如下：

```
template < > struct atomic<integral > {  
    bool is_lock_free() const volatile noexcept;  
    bool is_lock_free() const noexcept;  
    void store(integral, memory_order = memory_order_seq_cst) volatile noexcept;  
    void store(integral, memory_order = memory_order_seq_cst) noexcept;  
    integral load(memory_order = memory_order_seq_cst) const volatile noexcept;  
    integral load(memory_order = memory_order_seq_cst) const noexcept;  
    operator integral() const volatile noexcept;  
    operator integral() const noexcept;  
    integral exchange(integral, memory_order = memory_order_seq_cst) volatile noexcept;  
    integral exchange(integral, memory_order = memory_order_seq_cst) noexcept;  
    bool compare_exchange_weak(integral &, integral, memory_order, memory_order) volatile noexcept;  
    bool compare_exchange_weak(integral &, integral, memory_order, memory_order) noexcept;  
    bool compare_exchange_strong(integral &, integral, memory_order, memory_order) volatile noexcept;  
    bool compare_exchange_strong(integral &, integral, memory_order, memory_order) noexcept;  
    bool compare_exchange_weak(integral &, integral, memory_order = memory_order_seq_cst) volatile noexcept;  
};
```

```
bool compare_exchange_weak(integral &, integral, memory_order = memory_order_seq_cst) noexcept;
bool compare_exchange_strong(integral &, integral, memory_order = memory_order_seq_cst) volatile noexcept;
bool compare_exchange_strong(integral &, integral, memory_order = memory_order_seq_cst) noexcept;

integral fetch_add(integral, memory_order = memory_order_seq_cst) volatile noexcept;
integral fetch_add(integral, memory_order = memory_order_seq_cst) noexcept;
integral fetch_sub(integral, memory_order = memory_order_seq_cst) volatile noexcept;
integral fetch_sub(integral, memory_order = memory_order_seq_cst) noexcept;
integral fetch_and(integral, memory_order = memory_order_seq_cst) volatile noexcept;
integral fetch_and(integral, memory_order = memory_order_seq_cst) noexcept;
integral fetch_or(integral, memory_order = memory_order_seq_cst) volatile noexcept;
integral fetch_or(integral, memory_order = memory_order_seq_cst) noexcept;
integral fetch_xor(integral, memory_order = memory_order_seq_cst) volatile noexcept;
integral fetch_xor(integral, memory_order = memory_order_seq_cst) noexcept;
atomic() noexcept = default;
constexpr atomic(integral) noexcept;
atomic(const atomic &) = delete;
atomic & operator = (const atomic &) = delete;
atomic & operator = (const atomic &) volatile = delete;
integral operator = (integral) volatile noexcept;
integral operator = (integral) noexcept;
integral operator ++ (int) volatile noexcept;
integral operator ++ (int) noexcept;
integral operator -- (int) volatile noexcept;
integral operator -- (int) noexcept;
integral operator ++ () volatile noexcept;
integral operator ++ () noexcept;
integral operator -- () volatile noexcept;
integral operator -- () noexcept;
integral operator += (integral) volatile noexcept;
integral operator += (integral) noexcept;
integral operator -= (integral) volatile noexcept;
integral operator -= (integral) noexcept;
integral operator &= (integral) volatile noexcept;
integral operator &= (integral) noexcept;
integral operator |= (integral) volatile noexcept;
integral operator |= (integral) noexcept;
integral operator ^= (integral) volatile noexcept;
integral operator ^= (integral) noexcept;
};
```

由以上代码可知，该模板包含了多个算术运算符函数、构造函数、加法函数、减法函数、或函数、异或函数、比较并修改函数、修改函数、读取函数、存储函数和无锁属性判断函数。

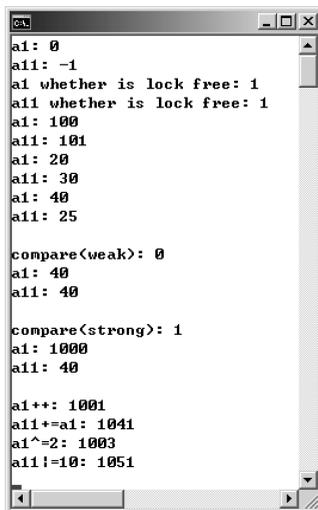
下面通过举例说明其用法。这种类型对象不能像传统形式那样复制数据，且增加了多个算术运算的函数，其他操作的使用几乎没有变化。

例 15-4

```
#include <iostream>
#include <atomic>
using namespace std;
int main(int argc, char* argv[])
{
    atomic<long> a1(-1);
    atomic<long> a1;
    cout << "a1: " << a1.load(memory_order_relaxed) << endl;
    cout << "a11: " << a11.load(memory_order_relaxed) << endl;
    bool y, y1;
    y = a1.is_lock_free();
    y1 = a11.is_lock_free();
    cout << "a1 whether is lock free: " << y << endl;
    cout << "a11 whether is lock free: " << y1 << endl;
    a1.store(100, memory_order_relaxed);
    a11.store(101, memory_order_relaxed);
    cout << "a1: " << a1.load(memory_order_relaxed) << endl;
    cout << "a11: " << a11.load(memory_order_relaxed) << endl;
    a1.exchange(20);
    a11.exchange(30);
    cout << "a1: " << a1.load(memory_order_relaxed) << endl;
    cout << "a11: " << a11.load(memory_order_relaxed) << endl;
    a1.fetch_add(20);
    a11.fetch_sub(5);
    cout << "a1: " << a1.load(memory_order_relaxed) << endl;
    cout << "a11: " << a11.load(memory_order_relaxed) << endl;
    cin.get();
    y1 = a1.compare_exchange_weak((long &)a11._My_val, 1000);
    cout << "compare (weak): " << y1 << endl;
    cout << "a1: " << a1.load(memory_order_relaxed) << endl;
    cout << "a11: " << a11.load(memory_order_relaxed) << endl;
    cin.get();
    y1 = a1.compare_exchange_strong((long &)a11._My_val, 1000);
    cout << "compare (strong): " << y1 << endl;
    cout << "a1: " << a1.load(memory_order_relaxed) << endl;
    cout << "a11: " << a11.load(memory_order_relaxed) << endl;
    cin.get();
    a1++;
    a11 += a1;
    cout << "a1 ++: " << a1.load(memory_order_relaxed) << endl;
    cout << "a11 += a1: " << a11.load(memory_order_relaxed) << endl;
    a1 ^= 2;
```

```
    a11 |= 10;
    cout << "a1^=2: " << a1.load(memory_order_relaxed) << endl;
    cout << "a11 |= 10: " << a11.load(memory_order_relaxed) << endl;
    cin.get();
    return 0;
}
```

例 15-4 的执行效果如图 15-4 所示。



```
a1: 0
a11: -1
a1 whether is lock free: 1
a11 whether is lock free: 1
a1: 100
a11: 101
a1: 20
a11: 30
a1: 40
a11: 25

compare<weak>: 0
a1: 40
a11: 40

compare<strong>: 1
a1: 1000
a11: 40

a1++: 1001
a11+=a1: 1041
a1^=2: 1003
a11|=10: 1051
```

图 15-4 例 15-4 的执行效果

15.3.3 针对指针的特殊化模板

本小节讲述原子类型针对指针的特殊化模板。

该模板的声明形式如下：

```
template <class T> struct atomic<T* > {
    bool is_lock_free() const volatile noexcept;
    bool is_lock_free() const noexcept;
    void store(T* , memory_order = memory_order_seq_cst) volatile noexcept;
    void store(T* , memory_order = memory_order_seq_cst) noexcept;
    T* load(memory_order = memory_order_seq_cst) const volatile noexcept;
    T* load(memory_order = memory_order_seq_cst) const noexcept;
    operator T* () const volatile noexcept;
    operator T* () const noexcept;
    T* exchange(T* , memory_order = memory_order_seq_cst) volatile noexcept;
    T* exchange(T* , memory_order = memory_order_seq_cst) noexcept;
    bool compare_exchange_weak(T* &, T* , memory_order, memory_order) volatile noexcept;
    bool compare_exchange_weak(T* &, T* , memory_order, memory_order) noexcept;
    bool compare_exchange_strong(T* &, T* , memory_order, memory_order) volatile noexcept;
```

```

    bool compare_exchange_strong(T* &, T*, memory_order, memory_order) noexcept;
    bool compare_exchange_weak(T* &, T*, memory_order = memory_order_seq_cst) volatile noexcept;
};

bool compare_exchange_weak(T* &, T*, memory_order = memory_order_seq_cst) noexcept;
bool compare_exchange_strong(T* &, T*, memory_order = memory_order_seq_cst) volatile noexcept;

bool compare_exchange_strong(T* &, T*, memory_order = memory_order_seq_cst) noexcept;
T* fetch_add(ptrdiff_t, memory_order = memory_order_seq_cst) volatile noexcept;
T* fetch_add(ptrdiff_t, memory_order = memory_order_seq_cst) noexcept;
T* fetch_sub(ptrdiff_t, memory_order = memory_order_seq_cst) volatile noexcept;
T* fetch_sub(ptrdiff_t, memory_order = memory_order_seq_cst) noexcept;

atomic() noexcept = default;
constexpr atomic(T*) noexcept;
atomic(const atomic&) = delete;
atomic& operator=(const atomic&) = delete;
atomic& operator=(const atomic&) volatile = delete;
T* operator=(T*) volatile noexcept;
T* operator=(T*) noexcept;
T* operator++(int) volatile noexcept;
T* operator++(int) noexcept;
T* operator--(int) volatile noexcept;
T* operator--(int) noexcept;
T* operator++() volatile noexcept;
T* operator++() noexcept;
T* operator--() volatile noexcept;
T* operator--() noexcept;
T* operator+=(ptrdiff_t) volatile noexcept;
T* operator+=(ptrdiff_t) noexcept;
T* operator-=(ptrdiff_t) volatile noexcept;
T* operator-=(ptrdiff_t) noexcept;
};

```

由以上代码可知，该模板同样包含了多个算术运算符函数、构造函数、加法函数、减法函数、或函数、异或函数、比较并修改函数、修改函数、读取函数、存储函数和无锁属性判断函数。只不过其内在成员替换成了指针类型 T^* 。

下面通过举例说明其用法。这种类型对象不能像传统形式那样复制数据。

例 15-5

```

#include <iostream>
#include <atomic>
using namespace std;
int main(int argc, char* argv[])
{
    int dim[10] = {1,2,3,4};

```

```
atomic<int*> a1;  
atomic<int*> a11(dim);  
cout << "a11: " << a11.load(memory_order_relaxed) << endl;  
a11.fetch_add(2); //向前移动指针  
cout << "* (a11+2): " << * a11.load(memory_order_relaxed) << endl;  
a1.store((int *)0xfffffe);  
cout << "a1: " << a1.load(memory_order_relaxed) << endl;  
a1.exchange((int *)0x897543);  
cout << "a1: " << a1.load(memory_order_relaxed) << endl;  
a1++;  
a11++;  
cout << "a1++ : " << a1.load(memory_order_relaxed) << endl;  
cout << "* (a11++): " << * a11.load(memory_order_relaxed) << endl;  
cin.get();  
return 0;  
}
```

例 15-5 的执行效果如图 15-5 所示。



```
cmd  
a11: 001FFA80  
*(a11+2): 3  
a1: 000FFFFE  
a1: 00897543  
a1++ : 00897547  
*(a11++): 4
```

图 15-5 例 15-5 的执行效果

15.4 小 结

本章主要是讲述了原子类型模板的一些基本属性和基本操作。原子类型的优点主要是用于多线程操作。但本章并没有涉及多线程的内容，仅对原子类型的各种操作进行了细致分析，并辅以实例说明其用法，旨在让读者对原子类型有个初步了解。多线程的内容在其他章节介绍。

第 16 章

线程控制类模板

本章主要讲述如何创建和管理线程、如何实行互斥、如何在线程之间交换条件和数值等内容。线程支持库所涉及的头文件包括 `<thread >`、`<mutex >`、`<condition_variable >` 和 `<future >`。在使用相应功能时，程序必须包含相应的头文件。

16.1 要求和性能

模板参数的名称用于表达类型性能。若参数是谓词，则应用于实际模板参数的运算符 `operator()` 应该返回一个数值。该数值应能转换为 `bool` 类型数值。

16.1.1 异常

本节中的部分函数用于描述可以抛出 `system_error` 类型的异常。这种异常是可以被抛出的，尤其是检测到任意的函数错误、调用操作系统错误、其他应用程序编程接口导致的错误，相应的异常会被抛出。分配内存错误也需要报警、报告。在实际应用中，尤其会发生以下情况：函数用于抛出 `system_error` 类型异常，并标明错误条件（例不允许该线程执行此操作）。假定函数执行期间，在执行 POSIX API 过程中，`EPERM` 错误被报告。由于 POSIX 指定了 `EPERM` 错误，若调用没有权利执行该操作，则在具体实施过程中，函数会抛出信息“the caller does not have the privilege to perform the operation”。

16.1.2 本地句柄

本节中的部分类拥有 `native_handle_type` 类型和 `native_handle` 类型的成员。这些成员及其语义的提出是预先定义实施的。这些成员在实施过程中，提供了实施细节。它们的名字被用于帮助轻便的编译时间检测。这些成员在实际使用过程中，其实并不是便捷型。

16.1.3 时序规定

本节中涉及的函数均采用了一个用于确定延时的参数。这些延时是指定的，或者用于持续时间，或者用于时间点类型。

在必要的实施过程中，当返回延时时，这标志着已经拥有一些延时。任何超越中断响应，函数返回和调度导致的延时，可描述为持续延时 D_i 。其实，这类延时应该为 0。更深入地讲，任何处理器和内存资源的冲突会导致“质量管理”延时，可称为持续延时 D_m 。持续时间延时可以发生变化，当然延时越短越好。对于带参数的成员函数，如果其名称是以“_for”结尾的，会产生一个相对的延时。在实际应用中，程序员应该使用稳态时钟测量这些函数的执行时间。如果给定持续时间变量 D_t ，延时的实时持续时间应该是“ $D_t + D_i + D_m$ ”。

如果成员函数的名称是以“_until”结尾的，并采用一个参数用于确定延时，那么这些函数会产生绝对的延时。在实际使用过程中，程序员应当使用确定时钟及利用固定时间点来确定函数的执行时间。对于既定的时钟时间点参数 C_t ，延时返回的时钟时刻点应当是“ $C_t + D_i + D_m$ ”，前提是在演示过程中，时钟始终没有被调整。如果在演示过程中，时钟被调整到时间 C_a ，该行为就变得复杂了，需要更细致地计算该时刻及延时。

事件的精度依赖于操作系统和硬件，这是众所周知的。最优的精度被称为固有精度。用于测量这些延时的时钟必须满足普通时钟的性能要求。

16.1.4 可锁定类型

线程是可以和其他可执行代理（或线程）并行的执行工作任务的。调用线程是可以根据实际情况决策的，可调用线程包含了调用本身。标准库模板 `unique_lock`、`lock_guard`、`lock`、`try_lock` 和条件变量均可以运行或者执行用户提供的可锁定对象。无论是基本可锁定性能、可锁定性能，还是定时可锁定性能，均是锁的功能和要求。“锁”的目的就在于获取和释放被操作对象的所有权。

1. 基本可锁定性能

长整型类型 `L` 满足基本可锁性能要求，例如下面表达式中的 `m` 代表长整型数据。

```
L m;  
m lock();           //获取线程的所有权  
...  
m unlock();        //释放所有权
```

2. 可锁定性能

如果 `L` 类型对象能满足基本可锁定性能要求，那么也可满足可锁定性能要求，例如，

```
m try_lock();      //尝试获取所有权
```

3. 定时可锁定性能

如果长整型变量满足可锁定性能要求，那么也可满足定时可锁定性能。例如，

```
m try_lock(rel_time); //rel_time 代表持续时间
```

上述语句的功能：在延时 `rel_time` 之后，获取线程锁的所有权。在执行过程中，在时间 `rel_time` 之内，只有在获取线程锁的所有权之后，函数才能返回。如果有异常发生，线程锁不能获取当前线程的操作权限。

```
m try_lock_until(abs_time); //在 abs_time 之后,尝试获取线程所有权
```

功能描述：在绝对延时（`abs_time`）之后，尝试获取线程锁的所有权。

16.2 线程类

C++ 最新标准中，标准模板库提供了线程类 `thread`。类模板 `thread` 主要用于创建和管理线程类。命名空间 `std` 中包含了若干线程相关的函数或者命名空间。

```
namespace std  
{
```

```

class thread; //线程类 thread
void swap(thread& x, thread& y) noexcept; //互换函数
namespace this_thread //命名空间 this_thread
{
    thread::id get_id() noexcept; //获取线程 id
    void yield() noexcept; //放弃 CPU 时间, 等待其他线程向前
    执行
    template <class Clock, class Duration> //函数 sleep_until()
    void sleep_until(const chrono::time_point<Clock, Duration>& abs_time);
    template <class Rep, class Period>
    void sleep_for(const chrono::duration<Rep, Period>& rel_time); //函数 sleep_for
}
}

```

16.2.1 线程类成员变量 id

名称空间 `std` 中包含了类 `thread::id`。该类仅包含一个构造 `id()` 函数。该类支持多数运算符的运算：`==`、`!`、`=`、`<`、`<=`、`>` 和 `>=`。模板类 `basic_ostream` 的运算符“`<<`”也支持对线程 `thread_id` 的参数类别。再者，模板类 `hash` 也包含对线程类 `thread_id` 的支持。

类 `thread_id` 的对象提供了独一无二的标识代码，所有线程对象的唯一标识值并不能代表在执行的线程本身。每一个在执行的线程均拥有一个相关联的 `thread_id` 类对象，这个对象不会等于其他线程相关联的 `thread_id` 类的对象。

类 `thread_id` 应该是一个可复制类。该库可能会重复使用已终止线程的 `thread_id` 类型对象。

类的构造函数会构造一个该类型的对象。构造器产生的对象不代表线程本身。

运算符 `operator == (thread_id x, thread_id y)` 返回 `true` 的条件：`x` 和 `y` 同样代表执行的线程，或者 `x` 和 `y` 都不代表执行的线程。

运算符 `operator != (thread_id x, thread_id y)` 的返回值是 `!(x == y)`。

运算符 `operator < (thread_id x, thread_id y)` 的返回值是个值。

运算符 `operator <= (thread_id x, thread_id y)` 返回值是 `!(y < x)`。

运算符 `operator > (thread_id x, thread_id y)` 返回值是 `(y < x)`。

运算符 `operator >= (thread_id x, thread_id y)` 返回值是 `!(x < y)`。

16.2.2 线程类成员函数

1. 构造器函数

线程构造器函数有两种形式：`thread()` 和 `thread(thread&&x)`。

如果新线程启动失败，程序会抛出异常：`system_error`。

当系统缺乏必需的资源而不能创建新线程，或者进程中限制了线程的数量时，系统会提示：资源无效，请重试。

第二种形式的构造函数用于构造一个线程，并将该线程设置为默认状态。

2. 线程析构器函数

析构函数为：`~thread()`。

析构函数的功能：如果线程是可加入进程的，那么调用该函数即会终止线程。否则，设置线程的状态至默认构造状态。

3. 成员函数 `operator = (thread&& x)`

该成员函数的功能：若线程可加入进程并运行，则调用函数 `terminate()`。若线程不能运行，则将函数参数代表线程的状态赋值给本线程，并将函数参数的状态恢复至默认构造状态。

4. 其他成员函数

1) `void swap (thread& x)`。该成员函数用于交换本线程和线程 `x` 的状态。

2) `bool joinable()`。该成员函数用于判断线程是否可执行。

3) `void join()`。该函数可执行的前提条件是 `joinable()` 返回 `true`。该成员函数用于将线程加入进程的执行，直至线程结束才继续执行程序。一旦发出异常，会抛出 `system_error` 类型的异常，通常会发生以下 3 种错误：

- `resource_deadlock_would_occur`：线程发生死锁。
- `no_such_process`：线程是无效线程。
- `invalid_argument`：线程无法执行。

4) `void detach()`。该函数可执行的前提条件是：`joinable()` 返回 `true`。该函数的功能是：线程启动后，若执行 `detach()`，则不等待线程返回，继续执行程序。如果被调用的线程没有发生阻塞，线程将继续执行。一旦函数 `detach()` 返回，指针 `this` 不再代表可能继续执行的线程。当之前 `this` 代表的线程终止执行时，程序将释放任何自身资源。同样，一旦发生异常，会抛出 `system_error` 类型的异常。函数执行时通常发生的错误如下：

- `no_such_process`：代表线程无效。
- `invalid_argument`：线程不可执行，不能加入进程。

5) `id get_id() const noexcept`。如果 `this` 不代表某线程，该函数返回默认构造的 `id` 类型对象；否则，该函数返回 `this -> ::get_id()`。

除了上述函数，线程类还包含一个静态成员：`unsigned hardware_concurrency()`。

正常情况下，该函数返回硬件线程数目，但有时返回的数值并不代表线程数目，无确定意义。

16.2.3 命名空间 `this_thread`

命名空间 `std` 中包含了一个命名空间 `this_thread`。其声明形式如下：

```
namespace std {
    namespace this_thread {
        thread::id get_id() noexcept;
        void yield() noexcept;
        template <class Clock, class Duration > void sleep_until (const chrono::time_point < Clock,
Duration > & abs_time);
        template <class Rep, class Period > void sleep_for (const chrono::duration < Rep, Period > &
rel_time);
```

```

    }
}

```

该名称空间也包含了诸多成员函数：`get_id()`、`yield()`、`sleep_until()`及`sleep_for()`。

1) 成员函数 `get_id()`。该函数用于返回当前执行线程的唯一标识。

2) 成员函数 `yield()`。该函数用于提供重新调度的机会，暂停本线程的运行，允许其他线程或进程执行。

3) 模板 `sleep_until()`函数。该函数用于在绝对时间 `abs_time` 之后，阻塞正在运行的线程。其声明形式为：

```

template <class Clock, class Duration>void sleep_until(const chrono::time_point<Clock, Duration>& abs_time);

```

4) 模板 `sleep_for()`函数。该函数用于阻塞当前运行的线程，时长为参数 `rel_time` 代表的时间。其声明形式为：

```

template <class Rep, class Period>void sleep_for(const chrono::duration<Rep, Period>& rel_time);

```

16.2.4 线程示例

例 16-1

```

#include <iostream>
#include <thread>
using namespace std;
int thread_id=0;
void thread_task_add()
{
    thread_id++;
    cout << "hello thread " << endl;
    cout << "thread_id(++)'s value :" << thread_id << endl;
}
void thread_task_sub()
{
    thread_id--;
    cout << "thread_id(--)'s value :" << thread_id << endl;
}
int main(int argc, char* argv[])
{
    thread t(thread_task_add);
    t.join();
    thread t1(thread_task_sub);
    t1.join();
    cin.get();
    return 0;
}

```

例 16-1 的执行效果如图 16-1 所示。

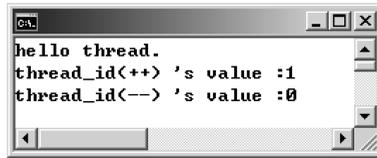


图 16-1 例 16-1 的执行效果

例 16-2

```
#include <iostream>  
#include <thread>  
using namespace std;  
  
void mysleep(std::chrono::microseconds us)  
{  
    auto s = std::chrono::high_resolution_clock::now();  
    auto e = s + us;  
    do {  
        std::this_thread::yield();  
    } while (std::chrono::high_resolution_clock::now() < e);  
}  
  
int main()  
{  
    auto start = std::chrono::high_resolution_clock::now();  
  
    mysleep(std::chrono::microseconds(10000));  
  
    auto elapsed = std::chrono::high_resolution_clock::now() - start;  
    std::cout << "waited for "  
        << std::chrono::duration_cast<std::chrono::microseconds>(elapsed).count()  
        << " microseconds\n";  
    cin.get();  
    return 0;  
}
```

例 16-2 的执行效果如图 16-2 所示。



图 16-2 例 16-2 的执行效果

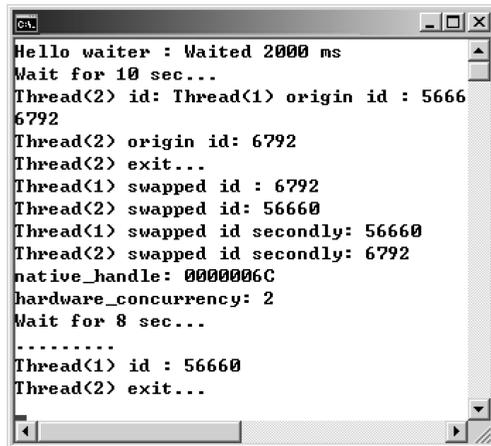
例 16-3

```
#include <iostream>
#include <thread>
#include <chrono>
#include <condition_variable>
#include <mutex>
#include <future>
using namespace std;
bool cont=0;
void thread_entry1 ()
{
    long a=10000;
    int sec=0;
    cout << "Wait for 10 sec.." << std::endl;
    while(a --)
    {
        chrono::milliseconds dura ( 1 );
        this_thread::sleep_for( dura );           //睡眠相对延时 dura 时间
    }
    cout << "Wait for 8 sec.." << endl;
    while(sec <= 8)
    {
        se C++;
        this_thread::sleep_until(chrono::system_clock::now() + chrono::seconds(sec));
        //睡眠至新的指定时间

        cout << ". ";
    }
    std::cout << std::endl;
    cout << "Thread(1) id : " << this_thread::get_id() << endl;
    cont=1;
}
void thread_entry2 ()
{
    cout << "Thread(2) id: " << this_thread::get_id() << endl;
    std::cout << "Thread(2) exit..." << std::endl;
}
int main(int argc, char* argv[])
{
    cout << "Hello waiter : ";
    chrono::milliseconds dura ( 2000 );
    this_thread::sleep_for( dura );
    cout << "Waited 2000 ms \n";
    thread t(thread_entry1);
    thread t2(thread_entry2);
    cout << "Thread(1) origin id : " << t.get_id() << endl;
```

```
cout << "Thread(2) origin id: " << t2.get_id() << endl;
t.swap(t2);
cout << "Thread(1) swapped id : " << t.get_id() << endl;
cout << "Thread(2) swapped id: " << t2.get_id() << endl;
t2.swap(t);
cout << "Thread(1) swapped id secondly: " << t.get_id() << endl;
cout << "Thread(2) swapped id secondly: " << t2.get_id() << endl;
if(t.joinable())
    t.detach();
if(t2.joinable())
    t2.join();
std::thread::native_handle_type hd=t.native_handle();
cout << "native_handle: " << hd << endl;
unsigned int tc=t.hardware_concurrency();
cout << "hardware_concurrency: " << tc << endl;
cin.get();
return 0;
}
```

例 16-3 的执行效果如图 16-3 所示。



```
GA
Hello waiter : Waited 2000 ms
Wait for 10 sec...
Thread(2) id: Thread(1) origin id : 5666
6792
Thread(2) origin id: 6792
Thread(2) exit...
Thread(1) swapped id : 6792
Thread(2) swapped id: 56660
Thread(1) swapped id secondly: 56660
Thread(2) swapped id secondly: 6792
native_handle: 0000006C
hardware_concurrency: 2
Wait for 8 sec...
.....
Thread(1) id : 56660
Thread(2) exit...
```

图 16-3 例 16-3 的执行效果

分析：由于程序中使用了两个线程，但是没有使用互斥等机制，导致两个线程竞争 CPU 资源。两个线程刚开始启动时程序的执行顺序较乱，导致程序的输出也是混乱的。

16.3 互 斥

本节讲述互斥机制，主要涉及 `mutexes`、`locks` 和 `call once`。其中互斥类型的类主要包括 `mutex`、`recursive_mutex`（递归型）、`timed_mutex`（超时型）以及 `recursive_timed_mutex`（递归并且超时）。互斥锁类型的类主要包括 `lock_guard` 和 `unique_lock()`。`call once()` 是一个模

板函数。相关类的声明形式如下：

```
namespace std {
    class mutex;
    class recursive_mutex;
    class timed_mutex;
    class recursive_timed_mutex;
    template <class Mutex> class lock_guard;
    template <class Mutex> class unique_lock;
    template <class Mutex>
    void swap(unique_lock<Mutex>& x, unique_lock<Mutex>& y) noexcept;
    template <class L1, class L2, class... L3> int try_lock(L1&, L2&, L3&...);
    template <class L1, class L2, class... L3> void lock(L1&, L2&, L3&...);
    struct once_flag {
        constexpr once_flag() noexcept;
        once_flag(const once_flag&) = delete;
        once_flag& operator = (const once_flag&) = delete;
        ...
    };
    template< class Callable, class ...Args > void call_once(once_flag& flag, Callable func,
Args&&... args);
}
```

16.3.1 mutex 模板类

一个 mutex 类型对象主要用于多线程之间保护数据以及确保数据的同步性。一个可执行程序或者线程从第一次调用 lock() 开始，直至调用 unlock() 为止，期间确保该线程归该线程所有。互斥对象可以是递归类型，也可以是非递归类型；可以保证瞬间获取一个或多个线程的对 CPU 的占有权。互斥类型对象提供唯一的所有权：在同一时刻，仅有一个线程可以拥有互斥对象。无论是递归型还是非递归型，互斥对象均具有此性质。

互斥类型主要是标准库类型 std::mutex、std::recursive_mutex、std::timed_mutex 和 std::recursive_timed_mutex。互斥类型对象必须满足可锁定性能要求。互斥类型包含默认的构造器和析构器。如果互斥类型对象初始化失败，系统会抛出异常 (system_error)。互斥类型对象是不能被复制和不能被移动的。

通常，错误条件的错误代码包含以下几种：

- resource_unavailable_try_again。该错误代码代表无效资源，请重新尝试。
- operation_not_permitted。该错误代码代表无操作权限。
- device_or_resource_busy。该错误代码代表其他线程正在占用 CPU 资源。
- invalid_argument。该错误代码代表该参数无效。

在多线程执行实施过程中，互斥类型对象要提供锁定和解锁两种操作。为证明数据竞争的存在，互斥对象的行为有些类似原子操作。例如，

若

```
mutex m;
```

则语句

```
m.lock();
```

执行该语句之后，该线程即拥有了该互斥对象；该线程可执行相应的操作。一旦操作发生错误，系统会抛出异常（`system_error`）。错误的代码通常包含以下 3 种：

- `operation_not_permitted`。该错误代码代表无操作权限。
- `resource_deadlock_would_occur`。该错误代码代表可能发生死锁现象。
- `device_or_resource_busy`。该错误代码代表程序中互斥类型对象已经锁定，阻塞该线程是不可能的。再如，

语句

```
m.try_lock();
```

执行该语句之后，该线程尝试获取互斥类型对象的所有权。如果不能获取所有权，`try_lock()` 函数迅速返回，不产生任何作用。即使其他线程没有占有改互斥类型对象时，也有可能尝试锁定（`try_lock()`）失败。又如，

语句

```
m.unlock();
```

该语句执行的前提条件是该线程已经拥有互斥类型 `m` 的所有权。执行该语句之后，该互斥类型的所有权即被释放。

1. 类 `mutex`

类 `mutex` 的声明形式如下：

```
class mutex {
public:
    constexpr mutex() noexcept;
    ~mutex();
    mutex(const mutex&) = delete;
    mutex& operator = (const mutex&) = delete;
    void lock();
    bool try_lock();
    void unlock();
    typedef implementation_defined native_handle_type;
    native_handle_type native_handle();
};
```

由上述内容可知，该类包含两个构造函数、一个析构函数、一个锁函数（`lock()`）、一个尝试锁函数（`try_lock()`）、一个解锁函数、一个赋值函数 `operator = ()` 以及 `native_handle()` 函数。

类 `mutex` 可以提供非递归互斥对象的唯一占有权。当某互斥类型对象被一个线程占有时，另一个线程无法获取该互斥类型对象的所有权，只有等待该线程释放（`unlock()`）互斥类型对象的所有权之后，才能获取该互斥类型对象的所有权。

下面使用例 16-4 来说明类 `mutex` 的使用方法。

例 16-4

```
#include <iostream>
#include <thread>
#include <mutex>
#include <chrono>
using namespace std;
mutex m,m2;
long shareddata[11];
void thread_procl ()
{   int index=0;
    for (index=0;index<5;index++)
    {
        m.try_lock();
        shareddata[2* index]=2* index;
        std::cout<<"thread 1 -- shareddata: "<<shareddata[2* index]<<std::endl;
        m.unlock();
        chrono::milliseconds sd(10);
        this_thread::sleep_for(sd);                //休眠10ms
    }
    std::cout<<"thread 1:  exit (0);"<<std::endl;
}
void thread_proc2 ()
{   int index_2=0;
    for (index_2=0; index_2<5; index_2++)
    {
        m.lock ();
        shareddata [2* index_2+1] =2* index_2+1;
        std:: cout<<" thread 2 -- shareddata: " <<shareddata [2* index_2+1] <<std::
endl;
        m.unlock ();
        chrono:: milliseconds sd (10);
        this_thread:: sleep_for (sd);                //休眠10ms
    }
    std:: cout<<" thread 2:  exit (0);" <<std:: endl;
}
int main (int argc, char* argv [])
{
    thread t1 (thread_procl);
    thread t2 (thread_proc2);
    t1.detach ();
    t2.join ();
    cin.get ();
    return 0;
}
```

例 16-4 的执行效果如图 16-4 所示。

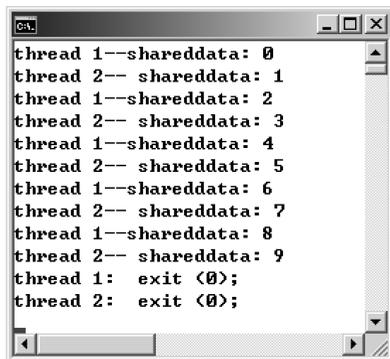


图 16-4 例 16-4 的执行效果

2. 递归式互斥对象类 recursive_mutex

递归式互斥对象类 (recursive_mutex) 的声明形式如下:

```
class recursive_mutex {
public:
    recursive_mutex();
    ~recursive_mutex();
    recursive_mutex(const recursive_mutex&) = delete;
    recursive_mutex& operator = (const recursive_mutex&) = delete;
    void lock();
    bool try_lock() noexcept;
    void unlock();
    typedef implementation_defined native_handle_type;
    native_handle_type native_handle();
};
```

由上述内容可知, 该类包含两个构造函数、一个析构函数、一个锁函数、一个解锁函数、一个尝试锁函数 (try_lock) 以及 native_handle() 函数。

该模板类提供了一个递归式互斥类型, 并且具有唯一的所有权。如果线程拥有该递归式互斥对象, 另一个线程尝试获取所有权时将失败或者发生阻塞, 直到第一个线程完全释放该所有权才能恢复。递归式互斥对象满足互斥类型的所有性能和要求。

通过调用 lock() 函数或者 try_lock(), 拥有递归式互斥对象的任意线程将获取额外级别的所有权。单个线程不确定能获得多少个级别的所有权。任意线程一旦获取最大级别的所有权, 再调用 try_lock() 时将失败, 如果再调用 lock() 会抛出 system_error 类型的异常。通过调用 lock() 和 try_lock(), 获取的每个级别的所有权, 线程需要调用 unlock() 解锁。只有该线程的所有级别的所有权全部释放之后, 其他线程才能获取获得所有权。

如果线程中的互斥对象遭到破坏, 或者当拥有线程所有权时发生线程终止的情况, 那么程序会发生不确定行为。

互斥对象的主要职责是保护共享数据。与独占式互斥量不同的是, 同一个线程在互斥量没有解锁的情况下可以再次进行加锁, 不过它们的加解锁次数需要一致。递归式互斥量可能用得比较少些。

通过以上论述可知,调用线程“拥有”一个 `recursive_mutex` 了一段时间,开始时,它成功地调用 `lock()` 或 `try_lock()` 函数。在此期间,该线程可能再次调用 `lock()` 或 `try_lock()`。所有权的时限结束时,线程会根据匹配的次数相应地调用 `unlock()`。

例 16-5

```
#include <iostream>
#include <thread>
#include <mutex>
#include <chrono>
using namespace std;
recursive_mutex mt;
long dim[11];
void mysleep(long ms)
{
    chrono::milliseconds md(10);
    this_thread::sleep_for(md);
}
void thread_procl()
{
    int index=0;
    for(index=0;index<5;index++)
    {
        mt.lock();
        dim[index* 2]=1;
        cout<<"The First Thread.. : " <<dim[index* 2+1] <<endl;
        mt.unlock();
        mysleep(10);
    }
    cout<<"The First Thread.. exit(0). " <<endl;
}
void thread_proc2()
{
    int index=0;
    for(index=0;index<5;index++)
    {
        mt.lock();
        dim[index* 2+1]=2;
        cout<<"The Second Thread.. : " <<dim[index* 2+1] <<endl;
        mt.unlock();
        mysleep(10);
    }
    cout<<"The Second Thread.. exit(0). " <<endl;
}
int main(int argc, char* argv[])
{
    thread t1(thread_procl);
```

```
thread t2(thread_proc2);  
t1.join();  
t2.join();  
cin.get();  
return 0;  
}
```

例 16-5 的执行效果如图 16-5 所示。

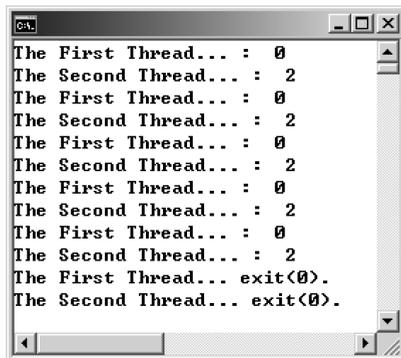


图 16-5 例 16-5 的执行效果

由以上示例可知，递归式互斥量可以代替普通互斥量。如果在程序中去掉互斥量的使用，程序中两个线程的竞争将是无序的，程序的输出也是乱序的。

3. 定时式互斥对象类 `timed_mutex`

定时式互斥对象类的声明形式如下：

```
class timed_mutex {  
public:  
    timed_mutex();  
    ~timed_mutex();  
    timed_mutex(const timed_mutex&) = delete;  
    timed_mutex& operator=(const timed_mutex&) = delete;  
    void lock();  
    bool try_lock();  
template <class Rep, class Period> bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);  
template <class Clock, class Duration> bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);  
    void unlock();  
    native_handle_type native_handle();  
};
```

该类和普通互斥量对象相比，增加了两个函数：`try_lock_for()`和`try_lock_until()`。

`try_lock_for()`函数接受一个时间范围作为参数，表示在这一段时间范围之内，若线程没有获得锁，则该线程被阻塞相应的时间长度；若在此期间其他线程释放了锁，则该线程可以

获得对互斥量的锁。若超时（即在指定时间内还是没有获得锁），则返回 `false`。

`try_lock_until()` 函数则接受一个时间点（绝对时间）作为参数，表示在指定时间点未到来之前若线程没有获得锁，则被阻塞住；若在此期间其他线程释放了锁，则该线程可以获得对互斥量的锁。如果超时（即在指定时间内还是没有获得锁），则返回 `false`。

定时式互斥对象提供了一种非递归式互斥量，该互斥量具有唯一的被占有权。如果一个线程拥有一个定时式互斥对象，那么另一个线程尝试获取该互斥对象的所有权时将发生失败（`try_lock`）或发生阻塞（`try_lock_for` 和 `try_lock_until`），直到拥有互斥对象权限的线程完全释放该所有权时，另一个线程才能获取该对象的所有权。

当发生以下情况时，程序可能会发生不确定性行为：

- 任意线程拥有的定时式互斥对象发生破坏。
- 当互斥对象尚没有被解锁时，线程发生终止或者结束。

例 16-6

```
#include <iostream>
#include <thread>
#include <chrono>
#include <mutex>
using namespace std;
timed_mutex tm;
void job1 ()
{
    bool bl = 0;
    int count = 5;
    while (count --)
    {
        bl = tm.try_lock_for (std::chrono::milliseconds (9));
        cout << "Thread 1: " << count << endl;
        if (bl)
            tm.unlock ();
    }
    std::cout << "Thread 1... exit (0). " << std::endl;
}
void job2 ()
{
    bool bl = 0;
    int count = 5;
    while (count --)
    {
        bl = tm.try_lock_for (std::chrono::milliseconds (11));
        cout << "Thread 2: " << count << endl;
        if (bl)
            tm.unlock ();
    }
    std::cout << "Thread 2... exit (0). " << std::endl;
}
```

```
    }  
    int main(int argc, char* argv[])  
    {  
        thread t1(job1);  
        thread t2(job2);  
        t1.join();  
        t2.join();  
        cin.get();  
        return 0;  
    }  
}
```

例 16-6 的执行效果如图 16-6 所示。

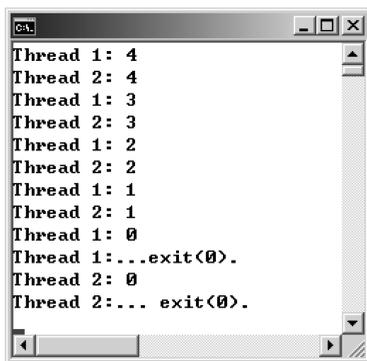


图 16-6 例 16-6 的执行效果

4. 递归式定时互斥对象类 recursive_timed_mutex

递归式定时互斥对象类的声明形式如下：

```
class recursive_timed_mutex {  
public:  
    recursive_timed_mutex();  
    ~recursive_timed_mutex();  
    recursive_timed_mutex(const recursive_timed_mutex&) = delete;  
    recursive_timed_mutex& operator=(const recursive_timed_mutex&) = delete;  
    void lock();  
    bool try_lock() noexcept;  
    template <class Rep, class Period>  
    bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);  
    template <class Clock, class Duration>  
    bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);  
    void unlock();  
    typedef implementation_defined native_handle_type;  
    native_handle_type native_handle();  
};
```

由以上内容可知，递归式定时互斥对象类的成员函数和定时式互斥对象类的成员函数基

本相同——既有递归式互斥的多锁作用，又有定时互斥的作用。

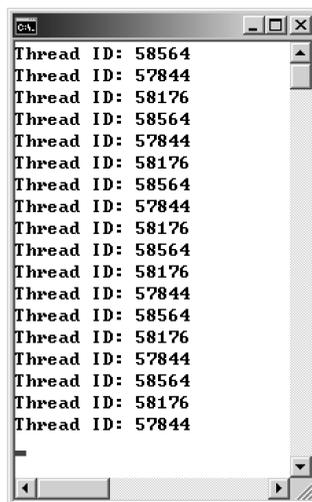
下面以例 16-7 来说明递归式定时互斥对象类的使用方法。

例 16-7

```
#include <iostream>
#include <thread>
#include <chrono>
#include <mutex>
using namespace std;
recursive_timed_mutex rtm;
void mysleep(short s)
{
    this_thread::sleep_for(chrono::seconds(s));
}
void job1()
{
    long cnt = 6;
    bool bl;
    while(cnt --)
    {
        bl = rtm.try_lock();
        if(bl)
        {
            cout << "Thread ID: " << this_thread::get_id() << endl;
            rtm.unlock();
        }
        mysleep(1);
    }
}
void job2()
{
    long cnt = 6;
    bool bl;
    while(cnt --)
    {
        bl = rtm.try_lock_for(chrono::milliseconds(10));
        if(bl)
        {
            cout << "Thread ID: " << this_thread::get_id() << endl;
            rtm.unlock();
        }
        mysleep(1);
    }
}
void job3()
{
```

```
bool bl;  
long cnt = 6;  
while(cnt --)  
{  
    bl = bl = rtm.try_lock_until(chrono::steady_clock::now() + chrono::milliseconds  
(10));  
    if(bl)  
    {  
        cout << "Thread ID: " << this_thread::get_id() << endl;  
        rtm.unlock();  
    }  
    mysleep(1);  
}  
}  
int main(int argc, char* argv[])  
{  
    thread t1(job1);  
    thread t2(job2);  
    thread t3(job3);  
    t1.join();  
    t2.join();  
    t3.join();  
    cin.get();  
    return 0;  
}
```

例 16-7 的执行效果如图 16-7 所示。



```
Thread ID: 58564  
Thread ID: 57844  
Thread ID: 58176  
Thread ID: 58564  
Thread ID: 57844  
Thread ID: 58176  
Thread ID: 58564  
Thread ID: 57844  
Thread ID: 58176  
Thread ID: 58564  
Thread ID: 57844  
Thread ID: 58176  
Thread ID: 58564  
Thread ID: 58176  
Thread ID: 57844  
Thread ID: 58564  
Thread ID: 58176  
Thread ID: 57844
```

图 16-7 例 16-7 的执行效果

16.3.2 lock 模板类

锁是一种互斥对象，其确保可锁定性对象的引用，并解锁可锁定性对象。可执行程序

(或线程) 可以使用一个锁, 以帮助管理可锁定性对象的占有权。锁通常可以占有可锁定性对象, 但不能管理可锁定对象的终个寿命期。一些锁的构造函数具有附加参数, 在锁对象存续期间, 这些附加参数可以处理可锁定对象。

本小节主要讲述两种互斥锁: `lock_guard` 和 `unique_lock`。

1. `lock_guard`

`lock_guard` 模板类的声明形式如下:

```
template <class Mutex > class lock_guard {
public:
    typedef Mutex mutex_type;
    explicit lock_guard(mutex_type& m);
    lock_guard(mutex_type& m, adopt_lock_t);
    ~lock_guard();
    lock_guard(lock_guard const&) = delete;
    lock_guard& operator = (lock_guard const&) = delete;
private:
    mutex_type& pm;
}
```

`lock_guard` 模板类包含两个构造函数, 这两个构造函数不能复制、不能移动, 且不能使用赋值符号。

`lock_guard` 类型对象可以控制可锁定对象的占有权, 使其在占有权在某范围内有效。`lock_guard` 对象确保可锁定对象的占有权贯穿 `lock_guard` 类型对象的整个生命期。`lock_guard` 和 `mutex` 类对象配合使用, 一旦将锁放进 `lock_guard` 中 (作为 `lock_guard` 构造函数的参数), `mutex` 自动加锁; `lock_guard` 析构时, 互斥类型对象 `mutex` 自动解锁。由此可知, `lock_guard` 类型对象是自动加锁和自动解锁的。`lock_guard` 是最简单的加锁、解锁办法。

`lock_guard` 模板类包含两种形式的构造函数:

```
explicit lock_guard(mutex_type& m);
lock_guard(mutex_type& m, adopt_lock_t);
```

其中, 第一种形式是自动加锁; 第二种形式包含两个参数, 第二个参数代表采用第一个参数指定的锁对象。`lock_guard` 模板类的具体使用方法详见例 16-8。

例 16-8

```
#include <iostream>
#include <thread>
#include <mutex>
#include <chrono>
using namespace std;
mutex mt;
void mysleep(int ms)
{
    chrono::seconds m(ms);
    this_thread::sleep_for(m);
}
```

```
void job1 ()
{
    long cnt =3;
    while(cnt --)
    {
        lock_guard<mutex> lg(mt);
        cout << "Thread ID : " << this_thread::get_id() << endl;
        mysleep(1);
    }
}
void job2 ()
{
    long cnt =3;
    while(cnt --)
    {
        mt.lock();
        lock_guard<mutex> lg(mt, adopt_lock);
        cout << "Thread ID : " << this_thread::get_id() << endl;
        mysleep(1);
    }
}
int main(int argc, char* argv[])
{
    thread t1(job1);
    thread t2(job2);
    t1.join();
    t2.join();
    cin.get();
    return 0;
}
```

例 16-8 的执行效果如图 16-8 所示。

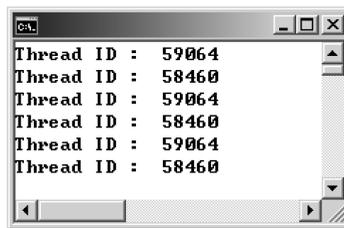


图 16-8 例 16-8 的执行效果

2. unique_lock 模板类

unique_lock 模板类的声明形式如下：

```
template <class Mutex> class unique_lock {
public:
    typedef Mutex mutex_type;
```

```

    unique_lock() noexcept;
    explicit unique_lock(mutex_type& m);
    unique_lock(mutex_type& m, defer_lock_t) noexcept;
    unique_lock(mutex_type& m, try_to_lock_t);
    unique_lock(mutex_type& m, adopt_lock_t);
template <class Clock, class Duration> unique_lock(mutex_type& m, const chrono::time_point<Clock,
Duration>& abs_time);
template <class Rep, class Period> unique_lock(mutex_type& m, const chrono::duration<Rep, Period
>& rel_time);
    ~unique_lock();
    unique_lock(unique_lock const&) = delete;
    unique_lock& operator = (unique_lock const&) = delete;
    unique_lock(unique_lock&& u) noexcept;
    unique_lock& operator = (unique_lock&& u) noexcept;
    void lock();
    bool try_lock();
template <class Rep, class Period> bool try_lock_for(const chrono::duration<Rep, Period>& rel_
time);
template <class Clock, class Duration> bool try_lock_until(const chrono::time_point<Clock, Dura
tion>& abs_time);
    void unlock();
    void swap(unique_lock& u) noexcept;
    mutex_type * release() noexcept;
    bool owns_lock() const noexcept;
    explicit operator bool () const noexcept;
    mutex_type* mutex() const noexcept;
private:
    mutex_type * pm;
    bool owns;
};

```

`unique_lock` 模板类具有多个构造函数，因此其使用形式也是多样的。此外，该模板类还具有一个有效的赋值符号。和其他锁对象不同，`unique_guard` 还拥有 `owns_lock()`、`release()` 等函数。其常用的构造函数（即常用的使用方法）包含以下几种：

```

    unique_lock() noexcept;
    explicit unique_lock(mutex_type& m);
    unique_lock(mutex_type& m, defer_lock_t) noexcept;
    unique_lock(mutex_type& m, try_to_lock_t);
    unique_lock(mutex_type& m, adopt_lock_t);
template <class Clock, class Duration> unique_lock(mutex_type& m, const chrono::time_point<Clock,
Duration>& abs_time); //锁定到某时刻
template <class Rep, class Period> unique_lock(mutex_type& m, const chrono::duration<Rep, Period
>& rel_time);
    unique_lock(unique_lock&& u) noexcept; //锁定一段时间

```

其中涉及 3 个参数：`defer_lock_t`、`try_to_lock` 和 `adopt_lock_t`。其意义分别如下：

- `defer_lock`。不在构造器中对互斥量加锁，假定该互斥量在线程中加锁。
- `try_lock`。通过调用成员函数 `try_lock` 在构造器中加锁。
- `adopt_lock`。采用当前的互斥量（锁），假定其在线程中加锁。

参数 `abs_time` 代表绝对时间（时刻），而参数 `rel_time` 代表相对时间。

类的加锁和解锁是通过成员函数 `lock`、`try_lock`、`try_lock_until`、`try_lock_for` 和 `unlock` 实现的。成员函数 `swap`、`release` 可以用于 `unique_guard` 对象。其中 `release()` 函数将 `unique_guard` 类对象管理的所有互斥锁全部释放，并返回该互斥锁的指针；`mutex()` 函数将获取当前的互斥锁；`owns_lock()` 函数用于判断互斥锁是否被锁定。

值得注意的是，由于 `unique_lock` 模板类具有的多种构造函数以及诸多成员函数，因此在定义其对象时，要注意模板中参数 `class Mutex` 的替换，详见例 16-9。

```
template <class Mutex > class unique_lock
```

例 16-9

```
#include <iostream>
#include <thread>
#include <mutex>
#include <chrono>
#include <ctime>
#include <ratio>
using namespace std;
std::mutex mt;
std::recursive_timed_mutex mt1;
void job1()
{
    unique_lock<mutex> lgm(mt,defer_lock);
    lgm.lock();
    std::cout << "Thread 1: ... " << std::endl;
    lgm.unlock();
}
void job2()
{
    unique_lock<mutex> ulm(mt);
    std::cout << "Thread 2: ... " << std::endl;
}
void job3()
{
    unique_lock<mutex> ulm3(mt,try_to_lock);
    if(ulm3.owns_lock())
    {
        std::cout << "Thread 3: " << "owned... " << std::endl;
        mutex* pm = ulm3.release();
        pm->unlock();
    }
    else
```

```
{
    std::cout << "Thread 3: " << "not owned... " << std::endl;
}
}
void job4 ()
{
    unique_lock <mutex > ulm4;
    ulm4.swap(unique_lock <mutex > (mt));
    if(ulm4.owns_lock())
    {
        std::cout << "Thread 4: owned... " << std::endl;
    }
}
void job5 ()
{
    mt.lock();
    unique_lock <mutex > ulm5(mt,adopt_lock);
    std::cout << "Thread 5: ... " << std::endl;
}
void job6 ()
{
    //chrono::seconds dura(3);
    const chrono::duration <int > dur(3);
    unique_lock <recursive_timed_mutex > ulm6(mt1,dur);
    std::cout << "Thread 6: ... " << std::endl;
}
void job7 ()
{
    chrono::steady_clock::time_point t1 = chrono::steady_clock().now();
    chrono::steady_clock::time_point t2 = t1 + chrono::seconds(1);
    unique_lock <recursive_timed_mutex > ulm7(mt1,t2);
    if(! ulm7.owns_lock())
    {
        std::cout << "Thread 7: not owned. " << std::endl;
    }
    else
    {
        std::cout << "Thread 7:  owned.. " << std::endl;
        ulm7.unlock();
    }
}
void job8 ()
{
    unique_lock <recursive_timed_mutex > ulm8(mt1,defer_lock);
    ulm8.lock();
    if(ulm8.owns_lock())
```

```
{
    std::cout << "Thread 8: lock() usage..." << std::endl;
    ulm8.unlock();
}
ulm8.try_lock();
if(ulm8.owns_lock())
{
    std::cout << "Thread 8: try_lock() usage..." << std::endl;
    ulm8.unlock();
}
ulm8.try_lock_for(chrono::seconds(1));
if(ulm8.owns_lock())
{
    std::cout << "Thread 8: try_lock_for() usage..." << std::endl;
    ulm8.unlock();
}
ulm8.try_lock_until(chrono::steady_clock().now() + chrono::seconds(1));
if(ulm8.owns_lock())
{
    std::cout << "Thread 8: try_lock_until() usage..." << std::endl;
    ulm8.unlock();
}
}
int main(int argc, char* argv[])
{
    thread t1(job1);
    thread t2(job2);
    thread t3(job3);
    thread t4(job4);
    thread t5(job5);
    thread t6(job6);
    thread t7(job7);
    thread t8(job8);

    t1.join();
    t2.join();
    t3.join();
    t4.join();
    t5.join();
    t6.join();
    t7.join();
    t8.join();
    cin.get();
    return 0;
}
```

例 16-9 的执行效果如图 16-9 所示。

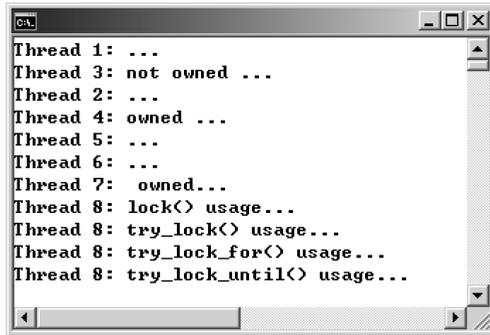


图 16-9 例 16-9 的执行效果

16.3.3 call_once

`call_once()` 是命名空间 `std` 中的公有成员函数。其函数声明形式为:

```
template<class Callable, class ...Args > void call_once (once_flag& flag, Callable&& func,
Args&&... args);
```

使用 `call_once()` 函数必然需要使用结构化标识 `once_flag`。该结构是不透明的数据结构, 在调用 `call_once` 函数时, 该结构可以确保不引起数据竞争或者死锁。

```
struct once_flag
{
    constexpr once_flag() noexcept;
    once_flag(const once_flag&) = delete;
    once_flag& operator = (const once_flag&) = delete;
};
```

该结构的构造函数为:

```
constexpr once_flag() noexcept;
```

下面讲述 `call_once()` 函数的详细使用方法。

`call_once()` 函数的执行是需要被调用的, 不是被动执行, 是主动执行。主动执行该函数时, 应该调用 `INVOKE (DECAY_COPY (std::forward<Callable> (func)))`, `DECAY_COPY (std::forward<Args> (args)) ...`)。若该调用抛出异常 (`system_error`), 则该执行过程是异常的; 否则, 函数会返回。

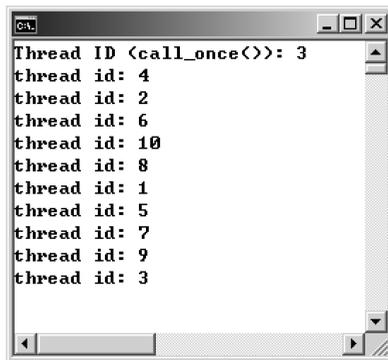
`once_flag` 相当于一个锁, 使用它的线程都会在上面等待, 但只有一个线程允许执行。如果该线程抛出异常, 那么从等待中的线程中选择一个, 重复上面的流程。

例 16-10

```
#include <iostream>
#include <thread>
#include <mutex>
#include <chrono>
using namespace std;
int winner; //公共数据,仅初始化一次即可
```

```
std::once_flag winner_flag; //标识
void set_winner (int x)
{   winner = x;
    std::cout << "Thread ID (call_once()): " << winner << std::endl;
}
void job1 (int id)
{   std::call_once (winner_flag, set_winner, id+2);
    std::this_thread::sleep_for (std::chrono::milliseconds (1000));
    std::cout << "thread id: " << id << std::endl;
}
void job2 (int id)
{   std::call_once (winner_flag, set_winner, id+2);
    std::this_thread::sleep_for (std::chrono::milliseconds (500));
    std::cout << "thread id: " << id << std::endl;
}
int main ()
{   std::thread threads [10];
    for (int i=0; i<5; ++i)
        threads [i* 2] = std::thread (job1, i* 2 +1);
    for (int i=0; i<5; ++i)
        threads [i* 2 +1] = std::thread (job2, i* 2 +1 +1);
    for (auto& th : threads) th.join();
    cin.get();
    return 0;
}
```

例 16-10 的执行效果如图 16-10 所示。



```
Thread ID <call_once(>>): 3
thread id: 4
thread id: 2
thread id: 6
thread id: 10
thread id: 8
thread id: 1
thread id: 5
thread id: 7
thread id: 9
thread id: 3
```

图 16-10 例 16-10 的执行效果

由程序的执行结果可知，set_winner() 函数仅被执行了 1 次，而线程总数目是 10 个。可见 call_once() 函数是发挥了作用的。

16.4 条件变量

C++ STL 提供了条件变量的支持。使用条件变量时，程序必须包含头文件 <condition_

variable >。该头文件主要包含了与条件变量相关的类和函数。相关的类包括 condition_variable 和 condition_variable_any，枚举类型 cv_status 以及 notify_all_at_thread_exit() 函数。

条件变量主要用于阻塞一个线程，直至被其他线程通知“条件满足或者到来了”。条件变量类 condition_variable 提供一种条件变量，该变量可以等待一个 unique_lock < mutex > 类的对象允许一些平台的最大效率。条件变量类 condition_variable_any 提供一种通用的条件变量 (condition_variable)。该条件变量可以等待任意用户提供的锁类型。

条件变量允许并发式激活：wait、wait_for、wait_until、notify_one 和 notify_all 等成员函数。notify_one 和 notify_all 函数的执行应该是原子性的。wait()、wait_for() 函数和 wait_until() 应该在以下 3 种情况下才被具体调用：

- 1) 释放互斥量，转而进入等待状态。
- 2) 非阻塞性的等待。
- 3) 重新获取阻塞状态。

16.4.1 类 condition_variable

类 condition_variable 的声明形式为：

```
class condition_variable {
public:
    condition_variable();
    ~condition_variable();
    condition_variable(const condition_variable&) = delete;
    condition_variable& operator = (const condition_variable&) = delete;
    void notify_one() noexcept;
    void notify_all() noexcept;
    void wait(unique_lock<mutex>& lock);
    template <class Predicate >
    void wait(unique_lock<mutex>& lock, Predicate pred);
    template <class Clock, class Duration >
    cv_status wait_until(unique_lock<mutex>& lock,
        const chrono::time_point<Clock, Duration>& abs_time);
    template <class Clock, class Duration, class Predicate >
    bool wait_until(unique_lock<mutex>& lock,
        const chrono::time_point<Clock, Duration>& abs_time,
        Predicate pred);
    template <class Rep, class Period >
    cv_status wait_for(unique_lock<mutex>& lock,
        const chrono::duration<Rep, Period>& rel_time);
    template <class Rep, class Period, class Predicate >
    bool wait_for(unique_lock<mutex>& lock,
        const chrono::duration<Rep, Period>& rel_time,
        Predicate pred);
    typedef implementation_defined native_handle_type;
    native_handle_type native_handle();
};
```

条件变量类 `condition_variable` 除了包含构造函数和析构函数之外，还包含 3 个等待函数：`wait`、`wait_for` 和 `wait_until`；两个 `notify` 通知函数：`notify_one()` 和 `notify_all()`。

- `wait()` 函数用于等待通知 (`notify`)。
- `wait_for()` 函数用于等待某时间长度，或者等待至通知。
- `wait_until()` 函数用于等待被通知，或者等待至某时间点。
- `notify_one()` 函数用于通知一个线程。
- `notify_all()` 函数用于通知所有线程。

公用 `notify_all_at_thread_exit()` 函数的用法为：

```
void notify_all_at_thread_exit(condition_variable& cond, unique_lock<mutex> lk);
```

`notify_all_at_thread_exit()` 函数的功能是：当调用本函数的线程退出时，所有依托条件变量 `cond` 上等待的线程全部收到通知。

下面以例 16-11 来说明条件变量类 `condition_variable` 的用法，以例 16-12 来说明 `std` 命名空间公用 `notify_all_at_thread_exit()` 函数的用法，以例 16-13 来说明 `notify_one()` 的用法，以例 16-14 来说明 `wait_for()` 和 `wait_until()` 的用法。

例 16-11

```
#include <iostream> // std::cout
#include <thread> // std::thread
#include <mutex> // std::mutex, std::unique_lock
#include <condition_variable> // std::condition_variable

std::mutex mtx;
std::condition_variable cv;
bool ready = false;
void print_id (int id) {
    std::unique_lock<std::mutex> lck (mtx);
    while (! ready) cv.wait (lck);
    std::cout << " thread " << id << '\n';
}
void go ()
{
    std::unique_lock<std::mutex> lck (mtx);
    ready = true;
    cv.notify_all ();
}
int main ()
{
    std::thread threads [10];
    for (int i=0; i<10; ++i)
        threads [i] = std::thread (print_id, i);
    std::cout << " 10 threads ready to race... \n";
    go ();
    for (auto& th : threads) th.join ();
}
```

```
std::cin.get();  
return 0;  
}
```

例 16-11 的执行效果如图 16-11 所示。

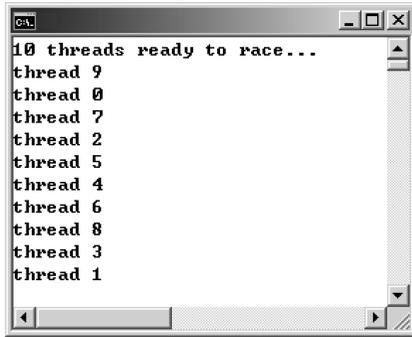


图 16-11 例 16-11 的执行效果

例 16-12

```
#include <iostream> // std::cout  
#include <thread> // std::thread  
#include <mutex> // std::mutex, std::unique_lock  
#include <condition_variable> // std::condition_variable  
using namespace std;  
std::mutex mtx;  
std::condition_variable cv;  
bool ready = false;  
void print_id (int id)  
{  
    std::unique_lock<std::mutex> lck(mtx);  
    while (! ready) cv.wait(lck);  
    std::cout << "thread " << id << '\n';  
}  
void go ()  
{  
    std::unique_lock<std::mutex> lck(mtx);  
    std::notify_all_at_thread_exit(cv, std::move(lck));  
    ready = true;  
    cout << "Thread go... " << endl;  
}  
int main ()  
{  
    std::thread threads[10];  
    // spawn 10 threads:  
    for (int i=0; i<10; ++i)  
        threads[i] = std::thread(print_id, i);
```

```

std::cout << "10 threads ready to race... \n";
std::thread(go).detach(); // go!
for (auto& th : threads) th.join();
std::cin.get();
return 0;
}

```

例 16-12 的执行效果如图 16-12 所示。

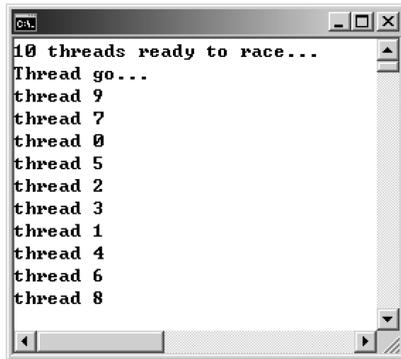


图 16-12 例 16-12 的执行效果

例 16-13

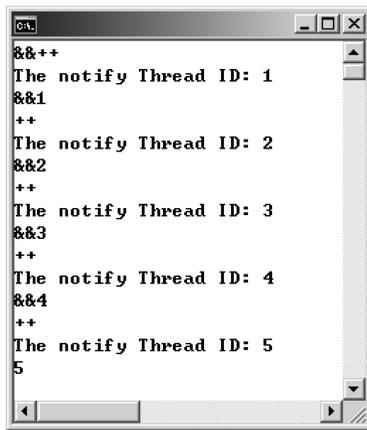
```

#include <iostream> // std::cout
#include <thread> // std::thread
#include <mutex> // std::mutex, std::unique_lock
#include <condition_variable> // std::condition_variable
using namespace std;
std::mutex mtx;
std::condition_variable cv1, cv2;
int cargo = 0;
void job2 () //消费者
{
    std::unique_lock<std::mutex> lck(mtx);
    cout << "&&";
    while (cargo == 0)
        cv2.wait(lck);
    cout << cargo << endl;
    cargo = 0;
    cv1.notify_one();
}
void job1 (int id) //生产者
{
    std::unique_lock<std::mutex> lck(mtx);
    cout << "++ " << endl;
    while (cargo! = 0)

```

```
        cv1.wait(lck);
        cargo = id;
        cout << "The notify Thread ID: " << id << endl;
        cv2.notify_one();
    }
int main ()
{
    std::thread t1[5],t2[5];
    for (int i=0; i<5; ++i)
    {
        t1[i] = std::thread(job2);
        t2[i] = std::thread(job1,i+1);
    }
    for (int i=0; i<5; ++i)
    {
        t2[i].join();
        t1[i].join();
    }
    cin.get();
    return 0;
}
```

例 16-13 的执行效果 (请读者根据输出结果, 琢磨一下线程的执行顺序) 如图 16-13 所示。



```
cmd
&&++
The notify Thread ID: 1
&&1
++
The notify Thread ID: 2
&&2
++
The notify Thread ID: 3
&&3
++
The notify Thread ID: 4
&&4
++
The notify Thread ID: 5
5
```

图 16-13 例 16-13 的执行效果

在讲述 `wait_for()` 和 `wait_until()` 两个成员函数时, 首先需要讲述的是枚举类型 `cv_status`。枚举类型 `cv_status` 包含如下两个枚举值:

1) `cv_status::no_timeout`。该枚举值表示 `wait_for` 或者 `wait_until` 没有超时, 即在规定的时段内线程收到了通知。

2) `cv_status::timeout`。该枚举值表示 `wait_for` 或者 `wait_until` 超时。

`wait_for()` 函数和 `wait_until()` 函数均有两种使用形式, 本节给出其中一种, 另一种读者可自己耐心琢磨一下。

例 16-14

```
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <chrono>
using namespace std;
mutex mt;
condition_variable cv1, cv2, cv3;

void job1(int arg)
{
    bool bl=0;
    std::unique_lock<std::mutex> lck1(mt);
    bl = cv1.wait_for(lck1, chrono::seconds(8));
    if(bl)
    {
        cout << "wait_for timeout. " << endl;
    }
    else
    {
        cout << "wait_for timing. " << endl;
    }
    cout << "Thread " << arg << endl;
}

void job2(int arg)
{
    bool bl=0;
    std::unique_lock<std::mutex> lck2(mt);
    bl = cv2.wait_for(lck2, chrono::seconds(7));
    if(bl)
    {
        cout << "wait_for timeout. " << endl;
    }
    else
    {
        cout << "wait_for timing. " << endl;
    }
    cout << "Thread " << arg << endl;
}

void job3(int arg)
{
    bool bl=0;
    std::unique_lock<std::mutex> lck3(mt);
    bl = cv3.wait_for(lck3, chrono::seconds(3));
```

```
        if (b1)
        {
            cout << "wait_for timeout. " << endl;
        }
        else
        {
            cout << "wait_for timing. " << endl;
        }
        cv1.notify_one();
        cv2.notify_one();
        cout << "Thread " << arg << endl;
    }
void job4(int arg)
{
    //bool b1=0;
    std::unique_lock<std::mutex> lck3(mt);
    auto nowt = chrono::system_clock().now();
    if (cv3.wait_until(lck3, chrono::seconds(arg) + nowt) == std::cv_status::timeout)
    {
        cout << "wait_for timeout. " << endl;
    }
    else
    {
        cout << "wait_for timing. " << endl;
    }
    cv3.notify_one();
    cout << "Thread 4" << endl;
}
int main(int argc, char* argv[])
{
    thread t1(job1,1);
    thread t2(job2,2);
    thread t3(job3,3);
    thread t4(job4,1);
    t1.join();
    t2.join();
    t3.join();
    t4.join();
    //cin.get();
    return 0;
}
```

例 16-14 的执行效果如图 16-14 所示。

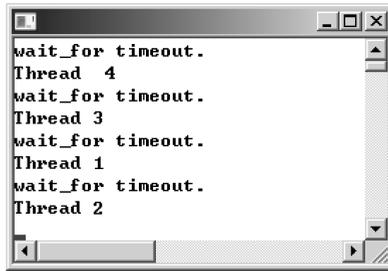


图 16-14 例 16-14 的执行效果

16.4.2 类 condition_variable_any

类 condition_variable_any 的声明形式为：

```
class condition_variable_any
{
public:
    condition_variable_any();
    ~condition_variable_any();
    condition_variable_any(const condition_variable_any&) = delete;
    condition_variable_any& operator = (const condition_variable_any&) = delete;
    void notify_one() noexcept;
    void notify_all() noexcept;
    template <class Lock> void wait (Lock& lock);
    template <class Lock, class Predicate> void wait (Lock& lock, Predicate pred);
    template <class Lock, class Clock, class Duration> cv_status wait_until
        (Lock& lock, const chrono::time_point<Clock, Duration>& abs_time);
    template <class Lock, class Clock, class Duration, class Predicate> bool wait_until
        (Lock& lock, const chrono::time_point<Clock, Duration>& abs_time, Predicate pred);
    template <class Lock, class Rep, class Period> cv_status wait_for
        (Lock& lock, const chrono::duration<Rep, Period>& rel_time);
    template <class Lock, class Rep, class Period, class Predicate> bool wait_for
        (Lock& lock, const chrono::duration<Rep, Period>& rel_time, Predicate pred);
};
```

由以上内容可知，该类包含 3 个构造函数，但最实用的还是第一种形式（无参数的）；还包含两个通知函数以及 6 个等待函数（两个 wait_for()、两个 wait_until() 和两个 wait()）。

条件变量 condition_variable_any 与 condition_variable 类似，只不过 condition_variable_any 的 wait() 函数可以接受任何可锁类型（lockable）的参数，而 condition_variable 只能接受 unique_lock < std::mutex > 类型的互斥量，除此以外，和 std::condition_variable 几乎完全

一样。

该类的构造函数不能采取复制或移动的方式初始化。

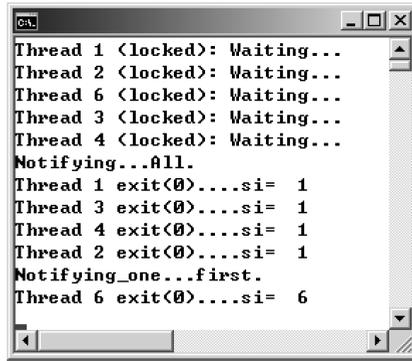
下面以示例的形式阐释条件变量类 `condition_variable_any` 的用法。例 16-15 主要用于说明 `notify_one()` 函数和 `notify_all()` 的使用方法。例 16-16 主要用于说明各个等待函数 (`wait`、`wait_for` 和 `wait_until`) 的使用用法。

例 16-15

```
#include <iostream>
#include <condition_variable>
#include <mutex>
#include <chrono>
#include <atomic>
#include <thread>
using namespace std;
std::atomic<int> si;
condition_variable_any cva, cvb;
mutex mt, mtl;
void job1 ()
{
    std::unique_lock<std::mutex> lk (mt);
    std::cout << " Thread 1 (locked): Waiting.. " <<std::endl;
    cva.wait (lk, [] {return si == 1;});
    std::cout << " Thread 1 exit (0) ... si = " <<si <<std::endl;
}
void job2 ()
{
    std::unique_lock<std::mutex> lk (mt);
    std::cout << " Thread 2 (locked): Waiting.. " <<std::endl;
    cva.wait (lk, [] {return si == 1;});
    std::cout << " Thread 2 exit (0) ... si = " <<si <<std::endl;
}
void job3 ()
{
    std::unique_lock<std::mutex> lk (mt);
    std::cout << " Thread 3 (locked): Waiting.. " <<std::endl;
    cva.wait (lk, [] {return si == 1;});
    std::cout << " Thread 3 exit (0) ... si = " <<si <<std::endl;
}
void job4 ()
{
    std::unique_lock<std::mutex> lk (mt);
    std::cout << " Thread 4 (locked): Waiting.. " <<std::endl;
    cva.wait (lk, [] {return si == 1;});
    std::cout << " Thread 4 exit (0) ... si = " <<si <<std::endl;
}
```

```
    }  
    void noti_all()  
    {  
        std::this_thread::sleep_for(std::chrono::seconds(1));  
        {  
            std::lock_guard<std::mutex> lk(mt);  
            std::cout << "Notifying... All. " << std::endl;  
        }  
        si=1;  
        cva.notify_all();  
    }  
    void job6()  
    {  
        std::unique_lock<std::mutex> lk(mt1);  
        std::cout << "Thread 6 (locked): Waiting.. " << std::endl;  
        cvb.wait(lk, []{return si == 6;});  
        std::cout << "Thread 6 exit(0)... si = " << si << std::endl;  
    }  
    void job7()  
    {  
        std::this_thread::sleep_for(std::chrono::seconds(5));  
        {  
            std::lock_guard<std::mutex> lk(mt1);  
            std::cout << "Notifying_one... first. " << std::endl;  
        }  
        si=6;  
        cvb.notify_one();  
    }  
    int main(int argc, char* argv[])  
    {  
        si.store(0);  
        thread t1(job1), t2(job2), t3(job3), t4(job4), t5(noti_all), t6(job6), t7(job7);  
        t1.join();  
        t2.join();  
        t3.join();  
        t4.join();  
        t5.join();  
        //this_thread::sleep_for(chrono::seconds(5)); //wait  
        t6.join();  
        t7.join();  
        cin.get();  
        return 0;  
    }  
}
```

例 16-15 的执行效果如图 16-15 所示。



```
Thread 1 <locked>: Waiting...
Thread 2 <locked>: Waiting...
Thread 6 <locked>: Waiting...
Thread 3 <locked>: Waiting...
Thread 4 <locked>: Waiting...
Notifying...All.
Thread 1 exit(0)...si= 1
Thread 3 exit(0)...si= 1
Thread 4 exit(0)...si= 1
Thread 2 exit(0)...si= 1
Notifying_one...first.
Thread 6 exit(0)...si= 6
```

图 16-15 例 16-15 的执行效果

例 16-16

```
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <chrono>
#include <atomic>
// #include <lock>
using namespace std;
condition_variable cv;
mutex mt;
atomic<int> ai;
void job1 ()
{
    std::unique_lock<mutex> lk (mt);
    cv.wait_for (lk, chrono::seconds (1));
    ai=1;
    cout << " Thread 1...exit (0) . ai = " << ai << endl;
}
void job2 ()
{
    std::unique_lock<mutex> lk (mt);
    cv.wait_until (lk, chrono::system_clock::now () + chrono::seconds (5));
    ai=2;
    cout << " Thread 1...exit (0) . ai = " << ai << endl;
}
void job3 ()
{
    std::unique_lock<mutex> lk (mt);
    cv.wait_for (lk, chrono::seconds (9));
    ai=3;
    cout << " Thread 1...exit (0) . ai = " << ai << endl;
}
```

```
    }  
    void noti_all()  
    {  
        this_thread::sleep_for(chrono::seconds(6));  
        cv.notify_all();  
        cout << "Thread noti_all : " << endl;  
    }  
    int main(int argc, char* argv[])  
    {  
        ai.store(0);  
        thread t1(job1), t2(job2), t3(job3), t4(noti_all);  
        t1.join();  
        t2.join();  
        t3.join();  
        t4.join();  
        cin.get();  
        return 0;  
    }  
}
```

例 16-16 的执行效果如图 16-16 所示。

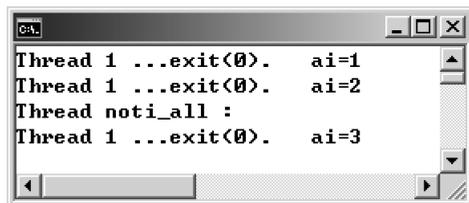


图 16-16 例 16-16 的执行效果

16.5 模板类 future

本节主要讲述 STL 所提供的获取异步任务或异常返回值的功能。这些值是和某个共享状态相联系的，异步任务可以输出它的返回值或者存储一个异常，并可以检查、等待该共享状态以及其他线程控制的共享状态，例如 `std::future` 的句柄和 `std::shared_future` 的句柄。头文件 `<future>` 中定义了：`promise`、`packeted_task`、`future`、`shared_future`、`asyn`、`launch` 和 `future_status`，还包含了 3 种错误报告的相关功能类和（`future_error`、`future_category` 函数和 `future_errc`）。这 3 种错误类放在 16.5.1 节介绍，有助于后面编程时直接使用。

16.5.1 模板类 `future_error`、`future_errc` 和 `future_category` 以及共享状态

`future` 模板类等也需要进行错误处理。

1. 错误处理

此处介绍 3 种错误处理办法：`future_category()`、`make_error_code()` 和 `make_error_condition()`。这 3 个函数的声明形式分别为：

```
const error_category& future_category () noexcept;
error_code make_error_code (future_errc e) noexcept;
error_condition make_error_condition (future_errc e) noexcept;
```

这 3 个函数的使用方法参见例 16-17。

例 16-17

```
#include <iostream>           // std::cerr
#include <future>             // std::promise, std::future_error, std::future_category

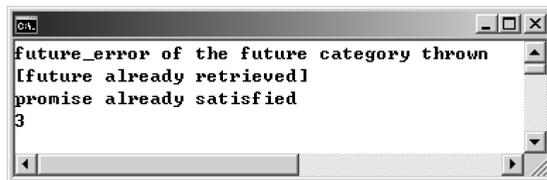
using namespace std;
int main ()
{
    std::promise<int> prom;

    try {
        prom.get_future();
        prom.get_future();    // throws a std::future_error of the future category
    }
    catch (std::future_error& e) {
        if (e.code().category() == std::future_category())
            std::cerr << "future_error of the future category thrown\n";
        if (e.code() == std::make_error_condition(std::future_errc::future_already_retrieved))
            std::cerr << "[future already retrieved] \n";
        else
            std::cerr << "[unknown exception] \n";

        error_code temp = make_error_code(future_errc::promise_already_satisfied);
        std::cout << e.what() << endl;
        cout << temp.value() << endl;
    }

    cin.get();
    return 0;
}
```

例 16-17 的执行效果如图 16-17 所示。



```
future_error of the future category thrown
[future already retrieved]
promise already satisfied
3
```

图 16-17 例 16-17 的执行效果

2. 类 future_error

类 future_error 是从 logic_error 派生而来的。其声明形式如下：

```
class future_error : public logic_error
{
public:
    future_error(error_code ec);           // exposition only
    const error_code& code() const noexcept;
    const char* what() const noexcept;
};
```

除了构造函数之外，该类仅包含两个成员函数：code()和 what()。

成员函数 code()的声明形式如下：

```
const error_code& code() const noexcept;
```

其作用是：返回传递给其构造函数的参数值。该参数值是在构造该类对象时传入的。

成员函数 what()的声明如下：

```
const char * what() const noexcept;
```

其作用是：返回异常的描述信息。

例 16-18

```
#include <iostream>           // std::cout
#include <future>             // std::promise, std::future_error
using namespace std;
int main ()
{
    std::promise<int> prom;
    try {prom.get_future();
        prom.get_future();    // throws std::future_error
    }
    catch (std::future_error& e) {
        cout << "future_error caught" << "(" << e.code() << ")" << " : " << e.what() << endl;
    }
    return 0;
}
```

例 16-18 的执行效果如图 16-18 所示。

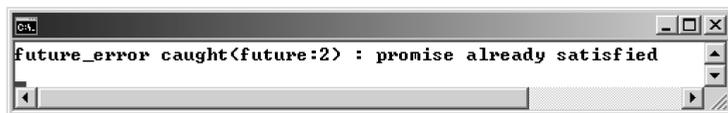


图 16-18 例 16-18 的执行效果

3. 类 future_errc

类 future_errc 是一个枚举类。其枚举值的对应表见表 16-1。

表 6-1 类 future-errc 枚举值的对应表

future_errc 标识	初始	描述
broken_promise	0	如果设置为 1 或者为一个异常，那么该 promise 和 future 的共享状态将被破坏
future_already_retrieved	1	已获取 future 类型对象
promise_already_satisfied	2	promise 类型对象被设置为一个值或异常
no_state	3	当对象没有共享状态时，如果有操作尝试获取其共享状态，那么返回值为 3

4. 共享状态

本章讲述的各种类在多数情况下是使用某个状态进行交流数据结果的，这个状态即共享状态。共享状态包含一些状态信息和一些结果，也可能是一些数值或是一种异常。

通常异步的返回对象是一种可以从“共享状态”中读取返回值对象。一种异步返回对象的等待函数是一种共享数据的阻塞等待。如果等待函数是因超时而返回的，该函数即变成时间等待函数。

异步提供者是一种给共享状态提供结果的对象。该共享状态的结果是通过相关函数设定的。设置这些共享状态的意义在于可以描述之前创建的状态对象。

当异步返回对象或者异步提供者用于释放其共享状态，这意味着：

- 1) 若返回对象或提供者保持最后的共享状态的引用，共享状态被破坏。
- 2) 返回的对象或者提供者放弃该共享状态的引用。

若异步提供者用于创建共享状态，这意味着：①提供者标记该状态为“ready”；②提供者不阻塞任意执行线程，而该线程希望等待其共享状态变为“ready”状态。

当异步提供者用于放弃它的共享状态，这意味着：

- 1) 如果该状态没有变为 ready，提供者会存储一个异常对象，并使其共享状态为 ready 状态。
- 2) 提供者释放共享状态。

如果该共享状态能保持一个数值，或一种异常需要返回，该共享状态将是 ready。等待一个共享状态直到其变为 ready，可能会激活代码而计算等待线程的结果。

某函数调用（其存储共享状态的结果）将和另一个函数调用（检测到 ready 状态，并导致设置）同步共享状态的等待函数成功返回之后，共享状态的存储会同步进行。

某些函数延时创建共享状态为 ready，直至正在运行的线程退出。在等待共享状态为 ready 之前，每个线程存储对象的破坏是有序的。

同一共享状态的结果的访问还有可能会发生冲突。

16.5.2 模板类 promise

类 promise 的声明形式如下：

```
template <class R>class promise
{
public:
    promise();
    template <class Allocator>promise(allocator_arg_t, const Allocator& a);
```

```

promise(promise&& rhs) noexcept;
promise(const promise& rhs) = delete;
~promise();
// assignment
promise& operator = (promise&& rhs) noexcept;
promise& operator = (const promise& rhs) = delete;
void swap(promise& other) noexcept;
// retrieving the result
future<R> get_future();
// setting the result
void set_value(see below);
void set_exception(exception_ptr p);
// setting the result with deferred notification
void set_value_at_thread_exit(const R& r);
void set_value_at_thread_exit(see below);
void set_exception_at_thread_exit(exception_ptr p);
};

template <class R> void swap(promise<R>& x, promise<R>& y) noexcept;
template <class R, class Alloc> struct uses_allocator<promise<R>, Alloc>;

```

由以上内容可知，该类包含 4 个构造函数、1 个返回其值的 `get_future()` 函数以及 4 个设置其值的 (`set_value()` 函数、`set_exception()`、`set_value_at_thread_exit()` (两种形式) 和 `set_exception_at_thread_exit()`)。

命名空间 `std` 中还包含两个相关的算法函数。

在 C++ 11 的联机帮助中，该类的模板也有 3 种形式：

- `template <class T> promise;`
- `template <class R> promise<R>;`
- `template <> promise<void>;`

其实 `promise` 类的对象可以存储相关类型的数值，并作为一种同步的点。在其构造函数中，`promise` 类型对象是和一种新的共享状态相关联的，这些对象可以存储一个数值或者一个异常（该异常需要是从 `exception` 类派生而来）。

通过调用 `get_future()` 函数，这个共享状态可以是和某 `future` 类型对象相关联。调用之后两个对象可以共享同一个共享对象。

- 1) `promise` 类型对象是异步提供者，并期望在某点设置该共享状态的数值。
- 2) `future` 类对象是一个异步返回对象，该对象可以返回共享状态的值，并等待其变为“ready”状态。

共享状态的生命期至少会持续到最后一个对象释放，或者最后一个对象被迫破坏掉。因此，如果它关联到某 `future` 对象时，它可以挽救该 `promise` 类型对象，并首先获得该对象。

各成员函数的功能分别介绍如下：

1) `getfuture()` 函数。该函数返回一个 `future` 类型对象，该对象对应关联一个共享状态。该 `future` 类型对象可以访问共享状态中存储的数值或者异常，其值是通过 `promise` 类型对象设置的。对于每个 `promise` 类型共享状态，仅仅可以返回一个 `future` 类型对象。该函数调用之后，

promise 类型对象期望创建共享状态为 ready。而设置共享状态为 ready，需要通过设置一个数值或者一个异常。否则，promise 对象在析构时会自动地设置一个 future_error 异常 (broken_promise) 来设置其自身的 ready 状态。简而言之，该函数的调用就是建立 promise 类型对象和 future 类型对象之间的关系。

2) setexception() 函数。该函数为 promise 类型对象设置异常，之后 promise 类型对象的共享状态标志变为 ready。

3) setvalue() 函数。该函数为 promise 类型对象设置共享状态的值，之后 promise 类型对象的共享状态标志变为 ready。

4) set_value_at_thread_exit() 函数。该函数用于设置共享状态的值，但是不将共享状态标识设置为 ready。当线程退出时，该 promise 类型对象会自动设置为 ready。若某个 std::future 对象与该 promise 对象的共享状态相关联，并且该 future 正在调用 get，则调用 get 的线程会被阻塞。当线程退出时，调用 future::get 的线程解除阻塞，同时 get 返回 set_value_at_thread_exit 所设置的值。注意：该函数已经设置了 promise 共享状态的值，若在线程结束之前有其他设置或者修改共享状态的值的操作，则会抛出 future_error (promise_already_satisfied)。

5) set_exception_at_thread_exit() 函数。该函数用于设置一个异常指针给共享状态，并不立即将共享状态设置为 ready。当该线程退出时，该函数会自动将 promise 类型对象的共享状态标志设置为 ready。

下面以例 16-19 详述类 promise 的使用方法。

注意：程序的使用分两次进行，所以包含两个效果图。

例 16-19

```
#include <iostream> // std::cout
#include <functional> // std::ref
#include <memory> // std::allocator, std::allocator_arg
#include <thread> // std::thread
#include <future> // std::promise, std::future
#include <mutex>
#include <stdlib.h>
#include <exception> // std::set_terminate
#include <cstdlib>
using namespace std;
std::mutex mt;
void job1 (std::promise<int> &prom)
{
    int x;
    lock_guard<mutex> lmt (mt);
    std::cout << "Please, enter an integer value (Thread 1): " << std::endl;
    std::cin.exceptions (std::ios::failbit); // throw on failbit
    try
    {
        std::cin >> x; // sets failbit if input is not int
        prom.set_value(x);
    }
}
```

```
    }
    catch (std::exception&)
    {
        prom.set_exception(std::current_exception());
    }
    cout << "Thread 1... exit(0). " << endl;
}

void job_print1 (std::future<int>& fut) {
    try
    {
        int x = fut.get();
        std::cout << "value: " << x << endl;
    }
    catch (std::exception& e)
    {
        std::cout << "[exception caught: " << e.what() << "]" << std::endl;
    }
}

void job3(std::promise<int>& prom)
{
    int x;
    lock_guard<mutex> lmt(mt);
    std::cout << "Please, enter an integer value(Thread 3): " << std::endl;
    std::cin.exceptions (std::ios::failbit); // throw on failbit
    try {
        std::cin >> x; // sets failbit if input is not int
        prom.set_value_at_thread_exit(x);
    }
    catch (std::exception&)
    {
        prom.set_exception_at_thread_exit(std::current_exception());
    }
    cout << "Thread 3... exit(0). " << std::endl;
}

void job_print3 (std::future<int>& fut)
{
    try
    {
        int x = fut.get();
        std::cout << "value: " << x << std::endl;
    }
    catch (std::exception& e)
    {
        std::cout << "[exception caught: " << e.what() << "]" << std::endl;
    }
}

int main ()
{
    std::promise<int> prom,prom3;
    std::future<int> fut = prom.get_future(); //获取对象
    std::future<int> fut3 = prom3.get_future(); //获取对象
    try{
        std::thread th1 (job_print1, std::ref(fut));
        std::thread th2 (job1, std::ref(prom));
```

```
std::thread th5 (job_print3, std::ref(fut3));
std::thread th6 (job3, std::ref(prom3));
th1.join();
th2.join();
th5.join();
th6.join();
}
catch(...)
{
}
return 0;
}
```

例 16-19 的执行效果如图 16-19 和图 16-20 所示。

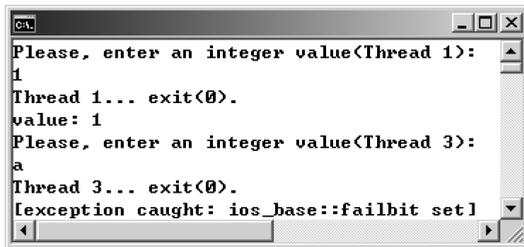


图 16-19 例 16-19 的执行效果 1

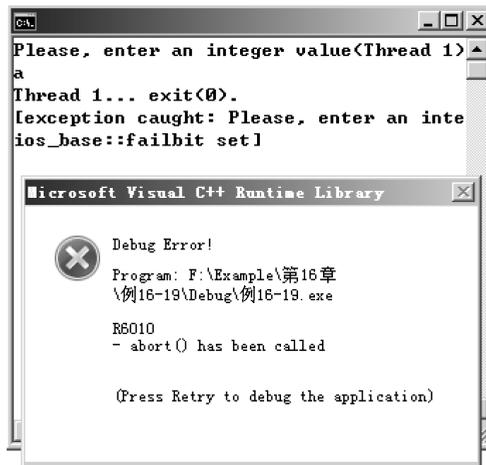


图 16-20 例 16-19 的执行效果 2

16.5.3 模板类 future

future 模板类定义了一种异步返回对象的类型，但不能和其他异步返回对象共享其共享状态。一个默认构造的 future 类型对象没有共享状态。具有共享状态的 future 类型对象可以通过异步函数创建，也可以通过移动式构造和共享其共享状态。future 类型对象可以通过相

关的函数设置其结果值，并共享同一共享状态。

调用成员函数（析构函数、赋值符号 =、valid() 等除外）的作用是不确定的。

该类的声明形式如下：

```
template <class R>class future
{
public:
future() noexcept;
future(future &&) noexcept;
future(const future& rhs) = delete;
~future();
future& operator = (const future& rhs) = delete;
future& operator = (future&&) noexcept;
shared_future<R> share ();
see below get ();
bool valid () const noexcept;
void wait () const;

template <class Rep, class Period> future_status wait_for (const chrono:: duration<Rep, Period>&
rel_time) const;
template <class Clock, class Duration> future_status wait_until (const chrono:: time_point<Clock,
Duration>&
abs_time) const;
};
```

上述类声明包含了两个有效的构造函数、一个析构函数、share() 函数、get() 函数、valid() 函数、wait() 函数、wait_for() 函数以及 wait_until() 函数。

下面逐一讲述各函数的用法：

- 1) 构造函数 future() 和 future (future&&) 用于构造该类型的对象，赋值符号 (operator =) 主要用于给该类的对象赋值。
- 2) share() 函数用于返回一个 shared_future 类型对象。
- 3) get() 函数。当共享状态是 “ready” 时，该函数返回存储在共享状态中的值。
- 4) valid() 函数用于判断该 future 类型对象是否和某共享状态关联。对于默认构造的 future 类型对象（未赋值），函数返回 false。如果函数是紧跟 get() 函数被调用，函数也返回 false。

5) wait() 函数用于等待共享状态至 “ready” 状态，若共享状态不是 “ready”，则程序被阻塞直至共享状态变为 “ready”。wait_for() 函数在时间段内等待状态变为 “ready”；wait_until() 函数是在某时刻之前等待状态变为 “ready”。

future_status 的详细定义见表 16-2。

表 16-2 future_status 的详细定义

值	描述	值	描述
future_status::ready	共享状态被设置	future_status::deferred	延迟的共享状态
future_status::timeout	共享状态超时		

例 16-20

```
#include <iostream>
#include <thread>
// #include <mutex>
#include <future>
#include <chrono>
using namespace std;

int job1 (int arg) //线程函数 1
{
    int x;
    x = arg;
    return x;
}

int job2 (int arg) //线程函数 2
{
    int y;
    y = arg;
    return y;
}

int job3 (int arg) //线程函数 3
{
    int y = 0;
    y = arg;
    return y;
}

int job4 (int arg) //线程函数 4
{
    int z = 0;
    z = arg;
    return z;
}

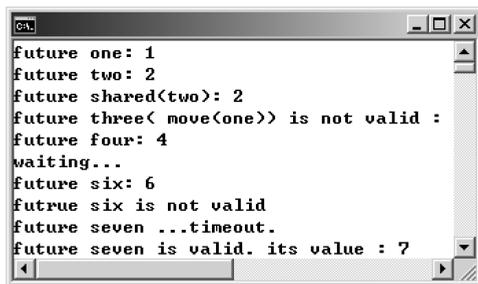
int job5 (int arg) //线程函数 5
{
    int z = 0;
    z = arg;
    this_thread::sleep_for (chrono::seconds (8));
    return z;
}

int main (int argc, char* argv [])
{
    int temp = -1;
    std::future<int> f1 = std::async (job1, 1); //wait_for ()
    while (f1.wait_for (chrono::milliseconds (100)) == std::future_status::timeout)
        std::cout << '.';
    temp = f1.get ();
    cout << " future one: " << temp << endl;
    std::future<int> f2 = std::async (job2, 2);
    temp = f2.get (); //get ()
    cout << " future two: " << temp << endl;
    std::shared_future<int> shf = f2.share (); //share ()
```

```
cout << "future shared(two): " << shf.get() << endl;
std::future<int> f3 = std::move(f1);
if(f3.valid()) //valid()
    cout << "future three( move(one)). " << f3.get() << endl;
else
    cout << "future three( move(one)) is not valid : " << endl;
std::future<int> f4 = std::async(job3,4);
std::future<int> f5 = std::move(f4);
if(f5.valid())
    cout << "future four: " << f5.get() << endl;
else
    cout << "future five( move(four)) is not valid : " << endl;
std::future<int> f6 = std::async(job4,6);
std::cout << "waiting... \n";
f6.wait(); //wait()
cout << "future six: " << f6.get() << endl;
if(f6.valid())
    cout << "future six is valid" << endl;
else
    cout << "future six is not valid" << endl;
std::future<int> f7 = std::async(job5,7);
auto fts = f7.wait_until (chrono::system_clock::now() + chrono::seconds(5)); //wait_until

if (fts == future_status:: timeout)
{
    cout << " future seven... timeout. " << endl;
}
else if (fts == future_status:: ready)
{
    cout << " future seven... ready. " << endl;
}
else
{
    cout << " future seven... deferred. " << endl;
}
if (f7.valid ())
{
    cout << " future seven is valid. its value : " << f7.get () << endl;
}
else
{
    cout << " future seven is not valid" << endl;
}
return 0;
}
```

例 16-20 的执行效果如图 16-21 所示。



```
future one: 1
future two: 2
future shared(two): 2
future three( move(one)) is not valid :
future four: 4
waiting...
future six: 6
futrue six is not valid
future seven ...timeout.
future seven is valid. its value : 7
```

图 16-21 例 16-20 的执行效果



注意 例 16-20 分别用 5 个线程和异步函数对模板类 `future` 的各个成员函数的使用方法进行了说明。此例题有助于读者理解模板类 `future`。

16.5.4 模板类 `shared_future`

`shared_future` 模板类定义了一种异步返回对象的类型。该类型可以和其他异步返回对象共享其状态。该类和 `future` 模板类很相似。`shared_future` 类型对象可以从 `future` 类型对象转换而来,也可以使用 `share()` 函数获取。但是, `future` 类型对象转换为 `shared_future` 类型对象之后,其自身即变为无效。

`shared_future` 类型比 `future` 类型更优越,一旦其共享状态变为 `ready`,其值可以返回多次,即可以多次使用 `get()` 函数获取其内存存储的数值。因此,在调用 `get()` 函数之后,其状态仍然有效,其所有权仍然有效。例 16-21 之外的使用方法参见例 16-20。

例 16-21

```
#include <iostream>
#include <future>
using namespace std;
int job1(int arg)
{
    int x=0;
    x=arg;
    return x;
}
int main(int argc, char* argv[])
{
    future<int> f1 = std::async(job1,1);
    if(f1.valid())
        cout << "future 1: " << f1.get() << endl;
    else
        cout << "future 1 is not valid. " << endl;
    future<int> f2 = std::async(job1,2);
    if(f2.valid())
        cout << "future 2: " << f2.get() << endl;
    else
```

```
cout << "future 2 is not valid ." << endl;
shared_future<int> f3 = std:: async (job1, 3);
shared_future<int> f4 = std:: move (f1);
if (f1.valid ())
    cout << " future 1: " << f1.get () << endl;
else
    cout << " future 1 is not valid. " << endl;
shared_future<int> f5 = f2.share ();
if (f2.valid ())
    cout << " future 2: " << f2.get () << endl;
else
    cout << " future 2 is not valid. " << endl;
int cnt = 3;
cout << " future 3 : " << endl;
while (cnt --)
{
    cout << f3.get () << " ;" << endl;
}
cout << " future 4: " << f4.get () << endl;
cout << " future 5: " << f5.get () << endl;
return 0;
}
```

例 16-21 的执行效果如图 16-22 所示。

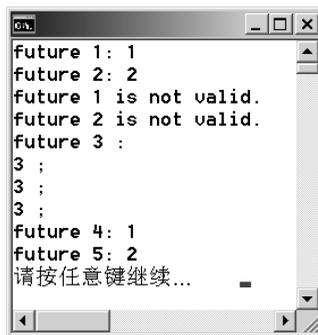


图 16-22 例 16-21 的执行效果

16.5.5 仿函数 async

STL 还提供了仿 `async()` 函数。该函数有两种形式：

```
template <class F, class... Args> future<typename result_of< F (Args...) >:: type> async (F&& f, Args&&... args);
template <class F, class... Args> future<typename result_of< F (Args...) >:: type>
    async (launch_policy, F&& f, Args&&... args);
```

其作用归根结底就是调用函数 `f`。由于调用形式是异步的，因此并不等待函数 `f` 执行完

毕即返回。函数 `f` 的返回值可以通过 `future` 类型对象进行访问，例通过 `future::get()` 函数。

第二种声明形式允许调用者选择一个特定的启动策略。第一种声明形式是自动选择启动策略，通常可用的启动策略为 `launch::async` | `launch::deferred`。

此函数可临时存储共享状态，或者是被应用的线程句柄，或者是函数的备份和参数 `args`，但并不使其状态设置为“ready”。一旦函数 `f` 的执行结束，共享状态将保存函数 `f` 的返回值，并标记状态为“ready”。

通常的启动策略有 3 种形式：`launch::async`、`launch::deferred` 以及 `launch::async` | `launch::deferred`。

1) 策略 `launch::async`。在 `async()` 函数被调用时即创建线程；返回的 `future` 类型对象被连接至被创建线程的末尾，即使共享状态不被访问，该对象的析构器和函数 `f` 的返回是同步的。因此，即使函数 `f` 返回值为 `void` 类型，其返回过程也不应该被认为是异步行为。

2) 策略 `launch::deferred`。延迟调用方式，需要等到 `future` 类对象调用 `wait()` 或者调用 `get()` 时，才创建线程。

3) 策略 `launch::async` | `launch::deferred`。自动选择策略。

函数 `f` 是以指针形式代入 `async()` 函数的。函数的返回值存储在共享状态中。如果函数 `f` 抛出异常，共享状态会被设置为异常状态。

参数 `args` 是传递给函数 `f` 的参数，其类型应该和函数 `f` 的具体参数类型相匹配。

`async()` 函数的返回值是一个 `future` 类型对象，当函数 `f` 执行完毕之后，其共享状态被设置为“ready”。其值可以通过 `future::get()` 函数获取。

说明该函数使用方法的例 16-22 其实完全可以使用 16.5.3 节的例 16-20 代替。

例 16-22

```
#pragma once
#include <iostream>
#include <future>
using namespace std;
long job1(long arg)
{
    int tmp=0;
    for(int id=1;id<=arg;id++)
        tmp+=id;
    return tmp;
}
int main(int argc, char* argv[])
{
    std::future<long> f1=std::async(job1,100);
    cout<<"Thread 1 return : "<<f1.get()<<endl;
    cin.get();           //等待回车键,程序退出
    return 0;
}
```

例 16-22 的执行效果如图 16-23 所示。

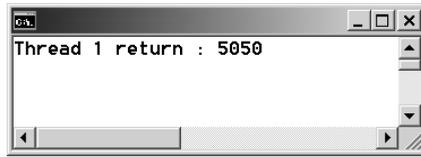


图 16-23 例 16-22 的执行效果

16.5.6 模板类 packaged_task

模板类 `packaged_task` 定义了包裹一个函数或可调用对象的一种类型，这使得函数或可调用对象的返回值被存储在 `future` 类型对象中。当 `packaged_task` 类型对象被激活之后，其存储任务会被激活，并且其结果或会被存储至共享状态中。共享该共享状态的任意 `future` 类型对象均可以访问该存储值。

该类的声明形式为：

```
template <class> class packaged_task; // undefined
template <class R, class... ArgTypes > class packaged_task <R(ArgTypes...) >
{
public:
    // construction and destruction
    packaged_task() noexcept;
    template <class F> explicit packaged_task(F&& f);
    template <class F, class Allocator > explicit packaged_task(allocator_arg_t, const Allocator&
a, F&& f);

    ~packaged_task();
    // no copy
    packaged_task(packaged_task&) = delete;
    packaged_task& operator = (packaged_task&) = delete;
    // move support
    packaged_task(packaged_task&& rhs) noexcept;
    packaged_task& operator = (packaged_task&& rhs) noexcept;
    void swap(packaged_task& other) noexcept;
    bool valid() const noexcept;
    // result retrieval
    future <R> get_future();
    // execution
    void operator () (ArgTypes... );
    void make_ready_at_thread_exit(ArgTypes... );
    void reset();
};
```

除了各种形式的构造函数之外，该类主要包含可用于返回 `future` 类型对象的 `get_future()`，`swap()` 函数和 `valid()`，`make_ready_at_thread_exit()` 函数和 `reset()` 以及一个有效的赋值符号。

而构造函数具有多种形式:

```

1) packaged_task() noexcept;
2) template <class F> explicit packaged_task(F&& f);
3) template <class F, class Allocator> explicit packaged_task(allocator_arg_t, const Allocator&
a, F&& f);
4) packaged_task(packaged_task&) = delete; //不支持 copy
5) packaged_task& operator=(packaged_task&) = delete; //不支持 copy
6) packaged_task(packaged_task&& rhs) noexcept; //支持 move

```

各函数的具体使用形式:

第一种构造函数用于构造一个默认空对象。该对象被初始化, 但是不包含共享状态, 不包含存储任务。

第二种构造函数 `template <class F> explicit packaged_task (F&& f)` 用于初始化一个共享状态, 并且被包装任务由参数 `f` 指定。

第三种构造函数是带自定义内存分配器的构造函数, 与默认构造函数类似, 但是使用自定义分配器来分配共享状态。

第六种构造函数用于获取 `rhs` 的共享状态, 并将 `rhs` 中的存储对象移动至新构造的对象中, 使 `rhs` 变为不再拥有共享状态。

`get_future()` 函数: 该函数用于返回该对象相关共享状态的 `future` 类型对象。

`valid()` 函数: 该函数用于判断当前 `packaged_task` 对象是否和共享状态相关联。

`make_ready_at_thread_exit()` 函数: 该函数会调用被包装的任务, 并向任务传递参数, 类似于 `std::packaged_task` 的 `operator()` 成员函数。但是与 `operator()` 函数不同的是, `make_ready_at_thread_exit` 并不会立即设置共享状态的标志为“ready”, 而是在线程退出时设置共享状态的标志。若与该 `packaged_task` 共享状态相关联的 `future` 对象在 `future::get()` 处等待, 则当前的 `future::get()` 调用会被阻塞, 直到线程退出。而一旦线程退出, `future::get()` 调用继续执行, 或者抛出异常。函数的参数就是对象的任务的参数。关于 `make_ready_at_thread_exit()` 函数的使用, 本书作者是使用 Visual Studio 2012 版本的 VC++ 开发环境。该开发环境提供的 `threadcall.cpp` 文件中包含如下代码:

```

static_Call_func_ret_CALL_FUNC_MODIFIER_Call_func(void* _Data)
{ // entry point for new thread
    _Call_func_ret_Res = 0;
#ifdef _HAS_EXCEPTIONS
    try { // don't let exceptions escape
        _Res = (_Call_func_ret)static_cast<_Pad*>(_Data) ->_Go();
    }
    catch (...)
    { // uncaught exception in thread
        int zero = 0;
        if (zero == 0)
            #if 1300 <= _MSC_VER
                terminate(); // to quiet diagnostics yanchy
            #else /* 1300 <= _MSC_VER */

```

```

        _XSTD terminate(); // to quiet diagnostics
    #endif /* 1300 <= _MSC_VER */
}
#else /* _HAS_EXCEPTIONS */
_Res = (_Call_func_ret)static_cast<_Pad* >(_Data) ->_Go();
#endif /* _HAS_EXCEPTIONS */
_Cnd_do_broadcast_at_thread_exit();
return (_Res);
}

```

在启动线程时，如果发生异常，在异常处理模块（catch 块）中，上述代码明显有错误：

```

int zero=0;
if(zero==0)
{
    ...
}

```

由于作者使用的是 V C++ 2012 版本，大于 1300，符合 #if 的条件，程序直接调用 terminate()，之后调用 abort()，程序会弹出异常对话框，终止运行。

该函数可能导致异常的情况如下：

异常类型	错误条件	描述
future_error	future_errc::no_state	该对象没有共享状态
future_error	future_errc::promise_already_satisfied	被存储的任务已经被调用

reset() 函数：该函数用于重置 packaged_task 的共享状态，但是保留之前被包装的任务。

swap() 函数：该函数用于交换 packaged_task 的共享状态。

operator() 函数：该函数用于调用该 packaged_task 对象所包装的对象。调用该函数一般会发生以下两种情况：

1) 若成功调用 packaged_task 所包装的对象，则返回值（如果被包装的对象有返回值的话）被保存在 packaged_task 的共享状态中。

2) 若调用 packaged_task 所包装的对象失败，并且抛出了异常，则异常也会被保存在 packaged_task 的共享状态中。

以上两种情况都使共享状态的标志变为 ready，因此其他等待该共享状态的线程可以获取共享状态的值或者异常并继续执行下去。

该模板类的具体使用方法参见例 16-23。例 16-24 用于讲述算法 swap() 在 packaged_task 类型对象交换信息中的应用。

例 16-23

```

#include <iostream>
#include <future>
#include <thread>
#include <utility>

```

```
using namespace std;
int job1(int arg)
{
    return arg* 5;
}
int main(int argc, char* argv[])
{
    std::packaged_task<int (int) > pt1;           // default - constructed
    std::packaged_task<int (int) >pt2 (job1);    //设置
    if (! pt2.valid ())
        return -1;
    pt1 = std::move (pt2);                       //赋值操作
    std::future<int > f1 = pt1.get_future ();    // get future
    pt1 (2);                                     // operator ()
    int value = f1.get ();                       // wait for the task to finish
                                                // and get result
    std::cout << " The stored value in future is " << value << endl;
    pt1.reset ();
    f1 = pt1.get_future ();                     // get future
    std::thread (std::move (pt1), 7).detach (); // 生成线程, 并调用任务
    std::cout << " The stored value in future is " << f1.get () << endl;
    return 0;
}
```

例 16-23 的执行效果如图 16-24 所示。

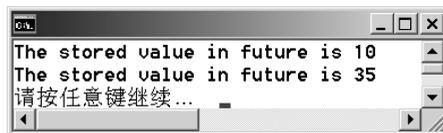


图 16-24 例 16-23 的执行效果

例 16-24

```
#include <iostream>
#include <future >
#include <thread >
#include <utility >
#include <algorithm >
using namespace std;
long job2(int arg1)
{
    return arg1;
}
int job1(int arg1)
{
    return arg1;
```

```
    }  
    int main(int argc, char* argv[])  
    {  
        std::packaged_task<long (int) > task1 (job1);  
        std::future<long> f1 = task1.get_future ();  
        task1 (1);  
        cout << " Thread 1 (shared_status): " << f1.get () << endl;  
  
        std::packaged_task<long (int) > task2 (job2);  
        std::future<long> f2 = task2.get_future ();  
        std::thread (std::move (task2), 5).detach ();  
        cout << " Thread 2 (shared_status): " << f2.get () << endl;  
        std::packaged_task<long (int) > task3 (job1);  
        std::packaged_task<long (int) > task4 (job2);  
        std::future<long> f3 = task3.get_future ();  
        std::future<long> f4 = task4.get_future ();  
        swap (task3, task4);  
        task3 (1);  
        std::thread (std::move (task4), 5).detach ();  
        cout << " Thread 3 (shared_status): " << f3.get () << endl;  
        cout << " Thread 4 (shared_status): " << f4.get () << endl;  
        cin.get ();  
        return 0;  
    }  
}
```

例 16-24 的执行效果如图 16-25 所示。

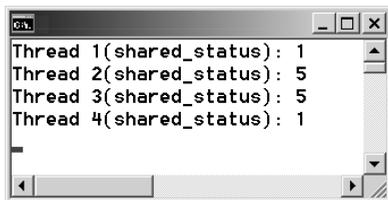


图 16-25 例 16-24 的执行效果

16.6 小结

本章首先介绍了线程控制的一些基础知识，在此基础上详细介绍了模板类 thread、模板类 mutex、模板类 lock、call_once 函数()、模板类 condition_variable、模板类 condition_variable_any、模板类 future_error、模板类 promise、模板类 future、模板类 shared_future、仿函数 async()以及模板类 packaged_task，并针对各类的不同特点，分别给出了详细的使用方法讲解。

本章主要讲述了和线程以及并发性相关的多个类的使用方法。多线程在实际工作中应用广泛，熟练掌握本章内容是非常有意义的。

第 17 章

正则表达式

1956 年，数学家 Stephen Kleene 发表了一篇题为“神经网络事件的表示法”的论文，在该论文中引入了“正则表达式”这一概念。他指出：正则表达式就是“正则集的代数”的表达式。

本章将讲述 C++ 程序中使用的正则表达式和正则搜索。该部分内容既是 C++ STL 的一部分，也是重要组成部分。本章内容主要包括：一种最基本的正则表达式类模板及其字符属性特点；两个特殊的处理字符顺序的类模板；一种保证正则表达式匹配的类模板；通过正则算法允许字符排序的一系列算法；两个用于枚举正则表达式匹配的迭代器类型。

正则表达式的优点及其好处如下：

- 搜索并测试字符串的模式。
- 替换文本。
- 根据模式匹配从字符串中提取一个字符串。
- 匹配整个输入。

17.1 定义及要求

校核（对）元素：在当前场景情况下，对一个或者多个字符进行校核，就像校核单一字符一样。

有限状态机：一种未指明的数据结构，用于表达一个正则表达式，该结构允许有效地匹配获得正则表达式。

格式指示符：一个或多个字符的序列匹配，该序列会被正则表达式的一部分代替。

相匹配：零个或更多个字符组成的序列通过正则表达式，当序列中的字符和预先定义模式的字符序列一致时，即认为这两个字符序列是相匹配的。

基本等值类：一个或多个字符的集合，这些元素共享同一个基本的排序键。该排序键的重要性取决于字符模型，而不是其特点（或强调）、实例及本地详细的裁剪。

正则表达式：从字符串集合中选择确定的字符串。

子表达式：正则表达式的子集。该子集通常使用符号标明。

正则表达式作为一类模板或者数据结果，必然有其性能和要求。首先，类模板 `basic_regex` 需要相关类型的集合和函数集，以便于实现其语法语义。这些类型和函数用于提供成员和函数集合，对应该模板类的参数 `trait`。需要使用“用于字符容器的类 `charT` 和相关的正则表达式特点类”来定义类模板 `basic_regex`，其形式如下：

```
basic_regex < charT, Traits >
```

17.2 类模板 `basic_regex`

若要使用类模板 `regex`，则必须包含头文件 `<regex>`。而头文件 `<regex>` 中还包含了头文件 `<initializer_list>`。可以说，与正则表达式相关的模板类、函数模板以及其成员函数是极其丰富的。

17.2.1 类模板 `basic_regex` 的声明

类 `basic_regex` 的声明形式及相关的类如下：

```
namespace std{
    namespace regex_constants{
        enum error_type;
    }
}

class regex_error;
template <class charT> struct regex_traits;
template <class charT, class traits = regex_traits < charT >> class basic_regex;
typedef basic_regex < char > regex;
typedef basic_regex < wchar_t > wregex;
template <class charT, class traits >
    void swap (basic_regex < charT, traits > &e1, basic_regex < charT, traits > &e2);
template <class BidirectionalIterator > class sub_match;
typedef sub_match < const char* > csub_match;
typedef sub_match < const wchar_t* > wsub_match;
typedef sub_match < string::const_iterator > ssub_match;
typedef sub_match < wstring::const_iterator > wssub_match;
template <class BiTer > bool operator == (const sub_match < BiTer > & lhs, const sub_match < BiTer > & rhs);
template <class BiTer > bool operator != (const sub_match < BiTer > & lhs, const sub_match < BiTer > & rhs);
template <class BiTer > bool operator < (const sub_match < BiTer > & lhs, const sub_match < BiTer > & rhs);
template <class BiTer > bool operator <= (const sub_match < BiTer > & lhs, const sub_match < BiTer > & rhs);
template <class BiTer > bool operator > (const sub_match < BiTer > & lhs, const sub_match < BiTer > & rhs);
template <class BiTer > bool operator >= (const sub_match < BiTer > & lhs, const sub_match < BiTer > & rhs);
template <class BiTer, class ST, class SA >
    bool operator == (const basic_string < typename iterator_traits < BiTer > ::value_type, ST, SA > & lhs,
        const sub_match < BiTer > & rhs);
template <class BiTer, class ST, class SA >
    bool operator != (const basic_string < typename iterator_traits < BiTer > ::value_type, ST, SA > & lhs,
```

```

    const sub_match<BiTer>& rhs);
template <class BiTer, class ST, class SA >
    bool operator < (const basic_string<typename iterator_traits<BiTer>::value_type, ST, SA >
& lhs,
    const sub_match<BiTer>& rhs);
template <class BiTer, class ST, class SA >
    bool operator > (const basic_string<typename iterator_traits<BiTer>::value_type, ST, SA >
& lhs,
    const sub_match<BiTer>& rhs);
template <class BiTer, class ST, class SA >
    bool operator > = (const basic_string<typename iterator_traits<BiTer>::value_type, ST, SA >
& lhs,
    const sub_match<BiTer>& rhs);
template <class BiTer, class ST, class SA >
    bool operator < = (const basic_string<typename iterator_traits<BiTer>::value_type, ST, SA >
& lhs,
    const sub_match<BiTer>& rhs);
template <class BiTer, class ST, class SA >
    bool operator == (const sub_match<BiTer>&lhs,
    const basic_string<typename iterator_traits<BiTer>::value_type, ST, SA >& rhs,
template <class BiTer, class ST, class SA >
    bool operator != (const sub_match<BiTer>&lhs,
    const basic_string<typename iterator_traits<BiTer>::value_type, ST, SA >& rhs,
template <class BiTer, class ST, class SA >
    bool operator < (const sub_match<BiTer>&lhs,
    const basic_string<typename iterator_traits<BiTer>::value_type, ST, SA >& rhs,
template <class BiTer, class ST, class SA >
    bool operator > (const sub_match<BiTer>&lhs,
    const basic_string<typename iterator_traits<BiTer>::value_type, ST, SA >&rhs);
template <class BiTer, class ST, class SA >
    bool operator > = (const sub_match<BiTer>&lhs,
    const basic_string<typename iterator_traits<BiTer>::value_type, ST, SA >&rhs);
template <class BiTer, class ST, class SA >
    bool operator < = (const sub_match<BiTer>&lhs,
    const basic_string<typename iterator_traits<BiTer>::value_type, ST, SA >&rhs);
template <class BiTer >
    bool operator == ( typename iterator_traits<BiTer>::value_type const * lhs,
    const sub_match<BiTer>&rhs);
template <class BiTer >
    bool operator != ( typename iterator_traits<BiTer>::value_type const * lhs,
    const sub_match<BiTer>&rhs);
template <class BiTer >
    bool operator < ( typename iterator_traits<BiTer>::value_type const * lhs,
    const sub_match<BiTer>&rhs);
template <class BiTer >
    bool operator > ( typename iterator_traits<BiTer>::value_type const * lhs,

```

```
    const sub_match<BiTer>&rhs);
template<class BiTer>
    bool operator>= (typename iterator_traits<BiTer>::value_type const * lhs,
        const sub_match<BiTer>&rhs);
template<class BiTer>
    bool operator<= (typename iterator_traits<BiTer>::value_type const * lhs,
        const sub_match<BiTer>&rhs);
template<class BiTer>
    bool operator==(const sub_match<BiTer> &lhs,
        typename iterator_traits<BiTer>::value_type const * rhs);
template<class BiTer>
    bool operator!=(const sub_match<BiTer> &lhs,
        typename iterator_traits<BiTer>::value_type const * rhs);
template<class BiTer>
    bool operator<(const sub_match<BiTer> &lhs,
        typename iterator_traits<BiTer>::value_type const * rhs);
template<class BiTer>
    bool operator>(const sub_match<BiTer> &lhs,
        typename iterator_traits<BiTer>::value_type const * rhs);
template<class BiTer>
    bool operator>= (const sub_match<BiTer> &lhs,
        typename iterator_traits<BiTer>::value_type const * rhs);
template<class BiTer>
    bool operator<= (const sub_match<BiTer> &lhs,
        typename iterator_traits<BiTer>::value_type const * rhs);
template<class BiTer>
    bool operator==(typename iterator_traits<BiTer>::value_type const &lhs,
        const sub_match<BiTer> &rhs);
template<class BiTer>
    bool operator!=(typename iterator_traits<BiTer>::value_type const &lhs,
        const sub_match<BiTer> &rhs);
template<class BiTer>
    bool operator<(typename iterator_traits<BiTer>::value_type const &lhs,
        const sub_match<BiTer> &rhs);
template<class BiTer>
    bool operator>(typename iterator_traits<BiTer>::value_type const &lhs,
        const sub_match<BiTer> &rhs);
template<class BiTer>
    bool operator>= (typename iterator_traits<BiTer>::value_type const &lhs,
        const sub_match<BiTer> &rhs);
template<class BiTer>
    bool operator<= (typename iterator_traits<BiTer>::value_type const &lhs,
        const sub_match<BiTer> &rhs);
template<class BiTer>
    bool operator==(const sub_match<BiTer> &lhs,
        typename iterator_traits<BiTer>::value_type const &rhs);
```

```

template < class BiTer >
    bool operator! = ( const sub_match<BiTer> &lhs,
        typename iterator_traits<BiTer>::value_type const & rhs);
template < class BiTer >
    bool operator < ( const sub_match<BiTer> &lhs,
        typename iterator_traits<BiTer>::value_type const & rhs);
template < class BiTer >
    bool operator > ( const sub_match<BiTer> &lhs,
        typename iterator_traits<BiTer>::value_type const & rhs);
template < class BiTer >
    bool operator > = ( const sub_match<BiTer> &lhs,
        typename iterator_traits<BiTer>::value_type const & rhs);
template < class BiTer >
    bool operator < = ( const sub_match<BiTer> &lhs,
        typename iterator_traits<BiTer>::value_type const & rhs);
template < class charT , class ST, class BiTer > basic_ostream < charT , ST > &
    operator << (basic_ostream < charT, ST > &os , const sub_match<BiTer> &m);
template < class BidirectionalIterator, class Allocator = allocator < sub_match<BidirectionalIt-
erator >>>
    class match_results;
typedef match_results < const char* > cmatch;
typedef match_result < const wchar_t* > wcmatch;
typedef match_results < string::const_iterator > smatch;
typedef match_results < wstring::const_iterator > wsmatch;
template < class BidirectionalIterator, class Allocator >
    bool operator == (const match_results < BidirectionalIterator, Allocator > &m1,
        const match_results < BidirectionalIterator, Allocator > &m2)
template < class BidirectionalIterator, class Allocator >
    bool operator! = (const match_results < BidirectionalIterator, Allocator > &m1,
        const match_results < BidirectionalIterator, Allocator > &m2)
template < class BidirectionalIterator, class Allocator >
    void swap (const match_results < BidirectionalIterator, Allocator > &m1,
        const match_results < BidirectionalIterator, Allocator > &m2)
template < class BidirectionalIterator, class Allocator, class charT, class traits >
    bool regex_match (BidirectionalIterator first, BidirectionalIterator last,
        match_results < BidirectionalIterator, Allocator > &m,
        const basic_regex < charT, traits > & e, regex_constants::match_flag_type flags =
            regex_constants::match_default);
template < class BidirectionalIterator, class charT, class traits >
    bool regex_match (BidirectionalIterator first, BidirectionalIterator last,
        const basic_regex < charT, traits > & e, regex_constants::match_flag_type flags =
            regex_constants::match_default);
template < class charT, class Allocator, class traits >
    bool regex_match (const charT* str, match_results < BidirectionalIterator, Allocator > &m,
        const basic_regex < charT, traits > & e, regex_constants::match_flag_type flags =
            regex_constants::match_default);

```

```
template <class ST, class SA, class Allocator, class charT, class traits >
    bool regex_match(const basic_string< chart, ST, SA >&s,
        match_results< BidirectionalIterator, Allocator >& m,
        const basic_regex< chart, traits >& e,
        regex_constants::match_flag_type flags = regex_constants::match_default);
template < class charT, class traits >
    bool regex_match(const charT* str,,
        const basic_regex< chart, traits >&e,
        regex_constants::match_flag_type flags = regex_constants::match_default);
template <class ST, class SA, class charT, class traits >
    bool regex_match(const basic_string< charT, ST, SA >&s,
        const basic_regex< charT, traits >& e,
        regex_constants::match_flag_type flags = regex_constants::match_default);
template <class BidirectionalIterator, class Allocator, class charT, class traits >
    bool regex_search(BidirectionalIterator first, BidirectionalIterator last,
        match_results< BidirectionalIterator, allocator >&m,
        const basic_regex< charT, traits >&e,
        regex_consts::match_flag_type flags = regex_consts::match_default);
template <class BidirectionalIterator, class charT, class traits >
    bool regex_search(BidirectionalIterator first, BidirectionalIterator last,
        const basic_regex< charT, traits >&e,
        regex_consts::match_flag_type flags = regex_consts::match_default);
template < class charT, class Allocator, class traits >
    bool regex_search(const charT* str,
        match_results< const charT* , Allocator >&m,
        const basic_regex< charT, traits >&e,
        regex_consts::match_flag_type flags = regex_consts::match_default);
template < class charT, class traits >
    bool regex_search(const charT* str,
        const basic_regex< charT, traits >&e,
        regex_consts::match_flag_type flags = regex_consts::match_default);
template < class ST, class SA, class charT, class traits >
    bool regex_search(const basic_string< charT, ST, SA >&s,
        const basic_regex< charT, traits >&e,
        regex_consts::match_flag_type flags = regex_consts::match_default);
template < class ST, class SA, class Allocator, class charT, class traits >
    bool regex_search(const basic_string< charT, ST, SA >&s,
        match_results< typename basic_string< charT, ST, SA >::const_iterator, Allocator >&m,
        const basic_regex< charT, traits >&e,
        regex_consts::match_flag_type flags = regex_consts::match_default);
template < class OutputIterator, class BidirectionalIterator, class traits, class charT, class
ST, class SA >
    OutputIterator
    regex_replace(OutputIterator out,
        BidirectionalIterator first, BidirectionalIterator last,
        const basic_regex< charT, tytraits >&e,
```

```

        const basic_string< charT, ST, SA > &fmt,
        regex_constants::match_flag_type flags = regex_constants::match_default);
template <class OutputIterator, class BidirectionalIterator, class traits, class charT >
    OutputIterator
    regex_replace(OutputIterator out,
        BidirectionalIterator first, BidirectionalIterator last,
        const basic_regex< charT, traits > &e,
        const charT* fmt,
        regex_constants::match_flag_type flags = regex_constants::match_default);
template <class traits, class charT, class ST, class SA, class FSF, class FSA >
    basic_string< charT, ST, SA >
    regex_replace(const basic_string< charT, ST, SA > &s,
        const basic_regex< charT, traits > &e,
        const basic_string< charT, FST, FSA > &fmt,
        regex_constants::match_flag_type flags = regex_constants::match_default);
template <class traits, class charT, class ST, class SA >
    basic_string< charT, ST, SA >
    regex_replace(const basic_string< charT, ST, SA > &s,
        const basic_regex< charT, traits > &e,
        const charT* fmt,
        regex_constants::match_flag_type flags = regex_constants::match_default);
template <class traits, class charT, class ST, class SA >
    basic_string< charT >
    regex_replace(const charT* s,
        const basic_regex< charT, traits > &e,
        const basic_string< charT, ST, SA > &fmt,
        regex_constants::match_flag_type flags = regex_constants::match_default);
template <class traits, class charT >
    basic_string< charT >
    regex_replace(const charT* s,
        const basic_regex< charT, traits > &e,
        const charT* fmt,
        regex_constants::match_flag_type flags = regex_constants::match_default);
template <class BidirectionalIterator, class charT = typename iterator_traits< BidirectionalI-
erator >::value_type,
        class traits = regex_traits< charT >>
    class regex_iterator;
typedef regex_iterator< const char* > cregex_iterator;
typedef regex_iterator< const wchar* > wcregex_iterator;
typedef regex_iterator< string::const_iterator > sregex_iterator;
typedef regex_iterator< wstring::const_iterator > wsregex_iterator;
template <class BidirectionalIterator, class charT = typename iterator_traits< BidirectionalI-
erator >::value_type,
        class traits = regex_traits< charT >>
    class regex_token_iterator;
typedef regex_token_iterator< const char* > cregex_token_iterator;

```

```
typedef regex_token_iterator<const wchar_t* > wcregex_token_iterator;
typedef regex_token_iterator<string::const_iterator > sregex_token_iterator;
typedef regex_token_iterator<wstring::const_iterator > wstring_regex_token_iterator;
}
```

17.2.2 名称空间 std::regex_constants

通过正则表达式库，命名空间 `std::regex_constants` 包含了标识性常量。该命名空间提供了 3 种语法类型 (`syntax_option_type`, `match_flag_type`, 和 `error_type`) 以及这 3 种类型的几个常量数据。

1. `syntax_option_type`

`syntax_option_type` 是位掩码类型。该类型的声明形式如下：

```
namespace std {
    namespace regex_constants {
        typedef T1 syntax_option_type;
        static constexpr syntax_option_type icsave = unspecified;
        static constexpr syntax_option_type nosubs = unspecified;
        static constexpr syntax_option_type optimize = unspecified;
        static constexpr syntax_option_type collate = unspecified;
        static constexpr syntax_option_type ECMAScript = unspecified;
        static constexpr syntax_option_type basic = unspecified;
        static constexpr syntax_option_type extended = unspecified;
        static constexpr syntax_option_type awk = unspecified;
        static constexpr syntax_option_type grep = unspecified;
        static constexpr syntax_option_type egrep = unspecified;
    }
}
```

设置其元素的意义见表 17-1。该类型的任意有效值应该设置为几种类型的值（其值类型通常为 `ECMAScript`、`basic`、`extended`、`awk`、`grep` 及 `egrep`）。

表 17-1 `syntax_option_type` 的功能

元 素	作 用
<code>icsave</code>	不区分大小写
<code>nosubs</code>	当执行匹配时，所有标记的子表达式被认为没有标记。如果所提供匹配的对被存储在指定结构 (<code>std::regex_match</code>) 中时，标记数量是零
<code>optimize</code>	匹配速度更快。正则对象的构造对象降低速度，否则程序输出时会发生不可测的结果
<code>collate</code>	字符范围应该具有地区敏感性
<code>ECMAScript</code>	通过正则表达式引擎确定的语法应该是被 ECMAScript 使用的语法
<code>basic</code>	通过正则表达式引擎确定的语法应该是被 POSIX 中正则表达式所应用的
<code>extended</code>	通过正则表达式引擎确定的语法应该是被 POSIX 中扩展正则表达式标准所应用的
<code>awk</code>	通过正则表达式引擎确定的语法应该是被 POSIX 中通用工具 <code>awk</code> 所使用的
<code>grep</code>	通过正则表达式引擎确定的语法应该是被 POSIX 中通用工具 <code>grep</code> 所使用的
<code>egrep</code>	通过正则表达式引擎确定的语法应该是被 POSIX 中通用工具 <code>egrep</code> 所使用的

2. regex_constants::match_flag_type

match_flag_type 也是位掩码类型。该类型的声明形式如下:

```
namespace std {
    namespace regex_constants {
        typedef T2 match_flag_type;
        static constexpr match_flag_type match_default = 0;
        static constexpr match_flag_type match_not_bol = unspecified;
        static constexpr match_flag_type match_not_eol = unspecified;
        static constexpr match_flag_type match_not_bow = unspecified;
        static constexpr match_flag_type match_not_eow = unspecified;
        static constexpr match_flag_type match_any = unspecified;
        static constexpr match_flag_type match_not_null = unspecified;
        static constexpr match_flag_type match_continuous = unspecified;
        static constexpr match_flag_type match_prev_avail = unspecified;
        static constexpr match_flag_type format_default = 0;
        static constexpr match_flag_type format_sed = unspecified;
        static constexpr match_flag_type format_no_copy = unspecified;
        static constexpr match_flag_type format_first_only = unspecified;
    }
}
```

此类型匹配一个正则表达式是按既定的语法规则，而不是按字符序列形式，详见表 17-2。

表 17-2 regex_constants::match_flag_type 的功能

元素	作用
match_not_bol	序列 [first, last] 的最后一个字符即使不是开头第一个，也被认为是第一个字符。正则表达式中的字符 “^” 不能匹配 [first, last]
match_not_eol	给定序列 [first, last] 中的最后一个字符，即使该字符不是此行的最后一个字符。正则表达式中的字符 “\$” 不能匹配 [last, last]
match_not_bow	表达式 “\b” 不应该匹配子序列 [first, last]
match_not_eow	表达式 “\b” 不应该匹配子序列 [first, last]
match_any	如果多于一个匹配是可能的，那么任意匹配是可接受的
match_not_null	表达式不应该匹配空序列
match_continuous	表达式仅仅能匹配以 first 开头的子序列
match_prev_avail	-- first 是一个有效的迭代位置。当标识被设置之后，按正则表达式算法和迭代器，参数 match_not_bol 和 match_not_bow 应该被忽略
format_default	当正则表达式的匹配被新的字符串代替时，新的字符串应该使用 ECMAScript 的替换函数规则构造。另外，在执行搜索和替换操作时，所有非重载发生的正则表达式应该被定位并替换，且不匹配正则表达式的输入部分应该原封不地复制到输出字符串中
format_sed	当正则表达式的匹配被新字符串替换之后，新的字符串应该使用 sed 通用工具定义的规则构造
format_no_copy	当搜索和替换操作时，字符序列的被搜索部分如果不能和正则表达式匹配，那么就不能将该串复制到输出字符串中
format_first_only	在某搜索和替换操作执行过程中，仅仅第一个匹配的正则表达式被替换

3. error_type

该类型的声明形式如下:

```

namespace std {
    namespace regex_constants {
        typedef T3 error_type;
        static constexpr error_type error_collate = unspecified;
        static constexpr error_type error_ctype = unspecified;
        static constexpr error_type error_escape = unspecified;
        static constexpr error_type error_backref = unspecified;
        static constexpr error_type error_brack = unspecified;
        static constexpr error_type error_paren = unspecified;
        static constexpr error_type error_brace = unspecified;
        static constexpr error_type error_badbrace = unspecified;
        static constexpr error_type error_range = unspecified;
        static constexpr error_type error_space = unspecified;
        static constexpr error_type error_badrepeat = unspecified;
        static constexpr error_type error_complexity = unspecified;
        static constexpr error_type error_stack = unspecified;
    }
}

```

`type_error` 是一种枚举类型。其类型的值代表了错误条件见表 17-3。

表 17-3 `error_type`

值	Error 条件
<code>error_collate</code>	表达式包含一个无效的校对元素名称
<code>error_ctype</code>	表达式包含一个无效的字符类名称
<code>error_escape</code>	表达式包含一个无效的 <code>escape</code> 字符或一个尾部的 <code>escape</code> 字符
<code>error_backref</code>	表达式包含一个无效的引用
<code>error_brack</code>	表达式包含不能匹配的符号 “[” 和 “]”
<code>error_paren</code>	表达式包含不能匹配的符号 “(” 和 “)”
<code>error_brace</code>	表达式包含不能匹配的符号 “{” 和 “}”
<code>error_badbrace</code>	表达式包含一个大括号内 ({}) 无效的范围
<code>error_range</code>	表达式包含一个无效的符号范围，像大多数编码中的 [b - a]
<code>error_space</code>	没有足够的内存转变表达式成为有限状态机
<code>error_badrepeat</code>	特殊字符 (* ? +) 不能处理
<code>error_complexity</code>	正则表达式尝试匹配的复杂性超越了预集合级别
<code>error_stack</code>	没有足够的内存用于判断是否正则表达式能匹配特定的字符序列

17.2.3 类 `regex_error`

类 `regex_error` 是由 `runtime_error` 派生而来的。其声明形式为：

```

class regex_error: public std::runtime_error{
public:
    explicit regex_error(regex_constants::error_type ecode);
    regex_constants::error_type code() const;
}

```

类 `regex_error` 定义了正则表达式库抛出异常对象的类型。

其构造函数 `regex_error()` 的形式:

```
regex_error(regex_constants::error_type ecode);
```

功能: 构造一个类 `regex_error` 类型的对象。

其构造函数

```
regex_constants::error_type code() const;
```

功能: 此函数返回构造函数的参数值。

例 17-1

```
#include <iostream>
#include <regex>
using namespace std;
int main ()
{
    try {
        std::regex myregex ("* ");
    }
    catch (std::regex_error& e) {
        if (e.code() == std::regex_constants::error_badrepeat)
            cerr << "Exception: A valid regular expression. \n" << endl;
        else
            cerr << "Some other regex exception happened. \n" << endl;
    }
    return 0;
}
```

例 17-1 的执行效果如图 17-1 所示。



图 17-1 例 17-1 的执行效果

17.2.4 类模板 `regex_traits`

此类型的声明形式如下:

```
template <class charT> struct regex_traits
{
    public:
        typedef charT char_type;
        typedef std::basic_string<char_type> string_type;
        typedef std::locale locale_type;
        typedef bitmask_type char_class_type;
        regex_traits(); //构造函数
```

```

static std::size_t length(const char_type* p);
charT translate(charT c) const;

template <class ForwardIterator> string_type transform(ForwardIterator first, ForwardIterator
last) const;
template <class ForwardIterator> string_type transform_primary(ForwardIterator first, ForwardIter-
ator last) const;
template <class ForwardIterator> string_type lookup_collatename(ForwardIterator first, ForwardIte-
rator last) const;
template <class ForwardIterator> char_class_type lookup_classname(ForwardIterator first, Forward-
Iterator last, bool
    icode = false) const;
    bool isctype(charT c, char_class_type f) const;
    int value(charT ch, int radix) const;
    locale_type imbue(locale_type l);
    locale_type getloc() const;
};

```

类 `regex_traits < char >` 和 `regex_traits < wchar_t >` 应该是有效的，并应该满足正则表达式特征类的要求。

类 `char_class_type` 通常习惯于代表字符的分类，该类型数据（变量）可作为函数 `lookup_classname()` 的返回值。

- 成员函数 `length()` 用于返回字符串的长度。
- 成员函数 `translate(charT c)` 用于翻译字符串。
- 成员函数 `transform()` 用于将参数序列转换成字符串。
- 成员函数 `transform_primary()` 用于返回字符串。
- 成员函数 `lookup_classname()` 用于返回字符类（见表 17-4）。

表 17-4 字符类

字符类	std::ctype_classification	字符类	std::ctype_classification
"alnum"	std::ctype_base::alnum	"punct"	std::ctype_base::punct
"alpha"	std::ctype_base::alpha	"space"	std::ctype_base::space
"blank"	std::ctype_base::blank	"upper"	std::ctype_base::upper
"cntrl"	std::ctype_base::cntrl	"xdigit"	std::ctype_base::xdigit
"digit"	std::ctype_base::digit	"d"	std::ctype_base::digit
"graph"	std::ctype_base::graph	"s"	std::ctype_base::space
"lower"	std::ctype_base::lower	"w"	std::ctype_base::alnum 选择性地添加
"print"	std::ctype_base::print		

注：字符串 "w" 返回的分类可能是完全一样的，在这种情况下，使用 "alnum" 添加 `isctype()'` 会更明确。

- 成员函数 `lookup_collatename()` 用于返回等价的校对名称。
- 成员函数 `isctype()` 用于检查字符串是否是一个类型。
- 成员函数 `value()` 用于返回某进制的数字字符的值。
- 成员函数 `imbue()` 用于转换场所和地点。
- 成员函数 `getloc()` 用于获取场景。

例 17-2

```

#include <iostream>
#include <regex>
#include <locale>
#include <string>
#include <iomanip>
#include <string>
using namespace std;

//for lookup_collatename
struct noisy_traits : std::regex_traits<char> {

    template< class Iter >
    string_type lookup_collatename( Iter first, Iter last ) const {
        string_type result = regex_traits::lookup_collatename( first, last );
        std::cout << "regex_traits<>::lookup_collatename(\""
            << string_type( first, last )
            << "\") returns \"" << result << "\"\n";
        return result;
    }
};

int main( int argc, char* argv[] )
{
    locale loc( "English_US.1251" ); //the "C" locale
    //std::locale::global( std::locale( "en_US.UTF-8" ) ); //设置新的环境,用 zh_CN.UTF-8 试试,
    //English_US.1251

    locale old = locale::global( loc );
    regex_traits<char> rtl;
    cout << "The regex locale is " << rtl.getloc().name() << endl;
    cout << "The regex length: " << rtl.length( "ABCDEFG12345" ) << endl;
    cout << "string length : " << regex_traits<char>::length( "Кошка" ) << endl;
    cout << "string length : " << regex_traits<wchar_t>::length( L"Кошка" ) << endl;
    cout << "string length : " << regex_traits<char>::length( "北京市是中国的首都." ) << endl;
    cout << "string length : " << regex_traits<wchar_t>::length( L"北京市是中国的首都." ) << endl;
    cout << "string length : " << regex_traits<char>::length( "北京市是中国的首都" ) << endl;
    cout << "string length : " << regex_traits<wchar_t>::length( L"北京市是中国的首都" ) << endl;
    regex_traits<char>::char_type ch = rtl.translate( 'B' );
    cout << ch << endl;
    ch = rtl.translate_nocase( 'B' );
    cout << "B ---- > (tolower): " << ch << endl;
    string st( "ANBFGJ" );
    regex_traits<char>::string_type st2;
    st2 = rtl.transform( &* st.begin(), &* ( --st.end() ) );
    st = rtl.transform_primary( &* st.begin(), &* ( --st.end() ) );
    cout << st2 << endl;
    cout << st << endl;
}

```

```
std::string sl("z|a");
std::regex rg("abcd", std::regex::ECMAScript);
std::cout << std::boolalpha << std::regex_match(sl, rg) << std::endl;
string sy("alnum");
auto a = rtl.lookup_classname(sy.begin(), sy.end());
cout << "classname: " << a << endl;
string tmps = rtl.lookup_collatename(sy.begin(), sy.end());
cout << "collatename: " << tmps << endl;
cout << "hex C==" << rtl.value('C', 16) << endl;
string sy2("space");
auto b = rtl.lookup_classname(sy2.begin(), sy2.end());
cout << "class name: " << b << endl;
cout << rtl.isctype('G', a) << endl;
cout << rtl.isctype('_', a) << endl;
cout << rtl.isctype(' ', a) << endl;
cout << rtl.isctype('G', b) << endl;
cout << rtl.isctype('_', b) << endl;
cout << rtl.isctype(' ', b) << endl;
auto myloc = rtl.getloc();
cout << myloc.name() << endl;
regex_traits<char>::locale_type locn(old);
rtl.imbue(locn);
myloc = rtl.getloc();
cout << '"' << myloc.name() << '"' << endl;
return 0;
}
```

例 17-2 的执行效果如图 17-2 所示。



```
English_United States.1251
The regex length: 12
string length :10
string length :5
string length :19
string length :10
string length :18
string length :9
B
B---->(tolower): b
J#J%01111100a
J#J%0000a
false
classname: 263
collatename: alnum
hex C=12
class name: 72
true
false
false
false
false
true
English_United States.1251
"C"
```

图 17-2 例 17-2 的执行效果

17.2.5 类 basic_regex 的使用

对于类似字符的类型 charT，尤其是在类模板 basic_regex 中，用于表示正则表达式中的字符序列时，都是采用 charT 类型字符。charT 代表一个像字符的类型。存储正则表达式的空间的分配和释放是必需的，并且可作为类 basic_regex 的成员函数。

类 basic_regex 的对象可以转换 charT 类型对象的顺序。正则表达式的异常错误是通过类 regex_error 来处理的。类模板 basic_regex 的声明形式如下：

```
namespace std {
template <class charT, class traits = regex_traits<charT> > class basic_regex
{
public:
    typedef charT value_type;
    typedef traits traits_type;
    typedef typename traits::string_type string_type;
    typedef regex_constants::syntax_option_type flag_type;
    typedef typename traits::locale_type locale_type;
    static constexpr regex_constants::syntax_option_type  icase = regex_constants::icase;
    static constexpr regex_constants::syntax_option_type  nosubs = regex_constants::
nosubs;
    static constexpr regex_constants::syntax_option_type  optimize = regex_constants::op-
optimize;
    static constexpr regex_constants::syntax_option_type  collate = regex_constants::col-
late;
    static constexpr regex_constants::syntax_option_type  ECMAScript = regex_constants::ECMAS-
cript;
    static constexpr regex_constants::syntax_option_type  basic = regex_constants::basic;
    static constexpr regex_constants::syntax_option_type  extended = regex_constants::ex-
tended;
    static constexpr regex_constants::syntax_option_type  awk = regex_constants::awk;
    static constexpr regex_constants::syntax_option_type  grep = regex_constants::grep;
    static constexpr regex_constants::syntax_option_type  egrep = regex_constants::egrep;
    basic_regex();
    explicit basic_regex(const charT* p, flag_type f = regex_constants::ECMAScript);
    basic_regex(const charT* p, size_t len, flag_type f = regex_constants::ECMAScript);
    basic_regex(const basic_regex&);
    basic_regex(basic_regex&&) noexcept;
template <class ST, class SA > explicit basic_regex(const basic_string<charT, ST, SA>& p,
    flag_type f = regex_constants::ECMAScript);
template <class ForwardIterator > basic_regex(ForwardIterator first, ForwardIterator last,
    flag_type f = regex_constants::ECMAScript);
    basic_regex(initializer_list<charT>, flag_type = regex_constants::ECMAScript);
```

```

~basic_regex();
basic_regex& operator = (const basic_regex&);
basic_regex& operator = (basic_regex&&) noexcept;
basic_regex& operator = (const charT* ptr);
basic_regex& operator = (initializer_list<charT> il);
template <class ST, class SA >basic_regex& operator = (const basic_string<charT, ST, SA
>& p);

basic_regex& assign(const basic_regex& that);
basic_regex& assign(basic_regex&& that) noexcept;
basic_regex& assign(const charT* ptr,
flag_type f = regex_constants::ECMAScript);
basic_regex& assign(const charT* p, size_t len, flag_type f);
template <class string_traits, class A >basic_regex& assign(const basic_string<charT,
string_traits,
A>& s, flag_type f = regex_constants::ECMAScript);
template <class InputIterator >basic_regex& assign(InputIterator first, InputIterator
last,
flag_type f = regex_constants::ECMAScript);
basic_regex& assign(initializer_list<charT>,
flag_type = regex_constants::ECMAScript);
unsigned mark_count() const;
flag_type flags() const;
locale_type imbue(locale_type loc);
locale_type getloc() const;
void swap(basic_regex&);
};
}

```

上述类声明包含 5 种数据类型、构造函数和析构函数、正则表达式赋值符号、7 个成员函数以及 10 个常量标识。

其中，5 种数据类型包括 `value_type`、`traits_type`、`string_type`、`flag_type` 和 `locale_type`。它们均是利用 `typedef` 重新定义而来的，分别代表其原来类型的新名称。

构造函数具有 8 种形式，参见例 17-3。

例 17-3

```

#include <iostream>
#include <string>
#include <regex>
using namespace std;
int main ()
{
    string pattern = "^.* $";
    regex first; // 默认方式
    regex second = first; // 复制
    regex third(pattern); // 初始化 string 对象
    regex fourth("<[^>]>"); // 初始化 string 表达式
}

```

```

    regex fifth(pattern.begin(),pattern.end()); // 以范围的形式进行初始化
    using namespace std::regex_constants;
    regex seventh("[0-9A-Z] +", regex_constants::ECMAScript);
    regex eighth("[0-9A-Z] +", ECMAScript); // 与上一行语句作用相同
    regex ninth("\\bd\\w +", ECMAScript | icase); // 多个标识
    string subject = "Duddy the duck \n";
    string replacement = "yup the duck 123456789 \n";
    cout << std::regex_replace(subject, ninth, replacement) << endl;
    return 0;
}

```

例 17-3 的执行效果如图 17-3 所示。

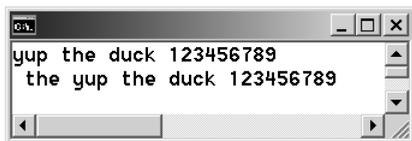


图 17-3 例 17-3 的执行效果

10 个常量标识见表 17-5。

表 17-5 常量标识表

flag*	功能	提示
icase	不计大小写	正则表达式匹配时不考虑大小写
nosubs	没有子表达式	匹配结果结构中不包括子表达式
optimize	优化匹配	匹配效率优越
collate	本地化灵敏度	字符范围受本地化约束
ECMAScript	ECMAScript 语法	正则表达式遵从这些语法
basic	基本 POSIX 语法	
extended	扩展 POSIX 语法	
awk	awk POSIX 语法	
grep	grep POSIX 语法	
egrep	egrep POSIX 语法	

其余的成员函数还包括 `assign()`、`mark_count()`、`flags()`、`imbue()`、`getloc()` 和 `swap()`。

`assign()` 函数主要用于给正则表达式类型对象赋值。

`markcount()` 函数主要用于确定正则表达式中子表达式的数量。

`flags()` 函数主要用于返回正则表达式中的句法标识。

`imbue()` 函数主要用于将 `locale` 类型对象和正则表达式相关联。

`getloc()` 函数主要用于获取本地化对象。

swap() 函数用于交换两个正则表达式之间的值。
各函数的使用方法参见例 17-4。

例 17-4

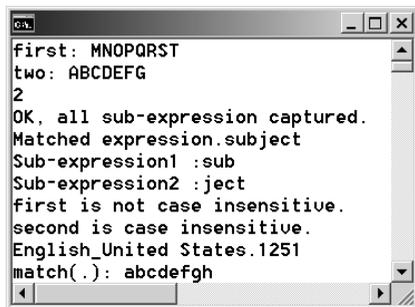
```
#include <iostream>
#include <string>
#include <regex>
#include <locale>
using namespace std;
int main ()
{
    std::string pattern ("ABCDEFGH");
    std::regex first;
    std::regex second ("123456");
    first.assign (second);
    second.assign ("MNOPQRST", std::regex::ECMAScript | std::regex::icase);
    std::string replacement = "seven";
    std::cout << "first: " << regex_replace ("MNOPQRST", first, replacement);
    std::cout << std::endl;
    std::cout << "two: " << regex_replace (pattern.c_str(), second, replacement);
    std::cout << std::endl;
    regex myrg ("(sub) ([a-z]* ).* ");
    cout << myrg.mark_count() << endl;
    cmatch m;
    regex_match ("subject", m, myrg);
    if (m.size() == myrg.mark_count() + 1)
    {
        cout << "OK, all sub-expression captured." << endl;
        cout << "Matched expression. " << m[0] << endl;
        for (unsigned i = 1; i < m.size(); ++i)
        {
            cout << "Sub-expression " << i << " : " << m[i] << endl;
        }
    }
    std::cout << "first ";
    std::cout << ( ( first.flags() & regex::icase ) == regex::icase ? "is":"is not");
    std::cout << " case insensitive. \n";
    std::cout << "second ";
    std::cout << ( ( second.flags() & regex::icase ) == regex::icase ? "is":"is not");
    std::cout << " case insensitive. \n";
    locale loc ("English_US.1251"); //the "C" locale
    myrg.imbue (loc);
    cout << myrg.getloc().name() << endl;
    cmatch mt;
    regex rx ("abcdefgh");
    regex rx2;
    swap (rx, rx2);
```

```

bool t = regex_search("abcdefghijk", mt, rx2);
cout << "match(.): " << mt.str() << endl;
return 0;
}

```

例 17-4 的执行效果如图 17-4 所示。



```

first: MNOPQRST
two: ABCDEFG
2
OK, all sub-expression captured.
Matched expression.subject
Sub-expression1 :sub
Sub-expression2 :ject
first is not case insensitive.
second is case insensitive.
English_United States.1251
match(.): abcdefgh

```

图 17-4 例 17-4 的执行效果

17.3 类模板 sub_match 和 match_results

类模板 `sub_match` 用于表示与具有特定标识的子表达式相匹配的字符序列。模板类 `match_result` 用于表示一个字符序列的集合，该集合表示一个正则表达式匹配的结果。集合的存储空间分配和释放必须通过模板类 `match_results` 的成员函数来实现。类模板 `match_results` 应该满足分配件式容器和序列式容器的要求。此外，仅有的序列式容器的预定义操作是可以支持的。默认构造的 `match_result` 类型对象没有全部建立结果状态。一个匹配的结果是作为完全的正则表达式匹配结果。当 `match_result` 类对象没有准备完毕时，调用大部分成员函数的结果是不可预期的。

存储类 `sub_match` 对象时，其下标为零的内容代表第 0 个子表达式。通常，`sub_match` 类的成员 `matched` 的值是真。类 `sub_match` 对象存储在下标为 `n` 的子正则表达式中。若子表达式 `n` 是一个正则表达式中的一部分，则 `sub_match` 类的成员 `matched` 的值是 `true`，并且成员 `first` 和 `second` 表明了字符序列的范围；反之，成员 `matched` 的值是 `false`，并且成员 `first` 和 `second` 指向被搜索到的序列末尾。

17.3.1 类模板 sub_match

此类的声明形式如下：

```

namespace std {
template <class BidirectionalIterator > class sub_match : public std::pair <BidirectionalIterator, BidirectionalIterator >
{
public:
typedef typename iterator_traits <BidirectionalIterator >::value_type value_type;

```

```
typedef typename iterator_traits <BidirectionalIterator>::difference_type difference
_type;

typedef BidirectionalIterator    iterator;
typedef basic_string <value_type>    string_type;
bool matched;
constexpr sub_match();
difference_type length() const;
operator string_type() const;
string_type str() const;
int compare(const sub_match& s) const;
int compare(const string_type& s) const;
int compare(const value_type* s) const;
};
}
```

由上述内容可知，该类模板包含 3 个类型：一个迭代器、一个布尔逻辑量以及 5 个成员函数（包括一个构造函数）。

`length()` 函数可返回子匹配序列的长度。若没有可匹配的序列，则返回 0。

`string_type()` 函数返回 `str()`。

`str()` 函数将 `sub_match` 类型对象转化为字符串。

`compare()` 函数用于实现对比。

类模板的具体使用方法参见例 17-5。

例 17-5

```
#include <regex>
#include <iostream>

int main()
{
    std::regex rx("c(a* )|(b)");
    std::cmatch mr;

    std::regex_search("xcaaay", mr, rx);           //判断是否有子字符串和正则表达式相匹配

    std::csub_match sub = mr[1];
    std::cout << "matched == " << std::boolalpha
              << sub.matched << std::endl;
    std::cout << "length == " << sub.length() << std::endl;

    std::csub_match::difference_type dif = std::distance(sub.first, sub.second);
    std::cout << "difference == " << dif << std::endl;

    std::csub_match::iterator first = sub.first;
    std::csub_match::iterator last = sub.second;
    std::cout << "range == " << std::string(first, last)
```

```

        << std::endl;
        std::cout << "string == " << sub << std::endl;
        std::csub_match::value_type * ptr = "aab";
        std::cout << "compare(\"aab\") == "
            << sub.compare(ptr) << std::endl;
        std::cout << "compare(string) == "
            << sub.compare(std::string("AAA")) << std::endl;
        std::cout << "compare(sub) == "
            << sub.compare(sub) << std::endl;

        return (0);
    }

```

例 17-5 的执行效果如下：

```

size: 3
matched == true
length == 4
difference == 4
range == caaa
string == caaa
compare("aab") == 1
compare(string) == 1
compare(sub) == 0
matched == true
length == 3
difference == 3
range == aaa
string == aaa
compare("aab") == -1
compare(string) == 1
compare(sub) == 0
matched == false
length == 0
difference == 0
range ==
string ==
compare("aab") == -1
compare(string) == -1
compare(sub) == 0

```

17.3.2 类模板 match_results

类模板 `match_results` 描述了一个对象。该对象控制一个不可调整的元素序列（该元素序列是通过正则表达式搜索产生的）。每个元素指向其子序列，这些子序列和相关规则相匹配。该类中所含内容是非常丰富的。其声明形式如下：

```
namespace std {
    template <class BidirectionalIterator, class Allocator = allocator<sub_match<BidirectionalI-
erator > >
    class match_results
    {
    public:
        typedef sub_match<BidirectionalIterator > value_type;
        typedef const value_type& const_reference;
        typedef const_reference reference;
        typedef implementation-defined const_iterator;
        typedef const_iterator iterator;
        typedef typename iterator_traits<BidirectionalIterator >::difference_type difference
_type;

        typedef typename allocator_traits<Allocator >::size_type size_type;
        typedef Allocator allocator_type;
        typedef typename iterator_traits<BidirectionalIterator >::value_type char_type;
        typedef basic_string<char_type > string_type;
        explicit match_results(const Allocator& a = Allocator());
        match_results(const match_results& m);
        match_results(match_results&& m) noexcept;
        match_results& operator = (const match_results& m);
        match_results& operator = (match_results&& m);
        ~match_results();
        bool ready() const;
        size_type size() const;
        size_type max_size() const;
        bool empty() const;
        difference_type length(size_type sub = 0) const;
        difference_type position(size_type sub = 0) const;
        string_type str(size_type sub = 0) const;
        const_reference operator[] (size_type n) const;
        const_reference prefix() const;
        const_reference suffix() const;
        const_iterator begin() const;
        const_iterator end() const;
        const_iterator cbegin() const;
        const_iterator cend() const;
    template <class OutputIter > OutputIter format (OutputIter out, const char_type* fmt_first, const
char_type* fmt_last,
        regex_constants::match_flag_type flags = regex_constants::format_default) const;
    template <class OutputIter, class ST, class SA > OutputIter format (OutputIter out, const basic
string<char_type, ST,
        SA>& fmt, regex_constants::match_flag_type flags = regex_constants::format_default) const;
    template <class ST, class SA > basic_string<char_type, ST, SA > format (const basic_string<char
type, ST, SA >& fmt,
        regex_constants::match_flag_type flags = regex_constants::format_default) const;
```

```
string_type  format (const char_type*  fmt, regex_constants::match_flag_type flags = regex_constants::
format_default ) const;
    allocator_type get_allocator () const;
    void  swap (match_results& that);
};
}
```

从上述内容可知，该类包含了大量的成员函数。下面分别介绍这些成员函数。

```
typedef sub_match<BidirectionalIterator>  value_type;
typedef const value_type&  const_reference;
typedef const_reference  reference;
typedef implementation-defined  const_iterator;
typedef const_iterator  iterator;
typedef typename iterator_traits<BidirectionalIterator>::difference_type difference_type;
typedef typename allocator_traits<Allocator>::size_type  size_type;
typedef Allocator allocator_type;
typedef typename iterator_traits<BidirectionalIterator>::value_type char_type;
typedef basic_string<char_type>  string_type;
```

上述代码分别定义了多个类型：value_type、const_reference、reference、const_iterator、iterator、difference_type、size_type、allocate_type、char_type 以及 string_type。

该类的构造函数包含了 3 种形式：

```
explicit match_results (const Allocator& a = Allocator ());
match_results (const match_results& m);
match_results (match_results&& m) noexcept;
```

第一种形式是默认的构造函数；第二种形式是利用 match_results 类型的常量实现构造该类对象；第三种形式是利用 match_results 类型的变量实现构造该类对象。

两个 operator = () 函数用于在同类型变量之间进行赋值。其声明形式为：

```
match_results& operator = (const match_results& m);
match_results& operator = (match_results&& m);
```

当类 match_results 的对象已经完全建立时，函数 ready() 返回 true；否则，返回 false。

size() 函数返回 match_results 类型对象中的子表达式个数。

empty() 函数用于判断该对象中是否为空。

length() 函数用于返回 match_result 类型对象中第 n 个匹配的子表达式的长度。

position() 函数用于返回各子表达式第一个字符在原来序列中的位置。

str() 函数用于将该类型对象转化为字符串。

prefix() 函数和 suffix() 分别返回原有序列中，目标序列之前或之后的部分。

begin() 函数和 end() 分别返回 match_results 类型对象中第一个子表达式的位置或最后位置。

cbegin() 函数和 cend() 的前缀 “c” 代表常量迭代式 const_iterator。

swap() 函数用于交换两个 match_result 类型对象的内容和大小。

format() 函数主要适用于格式化字符串。此函数的使用比较复杂，涉及的参数也比较复杂。

```
template <class OutputIter>OutputIter format (OutputIter out , const char_type* fmt_first , const
char_type* fmt_last ,
    regex_constants::match_flag_type flags = regex_constants::format_default) const;
template <class OutputIter, class ST, class SA > OutputIter format (OutputIter out, const basic
string<char_type, ST,
    SA>& fmt , regex_constants::match_flag_type flags = regex_constants::format_default) const;
template <class ST, class SA > basic_string<char_type, ST, SA > format (const basic_string<char
type, ST, SA>& fmt ,
    regex_constants::match_flag_type flags = regex_constants::format_default) const;
string_type format (const char_type* fmt , regex_constants::match_flag_type flags = regex_con
stants::
    format_default ) const;
```

其中参数 flags 是最复杂的，其参数值也最复杂，见表 17-1、表 17-2 和表 17-3。另外，格式字符串中替代符的功能详见表 17-6。

表 17-6 格式替代符的功能

字 符	功 能	字 符	功 能
\$ n	n 至多是两位数，代表第 n 个子表达式	\$ '	目标表达式之后的部分
\$ &	所有匹配内容的副本	\$ \$	单个符号 “\$”
\$ '	目标子表达式之前的部分		

该类的使用方法参见例 17-6。

注：类 smatch 是类模板 match_result 的实例化类。其定义形式为：

```
typedef match_results<string::const_iterator> smatch;
```

此外，类 cmatch 也是类模板 match_result 的实例化类。

例 17-6

```
#include <iostream>
#include <regex>
using namespace std;
int main(int argc, char* argv[])
{
    cmatch m1;
    if(m1.empty())
        cout << "m1 is empty. " << endl;
    else
        cout << "m1 is full. " << endl;
    string st1 = "subject";
```

```

regex rgx_rule("(sub)(.*)");
regex_match(stl.data(), m1, rgx_rule);
if(m1.ready())
    cout << "match ready.. " << endl;
if(m1.empty())
    cout << "m1 is empty. " << endl;
else
    cout << "m1 is full. " << endl;
    cout << "m1 size: " << m1.size() << endl;
for (unsigned i=0; i<m1.size(); ++i)
    cout << "match (" << i << ") " << " : " << m1[i] << endl;
for (unsigned i=0; i<m1.size(); ++i)
    cout << "match (" << i << ") length is " << " : " << m1[i].length() << endl;
for (unsigned i=0; i<m1.size(); ++i)
    cout << "match (" << i << ") position is " << " : " << m1.position(i) << endl;
for (unsigned i=0; i<m1.size(); ++i)
    cout << "match (" << i << ") content is " << " : " << m1[i].str() << endl;
cout << "match content is \" ";
    for (cmatch::iterator it = m1.begin(); it! =m1.end(); ++it)
    {
    cout << * it << ", ";
    }
cout << "\" " << endl;

string s2 ("there is a needle in this haystack");
smatch m2;
regex e2 ("needle");

cout << "searching for needle in [" << s2 << "]" \n";
regex_search ( s2, m2, e2 );

if (m2.ready())
{
    cout << m2[0] << " found! \n";
    cout << "prefix: [" << m2.prefix() << "]" << endl;
    cout << "suffix: [" << m2.suffix() << "]" << endl;
}
cout << m2.format ("the prefix expression  matched [ $' ]. ") << endl;
cout << m2.format ("the suffix expression  matched [ $' ]. ") << endl;
smatch m3;
m2.swap (m3);
for (unsigned i=0; i<m3.size(); ++i)
    cout << "match (" << i << ") content is " << " : " << m3[i].str() << endl;
cout << "\" " << endl;
cmatch m4;
m1.swap (m4);

```

```
for (unsigned i=0; i<m4.size();++i)
    cout << "match (" << i << ") content is "<<" : " << m4[i].str() << endl;
cout << "\"\" << endl;
cout << m4.format ("the expression 0 matched [ $ 0]. \n") << endl;
cout << m4.format ("the expression 1 matched [ $ 1]. \n") << endl;
cout << m4.format ("the expression 2 matched [4 $ 2]. \n") << endl;
cout << m4.format ("the entire expression  matched [ $ &]. \n") << endl;

return 0;
}
```

例 17-6 的执行结果如下:

```
m1 is empty.
match ready...
m1 is full.
m1 size: 3
match (0) : subject
match (1) : sub
match (2) : ject
match (0) length is : 7
match (1) length is : 3
match (2) length is : 4
match (0) position is : 0
match (1) position is : 0
match (2) position is : 3
match (0) content is : subject
match (1) content is : sub
match (2) content is : ject
match content is " subject, sub, ject, "
searching for needle in [there is a needle in this haystack]
needle found!
prefix: [there is a ]
suffix: [ in this haystack]
the prefix expression  matched [there is a ].
the suffix expression  matched [ in this haystack].
match (0) content is : needle
"
match (0) content is : subject
match (1) content is : sub
match (2) content is : ject
"
the expression 0 matched [subject].
the expression 1 matched [sub].
the expression 2 matched [ject].
the entire expression  matched [subject].
```

17.4 正则表达式相关的 3 种算法

C++ STL 还提供了和正则表达式相关的 3 种算法。这 3 种算法分别是匹配 (`regex_match`)、搜索 (`regex_search`) 和替代 (`regex_replace`)。本节的这些算法在使用过程中可能会由于各种原因抛出 `regex_error` 类型的异常。若异常类对象 `e` 被抛出, 则 `e.code()` 会返回两种可能的值: `error_complexity` 或 `regex_constants::error_stack`。

17.4.1 正则匹配算法 `regex_match`

正则匹配算法 `regex_match()` 的声明形式如下:

```
template <class BidirectionalIterator, class Allocator, class charT, class traits>bool regex_match
(BidirectionalIterator
    first, BidirectionalIterator last, match_results<BidirectionalIterator, Allocator> &m, const
basic_regex<charT, traits>
    & e, regex_constants::match_flag_type flags = regex_constants::match_default);

template <class BidirectionalIterator, class charT, class traits>bool regex_match(BidirectionalI-
terator first,
    BidirectionalIterator last, const basic_regex<charT, traits> &e, regex_constants::match_flag_
type flags
    = regex_constants::match_default);

template <class charT, class Allocator, class traits>bool regex_match(const charT* str, match_re-
sults<const charT*, Allo-
cator> &m, const basic_regex<charT, traits> &e, regex_constants::match_flag_type
    flags = regex_constants::match_default);

template <class ST, class SA, class Allocator, class charT, class traits>bool regex_match(const bas-
ic_string<charT, ST,
SA> &s, match_results<typename basic_string<charT, ST, SA>::const_iterator, Allocator> &m,
const basic_regex<charT,
traits> &e, regex_constants::match_flag_type flags = regex_constants::match_default);

template <class charT, class traits>bool regex_match(const charT* str, const basic_regex<charT,
traits> &e,
    regex_constants::match_flag_type flags = regex_constants::match_default);

template <class ST, class SA, class charT, class traits>bool regex_match(const basic_string<charT,
ST, SA> &s, const
    basic_regex<charT, traits> &e, regex_constants::match_flag_type flags = regex_constants::
match_default);
```

此算法用于判断给定表达式和给定字符序列之间是否存在一个匹配。若该匹配存在, 则函数返回 `true`; 否则, 返回 `false`。

上述第一种、第三种和第四种形式均包含了 `match_result` 类型参数 `m`, 虽略有区别, 但

大致相似。除了 `m.size()` 返回 0 或者 `m.empty()` 返回 true 时，参数 `m` 的作用是不确定的。参数 `m` 的作用见表 17-7。

表 17-7 regex_match 算法中 match_result 类型参数的作用

参 数	作 用
<code>m.size()</code>	返回 <code>e.mark_count()</code>
<code>m.empty()</code>	false
<code>m.prefix().first</code>	first
<code>m.prefix().second</code>	first
<code>m.prefix().matched</code>	false
<code>m.suffix().first</code>	last
<code>m.suffix().second</code>	last
<code>m.suffix().matched</code>	false
<code>m[0].first</code>	first
<code>m[0].second</code>	last
<code>m[0].matched</code>	若搜寻到完整的匹配，其值为 true
<code>m[n].first</code>	对于所有整数 $n < m.size()$ ，序列开始匹配子表达式 n 。另外，如果子表达式 n 不参与匹配，即是 last
<code>m[n].second</code>	对于所有整数 $n < m.size()$ ，第 n 个相匹配的序列结尾。如果子表达式 n 不参与匹配，即是 last
<code>m[n].matched</code>	对于所有整数 $n < m.size()$ ，如果子表达式 n 参与匹配，其值为 true

参数 `flags` 用于控制表达式是如何匹配字符序列的。其具体使用方法见表 17-8。

表 17-8 参数 flags 的作用

标 识	作 用	提 示
<code>match_default</code>	默认作用	默认匹配行为，常量拥有值 <code>zero**</code>
<code>match_not_bol</code>	非行的起始位置	第一个字符不认为是行的起始位置（“^”是不匹配的）
<code>match_not_eol</code>	非行的终止位置	最后一个字符不认为是行的末尾（“\$”是不匹配的）
<code>match_not_bow</code>	非单词的开始位置	“\b”不作为词语的开始
<code>match_not_eow</code>	非词语的末尾位置	“\b”不作为词语的末尾
<code>match_any</code>	<code>any_match</code>	若存在匹配的可能，则任意匹配均可
<code>match_not_null</code>	<code>not null</code>	空序列不能匹配
<code>match_continuous</code>	连续的	表达式必须匹配一个子表达式，从其第一个字符即开始匹配
<code>match_prev_avail</code>	之前有效	在第一个字符之前的一个或多个字符有效。此时 <code>match_not_bol</code> 和 <code>match_not_bow</code> 是无效的，是被忽略的
<code>format_default</code>	<code>default</code>	与 <code>match_default</code> 相同，常量拥有值 <code>zero**</code>
<code>format_sed</code>		这 3 个参数被 <code>regex_match</code> 忽略
<code>format_no_copy</code>		
<code>format_first_only</code>		

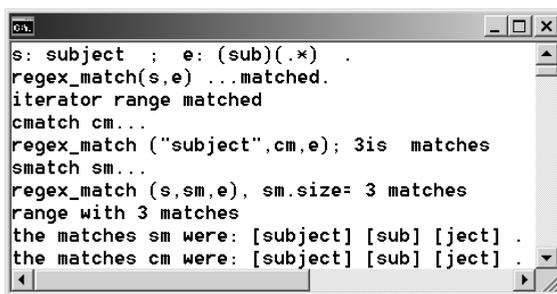
- 1) 这些位掩码标识名称在命名空间 `std::regex_constants` 中是有效的。
- 2) 若其余标识被设置, 则零值常量会被忽略。

关于算法 `regex_match()` 函数的使用方法参见例 17-7。

例 17-7

```
#include <iostream>
#include <string>
#include <regex>
using namespace std;
int main ()
{
    string s ("subject");
    regex e ("(sub)(.*)");
    cout << "s: subject ; e: (sub)(.*) ." << endl;
    if (regex_match ("subject", std::regex("(sub)(.*)")) )
        cout << "regex_match(s,e).....matched \n";
    else
        cout << "regex_match(s,e) is not matcged." << endl;
    if ( std::regex_match (s.begin(), s.end(), e) )
        cout << "iterator range matched\n";
    cmatch cm;    // same as std::match_results<const char* > cm;
    cout << "cmatch cm.." << endl;
    regex_match ("subject", cm,e);
    cout << "regex_match (\"subject\",cm,e); " << cm.size() << "is matches\n";
    smatch sm;    // same as std::match_results<string::const_iterator > sm;
    cout << "smatch sm.." << endl;
    regex_match (s,sm,e);
    cout << "regex_match (s,sm,e), sm.size = " << sm.size() << " matches\n";
    regex_match ( s.cbegin(), s.cend(), sm, e);
    cout << "range with " << sm.size() << " matches" << endl;
    std::cout << "the matches sm were: ";
    for (unsigned i=0; i<sm.size();++i) {
        std::cout << "[" << sm[i] << "] ";
    }
    cout << "." << endl;
    // using explicit flags:
    regex_match ("subject", cm, e, std::regex_constants::match_default );
    std::cout << "the matches cm were: ";
    for (unsigned i=0; i<cm.size();++i) {
        std::cout << "[" << cm[i] << "] ";
    }
    std::cout << "." << std::endl;
    return 0;
}
```

例 17-7 的执行效果如图 17-5 所示。



```

s: subject ; e: (sub)(.*) .
regex_match(s,e) ...matched.
iterator range matched
cmatch cm...
regex_match ("subject",cm,e); 3is matches
smatch sm...
regex_match (s,sm,e), sm.size= 3 matches
range with 3 matches
the matches sm were: [subject] [sub] [ject] .
the matches cm were: [subject] [sub] [ject] .

```

图 17-5 例 17-7 的执行效果



提示 希望读者认真体会函数模板 `regex_match()` 的各种用法, 并尽可能熟练掌握其中一或两种。

17.4.2 正则搜索算法 `regex_search`

正则搜索算法 `regex_search()` 的声明形式为:

```

template < class BidirectionalIterator, class Allocator, class charT, class traits > bool regex_
search(BidirectionalIterator first,
      BidirectionalIterator last,match_results <BidirectionalIterator, Allocator > & m,const basic_
regex < charT, traits > & e,
      regex_constants::match_flag_type flags = regex_constants::match_default);

template < class charT, class Allocator, class traits > bool regex_search(const charT* str,match_re-
sults < const charT* ,
      Allocator > & m,const basic_regex < charT, traits > & e,regex_constants::match_flag_type flags =
      regex_constants::match_default);

template < class ST, class SA, class Allocator, class charT, class traits > bool regex_search(const
basic_string < charT, ST,
SA > & s,match_results < typename basic_string < charT, ST, SA >::const_iterator,Allocator > & m,
const basic_regex < charT,
traits > & e,regex_constants::match_flag_type flags = regex_constants::match_default);

template < class BidirectionalIterator, class charT, class traits > bool regex_search(Bidirection-
alIterator first,
      BidirectionalIterator last,const basic_regex < charT, traits > & e,regex_constants::match_flag_
type flags =
      regex_constants::match_default);

template < class charT, class traits > bool regex_search(const charT* str,const basic_regex < charT,
traits > & e,
      regex_constants::match_flag_type flags = regex_constants::match_default);

template < class ST, class SA, class charT, class traits > bool regex_search(const basic_string <
charT, ST, SA > & s,
const basic_regex < charT, traits > & e,regex_constants::match_flag_type flags = regex_constants::
match_default);

```

此函数用于判断在`[first, end]`内是否存在与正则表达式`e`相匹配的子表达式。参数`flags`用于控制表达式如何和字符序列相匹配。若序列存在，此函数返回`true`；否则，返回`false`。参数`m`的作用见表 17-7。参数`flags`的作用见表 17-8。

函数模板`regex_search()`的具体使用方法参见例 17-8。

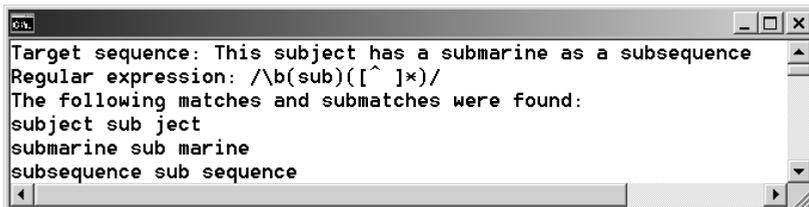
例 17-8

```
#include <iostream>
#include <string>
#include <regex>
using namespace std;
int main ()
{
    string s ("This subject has a submarine as a subsequence");
    smatch m;
    regex e ("\\b(sub) ([^ ]* )"); // matches words beginning by "sub"

    cout << "Target sequence: " << s << endl;
    cout << "Regular expression: /\\b(sub) ([^ ]*)/" << endl;
    cout << "The following matches and submatches were found:" << endl;

    while (regex_search (s,m,e))
    {
        for (auto x:m) //注意此处
            cout << x << " ";
        cout << endl;
        s = m.suffix().str(); //继续找
    }
    return 0;
}
```

例 17-8 的执行效果如图 17-6 所示。



```
Target sequence: This subject has a submarine as a subsequence
Regular expression: /\\b(sub) ([^ ]*)/
The following matches and submatches were found:
subject sub ject
submarine sub marine
subsequence sub sequence
```

图 17-6 例 17-8 的执行效果

17.4.3 正则替换算法 `regex_replace`

正则替换算法 `regex_replace` 的声明形式如下：

```
template < class OutputIterator, class BidirectionalIterator, class traits, class charT, class ST,
class SA > OutputIterator
    regex_replace (OutputIterator out, BidirectionalIterator first, BidirectionalIterator last,
```

```

const basic_regex<charT, traits>& e, const basic_string<charT, ST, SA>& fmt,
    regex_constants::match_flag_type flags = regex_constants::match_default);
template <class OutputIterator, class BidirectionalIterator, class traits, class charT> OutputIter-
ator
    regex_replace( OutputIterator out, BidirectionalIterator first, BidirectionalIterator last,
        const basic_regex<charT, traits>& e, const charT* fmt, regex_constants::match_flag_
type flags
        = regex_constants::match_default);

```

上述两种形式最复杂，功能也最复杂。其作用是构造一个迭代器对象 *i*，并使用 *i* 来枚举 `match_result` 类型的所有匹配。这些匹配应该全部包含在序列 `[first, last]` 中。若没有相关的匹配，并且 `(flags®ex_constant::format_no_copy)` 的值为 `false`，则调用 `copy(first, last, out)`。若有任意匹配被发现，对于每个匹配而言，如果满足条件！`(flags®ex_constants::format_no_copy)`，则调用 `std::copy(m.prefix().first, m.prefix().second, out)`；对于第一种形式，调用 `m.format(out, fmt, flags)`；对于第二种函数形式，调用 `m.format(out, fmt, fmt + char_traits<charT>::length(fmt), flags)`。

最终，如果该匹配被发现，并且！`(flags®ex_constants::format_no_copy)` 为 `true`，调用 `copy(last_m.suffix().first, last_m.suffix().second, out)`，该处 `last_m` 是最后一个匹配的备份。如果 `flags®ex_constants::format_first_only` 是非零数值，仅仅第一个匹配被替换。

```

template <class traits, class charT, class ST, class SA, class FST, class FSA> basic_string<charT,
ST, SA> regex_replace
    (const basic_string<charT, ST, SA>& s, const basic_regex<charT, traits>& e, const basic_
string<charT, FST, FSA>
    & fmt, regex_constants::match_flag_type flags = regex_constants::match_default);
template <class traits, class charT, class ST, class SA> basic_string<charT, ST, SA> regex_re-
place(const basic_string
    <charT, ST, SA>& s, const basic_regex<charT, traits>& e, const charT* fmt, regex_constants::
match_flag_type flags =
    regex_constants::match_default);

```

上述两种形式的作用是构造一个空字符串，并调用 `regex_replace(back_inserter(result), s.begin(), s.end(), e, fmt, flags)` 进行相应的替换工作。

```

template <class traits, class charT, class ST, class SA> basic_string<charT> regex_replace(const
charT* s, const
    basic_regex<charT, traits>& e, const basic_string<charT, ST, SA>& fmt, regex_constants::
match_flag_type flags
    = regex_constants::match_default);
template <class traits, class charT> basic_string<charT> regex_replace(const charT* s,
const basic_regex<charT, traits>& e, const charT* fmt, regex_constants::match_flag_type flags;

```

上述这两种形式的作用是构造一个空的 `string`，并调用 `regex_replace(back_inserter(result), s, s + char_traits<charT>::length)`。

函数的返回值是一个字符串对象，该字符串代表结果序列。

参数 `fmt` 的使用方法见表 17-6。参数 `flags` 的使用方法见表 17-9。

表 17-9 `flags` 的使用方法

标识	作用	提示
<code>match_default</code>	默认作用	默认匹配行为, 常量拥有值 <code>zero**</code>
<code>match_not_bol</code>	非行的起始位置	第一个字符不认为是行的起始位置 (“ <code>^</code> ” 是不匹配的)
<code>match_not_eol</code>	非行的终止位置	最后一个字符不认为是行的末尾 (“ <code>\$</code> ” 是不匹配的)
<code>match_not_bow</code>	非单词的开始位置	“ <code>\b</code> ” 不作为词语的开始
<code>match_not_eow</code>	非词语的末尾位置	“ <code>\b</code> ” 不作为词语的末尾
<code>match_any</code>	<code>any_match</code>	若存在匹配的可能, 任意匹配均可
<code>match_not_null</code>	<code>not null</code>	空序列不能匹配
<code>match_continuous</code>	连续的	表达式必须匹配一个子表达式, 从其第一个字符即开始匹配
<code>match_prev_avail</code>	之前有效	在第一个字符之前的一个或多个字符有效。此时 <code>match_not_bol</code> 和 <code>match_not_bow</code> 是无效的, 是被忽略的
<code>format_default</code>	<code>default</code>	和 <code>match_default</code> 是相同的, 这个常量拥有值: <code>zero**</code>
<code>format_sed</code>	<code>sed</code> (stream EDitor) 格式化	使用和 POSIX 中 <code>sed</code> 应用相同的原则, 来替换匹配
<code>format_no_copy</code>	不复制	当替换匹配时, 目标表达式中不匹配的正则表达式不进行复制
<code>format_first_only</code>	仅仅第一次出现	仅仅首次出现的正则表达式才进行替换

函数模板 `regex_replace()` 的使用方法参见例 17-9。

例 17-9

```
#include <iostream>
#include <string>
#include <regex>
#include <iterator>
using namespace std;
int main ()
{
    string s ("there is a subsequence in the string\n");
    // matches words beginning by "sub"
    regex e ("\\b(sub)([^\n]*)");
    // using string/c - string (3) version:
    cout << std::regex_replace (s,e,"sub- $ 2");
    // using range/c - string (6) version:
    string result;
    regex_replace (std::back_inserter(result), s.begin(), s.end(), e, "$ 2");
    cout << result;
    // with flags:
    cout << std::regex_replace (s,e,"$ 1 and $ 2",std::regex_constants::format_no_copy);
    cout << std::endl;
    return 0;
}
```

例 17-9 的执行效果如图 17-7 所示。

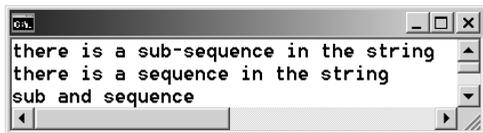


图 17-7 例 17-9 的执行效果

17.5 正则表达式的迭代器

头文件 `<regex>` 提供了两个迭代器类型：`regex_iterator` 和 `regex_token_iterator`。

17.5.1 迭代器 `regex_iterator`

类模板 `regex_iterator` 是一种迭代器适配器。`regex_iterator` 用于描述序列中正则表达式所涉及的所有迭代器序列。迭代器 `regex_iterator` 使用 `regex_search()` 找到序列中连续的正则表达式匹配。迭代器被构造之后，每次 `operator++` 被使用时，迭代器会找到并存储 `match_result` 类型的值。若达到序列的末尾，则迭代器变成等于序列末尾。默认的构造器构造序列尾端迭代器对象，只有合法的迭代器才可用于指向末尾位置。对序列末尾进行 `operator*` 的结果是不确定的。对于任何其他迭代器，使用 `operator*` 之后会返回一个 `match_result` 类型的常量。同样，在序列末尾时，运算符 `operator->` 的结果也是不确定的，对于其他迭代器的值，会返回一个 `match_result` 类型指针。另外，不可能存储内容至迭代器中。两个序列末端的迭代器总是相等的。序列末尾的迭代器是不等于非序列尾端的迭代器的。当两个非序列尾端迭代器使用相同参数构造时，这两个非序列尾端的迭代器是可以相等的。

迭代器模板的声明也是很复杂的，包含如下诸多内容：

```
namespace std {
template <class BidirectionalIterator, class charT = typename iterator_traits<BidirectionalItera-
tor>::value_type,
class traits = regex_traits<charT> > class regex_iterator
{
public:
    typedef basic_regex<charT, traits> regex_type;
    typedef match_results<BidirectionalIterator> value_type;
    typedef std::ptrdiff_t difference_type;
    typedef const value_type* pointer;
    typedef const value_type& reference;
    typedef std::forward_iterator_tag iterator_category;
    regex_iterator(); //构造一个尾端迭代器
    regex_iterator(BidirectionalIterator a, BidirectionalIterator b, const regex_type& re, regex_con-
stants::match_flag_type
    m = regex_constants::match_default); //在[a,b]范围内搜索 re,并获取其迭代器
    regex_iterator(const regex_iterator&);
    regex_iterator& operator = (const regex_iterator&);
```

```

bool operator == (const regex_iterator&) const;           //比较
bool operator != (const regex_iterator&) const;         //!= =
const value_type& operator* () const;
const value_type* operator -> () const;
regex_iterator& operator++ ();
regex_iterator operator++ (int);
private:
    // these members are shown for exposition only:
    BidirectionalIterator begin;
    BidirectionalIterator end;
    const regex_type* pregex;
    regex_constants::match_flag_type flags;
    match_results<BidirectionalIterator> match;
};
}

```

若要迭代所有正则搜索到的匹配表达式，则可以使用正则迭代器。这些迭代器是 `regex_iterator<>` 类型，具有通常的 `string` 类型或者字符序列类型的实例，并且具有前缀 `s`、`c` 或 `wc`。

其实对于迭代器而言，最常用的成员函数是构造函数、两个比较运算符（`==`、`!=`）、引用运算（`*`、`->`）以及自加函数（`++`）。

构造函数有 3 种形式，第 3 种是利用现有的迭代器构造新的迭代器。

```
regex_iterator();
```

其作用是构造一个序列尾端迭代器。

```

regex_iterator (BidirectionalIterator a, BidirectionalIterator b, const regex_type& re, regex_constants::match_flag_type
    m = regex_constants::match_default);           //在[a,b]范围内搜索 re,并返回其迭代器

```

其作用是：在序列 `[a, b]` 中搜索 `re`，然后确定其位置，并返回其迭代器。

例 17-10

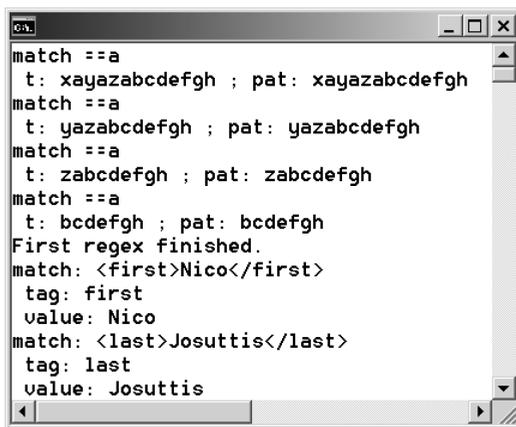
```

#include <string>
#include <regex>
#include <iostream>
#include <algorithm>
using namespace std;
const char * pat = "axayazabcdeefgh";
typedef std::regex_iterator<const char* > myiter;
string t;
int main()
{
    myiter::regex_type rx("a");
    myiter next(pat, pat + strlen(pat), rx);
    myiter end;
    for (;next != end; ++next)
    {
        cout << "match == " << next -> str() << endl;
    }
}

```

```
t = next -> suffix(). str();
pat = t. data();
cout << " t: " << t << " ; pat: " << pat << endl;
}
cout << "First regex finished. " << endl;
myiter it1(next);
myiter it2;
it2 = it1;
//another
string data = "<person> \n"
    " <first>Nico</first> \n"
    " <last>Josuttis</last> \n"
    "</person> \n";
regex reg("<(.*)> (.*) </(\\1)>");
sregex_iterator pos(data.cbegin(), data.cend(), reg);
sregex_iterator endl;
for (; pos != endl; ++pos) {
    cout << "match: " << pos->str() << endl;
    cout << " tag: " << pos->str(1) << endl;
    cout << " value: " << pos->str(2) << endl;
}
}
```

例 17-10 的执行效果如图 17-8 所示。



```
match ==a
t: xayazabcdefgh ; pat: xayazabcdefgh
match ==a
t: yazabcdefgh ; pat: yazabcdefgh
match ==a
t: zabcdefgh ; pat: zabcdefgh
match ==a
t: bcdefgh ; pat: bcdefgh
First regex finished.
match: <first>Nico</first>
tag: first
value: Nico
match: <last>Josuttis</last>
tag: last
value: Josuttis
```

图 17-8 regex_iterator 类型迭代器的用法

17.5.2 迭代器 regex_token_iterator

类模板 regex_token_iterator 是一个迭代器适配器，即它代表一个现存的迭代器序列。对于每次匹配的结果，类模板会枚举序列中的所有正则表达式，并展示所有的一个或多个子表达式。迭代器枚举的每个位置是一个 sub_match 类模板实例，该实例表明一个特定的子表达式是如何和正则表达式匹配的。

当类 `regex_token_iterator` 用于枚举一个单一的子表达式时，该迭代器执行字段分割——枚举每个不符合正则规则的子表达式，即和正则表达式不匹配的字段。

迭代器一旦被构造，将找到并存储一个 `regex_iterator < BidirectionalIterator >` 位置的值，并设置内部计数器 `N` 为 0。并且它也维护一个序列 `subs`，该序列中包含子表达式的清单，以备枚举之需。每次执行 `operator ++` 之后，计数器 `N` 会增加。如果 `N` 超过或者等于子表达式个数，迭代器增加器的成员 `position` 和计数器 `N` 会被设置为 0。

如果到达了序列的末尾，迭代器变成等于序列尾端迭代器的值。除被枚举的非子表达式具有下标 `index - 1`，迭代器会枚举最后一个子表达式，其中包含所有（最后一个匹配的子表达式尾端，至被枚举的输入序列尾端）字符，并且不会是空的子表达式。

默认构造器构造一个序列末端的迭代器对象，这是仅有的合法迭代器，用于尾端位置。当 `operator *` 应用于序列的一端时，其返回结果是不确定的。任意其他的迭代器值是一个 `sub_match < BidirectionalIterator >` 类型的常量值。序列端迭代器使用 `operator->` 时，其结果也是不确定的。对于任意迭代器的值，其返回结果是一个 `sub_match < BidirectionalIterator > *` 的值。

不能存储内容至 `regex_token_iterator` 类型迭代器中。两个末端迭代器总是相等的。一个序列末端迭代器是不能和非末端迭代器相等的。对于两个序列末端迭代器，如果是使用相同的参数构造而来的，那么这两个末端迭代器是可以相等的。

后缀迭代器是一个 `regex_token_iterator` 类型对象，该迭代器指向字符序列的末端。在后缀迭代器中，成员 `result` 保持一个指向数据 `suffix` 的指针，成员 `suffix.match` 的值是 `true`，`suffix.first` 指向最后序列的起始端，`suffix.second` 指向最后序列的末端。

如果 `subs[N] == -1`，或者 `(*position)[subs[N]]`，或任意 `subs [N]` 的其他数值，与其相匹配的是 `(*position).prefix()`。

正则迭代器有助于迭代匹配的子序列，然而有时也会处理匹配的子表达式之间的所有内容。若想使一个特征字符串成为分段的标识（被某些东西隔断），则可以将其作为特定一个正则表达式。类 `regex_token_iterator < >` 具有通常的实例，可用于字符串和字符序列。

再者，为了初始化迭代器的对象，可以传递给迭代器的参数包括序列的起始端、序列的末端和一个正则表达式。另外，还可以将标识化的元素赋予整型值：`-1` 意味着匹配的表达式之间的所有子序列；`0` 意味着所有匹配的正则表达式；其他数值意味着在正则表达式中相匹配的第 `n` 个子表达式。

该类的声明形式如下：

```
namespace std {
template <class BidirectionalIterator, class charT = typename iterator_traits<BidirectionalIterator>::value_type, class traits =
    regex_traits<charT> >
class regex_token_iterator {
public:
    typedef basic_regex<charT, traits> regex_type;
    typedef sub_match<BidirectionalIterator> value_type;
    typedef std::ptrdiff_t difference_type;
    typedef const value_type* pointer;
    typedef const value_type& reference;
```

```

        typedef std::forward_iterator_tag iterator_category ;
        regex_token_iterator ();
regex_token_iterator (BidirectionalIterator a, BidirectionalIterator b, const regex_type& re, int
submatch = 0,
        regex_constants::match_flag_type m = regex_constants::match_default);
regex_token_iterator (BidirectionalIterator a, BidirectionalIterator b, const regex_type& re, const
std::vector<int>&
        submatches, regex_constants::match_flag_type m = regex_constants::match_default);
regex_token_iterator (BidirectionalIterator a, BidirectionalIterator b, const regex_type& re, ini-
tializer_list<int> submatches,
        regex_constants::match_flag_type m = regex_constants::match_default);
template <std::size_t N> regex_token_iterator (BidirectionalIterator a, BidirectionalIterator b,
const regex_type& re, const
int (&submatches) [N], regex_constants::match_flag_type m = regex_constants::match_default);
        regex_token_iterator (const regex_token_iterator&);
        regex_token_iterator& operator = (const regex_token_iterator&);
        bool operator == (const regex_token_iterator&) const;
        bool operator != (const regex_token_iterator&) const;
        const value_type& operator* () const;
        const value_type* operator - > () const;
        regex_token_iterator& operator++ ();
        regex_token_iterator operator++ (int);
private: // data members for exposition only:
        typedef regex_iterator<BidirectionalIterator, charT, traits> position_iterator ;
        position_iterator position ;
        const value_type* result ;
        value_type suffix ;
        std::size_t N ;
        std::vector<int> subs ;
};
}

```

由上述可知，类模板中包含 5 个构造函数。第 1 个构造函数是无参数的；其余 4 个均包含相应的参数。

```
regex_token_iterator ();
```

其作用是构造一个序列末端的迭代器，并且会初始化向量组成员 `subs`，保存单个的 `submatch` 类型对象。其余的构造函数作用各不相同，均有轻微的差别。每个构造器均设置内部计数器 `N` 为 0。若 `position` 不是序列末端迭代器，则构造器设置 `result` 为当前匹配的地址。若 `subs` 中储存的任意数值为 -1，构造器设置 `*this` 指针为一个后缀迭代器；否则，构造器设置 `*this` 指针为序列末端迭代器。

该迭代器模板的使用方法参见例 17-11 和例 17-12。

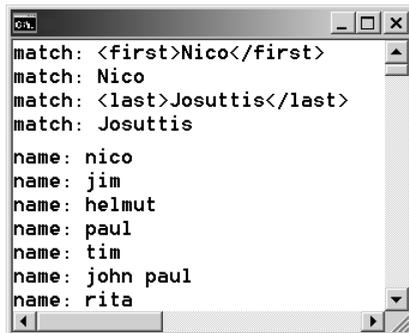
例 17-11

```

#include <string>
#include <regex>
#include <iostream>
#include <algorithm>
using namespace std;
int main()
{
    int dim[2] = {0,2};
    string data = "<person> \n"
        " <first>Nico</first> \n"
        " <last>Josuttis</last> \n"
        "</person> \n";
    regex reg("<(.* )>(.* )</(\\d)>"); //遍历所有相匹配的子串 (使用 regex_token_
        iterator):
    sregex_token_iterator pos(data.cbegin(),data.cend(), //序列形式
        reg, //间隔符号表达式
        dim);
    sregex_token_iterator end;
    for ( ; pos! =end ; ++pos )
    {
        cout << "match: " << pos->str() << endl;
    }
    cout << endl;
    string names = "nico, jim, helmut, paul, tim, john paul, rita";
    regex sep("[ \\t\\n]* [,;. ] [ \\t\\n]* "); //以“:”或“.”与空格间隔
    sregex_token_iterator p(names.cbegin(),names.cend(), //序列形式
        sep, // 间隔符
        -1); // -1: 间隔符之间的值
    sregex_token_iterator e;
    for ( ; p! =e ; ++p )
    {
        cout << "name: " << * p << endl;
    }
}

```

例 17-11 的执行效果如图 17-9 所示。



```

C:\>
match: <first>Nico</first>
match: Nico
match: <last>Josuttis</last>
match: Josuttis

name: nico
name: jim
name: helmut
name: paul
name: tim
name: john paul
name: rita

```

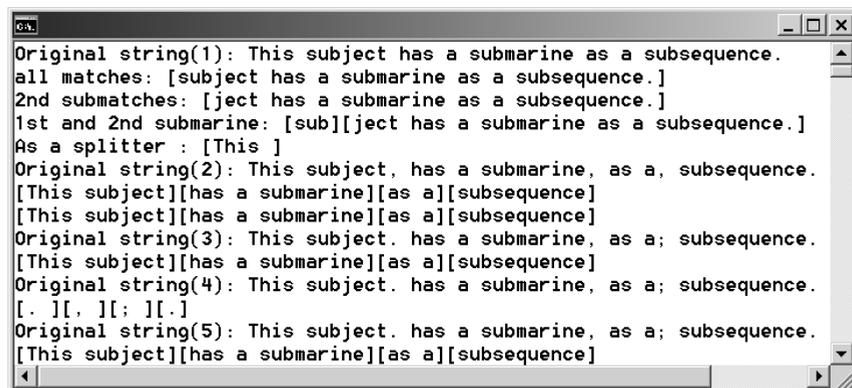
图 17-9 例 17-11 的执行效果

例 17-12

```
#include <iostream>
#include <regex>
#include <string>
using namespace std;
int main()
{
    string s("This subject has a submarine as a subsequence. ");
    cout << "Original string (1): " << s << endl;
    regex rgx("\\b(sub)([ ]*)");
    typedef regex_token_iterator <string::iterator> mytoken_iter;
    mytoken_iter rend;
    cout << "all matches: ";
    mytoken_iter a(s.begin(), s.end(), rgx);
    while(a != rend)
    {
        cout << "[" << *a++ << "]";
    }
    cout << endl;
    cout << "2nd submatches: ";
    mytoken_iter b(s.begin(), s.end(), rgx, 2);
    while(b != rend)
    {
        cout << "[" << *b++ << "]";
    }
    cout << endl;
    cout << "1st and 2nd submarine: ";
    int dim[2] = {1, 2};
    mytoken_iter c(s.begin(), s.end(), rgx, dim);
    while(c != rend)
    {
        cout << "[" << *c++ << "]";
    }
    cout << endl;
    sregex_token_iterator rend1;
    cout << "As a splitter : ";
    sregex_token_iterator d(s.cbegin(), s.cend(), rgx, -1);
    while(d != rend1)
    {
        cout << "[" << *d << "]";
        d++;
    }
    cout << endl;
    string s1("This subject, has a submarine, as a, subsequence. ");
    cout << "Original string (2): " << s1 << endl;
    regex rgx1("[ \\t\\n]* [,;.][ \\t\\n]*");//[ \\t\\n]* [,;.][ \\t\\n]* //\\b(,
```

```
//mytoken_iter e(s1.cbegin(),s1.cend(),rgx1,-1);
sregex_token_iterator e(s1.cbegin(),s1.cend(),rgx1,-1);
while(e!=rendl)
{
    cout<<"["<<*e<<"]";
    e++;
}
cout<<endl;
// cout<<"regex (\"[,]* \")"
regex rgx2("[ \\t\\n]* [,;.][ \\t\\n]*");
sregex_token_iterator f(s1.cbegin(),s1.cend(),rgx2,-1);
while(f!=rendl)
{
    cout<<"["<<*f<<"]";
    f++;
}
cout<<endl;
string s2("This subject. has a submarine, as a; subsequence.");
cout<<"Original string (3): "<<s2<<endl;
regex rgx3("[ \\t]* [,;.][ \\t]*");
sregex_token_iterator g(s2.cbegin(),s2.cend(),rgx3,-1);
while(g!=rendl)
{
    cout<<"["<<*g<<"]";
    g++;
}
cout<<endl;
cout<<"Original string (4): "<<s2<<endl;
regex rgx4("[ \\t]* [,;.][ \\t]*"); // [ \\t]* [,;.][ \\t]*
sregex_token_iterator h(s2.begin(),s2.end(),rgx4);
while(h!=rendl)
{
    cout<<"["<<*h<<"]";
    h++;
}
cout<<endl;
cout<<"Original string (5): "<<s2<<endl;
regex rgx5("[ \\t]* [,;.][ \\t]*"); //\\b(sub)([^]* )
sregex_token_iterator h1(s2.begin(),s2.end(),rgx5,-1);
while(h1!=rendl)
{
    cout<<"["<<*h1<<"]";
    h1++;
}
cout<<endl;
}
```

例 17-12 的执行效果如图 17-10 所示。



```
Original string(1): This subject has a submarine as a subsequence.
all matches: [subject has a submarine as a subsequence.]
2nd submatches: [ject has a submarine as a subsequence.]
1st and 2nd submarine: [sub][ject has a submarine as a subsequence.]
As a splitter : [This ]
Original string(2): This subject, has a submarine, as a, subsequence.
[This subject][has a submarine][as a][subsequence]
[This subject][has a submarine][as a][subsequence]
Original string(3): This subject. has a submarine, as a; subsequence.
[This subject][has a submarine][as a][subsequence]
Original string(4): This subject. has a submarine, as a; subsequence.
[. ][, ][; ][.]
Original string(5): This subject. has a submarine, as a; subsequence.
[This subject][has a submarine][as a][subsequence]
```

图 17-10 例 17-12 的执行效果

注: `sregex_token_iterator` 和 `cregex_token_iterator` 是 `regex_token_iterator` 的实例化类, 可供程序员直接使用。

17.6 小结

本章详细讲解了正则表达式的定义和要求、类模板 `basic_regex`、两个模板类 `sub_match` 和 `match_results`、3 种相关的算法 (`regex_replace`、`regex_match` 和 `regex_search`) 以及两个迭代式模板类。

本章内容比较复杂, 也比较难以理解。希望读者认真阅读此章内容, 并认真阅读例题。

附录 部分 C 函数库详解

使用 C 语言的价值在于使用其标准函数。在解决实际问题时，能方便地操作字符串和文件等对象是非常重要的。几乎没有哪种计算机语言能像 C 语言那样能出色地完成全部工作。C 函数库目前没有图形函数，没有全屏幕文本操作函数，信号机制相当弱，并且根本没有多任务机制和没有提供常规内存之外的内存支持。尽管 C 语言有上述缺陷，但它为所有程序提供了一套基本功能，无论是多任务、多窗口，还是其他更复杂的环境。C 函数库的缺陷在编译程序开发商和第三方函数库得到弥补。总而言之，C 函数库为程序设计提供了非常坚实的基础。

函数库的优点包括准确性、高效性和可移植性。编译程序的开发商在函数库出厂之前肯定做了全面的检测和测试，可以保证函数库的准确性。优秀的 C 程序员会大量使用标准函数库，所以如果编译程序的开发商能够提供一套出色的标准函数，就会在竞争中占优势。

每一个库函数，都需要一个对应的头文件。这个头文件提供该函数的原型。只有在源程序的开始部分包含了该头文件之后，该函数才可以被使用；否则，程序编译时该函数无法被识别。

附录 A 数学函数

头文件 `<math.h>` 声明了两种数据类型：数学函数和宏。两个数据类型分别为：`float_t` 和 `double_t`。这两个数据类型至少和 `float` 类型及 `double` 类型占用相同的字节。并且，`double_t` 类型至少要和 `float_t` 类型同等宽度。如果宏 `FLT_EVAL_METHOD` 等于 0，`float_t` 类型相当于 `float` 类型，`double_t` 类型相当于 `double` 类型；如果宏 `FLT_EVAL_METHOD` 等于 1，`float_t` 类型和 `double_t` 类型都相当于 `double` 类型；如果宏 `FLT_EVAL_METHOD` 等于 2，`float_t` 类型和 `double_t` 类型都相当于 `long double` 类型。`FLT_EVAL_METHOD` 等于其他值的情况至今还没有定义。

函数库中的函数原型均有 3 种：`double`、`float` 和 `long double`。此处只讲述 `double` 类型，对于其余类型，只要进行相应的替换即可。

A.1 数学函数库中的宏

数学函数库包含了一部分宏。常见的宏如下：

`HUGE_VAL`——表示正的 `double` 常量，代表正的无穷大。

`HUGE_VALF`——表示浮点类型的正值无穷大。

`HUGE_VALL`——表示 `long double` 类型的正的无穷大。

`INFINITY`——表示浮点类型 (`float`) 正的无穷大。

NAN——表示无效数字，即不是一个数。

FP_INFINITE——表示浮点类型的无穷大。

FP_NAN——表示无效数据。

FP_NORAMAL——表示正常状态。

FP_SUBNORMAL——表示异常状态。

FP_ZERO——表示浮点类型的 0。

FP_FAST_FMA——表示和 `fma()` 函数同样的效果。

FP_FAST_FMAF——表示浮点类型的 FP_FAST_FMA。

FP_FAST_FMAL——表示 long double 类型的 FP_FAST_FMA。

FP_ILOGB0——表示 `ilogb(x)` 函数的 `x` 参数为 0 时，函数的返回值。

FP_ILOGBNAN——表示 `ilogb(x)` 函数的 `x` 参数为 NAN 时，函数的返回值。

MATH_ERRNO——表示数学错误，整数 1。

MATH_ERREXCEPT——表示数学异常错误，整数 2。

A.2 浮点计算减法协议开关

在进行浮点计算时，FP_CONTRACT 宏可以用来判断是否发生减法运算。因为浮点数计算在计算机语言的底层是一个非常复杂的过程，如果处理不当会造成计算结果偏差。常见的用法：

```
#include <math.h>
#pragma STDC FP_CONTRACT on-off-switch
```

FP_CONTRACT 默认是 on 状态，其用法如下：

```
#include <stdio.h>
#include <float.h>
#pragma fp_contract (off)
void main()
{
...
}
```

A.3 数学库中的宏函数

C 语言的数学函数库还包含部分宏函数，主要有 `fpclassify()`、`isfinite()`、`isinf()`、`isnan()`、`isnormal()` 和 `signbit()`。使用这些宏函数时，必须包含头文件 `<math.h>`。

- `fpclassify(float x)` 函数返回值是一个分类宏，用于说明参数 `x` 的类型。
- `isfinite(float x)` 函数用于判断参数 `x` 是否是一个有限的值。当且仅当参数 `x` 是一个有限的值时，函数返回值是一个非零数。
- `isinf(float x)` 函数用于判断参数 `x` 是否是无限的数。当且仅当参数 `x` 是一个无限的数时，函数返回一个非零数。
- `isnan(float x)` 函数用于判断参数 `x` 是否是 NAN。当且仅当参数 `x` 是 NAN 时，函数返回一个非零数值。

- `isnormal (float x)` 函数用于判断参数 `x` 是否是一个正常的数值。当且仅当参数 `x` 是一个正常的数值时, 函数返回一个非零数值。
- `signbit (float x)` 函数用于判断参数 `x` 是否是一个有符号数 (即负数)。当且仅当参数 `x` 是一个负数时, 函数返回一个非零值。

A.4 三角函数和反三角函数

C 语言函数库提供了一系列的三角函数, 主要有 `acos()`、`asin()`、`atan()`、`atan2()`、`cos()`、`sin()`、`tan()`、`acosh()`、`asinh()`、`cosh()`、`sinh()` 和 `tanh()`。使用这些函数时, 需要包含头文件 `<math.h>`。

1) 三角余弦函数: `double cos (double x)`。

说明: 参数 `x` 是弧度数值。

2) 三角正弦函数: `double sin (double x)`。

说明: 参数 `x` 是弧度数值。

3) 三角正切函数: `double tan (double x)`。

说明: 参数 `x` 是弧度数值。

4) 反余弦函数: `double acos (double x)`。

说明: 参数 `x` 是 $[-1, 1]$ 内的实数, 函数返回值是在 $[0, \pi]$ 的实数。如果参数 `x` 超出了 $[-1, 1]$, 函数调用时将发生错误。

5) 反正弦函数: `double asin (double x)`。

说明: 参数 `x` 是 $[-1, 1]$ 内的实数, 函数返回值是在 $[-\pi/2, \pi/2]$ 的实数。如果参数 `x` 超出 $[-1, 1]$, 函数调用时将发生错误。

6) 反正切函数: `double atan (double x)` 和 `double atan2 (double y, double x)`。

说明: 这两个函数的返回值是在 $[-\pi/2, \pi/2]$ 的实数。函数 `atan2 (double y, double x)` 的功能是计算 y/x 的反正切值, 如果两个参数均为 0, 函数调用时将发生错误。

7) 双曲余弦函数: `double cosh (double x)`。

说明: 如果参数 `x` 的值特别大, 函数调用时会发生错误。

8) 双曲正弦函数: `double sinh (double x)`。

说明: 如果参数 `x` 的值特别大, 函数调用时会发生错误。

9) 双曲正切函数: `double tanh (double x)`。

说明: 该函数用于计算双曲正切函数的值。

10) 反双曲余弦函数: `double acosh (double x)`。

说明: 函数返回值在 $[0, \infty]$ 。如果参数 `x` 小于 1, 函数调用时将发生错误。Turbo C 2.0 的数学函数库没有提供本函数。

11) 反双曲正弦函数: `double asinh (double x)`。

说明: 该函数用于计算反双曲正弦函数的值。Turbo C 2.0 的数学函数库没有提供本函数。

12) 反双曲正切函数: `double atanh (double x)`。

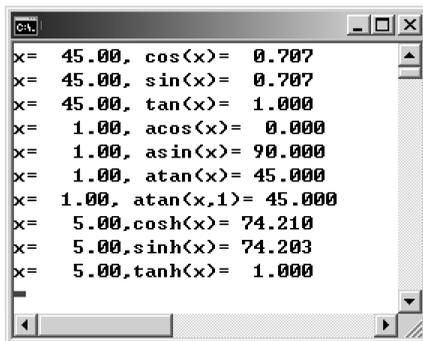
说明: 参数 `x` 必须在范围 $(-1, +1)$ 内。Turbo C 2.0 的数学函数库没有提供本函数。

例 A-1

```
#include <stdio.h>
#include <math.h>
void main()
{
    double x1,x2,x3,x4;
    double pi=3.1415926;
    double y=0;
    x1=45;
    y=cos(x1/180*pi);
    printf("x= %6.2f, cos(x)=%7.3f \n",x1,y);
    y=sin(x1/180*pi);
    printf("x= %6.2f, sin(x)=%7.3f \n",x1,y);
    y=tan(x1/180*pi);
    printf("x= %6.2f, tan(x)=%7.3f \n",x1,y);
    x2=1.0;
    y=acos(x2)/pi*180;
    printf("x= %6.2f, acos(x)=%7.3f \n",x2,y);
    y=asin(x2)/pi*180;
    printf("x= %6.2f, asin(x)=%7.3f \n",x2,y);
    y=atan(x2)/pi*180;
    printf("x= %6.2f, atan(x)=%7.3f \n",x2,y);
    y=atan2(x2,1)/pi*180;
    printf("x=%6.2f, atan(x,1)=%7.3f \n",x2,y);
    x3=5.0;
    y=cosh(x3);
    printf("x= %6.2f, cosh(x)=%7.3f \n",x3,y);
    y=sinh(x3);
    printf("x= %6.2f, sinh(x)=%7.3f \n",x3,y);
    y=tanh(x3);
    printf("x= %6.2f, tanh(x)=%7.3f \n",x3,y);

    getchar();
}
```

例 A-1 的执行效果如图 A-1 所示。



```

x= 45.00, cos(x)= 0.707
x= 45.00, sin(x)= 0.707
x= 45.00, tan(x)= 1.000
x= 1.00, acos(x)= 0.000
x= 1.00, asin(x)= 90.000
x= 1.00, atan(x)= 45.000
x= 1.00, atan(x,1)= 45.000
x= 5.00, cosh(x)= 74.210
x= 5.00, sinh(x)= 74.203
x= 5.00, tanh(x)= 1.000
```

图 A-1 例 A-1 的执行效果

A.5 指数和对数函数

1. 指数函数

C 语言数学函数库中的指数函数包括 `exp()`、`exp2()`、`expm1()`、`frexp()` 和 `ldexp()`。Turbo C 2.0 没有提供 `exp2()` 和 `expm1()`。

1) `exp()` 函数的原型为:

```
double exp(double x);
```

说明: 该函数用于计算指数 e^x 的值。其浮点类型的函数形式为 `expf()`, 长双精度型的函数原型为 `expl()`。如果参数 x 的值太大, 函数调用时可能发生错误。

2) `frexp()` 函数的原型为:

```
double frexp(double value, int * exp);
```

说明: 该函数用于将双精度类型的实数 `value` 表示成 $x * 2^{*exp}$ 的形式。如果参数 `value` 不是浮点数, 函数的返回结果是不确定的。否则, 函数返回值就是 x 的值, `*exp` 将返回一个整数。如果函数调用成功, x 是在 $[1/2, 1)$ 内的数值, 也有可能是 0。如果参数 `value` 为 0, 函数返回值和 `(*exp)` 都是零。如果参数 x 的值太大, 函数调用时可能发生错误。

3) `ldexp()` 函数的原型为:

```
(double ldexp(double x, int exp));
```

说明: 该函数用于求解表达式 $(x * 2^{*exp})$ 的值。如果参数比较大, 有可能调用时由于计算结果太大, 导致数值溢出。

4) `exp2()` 函数的原型为:

```
double exp2(double x);
```

说明: 该函数用于计算指数 2^x 的值。如果参数 x 的值太大, 函数调用时可能发生错误。

5) 函数 `expm1` 的原型为:

```
double expm1(double x);
```

说明: 该函数用于计算表达式 $(e^x - 1)$ 的值。如果参数 x 的值太大, 函数调用时可能发生错误。

2. 对数函数

数学函数库中的对数函数主要包括 `ilogb()`、`logb()`、`log()`、`log10()`、`loglp()`、`log2()`、`modf()`、`scalbn()` 和 `scalbln()`。其中, Turbo C 2.0 的函数库中没有提供 `ilogb()`、`logb()`、`loglp()`、`log2()`、`scalbn()` 和 `scalbln()`。

1) `ilogb()` 函数的原型为 `int ilogb (double x);`

说明: 该函数和 `logb()` 函数差不多, 只不过功能更完善。如果参数 x 等于 0, 函数调用时参数 x 自动转换为 `FP_ILOGB0`; 如果 x 是无限数, 函数调用时参数 x 自动转换为 `INT_MAX`; 如果 x 是 `NAN`, 函数调用时参数 x 自动转换为 `FP_ILOGBNAN`; 其他情况时, 就相当于调用函数 `logb()`。注意: 函数返回值是整数, 具有自动取整的功能。如果 x 等于 0, 函数调用时有可能发生错误。

2) `logb()` 函数的原型为:

```
double logb(double x);
```

说明：在调用 `logb(x)` 函数时，假定返回值是 y ，存在指数函数关系 $x = 2^y$ ，在表达式中， x 即和 `logb()` 参数 x 一致，`logb(x)` 的返回值即是 y 。如果参数 x 等于零，函数调用时会发生错误。

3) `log()` 函数的原型为

```
double log(double x);
```

说明：该函数是以 e 为底的自然对数，返回值是表达式 (\log_e^x) 的值。参数 x 为零时，函数调用时会发生错误。

4) `log10()` 函数的原型为

```
double log10(double x);
```

说明：该函数是以 10 为底的对数计算。如果参数 x 为 0，函数调用时会发生错误。

5) `loglp()` 函数的原型为

```
double loglp(double);
```

说明：该函数用于计算表达式 $(\log_e^{(1+x)})$ 的值。如果参数为 -1 ，函数调用时会发生错误。

6) `log2()` 函数的原型为

```
double log2(double x);
```

说明：该函数是以 2 为底的对数计算。如果参数 x 为 0，函数调用时会发生错误。

7) `modf()` 函数的原型为

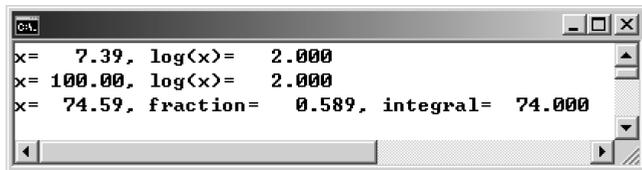
```
double modf(double value, double * iptr)
```

说明：对于浮点数 `value`，函数调用结束后，浮点数的整数部分保留在 `(* iptr)` 中，函数的返回值是浮点数的小数部分。

例 A-2

```
#include <stdio.h>
#include <math.h>
#include <conio.h>
void main()
{
    clrscr();
    double x3 = 7.389;
    double x4 = 100;
    double x5 = 74.589;
    double y = 0;
    double* iptr = NULL;
    y = log(x3);
    printf("x = %6.2f, log(x) = %7.3f \n", x3, y);
    y = log10(x4);
    printf("x = %6.2f, log(x) = %7.3f \n", x4, y);
    y = modf(x5, iptr);
    printf("x = %6.2f, fraction = %7.3f, integral = %7.3f
        \n", x5, y, (* iptr));
    getch();
}
```

例 A-2 的执行效果如图 A-2 所示



```
GA
x= 7.39, log(x)= 2.000
x= 100.00, log(x)= 2.000
x= 74.59, fraction= 0.589, integral= 74.000
```

图 A-2 例 A-2 的执行效果

A.6 幂函数和绝对值函数

C 语言数学函数库中的幂函数主要有 `cbrt()`、`hypot()`、`pow()` 和 `sqrt()`；绝对值函数主要有 `fabs()`。其中，Turbo C 2.0 没有提供函数 `cbrt()`。

1. 常见的幂函数

1) `cbrt()` 函数的原型为

```
double cbrt(double x);
```

说明：该函数用于求解参数 x 的立方根。

2) `hypot()` 函数的原型为

```
double hypot(double x, double y);
```

说明：该函数用于求解表达式 $(\sqrt{x^2 + y^2})$ 的值。

3) `pow()` 函数的原型为

```
double pow(double x, double y);
```

说明：该函数用于求解表达式 (x^y) 的值。如果 x 为负数，并且 y 不是整数，函数调用时会发生错误；如果 x 等于 0，并且 y 小于或者等于 0，函数调用时会发生错误。

4) `sqrt()` 函数的原型为

```
double sqrt(double x);
```

说明：该函数用于求解参数 x 的非负平方根。参数 x 必须是非负数，函数返回值是 x 的非负平方根。

2. 绝对值函数

`fabs()` 函数的原型为

```
double fabs(double x);
```

说明：该函数用于计算浮点数 x 的绝对值。

例 A-3

```
#include <stdio.h>
#include <math.h>
#include <conio.h>
void main()
{
    clrscr();
    double x1=20.0;
```

```
double x2 = -64.0;
double y=0.0;
y=hypot(x1,x2);
printf("x=%6.3f, x2=%6.3f, hypot(x1,x2) = %7.3f \n",x1,x2,y);
x1=2;
x2=3;
y=pow(x1,x2);
printf("x=%6.3f, y=%6.3f, pow(x,y)=%7.3f \n",x1,x2,y);
x1=16.0;
y=sqrt(x1);
printf("x=%6.3f, y=%7.3f\n", x1, y);
getchar();
}
```

例 A-3 的执行效果如图 A-3 所示。

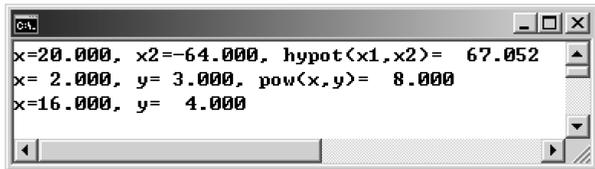


图 A-3 例 A-3 的执行效果

A.7 误差和 gamma 函数

C 语言数学函数库还提供了一套误差函数和 gamma 函数。误差函数主要有 erf() 和 erfc()。gamma 函数主要有 lgamma() 和 tgamma()。Turbo C 2.0 没有提供这 4 个函数。

1. 误差函数

1) erf() 函数的原型为

```
double erf(double x);
```

说明：该函数用于计算表达式 $\left(\frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt\right)$ 的值。

2) erfc() 函数的原型为

```
double erfc(double x)。
```

说明：该函数的返回值等于表达式 $(1 - \text{erf}(x))$ 的值。

2. gamma 函数

1) lgamma() 函数的原型为

```
double lgamma(double x);
```

说明：该函数用于求解表达式 $(\log_e |\Gamma(x)|)$ 的值。如果参数 x 的值太大，函数调用时会发生错误；如果参数 x 的小于或等于 0，函数调用时也会发生错误。

2) tgamma() 函数的原型为

```
double tgamma(double x);
```

说明：该函数用于求解表达式 $(\Gamma(x))$ 的值。其余同 lgamma() 函数。

A.8 近似取整函数

C 语言数学函数库还提供了一系列的近似取整函数, 主要包括 `ceil()`、`floor()`、`nearbyint()`、`rint()`、`lrint()`、`llrint()`、`round()`、`lround()`、`llround()` 和 `trunc()`。Turbo C 2.0 仅提供了 `ceil()` 和 `floor()`。

1) `ceil()` 函数的原型为

```
double ceil(double x);
```

说明: 该函数用于计算不小于 x 的最小整数值。

2) `floor()` 函数的原型为

```
double floor(double x);
```

说明: 该函数用于计算不大于参数 x 的最大整数值。

3) `nearbyint()` 函数的原型为

```
double nearbyint(double x);
```

说明: 该函数用于利用“四舍五入”原则, 求出距离参数 x 最近的整数值。

4) `rint()` 函数的原型为

```
double rint(double x);
```

说明: 该函数的功能是实现四舍五入取整。

函数 `lrint()` 和 `llrint()` 与函数 `rint()` 相仿。

5) `round()` 函数的原型为

```
double round(double x);
```

说明: 利用四舍五入原则, 结果为整数。

`lround()` 函数和 `llround()` 函数与 `round()` 函数基本一致。

例 A-4

```
#include <stdio.h>
#include <math.h>
#include <conio.h>
void main()
{
    clrscr();
    double x1 = 1.6;
    double x2 = 1.4;
    double y1 = 0;
    double y2 = 0;
    y1 = ceil(x1);
    y2 = floor(x1);
    printf("x1 = %5.3f, ceil(x1) = %5.3f, floor(x1) = %5.3f \n", x1, y1, y2);
    y1 = ceil(x2);
    y2 = floor(x2);
    printf("x2 = %5.3f, ceil(x1) = %5.3f, floor(x1) = %5.3f \n", x2, y1, y2);
    getch();
}
```

例 A-4 的执行效果如图 A-4 所示。

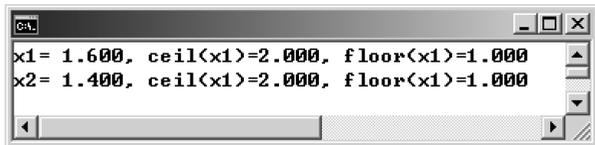


图 A-4 例 A-4 的执行效果

A.9 求余函数

C 语言数学函数库提供了 3 个求余函数，它们分别是 `fmod()`、`remainder()` 和 `remquo()`。Turbo C 2.0 没有提供 `remquo()` 函数和 `remainder()`。

1) `fmod()` 函数的原型为

```
double fmod(double x, double y);
```

说明：该函数用于求取表达式 (x/y) 的余数。

2) `remainder()` 函数的原型为

```
double remainder(double x, double y);
```

说明：该函数用于求取表达式 (x/y) 的余数。

3) `remquo()` 函数的原型为：

```
double remquo(double x, double y, int * quo);
```

说明：`remquo()` 和函数 `remainder()` 功能一样。二者的不同之处在于 `remquo()` 函数增加了 1 个整形指针类型的参数，一般不使用。

例 A-5

```
#include <stdio.h>
#include <math.h>
#include <conio.h>
void main()
{
    clrscr();
    double x1 = 20;
    double x2 = 12;
    double y = 0.0;
    y = fmod(x1, x2);
    printf("x1 = %6.3f, x2 = %6.3f, fmod(x1, x2) = %6.3f \n", x1, x2, y);
    getch();
}
```

例 A-5 的执行效果如图 A-5 所示。



图 A-5 例 A-5 的执行效果

A. 10 操作处理函数

C 语言数学函数库中的操作处理函数包括 `copysign()`、`nan()`、`nextafter()` 和 `nexttoward()`。Turbo C 2.0 的函数库没有提供这 4 个函数。

1) `copysign()` 函数的原型为

```
double copysign(double x, double y);
```

说明：该函数用于将参数 `y` 的符号赋予参数 `x`，函数返回值的大小和 `x` 的绝对值一致，符号和参数 `y` 的符号一致。如果参数 `x` 是 NaN，返回值即为 NaN。

2) `nan()` 函数的原型为

```
double nan(const * tagp);
```

说明：该函数的返回值为 NaN。

3) `nextafter()` 函数的原型为

```
double nextafter(double x, double y);
```

说明：该函数用于返回参数 `x` 在参数 `y` 方向上可以表示的最接近的数值，若 `x` 等于 `y`，则返回 `x`。例如，

```
nextafter(5.0, 4.0)
```

4) `nexttoward()` 函数的原型为

```
double nexttoward(double x, long double y);
```

说明：该函数的功能和 `nextafter()` 函数相同；如果 `x` 等于 `y` 时，函数返回值等于 `y`。

A. 11 最大值、最小值和正差函数

C 语言数学函数库中的最大值、最小值和正差函数分别为 `fmax()`、`fmin()` 和 `fdim()`。Turbo C 2.0 没有提供这 3 个函数。

1) `fdim()` 函数的原型为

```
double fdim(double x, double y);
```

说明：该函数用于判断两个参数的差是否为正值，如果为正值，函数返回值即为该差值；如果两个参数之差为负值，函数返回值为 0。

2) `fmax()` 函数的原型为

```
double fmax(double x, double y);
```

说明：该函数用于返回两个参数中较大的那个参数的值。

3) `fmin()` 函数的原型为

```
double fmin(double x, double y);
```

说明：该函数用于返回两个参数中较小的那个参数的值。

A. 12 浮点乘加函数

C 语言数学函数库包含了一个浮点乘加 `fma()` 函数。Turbo C 2.0 未提供本函数。

函数原型为

```
double fma(double x, double y, double z);
```

说明：该函数用于计算表达式 $(x * y + z)$ 的值。本函数是一个完全的三重运算。

A.13 比较函数（宏）

C 语言数学函数库还提供了一系列的比较宏函数。例如，`isgreater()`、`isgreaterequal()`、`isless()`、`islessequal()`、`islessgreater()`和 `isunordered()`。这些函数的原型分别为：

```
int isgreater(float x, float y);
int isgreaterequal(float x, float y);
int isless(float x, float y);
int islessequal(float x, float y);
int islessgreater(float x, float y);
int isunordered(float x, float y);
```

Turbo C 2.0 没有提供以上函数，但提供了相似的功能函数，例如，`isEqual()`、`isLessThan()`、`islower()`等函数。

附录 B 数据类型转换

C 语言函数库提供了一套数据类型转换函数，大致分为 4 类 `atoi()`、`atof()`和 `atol()`；`itoa()`、`ltoa()`和 `ultoa()`；`fcvt()`、`ecvt()`和 `gcvt()`。

B.1 字符转整数函数（`atoi()`和 `atol()`）

`atoi()`函数属于 C 标准库，使用时需要包含头文件 `<stdlib.h>`。其原型为：

```
int atoi(const char* str);
```

说明：该函数用于将字符串 `str` 转换成一个整数，并返回结果。参数 `str` 以数字开头，当函数从 `str` 中读到非数字字符时会结束转换，并将结果返回。

`atol()`函数的使用方法基本上和 `atoi()`相似。

例 B-1

```
#include <iostream>
#include <cstdlib>
using namespace std;
void main(int argc, char* argv[])
{
    char str[10] = "1234abcd";
    char str1[10] = "4567";
    char str2[10] = "b2345";
    int A = atoi(str);
    int B = atoi(str1);
    int C = atoi(str2);
    cout << "A: " << A << " , " << "B: " << B << " , " << "C : " << C << endl;
}
```

例 B-1 的执行效果为:

A: 1234 , B: 4567 , C :0

B.2 字符型转换浮点型函数 (atof() 和 atol())

atol() 函数的原型为:

```
long int atol ( const char * str )
```

atof() 函数的原型为:

```
double atof(const char* str)
```

说明: 该函数用于将字符串 `str` 转换为一个双精度数值, 并返回结果。参数 `str` 必须以有效数字开头, 允许以 “E” 或 “e” 除外的任意非数字字符结尾。

例 B-2

```
#include <iostream>
#include <cstdlib>
using namespace std;
void main(int argc, char* argv[])
{
    char str[10] = "12.345asd";
    char str1[10] = "123.67e2";
    char str2[10] = "566.ess";
    char str3[10] = "d67.32";
    char str4[10] = "65.3241";
    double A = atof(str);
    double B = atof(str1);
    double C = atof(str2);
    double D = atof(str3);
    double E = atof(str4);
    cout << "A: " << A << " , " << "B: " << B << " , " << "C: "
         << C << " , " << "D: " << D << " , " << "E: " << E << endl;
}
```

例 B-2 的执行效果为:

A: 12.345 , B: 12367 , C: 566 , D:0 , E:65.3241

B.3 整型数转字符串函数 (itoa()、ltoa() 和 ultoa())

itoa() 函数的原型为:

```
char * itoa(int value, char* string, int radix);
```

参数 `value` 是被转换的整数, `string` 是转换后储存的字符数组, `radix` 是转换进制 (2, 8, 10, 16)。

```
char* ltoa(long value, char* str, int radix);
```

参数 `value` 是长整型数值, `str` 是 `value` 转换而来的字符串, `radix` 是转换进制 (2-36)。

```
char* ultoa(unsigned long value, char* str, int radix);
```

参数 value 是要转换的数值，str 用于存储转换的结果，radix 是转换进制（2-36）。

例 B-3

```
#include <iostream>
#include <cstdlib>
using namespace std;
void main(int argc, char* argv[])
{
    //返回 0;
    int A = 37612;
    long B = 98756223;
    unsigned long C = 204123;
    char str[10] = "";
    char str1[10] = "";
    char str2[10] = "";
    itoa(A, str, 10);
    ltoa(A, str1, 10);
    ultoa(A, str2, 10);
    cout << A << ":" << str << endl;
    cout << A << ":" << str1 << endl;
    cout << A << ":" << str2 << endl;
    itoa(B, str, 10);
    ltoa(B, str1, 10);
    ultoa(B, str2, 10);
    cout << B << ":" << str << endl;
    cout << B << ":" << str1 << endl;
    cout << B << ":" << str2 << endl;
    itoa(C, str, 10);
    ltoa(C, str1, 10);
    ultoa(C, str2, 10);
    cout << C << ":" << str << endl;
    cout << C << ":" << str1 << endl;
    cout << C << ":" << str2 << endl;
}
```

例 B-3 的执行效果为：

```
37612:37612
37612:37612
37612:37612
98756223:98756223
98756223:98756223
98756223:98756223
204123:204123
204123:204123
204123:204123
```

B.4 浮点数转换字符串函数

浮点数转换字符串各函数的原型为:

```
char* _fcvt(double value, int count, int* dec, int* sign);
```

说明: `fcvt()` 函数用于转换浮点数为字符串。

```
char* _ecvt(double value, int count, int* dec, int* sign);
```

说明: `ecvt()` 函数用于转换双精度浮点型数为字符串, 转换结果中不包括十进制小数点。

```
char* _gcvt(double value, int digits, char* buffer);
```

说明: `gcvt()` 函数用于将浮点数转换成字符串, 同时返回一个指向字符串的存储位置的指针。

例 B-4

```
#include <iostream>
#include <cstdlib>
using namespace std;
void main(int argc, char* argv[])
{
    int decimal, sign;
    char * buffer;
    double source = 3.1415926535;
    buffer = _fcvt( source, 7, &decimal, &sign ); // C4996
    cout << "source: " << source << ", string: " << buffer << ", decimal:
        " << decimal << ", sign: " << sign << endl;
    double source1 = -2.9879;
    buffer = _fcvt( source1, 7, &decimal, &sign ); // C4996
    cout << "source: " << source1 << ", string: " << buffer << ", decimal:
        " << decimal << ", sign: " << sign << endl;

    buffer = _ecvt( source, 7, &decimal, &sign ); // C4996
    cout << "source: " << source << ", string: " << buffer << ", decimal:
        " << decimal << ", sign: " << sign << endl;

    buffer = _ecvt( source1, 7, &decimal, &sign ); // C4996
    cout << "source: " << source1 << ", string: " << buffer << ", decimal:
        " << decimal << ", sign: " << sign << endl;

    _gcvt( source, 10, buffer ); // C4996
    cout << "source: " << source << ", string: " << buffer << endl;

    _gcvt( source1, 10, buffer ); // C4996
    cout << "source: " << source1 << ", string: " << buffer << endl;
}
```

例 B-4 的执行效果为:

```
source: 3.14159 , string : 31415927 , decimal: 1 , sign :0  
source: -2.9879 , string : 29879000 , decimal: 1 , sign :1  
source: 3.14159 , string : 3141593 , decimal: 1 , sign :0  
source: -2.9879 , string : 2987900 , decimal: 1 , sign :1  
source: 3.14159 , string : 3.141592654  
source: -2.9879 , string : -2.9879
```



电话服务

服务咨询热线: 010-88361066

读者购书热线: 010-68326294

010-88379203

网络服务

机工官网: www.cmpbook.com

机工官博: weibo.com/cmp1952

金书网: www.golden-book.com

教育服务网: www.cmpedu.com

封面无防伪标均为盗版

为中华崛起传播智慧

地址:北京市百万庄大街22号

邮政编码:100037

策划编辑◎申永刚 / 封面设计◎马精明

标准库能够快速构建健壮程序；如果只选一本高级C++书籍，相信您会毫不犹豫的选择本书，尤其是老手。

C++领域权威，代表技术圈鼎力推荐！
一线C++工程师高水准技术审校！



机械工业出版社微信公众号



机械工业出版社科普平台
科技有的聊



机械工业出版社制造业资讯
制造业那些事儿

上架指导 计算机编程/C++语言

ISBN 978-7-111-51399-5



定价：125.00元