

普通高等教育公共基础课系列教材·计算机类

现代C++ 面向对象程序设计 实验指导

主编◎张 俊 张自力
主审◎张彦铎



科学出版社

普通高等教育公共基础课系列教材·计算机类

现代 C++面向对象程序设计 实验指导

主 编 张 俊 张自力

主 审 张彦铎

科学出版社

北 京

内 容 简 介

面向对象程序设计和 C++ 语言是一门尤其需要上机实践的重要课程,为了更好地适应现代 C++ 技术发展的需要,培养学生较高的程序设计能力和综合应用能力,并配合《现代 C++ 面向对象程序设计》(科学出版社出版,张俊、张自力主编)课程的教学需要,我们编写了本书。

全书包括两大部分,第 1 部分为实验指导,针对每章的能力要求设计了适宜的例题和上机练习题;第 2 部分为 STL 算法与容器参考,旨在为学习 STL 提供方便快捷的参考。全书尽量多地引入 C++11、C++14、C++17、C++20 等新标准的常用技术内容,并展现其应用,为编程能力的训练提供丰富的素材。

本书适合作为计算机科学与技术及相关专业的面向对象程序设计和 C++ 语言课程的教材,也可供编程爱好者自学使用和参考。

图书在版编目(CIP)数据

现代 C++ 面向对象程序设计实验指导/张俊,张自力主编. —北京:科学出版社, 2022.3

(普通高等教育公共基础课系列教材·计算机类)

ISBN 978-7-03-070997-4

I. ①现… II. ①张… ②张… III. ①C++ 语言-程序设计-高等学校-教材 IV. ①TP312.8

中国版本图书馆 CIP 数据核字(2021)第 260950 号

责任编辑:戴 薇 袁星星 / 责任校对:王万红

责任印制:吕春珉 / 封面设计:东方人华平面设计部

科 学 出 版 社 出 版

北京东黄城根北街 16 号

邮政编码:100717

<http://www.sciencep.com>

天津翔远印刷有限公司印刷

科学出版社发行 各地新华书店经销

*

2022 年 3 月第 一 版 开本:787×1092 1/16

2022 年 3 月第一次印刷 印张:14 3/4

字数:349 000

定价:43.00 元

(如有印装质量问题,我社负责调换〈翔远〉)

销售部电话 010-62136230 编辑部电话 010-62135927-2047

版权所有,侵权必究

前 言

自从 C++11、C++14、C++17、C++20、C++23 等新标准发布后，C++ 语言已经进入现代 C++ 阶段，其编程风格和语言特性发生了较大的变化。为了适应技术发展，本书于 2021 年在科学出版社的支持下编写完成，其特点如下。

(1) 按照现代 C++ 的程序设计要求，完善了第 1 部分实验指导中的示例及其源代码，较大幅度地引入各种新的语言特性，也列举了较为丰富的程序设计实例，尽量多地呈现 C++ 新标准的内容。同时按照每章的能力要求设计了适宜的例题和上机练习题，对一些比较综合的上机练习题提供编程提示。

(2) 新标准库引入更丰富的技术内容，简洁地呈现 STL 浩瀚的代码内容以便于学习者理解和上手，本书在第 2 部分整理总结了 STL 算法和容器，编写完善了 STL 标准库参考及其示例程序，以供学习者随时查阅。

本书由张俊、张自力任主编，具体编写分工如下：第 1 部分由张俊编写、第 2 部分由张自力编写，全书由张俊统稿，张彦铎主审。感谢张彦铎、吴云韬、王海晖、胡新荣等领导的关怀和支持，以及吕涛、王邯、田红梅、鲁统伟、赵世平等同事的帮助。

本书总结了编者多年来在 C++ 语言和面向对象程序设计教学实践中的经验，虽经改版，其间修订数次，但由于编者水平有限，书中难免存在疏漏和不足之处，恳请读者批评指正。

编 者

2021 年 9 月

目 录

第 1 部分 实 验 指 导

实验 1 实验环境及其配置	1
实验目的与要求	1
实验过程与内容	1
任务 1 在 VC++ 2019 中开发 C++控制台程序	2
任务 2 在 VC++ 2019 中配置 boost 库	4
典型程序与示例	6
实验题目与提示	9
实验 2 程序调试初步	11
实验目的与要求	11
程序错误与警告	11
任务 1 初识错误与警告	11
任务 2 辨别错误的类型	14
调试工具及应用	18
任务 3 初识调试工具与环境	18
任务 4 基本调试操作	22
实验题目与练习	31
实验 3 C++语言基础	32
实验目的与要求	32
实验过程与示例	32
实验题目与提示	37
实验 4 STL 常用算法与容器	40
实验目的与要求	40
实验过程与示例	40
实验题目与提示	45
实验 5 结构及其应用	48
实验目的与要求	48
实验过程与示例	48
实验题目与提示	54

实验 6 类与对象的定义	58
实验目的与要求	58
实验过程与示例	58
实验题目与提示	66
实验 7 类与对象的几个主题	69
实验目的与要求	69
实验过程与示例	69
实验题目与提示	74
实验 8 运算符重载	76
实验目的与要求	76
实验过程与示例	76
实验题目与提示	91
实验 9 模板	95
实验目的与要求	95
实验过程与示例	95
实验题目与提示	100
实验 10 标准模板库 STL	103
实验目的与要求	103
实验过程与示例	103
实验题目与提示	109
实验 11 继承与派生	111
实验目的与要求	111
实验过程与示例	111
实验题目与提示	115
实验 12 虚函数与多态性	117
实验目的与要求	117
实验过程与示例	117
实验题目与提示	121
实验 13 C++ 的 I/O 流	123
实验目的与要求	123
实验过程与示例	123
实验题目与提示	128

第 2 部分 STL 算法与容器参考

第 1 章 STL 算法参考	131
1.1 头文件<type_traits>中的常用工具函数	131
1.2 头文件<utility>中的常用辅助函数和工具	132
1.3 头文件<functional>中的辅助函数和工具	137
1.4 STL 常用算法	140
1.4.1 不变序列算法	140
1.4.2 可变序列算法	155
1.4.3 去除元素算法	166
1.4.4 序列变序算法	170
1.4.5 序列排序算法	177
1.4.6 已序序列算法	184
1.4.7 数值算法	193
1.4.8 迭代器相关算法	200
第 2 章 STL 容器参考	204
2.1 string 类	204
2.2 vector 类	211
2.3 list 类	213
2.4 deque 类	215
2.5 set/multiset 类	217
2.6 map/multimap 类	220
参考文献	223
附录	224
附录 1 宏 xr 的功能及实现	224
附录 2 函数 print() 的功能及实现	225
附录 3 宏 verify 的功能及实现	226

第 1 部分 实验指导

实验 1 实验环境及其配置

实验目的与要求

1. 实验目的

- (1) 熟悉 C++ 的开发环境及开发过程。
- (2) 了解 C++ 常用标准库的配置。
- (3) 了解控制台程序的基本特点。

2. 实验要求

- (1) 熟练应用 VC++ 2019 集成开发环境，正确完成从建立工程到运行程序的全过程。
- (2) 能够正确配置常用 C++ 库，如 boost 等。
- (3) 能够正确进行控制台程序的输入、输出操作。

实验过程与内容

可以用于 C++ 程序开发的工具和环境很多，其中 VC++ 6.0、VC++ 2005/2008 等都是 Windows 平台上较常用的、主流的集成开发环境 (integrated development environment, IDE)，也是初学者使用较多的 IDE。本书及主教材中的所有程序源代码都是在 VC++ 2019 上完成并测试通过的，其中应用到 VC++ 2019 自带的 STL，以及 boost 库。

开发 C++ 程序时，通常需要 3 个步骤：建立工程；建立并编辑文件；编译、链接、运行程序。以“输入两个数，求和并输出”程序为例（其源代码如例 1.1.1 所示），说明在 VC++ 2019 中开发 C++ 控制台 (console) 程序的基本过程。

【例 1.1.1】输入两个数，求和并输出。

```
#01      #include <iostream>
#02
#03      int main() {
#04          double a, b, c;                                //定义变量
#05
#06          std::cout << "Please enter two floating numbers: ";
```

```
#07      std::cin >> a >> b;                //键盘输入
#08
#09      c = a + b;                          //计算和
#10
#11      std::cout << "The sum of " << a << " and " << b
#12          << " is " << c << "." << std::endl;    //输出结果
#13  }
```

任务 1 在 VC++ 2019 中开发 C++控制台程序

在 VC++ 2019 中, 解决方案的概念类似于 VC++ 6.0 中工作空间的概念。在 VC++ 2019 中开发 C++控制台程序, 首先建立工程和解决方案, 然后建立文件。

1. 建立工程和解决方案

(1) 打开 VC++ 2019 软件, 依次选择 File→New→Project 选项, 或者按 Ctrl+Shift+N 组合键, 弹出 Create a new Project 对话框, 如图 1.1.1 所示。在该对话框中依次选择开发语言、选择平台、选择工程类型, 并确认是 Empty Project, 然后单击 Next 按钮, 退出此对话框。



图 1.1.1 建立工程和解决方案

(2) 在弹出的 Configure your new project 对话框中, 如图 1.1.2 所示, 在 Project name 文本框中输入工程名称, 在 Solution name 文本框中输入解决方案名称, 如 “Hello”; 接着单击 Location 文本框右侧的下拉按钮, 选择一个较浅的工程存放路径, 如 F:\; 然后选中 Place solution and project in the same directory 选项, 让解决方案和工程存放在同一层目录中; 最后单击 Create 按钮, 退出此对话框, 即可成功建立工程及解决方案。

建立工程和解决方案后, 在目录 F:\下会以输入的名称 “Hello” 建立子目录 F:\Hello 作为工程和解决方案的存放位置, 并在其中生成解决方案文件 Hello.sln、Hello.vcxproj 和其他一些文件。

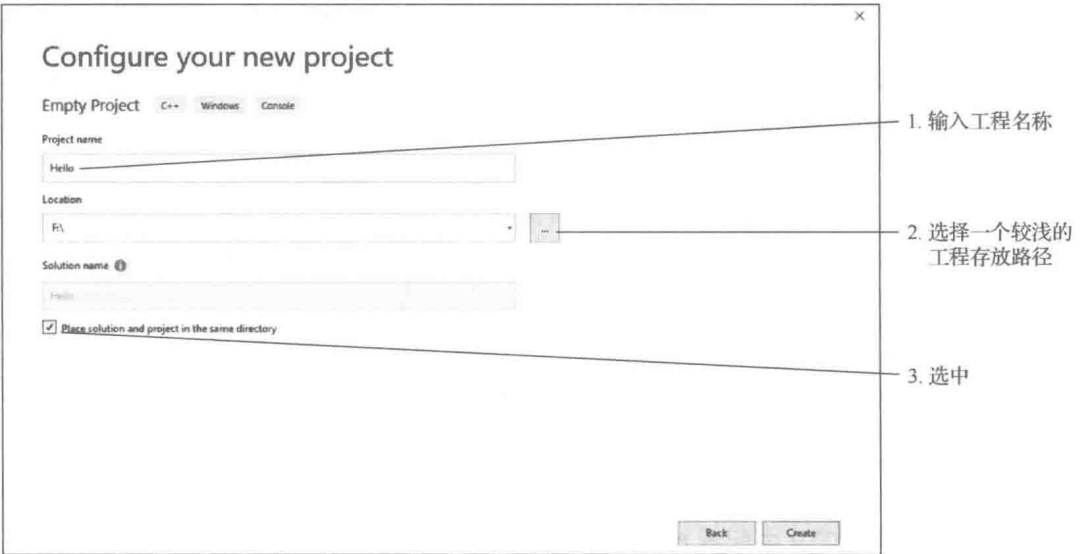


图 1.1.2 设置工程名称和存放路径

2. 建立文件、编辑文件

(1) 向工程中添加文件的方法有 3 种：①在解决方案的 Solution Explorer 窗口中，右击工程名称“Hello”，在弹出的快捷菜单中选择 Add→New Item 选项。②选择 Project→Add New Item 选项。③按 Ctrl+Shift+A 组合键。按这 3 种方法操作都会弹出 Add New Item 对话框，如图 1.1.3 所示。



图 1.1.3 向工程中添加文件

在此对话框中,选择新项类型,选择文件类型,输入文件名称,然后单击 Add 按钮结束文件的创建,退出该对话框,添加文件成功。需要注意的是,这里选择的文件类型是 C++ File (.cpp)即 C++源文件,后续随着程序复杂性不断增加,很多时候也需要建立 Header File (.h),即 C++头文件。

(2) 编辑文件,在 Source 窗口中编辑例 1.1.1 的源程序。VC++ 2019 与 VC++ 6.0 中的快捷键及其他编辑功能相同,但是各种自动提示和智能提示功能更为强大。

3. 编译、链接、运行程序

编译、链接程序文件,可以选择 Build→Build Hello 选项(或按 F7 键)。在程序编译、链接无误后,选择 Debug→Start Without Debugging 选项(或按 Ctrl+F5 组合键)运行生成的可执行程序。

任务 2 在 VC++ 2019 中配置 boost 库

boost 库中有很多强大的功能,极大地扩展了 C++的潜力和功能,其中很多函数和库逐渐纳入 ISO C++,因此它极具活力,吸引越来越多的人对它进行学习、研究和更新。关于 boost 库的版本更新等信息,可以从 boost 主页(www.boost.org)上详细了解。

如果对 boost 库的应用要求不高,如不需要调试或自己编译 boost 代码,则可以从网站(<https://sourceforge.net/projects/boost/files/boost-binaries/>)直接下载已经编译好的 Prebuilt windows binaries,本书编写时采用版本为 boost_1_76.0。该程序可以根据用户的选择直接把源代码、文档和二进制库文件下载安装到本地系统。

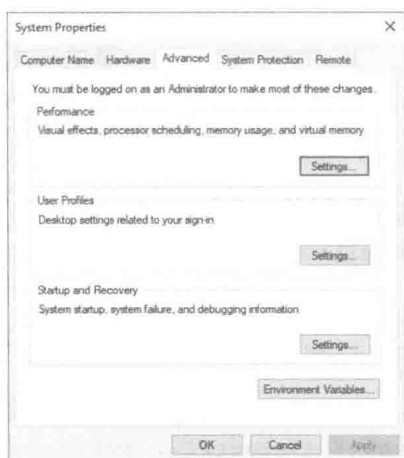
1. 下载安装程序并运行

下载程序 boost_1_76_0-msvc-14.2-32.exe(或者 boost_1_76_0-msvc-14.2-64.exe)并运行,在此过程中根据提示进行相应的选择,待下载完毕后,该安装程序会自动解压缩、编译生成库文件。以下假设这些文件在目录 D:\boost\boost_1_76_0 下。

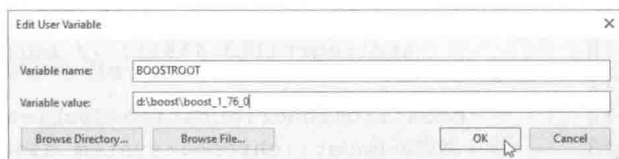
2. 设置路径

在 VC++ 2019 IDE 中设置头文件、库文件的路径。

(1) 在 My Computer 中添加环境变量 BOOSTROOT。右击 My Computer,选择 Properties→Advanced system settings 选项,在弹出的 System Properties 对话框中选择 Advanced 选项卡,单击 Environment Variables 选项,在弹出的 User variables 对话框中单击 New 按钮,弹出 Edit User Variable 对话框,如图 1.1.4 所示。在 Variable name 文本框中输入“BOOSTROOT”,在 Variable value 文本框中输入 boost 库的安装目录“d:\boost\boost_1_76_0”,然后单击 OK 按钮即可成功创建。



(a) System Properties 对话框



(b) 新建用户变量

图 1.1.4 新建环境变量 BOOSTROOT

(2) 在 VC++ 2019 中设置 boost 头文件和库文件的路径。右击新建工程 Hello (注意: 右击工程名称 Hello, 而不是解决方案名称 Solution 'Hello'), 在弹出的快捷菜单中选择 Properties 选项, 弹出 Hello Property Pages 对话框, 如图 1.1.5 所示, 在右侧选择 Include Directories 和 Library Directories 进行头文件和库文件路径的设置。

在 Include Directories 中加入 \$(BOOSTROOT); 在 Library Directories 中加入 \$(BOOSTROOT)\lib32-msvc-14.2。至此, boost 库在 VC++ 2019 中设置完毕。

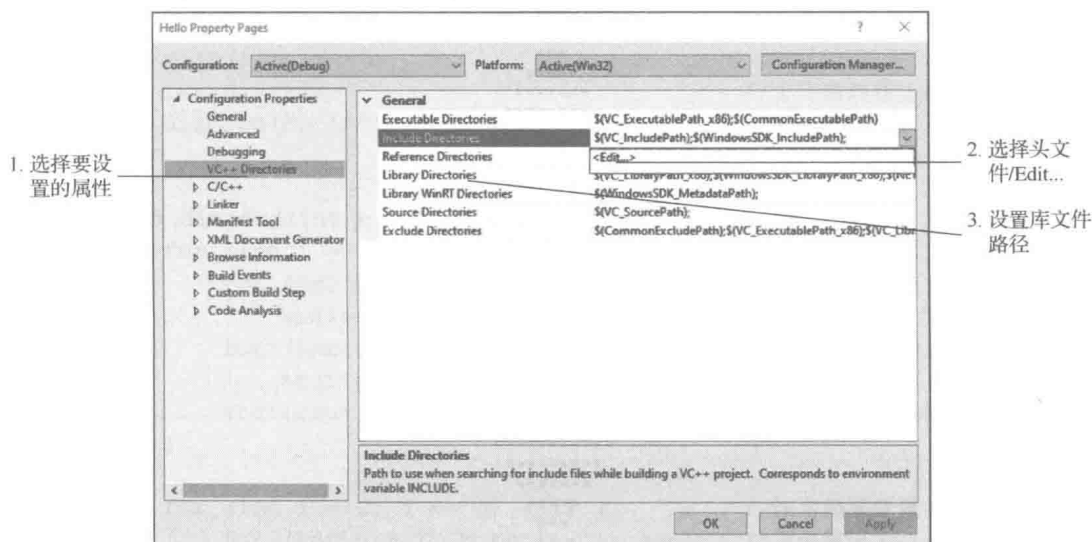


图 1.1.5 设置头文件和库文件路径

3. 测试所配置的环境

行文至此, 用下列程序先试试、感受一下 boost 库带来的新鲜感。

n) 的构成规律为前面有 $n-i$ 个空格, 接着有 $2i-1$ 个星号。故下列程序段会输出有 n 行的上三角形。

```
#01     for (int i = 1; i <= n; ++i) {           //总共输出 n 行
#02         for (int b = 1; b <= n - i; ++b)      //首先输出 n-i 个空格
#03             std::cout << " ";                //注意双引号之间有空格
#04         for (int s = 1; s <= 2 * i - 1; ++s)    //然后输出 2i-1 个星号
#05             std::cout << "*";
#06         std::cout << std::endl;                //本行结束, 换行
#07     }
```

仔细分析下三角形的构成规律: 当下三角形的行数为 n 时, 第 i 行 (i 的取值为 $1 \sim n$) 的构成规律为前面有 $i-1$ 个空格, 接着有 $2(n-i)+1$ 个星号。故下列程序段会输出有 n 行的下三角形。

```
#01     for (int i = 1; i <= n; ++i) {           //总共输出 n 行
#02         for (int b = 1; b <= i - 1; ++b)      //首先输出 i-1 个空格
#03             std::cout << " ";                //注意双引号之间有空格
#04         for (int s = 1; s <= 2 * (n - i) + 1; ++s) //输出 2(n-i)+1 个星号
#05             std::cout << "*";
#06         std::cout << std::endl;                //本行结束, 换行
#07     }
```

把上述两段程序合并到第一段程序的框架中, 则可以输出连续的菱形图案。为了方便应用, 把上述过程封装为函数 `diamond`, 并把上三角形的行数 n 和图案中的填充字符 `cfill` 设为参数。如下是实现菱形输出的完整程序。

```
#01     #include <iostream>
#02
#03     void diamond(int n, char cfill);
#04
#05     int main() {
#06         int n = 14;                             //上三角形有 14 行的菱形
#07         diamond(n, '*');                         //用*填充图案
#08     }
#09
#10     void diamond(int n, char cfill) {
#11         for (int i = 1; i <= n; ++i) {           //总共输出 n 行
#12             for (int b = 1; b <= n - i; ++b)      //首先输出 n-i 个空格
#13                 std::cout << " ";                //注意双引号之间有空格
#14             for (int s = 1; s <= 2 * i - 1; ++s) //然后输出 2i-1 个星号
#15                 std::cout << cfill;
#16             std::cout << std::endl;                //本行结束, 换行
#17         }
#18
#19         for (int i = 2; i <= n; ++i) {           //注意循环变量 i 从 2 开始
#20             for (int b = 1; b <= i - 1; ++b)      //首先输出 i-1 个空格
#21                 std::cout << " ";                //注意双引号之间有空格
#22             for (int s=1; s<=2*(n-i)+1; ++s)      //输出 2(n-i)+1 个星号
#23                 std::cout << cfill;
#24             std::cout << std::endl;                //本行结束, 换行
#25         }
#26     }
```

为了简化上述程序,可以先定义一个函数 `nchars()` 输出 `n` 个字符 `ch`, 函数原型可以如下:

```
void nchars(char ch, int n);
```

这样在每行输出空格和星号时调用该函数两次即可。

【例 1.1.4】猜数游戏。

程序产生一个随机数, 用户从键盘输入所猜测的数据, 程序比较输入数和随机数的大小并分别给以提示, 直到用户猜中所产生的随机数, 或者超过最大允许猜测的次数。

```
#01     #include <iostream>
#02     #include <ctime>
#03
#04     int GetNumber(bool to_print) {           //产生一个随机数
#05         srand(unsigned(time(NULL)));        //初始化随机数的种子
#06         int n = rand() % 90 + 10;            //产生一个[10,99]内的随机数
#07         if (to_print) {                     //若为真,则输出该随机数
#08             std::cout << "The number to guess is " << n << std::endl;
#09         }
#10         return n;                           //返回产生的随机数
#11     }
#12     int Compare(int dest, int guessed) {     //比较两数
#13         if (dest < guessed) {                 //所猜过大,给出提示
#14             std::cout << " ==>Greater than me. Please try again!\n";
#15             return 1;                         //返回 1
#16         }
#17         else if (dest == guessed) {           //猜中,给出提示
#18             std::cout << " ==>Congratulations, you got it!\n";
#19             return 0;                         //返回 0
#20         }
#21         else {                               //所猜过小,给出提示
#22             std::cout << " ==>Less than me. Please try again!\n";
#23             return -1;                        //返回-1
#24         }
#25     }
#26     int main() {
#27         int me = GetNumber(false);           //产生数,不显示
#28
#29         int max_times = 10;                  //最多猜 10 次
#30         int guess_times = 0;                 //已猜次数
#31
#32         for (; ;) {
#33             std::cout << "Please enter one number to match me: ";
#34             int guessed_number;
#35             std::cin >> guessed_number;       //键盘输入一个数
#36             int result = Compare(me, guessed_number); //比较,并给出提示
#37             ++guess_times;                    //增加已猜次数
#38             if (result!=0&&guess_times>=max_times) { //未猜中且超过次数
#39                 std::cout<<"==>Sorry,"<<max_times<<"guess exceeded!\n";
#40                 break;                       //退出循环,不继续猜
```

```

#41      }
#42      else if (result == 0) {                //猜中,显示猜的次数
#43          std::cout<<"==>Great! Just "<<guess_times << " guess!\n";
#44          break;                            //退出循环
#45      }                                    //其他情况继续循环
#46  }
#47  }

```

GetNumber()函数产生随机数,所带参数是一个标志,以决定是否在屏幕上显示所产生的随机数。Compare()函数比较两个数,并力求以生动的语言提示用户两个数的大小信息。

实验题目与提示

1. 记住常用操作的快捷键,能提高编码效率。请熟记表 1.1.1 的快捷键,并经常实践。若无特殊说明,该表中的快捷键可以同时用于 VC++ 6.0 和 VC++ 2005。

表 1.1.1 常用快捷键

	快捷键	功能描述
新建、打开	Ctrl+N	新建新文件/工程, VC++ 2019 中只能新建文件
	Ctrl+Shift+N	新建工程, 只用于 VC++ 2019
	Ctrl+O	打开文件
	Ctrl+F4	关闭当前打开的文件
	Ctrl+F6	在打开的文件之间切换
	Ctrl+S	保存文件
选择、编辑	Ctrl+A	全选
	Ctrl+→/←	以词为单位右移/左移
	Shift+→/←	向右边/左边选择文本
	Ctrl+C	复制选择的内容
	Ctrl+X	剪切选择的内容
	Ctrl+V	粘贴剪贴板内容到光标处
	Ctrl+U	选择的内容全部小写
	Ctrl+Shift+U	选择的内容全部大写
	Ctrl+Z	撤销上次修改
	Ctrl+Y	恢复上次修改
查找、替换	Ctrl+F	查找指定内容
	F3	重新上次查找
	Ctrl+H	文本替换
	Ctrl+G	定位到指定位置
注释、匹配	Ctrl+K, Ctrl+C	代码注释, 只用于 VC++ 2019
	Ctrl+K, Ctrl+U	取消注释, 只用于 VC++ 2019
	Ctrl+]	圆括号/花括号匹配
	Alt+F8	格式化选择的内容

续表

	快捷键	功能描述
运行、调试	Ctrl+F7	编译 (Compile) 文件
	F7	新建工程, VC++ 2019 中新建解决方案
	F4	定位到可能出错的程序行
	Ctrl+F5	运行程序
	F9	切换断点
	F5	在调试状态执行程序
	F10	跳过函数, 逐条执行调试
	F11	进入每个代码块, 逐步执行调试
	Ctrl+F10	运行到光标处

2. 对 VC++ 2019 IDE 进行定制: ①在 Source 窗口左侧显示行号; ②将程序中不同性质的标识符显示为不同的颜色; ③把常用于编译、链接和执行程序的按钮放置在自定义的工具栏上。

【提示】选择 Tools→Options 选项，在弹出的 Options 对话框的 Environment 和 Text Editor 等选项卡中进行设置。

[illegible]

图 1.1.7 空心菱形图案

3. 参考例 1.1.3, 输出空心菱形图案。如图 1.1.7 所示, 该图案外围是一个由星号组成的长方形, 中间是一个空心的菱形。

【提示】同样需要分析图案在每一行的构成规律。对于上半部分（包括最中间一行）的 n 行，第 i 行（ i 的取值为 $1 \sim n$ ）首先要输出 $n-i+1$ 个星号，然后输出 $2(i-1)$ 个空格，最后输出 $n-i+1$ 个星号。对于下半部分（不包括最中间一行）的 $n-1$ 行，第 i 行（ i 的取值为 $2 \sim n$ ）首先要输出 i 个星号，然后输出 $2(n-i)$ 个空格，最后输出 i 个星号。

4. 增强对菱形图案输出的控制：使它能够在控制台窗口，从当前位置向右下任意指定的行数和列数开始输出图案，如当前位置向下 5 行、向右 5 行。

【提示】对于指定的右下 m 行 n 列，首先应用 for 循环输出 m 个'\n'字符以向下 m 行，然后在输出每行图案的 for 循环之前使用一个 for 循环先输出 n 个空格。

5. 参考菱形图案输出的方法, 在控制台窗口, 根据给定的位置、大小输出正方形、长方形、 45° 和 135° 的斜线、正三角形等图案。

【提示】对于不同的图形，分析出每行的构成规律即可。

6. 对于例 1.1.4 中的猜数程序, 可以设计程序最多 7 次猜中该数 (因为范围在[10, 99]), 试实现该过程。

【提示】首先设定两个数 low 和 high 分别表示该随机数所在区间的起点和终点，然后计算 low 和 high 的中点 $med=(low+high)/2$ ，根据 Compare() 函数的返回值决定下一次判断的区间：若返回 1，则说明随机数比 med 小，应在左边区间 [low, med-1] 内猜测；若返回 0，则说明随机数就是 med；若返回 -1，则说明随机数比 med 大，应在右边区间 [med+1, high] 内猜测。使用迭代法或递归方法实现这个过程。由于 $2^6 < 100 < 2^7$ ，因此最多 7 次找到该数。

实验 2 程序调试初步

实验目的与要求

1. 实验目的

- (1) 熟悉 VC++或其他编译环境。
- (2) 掌握 C++程序调试的基本过程。

2. 实验要求

- (1) 了解常用编译环境的设置及其作用，熟练应用 VC++开发环境。
- (2) 掌握基本的调试技巧，能够顺利排除程序错误。

程序错误与警告

任务 1 初识错误与警告

1. 错误与警告的产生

程序错误和警告是每个编程人员都会遇到的，与它们的“斗争”越多，编程人员的编程经验就会越丰富，出错的机会就会越少。因此，排除程序错误和警告是编程人员提高编码能力的重要途径之一。

若程序代码不符合语法规则，或不能够准确地表达意图及需求，或不能够按照正确的逻辑处理问题，则分别会出现语法错误、语义错误和逻辑错误。语法错误较易修改，如关键字拼写错误，标点符号丢失等。只要对语法规则理解透彻并能熟练应用，在编译器的帮助和提示下，总能发现错误根源并全部改正。语义错误较隐蔽，如把判断两个数 a 和 b 是否相等的表达式写为 $a=b$ （正确表达式应为 $a==b$ ），改正这类错误需要编程人员的耐心和经验。逻辑错误可能是最难查找的一类错误了，如实现两个数值的交换时，颠倒了循环赋值语句的顺序，很多时候需要借助调试器来发现这类错误。错误必须要改正，否则不能继续编译生成可执行程序。

若程序错误程度不严重，则编译器提示为警告。程序中存在警告，仍然可以编译链接。有些警告是可以忽略的，如定义了变量而没有后续应用，或者不同类型数据相互赋值而导致精度损失。但是有些警告不能忽略，如对悬挂的指针进行引用操作，见下列程序段：


```
#01      int* p;           //定义指针而未初始化
#02      *p = 3;          //错误:直接对指针去引用
```

编译这个程序段,在 VC++ 2019 中会得到 1 个警告信息“Using uninitialized memory 'p'”(C6001,正在使用未初始化的内存'p')和 1 个错误信息“uninitialized local variable 'p' used”(C4700,局部变量 p 在应用之前没有初始化)。这个错误的存在,使这段程序不能完成编译。

作为一个良好的编程习惯,应该尽量改正程序中提示的所有警告和错误。有些警告信息可以通过编译指令 `#pragma warning` 进行不同级别的处理,这只需要设置不同的说明符即可, `disable` 表示关闭警告信息的显示, `error` 则把警告报告为错误(这提高了警告的级别), `once` 把指定的消息显示一次,如下列指令:

```
#pragma warning( disable : 4700; once : 4385; error : 164 )
```

关闭警告信息 4700,把警告信息 4385 显示一次,把警告信息 4164 报告为错误。

在上述程序段所在 `main()` 函数之前添加如下指令:

```
#pragma warning( disable : 4700)
```

可以完成编译和链接,但在运行程序时发生运行时错误,弹出如图 1.2.1 所示的错误提示框。

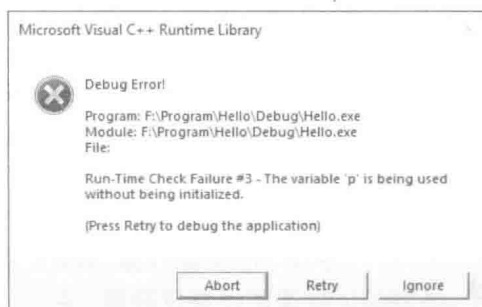


图 1.2.1 程序运行时弹出的错误提示框

很多时候,在程序中需要用到 C-String 处理函数,如 `strlen`、`strcpy` 等,但是编译环境提示这些函数 deprecated,建议更换为安全性更好的函数(Error C4996 'strcpy': This function or variable may be unsafe. Consider using `strcpy_s` instead. To disable deprecation, use `_CRT_SECURE_NO_WARNINGS`.),为了避免这些错误提示,可以在程序前面添加如下指令:

```
#pragma warning( disable : 4996 )
```

取消该错误提示。

以下以 VC++ 2019 这一典型的编译环境为例,讲解该环境下程序调试的基本设置和操作。

2. VC++ 2019 关于错误与警告的设置

为了方便起见,在 VC++ 2019 开发环境中,应开启“集中显示错误信息”(Always show Error List if build finishes with errors)功能,在 VC++ 2019 开发环境中选择 Tools→Options 选项,弹出 Options 对话框,如图 1.2.2 所示,在 Projects and Solutions→

General 选项中, 第一个选项即是, 它已经默认开启。这会在编译和链接之后把所有错误信息集中显示在屏幕下方的 Error List (错误列表) 窗口中, 如图 1.2.3 (a) 所示, 显示信息包括错误描述 (错误代码和提示信息)、错误所在的文件和行号。若在编译和链接阶段没有任何错误产生, 则编译成功的所有信息显示在屏幕下方的 Output (输出) 窗口中, 如图 1.2.3 (b) 所示。

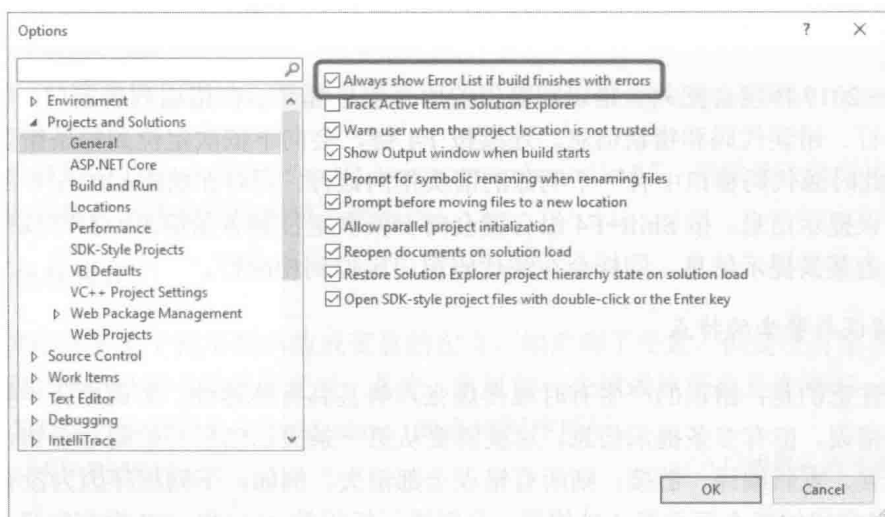
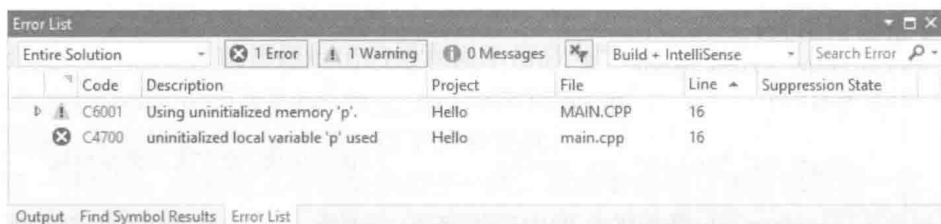
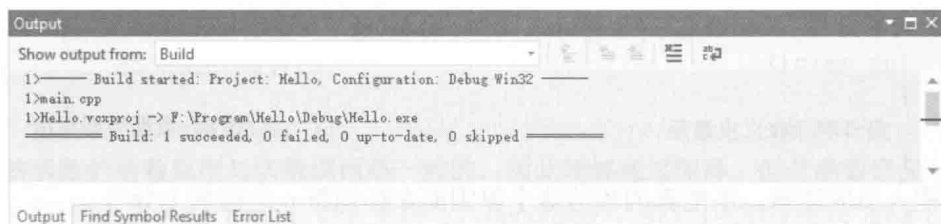


图 1.2.2 Options 对话框



(a) 有编译错误时的显示



(b) 编译成功时的显示

图 1.2.3 不同编译结果的显示

另外, 也可以在工程设置中设置对待警告的方式。在 Solution Explorer 窗口中, 右击工程名称, 在弹出的快捷菜单中选择 Properties 选项, 在弹出的 Property Pages 对话框中, 选择 Configuration Properties → C/C++ 标签, 可以在右侧选项中设置警告的级别 (Warning Level), 或者把警告报告为错误 (Treat Warnings As Errors)。

有时为了避免生成 Manifest File (清单文件) 产生错误, 可以关闭清单文件的产生, 方法为: 打开 Property Pages 对话框 (工程的属性页), 选择 Configuration Properties → Linker → Manifest File 选项, 把右侧的选项 Generate Manifest 设置为 No。同时选择 Configuration Properties → Manifest Tool 标签下的 Input and Output 选项, 把右侧的选项 Embed Manifest 设置为 No。

3. 错误和警告的查看

VC++ 2019 环境会把关于错误和警告的所有信息都显示在错误列表窗口, 包括文件名、所在行、错误代码和错误信息。连续按 F4 键, 会向下依次定位到每条错误提示所在的行, 此时源代码窗口中有一个明显的箭头指向该行, 同时在状态栏的左半部分会着重显示错误提示信息。按 Shift+F4 组合键会向上依次定位到各条信息, 也可以在错误列表窗口双击某条提示信息, 同样会在源代码窗口定位到相应行。

4. 错误与警告的特点

需要注意的是, 错误的产生有时显得虚张声势且具有迷惑性。所谓虚张声势, 有时只有一处错误, 但有多条提示信息, 这就需要从第一条提示信息开始耐心查错, 找到了这一处错误, 重新编译、链接, 则所有错误全部消失。例如, 下列程序因为没有书写引入 std 命名空间的指令而引发 4 处错误, 分别提示标识符 cout 和 endl 没有定义 (其中之一为 error C2065: 'cout': undeclared identifier), 加上该指令 using namespace std;, 重新编译, 则 4 条错误消失。

```
#01    #include <iostream>
#02    int main()
#03    {
#04        cout << "Hello" << endl;           //访问出错:没有引入所在空间
#05    }
```

所谓具有迷惑性, 是指错误提示信息有时不能准确地反映错误产生的真正原因。例如, 下列程序因为把类型标识符 int 错误地书写为 Int, 而导致错误提示信息 “在标识符 n 之前丢失了分号” (error C2146: syntax error : missing ';' before identifier 'n')。

```
#01    int main()
#02    {
#03        Int n;                               //错误:数据类型写错
#04    }
```

编程人员应注意, 不能真的相信这个原因而在标识符 n 之前加上分号。

任务 2 辨别错误的类型

对应于程序生成的 3 个阶段 (编译、链接、运行), 可能发生的错误有 4 种类型:

①编译错误, 以编译期间出现的语法错误为主。②链接错误, 在建立可执行文件的链接处理过程中发生的错误。③运行时错误, 程序在运行期间发生的错误。④逻辑错误 (包括语义错误), 程序运行时并无错误, 但产生的结果不正确或不能达到预期的结果。

1. 编译错误

这类错误往往是因为程序中出现了违反语法规则的代码。根据错误列表中的提示信息和所给出的行号,经过细心的分析一般较容易找出错误的原因。例如,下列程序中的流操纵算子 `endl` 被错误地拼写为 `end1` (即把字母 `l` 写成了数字 `1`)。

```
#01      #include <iostream>
#02      int main()
#03      {
#04          std::cout << "Hello" << std::end1; //拼写错误:字母 l 写成数字 1
#05      }
```

错误提示信息为“error C2039:'end1': is not a member of 'std'”。初学者往往受困于语法错误,这主要是对语法规则的理解不够深入。

2. 链接错误

链接错误常见于找不到函数或变量的定义,如声明了变量,但是没有定义;书写了函数原型,但是没有书写函数定义。其中,常见的一个错误例子就是在书写 `main()` 函数时,把该函数名称错误地拼写为 `mian`,如下列程序所示。

```
#01      int mian()                                //入口函数名拼写错误
#02      {
#03          int n = 1;
#04          return n;
#05      }
```

出现上述错误时,编译器给出的错误提示信息为“error LNK2019: unresolved external symbol `_main` referenced in function `__tmainCRTStartup`”。

程序中典型的链接错误主要有以下两种情况。

第一种情况:函数定义与函数原型不一致,而函数原型与函数调用相一致,因此对此函数的链接将会发生错误。例如,下列程序中定义的函数 `do_nothing`。

```
#01      void do_nothing();
#02      int main()
#03      {
#04          do_nothing();
#05      }
#06      void donothing() {}                        //函数定义前后不一致
```

该函数的函数原型与函数调用是一致的,因此能够通过编译。但是函数定义与函数调用不一致(把函数名 `do_nothing` 错误地拼写为 `donothing`),因而出现链接错误“error LNK2019: unresolved external symbol "void __cdecl do_nothing(void)" (?do_nothing@@YAXXZ) referenced in function `_main`”,其中标识符“`?do_nothing@@YAXXZ`”是按照调用规范对函数名进行修饰之后的结果。

第二种情况:类中 `static` 数据成员没有在类体外定义,对该 `static` 数据成员的访问将会引发链接错误,如下列程序所示。

```
#01      class Test {
#02      private:
```

```

#03         static int s;
#04         int d;
#05     public:
#06         Test(int a) : d(a) {s = d;}
#07     };
#08     //int Test::s = 0;           //静态数据成员没有定义（或初始化）
#09     int main() {
#10         Test t(1);
#11     }

```

上述程序在类 `Test` 中声明了 `static` 数据成员 `s`。只有对 `static` 数据成员进行了定义（或初始化），它才能被对象所用，对 `static` 数据成员的定义（或初始化）应该在类定义体外（而不是构造函数中）进行，显然上述程序没有做到这些。因此在 `main()` 函数中生成对象 `t` 而需要在构造函数中访问 `s` 时出现链接错误“error LNK2001: unresolved external symbol "private: static int Test::s" (?s@Test@@@0HA)”。为了改正这个错误，可以在第 8 行添加代码“`int Test::s = 0;`”实现 `static` 数据成员 `s` 的初始化。

3. 运行时错误

常见的运行时错误有被 0 除、溢出、下标越界、内存访问错误等。这类错误在编译、链接期间不会发生，而一直隐藏到运行期间才显露。例如，下列程序引发除 0 错误。

```

#01     int main()
#02     {
#03         int a = 2, b = 0;
#04         int c = a / b;           //除 0 错误
#05     }

```

该程序因为除以 0 而导致溢出错误。需要说明的是，对于除 0 错误，VC++ 2019 不再像之前版本的 VC++ 弹出如图 1.2.4 (a) 所示的提示框。

又如，下列程序因为下标越界而非法访问内存地址。

```

#01     int main() {
#02         int a[10], n = 10;
#03
#04         for (int i = 0; i <= n; ++i)           //下标越界:i 达到 n
#05             a[i] = i;
#06     }

```

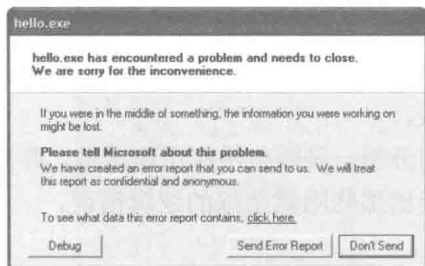
当 `i` 取值为 `n` 时，表达式 `a[i]` 访问到不存在的数组单元而引发内存访问错误，此时弹出的错误提示对话框如图 1.2.4 (b) 所示。会引发同样错误的还有如下程序：

```

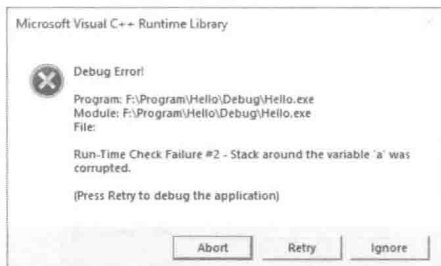
#01     #include <string.h>
#02     int main() {
#03         char buffer[] = "abc";
#04         strcpy(buffer, "Hello");           //内存访问错误
#05     }

```

字符数组 `buffer` 在初始化为字符串“abc”时把长度定义为 4，随后通过函数 `strcpy()` 的字符数组复制也会因为源字符串“Hello”过长而引发同样的内存访问错误，弹出类似于图 1.2.4 (b) 所示的错误提示对话框。



(a) 除 0 错 (非 VC++2019 窗口)



(b) 下标越界或内存访问出错

图 1.2.4 运行时错误

作为一个设计良好的系统,应该能够对这些可能发生的错误进行检测并进行异常处理,以便能够从错误中恢复。对于比较严重的错误,则应该果断中止其执行。

4. 逻辑错误

逻辑错误主要由程序的语义功能、事务逻辑错误导致。对于这类错误的排除往往需要借助调试器进行细致的分析。例如,下列程序段在累加从 1 到 100 之间的偶数时出现错误。

```
#01      int s = 0;
#02      for (int i = 1; i <= 100; ++i) {
#03          if (i / 2 == 0)                //判断偶数出错
#04              s += i;
#05      }
```

原因在于对偶数的判断准则出错:不是 $i/2 == 0$, 而是 $i \% 2 == 0$ 。

同样,下列程序在查找数组 `a` 中值为 3 的元素时出错。

```
#01      #include <iostream>
#02
#03      int main() {
#04          int a[] = {1, 2, 3, 4, 5, 6};
#05          int n = sizeof(a) / sizeof(*a);
#06
#07          int k = 3;
#08          for (int i = 0; i < n; ++i) {
#09              if (a[i] = k)                //运算符写错
#10                  std::cout << "find " << k << " at " << i << std::endl;
#11          }
#12          std::cout << "not found." << std::endl;
#13      }
```

如下是程序的输出结果。

```
find 3 at 0
find 3 at 1
find 3 at 2
find 3 at 3
find 3 at 4
```

```
find 3 at 5
not found.
```

这前后矛盾的输出结果绝非外行所为。仔细分析,发现错误的根源在于数组元素比较时用错了运算符:不是 $a[i] = k$, 而是 $a[i] == k$ 。

排除逻辑错误需要的是无限的细心和缜密的分析。只要善于应用调试器,并采取一些必要方法,如输出关键变量的取值,总能够查出那些隐藏至深的逻辑错误。

调试工具及应用

任务 3 初识调试工具与环境

VC++ 2019 的调试器 Debugger 具有单步执行的功能。在单步执行状态下,编程人员控制程序执行的过程,每次执行一行代码,然后把执行该行代码之后各变量、对象的状态和取值显示在不同的窗口,以供用户分析这些取值和状态的变化是否正确,从而找出其中错误的代码行。同时,该调试器支持 Edit and Continue 功能,即在调试时修改程序,然后继续调试过程而无须重新开始。

1. 调试器

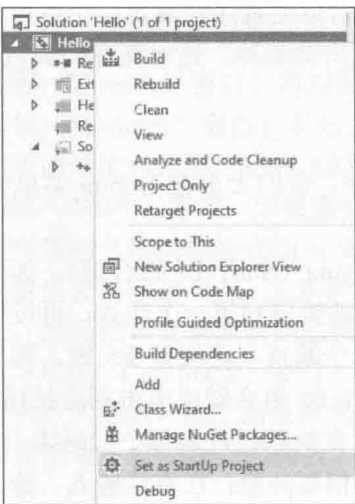
启动调试器进入调试状态,需要具备几个前提条件:①需要调试的工程在解决方案中是活动工程 Active Project (或启动工程 Startup Project);②需要调试的程序是调试版本;③程序没有编译、链接错误。

先看第一个条件,把当前工程设为活动工程(或启动工程)。若整个 Solution 中只有一个工程,则它已经被设为启动工程(工程名加黑显示)。若工作空间中有多于一个的工程,则可以在 Solution Explorer 窗口中右击工程名,在弹出的快捷菜单中选择 Set as Startup Project 选项,如图 1.2.5 (a) 所示。或者在 Solution Explorer 窗口单击工程名,然后选择 Project→Set as Startup Project 选项,如图 1.2.5 (b) 所示。

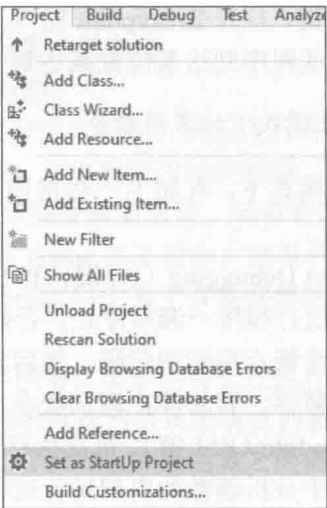
接着满足第二个条件,选择应用程序的调试版本。在 VC++ 环境中建立应用程序时,可以选择建立该程序的调试版本(Debug)或发行版本(Release),默认情况下会建立调试版本。调试版本会使编译器在目标文件中加入帮助调试的符号信息,这些信息存储在 Debug 目录下的.pdb 文件中。发布版本则没有这些信息。在 VC++ 2019 开发环境中,当前有效的解决方案配置显示在 Standard 工具栏上两个相邻的下拉列表中,如图 1.2.6 所示,在左侧下拉列表中选择程序的版本(默认为 Debug),在右侧下拉列表中选择程序的平台(Win32)。这也可以通过工程属性对话框来设置。

最后,为了进入调试状态,还需要确保程序没有编译、链接错误,即排除所有的语法错误。

至此,可以启动调试器。选择 Debug→Start Debugging 选项,或者单击 Standard 工具栏中的 Start Debugging 按钮,或者直接按 F5 键,都可以启动调试器,使程序进入调试模式。此时,出现 Debug 工具栏及常用调试命令按钮,如图 1.2.7 所示。



(a) 方法 1



(b) 方法 2

图 1.2.5 设置活动工程（或启动工程）

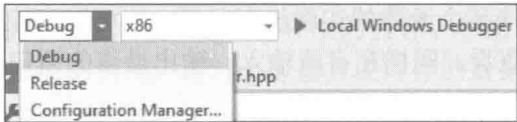


图 1.2.6 选择调试版本



图 1.2.7 常用调试命令按钮及工具栏

调试器有两种工作模式：单步执行和运行到指定位置后暂停。按 F5 键进入调试状态，程序很快自动完整执行一遍就结束，无法进入单步执行状态，也无法从某一个指定点开始执行，这需要设置断点（breakpoint）。

2. 断点

所谓断点，是让程序暂停的位置。在断点之前的程序代码都会被自动执行，自断点所在的程序行开始可以单步执行或自动执行。通常可以从有疑虑的代码行开始设置断点，断点的个数不限。

设置断点的快捷键为 F9。将鼠标指针定位到某程序行，按 F9 键，可见该行左侧有一个红色的实心圆，表示设置断点成功，再按一次 F9 键，该红色实心圆消失。因此，F9 键可实现断点的切换，即设置和清除。为了清除所有断点，可以选择 Debug→Delete All Breakpoints 选项，或者按 Ctrl+Shift+F9 组合键。

程序的执行不能在某个语句中间停止，而只能暂停在一个完整的语句之后。因此，断点不能设置在空行，系统会自动把该断点设置在空行的下一行。

为了设置更高级的断点，可以选择 Debug→Windows→Breakpoints 选项，或者按

Alt+F9 组合键, 打开 Breakpoint 窗口。高级断点可以使程序在满足某个条件之后自动暂停, 也可以使程序到达某位置或达到一定次数后自动暂停, 这对循环的调试很有用。

3. 调试模式下的常用命令

在调试模式下, 有如下一些常用的调试命令, 它们大多在 Debug 菜单中或在工具栏上。

(1) Start Debugging (F5 键) 和 Stop Debugging (Shift+F5 组合键)。如果程序没有断点, 完整运行程序一遍后停止。否则, 程序自动暂停到第一个断点, 再按一次 F5 键, 自动执行两个断点中间的代码, 然后暂停在第二个断点。连续按 F5 键, 程序在断点之间执行而后暂停。若想停止调试状态, 可按 Shift+F5 组合键或单击相应按钮。

(2) Step Into (F11 键)。每次执行一条语句, 在执行时进入每个代码块 (包括函数)。这个命令对于分析函数的内部运行过程很重要。但要注意: 在有流输入、输出操作的语句中 (如 `cout<<`, `cin>>`) 不要应用这个命令, 这会导致程序进入实现流输入、输出操作的库函数中, 用户往往对这些库函数不太感兴趣, 而且它们一般是没有问题的。

(3) Step Over (F10 键)。每次执行一条语句, 但是不会进入函数块。这是与 Step Into 相区别的地方。如果确信某个函数的实现没有问题, 则可以在该函数的调用语句处应用 F10 键跳过对该函数的查看。同样在有流输入、输出操作的语句中也应该使用 F10 键单步执行。

(4) Run to Cursor (Ctrl+F10 组合键)。这个命令没有出现在 VC++ 2019 的 Debug 菜单中, 但它是一个非常有用的方式。将鼠标指针定位到某个程序行, 按 Ctrl+F10 组合键, 则程序暂停到该程序行, 如同该行有一个断点。

需要注意的是, 无论使用哪种方式启动调试, 都可以用其他方式继续程序的执行和暂停。但是在源代码窗口中的程序都执行完毕之后, 一般要按 F5 键自动执行完程序收尾时的清理代码。

4. 调试器窗口

在调试状态下, 可以把鼠标指针直接放在某个变量上面停留 1s, 其取值会被显示, 这是查看变量取值最快捷的方法。此外, 在调试状态下还可以自动显示 (或通过菜单打开) 各种数据窗口和诊断工具, 对程序运行过程中的所有状态和各种数据进行详尽分析。例如, 通过 Debug→Windows 选项打开各种 Debugger 窗口 (Breakpoints、Output、Watch、Autos、Locals、Immediate、Call Stack、Threads、Modules、Processes、Memory、Disassembly、Registers 等), 这些窗口依次列显在屏幕下方; 通过 Debug→Windows→Show Diagnostic Tools 选项或按 Ctrl+Alt+F2 组合键可以打开各种诊断工具 Diagnostic Tools (Events、Memory Usage、CPU Usage 等), 这些工具列显在屏幕右方。各个数据窗口和诊断工具的显示内容或其功能简述如下。更详细的介绍可参考相关技术手册。

(1) 自动变量 (Autos) 窗口。显示当前状态下变量及其取值, 这些变量一般位于源代码窗口中黄色箭头所在的程序行及其前一行。该窗口自动显示的整数值可以设置为十进制或十六进制, 通过该窗口的右键菜单可以完成。

(2) 局部变量 (Locals) 窗口。显示当前函数中局部变量的取值, 随着程序流程进入不同的作用域, 该窗口显示在当前作用域中的变量及其取值。

(3) 线程 (Threads) 窗口。用以显示并控制线程。

(4) 模块 (Modules) 窗口。显示当前正在执行的代码块, 若发生程序崩溃, 通过比较崩溃发生的地址和该窗口所列 Address 的范围, 可以确定出错的模块。

(5) 监视 (Watch) 窗口。如果需要长期监视某些变量的取值, 或者表达式的取值, 就可以将该变量的名称或表达式输入 Watch 窗口的 Name 文本框中, 随着程序的执行, 该变量的取值也会显示在 Value 文本框中。为了监视代码中某个变量取值的变化, 可以选中该变量, 然后把该变量拖放到监视窗口。

(6) 调用堆栈 (Call Stack) 窗口。函数调用堆栈窗口显示当前正在执行的函数及在堆栈中等待执行的函数, 最上面被黄色箭头指示的函数是当前正在执行的函数, 下一个就是退出当前函数后将要执行的函数。Call Stack 窗口对于分析函数之间的嵌套调用非常有用。

(7) 断点 (Breakpoints) 窗口。显现当前程序中的所有断点, 通过该窗口, 可以修改现有断点, 也可以设置新断点。

(8) 输出 (Output) 窗口。显示当前程序所加载各个模块的信息。

(9) 内存 (Memory) 窗口。显示当前内存中的内容。利用 Memory 窗口可以看到应用程序所占用的内存空间的情况, 这对于检查大块内存的数据 (如缓冲区和大的字符串) 很有利。

(10) 反汇编 (Disassembly) 窗口。显示当前程序每条语句对应的汇编代码。

(11) 寄存器 (Registers) 窗口。显示处理器的寄存器状态。

(12) 诊断工具之事件查看器 (Events)。可以通过事件列表 (如模块加载、线程启动和系统配置) 查看应用程序的活动, 以帮助更好地诊断应用程序在 Visual Studio 探查器中的执行情况。该工具在列表视图中显示每个事件的信息, 如事件名称、时间戳和进程 ID 等。

(13) 诊断工具之内存使用情况查看器 (Memory Usage)。评估应用程序中的内存使用情况, 如堆上变量和对象的数量和大小。通常, 分析内存的最好方法是拍摄两张快照 Snapshot; 一张正好拍摄于发生可疑内存问题之前, 另一张拍摄于发生可疑内存问题之后。然后查看两张快照的差异, 发现实际更改的内容。

(14) 诊断工具之 CPU 使用率查看器 (CPU Usage)。分析应用程序的性能, 评估函数本身是否属于性能瓶颈。CPU 使用率视图按运行时间排序显示函数列表, 运行时间越长, 函数排在越前面。双击感兴趣的函数, 将会看到更加详细的调用和被调用函数的时间信息。

记住一些常用的调试快捷键往往会取得事半功倍的效果。如表 1.2.1 所示是常用调试命令的快捷键。

表 1.2.1 常用调试命令的快捷键

调试命令	快捷键	功能
Start Debugging	F5	自动执行/继续
Toggle Breakpoint	F9	切换断点
Restart	Ctrl+Shift+F5	重启调试
Stopping Debugging	Shift+F5	停止调试
Step Into	F11	单步执行（进入模块）
Step Over	F10	单步执行（跳过模块）
Step Out	Shift+F11	单步执行（跳出模块）
Run to Cursor	Ctrl+F10	运行到光标处

任务 4 基本调试操作

1. 程序运行分析

【例 1.2.1】查看变量取值。

首先建立一个 VC++ 2019 Win32 Console 工程，然后把下列程序输入编辑器，并编译链接。

```
#01      #include <iostream>
#02
#03      int main() {                                //输出变量的正负号
#04          double x = -3;
#05          int sign;
#06
#07          if (x > 0)                                //正数
#08              sign = 1;                            //符号为 1
#09          else if (x == 0)                          //零
#10              sign = 0;                            //符号为 0
#11          else                                      //负数
#12              sign = -1;                            //符号为-1
#13
#14          std::cout << "sign of " << x << " is " << sign << std::endl;
#15      }
```

为了查看上述程序的执行过程，以及各变量值的变化，下面开始单步执行上述程序。

第一步：启动调试器。在第 3 行（main()所在代码行）按 F9 键插入一个断点，然后按 F5 键自动执行到断点所在行。选择在 main()所在行插入断点的目的是观察程序执行的全部过程，因此让程序从进入 main()之前就开始单步执行。断点也可以设在第 4 行（main()函数体中的第 1 行）。

调试器启动后，调试模式下的界面如图 1.2.8 所示。在编辑窗口，一个黄色的箭头指示在第一个断点处。

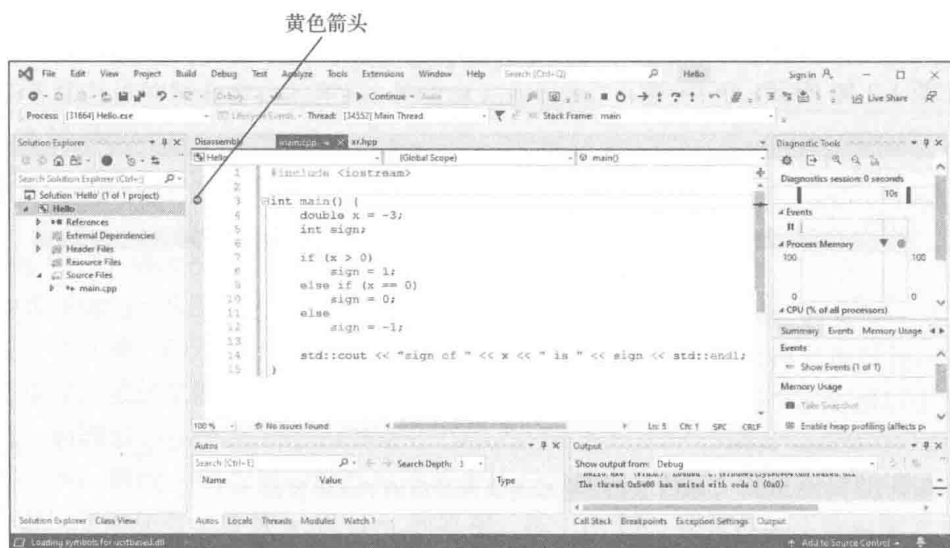


图 1.2.8 调试器启动后的界面

第二步：按 F10 键单步执行，观察变量的初始化情况。接着按 F10 键，黄色箭头向下移动到第 4 行。需要注意的是，在任何时刻，黄色箭头所指的代码行是即将开始执行的代码行。

当前执行的程序行是给变量 x 初始化的语句“double x=-3;”，可以看到，Autos 窗口中显示了变量 x，由于还没有执行初始化语句，因此 x 中当前的值是未被初始化的垃圾值。

继续按 F10 键，程序对变量 x 初始化完毕，同时完成变量 sign 的定义，因此直接跳到第 7 行 (if 语句) 开始执行。如图 1.2.9 所示，此时 Autos 窗口中显示 x 初始化后的值，而 sign 没有初始化，显示为垃圾值。

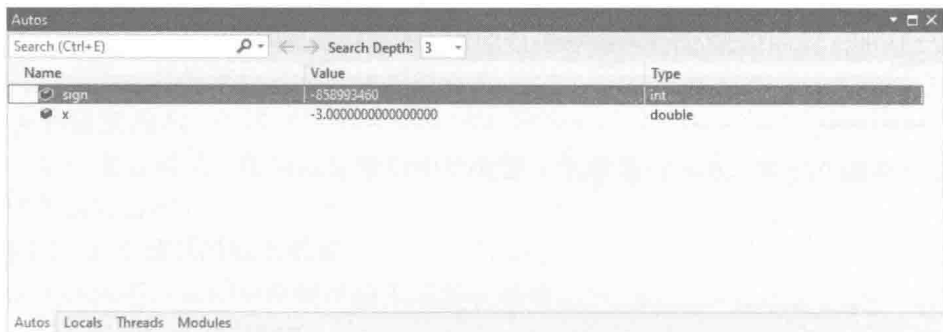


图 1.2.9 完成变量初始化后的 Autos 窗口

第三步：继续单步执行，观察逻辑判断的执行情况。黄色箭头当前所指为 if 语句，由于变量 x 的值为-3，显然不满足 if 语句中的条件 $x > 0$ ，因此按 F10 键查看程序判断情况。果然，程序没有执行满足条件 $x > 0$ 后的语句“sign=1;”，而是继续向下判断 else if 中的条件，显然不满足条件 $x = 0$ 。接着按 F10 键，语句 else 包括了对于 $x < 0$ 的判断，

程序在此满足 `else` 的条件，因此应该执行语句“`sign=-1;`”。再次按 F10 键，驱使黄色箭头指到第 14 行，可以看到程序的确执行了该语句，变量 `sign` 被赋值为 -1。此时，Autos 窗口如图 1.2.10 所示。

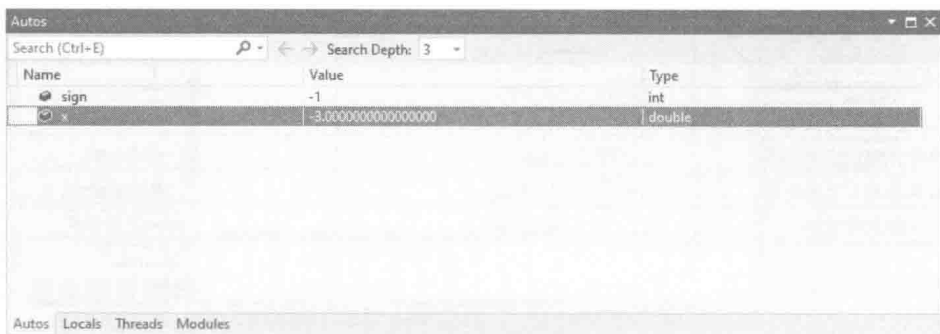


图 1.2.10 完成逻辑判断后的 Autos 窗口

第四步：重新再来，添加对变量 `x` 的监视。细心的读者可能会看到，在上述第三步中，当黄色箭头指到第 7 行时，接着按两次 F10 键，当黄色箭头指到第 12 行时，Autos 窗口中关于变量 `x` 及其值的显示消失了。这是因为，在当前语句中，没有引用变量 `x`，因此 Autos 窗口不会显示当前没有用到的变量。在此之前，即黄色箭头指到 `if` 语句和 `else if` 语句时，这两个语句中都包括对变量 `x` 和 `sign` 的引用，因此在 Autos 窗口中会显示这两个变量。

为了能够一直观察变量 `x`，需要把它添加到 Watch 窗口。单击 Watch 窗口的 Watch1 标签中 Name 列的第一行，输入 `x`，然后按 Enter 键，这时在 Value 列就会显示其值。当变量的名称太长，如此输入可能会发生拼写错误，因此可以选中变量的名称，然后拖放到 Name 列的空行中。使用这个方法把变量 `sign` 也添加到 Watch 窗口，此时（程序完成了所有 `if/else if` 语句的判断，黄色箭头指到第 14 行）Watch 窗口如图 1.2.11 所示。

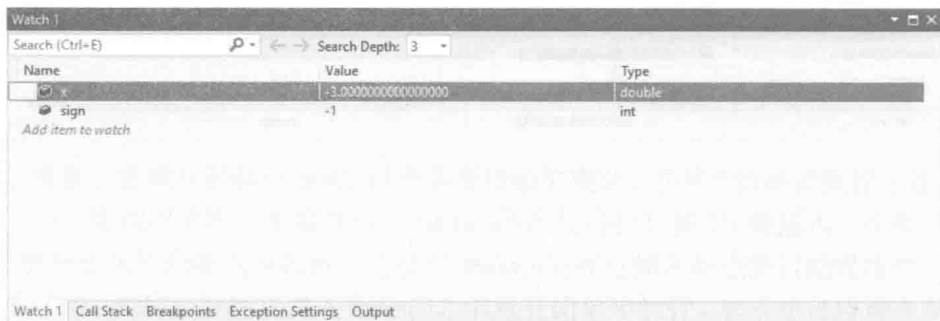


图 1.2.11 添加监视后的 Watch 窗口

请读者自行试验快速查看变量取值的方法：将鼠标指针停放在某个变量上至少 1s，调试器会自动显示该变量的值。

第五步：继续单步执行，完成程序过程。注意，此时黄色箭头指在最后一行输出语句上。按 F10 键，程序执行输出。需要注意，在按 F5 键启动调试模式后，一个控制台窗口会自动弹出，并隐藏在后台，但是在任务栏上有其显示。这个控制台窗口是显示程

序输出、提供键盘输入的场所。因此，在执行完上述输出语句后，该语句的显示就写在了该控制台窗口，单击任务栏上该控制台窗口的图标，把它显示在前台，如图 1.2.12 所示。

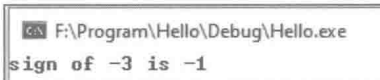


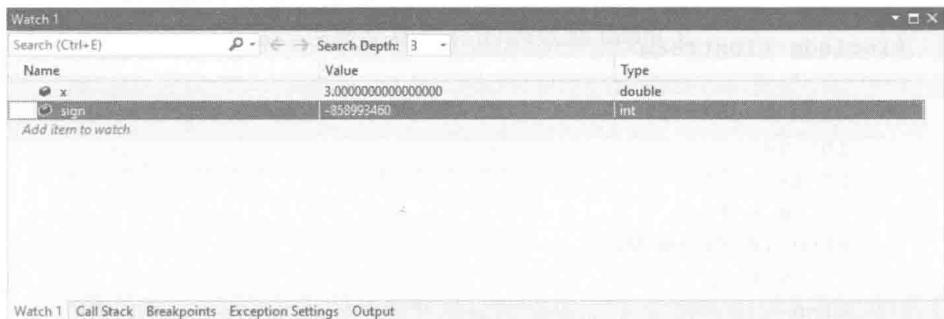
图 1.2.12 程序输出

至此（黄色箭头定位到最后一行的右花括号），按 F5 键结束程序单步执行，程序执行完毕，注意不要继续按 F10 键。

【例 1.2.2】修改变量取值。

接着考虑这个问题，在上述调试过程中，我们只是从头到尾对值为负数（-3）的变量 x 进行了计算，而对于 x 小于 0 和等于 0 的情况没有看到计算其符号的过程。其实这很容易实现。虽然在程序中固定了 x 的取值为 -3，但是我们可以在 Watch 窗口中任意修改变量 x 的取值。

第一步：修改 x 的取值，观察程序单步执行的情况。按 F5 键重新启动调试器，在 Watch 窗口中添加对于变量 x 和 $sign$ 的监视，按 F10 键，让黄色箭头指向第 7 行的 if 语句。双击 Watch 窗口中 Watch1 标签变量 x 所在行的 Value 列，把其中的 -3 改为 3，然后按 Enter 键，此时 Watch 窗口如图 1.2.13 所示。

图 1.2.13 修改 x 的值为 3 之后的 Watch 窗口

由于 x 的取值变为 3，满足条件 $x > 0$ ，因此程序应该执行 if 语句中的 $sign=1$ 。按 F10 键验证这个判断，果然黄色箭头定位到第 8 行的 “ $sign=1$;”，接着按 F10 键，可以看到变量 $sign$ 的值变为 1。

第二步：重启调试，在 Watch 窗口中把变量 x 的取值改为 0，按 F10 键单步执行，分析程序的执行过程。

【例 1.2.3】在调试时输入数据。

修改上述程序，请用户从键盘输入变量 x 的值。

```
#01    #include <iostream>
#02
#03    int main() {
#04        double x;
#05        std::cout << "Please enter a number: ";
#06        std::cin >> x;
#07
#08        int sign;
```

```

#09         if (x > 0)
#10             sign = 1;
#11         else if (x == 0)
#12             sign = 0;
#13         else
#14             sign = -1;
#15
#16         std::cout << "sign of " << x << " is " << sign << std::endl;
#17     }

```

第一步：按 F9 键在第 3 行 `main()` 所在行插入一个断点，按 F5 键启动调试器，注意一个控制台窗口自动弹出，其图标显示在任务栏上。

第二步：按两次 F10 键，`cout` 语句的输出信息显示在控制台窗口，再按 F10 键，此时调试器由 Debugging 状态变为 Running 状态，控制台窗口显示在前台，在该窗口中输入 3，然后按 Enter 键，此时调试器由 Running 状态变回 Debugging 状态，控制台窗口显示在后台，Autos 窗口中 `x` 的值为 3。

第三步：一直按 F10 键，完成程序的单步执行，最后按 F5 键结束程序调试状态。

【例 1.2.4】查看函数调用。

修改上述程序，把计算符号值的过程封装在函数 `sign()` 中。

```

#01     #include <iostream>
#02
#03     int sign(double x) {                //返回 x 的符号
#04         int s;
#05         if (x > 0)
#06             s = 1;
#07         else if (x == 0)
#08             s = 0;
#09         else
#10             s = -1;
#11
#12         return s;
#13     }
#14
#15     int main() {
#16         double d;
#17         std::cout << "Please enter a number: ";
#18         std::cin >> d;
#19
#20         std::cout << "sign of " << d << " is " << sign(d) << std::endl;
#21     }

```

第一步：按 F9 键在第 15 行 `main()` 所在行插入一个断点，按 F5 键启动调试器，按 F10 键单步执行，并从控制台窗口输入变量 `d` 的值为 3，然后按 Enter 键，此时黄色箭头指到第 20 行 `cout` 所在代码行。

第二步：由于在此代码行有对函数 `sign()` 调用的表达式 `sign(d)`，如果我们很清楚该函数执行的过程，并知道执行该函数后的结果，则可以跳过该函数的执行过程，并按 F5 键结束程序调试。但是，现在希望进入该函数内部查看该函数执行的过程。因此在

第 20 行按 F11 键，此时黄色箭头跳转到函数 `sign()` 定义体所在行（第 3 行）。

此时，从 Autos 窗口可以看出，经过参数传递后，实参 `d` 的值 3 传给了形参 `x`。按 Alt+7 组合键，打开 Call Stack 窗口，如图 1.2.14 所示，该窗口显示当前正在执行的函数是 `sign()`，其后将执行的函数是 `main()`。

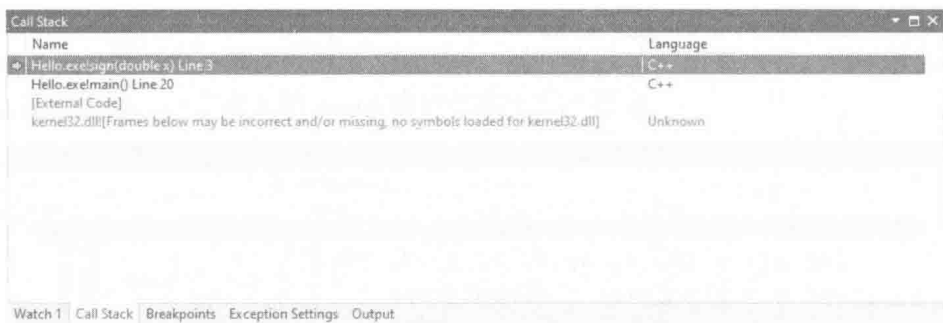


图 1.2.14 调用函数 `sign()` 后的 Call Stack 窗口

第三步：一直按 F10 键，结束对函数 `sign()` 单步执行过程的查看，此时黄色箭头回到第 20 行。Autos 窗口显示函数 `sign()` 的返回值为 1。再查看 Call Stack 窗口，如图 1.2.15 所示，退出函数 `sign()` 之后，当前正在执行的函数是 `main()`。

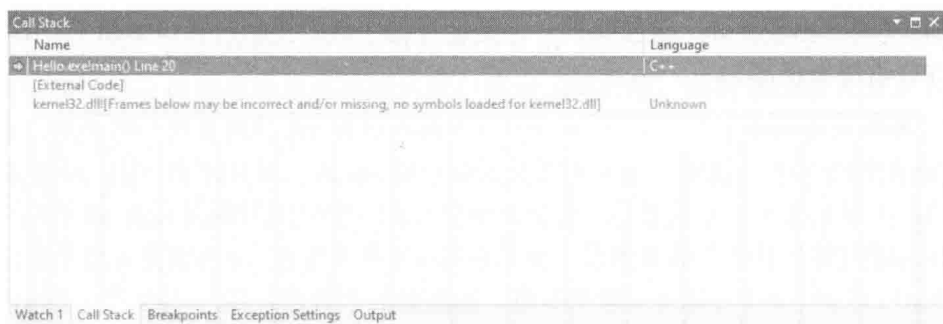


图 1.2.15 完成函数 `sign()` 调用后的 Call Stack 窗口

第四步：先后按 F10 键和 F5 键，结束程序的调试。

从对上述分析过程的分析可以看出，调试器只是提供了一些辅助工具，这些工具能够从不同侧面揭示程序运行的过程。但是真正起决定作用的还是程序员，只有把调试器提供的信息进行综合分析，并结合程序员的判断，才能跟踪程序正确执行的过程。那些认为“只要单步执行程序了，调试器就能够帮助我们找出程序错误”的想法是不对的。

2. 程序错误排除

【例 1.2.5】循环调试。

某同学粗心，写了一个函数 `my_sort()` 对数组 `a` 从小到大排序，其排序算法采用直接选择法，然后把排序之后的数组元素显示在屏幕上，如下是该同学的程序代码。

```
#01     #include <iostream>
#02
```



```

#03     void my_sort(int a[], int n) {                //有问题的直接选择法排序
#04         for (int i = 0; i < n - 1; ++i) {
#05             for (int j = i + 1; j < n; ++j) {
#06                 if (a[i] > a[j]) {
#07                     int t = a[i];
#08                     a[j] = a[i];
#09                     a[i] = t;
#10                 }
#11             }
#12         }
#13     }
#14
#15     int main() {
#16         int a[] = {2, 6, 3, 4, 1, 5, 7, 8};
#17         int n = sizeof(a) / sizeof(*a);
#18
#19         my_sort(a, n);
#20
#21         for (int i = 0; i < n; ++i)
#22             std::cout << a[i] << "\t";
#23         std::cout << std::endl;
#24     }

```

该程序没有出现编译、链接错误，但是运行程序后的控制台窗口如图 1.2.16 所示，而且 CPU 占用率极高。

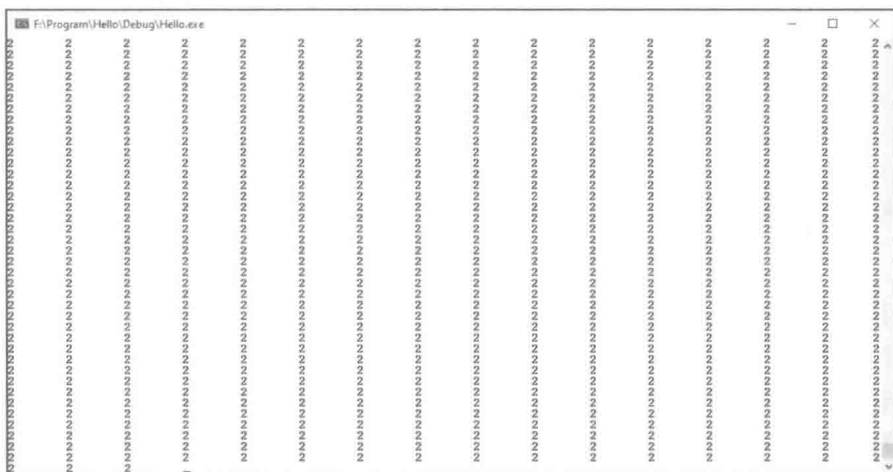


图 1.2.16 运行数组排序后的结果

第一阶段：排除输出错误。

预期的程序运行结果应该是 1 2 3 4 5 6 7 8。但是显示结果与预想差很远，而且程序还在不停地向控制台窗口写入，CPU 占用率高，这都说明循环控制进入死循环。

先暂且不管排序算法是否正确，在 `main()` 函数中把对该函数调用的语句（即第 19 行 `my_sort(a, n)`）注释掉，然后重新编译链接程序，并运行。结果还是与图 1.2.16 所示相同，这更加证明我们的判断是对的，输出数组元素的 `for` 循环出了问题。解决此问题

的步骤如下。

第一步：将鼠标指针定位到第 22 行，即 `main()` 函数中 `for` 循环所在的代码行，按 `Ctrl+F10` 组合键，运行到光标所在的行（第 22 行）。此时，从 `Autos` 窗口可以看出，在进入 `for` 循环执行之前，数组元素的个数是 8，数组 `a` 的首地址是 `0x009cfd28`，首元素的值是 2。

在 `Autos` 窗口中，单击数组 `a` 前面的 `+`，展开数组 `a` 的所有内容，可以看到数组 `a` 中所有元素的值。一个放大的 `Autos` 窗口如图 1.2.17 所示。

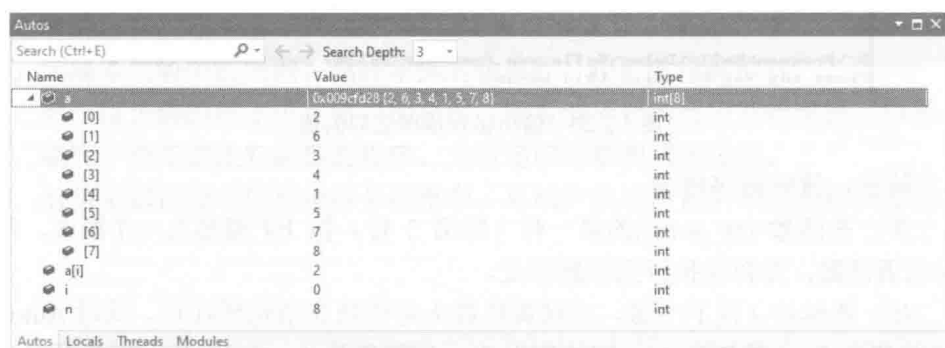


图 1.2.17 放大的 `Autos` 窗口

第二步：按一次 `F10` 键，由于 `i` 值为 0 小于 `n`，故 `for` 循环的条件表达式 `i < n` 为真，应该执行循环体，故此次单步在控制台窗口输出 `a[0]` 的值，同时黄色箭头回到 `for` 语句所在行，因为下一步将执行 `for` 语句的循环语句 `++n`。

第三步：按一次 `F10` 键，`Autos` 窗口如图 1.2.18 所示。此时，细心的读者就会发现问题了，在 `for` 循环的循环语句中，应该是循环变量 `i` 自增到下一个下标值 1，结果却是数组元素个数 `n` 增加到 9，而循环变量 `i` 没有变化。这就解释了为什么输出数组元素时，总是输出 2（即 `a[0]`），而且总是不停地输出（循环变量 `i` 永远小于 `n`，因为 `i` 不变，而 `n` 不断自增）。

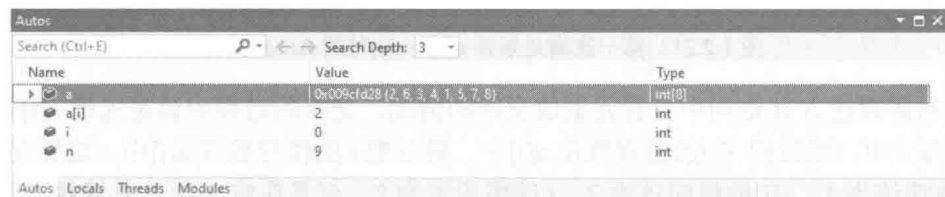


图 1.2.18 循环一次后的 `Autos` 窗口

至此明白问题所在，本来应该自增循环变量 `i`，却使数组元素个数 `n` 不断自增，原来是 `for` 循环中的循环语句写错了，本来应该是 `++i`，却写成了 `++n`。据此改正 `for` 循环语句如下：

```
#01     for (int i = 0; i < n; ++i)                //不应是++n
#02         cout << a[i] << "\t";
```

修改程序之后，重新编译链接，得出运行结果，如图 1.2.19 所示。从图中可见，输出排序之前数组的值 2 6 3 4 1 5 7 8 是对的。去掉对排序函数调用语句的注释，重新执行

程序, 得到运行结果, 如图 1.2.20 所示。从图中可见, 排序之后的结果 2 6 6 6 6 6 7 8 还是不正确。因此, 很明显是排序算法出了问题。

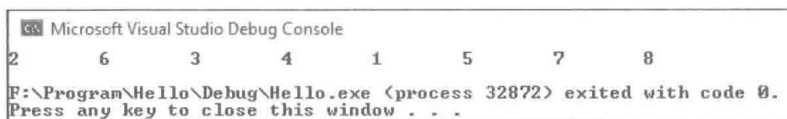


图 1.2.19 输出排序之前数组的值

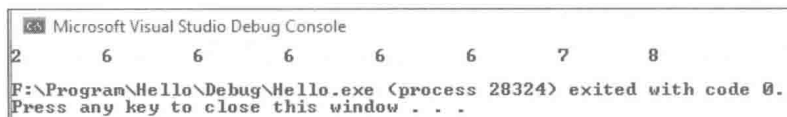


图 1.2.20 输出错误排序之后的值

第二阶段: 排除排序错误。

第一步: 在函数 `my_sort()` 的第一行 (即第 3 行) 按 F9 键插入一个断点, 然后按 F5 键启动调试器, 并自动执行到该断点处。

第二步: 连续按 3 次 F10 键, 直到黄色箭头定位到 `if` 语句所在行。此时 Autos 窗口显示: `i` 的值为 0, `j` 的值为 1, `a[i]` 的值为 2, `a[j]` 的值为 6。由于 `a[i] > a[j]` 为假, 所以不执行 `if` 语句中的交换动作。

第三步: 为了观察 `if` 语句中值交换的情况, 在 `if` 语句所在行插入一个断点。然后重复按 3 次 F5 键, 直到 Autos 窗口中第一次显示的 `a[i]` 值大于 `a[j]` 值, 此时 `i` 为 0, `j` 为 4, Autos 窗口如图 1.2.21 所示。此时可以把元素 `a[i]`、`a[j]` 和变量 `t` 都加入 Watch 窗口。

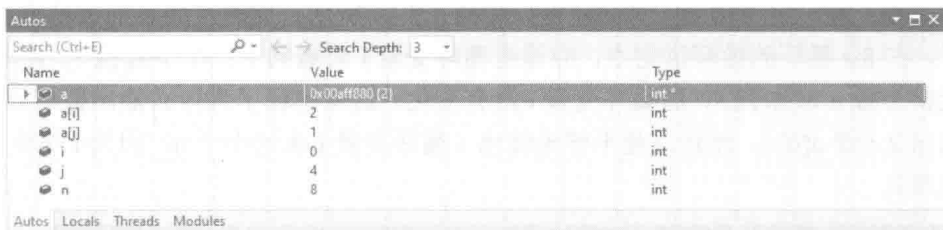


图 1.2.21 第一次满足条件 `a[i] > a[j]` 时的 Autos 窗口

此时需要进入 `if` 语句中查看元素值交换的情况。交换的过程应该是先把 `a[i]` 存放在临时变量 `t` 中, 然后把 `a[j]` 的值存放在 `a[i]` 中, 最后把 `t` 的值存放在 `a[j]` 中。这样交换后, `a[i]` 的值应该为 1, `a[j]` 的值应该为 2, `t` 的值应该为 2。但是在按 3 次 F10 键执行完一遍 `if` 语句之后, 元素 `a[i]`、`a[j]` 和变量 `t` 的值都是 2。这说明交换语句的顺序出了问题。

仔细查看 `if` 语句中循环赋值的 3 条语句, 发现的确是它们写错了。

第四步: 如下第 5~7 行修改程序, 重新编译链接, 运行程序, 发现结果正确。正确的排序算法 `my_sort` 代码如下:

```
#01 void my_sort(int a[], int n) { //改正之后的函数
#02     for (int i = 0; i < n - 1; ++i) {
#03         for (int j = i + 1; j < n; ++j) {
#04             if (a[i] > a[j]) {
```

```
#05             int t = a[i];
#06             a[i] = a[j];
#07             a[j] = t;
#08             }
#09         }
#10     }
#11 }
```

实验题目与练习

1. 完成本实验中的例题，熟练掌握各种操作，记住常用快捷键。
2. 对一段正确的程序进行错误的修改，观看编译器的反应及错误提示信息。
3. 编写一个带有递归函数的程序，分析递归函数的调用过程。
4. 对常见错误提示信息进行分类整理，总结产生该错误的原因及解决方法。

实验3 C++语言基础

实验目的与要求

1. 实验目的

- (1) 理解数据类型、运算符等基本概念。
- (2) 掌握常用控制结构。
- (3) 理解引用的概念。
- (4) 掌握 C++ 新增的 4 种函数机制。
- (5) 掌握数组、指针和字符串的使用方法。

2. 实验要求

- (1) 能够正确应用数据类型，对各类表达式正确求值。
- (2) 熟练应用控制结构解决比较简单的问题。
- (3) 能够正确设置引用型函数参数和函数返回类型。
- (4) 熟练应用 4 种函数机制，尤其是函数模板。
- (5) 熟练应用数组、指针和字符串，并能应用它们设计简单的算法。

实验过程与示例

在本次实验中，需要按照实验目的和要求，依次练习数据类型、基本运算、常用控制结构、引用与函数、函数模板、数组、指针与字符串等相关知识。

以下是一些较典型的示例，它们分别对基本数据类型的属性相关的操作、基本运算与控制结构、引用与函数、函数模板、指针与二维数组的应用进行示范。

【例 1.3.1】基本数据类型的属性应用示例。

现代 C++ 语言提供了很多关于数据类型计算的工具，其中本例所示即为关于基本数据类型属性获取的工具函数。其他更多的属性工具可以查阅标准库。

```
#01     #include <limits>
#02     #include "xr.hpp"
#03     using namespace std;
#04
#05     template <typename T>
#06     void print_type_properties() {
#07         cout << "\n\t min: " << numeric_limits<T>::min() << endl;
#08         cout << "\t max: " << numeric_limits<T>::max() << endl;
```

```

#09     cout << "\t bits: " << numeric_limits<T>::digits << endl;
#10     cout << "\t decdigits: " << numeric_limits<T>::digits10 << endl;
#11     cout << "\t integral: " << numeric_limits<T>::is_integer << endl;
#12     cout << "\t bounded: " << numeric_limits<T>::is_bounded << endl;
#13     cout << "\t signed: " << numeric_limits<T>::is_signed << endl;
#14     cout << "\t specialized: " << numeric_limits<T>::is_specialized << endl;
#15     cout << "\t exact: " << numeric_limits<T>::is_exact << endl;
#16     cout << "\t infinity: " << numeric_limits<T>::has_infinity << endl;
#17     cout << "\t modulo: " << numeric_limits<T>::is_modulo << endl;
#18 }
#19
#20 int main() {
#21     xrv(print_type_properties<char>());
#22     xrv(print_type_properties<int>());
#23     xrv(print_type_properties<unsigned int>());
#24     xrv(print_type_properties<long>());
#25     xrv(print_type_properties<float>());
#26     xrv(print_type_properties<double>());
#27     xrv(print_type_properties<long double>());
#28 }

```

由于输出结果较多，此处不附。请对照标准库了解上述类型属性的具体含义。

【例 1.3.2】基本运算与控制结构：简易算术计算器的应用示例。

从键盘上输入形如 $1 + 2$ 的中缀表达式，即运算符在两个操作数的中间，计算该表达式的值。为了简单起见，本计算器仅计算包含一个运算符的二元算术表达式。

```

#01     #include <iostream>
#02     #include <cmath>
#03
#04     int main()
#05     {
#06         double a, b, c;                //存放操作数和结果
#07         char op;                        //存放运算符
#08
#09         for(;;) {                       //无限循环，持续计算
#10             std::cout << "Please enter an expression like 1 + 2: ";
#11             std::cin >> a >> op >> b;    //输入格式：1 + 2
#12
#13             switch(op) {                 //根据运算符进行不同的运算
#14                 case '+': c = a + b;      break; //字符为+：加法运算
#15                 case '-': c = a - b;      break; //字符为-：减法运算
#16                 case '*': c = a * b;      break; //字符为*：乘法运算
#17                 case '/': c = a / b;      break; //字符为/：除法运算
#18                 case '%': c = fmod(a, b); break; //字符为%：取余运算
#19
#20                 default: return 0;        //其他运算暂不支持
#21             }                             //以下输出运算结果
#22             std::cout << "value of " << a << op << b << " is " << c << ".\n";
#23         }
#24     }

```

在输入表达式时把操作数和运算符分别存放到不同类型的变量中, 然后判断所输入的运算符, 从而执行不同的运算。程序的主要结构是在 `switch/case` 语句中对运算符进行判断计算。为连续计算, 该判断计算过程在一个 `for` 无限循环中。上述程序对 5 种算术运算 (+、-、*、/、%) 进行判断计算, 若输入其他运算符的表达式, 则退出循环计算过程。

请思考下列问题:

(1) `double` 型数据是不能应用 % 运算符进行取余运算的, 上述程序为了实现取余运算, 采用数学库函数 `fmod()`, 请考虑实现该函数取余的过程。

(2) 如第 9 行所示, 上述程序主要是一个无限循环, 如何才能结束循环而退出程序呢?

【例 1.3.3】引用与函数: 去除重复元素的应用示例。

把数组 `a` 的 `n` 个元素复制到数组 `b` 中, 但是只在 `b` 中保留重复数组元素的第一次出现。

```
#01     #include <iostream>
#02
#03     size_t find_key(int a[], size_t n, int key);    //线性查找
#04     void remove_duplicate(int a[], size_t n, int b[], size_t &m);
#05                                           //去除重复
#06     void print_array(int a[], size_t n);          //输出数组
#07
#08     int main()
#09     {
#10         int a[] = {2, 1, 2, 1, 2, 3, 3, 4, 3, 2}; //源数组
#11         const size_t N = sizeof(a) / sizeof(*a); //元素个数, 定义为常量
#12         int b[N];                                //定义为等长数组
#13         size_t m;                                 //保存不重复元素的个数
#14
#15         std::cout << "firstly: ";
#16         print_array(a, N);                        //首先输出数组元素
#17
#18         remove_duplicate(a, N, b, m);             //去除重复元素
#19
#20         std::cout << "finally: ";
#21         print_array(b, m);                        //输出结果数组 b
#22     }
#23
#24     size_t find_key(int a[], size_t n, int key) { //在数组中查找 key
#25         size_t idx = 0;                          //起始下标
#26         while (idx != n && a[idx] != key)          //尚未找到
#27             ++idx;                                //继续前进
#28         return idx;                               //返回结果下标, 其值为 n 表示未找到
#29     }
#30     //去除数组 a 中的重复元素, 结果存放在数组 b 中, 不重复元素的个数为 m
#31     void remove_duplicate(int a[], size_t n, int b[], size_t &m) {
#32         m = 0;                                    //计数器或存放下标清零
#33         for (size_t idx = 0; idx != n; ++idx) { //对 a 中的每个元素
#34             size_t p = find_key(b, m, a[idx]); //查找是否在 b 中出现
```

```

#35         if (p == m) {                                //若没有找到
#36             b[m] = a[idx];                            //可以存放在数组 b 中
#37             ++m;                                       //增加计数器 (也表示下一个元素的存放位置)
#38         }
#39     }
#40 }
#41 void print_array(int a[], size_t n) {                  //输出数组元素
#42     for (size_t idx = 0; idx != n; ++idx)
#43         std::cout << a[idx] << "\t";
#44     std::cout << std::endl;
#45 }

```

去除重复元素的过程是,把数组 *a* 中的元素逐个复制存放 to 数组 *b* 中,但是在存放之前,首先查找数组 *b* 现有的元素中是否已经存在将要存放的新元素,因而需要通过函数 `find_key()` 在数组 *b* 现有的元素中查找 (表达式 `find_key(b, m, a[idx])`), 如果没有找到,则存放元素,同时增加数组 *b* 中元素的个数 *m* (该值同时也作为新元素存放的下标)。

请思考下列问题:

(1) 对函数 `remove_duplicate()` 的第三个参数 *m* 的理解是本程序的难点。由于数组 *b* 中的元素是一个一个地存放进去的,在存放之前先要知道存放位置的下标 (该值同时也作为数组元素的个数),因此在调用函数 `remove_duplicate()` 之后,需要知道数组 *b* 中存放了多少个元素,这需要以引用形式传出参数 *m* 的值,请思考:能否以值传递方式传递参数 *m*? 为了使用 *m* 计数,需要做哪些工作? 对应形参 *m* 的实参是如何传递的?

(2) 为了节省程序的存储空间,也可以仅在原数组中通过移动不重复元素去覆盖重复元素,请实现这种算法。

(3) 请尝试以指针方式传递 `remove_duplicate()` 函数的第四个参数,并修改该函数调用的方式,比较这两种方式。

【例 1.3.4】函数模板与指针: 计算数组中值最大的元素的应用示例。

```

#01     #include "xr.hpp"
#02
#03     template <typename T>                                //函数模板
#04     T& array_max(T* a, size_t n) {                      //计算值最大的第一个元素
#05         T* m(a);                                       //假定首元素值最大
#06         for (T* p = a + 1; p != a + n; ++p) {        //与后面元素比较
#07             if (*p > *m)                                //若某元素大于当前标准
#08                 m = p;                                  //修改 m 指向它
#09         }
#10         return *m;                                     //返回最大值
#11     }
#12
#13     int main()
#14     {
#15         int a[] = {12, 23, 45, 25, 30, 80, 20};
#16         size_t n = sizeof(a) / sizeof(*a);
#17
#18         xr(array_max(a, n));                            //计算最大值
#19         array_max(a, n) /= 10;                         //把最大值缩小

```



```
#20         xr(array_max(a, n));
#21     }
```

本程序的关键在于函数 `array_max()` 的返回值是引用类型，这意味着该函数调用表达式可以用作左值，从而可以修改所返回的值（如 `array_max(a, n) /= 10` 所示）。当再次计算最大值时，最大值的元素可能变为其他元素。

请思考下列问题:

- (1) 分析求取数组元素最大值的算法 `array_max()`, 为什么指针 `p` 从 `a+1` 开始呢?
- (2) 如果函数 `array_max()` 以值方式返回结果(即返回类型设为 `T`), 表达式 `array_max(a, n)/=10` 还成立吗?

【例 1.3.5】二维数组：学生成绩计算的应用示例。

设有 30 个学生的 5 门课程成绩，请计算每个学生的平均成绩、每门课程的平均成绩，以及此次考试全班学生的平均成绩。为了简化程序的输入和输出，下面假设只有 3 个学生和 3 门课程。待程序调试好后，修改下列第 4、12 行的常量即可计算 30 个学生、5 门课程的成绩。

```

#01 #include <iostream>
#02 #include <iomanip>
#03
#04 #define SubjectNum 3 //课程门数
#05
#06 void input(double (*s)[SubjectNum + 1], size_t m, size_t n); //输入数据
#07 //输入数据
#08 void print(double (*s)[SubjectNum + 1], size_t m, size_t n); //输出数据
#09 //输出数据
#10 void calc (double (*s)[SubjectNum + 1], size_t m, size_t n); //计算成绩
#11 //计算成绩
#12 int main()
#13 {
#14     const size_t StudentNum = 3; //学生人数
#15     double score[StudentNum + 1][SubjectNum + 1] = {10, 20, 30, 0,
#16         40, 50, 60, 0, 70, 80, 90, 0};
#17     //存放成绩,多出1行1列用于存放各种总成绩
#18     std::cout << "Original scores: " << std::endl;
#19     print(score, StudentNum, SubjectNum); //输出原始数组
#20     calc(score, StudentNum, SubjectNum); //计算成绩
#21     std::cout << "Scores including sum: " << std::endl;
#22     print(score, StudentNum + 1, SubjectNum + 1); //输出计算结果
#23 }
#24 //按行输入每个学生的成绩
#25 void input(double (*s)[SubjectNum + 1], size_t m, size_t n) {
#26     for (size_t i = 0; i != m; ++i) {
#27         std::cout << "Please enter " << n
#28             << " scores of student " << i + 1 << ": ";
#29         for (size_t j = 0; j != n; ++j)
#30             std::cin >> s[i][j];
#31     }
#32 } //按行输出所有学生的成绩

```

```

#33 void print(double (*s)[SubjectNum + 1], size_t m, size_t n) {
#34     for (size_t i = 0; i != m; ++i) {
#35         for (size_t j = 0; j != n; ++j)
#36             std::cout << std::setw(4) << s[i][j];
#37         std::cout << std::endl;
#38     }
#39 } //计算所有学生和课程总成绩
#40 void calc (double (*s)[SubjectNum + 1], size_t m, size_t n) {
#41     for (size_t i = 0; i != m; ++i) {
#42         s[i][n] = 0; //存放第 i 个学生的总成绩
#43         for (size_t j = 0; j != n; ++j)
#44             s[i][n] += s[i][j]; //累加第 i 行
#45     }
#46
#47     for (size_t j = 0; j != n; ++j) {
#48         s[m][j] = 0; //存放第 j 门课程的总成绩
#49         for (size_t i = 0; i != m; ++i)
#50             s[m][j] += s[i][j]; //累加第 j 列
#51     }
#52
#53     s[m][n] = 0; //存放所有学生的总成绩
#54     for (size_t j = 0; j != n; ++j)
#55         s[m][n] += s[m][j]; //累加最后一行
#56 }

```

理解指向数组的指针是本程序的关键。3 个函数 `input()`、`print()` 和 `calc()` 都以二维数组为计算对象, 因此在形参的设置上都以指向数组的指针作为第一个参数, 该指针所指数组要求列宽度为 `SubjectNum+1`, 元素类型为 `double`。这与实参数组 `score` 正好匹配。

请思考下列问题:

(1) 为求每门课和每个学生的平均成绩, 上述程序是如何存储它们的呢? 由此需要访问不同位置的数组元素, 请仔细分析上述程序是如何访问特殊位置元素的。

(2) 对于本例中的 3 个自定义函数, 除了用指向数组的指针作为第一个参数, 也可以用二维数组类型作为第一个参数, 请以此方式定义这 3 个函数。

实验题目与提示

1. 扩展例 1.3.1 的简易计算器, 使它能够计算其他类型的二元运算表达式。

【提示】增加 `switch` 语句的 `case` 标号, 使它能够支持二元的关系运算和逻辑运算等。

2. 仿照例 1.3.2, 把某数组中的偶数挑选出来形成一个新的数组, 并输出偶数数组中的元素。

【提示】在把偶数逐渐存放到偶数数组的过程中, 需要一个元素个数计数器 (同时用作元素存放位置), 为把该数据传出函数, 需要将其设置为函数的引用参数。这可以仿照例 1.3.2 完成。

3. 编写函数 `ltrim()` 去除字符串中的前导空格字符, 并定义函数 `rtrim()`, 用于去除

字符串中后面连续的空格字符。

【提示】若字符串用 C 字符数组存放，为了去除后面连续的空格，可以直接把第一个空格字符替换为字符串结束符。去除前导空格字符，有两种处理方法：第一种方法是，逐个字符前移，可以用下标 i 和 j 分别指向第一个空格和第一个非空格字符，在每个位置上，把下标 j 处的字符存放于下标 i ，然后同时前移两个下标。在填完空格字符后，把下标 j 及其后的字符逐个前移，结束时在最后一个字符后面添加一个字符串结束符。第二种方法是，可以简单地借助 C 字符串处理函数。在找到第一个非空格字符后，应用函数 `strcpy()` 把非空格字符串复制到该字符数组的第一个位置即可。若字符串用 STL 类 `string` 存放，可以应用该类的成员函数 `replace` 把空格替换掉，或者使用 `erase` 删除空格字符。

4. 自定义函数模板实现 STL 算法 `count()` 的功能，统计区间中与某值相等元素的个数。

【提示】定义函数模板，把需要比对的元素设为数组的参数。在函数体中，设置计数器准备计数。通过 `for` 循环逐个访问区间元素，若该元素与待查找的值相等，则增加计数器。扫描整个区间，得出统计结果，然后返回计数器的值。

5. 自定义函数模板实现 STL 算法 `max_element()` 的功能，计算区间中值最大的元素。

【提示】首先把区间第一个元素作为当前最大值的标准，然后通过 `for` 循环开始访问其后的每个元素，若有某元素比当前最大值还大，则修改当前最大值。当扫描完整个区间，则当前最大值为最终最大值。注意，在记录最大值时，应记录它的下标或地址。所定义的函数模板最好以引用形式返回最大值。

6. 扩展例 1.3.4，在二维数组中按照学生平均成绩排序。

【提示】排序算法可以选用直接选择法，或者冒泡法。需要注意的是，交换两个元素时，应同时交换该两列（按课程平均成绩排序）、该两行（按学生平均成绩排序）的所有元素。这需要熟练操作二维数组元素的访问。

7. 牛顿迭代法：对于方程 $f(x)=0$ ，若从一个合适的初值开始，用公式

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

经过多次迭代，则可以求得方程的近似根。以一元三次方程 $f(x) = x^3 - 16x^2 + 73x - 90 = 0$ 为目标，求出它的 3 个根 2、5、9。

【提示】首先需要把 $f(x)$ 及其导数代入牛顿迭代公式并推导出迭代公式，然后依次在 2、5、9 附近给定一个初始值（如 1.5、4.5、9.5）开始迭代。当前后两次计算的 x 值相差很小（差值的绝对值小于 1×10^{-8} ）时，认为已经达到迭代的精度，从而终止迭代。注意，在对多项式求值时，应按照秦九韶算法计算。

8. 括号配对：为了检查 C++ 表达式中的圆括号数量是否相等，可以采用计数法来简单地判断两种情况，开闭圆括号的出现次数不等；在某一位置之前闭括号多于开括号。请编程实现。

【提示】以全局函数 `getline()` 读入存储为 `string` 形式的表达式，从第一个字符开始比对，同时设置计数器开始计数，若遇到字符 '('，则增加计数器，若遇到字符 ')'，则减小计数器。当整个表达式扫描完，若该计数器为零，则表明括号匹配；若计数器为正，则

说明开括号多于闭括号；若计数器为负，则说明闭括号多于开括号。

9. 数学黑洞：任意输入一个四位数，其各位数字不能全部相同。将组成该数的各位数字重新排列，组成一个最大数和一个最小数，然后将它们相减，其差仍为一个自然数，重复进行上述运算，观察并总结最终的结果。

【提示】对于该四位数 n ，通过运算 $n/1000$ 、 $n\%1000/100$ 、 $n\%100/10$ 、 $n\%10$ 依次得到它的千位、百位、十位和个位上的数字。再定义两个函数分别对 4 个参数按照从大到小、从小到大排序。把排序之后的数字组合成两个四位数，然后相减。把所得的差作为最开始的四位数，用循环重复 20 次这个过程，观察所得的结果，然后得出结论。

10. 整数 n 的因子：求出 n 的所有因子（不包括自身），并保存在数组中，从函数中返回该数组，在 `main()` 函数中输出这些因子。

【提示】定义一个函数，以整数 n 为第一个参数，在函数中以 $[1, n-1]$ 内的整数试探它们是否能够整除整数 n ，若能则把它们通过成员函数 `push_back()` 存入 `vector` 容器中，然后以引用参数形式传出该容器对象，最后以下标方式或迭代器方式输出容器中的元素。

实验 4 STL 常用算法与容器

实验目的与要求

1. 实验目的

- (1) 掌握 STL 常用算法。
- (2) 理解容器的概念，掌握常用的 STL 容器类。

2. 实验要求

- (1) 能够正确、熟练地应用 STL 算法解决比较简单的问题。
- (2) 熟练应用 `vector` 和 `string` 类的常用操作。

实验过程与示例

本次实验的主要目的是，练习并熟记 STL 常用算法和常用容器类的操作。本实验按照教材上对算法的分类，分别给出常用算法的典型示例，并对常用容器类 `vector` 和 `string` 进行了典型应用说明。

【例 1.4.1】算法应用 1——分析学生成绩。

将某次考试中的 n 个学生成绩存储在数组中，下面应用 STL 常用算法对这些成绩进行常见分析。

```
#01     #include <iostream>
#02     #include <algorithm>
#03     #include <functional>
#04
#05     void print_score(double d) {std::cout << d << "\t";} //输出数据
#06     bool lower_than_60(double d) {return d < 60;} //表示不及格的分数
#07
#08     int main()
#09     {
#10         double s[] = {85, 76, 60, 54, 89, 90, 96, 45, 85, 85, 98};
#11         size_t n = sizeof(s) / sizeof(*s); //成绩数组及其长度
#12
#13         std::cout << "all scores are : \n";
#14         std::for_each(s, s + n, print_score); //输出所有成绩
#15
#16         double k = 85;
#17         size_t m = std::count(s, s + n, k); //有几个 85 分
```

```

#18      std::cout << "\nthere are " << m << " scores equal to "
#19      << k << std::endl;
#20      double *p = std::find(s, s + n, k);    //第一个 85 分在哪儿
#21      std::cout << "first " << k << " is at " << p - s << std::endl;
#22      p = std::search_n(s, s + n, 2, k);    //连续两个 85 分在哪儿
#23      std::cout << "first 2 consecutive scores of " << k
#24      << " is at " << p - s << std::endl;
#25      p = std::search_n(s, s + n, 2, k, std::greater<double>());
#26      std::cout << "first 2 consecutive scores greater than " << k
#27      << " is at " << p - s << std::endl; //连续两个大于 85 分的成绩在哪儿
#28
#29      m = std::count_if(s, s + n, lower_than_60);    //多少人不及格
#30      std::cout << "there are " << m << " scores failed to pass the exam.\n";
#31      p = std::find_if(s, s + n, lower_than_60); //第一个不及格的在哪儿
#32      std::cout << "first score failed is at " << p - s << std::endl;
#33
#34      p = std::min_element(s, s + n);    //最低分是多少
#35      std::cout << "score " << *p << " is the lowest." << std::endl;
#36      p = std::max_element(s, s + n);    //最高分是多少
#37      std::cout << "score " << *p << " is the highest." << std::endl;
#38  }

```

上述程序中, `for_each(s, s+n, print_score)` 输出数组中的所有成绩。`count(s, s+n, k)` 统计分数 85 的出现次数。`find(s, s+n, k)` 计算分数 85 第一次出现的位置。`search_n(s, s+n, 2, k)` 计算分数 85 第一次连续两次出现的位置。`search_n(s, s+n, 2, k, greater<double>())` 计算连续两个大于 85 的分数第一次出现的位置。`count_if(s, s+n, lower_than_60)` 统计不及格(分数低于 60 分)的人数。`find_if(s, s+n, lower_than_60)` 统计第一个不及格分数出现的位置。`min_element(s, s+n)` 计算这些分数中的最低分; `max_element(s, s+n)` 计算这些分数中的最高分。

【思考】如何定义函数表示不同的分数段? 请使用函数 `count_if()` 继续统计各分数段的分布人数。

【例 1.4.2】`pair` 工具类的使用: 计算两个数的最大值和最小值。

如果需要从函数中同时返回两个值, 可以使用 `pair` 进行封装。它可以把不同类型的两个数据通过工具函数 `make_pair()` 封装成一个整体返回。对它的两个成员分别通过 `first`、`second` 访问可以得到。

```

#01      #include <utility>    //for pair
#02      #include "xr.hpp"
#03
#04      template <typename T>
#05      std::pair<T, T> max_min(T a, T b) {
#06          T tm = std::max(a, b);    //调用 STL 函数 max 求最大值
#07          T tn = std::min(a, b);    //调用 STL 函数 min 求最小值
#08          return std::make_pair(tm, tn);    //把两个数封装成一个 pair
#09      }
#10
#11      template <typename T>

```

```

#12     void print(std::pair<T, T> p) {
#13         std::cout << "max: " << p.first           //输出第一个数
#14         << "\tmin: " << p.second << std::endl;    //输出第二个数
#15     }
#16
#17     int main()
#18     {
#19         xrv(print(max_min('a', 'b')));
#20         xrv(print(max_min(20, 10)));
#21         xrv(print(max_min(1.5, 2.5)));
#22     }

```

程序的输出结果如下:

```

#19: [print(max_min('a', 'b'))] ==>max: b      min: a
#20: [print(max_min(20, 10))]   ==>max: 20     min: 10
#21: [print(max_min(1.5, 2.5))] ==>max: 2.5    min: 1.5

```

【例 1.4.3】算法应用 2——不同颜色球的组合。

从 5 种不同颜色的球中无放回地取出 3 个球,求这 3 个球可能的颜色组合及其种数。

```

#01     #include <iostream>
#02     #include <algorithm>
#03     #include <string>
#04     //定义枚举类型表示 5 种颜色, 注意各常量表示的整数值 (二进制数的不同位)
#05     enum Color {red = 1, green = 2, blue = 4, black = 8, white = 16};
#06
#07     void printcolor(Color c) {                //输出各枚举常量对应的单词
#08         std::string s;
#09         switch (c) {                          //输出整数值及其单词
#10             case red:   s = " 1:red "; break;
#11             case green: s = " 2:green"; break;
#12             case blue:  s = " 4:blue "; break;
#13             case black: s = " 8:black"; break;
#14             case white: s = "16:white"; break;
#15         }
#16         std::cout << s << "\t";
#17     }
#18     //把一组颜色值通过位或运算合并成一个数
#19     int color2int(Color* first, Color* end)    { //合并数组中的颜色值
#20         int n;
#21         for (n = *first; first != end; ++first) //对于所有的颜色
#22             n = n | *first;                    //通过位或组合在一个数中
#23         return n;                              //返回和值
#24     }
#25     //实现上面函数相反的功能:通过位与运算从一个数中拆出所包含的颜色值
#26     void int2color(int n) {                    //分析 n 中包含的颜色
#27         std::cout << n << "\t";              //先输出 n
#28         if (n & red) printcolor(red);         //red 为二进制的 1
#29         if (n & green) printcolor(green);     //green 为二进制的 10
#30         if (n & blue) printcolor(blue);       //blue 为二进制的 100
#31         if (n & black) printcolor(black);     //black 为二进制的 1000

```

```

#32         if (n & white) printcolor(white);           //white 为二进制的 10000
#33         std::cout << std::endl;
#34     }
#35
#36     int main()
#37     {
#38         Color ball[] = {red, green, blue, black, white}; //五色球数组
#39         size_t n = sizeof(ball) / sizeof(*ball);         //元素个数
#40         int color[100];                                   //存放 3 球排列
#41         size_t num = 0;                                   //排列的种数
#42
#43         while (std::next_permutation(ball, ball + n)) { //对任一排列
#44             std::for_each(ball, ball + n, printcolor); //输出所有颜色
#45             int color_index = color2int(ball, ball + 3); //计算前 3 球对应的值
#46             if (std::find(color, color + num, color_index) == color + num) {
#47                 color[num] = color_index;               //该值不重复, 则保存
#48                 ++num;                                   //增加计数
#49             }
#50             std::cout << std::endl;
#51         }
#52         std::cout << "****Below are " << num << " results****" << std::endl;
#53         std::for_each(color, color + num, int2color);
#54     }

```

能够得出所有球的排列, 主要依赖函数 `next_permutation()`。为了求出 3 个球所有可能的颜色组合情况, 上述程序主要采用了两个步骤: 第一步, 在表达式 `next_permutation(ball, ball+n)` 中调用算法 `next_permutation()` 列出所有可能的颜色排列情况。由于在全排列中, 所有颜色在每个位置上出现的次数是等同的, 因此, 对于取出 3 个球, 可以选择全排列中前 3 个位置上颜色排列的情况进行组合分析。第二步, 应用表达式 `color2int(ball, ball+3)` 分析所有排列中前 3 个位置上颜色的分布情况, 从而得知取出 3 种颜色的组合情况。

为了辅助问题解决, 上述程序采用位运算以简化问题处理, 主要涉及 3 个地方。

第 1 处: 在定义枚举类型 `Color` 时, 让每种颜色常量对应一个二进制位, 这种处理方式便于统计不同颜色球的组合。

第 2 处: 函数 `color2int()` 把每个排列前 3 个位置的颜色组合成一个整数, 这主要是通过位或运算实现的。在把每个排列前 3 个位置的颜色对应成一个颜色索引值 `color_index` 后, 就可以把此整数存放在数组 `color` 中, 从而可以运用算法 `find()` 查看此 3 种颜色的组合是否已经出现并被统计, 如 `find(color, color+num, color_index) == color+num` 所示。若新出现一种颜色组合, 即算法 `find()` 在现有颜色组合中查找失败, 则在数组 `color` 中存放此新的颜色组合, 如 `color[num]=color_index` 所示, 同时用 `++num` 增加组合的种数。

第 3 处: 为了从颜色索引值中输出它所对应的 3 种颜色, 函数 `int2color()` 对颜色索引值 `n` 分别与每个颜色常量进行位与运算, 从而判断该索引值所对应的 3 种颜色组合。

【思考】请应用 `next_permutation()` 算法得出数组元素的全排列。

【例 1.4.4】vector 容器应用——Eratosthenes 筛选法求质数。

```

#01     #include <iostream>
#02     #include <vector>
#03     #include <cmath>
#04
#05     int main()
#06     {
#07         const size_t MaxNum = 500;           //判断从 1 到 500 之间的质数
#08         std::vector<size_t> v(MaxNum + 1);    //多 1 个元素:下标与数一致
#09
#10         std::fill_n(v.begin(), MaxNum, 1);   //填充为 1,假定全为质数
#11
#12         for (size_t i = 2; i * i <= MaxNum; ++i) {
#13             for (size_t j = 2; j <= MaxNum / i; ++j) { //j 是 i 可能的倍数
#14                 v[i * j] = 0;                  //把 i 所有的倍数都去掉
#15             }
#16         }
#17
#18         for (size_t i = 2; i <= MaxNum; ++i) {
#19             if(v[i] == 1) {                    //未被改为零,则为质数
#20                 std::cout << i << "\t";
#21             }
#22         }
#23         std::cout << std::endl;
#24     }

```

筛选法的基本思想是从 2 到该数的平方根作为除数,逐个筛选去除每个数的倍数。上述程序中,fill_n(v.begin(), MaxNum, 1)先把容器 v 中每个元素填充以数值 1,即先假定该元素对应的下标是质数。嵌套的 for 循环依次去除 2、3…的倍数(即筛选法的思想),去除的方法是把相应下标的元素重设为 0。最后一个 for 循环输出元素不为 0 的下标,即剩下的质数。

【例 1.4.5】string 类应用——文件名分解。

现有完整的文件名 full_name,是形如"f:\\workshop\\Program\\Hello\\main.cpp"的字符串,要求从该文件名中分离出驱动器名 driver_name ("f")、路径名 path_name ("f:\\workshop\\Program\\Hello")、文件名 file_name ("main.cpp")、扩展名 ext_name ("cpp")。

```

#01     #include <string>
#02     #include "xr.hpp"
#03
#04     void splitFileName(const std::string& full_name,
#05                       std::string& driver_name, std::string& path_name,
#06                       std::string& file_name, std::string& ext_name);
#07
#08     int main()
#09     {
#10         std::string driver, path, file, ext;
#11

```

```

#12     std::string full("f:\\workshop\\Program\\Hello\\main.cpp");
#13     splitFileName(full, driver, path, file, ext);
#14     xr(full); xr(driver); xr(path); xr(file); xr(ext);
#15 }
#16
#17 void splitFileName(const std::string& full_name,
#18                  std::string& driver_name, std::string& path_name,
#19                  std::string& file_name, std::string& ext_name)
#20 {
#21     if (full_name.empty()) {                //全路径为空,则所有为空
#22         driver_name = ""; path_name = "";
#23         file_name = ""; ext_name = "";
#24         return;
#25     }
#26
#27     std::string::size_type idx;
#28     idx = full_name.find_first_of(':');      //':'之前为驱动器名
#29     if (idx != std::string::npos)          //找到驱动器字符
#30         driver_name.assign(full_name, 0, idx - 0); //保存到字符串对象中
#31
#32     idx = full_name.rfind("\\");            //查找最后一个"\\\"
#33     if (idx != std::string::npos) {
#34         path_name = full_name.substr(0, idx - 0); //其前为路径名
#35         file_name = full_name.substr(idx + 1);    //其后为文件名
#36     }
#37
#38     idx = full_name.find_last_of('.');      //查找最后一个'.'其后为扩展名
#39     if (idx != std::string::npos)          //找到
#40         ext_name = full_name.substr(idx + 1); //截取子串保存到字符串对象中
#41 }

```

本题主要应用 STL 类 `string` 的常用操作,对字符串进行查找和分割。成员函数 `find_first_of()`、`rfind()`、`find_last_of()` 分别查找字符 ':' 的第一次出现位置、"\" 的最后一个出现位置、字符 '.' 的最后一次出现位置,若查找成功,则它们的返回值不是 `std::string::npos`。成员函数 `assign()` 把参数指定的字符串赋值给对象。成员函数 `substr()` 截取并返回参数指定的子字符串。

程序的输出结果如下:

```

#14: full    ==>f:\workshop\Program\Hello\main.cpp
#14: driver  ==>f
#14: path    ==>f:\workshop\Program\Hello
#14: file    ==>main.cpp
#14: ext     ==>cpp

```

实验题目与提示

1. 应用 STL 算法求出数组中所有相同元素,以及这些重复元素出现的次数。

【提示】应用算法 `sort()` 对数组元素排序,然后统计相邻相同的元素的个数,即每个

元素重复的次数。

2. 使用 STL 函数 `remove()` 删除与某值相等的所有元素, 自定义函数 `remove_first()` 删除自某位置开始与某值相等的第一个元素。

【提示】在函数 `remove_first()` 内部, 应用算法 `find()` 查找从指定的位置开始与该值相等的第一个元素, 若找到, 则可通过“把最后一个元素移动到该位置”以实现删除该元素的目的。

3. 自定义函数实现 STL 函数 `reverse()` 的功能, 即把一维数组中的所有元素逆转顺序。

【提示】此题可用迭代算法或递归算法实现。对于迭代算法, 设置两个指针分别指向数组的首元素和尾元素, 当一个指针始终在另一个指针的左边时, 开始在每个位置上交换它们所指的元素, 然后移动两个指针向数组中间靠拢。对于递归算法, 把表示数组首元素和尾元素的指针或下标设置为函数的参数, 交换两个位置上的元素, 然后前移首元素的地址, 后移尾元素的地址, 开始递归调用。当两个指针重合或改变了相对位置时, 可以终止递归。

4. 自定义函数实现 STL 函数 `binary_search()` 的功能, 即在已序一维数组中查找某个元素。

【提示】在已序 (设为从小到大) 数组上的查找, 可以用迭代算法或递归算法实现。对于递归算法, 把数组的起点和终点设置为函数的参数, 首先计算起点和终点的中间位置元素, 若待查找元素等于该中间位置元素, 则查找成功, 返回位置, 退出函数。若待查找元素大于该中间位置元素, 则把查找范围修改为中间位置的右侧范围, 继续函数调用。若待查找元素小于该中间位置元素, 则把查找范围修改为中间位置的左侧范围, 继续函数调用。

5. 编写函数, 把一维数组中某个元素全部替换成另一个值。

【提示】设置一个循环过程, 首先应用算法 `find()` 查找该元素的出现位置, 若未找到, 则可退出循环结束整个过程; 若找到, 则把该元素替换为另一个值。从该位置的后续位置重复循环过程。

6. 编写函数, 求两个数组中相同的所有元素。

【提示】对元素个数较少的数组设置循环过程。应用算法 `find()` 依次在元素较多数组中查找每个元素 (来自个数较少的数组), 若找到, 则为共同元素, 存放到一个数组中 (或 `vector` 容器中)。

7. 编写函数, 比较两个字符串是否相等 (具有相同的字符元素)。

【提示】由于比较两个 `string` 类型的字符串非常简单 (运算符 `==` 即可完成), 在此仅考虑 C 字符串。设置两个字符指针分别指向两个字符串的首元素, 设置一个循环过程开始比较判断, 当这两个指针所指都不为字符串结束符时, 比较它们所指的字符元素, 若相等, 则驱动两个指针同时前移继续比较; 若不相等, 则可结束循环退出整个过程并输出两个字符串不等。判断退出循环后两个指针的值, 若一个为空而另一个不为空, 则表明两个字符串不等; 若两个字符指针同时为空, 则表明两个字符串相等。

8. 找出字符串中某个字符出现的所有位置。

【提示】根据字符串长度设置一个循环过程，在该循环过程中以下标方式访问字符元素，若某位置元素与该字符相等，则把该位置下标存放在数组或 `vector` 容器中，直到循环结束，显示结果内容。

9. 编写函数，判断数组是否有序：数组元素从小到大（或从大到小）排列称为有序。

【提示】至少有两种方法可以判断。第一种方法：把数组元素排序，然后比对排序前后的数组元素是否相等，若不等，则说明最开始的数组失序。第二种方法：类似于累加求和，首先设置一个 `bool` 变量（作用类似于累加器），根据数组元素个数设置一个循环，在循环过程中，对数组相邻的两个元素进行小于或大于的关系运算（如 `<` 或 `>`），并把这些结果通过逻辑与运算合并到该 `bool` 变量中。若退出循环后，该变量结果为真，说明数组元素有序，否则失序。

10. 去除字符串中所有非字母字符，然后改变字符串中的所有字母的大小写形式。

【提示】考虑存储为 `string` 类型的字符串。根据字符串长度设置一个循环过程，在该循环中应用成员函数 `find_first_not_of()` 查找第一个非字母字符（以 52 个字母组成的字符串作为该函数的参数），然后应用成员函数 `erase()` 删除该字符，直到退出循环过程，结束处理。为了简单地改变字母的大小写形式，可以首先定义一个函数（以 `char` 引用类型作为参数），该函数判断字母的大小写形式，然后做出改变。把该函数作为算法 `for_each()` 的第三个参数，即可改变字符串中所有字母的大小写形式。

实验 5 结构及其应用

实验目的与要求

1. 实验目的

- (1) 理解结构、枚举的概念。
- (2) 理解 list 容器及其常用操作。
- (3) 理解结构化程序设计的思想和方法。
- (4) 理解程序的多文件结构。

2. 实验要求

- (1) 能够自定义结构、枚举等数据类型解决比较简单的问题。
- (2) 能够正确应用 list 容器。
- (3) 熟练应用结构化程序设计的思想和方法解决实际问题。
- (4) 能够正确按照“接口与实现相分离”的原则组织程序文件。

实验过程与示例

在本次实验中，需要按照实验目的和要求，依次练习结构、枚举等自定义数据类型的操作，并对 STL 容器类 list 进行必要的练习以期熟练应用。以下先结合一个实例讲解 STL 容器类 list 的常用操作。

【例 1.5.1】求解二元一次方程的整数解。

本例在整数集合范围内讨论二元一次方程在第一象限的解。首先定义结构类型 Equation 和 Solution 分别表示二元一次方程及其整数解。然后定义函数 solveSingleEquation() 求解单个方程 e，并把所有的整数解存放在容器 list 对象 s 中。函数 solveEquationGroup() 用于求解两个整数方程 e1 和 e2 的公共解，并把整数解存放在容器 list 对象 s 中。函数 printEquation() 输出某个方程 e，函数 printSolution() 输出某个整数解 s。

为了有效地组织程序、并保持清晰的文件结构，应该按照“接口与实现相分离”的原则组织程序。所谓接口，即存放类或结构声明并向客户提供成员函数原型的头文件。所谓实现，即存放成员函数定义的源文件。使用这种方式组织文件，只需要向客户提供表示接口的头文件，而不需要提供类链接的目标码。

在接口文件中，一般存放类型定义、常量定义、函数原型声明、内联函数定义，以

及对其他头文件的包含指令等, 这些声明一般开放给各个客户程序, 每个客户程序只需包含该头文件就可以应用相应的函数和数据类型。在实现文件中 (一般是与头文件同名的源文件), 一般存放函数定义、变量或对象的定义等, 这些实现代码是无须开放给客户程序的。在应用文件中, 直接包含表示接口的头文件, 然后对数据类型及其函数进行测试。

总结上述“接口与实现相分离”组织程序的方式, 一般把程序组织成接口文件 (类似 Equation.h)、实现文件 (类似 Equation.cpp) 和应用文件 (类似 main.cpp)。接口文件存放数据类型的定义、函数原型、常量定义、全局变量声明等, 以及防止多重包含的条件编译指令。实现文件存放函数定义、全局变量定义等。应用文件包含对数据类型或相关函数的测试代码。

下面将本例的程序按照如下 3 个步骤分成 3 个文件。

第一步: 新建头文件 Equation.h, 在该文件中输入如下内容。

```
#01     #ifndef EQUATION_SOLUTION_STRUCT
#02     #define EQUATION_SOLUTION_STRUCT
#03
#04     #include <list>
#05
#06     struct Equation {                                //表示二元一次方程 ax+by=c
#07         int a, b, c;                                //方程的系数都是整数
#08     };
#09
#10     struct Solution {                                //表示二元一次方程的整数解
#11         int x, y;                                    //在整数范围内求解方程
#12     };
#13
#14     void solveSingleEquation(const Equation& e, std::list<Solution>& s);
#15     void solveEquationGroup(const Equation& e1, const Equation& e2,
#16                             std::list<Solution>& s);
#17     void printEquation(const Equation& e);
#18     void printSolution(const Solution& s);
#19
#20     #endif // EQUATION_SOLUTION_STRUCT
```

为了防止头文件被多重包含而引发错误 (多次包含某头文件会造成其中的数据类型被定义多次而出错), 常需要在定义有数据类型的头文件中设置用于防止多重包含的条件编译指令, 该指令有 3 行内容: #ifndef 判断符号常量 EQUATION_SOLUTION_STRUCT 定义过没有, 该常量与头文件及其中的数据类型相关, 但在各头文件之间互不相同。若指令 #ifndef 判断为真, 则由 #define 定义该常量, 并且编译随后的所有代码, 直到遇到指令 #endif 结束需要编译的代码范围。这 3 行条件编译指令构成该头文件的守护代码。需要注意的是, 一定要保证在同一个工程内每个头文件中所定义的符号常量各不相同。

第二步: 新建源文件 Equation.cpp, 在该文件中输入如下内容。

```
#01     #include <iostream>
#02     #include <list>
#03     #include "Equation.h"
```

```

#04
#05     void solveSingleEquation(const Equation& e, std::list<Solution>& s){
#06         s.clear(); //清空容器
#07         for (int x = 0; x <= e.c / e.a; ++x) { //对于集合[0, e.c / e.a]
#08             for (int y = 0; y <= e.c / e.b; ++y){ //对于集合[0, e.c / e.b]
#09                 if (x * e.a + y * e.b == e.c) { //试探每对组合是否满足方程
#10                     Solution t = {x, y}; //若是, 则组合成一个解
#11                     s.push_back(t); //并存放在容器中
#12                 }
#13             }
#14         }
#15     }
#16
#17     void solveEquationGroup(const Equation& e1, const Equation& e2,
#18                             std::list<Solution>& s) {
#19         solveSingleEquation (e1, s); //求解第一个方程
#20
#21         std::list<Solution>::iterator iter;
#22         for (iter = s.begin(); iter != s.end(); /* ++iter */) {
#23             const Solution& rs = *iter; //对第一个方程的每个解
#24             if (e2.a * rs.x + e2.b * rs.y != e2.c) //若不满足第二个方程
#25                 iter = s.erase(iter); //则去除
#26             else //否则
#27                 ++iter; //予以保留
#28         }
#29     }
#30
#31     void printEquation(const Equation& e) { //输出方程
#32         std::cout << e.a << "x+" << e.b << "y=" << e.c;
#33     }
#34
#35     void printSolution(const Solution& s) { //输出解
#36         std::cout << "(" << s.x << ", " << s.y << ") \n";
#37     }

```

为了应用头文件中声明的函数、类型或常量等,一般需要在该源文件中包含对应的头文件,如文件包含指令`#include "Equation.h"`。

函数 `solveSingleEquation()` 求解整数方程 `e` 的整数解,并存放在容器 `s` 中。由于容器 `s` 需要保存在函数中求得的解,故参数 `s` 以引用方式传递。求解的过程采用穷举法。对于方程 `e` 而言, x 坐标的范围为 $[0, e.c / e.a]$, y 坐标的范围为 $[0, e.c / e.b]$,把在此范围内的任意两个数组组合成一个 `Solution` 对象去试探方程 `e`,若能够满足该方程,则该对象为方程的解,存放在容器 `s` 中。

函数 `solveEquationGroup()` 求解两个方程 `e1` 和 `e2` 的公共解,并存放在容器 `s` 中。由于容器 `s` 需要保存在函数中求得的解,故参数 `s` 以引用方式传递。求解的过程同样采用穷举法。先调用函数 `solveSingleEquation (e1, s)` 求出方程 `e1` 所有的整数解,存放在容器 `s` 中;然后用该容器中的每个解去试探方程 `e2`,若不能满足方程 `e2`,则从容器 `s` 中去除,否则予以保留。以这种方式求解二元一次方程组,不管这两个方程是相交(有一解),

还是平行（无公共解）、重合（有多个解），函数 `solveEquationGroup()` 都能够处理。

函数 `printEquation()` 输出方程 `e`，形式为 $ax+by=c$ 。函数 `printSolution()` 输出解 `s`，形式为 (x, y) 。

第三步：新建源文件 `main.cpp`，并在其中输入如下内容。

```
#01  #include <iostream>
#02  #include <list>
#03  #include <algorithm>
#04  #include "Equation.h"
#05
#06  int main()
#07  {
#08      Equation e1 = {2, 3, 25}, e2 = {3, 4, 36}; //生成两个方程
#09
#10      std::list<Solution> s;                //生成容器以存放解
#11      solveSingleEquation(e1, s);           //求解方程 e1
#12      std::cout << "solutions of ";
#13      printEquation(e1);                   //输出方程 e1
#14      std::cout << ": \n";
#15      std::for_each(s.begin(), s.end(), printSolution);
#16                                     //输出方程 e1 的所有解
#17      solveSingleEquation(e2, s);           //求解方程 e2
#18      std::cout << "solutions of ";
#19      printEquation(e2);                   //输出方程 e2
#20      std::cout << ": \n";
#21      std::for_each(s.begin(), s.end(), printSolution);
#22                                     //输出方程 e2 的所有解
#23      solveEquationGroup(e1, e2, s);        //求解 e1 和 e2 的公共解
#24      if (s.empty ())
#25          std::cout << "Sorry, no integer solution.";
#26      else {
#27          std::cout << "common solutions of them: \n";
#28          std::for_each(s.begin(), s.end(), printSolution); //输出公共解
#29      }
#30 }
```

源文件 `main.cpp` 中一般存放驱动函数 `main()`，通过在该函数中调用自定义函数，以测试程序功能是否满足要求。在该文件中，首先生成两个方程的对象 `e1` 和 `e2`。首先调用函数 `solveSingleEquation(e1, s)` 求解方程 `e1` 的所有解，并存放在 `list` 容器对象 `s` 中，并应用算法 `for_each(s.begin(), s.end(), printSolution)` 输出所有的解。对方程 `e2` 进行同样的计算。最后应用函数 `solveEquationGroup(e1, e2, s)` 求解它们所组成方程组的公共解。

【例 1.5.2】 学生成绩及其排名的应用示例。

定义学生结构类型 `Student`，结构成员包括学号 (`sid`)、姓名 (`name`)、成绩 (`score`) 和排名 (`rank`)。按照成绩从高到低排序，并确定其排名。成绩相同者，其排名也应该相同。

```
#01  #include <iostream>
#02  #include <algorithm>
#03  #include <vector>
#04  #include <string>
```



```

#05
#06     struct Student {                                //定义学生结构类型
#07         int sid;                                    //学号
#08         std::string name;                            //姓名
#09         double score;                                //成绩
#10         int rank;                                    //名次
#11     };
#12
#13     bool compByScore(const Student& a, const Student& b) { //定义比较准则
#14         return a.score > b.score;
#15     }
#16
#17     int main() {
#18         Student a[] = { {2008001, "Jordan", 92}, {2008002, "Madonna", 75},
#19                         {2008003, "Jolie", 86}, {2008004, "Hanks", 96},
#20                         {2008005, "Cruise", 80}, {2008006, "Beckham", 88},
#21                         {2008007, "Figo", 86}, {2008008, "Avril", 88},
#22                         {2008009, "Raul", 86}, {2008010, "Kaka", 80} };
#23         size_t n = sizeof(a) / sizeof(*a);
#24
#25         std::vector<Student> v{a, a+n}; //把数组元素存放到容器中
#26         auto printAllStudents = [&v](). { //定义 lambda 函数输出容器中所有对象
#27             auto printStudent = [](const Student& s) {
#28                 //定义 lambda 函数输出 Student
#29                 std::cout << s.sid << "\t" << s.name << "\t"
#30                     << s.score << "\t" << s.rank << std::endl;
#31             }; //以下输出 vector 容器中的所有元素
#32             std::for_each(v.begin(), v.end(), printStudent);
#33         };
#34
#35         std::cout << "before sort: " << std::endl;
#36         printAllStudents(); //输出所有元素
#37
#38         std::sort(v.begin(), v.end(), compByScore); //按照成绩排序
#39
#40         for (auto iter = v.begin(); iter != v.end(); ++iter)
#41             iter->rank = int(std::distance(v.begin(), iter) + 1); //计算名次
#42
#43         std::cout << "after sort: " << std::endl;
#44         printAllStudents(); //输出排名之后的信息
#45
#46         for (auto iter = v.begin(); iter != v.end(); ++iter) {
#47             //查找每个成绩重复出现的区间[pr->first, pr->second)
#48             auto pr = std::equal_range(iter, v.end(), *iter, compByScore);
#49             //相同的成绩应该是相同的位次
#50             for (auto iter2 = pr.first; iter2 != pr.second; ++iter2)
#51                 iter2->rank = iter->rank; //全部置为第一个成绩的位次
#52         }
#53

```

```

#54         std::cout << "after reRank: " << std::endl;
#55         printAllStudents();
#56     }

```

上述程序 `vector<Student> v{a, a+n}` 首先应用容器类 `vector` 对象 `v` 存储结构数组 `a` 中的 `Student` 对象，然后定义两个 `lambda` 函数配合输出容器中所有的 `Student` 对象。其中，`lambda` 函数 `printStudent()` 输出单个 `Student` 对象，`lambda` 函数 `printAllStudents()` 通过 `[&v]` 引用捕捉容器对象 `v`，然后 `for_each(v.begin(), v.end(), printStudent)` 在算法 `for_each()` 中借助 `lambda` 函数 `printStudent()` 输出所有的 `Student` 对象。定义 `lambda` 函数 `printAllStudents()` 便于后面复用。

接着程序调用 STL 算法 `sort(v.begin(), v.end(), compByScore())` 对容器内的元素排序，第三个参数 `compByScore()` 表示比较准则，也就是说在排序过程中，两个 `Student` 对象按照哪个属性进行比较来决定相对顺序。函数 `compByScore()` 定义为全局函数，`a.score>b.score` 将对象的 `score` 属性通过 `operator>` 进行比较，意味着对 `Student` 对象按照成绩从高到低排序。

排序完成之后，需要计算每个 `Student` 对象的名次，这可以通过算法 `distance(v.begin(), iter)+1` 确定该元素相对首元素的距离，并加 1 把该距离转化为该学生对象的排名。为了让成绩相同者具有相同的排名，上述程序首先通过算法 `equal_range(iter, v.end(), *iter, compByScore)` 查找容器 `v` 中所有具有相同成绩的元素，该算法返回与当前元素具有相同 `score` 取值的元素子区间（起点为 `pr.first`，终点为 `pr.second`），在该子区间中，所有元素都应该具有与第一个元素相同的名次，这通过赋值 `iter2->rank=iter->rank` 完成。

程序完整的输出结果如下：

```

before sort:
2008001 Jordan      92      0
2008002 Madonna    75      0
2008003 Jolie      86      0
2008004 Hanks      96      0
2008005 Cruise     80      0
2008006 Beckham    88      0
2008007 Figo       86      0
2008008 Avril      88      0
2008009 Raul       86      0
2008010 Kaka       80      0
after sort:
2008004 Hanks      96      1
2008001 Jordan     92      2
2008006 Beckham    88      3
2008008 Avril      88      4
2008003 Jolie      86      5
2008007 Figo       86      6
2008009 Raul       86      7
2008005 Cruise     80      8
2008010 Kaka       80      9
2008002 Madonna    75     10
after reRank:

```

2008004	Hanks	96	1
2008001	Jordan	92	2
2008006	Beckham	88	3
2008008	Avril	88	3
2008003	Jolie	86	5
2008007	Figo	86	5
2008009	Raul	86	5
2008005	Cruise	80	8
2008010	Kaka	80	8
2008002	Madonna	75	10

实验题目与提示

1. 讨论平面上直线之间的位置关系。

(1) 定义平面上直线结构类型 `Line`，以及平面上点结构类型 `Point`。

(2) 定义枚举类型 `LinePosRel`，用以描述平面上直线的 3 种位置关系（重合、平行、相交）。

(3) 讨论两条直线的相互位置关系，并求出相交时的交点。

(4) 把程序组织成多文件结构。

【提示】(1) 可以按照如下方式定义直线结构类型和点结构类型。

```
#01     struct Line {                                //描述直线类型
#02         double a, b, c;                          //直线方程:ax+by+c=0
#03     };
#04     struct Point {                                //描述点类型
#05         double x, y;                              //平面上的点
#06     };
```

(2) 可以定义如下枚举类型描述直线的位置关系。

```
enum LinePosRel {Coincident, Parallel, Acrossing}; //描述位置关系
```

(3) 可以定义如下函数计算两条直线 `a` 和 `b` 的位置关系。

```
LinePosRel Position(const Line& a, const Line& b, Point& p);
//判断位置关系
```

若直线 `a` 和 `b` 相交，则返回值为 `Acrossing`，同时引用参数 `p` 中存放的交点。若两直线平行，则返回值为 `Parallel`，但是参数 `p` 中不存放任何点。若两直线重合，则返回值为 `Coincident`，但是参数 `p` 中不存放任何点。在计算过程中，可以应用克莱姆法则计算两条直线的交点。

(4) 把直线和点的结构类型定义、枚举类型定义及函数 `Position()` 的原型放置在接口文件中，把函数 `Position()` 的定义放置在实现文件中。在 `main()` 函数中应用上述类型和函数进行计算判断。

2. 一元多项式及其根的求解。

(1) 定义多项式结构类型 `Polynomial`。

(2) 给定自变量一个值，计算多项式的值。

(3) 编写函数对多项式微分。

(4) 利用牛顿迭代法求该多项式的某根。

(5) 把程序组织成多文件结构。

【提示】(1) 首先需要定义如下表示多项式项的结构类型 `Item`。

```
#01 struct Item {
#02     double coefficient;           //项的系数
#03     int exponent;                //项的次数
#04 };
```

在该结构中, 成员 `coefficient` 表示该项的系数, 成员 `exponent` 表示该项的次数。在此基础上定义如下多项式结构类型。

```
#01 struct Polynomial {             //描述多项式类型
#02     list<Item> poly;             //存放于 list 容器中
#03 };
```

由于不同的多项式所拥有的项数是不同的, 因此选用 `list` 容器存放该多项式中所有的项。本程序假设多项式中的项都具有不同的指数, 否则需要一些额外的处理才能将其存放为合理的多项式。可以对结构类型 `Item` 定义如下函数 `ItemPrint()` 实现项的输出, 然后在此基础上定义如下函数 `PolyPrint()` 实现多项式的输出。

```
void ItemPrint(const Item& i);      //输出某项
void PolyPrint(const Polynomial& p); //输出整个多项式
```

(2) 首先定义如下函数 `ItemValue()` 计算当自变量取值为 x 时项 i 的值。

```
double ItemValue(const Item& i, double x); //计算某项的值
```

该函数返回所计算的该项的值。

接着定义如下函数 `PolyValue()` 计算多项式 p 在某一点 x 的值。

```
double PolyValue(const Polynomial& p, double x); //计算多项式的值
```

该函数返回所计算的多项式的值。在函数 `PolyValue()` 计算多项式值的过程中, 可以对多项式中的每一项调用函数 `ItemValue()`, 首先计算各项在给定点 x 的值, 然后把这些值累加即得到所求的多项式的值。

(3) 首先定义如下函数 `ItemDiff()` 计算某项 i 的微分。

```
Item ItemDiff(const Item& i);      //对某项微分
```

该函数返回该项的微分结果 (仍为一个项)。

接着定义如下函数 `PolyDiff()` 计算多项式 p 的微分多项式。

```
Polynomial PolyDiff(const Polynomial & p); //对多项式微分
```

该函数返回所计算的多项式的微分 (仍为一个多项式)。在计算多项式 p 的微分多项式过程中, 可以调用函数 `ItemDiff()` 得出每项的微分结果, 然后把它们存放为一个新的多项式 (应用 `list` 的方法 `push_back()`)。

(4) 以上已经实现计算多项式及其微分多项式、多项式在某点的值, 现在可以定义如下函数 `Root()`, 利用牛顿迭代法求多项式 p 在某初始点 `init` 附近的根。

```
double Root(const Polynomial& p, double init); //计算多项式的根
```

该函数返回所求的结果。计算过程按照牛顿迭代法进行, 在此不再赘述。

(5) 首先定义结构 `Item` 的接口文件, 在其中放置结构 `Item` 的定义, 以及函数 `ItemPrint()`、`ItemValue()`、`ItemDiff()` 的函数原型。然后定义结构 `Item` 的实现文件, 在其中放置函数 `ItemPrint()`、`ItemValue()`、`ItemDiff()` 的函数定义。对于结构 `Polynomial`

来说, 同样处理: 首先定义结构 `Polynomial` 的接口文件, 在其中放置结构 `Polynomial` 的定义, 以及函数 `PolyPrint()`、`PolyValue()`、`PolyDiff()`、`Root()` 的函数原型。然后定义结构 `Polynomial` 的实现文件, 在其中放置函数 `PolyPrint()`、`PolyValue()`、`PolyDiff()`、`Root()` 的函数定义。最后在 `main()` 函数中计算输出某多项式的根。

3. 定义学生信息结构类型 `StudentInfo`, 结构成员包括学号、姓名、性别、3 门课程成绩。

(1) 定义结构类型 `StudentInfo`, 其中姓名定义为 `string` 类型, 性别定义为枚举类型。

(2) 定义函数, 从键盘输入学生信息。

(3) 输出某个学生的 3 门课程成绩和平均课程成绩。

(4) 选用合适容器类存储多个 `StudentInfo` 变量。

(5) 计算并输出每门课程的全班平均分。

(6) 按平均课程成绩对班级学生由高到低排序, 然后输出。

(7) 把程序组织成多文件结构。

【提示】(1) 为描述性别, 首先定义枚举类型 `Gender`, 然后分别定义函数 `InputGender()`、`PrintGender()`, 实现该类型数据的输入和输出。

```
enum Gender {female, male};           //描述性别
void InputGender(Gender& g);           //输入性别
void PrintGender(const Gender& g);     //输出性别
```

接着定义结构类型 `StudentInfo`, 描述学生所有的信息。

```
#01  #define CourseNum 3                //课程门数
#02  struct StudentInfo {               //描述学生所有的信息
#03      string id;                     //学号
#04      string name;                   //姓名
#05      Gender gender;                 //性别
#06      double score[CourseNum + 1];  //课程成绩
#07  };
```

为了灵活处理课程成绩, 以上定义宏 `CourseNum` 表示课程门数, 同时在表示成绩的数组 `score` 中多一个元素用以存放所有课程的平均成绩。

(2) 定义函数 `InputStudentInfo()`, 实现学生信息的输入。

```
void InputStudentInfo(StudentInfo& si); //输出学生信息
```

在输入性别时借用函数 `InputGender()`。在输入课程成绩后, 同时计算其平均成绩。

(3) 定义函数 `PrintStudentInfo()`, 实现学生信息的输出。

```
void PrintStudentInfo(const StudentInfo& si); //输出学生信息
```

(4) 可以选用容器 `vector` 存放多个学生的信息, 这样存放后, 实际上形成了班级结构类型, 因此定义班级结构类型 `Class`。

```
#01  struct Class {                     //描述班级类型
#02      vector<StudentInfo> vs;        //存放于 vector 容器中
#03  };
```

同时定义函数 `InputClassInfo()` 输入整个班级所有学生的信息, 定义函数 `PrintClassInfo()` 输出所有学生的信息。

```
void InputClassInfo(Class& c);          //输入班级学生的信息
void PrintClassInfo(const Class& c);    //输出班级学生的信息
```

(5) 定义函数 `AvgOfCourse()` 计算某门课程的全班平均成绩。

```
double AvgOfCourse(int i); //计算某门课程的平均成绩
```

由于课程有多门, 而且成绩存放在一个数组中, 因此用一个表示成绩下标的参数 `i` 指明所计算的课程。

(6) 定义函数 `Sort()` 对全班所有学生按照平均课程成绩排序。

```
void Sort(Class& c); //按照平均课程成绩排序
```

由于所有学生信息存放在 `vector` 容器中, 为了借用 STL 算法 `sort()` 实现对容器元素的排序, 需要对结构类型 `StudentInfo` 定义如下函数 `CmpAvgScore()` 表明排序的准则。

```
bool CmpAvgScore(const StudentInfo& a, const StudentInfo& b); //排序的准则
```

(7) 对类型 `Gender`、`StudentInfo`、`Class` 分别定义接口文件和实现文件存放各自的声明和函数定义。对于上述函数, 参数类型是哪个类型, 就应该在该类型的接口文件和实现文件中分别声明和定义该函数。

实验 6 类与对象的定义

实验目的与要求

1. 实验目的

- (1) 掌握类定义的语法。
- (2) 掌握构造函数、析构函数的概念。
- (3) 掌握几个特殊的构造函数。
- (4) 理解对象数组和对象指针的概念。
- (5) 理解类的复合关系。

2. 实验要求

- (1) 能够正确定义类的类型。
- (2) 能够正确分析并定义构造函数、析构函数。
- (3) 能够根据需求定义用于不同目的的构造函数。
- (4) 能够正确分析并应用对象数组和对象指针。
- (5) 能够正确定义类以表示不同概念之间的复合关系。

实验过程与示例

在本次实验中，需要重点练习构造函数、析构函数的定义内容，能够根据需要在自定义类中提供默认构造函数、复制构造函数/转移构造函数、复制赋值运算符函数/转移赋值运算符函数等成员函数，并理解各自适用的场合。能够正确设计描述类之间复合关系的程序。

需要说明的是，在组织包含类、结构等自定义数据类型的程序时，一般需要按照“接口与实现相分离”的原则，把程序组织成多文件结构，如表 1.6.1 所示，在这种结构中，每个类通常对应于一个头文件（类的声明）和一个源文件（成员函数的定义）。

表 1.6.1 程序的多文件结构

文件名称	文件类型	存放内容
MyClass.h	类的接口文件	类的声明；防止多重包含的条件编译指令
MyClass.cpp	类的实现文件	类成员函数的定义；必要的文件包含指令
main.cpp	类的应用文件	main()函数；必要的文件包含指令

除与本书配套的主教材上的相关例题外，以下是一些较典型的示例，它们分别对数据成员作为简单变量、字符数组等不同情况下的构造函数、析构函数的定义进行了示范，最后举例说明了类的复合关系的应用。

【例 1.6.1】复数类 Complex 的应用示例。

定义复数类，实现复数的加、减、乘、除运算。按照“接口与实现相分离”的原则，本程序由 Complex.h、Complex.cpp 和 main.cpp 这 3 个文件组成。

头文件 Complex.h 中存放类 Complex 的定义、成员函数的原型声明，该文件内容如下：

```
#01     #ifndef COMPLEX_CLASS
#02     #define COMPLEX_CLASS
#03
#04     #include <iostream>
#05
#06     class Complex {                                //定义复数类
#07     private:
#08         double real, imag;                          //实部,虚部
#09     public:
#10         Complex(double r = 0, double i = 0) noexcept; //常规构造函数
#11         Complex(const Complex& c);                    //复制构造函数
#12         Complex(Complex&& c) noexcept;                //转移构造函数
#13         ~Complex() noexcept;                          //析构函数
#14
#15         Complex& operator = (const Complex& rhs); //复制赋值运算符函数
#16         Complex& operator = (Complex&& rhs) noexcept; //转移赋值运算符函数
#17
#18         void setreal(double r) { real = r; }          //real 的 Set 函数
#19         void setimag(double i) { imag = i; }          //imag 的 Set 函数
#20         double getreal() const { return real; }       //real 的 Get 函数
#21         double getimag() const { return imag; }       //imag 的 Get 函数
#22
#23         void print(std::ostream& os = std::cout) const; //输出函数
#24
#25         double magnitude() const;                     //计算模长
#26         double angle() const;                         //计算幅角
#27         Complex conjugate() const;                    //计算共轭
#28         Complex make_by_mag_ang(double mag, double ang) const; //重构复数
#29         Complex pow(int n) const;                     //计算乘方
#30
#31         Complex plus(const Complex& rhs);              //两个复数的加法
#32         Complex minus(const Complex& rhs);            //两个复数的减法
#33         Complex multiplies(const Complex& rhs);       //两个复数的乘法
#34         Complex divides(const Complex& rhs);          //两个复数的除法
#35     };
#36
#37     #endif // COMPLEX_CLASS
```

上述头文件中定义了类 Complex，其中声明了数据成员及常规构造函数、复制构造

函数、转移构造函数、复制赋值运算符函数、转移赋值运算符函数、析构函数等成员函数，提供了针对两个数据成员的 Set() / Get() 函数，复数相关的计算函数及算术加、减、乘、除运算的函数。请仔细分析各个函数原型的设置方式。

源文件 Complex.cpp 中存放类 Complex 成员函数的定义，该文件内容如下：

```
#01     #include <cmath>
#02     #include <cassert>
#03     #include "Complex.h"
#04
#05     Complex::Complex(double r, double i) noexcept //常规构造函数
#06         :real(r), imag(i) {                      //成员初始化值列表
#07     }
#08     Complex::Complex(const Complex& c)              //复制构造函数
#09         :real(c.real), imag(c.imag) {              //复制样本的数据成员
#10     }
#11     Complex::Complex(Complex&& c) noexcept          //转移构造函数
#12         : real(std::move(c.real)), imag(std::move(c.imag)) {
#13         //转移样本的数据成员
#14     }
#15     Complex::~~Complex() noexcept {                 //析构函数
#16     }
#17     Complex& Complex::operator = (const Complex& rhs) { //复制赋值运算符函数
#18         real = rhs.real;                            //复制右操作数的成员 real
#19         imag = rhs.imag;                            //复制右操作数的成员 imag
#20         return *this;                               //返回左操作数
#21     }
#22     Complex& Complex::operator = (Complex&& rhs) noexcept {
#23         //转移赋值运算符函数
#24         real = std::move(rhs.real);                  //转移右操作数的成员 real
#25         imag = std::move(rhs.imag);                  //转移右操作数的成员 imag
#26         return *this;                               //返回左操作数
#27     }
#28     void Complex::print(std::ostream& os) const { //输出函数
#29         os << real                                  //输出实部
#30             << std::showpos << imag << "i"          //输出虚部和复数符号
#31             << std::noshowpos << std::endl;         //输出换行
#32     }
#33     double Complex::magnitude() const {              //计算模长
#34         return std::sqrt(real * real + imag * imag);
#35     }
#36     double Complex::angle() const {                  //计算幅角
#37         return std::atan(imag / real);
#38     }
#39     Complex Complex::conjugate() const {              //共轭复数
#40         return Complex(real, -imag);
#41     }
#42     //根据模长和幅角构造一个复数
#43     Complex Complex::make_by_mag_ang(double mag, double ang) const {
#44         double x = mag * cos(ang);                  //计算实部
```

```
#45         double y = mag * sin(ang);           //计算虚部
#46         return Complex(x, y);               //转成复数
#47     }
#48     Complex Complex::pow(int n) const {        //计算乘方
#49         double mag = magnitude();             //计算模长
#50         double ang = angle();                 //计算幅角
#51         mag = std::pow(mag, n);               //模长乘方
#52         ang *= n;                             //幅角乘倍
#53         return make_by_mag_ang(mag, ang);     //转换为复数
#54     }
#55
#56     Complex Complex::plus(const Complex& rhs) { //复数加法
#57         Complex c;                             //临时变量存放结果
#58         c.real = real + rhs.real;              //计算实部
#59         c.imag = imag + rhs.imag;             //计算虚部
#60         return c;                             //返回结果
#61     }
#62     Complex Complex::minus(const Complex& rhs) { //复数减法
#63         Complex c;                             //临时变量存放结果
#64         c.real = real - rhs.real;              //计算实部
#65         c.imag = imag - rhs.imag;             //计算虚部
#66         return c;
#67     }
#68     Complex Complex::multiplies(const Complex& rhs) { //复数乘法
#69         Complex c;                             //以下分别计算实部和虚部
#70         c.real = real * rhs.real - imag * rhs.imag;
#71         c.imag = real * rhs.imag + imag * rhs.real;
#72         return c;                             //返回结果
#73     }
#74     Complex Complex::divides(const Complex& rhs) { //复数除法
#75         double m = rhs.real * rhs.real + rhs.imag * rhs.imag;
#76         assert(m != 0);                       //确保除数不为零
#77
#78         Complex c;
#79         double r = real * rhs.real + imag * rhs.imag;
#80         double i = imag * rhs.real - real * rhs.imag;
#81
#82         c.real = ( r / m );                   //计算实部
#83         c.imag = ( i / m );                   //计算虚部
#84         return c;                             //返回结果
#85     }
```

源文件 **Complex.cpp** 对类中的成员函数在类定义体外进行定义。各构造函数、赋值运算符函数按照语法规则和功能要求加以实现。各工具函数按照复数计算规则进行计算。

源文件 **main.cpp** 中存放 **main()** 函数，该文件内容如下：

```
#01     #include "Complex.h"
#02
#03     int main() {
```

```

#04      Complex a(14, 2);
#05      Complex b(a);
#06      b.setreal(3); b.setimag(4);      b.print();      //3+4i
#07      Complex c(std::move(a));      c.print();      //14+2i
#08
#09      c = a.plus(b);      c.print();      //17+6i
#10      c = a.minus(b);      c.print();      //11-2i
#11      c = a.multiplies(b);      c.print();      //34+62i
#12      c = a.divides(b);      c.print();      //2-2i
#13
#14      std::cout << a.multiplies(a.conjugate()).getreal() << std::endl;
#15                                     //200
#16      std::cout << a.magnitude() * a.magnitude() << std::endl; //200
#17
#18      c = a.pow(4); c.print();      //33728+21504i
#19      b = a.multiplies(a);
#20      b = b.multiplies(a);
#21      b = b.multiplies(a);
#22      b.print();      //33728+21504i
#23      }

```

源文件 `main.cpp` 中的 `main()` 函数对上述功能进行了测试。程序运行结果注释在各行之后。其中, `Complex b(a)` 测试复制构造函数, `Complex c(std::move(a))` 测试转移构造函数, 其他各函数调用分别测试相关工具函数。算术运算表达式, 如 `c=a.plus(b)` 测试转移赋值运算符函数。表达式 `c=a.pow(4)` 测试计算模长、幅角、转移构造函数、转移赋值运算符函数等。

【例 1.6.2】学生类 `Student` 的应用示例。

定义学生类, 计算所有学生的平均成绩。

```

#01      #include <iostream>
#02      #include <algorithm>      //for for_each
#03      #include <functional>      //for mem_fn
#04      #include <string>      //for string
#05
#06      class Student {      //定义学生类型
#07      private:
#08          std::string name;      //姓名
#09          double score;      //成绩
#10      public:
#11          Student(const char* sn = "", double ds = 0) noexcept //构造函数
#12              :name(sn), score(ds) {      //成员初始化值列表
#13          }
#14          Student(const Student& s)      //复制构造函数
#15              :name(s.name), score(s.score) { //成员初始化值列表
#16          }
#17          Student(Student&& s) noexcept      //转移构造函数
#18              :name(std::move(s.name)),      //成员初始化值列表
#19              score(std::move(s.score)) {
#20      }

```

```

#21     Student& operator = (const Student& rhs) { //复制赋值运算符函数
#22         if (this != &rhs) {                //防止自赋值
#23             name = rhs.name;                //复制 name
#24             score = rhs.score;              //复制 score
#25         }
#26         return *this;                        //返回左操作数
#27     }
#28     Student& operator=(Student&& rhs) noexcept { //转移赋值运算符函数
#29         if (this != &rhs) {                //防止自赋值
#30             name = std::move(rhs.name);      //转移 name
#31             score = std::move(rhs.score);    //转移 score
#32         }
#33         return *this;                        //返回左操作数
#34     }
#35     double GetScore() const { return score; } //获取 score
#36     void Print() { std::cout << name << ":\t" << score << "\n"; }
#37 };
#38 //用于算法 sort 的参数
#39 bool compByScore(const Student& a, const Student& b) { //定义比较准则
#40     return a.GetScore() < b.GetScore();      //按照 score 比较
#41 }
#42
#43 int main() {
#44     const size_t N = 4;                      //定义常量以表示数组长度
#45     Student s[N];                            //生成对象数组
#46     s[0] = Student("Tom", 85);               //为第一个元素赋值
#47     s[1] = Student("Jerry", 90);             //为第二个元素赋值
#48     s[2] = Student("Goofy", 70);             //为第三个元素赋值
#49     s[3] = Student("Mickey", 85);            //为第四个元素赋值
#50
#51     std::cout << "before sort:\n";           //以下输出所有对象信息
#52     std::for_each(s, s + N, std::mem_fn(&Student::Print));
#53     std::sort(s, s + N, compByScore);        //按成绩排序
#54     std::cout << "after sort:\n";           //以下输出排序结果
#55     std::for_each(s, s + N, std::mem_fn(&Student::Print));
#56
#57     double avg = 0;                          //计算平均成绩
#58     for (size_t i = 0; i != N; ++i)
#59         avg += s[i].GetScore();
#60     avg /= N;
#61
#62     std::cout << "average score is " << avg << std::endl;
#63 }

```

上述程序定义学生类 `Student`，并在其中提供了构造函数、复制构造函数、转移构造函数和复制赋值运算符函数、转移赋值运算符函数。为了方便访问私有数据成员 `score`，`Student` 类中提供了公有成员函数 `GetScore()`。

`main()` 函数中定义了对象数组，并通过赋值运算符给每个对象元素赋以确定的值。`for_each(s, s+N, mem_fn(&Student::Print))` 运用算法 `for_each()` 为数组中的每个元素调用

分析输出结果, 请思考下列问题:

(2) `main()`函数中的 `Student s[N];`语句可生成 4 个对象元素, 需要调用类的默认构造函数, 请验证。

(4) `main()`函数中用到算法 `for_each()`、算法 `sort()`对数组元素进行操作,它们的第三个参数分别采用不同的形式定义和传入,请分析并熟记这两种常用的情形(全局函数和成员函数作为算法的参数)。

复合是两个类之间相互关联的一种常见形式。下面是关于类的复合示例。

```
#01 #include <cmath>
#02 #include "xr.hpp"
#03
#04 class Point { //定义点类
#05 private:
#06     double x, y; //横纵坐标
#07 public:
#08     Point(double a, double b) :x(a), y(b) //构造函数
#09     {}
#10     Point(const Point& p) { //复制构造函数
#11         if (this != &p) { //防止自复制
#12             x = p.x; //复制样本的 x
#13             y = p.y; //复制样本的 y
#14         }
#15     }
#16     Point& operator = (const Point& rhs) { //赋值运算符函数
#17         if (this != &rhs) { //防止自赋值
#18             x = rhs.x; //复制右操作数的 x
#19             y = rhs.y; //复制右操作数的 y
#20         }
#21         return *this; //返回左操作数
#22     }
#23     double GetX() const {return x;} //读取 x
#24     double GetY() const {return y;} //读取 y
#25 };
#26
#27 class Line { //定义直线类
#28 private:
```

```
#29         Point start, end;                                //对象成员:起点和终点
#30     public:
#31         Line(double a, double b, double c, double d) //构造函数
#32             :start(a, b), end(c, d)                  //成员初始化列表
#33         {}
#34         Line(const Point& a, const Point& b)           //构造函数
#35             :start(a), end(b)                         //成员初始化列表
#36         {}
#37         Line(const Line& l)                           //复制构造函数
#38             :start(l.start), end(l.end)               //成员初始化列表
#39         {}
#40         Line& operator = (const Line& rhs) {           //赋值运算符函数
#41             if (this != &rhs) {                       //防止自赋值
#42                 start = rhs.start;                    //取得右操作数的起点
#43                 end = rhs.end;                        //取得右操作数的终点
#44             }
#45             return *this;                             //返回左操作数
#46         }
#47         double Length() const {                       //计算线段长度
#48             double dx = start.GetX() - end.GetX();
#49             double dy = start.GetY() - end.GetY();
#50             return sqrt(dx * dx + dy * dy);
#51         }
#52         Point MidPoint() const {                     //计算线段中点
#53             double x = (start.GetX() + end.GetX()) / 2; //横坐标中点
#54             double y = (start.GetY() + end.GetY()) / 2; //纵坐标中点
#55             return Point(x, y);                      //返回结果
#56         }
#57     };
#58
#59     int main()
#60     {
#61         Point a(1, 2), b(1, 4);                      //生成两个点
#62         Line la(a, b);                                //构造一条直线
#63         Line lb(0, 1, 0, 5), lc(lb);                 //构造另两条直线
#64
#65         xr(la.Length()); xr(lb.Length()); xr(lc.Length());
#66
#67         lc = Line(la.MidPoint(), lb.MidPoint());
#68         xr(lc.Length());
#69     }
```

上述程序首先定义 Point 类,并在其中提供了必要的操作。由于一条直线由两个点组成,Line 类与 Point 类之间满足“has-a”的关系,因此,类 Line 以 Point 类的对象作为数据成员。

分析程序输出结果,请思考下列问题:

(1) main()函数中生成对象 a、b、la、lb 和 lc 时分别调用了哪些函数,其先后次序如何?

(2) 在定义 Line 类的 3 个构造函数及赋值运算符函数时, 对应调用了对象成员 start 和 end 的构造、复制和赋值方法, 请分析并掌握 Line 类函数的定义方法。

(3) 以多文件结构组织上述程序, 记住让每个类对应两个文件 (头文件存放类的声明、源文件存放类中成员函数的定义)。

实验题目与提示

1. 参考例 1.6.2, 定义员工类 Worker, 其属性有工号 (id)、姓名 (name)、年龄 (age)、性别 (gender)、工资 (salary) 等。

(1) 定义该类的构造函数、复制构造函数/转移构造函数和复制赋值运算符函数/转移赋值运算符函数, 并输出各对象的所有信息, 请分析各函数调用的场合。

(2) 定义对象数组存放多个对象元素。

(3) 按照工资金额从高到低排序。

(4) 计算工资最高者、最低者及平均工资。

(5) 比较男性员工和女性员工的平均工资。

【提示】(1) 可以如下定义类 Worker。

```
#01     class Worker {                                //员工类型
#02     private:
#03         string id;                                //工号
#04         string name;                              //姓名
#05         int age;                                   //年龄
#06         Gender gender;                             //性别
#07         double salary;                             //工资
#08     };

```

其中, 性别 Gender 是如下定义的枚举类型。

```
#01     enum Gender {female, male};

```

将该枚举类型定义在类 Worker 之前, 同时在类中声明如下成员函数。

```
#01     public:
#02         Worker(const char* sid = "", const char* sn = "", int na = 0,
#03             Gender g = female, double sa = 0) noexcept; //构造函数
#04         Worker(const Worker& w);                        //复制构造函数
#05         Worker(Worker&& w) noexcept;                    //转移构造函数
#06         Worker& operator = (const Worker& rhs);        //复制赋值运算符函数
#07         Worker& operator = (Worker&& rhs) noexcept;    //转移赋值运算符函数
#08         void Print(std::ostream& os = std::cout);      //输出对象信息

```

为了方便后续对数据成员 salary 的访问, 提供 Get() 函数, 如下所示:

```
double GetSalary() const {return salary; }           //定义在类 Worker 中

```

(2) 在 main() 函数中定义对象数组 w。

```
Worker w[4];

```

然后向数组元素中通过赋值运算给 4 个对象数据赋值, 或者运用 vector 容器存放多个对象元素。

```
vector<Worker> vw;

```

然后通过 `push_back()` 方法向容器中填充 4 个对象数据。

(3) 可以应用 STL 算法 `sort` 实现排序, 为此需要首先定义对象的比较准则。

```
bool cmpWorkerBySalary(const Worker& a, const Worker& b);
```

把这个函数定义在 `main()` 函数之前即可, 在该函数中表明是按照数据成员 `salary` 进行比较的。把该函数填作算法 `sort()` 的参数, 即可实现按照工资排序。

(4) 分别运用 STL 算法 `max_element()`、`min_element()` 和 `accumulate()` 计算工资最高者、最低者和总和, 只要参考例 1.6.2 的 `main()` 函数定义适当的比较准则和加法准则即可。

(5) 对数组元素依次判断其性别取值, 按照不同的性别分别进行累加, 然后求和计算平均值。

2. 参考例 1.6.3, 定义点类 `Point` 和矩形类 `Rectangle`。

(1) 在每个类中提供必要的成员函数构造对象、输出信息。

(2) 计算两个点之间的距离。

(3) 计算矩形的面积和中心点。

(4) 判断点与矩形的位置关系。

【提示】(1) 仿照例 1.6.3 定义点 `Point` 类, 如下定义矩形类 `Rectangle`。

```
#01      class Rectangle {                                //矩形类型
#02      private:
#03          Point bottomLeft;                            //左下角点
#04          Point upRight;                               //右上角点
#05      };
```

描述一个矩形, 只需要它的左下角点和右上角点即可。

(2) 在类 `Point` 中提供如下成员函数计算两点之间的距离。

```
double Length(const Point& p) const;
```

(3) 在类 `Rectangle` 中定义如下函数 `Area()` 计算矩形的面积, 函数 `CenterPoint()` 计算矩形的中心点。

```
double Area() const;
Point CenterPoint() const;
```

在计算矩形面积时, 首先根据两点距离函数 `Length()` 计算矩形的两个边长, 然后计算矩形面积。在计算矩形中心点时, 计算出左下角点和右上角点的中点, 即可得出矩形的中心点。

(4) 点可能在矩形的内部、外部或刚好位于矩形的边上, 这些都可以通过其坐标的位置关系加以判断。首先定义如下枚举类型 `Rel` 描述这 3 种位置关系。

```
enum Rel {PtOnRect, PtInRect, PtOutRect};
```

然后定义如下全局函数 `position()` 判断点 `p` 和矩形 `r` 的位置关系。

```
Rel position(const Point& p, const Rectangle& r);
```

该函数需要访问类 `Point` 和 `Rectangle` 的私有成员, 请为它们提供公有的 `Get()` 成员函数。

3. 模拟学校的构成。

(1) 定义学生类 `Student`, 每个学生有学号、姓名、性别、年龄、成绩等信息。

(2) 定义班级类 `Class`, 每个班级有专业名称、班级编号、学生人数等信息, 以及 N

个学生。

(3) 定义学校类 `University`，其属性有学校名称、所在城市等信息，假设该学校有 `M` 个班级。

(4) 统计该校总人数、男生人数、女生人数。

(5) 找出该校成绩最好的前 10 个学生予以奖励。

【提示】本题主要用到类的复合关系。首先定义学生类 `Student`；然后定义班级类 `Class`，它需要 `Student` 数组作为本类的数据成员；接着定义学校类 `University`，它需要 `Class` 数组作为本类的数据成员。为每个类提供基本的输入、输出函数，以及必要的访问函数。

实验 7 类与对象的几个主题

实验目的与要求

1. 实验目的

- (1) 理解关键字 `this` 的用法。
- (2) 理解关键字 `const` 的用法。
- (3) 理解关键字 `new/delete` 的用法。
- (4) 理解关键字 `friend` 的用法。
- (5) 理解关键字 `static` 的用法。

2. 实验要求

- (1) 能够应用 `this` 访问成员、防止自复制、自赋值。
- (2) 能够在定义成员函数时正确应用 `const`。
- (3) 能够正确构造、复制指针型数据成员。
- (4) 能够正确定义并应用类的友元函数。
- (5) 能够正确应用类的静态成员。

实验过程与示例

在本次实验中, 需要继续练习类的构造函数、析构函数等重要成员函数的定义内容, 同时加强对几个关键字用法的理解, 能够在类的复制构造函数/转移构造函数和复制赋值运算符函数/转移赋值运算符函数中正确应用关键字 `this`、定义类的 `const` 成员函数, 能够结合教材中关于 `String` 类的定义理解并正确定义具有指针型数据成员的类的构造函数、析构函数、复制构造函数和赋值运算符函数。会利用友元机制访问类的私有数据成员等。

除与本书配套的主教材上的相关例题外, 以下是一些较典型的示例, 它们分别对 `this`、`const`、`static`、`friend` 等关键字的用法进行了示范。

【例 1.7.1】日期类 `Date` 的应用示例。

定义日期类 `Date`, 判断是否是闰年, 计算某个日期相对基准日期的相隔天数, 并进行相反方向的计算。下面以多文件结构给出程序代码。

头文件 `Date.h` 的内容如下:

```
#01      #ifndef DATE_CLASS
```

```

#02     #define DATE_CLASS
#03
#04     class Date
#05     {
#06     private:
#07         int year, month, day;
#08     private:
#09         //以下定义计算的基准,即从 1970 年 1 月 1 日开始
#10         enum {YearBase = 1970};           //年的基准
#11         enum {MonthBase = 1};             //月的基准
#12         enum {DayBase = 1};               //日的基准
#13
#14         static int nDaysofMonth[13];       //列举每个月的天数
#15         void AdjustFeb() const;           //针对闰年,修改 2 月份天数
#16     public:
#17         Date(int y=YearBase, int m=MonthBase, int d=DayBase) noexcept;
#18         Date(const Date& d);               //复制构造函数
#19         Date& operator = (const Date& rhs); //赋值运算符函数
#20
#21         void SetDate(int y, int m, int d); //设置日期
#22         void Print() const;
#23
#24         static bool IsLeapYear(int y);      //判断是否是闰年
#25         static int DaysOfThisYear(int y);   //该年的总天数
#26
#27         int Date2Days() const; //将日期转换成从基准开始所经过的天数
#28         static Date Days2Date(int n); //从基准开始,经过 n 天之后,是什么日期
#29     };
#30     #endif //DATE_CLASS

```

上述程序定义了日期类 `Date`。同时定义了日期的基准为 1970 年 1 月 1 日。为了便于计算, `Date` 类以 `static` 数据成员 `nDaysofMonth` 提供数组列举了每月的天数, 而函数 `AdjustFeb()` 根据闰年的情况修改二月份的天数。除基本的构造、复制和赋值运算符外, 类 `Date` 还提供了设置一个日期的方法 `SetDate()`、输出日期的方法 `Print()`。函数 `IsLeapYear()` 判断某年是否是闰年, 函数 `DaysOfThisYear()` 返回某年的天数。函数 `Date2Days()` 和 `Days2Date()` 是类中最为重要的两个方法, 前一个函数把日期转化为相对基准的天数, 后一个函数做相反的转换。

源文件 `Date.cpp` 的内容如下:

```

#01     #include <iostream>
#02     #include <cassert>
#03     #include "Date.h"
#04     //列举每个月的天数: 为符合习惯,用月份作为下标
#05     int Date::nDaysofMonth[13] = {0,
#06         31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
#07
#08     Date::Date(int y, int m, int d) noexcept {
#09         SetDate(y, m, d);
#10     }

```

```
#11 Date::Date(const Date& d) {
#12     if (this != &d) {
#13         year    = d.year;
#14         month    = d.month;
#15         day      = d.day;
#16     }
#17 }
#18 Date& Date::operator = (const Date& rhs) {
#19     if (this != &rhs) {
#20         year    = rhs.year;
#21         month    = rhs.month;
#22         day      = rhs.day;
#23     }
#24     return *this;
#25 }
#26 bool Date::IsLeapYear(int y) {
#27     return (y % 4 == 0 && y % 100 != 0) || (y % 400 == 0);
#28 }
#29 int Date::DaysOfThisYear(int y) {           //该年的总天数
#30     if (IsLeapYear(y)) return 366;
#31     else return 365;
#32 }
#33 void Date::SetDate(int y, int m, int d) {
#34     year = y >= YearBase ? y : YearBase; //设置年份
#35     month = m >= 1 && m <= 12 ? m : MonthBase; //设置月份
#36
#37     this->AdjustFeb();                     //考虑闰年的影响
#38     day = d >= 1 && d <= nDaysofMonth[month] ? d : DayBase;
#39 }
#40 void Date::AdjustFeb() const {
#41     if (IsLeapYear(year))                 //若是闰年
#42         nDaysofMonth[2] = 29;             //设为 29 天
#43     else                                   //不是闰年
#44         nDaysofMonth[2] = 28;             //仍改回 28 天
#45 }
#46 int Date::Date2Days() const {
#47     this->AdjustFeb();                     //考虑闰年的影响
#48
#49     int s = 0;                             //累加求和,先初始化为 0
#50     s += day;                             //先计算经过的天数
#51     for (int m = 1; m < month; ++m)
#52         s += nDaysofMonth[m];             //再计算经过整月的天数
#53     for (int y = YearBase; y < year; ++y)
#54         s += DaysOfThisYear(y);          //最后计算经过整年的天数
#55     return s;                             //返回求出的天数和
#56 }
#57 Date Date::Days2Date(int n) {
#58     Date d;
#59     int y = YearBase;
```

```

#60         while (n >= DaysOfThisYear(y)) { //判断 n 是否超过整年的天数
#61             n -= DaysOfThisYear(y); //从中将其去除
#62             ++y; //得到经过的年数
#63             if (n == DaysOfThisYear(y)) //若剩下的天数刚好够一整年
#64                 break; //恰好为年末,不用再减
#65         }
#66         d.year = y; //y 为所求年份
#67         d.AdjustFeb(); //考虑闰年的影响
#68
#69         int m = 1;
#70         while (n >= nDaysofMonth[m]) { //判断剩余天数是否超过整月
#71             n -= nDaysofMonth[m]; //从中将其去除
#72             ++m; //得到经过的月数
#73             if (n == nDaysofMonth[m]) //若剩下的天数刚好够一个月
#74                 break; //恰好为月末,不用再减
#75         }
#76         d.month = m; //得到经过的月份
#77         d.day = n; //剩余则为经过的天数
#78         return d; //返回得到的当前日期对象
#79     }
#80     void Date::Print() const {
#81         std::cout << this->year << "-" << this->month
#82             << "-" << this->day << std::endl;
#83     }

```

源文件 main.cpp 的内容如下:

```

#01     #include "Date.h"
#02     #include "xr.hpp"
#03
#04     int main()
#05     {
#06         Date d(2008, 8, 8); //生成一个日期对象
#07         xrv(d.Print()); //输出
#08         xrv(d.Date2Days()); //计算相对基准的天数
#09
#10         d = Date::Days2Date(d.Date2Days()); //把上面的天数再转换为日期
#11         xrv(d.Print()); //应该仍为 2008-8-8
#12
#13         d = Date::Days2Date(2000); //从基准经过 2000 天
#14         xrv(d.Print());
#15     }

```

分析程序输出结果,并思考下列问题:

(1) 上述程序把列举每个月天数的数组 nDaysofMonth 设为类的 static 数据成员,请考虑此举的必要性。同时成员函数 IsLeapYear()、DaysOfThisYear()、Days2Date() 都设为类的 static 成员函数,请思考其中原委。

(2) 在类 Date 的成员函数中,函数 AdjustFeb()、Print()、Date2Days() 是 const 成员函数,请分析为什么要把它们声明为 const 成员函数。

(3) 成员函数 Days2Date() 计算从基准日期经过 n 天之后的日期,而成员函数

Date2Days()计算从基准日期到当前日期经过了多少天,这两个计算过程是相反的。请分析它们的实现过程,并以它们为基础实现本讲后面的习题。

【例 1.7.2】学生类 Student 的应用示例。

本题综合练习对象数组及 friend()函数的定义和应用。

```
#01  #include <iostream>
#02  #include <algorithm>
#03  #include <functional>
#04  #include <vector>
#05  #include <string>
#06
#07  class Student {                                //定义学生类型
#08  private:
#09      int sid;                                    //学号
#10      std::string name;                          //姓名
#11      double score;                              //成绩
#12  public:
#13      Student(int id, const char* str, double s)
#14          :sid(id), name(str), score(s)
#15      {}
#16      void Print() const {
#17          std::cout << this->sid << "\t" << this->name << "\t"
#18              << this->score << std::endl;
#19      }
#20      friend bool compBySID(const Student& a, const Student& b);
#21      friend bool compByScore(const Student& a, const Student& b);
#22  };
#23  bool compBySID(const Student& a, const Student& b) { //按学号比较
#24      return a.sid < b.sid;
#25  }
#26  bool compByScore(const Student& a, const Student& b) { //按成绩比较
#27      return a.score > b.score;
#28  }
#29  int main() {
#30      std::vector<Student> vs;
#31      vs.push_back(Student(200803, "Tom", 85));
#32      vs.push_back(Student(200802, "Jerry", 90));
#33      vs.push_back(Student(200801, "Goofy", 70));
#34      vs.push_back(Student(200805, "Mickey", 85));
#35      vs.push_back(Student(200804, "Minnie", 96));
#36
#37      std::cout << "before sort:\n"; //输出容器元素
#38      std::for_each(vs.begin(), vs.end(),
#39          std::mem_fun_ref(&Student::Print));
#40
#41      std::cout << "sort by sid:\n";
#42      std::sort(vs.begin(), vs.end(), compBySID); //按学号排序
#43      std::for_each(vs.begin(), vs.end(),
#44          std::mem_fun_ref(&Student::Print));
```

```

#45
#46         std::cout << "sort by score:\n";
#47         std::sort(vs.begin(), vs.end(), compByScore); //按成绩排序
#48         std::for_each(vs.begin(), vs.end(),
#49                         std::mem_fun_ref(&Student::Print));
#50     }

```

本题的重点在于如何定义类的友元函数作为算法 `sort` 的排序准则,之所以要定义为友元函数,是为了方便地访问类的数据成员。

分析上述程序的输出结果,并思考:为了按照不同的准则排序,需要定义排序准则 `compBySID()` 和 `compByScore()`,该准则通常是带有两个参数、且返回类型为 `bool` 的函数或函数对象,请参考上述程序,定义按照姓名先后排序的准则。

实验题目与提示

参考例 1.7.1, 请继续增加类 `Date` 的功能。

(1) 将日期转换为字符串,转换的格式由格式字符串说明,如格式字符串 `"M/d/yy"` 表示输出日期为 `"5/30/21"` (月份和日期数字不带有前导 0, 年份表示为两位数字), 格式字符串 `"MM/dd/yyyy"` 表示输出日期为 `"05/30/2021"` (月份和日期数字带有前导 0, 年份表示为 4 位数字), 格式字符串 `"MMMM dd, yyyy"` 表示输出日期为 `"May 30, 2021"`。

(2) 提供一个成员函数,返回用 `time.h` 头文件中的标准库函数读取的当前系统日期。

(3) 计算当前日期是该年的第几天,是该年的第几个星期。

(4) 分别计算当前日期的后一个日期和前一个日期。

(5) 计算任意两个日期之间相隔的天数。

(6) 计算从当前日期再过 `n` 天之后的日期。

(7) 比较两个日期是否相等及其先后关系。

【提示】(1) 定义如下成员函数 `toString()`, 它带有一个字符串类型的参数表示输出的格式, 其函数原型如下:

```
string toString(string_view format_flag) const;
```

在函数实现过程中,应用 `sprintf()` 函数把 `int` 类型转换为字符串内容,然后把所得 C 字符串转换为 `string` 对象并返回。

在类 `Date` 的定义体中添加如下函数原型:

```

static Date Today();           //获取系统当前日期
int nthDay() const;           //当前日期是该年的第几天
int nthWeek() const;          //当前日期是该年的第几个星期
Date NextDate() const;        //计算当前日期的后一天
Date PrevDate() const;        //计算当前日期的前一天
int DiffDate(const Date& d) const; //两个日期相差的天数
Date AfterNDays(int n) const;  //经过 n 天之后的日期
bool Equal(const Date& d) const; //两个日期是否相等
bool Earlier(const Date& d) const; //哪个日期更早

```

(2) 定义 `static` 成员函数 `Today()` 获取系统当前日期。可以借用系统函数 `time()`。详

细使用方法请查询 MSDN。

(3) 在定义函数 `nthDay()` 时, 先把整月的天数累加起来, 然后加上目前的天数。在定义函数 `nthWeek()` 时, 把函数 `nthDay()` 的返回结果除以 7 即可。

(4) 定义函数 `NextDate()` 和 `PrevDate()` 最简单的方法是借助已有的两个函数 `Date2Days()` 和 `Days2Date()`。先用函数 `Date2Days()` 计算从基准日期到当前日期经过的天数, 然后把该天数加 1 (对应于 `NextDate()`) 或减 1 (对应于 `PrevDate()`), 再把所得天数用函数 `Days2Date` 转换为日期。

(5) 定义函数 `DiffDate()` 最简单的方法是借助函数 `Date2Days()`。先后计算出两个日期相对基准的天数, 然后用这两个天数相减即可。

(6) 定义函数 `AfterNDays()` 最简单的方法是借助已有的两个函数 `Date2Days()` 和 `Days2Date()`。先用函数 `Date2Days()` 计算从基准日期到当前日期经过的天数, 然后把该天数加 `n`, 再把所得天数用函数 `Days2Date()` 转换为日期。

(7) 定义函数 `Equal()` 时, 分别比较两个对象的年、月、日是否相等。若全部相等, 则相同。定义函数 `Earlier()` 时, 对两个对象的年、月、日依次比较, 谁最先出现较小的数, 则该对象较早。

实验 8 运算符重载

实验目的与要求

1. 实验目的

- (1) 理解运算符重载的语法和形式。
- (2) 掌握常见运算符的重载方法。

2. 实验要求

- (1) 能够正确分析运算符函数的形式。
- (2) 熟练定义常用的运算符函数。

实验过程与示例

在本次实验中，需要重点练习常见运算符的重载，在实际应用中能够根据需求，正确定义运算符函数。

除与本书配套的主教材上的相关例题外，以下是一些较典型的示例，它们分别对算术运算类、关系逻辑运算类、增量/减量运算类等不同性质的运算符重载进行了示范。

1. 算术运算类的运算符重载

【例 1.8.1】有理数类 Rational 的应用示例。

数学中类似 $1/2$ 、 $8/6$ 的数称为有理数。有理数类 Rational 是运算符重载的一个非常典型的类，它能够实现很多运算符的运算，在这些运算符重载实现的过程中，也能够体现很多方法和思想。本程序较为全面地实现了 Rational 类的功能，因而显得篇幅较多，但是整个代码结构较清晰。下面把 Rational 类的声明、实现和应用分为 3 个文件。

头文件 Rational.h 中存放了类 Rational 的接口，其内容如下：

```
#01     #ifndef RATIONAL_CLASS
#02     #define RATIONAL_CLASS
#03     #include <iostream>
#04     class Rational
#05     {
#06     private:
#07         int numerator, denominator;           //分子，分母
#08         //以下为一组工具函数
#09         static int digits(double d);           //计算浮点数小数部分的位数
```

```

#10     static int gcd(int a,int b);           //计算两个数的最大公约数
#11     Rational normalize();                 //对有理数进行规范化
#12
#13 public:
#14     Rational(int n = 0, int d = 1) noexcept ;//给定分子和分母
#15     Rational(double d);                     //由浮点数构造有理数
#16     Rational(const Rational& r);           //复制构造函数
#17
#18     Rational& operator = (const Rational& rhs);    //赋值运算符函数
#19     //以下一组函数实现两个 Rational 之间的算术运算
#20     Rational operator + (const Rational& rhs) const; //Rational+Rational
#21     Rational operator - (const Rational& rhs) const; //Rational-Rational
#22     Rational operator * (const Rational& rhs) const; //Rational*Rational
#23     Rational operator / (const Rational& rhs) const; //Rational/Rational
#24     //以下一组函数实现两个 Rational 之间的复合算术运算
#25     Rational& operator += (const Rational& rhs); //Rational+=Rational
#26     Rational& operator -= (const Rational& rhs); //Rational-=Rational
#27     Rational& operator *= (const Rational& rhs); //Rational*=Rational
#28     Rational& operator /= (const Rational& rhs); //Rational/=Rational
#29     //以下一组函数实现 int 与 Rational 之间的算术运算
#30     friend Rational operator + (int n, const Rational& rhs); //n+Rational
#31     friend Rational operator - (int n, const Rational& rhs); //n-Rational
#32     friend Rational operator * (int n, const Rational& rhs); //n*Rational
#33     friend Rational operator / (int n, const Rational& rhs); //n/Rational
#34     //以下一组函数实现 int 与 Rational 之间的复合算术运算
#35     friend int& operator += (int& n, const Rational& rhs);
#36     friend int& operator -= (int& n, const Rational& rhs);
#37     friend int& operator *= (int& n, const Rational& rhs);
#38     friend int& operator /= (int& n, const Rational& rhs);
#39     //以下一组函数实现 double 与 Rational 之间的算术运算
#40     friend Rational operator + (double d, const Rational& rhs);
#41     friend Rational operator - (double d, const Rational& rhs);
#42     friend Rational operator * (double d, const Rational& rhs);
#43     friend Rational operator / (double d, const Rational& rhs);
#44     //以下一组函数实现 double 与 Rational 之间的复合算术运算
#45     friend double& operator += (double& d, const Rational& rhs);
#46     friend double& operator -= (double& d, const Rational& rhs);
#47     friend double& operator *= (double& d, const Rational& rhs);
#48     friend double& operator /= (double& d, const Rational& rhs);
#49     //以下一组函数实现两个 Rational 之间的关系运算
#50     bool operator > (const Rational& rhs) const; //Rational>Rational
#51     bool operator == (const Rational& rhs) const; //Rational==Rational
#52     bool operator >= (const Rational& rhs) const; //Rational>=Rational
#53     bool operator < (const Rational& rhs) const; //Rational<Rational
#54     bool operator <= (const Rational& rhs) const; //Rational<=Rational
#55     bool operator != (const Rational& rhs) const; //Rational!=Rational
#56     //以下一组函数实现 int 与 Rational 之间的关系运算
#57     friend bool operator > (int n, const Rational& rhs); //n>Rational
#58     friend bool operator >= (int n, const Rational& rhs); //n>=Rational

```

```

#59     friend bool operator < (int n, const Rational& rhs); //n<Rational
#60     friend bool operator <= (int n, const Rational& rhs); //n<=Rational
#61     friend bool operator == (int n, const Rational& rhs); //n==Rational
#62     friend bool operator != (int n, const Rational& rhs); //n!=Rational
#63     //以下一组函数实现 double 与 Rational 之间的关系运算
#64     friend bool operator > (double d, const Rational& rhs);
#65     friend bool operator >= (double d, const Rational& rhs);
#66     friend bool operator < (double d, const Rational& rhs);
#67     friend bool operator <= (double d, const Rational& rhs);
#68     friend bool operator == (double d, const Rational& rhs);
#69     friend bool operator != (double d, const Rational& rhs);
#70     //以下一组函数计算正负数
#71     Rational operator + () const;           //+Rational
#72     Rational operator - () const;           //-Rational
#73     //以下一组函数实现增量和减量运算
#74     Rational& operator ++ ();               //++Rational
#75     Rational operator ++ (int);             //Rational++
#76     Rational& operator -- ();               //--Rational
#77     Rational operator -- (int);             //Rational--
#78     //以下一组函数是转换运算符函数
#79     operator int () const;                  //int(Rational)
#80     operator double () const;               //double(Rational)
#81     //以下一组函数实现流插入和提取运算
#82     friend std::ostream& operator << (std::ostream& os,
#83         const Rational& rhs);
#84     friend std::istream& operator >> (std::istream& is,
#85         Rational& rhs);
#86     };
#87     #endif //RATIONAL_CLASS

```

分子 `numerator` 和分母 `denominator` 是类 `Rational` 的两个数据成员, 注意它们都是 `int` 类型。成员函数 `digit()`、`gcd()` 和 `normalize()` 是类的一组工具函数。`Rational` 类的 3 个构造函数和一个赋值运算符函数实现了 `Rational` 对象的构造、复制和赋值, 其中参数为 `double` 类型的构造函数是类的转换构造函数, 可以把 `double` 类型的数据转换为 `Rational` 对象。类 `Rational` 先后声明了如下一些数学运算函数: ①算术运算和复合算术运算; ②关系运算; ③一元算术加法和减法运算; ④增量和减量运算; ⑤类型转换运算; ⑥流插入和提取运算。

源文件 `Rational.cpp` 中存放了类 `Rational` 的实现过程, 其内容如下:

```

#01     #include <iostream>
#02     #include <cassert>                       //for assert
#03     #include <cmath>                          //for pow
#04     #include "Rational.h"
#05
#06     int Rational::digits(double d) {           //计算浮点数小数部分的位数
#07         char buffer[64];                     //临时存储字符串
#08         sprintf(buffer, "%g", d);             //把 double 转换为字符串
#09         int n = strlen(buffer);                //计算字符串长度
#10         int i = 0;                            //定义下标并置零

```

```

#11         while (buffer[i] != '\0' && buffer[i] != '.') //寻找小数点
#12             ++i;                                     //下标前进
#13         if (i == n)                                   //若没有找到小数点
#14             return 0;                                 //则小数部分的位数为 0
#15         return n - i - 1;                             //否则返回小数部分的位数
#16     }
#17
#18     int Rational::gcd(int a, int b) {                 //计算两个数的最大公约数
#19         int r;                                       //临时保存余数
#20         while(b != 0) {                             //除数不能为 0
#21             r = a % b;                               //计算余数
#22             a = b;                                   //替换被除数
#23             b = r;                                   //替换除数
#24         }                                           //循环终止,余数为 0
#25         return a;                                   //返回所求最大公约数
#26     }
#27
#28     Rational Rational::normalize () {                //对有理数进行规范化
#29         int n = gcd(this->numerator, this->denominator); //先求最大公约数
#30         assert(n != 0);                             //确保最大公约数不为 0 (除非分子分母都为 0)
#31         this->numerator /= n;                         //分子化简
#32         this->denominator /= n;                       //分母化简
#33         if(this->denominator < 0) {                  //分母是否仍为 0
#34             this->numerator *= -1;                   //分子变号
#35             this->denominator *= -1;                 //分母为正
#36         }
#37         return *this;                               //返回化简结果
#38     }
#39
#40     Rational::Rational (int n, int d) noexcept { //给定分子和分母构造对象
#41         this->numerator = n;                         //分子直接赋值
#42         this->denominator = (d == 0 ? 1 : d);         //分母先判断再赋值
#43     }
#44
#45     Rational::Rational(double d) {                  //由浮点数构造有理数
#46         int n = Rational::digits(d);                //先计算小数部分的位数
#47         double e = pow(10.0, n);                    //化整需要的倍数
#48
#49         this->numerator = int(d * e);                 //化整之后,有理数的分子
#50         this->denominator = int(e);                 //倍数为分母
#51         this->normalize();                            //规范化
#52     }
#53
#54     Rational::Rational(const Rational& r) {          //复制构造函数
#55         this->numerator = r.numerator;               //复制分子
#56         this->denominator = r.denominator;          //复制分母
#57     }
#58
#59     Rational& Rational::operator = (const Rational& rhs) { //赋值运算

```

```

#60         this->numerator = rhs.numerator;           //复制分子
#61         this->denominator = rhs.denominator;       //复制分母
#62         return *this;                               //返回当前对象
#63     }
#64     //以下一组函数实现两个 Rational 之间的算术运算
#65     Rational Rational::operator + (const Rational& rhs) const {
#66         Rational r;                                   //生成临时对象保存结果
#67         r.numerator = this->numerator * rhs.denominator
#68             + this->denominator * rhs.numerator;     //计算分子
#69         r.denominator = this->denominator * rhs.denominator; //计算分母
#70         return r.normalize();                         //规范化之后返回
#71     }
#72     Rational Rational::operator - (const Rational& rhs) const {
#73         Rational r;                                   //生成临时对象保存结果
#74         r.numerator = this->numerator * rhs.denominator
#75             - this->denominator * rhs.numerator;     //计算分子
#76         r.denominator = this->denominator * rhs.denominator; //计算分母
#77         return r.normalize();                         //化简之后返回
#78     }
#79     Rational Rational::operator * (const Rational& rhs) const {
#80         Rational r;                                   //生成临时对象保存结果
#81         r.numerator = this->numerator * rhs.numerator; //计算分子
#82         r.denominator = this->denominator * rhs.denominator; //计算分母
#83         return r.normalize();                         //化简之后返回
#84     }
#85     Rational Rational::operator / (const Rational& rhs) const {
#86         assert(rhs.numerator != 0);
#87         Rational r;                                   //生成临时对象保存结果
#88         r.numerator = this->numerator * rhs.denominator; //计算分子
#89         r.denominator = this->denominator * rhs.numerator; //计算分母
#90         return r.normalize();                         //化简之后返回
#91     }
#92     //以下一组函数实现两个 Rational 之间的复合算术运算
#93     Rational& Rational::operator += (const Rational& rhs) {
#94         return *this = *this + rhs;                   //重用+=和=
#95     }
#96     Rational& Rational::operator -= (const Rational& rhs) {
#97         return *this = *this - rhs;                   //重用-=和=
#98     }
#99     Rational& Rational::operator *= (const Rational& rhs) {
#100        return *this = *this * rhs;                   //重用*=和=
#101    }
#102    Rational& Rational::operator /= (const Rational& rhs) {
#103        return *this = *this / rhs;                   //重用/=和=
#104    }
#105    //以下一组函数实现整型数 n 与 Rational 对象之间的算术运算
#106    Rational operator + (int n, const Rational& rhs) {
#107        return Rational(n) + rhs;                     //重用+
#108    }

```

```

#109 Rational operator - (int n, const Rational& rhs) {
#110     return Rational(n) - rhs;           //重用-
#111 }
#112 Rational operator * (int n, const Rational& rhs) {
#113     return Rational(n) * rhs;           //重用*
#114 }
#115 Rational operator / (int n, const Rational& rhs) {
#116     return Rational(n) / rhs;           //重用/
#117 }
#118 //以下一组函数实现整型数 n 与 Rational 对象之间的复合算术运算
#119 int& operator += (int& n, const Rational& rhs) {
#120     n = n * rhs.denominator + rhs.numerator;
#121     return n /= rhs.denominator;
#122 }
#123 int& operator -= (int& n, const Rational& rhs) {
#124     n = n * rhs.denominator - rhs.numerator;
#125     return n /= rhs.denominator;
#126 }
#127 int& operator *= (int& n, const Rational& rhs) {
#128     n = n * rhs.numerator;
#129     return n /= rhs.denominator;
#130 }
#131 int& operator /= (int& n, const Rational& rhs) {
#132     assert(rhs.numerator != 0);
#133     n = n * rhs.denominator;
#134     return n /= rhs.numerator;
#135 }
#136 //以下一组函数实现浮点数 d 与 Rational 对象之间的算术运算
#137 Rational operator + (double d, const Rational& rhs) {
#138     return Rational(d) + rhs;           //重用+
#139 }
#140 Rational operator - (double d, const Rational& rhs) {
#141     return Rational(d) - rhs;           //重用-
#142 }
#143 Rational operator * (double d, const Rational& rhs) {
#144     return Rational(d) * rhs;           //重用*
#145 }
#146 Rational operator / (double d, const Rational& rhs) {
#147     return Rational(d) / rhs;           //重用/
#148 }
#149 //以下一组函数实现浮点数 d 与 Rational 对象之间的复合算术运算
#150 double& operator += (double& d, const Rational& rhs) {
#151     return d += (double)rhs.numerator / rhs.denominator;
#152 }
#153 double& operator -= (double& d, const Rational& rhs) {
#154     return d -= (double)rhs.numerator / rhs.denominator;
#155 }
#156 double& operator *= (double& d, const Rational& rhs) {
#157     return d *= (double)rhs.numerator / rhs.denominator;

```

```

#158     }
#159     double& operator /= (double& d, const Rational& rhs) {
#160         assert(rhs.numerator != 0);
#161         return d /= (double)rhs.numerator / rhs.denominator;
#162     }
#163     //以下实现两个 Rational 对象之间的关系运算
#164     bool Rational::operator < (const Rational& rhs) const {
#165         return (*this - rhs).numerator < 0; //判断差的分子是否小于 0
#166     }
#167     bool Rational::operator == (const Rational& rhs) const {
#168         return this->numerator * rhs.denominator
#169             == this->denominator * rhs.numerator; //判断交叉相乘的结果
#170     }
#171     bool Rational::operator > (const Rational& rhs) const {
#172         return rhs < *this; //重用<
#173     }
#174     bool Rational::operator >= (const Rational& rhs) const {
#175         return !(*this < rhs); //重用<
#176     }
#177     bool Rational::operator <= (const Rational& rhs) const {
#178         return !(rhs < *this); //重用<
#179     }
#180     bool Rational::operator != (const Rational& rhs) const {
#181         return !(*this == rhs); //重用==
#182     }
#183     //以下计算整数 n 与 Rational 对象之间的大小及相等关系
#184     bool operator > (int n, const Rational& rhs) {
#185         return n > (double)rhs.numerator / rhs.denominator;
#186     }
#187     bool operator >= (int n, const Rational& rhs) {
#188         return n >= (double)rhs.numerator / rhs.denominator;
#189     }
#190     bool operator < (int n, const Rational& rhs) {
#191         return n < (double)rhs.numerator / rhs.denominator;
#192     }
#193     bool operator <= (int n, const Rational& rhs) {
#194         return n <= (double)rhs.numerator / rhs.denominator;
#195     }
#196     bool operator == (int n, const Rational& rhs) {
#197         return n == (double)rhs.numerator / rhs.denominator;
#198     }
#199     bool operator != (int n, const Rational& rhs) {
#200         return n != (double)rhs.numerator / rhs.denominator;
#201     }
#202     //以下计算浮点数 d 与 Rational 对象之间的大小及相等关系
#203     bool operator > (double d, const Rational& rhs) {
#204         return d > (double)rhs.numerator / rhs.denominator;
#205     }
#206     bool operator >= (double d, const Rational& rhs) {

```

```

#207         return d >= (double)rhs.numerator / rhs.denominator;
#208     }
#209     bool operator < (double d, const Rational& rhs) {
#210         return d < (double)rhs.numerator / rhs.denominator;
#211     }
#212     bool operator <= (double d, const Rational& rhs) {
#213         return d <= (double)rhs.numerator / rhs.denominator;
#214     }
#215     bool operator == (double d, const Rational& rhs) {
#216         return d == (double)rhs.numerator / rhs.denominator;
#217     }
#218     bool operator != (double d, const Rational& rhs) {
#219         return d != (double)rhs.numerator / rhs.denominator;
#220     }
#221     //以下计算 Rational 对象的正负数
#222     Rational Rational::operator + () const {
#223         return *this;                                //实为本身
#224     }
#225     Rational Rational::operator - () const {
#226         return Rational(-this->numerator, this->denominator); //分子取反
#227     }
#228     //以下计算增量与减量
#229     Rational& Rational::operator ++ () {
#230         this->numerator += this->denominator; //分子增加
#231         return *this;
#232     }
#233     Rational Rational::operator ++ (int) {
#234         Rational temp(*this);
#235         ++ (*this);                                //重用前置自增运算
#236         return temp;
#237     }
#238     Rational& Rational::operator -- () {
#239         this->numerator -= this->denominator; //分子减少
#240         return *this;
#241     }
#242     Rational Rational::operator -- (int) {
#243         Rational temp(*this);
#244         -- (*this);                                //重用前置自减运算
#245         return temp;
#246     }
#247     //以下为 Rational 类对象的转换运算符
#248     Rational::operator int () const {                //转换为 int 值
#249         return this->numerator / this->denominator; //int 值相除
#250     }
#251     Rational::operator double () const {            //转换为 double 值
#252         return double(this->numerator) / this->denominator; //double 值相除
#253     }
#254     //以下实现流插入运算和流提取运算
#255     std::ostream& operator << (std::ostream& os,

```



```

#256                                     const Rational& rhs) {
#257         os << rhs.numerator << "/" << rhs.denominator;
#258         return os;
#259     }
#260     std::istream& operator >> (std::istream& is,
#261                               Rational& rhs){
#262         char dump;
#263         is >> rhs.numerator >> dump >> rhs.denominator;
#264         return is;
#265     }

```

在实现 `Rational` 类的各类运算时，很多运算符函数重用了前面已经实现的运算符，这样既可以简化代码，避免冗余，又可以保证代码的一致性，当某运算符函数的实现过程发生改变，重用该代码的其他运算符函数自动更新。例如，两个 `Rational` 对象之间的算术运算是基本型运算，它们之间的复合算术运算、`int` 与 `Rational` 之间的算术运算、`double` 与 `Rational` 之间的算术运算就重用了基本型的算术运算。但是有些运算符没有重用已有的代码，这是考虑到它们可以有更为简单的实现方式，如 `int` 与 `Rational` 之间的复合算术运算、`double` 与 `Rational` 之间的复合算术运算。

源文件 `main.cpp` 对类 `Rational` 的所有功能进行了测试，其内容如下：

```

#01     #include "Rational.h"
#02     #include "xr.hpp"
#03
#04     int main()
#05     {
#06         Rational r1(30, 9), r2(8.125), r;
#07         xr(r1); xr(r2); xr(r);
#08
#09         xr(r1 + r2); xr(r1 - r2); xr(r1 * r2); xr(r1 / r2);
#10
#11         r = 0;
#12         xr(r += r1); xr(r -= r1); xr(r *= r1); xr(r /= r1);
#13
#14         xr(r1 > r2); xr(r1 >= r2); xr(r1 < r2);
#15         xr(r1 <= r2); xr(r1 == r2); xr(r1 != r2);
#16
#17         int n = 3;
#18         xr(n + r2); xr(n - r2); xr(n * r2); xr(n / r2);
#19         xr(n += r2); xr(n -= r2); xr(n *= r2); xr(n /= r2);
#20
#21         xr(n > r2); xr(n >= r2); xr(n < r2);
#22         xr(n <= r2); xr(n == r2); xr(n != r2);
#23
#24         double d = 3.14;
#25         xr(d + r1); xr(d - r1); xr(d * r1); xr(d / r1);
#26         xr(d += r1); xr(d -= r1); xr(d *= r1); xr(d /= r1);
#27
#28         xr(d > r1); xr(d >= r1); xr(d < r1);
#29         xr(d <= r1); xr(d == r1); xr(d != r1);

```

```

#30
#31         xr(+r1); xr(-r1);
#32
#33         xr(++r2); xr(r2++); xr(--r2); xr(r2--);
#34
#35         xr(int(r2)); xr(double(r2));
#36     }

```

请仔细分析本例程中运算符重载的方法,掌握各类运算符重载的一般规律,从而达到熟练应用的程度。

2. 关系运算符和逻辑非运算符的重载

【例 1.8.2】字符串类 String 的应用示例。

本例重点练习关系运算符重载,还包括逻辑非运算符和转换运算符的重载。下列程序组织为多文件结构。

头文件 String.h 的内容如下:

```

#01     #ifndef STRING_CLASS
#02     #define STRING_CLASS
#03
#04     #include <iostream>
#05
#06     class String                                //定义字符串类
#07     {
#08     private:
#09         char *pstr;                            //存放字符的首地址
#10         size_t sz;                            //字符个数
#11     public:
#12         String(const char* s = "") noexcept;    //默认构造函数
#13         String(const char* s, size_t n);        //构造函数
#14         String(const String& s);                //字符串复制
#15         ~String() noexcept;                    //析构函数
#16
#17         const char* str() const {return this->pstr;} //转换为C字符串
#18         bool operator ! () const {return this->sz == 0;} //判断是否为空
#19         operator const char* () const {return this->pstr;} //转换运算符
#20
#21         String& operator = (const String& rhs);    //字符串赋值
#22         String operator + (const String& rhs) const; //字符串连接
#23         String& operator += (const String& rhs);    //字符串追加
#24
#25         bool operator < (const String& rhs) const; //比较大小
#26         bool operator == (const String& rhs) const; //判断相等
#27
#28         friend std::ostream& operator << (std::ostream& os, const String& rhs);
#29         friend std::istream& operator >> (std::istream& is, String& rhs);
#30     };
#31
#32     #endif //STRING_CLASS

```

源文件 String.cpp 的内容如下:

```
#01     #include <iostream>
#02     #include <cstring>
#03     #include <cassert>
#04     #include "String.h"
#05
#06     String::String(const char* s) noexcept { //由 C 字符串构造 String
#07         this->sz = strlen(s);                //获取参数的长度
#08         this->pstr = new char[sz + 1];        //按此长度加 1 申请内存
#09         assert(this->pstr != NULL);           //确保申请成功
#10         strcpy(this->pstr, s);                //复制字符元素
#11     }
#12     String::String(const char* s, size_t n) { //构造函数
#13         this->sz = std::min(strlen(s), n);    //计算可复制字符的最大长度
#14         this->pstr = new char[sz + 1];        //按此长度加 1 申请内存
#15         assert(this->pstr != NULL);           //确保申请成功
#16         strncpy(this->pstr, s, sz);           //复制指定字符元素
#17         this->pstr[sz] = '\0';                //追加字符串结束符
#18     }
#19     String::String(const String& s) { ,        //字符串复制
#20         if (this != &s) {                    //防止自复制
#21             this->sz = s.sz;                  //获取样本的长度
#22             this->pstr = new char[sz + 1];    //按此长度加 1 申请内存
#23             assert(this->pstr != NULL);       //确保申请成功
#24             strcpy(this->pstr, s.pstr);       //复制样本字符串元素
#25         }
#26     }
#27     String::~String() noexcept {              //析构函数
#28         delete [] this->pstr;                 //释放字符数组内存
#29         this->pstr = NULL;                    //指针置空
#30         this->sz = 0;                         //个数清零
#31     }
#32     String& String::operator = (const String& rhs) { //字符串赋值
#33         if (this != &rhs) {                  //防止自赋值
#34             if (this->sz != rhs.sz) {         //两字符串是否等长
#35                 this->sz = rhs.sz;            //获取右操作数的长度
#36                 delete [] this->pstr;         //释放左操作数旧有内存
#37                 this->pstr = new char[this->sz + 1]; //按新长度加 1 申请内存
#38                 assert(this->pstr != NULL);   //确保申请成功
#39             }
#40             strcpy(this->pstr, rhs.pstr);     //复制右操作数字符串元素
#41         }
#42         return *this;                         //返回左操作数
#43     }
#44     String String::operator + (const String& rhs) const { //字符串连接
#45         size_t sz = this->sz + rhs.sz;        //获取总长度
#46         char* buffer = new char[sz + 1];     //按此长度加 1 申请内存
#47         assert(buffer != NULL);               //确保申请成功
#48         strcpy(buffer, this->pstr);            //先复制左操作数的字符
```

```

#49         strcat(buffer, rhs.pstr);           //再复制右操作数的字符
#50         String result(buffer);              //用此字符串构造结果对象
#51         delete [] buffer;                   //释放申请的内存
#52         return result;                      //返回结果对象
#53     }
#54     String& String::operator += (const String& rhs) { //字符串追加
#55         return *this = *this + rhs;           //重用+ 和 =
#56     }
#57     bool String::operator < (const String& rhs) const { //判断字符串的小于关系
#58         return strcmp(this->pstr, rhs.pstr) < 0; //用函数 strcmp 判断
#59     }
#60     bool String::operator == (const String& rhs) const { //判断两个字符串是否相等
#61         return strcmp(this->pstr, rhs.pstr) == 0; //用函数 strcmp 判断
#62     }
#63     std::ostream& operator << (std::ostream& os, const String& rhs) {
#64         return os << rhs.str();               //输出字符元素
#65     }
#66     std::istream& operator >> (std::istream& is, String& rhs) {
#67         char buffer[128];                      //假定输入不会超过 128 字符
#68         is.getline(buffer, 128);               //先存放在此数组中
#69         rhs = String(buffer, is.gcount());     //由此构造结果对象
#70         return is;                             //返回流对象
#71     }

```

源文件 main.cpp 的内容如下:

```

#01     #include "String.h"
#02     #include "xr.hpp"
#03
#04     int main()
#05     {
#06         String a, b, c;
#07         xr(!a); xr(!b); xr(!c);
#08         std::cout << "Please enter 2 strings: ";
#09         std::cin >> a >> b;
#10         xr(a); xr(b); xr(!a); xr(!b);
#11
#12         c = String(a.str(), 5);
#13         c += b; xr(c);
#14
#15         xr(a < c);
#16         xr(a == c);
#17     }

```

分析程序输出结果, 并思考下列问题:

(1) 参考主教材上 String 类的相关示例, 为该类添加转移语义的支撑函数, 思考复制语义和转移语义的适用场合。

(2) 关系运算符的重载有什么特点?

(3) 转换运算符函数和转换构造函数一般是相对应的, 请分析上述程序中这两个成员函数。

(4) 请分析对 `String` 类重载流提取运算符时要注意什么问题。

(5) 上述程序中只重载了关系运算符中的 `<` 和 `==`，请基于这两个运算符，重载其余 4 个关系运算符。

(6) 为了计算 `const char*>String` 型关系表达式，请以友元函数形式重载关系运算符，并把该运算符函数的第 1 个参数设置成 `const char*` 类型，把第 2 个参数设置成 `String` 类型。

(7) 参考标准库中的 `String` 类，当左操作数或右操作数为右值引用 `rv` 时，请为本类中的 `operator+` 添加相关运算符函数，实现相关表达式的计算，如 `str+rv` 型、`rv+cstr` 型表达式的计算。

3. 增量/减量运算符的重载

【例 1.8.3】时间类 `Time` 的应用示例。

本题重点练习增量/减量运算符的重载，同时包括几个对 `Time` 类有意义的运算符函数。下面以多文件结构给出程序代码。

头文件 `Time.h` 的内容如下：

```
#01     #ifndef TIME_CLASS
#02     #define TIME_CLASS
#03
#04     #include <iostream>
#05
#06     class Time                                //定义时间类
#07     {
#08     private:
#09         int hour, minute, second;              //小时,分,秒
#10
#11         static int Time2Seconds(const Time& t); //把时间 t 化为秒数
#12         static Time Seconds2Time(int s);        //把秒数化为时间
#13
#14     public:
#15         Time(int h = 0, int m = 0, int s = 0) noexcept; //构造函数
#16
#17         Time& operator ++ ();                    //++Time
#18         Time operator ++ (int);                  //Time++
#19         Time& operator -- ();                    //--Time
#20         Time operator -- (int);                  //Time--
#21
#22         int operator - (const Time& rhs) const; //Time - Time
#23         Time operator + (int s) const;           //Time + int
#24
#25         bool operator == (const Time& rhs) const; //Time == Time
#26
#27         friend std::ostream& operator << (std::ostream& os, const Time& rhs);
#28         friend std::istream& operator >> (std::istream& is, Time& rhs);
#29     };
#30
```

#31 #endif //TIME_CLASS

源文件 Time.cpp 的内容如下:

```
#01        #include "Time.h"
#02
#03        Time::Time(int h, int m, int s) noexcept {        //构造时间对象
#04                hour     = ( h >= 0 && h < 24 ? h : 0 );
#05                minute  = ( m >= 0 && m < 60 ? m : 0 );
#06                second  = ( s >= 0 && s < 60 ? s : 0 );
#07        }
#08        Time& Time::operator ++ () {                        //++Time
#09                ++this->second;                                //首先增加秒
#10                if (this->second >= 60) {                        //若超过进制
#11                        this->second -= 60;                        //则按进制取模
#12                        ++this->minute;                        //同时增加分
#13                        if (this->minute >= 60) {                        //若超过进制
#14                                this->minute -= 60;                        //则按进制取模
#15                                ++this->hour;                        //同时增加小时
#16                                if (this->hour >= 24)                        //若超过进制
#17                                        this->hour -= 24;                        //则按进制取模
#18                        }
#19                }
#20                return *this;                                //返回当前对象
#21        }
#22        Time Time::operator ++ (int) {                        //Time++
#23                Time temp(*this);                                //生成临时对象保存当前对象
#24                ++ (*this);                                //重用++Time
#25                return temp;                                //返回临时对象
#26        }
#27        Time& Time::operator -- () {                        //--Time
#28                --this->second;                                //减少秒数
#29                if (this->second < 0) {                        //若向下溢出
#30                        this->second += 60;                        //则按进制取模
#31                        --this->minute;                        //同时减少分
#32                        if (this->minute < 0) {                        //若向下溢出
#33                                this->minute += 60;                        //则按进制取模
#34                                --this->hour;                        //同时减少小时
#35                                if (this->hour < 0)                        //若向下溢出
#36                                        this->hour += 24;                        //则按进制取模
#37                        }
#38                }
#39                return *this;                                //返回当前对象
#40        }
#41        Time Time::operator -- (int) {                        //Time--
#42                Time temp(*this);                                //生成临时对象保存当前对象
#43                -- *this;                                //重用--Time
#44                return temp;                                //返回临时对象
#45        }
#46        int Time::Time2Seconds(const Time& t) { //把时间 t 化为秒数
#47                int s = t.second;                                //取得秒数
```

```

#48         s += t.minute * 60;                //累加分
#49         s += t.hour * 3600;                //累加小时
#50         return s;                          //返回总秒数
#51     }
#52     Time Time::Seconds2Time(int s) {         //把秒数化为时间
#53         Time t;                             //定义对象保存结果
#54         t.hour = s / 3600; s -= t.hour * 3600; //计算小时
#55         t.minute = s / 60; s -= t.minute * 60; //计算分
#56         t.second = s;                       //计算秒
#57         return t;                           //返回结果
#58     }
#59     int Time::operator - (const Time& rhs) const { //Time - Time
#60         int l = Time::Time2Seconds(*this); //把左操作数化为秒数
#61         int r = Time::Time2Seconds(rhs);    //把右操作数化为秒数
#62         if (l > r) return l - r;             //大者减去小者
#63         else return r - l;                  //大者减去小者
#64     }
#65     Time Time::operator + (int s) const { //Time + int
#66         int l = Time::Time2Seconds(*this); //左操作数化为秒数
#67         return Time::Seconds2Time(l + s);    //总秒数化为时间
#68     }
#69     bool Time::operator == (const Time& rhs) const { //Time == Time
#70         return this->hour == rhs.hour &&    //小时相等
#71             this->minute == rhs.minute &&    //且分相等
#72             this->second == rhs.second;      //且秒相等
#73     }
#74     std::ostream& operator << (std::ostream& os, const Time& rhs)
#75     {
#76         os << rhs.hour << ":" << rhs.minute << ":" << rhs.second;
#77         return os;
#78     }
#79     std::istream& operator >> (std::istream& is, Time& rhs)
#80     {
#81         char dump;
#82         is >> rhs.hour >> dump >> rhs.minute >> dump >> rhs.second;
#83         return is;
#84     }

```

源文件 main.cpp 的内容如下:

```

#01     #include "Time.h"
#02     #include "xr.hpp"
#03
#04     int main()
#05     {
#06         Time s, t, r;
#07         std::cout << "Please enter 2 times:(hh:mm:ss)";
#08         std::cin >> s >> t;
#09         xr(s); xr(t);
#10
#11         r = ++s; xr(r); xr(s);

```

```

#12         r = s++; xr(r); xr(s);
#13
#14         r = --t; xr(r); xr(t);
#15         r = t--; xr(r); xr(t);
#16
#17         xr(s); xr(t);
#18         r = s + (t - s);
#19         xr(r == t);
#20         r = s + (s - t);
#21         xr(r == t);
#22     }

```

分析程序的输出结果，并思考下列问题：

- (1) 成员函数 Time2Seconds() 和 Seconds2Time() 为什么定义为 static?
- (2) 增量运算符和减量运算符的重载有什么特点?
- (3) 在重载增量/减量运算符时是如何重用已有代码的?
- (4) 上述程序是如何实现秒数与 Time 对象之间的相互转换的?

实验题目与提示

1. 使用运算符重载实现类 Date 的功能。

- (1) 重载增量/减量运算符分别计算当前日期的后一个日期和前一个日期。
- (2) 重载减法运算符计算 Date-Date 型表达式，求任意两个日期之间相隔的天数。
- (3) 重载减法运算符计算 Date-int 型表达式，求在当前日期之前 n 天的日期。
- (4) 重载加法运算符计算 Date+int 型表达式，求在当前日期之后 n 天的日期。
- (5) 重载关系运算符比较两个日期的前后顺序和相等关系。

【提示】关于 Date 类的一些计算，请参考实验 7 中例 1.7.1 及其后的实验题目 1，此处不再详细提示。

2. 定义 CInt 类以模拟 C++ 基本类型 int 的功能。

- (1) 提供必要的成员函数和运算符实现对象的构造、复制、赋值和输入/输出等操作。
- (2) 重载运算符实现所有可能的算术运算、关系运算、增量和减量运算、类型转换运算等。
- (3) 提供函数能够求出整数的长度；取出任意数位；组成逆序整数。

【提示】如下定义类 CInt，其中每组运算符函数只给出了一个作为示例，其余请自行补充完整。

```

#01     class CInt {                                //描述整数
#02     private:
#03         int n;                                    //数据成员
#04     public:
#05         CInt(int m);                               //构造函数
#06         CInt(const CInt& ci);                       //复制构造函数
#07         CInt& operator = (const CInt& rhs);         //赋值运算符函数
#08         //以下为流输入/输出运算

```



```

#09      friend std::ostream& operator << (std::ostream& os, const CInt& rhs);
#10      friend std::istream& operator >> (std::istream& is, CInt& rhs);
#11      //以下为算术运算和复合算术运算
#12      CInt operator + (const CInt& rhs) const;           //CInt+CInt
#13      CInt operator += (const CInt& rhs);           //CInt+=CInt
#14      //以下为算术运算和复合算术运算
#15      friend CInt operator + (int n, const CInt& rhs);   //n+CInt
#16      friend int& operator += (int& n, const CInt& rhs); //n+=CInt
#17      //以下为关系运算
#18      bool operator < (const CInt& rhs) const;          //CInt<CInt
#19      friend bool operator < (int n, const CInt& rhs);   //n<CInt
#20      //以下为一元算术加法和减法运算
#21      CInt operator + () const;                         //+CInt
#22      //以下为增量和减量运算
#23      CInt& operator ++ ();                             //++CInt
#24      //以下为类型转换运算
#25      operator int () const;                            //int(CInt)
#26      //以下为一组工具函数
#27      int Length() const;                               //计算整数的长度
#28      int Digit(int i) const;                           //数位 i 上的数字
#29      int Rev() const;                                  //逆序组成的整数
#30      };

```

3. 实现矩阵类 Matrix 的功能。

- (1) 提供必要的成员函数和运算符实现对象的构造、复制、赋值和输入/输出等操作。
- (2) 重载函数调用运算符提供下标访问能力。
- (3) 重载运算符实现矩阵的加、减、乘运算等。

【提示】如下定义类 Matrix，其中算术运算符函数只给出了加法作为示例，其余请自行补充完整。

```

#01      class Matrix {                                     //描述矩阵
#02      private:
#03          enum {Row = 10, col = 10};                     //行和列的最大长度
#04          double marray[Row][Col];                       //二维静态数组
#05          size_t row, col;                                //实际存放的行数和列数
#06      public:
#07          Matrix(size_t r, size_t c);                     //构造函数
#08          Matrix(const Matrix& m);                         //复制构造函数
#09          Matrix& operator = (const Matrix& rhs);          //赋值运算符函数
#10          //以下为数据成员的访问函数
#11          size_t rsize() const;                            //获取实际行数
#12          size_t csize() const;                            //获取实际列数
#13          //以下为下标访问函数
#14          double& operator () (size_t r, size_t c);       //下标访问函数
#15          const double& operator () (size_t r, size_t c) const; //下标访问函数
#16          //以下为算术运算和复合算术运算
#17          Matrix operator + (const Matrix& rhs) const;    //Matrix+Matrix
#18          Matrix& operator += (const Matrix& rhs);        //Matrix+=Matrix
#19          //以下为流输入/输出运算

```

```
#20     friend std::ostream& operator << (std::ostream& os, const Matrix& rhs);
#21     friend std::istream& operator >> (std::istream& is, Matrix& rhs);
#22 };
```

4. 定义长整数类 `HugeInt`, 以便在远远超出 `double` 表示范围的整型数据之间进行数学运算。

[illegible]

(2) 提供必要的成员函数和运算符实现对象的构造、复制、赋值和输入/输出等操作。

(3) 重载运算符实现 HugelInt 之间的算术运算、关系运算、增量和减量运算、类型转换运算等。

【提示】如下定义类 HugeInt，其中每组运算符函数只给出了一个作为示例，其余请自行补充完整。

```

#01 class HugeInt {                                //描述长整数
#02 private:
#03     std::string hint;                          //存放字符串形式的整数
#04 public:
#05     HugeInt(int n);                             //根据整数构造
#06     HugeInt(const char* s);                     //根据字符串构造
#07     HugeInt(const HugeInt& m);                   //复制构造函数
#08     HugeInt& operator = (const HugeInt& rhs);    //赋值运算符函数
#09     //以下为算术运算和复合算术运算
#10     HugeInt operator + (const HugeInt& rhs) const; //HugeInt+HugeInt
#11     HugeInt& operator += (const HugeInt& rhs);    //HugeInt+=HugeInt
#12     //以下为算术运算和复合算术运算(整数形式)
#13     friend HugeInt operator + (int n, const HugeInt& rhs); //n+HugeInt
#14     friend int& operator += (int& n, const HugeInt& rhs); //n+=HugeInt
#15     //以下为算术运算和复合算术运算(字符串形式)
#16     friend HugeInt operator + (const char* s, const HugeInt& rhs); //s+HugeInt
#17     friend char* & operator += (char* & s, const HugeInt& rhs); //s+=HugeInt
#18     //以下为关系运算
#19     bool operator < (const HugeInt& rhs) const;  //HugeInt<HugeInt
#20     friend bool operator < (int n, const HugeInt& rhs); //n<HugeInt
#21     friend bool operator < (const char* s, const HugeInt& rhs); //s<HugeInt
#22     //以下为一元算术加法和减法运算
#23     HugeInt operator + () const;                //+HugeInt
#24     //以下为增量和减量运算
#25     HugeInt& operator ++ ();                     //++HugeInt
#26     //以下为类型转换运算
#27     operator int () const;                       //int(HugeInt)
#28     operator const char* () const;              //(const char*)HugeInt
#29     //以下为流输入/输出运算
#30     friend std::ostream& operator << (std::ostream& os, const HugeInt& rhs);
#31     friend std::istream& operator >> (std::istream& is, HugeInt& rhs);
#32 };

```

5. 完善一元多项式类 Polynomial 的功能。

(1) 提供必要的成员函数和运算符实现对象的构造、复制、赋值和输入/输出等操作。

(2) 重载运算符实现多项式之间的加、减、乘运算等。

【提示】首先如下定义类 `Item` 表示多项式的项，其中重载了一些必要的运算符，每组运算符只给出了一个示例，其余请自行补充完整。然后如下定义类 `Polynomial`，其中每组运算符函数只给出了一个示例，其余请自行补充完整。

```
#01     class Item {                                //描述多项式的项
#02     private:
#03         double coefficient;                       //每项的系数
#04         int exponent;                             //每项的次数
#05     public:
#06         Item(double d, int e);                     //构造函数
#07         Item(const Item& i);                       //复制构造函数
#08         Item& operator = (const Item& rhs);        //赋值运算符函数
#09         //以下为算术运算和复合算术运算
#10         Item operator + (const Item& rhs) const;   //Item+Item
#11         Item& operator += (const Item& rhs);      //Item+=Item
#12         //以下为关系运算
#13         bool operator < (const Item& rhs) const;  //按照次数比较大小
#14         //以下计算在某点的值
#15         double operator () (double x) const;     //Item(x)
#16         //以下为一元算术加法和减法运算
#17         Item operator + () const;                 //+Item
#18         //以下为流输入/输出运算
#19         friend std::ostream& operator << (std::ostream& os, const Item& rhs);
#20         friend std::istream& operator >> (std::istream& is, Item& rhs);
#21     };
#22
#23     class Polynomial {                            //描述多项式
#24     private:
#25         std::list<Item> poly;                     //存放在链表中的项组成多项式
#26     public:
#27         Polynomial();                              //构造函数
#28         Polynomial(const Polynomial& m);           //复制构造函数
#29         Polynomial& operator = (const Polynomial& rhs); //赋值运算符函数
#30         //以下为算术运算和复合算术运算
#31         Polynomial operator + (const Polynomial& rhs) const; //Poly+Poly
#32         Polynomial& operator += (const Polynomial& rhs); //Poly+=Poly
#33         //以下计算多项式在某点的值
#34         double operator () (double x) const;     //Poly(x)
#35         //以下为一元算术加法和减法运算
#36         Polynomial operator + () const;           //+Poly
#37         //以下为流输入/输出运算
#38         friend std::ostream& operator<<(std::ostream& os, const Polynomial& rhs);
#39         friend std::istream& operator>>(std::istream& is, Polynomial& rhs);
#40     };
```

实验 9 模 板

实验目的与要求

1. 实验目的

- (1) 理解模板的概念。
- (2) 掌握函数模板和模板函数的概念。
- (3) 理解类模板和模板类的概念。

2. 实验要求

- (1) 理解模板定义的语法，能够判别正确定义的模板形式。
- (2) 熟练定义并正确应用函数模板。
- (3) 能够正确分析类模板，对已有的类模板进行正确实例化。

实验过程与示例

在本次实验中，需要重点练习函数模板的定义，并且能够在实际应用中根据需求定义正确的函数模板。在正确分析类模板的基础上，能够定义一些比较简单的类模板，并进行正确的实例化应用。

除与本书配套的主教材上的相关例题外，以下是一些较典型的示例，它们分别对函数模板的定义、类模板的定义和实例化等应用进行了示范。

1. 定义函数模板

【例 1.9.1】3 个数排序的应用示例。

本题重在练习分析函数模板对模板参数类型的需求。

```
#01     #include <string>
#02     #include "xr.hpp"
#03
#04     template <typename T>
#05     void myswap(T& a, T& b) {                //交换两个数
#06         T t(a);                             //需要复制构造函数
#07         a = b, b = t;                       //需要赋值运算符函数
#08     }                                        //需要析构函数
#09     template <typename T>
#10     void sort3(T& a, T& b, T& c) {          //3 个数排序
```

```

#11         if (a < b) myswap(a, b);           //需要 operator <
#12         if (a < c) myswap(a, c);
#13         if (b < c) myswap(b, c);
#14     }
#15     template <typename T>
#16     void print(const T& t) {                 //输出
#17         std::cout << t << std::endl;       //需要 operator <<
#18     }
#19
#20     class Student {                         //定义学生类
#21     private:
#22         int sid;                             //学号
#23         std::string name;                   //姓名
#24         double score;                       //成绩
#25     public:
#26         Student(int id, const char* str, double s)
#27             :sid(id), name(str), score(s)
#28         {}
#29         Student(const Student& s) {           //提供复制构造
#30             if (this != &s) {
#31                 this->sid = s.sid;
#32                 this->name = s.name;
#33                 this->score = s.score;
#34             }
#35         }
#36         Student& operator = (const Student& rhs) { //提供赋值运算
#37             if (this != &rhs) {
#38                 this->sid = rhs.sid;
#39                 this->name = rhs.name;
#40                 this->score = rhs.score;
#41             }
#42             return *this;
#43         }
#44         bool operator < (const Student& rhs) const { //提供小于比较
#45             return this->score < rhs.score;
#46         }
#47         friend std::ostream& operator << (std::ostream& os,
#48             const Student& rhs) {             //提供流插入运算
#49             os << rhs.sid << "\t" << rhs.name << "\t" << rhs.score;
#50             return os;
#51         }
#52     };
#53
#54     int main()
#55     {
#56         Student a(200803, "Tom", 85);
#57         Student b(200802, "Jerry", 90);
#58         Student c(200801, "Goofy", 70);
#59

```

```
#60         sort3(a, b, c);
#61         print(a); print(b); print(c);
#62     }
```

上述程序中, 函数 `sort3()` 对 3 个数从大到小排序, 排序的过程是 3 个数两两相比, 逆序则交换, 函数 `myswap()` 专用于交换两个数据。`main()` 函数对 3 个 `Student` 对象排序, 然后输出。

分析程序输出结果, 并思考下列问题:

(1) 函数 `myswap()` 交换两个数据, 在其实现过程中, 先由类型 `T` 的对象 `a` 复制构造类型 `T` 的对象 `t`; 接着在对象之间赋值。因此函数 `myswap()` 需要类 `Student` 中提供复制构造函数和赋值运算符函数, 请运用宏 `xr` 在相关函数中输出信息以证明这一分析。

(2) 函数 `sort3()` 对 3 个数两两比较以排序, 需要使用运算符 `<` 比较类型 `T` 的对象, 这要求 `Student` 类中提供运算符函数 `operator<`, 请运用宏 `xr` 在相关函数中输出信息以证明这一分析。

(3) 函数 `print()` 输出数据, 在其实现过程中, 需要使用流插入运算符输出类型 `T` 的对象, 这需要类 `Student` 中重载流插入运算符, 请运用宏 `xr` 在相关函数中输出信息以证明这一分析。

2. 类模板及其实例化

【例 1.9.2】Array 类的应用示例。

本例自定义一维动态数组类, 它能够根据运行时变量的取值生成指定长度的数组, 并具有复制、赋值、下标访问和判断相等等功能。

```
#01     #include <cassert>
#02     #include "xr.hpp"
#03
#04     template <typename T>
#05     class Array {                                //定义动态数组类型
#06     private:
#07         T* parray;                                //存放元素的首地址
#08         size_t sz;                                //元素个数
#09     public:
#10         Array(size_t n = 10) noexcept;           //默认构造函数
#11         Array(T* a, size_t n);                     //构造函数
#12         Array(const Array<T>& a);                 //复制构造函数
#13         ~Array() noexcept;                         //析构函数
#14
#15         Array<T>& operator = (const Array<T>& rhs); //赋值运算符函数
#16         size_t size() const {return this->sz;}    //获取元素个数
#17         T& operator [] (size_t idx);              //下标运算:用作左值
#18         const T& operator [] (size_t idx) const;  //下标运算:用作右值
#19         bool operator == (const Array<T>& rhs);    //判断相等
#20         void Print(std::ostream& os = std::cout) const; //输出元素
#21     };
#22     template <typename T>
#23     Array<T>::Array(size_t n) noexcept {          //构造长度为n的数组
```

```

#24         this->sz = n;                                //设置数组长度
#25         this->array = new T[this->sz];                //按此长度申请内存
#26         assert(this->array != NULL);                  //确保申请成功
#27     }
#28     template <typename T>
#29     Array<T>::Array(T* a, size_t n) {                  //由长度为n的数组a构造动态数组
#30         this->sz = n;                                //设置数组长度
#31         this->array = new T[this->sz];                //按此长度申请内存
#32         assert(this->array != NULL);                  //确保申请成功
#33
#34         for (size_t i = 0; i != sz; ++i)              //复制元素到数组内存中
#35             this->array[i] = a[i];
#36     }
#37     template <typename T>
#38     Array<T>::~~Array() noexcept {                    //析构函数
#39         delete [] this->array;                          //释放内存
#40         this->array = NULL;                            //指针置空
#41         this->sz = 0;                                  //个数清零
#42     }
#43     template <typename T>
#44     Array<T>::Array(const Array<T>& a) {              //数组复制
#45         if (this != &a) {                              //防止自复制
#46             this->sz = a.sz;                          //取得样本的长度
#47             this->array = new T[this->sz];              //按此长度申请内存
#48             assert(this->array != NULL);              //确保申请成功
#49
#50             for (size_t i = 0; i != sz; ++i) //逐元素复制
#51                 this->array[i] = a.array[i];
#52         }
#53     }
#54     template <typename T>
#55     Array<T>& Array<T>::operator = (const Array<T>& rhs) { //数组赋值
#56         if (this != &rhs) {                          //防止自赋值
#57             if (this->sz != rhs.sz) {                  //两个数组是否等长
#58                 this->sz = rhs.sz;                    //取得相同的长度
#59                 delete [] this->array;                  //释放左操作数的旧有长度
#60                 this->array = new T[this->sz];          //按此长度申请内存
#61                 assert (this->array != NULL);          //确保申请成功
#62             }
#63             for (size_t i = 0; i != sz; ++i)          //逐元素复制
#64                 this->array[i] = rhs.array[i];
#65         }
#66         return *this;                                //返回左操作数
#67     }
#68     template <typename T>
#69     T& Array<T>::operator [] (size_t idx) { //用作左值的下标运算
#70         assert(idx >= 0 && idx < this->sz);
#71         return this->array[idx];
#72     }

```

```

#73     template <typename T>
#74     const T& Array<T>::operator[](size_t idx) const { //用作右值的下标运算
#75         assert(idx >= 0 && idx < this->sz);
#76         return this->array[idx];
#77     }
#78     template <typename T>
#79     bool Array<T>::operator == (const Array<T>& rhs) { //判断相等
#80         if (this->sz == rhs.sz) { //长度相等
#81             for (size_t i = 0; i != this->sz; ++i) {
#82                 if (this->array[i] != rhs.array[i]) //只要有一个元素不相等
#83                     return false; //则不相等
#84             }
#85             return true; //所有元素都相等,则相等
#86         }
#87         return false; //长度不等,则不等
#88     }
#89     template <typename T>
#90     void Array<T>::Print(std::ostream& os) const { //输出数组
#91         for (size_t i = 0; i != this->sz; ++i)
#92             os << this->array[i] << "\t";
#93         os << std::endl;
#94     }
#95     template <typename T>
#96     Array<T> combine(const Array<T>& a, const Array<T>& b) { //合并数组
#97         Array<T> c(a.size() + b.size()); //生成长度为两者之和的空数组
#98
#99         size_t j = 0;
#100        for (size_t i = 0; i != a.size(); ++i, ++j) //复制第一个数组的元素
#101            c[j] = a[i];
#102        for (size_t i = 0; i != b.size(); ++i, ++j) //复制第二个数组的元素
#103            c[j] = b[i];
#104        return c; //返回结果
#105    }
#106    int main()
#107    {
#108        int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
#109        size_t n = sizeof(a) / sizeof(*a);
#110
#111        Array<int> na(a, 4); //用前 4 个元素生成数组
#112        xrv(na.Print());
#113        Array<int> nb(a + 4, n - 4); //用后 4 个元素生成数组
#114        xrv(nb.Print());
#115
#116        Array<int> nc(combine(na, nb)); //合并数组
#117        xrv(nc.Print());
#118
#119        xr(nc == Array<int>(a, n)); //两者应该相等
#120
#121        double d[] = {1, 2, 3, 4, 5, 6, 7, 8};

```



```

#122         size_t m = sizeof(d) / sizeof(*d);
#123
#124         Array<double> da(d, 4);
#125         xrv(da.Print());
#126         Array<double> db(d + 4, m - 4);
#127         xrv(db.Print());
#128
#129         Array<double> dc(combine(da, db));
#130         xrv(dc.Print());
#131
#132         xr(dc == Array<double>(d, m));
#133     }

```

上述程序定义了类模板 `Array`，并在类中提供了必要的构造、析构、复制、赋值、比较及元素访问等操作。`main()` 函数中把该类模板实例化为两种模板类：`Array<int>`、`Array<double>`，并根据这两种类型实例化了不同的对象，如 `na`、`nb`、`da`、`db` 等。

分析程序的输出结果，并思考下列问题：

(1) 定义类模板的重要工作是定义类中的每个成员函数，这又化归为函数模板的定义，请分析上述类成员函数模板的定义，并比较它们与普通函数模板的定义有何区别。

(2) 在实例化类模板时，给定不同的模板参数就会得到不同的模板类。感兴趣的读者，请自定义一种类型，如 `Student`，并把它作为上述类模板的模板参数而得到模板类 `Array<Student>`，生成该模板类的对象，并对其中的元素进行排序等操作。

(3) 请参考教材中 `Array` 类模板的定义，为本类添加转移语义的支持函数，并比较分析复制语义及转移语义支持函数的实现过程及使用场合的异同。

实验题目与提示

1. 编写函数模板，计算并返回两个数的最小值，要求能正确处理 C 风格的字符串。

【提示】可以如下定义函数模板 `my_min()` 计算两个数的最小值。

```

template <typename T>
T my_min(const T& a, const T& b);

```

该函数模板默认以关系运算符“<”对两个数进行大小比较，这不适合 C 字符串的比较，因此需要对该函数模板进行如下重载。

```

const char* my_min(const char* a, const char* b);

```

若把两个数的比较准则定义为函数模板的参数，如下所示，则可适用于任何数据的大小比较和计算。

```

template <typename T, class BinaryFunc>
T my_min(const T& a, const T& b, BinaryFunc bf);

```

2. 自定义函数模板对有 `n` 个元素的数组 `a` 排序，并把排序准则设置成函数参数。

【提示】可以定义如下函数模板，其中类型 `BinaryFunc` 表示排序准则。

```

template <typename T, class BinaryFunc>
void my_sort(T* a, int n, BinaryFunc bf);

```

3. 编写函数模板，统计数组中具有某特征的元素个数，并把该特征设置成函数

参数。

【提示】可以定义如下函数模板，其中类型 `Pred` 为表示特征的谓词。返回值为元素个数。

```
template <typename T, class Pred>
int my_count_if(T* a, int n, Pred pr);
```

4. 编写函数模板，把数组中具有某特征的元素全部替换为另一个值。

【提示】可以定义如下函数模板，其中类型 `Pred` 为表示特征的谓词。`newValue` 为替换之后的值。

```
template <typename T, class Pred>
void my_replace_if(T* a, int n, Pred pr, const T& newValue);
```

5. 编写类模板 `div` 对两个数进行除法运算，运算结果保存为商和余数。

【提示】可以仿照 STL 算术运算类函数对象的定义，来定义带余数的除法类模板 `div`。定义的关键是要在类中重载函数调用运算符函数。

```
#01     template <typename T>
#02     class div {
#03     private:
#04         T dividend, divisor;           //被除数和除数
#05     public:
#06         div(const T& a, const T& b);     //构造函数对象
#07         std::pair<T, T> operator () () const; //计算商和余数
#08     };
```

为了同时保存商和余数，用到了 STL 的 `pair` 类。计算过程把商和余数封装成一个 `pair` 对象然后返回。

6. 以非类型模板参数实现二维数组模板的静态存储，提供构造、复制和赋值、下标访问、流输入/输出等能力。

【提示】可以如下定义二维数组。

```
#01     template <typename T, int M, int N>
#02     class Array2D;           //前向声明
#03     //以下为两个函数模板的前向声明
#04     template <typename K, int P, int Q>
#05     std::ostream& operator << (std::ostream& os, const Array2D<K, P, Q>& rhs);
#06     template <typename K, int P, int Q>
#07     std::istream& operator >> (std::istream& is, Array2D<K, P, Q>& rhs);
#08
#09     template <typename T, int M, int N>           //M、N表示二维数组的行和列
#10     class Array2D {
#11     private:
#12         T parray[M][N];           //静态存储
#13     public:
#14         Array2D(const T* a, int n);           //由一维数组构造
#15         Array2D(const Array2D<T, M, N>& a);   //实现复制
#16         Array2D<T, M, N>& operator=(const Array2D<T, M, N>); //实现赋值
#17
#18         T& operator () (int r, int c);         //下标访问(左值版本)
#19         const T& operator () (int r, int c) const; //下标访问(右值版本)
```

```
#20      //以下实现流输入/输出运算
#21      friend std::ostream& operator << <T, M, N> (std::ostream& os,
#22          const Array2D<T, M, N>& rhs);
#23      friend std::istream& operator >> <T, M, N> (std::istream& is,
#24          Array2D<T, M, N>& rhs);
#25      };
```

实验 10 标准模板库 STL

实验目的与要求

1. 实验目的

- (1) 理解泛型程序设计的概念。
- (2) 掌握常用 STL 算法的用法。
- (3) 掌握常用 STL 容器的用法。

2. 实验要求

- (1) 理解关于迭代器、函数对象等重要概念。
- (2) 正确理解 STL 算法的功能及其参数。
- (3) 正确定义作为算法参数的函数对象。
- (4) 熟练应用 STL 容器及其操作解决比较简单的问题。

实验过程与示例

在本次实验中，需要重点练习 STL 算法和容器的应用，并且能够在实际应用中根据需求，选用合适的算法和容器。正确理解常用算法和容器操作的参数含义，并定义合适的函数、函数对象或 lambda 函数作为算法或操作的参数。能够自定义兼容于 STL 要求的简单算法和容器。

除与本书配套的主教材上的相关例题外，以下是一些较典型的示例，它们分别对各类函数对象的定义与应用、常用容器的基本操作等应用进行了示范。

1. 常用算法的典型示例

【例 1.10.1】 无参函数对象的定义应用示例。

```
#01     #include <iostream>
#02     #include <vector>
#03     #include <algorithm>
#04     #include <cstdlib>
#05     #include <ctime>
#06
#07     class rnd {                                //产生在某区间的随机数
#08     private:
#09         int low, high;                          //区间[low, high)
```

```

#10     public:
#11         rnd(int l, int h) :low(l), high(h) {
#12             srand(unsigned(time(NULL))); //初始化随机数种子
#13         }
#14         int operator () () const { //无参操作
#15             return low + rand() % (high - low + 1); //产生在该区间的随机数
#16         }
#17     };
#18     int main() {
#19         std::vector<int> vs(10); //可容纳 10 个元素的容器
#20         std::generate(vs.begin(), vs.end(), rnd(10, 99));
#21                                     //填充 [10, 99] 之间的元素
#22         std::for_each(vs.begin(), vs.end(), //逐个输出元素
#23             [](int x) {std::cout << x << "\t"; } ); //表示输出操作的 lambda 函数
#24     }

```

无参函数对象不是说在构造该对象时无须提供任何参数，而是在把它当作函数调用时无须提供任何函数实参。上述程序定义无参函数对象 `rnd` 产生在某区间内的随机数。分析程序的输出结果，体会无参函数对象的定义和应用。与之对比的是，算法 `for_each()` 在第三个参数传入了一个表示输出操作的 `lambda` 函数。同样是作为函数的参数，定义函数对象 `rnd()` 时却显得尤为臃肿，不如 `lambda` 函数简洁。

【例 1.10.2】一元函数对象的定义应用示例。

```

#01     #include <iostream>
#02     #include <iterator>
#03     #include <algorithm>
#04
#05     template <typename T>
#06     class iseven { //判断某数是否偶数
#07     public:
#08         bool operator () (const T& n) const { //一元函数对象
#09             return n % 2 == 0; //判断是否偶数
#10         }
#11     };
#12     //template <typename T>
#13     //class EqualTo { //判断与某值是否相等
#14     //private:
#15     //    T n; //待比较的值
#16     //public:
#17     //    EqualTo(const T& t) :n(t) {} //设置待比较的值
#18     //    bool operator () (const T& m) const {
#19     //        return n == m; //其他数与待比较值是否相等
#20     //    }
#21     //};
#22     int main() {
#23         int a[] { 1, 2, 3, 3, 3, 4, 5, 6, 3 };
#24         auto n = sizeof(a) / sizeof(*a);
#25         std::copy(a, a + n, std::ostream_iterator<int>(std::cout, " "));
#26         std::cout << std::endl;

```

```

#27
#28     auto m = std::count_if(a, a + n, iseven<int>()); //统计偶数个数
#29     std::cout << m << std::endl;                //3
#30
#31     //m = std::count_if(a, a + n, EqualTo<int>(3)); //统计等于 3 的元素个数
#32
#33     int to_cmp{ 3 };
#34     m = std::count_if(a, a + n,
#35         [to_cmp](int x) {return x == to_cmp; }); //统计等于 3 的元素个数
#36     std::cout << m << std::endl;                //4
#37 }

```

一元函数对象不是说在构造该对象时只需要提供一个参数，而是在把它当作函数调用时只需要提供一个函数实参。上述程序中，算法 `count_if()` 对满足某特征的元素计数，表达该特征的只能是一元谓词。“判断某数是否偶数”本身是一元谓词，类模板 `iseven` 是实现该一元谓词的函数对象，其对象 `iseven<int>()` 用作算法 `count_if()` 的参数，统计数组 `a` 中偶数元素的个数。

“判断两数是否相等”则是二元操作（需要两个操作数来比较），如何把二元操作实现为一元操作，类模板 `EqualTo` 进行了示范，关键在于把二元操作中的某个操作数设为类的状态值（如数据成员 `n`），并通过构造函数来设置该值，而在定义函数调用运算符时只需设置一个参数表示其他需要逐个比较的数，因而函数对象 `EqualTo<int>(3)` 就表示“判断与 3 是否相等”的一元谓词，它用作算法 `count_if()` 的参数，统计数组 `a` 中“与 3 相等”的元素个数。但是这种实现方法被注释掉了，仍然保留作为参考。

上述程序对“判断与 3 是否相等”的操作提供了两种实现办法，其中定义函数对象 `EqualTo` 的方法在 `main()` 函数中并没有实际用到，取而代之的是定义了一个 `lambda` 函数，该函数捕捉表示比较对象的变量 `to_cmp`，然后以非常简洁的方式实现了判断是否（与任意设定的值）相等的操作。分析程序的输出结果，体会一元函数对象的定义和应用。

【例 1.10.3】二元函数对象的定义应用示例。

```

#01     #include <functional>
#02     #include <string>
#03     #include <vector>
#04     #include "xr.hpp"
#05     #include "print.hpp"
#06
#07     class Student {
#08     private:
#09         int sid;
#10         std::string name;
#11         double score;
#12     public:
#13         Student(int id, const char* str, double s)
#14             :sid(id), name(str), score(s)
#15         {}
#16         bool operator < (const Student& rhs) const { //默认比较准则
#17             return this->score < rhs.score;        //按照成绩比较

```

```

#18     }
#19     friend std::ostream& operator<<(std::ostream& os,
#20         const Student& rhs){
#21         os << rhs.sid << "\t" << rhs.name << "\t" << rhs.score;
#22         return os;
#23     }
#24     friend bool cmpByID(const Student& lhs, const Student& rhs);
#25     friend class cmpByName;
#26 };
#27
#28 bool cmpByID(const Student& lhs, const Student& rhs) { //二元操作
#29     return lhs.sid < rhs.sid;                        //按照学号比较
#30 }
#31 class cmpByName {
#32 public:                                                //二元操作
#33     bool operator ()(const Student& lhs, const Student& rhs) const{
#34         return lhs.name < rhs.name;                  //按照姓名比较
#35     }
#36 };
#37
#38 int main()
#39 {
#40     std::vector<Student> vs;
#41     vs.push_back(Student(200803, "Tom", 85));
#42     vs.push_back(Student(200802, "Jerry", 90));
#43     vs.push_back(Student(200801, "Goofy", 70));
#44     vs.push_back(Student(200805, "Mickey", 70));
#45     vs.push_back(Student(200807, "Minnie", 92));
#46     vs.push_back(Student(200804, "Donald", 98));
#47     vs.push_back(Student(200806, "Pluto", 87));
#48
#49     auto ps = std::max_element(vs.begin(), vs.end()); xr(*ps);
#50     ps = std::max_element(vs.begin(), vs.end(), cmpByID); xr(*ps);
#51     ps = std::max_element(vs.begin(), vs.end(), cmpByName()); xr(*ps);
#52
#53     std::sort(vs.begin(), vs.end()); //默认按成绩排序
#54     xrv(print(vs.begin(), vs.end(), "sort by score:\n", "\n"));
#55     std::sort(vs.begin(), vs.end(), cmpByID); //按学号排序
#56     xrv(print(vs.begin(), vs.end(), "sort by ID:\n", "\n"));
#57     std::sort(vs.begin(), vs.end(), cmpByName()); //按姓名排序
#58     xrv(print(vs.begin(), vs.end(), "sort by name:\n", "\n"));
#59 }

```

上述程序中，算法 `max_element()` 和 `sort()` 的第一次调用 `max_element(vs.begin(), vs.end())` 和 `sort(vs.begin(), vs.end())` 默认以 `Student` 类中定义的 `operator <` 作为比较准则，即按照成绩和小于号进行比较。

类 `Student` 的友元函数 `cmpByID()` 是二元操作，它在算法 `max_element()` 和 `sort()` 的第二次调用时作为它们的第三个的参数，表示以 `Student` 对象的数据成员 `sid` 和小于号作

为比较和排序的准则。二元函数对象 `cmpByName()` 分别在算法 `max_element()` 和 `sort()` 的第三次调用中作为它们的第三个参数,表示以 `Student` 对象的数据成员 `name` 和小于号作为比较和排序的准则。

分析程序的输出结果,体会二元函数对象的定义和应用。然后用 `lambda` 函数分别实现这 3 个比较准则。

2. 数组外包类 `array_wrapper`

所谓数组外包类,是指能够改变普通数组使之能够与 STL 算法和容器相匹配和混用的类。

【例 1.10.4】数组外包类 `wrapper` 的应用示例。

```
#01  #include <iostream>
#02  #include <algorithm>
#03  #include <iterator>
#04  #include <cassert>
#05
#06  template<typename T>
#07  class array_wrapper { //数组适配器:改变普通数组使之能用于 STL 算法和容器
#08  private:
#09      T* start;           //数组元素的首地址
#10      size_t sz;         //数组元素的个数
#11
#12      array_wrapper(const array_wrapper<T>&) = delete; //禁止复制
#13      array_wrapper& operator=(const array_wrapper<T>&)=delete; //禁止赋值
#14
#15  public:
#16      array_wrapper(T* a, size_t n)           //对数组 a 适配
#17          :start(a), sz(n)
#18      {}
#19
#20      // type definitions
#21      typedef T          value_type;          //数组元素类型
#22      typedef T* iterator;                    //适配器迭代器
#23      typedef const T* const_iterator;        //适配器 const 迭代器
#24      typedef T& reference;                   //元素类型引用
#25      typedef const T& const_reference;       //元素类型 const 引用
#26      typedef size_t     size_type;           //下标类型
#27      typedef ptrdiff_t  difference_type;     //迭代器相对距离类型
#28
#29      // iterator support
#30      iterator begin() { return start; }       //区间首地址
#31      const_iterator begin() const { return start; } //区间首地址:const 版本
#32      iterator end() { return start + sz; }    //区间终点
#33      const_iterator end() const { return start + sz; } //区间终点:const 版本
#34
#35      // direct element access
#36      const_reference operator[] (size_t i) const { //用作右值的下标运算
```



```

#37         assert(i >= 0 && i < sz);
#38         return start[i];
#39     }
#40     reference operator[] (size_t i) {           //用作左值的下标运算
#41         return const_cast<int&>(std::as_const(*this)[i]);
#42     }
#43
#44     // size is constant
#45     size_type size() const { return sz; }       //获取元素个数
#46     size_type max_size() const { return sz; }   //获取元素个数的最大值
#47
#48     // conversion to ordinary array
#49     T* c_arr() { return start; }               //转换成普通数组
#50 };
#51
#52 int main() {
#53     int a[]{ 2, 1, 3, 6, 5, 3, 3, 4 };
#54     auto n = sizeof(a) / sizeof(*a);
#55
#56     array_wrapper<int> aw(a, n);                //对数组 a 进行适配
#57
#58     auto m = std::count(aw.begin(), aw.end(), 3); //统计 3 的个数
#59     std::cout << m << std::endl;               //3
#60
#61     std::cout << "firstly, a: ";
#62     array_wrapper<int>::iterator iter;          //用迭代器方式输出元素
#63     for (iter = aw.begin(); iter != aw.end(); ++iter)
#64         std::cout << *iter << "\t";
#65     std::cout << std::endl;
#66
#67     std::for_each(aw.begin(), aw.end(),
#68         [](int& x) {x *= 2; });                 //用于算法:扩大至 2 倍
#69
#70     std::cout << "after dbl, a: ";
#71     std::ostream_iterator<int> screen(std::cout, "\t");
#72     std::copy(aw.begin(), aw.end(), screen);    //用于算法:区间复制
#73     std::cout << std::endl;
#74
#75     std::sort(aw.begin(), aw.end());            //用于算法:排序
#76     iter = std::unique(aw.begin(), aw.end());   //去除重复元素
#77     std::cout << "after unique, a: ";
#78     std::copy(aw.begin(), iter, screen);        //区间复制
#79     std::cout << std::endl;
#80
#81     std::cout << "finally, a: ";
#82     for (size_t i = 0; i != aw.size(); ++i)    //下标方式输出元素
#83         std::cout << aw[i] << "\t";
#84     std::cout << std::endl;
#85 }

```

在上述程序中, 类 `array_wrapper` 提供了构造函数 `array_wrapper(T* a, size_t n)` 以与普通数组进行配接; 并定义了系列数据类型 `value_type`、`iterator`、`reference`、`size_type`、`difference_type` 等; 并提供了 `begin()` 和 `end()` 分别访问区间首末元素位置; 提供了 `operator[]` 提供随机访问能力; 提供了 `size()` 访问元素个数的操作; 提供了 `c_arr()` 操作把数组外包类还原为普通数组。由于该外包类只访问数组元素, 而不具备复制源数组的能力, 因此把该类的复制构造函数和赋值运算符函数设为显式 “=delete” 而禁止了对该类对象的复制。

在 `main()` 函数中, `array_wrapper<int> aw(a, n)` 把 `int` 型数组 `a` 包装成 `array_wrapper` 类的对象 `aw`。并在随后的各行分别用算法 `count()`、`for_each()`、`copy()`、`sort()`、`unique()` 测试了外包类的各项操作能力。

实验题目与提示

1. 定义 `Student` 类型描述学生的学号、姓名、性别和成绩, 选用一种容器 (`vector`、`list`、`deque`、`set`、`map`) 存放它的多个对象。

- (1) 统计男生和女生的人数。
- (2) 计算平均成绩、最高成绩和最低成绩。
- (3) 统计各分数段的人数。
- (4) 分别按照学号、姓名和成绩排序。
- (5) 统计具有某个相同姓名的学生人数。
- (6) 查找同名的某学号的学生, 并删除他的所有信息。

【提示】如下定义 `Student` 类型作为容器元素的类型。

```
#01     enum Gender {female, male};           //描述性别
#02
#03     class Student {                       //描述学生
#04     private:
#05         string id;                        //学号
#06         string name;                      //姓名
#07         Gender gender;                    //性别
#08         double score;                     //成绩
#09     };
```

无论选用哪一种容器存放 `Student` 类的对象, 都需要选用适当的算法并定义正确的函数对象来完成本题的所有要求。在向容器中存放对象时, 这 5 种容器都支持 `insert()` 方法, 只是各自所带的函数不同。此外, 序列式容器支持更简单地插入运算方法, `vector`、`list` 和 `deque` 都可以使用 `push_back()` 方法, `deque` 还可以使用 `push_front()` 方法。

(1) 选用 STL 算法 `count_if()`, 并定义正确的函数对象作为它的参数。

(2) 分别选用 STL 算法 `accumulate()`、`max_element()`、`min_element()` 来计算容器中学生对象的总成绩、最高成绩和最低成绩。对于算法 `accumulate()` 来说, 要定义正确的累加准则 (累加数据成员 `score`) 作为它的第三个参数。对于算法 `max_element()`、`min_element()` 来说, 要定义正确的比较准则 (比较数据成员 `score`) 作为它们的第三个

参数。

(3) 选用 STL 算法 `count_if()`，并定义正确的函数对象作为它的参数。

(4) 选用 STL 算法 `sort()` (`list` 容器则用其自有的成员函数 `sort()`)，并定义正确的函数对象作为比较准则 (定义 3 个函数对象，分别以学号、姓名和成绩作为比较对象)。

(5) 排序之后，选用算法 `equal_range()` (或 `lower_bound()` 和 `upper_bound()`)，并定义正确的函数对象作为它的参数 (以数据成员 `name` 作为比较对象) 进行查找。

(6) 在上一题所查找的区间中，应用算法 `find_if()` 并定义正确的函数对象 (以数据成员 `id` 作为比较对象) 进行查找，然后应用容器的成员函数 `erase()` 删除所查找到的元素。

2. 从键盘输入一行字符，应用容器类，统计每个字符出现的次数。

【提示】选用 `map` 容器，以字符作为容器的键值，该字符出现的次数作为元素值。定义容器对象如下：

```
std::map<char, int> m;
```

以 `map` 容器的下标运算或 `insert()` 函数向容器对象 `m` 中存放数据，通过成员函数 `find()` 查找某个字符的次数。

3. 以 `map/multimap` 容器实现一个小型词典，输出一个单词的所有释义。

【提示】定义如下 `map` 容器对象。

```
std::map<std::string, std::set<std::string> > m;
```

以单词作为容器的键值，该单词所有的释义都存放在一个 `set` 容器中。以 `map` 容器的下标运算或 `insert()` 函数向容器对象 `m` 中存放数据，通过成员函数 `find()` 查找某个单词的释义。

实验 11 继承与派生

实验目的与要求

1. 实验目的

- (1) 理解继承的概念。
- (2) 掌握派生类构造函数的定义。
- (3) 理解赋值兼容规则。

2. 实验要求

- (1) 能够正确应用继承描述多个类之间的关系。
- (2) 熟练定义派生类的构造函数。
- (3) 正确分析派生类对象的构造和析构过程。
- (4) 能够在合适的场合正确应用该规则。

实验过程与示例

在本次实验中，需要重点练习派生类的定义，并且能够在实际应用中根据需求，正确选用继承以描述概念之间的“is-a”关系，能够正确定义派生类中的构造函数，并能够正确分析派生类对象的构造过程。

除与本书配套的主教材上的相关例题外，以下是一些较典型的示例，它们分别对派生类对象的构造和析构、对象成员的构造和析构、赋值兼容规则等应用进行了示范。

1. 派生类对象的构造和析构

【例 1.11.1】 Person 类、Student 类和 Undergraduate 类的应用示例。

本例重在练习派生类的构造函数的定义，以及派生类对象构造过程的分析。

```
#01     #include <iostream>
#02     #include <string>
#03
#04     class Person {                               //基类:人员类
#05     protected:                                   //注意权限
#06         std::string name;                         //姓名
#07         int age;                                   //年龄
#08     public:
#09         Person(const char* n, int a):name(n), age(a) {} //构造函数
```

```

#10         void PrintPerson(std::ostream& os = std::cout) const {
#11             os << name << "\t" << age << "\n";
#12         }
#13     };
#14
#15     class Student : public Person {           //从 Person 类继承
#16     protected:                               //注意权限
#17         double score;                         //增加新的属性:分数
#18     public:
#19         Student(const char* n, int a, double s) //派生类的构造函数
#20             :Person(n, a), score(s)           //调用基类的构造函数
#21         {}
#22         void PrintStudent(std::ostream& os = std::cout) const {
#23             os << name << "\t" << age << "\t"
#24             << score << std::endl;
#25         }
#26     };
#27
#28     class Undergraduate : public Student { //从 Student 类继承
#29     private:
#30         std::string speciality;           //增加新的属性:专业
#31     public:
#32         Undergraduate(const char* n, int a, double s, char* sp)
#33             :Student(n, a, s), speciality(sp) //调用直接基类的构造函数
#34         {}
#35         void PrintUndergraduate(std::ostream& os = std::cout) const {
#36             os << name << "\t" << age << "\t"
#37             << score << "\t" << speciality << std::endl;
#38         }
#39     };
#40
#41     int main() {
#42         Person Tom("Tom", 20);               //构造基类的对象
#43         Tom.PrintPerson();
#44
#45         Student Jerry("Jerry", 18, 90);      //构造派生类的对象
#46         Jerry.PrintStudent();
#47
#48         Undergraduate Mickey("Mickey", 22, 85, "MBA");//构造派生类的对象
#49         Mickey.PrintUndergraduate();
#50     }

```

这是关于派生类构造函数定义的一个典型例子。每个派生类需要调用其直接基类的构造函数以初始化从直接基类继承而来的数据成员，但是不直接调用其间接基类的构造函数。这是在有层继承时需要注意的一个问题。

在分析本程序时，需要注意如下几个问题：

- (1) 请分析 main() 函数中 3 个对象 Tom、Jerry、Mickey 构造、析构的顺序。
- (2) 派生类在调用基类构造函数时通过成员初始化列表语法来初始化从基类得到的

数据成员，仔细分析该语法形式，并思考在设置派生类构造函数参数时要注意的问题。

(3) 派生类的输出函数（如 Student 类的 PrintStudent() 函数、Undergraduate 类的 PrintUndergraduate() 函数）为要访问基类的数据成员，一般把它们的访问权限设置为 protected。比较 private 与 protected 的区别。

2. 对象成员的构造和析构

【例 1.11.2】圆类 Circle 和圆环类 Ring 的应用示例。

本例重在练习对象成员构造和析构顺序的分析。

```
#01    #include <cmath>
#02    #include "xr.hpp"
#03
#04    class Point {                                //描述平面上的点
#05    private:
#06        double x, y;                            //横纵坐标
#07    public:
#08        Point(double a, double b)                //构造函数
#09            :x(a), y(b)
#10        {}
#11        double distance(const Point& p) const { //计算两点之间的距离
#12            double dx = x - p.x;
#13            double dy = y - p.y;
#14            return sqrt(dx * dx + dy * dy);
#15        }
#16    };
#17
#18    class Circle {                                //圆类
#19    protected:                                    //注意访问权限
#20        Point center;                             //对象成员:圆的中心
#21        double radius;                            //圆的半径
#22    public:
#23        Circle(const Point& p, double r)          //构造函数
#24            :center(p), radius(r)                 //初始化对象成员
#25        {}
#26        double Area() const {                    //计算圆的面积
#27            return 3.14 * radius * radius;
#28        }
#29        bool IsPointIn(const Point& p) const { //判断点是否在圆内
#30            return p.distance(center) < radius;
#31        }
#32    };
#33
#34    class Ring : public Circle {                   //派生类:圆环类
#35    private:
#36        double outerRadius;                       //增加新的属性:外圆半径
#37    public:
#38        Ring(const Point& p, double r, double outer) //构造函数
#39            :Circle(p, r), outerRadius(outer)      //调用基类的构造函数
```

```

#40         {}
#41         double Area() const { //计算面积
#42             return 3.14 * outerRadius * outerRadius
#43                 - 3.14 * radius * radius;
#44         }
#45         bool IsPointIn(const Point& p) const { //判断点是否在圆环内
#46             double d = p.distance(center); //计算到圆心的距离
#47             return d > radius && d < outerRadius; //是否在两个半径之间
#48         }
#49     };
#50
#51     int main() {
#52         Point p(3, 4); //生成圆心点
#53
#54         Circle c(p, 10); //构造一个圆
#55         xr(c.Area()); //计算圆的面积
#56         xr(c.IsPointIn(p)); //判断点与圆的位置关系
#57
#58         Ring cq(p, 20, 40); //构造一个圆环
#59         xr(cq.Area()); //计算圆环的面积
#60         xr(cq.IsPointIn(p)); //判断点与圆环的位置关系
#61     }

```

分析上述程序的输出结果，并思考下列问题：

(1) 分析圆环类 `Ring` 构造函数的定义，它是如何构造通过继承得到的基类的数据成员的呢？

(2) 请分析 `main()` 函数中对象 `c` 和 `cq` 的构造过程，并通过宏 `xr` 来追踪相应函数的调用过程。

(3) 基类 `Circle` 和派生类 `Ring` 中同时定义有成员函数 `Area()` 和 `IsPointIn()`，请分析 `main()` 函数中对这两个函数的调用情况。

3. 通过继承扩展类的功能

【例 1.11.3】扩展 STL 容器 `forward_list` 类的应用示例。

主教材在例 7.13 中提供了一个示例，讲解如何通过继承获得类的现有功能，同时在此基础上扩展增加其他功能，也可以通过显式 “=delete” 去除从基类中获得、但是不适用于本类的那些功能。现将其基本做法复制如下：

```

#01     #include <list>
#02
#03     template <class T>
#04     class my_list : public std::list<T> { //在基类现有功能上扩展
#05     public:
#06         my_list(T* beg, T* end):std::list<T>(beg, end) {}
#07
#08         const T& at(size_t pos) const { //增添新方法
#09             size_t idx{0}; //定义下标,从零开始
#10             auto iter{this->begin()}; //定义迭代器,从头开始

```

```

#11         while (idx != pos && iter != this->end()) //尚未到达目的地
#12             ++idx, ++iter;                        //同步前移
#13         return *iter;                             //返回下标对应的元素
#14     }
#15
#16     const T& front() const {return at(0);} //覆盖从基类得到的方法
#17 };

```

上述程序在继承 STL 容器 `list<>` 的基础上, 增添了新方法 `at()` 模拟实现下标函数。尽管这不是一个效率很高的做法, 但是提供了“通过继承扩展基类功能”的方法示例。

STL 容器 `forward_list<>` 是一个轻量级的链表类。它实现为单链表 (`list<>` 实现为双链表), 优点是内存占用较少, 各种操作的性能也快。但是由于它行为受限, 很多函数都不提供。`forward_list<>` 不提供成员函数 `size()` 获取容器中元素的个数。`forward_list<>` 没有指向最末元素的锚点, 不提供处理最末元素的成员函数 `back()`、`push_back()`、`pop_back()` 等。`forward_list<>` 类的完整功能请参考标准库。

请仿照上述示例, 继承 `forward_list<>` 类, 扩展其功能, 实现自己的单链表类 `linked_list<>`。在其中: ①提供成员函数 `size()` 以获取容器中元素的个数; ②提供成员函数 `at()` 模拟实现下标函数, 实现常用的 `front()` 等成员访问操作, 注意同时提供这些成员函数的左值版和右值版; ③提供处理最末元素的成员函数 `back()`、`push_back()`、`pop_back()` 等; ④提供一个有序插入函数 `insert_ordered(e, comp)` 把元素 `e` 插入单链表中后保持有序 (默认非递减排列), `comp` 表示比较准则。

实验题目与提示

1. 编写程序, 实现下列要求。

(1) 定义基类 `Employee`, 并分别从该类派生出 `Manager` 类和 `HourlyWorker` 类。

(2) `Employee` 类的属性包括姓名 (`name`) 和工号 (`ID`), `Manager` 类的属性包括工资 (`salary`), `HourlyWorker` 类的属性包括 `wage` 和 `hours`, 分别表示每小时工资数和月工作小时数。

(3) 在各个类中提供必要的操作, 以构造、析构、修改、输出对象。

【提示】按照要求定义各基类和派生类。本题的重点在于在各类中提供相同的成员函数计算各自的工资, 并能正确计算。

2. 实现计算机 `Computer` 的面向对象描述。

(1) 每个计算机配件 (`ComputerAccessory`) 都有制造商 (`manufacturer`) 和价格 (`price`) 两种属性。

(2) 主板 (`MotherBoard`)、内存 (`Memory`)、显示器 (`Monitor`) 是典型的计算机配件。芯片组 (`chipset`)、内存容量 (`capacity`)、显示器类型 (`mtype`) 分别是这 3 种配件的重要特征。

(3) 主板、内存、显示器是计算机 `Computer` 的重要组成部分。

(4) 请定义类描述各类事物, 并提供必要的操作, 计算装配一台计算机需要的价钱。

【提示】(1) 定义计算机配件类 `ComputerAccessory` 作为基类, 以 `manufacturer` 和 `price` 作为数据成员, 设置成员函数 `GetPrice()` 访问数据成员 `price`。

(2) 定义主板类 `MotherBoard` 从 `ComputerAccessory` 类派生, 以芯片组 `chipset` 作为数据成员。定义内存类 `Memory` 从 `ComputerAccessory` 类派生, 以内存容量 `capacity` 作为数据成员。定义显示器类 `Monitor` 从 `ComputerAccessory` 类派生, 以显示器类型 `mtype` 作为数据成员。在 3 个类中重定义基类的 `GetPrice()` 函数计算各自的价钱, 并提供合适的构造函数和输出函数。

(3) 定义计算机类 `Computer`, 以 `MotherBoard`、`Memory`、`Monitor` 这 3 个类的对象作为数据成员。在 `Computer` 类中定义 `GetPrice()` 函数计算计算机的价钱, 并提供合适的构造函数和输出函数。

(4) 在 `main()` 函数中定义 `Computer` 类的对象, 调用 `GetPrice()` 函数计算它的价钱。

3. 编写程序, 实现下列要求。

(1) 定义人员类 `Person`, 其属性 (protected 权限) 有姓名、性别、年龄。

(2) 派生出学生类 `Student`, 添加属性: 学号、入学时间和入学成绩。

(3) 从 `Person` 类再派生出教师类 `Teacher`, 添加属性: 职务、部门、工作时间。

(4) 由 `Student` 类派生出研究生类 `Graduate`, 添加属性: 研究方向和导师。

(5) 由 `Graduate` 和 `Teacher` 共同派生出研究生导师类 `Supervisor`。

(6) 在每个类中提供必要的成员函数, 分别实现对象的构造、析构、输入和输出等操作。

(7) 在 `main()` 函数中定义各种对象, 并分别测试它们的操作。

【提示】本例涉及多重继承及虚拟继承。

(1) 定义人员类 `Person` 作为基类, 并提供合适的构造函数、访问函数和输出函数。

(2) 以 `Person` 类作为虚基类, 定义派生类 `Student`。以 `Person` 类作为虚基类, 定义派生类 `Teacher`。

(3) 以 `Student` 类作为基类, 定义派生类 `Graduate`。以 `Graduate` 类和 `Teacher` 类作为基类, 定义派生类 `Supervisor`。

(4) 在各类中提供合适的构造函数、访问函数, 并定义成员函数 `Print()` 输出各自的信息。在 `main()` 函数中定义对象并测试各自的 `Print()` 函数。

4. 拓展练习。

MFC (microsoft foundation library) 是在 VC++ 环境下编写可视化程序的重要工具库。如同 STL 提供了很多通用函数和工具, MFC 则以 MVC 模式提供了实现 Windows 平台下可视化程序开发的重要工具, 如文档类 `CDocument`、视图类 `CView`、应用程序类 `CWinApp`。在用户开发程序的过程, 就是继承 MFC 中这些现有的类, 然后在其中添加数据和功能。请在自己熟悉的 VC++ 环境中, 尝试编写 MFC 应用程序, 并分析理解其中类之间的关系, 然后在适当的地方添加自己的数据和功能实现简单的应用, 如在程序界面上输出本班同学的基本信息, 或者绘制简单的图形。体会程序中各类之间的继承关系。

实验 12 虚函数与多态性

实验目的与要求

1. 实验目的

- (1) 理解虚函数和动态关联的概念。
- (2) 理解纯虚函数和抽象基类的概念。
- (3) 理解多态性的概念。

2. 实验要求

- (1) 能够正确定义并应用虚函数实现程序的动态关联。
- (2) 能够正确定义纯虚函数、抽象基类及其派生类。
- (3) 能够正确应用多态性解决比较简单的问题。

实验过程与示例

在本次实验中,需要重点练习虚函数、纯虚函数和抽象基类等的重要概念及其应用,并能够在实际应用中,根据需求正确定义虚函数、纯虚函数以解决多态性相关的问题。

除与本书配套的主教材上的相关例题外,以下是一些较典型的示例,它们分别对虚函数、纯虚函数和抽象类及多态性和对象操作的一般化等概念及其应用进行了示范。

1. 虚函数与动态关联

【例 1.12.1】Person-Student-Worker 类的应用示例。

本例重在练习虚函数及其在程序的动态关联中的作用。注意成员函数 Print()和 DoSomething()的性质。

```
#01     #include <string>
#02     #include "xr.hpp"
#03
#04     class Person {                                //人员类作为基类
#05     private:
#06         int pid;                                  //身份证号
#07         std::string name;                          //姓名
#08         int age;                                   //年龄
#09     public:
#10         Person(int id = 0, const char* str = "", int a = 0) noexcept
```

```

#11         :pid(id), name(str), age(a)    //初始化数据成员
#12     {}
#13     /*virtual*/ void Print() const {    //暂时保留注释
#14         std::cout << pid << ": " << name << ": " << age << "\t";
#15     }
#16     /*virtual*/ void DoSomething() const { //暂时保留注释
#17         std::cout << "I am a person, I need doing something.\n";
#18     }
#19 };
#20
#21 class Student : public Person {          //学生类作为派生类
#22 private:
#23     double score;                        //成绩
#24 public:
#25     Student(int id = 0, const char* str = "", int a = 0, double s = 0) noexcept
#26         :Person(id, str, a), score(s)    //初始化基类成员和自己的数据成员
#27     {}
#28     void Print() const {                 //输出 Student 类的信息
#29         Person::Print();                 //调用基类的函数
#30         std::cout << "score: " << score << std::endl;
#31     }
#32     void DoSomething() const {           //说明对象的工作
#33         std::cout << "I am a student, I need studying.\n";
#34     }
#35 };
#36
#37 class Worker : public Person {           //派生类:员工类
#38 private:
#39     double salary;                       //工资
#40 public:
#41     Worker(int id = 0, const char* str = "", int a = 0, double s = 0) noexcept
#42         :Person(id, str, a), salary(s)   //初始化基类成员和自己的数据成员
#43     {}
#44     void Print() const {                 //输出 Worker 类的信息
#45         Person::Print();                 //调用基类的函数
#46         std::cout << "salary: " << salary << std::endl;
#47     }
#48     void DoSomething() const {           //说明对象的工作
#49         std::cout << "I am a worker, I need working.\n";
#50     }
#51 };
#52
#53 void testByObj(Person p) {               //以值形式的对象作为参数
#54     p.Print();
#55     p.DoSomething();
#56 }
#57 void testByPtr(Person* p) {              //以对象指针作为参数
#58     p->Print();
#59     p->DoSomething();

```

```

#60     }
#61     void testByRef(Person& p) {                //以对象引用作为参数
#62         p.Print();
#63         p.DoSomething();
#64     }
#65
#66     int main()
#67     {
#68         Person p(200801, "Tom", 20);
#69         Student s(200802, "Jerry", 22, 85);
#70         Worker w(200803, "Mickey", 24, 2000);
#71
#72         xrv(testByObj(p)); xrv(testByObj(s)); xrv(testByObj(w));
#73         xrv(testByPtr(&p)); xrv(testByPtr(&s)); xrv(testByPtr(&w));
#74         xrv(testByRef(p)); xrv(testByRef(s)); xrv(testByRef(w));
#75     }

```

上述程序中, `Person` 类是基类, 由它派生出 `Student` 类和 `Worker` 类, 在每个类中都提供了成员函数 `Print()` 和 `DoSomething()` 分别输出对象信息、说明对象的工作。同时定义 3 个函数 `testByObj()`、`testByPtr()`、`testByRef()` 分别以对象、对象指针、对象引用测试成员函数 `Print()` 和 `DoSomething()` 的动态关联情况。

分析程序的输出结果, 并思考下列问题:

(1) 上述程序的输出结果是否达到了程序的预期, 若没有, 请思考问题所在, 并改正程序中的问题。

(2) 分析改正之后程序的输出结果, 并总结 3 个函数 `testByObj()`、`testByPtr()`、`testByRef()` 实现动态关联的情况。

2. 纯虚函数和抽象基类

【例 1.12.2】对象操作的一般化的应用示例。

```

#01     #include <iostream>
#02     #include <vector>
#03
#04     class Shape {                                //抽象基类
#05     public:
#06         virtual const char* NameOfShape() const = 0; //纯虚函数:输出类名
#07         virtual void Draw() const = 0;             //纯虚函数:模拟绘制
#08         virtual double Area() const = 0;           //纯虚函数:计算面积
#09     };
#10
#11     class Point : public Shape {                  //派生类:继承接口
#12     public:
#13         Point() noexcept {}                       //以下重定义纯虚函数
#14         const char* NameOfShape() const {return "Point";}
#15         void Draw() const {std::cout << "Drawing a Point...\n";}
#16         double Area() const {return 0;}
#17     };
#18

```

```

#19 class Square : public Shape { //派生类:继承接口
#20 private:
#21     double length; //增加自己的属性
#22 public:
#23     Square(double l = 0) noexcept :length(l) {}
#24     //以下重定义纯虚函数
#25     const char* NameOfShape() const {return "Square";}
#26     void Draw() const {std::cout << "Drawing a Square...\n";}
#27     double Area() const {return length * length;}
#28 };
#29
#30 class Rectangle : public Shape { //派生类:继承接口
#31 private:
#32     double length, width; //增加自己的属性
#33 public:
#34     Rectangle(double l = 0, double w = 0) noexcept
#35         :length(l), width(w)
#36     {} //以下重定义纯虚函数
#37     const char* NameOfShape() const {return "Rectangle";}
#38     void Draw() const {std::cout << "Drawing a Rectangle...\n";}
#39     double Area() const {return length * width;}
#40 };
#41
#42 class Circle : public Shape { //派生类:继承接口
#43 private:
#44     double radius; //增加自己的属性
#45 public:
#46     Circle(double r = 0) noexcept :radius(r)
#47     {} //以下重定义纯虚函数
#48     const char* NameOfShape() const {return "Circle";}
#49     void Draw() const {std::cout << "Drawing a Circle...\n";}
#50     double Area() const {return 3.14 * radius * radius;}
#51 };
#52
#53 int main()
#54 {
#55     std::vector<Shape*> vs; //定义存放基类的容器
#56     vs.push_back(new Point); //存放派生类 Point 的对象
#57     vs.push_back(new Square(10)); //存放派生类 Square 的对象
#58     vs.push_back(new Rectangle(10, 20)); //存放派生类 Rectangle 的对象
#59     vs.push_back(new Circle(10)); //存放派生类 Circle 的对象
#60
#61     for (size_t i = 0; i != vs.size(); ++i) { //统一操作对象
#62         std::cout << vs[i]->NameOfShape() << ": "; //输出各自的类名
#63         vs[i]->Draw(); //模拟绘制各对象
#64         std::cout << "Area: " << vs[i]->Area() << std::endl;
#65         //计算各对象面积
#66     }
#67

```

```
#68         for (size_t i = 0; i != vs.size(); ++i)
#69             delete vs[i];           //释放各堆对象
#70     }
```

本程序示范了“对象操作一般化”的优点。本程序的重点在于向基类容器中存放派生类的对象，并对各派生类对象进行了统一的操作（不用考虑各自的差别和实现细节），从而简化了程序的处理流程。这一切的基础就在于：首先定义抽象基类 `Shape`，并以纯虚函数形式规定其派生类的共同接口；然后在派生类中重定义各个纯虚函数。

分析程序的输出结果，并思考下列问题：

(1) 如果没有纯虚函数进行动态关联，则程序中需要设置选择结构（if/else if 或 switch/case）对数组中存放的对象的类型进行判断，然后调用各自的操作。请尝试在 `main()` 函数的第一个 for 循环中应用选择结构实现这个处理方法。

(2) 请增加一个派生类 `Line`，并在其中定义成员函数 `NameOfShape()`、`Draw()`，并向 `main()` 函数容器 `vs` 中增加一个 `Line` 类的对象，测试对该对象的操作。

实验题目与提示

1. 参考例 1.12.2，编写程序，实现如下要求。

(1) 定义抽象基类 `Shape`，在其中定义 4 个纯虚函数作为接口：函数 `NameOfType()` 输出类名、函数 `Draw()` 模拟绘制、函数 `Area()` 计算面积、函数 `Perimeter()` 计算周长。

(2) 从基类 `Shape` 派生出 5 个派生类：`Circle`（圆形）、`Square`（正方形）、`Rectangle`（矩形）、`Trapezoid`（梯形）、`Triangle`（三角形）。在派生类中实现各个接口函数。

(3) 一个卡通娃娃由多个图形构成，如三角形、圆、正方形、长方形、梯形，计算该图案的面积及周长。

【提示】本题的关键在于抽象基类和纯虚函数的定义。按照 (1)、(2) 的要求，首先定义类 `Shape` 作为抽象基类，并在类中定义 4 个纯虚函数。然后以 `Shape` 类作为基类，定义 5 个派生类，在每个派生类中定义 4 个接口函数，分别输出类名、绘制图形、计算面积、计算周长。

(3) 在 `main()` 函数中定义基类指针的数组或容器，分别存放派生类的对象，然后通过 for 循环统一操作计算各图形的面积和周长。

2. 编写程序，实现下列要求。

(1) 定义员工类 `Employee` 作为基类，其属性有姓名（name）、工号（ID）、参加工作时间（worktime）。

(2) 从员工类派生出经理（`Manager`）、销售人员（`Salesman`）、技术工人（`Technician`）和合同工（`ContractWorker`）。类 `Manager` 的属性有月工资（salary）。类 `Salesman` 的属性有基本工资（basic）、每月销售额（sales）、按销售额提成的比例（rate）。类 `Technician` 的属性有技术职称（tech_post）和职称工资（post_salary）。类 `ContractWorker` 的属性有每小时工资数（wage）和月工作时数（hours）。

(3) 在所有类中提供必要的构造、析构、输入、输出操作，并定义函数 `ComputeSalary()`

计算工资。

(4) 设计简易的工资管理系统计算公司不同员工的工资。按照工号顺序, 输出员工每月的工资报表。

【提示】本题的关键在于抽象基类和纯虚函数的定义, 涉及的纯虚函数有输入函数、输出函数和计算工资的函数。按照 (1)、(2)、(3) 的要求, 分别定义基类和各派生类, 在其中定义各自的数据成员, 以及输入、输出和计算工资的函数。(4) 在 `main()` 函数中定义基类指针的数组或容器, 分别存放派生类的对象, 输入各自的信息, 计算各自的工资, 按照工号排序, 然后输出各自的工号、姓名和工资。

3. 拓展练习。

学习 MFC 中 Dialog-based 应用程序开发, 仿照 Windows 操作系统自带的计算器程序 (`calc.exe`), 设计实现自己的简易计算器, 并体会其中虚函数的用途。

实验 13 C++的 I/O 流

实验目的与要求

1. 实验目的

- (1) 掌握 C++标准流常用成员函数的用法。
- (2) 掌握 C++格式控制的各种方式。
- (3) 掌握 C++文件流的操作。

2. 实验要求

- (1) 熟练应用常见 I/O 函数并能正确定义流运算符函数。
- (2) 熟练应用格式控制的各种方式实现数据的格式化。
- (3) 熟练应用 C++文件的读写操作实现对象数据的存储和读取。

实验过程与示例

在本次实验中，需要以文件流为载体，重点练习 C++的 I/O 流所提供的各种功能和操作，如常用于读写操作的成员函数，以及格式控制的标志位、成员函数和流操纵算子。要求能够在应用中根据实际需求，选用适合的文件流类及其操作解决实际问题。

除与本书配套的主教材上的相关例题外，以下是一些较典型的示例，它们分别对流操纵算子的定义与应用、全局函数 `getline()` 提取字符内容、流迭代器用于文件操作等应用进行了示范。

【例 1.13.1】流操纵算子的定义与应用的应用示例。

本书在实验 1 的例 1.1.3 中曾经给出一个输出菱形图案的示例。在本次实验中，将应用流操作算子简化菱形图案输出的程序。在菱形图案输出的过程中，大部分工作用于连续输出多个字符，如输出空格和星号。为了使程序过程简明清晰，本例将首先以流操纵算子的方式实现多个字符的连续输出，然后把菱形图案输出的过程定义为一个参数化的流操纵算子。

```
#01      #include <iostream>
#02
#03      class nchars {                               //封装要连续输出的数据
#04      private:
#05          size_t n;                                //要重复输出的次数
#06          char ch;                                  //要输出的字符
```



```

#07     public:
#08         nchars(size_t m, char c) :n(m), ch(c) {}//保存要连续输出的数据
#09         friend std::ostream& operator << (std::ostream& os,
#10             const nchars& rhs) {                //以下连续输出多个字符
#11             for (size_t i = 0; i != rhs.n; ++i)
#12                 os << rhs.ch;
#13             return os;
#14         }
#15     };
#16
#17     class diamond {                                //输出菱形图案的算子类型
#18     private:
#19         size_t row;                                //菱形中上三角的行数
#20         char cfill;                                //组成菱形的字符
#21     public:
#22         diamond(size_t r, char c) :row(r), cfill(c) {} //菱形的组成数据
#23         friend std::ostream& operator << (std::ostream& os,
#24             const diamond& rhs) {                //以下实现菱形绘制
#25             for (size_t i = 1; i <= rhs.row; ++i) { //第一步:绘制上三角
#26                 os << nchars(rhs.row-i, ' ')    //先输出 n-i 个空格
#27                     << nchars(2*i-1, rhs.cfill) //再输出 2i-1 个字符
#28                     << std::endl;                //最后换行
#29             }
#30             for (size_t i = 2; i <= rhs.row; ++i) { //第二步:绘制下三角
#31                 os << nchars(i-1, ' ')          //输出 i-1 个空格
#32                     << nchars(2*(rhs.row-i)+1, rhs.cfill)
#33                                     //输出 2(n-i)+1 个字符
#34                     << std::endl;                //最后换行
#35             }
#36             return os;
#37         }
#38     };
#39
#40     int main() {
#41         std::cout << diamond(14, '*') << std::endl; //定义操纵算子绘制菱形
#42     }

```

定义一个参数化的流操纵算子其实很容易,关键要在类中重载流插入运算符函数。本例程序首先定义了类 `nchars` 封装要连续输出的字符及其次数,然后以流插入运算符函数形式实现连续输出的过程,这个函数也使 `nchars` 成为一个流操纵算子类型。

本例程序同时定义了类 `diamond` 封装菱形图案的组成元素及其绘制过程。组成元素主要包括菱形的行数(用上三角形的行数表示)和填充菱形的字符。绘制过程以流插入运算符函数的形式实现,其中以流操纵算子形式运用 `nchars` 对象实现多个字符的连续输出,如 `os<<nchars(i-1,'')` 会输出 `i-1` 个空格。类 `diamond` 中的流插入运算符函数使 `diamond` 成为一个参数化的流操纵算子类型。

`main()` 函数非常轻松地以参数化的流操纵算子实现菱形图案的绘制。算子 `diamond(14, '*')` 说明菱形有 14 行,填充字符为星号,函数调用表达式 `std::cout<<diamond(14, '*')` 调用

类 `diamond` 的流插入运算符函数实现了菱形的绘制。

【例 1.13.2】给文本文件添加行号。

正如本书中源代码所示，每一行内容都标以行号以便分析。为了编写行号，首先要对文本文件的行数进行统计。可以应用全局函数 `getline()` 读取文件中的一行字符内容，函数 `getline()` 可以一次读取输入流中的一行字符内容，并默认以回车符为分隔符。对于以回车键结束的文本文件，如 C/C++ 源文件和头文件，可以如下列程序所示在左侧编以行号。

```
#01     #include <fstream>
#02     #include <sstream>                                //for ostringstream
#03     #include <string>
#04     #include <iomanip>
#05     #include "verify.hpp"
#06
#07     std::string int2str(int n, int w = 2) {              //把整数转换成字符串
#08         std::ostringstream oss;                        //定义字符串输出流对象
#09         oss << "#" << std::setfill('0');                //以#为前导,以0填充
#10         oss << std::setw(w) << n;                       //占位宽度为w
#11         return oss.str();                                //返回格式化后的字符串
#12     }
#13
#14     int main() {
#15         std::string srcfile("main.cpp");                //源数据所在的文件
#16         std::string dstfile("lined_" + srcfile);         //目标数据的文件
#17
#18         std::ifstream ifs(srcfile.c_str());              //定义文件输入流对象
#19         verify(ifs);
#20         std::ofstream ofs(dstfile.c_str());              //定义文件输出流对象
#21         verify(ofs);
#22
#23         int num = 0;                                      //行数计数
#24         std::string buffer;                              //存放每行内容
#25         while (std::getline(ifs, buffer)) {              //读取文件的一行
#26             ++num;                                       //增加行数
#27             buffer = int2str(num) + "\t" + buffer;        //格式化行号,并放置在行前
#28             ofs << buffer << std::endl;                 //向文件输出带行号的内容
#29             std::cout << buffer << std::endl;           //同时在屏幕上显示
#30         }
#31
#32         ofs.close();                                     //关闭文件流
#33         ifs.close();                                     //关闭文件流
#34     }
```

给文件编写行号的思路是，用一个行数计数器统计文件中的行数（如本例程序中的变量 `num`），然后用全局函数 `getline` 每次从文件流中读取一行内容，同时增加行号计数器，并把格式化后的行号插入该行内容的前面，然后输出到另外一个文件流中。当函数 `getline()` 提取到文件结束符 EOF 时，文件流对象 `ifs` 出错，`while` 循环对该对象的测试为假。

本例程序定义函数 `int2str()` 把整型的行数 `n` 进行格式化：转换成以 # 为前导、宽度为

【例 1.13.3】 学生类 Student: 使用流迭代器实现文件的读写操作。

```

#01 #include <iostream>
#02 #include <fstream> //for file stream
#03 #include <string> //for string
#04 #include <vector> //for vector
#05 #include <algorithm> //for copy
#06 #include <numeric> //for accumulate
#07 #include "verify.hpp" //for verify
#08
#09 class Student { //描述学生类型
#10 private:
#11     size_t id; //学号
#12     std::string name; //姓名
#13     int age; //年龄
#14     double score; //成绩
#15 public: //以下函数实现默认构造
#16     Student(size_t i = 0, const char* s = "", int a = 0, double d = 0) noexcept
#17         : id(i), name(s), age(a), score(d)
#18     {}
#19     //以下函数实现流插入和流提取运算
#20     friend std::ostream& operator << (std::ostream& os,
#21         const Student& rhs);
#22     friend std::istream& operator >> (std::istream& is,
#23         Student& rhs);
#24     //以下函数用作算法 accumulate 的参数
#25     friend double add_score(double s, const Student& rs);
#26 };
#27
#28 std::ostream& operator << (std::ostream& os,
#29     const Student& rhs) {
#30     os << rhs.id << std::endl; //输出学号
#31     os << rhs.name << std::endl; //输出姓名
#32     os << rhs.age << std::endl; //输出年龄
#33     os << rhs.score << std::endl; //输出成绩
#34     return os;
#35 }
#36
#37 std::istream& operator >> (std::istream& is,
#38     Student& rhs) {
#39     is >> rhs.id >> std::ws; //提取学号及其后的空白符
#40     std::getline(is, rhs.name); //提取姓名及其后的空白符
#41     is >> rhs.age >> std::ws; //提取年龄及其后的空白符
#42     is >> rhs.score >> std::ws; //提取成绩及其后的空白符
#43     return is;

```

```

#44     }
#45
#46     double add_score(double s, const Student& rs) {    //定义累加的方式
#47         return s + rs.score;                        //累加成绩
#48     }
#49
#50     int main() {
#51         std::vector<Student> vs;                    //定义容器存放对象
#52         vs.push_back(Student(2008001, "Tom", 18, 85));
#53         vs.push_back(Student(2008002, "Jerry", 22, 95));
#54         vs.push_back(Student(2008003, "Mike", 20, 88));
#55         vs.push_back(Student(2008004, "John", 19, 90));
#56
#57         const char* filename("Student.txt"); //文件名
#58         std::ofstream ofs(filename);         //定义文件输出流对象
#59         verify(ofs);                         //确保成功建立
#60         std::ostream_iterator<Student> fout(ofs, "\n"); //定义输出流迭代器对象
#61         std::copy(vs.begin(), vs.end(), fout); //向文件中写入对象
#62         ofs.close();                         //关闭文件
#63
#64         vs.clear();                          //清空容器
#65
#66         std::ifstream ifs(filename);          //定义文件输入流对象
#67         verify(ifs);                         //确保成功建立
#68         std::istream_iterator<Student> finbeg(ifs);
#69                                         //定义输入流迭代器对象作为起点
#70         std::istream_iterator<Student> finend;
#71                                         //定义输入流迭代器对象作为终点
#72         std::copy(finbeg, finend, std::back_inserter(vs));
#73                                         //从文件中读取对象
#74         ifs.close();                       //关闭文件
#75
#76         std::ostream_iterator<Student> sout(std::cout, "\n");
#77                                         //输出流迭代器对象
#78         std::copy(vs.begin(), vs.end(), sout); //向屏幕输入对象
#79                                         //以下累加总成绩
#80         double sum = std::accumulate(vs.begin(), vs.end(), 0.0, add_score);
#81         std::cout << "Average Score is: " << sum / vs.size() << ".\n";
#82     }

```

本程序以类 `Student` 为例示范了各种读写操作，主要过程为，首先向 `vector` 容器中插入几个 `Student` 对象，然后把对象写入文件中，接着从文件中读出这些对象，最后计算它们的平均成绩。

本程序示范了流输入/输出的一种简洁方式：应用算法 `copy()` 和输入/输出流迭代器。算法 `copy()` 是在不同流迭代器之间复制数据的通用函数，作为其参数的迭代器既可以是容器迭代器（如本例中的 `vector` 容器的迭代器），也可以是输入/输出流迭代器（如本例中的 `fout`、`finbeg`、`finend`、`sout`）。对于流迭代器而言，当与之关联的流对象是文件流对象时（如本例中的 `ofs`、`ifs`），则该流迭代器对象（如本例中的 `fout`、`finbeg`、`finend`）可

以读写文件流中的数据；当与之关联的流对象是标准流对象时（如 `std::cout`），则该流迭代器对象（如本例中的 `sout`）可以读写标准流中的数据（向屏幕输出或从键盘输入）。在定义输入/输出流迭代器时需要注意，输出流迭代器只有起点（如本例中的 `fout` 和 `sout`），没有终点；而输入流迭代器既有起点（如本例中的 `finbeg`），又有终点（如本例中的 `finend`）。输入流迭代器的终点要默认构造。本例没有示范标准输入流迭代器的用法，因为 `vector` 容器中的数据是从程序中设定的，可以在 `main()` 函数的开始用下列语句取代 `push_back()` 以便从键盘输入数据。

```
#01      std::istream_iterator<Student> sinbeg(std::cin); //输入流迭代器起点
#02      std::istream_iterator<Student> sinend;          //输入流迭代器终点
#03      std::copy(sinbeg, sinend, std::back_inserter(vs));
#04                                           //从键盘输入数据到 vs 中
```

不管输入/输出流迭代器是什么类型，都需要在其模板参数表中以 `Student` 类作为参数，因为它们读写的数据是 `Student` 类的对象。

为了应用算法 `copy()` 和输入/输出流迭代器实现对象的读写操作，需要在 `Student` 类中定义流插入运算符函数和流提取运算符函数（如本例中定义的 `operator<<` 和 `operator>>`），因为它们分别会被输出/输入流对象（如本例的 `ofs/ifs`）用于操作迭代器（如本例中的 `fout`、`finbeg`）所指的元素，其操作类似于 `ofs<< *fout` 和 `ifs>> *finbeg`。同时还要在类 `Student` 中定义默认构造函数（本例的构造函数具有默认构造的能力），该默认构造函数会在构造输入流迭代器对象时用到。

作为迭代器的一种，`back_inserter` 用于向空容器中插入元素，这是在应用算法 `copy()` 时尤其要注意的。当 `back_inserter` 在插入元素时可能会造成指向该 `vector` 容器的其他迭代器失效，而算法 `copy()` 通过赋值运算向容器中存放元素，如果目标迭代器没有足够多的空间，则算法 `copy()` 可能会引发“`vector` 迭代器不可去引用”的运行时错误。

最后还需要说明的是，定义友元函数 `add_score()` 的目的主要是用作算法 `accumulate()` 的参数，该算法累加区间 `[vs.begin(), vs.end())` 中的元素值，累加的初值由第三个参数指定（0.0 同时表示累加结果为 `double` 类型），第四个参数（即函数 `add_score()`）定义累加 `Student` 对象的方式（即累加数据成员 `score`）。之所以把函数 `add_score()` 定义为类 `Student` 的友元函数，是因为该函数需要访问类 `Student` 的 `private` 数据成员 `score`。

实验题目与提示

1. 参考例 1.13.1，以文本方式实现一个图形输出的控制台程序：对于基本的二维图形（直线、正反对角线、直角三角形、菱形、长方形、正方形、正三角形、圆等），按照用户指定的方式在控制台窗口生成该图形。该程序要实现基本的交互，图形的类型、大小和填充字符由用户从键盘输入。图形除了显示在屏幕上，还要保存到文本文件中以便粘贴到其他文件中。

【提示】本题要用到纯虚函数和多态性、文件流及格式化输出。定义抽象基类 `Shape`，在其中定义纯虚函数 `Draw()` 作为图形绘制的接口。

```
void Draw(std::ostream& os) const;
```

在绘制各二维图形时, 需要考虑控制台窗口输出的特点: 从上到下依次输出各行, 在每行上从左到右依次输出各列。还需要考虑每个图形的构成规律, 如空格和星号的个数变化规律。对于正三角形和圆需要计算整型化后的坐标, 然后输出它们的近似图案。

2. 设计一个简易的朋友通讯录管理程序。

(1) 定义朋友类型 `Friend`, 属性有姓名 (`name`)、性别 (`gender`)、年龄 (`age`)、手机 (`tel`)、QQ 号 (`qq`)、邮箱地址 (`E-mail`) 等。

(2) 实现从键盘输入朋友信息, 并选用合适的容器存放所有信息。假定每个朋友的手机号、QQ 号、邮箱地址都是唯一的, 而且朋友没有重名的。

(3) 实现朋友信息的维护。当某些朋友的通信方式有所改变时, 要及时更新。

(4) 实现所有信息文件的保存和载入。当程序退出运行时, 把所有数据保存到文件中。当程序启动时, 若发现同目录下有指定名称的数据文件, 则把该文件中的数据载入到内存中。

【提示】(1) 按照要求定义类 `Friend`, 并提供必要的构造函数。

(2) 根据不重复的假设, 可以选用容器 `set` 来存放 `Friend` 对象。在类中重载流提取运算符函数, 实现从键盘输入对象。

(3) 应用容器 `set` 的查找函数 (`find()`或其他), 找到需要修改的数据, 进行必要的修改。

(4) 在类中重载流插入运算符函数, 实现从文件输出对象。应用文件的输入、输出操作, 实现对象数据的保存和载入。

3. 设计一个自动生成代码的程序, 该程序根据给定的类名, 自动生成该类的头文件和源文件, 并在头文件中生成常用函数的声明, 在源文件中生成函数定义的框架。

【提示】通过键盘输入类名及其属性 (类型和名称)。由于类的构造函数、复制构造函数、赋值运算符函数、析构函数、访问函数、流输入/输出运算符函数及可能的运算符等都具有标准的形式, 因此可以通过文件输出流向头文件和源文件中逐个输出函数原型和函数定义框架。例如, 对于赋值运算符函数, 可以在头文件中输出下列函数原型:

```
MyClass& operator = (const MyClass& rhs);
```

在源文件中输出下列函数定义框架:

```
MyClass& MyClass::operator = (const MyClass& rhs) {
    //TODO: add codes here
    return *this;
}
```

4. 对 C/C++源文件和头文件进行一些简单的分析和处理, 然后报告结果。

(1) 检查文件中的花括号、圆括号和方括号及注释符 `/* */` 是否配对。

(2) 统计文件中常用关键字出现的频数。

(3) 统计文件中所定义的类型、结构、枚举等类型的个数。

(4) 清除 C/C++源文件和头文件中的所有注释。为了防止误删, 应先对该文件进行备份。

【提示】应用全局函数 `getline()` 依次读取文件中的一行, 文件内容保存为 `string` 对象。然后应用 `string` 类的查找、取子串等操作对每行内容进行分析。

(1) 可以应用容器 `stack` 来判断这些符号是否配对, 也可以用简单计数法来判断。

例如，设置计数器保存花括号的个数，遇到左花括号则把计数器加一，遇到右花括号则把计数器减一，最后判断该计数器的取值，即可知是否配对。

(2)、(3) 应用 `std::map<std::string, int>` 容器存放关键字及其频数。

(4) 单行注释符 `//` 可以逐行判断，而 C 注释符 `/* */` 可能嵌套而需要应用 `getline()` 函数读取多行内容。截取注释符之间的内容，不要把它们写入新的文件中。

第 2 部分 STL 算法与容器参考

第 1 章 STL 算法参考

1.1 头文件<type_traits>中的常用工具函数

STL 在头文件<type_traits>中提供了很多关于数据类型的分类、类型属性、类型关系等判断的工具，如表 2.1.1～表 2.1.3 所示。

表 2.1.1 基本类型分类

表达式	功能
is_void<T>::value, is_void_v<T>	是否是 void 类型
is_null_pointer<T>::value is_null_pointer_v<T>	是否是 nullptr 类型
is_integral<T>::value, is_integral_v<T>	是否是整型，如 bool、char、short、int、long
is_floating_point<T>::value is_floating_point_v<T>	是否是浮点类型，如 float、double、long double
is_array<T>::value, is_array_v<T>	是否是数组类型
is_enum<T>::value; is_enum_v<T>，	是否是枚举类型
is_union<T>::value; is_union_v<T>	是否是联合类型
is_class<T>::value; is_class_v<T>	是否是类类型
is_function<T>::value, is_function_v<T>	是否是函数类型
is_pointer<T>::value, is_pointer_v<T>	是否是指针类型
is_lvalue_reference<T>::value, is_lvalue_reference_v<T>	是否是左值引用类型
is_rvalue_reference<T>::value, is_rvalue_reference_v<T>	是否是右值引用类型
is_member_object_pointer<T>::value, is_member_object_pointer_v<T>	是否是 non-static 成员对象指针类型
is_member_function_pointer<T>::value, is_member_function_pointer_v<T>	是否是 non-static 成员函数指针类型

表 2.1.2 复合类型分类

表达式	功能
<code>is_fundamental<T>::value, is_fundamental_v<T></code>	是否是基本类型, 如算术类型、void、 <code>nullptr_t</code>
<code>is_arithmetic<T>::value, is_arithmetic_v<T></code>	是否是算术类型, 如整型和浮点型
<code>is_scalar<T>::value, is_scalar_v<T></code>	是否是标量类型, 如可能带有 <code>cv</code> 修饰符的算术类型、指针、指向成员的指针、枚举、 <code>nullptr_t</code>
<code>is_object<T>::value, is_object_v<T></code>	是否是对象类型, 如除函数、引用、void 类型外的可能带有 <code>cv</code> 修饰符的类型
<code>is_compound<T>::value, is_compound_v<T></code>	是否是复合类型, 如数组、函数、对象指针、函数指针、成员对象指针、成员函数指针、引用、类、联合、枚举, 以及任何带有 <code>cv</code> 修饰符的类型
<code>is_reference<T>::value, is_reference_v<T></code>	是否是引用类型, 如左值引用、右值引用
<code>is_member_pointer<T>::value</code> <code>is_member_pointer_v<T></code>	是否是指向 non-static 成员对象或成员函数的指针类型

表 2.1.3 类型的属性

表达式	功能
<code>is_const<T>::value, is_const_v<T></code>	是否是 <code>const</code> 修饰的类型
<code>is_volatile<T>::value, is_volatile_v<T></code>	是否是 <code>volatile</code> 修饰的类型
<code>is_trivial<T>::value, is_trivial_v<T></code>	是否是平凡的类型
<code>is_standard_layout<T>::value</code> <code>is_standard_layout_v<T></code>	是否是标准布局的类型
<code>is_pod<T>::value, is_pod_v<T></code>	是否是 pod 类型
<code>is_literal_type<T>::value</code> <code>is_literal_type_v<T></code>	是否是字面类型
<code>is_empty<T>::value, is_empty_v<T></code>	是否是空类型
<code>is_polymorphic<T>::value, is_polymorphic_v<T></code>	是否是多态类型
<code>is_abstract<T>::value, is_abstract_v<T></code>	是否是抽象类类型
<code>is_final<T>::value, is_final_v<T></code>	是否是 <code>final</code> 类类型
<code>is_aggregate<T>::value, is_aggregate_v<T></code>	是否是聚集类型
<code>is_signed<T>::value, is_signed_v<T></code>	是否有符号类型
<code>is_unsigned<T>::value, is_unsigned_v<T></code>	是否是无符号类型
<code>is_bounded_array<T>::value</code> <code>is_bounded_array_v<T></code>	是否有界数组
<code>is_unbounded_array<T>::value</code> <code>is_unbounded_array_v<T></code>	是否是未知边界的数组类型
<code>is_scoped_enum<T>::value, is_scoped_enum_v<T></code>	是否是强类型枚举类型

1.2 头文件<utility>中的常用辅助函数和工具

STL 在头文件<utility>中提供了一些有用的辅助函数和实用工具, 如 `pair<>`、

tuple、swap()、move()、forward()等，它们大多实现为函数模板，用法比较简单；包含了一些头文件，如<initializer_list>；定义了一些子命名空间，如rel_ops。

1. initializer_list

列表 initializer_list 在 C++11 等新标准中得到了广泛的应用。它会以列表形式创建一个临时数组。为了使用该类，可以单独包含头文件<initializer_list>。

在 3 种情况下会自动创建列表：①当构造函数接收列表形式的参数时，使用列表初始化对象；②当赋值运算符函数或一个普通函数接收列表形式的参数时，以列表作为右操作数进行赋值或作为参数调用该函数；③绑定到 auto 类型，包括在 ranged-for 循环中。它的一些常用操作有构造及赋值、成员访问等，如表 2.1.4 和表 2.1.5 所示。

表 2.1.4 构造及赋值

构造 initializer_list 对象的方式	功能
initializer_list<T> l{args...}	以参数 args...构造列表 l
auto l{args...}	以参数 args...构造列表 l

表 2.1.5 成员访问

表达式	功能
l.size()	获取列表中元素的个数
l.begin(), l.end()	获取列表中第一个元素的地址，最后一个元素地址的下一个地址
begin(l), end(l)	获取列表中第一个元素的地址，最后一个元素地址的下一个地址
rbegin(l), rend(l)	获取列表中逆向第一个元素的迭代器，逆向最后一个元素的迭代器的下一个地址
crbegin(l), crend(l)	获取列表中逆向第一个元素的 const 迭代器，逆向最后一个元素的 const 迭代器的下一个地址
empty(l)	判断列表是否为空
data(l)	获取指向列表底层数组的指针

【例 2.1.1】initializer_list 应用示例。

```
#01  #include <initializer_list>
#02  #include "xr.hpp"
#03
#04  int main() {
#05      std::initializer_list<int> l{ 1, 2, 3, 4, 5 };
#06      xr(l.size()); xr(empty(l)); xr(data(l));
#07
#08      for (auto i : l)
#09          std::cout << i << "\t";
#10      std::cout << std::endl;
#11
#12      for (auto iter{l.begin()}; iter != l.end(); ++iter)
#13          std::cout << *iter << "\t";
#14      std::cout << std::endl;
#15  }
```

程序的运行结果如下：

```
#06: [l.size()] ==>[5]
#06: [empty(l)] ==>[false]
#06: [data(l)] ==>[0x7ff606404040]
1      2      3      4      5
1      2      3      4      5
```

2. pair 和 make_pair()

如果需要把相互关联的两个数据封装成为一个整体处理，如从函数中返回两个值，则类模板 `pair` 是一个较好的选择。`pair` 对象可以由两个不同类型（heterogeneous objects）的数据对象构成，而且 `pair` 类中提供成员访问这两个数据的类型和数值。它的一些常用操作有构造及赋值、成员访问等，如表 2.1.6～表 2.1.9 所示。

表 2.1.6 构造 `pair` 对象

表达式	功能
<code>pair<T1, T2> p</code>	默认构造 <code>p</code> ，成员的值各自为 <code>T1</code> 、 <code>T2</code> 的默认值
<code>pair<T1, T2> p(x, y)</code>	封装 <code>x</code> 和 <code>y</code> 为 <code>p</code> （复制构造）
<code>pair<T1, T2> p(rv1, rv2)</code>	封装 <code>rv1</code> 和 <code>rv2</code> 为 <code>p</code> （转移构造）
<code>pair<T1, T2> p(p2)</code>	复制 <code>p2</code> 构造 <code>p</code> （复制构造）
<code>pair<T1, T2> p(rv)</code>	转移 <code>p2</code> 构造 <code>p</code> （转移构造）
<code>make_pair(x, y)</code>	封装 <code>x</code> 和 <code>y</code> 为一个 <code>pair</code> ，自动推断 <code>pair</code> 类型
<code>make_pair(rv1, rv2)</code>	封装 <code>rv1</code> 和 <code>rv2</code> 为一个 <code>pair</code> ，自动推断 <code>pair</code> 类型

表 2.1.7 为 `pair` 对象赋值

表达式	功能
<code>p = p2</code>	把 <code>p</code> 赋值为 <code>p2</code> （复制赋值），允许隐式类型转换
<code>p = rv</code>	把 <code>p</code> 赋值为 <code>rv</code> （转移赋值），允许隐式类型转换

表 2.1.8 交换 `pair` 对象

表达式	功能
<code>p.swap(p2)</code>	交换 <code>p</code> 和 <code>p2</code> 的成员取值
<code>swap(p, p2)</code>	交换 <code>p</code> 和 <code>p2</code> 的成员取值

表 2.1.9 成员访问

表达式	功能
<code>first, second</code>	获取成员 <code>first</code> 、 <code>second</code> 的取值
<code>first_type, second_type</code>	获取成员 <code>first</code> 、 <code>second</code> 的数据类型
<code>get<0>(p), get<1>(p)</code>	以 tuple-like 的方法获取成员 <code>first</code> 、 <code>second</code> 的取值，编译期常量索引只能取值为 0 或 1
<code>get<T>(p), get<U>(p)</code>	以 tuple-like 的方法获取成员 <code>first</code> 、 <code>second</code> 的取值，两者的数据类型一定要不同

【例 2.1.2】`pair` / `make_pair` 应用示例。

```
#01    #include <utility>
#02    #include "xr.hpp"
```

```

#03
#04     auto div_with_rem(double a, double b) {      //设返回类型为 auto
#05         int quotient = int(a / b);              //整商
#06         double remainder = a - quotient * b;     //余数
#07
#08         return std::make_pair(quotient, remainder); //返回 pair
#09     }
#10     std::pair<int, double> div_with_rem2(double a, double b) {
#11         int quotient = int(a / b);              //整商
#12         double remainder = a - quotient * b;     //余数
#13
#14         return { quotient, remainder };          //以列表形式返回值
#15     }
#16
#17     int main() {
#18         double a, b;
#19         std::pair<int, double> p;
#20
#21         b = 0.78, a = 4 * b + 0.6;
#22         p = div_with_rem(a, b);
#23         xr(p.first); xr(p.second);                //分别为 4, 0.6
#24
#25         b = 0.8, a = 3 * b + 1.8;
#26         p = div_with_rem2(a, b);
#27         xr(std::get<int>(p)); xr(std::get<double>(p)); //分别为 5, 0.2
#28     }

```

上述程序中，函数 `div_with_rem()`/`div_with_rem2()` 需要同时返回除法的整商 `quotient`（类型为 `int`）和余数 `remainder`（类型为 `double`），这两个函数分别演示了对这两个数据进行封装的两种方式。函数 `div_with_rem()` 以工具函数 `make_pair` 封装，它能够自动推断类型，因此返回类型设为 `auto`，函数 `div_with_rem2()` 以列表形式返回 `pair`，因此显式设置返回类型为 `pair<int, double>`。

3. `swap()`

交换是一个非常重要的操作，所有数据类型都应支持该操作。`swap()` 函数详细介绍见表 2.1.10。

表 2.1.10 `swap()` 函数

算法名称	<code>swap</code>
函数原型 1	<pre>template<class Type> void swap(Type& a, Type& b);</pre>
函数功能 1	交换两个数据
函数原型 2	<pre>template< class T2, size_t N > void swap(T2(&a)[N], T2(&b)[N]);</pre>
函数功能 2	交换两个长度相同的数组
函数参数	<code>a</code> 表示左操作数； <code>b</code> 表示右操作数
返回值	无

【例 2.1.3】swap()函数的应用示例。

```
#01      #include <utility>
#02      #include "xr.hpp"
#03
#04      void sort3(int& a, int& b, int& c) {
#05          if (a > b) std::swap(a, b);
#06          if (a > c) std::swap(a, c);
#07          if (b > c) std::swap(b, c);
#08      }
#09      int main() {
#10          int x = 3, y = 6, z = 5;
#11          sort3(x, y, z);
#12          xr(x); xr(y); xr(z);                //结果分别为 3 5 6
#13
#14          int a[]{ 1, 2, 3 }, b[]{ 4, 5, 6 };
#15          std::swap(a, b);
#16          for (auto i : a)
#17              std::cout << i << "\t";
#18          std::cout << std::endl;
#19      }
```

4. exchange()

exchange()函数（表 2.1.11）常用于实现转移赋值运算符函数和转移构造函数。

表 2.1.11 exchange()函数

算法名称	exchange
函数原型	template<class T, class U = T> T exchange(T& obj, U&& new_value);
函数功能	使用 new_value 替换 obj 的值，然后返回 obj 的旧值
函数参数	obj 表示代替换值的对象；new_value 表示要赋给 obj 的值
返回值	obj 的旧值

【例 2.1.4】exchange()函数的应用示例。

```
#01      #include <iostream>
#02      #include <utility>
#03      #include <vector>
#04      #include <iterator>
#05
#06      void f() { std::cout << "msg from f()"; }
#07      int main() {
#08          std::vector<int> v;
#09          // 由于第二个模板参数为默认值，因此可以用列表作为第二个参数
#10          // 如下等价于：std::exchange(v, std::vector<int>{1,2,3,4,5});
#11          std::exchange(v, { 1, 2, 3, 4, 5 });
#12          std::copy(begin(v), end(v),
#13                  std::ostream_iterator<int>(std::cout, ", "));
#14          std::cout << "\n";                //以上输出 1, 2, 3, 4, 5,
#15      }
```

```
#16      void(*g)();
#17      // 由于第二个模板参数为默认值, 因此可以用常规函数作为第二个参数
#18      // 如下等价于: std::exchange(fun, static_cast<void(*)>()(f))
#19      std::exchange(g, f);
#20      g();                                //输出 msg from f()
#21  }
```

1.3 头文件<functional>中的辅助函数和工具

STL 头文件<functional>提供了一些常用的函数对象库和标准的 hash 函数, 如表 2.1.12~表 2.1.16 所示。

表 2.1.12 bind()函数

算法名称	bind
函数原型	template< class F, class... Args > bind(F&& f, Args&&... args);
函数功能	生成函数 f 的调用转发, 一些参数绑定到 args
函数参数	f 表示 callable 对象; args 表示需要绑定的参数列表
返回值	函数对象

表 2.1.13 bind_front()函数

算法名称	bind_front
函数原型	template <class F, class... Args> constexpr bind_front(F&& f, Args&&... args);
函数功能	生成函数 f 的调用转发, 将前面几个参数绑定到 args
函数参数	f 表示 callable 对象; args 表示需要绑定的参数列表
返回值	函数对象

表 2.1.14 invoke()函数

算法名称	invoke
函数原型	template< class F, class... Args> constexpr std::invoke_result_t<F, Args...> invoke(F&& f, Args&&... args) noexcept;
函数功能	用参数 args 调用可调用对象 f
函数参数	f 表示 callable 对象; args 表示需要绑定的参数列表
返回值	函数对象

表 2.1.15 mem_fn()函数

算法名称	mem_fn
函数原型	template< class M, class T > mem_fn(M T::* pm) noexcept;
函数功能	生成指向成员的函数指针的函数封装对象
函数参数	pm 表示需要封装的成员函数的指针
返回值	函数对象

表 2.1.16 not_fn()函数

算法名称	not_fn
函数原型	template< class F> not_fn(F&& f);
函数功能	生成函数的调用转发, 返回值是可调用对象 f 的逻辑否! 运算后的结果
函数参数	f 表示 callable 对象
返回值	无

【例 2.1.5】bind()函数的应用示例。

```
#01      #include <iostream>
#02      #include <functional>
#03
#04      void f(int n1, int n2, int& n3) {
#05          n3 = n1 - n2;
#06      }
#07
#08      int main() {
#09          using namespace std::placeholders; // for _1, _2, _3...
#10          int x = 10, y = 30, z = 5;
#11          auto f1 = std::bind(f, _2, _3, _1);
#12          f1(x, y, z);                      // f(y, z, x), x = y-z
#13          std::cout << x << " " << y << " " << z << std::endl; // 25 30 5
#14      }
```

【例 2.1.6】bind_front()函数的应用示例。

```
#01      #include <functional>
#02      #include "xr.hpp"
#03
#04      int main() {
#05          int x = 10;
#06          auto minus = [](int x, int y){return x-y;};
#07          auto f_minus = std::bind_front(minus, 100); // minus(100, x)
#08          xr(f_minus(x));                          // 90
#09      }
```

【例 2.1.7】invoke()函数的应用示例。

```
#01      #include <functional>
#02      #include <iostream>
#03
#04      void print_x(int x) {
#05          std::cout << x << '\n';
#06      }
#07
#08      struct print {
#09          void operator()(int x) const {
#10              std::cout << x << '\n';
#11          }
#12      };
#13
#14      int main() {
```

```

#15      // invoke a global function
#16      std::invoke(print_x, 1);                      //1
#17
#18      // invoke a lambda
#19      std::invoke([]() { print_x(2); });            //2
#20
#21      // invoke a function object
#22      std::invoke(print(), 3);                      //3
#23      }

```

【例 2.1.8】mem_fn()函数的用法示例。

```

#01      #include <functional>
#02      #include <iostream>
#03
#04      struct MyStruct {
#05          int data = 7;
#06          void sayHello() {
#07              std::cout << "Hello, world.\n";
#08          }
#09          void print(int x) {
#10              std::cout << "x: " << x << '\n';
#11          }
#12      };
#13
#14      int main() {
#15          MyStruct ms;
#16
#17          auto f1 = std::mem_fn(&MyStruct::sayHello);
#18          f1(ms);                                     //Hello, world.
#19
#20          auto f2 = std::mem_fn(&MyStruct::print);
#21          f2(ms, 2);                                  //x: 2
#22
#23          auto f3 = std::mem_fn(&MyStruct::data);
#24          std::cout << "data: " << f3(ms) << '\n';    //data: 7
#25      }

```

【例 2.1.9】not_fn()函数的用法示例。

```

#01      #include <functional>
#02      #include <iostream>
#03      #include <algorithm>
#04
#05      int main() {
#06          int a[] {1, 2, 3, 4, 5};
#07          auto n = sizeof(a) / sizeof(*a);
#08
#09          auto f = [](int x) {return x % 2 == 0; };
#10          auto m = std::count_if(a, a + n, std::not_fn(f));
#11          std::cout << m << std::endl;                //输出奇数的个数 3
#12      }

```


1.4 STL 常用算法

1.4.1 不变序列算法

常用不变序列算法的应用形式及功能如表 2.1.17 所示,这类算法不改变区间中元素的值和次序。

表 2.1.17 不变序列算法的应用形式及功能

算法应用的形式	算法的功能
<code>all_of(first, last, p)</code>	区间 <code>[first, last)</code> 中是否每个元素都对一元谓词 <code>p</code> 返回为 <code>true</code>
<code>any_of(first, last, p)</code>	区间 <code>[first, last)</code> 中是否至少有一个元素对一元谓词 <code>p</code> 返回为 <code>true</code>
<code>none_of(first, last, p)</code>	区间 <code>[first, last)</code> 中是否没有元素对一元谓词 <code>p</code> 返回为 <code>true</code>
<code>for_each(first, last, func)</code>	对区间 <code>[first, last)</code> 中的每个元素 <code>e</code> 执行不可改变操作 <code>func(e)</code>
<code>for_each_n(first, N, func)</code>	对区间 <code>[first, first+N)</code> 中的每个元素 <code>e</code> 执行不可改变操作 <code>func(e)</code>
<code>count(first, last, val)</code>	计算区间 <code>[first, last)</code> 中值等于 <code>val</code> 的元素个数
<code>count_if(first, last, pr)</code>	计算区间 <code>[first, last)</code> 中使表达式 <code>pr(e)</code> 值为 <code>true</code> 的元素个数
<code>min_element(first, last[, comp])</code>	返回区间 <code>[first, last)</code> 中值最小元素的首次出现位置, <code>comp</code> 为比较准则
<code>max_element(first, last[, comp])</code>	返回区间 <code>[first, last)</code> 中值最大元素的首次出现位置, <code>comp</code> 为比较准则
<code>minmax_element(first, last[, comp])</code>	同时返回区间 <code>[first, last)</code> 中值最小、值最大元素首次出现的位置, <code>comp</code> 为比较准则, 返回类型为 <code>pair</code>
<code>find(first, last, val)</code>	查找区间 <code>[first, last)</code> 中值为 <code>val</code> 元素的首次出现位置
<code>find_if(first, last, pr)</code>	计算区间 <code>[first, last)</code> 中使表达式 <code>pr(e)</code> 为 <code>true</code> 元素的首次出现位置
<code>find_if_not(first, last, pr)</code>	计算区间 <code>[first, last)</code> 中使表达式 <code>pr(e)</code> 为 <code>false</code> 元素的首次出现位置
<code>search_n(first, last, count, val [, comp])</code>	计算区间 <code>[first, last)</code> 中“连续 <code>count</code> 个值为 <code>val</code> 的元素集合”的首次出现位置, <code>comp</code> 为比较准则
<code>search(first1, last1, first2, last2[, comp])</code>	计算源区间 <code>[first1, last1)</code> 与目标区间 <code>[first2, last2)</code> 元素第一次完全相同的位置, <code>comp</code> 为比较准则
<code>mismatch(first1, last1, first2[, last2, comp])</code>	计算区间 <code>[first1, last1)</code> 和以 <code>first2</code> 为起点的区间 (或 <code>[first2, last2)</code>) 中第一次失配元素的位置, 返回值为 <code>pair</code> , <code>comp</code> 为比较准则
<code>find_end(first1, last1, first2, last2[, comp])</code>	计算源区间 <code>[first1, last1)</code> 与目标区间 <code>[first2, last2)</code> 元素最后一次完全相同的位置, <code>comp</code> 为比较准则
<code>find_first_of(first1, last1, first2, last2[, comp])</code>	查找目标区间 <code>[first2, last2)</code> 中任一元素在源区间 <code>[first1, last1)</code> 中第一次出现的位置, <code>comp</code> 为比较准则
<code>adjacent_find(first, last[, comp])</code>	查找区间 <code>[first, last)</code> 中第一对相邻且相等的元素, <code>comp</code> 为比较准则
<code>equal(first1, last1, first2[, comp])</code>	判断区间 <code>[first1, last1)</code> 与以 <code>first2</code> 为起点的区间元素是否相同, <code>comp</code> 为比较准则
<code>lexicographical_compare(first1, last1, first2, last2[, comp])</code>	判断区间 <code>[first1, last1)</code> 元素是否以字典顺序小于区间 <code>[first2, last2)</code> 元素, <code>comp</code> 为比较准则

注: 上述算法中带有`[,comp]`者,表示该算法实际有两种形式:一种无须参数 `comp`,另一种则需要参数 `comp`。后同。
`e` 为区间元素。

1. all_of / any_of / none_of

all_of / any_of / none_of 三种算法的介绍详见表 2.1.18～表 2.1.20。

表 2.1.18 all_of 算法

算法名称	all_of
函数原型	template<class InputIterator, class UnaryPredicate> bool all_of(InputIterator first, InputIterator last, UnaryPredicate p);
函数功能	区间[first, last)中是否每个元素都对一元谓词 p 返回为 true

表 2.1.19 any_of 算法

算法名称	any_of
函数原型	template<class InputIterator, class UnaryPredicate> bool any_of(InputIterator first, InputIterator last, UnaryPredicate p);
函数功能	区间[first, last)中是否至少有一个元素对一元谓词 p 返回为 true

表 2.1.20 none_of 算法

算法名称	none_of
函数原型	template<class InputIterator, class UnaryPredicate> bool none_of(InputIterator first, InputIterator last, UnaryPredicate p);
函数功能	区间[first, last)中是否没有元素对一元谓词 p 返回为 true
函数参数	first 表示区间起点; last 表示区间终点; p 表示一元函数对象
返回值	bool

【例 2.1.10】all_of / any_of / none_of 算法的应用示例。

```
#01      #include <vector>
#02      #include <numeric>
#03      #include <algorithm>
#04      #include <iterator>
#05      #include <iostream>
#06
#07      int main() {
#08          std::vector<int> v(10, 2);
#09          std::partial_sum(v.cbegin(), v.cend(), v.begin());
#10          std::cout << "Among the numbers: ";
#11          std::copy(v.cbegin(), v.cend(),
#12                  std::ostream_iterator<int>(std::cout, " "));
#13          std::cout << '\n';
#14
#15          int d = 2;
#16          if (std::all_of(v.cbegin(), v.cend(),
#17                          [d](int x) { return x % d == 0; })) {
#18              std::cout << "All numbers are even\n";
#19          }
#20
#21          d = 3;
```

```
#22         if (std::none_of(v.cbegin(),v.cend(),
#23                               [d](int x) { return x % d == 0; })) {
#24             std::cout << "None of them are divisible by 3\n";
#25         }
#26
#27         d = 5;
#28         if (std::any_of(v.cbegin(),v.cend(),
#29                               [d](int x) {return x % d == 0; })) {
#30             std::cout << "At least one number is divisible by 5\n";
#31         }
#32     }
```

程序的输出结果如下：

```
Among the numbers: 2 4 6 8 10 12 14 16 18 20
All numbers are even
At least one number is divisible by 5
```

2. for_each / for_each_n

for_each / for_each_n 算法介绍详见表 2.1.21 和表 2.1.22。

表 2.1.21 for_each 算法

算法名称	for_each
函数原型	template<class InputIterator, class Function> Function for_each(InputIterator first, InputIterator last, Function func);
函数功能	对区间[first, last)中的每个元素 e 执行不可改变操作 func(e)
函数参数	func 表示一元函数对象；first 表示区间起点；last 表示区间终点
返回值	函数对象 func

表 2.1.22 for_each_n 算法

算法名称	for_each_n
函数原型	template<class InputIterator, class Size, class UnaryFunction> InputIterator for_each_n(InputIterator first, Size n, UnaryFunction func);
函数功能	对区间[first, first+n)中的每个元素 e 执行不可改变操作 func(e)
函数参数	func 表示一元函数对象；first 表示区间起点；n 表示区间元素个数
返回值	first + n

运用这两个算法的关键在于函数对象 func 的定义，该函数对象 func 应该定义为一元函数，至于该函数有否返回值并不重要。

【例 2.1.11】for_each / for_each_n 算法的应用示例。

```
#01     #include <iostream>
#02     #include <algorithm>
#03
#04     void print(int m) { std::cout << m << "\t"; }
#05     void mulBy10(int& m) { m *= 10; }
#06
#07     int main() {
```

```
#08      int a[]{ 3, 4, 5, 6, 7 };
#09      auto n = sizeof(a) / sizeof(*a);
#10
#11      std::for_each(a, a + n, print);           //输出 3 4 5 6 7
#12      std::cout << std::endl;
#13
#14      std::for_each(a, a + n, mulBy10);
#15
#16      std::for_each_n(a, n, print);             //输出 30 40 50 60 70
#17      std::cout << std::endl;
#18      }
```

从例 2.1.11 可以看出，若需要定义函数通过算法 `for_each()` 改变区间中的元素，则应以引用传递该函数的参数，如函数 `mulBy10()` 所示；否则，应该以值或 `const` 引用传递函数参数，如函数 `print()` 所示。

3. count / count_if

`count / count_if` 算法介绍详见表 2.1.23 和表 2.1.24。

表 2.1.23 count 算法

算法名称	count
函数原型	template<class InputIterator, class Type> typename iterator_traits<InputIterator>::difference_type count(InputIterator first, InputIterator last, const Type& val);
函数功能	计算区间[first, last)中值等于 val 的元素个数
函数参数	val 表示元素值；first 表示区间起点；last 表示区间终点
返回值	元素个数（类型为 iterator_traits<InputIterator>::difference_type）

表 2.1.24 count_if 算法

算法名称	count_if
函数原型	template<class InputIterator, class Predicate> typename iterator_traits<InputIterator>::difference_type count_if(InputIterator first, InputIterator last, Predicate pr);
函数功能	计算区间[first, last)中使表达式 pr(e)值为 true 的元素个数
函数参数	pr 表示一元谓词；first 表示区间起点；last 表示区间终点
返回值	元素个数（类型为 iterator_traits<InputIterator>::difference_type）

运用算法 `count_if` 的关键在于一元谓词 `pr` 的定义，所谓一元谓词是指只带一个参数、返回类型为 `bool` 型的函数。

【例 2.1.12】`count / count_if` 算法的应用示例。

```
#01      #include <iostream>
#02      #include <algorithm>
#03      #include "print.hpp"
#04
#05      bool iseven(int m) {return m % 2 == 0;}
#06      bool greaterthan3(int m) {return m > 3;}
```

```

#07
#08     int main()
#09     {
#10         int a[] = {1, 4, 3, 5, 3, 6, 3, 8, 3};
#11         size_t n = sizeof(a) / sizeof(*a);
#12         print(a, a + n, "a: ");
#13
#14         size_t m;
#15         m = std::count(a, a + n, 3);
#16         std::cout << "3 presents " << m << " times.\n";
#17
#18         m = std::count_if(a, a + n, iseven);
#19         std::cout << m << " evens in array.\n";
#20
#21         m = std::count_if(a, a + n, greaterthan3);
#22         std::cout << m << " numbers are greater than 3.\n";
#23     }

```

上述程序中，表达式 `count(a, a + n, 3)` 统计值为 3 的元素个数；表达式 `count_if(a, a + n, iseven)` 统计偶数元素的个数；表达式 `count_if(a, a + n, greaterthan3)` 统计值大于 3 的元素个数。

程序的输出结果如下：

```

a: 1   4   3   5   3   6   3   8   3
3 presents 4 times.
3 evens in array.
4 numbers are greater than 3.

```

4. max_element / min_element / minmax_element

`max_element` / `min_element` / `minmax_element` 这三个算法介绍详见表 2.1.25～表 2.1.27。

表 2.1.25 max_element 算法

算法名称	max_element
函数原型 1	template<class ForwardIterator> ForwardIterator max_element(ForwardIterator first, ForwardIterator last);
函数功能 1	计算区间[first, last)中值最大元素的首次出现位置，以 operator< 作为比较准则
函数原型 2	template<class ForwardIterator, class BinaryPredicate> ForwardIterator max_element(ForwardIterator first, ForwardIterator last, BinaryPredicate comp);
函数功能 2	计算区间[first, last)中值最大元素的首次出现位置，以二元谓词 comp 作为比较准则
函数参数	first 表示区间起点；last 表示区间终点；comp 表示比较准则
返回值	值最大元素的首次出现位置

表 2.1.26 min_element 算法

算法名称	min_element
函数原型 1	template<class ForwardIterator> ForwardIterator min_element(ForwardIterator first, ForwardIterator last);
函数功能 1	计算区间[first, last)中值最小元素的首次出现位置，以 operator<作为比较准则

续表

函数原型 2	template<class ForwardIterator, class BinaryPredicate> ForwardIterator min_element(ForwardIterator first, ForwardIterator last, BinaryPredicate comp);
函数功能 2	计算区间[first, last)中值最小元素的首次出现位置，以二元谓词 comp 作为比较准则
函数参数	first 表示区间起点；last 表示区间终点；comp 表示比较准则
返回值	值最小元素的首次出现位置

表 2.1.27 minmax_element 算法

算法名称	minmax_element
函数原型 1	template<class ForwardIterator> pair<ForwardIterator, ForwardIterator> minmax_element(ForwardIterator first, ForwardIterator last);
函数功能 1	同时计算区间[first, last)中值最小元素、值最大元素的首次出现位置，以 operator<作为比较准则
函数原型 2	template<class ForwardIterator, class BinaryPredicate> pair<ForwardIterator, ForwardIterator> minmax_element(ForwardIterator first, ForwardIterator last, BinaryPredicate comp);
函数功能 2	计算区间[first, last)中值最小元素的首次出现位置，以二元谓词 comp 作为比较准则
函数参数	first 表示区间起点；last 表示区间终点；comp 表示比较准则
返回值	值最小元素的首次出现位置

运用带有函数对象版本的算法 max_element/min_element/minmax_element 时，需要定义二元谓词函数或函数对象，这些函数类型带有两个参数且返回类型为 bool。

【例 2.1.13】max_element / min_element/minmax_element 算法的应用示例。

```
#01     #include <iostream>
#02     #include <algorithm>
#03     #include "print.hpp"
#04
#05     bool compByAbs(int m, int n) { return m * m < n * n; }
#06
#07     int main() {
#08         int a[] = { 1, 4, -5, 2, -6, 3, -8, 7 };
#09         size_t n = sizeof(a) / sizeof(*a);
#10         print(a, a + n, "a: ");
#11
#12         auto p = std::max_element(a, a + n);
#13         std::cout << *p << " is the max element." << std::endl;
#14
#15         p = std::max_element(a, a + n, compByAbs);
#16         std::cout << *p << " is the max square element." << std::endl;
#17
#18         p = std::min_element(a, a + n);
#19         std::cout << *p << " is the min element." << std::endl;
#20
#21         p = std::min_element(a, a + n, compByAbs);
#22         std::cout << *p << " is the min square element." << std::endl;
#23
```

```

#24      auto [pmin, pmax] = std::minmax_element(a, a + n);
#25      std::cout << *pmin << " is the min element." << std::endl;
#26      std::cout << *pmax << " is the max element." << std::endl;
#27
#28      auto [pmin2, pmax2] = std::minmax_element(a, a + n, compByAbs);
#29      std::cout << *pmin2 << " is the min square element." << std::endl;
#30      std::cout << *pmax2 << " is the max square element." << std::endl;
#31      }

```

程序的输出结果如下:

```

a: 1    4    -5    2    -6    3    -8    7
7 is the max element.
-8 is the max square element.
-8 is the min element.
1 is the min square element.
-8 is the min element.
7 is the max element.
1 is the min square element.
-8 is the max square element.

```

5. find / find_if / find_if_not

find / find_if / find_if_not 这三个算法介绍详见表 2.1.28~表 2.1.30。

表 2.1.28 find 算法

算法名称	find
函数原型	template<class InputIterator, class Type> InputIterator find(InputIterator first, InputIterator last, const Type& val);
函数功能	查找区间[first, last)中值为 val 元素的首次出现位置
函数参数	val 表示元素值; first 表示区间起点; last 表示区间终点
返回值	元素首次出现的位置。若查找失败, 则返回值为第二个参数

表 2.1.29 find_if 算法

算法名称	find_if
函数原型	template<class InputIterator, class Predicate> InputIterator find_if(InputIterator first, InputIterator last, Predicate pr);
函数功能	计算区间[first, last)中使表达式 pr(e)为 true 元素的首次出现位置
函数参数	pr 表示一元函数对象; first 表示区间起点; last 表示区间终点
返回值	元素首次出现的位置。若查找失败, 则返回值为第二个参数

表 2.1.30 find_if_not 算法

算法名称	find_if_not
函数原型	template<class InputIterator, class Predicate> InputIterator find_if_not(InputIterator first, InputIterator last, Predicate pr);
函数功能	计算区间[first, last)中使表达式 pr(e)为 false 元素的首次出现位置
函数参数	pr 表示一元函数对象; first 表示区间起点; last 表示区间终点
返回值	元素首次出现的位置。若查找失败, 则返回值为第二个参数

运用算法 `find_if`/`find_if_not` 时需要定义一元谓词函数或函数对象，这些函数类型带有一个参数且返回类型为 `bool`。

【例 2.1.14】`find`/`find_if`/`find_if_not` 算法的应用示例。

```
#01     #include <iostream>
#02     #include <algorithm>
#03     #include "print.hpp"
#04
#05     bool iseven(int m) { return m % 2 == 0; }
#06
#07     int main() {
#08         int a[] { 1, 4, 5, 2, 6, 5, 8, 7 };
#09         auto n = sizeof(a) / sizeof(*a);
#10         print(a, a + n);
#11
#12         auto p = std::find(a, a + n, 5);
#13         if (p != a + n)
#14             std::cout << "first 5 is at " << p - a << std::endl;
#15
#16         p = std::find_if(a, a + n, iseven);
#17         if (p != a + n)
#18             std::cout << "first even is at " << p - a << std::endl;
#19
#20         p = std::find_if_not(a, a + n, iseven);
#21         if (p != a + n)
#22             std::cout << "first odd is at " << p - a << std::endl;
#23     }
```

程序的输出结果如下：

```
1      4      5      2      6      5      8      7
first 5 is at 2
first even is at 1
first odd is at 0
```

6. `search_n`/`search`

`search_n`/`search` 算法介绍详见表 2.1.31 和表 2.1.32。

表 2.1.31 `search_n` 算法

算法名称	<code>search_n</code>
函数原型 1	<code>template<class FwdIterator, class Size, class Type></code> <code>FwdIterator search_n(FwdIterator first, FwdIterator last, Size count, const Type& val);</code>
函数功能 1	计算区间 <code>[first, last)</code> 中“连续 <code>count</code> 个值为 <code>val</code> 的元素集合”的首次出现位置
函数原型 2	<code>template<class FwdIterator, class Size, class Type, class BinaryPredicate></code> <code>FwdIterator search_n(FwdIterator first, FwdIterator last, Size count, const Type& val, BinaryPredicate comp);</code>
函数功能 2	计算区间 <code>[first, last)</code> 中“连续 <code>count</code> 个元素使 <code>comp(e, val)</code> 为 <code>true</code> 的元素集合”的首次出现位置
函数参数	<code>count</code> 表示元素个数； <code>val</code> 表示元素值； <code>first</code> 表示区间起点； <code>last</code> 表示区间终点； <code>comp</code> 表示比较准则
返回值	元素首次出现的位置

表 2.1.32 search 算法

算法名称	search
函数原型 1	template<class FwdIterator1, class FwdIterator2> FwdIterator1 search(FwdIterator1 first1, FwdIterator1 last1, FwdIterator2 first2, FwdIterator2 last2);
函数功能 1	计算源区间[first1, last1)与目标区间[first2, last2)元素第一次完全相同的位置
函数原型 2	template<class FwdIterator1, class FwdIterator2, class BinaryPredicate> FwdIterator1 search(FwdIterator1 first1, FwdIterator1 last1, FwdIterator2 first2, FwdIterator2 last2, BinaryPredicate comp);
函数功能 2	与上述版本类似, 只是比较准则为“使表达式 comp(e1, e2)值为 true”
函数参数	first1 表示源区间起点; last1 表示源区间终点; first2 表示目标区间起点; last2 表示目标区间终点; comp 表示比较准则
返回值	目标区间在源区间中首次出现的位置

运用带有函数对象版本的算法 search_n/search 时, 关键在于函数对象的定义, 要求该函数类型带有两个参数, 返回类型为 bool。

【例 2.1.15】search_n / search 算法的应用示例。

```

#01     #include <iostream>
#02     #include <algorithm>
#03
#04     bool mIsNegativeOfn(int m, int n) { return m + n == 0;}
#05     bool mIsNotMultipleOfn(int m, int n) { return m % n != 0; }
#06
#07     int main()
#08     {
#09         int a[] = {1, 2, 4, 5, 5, 6, 8, -1, -2, -4, 5, 5, 7};
#10         size_t n = sizeof(a) / sizeof(*a);
#11
#12         int* p;
#13         p = std::search_n(a, a + n, 2, 5);
#14         if (p != a + n)
#15             std::cout << "2 consecutive 5 start from " << p - a << std::endl;
#16
#17         p = std::search_n(a, a + n, 3, 2, mIsNotMultipleOfn);
#18         if (p != a + n)
#19             std::cout << "3 consecutive oddes start from " << p - a << std::endl;
#20
#21         p = std::search(a, a + n, a + 4, a + 7);
#22         if (p != a + n)
#23             std::cout << "{5, 6, 8} in array start from " << p - a << std::endl;
#24
#25         p = std::search(a, a + n, a, a + 3, mIsNegativeOfn);
#26         if (p != a + n)
#27             std::cout << "negative of {1, 2, 4} start from " << p - a << std::endl;
#28     }

```

上述程序中, 表达式 search_n(a, a + n, 2, 5) 计算连续 2 个 5 首次出现的位置; 表达式 search_n(a, a + n, 3, 2, mIsNotMultipleOfn) 计算连续 3 个与数值 2 满足函数关系

mIsNotMultipleOfn 的元素首次出现的位置；表达式 search(a, a + n, a + 4, a + 7)计算子数组 {5, 6, 8} 首次出现的位置；表达式 search(a, a + n, a, a + 3, mIsNegativeOfn)计算与子数组 {1, 2, 4} 元素满足函数关系 mIsNegativeOfn 的元素首次出现的位置。

程序的输出结果如下：

```
2 consecutive 5 start from 3
3 consecutive oddes start from 10
{5, 6, 8} in array start from 4
negative of {1, 2, 4} start from 7
```

7. mismatch

mismatch 算法介绍详见表 2.1.33。

表 2.1.33 mismatch 算法

算法名称	mismatch
函数原型 1	template<class InputIterator1, class InputIterator2> pair<InputIterator1, InputIterator2> mismatch(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2);
函数功能 1	计算源区间[first1, last1)与目标区间[first2, last2)元素最后一次完全相同的位置
函数原型 2	template<class InputIterator1, class InputIterator2, class BinaryPredicate > pair<InputIterator1, InputIterator2> mismatch(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, BinaryPredicate comp);
函数功能 2	与上述版本类似，只是比较准则为“使表达式 comp(e1, e2)值为 true”
函数参数	first1 表示源区间起点；last1 表示源区间终点；first2 表示目标区间起点；last2 表示目标区间终点；comp 表示比较准则
返回值	第一个不相等两元素的位置

运用算法 mismatch 的难点在于理解返回值 pair。运用带有函数对象版本的算法 mismatch 时，关键在于函数对象的定义，要求该函数类型带有两个参数，返回类型为 bool。

【例 2.1.16】mismatch 算法的应用示例。

```
#01      #include <iostream>
#02      #include <string_view>
#03      #include <algorithm>
#04
#05      std::string mirror_ends(std::string_view in) {
#06          return std::string(in.begin(),
#07              std::mismatch(in.begin(), in.end(), in.rbegin()).first);
#08      }
#09
#10      int main() {
#11          std::cout << mirror_ends("abXYZba") << '\n'      //ab
#12              << mirror_ends("abca") << '\n'                //a
#13              << mirror_ends("aba") << '\n';                //aba
#14      }
```

8. find_end

find_end 算法介绍详见表 2.1.34。

表 2.1.34 find_end 算法

算法名称	find_end
函数原型 1	template<class FwdIterator1, class FwdIterator2> FwdIterator1 find_end(FwdIterator1 first1, FwdIterator1 last1, FwdIterator2 first2, FwdIterator2 last2);
函数功能 1	计算源区间[first1, last1)与目标区间[first2, last2)元素最后一次完全相同的位置
函数原型 2	template<class FwdIterator1, class FwdIterator2, class BinaryPredicate> FwdIterator1 find_end(FwdIterator1 first1, FwdIterator1 last1, FwdIterator2 first2, FwdIterator2 last2, BinaryPredicate comp);
函数功能 2	与上述版本类似，只是比较准则为“使表达式 comp(e1, e2)值为 true”
函数参数	first1 表示源区间起点; last1 表示源区间终点; first2 表示目标区间起点; last2 表示目标区间终点; comp 表示比较准则
返回值	目标区间在源区间中最后一次出现的位置

运用带有函数对象版本的算法 find_end 时，关键在于函数对象的定义，要求该函数类型带有两个参数，返回类型为 bool。

【例 2.1.17】find_end 算法的应用示例。

```
#01     #include <iostream>
#02     #include <algorithm>
#03
#04     bool mIsNegativeOfn(int m, int n) { return m + n == 0;}
#05
#06     int main()
#07     {
#08         int a[] = {1, 2, 5, 6, 5, 6, -1, -2, -1, -2};
#09         size_t n = sizeof(a) / sizeof(*a);
#10
#11         int* p;
#12         p = std::find_end(a, a + n, a + 2, a + 4);
#13         if (p != a + n)
#14             std::cout << "{5, 6} in array start from " << p - a << std::endl;
#15
#16         p = std::find_end(a, a + n, a, a + 2, mIsNegativeOfn);
#17         if (p != a + n)
#18             std::cout << "negative of {1, 2} start from " << p - a << std::endl;
#19     }
```

上述程序中，表达式 find_end(a, a + n, a + 2, a + 4)在数组 a 中查找子数组 {5, 6} 最后一次出现的位置。表达式 find_end(a, a + n, a, a + 2, mIsNegativeOfn)查找子数组 {1, 2} 的相反数最后一次出现的位置。

程序的输出结果如下：

```
{5, 6} in array start from 4
negative of {1, 2} start from 8
```

9. find_first_of

find_first_of 算法介绍详见表 2.1.35。

表 2.1.35 find_first_of 算法

算法名称	find_first_of
函数原型 1	template<class FwdIterator1, class FwdIterator2> FwdIterator1 find_first_of(FwdIterator1 first1, FwdIterator1 last1, FwdIterator2 first2, FwdIterator2 last2);
函数功能 1	查找目标区间[first2, last2)中任一元素在源区间[first1, last1)中第一次出现的位置
函数原型 2	template<class FwdIterator1, class FwdIterator2, class BinaryPredicate> FwdIterator1 find_first_of(FwdIterator1 first1, FwdIterator1 last1, FwdIterator2 first2, FwdIterator2 last2, BinaryPredicate comp);
函数功能 2	与上述版本类似，只是比较准则为“使表达式 comp(e1, e2)值为 true”
函数参数	first1 表示源区间起点；last1 表示源区间终点；first2 表示目标区间起点；last2 表示目标区间终点；comp 表示比较准则
返回值	元素出现的位置

运用带有函数对象版本的算法 find_first_of 时，关键在于函数对象的定义，要求该函数类型带有两个参数，返回类型为 bool。

【例 2.1.18】find_first_of 算法的应用示例。

```
#01     #include <iostream>
#02     #include <algorithm>
#03
#04     bool mIsNegativeOfn(int m, int n) { return m + n == 0;}
#05
#06     int main()
#07     {
#08         int a[] = {1, 2, 5, 6, 5, 6, -2, -1, -1, -2};
#09         size_t n = sizeof(a) / sizeof(*a);
#10
#11         int* p;
#12         p = std::find_first_of(a, a + n, a + 3, a + 5);
#13         if (p != a + n)
#14             std::cout << "first occurrence of element in {6, 5} is at "
#15                 << p - a << std::endl;
#16
#17         p = std::find_first_of(a, a + n, a, a + 2, mIsNegativeOfn);
#18         if (p != a + n)
#19             std::cout << "first occurrence of negative element "
#20                 << "of {1, 2} is at " << p - a << std::endl;
#21     }
```

上述程序中，表达式 find_first_of(a, a + n, a + 3, a + 5)查找子数组 {6, 5} 中任一元素在数组 a 中的第一次出现位置。表达式 find_first_of(a, a + n, a, a + 2, mIsNegativeOfn)查找子数组 {1, 2} 中任一元素的相反数在数组 a 中的第一次出现位置。

程序的输出结果如下：

```
first occurrence of element in {6, 5} is at 2
```

first occurrence of negative element of {1, 2} is at 6

10. adjacent_find

adjacent_find 算法介绍详见表 2.1.36。

表 2.1.36 adjacent_find 算法

算法名称	adjacent_find
函数原型 1	template<class FwdIterator> FwdIterator adjacent_find(FwdIterator first, FwdIterator last);
函数功能 1	查找区间[first, last)中第一对相邻且相等的元素
函数原型 2	template<class FwdIterator, class BinaryPredicate> FwdIterator adjacent_find(FwdIterator first, FwdIterator last, BinaryPredicate comp);
函数功能 2	与上述版本类似，只是比较准则为“相邻元素使 comp(e, nextE)值为 true”。e、nextE 为相邻两元素
函数参数	first 表示区间起点；last 表示区间终点；comp 表示比较准则
返回值	第一个元素出现的位置

运用带有函数对象版本的算法 adjacent_find 时，关键在于函数对象的定义，要求该函数类型带有两个参数，返回类型为 bool。

【例 2.1.19】adjacent_find 算法的应用示例。

```
#01      #include "print.hpp"
#02
#03      bool hasEqualAbs(int m, int n) { return m * m == n * n;}
#04
#05      int main()
#06      {
#07          int a[] = {1, 2, -2, 2, -2, -2, 3, 3, 5};
#08          size_t n = sizeof(a) / sizeof(*a);
#09          print(a, a + n);
#10
#11          int* p;
#12          p = std::adjacent_find(a, a + n);
#13          if (p != a + n)
#14              std::cout << "first two adjacent equal element is at "
#15                  << p - a << std::endl;
#16
#17          p = std::adjacent_find(a, a + n, hasEqualAbs);
#18          if (p != a + n)
#19              std::cout<<"first two adjacent element with equal abs
#20                  value is at "
#21                  << p - a << std::endl;
#22      }
```

上述程序中，adjacent_find(a, a + n)查找数组中第一对相邻、且值相等元素中第一个元素的出现位置。adjacent_find(a, a + n, hasEqualAbs)查找数组中第一对相邻、且具有相等绝对值元素中第一个元素的出现位置。

程序的输出结果如下：

1 2 -2 2 -2 -2 3 3 5

```
first two adjacent equal element is at 4
first two adjacent element with equal abs value is at 1
```

11. equal

equal 算法介绍详见表 2.1.37。

表 2.1.37 equal 算法

算法名称	equal
函数原型 1	template<class InputIterator1, class InputIterator2> bool equal(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2);
函数功能 1	判断区间[first1, last1)与以 first2 为起点的区间元素是否相同
函数原型 2	template<class InputIterator1, class InputIterator2, class BinaryPredicate> bool equal(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, BinaryPredicate comp);
函数功能 2	与上述版本类似，只是比较准则为“两个区间对应元素使 comp(e1, e2)值为 true”。e1、e2 为两区间对应的元素
函数参数	first1 表示第 1 个区间起点；last1 表示第 1 个区间终点；first2 表示第 2 个区间起点；comp 表示比较准则
返回值	true 或 false

运用带有函数对象版本的 equal 算法时，关键在于函数对象的定义，要求该函数类型带有两个参数，返回类型为 bool。

【例 2.1.20】equal 算法的应用示例。

```
#01      #include "print.hpp"
#02      #include "xr.hpp"
#03
#04      bool hasEqualAbs(int m, int n) { return m * m == n * n;}
#05
#06      int main()
#07      {
#08          int a[] = {2, 2, 2, -2, -2};
#09          size_t n = sizeof(a) / sizeof(*a);
#10          xrv(print(a, a + n));
#11
#12          xr(std::equal(a, a + 2, a + 1));
#13          xr(std::equal(a, a + 3, a + 2));
#14          xr(std::equal(a, a + 3, a + 2, hasEqualAbs));
#15      }
```

上述程序中，equal(a, a + 2, a + 1)比较子数组[a, a + 2)和[a + 1, a + 3)是否相等；equal(a, a + 3, a + 2)比较子数组[a, a + 3)和[a + 2, a + 5)是否相等；equal(a, a + 3, a + 2, hasEqualAbs)以具有相等绝对值为准则比较子数组[a, a + 3)和[a + 2, a + 5)。

程序的输出结果如下：

```
#10: print(a, a + n)      ==>2    2    -2  -2
#12: std::equal(a, a + 2, a + 1)  ==>true
#13: std::equal(a, a + 3, a + 2)  ==>false
#14: std::equal(a, a + 3, a + 2, hasEqualAbs) ==>true
```

12. lexicographical_compare

lexicographical_compare 算法介绍详见表 2.1.38。

表 2.1.38 lexicographical_compare 算法

算法名称	lexicographical_compare
函数原型 1	template<class InputIterator1, class InputIterator2> bool lexicographical_compare(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2);
函数功能 1	判断区间[first1, last1)元素是否以字典顺序小于区间[first2, last2)元素, 比较准则为 operator <
函数原型 2	template<class InputIterator1, class InputIterator2, class BinaryPredicate> bool lexicographical_compare(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, BinaryPredicate comp);
函数功能 2	与上述版本类似, 只是比较准则为“两个区间对应元素使 comp(e1, e2)值为 true”
函数参数	first1 表示第 1 个区间起点; last1 表示第 1 个区间终点; first2 表示第 2 个区间起点; last2 表示第 2 个区间终点; comp 表示比较准则
返回值	若第一个区间小于第二个区间, 则返回值为 true, 否则为 false

所谓“字典顺序”是指依次比较两个区间对应位置的元素值, 并根据比较结果进行判断: ①若两个区间中对应元素不等, 则该两个元素的比较结果作为整个区间比较的结果; ②若两个区间长度相同, 并且所有位置的元素值也相同, 则这两个区间相同; ③若第一个区间的所有元素与第二个区间对应位置的元素值都相同, 但是元素个数少于第二个区间, 则第二个区间大。

运用带有函数对象版本的算法 lexicographical_compare 时, 关键在于函数对象的定义, 要求该函数类型带有两个参数, 返回类型为 bool。

【例 2.1.21】lexicographical_compare 算法的应用示例。

```
#01     #include <algorithm>
#02     #include <string>
#03     #include <cctype>
#04     #include "xr.hpp"
#05
#06     bool ignoreCase(char a, char b) { return tolower(a) < tolower(b); }
#07
#08     #define lexi_cmp std::lexicographical_compare
#09
#10     int main()
#11     {
#12         std::string s("abcdefg"), t("abcd"), r("ABCD");
#13
#14         xr(lexi_cmp(t.begin(), t.end(), s.begin(), s.end()));
#15         xr(lexi_cmp(s.begin(), s.end(), t.begin(), t.end()));
#16         xr(lexi_cmp(s.begin(), s.begin() + 3, t.begin(), t.end()));
#17         xr(lexi_cmp(t.begin(), t.end(), t.begin(), t.end()));
#18         xr(lexi_cmp(r.begin(), r.end(), t.begin(), t.end()));
#19         xr(lexi_cmp(r.begin(), r.end(), t.begin(), t.end(), ignoreCase));
#20     }
```

程序的输出结果如下：

```
#14: lexi_cmp(t.begin(), t.end(), s.begin(), s.end())      ==>true
#15: lexi_cmp(s.begin(), s.end(), t.begin(), t.end())      ==>false
#16: lexi_cmp(s.begin(), s.begin() + 3, t.begin(), t.end()) ==>true
#17: lexi_cmp(t.begin(), t.end(), t.begin(), t.end())      ==>false
#18: lexi_cmp(r.begin(), r.end(), t.begin(), t.end())      ==>true
#19: lexi_cmp(r.begin(), r.end(), t.begin(), t.end(), ignoreCase) ==>false
```

1.4.2 可变序列算法

常用可变序列算法的应用形式及功能如表 2.1.39 所示，这类算法常改变区间元素的值。

表 2.1.39 可变序列算法的应用形式及功能

算法应用形式	算法功能
for_each(first, last, func)	对区间[first, last)中每个元素执行可改变操作 func(e)
copy(first, last, dstBeg)	把源区间[first, last)元素复制到以 dstBeg 为起点的目标区间
copy_if(first, last, dstBeg, pr)	把源区间[first, last)中使谓词 pr 为真的元素复制到以 dstBeg 为起点的目标区间
copy_n(first, n, dstBeg)	把源区间[first, first+n)元素复制到以 dstBeg 为起点的目标区间
copy_backward(first, last, dstEnd)	把源区间[first, last)元素复制到以 dstEnd 为终点的目标区间
move(first, last, dstBeg)	把源区间[first, last)元素转移到以 dstBeg 为起点的目标区间
move_backward(first, last, dstEnd)	把源区间[first, last)元素转移到以 dstEnd 为终点的目标区间
transform(first, last, result, func)	把源区间[first, last)元素 e 逐个施加运算 func(e)后的结果复制到以 result 为起点的目标区间
swap_range(first1, last1, first2)	互换区间[first1, last1)与以 first2 为起点的区间中的元素
fill(first, last, val)	把区间[first, last)元素都赋值为 val
fill_n(first, count, val)	把以 first 为起点的区间中前 count 个元素都赋值为 val
generate(first, last, gen)	把区间[first, last)元素依次赋值为函数 gen() 的返回值
generate_n(first, count, gen)	把以 first 为起点的区间中前 count 个元素依次赋值为函数 gen() 的返回值
replace(first, last, oldVal, newVal)	把区间[first, last)中值为 oldVal 的元素赋值为 newVal
replace_if(first, last, pr, val)	把区间[first, last)中使 pr(e)为 true 的元素赋值为 val
replace_copy(first, last, result, oldVal, newVal)	把区间[first, last)元素复制到以 result 为起点的区间中，同时把旧值 oldVal 替换为新值 newVal
replace_copy_if(first, last, result, pr, val)	把区间[first, last)元素复制到以 result 为起点的区间，同时把使 pr(e)为 true 的元素替换为 val
sample(first, last, out, n, g)	从区间[first, last)中随机（按随机数生成器 g）选中 n 个元素写入以 out 为起点的区间
shift_left(first, last, n)	把区间[first, last)中的元素向区间起点方向移动 n 个位置
shift_right(first, last, n)	把区间[first, last)中的元素向区间终点方向移动 n 个位置

1. copy / copy_if / copy_n / copy_backward

copy / copy_if / copy_n / copy_backward 算法介绍详见表 2.1.40～表 2.1.43。

表 2.1.40 copy 算法

算法名称	copy
函数原型	template<class InputIterator, class OutputIterator> OutputIterator copy(InputIterator first, InputIterator last, OutputIterator dstBeg);
函数功能	把源区间[first, last)元素复制到以 dstBeg 为起点的目标区间
函数参数	first 表示源区间起点; last 表示源区间终点; dstBeg 表示目标区间起点
返回值	目标区间终点

表 2.1.41 copy_if 算法

算法名称	copy_if
函数原型	template< class InputIterator, class OutputIterator, class UnaryPredicate > OutputIterator copy_if(InputIterator first, InputIterator last, OutputIterator dstBeg, UnaryPredicate pred);
函数功能	把源区间[first, last)中使谓词 pred 为真的元素复制到以 dstBeg 为起点的目标区间
函数参数	first 表示源区间起点; last 表示源区间终点; dstBeg 表示目标区间起点; pred 表示一元谓词
返回值	目标区间终点

表 2.1.42 copy_n 算法

算法名称	copy_n
函数原型	template<class InputIterator, class Size, class OutputIterator> OutputIterator copy_n(InputIterator first, Size count, OutputIterator dstBeg);
函数功能	把以 first 为起点的源区间中前 count 个元素复制到以 dstBeg 为起点的目标区间
函数参数	first 表示源区间起点; count 表示元素个数; dstBeg 表示目标区间起点
返回值	目标区间终点

表 2.1.43 copy_backward 算法

算法名称	copy_backward
函数原型	template<class BidirectionalIterator1, class BidirectionalIterator2> BidirectionalIterator2 copy_backward(BidirectionalIterator1 first, BidirectionalIterator1 last, BidirectionalIterator2 dstEnd);
函数功能	把源区间[first, last)元素复制到以 dstEnd 为终点的目标区间
函数参数	first 表示源区间起点; last 表示源区间终点; dstEnd 表示目标区间终点
返回值	目标区间起点

注意：算法 copy 与 copy_backward 的区别在于，前者前向遍历（forward），后者反向遍历（backward）。复制同一个区间时，copy 从目标区间的起点开始前向存放元素，copy_backward 从目标区间的终点开始反向存放元素。

【例 2.1.22】copy / copy_if / copy_n / copy_backward 算法的应用示例。

```
#01     #include <iostream>
#02     #include <algorithm>
#03
#04     int main()
#05     {
#06         int a[] = {1, 2, 3, 4, 5, 6};
#07         const size_t n = sizeof(a) / sizeof(*a);
```

```

#08      std::ostream_iterator<int> screen(std::cout, "\\t");
#09      int *p;
#10
#11      std::cout << "a: ";
#12      std::copy(a, a + n, screen);
#13      std::cout << std::endl;
#14
#15      std::cout << "a even: ";
#16      std::copy_if(a, a + n, screen, [](int x){return x % 2 == 0;});
#17      std::cout << std::endl;
#18
#19      std::cout << "a: ";
#20      std::copy_n(a, 3, screen);
#21      std::cout << std::endl;
#22
#23      int b[n];
#24      p = std::copy(a, a + n, b);
#25      std::cout << "b: ";
#26      std::copy(b, p, screen);
#27      std::cout << std::endl;
#28
#29      int c[n];
#30      p = std::copy_backward(a, a + 3, c + 5);
#31      std::cout << "c: ";
#32      std::copy(p, c + 5, screen);
#33      std::cout << std::endl;
#34      }

```

上述程序中, `copy(a, a+n, screen)` 把数组 `a` 的所有元素复制到以 `screen` 为起点的输出流迭代器中 (即向屏幕输出); `copy_if(a, a+n, screen, [](int x){return x%2==0;})` 把数组 `a` 中的偶数输出到屏幕上; `copy_n(a, 3, screen)` 把数组 `a` 的前 3 个元素输出到屏幕上; `copy(a, a+n, b)` 把数组 `a` 的所有元素复制到数组 `b` 中; `copy_backward(a, a+3, c+5)` 把数组 `a` 的前 3 个元素复制到数组 `c` 中, 从终点 `c+5` 反向存放。

程序的输出结果如下:

```

a: 1    2    3    4    5    6
a even: 2    4    6
a: 1    2    3
b: 1    2    3    4    5    6
c: 1    2    3

```

2. move / move_backward

`move / move_backward` 算法介绍详见表 2.1.44 和表 2.1.45。

表 2.1.44 move 算法

算法名称	move
函数原型	template<class InputIterator, class OutputIterator> OutputIterator move(InputIterator first, InputIterator last, OutputIterator dstBeg);
函数功能	把源区间[first, last)元素转移到以 dstBeg 为起点的目标区间
函数参数	first 表示源区间起点; last 表示源区间终点; dstBeg 表示目标区间起点
返回值	目标区间终点

表 2.1.45 move_backward 算法

算法名称	move_backward
函数原型	template<class BidirectionalIterator1, class BidirectionalIterator2> BidirectionalIterator2 move_backward(BidirectionalIterator1 first, BidirectionalIterator1 last, BidirectionalIterator2 dstEnd);
函数功能	把源区间[first, last)元素转移到以 dstEnd 为终点的目标区间
函数参数	first 表示源区间起点; last 表示源区间终点; dstEnd 表示目标区间终点
返回值	目标区间起点

注意：算法 move/move_backward 是相对于 copy/copy_backward 而设置的，在复制元素的过程中采用 move 操作。算法 move_backward 从目标区间的终点开始反向存放元素。

【例 2.1.23】move_backward 算法的应用示例。

```
#01     #include <algorithm>
#02     #include <iostream>
#03     #include <iterator>
#04     #include <string>
#05     #include <string_view>
#06     #include <vector>
#07
#08     using container = std::vector<std::string>;
#09
#10     void print(std::string_view comment, const container& src,
#11                const container& dst = {}) {
#12         auto prn = [](std::string_view name, const container& cont) {
#13             std::cout << name;
#14             for (const auto& s : cont) {
#15                 std::cout << (s.empty() ? "[moved]" : s.data()) << ' ';
#16             }
#17             std::cout << '\n';
#18         };
#19         std::cout << comment << '\n';
#20         prn("src: ", src);
#21         if (dst.empty()) return;
#22         prn("dst: ", dst);
#23     }
#24
#25     int main() {
#26         container src{ "how", "are", "you" };
```

```
#27     container dst{ "fine", "thank", "you", "[and, you]" };
#28     print("Non-overlapping case; before move_backward:", src, dst);
#29     std::move_backward(src.begin(), src.end(), dst.end());
#30     print("After move_backward:", src, dst);
#31
#32     src = { "where", "are", "you", "from" };
#33     print("Overlapping case; before move_backward:", src);
#34     std::move_backward(src.begin(), std::next(src.begin(), 3),
#35     src.end());
#36     print("After move_backward:", src);
#37 }
```

程序的输出结果如下：

```
Non-overlapping case; before move_backward:
src: how are you
dst: fine thank you [and, you]
After move_backward:
src: [moved] [moved] [moved]
dst: fine how are you
Overlapping case; before move_backward:
src: where are you from
After move_backward:
src: [moved] where are you
```

3. transform

transform 算法介绍详见表 2.1.46。

表 2.1.46 transform 算法

算法名称	transform
函数原型 1	template<class InputIterator, class OutputIterator, class UnaryFunction> OutputIterator transform(InputIterator first, InputIterator last, OutputIterator result, UnaryFunction func);
函数功能 1	把源区间[first, last)元素 e 逐个施加运算 func(e)后的结果复制到以 result 为起点的目标区间
函数参数 1	first 表示源区间起点；last 表示源区间终点；result 表示目标区间起点；func 表示一元函数对象
返回值 1	目标区间终点
函数原型 2	template<class InputIterator1, class InputIterator2, class OutputIterator, class BinaryFunction> OutputIterator transform(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, OutputIterator result, BinaryFunction func);
函数功能 2	把源区间[first1, last1)元素 e1 和以 first2 为起点的区间元素 e2 施加运算 func(e1, e2)后的结果复制到以 result 为起点的目标区间中
函数参数 2	first1 表示第 1 个区间起点；last1 表示第 1 个源区间终点； first2 表示第 2 个区间起点；result 表示目标区间起点；func 表示二元函数对象
返回值 2	目标区间终点

算法 transform 的第 1 个版本需要一元函数对象作为参数，该函数类型带一个参数且返回一个值。算法 transform 的第 2 个版本需要二元函数对象作为参数，该函数类型带两个参数且返回一个值。

【例 2.1.24】transform 算法的应用示例。

```

#01     #include "print.hpp"
#02     #include "xr.hpp"
#03
#04     int mulBy10(int m) {return m * 10;}
#05     int calc(int n, int m) {return m / 10 * n;}
#06
#07     int main()
#08     {
#09         int a[] = {1, 2, 3, 4, 5, 6};
#10         const size_t n = sizeof(a) / sizeof(*a);
#11         xrv(print(a, a + n));
#12
#13         int *p;
#14
#15         int b[n];
#16         p = std::transform(a, a + n, b, mulBy10);
#17         xrv(print(b, p));
#18
#19         int c[n];
#20         p = std::transform(a, a + n, b, c, calc);
#21         xrv(print(c, p));
#22     }

```

上述程序中, transform(a, a + n, b, mulBy10)把数组 a 的所有元素乘以 10 后复制到数组 b 中; transform(a, a + n, b, c, calc)把数组 a 的所有元素及数组 b 的所有元素施加运算 calc 后的结果(实际为数组 a 元素的平方值)复制到数组 c 中。

程序的输出结果如下:

```

#11: print(a, a + n)    ==>1   2   3   4   5   6
#17: print(b, p)        ==>10  20  30  40  50  60
#21: print(c, p)        ==>1   4   9  16  25  36

```

4. swap_range

swap_range 算法介绍详见表 2.1.47。

表 2.1.47 swap_range 算法

算法名称	swap_range
函数原型	template<class FwdIterator1, class FwdIterator2> FwdIterator2 swap_ranges(FwdIterator1 first1, FwdIterator1 last1, FwdIterator2 first2);
函数功能	互换区间[first1, last1)与以 first2 为起点的区间中的元素
函数参数	first1 表示第 1 个区间起点; last1 表示第 1 个区间终点; first2 表示第 2 个区间起点
返回值	第 2 个区间终点

【例 2.1.25】swap_range 算法的应用示例。

```

#01     #include "print.hpp"
#02     #include "xr.hpp"
#03

```

```

#04     int main()
#05     {
#06         int a[] = {1, 2, 3, 4, 5, 6};
#07         size_t n = sizeof(a) / sizeof(*a);
#08         xrv(print(a, a + n));
#09
#10         int b[] = {10, 20, 30, 40, 50, 60, 70, 80, 90};
#11         size_t m = sizeof(b) / sizeof(*b);
#12         xrv(print(b, b + m));
#13
#14         int *p = std::swap_ranges(a, a + n, b);
#15         xrv(print(a, a + n));
#16         xrv(print(b, p));
#17         xrv(print(b, b + m));
#18     }

```

上述程序中, `swap_ranges(a, a + n, b)` 对应交换数组 `a` 的所有元素与数组 `b` 中的元素, 由于数组 `b` 的元素个数较多, 故只有部分元素被交换。

程序的输出结果如下:

```

#08: print(a, a + n)    ==>1    2    3    4    5    6
#12: print(b, b + m)    ==>10   20   30   40   50   60   70   80   90
#15: print(a, a + n)    ==>10   20   30   40   50   60
#16: print(b, p)        ==>1    2    3    4    5    6
#17: print(b, b + m)    ==>1    2    3    4    5    6    70   80   90

```

5. fill / fill_n

`fill / fill_n` 算法介绍详见表 2.1.48 和表 2.1.49。

表 2.1.48 fill 算法

算法名称	fill
函数原型	template<class ForwardIterator, class Type> void fill(ForwardIterator first, ForwardIterator last, const Type& val);
函数功能	把区间[first, last)元素都赋值为 val
函数参数	first 表示区间起点; last 表示区间终点; val 表示元素值
返回值	无

表 2.1.49 fill_n 算法

算法名称	fill_n
函数原型	template<class OutputIterator, class Size, class Type> void fill_n(OutputIterator first, Size count, const Type& val);
函数功能	把以 first 为起点的区间中前 count 个元素都赋值为 val
函数参数	first 表示区间起点; count 表示元素个数; val 表示元素值
返回值	无

【例 2.1.26】fill / fill_n 算法的应用示例。

```

#01     #include "print.hpp"
#02     #include "xr.hpp"

```

```

#03
#04     int main()
#05     {
#06         int a[] = {1, 2, 3, 4, 5, 6, 7};
#07         size_t n = sizeof(a) / sizeof(*a);
#08         xrv(print(a, a + n));
#09
#10         std::fill(a, a + 3, 10);
#11         xrv(print(a, a + n));
#12
#13         std::fill_n(a + 3, n - 3, 20);
#14         xrv(print(a, a + n));
#15     }

```

上述程序中, `fill(a, a+3, 10)` 把数组 `a` 的前 3 个元素都赋值为 10; `fill_n(a+3, n-3, 20)` 把从 `a+3` 开始的所有元素都赋值为 20。

程序的输出结果如下:

```

#08: print(a, a + n)    ==>1   2   3   4   5   6   7
#11: print(a, a + n)    ==>10  10  10  4   5   6   7
#14: print(a, a + n)    ==>10  10  10  20  20  20  20

```

6. generate / generate_n

`generate` / `generate_n` 算法介绍详见表 2.1.50 和表 2.1.51。

表 2.1.50 `generate` 算法

算法名称	<code>generate</code>
函数原型	<pre>template<class ForwardIterator, class Generator> void generate(ForwardIterator first, ForwardIterator last, Generator gen);</pre>
函数功能	把区间 <code>[first, last)</code> 元素依次赋值为函数 <code>gen()</code> 的返回值
函数参数	<code>first</code> 表示区间起点; <code>last</code> 表示区间终点; <code>gen</code> 表示无参函数对象
返回值	无

表 2.1.51 `generate_n` 算法

算法名称	<code>generate_n</code>
函数原型	<pre>template<class OutputIterator, class Size, class Generator> void generate_n(OutputIterator first, Size count, Generator gen);</pre>
函数功能	把以 <code>first</code> 为起点的区间中前 <code>count</code> 个元素依次赋值为函数 <code>gen()</code> 的返回值
函数参数	<code>first</code> 表示区间起点; <code>count</code> 表示元素个数; <code>gen</code> 表示无参函数对象
返回值	无

【例 2.1.27】`generate` / `generate_n` 算法的应用示例。

```

#01     #include <ctime>
#02     #include "print.hpp"
#03     #include "xr.hpp"
#04
#05     int rnd() {return 10 + rand() % 90;}
#06

```

```
#07     int main()
#08     {
#09         int a[] = {1, 2, 3, 4, 5, 6, 7};
#10         size_t n = sizeof(a) / sizeof(*a);
#11         xrv(print(a, a + n));
#12
#13         srand(unsigned(time(NULL)));
#14
#15         std::generate(a, a + n, rnd);
#16         xrv(print(a, a + n));
#17
#18         std::generate_n(a, n, rnd);
#19         xrv(print(a, a + n));
#20     }
```

程序的输出结果如下：

```
#11: print(a, a + n)    ==>1    2    3    4    5    6    7
#16: print(a, a + n)    ==>96   83   62   59   19   47   55
#19: print(a, a + n)    ==>63   69   45   32   51   87   88
```

7. replace / replace_if / replace_copy / replace_copy_if

replace / replace_if / replace_copy / replace_copy_if 这4个算法的介绍详见表 2.1.52～表 2.1.55。

表 2.1.52 replace 算法

算法名称	replace
函数原型	template<class ForwardIterator, class Type> void replace(ForwardIterator first, ForwardIterator last, const Type& oldVal, const Type& newVal)
函数功能	把区间[first, last)中值为 oldVal 的元素替换为 newVal
函数参数	first 表示区间起点；last 表示区间终点；oldVal 表示待替换的旧值；newVal 表示替换后的新值
返回值	无

表 2.1.53 replace_if 算法

算法名称	replace_if
函数原型	template<class ForwardIterator, class Predicate, class Type> void replace_if(ForwardIterator first, ForwardIterator last, Predicate pr, const Type& val);
函数功能	把区间[first, last)中使 pr(e)值为 true 的元素替换为 val
函数参数	first 表示区间起点；last 表示区间终点；pr 表示一元谓词；val 表示替换后的新值
返回值	无

表 2.1.54 replace_copy 算法

算法名称	replace_copy
函数原型	template<class InputIterator, class OutputIterator, class Type> OutputIterator replace_copy(InputIterator first, InputIterator last, OutputIterator result, const Type& oldVal, const Type& newVal);
函数功能	把区间[first, last)元素复制到以 result 为起点的区间中，同时把旧值 oldVal 替换为新值 newVal

续表

函数参数	first 表示源区间起点; last 表示源区间终点; result 表示目标区间起点; oldVal 表示待替换的旧值; newVal 表示替换后的新值
返回值	目标区间的终点

表 2.1.55 replace_copy_if 算法

算法名称	replace_copy_if
函数原型	template<class InputIterator, class OutputIterator, class Predicate, class Type> OutputIterator replace_copy_if(InputIterator first, InputIterator last, OutputIterator result, Predicate pr, const Type& val);
函数功能	把区间[first, last)元素复制到以 result 为起点的区间, 同时把使 pr(e)为 true 的元素替换为 val
函数参数	first 表示区间起点; last 表示区间终点; result 表示目标区间起点; pr 表示一元谓词; val 表示替换后的 新值
返回值	目标区间的终点

运用算法 replace_if/replace_copy_if 时, 需要定义一元谓词函数或函数对象, 这些函数类型带有一个参数且返回类型为 bool。

【例 2.1.28】replace / replace_if / replace_copy / replace_copy_if 算法的应用示例。

```
#01     #include "print.hpp"
#02     #include "xr.hpp"
#03
#04     bool isMultipleOf10(int m) {return m % 10 == 0;}
#05
#06     int main()
#07     {
#08         int a[] = {1, 2, 3, 4, 3, 5, 6, 3, 7};
#09         const size_t n = sizeof(a) / sizeof(*a);
#10         xrv(print(a, a + n));
#11
#12         std::replace(a, a + n, 3, 30);
#13         xrv(print(a, a + n));
#14         std::replace_if(a, a + n, isMultipleOf10, 3);
#15         xrv(print(a, a + n));
#16
#17         int b[n];
#18         int* p = std::replace_copy(a, a + n, b, 3, 30);
#19         xrv(print(a, a + n));
#20         xrv(print(b, p));
#21         p = std::replace_copy_if(b, p, a, isMultipleOf10, 3);
#22         xrv(print(b, b + n));
#23         xrv(print(a, p));
#24     }
```

程序的输出结果如下：

```
#10: print(a, a + n)    ==>1    2    3    4    3    5    6    3    7
#13: print(a, a + n)    ==>1    2   30    4   30    5    6   30    7
#15: print(a, a + n)    ==>1    2    3    4    3    5    6    3    7
#19: print(a, a + n)    ==>1    2    3    4    3    5    6    3    7
#20: print(b, p)        ==>1    2   30    4   30    5    6   30    7
#22: print(b, b + n)    ==>1    2   30    4   30    5    6   30    7
#23: print(a, p)        ==>1    2    3    4    3    5    6    3    7
```

8. sample

sample 算法介绍详见表 2.1.56。

表 2.1.56 sample 算法

算法名称	sample
函数原型	template< class PopulationIterator, class SampleIterator, class Distance, class URBG > SampleIterator sample(PopulationIterator first, PopulationIterator last, SampleIterator out, Distance n, URBG&& g);
函数功能	从区间[first, last)中随机（按随机数生成器 g）选中 n 个元素写入以 out 为起点的区间
函数参数	first 表示区间起点；last 表示区间终点；out 表示采样区间起点；n 表示采样数量；g 表示随机数生成器
返回值	采样区间终点

【例 2.1.29】sample 算法的应用示例。

```
#01    #include <iostream>
#02    #include <random>
#03    #include <string>
#04    #include <iterator>
#05    #include <algorithm>
#06
#07    int main() {
#08        std::string in = "hgfedcba", out;
#09        std::sample(in.begin(), in.end(), std::back_inserter(out),
#10                    5, std::mt19937{ std::random_device{}() });
#11        std::cout << in << ", 5 sampled: " << out << '\n';
#12    }
```

9. shift_left / shift_right

shift_left / shift_right 算法介绍详见表 2.1.57 和表 2.1.58。

表 2.1.57 shift_left 算法

算法名称	shift_left
函数原型	template< class ForwardIt > ForwardIt shift_left(ForwardIt first, ForwardIt last, typename std::iterator_traits<ForwardIt>::difference_type n);
函数功能	把区间[first, last)中的元素向区间起点方向移动 n 个位置

表 2.1.58 shift_right 算法

算法名称	shift_right
函数原型	template< class ForwardIt > ForwardIt shift_right(ForwardIt first, ForwardIt last, typename std::iterator_traits<ForwardIt>::difference_type n);
函数功能	把区间[first, last)中的元素向区间终点方向移动 n 个位置
函数参数	first 表示区间起点; last 表示区间终点; n 表示移动距离
返回值	结果区间终点

【例 2.1.30】 shift_left / shift_right 算法的应用示例。

```
#01      #include <iostream>
#02      #include <vector>
#03      #include <algorithm>
#04
#05      int main() {
#06          std::vector<int> v{ 1, 2, 3, 4, 5, 6, 7 };
#07          auto print = [&v]() {
#08              for (const auto e : v) std::cout << e << " ";
#09              std::cout << std::endl;
#10          };
#11          print(); //1 2 3 4 5 6 7
#12
#13          std::shift_left(begin(v), end(v), 3);
#14          print(); //4 5 6 7 5 6 7
#15
#16          std::shift_right(begin(v), end(v), 2);
#17          print(); //4 5 4 5 6 7 5
#18      }
```

1.4.3 去除元素算法

常用去除元素算法的应用形式及功能如表 2.1.59 所示，这类算法从区间中去除满足条件的元素。

表 2.1.59 去除元素算法的应用形式及功能

算法的应用形式	算法的功能
remove(first, last, val)	去除区间[first, last)中所有值为 val 的元素
remove_if(first, last, pr)	去除区间[first, last)中所有使 pr(e)为 true 的元素 e
remove_copy(first, last, result, val)	把区间[first, last)元素复制到以 result 为起点的区间中，同时去除所有值为 val 的元素
remove_copy_if(first, last, result, pr)	把区间[first, last)元素复制到以 result 为起点的区间中，同时去除所有使 pr(e)为 true 的元素
unique(first, last[,pr])	去除区间[first, last)中连续相等的元素，保留该元素第一次出现的位置
unique_copy(first, last, result [,pr])	把区间[first, last)元素复制到以 result 为起点的区间中，同时去除所有与前一元素相同的元素

1. remove / remove_if / remove_copy / remove_copy_if

remove / remove_if / remove_copy / remove_copy_if 算法介绍详见表 2.1.60～表 2.1.63。

表 2.1.60 remove 算法

算法名称	remove
函数原型	template<class ForwardIterator, class Type> ForwardIterator remove(ForwardIterator first, ForwardIterator last, const Type& val);
函数功能	去除区间[first, last)中值为 val 的所有元素
函数参数	first 表示区间起点; last 表示区间终点; val 表示待去除的元素值
返回值	新区间终点

表 2.1.61 remove_if 算法

算法名称	remove_if
函数原型	template<class ForwardIterator, class Predicate> ForwardIterator remove_if(ForwardIterator first, ForwardIterator last, Predicate pr);
函数功能	去除区间[first, last)中使 pr(e)值为 true 的元素 e
函数参数	first 表示区间起点; last 表示区间终点; pr 表示一元谓词
返回值	新区间终点

表 2.1.62 remove_copy 算法

算法名称	remove_copy
函数原型	template<class InputIterator, class OutputIterator, class Type> OutputIterator remove_copy(InputIterator first, InputIterator last, OutputIterator result, const Type& val);
函数功能	把区间[first, last)元素复制到以 result 为起点的区间中, 同时去除值为 val 的元素
函数参数	first 表示源区间起点; last 表示源区间终点; result 表示目标区间起点; val 表示待去除的元素值
返回值	目标区间的终点

表 2.1.63 remove_copy_if 算法

算法名称	remove_copy_if
函数原型	template<class InputIterator, class OutputIterator, class Predicate> OutputIterator remove_copy_if(InputIterator first, InputIterator last, OutputIterator result, Predicate pr);
函数功能	把区间[first, last)元素复制到以 result 为起点的区间中, 同时去除使 pr(e)值为 true 的元素 e
函数参数	first 表示区间起点; last 表示区间终点; result 表示目标区间起点; pr 表示一元谓词
返回值	目标区间的终点

运用算法 remove_if/remove_copy_if 时, 需要定义一元谓词函数或函数对象, 这些函数类型带有一个参数且返回类型为 bool。

【例 2.1.31】remove / remove_if / remove_copy / remove_copy_if 算法的应用示例。

```
#01     #include "print.hpp"
#02     #include "xr.hpp"
#03
#04     bool iseven(int m) {return m % 2 == 0;}
#05
```

```
#06      int main()
#07      {
#08          int a[] = {1, 2, 3, 4, 3, 5, 6, 3, 7};
#09          const size_t n = sizeof(a) / sizeof(*a);
#10          xrv(print(a, a + n));
#11
#12          int b[n];
#13
#14          int* bp = std::remove_copy(a, a + n, b, 3);
#15          xrv(print(a, a + n));
#16          xrv(print(b, bp));
#17
#18          int* ap = std::remove(a, a + n, 3);
#19          xrv(print(a, ap));
#20
#21          bp = std::remove_copy_if(a, ap, b, iseven);
#22          xrv(print(a, ap));
#23          xrv(print(b, bp));
#24
#25          ap = std::remove_if(a, ap, iseven);
#26          xrv(print(a, ap));
#27      }
```

程序的输出结果如下：

```
#10: print(a, a + n)    ==>1    2    3    4    3    5    6    3    7
#15: print(a, a + n)    ==>1    2    3    4    3    5    6    3    7
#16: print(b, bp)       ==>1    2    4    5    6    7
#19: print(a, ap)       ==>1    2    4    5    6    7
#22: print(a, ap)       ==>1    2    4    5    6    7
#23: print(b, bp)       ==>1    5    7
#26: print(a, ap)       ==>1    5    7
```

2. unique / unique_copy

unique / unique_copy 算法介绍详见表 2.1.64 和表 2.1.65。

表 2.1.64 unique 算法

算法名称	unique
函数原型 1	template<class ForwardIterator> ForwardIterator unique(ForwardIterator first, ForwardIterator last);
函数功能 1	去除区间[first, last)中所有相邻且相等的元素，保留第一次出现的位置
函数原型 2	template<class ForwardIterator, class BinaryPredicate> ForwardIterator unique(ForwardIterator first, ForwardIterator last, BinaryPredicate comp);
函数功能 2	去除区间[first, last)中每个“位于元素 e 后且使 comp(elem, e)值为 true”的元素
函数参数	first 表示区间起点；last 表示区间终点；comp 表示比较准则
返回值	新区间终点

表 2.1.65 unique_copy 算法

算法名称	unique_copy
函数原型 1	template<class InputIterator, class OutputIterator> OutputIterator unique_copy(InputIterator first,InputIterator last, OutputIterator result);
函数功能 1	把区间[first, last)元素复制到以 result 为起点的区间中，同时去除所有与前一元素相同的元素
函数原型 2	template<class InputIterator, class OutputIterator, class BinaryPredicate> OutputIterator unique_copy(InputIterator first, InputIterator last, OutputIterator result, BinaryPredicate comp);
函数功能 2	把区间[first, last)元素复制到以 result 为起点的区间中，同时去除“位于元素 e 后且使 comp(elem, e)值为 true”的元素
函数参数	first 表示源区间起点；last 表示源区间终点；result 表示目标区间起点；comp 表示比较准则
返回值	目标区间的终点

运用带有函数对象版本的算法 unique/unique_copy 时，需要定义二元谓词函数或函数对象，这些函数类型带有两个参数且返回类型为 bool。

【例 2.1.32】unique / unique_copy 算法的应用示例。

```
#01      #include "print.hpp"
#02      #include "xr.hpp"
#03
#04      bool bothOdd(int m, int n) {return m % 2 != 0 && n % 2 != 0;}
#05
#06      int main()
#07      {
#08          int a[] = {1, 5, 3, 3, 3, 4, 3, 5, 5, 5};
#09          const size_t n = sizeof(a) / sizeof(*a);
#10          xrv(print(a, a + n));
#11
#12          int b[n];
#13
#14          int* bp = std::unique_copy(a, a + n, b);
#15          xrv(print(b, bp));
#16          int* ap = std::unique(a, a + n);
#17          xrv(print(a, ap));
#18
#19          bp = std::unique_copy(a, ap, b, bothOdd);
#20          xrv(print(b, bp));
#21          ap = std::unique(a, ap, bothOdd);
#22          xrv(print(a, ap));
#23      }
```

上述程序中，unique_copy(a, a+n, b)把数组 a 中相邻且不重复的元素赋值到数组 b 中；unique(a, a+n)去除数组 a 中相邻且重复的元素；unique_copy(a, ap, b, bothOdd)把数组 a 中相邻且不是奇数的元素赋值到数组 b 中；unique(a, ap, bothOdd)去除数组 a 中相邻且是奇数的元素。

程序的输出结果如下：

```
#10: print(a, a + n)    ==>1    5    3    3    3    4    3    5    5    5
#15: print(b, bp)      ==>1    5    3    4    3    5
```

```
#17: print(a, ap)      ==>1   5   3   4   3   5
#20: print(b, bp)      ==>1   4   3
#22: print(a, ap)      ==>1   4   3
```

1.4.4 序列变序算法

常用序列变序算法的应用形式及功能如表 2.1.66 所示，这类多以改变区间元素的相对顺序为目的。

表 2.1.66 序列变序算法的应用形式及功能

算法的应用形式	算法的功能
reverse(first, last)	逆转区间[first, last)中的所有元素
reverse_copy(first, last, result)	把区间[first, last)元素复制到以 result 为起点的区间中，同时逆转元素顺序
rotate(first, middle, last)	旋转区间[first, last)元素，使[middle, last)在[first, middle)之前
rotate_copy(first, middle, last, result)	把区间[middle, last)和[first, middle)元素依次复制到以 result 为起点的区间中
next_permutation(first, last[, comp])	重排区间[first, last)元素顺序，使该顺序依字典升序为当前排列的后一个排列
prev_permutation(first, last[, comp])	重排区间[first, last)元素顺序，使得该顺序依字典降序为当前排列的前一个排列
is_permutation(first1, last1, first2[, last2, comp])	如果区间[first1, last1)的某个排列与区间[first2, first2+(last1-first1)或[first2, last2)相等，则返回 true
random_shuffle(first, last[,rand])	随机重排区间[first, last)元素顺序，rand 为外部传入的随机数生成器
shuffle(first, last, g)	随机重排区间[first, last)元素顺序，g 为随机数生成器
partition(first, last, pr)	把区间[first, last)中使 pr(e)为 true 的元素 e 移动到区间前部
stable_partition(first, last[,pr])	与 partition 同，只是保持元素的相对顺序
is_partitioned(first, last, pr)	如果区间[first, last)中使 pr(e)为 true 的元素 e 都出现在区间前部，则返回 true
partition_copy(first, last, firsttrue, firstfalse, pr)	把区间[first, last)中的元素 e 复制到两个不同的区间，使 pr(e)为 true 的元素复制到 d_first_true 为起点的区间，使 pr(e)为 false 的元素复制到 d_first_false 为起点的区间，返回值为这两个区间终点构成的 pair
partition_point(first, last, pr)	检查经算法 partition 划分过的区间[first, last)，返回第一个不满足谓词 pr 的元素，如果所有元素都满足谓词 pr，则返回 last

1. reverse / reverse_copy

reverse / reverse_copy 算法介绍详见表 2.1.67 和表 2.1.68。

表 2.1.67 reverse 算法

算法名称	reverse
函数原型	template<class BidirectionalIterator> void reverse(BidirectionalIterator first, BidirectionalIterator last);
函数功能	逆转区间[first, last)中所有元素
函数参数	first 表示区间起点；last 表示区间终点
返回值	无

表 2.1.68 reverse_copy 算法

算法名称	reverse_copy
函数原型	template<class BidirectionalIterator, class OutputIterator> OutputIterator reverse_copy(BidirectionalIterator first, BidirectionalIterator last, OutputIterator result);
函数功能	把区间[first, last)元素复制到以 result 为起点的区间中，同时逆转元素顺序
函数参数	first 表示源区间起点; last 表示源区间终点; result 表示目标区间起点
返回值	目标区间的终点

【例 2.1.33】reverse / reverse_copy 算法的应用示例。

```
#01      #include "print.hpp"
#02      #include "xr.hpp"
#03
#04      int main()
#05      {
#06          int a[] = {1, 2, 3, 4, 5, 6, 7, 8};
#07          const size_t n = sizeof(a) / sizeof(*a);
#08          xrv(print(a, a + n));
#09
#10          int b[n];
#11
#12          int* bp = std::reverse_copy(a, a + n, b);
#13          xrv(print(b, bp));
#14          std::reverse(a, a + n);
#15          xrv(print(a, a + n));
#16      }
```

程序的输出结果如下：

```
#08: print(a, a + n)    ==>1    2    3    4    5    6    7    8
#13: print(b, bp)      ==>8    7    6    5    4    3    2    1
#15: print(a, a + n)    ==>8    7    6    5    4    3    2    1
```

2. rotate / rotate_copy

rotate / rotate_copy 算法介绍详见表 2.1.69 和表 2.1.70。

表 2.1.69 rotate 算法

算法名称	rotate
函数原型	template<class ForwardIterator> void rotate(ForwardIterator first, ForwardIterator middle, ForwardIterator last);
函数功能	旋转区间[first, last)元素，使[middle, last)在[first, middle)之前
函数参数	first 表示区间起点; last 表示区间终点; middle 表示区间中的一个位置
返回值	无

表 2.1.70 rotate_copy 算法

算法名称	rotate_copy
函数原型	template<class ForwardIterator, class OutputIterator> OutputIterator rotate_copy(ForwardIterator first, ForwardIterator middle, ForwardIterator last, OutputIterator result);

续表

函数功能	把区间[middle, last)和[first, middle)元素依次复制到以 result 为起点的区间中
函数参数	first 表示源区间起点; last 表示源区间终点; middle 表示区间中的一个位置; result 表示目标区间起点
返回值	目标区间的终点

【例 2.1.34】 rotate / rotate_copy 算法的应用示例。

```
#01    #include "print.hpp"
#02    #include "xr.hpp"
#03
#04    int main()
#05    {
#06        int a[] = {1, 2, 3, 4, 5, 6, 7, 8};
#07        const size_t n = sizeof(a) / sizeof(*a);
#08        xrv(print(a, a + n));
#09
#10        int b[n];
#11        int* p = std::find(a, a + n, 4);
#12        int* bp = std::rotate_copy(a, p, a + n, b);
#13        xrv(print(b, bp));
#14
#15        p = std::find(b, bp, 7);
#16        std::rotate(b, p, bp);
#17        xrv(print(b, bp));
#18    }
```

程序的输出结果如下:

```
#08: print(a, a + n)    ==>1   2   3   4   5   6   7   8
#13: print(b, bp)      ==>4   5   6   7   8   1   2   3
#17: print(b, bp)      ==>7   8   1   2   3   4   5   6
```

3. next_permutation / prev_permutation / is_permutation

next_permutation / prev_permutation / is_permutation 算法介绍详见表 2.1.71~表 2.1.73。

表 2.1.71 next_permutation 算法

算法名称	next_permutation
函数原型 1	template<class BidirectionalIterator> bool next_permutation(BidirectionalIterator first, BidirectionalIterator last);
函数功能 1	重排区间[first, last)元素顺序, 使该顺序依字典升序为当前排列的后一个排列
函数原型 2	template<class BidirectionalIterator, class BinaryPredicate> bool next_permutation(BidirectionalIterator first, BidirectionalIterator last, BinaryPredicate comp);
函数功能 2	与上述版本相同, 只是比较准则为 comp
函数参数	first 表示区间起点; last 表示区间终点; comp 表示排序准则
返回值	直到再没有下一个顺序能够按照字典序升序, 则返回 false; 否则返回 true

表 2.1.72 prev_permutation 算法

算法名称	prev_permutation
函数原型 1	template<class BidirectionalIterator> bool prev_permutation(BidirectionalIterator first, BidirectionalIterator last);
函数功能 1	重排区间[first, last)元素顺序，使该顺序依字典降序为当前排列的前一个排列
函数原型 2	template<class BidirectionalIterator, class BinaryPredicate> bool prev_permutation(BidirectionalIterator first, BidirectionalIterator last, BinaryPredicate comp);
函数功能 2	与上述版本相同，只是比较准则为 comp
函数参数	first 表示区间起点；last 表示区间终点；comp 表示排序准则
返回值	直到再没有下一个顺序能够按照字典序降序，则返回 false；否则返回 true

表 2.1.73 is_permutation 算法

算法名称	is_permutation
函数原型 1	template< class ForwardIt1, class ForwardIt2 > bool is_permutation(ForwardIt1 first1, ForwardIt1 last1, ForwardIt2 first2);
函数功能 1	如果区间[first1, last1)的某个排列与区间[first2, first2+(last1-first1)相等，则返回 true，比较准则为 operator==
函数原型 2	template< class ForwardIt1, class ForwardIt2, class BinaryPredicate > bool is_permutation(ForwardIt1 first1, ForwardIt1 last1, ForwardIt2 first2, BinaryPredicate p);
函数功能 2	如果区间[first1, last1)的某个排列与区间[first2, first2 + (last1 - first1)相等，则返回 true，比较准则为二元谓词 p
函数原型 3	template< class ForwardIt1, class ForwardIt2 > bool is_permutation(ForwardIt1 first1, ForwardIt1 last1, ForwardIt2 first2, ForwardIt2 last2);
函数功能 3	如果区间[first1, last1)的某个排列与区间[first2, last2)相等，则返回 true，比较准则为 operator==
函数原型 4	template< class ForwardIt1, class ForwardIt2, class BinaryPredicate > bool is_permutation(ForwardIt1 first1, ForwardIt1 last1, ForwardIt2 first2, ForwardIt2 last2, BinaryPredicate p);
函数功能 4	如果区间[first1, last1)的某个排列与区间[first2, last2)相等，则返回 true，比较准则为二元谓词 p
函数参数	first1 表示区间 1 起点；last1 表示区间 1 终点；first2 表示区间 2 起点；last2 表示区间 2 终点；p 表示比较准则
返回值	bool

运用带有函数对象版本的算法 next_permutation/prev_permutation 时，需要定义二元谓词函数或函数对象，这些函数类型带有两个参数且返回类型为 bool。

【例 2.1.35】next_permutation / prev_permutation / is_permutation 算法的应用示例。

```
#01      #include <algorithm>
#02      #include "xr.hpp"
#03      #include "print.hpp"
#04
#05      int main() {
#06          int a[]={ 1, 3, 2 };
#07          auto n = sizeof(a) / sizeof(*a);
#08
#09          std::cout << "Ascending firstly: ";
#10          xrv(print(a, a + n));
#11          while (std::next_permutation(a, a + n))
#12              print(a, a + n);
```

```
#13         std::cout << "finally: ";
#14         xrv(print(a, a + n));
#15
#16         std::cout << "Descending firstly: ";
#17         xrv(print(a, a + n));
#18
#19         std::prev_permutation(a, a + n);
#20         xrv(print(a, a + n));
#21
#22         while (std::prev_permutation(a, a + n))
#23             print(a, a + n);
#24         std::cout << "finally: ";
#25         xrv(print(a, a + n));
#26         std::cout << std::endl;
#27
#28         int v1[]{ 1,2,3,4,5 };
#29         int v2[]{ 3,5,4,1,2 };
#30         int v3[]{ 3,5,4,1,1 };
#31
#32         xr(std::is_permutation(v1, v1 + 5, v2)); //true
#33         xr(std::is_permutation(v1, v1 + 5, v3)); //false
#34     }
```

4. random_shuffle / shuffle

random_shuffle / shuffle 算法介绍详见表 2.1.74 和表 2.1.75。

表 2.1.74 random_shuffle 算法

算法名称	random_shuffle
函数原型 1	template<class RandomAccessIterator> void random_shuffle(RandomAccessIterator first, RandomAccessIterator last);
函数功能 1	用内部随机数生成器随机重排区间[first, last)元素的顺序
函数原型 2	template<class RandomAccessIterator, class RandomNumberGenerator> void random_shuffle(RandomAccessIterator first, RandomAccessIterator last, RandomNumberGenerator& rand);
函数功能 2	用随机数生成器 rand 随机重排区间[first, last)元素的顺序
函数参数	first 表示区间起点；last 表示区间终点；rand 表示外部传入的随机数生成器
返回值	无

表 2.1.75 shuffle 算法

算法名称	shuffle
函数原型	template< class RandomIt, class URBG > void shuffle(RandomIt first, RandomIt last, URBG&& g);
函数功能	用随机数生成器 g 随机重排区间[first, last)元素的顺序
函数参数	first 表示区间起点；last 表示区间终点；g 表示外部传入的随机数生成器
返回值	无

【例 2.1.36】 random_shuffle / shuffle 算法的应用示例。

```
#01     #include <random>
```

```
#02     #include <algorithm>
#03     #include <iterator>
#04     #include <iostream>
#05     #include <vector>
#06
#07     int main() {
#08         std::vector<int> v { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
#09         auto print = [&v]() {
#10             std::copy(v.begin(),v.end(),
#11                 std::ostream_iterator<int>(std::cout," "));
#12             std::cout << "\n";
#13         };
#14         std::random_device rd;
#15         std::mt19937 g(rd());
#16
#17         std::random_shuffle(v.begin(), v.end());
#18         std::cout << "random_shuffle: ";
#19         print();
#20
#21         std::shuffle(v.begin(), v.end(), g);
#22         std::cout << "shuffle: ";
#23         print();
#24     }
```

程序的输出结果如下：

```
random_shuffle: 9 2 10 3 1 6 8 4 5 7
shuffle: 8 5 2 4 9 7 3 6 10 1
```

5. partition / stable_partition / is_partitioned / partition_copy / partition_point

partition / stable_partition / is_partitioned / partition_copy / partition_point 算法介绍详见表 2.1.76～表 2.1.80。

表 2.1.76 partition 算法

算法名称	partition
函数原型	template<class BidirectionalIterator, class Predicate> BidirectionalIterator partition(BidirectionalIterator first, BidirectionalIterator last,Predicate pr);
函数功能	把区间[first, last)中使 pr(e)为 true 的元素 e 移动到区间前部
函数参数	first 表示区间起点；last 表示区间终点；pr 表示一元谓词
返回值	第 1 个使 pr(e)为 false 元素的位置

表 2.1.77 stable_partition 算法

算法名称	stable_partition
函数原型	template<class BidirectionalIterator, class Predicate> BidirectionalIterator stable_partition(BidirectionalIterator first, BidirectionalIterator last,Predicate pr);
函数功能	与上述算法同，只是保持元素的相对顺序
函数参数	first 表示区间起点；last 表示区间终点；pr 表示一元谓词
返回值	第 1 个使 pr(e)为 false 元素的位置

表 2.1.78 is_partitioned 算法

算法名称	is_partitioned
函数原型	template<class InputIterator, class UnaryPredicate> bool is_partitioned(InputIterator first, InputIterator last, UnaryPredicate pr);
函数功能	如果区间[first, last)中使 pr(e)为 true 的元素 e 都出现在区间前部, 则返回 true
函数参数	first 表示区间起点; last 表示区间终点; pr 表示一元谓词
返回值	如果区间为空则返回 true, 或者如果区间被谓词 pr 划分了也为 true

表 2.1.79 partition_copy 算法

算法名称	partition_copy
函数原型	template< class InputIt, class OutputIt1, class OutputIt2, class UnaryPredicate > std::pair<OutputIt1, OutputIt2> partition_copy(InputIt first, InputIt last, OutputIt1 d_first_true, OutputIt2 d_first_false, UnaryPredicate pr);
函数功能	把区间[first, last)中的元素 e 复制到两个不同的区间, 使 pr(e)为 true 的元素复制到以 d_first_true 为起点的区间, 使 pr(e)为 false 的元素复制到以 d_first_false 为起点的区间, 返回值为这两个区间起点构成的 pair
函数参数	first 表示区间起点; last 表示区间终点; d_first_true 表示满足谓词 pr 的元素构成的区间的起点; d_first_false 表示不满足谓词 pr 的元素构成的区间的起点; pr 表示一元谓词
返回值	返回 pair, 分别是以 d_first_true 为起点的区间终点和以 d_first_false 为起点的区间终点

表 2.1.80 partition_point 算法

算法名称	partition_point
函数原型	template< class ForwardIt, class UnaryPredicate > ForwardIt partition_point(ForwardIt first, ForwardIt last, UnaryPredicate pr);
函数功能	检查经算法 partition 划分过的区间[first, last), 返回第一个不满足谓词 pr 的元素, 如果所有元素都满足谓词 pr, 则返回 last
函数参数	first 表示区间起点; last 表示区间终点; pr 表示一元谓词
返回值	返回值为第一个不满足谓词 pr 的元素, 如果所有元素都满足谓词 pr, 则返回 last

【例 2.1.37】partition / stable_partition / is_partitioned / partition_copy / partition_point 算法的应用示例。

```

#01    #include <algorithm>
#02    #include <iostream>
#03    #include "xr.hpp"
#04
#05    int main() {
#06        int a[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
#07        auto n = sizeof(a) / sizeof(*a);
#08        auto print = [](int* beg, int*end) {
#09            std::copy(beg, end, std::ostream_iterator<int>(std::cout, " "));
#10            std::cout << "\n";
#11        };
#12        auto is_even = [](int x) { return x % 2 == 0; };
#13
#14        xr(std::is_partitioned(a, a+n, is_even));           //false
#15
#16        std::partition(a, a+n, is_even);

```

```

#17      xr(std::is_partitioned(a, a + n, is_even));    //true
#18
#19      int atrue[5], afalse[5];
#20      std::partition_copy(a, a + n, atrue, afalse, is_even);
#21      print(atrue, atrue + 5);    //使谓词为 true 的区间: 10 2 8 4 6
#22      print(afalse, afalse + 5);  //使谓词为 false 的区间: 5 7 3 9 1
#23
#24      auto p = std::partition_point(a, a+n, is_even);
#25      print(a, p);                //划分点之前的区间: 10 2 8 4 6
#26      print(p, a + n);            //划分点之后的区间: 5 7 3 9 1
#27      }

```

1.4.5 序列排序算法

常用序列排序算法的应用形式及功能如表 2.1.81 所示，这是一类特殊的变序算法，它们使区间元素有序。

表 2.1.81 序列排序算法的应用形式及功能

算法的应用形式	算法的功能
<code>sort(first, last[,comp])</code>	对区间 <code>[first, last)</code> 元素排序， <code>comp</code> 为比较准则
<code>stable_sort(first, last[,comp])</code>	与 <code>sort</code> 相同，只是保持元素的相对顺序， <code>comp</code> 为比较准则
<code>is_sorted(first, last[,comp])</code>	检查区间 <code>[first, last)</code> 元素是否以非递减顺序排列， <code>comp</code> 为比较准则
<code>is_sorted_until(first, last[,comp])</code>	查找区间 <code>[first, last)</code> 中从 <code>first</code> 开始的最大区间，其元素以非递减顺序排列，比较准则默认为 <code>operator<</code> ，否则为 <code>comp</code>
<code>partial_sort(first, sortEnd, last[,comp])</code>	局部排序，使区间 <code>[first, sortEnd)</code> 元素有序， <code>comp</code> 为比较准则
<code>partial_sort_copy(first1, last1, first2, last2[,comp])</code>	把源区间 <code>[first1, last1)</code> 元素复制到目标区间 <code>[first2, last2)</code> ，同时对元素排序， <code>comp</code> 为比较准则
<code>nth_element(first, nth, last[,comp])</code>	对区间 <code>[first, last)</code> 元素排序，使在位置 <code>nth</code> 之前的元素都小于等于它，在位置 <code>nth</code> 之后的元素都大于它， <code>comp</code> 为比较准则
<code>make_heap(first, last[,comp])</code>	把区间 <code>[first, last)</code> 转化为 <code>heap</code> ， <code>comp</code> 为比较准则
<code>push_heap(first, last[,comp])</code>	向区间 <code>[first, last-1)</code> 中加入元素 <code>last-1</code> ，并保持区间 <code>[first, last)</code> 为 <code>heap</code> ， <code>comp</code> 为比较准则
<code>pop_heap(first, last[,comp])</code>	把区间 <code>[first, last)</code> 第 1 个元素移到最后，并保持区间 <code>[first, last-1)</code> 为 <code>heap</code> ， <code>comp</code> 为比较准则
<code>sort_heap(first, last[,comp])</code>	把 <code>heap</code> 区间 <code>[first, last)</code> 转化为有序序列，则该区间不再成为 <code>heap</code> ， <code>comp</code> 为比较准则
<code>is_heap(first, last[,comp])</code>	判断区间 <code>[first, last)</code> 中的元素是否最大堆，比较准则默认为 <code>operator<</code> ，否则为 <code>comp</code>
<code>is_heap_until(first, last[,comp])</code>	查找区间 <code>[first, last)</code> 中从 <code>first</code> 开始的最大区间，其元素构成最大堆，比较准则默认为 <code>operator<</code> ，否则为 <code>comp</code>

1. `sort` / `stable_sort` / `is_sorted` / `is_sorted_until`

`sort` / `stable_sort` / `is_sorted` / `is_sorted_until` 算法介绍详见表 2.1.82～表 2.1.85。

表 2.1.82 sort 算法

算法名称	sort
函数原型 1	template<class RandomAccessIterator> void sort(RandomAccessIterator first, RandomAccessIterator last);
函数功能 1	对区间[first, last)元素从小到大排序, 排序准则为 operator <
函数原型 2	template<class RandomAccessIterator, class BinaryPredicate> void sort(RandomAccessIterator first, RandomAccessIterator last, BinaryPredicate comp);
函数功能 2	对区间[first, last)元素从小到大排序, 排序准则为 comp
函数参数	first 表示区间起点; last 表示区间终点; comp 表示排序准则
返回值	无

表 2.1.83 stable_sort 算法

算法名称	stable_sort
函数原型 1	template<class BidirectionalIterator> void stable_sort(BidirectionalIterator first, BidirectionalIterator last);
函数功能 1	与算法 sort 对应版本相同, 只是保持元素的相对顺序
函数原型 2	template<class RandomAccessIterator, class BinaryPredicate> void stable_sort(RandomAccessIterator first, RandomAccessIterator last, BinaryPredicate comp);
函数功能 2	与算法 sort 对应版本相同, 只是保持元素的相对顺序
函数参数	first 表示区间起点; last 表示区间终点; comp 表示排序准则
返回值	无

表 2.1.84 is_sorted 算法

算法名称	is_sorted
函数原型 1	template<class RandomAccessIterator> bool is_sorted(RandomAccessIterator first, RandomAccessIterator last);
函数功能 1	检查区间[first, last)元素是否以非递减顺序排列, 比较准则为 operator <
函数原型 2	template<class RandomAccessIterator, class BinaryPredicate> void is_sorted(RandomAccessIterator first, RandomAccessIterator last, BinaryPredicate comp);
函数功能 2	检查区间[first, last)元素是否以非递减顺序排列, comp 为比较准则
函数参数	first 表示区间起点; last 表示区间终点; comp 表示排序准则
返回值	bool

表 2.1.85 is_sorted_until 算法

算法名称	is_sorted_until
函数原型 1	template< class ForwardIt > ForwardIt is_sorted_until(ForwardIt first, ForwardIt last);
函数功能 1	查找区间[first, last)中从 first 开始的最大区间, 其元素以非递减顺序排列, 比较准则为 operator <
函数原型 2	template< class ForwardIt, class Compare > ForwardIt is_sorted_until(ForwardIt first, ForwardIt last, Compare comp);
函数功能 2	查找区间[first, last)中从 first 开始的最大区间, 其元素以非递减顺序排列, 比较准则为 comp
函数参数	first 表示区间起点; last 表示区间终点; comp 表示排序准则
返回值	返回以 first 为起点的最大区间的终点

运用带有函数对象版本的算法 `sort/stable_sort` 时，需要定义二元谓词函数或函数对象，这些函数类型带有两个参数且返回类型为 `bool`。

【例 2.1.38】`sort / stable_sort / is_sorted / is_sorted_until` 算法的应用示例。

```
#01      #include "print.hpp"
#02      #include "xr.hpp"
#03
#04      bool compByAbs(int m, int n) { return m * m < n * n; }
#05
#06      int main() {
#07          int a[]={ 1, -3, 2, 5, -4, -5, 6, -7 };
#08          auto n = sizeof(a) / sizeof(*a);
#09          xrv(print(a, a + n));
#10
#11          std::sort(a, a + n);
#12          xrv(print(a, a + n));
#13
#14          std::sort(a, a + n, compByAbs);
#15          xrv(print(a, a + n));
#16
#17          std::stable_sort(a, a + n);
#18          xrv(print(a, a + n));
#19
#20          std::stable_sort(a, a + n, compByAbs);
#21          xrv(print(a, a + n));
#22
#23          xr(std::is_sorted(a, a + n));
#24
#25          auto p = std::is_sorted_until(a, a + n);
#26          xrv(print(a, p)); xrv(print(p, a + n));
#27      }
```

程序的输出结果如下：

```
#09: [print(a, a + n)] ==>1   -3   2   5   -4   -5   6   -7
#12: [print(a, a + n)] ==>-7  -5  -4  -3   1   2   5   6
#15: [print(a, a + n)] ==>1   2   -3  -4   -5   5   6   -7
#18: [print(a, a + n)] ==>-7  -5  -4  -3   1   2   5   6
#21: [print(a, a + n)] ==>1   2   -3  -4   -5   5   6   -7
#23: [std::is_sorted(a, a + n)] ==>[false]
#26: [print(a, p)]      ==>1   2
#26: [print(p, a + n)] ==>-3  -4  -5   5   6   -7
```

2. `partial_sort / partial_sort_copy`

`partial_sort / partial_sort_copy` 算法介绍详见表 2.1.86 和表 2.1.87。

表 2.1.86 `partial_sort` 算法

算法名称	<code>partial_sort</code>
函数原型 1	<code>template<class RandomAccessIterator></code> <code>void partial_sort(RandomAccessIterator first, RandomAccessIterator sortEnd, RandomAccessIterator last);</code>

续表

函数功能 1	局部排序, 使区间[first, sortEnd)元素有序, 排序准则为 operator <
函数原型 2	template<class RandomAccessIterator, class BinaryPredicate> void partial_sort(RandomAccessIterator first, RandomAccessIterator sortEnd, RandomAccessIterator last, BinaryPredicate comp);
函数功能 2	局部排序, 使区间[first, sortEnd)元素有序, 排序准则为 comp
函数参数	first 表示区间起点; last 表示区间终点; sortEnd 表示排序区间终点; comp 表示排序准则
返回值	无

表 2.1.87 partial_sort_copy 算法

算法名称	partial_sort_copy
函数原型 1	template<class InputIterator, class RandomAccessIterator> RandomAccessIterator partial_sort_copy(InputIterator first1, InputIterator last1, RandomAccessIterator first2, RandomAccessIterator last2);
函数功能 1	把源区间[first1, last1)元素复制到目标区间[first2, last2), 同时对元素按照准则 operator < 排序
函数原型 2	template<class InputIterator, class RandomAccessIterator, class BinaryPredicate> RandomAccessIterator partial_sort_copy(InputIterator first1, InputIterator last1, RandomAccessIterator first2, RandomAccessIterator last2, BinaryPredicate comp);
函数功能 2	把源区间[first1, last1)元素复制到目标区间[first2, last2), 同时对元素按照准则 comp 排序
函数参数	first1 表示源区间起点; last1 表示源区间终点; first2 表示目标区间起点; last2 表示目标区间终点; comp 表示排序准则
返回值	目标区间中第一个未被覆盖的元素位置

运用带有函数对象版本的算法 partial_sort/partial_sort_copy 时, 需要定义二元谓词函数或函数对象, 这些函数类型带有两个参数且返回类型为 bool。

【例 2.1.39】partial_sort / partial_sort_copy 算法的应用示例。

```
#01     #include "print.hpp"
#02     #include "xr.hpp"
#03
#04     bool compByAbs(int m, int n) {return m * m < n * n;}
#05
#06     int main()
#07     {
#08         int a[] = {1, -3, 2, 5, -4, 6, -7};
#09         const size_t n = sizeof(a) / sizeof(*a);
#10         xrv(print(a, a + n));
#11
#12         std::partial_sort(a, a + 3, a + n);
#13         xrv(print(a, a + n));
#14
#15         std::partial_sort(a, a + 4, a + n, compByAbs);
#16         xrv(print(a, a + n));
#17
#18         int b[n + 3];
#19         int* p = std::partial_sort_copy(a, a + n, b, b + 4);
```

```
#20         xrv(print(b, p));
#21
#22         p = std::partial_sort_copy(a, a + n, b, b + n + 3);
#23         xrv(print(b, p));
#24
#25         p =std::partial_sort_copy(a, a + n, b, b + 4, compByAbs);
#26         xrv(print(b, p));
#27     }
```

程序的输出结果如下：

```
#10: print(a, a + n)    ==>1    -3  2   5   -4  6   -7
#13: print(a, a + n)    ==>-7    -4  -3  5    2   6    1
#16: print(a, a + n)    ==>1     2   -3  -4   -7  6    5
#20: print(b, p)        ==>-7    -4  -3  1
#23: print(b, p)        ==>-7    -4  -3  1   2   5    6
#26: print(b, p)        ==>1     2   -3  -4
```

3. nth_element

nth_element 算法介绍详见表 2.1.88。

表 2.1.88 nth_element 算法

算法名称	nth_element
函数原型 1	template<class RandomAccessIterator> void nth_element(RandomAccessIterator first, RandomAccessIterator nth, RandomAccessIterator last);
函数功能 1	对区间[first, last)元素排序，使在位置 nth 之前的元素都小于等于它，在位置 nth 之后的元素都大于它，排序准则为 operator <
函数原型 2	template<class RandomAccessIterator, class BinaryPredicate> void nth_element(RandomAccessIterator first, RandomAccessIterator nth, RandomAccessIterator last, BinaryPredicate comp);
函数功能 2	对区间[first, last)元素排序，使在位置 nth 之前的元素都小于等于它，在位置 nth 之后的元素都大于它，排序准则为 comp
函数参数	first 表示区间起点；last 表示区间终点；nth 表示区间中的有效位置；comp 表示排序准则
返回值	无

运用带有函数的版本对象算法 nth_element 时，需要定义二元谓词函数或函数对象，这些函数类型带有两个参数且返回类型为 bool。

【例 2.1.40】nth_element 算法的应用示例。

```
#01     #include "print.hpp"
#02     #include "xr.hpp"
#03
#04     bool compByAbs(int m, int n) {return m * m < n * n;}
#05
#06     int main()
#07     {
#08         int a[] = {1, -3, 2, -1, -5, -4, 6};
#09         const size_t n = sizeof(a) / sizeof(*a);
#10         xrv(print(a, a + n));
```

```
#11
#12         std::nth_element(a, a + 3, a + n);
#13         xrv(print(a, a + n));
#14
#15         std::nth_element(a, a + 3, a + n, compByAbs);
#16         xrv(print(a, a + n));
#17     }
```

程序的输出结果如下：

```
#10: print(a, a + n)    ==>1    -3    2    -1    -5    -4    6
#13: print(a, a + n)    ==>-5    -4    -3    -1    1    2    6
#16: print(a, a + n)    ==>-1    1    2    -3    -4    -5    6
```

4. make_heap / push_heap / pop_heap / sort_heap / is_heap / is_heap_until

make_heap / push_heap / pop_heap / sort_heap / is_heap / is_heap_until 算法介绍详见表 2.1.89～表 2.1.94。

表 2.1.89 make_heap 算法

算法名称	make_heap
函数原型 1	template<class RandomAccessIterator> void make_heap(RandomAccessIterator first, RandomAccessIterator last);
函数功能 1	把区间[first, last)转化为 heap，排序准则为 operator <
函数原型 2	template<class RandomAccessIterator, class BinaryPredicate> void make_heap(RandomAccessIterator first, RandomAccessIterator last, BinaryPredicate comp);
函数功能 2	把区间[first, last)转化为 heap，排序准则为 comp
函数参数	first 表示区间起点；last 表示区间终点；comp 表示排序准则
返回值	无

表 2.1.90 push_heap 算法

算法名称	push_heap
函数原型 1	template<class RandomAccessIterator> void push_heap(RandomAccessIterator first, RandomAccessIterator last);
函数功能 1	向区间[first, last-1)中加入元素 last-1，并保持区间[first, last)为 heap，排序准则为 operator <
函数原型 2	template<class RandomAccessIterator, class BinaryPredicate> void push_heap(RandomAccessIterator first, RandomAccessIterator last, BinaryPredicate comp);
函数功能 2	向区间[first, last-1)中加入元素 last-1，并保持区间[first, last)为 heap，排序准则为 comp
函数参数	first 表示区间起点；last 表示区间终点；comp 表示排序准则
返回值	无

表 2.1.91 pop_heap 算法

算法名称	pop_heap
函数原型 1	template<class RandomAccessIterator> void pop_heap(RandomAccessIterator first, RandomAccessIterator last);
函数功能 1	把区间[first, last)第 1 个元素移到最后，并保持区间[first, last-1)为 heap，排序准则为 operator <

续表

函数原型 2	template<class RandomAccessIterator, class BinaryPredicate> void pop_heap(RandomAccessIterator first, RandomAccessIterator last, BinaryPredicate comp);
函数功能 2	把区间[first, last)第 1 个元素移到, 并保持区间[first, last-1)为 heap, 排序准则为 comp
函数参数	first 表示区间起点; last 表示区间终点; comp 表示排序准则
返回值	无

表 2.1.92 sort_heap 算法

算法名称	sort_heap
函数原型 1	template<class RandomAccessIterator> void sort_heap(RandomAccessIterator first, RandomAccessIterator last);
函数功能 1	把 heap 区间[first, last)转化为有序序列, 排序准则为 operator <
函数原型 2	template<class RandomAccessIterator, class BinaryPredicate> void sort_heap(RandomAccessIterator first, RandomAccessIterator last, BinaryPredicate comp);
函数功能 2	把 heap 区间[first, last)转化为有序序列, 排序准则为 comp
函数参数	first 表示区间起点; last 表示区间终点; comp 表示排序准则
返回值	无

表 2.1.93 is_heap 算法

算法名称	is_heap
函数原型 1	template<class RandomAccessIterator> bool is_heap(RandomAccessIterator first, RandomAccessIterator last);
函数功能 1	判断区间[first, last)中的元素是否最大堆, 比较准则默认为 operator <
函数原型 2	template<class RandomAccessIterator, class BinaryPredicate> bool is_heap(RandomAccessIterator first, RandomAccessIterator last, BinaryPredicate comp);
函数功能 2	判断区间[first, last)中的元素是否最大堆, 比较准则为 comp
函数参数	first 表示区间起点; last 表示区间终点; comp 表示排序准则
返回值	bool

表 2.1.94 is_heap_until 算法

算法名称	is_heap_until
函数原型 1	template<class RandomAccessIterator> RandomAccessIterator is_heap_until(RandomAccessIterator first, RandomAccessIterator last);
函数功能 1	查找区间[first, last)中从 first 开始的最大区间, 其元素构成最大堆, 比较准则默认为 operator <
函数原型 2	template<class RandomAccessIterator, class BinaryPredicate> RandomAccessIterator is_heap_until(RandomAccessIterator first, RandomAccessIterator last, BinaryPredicate comp);
函数功能 2	查找区间[first, last)中从 first 开始的最大区间, 其元素构成最大堆, 比较准则为 comp
函数参数	first 表示区间起点; last 表示区间终点; comp 表示排序准则
返回值	返回以 first 为起点的最大区间的终点

运用带有函数对象版本的算法 make_heap/push_heap/pop_heap/sort_heap 时, 需要定义二元谓词函数或函数对象, 这些函数类型带有两个参数且返回类型为 bool。

【例 2.1.41】 make_heap / push_heap / pop_heap / sort_heap / is_heap / is_heap_until 算法的应用示例。

```
#01      #include "print.hpp"
#02      #include "xr.hpp"
#03
#04      int main() {
#05          int a[] { 1, 3, 2, 6, 5, 4, 0, 0 };
#06          auto n = sizeof(a) / sizeof(*a);
#07
#08          auto pend = a + n - 2;
#09          xrv(print(a, pend));
#10          xr(std::is_heap(a, pend));
#11
#12          auto p = std::is_heap_until(a+1, pend);
#13          xrv(print(a+1, p));
#14
#15          std::make_heap(a, pend);
#16          xrv(print(a, pend));
#17
#18          *pend = 2, ++pend;
#19          std::push_heap(a, pend);
#20          xrv(print(a, pend));
#21
#22          std::pop_heap(a, pend);
#23          --pend;
#24          xrv(print(a, pend));
#25
#26          std::sort_heap(a, pend);
#27          xrv(print(a, pend));
#28      }
```

程序的输出结果如下：

```
#09: [print(a, pend)] ==>1   3       2       6       5       4
#10: [std::is_heap(a, pend)] ==>[false]
#13: [print(a+1, p)] ==>3   2
#16: [print(a, pend)] ==>6   5       4       3       1       2
#20: [print(a, pend)] ==>6   5       4       3       1       2       2
#24: [print(a, pend)] ==>5   3       4       2       1       2
#27: [print(a, pend)] ==>1   2       2       3       4       5
```

1.4.6 已序序列算法

常用已序序列算法的应用形式及功能如表 2.1.95 所示，这类算法常对已序区间进行查找、比较、合并等操作。

表 2.1.94 已序序列算法的应用形式及功能

算法的应用形式	算法的功能
<code>binary_search(first, last, val[,comp])</code>	判断已序区间 <code>[first, last)</code> 中是否存在值为 <code>val</code> 的元素， <code>comp</code> 为比较准则
<code>include(first1, last1, first2, last2[,comp])</code>	判断已序区间 <code>[first1, last1)</code> 中是否包含已序区间 <code>[first2, last2)</code> ， <code>comp</code> 为比较准则
<code>lower_bound(first, last, val[,comp])</code>	返回已序区间 <code>[first, last)</code> 中第一个大于等于 <code>val</code> 元素的位置， <code>comp</code> 为比较准则
<code>upper_bound(first, last, val[,comp])</code>	返回已序区间 <code>[first, last)</code> 中第一个大于 <code>val</code> 元素的位置， <code>comp</code> 为比较准则
<code>equal_range(first, last, val[,comp])</code>	返回已序区间 <code>[first, last)</code> 中值等于 <code>val</code> 的元素集合的起点和终点， <code>comp</code> 为比较准则
<code>merge(first1, last1, first2, last2, result [,comp])</code>	把已序区间 <code>[first1, last1)</code> 和 <code>[first2, last2)</code> 合并成以 <code>result</code> 为起点的有序区间，保留两个区间中的所有元素， <code>comp</code> 为比较准则
<code>inplace_merge(first, middle, last[,comp])</code>	把已在一个序列中的有序子区间 <code>[first, middle)</code> 和 <code>[middle, last)</code> 合并成有序区间 <code>[first, last)</code> ，保留两个子区间中的所有元素， <code>comp</code> 为比较准则
<code>set_union(first1, last1, first2, last2, result[,comp])</code>	把已序区间 <code>[first1, last1)</code> 和 <code>[first2, last2)</code> 合并成以 <code>result</code> 为起点的有序区间，以较多次数保留重复元素， <code>comp</code> 为比较准则
<code>set_intersection(first1, last1, first2, last2, result [,comp])</code>	把已序区间 <code>[first1, last1)</code> 和 <code>[first2, last2)</code> 中相同元素合并成以 <code>result</code> 为起点的有序区间，以较少次数保留重复元素， <code>comp</code> 为比较准则
<code>set_difference(first1, last1, first2, last2, result [,comp])</code>	把只存在于已序区间 <code>[first1, last1)</code> 中、而不出现于已序区间 <code>[first2, last2)</code> 中的元素合并成以 <code>result</code> 为起点的有序区间，以重复次数正差保留第一个区间中的重复元素， <code>comp</code> 为比较准则
<code>set_symmetric_difference(first1, last1, first2, last2, result [,comp])</code>	去除已序区间 <code>[first1, last1)</code> 和 <code>[first2, last2)</code> 中相同的元素，把两个区间中剩余元素合并为以 <code>result</code> 为起点的有序区间，以重复次数正差保留两个区间中的重复元素， <code>comp</code> 为比较准则

1. `binary_search`

`binary_search` 算法介绍详见表 2.1.96。

表 2.1.96 `binary_search` 算法

算法名称	<code>binary_search</code>
函数原型 1	<code>template<class ForwardIterator, class Type></code> <code>bool binary_search(ForwardIterator first, ForwardIterator last, const Type& val);</code>
函数功能 1	判断已序区间 <code>[first, last)</code> 中是否存在值为 <code>val</code> 的元素
函数原型 2	<code>template<class ForwardIterator, class Type, class BinaryPredicate></code> <code>bool binary_search(ForwardIterator first, ForwardIterator last, const Type& val, BinaryPredicate comp);</code>
函数功能 2	判断已序区间 <code>[first, last)</code> 中是否存在使 <code>comp(e1, e2)</code> 为 <code>true</code> 的元素 <code>e</code>
函数参数	<code>first</code> 表示区间起点； <code>last</code> 表示区间终点； <code>val</code> 表示元素值； <code>comp</code> 表示排序准则
返回值	若存在，则返回 <code>true</code> ；否则返回 <code>false</code>

运用带有函数对象版本的算法 `binary_search` 时，需要定义二元谓词函数或函数对象，这些函数类型带有两个参数且返回类型为 `bool`。

【例 2.1.42】binary_search 算法的应用示例。

```
#01      #include "print.hpp"
#02      #include "xr.hpp"
#03
#04      bool compByAbs(int m, int n) {return m * m < n * n;}
#05
#06      int main()
#07      {
#08          int a[] = {1, 3, 2, -1, 5, -4};
#09          const size_t n = sizeof(a) / sizeof(*a);
#10
#11          std::stable_sort(a, a + n);
#12          xrv(print(a, a + n));
#13          xr(std::binary_search(a, a + n, 3));
#14
#15          std::stable_sort(a, a + n, compByAbs);
#16          xrv(print(a, a + n));
#17          xr(std::binary_search(a, a + n, 4, compByAbs));
#18      }
```

程序的输出结果如下：

```
#12: print(a, a + n)    ==>-4  -1  1  2  3  5
#13: std::binary_search(a, a + n, 3)    ==>true
#16: print(a, a + n)    ==>-1  1  2  3  -4  5
#17: std::binary_search(a, a + n, 4, compByAbs)    ==>true
```

2. includes

includes 算法介绍详见表 2.1.97。

表 2.1.97 includes 算法

算法名称	includes
函数原型 1	template<class InputIterator1, class InputIterator2> bool includes(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last1);
函数功能 1	判断已序区间[first1, last1)中是否包含已序区间[first2, last2)
函数原型 2	template<class InputIterator1, class InputIterator2, class BinaryPredicate> bool includes(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last1, BinaryPredicate comp);
函数功能 2	与上述算法类似，只是比较准则为 comp
函数参数	first1 表示区间起点；last1 表示区间终点；first2 表示子区间起点；last2 表示子区间终点；comp 表示排序准则
返回值	若存在，则返回 true；否则返回 false

运用带有函数对象版本的算法 includes 时，需要定义二元谓词函数或函数对象，这些函数类型带有两个参数且返回类型为 bool。

【例 2.1.43】includes 算法的应用示例。

```
#01      #include "print.hpp"
#02      #include "xr.hpp"
```

```
#03
#04     bool compByAbs(int m, int n) {return m * m < n * n;}
#05
#06     int main()
#07     {
#08         int a[] = {1, 3, 2, -1, 5, -4};
#09         const size_t n = sizeof(a) / sizeof(*a);
#10         int b[] = {-2, 1, 3};
#11         const size_t m = sizeof(b) / sizeof(*b);
#12
#13         std::stable_sort(a, a + n);
#14         xrv(print(a, a + n));
#15         std::stable_sort(b, b + m);
#16         xrv(print(b, b + m));
#17         xr(std::includes(a, a + n, b, b + m));
#18
#19         std::stable_sort(a, a + n, compByAbs);
#20         xrv(print(a, a + n));
#21         std::stable_sort(b, b + m, compByAbs);
#22         xrv(print(b, b + m));
#23         xr(std::includes(a, a + n, b, b + m, compByAbs));
#24     }
```

程序的输出结果如下:

```
#14: print(a, a + n)    ==>-4  -1  1  2  3  5
#16: print(b, b + m)    ==>-2  1  3
#17: std::includes(a, a + n, b, b + m) ==>false
#20: print(a, a + n)    ==>-1  1  2  3  -4  5
#22: print(b, b + m)    ==>1  -2  3
#23: std::includes(a, a + n, b, b + m, compByAbs) ==>>true
```

3. lower_bound / upper_bound / equal_range

算法 lower_bound (表 2.1.98) 返回的是可以插入数值 val 且不影响已序性的第一个位置。

表 2.1.98 lower_bound 算法

算法名称	lower_bound
函数原型 1	template<class ForwardIterator, class Type> ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last, const Type& val);
函数功能 1	在已序区间[first, last)寻找第一个大于等于 val 元素的位置
函数原型 2	template<class ForwardIterator, class Type, class BinaryPredicate> ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last, const Type& val, BinaryPredicate comp);
函数功能 2	与上述版本类似，只是排序准则为 comp
函数参数	first 表示区间起点; last 表示区间终点; val 表示元素值; comp 表示排序准则
返回值	第一个大于等于 val 的元素位置

算法 upper_bound (表 2.1.99) 返回的是可以插入数值 val 且不影响已序性的最后一

个位置。

表 2.1.99 upper_bound 算法

算法名称	upper_bound
函数原型 1	template<class ForwardIterator, class Type> ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last, const Type& val);
函数功能 1	在已序区间[first, last)寻找第一个大于 val 元素的位置
函数原型 2	template<class ForwardIterator, class Type, class BinaryPredicate> ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last, const Type& val, BinaryPredicate comp);
函数功能 2	与上述版本类似，只是排序准则为 comp
函数参数	first 表示区间起点；last 表示区间终点；val 表示元素值；comp 表示排序准则
返回值	第一个大于 val 的元素位置

算法 equal_range (表 2.1.100) 同时完成了算法 lower_bound 和 upper_bound 的工作。

表 2.1.100 equal_range 算法

算法名称	equal_range
函数原型 1	template<class ForwardIterator, class Type> pair<ForwardIterator, ForwardIterator> equal_range (ForwardIterator first, ForwardIterator last, const Type& val);
函数功能 1	在已序区间[first, last)寻找值等于 val 的元素集合位置
函数原型 2	template<class ForwardIterator, class Type, class BinaryPredicate> pair<ForwardIterator, ForwardIterator> equal_range(ForwardIterator first, ForwardIterator last, const Type& val, BinaryPredicate comp);
函数功能 2	与上述版本类似，只是排序准则为 comp
函数参数	first 表示区间起点；last 表示区间终点；val 表示元素值；comp 表示排序准则
返回值	值等于 val 元素的起点和终点组成的 pair

运用带有函数对象版本的算法 lower_bound/upper_bound/equal_range 时，需要定义二元谓词函数或函数对象，这些函数类型带有两个参数且返回类型为 bool。

【例 2.1.44】lower_bound / upper_bound / equal_range 算法的应用示例。

```
#01      #include "print.hpp"
#02      #include "xr.hpp"
#03
#04      bool compByAbs(int m, int n) {return m * m < n * n;}
#05
#06      int main()
#07      {
#08          int a[] = {1, 3, 2, -2, 5, -4, 2, -2, 2};
#09          const size_t n = sizeof(a) / sizeof(*a);
#10
#11          std::stable_sort(a, a + n);
#12          print(a, a + n, "sort by < : ");
#13
#14          int* p = std::lower_bound(a, a + n, 2);
#15          std::cout << "first 2 is at " << p - a << std::endl;
#16          p = std::upper_bound(a, a + n, 2);
```

```
#17      std::cout << "last 2 is at " << p - a << std::endl;
#18      std::pair<int*, int*> pr = std::equal_range(a, a + n, 2);
#19      std::cout << "value 2 in range [" << pr.first - a << ", "
#20              << pr.second - a << ")\n";
#21
#22      std::stable_sort(a, a + n, compByAbs);
#23      print(a, a + n, "sort by abs: ");
#24
#25      p = std::lower_bound(a, a + n, 2, compByAbs);
#26      std::cout << "first abs 2 is at " << p - a << std::endl;
#27      p = std::upper_bound(a, a + n, 2, compByAbs);
#28      std::cout << "last abs 2 is at " << p - a << std::endl;
#29      pr = std::equal_range(a, a + n, 2, compByAbs);
#30      std::cout << "abs value 2 in range [" << pr.first - a << ", "
#31              << pr.second - a << ")\n";
#32      }
```

程序的输出结果如下：

```
sort by < : -4 -2 -2 1 2 2 2 3 5
first 2 is at 4
last 2 is at 7
value 2 in range [4, 7).
sort by abs: 1 -2 -2 2 2 2 3 -4 5
first abs 2 is at 1
last abs 2 is at 6
abs value 2 in range [1, 6).
```

4. merge / inplace_merge

merge / inplace_merge 算法介绍详见表 2.1.101 和表 2.1.102。

表 2.1.101 merge 算法

算法名称	merge
函数原型 1	template<class InputIterator1, class InputIterator2, class OutputIterator> OutputIterator merge(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result);
函数功能 1	把已序区间[first1, last1)和[first2, last2)合并成以 result 为起点的有序区间，保留两个区间中的所有元素
函数原型 2	template<class InputIterator1, class InputIterator2, class OutputIterator, class BinaryPredicate> OutputIterator merge(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result, BinaryPredicate comp);
函数功能 2	与上述版本类似，只是比较准则为 comp
函数参数	first1 表示第 1 个区间起点；last1 表示第 1 个区间终点；first2 表示第 2 个区间起点；last2 表示第 2 个区间终点；result 表示目标区间起点；comp 表示排序准则
返回值	目标区间终点

表 2.1.102 inplace_merge 算法

算法名称	inplace_merge
函数原型 1	template<class BidirectionalIterator> void inplace_merge(BidirectionalIterator first, BidirectionalIterator middle, BidirectionalIterator last);
函数功能 1	把已在一个序列中的有序子区间[first, middle)和[middle, last)合并成有序区间[first, last)，保留两个子区间中的所有元素
函数原型 2	template<class BidirectionalIterator, class BinaryPredicate> void inplace_merge(BidirectionalIterator first, BidirectionalIterator middle, BidirectionalIterator last, BinaryPredicate comp);
函数功能 2	与上述版本类似，只是比较准则为 comp
函数参数	first 表示区间起点；last 表示区间终点；middle 表示区间中间有效位置；comp 表示排序准则
返回值	无

运用带有函数对象版本的算法 merge/inplace_merge 时，需要定义二元谓词函数或函数对象，这些函数类型带有两个参数且返回类型为 bool。

【例 2.1.45】merge / inplace_merge 算法的应用示例。

```
#01     #include "print.hpp"
#02     #include "xr.hpp"
#03
#04     bool compByAbs(int m, int n) {return m * m < n * n;}
#05
#06     int main()
#07     {
#08         int a[] = {1, 3, -2, 2};
#09         const size_t n = sizeof(a) / sizeof(*a);
#10         int b[] = {-5, 4, -2, 2, -3, 2};
#11         const size_t m = sizeof(b) / sizeof(*b);
#12
#13         std::sort(a, a + n);
#14         print(a, a + n, "a: ");
#15         std::sort(b, b + m);
#16         print(b, b + m, "b: ");
#17
#18         int c[n + m];
#19         int* p = std::merge(a, a + n, b, b + m, c);
#20         print(c, p, "c: ");
#21
#22         std::sort(c, c + 4, compByAbs);
#23         print(c, c + 4, "c[0, 4): ");
#24         std::sort(c + 4, p, compByAbs);
#25         print(c + 4, p, "c[4, ..): ");
#26         std::inplace_merge(c, c + 4, p, compByAbs);
#27         print(c, p, "c: ");
#28     }
```

程序的输出结果如下：

a: -2 1 2 3

b: -5 -3 -2 2 2 4
c: -5 -3 -2 -2 1 2 2 2 3 4
c[0, 4): -2 -2 -3 -5
c[4, ..): 1 2 2 2 3 4
c: 1 -2 -2 2 2 2 -3 3 4 -5

5. set_union / set_intersection / set_difference / set_symmetric_difference

set_union / set_intersection / set_difference / set_symmetric_difference 算法介绍详见表 2.1.103~表 2.1.106。

表 2.1.103 set_union 算法

算法名称	set_union
函数原型 1	template<class InputIterator1, class InputIterator2, class OutputIterator> OutputIterator set_union(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result);
函数功能 1	把已序区间[first1, last1)和[first2, last2)合并成以 result 为起点的有序区间，以较多次数保留重复元素
函数原型 2	template<class InputIterator1, class InputIterator2, class OutputIterator, class BinaryPredicate> OutputIterator set_union(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result, BinaryPredicate comp);
函数功能 2	与上述版本类似，只是比较准则为 comp
函数参数	first1 表示第 1 个区间起点；last1 表示第 1 个区间终点；first2 表示第 2 个区间起点；last2 表示第 2 个区间终点；result 表示目标区间起点；comp 表示排序准则
返回值	目标区间终点

表 2.1.104 set_intersection 算法

算法名称	set_intersection
函数原型 1	template<class InputIterator1, class InputIterator2, class OutputIterator> OutputIterator set_intersection(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result);
函数功能 1	把已序区间[first1, last1)和[first2, last2)中相同的元素合并成以 result 为起点的有序区间，以较少次数保留重复元素
函数原型 2	template<class InputIterator1, class InputIterator2, class OutputIterator, class BinaryPredicate> OutputIterator set_intersection(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result, BinaryPredicate comp);
函数功能 2	与上述版本类似，只是比较准则为 comp
函数参数	first1 表示第 1 个区间起点；last1 表示第 1 个区间终点；first2 表示第 2 个区间起点；last2 表示第 2 个区间终点；result 表示目标区间起点；comp 表示排序准则
返回值	目标区间终点

表 2.1.105 set_difference 算法

算法名称	set_difference
函数原型 1	template<class InputIterator1, class InputIterator2, class OutputIterator> OutputIterator set_difference(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result);

续表

函数功能 1	把只存在于已序区间[first1, last1)中、而不出现于已序区间[first2, last2)中的元素合并成以 result 为起点的有序区间，以重复次数正差保留第一个区间中的重复元素
函数原型 2	template<class InputIterator1, class InputIterator2, class OutputIterator, class BinaryPredicate> OutputIterator set_difference(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result, BinaryPredicate comp);
函数功能 2	与上述版本类似，只是比较准则为 comp
函数参数	first1 表示第 1 个区间起点；last1 表示第 1 个区间终点；first2 表示第 2 个区间起点； last2 表示第 2 个区间终点；result 表示目标区间起点；comp 表示排序准则
返回值	目标区间终点

表 2.1.106 set_symmetric_difference 算法

算法名称	set_symmetric_difference
函数原型 1	template<class InputIterator1, class InputIterator2, class OutputIterator> OutputIterator set_symmetric_difference(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result);
函数功能 1	去除已序区间[first1, last1)和[first2, last2) 中相同的元素，把两个区间中剩余元素合并为以 result 为起点的有序区间，以重复次数正差保留两个区间中的重复元素
函数原型 2	template<class InputIterator1, class InputIterator2, class OutputIterator, class BinaryPredicate> OutputIterator set_symmetric_difference(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result, BinaryPredicate comp);
函数功能 2	与上述版本类似，只是比较准则为 comp
函数参数	first1 表示第 1 个区间起点；last1 表示第 1 个区间终点；first2 表示第 2 个区间起点； last2 表示第 2 个区间终点；result 表示目标区间起点；comp 表示排序准则
返回值	目标区间终点

运用带有函数对象版本的算法 set_union/set_intersection/set_difference/set_symmetric_difference 时，需要定义二元谓词函数或函数对象，这些函数类型带有两个参数且返回类型为 bool。

【例 2.1.46】merge 集合运算 set_X 算法的应用示例。

```
#01      #include "print.hpp"
#02
#03      int main()
#04      {
#05          int a[] = {1, 2, 2, 4, 6, 6, 7, 7, 9};
#06          const size_t n = sizeof(a) / sizeof(*a);
#07          int b[] = {2, 2, 2, 3, 5, 6, 6, 7, 8};
#08          const size_t m = sizeof(b) / sizeof(*b);
#09          print(a, a + n, "a:\t");
#10          print(b, b + m, "b:\t");
#11
#12          int c[n + m];
#13          int* p = std::merge(a, a + n, b, b + m, c);
#14          print(c, p, "merge:\t");
#15
```

```
#16      p = std::set_union(a, a + n, b, b + m, c);
#17      print(c, p, "union:\t");
#18
#19      p = std::set_intersection(a, a + n, b, b + m, c);
#20      print(c, p, "sect:\t");
#21
#22      p = std::set_difference(a, a + n, b, b + m, c);
#23      print(c, p, "diff:\t");
#24
#25      p = std::set_symmetric_difference(a, a + n, b, b + m, c);
#26      print(c, p, "sdiff:\t");
#27      }
```

程序的输出结果如下：

```
a:      1  2  2  4  6  6  7  7  9
b:      2  2  2  3  5  6  6  7  8
merge:  1  2  2  2  2  2  3  4  5  6  6  6  6  7  7  7
      8  9
union:  1  2  2  2  3  4  5  6  6  7  7  8  9
sect:   2  2  6  6  7
diff:   1  4  7  9
sdiff:  1  2  3  4  5  7  8  9
```

1.4.7 数值算法

常用数值算法的应用形式及功能如表 2.1.107 所示，这类算法主要用于数值计算，以及在相对值和绝对值之间转换。

表 2.1.107 数值算法的应用形式及功能

算法应用形式	算法功能
accumulate(first, last, val)	计算 val + *first + *(first+1) +...+ *(last-1)
accumulate(first, last, val, op)	计算 val op *first op *(first+1) op...op *(last-1)
reduce(first, last, init)	计算 val + *first + *(first+1) +...+ *(last-1)
reduce(first, last, init, op)	计算 val op *first op *(first+1) op...op *(last-1)
inner_product(first1, last1, first2, val)	计算 val + (*first1 * *first2) + (*(first1+1) * *(first2+1)) +...+ (*(last1-1) * *(first2+(last1-first1-1)))
inner_product(first1, last1, first2, val, op1, op2)	计算 val op1 (*first1 op2 *first2) op1 (*(first1+1) op2 *(first2+1)) op1... op1 (*(last1-1) op2 *(first2+(last1-first1-1)))
partial_sum(first, last, result)	把相对值转换为绝对值，即以 result 为起点的区间是下列数据序列： *first, *first+ *(first+1), *first+ *(first+1)+ *(first+2),..., *first+ *(first+1) +...+ *(last-1)
partial_sum(first, last, result, op)	把相对值转换为绝对值，即以 result 为起点的区间是下列数据序列： *first, *first op *(first+1), *first op *(first+1) op *(first+2),..., *first op *(first+1) op...op *(last-1)
inclusive_scan(first, last, result)	把相对值转换为绝对值，即以 result 为起点的区间是下列数据序列： *first, *first+ *(first+1), *first+ *(first+1)+ *(first+2),..., *first+ *(first+1) +...+ *(last-1)

续表

算法应用形式	算法功能
<code>inclusive_scan(first, last, result, op, init)</code>	把相对值转换为绝对值，即以 <code>result</code> 为起点的区间是下列数据序列： <code>init op *first, init op *first op *(first+1), init op *first op *(first+1) op</code> <code>*(first+2),...,init op *first op *(first+1) op...op *(last-1)</code>
<code>exclusive_scan(first, last, result, init)</code>	把相对值转换为绝对值，即以 <code>result</code> 为起点的区间是下列数据序列： <code>init, init + *first, init + *first+ *(first+1), init + *first+ *(first+1)+</code> <code>*(first+2),..., init + *first+ *(first+1) +...+ *(last-2)</code>
<code>exclusive_scan(first, last, result, init, op)</code>	把相对值转换为绝对值，即以 <code>result</code> 为起点的区间是下列数据序列： <code>init, init op *first, init op *first op *(first+1), init op *first op *(first+1) op</code> <code>*(first+2),...,init op *first op *(first+1) op...op *(last-2)</code>
<code>adjacent_difference(first, last, result)</code>	把绝对值转换为相对值，即以 <code>result</code> 为起点的区间是下列数据序列： <code>*first, *(first+1)- *first, *(first+2)- *(first+1),..., *(last-1)- *(last-2)</code>
<code>adjacent_difference (first, last, result, op)</code>	把绝对值转换为相对值，即以 <code>result</code> 为起点的区间是下列数据序列： <code>*first, *(first+1)op *first, *(first+2)op *(first+1),..., *(last-1)op *(last-2)</code>
<code>iota(first, last, value)</code>	用从 <code>value</code> 连续增加的数值填充区间 <code>[first, last)</code>

1. accumulate / reduce

accumulate / reduce 算法介绍详见表 2.1.108 和表 2.1.109。

表 2.1.108 accumulate 算法

算法名称	accumulate
函数原型 1	<code>template<class InputIterator, class Type></code> <code>Type accumulate(InputIterator first, InputIterator last, Type val);</code>
函数功能 1	计算 <code>val + *first + *(first+1) +...+ *(last-1)</code>
函数原型 2	<code>template<class InputIterator, class Type, class BinaryOperation></code> <code>Type accumulate(InputIterator first, InputIterator last, Type val, BinaryOperation op);</code>
函数功能 2	计算 <code>val op *first op *(first+1) op...op *(last-1)</code>
函数参数	<code>first</code> 表示区间起点； <code>last</code> 表示区间终点； <code>val</code> 表示初始值； <code>op</code> 表示二元运算函数对象
返回值	数值结果

表 2.1.109 reduce 算法

算法名称	reduce
函数原型 1	<code>template<class InputIterator, class Type></code> <code>Type accumulate(InputIterator first, InputIterator last, Type val);</code>
函数功能 1	计算 <code>val + *first + *(first+1) +...+ *(last-1)</code>
函数原型 2	<code>template<class InputIterator, class Type, class BinaryOperation></code> <code>Type accumulate(InputIterator first, InputIterator last, Type val, BinaryOperation op);</code>
函数功能 2	计算 <code>val op *first op *(first+1) op...op *(last-1)</code>
函数参数	<code>first</code> 表示区间起点； <code>last</code> 表示区间终点； <code>val</code> 表示初始值； <code>op</code> 表示二元运算函数对象
返回值	数值结果

运用带有函数对象版本的算法 `accumulate/reduce` 时，需要定义二元运算函数对象，该函数类型要求带有两个参数且返回一个值。`reduce` 与 `accumulate` 非常相像，除了区间

元素可能分组并以任意顺序排列。

【例 2.1.47】accumulate/reduce 算法的应用示例。

```
#01     #include <numeric>
#02     #include <functional>
#03     #include "print.hpp"
#04     #include "xr.hpp"
#05
#06     int add(int m, int n) { return m + n; }
#07     int sub(int m, int n) { return m - n; }
#08     int mul(int m, int n) { return m * n; }
#09
#10     int main() {
#11         int a[] { 1, 2, 3, 4, 5 };
#12         auto n = sizeof(a) / sizeof(*a);
#13         print(a, a + n, "a:\t");
#14
#15         xr(std::accumulate(a, a + n, 0));           //结果为 15
#16         xr(std::accumulate(a, a + n, 1000));        //结果为 1015
#17
#18         xr(std::accumulate(a, a + n, 0, add));       //结果为 15
#19         xr(std::accumulate(a, a + n, 0, std::plus<int>()));
#20                                                     //结果为 15
#21         xr(std::accumulate(a, a + n, 0, sub));       //结果为-15
#22         xr(std::accumulate(a, a + n, 0, std::minus<int>()));
#23                                                     //结果为-15
#24         xr(std::accumulate(a, a + n, 0, mul));       //结果为 0
#25         xr(std::accumulate(a, a + n, 1, std::multiplies<int>()));
#26                                                     //结果为 120
#27         xr(std::reduce(a, a + n, 0));                //结果为 15
#28         xr(std::reduce(a, a + n, 1000));             //结果为 1015
#29
#30         xr(std::reduce(a, a + n, 0, add));            //结果为 15
#31         xr(std::reduce(a, a + n, 0, std::plus<int>()));
#32                                                     //结果为 15
#33         xr(std::reduce(a, a + n, 0, sub));            //结果为-15
#34         xr(std::reduce(a, a + n, 0, std::minus<int>()));
#35                                                     //结果为-15
#36         xr(std::reduce(a, a + n, 0, mul));            //结果为 0
#37         xr(std::reduce(a, a + n, 1, std::multiplies<int>()));
#38                                                     //结果为 120
#39     }
```

2. inner_product

inner_product 算法介绍详见表 2.1.110。

表 2.1.110 inner_product 算法

算法名称	inner_product
函数原型 1	template<class InputIterator1, class InputIterator2, class Type> Type inner_product(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, Type val);
函数功能 1	计算 $val + (*first1 * *first2) + (*(first1+1) * *(first2+1)) + \dots + (*(last1-1) * *(first2+(last1-first1)-1))$
函数原型 2	template<class InputIterator1, class InputIterator2, class Type, class BinaryOperation1, class BinaryOperation2> Type inner_product(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, Type val, BinaryOperation1 op1, BinaryOperation2 op2);
函数功能 2	计算 $val\ op1\ (*first1\ op2\ *first2)\ op1\ (*(first1+1)\ op2\ *(first2+1))\ op1\ \dots$ $op1\ (*(last1-1)\ op2\ *(first2+(last1-first1)-1))$
函数参数	first1 表示第 1 个区间起点; last1 表示第 1 个区间终点; first2 表示第 2 个区间起点; val 表示初始值; op1 表示二元运算函数对象; op2 表示二元运算函数对象
返回值	数值结果

运用带有函数对象版本的算法 inner_product 时，需要定义两个二元运算函数对象，该函数类型要求带有两个参数且返回一个值。注意第二个函数应用于两个区间对应的元素。

【例 2.1.48】inner_product 算法的应用示例。

```
#01     #include <numeric>
#02     #include "print.hpp"
#03     #include "xr.hpp"
#04
#05     int add(int m, int n) {return m + n;}
#06     int mul(int m, int n) {return m * n;}
#07
#08     int main()
#09     {
#10         int a[] = {1, 2, 3, 4, 5};
#11         const size_t n = sizeof(a) / sizeof(*a);
#12         print(a, a + n, "a:\t");
#13
#14         xr(std::inner_product(a, a + n, a, 0));           //结果为 55
#15
#16         xr(std::inner_product(a, a + n, a, 0, add, mul)); //结果为 55
#17         xr(std::inner_product(a, a + n, a, 1, mul, add)); //结果为 3840
#18     }
```

3. partial_sum / inclusive_scan / exclusive_scan

partial_sum / inclusive_scan / exclusive_scan 算法介绍详见表 2.1.111~表 2.1.113。

表 2.1.111 partial_sum 算法

算法名称	partial_sum
函数原型 1	template<class InputIterator, class OutputIterator> OutputIterator partial_sum(InputIterator first, InputIterator last, OutputIterator result);
函数功能 1	把相对值转换为绝对值，即以 result 为起点的区间是下列数据序列： *first, *first+ *(first+1), *first+ *(first+1)+ *(first+2),..., *first+ *(first+1) +...+ *(last-1)
函数原型 2	template<class InputIterator, class OutputIterator, class BinaryOperation> OutputIterator partial_sum(InputIterator first, InputIterator last, OutputIterator result, BinaryOperation op);
函数功能 2	把相对值转换为绝对值，即以 result 为起点的区间是下列数据序列： *first, *first op *(first+1), *first op *(first+1) op *(first+2),..., *first op *(first+1) op...op *(last-1)
函数参数	first 表示源区间起点；last 表示源区间终点；result 表示目标区间起点；op 表示二元运算函数对象
返回值	目标区间终点

表 2.1.112 inclusive_scan 算法

算法名称	inclusive_scan
函数原型 1	template<class InputIterator, class OutputIterator> OutputIterator inclusive_scan(InputIterator first, InputIterator last, OutputIterator result);
函数功能 1	把相对值转换为绝对值，即以 result 为起点的区间是下列数据序列： *first, *first+ *(first+1), *first+ *(first+1)+ *(first+2),..., *first+ *(first+1) +...+ *(last-1)
函数原型 2	template<class InputIterator, class OutputIterator, class BinaryOperation> OutputIterator inclusive_scan(InputIterator first, InputIterator last, OutputIterator result, BinaryOperation op, T init);
函数功能 2	把相对值转换为绝对值，即以 result 为起点的区间是下列数据序列： init op *first, init op *first op *(first+1), init op *first op *(first+1) op *(first+2),..., init op *first op *(first+1) op...op *(last-1)
函数参数	first 表示源区间起点；last 表示源区间终点；result 表示目标区间起点；op 表示二元运算函数对象；init 表示初始值
返回值	目标区间终点

表 2.1.113 exclusive_scan 算法

算法名称	exclusive_scan
函数原型 1	template<class InputIterator, class OutputIterator> OutputIterator exclusive_scan(InputIterator first, InputIterator last, OutputIterator result T init);
函数功能 1	把相对值转换为绝对值，即以 result 为起点的区间是下列数据序列： init, init + *first, init + *first+ *(first+1), init + *first+ *(first+1)+ *(first+2),..., init + *first+ *(first+1) +...+ *(last-2)
函数原型 2	template<class InputIterator, class OutputIterator, class BinaryOperation> OutputIterator exclusive_scan(InputIterator first, InputIterator last, OutputIterator result, T init, BinaryOperation op);
函数功能 2	把相对值转换为绝对值，即以 result 为起点的区间是下列数据序列： init, init op *first, init op *first op *(first+1), init op *first op *(first+1) op *(first+2),..., init op *first op *(first+1) op...op *(last-2)
函数参数	first 表示源区间起点；last 表示源区间终点；result 表示目标区间起点；op 表示二元运算函数对象；init 表示初始值
返回值	目标区间终点

运用带有函数对象版本的算法 partial_sum 时，需要定义二元运算函数对象，该函数

类型要求带有两个参数且返回一个值。算法 `inclusive_scan` 与 `partial_sum` 相似，它在第 i 个和中包含第 i 个位置的元素；算法 `exclusive_scan` 与 `partial_sum` 相似，它在第 i 个和中不包含第 i 个位置的元素。

【例 2.1.49】 `partial_sum` / `exclusive_scan` / `inclusive_scan` 算法的应用示例。

```
#01     #include <numeric>
#02     #include "print.hpp"
#03     #include "xr.hpp"
#04
#05     int add(int m, int n) { return m + n; }
#06     int mul(int m, int n) { return m * n; }
#07
#08     int main() {
#09         int a[]{ 3, 1, 4, 1, 5, 9, 2, 6 };
#10         auto n = sizeof(a) / sizeof(*a);
#11         std::ostream_iterator<int> screen(std::cout, " ");
#12         xrv(print(a, a+n));
#13
#14         xrv(std::partial_sum(a, a + n, screen));
#15         std::cout << std::endl;
#16         xrv(std::partial_sum(a, a + n, screen, add));
#17         std::cout << std::endl;
#18         xrv(std::partial_sum(a, a + n, screen, mul));
#19         std::cout << std::endl;
#20
#21         xrv(std::exclusive_scan(a, a + n, screen, 0));
#22         std::cout << std::endl;
#23         xrv(std::inclusive_scan(a, a+n, screen));
#24         std::cout << std::endl;
#25
#26         xrv(std::exclusive_scan(a, a + n, screen, 1, mul));
#27         std::cout << std::endl;
#28         xrv(std::inclusive_scan(a, a + n, screen, mul));
#29         std::cout << std::endl;
#30     }
```

程序的输出结果如下：

```
#12: [print(a, a+n)]    ==>3  1  4  1  5  9  2  6
#14: [std::partial_sum(a, a + n, screen)]    ==>3 4 8 9 14 23 25 31
#16: [std::partial_sum(a, a + n, screen, add)] ==>3 4 8 9 14 23 25 31
#18: [std::partial_sum(a, a + n, screen, mul)] ==>3 3 12 12 60 540 1080 6480
#21: [std::exclusive_scan(a, a + n, screen, 0)] ==>0 3 4 8 9 14 23 25
#23: [std::inclusive_scan(a, a+n, screen)]    ==>3 4 8 9 14 23 25 31
#26: [std::exclusive_scan(a, a + n, screen, 1, mul)] ==>1 3 3 12 12 60 540 1080
#28: [std::inclusive_scan(a, a + n, screen, mul)] ==>3 3 12 12 60 540 1080 6480
```

4. adjacent_difference

`adjacent_difference` 算法介绍详见表 2.1.114。

表 2.1.114 adjacent_difference 算法

算法名称	adjacent_difference
函数原型 1	template<class InputIterator, class OutputIterator> OutputIterator adjacent_difference(InputIterator first, InputIterator last, OutputIterator result);
函数功能 1	把绝对值转换为相对值，即以 result 为起点的区间是下列数据序列： *first, *(first+1)-*first, *(first+2)- *(first+1),...,*(last-1)-*(last-2)
函数原型 2	template<class InputIterator, class OutputIterator, class BinaryOperation> OutputIterator adjacent_difference(InputIterator first, InputIterator last, OutputIterator result, BinaryOperation op);
函数功能 2	把绝对值转换为相对值，即以 result 为起点的区间是下列数据序列： *first, *(first+1)op *first, *(first+2)op *(first+1),...,*(last-1)op *(last-2)
函数参数	first 表示源区间起点；last 表示源区间终点；result 表示目标区间起点；op 表示二元运算函数对象
返回值	目标区间终点

运用带有函数对象版本的算法 adjacent_difference 时，需要定义二元运算函数对象，该函数类型要求带有两个参数且返回一个值。

【例 2.1.50】 adjacent_difference 算法的应用示例。

```
#01      #include <numeric>
#02      #include "print.hpp"
#03
#04      int add(int m, int n) {return m + n;}
#05      int sub(int m, int n) {return m - n;}
#06      int mul(int m, int n) {return m * n;}
#07
#08      int main()
#09      {
#10          int a[] = {1, 2, 3, 4, 5};
#11          const size_t n = sizeof(a) / sizeof(*a);
#12          print(a, a + n, "a:\t");
#13
#14          int b[n];
#15          int* p = std::adjacent_difference(a, a + n, b);
#16          print(b, p, "b:\t");
#17
#18          p = std::adjacent_difference(a, a + n, b, sub);
#19          print(b, p, "b:\t");
#20
#21          p = std::adjacent_difference(a, a + n, b, add);
#22          print(b, p, "b:\t");
#23
#24          p = std::adjacent_difference(a, a + n, b, mul);
#25          print(b, p, "b:\t");
#26      }
```

程序的输出结果如下：

```
a:  1   2   3   4   5
b:  1   1   1   1   1
b:  1   1   1   1   1
b:  1   3   5   7   9
```

b: 1 2 6 12 20

5. iota

iota 算法介绍详见表 2.1.115。

表 2.1.115 iota 算法

算法名称	iota
函数原型	template< class ForwardIt, class T > void iota(ForwardIt first, ForwardIt last, T value);
函数功能	用从 value 连续增加的数值填充区间[first, last)
函数参数	first 表示区间起点; last 表示区间终点; value 表示初始值
返回值	无

【例 2.1.51】iota 算法的应用示例。

```
#01      #include <algorithm>
#02      #include <iostream>
#03      #include <list>
#04      #include <numeric>
#05      #include <random>
#06      #include <vector>
#07
#08      int main() {
#09          std::list<int> l(10);
#10          std::iota(l.begin(), l.end(), 1);
#11
#12          std::vector<std::list<int>::iterator> v(l.size());
#13          std::iota(v.begin(), v.end(), l.begin());
#14
#15          std::shuffle(v.begin(), v.end(),
#16                      std::mt19937{ std::random_device{}() });
#17
#18          std::cout << "Contents of the list: ";
#19          for (auto n : l) std::cout << n << ' ';
#20          std::cout << '\n';
#21
#22          std::cout << "Contents of the list, shuffled: ";
#23          for (auto i : v) std::cout << *i << ' ';
#24          std::cout << '\n';
#25      }
```

1.4.8 迭代器相关算法

常用迭代器算法的应用形式及功能如表 2.1.116 所示，这类算法主要用于驱使迭代器移动、计算迭代器相对距离、交换迭代器元素等。

表 2.1.116 迭代器算法的应用形式及功能

算法的应用形式	算法的功能
advance(iter, n)	驱动迭代器 iter 移动 n 个元素的相对距离
distance(first, second)	计算迭代器 first 与 second 的相对距离（有正负）
iter_swap(first, second)	交换迭代器 first 与 second 所指元素的值
next(iter[, n])	获取迭代器 iter 的第 n 个后继；n 默认 1
prev(iter[, n])	获取迭代器 iter 的第 n 个前驱；n 默认 1

1. advance

advance 算法介绍详见表 2.1.117。

表 2.1.117 advance 算法

算法名称	advance
函数原型	template<class InputIterator, class Distance> void advance(InputIterator& iter, Distance n);
函数功能	驱动迭代器 iter 移动 n 个元素的相对距离
函数参数	iter 表示待移动的迭代器；n 表示偏移距离
返回值	无

【例 2.1.52】advance 算法的应用示例。

```
#01      #include <algorithm>
#02      #include "xr.hpp"
#03
#04      int main()
#05      {
#06          int a[] = {10, 20, 30, 40, 50, 60, 70, 80, 90};
#07          size_t n = sizeof(a) / sizeof(*a);
#08
#09          int* p = a + 3;          xr(a[p-a]);          //结果为 40
#10          std::advance(p, 4);      xr(a[p-a]);          //结果为 80
#11          std::advance(p, -5);      xr(a[p-a]);          //结果为 30
#12      }
```

2. distance

distance 算法介绍详见表 2.1.118。

表 2.1.118 distance 算法

算法名称	distance
函数原型	template<class InputIterator> typename iterator_traits<InputIterator>::difference_type distance(InputIterator first, InputIterator second);
函数功能	计算迭代器 first 与 second 之间的相对距离
函数参数	first 表示迭代器；second 表示迭代器
返回值	相对距离

【例 2.1.53】distance 算法的应用示例。

```

#01    #include <algorithm>
#02    #include "xr.hpp"
#03
#04    int main()
#05    {
#06        int a[10];
#07        int* p = a + 3;
#08        xr(std::distance(a, p));           //结果为 3
#09        xr(std::distance(p, a));           //结果为-3
#10
#11        std::advance(p, 4);
#12        xr(std::distance(a, p));           //结果为 7
#13
#14        std::advance(p, -5);
#15        xr(std::distance(a, p));           //结果为 2
#16    }

```

3. iter_swap

iter_swap 算法介绍详见表 2.1.119。

表 2.1.119 iter_swap 算法

算法名称	iter_swap
函数原型	<pre>template<class ForwardIterator1, class ForwardIterator2> void iter_swap(ForwardIterator1 first, ForwardIterator2 second);</pre>
函数功能	交换迭代器 first 与 second 所指元素的值
函数参数	first 表示迭代器; second 表示迭代器
返回值	无

【例 2.1.54】iter_swap 算法的应用示例。

```

#01    #include <algorithm>
#02    #include "xr.hpp"
#03
#04    int main()
#05    {
#06        int a[] = {10, 20, 30, 40, 50, 60, 70, 80, 90};
#07        size_t n = sizeof(a) / sizeof(*a);
#08
#09        std::iter_swap(a, a + 5);
#10        xr(*a); xr(*(a + 5));           //结果分别为 60, 10
#11    }

```

4. next / prev

next / prev 算法介绍详见表 2.1.120 和表 2.1.121。

表 2.1.120 next 算法

算法名称	next
函数原型	template< class BidirIt > BidirIt prev(BidirIt it, typename std::iterator_traits<BidirIt>::difference_type n = 1);
函数功能	获取迭代器 it 的第 n 个后继
函数参数	it 表示迭代器; n 表示需要前进的元素个数
返回值	迭代器 it 的第 n 个后继

表 2.1.121 prev 算法

算法名称	prev
函数原型	template< class BidirIt > BidirIt prev(BidirIt it, typename std::iterator_traits<BidirIt>::difference_type n = 1);
函数功能	获取迭代器 it 的第 n 个前驱
函数参数	it 表示迭代器; n 表示需要后退的元素个数
返回值	迭代器 it 的第 n 个前驱

【例 2.1.55】next / prev 算法的应用示例。

```
#01      #include <iostream>
#02      #include <vector>
#03
#04      int main() {
#05          std::vector<int> v{ 4, 5, 1, 6, 2, 3 };
#06
#07          auto it = v.begin();
#08          auto nx = std::next(it, 4);
#09          std::cout << *it << ' ' << *nx << '\n';           //4 2
#10          nx = std::prev(nx, 1);
#11          std::cout << ' ' << *nx << '\n';                 //6
#12
#13          it = v.end();
#14          nx = std::next(it, -4);
#15          std::cout << ' ' << *nx << '\n';                 //1
#16          nx = std::prev(nx, -2);
#17          std::cout << ' ' << *nx << '\n';                 //2
#18      }
```


第 2 章 STL 容器参考

2.1 string 类

1. 构造、赋值及输入、输出

string 类的构造、赋值及输入、输出等操作如表 2.2.1 所示。这些操作使 string 对象得以生成、具有明确内容、输出显示。

表 2.2.1 string 类的构造、赋值及输入、输出操作

表达式	功能
构造 string 对象	
string s	默认构造空对象 s
string s2(s1)	由 s1 复制构造 s2
string s2(s1, idx)	以 s1 中从 idx 开始的字符作为初始值，构造 s2
string s2(s1, idx, num)	以 s1 中从 idx 开始、最多 num 个字符作为初始值，构造 s2
string s(cstr)	以 C-string cstr 作为初始值，构造 s
string s(chars, len)	以字符数组 chars 的前 len 个字符作为初始值，构造 s
string s(num, c)	以 num 个字符 c 作为初始值，构造 s
string s(beg, end)	以区间[beg, end)中的字符作为初始值，构造 s
为 string 对象赋值（下列都返回当前对象）	
s2 = s1	把 s2 赋值为 s1
s2.assign(s1)	把 s2 赋值为 s1
s2.assign(s1, idx, num)	把“s1 中从 idx 开始的最多 num 个字符”赋值给 s2
s = cstr	把 s 赋值为 C-string cstr
s.assign(cstr)	把 s 赋值为 C-string cstr
s.assign(chars, len)	把 s 赋值为“字符数组 chars 的前 len 个字符”
s = c	把 s 赋值为字符 c
s.assign(num, c)	把 s 赋值为“num 个字符 c”
s2.swap(s1)	交换 s1 和 s2 的内容，可用于赋值
swap(s1, s2)	交换 s1 和 s2 的内容，可用于赋值
输入 string 对象	
cin >> s	从标准输入设备读入字符串 s
getline(cin, s)	从标准输入设备读入字符串 s
getline(cin, s, delim)	从标准输入设备读入整行字符到字符串 s，遇到分隔符 delim 时结束
输出 string 对象	
cout << s	向标准输出设备写入字符串 s

2. 大小与存取

string 类的大小与存取操作如表 2.2.2 所示，这些操作提供了访问 string 对象的大小及元素个数、存取字符元素的方法。

表 2.2.2 string 类的大小与存取操作

表达式	功能
大小及容量	
s.size()	返回 s 中字符的个数
s.length()	返回 s 的长度，对应于 C-string 函数 strlen
s.empty()	判断 s 是否为空，用于此目的时，比 size 和 length 快
s.max_size()	返回 s 最多能够包含的字符个数
s.capacity()	返回 s 的当前存储空间容量，即能容纳字符的最大数目
s.resize(num)	将 s 当前的字符个数调整为 num，若元素增多，则其值为 '\0'
s.resize(num, c)	将 s 当前的字符个数调整为 num，若元素增多，则其值为 c
s.reserve(n)	将 s 的存储空间容量预先保留为 n
存取元素	
string::npos	特殊标志，表示查找失败或所有剩余字符，值为-1
s.c_str()	以 C-string 形式返回 s 的内容，以 '\0' 结束
s.data()	以字符数组形式返回 s 的内容，无字符串结束符 '\0'
s.copy(buf, size)	把 s 的前 size 个字符复制到 buf 中，返回实际复制的字符数
s.copy(buf, size, idx)	把“s 中从 idx 开始的 size 个字符”复制到 buf 中，返回实际复制的字符数
s[idx]	返回 s 中下标为 idx 的字符，不检查下标的合法性
s.at(idx)	返回 s 中下标为 idx 的字符，进行下标合法性检查
string::iterator	迭代器类型，对 string 而言是 char*
s.begin()	返回 s 中首字符的位置
s.end()	返回 s 中末字符存放位置的后一个位置
s.rbegin()	返回 s 中逆向首字符的位置（即最后一个元素的位置）
s.rend()	返回 s 中逆向末字符位置的后一个位置（即首字符的前一个位置）

3. 插入与追加

string 类的插入与追加操作如表 2.2.3 所示，这些操作提供了在 string 对象中插入与追加子串的方法。

表 2.2.3 string 类的插入与追加操作

表达式	功能
字符串插入（后带*者，不返回当前对象，其余返回）	
s2.insert(idx, s1)	把 s1 插入 s2 的下标 idx 前
s2.insert(idx, s1, slidx, slnum)	把 s1 中从 slidx 开始的最多 slnum 个字符插入 s2 的 idx 前
s.insert(idx, cstr)	把 C-string cstr 插入 s 的 idx 位置前
s.insert(idx, chars, len)	把字符数组 chars 的前 len 个字符插入 s 的下标 idx 前

续表

表达式	功能
<code>s.insert(idx, num, c)</code>	把 num 个字符 c 插入 s 的下标 idx 前
<code>s.insert(pos, num, c)*</code>	把 num 个字符 c 插入 s 的位置 pos 前
<code>s.insert(pos, c)*</code>	把字符 c 插入 s 的位置 pos 前
<code>s.insert(pos, beg, end)*</code>	把区间[beg, end)中的所有字符插入 s 的位置 pos 前
字符串追加(下列都返回当前对象)	
<code>s2+=s1</code>	把字符串 s1 追加到 s2 的末端
<code>s2.append(s1)</code>	把字符串 s1 追加到 s2 的末端
<code>s2.append(s1, idx, num)</code>	把 s1 中从 idx 开始的最多 num 个字符追加到 s2 的末端
<code>s+=cstr</code>	把 C-string cstr 追加到字符串 s 的末端
<code>s.append(cstr)</code>	把 C-string cstr 追加到字符串 s 的末端
<code>s.append(chars, len)</code>	把字符数组 chars 的前 len 个字符追加到字符串 s 的末端
<code>s.append(num, c)</code>	把 num 个字符 c 追加到字符串 s 的末端
<code>s.push_back(c)</code>	把字符 c 追加到 s 的末端
<code>s+=c</code>	把字符 c 追加到字符串 s 的末端, 同 <code>s.push_back(c)</code>
<code>s.append(beg, end)</code>	把区间[beg, end)中的所有字符追加到 s 的末端

4. 截取与连接

string 类的截取与连接操作如表 2.2.4 所示, 这些操作提供了在 string 对象中截取子串、连接两个 string 对象的方法。

表 2.2.4 string 类的截取与连接操作

表达式	功能
截取子字符串	
<code>s.substr()</code>	返回 s 的副本, 即 s 的全部内容都作为子串
<code>s.substr(idx)</code>	返回 s 中从下标 idx 开始的子串的副本
<code>s.substr(idx, len)</code>	返回 s 中从下标 idx 开始的最多 len 个字符所组成子串的副本
字符串连接	
<code>s1+s2</code>	返回连接字符串 s1 和字符串 s2 后的字符串
<code>s+cstr</code>	返回连接字符串 s 和 C 字符串 cstr 后的字符串
<code>cstr+s</code>	返回连接 C 字符串 cstr 和字符串 s 后的字符串
<code>s+c</code>	返回连接字符串 s 和字符 c 后的字符串
<code>c+s</code>	返回连接字符 c 和字符串 s 后的字符串

5. 替换与删除

string 类的替换与删除操作如表 2.2.5 所示, 这些操作提供了替换与删除 string 对象中字符元素的方法。

表 2.2.5 string 类的替换与删除操作

表达式	功能
替换字符（下列都返回当前对象）	
s2.replace(idx, len, s1)	把 s2 中“从 idx 开始的最多 len 个字符”替换为 s1
s2.replace(beg, end, s1)	把 s2 中“位于区间[beg, end)的字符”替换为 s1
s2.replace(idx, len, s1, slidx, sllen)	把“s2 中从 idx 开始的最多 len 个字符”替换为“s1 中从 slidx 开始的最多 sllen 个字符”
s.replace(idx, len, cstr)	把“s 中从 idx 开始的最多 len 个字符”替换为 C-string cstr
s.replace(beg, end, cstr)	把“s 中位于区间[beg, end)的字符”替换为 C-string cstr
s.replace(idx, num, chars, len)	把“s 中从 idx 开始的最多 num 个字符”替换为“字符数组 chars 的前 len 个字符”
s.replace(beg, end, chars, len)	把“s 中位于区间[beg, end)的字符”替换为“字符数组 chars 的前 len 个字符”
s.replace(idx, len, num, c)	把“s 中从 idx 开始的最多 len 个字符”替换为“num 个字符 c”
s.replace(beg, end, num, c)	把“s 中位于区间[beg, end)的字符”替换为“num 个字符 c”
s.replace(beg, end, newBeg, newEnd)	把“s 中的区间[beg, end)”替换为“区间[newBeg, newEnd)”
删除字符	
s.clear()	删除 s 的所有字符，无返回值
s.erase()	删除 s 的所有字符，返回当前对象
s.erase(idx)	删除 s 中“从下标 idx 开始的所有字符”，返回当前对象
s.erase(idx, len)	删除 s 中“从下标 idx 开始的最多 len 个字符”，返回当前对象
s.erase(pos)	删除 s 中“位于 pos 的字符”，返回位于 pos 之后的字符
s.erase(beg, end)	删除 s 中“区间[beg, end)中的所有字符”，返回位于 end 的字符

6. 查找字符或字符串

string 类的查找操作如表 2.2.6 所示，这些操作提供了在 string 对象中查找子串或字符的各种方法。

表 2.2.6 string 类的查找操作

表达式	功能
查找字符（若查找失败，下列都返回 string::npos）	
s.find(c)	在 s 中正向查找字符 c，若成功则返回其在 s 中第一次出现时的下标
s.find(c, idx)	在 s 中从下标 idx 开始，正向查找字符 c，若成功则返回其在 s 中第一次出现时的下标
s.rfind(c)	在 s 中逆向查找字符 c，若成功则返回其在 s 中逆向第一次出现时的下标
s.rfind(c, idx)	在 s 中从下标 idx 开始，逆向查找字符 c，若成功则返回其在 s 中逆向第一次出现时的下标
查找子字符串（若查找失败，下列都返回 string::npos）	
s.find(sub)	在 s 中正向查找“子串 sub”，若成功则返回其在 s 中第一次出现时的下标
s.find(sub, idx)	从 s 的下标 idx 开始，正向查找“子串 sub”，若成功则返回其第一次在 s 中出现时的下标
s.rfind(sub)	在 s 中逆向查找“子串 sub”，若成功则返回其逆向第一次在 s 中出现时的下标
s.rfind(sub, idx)	从 s 的下标 idx 开始，逆向查找“子串 sub”，若成功则返回其逆向第一次在 s 中出现时的下标
s.find(cstr)	在 s 中正向查找“C-string cstr”，若成功则返回其第一次在 s 中出现时的下标

续表

表达式	功能
<code>s.find(cstr, idx)</code>	从 <code>s</code> 的下标 <code>idx</code> 开始, 正向查找 “C-string <code>cstr</code> ”, 若成功则返回其第一次在 <code>s</code> 中出现时的下标
<code>s.rfind(cstr)</code>	在 <code>s</code> 中逆向查找 “C-string <code>cstr</code> ”, 若成功则返回其逆向第一次在 <code>s</code> 中出现时的下标
<code>s.rfind(cstr, idx)</code>	从 <code>s</code> 的下标 <code>idx</code> 开始, 逆向查找 “C-string <code>cstr</code> ”, 若成功则返回其逆向第一次在 <code>s</code> 中出现时的下标
<code>s.find(chars, idx, len)</code>	从 <code>s</code> 的下标 <code>idx</code> 开始, 正向查找 “字符数组 <code>chars</code> 中长度为 <code>len</code> 的子字符串”, 若成功则返回其第一次在 <code>s</code> 中出现时的下标
<code>s.rfind(chars, idx, len)</code>	从 <code>s</code> 的下标 <code>idx</code> 开始, 逆向查找 “字符数组 <code>chars</code> 中长度为 <code>len</code> 的子字符串”, 若成功则返回其逆向第一次在 <code>s</code> 中出现时的下标
<code>s.find_first_of(sub)</code>	在 <code>s</code> 中查找第一个 “与 <code>sub</code> 中某字符相同的字符”, 若成功则返回其在 <code>s</code> 中的下标
<code>s.find_first_of(sub, idx)</code>	从 <code>s</code> 的下标 <code>idx</code> 开始, 查找第一个 “与 <code>sub</code> 中某字符相同的字符”, 若成功则返回其在 <code>s</code> 中的下标
<code>s.find_first_not_of(sub)</code>	在 <code>s</code> 中查找第一个 “与 <code>sub</code> 中所有字符都不同的字符”, 若成功则返回其在 <code>s</code> 中的下标
<code>s.find_first_not_of(sub, idx)</code>	从 <code>s</code> 的下标 <code>idx</code> 开始, 查找第一个 “与 <code>sub</code> 中所有字符都不相同的字符”, 若成功则返回其在 <code>s</code> 中的下标
<code>s.find_first_of(cstr)</code>	在 <code>s</code> 中查找第一个 “与 C-string <code>cstr</code> 中某字符相同的字符”, 若成功则返回其在 <code>s</code> 中的下标
<code>s.find_first_of(cstr, idx)</code>	从 <code>s</code> 的下标 <code>idx</code> 开始, 查找第一个 “与 C-string <code>cstr</code> 中某字符相同的字符”, 若成功则返回其在 <code>s</code> 中的下标
<code>s.find_first_not_of(cstr)</code>	在 <code>s</code> 中查找第一个 “与 C-string <code>cstr</code> 中所有字符都不同的字符”, 若成功则返回其在 <code>s</code> 中的下标
<code>s.find_first_not_of(cstr, idx)</code>	从 <code>s</code> 的下标 <code>idx</code> 开始, 查找第一个 “与 C-string <code>cstr</code> 中所有字符都不相同的字符”, 若成功则返回其在 <code>s</code> 中的下标
<code>s.find_first_of(chars, idx, len)</code>	从 <code>s</code> 的下标 <code>idx</code> 开始, 查找第一个 “与字符数组 <code>chars</code> 的前 <code>len</code> 个字符中某字符相同的字符”, 若成功则返回其在 <code>s</code> 中的下标
<code>s.find_first_not_of(chars, idx, len)</code>	从 <code>s</code> 的下标 <code>idx</code> 开始, 查找第一个 “与字符数组 <code>chars</code> 的前 <code>len</code> 个字符都不同的字符”, 若成功则返回其在 <code>s</code> 中的下标
<code>s.find_first_of(c)</code>	在 <code>s</code> 中查找第一个 “与字符 <code>c</code> 相同的字符”, 若成功则返回其在 <code>s</code> 中的下标
<code>s.find_first_of(c, idx)</code>	从 <code>s</code> 的下标 <code>idx</code> 开始, 查找第一个 “与字符 <code>c</code> 相同的字符”, 若成功则返回其在 <code>s</code> 中的下标
<code>s.find_first_not_of(c)</code>	在 <code>s</code> 中查找第一个 “与字符 <code>c</code> 不同的字符”, 若成功则返回其在 <code>s</code> 中的下标
<code>s.find_first_not_of(c, idx)</code>	从 <code>s</code> 的下标 <code>idx</code> 开始, 查找第一个 “与字符 <code>c</code> 不同的字符”, 若成功则返回其在 <code>s</code> 中的下标
<code>s.find_last_of(sub)</code>	在 <code>s</code> 中查找最后一个 “与 <code>sub</code> 中某字符相同的字符”, 若成功则返回其在 <code>s</code> 中的下标
<code>s.find_last_of(sub, idx)</code>	从 <code>s</code> 的下标 <code>idx</code> 开始, 查找最后一个 “与 <code>sub</code> 中某字符相同的字符”, 若成功则返回其在 <code>s</code> 中的下标
<code>s.find_last_not_of(sub)</code>	在 <code>s</code> 中查找最后一个 “与 <code>sub</code> 中所有字符都不同的字符”, 若成功则返回其在 <code>s</code> 中的下标
<code>s.find_last_not_of(sub, idx)</code>	从 <code>s</code> 的下标 <code>idx</code> 开始, 查找最后一个 “与 <code>sub</code> 中所有字符都不相同的字符”, 若成功则返回其在 <code>s</code> 中的下标
<code>s.find_last_of(cstr)</code>	在 <code>s</code> 中查找最后一个 “与 C-string <code>cstr</code> 中某字符相同的字符”, 若成功则返回其在 <code>s</code> 中的下标

续表

表达式	功能
s.find_last_of(cstr, idx)	从 s 的下标 idx 开始, 查找最后一个“与 C-string cstr 中某字符相同的字符”, 若成功则返回其在 s 中的下标
s.find_last_not_of(cstr)	在 s 中查找最后一个“与 C-string cstr 中所有字符都不同的字符”, 若成功则返回其在 s 中的下标
s.find_last_not_of(cstr, idx)	从 s 的下标 idx 开始, 查找最后一个“与 C-string cstr 中所有字符都不相同的字符”, 若成功则返回其在 s 中的下标
s.find_last_of(chars, idx, len)	从 s 的下标 idx 开始, 查找最后一个“与字符数组 chars 的前 len 个字符中某字符相同的字符”, 若成功则返回其在 s 中的下标
s.find_last_not_of(chars, idx, len)	从 s 的下标 idx 开始, 查找最后一个“与字符数组 chars 的前 len 个字符中所有字符都不同的字符”, 若成功则返回其在 s 中的下标
s.find_last_of(c)	在 s 中查找最后一个“与字符 c 相同的字符”, 若成功则返回其在 s 中的下标
s.find_last_of(c, idx)	从 s 的下标 idx 开始, 查找最后一个“与字符 c 相同的字符”, 若成功则返回其在 s 中的下标
s.find_last_not_of(c)	在 s 中查找最后一个“与字符 c 不同的字符”, 若成功则返回其在 s 中的下标
s.find_last_not_of(c, idx)	从 s 的下标 idx 开始, 查找最后一个“与字符 c 不同的字符”, 若成功则返回其在 s 中的下标

对于上述查找操作, 表 2.2.7 总结了各个查找函数的功能。

表 2.2.7 查找函数及其功能

string 查找函数	功能
find	查找第一个与模式相等的字符
rfind	查找最后一个与模式相等的字符
find_first_of	查找第一个“与模式中某字符相等”的字符
find_last_of	查找最后一个“与模式中某字符相等”的字符
find_first_not_of	查找第一个“与模式中所有字符都不相等”的字符
find_last_not_of	查找最后一个“与模式中所有字符都不相等”的字符

7. 大小与相等比较

string 类的关系运算如表 2.2.8 所示, 这些操作以运算符和函数的形式提供。

表 2.2.8 string 类的大小与相等比较操作

表达式	功能
关系运算符	
s2 </<=>/>==/!= s1	s2 与 s1 进行大小比较或相等比较
s </<=>/>==/!= cstr	s 与 C-string cstr 的比较
compare 函数: (返回 0 表示相等, 负数表示小于, 正数表示大于)	
s2.compare(s1)	比较 s2 和 s1 (注意顺序)
s2.compare(idx, len, s1)	比较 s2 中“从 idx 开始的最多 len 个字符”和 s1
s2.compare(idx, len, s1, s1idx, s1len)	比较“s2 中从 idx 开始的最多 len 个字符”和“s1 中从 s1idx 开始的最多 s1len 个字符”
s.compare(cstr)	比较 s 和 C-string cstr
s.compare(idx, len, cstr)	比较“s 中从 idx 开始的最多 len 个字符”和 C-string cstr
s.compare(idx, len, chars, chlen)	比较“s 中从 idx 开始的最多 len 个字符”和“字符数组 chars 的前 chlen 个字符”

8. 元素及迭代器类型

如同 STL 其他容器，string 类也具有元素类型及迭代器类型，如表 2.2.9 所示。

表 2.2.9 string 类的元素类型及迭代器类型

类型	含义
元素类型	
string::traits_type	字符特征类型，即 char_traits<char>
string::value_type	字符类型，即 char
string::pointer	字符指针类型，即 char*
string::const_pointer	字符 const 指针类型，即 const char*
string::reference	字符引用类型，即 char&
string::const_reference	字符 const 引用类型，即 const char&
string::size_type	string 中字符个数类型，无符号整型，常为 size_t
string::difference_type	string 中迭代器距离类型，有符号整型，常为 ptrdiff_t
迭代器类型	
string::iterator	迭代器类型，即 char*
string::const_iterator	const 迭代器类型，即 const char*
string::reverse_iterator	逆向迭代器类型
string::const_reverse_iterator	const 逆向迭代器类型

9. 字符类型函数

C 头文件<ctype.h>和 C++头文件<cctype>中声明了许多字符类型函数，如表 2.2.10 所示，这些函数提供了判断处理单个字符的方法。

表 2.2.10 字符类型函数

函数调用表达式	功能
isalnum(c)	判断 c 是否是字母或数字
isalpha(c)	判断 c 是否是字母
isctrl(c)	判断 c 是否是控制字符
isprint(c)	判断 c 是否是可打印字符
isgraph(c)	判断 c 是否是除空格外的可打印字符
ispunct(c)	判断 c 是否是标点符号
isspace(c)	判断 c 是否是空白字符
isdigit(c)	判断 c 是否是数字
isxdigit(c)	判断 c 是否是十六进制数字
islower(c)	判断 c 是否是小写字母
isupper(c)	判断 c 是否是大写字母
tolower(c)	把大写字母 c 转换成小写字母
toupper(c)	把小写字母 c 转换成大写字母

2.2 vector 类

1. 构造及赋值

vector 类的构造及赋值操作如表 2.2.11 所示，这些操作提供了构造 vector 容器、在 vector 容器间赋值的方法。

表 2.2.11 vector 类的构造及赋值操作

表达式	功能
构造 vector 容器	
vector<T> v	默认构造空的 vector
vector<T> v1(v2)	由 v2 复制构造 v1
vector<T> v(n)	构造有 n 个元素的 vector，元素的值为 T()
vector<T> v(n, val)	构造有 n 个元素的 vector，元素的值都为 val
vector<T> v(beg, end)	以区间[beg, end)中元素的值为初始值，构造 vector
为 vector 容器赋值	
v1 = v2	把 v1 赋值为 v2
v.assign(n, val)	把 v 赋值为 n 个 val
v.assign(beg, end)	把 v 赋值为区间[beg, end)中的元素值
v1.swap(v2)	交换 v1 和 v2 的值

2. 大小及存取

vector 类的大小及存取操作如表 2.2.12 所示，这些操作提供了访问 vector 容器大小、存取 vector 容器中元素的方法。

表 2.2.12 vector 类的大小及存取操作

表达式	功能
大小及容量	
v.size()	返回当前元素的个数
v.capacity()	返回当前存储空间的容量，即能容纳元素的最大数目
v.empty()	判断容器是否为空
v.resize(n)	将当前元素的个数调整为 n，若元素增多，则其值为 T()
v.resize(n, val)	将当前元素的个数调整为 n，若元素增多，则其值为 val
v.reserve(n)	将存储空间的容量预先保留为 n
存取元素	
v.begin()	返回首元素的位置
v.end()	返回最后一个元素存放位置的后一个位置
v.rbegin()	返回逆向首元素（即正向最后一个元素）的位置
v.rend()	返回逆向尾元素的后一个元素的位置（即正向首元素的前一个位置）
v.front()	返回首元素，不检查容器是否为空

续表

表达式	功能
v.back()	返回最后一个元素，不检查容器是否为空
v[index]	返回下标为 index 的元素，不检查下标的合法性
v.at(index)	返回下标为 index 的元素，进行下标合法性检查

3. 添加与删除

vector 类的添加与删除操作如表 2.2.13 所示，这些操作提供了向 vector 容器中插入元素、从 vector 容器中删除元素的方法。

表 2.2.13 vector 类的添加与删除操作

表达式	功能
添加元素	
v.push_back(val)	在容器尾部添加值为 val 的元素
v.insert(pos, val)	在 pos 的前一个位置插入值为 val 的元素，返回新元素的位置
v.insert(pos, n, val)	在 pos 的前一个位置插入 n 个值为 val 的元素
v.insert(pos, beg, end)	在 pos 的前一个位置插入区间[beg, end)中的元素
删除元素	
v.pop_back()	删除最后一个元素，不回传其值
v.erase(pos)	删除位置 pos 上的元素，返回下一个元素的位置
v.erase(beg, end)	删除区间[beg, end)中的元素，返回下一个元素的位置
v.clear()	清空容器，删除所有元素

4. 元素及迭代器类型

vector 类的元素类型及迭代器类型如表 2.2.14 所示。

表 2.2.14 vector 类的元素类型及迭代器类型

类型	含义
元素类型	
vector<T>::value_type	容器中元素的类型，即 T
vector<T>::pointer	T 类型的指针
vector<T>::const_pointer	T 类型的 const 指针
vector<T>::reference	T 类型的引用
vector<T>::const_reference	T 类型的 const 引用
vector<T>::size_type	容器中元素的个数类型，无符号整型，常为 size_t
vector<T>::difference_type	容器中两个迭代器距离类型，有符号整型，常为 ptrdiff_t
迭代器类型	
vector<T>::iterator	迭代器类型
vector<T>::const_iterator	const 迭代器类型
vector<T>::reverse_iterator	逆向迭代器类型
vector<T>::const_reverse_iterator	const 逆向迭代器类型

2.3 list 类

1. 构造及赋值

list 类的构造及赋值操作如表 2.2.15 所示，这些操作提供了构造 list 容器、在 list 容器间赋值的方法。

表 2.2.15 list 类的构造及赋值操作

表达式	功能
构造 list 容器	
list<T> l	默认构造空的 l
list<T> l2(l1)	由 l1 复制构造 l2
list<T> l(n)	构造有 n 个元素的链表 l，元素值都为 T()
list<T> l(n, elem)	构造有 n 个元素的链表 l，元素值都为 elem
list<T> l(beg, end)	以区间[beg, end)中的元素作为初始值，构造 l
为 list 容器赋值	
l2 = l1	把 l2 赋值为 l1
l.assign(n, elem)	把 l 赋值为“值为 elem 的 n 个元素”
l.assign(beg, end)	把 l 赋值为“区间[beg, end)中的元素”
l2.swap(l1)	交换 l1 和 l2 的内容，可用于赋值
swap(l1, l2)	交换 l1 和 l2 的内容，可用于赋值

2. 大小及存取

list 类的大小及存取操作如表 2.2.16 所示，这些操作提供了访问 list 容器大小、存取 list 容器中元素的方法。

表 2.2.16 list 类的大小及存取操作

表达式	功能
大小及容量	
l.size()	返回 l 中元素的个数
l.empty()	判断 l 是否为空，用于此目的时，比 size 更快
l.max_size()	返回 l 中最大可能的元素个数
l.resize(n)	把 l 中元素的个数调整为 n，若元素增多，则增加元素的值为 T()
l.resize(n, elem)	把 l 中元素的个数调整为 n，若元素增多，则增加元素的值为 elem
存取及访问	
l.front()	返回 l 的第一个元素，不检查链表是否为空
l.back()	返回 l 的最后一个元素，不检查链表是否为空
l.begin()	返回 l 中第一个元素的位置
l.end()	返回 l 中最后一个元素的位置的后一个位置
l.rbegin()	返回 l 中逆向第一个元素的位置
l.rend()	返回 l 中逆向最后一个元素的位置的后一个位置

3. 插入与删除

list 类的插入与删除操作如表 2.2.17 所示，这些操作提供了向 list 容器中插入元素、从 list 容器中删除元素的方法。

表 2.2.17 list 类的插入与删除操作

表达式	功能
插入元素	
l.insert(pos, elem)	在 l 的位置 pos 前插入元素 elem，并返回刚插入元素的位置
l.insert(pos, n, elem)	在 l 的位置 pos 前插入 n 个元素 elem，无返回值
l.insert(pos, beg, end)	把区间[beg, end)的元素插入 l 的位置 pos 前，无返回值
追加元素	
l.push_front(elem)	把元素 elem 追加到 l 的首端
l.push_back(elem)	把元素 elem 追加到 l 的末端
删除元素	
l.clear()	删除 l 的所有元素，无返回值
l.pop_front()	删除 l 的第一个元素，无返回值
l.pop_back()	删除 l 的最后一个元素，无返回值
l.remove(val)	删除 l 中所有值为 val 的元素
l.remove_if(pr)	删除 l 中使一元谓词 pr 为 true 的所有元素
l.erase(pos)	删除 l 中位置为 pos 的元素，并返回下一元素的位置
l.erase(beg, end)	删除 l 中区间[beg, end)的所有元素，并返回下一元素的位置
l.unique()	删除 l 中相邻、相等元素，保留其第一次出现
l.unique(pr)	删除 l 中相邻、且使二元谓词 pr 为 true 的元素对，保留第一个元素

4. 合并、逆转及排序

list 类的合并、逆转及排序操作如表 2.2.18 所示，这些操作提供了改变 list 容器中元素顺序的方法。

表 2.2.18 list 类的合并、逆转及排序操作

表达式	功能
合并链表	
l2.splice(pos, l1)	从 l1 中删除所有元素，把它们插入 l2 的位置 pos 前
l2.splice(pos, l1, pos1)	从 l1 中删除位于 pos1 的元素，把它插入 l2 的位置 pos 前
l2.splice(pos, l1, beg, end)	从 l1 中删除区间[beg, end)元素，把它们插入 l2 的位置 pos 前
l2.merge(l1)	从 l1 中删除所有元素，把它们插入 l2 中，然后对 l2 从小到大排序。注意 l1 和 l2 必须都是有序的
l2.merge(l1, comp)	功能同上述版本，只是排序准则为 comp
逆转链表	
l.reverse()	把链表 l 中的元素全部逆转
链表排序	
l.sort()	把链表 l 中的元素排序，默认从小到大
l.sort(comp)	把链表 l 中的元素排序，排序准则为 comp

5. 大小与相等比较

如表 2.2.19 所示，list 容器之间的大小与相等关系可以使用关系运算符进行判断。

表 2.2.19 list 类的大小与相等比较

表达式	功能
关系运算符	
<code>l2 </<=>/> >= /!= l1</code>	l2 与 l1 进行大小比较或相等比较

6. 元素及迭代器类型

list 类的元素类型及迭代器类型如表 2.2.20 所示。

表 2.2.20 list 类的元素类型及迭代器类型

类型	含义
元素类型	
<code>list<T>::value_type</code>	容器中元素的类型，即 T
<code>list<T>::pointer</code>	T 类型的指针
<code>list<T>::const_pointer</code>	T 类型的 const 指针
<code>list<T>::reference</code>	T 类型的引用
<code>list<T>::const_reference</code>	T 类型的 const 引用
<code>list<T>::size_type</code>	容器中元素的个数类型，无符号整型，常为 <code>size_t</code>
<code>list<T>::difference_type</code>	容器中两个迭代器距离类型，有符号整型，常为 <code>ptrdiff_t</code>
迭代器类型	
<code>list<T>::iterator</code>	迭代器类型
<code>list<T>::const_iterator</code>	const 迭代器类型
<code>list<T>::reverse_iterator</code>	逆向迭代器类型
<code>list<T>::const_reverse_iterator</code>	const 逆向迭代器类型

2.4 deque 类

1. 构造及赋值

deque 类的构造及赋值操作如表 2.2.21 所示，这些操作提供了构造 deque 容器、在 deque 容器之间赋值的方法。

表 2.2.21 deque 类的构造及赋值操作

表达式	功能
构造 deque 容器	
<code>deque<T> d</code>	默认构造空的 deque
<code>deque<T> d1(d2)</code>	由 d2 复制构造 d1
<code>deque<T> d(n)</code>	构造有 n 个元素的容器 d，元素的值都为 T()

续表

表达式	功能
deque<T> d(n, val)	构造有 n 个元素的容器 d，元素的值都为 val
deque<T> d(beg, end)	以区间[beg, end)中元素的值为初始值，构造容器 d
为 deque 容器赋值	
d1 = d2	把 d1 赋值为 d2
d.assign(n, val)	把 d 赋值为 n 个 val
d.assign(beg, end)	把 d 赋值为区间[beg, end)中的元素值
d1.swap(d2)	交换 d1 和 d2 的值
swap(d1, d2)	交换 d1 和 d2 的值

2. 大小及存取

deque 类的大小及存取操作如表 2.2.22 所示，这些操作提供了访问 deque 容器元素个数、存取 deque 容器元素的方法。

表 2.2.22 deque 类的大小及存取操作

表达式	功能
大小及容量	
d.size()	返回容器中当前元素的个数
d.max_size()	返回当前存储空间能容纳元素的最大数目
d.empty()	判断容器是否为空
d.resize(n)	将当前元素的个数调整为 n，若元素增多，则其值为 T()
d.resize(n, val)	将当前元素的个数调整为 n，若元素增多，则其值为 val
存取元素	
d.begin()	返回首元素的位置
d.rbegin()	返回逆向首元素的位置
d.end()	返回最后一个元素存放位置的后一个位置
d.rend()	返回逆向向最后一个元素存放位置的后一个位置
d.front()	返回首元素，不检查容器是否为空
d.back()	返回最后一个元素，不检查容器是否为空
d[index]	返回下标为 index 的元素，不检查下标的合法性
d.at(index)	返回下标为 index 的元素，进行下标合法性检查
大小关系及相等性	
d1 < /> = / != d2	比较两个同类型容器 d1、d2 的大小关系和相等性

3. 插入与删除

deque 类的插入与删除操作如表 2.2.23 所示，这些操作提供了向 deque 容器中插入元素、从 deque 容器中删除元素的方法。

表 2.2.23 deque 类的插入与删除操作

表达式	功能
插入元素	
d.push_front(val)	在容器首部添加值为 val 的元素
d.push_back(val)	在容器尾部添加值为 val 的元素
d.insert(pos, val)	在 pos 之前插入值为 val 的元素，返回新元素的位置
d.insert(pos, n, val)	在 pos 之前插入 n 个值为 val 的元素，无返回值
d.insert(pos, beg, end)	在 pos 之前插入区间[beg, end)中的元素，无返回值
删除元素	
d.pop_front()	删除第一个元素，不回传其值
d.pop_back()	删除最后一个元素，不回传其值
d.erase(pos)	删除位置 pos 上的元素，返回下一个元素的位置
d.erase(beg, end)	删除区间[beg, end)中的元素，返回下一个元素的位置
d.clear()	清空容器，删除所有元素

4. 元素及迭代器类型

deque 类的元素类型及迭代器类型如表 2.2.24 所示。

表 2.2.24 deque 类的元素类型及迭代器类型

表达式	功能
元素类型	
deque<T>::value_type	容器中元素的类型，即 T
deque<T>::pointer	T 类型的指针
deque<T>::const_pointer	T 类型的 const 指针
deque<T>::reference	T 类型的引用
deque<T>::const_reference	T 类型的 const 引用
deque<T>::size_type	容器中元素的个数类型，无符号整型，常为 size_t
deque<T>::difference_type	容器中两个迭代器距离类型，有符号整型，常为 ptrdiff_t
迭代器类型	
deque<T>::iterator	迭代器类型
deque<T>::const_iterator	const 迭代器类型
deque<T>::reverse_iterator	逆向迭代器类型
deque<T>::const_reverse_iterator	const 逆向迭代器类型

2.5 set/multiset 类

1. 构造及赋值

set/multiset 类的构造及赋值操作如表 2.2.25 所示，这些操作提供了构造 set/multiset 容器、在 set/multiset 容器之间赋值的方法。

表 2.2.25 set/multiset 类的构造及赋值操作

表达式	功能
构造 set/multiset 容器	
set s	默认构造空的 set/multiset
set s(op)	以 op 为比较准则，构造空的 set/multiset
set s2(s1)	由 s1 复制构造 s2
set s(beg, end)	以区间[beg, end)中元素的值为初始值，构造 s
set s(beg, end, op)	以 op 为比较准则、区间[beg, end)中的元素为初始值，构造 s
上述 set 可为下列类型之一	
set<T>	set 容器类型，默认以 operator < 作为比较准则
set<T, op>	set 容器类型，以 op 作为比较准则
multiset<T>	multiset 容器类型，默认以 operator < 作为比较准则
multiset<T, op>	multiset 容器类型，以 op 作为比较准则
为 set/multiset 容器赋值	
s1=s2	把 s1 赋值为 s2
s1.swap(s2)	交换 s1 和 s2 的值
swap(s1, s2)	交换 s1 和 s2 的值

2. 大小及存取

set/multiset 类的大小及存取、比较操作如表 2.2.26 所示，这些操作提供了访问 set/multiset 容器中元素个数、存取 set/multiset 容器元素、比较 set/multiset 容器对象的大小及相等性的方法。

表 2.2.26 set/multiset 类的大小及存取、比较操作

表达式	功能
大小及容量	
s.size()	返回容器中当前元素的个数
s.max_size()	返回当前存储空间能容纳元素的最大数目
s.empty()	判断容器是否为空
存取元素	
s.begin()	返回首元素的位置
s.rbegin()	返回逆向首元素的位置
s.end()	返回最后一个元素存放位置的后一个位置
s.rend()	返回逆向最后一个元素存放位置的后一个位置
s.key_comp()	返回键值 key 比较的函数对象（类型为 key_compare）
s.value_comp()	返回值 value 比较的函数对象（类型为 value_compare）
大小关系及相等性	
s1 </> s2	比较两个同类型容器 s1、s2 的大小关系和相等性

3. 查找、插入与删除

set/multiset 类的查找、插入与删除操作如表 2.2.27 所示, 这些操作提供了在 set/multiset

容器中查找元素、向 set/multiset 容器中插入元素、从 set/multiset 容器中删除元素的方法。

表 2.2.27 set/multiset 类的查找、插入与删除操作

表达式	功能
查找元素	
s.count(k)	返回键值 key 为 k 的元素个数，对 set 来说非 0 即 1
s.find(k)	返回键值 key 为 k 的首元素位置
s.lower_bound(k)	返回键值 key 大于等于 k 的第一个元素位置
s.upper_bound(k)	返回键值 key 大于 k 的第一个元素位置
s.equal_range(k)	以 pair 返回键值 key 等于 k 的所有元素组成区间的起点和终点
插入元素	
s.insert(val)	插入 value 值为 val 的元素，set 以 pair 返回该元素位置和插入是否成功的状态；multiset 仅返回所插入值的位置
s.insert(pos, val)	插入 value 值为 val 的元素，返回新元素的位置，pos 表示提示位置
s.insert(beg, end)	插入区间[beg, end)中的元素，无返回值
删除元素	
s.erase(k)	删除 key 值为 k 的元素，返回删除元素个数，对 set 来说非 0 即 1
s.erase(pos)	删除位置 pos 上的元素，无返回值
s.erase(beg, end)	删除区间[beg, end)中的所有元素，无返回值
s.clear()	清空容器，删除所有元素

4. 元素及迭代器类型

set/multiset 类的元素类型及迭代器类型如表 2.2.28 所示。

表 2.2.28 set/multiset 类的元素类型及迭代器类型

类型	含义
元素类型（set 表示容器类型 set 或者 multiset）	
set<T>::value_type	容器中元素值 value 的类型，即 T
set<T>::key_type	容器中元素键值 key 的类型，即 T
set<T>::key_compare	比较键值 key 的函数对象
set<T>::value_compare	比较元素值 value 的函数对象
set<T>::pointer	T 类型的指针
set<T>::const_pointer	T 类型的 const 指针
set<T>::reference	T 类型的引用
set<T>::const_reference	T 类型的 const 引用
set<T>::size_type	容器中元素的个数类型，无符号整型，常为 size_t
set<T>::difference_type	容器中两个迭代器距离类型，有符号整型，常为 ptrdiff_t
迭代器类型（set 表示容器类型 set 或者 multiset）	
set<T>::iterator	迭代器类型
set<T>::const_iterator	const 迭代器类型
set<T>::reverse_iterator	逆向迭代器类型
set<T>::const_reverse_iterator	const 逆向迭代器类型

2.6 map/multimap 类

1. 构造及赋值

map/multimap 类的构造及赋值操作如表 2.2.29 所示, 这些操作提供了构造 map/multimap 容器、在 map/multimap 容器之间赋值的方法。

表 2.2.29 map/multimap 类的构造及赋值操作

表达式	功能
构造 map/multimap 容器	
map m	默认构造空的 map/multimap
map m(op)	以 op 为比较准则, 构造空的 map/multimap
map m2(m1)	由 m1 复制构造 m2
map m(beg, end)	以区间[beg, end)中元素的值为初始值, 构造 m
map m(beg, end, op)	以 op 为比较准则、区间[beg, end)元素为初始值, 构造 m
上述 map 可为下列类型之一	
map<Key, T>	map 容器类型, 默认以 operator < 作为比较准则
map<Key, T, op>	map 容器类型, 以 op 作为比较准则
multimap<Key, T>	multimap 容器类型, 默认以 operator < 作为比较准则
multimap<Key, T, op>	multimap 容器类型, 以 op 作为比较准则
为 map/multimap 容器赋值	
m1 = m2	把 m1 赋值为 m2
m1.swap(m2)	交换 m1 和 m2 的值
swap(m1, m2)	交换 m1 和 m2 的值

2. 大小及存取

map/multimap 类的大小及存取、比较操作如表 2.2.30 所示, 这些操作提供了访问 map/multimap 容器中的元素个数、访问 map/multimap 容器元素、比较 map/multimap 容器大小及相等性的方法。

表 2.2.30 map/multimap 类的大小及存取、比较操作

表达式	功能
大小及容量	
m.size()	返回容器中当前元素的个数
m.max_size()	返回当前存储空间能容纳元素的最大数目
m.empty()	判断容器是否为空
存取元素	
m.begin()	返回首元素的位置
m.rbegin()	返回逆向首元素的位置
m.end()	返回最后一个元素存放位置的后一个位置

续表

表达式	功能
m.rend()	返回逆向最后一个元素存放位置的后一个位置
m.key_comp()	返回键值 key 比较的函数对象（类型为 key_compare）
m.value_comp()	返回值 value 比较的函数对象（类型为 value_compare）
m[k]	返回键值 k 所关联 value 的引用（即 mapped_type&）。若 k 值未出现于容器 m 中，则该操作会把 mapped_type 类的默认对象插入容器中。multimap 未定义此操作
大小关系及相等性	
m1 <=>/>= m2	比较两个同类型容器 m1、m2 的大小关系和相等性

3. 查找、插入与删除

map/multimap 类的查找、插入与删除操作如表 2.2.31 所示，这些操作提供了在 map/multimap 容器中查找元素、向 map/multimap 容器中插入元素、从 map/multimap 容器中删除元素的方法。

表 2.2.31 map/multimap 类的查找、插入与删除操作

表达式	功能
查找元素	
m.count(k)	返回键值 key 为 k 的元素个数，对 map 来说非 0 即 1
m.find(k)	返回键值 key 为 k 的首元素位置
m.lower_bound(k)	返回键值 key 大于等于 k 的第一个元素位置
m.upper_bound(k)	返回键值 key 大于 k 的第一个元素位置
m.equal_range(k)	以 pair 返回键值 key 等于 k 的所有元素组成区间的起点和终点
插入元素	
m.insert(val)	插入 value 值为 val 的元素，map 以 pair 返回该元素位置和插入是否成功的状态；multimap 仅返回所插入值的位置
m.insert(pos, val)	插入 value 值为 val 的元素，返回新元素的位置，pos 表示提示位置
m.insert(beg, end)	插入区间[beg, end)中的元素，无返回值
删除元素	
m.erase(k)	删除 key 值为 k 的元素，返回删除元素个数，对 map 来说非 0 即 1
m.erase(pos)	删除位置 pos 上的元素，无返回值
m.erase(beg, end)	删除区间[beg, end)中的所有元素，无返回值
m.clear()	清空容器，删除所有元素

4. 元素及迭代器类型

map/multimap 类的元素类型及迭代器类型如表 2.2.32 所示。

表 2.2.32 map/multimap 类的元素类型及迭代器类型

类型	含义
元素类型 (map 表示容器类型 map 或 multimap)	
map<Key, T>::key_type	容器中元素键值 key 的类型, 即 Key
map<Key, T>::mapped_type	与键值 key 相关联的元素值 value 的类型, 即 T
map<Key, T>::value_type	容器中元素的类型, 即 pair<const Key, T>
map<Key, T, OP>::key_compare	比较键值 key 的函数对象 op
map<Key, T>::value_compare	比较元素值 value 的函数对象
map<Key, T>::pointer	value_type 类型的指针
map<Key, T>::const_pointer	value_type 类型的 const 指针
map<Key, T>::reference	value_type 类型的引用
map<Key, T>::const_reference	value_type 类型的 const 引用
map<Key, T>::size_type	容器中元素的个数类型, 无符号整型, 常为 size_t
map<Key, T>::difference_type	容器中两个迭代器距离类型, 有符号整型, 常为 ptrdiff_t
迭代器类型 (map 表示容器类型 map 或 multimap)	
map<Key, T>::iterator	迭代器类型
map<Key, T>::const_iterator	const 迭代器类型
map<Key, T>::reverse_iterator	逆向迭代器类型
map<Key, T>::const_reverse_iterator	const 逆向迭代器类型

参考文献

- Ivor Horton, Peter Van Weert, 2019. C++17 入门经典[M]. 卢旭红, 张骏温, 译. 5 版. 北京: 清华大学出版社.
- Marius Bancila, 2020. Modern C++ Programming Cookbook[M]. 2nd ed. Birmingham: Packet Publishing Ltd..
- Matthew H. Austern, 2003. 泛型编程与 STL[M]. 侯捷, 译. 北京: 中国电力出版社.
- Michael Wong, IBM XL 编译器中国开发团队, 2013. 深入理解 C++11: C++11 新特性解析与应用[M]. 北京: 机械工业出版社.
- Nicolai M. Josuttis, 2015. C++标准库[M]. 侯捷, 译. 2 版. 北京: 电子工业出版社.
- Nicolai M. Josuttis, 2019. C++17: The Complete Guide[M]. Victoria: Leanpub.
- P.J. Plauger, Alexander A. Stepanov, Meng Lee, et al., 2002. C++ STL 中文版[M]. 王昕, 译. 北京: 中国电力出版社.
- Paul Deitel, Harvey Deitel, 2014. C++ how to Programmer [M]. 10th ed. New York: Pearson Education Inc..
- Stephan Roth, 2021. Clean C++ 20: Sustainable Software Development Patterns and Best Practices[M]. 2nd ed. Berkeley: Apress.

附录

附录 1 宏 xr 的功能及实现

头文件"xr.hpp"中主要定义了两个宏，它们能够起到表达式定位、以字符串形式输出表达式、对表达式求值并输出等主要功能。如附表 1 所示，宏 xr 和 xrv 主要用于分析表达式及其值。

附表 1 宏 xr 的应用及功能

宏调用表达式	功能
xr(expr)	可用于能够产生值的任意表达式，依次输出表达式所在行号、表达式、表达式的值
xrv(funcall_expr)	主要用于无返回值的函数调用表达式，依次输出表达式所在行号、表达式

若在文件包含指令#include "xr.hpp"前面定义宏 NDEBUG，则有：

(1) 宏 xr 仅对表达式 expr 求值，不输出所在行号、不输出表达式、不输出表达式的值。

(2) 宏 xrv 仅执行函数调用，不输出所在行号、不输出表达式。

头文件"xr.hpp"的内容如下：

```
#01      #ifndef EXPRESSION_RESULT
#02      #define EXPRESSION_RESULT
#03
#04      #include <iostream>
#05      #include <iomanip>
#06
#07      namespace xr {
#08
#09          class line_number {                                //输出行号的效用算子
#10          private:
#11              int num;                                       //行号
#12              int width;                                    //所占宽度
#13          public:
#14              line_number(int n, int w):num(n), width(w) {}
#15              //输出行号:首先输出前导符#,接着输出用 0 填充的指定宽度的行号
#16              friend std::ostream& operator << (std::ostream&os,
#17              const line_number& rhs){
#18                  os << "#" << std::setw(rhs.width) << std::setfill('0')
#19                  << rhs.num << std::setfill(' ');
#20                  return os;
#21              }
#22      };
```

```

#23     } // namespace
#24
#25     #ifndef NDEBUG                                //不定义此符号,则起作用
#26
#27     #define xr(expr) std::cout << xr::line_number(__LINE__, 2) \
#28         << ": " << #expr << "\t==>";                                \
#29         std::cout << std::boolalpha << (expr)                                \
#30         << std::noboolalpha << std::endl;
#31
#32     #define xrv(expr) std::cout << xr::line_number(__LINE__, 2) \
#33         << ": " << #expr << "\t==>";                                \
#34         expr
#35
#36     #else // NDEBUG                                //若定义此符号,则不起作用
#37
#38     #define xr(expr) (expr)
#39     #define xrv(expr) expr
#40
#41     #endif // NDEBUG
#42
#43     #endif // EXPRESSION_RESULT

```

在上述程序实现过程中,主要用到 C++ I/O 流中自定义流操作算子(请参考课本具体例子),以及 C++ 中预定义的宏 `__LINE__` (实现定位的最大功臣)。

附录 2 函数 `print()` 的功能及实现

头文件 `"print.hpp"` 中重载定义了两个全局函数 `print()`, 它们分别输出区间、容器元素, 并允许同时输出提示信息。

第一个函数模板 `print()` 主要用于在屏幕上输出区间 `[beg, end)` 元素值, 参数 `msg` 表示在输出元素值的同时给出的提示信息, 参数 `delim` 表示输出元素值之间的间隔。基于第一个函数模板 `print()`, 第二个函数模板 `print()` 主要用于在屏幕上输出容器 `c` 中的元素值。它们的实现过程如下所示。

```

#01     #ifndef PRINT_STL_RANGE_CONTAINER
#02     #define PRINT_STL_RANGE_CONTAINER
#03
#04     #include <iostream>
#05     #include <algorithm>
#06     #include <string_view>
#07
#08     template <class Iterator> //迭代器类型
#09     void print(Iterator beg, Iterator end, //待输出区间[beg,end)
#10         std::string_view msg = "", //提示信息
#11         std::string_view delim = "\t") //元素之间的分隔符
#12     {
#13         using val_type
#14         = typename std::iterator_traits<Iterator>::value_type;

```

#29

附录3 宏 verify 的功能及实现

输出流对象 ofs、流对象 fs 是否成功打开并关联到磁盘文件。宏 verify 对它们进行封装, 并提供了统一的调用形式。在应用时, 只需要把待检验的文件流对象 fs 作为宏 verify 的参数即可。它们的实现过程如下所示。

#24

```
#25         bool verify_file(std::fstream& fs) {    //校验文件流对象
#26             if (!fs) {                          //若文件操作失败
#27                 std::cout<<"Error: failed to open file!";//输出提示信息
#28                 return false;                    //返回 false 表示失败
#29             }
#30             return true;                          //返回 true 表示成功
#31         }
#32     }
#33
#34     #define verify(fs) assert(xr::verify_file(fs))//定义校验宏进行分发
#35
#36     #endif //VERIFY_FILE
```


(TP-8926. 0101)



现代C++ 面向对象程序设计 实验指导

扫一扫



科学出版社 技术分社
<http://www.abook.cn>

www.sciencep.com

ISBN 978-7-03-070997-4



9 787030 709974 >

定价: 43.00 元

[General Information]

书名=现代C++面向对象程序设计实验指导

页数=227

SS号=15116566