

# C++

## 从入门到精通

**14**小时语音视频讲解

明日科技 编著



✓ 实例资源库    ✓ 模块资源库    ✓ 项目资源库  
✓ 面试资源库    ✓ 测试题库系统    ✓ PPT电子课件

(第2版)

循序渐进, 实战讲述

基础知识 ⇨ 核心技术 ⇨ 高级应用 ⇨ 项目实战

166个应用实例, 21个典型应用, 4个项目案例(光盘含3个)

海量资源, 可查可练

除本书配套的14小时视频讲解外, 根据学习顺序, 光盘还额外配备如下海量开发资源库:

实例资源库(881个实例) ⇨ 模块资源库(15个典型模块) ⇨  
项目资源库(15个项目案例) ⇨ 测试题库系统(616道测试题)  
⇨ 面试资源库(371个面试真题)

在线解答, 高效学习

QQ: 400 675 1066(可容纳10万人在线)

官方网站: [www.mingribook.com](http://www.mingribook.com)





软件开发视频大讲堂

# C++从入门到精通

(第2版)

(累计第5次印刷, 总印数16500册)

明日科技 编著

清华大学出版社

北 京

## 内 容 简 介

《C++从入门到精通(第2版)》从初学者角度出发,以通俗易懂的语言,丰富多彩的实例,详细讲解了C++语言的基础知识。全书共分18章,包括绪论,数据类型,表达式与语句,条件判断语句,循环语句,函数,数组、指针和引用,构造数据类型,面向对象编程,类和对象,继承与派生,模板,STL标准模板库,RTTI与异常处理,程序调试,文件操作,网络通信,图书管理系统。书中所有知识都结合具体实例进行介绍,涉及的程序代码给出了详细的注释,可以使读者轻松领会C++语言的强大,快速提高开发技能。另外,本书除了纸质内容之外,配书光盘中还给出了海量开发资源库,主要内容如下:

- |   |   |
|---|---|
| <input checked="" type="checkbox"/> 语音视频讲解:总时长14小时,共94段   | <input checked="" type="checkbox"/> 实例资源库:881个实例及源码详细分析     |
| <input checked="" type="checkbox"/> 模块资源库:15个经典模块开发过程完整展现 | <input checked="" type="checkbox"/> 项目案例资源库:15个企业项目开发过程完整展现 |
| <input checked="" type="checkbox"/> 测试题库系统:616道能力测试题目     | <input checked="" type="checkbox"/> 面试资源库:371个企业面试真题        |
| <input checked="" type="checkbox"/> PPT电子教案               |   |

本书适合作为软件开发入门者的自学用书,也适合作为高等院校相关专业的教学参考书,也可供开发人员查阅、参考。

本书封面贴有清华大学出版社防伪标签,无标签者不得销售。

版权所有,侵权必究。侵权举报电话:010-62782989 13701121933

图书在版编目(CIP)数据

C++从入门到精通/明日科技编著. —2版. —北京:清华大学出版社,2012.9  
(软件开发视频大讲堂)

ISBN 978-7-302-28847-3

I. ①C… II. ①明… III. ①C语言-程序设计 IV. ①TP312

中国版本图书馆CIP数据核字(2012)第103908号

责任编辑:赵洛育

封面设计:刘洪利

版式设计:文森时代

责任校对:柴燕

责任印制:沈露

出版发行:清华大学出版社

网 址: <http://www.tup.com.cn>, <http://www.wqbook.com>

地 址:北京清华大学学研大厦A座 邮 编:100084

社总机:010-62770175 邮 购:010-62786544

投稿与读者服务:010-62776969, [c-service@tup.tsinghua.edu.cn](mailto:c-service@tup.tsinghua.edu.cn)

质量反馈:010-62772015, [zhiliang@tup.tsinghua.edu.cn](mailto:zhiliang@tup.tsinghua.edu.cn)

印装者:北京鑫海金澳胶印有限公司

经 销:全国新华书店

开 本:203mm×260mm 印 张:26.5 字 数:711千字  
(附海量开发资源库DVD1张)

版 次:2010年7月第1版 2012年9月第2版 印 次:2012年9月第1次印刷

印 数:1~6500

定 价:59.80元

产品编号:046166-01



# 如何使用本书开发资源库

在学习《C++从入门到精通（第2版）》一书时，配合随书光盘提供了“Visual C++开发资源库”系统，可以帮助读者快速提升编程水平和解决实际问题的能力。《C++从入门到精通（第2版）》和Visual C++开发资源库配合学习流程如图1所示。

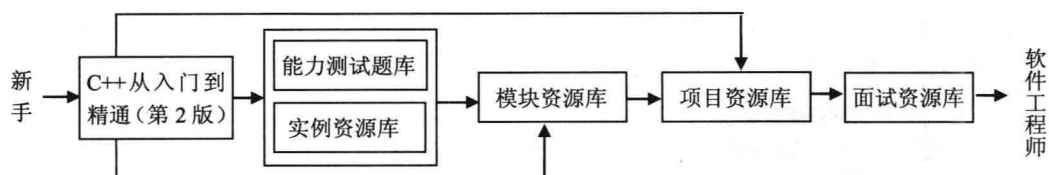


图1 从入门到精通与开发资源库配合学习流程图

打开光盘的“Visual C++开发资源库”文件夹，运行 Visual C++开发资源库.exe 程序，即可进入“Visual C++开发资源库”系统，界面如图2所示。

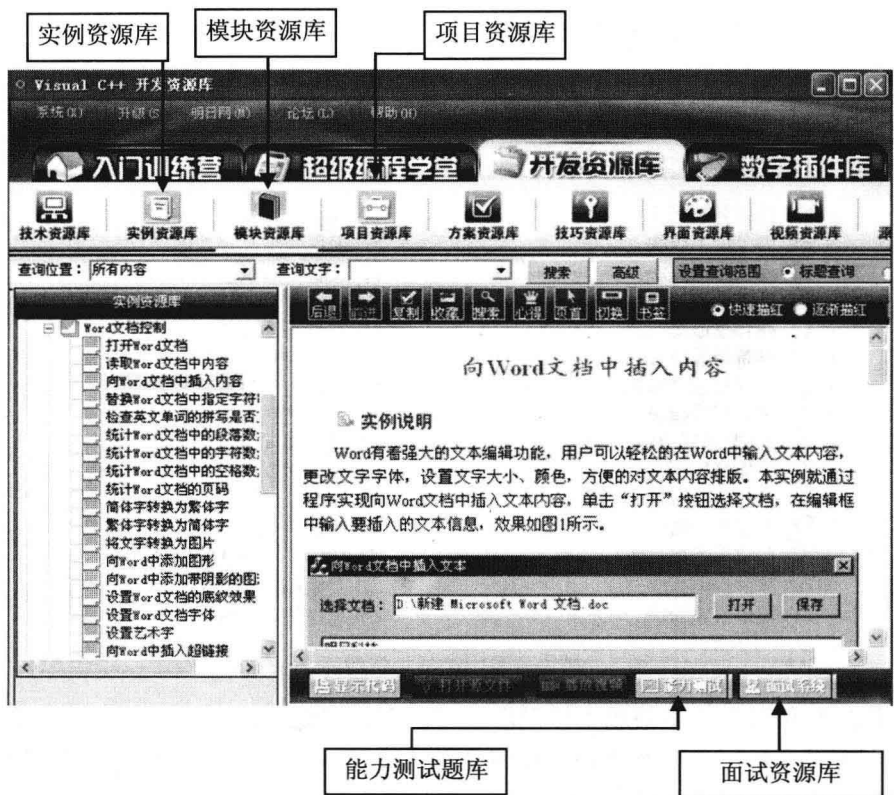


图2 Visual C++开发资源库主界面

在学习《C++从入门到精通（第2版）》某一章节时，可以配合实例资源库的相应章节，利用实例资源库提供的大量热点实例和关键实例巩固所学编程技能，提高编程兴趣和自信心。也可以配合能力测试题库的对应章节进行测试，检验学习成果，具体流程如图3所示。

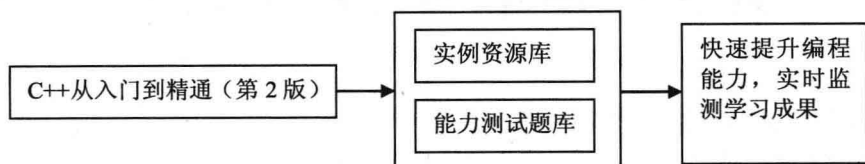


图3 使用实例资源库和能力测试题库

对于数学逻辑能力和英语基础较为薄弱的读者，或者想了解个人数学逻辑思维能力 and 编程英语基础的用户，本书提供了数学及逻辑思维能力测试和编程英语能力测试供练习和测试，如图4所示。

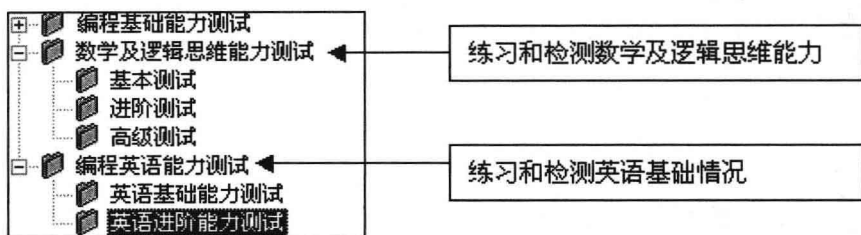


图4 数学及逻辑思维能力测试和编程英语能力测试目录

当《C++从入门到精通（第2版）》学习完成时，可以配合模块资源库和项目资源库的30个模块和项目，全面提升个人综合编程技能和解决实际开发问题的能力，为成为C++软件开发工程师打下坚实基础。具体模块和项目目录如图5所示。

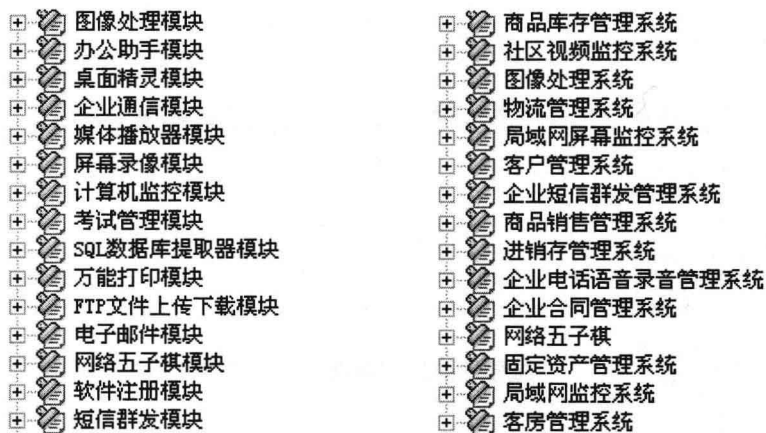


图5 模块资源库和项目资源库目录

万事俱备，该到软件开发的主战场上接受洗礼了。面试资源库提供了大量国内外软件企业的常见面试真题，同时还提供了程序员职业规划、程序员面试技巧、企业面试真题汇编和虚拟面试系统等精彩内容，是程序员求职面试的绝佳指南。面试资源库具体内容如图6所示。



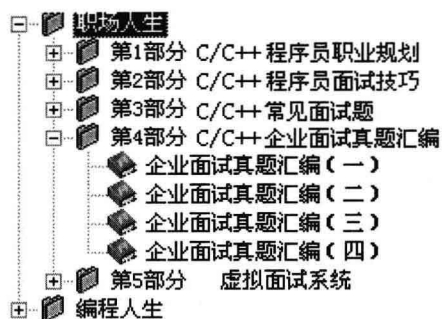


图6 面试资源库具体内容

如果您在使用本书开发资源库时遇到问题，读者朋友可加我们的 QQ：4006751066（可容纳 10 万人），我们将竭诚为您服务。

# 前言

## Preface

**丛书说明：**“软件开发视频大讲堂”（第1版）2008年出版以来，因为首次全程配备视频，编写细腻，易学实用，在计算机图书市场上产生了强烈反响，多个品种被评为“全国优秀畅销书”，

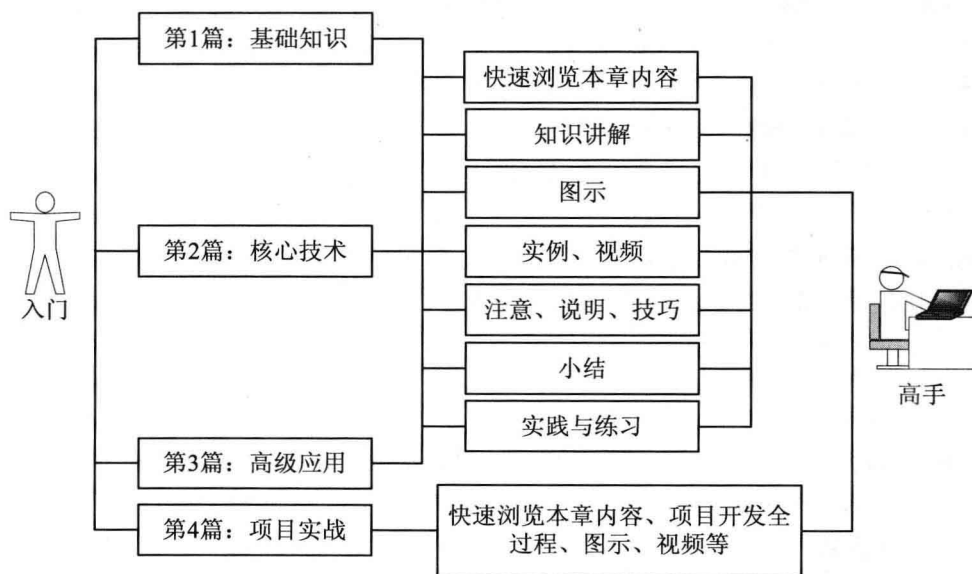
2010年7月改版以后，直到现在，在全国计算机零售图书排行榜的软件开发类排行中，持续名列前茅。丛书累计销售近40万册，被百余所高校计算机相关专业、软件学院选为教材，在众多的软件开发类零售图书中成为一支最耀眼的品牌。

第3版着重在前两版的基础上，修改原有的疏漏，大部分重新录制了视频，提供了从入门学习，到实例应用，到模块开发，到项目开发，到能力测试，直到面试等各个阶段的海量资源库。为了方便教学，还提供了教学课件PPT，读者可登录清华大学出版社网站直接下载。

C++语言是在C语言基础上发展起来的，它在C语言基础上融入了许多新的编程理念，这些理念有利于程序的开发。从语言角度来说，C++语言是个规范，它规范程序员如何进行面向对象程序开发。C++具有C语言操作底层的能力，同时还具有提高代码复用率的面向对象编程技术，是一种语句更加灵活、使用更加简捷、技术更加全面的编程利器。

## 本书内容

本书提供了从入门到编程高手所必备的各类知识，共分4篇，大体结构如下图所示。



**第1篇：基础知识。**本篇讲解C++语言基础部分，只有具备了牢固的基础知识才能更快地掌握更



高级的技术内容。通过对 C++语言的历史和特性、选择 C++语言的开发环境、算法、C++语言的数据类型、运算符与表达式、常用的数据输入/输出函数、选择结构程序设计和循环控制这些内容的介绍，结合流程图和实例，并通过视频的指导讲解，为以后编程奠定坚实的基础。

**第2篇：核心技术。**本篇介绍了 C++语言的关于面向对象方面的内容，理解面向对象这个概念，应用类类型创建对象，掌握什么是继承和派生，利用多态进行面向对象开发。

**第3篇：高级应用。**模板是 STL 的基础，通过对模板的介绍，使读者能够理解 STL 的构造。文件操作也是程序开发过程中必不可少的技术，掌握文件操作是奠定开发大项目的基础，通过对 RTTI 的介绍使读者对面向对象开发有更深入的理解。网络通信是仅次于文件技术的另一个关键技术，通过实例，读者可以掌握基本的网络通信。

**第4篇：项目实战。**本篇通过一个图书管理系统，运用软件工程的设计思想，讲解如何进行软件项目的开发。书中按照编写需求分析→系统设计→功能设计→创建项目→实现项目模块功能→运行项目的流程进行介绍，带领读者一步步亲身体验开发项目的全过程。

## 本书特点

- ❑ **由浅入深，循序渐进。**本书以初、中级程序员为对象，先从 C++语言基础学起，再到 C++语言的程序结构，然后学习 C++语言的高级应用，最后学习开发一个完整的项目。讲解过程中步骤详尽、版式新颖，并且在程序中会有相应的实例帮助读者更好地理解所讲解的知识，在实例讲解时分步分析，可使读者在阅读时一目了然，从而快速把握书中内容。
- ❑ **语音视频，讲解详尽。**书中每一章均提供声图并茂的视频教学录像，读者可以根据书中提供的视频位置在光盘中找到相应文件。这些视频能够引导初学者快速入门，感受编程的快乐和成就感，增强进一步学习的信心，从而快速成为编程高手。
- ❑ **实例典型，轻松易学。**通过例子学习是最好的学习方式，本书通过一个知识点、一个例子、一个结果、一段评析、一个综合应用的模式，透彻详尽地讲述了实际开发中所需的各类知识。另外，为了便于读者阅读程序代码，快速学习编程技能，书中几乎每行代码都给出了注释。
- ❑ **精彩栏目，贴心提醒。**本书根据需要在各章使用了很多“注意”、“说明”、“技巧”等小栏目，让读者可以在学习过程中更轻松的理解相关知识点及概念，更快地掌握个别技术的应用技巧。
- ❑ **应用实践，随时练习。**书中几乎每章都提供了“实践与练习”，读者能够通过对问题的解答重新回顾、熟悉所学的知识，举一反三，为进一步学习做好充分的准备。

## 读者对象

- |  |  |
|--|--|
| <input checked="" type="checkbox"/> 初学编程的自学者         | <input checked="" type="checkbox"/> 编程爱好者        |
| <input checked="" type="checkbox"/> 大、中专院校的老师 and 学生 | <input checked="" type="checkbox"/> 相关培训机构的老师和学员 |
| <input checked="" type="checkbox"/> 毕业设计的学生          | <input checked="" type="checkbox"/> 初、中级程序开发人员   |
| <input checked="" type="checkbox"/> 程序测试及维护人员        | <input checked="" type="checkbox"/> 参加实习的“菜鸟”程序员 |

## 读者服务

为了方便解决本书疑难问题，读者朋友可加我们的 QQ: 4006751066 (可容纳 10 万人)，也可以登录 [www.mingribook.com](http://www.mingribook.com) 留言，我们将竭诚为您服务。

## 致读者

本书由 C++ 程序开发团队组织编写，主要编写人员有赵永发、高文财、王小科、寇长梅、赵会东、王国辉、陈丹丹、李伟、刘欣、李慧、潘凯华、李继业、刘淇、王双、赵旭阳、陈媛、顾彦玲、陈英、刘莉莉、曹飞飞、朱晓、高春艳、房大伟、刘云峰、吕双、顾丽丽、孟范胜、董大永、李继业、尹强、张磊、王军、刘彬彬、卢瀚、安剑、巩建华、刘锐宁、李伟明、梁水、李鑫、孙秀梅、李钟尉等。在编写本书的过程中，我们始终本着科学、严谨的态度，力求精益求精，但错误、疏漏之处在所难免，敬请广大读者批评指正。

感谢您购买本书，希望本书能成为您编程路上的领航者。

“零门槛”编程，一切皆有可能。

祝读书快乐！

编 者



# 光盘“开发资源库”目录

## 第 1 大部分 实例资源库

(881 个完整实例分析, 光盘路径: 开发资源库/实例资源库)



### 语言基础

















































































- 输出问候语
- 输出带边框的问候语
- 不同类型数据的输出
- 输出字符表情
- 获取用户输入的用户名
- 简单的字符加密
- 实现两个变量的互换
- 判断性别
- 用宏定义实现值互换
- 简单的位运算
- 整数加减法练习
- 李白喝酒问题
- 桃园三结义
- 何年是闰年
- 小球称重
- 购物街中的商品价格竞猜
- 促销商品的折扣计算
- 利用 switch 语句输出倒三角形
- PK 少年高斯
- 灯塔数量
- 上帝创世的秘密
- 小球下落
- 再现乘法口诀表
- 判断名次
- 序列求和
- 简单的级数运算
- 求一个正整数的所有因子
- 一元钱兑换方案








































- 加油站加油
- 买苹果问题
- 猴子吃桃
- 老师分糖果
- 新同学的年龄
- 百钱百鸡问题
- 彩球问题
- 集邮册中的邮票数量
- 用#打印三角形
- 用\*打印图形
- 绘制余弦曲线
- 打印杨辉三角
- 计算某日是该年第几天
- 斐波那契数列
- 角谷猜想
- 哥德巴赫猜想
- 四方定理
- 尼科彻斯定理
- 魔术师的秘密





























### 控件应用

- 文本背景的透明处理
- 具有分隔条的静态文本控件
- 设计群组控件
- 电子时钟
- 模拟超链接效果
- 使用静态文本控件数组设计简易拼图
- 多行文本编辑的编辑框
- 输入时显示选择列表
- 七彩编辑框效果











-  如同话中题字
-  金额编辑框
-  密码安全编辑框
-  个性字体展示
-  在编辑框中插入图片数据
-  RTF 文件读取器
-  在编辑框中显示表情动画
-  位图和图标按钮
-  问卷调查的程序实现
-  热点效果的图像切换
-  实现图文并茂效果
-  按钮七巧板
-  动画按钮
-  向组合框中插入数据
-  输入数据时的辅助提示
-  列表宽度的自动调节
-  颜色组合框
-  枚举系统盘符
-  QQ 登录式的用户选择列表
-  禁止列表框信息重复
-  在两个列表框间实现数据交换
-  上下移动列表项位置
-  实现标签式选择
-  要提示才能看得见
-  水平方向的延伸
-  为列表框换装
-  使用滚动条显示大幅位图
-  滚动条的新装
-  颜色变了
-  进度的百分比显示
-  程序中的调色板
-  人靠衣装
-  头像选择形式的登录窗体
-  以报表显示图书信息
-  实现报表数据的排序
-  在列表中编辑文本
-  QQ 抽屉界面
-  以树状结构显示城市信息
-  节点可编辑
-  节点可拖动
-  选择你喜欢的省、市
-  树控件的服装设计
-  目录树
-  界面的分页显示
-  标签中的图标设置
-  迷你星座查询器
-  设置系统时间
-  时间和月历的同步
-  实现纪念日提醒
-  对数字进行微调
-  为程序添加热键
-  获得本机的 IP 地址
-  AVI 动画按钮
-  GIF 动画按钮
-  图文按钮
-  不规则按钮
-  为编辑框设置新的系统菜单
-  为编辑框控件添加列表选择框
-  多彩边框的编辑框
-  改变编辑框文本颜色
-  不同文本颜色的编辑框
-  位图背景编辑框
-  电子计时器
-  使用静态文本控件设计群组框
-  制作超链接控件
-  利用列表框控件实现标签式数据选择
-  具有水平滚动条的列表框控件
-  列表项的提示条
-  位图背景列表框控件
-  将数据表中的字段添加到组合框控件
-  带查询功能的组合框控件
-  自动调整组合框的宽度
-  多列显示的组合框
-  带图标的组合框
-  显示系统盘符组合框
-  Windows 资源管理器
-  利用列表视图控件浏览数据
-  利用列表视图控件制作导航界面
-  在列表视图中拖动视图项
-  具有排序功能的列表视图控件

-  具有文本录入功能的列表视图控件
-  使用列表视图设计登录界面
-  多级数据库树状结构数据显示
-  带复选功能的树状结构
-  三态效果树控件
-  修改树控件节点连线颜色
-  位图背景树控件
-  显示磁盘目录
-  树型提示框
-  利用 RichEdit 显示 Word 文档
-  利用 RichEdit 控件实现文字定位与标识
-  利用 RichEdit 控件显示图文数据
-  在 RichEdit 中显示不同字体和颜色的文本
-  在 RichEdit 中显示 GIF 动画
-  自定义滚动条控件
-  渐变颜色的进度条
-  应用工具提示控件
-  使用滑块控件设置颜色值
-  绘制滑块控件
-  应用标签控件
-  自定义标签控件
-  向窗体中动态添加控件
-  公交线路模拟
-  设计字体按钮控件
-  设计 XP 风格按钮
-  类似瑞星的目录显示控件
-  绘制分割条
-  显示 GIF 的 ATL 控件
-  类似 Windows 资源管理器的列表视图控件
-  漂亮的热点按钮
-  QQ 抽屉效果的列表视图控件
-  设计类似 QQ 的编辑框安全控件
-  设计电子表格形式的计时器
-  文字显示的进度条控件
-  将 XML 文件树结构信息添加到树控件中
-  读取 RTF 文件到编辑框中
-  个性编辑框
-  设计颜色选择框控件
-  设计图片预览对话框

## ☐ 菜单

-  根据表中数据动态生成菜单
-  创建级联菜单
-  带历史信息的菜单
-  绘制渐变效果的菜单
-  带图标的程序菜单
-  根据 INI 文件创建菜单
-  根据 XML 文件创建菜单
-  为菜单添加核对标记
-  为菜单添加快捷键
-  设置菜单是否可用
-  将菜单项的字体设置为粗体
-  多国语言菜单
-  可以下拉的菜单
-  左侧引航条菜单
-  右对齐菜单
-  鼠标右键弹出菜单
-  浮动的菜单
-  更新系统菜单
-  任务栏托盘弹出菜单
-  单文档右键菜单
-  工具栏下拉菜单
-  编辑框右键菜单
-  列表控件右键菜单
-  工具栏右键菜单
-  在系统菜单中添加菜单项
-  个性化的弹出菜单

## ☐ 工具栏和状态栏

-  带图标的工具栏
-  带背景的工具栏
-  定制浮动工具栏
-  创建对话框工具栏
-  根据菜单创建工具栏
-  工具栏按钮的热点效果
-  定义 XP 风格的工具栏
-  根据表中数据动态生成工具栏
-  工具栏按钮单选效果
-  工具栏按钮多选效果

- ☐ 固定按钮工具栏
- ☐ 可调整按钮位置的工具栏
- ☐ 具有提示功能的工具栏
- ☐ 在工具栏中添加编辑框
- ☐ 带组合框的工具栏
- ☐ 工具栏左侧双线效果
- ☐ 多国语音工具栏
- ☐ 显示系统时间的状态栏
- ☐ 使状态栏随对话框的改变而改变
- ☐ 带进度条的状态栏
- ☐ 自绘对话框动画效果的状态栏
- ☐ 滚动字幕的状态栏
- ☐ 带下拉菜单的工具栏
- ☐ 动态设置是否显示工具栏按钮文本

## 第2大部分 模块资源库

(15 个经典模块, 光盘路径: 开发资源库/模块资源库)

### 模块 1 图像处理模块

- ☐ 图像处理模块概述
  - ☐ 模块概述
  - ☐ 功能结构
  - ☐ 模块预览
- ☐ 关键技术
  - ☐ 位图数据的存储形式
  - ☐ 任意角度旋转图像
  - ☐ 实现图像缩放
  - ☐ 在 Visual C++ 中使用 GDI+ 进行图像处理
  - ☐ 实现图像的水印效果
  - ☐ 浏览 PSD 文件
  - ☐ 利用滚动窗口浏览图片
  - ☐ 使用子对话框实现图像的局部选择
- ☐ 图像旋转模块设计
- ☐ 图像平移模块设计
- ☐ 图像缩放模块设计
- ☐ 图像水印效果模块设计
- ☐ 位图转换为 JPEG 模块设计
- ☐ PSD 文件浏览模块设计
- ☐ 照片版式处理模块设计

### 模块 2 办公助手模块

- ☐ 办公助手模块概述
  - ☐ 模块概述
  - ☐ 功能结构

- ☐ 模块预览
- ☐ 关键技术
  - ☐ 如 QQ 般自动隐藏
  - ☐ 按需要设计编辑框
  - ☐ 设计计算器的圆角按钮
  - ☐ 回行数据在 INI 文件中的读取与写入
  - ☐ 根据数据库数据生成复选框
  - ☐ 饼形图显示投票结果
- ☐ 主窗体设计
- ☐ 计算器设计
- ☐ 便利贴设计
- ☐ 加班模块设计
- ☐ 投票项目模块设计

### 模块 3 桌面精灵模块

- ☐ 桌面精灵模块概述
  - ☐ 模块概述
  - ☐ 功能结构
  - ☐ 模块预览
- ☐ 关键技术
  - ☐ 阳历转换成阴历的算法
  - ☐ 时钟的算法
  - ☐ 实现鼠标穿透
  - ☐ 窗体置顶及嵌入桌面
  - ☐ 添加系统托盘
  - ☐ 开机自动运行

- ▢ 自绘右键弹出菜单
- ▢ 带图标的按钮控件
- ▢ **主窗体设计**
- ▢ **新建备忘录模块设计**
- ▢ **新建纪念日模块设计**
- ▢ **纪念日列表模块设计**
- ▢ **窗口设置模块设计**
- ▢ **提示窗口模块设计**

#### 模块 4 企业通信模块

- ▢ **企业通信模块概述**
  - ▢ 模块概述
  - ▢ 功能结构
  - ▢ 模块预览
- ▢ **关键技术**
  - ▢ 设计支持 QQ 表情的 ATL 控件
  - ▢ 向 CRichEditCtrl 控件中插入 ATL 控件
  - ▢ 向 CRichEditCtrl 控件中插入 ATL 控件
  - ▢ 使用 XML 文件实现组织结构的客户端显示
  - ▢ 在树控件中利用节点数据标识节点的类型（部门信息、男职员、女职员）
  - ▢ 定义数据报结构，实现文本、图像、文件数据的发送与显示
  - ▢ 数据报粘报的简单处理
  - ▢ 实现客户端掉线的自动登录
- ▢ **服务器主窗口设计**
- ▢ **部门设置模块设计**
- ▢ **帐户设置模块设计**
- ▢ **客户端主窗口设计**
- ▢ **登录模块设计**
- ▢ **信息发送窗口模块设计**

#### 模块 5 媒体播放器模块

- ▢ **媒体播放器模块概述**
  - ▢ 模块概述
  - ▢ 模块预览
- ▢ **关键技术**
  - ▢ 如何使用 Direct Show 开发包
  - ▢ 使用 Direct Show 开发程序的方法
  - ▢ 使用 Direct Show 如何确定媒体文件播放完成

- ▢ 使用 Direct Show 进行音量和播放进度的控制
- ▢ 使用 Direct Show 实现字幕叠加
- ▢ 使用 Direct Show 实现亮度、饱和度和对比度调节
- ▢ 设计显示目录和文件的树视图控件

- ▢ **媒体播放器主窗口设计**
- ▢ **视频显示窗口设计**
- ▢ **字幕叠加窗口设计**
- ▢ **视频设置窗口设计**
- ▢ **文件播放列表窗口设计**

#### 模块 6 屏幕录像模块

- ▢ **屏幕录像模块概述**
  - ▢ 模块概述
  - ▢ 功能结构
- ▢ **关键技术**
  - ▢ 屏幕抓图
  - ▢ 抓图时抓取鼠标
  - ▢ 将位图数据流写入 AVI 文件
  - ▢ 将 AVI 文件转换成位图数据
  - ▢ 获得 AVI 文件属性
  - ▢ 根据运行状态显示托盘图标
  - ▢ 获得磁盘的剩余空间
  - ▢ 动态生成录像文件名
- ▢ **主窗体设计**
- ▢ **录像截取模块设计**
- ▢ **录像合成模块设计**

#### 模块 7 计算机监控模块

- ▢ **计算机监控模块概述**
  - ▢ 开发背景
  - ▢ 需求分析
  - ▢ 模块预览
- ▢ **关键技术**
  - ▢ 获取屏幕设备上下文存储为位图数据流
  - ▢ 将位图数据流压缩为 JPEG 数据流
  - ▢ 将 JPEG 数据流分成多个数据报发送到服务器
  - ▢ 将多个数据报组合为一个完整的 JPEG 数据流
  - ▢ 根据 JPEG 数据流显示图像
  - ▢ 双击实现窗口全屏显示
- ▢ **客户端主窗口设计**

☐ 服务器端主窗口设计

☐ 远程控制窗口设计

## 模块 8 考试管理模块

☐ 考试管理模块概述

☐ 关键技术

- 在主窗体显示之前显示登录窗口
- 随机抽题算法
- 编辑框控件设置背景图片
- 显示欢迎窗体
- 计时算法
- 保存答案算法
- 工具栏按钮提示功能实现
- 图标按钮的实现

☐ 数据库设计

- 数据库分析
- 设计表结构

☐ 学生前台考试模块

- 学生考试功能实现
- 学生查分功能实现

☐ 教师后台管理模块

- 后台管理主窗口
- 学生信息管理功能实现
- 试题管理功能实现
- 学生分数查询功能实现

## 模块 9 SQL 数据库提取器模块

☐ SQL 数据库提取器概述

- 模块概述
- 功能结构

☐ 关键技术

- 获得数据表、视图和存储过程
- 获得表结构
- 向 WORD 文档中插入表格
- 向 WORD 表格中插入图片
- 向 EXCEL 表格中插入图片
- 使用 bcp 实用工具导出数据

☐ 主窗体设计

☐ 附加数据库模块设计

☐ 备份数据库模块设计

☐ 数据导出模块设计

☐ 配置 ODBC 数据源模块设计

## 模块 10 万能打印模块

☐ 万能打印模块概述

☐ 关键技术

- 滚动条设置
- 打印中的页码计算和分页预览功能算法
- 数据库查询功能
- 打印控制功能
- 如何解决屏幕和打印机分辨率不统一问题
- 打印新一页

☐ 主窗体设计

☐ Access 数据库选择窗体

☐ SQL Server 数据库选择窗体

☐ 数据库查询模块

☐ 打印设置模块

☐ 打印预览及打印模块

.....

# 第 3 大部分 项目资源库

(15 个企业开发项目, 光盘路径: 开发资源库/项目资源库)

## 项目 1 商品库存管理系统

☐ 系统分析

- 使用 UML 用例图描述商品库存管理系统需求
- 系统流程

系统目标

☐ 系统总体设计

- 系统功能结构设计
- 编码设计

## [-] 数据库设计

- [ ] 创建数据库
- [ ] 创建数据表
- [ ] 数据库逻辑结构设计
- [ ] 数据字典
- [ ] 使用 Visual C++6.0 与数据库连接
- [ ] 如何使用 ADO
- [ ] 重新封装 ADO

## [-] 程序模型设计

- [ ] 从这里开始
- [ ] 类模型分析
- [ ] CBaseComboBox 类分析

## [-] 主程序界面设计

- [ ] 主程序界面开发步骤
- [ ] 菜单资源设计

## [-] 主要功能模块详细设计

- [ ] 商品信息管理
- [ ] 出库管理
- [ ] 调货管理
- [ ] 地域信息管理
- [ ] 库存盘点

## [-] 经验漫谈

- [ ] Windows 消息概述
- [ ] 消息映射
- [ ] 消息的发送
- [ ] 运行时刻类型识别宏
- [ ] MFC 调试宏

## [-] 程序调试与错误处理

- [ ] 零记录时的错误处理
- [ ] 在系统登录时出现的错误

## [-] 对话框资源对照说明

## 项目 2 社区视频监控系统

### [-] 开发背景和系统分析

- [ ] 开发背景
- [ ] 需求分析
- [ ] 可行性分析
- [ ] 编写项目计划书

### [-] 系统设计

- [ ] 系统目标

### [ ] 系统功能结构

- [ ] 系统预览
- [ ] 业务流程图
- [ ] 编码规则
- [ ] 数据库设计

### [-] 公共模块设计

### [-] 主窗体设计

### [-] 用户登录模块设计

### [-] 监控管理模块设计

### [-] 无人广角自动监控模块设计

### [-] 视频回放模块设计

### [-] 开发技巧与难点分析

### [-] 监控卡的选购及安装

- [ ] 监控卡选购分析
- [ ] 监控卡安装
- [ ] 视频采集卡常用函数

## 项目 3 图像处理系统

### [-] 总体设计

- [ ] 需求分析
- [ ] 可行性分析
- [ ] 项目规划
- [ ] 系统功能架构图

### [-] 系统设计

- [ ] 设计目标
- [ ] 开发及运行环境
- [ ] 编码规则

### [-] 技术准备

- [ ] 基本绘图操作
- [ ] 内存画布设计
- [ ] 自定义全局函数
- [ ] 自定义菜单
- [ ] 自定义工具栏

### [-] 主要功能模块的设计

- [ ] 系统架构设计
- [ ] 公共模块设计
- [ ] 主窗体设计
- [ ] 显示位图模块设计
- [ ] 显示 JPEG 模块设计



显示 GIF 模块设计

位图转换为 JPEG 模块设计

位图旋转模块设计

线性变换模块设计

手写数字识别模块设计

#### ▣ 疑难问题分析解决

读取位图数据

位图旋转时解决位图字节对齐

#### ▣ 文件清单

### 项目 4 物流管理系统

#### ▣ 系统分析

概述

可行性分析

系统需求分析

#### ▣ 总体设计

项目规划

系统功能结构图

#### ▣ 系统设计

设计目标

数据库设计

系统运行环境

#### ▣ 功能模块设计

构建应用程序框架

封装数据库

主窗口设计

基础信息基类

支持扫描仪辅助录入功能业务类

业务类

业务查询类

统计汇总类

审核类

派车单写 IC 卡模块

配送申请模块

三检管理模块

报关过程监控模块

数据备份模块

数据恢复模块

库内移动模块

公司设置模块

报关单管理模块

报关单审核模块

配送审核模块

派车回场确计模块

系统提示模块

查验管理模块

系统初始化模块

系统登录模块

通关管理模块

权限设置模块

商品入库排行分析模块

系统注册模块

在途反馈模块

#### ▣ 疑难问题分析与解决

库内移动

根据分辨率画背景

#### ▣ 程序调试

#### ▣ 文件清单

### 项目 5 局域网屏幕监控系统

#### ▣ 系统分析

需求分析

可行性分析

#### ▣ 总体设计

项目规划

系统功能架构图

#### ▣ 系统设计

设计目标

开发及运行环境

#### ▣ 技术准备

套接字函数

套接字的初始化

获取套接字数据接收的事件

封装数据报

将屏幕图像保存为位图数据流

读写 INI 文件

使用 GDI+

#### ▣ 主要功能模块的设计

- 客户端模块设计
  - 服务器端模块设计
- 疑难问题分析解决
    - 使用 GDI+ 产生的内存泄露
    - 释放无效指针产生地址访问错误
  - 文件清单

## 项目 6 客户管理系统

- 系统分析
    - 概述
    - 需求分析
    - 可行性分析
  - 总体设计
    - 项目规划
    - 系统功能架构图
  - 系统设计
    - 设计目标
    - 开发及运行环境
    - 数据库设计
  - 技术准备
    - 数据库的封装
    - 封装 ADO 数据库的代码分析
  - 主要功能模块设计
    - 主窗体
    - 客户信息
    - 联系人信息
    - 联系人信息查询
    - 关于模块
    - 增加操作员模块
    - 客户反馈满意程度查询
    - 客户反馈模块
    - 客户呼叫中心模块
    - 客户级别设置模块
    - 客户满意程度设置模块
    - 客户投诉模块
    - 登录界面
    - 密码修改模块
    - 客户信息查询模块
    - 区域信息模块

- 企业类型模块
- 企业性质模块
- 企业资质设置模块
- 客户投诉满意程度查询
- 业务往来模块

- 疑难问题分析与解决
    - 使用 CtabCtrl 类实现分页的 2 种实现方法
    - ADO 不同属性和方法的弊端及解决方法
  - 程序调试
  - 文件清单

## 项目 7 企业短信群发管理系统

- 开发背景和系统分析
    - 开发背景
    - 需求分析
    - 可行性分析
    - 编写项目计划书
  - 系统设计
    - 系统目标
    - 系统功能结构图
    - 系统预览
    - 业务流程图
    - 数据库设计
  - 公共类设计
    - 自定义 SetHBitmap 方法
    - 处理 WM\_MOUSEMOVE 事件
  - 主窗口设计
  - 短信猫设置模块设计
  - 电话簿管理模块设计
  - 常用语管理模块设计
  - 短信息发送模块设计
  - 短信息接收模块设计
  - 开发技巧与难点分析
    - 显示“收到新信息”对话框
    - 制作只允许输入数字的编辑框
  - 短信猫应用

## 项目 8 商品销售管理系统

- 系统分析

- 用 UML 顺序图描述销售业务处理流程
  - 业务流程
  - 系统的总体设计思想
- 系统设计
  - 系统功能设计
  - 数据库设计
- 主界面设计
- 主要功能模块详细设计
  - 系统登录模块
  - 基础信息查询基类
  - 客户信息管理
  - 销售管理
  - 业务查询基类
  - 权限设置
- 经验漫谈
  - 大小写金额的转化函数 MoneyToChineseCode
  - 怎样取得汉字拼音简码
  - 怎样在字符串前或后生成指定数量的字符
  - 日期型(CTime)与字符串(CString)之间的转换
  - Document 与 View 之间的相互作用
  - 列表框控件(List Box)的使用方法
  - 组合框控件(Combo Box)的使用方法
- 程序调试及错误处理
  - 截获回车后的潜在问题
  - 数据恢复时的错误
- 对话框资源对照说明

## 项目 9 进销存管理系统

- 概述
  - 系统需求分析
  - 可行性分析
- 总体设计
  - 项目规划
  - 系统功能结构图
- 系统设计
  - 设计目标
  - 系统运行环境
  - 数据库设计
- 功能模块设计
  - 主窗口设计

- 系统登录管理
  - 商品销售管理
  - 商品入库管理
  - 调货登记管理
  - 权限设置管理
- 疑难问题分析与解决
  - 使 CListCtrl 控件可编辑
  - 显示自动提示窗口(CListCtrlPop)
  - 处理局部白色背景
  - 给编辑框加一个下划线
  - 修改控件字体
- 程序调试
  - 使用调试窗口
  - 输出信息到“Output”窗口
  - 处理内存泄漏问题
- 文件清单

## 项目 10 企业电话语音录音管理系统

- 开发背景和需求分析
  - 开发背景
  - 需求分析
- 系统设计
  - 系统目标
  - 系统功能结构
  - 系统预览
  - 业务流程图
  - 数据库设计
- 公共模块设计
- 主窗体设计
- 来电管理模块设计
- 电话录音管理模块设计
- 员工信息管理模块设计
- 产品信息管理模块设计
- 开发技巧与难点分析
  - 为程序设置系统托盘
  - 对话框的显示
- 语音卡函数介绍

.....

## 第4大部分 能力测试资源库

(616 道能力测试题目, 光盘路径: 开发资源库/能力测试)

### 第1部分 Visual C++ 编程基础能力测试



.....

### 第2部分 数学及逻辑思维能力测试

- ☐  基本测试
-  进阶测试

-  高级测试



### 第3部分 编程英语能力测试

- ☐  英语基础能力测试
-  英语进阶能力测试






## 第5大部分 面试系统资源库

(371 项面试真题, 光盘路径: 开发资源库/面试系统)





### 第1部分 C、C++程序员职业规划







- ☐  你了解程序员吗
-  程序员自我定位

### 第2部分 C、C++程序员面试技巧

- ☐  面试的三种方式
-  如何应对企业面试
-  英语面试
-  电话面试
-  智力测试

### 第3部分 C、C++常见面试题

- ☐  C/C++语言基础面试题
-  字符串与数组面试题
-  函数面试题
-  指针与引用面试题

-  预处理和内存管理面试真题
-  位运算面试真题
-  面向对象面试真题
-  继承与多态面试真题
-  数据结构与常用算法面试真题
-  排序与常用算法面试真题

### 第4部分 C、C++ 企业面试真题汇编

- ☐  企业面试真题汇编(一)
-  企业面试真题汇编(二)
-  企业面试真题汇编(三)
-  企业面试真题汇编(四)




### 第5部分 VC 虚拟面试系统





.....


# 目 录

## Contents


## 第1篇 基础知识

第1章 绪论.....	3	2.3.2 实型常量.....	21
 视频讲解: 1 小时 18 分钟		2.3.3 字符常量.....	22
1.1 C++历史背景.....	4	2.3.4 字符串常量.....	23
1.1.1 20 世纪最伟大的发明.....	4	2.3.5 其他常量.....	23
1.1.2 C++发展历程.....	4	2.4 变量.....	23
1.1.3 C++中的杰出人物.....	5	2.4.1 标识符.....	24
1.2 常用开发环境.....	6	2.4.2 变量与变量说明.....	24
1.2.1 Visual C++ 6.0.....	6	2.4.3 整型变量.....	25
1.2.2 Visual C++ 2008.....	7	2.4.4 实型变量.....	25
1.2.3 GCC/G++.....	8	2.4.5 变量赋值.....	26
1.2.4 Dev-C.....	8	2.4.6 变量赋初值.....	26
1.2.5 Eclipse.....	8	2.4.7 字符变量.....	27
1.3 认知 C++程序代码.....	9	2.5 数据输入与输出.....	28
1.4 C++工程项目文件.....	10	2.5.1 控制台屏幕.....	28
1.5 使用 VC 创建程序.....	11	2.5.2 C++语言中的流.....	28
1.6 编译与连接过程.....	13	2.5.3 流操作的控制.....	30
1.7 C++的特点.....	15	2.6 小结.....	36
1.8 小结.....	16	2.7 实践与练习.....	36
第2章 数据类型.....	17	第3章 表达式与语句.....	37
 视频讲解: 1 小时 26 分钟		 视频讲解: 55 分钟	
2.1 第一个 C++程序.....	18	3.1 运算符.....	38
2.1.1 #include 指令.....	18	3.1.1 算术运算符.....	38
2.1.2 注释.....	18	3.1.2 关系运算符.....	39
2.1.3 main 函数.....	19	3.1.3 逻辑运算符.....	40
2.1.4 函数体.....	19	3.1.4 赋值运算符.....	41
2.1.5 函数返回值.....	19	3.1.5 位运算.....	42
2.2 数据类型.....	19	3.1.6 移位运算符.....	43
2.3 常量及符号.....	20	3.1.7 sizeof 运算符.....	45
2.3.1 整型常量.....	21	3.1.8 条件运算符.....	45


3.1.9 逗号运算符.....	46	5.7.1 阿姆斯壮数.....	84
3.2 结合性和优先级.....	46	5.7.2 巴斯卡三角形.....	85
3.3 表达式.....	47	5.7.3 对输入的分数进行排名.....	86
3.3.1 算术表达式.....	48	5.8 小结.....	87
3.3.2 关系表达式.....	48	5.9 实践与练习.....	87
3.3.3 条件表达式.....	48		
3.3.4 赋值表达式.....	49		
3.3.5 逻辑表达式.....	49		
3.3.6 逗号表达式.....	49		
3.3.7 表达式中的类型转换.....	50		
3.4 语句.....	53		
3.5 小结.....	54		
3.6 实践与练习.....	54		
<b>第4章 条件判断语句.....</b>	<b>55</b>		
 <b>视频讲解: 35 分钟</b>			
4.1 决策分支.....	56		
4.2 判断语句.....	57		
4.2.1 第一种形式的判断语句.....	57		
4.2.2 第二种形式的判断语句.....	58		
4.2.3 第三种形式的判断语句.....	60		
4.3 使用条件运算符进行判断.....	61		
4.4 switch 语句.....	63		
4.5 判断语句的嵌套.....	66		
4.6 小结.....	68		
4.7 实践与练习.....	68		
<b>第5章 循环语句.....</b>	<b>69</b>		
 <b>视频讲解: 53 分钟</b>			
5.1 while 循环.....	70		
5.2 do...while 循环.....	71		
5.3 while 与 do...while 比较.....	73		
5.4 for 循环语句.....	74		
5.5 循环控制.....	78		
5.5.1 控制循环的变量.....	78		
5.5.2 break 语句.....	79		
5.5.3 continue 语句.....	80		
5.5.4 goto 语句.....	81		
5.6 循环嵌套.....	82		
5.7 循环应用实例.....	84		
		<b>第6章 函数.....</b>	<b>89</b>
		 <b>视频讲解: 1 小时 14 分钟</b>	
		6.1 函数概述.....	90
		6.1.1 函数的定义.....	90
		6.1.2 函数的声明.....	90
		6.2 函数参数及返回值.....	92
		6.2.1 返回值.....	92
		6.2.2 空函数.....	92
		6.2.3 形参与实参.....	93
		6.2.4 默认参数.....	93
		6.2.5 可变参数.....	95
		6.3 函数调用.....	96
		6.3.1 传值调用.....	96
		6.3.2 嵌套调用.....	98
		6.3.3 递归调用.....	98
		6.4 变量作用域.....	102
		6.5 重载函数.....	103
		6.6 内联函数.....	104
		6.7 变量的存储类别.....	105
		6.7.1 auto 变量.....	105
		6.7.2 static 变量.....	106
		6.7.3 register 变量.....	108
		6.7.4 extern 变量.....	108
		6.8 小结.....	109
		6.9 实践与练习.....	109
		<b>第7章 数组、指针和引用.....</b>	<b>111</b>
		 <b>视频讲解: 1 小时 27 分钟</b>	
		7.1 一维数组.....	112
		7.1.1 一维数组的声明.....	112
		7.1.2 一维数组的引用.....	112
		7.1.3 一维数组的初始化.....	113
		7.2 二维数组.....	114
		7.2.1 二维数组的声明.....	114

7.2.2 二维数组元素的引用 .....	115	8.1.1 结构体定义 .....	148
7.2.3 二维数组的初始化 .....	115	8.1.2 结构体变量 .....	149
7.3 字符数组 .....	117	8.1.3 结构体成员及初始化 .....	149
7.4 指针 .....	124	8.1.4 结构体的嵌套 .....	152
7.4.1 变量与指针 .....	124	8.1.5 结构体大小 .....	153
7.4.2 指针运算符和取地址运算符 .....	127	8.2 结构体与函数 .....	154
7.4.3 指针运算 .....	128	8.2.1 结构体变量作函数参数 .....	155
7.5 指针与数组 .....	130	8.2.2 结构体指针作函数参数 .....	155
7.5.1 数组的存储 .....	130	8.3 结构体数组 .....	156
7.5.2 指针与一维数组 .....	130	8.3.1 结构体数组声明与引用 .....	157
7.5.3 指针与二维数组 .....	132	8.3.2 指针访问结构体数组 .....	158
7.5.4 指针与字符数组 .....	135	8.4 共用体 .....	159
7.6 指向函数的指针 .....	136	8.4.1 共用体的定义与声明 .....	159
7.7 引用 .....	137	8.4.2 共用体的大小 .....	160
7.7.1 使用引用传递参数 .....	139	8.4.3 共用体的特点 .....	161
7.7.2 指针传递参数 .....	140	8.5 枚举类型 .....	161
7.7.3 数组作函数参数 .....	141	8.5.1 枚举类型的声明 .....	161
7.8 指针数组 .....	143	8.5.2 枚举类型变量 .....	162
7.9 小结 .....	145	8.5.3 枚举类型的运算 .....	163
7.10 实践与练习 .....	145	8.6 自定义数据类型 .....	165
第8章 构造数据类型 .....	147	8.7 小结 .....	166
 视频讲解: 59 分钟		8.8 实践与练习 .....	166
8.1 结构体 .....	148		

## 第2篇 核心技术

第9章 面向对象编程 .....	169	第10章 类和对象 .....	177
 视频讲解: 32 分钟		 视频讲解: 1 小时 1 分钟	
9.1 面向对象概述 .....	170	10.1 C++类 .....	178
9.2 面向对象与面向过程编程 .....	171	10.1.1 类概述 .....	178
9.2.1 面向过程编程 .....	171	10.1.2 类的声明与定义 .....	178
9.2.2 面向对象编程 .....	171	10.1.3 类的实现 .....	180
9.2.3 面向对象的特点 .....	172	10.1.4 对象的声明 .....	184
9.3 统一建模语言 .....	172	10.2 构造函数 .....	186
9.3.1 统一建模语言概述 .....	172	10.2.1 构造函数概述 .....	186
9.3.2 统一建模语言的结构 .....	173	10.2.2 复制构造函数 .....	188
9.3.3 面向对象的建模 .....	175	10.3 析构函数 .....	190
9.4 小结 .....	175	10.4 类成员 .....	192



10.4.1 访问类成员.....	192	11.1.1 类的继承.....	216
10.4.2 内联成员函数.....	194	11.1.2 继承后可访问性.....	218
10.4.3 静态类成员.....	195	11.1.3 构造函数访问顺序.....	220
10.4.4 隐藏的 this 指针.....	197	11.1.4 子类隐藏父类的成员函数.....	222
10.4.5 嵌套类.....	198	11.2 重载运算符.....	225
10.4.6 局部类.....	199	11.2.1 重载运算符的必要性.....	225
10.5 友元.....	200	11.2.2 重载运算符的形式与规则.....	226
10.5.1 友元概述.....	200	11.2.3 重载运算符的运算.....	228
10.5.2 友元类.....	202	11.2.4 转换运算符.....	230
10.5.3 友元方法.....	203	11.3 多重继承.....	231
10.6 命名空间.....	206	11.3.1 多重继承定义.....	231
10.6.1 使用命名空间.....	206	11.3.2 二义性.....	233
10.6.2 定义命名空间.....	206	11.3.3 多重继承的构造顺序.....	234
10.6.3 在多个文件中定义命名空间.....	209	11.4 多态.....	235
10.6.4 定义嵌套的命名空间.....	210	11.4.1 虚函数概述.....	236
10.6.5 定义未命名的命名空间.....	212	11.4.2 利用虚函数实现动态绑定.....	236
10.7 小结.....	212	11.4.3 虚继承.....	237
10.8 实践与练习.....	213	11.5 抽象类.....	239
第 11 章 继承与派生.....	215	11.5.1 纯虚函数.....	239
 视频讲解: 57 分钟		11.5.2 实现抽象类中的成员函数.....	241
11.1 继承.....	216	11.6 小结.....	242
		11.7 实践与练习.....	243

## 第 3 篇 高级应用

第 12 章 模板.....	247	12.3.1 定制类模板.....	258
 视频讲解: 49 分钟		12.3.2 定制类模板成员函数.....	260
12.1 函数模板.....	248	12.3.3 模板部分定制.....	261
12.1.1 函数模板的定义.....	248	12.4 链表类模板.....	262
12.1.2 函数模板的作用.....	249	12.4.1 链表.....	263
12.1.3 重载函数模板.....	251	12.4.2 链表类模板.....	265
12.2 类模板.....	252	12.4.3 类模板的静态数据成员.....	267
12.2.1 类模板的定义与声明.....	252	12.5 小结.....	269
12.2.2 简单类模板.....	254	12.6 实践与练习.....	269
12.2.3 默认模板参数.....	255	第 13 章 STL 标准模板库.....	271
12.2.4 为具体类型的参数提供默认值.....	255	 视频讲解: 35 分钟	
12.2.5 有界数组模板.....	256	13.1 序列容器.....	272
12.3 模板的使用.....	258	13.1.1 向量类模板.....	272

13.1.2 双端队列类模板.....	274	15.2.1 语法错误.....	328
13.1.3 链表类模板.....	276	15.2.2 连接错误.....	329
13.2 结合容器.....	278	15.2.3 运行时错误.....	329
13.2.1 set 类模板.....	278	15.2.4 逻辑错误.....	330
13.2.2 multiset 类模板.....	282	15.3 调试工具的使用.....	330
13.2.3 map 类模板.....	286	15.3.1 创建调试程序.....	331
13.2.4 multimap 类模板.....	288	15.3.2 进入调试状态.....	332
13.3 算法.....	289	15.3.3 Watch 窗口.....	332
13.3.1 非修正序列算法.....	289	15.3.4 Call Stack 窗口.....	332
13.3.2 修正序列算法.....	292	15.3.5 Memory 窗口.....	333
13.3.3 排序算法.....	295	15.3.6 Variables 窗口.....	333
13.3.4 数值算法.....	301	15.3.7 Registers 窗口.....	333
13.4 迭代器.....	305	15.3.8 Disassembly 窗口.....	334
13.4.1 输出迭代器.....	305	15.4 调试的基本应用.....	334
13.4.2 输入迭代器.....	306	15.4.1 变量的跟踪与查看.....	334
13.4.3 前向迭代器.....	307	15.4.2 位置断点的使用.....	335
13.4.4 双向迭代器.....	307	15.4.3 数据断点的使用.....	337
13.4.5 随机访问迭代器.....	308	15.5 调试的高级应用.....	338
13.5 小结.....	309	15.5.1 在调试时修改变量的值.....	338
13.6 实践与练习.....	309	15.5.2 在循环中调试.....	339
第 14 章 RTTI 与异常处理.....	311	15.6 小结.....	340
 视频讲解: 22 分钟		15.7 实践与练习.....	340
14.1 RTTI (运行时类型识别).....	312	第 16 章 文件操作.....	341
14.1.1 什么是 RTTI.....	312	 视频讲解: 58 分钟	
14.1.2 RTTI 与引用.....	313	16.1 文件流.....	342
14.1.3 RTTI 与多重继承.....	314	16.1.1 C++中的流类库.....	342
14.1.4 RTTI 映射语法.....	315	16.1.2 类库的使用.....	342
14.2 异常处理.....	317	16.1.3 ios 类中的枚举常量.....	343
14.2.1 抛出异常.....	318	16.1.4 流的输入/输出.....	343
14.2.2 异常捕获.....	319	16.2 文件打开.....	344
14.2.3 异常匹配.....	322	16.2.1 打开方式.....	344
14.2.4 标准异常.....	324	16.2.2 默认打开模式.....	345
14.3 小结.....	324	16.2.3 打开文件同时创建文件.....	346
14.4 实践与练习.....	325	16.3 文件的读写.....	347
第 15 章 程序调试.....	327	16.3.1 文件流.....	347
 视频讲解: 33 分钟		16.3.2 写文本文件.....	349
15.1 选择正确的调试方法.....	328	16.3.3 读取文本文件.....	349
15.2 程序错误常见的 4 种类型.....	328	16.3.4 二进制文件的读写.....	350

16.3.5 实现文件复制.....	351	17.1.3 IP 地址 .....	363
16.4 文件指针移动操作 .....	352	17.1.4 数据包格式 .....	364
16.4.1 文件错误与状态 .....	352	17.2 套接字 .....	366
16.4.2 文件的追加 .....	353	17.2.1 Winsock 套接字 .....	366
16.4.3 文件结尾的判断 .....	354	17.2.2 Winsock 的使用 .....	366
16.4.4 在指定位置读写文件 .....	356	17.2.3 套接字阻塞模式 .....	371
16.5 文件和流的关联和分离 .....	357	17.2.4 字节顺序 .....	371
16.6 删除文件 .....	358	17.2.5 面向连接流 .....	372
16.7 小结 .....	359	17.2.6 面向无连接流 .....	372
16.8 实践与练习 .....	359	17.3 简单协议通信 .....	373
第 17 章 网络通信 .....	361	17.3.1 服务端 .....	373
📺 视频讲解: 39 分钟		17.3.2 客户端 .....	375
17.1 TCP/IP 协议 .....	362	17.3.3 实例的运行 .....	377
17.1.1 OSI 参考模型 .....	362	17.4 小结 .....	377
17.1.2 TCP/IP 参考模型 .....	362	17.5 实践与练习 .....	377

## 第 4 篇 项目实战

第 18 章 图书管理系统 .....	381	18.2 图书类 .....	383
📺 视频讲解: 42 分钟		18.3 主程序 .....	387
18.1 系统设计 .....	382	18.4 添加图书 .....	391
18.1.1 需求分析 .....	382	18.5 显示图书信息 .....	391
18.1.2 系统目标 .....	382	18.6 删除图书 .....	394
18.1.3 系统功能结构 .....	382	18.7 小结 .....	394

# 第 1 篇

## 基础知识

- » 第 1 章 绪论
- » 第 2 章 数据类型
- » 第 3 章 表达式与语句
- » 第 4 章 条件判断语句
- » 第 5 章 循环语句
- » 第 6 章 函数
- » 第 7 章 数组、指针和引用
- » 第 8 章 构造数据类型

本篇讲解 C++ 语言基础部分，只有具备了牢固的基础知识才能更快地掌握更高级的技术内容。通过对 C++ 语言的历史和特性、选择 C++ 语言的开发环境、算法、C++ 语言的数据类型、运算符与表达式、常用的数据输入/输出函数、选择结构程序设计和循环控制这些内容的介绍，结合流程图和实例，并通过视频的指导讲解，为以后编程奠定坚实的基础。



# 第 1 章

---

## 绪论

(  视频讲解：1 小时 18 分钟 )

C++是当今最为流行的编程语言之一，它是在C语言基础上发展起来的。随着面向对象编程思想的发展，C++也融入了新的编程理念，这些理念有利于程序的开发。C++从语言角度来讲也是个规范，随着规范的发布，许多C++编译器不断涌现，不同的C++编译器也带来了不同的语言特性，这给程序员带来了广阔的选择空间。

通过阅读本章，您可以：

- » 了解 C++ 的发展历程
- » 了解为 C++ 发展做出杰出贡献的人物
- » 掌握主要的 C++ 编译器及开发环境
- » 掌握 C++ 项目文件及编译工程

## 1.1 C++历史背景

 视频讲解：光盘\TM\lx\1\C++历史背景.exe

学习一门语言，首先要对这门语言有一定的了解，要知道这门语言能做什么，要怎样才能学好。本节将对 C++ 语言的历史背景进行简单的介绍，使读者对 C++ 语言有一个简单而直接的印象。

### 1.1.1 20 世纪最伟大的发明

计算机的出现给人们的生活带来了巨大的改变，那么它是如何发展起来的呢？开始人们致力于能够进行四则运算的机器，是通过机械齿轮运作的加法器，而后是精度只有 12 位的乘法计算器，直到 1847 年 Charles Babbages 开发出能计算 31 位精度的机械式差分机，这台差分机被普遍认为是世界第一台机械式计算机。随着电子物理的发展，真空二极管、真空三极管问世，到 1939 年第一部用真空管计算的机器被研制出来，该机器是能进行 16 位加法运算的机器；随后，氖气灯（霓虹灯）存储器、复杂数字计算机（断电器计数器）、可编写程序的计数机，被一一研制出来。1946 年，第一台电子管计算机 ENIAC 在美国被研制出来，这台计算机占地 170 平方米，重 30 吨，有 1.8 万个电子管，用十进制计算，每秒运算 5000 次。计算机从此进入了电子计算机时代，期间经历了真空管计算机、晶体管计算机、集成电路计算机、大规模集成电路计算机 4 个阶段，每一个阶段都是随着电子物理的发展而发展的，后来晶体管的出现取代了电子管，将电子元件结合到一片小小的硅片上，形成集成电路（IC），在一个芯片上容纳几百个甚至上千个电子元件形成了大规模集成电路（LSI），直到现在已经出现了 32 纳米制作的电子芯片，可谓是发展迅速。计算机运行速度越来越快，从第一台计算机的每秒 5000 次到现在的 2GHz。

现在计算机已经应用到各个领域，包括科学计算、信号检测、数据管理、辅助设计等，人们的生活已经渐渐离不开它，所以说计算机是 20 世纪最伟大的发明。

### 1.1.2 C++发展历程

早期的计算机程序语言就是计算机控制指令，每条指令为一组二进制数，不同的计算机都有不同的计算机指令集。使用二进制指令集开发程序是件很头痛的事，需要记住大量的二进制数，为了便于记忆，人们将二进制数用字母组合代替，以字符串关键字代替二进制机器码的编程语言称为汇编语言，汇编语言被称为是低级语言。虽然汇编语言比机器码容易记忆，但仍然存在可读性差的缺点，大量的跳转指令和地址值很难让程序员在很短的时间理解程序的意思，于是编程语言进入了高级语言时代。

第一个高级语言是美国尤尼法克公司在 1952 年研制成功的 Short Code，但被广泛使用的高级语言是 FORTRAN，它是由美国科学家巴克斯设计并在 IBM 公司的计算机上实现的。但 FORTRAN 语言和 Algol60 主要应用于科学和工程计算，随后出现了 Pascal 和 C 语言。C 语言是在其他语言基础上发展起来的。首先是 Richard Martin 开发一种高级语言 BCPL，随后 Ken Thompson 使用 BCPL 语言对其进行



了简化,形成一门新的语言——B语言,但B语言没有类型的概念,Dennis Ritchie对B语言进行研究和改进,在B语言基础上添加了结构和类型,并将这个改进后的语言命名为C语言,寓意很简单,因为字母C是字母B的下一个字母,预示着语言的发展。

本书所讲述的C++语言就是从C语言发展而来的。Stroustrup经过钻研在C语言中加入类的概念,C++最初的名字是C with Class,到1983年12月由Rick Mascitti建议改名为CPlusPlus,即C++。最开始提出类概念的语言是Simula,它具有很高的灵活性,但无法胜任比较大型的程序。此后在Simula语言基础上发展的语言Smalltalk才是真正的面面向对象语言,但Smalltalk-80不支持多继承。

C++从Simula继承了类的概念,从Algol68继承了运算符重载、引用以及在任何地方声明变量的能力,从BCPL获得了“//”注释,从Ada得到了模板、名字空间,从Ada、Clu和ML取来了异常。

### 1.1.3 C++中的杰出人物



Dennis M. Ritchie

Dennis M. Ritchie 被称为C语言之父,UNIX之父,生于1941年9月9日,哈佛大学数学博士,现任朗讯科技公司贝尔实验室(原AT&T实验室)下属的计算机科学研究系统软件研究部的主任一职。他开发了C语言,并著有《C程序设计语言》(*The C Programming Language*)一书,还和Ken Thompson一起开发了UNIX操作系统。他因杰出的工作得到了众多计算机组织的公认和表彰,1983年,获得美国计算机协会颁发的图灵奖(又称计算机界的诺贝尔奖),还获得过C&C基金奖、电气和电子工程师协会优秀奖章、美国国家技术奖章等多项大奖。



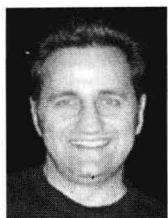
Bjarne Stroustrup

Bjarne Stroustrup 1950年出生于丹麦,先后毕业于丹麦阿鲁斯大学和英国剑桥大学,AT&T大规模程序设计研究部门负责人,AT&T贝尔实验室和ACM成员。1979年,Stroustrup开始开发一种语言,当时称为“C with Class”,后来演化为C++。1998年,ANSI/ISO C++标准建立,同年,Stroustrup推出其经典著作*The C++ Programming Language*的第三版。



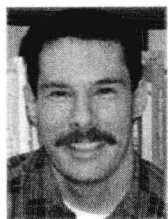
Scott Meyers

Scott Meyers 是世界顶级的C++软件开发技术权威人士之一,他拥有Brown University的计算机科学博士学位,其著作*Effective C++*和*More Effective C++*很受编程人员的喜爱。Scott Meyers曾经是《C++ Report》的专栏作家,为《C/C++ Users Journal》和《Dr. Dobbs's Journal》撰过稿,为全球范围内的客户提供咨询活动。他还是Advisory Boards for NumeriX LLC和InfoCruiser公司的成员。



Andrei Alexandrescu

Andrei Alexandrescu 被认为是新一代 C++ 天才的代表人物，2001 年撰写了经典名著 *Modern C++ Design*，其中对 Template 技术进行了精湛运用，第一次将模板作为参数在模板编程中使用，该书震撼了整个 C++ 社群，开辟了 C++ 编程领域的“Modern C++”新时代。此外，他还与 Herb Sutter 合著了 *C++ Coding Standards*。他在对象复制（objectcopying）、对齐约束（alignment constraint）、多线程编程、异常安全和搜索等领域作出了巨大贡献。



Herb Sutter

Herb Sutter 是 C++ Standard Committee 的主席，作为 ISO/ANSI C++ 标准委员会的委员，Herb Sutter 是 C++ 程序设计领域屈指可数的大师之一。他的 Exceptional 系列三本书（*Exceptional C++*、*More Exceptional C++* 和 *Exceptional C++ Style*）成为 C++ 程序员的必读书。他是深受程序员喜爱的技术讲师和作家，是《C/C++ Users Journal》的撰稿编辑和专栏作者，曾发表了上百篇软件开发方面的技术文章和论文。他还担任 Microsoft Visual C++ 架构师，和 Stan Lippman 一道在微软主持 VC 2005（即 C++/CLI）的设计。



Andrew Koenig

Andrew Koenig 是 AT&T 公司 Shannon 实验室大规模编程研究部门中的成员，同时也是 C++ 标准委员会的项目编辑，是一位真正的 C++ 内部权威。Andrew Koenig 的编程经验超过 30 年，其中有 15 年在使用 C++，已经出版了超过 150 篇和 C++ 有关的论文，并且在世界范围内就这个主题进行过多次演讲，对 C++ 的最大贡献是带领 Alexander Stepanov 将 STL 引入 C++ 标准。

## 1.2 常用开发环境

 视频讲解：光盘\TM\1\1\常用开发环境.exe

在使用 C++ 语言时，需要选择一款开发环境，那么有哪些环境可供用户选择呢？下面就对一些常用的 C++ 语言开发环境进行简单的介绍。

### 1.2.1 Visual C++ 6.0

Visual C++ 6.0 是由微软开发的 C++ 开发环境，它是 Visual Studio 集成开发环境中的一员。Visual C++ 6.0 可以创建 Windows 应用程序、DLL 动态链接库、COM 组件以及 ActiveX 控件等。Visual C++ 6.0 开发环境如图 1.1 所示。

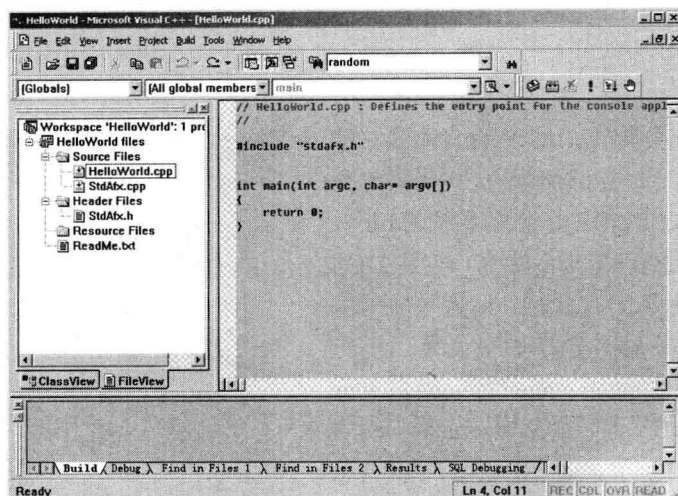


图 1.1 Visual C++ 6.0 开发环境

## 1.2.2 Visual C++ 2008

Visual C++ 2008 是微软继 Visual C++ 6.0 之后新设计的集成开发环境，它更加支持 C++ 标准规范，而且还支持托管程序的编译和 WebService。应该说 Visual C++ 2008 更加强大，无论是功能上还是编译速率上都有很大提高。Visual C++ 2008 开发环境如图 1.2 所示。

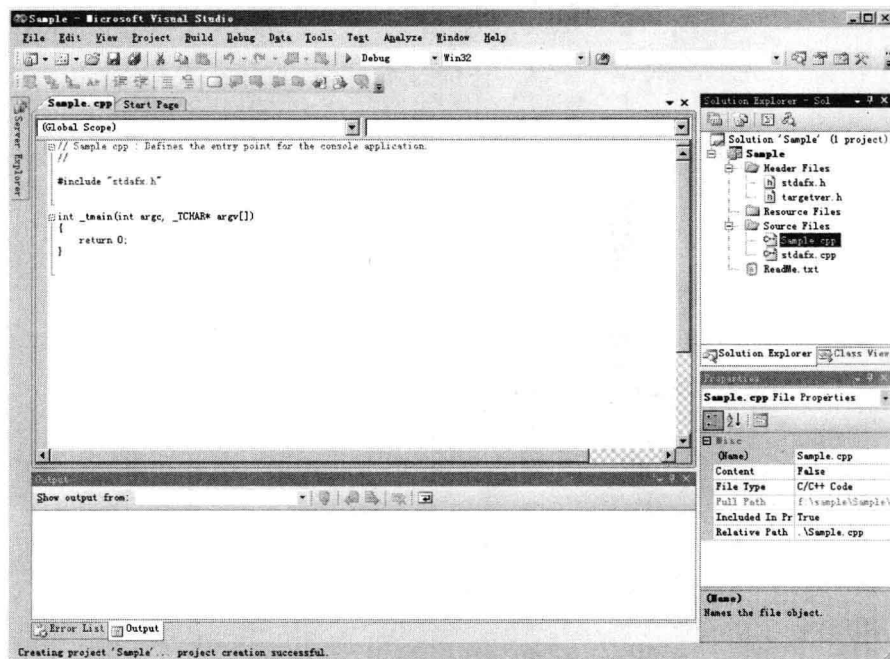


图 1.2 Visual C++ 2008 开发环境

### 1.2.3 GCC/G++

Linux 系统中一般都会带有 C/C++ 的编译器，能够编译 C 代码的是 GCC，能够编译 C++ 代码的是 G++，如果是在没有 GDK 或 GDE 等界面系统的 Linux 系统下，编写代码需要使用 VI 文本命令，它和 DOS 下的 TYPE 命令很像，由于没有鼠标，只能顺序地编写代码，使用它编写代码对于程序员来说相当不方便。但在有界面的 Linux 系统下，系统都会提供可视化的文本编辑器，其中比较有名的就是 emacs，因为 emacs 集成了编译代码的菜单，通过 emacs 可以直接编写代码并编辑，并且可以直接执行编译后的程序。emacs 开发环境如图 1.3 所示。

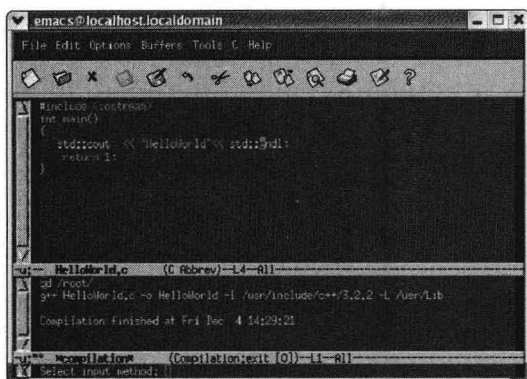


图 1.3 emacs 开发环境

### 1.2.4 Dev-C

Dev-C 是将 Linux 系统的 GCC/G++ 移到 Windows 系统后的产物，具体来讲，编译代码的程序是 GCC.exe 和 G++.exe。这两个程序属于开源项目 mingw，Dev-C 只是能够调用 GCC.exe 和 G++.exe 来编译程序的代码编辑器。Dev-C 开发环境如图 1.4 所示。

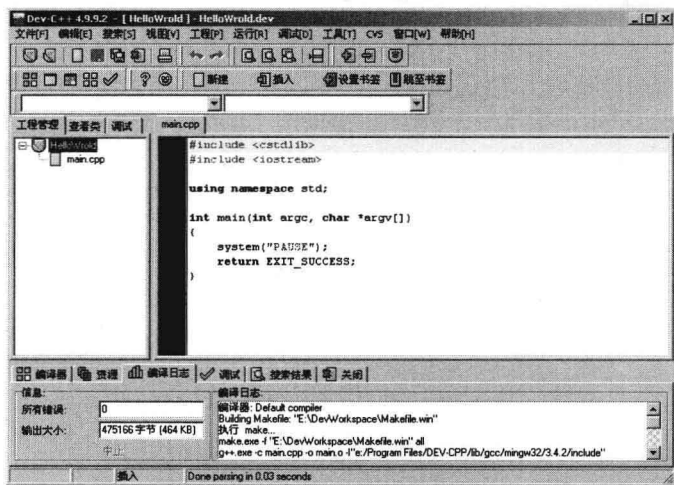


图 1.4 Dev-C 开发环境

### 1.2.5 Eclipse

Eclipse 是 IBM 开发的、早期用来编写 Java 代码的编辑器，但由于 CDT 插件的出现，使 Eclipse

也可以用来编写 C/C++ 代码。同 Dev-C 一样, Eclipse 也使用开源项目下的 GCC.exe 和 G++.exe 来编译代码, 所以在使用 Eclipse 前一定要注意设置好相关路径。Eclipse 开发环境如图 1.5 所示。

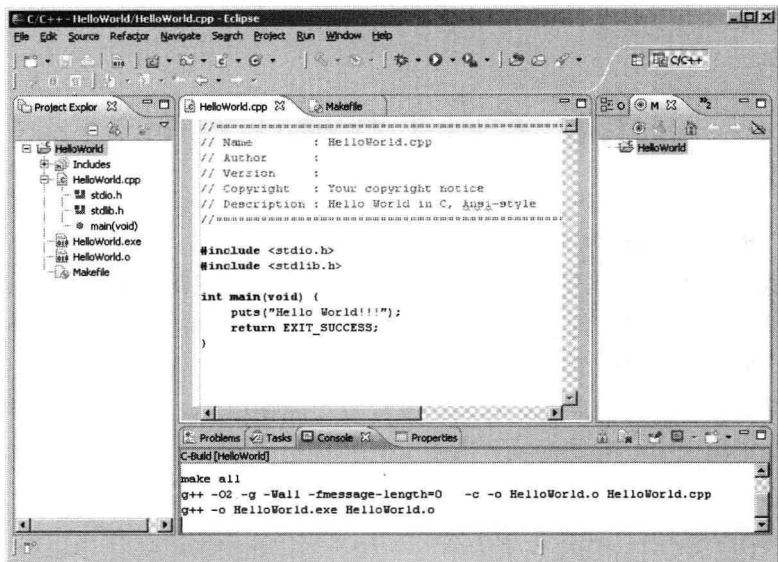


图 1.5 Eclipse 开发环境

### 1.3 认知 C++ 程序代码

 视频讲解: 光盘\TM\lx\1\认知 C++ 程序代码.exe

C++ 程序代码是由预编译指令、宏定义指令、注释、主函数、自定义函数等很多部分组成的, 这些部分都是后文讲述的主要内容。下面是一段很小但涉及 C++ 语言概念比较多的代码, 如图 1.6 所示。

```

#define options 宏定义指令
#ifdef options
#include <iostream.h>
/*
***** Sample.cpp
*/
/*
*****
*/
int ShowMessage(); 函数声明
int main(int argc, char* argv[]) 主函数
{
    int iResult;
    iResult=ShowMessage();//自定义函数ShowMessage
    if(iResult<0) 注释
        cout << "ShowMessage Error" << endl;
    return 0;
}
int ShowMessage() 自定义函数
{
    try
    {
        cout << "Hello World!" << endl;
        return 0;
    }
    catch(...) 捕捉错误代码
    {
        cout << "throw exception" << endl;
        throw "error occurred";
    }
}
#endif 预编译指令
    
```

图 1.6 代码介绍

图 1.6 所示的代码中含有头文件引用、函数作用空间、库函数调用、赋值运算、关系判断、流输出等很多 C++ 语言方面的概念，各概念之间通过一定规则罗列在一起，编译器会根据这些规则将代码编译成能够在机器上执行的应用程序。

## 1.4 C++ 工程项目文件

### 视频讲解：光盘\TM\lx\1\C++工程项目文件.exe

Windows 操作系统主要是用来管理数据的，而数据是以文件的形式存储在磁盘上的。文件可以通过扩展名来区分不同的类型，C++ 的代码文件就有两种类型，一种是源文件，一种是头文件。头文件中添加的是定义和声明函数部分，源文件中则是在头文件中定义函数的实现部分；源文件主要以 .cpp 为扩展名，而头文件主要以 .h 为扩展名。有的开发环境可能使用 .cxx、.chh 来作为源文件的扩展名。

对一个比较大的工程而言，它的源文件和头文件可能会比较多，为了管理这些源文件，不同的编译器还提供了管理代码的工程项目文件，不同开发环境的工程项目文件也会不同。

(1) Dev-C 的工程项目文件如图 1.7 所示。

- ☒ main.cpp: 源文件。
- ☒ Sample.dev: 工程文件。
- ☒ Makefile.win: make 程序执行时使用的文件，用于自动编译源代码文件。
- ☒ main.o: 编译后的目标文件。
- ☒ Sample.exe: 连接后生成的程序。

(2) VC 的工程项目文件如图 1.8 所示。

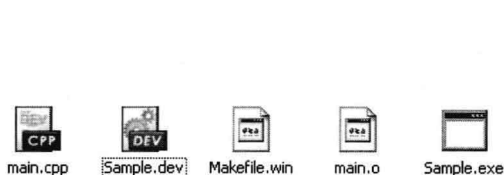


图 1.7 Dev-C 的工程项目文件

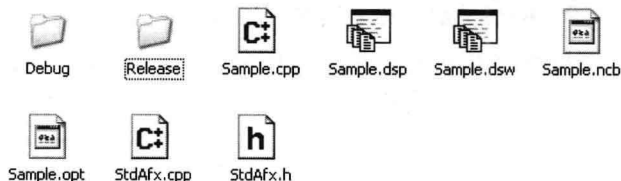


图 1.8 VC 的工程项目文件

- ☒ Debug: 存储编译后程序文件夹，带有调试信息的程序。
- ☒ Release: 存储编译后程序文件夹，最终程序。
- ☒ Sample.cpp: 源文件。
- ☒ Sample.dsp: VC 的工程文件。
- ☒ Sample.dsw: VC 的工作空间文件。
- ☒ Sample.ncb: VC 的用于声明的数据库文件。
- ☒ Sample.opt: VC 存储用户选项的文件。
- ☒ StdAfx.cpp: 向导生成的标准源文件，代码中涉及 MFC 类库内容时使用该文件。
- ☒ StdAfx.h: 向导生成的标准头文件。

**注意**

Debug 与 Release 的区别在于, Debug 是含有调试信息的应用程序, Debug 文件夹下的程序可以设置断点调试, 而且 Debug 文件夹下的程序要比 Release 文件夹下的程序大。

(3) Eclipse 的工程项目文件如图 1.9 所示。

- ☒ .cproject: Eclipse 工程相关信息文件。
- ☒ .project: Eclipse 工程文件。
- ☒ Makefile: make 程序执行时使用的文件, 用于自动编译源代码文件。
- ☒ Sample.cpp: 源文件。
- ☒ Sample.exe: 连接后生成的程序。
- ☒ Sample.o: 编译后的目标文件。



图 1.9 Eclipse 的工程项目文件

通过工程项目文件的扩展名, 就可以知道代码文件使用哪种开发环境管理。

## 1.5 使用 VC 创建程序

视频讲解: 光盘\TM\1\1\使用 VC 创建程序.exe

VC 可以通过两种方式创建 HelloWorld 程序, 一种是使用向导直接创建, 一种是创建空工程后, 手动向工程中添加源文件并写入代码。

### 1. 使用 Visual C++ 6.0 创建 Hello World! 程序

具体创建步骤如下:

(1) 启动 Visual C++ 6.0, 选择 File/New 命令, 弹出创建工程的向导, 如图 1.10 所示。

(2) 在列表中选择 Win32 Console Application 工程类型, 在 Project name 文本框中输入工程名 Sample, 在 Location 文本框中设置工程的保存路径为 D:\Sample。然后单击 OK 按钮, 弹出 Win32 Console Application-Step 1 of 1 对话框, 如图 1.11 所示。

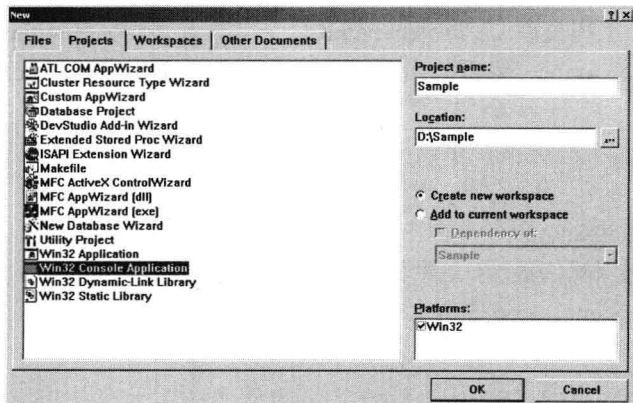


图 1.10 VC 创建工程向导

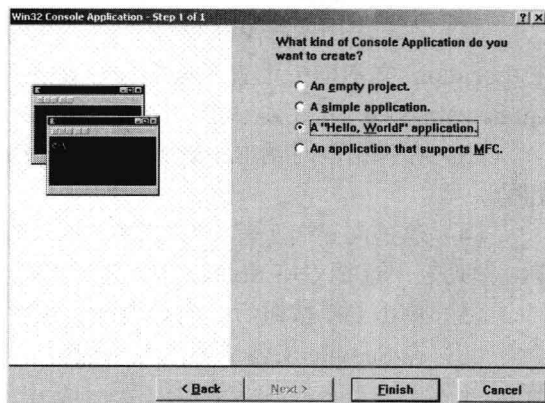


图 1.11 VC 工程向导第一步对话框



(3) 使用向导可以创建 4 种类型的工程。

- ☑ An empty project: 创建一个空的工程，工程中没有任何源文件和头文件。
- ☑ A simple application: 创建的工程中含有两个源文件（Sample.cpp 和 StdAfx.cpp）和一个头文件（StdAfx.h），并且 Sample.cpp 源文件中有一个不做任何操作的 main 函数。
- ☑ A "Hello,World!" application: 创建的工程中也含有两个源文件（Sample.cpp 和 StdAfx.cpp）和一个头文件（StdAfx.h），但 Sample.cpp 源文件中的 main 函数有一条输出“Hello World!”字符的 printf 语句。
- ☑ An application that supports MFC: 创建了支持 MFC 类库的工程。MFC 类库由微软开发，使用 MFC 类库可以加快程序开发的速度。

(4) 选择 A "Hello,World!" application 工程类型，单击 Finish 按钮，向导会创建能够在控制台输出“Hello World!”字符串的应用程序。创建完的工程如图 1.12 所示。

(5) 此时通过 Build/Execute 菜单执行应用程序就可以看到程序运行结果，如图 1.13 所示。

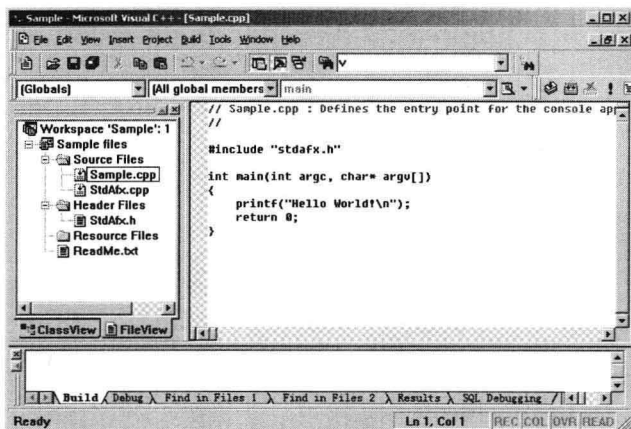


图 1.12 VC 开发环境

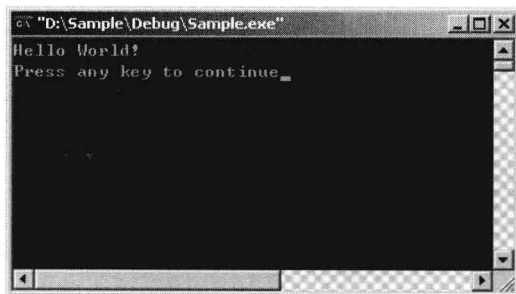


图 1.13 程序运行结果

## 2. 创建空工程，手动添加代码文件实现在控制台输出字符串“Hello World!”

具体创建步骤如下：

- (1) 启动 Visual C++ 6.0，选择 File/New 命令，弹出创建工程的向导。
- (2) 在列表中选择 Win32 Console Application 工程类型，在 Project name 命令中输入工程名 Sample，在 Location 文本框中设置工程的保存路径为 D:\Sample。然后单击 OK 按钮，弹出 Win32 Console Application-Step 1 of 1 对话框。
- (3) 在弹出的对话框中选择 An empty project 工程类型，单击 Finish 按钮，向导会创建一个空的工程。
- (4) 通过向导向工程中添加源文件。选择 File/New 命令，弹出创建工程的向导，选择 Files 选项卡，在列表中选择 C++ Source File 选项，在 File 文本框中输入文件名 sample，如图 1.14 所示。
- (5) 单击 OK 按钮后，向导会向工程中添加 Sample.cpp 文件。
- (6) 在 Sample 文件中输入如下代码：

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

```
void main()
{
    cout << "Hello World!" << endl;
}
```

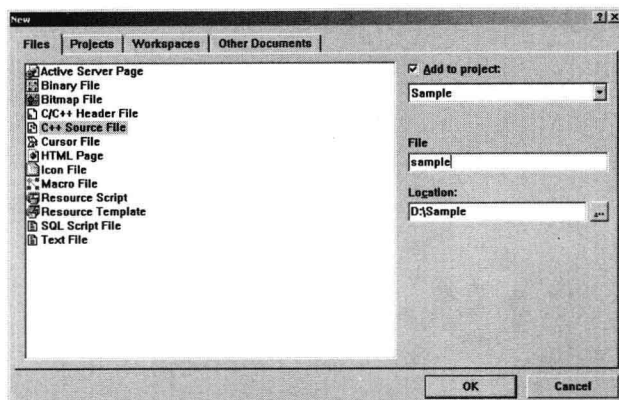


图 1.14 添加文件对话框

(7) 通过 Build/Execute 菜单执行应用程序即可看到程序的运行结果。

## 1.6 编译与连接过程

 视频讲解：光盘\TM\lx\1\编译与连接过程.exe

开发应用程序可以分为编辑、编译、连接、执行 4 个步骤。

### 1. 编辑

编辑就是在文本编辑器中输入代码，并对代码字符进行增、删、改，然后将输入的内容保存成文件。如图 1.15 所示，输入 Hello World 程序代码，并将代码保存成 Sample.cpp 文件。

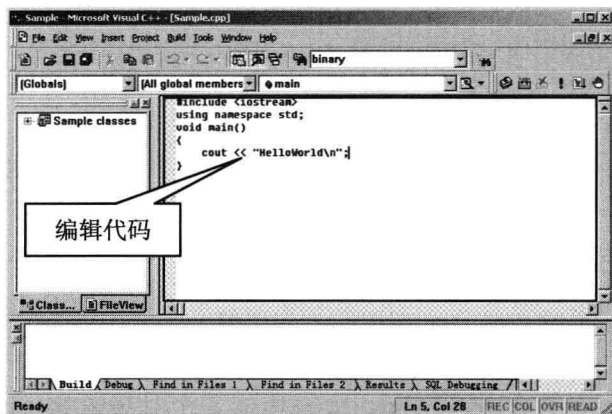


图 1.15 编辑代码

## 2. 编译

编译就是将代码文件编译成目标文件。如图 1.16 所示，编译过程就是将 Sample.cpp 编译成 Sample.obj。

在 VC 开发环境中，单击编译按钮后 VC 开发环境对输入的代码进行编译，编译按钮如图 1.17 所示。



图 1.16 编译文件

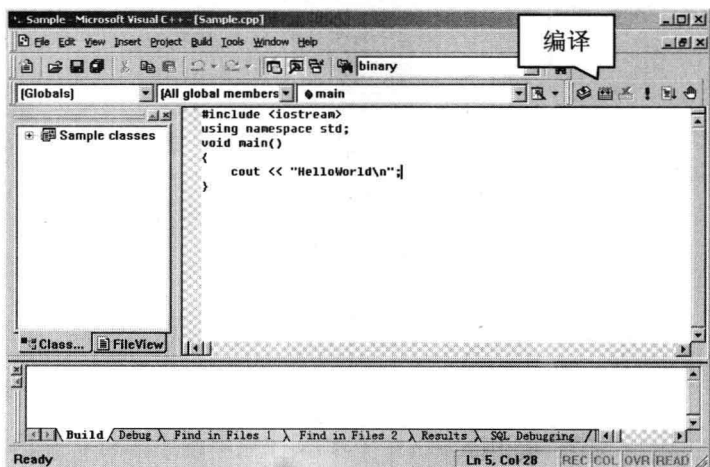


图 1.17 执行编译命令

单击编译按钮后 VC 开发环境自动对代码进行编译和连接，整个编译过程如图 1.18 所示。

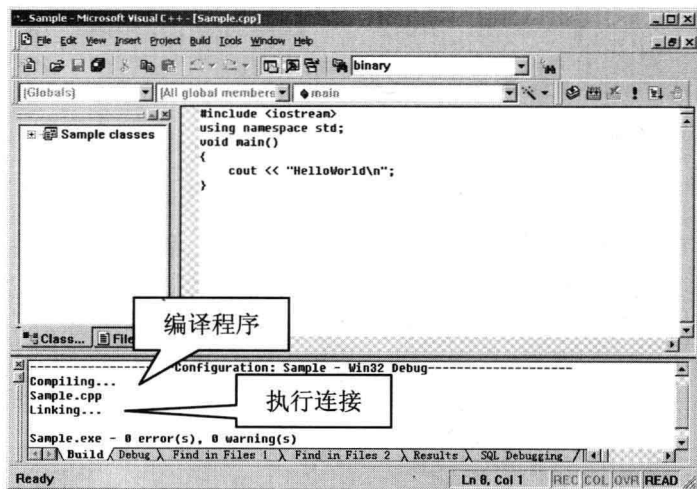


图 1.18 编译过程

## 3. 连接

连接就是将编译后的目标文件连接成可执行的应用程序。如将 Sample.obj 和 lib 库文件连接成 Sample.exe 可执行程序。lib 库是编译好的提供给用户使用的目标模块，在有多个源文件的工程中，例如 Sample1.cpp、Sample2.cpp、Sample3.cpp，会编译成多个目标模块 Sample1.obj、Sample2.obj、Sample3.obj，链接器会将程序所涉及的目标模块连接成可执行程序，如图 1.19 所示。

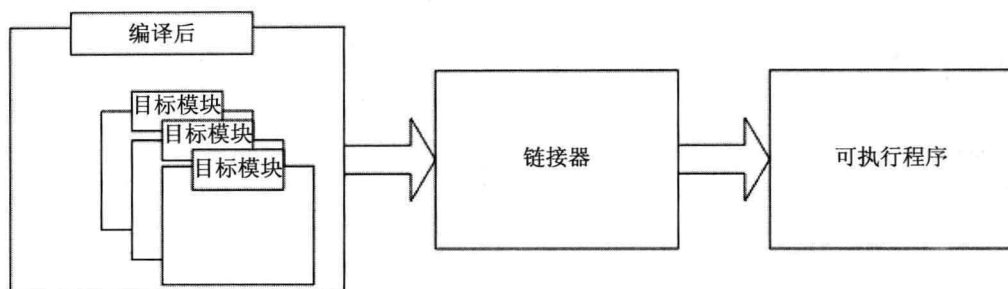


图 1.19 连接过程

#### 4. 执行

执行就是执行生成的应用程序。VC 开发环境下集成了运行按钮，单击运行按钮后开发环境自动执行生成的程序，运行按钮如图 1.20 所示。

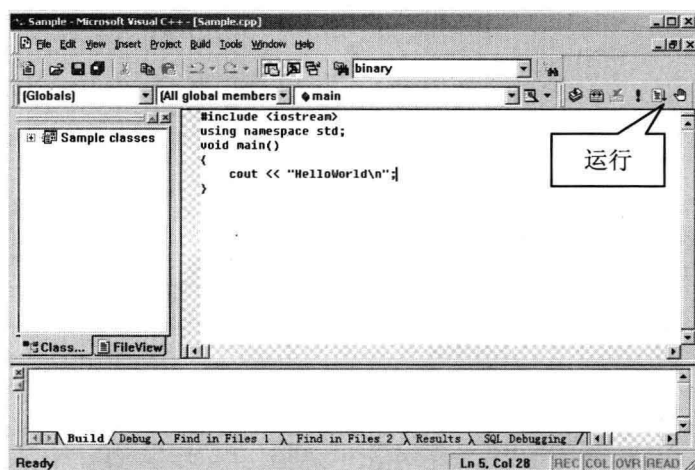


图 1.20 运行按钮

## 1.7 C++的特点

视频讲解：光盘\TM\1\1\C++的特点.exe

C++语言的运算符十分丰富，共有 30 多个，有算术、关系、逻辑、位、赋值、指针、条件、逗号、下标、类型转换等多种类型。

C++语言的数据结构多样，有整型、实型、字符型、枚举类型等基本类型，有数组、结构体、共用体等构造类型以及指针类型，还为用户提供了自定义数据类型，能够实现复杂的数据结构，还可以定义类实现面向对象编程，类和指针结合可以实现高效的程序。

C++语言的控制语句形式多样、使用方便。有两路分支、多路分支和虚幻结构等控制语句，便于结构化模块的实现和控制，结合面向对象编程便于程序的编制和维护。

C++语言是一种面向对象的程序设计语言，采用抽象和实际相结合的方式，各对象间使用消息进行通信，对象通过继承方法增加了代码的复用率。

C++语言继承了 C 语言的特性，可以直接访问地址，进行位运算，从而能对硬件进行操作。C++语句具有编写简单方便、便于理解的优点，还具有低级语言的与硬件结合紧密的优点。

C++语句具有很强的移植性，用 C++编写的程序基本不用太多修改就可用于不同型号的计算机上，C++标准可在多种操作系统下使用。


## 1.8 小 结

任何编程语言都有它的时代性，都是不断发展的。C++现在是一个成熟的语言，首先要理解 C++大师新的编程理念，然后选择自己喜欢的开发环境，可以选择微软的 Visual C++，也可以选择 Dev-C 和 Eclipse。

# 第 2 章

---

## 数据类型

(  视频讲解：1 小时 26 分钟 )

数据类型是 C++ 语言的基础，要学习一门编程语言首先要掌握它的数据类型，不同的数据类型占用不同的内存空间，合理定义数据类型可以优化程序的运行。本章将介绍数据类型及数据类型的输出。

通过阅读本章，您可以：

- » 了解 C++ 程序的各部分
- » 掌握数据类型的分类
- » 掌握变量和常量的使用
- » 了解数据的输入与输出

## 2.1 第一个 C++ 程序

 视频讲解：光盘\TM\lx\2\第一个 C++ 程序.exe

学习编程的第一步是先写一个最简单的程序。学习任何编程语言都需要写一个“HelloWorld”程序，下面是最简单的 C++ 程序，同样也是一个 HelloWorld 程序。

```
#include<iostream>
using namespace std;
void main()
{
    cout << "HelloWorld\n";
}
```

最简单的程序输出结果如图 2.1 所示。

最简单的 C++ 程序中包含了头文件引用、应用命名空间、主函数、字符串常量、数据流等几部分，这些都是 C++ 程序中经常用到的。下面对 C++ 常用的概念进行介绍。

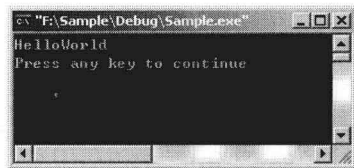


图 2.1 第一个 C++ 程序

### 2.1.1 #include 指令

C++ 的程序中带“#”号的语句被称为宏定义或预编译指令，关于什么是 C++ 中的语句，什么是宏定义或预编译指令会在后面章节讲到。#include 在代码中是包含和引用的意思，第一行代码“#include <iostream>”就是说明代码要引用 iostream 文件内容，编译器在编译程序时会将 iostream 中的内容在“#include <iostream>”处展开。

### 2.1.2 注释

代码注释是禁止语句的执行，编译器不会对注释的语句进行编译。C++ 中有两种注释方法，其中“//”是单行注释，单行注释只能注释符号“//”后面的内容，到本行代码结束的位置结束；“/\*\*/”是多行注释，多行注释的使用方法是符号“/\*”放在将要注释代码的前面，符号“\*/”放在将要注释代码的末尾，符号“/\*”和“\*/”中间的内容就会被注释，另外多行注释中不允许嵌套多行注释，例如 /\*/\*\*\*/，最后出现的“\*/”符号将会无效。在第一个 C++ 程序中加入注释，代码如下：

```
/*sample.cpp*/
#include<iostream>           //头文件引用
using namespace std;        //命名空间
void main()                  //主函数
{
    cout << "HelloWorld\n"; //执行输出
}
```

```
//cout << "end";  
}
```

注释不仅是在调试时使用，开发人员也可以在代码中加入注释，用来说明代码的用意，这样可以方便日后自己或别人查看。

### 2.1.3 main 函数

单词 `main` 代表主函数的意思，`main` 函数是程序执行的入口，程序从 `main` 函数的第一条指令开始执行，直到 `main` 函数结束，整个程序也将执行结束。注意函数的格式单词 `main` 后面有个小括号“`()`”，小括号内是放参数的地方。函数相关的内容将在后面章节讲到。

### 2.1.4 函数体

大括号“`{}`”中的内容是需要执行的内容，称为函数体，函数体是按代码的先后顺序执行的，写在前面的代码先执行，写在后面的代码后执行。代码“`cout << "HelloWorld\n";`”表示通过输出流输出单词“`HelloWorld`”，单词 `HelloWorld` 两边的双引号代表单词是字符串常量，`cout` 表示输出流，`<<` 表示将字符串传送到输出流中。

### 2.1.5 函数返回值

单词 `void` 表示函数的返回值，函数的返回值是用来判断函数执行情况以及返回函数执行结果的。`void` 代表不返回任何数据。如果要返回数据还需要使用 `return` 语句。

## 2.2 数据类型

#### 视频讲解：光盘\TM\lx\2\数据类型.exe

计算机能够运行高低电平组成的二进制数据，开发人员编写的程序代码需要经过编译器转换成二进制数据，才能被计算机识别并执行，这种由编译器将代码转换成计算机能识别的二进制数据的过程称为编译过程。

能被计算机识别的二进制数据被称为机器码，汇编语言是最接近机器码的一种低级语言，汇编语言的编译器是将汇编代码直接解释成机器码，所谓解释就是一种对应关系，一条语句代表一个机器码。通过汇编代码就可以很清楚地了解程序在 CPU 的执行过程。

计算机的运算是通过 CPU 完成的，执行运算过程时首先需要将数据存放到 CPU 的寄存器中，然后 CPU 根据机器码指令执行运算。存放到 CPU 寄存器中的数据都是从内存中读取的，内存是存储数据的地方，程序运行时，被编译器编译后的二进制数据会被读取到计算机内存中，然后由 CPU 执行。



数据类型可以决定用多大的内存来存储用户的数据。汇编语言中没有数据类型这个概念，汇编语言中除了直接操作用户数据外，都是通过地址值来操作用户数据。在高级语言中通过地址查找用户数据的任务都由编译器来完成，开发人员不用亲自管理地址空间，不用思考哪些数据应该存储在哪块内存地址下。

C++是数据类型非常丰富的语言，常用的数据类型如图 2.2 所示。



图 2.2 C++常用数据类型

掌握 C++语言的数据类型和运算符是学习 C++语言的基础。下面将对不同数据类型进行介绍。

## 2.3 常量及符号

 视频讲解：光盘\TM\lx\2\常量及符号.exe

在程序运行过程中，其值不能改变的量称为常量。常量可分为整型常量、实型常量、字符常量和字符串常量。例如：

```

#include<iostream>
using namespace std;
void main()
{
    cout << 2009 << endl;
    cout << 3.14 << endl;
    cout << 'a' << endl;
    cout << "HelloWorld"<< endl;
}
  
```

上面代码通过 cout 向屏幕输出 4 行内容。cout 是输出流，实现向屏幕输出不同类型的数据。代码中 2009 是整型常量，3.14 是浮点型常量（实型常量），'a'是字符常量，"HelloWorld"是字符串常量。

### 2.3.1 整型常量

整型常量可以分为有符号整型常量和无符号整型常量。

-225 代表一个负数，+1024 代表一个正整数，正整数前面的“+”符号可以省略，即+1024 和 1024 表示的意义相同。

由于基本的数据类型里除了整型外，还有长整型和短整型，所以整型常量也有长整型常量和短整型常量之别。长整型常量不是可以无限大的，它的最大值是有限定的，根据 CPU 寄存器位数的不同以及编译器的不同，最大的整型常量值也会不同。



#### 注意

4294967295 是 32 位 CPU 寄存器以及 VC6 编译器所允许的最大正整数。

整型常量在编写代码时不仅可以写成十进制整数形式，也可以写成十六进制或八进制整数形式。

(1) 八进制形式整型常量必须以 0 开头，即以 0 作为八进制数的前缀，每位取值范围是 0~7。八进制数通常是无符号数。

以下各数是合法的八进制数：016、0101、0128。

以下各数不是合法的八进制数：256 无前缀 0，它代表十进制整型常量；0396 中数字 9 不是八进制应有的取值。

(2) 十六进制整型常量的前缀为 0X 或 0x，其数码取值范围为 0~9，以及 A~F 或 a~f。

以下各数是合法的十六进制整常数：0X2A1、0XC5、0XFFFF。

以下各数不是合法的十六进制整常数：5A 无前缀 0X；0X3N 中含有非十六进制数 N。



#### 注意

合法是指能通过编译器编译，非法则是不能通过编译器编译。

### 2.3.2 实型常量

实型常量也称为浮点数，只能采用十进制形式表示。它有两种表示形式，即小数表示法和指数表示法。

#### 1. 小数表示法

使用这种表示法，实型常量由整数部分和小数部分组成，整数部分和小数部分每位取值范围是 0~9，中间用小数点分隔。例如，0.0、3.25、0.00596、5.0、536.、-5.3、-0.002 等均为合法的实型常量。

整数部分和小数部分有时可以不必同时出现，例如 2 和 2.。

#### 2. 指数表示法

指数表示法也称科学记数法，指数部分以符号“e”或“E”开始，但必须是整数，并且符号“e”

或“E”两边都必须有一个数，例如 1.2e20 和-3.4e-2。

以下不是合法的实型常量：E5 是 E 之前无数字；3E3.5 是 E 后面有小数。



**说明**

在字母 e（或 E）之前的小数部分中，小数点左边应有一位（且只能有一个）非零的数字，成为规范化的指数形式。

科学记数法中 23e3 表示  $23 \times 10^3$ 。

L 代表是长整型。L 可以是大写也可以是小写，在编写代码时可以不写。此类的符号还有 U 和 u 代表无符号。例如，255U 或 255u 都代表无符号整型常量 255。

符号 L 或 l 与符号 U 或 u 可以一起使用。65536lu 代表无符号长整型常量 65536。

C++编译系统把用这种形式表示的浮点数按双精度常量处理，在内存中占 8 个字节。如果在实数的数字之后加字母 F 或 f，表示此数为单精度浮点数，如 1234F 和-43f，占 4 个字节。如果加字母 L 或 l，表示此数为长双精度数（long double）。

### 2.3.3 字符常量

字符常量是用单引号括起来的一个字符，例如 ‘a’ 和 ‘?’ 都是合法的字符常量。在对代码编译时，编译器会根据 ASCII 码表将字符常量转换成整型常量。字符 ‘a’ 的 ASCII 码值是 97，字符 ‘A’ 的 ASCII 码值是 41，字符 ‘?’ 的 ASCII 码值是 63。ASCII 码表中还有很多通过键盘无法输入的字符，可以使用 ‘\ddd’ 或 ‘\xhh’ 来引用这些字符。可以使用 ‘\ddd’ 或 ‘\xhh’ 来引用 ASCII 码表中的所有字符。ddd 是 1~3 位八进制数所代表的字符，\xhh 是 1~2 位十六进制数所代表的字符。例如 ‘\101’ 表示 ASCII 码 “A”，‘\XOA’ 表示换行等。

下面是转义字符应用的实例。

```
#include<iostream>
void main()
{
    std::cout << "A" <<std::endl;
    std::cout << "\101" <<std::endl;
    std::cout << "\x41" <<std::endl;
    std::cout << "\052,\x1E" <<std::endl;
}
```

实例运行结果如图 2.3 所示。

转义字符是特殊的字符常量，使用时以字符 ‘\’ 代表开始转义，和后面不同的字符表示转义后的字符。转义字符如表 2.1 所示。



图 2.3 运行结果

表 2.1 转义字符说明

转义字符	意 义	ASCII 代码
\0	空字符	0
\n	换行	10
\t	水平制表	9

续表

转义字符	意 义	ASCII 代码
\b	退格	8
\r	回车	13
\f	换页	12
\\	反斜杠	93
\'	单引号字符	39
\"	双引号字符	34
\a	响铃	7

### 2.3.4 字符串常量

字符串常量是由一对双引号括起来的零个或多个字符序列，例如“welcome to our school”、“hello girl”。“”可以表示一个空字符串。

同样，''可以表示空字符，而 NULL 是一种特殊的数据类型，表示空的意思。有的编译器把它编译成零，有的则编译成其他值。

字符串常量实际上是一个字符数组，可以将字符串分解成若干个字符，字符的数量是字符串的长度。字符串常量一般都是用来给字符数组变量赋值或是直接作为实参传递，为告知编译器字符串已经结束。一般在给字符数组赋初值时在字符串的末尾加上字符 '\0'，表示字符结束，如果不加字符结束标志，有可能会出现意想不到的错误。

字符常量 'A' 与字符串常量 "A" 是不同的，字符串常量 "A" 是由 'A' 和 '\0' 两个字符组成的，字符串的长度是 2，字符常量 'A' 只是一个字符，没有长度。

### 2.3.5 其他常量

前文讲到的都是普通的常量，常量还包括布尔常量、枚举常量、宏定义常量等。

- ☑ 布尔 (bool) 常量：布尔常量只有两个，一个是 true，表示真；一个是 false，表示假。
- ☑ 枚举常量：枚举型数据中定义的成员也都是常量，这将在后文介绍。
- ☑ 宏定义常量：通过 #define 宏定义的一些值也是常量。例如：

```
#define PI 3.1415
```

其中 PI 就是常量。

## 2.4 变 量

 视频讲解：光盘\TM\lx\2\变量.exe

前面已经介绍了常量，知道常量的值是不可改变的，所以称为常量，那么值可以改变的量是否叫

变量呢？答案是肯定的，本节将介绍变量的相关知识。

### 2.4.1 标识符

标识符（identifier）是用来对 C++ 程序中的常量、变量、语句标号以及用户自定义函数的名称进行标识的符号。

标识符命名规则：

- （1）由字母、数字及下划线组成，且不能以数字开头。
- （2）大写和小写字母代表不同意义。
- （3）不能与关键字同名。
- （4）尽量“见名知义”，应该受一定规范的约束。

如 6A、ABC\*（不能使用\*）、case（是保留字）都是不合法的标识符。

C++ 有许多保留关键字，如表 2.2 所示。

表 2.2 C++保留关键字

asm	auto	break	case	catch	char	class	const	continue
default	delete	do	double	else	enum	extern	float	for
friend	goto	if	inline	int	long	new	operator	overload
private	protected	public	register	return	short	signed	sizeof	static
struct	switch	this	template	throw	try	typedef	union	unsigned
virtual	void	volatile	while					

### 2.4.2 变量与变量说明

变量是指程序在运行时其值可改变的量。每个变量都由一个变量名标识，每个变量又具有一个特定的数据类型。

变量在使用之前一定要定义或说明，变量声明的一般形式如下：

**[修饰符] 类型 变量名标识符；**

类型是变量类型的说明符，说明变量的数据类型。修饰符是任选的，可以没有。

多个同一类型的变量可以在一行中声明，不同变量名用逗号运算符隔开。例如：

```
int a,b,c;
```

与

```
int a;
int b;
int c;
```

两者等价。

### 2.4.3 整型变量

整型变量可以分为短整型、整型和长整型，变量类型说明符分别是 `short`、`int`、`long`。根据是否有符号还可分为以下 6 种：

整型	[signed] int
无符号整型	unsigned [int]
有符号短整型	[signed] short [int]
无符号短整型	unsigned short [int]
有符号长整型	[signed] long [int]
无符号长整型	unsigned long [int]

方括号中的关键字表示可以省略，例如 `[signed] int` 可以写成 `int`。

短整型 `short`，在内存中占用两个字节的空間，可以表示的数值范围是  $-32768 \sim 32767$ ，如果是无符号短整型 `unsigned short`，表示的数值范围是  $0 \sim 65535$ 。整型 `int` 占用 4 个字节的空間，有符号整型表示的数值范围是  $-2147483648 \sim 2147483648$ ，无符号整型 `unsigned int` 表示的数值范围是  $0 \sim 4294967295$ 。长整型与整型占用的字节数相同，表示的数值范围也相同，具体如表 2.3 所示。

表 2.3 整型变量范围

关 键 字	类 型	数 值 范 围	字 节 数
<code>short</code>	短整型	$-32768 \sim 32767$ ，即 $-2^{15} \sim 2^{15}-1$	2
<code>unsigned short</code>	无符号短整型	$0 \sim 65535$ ，即 $0 \sim 2^{16}-1$	2
<code>int</code>	整型	$-2147483648 \sim 2147483648$ ，即 $-2^{31} \sim 2^{31}-1$	4
<code>unsigned int</code>	无符号整型	$0 \sim 4294967295$ ，即 $0 \sim 2^{32}-1$	4
<code>long int</code>	长整型	$-2147483648 \sim 2147483648$ ，即 $-2^{31} \sim 2^{31}-1$	4
<code>unsigned long</code>	无符号长整型	$0 \sim 4294967295$ ，即 $0 \sim 2^{32}-1$	4

### 2.4.4 实型变量

实型变量又可称为浮点型变量，可分为单精度（`float`）、双精度（`double`）和长双精度（`long double`）3 种。

Visual C++ 6.0 中，对 `float` 提供 6 位有效数字，对 `double` 提供 15 位有效数字，并且 `float` 和 `double` 的数值范围不同。对 `float` 分配 4 个字节，对 `double` 和 `long double` 分配 8 个字节。

#### 1. 单精度

类型说明符为 `float`，该实型数据在内存中占 4 个字节，表示的数值范围是  $-3.4e38 \sim 3.4e38$ 。例如：

```
float a;
```

#### 2. 双精度

类型说明符为 `double`，该实型数据在内存中占 8 个字节，表示的数值范围是  $-1.7e308 \sim 1.7e308$ 。

例如：

```
double b;
```

### 3. 长双精度

类型说明符为 `long double`，该实型数据在内存中占 10 个字节，表示的数值范围是  $-1.1\text{e}4932 \sim 1.1\text{e}4932$ 。

例如：

```
long double c;
```

## 2.4.5 变量赋值

变量值是动态改变的，每次改变都需要进行赋值运算。变量赋值的形式如下：

```
变量名标识符 = 表达式
```

变量名标识符就是声明变量时定义的，表达式将在后面的章节中讲到。例如：

```
int i;    //声明变量  
i=100;   //给变量赋值
```

声明 `i` 是一个整型变量，100 是一个常量。

```
int i,j;  //声明变量  
i=100;   //给变量赋值  
j=i;     //将一个变量的值赋给另一个变量
```

## 2.4.6 变量赋初值

可以在声明变量时就把数值赋给变量，这个过程叫做变量赋初值，赋初值的情况有以下几种：

(1) `int x=5;`

表示定义 `x` 为有符号的基本整型变量，赋初值为 5。

(2) `int x,y,z=6;`

表示定义 `x`、`y`、`z` 为有符号的基本整型变量，`z` 赋初值为 6。

(3) `int x=3,y=3,z=3;`

表示定义 `x`、`y`、`z` 为有符号的基本整型变量，且赋予的初值均为 3。



### 注意

定义变量并赋初值时可以写成 “`int x=3,y=3,z=3;`”，但不可写成 “`int a=b=c=3;`” 这种形式。



### 2.4.7 字符变量

字符变量的类型说明符为 `char`，一个字符变量占用 1 字节内存单元。例如，“`char ch1;`” 定义一个字符变量 `ch1`；“`ch1='a';`” 给字符变量赋值。

字符变量值在内存中存储的是字符的 ASCII 码，即一个无符号整数，其形式与整型变量的存储形式一样，字符型数据与整型数据之间通用。

(1) 一个字符型数据，既可以字符形式输出，也可以整数形式输出。

**【例 2.1】** 字符型数据与整型数据间运算。(实例位置：光盘\TM\sl\2\1)

```
#include<iostream>
using namespace std;
void main()
{
    char c1,c2;
    c1='a';
    c2='b';
    int i1,i2;
    printf("%c,%d\n%c,%d",c1,c1,c2,c2);
}
```

程序运行结果如图 2.4 所示。

(2) 允许对字符数据进行算术运算，此时就是对它们的 ASCII 码值进行算术运算。

**【例 2.2】** 字符型数据进行算术运算。(实例位置：光盘\TM\sl\2\2)

```
#include<iostream>
using namespace std;
void main()
{
    char ch1,ch2;
    ch1='a'; ch2='B';
    printf("ch1=%c,ch2=%c\n",ch1-32,ch2+32); //给 ch1、ch2 赋值
    printf("ch1+10=%d\n", ch1+10); //用字符形式输出一个大于 256 的数值
    printf("ch1+10=%c\n", ch1+10);
    printf("ch2+10=%d\n", ch2+10);
    printf("ch2+10=%c\n", ch2+10);
}
```

程序运行结果如图 2.5 所示。

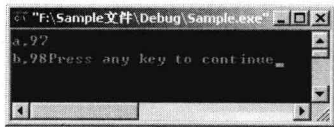


图 2.4 字符型数据与整型数据间运算

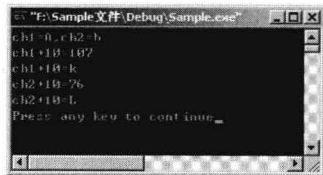


图 2.5 字符型数据进行算术运算



## 2.5 数据输入与输出

 视频讲解：光盘\TM\lx\2\数据输入与输出.exe

在用户与计算机进行交互的过程中，数据输入和数据输出是必不可少的操作过程，计算机需要通过输入获取来自用户的操作指令，并通过输出来显示操作结果。本节将介绍数据输入与输出的相关内容。

### 2.5.1 控制台屏幕

在 NT 内核的 Windows 操作系统中为保留 DOS 系统的风格，提供了控制台程序，通过系统的“开始”→“运行”菜单，运行 cmd.exe 程序，可以启动控制台程序。在控制台可以运行 DIR、CD、DELETE 等 DOS 系统中的文件操作命令，也可以用来启动 Windows 的程序。控制台屏幕如图 2.6 所示。

使用 Visual C++ 6.0 创建的控制台工程的程序都将运算结果输出到这个控制台屏幕上，它是程序显示输出结果的地方。

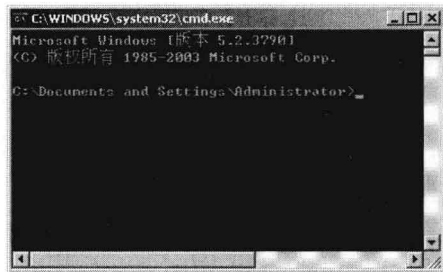


图 2.6 控制台

### 2.5.2 C++语言中的流

在 C++ 语言中，数据的输入和输出包括标准输入/输出设备（键盘、显示器）、外部存储介质（磁盘）上的文件和内存的存储空间 3 个方面的输入/输出。对标准输入/输出设备的输入/输出简称为标准 I/O，对在外存磁盘上文件的输入/输出简称为文件 I/O，对内存中指定的字符串存储空间的输入/输出简称为串 I/O。

C++ 语言中把数据之间的传输操作称为流。C++ 中的流既可以表示数据从内存传送到某个载体或设备中，即输出流；也可以表示数据从某个载体或设备传送到内存缓冲区变量中，即输入流。C++ 中的所有流都是相同的，但文件可以不同（文件流会在后面讲到）。使用流以后，程序用流统一对各种计算机设备和文件进行操作，使程序与设备、文件无关，从而提高了程序设计的通用性和灵活性。

C++ 语言定义了 I/O 类库供用户使用，标准 I/O 操作有 4 个类对象，分别是 cin、cout、cerr 和 clog。其中 cin 代表标准输入设备键盘，也称为 cin 流或标准输入流。cout 代表标准输出显示器，也称为 cout 流或标准输出流，当进行键盘输入操作时使用 cin 流，当进行显示器输出操作时使用 cout 流，当进行错误信息输出操作时使用 cerr 或 clog 流。

C++ 的流通过重载运算符“<<”和“>>”执行输入和输出操作。输出操作是向流中插入一个字符序列，因此，在流操作中，将左移运算符“<<”称为插入运算符。输入操作是从流中提取一个字符序列，因此，将右移运算符“>>”称为提取运算符。

### 1. cout 语句的一般格式

```
cout<<表达式 1<<表达式 2<<.....<<表达式 n;
```

cout 代表显示器，执行 cout << x 操作就相当于把 x 的值输出到显示器。

先把 x 的值输出到显示器屏幕上，在当前屏幕光标位置显示出来，然后 cout 流恢复到等待输出的状态，以便继续通过插入操作符输出下一个值。当使用插入操作符向一个流输出一个值后，再输出下一个值时将被紧接着放在上一个值的后面，所以为了让流中前后两个值分开，可以在输出一个值后接着输出一个空格，或一个换行符，或是其他所需要的字符或字符串。

一个 cout 语句可以分写成若干行。例如：

```
cout<< "Hello World!" <<endl;
```

可以写成：

```
cout<< "Hello" //注意行末尾无分号
<<" "
<<"World!"
<<endl;      //语句最后有分号
```

也可写成多个 cout 语句：

```
cout<< "Hello"; //语句末尾有分号
cout <<" ";
cout <<"World!.";
cout<<endl;
```

以上 3 种情况的输出均正确。

### 2. cin 语句的一般格式

```
cin>>变量 1>>变量 2>>.....>>变量 n;
```

cin 代表键盘，执行 cin>>x 就相当于把键盘输入的数据赋值给变量。

当从键盘上输入数据时，只有当输入完数据并按下 Enter 键后，系统才把该行数据存入到键盘缓冲区，供 cin 流顺序读取给变量。另外，从键盘上输入的每个数据之间必须用空格或回车符分开，因为 cin 为一个变量读入数据时是以空格或回车符作为其结束标志的。

当 cin>>x 操作中的 x 为字符指针类型时，则要求从键盘的输入中读取一个字符串，并把该字符串赋值给 x 所指向的存储空间，若 x 没有事先指向一个允许写入信息的存储空间，则无法完成输入操作。另外，从键盘上输入的字符串，其两边不能带有双引号定界符，若有则只作为双引号字符看待。对于输入的字符也是如此，不能带有单引号定界符。

cin 函数相当于 c 库函数中的 scanf，将用户的输入赋值给变量。例如：

```
#include<iostream.h>
void main()
{
```

```

int ilnput;
cout << "Please input a number:" << endl;
cin >> ilnput;
cout << "the number is:" << ilnput << endl;
}

```

实例将用户输入的数打印出来。

**【例 2.3】** 简单输出字符。（实例位置：光盘\TM\sl\2\3）

```

#include<iostream.h>
void main()
{
    int i=0;
    cout << i << endl;
    cout << "HelloWorld" << endl;
}

```

运行程序，将向控制台屏幕输出变量 *i* 的值和 HelloWorld 字符串，运行结果如图 2.7 所示。

endl 是向流的末尾部位加入换行符。*i* 是一个整型变量，在输出流中自动将整型变量转换成字符串输出。

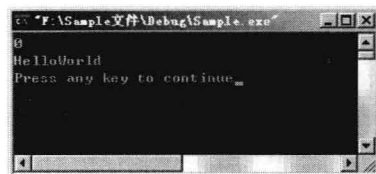


图 2.7 向控制台屏幕输出 HelloWorld 字符串

### 2.5.3 流操作的控制

在头文件 `iomanip.h` 中定义了一些控制流输出格式的函数，默认情况下整型数按十进制形式输出，也可以通过 `hex` 将其设置为十六进制输出。流操作的控制的具体函数如下。

#### (1) `long setf(long f);`

根据参数 *f* 设置相应的格式标志，返回此前的设置。该参数 *f* 所对应的实参为无名枚举类型中的枚举常量（又称格式化常量），可以同时使用一个或多个常量，每两个常量之间要用按位或操作符连接。如需要左对齐输出，并使数值中的字母大写时，则调用该函数的实参为 `ios::left|ios::uppercase`。

#### (2) `long unsetf(long f);`

根据参数 *f* 清除相应的格式化标志，返回此前的设置。如果要清除此前的左对齐输出设置，恢复默认的右对齐输出设置，则调用该函数的实参为 `ios::left`。

#### (3) `int width();`

返回当前的输出域宽。若返回数值为 0，则表明没为刚才输出的数值设置输出域宽。输出域宽是指输出的值在流中所占有的字节数。

#### (4) `int width(int w);`

设置下一个数据值的输出域宽为 *w*，返回为输出上一个数据值所规定的域宽，若无规定则返回 0。注意，此设置不是一直有效，而只是对下一个输出数据有效。

#### (5) `setiosflags(long f);`

设置 *f* 所对应的格式标志，功能与 `setf(long f)` 成员函数相同，当然，在输出该操作符后返回的是一个输出流。如果采用标准输出流 `cout` 输出它时，则返回 `cout`。输出每个操作符后都是如此，即返回输

出它的流，以便向流中继续插入下一个数据。

(6) `resetiosflags(long f);`

清除 `f` 所对应的格式化标志，功能与 `unsetf(long f)` 成员函数相同。输出后返回一个流。

(7) `setfill(int c);`

设置填充字符的 ASCII 码为 `c` 的字符。

(8) `setprecision(int n);`

设置浮点数的输出精度为 `n`。

(9) `setw(int w);`

设置下一个数据的输出域宽为 `w`。

数据输入/输出的格式控制还有更简便的形式，就是使用头文件 `iomanip.h` 中提供的操作符。使用这些操作符不需要调用成员函数，只要把它们作为插入操作符的输出对象即可。

- ☑ `dec`: 转换为按十进制输出整数，是默认的输出格式。
- ☑ `oct`: 转换为按八进制输出整数。
- ☑ `hex`: 转换为按十六进制输出整数。
- ☑ `ws`: 从输出流中读取空白字符。
- ☑ `endl`: 输出换行符 `\n` 并刷新流。刷新流是指把流缓冲区的内容立即写入到对应的物理设备上。
- ☑ `ends`: 输出一个空字符 `\0`。
- ☑ `flush`: 只刷新一个输出流。

**【例 2.4】** 控制打印格式程序。(实例位置：光盘\TM\sl\2\4)

```
#include<iostream>
#include<iomanip>
using namespace std;
void main()
{
    double a=123.456789012345;
    cout << a << endl;
    cout << setprecision(9) << a << endl;
    cout << setprecision(6); //恢复默认格式（精度为 6）
    cout << setiosflags(ios::fixed);
    cout << setiosflags(ios::fixed) << setprecision(8) << a << endl;
    cout << setiosflags(ios::scientific) << a << endl;
    cout << setiosflags(ios::scientific) << setprecision(4) << a << endl;
}
```

程序运行结果如图 2.8 所示。

**【例 2.5】** 整数输出的实例。(实例位置：光盘\TM\sl\2\5)

```
#include<iostream>
#include<iomanip>
using namespace std;
void main()
{
    int b=123456; //对 b 赋初值
```

```

cout << b << endl;           //输出: 123456
cout << hex << b << endl;     //输出: 1e240
cout << setiosflags(ios::uppercase) << b << endl; //输出: 1E240
cout << setw(10) << b << ',' << b << endl;       //输出: 123456, 123456
cout << setfill('*') << setw(10) << b << endl;     //输出: **** 123456
cout << setiosflags(ios::showpos) << b << endl;     //输出: +123456
}

```

程序运行结果如图 2.9 所示。

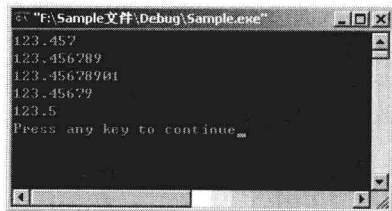


图 2.8 控制打印格式程序

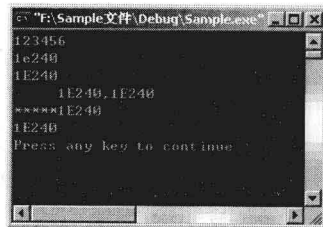


图 2.9 整数输出

**【例 2.6】** 输出大写的十六进制。（实例位置：光盘\TM\sl\2\6）

```

#include<iostream.h>
#include<iomanip.h>
void main()
{
    int i=0x2F,j=255;
    cout << i << endl;
    cout << hex << i << endl;
    cout << hex << j << endl;
    cout << hex << setiosflags(ios::uppercase) << j << endl;
}

```

程序运行结果如图 2.10 所示。

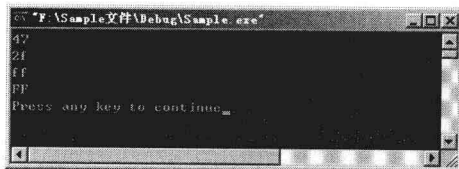


图 2.10 输出大写的十六进制

**【例 2.7】** 控制输出精确度。（实例位置：光盘\TM\sl\2\7）

```

#include<iostream>
using namespace std;
void main()
{
    int x=123;
    double y=-3.1415;
    cout << "x=";
}

```



```

cout.width(10);
cout << x;
cout << "y=";
cout.width(10);
cout << y << endl;
cout.setf(ios::left);
cout << "x=";
cout.width(10);
cout << x;
cout << "y=";
cout << y << endl;
cout.fill('*');
cout.precision(4);
cout.setf(ios::showpos);
cout << "x=";
cout.width(10);
cout << x;
cout << "y=";
cout.width(10);
cout << y << endl;
}

```

程序运行结果如图 2.11 所示。

【例 2.8】 流输出小数控制。(实例位置：光盘\TM\sl\2\8)

```

#include<iostream>
using namespace std;
void main()
{
    float x=20,y=-400.00;
    cout << x << ' ' << y << endl;
    cout.setf(ios::showpoint);           //强制显示小数点和无效 0
    cout << x << ' ' << y << endl;
    cout.unsetf(ios::showpoint);
    cout.setf(ios::scientific);          //设置按科学表示法输出
    cout << x << ' ' << y << endl;
    cout.setf(ios::fixed);               //设置按定点表示法输出
    cout << x << ' ' << y << endl;
}

```

程序运行结果如图 2.12 所示。

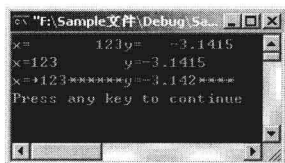


图 2.11 控制输出精确度

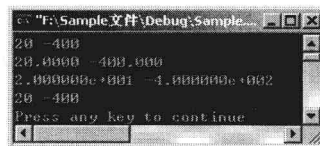


图 2.12 流输出小数控制

C++语言中还保留着C语言中的屏幕输出函数 `printf`。使用 `printf` 可以将任意数量、类型的数据输出到屏幕。`printf` 函数的声明形式如下：

```
printf("[控制格式]... [控制格式]...",数值列表);
```

函数 `printf` 是变参函数，数值列表中可以有多个数值，数值的个数不是确定的，每个数值之间用逗号运算符隔开；控制格式表示数值以哪种格式输出，控制格式的数量要与数值的个数一致，否则程序运行时会产生错误。

控制格式是由%+特定字符构成的，形式如下：

`%[*][域宽][长度]类型`

\*代表可以使用占位符，域宽表示输出的长度。如果输出的内容没有域宽长，用占位符占位；如果比域宽长，就按实际内容输出，以适应域宽。长度决定输出内容的长度，例如 `%d` 代表以整型数据格式输出。输出类型如表 2.4 所示。

表 2.4 输出类型

输出类型	格式字符意义
d	以十进制形式输出带符号整数（正数不输出符号）
o	以八进制形式输出无符号整数（不输出前缀 o）
x	以十六进制形式输出无符号整数（不输出前缀 ox）
u	以十进制形式输出无符号整数
c	输出单个字符
s	输出字符串
f	以小数形式输出单、双精度实数
e	以指数形式输出单、双精度实数
g	以 <code>%f</code> 或 <code>%e</code> 中较短的输出宽度输出单、双精度实数

(1) `d` 格式符，以十进制形式输出整数。有以下几种用法：

- ☑ `%d`，按整型数据的实际长度输出。
- ☑ `.*md`，`m` 为指定的输出字段的宽度。如果数据的位数小于 `m`，用\*所指定的字符占位，如果\*未指定用空格占位，若大于 `m`，则按实际位数输出。
- ☑ `%ld`，输出长整型数据。

【例 2.9】 输出占位符。（实例位置：光盘\TM\sl\2\9）

```
#include<iostream>
void main()
{
    printf("%4d\n",1);           //用空格做占位符
    printf("%04d\n",1);         //用 0 做占位符
    int a=10,b=20;
    printf("%d%d\n",a,b);       //相当于字符连接
}
```

程序运行结果如图 2.13 所示。

(2) o 格式符，以八进制形式输出整数。有以下几种用法：

- ☑ %o，按整型数据的实际长度输出。
- ☑ %\*mo，m 为指定的输出字段的宽度。如果数据的位数小于 m，用\*所指定的字符占位，如果\*未指定用空格占位，若大于 m，则按实际位数输出。
- ☑ %lo，输出长整型数据。

(3) x 格式符，以十六进制形式输出整数。有以下几种用法：

- ☑ %x，按整型数据的实际长度输出。
- ☑ %\*mx，m 为指定的输出字段的宽度。如果数据的位数小于 m，用\*所指定的字符占位，如果\*未指定用空格占位，若大于 m，则按实际位数输出。
- ☑ %lx，输出长整型数据。

(4) s 格式符，用来输出一个字符串。有以下几种用法：

- ☑ %s，将字符串按实际长度输出。
- ☑ %\*ms，输出的字符串占 m 列，如字符串本身长度大于 m，则突破 m 的限制，用\*所指定的字符占位，如果\*未指定用空格占位，若字符串长度小于 m，则左补空格。
- ☑ %-ms，如果字符串长度小于 m，则在 m 列范围内，字符串向左靠，右补空格。
- ☑ %m.ns，输出占 m 列，但只取字符串中左端 n 个字符。这 n 个字符输出在 m 列的右侧，左补空格。
- ☑ %-m.ns，输出长整型数据。输出占 m 列，但只取字符串中左端 n 个字符。这 n 个字符输出在 m 列的左侧，右补空格。

【例 2.10】 输出格式控制字符串。(实例位置：光盘\TM\sl\2\10)

```
#include<iostream>
void main()
{
    char *str="helloworld";
    printf("%s\n%10.5s\n%-10.2s\n%.3s",str,str,str,str);
}
```

程序运行结果如图 2.14 所示。

(5) f 格式符，以小数形式输出实型数据。有以下几种用法：

- ☑ %f，不指定字段宽度，整数部分全部输出，小数部分输出 6 位。
- ☑ %m.nf，输出的数据占 m 列，其中有 n 位小数。如果数值长度小于 m，则左端补空格。
- ☑ %-m.nf，输出的数据占 m 列，其中有 n 位小数。如果数值长度小于 m，则右端补空格。

【例 2.11】 输出格式控制。(实例位置：光盘\TM\sl\2\11)

```
#include<iostream>
void main()
{
```

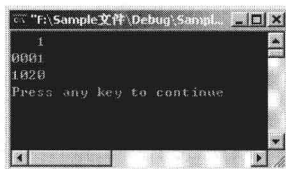


图 2.13 输出占位符

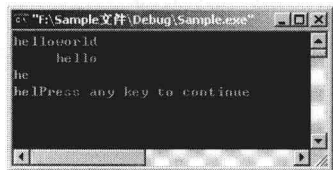


图 2.14 输出浮点



```
float i=2998.453257845;
double j=2998.453257845;
printf("%f\n%15.2f\n%-10.3f\n%f",i,i,i,j);    /*以指定的格式输出 i 和 j*/
}
```

程序运行结果如图 2.15 所示。

(6) e 格式符，以指数形式输出实型数据。有以下几种用法：

- ☒ %e，不指定输出数据所占的宽度和小数位数。
- ☒ %m.ne，输出的数据占 m 位，其中有 n 位小数。如果数值长度小于 m，则左端补空格。
- ☒ %-m.ne，输出的数据占 m 位，其中有 n 位小数。如果数值长度小于 m，则右端补空格。

【例 2.12】 科学记数法输出。（实例位置：光盘\TM\sl2\12）

```
#include<iostream>
void main()
{
    float i=2998.453257845;
    double j=2998.453257845;
    printf("%e\n%15.2e\n%-10.3e\n%e",i,i,i,j);
}
```

程序运行结果如图 2.16 所示。

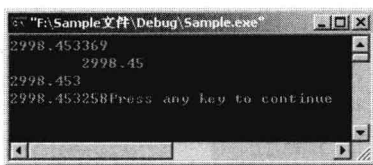


图 2.15 输出格式控制

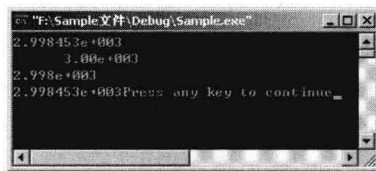


图 2.16 科学记数法输出

## 2.6 小 结

本章主要讲述了 C++ 语言的第一个程序以及常用的数据类型，掌握数据类型对以后编程十分重要。本章还介绍了两种流的输出，可以使用流的输出来调试程序，查看输出结果。

## 2.7 实践与练习

1. 使用字符变量，在控制台上输出一句“NiceWork!”。（答案位置：光盘\TM\sl2\13）
2. 在自定义的函数中使用 static 静态局部整型变量，计算 3 的立方等于多少。（答案位置：光盘\TM\sl2\14）
3. 定义一个整型变量，并为其赋值 123，使用两种方法进行输出，一种是用 printf 输出语句进行输出，另一种是使用 C++ 语言中的流进行输出。（答案位置：光盘\TM\sl2\15）

# 第 3 章

---

## 表达式与语句


(  视频讲解：55 分钟 )

程序是由不同语句组成的，掌握程序语句的写法，也就等于掌握了编程语言的语法。程序主要是用来完成计算的，避免不了要使用运算符。C++提供了丰富的运算符，方便开发人员使用，这也是 C++语言灵活的体现。本章将讲述程序开发的关键部分——表达式与语句。

通过阅读本章，您可以：

- » 掌握常用的运算符号
- » 掌握运算符之间的优先级
- » 了解由不同运算符组成的表达式
- » 了解什么是语句

## 3.1 运算符

 视频讲解：光盘\TM\lx\3\运算符.exe

运算符就是具有运算功能的符号。C++语言中有丰富的运算符，其中有很多运算符都是从 C 语言继承下来的，它新增的运算符有作用域运算符“::”和成员指针运算符“->”。

和 C 语言一样，根据使用运算符的对象个数，将运算符分为单目运算符、双目运算符和三目运算符。根据使用运算符的对象之间的关系，将运算符分为算术运算符、关系运算符、逻辑运算符、赋值运算符和逗号运算符。

### 3.1.1 算术运算符

算术运算主要指常用的加（+）、减（-）、乘（\*）、除（/）四则运算。算术运算符中有单目运算符和双目运算符，如表 3.1 所示。

表 3.1 算术运算符

操 作 符	功 能	目 数	用 法
+	加法运算符	双目	expr1 + expr2
-	减法运算符	双目	expr1 - expr2
*	乘法运算符	双目	expr1 * expr2
/	除法运算符	双目	expr1 / expr2
%	模运算	双目	expr1 % expr2
++	自增加	单目	++expr 或 expr++
--	自减少	单目	--expr 或 expr--

#### 说明

expr 表示使用运算符的对象，可以是表达式、变量和常量。

（1）+是加法运算符，可以进行两个对象的加法运算，例如，1+1 表示两个常量相加；i+1 表示变量和常量相加；x+y 表示两个变量相加；+100 表示有符号的常量，强调常量是正数。

（2）-是减法运算符，可以进行两个对象的减法运算，例如，1-1 表示两个常量相减；j-1 表示变量和常量相减；x-y 表示两个变量相减；-100 表示有符号的常量，强调常量是一个负值。

（3）\*是乘法运算符，可以进行两个对象的乘法运算，例如，2\*3 表示两个常量相乘。

（4）/是除法运算符，可以进行两个对象的除法运算，例如，2/3 表示两个常量相除，/运算符左侧的是被除数，也称分子；/运算符右侧的是除数，也称为分母。

在进行除法运算时，除数或分母不可以为 0，为 0 会产生溢出，处理器抛出异常。例如，2/0 表示

不合法运算，0/2 表示合法运算，计算结果是 0。

两个整型数值进行除法运算时返回的结果可能是一个小数，小数点后的数值会被舍去。

(5) %是模运算符，求两个整型的数值或变量在进行除法运算后的余数。例如，5%2 表示两个常量进行求模运算，计算结果是 1。

(6) ++是自加运算符，属于单目运算符。有++expr 和 expr++两种形式，++expr 表示 expr 自加 1 后再进行其他运算；expr++表示 expr 先参加完其他运算后再进行自加 1，expr 只能是变量。例如，i++ 表示 i 自增 1 后再参与其他运算；++i 表示 i 参与运算后，i 的值再自增；1++表示不合法。

(7) --是自减运算符，属于单目运算符。有--expr 和 expr--两种形式，--expr 表示 expr 自减 1 后再进行其他运算；expr--表示 expr 先参加完其他运算后再进行自减 1，expr 只能是变量。例如，i-- 表示 i 自减 1 后再参与其他运算；--i 表示 i 参与运算后，i 的值再自减；1--表示不合法。

### 3.1.2 关系运算符

关系运算主要是对两个对象进行比较，运算结果是逻辑常量真或假。关系运算符如表 3.2 所示。

表 3.2 关系运算符

操 作 符	功 能	目 数	用 法
<	小于	双目	expr1 < expr2
>	大于	双目	expr1 > expr2
>=	大于或等于	双目	expr1 >= expr2
<=	小于或等于	双目	expr1 <= expr2
==	恒等	双目	expr1 == expr2
!=	不等	双目	expr1 != expr2

(1) <是比较两个对象的大小，前者小于后者，运算结果为真。例如，a<b 表示两个变量进行比较，如果变量 a 的值小于变量 b 的值，运算结果为真；2<1 的运算结果为假。

(2) >是比较两个对象的大小，前者大于后者，运算结果为真。例如，a>b 表示对两个变量进行比较，如果变量 a 的值大于变量 b 的值，运算结果为真；2>1 的运算结果为真。

(3) >=是比较两个对象的大小，前者大于或等于后者，运算结果为真。例如，3>=2 的运算结果为真；2>=2 的运算结果为真。

(4) <=是比较两个对象的大小，前者小于或等于后者，运算结果为真。例如，1<=2 的运算结果为真。

(5) ==是对两个对象进行判断，前者恒等于后者，运算结果为真。例如，a==b 表示对两个变量进行比较，如果变量 a 的值恒等于变量 b 的值，运算结果为真。

(6) !=是对两个对象进行判断，前者不等于后者，运算结果为真。例如，a!=b 表示对两个变量进行比较，如果变量 a 的值不等于变量 b 的值，运算结果为真。

关系运算符都是双目运算符，其结合性均为左结合。关系运算符的优先级低于算术运算符，高于赋值运算符。在 6 个关系运算符中，<、<=、>、>=的优先级相同，高于==和!=，==和!=的优先级相同。

### 3.1.3 逻辑运算符

逻辑运算符是对真和假这两种逻辑值进行运算，运算后的结果仍是一个逻辑值。逻辑运算符如表 3.3 所示。

表 3.3 逻辑运算符

操 作 符	功 能	目 数	用 法
&&	逻辑与	双目	expr1 && expr2
	逻辑或	双目	expr1    expr2
!	逻辑非	单目	!expr

(1) &&是对两个对象进行与运算，当两个对象都为真时，结果为真；有一个对象为假或两个对象都为假时，结果为假。例如，真&&假的结果为假；真&&真的结果为真；假&&假的结果为假。

(2) ||是对两个对象进行或运算，当两个对象都为假时，结果为假，有一个对象为真或两个对象都为真时，结果为真。例如，真||假的结果为真；真||真的结果为真；假||假的结果为假。

(3) !是对一个对象取反运算，当对象为真时，运算结果为假；当对象为假时，运算结果为真。例如，!真的运算结果为假，!假的运算结果为真。

变量 a 和 b 的逻辑运算如表 3.4 所示。

表 3.4 逻辑运算结果

a	b	a&& b	a  b	!a	!b
0	0	0	0	1	1
0	非 0	0	1	1	0
非 0	0	0	1	0	1
非 0	非 0	1	1	0	0

#### 说明

用 1 代表真，用 0 代表假。

其中的表达式仍可以是逻辑表达式，从而组成了嵌套的情形。例如，(a||b)&& c 是根据逻辑运算符的左结合性。

**【例 3.1】** 求逻辑表达式的值。（实例位置：光盘\TM\sl\3\1）

```
#include<iostream>
using namespace std;
void main()
{
    int i=5,j=8,k=12,l=4,x1,x2;
    x1=i>j&& k>l;
    x2=! (i>j)&& k>l;
```

```
printf("%d,%d\n",x1,x2);
}
```

程序运行结果如图 3.1 所示。



图 3.1 运行结果

### 3.1.4 赋值运算符

赋值运算符分为简单赋值运算符和复合赋值运算符，复合赋值运算符又称为带有运算的赋值运算符，简单赋值运算符就是给变量赋值的运算符。例如：

**变量 = 表达式**

等号“=”就为简单赋值运算符。

C++提供了很多复合赋值运算符，如表 3.5 所示。

表 3.5 赋值运算符

操 作 符	功 能	目 数	用 法
+=	加法赋值	双目	expr1 += expr2
-=	减法赋值	双目	expr1 -= expr2
*=	乘法赋值	双目	expr1 *= expr2
/=	除法赋值	双目	expr1 /= expr2
%=	模运算赋值	双目	expr1 %= expr2
<<=	左移赋值	双目	expr1 <<= expr2
>>=	右移赋值	双目	expr1 >>= expr2
&=	按位与运算并赋值	双目	expr1 &= expr2
=	按位或运算并赋值	双目	expr1  = expr2
^=	按位异或运算并赋值	双目	expr1 ^= expr2

复合赋值运算符都有等同的简单赋值运算符和其他运算的组合。例如：

a+=b 等价于 a=a+b

a/=b 等价于 a=a/b

a>>=b 等价于 a=a>>b

a|=b 等价于 a=a|b

a-=b 等价于 a=a-b

a%=b 等价于 a=a%b

a&=b 等价于 a=a&b

a\*=b 等价于 a=a\*b

a<<=b 等价于 a=a<<b

a^=b 等价于 a=a^b

复合赋值运算符都是双目运算符，C++采用这种运算符可以更高效地进行运算，编译器在生成目标代码时能够直接优化，可以使程序代码更小。这种书写形式也非常简洁，使得代码更紧凑。

复合赋值运算符将运算结果返回，作为表达式的值，同时把操作数 1 对应的变量设为运算结果值。例如：

```
int a=6;
a*=5;
```

运算结果是：a 的值为 30。  
a\*=5 等价于 a=a\*5，a\*5 的运算结果作为临时变量赋给了变量 a。

3.1.5 位运算

位运算符有位逻辑与、位逻辑或、位逻辑异或和取反运算符，其中位逻辑与、位逻辑或、位逻辑异或为双目运算符，取反运算符为单目运算符。位运算如表 3.6 所示。

表 3.6 位运算操作符

操 作 符	功 能	目 数	用 法
&	位逻辑与	双目	expr1 & expr2
	位逻辑或	双目	expr1   expr2
^	位逻辑异或	双目	expr1 ^ expr2
~	取反运算符	单目	~expr

在双目运算符中，位逻辑与优先级最高，位逻辑或次之，位逻辑异或最低。

(1) 位逻辑与实际上是将操作数转换成二进制表示方式，然后将两个二进制操作数对象从低位（最右边）到高位对齐，每位求与，若两个操作数对象同一位都为 1，则结果对应位为 1，否则结果中对应位为 0。例如 12 和 8 经过位逻辑与运算后得到的结果是 8。

0000 0000 0000 1100

(十进制 12 原码表示)

& 0000 0000 0000 1000

(十进制 8 原码表示)

0000 0000 0000 1000

(十进制 8 原码表示)



说明

十进制在用二进制表示时有原码、反码、补码多种表示方式。

(2) 位逻辑或实际上是将操作数转换成二进制表示方式，然后将两个二进制操作数对象从低位（最右边）到高位对齐，每位求或，若两个操作数对象同一位都为 0，则结果对应位为 0，否则结果中对应位为 1。例如，31 和 22 经过位逻辑或运算后得到的结果是 31。

0000 0000 0000 0100

(十进制 4 原码表示)

| 0000 0000 0000 1000

(十进制 8 原码表示)

0000 0000 0000 1100

(十进制 12 原码表示)



(3) 位逻辑异或实际上是将操作数转换成二进制表示方式,然后将两个二进制操作数对象从低位(最右边)到高位对齐,每位求异或,若两个操作数对象同一位不同时为1,则结果对应位为1,否则结果中对应位为0。例如,4和8经过位逻辑异或运算后的结果是12。

```

0000 0000 0001 1111      (十进制 31 原码表示)
^ 0000 0000 0001 0110    (十进制 22 原码表示)
-----
0000 0000 0000 1001      (十进制 9 原码表示)

```

(4) 取反运算符,实际上是将操作数转换成二进制表示方式,然后将各位二进制位由1变为0,由0变为1。例如,41883取反运算后得到的结果是23652。

```

~ 1010 0011 1001 1011    (十进制 41883 原码表示)
   0101 1100 0110 0100    (十进制 23652 原码表示)

```

逻辑位运算符实际上是算术运算符,用该运算符组成的表达式的值是算术值。

### 3.1.6 移位运算符

移位运算有两个,分别是左移<<和右移>>,这两个运算符都是双目的。

(1) 左移是将一个二进制操作数对象按指定的移动位数向左移,左边(高位端)溢出的位被丢弃,右边(低位端)的空位用0补充。右移相当于乘以2的幂,如图3.2所示。



图 3.2 左移位运算

例如操作数41883的二进制是1010 0011 1001 1011,左移一位变成18230,左移两位变成36460,运行过程如图3.3所示。



图 3.3 左移位运算过程

(2) 右移是将一个二进制操作数对象按指定的位数向右移动,右边(低位端)溢出的位被丢弃,



左边（高位端）的空位或者一律用 0 填充，或者用被移位操作数的符号位填充，运算结果和编译器有关，在使用补码的机器中，正数的符号位为 0，负数的符号为 1。右移位运算相当于除以 2 的幂，如图 3.4 所示。



图 3.4 右移位运算

例如操作数 41883 的二进制是 1010 0011 1001 1011，右移一位变成 20941，右移两位变成 10470，运行过程如图 3.5 所示。

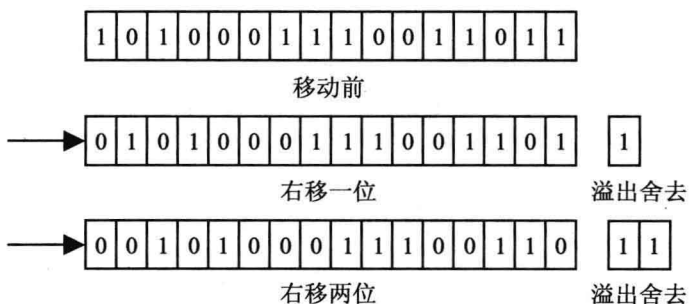


图 3.5 右移位运算过程

### 【例 3.2】 左移运算。（实例位置：光盘\TM\sl\3\2）

```
#include<iostream>
using namespace std;
void main()
{
    int a=0x40,b;
    b=a<<1;
    cout << b << endl;
}
```

运算结果是：

128

由于位运算的速度很快，在程序中遇到表达式乘以或除以 2 的幂的情况，一般采用位运算来代替。

### 【例 3.3】 使用移位运算。（实例位置：光盘\TM\sl\3\3）

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

```

void main()
{
    long nWord=0x12345678;
    int nBits;
    nBits=nWord & 0xFFFF;
    printf("low bits are 0x%x\n",nBits);
    nBits=(nWord & 0xFFFF0000)>>16;
    printf("hight bits are 0x%x\n",nBits);
}

```

运算结果如图 3.6 所示。

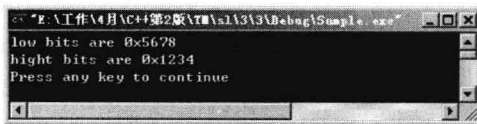


图 3.6 运算结果

### 3.1.7 sizeof 运算符

sizeof 是一个很像函数的运算符，也是唯一一个用到字母的运算符。该运算符有以下两种形式：

```

sizeof(类型说明符)
sizeof(表达式)

```

功能是返回指定的数据类型或表达式值的数据类型在内存中占用的字节数。

#### 说明

由于 CPU 寄存器的位数不同，同种数据类型占用的内存字节数目也可能不同。

例如：

```
sizeof(char)
```

返回 1，说明 char 类型占用 1 个字节。

```
sizeof(void *)
```

返回 4，说明空指针占用 4 个字节。

```
sizeof(66)
```

返回 4，说明常量占用 4 个字节。

### 3.1.8 条件运算符

条件运算符是 C++ 语言中仅有的一个三目运算符，该运算符需要 3 个运算数对象，形式如下：

```
<表达式 1> ? <表达式 2> : <表达式 3>
```

表示式 1 是一个逻辑值，可以为真或假。若表达式 1 为真，则运算结果是表达式 2，若表达式 1 为假，则运算结果是表达式 3。这个运算相当于一个 if 语句。

3.1.9 逗号运算符

C++语言中逗号“,”也是一种运算符,称为逗号运算符。逗号运算符的优先级别最低,结合方向自左至右,其功能是把两个表达式连接起来组成一个表达式。逗号运算符是一个多目运算符,并且操作数的个数不限定,可以将任意多个表达式组成一个表达式。

例如:

```
x,y,z
a=1,b=2
```

3.2 结合性和优先级

 视频讲解: 光盘\TM\lx\3\结合性和优先级.exe

运算符优先级决定了在表达式中各个运算符执行的先后顺序,高优先级运算符要先于低优先级运算符进行运算。例如根据先乘除后加减的原则,表达式  $a+b*c$  会先计算  $b*c$ ,得到结果后再与  $a$  相加。在优先级相同的情况下,则按从左到右的顺序进行计算。

当表达式中出现了括号时,会改变优先级。先计算括号中的子表达式值,再计算整个表达式的值。

运算符的结合方式有两种,左结合和右结合。左结合表示运算符优先与其左边的标识符结合进行运算,例如加法运算;右结合表示运算符优先与其右边的标识符结合,例如单目运算符+、-。

同一优先级的优先级别相同,运算次序由结合方向决定。例如  $1*2/3$ , $*$ 和/ $的优先级别相同,其结合方向自左向右,等价于(1*2)/3。$

运算符的优先级如表 3.7 所示。


表 3.7 运算符优先级

运 算 符	名 称	优 先 级	结 合 性
() [] > .	圆括号 下标 取类或结构分量 取类或结构成员	1 (最高)	→
! ~ ++ -- - & * (类型) sizeof	逻辑非 按位取反 自增 1 自减 1 取负 取地址 取内容 强制类型转换 长度计算	2	←

续表

运 算 符	名 称	优 先 级	结 合 性
* / %	乘 除 整数取模	3	→
+ -	加 减	4	→
<< >>	左移 右移	5	→
< <= > >=	小于 小于等于 大于 大于等于	6	→
== !=	恒等 不等于	7	→
&	按位与	8	→
~	按位异或	9	→
	按位或	10	→
&&	逻辑与	11	→
	逻辑或	12	→
?:	条件	13	→
= /= %= *= -= >>= <<= &= ^  =	赋值 /运算并赋值 %运算并赋值 *运算并赋值 -运算并赋值 >>运算并赋值 <<运算并赋值 &运算并赋值 ^运算并赋值  运算并赋值	14	→ ← ←
,	逗号（顺序求值）	15（最低）	→

### 3.3 表 达 式

 视频讲解：光盘\TM\lx\3\表达式.exe

表达式由运算符、括号、数值对象或变量等几个元素构成。一个数值对象是最简单的表达式，一个表达式可以看作一个数学函数，带有运算符的表达式通过计算将返回一个数值。例如：

```
1 + 1
3.1415926
```

```
i + 1
x > y
100 >> 2
j * 3
```

当表达式有两个或多个运算符时，表达式称为复杂表达式，运算符执行的先后顺序由它们的优先级和结合性决定。例如：

```
(X+Y)*Z
a*x+b*y+z
```

一个表达式的值的数据类型由运算符的种类和操作数的数据类型决定。

带运算符的表达式根据运算符的不同，可以分成算术表达式、关系表达式、逻辑表达式、条件表达式和赋值表达式等几类。

### 3.3.1 算术表达式

算术表达式的一般形式如下：

```
表达式 算术运算符 表达式
```

算术表达式由算术运算符把表达式连接而成，其值的计算很简单，其值的数据类型按下述规定确定：若所有运算符数量类型相同，则表达式运算结果的数据类型和操作数的数据类型相同；若操作数的数据类型不同，就需要转换，表达式运算结果的数据类型取精度最高的数据类型。

### 3.3.2 关系表达式

关系表达式的一般形式如下：

```
表达式 关系运算符 表达式
```

关系表达式一般只出现在三目运算符、if 语句和循环语句的判断条件中。关系表达式的运算结果都是逻辑型，只能取 true 或 false。数值 0 表示 false，非 0 代表 true。

### 3.3.3 条件表达式

条件表达式的一般形式如下：

```
关系表达式 ? 表达式 : 表达式
```

条件表达式的值和数据类型取决于“？”前表达式的真假，若为真，则整个表达式的运算结果和数据类型和“:”前的操作数相同；若为假，则整个表达式的值和数据类型和“:”后的操作数相同。

### 3.3.4 赋值表达式

赋值表达式的一般形式如下：

**表达式 赋值运算符 表达式**

赋值运算符的值和数据类型的第一个操作数对象赋值完毕后的值和数据类型相同。

由于赋值运算符的结合性是从右至左，因此可以出现连续赋值的表达式。

### 3.3.5 逻辑表达式

逻辑表达式的一般形式为：

**表达式 逻辑运算符 表达式**

逻辑表达式用逻辑运算符将关系表达式连接起来。逻辑表达式的值也是逻辑型，只能取真值 `true` 或假值 `false`。

其中的表达式也可以是逻辑表达式，从而组成了嵌套的情形。例如，`(a||b)&& c` 根据逻辑运算符的左结合性，也可写为 `a||b&& c`。逻辑表达式的值是式中各种逻辑运算的最后值，以 1 和 0 分别代表真和假。

逻辑表达式的注意事项如下：

(1) 逻辑运算符两侧的操作数，除可以是 0 和非 0 的整数外，也可以是其他任何类型的数据，如实型、字符型等。

(2) 在计算逻辑表达式时，只有在必须执行下一个表达式才能求解时，才求解该表达式，也就是说并不是所有的表达式都被求解。

☑ 对于逻辑与运算，如果第一个操作数被判定为假，系统不再判定或求解第二操作数。

☑ 对于逻辑或运算，如果第一个操作数被判定为真，系统不再判定或求解第二操作数。

### 3.3.6 逗号表达式

C++语言中逗号“,”也是一种运算符，称为逗号运算符。逗号运算符的优先级别最低，结合方向自左至右，其功能是把两个表达式连接起来组成一个表达式，称为逗号表达式。

其一般形式如下：

**表达式 1, 表达式 2**

其求值过程是先求解表达式 1，再求解表达式 2，并以表达式 2 的值作为整个逗号表达式的值。

逗号表达式的一般形式可以扩展为：

**表达式 1, 表达式 2, 表达式 3, ..., 表达式 n**

该逗号表达式的值为表达式 n 的值。

整个逗号表达式的值和类型由最后一个表达式决定。计算一个逗号表达式的值时，从左至右依次计算各个表达式的值，最后计算的一个表达式的值和类型便是整个逗号表达式的值和类型。

逗号表达式的用途仅在于解决只能出现一个表达式的地方却要出现多个表达式的问题。

【例 3.4】 逗号运算符应用。（实例位置：光盘\TM\sl\3\4）

```
#include<iostream>
using namespace std;
void main()
{
    int a=4,b=6,c=8,res1,res2;
    res1=a,res2=b+c;
    for(int i=0,j=0;i<2;i++)
    {
        printf("y=%d,x=%d\n",res1,res2);
    }
}
```

程序运行结果如图 3.7 所示。

实例中多处用到了逗号表达式，变量赋初值时、for 循环语句中、printf 打印语句中。其中“res1=a,res2=b+c;”比较难理解，其中 res2 等于整个逗号表达式的值，也就是表达式 2 的值，res1 是第一个表达式的值。

逗号表达式的注意事项如下：

（1）逗号表达式可以嵌套。例如：

表达式 1,(表达式 2,表达式 3)

嵌套的逗号表达式可以转换成扩展形式，扩展形式如下：

表达式 1,表达式 2,...,表达式 n

整个逗号表达式的值等于表达式 n 的值。

（2）程序中使用逗号表达式，通常是要分别求逗号表达式内各表达式的值，并不一定要求整个逗号表达式的值。

（3）并不是在所有出现逗号的地方都组成逗号表达式，如在变量说明中，函数参数表中逗号只是用作各变量之间的间隔符。

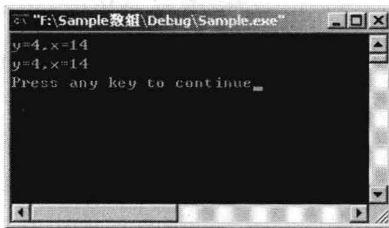



图 3.7 运行结果

### 3.3.7 表达式中的类型转换

 视频讲解：光盘\TM\lx\3\表达式中的类型转换.exe

变量的数据类型转换的方法有两种：一种是隐式转换，一种是强制转换。

#### 1. 隐式转换

隐式转换发生在不同数据类型的量混合运算时，由编译系统自动完成。



隐式转换遵循以下规则：

(1) 若参与运算量的类型不同，则先转换成同一类型，然后进行运算。赋值时会把赋值类型和被赋值类型转换成同一类型，一般赋值号右边量的类型将转换为左边量的类型。如果右边量的数据类型长度比左边长时，将丢失一部分数据，这样会降低精度，丢失的部分按四舍五入向前舍入。

(2) 转换按数据由低到高顺序执行，以保证精度不降低。

- ☑ int 型和 long 型运算时，先把 int 型转成 long 型后再进行运算。
- ☑ 所有的浮点运算都是以双精度进行的，即使仅含 float 单精度量运算的表达式，也要先转换成 double 型，再作运算。
- ☑ char 型和 short 型参与运算时，必须先转换成 int 型。

类型转换的顺序如图 3.8 所示。

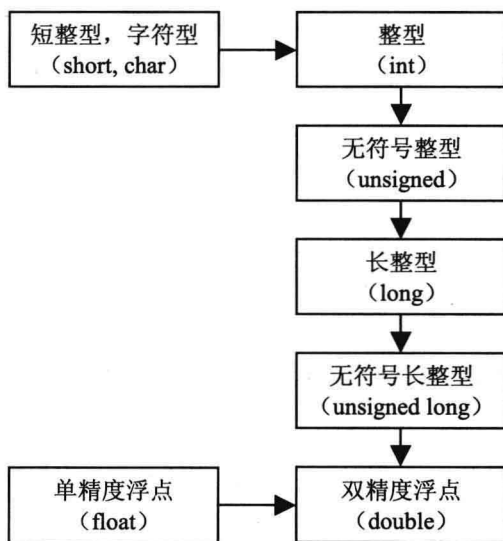


图 3.8 数据类型转换

【例 3.5】 隐式类型转换。（实例位置：光盘\TM\sl\3\5）

```

#include<iostream>
using namespace std;
void main()
{
    double result;
    char a='k';
    int b=10;
    float e=1.515;
    result=(a+b)-e;
    printf("%f\n",result);
}
  
```

程序运行结果为：



```
115.485000
```

## 2. 强制类型转换

强制类型转换是通过类型转换运算来实现的，其一般形式为：

```
类型说明符 (表达式)
```

或

```
(类型说明符) 表达式
```

其功能是把表达式的运算结果强制转换成类型说明符所表示的类型。

例如：

```
(float) x;
```

表示把 x 转换为单精度型。

```
(int)(x+y);
```

表示把 x+y 的结果转换为整型。

```
int(1.3)
```

表示一个整数。

强制类型转换后不改变数据说明时对该变量定义的类型。例如：

```
double x;
(int)x;
```

x 仍为双精度类型。

使用强制转换的优点是编译器不必自动进行两次转换，而由程序员负责保证类型转换的正确性。


**【例 3.6】** 强制类型转换应用。（实例位置：光盘\TM\3\6）

```
#include<iostream>
using namespace std;
void main()
{
    float i,j;
    int k;
    i=60.25;
    j=20.5;
    k=(int)i+(int)j;
    cout << k << endl;
}
```

程序运行结果为：

```
80
```

## 3.4 语 句

 视频讲解：光盘\TM\lx\3\语句.exe

在 C++ 程序中，语句是最小的可执行单元，一条语句由一个分号结束。

C++ 程序语句按其功能可以划分为两类，一类是用于描述计算机执行操作运算的，称为操作运算语句；另一类是用于控制操作运算执行顺序的，称为流程控制语句。任何程序设计语句都具备流程控制的功能。基本的控制结构有 3 种：顺序结构、选择结构和循环结构。

顺序结构指按照语句在程序中的先后次序一条一条顺次执行。顺序结构是自然形成的，不需要控制，按默认的顺序执行，顺序控制语句就是一条简单的语句。

### 1. 表达式语句

表达式语句是由表达式后面加上一个分号组成的。表达式有很多种，如关系表达式、逻辑表达式、算术表达式等，但关系表达式、逻辑表达式多用于循环或选择结构中，只有赋值表达式多用于赋值语句。赋值表达式后面加上一个分号可以形成赋值语句，将右边的表达式（算术表达式）的结果赋给左边的变量。一个赋值语句中可以包含多个赋值表达式。

### 2. 空语句

空语句只有一个分号，表示什么也不做。空语句经常出现在选择或循环语句中，表示某个分支或循环体不执行具体的操作，也用于编制程序的初始阶段，在搭建程序的模块框架中，先用空语句占位，接下来再逐步细化和补充。

例如：

```
while(a < b)
;
```

上面是一个循环语句，表示当变量  $a$  小于变量  $b$  时，循环体中要进行某种操作，但不确定循环体应该实现什么功能，所以需要使用空语句占位。空语句语法上是正确的。

### 3. 复合语句

复合语句是若干条语句的一个集合，它在语法上是一个整体，相当于一个语句，其语法形式是由一对大括号将若干条语句括起来。复合语句经常出现在选择或循环结构中，选择语句的分支和循环语句的循环体由多条语句组成时，用大括号括起来形成一条复合语句，起到层次划分的作用。一对大括号形成了一个范围，这个范围也是变量的作用范围，也可以将大括号内的代码称之为程序段。在能使用简单语句的地方，都能够使用复合语句。在一个复合语句中可以包含另外一个或多个复合语句。

例如：

```
{
    x=1;
```

```
y=2;  
a=x+y;  
}
```

一个复合语句的大括号外面不能再写分号。

#### 4. 函数调用语句

函数由函数名、带实际参数表的圆括号组成，函数调用语句就是在函数后加上一个分号。调用主要指程序执行到函数调用语句时，会跳转到相应的函数体中执行该函数体中的内容，执行完所有内容后返回到函数调用语句处，执行调用语句下面的语句。可以调用的函数主要有系统库函数和自定义函数。

顺序、选择、循环是结构化程序的3种基本结构。选择结构语句、循环结构语句会在后面章节讲到。

### 3.5 小 结

本章介绍了C++语言中的运算符，以及由运算符组成的表达式和语句，不同运算符有不同的运算规则，掌握这些规则是开发程序的关键。运算符的相关规则关系到程序的运算结果，运算符的优先级是开发人员必须掌握的，学习时要多加注意。与运算符相关的表达式及语句，都是程序的基本组成部分，要理解各语句之间的关系。


### 3.6 实践与练习

1. 不借助第3个变量交换两个变量的值。（答案位置：光盘\TM\sl\3\7）
2. 使用复合运算符计算  $a+=a*=a/=a-6$  的结果。（答案位置：光盘\TM\sl\3\8）

# 第 4 章

---

## 条件判断语句

(  视频讲解：35 分钟 )

判断语句是重要的程序控制语句，开发程序时会大量运用判断语句。判断语句有很多形式，灵活运用各种形式的判断语句可以提高软件的效率，并且逻辑性强的判断语句容易阅读，可以起到简化代码的作用。

通过阅读本章，您可以：

- » 掌握 3 种形式的判断语句
- » 了解条件运算符与判断语句的转换
- » 掌握 switch 分支语句
- » 掌握判断语句的嵌套

## 4.1 决策分支

 视频讲解：光盘\TM\lx\4\决策分支.exe

计算机的主要功能是为用户提供计算功能，但在计算的过程中会遇到各种各样的情况，针对不同的情况会有不同的处理方法，这就要求程序开发语言要有处理决策的能力。汇编语言使用判断指令和跳转指令实现决策，高级语言使用选择判断语句实现决策。

一个决策系统就是一个分支结构，这种分支结构就像一个树形结构，每到一个节点都需要做决定，就像人走到十字路口，是向前走、向左走或是向右走都需要做决定。不同的分支代表不同的决定。例如十字路口的分支结构如图 4.1 所示。

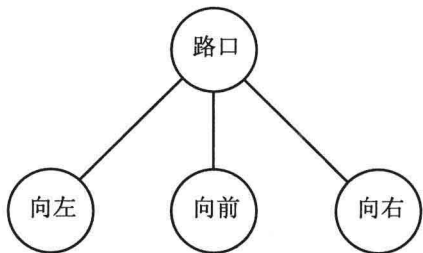


图 4.1 分支结构

为描述决策系统的流通，设计人员开发了流程图。流程图使用图形方式描述系统不同状态的不同处理方法。开发人员使用流程图表现程序的结构。

主要的流程图符号如图 4.2 所示。

使用流程图描述十字路口转向的决策，利用方位做决定，判断是否是南方，如果是南方，向前行，如果不是南方，寻找南方，如图 4.3 所示。

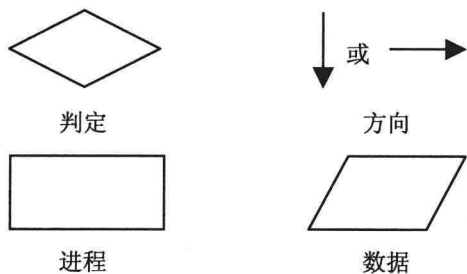


图 4.2 主要的流程图符号

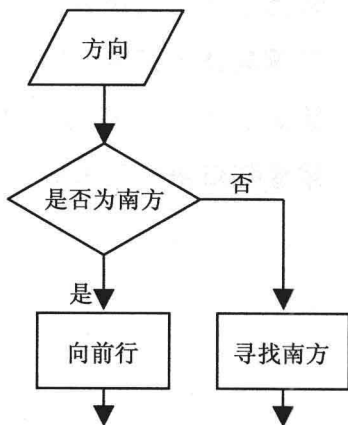


图 4.3 流程图

程序中使用选择判断语句来做决策，选择判断是编程语言的基础语句，在 C++ 语言中有 3 种形式的选择判断语句，同时提供了 switch 语句简化多分支决策的处理。下面对选择判断语句进行介绍。



**说明**

选择判断语句可以简称为判断语句，有的书中也称其为分支语句。

## 4.2 判断语句

 视频讲解：光盘\TM\lx\4\判断语句.exe

### 4.2.1 第一种形式的判断语句

C++语言中使用 if 关键字来组成判断语句，第一种判断语句的形式如下：

```
if(表达式)  
语句
```

表达式一般为关系表达式，表达式的运算结果应该是真或假（true 或 false）。如果表达式为真，执行语句，如果为假就跳过，执行下一条语句。用流程图表示第一种判断语句如图 4.4 所示。

**【例 4.1】** 判断输入数是否为奇数。（实例位置：光盘\TM\sl\4\1）

```
#include<iostream>  
using namespace std;  
void main()  
{  
    int iInput;  
    cout << "Input a value:" << endl;  
    cin >> iInput; //输入一整型数  
    if(iInput%2!=0)  
        cout << "The value is odd number" << endl;  
}
```

用流程图来描述判断语句的执行过程，如图 4.5 所示。

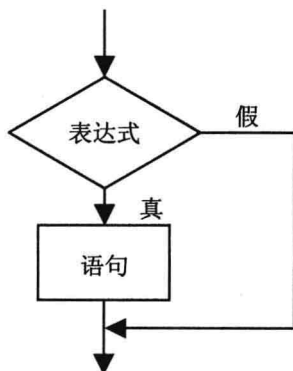


图 4.4 第一种形式的判断语句

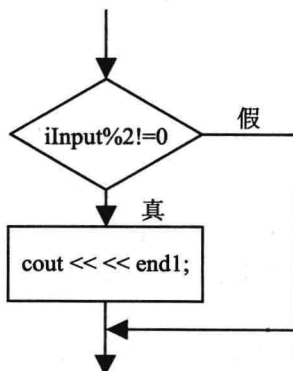


图 4.5 判断语句的执行过程

程序分两步执行。

(1) 定义一个整型变量 `iInput`，然后使用 `cin` 获得用户输入的整型数据。

(2) 对变量 `iInput` 的值与 2 进行 % 运算，如果运算结果不为 0，表示用户输入的是奇数，输出字符串 “The value is odd number”。如果运算结果为 0，则不进行任何输出，程序执行完毕。



#### 说明

整数与 2 进行 % 运算，结果只有 0 或 1 两种情况。

要注意第一种形式的判断语句的书写格式。

判断语句：

```
if(a>b)
    max=a;
```

可以写成：

```
if(a>b) max=a;
```

但不建议使用 “`if(a>b) max=a;`” 这种书写方式，这种方式不便于阅读。

判断形式中的语句可以是复合语句，也就是说可以用大括号括起多条简单语句。例如：

```
if(a>b)
{
    tmp=a;
    b=a;
    a=tmp;
}
```

## 4.2.2 第二种形式的判断语句

第二种形式的判断语句使用了 `else` 关键字，形式如下：

```
if(表达式)
    语句 1;
else
    语句 2;
```

表达式是一个关系表达式，表达式的运算结果应该是真或假（`true` 或 `false`），如果表达式的值为真，执行语句 1，为假则执行语句 2。

第二种形式的判断语句相当于汉语里的 “如果……那么……”，用流程图表示第二种判断语句，如图 4.6 所示。

**【例 4.2】** 根据分数判断是否优秀。（实例位置：光盘\TM\sl\4\2）

```
#include<iostream>
using namespace std;
void main()
{
```



```

int iInput;
cin >> iInput;
if(iInput>90)
    cout << "It is Good" << endl;
else
    cout << "It is not Good" << endl;
}

```

用流程图来描述判断语句的执行过程，如图 4.7 所示。

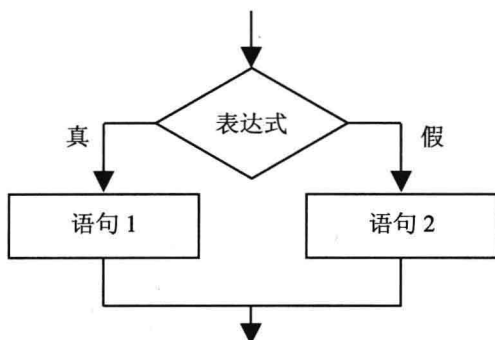


图 4.6 第二种判断语句

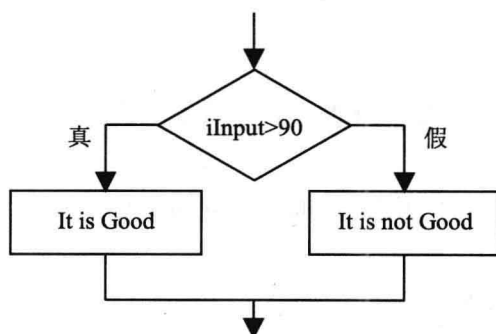


图 4.7 判断语句的执行过程

程序需要和用户交互，用户输入一个数值，将该数值赋值给 `iInput` 变量，然后判断用户输入的数据是否大于 90，如果大于 90，输出字符串 “It is Good”，否则输出字符串 “It is not Good”。

**【例 4.3】** 改进的奇偶性判别。（实例位置：光盘\TM\sl\4\3）

```

#include<iostream>
using namespace std;
void main()
{
    int iInput;
    cout << "Input a value:" << endl;
    cin >> iInput; //输入一整数
    if(iInput%2!=0)
        cout << "The value is odd number" << endl;
    else
        cout << "The value is even number" << endl;
}

```

程序分两步执行。

- (1) 定义一个整型变量 `iInput`，然后使用 `cin` 获得用户输入的整型数据。
- (2) 对变量 `iInput` 的值与 2 进行 % 运算，如果运算结果不为 0，表示用户输入的是奇数，输出字符串 “The value is odd number”；如果运算结果为 0，表示用户输入的是偶数，输出字符串 “The value is even number”，最后程序执行完毕。

else 使用时的注意事项如下：

- (1) else 不能单独使用，必须和关键字 `if` 一起出现。“`else(a>b) max=a`”是不合法的。
- (2) else 后跟的语句可以是复合语句。例如：



```
f(a>b)
{
    max=a;
    cout << a << endl;
}
else
{
    max=b;
    cout << b << endl;
}
```

### 4.2.3 第三种形式的判断语句

第三种形式的判断语句是可以进行多次判断的语句，每判断一次就缩小一定的检查范围，其形式如下：

```
if(表达式 1)
    语句 1;
else if(表达式 2)
    语句 2;
else if(表达式 3)
    语句 3
...
else if(表达式 m)
    语句 m;
else
    语句 n;
```

表达式一般为关系表达式，表达式的运算结果应该是真或假（true 或 false）。如果表达式为真，执行语句，如果表达式为假就跳过，执行下一条语句。用流程图表示第三种判断语句，如图 4.8 所示。

**【例 4.4】** 根据成绩划分等级。（实例位置：光盘\TM\sl\4\4）

```
#include<iostream>
using namespace std;
void main()
{
    int ilnput;
    cin >> ilnput;
    if(ilnput>=90)
    {
        cout << "very good" <<endl;
    }
    else if(ilnput>=80&& ilnput<90)
    {
        cout << "good" <<endl;
    }
    else if(ilnput>=70 && ilnput <80)
```

```

{
    cout << "good" << endl;
}
else if(input >= 60 && input < 70)
{
    cout << "normal" << endl;
}
else if(input < 60)
{
    cout << "failure" << endl;
}
}

```

程序需要用户输入整型数值，然后判断数值是否大于 90，如果大于 90，输出“very good”字符串；否则继续判断，判断是否小于 90 大于 80，如果小于 90 大于 80，输出“good”字符串；否则继续判断，依此类推，最后判断是否小于 60，如果小于 60，输出“failure”字符串，最后没有使用 else 再进行判断。

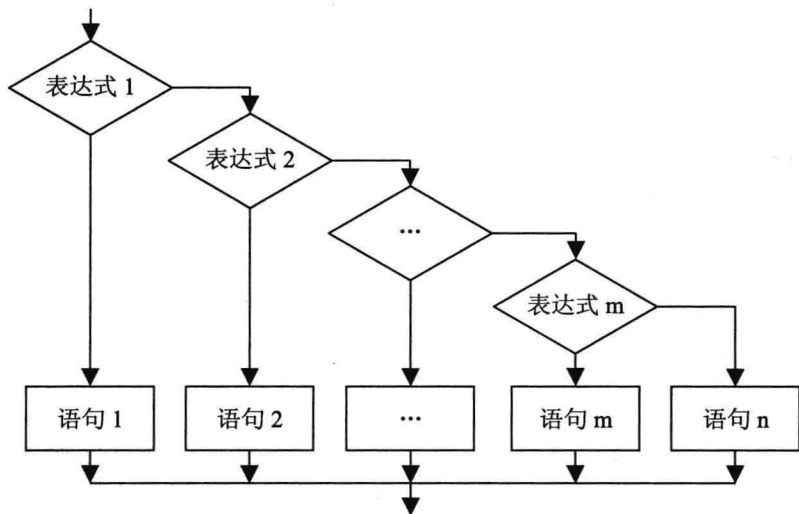


图 4.8 第三种判断语句

### 4.3 使用条件运算符进行判断

 视频讲解：光盘\TM\lx\4\使用条件运算符进行判断.exe

条件运算符是一个三目运算符，它能像判断语句一样完成判断。例如：

```
max=(iA > iB) ? iA : iB;
```

首先比较 iA 和 iB 的大小，如果 iA 大于 iB 就取 iA 的值，否则取 iB 的值。

可以将条件运算符改为判断语句。例如：

```
if(iA > iB)
    max= iA;
else
    max= iB;
```

**【例 4.5】** 用条件运算符完成判断数的奇偶性。(实例位置: 光盘\TM\sl\4\5)

```
#include<iostream>
using namespace std;
void main()
{
    int iInput;
    cout << "Input number" << endl;
    cin >> iInput; //从键盘中输入一个数
    (iInput%2!=0) ? cout << "The value is odd number" : cout << "The value is even number";
    cout << endl;
}
```

该程序使用条件运算符完成判断数的奇偶性, 比使用判断语句时的代码要简洁。程序同样完成由用户输入整型数, 然后和 2 进行%运算, 如果运算结果不为 0, 是奇数, 否则是偶数。

**【例 4.6】** 用条件表达式判断一个数是否是 3 和 5 的整倍数。(实例位置: 光盘\TM\sl\4\6)

```
#include<iostream>
using namespace std;
void main()
{
    int iInput;
    cout << "Input number" << endl;
    cin >> iInput; //从键盘中输入一个数
    (iInput%3==0 && iInput%5==0)?cout << "yes" : cout<<"no";
    cout << endl;
}
```

程序需要用户输入一个整型数, 然后用%运算判断能否被 3 整除, 以及能否被 5 整除, 如果同时能被 3 和 5 整除, 说明输入的整型数是 3 和 5 的整倍数。

条件运算符可以嵌套, 例如:

表达式 1?(表达式 a?表达式 b:表达式 c);:表达式 1;

**【例 4.7】** 用条件表达式判断一个数是否是 3 和 5 的整倍数。(实例位置: 光盘\TM\sl\4\7)

```
#include<iostream>
using namespace std;
void main()
{
    int iInput;
    cout << "Input number" << endl;
    cin >> iInput; //从键盘中输入一个数
    (iInput%3==0)?
        ((iInput%5==0) ? cout << "yes" : cout << "no" )
```

```

        : cout << "no";
        cout << endl;
    }

```

例 4.7 和例 4.6 完成同一个目标，都是通过%运算来判断输入的整型数是否是 3 和 5 的整倍数，但例 4.7 中使用了条件运算符的嵌套。由于条件运算符的嵌套后的代码不容易阅读，一般不建议使用。

## 4.4 switch 语句

 视频讲解：光盘\TM\lx\4\switch 语句.exe

C++语言提供了一种用于多分支选择的 switch 语句。可以使用 if 判断语句做多分支结构程序，但当分支足够多时，if 判断语句会造成代码容易混乱，可读性也很差，如果使用不当就会产生表达式上的错误，所以建议在仅有两个分支或分支数少时使用 if 判断语句，而在分支比较多时使用 switch 语句。

switch 语句的一般形式如下：

```

switch(表达式)
{
case 常量表达式 1:
    语句 1;
    break;
case 常量表达式 2:
    语句 2;
    break;
...
case 常量表达式 n:
    语句 n;
    Break;
default:
    语句 n+1;
}

```

表达式是一个算术表达式，需要计算出表达式的值，该值应该是一个整型数或是一个字符，如果是浮点数，可能会因为精度的不精确而产生错误。

switch 是分支的入口，开始判断是在 case 分语句中，用表达式的值逐一和 case 语句中的值进行比较，有匹配成功的就使用“break;”跳出 switch 语句，如果没有匹配成功的，就执行 default 分句。

default 分句是可以不写的，如果不写 default 分句，case 分语句中没有匹配成功的就不进行任何操作。

**【例 4.8】** 根据输入的字符输出字符串。（实例位置：光盘\TM\sl\4\8）

```

#include<iostream>
#include<iomanip>
using namespace std;
void main()
{
    char ilnput;

```



```
cin >> ilnput;
switch(ilnput)
{
case 'A':
cout << "very good" << endl;
break;
case 'B':
cout << "good" << endl;
break;
case 'C':
cout << "normal" << endl;
break;
case 'D':
cout << "failure" << endl;
break;
default:
cout << "input error" << endl;
}
}
```

程序需要用户输入一个字符，当用户输入字符‘A’时，向屏幕输出“very good”字符串；输入字符‘B’时，向屏幕输出“good”字符串；输入字符‘C’时，向屏幕输出“normal”字符串；输入字符‘D’时，向屏幕输出“failure”字符串；输入其他字符时，向屏幕输出“input error”字符串。

可以将 switch 的判断结构，改为第一种形式的判断语句。

**【例 4.9】** 根据输入的字符输出字符串。（实例位置：光盘\TM\sl\4\9）

```
#include<iostream>
using namespace std;
void main()
{
int ilnput;
cin >> ilnput;
if(ilnput == 'A')
{
cout << "very good" << endl;
return ;
}
if(ilnput == 'B')
{
cout << "good" << endl;
return ;
}
if(ilnput == 'C')
{
cout << "normal" << endl;
return ;
}
if(ilnput == 'D')
{
```

```

        cout << "failure" << endl;
        return ;
    }
    cout << "input error" << endl;
}

```

例 4.9 和例 4.8 完成的功能基本相同。当用户输入字符 A 后，输出字符串 “very good”，所不同的是，输出完字符串后，使用 `return` 跳出主函数，并结束程序，不执行下面的语句。同样输入字符 B、C 和 D 后也输出对应的字符串后跳出主函数并结束程序。

也可以将 `switch` 的判断结构，改为第三种形式的判断语句。

**【例 4.10】** 根据输入的字符输出字符串。(实例位置：光盘\TM\sl\4\10)

```

#include<iostream>
using namespace std;
void main()
{
    char ilnput;
    cin >> ilnput;
    if(ilnput == 'A')
    {
        cout << "very good" << endl;
        return ;
    }else if(ilnput == 'B')
    {
        cout << "good" << endl;
        return ;
    }else if(ilnput == 'C')
    {
        cout << "normal" << endl;
        return ;
    }else if(ilnput == 'D')
    {
        cout << "failure" << endl;
        return ;
    }else
        cout << "input error" << endl;
}

```

同样，本程序也是根据用户输入的字符不同输出不同的字符串。

`switch` 语句中每个 `case` 语句都使用 “`break;`” 语句跳出，该语句可以省略。由于程序默认执行程序是顺序执行，当语句匹配成功后，其后面的每条 `case` 语句都会被执行，而不进行判断。例如：

```

#include<iostream>
using namespace std;
void main()
{
    int ilnput;
    cin >> ilnput;
    switch(ilnput)

```

```

{
case 1:
    cout << "Monday" << endl;
case 2:
    cout << "Tuesday" << endl;
case 3:
    cout << "Wednesday" << endl;
case 4:
    cout << "Thursday" << endl;
case 5:
    cout << "Friday" << endl;
case 6:
    cout << "Saturday" << endl;
case 7:
    cout << "Sunday" << endl;
default:
    cout << "Input error" << endl;
}
}

```

当输入 1 时，程序运行结果如图 4.9 所示。

当输入 7 时，程序运行结果如图 4.10 所示。



图 4.9 运行结果 1

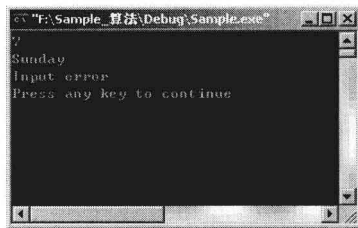


图 4.10 运行结果 2

程序想要实现根据输入的 1~7 中的任意整型数，然后输出整型数对应的英文星期名称，但由于 switch 语句中的各 case 分句没有及时使用“break;”语句跳出，导致意想不到的结果输出。

## 4.5 判断语句的嵌套

 视频讲解：光盘\TM\1x\4\判断语句的嵌套.exe

前面讲过 3 种形式的判断语句，这 3 种形式的判断语句都可以嵌套判断语句。例如，在第一种形式的判断语句中嵌套第二种形式的判断语句，形式如下：

```

if(表达式 1)
{
    if(表达式 2)
        语句 1;
    else

```



```

    语句 2;
}

```

在第二种形式的判断语句中嵌套第二种形式的判断语句，形式如下：

```

if(表达式 1)
{
    if(表达式 2)
        语句 1;
    else
        语句 2;
}
else
{
    if(表达式 2)
        语句 1;
    else
        语句 2;
}

```

判断语句可以有多种嵌套方式，可以根据具体需要进行设计，但一定要注意逻辑关系的正确处理。

**【例 4.11】** 判断是否是闰年。（实例位置：光盘\TM\sl\4\11）

```

#include<iostream>
using namespace std;
void main()
{
    int iYear;
    cout << "please input number" << endl;
    cin >> iYear;
    if(iYear%4==0)
    {
        if(iYear%100==0)
        {
            if(iYear%400==0)
                cout << "It is a leap year" << endl;
            else
                cout << "It is not a leap year" << endl;
        }
        else
            cout << "It is not a leap year" << endl;
    }
    else
        cout << "It is not a leap year" << endl;
}

```

判断闰年的方法是看该年份能否被 4 整除、不能被 100 整除但能被 400 整除。程序使用判断语句对这 3 个条件逐一判断，先判断年份能否被 4 整除—— $iYear \% 4 == 0$ ，如果不能整除则输出字符串 “It is not a leap year”，如果能整除，继续判断能否被 100 整除—— $iYear \% 100 == 0$ ，如果不能整除则输出字符串 “It is not a leap year”，如果能整除，继续判断能否被 400 整除—— $iYear \% 400 == 0$ ，如果能整除则输

出字符串 “It is a leap year”，不能整除则输出字符串 “It is not a leap year”。

可以简化判断是否是闰年的实例代码，用一条判断语句来完成。

**【例 4.12】** 判断是否是闰年。（实例位置：光盘\TM\sl\4\12）

```
#include<iostream>
using namespace std;
void main()
{
    int iYear;
    cout << "please input number" << endl;
    cin >> iYear;
    if(iYear%4==0 && iYear%100!=0 || iYear%400==0)
        cout << "It is a leap year" << endl;
    else
        cout << "It is not a leap year" << endl;
}
```

程序中将能否被 4 整除、不能被 100 整除但能被 400 整除这 3 个条件用一个表达式来完成。表达式是一个复合表达式，进行了 3 次算术运算和两次逻辑运算，算术运算判断能否被整除，逻辑运算判断是否满足 3 个条件。

使用判断语句嵌套时要注意 else 关键字要和 if 关键字成对出现，并且遵守临近原则，即 else 关键字和自己最近的 if 语句构成一对。另外，判断语句应尽量使用复合语句，以免产生二义性，导致书写格式的运行结果和设计时的不一致。

## 4.6 小 结

本章主要讲解了 C++语言中各种形式的判断语句，每种形式的语句都可以用另外一种格式代替，这增加了开发程序的灵活性。如果是简单的判断建议用条件运算符，如果是分支较多的逻辑判断，建议使用 switch 语句，还要特别注意判断语句的书写格式，避免产生二义性。

## 4.7 实践与练习

1. 编写一个程序，计算增加后的工资。（答案位置：光盘\TM\sl\4\13）

要求：基本工资大于或等于 5000 元，增加 10%工资；

若大于或等于 2500 元，且小于 5000 元，则增加 15%工资；

若小于 2500 元，则增加 20%工资。

2. 设计一个程序，要求从键盘上输入两个任意的整数，输出其中较大的数。（答案位置：光盘\TM\sl\4\14）

3. 任意输入 3 个整数，编程实现对这 3 个整数进行由小到大排序。（答案位置：光盘\TM\sl\4\15）

# 第 5 章

---

## 循环语句

(  视频讲解：53 分钟 )

循环控制就是控制程序重复执行，当不符合循环条件时停止循环。使用循环结构可以使程序代码更加简洁，减少冗余。掌握循环结构是程序设计的最基本要求，本章主要介绍了 while 循环、do...while 循环和 for 循环语句，这 3 种循环语句可以相互转换，达到同一目标可以运用多种方法。

通过阅读本章，您可以：

- » 了解 3 种循环语句
- » 掌握各种循环的区别
- » 了解循环的跳转
- » 掌握循环的嵌套

## 5.1 while 循环

 视频讲解：光盘\TM\lx\5\while 循环.exe

while 循环语句的一般形式如下：

**while(表达式) 语句**

表达式一般是一个关系表达式或一个逻辑表达式，表达式的值应该是一个逻辑值真或假（true 和 false），当表达式的值为真时开始循环执行语句，当表达式的值为假时退出循环，执行循环外的下一条语句。循环每次都是执行完语句后回到表达式处重新开始判断，重新计算表达式的值，一旦表达式的值为假时就退出循环，为真时就继续执行语句。while 循环可以用流程来演示执行过程，如图 5.1 所示。

语句可以是复合语句，也就是用大括号括起多条简单语句，大括号及其所包括的语句，被称为循环体，循环主要指循环执行循环体的内容。

**【例 5.1】** 使用 while 循环计算从 1 到 10 的累加。（实例位置：光盘\TM\sl\5\1）

1 到 10 的累加就是计算  $1+2+\cdots+10$ ，需要有一个变量从 1 变化到 10，将该变量命名为  $i$ ；还需要另外一个临时变量不断和该变量进行加法运算，并记录运算结果，将临时变量命名为  $sum$ ，变量  $i$  每增加 1 时，就和变量  $sum$  进行一次加法运算，变量  $sum$  记录的是累加的结果。程序需要使用循环语句，使用 while 循环需要将循环语句的结束条件设置为  $i \leq 10$ ，循环流程如图 5.2 所示。

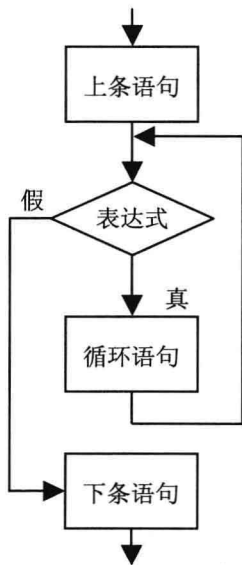


图 5.1 while 循环

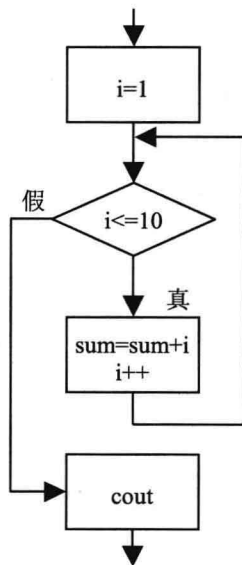


图 5.2 使用 while 循环计算从 1 到 10 的累加

程序代码如下：

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

```

void main()
{
    int sum=0,i=1;
    while(i<=10)
    {
        sum=sum+i;
        i++;
    }
    cout << "the result : " << sum << endl;
}

```

程序运行结果如图 5.3 所示。

程序先对变量 `sum` 和 `i` 进行初始化, `while` 循环语句的表达式是 `i<=10`, 所要执行的循环体是一个复合语句, 由“`sum=sum+i;`”和“`i++;`”两条简单语句构成, 语句“`sum=sum+i;`”完成累加, 语句“`i++;`”完成由 1 到 10 的递增变化。



图 5.3 程序运行结果

使用 `while` 循环的注意事项如下:

- (1) 表达式不可以为空, 表达式为空不合法。
- (2) 表达式可以用非 0 代表逻辑值真 (true), 用 0 代表逻辑值假 (false)。
- (3) 循环体中必须有改变条件表达式值的语句, 否则将成为死循环。

例如:

```

while(1)
{
    ...
}

```

是一个无限循环语句。

例如:

```

while(0)
{
    ...
}

```

是一个不会进行循环的语句。

## 5.2 do...while 循环

 视频讲解: 光盘\TM\lx\5\do...while 循环.exe

do...while 循环语句的一般形式如下:

```
do
语句
while(表达式)
```

do 为关键字，必须与 while 配对使用。do 与 while 之间的语句称为循环体，该语句同样是用大括号“{}”括起来的复合语句。循环语句中的表达式与 while 语句中的相同，也多为关系表达式或逻辑表达式。但特别值得注意的是 do...while 语句后要有分号“;”。do...while 循环可以用流程来演示执行过程，如图 5.4 所示。

do...while 循环的执行顺序是先执行循环体的内容，然后判断表达式的值，如果表达式的值为真就跳到循环体处继续执行循环体，循环一直到表达式的值为假。表达式的值为假时跳出循环，执行下一条语句。

**【例 5.2】** 使用 do...while 循环计算 1 到 10 的累加。（实例位置：光盘\TM\sl\5\2）

1 到 10 的累加就是计算  $1+2+\dots+10$ ，前面的实例使用 while 循环语句实现了 1 到 10 的累加，do...while 循环和 while 循环实现累加的循环体语句相同，只是执行循环体的先后顺序不同，do...while 循环程序执行顺序如图 5.5 所示。

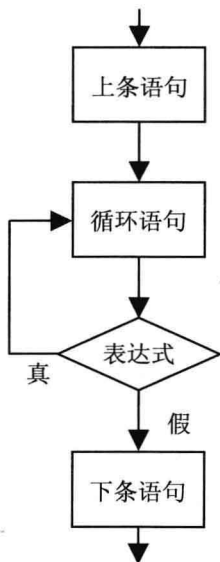


图 5.4 do...while 循环

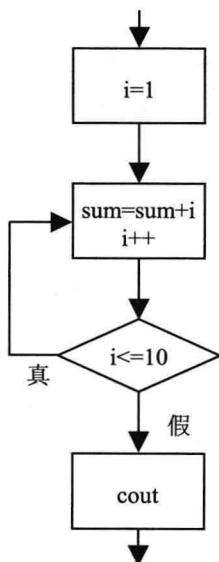


图 5.5 使用 do...while 循环计算 1 到 10 的累加

程序代码如下：

```
#include<iostream>
using namespace std;
void main()
{
    int sum=0,i=1;
    do
    {
        sum=sum+i;
        i++;
    }while(i<=10);
```



```
cout << "the result : " << sum << endl;
}
```

程序运行结果如图 5.6 所示。

程序使用变量 `sum` 作为记录累加的结果，变量 `i` 完成由 1 到 10 的变化，程序先将变量 `sum` 初始化为 0，将变量 `i` 初始化为 1，先执行循环体变量 `sum` 和变量 `i` 的加法运算，并将运算结果保存到变量 `sum`，然后变量 `i` 进行自加运算，接着判断循环条件，看变量 `i` 的值是否已经大于 10，如果变量 `i` 大于 10 就跳出循环，小于或等于 10 就继续执行循环体语句。

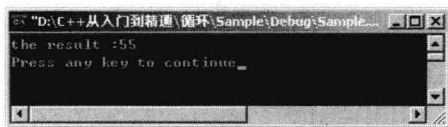


图 5.6 程序运行结果

do...while 循环的注意事项如下：

- (1) 循环先执行循环体，如果循环条件不成立，循环体已经执行一次了，使用时注意变量变化。
- (2) 表达式不可以为空，表达式为空不合法。
- (3) 表达式可以用非 0 代表逻辑值真 (true)，用 0 代表逻辑值假 (false)。
- (4) 循环体中必须有改变条件表达式值的语句，否则将成为死循环。
- (5) 循环语句后要有分号 “;”。

## 5.3 while 与 do...while 比较

 视频讲解：光盘\TM\lx\5\while 与 do...while 比较.exe

可以通过设置起始循环条件不成立循环语句，来观察 while 和 do...while 的不同。将变量 `i` 初始值设置为 0，然后将循环表达式设置为 `i>1`，显然循环条件不成立。循环体执行的是对变量 `j` 的加 1 运算，通过输出变量 `j` 在循环前的值和循环后的值来进行比较。

**【例 5.3】** 使用 do...while 循环进行计算。(实例位置：光盘\TM\sl\5\3)

使用 do...while 循环进行计算，代码如下：

```
#include<iostream>
using namespace std;
void main()
{
    int i=0,j=0;
    cout << "before do_while j=" << j << endl;
    do
    {
        j++;
    }while(i>1);
    cout << " after do_while j=" << j << endl;
}
```

程序运行结果如图 5.7 所示。

**【例 5.4】** 使用 while 循环进行计算。(实例位置：光盘\TM\sl\5\4)



使用 while 循环进行计算，代码如下：

```
#include<iostream>
using namespace std;
void main()
{
    int i=0,j=0;
    cout << "before while j=" << j << endl;
    while(i>1)
    {
        j++;
    }
    cout << "after while j=" << j << endl;
}
```

程序运行结果如图 5.8 所示。

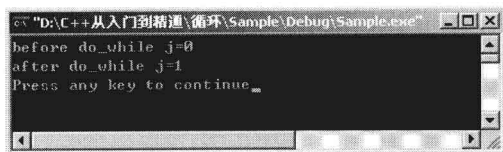


图 5.7 do...while 循环

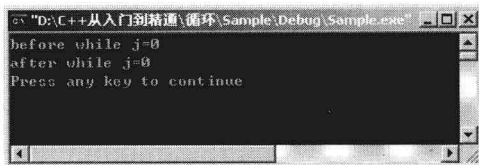


图 5.8 while 循环

使用 do...while 循环后变量 j 的值为 1，而使用 while 循环后变量 j 的值仍为 0。

## 5.4 for 循环语句

 视频讲解：光盘\TM\lx\5\for 循环语句.exe

for 循环语句的一般格式如下：

for(表达式 1;表达式 2;表达式 3) 语句

- ☑ 表达式 1：该表达式通常是一个赋值表达式，负责设置循环的起始值，也就是给控制循环的变量赋初值。
- ☑ 表达式 2：该表达式通常是一个关系表达式，用控制循环的变量和循环变量允许的范围值进行比较。
- ☑ 表达式 3：该表达式通常是一个赋值表达式，对控制循环的变量进行增大或减小。
- ☑ 语句：语句仍然是复合语句。

for 循环语句的执行过程如下：

- (1) 先求解表达式 1。
- (2) 求解表达式 2，若其值为真，则执行 for 语句中指定的内嵌语句，然后执行 (3)。若表达式 2 值为 0，则结束循环，转到 (5)。

- (3) 求解表达式 3。
- (4) 返回 (2) 继续执行。
- (5) 循环结束，执行 for 语句下面的一条语句。

上面的 5 个步骤也可以用图 5.9 表示。

**【例 5.5】** 用 for 循环计算从 1 到 10 的累加。(实例位置：光盘\TM\sl\5\5)

for 循环不同于 while 循环和 do...while 循环，它有 3 个表达式，需要正确设置这 3 个表达式。计算累加需要一个能由 1 到 10 递增变化的变量  $i$  和一个记录累加和的变量  $sum$ ，for 循环的表达式中可以对变量进行初始化，以及实现变量由 1 到 10 的递增变化。循环执行顺序如图 5.10 所示。

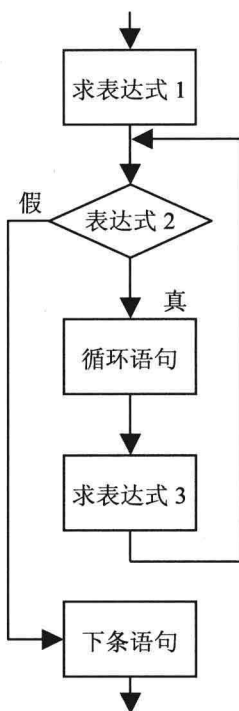


图 5.9 for 循环执行过程

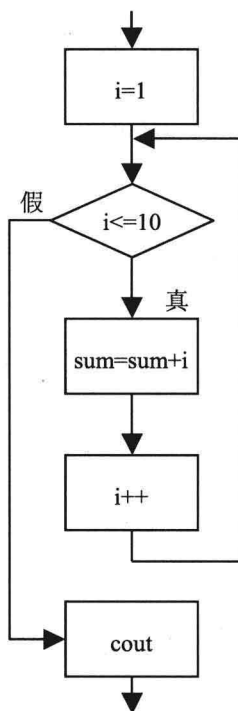


图 5.10 for 循环执行顺序

程序代码如下：

```

#include<iostream>
using namespace std;
void main()
{
    int sum=0;
    int i;
    for(i=1;i<=10;i++)    //for 循环语句
        sum+=i;
    cout << "the result : " << sum << endl;
}
  
```

程序运行结果如图 5.11 所示。

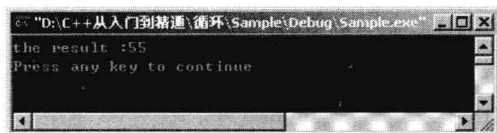


图 5.11 运行结果

程序中“for(i=1;i<=10;i++) sum+=i;”就是一个循环语句，“sum+=i;”是循环体语句，其中 i 就是控制循环的变量，i=0 是表达式 1，i<=10 是表达式 2，i++是表达式 3；表达式 1 将循环控制变量 i 赋初始值为 0，表达式 2 中 10 是循环变量允许的范围，也就是说 i 不能大于 10，大于 10 时将不执行语句“sum+=i;”。语句“sum+=i;”是使用了带运算的赋值语句，它等同于语句“sum = sum + i;”。“sum+=i;”语句一共执行了 10 次，i 的值是从 1 到 10 变化，完成 1 到 10 的累加。

for 循环的注意事项如下：

(1) for 语句可以在表达式 1 中直接声明变量。

在表达式外声明变量。例如：

```
#include<iostream>
using namespace std;
void main()
{
    int sum=0,i;           //在表达式外声明变量
    for(i=0;i<=10;i++)
        sum+=i;
    cout <<sum << endl;
}
```

在表达式内声明变量。例如：

```
#include<iostream>
using namespace std;
void main()
{
    for(int i=0,sum=0;i<=10;i++) //在表达式内声明变量
        sum+=i;
    cout <<sum << endl;
}
```

在循环语句中声明变量，也相当于在函数内声明了变量，如果在表达式 1 中声明两个相同变量，编译器将报错。例如：

```
void main()
{
    for(int i=0,sum=0;i<=10;i++) //在循环语句中声明变量
        sum+=i;
    for(int i=0,sum=0;i<=10;i++) //不合法，编译器报错
        sum+=i;
    cout <<sum << endl;
}
```

(2) for 循环中的表达式 1、表达式 2、表达式 3 都可以省略。

#### ☑ 省略表达式 1

如果省略表达式 1，且控制变量在循环外声明了并赋初值，程序能编译通过并且正确运行。例如：

```
#include<iostream>
using namespace std;
void main()
{
    int sum=0;
    int i=0;                //将循环控制变量拿到循环语句外声明并赋初值
    for(;i<=10;i++)
        sum+=i;
    cout <<sum << endl;
}
```

程序仍是计算从 1 到 10 的累加的。

如果控制变量在循环外声明了但没有赋初值，程序能编译通过，但运行结果不是用户所期待的。因为编译器会为变量赋一个默认的初值，该初值一般为一个比较大的负数，所以会造成运行结果不正确。

#### ☑ 省略表达式 2

省略了表达式 2 也就是省略了循环判断语句，即没有循环的终止条件，循环变成无限循环。

#### ☑ 省略表达式 3

省略表达式 3 后循环也是无限循环，因为控制循环的变量永远都是初始值，永远符合循环条件。

#### ☑ 省略表达式 1 和表达式 2

for 循环语句如果省略表达式 1 和表达式 2，就和 while 循环一样了。例如：

```
#include<iostream>
using namespace std;
void main()
{
    int sum=0;
    int i=0;
    for(;i<=10;)
    {
        sum=sum+i;
        i++;
    }
    cout << "the result :." << sum << endl;
}
```

#### ☑ 3 个表达式同时省略

for 循环语句如果省略 3 个表达式，就会变成无限循环。无限循环就是死循环，它会使程序进入瘫痪状态。使用循环时，建议使用计数控制，也就是说循环执行到指定次数，就跳出循环。例如：

```
void main()
{
    int iCount=0;           //声明用于计数的变量
```



```

for(;;)
{
    ...
    iCount++;           //每循环一次，计数器加一
    if(iCount>200000)    //如果循环次数大于 200000，跳出循环
        return;
}
cout << "the loop end" << endl;
}

```

## 5.5 循环控制

 视频讲解：光盘\TMlx\5\循环控制.exe

循环控制包含两方面的内容，一方面是控制循环变量的变化方式，一方面是控制循环的跳转。控制循环的跳转需要用到 `break` 和 `continue` 两个关键字，这两条跳转语句的跳转效果不同，`break` 是中断循环，`continue` 是跳出本次循环体的执行。

### 5.5.1 控制循环的变量

无论是 `for` 循环还是 `while`、`do...while` 循环，都需要循环一个控制循环的变量，`while`、`do...while` 循环的控制变量变化可以是显式的也可以是隐式的。例如在读取文件时，在 `while` 循环中循环读取文件内容，但程序中没有出现控制变量。代码如下：

```

#include<iostream>
#include<fstream>
using namespace std;
void main()
{
    ifstream ifile("test.dat",std::ios::binary);
    if(!ifile.fail())
    {
        while(!ifile.eof())           //判断文件是否结束
        {
            char ch;
            ifile.get(ch);             //获取文件内容
            if(!ifile.eof())           //如果是文件结束，就不进行最后输出
                std::cout << ch;
        }
    }
}

```

程序中 `while` 循环中的表达式是判断文件指针是否指向文件末尾，如果文件指针指向文件末尾，就跳出循环。起始程序中控制循环的变量是文件的指针，文件的指针在读取文件时不断变化。

for 循环的循环控制变量的变化方式有两种，一个是递增方式，一个是递减方式。使用递增方式还是递减方式和变量的初值和范围值的比较有关。

如果初值大于限定范围的值，表达式 2 是大于关系 (>) 判定的不等式，使用递减方式。

如果初值小于限定的范围值，表达式 2 是小于关系 (<) 判定的不等式，使用递增方式。

前文使用 for 循环计算 1 到 10 的累计和使用的是递增方式，也可以使用递减方式计算 1 到 10 的累计和。代码如下：

```
#include<iostream>
using namespace std;
void main()
{
    int sum=0;    //定义存储累加和变量
    for(int i=10;i>=1;i--)
        sum+=i; //进行累加
    cout << "the result :"<<sum << endl;
}
```

程序中 for 循环的表达式 1 中声明变量并赋初值 10，表达式 2 中限定范围的值就是 1，不等式是循环控制变量 i 是否大于等于 1，如果小于 1 就停止循环，循环控制变量就是由 10 到 1 递减变化。程序输出结果仍是 “the result :55”。

## 5.5.2 break 语句

使用 break 语句可以跳出 switch 结构。在循环结构中，同样也可用 break 语句跳出当前循环体，从而中断当前循环。

在 3 种循环语句中使用 break 语句的形式如图 5.12 所示。

<pre>while(...) {     ...     break;     ... }</pre>	<pre>do {     ...     break;     ... }while(...);</pre>	<pre>for {     ...     break;     ... }</pre>
--	---	---

图 5.12 break 语句的使用形式

**【例 5.6】** 使用 break 跳出循环。（实例位置：光盘\TM\sl\5\6）

```
#include<iostream>
using namespace std;
void main()
{
    int i,n,sum;
    sum=0;
    cout<<"input 10 number" << endl;
    for(i=1;i<=10;i++)
    {
        cout<< i<< ". " << " ";
    }
```

```

        cin >> n;
        if(n<0)    //判断输入是否为负数
            break;
        sum+=n; //对输入的数进行累加
    }
    cout << "The Result : " sum << endl;
}

```

程序中需要用户输入 10 个数，然后计算 10 个数的和。当输入数为负数时，就停止循环不再进行累加，输出前面累加结果。例如输入 4 次数字 1，最后输入数字-1，程序运行结果如图 5.13 所示。

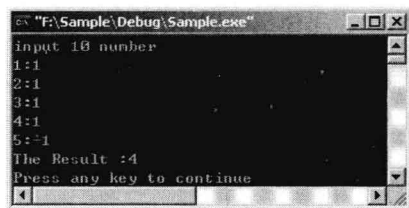


图 5.13 运行结果



#### 注意

如果遇到循环嵌套的情况，break 语句将只会使程序流程跳出包含它的最内层的循环结构，只跳出一层循环。

### 5.5.3 continue 语句

continue 语句是针对 break 语句的补充。continue 不是立即跳出循环体，而是跳过本次循环结束前的语句，回到循环的条件测试部分，重新开始执行循环。在 for 循环语句中遇到 continue 后，首先执行循环的增量部分，然后进行条件测试。在 while 和 do...while 循环中，continue 语句使控制直接回到条件测试部分。

在 3 种循环语句中使用 continue 语句的形式如图 5.14 所示。

```

while(...)    do        for
{
    ...
    continue;    continue;    continue;
    ...
}                }while(...);    }

```

图 5.14 continue 语句的使用形式

**【例 5.7】** 使用 continue 跳出循环。（实例位置：光盘\TM\sl\5\7）

```

#include<iostream>
using namespace std;
void main()
{
    int i,n,sum;
    sum=0;

```



```

cout<< "input 10 number" << endl;
for(i=1;i<=10;i++)
{
    cout<< i<< ":" << ";
    cin >> n;
    if(n<0)    //判断输入是否为负数
        continue;
    sum+=n; //对输入的数进行累加
}
cout << "The Result : " sum << endl;
}

```

程序中需要用户输入 10 个数，然后计算 10 个数的和。当输入数为负数时，不执行“sum+=n;”语句，也就是不对负数进行累加。例如输入 10 个数全为 1，输出结果为 10。

### 5.5.4 goto 语句

goto 语句又称为无条件跳转语句，用于改变语句的执行顺序。goto 语句的一般格式为：

```
goto 标号;
```

其中，标号是用户自定义的一个标识符，以冒号结束。下面利用 goto 语句实现 1 到 100 的累加求和。

**【例 5.8】** 使用 goto 语句实现循环。(实例位置：光盘\TM\sl\5\8)

```

#include<iostream>
using namespace std;
void main()
{
    int ivar = 0;           //定义一个整型变量，初始化为 0
    int num = 0;           //定义一个整型变量，初始化为 0
label:                    //定义一个标签
    ivar++;               //ivar 自加 1
    num += ivar;          //累加求和
    if(ivar < 10)         //判断 ivar 是否小于 10
    {
        goto label;      //转向标签
    }
    cout << num << endl;
}

```

程序中利用标签实现循环功能。当语句执行到“if(ivar<100)”时，如果条件为真，跳转转到标签定义“label:”处。这是一种古老的跳转语句，它会使程序的执行顺序变得混乱，CPU 需要不停地跳转，效率比较低，因此，在开发程序时慎用 goto 语句。

在使用 goto 语句时的说明：

(1) 使用 goto 语句时，应注意标签的定义。在定义标签时，其后不能紧接着出现“}”符号。例如，下面的代码是非法的。

```

int ivar = 0;           //定义一个整型变量，初始化为 0
int num = 0;           //定义一个整型变量，初始化为 0
{
    //其他操作
label:                 //定义一个标签
}

```

在上述代码中定义标签时，其后没有执行代码了，所以出现编译错误。如果程序中出现上述情况，可以在标签后添加一条语句，以解决编译错误。

(2) 在使用 goto 语句时还应注意 goto 语句不能越过复合语句之外的变量定义的语句。例如，下面的 goto 语句是非法的。

```

goto label;           //跳转到标签
int i = 10;           //声明一个变量，初始化为 10
label:                //定义标签
    cout<<"goto" << endl; //输出信息

```

在上述代码中 goto 语句试图越过变量 i 的定义，导致编译错误。解决上述问题的方法是将变量的声明放在复合语句中。例如下面的代码将是合法的。

```

goto label;           //跳转到标签
{
    int i = 10;        //声明一个变量，初始化为 10
}
label:                //定义标签
    cout<<"goto"<< endl; //输出信息

```

## 5.6 循环嵌套

 视频讲解：光盘\TM\lx\5\循环嵌套.exe

循环有 for、while、do...while 3 种方式，这 3 种循环可以相互嵌套。例如，在 for 循环中套用 for 循环。

```

for(...)
{
    for(...)
    {
        ...
    }
}

```

在 while 循环中套用 while 循环。

```

while(...)
{
    while(...)

```

```

{
    ...
}

```

在 while 循环中套用 for 循环。

```

while(...)
{
    for(...)
    {
        ...
    }
}

```

### 【例 5.9】 打印三角形。(实例位置：光盘\TM\sl\5\9)

使用嵌套的 for 循环来输出由字符\*组成的三角形。

```

#include<iostream>
using namespace std;
void main()
{
    int i, j, k;
    for(i = 1; i <= 5; i++)           //控制行数
    {
        for(j = 1; j <= 5-i; j++)     //控制空格数
            cout << " ";
        for(k = 1; k <= 2 * i - 1; k++) //控制打印*号的数量
            cout << "*";
        cout << endl;
    }
}

```

程序中一共输出 5 行字符，最外面的 for 循环控制输出的行数，嵌套的第一个循环控制字符\*前的空格数，第二个 for 循环控制输出字符\*的个数。第一个循环随着行数的增加，字符\*前的空格数越来越少；第二个循环输出和行号有关的奇数个字符\*。程序运行结果如图 5.15 所示。

### 【例 5.10】 输出乘法口诀表。(实例位置：光盘\TM\sl\5\10)

使用嵌套的 for 循环输出乘法口诀表。

```

#include<iostream>
#include<iomanip>
using namespace std;
void main(void)
{
    int i, j;
    i=1;
    j=1;
    for(i=1; i<10; i++)
    {

```

```

    for(j=1;j<i+1;j++)
        cout << setw(2) << i << "*" << j << " = " << setw(2) << i*j;
    cout << endl;
}

```

程序使用了两层 for 循环，第一个循环由 1 到 9 变化，第二个循环则是控制随着行数的增加，列数也增加，最后形成第 9 行有 9 列。程序运行结果如图 5.16 所示。

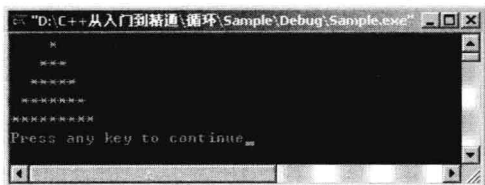


图 5.15 运行结果

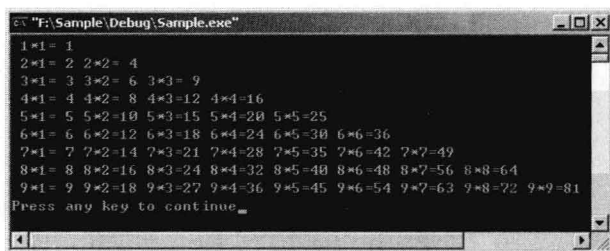


图 5.16 乘法口诀表

## 5.7 循环应用实例

 视频讲解：光盘\TM\lx\5\循环应用实例.exe

本节通过几个实例来演示不同循环的具体应用。

### 5.7.1 阿姆斯壮数

**【例 5.11】** 阿姆斯壮数。（实例位置：光盘\TM\sl\5\11）

在 3 位的整数中，形如  $153=1^3+5^3+3^3$  这样的数称为阿姆斯壮数。求阿姆斯壮数需要分别计算出 3 位整数的百位、十位和个位，然后对 3 个数分别求立方，最后判断是否为阿姆斯壮数。

```

#include<iostream>
using namespace std;
int main()
{
    int a,b,c;
    int input;
    for(input=100;input<=999;input++)
    {
        a=input / 100;           //求百位
        b=(input % 100) / 10;    //求十位
        c=input % 10;           //求个位
        if(a*a*a+b*b*b+c*c*c == input)
            cout << input << endl;
    }
}

```



```
    return 0;
}
```

程序运行结果如图 5.17 所示。

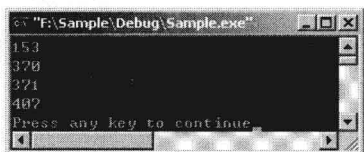


图 5.17 阿姆斯壮数

## 5.7.2 巴斯卡三角形

**【例 5.12】 巴斯卡三角形。**（实例位置：光盘\TM\sl\5\12）

巴斯卡三角形是两个边全输出 1，三角形的内部用上一行相邻两个数之和表示，如图 5.18 所示。

```
#include<iostream>
#include<iomanip>
using namespace std;
long combi(int n,int r)
{
    int i;
    long p=1;
    for(i=1;i<=r;i++)
        p=p*(n-i+1)/i;
    return p;
}
void main()
{
    int n,r,t;
    for(n=0;n<=12;n++)                //控制行数
    {
        for(r=0;r<=n;r++)
        {
            int i;
            if(r==0)
            {
                for(i=0;i<=(12-n);i++)
                    cout << " ";        //每行第一个元素的位置
            }
            else
                cout << " ";            //每个数之间空两个格
            cout << setw(3) << combi(n,r);
        }
        cout << endl;
    }
}
```

程序使用了循环嵌套来控制显示格式，变量  $n$  代表行数，变量  $r$  代表每行元素数，自定义函数 `combi` 计算每个位置应该放置的数。程序运行结果如图 5.18 所示。

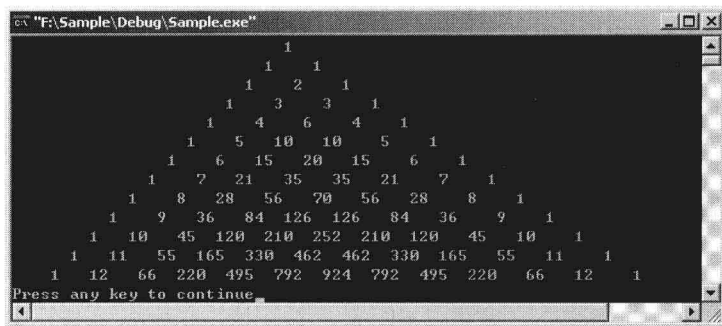


图 5.18 巴斯卡三角形

### 5.7.3 对输入的分数进行排名

**【例 5.13】** 对输入的分数进行排名。（实例位置：光盘\TM\sl\5\13）

通过输入不同的分数，然后计算不同分数之间的排名。

```
void main()
{
    int score[101]={0};
    int juni[102]={0};
    int count=0,i;
    do{
        cout << "Input score: " ;
        cin >> score[count++];
    }while(score[count-1]!=-1);
    count--;
    for(i=0;i<count;i++)
        juni[score[i]]++;
    juni[101]=1;
    for(i=100;i>=0;i--)
        juni[i]=juni[i]+juni[i+1];
    cout << "Result: " << endl;
    for(i=0;i<count;i++)
    {
        cout << score[i] << "is";
        cout << juni[score[i]+1] << endl;
    }
}
```

程序使用 score 数组记录输入的分数，使用 juni 数组记录排名。首先通过 do...while 循环获取输入的分数，最多只能输入 100 个分数，并且输入的分数不能超过 100 分；然后使用 for 循环进行排序，有 100 分就在 juni 数组 100 的位置标 1，接着使用 for 循环对 juni 数组中的数相邻两项进行累加。最后使用 for 循环将分数和排名输出。

程序运行结果如图 5.19 所示。



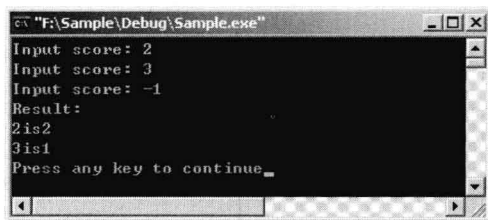


图 5.19 得分排名

## 5.8 小 结

本章主要介绍了 for、while、do...while 3 种循环，其中使用比较灵活的是 for 循环，比较简单的是 while 循环。同样一个目标使用 3 种循环都可以实现，最终选择哪种循环来实现要根据每个开发人员对需求的理解，但一般建议使用 for 循环。

## 5.9 实践与练习

1. 要求使用 for 循环，打印出大写字母的 ASCII 码对照表。（答案位置：光盘\TM\sl\5\14）
2. 输出 0~100 之间不能被 3 整除的数。提示：使用 for 进行循环检查操作，使用 continue 语句结束不符合条件的情况。（答案位置：光盘\TM\sl\5\15）
3. 用\*打印菱形。（答案位置：光盘\TM\sl\5\16）


```
*  
***  
*****  
*****  
*****  
***  
*
```



# 第 6 章

---

## 函数

(  视频讲解：1 小时 14 分钟 )

程序是由函数组成的，一个函数就是程序中的一个模块。函数可以相互调用，可以将相互联系密切的语句都放到一个函数内，也可以将复杂的函数分解成多个子函数。函数本身也有很多特点，熟练掌握函数的特点可以将程序的结构设计得更合理。

通过阅读本章，您可以：

- » 了解函数工作机制
- » 掌握函数调用
- » 掌握重载函数
- » 了解内联函数

## 6.1 函数概述

 视频讲解：光盘\TM\lx\6\函数概述.exe

函数就是能够实现特定功能的程序模块，它可以是只有一条语句的简单函数，也可以是包含许多子函数的复杂函数；函数有别人写好的存放在库里的库函数，也有开发人员自己写的自定义函数；函数根据功能可以分为字符函数、日期函数、数学函数、图形函数、内存函数等。一个程序可以只有一个主函数，但不可以没有函数。

### 6.1.1 函数的定义

函数定义的一般形式如下：

类型标识符 函数名(形式参数列表)

```
{  
    变量的声明  
    语句  
}
```

- ☑ 类型标识符：用来标识函数的返回值类型，可以根据函数的返回值判断函数的执行情况，通过返回值也可以获取想要的数。类型标识符可以是整型、字符型、指针型、对象的数据类型。
- ☑ 形式参数列表：由各种类型变量组成的列表，各参数之间用逗号间隔，在进行函数调用时，主调函数对变量进行赋值。

关于函数定义的一些说明：

(1) 形式参数列表可以为空，这样就定义了不需要参数的函数。例如：

```
int ShowMessage()  
{  
    int i=0;  
    cout << i << endl;  
    return 0;  
}
```

函数 ShowMessage 通过 cout 流输出变量 i 的值。

(2) 函数后面的大括号表示函数体，在函数体内进行变量的声明和添加实现语句。

### 6.1.2 函数的声明

调用一个函数前必须先声明函数的返回值类型和参数类型。例如：

```
int SetIndex(int i);
```

函数声明被称为函数原型，函数声明时可以省略变量名。例如：

```
int SetIndex(int);
```

下面通过实例来介绍如何在程序中声明、定义和使用函数。

**【例 6.1】** 声明、定义和使用函数。（实例位置：光盘\TM\sl\6\1）

```
#include<iostream>
using namespace std;
void ShowMessage();    //函数声明语句
void ShowAge();        //函数声明语句
void ShowIndex();      //函数声明语句
void main()
{
    ShowMessage();    //函数调用语句
    ShowAge();        //函数调用语句
    ShowIndex();      //函数调用语句
}
void ShowMessage()
{
    cout << "HelloWorld!" << endl;
}
void ShowAge()
{
    int iAge=23;
    cout << "age is :" << iAge << endl;
}
void ShowIndex()
{
    int iIndex=10;
    cout << "Index is :" << iIndex << endl;
}
```

程序运行结果如图 6.1 所示。

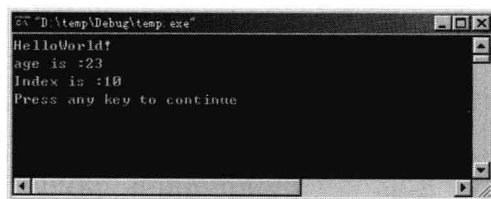


图 6.1 运行结果

程序定义和声明了 ShowMessage、ShowAge、ShowIndex 函数，并进行了调用，通过函数中的输出语句进行输出。

## 6.2 函数参数及返回值

 视频讲解：光盘\TM\lx\6\函数参数及返回值.exe

### 6.2.1 返回值

函数的返回值是指函数被调用之后，执行函数体中的程序段所取得的并返回给主调函数的值，函数的返回值通过 `return` 语句返回给主调函数。`return` 语句的一般形式如下：

```
return(表达式);
```

关于返回值的说明：

(1) 函数返回值的类型和函数定义中函数的类型标识符应保持一致。如果两者不一致，则以函数类型为准，自动进行类型转换。

(2) 如函数值为整型，在函数定义时可以省去类型标识符。

(3) 在函数中允许有多个 `return` 语句，但每次调用只能有一个 `return` 语句被执行，因此只能返回一个函数值。

(4) 不返回函数值的函数，可以明确定义为“空类型”，类型标识符为“`void`”。例如：

```
void ShowIndex()
{
    int iIndex=10;
    cout << "Index is :." << iIndex << endl;
}
```

(5) 类型标识符为 `void` 的函数不能进行赋值运算及值传递。例如：

```
i= ShowIndex();           //不能进行赋值
SetIndex(ShowIndex);      //不能进行值传递
```

 说明

为了降低程序出错的几率，凡不要求返回值的函数都应定义为空类型。

### 6.2.2 空函数

没有参数和返回值，函数的作用域也为空的函数就是空函数。

```
void setWorkSpace(){ }
```



调用此函数时，什么工作也不做，没有任何实际意义。在主函数 main 函数中调用 setWorkspace 函数时，这个函数没有起到任何作用。例如：

```
void setWorkspace(){ }
void main()
{
    setWorkspace();
}
```

空函数存在的意义是：在程序设计中往往根据需要确定若干模块，分别由一些函数来实现。而在第一阶段只设计最基本的模块，其他一些次要功能或锦上添花的功能则在以后需要时陆续补上。在编写程序的开始阶段，可以在将来准备扩充功能的地方写上一个空函数，这些函数没有开发完成，先占一个位置，以后用一个编好的函数代替它。这样做可以使程序的结构清楚，可读性好，以后扩充新功能方便，对程序结构影响不大。

### 6.2.3 形参与实参

函数定义时如果参数列表为空，说明函数是无参函数；如果参数列表不为空，就称为有参函数。有参函数中的参数在函数声明和定义时被称为“形式参数”（简称形参），在函数被调用时被赋予具体值，具体的值被称为“实际参数”（简称实参）。形参与实参如图 6.2 所示。

实参与形参的个数应相等，类型应一致。实参与形参按顺序对应，函数被调用时会一一传递数据。

形参与实参的区别：

（1）在定义函数中指定的形参，在未出现函数调用时，它们并不占用内存中的存储单元。只有在发生函数调用时，函数的形参才被分配内存单元，在调用结束后，形参所占的内存单元也被释放。

（2）实参应该是确定的值。在调用时将实参的值赋给形参，如果形参是指针类型，就将地址值传递给形参。

（3）实参与形参的类型应相同。

（4）实参与形参之间是单项传递，只能由实参传递给形参，而不能由形参传回来给实参。

实参与形参之间存在一个分配空间和参数值传递的过程，这个过程是在函数调用时发生的。C++ 支持引用型变量，引用型变量则没有值传递的过程，这将在后文讲到。

```

      形参  形参
int function(int a,int b);
void main()
{
      实参  实参
    function(3,4);

    cout << "the loop end" << endl;
}
int function(int a,int b)
{
    return a+b;
}
```

图 6.2 形参与实参

### 6.2.4 默认参数

在调用有参函数时，如果经常需要传递同一个值到调用函数，在定义函数时，可以为参数设置一个默认值，这样在调用函数时可以省略一些参数，此时程序将采用默认值作为函数的实际参数。下面

的代码定义了一个具有默认值参数的函数。

```
void OutputInfo(const char* pchData = "One world,one dream!")
{
    cout << pchData << endl;    //输出信息
}
```

**【例 6.2】** 调用默认参数的函数。（实例位置：光盘\TM\sl\6\2）

实例输出两行字符串，一行是函数默认参数，一行是通过传字符串实参。程序代码如下：

```
#include<iostream>
using namespace std;
void OutputInfo(const char* pchData = "One world,one dream!")
{
    cout << pchData << endl;    //输出信息
}
void main()
{
    OutputInfo();                //利用默认值作为函数实际参数
    OutputInfo("Beijing 2008 Olympic Games!");    //直接传递实际参数
}
```

程序运行结果如图 6.3 所示。

在定义函数默认参数时，如果函数具有多个参数，应保证默认参数出现在参数列表的右方，没有默认值的参数出现在参数列表的左方，即默认参数不能出现在非默认参数的左方。例如，下面的函数定义是非法的：

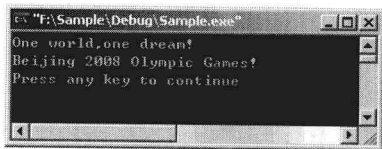


图 6.3 调用默认参数的函数

```
int GetMax(int x,int y=10 ,int z)    //非法的函数定义，默认参数 y 出现在参数 z 的左方
{
    if(x < y)                        //x 与 y 进行比较
        x = y;                      //赋值
    if(x < z)                        //x 与 z 进行比较
        x = z;                      //赋值
    return x;                       //返回 x
}
```

程序中默认参数 y 出现在非默认参数 z 的左方，导致了编译错误。正确的做法是将默认参数放置在参数列表的右方。例如：

```
int GetMax(int x,int y ,int z=10)    //定义默认参数
{
    if(x < y)                        //x 与 y 进行比较
        x = y;                      //赋值
    if(x < z)                        //x 与 z 进行比较
        x = z;                      //赋值
    return x;                       //返回 x
}
```

## 6.2.5 可变参数

库函数 `printf` 就是一个可变参数函数，它的参数列表会显示“...”。`printf` 函数原型格式如下：

```
_CRTIMP int _cdecl printf(const char *, ...);
```

“...”代表的含义是函数的参数是不固定的，可以传递一个或多个参数。对于 `printf` 函数来说，可以输出一项信息，也可以同时输出多项信息。例如：

```
printf("%d\n",2008); //输出一项信息
printf("%s-%s-%s\n","Beijing","2008","Olympic Games"); //输出多项信息
```

声明可变参数的函数和声明普通函数一样，只是参数列表中有一个“...”。例如：

```
void OutputInfo(int num,...) //定义可变参数函数
```

对于可变参数的函数，在定义函数时需要一一读取用户传递的实际参数。可以使用 `va_list` 类型和 `va_start`、`va_arg`、`va_end` 3 个宏读取传递到函数中的参数值。使用可变参数需要引用 `STDARG.H` 头文件。下面以一个具体的实例介绍可变参数函数的定义及使用。

**【例 6.3】** 定义并调用可变参数函数。（实例位置：光盘\TM\sl\6\3）

```
#include<iostream>
#include<STDARG.H> //需要包含该头文件
using namespace std;
void OutputInfo(int num,...) //定义一个可变参数函数
{
    va_list arguments; //定义 va_list 类型变量
    va_start(arguments,num);
    while(num-->0) //读取所有参数的数据
    {
        char* pchData = va_arg(arguments,char*); //获取字符串数据
        int iData = va_arg(arguments,int); //获取整型数据
        cout<< pchData << endl; //输出字符串
        cout << iData << endl; //输出整数
    }
    va_end(arguments);
}

void main()
{
    OutputInfo(2,"Beijing",2008,"Olympic Games",2008); //调用 OutputInfo 函数
}
```

程序运行结果如图 6.4 所示。

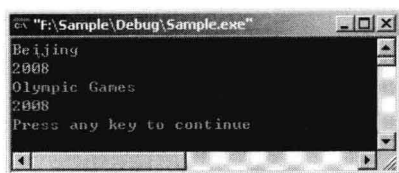


图 6.4 调用可变参数函数

## 6.3 函数调用

 视频讲解：光盘\TM\lx\6\函数调用.exe

声明完函数后就需要在源代码中调用该函数。整个函数的调用过程被称为函数调用。标准 C++ 是一种强制类型检查的语言，在调用函数前，必须把函数的参数类型和返回值类型告知编译。

函数调用的一些说明：

- (1) 首先被调用的函数必须是已经存在的函数（是库函数或用户自己定义的函数）。
- (2) 如果使用库函数，还需要将库函数对应的头文件引入，这需要使用预编译指令 `#include`。
- (3) 如果使用用户自定义函数，一般还应该在主调函数中对被调用的函数作声明。



**注意**

C++ 和 C 不同，简单地用不同的原型来说明同一个函数是无法通过 C++ 语法检查的。

### 6.3.1 传值调用

主调函数和被调用函数之间有数据传递关系，换句话说，主调函数将实参数值复制给被调用函数的形参处，这种调用方式被称为传值调用。如果传递的实参是结构体对象，值传递方式的效率是低下的，可以通过传指针或使用变量的引用来替换传值调用。传值调用是函数调用的基本方式。

**【例 6.4】** 使用传值调用。（实例位置：光盘\TM\sl\6\4）

```
#include<iostream.h>
void swap(int a,int b)
{
    int tmp;
    tmp=a;
    a=b;
    b=tmp;
}
void main()
{
    int x,y;
```



```

cout << "输入两个数" << endl;
cin >> x;
cin >> y;

if(x<y)
    swap(x,y);
cout << "x=" << x << endl;
cout << "y=" << y << endl;
}

```

程序运行结果如图 6.5 所示。

程序本意是想实现当  $x$  小于  $y$  时交换  $x$  和  $y$  的值，但结果并没有实现，主要原因是调用 `swap` 函数时复制了变量  $x$  和  $y$  的值，而并非变量本身。如果将函数 `swap` 在调用处展开，程序本意就可以实现。例如修改代码如下：

```

#include<iostream>
using namespace std;
void main()
{
    int x,y;
    cout << "输入两个数" << endl;
    cin >> x;
    cin >> y;
    int tmp;
    if(x<y)
    {
        tmp=x;
        x=y;
        y=tmp;
    }
    cout << "x=" << x << endl;
    cout << "y=" << y << endl;
}

```

程序运行如图 6.6 所示。

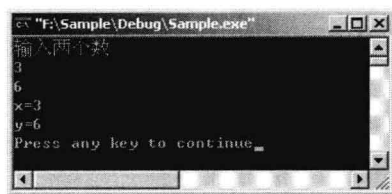


图 6.5 使用传值调用

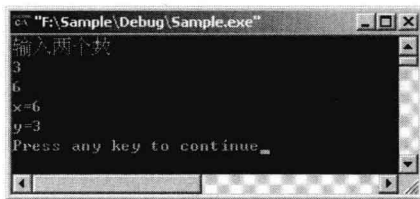


图 6.6 展开函数调用

程序代码是开发人员模拟函数调用时展开 `swap` 函数的代码，函数的调用就是由编译器来完成代码的展开工作，但不是真的展开，而是移到 `swap` 函数处执行，执行过程类似展开。使用函数调用时要注意函数调用时有值传递过程。通过函数调用的方式实现交换变量的值，可以通过使用指针传地址和变量引用的方式实现，这在后面的章节会讲到。

函数调用中发生的数据传递是单向的，只能把实参的值传递给形参，在函数调用过程中，形参的值发生改变，实参的值不会发生变化。

### 6.3.2 嵌套调用

在自定义函数中调用其他自定义函数，这种调用方式称为嵌套调用。例如：

```
#include<iostream>
using namespace std;
void ShowMessage()           //定义函数
{
    cout <<"The ShowMessage function" << endl;
}

void Display()
{
    ShowMessage();           //嵌套调用
}

void main()
{
    Display();
}
```

在函数嵌套调用时要注意，不要在函数体内定义函数，如下代码便是错误的：

```
int main()
{
    void Display()           //错误，不能在函数体内定义函数
    {
        cout << "I want to show the Nesting function" << endl;
    }
    return 0;
}
```

嵌套调用对调用的层数是没有要求的，但个别的编译器可能会有一些限制，使用时应注意。

### 6.3.3 递归调用

直接或间接调用自己的函数被称为递归函数（recursive function）。

使用递归方法解决问题的特点是：问题描述清楚、代码可读性强、结构清晰，代码量比使用非递归方法少。缺点是递归程序的运行效率比较低，无论是从时间角度还是从空间角度都比非递归程序差。对于时间复杂度和空间复杂度要求较高的程序，使用递归函数调用要慎重。

递归函数必须定义一个停止条件，否则函数将永远递归下去。



**【例 6.5】 汉诺 (Hanoi) 塔问题。(实例位置: 光盘\TM\sl\6\5)**

有 3 个立柱垂直矗立在地面, 给这 3 个立柱分别命名为 A、B、C。开始时立柱 A 上有 64 个圆盘, 这 64 个圆盘大小不一, 并且按从小到大的顺序依次摆放在立柱 A 上, 如图 6.7 所示。现在的问题是要将立柱 A 上的 64 个圆盘移到立柱 C 上, 并且每次只允许移动一个圆盘, 在移动过程中始终保持大盘在下, 小盘在上。

分析程序:

先假设移动 4 个圆盘, 立柱 A 上的圆盘按由上到下的顺序分别命名为 a、b、c、d, 如图 6.7 所示。

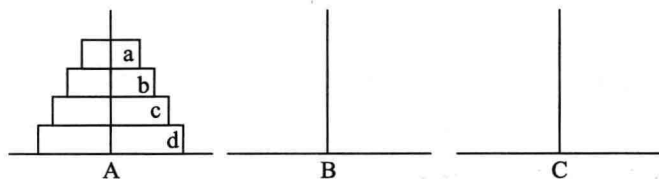


图 6.7 圆盘原始状态

先考虑将 a 和 b 移动到立柱 C 上。移动顺序是 a→B, b→C, a→C, 移动结果如图 6.8 所示。

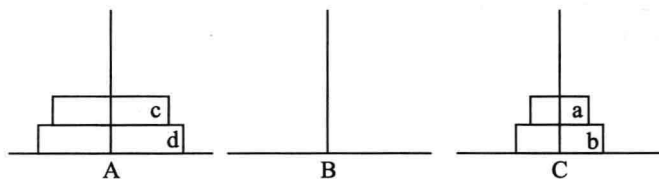


图 6.8 移动两个圆盘到目标

如果要将 c 也移动到 C 上, 就要暂时将 c 移动到 B, 然后再移动 a 和 b。移动顺序是 c→B, a→A, b→B, a→B, d→c, 移动结果如图 6.9 所示。

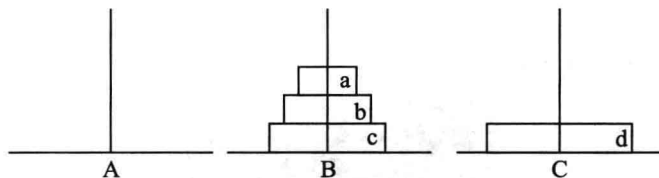


图 6.9 移动 3 个圆盘到目标

最后是完成 4 个圆盘的移动, 移动顺序是 a→C, b→A, a→A, c→C, a→B, b→C, a→C。

总结一下:

要将 4 个圆盘移动到指定立柱总共需要移动 15 次。

在移动过程中将两个圆盘移动到指定立柱需要移动 3 次, 分别是 a→B, b→C, a→C。

在移动过程中将 3 个圆盘移动到指定立柱需要移动 7 次, 分别是 a→B, b→C, a→C, c→B, a→A, b→B, a→B。

移动次数可以总结为是  $2^n - 1$  次。

在移动过程中可以将 a、b、c 3 个圆盘看成是一个圆盘, 移动 4 个圆盘的过程就像是在移动两个圆盘。还可以将 a、b、c 这 3 个圆盘中的 a、b 两个圆盘看成是一个圆盘, 移动 3 个圆盘也像是在移动两

个圆盘。可以使用递归的思路来移动  $n$  个圆盘。

移动  $n$  个圆盘可以分成 3 个步骤：

- (1) 把 A 上的  $n-1$  个圆盘移到 B 上。
- (2) 把 A 上的一个圆盘移到 C 上。
- (3) 把 B 上的  $n-1$  个圆盘移到 C 上。

程序代码如下：

```
#include<iostream>
using namespace std;
long lCount;
void move(int n,char x,char y,char z)    //将 n 个圆盘从 x 针借助 y 针移到 z 针上
{
    if(n==1)
        cout << "Times:" << ++lCount << x << "-" << z << endl;
    else
    {
        move(n-1,x,z,y);
        cout << "Times:" << ++lCount << x << "-" << z << endl;
        move(n-1,y,x,z);
    }
}
void main()
{
    int n ;
    lCount=0;
    cout << "please input a number" << endl;
    cin >> n ;
    move(n,'a','b','c');
}
```

程序运行结果如图 6.10 所示。

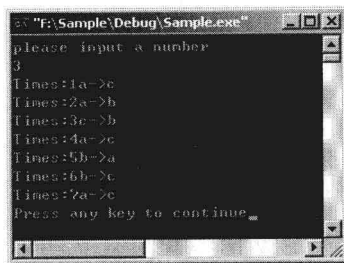


图 6.10 汉诺塔问题

输入数字 3，表示移动 3 个圆盘，程序打印出挪动 3 个圆盘的步骤。

**【例 6.6】** 求  $n$  的阶乘。（实例位置：光盘\TM\sl\6\6）

```
#include<iostream>
using namespace std;
long Fac(int n)
```

```

{
    if(n==0)
        return 1;
    else
        return n*Fac(n-1);
}
void main()
{
    int n ;
    long f;
    cout << "please input a number" << endl;
    cin >> n ;
    f=Fac(n);
    cout << "Result :." << f << endl;
}

```

程序运行结果如图 6.11 所示。

程序中 Fac 函数实现了计算  $n$  的阶乘。以  $n$  等于 4 为例,  $4!$  等于  $4*3!$ ,  $3!$  等于  $3*2!$ , …… ,  $1!$  等于 1。当计算 4 的阶乘时, 只要知道 3 的阶乘就可以了,  $4*3!$  等于  $4!$ 。同理, 计算 3 的阶乘, 只要知道 2 的阶乘就可以了, 依此类推。1 的阶乘为 1, 知道了 1 的阶乘, 就可以计算 2 的阶乘, 知道 2 的阶乘就可以计算 3 的阶乘……

在上面的递归函数中, 如果传递一个很大的数作为参数, 会导致堆栈溢出, 因为每调用一个函数, 系统会为函数的参数分配堆栈空间。对于上述的递归函数 Fac, 完全可以用连续乘积的方式实现。

**【例 6.7】** 利用循环求  $n$  的阶乘。(实例位置: 光盘\TM\sl\6\7)

```

#include<iostream>
using namespace std;
typedef unsigned int UINT;           //自定义类型
long Fac(const UINT n)               //定义函数
{
    long ret = 1;                    //定义结果变量
    for(int i=1; i<=n; i++)          //累计乘积
    {
        ret *= i;
    }
    return ret;                      //返回结果
}

void main()
{
    int n ;
    long f;
    cout << "please input a number" << endl;
    cin >> n ;
    f=Fac(n);
    cout << "Result :." << f << endl;
}

```

程序运行结果如图 6.12 所示。

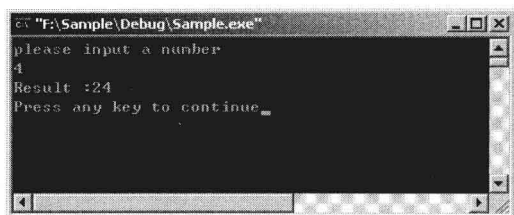


图 6.11 计算阶乘

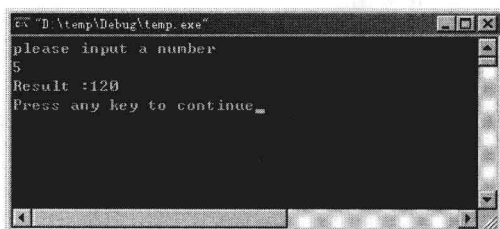


图 6.12 利用循环求 n 的阶乘

## 6.4 变量作用域

 视频讲解：光盘\TM\lx\6\变量作用域.exe

根据变量声明的位置可以将变量分为局部变量及全局变量，在函数体内定义的变量称为局部变量，在函数体外定义的变量称为全局变量。例如：

```
#include<iostream>
using namespace std;
int iTotalCount;           //全局变量
int GetCount();
void main()
{
    int iTotalCount=100;    //局部变量
    cout << iTotalCount << endl;
    cout << GetCount() << endl;
}
int GetCount()
{
    iTotalCount=200;        //给全局变量赋值
    return iTotalCount;
}
```

程序运行结果如图 6.13 所示。

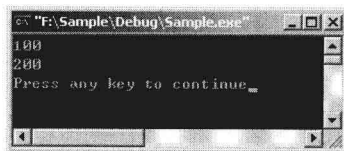


图 6.13 局部变量与全局变量

变量都有它的生命期，全局变量在程序开始时创建并分配空间，在程序结束时释放内存并销毁；局部变量是在函数调用时创建，并在栈中分配内存，在函数调用结束后销毁并释放。



## 6.5 重载函数

 视频讲解：光盘\TM\lx\6\重载函数.exe

定义同名的变量，程序会编译出错，定义同名的函数也会带来冲突的问题，但 C++中使用了名字重组的技术，通过函数的参数类型来识别函数，所谓重载函数就是指多个函数具有相同的函数标识符，但参数类型或参数个数不同。函数调用时，编译器以参数的类型及个数来区分调用哪个函数。下面的实例定义了重载函数。

**【例 6.8】** 使用重载函数。(实例位置：光盘\TM\sl\6\8)

```
#include<iostream>
using namespace std;
int Add(int x ,int y)           //定义第一个重载函数
{
    cout << "int add" << endl;   //输出信息
    return x + y;               //设置函数返回值
}
double Add(double x,double y)  //定义第二个重载函数
{
    cout << "double add" << endl; //输出信息
    return x + y;               //设置函数返回值
}
int main()
{
    int ivar = Add(5,2);         //调用第一个 Add 函数
    float fvar = Add(10.5,11.4); //调用第二个 Add 函数
    return 0;
}
```

程序运行结果如图 6.14 所示。

程序中定义了两个相同函数标识符的函数，函数名都为 Add，在 main 调用 Add 函数时实参类型不同，语句“int ivar = Add(5,2);”的实参类型是整型，语句“float fvar = Add(10.5,11.4);”的实参类型是双精度，编译器可以区分这两个函数，会正确调用相应的函数。

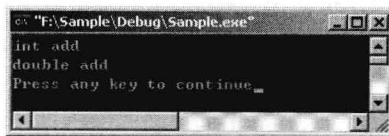


图 6.14 重载函数

在定义重载函数时，应注意函数的返回值类型不作为区分重载函数的一部分。下面的函数重载是非法的。

```
int Add(int x ,int y)           //定义一个重载函数
{
    return x + y;
}
double Add(int x,int y)         //定义一个重载函数
```

```
{
    return x + y;
}
```

## 6.6 内联函数

 视频讲解：光盘\TM\lx\6\内联函数.exe

通过 `inline` 关键字可以把函数定义为内联函数，编译器会在每个调用该函数的地方展开一个函数的副本。

在下面的程序中创建了一个 `IntegerAdd` 函数，并进行了调用。

```
#include<iostream>
using namespace std;
inline int IntegerAdd(int x,int y);
void main()
{
    int a;
    int b;
    int iresult=IntegerAdd(a,b);
}
int IntegerAdd(int x,int y)
{
    return x+y;
}
```

`IntegerAdd` 函数被定义为内联函数，其执行代码如下：

```
#include<iostream>
using namespace std;
inline int IntegerAdd(int x,int y);
void main()
{
    int a;
    int b;
    int iresult= a+b;
}
```

使用内联函数可以减少函数调用带来的开销（在程序所在文件内移动指针寻找调用函数地址带来的开销），但它只是一种解决方案，编译器可以忽略内联的声明。

应该在函数实现代码很简短或者调用该函数次数相对较少的情况下将函数定义为内联函数，一个递归函数不能在调用点完全展开，一个一千行代码的函数也不大可能在调用点展开，内联函数只能在优化程序时使用。在抽象数据类设计中，内联函数对支持信息隐藏起着主要作用。

如果某个内联函数要作为外部全局函数，即它将被多个源代码文件使用，那么就把它定义在头文件里，在每个调用该内联函数的源文件中包含该头文件，这种方法保证对每个内联函数只有一个定义，



防止在程序的生命期中引起无意的不匹配。

## 6.7 变量的存储类别

 视频讲解：光盘\TM\lx\6\变量的存储类别.exe

存储类别是变量的属性之一，C++语言中定义了4种变量的存储类别，分别是 **auto** 变量、**static** 变量、**register** 变量和 **extern** 变量。变量存储方式不同会使变量的生存期不同，生存期表示了变量存在的时间。生存期和变量作用域是从时间和空间这两个不同的角度来描述变量的特性。

静态存储变量通常是在变量定义时就分配固定的存储单元并一直保持不变，直至整个程序结束。前面讲过的全局变量即属于此类存储方式，它们存放在静态存储区中。动态存储变量是在程序执行过程中使用它时才分配存储单元，使用完毕立即将该存储单元释放。前面讲过的函数的形式参数，在函数定义时并不给形参分配存储单元，只是在函数被调用时才予以分配，调用函数完毕立即释放，此类变量存放在动态存储区中。从以上分析可知，静态存储变量是一直存在的，而动态存储变量则时而存在时而消失。

### 6.7.1 auto 变量

这种存储类型是C++语言程序中默认的存储类型。函数内未加存储类型说明的变量均视为自动变量，也就是说自动变量可省去关键字 **auto**。例如：

```
{  
int i,j,k;  
...  
}
```

等价于：

```
{  
auto int i,j,k;  
...  
}
```

自动变量具有以下特点：

(1) 自动变量的作用域仅限于定义该变量的个体内。在函数中定义的自动变量，只在该函数内有效；在复合语句中定义的自动变量，只在该复合语句中有效。例如：

```
int Show()  
{  
    auto int x,y;  
    if(true)  
    {
```

```

    auto char ch;
    cout << ch << endl; //正确
    cout << x << endl; //正确
}
cout << ch << endl;      //错误
cout << x << endl;      //正确
}

```

(2) 自动变量属于动态存储方式, 变量分配的内存是在栈中, 当函数调用结束后, 自动变量的值会被释放。同样, 在复合语句中定义的自动变量, 在退出复合语句后也不能再使用, 否则将引起错误。

(3) 由于自动变量的作用域和生存期都局限于定义它的个体内 (函数或复合语句内), 因此不同的个体中允许使用同名的变量而不会混淆。即使在函数内定义的自动变量也可与该函数内部的复合语句中定义的自动变量同名。

**【例 6.9】** 输出不同生存期的变量值。(实例位置: 光盘\TM\sl\6.9)

```

#include<iostream>
using namespace std;
void main()
{
    auto int i,j,k;
    cout <<"input the number:" << endl;
    cin >> i >> j;
    k=i+j;
    if(i!=0 && j!=0)
    {
        auto int k;
        k=i-j;
        cout << "k :." << k << endl;      //输出变量 k 的值
    }
    cout << "k :." << k << endl;      //输出变量 k 的值
}

```

程序运行结果如图 6.15 所示。

程序两次输出自动变量 k。第一次输出的是 i-j 的值, 第二次输出的是 i+j 的值。虽然变量名都为 k, 但其实是两个不同的变量。

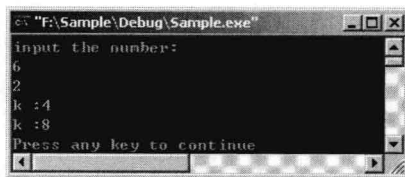


图 6.15 输出不同生存期的变量值

## 6.7.2 static 变量

在声明变量时加关键字 `static`, 可以将变量声明为静态变量。静态局部变量的值在函数调用结束后不消失, 静态全局变量只能在本源文件中使用。例如下面的代码声明变量为静态变量:

```

static int a,b;
static float x,y;
static int a[3]={0,1,2};

```

静态变量属于静态存储方式，它具有以下特点：

(1) 静态变量在函数内定义，在程序退出时释放，在程序整个运行期间都不释放，也就是说它的生存期为整个源程序。

(2) 静态变量的作用域与自动变量相同，在函数内定义就在函数内使用，尽管该变量还继续存在，但不能使用它，如再次调用定义它的函数时，它又可继续使用。

(3) 编译器会为静态局部变量赋予 0 值。

下面通过实例介绍 static 变量的用法。

**【例 6.10】** 使用 static 变量实现累加。(实例位置：光盘\TM\sl\6\10)

```
#include<iostream>
using namespace std;
int add(int x)
{
    static int n=0;
    n=n+x;
    return n;
}
void main()
{
    int i,j,sum;
    cout << "input the number:" << endl;
    cin >> i;
    cout << "the result is:" << endl;
    for(j=1;j<=i;j++)
    {
        sum=add(j);
        cout << j << ":" << sum << endl;
    }
}
```

程序运行结果如图 6.16 所示。

程序中 n 是静态局部变量，每次调用函数 add 时，静态局部变量 n 都保存了前次被调用后留下的值。所以当输入循环次数 3 时，变量 sum 累加的结果是 6，而不是 3。

如果去除 static 关键字后，程序运行结果如图 6.17 所示。

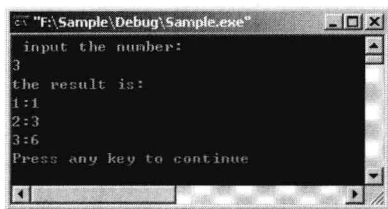


图 6.16 使用 static 变量实现累加

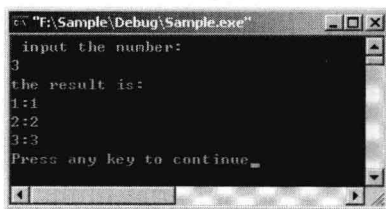


图 6.17 运行结果

当输入循环次数 3 时，变量 sum 累加的结果是 3。变量 n 不再使用静态存储区空间，每次调用后

变量 `n` 的值都被释放，再次调用时 `n` 的值为初始值 0。

### 6.7.3 register 变量

通常变量的值存放在内存中，当对一个变量频繁读写时，需要反复访问内存储器，此时将花费大量的存取时间。为了提高效率，C++语言可以将变量声明为寄存器变量，这种变量将局部变量的值存放在 CPU 中的寄存器中，使用时不需要访问内存，而直接从寄存器中读写。寄存器变量的声明符是 `register`。

对寄存器变量的说明：

- (1) 寄存器变量属于动态存储方式。凡需要采用静态存储方式的变量不能定义为寄存器变量。
- (2) 编译程序会自动决定哪个变量使用寄存器存储。`register` 可以起到程序优化的作用。

### 6.7.4 extern 变量

在一个源文件中定义的变量和函数只能被本源文件中的函数调用，一个 C++ 程序中会有许多源文件，那么如何使用非本源文件中的全局变量呢？C++ 提供了 `extern` 关键字来解决这个问题。在使用其他源文件中的全局变量时，只需要在本源文件中使用 `extern` 关键字来声明这个变量即可。例如，在 `Sample1.cpp` 源文件中定义全局变量 `a`、`b`、`c`，代码如下：

```
int a,b;    //外部变量定义
char c;     //外部变量定义
void main()
{
    cout << a << endl;
    cout << b << endl;
    cout << c << endl;
}
```

在 `Sample2.cpp` 源文件中要使用 `Sample1.cpp` 源文件中的全局变量 `a`、`b`、`c`，代码如下：

```
extern int a,b; //外部变量说明
extern char c;  //外部变量说明
func(int x,y)
{
    cout << a << endl;
    cout << b << endl;
    cout << c << endl;
}
```

在 `Sample2.cpp` 源文件中，编译系统不再为全局变量 `a`、`b`、`c` 分配内存空间，而是改变全局变量 `a`、`b`、`c` 的值，在 `Sample1.cpp` 源文件中输出值也会发生变化。

## 6.8 小 结

本章主要介绍函数的使用，使用函数时要了解函数的返回值、函数的参数以及函数的调用方式。变量的作用域和函数有关，函数的递归调用可以帮助开发人员设计出思路明了的程序，内联函数可以提高程序的运算效率，函数重载则解决了代码复用中函数名冲突的问题。

## 6.9 实践与练习

1. 开发一个程序，要求设计一个函数进行减法运算。（答案位置：光盘\TM\sl\6\11）
2. 定义一个标识符为 `Max` 的函数，其函数功能是判断两个整数的大小，并将较大的整数显示出来。（答案位置：光盘\TM\sl\6\12）





# 第 7 章

---

## 数组、指针和引用

(  视频讲解：1 小时 27 分钟 )

数组是有序数据的集合，可以减少同种类型变量的声明，指针则是可以操作内存数据的变量，引用则是变量的别名。数组的首地址可以看作是指针，而通过指针也可以操作数组，指针和引用在函数的参数传递时可以相互替代。指针是一个双刃剑，能够带来效率的提升，也会给程序带来意想不到的灾难。

通过阅读本章，您可以：

- » 掌握指针和数组的应用
- » 掌握指针与函数的应用
- » 掌握引用的应用
- » 掌握指针数组的应用

## 7.1 一维数组

 视频讲解：光盘\TM\lx\7\一维数组.exe

### 7.1.1 一维数组的声明

在程序设计中，将同一数据类型的数据按一定形式有序地组织起来，这些有序数据的集合就称为数组。一个数组有一个统一的数组名，可以通过数组名和下标来唯一确定数组中的元素。

一维数组的声明形式如下：

**数据类型 数组名[常量表达式]**

例如：

```
int a[10];           //声明一个整型数组，包含 10 个元素
char name[128];      //声明一个字符数组，包含 128 个元素
float price[20];     //声明一个浮点数组，包含 20 个元素
```

使用数组的说明：

- (1) 数组名的命名规则和变量名相同。
- (2) 数组名后面的括号是方括号，方括号内是常量表达式。
- (3) 常量表达式表示元素的个数，即数组的长度。
- (4) 定义数组的常量表达式不能是变量，因为数组的大小不能动态定义。例如：

```
int a[]; //不合法
```

### 7.1.2 一维数组的引用

一维数组引用的一般形式如下：

**数组名[下标]**

例如：

```
int a[10]; //声明数组
```

a[0]、a[1]、a[2]、a[3]、a[4]、a[5]、a[6]、a[7]、a[8]、a[9]，是对数组 a 中 10 个元素的引用。

一维数组引用的说明：

- (1) 数组元素的下标起始值为 0 而不是 1。
- (2) a[10]是不存在的数组元素，引用 a[10]非法。



`a[10]`属于下标越界，下标越界容易造成程序瘫痪。

### 7.1.3 一维数组的初始化

数组元素初始化的方式有两种，一种是对单个元素逐一赋值，另一种是使用聚合方式赋值。

(1) 单一数组元素赋值

`a[0]=0`就是对单一数组元素赋值，也可以通过变量控制下标的方式进行赋值。例如：

```
#include<iostream>
using namespace std;
void main()
{
    char a[3];
    a[0]='a';
    a[2]='c';
    int i=0;
    cout << a[i] << endl;
}
```

程序运行结果如图 7.1 所示。

(2) 聚合方式赋值

数组不仅可以逐一对数组元素赋值，还可以通过大括号进行多个元素的赋值。例如：

```
int a[12]={1,2,3,4,5,6,7,8,9,10,11,12};
```

或

```
int a[ ]={1,2,3,4,5,6,7,8,9,10,11,12}; //编译器能够获得数组元素个数
```

或

```
int a[12]={1,2,3,4,5,6,7}; //前 7 个元素被赋值，后面 5 个元素的值为 0
```

下面通过实例来看一下如何为一维数组的数组元素赋值。

**【例 7.1】** 一维数组赋值。(实例位置：光盘\TM\sl\7\1)

```
#include<iostream>
using namespace std;
void main()
{
    int i,a[10];
    //利用循环，分别为 10 个元素赋值
    for(i=0;i<10;i++)
        a[i]=i;
    //将数组中的 10 个元素输出到显示设备
```

```

for(i=0;i<10;i++)
    cout << a[i] << endl;
}

```

程序运行结果如图 7.2 所示。

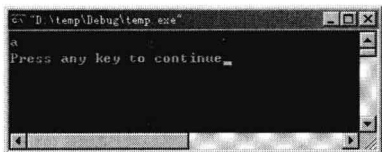


图 7.1 单一数组元素赋值



图 7.2 一维数组赋值

程序实现通过 for 循环将 int a[10]定义的数组中的每个元素赋值，然后再循环通过 cout 函数将数组中的元素值输出到显示设备。

## 7.2 二维数组

视频讲解：光盘\TM\lx\7\二维数组.exe

### 7.2.1 二维数组的声明

二维数组声明的一般形式为：

**数据类型 数组名[常量表达式 1][常量表达式 2]**

例如：

```

int a[3][4];           //声明具有 3 行 4 列元素的整型数组
float myArray[4][5];   //声明具有 4 行 5 列元素的浮点型数组

```

一维数组描述的是一个线性序列，二维数组描述的则是一个矩阵。常量表达式 1 代表行的数量，常量表达式 2 代表列的数量。

二维数组可以看作是一种特殊的一维数组，如图 7.3 所示，虚线左侧为 3 个一维数组的首元素，二维数组是由 A[0]、A[1]、A[2]这 3 个一维数组组成，每个一维数组都包含 4 个元素。

使用数组的说明：

- (1) 数组名的命名规则和变量名相同。
- (2) 二维数组有两个下标，所以要有两个中括号。例如：

A[0][0]	A[0][1]	A[0][2]	A[0][3]
A[1][0]	A[1][1]	A[1][2]	A[1][3]
A[2][0]	A[2][1]	A[2][2]	A[2][3]

图 7.3 二维数组

```
int a[3,4] //不合法
int a[3:4] //不合法
```

(3) 下标运算符中的整数表达式代表数组每一个维的长度，它们必须是正整数，其乘积确定了整个数组的长度。例如：

```
int a[3][4]
```

其长度就是  $3 \times 4 = 12$ 。

(4) 定义数组的常量表达式不能是变量，因为数组的大小不能动态定义。例如：

```
int a[i][j]; //不合法
```

## 7.2.2 二维数组元素的引用

二维数组元素的引用形式为：

```
数组名[下标][下标]
```

二维数组元素的引用和一维数组基本相同。例如：

```
a[2-1][2*2-1] //合法
a[2,3],a[2-1,2*2-1] //不合法
```

## 7.2.3 二维数组的初始化

二维数组元素初始化的方式和一维数组相同，也分为单个元素逐一的赋值和使用聚合方式赋值。例如：

```
myArray[0][1]=12; //单个元素初始化
int a[3][4]={1,2,3,4,5,6,7,8,9,10,11,12}; //使用聚合方式赋值
```

使用聚合方式给数组赋值等同于分别对数组中的每个元素进行赋值。例如：

```
int a[3][4]={1,2,3,4,5,6,7,8,9,10,11,12};
```

等同于执行如下语句：

```
a[0][0]=1;a[0][1]=2;a[0][2]=3;a[0][3]=4;
a[1][0]=5;a[1][1]=6;a[1][2]=7;a[1][3]=8;
a[2][0]=9;a[2][1]=10;a[2][2]=11;a[2][3]=12;
```

二维数组中元素排列的顺序是按行存放，即在内存中先顺序存放第 1 行的元素，再存放第 2 行的元素。例如 “`int a[3][4]={1,2,3,4,5,6,7,8,9,10,11,12};`” 的赋值顺序是：

先给第 1 行元素赋值：`a[0][0]->a[0][1]->a[0][2]->a[0][3]`。

再给第2行元素赋值：a[1][0]->a[1][1]->a[1][2]->a[1][3]。

最后给第3行元素赋值：a[2][0]->a[2][1]->a[2][2]->a[2][3]。

数组元素的位置以及对应数值如图7.4所示。

A[0][0]	A[0][1]	A[0][2]	A[0][3]	1	2	3	4
A[1][0]	A[1][1]	A[1][2]	A[1][3]	5	6	7	8
A[2][0]	A[2][1]	A[2][2]	A[2][3]	9	10	11	12

数组位置

数值位置

图7.4 数组位置对应的数值

使用聚合方式赋值，还可以按行进行赋值，例如：

```
int a[3][4]={1,2,3,4},{5,6,7,8},{9,10,11,12};
```

二维数组可以只对前几个元素赋值。例如：

```
a[3][4]={1,2,3,4}; //相当于给第1行赋值，其余数组元素全为0
```

数组元素是左值，可以出现在表达式中，也可以对数组元素进行计算。例如：

```
b[1][2]=a[2][3]/2;
```

下面通过实例来熟悉二维数组的操作，实例将实现将二维数组中行数据和列数据相互互换的功能。

**【例7.2】** 将二维数组行列对换。（实例位置：光盘\TM\sl\7\2）

```
int fun(int array[3][3])
{
    int i,j,t;
    for(i=0;i<3;i++)
        for(j=0;j<i;j++)
        {
            t=array[i][j];
            array[i][j]=array[j][i];
            array[j][i]=t;
        }
    return 0;
}

void main()
{
    int i,j;
    int array[3][3]={1,2,3},{4,5,6},{7,8,9};
    cout << "Converted Front" << endl;
    for(i=0;i<3;i++)
    {
```



```

        for(j=0;j<3;j++)
            cout << setw(7) << array[i][j] ;
        cout<< endl;
    }
    fun(array);
    cout << "Converted result" <<endl;
    for(i=0;i<3;i++)
    {
        for(j=0;j<3;j++)
            cout << setw(7) << array[i][j] ;
        cout<< endl;
    }
}

```

程序运行结果如图 7.5 所示。

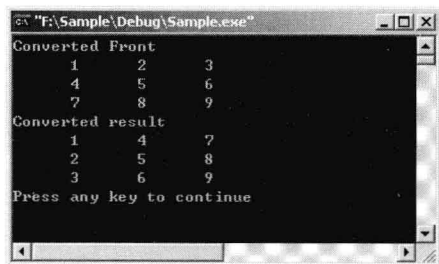


图 7.5 将二维数组行列对换

程序首先输出二维数组 `array` 中的元素,然后调用自定义函数 `fun` 将数组中的行元素转换为列元素,最后输出转换后的结果。

## 7.3 字符数组

### 视频讲解: 光盘\TM\lx\7\字符数组.exe

用来存放字符数据的数组是字符数组,字符数组中的一个元素存放一个字符。字符数组具有数组的共同属性。由于字符串应用广泛,C和C++专门为它提供了许多方便的用法和函数。

#### 1. 声明一个字符串数组

```
char pWord[11];
```

#### 2. 字符串数组赋值方式

可以对数组元素逐一赋值。例如:

```

pWord[0]='H' pWord[1]='E' pWord[2]='L' pWord[3]='L'
pWord[4]='O' pWord[5]=' ' pWord[6]='W' pWord[7]='O'
pWord[8]='R' pWord[9]='L' pWord[10]='D'

```

也可以使用聚合方式赋值。例如：

```
char pWord[]={'H','E','L','L','O',' ','W','O','R','L','D'};
```

如果大括号中提供的初值个数大于数组长度，则按语法错误处理。如果初值个数小于数组长度，则只将这些字符赋给数组中前面那些元素，其余元素自动定义为空字符。如果提供的初值个数与预定的数组长度相同，在定义时可以省略数组长度，系统会自动根据初值个数确定数组长度。

### 3. 字符数组的一些说明

聚合方式只能在数组声明时使用。例如：

```
char pWord[5];
pWord={'H','E','L','L','O'}; //错误
```

字符数组不能给字符数组赋值。例如：

```
char a[5]= {'H','E','L','L','O'};
char b[5];
a=b; //错误
a[0]=b[0]; //正确
```

### 4. 字符串和字符串结束标志

字符数组常作字符串使用，作为字符串要有字符串结束符“\0”。

可以使用字符串为字符数组赋值。例如：

```
char a[]="HELLO WORLD";
```

等同于：

```
char a[]="HELLO WORLD\0";
```

字符串结束符“\0”主要告知字符串处理函数字符串已经结束了，不需要再输出了。

下面通过实例来看一下使用字符串结束符“\0”和不使用字符串结束符“\0”的区别。

**【例 7.3】** 使用字符串结束符“\0”防止出现非法字符。（实例位置：光盘\TM\sl\7\3）

未使用字符串结束符“\0”的程序，代码如下：

```
#include<iostream>
using namespace std;
void main()
{
    int i;
    char array[12];
    array[0]='a';
    array[1]='b';
    printf("%s\n",array);
}
```

程序运行结果如图 7.6 所示。

使用字符串结束符“\0”的程序代码如下：

```
#include<iostream>
using namespace std;
void main()
{
    int i;
    char array[12];
    array[0]='a';
    array[1]='b';
    array[2]='\0';
    printf("%s\n",array);
}
```

程序运行结果如图 7.7 所示。



图 7.6 未使用字符串结束符“\0”

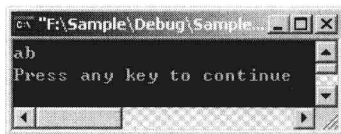


图 7.7 使用字符串结束符“\0”

printf 函数使用%s 格式可以输出字符串，如果字符串中没有结束符，函数会将整个字符数组输出。array 字符数组中只有前两个字符初始化了，所以未使用字符串结束符“\0”的程序会出现乱码。

下面通过实例来熟悉在程序中对字符数组的操作。

**【例 7.4】** 输出字符数组中的内容。（实例位置：光盘\TM\sl\7\4）

```
#include<iostream>
using namespace std;
void main()
{
    int i;
    char array[12]={'H','E','L','L','O',' ','W','O','R','L','D'};
    for(i=0;i<12;i++)
        cout<<array[i];
    cout << endl;
}
```

程序运行结果如图 7.8 所示。

## 5. 字符串处理函数

### ☒ strcat 函数

字符串连接函数 strcat 的格式如下：

strcat(字符数组 1,字符数组 2)

其功能是将字符数组 2 中的字符串连接到字符数组 1 中字符串的后面，并删去字符串 1 后的串结

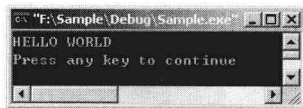


图 7.8 字符数组中的内容

束标志“\0”。

下面通过实例使用 `strcat` 函数将两个字符串连接在一起。

**【例 7.5】** 连接字符串。（实例位置：光盘\TM\sl\7\5）

```
#include<iostream>
#include<string>
using namespace std;
void main()
{
    char str1[30],str2[20];
    cout<<"please input string1:"<< endl;
    gets(str1);
    cout<<"please input string2:"<<endl;
    gets(str2);
    strcat(str1,str2);
    cout <<"Now the string1 is:"<<endl;
    puts(str1);
}
```

程序运行结果如图 7.9 所示。

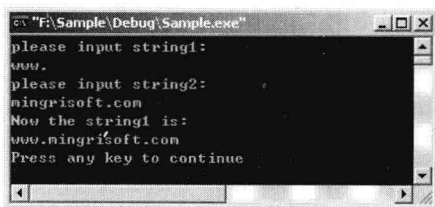


图 7.9 连接字符串

### 说明

在使用 `strcat` 函数时要注意，字符数组 1 的长度要足够大，否则不能装下连接后的字符串。

连接两个字符串时，可以不使用 `strcat` 函数。

下面通过实例来实现不使用 `strcat` 函数连接两个字符串的功能。

**【例 7.6】** 不使用 `strcat` 函数连接两个字符串。（实例位置：光盘\TM\sl\7\6）

```
#include<iostream>
using namespace std;
void main()
{
    int i=0,j=0;                //定义整型变量
    char a[100],b[50];          //定义字符型数组
    cout <<"please input string1:" << endl;
    cin >> a;                    //输入字符串存于数组 a 中
    cout << "please input string2:" << endl;
    cin >> b;                    //输入字符串存于数组 b 中
    while(a[i]!='\0')            //逐个遍历数组 a 中的元素，直到遇到字符串结束标志
```

```

    i++;
    while(b[j]!='\0')           //逐个遍历数组 b 中的元素，直到遇到字符串结束标志
        a[i++]=b[j++];         //将数组 b 中的元素存入数组 a 中，并从数组 a 原来存放“\0”位置开始覆盖“\0”
        a[j]='\0';             //在合并后的两个字符串的最后加“\0”
    cout << a << endl;         //输出合并后的字符串
}

```

本例的关键是在将后一个字符串连接到前一个字符串时，要先判断前一个字符串的结束标志在什么位置，只有找到了前一个字符串的结束标志才能连接后一个字符串。

#### ☑ strcpy 函数

字符串复制函数 strcpy 的格式如下：

strcpy(字符数组 1, 字符数组 2)

其功能是将把字符数组 2 中的字符串复制到字符数组 1 中。字符串结束标志“\0”也一同复制。

#### 说明

- ① 要求字符数组 1 应有足够的长度，否则不能全部装入所复制的字符串。
- ② 字符数组 1 必须写成数组名形式，而字符数组 2 可以是字符数组名，也可以是一个字符串常量，这时相当于把一个字符串赋予一个字符数组。

为了使读者更好地了解 strcpy 函数，下面通过实例使用 strcpy 函数来实现字符串复制的功能。

**【例 7.7】** 字符串复制。(实例位置：光盘\TM\sl\7\7)

```

#include<iostream>
#include<string>
using namespace std;
void main()
{
    char str1[30],str2[20];
    cout<<"please input string1:"<< endl;
    gets(str1);
    cout<<"please input string2:"<<endl;
    gets(str2);
    strcpy(str1,str2);
    cout<<"Now the string1 is:\n"<<endl;
    puts(str1);
}

```

程序运行结果如图 7.10 所示。

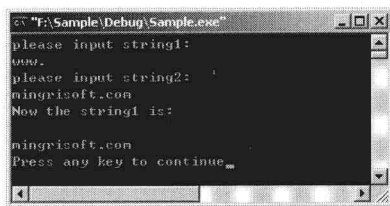


图 7.10 字符串复制



**说明**

strcpy 函数实质上是字符数组 2 中的字符串覆盖字符数组 1 中的内容, 而 strcat 函数则不存在覆盖等问题, 只是单纯地将字符数组 2 中的字符串连接到字符数组 1 中的字符串后面。

☒ strcmp 函数

字符串比较函数 strcmp 的格式如下:

```
strcmp(字符数组 1, 字符数组 2)
```

其功能是按照 ASCII 码顺序比较两个数组中的字符串, 并由函数返回值返回比较结果。如果字符串 1=字符串 2, 返回值为 0; 如果字符串 1>字符串 2, 返回值为一正数; 如果字符串 1<字符串 2, 返回值为负数。

该函数可用于比较两个字符串常量, 或比较数组和字符串常量。例如:

```
strcmp(str1, str2);
```

该语句是两个数组进行比较。

```
strcmp(str1, "hello");
```

该语句是一个数组与一个字符串进行比较。

```
strcmp("hello", "how");
```

该语句是两个字符串进行比较。

**说明**

进行比较时若出现不同的字符, 则以第一个不同的字符的比较结果作为整个比较的结果。

下面通过实例来看一下如何使用 strcmp 函数对字符串进行比较。

**【例 7.8】** 字符串比较。(实例位置: 光盘\TM\sl\7\8)

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

#include<string>
void main()
{
    char str1[30], str2[20];
    int i=0;
    cout<<"please input string1:"<< endl;
    gets(str1);
    cout<<"please input string2:"<< endl;
    gets(str2);
    i=strcmp(str1, str2);
```



```

if(i>0)
cout <<"str1>str2"<<endl;
else
if(i<0)
cout <<"str1<str2"<<endl;
else
cout <<"str1=str2"<<endl;
}

```

程序运行结果如图 7.11 所示。

#### ☑ strlen 函数

测字符串长度函数 strlen 的格式如下：

strlen(字符数组名)

其功能是测字符串的实际长度（不含字符串结束标志“\0”），函数返回值为字符串的实际长度。下面通过实例调用 strlen 函数来实现获取字符串长度的功能。

**【例 7.9】** 获取字符串长度。（实例位置：光盘\TM\sl\7\9）

```

#include<iostream>
#include<string>
using namespace std;
void main()
{
    char str1[30],str2[20];
    int len1,len2;
    cout<<"please input string1:"<< endl;
    gets(str1);
    cout<<"please input string2:"<<endl;
    gets(str2);
    len1=strlen(str1);
    len2=strlen(str2);
    cout <<"the length of string1 is:"<< len1 <<endl;
    cout <<"the length of string2 is:"<< len2 <<endl;
}

```

程序运行结果如图 7.12 所示。

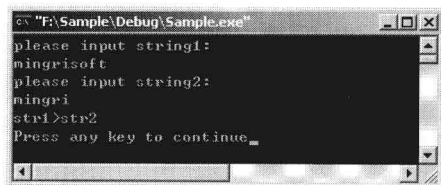


图 7.11 字符串比较

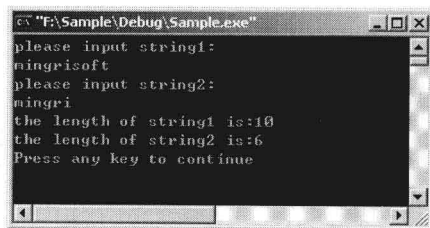



图 7.12 获取字符串长度

## 7.4 指 针

 视频讲解：光盘\TM\lx\7\指针.exe

### 7.4.1 变量与指针

系统的内存就像是带有编号的小房间，如果想使用内存就需要得到房间号。如图 7.13 所示，定义一个整型变量 *i*，它需要 4 个字节，所以编译器为变量 *i* 分配了编号从 4001 到 4004 的房间，每个房间代表一个字节。

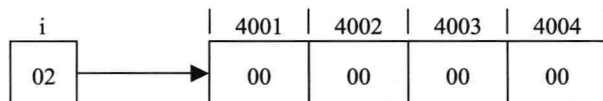


图 7.13 整型变量 *i*

各个变量连续地存储在系统的内存中，如图 7.14 所示，两个整型变量 *i* 和 *j* 存储在内存中。

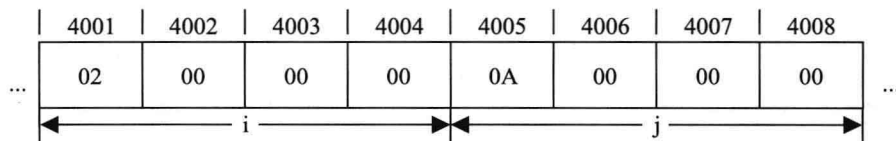


图 7.14 整型变量 *i* 和 *j*

在程序代码中通过变量名来对内存单元进行存取操作，但是代码经过编译后已经将变量名转换为该变量在内存的存放地址，对变量值的存取都是通过地址进行的。例如语句“*i*+*j*;”的执行过程是根据变量名与地址的对应关系，找到变量 *i* 的地址 4001，然后从 4001 开始读取 4 个字节数据放到 CPU 的寄存器中，再找到变量 *j* 的地址 4005，从 4005 开始读取 4 个字节的数据放到 CPU 的另一个寄存器中，通过 CPU 的加法中断计算出结果。

在低级语言的汇编语言中都是直接通过地址来访问内存单元，而在高级语言中才使用变量名访问内存单元，C 语言作为高级语言却提供了通过地址来访问内存单元的方法，C++语法也继承了这一特性。

由于通过地址能访问指定的内存存储单元，可以说地址“指向”该内存单元，例如房间号 4001 指向系统内存中的一个字节。地址可以形象地称为指针，意思是通过指针能找到内存单元。一个变量的地址称为该变量的指针。如果有一个变量专门用来存放另一个变量的地址，它就是指针对量。在 C++语言中有专门用来存放内存单元地址的变量类型，就是指针对类型。

指针是一种数据类型，通常所说的指针就是指针对量，它是一个专门用来存放地址的变量，而变量的指针主要指变量在内存中的地址。变量的地址在编写代码时无法获取，只有在程序运行时才可以得到。

#### 1. 指针的声明

声明指针的一般形式如下：

数据类型标识符 \*指针变量名

例如:

```
int *p_iPoint; //声明一个整型指针
float *a,*b    //声明两个浮点型指针
```

## 2. 指针的赋值

指针可以在初始化时赋值,也可以在后期赋值。

☒ 在初始化时赋值

```
int i=100;
int *p_iPoint=&i;
```

☒ 在后期赋值

```
int i=100;
p_iPoint =&i;
```

### 说明

通过变量名访问一个变量是直接的,而通过指针访问一个变量是间接的。

## 3. 关于指针使用的说明

(1) 指针变量名是 `p`,而不是 `*p`。

`p=&i` 的意思是取变量 `i` 的地址赋给指针变量 `p`。

下面的实例可以获取变量的地址,并将获取的地址值输出。

**【例 7.10】** 输出变量的地址值。(实例位置:光盘\TM\sl\7\10)

```
#include<iostream>
using namespace std;
void main()
{
    int a=100;
    int *p=&a;
    printf("%d\n",p);    //获取地址值
}
```

程序运行结果如图 7.15 所示。

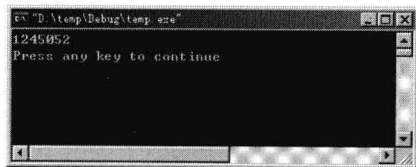


图 7.15 输出变量的地址值

实例可以通过 `printf` 函数直接将地址值输出。由于变量是由系统分配空间，所以变量的地址不是固定不变的。



### 注意

在定义一个指针之后，一般要使指针有明确的指向。与常规的变量未赋值相同，没有明确指向的指针不会引起编译器出错，但是对于指针则可能导致无法预料的或者隐藏的灾难性后果，所以指针一定要赋值。

(2) 指针变量不可以直接赋值。例如：

```
int a=100;
int *p;
p=100;
```

编译不能通过，有“error C2440: '=' : cannot convert from 'const int' to 'int \*'”错误提示。

如果强行赋值，使用指针运算符\*提取指针所指变量时会出错。例如：

```
int a=100;
int *p;
p=(int*)100;           //通过强制转换将 100 赋值给指针变量
printf("%d",p);         //输出地址，能够输出地址
printf("%d",*p);        //输出指针指向的值，出错语句
```

(3) 不能将\*p当变量使用。例如：

```
int a=100;
int *p;
*p=100;                //指针没有获得地址
printf("%d",p);         //输出地址，出错语句
printf("%d",*p);        //输出指针指向的值，出错语句
```

上面代码可以编译通过，但运行时会弹出错误提示对话框，如图 7.16 所示。

下面的实例通过指针来实现比较数据大小的功能。

**【例 7.11】** 使用指针比较两个数的大小。（实例位置：光盘\TM\sl\7\11）

```
#include<iostream>
using namespace std;
void main()
{
    int *p1,*p2;
    int *p;           //临时指针
    int a,b;
    cout << "input a: " << endl;
    cin >> a;
    cout << "input b: " << endl;
    cin >> b;
    p1=&a;p2=&b;
    if(a<b)
    {
```

```

    p=p1;
    p1=p2;
    p2=p;
}
cout << "a=" << a;
cout << " ";
cout << "b=" << b;
cout << endl;
cout << "max=" << *p1 << ",min=" << *p2 << endl;
}

```

程序运行结果如图 7.17 所示。

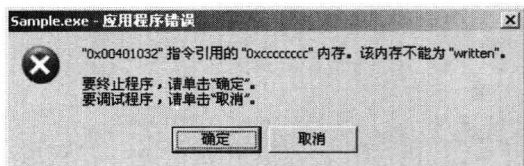


图 7.16 错误提示

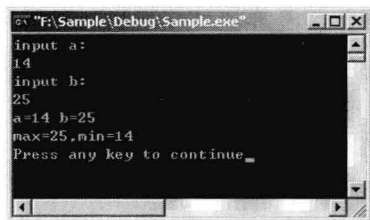


图 7.17 使用指针比较两个数大小

## 7.4.2 指针运算符和取地址运算符

### 1. 指针运算符和取地址运算符简介

\*和&是两个运算符，\*是指针运算符，&是取值运算符。

如图 7.18 所示，变量 i 的值为 100，存储在内存地址为 4009 的地方，取地址运算符&使指针变量 p 得到地址 4009。

如图 7.19 所示，指针变量存储的是地址编号 4009，指针通过指针运算符可以得到地址 4009 所对应的数值。

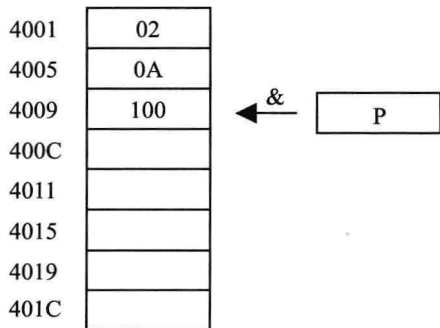


图 7.18 取地址

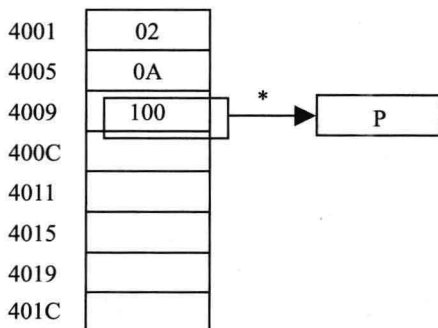


图 7.19 指针运算

下面的实例通过指针实现输出指针对应的数值的功能。

**【例 7.12】** 输出指针对应的数值。（实例位置：光盘\TM\sl\7\12）



```
#include<iostream>
using namespace std;
void main()
{
    int a=100;
    int *p=&a;
    cout << " a=" << a << endl;
    cout << "*p=" << *p << endl;
}
```

## 2. 指针运算符和取地址运算符的说明

声明并初始化指针变量时同时用到了\*和&这两个运算符。例如：

```
int *p=&a;
```

该语句等同于如下语句：

```
int *(p=&a);
```

如果写成 “\*p=&a;” 程序会报错。

“&\*p” 中的 p 只能是指针变量，如果将\*放在变量名前，编译时会有逻辑错误。例如：

```
#include<iostream>
using namespace std;
void main()
{
    int a=100;
    int *p;
    printf("%d",&*a);
}
```

编译程序会出现 “error C2100: illegal indirection” 的错误提示。

## 3. &\*p 和 \*&a 的区别

&和\*的运算符优先级别相同，按自右而左的方向结合，因此&\*p 是先进行\*运算，\*p 相当于变量 a；再进行&运算，&\*p 就相当于取变量 a 的地址。\*&a 是先计算&运算符，&a 就是取变量 a 的地址；然后进行\*运算，\*&a 就相当于取变量 a 所在地址的值，实际就是变量 a。

### 7.4.3 指针运算

指针变量存储的是地址值，对指针作运算就等于对地址作运算。下面通过实例来使读者了解指针的运算。

**【例 7.13】** 输出指针运算后地址值。（实例位置：光盘\TM\sl\7\13）

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



```

void main()
{
    int a=100;
    int *p=&a;
    printf("address:%d\n",p);
    p++;
    printf("address:%d\n",p);
    p--;
    printf("address:%d\n",p);
    p--;
    printf("address:%d\n",p);
}

```

程序运行结果如图 7.20 所示。

程序首先输出的是指向变量 a 的指针地址值 1245052，然后对指针分别进行自加运算、自减运算、自减运算，输出的结果分别是 1245056、1245052、1245048。

指针进行一次加 1 运算，其地址值并没有加 1，而是增加了 4，这和声明指针的类型有关。

p++是对指针作自加运算，相当于语句“p=p+1”，地址是按字节存放数据，但指针加 1 并不代表地址值加 1 个字节，而是加上指针数据类型所占的字节宽度。要获取字节宽度需要使用 sizeof 关键字，例如整型的字节宽度是 sizeof(int)，sizeof(int)的值为 4；双精度整型的字节宽度是 sizeof(double)，其值为 8。将实例中的 int 指针类型改为 double，看看运行结果，代码如下：

```

#include<iostream>
using namespace std;
void main()
{
    double a=100;
    double *p=&a;
    printf("address:%d\n",p);
    p++;
    printf("address:%d\n",p);
    p--;
    printf("address:%d\n",p);
    p--;
    printf("address:%d\n",p);
}

```

程序运行结果如图 7.21 所示。

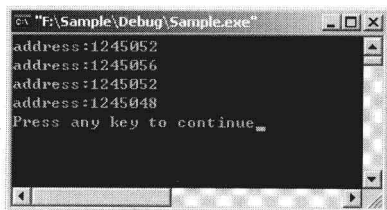


图 7.20 输出指针运算后的地址值

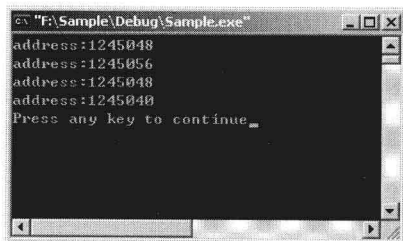


图 7.21 运行结果

**说明**

定义指针变量时必须指定一个数据类型。指针变量的数据类型用来指定该指针变量所指向数据的类型。

## 7.5 指针与数组



视频讲解：光盘\TM\lx\7\指针与数组.exe

### 7.5.1 数组的存储

数组，作为同名、同类型元素的有序集合，被顺序存放在一块连续的内存中，而且每个元素存储空间的大小相同。数组中第一个元素的存储地址就是整个数组的存储首地址，该地址放在数组名中。

对于一维数组而言，其结构是线性的，所以数组元素按下标值由小到大的顺序依次存放在一块连续的内存中。在内存中存储一维数组如图 7.22 所示。

对于二维数组而言，用矩阵方式存储元素，在内存中仍然是线性结构。

4001	a[0]
4005	a[1]
4009	a[2]
400C	a[3]
4011	a[4]
4015	a[5]
4019	a[6]
401C	a[7]

图 7.22 一维数组的存储

### 7.5.2 指针与一维数组

系统需要提供一定量连续的内存来存储数组中的各元素，内存都有地址，指针变量就是存放地址的变量，如果把数组的地址赋给指针变量，就可以通过指针变量来引用数组。引用数组元素有两种方法：下标法和指针法。

通过指针引用数组，就要先声明一个数组，再声明一个指针。例如：

```
int a[10];
int * p;
```

然后通过&运算符获取数组中元素的地址，再将地址值赋给指针变量。例如：

```
p=&a[0];
```

把 a[0] 元素的地址赋给指针变量 p，即 p 指向 a 数组的第 0 号元素，如图 7.23 所示。

下面通过实例使读者了解指针和数组间的操作，实例将实现通过指针变量获取数组中元素的功能。

**【例 7.14】** 通过指针变量获取数组中的元素。（实例位置：光盘\TM\sl\7\14）

```
#include<iostream>
using namespace std;
void main()
```

```

{
    int i,a[10];
    int *p;
    //利用循环，分别为 10 个元素赋值
    for(i=0;i<10;i++)
        a[i]=i;
    //将数组中的 10 个元素输出到显示设备
    p=&a[0];
    for(i=0;i<10;i++,p++)
        cout << *p << endl;
}

```

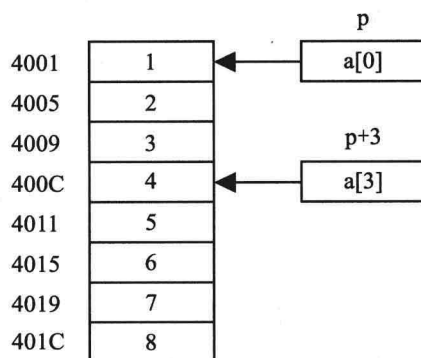


图 7.23 指针指向数组元素

如果指针变量  $p$  已指向数组中的一个元素，则  $p+1$  指向同一数组中的下一个元素。

$p+i$  和  $a+i$  是  $a[i]$  的地址。 $a$  代表首元素的地址， $a+i$  也是地址，对应数组元素  $a[i]$ 。

$*(p+i)$  或  $*(a+i)$  是  $p+i$  或  $a+i$  所指向的数组元素，即  $a[i]$ 。

程序中使用指针获取数组首元素的地址，也可以将数组名赋值给指针，然后通过指针访问数组。

实现代码如下：

```

#include<iostream>
using namespace std;
void main()
{
    int i,a[10];
    int *p;
    //利用循环，分别为 10 个元素赋值
    for(i=0;i<10;i++)
        a[i]=i;
    //将数组中的 10 个元素输出到显示设备
    p=a;
    for(i=0;i<10;i++,p++)
        cout << *p << endl;
}

```

程序中还可以使用数组地址来进行计算，用  $a+i$  表示数组  $a$  中的第  $i$  个元素的地址，然后通过指针

运算符就可以获得数组元素的值。实现代码如下：

```
#include<iostream>
using namespace std;
void main()
{
    int i,a[10];
    int *p;
    //利用循环，分别为 10 个元素赋值
    for(i=0;i<10;i++)
        a[i]=i;
    //将数组中的 10 个元素输出到显示设备
    p=a;
    for(i=0;i<10;i++)
        cout << *(a+i) << endl;
}
```

### 7.5.3 指针与二维数组

可以将一维数组的地址赋给指针变量，同样也可以将二维数组的地址赋给指针变量，因为一维数组的内存地址是连续的，二维数组的内存地址也是连续的，所以可以将二维数组看作是一维数组。二维数组中各元素的地址如图 7.24 所示。

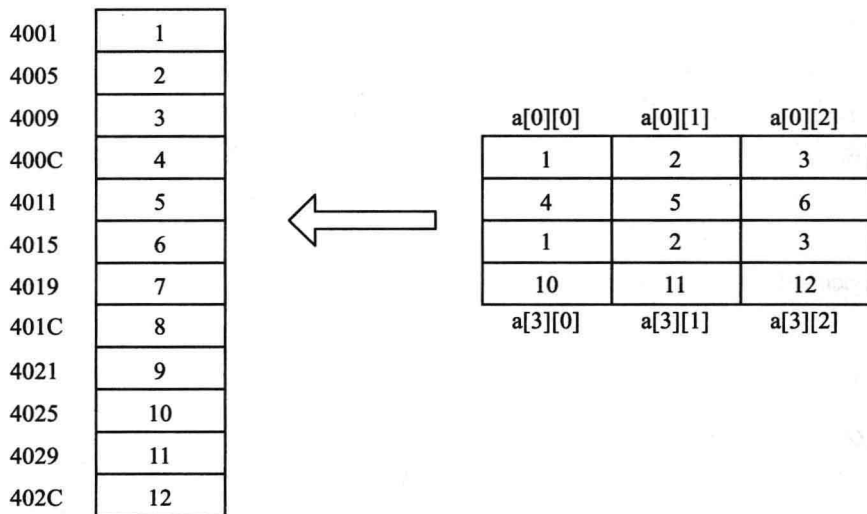


图 7.24 二维数组中各元素的地址

因为多维数组可以看作是一维数组，本例实现将多维数组转换成一维数组的功能。

**【例 7.15】** 将多维数组转换成一维数组。（实例位置：光盘\TM\sl\7\15）

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



```

void main()
{
    int array1[3][4]={{1,2,3,4},
        {5,6,7,8},
        {9,10,11,12}};
    int array2[12]={0};
    int row,col,i;
    cout << "array old" << endl;
    for(row=0;row<3;row++)
    {
        for(col=0;col<4;col++)
        {
            cout << array1[row][col];
        }
        cout << endl;
    }
    cout << "array new" << endl;
    for(row=0;row<3;row++)
    {
        for(col=0;col<4;col++)
        {
            i=col+row*4;
            array2[i]=array1[row][col];
        }
    }
    for(i=0;i<12;i++)
        cout << array2[i] << endl;
}

```

程序运行结果如图 7.25 所示。

使用指针引用二维数组和引用一维数组相同，首先声明一个二维数组和一个指针变量。例如：

```

int a[4][3];
int * p;

```

a[0]是二维数组中第一个元素的地址，可以将该地址值直接赋给指针变量。例如：

```

p=a[0];

```

此时使用指针 p 就可以引用二维数组中的元素了。

为了更好地操作二维数组，下面通过实例来实现使用指针变量遍历二维数组的功能。

**【例 7.16】** 使用指针变量遍历二维数组。（实例位置：光盘\TM\sl\7\16）

```

#include<iostream>
#include<iomanip>
using namespace std;
void main()
{
    int a[4][3]={1,2,3,4,5,6,7,8,9,10,11,12};
    int *p;

```

```

p=a[0];
for(int i=0;i<sizeof(a)/sizeof(int);i++)
{
    cout << "address:";
    cout << a[i];
    cout << " is ";
    cout << *p++ << endl;
}
}

```

程序运行结果如图 7.26 所示。

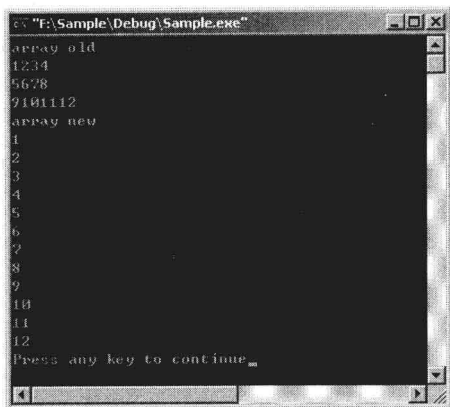


图 7.25 将多维数组转换成一维数组

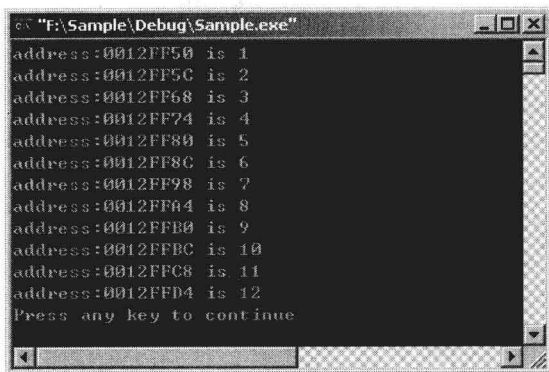


图 7.26 使用指针变量遍历二维数组

程序中通过\*p对二维数组中的所有元素都进行了引用，如果想对二维数组中某一行中的某一列元素进行引用，就需要将二维数组不同行的首元素地址赋给指针变量。如图 7.27 所示，可以将 4 个行首元素地址赋给变量 p。其中 a 代表二维数组的地址，通过指针运算符可以获取数组中的元素。

(1) a+n 表示第 n 行的首地址。

(2) &a[0][0]既可以看作数组 0 行 0 列的首地址，也可以看作二维数组的首地址。&a[m][n]就是第 m 行 n 列元素的地址。

(3) &a[0]是第 0 行的首地址，当然&a[n]就是第 n 行的首地址。

(4) a[0]+n 表示第 0 行第 n 个元素的地址。

(5) \*(a+n)+m 表示第 n 行第 m 列元素。

(6) \*(a[n]+m)表示第 n 行第 m 列元素。

【例 7.17】 使用数组地址将二维数组输出。（实例位置：光盘\TM\sl\7\17）

```

#include<iostream>
using namespace std;
void main()
{
    int i,j;
    int a[4][3]={1,2,3},{4,5,6},{7,8,9},{10,11,12}};
    cout << "the array is: " << endl;
    for(i=0;i<4;i++)          //行

```



```

{
    for(j=0;j<3;j++)    //列
        cout <<*(a+i+j) << endl;
}

```

程序运行结果如图 7.28 所示。

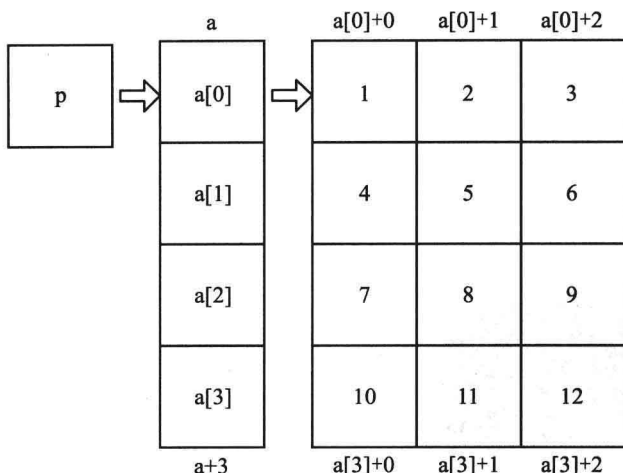


图 7.27 指针指向二维数组

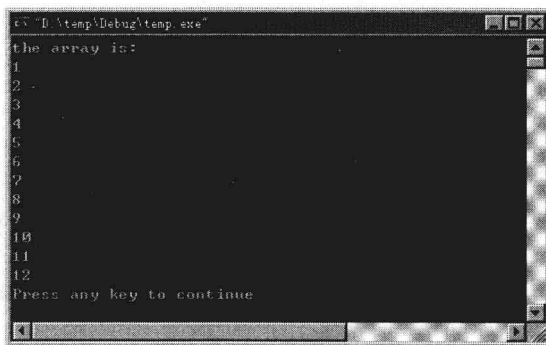


图 7.28 使用数组地址将二维数组输出

## 7.5.4 指针与字符数组

字符数组是一个一维数组，使用指针同样也可以引用字符数组。引用字符数组的指针为字符指针，字符指针就是指向字符型内存空间的指针变量，其一般的定义语句如下：

```
char *p;
```

字符数组就是一个字符串，通过字符指针可以指向一个字符串。例如，语句

```
char *string="www.mingri.book";
```

等价于下面两个语句：

```
char *string;
string="www.mingri.book";
```

为了使读者更好地了解指针与字符数组间的操作，下面通过实例实现连接两个字符数组的功能。

**【例 7.18】** 连接两个字符数组。（实例位置：光盘\TM\sl\7\18）

```

#include<iostream>
using namespace std;
void main()
{
    char str1[50],str2[30],*p1,*p2;

```

```

p1=str1;
p2=str2;
cout << "please input string1:"<< endl;
gets(str1);
cout << "please input string2:"<< endl;
gets(str2);
while(*p1!='\0')
p1++;
while(*p2!='\0')
*p1++=*p2++;
*p1='\0';
cout << "the new string is:"<< endl;
puts(str1);
}

```

程序运行结果如图 7.29 所示。

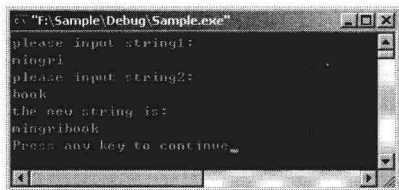


图 7.29 连接两个字符数组

程序需要用户输入两个字符数组，然后通过字符指针将两个字符串连接起来。

## 7.6 指向函数的指针

 视频讲解：光盘\TM\lx\7\指向函数的指针.exe

指针变量也可以指向一个函数。一个函数在编译时被分配给一个入口地址，这个函数入口地址就称为函数的指针。可以用一个指针变量指向函数，然后通过该指针变量调用此函数。

一个函数可以返回一个整型值、字符值、实型值等，也可以返回指针型的数据，即地址。其概念与之前类似，只是返回的值的类型是指针类型而已。返回指针类型的值的函数简称为指针函数。

定义指针函数的一般形式为：

**类型名 \*函数名(参数表列);**

例如，定义一个具有两个参数和一个返回值的函数的指针，其代码如下：

```
int *a(int x,int y);
```

下面通过实例实现使用指针函数进行平均值计算的功能。

**【例 7.19】** 使用指针函数进行平均值计算。（实例位置：光盘\TM\sl\7\19）

在未使用指针函数计算平均值时，可以通过自定义函数 avg 来进行平均值计算。计算哪两个整型数的平均值，直接将两个整型数传递给 avg 函数，avg 函数完成计算后将计算结果传出，其实现代码如下：

```

#include<iostream>
#include<iomanip>
using namespace std;
int avg(int a,int b);
void main()
{
    int iWidth,iLenght,iResult;
    i=10;
    j=30;
    iResult=avg(i,j);
    cout << iResult <<endl;
}

int avg(int a,int b)
{
    return (a+b)/2;
}

```

avg 函数是一个具有两个参数和一个返回值的函数，可以定义一个指针函数指向该函数，指针函数必须具有两个整型参数和一个整型返回值的形式。使用指针函数进行平均值计算的代码如下：

```

#include<iostream>
#include<iomanip>
using namespace std;
int avg(int a,int b);
void main()
{
    int iWidth,iLenght,iResult;
    iWidth=10;
    iLenght=30;
    int (*pFun)(int,int); //定义函数指针
    pFun=avg;

    iResult=(*pFun)(iWidth,iLenght);
    cout << iResult <<endl;
}

int avg(int a,int b)
{
    return (a+b)/2;
}

```

指针 pFun 是指向 avg 函数的函数指针，调用 pFun 函数指针，就和调用函数 avg 一样。

## 7.7 引 用

 视频讲解：光盘\TM\7\引用.exe

引用实际上是一种隐式指针，它为对象建立一个别名，通过操作符&来实现。&是取地址操作符，

通过它可以获得地址。

引用的形式如下：

**数据类型 & 表达式;**

例如：

```
int a=10;
int & ia=i;
ia=2;
```

定义了一个引用变量 `ia`，它是变量 `a` 的别名，对 `ia` 的操作与对 `a` 的操作完全一样。`ia=2` 把 2 赋给 `a`，`&ia` 返回 `a` 的地址。执行 `ia=2` 和执行 `a=2` 等价。

使用引用的说明：

- (1) 一个 C++ 引用被初始化后，无法使用它再去引用另一个对象，它不能被重新约束。
- (2) 引用变量只是其他对象的别名，对它的操作与原来对象的操作具有相同作用。

(3) 指针变量与引用有两点主要区别：一是指针是一种数据类型，而引用不是一个数据类型，指针可以转换为它所指向变量的数据类型，以便赋值运算符两边的类型相匹配；而在使用引用时，系统要求引用和变量的数据类型必须相同，不能进行数据类型转换。二是指针变量和引用变量都用来指向其他变量，但指针变量使用的语法要复杂一些；而在定义了引用变量后，其使用方法与普通变量相同。

例如：

```
int a;
int *pa = &a;
int &ia=a;
```

- (4) 引用应该初始化，否则会报错。

例如：

```
int a;
int b;
int &a;
```

编译器会报出 “references must be initialized” 这样的错误，造成编译不能通过。

下面通过实例使读者更好地了解引用的使用，现输出引用的功能。

**【例 7.20】** 输出引用。(实例位置：光盘\TM\sl\7\20)

```
#include <iostream>
using namespace std;
void main()
{
    int a;
    int & ref_a = a;
    a=100;
    cout << "a=" << a << endl;
    cout << "ref_a=" << ref_a << endl;
    a=2;
```



```

cout << "a=" << a << endl;
cout << "ref_a=" << ref_a << endl;
int b=20;
ref_a=b;
cout << "a=" << a << endl;
cout << "ref_a=" << ref_a << endl;
ref_a--;
cout << "a=" << a << endl;
cout << "ref_a=" << ref_a << endl;
}

```

程序声明了变量 `a` 和一个对变量 `a` 的引用 `ref_a`，通过不断地改变变量 `a` 和引用 `ref_a` 的值使读者了解引用的使用，然后将改变的结果输出。程序运行结果如图 7.30 所示。

### 7.7.1 使用引用传递参数

在 C++ 语言中，函数参数的传递方式主要有两种，分别为值传递和引用传递。所谓值传递，是指在函数调用时，将实际参数的值赋值一份传递到调用函数中，这样如果在调用函数中修改了参数的值，其改变不会影响到实际参数的值。而引用传递则恰恰相反，如果函数按引用方式传递，在调用函数中修改了参数的值，其改变会影响到实际参数。

**【例 7.21】** 通过引用交换数值。（实例位置：光盘\TM\sl\7\21）

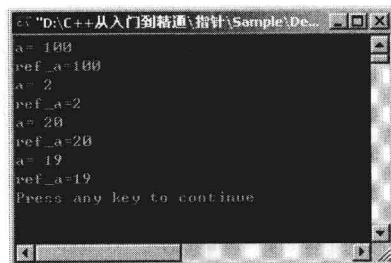


图 7.30 输出引用

```

#include<iostream>
using namespace std;
swap(int &a,int &b)
{
    int tmp;
    tmp=a;
    a=b;
    b=tmp;
}
void main()
{
    int x,y;
    cout << "input two number " << endl;
    cin >> x;
    cin >> y;

    if(x<y)
        swap(x,y);
    cout << "x=" << x << endl;
    cout << "y=" << y << endl;
}

```

程序运行结果如图 7.31 所示。

程序中自定义了函数 `swap`，该函数定义了两个引用参数。用户输入两个值，如果第一次输入的数值比第二次输入的数值小，则调用 `swap` 函数交换用户输入的数值。如果使用值传递方式，`swap` 函数就不能实现交换。

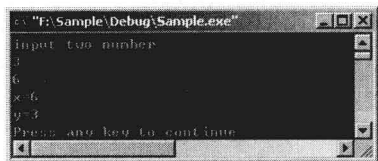


图 7.31 通过引用交换数值

## 7.7.2 指针传递参数

指针变量可以作为函数参数。使用指针变量传递参数和使用引用传递参数方式的执行效果相同，可以对同一个函数使用两种方式传递参数进行对比，观察程序的执行结果。

为了使读者更好地了解指针作为参数进行传递的操作，下面通过指针传递参数实现变量值的交换功能。

**【例 7.22】** 调用自定义函数交换两个变量值。（实例位置：光盘\TM\sl\7\22）

```
#include<iostream>
swap(int *a,int *b)
{
    int tmp;
    tmp=*a;
    *a=*b;
    *b=tmp;
}
void main()
{
    int x,y;
    int *p_x,*p_y;
    cout << "input two number " << endl;
    cin >> x;
    cin >> y;
    p_x=&x;p_y=&y;
    if(x<y)
        swap(p_x,p_y);
    cout << "x=" << x << endl;
    cout << "y=" << y << endl;
}
```

程序运行结果如图 7.32 所示。

从图 7.32 中可以看出，执行结果为输入“3，6”，输出“6，3”。

`swap` 函数是用户自定义函数，在 `main` 函数中调用该函数交换变量 `a` 和 `b` 的值。`swap` 函数的两个形参被传入了两个地址值，也就是传入了两个指针变量，在 `swap` 函数的函数体内使用整型变量 `tmp` 作为中转变量，将两个指针变量所指向的数值进行交换。在 `main` 函数内首先获取输入的两个数值，分别传递给变量 `x` 和 `y`，



图 7.32 调用自定义函数交换两变量值



然后比较两个变量  $x$  和  $y$  的大小, 如果  $x$  小于  $y$ , 就将  $x$  和  $y$  的地址值传递给 `swap` 函数, 调用 `swap` 函数将变量  $x$  和  $y$  的数值互换。

通过指针传递参数和通过引用传递参数一样, 都可以减少值传递带来的开销。但在开发程序时是使用指针还是使用引用类型作为函数参数呢? 实际上, 使用指针和引用类型作为函数参数各有优缺点, 应视具体环境而定。对于引用类型, 引用必须被初始化为一个对象, 并且不能使它再指向其他对象, 因为对应用赋值实际上是对目标对象赋值。这是引用类型的缺点, 但也是引用类型的优点, 因为在函数中不用验证引用参数的合法性。例如, 下面的函数调用是非法的。

```
void ValuePass(int &var)           //定义一个函数, 使用引用类型作为参数
{
    var = 10;                      //设置参数的值
    cout << var << endl;          //输出参数
}
int main(int argc, char* argv[])
{
    ValuePass(0);                  //非法的函数调用
    return 0;
}
```

程序中, 如果 `ValuePass` 采用指针作为函数参数, 使用 “`ValuePass(0);`” 语句调用是合法的, 但却带来了隐患, 因为 `0` 被认为是空指针, 对空指针操作必然会导致地址访问错误。因此对于指针对象作为函数参数, 函数体中需要验证指针参数是否为空, 这是使用指针类型作为函数参数的缺点。但是, 使用指针对象作为函数参数, 用户可以随意修改指针参数指向的对象, 这是引用类型参数所不能的。

### 7.7.3 数组作函数参数

在函数调用过程中, 有时需要传递多个参数, 如果传递的参数都是同一类型则可以通过数组的方式来传递参数, 作为参数的数组可以是一维数组, 也可以是多维数组。使用数组作函数参数最典型的的就是 `main` 函数。带参数的 `main` 函数的形式如下:

```
main(int argc, char *argv[])
```

`main` 函数中的参数可以获取程序运行的命令参数, 命令参数就是执行应用程序时后面带的参数。例如在 CMD 控制台执行 `dir` 命令, 可以带上 `/w` 参数, “`dir /w`” 命令是以多列的形式显示出文件夹内的文件名。`main` 函数中参数 `argc` 是获取命令参数的个数, `argv` 是字符指针数组, 可以获取具体的命令参数。

**【例 7.23】** 获取命令参数。(实例位置: 光盘\TM\sl\7\23)

```
#include<iostream>
using namespace std;
void main(int argc, char *argv[])
{
    cout << "the list of parameter:" << endl;
    while(argc>1)
    {
```

```

    ++argv;
    cout << *argv << endl;
    --argc;
}
}

```

上面代码在工程 sample 中将生成 sample.exe 应用程序，在执行 sample.exe 时在后面加上参数，程序就会输出命令参数。例如，执行命令及运行结果如图 7.33 所示。

程序执行时输入命令参数 “/a /b /c”，程序运行以后将 3 个命令参数输出，每个参数都是以空格隔开，应用程序后有 3 个空格，代表程序有 3 个命令参数，argc 的值就为 3。

二维数组在作为函数参数时，可以将二维数组转换成一个一维的指针数组。main 函数中的 argv 参数就可以是一个二维的字符数组。

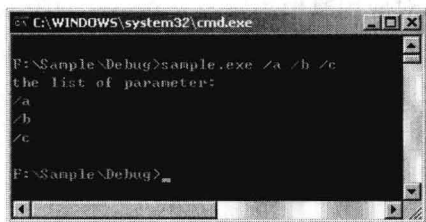


图 7.33 获取命令参数

**【例 7.24】** 输出每行数组中的最小值。（实例位置：光盘\TM\sl\7\24）

```

#include<iostream>
using namespace std;
#define N 4
void mix(int (*a)[N],int m)                                //进行比较和交换的函数
{
    int value,i,j;
    for(i=0;i<m;i++)
    {
        value=*(a+i);
        for(j=0;j<N;j++)
        {
            if(*(a+i)+j)<value)
            {
                value=*(a+i)+j;
            }
        }
        cout <<"line " << i ;
        cout <<".the mix number is " << value << endl;    //输出最小值
    }
}
void main()
{
    int a[3][N],i,j;
    int (*p)[N];
    p=&a[0];
    cout << "please input:" << endl;
    for(i=0;i<3;i++)
    {
        for(j=0;j<N;j++)
        {
            cin >> a[i][j];
        }
        mix(p,3);
    }
}

```

程序运行结果如图 7.34 所示。

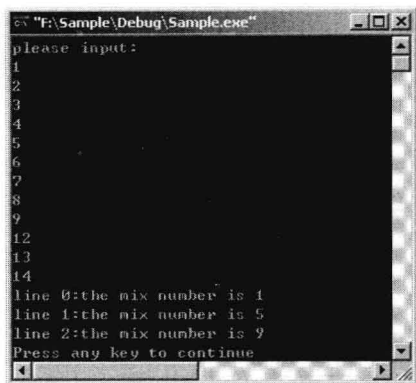


图 7.34 输出每行数组中的最小值

程序需要用户输入 12 个数值来作为一个 3 行 4 列数组的元素，然后按行进行比较，输出每行最小的元素。 $*(a+i)$ 代表数组每行第一个元素， $*(*(a+i)+j)$ 代表数组指定行中的某个列元素。函数 `mix` 对数组每行元素逐一进行比较，将最小值赋给变量 `value`，然后输出变量 `value` 的值。

## 7.8 指针数组

 视频讲解：光盘\TM\lx\7\指针数组.exe

数组中的元素均为指针变量的数组称为指针数组，一维指针数组的定义形式为：

类型名\*数组名[数组长度];

例如：

```
int *p[4];
```

指针数组中的数组名也是一个指针变量，该指针变量为指向指针的指针。

例如：

```
int *p[4];
int a=1;
*p[0]=&a;
```

`p` 是一个指针数组，它的每一个元素是一个指针型数据（其值为地址），指针数组 `p` 的第一个值是变量 `a` 的地址。指针数组中的元素可以使用指向指针的指针来引用。例如：

```
int *(*p);
```

\*运算符表示 `p` 是一个指针变量， $*(p)$ 表示指向指针的指针，\*运算符的结合性是从右到左，因此“`int *(*p);`”可写成“`int **p;`”。

指向指针的指针获取指针数组中的元素和利用指针获取一维数组中的元素方法相同，如图 7.35 所示。

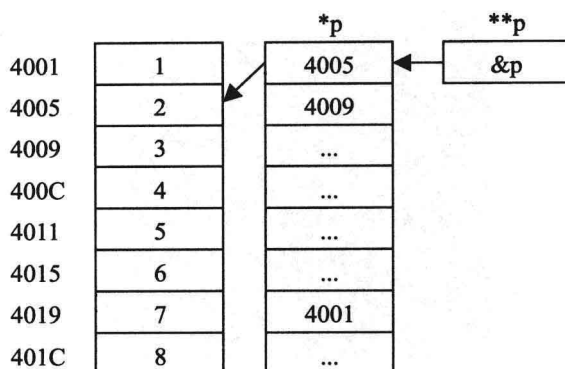


图 7.35 指向指针的指针

第一次进行指针\*运算获取到的是一个地址值，再进行一次指针\*运算，就可以获取到具体值。

**【例 7.25】** 将指针数组中各个元素分别指向若干个字符串。（实例位置：光盘\TM\sl\7\25）

```
#include<iostream>
using namespace std;
void sort(char *name[],int n)                //对字符串进行排序
{
    char *temp;
    int i,j,k;
    for(i=0;i<n-1;i++)
    {
        k=i;
        for(j=i+1;j<n;j++)
            if(strcmp(name[k],name[j])>0) k=j;
        if(k!=i)
        {
            temp=name[i];name[i]=name[k];name[k]=temp;
        }
    }
}
void print(char *name[],int n)                //输出字符串数组中的元素
{
    int i=0;
    char *p;
    p=name[0];
    while(i<n)
    {
        p=(name+i++);
        cout<<p<<endl;
    }
}
int main( )
{
    char *name[]={"mingri","soft","C++","mr"}; //定义指针数组
    int n=4;
```



```

sort(name,n);
print(name,n);
return 0;
}

```

程序运行结果如图 7.36 所示。

程序中的 print 函数中，数组名 name 代表该指针数组首元素的地址，name+i 是 name[i] 的地址。由于 name[i] 的值是地址（即指针），因此 name+i 就是指向指针型数据的指针。还可以设置一个指针变量 p，它指向指针数组的元素。p 就是指向指针型数据的指针变量，它所指向的字符串如图 7.37 所示。

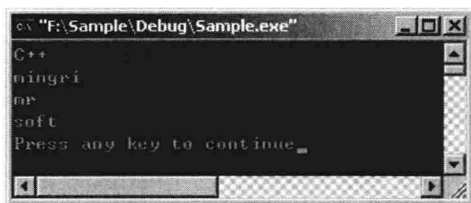


图 7.36 将指针数组中各个元素分别指向若干个字符串

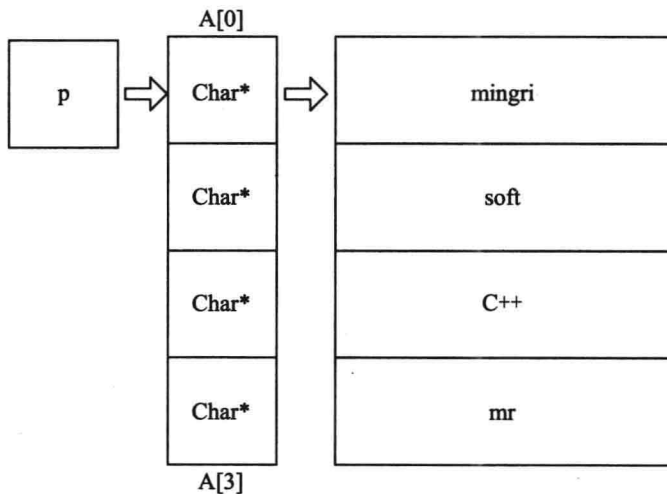


图 7.37 p 所指向的字符串

利用指针变量访问另一个变量就是间接访问。如果在一个指针变量中存放一个目标变量的地址，这就是单级间址。指向指针的指针用的是二级间址方法，还有三级间址和四级间址，但二级间址应用最为普遍。

## 7.9 小 结

指针是 C++ 语言中的重点及难点，它可以控制变量，可以操作数组，也可以指向函数，所以一定要理解指针的基本用法。本章讲述的都是指针的基本用法，要从概念上区分指针与变量的区别，要区分指针和数组首元素，既要学会使用指针传递参数，也要学会使用引用传递参数。

## 7.10 实践与练习

1. 使用字符数组和实型数组分别存储学生姓名和成绩，并通过对学生成绩的排序，按照名次输出

字符数组中对应的学生姓名。（答案位置：光盘\TM\sl\7\26）

2. 开发一个程序，通过二维数组存储星期信息，当输入 0~6 之间的数字时输出对应的星期信息。（答案位置：光盘\TM\sl\7\27）


3. 要求设计一个 3 行 4 列的二维数组，依次保存数据 1、3、5、7、9、11、13、15、17、19、21、23，然后使用指针的方式将数组中的数据输出显示。（答案位置：光盘\TM\sl\7\28）



# 第 8 章

---

## 构造数据类型


(  视频讲解：59 分钟 )

结构体可以将不同类型组合在一起形成一个新的类型，这个类型是对数据的整合，以使代码更加简洁。共用体和结构体很相近，它像一个存储空间可变的数据类型，使程序设计更加灵活。枚举则是特殊的常量，可以增加代码的可读性；自定义类型更是增加了代码的复用性。

通过阅读本章，您可以：

- » 掌握结构体
- » 了解共用体
- » 掌握枚举类型
- » 掌握自定义数据类型

## 8.1 结 构 体

 视频讲解：光盘\TM\lx\8\结构体.exe

### 8.1.1 结构体定义

整型、长整型、字符型、浮点型这些数据类型只能记录单一的数据，这些数据类型只能被称作基础数据类型。如果要描述一个人的信息，就需要定义多个变量来记录这些信息，例如，身高需要一个变量，体重需要一个变量，姓名需要一个变量，年龄需要一个变量。如果有一个类型可以将这些变量包含在一起，则会大大减少程序代码的离散性，使程序代码阅读更加符合逻辑。结构体则是实现这一功能的类型。

结构体的定义如下：

```
struct 结构体类型名
{
    成员类型    成员名;
    .....
    成员类型    成员名;
};
```

**struct** 就是定义结构体的关键字。结构体类型名是一种标识符，该标识符代表一个新的变量。结构体使用大括号将成员括起来，每个成员都有自己的类型，成员类型可以是常规的基础类型，也可以是自定义类型，同时还可以是一个类类型。

例如，定义一个简单员工信息的结构体，代码如下：

```
struct PersonInfo
{
    int index;
    char name[30];
    short age;
};
```

结构体类型名是 **PersonInfo**，在结构体中定义了 3 个不同类型的变量。这 3 个变量就好像是 3 个球放到了一个盒子里，只要能找到这个盒子就能找到这 3 个球。同样，找到名字为 **PersonInfo** 的结构体，就可以找到结构体下的变量。这 3 个变量的数据类型各不相同，有字符串型，有整型，分别定义了员工的编号、姓名和年龄。



**说明**

给结构体下个定义，就是由多个不同类型的数据组成的数据集合。而数组则是相同元素的集合。

### 8.1.2 结构体变量

结构体是一个构造类型，前面只是定义了结构体，形成一个新的数据类型，还需要使用该数据类型来定义变量。

结构体变量有两种声明形式。第一种声明形式是在定义结构体后，使用结构体类型名声明。例如：

```
struct PersonInfo
{
    int index;
    char name[30];
    short age;
};
PersonInfo plnfo;
```

另一种声明形式是定义结构体时直接声明。例如：

```
struct PersonInfo
{
    int index;
    char name[30];
    short age;
} plnfo;
```

直接声明结构体变量时，可以声明多个变量。例如：

```
struct PersonInfo
{
    int index;
    char name[30];
    short age;
} plnfo1, plnfo2;
```

### 8.1.3 结构体成员及初始化

引用结构体成员有两种方式，一种是声明结构体变量后，通过成员运算符“.”引用；一种是声明结构体指针变量，使用指向“->”运算符引用。

(1) 使用成员运算符“.”引用结构体成员的一般形式如下：

结构体变量名.成员名

例如：

```
struct PersonInfo
{
    int index;
```

```

    char name[30];
    short age;
} plInfo;
plInfo.index
plInfo.name
plInfo.age

```

引用结构体成员后，就可以分别对结构体成员进行赋值，对于每个结构体成员就和使用普通变量一样。

下面通过实例来看如何为结构体成员赋值。

**【例 8.1】** 为结构体成员赋值。（实例位置：光盘\TM\sl\8\1）

```

#include<iostream>
using namespace std;
void main()
{
    struct PersonInfo
    {
        int index;
        char name[30];
        short age;
    } plInfo;
    plInfo.index=0;
    strcpy(plInfo.name,"张三");
    plInfo.age=20;
    cout << plInfo.index << endl;
    cout << plInfo.name << endl;
    cout << plInfo.age << endl;
}

```

程序运行结果如图 8.1 所示。



图 8.1 为结构体成员赋值

程序中分别引用结构体的每个成员，然后赋值，其中为字符数组赋值需要使用字符串复制函数 strcpy。结构体可以在定义时直接对结构体变量赋值。例如：

```

struct PersonInfo
{
    int index;
    char name[30];
    short age;
} plInfo={0,"张三",20};

```

(2) 在定义结构体时，可以同时声明结构体指针变量。例如：

```
struct PERSONINFO
{
    int index;
    char name[30];
    short age;
}*pPersonInfo;
```

如果要引用指针结构体变量的成员，需要使用指向“->”运算符。一般形式如下：

结构体指针变量->成员名

例如：

```
pPersonInfo-> index
pPersonInfo-> name
pPersonInfo-> age
```



**注意**

指针结构体指针变量只有初始化后才可以使用。

下面通过实例来看如何通过结构体指针变量来引用结构体成员。

**【例 8.2】** 使用结构体指针变量引用结构体成员。（实例位置：光盘\TM\sl\8\2）

```
#include<iostream>
using namespace std;
void main()
{
    struct PERSONINFO
    {
        int index;
        char name[30];
        short age;
    }*pPersonInfo, pInfo={0,"张三",20};
    pPersonInfo=&pInfo;
    cout << pPersonInfo->index << endl;
    cout << pPersonInfo->name << endl;
    cout << pPersonInfo->age << endl;
}
```

程序运行结果如图 8.2 所示。

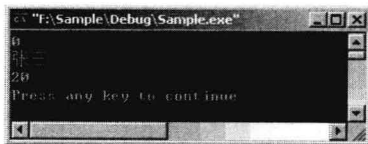


图 8.2 使用结构体指针变量引用结构体成员



### 8.1.4 结构体的嵌套

定义完结构体后就形成了一个新的数据类型。C++语言在定义结构体时可以声明其他已定义好的结构体变量，也可以在定义结构体时定义子结构体。

(1) 在结构体中定义子结构体。例如：

```
struct PersonInfo
{
    int index;
    char name[30];
    short age;
    struct WorkPlace
    {
        char Address[150];
        char PostCode[30];
        char GateCode[50];
        char Street[100];
        char Area[50];
    };
};
```

(2) 在定义时声明其他已定义好的结构体变量。例如：

```
struct WorkPlace
{
    char Address[150];
    char PostCode[30];
    char GateCode[50];
    char Street[100];
    char Area[50];
};
struct PersonInfo
{
    int index;
    char name[30];
    short age;
    WorkPlace myWorkPlace;
};
```

通过上面的两种形式都可以完成结构体的嵌套，下面的实例通过第一种方式来实现结构体的嵌套。

**【例 8.3】** 使用嵌套的结构体。（实例位置：光盘\TM\sl\8\3）

```
#include<iostream>
using namespace std;
void main()
{
    struct PersonInfo
```



```

{
    int index;
    char name[30];
    short age;
    struct WorkPlace
    {
        char Address[150];
        char PostCode[30];
        char GateCode[50];
        char Street[100];
        char Area[50];
    }WP;
}

PersonInfo plnfo;
strcpy(plnfo.WP.Address,"House");
strcpy(plnfo.WP.PostCode,"10000");
strcpy(plnfo.WP.GateCode,"302");
strcpy(plnfo.WP.Street,"Lan Tian");
strcpy(plnfo.WP.Area,"china");

cout << plnfo.WP.Address << endl;
cout << plnfo.WP.PostCode << endl;
cout << plnfo.WP.GateCode << endl;
cout << plnfo.WP.Street << endl;
cout << plnfo.WP.Area << endl;
}

```

程序运行结果如图 8.3 所示。

程序在 PersonInfo 结构体中嵌套了 WorkPlace 结构体，然后分别对 WorkPlace 子结构体中的成员进行赋值，最后将 WorkPlace 子结构体中的成员输出。

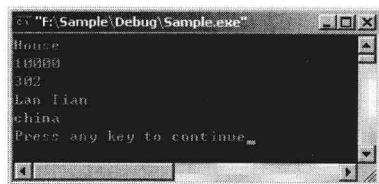


图 8.3 使用嵌套的结构体

### 8.1.5 结构体大小

结构体是一种构造的数据类型，数据类型都与占用内存多少有关。在没有字符对齐要求或结构成员对齐单位为 1 时，结构体变量的大小是定义结构体时各成员大小之和。例如：

```

struct PersonInfo
{
    int index;
    char name[30];
    short age;
};

```

其中 PersonInfo 结构体的大小是成员 name、成员 index 和成员 age 大小之和。成员 name 是字符数组，

一个字符占用 1 个字节，共占用 30 个字节；成员 index 是整型数据，在 32 位系统中占用 4 个字节；age 是短整型，在 32 位系统中占用 2 个字节。所以 PersonInfo 结构体的大小是  $30+4+2=36$  字节。

可以使用 sizeof 运算符获取结构体大小。例如：

```
#include<iostream>
using namespace std;
void main()
{
    struct PersonInfo
    {
        int index;
        char name[30];
        short age;
    }pInfo;
    cout << sizeof(pInfo) << endl;
}
```

程序使用 sizeof 运算符输出的结果仍然是 36。

如果更改结构成员对齐单位，PersonInfo 结构体实际占用的内存空间就不是 36 字节了。在 VC 6.0 中可以通过修改工程属性来改变结构成员对齐单位。通过选择 Project/Settings 菜单命令打开 Project Settings 对话框，如图 8.4 所示，选择 C/C++ 选项卡，在 Category 下拉列表框中选择 Code Generation 选项，然后通过 Struct member alignment 下拉列表框即可改变结构成员对齐单位。

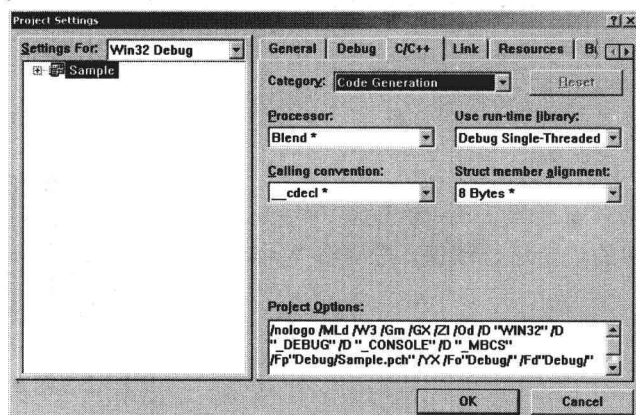


图 8.4 Project Settings 对话框

默认结构成员对齐单位是 8 个字节，结构成员对齐单位在使用结构体变量传送数据时能看到其差异。

## 8.2 结构体与函数

 视频讲解：光盘\TM\lx\8\结构体与函数.exe

结构体数据类型在 C++ 语言中是可以作为函数参数传递的，可以直接使用结构体变量作函数的参数，

也可以使用结构体指针变量作函数的参数。

### 8.2.1 结构体变量作函数参数

可以把结构体变量当普通变量一样作为函数参数，这样可以减少函数参数的个数，使代码看起来更简洁。

下面通过实例来了解如何使用结构体变量作函数参数进行传递。

**【例 8.4】** 使用结构体变量作函数参数。（实例位置：光盘\TM\sl\8\4）

```
#include<iostream>
using namespace std;
struct PersonInfo                                //定义结构体
{
    int index;
    char name[30];
    short age;
};
void ShowStuctMessage(struct PersonInfo MyInfo) //自定义函数，输出结构体变量成员
{
    cout << MyInfo.index << endl;
    cout << MyInfo.name << endl;
    cout << MyInfo.age<< endl;
}
void main()
{
    PersonInfo pInfo;                            //声明结构体
    pInfo.index=1;
    strcpy(pInfo.name,"张三");
    pInfo.age=20;
    ShowStuctMessage(pInfo);                     //调用自定义函数
}
```

程序运行结果如图 8.5 所示。

程序中自定义了函数 ShowStuctMessage，该函数使用 PersonInfo 结构体作为参数。如果不使用结构体作为参数，函数需要将 index、name、age 3 个成员分别定义为参数。

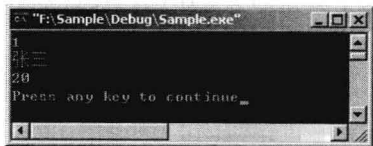


图 8.5 使用结构体变量作函数参数

### 8.2.2 结构体指针作函数参数

使用结构体指针变量作函数参数时传递的只是地址，减少了时间和空间上的开销，能够提高程序的运行效率。这种方式在实际应用中效果比较好。

下面通过实例来看如何使用结构体指针作函数参数进行传递。

**【例 8.5】** 使用结构体指针变量作函数参数。（实例位置：光盘\TM\sl\8\5）

```
#include<iostream>
using namespace std;
struct PersonInfo
{
    int index;
    char name[30];
    short age;
};
void ShowStuctMessage(struct PersonInfo *pInfo)
{
    cout << pInfo->index << endl;
    cout << pInfo->name << endl;
    cout << pInfo->age << endl;
}
void main()
{
    PersonInfo pInfo;
    pInfo.index=1;
    strcpy(pInfo.name,"张三");
    pInfo.age=20;
    ShowStuctMessage(&pInfo);
}
```

程序运行结果如图 8.6 所示。

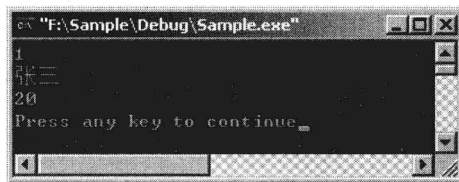


图 8.6 使用结构体指针变量作函数参数

例 8.5 和例 8.4 的运行结果相同，但在程序执行效率上，使用结构体指针作函数参数方式要快很多。

## 8.3 结构体数组

 视频讲解：光盘\TM\lx\8\结构体数组.exe

数组的元素也可以是结构类型的，因此可以构成结构体数组。结构体数组的每一个元素都是具有相同结构类型的下标结构变量。



### 8.3.1 结构体数组声明与引用

结构体数组可以在定义结构体时声明，可以使用结构体变量声明，也可以直接声明结构体数组而无须定义结构体名。

(1) 在定义结构体时直接声明。例如：

```
struct PersonInfo
{
    int index;
    char name[30];
    short age;
}Person[5];
```

(2) 使用结构体变量声明。例如：

```
struct PersonInfo
{
    int index;
    char name[30];
    short age;
}pInfo;
PersonInfo Person[5]
```

(3) 直接声明结构体数组。例如：

```
struct
{
    int index;
    char name[30];
    short age;
}Person[5];
```

也可以在声明结构体数组时直接对数组进行初始化。

```
struct PersonInfo
{
    int index;
    char name[30];
    short age;
}Person[5]={1,"张三",20},
           {2,"李可可",21},
           {3,"宋桥",22},
           {4,"元员",22},
           {5,"王冰冰",22}};
```



说明

当对全部元素作初始化赋值时，也可不给出数组长度。

### 8.3.2 指针访问结构体数组

指针变量可以指向一个结构数组，这时结构指针变量的值是整个结构数组的首地址。结构指针变量也可指向结构数组的一个元素，这时结构指针变量的值是该结构数组元素的首地址。

**【例 8.6】** 使用指针访问结构体数组。（实例位置：光盘\TM\sl\8\6）

```
#include<iostream>
using namespace std;
void main()
{
    struct PersonInfo
    {
        int index;
        char name[30];
        short age;
    }Person[5]={1,"张三",20},
               {2,"李可可",21},
               {3,"宋桥",22},
               {4,"元员",22},
               {5,"王冰冰",22};

    struct PersonInfo *pPersonInfo;
    pPersonInfo=Person;
    for(int i=0;i<5;i++,pPersonInfo++)
    {
        cout << pPersonInfo->index << endl;
        cout << pPersonInfo->name << endl;
        cout << pPersonInfo->age << endl;
    }
}
```

程序运行结果如图 8.7 所示。

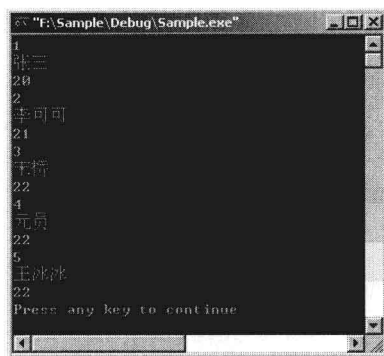



图 8.7 使用指针访问结构体数组

程序的关键在于 `pPersonInfo++` 的运算上，`pPersonInfo` 指针开始指向数组的首元素，结构体指针自



加 1，其结果使 pPersonInfo 指针指向了数组的下一个元素。

## 8.4 共用体

 视频讲解：光盘\TM\lx\8\共用体.exe

所谓共用体数据类型是指将不同的数据类型组织为一个整体，它和结构体有些类似，但共用体在内存中占用首地址相同的一段存储单元。因为共用体的关键字为 **union**，中文意思为联合，所以共用体也称为联合体。

### 8.4.1 共用体的定义与声明

定义共用体类型的一般形式为：

```
union 共用体类型名
{
    成员类型    共用体成员名 1;
    成员类型    共用体成员名 2;
    ...
    成员类型    共用体成员名 n;
};
```

**union** 是定义共用体数据类型的关键字，共用体类型名是一个标识符，该标识符以后就是一个新的数据类型。成员类型是常规的数据类型，用来设置共用体成员存储空间。

声明共用体数据类型变量有以下几种方式。

(1) 先定义共用体，然后声明共用体变量。例如：

```
union myUnion
{
    int i;
    char ch;
    float f;
};
myUnion u;    //声明变量
```

(2) 直接在定义时声明共用体变量。例如：

```
union myUnion
{
    int i;
    char ch;
    float f;
}u;    //直接声明变量
```

(3) 直接声明共用体变量。例如：

```
union
{
    int i;
    char ch;
    float f;
}u;
```

第三种方式省略了共用体类型名，直接声明了变量 `u`。

引用共用体对象成员和引用结构体对象类型的方式相同，也是使用 “.” 运算符。例如：

```
u.i
u.ch
u.f
```

上面是对共用体 `u` 的 3 个成员的引用，但要注意不能引用共用体变量，而只能引用共用体变量中的成员。例如直接引用 `u` 是错误的。

## 8.4.2 共用体的大小

共用体每个成员分别占有自己的内存单元。共用体变量所占的内存长度等于最长的成员的长度。一个共用体变量不能同时存放多个成员的值，某一时刻只能存放其中的一个成员的值，这就是最后赋予它的值。

**【例 8.7】** 使用共用体变量。（实例位置：光盘\TM\sl\8\7）

```
#include<iostream>
using namespace std;
union myUnion
{
    int iData;
    char chData;
    float fData;
}uStruct;
int main()
{
    uStruct.chData='A';
    uStruct.fData=0.3;
    uStruct.iData=100;
    cout << uStruct.chData << endl;
    cout << uStruct.fData << endl;
    cout << uStruct.iData << endl;    //正确显示
    uStruct.iData=100;
    uStruct.fData=0.3;
    uStruct.chData='A';
    cout << uStruct.chData << endl;    //正确显示
    cout << uStruct.fData << endl;
    cout << uStruct.iData << endl;
    uStruct.iData=100;
```

```

uStruct.chData='A';
uStruct.fData=0.3;
cout << uStruct.chData << endl;
cout << uStruct.fData << endl;    //正确显示
cout << uStruct.iData << endl;
return 0;
}

```

程序运行结果如图 8.8 所示。

程序中按不同顺序为 uStruct 变量的 3 个成员赋值,结果显示只有最后赋值的成员能正确显示。

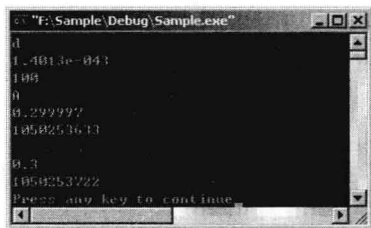


图 8.8 使用共用体变量

### 8.4.3 共用体的特点

共用体数据类型有以下几个特点:

- (1) 使用共用体变量的目的是希望用同一个内存段存放几种不同类型的数据,但要注意在每一瞬时只能存放其中一种,而不是同时存放几种。
- (2) 能够访问的是共用体变量中最后一次被赋值的成员,在对一个新的成员赋值后原有的成员就失去作用。
- (3) 共用体变量的地址和它的各成员的地址都是同一地址。
- (4) 不能对共用体变量名赋值;不能企图引用变量名来得到一个值;不能在定义共用体变量时对其初始化;不能用共用体变量名作为函数参数。

## 8.5 枚举类型

 视频讲解: 光盘\TM\lx\8\枚举类型.exe

枚举就是一一列举的意思,在 C++ 语言中枚举类型是一些标识符的集合,从形式上看枚举类型就是用大括号将不同标识符名称放在一起。用枚举类型声明的变量,其变量的值只能取自括号内的这些标识符。

### 8.5.1 枚举类型的声明

枚举类型定义有以下两种声明形式。

- (1) 枚举类型的一般形式。

```
enum 枚举类型名 {标识符列表};
```

例如:

```
enum weekday{Sunday,Monday,Tuesday,Wednesday,Thursday,Friday,Saturday};
```

enum 是定义枚举类型的关键字，weekday 是新定义的类型名，大括号内就是枚举类型变量应取的值。

(2) 带赋值的枚举类型声明形式。

```
enum 枚举类型名
{
    标识符[=整型常数],
    标识符[=整型常数],
    .....
    标识符[=整型常数],
} 枚举变量;
```

例如：

```
enum weekday{Sunday=0,Monday=1,Tuesday=2,Wednesday=3,Thursday=4,Friday=5,Saturday=6};
```

使用枚举类型的说明：

(1) 编译器默认将标识符自动赋予整型常数。例如：

```
enum weekday{Sunday,Monday,Tuesday,Wednesday,Thursday,Friday,Saturday};
enum weekday{Sunday=0,Monday=1,Tuesday=2,Wednesday=3,Thursday=4,Friday=5,Saturday=6};
```

(2) 可以自行修改整型常数的值。例如：

```
enum weekday{Sunday=2,Monday=3,Tuesday=4,Wednesday=5,Thursday=0,Friday=1,Saturday=6};
```

(3) 如果只给前几个标识符赋整型常数，编译器会给后面的标识符自动累加赋值。例如：

```
enum weekday{Sunday=7,Monday=1,Tuesday,Wednesday,Thursday,Friday,Saturday};
```

相当于：

```
enum weekday{Sunday=7,Monday=1,Tuesday=2,Wednesday=3,Thursday=4,Friday=5,Saturday=6};
```

## 8.5.2 枚举类型变量

在声明了枚举类型之后，可以用它来定义变量。例如：

```
enum weekday{Sunday,Monday,Tuesday,Wednesday,Thursday,Friday,Saturday};
[enum] weekday myworkday;
```

其中 myworkday 是 weekday 的变量。在 C 语言中，枚举类型名包括关键字 enum，在 C++ 语言中允许不写 enum 关键字。

关于使用枚举类型变量的说明：

(1) 枚举变量的值只能是 Sunday 到 Saturday 之一。例如：

```
myworkday = Tuesday;
myworkday = Saturday;
```



(2) 一个整型数据不能直接赋给一个枚举变量。例如：

```
enum weekday{Sunday=7,Monday=1,Tuesday,Wednesday,Thursday,Friday,Saturday};
enum weekday day;
```

则“day=(enum weekday)3;”等价于“day=Wednesday;”。“day=3;”是错误的。

整型数据虽然不能直接为枚举类型变量赋值，但是可以通过强制类型转换，将整型数据转换为合适的枚举型数值。

**【例 8.8】** 枚举变量的赋值。(实例位置：光盘\TM\sl\8\8)

```
#include<iostream>
using namespace std;
void main()
{
    enum Weekday {Sunday,Monday,Tuesday,Wednesday,Thursday,Friday,Saturday};
    int a=2,b=1;
    Weekday day;
    day=(Weekday)a;
    cout << day << endl;
    day=(Weekday)(a-b);
    cout << day << endl;
    day=(Weekday)(Sunday+Wednesday);
    cout << day << endl;
    day=(Weekday)5;
    cout << day << endl;
}
```

程序运行结果如图 8.9 所示。

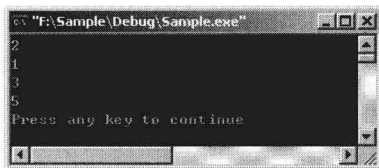


图 8.9 枚举变量的赋值

程序中使用了各种形式的赋值，其原理都是一样的，都是通过强制转换为枚举变量赋值。

(3) 可以直接定义枚举变量。例如：

```
enum{sun, mon, tue, wed, thu, fri, sat} workday,week_end;
```

### 8.5.3 枚举类型的运算

枚举值相当于整数，可以用枚举值来进行一些运算。

(1) 利用枚举值作判断比较。

枚举值可以和整型变量一起比较，枚举值和枚举值之间也可以比较。

**【例 8.9】** 枚举值的比较运算。（实例位置：光盘\TM\sl\8\9）

```
#include<iostream>
using namespace std;
enum Weekday {Sunday,Monday,Tuesday,Wednesday,Thursday,Friday,Saturday};
void main()
{
    Weekday day1,day2;
    day1=Monday;
    day2=Saturday;
    int n;
    n=day1;
    n=day2+1;
    if(n>day1)           //可以比较
        cout << "n>day1" << endl;
    if(day1<day2)
        cout << "day1<day2" << endl;
}
```

程序运行结果如图 8.10 所示。

程序进行变量 *n* 和枚举变量 *day1* 的比较以及枚举变量 *day1* 和 *day2* 的比较。

（2）利用枚举类型进行减法运算。枚举值可以直接进行减法运算。

**【例 8.10】** 使用枚举值进行减法运算。（实例位置：光盘\TM\sl\8\10）

```
#include<iostream>
using namespace std;
void main()
{
    enum weekday{Sunday=2,Monday=3,Tuesday=4,Wednesday=5,Thursday=0,Friday=1,Saturday=6};

    weekday m_eType1;
    weekday m_eType2;
    m_eType1=Sunday;//Sunday=2
    m_eType2=Friday;//Friday=1
    cout << m_eType1-m_eType2 << endl;
}
```

程序运行结果如图 8.11 所示。

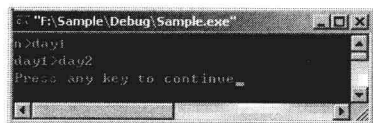


图 8.10 枚举值的比较运算

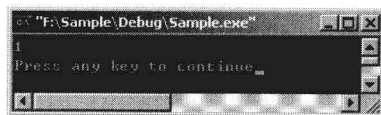


图 8.11 使用枚举值进行减法运算

程序直接将两个枚举变量 *m\_eType1* 和 *m\_eType2* 进行减法运算，然后将运算结果输出。



## 8.6 自定义数据类型

 视频讲解：光盘\TM\lx\8\自定义数据类型.exe

C++语言提供了丰富的数据类型，对于一些由用户自己定义的构造型数据类型，还允许用户自己定义类型说明符，也就是说由用户为定义的数据类型名另外再取一个别名，以便简化对类型名的引用，或增加程序的可读性。这个功能由类型定义符 `typedef` 完成。`typedef` 的使用形式如下：

```
typedef <原类型名> <新类型名>
```

原类型名是任意已定义的数据类型，包括系统的各种基本数据类型名以及用户自定义的构造类型名。新类型名是用户自己命名的标识符，在以后变量的声明中可以直接使用该名称。

例如：

```
typedef int INTEGER;
INTEGER a,b;
```

等价于：

```
int a,b;
```

下面通过实例来介绍如何在程序中使用自定义的数据类型。

**【例 8.11】** 使用自定义数据类型。（实例位置：光盘\TM\lx\8\11）

```
typedef char * CString;
#include<iostream.h>
void main()
{
    CString str;
    char temp[]="Hello World";
    str=temp;
    cout << str << endl;
}
```

程序运行结果如图 8.12 所示。



图 8.12 使用自定义数据类型

程序将字符指针重命名为 `CString`，`CString` 就代表字符指针。自定义数据类型的大小同原数据类型一样。例如：

```
#include<iostream.h>
void main()
```

```
{  
    typedef char * CString;  
    cout << sizeof(CString) << endl;  
    cout << sizeof(char*) << endl;  
}
```

程序使用 `sizeof` 运算符得到数据类型的大小。`CString` 同 `char*` 一样，都是占用 4 个字节空间。

## 8.7 小 结

本章主要介绍了结构体、共用体、枚举类型及自定义类型。使用 C 语言开发的程序一般都大量使用结构体，在 C++ 语言中更是增加了结构体的功能，程序设计阶段应多将关联紧密的数据组合成一个结构体，便于阅读及二次开发。

## 8.8 实践与练习

1. 输入学生期中成绩、期末成绩、平时考核成绩，按 30%、50%、20% 计算学生的综合成绩。（答案位置：光盘\TM\sl\8\12）
2. 设计一个选票程序。假设有 3 个候选人，每一次输入要选择候选人姓名，最后输出每个人的得票结果。（答案位置：光盘\TM\sl\8\13）

# 第 2 篇

## 核心技术

» 第 9 章 面向对象编程

» 第 10 章 类和对象

» 第 11 章 继承与派生


本篇介绍了 C++ 语言的关于面向对象方面的内容，理解面向对象这个概念，应用类类型创建对象，掌握什么是继承和派生，利用多态进行面向对象开发。



# 第 9 章

---

## 面向对象编程

(  视频讲解：32 分钟 )

面向对象编程可以有效解决代码复用问题，它不同于以往的面向过程编程，面向过程编程需要将功能细分，而面向对象编程需要将不同功能抽象到一起。本章通过具体 UML 建模来演示面向对象编程思想，通过对一个程序的前期分析了解如何使用面向对象编程。

通过阅读本章，您可以：

- » 了解面向对象模式
- » 了解面向对象编程与面向过程编程的区别
- » 了解面向对象编程开发过程
- » 掌握简单建模



## 9.1 面向对象概述

 视频讲解：光盘\TM\lx\9\面向对象概述.exe

面向对象（Object Oriented）的英文缩写是 OO，它是一种设计思想，现在这种思想已经不单应用在软件设计上，数据库设计、计算机辅助设计（CAD）、网络结构设计、人工智能算法设计等领域都开始应用这种思想。

面向对象中的对象（Object），指的是客观世界中存在的对象，这个对象具有唯一性，对象之间各不相同，各有各的特点，每一个对象都有自动的运动规律和内部状态。对象与对象之间又是可以相互联系、相互作用的。概括地讲，面向对象技术是一种从组织结构上模拟客观世界的方法。

针对面向对象思想应用的不同领域，面向对象又可以分为面向对象分析（Object Oriented Analysis, OOA）、面向对象设计（Object Oriented Design, OOD）、面向对象编程（Object Oriented Programming, OOP）、面向对象测试（Object Oriented Test, OOT）和面向对象维护（Object Oriented Soft Maintenance, OOSM）。

客观世界中任何一个事物都可以看成一个对象，每个对象有属性和行为两个要素。属性就是对象的内部状态及自身的特点，行为就是改变自身状态的动作。

面向对象中的对象也可以是一个抽象的事物，可以从类似的事物中抽象出一个对象，例如圆形、正方形、三角形，可以抽象得出的对象是简单图形，简单图形就是一个对象，它有自己的属性和行为，图形中边的个数是它的属性，图形的面积也是它的属性，输出图形的面积就是它的行为。

面向对象有三大特点，即封装、继承和多态。

### （1）封装

封装有两个作用，一个是将不同的小对象封装成一个大对象，另一个是把一部分内部属性和功能对外界屏蔽。例如一辆汽车，它是一个大对象，它由发动机、底盘、车身和轮子等这些小对象组成。在设计时可以先对这些小对象进行设计，然后小对象之间通过相互联系确定各自大小等方面的属性，最后就可以安装成一辆汽车。

### （2）继承

继承是和类密切相关的概念。继承性是子类自动共享父类数据结构和方法的机制，这是类之间的一种关系。在定义和实现一个类时，可以在一个已经存在的类的基础之上进行，把这个已经存在的类所定义的内容作为自己的内容，并加入若干新的内容。

在类层次中，子类只继承一个父类的数据结构和方法，称为单重继承，子类继承了多个父类的数据结构和方法，则称为多重继承。

在软件开发中，类的继承性使所建立的软件具有开放性、可扩充性，这是信息组织与分类的行之有效的办法，它简化了对象、类的创建工作量，增加了代码的可重用性。

继承性是面向对象程序设计语言不同于其他语言的最重要的特点，是其他语言所没有的。采用继承性，使公共的特性能够共享，提高了软件的重用性。

### （3）多态

多态性是指相同的行为可作用于多种类型的对象上并获得不同的结果。不同的对象，收到同一消息可以产生不同的结果，这种现象称为多态性。多态性允许每个对象以适合自身的方式去响应共同的消息。

## 9.2 面向对象与面向过程编程

 视频讲解：光盘\TM\lx\9\面向对象与面向过程编程.exe

### 9.2.1 面向过程编程

面向过程编程的主要思想是先做什么后做什么，在一个过程中实现特定功能。一个大的实现过程还可以分成多个模块，各个模块可以按功能进行划分，然后组合在一起实现特定功能。在面向过程编程中，程序模块可以是一个函数，也可以是整个源文件。

面向过程编程主要以数据为中心，传统的面向过程的功能分解法属于结构化分析方法。分析者将对象系统的现实世界看作一个大的处理系统，然后将其分解为若干个子处理过程，解决系统的总体控制问题。在分析过程中，用数据描述各子处理过程之间的联系，整理各子处理过程的执行顺序。

面向过程编程一般流程如下：

现实世界→面向过程建模（流程图，变量，函数）→面向过程语言→执行求解

面向过程编程的稳定性、可修改性和可重用性都比较差。

#### （1）软件重用性差

重用性是指同一事物不经修改或稍加修改就可多次重复使用的性质。软件重用性是软件工程追求的目标之一。处理不同的过程都有不同的结构，当过程改变时，结构也需要改变，前期开发的代码无法得到充分的再利用。

#### （2）软件可维护性差

软件工程强调软件的可维护性，强调文档资料的重要性，规定最终的软件产品应该由完整、一致的配置成分组成。在软件开发过程中，始终强调软件的可读性、可修改性和可测试性是软件的重要的质量指标。面向过程编程由于软件的重用性差，造成维护时其费用和成本也很高，而且大量修改的代码存在着许多未知的漏洞。

#### （3）开发出的软件不能满足用户需要

大型软件系统一般涉及各种不同领域的知识，面向过程编程往往描述软件的各个最低层的、针对不同领域设计不同的结构及处理机制，当用户需求发生变化时，就要修改最低层的结构。当处理用户需求变化较大时，面向过程编程将无法修改，可能导致软件的重新开发。

### 9.2.2 面向对象编程

面向过程编程有费解的数据结构、复杂的组合逻辑、详细的过程和数据之间的关系、高深的算法，面向过程开发的程序可以描述成算法加数据结构。面向过程开发是分析过程与数据之间的边界在哪里，进而解决问题。面向对象则是从另一种角度思考，将编程思维设计成符合人的思维逻辑。

面向对象程序设计者的任务包括两个方面：一是设计所需的各种类和对象，即决定把哪些数据和操

作封装在一起；二是考虑怎样向有关对象发送消息，以完成所需的任务。这时它如同一个总调度，不断地向各个对象发出消息，让这些对象活动起来（或者说激活这些对象），完成自己职责范围内的工作。

各个对象的操作完成了，整体任务也就完成了。显然，对一个大型任务来说，面向对象程序设计方法是十分有效的，它能大大降低程序设计人员的工作难度，减少出错几率。

面向对象开发的程序可以描述成“对象+消息”。面向对象编程一般流程如下：

现实世界→面向对象建模（类图，对象，方法）→面向对象语言→执行求解

### 9.2.3 面向对象的特点

面向对象技术充分体现了分解、抽象、模块化、信息隐藏等思想，有效提高软件生产率、缩短软件开发时间、提高软件质量，是控制复杂度的有效途径。

面向对象不仅适合普通人员，也适合经理人员。降低维护开销的技术可以释放管理者的资源，将其投入到待处理的应用中。在经理们看来，面向对象不是纯技术的，它既能给企业的组织也能给经理的工作带来变化。

当一个企业采纳了面向对象，其组织将发生变化。类的重用需要类库和类库管理人员，每个程序员都要加入到两个组中的一个：一个是设计和编写新类组，另一个是应用类创建新应用程序组。面向对象不太强调编程，需求分析相对地将变得更加重要。

面向对象编程主要有代码容易修改、代码复用性高、满足用户需求 3 个特点。

#### （1）代码容易修改

面向对象编程的代码都是封装在类里面，如果类的某个属性发生变化，只需要修改类中成员函数的实现即可，其他的程序函数不发生改变。如果类中属性变化较大，则使用继承的方法重新派生新类。

#### （2）代码复用性高

面向对象编程的类都是具有特定功能的封装，需要使用类中特定的功能，只需要声明该类并调用其成员函数即可。如果需要的功能在不同类，还可以进行多重继承，将不同类的成员封装到一个类中。功能的实现可以像积木一样随意组合，大大提高了代码的复用性。

#### （3）满足用户需求

由于面向对象编程的代码复用性高，用户的要求发生变化时，只需要修改发生变化的类。如果用户的要求变化较大时，就对类进行重新组装，将变化大的类重新开发，功能没有发生变化的类可以直接拿来使用。面向对象编程可以及时地响应用户需求的变化。

## 9.3 统一建模语言

 视频讲解：光盘\TM\lx\9\统一建模语言.exe

### 9.3.1 统一建模语言概述

模型是用某种工具对同类或其他工具的表达方式，是系统语义的完整抽象。模型可以分解为包的

层次结构，最外层的包对应于整个系统。模型的内容是从顶层包到模型元素的包所含关系的闭包。

模型可以用于捕获精确的表达项目的需求和应用领域中的知识，以使各方面的利益相关者能够相互理解并达成一致。

UML 是统一建模语言的英文缩写，它是一种直观化、明确化、构建和文档化软件系统产物的通用可视化建模语言。UML 记录了被构建系统的有关决定和理解，可用于对系统的理解、设计、浏览、配置以及信息控制。UML 的应用贯穿在系统开发的需求分析、分析、设计、构造、测试 5 个阶段，它包括概念的语义、表示法和说明，提供静态、动态、系统环境及组织结构的建模。建模语言是一种图形化的文档描述性语言，解决的核心问题是沟通障碍的问题，但 UML 是总结了以往建模技术的经验并吸收当今优秀成果的标准建模方法。

### 9.3.2 统一建模语言的结构

UML 由图和元模型共同组成，其中图是 UML 的语法，而元模型则是给出的图的意思，它是 UML 的语义。UML 的语义定义在一个 4 层抽象级建模概念框架中，这 4 层结构分别是：

#### ☑ 元介质模型层

该层描述基本的类型、属性、关系，这些元素都用于定义 UML 元模型。元介质模型强调用少数功能较强的模型成分来组合表达复杂的语义。每一个方法和技术都应在相对独立的抽象层次上。

#### ☑ 元模型层

该层组成了 UML 的基本元素，包括面向对象和面向组件的概念，这一层的每个概念都在元介质模型的“事物”的实例中。

#### ☑ 模型层

该层组成了 UML 的模型，这一层中的每个概念都是在元模型层中概念的一个实例，这一层的模型通常叫做类模型或类型模型。

#### ☑ 用户模型层

该层中的所有元素都是 UML 模型的例子。这一层中的每个概念都是模型层的一个实例，也是元模型层的一个实例。这一层的模型通常叫做对象模型或实例模型。

UML 使用模型来描述系统的结构或静态特征，以及行为或动态特征，它通过不同的视图来体现行为或动态特征。常用的视图有以下几种：

##### (1) 用例视图

该视图强调以用户的角度所看到的或需要的系统功能为出发点建模。这种视图有时也被称为用户模型视图。

##### (2) 逻辑视图

该视图用于展现系统的静态和结构组成及其特征，它也被称为结构模型视图或静态视图。

##### (3) 并发视图

该视图体现了系统的动态或者行为特征，它也被称为行为模型视图、过程视图、写作视图或者动态视图。

##### (4) 组件视图

该视图体现了系统实现的结构和行为特征，它有时也被称为模型实现视图。

### （5）开发视图

该视图体现了系统实现环境的结构和行为特征，它也被称为物理视图。

UML 的视图都是由一个或多个图共同组成。一个图体现一个系统架构的某个功能，所有的图一起组成了系统的完整视图。UML 提供了 9 种不同的图，分别是用例图、类图、对象图、组件图、配置图、序列图、写作图、状态图和活动图。

活动图如图 9.1 所示。

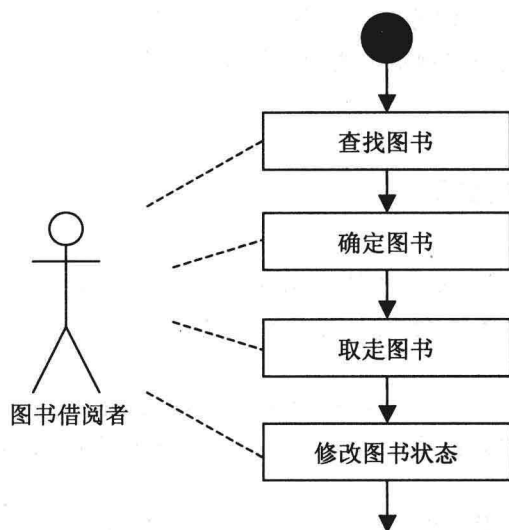


图 9.1 活动图

一个图书借阅者使用图书管理系统时要先进行查找图书动作，然后确定想要的书，接着是取走图书，最后查询和修改图书在系统中的状态。

UML 除提供了 9 种视图以外还提供了包图和交互图，包图如图 9.2 所示。

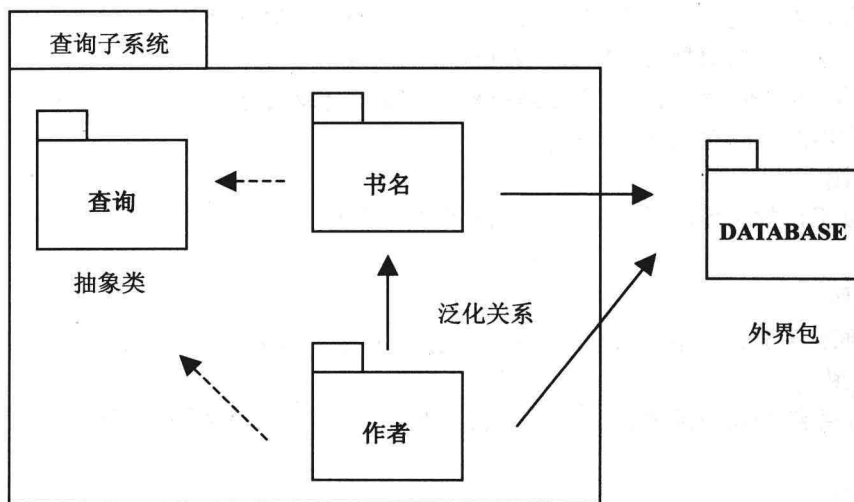


图 9.2 包图

图书管理系统中的查询模块，包括了查询抽象类包、通过书名查询包、通过作者查询包。通过书名查询包和通过作者查询包都派生于查询抽象类包，并且都调用其他子系统下的数据库包。

包图描述了类的结构，交互图则描述了类对象的交互步骤，交互图如图 9.3 所示。

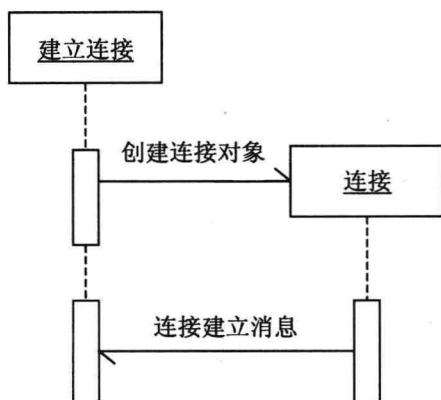


图 9.3 交互图

交互图中演示的是建立连接动作对象和连接对象的交互过程，首先发送“创建连接对象”消息，当连接对象创建完后，返回“连接建立消息”给建立连接动作对象。

### 9.3.3 面向对象的建模

面向对象的建模是一种新的思维方式，是一种关于计算机和信息结构化的新思维。面向对象的建模，把系统看作相互协作的对象，这些对象是结构和行为的封装，都属于某个类，那些类具有某种层次化的结构。系统的所有功能通过对象之间相互发送消息来获得。面向对象的建模可以视为一个包含以下元素的概念框架：抽象、封装、模块化、层次、分类、并行、稳定、可重用和可扩展性。

## 9.4 小 结

了解面向对象编程与面向过程编程的区别可以更好地理解面向对象编程，面向对象编程需要通过好的模型才能发挥其优点，而好的模型需要大量的代码积累和反复的测试才能形成。读者可以通过本章了解面向对象编程的思路以及使用 UML 来描述编程思路，掌握面向对象编程的方法。






# 第10章

---

## 类和对象

( 视频讲解：1 小时 1 分钟)

C++既可以开发面向过程的应用程序，也可以开发面向对象的应用程序。类是对象的实现，面向对象中的类是抽象概念，而类是程序开始过程中定义的一个对象，用类定义的对象可以是现实生活中的真实对象，也可以是从现实生活中抽象的对象。

通过阅读本章，您可以：

- » 了解对象的声明
- » 了解构造函数和析构函数
- » 了解类成员的使用
- » 掌握友元
- » 掌握命名空间

## 10.1 C++类

 视频讲解：光盘\TM\lx\10\C++类.exe

### 10.1.1 类概述

面向对象中的对象需要通过定义类来声明，对象一词是一种形象的说法，在编写代码过程中则是通过定义一个类来实现。

C++类不同于汉语中的类、分类、类型，它是一个特殊的概念，可以是对统一类型事物进行抽象处理，也可以是一个层次结构中的不同层次节点。例如将客观世界看成一个 Object 类，动物是客观世界中的一小部分，定义为 Animal 类，狗是一种哺乳动物，是动物的一类，定义为 Dog 类，鱼也是一种动物，定义为 Fish 类，则类的层次关系如图 10.1 所示。

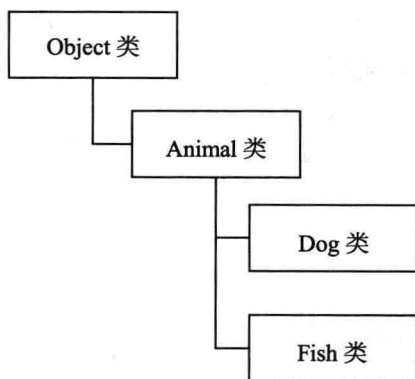


图 10.1 类的层次关系

类是一个新的数据类型，它和结构体有些相似，是由不同数据类型组成的集合体，但类要比结构体增加了操作数据的行为，这个行为就是函数。

### 10.1.2 类的声明与定义

在 10.1.1 节中已经对类的概念进行说明，可以看出类是用户自己指定的类型。如果程序中要用到类这种类型，就必须自己根据需要进行声明，或者使用别人设计好的类。下面来看一下如何设计一个类。

类的声明格式如下：

```
class 类名标识符
{
    [public:]
    [数据成员的声明]
```

```

[成员函数的声明]
[private:]
[数据成员的声明]
[成员函数的声明]
[protected:]
[数据成员的声明]
[成员函数的声明]
};

```

类的声明格式的说明如下：

- ☑ `class` 是定义类结构体的关键字，大括号内被称为类体或类空间。
- ☑ 类名标识符指定的就是类名，类名就是一个新的数据类型，通过类名可以声明对象。
- ☑ 类的成员有函数和数据两种类型。
- ☑ 大括号内是定义和声明类成员的地方，关键字 `public`、`private`、`protected` 是类成员访问的修饰符。

类中的数据成员的类型可以是任意的，包含整型、浮点型、字符型、数组、指针和引用等，也可以是对象。另一个类的对象可以作为该类的成员，但是自身类的对象不可以作为该类的成员，而自身类的指针或引用则可以作为该类的成员。



定义类结构体和定义结构体时大括号后要有分号。

例如，给出一个员工信息类声明：

```

class CPerson
{
    /*数据成员*/
    int m_iIndex;           //声明数据成员
    char m_cName[25];       //声明数据成员
    short m_shAge;          //声明数据成员
    double m_dSalary;       //声明数据成员
    /*成员函数*/
    short getAge();          //声明成员函数
    int setAge(short sAge)   //声明成员函数
    int getIndex();         //声明成员函数
    int setIndex(int iIndex); //声明成员函数
    char* getName();        //声明成员函数
    int setName(char cName[25]); //声明成员函数
    double getSalary();     //声明成员函数
    int setSalary(double dSalary); //声明成员函数
};

```

在代码中，`class` 关键字是用来定义类这种类型的，`CPerson` 是定义的员工信息类名称，在大括号中包含了 4 个数据成员分别表示 `CPerson` 类的属性，包含了 8 个成员函数表示 `CPerson` 类的行为。

### 10.1.3 类的实现

10.1.2 节只是在 CPerson 类中声明了类的成员。然而要使用这个类中的方法，即成员函数，还要对其定义具体的操作。下面来介绍如何定义类中的方法。

第一种方法是将类的成员函数都定义在类体内。

以下代码都在 person.h 头文件内，类的成员函数都定义在类体内。

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
class CPerson
{
public:
    //数据成员
    int m_iIndex;
    char m_cName[25];
    short m_shAge;
    double m_dSalary;
    //成员函数
    short getAge() { return m_shAge; }
    int setAge(short sAge)
    {
        m_shAge=sAge;
        return 0;
    }
    int getIndex() { return m_iIndex; }
    int setIndex(int iIndex)
    {
        m_iIndex=iIndex;
        return 0;
    }
    char* getName()
    { return m_cName; }
    int setName(char cName[25])
    {
        strcpy(m_cName,cName);
        return 0;
    }
    double getSalary() { return m_dSalary; }
    int setSalary(double dSalary)
    {
        m_dSalary=dSalary;
        return 0;
    }
};
```



第二种方法，也可以将类体内的成员函数的实现放在类体外，但如果类成员定义在类体外，需要用域运算符“::”，放在类体内和类体外的效果是一样的。

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
class CPerson
{
public:
    //数据成员
    int m_iIndex;
    char m_cName[25];
    short m_shAge;
    double m_dSalary;
    //成员函数
    short getAge();
    int setAge(short sAge);
    int getIndex() ;
    int setIndex(int iIndex);
    char* getName() ;
    int setName(char cName[25]);
    double getSalary() ;
    int setSalary(double dSalary);
};
//类成员函数的实现部分
short CPerson::getAge()
{
    return m_shAge;
}
int CPerson::setAge(short sAge)
{
    m_shAge=sAge;
    return 0;                //执行成功返回 0
}
int CPerson::getIndex()
{
    return m_iIndex;
}
int CPerson::setIndex(int iIndex)
{
    m_iIndex=iIndex;
    return 0;                //执行成功返回 0
}
char* CPerson::getName()
{
    return m_cName;
}
int CPerson::setName(char cName[25])
{

```



```

        strcpy(m_cName,cName);
        return 0;                                //执行成功返回 0
    }
    double CPerson::getSalary()
    {
        return m_dSalary;
    }
    int CPerson::setSalary(double dSalary)
    {
        m_dSalary=dSalary;
        return 0;                                //执行成功返回 0
    }

```

前面两种方式都是将代码存储在同一个文件内。C++语言可以实现将函数的声明和函数的定义放在不同的文件内，一般在头文件放入函数的声明，在实现文件放入函数的实现。同样可以将类的定义放在头文件中，将类成员函数的实现放在实现文件内。存放类的头文件和实现文件最好和类名相同或相似。例如将 CPerson 类的声明部分放在 person.h 文件内，代码如下：

```

#include<stdio.h>
#include<stdlib.h>
#include<string.h>
class CPerson
{
public:
    //数据成员
    int m_iIndex;
    char m_cName[25];
    short m_shAge;
    double m_dSalary;
    //成员函数
    short getAge();
    int setAge(short sAge);
    int getIndex();
    int setIndex(int iIndex);
    char* getName();
    int setName(char cName[25]);
    double getSalary();
    int setSalary(double dSalary);
};

```

将 CPerson 类的实现部分放在 person.cpp 文件内，代码如下：

```

#include "person.h"
//类成员函数的实现部分
short CPerson::getAge()
{
    return m_shAge;
}
int CPerson::setAge(short sAge)

```

```

{
    m_shAge=sAge;
    return 0;                                //执行成功返回 0
}
int CPerson::getIndex()
{
    return m_iIndex;
}
int CPerson::setIndex(int iIndex)
{
    m_iIndex=iIndex;
    return 0;                                //执行成功返回 0
}
char* CPerson::getName()
{
    return m_cName;
}
int CPerson::setName(char cName[25])
{
    strcpy(m_cName,cName);
    return 0;                                //执行成功返回 0
}
double CPerson::getSalary()
{
    return m_dSalary;
}
int CPerson::setSalary(double dSalary)
{
    m_dSalary=dSalary;
    return 0;                                //执行成功返回 0
}

```

此时整个工程所有文件如图 10.2 所示。

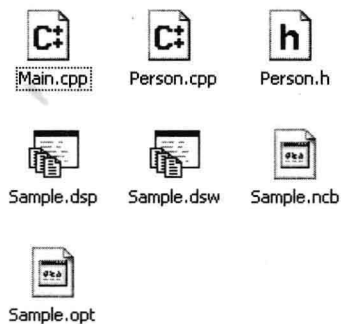


图 10.2 所有工程文件

关于类的实现有两点说明：

(1) 类的数据成员需要初始化，成员函数还要添加实现代码。类的数据成员不可以在类的声明中初始化。例如：

```

class CPerson
{
    //数据成员
    int m_iIndex=1;           //错误写法，不应该初始化的
    char m_cName[25]="Mary";  //错误写法，不应该初始化的
    short m_shAge=22;         //错误写法，不应该初始化的
    double m_dSalary=1700.00; //错误写法，不应该初始化的
    //成员函数
    short getAge();
    int setAge(short sAge)
    int getIndex();
    int setIndex(int iIndex);
    char* getName();
    int setName(char cName[25]);
    double getSalary();
    int setSalary(double dSalary);
};

```

上面代码是不能通过编译的。

(2) 空类是 C++中最简单的类，其声明方式如下：

```
class CPerson{ };
```

空类只是起到占位的作用，在需要时再定义类成员及实现。

### 10.1.4 对象的声明

定义一个新类后，就可以通过类名来声明一个对象。声明的形式如下：

**类名 对象名表**

类名是定义好的新类的标识符，对象名表中是一个或多个对象的名称，如果声明的是多个对象就用逗号运算符分隔。

例如声明一个对象如下：

```
CPerson p;
```

声明多个对象如下：

```
CPerson p1,p2,p3;
```

声明完对象就是对象的引用了，对象的引用有两种方式，一种是成员引用方式，一种是对象指针方式。

(1) 成员引用方式

成员变量引用的表示如下：

**对象名.成员名**

这里“.”是一个运算符，该运算符的功能是表示对象的成员。

成员函数引用的表示如下：

对象名. 成员名(参数表)

例如：

```
CPerson p;
p.m_iIndex;
```

## (2) 对象指针方式

对象声明形式中的对象名表，除了是用逗号运算符分隔的多个对象名外，还可以是对象名数组、对象名指针和引用形式的对象名。

声明一个对象指针：

```
CPerson *p;
```

但要想使用对象的成员，需要“->”运算符，它是表示成员的运算符，与“.”运算符的意义相同。“->”用来表示对象指针所指的成员，对象指针就是指向对象的指针，例如：

```
CPerson *p;
p->m_iIndex;
```

下面的对象数据成员的两种表示形式是等价的：

对象指针名->数据成员

与

(\*对象指针名).数据成员

同样，下面成员函数的两种表示形式是等价的：

对象指针名->成员名(参数表)

与

(\*对象指针名).成员名(参数表)

例如：

```
CPerson *p;
(*p).m_iIndex;           //对类中的成员进行引用
p->m_iIndex;              //对类中的成员进行引用
```

### 【例 10.1】 对象的引用。（实例位置：光盘\TM\sl\10\1）

在本实例中，利用前文声明的类定义对象，然后使用该对象引用其中成员。

```
#include<iostream.h>
#include "Person.h"
void main()
{
    int iResult=-1;
    CPerson p;
```



```

iResult=p.setAge(25);
if(iResult>=0)
    cout << "m_shAge is:" << p.getAge() << endl;

iResult=p.setIndex(0);
if(iResult>=0)
    cout << "m_iIndex is:" << p.getIndex() << endl;

char bufTemp[]="Mary";
iResult=p.setName(bufTemp);
if(iResult>=0)
    cout << "m_cName is:" << p.getName() << endl;

iResult=p.setSalary(1700.25);
if(iResult>=0)
    cout << "m_dSalary is:" << p.getSalary() << endl;
}

```

在实例中可以看到，首先使用 CPerson 类定义对象 p，然后使用 p 引用类中的成员函数。

p.setAge(25) 引用类中的 setAge 成员函数，将参数中的数据赋值给数据成员，设置对象的属性。函数的返回值赋给 iResult 变量，通过 iResult 变量值判断函数 setAge 为数据成员赋值是否成功。如果成功再使用 p.getAge 函数得到赋值的数据，然后将其输出显示。

之后使用对象 p 依次引用成员函数 setIndex、setName 和 setSalary，然后通过对 iResult 变量的判断，决定是否引用成员函数 getIndex、getName 和 getSalary。

## 10.2 构造函数

 视频讲解：光盘\TM\lx\10\构造函数.exe

### 10.2.1 构造函数概述

在类的实例进入其作用域时，也就是建立一个对象，构造函数就会被调用，那么构造函数的作用是什么呢？当建立一个对象时，常常需要做某些初始化的工作，例如对数据成员进行赋值设置类的属性，而这些操作刚好放在构造函数中完成。

前文介绍过结构的相关知识，在对结构进行初始化时，可以使用下面的方法，例如：

```

struct PersonInfo
{
    int index;
    char name[30];
    short age;
};

```

```
void InitStruct()
{
    PersonInfo p={1,"mr",22};
}
```

但是类不能像结构体一样初始化，其构造方法如下：

```
class CPerson
{
public:
    CPerson();           //构造函数
    int m_iIndex;
    int getIndex();
};
//构造函数
CPerson::CPerson()
{
    m_iIndex=10;
}
```

CPerson 是默认构造函数，如果不显式的写上函数的声明也可以。

构造函数是可以有参数的，修改上面的代码，使其构造函数带有参数，例如：

```
class CPerson
{
public:
    CPerson(int iIndex);   //构造函数
    int m_iIndex;
    int setIndex(int iIndex);
};
//构造函数
CPerson::CPerson(int iIndex)
{
    m_iIndex= iIndex;
}
```

**【例 10.2】** 使用构造函数进行初始化操作。（实例位置：光盘\TM\sl\10\2）

```
#include<iostream>
using namespace std;
//定义 CPerson 类
class CPerson
{
public:
    CPerson();
    CPerson(int iIndex,short m_shAge,double m_dSalary);
    int m_iIndex;
    short m_shAge;
    double m_dSalary;
```



```

    int getIndex();
    short getAge();
    double getSalary();
};
//在默认构造函数中初始化
CPerson::CPerson()
{
    m_iIndex=0;
    m_shAge=10;
    m_dSalary=1000;
}
//在带参数的构造函数中初始化
CPerson::CPerson(int iIndex,short m_shAge,double m_dSalary)
{
    m_iIndex=iIndex;
    m_shAge=m_shAge;
    m_dSalary=m_dSalary;
}
int CPerson::getIndex()
{
    return m_iIndex;
}
//在 main 函数中输出类的成员值
void main()
{
    CPerson p1;
    cout << "m_iIndex is:" << p1.getIndex() << endl;

    CPerson p2(1,20,1000);
    cout << "m_iIndex is:" << p2.getIndex() << endl;
}

```

程序运行结果如图 10.3 所示。

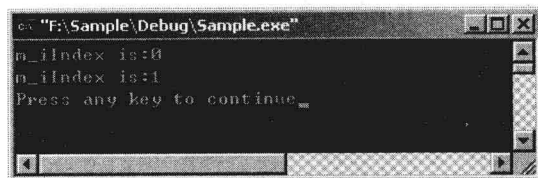


图 10.3 使用构造函数进行初始化操作

程序声明了两个对象 p1 和 p2，p1 使用默认构造函数初始化成员变量，p2 使用带参数的构造函数初始化，所以在调用同一个类成员函数 getIndex 时输出结果不同。

## 10.2.2 复制构造函数

在开发程序时可能需要保存对象的副本，以便在后面执行的过程中恢复对象的状态。那么如何用

一个已经初始化的对象来新生成一个一模一样的对象？答案是使用复制构造函数来实现。复制构造函数就是函数的参数是一个已经初始化的类对象。

**【例 10.3】** 使用复制构造函数。（实例位置：光盘\TM\sl\10\3）

在头文件 Person.h 中声明和定义类，代码如下：

```
class CPerson
{
public:
    CPerson(int iIndex,short shAge,double dSalary);    //构造函数
    CPerson(CPerson & copyPerson);                    //复制构造函数
    int m_iIndex;
    short m_shAge;
    double m_dSalary;
    int getIndex();
    short getAge();
    double getSalary();
};
//构造函数
CPerson::CPerson(int iIndex,short shAge,double dSalary)
{
    m_iIndex=iIndex;
    m_shAge=shAge;
    m_dSalary=dSalary;
}
//复制构造函数
CPerson::CPerson(CPerson & copyPerson)
{
    m_iIndex=copyPerson.m_iIndex;
    m_shAge=copyPerson.m_shAge;
    m_dSalary=copyPerson.m_dSalary;
}
short CPerson::getAge()
{
    return m_shAge;
}
int CPerson::getIndex()
{
    return m_iIndex;
}
double CPerson::getSalary()
{
    return m_dSalary;
}
```

在主程序文件中实现类对象的调用，代码如下：

```
#include<iostream>
#include "Person.h"
using namespace std;
```

```

void main()
{
    CPerson p1(20,30,100);
    CPerson p2(p1);
    cout << "m_iIndex of p1 is:" << p2.getIndex() << endl;
    cout << "m_shAge of p1 is:" << p2.getAge() << endl;
    cout << "m_dSalary of p1 is:" << p2.getSalary() << endl;
    cout << "m_iIndex of p2 is:" << p2.getIndex() << endl;
    cout << "m_shAge of p2 is:" << p2.getAge() << endl;
    cout << "m_dSalary of p2 is:" << p2.getSalary() << endl;
}

```

程序运行结果如图 10.4 所示。

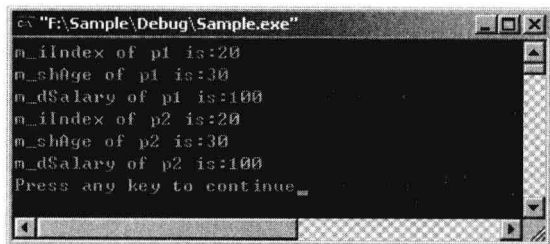


图 10.4 使用复制构造函数

程序中先用带参数的构造函数声明对象 p1，然后通过复制构造函数声明对象 p2，因为 p1 已经是初始化完成的类对象，可以作为复制构造函数的参数。通过输出结果可以看出，两个对象是相同的。

## 10.3 析构函数

### 视频讲解：光盘\TM\lx\10\析构函数.exe

构造函数和析构函数是类体定义中比较特殊的两个成员函数，因为它们两个都没有返回值，而且构造函数名标识符和类名标识符相同，析构函数名标识符就是在类名标识符前面加“~”符号。

构造函数主要是用来在对象创建时，给对象中的一些数据成员赋值，主要目的就是来初始化对象。析构函数的功能是用来释放一个对象的，在对象删除前，用它来做一些清理工作，它与构造函数的功能正好相反。

**【例 10.4】** 使用析构函数。（实例位置：光盘\TM\sl\10\4）

在头文件 Person.h 中声明和定义类，代码如下：

```

#include<iostream>
#include<string.h>
using namespace std;
class CPerson
{
public:
    CPerson();

```



```

~CPerson();           //析构函数
char* m_pMessage;
void ShowStartMessage();
void ShowFrameMessage();
};
CPerson::CPerson()
{
    m_pMessage = new char[2048];
}
void CPerson::ShowStartMessage()
{
    strcpy(m_pMessage, "Welcome to MR");
    cout << m_pMessage << endl;
}
void CPerson::ShowFrameMessage()
{
    strcpy(m_pMessage, "*****");
    cout << m_pMessage << endl;
}
CPerson::~~CPerson()
{
    delete[] m_pMessage;
}

```

在主程序文件中实现类对象的调用，代码如下：

```

#include<iostream>
using namespace std;
#include "Person.h"
void main()
{
    CPerson p;
    p.ShowFrameMessage();
    p.ShowStartMessage();
    p.ShowFrameMessage();
}

```

程序运行结果如图 10.5 所示。

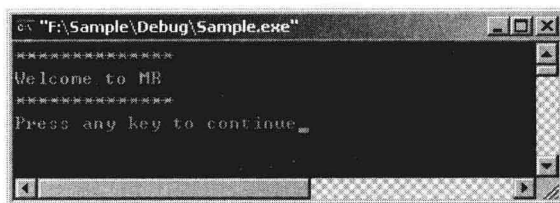


图 10.5 使用析构函数

程序在构造函数中使用 `new` 为成员 `m_pMessage` 分配空间，在析构函数中使用 `delete` 释放由 `new` 分配的空间。成员 `m_pMessage` 为字符指针，在 `ShowStartMessage` 成员函数中输出字符指针所指向的

内容。

使用析构函数的注意事项如下：

- ☑ 一个类中只可能定义一个析构函数。
- ☑ 析构函数不能重载。
- ☑ 构造函数和析构函数不能使用 `return` 语句返回值，不用加上关键字 `void`。

构造函数和析构函数的调用环境：

(1) 自动变量的作用域是某个模块，当此模块被激活时，自动变量调用构造函数，当退出此模块时，会调用析构函数。

(2) 全局变量在进入 `main` 函数之前会调用构造函数，在程序终止时会调用析构函数。

(3) 动态分配的对象在使用 `new` 为对象分配内存时会调用构造函数；使用 `delete` 删除对象时会调用析构函数。

(4) 临时变量是为支持计算，由编译器自动产生的。临时变量的生存期的开始和结尾会调用构造函数和析构函数。

## 10.4 类 成 员

📺 视频讲解：光盘\TM\lx\10\类成员.exe

### 10.4.1 访问类成员

类的三大特点之一就是具有封装性，封装在类里面的数据可以设置成对外可见或不可见。通过关键字 `public`、`private`、`protected` 可以设置类中数据成员对外是否可见，也就是其他类是否可以访问该数据成员。

关键字 `public`、`private`、`protected` 说明类成员是共有的、私有的，还是保护的。这 3 个关键字将类划分为 3 个区域，在 `public` 区域的类成员可以在类作用域外被访问，而 `private` 区域和 `protected` 区域的类成员只能在类作用域内被访问，如图 10.6 所示。

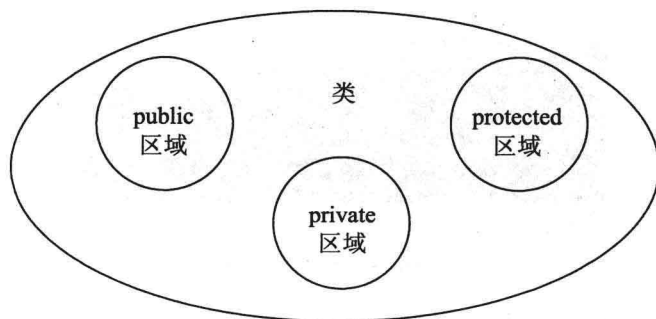


图 10.6 类成员属性

这3种类成员的属性如下：

- ☑ `public` 属性的成员对外可见，对内可见。
- ☑ `private` 属性的成员对外不可见，对内可见。
- ☑ `protected` 属性的成员对外不可见，对内可见，且对派生类是可见的。

如果在类定义时没有加任何关键字，默认状态类成员都在 `private` 区域。

例如在头文件 `person.h` 中：

```
class CPerson
{
    int m_iIndex;
    int getIndex() { return m_iIndex; }
    int setIndex(int iIndex)
    {
        m_iIndex=iIndex;
        return 0;                //执行成功返回 0
    }
};
```

实现文件 `person.cpp` 中：

```
#include<iostream.h>
#include "Person.h"
void main()
{
    CPerson p;
    p.m_iIndex=100;                //错误
    cout << "m_iIndex is:" << p.getIndex() << endl; //错误
}
```

在编译上面的代码时，会发现编译不能通过，这是什么原因呢？

因为在默认状态下，类成员的属性为 `private`，这样类成员只能被类中的其他成员访问，而不能被外部访问。例如，`CPerson` 类中的 `m_iIndex` 数据成员，只能在类体的作用域内被访问和赋值，数据类型为 `CPerson` 类的对象 `p`，就无法对 `m_iIndex` 数据成员进行赋值。

有了不同区域，开发人员可以根据需求来进行封装。将不想让其他类访问和调用的类成员定义在 `private` 区域和 `protected` 区域，这就保证了类成员的隐蔽性。需要注意的是，如果将成员的属性设置为 `protected`，那么继承类也可以访问父类的保护成员，但是不能访问类中的私有成员。

关键字的作用范围是，直到下一次出现另一个关键字为止。例如：

```
class CPerson
{
private:
    int m_iIndex;                //私有属性成员
public:
    int getIndex() { return m_iIndex; }    //公有属性成员
    int setIndex(int iIndex)              //公有属性成员
    {
        ...
    }
};
```



```

        m_iIndex=iIndex;
        return 0;                                //执行成功返回 0
    }
};

```

在上面的代码中，`private` 访问权限控制符设置 `m_iIndex` 成员变量为私有，`public` 关键字下面的成员函数设置为公有，从中可以看出 `private` 的作用域到 `public` 出现时为止。

## 10.4.2 内联成员函数

在定义函数时，可以使用 `inline` 关键字将函数定义为内联函数。在定义类的成员函数时，也可以使用 `inline` 关键字将成员函数定义为内联成员函数。其实，对于成员函数来说，如果其定义是在类体中，即使没有使用 `inline` 关键字，该成员函数也被认为是内联成员函数。例如：

```

class CUser                                //定义一个 CUser 类
{
private:
    char m_Username[128];                  //定义数据成员
    char m_Password[128];
public:
    inline char* GetUsername()const;        //定义一个内联成员函数
};
char* CUser::GetUsername()const            //实现内联成员函数
{
    return (char*)m_Username;
}

```

程序中，使用 `inline` 关键字将类中的成员函数设置为内联成员函数。此外，也可以在类成员函数的实现部分使用 `inline` 关键字标识函数为内联成员函数。例如：

```

class CUser                                //定义一个 CUser 类
{
private:
    char m_Username[128];                  //定义数据成员
    char m_Password[128];
public:
    char* GetUsername()const;              //定义成员函数
};
inline char* CUser::GetUsername()const      //函数为内联成员函数
{
    return (char*)m_Username;              //设置返回值
}

```

程序中的代码演示了在何处使用关键字 `inline`。对于内联函数来说，程序会在函数调用的地方直接插入函数代码，如果函数体语句较多，则会导致程序代码膨胀。如果将类的析构函数定义为内联函数，可能会导致潜在的代码膨胀。

### 10.4.3 静态类成员

本节之前所定义类成员，都是通过对象来访问的，不能通过类名直接访问。如果将类成员定义为静态类成员，则允许使用类名直接访问。静态类成员是在类成员定义前使用 `static` 关键字标识。例如：

```
class CBook
{
public:
    static unsigned int m_Price;           //定义一个静态数据成员
};
```

在定义静态数据成员时，通常需要在类体外部对静态数据成员进行初始化。例如：

```
unsigned int CBook::m_Price = 10;         //初始化静态数据成员
```

对于静态成员来说，不仅可以通过对象访问，还可以直接使用类名访问。例如：

```
int main(int argc, char* argv[])
{
    CBook book;                          //定义一个 CBook 类对象 book
    cout << CBook::m_Price << endl;      //通过类名访问静态成员
    cout << book.m_Price << endl;        //通过对象访问静态成员
    return 0;
}
```

在一个类中，静态数据成员是被所有的类对象所共享的，这就意味着无论定义多少个类对象，类的静态数据成员只有一份，同时，如果某一个对象修改了静态数据成员，其他对象的静态数据成员（实际上是同一个静态数据成员）也将改变。

对于静态数据成员，还需要注意以下几点。

☒ 静态数据成员可以是当前类的类型，而其他数据成员只能是当前类的指针或应用类型。

在定义类成员时，对于静态数据成员，其类型可以是当前类的类型，而非静态数据成员则不可以，除非数据成员的类型为当前类的指针或引用类型。例如：

```
class CBook
{
public:
    static unsigned int m_Price ;
    CBook m_Book;                          //非法的定义，不允许在该类中定义所属类的对象
    static CBook m_VCbook;                 //正确，静态数据成员允许定义类的所属类对象
    CBook *m_pBook;                        //正确，允许定义类的所属类型的指针类型对象
};
```

☒ 静态数据成员可以作为成员函数的默认参数。

在定义类的成员函数时，可以为成员函数指定默认参数，其参数的默认值也可以是类的静态数据成员，但是不同的数据成员则不能作为成员函数的默认参数。例如：

```

class CBook                                     //定义 CBook 类
{
public:
    static unsigned int m_Price ;               //定义一个静态数据成员
    int m_Pages;                               //定义一个普通数据成员
    void OutputInfo(int data = m_Price)         //定义一个函数，以静态数据成员作为默认参数
    {
        cout << data << endl;                 //输出信息
    }
    void OutputPage(int page = m_Pages)         //错误的定义，类的普通数据成员不能作为默认参数
    {
        cout << page << endl;                 //输出信息
    }
};

```

在介绍完类的静态数据成员之后，下面介绍类的静态成员函数。定义类的静态成员函数与定义普通的成员函数类似，只是在成员函数前添加 `static` 关键字。例如：

```

static void OutputInfo();                       //定义类的静态成员函数

```

类的静态成员函数只能访问类的静态数据成员，而不能访问普通的数据成员。例如：

```

class CBook                                     //定义一个类 CBook
{
public:
    static unsigned int m_Price ;               //定义一个静态数据成员
    int m_Pages;                               //定义一个普通数据成员
    static void OutputInfo()                   //定义一个静态成员函数
    {
        cout << m_Price << endl;              //正确的访问
        cout << m_Pages << endl;              //非法的访问，不能访问非静态数据成员
    }
};

```

在上述代码中，语句“`cout << m_Pages << endl;`”是错误的，因为 `m_Pages` 是非静态数据成员，不能在静态成员函数中访问。

此外，对于静态成员函数不能定义为 `const` 成员函数，即静态成员函数末尾不能使用 `const` 关键字。例如，下面的静态成员函数的定义是非法的。

```

static void OutputInfo()const;                  //错误的定义，静态成员函数末尾不能使用 const 关键字

```

在定义静态数据成员函数时，如果函数的实现代码处于类体之外，则在函数的实现部分不能再标识 `static` 关键字。例如，下面的函数定义是非法的。

```

static void CBook::OutputInfo()                 //错误的函数定义，不能使用 static 关键字
{
    cout << m_Price << endl;                 //输出信息
}

```



上述代码如果去掉 static 关键字则是正确的。例如：

```
void CBook::OutputInfo()           //正确的函数定义
{
    cout << m_Price<< endl;       //输出信息
}
```

#### 10.4.4 隐藏的 this 指针

对于类的非静态成员，每一个对象都有自己的一份拷贝，即每个对象都有自己的数据成员，不过成员函数却是每个对象共享的。那么调用共享的成员函数是如何找到自己的数据成员的呢？答案就是通过类中隐藏的 this 指针。下面通过对实例的讲解来说明 this 指针的作用。

例如，每一个对象都有自己的一份拷贝：

```
class CBook                       //定义一个 CBook 类
{
public:
    int m_Pages;                  //定义一个数据成员
    void OutputPages()            //定义一个成员函数
    {
        cout<<m_Pages<<endl;     //输出信息
    }
};

int main(int argc, char* argv[])
{
    CBook vbBook,vcBook;          //定义两个 CBook 类对象
    vbBook.m_Pages = 512;         //设置 vbBook 对象的成员数据
    vcBook.m_Pages = 570;         //设置 vcBook 对象的成员数据
    vbBook.OutputPages();          //调用 OutputPages 方法输出 vbBook 对象的数据成员
    vcBook.OutputPages();          //调用 OutputPages 方法输出 vcBook 对象的数据成员
    return 0;
}
```

程序运行结果如图 10.7 所示。

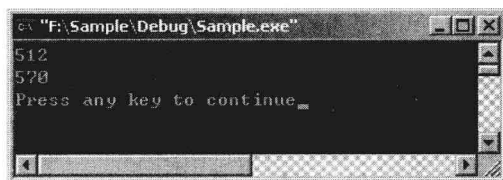


图 10.7 访问对象的数据成员

从图 10.7 中可以发现，vbBook 和 vcBook 两个对象均有自己的数据成员 m\_Pages，在调用 OutputPages 成员函数时输出的均是自己的数据成员。在 OutputPages 成员函数中只是访问了 m\_Pages 数据成员，每个对象在调用 OutputPages 方法时是如何区分自己的数据成员呢？答案是通过 this 指针。在每个类的成员函数（非静态成员函数）中都隐含包含一个 this 指针，指向被调用对象的指针，其类

型为当前类类型的指针类型，在 `const` 方法中，为当前类类型的 `const` 指针类型。当 `vbBook` 对象调用 `OutputPages` 成员函数时，`this` 指针指向 `vbBook` 对象，当 `vcBook` 对象调用 `OutputPages` 成员函数时，`this` 指针指向 `vcBook` 对象。在 `OutputPages` 成员函数中，用户可以显式地使用 `this` 指针访问数据成员。例如：

```
void OutputPages()
{
    cout <<this->m_Pages<<endl;           //使用 this 指针访问数据成员
}
```

实际上，编译器为了实现 `this` 指针，在成员函数中自动添加了 `this` 指针对数据成员的方法，类似于上面的 `OutputPages` 方法。此外，为了将 `this` 指针指向当前调用对象，并在成员函数中能够使用，在每个成员函数中都隐含包含一个 `this` 指针作为函数参数，并在函数调用时将对象自身的地址隐含作为实际参数传递。例如，以 `OutputPages` 成员函数为例，编译器将其定义为：

```
void OutputPages(CBook* this)           //隐含添加 this 指针
{
    cout <<this->m_Pages<<endl;
}
```

在对象调用成员函数时，传递对象的地址到成员函数中。以 “`vc.OutputPages();`” 语句为例，编译器将其解释为 “`vbBook.OutputPages(&vbBook);`”，这就使得 `this` 指针合法，并能够在成员函数中使用。

### 10.4.5 嵌套类

C++语言允许在一个类中定义另一个类，这被称之为嵌套类。例如，下面的代码在定义 `CList` 类时，在内部又定义了一个嵌套类 `CNode`。

```
#define MAXLEN 128                               //定义一个宏
class CList                                       //定义 CList 类
{
public:                                           //嵌套类为公有的
    class CNode                                  //定义嵌套类 CNode
    {
        friend class CList;                     //将 CList 类作为自己的友元类
    private:
        int m_Tag;                               //定义私有成员
    public:
        char m_Name[MAXLEN];                     //定义共有数据成员
    };                                           //CNode 类定义结束
public:
    CNode m_Node;                                //定义一个 CNode 类型数据成员
    void SetNodeName(const char *pchData)         //定义成员函数
    {
        if(pchData != NULL)                     //判断指针是否为空
        {
            strcpy(m_Node.m_Name,pchData);       //访问 CNode 类的共有数据
        }
    }
}
```



```

    }
}
void SetNodeTag(int tag)           //定义成员函数
{
    m_Node.m_Tag = tag;           //访问 CNode 类的私有数据
}
};

```

上述代码在嵌套类 CNode 中定义了一个私有成员 m\_Tag，定义了一个公有成员 m\_Name，对于外围类 CList 来说，通常它不能够访问嵌套类的私有成员，虽然嵌套类是在其内部定义的。但是，上述代码在定义 CNode 类时将 CList 类作为自己的友元类，这使得 CList 类能够访问 CNode 类的私有成员。

对于内部的嵌套类来说，只允许其在外围的类域中使用，在其他类域或者作用域中是不可见的。例如下面的定义是非法的。

```

int main(int argc, char* argv[])
{
    CNode node;                   //错误的定义，不能访问 CNode 类
    return 0;
}

```

上述代码在 main 函数的作用域中定义了一个 CNode 对象，导致 CNode 没有被声明的错误。对于 main 函数来说，嵌套类 CNode 是不可见的，但是可以通过使用外围的类域作为限定符来定义 CNode 对象。如下的定义将是合法的。

```

int main(int argc, char* argv[])
{
    CList::CNode node;           //合法的定义
    return 0;
}

```

上述代码通过使用外围类域作为限定符访问到了 CNode 类。但是这样做通常是不合理的，也是有限制条件的。因为既然定义了嵌套类，通常都不允许在外界访问，这违背了使用嵌套类的原则。其次，在定义嵌套类时，如果将其定义为私有的或受保护的，即使使用外围类域作为限定符，外界也无法访问嵌套类。

### 10.4.6 局部类

类的定义可以放在头文件中，也可以放在源文件中。还有一种情况，类的定义也可以放置在函数中，这样的类被称之为局部类。

例如，定义一个局部类 CBook：

```

void LocalClass()                //定义一个函数
{
    class CBook                  //定义一个局部类 CBook
    {

```

```

private:
    int m_Pages;                //定义一个私有数据成员
public:
    void SetPages(int page)      //定义公有成员函数
    {
        if(m_Pages != page)
            m_Pages = page;      //为数据成员赋值
    }
    int GetPages()               //定义公有成员函数
    {
        return m_Pages;         //获取数据成员信息
    }
};
CBook book;                    //定义一个 CBook 对象
book.SetPages(300);             //调用 SetPages 方法
cout << book.GetPages()<< endl; //输出信息
}

```

上述代码在 LocalClass 函数中定义了一个类 CBook，该类被称为局部类。对于局部类 CBook，在函数之外是不能够被访问的，因为局部类被封装在了函数的局部作用域中。

## 10.5 友元

 视频讲解：光盘\TM\10\友元.exe

### 10.5.1 友元概述

在讲述类的内容时说明了隐藏数据成员的好处，但是有些时候，类会允许一些特殊的函数直接读写其私有数据成员。

使用 `friend` 关键字可以让特定的函数或者别的类的所有成员函数对私有数据成员进行读写。这既可以保持数据的私有性，又能够使特定的类或函数直接访问私有数据。

有时候，普通函数需要直接访问一个类的保护或私有数据成员。如果没有友元机制，则只能将类的数据成员声明为公共的，从而任何函数都可以无约束地访问它。

普通函数需要直接访问类的保护或私有数据成员的原因主要是为提高效率。

例如，没使用友元函数的情况如下：

```

#include<iostream.h>
class CRectangle
{
public:
    CRectangle()
    {

```

```

        m_iHeight=0;
        m_iWidth=0;
    }
    CRectangle(int iLeftTop_x,int iLeftTop_y,int iRightBottom_x,int iRightBottom_y)
    {
        m_iHeight=iRightBottom_y-iLeftTop_y;
        m_iWidth=iRightBottom_x-iLeftTop_x;
    }

    int getHeight()
    {
        return m_iHeight;
    }
    int getWidth()
    {
        return m_iWidth;
    }
protected:
    int m_iHeight;
    int m_iWidth;
};
int ComputerRectArea(CRectangle & myRect)           //不是友元函数的定义
{
    return myRect.getHeight()*myRect.getWidth();
}
void main()
{
    CRectangle rg(0,0,100,100);
    cout << "Result of ComputerRectArea is :"<< ComputerRectArea(rg) << endl;
}

```

在代码中可以看到，ComputerRectArea 函数在定义时只能对类中的函数进行引用，因为类中的函数属性都为公有属性，对外是可见的，但是数据成员的属性为受保护属性，对外是不可见的，所以只能使用公有成员函数得到想要的值。

下面来介绍使用友元函数的情况。例如：

```

#include<iostream.h>
class CRectangle
{
public:
    CRectangle()
    {
        m_iHeight=0;
        m_iWidth=0;
    }
    CRectangle(int iLeftTop_x,int iLeftTop_y,int iRightBottom_x,int iRightBottom_y)
    {
        m_iHeight=iRightBottom_y-iLeftTop_y;
        m_iWidth=iRightBottom_x-iLeftTop_x;
    }
};

```



```

    }
    int getHeight()
    {
        return m_iHeight;
    }
    int getWidth()
    {
        return m_iWidth;
    }
    friend int ComputerRectArea(CRectangle & myRect);    //声明为友元函数
protected:
    int m_iHeight;
    int m_iWidth;
};
int ComputerRectArea(CRectangle & myRect)                //友元函数的定义
{
    return myRect.m_iHeight*myRect.m_iWidth;
}
void main()
{
    CRectangle rg(0,0,100,100);
    cout << "Result of ComputerRectArea is :"<< ComputerRectArea(rg) << endl;
}

```

在 ComputerRectArea 函数的定义中可以看到使用 CRectangle 的对象可以直接引用其中的数据成员，这是因为在 CRectangle 类中将 ComputerRectArea 函数声明为友元了。

从中可以看到使用友元保持了 CRectangle 类中数据的私有性，起到了隐藏数据成员的好处，又使得特定的类或函数可以直接访问这些隐藏数据成员。

## 10.5.2 友元类

对于类的私有方法，只有在该类中允许访问，其他类是不能访问的。但在开发程序时，如果两个类的耦合度比较紧密，能够在一个类中访问另一个类的私有成员会带来很大的方便。C++语言提供了友元类和友元方法（或者称为友元函数）来实现访问其他类的私有成员。当用户希望另一个类能够访问当前类的私有成员时，可以在当前类中将另一个类作为自己的友元类，这样在另一个类中就可以访问当前类的私有成员了。例如定义友元类：

```

class CItem                                            //定义一个 CItem 类
{
private:
    char m_Name[128];                                //定义私有的数据成员
    void OutputName()                                //定义私有的成员函数
    {
        printf("%s\n",m_Name);                      //输出 m_Name
    }
}

```

```

public:
    friend class CList;           //将 CList 类作为自己的友元类
    void SetItemName(const char* pchData) //定义共有成员函数，设置 m_Name 成员
    {
        if(pchData != NULL)      //判断指针是否为空
        {
            strcpy(m_Name, pchData); //赋值字符串
        }
    }
    CItem()                       //构造函数
    {
        memset(m_Name, 0, 128);    //初始化数据成员 m_Name
    }
};
class CList                      //定义类 CList
{
private:
    CItem m_Item;                //定义私有的数据成员 m_Item
public:
    void OutputItem();           //定义共有成员函数
};
void CList::OutputItem()         //OutputItem 函数的实现代码
{
    m_Item.SetItemName("BeiJing"); //调用 CItem 类的共有方法
    m_Item.OutputName();           //调用 CItem 类的私有方法
}

```

在定义 CItem 类时，使用 friend 关键字将 CList 类定义为 CItem 类的友元，这样 CList 类中的所有方法都可以访问 CItem 类中的私有成员了。在 CList 类的 OutputItem 方法中，语句“m\_Item.OutputName()”演示了调用 CItem 类的私有方法 OutputName。

### 10.5.3 友元方法

在开发程序时，有时需要控制另一个类对当前类的私有成员的方法。例如，假设需要实现只允许 CList 类的某个成员访问 CItem 类的私有成员，而不允许其他成员函数访问 CItem 类的私有数据，便可以通过定义友元函数来实现。在定义 CItem 类时，可以将 CList 类的某个方法定义为友元方法，这样就限制了只有该方法允许访问 CItem 类的私有成员。

【例 10.5】 定义友元方法。（实例位置：光盘\TM\sl\10\5）

```

class CItem;                    //前导声明 CItem 类
class CList                     //定义 CList 类
{
private:
    CItem * m_pItem;           //定义私有数据成员 m_pItem
public:
    CList();                   //定义默认构造函数
}

```



```

    ~CList();                //定义析构函数
    void OutputItem();        //定义 OutputItem 成员函数
};
class CItem                  //定义 CItem 类
{
    friend void CList::OutputItem(); //声明友元函数
private:
    char m_Name[128];         //定义私有数据成员
    void OutputName()         //定义私有成员函数
    {
        printf("%s\n",m_Name); //输出数据成员信息
    }
public:
    void SetItemName(const char* pchData) //定义共有方法
    {
        if(pchData != NULL) //判断指针是否为空
        {
            strcpy(m_Name,pchData); //赋值字符串
        }
    }
    CItem()                  //构造函数
    {
        memset(m_Name,0,128); //初始化数据成员 m_Name
    }
};
void CList::OutputItem()    //CList 类的 OutputItem 成员函数的实现
{
    m_pltem->SetItemName("BeiJing"); //调用 CItem 类的共有方法
    m_pltem->OutputName();           //在友元函数中访问 CItem 类的私有方法 OutputName
}
CList::CList()              //CList 类的默认构造函数
{
    m_pltem = new CItem();    //构造 m_pltem 对象
}
CList::~~CList()            //CList 类的析构函数
{
    delete m_pltem;          //释放 m_pltem 对象
    m_pltem = NULL;          //将 m_pltem 对象设置为空
}
int main(int argc, char* argv[]) //主函数
{
    CList list;               //定义 CList 对象 list
    list.OutputItem();         //调用 CList 的 OutputItem 方法
    return 0;
}

```

上述代码中,在定义 CItem 类时,使用 friend 关键字将 CList 类的 OutputItem 方法设置为友元函数,在 CList 类的 OutputItem 方法中访问了 CItem 类的私有方法 OutputName。程序运行结果如图 10.8 所示。

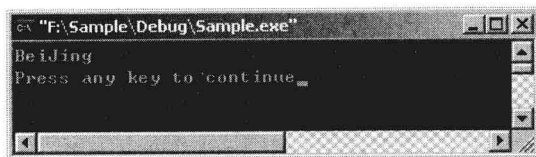


图 10.8 友元函数

对于友元函数来说，不仅可以是类的成员函数，还可以是一个全局函数。例如：

```

class CItem                                     //定义 CItem 类
{
friend void OutputItem(CItem *pItem);          //将全局函数 OutputItem 定义为友元函数
private:
    char m_Name[128];                          //定义数据成员
    void OutputName()                          //定义私有方法
    {
        printf("%s\n",m_Name);                //输出信息
    }
public:
    void SetItemName(const char* pchData)       //定义共有方法
    {
        if(pchData != NULL)                  //判断指针是否为空
        {
            strcpy(m_Name,pchData);           //赋值字符串
        }
    }
    CItem()                                    //定义构造函数
    {
        memset(m_Name,0,128);                //初始化数据成员
    }
};
void OutputItem(CItem *pItem)                //定义全局函数
{
    if(pItem != NULL)                         //判断参数是否为空
    {
        pItem->SetItemName("同一个世界，同一个梦想\n"); //调用 CItem 类的共有方法
        pItem->OutputName();                   //调用 CItem 类的私有方法
    }
}
int main(int argc, char* argv[])              //主函数
{
    CItem Item;                               //定义一个 CItem 类对象 Item
    OutputItem(&Item);                         //通过全局函数访问 CItem 类的私有方法
    return 0;
}

```

在上面的代码中，定义全局函数 `OutputItem`，在 `CItem` 类中使用 `friend` 关键字将 `OutputItem` 函数声明为友元函数。而 `CItem` 类中 `OutputName` 函数的属性是私有的，那么对外是不可见的。因为

OutputItem 是 CItem 类的友元函数，所以可以引用类中的私有成员。

## 10.6 命名空间

 视频讲解：光盘\TM\lx\10\命名空间.exe

### 10.6.1 使用命名空间

在一个应用程序的多个文件中可能会存在同名的全局对象，这样会导致应用程序的链接错误。使用命名空间是消除命名冲突的最佳方式。

例如，下面的代码定义了两个命名空间。

```
namespace MyName1
{
    int iInt1=10;
    int iInt2=20;
};

namespace MyName2
{
    int iInt1=10;
    int iInt2=20;
};
```

在上面的代码中，namespace 是关键字，而 MyName1 和 MyName2 是定义的两个命名空间名称，大括号中是所属命名空间中的对象。虽然在两个大括号中定义的变量是一样的，但是因为在不同的命名空间中，所以避免了标识符的冲突，保证了标识符的唯一性。

总而言之，命名空间就是一个命名的范围区域，程序员在这个特定的范围内创建的所有标识符都是唯一的。

### 10.6.2 定义命名空间

在 10.6.1 节中了解了有关命名空间的作用和使用的意义，本节具体来介绍如何定义命名空间。命名空间的定义格式为：

```
namespace 名称
{
    常量、变量、函数等对象的定义
}
```

定义命名空间要使用关键字 namespace，例如：



```
namespace MyName
{
    int iInt1=10;
    int iInt2=20;
};
```

在代码中，MyName 就是定义的命名空间的名称，在大括号中定义了两个整型变量 iInt1 和 iInt2，那么这两个整型变量就是属于 MyName 这个命名空间范围内的。

命名空间定义完成，如何使用其中的成员呢？在讲解类时曾介绍过使用作用域限定符“::”来引用类中的成员，在这里依然使用“::”来引用空间中的成员。引用空间成员的一般形式是：

命名空间名称::成员;

例如引用 MyName 命名空间中的成员：

MyName::iInt1=30;

**【例 10.6】** 定义命名空间。（实例位置：光盘\TM\sl\10\6）

在本实例中，定义命名空间包含变量成员，使其具有唯一性。

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

namespace MyName1                                //定义命名空间
{
    int iValue=10;
};

namespace MyName2                                //定义命名空间
{
    int iValue=20;
};

int iValue=30;                                    //全局变量

int main()
{
    cout<<MyName1::iValue<<endl;                //引用 MyName1 命名空间中的变量
    cout<<MyName2::iValue<<endl;                //引用 MyName2 命名空间中的变量
    cout<<iValue<<endl;
    return 0;
}
```

程序中使用 namespace 关键字定义两个命名空间，分别是 MyName1 和 MyName2。在两个命名空间范围中，都定义了变量 iValue，不过对其赋值分别为 10 和 20。

在源文件中又定义一个全局变量 iValue，赋值为 30。在主函数 main 中分别调用命名空间中的 iValue 变量和全局变量，将值进行输出显示。MyName1::iValue 表示引用 MyName1 命名空间中的变量，而 MyName2::iValue 表示引用 MyName2 命名空间中的变量，iValue 是全局变量。

通过使用命名空间的方法，虽然定义相同名称的变量表示不同的值，但是也可以正确的进行引用显示。

程序运行结果如图 10.9 所示。

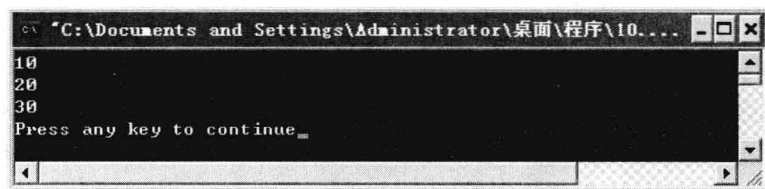


图 10.9 定义命名空间

还有另一种引用命名空间中成员的方法，就是使用 `using namespace` 语句。一般形式为：

`using namespace 命名空间名称;`

例如在源程序中包含 `MyName` 命名空间：

```
using namespace MyName;
int i=30;
```

如果使用 `using namespace` 语句，则在引用空间中的成员时直接使用就可以。

【例 10.7】 使用 `using namespace` 语句。（实例位置：光盘\TM\sl\10\7）

在本实例中使用 `using namespace` 语句将命名空间包含在程序中，引用命名空间中的变量。

```
#include<iostream>

namespace MyName                //定义命名空间
{
    int iValue=10;              //定义整型变量
}

using namespace std;             //使用命名空间 std
using namespace MyName;         //使用命名空间 MyName

int main()
{
    cout<<iValue<<endl;        //输出命名空间中的变量
    return 0;
}
```

在程序中先定义命名空间 `MyName`，之后使用 `using namespace` 语句，使用 `MyName` 命名空间，这样在 `main` 函数中使用的 `iValue` 变量就是指 `MyName` 命名空间中的 `iValue` 变量。

程序运行结果如图 10.10 所示。

需要注意的是，如果定义多个命名空间，并且在这些命名空间中都有相同标识符的成员，那么使用 `using namespace` 语句进行包含在引用成员时就会产生歧义性。这时最好还是使用作用域限定符来进行引用。



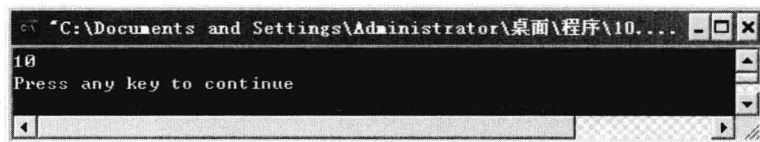


图 10.10 使用 using namespace 语句

### 10.6.3 在多个文件中定义命名空间

在定义命名空间时，通常在头文件中声明命名空间中的函数，在源文件中定义命名空间中的函数，将程序的声明与实现分开。例如，在头文件中声明命名空间函数：

```
namespace Output
{
    void Demo();           //声明函数
}
```

在源文件中定义函数：

```
void Output::Demo()       //定义函数
{
    cout<<"This is a  function!\n";
}
```

在源文件中定义函数时，注意要使用命名空间名作为前缀，表明实现的是命名空间中定义的函数，否则将是定义一个全局函数。

**【例 10.8】** 在多个文件中定义命名空间。（实例位置：光盘\TM\sl\10\8）

在本实例中，将命名空间的定义放在头文件中，而将命名空间中有关成员的定义放在源文件中。

```
////////////////////////////////////
//Detach.h 头文件中
////////////////////////////////////

namespace Output
{
    void Demo();           //声明函数
}

////////////////////////////////////
//Detach.cpp 源文件中
////////////////////////////////////

#include<iostream>
#include"Detach.h"
using namespace std;

void Output::Demo()       //定义函数
```

```

{
    cout<<"This is a  function!\n";
}

int main()
{
    Output::Demo();           //调用函数
    return 0;
}

```

将命名空间中的定义和命名空间中成员的具体操作分开，这样更符合程序编写规范并且非常易于修改和观察。

程序运行结果如图 10.11 所示。

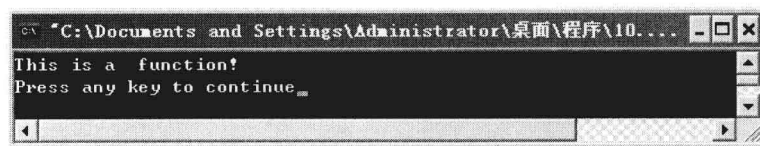


图 10.11 在多个文件中定义命名空间

在 Detach.cpp 头文件中还可以定义 Output 命名空间。例如：

```

namespace Output
{
    void show()
    {
        cout<<"This is show function"<<endl;
    }
}

```

此时，命名空间 Output 中的内容为两个文件 Output 命名空间内容的“总和”。因此，如果在 login.cpp 文件的 Output 命名空间中定义一个函数名称为 Demo 是非法的，因为进行了重复的定义，这时编译器会提示 Demo 已经有一个函数体。

#### 10.6.4 定义嵌套的命名空间

命名空间可以定义在其他的命名空间中，在这种情况下，仅通过使用外部的命名空间作为前缀，程序便可以引用在命名空间之外定义的其他标识符。然而，在命名空间内不定的标识符需要作为外部命名空间和内部命名空间名称的前缀出现。例如：

```

namespace Output
{
    void Show()           //定义函数
    {
        cout<<"Output's function!"<<endl;
    }
}

```

```

namespace MyName
{
    void Demo()                //定义函数
    {
        cout<<"MyName's function!"<<endl;
    }
}

```

上述代码中，在 `Output` 命名空间中又定义了一个命名空间 `MyName`，如果程序中访问 `MyName` 命名空间中的对象，可以使用外层的命名空间和内层的命名空间作为前缀。例如：

```

Output::MyName::Demo();        //调用 MyName 命名空间中的函数

```

用户也可以直接使用 `using` 命令引用嵌套的 `MyName` 命名空间。例如：

```

using namespace Output::MyName; //引用嵌套的 MyName 命名空间
Demo();                          //调用 MyName 命名空间中的函数

```

上述代码中，“`using namespace Output::MyName;`”语句只是引用了嵌套在 `Output` 命名空间中的 `MyName` 命名空间，并没有引用 `Output` 命名空间，因此试图访问 `Output` 命名空间中定义的对象是非法的。例如：

```

using namespace Windows::GDI;
show(); //错误的访问，无法访问 Output 命名空间中的函数

```

#### 【例 10.9】 定义嵌套的命名空间。（实例位置：光盘\TM\sl\10\9）

在本实例中定义嵌套命名空间，使用命名空间名称选择调用函数。

```

#include<iostream>
using namespace std;

namespace Output                //定义命名空间
{
    void Show()                 //定义函数
    {
        cout<<"Output's function!"<<endl;
    }
    namespace MyName            //定义嵌套命名空间
    {
        void Demo()            //定义函数
        {
            cout<<"MyName's function!"<<endl;
        }
    }
}

int main()

```

```

{
    Output::Show();           //调用 Output 命名空间中的函数
    Output::MyName::Demo();   //调用 MyName 命名空间中的函数
    return 0;
}

```

在程序中定义了 Output 命名空间，在其中又定义了命名空间 MyName。

Show 函数属于 Output 命名空间中的成员，而 Demo 函数属于 MyName 命名空间中的成员。在 main 函数中调用 Show 和 Demo 函数时，要将所属的命名空间的作用范围写出。Output::Show 表示在 Output 命名空间范围的 Show 函数，Output::MyName::Demo 表示在嵌套命名空间 MyName 中的成员函数。

程序运行结果如图 10.12 所示。

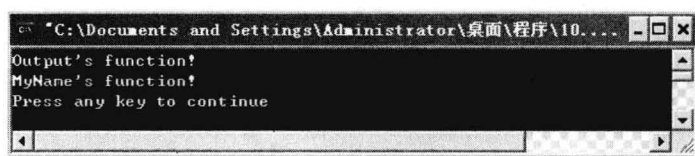


图 10.12 定义嵌套的命名空间

### 10.6.5 定义未命名的命名空间

尽管为命名空间指定名称是有益的，但是 C++中也允许在定义中省略命名空间的名称来简单的定义未命名的命名空间。

例如定义一个包含两个整型变量的未命名的命名空间：

```

namespace
{
    int iValue1=10;
    int iValue2=20;
}

```

事实上在未命名空间中定义的标识符被设置为全局命名空间，但这样就违背了命名空间的设置原则。所以，未命名的命名空间没有被广泛应用。

## 10.7 小 结

通过本章的学习，使得读者进入了有关面向对象的程序设计。在面向对象的程序设计中，其设计思路和人们日常生活中处理问题的方法相同，类是实现面向对象程序设计的基础。本章介绍了有关 C++ 中类的基础概念，讲解如何声明类并且介绍了如何实现一个类。之后介绍了有关类中构造函数和析构函数的作用，还有类成员的相关内容。最后介绍了使用友元来访问类中不可见的成员，讲解 C++ 语言中命名空间的使用。

## 10.8 实践与练习

1. 求解一个圆柱体的体积。编写一个基于对象的程序，并要求实现下面的功能：

(1) 根据输入，得到圆柱体的半径、高（其中 $\pi$ 为3.14）。

(2) 计算圆柱体的体积，并输出。

**(答案位置：光盘\TM\sl\10\10)**

2. 开发一个程序，分别定义学生类(CStudent)。要求：

(1) 在CStudent类中包含“姓名”、“性别”、“年级”、“班级”数据成员。

(2) 在CStudent类中声明成员函数SetInfor，该函数的作用是设置成员变量的数值。

(3) 在CStudent类中声明ShowInfor函数，将其中的数据信息输出。

**(答案位置：光盘\TM\sl\10\11)**





# 第 11 章

---

## 继承与派生


(  视频讲解：57 分钟 )

继承与派生是面向对象程序设计的两个重要特性，继承是从已有的类那里得到已有的特性，已有的类为基类或父类，新类被称为派生类或子类。继承与派生是从不同角度说明类之间的关系，这种关系包含了访问机制、多态和重载等。

通过阅读本章，您可以：

- » 了解访问控制
- » 掌握重载运算符
- » 掌握虚函数
- » 掌握多态
- » 掌握抽象类

## 11.1 继 承

 视频讲解：光盘\TM\lx\11\继承.exe

继承（inheritance）是面向对象的主要特征（此外还有封装和多态）之一，它使得一个类可以从现有类中派生，而不必重新定义一个新类。继承的实质就是用已有的数据类型创建新的数据类型，并保留已有数据类型的特点，以旧类为基础创建新类，新类包含了旧类的数据成员和成员函数，并且可以在新类中添加新的数据成员和成员函数。旧类被称为基类或父类，新类被称为派生类或子类。

### 11.1.1 类的继承

类继承的形式如下：

```
class 派生类名标识符: [继承方式] 基类名标识符
{
    [访问控制修饰符:]
    [成员声明列表]
};
```

继承方式有 3 种派生类型，分别为共有型（public）、保护型（protected）和私有型（private）；访问控制修饰符也是 public、protected、private 3 种类型；成员声明列表中包含类的成员变量及成员函数，是派生类新增的成员。“:” 是一个运算符，表示基类和派生类之间的继承关系，如图 11.1 所示。

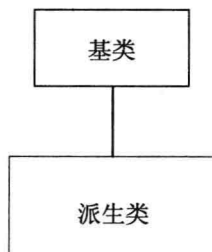


图 11.1 继承关系

例如，定义一个继承员工类的操作员类。首先定义一个员工类，它包含员工 ID、员工姓名、所属部门等信息。

```
class CEmployee                                //定义员工类
{
public:
    int m_ID;                                  //定义员工 ID
    char m_Name[128];                          //定义员工姓名
    char m_Depart[128];                       //定义所属部门
};
```

然后定义一个操作员类，通常操作员属于公司的员工，它包含员工 ID、员工姓名、所属部门等信息，此外还包含密码信息、登录方法等。

```
class COperator :public CEmployee           //定义一个操作员类，从 CEmployee 类派生而来
{
public:
    char m_Password[128];                  //定义密码
    bool Login();
};
```

操作员类是从员工类派生的一个新类，新类中增加密码信息、登录方法等信息，员工 ID、员工姓名等信息直接从员工类中继承得到。

**【例 11.1】** 类的继承。(实例位置：光盘\TM\sl\11\1)

```
#include<iostream>
using namespace std;
class CEmployee           //定义员工类
{
public:
    int m_ID;              //定义员工 ID
    char m_Name[128];       //定义员工姓名
    char m_Dept[128];       //定义所属部门
    CEmployee()             //定义默认构造函数
    {
        memset(m_Name,0,128); //初始化 m_Name
        memset(m_Dept,0,128); //初始化 m_Dept
    }
    void OutputName()       //定义共有成员函数
    {
        cout <<"员工姓名"<<m_Name<<endl; //输出员工姓名
    }
};
class COperator :public CEmployee //定义一个操作员类，从 CEmployee 类派生而来
{
public:
    char m_Password[128];    //定义密码
    bool Login()             //定义登录成员函数
    {
        if(strcmp(m_Name,"MR")==0 && //比较用户名
            strcmp(m_Password,"KJ")==0) //比较密码
        {
            cout<<"登录成功!"<<endl; //输出信息
            return true;              //设置返回值
        }
        else
        {
            cout<<"登录失败!"<<endl; //输出信息
            return false;             //设置返回值
        }
    }
};
```

```

    }
};
int main(int argc, char* argv[])
{
    COperator optr;                //定义一个 COperator 类对象
    strcpy(optr.m_Name,"MR");      //访问基类的 m_Name 成员
    strcpy(optr.m_Password,"KJ");  //访问 m_Password 成员
    optr.Login();                  //调用 COperator 类的 Login 成员函数
    optr.OutputName();             //调用基类 CEmployee 的 OutputName 成员函数
    return 0;
}

```

程序中 CEmployee 类是 COperator 类的基类，也就是父类。COperator 类将继承 CEmployee 类的所有非私有成员（private 类型成员不能被继承）。optr 对象初始化 m\_Name 和 m\_Password 成员后，调用了 Login 成员函数，程序运行结果如图 11.2 所示。



图 11.2 访问父类成员函数

用户在父类中派生子类时，可能存在一种情况，即在子类中定义了一个与父类同名的成员函数，此时称为子类隐藏了父类的成员函数。例如，重新定义 COperator 类，添加一个 OutputName 成员函数。

### 11.1.2 继承后可访问性

继承方式有 public、private、protected 3 种，其说明如下：

#### ☒ public（共有型派生）

共有型派生表示对于基类中的 public 数据成员和成员函数，在派生类中仍然是 public，对于基类中的 private 数据成员和成员函数，在派生类中仍然是 private。例如：

```

class CEmployee
{
public:
    void Output()
    {
        cout << m_ID << endl;
        cout << m_Name << endl;
        cout << m_Depart << endl;
    }
private:
    int m_ID;
    char m_Name[128];
    char m_Depart[128];
};
class COperator :public CEmployee
{
public:
    void Output()

```



```

    {
        cout << m_ID << endl;           //引用基类的私有成员, 错误
        cout << m_Name << endl;         //引用基类的私有成员, 错误
        cout << m_Depart << endl;       //引用基类的私有成员, 错误
        cout << m_Password << endl;     //正确
    }
private:
    char m_Password[128];
    bool Login();
};

```

COperator 类无法访问 CEmployee 类中的 private 数据成员 m\_ID、m\_Name 和 m\_Depart, 如果将 CEmployee 类中的所有程序都设置为 public 后, COperator 类才能访问 CEmployee 类中的所有成员。例如:

```

class CEmployee
{
public:
    void Output()
    {
        cout << m_ID << endl;
        cout << m_Name << endl;
        cout << m_Depart << endl;
    }
//private :
    int m_ID;
    char m_Name[128];
    char m_Depart[128];
};
class COperator :public CEmployee
{
public:
    void Output()
    {
        cout << m_ID << endl;           //正确
        cout << m_Name << endl;         //正确
        cout << m_Depart << endl;       //正确
        cout << m_Password << endl;     //正确
    }
private:
    char m_Password[128];
    bool Login();
};

```

#### ☒ private (私有型派生)

私有型派生表示对于基类中的 public、protected 数据成员和成员函数, 在派生类中可以访问。基类中的 private 数据成员, 在派生类中不可以访问。例如:

```

class CEmployee
{

```

```

public:
void Output()
{
    cout << m_ID << endl;
    cout << m_Name << endl;
    cout << m_Depart << endl;
}

int m_ID;
protected:
    char m_Name[128];
private:
    char m_Depart[128];
};
class COperator :private CEmployee
{
public:
    void Output()
    {
        cout << m_ID << endl;           //正确
        cout << m_Name << endl;         //正确
        cout << m_Depart << endl;       //错误
        cout << m_Password << endl;     //正确
    }
private:
    char m_Password[128];
    bool Login();
};

```

#### ☑ protected（保护型派生）

保护型派生表示对于基类中的 `public`、`protected` 数据成员和成员函数，在派生类中均为 `protected`。`protected` 类型在派生类定义时可以访问，用派生类声明的对象不可以访问，也就是说在类体外不可以访问。`protected` 成员可以被基类的所有派生类使用。这一性质可以沿继承树无限向下传播。

因为保护类的内部数据不能被随意更改，实例类本身负责维护，这就起到很好的封装性作用。把一个类分作两部分，一部分是公共的，另一部分是保护的，保护成员对于使用者来说是不可见的，也是不需了解的，这就减少了类与其他代码的关联程度。类的功能是独立的，它不依赖于应用程序的运行环境，既可以放到这个程序中使用，也可以放到那个程序中使用。这就能够非常容易地用一个类替换另一个类。类访问限制的保护机制使人们编制的应用程序更加可靠和易维护。

### 11.1.3 构造函数访问顺序

由于父类和子类中都有构造函数和析构函数，那么子类对象在创建时是父类先进行构造，还是子类先进行构造呢？同样在子类对象释放时，是父类先进行释放，还是子类先进行释放？这些都有个先后顺序问题。答案是当从父类派生一个子类并声明一个子类的对象时，它将先调用父类的构造函数，然后调用当前类的构造函数来创建对象；在释放子类对象时，先调用的是当前类的析构函数，然后是

父类的析构函数。

**【例 11.2】** 构造函数访问顺序。(实例位置: 光盘\TM\sl\11\2)

```
#include<iostream>
using namespace std;
class CEmployee                                //定义 CEmployee 类
{
public:
    int m_ID;                                  //定义数据成员
    char m_Name[128];                          //定义数据成员
    char m_Depart[128];                        //定义数据成员
    CEmployee()                                //定义构造函数
    {
        cout << "CEmployee 类构造函数被调用"<< endl; //输出信息
    }
    ~CEmployee()                               //析构函数
    {
        cout << "CEmployee 类析构函数被调用"<< endl; //输出信息
    }
};
class COperator :public CEmployee              //从 CEmployee 类派生一个子类
{
public:
    char m_Password[128];                      //定义数据成员
    COperator()                                //定义构造函数
    {
        strcpy(m_Name,"MR");                  //设置数据成员
        cout << "COperator 类构造函数被调用"<< endl; //输出信息
    }
    ~COperator()                               //析构函数
    {
        cout << "COperator 类析构函数被调用"<< endl; //输出信息
    }
};
int main(int argc, char* argv[])              //主函数
{
    COperator optr;                             //定义一个 COperator 对象
    return 0;
}
```

程序运行结果如图 11.3 所示。

从图 11.3 中可以发现,在定义 COperator 类对象时,首先调用的是父类 CEmployee 的构造函数,然后是 COperator 类的构造函数。子类对象的释放过程则与其构造过程恰恰相反,先调用自身的析构函数,然后再调用父类的析构函数。

在分析完对象的构建、释放过程后,会考虑这样一种情况:定义一个基类类型的指针,调用子类的构造函数为其构建对象,当对象释放时,是直接调用父类的析

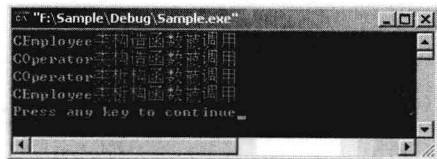


图 11.3 构造函数调用顺序



构造函数还是先调用子类的析构函数，再调用父类的析构函数呢？答案是如果析构函数是虚函数，则先调用子类的析构函数，然后再调用父类的析构函数；如果析构函数不是虚函数，则只调用父类的析构函数。可以想象，如果在子类中为某个数据成员在堆中分配了空间，父类中的析构函数不是虚成员函数，将使子类的析构函数不被调用，其结果是对象不能被正确地释放，导致内存泄漏的产生。因此，在编写类的析构函数时，析构函数通常是虚函数。构造函数调用顺序不受基类在成员初始化表中是否存在以及被列出的顺序的影响。

### 11.1.4 子类隐藏父类的成员函数

如果子类中定义了一个和父类一样的成员函数，那么子类对象是调用父类中的成员函数，还是调用子类中的成员函数呢？答案是调用子类中的成员函数。

【例 11.3】 子类隐藏父类的成员函数。（实例位置：光盘\TM\sl\11\3）

```
#include<iostream>
using namespace std;
class CEmployee                                //定义 CEmployee 类
{
public:
    int m_ID;
    char m_Name[128];                            //定义数据成员
    char m_Depart[128];                          //定义数据成员

    CEmployee()                                //定义构造函数
    {
    }
    ~CEmployee()                               //析构函数
    {
    }
    void OutputName()                          //定义 OutputName 成员函数
    {
        cout << "调用 CEmployee 类的 OutputName 成员函数:"<< endl;    //输出操作员姓名
    }                                           //定义数据成员
};
class COperator :public CEmployee              //定义 COperator 类
{
public:
    char m_Password[128];                      //定义数据成员
    void OutputName()                          //定义 OutputName 成员函数
    {
        cout << "调用 COperator 类的 OutputName 成员函数:"<< endl;    //输出操作员姓名
    }
};
int main(int argc, char* argv[])              //主成员函数
{
```

```

COperator optr;                                //定义 COperator 对象
optr.OutputName();                             //调用 COperator 类的 OutputName 成员函数
return 0;
}

```

程序运行结果如图 11.4 所示。

从图 11.4 中可以发现, 语句 “**optr.OutputName();**” 调用的是 COperator 类的 OutputName 成员函数, 而不是 CEmployee 类的 OutputName 成员函数。如果用户想要访问父类的 OutputName 成员函数, 需要显式使用父类名。例如:

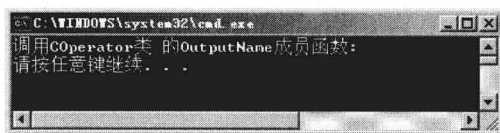


图 11.4 隐藏基类成员函数

```

COperator optr;                                //定义一个 COperator 类
strcpy(optr.m_Name,"MR");                      //赋值字符串
optr.OutputName();                             //调用 COperator 类的 OutputName 成员函数
optr.CEmployee::OutputName();                 //调用 CEmployee 类的 OutputName 成员函数

```

如果子类中隐藏了父类的成员函数, 则父类中所有同名的成员函数 (重载的函数) 均被隐藏, 因此下面黑体部分代码是错误的。例如:

```

class CEmployee                                //定义 CEmployee 类
{
public:
    int m_ID;                                  //定义数据成员
    char m_Name[128];                          //定义数据成员
    char m_Depart[128];                       //定义数据成员
    CEmployee()
    {
        memset(m_Name,0, 128);                //初始化数据成员
        memset(m_Depart,0, 128);              //初始化数据成员
        cout << "员工类构造函数被调用"<< endl; //输出信息
    }
    void OutputName()                          //定义重载成员函数
    {
        cout << "员工姓名: "<<m_Name<< endl; //输出信息
    }
    void OutputName(const char* pchData)        //定义重载成员函数
    {
        if(pchData != NULL)                   //判断参数是否为空
        {
            strcpy(m_Name,pchData);            //复制字符串
            cout << "设置并输出员工姓名:"<<pchData<< endl; //输出信息
        }
    }
};
class COperator :public CEmployee              //定义 COperator 类
{

```



```

public:
    char m_Password[128];           //定义数据成员
    void OutputName()               //定义 OutputName 成员函数，隐藏基类的成员函数
    {
        cout << "操作员姓名: "<<m_Name<< endl; //输出信息
    }
    bool Login()                   //定义 Login 成员函数
    {
        if(strcmp(m_Name,"MR")==0 && //比较用户名称
            strcmp(m_Password,"KJ")==0) //比较用户密码
        {
            cout << "登录成功"<< endl; //输出信息
            return true;                //设置返回值
        }
        else
        {
            cout << "登录失败"<< endl; //输出信息
            return false;              //设置返回值
        }
    }
};

int main(int argc, char* argv[])
{
    COperator optr;                 //定义 COperator 类对象
    optr.OutputName("MR");          //错误的代码，不能访问基类的重载成员函数
    return 0;
}

```

程序中，在 CEmployee 类中定义了重载的 OutputName 成员函数，而在 COperator 类中又定义了一个 OutputName 成员函数，导致父类中的所有同名成员函数被隐藏。语句“optr.OutputName("MR");”是错误的。如果用户想要访问被隐藏的父类成员函数，依然需要指定父类名称。例如：

```

COperator optr;           //定义一个 COperator 对象
optr.CEmployee::OutputName("MR"); //调用基类中被隐藏的成员函数

```

在派生完一个子类后，可以定义一个父类的类型指针，通过子类的构造函数为其创建对象。例如：

```

CEmployee *pWorker = new COperator (); //定义 CEmployee 类型指针，调用子类构造函数

```

如果使用 pWorker 对象调用 OutputName 成员函数，例如执行“pWorker->OutputName();”语句，调用的是 CEmployee 类的 OutputName 成员函数还是 COperator 类的 OutputName 成员函数呢？答案是调用 CEmployee 类的 OutputName 成员函数。编译器对 OutputName 成员函数进行的是静态绑定，即根据对象定义时的类型来确定调用哪个类的成员函数。由于 pWorker 属于 CEmployee 类型，因此调用的是 CEmployee 类的 OutputName 成员函数。那么是否有成员函数执行“pWorker->OutputName();”语句调用 COperator 类的 OutputName 成员函数呢？答案是通过定义虚函数可以实现。虚函数会在后面章节讲到。

## 11.2 重载运算符

 视频讲解：光盘\TM\lx\11\重载运算符.exe

运算符实际上是一个函数，所以运算符的重载实际上是函数的重载。编译程序对运算符重载的选择，遵循函数重载的选择原则。当遇到不很明显的运算时，编译程序会去寻找与参数相匹配的运算符函数。

### 11.2.1 重载运算符的必要性

C++语言中的数据类型分为基础数据类型和构造数据类型，基础数据类型可以直接完成算术运算。例如：

```
#include<iostream>
using namespace std;
void main()
{
    int a=10;
    int b=20;
    cout << a+b << endl;    //两个整型变量相加
}
```

程序中实现了两个整型变量的相加，可以正确输出运行结果 30，通过两个浮点变量、两个双精度变量都可以直接运用加法运算符“+”求和。但是类属于新构造的数据类型，类的两个对象就无法通过加法运算符来求和。例如：

```
#include<iostream>
using namespace std;
class CBook
{
public:
    CBook(int iPage)
    {
        m_iPage=iPage;
    }
    void display()
    {
        cout << m_iPage << endl;
    }
protected:
    int m_iPage;
};
void main()
```

```

{
    CBook bk1(10);
    CBook bk2(20);
    CBook tmp(0);
    tmp=bk1+bk2;    //错误
    tmp.display();
}

```

当编译器编译到语句“bk1+bk2”时就会报错，因为编译器不知道如何进行两个对象的相加。要实现两个类对象的加法运算有两种方法，一种是通过成员函数，另一种是重载操作符。

下面给出通过成员函数的方法实现求和的实例。

```

#include<iostream>
using namespace std;
class CBook
{
public:
    CBook(int iPage)
    {
        m_iPage=iPage;
    }
    int add(CBook a)
    {
        return m_iPage+a.m_iPage;
    }
protected:
    int m_iPage;
};

void main()
{
    CBook bk1(10);
    CBook bk2(20);
    cout << bk1.add(bk2) << endl;
}

```

程序可以正确输出运行结果 30。使用成员函数实现求和形式比较单一，并且不利于代码复用。如果要实现多个对象的累加，其代码的可读性会大大降低。使用重载操作符方法就可以解决这些问题。

### 11.2.2 重载运算符的形式与规则

重载运算符的声明形式如下：

```
operator 类型名();
```

`operator` 是需要重载的运算符，整个语句没有返回类型，因为类型名就代表了它的返回类型。重载运算符将对象转换成类型名规定的类型，转换时的形式就像强制转换一样，但如果没有重载运算符定



义，直接用强制转换编译器将无法通过编译。

重载运算符不可以是新创建的运算符，只能是 C++ 语言中已有的运算符。可以重载的运算符如下：

- ☑ 算术运算符：+、-、\*、/、%、++、--。
- ☑ 位操作运算符：&、|、~、^、>>、<<。
- ☑ 逻辑运算符：!、&&、||。
- ☑ 比较运算符：<、>、>=、<=、==、!=。
- ☑ 赋值运算符：=、+=、-=、\*=、/=、%=、&=、|=、^=、<<=、>>=。
- ☑ 其他运算符：[]、()、->、逗号、new、delete、new[]、delete[]、->\*。

并不是所有的 C++ 语言中已有的运算符都可以重载，不允许重载的运算符有 “.”、“\*”、“::”、“?” 和 “:”。

重载运算符时不能改变运算符操作数的个数、运算符原有的优先级、运算符原有的结合性及运算符原有的语法结构，即单目运算符只能重载为单目运算符，双目运算符只能重载为双目运算符。重载运算符含义必须清楚，不能有二义性。

**【例 11.4】** 通过重载运算符实现求和。（实例位置：光盘\TM\sl\11\4）

```
#include<iostream>
using namespace std;
class CBook
{
public:
    CBook(int iPage)
    {
        m_iPage=iPage;
    }
    CBook operator+(CBook b)
    {
        return CBook(m_iPage+b.m_iPage);
    }
    void display()
    {
        cout << m_iPage << endl;
    }
protected:
    int m_iPage;
};

void main()
{
    CBook bk1(10);
    CBook bk2(20);
    CBook tmp(0);
    tmp= bk1+bk2;
    tmp.display();
}
```

类 CBook 重载了求和运算符后，由它声明的两个对象 bk1 和 bk2 可以像两个整型变量一样相加。

### 11.2.3 重载运算符的运算

重载运算符后可以完成对象和对象之间的运算，同样也可以通过重载运算实现对象和普通类型数据的运算。例如：

```
#include<iostream>
using namespace std;
class CBook
{
public:
    int m_Pages;
    void OutputPages()
    {
        cout << m_Pages<< endl;
    }
    CBook()
    {
        m_Pages=0;
    }
    CBook operator+(const int page)
    {
        CBook bk;
        bk.m_Pages = m_Pages + page;
        return bk;
    }
};
void main()
{
    CBook vbBook,vfBook;
    vfBook = vbBook + 10;
    vfBook. OutputPages();
}
```

通过修改运算符的参数为整型数类型，可以实现 CBook 对象与整型数相加。

对于两个整型变量相加，用户可以调换加数和被加数的顺序，因为加法符合交换律。但是，对于通过重载运算符实现的两个不同类型的对象相加，则不可以，因此下面的语句是非法的。

```
vfBook = 10 + vbBook; //非法的代码
```

对于++和--运算符，由于涉及前置运算和后置运算，在重载这类运算符时如何区分呢？默认情况下，如果重载运算符没有参数，则表示是前置运算。例如：

```
void operator++() //前置运算
{
    ++m_Pages;
}
```



如果重载运算符使用了整数作为参数，则表示是后置运算，此时的参数值可以被忽略，它只是一个标识，标识后置运算。例如：

```
void operator++(int)                //后置运算
{
    ++m_Pages;
}
```

默认情况下，将一个整数赋值给一个对象是非法的，可以通过重载赋值运算符将其变为合法的。例如：

```
void operator = (int page)          //重载赋值运算符
{
    m_Pages = page;
}
```

10.2.2 节介绍了通过复制构造函数将一个对象复制成另一个对象，那么通过重载赋值运算符也可以实现将一个整型数复制给一个对象。例如：

```
#include<iostream>
using namespace std;
class CBook
{
public:
    int m_Pages;
    void OutputPages()
    {
        cout << m_Pages<< endl;
    }
    CBook(int page)
    {
        m_Pages = page;
    }
    operator=(const int page)
    {
        m_Pages = page;
    }
};
void main()
{
    CBook mybk(0);
    mybk=100;
    mybk.OutputPages();
}
```

程序中重载了赋值运算符，给 mybk 对象赋值 100，并通过 OutputPages 成员函数将该值输出。也可以通过重载构造函数将一个整数赋值给一个对象。例如：

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

```

class CBook
{
public:
    int m_Pages;
    void OutputPages()
    {
        cout << m_Pages << endl;
    }
    CBook()
    {
        ;
    }
    CBook(int page)
    {
        m_Pages = page;
    }
};

void main()
{
    CBook vbBook;
    vbBook = 200;
    vbBook.OutputPages();
}

```

程序中定义了一个重载的构造函数，以一个整型数作为函数参数，这就可以将一个整型数赋值给一个 CBook 类对象。语句“vbBook = 200;”将调用构造函数 CBook(int page)重新构造一个 CBook 对象，并将其赋值给 vbBook 对象。

#### 11.2.4 转换运算符

C++语言中普通的数据类型可以进行强制类型转换，例如：

```

int i=10;
double d;
d=(double)i;

```

程序中将整型数 i 强制转换成双精度型。

语句

```
d=(double)i;
```

等同于

```
d= double(i);
```

double()在 C++语言中被称为转换运算符。通过重载转换运算符可以将类对象转换成想要的数。

【例 11.5】 转换运算符。（实例位置：光盘\TM\sl\11\5）

```

#include<iostream>
using namespace std;
class CBook
{
public:
    CBook(double iPage=0);
    operator double()
    {
        return m_iPage;
    }

protected:
    int m_iPage;
};
CBook:: CBook(double iPage)
{
    m_iPage=iPage;
}
void main()
{
    CBook bk1(10.0);
    CBook bk2(20.00);
    cout << "bk1+bk2=" << double(bk1)+double(bk2) << endl;
}

```

程序运行结果如图 11.5 所示。

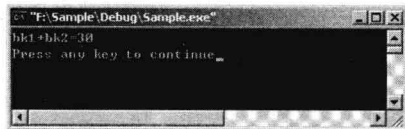


图 11.5 转换运算符

程序重载了转换运算符 `double()`，然后将类 `CBook` 的两个对象强制转换为 `double` 类型后再进行求和，最后输出求和的结果。

## 11.3 多重继承

 视频讲解：光盘\TM\lx\11\多重继承.exe

前文介绍的继承方式属于单继承，即子类只从一个父类继承共有的和受保护的成员。与其他面向对象语句不同，C++语言允许子类从多个父类继承共有的和受保护的成员，这被称为多重继承。

### 11.3.1 多重继承定义

多重继承是指有多个基类名标识符，其声明形式如下：



```
class 派生类名标识符: [继承方式] 基类名标识符 1,...,访问控制修饰符 基类名标识符 n
{
    [访问控制修饰符:]
    [成员声明列表]
};
```

声明形式中有“:”运算符, 基类名标识符之间用“,”运算符分开。

例如, 鸟能够在天空飞翔, 鱼能够在水里游, 而水鸟既能够在天空飞翔, 又能够在水里游。那么在定义水鸟类时, 可以将鸟和鱼同时作为其基类。

```
class CBird //定义鸟类
{
public:
    void FlyInSky() //定义成员函数
    {
        cout << "鸟能够在天空飞翔"<< endl; //输出信息
    }
    void Breath() //定义成员函数
    {
        cout << "鸟能够呼吸"<< endl; //输出信息
    }
};
class CFish //定义鱼类
{
public:
    void SwimInWater() //定义成员函数
    {
        cout << "鱼能够在水里游"<< endl; //输出信息
    }
    void Breath() //定义成员函数
    {
        cout << "鱼能够呼吸"<< endl; //输出信息
    }
};
class CWaterBird: public CBird, public CFish //定义水鸟类, 从鸟类和鱼类派生
{
public:
    void Action() //定义成员函数
    {
        cout << "水鸟既能飞又能游"<< endl; //输出信息
    }
};
int main(int argc, char* argv[]) //主函数
{
    CWaterBird waterbird; //定义水鸟对象
    waterbird.FlyInSky(); //调用从鸟类继承而来的 FlyInSky 成员函数
    waterbird.SwimInWater(); //调用从鱼类继承而来的 SwimInWater 成员函数
    return 0;
}
```

程序运行结果如图 11.6 所示。

程序中定义了鸟类 CBird 和鱼类 CFish，然后从鸟类和鱼类派生了一个子类水鸟类 CWaterBird。水鸟类自然继承了鸟类和鱼类的所有共有和受保护的成员，因此 CWaterBird 类对象能够调用 FlyInSky 和 SwimInWater 成员函数。在 CBird 类中提供了一个 Breath 成员函数，在 CFish 类中同样提供了 Breath 成员函数，如果 CWaterBird 类对象调用 Breath 成员函数，将会执行哪个类的 Breath 成员函数呢？答案是将会出现编译错误，编译器将产生歧义，不知道具体调用哪个类的 Breath 成员函数。为了让 CWaterBird 类对象能够访问 Breath 成员函数，需要在 Breath 成员函数前具体指定类名。例如：

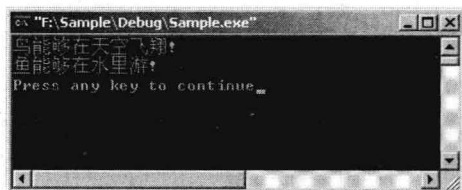


图 11.6 多重继承

```
waterbird.CFish::Breath();           //调用 CFish 类的 Breath 成员函数
waterbird.CBird::Breath();           //调用 CBird 类的 Breath 成员函数
```

在多重继承中存在这样一种情况，假如 CBird 类和 CFish 类均派生于同一个父类，例如 CAnimal 类，那么当从 CBird 类和 CFish 类派生子类 CWaterBird 时，在 CWaterBird 类中将存在两个 CAnimal 类的复制。能否在派生 CWaterBird 类时，使其只存在一个 CAnimal 基类呢？为了解决该问题，C++ 语言提供了虚继承的机制，将会在后面章节讲到。

### 11.3.2 二义性

派生类在调用成员函数时，先在自身的作用域内寻找，如果找不到，会到基类中寻找，但当派生类继承的基类中有同名成员时，派生类中就会出现来自不同基类的同名成员。例如：

```
class CBaseA
{
public:
    void function();
};
class CBaseB
{
public:
    void function();
};
class CDeriveC:public CBaseA,public CBaseB
{
public:
    void function();
};
```

CBaseA 和 CBaseB 都是 CDeriveC 的父类，并且两个父类中都含有 function 成员函数，CDeriveC 将不知道调用哪个基类的 function 成员函数，这就产生了二义性。



### 11.3.3 多重继承的构造顺序

11.1.3 节讲过，单一继承是先调用基类的构造函数，然后调用派生类的构造函数，但多重继承将如何调用构造函数呢？多重继承中的基类构造函数被调用的顺序以类派生表中声明的顺序为准。派生表就是多重继承定义中继承方式后面的内容，调用顺序就是按照基类名标识符的前后顺序进行的。

**【例 11.6】** 多重继承的构造顺序。（实例位置：光盘\TM\sl\11\6）

```
#include<iostream>
using namespace std;
class CBicycle
{
public:
    CBicycle()
    {
        cout << "Bicycle Construct" << endl;
    }
    CBicycle(int iWeight)
    {
        m_iWeight=iWeight;
    }
    void Run()
    {
        cout << "Bicycle Run" << endl;
    }

protected:
    int m_iWeight;
};

class CAirplane
{
public:
    CAirplane()
    {
        cout << "Airplane Construct " << endl;
    };
    CAirplane(int iWeight)
    {
        m_iWeight=iWeight;
    }
    void Fly()
    {
        cout << "Airplane Fly " << endl;
    }

protected:
```

```

    int m_iWeight;
};

class CAirBicycle : public CBicycle, public CAirplane
{
public:
    CAirBicycle()
    {
        cout << "CAirBicycle Construct" << endl;
    }
    void RunFly()
    {
        cout << "Run and Fly" << endl;
    }
};

void main()
{
    CAirBicycle ab;
    ab.RunFly();
}

```

程序运行结果如图 11.7 所示。

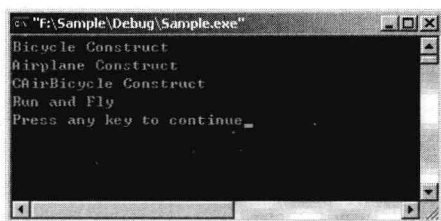


图 11.7 多重继承的构造顺序

程序中基类的声明顺序是先 CBicycle 类后 CAirplane 类，所以对象的构造顺序就是先 CBicycle 类后 CAirplane 类，最后是 CAirBicycle 类。

## 11.4 多 态

 视频讲解：光盘\TM\lx\11\多态.exe

多态性 (polymorphism) 是面向对象程序设计的一个重要特征，利用多态性可以设计和实现一个易于扩展的系统。在 C++ 语言中，多态性是指具有不同功能的函数可以用同一个函数名，这样就可以用一个函数名调用不同内容的函数，发出同样的消息被不同类型的对象接收时，导致完全不同的行为。这里所说的消息主要指类的成员函数的调用，而不同的行为是指不同的实现。

多态性通过联编实现。联编是指一个计算机程序自身彼此关联的过程。按照联编所进行的阶段不同，可分为两种不同的联编方法：静态联编和动态联编。在 C++ 语句中，根据联编的时刻不同，存在

两种类型多态性，即函数重载和虚函数。

### 11.4.1 虚函数概述

在类的继承层次结构中，在不同的层次中可以出现名字、参数个数和类型都相同而功能不同的函数。编译器按照先自己后父类的顺序进行查找覆盖，如果子类有与父类相同原型的成员函数时，要想调用父类的成员函数，需要对父类重新引用调用。虚函数则可以解决子类和父类相同原型成员函数的函数调用问题。虚函数允许在派生类中重新定义与基类同名的函数，并且可以通过基类指针或引用来访问基类和派生类中的同名函数。

在基类中用 `virtual` 声明成员函数为虚函数，在派生类中重新定义此函数，改变该函数的功能。在 C++ 语言中虚函数可以继承，当一个成员函数被声明为虚函数后，其派生类中的同名函数都自动成为虚函数，但如果派生类没有覆盖基类的虚函数，则调用时调用基类的函数定义。

覆盖和重载的区别是，重载是同一层次函数名相同，覆盖是在继承层次中成员函数的函数原型完全相同。

### 11.4.2 利用虚函数实现动态绑定

多态主要体现在虚函数上，只要有虚函数存在，对象类型就会在程序运行时动态绑定。动态绑定的实现方法是定义一个指向基类对象的指针变量，并使它指向同一类族中需要调用该函数的对象，通过该指针变量调用此虚函数。

【例 11.7】 利用虚函数实现动态绑定。（实例位置：光盘\TM\sl\11\7）

```
#include<iostream>
using namespace std;
class CEmployee                                //定义 CEmployee 类
{
public:
    int m_ID;                                  //定义数据成员
    char m_Name[128];                          //定义数据成员
    char m_Depart[128];                       //定义数据成员
    CEmployee()                               //定义构造函数
    {
        memset(m_Name,0,128);                //初始化数据成员
        memset(m_Depart,0,128);              //初始化数据成员
    }
    virtual void OutputName()                 //定义一个虚成员函数
    {
        cout << "员工姓名: "<<m_Name << endl; //输出信息
    }
};
class COperator :public CEmployee             //从 CEmployee 类派生一个子类
{
```

```

public:
    char m_Password[128];           //定义数据成员
    void OutputName()               //定义 OutputName 虚函数
    {
        cout << "操作员姓名: "<<m_Name<< endl; //输出信息
    }
};
int main(int argc, char* argv[])
{
    CEmployee *pWorker = new COperator(); //定义 CEmployee 类型指针, 调用 COperator 类构造函数
    strcpy(pWorker->m_Name,"MR");         //设置 m_Name 数据成员信息
    pWorker->OutputName();                 //调用 COperator 类的 OutputName 成员函数
    delete pWorker;                       //释放对象
    return 0;
}

```

上述代码中, 在 CEmployee 类中定义了一个虚函数 OutputName, 在子类 COperator 中改写了 OutputName 成员函数, 其中 COperator 类中的 OutputName 成员函数即使没有使用 virtual 关键字仍为虚函数。并且定义一个 CEmployee 类型的指针, 调用 COperator 类的构造函数构造对象。

程序运行结果如图 11.8 所示。

从图 11.8 中可以发现, “pWorker->OutputName();” 语句调用的是 COperator 类的 OutputName 成员函数。虚函数有以下几方面限制。

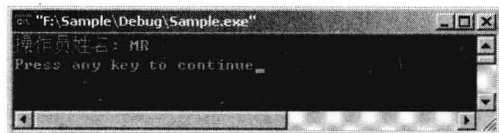


图 11.8 利用虚函数实现动态绑定

- (1) 只有类的成员函数才能为虚函数。
- (2) 静态成员函数不能是虚函数, 因为静态成员函数不受限于某个对象。
- (3) 内联函数不能是虚函数, 因为内联函数是不能在运行中动态确定其位置的。
- (4) 构造函数不能是虚函数, 析构函数通常是虚函数。

## 11.4.3 虚继承

11.3.1 节讲到从 CBird 类和 CFish 类派生子类 CWaterBird 时, 在 CWaterBird 类中将存在两个 CAnimal 类的复制。那么如何在派生 CWaterBird 类时使其只存在一个 CAnimal 基类呢? C++ 语言提供的虚继承机制, 解决了这个问题。

**【例 11.8】** 虚继承。(实例位置: 光盘\TM\sl\11\8)

```

#include<iostream>
using namespace std;
class CAnimal           //定义一个动物类
{
public:
    CAnimal()            //定义构造函数
    {
        cout << "动物类被构造"<< endl; //输出信息
    }
}

```



```

}
    void Move()                                //定义成员函数
    {
        cout << "动物能够移动"<< endl;        //输出信息
    }
};
class CBird : virtual public CAnimal            //从 CAnimal 类虚继承 CBird 类
{
public:
    CBird()                                    //定义构造函数
    {
        cout << "鸟类被构造"<< endl;          //输出信息
    }
    void FlyInSky()                            //定义成员函数
    {
        cout << "鸟能够在天空飞翔"<< endl;    //输出信息
    }
    void Breath()                             //定义成员函数
    {
        cout << "鸟能够呼吸"<< endl;          //输出信息
    }
};
class CFish: virtual public CAnimal            //从 CAnimal 类虚继承 CFish 类
{
public:
    CFish()                                    //定义构造函数
    {
        cout << "鱼类被构造"<< endl;          //输出信息
    }
    void SwimInWater()                        //定义成员函数
    {
        cout << "鱼能够在水里游"<< endl;      //输出信息
    }
    void Breath()                             //定义成员函数
    {
        cout << "鱼能够呼吸"<< endl;          //输出信息
    }
};
class CWaterBird: public CBird, public CFish    //从 CBird 类和 CFish 类派生子类 CWaterBird
{
public:
    CWaterBird()                              //定义构造函数
    {
        cout << "水鸟类被构造"<< endl;        //输出信息
    }
    void Action()                             //定义成员函数
    {
        cout << "水鸟既能飞又能游"<< endl;    //输出信息
    }
};

```



```

int main(int argc, char* argv[])           //主函数
{
    CWaterBird waterbird;                  //定义水鸟对象
    return 0;
}

```

程序运行结果如图 11.9 所示。

上述代码在定义 CBird 类和 CFish 类时使用了关键字 virtual 从基类 CAnimal 派生而来。实际上,虚继承对于 CBird 类和 CFish 类没有多少影响,却对 CWaterBird 类产生了很大影响。CWaterBird 类中不再有两个 CAnimal 类的复制,而只存在一个 CAnimal 的复制。

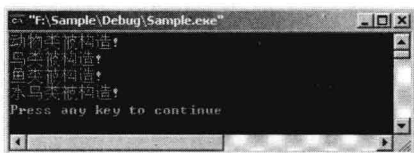


图 11.9 虚继承

通常,在定义一个对象时,先依次调用基类的构造函数,最后才调用自身的构造函数。但是对于虚继承来说,情况有些不同。在定义 CWaterBird 类对象时,先调用基类 CAnimal 的构造函数,然后调用 CBird 类的构造函数,这里 CBird 类虽然为 CAnimal 的子类,但是在调用 CBird 类的构造函数时将不再调用 CAnimal 类的构造函数。对于 CFish 类也是同样的道理。

在程序开发过程中,多继承虽然带来了很方便,但是很少有人愿意使用它,因为多继承会带来很多复杂的问题,并且它能够完成的功能通过单继承同样可以实现。如今流行的 C#、Delphi、Java 等面向对象语言没有提供多继承的功能,而是只采用单继承是经过设计者充分考虑的。因此,读者在开发应用程序时,如果能够使用单继承实现,尽量不要使用多继承。

## 11.5 抽 象 类

 视频讲解: 光盘\TM\lx\11\抽象类.exe

包含有纯虚函数的类称为抽象类,一个抽象类至少具有一个纯虚函数。抽象类只能作为基类派生出的新的子类,而不能在程序中被实例化(即不能说明抽象类的对象),但是可以使用指向抽象类的指针。在开发程序过程中并不是所有代码都是由软件构造师自己写的,有时需要调用库函数,有时分给别人写。一名软件构造师可以通过纯虚函数建立接口,然后让程序员填写代码实现接口,而自己主要负责建立抽象类。

### 11.5.1 纯虚函数

纯虚函数(Pure Virtual Function)是指被标明为不具体实现的虚成员函数,它不具备函数的功能。许多情况下,在基类中不能给虚函数一个有意义的定义,这时可以在基类中将它说明为纯虚函数,而其实现留给派生类去做。纯虚函数不能被直接调用,仅起到提供一个与派生类相一致的接口的作用。声明纯虚函数的形式为:

```
virtual 类型 函数名(参数表列)=0;
```

纯虚函数不可以被继承。当基类是抽象类时，在派生类中必须给出基类中纯虚函数的定义，或在该类中再声明其为纯虚函数。只有在派生类中给出了基类中所有纯虚函数的实现时，该派生类才不再成为抽象类。

**【例 11.9】** 创建纯虚函数。（实例位置：光盘\TM\sl\11\9）

```
#include<iostream>
using namespace std;
class CFigure
{
public:
    virtual double getArea() =0;
};
const double PI=3.14;
class CCircle : public CFigure
{
private:
    double m_dRadius;
public:
    CCircle(double dR){m_dRadius=dR;}
    double getArea()
    {
        return m_dRadius*m_dRadius*PI;
    }
};
class CRectangle : public CFigure
{
protected:
    double m_dHeight,m_dWidth;
public:
    CRectangle(double dHeight,double dWidth)
    {
        m_dHeight=dHeight;
        m_dWidth=dWidth;
    }
    double getArea()
    {
        return m_dHeight*m_dWidth;
    }
};
void main()
{
    CFigure *fg1;
    fg1= new CRectangle(4.0,5.0);
    cout << fg1->getArea() << endl;
    delete fg1;
    CFigure *fg2;
    fg2= new CCircle(4.0);
    cout << fg2->getArea() << endl;
```

```
delete fg2;
}
```

程序运行结果如图 11.10 所示。

程序定义了矩形类 `CRectangle` 和圆形类 `CCircle`，两个类都派生于图形类 `CFigure`。图形类是一个在现实生活中不存在的对象，抽象类面积的计算方法不确定，所以，将图形类 `CFigure` 的面积计算方法设置为纯虚函数，这样圆形有圆形面积的计算方法，矩形有矩形面积的计算方法，每个继承自 `CFigure` 的对象都有自己的面积，通过 `getArea` 成员函数就可以获取面积值。

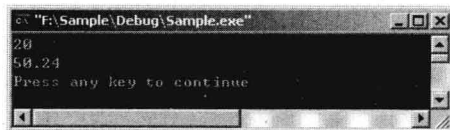


图 11.10 创建纯虚函数



### 注意

对于包含纯虚函数的类来说，是不能够实例化的，“`CFigure figure;`”是错误的。

## 11.5.2 实现抽象类中的成员函数

抽象类通常用于作为其他类的父类，从抽象类派生的子类如果是抽象类，则子类必须实现父类中的所有纯虚函数。

【例 11.10】 实现抽象类中的成员函数。（实例位置：光盘\TM\sl\11\10）

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

class CEmployee //定义 CEmployee 类
{
public:
    int m_ID; //定义数据成员
    char m_Name[128]; //定义数据成员
    char m_Depart[128]; //定义数据成员
    virtual void OutputName() = 0; //定义抽象成员函数
};

class COperator :public CEmployee //定义 COperator 类，派生于 CEmployee 类
{
public:
    char m_Password[128]; //定义数据成员
    void OutputName() //实现父类中的纯虚成员函数
    {
        cout << "操作员姓名: "<<m_Name<< endl; //输出信息
    }
    COperator() //定义 COperator 类的默认构造函数
    {
        strcpy(m_Name,"MR"); //设置数据成员 m_Name 信息
    }
};
```



```

};
class CSystemManager :public CEmployee           //定义 CSystemManager 类
{
public:
    char m_Password[128];                        //定义数据成员
    void OutputName()                            //实现父类中的纯虚成员函数
    {
        cout << "系统管理员姓名: "<<m_Name<< endl; //输出信息
    }
    CSystemManager()                             //定义 CSystemManager 类的默认构造函数
    {
        strcpy(m_Name,"SK");                     //设置数据成员 m_Name 信息
    }
};

int main(int argc, char* argv[])                //主函数
{
    CEmployee *pWorker;                          //定义 CEmployee 类型指针对象
    pWorker = new COperator();                   //调用 COperator 类的构造函数, 为 pWorker 赋值
    pWorker->OutputName();                       //调用 COperator 类的 OutputName 成员函数
    delete pWorker;                             //释放 pWorker 对象
    pWorker = NULL;                             //将 pWorker 对象设置为空
    pWorker = new CSystemManager();              //调用 CSystemManager 类的构造函数, 为 pWorker 赋值
    pWorker->OutputName();                       //调用 CSystemManager 类的 OutputName 成员函数
    delete pWorker;                             //释放 pWorker 对象
    pWorker = NULL;                             //将 pWorker 对象设置为空
    return 0;
}

```

程序中从 CEmployee 类派生了两个子类, 分别为 COperator 和 CSystemManager。这两个类分别实现了父类的纯虚成员函数 OutputName。同样的一条语句 “pWorker->OutputName();”, 由于 pWorker 指向的对象不同, 其行为也不同。程序运行结果如图 11.11 所示。



图 11.11 实现抽象类中的成员函数

## 11.6 小 结

本章介绍了面向对象程序设计中的关键技术——继承与派生, 继承和派生在使用上还涉及二义性、访问顺序、运算符重载等许多技术问题, 正确理解和处理这些技术有利于掌握继承的使用方法。继承中还涉及多重继承, 这增加了面向对象开发的灵活性。面向对象可以建立抽象类, 由抽象类派生新类, 可以形成对类的一定管理。

## 11.7 实践与练习

1. 开发一个程序，定义一个类，包含“商品”和“售价”两个成员变量，要求重载运算符“<<”，可对类中的成员进行输出。(答案位置：光盘\TM\sl\11\11)

2. 开发一个程序，分别定义教师类 (Teacher) 和职位类 (Level)，采用多重继承方式由这两个类派生出新类 Teacher\_Level (教师和职位信息)。

要求：

(1) 在 Teacher 类中包含“职称”数据成员，在 Level 类中包含“职务”数据成员，在 Teacher\_Level 类中包含“工资”数据成员。

(2) 在 Teacher\_Level 类中使用 Show 函数将信息输出。(答案位置：光盘\TM\sl\11\12)





# 第 3 篇

## 高级应用

- » 第 12 章 模板
- » 第 13 章 STL 标准模板库
- » 第 14 章 RTTI 与异常处理
- » 第 15 章 程序调试
- » 第 16 章 文件操作
- » 第 17 章 网络通信

模板是 STL 的基础，通过对模板的介绍，使读者能够理解 STL 的构造。文件操作也是程序开发过程中必不可少的技术，掌握文件操作是奠定开发大项目的基础，通过对 RTTI 的介绍使读者对面向对象开发有更深入的理解。网络通信是仅次于文件技术的另一个关键技术，通过实例读者可以掌握基本的网络通信。



# 第12章

---

## 模板

(  视频讲解：49 分钟 )

模板是 C++ 语言的高级特性，分为函数模板和类模板两大类。模板使程序员能够快速建立具有类型安全的类库集合和函数集合，它的实现大大方便了大规模软件开发。对于程序员来说，要完全掌握 C++ 模板的用法并不容易。本章将介绍 C++ 模板的基本概念、函数模板和类模板，使读者有效地掌握模板的用法，正确使用 C++ 系统日益庞大的标准模板库 STL。

通过阅读本章，您可以：

- » 掌握函数模板
- » 掌握类模板
- » 了解链表
- » 掌握链表模板

## 12.1 函数模板

 视频讲解：光盘\TM\lx\12\函数模板.exe

函数模板不是一个实在的函数，编译器不能为其生成可执行代码。定义函数模板后只是一个对函数功能框架的描述，当它具体执行时，将根据传递的实际参数决定其功能。

### 12.1.1 函数模板的定义

函数模板定义的一般形式如下：

```
template <类型形式参数表>
返回类型 函数名(形式参数表)
{
    ...    //函数体
}
```

template 为关键字，表示定义一个模板；尖括号“<>”表示模板参数，模板参数主要有两种，一种是模板类型参数，另一种是模板非类型参数。模板类型参数使用关键字 class 或 typedef 开始，其后是一个用户定义的合法标识符。模板非类型参数与普通参数定义相同，通常为一个常数。

可以将声明函数模板分成 template 部分和函数名部分。例如：

```
template<class T>
void fun(T t)
{
    ...    //函数实现
}
```

定义一个求和的函数模板，例如：

```
template <class type>           //定义一个模板类型
type Sum(type xvar,type yvar)  //定义函数模板
{
    return xvar + yvar;
}
```

在定义完函数模板之后，需要在程序中调用函数模板。下面的代码演示了 Sum 函数模板的调用。

```
int iret = Sum(10,20);           //实现两个整数的相加
double dret = Sum(10.5,20.5);    //实现两个实数的相加
```

如果采用如下的形式调用 Sum 函数模板，将会出现错误。

```
int iret = Sum(10.5,20);          //错误的调用
double dret = Sum(10,20.5);       //错误的调用
```



上述代码中为函数模板传递了两个类型不同的参数，编译器产生了歧义。如果用户在调用函数模板时显式标识模板类型，就不会出现错误了。例如：

```
int iret = Sum<int>(10.5,20);           //正确地调用函数模板
double dret = Sum<double>(10,20.5);     //正确地调用函数模板
```

用函数模板生成实际可执行的函数又称为模板函数。函数模板与模板函数不是一个概念。从本质上讲，函数模板是一个“框架”，它不是真正可以编译生成代码的程序，而模板函数是把函数模板中的类型参数实例化后生成的函数，它和普通函数本质是相同的，可以生成可执行代码。

### 12.1.2 函数模板的作用

假设求两个函数之中最大者，如果想求整型数和实型数需要定义以下两个函数：

```
int max(int a, int b)
{
    return a>b?a: b;           //返回最大值
}
float max(float a, float b)
{
    return a>b?a: b;           //返回最大值
}
```

能不能通过一个 max 函数来完成既求整型数之间最大者又求实型数之间最大者呢？可以使用函数模板以及#define 宏定义实现。

#define 宏定义可以在预编译期对代码进行替换。例如：

```
#define max(a,b) ((a) > (b) ? (a) : (b))
```

上述代码可以求整型数最大值和实型数最大值。但宏定义#define 只是进行简单替换，它无法对类型进行检查，有时计算结果可能不是预计的。例如：

```
#include<iostream>
#include<iomanip>
using namespace std;
#define max(a,b) ((a) > (b) ? (a) : (b))
void main()
{
    int m=0,n=0;
    cout << max(m,++n) << endl;
    cout << m << setw(2) << endl;
}
```

程序运行结果如图 12.1 所示。

程序运行的预期结果应该是 1 和 0，为什么输出这样的结果呢？原因在于宏替换之后“++n”被执行了两次，因此 n 的值是 2 不是 1。

宏是预编译指令，很难调试，无法单步进入宏的代码中。模板函数和#define 宏定义相似，但模板函数是用模板实例化得到的函数，它与普通函数没有本质区别，可以重载模板函数。

使用模板求最大值的代码如下：

```
template<class Type>
type max(Type a,Type b)
{
    if(a > b)
        return a;
    else
        return b;
}
```

调用模板函数 max 可以正确计算整型数和实型数的最大值。例如：

```
cout << "最大值: " << max(10,1) << endl;
cout << "最大值: " << max(200.05,100.4) << endl;
```

**【例 12.1】** 使用数组作为模板参数。（实例位置：光盘\TM\sl\12\1）

```
#include<iostream>
using namespace std;
template <class type,int len>           //定义一个模板类型
type Max(type array[len])             //定义函数模板
{
    type ret = array[0];               //定义一个变量
    for(int i=1; i<len; i++)           //遍历数组元素
    {
        ret = (ret > array[i])? ret : array[i]; //比较数组元素大小
    }
    return ret;                        //返回最大值
}
void main()
{
    int array[5] = {1,2,3,4,5};        //定义一个整型数组
    int iret = Max<int,5>(array);       //调用函数模板 Max
    double dset[3] = {10.5,11.2,9.8};  //定义实数数组
    double dret = Max<double,3>(dset);  //调用函数模板 Max
    cout << dret << endl;
}
```

程序运行结果如图 12.2 所示。

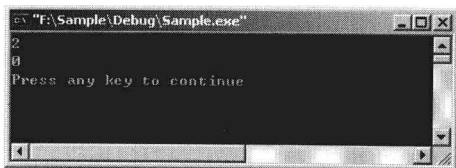


图 12.1 利用宏定义求最大值

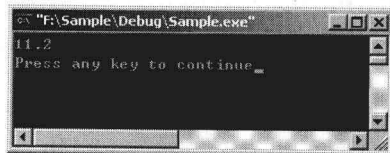


图 12.2 使用数组作为模板参数

程序中定义一个函数模板 Max, 用来求数组中元素的最大值, 其中模板参数使用模板类型参数 type 和模板非类型参数 len, 参数 type 声明了数组中的元素类型, 参数 len 声明了数组中的元素个数, 给定数组元素后, 程序将数组中的最大值输出。

### 12.1.3 重载函数模板

整型数和实型数编译器可以直接进行比较, 所以使用函数模板后也可以直接进行比较, 但如果是字符指针指向的字符串该如何比较呢? 答案是通过重载函数模板来实现。通常字符串需要使用库函数来进行比较, 下面介绍通过重载函数模板实现字符串的比较。

【例 12.2】 求出字符串的最小值。(实例位置: 光盘\TM\sl\12\2)

```
#include<iostream >
#include<string >
using namespace std;
template<class Type>
type min(Type a,Type b)           //定义函数模板
{
    if(a < b)
        return a;
    else
        return b;
}
char * min(char * a,char * b)     //重载函数模板
{
    if(strcmp(a,b))
        return b;
    else
        return a;
}
void main()
{
    cout << "最小值: " << min(10,1) << endl;
    cout << "最小值: " << min('a','b') << endl;
    cout << "最小值: " << min("hi","mr") << endl;
}
```

程序运行结果如图 12.3 所示。



图 12.3 求出字符串的最小值

程序在重载的函数模板 min 的实现中, 使用 strcmp 库函数来完成字符串的比较, 此时使用 min 函数可以比较整型数据、实型数据、字符数据和字符串数据。

## 12.2 类 模 板

 视频讲解：光盘\TM\lx\12\类模板.exe

使用 `template` 关键字不但可以定义函数模板，也可以定义类模板。类模板代表一族类，是用来描述通用数据类型或处理方法的机制，它使类中的一些数据成员和成员函数的参数或返回值可以取任意数据类型。类模板可以说是用类生成类，减少了类的定义数量。

### 12.2.1 类模板的定义与声明

类模板的一般定义形式如下：

```
template <类型形式参数表>
class 类模板名
{
...    //类模板体
};
```

类模板成员函数的定义形式如下：

```
template <类型形式参数表>
返回类型 类模板名 <类型名表>::成员函数名(形式参数列表)
{
...    //函数体
}
```

`template` 是关键字，类型形式参数表与函数模板定义相同。类模板的成员函数定义时的类模板名与类模板定义时要一致，类模板不是一个真实的类，需要重新生成类，生成类的形式如下：

```
类模板名<类型实在参数表>
```

用新生成的类定义对象的形式如下：

```
类模板名<类型实在参数表> 对象名
```

其中类型实在参数表应与该类模板中的类型形式参数表匹配。用类模板生成的类称为模板类。类模板和模板类不是同一个概念，类模板是模板的定义，不是真实的类，定义中要用到类型参数；模板类本质上与普通类相同，它是类模板的类型参数实例化之后得到的类。

定义一个容器的类模板，代码如下：

```
template<class Type>
class Container
{
    Type titem;
```



```

public:
Container(){ };
void begin(const Type& tNew);
void end(const Type& tNew);
void insert(const Type& tNew);
void empty(const Type& tNew);
};

```

和普通类一样，需要对类模板成员函数进行定义，代码如下：

```

void Container<type>::begin(const Type& tNew)           //容器的第一个元素
{
    tItem=tNew;
}
void Container<type>::end(const Type& tNew)             //容器的最后一个元素
{
    tItem=tNew;
}
void Container<type>::insert(const Type& tNew)          //向容器中插入元素
{
    tItem=tNew;
}
void Container<type>::empty(const Type& tNew)           //清空容器
{
    tItem=tNew;
}

```

将模板类的参数设置为整型，然后用模板类声明对象，代码如下：

```

Container<int> myContainer;                             //声明 Container<int>类对象

```

声明对象后，就可以调用类成员函数，代码如下：

```

int i=10;
myContainer.insert(i);

```

在类模板定义中，类型形式参数表中的参数也可以是其他类模板。例如：

```

template<template<class A> class B>
class CBase
{
private:
    B<int> m_n;
}

```

类模板也可以进行继承。例如：

```

template<class T>
class CDerived public T
{
public:
    CDrived();
}

```



```
};
template<class T>
CDerived<T>::CDerived() : T()
{
    cout << "" << endl;
}
void main()
{
    CDerived<CBase1> D1;
    CDerived<CBase1> D1;
}
```

T 是一个类，CDerived 继承自该类，CDerived 可以对类 T 进行扩展。

## 12.2.2 简单类模板

类模板中的类型形式参数表可以在执行时指定，也可以在定义类模板时指定。下面介绍类型参数如何在执行时指定。

**【例 12.3】** 简单类模板。（实例位置：光盘\TM\sl\12\3）

```
#include<iostream>
using namespace std;
template<class T1,class T2>
class MyTemplate
{
    T1 t1;
    T2 t2;
public:
    MyTemplate(T1 tt1,T2 tt2)
    {t1 =tt1, t2=tt2;}
    void display()
    { cout << t1 << ' ' << t2 << endl;}
};
void main()
{
    int a=123;
    double b=3.1415;
    MyTemplate<int ,double> mt(a,b);
    mt.display();
}
```

程序运行结果如图 12.4 所示。



图 12.4 简单类模板

程序中的 MyTemplate 是一个模板类，它使用整型和双精度型作为参数。

### 12.2.3 默认模板参数

默认模板参数就是在类模板定义时设置类型形式参数表中一个类型参数的默认值，该默认值是一个数据类型。有默认的数据类型参数后，在定义模板新类时就可以不进行指定。

【例 12.4】 默认模板参数。（实例位置：光盘\TM\sl\12\4）

```
#include<iostream>
using namespace std;
template<class T1,class T2 = int>
class MyTemplate
{
    T1 t1;
    T2 t2;
public:
    MyTemplate(T1 tt1,T2 tt2)
    {t1=tt1;t2=tt2;}
    void display()
    {
        cout<< t1 << ' ' << t2 << endl;
    }
};
void main()
{
    int a=123;
    double b=3.1415;
    MyTemplate<int ,double> mt1(a,b);
    MyTemplate<int> mt2(a,b);
    mt1.display();
    mt2.display();
}
```

程序运行结果如图 12.5 所示。

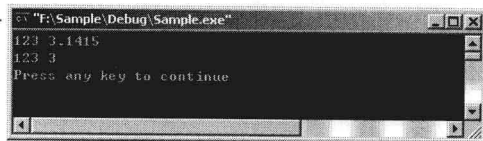


图 12.5 默认模板参数

### 12.2.4 为具体类型的参数提供默认值

默认模板参数是类模板中由默认的数据类型作参数，在模板定义时还可以为默认的数据类型声明变量，并且为变量赋值。

【例 12.5】 为具体类型的参数提供默认值。（实例位置：光盘\TM\sl\12\5）

```
#include<iostream>
using namespace std;
template<class T1,class T2,int num= 10 >
```

```

class MyTemplate
{
    T1 t1;
    T2 t2;
public:
    MyTemplate(T1 tt1,T2 tt2)
    {t1 =tt1+num, t2=tt2+num;}
    void display()
    { cout << t1 << ' ' << t2 <<endl;}
};
void main()
{
    int a=123;
    double b=3.1415;
    MyTemplate<int ,double> mt1(a,b);
    MyTemplate<int ,double ,100> mt2(a,b);
    mt1.display();
    mt2.display();
}

```

程序运行结果如图 12.6 所示。

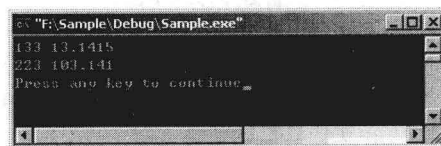


图 12.6 为具体类型的参数提供默认值

### 12.2.5 有界数组模板

C++语言不能检查数组下标是否越界，如果下标越界会造成程序崩溃，程序员在编辑代码时很难找到下标越界错误。那么如何能让数组进行下标越界检测呢？答案是建立数组模板，在模板定义时对数组的下标进行检查。

在模板中想要获取下标值，需要重载数组下标运算符“[]”，重载数组下标运算符后使用模板类实例化的数组，就可以进行下标越界检测了。例如：

```

#include<cassert>
template<class T,int b>
class Array
{
    T& operator[] (int sub)
    {
        assert(sub>=0&& sub<b);
    }
};

```

程序中使用了 `assert` 来进行警告处理，当有下标越界情况发生时就弹出对话框警告，然后输出出现错误的代码位置。`assert` 函数需要使用 `cassert` 头文件。

数组模板的应用实例如下：

```

#include<iostream>
#include<iomanip>

```



```
#include<cassert>
using namespace std;

class Date
{
    int iMonth,iDay,iYear;
    char Format[128];
public:
    Date(int m=0,int d=0,int y=0)
    {
        iMonth=m;
        iDay=d;
        iYear=y;
    }
    friend ostream& operator<<(ostream& os,const Date t)
    {
        cout << "Month: " << t.iMonth << ' ';
        cout << "Day: " << t.iDay << ' ';
        cout << "Year: " << t.iYear << ' ';
        return os;
    }
    void Display()
    {
        cout << "Month: " << iMonth;
        cout << "Day: " << iDay;
        cout << "Year: " << iYear;
        cout << endl;
    }
};

template<class T,int b>
class Array
{
    T elem[b];
public:
    Array(){ }
    T& operator[ ] (int sub)
    {
        assert(sub>=0&& sub<b);
        return elem[sub];
    }
};

void main()
{
    Array<Date,3> dateArray;
    Date dt1(1,2,3);
    Date dt2(4,5,6);
```

```

    Date dt3(7,8,9);
    dateArray[0]=dt1;
    dateArray[1]=dt2;
    dateArray[2]=dt3;
    for(int i=0;i<3;i++)
        cout << dateArray[i] << endl;
    Date dt4(10,11,13);
    dateArray[3] = dt4;           //弹出警告
    cout << dateArray[3] << endl;
}

```

程序运行结果如图 12.7 所示。

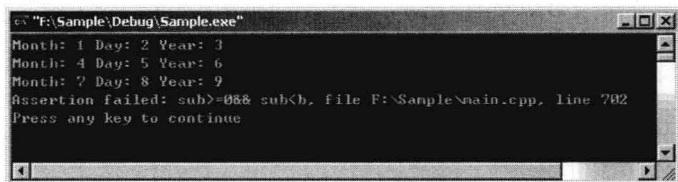


图 12.7 数组模板

程序能够及时发现 dateArray 已经越界，因为定义数组时指定数组的长度为 3，当数组下标为 3 时说明数组中有 4 个元素，所以程序执行到 dateArray[3]时，弹出错误警告。

## 12.3 模板的使用

### 视频讲解：光盘\TM\1x\12\模板的使用.exe

定义完模板类后如果想扩展模板新类的功能，需要对类模板进行覆盖，使模板类能够完成特殊功能。覆盖操作可以针对整个类模板、部分类模板以及类模板的成员函数。这种覆盖操作称为定制。

### 12.3.1 定制类模板

定制一个类模板，然后覆盖类模板中所定义的所有成员。

**【例 12.6】** 定制类模板。（实例位置：光盘\TM\sl\12\6）

```

#include<iostream>
using namespace std;
class Date
{
    int iMonth,iDay,iYear;
    char Format[128];
public:
    Date(int m=0,int d=0,int y=0)
    {

```



```
iMonth=m;
iDay=d;
iYear=y;
}
friend ostream& operator<<(ostream& os,const Date t)
{
    cout << "Month: " << t.iMonth << ' ';
    cout << "Day: " << t.iDay<< ' ';
    cout << "Year: " << t.iYear<< ' ';
    return os;
}
void Display()
{
    cout << "Month: " << iMonth;
    cout << "Day: " << iDay;
    cout << "Year: " << iYear;
    cout << endl;
}
};

template<class T>
class Set
{
    T t;
public:
    Set(T st) : t(st) {}
    void Display()
    {
        cout << t << endl;
    }
};

class Set<Date>
{
    Date t;
public:
    Set(Date st): t(st){ }
    void Display()
    {
        cout << "Date : " << t << endl;
    }
};

void main()
{
    Set<int> intset(123);
    Set<Date> dt =Date(1,2,3);
    intset.Display();
    dt.Display();
}
```

程序运行结果如图 12.8 所示。

程序中定义了 Set 类模板, 该模板中有一个构造函数和一个 Display 成员函数。Display 成员函数负责输出成员的值。使用类 Date 定制了整个类模板, 也就是说模板类中构造函数中的参数是 Date 对象, Display 成员函数输出的也是 Date 对象。定制类模板相当于实例化一个模板类。

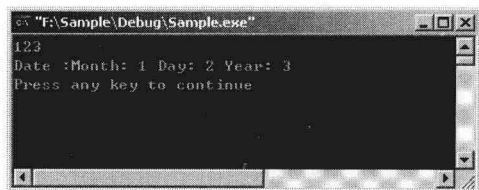


图 12.8 定制类模板

### 12.3.2 定制类模板成员函数

定制一个类模板, 然后覆盖类模板中指定的成员。

**【例 12.7】** 定制类模板成员函数。(实例位置: 光盘\TM\sl\12\7)

```
#include<iostream>
using namespace std;
class Date
{
    int iMonth,iDay,iYear;
    char Format[128];
public:
    Date(int m=0,int d=0,int y=0)
    {
        iMonth=m;
        iDay=d;
        iYear=y;
    }
    friend ostream& operator<<(ostream& os,const Date t)
    {
        cout << "Month: " << t.iMonth << ' ';
        cout << "Day: " << t.iDay << ' ';
        cout << "Year: " << t.iYear << ' ';
        return os;
    }
    void Display()
    {
        cout << "Month: " << iMonth;
        cout << "Day: " << iDay;
        cout << "Year: " << iYear;
        cout << std::endl;
    }
};
template<class T>
class Set
{
    T t;
public:
```

```

        Set(T st) : t(st) {}
        void Display();
};
template <class T>
void Set<T>::Display()
{
    cout << t << endl;
}
void Set<Date>::Display()
{
    cout << "Date: " << t << endl;
}
void main()
{
    Set<int> intset(123);
    Set<Date> dt = Date(1,2,3);
    intset.Display();
    dt.Display();
}

```

程序运行结果如图 12.9 所示。

程序中定义了 Set 类模板，该模板中有一个构造函数和一个 Display 成员函数。程序对模板类中的 Display 函数进行覆盖，使其参数类型设置为 Date 类，这样在使用 Display 函数输出时就会调用 Date 类中的 Display 函数进行输出。

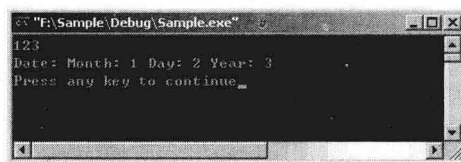


图 12.9 定制类模板成员函数

### 12.3.3 模板部分定制

定制一个类模板，然后覆盖类模板类型参数表中的一个参数。

**【例 12.8】** 模板部分定制。（实例位置：光盘\TM\sl\12\8）

```

#include<iostream>
using namespace std;
template <class T1,class T2>
class MyTemplate
{
    T1 obj1;
    T2 obj2;
public:
    MyTemplate(T1 o1,T2 o2) : obj1(o1),obj2(o2){}
    void display()
    {
        cout << "Object Display" << endl;
        cout << "Object 1:" << obj1 << endl;
        cout << "Object 2:" << obj2 << endl;
        cout << endl;
    }
}

```



```

};
template<class T>
class MyTemplate<T, char>
{
    T obj1,obj2;
public:
    MyTemplate(T o1,char c) : obj1(o1),obj2(o1)
    {obj2+=(int)c;}
    void display()
    {
        cout << "Object Display" << endl;
        cout << "Object 1:" << obj1 << endl;
        cout << "Object 2:" << obj2 << endl;
        cout << endl;
    }
};
int main()
{
    MyTemplate<int,int>mt1(10,20);
    MyTemplate<int,int>mt2(10,'b');
    mt1.display();
    mt2.display();
    return 1;
}

```

程序运行结果如图 12.10 所示。

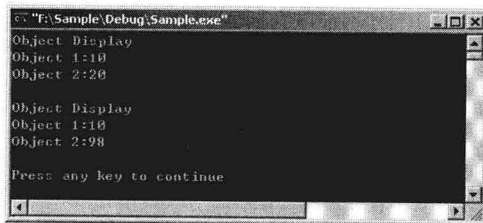


图 12.10 模板部分定制

程序中的 MyTemplate 类模板的一个参数被覆盖为 char，在模板类 MyTemplate 的构造函数中，用第一个参数的对象和 char 值相加。如果第一个参数被设置为 int 类型，那么 char 可以转换为 int 类型，完成和第一个参数的实例值的相加。

## 12.4 链表类模板

 视频讲解：光盘\TM\12\链表类模板.exe

链表是一种常用的数据结构，创建链表类模板就是创建一个对象的容器，在容器内可以对不同类型的对象进行插入、删除和排序等操作。C++标准模板中有链表类模板，本节将主要实现简单的链表

类模板。

### 12.4.1 链表

在介绍类模板之前，先来设计一个简单的单向链表。链表的功能包括向尾节点添加数据、遍历链表中的节点和在链表结束时释放所有节点。例如定义一个链表类，代码如下：

```
class CNode                                     //定义一个节点类
{
public:
    CNode *m_pNext;                             //定义一个节点指针，指向下一个节点
    int m_Data;                                  //定义节点的数据
    CNode()                                       //定义节点类的构造函数
    {
        m_pNext = NULL;                         //将 m_pNext 设置为空
    }
};
class CList                                     //定义链表类 CList
{
private:
    CNode *m_pHeader;                           //定义头节点
    int m_NodeSum;                               //节点数量
public:
    CList()                                       //定义链表的构造函数
    {
        m_pHeader = NULL;                       //初始化 m_pHeader
        m_NodeSum = 0;                           //初始化 m_NodeSum
    }
    CNode* MoveTrail()                           //移动到尾节点
    {
        CNode* pTmp = m_pHeader;                //定义一个临时节点，将其指向头节点
        for(int i=1;i<m_NodeSum;i++)             //遍历节点
        {
            pTmp = pTmp->m_pNext;                //获取下一个节点
        }
        return pTmp;                             //返回尾节点
    }
    void AddNode(CNode *pNode)                   //添加节点
    {
        if(m_NodeSum == 0)                       //判断链表是否为空
        {
            m_pHeader = pNode;                   //将节点添加到头节点中
        }
        else                                       //链表不为空
        {
            CNode* pTrail = MoveTrail();         //搜索尾节点
            pTrail->m_pNext = pNode;              //在尾节点处添加节点
        }
    }
};
```



```

    }
    m_NodeSum++; //使链表节点数量加 1
}
void PassList() //遍历链表
{
    if(m_NodeSum > 0) //判断链表是否为空
    {
        CNode* pTmp = m_pHeader; //定义一个临时节点, 将其指向头节点
        printf("%4d", pTmp->m_Data); //输出节点数据
        for(int i=1; i<m_NodeSum; i++) //遍历其他节点
        {
            pTmp = pTmp->m_pNext; //获取下一个节点
            printf("%4d", pTmp->m_Data); //输出节点数据
        }
    }
}
~CList() //定义链表析构函数
{
    if(m_NodeSum > 0) //链表不为空
    {
        CNode *pDelete = m_pHeader; //定义一个临时节点, 指向头节点
        CNode *pTmp = NULL; //定义一个临时节点
        for(int i=0; i<m_NodeSum; i++) //遍历节点
        {
            pTmp = pDelete->m_pNext; //获取下一个节点
            delete pDelete; //释放当前节点
            pDelete = pTmp; //将下一个节点设置为当前节点
        }
        m_NodeSum = 0; //将 m_NodeSum 设置为 0
        pDelete = NULL; //将 pDelete 设置为空
        pTmp = NULL; //将 pTmp 设置为空
    }
    m_pHeader = NULL; //将 m_pHeader 设置为空
}
};

```

链表类 CList 以 CNode 作为元素, 通过 MoveTrail 成员函数将链表指针移动到末尾, 通过 AddNode 成员函数添加一个节点。

下面声明一个链表对象, 向其中添加节点, 并遍历链表节点。代码如下:

```

int main(int argc, char* argv[])
{
    CList list; //定义链表对象
    for(int i=0; i<5; i++) //利用循环向链表中添加 5 个节点
    {
        CNode *pNode = new CNode(); //构造节点对象
        pNode->m_Data = i; //设置节点数据
        list.AddNode(pNode); //添加节点到链表
    }
}

```

```
list.PassList();           //遍历节点
cout << endl;             //输出换行
return 0;
}
```

程序运行结果如图 12.11 所示。

程序向链表中添加了 5 个元素, 然后调用 PassList 成员函数完成对链表元素的遍历。

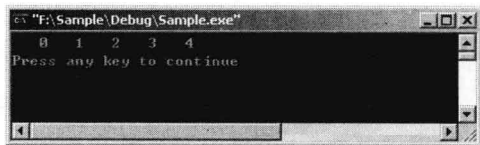


图 12.11 简单链表

## 12.4.2 链表类模板

链表类 CList 的一个最大缺陷就是链表不够灵活, 其节点只能是 CNode 类型。让 CList 能够适应各种类型的节点的最简单方法就是使用类模板。类模板的定义与函数模板类似, 以关键字 **template** 开始, 其后是由尖括号 “<>” 构成的模板参数。下面重新修改链表类 CList, 以类模板的形式进行改写, 代码如下:

```
template<class Type>           //定义类模板
class CList                    //定义 CList 类
{
private:
    Type *m_pHeader;           //定义头节点
    int m_NodeSum;             //节点数量
public:
    CList()                    //定义构造函数
    {
        m_pHeader = NULL;      //将 m_pHeader 置为空
        m_NodeSum = 0;         //将 m_NodeSum 置为 0
    }
    Type* MoveTrail()           //获取尾节点
    {
        Type *pTmp = m_pHeader; //定义一个临时节点, 将其指向头节点
        for(int i=1; i<m_NodeSum; i++) //遍历链表
        {
            pTmp = pTmp->m_pNext; //将下一个节点指向当前节点
        }
        return pTmp;           //返回尾节点
    }
    void AddNode(Type *pNode)    //添加节点
    {
        if(m_NodeSum == 0)      //判断链表是否为空
        {
            m_pHeader = pNode;  //在头节点处添加节点
        }
        else                    //链表不为空
        {
            Type* pTrail = MoveTrail(); //获取尾节点

```

```

        pTrail->m_pNext = pNode;           //在尾节点处添加节点
    }
    m_NodeSum++;                          //使节点数量加 1
}
void PassList()                          //遍历链表
{
    if(m_NodeSum > 0)                    //判断链表是否为空
    {
        Type* pTmp = m_pHeader;          //定义一个临时节点, 将其指向头节点
        printf("%4d", pTmp->m_Data);       //输出头节点数据
        for(int i=1; i<m_NodeSum; i++)    //利用循环访问节点
        {
            pTmp = pTmp->m_pNext;          //获取下一个节点
            printf("%4d", pTmp->m_Data);    //输出节点数据
        }
    }
}
~CList()                                 //定义析构函数
{
    if(m_NodeSum > 0)                    //判断链表是否为空
    {
        Type *pDelete = m_pHeader;       //定义一个临时节点, 将其指向头节点
        Type *pTmp = NULL;               //定义一个临时节点
        for(int i=0; i< m_NodeSum; i++)   //利用循环遍历所有节点
        {
            pTmp = pDelete->m_pNext;       //将下一个节点指向当前节点
            delete pDelete;                //释放当前节点
            pDelete = pTmp;                //将当前节点指向下一个节点
        }
        m_NodeSum = 0;                   //设置节点数量为 0
        pDelete = NULL;                  //将 pDelete 置为空
        pTmp = NULL;                     //将 pTmp 置为空
    }
    m_pHeader = NULL;                    //将 m_pHeader 置为空
}
};

```

上述代码利用类模板对链表类 CList 进行了修改, 实际上是在原来链表的基础上将链表中出现 CNode 类型的地方替换为模板参数 Type。下面再定义一个节点类 CNet, 演示模板类 CList 是如何适应不同的节点类型的。

**【例 12.9】** 使用 CList 类模板。(实例位置: 光盘\TM\sl\12\9)

```

class CNet                               //定义一个节点类
{
public:
    CNet *m_pNext;                       //定义一个节点类指针
    char m_Data;                          //定义节点类的数据成员
    CNet()                               //定义构造函数
    {

```



```

        m_pNext = NULL;                //将 m_pNext 置为空
    }
};
int main(int argc, char* argv[])
{
    CList<CNode> nodelist;              //构造一个类模板实例
    for(int n=0; n<5; n++)              //利用循环向链表中添加节点
    {
        CNode *pNode = new CNode();    //创建节点对象
        pNode->m_Data = n;              //设置节点数据
        nodelist.AddNode(pNode);        //向链表中添加节点
    }
    nodelist.PassList();                //遍历链表
    cout << endl;                      //输出换行
    CList<CNet> netlist;                //构造一个类模板实例
    for(int i=0; i<5; i++)              //利用循环向链表中添加节点
    {
        CNet *pNode = new CNet();      //创建节点对象
        pNode->m_Data = 97+i;           //设置节点数据
        netlist.AddNode(pNode);         //向链表中添加节点
    }
    netlist.PassList();                //遍历链表
    cout << endl;                      //输出换行
    return 0;
}

```

程序运行结果如图 12.12 所示。

类模板 CList 虽然能够使用不同类型的节点，但是对节点的类型是有一定要求的。第一，节点类必须包含一个指向自身的指针类型成员 m\_pNext，因为在 CList 中访问了 m\_pNext 成员；第二，节点类中必须包含数据成员 m\_Data，其类型被限制为数字类型或有序类型。

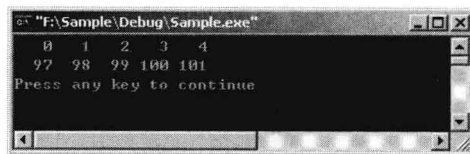


图 12.12 使用 CList 类模板

### 12.4.3 类模板的静态数据成员

在类模板中可以定义静态的数据成员，类模板中的每个实例都有自己的静态数据成员，而不是所有的类模板实例共享静态数据成员。为了说明这一点，笔者对模板类 CList 进行简化，向其中添加一个静态数据成员，并初始化静态数据成员。

【例 12.10】 在类模板中使用静态数据成员。（实例位置：光盘\TM\sl\12\10）

```

#include<iostream>
using namespace std;
template <class Type>
class CList                                //定义 CList 类
{
private:

```

```

    Type *m_pHeader;
    int   m_NodeSum;
public:
    static int m_ListValue;           //定义静态数据成员
    CList()
    {
        m_pHeader = NULL;
        m_NodeSum = 0;
    }
};

class CNode                          //定义 CNode 类
{
public:
    CNode *m_pNext;
    int   m_Data;
    CNode()
    {
        m_pNext = NULL;
    }
};

class CNet                          //定义 CNet 类
{
public:
    CNet *m_pNext;
    char  m_Data;
    CNet()
    {
        m_pNext = NULL;
    }
};

template <class Type>
int CList<Type>::m_ListValue = 10;   //初始化静态数据成员

int main(int argc, char* argv[])
{
    CList<CNode> nodelist;
    nodelist.m_ListValue = 2008;
    CList<CNet> netlist;
    netlist.m_ListValue = 88;
    cout<<nodelist.m_ListValue<< endl;
    cout<<netlist.m_ListValue<<endl;
    return 0;
}

```

程序运行结果如图 12.13 所示。

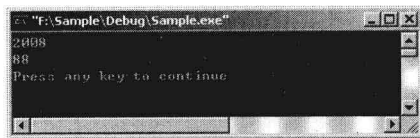


图 12.13 类模板的静态数据成员



由于模板实例 `nodelist` 和 `netlist` 均有各自的静态数据成员，所以 `m_ListValue` 的值是不同的。但是对于同一类型的模板实例，其静态数据成员是共享的。

## 12.5 小 结

模板是 C++ 语言的高级特性，一个模板可以定义一组函数或类，它使用数据类型和类名作为参数，建立具有类型安全的类库集合和函数集合。模板可以对作为模板参数的数据类型进行相同的操作，大大减少了代码量，提高了代码效率，更方便了大规模软件的开发。标准 C++ 库（STL）在很大程度上依赖于模板。通过本章的学习，可以使读者对 C++ 语言有更深入的了解。

## 12.6 实践与练习

1. 要求设计一个函数模板，该函数模板功能是计算两个数相加的结果，并将结果返回。（答案位置：光盘\TM\sl\12\11）
2. 设计一个类模板，在类模板中声明一个成员函数，用该函数得到两个数据，然后再设计成员函数 `max` 得到两个数据间的最大值。（答案位置：光盘\TM\sl\12\12）



# 第13章

---

## STL 标准模板库

(  视频讲解：35 分钟 )

STL 的英文全称为 Standard Template Library，主要目的是为标准化组件提供类模板进行范型编程。STL 技术是对原有 C++ 技术的一种补充，具有通用性好、效率高、数据结构简单、安全机制完善等特点。STL 是一些容器的集合，这些容器在算法库的支持下使程序开发变得更加简单和高效。

通过阅读本章，您可以：

- » 了解 STL
- » 掌握容器
- » 掌握算法
- » 掌握迭代器

## 13.1 序列容器

 视频讲解：光盘\TM\lx\13\序列容器.exe

STL 提供很多容器，每种容器都提供一组操作行为。序列容器（sequence）只提供插入功能，其中的元素都是有序的，但并未排序。序列容器包括 vector 向量、deque 双端队列和 list 链表。

### 13.1.1 向量类模板

向量（vector）是一种随机访问的数组类型，提供了对数组元素的快速、随机访问，以及在序列尾部快速、随机的插入和删除操作。它是大小可变的向量，在需要时可以改变其大小。

使用向量类模板需要创建 vector 对象，创建 vector 对象有以下几种方法：

☑ `std::vector<type> name;`

该方法创建了一个名为 name 的空 vector 对象，该对象可容纳类型为 type 的数据。例如，为整型值创建一个空 `std::vector` 对象可以使用这样的语句：

```
std::vector<int> intvector;
```

☑ `std::vector<type> name(size);`

该方法用来初始化具有 size 个元素的 vector 对象。

☑ `std::vector<type> name(size,value);`

该方法用来初始化具有 size 个元素的 vector 对象，并将对象的初始值设为 value。

☑ `std::vector<type> name(myvector);`

该方法使用复制构造函数，用现有的向量 myvector 创建了一个 vector 对象。

☑ `std::vector<type> name(first,last);`

该方法创建了元素在指定范围内的向量，first 代表起始范围，last 代表结束范围。

vector 对象的主要成员继承于随机接入容器和反向插入序列，主要成员函数及说明如表 13.1 所示。

表 13.1 vector 对象主要成员函数及说明

函 数	说 明
<code>assign(first,last)</code>	用迭代器 first 和 last 所辖范围内的元素替换向量元素
<code>assign(num,val)</code>	用 val 的 num 个副本替换向量元素
<code>at(n)</code>	返回向量中第 n 个位置元素的值
<code>back</code>	返回对向量末尾元素的引用
<code>begin</code>	返回指向向量中第一个元素的迭代器
<code>capacity</code>	返回当前向量最多可以容纳的元素个数
<code>clear</code>	删除向量中所有元素
<code>empty</code>	如果向量为空，则返回 true 值
<code>end</code>	返回指向向量中最后一个元素的迭代器

续表

函 数	说 明
erase(start,end)	删除迭代器 start 和 end 所辖范围内的向量元素
erase(i)	删除迭代器 i 所指向的向量元素
front	返回对向量起始元素的引用
insert(i,x)	把值 x 插入向量中由迭代器 i 所指明的位置
insert(i,start,end)	把迭代器 start 和 end 所辖范围内的元素插入到向量中由迭代器 i 所指明的位置
insert(i,n,x)	把 x 的 n 个副本插入到向量中由迭代器 i 所指明的位置
max_size	返回向量的最大容量（最多可以容纳的元素个数）
pop_back	删除向量最后一个元素
push_back(x)	把值 x 放在向量末尾
rbegin	返回一个反向迭代器，指向向量末尾元素之后
rend	返回一个反向迭代器，指向向量起始元素
reverse	颠倒元素的顺序
resize(n,x)	重新设置向量大小 n，新元素的值初始化为 x
size	返回向量的大小（元素的个数）
swap(vector)	交换两个向量的内容

下面通过实例进一步学习 vector 类模板的应用。

【例 13.1】 vector 类模板的应用。（实例位置：光盘\TM\sl\13\1）

```

#include "stdafx.h"
#include<iostream>
#include<vector>
#include<algorithm>
using namespace std;
void Output(char val)
{
    cout << val << ' ';
}
int main()
{
    vector<char> charVector;                //创建字符型向量
    charVector.push_back('Z');              //在向量中插入数据
    charVector.push_back('D');
    charVector.push_back('S');
    charVector.push_back('A');
    charVector.push_back('E');
    charVector.push_back('C');
    charVector.push_back('U');
    charVector.push_back('V');
    cout << "Contents of vector:";
    for_each(charVector.begin(),charVector.end(),Output); //循环并显示向量中的元素
    sort(charVector.begin(),charVector.end());           //对向量中的元素进行排序
    cout << std::endl<< "Contents of vector:";
}

```



```

    for_each(charVector.begin(),charVector.end(),Output); //循环并显示向量中的元素
    cout << endl;
    return 0;
}

```

程序运行结果如图 13.1 所示。

程序中利用 vector 对象创建字符型向量，并实现向量元素的插入和访问。

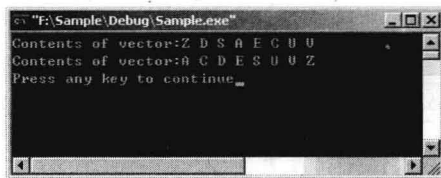


图 13.1 vector 类模板的应用

### 13.1.2 双端队列类模板

双端队列（deque）是一种随机访问的数据类型，提供了在序列两端快速插入和删除操作的功能，它可以在需要时修改其自身的大小，主要完成标准 C++ 数据结构中队列的功能。

使用双端队列类模板需要创建 deque 对象，创建 deque 对象有以下几种方法：

☑ `std::deque<type> name;`

该方法创建了一个名为 name 的空 deque 对象，该对象可容纳数据类型为 type 的数据。例如，为整型值创建一个空 `std::deque` 对象可以使用这样的语句：

```
std::deque<int> int deque;
```

☑ `std::deque<type> name(size);`

该方法创建一个大小为 size 的 deque 对象。

☑ `std::deque<type> name(size,value);`

该方法创建一个大小为 size 的 deque 对象，并将对象的每个值设为 value。

☑ `std::deque<type> name(mydeque);`

该方法使用复制构造函数，用现有的双端队列 mydeque 创建一个 deque 对象。

☑ `std::deque<type> name(first,last);`

该方法创建了元素在指定范围内的双端队列，first 代表起始范围，last 代表结束范围。

deque 对象的主要成员函数及说明如表 13.2 所示。

表 13.2 deque 对象主要成员函数及说明

函 数	说 明
<code>assign(first,last)</code>	用迭代器 first 和 last 所辖范围内的元素替换双端队列元素
<code>assign(num,val)</code>	用 val 的 num 个副本替换双端队列元素
<code>at(n)</code>	返回双端队列中第 n 个位置元素的值
<code>back</code>	返回一个对双端队列最后一个元素的引用
<code>begin</code>	返回指向双端队列中第一个元素的迭代器
<code>clear</code>	删除双端队列中所有元素
<code>empty</code>	如果双端队列为空，则返回 true 值
<code>end</code>	返回指向双端队列最后一个元素的迭代器
<code>erase(start,end)</code>	删除迭代器 start 和 end 所辖范围内的双端队列元素

续表

函 数	说 明
erase(i)	删除迭代器 i 所指向的双端队列元素
front	返回一个对双端队列第一个元素的引用
insert(i,x)	把值 x 插入向量中由迭代器 i 所指明的位置
insert(i,start,end)	把迭代器 start 和 end 所辖范围内的元素插入到双端队列中由迭代器 i 所指明的位置
insert(i,n,x)	把 x 的 n 个副本插入到双端队列中由迭代器 i 所指明的位置
max_size	返回双端队列的最大容量（最多可以容纳的元素个数）
pop_back	删除双端队列最后一个元素
pop_front	删除双端队列第一个元素
push_back(x)	把值 x 放在双端队列末尾
push_front(x)	把值 x 放在双端队列开始
rbegin	返回一个反向迭代器，指向双端队列最后一个元素之后
rend	返回一个反向迭代器，指向双端队列第一个元素
resize(n,x)	重新设置双端队列大小 n，新元素的值初始化为 x
size	返回双端队列的大小（元素的个数）
swap(vector)	交换两个双端队列的内容

## 【例 13.2】 双端队列类模板应用。（实例位置：光盘\TM\sl\13\2）

```

#include<iostream>
#include<deque>
using namespace std;
int main()
{
    deque<int > intdeque;
    intdeque.push_back(2);
    intdeque.push_back(3);
    intdeque.push_back(4);
    intdeque.push_back(7);
    intdeque.push_back(9);
    cout << "Deque: old" << endl;
    for(int i=0;i< intdeque.size();i++)
    {
        cout << "intdeque[" << i << "]:";
        cout << intdeque[i] << endl;
    }
    cout << endl;
    intdeque.pop_front();           //删除队头元素
    intdeque.pop_front();
    intdeque[1]=33;
    cout << "Deque: new" << endl;
    for(i=0;i<intdeque.size();i++)
    {
        cout << "intdeque[" << i << "]:";
        cout << intdeque[i] << " ";
    }
}

```

```
}  
cout << endl;  
return 0;  
}
```

程序运行结果如图 13.2 所示。

程序定义了一个空的类型为 `int` 的 `deque` 变量，然后用函数 `push_back` 把值插入到 `deque` 变量中，并把 `deque` 变量显示出来，最后删除 `deque` 变量中的第一个元素，并把删除后的 `deque` 变量中的第 2 个元素赋值。

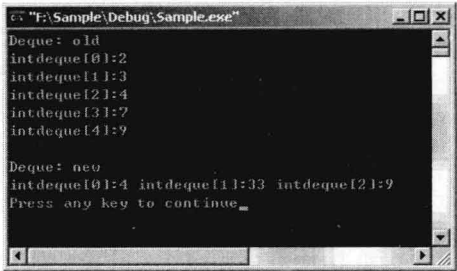


图 13.2 双端队列类模板应用

### 13.1.3 链表类模板

链表 (`list`)，即双向链表容器，它不支持随机访问，访问链表元素要指针从链表的某个端点开始，插入和删除操作所花费的时间是固定的，和该元素在链表中的位置无关。`list` 在任何位置插入和删除动作都很快，不像 `vector` 只在末尾进行操作。

使用链表类模板需要创建 `list` 对象，创建 `list` 对象有以下几种方法：

☒ `std::list<type> name;`

该方法创建了一个名为 `name` 的空 `list` 对象，该对象可容纳数据类型为 `type` 的数据。例如，为整型值创建一个空 `std::vector` 对象可以使用这样的语句：

```
std::list<int> intlist;
```

☒ `std::list<type> name(size);`

该方法初始化具有 `size` 个元素的 `list` 对象。

☒ `std::list<type> name(size,value);`

该方法初始化具有 `size` 个元素的 `list` 对象，并将对象的每个元素设为 `value`。

☒ `std::list<type> name(mylist);`

该方法使用复制构造函数，用现有的链表 `mylist` 创建了一个 `list` 对象。

☒ `std::list<type> name(first,last);`

该方法创建了元素在指定范围内的链表，`first` 代表起始范围，`last` 代表结束范围。

`list` 对象的主要成员函数及说明如表 13.3 所示。

表 13.3 `list` 对象主要成员函数及说明

函 数	说 明
<code>assign(first,last)</code>	用迭代器 <code>first</code> 和 <code>last</code> 所辖范围内的元素替换链表元素
<code>assign(num,val)</code>	用 <code>val</code> 的 <code>num</code> 个副本替换链表元素
<code>back</code>	返回一个对链表最后一个元素的引用
<code>begin</code>	返回指向链表中第一个元素的迭代器
<code>clear</code>	删除链表中所有元素
<code>empty</code>	如果链表为空，则返回 <code>true</code> 值

续表

函 数	说 明
end	返回指向链表最后一个元素的迭代器
erase(start,end)	删除迭代器 start 和 end 所辖范围内的链表元素
erase(i)	删除迭代器 i 所指向的链表元素
front	返回一个对链表第一个元素的引用
insert(i,x)	把值 x 插入链表中由迭代器 i 所指明的位置
insert(i,start,end)	把迭代器 start 和 end 所辖范围内的元素插入到链表中由迭代器 i 所指明的位置
insert(i,n,x)	把 x 的 n 个副本插入到链表中由迭代器 i 所指明的位置
max_size	返回链表的最大容量（最多可以容纳的元素个数）
pop_back	删除链表最后一个元素
pop_front	删除链表第一个元素
push_back(x)	把值 x 放在链表末尾
push_front(x)	把值 x 放在链表开始
rbegin	返回一个反向迭代器，指向链表最后一个元素之后
rend	返回一个反向迭代器，指向链表第一个元素
resize(n,x)	重新设置链表大小 n，新元素的值初始化为 x
reverse	颠倒链表元素的顺序
size	返回链表的大小（元素的个数）
swap(listref)	交换两个链表的内容

## 【例 13.3】 链表类模板应用。（实例位置：光盘\TM\sl\13\3）

```

#include<iostream>
#include<list>
using namespace std;
int main()
{
    char cTemp;
    list<char> charlist;
    for(int i=0;i<5;i+=3)
    {
        cTemp='a'+i;
        charlist.push_front(cTemp);
    }
    cout << "list old:" << endl;
    list<char>::iterator it;
    for(it=charlist.begin();it!=charlist.end();it++)
    {
        cout << *it << endl;
    }
    list<char>::iterator itstart=charlist.begin();
    charlist.insert(++itstart,2,'A');
    cout << "list old" << endl;
}

```

```

for(it=charlist.begin();it!=charlist.end();it++)
{
    cout << *it << endl;
}
return 0;
}

```

程序运行结果如图 13.3 所示。

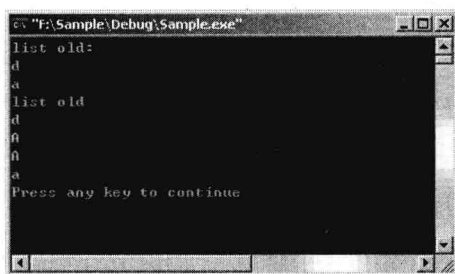


图 13.3 链表类模板应用

程序实现向链表的任意位置插入元素，`itstart` 迭代器就是负责插入的迭代器，`insert` 实现在第 2 个位置连续插入两个字符 ‘A’。

## 13.2 结 合 容 器

结合容器（associative 容器）是 STL 提供的容器的一种，其中的元素都是经过排序的，它主要通过关键字的方式来提高查询的效率。结合容器包括 `set`、`multiset`、`map`、`multimap` 和 `hash table`，本章主要介绍 `set`、`multiset`、`map` 和 `multimap`。

### 13.2.1 set 类模板

 视频讲解：光盘\TM\lx\13\set 类模板.exe

`set` 类模板又称为集合类模板，一个集合对象像链表一样顺序地存储一组值。在一个集合中，集合元素既充当存储的数据，又充当数据的关键字。

可以使用下面的几种方法来创建 `set` 对象：

☒ `std::set<type,predicate>name;`

这种方法创建了一个名为 `name`，并且包含 `type` 类型数据的 `set` 空对象。该对象使用谓词所指定的函数来对集合中的元素进行排序。例如，要给整数创建一个空 `set` 对象，可以这样写：

```
std::set<int,std::less<int>> intset;
```

☒ `std::set<type,predicate> name(myset)`



这种方法使用了复制构造函数，从一个已存在的集合 `myset` 中生成一个 `set` 对象。

☑ `std::set<type,predicate> name(first,last)`

这种方法从一定范围的元素中根据多重指示器所指示的起始与终止位置创建一个集合。

`set` 类中的方法说明如表 13.4 所示。

表 13.4 `set` 类中的方法说明

函 数	说 明
<code>begin</code>	返回指向集合中第一个元素的迭代器
<code>clear</code>	删除集合中所有元素
<code>count(x)</code>	返回集合中值为 <code>x</code> (0 或 1) 的元素个数
<code>empty</code>	如果集合为空，则返回 <code>true</code> 值
<code>end</code>	返回指向集合中最后一个元素的迭代器
<code>equal_range(x)</code>	返回表示 <code>x</code> 下界和上界的两个迭代器，下界表示集合中第一个值等于 <code>x</code> 的元素，上界表示第一个值大于 <code>x</code> 的元素
<code>erase(i)</code>	删除由迭代器 <code>i</code> 所指向的集合元素
<code>erase(start,end)</code>	删除由迭代器 <code>start</code> 和 <code>end</code> 所指范围内的集合元素
<code>erase(x)</code>	删除集合中值为 <code>x</code> 的元素
<code>find(x)</code>	返回一个指向 <code>x</code> 的迭代器。如果 <code>x</code> 不存在，返回的迭代器等于 <code>end</code>
<code>insert(i,x)</code>	把值 <code>x</code> 插入集合。 <code>x</code> 的插入位置从迭代器 <code>i</code> 所指明的元素处开始查找
<code>insert(start,end)</code>	把迭代器 <code>start</code> 和 <code>end</code> 所指范围内的值插入集合中
<code>insert(x)</code>	把 <code>x</code> 插入集合
<code>lower_bound(x)</code>	返回一个迭代器，指向位于 <code>x</code> 之前且紧邻 <code>x</code> 的元素
<code>max_size</code>	返回集合的最大容量
<code>rbegin</code>	返回一个反向迭代器，指向集合的最后一个元素
<code>rend</code>	返回一个反向迭代器，指向集合的第一个元素
<code>size</code>	返回集合的大小
<code>swap(set)</code>	交换两个集合的内容
<code>upper_bound(x)</code>	返回一个指向 <code>x</code> 的迭代器
<code>value_comp</code>	返回 <code>value_compare</code> 类型的对象，该对象用于判断集合中元素的先后次序

下面通过一些操作来实现对 `set` 对象的应用。

**【例 13.4】** 创建整型类型的集合，并在该集合中实现数据的插入。(实例位置：光盘\TM\sl\13\4)

```
#include<iostream>
#include<set>
using namespace std;
void main()
{
    set<int> iSet;           //创建整型集合
    iSet.insert(1);          //插入数据
    iSet.insert(3);
    iSet.insert(5);
    iSet.insert(7);
    iSet.insert(9);
}
```

```

cout << "set:" << endl;
set<int>::iterator it;           //循环并输出集合中的数据
for(it=iSet.begin();it!=iSet.end();it++)
    cout << *it << endl;
}

```

程序运行结果如图 13.4 所示。

**【例 13.5】** 利用 set 对象创建一个整型类型的集合，并删除集合中的元素。（实例位置：光盘\TM\sl\13\5）

```

#include<iostream>
#include<set>
using namespace std;
void main()
{
    set<int> iSet;                //创建一个整型集合
    iSet.insert(1);               //向集合中插入元素
    iSet.insert(3);
    iSet.insert(5);
    iSet.insert(7);
    iSet.insert(9);
    cout << "old set:" << endl;
    set<int>::iterator it;        //循环集合显示元素值
    for(it=iSet.begin();it!=iSet.end();it++)
        cout << *it << endl;
    it=iSet.begin();
    iSet.erase(++it);             //删除集合中的元素
    cout << "new set:" << endl;
    for(it=iSet.begin();it!=iSet.end();it++) //循环集合，显示元素删除后的集合
        cout << *it << endl;
}

```

程序运行结果如图 13.5 所示。

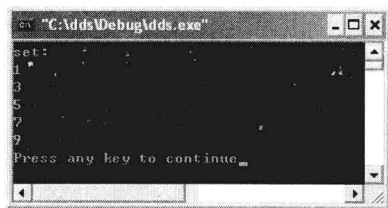


图 13.4 插入数据

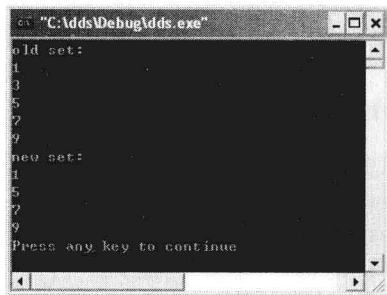


图 13.5 删除元素

**【例 13.6】** 创建一个字符型的 set 对象，插入元素值，并通过指定的字符在集合中查找元素。（实例位置：光盘\TM\sl\13\6）

```

#include<iostream>

```

```

#include<set>
using namespace std;
void main()
{
    set<char> cSet;                //利用 set 对象创建字符类型的集合
    cSet.insert('B');              //插入元素
    cSet.insert('C');
    cSet.insert('D');
    cSet.insert('A');
    cSet.insert('F');
    cout << "old set:" << endl;
    set<char>::iterator it;        //循环显示集合中的元素
    for(it=cSet.begin();it!=cSet.end();it++)
        cout << *it << endl;
    char cTmp;
    cTmp='D';
    it=cSet.find(cTmp);            //在集合中查找指定的元素
    cout << "start find:" << cTmp << endl;
    if(it==cSet.end())            //未找到元素
        cout << "not found" << endl;
    else                          //找到元素
        cout << "found" << endl;
    cTmp='G';
    it=cSet.find(cTmp);            //查找指定的元素
    cout << "start find:" << cTmp << endl;
    if(it==cSet.end())            //未找到元素
        cout << "not found" << endl;
    else                          //找到元素
        cout << "found" << endl;
}

```

程序运行结果如图 13.6 所示。

**【例 13.7】** 创建两个集合，分别向集合中插入数据，并对集合进行比较。（实例位置：光盘\TM\sl\13\7）

```

#include<iostream>
#include<set>
using namespace std;
void main()
{
    set<char> cSet1;              //建立集合 1
    cSet1.insert('C');            //向集合 1 插入元素
    cSet1.insert('D');
    cSet1.insert('A');
    cSet1.insert('F');
    cout << "set1:" << endl;
    set<char>::iterator it;
    for(it=cSet1.begin();it!=cSet1.end();it++) //显示集合 1 中的元素
        cout << *it << endl;
}

```

```

set<char> cSet2;           //建立集合 2
cSet2.insert('B');        //向集合 2 插入元素
cSet2.insert('C');
cSet2.insert('D');
cSet2.insert('A');
cSet2.insert('F');
cout << "set2:" << endl;
for(it=cSet2.begin();it!=cSet2.end();it++) //显示集合 2 中的元素
    cout << *it << endl;
if(cSet1==cSet2)
    cout << "set1= set2";
else if(cSet1 < cSet2)
    cout << "set1< set2";
else if(cSet1 > cSet2)
    cout << "set1> set2";
cout << endl;
}

```

程序运行结果如图 13.7 所示。

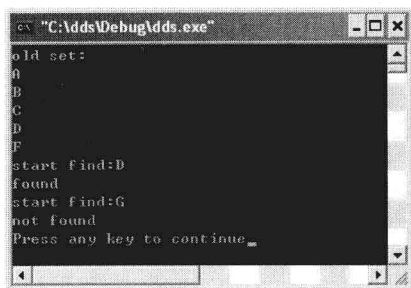


图 13.6 查找元素



图 13.7 比较集合

## 13.2.2 multiset 类模板

multiset 使程序能顺序存储一组数据。与集合类类似，多重集合的元素既可以作为存储的数据又可以作为数据的关键字。然而，与集合类不同的是多重集合类可以包含重复的数据。下面列出了几种创建多重集合的方法：

☑ `std::multiset<type,predicate> name;`

这种方法创建了一个名为 name，并且包含 type 类型数据的 multiset 空对象。该对象使用谓词所指定的函数来对集合中的元素进行排序。例如，要给整数创建一个空 multiset 对象，可以这样写：

```
std::multiset<int, std::less<int> > intset;
```



**注意**

`less<int>`表达式后要有空格。



☑ `std::multiset <type,predicate> name(mymultiset)`

这种方法使用了复制构造函数，从一个已经存在的集合 `mymultiset` 中生成一个 `multiset` 对象。

☑ `std::multiset <type,predicate> name(first,last)`

这种方法从一定范围的元素中根据指示器所指示的起始与终止位置创建一个集合。

`multiset` 类中的方法说明如表 13.5 所示。

表 13.5 `multiset` 类中的方法说明

函 数	说 明
<code>begin</code>	返回指向集合中第一个元素的迭代器
<code>clear</code>	删除集合中所有元素
<code>count(x)</code>	返回集合中值为 <code>x</code> (0 或 1) 的元素个数
<code>empty</code>	如果集合为空，则返回 <code>true</code> 值
<code>end</code>	返回指向集合中最后一个元素的迭代器
<code>equal_range(x)</code>	返回表示 <code>x</code> 下界和上界的两个迭代器，下界表示集合中第一个值等于 <code>x</code> 的元素，上界表示第一个值大于 <code>x</code> 的元素
<code>erase(i)</code>	删除由迭代器 <code>i</code> 所指向的集合元素
<code>erase(start,end)</code>	删除由迭代器 <code>start</code> 和 <code>end</code> 所指范围内的集合元素
<code>erase(x)</code>	删除集合中值为 <code>x</code> 的元素
<code>find(x)</code>	返回一个指向 <code>x</code> 的迭代器。如果 <code>x</code> 不存在，返回的迭代器等于 <code>end</code>
<code>insert(i,x)</code>	把值 <code>x</code> 插入集合。 <code>x</code> 的插入位置从迭代器 <code>i</code> 所指明的元素处开始查找
<code>insert(start,end)</code>	把迭代器 <code>start</code> 和 <code>end</code> 所指范围内的值插入集合中
<code>insert(x)</code>	把 <code>x</code> 插入集合
<code>lower_bound(x)</code>	返回一个迭代器，指向位于 <code>x</code> 之前且紧邻 <code>x</code> 的元素
<code>max_size</code>	返回集合的最大容量
<code>rbegin</code>	返回一个反向迭代器，指向集合最后一个元素
<code>rend</code>	返回一个反向迭代器，指向集合的第一个元素
<code>size</code>	返回集合的大小
<code>swap(set)</code>	交换两个集合的内容
<code>upper_bound(x)</code>	返回一个指向 <code>x</code> 的迭代器
<code>value_comp</code>	返回 <code>value_compare</code> 类型的对象，该对象用于判断集合中元素的先后次序

下面通过一些操作来实现对 `multiset` 对象的应用。

**【例 13.8】** 创建整型类型的集合，并在该集合中实现数据的插入。(实例位置：光盘\TM\sl\13\8)

```
#include "stdafx.h"
#include<iostream>
#include<set>
using namespace std;
void main()
{
    multiset<int> imultiset;           //创建整型多重集合
    imultiset.insert(1);              //插入数据
    imultiset.insert(3);
```



```

    multiset.insert(5);
    multiset.insert(5);
    multiset.insert(9);
    cout << "multiset:" << endl;
    multiset<int>::iterator it;           //循环并输出集合中的数据
    for(it=multiset.begin();it!=multiset.end();it++)
        cout << *it << endl;
}

```

程序运行结果如图 13.8 所示。

**【例 13.9】** 利用 multiset 对象创建一个整型类型的集合，并删除集合中的元素。（实例位置：光盘\TM\sl\13\9）

```

#include "stdafx.h"
#include<iostream>
#include<set>
using namespace std;
void main()
{
    multiset<int> imultiset;           //创建一个整型多重集合
    imultiset.insert(1);               //向集合中插入元素
    imultiset.insert(3);
    imultiset.insert(5);
    imultiset.insert(7);
    imultiset.insert(9);
    cout << "old multiset:" << endl;
    multiset<int>::iterator it;
    for(it=imultiset.begin();it!=imultiset.end();it++) //循环集合显示元素值
        cout << *it << endl;
    it=imultiset.begin();
    imultiset.erase(++it);            //删除集合中的元素
    cout << "new multiset:" << endl;
    for(it=imultiset.begin();it!=imultiset.end();it++) //循环集合，显示元素删除后的集合
        cout << *it << endl;
}

```

程序运行结果如图 13.9 所示。

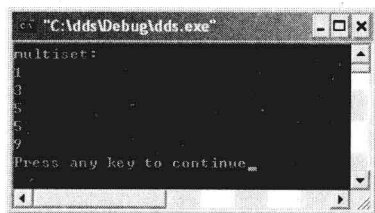


图 13.8 数据插入

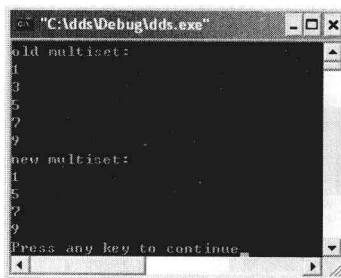


图 13.9 元素删除

**【例 13.10】** 创建一个字符型的 multiset 对象，插入元素值，并通过指定的字符在集合中查找元

素。(实例位置: 光盘\TM\sl\13\10)

```
#include "stdafx.h"
#include<iostream>
#include<set>
using namespace std;
void main()
{
    multiset<char> cmultiset;           //利用 multiset 对象创建字符类型的集合
    cmultiset.insert('B');              //插入元素
    cmultiset.insert('C');
    cmultiset.insert('D');
    cmultiset.insert('A');
    cmultiset.insert('F');
    cout << "old multiset:" << endl;
    multiset<char>::iterator it;        //循环显示集合中的元素
    for(it=cmultiset.begin();it!=cmultiset.end();it++)
        cout << *it << endl;
    char cTmp;
    cTmp='D';
    it=cmultiset.find(cTmp);            //在集合中查找指定的元素
    cout << "start find:" << cTmp << endl;
    if(it==cmultiset.end())             //未找到元素
        cout << "not found" << endl;
    else                                //找到元素
        cout << "found" << endl;
    cTmp='G';
    it=cmultiset.find(cTmp);            //查找指定的元素
    cout << "start find:" << cTmp << endl;
    if(it==cmultiset.end())             //未找到元素
        cout << "not found" << endl;
    else                                //找到元素
        cout << "found" << endl;
}
```

程序运行结果如图 13.10 所示。

**【例 13.11】** 创建两个多重集合, 分别向集合中插入数据, 并对集合进行比较。(实例位置: 光盘\TM\sl\13\11)

```
#include "stdafx.h"
#include<iostream>
#include<set>
using namespace std;
void main()
{
    multiset<char> cmultiset1;          //建立集合 1
    cmultiset1.insert('C');              //向集合 1 插入元素
    cmultiset1.insert('D');
    cmultiset1.insert('A');
```

```

cmultiset1.insert('F');
cout << "multiset1:" << endl;
multiset<char>::iterator it;
for(it=cmultiset1.begin();it!=cmultiset1.end();it++) //显示集合 1 中的元素
    cout << *it << endl;
multiset<char> cmultiset2; //建立集合 2
cmultiset2.insert('B'); //向集合 2 插入元素
cmultiset2.insert('C');
cmultiset2.insert('D');
cmultiset2.insert('A');
cmultiset2.insert('F');
cout << "multiset2:" << endl;
for(it=cmultiset2.begin();it!=cmultiset2.end();it++) //显示集合 2 中的元素
    cout << *it << endl;
if(cmultiset1==cmultiset2)
    cout << "multiset1= multiset2";
else if(cmultiset1 < cmultiset2)
    cout << "multiset1< multiset2";
else if(cmultiset1 > cmultiset2)
    cout << "multiset1> multiset2";
cout << endl;
}

```

程序运行结果如图 13.11 所示。

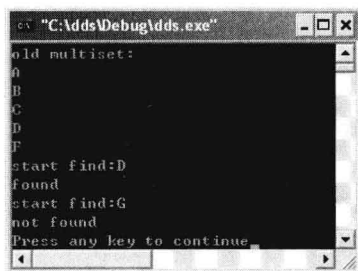


图 13.10 元素查找



图 13.11 比较集合

### 13.2.3 map 类模板

map 对象按顺序存储一组值，其中每个元素与一个检索关键字关联。map 与 set 和 multiset 不同，set 和 multiset 中元素既被作为存储的数据又被作为数据的关键字，而 map 类型中元素的数据和关键字是分开的。创建 map 类模板的方法如下：

☑ map<key,type,predicate> name;

这种方法创建了一个名为 name，并且包含 type 类型数据的 map 空对象。该对象使用谓词所指定的函数来对集合中的元素进行排序。例如，要给整数创建一个空 map 对象，可以这样写：

```
std::map<int,int,std::less<int>> intmap;
```



☑ `map<key,type,predicate> name(mymap);`

这种方法使用了复制构造函数，从一个已存在的映射 `mymap` 中生成一个 `map` 对象。

☑ `map<key,type,predicate> name(first,last);`

这种方法从一定范围的元素中根据多重指示器所指示的起始与终止位置创建一个映射。

下面通过一些操作来实现对 `map` 对象的应用。

**【例 13.12】** 创建一个 `map` 映射对象，并向该映射中插入新的元素。(实例位置：光盘\TM\sl\13\12)

```
#include<iostream>
#include<map>
using namespace std;
void main()
{
    map<int ,char> cMap;                                //创建 map 映射对象
    cMap.insert(map<int,char>::value_type(1,'B'));        //插入新元素
    cMap.insert(map<int,char>::value_type(2,'C'));
    cMap.insert(map<int,char>::value_type(4,'D'));
    cMap.insert(map<int,char>::value_type(5,'G'));
    cMap.insert(map<int,char>::value_type(3,'F'));
    cout << "map" << endl;
    map<int ,char>::iterator it;                          //循环 map 映射并显示元素值
    for(it=cMap.begin();it!=cMap.end();it++)
    {
        cout << (*it).first << "->";
        cout << (*it).second << endl;
    }
}
```

程序运行结果如图 13.12 所示。

**【例 13.13】** 创建一个 `map` 映射对象，并使用下标插入新的元素。(实例位置：光盘\TM\sl\13\13)

```
#include<iostream>
#include<map>
using namespace std;
void main()
{
    map<int,char> cMap;                                //创建 map 映射对象
    cMap[1]='B';                                         //插入新元素
    cMap[2]='C';
    cMap[3]='D';
    cMap[4]='G';
    cMap[5]='F';
    cout << "map" << endl;
    map<int ,char>::iterator it;                          //循环映射对象中的元素，并显示值
    for(it=cMap.begin();it!=cMap.end();it++)
    {
        cout << (*it).first << "->";
        cout << (*it).second << endl;
    }
}
```

程序运行结果如图 13.13 所示。

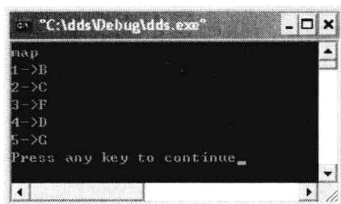


图 13.12 插入元素

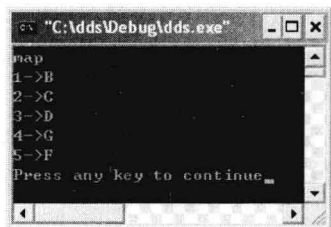


图 13.13 使用下标插入元素

### 13.2.4 multimap 类模板

multimap 能够顺序存储一组值，它与 map 相同的是每一个元素都包含一个关键字以及与之联系的数据项，与 map 不同的是多重映射可以包含重复的数据值，并且不能使用[]操作符向多重映射中插入元素。

构造 multimap 类模板的方法如下：

❑ multimap<key,type,predicate> name;

这种方法创建了一个名为 name，并且包含 type 类型数据的 multimap 空对象。该对象使用谓词所指定的函数来对集合中的元素进行排序。例如，要给整数创建一个空 multimap 对象，可以这样写：

```
std::multimap<int,int, std::less<int> > intmap;
```

❑ multimap<key,type,predicate> name(mymap);

这种方法使用了复制构造函数，从一个已存在的映射 mymap 中生成一个 multimap 对象。

❑ multimap<key,type,predicate> name(first,last);

这种方法从一定范围的元素中根据多重指示器所指示的起始与终止位置创建一个多重映射。

下面通过一些操作来实现对 multimap 对象的应用。

【例 13.14】 创建一个 multimap 映射对象，并向该映射中插入新的元素。（实例位置：光盘\TM\sl\13\14）

```
#include<iostream>
#include<map>
using namespace std;
void main()
{
    map<int ,char> cMap;                //创建 map 映射对象
    cMap.insert(map<int,char>::value_type(1,'B')); //插入新元素
    cMap.insert(map<int,char>::value_type(2,'C'));
    cMap.insert(map<int,char>::value_type(4,'C'));
    cMap.insert(map<int,char>::value_type(5,'G'));
    cMap.insert(map<int,char>::value_type(3,'F'));
    cout << "map" << endl;
    map<int ,char>::iterator it;        //循环 map 映射并显示元素值
```



```
for(it=cMap.begin();it!=cMap.end();it++)
{
    cout << (*it).first << "->";
    cout << (*it).second << endl;
}
}
```

程序运行结果如图 13.14 所示。

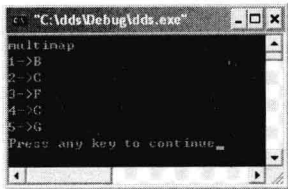


图 13.14 插入新元素

## 13.3 算 法

 视频讲解：光盘\TM\lx\13\算法.exe

算法（algorithm）是 STL 的中枢，STL 提供了算法库，算法库中都是模板函数。迭代器主要负责从容器中获取一个对象，算法与具体对象在容器中的什么位置等细节无关。每个算法都是参数化一个或多个迭代器类型的函数模板。

标准算法分 4 个类别：非修正序列算法、修正序列算法、排序算法和数值算法。

### 13.3.1 非修正序列算法

非修正序列算法不修改它们所作用的容器，例如计算元素个数或查找元素的函数。STL 中提供的非修正序列算法如表 13.6 所示。

表 13.6 STL 中提供的非修正序列算法

函 数	说 明
adjacent_find(first,last)	搜索相邻的重复元素
count(first,last,val)	计数
equal(first,last,first2)	判断是否相等
find(first,last,val)	搜索
find_end(first,last,first2,last2)	搜索某个子序列的最后一次出现地点
find_first(first,last,first2,last2)	搜索某些元素的首次出现地点
for_each(first,last,func)	对 first 到 last 范围内的各个元素执行函数 func 定义的操作
mismatch(first,last,first2)	找出不吻合点
search(first,last,first2,last2)	搜索某个子序列

**☑ adjacent\_find(first,last)**

返回一个迭代器，指向第一对同值元素对的第一个元素。函数在迭代器 `first` 和 `last` 指明的范围内查找。此函数还有一个谓词版本，其第 3 个实参是一个比较函数。

**【例 13.15】** 应用 `adjacent_find` 算法搜索相邻的重复元素。（实例位置：光盘\TM\sl\13\15）

```
//adjacent_find
#include<iostream>
#include<set>
#include<algorithm>
using namespace std;
void main()
{
    multiset<int, less<int> > intSet;
    intSet.insert(7);
    intSet.insert(5);
    intSet.insert(1);
    intSet.insert(5);
    intSet.insert(7);
    cout << "Set:" << " ";
    multiset<int, less<int> >::iterator it = intSet.begin();
    for(int i=0; i<intSet.size(); ++i)
        cout << *it++ << ' ';
    cout << endl;
    cout << "第一次匹配: ";
    it = adjacent_find(intSet.begin(), intSet.end());
    cout << *it++ << ' ';
    cout << *it << endl;
    cout << "第二次匹配: ";
    it = adjacent_find(it, intSet.end());
    cout << *it++ << ' ';
    cout << *it << endl;
}
```

程序运行结果如图 13.15 所示。

程序中定义了整型的 `multiset` 容器，以及该容器的迭代器 `it`，在 `multiset` 容器中有两个重复的值，使用 `adjacent_find` 算法将这两个元素输出。

**☑ count(first,last,val)**

返回容器中值为 `val` 的元素个数。函数在迭代器 `first` 和 `last` 指明的范围内查找。

**【例 13.16】** 应用 `count` 算法计算相同元素的个数。（实例位置：光盘\TM\sl\13\16）

```
#include<iostream>
#include<set>
#include<algorithm>
using namespace std;
void main()
{
    multiset<int, less<int> > intSet;
```

```

intSet.insert(7);
intSet.insert(5);
intSet.insert(1);
intSet.insert(5);
intSet.insert(7);
cout << "Set:";
    multiset<int,less<int> >::iterator it =intSet.begin();
for(int i=0;i<intSet.size();++i)
    cout << *it++ << ' ';
cout << endl;

int cnt =count(intSet.begin(),intSet.end(),5);
cout << "相同元素数量:" << cnt <<endl;
}

```

程序运行结果如图 13.16 所示。

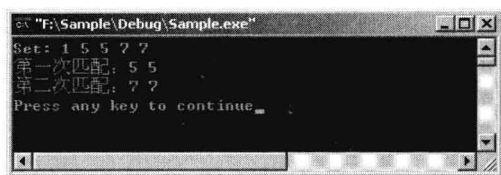


图 13.15 应用 adjacent\_find 算法搜索相邻的重复元素

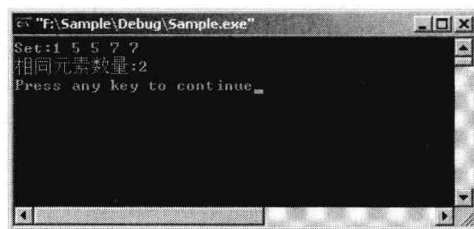


图 13.16 应用 count 算法计算相同元素的个数

程序中 multiset 容器有两个相同元素，使用 count 算法获取容器中重复的元素。

☒ for\_each(first,last,func)

对 first 到 last 范围内的各个元素执行函数 func 定义的操作。

【例 13.17】 使用 for\_each 算法输出容器内的元素。(实例位置：光盘\TM\sl\13\17)

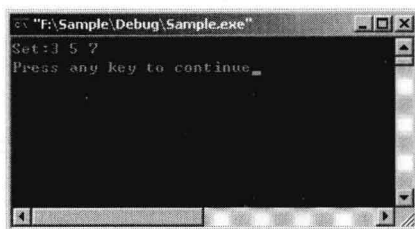
```

#include<iostream>
#include<set>
#include<algorithm>
using namespace std;
void Output(int val)
{
    cout << val << ' ';
}
void main()
{
    multiset<int,less<int> > intSet;
    intSet.insert(7);
    intSet.insert(5);
    intSet.insert(3);
    cout << "Set:";
    for_each(intSet.begin(),intSet.end(),Output);
    cout << endl;
}

```

程序运行结果如图 13.17 所示。

程序中定义 `Output` 函数用来输出变量值，调用 `for_each` 算法将 `multiset` 容器中的值不断地传输给 `Output` 函数，执行 `for_each` 算法相当于执行了一个循环语句。



### 13.3.2 修正序列算法

修正序列算法的有些操作会改变容器的内容。例如，图 13.17 使用 `for_each` 算法输出容器内的元素。把一个容器的部分内容复制到同一个容器的另一部分，或者用指定值填充容器。STL 的修正序列算法提供了这类操作，如表 13.7 所示。

表 13.7 STL 的修正序列算法

函 数	算 法
<code>copy(first,last,first2)</code>	复制
<code>copy_backward(first,last,first2)</code>	逆向复制
<code>fill(first,last,val)</code>	改填元素值
<code>generate(first,last,func)</code>	以指定动作的运算结果填充特定范围内的元素
<code>partition(first,last,pred)</code>	切割
<code>random_shuffle(first,last)</code>	随机重排
<code>remove(first,last,val)</code>	移除某种元素，但不删除
<code>replace(first,last,val1,val2)</code>	取代某种元素
<code>rotate(first,middle,last)</code>	旋转
<code>reverse(first,last)</code>	颠倒元素次序
<code>swap(it1,it2)</code>	置换
<code>swap_ranges(first,last,first2)</code>	置换指定范围
<code>transform(first,last,first2,func)</code>	以两个序列为基础，交互作用产生第 3 个序列
<code>unique(first,last)</code>	将重复的元素折叠缩编，变成唯一的

#### ☑ `fill(first,last,val)`

把值 `val` 复制到迭代器 `first` 和 `last` 指明范围内的各个元素中。

**【例 13.18】** 应用 `fill` 算法对容器元素赋值。（实例位置：光盘\TM\sl\13\18）

```
#include<iostream>
#include<vector>
#include<algorithm>
using namespace std;
void Output(int val)
{
    cout << val << ' ';
}
void main()
{
    vector<int > intVect;
```



```

for(int i=0;i<10;++i)
    intVect.push_back(i);
cout << "Vect :";
for_each(intVect.begin(),intVect.end(),Output);
fill(intVect.begin(),intVect.begin()+5,0);
cout << endl;
cout << "Vect :";
for_each(intVect.begin(),intVect.end(),Output);
cout << endl;
}

```

程序运行结果如图 13.18 所示。

程序中向 vector 容器中添加 0~9 共 10 个元素，然后使用 fill 算法将前 5 个元素的值全改为 0，通过在修改前输出全部元素和在修改后输出全部元素，可以观察到 fill 算法的执行效果。

#### ☑ random\_shuffle(first,last)

把迭代器 first 和 last 指明范围内的元素顺序随机打乱。

【例 13.19】 应用 random\_shuffle 算法将元素顺序随机打乱。（实例位置：光盘\TM\sl\13\19）

```

#include<iostream>
#include<vector>
#include<algorithm>
using namespace std;
void Output(int val)
{
    cout << val << ' ';
}
void main()
{
    vector<int> intVect;
    for(int i=0;i<10;++i)
        intVect.push_back(i);
    cout << "Vect :";
    for_each(intVect.begin(),intVect.end(),Output);
    random_shuffle(intVect.begin(),intVect.end());
    cout << endl;
    cout << "Vect :";
    for_each(intVect.begin(),intVect.end(),Output);
}

```

程序运行结果如图 13.19 所示。

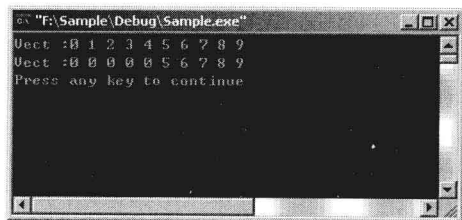


图 13.18 应用 fill 算法对容器元素赋值

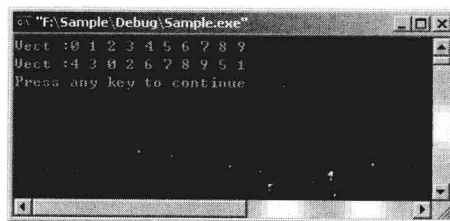


图 13.19 应用 random\_shuffle 算法将元素顺序随机打乱



程序中应用 `random_shuffle` 算法将 `vector` 容器内元素的排列顺序打乱，原来 `vector` 容器内的元素是从 0 到 9 顺序排列的，打乱后元素顺序没有任何规律。

#### ☑ `partition(first,last,pred)`

把一个容器划分成两部分，第 1 部分包含令谓词 `pred` 返回 `true` 值的元素，第 2 部分包含令谓词 `pred` 返回 `false` 值的元素。函数返回的迭代器指向两部分的分界点元素。

**【例 13.20】** 应用 `partition` 算法将容器分组。（实例位置：光盘\TM\sl\13\20）

```
#include<iostream>
#include<vector>
#include<algorithm>
using namespace std;
void Output(int val)
{
    cout << val << ' ';
}
bool equals5(int val)
{
    return val==5;
}
void main()
{
    vector<int > intVect;
    intVect.push_back(7);
    intVect.push_back(3);
    intVect.push_back(5);
    cout << "Vect .:";
    for_each(intVect.begin(),intVect.end(),Output);
    partition(intVect.begin(),intVect.end(),equals5);
    cout << endl;
    cout << "Vect .:";
    for_each(intVect.begin(),intVect.end(),Output);
    cout << endl;
}
```

程序运行结果如图 13.20 所示。

#### ☑ `rotate(first,middle,last)`

把从 `middle` 到 `last` 范围内的元素作旋转运算，并放置到从 `first` 开始的子序列中。

**【例 13.21】** 应用 `rotate` 算法。（实例位置：光盘\TM\sl\13\21）

```
//rotate
#include<iostream>
#include<vector>
#include<algorithm>
using namespace std;
void Output(int val)
{
    cout << val << ' ';
```

```
}
void main()
{
    vector<char> charVect;
    charVect.push_back('B');
    charVect.push_back('A');
    charVect.push_back(' ');
    charVect.push_back('M');
    charVect.push_back('R');
    charVect.push_back(' ');
    charVect.push_back('K');

    cout << "Vect :";
    for_each(charVect.begin(),charVect.end(),Output);
    rotate(charVect.begin(),charVect.begin()+6,charVect.end());
    cout << endl;
    cout << "Vect :";
    for_each(charVect.begin(),charVect.end(),Output);
    cout << endl;
}
```

程序运行结果如图 13.21 所示。

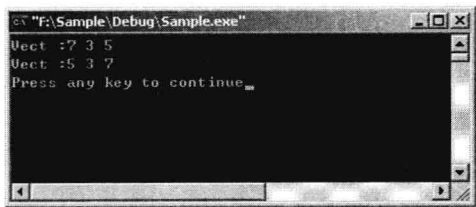


图 13.20 应用 partition 算法将容器分组

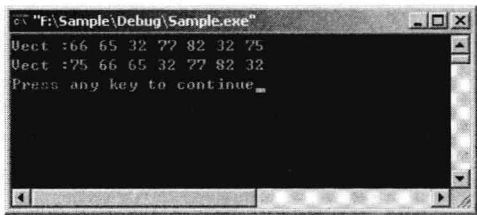


图 13.21 应用 rotate 算法

程序使用旋转运算对容器内的元素重新排序。旋转运算就是以元素为中心，将前后对应的两个元素对调位置。

13.3.3 排序算法

排序算法的特点是对容器的内容进行不同方式的排序，例如 sort 算法。排序算法如表 13.8 所示。

表 13.8 排序算法

函 数	说 明
binary_search(first,last,val)	二元搜索
equal_range(first,last,val)	判断是否相等，并返回一个区间
includes(first,last,first2,last2)	包含于
lexicographical_compare(first,last,first2,last2)	以字典排列方式作比较
lower_bound(first,last,val)	下限
make_heap(first,last)	制造一个 heap

续表

函 数	说 明
max(val1,val2)	最大值
max_element(first,last)	最大值所在位置
merge(first,last,first2,last2,result)	合并两个序列
min(val1,val2)	最小值
min_element(first,last)	最小值所在位置
next_permutation(first,last)	获得下一个排列组合
nth_element(first,nth,last)	重新排序列中第 <i>n</i> 个元素的左右两端
partial_sort_copy(first,last,first2,last2,result)	局部排序并复制到其他位置
partial_sort(first,middle,last)	局部排序
pop_heap(first,last)	从 heap 内取出一个元素
prev_permutation(first,last)	改变迭代器 first 和 last 指明范围内的元素排列，使新排列是下一个比原排列小的排列。此函数的另一个版本以谓词作为第 3 个实参
push_heap(first,last)	将一个元素压入 heap
set_difference(first,last,first2,last2,result)	获得前一个排列组合
set_intersection(first,last,first2,last2,result)	交集
set_symmetric_difference(first,last,first2,last2,result)	差集
set_union(first,last,first2,last2,result)	联集
sort(first,last)	排序
sort_heap(first,last)	对 heap 排序
stable_sort(first,last)	排序并保持等值元素的相对次序
upper_bound(first,last,val)	上限

☒ sort(first,last)

对迭代器 first 和 last 指明范围内的元素排序。此函数的另一个版本以谓词作为第 3 个实参。

**【例 13.22】** 应用 sort 算法。（实例位置：光盘\TM\sl\13\22）

```
#include<iostream>
#include<vector>
#include<algorithm>
using namespace std;
void Output(int val)
{
    cout << val << ' ';
}
void main()
{
    vector<char> charVect;
    charVect.push_back('M');
    charVect.push_back('R');
    charVect.push_back('K');
    charVect.push_back('J');
```

```

charVect.push_back('H');
charVect.push_back('I');
cout << "Vect :";
for_each(charVect.begin(),charVect.end(),Output);
sort(charVect.begin(),charVect.end());
cout << endl;
cout << "Vect :";
for_each(charVect.begin(),charVect.end(),Output);
cout << endl;
}

```

程序运行结果如图 13.22 所示。

程序中应用 sort 算法对 vector 容器内的元素进行递增排序。

#### ☒ partial\_sort(first,middle,last)

对迭代器 first 和 last 指明范围内的元素进行排序，把排序后的前一部分元素放至序列的前一部分（由 first 和 middle 指明的范围），其余元素放至后一部分（由 middle 和 last 指明的范围）。此函数的另一个版本以谓词作为第 4 个实参。

**【例 13.23】** 应用 partial\_sort 算法。（实例位置：光盘\TM\sl\13\23）

```

#include<iostream>
#include<vector>
#include<algorithm>
#include<string>
using namespace std;
void Output(const string& val)
{
    cout << val << endl;
}
void main()
{
    vector<string> strVect;
    strVect.push_back("Sunday");
    strVect.push_back("Monday");
    strVect.push_back("Tuesday");
    strVect.push_back("Wednesday");
    strVect.push_back("Thursday");
    strVect.push_back("Friday");
    strVect.push_back("Saturday");
    cout << "Vect :";
    for_each(strVect.begin(),strVect.end(),Output);
    partial_sort(strVect.begin(),strVect.begin()+3,strVect.end());
    cout << endl;
    cout << "Vect :";
    for_each(strVect.begin(),strVect.end(),Output);
}

```

程序运行结果如图 13.23 所示。



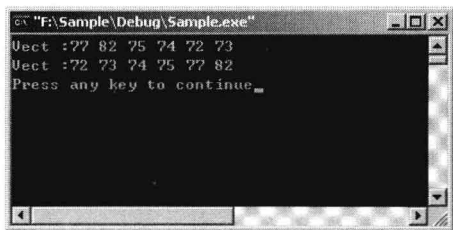


图 13.22 应用 sort 算法

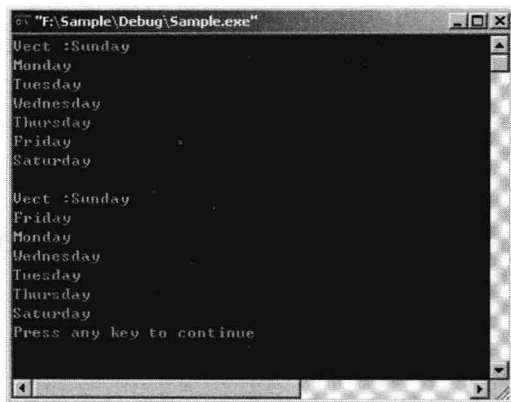


图 13.23 应用 partial\_sort 算法

程序中应用 `partial_sort` 算法对 `vector` 容器内的元素进行排序，将排序后的第 2~第 4 个元素放到容器的前一部分。

☒ `nth_element(first,nth,last)`

在迭代器 `first` 和 `last` 指明范围的已排序元素中查找 `val`。如果找到值为 `val` 的元素，返回 `true`，否则返回 `false` 值。此函数的另一个版本以谓词作为第 4 个实参。

**【例 13.24】** 应用 `nth_element` 算法。（实例位置：光盘\TM\sl\13\24）

```
#include<iostream>
#include<vector>
#include<algorithm>
#include<string>
using namespace std;
void Output(const string& val)
{
    cout << val << endl;
}
void main()
{
    vector<string> strVect;
    strVect.push_back("Sunday");
    strVect.push_back("Monday");
    strVect.push_back("Tuesday");
    strVect.push_back("Wednesday");
    strVect.push_back("Thursday");
    strVect.push_back("Friday");
    strVect.push_back("Saturday");
    cout << "Vect : " << endl;;
    for_each(strVect.begin(),strVect.end(),Output);
    cout << endl;
    nth_element(strVect.begin(),strVect.begin()+5,strVect.end());
    cout << "Vect : " << endl;
    for_each(strVect.begin(),strVect.end(),Output);
}
```



程序运行结果如图 13.24 所示。

程序中应用 `nth_element` 算法查找容器内的最后一个元素, `nth_element` 算法返回 `true` 值。应用完 `nth_element` 算法后, 容器内的元素被重新排序。

☑ `merge(first,last,first2,last2,result)`

把迭代器 `first` 和 `last` 指明范围内的已排序元素与 `first2` 和 `last2` 指明范围内的已排序元素合并, 并把重新排序后的元素放在从 `result` 开始的序列中。此函数的另一个版本以谓词作为第 6 个实参。

**【例 13.25】** 应用 `merge` 算法。(实例位置: 光盘\TM\sl\13\25)

```
#include<iostream>
#include<vector>
#include<algorithm>
#include<string>
using namespace std;
void Output(const string& val)
{
    cout << val << endl;
}
void main()
{
    vector<string> strVect1;
    vector<string> strVect2;
    strVect1.push_back("Sunday");
    strVect1.push_back("Monday");
    strVect1.push_back("Tuesday");
    strVect1.push_back("Wednesday");
    strVect2.push_back("Thursday");
    strVect2.push_back("Friday");
    strVect2.push_back("Saturday");
    strVect2.push_back("Over");
    cout << "Vect1 :." << endl;
    for_each(strVect1.begin(),strVect1.end(),Output);
    cout << endl;
    cout << "Vect2 :." << endl;
    for_each(strVect2.begin(),strVect2.end(),Output);
    cout << endl;
    sort(strVect1.begin(),strVect1.end());
    sort(strVect2.begin(),strVect2.end());
    int size =strVect1.size()+strVect2.size();
    vector<string> strVect3(size);
    merge(strVect1.begin(),strVect1.end(),
          strVect2.begin(),strVect2.end(),
          strVect3.begin());
    cout << "Vect3 :." << endl;
    for_each(strVect3.begin(),strVect3.end(),Output);
}
```

程序运行结果如图 13.25 所示。

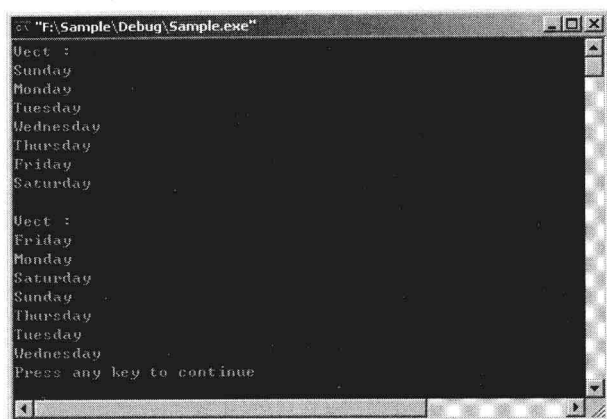


图 13.24 应用 nth\_element 算法

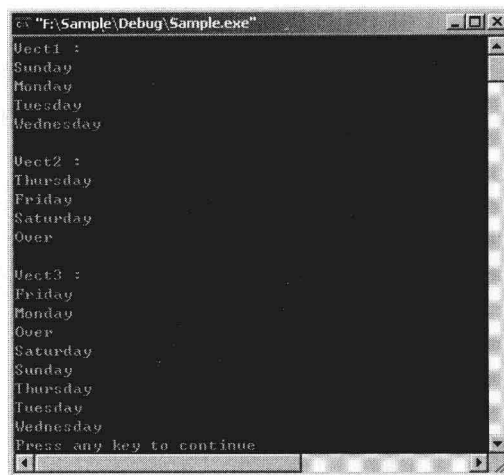


图 13.25 应用 merge 算法

程序中应用 merge 算法将两个容器内的元素重新组合到另一个容器内。

☒ includes(first,last,first2,last2)

在迭代器 first 和 last 指明的范围内，如果含有一个用 first2 和 last2 指明范围的已排序序列，则返回 true 值。此函数的另一个版本以谓词作为第 5 个实参。

【例 13.26】 应用 includes 算法。（实例位置：光盘\TM\sl\13\26）

```
#include<iostream>
#include<vector>
#include<algorithm>
#include<string>
using namespace std;
void Output(const string& val)
{
    cout << val << endl;
}
void main()
{
    vector<string> strVect1;
    vector<string> strVect2;
    strVect1.push_back("Sunday");
    strVect1.push_back("Monday");
    strVect1.push_back("Over");
    strVect1.push_back("Wednesday");
    strVect2.push_back("Monday");
    strVect2.push_back("Sunday");
    strVect2.push_back("Over");
    strVect2.push_back("Saturday");
    sort(strVect1.begin(),strVect1.end());
    sort(strVect2.begin(),strVect2.end());
    cout << "Vect1 : " << endl;
    for_each(strVect1.begin(),strVect1.end(),Output);
```

```

cout << endl;
cout << "Vect2 : " << endl;
for_each(strVect2.begin(),strVect2.end(),Output);
cout << endl;
bool result=includes(strVect1.begin(),strVect1.end(),
    strVect2.begin()+1,strVect2.begin()+2);
if(result)
    cout << "result : OK" << endl;
else
    cout <<"result : ERROR"<<endl;
}

```

程序运行结果如图 13.26 所示。

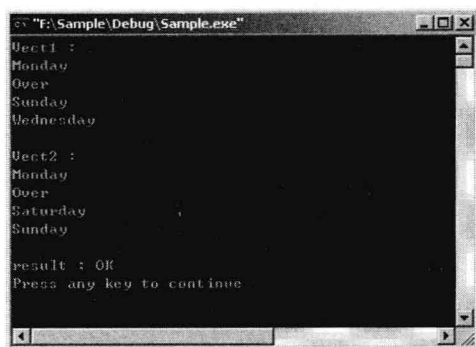


图 13.26 应用 includes 算法

程序判断在 strVect2 内第 2 个元素是否包含于 strVect1 内。

### 13.3.4 数值算法

数值算法是对容器的内容进行数值计算。

STL 的数值算法实现了 4 种类型的计算，可以在一个值序列上进行这些计算。数值算法如表 13.9 所示。

表 13.9 数值算法

函 数	说 明
accumulate(first,last,init)	元素累加
inner_product(first,last,first2,init)	内积
partial_sum(first,last,result)	局部总和
adjacent_difference(first,last,result)	相邻元素的差额

#### ☒ accumulate(first,last,init)

计算 init 与迭代器 first 和 last 指明范围内各元素值的总和，并返回结果。

【例 13.27】 应用 accumulate 算法。(实例位置：光盘\TM\sl\13\27)



```

#include<iostream>
#include<vector>
#include<algorithm>
#include<numeric>
using namespace std;
void Output(int val)
{
    cout << val << ' ';
}
void main()
{
    vector<int> intVect;
    for(int i=0;i<5;i++)
        intVect.push_back(i);
    cout << "Vect";
    std::for_each(intVect.begin(),intVect.end(),Output);
    int result = accumulate(intVect.begin(),intVect.end(),5);
    cout << endl;
    cout << "Result :." << result << endl;
}

```

程序运行结果如图 13.27 所示。

程序对容器内的元素值进行累加，结果是  $(1+2) + (2+3) + (3+4)$ 。

#### ☒ inner\_product(first,last,first2,init)

先计算迭代器 first 和 last 指明的序列与从 first2 开始的相同长度的序列的内积，然后返回内积与 init 的和。计算内积的步骤为：依次把第 1 个序列的各个元素值与第 2 个序列对应位置的元素值相乘，再计算各个乘积的和。

**【例 13.28】** 应用 inner\_product 算法。（实例位置：光盘\TM\sl\13\28）

```

#include<iostream>
#include<vector>
#include<algorithm>
#include<numeric>
using namespace std;
void Output(int val)
{
    cout << val << ' ';
}
void Output2(int val)
{
    cout << val << endl;
}
void main()
{
    vector<int> intVect1;
    vector<int> intVect2;

    for(int i=0;i<5;i++)

```

```

    intVect1.push_back(i);
    for(int j=2;j<8;j++)
        intVect2.push_back(j);
    cout << "Vect";
    cout << "Vect1 .:";
    std::for_each(intVect1.begin(),intVect1.end(),Output);
    cout << endl;
    cout << "Vect2 .:" << endl;
    std::for_each(intVect2.begin(),intVect2.end(),Output2);
    cout << endl;
    int result = inner_product(intVect1.begin(),intVect1.end(),
        intVect1.begin(),0);
    cout << endl;
    cout << "Result .:" << result << endl;
}

```

程序运行结果如图 13.28 所示。

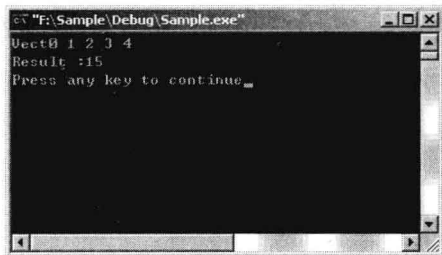


图 13.27 应用 accumulate 算法

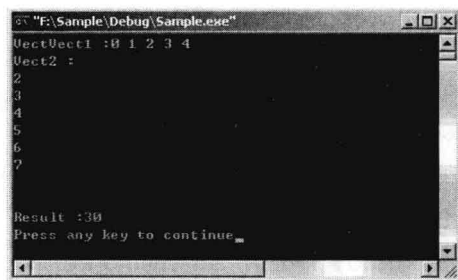


图 13.28 应用 inner\_product 算法

#### ☑ partial\_sum(first,last,result)

对迭代器 first 和 last 指明范围内的元素进行部分求和，然后把计算结果赋予从 result 开始的序列。部分求和的计算步骤为：首先把原序列的第 1 个元素赋值为新序列的第 1 个元素，然后把原序列的前两个元素相加，并把结果赋值给新序列的第 2 个元素，再把原序列的第 2 个和第 3 个元素相加，并把结果赋值给新序列的第 3 个元素，依此类推，直到计算完原序列的全部元素为止。

**【例 13.29】** 应用 partial\_sum 算法实现局部总和。(实例位置：光盘\TM\sl\13\29)

```

//partial_sum
#include<iostream>
#include<vector>
#include<algorithm>
#include<numeric>
using namespace std;
void Output(int val)
{
    cout << val << ' ';
}
void Output2(int val)
{
    cout << val << endl;
}

```



```

}
void main()
{
    vector<int> intVect1;
    vector<int> intVect2(5);
    for(int i=3;i<15;i+=2)
        intVect1.push_back(i);
    cout << "Vect1 :";
    std::for_each(intVect1.begin(),intVect1.end(),Output);
    cout << endl;
    partial_sum(intVect1.begin(),intVect1.end(),intVect2.begin());
    cout << "Vect2 : " << endl;
    std::for_each(intVect2.begin(),intVect2.end(),Output2);
    cout << endl;
}

```

程序运行结果如图 13.29 所示。

#### ☒ adjacent\_difference(first,last,result)

计算迭代器 `first` 和 `last` 指明范围内相邻元素的差值，并把结果赋予从 `result` 开始的序列。相邻差值的计算步骤为：首先把原序列的第 1 个元素赋值给新序列的第 1 个元素，再把原序列的第 2 个元素减去第 1 个元素，并把结果赋值给新序列的第 2 个元素，然后把原序列的第 3 个元素减去第 2 个元素，并把结果赋值给新序列的第 3 个元素，依此类推，直到计算完原序列的全部元素为止。此函数的另一个版本以谓词作为第 4 个实参。

**【例 13.30】** 应用 `adjacent_difference` 算法。（实例位置：光盘\TM\sl\13\30）

```

adjacent_difference
#include<iostream>
#include<vector>
#include<algorithm>
#include<numeric>
using namespace std;
void Output(int val)
{
    cout << val << ' ';
}
void main()
{
    vector<int> intVect1;

    intVect1.push_back(7);
    intVect1.push_back(3);
    intVect1.push_back(5);
    cout << "Vect1 :";
    std::for_each(intVect1.begin(),intVect1.end(),Output);
    cout << endl;
    vector<int> intVect2(intVect1.size());
    adjacent_difference(intVect1.begin(),intVect1.end(),intVect2.begin());
}

```

```

partial_sum(intVect1.begin(),intVect1.end(),intVect2.begin());
cout << "Vect2 :";
std::for_each(intVect2.begin(),intVect2.end(),Output);
cout << endl;
}

```

程序运行结果如图 13.30 所示。

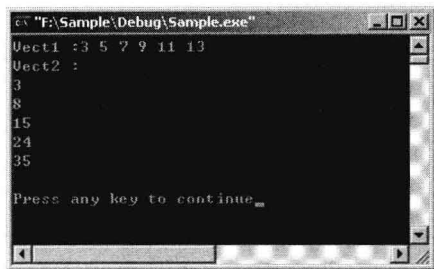


图 13.29 应用 partial\_sum 算法实现局部总和

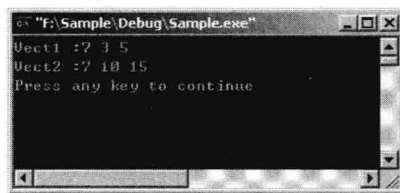


图 13.30 应用 adjacent\_difference 算法

## 13.4 迭代器

 视频讲解：光盘\TM\lx\13\迭代器.exe

迭代器相当于指向容器元素的指针，它在容器内可以向前移动，也可以作向前或向后双向移动。有专为输入元素准备的迭代器，有专为输出元素准备的迭代器，还有可以进行随机操作的迭代器，这为访问容器提供了通用方法。

### 13.4.1 输出迭代器

输出迭代器只用于写一个序列，它可以进行递增和提取操作。

**【例 13.31】** 应用输出迭代器。（实例位置：光盘\TM\sl\13\31）

```

#include<iostream>
#include<vector>
using namespace std;
void main()
{
    vector<int> intVect;
    for(int i=0;i<10;i+=2)
        intVect.push_back(i);
    cout << "Vect : " << endl;
    vector<int>::iterator it=intVect.begin();
    while(it!=intVect.end())
        cout << *it++ << endl;
}

```

程序运行结果如图 13.31 所示。

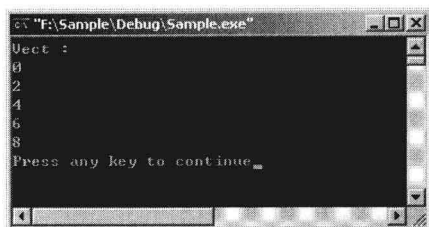


图 13.31 应用输出迭代器

程序使用整型向量的输出迭代器，输出向量中的所有元素。

### 13.4.2 输入迭代器

输入迭代器只用于读一个序列，它可以进行递增、提取和比较操作。

【例 13.32】 应用输入迭代器。（实例位置：光盘\TM\sl\13\32）

```
#include<iostream>
#include<vector>
using namespace std;
void main()
{
    vector<int> intVect(5);
    vector<int>::iterator out=intVect.begin();
    *out++ = 1;
    *out++ = 3;
    *out++ = 5;
    *out++ = 7;
    *out=9;
    cout << "Vect :";
    vector<int>::iterator it =intVect.begin();
    while(it!=intVect.end())
        cout << *it++ << ' ';
    cout << endl;
}
```

程序运行结果如图 13.32 所示。

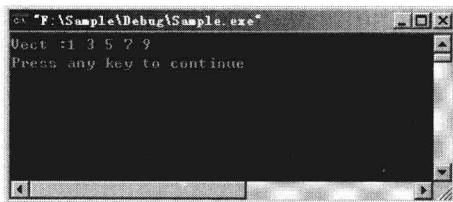


图 13.32 应用输入迭代器

程序使用输入迭代器向向量容器内添加元素，最后将添加的元素输出到屏幕。

### 13.4.3 前向迭代器

前向迭代器既可用于读，也可用于写，它不仅具有输入和输出迭代器的功能，还具有保存其值的功能，从而能够从迭代器原来的位置开始重新遍历序列。

**【例 13.33】** 应用前向迭代器。（实例位置：光盘\TM\sl\13\33）

```
#include<iostream>
#include<vector>
using namespace std;
void main()
{
    vector<int> intVect(5);
    vector<int>::iterator it=intVect.begin();
    vector<int>::iterator saveIt=it;
    *it++ = 12;
    *it++ = 21;
    *it++ = 31;
    *it++ = 41;
    *it=9;
    cout << "Vect :";
    while(saveIt!=intVect.end())
        cout << *saveIt++ << ' ';
    cout << endl;
}
```

程序运行结果如图 13.33 所示。

程序中使用 saveIt 迭代器保存了 it 迭代器的内容，并使用 it 迭代器向容器中添加元素，通过 saveIt 迭代器将容器内的元素输出。

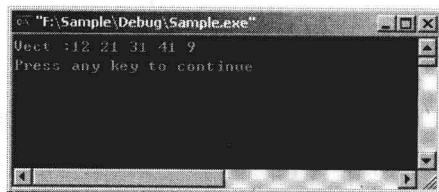


图 13.33 应用前向迭代器

### 13.4.4 双向迭代器

双向迭代器既可用于读，也可用于写，它与前向迭代器类似，只是双向迭代器可作递增和递减操作。

**【例 13.34】** 应用双向迭代器。（实例位置：光盘\TM\sl\13\34）

```
#include<iostream>
#include<vector>
using namespace std;
void main()
{
    vector<int> intVect(5);
    vector<int>::iterator it=intVect.begin();
```



```

vector<int>::iterator saveIt=it;
*it++ = 1;
*it++ = 3;
*it++ = 5;
*it++ = 7;
*it=9;
cout << "Vect :";
while(saveIt!=intVect.end())
    cout << *saveIt++ << ' ';
cout << endl;
do
    cout << *--saveIt << endl;
while(saveIt != intVect.begin());
cout << endl;
}

```

程序运行结果如图 13.34 所示。

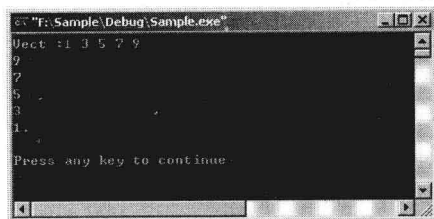


图 13.34 应用双向迭代器

程序中使用 saveIt 迭代器保存了 it 迭代器的内容，并使用 it 迭代器向容器中添加元素，通过 saveIt 迭代器以从前向后和从后向前两种顺序将容器内的元素输出。

### 13.4.5 随机访问迭代器

随机访问迭代器是最强大的迭代器类型，不仅具有双向迭代器的所有功能，还能使用指针的算术运算和所有比较运算。

**【例 13.35】** 应用随机访问迭代器。（实例位置：光盘\TM\sl\13\35）

```

#include<iostream>
#include<vector>
using namespace std;
void main()
{
    vector<int> intVect(5);
    vector<int>::iterator it=intVect.begin();
    *it++ = 1;
    *it++ = 3;
    *it++ = 5;
    *it++ = 7;
    *it=9;
}

```



```
cout << "Vect Old:";
for(it=intVect.begin();it!=intVect.end();it++)
    cout << *it << ' ';
it= intVect.begin();
*(it+2)=100;
cout << endl;
cout << "Vect :";
for(it=intVect.begin();it!=intVect.end();it++)
    cout << *it << ' ';
cout << endl;
}
```

程序运行结果如图 13.35 所示。

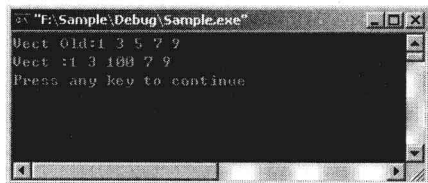


图 13.35 应用随机访问迭代器

## 13.5 小 结

本章主要介绍了标准模板库中的容器、算法和迭代器。这三者是标准模板库的核心内容，并且相互联系非常密切。迭代器是访问容器中的元素，算法是对容器中的元素进行操作。每种容器都有各自的特点，只有熟练掌握这些特点才能将标准模板库的作用充分发挥，并且应尽可能多地使用标准模板库提供的算法，这样可以节省许多开发时间。

## 13.6 实践与练习


1. 使用元素颠倒算法 `reverse`，实现对 `vector` 容器内的元素进行颠倒排序。（答案位置：光盘\TM\sl\13\36）
2. 使用删除容器中重复元素算法 `unique`，实现对 `vector` 容器内重复元素的删除。（答案位置：光盘\TM\sl\13\37）



# 第14章

---

## RTTI 与异常处理

(  视频讲解：22 分钟 )

面向对象编程的一个特点是运行时进行类型识别，这是对面向对象中多态的支持，使用 RTTI 能够使类的设计更加抽象，更加符合人们的思维。对象的动态生成，能够增加设计的灵活性，而异常处理则是在程序运行时对可能发生的错误进行控制，防止系统灾难性错误的发生。

通过阅读本章，您可以：

- » 掌握 RTTI 的使用
- » 了解什么是异常
- » 掌握异常的捕获
- » 掌握标准异常

## 14.1 RTTI（运行时类型识别）

 视频讲解：光盘\TM\lx\14\RTTI（运行时类型识别）.exe

运行时类型识别（Run-time Type Identification, RTTI）是在只有一个指向基类的指针或引用时所确定的一个对象的类型。

在编写程序的过程中，往往只提供了一个对象的指针，但通常在使用时需要明确这个指针的确切类型。利用 RTTI 就可以方便地获取某个对象指针的确切类型并进行控制。

### 14.1.1 什么是 RTTI

RTTI 可以在程序运行时通过某一对象的指针确定该对象的类型。许多程序设计人员都使用过虚基类编写面向对象的功能，通常在基类中定义了所有子类的通用属性或行为。但有些时候子类会存在属于自己的一些公有的属性或行为，这时通过基类对象的指针如何调用子类特有的属性和行为呢？首先，需要确定的就是这个基类对象属于哪个子类，然后将该对象转换成子类对象再进行调用。

如图 14.1 展示了具有特有功能的类。

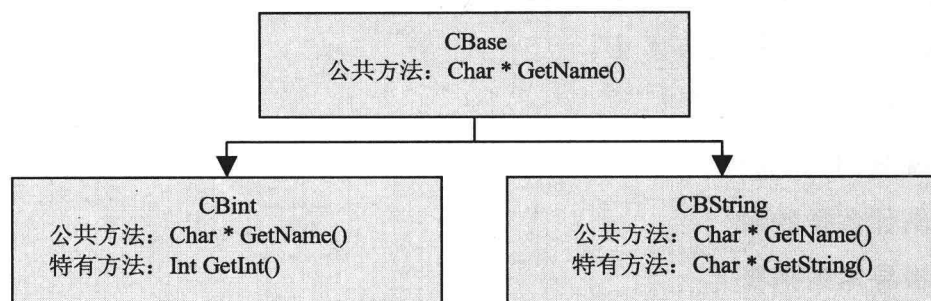


图 14.1 具有特有功能的类

从图 14.1 可以看出 CBint 类和 CString 类都继承于 CBase 类，这 3 个类存在一个公共方法 GetName()，而 CBint 类有自己的特有方法 GetInt()，CString 类有自己的特有方法 GetString()。如果想通过 CBase 类的指针调用 CBint 类或 CString 类的特有方法时就必须确定指针的具体类。下面的代码完成了这样的功能。

```

class CBase                                //基类
{
public:
    virtual char * GetName()=0;             //虚方法
};

class CBint:public CBase
{
public:

```

```

    char * GetName() { return "CBint"; }
    int GetInt(){ return 1; }
};

class CBString:public CBase
{
public:
    char * GetName() { return "CBString"; }
    char * GetString(){ return "Hello"; }
};

int main(int argc, char* argv[])
{
    CBase * B1 = (CBase *)new CBint();
    printf(B1->GetName());
    CBint *B2 = static_cast<CBint*>(B1);           //静态转换
    if(B2)
        printf("%d",B2->GetInt());
    CBase * C1 = (CBase *)new CBString();
    printf(C1->GetName());
    CBString *C2 = static_cast<CBString *>(C1);
    if(C2)
        printf(C2->GetString());
    return 0;
}

```

从上面代码可以看出基类 CBase 的指针 B1 和 C1 分别指向了 CBint 类与 CBString 类的对象,并且在程序运行时基类通过 static\_cast 进行了转换,这样就形成了一个运行时类型识别的过程。

### 14.1.2 RTTI 与引用

RTTI 必须能与引用一起工作。指针与引用存在明显不同,因为引用总是由编译器逆向引用,而一个指针的类型或它指向的类型可能要检测。如下面的代码定义了一个子类和一个基类。

```

#include "stdafx.h"
#include "typeinfo.h"

class CB
{
public:
    int GetInt(){ return 1;};
};

class CI:public CB
{
};

```

通过下面的代码可以看出, typeid() 获取的指针是基类类型,而不是子类类型或派生类类型, typeid() 获取的引用是子类类型。



```
int main(int argc, char* argv[])
{
    CB *p = new CI();
    CB &t = *p;
    if(typeid(p) == typeid(CB*))
        printf("指针类型是基类类型! \n");
    if(typeid(p) != typeid(CI*))
        printf("指针类型不是子类类型! \n");
    if(typeid(t) == typeid(CB))
        printf("引用类型是基类类型! \n");
    return 0;
}
```

与此相反，指针指向的类型在 `typeid()` 看来是派生类而不是基类，而用一个引用的地址时产生的是基类而不是派生类。

```
if(typeid(*p) == typeid(CB))
    printf("指针类型是基类类型! \n");
if(typeid(*p) != typeid(CI))
    printf("指针类型不是子类类型! \n");
if(typeid(&t) == typeid(CB*))
    printf("引用类型是基类类型! \n");
if(typeid(&t) != typeid(CI*))
    printf("引用类型不是子类类型! \n");
```

### 14.1.3 RTTI 与多重继承

RTTI 是一个功能非常强大的功能，对于面向对象的编程方法，如果在类继承时使用了 `virtual` 虚基类，RTTI 仍然可以准确地获取对象在运行时的信息。

例如，下面的代码通过虚基类的形式继承了父类，通过 RTTI 获取基类指针对象的信息。

```
#include "stdafx.h"
#include "typeinfo.h"
#include "iostream.h"

class CB //基类
{
    virtual void dowork(){}; //虚方法
};

class CD1:virtual public CB
{
};

class CD2:virtual public CB
{
};
```

```

class CD3:public CD1,public CD2
{
public:
    char *Print(){ return "Hello";};
};

int main(int argc, char* argv[])
{
    CB * p = new CD3();           //向上转型
    cout << typeid(*p).name() << endl; //获取指针信息
    CD3 * pd3 = dynamic_cast<CD3*>(p); //动态转型
    if(pd3)
        cout << pd3->Print() << endl;
    return 0;
}

```

即使只提供一个 virtual 基类指针，typeid()也能准确地检测出实际对象的名字。用动态映射同样也会工作得很好，但编译器不允许试图用原来的方法强制映射。例如：

```
CD3 *pd3 = (CD3 *)p;           //错误转换
```

编译器知道这不可能正确，所以它要求用户使用动态映射。

### 14.1.4 RTTI 映射语法

无论什么时候用类型映射，都是在打破类型系统，这实际上是在告诉编译器，即使知道一个对象的确切类型，还是可以假定认为它是另外一种类型。这本身就是一件很危险的事情，也是一个容易发生错误的地方。

为了解决这个问题，C++用保留字 dynamic\_cast、const\_cast、static\_cast 和 reinterpret\_cast 提供了一个统一的类型映射语法，为需要进行动态映射时提供了一个解决问题的可能。这意味着那些已有的映射语法已经被重载得太多，不能再支持任何其他的功能了。

☑ dynamic\_cast: 用于安全类型的向下映射。

例如，通过 dynamic\_cast 实现基类指针的向下转型。

```

#include "stdafx.h"
#include "iostream.h"
class CBase
{
public:
    virtual void Print(){ cout << "CBase" << endl; }
};

class CChild:public CBase
{
public:

```

```

    void Print(){ cout << "CChild" << endl; }
};

int main(int argc, char* argv[])
{
    CBase *p = new CChild();
    p->Print();
    CChild *d = dynamic_cast<CChild*>(p);
    d->Print();
    return 0;
}

```

☒ **const\_cast**: 用于映射常量和变量。

如果想把一个 const 转换为非 const，就要用到 const\_cast。这是可以用 const\_cast 的唯一转换，如果还有其他的转换牵涉进来，它必须分开来指定，否则会有一个编译错误。

例如，在常方法中修改成员变量和常量的值。

```

#include "stdafx.h"
#include "iostream.h"

class CX
{
protected:
    int m_count;
public:
    CX(){m_count = 10;}
    void f() const                //常方法，不能修改成员变量
    {
        (const_cast<CX*>(this))->m_count = 8;    //修改成员变量
        cout << m_count << endl;
    }
};

int main(int argc, char* argv[])
{
    CX *p = new CX();
    p->f();
    const int i = 10;            //常量
    int *n = const_cast<int*>(&i);    //转为非常量
    *n = 5;
    cout << *n << endl;
    return 0;
}

```

☒ **static\_cast**: 为了行为良好和行为较好使用的映射，如向上转型和类型自动转换。

例如，通过 static\_cast 将子类指针向上转成基类指针。

```

#include "stdafx.h"
#include "iostream.h"

```



```

class CB                                //基类
{
public:
    virtual void print(){ cout << "class CB" << endl;} //虚方法
};

class CD:public CB                      //派生类
{
public:
    void print(){ cout << "class CD" << endl;} //覆盖
};

int main(int argc, char* argv[])
{
    CD *p = new CD();
    p->print();
    CB *b = static_cast<CB*>(p);        //向上转型
    b->print();
    return 0;
}

```

☑ `reinterpret_cast`: 将某一类型映射回原有类型时使用。

例如, 将整型转成字符型, 再由 `reinterpret_cast` 转换回原类型。

```

#include "stdafx.h"
#include "iostream.h"

int main(int argc, char* argv[])
{
    int n = 97;
    char p[4] = {0};
    p[0] = (char)n;
    cout << p << endl;
    int *f = reinterpret_cast<int*>(&p); //定义与整型大小相同的字符数组
                                         //第一个元素为 97
    cout << *f << endl;
    return 0;
}

```

## 14.2 异常处理

 视频讲解: 光盘\TM\14\异常处理.exe

异常处理是程序设计中除调试之外的另一种错误处理方法, 它往往被大多数程序设计人员在实际设计中忽略。异常处理引起的代码膨胀将不可避免地增加程序阅读的困难, 这对于程序设计人员来说是十分烦恼的。异常处理与真正的错误处理有一定区别, 异常处理不但可以对系统错误做出反应, 还

可以对人为制造的错误做出反应并处理。下面将向读者介绍 C++ 语言对于异常处理的方法。

### 14.2.1 抛出异常

当程序执行到某一函数或方法内部时，程序本身出现了一些异常，但这些异常并不能由系统所捕获，这时就可以创建一个错误信息，再由系统捕获该错误信息并处理。创建错误信息并发送这一过程就是抛出异常。

最初异常信息的抛出只是定义一些常量，这些常量通常是整型值或是字符串信息。下面的代码是通过整型值创建的异常抛出。

```
#include "stdafx.h"
#include "iostream.h"

int main(int argc, char* argv[])
{
    try
    {
        throw 1;           //抛出异常
    }
    catch(int error)
    {
        if(error == 1)      //异常信息
            cout << "产生异常" << endl;
    }
    return 0;
}
```

在 C++ 语言中，异常的抛出是使用 `throw` 关键字来实现的，在这个关键字的后面可以跟随任何类型的值。在上面的代码中将整型值 1 作为异常信息抛出，当异常捕获时就可以根据该信息进行异常的处理。

异常的抛出还可以使用字符串作为异常信息进行发送，代码如下：

```
#include "stdafx.h"
#include "iostream.h"

int main(int argc, char* argv[])
{
    try
    {
        throw "异常产生! "; //抛出异常
    }
    catch(char * error)
    {
        cout << error << endl;
    }
    return 0;
}
```



可以看到，字符串形式的异常信息适合于异常信息的显示，但并不适合于异常信息的处理。那么是否可以将整型信息与字符串信息结合起来作为异常信息进行抛出呢？之前说过，`throw` 关键字后面跟随的是类型值，所以不但可以跟随基本数据类型的值，还可以跟随类类型的值，这就可以通过类的构造函数将整型值与字符串结合在一起，并且还可以同时应用更加灵活的功能。

例如，将错误 ID 和错误信息以类对象的形式进行异常抛出。

```
#include "stdafx.h"
#include "iostream.h"
#include "string.h"

class CCustomError                                //异常类
{
private:
    int m_ErrorID;                                //异常 ID
    char m_Error[255];                            //异常信息
public:
    CCustomError(int ErrorID,char *Error)          //构造函数
    {
        m_ErrorID = ErrorID;
        strcpy(m_Error,Error);
    }
    int GetErrorID(){ return m_ErrorID; }          //获取异常 ID
    char * GetError(){ return m_Error; }          //获取异常信息
};

int main(int argc, char* argv[ ])
{
    try
    {
        throw (new CCustomError(1,"出现异常! ")); //抛出异常
    }
    catch(CCustomError* error)
    {
        //输出异常信息
        cout << "异常 ID: " << error->GetErrorID() << endl;
        cout << "异常信息: " << error->GetError() << endl;
    }
    return 0;
}
```

代码中定义了一个异常类，这个类包含了两个内容，一个是异常 ID，也就是异常信息的编号；另一个是异常信息，也就是异常的说明文本。通过 `throw` 关键字抛出异常时，需要指定这两个参数。

### 14.2.2 异常捕获

异常捕获是指当一个异常被抛出时，不一定就在异常抛出的位置来处理这个异常，而是可以在别的地方通过捕获这个异常信息后再进行处理。这样不仅增加了程序结构的灵活性，也提高了异常处理

的方便性。

如果在函数内抛出一个异常（或在函数调用时抛出一个异常），将在异常抛出时退出函数。如果不想在异常抛出时退出函数，可在函数内创建一个特殊块用于解决实际程序中的问题。这个特殊块由 `try` 关键字组成，例如：

```
try
{
//抛出异常
}
```

异常抛出信号发出后，一旦被异常处理器接收到就被销毁。异常处理器应具备接收任何异常的能力。异常处理器紧随 `try` 块之后，处理的方法由关键字 `catch` 引导。

```
try
{
}
catch(type obj)
{
}
```

异常处理部分必须直接放在测试块之后。如果一个异常信号被抛出，异常处理器中第一个参数与异常抛出对象相匹配的函数将捕获该异常信号，然后进入相应的 `catch` 语句，执行异常处理程序。`catch` 语句与 `switch` 语句不同，它不需要在每个 `case` 语句后加入 `break` 去中断后面程序的执行。

下面通过 `try...catch` 语句来捕获一个异常。代码如下：

```
#include "stdafx.h"
#include "iostream.h"
#include "string.h"

class CcustomError           //异常类
{
private:
    int m_ErrorID;           //异常 ID
    char m_Error[255];       //异常信息
public:
    CcustomError()           //构造函数
    {
        m_ErrorID = 1;
        strcpy(m_Error, "出现异常!");
    }
    int GetErrorID(){ return m_ErrorID; } //获取异常 ID
    char * GetError(){ return m_Error; }  //获取异常信息
};

int main(int argc, char* argv[])
{
    try
    {
```

```

        throw(new CCustomError());           //抛出异常
    }
    catch(CCustomError* error)
    {
        //输出异常信息
        cout << "异常 ID: " << error->GetErrorID() << endl;
        cout << "异常信息: " << error->GetError() << endl;
    }
    return 0;
}

```

在上面的代码中可以看到 try 语句块用于捕获 throw 所抛出的异常。对于 throw 异常的抛出，可以直接写在 try 语句块的内部，也可以写在函数或类方法的内部，但函数或方法必须写在 try 语句块的内部才可以捕获到异常。

异常处理器可以成组的出现，同时根据 try 语句块获取的异常信息处理不同的异常。代码如下：

```

int main(int argc, char* argv[])
{
    try
    {
        throw "字符串异常! ";
        //throw(new CCustomError());           //抛出异常
    }
    catch(CCustomError* error)
    {
        //输出异常信息
        cout << "异常 ID: " << error->GetErrorID() << endl;
        cout << "异常信息: " << error->GetError() << endl;
    }
    catch(char * error)
    {
        cout << "异常信息: " << error << endl;
    }
    return 0;
}

```

有时并不一定在列出的异常处理中包含所有可能发生的异常类型，所以 C++ 提供了可以处理任何类型异常的方法，就是在 catch 后面的括号内添加“...”。代码如下：

```

int main(int argc, char* argv[])
{
    try
    {
        throw "字符串异常! ";
        //throw(new CCustomError());           //抛出异常
    }
    catch(CCustomError* error)
    {
        //输出异常信息
    }
}

```



```

        cout << "异常 ID: " << error->GetErrorID() << endl;
        cout << "异常信息: " << error->GetError() << endl;
    }
    catch(char * error)
    {
        cout << "异常信息: " << error << endl;
    }
    catch(...)
    {
        cout << "未知异常信息! " << endl;
    }
    return 0;
}

```

有时需要重新抛出刚接收到的异常，尤其是在程序无法得到有关异常的信息而用省略号捕获任意的异常时。这些工作通过加入不带参数的 `throw` 就可完成，例如：

```

catch (...) {
    cout << "未知异常! " << endl;
    throw;
}

```

如果一个 `catch` 语句忽略了一个异常，那么这个异常将进入更高层的异常处理环境。由于每个异常抛出的对象是被保留的，所以更高层的异常处理器可抛出来自这个对象的所有信息。

### 14.2.3 异常匹配

当程序中有异常抛出时，异常处理系统会根据异常处理器的顺序找到最近的异常处理块，并不会搜索更多的异常处理块。

异常匹配并不要求异常与异常处理器进行完美匹配，一个对象或一个派生类对象的引用将与基类处理器进行匹配。若抛出的是类对象的指针，则指针会匹配相应的对象类型，但不会自动转换成其他对象的类型。例如：

```

#include "stdafx.h"

class CExcept1{ };
class CExcept2
{
public:
    CExcept2(CExcept1& e){ }
};

int main(int argc, char* argv[ ])
{
    try
    {
        throw CExcept1();
    }
}

```

```

    }
    catch (CExcept2)
    {
        printf("进入 CExcept2 异常处理器! \n");
    }
    catch(CExcept1)
    {
        printf("进入 CExcept1 异常处理器! \n");
    }
    return 0;
}

```

从上面代码可以认为第一个异常处理器会使用构造函数进行转换，将 CExcept1 转换为 CExcept2 对象，但实际上系统在异常处理期间并不会执行这样的转换，而是在 CExcept1 处终止。

通过下面的代码演示基类处理器如何捕获派生类的异常。

```

#include "stdafx.h"
#include "iostream.h"

class CExcept
{
public:
    virtual char *GetError(){ return "基类处理器"; }
};

class CDerive : public CExcept
{
public:
    char *GetError(){ return "派生类处理器"; }
};

int main(int argc, char* argv[])
{
    try
    {
        throw CDerive();
    }
    catch(CExcept)
    {
        cout << "进入基类处理器\n";
    }
    catch(CDerive)
    {
        cout << "进入派生类处理器\n";
    }
    return 0;
}

```

从上面的代码可以看出，虽然抛出的异常是 CDerive 类，但由于异常处理器的第一个是 CExcept 类，该类是 CDerive 类的基类，所以将进入此异常处理器内部。为了正确地进入指定的异常处理器，



在对异常处理器进行排列时应将派生类排在前面，而将基类排在后面。

## 14.2.4 标准异常

用于 C++ 标准库的一些异常可以直接应用到程序中，应用标准异常类会比应用自定义异常类简单容易得多。如果系统提供的标准异常类不能满足需要，就不可以在这些标准异常类基础上进行派生。下面给出了 C++ 提供的一些标准异常：

```
namespace std
{
    //exception 派生
    class logic_error;           //逻辑错误，在程序运行前可以检测出来
    //logic_error 派生
    class domain_error;         //违反了前置条件
    class invalid_argument;     //指出函数的一个无效参数
    class length_error;         //指出有一个超过类型 size_t 的最大可表现值长度的对象的企图
    class out_of_range;        //参数越界
    class bad_cast;             //在运行时类型识别中有一个无效的 dynamic_cast 表达式
    class bad_typeid;           //报告在表达式 typeid(*p) 中有一个空指针 p
    //exception 派生
    class runtime_error;        //运行时错误，仅在程序运行中检测到
    //runtime_error 派生
    class range_error;          //违反后置条件
    class overflow_error;       //报告一个算术溢出
    class bad_alloc;           //存储分配错误
}
```

注意观察上述类的层次结构可以看出，标准异常都派生自一个公共的基类 `exception`。基类包含必要的多态性函数提供异常描述，可以被重载。下面是 `exception` 类的原型。

```
class exception
{
public:
    exception() throw();
    exception(const exception& rhs) throw();
    exception& operator=(const exception& rhs) throw();
    virtual ~exception() throw();
    virtual const char* what() const throw();
};
```

## 14.3 小 结

本章主要介绍 RTTI 的使用以及如何进行异常处理。通过运行时类型识别的学习，可以丰富类的设

计思路，加强对面向对象的理解，有助于理解类间的类型转换。程序中出现异常是不可避免的，异常处理则能够帮助程序开发人员尽快发现错误所在。为了减少错误的发生，应尽量掌握更多的异常处理方式。

## 14.4 实践与练习

三角形面积公式为  $s = \sqrt{p(p-a)(p-b)(p-c)}$ 。

其中  $p = \frac{a+b+c}{2}$ 。

式中  $a, b, c$  是三角形边长。

编写一个求三角形面积的程序。

程序中可以捕获以下几类异常：

- (1) 边长不正确，要求应该是正数。
- (2) 边长组合不正确，要求任意两边之和大于第三边。（实例位置：光盘\TM\sl\14\1）



# 第15章

---

## 程序调试

(  视频讲解：33 分钟 )

程序调试是程序开发人员必不可少的一项工作，甚至占去了程序开发人员大部分的开发时间。正确的使用调试方法不但可以提高工作效率，而且可以减轻程序开发人员的工作负担。本章主要介绍了程序错误的常见类型、常用的调试工具、调试的基本应用和高级应用。

通过阅读本章，您可以：

- » 了解程序错误常见的 4 种类型
- » 掌握调试工具的使用
- » 掌握调试的基本应用
- » 熟悉调试的高级应用

## 15.1 选择正确的调试方法

 视频讲解：光盘\TM\lx\15\选择正确的调试方法.exe

在程序设计的过程中，无论你有多么丰富的编程经验，或是多么熟练的技术水平，都会不可避免地出现程序错误。有经验的程序员早就认识到，解决程序错误的最好方法就是使用调试。任何一种程序开发工具都具有其独立的调试系统，而且十分成熟，所以只要掌握好开发工具自身的调试系统，就可以很好地解决编程过程中出现的错误。

在程序开发的过程中，逻辑错误和运行时错误通常是最普遍而且是最容易发生的，所以当程序出现错误时，首先要确定程序是因逻辑错误还是运行时错误而导致的，再根据情况选择适当的调试方法解决错误。

## 15.2 程序错误常见的 4 种类型

 视频讲解：光盘\TM\lx\15\程序错误常见的 4 种类型.exe

应用程序可能遇到的错误类型主要有 4 种：语法错误、连接错误、运行时错误和逻辑错误。这些错误中的大多数均发生在所编写的 C++ 程序向其可执行形式转化的过程中。另外，在程序编译的过程中由于所要建立的可执行文件版本的不同（调试版本和发行版本），可能出现不同的错误后果。

### 15.2.1 语法错误

在程序设计过程中，不论是初学者还是经验丰富的程序员都会或多或少的发生语法错误。语法错误就是在编写程序代码时违反了 C++ 的语法规则，而一旦违反了语法规则，在程序进行编译时开发工具就会提示编译出错的信息。

例如，在编写代码时最常使用的字符串输出功能：

```
#include "stdafx.h"

int main(int argc, char* argv[])
{
    printf("Hello World!\n")
    printf("大家好！");
    return 0;
}
```

通过编译上面的代码系统会提示这样的错误信息：

```
C:\demo11\demo11.cpp(9) : error C2146: syntax error : missing ';' before identifier 'printf'
Error executing cl.exe.
demo11.exe - 1 error(s), 0 warning(s)
```



当编译器将源代码翻译为可以在计算机上运行的可执行程序时，编译器将试图定位和报告出尽可能多的语法错误，并对合法但以后可能引起错误的潜在问题给出警告。Visual C++编译器可以报告出声明但从未使用过的变量和在初始化之前对变量的使用，甚至可以指出所产生的逻辑流程将禁止某些代码段被执行——这种问题被标记为 **unreachable code**（不可到达的代码）。

上面的提示信息明确地指出在 `printf` 语句中缺少“;”号，但有些初学编程的读者却根本不看编译器所提示的信息，而是花费大量时间来找程序出错的原因，而有经验的程序员看到编译器提示的信息后便立刻修正了程序中的错误。

## 15.2.2 连接错误

出现连接错误最常见的一种情况就是在使用动态链接库时，虽然已对 `lib` 文件进行了载入，但 `lib` 文件与动态链接库所在的位置和可执行文件并不在同一个目录下，导致程序在进行编译时出现连接错误。连接错误与其他的错误类型在本质上是有所区别的，如语法错误是由于违反语法规则产生的，而连接错误则是由于在编译可执行文件时缺少外部连接文件所产生的。

例如，定义了一个名为 `demodll.lib` 的动态链接库，通过应用程序调用此 DLL 中的函数。

在应用程序的头文件中添加如下代码：

```
extern "C" __declspec(dllexport) void ShowHello();  
#pragma comment(lib,"demodll.lib")
```

在对程序进行编译时，编译器会寻找名为 `demodll.lib` 的文件，这个文件是与 `demodll.dll` 文件一起编译生成的。但如果这个 `lib` 文件并没有放在可执行文件的相同目录下，在进行编译时会提示下面的错误信息：

```
Linking...  
LINK : fatal error LNK1104: cannot open file "demodll.lib"  
Error executing link.exe.  
demo12.exe - 1 error(s), 0 warning(s)
```

通过上面的错误信息可以看出，在编译器连接 `demodll.lib` 文件时没有找到相应的文件，所以提示“文件无法打开”的错误信息。

## 15.2.3 运行时错误

运行时错误并不是在程序进行编译时产生的，而是在程序编译后没有出现任何错误提示的情况下程序运行时发生的异常现象。运行时错误与语法错误或连接错误不同，它在编译时并没有给出错误提示，所以不能简单的从提示信息上进行处理。但 Visual C++开发工具为编程人员提供了强大的错误处理能力，可以通过在程序代码中设置错误断点来检测并处理运行时错误。

例如，定义一个数组变量，并向这个数组变量中赋值。

```
#include "stdafx.h"  
#include "iostream.h"
```

```
int main(int argc, char* argv[])
{
    int array[10];
    for (int i = 0; i <= 10; i++)
        cin >> array[i];
    printf("Hello World!\n");
    return 0;
}
```

程序运行过程中，在为第 11 个元素赋值时将提示异常信息，如图 15.1 所示。

通过图中显示的错误信息可以知道是使用了不存在的内存地址，但不能确定是什么地方使用了不存在的内存地址。所以需要设置断点，跟踪程序执行的每一步，直到遇到程序出错的位置再判断异常产生的原因并处理。利用断点进行程序调试的方法将在后文介绍。

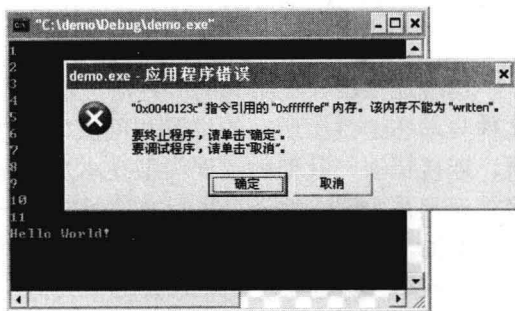


图 15.1 运行时错误信息

## 15.2.4 逻辑错误

逻辑错误是最难处理的一种错误类型，因为导致这种错误出现的原因是对某一问题解决方案的错理解，更糟糕的是编译器并不能捕获并处理逻辑错误。对于这种错误并不能看到任何错误信息，只能看到错误的结果或是导致程序的终止。在这种情况下就只能通过各种不同的数据来测试程序的运行结果是否正确。

虽然编译器不能捕获程序中的逻辑错误，但可以使用调试器或其他方法来解决逻辑错误。这里介绍两种常用的逻辑错误处理方法：

一是利用调试器设置断点，并跟踪程序执行的每条语句。在跟踪的同时对程序中的变量值进行验证，查看程序出现逻辑错误的位置。

二是利用字符串输出语句，在程序中需要输出验证信息的位置将变量值以字符串的形式进行输出。这样就可以很快的查看出程序中出现逻辑错误的位置。

## 15.3 调试工具的使用

 视频讲解：光盘\TM\lx\15\调试工具的使用.exe

初学编程的读者在实际的编程过程中会遇到许多问题，能快速找到并解决这些问题的唯一方法就是使用程序调试。程序调试在每个语言的集成开发环境中都存在，只是一些初学者将这一功能忽略，导致遇到程序错误时不知所措。

### 15.3.1 创建调试程序

在进入程序调试的学习之前，先来创建一个用于程序调试的程序。创建步骤如下：

- (1) 选择 File（文件）菜单中的 New（新建）命令，打开 New（新建）对话框。
- (2) 在 Project name 文本框中输入“DebugProject”，在左侧的工程列表中选择 Win32 Console Application 选项用于创建一个控制台应用程序，如图 15.2 所示。
- (3) 单击 OK 按钮进入下一个页面，在这个页面中选 A "Hello,World!" application 单选按钮，创建一个 Hello World 工程，单击 Finish 按钮完成工程的创建，如图 15.3 所示。

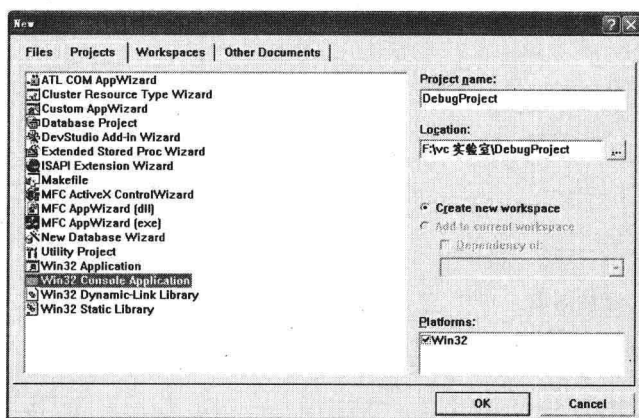


图 15.2 New 对话框

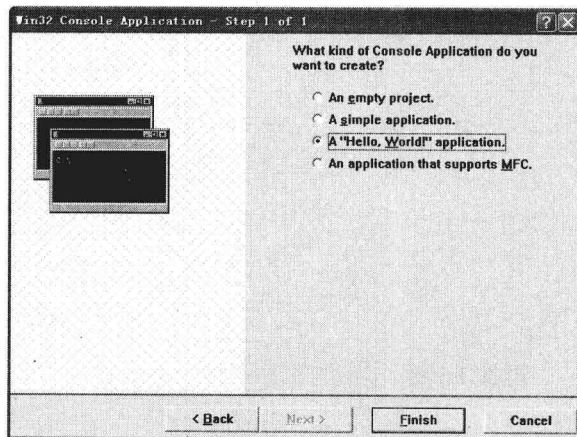


图 15.3 应用程序向导

- (4) 修改程序代码用于程序调试时使用，代码如下：

```
#include "stdafx.h"
#include "string.h"           //字符串函数头文件

int main(int argc, char* argv[])
{
    printf("Hello World!\n");
    //添加代码开始
    char *str = new char[100]; //定义字符串变量
    strcpy(str, "Hello Word!"); //给字符串赋值
    int s,a,b;                 //定义整型变量
    a = 5;                     //赋初值
    b = 10;
    s = a + b;                 //求和
    printf("str:%s\n", str);   //输出字符串
    printf("s:%d\n", s);       //输出求和结果
    //添加代码结束
    return 0;
}
```

程序运行结果如图 15.4 所示。

### 15.3.2 进入调试状态

按 F5 键即可进入调试状态，但此时由于没有指定断点，所以程序的运行与普通的运行没有什么区别。因此，在进入调试状态前应先将光标定位在断点所在行，然后按 F9 键添加断点，如图 15.5 所示。

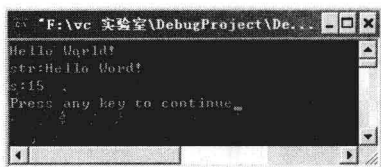


图 15.4 运行结果

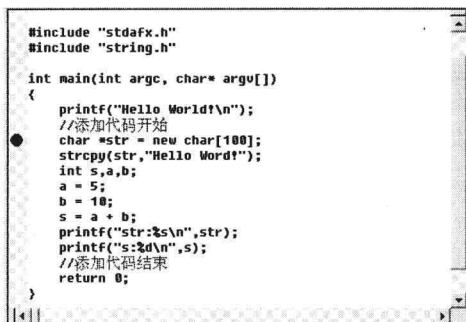


图 15.5 添加断点

再按 F5 键，当程序运行到断点所在行时就会停下来，这时就可以查看程序运行时的信息了。

### 15.3.3 Watch 窗口

Watch 窗口用于在调试期显示程序中所定义的变量或表达式的值，在代码编辑区中拖放选中的变量到该窗口，即可显示该变量的值。操作步骤如下：

(1) 进入调试状态，单击 Debug（调试）工具栏中的 Watch 按钮打开 Watch 窗口。

(2) 选中变量 s，然后将变量拖放到 Watch 窗口中，如图 15.6 所示。

(3) 按 F10 键单步执行到变量被赋值后的代码，即可看到变量值的变化。

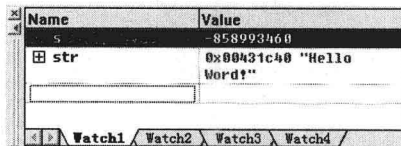


图 15.6 Watch 窗口

### 15.3.4 Call Stack 窗口

Call Stack 窗口用来显示栈中被调用但还未返回的函数。操作步骤如下：

(1) 进入调试状态，单击 Debug 工具栏中的 Call Stack 按钮打开 Call Stack 窗口。

(2) 由于在 main 函数中设置了断点，所以程序运行到断点处停止，Call Stack 窗口将显示 main 函数的信息，如图 15.7 所示。

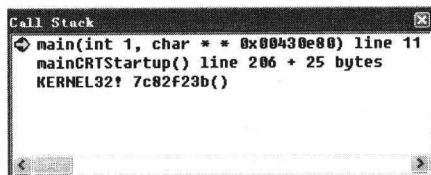


图 15.7 Call Stack 窗口

### 15.3.5 Memory 窗口

Memory 窗口用于显示内存中的数据。操作步骤如下：

- (1) 进入调试状态，单击 Debug 工具栏中的 Memory 按钮打开 Memory 窗口。
- (2) 单步运行程序到 strcpy 函数所在行，此时在 Watch 窗口中可看到 str 变量的地址，将该地址输入到 Memory 窗口中按 Enter 键，单步执行到下一行即可看到 str 变量在内存中的值，如图 15.8 所示。

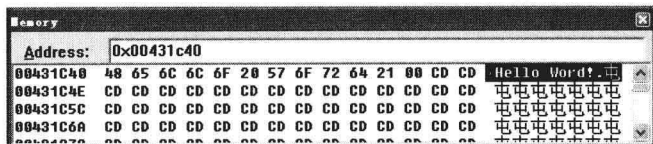


图 15.8 Memory 窗口

### 15.3.6 Variables 窗口

Variables 窗口用来显示变量的值，在这个窗口中所显示的变量是由系统自动生成的，不可以进行修改。进入调试状态后单击 Debug 工具栏中的 Variables 按钮打开该窗口，在窗口中有 3 个选项卡：Auto、Locals 和 this，如图 15.9 所示。

选项卡的功能说明如下。

- ☒ Auto (自动)：显示程序运行当前行和前一行所使用的变量的值。
- ☒ Locals (本地)：显示程序运行的当前函数中所包含的变量值。
- ☒ this (this 指针)：显示当前 this 指针所指向的类对象的信息。

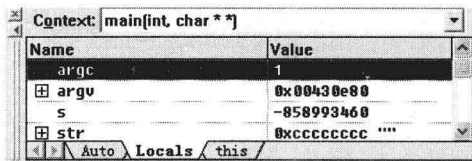


图 15.9 Variables 窗口

### 15.3.7 Registers 窗口

Registers 窗口显示了当前 CPU 各个寄存器的值。运行程序进入调试状态，在 Debug 工具栏中单击 Registers 按钮打开该窗口，通过该窗口可以查看寄存器信息，如图 15.10 所示。



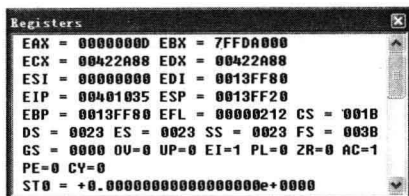


图 15.10 Registers 窗口

### 15.3.8 Disassembly 窗口

Disassembly 窗口用来显示程序源代码的反汇编代码。运行程序进入调试状态，单击 Debug 工具栏中的 Disassembly 按钮打开该窗口，如图 15.11 所示。

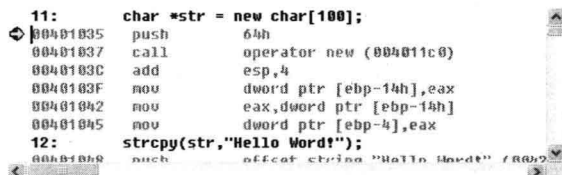


图 15.11 Disassembly 窗口

## 15.4 调试的基本应用

视频讲解：光盘\TM\15\调试的基本应用.exe

调试是解决程序出现错误的有效途径，通过简单的调试操作可以解决程序设计中出现的大部分错误。

### 15.4.1 变量的跟踪与查看

变量的跟踪与查看是在设置了断点后进入调试运行状态，按 F10 或 F11 键单步执行到变量所在的代码行时，在 Variables 窗口中将显示当前代码行中变量的值，这样就可以在程序运行时查看当前行变量的值，再根据这个值判断程序计算是否正确。

例如，循环输出一组数的累加和。

```
#include "stdafx.h"
#include "iostream.h"

int main(int argc, char* argv[])
{
    int sum = 0;
    for(int i = 0; i < 10; i++, i++)
    {
        sum += i;
    }
}
```

```

        cout << sum << '\n';
    }
    return 0;
}

```

当断点位于“sum +=i;”语句上时通过 Variables 窗口可以同时看到变量 sum 和 i 的值,如图 15.12 所示。

在 Variables 窗口中, Name 代表了变量的名称, Value 代表了变量的值。而且每次相应变量发生改变时 Value 中的值会自动发生改变。

使用 Auto 选项卡观察时存在一个问题,即其自动跟踪的范围过于狭窄。此时,可以选择 Variables 窗口的 Locals 选项卡,观察所有的局部变量。

Variables 窗口所显示的变量信息是由系统决定的,所以不能控制变量的显示范围。但通过 Watch 窗口可以自由地添加或删除当前需要显示的变量。同样使用上面的代码,进入调试运行状态,在调试工具栏中单击 Watch 按钮打开 Watch 窗口,在 Name 列中输入需要显示的变量的名称,如图 15.13 所示。

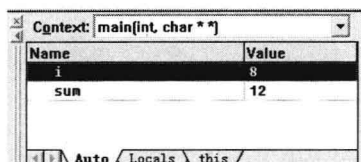


图 15.12 跟踪变量

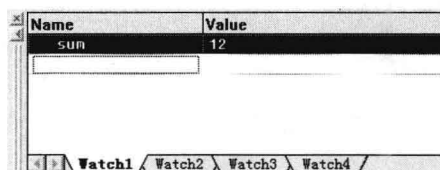


图 15.13 查看变量值

在 Watch 窗口中可以看到, sum 是由程序设计人员添写的,这样就可以根据需要在窗口中显示需要查看的变量。当需要删除变量的显示时,按 Delete 键即可。同时,还可以看到在 Watch 窗口的下方有多个选项卡,在每一个选项卡中都可以设置不同的变量进行跟踪查看。

## 15.4.2 位置断点的使用

位置断点就是在指定的位置上设置断点。通常程序设计人员使用的断点都属于位置断点,也就是在指定的代码行上按 F9 键添加一个断点,但这类断点只起到在程序调试运行到代码所在行时中断程序运行的作用。

位置断点的使用有一定的技巧,程序设计人员可以控制位置断点在什么条件下有效,这样就使断点和调试更加灵活地结合起来,更容易处理复杂的调试情况。

例如,在一个循环的过程中可能会出现异常,此时就可以使用位置断点。

```

#include "stdafx.h"
#include "iostream.h"

int main(int argc, char* argv[])
{
    int sum = 0;
    int n = 552;
    for(int i = 0; i < 100; i++,i++)

```

```

{
    sum += i;
    int k = sum;
    k /= (sum - n);
}
return 0;
}

```

在上面的代码执行时，for 循环语句中会出现异常。但由于循环的次数非常多，所以不能简单地使用单步调试。为了快速地达到调试的效果，在“sum += i;”这行代码上按 F9 键添加一个调试断点，然后按 Ctrl+B 组合键打开 Breakpoints 对话框，选中断点，如图 15.14 所示。

然后单击 Condition 按钮打开 Breakpoint Condition 对话框，在最下面的文本框中输入 skip（循环）的最大次数，单击 OK 按钮，如图 15.15 所示。

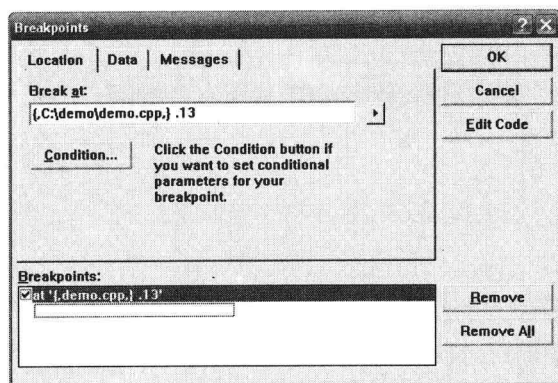


图 15.14 Breakpoints 对话框

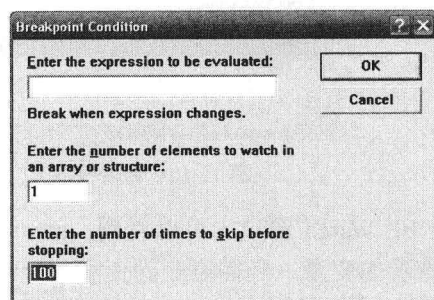


图 15.15 Breakpoint Condition 对话框

位置断点设置完成后，按 F5 键开始调试运行，此时程序运行报错。按 Ctrl+B 组合键打开 Breakpoints 对话框，在对话框的下方可以看到运行的总次数为 100 次，有 76 次没有运行，如图 15.16 所示。

关闭 Breakpoints 对话框，按 Shift+F5 组合键退出调试状态。按 Ctrl+B 组合键打开 Breakpoints 对话框，单击 Condition 按钮将 skip 的值由 100 修改为 23，单击 OK 按钮，如图 15.17 所示。

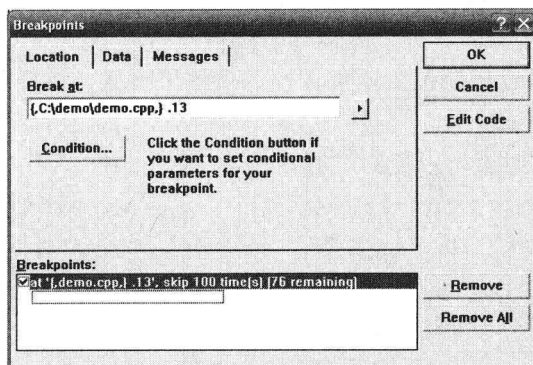


图 15.16 Breakpoints 对话框

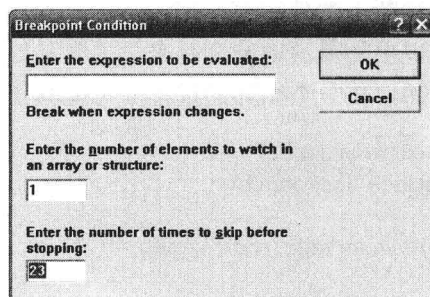


图 15.17 修改 skip 值

再按 F5 键调试运行，此时按 F10 键单步运行就可以立刻定位到程序出错的位置。

### 15.4.3 数据断点的使用

数据断点是对程序代码中的数据进行判断或操作引发的中断行为。数据断点与位置断点不同，是在程序的调试时期根据某变量的数据条件添加的断点，最常用于解决数据的溢出而导致的程序错误。

例如，在对字符串数组进行赋值时，很容易出现内存溢出的现象。

```
#include "stdafx.h"
#include "iostream.h"
#include "string.h"

int main(int argc, char* argv[])
{
    char str1[10];
    char str2[4];
    strcpy(str1, "mrsoft");
    printf("%s\n", str1);

    strcpy(str2, "mingribook");
    printf("%s\n", str1);
    printf("%s\n", str2);
    return 0;
}
```

上面代码的输出结果如图 15.18 所示。

从上面代码的运行结果可以看出程序存在错误，即 str1 在第二次输出时结果不正确。在程序中只为 str1 赋了一次值，为什么会出现两个结果呢？当问题不好解决时，可以使用数据断点跟踪所有对 str1 赋值的位置，然后查看出现错误的原因。

下面介绍如何使用数据断点解决字符串数组赋值时内存溢出的问题。首先在第一个 strcpy 出现的位置添加一个普通断点，按 F5 键进入调试运行状态，然后按 Ctrl+B 组合键打开 Breakpoints 对话框。在 Data 选项卡中设置需要查看变量的名字 str1，单击 OK 按钮，如图 15.19 所示。

当数据断点设置完成后，按 F10 键执行正确的数据赋值与数据输出语句。这是因为这些语句都会触发数据断点，所以需要跳过这些正确的修改语句，当断点定位在第 2 个 strcpy 函数所在的语句执行时，按 F5 键将会弹出 str1 数据发生改变的提示信息，如图 15.20 所示。

这个 strcpy 是给 str2 赋值的，为什么会修改 str1 的值呢？原来 str2 的长度只有 4 个字节，而在赋值时却超出了范围，所以就修改了 str1 的值。就这样，通过数据断点，快速地解决了程序中出现的异常错误。

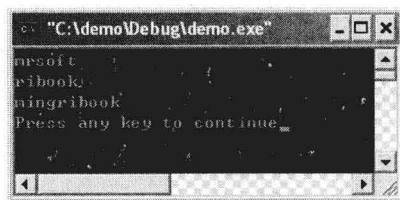


图 15.18 数组赋值



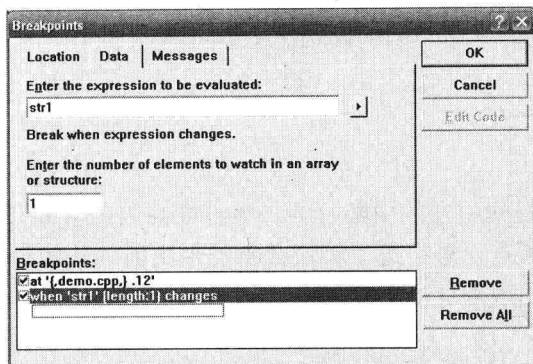


图 15.19 设置数据断点

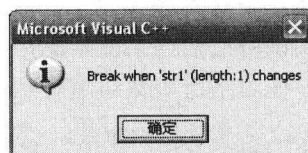


图 15.20 str1 数据发生改变

## 15.5 调试的高级应用

### 视频讲解：光盘\TM\15\调试的高级应用.exe

程序调试是编程过程中必不可少的一项工作，简单的程序错误通过简单的调试方法就可以解决，但有些程序错误只凭一些基本的调试技术是无法解决的，本节将向读者介绍一些高级的调试方法用于解决更为复杂的程序错误。

### 15.5.1 在调试时修改变量的值

Visual C++ 6.0 这个开发工具提供了强大的程序调试功能，不但可以满足各种调试的需要，还可以在程序的调试时期修改某一变量的值并继续调试。这一功能不但提高了程序调试的效率，还增加了调试的灵活性，使程序设计人员更好地解决程序中出现的错误。

例如，通过对字符的判断显示不同的结果。

```
#include "stdafx.h"
#include "iostream.h"
#include "string.h"

int main(int argc, char* argv[])
{
    char flgstr = 'Y';
    char outstr[255];
    if(flgstr == 'Y')
        strcpy(outstr, "今天晚上同学聚会。");
    else
        strcpy(outstr, "晚上同学聚会取消。");
    cout << outstr;
    return 0;
}
```



在代码的 if 语句上添加一个普通断点，按 F5 键调试运行。此时需要将 flgstr 的值由 'Y' 修改成 'N'，在 Variables 窗口中找到这个变量直接修改其值为 'N'，该变量将以红色显示，如图 15.21 所示。

按 F10 键继续运行，可以看到 outstr 的值是条件为 false 时的结果，如图 15.22 所示。

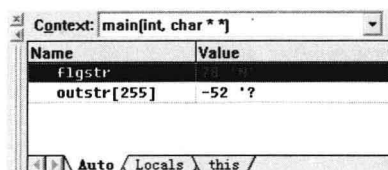


图 15.21 修改变量值

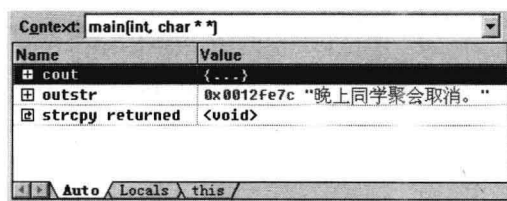


图 15.22 outstr 变量输出结果

通过上面的功能，程序设计人员就可以在调试过程中任意修改变量的值，而且不需要重新编译程序就可以继续调试。对于这一功能同样可以在 Watch 窗口中使用，如图 15.23 所示。

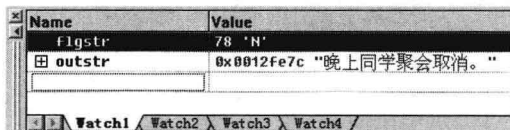


图 15.23 在 Watch 窗口中修改变量值

## 15.5.2 在循环中调试

在循环语句中进行调试是一项十分复杂的工作，由于循环的次数通常都会很多，所以单步调试可能需要花费大量的时间，这样就必须寻找另外的方法对循环进行调试。在 15.4 节中介绍位置断点时讲解了循环语句中调试的方法，使用这种方法虽然可以快速定位到程序出错的位置，但并不能跟踪变量的值。本节将介绍另一种循环调试的方法，这种方法就是利用 TRACE 宏将循环中所使用的变量结果显示在 Debug 窗口中。

例如，在一个循环的过程中可能会出现异常，就可以使用 TRACE 进行调试查看。

```
int sum = 0;
int n = 552;
for(int i = 0; i < 100; i++, i++)
{
    sum += i;
    int k = sum;
    k /= (sum - n);
    TRACE("i=%d\n", i);
}
```

这段代码必须写在支持 MFC 的应用程序中，因为 TRACE 宏是定义在 MFC 中的。运行此程序时不用设置断点，直接按 F5 键调试运行，变量 i 的值就会显示在 Debug 输出窗口中，如图 15.24 所示。

通过图 15.24 就可以看出循环在什么位置出错，并能同时看到相应变量的值，这有助于对程序中的错误进行分析。这种方法不但适用于循环程序，也适用于其他错误的处理。

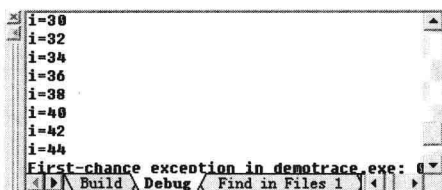


图 15.24 在 Debug 窗口中输出变量的值

## 15.6 小 结

对于程序开发人员来说，程序调试是必不可少的能力之一。因为任何程序在开发完成以后，都需要进行调试，以修改运行中的错误，达到用户的需求标准。所以本章着重对程序错误的常见类型和调试工具进行了讲解，并且对调试的应用进行了举例介绍。

## 15.7 实践与练习

以下程序完成简单交换法排序功能，调试运行程序使其完成预期功能。（实例位置：光盘\TM\sl\15\1）

```
#include<iostream.h>
main()
{
    int a[10],i,j,t,m;
    for(i=0;i<9;i++)
    {
        for(j=i+1;j<10;j++)
            if(a[i]<a[j])
                m=j;
        t=a[i];a[i]=a[m];a[m]=t;
    }
    for(i=0;i<10;i++)
        cout<<a[i]<<" ";
    return 0;
}
```

# 第16章

---

## 文件操作

(  视频讲解：58 分钟 )

文件操作是程序开发中不可缺少的一部分，任何需要存储数据的软件都需要进行文件操作。文件操作包括打开文件、读文件和写文件，掌握读文件和写文件的同时，还要理解文件指针的移动，这能够控制读文件和写文件的位置。

通过阅读本章，您可以：

- » 了解文件流
- » 掌握文件的打开方式
- » 掌握文件的读写操作
- » 掌握文件随机访问

## 16.1 文 件 流

 视频讲解：光盘\TM\lx\16\文件流.exe

### 16.1.1 C++中的流类库

C++语言中为不同类型数据的标准输入和输出定义了专门的类库，类库中主要有 `ios`、`istream`、`ostream`、`iostream`、`ifstream`、`ofstream`、`fstream`、`istrstream`、`ostrstream` 和 `strstream` 等类。`ios` 为根基类，它直接派生 4 个类，即输入流类 `istream`、输出流类 `ostream`、文件流基类 `fstreambase` 和字符串流基类 `strstreambase`。输入文件流类 `ifstream` 同时继承了输入流类和文件流基类，输出文件流类 `ofstream` 同时继承了输出流类和文件流基类，输入字符串流类 `istrstream` 同时继承了输入流类和字符串流基类，输出字符串流类 `ostrstream` 同时继承了输出流类和字符串流基类，输入/输出流类 `iostream` 同时继承了输入流类和输出流类，输入/输出文件流类 `fstream` 同时继承了输入/输出流类和文件流基类，输入/输出字符串流类 `strstream` 同时继承了输入/输出流类和字符串流基类。类库关系如图 16.1 所示。

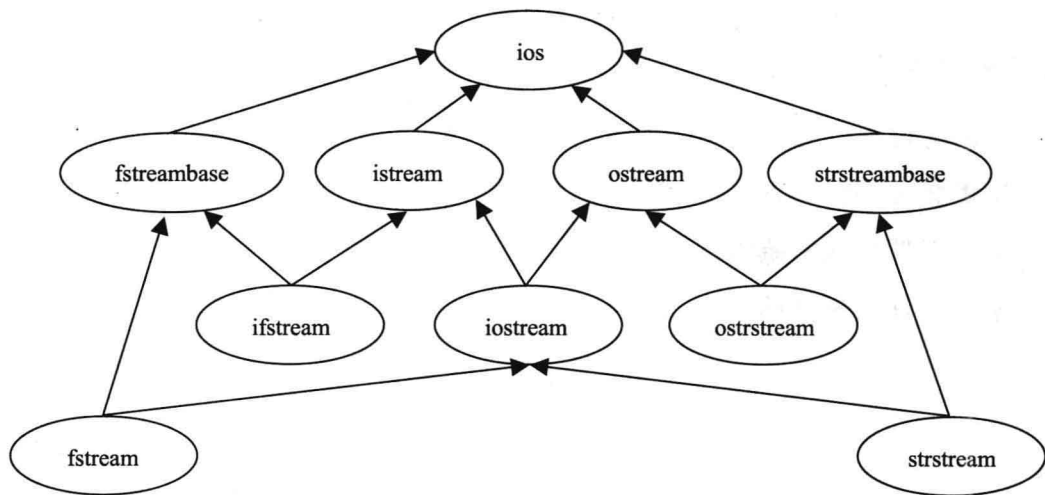


图 16.1 类库关系图

### 16.1.2 类库的使用

C++系统中的 I/O 标准类，都定义在 `iostream.h`、`fstream.h` 和 `strstream.h` 这 3 个头文件中，各头文件包含的类如下：

- ☑ 进行标准 I/O 操作时使用 `iostream.h` 头文件，它包含 `ios`、`iostream`、`istream` 和 `ostream` 等类。
- ☑ 进行文件 I/O 操作时使用 `fstream.h` 头文件，它包含 `fstream`、`ifstream`、`ofstream` 和 `fstreambase`

等类。

- ❑ 进行串 I/O 操作时使用 `strstream.h` 头文件,它包含 `strstream`、`istrstream`、`ostrstream`、`strstreambase` 和 `iostream` 等类。

要进行什么样的操作,只要引入头文件就可以使用类进行操作了。

### 16.1.3 ios 类中的枚举常量

在根基类 `ios` 中定义了用户需要使用的枚举类型,由于它们是在公用成员部分定义的,所以其中的每个枚举类型常量在加上 `ios::` 前缀后都可以被本类成员函数和所有外部函数访问。

在 3 个枚举类型中有一个无名枚举类型,其中定义的每个枚举常量都是用于设置控制输入/输出格式的标志的。该枚举类型定义如下:

```
enum{skipws,left,right,internal,dec,oct,hex,showbase,showpoint,uppercase,showpos,scientific,fixed,unitbuf,stdio};
```

主要枚举常量的含义如下。

- ❑ `skipws`: 利用它设置对应标志后,从流中输入数据时跳过当前位置及后面的所有连续的空白字符,从第一个非空白字符起读数,否则不跳过空白字符。空格、制表符 `\t`、回车符 `\r` 和换行符 `\n` 统称为空白符。默认为设置。
- ❑ `left`: 靠左对齐输出数据。
- ❑ `right`: 靠右对齐输出数据。
- ❑ `internal`: 显示占满整个域宽,用填充字符在符号和数值之间填充。
- ❑ `dec`: 用十进制输出数据。
- ❑ `hex`: 用十六进制输出数据。
- ❑ `showbase`: 在数值前显示基数符,八进制基数符是 `0`,十六进制基数符是 `0x`。
- ❑ `showpoint`: 强制输出的浮点数中带有小数点和小数尾部的无效数字 `0`。
- ❑ `uppercase`: 用大写输出数据。
- ❑ `showpos`: 在数值前显示符号。
- ❑ `scientific`: 用科学记数法显示浮点数。
- ❑ `fixed`: 用固定小数点位数显示浮点数。

### 16.1.4 流的输入/输出

通过前文的学习,相信读者已经对文件流有了一定的了解,现在就通过实例来介绍如何在程序中使用流进行输出。

**【例 16.1】** 字符相加。(实例位置: 光盘\TM\sl\16\1)

```
#include<iostream.h>
#include<strstream.h>
void main()
```



```

{
    char buf[ ]="12345678";
    int i,j;
    istringstream s1(buf);
    s1 >> i;           //将字符串转换为数字
    istringstream s2(buf,3);
    s2 >> j;           //将字符串转换为数字
    cout << i+j << endl; //两个数字相加
}

```

程序运行结果如图 16.2 所示。

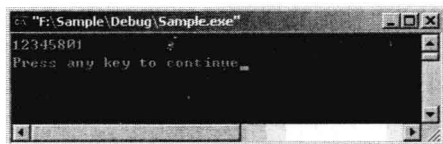


图 16.2 字符相加

## 16.2 文件打开

 视频讲解：光盘\TM\lx\16\文件打开.exe

### 16.2.1 打开方式

只有使用文件流与磁盘上的文件进行连接后才能对磁盘上的文件进行操作，这个连接过程称为打开文件。

打开文件的方式有以下两种。

(1) 在创建文件流时利用构造函数打开文件，即在创建流时加入参数。语法结构如下：

**<文件流类> <文件流对象名>(<文件名>,<打开方式>)**

其中文件流类可以是 `fstream`、`ifstream` 和 `ofstream` 中的一种。文件名指的是磁盘文件的名称，包括磁盘文件的路径名。打开方式在 `ios` 类中定义，有输入方式、输出方式、追加方式等。

- ☑ `ios::in`：用输入方式打开文件，文件只能读取，不能改写。
- ☑ `ios::out`：以输出方式打开文件，文件只能改写，不能读取。
- ☑ `ios::app`：以追加方式打开文件，打开后文件指针在文件尾部，可改写。
- ☑ `ios::ate`：打开已存在的文件，文件指针指向文件尾部，可读可写。
- ☑ `ios::binary`：以二进制方式打开文件。
- ☑ `ios::trunc`：打开文件进行写操作，如果文件已经存在，清除文件中的数据。
- ☑ `ios::nocreate`：打开已经存在的文件，如果文件不存在，打开失败，不创建。
- ☑ `ios::noreplace`：创建新文件，如果文件已经存在，打开失败，不覆盖。

参数可以结合运算符“|”使用，例如：

- ☑ `ios::in|ios::out`：以读写方式打开文件，对文件可读可写。
- ☑ `ios::in|ios::binary`：以二进制方式打开文件，进行读操作。

使用相对路径打开文件 `test.txt` 进行写操作：

```
ofstream outfile("test.txt",ios::out);
```

使用绝对路径打开文件 `test.txt` 进行写操作：

```
ofstream outfile("c:\\test.txt",ios::out);
```



字符“\”表示转义，如果使用“c:\”则必须写成“c:\\”。

(2) 利用 `open` 函数打开磁盘文件。语法结构如下：

```
<文件流对象名>.open(<文件名>,<打开方式>);
```

文件流对象名是一个已经定义了的文件流对象。

```
ifstream infile;
infile.open("test.txt",ios::out);
```

使用两种方式中的任意一种打开文件后，如果打开成功，文件流对象为非 0 值；如果打开失败，则文件流对象为 0。可以使用以下语句检测一个文件是否打开成功：

```
void open(const char * filename,int mode,int prot=filebuf::openprot)
```

`prot` 决定文件的访问方式，取值说明如下：

- ☑ 0 表示为普通文件。
- ☑ 1 表示为只读文件。
- ☑ 2 表示为隐含文件。
- ☑ 4 表示为系统文件。

## 16.2.2 默认打开模式

如果没有指定打开方式参数，编译器会使用默认值。

```
std::ofstream std::ios::out | std::ios::trunk
std::ifstream std::ios::in
std::fstream 无默认值
```

文件打开模式根据用户的需要有不同的组合，下面就对各个模式的效果进行介绍。文件打开模式如表 16.1 所示。

表 16.1 文件打开模式

打 开 方 式	效 果	文 件 存 在	文件不存在
in	为读而打开	必须存在	错误
out	为写而打开	覆盖	创建
out   trunc	为写而打开	覆盖	创建
out   app	为在文件结尾处写而打开	不覆盖	创建
in   out	为输入/输出而打开	必须存在	错误
in   out   trunc	为输入/输出而打开	覆盖	创建
in   out   app	为输入/输出而打开	不覆盖	创建

### 16.2.3 打开文件同时创建文件

通过前文的学习，相信读者已经对文件操作的知识有了一定的了解。为了使读者更好地掌握前面学习的内容，下面通过实例进一步介绍。

**【例 16.2】** 创建文件。（实例位置：光盘\TM\sl\16\2）

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
    ofstream ofile;
    cout << "Create file1" << endl;
    ofile.open("test.txt");
    if(!ofile.fail())
    {
        ofile << "name1" << " ";
        ofile << "sex1" << " ";
        ofile << "age1";
        ofile.close();
        cout << "Create file2" << endl;
        ofile.open("test2.txt");
        if(!ofile.fail())
        {
            ofile << "name2" << " ";
            ofile << "sex2" << " ";
            ofile << "age2";
            ofile.close();
        }
    }
    return 0;
}
```

运行程序，将会创建两个文件，由于 `ofstream` 默认打开方式是 `std::ios::out | std::ios::trunc`，所以当文件夹内没有 `test.txt` 文件和 `test2.txt` 文件时，会创建这两个文件，并向文件写入字符串。向 `test.txt` 文

件写入字符串“name1 sex1 age1”;向 test2.txt 文件写入字符串“name2 sex2 age2”。如果文件夹内有 test.txt 文件和 test2.txt 文件时,程序会覆盖原有文件而重新写入。

## 16.3 文件的读写

 视频讲解: 光盘\TM\16\文件的读写.exe

在对文件进行操作时,必然离不开读写文件。在使用程序查看文件内容时,首先要读取文件,而要修改文件内容时,则需要向文件中写入数据。本节主要介绍通过程序对文件进行读写操作。

### 16.3.1 文件流

(1) 流类型。

流可以分为3类,即输入流、输出流和输入/输出流,相应地必须将流说明为 ifstream、ofstream 和 fstream 类的对象。

```
ifstream ifile;    //声明一个输入流
ofstream ofile;    //声明一个输出流
fstream iofile;    //声明一个输入/输出流
```

说明了流对象之后,可以使用函数 open 打开文件。文件的打开即是在流与文件之间建立一个连接。

(2) 文件流成员函数。

ofstream 和 ifstream 类有很多用于磁盘文件管理的函数。

- ☒ attach: 在一个打开的文件与流之间建立连接。
- ☒ close: 刷新未保存的数据后关闭文件。
- ☒ flush: 刷新流。
- ☒ open: 打开一个文件并把它与流连接。
- ☒ put: 把一个字节写入流中。
- ☒ rdbuf: 返回与流连接的 filebuf 对象。
- ☒ seekp: 设置流文件指针位置。
- ☒ setmode: 设置流为二进制或文本模式。
- ☒ tellp: 获取流文件指针位置。
- ☒ write: 把一组字节写入流中。

(3) fstream 成员函数。

fstream 成员函数如表 16.2 所示。

表 16.2 fstream 成员函数

函 数 名	功 能 描 述
get(c)	从文件读取一个字符
getline(str,n,'n')	从文件读取字符存入字符串 str 中,直到读取 n-1 个字符或遇到'\n'时结束

续表

函 数 名	功 能 描 述
peek()	查找下一个字符，但不从文件中取出
put(c)	将一个字符写入文件
putback(c)	对输入流放回一个字符，但不保存
eof	如果读取超过 eof，返回 true
ignore(n)	跳过 n 个字符，参数为空时，表示跳过下一个字符

**说明**

表中参数 c、str 为 char 型，参数 n 为 int 型。

通过上面的介绍，读者已经对写入流有了一定的了解，下面就通过使用 ifstream 和 ofstream 对象实现读写文件的功能。

**【例 16.3】** 使用 ifstream 和 ofstream 对象实现读写文件的功能。（实例位置：光盘\TM\sl\16\3）

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
    char buf[128];
    ofstream ofile("test.txt");
    for(int i=0;i<5;i++)
    {
        memset(buf,0,128);
        cin >> buf;
        ofile << buf;
    }
    ofile.close();
    ifstream ifile("test.txt");
    while(!ifile.eof())
    {
        char ch;
        ifile.get(ch);
        if(!ifile.eof())
            cout << ch;
    }
    cout << endl;
    ifile.close();
    return 0;
}
```

程序运行结果如图 16.3 所示。

程序首先使用 ofstream 类创建并打开 test.txt 文件，然后需要用户输入 5 次数据，程序把这 5 次输入的数据全部写入 test.txt 文件，接着关闭 ofstream 类打开的文件，再用 ifstream 类打开文件，将文件



中的内容输出。

### 16.3.2 写文本文件

文本文件是程序开发经常用到的文件，使用“记事本”程序就可以打开文本文件。文本文件以.txt 作为扩展名，16.3.1 节已经使用 ifstream 和 ofstream 类创建并写入了文本文件，下面主要应用 fstream 向文本文件写入数据。

【例 16.4】 向文本文件写入数据。（实例位置：光盘\TM\sl\16\4）

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
    fstream file("test.txt",ios::out);
    if(!file.fail())
    {
        cout << "start write " << endl;
        file << "name" << " ";
        file << "sex" << " ";
        file << "age" << endl;
    }
    else
        cout << "can not open" << endl;
    file.close();
    return 0;
}
```

程序通过 fstream 类的构造函数打开文本文件 test.txt，然后向文本文件写入了字符串“name sex age”。

### 16.3.3 读取文本文件

前面介绍了如何写入文件信息，下面通过实例来介绍如何读取文本文件的内容。

【例 16.5】 读取文本文件内容。（实例位置：光盘\TM\sl\16\5）

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
    fstream file("test.txt",ios::in);
    if(!file.fail())
    {
        while(!file.eof())
```

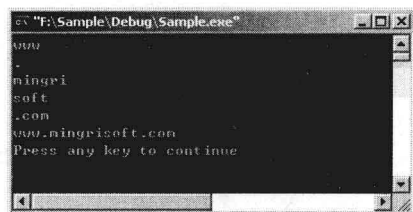


图 16.3 运行结果

```

    {
        char buf[128];
        file.getline(buf,128);
        if(file.tellg()>0)
        {
            cout << buf;
            cout << endl;
        }
    }
else
    cout << "can not open" << endl;;
file.close();
return 0;
}

```

程序首先打开文本文件 test.txt，文件的内容如图 16.4 所示：然后读取文本文件 test.txt 中的内容，并将其输出，运行结果如图 16.5 所示。

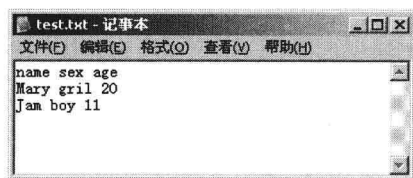


图 16.4 文本文件内容

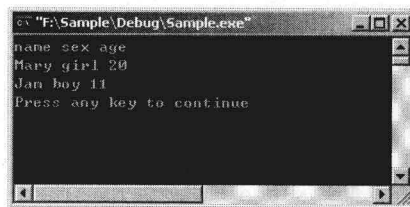


图 16.5 读取文本文件

### 16.3.4 二进制文件的读写

文本文件中的数据都是 ASCII 码，如果要读取图片的内容，就不能使用读取文本文件的方法了。以二进制方式读写文件，需要使用 `ios::binary` 模式，下面通过实例来实现这一功能。

**【例 16.6】** 使用 `read` 读取文件。（实例位置：光盘\TM\sl\16\6）

```

#include<iostream>
#include<fstream>
using namespace std;
int main()
{
    char buf[50];
    fstream file;
    file.open("test.dat",ios::binary|ios::out);
    for(int i=0;i<2;i++)
    {
        memset(buf,0,50);
        cin >> buf;
        file.write(buf,50);
        file << endl;
    }
}

```

```

}
file.close();
file.open("test.dat",ios::binary|ios::in);
while(!file.eof())
{
    memset(buf,0,50);
    file.read(buf,50);
    if(file.tellg()>0)
        cout << buf;
}
cout << endl;
file.close();
return 0;
}

```

程序运行结果如图 16.6 所示。

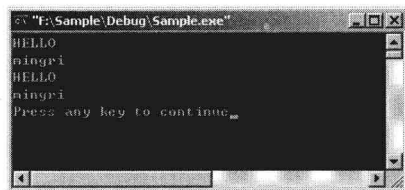


图 16.6 读取文件

程序需要用户输入两次数据，然后通过 `fstream` 以二进制方式写入到文件，再通过 `fstream` 以二进制方式读取出来并输出。对二进制数据读取需要使用 `read` 方法，写入二进制数据需要使用 `write` 方法。

#### 说明

`cout` 遇到结束符 “\0” 就停止输出。在以二进制存储数据的文件中会有很多结束符 “\0”，遇到结束符 “\0” 并不代表数据已经结束。

### 16.3.5 实现文件复制

用户在进行程序开发时，有时需要用到复制等操作，下面就来介绍复制文件的方法。

**【例 16.7】** 文件复制。（实例位置：光盘\TM\sl\16\7）

```

#include<iostream>
#include<fstream>
#include<iomanip>
using namespace std;
int main()
{
    ifstream infile;
    ofstream outfile;
    char name[20];
    char c;

```



```

cout<<"请输入文件: "<<"\n";
cin>>name;
infile.open(name);
if(!infile)
{
    cout<<"文件打开失败! ";
    exit(1);
}
strcat(name,"副本");
cout<<"start copy" << endl;
outfile.open(name);
if(!outfile)
{
    cout<<"无法复制";
    exit(1);
}
while(infile.get(c))
{
    outfile << c;
}
cout<<"start end"<< endl;
infile.close();
outfile.close();
return 0;
}

```

程序需要用户输入一个文件名，然后使用 `infile` 打开文件，接着在文件名后加上“副本”两个字，并用 `outfile` 创建该文件，然后通过一个循环将原文件中的内容复制到目标文件内，完成文件的复制。

## 16.4 文件指针移动操作

 视频讲解：光盘\TM\16\文件指针移动操作.exe

在读写文件的过程中，有时用户不需要对整个文件进行读写，而是对指定位置的一段数据进行读写操作，这时就需要通过移动文件指针来完成。

### 16.4.1 文件错误与状态

在 I/O 流的操作过程中可能出现各种错误，每一个流都有一个状态标志字，以指示是否发生了错误及出现了哪种类型的错误，这种处理技术与格式控制标志字是相同的。`ios` 类定义了以下枚举类型：

```

enum io_state
{
    goodbit=0x00, //不设置任何位，一切正常
    eofbit=0x01,  //输入流已经结束，无字符可读入

```

```
failbit=0x02,    //上次读/写操作失败, 但流仍可使用
badbit=0x04,    //视图进行无效的读/写操作, 流不再可用
bardfail=0x80   //不可恢复的严重错误
};
```

对应于标志字各状态位, ios 类还提供了以下成员函数来检测或设置流的状态。

```
int rdstate();
int eof();
int fail();
int bad();
int good();
int clear(int flag=0);
```

为提高程序的可靠性, 应在程序中检测 I/O 流的操作是否正常。例如用 fstream 默认方式打开文件时, 如果文件不存在, fail 函数就能检测到错误发生, 然后通过 rdstate 方法获得文件状态。

```
fstream file("test.txt");
if(file.fail())
{
    cout << file.rdstate << endl;
}
```

## 16.4.2 文件的追加

在写入文件时, 有时用户不会一次性写入全部数据, 而是在写入一部分数据后再根据条件向文件中追加写入, 例 16.8 将实现这一功能。

**【例 16.8】** 文件追加。(实例位置: 光盘\TM\sl\16\8)

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
    ofstream ofile("test.txt", ios::app);
    if(!ofile.fail())
    {
        cout << "start write " << endl;
        ofile << "Mary ";
        ofile << "girl ";
        ofile << "20 ";
    }
    else
        cout << "can not open";
    return 0;
}
```



程序将字符串“Mary girl 20”追加到文本文件 test.txt 中，文本文件 test.txt 中的内容没有被覆盖。如果 test.txt 文件不存在则创建该文件并写入字符串“Mary girl 20”。

追加可以使用其他方法实现，例如先打开文件然后通过 seekp 方法将文件指针移到末尾，再向文件中写入数据，整个过程和使用参数取值一样。使用 seekp 方法实现追加的代码如下：

```
fstream iofile("test.dat",ios::in|ios::out|ios::binary);
if(iofile)
{
    iofile.seekp(0,ios::end);    //为了写入移动
    iofile << endl;
    iofile << "我是新加入的"
    iofile.seekg(0);            //为了读取移动
    int i=0;
    char data[100];
    while(!iofile.eof && i< sizeof(data))
        iofile.get(data[i++]);
    cout << data;
}
```

程序首先打开 test.dat 文件，查找文件的末尾，并在末尾加入字符串，然后再将文件指针移到文件开始处，输出文件的内容。

### 16.4.3 文件结尾的判断

在操作文件时，经常需要判断文件是否结束，使用 eof 方法可以实现。另外也可以通过其他方法来判断，例如使用流的 get 方法，如果文件指针指向文件末尾，get 方法获取不到数据就返回-1，这也可以作为判断结束的方法。例如：

```
ifstream ifile("test.txt");
if(!ifile)
{
    cerr << "open fail" << endl;
}
char ch;
while(ifile.get(ch))
{
    cout << ch;
}
cout << endl;
ifile.close();
```

程序实现输出 test.txt 文件的内容，同样的功能使用 eof 方法也可以实现。例如：

```
ifstream ifile("test.txt");
if(!ifile.fail())
{
    while(!ifile.eof())
```

```

{
    char ch;
    ifile.get(ch);
    if(!ifile.eof())    //差一个空格
        cout << ch;
}
ifile.close();
}

```

程序仍然是输出 test.txt 文件中的内容，但使用 eof 方法需要多判断一步。

很多地方需要使用 eof 方法来判断文件是否已经读取到末尾，下面通过实例来讲述如何使用 eof 方法判断文件是否结束。

**【例 16.9】** 记录并输出空格位置。(实例位置：光盘\TM\sl\16\9)

```

#include<iostream>
#include<fstream>
using namespace std;
int main()
{
    ifstream ifile("test.txt");
    if(!ifile.fail())
    {
        while(!ifile.eof())
        {
            char ch;
            streampos sp = ifile.tellg();
            ifile.get(ch);
            if(ch == ' ')
            {
                cout << "position:" << sp ;
                cout << "is blank "<< endl;
            }
        }
    }
    return 0;
}

```

程序打开文本文件 test.txt，文件的内容如图 16.7 所示。

程序运行结果如图 16.8 所示。



图 16.7 文本文件内容

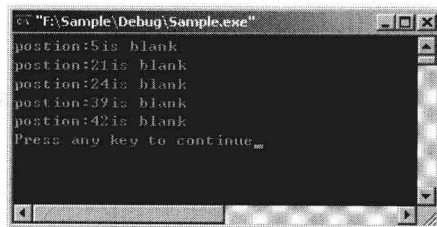


图 16.8 记录并输出空格位置

## 16.4.4 在指定位置读写文件

要实现在指定位置读写文件的功能，首先要了解文件指针是如何移动的，下面将介绍用于设置文件指针位置的函数。

- ☑ seekg: 位移字节数，相对位置用于输入文件中指针的移动。
- ☑ seekp: 位移字节数，相对位置用于输出文件中指针的移动。
- ☑ tellg: 用于查找输入文件中的文件指针位置。
- ☑ tellp: 用于查找输出文件中的文件指针位置。

位移字节数是移动指针的位移量，相对位置是参照位置。取值如下：

- ☑ ios::beg: 文件头部。
- ☑ ios::end: 文件尾部。
- ☑ ios::cur: 文件指针的当前位置。

例如 seekg(0,ios::beg)是将文件指针移动到相对于文件头 0 个偏移量的位置，即指针在文件头。

**【例 16.10】** 输出文件指定位置的内容。（实例位置：光盘\TM\sl\16\10）

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
    ifstream ifile;
    char cFileSelect[20];
    cout << "input filename:";
    cin >> cFileSelect;
    ifile.open(cFileSelect);
    if(!ifile)
    {
        cout << cFileSelect << "can not open" << endl;
        return 0;
    }
    ifile.seekg(0,ios::end);
    int maxpos=ifile.tellg();
    int pos;
    cout << "Position:";
    cin >> pos;
    if(pos > maxpos)
    {
        cout << "is over file lenght" << endl;
    }
    else
    {
        char ch;
        ifile.seekg(pos);
        ifile.get(ch);
    }
}
```



```

        cout << ch << endl;
    }
    ifile.close();
    return 1;
}

```

如果用户输入的文件名是 test.txt, 在 test.txt 文件中含有字符串 “www.mingrisoft.com”, 则程序运行结果如图 16.9 所示。

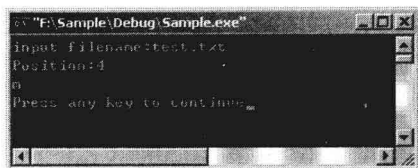


图 16.9 输出文件指定位置的内容

通过 maxpos 可以获得文件长度, 例 16.10 就通过 maxpos 获得了文件长度, 并输出了文件指定位置后的内容。

## 16.5 文件和流的关联和分离

 视频讲解: 光盘\TM\16\文件和流的关联和分离.exe

一个流对象可以在不同时间表示不同文件。在构造一个流对象时, 不用将流和文件绑定。使用流对象的 open 成员函数动态与文件关联, 如果要关联其他文件就调用 close 成员函数关闭当前文件与流的连接, 再通过 open 成员函数建立与其他文件的连接。下面通过实例来实现文件和流的关联和分离功能。

**【例 16.11】** 文件和流的关联和分离。(实例位置: 光盘\TM\16\11)

```

#include<iostream>
#include<fstream>
using namespace std;
int main()
{
    const char* filename="test.txt";
    fstream iofile;
    iofile.open(filename,ios::in);
    if(iofile.fail())
    {
        iofile.clear();
        iofile.open(filename, ios::in| ios::out| ios::trunc);
    }
    else
    {
        iofile.close();
        iofile.open(filename, ios::in| ios::out| ios::ate);
    }
}

```

```

    }
    if(liofile.fail())
    {
        iofile << "我是新加入的";
        iofile.seekg(0);
        while(!iofile.eof())
        {
            char ch;
            iofile.get(ch);
            if(liofile.eof())
                cout << ch;
        }
        cout << endl;
    }
    return 0;
}

```

程序打开文本文件 test.txt, 文件的内容如图 16.10 所示。

程序运行结果如图 16.11 所示。

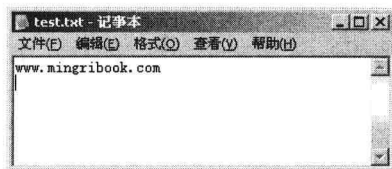


图 16.10 文本文件内容

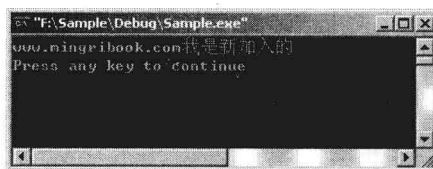


图 16.11 运行结果

程序需要用户输入文件名, 然后使用 `fstream` 的 `open` 函数打开文件, 如果文件不存在就通过在 `open` 函数中指定 `ios::in` | `ios::out` | `ios::trunc` 参数取值创建该文件, 然后向文件中写入数据, 接着将文件指针指向开始处, 最后输出文件内容。程序在第一次调用 `open` 函数打开文件后, 如果文件存在, 则调用 `close` 函数将文件流与文件分离, 接着再调用 `open` 函数建立文件流与文件的关联。

## 16.6 删除文件

 视频讲解: 光盘\TM\16\删除文件.exe

前面介绍了文件的创建以及文件的读写, 本节通过一个具体实例来讲述如何在程序中将一个文件删除。

**【例 16.12】** 删除文件。(实例位置: 光盘\TM\16\12)

```

#include<iostream>
#include<iomanip>
using namespace std;
int main()
{
    char file[50];

```



```
cout <<"Input file name: "<<"\n";
cin >>file;
if(!remove(file))
{
    cout <<"The file:"<<file<<"已删除"<<"\n";
}
else
{
    cout <<"The file:"<<file<<"删除失败"<<"\n";
}
}
```

程序通过 `remove` 函数将用户输入的文件删除。`remove` 函数是系统提供的函数，可以删除指定的磁盘文件。

## 16.7 小 结

本章主要介绍使用文件流进行文件操作，文件在打开时可以控制文件是为写打开还是为读打开，控制打开模式可以控制执行效率，掌握文件的随机读取就可以快速读取想要的数数据，可以实现文件中数据的修改及插入。另外，本章还介绍了使用一个文件流打开多个文件的实现方法，使读者掌握文件流和文件的区别。

## 16.8 实践与练习

1. 将一个文件中的字母复制到指定的文件中。(答案位置：光盘\TM\sl\16\13)
2. 设计一个登录功能，使用文件记录登录用户信息。(答案位置：光盘\TM\sl\16\14)



# 第17章

---

## 网络通信

(  视频讲解：39 分钟 )

随着使用网络的群体日益庞大，对网络软件的需求也越来越大，网络通信已成为程序员必须掌握的技术。网络通信技术中涉及的协议和概念很多，有阻塞函数和非阻塞函数、TCP/IP 协议、客户端和服务端。本章通过具体实例讲述如何建立 socket 通信。

通过阅读本章，您可以：

- » 了解网络模式
- » 掌握基础通信协议
- » 了解如何使用通信函数
- » 掌握通信连接的建立

## 17.1 TCP/IP 协议

 视频讲解：光盘\TM\lx\17\TCP/IP 协议.exe

### 17.1.1 OSI 参考模型

开发式系统互联（Open System Interconnection, OSI），是国际标准化组织（ISO）为了实现计算机网络的标准化而颁布的参考模型。OSI 参考模型采用分层的划分原则，将网络中的数据传输划分为 7 层，每一层使用下层的服务，并向上层提供服务。表 17.1 描述了 OSI 参考模型的结构。

表 17.1 OSI 参考模型

层 次	名 称	功 能 描 述
第 7 层	应用层（Application）	应用层负责网络中应用程序与网络操作系统之间的联系。例如，建立和结束使用者之间的连接，管理建立相互连接使用的应用资源
第 6 层	表示层（Presentation）	表示层用于确定数据交换的格式，它能够解决应用程序之间在数据格式上的差异，并负责设备之间所需要的字符集和数据的转换
第 5 层	会话层（Session）	会话层是用户应用程序与网络层的接口，它能够建立与其他设备的连接，即会话。并且它能够对会话进行有效的管理
第 4 层	传输层（Transport）	传输层提供会话层和网络层之间的传输服务，该服务从会话层获得数据，必要时对数据进行分割，然后传输层将数据传递到网络层，并确保数据能正确无误地传送到网络层
第 3 层	网络层（Network）	网络层能够将传输的数据封包，然后通过路由选择、分段组合等控制，将信息从源设备传送到目标设备
第 2 层	数据链路层（Data Link）	数据链路层主要是修正传输过程中的错误信号，它能够提供可靠的通过物理介质传输数据的方法
第 1 层	物理层（Physical）	利用传输介质为数据链路层提供物理连接，它规范了网络硬件的特性、规格和传输速度

OSI 参考模型的建立不仅创建了通信设备之间的物理通道，还规划了各层之间的功能，为标准化组合和生产厂家定制协议提供了基本原则，它有助于用户了解复杂的协议，例如 TCP/IP、X.25 协议等。用户可以将这些协议与 OSI 参考模型对比，进而了解这些协议的工作原理。

### 17.1.2 TCP/IP 参考模型

TCP/IP（Transmission Control Protocol/Internet Protocol，传输控制协议/网际协议）协议是互联网上最流行的协议，但它并不完全符合 OSI 的 7 层参考模型。TCP/IP 通信协议采用了 4 层的层级结构，每一层都呼叫它的下一层所提供的网络来完成自己的需求。这 4 层分别为：

- ☑ 应用层：应用程序间沟通的层，如简单电子邮件传输（SMTP）、文件传输协议（FTP）、网络远程访问协议（Telnet）等。
- ☑ 传输层：在此层中提供了节点间的数据传送服务，如传输控制协议（TCP）、用户数据包协议（UDP）等，TCP 和 UDP 给数据包加入传输数据并把它传输到下一层中。这一层负责传送数据，并且确定数据已被送达并接收。
- ☑ 互联网络层：负责提供基本的数据封包传送功能，让每一块数据包都能够到达目的主机（但不检查是否被正确接收），如网际协议（IP）。
- ☑ 网络接口层：对实际的网络媒体的管理，定义如何使用实际网络（如 Ethernet、Serial Line 等）来传送数据。

### 17.1.3 IP 地址

IP 被称为网际协议，Internet 上使用的一个关键的底层协议就是 IP 协议。我们利用一个共同遵守的通信协议，使 Internet 成为一个允许连接不同类型的计算机和不同操作系统的网络。要使两台计算机彼此之间进行通信，必须使两台计算机使用同一种“语言”。通信协议正像两台计算机交换信息所使用的共同语言，它规定了通信双方在通信中所应共同遵守的规定。

IP 协议具有能适应各种各样网络硬件的灵活性，对底层网络硬件几乎没有任何要求，任何一个网络只要可以从一个地点向另一个地点传送二进制数据，就可以使用 IP 协议加入 Internet。

如果希望在 Internet 上进行交流和通信，则每台连上 Internet 的计算机都必须遵守 IP 协议。为此，使用 Internet 的每台计算机都必须运行 IP 软件，以便时刻准备发送或接收信息。

IP 地址是由 IP 协议规定的，由 32 位的二进制数表示。最新的 IPv6 协议将 IP 地址升为 128 位，这使得 IP 地址更加广泛，能够很好地解决目前 IP 地址紧缺的情况。但是 IPv6 协议距离实际应用还有一段距离，目前多数操作系统和应用软件都是以 32 位的 IP 地址为基准。

32 位的 IP 地址主要分为两部分，即前缀和后缀。前缀表示计算机所属的物理网络，后缀确定该网络上的唯一一台计算机。在 Internet 上，每一个物理网络都有一个唯一的网络号，根据网络号的不同，可以将 IP 地址分为 5 类，即 A 类、B 类、C 类、D 类和 E 类。其中，A 类、B 类和 C 类属于基本类，D 类用于多播发送，E 类属于保留类。表 17.2 描述了各类 IP 地址的范围。

表 17.2 各类 IP 地址范围

类 型	范 围
A 类	0.0.0.0~127.255.255.255
B 类	128.0.0.0~191.255.255.255
C 类	192.0.0.0~223.255.255.255
D 类	224.0.0.0~239.255.255.255
E 类	240.0.0.0~247.255.255.255

在上述 IP 地址中，有几个 IP 地址是特殊的，有其单独的用途。

#### ☑ 网络地址

在 IP 地址中主机地址为 0 的表示网络地址。例如，128.111.0.0。



☒ 广播地址

在网络号后所有位全是 1 的 IP 地址, 表示广播地址。

☒ 回送地址

127.0.0.1 表示回送地址, 用于测试。

## 17.1.4 数据包格式

TCP/IP 协议的每层都会发送不同的数据包, 常用的有 IP 数据包、TCP 数据包、UDP 数据包和 ICMP 数据包。

### (1) IP 数据包

IP 数据包是在 IP 协议间发送的, 主要在以太网与网际协议模块之间传输, 提供无链接数据包传输。IP 协议不保证数据包的发送, 但最大限度的发送数据。IP 协议结构定义如下:

```
typedef struct HeadIP {
    unsigned char  headerlen:4;    //首部长度, 占 4 位
    unsigned char  version:4;      //版本, 占 4 位
    unsigned char  servertype;      //服务类型, 占 8 位, 即 1 个字节
    unsigned short totallen;        //总长度, 占 16 位
    unsigned short id;              //与 idoff 构成标识, 共占 16 位, 前 3 位是标识, 后 13 位是片偏移
    unsigned short idoff;
    unsigned char  ttl;             //生存时间, 占 8 位
    unsigned char  proto;           //协议, 占 8 位
    unsigned short checksum;        //首部检验和, 占 16 位
    unsigned int   sourceIP;        //源 IP 地址, 占 32 位
    unsigned int   destIP;          //目的 IP 地址, 占 32 位
}HeadIP;
```

理论上, IP 数据包的最大长度是 65535 字节, 这是由 IP 首部 16 位总长度字段所限制的。

### (2) TCP 数据包

传输控制协议 TCP 是一种提供可靠数据传输的通行协议, 它在网际协议模块和 TCP 模块之间传输, TCP 数据包分 TCP 包头和数据两部分。TCP 包头包含了源端口、目的端口、序列号、确认序列号、头部长度、码元比特、窗口、校验和、紧急指针、可选项、填充位和数据区, 在发送数据时, 应用层的数据传输到传输层, 加上 TCP 的 TCP 包头, 数据就构成了包文。报文是网际层 IP 的数据, 如果再加上 IP 首部, 就构成了 IP 数据包。TCP 包头结构定义如下:

```
typedef struct HeadTCP {
    WORD  SourcePort;    //16 位源端口号
    WORD  DePort;        //16 位目的端口
    DWORD SequenceNo;    //32 位序号
    DWORD ConfirmNo;     //32 位确认序号
    BYTE  HeadLen;       //与 Flag 为一个组成部分, 首部长度, 占 4 位, 保留 6 位, 6 位标识, 共 16 位
    BYTE  Flag;
    WORD  WndSize;       //16 位窗口大小
    WORD  CheckSum;      //16 位校验和
```

```
WORD UrgPtr;           //16 位紧急指针
} HeadTCP;
```

TCP 提供了一个完全可靠的、面向连接的、全双工的（包含两个独立且方向相反的连接）流传输服务，允许两个应用程序建立一个连接，并在全双工方向上发送数据，然后终止连接。每一个 TCP 连接可靠地建立并完善地终止，在终止发生前，所有数据都会被可靠地传送。

TCP 比较有名的概念就是 3 次握手，所谓 3 次握手指通信双方彼此交换 3 次信息。3 次握手是在数据包丢失、重复和延迟的情况下，确保通信双方信息交换确定性的充分必要条件。



可靠传输服务软件都是面向数据流的。

### （3）UDP 数据包

用户数据包协议 UDP 是一个面向无连接的协议，采用该协议后，两个应用程序不需要先建立连接，它为应用程序提供一次性的数据传输服务。UDP 协议工作在网际协议模块与 UDP 模块之间，不提供差错恢复，不能提供数据重传，所以使用 UDP 协议的应用程序都比较复杂，例如 DNS（域名解析服务）应用程序。UDP 数据包包头结构如下：

```
typedef struct HeadUDP {
    WORD SourcePort;           //16 位源端口号
    WORD DePort;               //16 位目的端口
    WORD Len;                   //16 为 UDP 长度
    WORD ChkSum;                //16 位 UDP 校验和
} HeadUDP;
```

UDP 数据包分为伪首部和首部两个部分，首部包含原 IP 地址、目标 IP 地址、协议字、UDP 长度、源端口、目的端口、包文长度、校验和、数据区，是为了计算和检验而设置的。伪首部包含 IP 首部一些字段，其目的是让 UDP 两次检查数据是否正确到达目的地。使用 UDP 协议时，协议字为 17，包文长度包括头部和数据区的总长度，最小 8 个字节。校验和是以 16 位为单位，各位求补（首位为符号位）将和相加，然后再求补。现在的大部分系统都默认提供了可读写大于 8192 字节的 UDP 数据包（使用这个默认值是因为 8192 是 NFS 读写用户数据数的默认值）。因为 UDP 协议是无差错控制的，所以发送过程与 IP 协议类似，即 IP 分组，然后用 ARP 协议来解析物理地址，最后发送。

### （4）ICMP 数据包

ICMP 协议被称为网际控制包文协议。作为 IP 协议的附属协议，ICMP 协议用来与其他主机或路由器交换错误包文和其他重要信息，可以将某个设备的故障信息发送到其他设备上。ICMP 数据包包头结构如下：

```
typedef struct HeadICMP {
    BYTE Type;                 //8 位类型
    BYTE Code;                  //8 位代码
    WORD ChkSum;                //16 位校验和
} HeadICMP;
```

## 17.2 套 接 字

 视频讲解：光盘\TM\lx\17\套接字.exe

所谓套接字，实际上是一个指向传输提供者的句柄。在 Winsock 中，就是通过操作该句柄来实现网络通信和管理的。根据性质和作用的不同，套接字可以分为 3 种，即原始套接字、流式套接字和数据包套接字。原始套接字是在 Winsock 2.0 规范中提出的，它能够使程序开发人员对底层的网络传输机制进行控制，在原始套接字下接收的数据中含有 IP 头。流式套接字提供了双向、有序、可靠的数据传输服务，该类型套接字在通信前需要双方建立连接，大家熟悉的 TCP 协议采用的就是流式套接字。与流式套接字对应的是数据包套接字，数据包套接字提供双向的数据流，但是它不能保证数据传输的可靠性、有序性和无重复性，UDP 协议采用的就是数据包套接字。

### 17.2.1 Winsock 套接字

套接字是网络通信的基石，是网络通信的基本构件，最初由加利福尼亚大学 Berkeley 学院为 UNIX 开发的网络通信编程接口。为了在 Windows 操作系统上使用套接字，20 世纪 90 年代初，微软和第三方厂商共同制定了一套标准，即 Windows Socket 规范，简称 Winsock。1993 年 1 月起 Winsock 1.1 成为业界的一项标准，它为通用的 TCP/IP 应用程序提供了超强并灵活的 API，但 Winsock 1.1 把 API 限定在 TCP/IP 的范畴里，它不像 Berkeley 模型一样可以支持多种协议，所以 Winsock 2.0 进行了扩展，开始支持 IPX/SPX 和 DECNet 等协议。Winsock 2.0 允许多种协议栈的并存，可以使应用程序适用于不同的网络名和网络地址。

### 17.2.2 Winsock 的使用

Windows 系统提供的套接字函数通常封装在 Ws2\_32.dll 动态链接库中，其头文件 Winsock2.h 提供了套接字函数的原型，库文件 Ws2\_32.lib 提供了 Ws2\_32.dll 动态链接库的输出节。在使用套接字函数前，用户需要引用 Winsock2.h 头文件，并链接 Ws2\_32.lib 库文件。例如：

```
#include "winsock2.h"           //引用头文件
#pragma comment(lib,"ws2_32.lib") //链接库文件
```

此外，在使用套接字函数前还需要初始化套接字，可以使用 WSAStartup 函数来实现。例如：

```
WSADATA wsd;           //定义 WSADATA 对象
WSAStartup(MAKEWORD(2,2),&wsd); //初始化套接字
```

常用的套接字函数如下：

#### ☒ WSAStartup 函数

该函数用于初始化 Ws2\_32.dll 动态链接库。在使用套接字函数之前，一定要初始化 Ws2\_32.dll 动

态链接库。语法如下：

```
int WSASStartup(WORD wVersionRequested,LPWSADATA lpWSADATA);
```

wVersionRequested: 表示调用者使用的 Winsock 的版本, 高字节记录修订版本, 低字节记录主版本。例如, 如果 Winsock 的版本为 2.1, 则高字节记录 1, 低字节记录 2。

lpWSADATA: 是一个 WSADATA 结构指针, 详细记录了 Windows 套接字的相关信息, 其定义如下:

```
typedef struct WSADATA {
    WORD        wVersion;
    WORD        wHighVersion;
    char        szDescription[WSADESCRIPTION_LEN+1];
    char        szSystemStatus[WSASYS_STATUS_LEN+1];
    unsigned short iMaxSockets;
    unsigned short iMaxUdpDg;
    char FAR *   lpVendorInfo;
} WSADATA, FAR * LPWSADATA;
```

- wVersion: 调用者使用的 Ws2\_32.dll 动态库的版本号。
- wHighVersion: Ws2\_32.dll 支持的最高版本, 通常与 wVersion 相同。
- szDescription: 套接字的描述信息, 通常没有实际意义。
- szSystemStatus: 系统的配置或状态信息, 通常没有实际意义。
- iMaxSockets: 最多可以打开多少个套接字。在套接字版本 2 或以后的版本中, 该成员将被忽略。
- iMaxUdpDg: 数据包的最大长度。在套接字版本 2 或以后的版本中, 该成员将被忽略。
- lpVendorInfo: 套接字的厂商信息。在套接字版本 2 或以后的版本中, 该成员将被忽略。

#### ☒ socket 函数

该函数用于创建一个套接字。语法如下:

```
SOCKET socket(int af,int type, int protocol);
```

- af: 一个地址家族。通常为 AF\_INET。
- type: 套接字类型。如果为 SOCK\_STREAM, 表示创建面向连接的流式套接字; 为 SOCK\_DGRAM, 表示创建面向无连接的数据报套接字; 为 SOCK\_RAW, 表示创建原始套接字。对于这些值, 用户可以在 Winsock2.h 头文件中找到。
- protocol: 套接口所用的协议。如果用户不指定, 可以设置为 0。
- 返回值: 函数返回值是创建的套接字句柄。

#### ☒ bind 函数

该函数用于将套接字绑定到指定的端口和地址上。语法如下:

```
int bind(SOCKET s,const struct sockaddr FAR* name,int namelen);
```

- s: 套接字标识。
- name: 一个 sockaddr 结构指针。该结构中包含了要结合的地址和端口号。
- namelen: 确定 name 缓冲区的长度。

➤ 返回值：如果函数执行成功，返回值为 0，否则返回 `SOCKET_ERROR`。

#### ☑ listen 函数

该函数用于将套接字设置为监听模式。对于流式套接字，必须处于监听模式才能够接收客户端套接字的连接。语法如下：

```
int listen(SOCKET s, int backlog);
```

- s：套接字标识。
- backlog：等待连接的最大队列长度。例如，如果 backlog 被设置为 2，此时有 3 个客户端同时发出连接请求，那么前两个客户端连接会放置在等待队列中，第 3 个客户端会得到错误信息。

#### ☑ accept 函数

该函数用于接受客户端的连接。在流式套接字中，套接字处于监听状态才能接受客户端的连接。语法如下：

```
SOCKET accept(SOCKET s, struct sockaddr FAR* addr, int FAR* addrlen);
```

- s：是一个套接字，它应处于监听状态。
- addr：是一个 `sockaddr_in` 结构指针，包含一组客户端的端口号、IP 地址等信息。
- addrlen：用于接收参数 addr 的长度。
- 返回值：一个新的套接字，它对应于已经接受的客户端连接，对于该客户端的所有后续操作，都应使用这个新的套接字。

#### ☑ closesocket 函数

该函数用于关闭套接字。语法如下：

```
int closesocket (SOCKET s);
```

s 标识一个套接字。如果参数 s 设置为 `SO_DONTLINGER` 选项，则调用该函数后会立即返回，但此时如果有数据尚未传送完毕，会继续传递数据，然后才关闭套接字。

#### ☑ connect 函数

该函数用于发送一个连接请求。语法如下：

```
int connect(SOCKET s,const struct sockaddr FAR* name,int namelen );
```

- s：是一个套接字。
- name：套接字 s 想要连接的主机地址和端口号。
- namelen：name 缓冲区的长度。
- 返回值：如果函数执行成功，返回值为 0，否则返回 `SOCKET_ERROR`。用户可以通过 `WSAGETLASTERROR` 得到其错误描述。

#### ☑ htons 函数

该函数将一个 16 位的无符号短整型数据由主机排列方式转换为网络排列方式。语法如下：

```
u_short htons(u_short hostshort );
```



- **hostshort**: 一个主机排列方式的无符号短整型数据。
- **返回值**: 16 位的网络排列方式数据。

#### ☑ **htonl 函数**

该函数将一个无符号长整型数据由主机排列方式转换为网络排列方式。语法如下:

```
u_long htonl(u_long hostlong);
```

- **hostlong**: 一个主机排列方式的无符号长整型数据。
- **返回值**: 32 位的网络排列方式数据。

#### ☑ **inet\_addr 函数**

该函数将一个由字符串表示的地址转换为 32 位的无符号长整型数据。语法如下:

```
unsigned long inet_addr(const char FAR * cp);
```

- **cp**: 一个 IP 地址的字符串。
- **返回值**: 32 位无符号长整数据。

#### ☑ **recv 函数**

该函数用于从面向连接的套接字中接收数据。语法如下:

```
int recv(SOCKET s,char FAR* buf,int len,int flags);
```

- **s**: 一个套接字。
- **buf**: 接收数据的缓冲区。
- **len**: buf 的长度。
- **flags**: 函数的调用方式。如果为 MSG\_PEEK, 表示查看传来的数据, 在序列前端的数据会被复制一份到返回缓冲区中, 但是这个数据不会从序列中移走; 如果为 MSG\_OOB, 表示用来处理 Out-Of-Band 数据, 也就是外带数据。

#### ☑ **send 函数**

该函数用于在面向连接方式的套接字间发送数据。语法如下:

```
int send(SOCKET s,const char FAR * buf, int len,int flags);
```

- **s**: 一个套接字。
- **buf**: 存放要发送数据的缓冲区。
- **len**: 缓冲区长度。
- **flags**: 函数的调用方式。

#### ☑ **select 函数**

该函数用来检查一个或多个套接字是否处于可读、可写或错误状态。语法如下:

```
int select(int nfds,fd_set FAR * readfds,fd_set FAR * writefds,fd_set FAR * exceptfds, const struct timeval FAR * timeout);
```

- **nfds**: 无实际意义, 只是为了和 UNIX 下的套接字兼容。
- **readfds**: 一组被检查可读的套接字。
- **writefds**: 一组被检查可写的套接字。

- exceptfds: 被检查有错误的套接字。
- timeout: 函数的等待时间。

#### ☑ WSACleanup 函数

该函数用于释放在 Ws2\_32.dll 动态链接库初始化时分配的资源。语法如下:

```
int WSACleanup(void);
```

#### ☑ WSAAsyncSelect 函数

该函数用于将网络中发生的事件关联到窗口的某个消息中。语法如下:

```
int WSAAsyncSelect(SOCKET s, HWND hWnd, unsigned int wMsg, long lEvent);
```

- s: 一个套接字。
- hWnd: 接收消息的窗口句柄。
- wMsg: 窗口接收来自套接字中的消息。
- lEvent: 网络中发生的事件。

#### ☑ ioctlsocket 函数

该函数用于设置套接字的 I/O 模式。语法如下:

```
int ioctlsocket(SOCKET s, long cmd, u_long FAR* argp);
```

- s: 待更改 I/O 模式的套接字。
- cmd: 对套接字的操作命令。如果为 FIONBIO, 当 argp 为 0 时, 表示禁止非阻塞模式, 当 argp 非 0 时, 表示设置非阻塞模式; 如果为 FIONREAD, 表示从套接字中可以读取的数据量; 如果为 SIOCATMARK, 表示所有的外带数据都已被读入。这个命令仅适用于流式套接字, 并且该套接字已被设置为可以在线接收外带数据 (SO\_OOBINLINE)。
- argp: 命令参数。

以下是 Winsock 2.0 新增的函数:

- ☑ WSAAccept: accept 函数扩展版本, 它支持条件接收和套接口分组。
- ☑ WSACloseEvent: 释放一个事件对象。
- ☑ WSAConnect: connect 函数的扩展版本, 它支持连接数据交换和 QoS 规范。
- ☑ WSACreateEvent: 创建一个事件对象。
- ☑ WSADuplicateSocket: 为一个共享套接口创建一个新的套接口描述字。
- ☑ WSAEnumNetworkEvents: 检查是否有网络事件发生。
- ☑ WSAEnumProtocols: 得到每个可以使用的协议信息。
- ☑ WSAEventSelect: 把网络事件和一个事件对象连接。
- ☑ WSAGetOverlappedResu: 得到重叠操作的完成状态。
- ☑ WSAGetQOSByName: 对于一个传输协议服务名字提供相应的 QoS 参数。
- ☑ WSAHtonl: htonl 函数的扩展版本。
- ☑ WSAHtons: htons 函数的扩展版本。
- ☑ WSAIoctl: ioctlsocket 函数的允许重叠操作的版本。
- ☑ WSAJoinLeaf: 在多点对话中计入一个叶节点。

- ☑ WSAntohl: ntohl 函数的扩展版本。
- ☑ WSANTohs: ntohs 函数的扩展版本。
- ☑ WSARecv: recv 函数的扩展版本, 它支持分散/聚焦 I/O 和冲抵套接口操作。
- ☑ WSARecvDisconnect: 终止套接口的接收操作。如果套接口是基于连接的, 得到拆除数据。
- ☑ WSARecvFrom: recvfrom 函数的扩展版本, 它支持分散/聚焦 I/O 和冲抵套接口操作。
- ☑ WSAResetEvnet: 重新初始化一个数据对象。
- ☑ WSASend: send 函数的扩展版本, 它支持分散/聚焦 I/O 和冲抵套接口操作。
- ☑ WSASendDisconnect: 启动一系列拆除套接口连接的操作, 并且可以选择发送拆除数据。
- ☑ WSASendTo: sendto 函数的扩展版本, 它支持分散/聚焦 I/O 和冲抵套接口操作。
- ☑ WSASetEvent: 设置一个数据对象。
- ☑ WSASocket: socket 函数的扩展版本, 它以一个 `PROTOCOL_INFO` 结构作为输入参数并且允许创建重叠套接口, 它还允许创建套接口组。
- ☑ WSAWaitForMultipleEvent: 阻塞多个事件对象。

### 17.2.3 套接字阻塞模式

依据套接字函数执行方式的不同, 可以将套接字分为两类, 即阻塞套接字和非阻塞套接字。在阻塞套接字中, 套接字函数的执行会一直等待, 直到函数调用完成才返回。这主要出现在 I/O 操作过程中, 在 I/O 操作完成之前不会将控制权交给程序。这也意味着在一个线程中同时只能进行一项 I/O 操作, 其后的 I/O 操作必须等待正在执行的 I/O 操作完成后才会执行。在非阻塞套接字中, 套接字函数的调用会立即返回, 将控制权交给程序。默认情况下, 套接字为阻塞套接字。为了将套接字设置为非阻塞套接字, 需要使用 `ioctlsocket` 函数。例如, 下面的代码在创建一个套接字后, 将套接字设置为非阻塞套接字。

```
unsigned long nCmd;
SOCKET clientSock = socket(AF_INET, SOCK_STREAM, 0);           //创建套接字
int nState = ioctlsocket(clientSock, FIONBIO, &nCmd);          //设置非阻塞模式
if(nState != 0)                                                //设置套接字非阻塞模式失败
{
    TRACE("设置套接字非阻塞模式失败");
}
```

将程序设置成非阻塞套接字后, Winsock 通过异步选择函数 `WSAAsyncSelect` 来实现非阻塞通信。方法是由该函数指定某种网络事件 (如有数据到达、可以发送数据、有程序请求连接等), 当被指定的网络事件发生时, 由 Winsock 发送程序事先约定的消息, 程序就可以根据这些消息做相应的处理。

### 17.2.4 字节顺序

有时不同的计算机结构使用不同的字节顺序存储数据, 例如基于 Intel 的计算机存储数据的顺序与 Macintosh (Motorola) 计算机相反。通常, 用户不必为在网络上发送和接收数据的字节顺序转换担心, 但在有些情况下, 必须转换字节顺序。例如, 程序中将指定的整数设置为套接字的端口号, 在绑定端

口号之前，必须将端口号从主机顺序转换为网络顺序。

### 17.2.5 面向连接流

面向连接流主要指通信双方在通信前先建立连接。建立连接的步骤如下：

- ☑ 创建套接字 Socket。
- ☑ 将创建的套接字绑定（bind）到本地的地址和端口上。
- ☑ 服务端设置套接字的状态为监听状态（listen），准备接受客户端的连接请求。
- ☑ 服务端接受请求（accept），同时返回得到一个用于连接的新套接字。
- ☑ 使用这个新套接字进行通信（通信函数使用 send/rcv）。
- ☑ 释放套接字资源（closesocket）。

整个过程分为客户端和服务端，两端连接过程如图 17.1 所示。

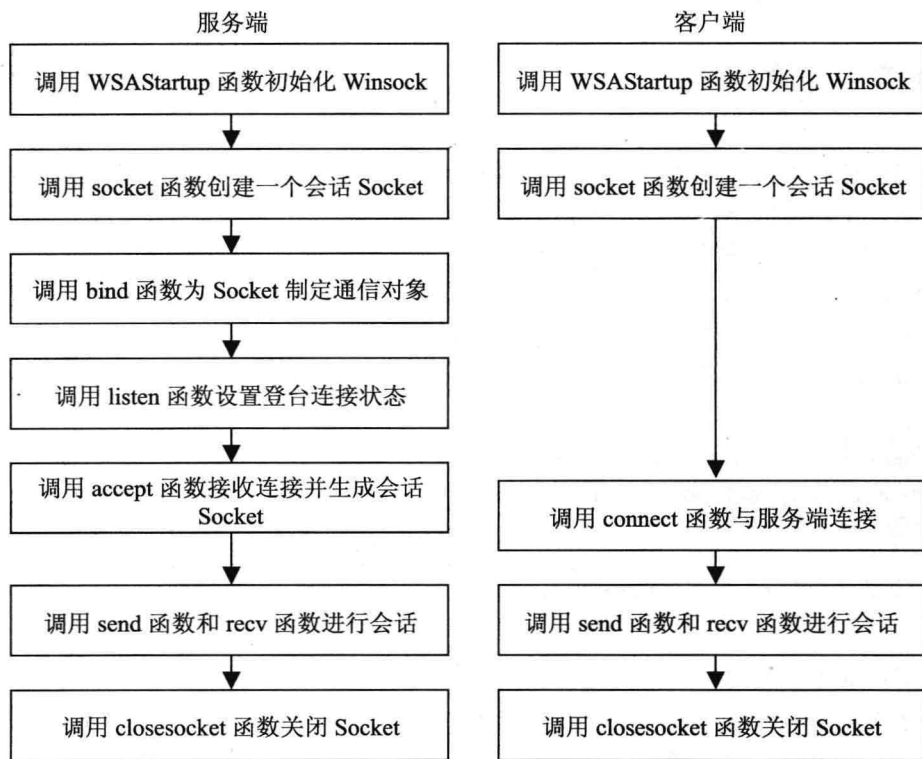


图 17.1 面向连接流

### 17.2.6 面向无连接流

所谓面向无连接流主要指通信双方通信前不需要建立连接，服务端和客户端使用相同的处理过程，如图 17.2 所示。

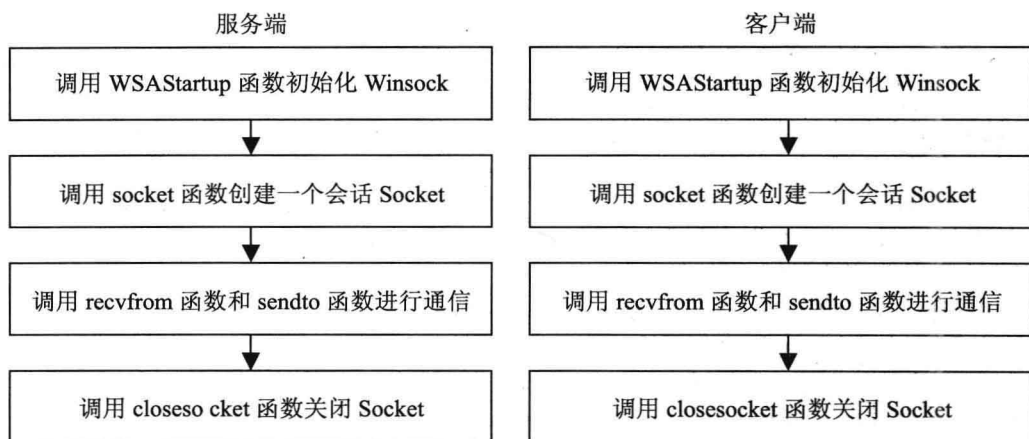


图 17.2 面向无连接流

## 17.3 简单协议通信

视频讲解：光盘\TM\1x\17\简单协议通信.exe

通过前面的学习，读者可能对使用套接字建立通信应用有了一定了解，下面通过具体实例进一步讲述如何使用套接字进行通信。实例主要完成一个简单协议的通信过程，使用的是面向连接方式建立连接，并且是阻塞的方式。实例分为服务端和客户端。

### 17.3.1 服务端

服务端主要使用多线程技术建立连接，也就是说一个服务端可以连接多个客户端，连接客户端的数据可以进行限定，程序中设置最大连接数为 20。当客户端有连接请求发过来时，向客户端发送字符串 THIS IS SERVER，并启动一个线程等待客户端发送消息过来。

如果客户端发送字符 A 过来后，服务端返回 B；发送字符 C 过来后，服务端返回 D；发送 exit 后，服务端关闭线程。

【例 17.1】 服务端。（实例位置：光盘\TM\sl\17\1）

```

#include<iostream.h>
#include<stdlib.h>
#include "winsock2.h"           //引用头文件
#pragma comment(lib,"ws2_32.lib") //引用库文件
//线程实现函数

DWORD WINAPI threadpro(LPVOID pParam)
{
    SOCKET hsock=(SOCKET)pParam;
    char buffer[1024];

```



```

char sendBuffer[1024];
if(hsock!=INVALID_SOCKET)
    cout << "Start Receive" << endl;

while(1) //循环接收发送的内容
{
    int num= recv(hsock,buffer,1024,0); //阻塞函数，等待接收内容
    if(num>=0)
        cout << "Receive form client " << buffer << endl;
    if(!strcmp(buffer,"A"))
    {
        memset(sendBuffer,0,1024);
        strcpy(sendBuffer,"B");
        int ires=send(hsock,sendBuffer,sizeof(sendBuffer),0); //回送信息
        cout << "Send to client" << sendBuffer << endl;
    }
    else if(!strcmp(buffer,"C"))
    {
        memset(sendBuffer,0,1024);
        strcpy(sendBuffer,"D");
        int ires=send(hsock,sendBuffer,sizeof(sendBuffer),0); //回送信息
        cout << "Send to client" << sendBuffer << endl;
    }
    else if(!strcmp(buffer,"exit"))
    {
        cout << "Client Close" << endl;
        cout << "Server Process Close" << endl;
        return 0;
    }
    else
    {
        memset(sendBuffer,0,1024);
        strcpy(sendBuffer,"ERR");
        int ires=send(hsock,sendBuffer,sizeof(sendBuffer),0);
        cout << "Send to client" << sendBuffer << endl;
    }
}
return 0;
}
//主函数
void main()
{
    WSADATA wsd; //定义 WSADATA 对象
    WSAStartup(MAKEWORD(2,2),&wsd);
    SOCKET m_SockServer;
    sockaddr_in serveraddr;
    sockaddr_in serveraddrfrom;
    SOCKET m_Server[20];

    serveraddr.sin_family = AF_INET; //设置服务器地址家族

```

```

serveraddr.sin_port = htons(4600); //设置服务器端口号
serveraddr.sin_addr.S_un.S_addr = inet_addr("127.0.0.1");

m_SockServer = socket(AF_INET, SOCK_STREAM, 0);

int i=bind(m_SockServer, (sockaddr*)&serveraddr, sizeof(serveraddr));
cout << "bind:" << i << endl;

int iMaxConnect=20; //最大连接数
int iConnect=0;
int iLisRet;
char buf[]="THIS IS SERVER\0"; //向客户端发送的内容
char WarnBuf[]="It is voer Max connect\0";
int len=sizeof(sockaddr);
while(1)
{
    iLisRet=listen(m_SockServer, 0); //进行监听
    m_Server[iConnect]=accept(m_SockServer, (sockaddr*)&serveraddr, &len); //同意建立连接

    if(m_Server[iConnect]!=INVALID_SOCKET)
    {
        int ires=send(m_Server[iConnect], buf, sizeof(buf), 0); //发送字符过去
        cout << " accept" << ires << endl; //显示已经建立连接次数
        iConnect++;
        if(iConnect > iMaxConnect)
        {
            int ires=send(m_Server[iConnect], WarnBuf, sizeof(WarnBuf), 0);
        }
        else
        {
            HANDLE m_Handle; //线程句柄
            DWORD nThreadId = 0; //线程 ID
            m_Handle = (HANDLE)::CreateThread(NULL,
            0, threadpro, (LPVOID)m_Server[iConnect], 0, &nThreadId); //启动线程
        }
    }
}
WSACleanup();
}

```

程序中建立连接的 IP 只限制在本机，可以通过修改 `inet_addr("127.0.0.1")` 表达式的值，来设置需要的 IP。

### 17.3.2 客户端

客户端程序主要完成向服务端发送连接请求，然后由用户输入要发送的字符，发送的字符限定在“A”、“C”和“exit”。



## 【例 17.2】 客户端。(实例位置: 光盘\TM\sl\17\2)

```

#include<iostream.h>
#include<stdlib.h>
#include "winsock2.h"
#include<time.h>                                //引用头文件
#pragma comment(lib,"ws2_32.lib")

void main()
{
    WSADATA wsd;                                //定义 WSADATA 对象
    WSStartup(MAKEWORD(2,2),&wsd);
    SOCKET      m_SockClient;
    sockaddr_in clientaddr;

    clientaddr.sin_family = AF_INET;             //设置服务器地址家族
    clientaddr.sin_port = htons(4600);          //设置服务器端口号
    clientaddr.sin_addr.S_un.S_addr = inet_addr("127.0.0.1");
    m_SockClient = socket ( AF_INET,SOCK_STREAM, 0 );
    int i=connect(m_SockClient,(sockaddr*)&clientaddr,sizeof(clientaddr)); //连接超时
    cout << "connect" << i << endl;

    char buffer[1024];
    char inBuf[1024];
    int num;
    num = recv(m_SockClient,buffer,1024,0);      //阻塞
    if(num > 0)
    {
        cout << "Receive form server" << buffer << endl; //欢迎信息
        while(1)
        {
            num=0;
            cin >> inBuf;
            if(!strcmp(inBuf,"exit"))
            {
                send(m_SockClient,inBuf,sizeof(inBuf),0); //发送退出指令
                return;
            }
            send(m_SockClient,inBuf,sizeof(inBuf),0);
            num= recv(m_SockClient,buffer,1024,0); //接收客户端发送过来的数据
            if(num>=0)
                cout << "Receive form server" << buffer << endl;
        }
    }
}

```

### 17.3.3 实例的运行

首先启动服务端，然后启动客户端，在客户端输入字符“A”，然后输入“C”，最后输入“exit”退出客户端，服务端运行如图 17.3 所示。

客户端运行如图 17.4 所示。

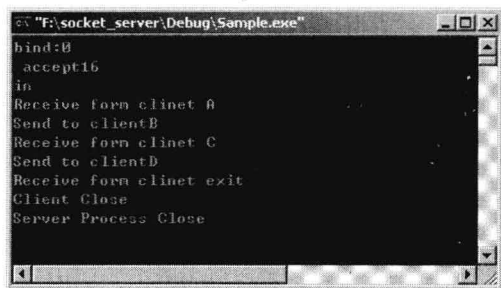


图 17.3 服务端

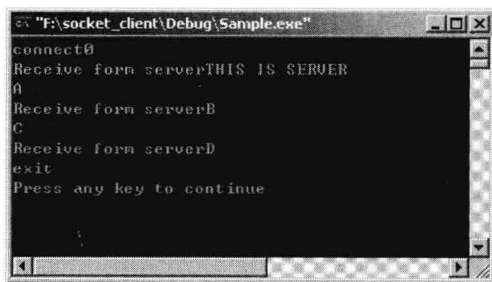


图 17.4 客户端

## 17.4 小 结

本章主要介绍了 TCP/IP 通信协议、OSI 参考模型和 Windows 系统提供的建立套接字通信的函数，可以结合实例了解套接字通信函数的具体使用情况。网络技术是一门学科，在使用网络编程之前，应该掌握基础的网络技术，理解 TCP/IP 协议，为网络编程打好基础。

## 17.5 实践与练习

1. 设计程序，要求当客户端连接到服务端时，服务端会显示连接的提示信息，并反馈信息给客户。  
(答案位置：光盘\TM\sl\17\3)
2. 修改例 17.1 和例 17.2，使其程序是基于 UDP 的网络聊天程序。(答案位置：光盘\TM\sl\17\4)





# 第 4 篇

## 项目实战

### » 第 18 章 图书管理系统

本篇通过一个图书管理系统，运用软件工程的设计思想，讲解如何进行软件项目的开发。书中按照编写需求分析→系统设计→功能设计→创建项目→实现项目模块功能→运行项目的流程进行介绍，带领读者一步一步亲身体验开发项目的全过程。



# 第18章

---

## 图书管理系统

(  视频讲解：42 分钟 )

通过前面章节的学习，读者已对 C++ 的语法有所了解，把所学的知识都应用到实际开发中是本章所要完成的任务。随着信息的爆炸式发展，好的图书作品不断出现，这就要求用一个系统来管理日益丰富的图书，以便对图书进行快速检索。本章主要实现一个简单的图书管理系统。

通过阅读本章，您可以：

- » 了解软件整体设计
- » 掌握类的实际应用
- » 掌握分页数据浏览
- » 掌握文件存储数据

## 18.1 系统设计

 视频讲解：光盘\TM\lx\18\系统设计.exe

### 18.1.1 需求分析

明日图书城是一家书店，为方便消费者对图书的检索，特开发此系统。系统应符合以下几个方面的要求：

- ☒ 可以真正地实现对图书信息的管理。
- ☒ 系统的功能操作要方便、易懂，不要有多余或复杂的操作。
- ☒ 信息输出格式便于浏览。

### 18.1.2 系统目标

对于图书管理这样的管理系统，必须要满足使用方便、操作灵活和安全性好等设计需求。本系统在设计时应该满足以下几个目标：

- ☒ 图书的录入使用交互方式。
- ☒ 能够浏览文件中存储的全部图书。
- ☒ 图书信息在屏幕上的输出要有固定格式。
- ☒ 系统最大限度地实现易维护性和易操作性。
- ☒ 系统运行稳定、安全可靠。
- ☒ 开发的操作系统为 Windows XP、Windows 2000 或 Windows 2003，数据库采用二进制文件，开发工具为 Visual C++ 6.0。

### 18.1.3 系统功能结构

系统功能结构如图 18.1 所示。

- ☒ 图书录入模块：该模块主要是提供给图书管理者使用。图书管理者应用该模块将图书信息录入到系统，系统将图书信息保存到文件中。
- ☒ 浏览全部图书记录模块：该模块提供给读者和图书管理者使用。图书管理者可以通过该模块查看图书是否存在，以及获取图书的编号，方便日后的删除；读者可以根据该模块了解到图书的价格和作者等信息，从而决定是否购买。
- ☒ 删除图书模块：该模块主要是提供给图书管理者使用。图书管理者可以通过该模块删除书店中已经销售完的图书信息。

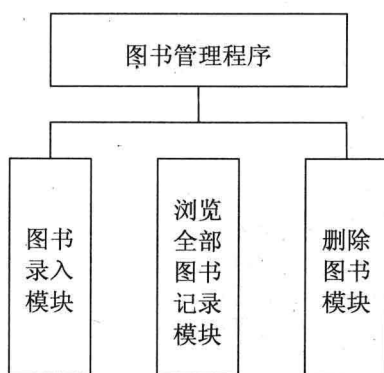


图 18.1 系统功能结构

## 18.2 图 书 类

 视频讲解：光盘\TM\lx\18\图书类.exe

图书管理系统需要创建 CBook 类，通过 CBook 类实现图书记录的写入和删除，还可以通过 CBook 类查看每条图书的信息。CBook 类中包含 m\_cName、m\_cIsbn、m\_cPrice 和 m\_cAuthor 4 个成员变量，分别代表图书的名称、ISBN 编号、价格和作者。在类设计时，可以将成员变量看作属性，这样类中还需要有设置属性和获取属性的成员函数，设置属性的函数以 Set 开头，获取属性的函数以 Get 开头。图书类设计图如图 18.2 所示。

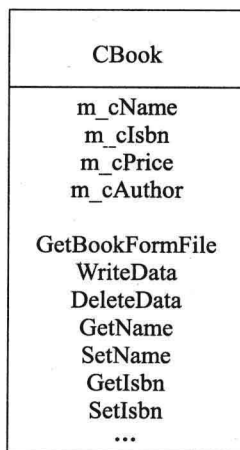


图 18.2 图书类设计图

图书类定义在头文件 Book.h 中，代码如下：

```
#define NUM1 128
#define NUM2 50
class CBook
```



```

{
public:
    CBook(){ }
    CBook(char* cName,char* clsbn,char* cPrice,char* cAuthor);
    ~CBook(){ }
public:
    char* GetName();           //获取图书名称
    void SetName(char* cName); //设置图书名称
    char* GetIsbn();           //获取图书 ISBN 编号
    void SetIsbn(char* clsbn);  //设置图书 ISBN 编号
    char* GetPrice();           //获取图书价格
    void SetPrice(char* cPrice); //设置图书价格
    char* GetAuthor();          //获取图书作者
    void SetAuthor(char* cAuthor); //设置图书作者
    void WriteData();
    void DeleteData(int iCount);
    void GetBookFromFile(int iCount);
protected:
    char m_cName[NUM1];
    char m_clsbm[NUM1];
    char m_cPrice[NUM2];
    char m_cAuthor[NUM2];
};

```

图书类成员函数的实现都存储在实现文件 Book.cpp 内。

```

#include "Book.h"
#include<string>
#include<fstream>
#include<iostream>
#include<iomanip>
using namespace std;
CBook::CBook(char* cName,char* clsbn,char* cPrice,char* cAuthor)
{
    strncpy(m_cName,cName,NUM1);
    strncpy(m_clsbm,clsbn,NUM1);
    strncpy(m_cPrice,cPrice,NUM2);
    strncpy(m_cAuthor,cAuthor,NUM2);
}
char* CBook::GetName()
{
    return m_cName;
}
void CBook::SetName(char* cName)
{
    strncpy(m_cName,cName,NUM1);
}
char* CBook::GetIsbn()
{

```

```

        return m_clsbn;
    }
    void CBook::SetIsbn(char* clsbn)
    {
        strncpy(m_clsbn,clsbn,NUM1);
    }
    char* CBook::GetPrice()
    {
        return m_cPrice;
    }
    void CBook::SetPrice(char* cPrice)
    {
        strncpy(m_cPrice,cPrice,NUM2);
    }
    char* CBook::GetAuthor()
    {
        return m_cAuthor;
    }
    void CBook::SetAuthor(char* cAuthor)
    {
        strncpy(m_cAuthor,cAuthor,NUM2);
    }
}

```

函数 WriteData、GetBookFromFile、DeleteData 是类对象读写文件的函数，相当于操作数据库的接口。

(1) 成员函数 WriteData 主要实现将图书对象写入到文件中。

```

void CBook::WriteData()
{
    ofstream ofile;
    ofile.open("book.dat",ios::binary|ios::app);
    try
    {
        ofile.write(m_cName,NUM1);
        ofile.write(m_clsbn,NUM1);
        ofile.write(m_cPrice,NUM2);
        ofile.write(m_cAuthor,NUM2);
    }
    catch(...)
    {
        throw "file error occurred";
        ofile.close();
    }
    ofile.close();
}

```

(2) 成员函数 GetBookFromFile 能够实现从文件中读取数据来构建对象。

```

void CBook::GetBookFromFile(int iCount)
{
    char cName[NUM1];

```



```

char clsbn[NUM1];
char cPrice[NUM2];
char cAuthor[NUM2];
ifstream ifile;
ifile.open("book.dat",ios::binary);
try
{
    ifile.seekg(iCount*(NUM1+NUM1+NUM2+NUM2),ios::beg);
    ifile.read(cName,NUM1);
    if(ifile.tellg()>0)
        strncpy(m_cName,cName,NUM1);
    ifile.read(clsbn,NUM1);
    if(ifile.tellg()>0)
        strncpy(m_clsbn,clsbn,NUM1);
    ifile.read(cPrice,NUM2);
    if(ifile.tellg()>0)
        strncpy(m_clsbn,clsbn,NUM2);
    ifile.read(cAuthor,NUM2);
    if(ifile.tellg()>0)
        strncpy(m_cAuthor,cAuthor,NUM2);
}
catch(...)
{
    throw "file error occurred";
    ifile.close();
}
ifile.close();
}

```

(3) 成员函数 DeleteData 负责将图书信息从文件中删除。

```

void CBook::DeleteData(int iCount)
{
    long respos;
    int iDataCount=0;
    fstream file;
    fstream tmpfile;
    ofstream ofile;
    char cTempBuf[NUM1+NUM1+NUM2+NUM2];
    file.open("book.dat",ios::binary|ios::in|ios::out);
    tmpfile.open("temp.dat",ios::binary|ios::in|ios::out|ios::trunc);
    file.seekg(0,ios::end);
    respos=file.tellg();
    iDataCount=respos/(NUM1+NUM1+NUM2+NUM2);
    if(iCount < 0 && iCount > iDataCount)
    {
        throw "Input number error";
    }
    else
    {

```

```

file.seekg((iCount)*(NUM1+NUM1+NUM2+NUM2),ios::beg);
for(int j=0;j<(iDataCount-iCount);j++)
{
    memset(cTempBuf,0,NUM1+NUM1+NUM2+NUM2);
    file.read(cTempBuf,NUM1+NUM1+NUM2+NUM2);
    tmpfile.write(cTempBuf,NUM1+NUM1+NUM2+NUM2);
}
file.close();
tmpfile.seekg(0,ios::beg);
ofile.open("book.dat");
ofile.seekp((iCount-1)*(NUM1+NUM1+NUM2+NUM2),ios::beg);
for(int i=0;i<(iDataCount-iCount);i++)
{
    memset(cTempBuf,0,NUM1+NUM1+NUM2+NUM2);
    tmpfile.read(cTempBuf,NUM1+NUM1+NUM2+NUM2);
    ofile.write(cTempBuf,NUM1+NUM1+NUM2+NUM2);
}
}
tmpfile.close();
ofile.close();
remove("temp.dat");
}

```

## 18.3 主 程 序

 视频讲解：光盘\TM\lx\18\主程序.exe

程序主界面如图 18.3 所示。

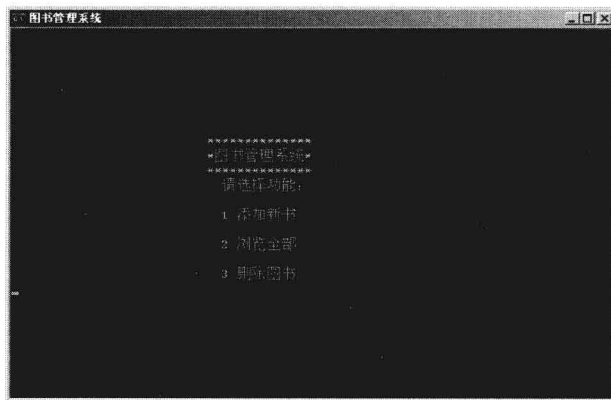


图 18.3 程序主界面

要实现系统，需要对引用库函数添加头文件引用。头文件引用和宏定义的代码如下：

```

#include<iostream>
#include<iomanip>

```

```
#include<stdlib.h>
#include<conio.h>
#include<string.h>
#include<fstream>
#include "Book.h"

#define CMD_COLS 80
#define CMD_LINES 25
using namespace std;
```

除主函数外，系统自定义了许多函数，主要函数及功能如下。

- ☑ void SetScreenGrid: 设置屏幕显示行数和列数。
- ☑ void ClearScreen: 清除屏幕信息。
- ☑ void SetSysCaption(const char \*pText): 设置窗体标题栏。
- ☑ void ShowWelcome: 显示欢迎信息。
- ☑ void ShowRootMenu: 显示开始菜单。
- ☑ void WaitView(int iCurPage): 浏览数据时等待用户操作。
- ☑ void WaitUser: 等待用户操作。
- ☑ void GuideInput: 使用向导添加图书信息。
- ☑ int GetSelect: 获得用户菜单选择。
- ☑ long GetFileLength(ifstream & ifs): 获取文件长度。
- ☑ void ViewData(int iSelPage): 浏览所有图书记录。
- ☑ void DeleteBookFromFile: 在文件中产生图书信息。
- ☑ void mainloop: 主循环。

(1) 在控制台中输入 mode 命令可以设置控制显示信息的行数、列数和背景颜色等信息。SetScreenGrid 函数主要通过 system 函数来执行 mode 命令，CMD\_COLS 和 CMD\_LINES 是宏定义中的值。

```
void SetScreenGrid()
{
    char sysSetBuf[80];
    sprintf(sysSetBuf,"mode con cols=%d lines=%d",CMD_COLS,CMD_LINES);
    system(sysSetBuf);
}
```

(2) SetSysCaption 函数主要完成在控制台的标题栏上显示 Sample 信息。控制台的标题栏信息可以使用 title 命令来设置，函数中使用 system 函数来执行 title 命令。

```
void SetSysCaption()
{
    system("title Sample");
}
```

(3) 函数 ClearScreen 主要通过 system 函数来执行 cls 命令，完成控制台屏幕信息的清除。

```
void ClearScreen()
{
```



```

system("cls");
}

```

(4) SetSysCaption 函数共有两个版本, 这是 SetSysCaption 函数的另一个版本, 主要实现在控制台的标题栏上显示指定字符。

```

void SetSysCaption( const char *pText)
{
    char sysSetBuf[80];
    sprintf(sysSetBuf,"title %s",pText);
    system(sysSetBuf);
}

```

(5) ShowWelcome 函数在屏幕上显示“图书管理系统”字样的欢迎信息, “图书管理系统”字样应尽量显示在屏幕的中央位置。

```

void ShowWelcome()
{
    for(int i=0;i<7;i++)
    {
        cout << endl;
    }
    cout << setw(40);
    cout << "*****" << endl;
    cout << setw(40);
    cout << "图书管理系统" << endl;
    cout << setw(40);
    cout << "*****" << endl;
}

```

(6) ShowRootMenu 函数主要显示系统的主菜单, 系统中有 3 个菜单选项, 分别是“添加新书”、“浏览全部”、“删除图书”。3 个菜单选项是进入系统 3 个模块的入口。

```

void ShowRootMenu()
{
    cout << setw(40);
    cout << "请选择功能" << endl;
    cout << endl;
    cout << setw(38);
    cout << "1 添加新书" << endl;
    cout << endl;
    cout << setw(38);
    cout << "2 浏览全部" << endl;
    cout << endl;
    cout << setw(38);
    cout << "3 删除图书" << endl;
}

```

(7) WaitUser 函数主要负责当程序完成某一模块的执行后, 等待用户进行处理。用户可以选择返回主菜单, 也可以直接退出系统。

```

void WaitUser()
{
    int iInputPage=0;
    cout << "enter 返回主菜单 q 退出" << endl;
    char buf[256];
    gets(buf);
    if(buf[0]=='q')
        system("exit");
}

```

(8) main 函数是程序的入口，主要调用了 SetScreenGrid、SetSysCaption 和 mainloop 3 个函数，其中 mainloop 函数是主函数，负责模块执行的调度，代码如下：

```

void mainloop()
{
    ShowWelcome();
    while(1)
    {
        ClearScreen();
        ShowWelcome();
        ShowRootMenu();
        switch(GetSelect())
        {
            case 1:
                ClearScreen();
                GuideInput();
                break;
            case 2:
                ClearScreen();
                ViewData();
                break;
            case 3:
                ClearScreen();
                DeleteBookFromFile();
                break;
        }
    }
}

```

(9) 函数 GetSelect 主要负责获取用户菜单的选择。

```

int GetSelect()
{
    char buf[256];
    gets(buf);
    return atoi(buf);
}

```

其他函数都应用在图书录入模块、浏览全部图书记录模块和删除图书模块中，这将在具体模块中讲解。

## 18.4 添加图书

 视频讲解：光盘\TM\lx\18\添加图书.exe

添加图书需要调用函数 GuideInput 来完成，需要用户分别输入书名、ISBN 编号、价格和作者，然后利用 CBook 类构建一个 CBook 对象，通过 CBook 对象的成员函数 WriteData 将图书信息写入文件。

```
void GuideInput()
{
    char inName[NUM1];
    char inIsdn[NUM1];
    char inPrice[NUM2];
    char inAuthor[NUM2];

    cout << "输入书名" << endl;
    cin >> inName;
    cout << "输入 ISBN" << endl;
    cin >> inIsdn;
    cout << "输入价格" << endl;
    cin >> inPrice;
    cout << "输入作者" << endl;
    cin >> inAuthor;
    CBook book(inName,inIsdn,inPrice,inAuthor);
    book.WriteData();
    cout << "Write Finish" << endl;
    WaitUser();
}
```

程序运行结果如图 18.4 所示。



图 18.4 添加图书

## 18.5 显示图书信息

 视频讲解：光盘\TM\lx\18\显示图书信息.exe

显示图书信息主要通过函数 ViewData 来完成。函数 ViewData 按页数显示图书记录，每页可以显



示 20 条记录。在函数 ViewData 中直接使用文件流类打开存储图书信息的文件 book.dat，然后根据页序号读取文件内容，因为每条图书记录的长度相同，这样就很容易计算出每条记录在文件中的位置，然后将文件指针移动到每页第一条图书记录处，顺序地从文件中读取 20 条记录，并将信息显示在屏幕上。

```
void ViewData(int iSelPage = 1)
{
    int iPage=0;
    int iCurPage=0;
    int iDataCount=0;
    char inName[NUM1];    //存储图书名称的变量
    char inIsbn[NUM1];    //存储图书 ISBN 编号的变量
    char price[NUM2];     //存储图书价格的变量
    char inAuthor[NUM2];  //存储图书作者的变量
    bool bIndex=false;
    int iFileLength;
    iCurPage=iSelPage;
    ifstream ifile;
    ifile.open("book.dat",ios::binary);//
    iFileLength=GetFileLength(ifile);
    iDataCount=iFileLength/(NUM1+NUM1+NUM2+NUM2)    //根据文件长度，计算文件中总的记录数
    if(iDataCount>=1)
        bIndex=true;
    iPage=iDataCount / 20+1;
    ClearScreen();    //清除屏幕信息
    cout << " 共有记录" << iDataCount <<" ";
    cout << " 共有页数" << iPage << " ";
    cout << " 当前页数" << iCurPage << " ";
    cout << " n 显示下一页 m 返回" << endl;
    cout << setw(5)<<"Index" ;
    cout << setw(22) << "Name" << setw(22) << "Isbn" ;
    cout << setw(15) << "Price" << setw(15) << "Author";
    cout << endl;
    try
    {
        //根据图书记录编号查找在文件中的位置
        ifile.seekg((iCurPage-1)*20*(NUM1+NUM1+NUM2+NUM2),ios::beg);
        if(!ifile.fail())
        {
            for(int i=1;i<21;i++)
            {
                memset(inName,0,128);    //将变量清零
                memset(inIsbn,0,128);
                memset(price,0,50);
                memset(inAuthor,0,50);
                if(bIndex)
                    cout <<setw(3)<< ((iCurPage-1)*20+i);
                ifile.read(inName,NUM1);    //读取图书名称
                cout <<setw(24)<< inName;
                ifile.read(inIsbn,NUM1);    //读取图书 ISBN 编号
```

```

        cout <<setw(24)<< inIsbn;
        ifile.read(price,NUM2);           //读取图书价格
        cout <<setw(12)<< price;
        ifile.read(inAuthor,NUM2);       //读取图书作者
        cout <<setw(12)<< inAuthor;
        cout << endl;
        if(ifile.tellg()<0)
            bIndex=false;
        else
            bIndex=true;
    }
}
catch(...)
{
    cout << "throw file exception" << endl;
    throw "file error occurred";         //抛出异常
    ifile.close();                       //异常后关闭文件流
}
if(iCurPage<iPage)
{
    iCurPage=iCurPage+1;
    WaitView(iCurPage);                 //等待用户处理
}
else
{
    WaitView(iCurPage);                 //等待用户处理
}
ifile.close();
}

```

函数 `GetFileLength` 用来获取文件的长度。函数需要指定一个文件流对象，然后根据文件流的 `tellg` 函数计算出文件流绑定的文件长度。计算过程是先通过 `tellg` 获取文件指针的位置，然后通过 `seekg` 函数将文件指针移到文件末尾，再通过 `tellg` 获取文件指针的位置，此时的文件位置就是文件的长度，最后再通过 `seekg` 函数将文件指针恢复到原来位置。

```

long GetFileLength(ifstream & ifs)
{
    long tmppos;
    long respos;
    tmppos=ifs.tellg();
    ifs.seekg(0,ios::end);
    respos=ifs.tellg();
    ifs.seekg(tmppos,ios::beg);
    return respos;
}

```

程序运行结果如图 18.5 所示。



Index	Name	ISBN	Price	Author
1	C++技术大全	978-8-122-1111-4	90	陈晨
2	MyEclipse入门	123-7-121-8776-0	43	李可
3	Java入门	234-9-443-87467-0	78	白雪
4	C实例大全	445-9-34-23464-2	67	高云
5	如何学好C#	345-5-234-345754-3	90	林昊
6				

图 18.5 显示图书信息

## 18.6 删除图书

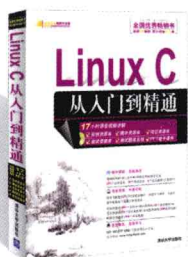
视频讲解：光盘\TM\1x\18\删除图书.exe

删除图书主要通过调用函数 `DeleteBookFromFile` 来完成，而在函数 `DeleteBookFromFile` 中则主要调用 `CBook` 类的 `DeleteData` 成员函数来完成。`DeleteData` 成员函数需要设置删除图书在文件中的顺序编号，在浏览图书时可以看到此编号。`DeleteBookFromFile` 函数的实现代码如下：

```
void DeleteBookFromFile()
{
    int iDelCount;
    cout << "Input delete index" << endl;
    cin >> iDelCount;
    CBook tmpbook;
    tmpbook.DeleteData(iDelCount);
    cout << "Delete Finish" << endl;
    WaitUser();
}
```

## 18.7 小 结

至此，一个简单的图书管理系统就完成了，本系统的开发过程符合软件工程方面的要求，但由于篇幅的关系，系统设计得比较小，没有应用模型，读者可以使用一些模型技术来不断完善该系统。



ISBN 978-7-302-28485-7



ISBN 978-7-302-28486-4



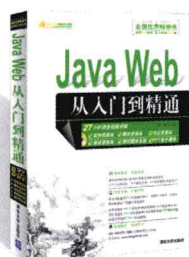
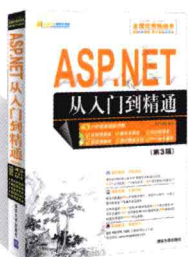
ISBN 978-7-302-28750-6



ISBN 978-7-302-28751-3



ISBN 978-7-302-28752-0



ISBN 978-7-302-28753-7



ISBN 978-7-302-28754-4



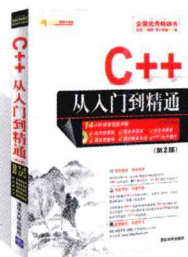
ISBN 978-7-302-28755-1



ISBN 978-7-302-28756-8



ISBN 978-7-302-28757-5



ISBN 978-7-302-28758-2



ISBN 978-7-302-28847-3



ISBN 978-7-302-28848-0



ISBN 978-7-302-28853-4



ISBN 978-7-302-28932-6



清华大学出版社数字出版网站

**WQBook** 书文局泉  
www.wqbook.com

ISBN 978-7-302-28847-3



9 787302 288473 >

定价：59.80元

封面  
书名  
版权  
前言  
目录

## 第1篇 基础知识

### 第1章 绪论

- 1.1 C++历史背景
  - 1.1.1 20世纪最伟大的发明
  - 1.1.2 C++发展历程
  - 1.1.3 C++中的杰出人物
- 1.2常用开发环境
  - 1.2.1 Visual C++ 6.0
  - 1.2.2 Visual C++ 2008
  - 1.2.3 GCC/G++
  - 1.2.4 Dev-C
  - 1.2.5 Eclipse
- 1.3认知C++程序代码
- 1.4 C++工程项目文件
- 1.5使用VC创建程序
- 1.6编译与连接过程
- 1.7 C++的特点
- 1.8小结

### 第2章 数据类型

- 2.1第一个C++程序
  - 2.1.1 #include指令
  - 2.1.2注释
  - 2.1.3 main函数
  - 2.1.4函数体
  - 2.1.5函数返回值
- 2.2数据类型
- 2.3常量及符号
  - 2.3.1整型常量
  - 2.3.2实型常量
  - 2.3.3字符常量
  - 2.3.4字符串常量
  - 2.3.5其他常量
- 2.4变量
  - 2.4.1标识符
  - 2.4.2变量与变量说明
  - 2.4.3整型变量
  - 2.4.4实型变量

- 2.4.5变量赋值
  - 2.4.6变量赋初值
  - 2.4.7字符变量
- 2.5数据输入与输出
  - 2.5.1控制台屏幕
  - 2.5.2 C++语言中的流
  - 2.5.3流操作的控制
- 2.6小结
- 2.7实践与练习
- 第3章 表达式与语句
  - 3.1运算符
    - 3.1.1算术运算符
    - 3.1.2关系运算符
    - 3.1.3逻辑运算符
    - 3.1.4赋值运算符
    - 3.1.5位运算
    - 3.1.6移位运算符
    - 3.1.7 sizeof运算符
    - 3.1.8条件运算符
    - 3.1.9逗号运算符
  - 3.2结合性和优先级
  - 3.3表达式
    - 3.3.1算术表达式
    - 3.3.2关系表达式
    - 3.3.3条件表达式
    - 3.3.4赋值表达式
    - 3.3.5逻辑表达式
    - 3.3.6逗号表达式
    - 3.3.7表达式中的类型转换
  - 3.4语句
  - 3.5小结
  - 3.6实践与练习
- 第4章 条件判断语句
  - 4.1决策分支
  - 4.2判断语句
    - 4.2.1第一种形式的判断语句
    - 4.2.2第二种形式的判断语句
    - 4.2.3第三种形式的判断语句
  - 4.3使用条件运算符进行判断
  - 4.4 switch语句
  - 4.5判断语句的嵌套
  - 4.6小结

#### 4.7实践与练习

### 第5章 循环语句

#### 5.1 while循环

#### 5.2 do...while循环

#### 5.3 while与do...while比较

#### 5.4 for循环语句

#### 5.5循环控制

##### 5.5.1控制循环的变量

##### 5.5.2 break语句

##### 5.5.3 continue语句

##### 5.5.4 goto语句

#### 5.6循环嵌套

#### 5.7循环应用实例

##### 5.7.1阿姆斯壮数

##### 5.7.2巴斯卡三角形

##### 5.7.3对输入的分数进行排名

#### 5.8小结

#### 5.9实践与练习

### 第6章 函数

#### 6.1函数概述

##### 6.1.1函数的定义

##### 6.1.2函数的声明

#### 6.2函数参数及返回值

##### 6.2.1返回值

##### 6.2.2空函数

##### 6.2.3形参与实参

##### 6.2.4默认参数

##### 6.2.5可变参数

#### 6.3函数调用

##### 6.3.1传值调用

##### 6.3.2嵌套调用

##### 6.3.3递归调用

#### 6.4变量作用域

#### 6.5重载函数

#### 6.6内联函数

#### 6.7变量的存储类别

##### 6.7.1 auto变量

##### 6.7.2 static变量

##### 6.7.3 register变量

##### 6.7.4 extern变量

#### 6.8小结

#### 6.9实践与练习



## 第7章 数组、指针和引用

### 7.1 一维数组

#### 7.1.1 一维数组的声明

#### 7.1.2 一维数组的引用

#### 7.1.3 一维数组的初始化

### 7.2 二维数组

#### 7.2.1 二维数组的声明

#### 7.2.2 二维数组元素的引用

#### 7.2.3 二维数组的初始化

### 7.3 字符数组

### 7.4 指针

#### 7.4.1 变量与指针

#### 7.4.2 指针运算符和取地址运算符

#### 7.4.3 指针运算

### 7.5 指针与数组

#### 7.5.1 数组的存储

#### 7.5.2 指针与一维数组

#### 7.5.3 指针与二维数组

#### 7.5.4 指针与字符数组

### 7.6 指向函数的指针

### 7.7 引用

#### 7.7.1 使用引用传递参数

#### 7.7.2 指针传递参数

#### 7.7.3 数组作函数参数

### 7.8 指针数组

### 7.9 小结

### 7.10 实践与练习

## 第8章 构造数据类型

### 8.1 结构体

#### 8.1.1 结构体定义

#### 8.1.2 结构体变量

#### 8.1.3 结构体成员及初始化

#### 8.1.4 结构体的嵌套

#### 8.1.5 结构体大小

### 8.2 结构体与函数

#### 8.2.1 结构体变量作函数参数

#### 8.2.2 结构体指针作函数参数

### 8.3 结构体数组

#### 8.3.1 结构体数组声明与引用

#### 8.3.2 指针访问结构体数组

### 8.4 共用体

#### 8.4.1 共用体的定义与声明

- 8.4.2共用体的大小
    - 8.4.3共用体的特点
  - 8.5枚举类型
    - 8.5.1枚举类型的声明
    - 8.5.2枚举类型变量
    - 8.5.3枚举类型的运算
  - 8.6自定义数据类型
  - 8.7小结
  - 8.8实践与练习
- 第2篇 核心技术
  - 第9章 面向对象编程
    - 9.1面向对象概述
    - 9.2面向对象与面向过程编程
      - 9.2.1面向过程编程
      - 9.2.2面向对象编程
      - 9.2.3面向对象的特点
    - 9.3统一建模语言
      - 9.3.1统一建模语言概述
      - 9.3.2统一建模语言的结构
      - 9.3.3面向对象的建模
    - 9.4小结
  - 第10章 类和对象
    - 10.1 C++类
      - 10.1.1类概述
      - 10.1.2类的声明与定义
      - 10.1.3类的实现
      - 10.1.4对象的声明
    - 10.2构造函数
      - 10.2.1构造函数概述
      - 10.2.2复制构造函数
    - 10.3析构函数
    - 10.4类成员
      - 10.4.1访问类成员
      - 10.4.2内联成员函数
      - 10.4.3静态类成员
      - 10.4.4隐藏的this指针
      - 10.4.5嵌套类
      - 10.4.6局部类
    - 10.5友元
      - 10.5.1友元概述
      - 10.5.2友元类
      - 10.5.3友元方法

## 10.6命名空间

### 10.6.1使用命名空间

### 10.6.2定义命名空间

### 10.6.3在多个文件中定义命名空间

### 10.6.4定义嵌套的命名空间

### 10.6.5定义未命名的命名空间

## 10.7小结

## 10.8实践与练习

# 第11章 继承与派生

## 11.1继承

### 11.1.1类的继承

### 11.1.2继承后可访问性

### 11.1.3构造函数访问顺序

### 11.1.4子类隐藏父类的成员函数

## 11.2重载运算符

### 11.2.1重载运算符的必要性

### 11.2.2重载运算符的形式与规则

### 11.2.3重载运算符的运算

### 11.2.4转换运算符

## 11.3多重继承

### 11.3.1多重继承定义

### 11.3.2二义性

### 11.3.3多重继承的构造顺序

## 11.4多态

### 11.4.1虚函数概述

### 11.4.2利用虚函数实现动态绑定

### 11.4.3虚继承

## 11.5抽象类

### 11.5.1纯虚函数

### 11.5.2实现抽象类中的成员函数

## 11.6小结

## 11.7实践与练习

# 第3篇 高级应用

## 第12章 模板

## 12.1函数模板

### 12.1.1函数模板的定义

### 12.1.2函数模板的作用

### 12.1.3重载函数模板

## 12.2类模板

### 12.2.1类模板的定义与声明

### 12.2.2简单类模板

### 12.2.3默认模板参数

- 12.2.4为具体类型的参数提供默认值
    - 12.2.5有界数组模板
  - 12.3模板的使用
    - 12.3.1定制类模板
    - 12.3.2定制类模板成员函数
    - 12.3.3模板部分定制
  - 12.4链表类模板
    - 12.4.1链表
    - 12.4.2链表类模板
    - 12.4.3类模板的静态数据成员
  - 12.5小结
  - 12.6实践与练习
- 第13章STL标准模板库
- 13.1序列容器
    - 13.1.1向量类模板
    - 13.1.2双端队列类模板
    - 13.1.3链表类模板
  - 13.2结合容器
    - 13.2.1 set类模板
    - 13.2.2 multiset类模板
    - 13.2.3 map类模板
    - 13.2.4 multimap类模板
  - 13.3算法
    - 13.3.1非修正序列算法
    - 13.3.2修正序列算法
    - 13.3.3排序算法
    - 13.3.4数值算法
  - 13.4迭代器
    - 13.4.1输出迭代器
    - 13.4.2输入迭代器
    - 13.4.3前向迭代器
    - 13.4.4双向迭代器
    - 13.4.5随机访问迭代器
  - 13.5小结
  - 13.6实践与练习
- 第14章RTTI与异常处理
- 14.1 RTTI (运行时类型识别)
    - 14.1.1什么是RTTI
    - 14.1.2 RTTI与引用
    - 14.1.3 RTTI与多重继承
    - 14.1.4 RTTI映射语法
  - 14.2异常处理

- 14.2.1 抛出异常
  - 14.2.2 异常捕获
  - 14.2.3 异常匹配
  - 14.2.4 标准异常
- 14.3 小结
- 14.4 实践与练习
- 第15章 程序调试
  - 15.1 选择正确的调试方法
  - 15.2 程序错误常见的4种类型
    - 15.2.1 语法错误
    - 15.2.2 连接错误
    - 15.2.3 运行时错误
    - 15.2.4 逻辑错误
  - 15.3 调试工具的使用
    - 15.3.1 创建调试程序
    - 15.3.2 进入调试状态
    - 15.3.3 Watch窗口
    - 15.3.4 Call Stack窗口
    - 15.3.5 Memory窗口
    - 15.3.6 Variables窗口
    - 15.3.7 Registers窗口
    - 15.3.8 Disassembly窗口
  - 15.4 调试的基本应用
    - 15.4.1 变量的跟踪与查看
    - 15.4.2 位置断点的使用
    - 15.4.3 数据断点的使用
  - 15.5 调试的高级应用
    - 15.5.1 在调试时修改变量的值
    - 15.5.2 在循环中调试
  - 15.6 小结
  - 15.7 实践与练习
- 第16章 文件操作
  - 16.1 文件流
    - 16.1.1 C++中的流类库
    - 16.1.2 类库的使用
    - 16.1.3 ios类中的枚举常量
    - 16.1.4 流的输入/输出
  - 16.2 文件打开
    - 16.2.1 打开方式
    - 16.2.2 默认打开模式
    - 16.2.3 打开文件同时创建文件
  - 16.3 文件的读写



- 16.3.1 文件流
    - 16.3.2 写文本文件
    - 16.3.3 读取文本文件
    - 16.3.4 二进制文件的读写
    - 16.3.5 实现文件复制
  - 16.4 文件指针移动操作
    - 16.4.1 文件错误与状态
    - 16.4.2 文件的追加
    - 16.4.3 文件结尾的判断
    - 16.4.4 在指定位置读写文件
  - 16.5 文件和流的关联和分离
  - 16.6 删除文件
  - 16.7 小结
  - 16.8 实践与练习
- 第17章 网络通信
- 17.1 TCP/IP协议
    - 17.1.1 OSI 参考模型
    - 17.1.2 TCP/IP 参考模型
    - 17.1.3 IP地址
    - 17.1.4 数据包格式
  - 17.2 套接字
    - 17.2.1 Winsock套接字
    - 17.2.2 Winsock的使用
    - 17.2.3 套接字阻塞模式
    - 17.2.4 字节顺序
    - 17.2.5 面向连接流
    - 17.2.6 面向无连接流
  - 17.3 简单协议通信
    - 17.3.1 服务端
    - 17.3.2 客户端
    - 17.3.3 实例的运行
  - 17.4 小结
  - 17.5 实践与练习
- 第4篇 项目实战
- 第18章 图书管理系统
- 18.1 系统设计
    - 18.1.1 需求分析
    - 18.1.2 系统目标
    - 18.1.3 系统功能结构
  - 18.2 图书类
  - 18.3 主程序
  - 18.4 添加图书

18.5显示图书信息

18.6删除图书

18.7小结