

计算机程序设计学 习指南

*Guide to Computer Program-
ming*

作者：钱院学辅程设编写小组

2020 年 2 月 8 日

钱学森书院学业辅导中心

QIAN YUAN XUE FU

XI'AN JIAOTONG UNIVERSITY

作品信息

- 标题：计算机程序设计学习指南 - *Guide to Computer Programming*
- 作者：钱院学辅程设编写小组
- 校对排版：钱院学辅排版组
- 出品时间：2020 年 2 月 8 日
- 总页数：93

许可证说明

 知识共享 (Creative Commons) BY-NC-ND 4.0 协议

本作品采用 **CC 协议** 进行许可。使用者可以在给出作者署名及资料来源的前提下对本作品进行转载，但不得对本作品进行修改，亦不得基于本作品进行二次创作，不得将本作品运用于商业用途。

本作品已发布于 **GitHub** 之上，发布地址为：

<https://github.com/qyxf/Tutorials/>

本作品的版本号为 **v1.0**。


前言

计算机程序设计 (Computer Programming) 是钱学森试验班的同学在大一下学期要学习的一门必修课程。这门课程主要讲授以 C++ 语言为基础的编程知识, 涵盖了从基本语法、数组、指针、函数等编程基础, 到类与对象、继承、多态等 C++ 特色知识。本课程一周只有 2 学时的课堂教学, 之后就是通过一个个的实验来熟悉掌握编程知识, 不断提高编程能力。对于初涉编程领域的同学来说, 这门课会相对比较棘手。





为了满足同学们的预习需求与复习需要, 从 2019 年 7 月中旬起, 钱院学辅组织了几位 18 级刚刚修过计算机程序设计课程的同学编写了这一份手册。它主要是一本预习指南, 但更是编写成员们的学习心得汇总。诚然, 有很多网课都可解决这一问题, 比如中国大学 mooc 上翁恺老师的课程就帮助笔者度过了预习这一难关。不过相信这份指南可以帮你更加有针对性地解决你在这门课程上可能遇到的问题。

在本手册发布之前, 已经有不少同学在询问编写情况。在此, 感谢各位同学对钱院学辅的关注。本题解由钱院学辅程设编写小组完成, 成员包括: 钱学森 82 班路敬宇、钱学森 82 班彭贺宇、钱学森 84 班李浩天、钱学森 84 班杨雨龙。其中, 路敬宇编写了第一、二章的内容, 彭贺宇编写了第三、四章的内容, 李浩天编写了第五到九章, 杨雨龙编写了第十到十三章的内容并对第二、四、五章做了一些补充。此外, 钱学森 82 班黄子硕、钱学森 82 班赵恒欣、钱学森 84 班费立涵三位同学也对稿件的编写提供了宝贵的建议。本手册采用 \LaTeX 整理, 排版由李浩天完成。在此, 向这些同学表示感谢。

虽然本手册编写前后历经六个月有余, 但笔误、错漏等在所难免, 需要各位读者帮助我们指正。如您在参考的过程中发现有任何错误之处, 欢迎您通过下面的方式联系我们, 帮助我们改进这份资料:

-  GitHub 平台论坛 (推荐, 但需要注册): <https://github.com/qyxf/BookHub/issues>



-  学辅信息站: <https://qyxf.site>
-  钱院学辅邮箱: qianyuanxuefu@163.com
-  钱院学辅官方答疑墙: 钱小辅 qq: 206713407
-  钱院学辅交流分享 qq 群 (可进群直接联系编写成员): 852768981

最后, 祝各位同学元宵节快乐, 也希望疫情早日结束!

钱院学辅程设编写小组

2020 年 2 月 8 日



目录

第一章 C++ 集成开发环境的使用	1
§1.1 集成开发环境概述	1
1.1.1 我们使用的 IDE	1
§1.2 C++ 程序的基本特点	2
1.2.1 C++ 程序基本结构	2
1.2.2 C++ 程序的基本要素	3
1.2.3 输入、编译、调试和运行一个 C++ 程序	3
1.2.4 编译预处理	4
1.2.5 *名字空间 <i>namespace</i>	4
第二章 基本数据类型及其运算	5
§2.1 数据类型	5
2.1.1 整型数据	5
2.1.2 实型数据	5
2.1.3 常量	6
§2.2 修饰符	6
§2.3 变量	6
§2.4 表达式	6
2.4.1 问号表达式与逗号表达式	7
2.4.2 *位运算表达式	7
2.4.3 自增/减运算符和表达式的副作用	7
§2.5 补充：基本数据类型	7
2.5.1 必要的基础知识	7
2.5.2 基本数据类型占内存大小	8
2.5.3 易错点：浮点数判相等	9
2.5.4 表达式的值和表达式的副作用	9
2.5.5 位运算的应用	11
第三章 选择与循环程序设计	12
§3.1 运算的流程控制	12
§3.2 选择控制结构	12
3.2.1 <i>if</i> 语句	12
3.2.2 <i>switch</i> 语句	13
3.2.3 关于选择语句的其他说明	15

§3.3 循环控制结构	17
3.3.1 <i>for</i> 语句	17
3.3.2 <i>while</i> 语句	17
3.3.3 <i>do while</i> 语句	18
§3.4 关于循环语句的说明	19
3.4.1 一定记得敲大括号	19
3.4.2 关于循环套用	19
3.4.3 三种循环的选取	20
第四章 数组与字符串的输入与输出	21
§4.1 数组与字符串	21
4.1.1 分类	21
4.1.2 两者相同点	21
4.1.3 两者不同点	22
4.1.4 其他注意事项	22
4.1.5 补充：数组	25
第五章 函数的声明、定义和调用	26
§5.1 模块与函数的概念	26
§5.2 函数的定义与调用	26
5.2.1 函数的定义	26
5.2.2 函数的调用	28
§5.3 函数的参数	29
5.3.1 形式参数的作用	29
5.3.2 函数间的参数传递	29
5.3.3 初始化形参	30
§5.4 函数原型与调用方式	31
5.4.1 函数原型	31
5.4.2 函数的调用方式	32
§5.5 局部变量与全局变量	33
§5.6 补充内容	34
5.6.1 关于 <i>main</i>	34
5.6.2 常用库函数	35
5.6.3 变量的存储类别	38
5.6.4 变量在内存中的布局	38

§5.7 常用函数附录	39
5.7.1 取系统时间	39
5.7.2 <i>rand</i> 函数	39
第六章 指针的声明和使用	42
§6.1 指针的概念	42
6.1.1 地址	42
6.1.2 指针	42
§6.2 指针的声明及运算	42
6.2.1 指针变量的声明	42
6.2.2 *和&运算符	43
6.2.3 指针的赋值运算	44
6.2.4 指针的关系运算	44
6.2.5 指针的算术运算	45
§6.3 指针与数组	45
6.3.1 数组	45
6.3.2 指针与字符数组	46
§6.4 动态存储分配	48
§6.5 补充内容	50
6.5.1 动态申请二维数组	50
6.5.2 指针和引用	50
6.5.3 指针数组	52
6.5.4 指向指针的指针	52
6.5.5 指针与结构体	52
6.5.6 指针数组的初始化	53
6.5.7 <i>void</i> 和 <i>const</i> 类型的指针	53
第七章 函数和指针程序设计	55
§7.1 指针作为函数的参数	55
§7.2 指针作为函数返回值	55
§7.3 指向函数的指针	56
§7.4 函数嵌套	57
7.4.1 递归函数	57
7.4.2 重载函数	58
7.4.3 带参数的 <i>main</i> () 函数	59

第八章 结构体与枚举	61
§8.1 结构体	61
§8.2 枚举	62
第九章 类和对象	63
§9.1 类与对象的声明	63
9.1.1 类的声明	63
9.1.2 对象的定义	64
§9.2 成员函数	64
§9.3 对象的访问	65
§9.4 静态成员	66
§9.5 结构体与类的区别与联系	66
第十章 类的构造与析构	67
§10.1 知识要点	67
10.1.1 构造函数	67
10.1.2 析构函数	67
10.1.3 对象和指针	68
10.1.4 <i>string</i> 类的成员函数	68
§10.2 重难点分析	68
10.2.1 什么是缺省构造函数?	68
10.2.2 拷贝构造函数	69
第十一章 继承	75
§11.1 知识要点	75
11.1.1 继承的基本概念	75
11.1.2 派生类	75
11.1.3 派生类的继承方式和访问属性	75
11.1.4 派生类的构造函数和析构函数	76
§11.2 重难点分析	76
11.2.1 软件重用的办法	76
第十二章 多态性	80
§12.1 知识要点	80
12.1.1 多态性概述	80
12.1.2 派生类对象替换基类对象	80
12.1.3 虚函数	80
12.1.4 运算符重载	81

§12.2 重难点分析	85
12.2.1 <i>const</i> 修饰符	85
12.2.2 赋值运算符重载	89
第十三章 标准库和输入输出流	91
§13.1 知识要点	91
13.1.1 <i>C++</i> 流概述	91
§13.2 重难点分析	91
13.2.1 文件读写	91

现代汉语词典

第一章 C++ 集成开发环境的使用

§1.1 集成开发环境概述

集成开发环境（后文用 IDE 代替）是指用于提供程序开发环境的应用程序，一般包括代码编辑器、编译器、调试器和图形用户界面等工具。集成了代码编写功能、分析功能、编译功能、调试功能等一体化的开发软件服务套。所有具备这一特性的软件或者软件套（组）都可以叫集成开发环境。如微软的 VisualStudio 系列等。

所谓工欲善其事，必先利其器。你需要找一个优秀的母鸡（指合格合适的 IDE），并予以精心照顾（指认真写程序），才能获得一颗颗好蛋（指没有 bug 能够出色高时空效率完成任务的程序）。

IDE 的种类有许多。一个 IDE 之所以能够叫做 IDE，并不是编者“诶我觉得这个可以有”就能叫的。IDE 也有基本法。对 C 语言而言，最早的基本法是 ANSIC，后来也出现了 StandardC、ISOC、C++11 这样的标准（不过考虑到教学不会涉及到这些内容，笔者在这里也仅仅只做一个简要的介绍）。不同的编译器甚至不同版本的同一个编译器可能会遵循不同的标准，所以即使是能让一个母鸡下出完美的蛋的培养方案，原封不动地拿去伺候它的长辈，也可能导致“颗粒无收”的结局。

IDE 具体内容因为笔者也不大了解，并且学习计算机程序设计这门课也没必要了解太深，此处就不再深说，也免得说错什么误人子弟。

1.1.1 我们使用的 IDE

这门课顾刚老师的官方指定是 DEV-C++ 或 VisualStudio（下文简称 VS）。好像 17 届钱班用的是 VS，到笔者所在的 18 届改用 DEV-C++ 了。所以后边会用什么笔者不好判断，所以在此把二者都简单说说吧。

1) 关于 VS



VS 笔者用的不多，不甚了解。不过大一小学期如果有同志想选写小游戏的课程的话，那个课程使用的编译器就是 VS。具体的安装、运行过程书上说的其实已经很清楚了。相较于 DEV-C++，VS 更加专业，同时也能拓展运行其他语言，所以适用性更强。很多专业人士用的就是 VS。同样的，其缺点也十分明显：对初学者而言，操作按钮界面也太花花绿绿了吧（完全不知道都是些什么.jpg）。

2) 关于 DEV-C++

Dev-C++ 的优点是功能简洁，适合于在教学中供 C/C++ 语言初学者使用；缺点是它的功能并不完善，甚至还存在一些逻辑漏洞，所以它适合也只适合于初学者使用，在商业级的软件开发中没有其身影。

初学 C++ 的时候，用 DEV-C++ 是一个不错的选择，因为他没有 VS 那样令人眼花缭乱的界面按钮，并且考虑到最后的考试的代码也就撑死 100 行左右的样子，DEV-C++ 的优势就大了。

个人感觉，DEV-C++ 足够本学期学程序设计用，并且它本身体积也小，也不用纠结要不要放 C 盘这样的问题，实属入门菜鸟の利器。

另外说一句，顾刚老师在电子教室给的 DEV-C++ 版本可能不是最新版本。考虑到他有时会测试你的实验报告代码，所以为了减少不必要的麻烦，防止因为版本不同造成的无法执行，建议还是采用他给出的软件进行编译。

§1.2 C++ 程序的基本特点

1.2.1 C++ 程序基本结构

先看一个样例

```
//Example-C++程序基本结构
//在屏幕上显示“程序设计小助手”字样
#include<iostream>
int main(){
    std :: cout << "程序设计小助手" << std :: endl ;
    return 0;
}
```

//-> 表示注释。非程序执行语句，用于程序猿之间交流以及程序猿自身隔日看程序时起到提醒作用。

#-> 表示预处理。通常是引用库函数或者是定义全局变量、函数。



写程序为什么要预处理？有时我们在编写程序的时候可能需要它满足一些功能，但是这些功能已经有一些前人能够以最优效率完成了，所以他们就留下了叫做“库函数”的暗盒，我们只需要按照暗盒的使用规则就可以用更简单的代码做到相同的功能。

`int main()` -> 主函数，一个程序有且只有一个主函数，也是程序执行的主体部分。一个 C++ 程序可以有多个函数，但是只能有且只有一个主函数——主函数是程序运行的主体。

`return 0` -> 表示主函数的结束。这句话是给计算机说的：“啊这个程序已经执行完了，辛苦你了，你可以洗洗睡了。”

高版本的 IDE 通常不写这句话程序也是能执行的这样更加人性化（计算机发现没有需要执行的指令就自己洗洗睡去了）。写这句话就好像睡前需要洗脸刷牙，你想直接睡觉也没人拦你，但总是做了总是好的（起码针对考试来说，不写这个好像是要扣分的）。

1.2.2 C++ 程序的基本要素

具体内容，诸如关键词、标点符号、注释什么的书上说的很清楚，不过对于新手来说可能“看不清楚”，对于一些概念看不明白、理解不能。对此，笔者的建议是可以先“不求甚解”，等到学到输入输出流再返回来看这一章节，就能更好地理解这些操作、内容都是什么意思。至于现阶段，先记住就行了，不用太纠结于 why。

一句题外话，等到学完了所有内容，回过头看时，你可能也会和笔者有相同的看法：整本教材的编排并不是很好。教材是分类式编排，就导致其实有些书上内容知识点出现的比较早但是讲的比较晚（如枚举类型虽然在第二章就出现了，但是讲到它算是比较靠后了；又比如输入输出流，就是 `#include<iostream>` 的解释在最后才学）。所以有些东西，先记住，再理解是一个较为明智的选择。

1.2.3 输入、编译、调试和运行一个 C++ 程序

谁能保证自己一次就能写出没有任何错误、符合要求的程序呢？恐怕是资深程序猿也不能做到这一点。

所以对于我们新手来说，调试、修改、再调试是前往成功道路上必不可少的弯路。期间可以无奈，找朋友吐槽这只鸡就是不下蛋会不会是公鸡；可以暴躁，亲切问候一下计算机的祖上（什么这台电脑没有祖上？那没事了），但是请不要心急。心急是写程序的大忌。想要拿到好成绩的话，坚持与耐心是必不



可少的。

另外还有一个小建议，代码的书写也很重要，特别是代码的对齐。整洁的代码卷面更有利于隔夜代码的复习或他人的研读。所以建议大家从一开始就规范自己的代码书写，养成良好的学习习惯。

1.2.4 编译预处理

这一部分最主要的应用是在程序中进行宏定义——不过虽然宏定义很简洁方便，但是也是一大受人吐槽的点，因为声明之后会有不利影响，比如占内存、影响后续程序变量名字使用之类的。并且宏定义之后是一个常量而非变量，灵活性差，一些常数比如 `pi` 的声明倒可以先宏定义一下其值，还算方便。具体的使用可以看一下老师给的范例，你会有更好地了解。

不过也要声明一下，函数预处理我个人没用过，看的范例印象里也没有人用，所以要不要深学就看个人了。

1.2.5 * 名字空间 namespace

一句话概括：这个学期你们除了 `using namespace std;` 之外不会再写任何一句和这个有关系的语句。应试（或者说实用）角度意义不大。具体什么东西未来章节会详细说明的。



第二章 基本数据类型及其运算

§2.1 数据类型

程序的主要任务就是对数据进行处理，而数据的类型多种多样，如文字类型、图像类型、声音类型……其中最基本最常见的是数值类型数据和文字类型数据。

计算机在处理数据时，首先会将数据存放在内存中，那么显然不同类型数据存放格式应当不同；而有时同一类型数据也可以根据问题的特点进行不同类型的储存。例如数字类型数据按照储存格式可以分为整型、浮点型、双精度型等几种类型；文字类型数据也能分为单个字符和字符串。

2.1.1 整型数据

(看看隔壁 python 的整型数据，C++ 简直就是辣鸡)

使用注意事项：`int` 的能力是有极限的（指存在储存数字的范围），笔者从短暂的程序学习当中学到一件事，越是玩弄整数运算，就越会发现 `int` 的能力是有极限的，除非超越 `int`……

好了不玩梗了。一般而言，`int` 在使用时可以利用其整数除法的特点在程序中做些文章，至于要涉及到小数运算用 `double` 就行了，虽然这样有违背程序空间效率的原则，但是可以无脑爽，减少思考量，保护自己的头发（划掉）。当然，`int` 在很多地方还是不可被代替的。如 `switch` 语句，只能输入 `int` 型变量。

2.1.2 实型数据

分为 `float` 浮点和 `double` 双精度两种类型。

有一说一，`float` 确实用处不算大——如果不考虑什么空间效率尤为如此。既生 `double` 何生 `float`（当然这是我个人理解的玩笑话）。



2.1.3 常量

常量分为两大类，数据和字符（其实变量也是）。

字符使用 ASCII 码进行编译，可以掌握记牢一些常用的转义字符，比如 `\b`、`\t`、`\n` 指令还是比较常见常用的。

要特别注意字符串的尾部有 `\0` 这样的结束符存在占位，这也是后文申请字符串数组时常见的一个问题：要不要 +1 个空位（不是复读那个 +1）。

§2.2 修饰符

修饰符这玩意 其实你写程序不用修饰符也可以完成题目，不过顾老师会在他的范例程序里边用，又考虑到“存在即合理”，所以最好还是看看吧。

要格外注意 `const` 这一常量修饰符。在后边函数、类章节中这一修饰符将成为一大杀器，很多人学完了这门课程都不知道说出个所以然。因为你们还没有深接触，这里不加赘述，仅仅提醒这个的重要性。

类型修饰符可以看看。

§2.3 变量

变量就像是我们日常用语中的指示代词，究竟指什么东西要根据上文语境对其“赋值”，并且会因为下文“重新赋值”而变化其内涵。并且就像“TA”和“TA 们”作为变量会指示不同数量的事物一样（比如你总不会用“它们”来指代一名人类吧），不同类型的数据要使用对应类型的变量声明、赋值。

具体语句用法可以见书，书上的说明个人觉得还是挺明确的。在这里也提个醒，书上是存在一些错误的，不要将书本内容奉为圭臬，尽信书不如无书。

§2.4 表达式

C++ 中的表达式有很多，在之后你们也会进行更加系统的学习。表达式同时也很重要，是程序必不可少的要素。表达式可能会迟到，但是永远不会缺席。

有时要注意表达式运算符的运行顺序。如果真的脑子一片浆糊，可以使用 `()` 括出你想要的运算顺序。



2.4.1 问号表达式与逗号表达式

问号表达式其实可以视为一个浓缩的 `if` 语句。可以在一定程度上减少代码行数，简化程序，所以还是挺值得一用的。

逗号表达式可以将几行语句叠加在一起，可以减少程序的行数。个人比较喜欢将若干行可以视为整体的代码整合在一行里，比如声明多个同类型变量，或者是几个变量交换自己的值的代码这样的情况。合理使用逗号表达式，可以提升程序的可读性和简洁性。

2.4.2 * 位运算表达式

看书就行，有他的作用，不过现阶段学习可以无视。

我有朋友（这个朋友不是我自己.jpg）说见过一个提取数字位数的用位运算表达式的神程序，不过对于入门阶段的我们就不用过多考虑这个功能了。

除了在理论学习碰见这个，实战——起码对于入门新手——作用不大。

2.4.3 自增/减运算符和表达式的副作用

这两个内容放在一起说，原因在于其作用类似，都是在原先变量基础上进行运算，不过自增/减语句只在原数据基础上 ± 1 ，而 “`+-*/`” \cup “`=`” 的组合则是按照你想的来在原基础上变化。

合理利用这些式子，可以提升时空效率，同时程序也更加简洁易懂。

§2.5 补充：基本数据类型

2.5.1 必要的基础知识

1.bit：是计算机中最小的数据单位，简写为 `b`。他表示逻辑器件的状态：`1` 表示闭合，`0` 表示断开。

2. 字节（Byte）：8 位二进制码称为 1 字节



2.5.2 基本数据类型占内存大小

数据类型	字节长度	位数
int	4	32
float	4	32
double	8	64
char	1	8
bool	1	8
long	4	32
long long	8	64

tips: 用 `sizeof()` 库函数可以很方便的求出任意变量的内存空间大小。源码如下:

```
#include<iostream>
using namespace std;
int main(){
    int a = 0;
    float b = 0.0;
    double c = 0.0;
    bool d = 0;
    char e = 'x';
    long f = 0;
    long long g = 0;
    cout<<"int: "<<sizeof(a)<<endl;
    cout<<"float: "<<sizeof(b)<<endl;
    cout<<"double: "<<sizeof(c)<<endl;
    cout<<"bool: "<<sizeof(d)<<endl;
    cout<<"char: "<<sizeof(e)<<endl;
    cout<<"long: "<<sizeof(f)<<endl;
    cout<<"long long: "<<sizeof(g)<<endl;
    return 0;
}
```



```
C:\Users\lenovo\Documents\C++\cpp\memory.exe
int: 4
float: 4
double: 8
bool: 1
char: 1
long: 4
long long: 8

-----
Process exited after 0.4156 seconds with return value 0
请按任意键继续. . .
```

int 型数据的最大值是 $2^{31}-1 = 2,147,483,647$ ；在 10^9 数量级。

long long 型数据的最大值是 $2^{63}-1 = 9,223,372,036,854,775,807$ ；在 10^{18} 数量级。牢记这两项数据不仅可以帮助我们选择合适的数据类型，对于调试一些由于溢出而导致的 BUG 也很有帮助。

2.5.3 易错点：浮点数判相等

阅读以下程序：

```
#include<iostream>
using namespace std;
int main(){
    for (double i = 0.0; i != 10.0; i+=0.1)
        cout<<i<<" ";
    return 0;
}
```

请问最终会有多少个数字被输出？

答案是：这是一个死循环。

原因就在于浮点数的精度问题。由于计算机当中浮点数的表示永远存在一定的误差，因此以上程序里面的 i 是永远也不会恰好等于 10.0 的。如果要判定两个浮点数相等，最好的办法是设置一个允许的误差 ϵ 比如 $1e-7$ (这是 1×10^{-7} 的意思)，只要这两个浮点数之间的距离小于 ϵ ，就认为这两个浮点数相等。

2.5.4 表达式的值和表达式的副作用

这一部分的概念一开始比较难于接受，要尽量仔细听老师讲，逐渐熟悉表达式的概念。

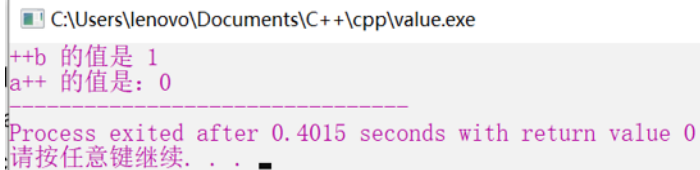


表达式的值是指表达式的运算结果。表达式的副作用一般是指带有赋值运算符“=”的表达式，还有“++，-”这一类表达式，除了有它本身的值之外还能修改某个变量的值。

(1) 赋值运算符的值是赋值号左边那个变量的值

(2) 前缀++表达式的值是变量自增1之后的值，即“先++后取值”；后缀++表达式的值是变量自增1之前的值，即“先取值后++”；读者可以自己编程来验证这件事，我这里验证了，如下：

```
#include<iostream>
using namespace std;
int main(){
    int a = 0;
    int b = 0;
    cout<<"++b 的值是 "<<++b<<endl;
    cout<<"a++ 的值是："<<a++;
    return 0;
}
```



C:\Users\lenovo\Documents\C++\cpp\value.exe

```
++b 的值是 1
a++ 的值是: 0
-----
Process exited after 0.4015 seconds with return value 0
请按任意键继续. . .
```

还可以结合数组来理解这件事：

```
#include<iostream>
using namespace std;
int main(){
    int A[3] = {1,2,3};
    int pos = 0;
    //先取值后++, pos++表达式的值是0，因此取到的是A[0]
    cout<<"A[pos++] = "<<A[pos++]<<endl;
    pos = 0;
    //先++后取值， ++pos表达式的值是1，因此取到的是A[1]
    cout<<"A[++pos] = "<<A[++pos]<<endl;
    return 0;
}
```



```
C:\Users\lenovo\Documents\C++\cpp\value.exe
A[pos++] = 1
A[++pos] = 2
-----
Process exited after 0.3587 seconds with return value 0
请按任意键继续. . .
```

2.5.5 位运算的应用

(1) 如何将一个整数的某个二进制位置成 1? (在计算机当中这个操作叫做 set)

如何将一个整数的某个二进制位置成 0? (在计算机当中这个操作叫做 reset)

如何将一个整数的某个二进制位翻转?

如何判断一个整数的某个二进制位是否为 1?

答案:

```
set: num|=1<<x;
reset: num&= ~(1<<x);
flip: num^=1<<x
test: num>>x &1
```

(2) 课本 P34 例 2-6 取一个整型变量的最低 4 位: 将其与 16 进制数 0X00F 做按位与运算: $x \&= 0x0f$;

(3) 一句话交换: 我们通常用的交换两个变量的值是需要借助于第三个变量, 而且要三句话才能完成。但是用异或运算一句话就可以, 而且不需要借助第三个变量: $a^= b \wedge a^= b$;



第三章 选择与循环程序设计

§3.1 运算的流程控制

主要分为三种：顺序控制结构，选择控制结构，循环控制结构。其中顺序控制结构没有什么好说的，下面主要介绍选择与循环程序设计。

§3.2 选择控制结构

3.2.1 if 语句

经典结构，具体用法如下：

```
if(条件){  
    <语句1(符合条件是执行的语句)>;  
}  
else{  
    <语句2(不符合条件是执行的语句)>;  
}
```

ps3.1:

强烈建议初学者使用这种语句, 以养成在最后记得加大括号的习惯
另一种可以接受的写法:

```
if(条件)  
    <语句1(符合条件是执行的语句)>;  
else  
    <语句2(不符合条件是执行的语句)>;
```

ps3.2:



语句 1 和语句 2 中不能有分号 (切记), 否则会自动跳出控制结构。
例如下面的程序:

```
#include<iostream>
using namespace std;
int main()
{
    int a = 0 ;
    if( a == 0 )
        cout << "a=0" << endl ;
    else
        cout << "a=1" << endl ;
        cout << "a!=2" ;
    return 0 ;
}
```

运行结果如下:



```
a=0
a!=2
```

ps3.3:

if 语句中没有 else 也是可以接受的, 例如:

```
#include<iostream>
using namespace std;
main(){
    int a=0;
    if(a==0)
        cout<<"a=0"<<endl;
    return 0;
}
```

运行结果如下:



```
a=0
```

3.2.2 switch 语句

也是一种十分经典的结构, 具体用法如下:



```

switch (<变量名>){
    case <变量值1>:
        <语句1(当变量为变量值1时执行的语句)>;
        break;
    case <变量值2>:
        <语句2(当变量为变量值2时执行的语句)>;
        break;
    .....
    default
        <语句(当变量不等于上述变量值时执行的语句)>;
}

```

ps3.4:

此处的 **break** 一定要记得打（除了最后一个）。若是忘记打，例如：

```

#include<iostream>
using namespace std;
main(){
    int a=1;
    switch (a){
        case 0:
            cout<<"a=0"<<endl;
            break;
        case 1:
            cout<<"a=1"<<endl;
        case 2:
            cout<<"a=2"<<endl;
            break;
        default:
            cout<<"我也不知道a是几"<<endl;
    };
    return 0;
}

```

结果如下：

```

a=1
a=2

```



ps3.5:

事实上，以下语句 3-1 与 switch 语句 3-2 等价：

语句 3-1:

```
if(a==0)
    goto a0;
else if(a==1)
    goto a1;
else
    goto de;
a0:b=0;
goto end;
a1:b=1;
goto end;
de:b=2;
end:cout<<b;
```

语句 3-2:

```
switch(a) {
    case 1:
        b=1;
        break;
    case 0:
        b=0;
        break;
    default:
        b=2;
}
cout<<b;
```

3.2.3 关于选择语句的其他说明

关于 else if 的说明

对于 else if，我们应当以如下的方式去理解：(注意缩进哦)

语句 3.3



```
if(<条件1>
    <语句1>;
else if(<条件2>
    <语句2>;
else if(<条件3>
    <语句3>
```

相当于以下语句语句 3.4

```
if(<条件1>
    <语句1>;
else
    if(<条件2>
        <语句2>;
else
    if(<条件3>
        <语句3>;
```

多 if 以分号结尾

对多个 if 以分号结尾，c++ 是这样定义的认为前面的 if 已经结束了，else 至多只能有一个，表示对最后一个 if 不成立时所执行的语句，例如：

```
#include<iostream>
using namespace std;
main(){
    int a=0;
    if(a==0)
        cout<<"a=0"<<endl;
    //此处若是没有if，则认为此选择语句以没有else的方式结束
    if(a==1)
        cout<<"a=1"<<endl;
    else
        cout<<"a=2";
    return 0;
}
```



```
a=0
a=2
```

运行结果如图：

§3.3 循环控制结构

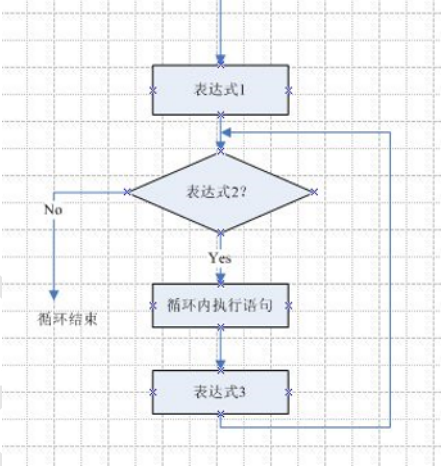
3.3.1 for 语句

for 语句的调用长这个样子：

```
for(<语句1>;<条件>;<语句2>){
    <语句3>;
}
```

它指的是先执行语句 1, 然后判断条件, 若为真, 执行语句 3, 然后执行语句 2, 再判断条件, 若为真, 执行语句 3, 然后执行语句 2……直到条件不满足跳出循环。

引用一下来自网页的程序框图：



ps3.6：对于语句 3 的要求与 if 中的结构要求相同，若是没有分号，大括号是可以省略的。（此后的 while 语句和 do while 语句也是如此，不再赘述）

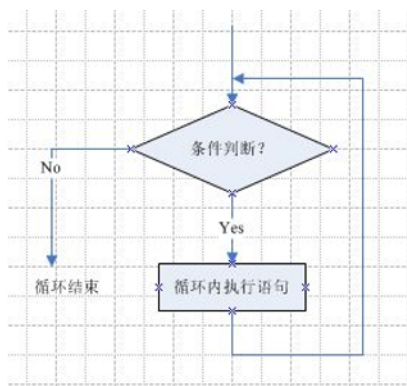
3.3.2 while 语句

while 语句是这样的：

```
while(<条件>){
    <语句>;
}
```

它的意思是先判断条件，然后执行语句，再判断条件，执行语句……直到条件不满足时跳出。

再引用一下来自网页的程序框图



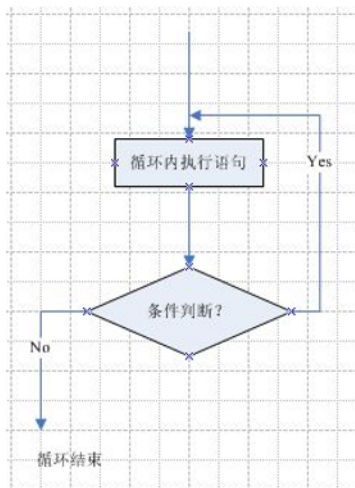
3.3.3 do while 语句

do while 语句是这样的：

```
do{
    <语句>;
} while(<条件>); //（注意这里的分号）
```

它的意思是先执行语句，再判断条件，执行语句，再判断条件……直到条件不满足时跳出。又引用一次它的程序框图





§3.4 关于循环语句的说明

3.4.1 一定记得敲大括号

一定记得敲大括号，一定记得敲大括号，一定记得敲大括号（重要的事情说三遍）

3.4.2 关于循环套用

emmm，不如直接举例子吧：

目标：编程求出 2 到 20 的因数，并打印出来
实现源程序如下：



```

#include<iostream>
using namespace std;
main(){
    int a=2;
    //在下面的语句中有多个';', 所以需要{}来辅助
    for(;a<=20;a++){
        cout<<a<<"的因数有: ";
        //下行和下行有且只有一个分号, 故可以不写{}
        for(int b=1;b<=a;b++)
            if(a%b==0)
                cout<<b<<' ';
        cout<<endl;
    }
    return 0;
}

```

3.4.3 三种循环的选取

建议在使用循环时优先使用 **for** 循环 (真的好用), 因为 **for** 循环可以将语句和变量分离开来, 使得程序更加易读。虽然事实上, 三种循环语句是可以互相转换的 (顾老师的某次作业就是三种循环语句的转换, 故具体如何转换我是不会告诉你的 hiahiahia)



第四章 数组与字符串的输入与输出

§4.1 数组与字符串

4.1.1 分类

浮点数组 (float)

双精度数组 (double)

整数数组 (int)

字符串 (char)

4.1.2 两者相同点

它们的生成方式是相似的，都长这样：

< 变量类型 > < 变量名称 > [数组或字符串长度];

且都可以在生成时直接定义是多少，并且用这种用法时可以不写长度哦

< 变量类型 > < 变量名称 > [数组或字符串长度（可不写）] = {变量组 (用逗号连接)};

ps4.1: 数组与字符串中的第一个元素的编号为 0，譬如数组 a 的第一个元素为 a[0]。所以数组的长度是不能取到的。比如如下语句 4.1 就是错误的。

语句 4.1:

```
int a[5];  
a[5]=3;
```

ps4.2: 数组与字符串中的元素是连续存储的，因而想要在数组中的某个位置加入一个元素是一件比较困难的事情，需要用如下的语句完成。例如在原来有 6 个数的数组 a 中的第 4 个位置插入 3，可以使用语句 4.2。



语句 4.2:

```
int a[7];  
for(int i=3;i<6;i++) a[i+1]=a[i];  
a[3]=3;
```

4.1.3 两者不同点

字符串可以直接 `cout` 名称从而打印出来，而数组必须将元素一个一个输出。基于这个原因，字符串需要以阿斯克码 0 结尾。因而一个长度为 3 的字符串在声明时至少要声明长度为 4。而且切记在声明或更改字符串时要将结尾改为 0。

4.1.4 其他注意事项

二维数组

事实上，所谓二维数组，就是一个数组组成的集合所构成的数据结构，如果把集合用 `{}` 表示，那二维数组就是这样的 `{{000}}`。

多维以此类推。

关于排序

顾老师讲述了冒泡排序的基本做法，在这里我补充一个比较牛皮的（我认为）。因为这种做法的复杂度是 $O(n \log n)$ ，而冒泡排序则是 $O(n^2)$



语句 4.3:

```
void quicksort(int l, int r){
    if (l >= r) return;
    int mid = partition(l, r);
    quicksort(l, mid - 1);
    quicksort(mid + 1, r);
}

int partition(int l, int r){
    int x = a[l]; //轴值
    while (l < r){
        while (l < r && a[r] >= x)
            r--;
        if (l < r)
            a[l++] = a[r];
        while (l < r && a[l] <= x)
            l++;
        if (l < r)
            a[r--] = a[l];
    }
    a[l] = x; //此时l=r, 处于中间位置
    return l;
}
```

执行 `quicksort(start,end)` 之后, 数组 `a` 中的从 `start` 位到 `end` 位的大小便有了定义。

事实上, `c++` 自带了排序功能, 只需要在源文件中多包含一个 `algorithm`, 然后定义了先后顺序, 就可以进行排序, 如程序:



```

#include<iostream>
#include<algorithm>
using namespace std;
bool cmp1(int a,int b){
    return a>b;
}
bool cmp2(int a,int b){
    return b>a;
}
int main(){
    int a[5]={3,1,4,2,5};
    for(int i=0;i<5;i++)
        cout<<a[i]<<' '; //排序前
    cout<<endl;
    sort(a,a+5,cmp1); //倒叙排序
    for(int i=0;i<5;i++)
        cout<<a[i]<<' ';
    cout<<endl;
    sort(a,a+5,cmp2); //正叙排序
    for(int i=0;i<5;i++)
        cout<<a[i]<<' ';
    return 0;
}

```

的执行结果如下:

```

3 1 4 2 5
5 4 3 2 1
1 2 3 4 5

```

cmp 函数是一个 **bool** 函数，其实它叫什么都可以。

a 和 **b** 可以任意取名。若前面元素（比如 **a**）应当在后面元素的前面（比如 **b**）则输出 1，否则输出 0。换句话说，只要保证每两个相邻元素按照顺序输入函数（比如 **cmp1** 或 **cmp2**）的结果都是 1，则排序完成。



关于字符串

字符串有一个专门的类 `string` 类来定义与管理（当然我们基础的 `cpp` 绝对是用不到的），但是对于 `char` 类，依然有 `string` 库中的函数可以尽情享受，具体可以参见百度百科的 `string` 库中的函数。

4.1.5 补充：数组

数组的本质是一串连续的内存空间（可以想象数组是从头到尾一字排开的线性结构）。数组名就是这一片内存空间的首地址。由于其连续存储的特点，数组可以实现随机存取，即只需要一个变量（即数组下标）就可以根据首地址直接定位到该变量的存储位置。多维数组的本质仍然是一串连续的内存空间。二维数组名可以看成是一个二维指针。维数组中存的不是变量的值，而是每一个行向量的指针。根据每一个行向量的首地址进而可以确定这一行任意一个变量的存储位置。三维数组中存的是二维指针，以此类推。请注意：不管数组是多少维，它在计算机当中的存储结构永远是线性结构。



第五章 函数的声明、定义和调用

§5.1 模块与函数的概念

我们在写一个程序的时候，起码先是会在脑海中有一个做事情步骤，我们叫它流程图。这时，你已经将一个相对来说比较大的问题分解成了许许多多的小问题，这就是模块化的概念。

模块指完成一个特定功能的语句序列。模块的基本特征时仅有一个入口和一个出口；开始执行模块时，只能从其入口处开始执行，执行完模块后，必须从模块的出口转而执行其他模块（下一个步骤）。

在 C++ 中，函数是构成程序的基本模块，每个函数都具有自己相对独立的功能。C++ 的函数有三种：程序开始执行首先调用的主函数（`main ()` 函数）、系统自带的库函数（需要引用库函数所在的文件名，即在代码开始处的 `#include<...>`）和用户自己写的函数。

简单来讲，函数就是在 C++ 中用于解决那一个个小问题的方法。

合理编写用户自定义函数，便于阅读和调试程序（最大的好处就是方便找 bug），是结构化程序设计方法的主要内容之一。

补充：每一对花括号 `{ }` 都是一个 `scope`，所有在花括号内定义的变量都是局部变量，出了花括号之后就会消失。

§5.2 函数的定义与调用

5.2.1 函数的定义

函数必须要先定义才能使用，定义格式如下：

```
<函数值类型标识符> 函数名 (<形式参数表>) {  
    <函数体>  
}
```



突然蹦出来这么多名词，可能你一下子有点接受不来。不过看下面这个例子，相信你能对函数的定义方式有初步的了解：

```
//example5-1 求和函数
//从begin的值逐一加到end的值，简单起见我们保证begin<=end
void sum(int begin, int end){
    int i, sum=0;
    for(i=begin; i<=end; i++){
        sum+=i;
    }
    cout<<"从"<<begin<<"到"<<end<<"的和是："<<sum<<endl;
}
```

这是一个非常简单的函数，如果你已经能够看懂了，请跳过本段。首先 **void** 对应上面的 <函数值类型标识符>，这个类型标识符指的是函数的返回类型。一般来讲，函数是要有返回值的，而这个函数值类型标识符就指的是函数的返回值的类型。与变量定义类似，**int** 表示是整数，**double** 表示是双精度浮点数，**char** 是字符类型，特别的，**void** 指的是没有返回类型，如上面的函数不需要返回任何东西。当然，我们可以把它改成需要返回一个整数的（见 **example5.2.2**）。之后再说函数名，顾名思义，就是你给函数起的名字，需要调用它的时候需要叫一下它的名字。形式参数表，就是函数将要用到的初始数据，具体内容见 5.3 节函数的参数。函数体，就是完成指定功能的语句序列，也就是你想在函数中执行的语句，调用函数之后将会执行这些语句。

```
//example5-2 求和函数（从begin的值逐一加到end的值并返回和值）
int sum(int begin, int end){
    int i, sum=0;
    for(i=begin; i<=end; i++){
        sum+=i;
    }
    return sum;
}
```

函数的返回值通过语句“**return** 表达式;”来实现。因为函数只能有一个返回值，所以默认当你返回一个值的时候，就宣告了函数本次执行的终结（**main** 函数中返回 0 表示程序顺利地运行到了 **return 0**; 这一行），在调用期间所分配



的变量全部都会被释放（也就是说函数里面东西没了，求和函数中你想在函数外使用 **sum** 这个变量的话，它也不再是函数中的这个 **sum** 了）。

注意：C 语言不允许函数的嵌套定义。

5.2.2 函数的调用

我们已经会定义一个函数了！不过，我们的函数不是用来造着玩的，我们得用它。

除了主函数，任何函数不能自发运行，必须要通过调用来启动运行。

调用的方式很简单，在主函数中叫一下它的名字就行了。如下：

```
//example5-3 求阶乘函数
int fac(int n){
    int result=1;
    if(n<0) return -1;//表示不合法
    else if(n==0) return 1;//特判结果
    while(n>1){
        result*=n;
        n--;
    }
    return result;
}
int main(){
    int n;
    cin>>n;
    cout<<n<<"!="<<fac(n)<<endl;
}
```

好了，我相信你已经懂了，不过还有几点需要注意一下：

函数调用的时候传入实际参数（与形式参数对应，后面会讲）与函数定义时的形式参数要绝对匹配！就是说，参数的类型、参数的个数、参数的次序必须完全对应。如果不对应，你可以试一下会发生什么邪恶的事情。

函数的返回类型表示他的返回值的类型，比如返回了 **int** 那么你就可以把它当做一个 **int** 的整数来对待，如果你强行用 **double facn=fac (n)** 的话，系统会自动给 **facn** 后面补 0，但如果函数 **f ()** 返回的是 **double** 类型，那么 **int**



`value_of_f=f ()` 就可能得不到你想要的结果（系统强制类型转换成了 `int` 给你赋进去了）。当然，如果是 `void` 类型那么就不能用来赋值或运算。

调用一个函数的时候必须在调用语句的前方有函数的声明，不然编译器会报错（提示如下语句：`[Error] 'xxx' was not declared in this scope`）。有关函数声明的问题我们会在 5.4 函数原型部分细讲；现在你记住必须得把函数写在调用它的语句之前就行了。

函数内部也可以调用其他函数。

§5.3 函数的参数

终于到参数了。还记得函数定义的时候剩下的遗留问题吗：形式参数表到底是干啥的？

首先，我们需要了解，函数的参数有两种，一种叫形式参数，另一种叫实际参数。在定义函数时，形式参数在函数名后面的一对圆括号内。其实，形式参数就是你在给函数传值的时候用于接收传入值的变量名称，并且在函数中，对这个值的操作都会通过这个变量名。相对来讲，实际参数就是指真正从外部传进函数的那个值。

5.3.1 形式参数的作用

表示将从主函数中接收到哪些类型的信息。

例如：`double f(int x,double y,char z)` 就表示 `f ()` 这个函数在被调用时需要传入三个值，第一个是整数类型的，第二个是双精度浮点类型的，第三个是字符类型的，传进的三个值分别被赋进 `x`，`y`，`z`，这里，`x`，`y`，`z` 就是形式参数。

形式参数只可以再函数体内部使用，可以输入、输出、赋值和参与运算，但一旦离开函数体，形式参数就离开了它的作用域，不再起作用了。

5.3.2 函数间的参数传递

调用函数时需要进行参数传递，参数传递的方向是由实参传给形参；传递过程发生的事情是先计算实参表达式的值，再将该值传递给对应的形参。

再次强调，实参和形参必须完全匹配！（数据类型、个数、顺序）

补充：C++ 函数参数传递的两种方式

(1) 用实参和形参、返回值进行传递。这种方法的局限性在于函数只能有一个返回值



(2) 用指针来传参。可以把指针作为函数的参数或者返回值，这样一次可以传一个数组

(3) 用引用来传参。且看下例：这个函数的功能是删除字符串 **a** 中的某个字符。返回值用来显示操作是否成功。并且把被删字符的值传给 **it**

```
bool remove(char a[ ], char &it, int curr){
    if (!strlen(a)){
        cout<<"表空不能删除！"<<endl;
        return false;
    }
    if (curr < 0 || curr >= strlen(a)){
        cout<<"位置不合法不能删除！"<<endl;
        return false;
    }
    it = a[curr];
    for (int i = curr; i < strlen(a)-1; i++) a[i] = a[i+1];
    return true;
}
```

以上函数既要实现返回操作是否成功的信息，又要返回被删除的字符。仅仅依靠 **return** 显然不够用。但是引用可以帮我们做到这一点：传一个 **char** 类型的引用进去，在函数体内利用引用可以修改 **it** 所指代变量的值，这样就实现了参数传递。

5.3.3 初始化形参

在定义形参时，我们被允许为形参指定默认的值，即初始化。具体初始化的方式如下：

```
//example5-4 求圆形面积函数
#define PI 3.14159//宏定义常数
//上一行或写成const定义常数的形式：const int PI=3.14159
double Area(int r=0){
    return PI*r*r;
}
```

如上，函数 **Area** 的形式参数中 “**r=0**” 就是在为形参 **r** 赋初始值。



初始化形参的操作既可以在函数的定义中执行，也可以在函数原型（见 5.4）中执行，不过通常我们将初始化操作放在该函数在源程序中第一次出现的地方。在调用函数的时候如果传入了实参，那么实参将会覆盖初始化的值，即形参会使用实参的值。如果没有传入实参的值，那么形参就默认使用初始化的值。

下面看一个例题：

Example5-5 以下哪些函数原型或定义方法是正确的？

A. void f(int x=1, int y=2, int z);

B. void f(int x, int y=2, int z=3);

C. void f(int x=1, int y, int z=3);

D. void f(int x, int y, int z);

E. void f(int x=1, int y=2, int z=3);

答案：BDE

解析：一个函数的形参全部没有初始化是允许的，全部进行初始化也是允许的。不允许的情况出现在当不是所有形参都被初始化的时候，初始化的顺序没有按照从后到前的顺序。例如 A 选项，**xy** 均被初始化，**z** 未被初始化，当函数的调用者传入了三个参数还好，如果只传入了一个参数，那么根据接受顺序，**x** 将接受传入的值，**y** 会被初始化成 2，而 **z** 并没有接受任何值也没有被初始化，这是不合法的。

§5.4 函数原型与调用方式

5.4.1 函数原型

C++ 中，函数与变量类似，在使用前应事先说明它的存在。函数的定义是一种说明方式，所以在以前的程序中，函数定义必须放在需要调用该函数的程序的前方。另外，还有一种说明的方式，就是函数原型，这种说明也被称为函数声明。函数原型的一般格式为：

<函数返回值的类型标识符> 函数名 (<形式参数表>);

注意行尾的分号不可少，不然编译器会报错。你可以看出来，其实就是把函数定义的左大括号前的部分拿出来后面再加上一个分号就是了（没错，就这么简单）。如果你已经在程序开头声明过函数的存在了，那么函数的定义放在哪里就无关紧要了；为了体现程序的简洁性和结构化，你可以把函数定义放在 **main** 后。在函数原型中，参数的名字可以省略，但必须指定清除参数的类型。



```
//example5-6函数原型
#include<iostream>
using namespace std;
bool isEqual(int,int);
int main(){
    int a,b;
    cin>>a>>b;
    cout<<isBig(a,b);
    return 0;
}
bool isEqual(int x,int y){
    return x==y;
}
```

5.4.2 函数的调用方式

根据传入参数的方式不同，函数的调用分为三种：值调用、引用调用和地址调用。地址调用涉及指针的相关知识，我们放在指针中再讲。

第一种，值调用。

看到现在你之前见过的所有传参方式都是值调用。实参直接并且也仅仅通过赋值的方式将值交给形参就是值调用，应该不用我说太多了吧。注意，在函数体内对形参的任何修改都不会影响实参的值，所以多数情况下你需要 **return** 一个你运行的结果。

第二种，引用调用。

所谓引用，它其实是一个新的变量类型。格式为：

```
<数据类型> &引用名=目标名；
//example5-7引用
int a;
int &b=a;
```

通过引用，我们可以认为引用的对象是被引用的对象的别名。在上述例子中，修改 **b** 的值的同时，**a** 的值也会跟着改变。所以，我们在利用引用传值的时候，函数内部对形参的修改也会影响函数外部的变量。下面是一个很典型的例子：



//example5-8失败的交换两个变量的值的函数

```
void swap(int x,int y){  
    int tmp;  
    tmp=x;  
    x=y;  
    y=x;  
}
```

//example5-9成功的交换两个变量的值的函数

```
void swap(int &x,int &y){  
    int tmp;  
    tmp=x;  
    x=y;  
    y=x;  
}
```

只有一处变化，就是在传参的时候使用引用调用还是值调用，就会引起结果的变化。

§5.5 局部变量与全局变量

首先，我们来一起了解一下什么是作用域。作用域是指程序中使一个标识符有意义的一段区域，该标识符在此段区域内是可被访问的。也就是说，作用域表示了一个变量能够存活区域，离开这个区域，某个变量就“死”了，失效了，没意义了，不能被访问了。

根据作用域的不同，我们可以将变量分为局部变量和全局变量。

局部变量是在函数或者分程序中声明的变量，只能在该函数或分程序的范围内生效。全局变量是指在所有函数之外，可为源程序中所有位于该全局变量声明之后的所有函数共同使用。

也许你还不是很清楚。记不记得函数定义的时候是要写一对大括号的？那个大括号就代表着函数中变量的作用域。更进一步地说，如果是在某个大括号里面声明或定义的变量，就意味着离开这个大括号之后，那个变量就离开了它的作用域，就失效了。那么，不位于任何大括号内的变量定义或声明，就是全局变量了。（是不是很容易辨别？）

注意：如果程序中全局变量和局部变量的名称相同，那么位于大括号内的局部变量会“暂时”覆盖全局变量的值，就是说全局变量在这个作用域内隐化



了，离开这大括号后，全局变量会再次显化。

如果你就是想要在一个大括号里让全局变量和局部变量名字相同，然而需要同时使用，那么也有方法，采用一元作用域符可以表示下个变量属于哪个作用域范围。例如：

```
//example5-10
#include<iostream>
using namespace std;
int x=0;
int main(){
    int x=5;
    cout<<" 全局变量x=" <<::x<<endl;
    cout<<" 局部变量x=" <<x<<endl;
    return 0;
}
```

如上，全局变量会输出 0，局部变量会输出 5。因为全局变量不属于任何一个作用域，所以“::”前什么都不用写。不过为了使程序阅读者更加清楚每个变量的含义，还是不推荐将全局变量和局部变量定义成同名。

§5.6 补充内容

5.6.1 关于 main

这里介绍一些关于 **main** 函数的常识。

首先，一个 C++ 程序中只能有一个全局 **main** 函数。因为程序总是从 **main** 函数开始执行，且总是默认从 **main** 函数的 **return** 语句或结尾处结束运行。

注意，C/C++ 中从来没有定义过 **void main()**，也就是说，**main** 函数必须有返回值。（可能你用这种写法编译仍然通过了，不过因为这种写法十分不正规，这里还是不推荐了）

可能你在哪里注意到这样的写法：

```
main () {
... ..
}
```

这是 C89 的一种省略写法，这样写默认会补上 **int** 的返回类型并返回 0。

C89/C99/C11 标准文档中只提供了两种 **main** 函数的写法：



```
int main(void) { /* ... */ } // 整数类型主函数(无类型)
int main(int argc, char *argv[]) { /* ... */ }
// 整数类型主函数(整数类型 统计参数个数, 字符类型*数组指针至字符[])
```

有关第二种写法中参数的具体意义和用法可自行查看书 P99 相关内容。(悄悄告诉你 7.4.3 节也会有所介绍哦)

也就是说, 其他的写法都是不符合标准的, 有的是历史遗留问题, 有的是编译器扩展, 更有的不知从何而来。当然对于带参数的写法, 其参数名可自定义。另外对于除标准提供的以外的写法, 不同的编译器有不同的处理策略, 有的可能编译不过, 有的可能报警告, 但一个“正经”的编译器是不会对标准提供的形式产生质疑的。

至于最常见的写法 `int main()`, 则是没有指明是否传入参数, 一般来讲可以认为是没有参数。

5.6.2 常用库函数

以下内容读者大致浏览一番即可(不看也罢, 后期需要使用了直接上网搜索或来此处查找)。下述内容只包含本课程中可能会用到的库函数, 欲了解更多可以查看原文网址。

C++ 标准库的内容分为 10 类, 分别是:

C1. 标准库中与语言支持功能相关的头文件

`<cstdlib>`

提供支持程序启动和终止的宏和函数。这个头文件还声明了许多其他杂项函数, 例如搜索和排序函数, 从字符串转换为数值等函数。它与对应的标准 C 头文件 `stdlib.h` 不同, 定义了 `abort(void)`。 `abort()` 函数还有额外的功能, 它不为静态或自动对象调用析构函数, 也不调用传给 `atexit()` 函数的函数。它还定义了 `exit()` 函数的额外功能, 可以释放静态对象, 以注册的逆序调用用 `atexit()` 注册的函数。清除并关闭所有打开的 C 流, 把控制权返回给主机环境。

`<new>`

支持动态内存分配

C2. 支持流输入/输出的头文件

`<iostream>`

支持标准流 `cin`、`cout`、`cerr` 和 `clog` 的输入和输出, 它还支持多字节字符标准流 `wcin`、`wcout`、`wcerr` 和 `wclog`。

`<iomanip>`



提供操纵程序，允许改变流的状态，从而改变输出的格式。

<ios>

定义 **iostream** 的基类

<istream>

为管理输出流缓存区的输入定义模板类

<ostream>

为管理输出流缓存区的输出定义模板类

<fstream>

支持文件的流输入输出

<cstdio>

为标准流提供 C 样式的输入和输出

C3. 与诊断功能相关的头文件

C4. 定义工具函数的头文件

<functional>

定义了许多函数对象类型和支持函数对象的功能，函数对象是支持 **operator()** 函数调用运算符的任意对象

<ctime>

支持系统时钟函数

C5. 支持字符串处理的头文件

<string>

为字符串类型提供支持和定义，包括单字节字符串 (由 **char** 组成) 的 **string** 和多字节字符串 (由 **wchar_t** 组成)

<cstring>

为处理非空字节序列和内存块提供函数。这不同于对应的标准 C 库头文件，几个 C 样式字符串的一般 C 库函数被返回值为 **const** 和非 **const** 的函数对替代了

<cstdlib>

为把单字节字符串转换为数值、在多字节字符和多字节字符串之间转换提供函数

C6. 定义容器类的模板的头文件

<vector>

定义 **vector** 序列模板，这是一个大小可以重新设置的数组类型，比普通数组更安全、更灵活

<list>



定义 **list** 序列模板，这是一个序列的链表，常常在任意位置插入和删除元素

<deque>

定义 **deque** 序列模板，支持在开始和结尾的高效插入和删除操作

<queue>

为队列 (先进先出) 数据结构定义序列适配器 **queue** 和 **priority_queue**

<stack>

为堆栈 (后进先出) 数据结构定义序列适配器 **stack**

<map>

map 是一个关联容器类型，允许根据键值是唯一的，且按照升序存储。

multimap 类似于 **map**，但键不是唯一的。

<set>

set 是一个关联容器类型，用于以升序方式存储唯一值。**multiset** 类似于 **set**，但是值不必是唯一的。

<bitset>

为固定长度的位序列定义 **bitset** 模板，它可以看作固定长度的紧凑型 **bool** 数组

C7. 支持迭代器的头文件

<iterator>

给迭代器提供定义和支持

C8. 有关算法的头文件

<algorithm>

提供一组基于算法的函数，包括置换、排序、合并和搜索

<cstdlib>

声明 C 标准库函数 **bsearch()** 和 **qsort()**，进行搜索和排序

C9. 有关数值操作的头文件

<cmath>

这是 C 数学库，其中还附加了重载函数，以支持 C++ 约定

<cstdlib>

提供的函数可以提取整数的绝对值，对整数进行取余数操作

C10. 有关本地化的头文件

<locale>

提供的本地化包括字符类别、排序序列以及货币和日期表示。

来源：CSDN



原文：<https://blog.csdn.net/sxhelijian/article/details/7552499>

5.6.3 变量的存储类别

在 C++ 中，由于变量存在的时间不同，可将其分为四类，即自动变量、静态变量、寄存器变量、外部变量。

自动变量的特点是在程序运行到该变量的作用域才为其分配存储空间。关键字为 **auto**。前面所说的局部变量全部都是自动变量。一般在函数体或程序块中声明的变量，其存储类别的缺省形式是自动变量，当然你可以写全称来声明如下：

```
auto int a,b,c=0;
```

静态变量的特点是程序开始运行前就为其分配了相应的存储空间。在程序的整个运行期间静态变量始终位于这些空间中，也就是说，静态变量的生存期就是程序的运行期。所以，当你能在某些函数中保留某个变量的值，以便下次再调用的时候延续使用，你就可以考虑一下使用静态本地变量。静态变量的关键字为 **static**，具体使用方法如下：

```
static int x=0;
```

寄存器变量并不存储在内存存储器中，而是在寄存器中，所以访问起来会很快。笔者并未使用过寄存器变量，详细用法可百度。声明样例如下：

```
register int i,j;
```

外部变量是指使用 **extern** 说明的某个变量。也就是说在某个函数中使用 **extern** 说明的变量就是外部变量，而非一个同名的局部变量。使用方法如下：

```
extern int t;
```

5.6.4 变量在内存中的布局

编译系统为一个 C++ 程序的运行分配了 5 个内存区域，如表格所示：

堆区（动态数据）
栈区（函数局部数据）
main () 函数局部数据
全局数据区（全局、静态变量、常量）
代码区（程序代码）



§5.7 常用函数附录

5.7.1 取系统时间

运行此程序，你会明白各个变量代表的意义。

```
#include<ctime>
#include<iostream>
using namespace std;
int main(){
    time_t tt=time(NULL);
    tm* t=localtime(&tt);
    cout<<t->tm_year+1900<<endl;
    cout<<t->tm_mon+1<<endl;
    cout<<t->tm_mday<<endl;
    cout<<t->tm_hour<<endl;
    cout<<t->tm_min<<endl;
    cout<<t->tm_sec<<endl;
    return 0;
}
```

5.7.2 rand 函数

这里介绍一下利用 **rand** 函数来获得随机数。

如果你想要调用 **rand** 函数，那么需要在程序一开始 **include** 一下 **<cstdlib>** 标准函数库。

具体使用格式如下：



```
//rand() 示例
#include<iostream>
#include<cstdlib>
using namespace std;
int main(){
    for(int i=1;i<=20;i++){
        cout<<rand()<<'\t';
        if(i%5==0) cout<<'\n';
    }
    return 0;
}
```

你可能会发现，每次调用 **rand** 函数生成的随机数并不“随机”，它是由事先指定的一系列数依次输出而来的。调用 **rand** 函数会返回一个 **[0,RAND_MAX]** 范围内的随机整数，在 C 语言标准库 **<stdlib.h>** 中定义了名为 **RAND_MAX** 的宏，C11 标准并未规定它的具体数值，但规定了不应小于 **32767**。

如果你想要生成一个“真正”的随机数列，你可以利用 **srand** 函数 () 加入时间种子。

```
//srand() 示例
#include<iostream>
#include<cstdlib>
#include<ctime>//time(NULL) 需要
using namespace std;
int main(){
    srand((unsigned)time(NULL));
    for(int i=1;i<=20;i++){
        cout<<rand()<<'\t';
        if(i%5==0) cout<<'\n';
    }
    return 0;
}
```

srand 函数的原型为 **void srand(unsigned seed)**; 其中 **seed** 就是 **srand()** 函数的种子，用来初始化 **srand()** 的起始值。系统在调用 **rand** 函数之前都会自动调用 **srand** 函数，如果没有人为提前调用过 **srand** 函数，那么系统将默认将 **1**



作为 `seed` 的初始值，由于给定了一个定值，每次调用 `rand()` 产生的随机数列都是一样的。

现在可以生成“随机”整数了！如果想要得到某个区间 $[a,b]$ 内的随机整数，那么 `rand()%(b-a)+a` 就可以得到；想要生成 $[a,b]$ 内的随机浮点数可以用 `(double)rand()/RAND_MAX*(b-a)+a` 得到。



第六章 指针的声明和使用

§6.1 指针的概念

6.1.1 地址

程序在运行过程中，需要一定的内存来存放程序本身、函数、变量、常数、数组和对象等。计算机用于存放数据或信息的内存是有编号的，不同内存的编号不同，这个编号就是该内存单元的地址，可以用一个没有符号的整数来表示。

凡是在内存中存储的信息都应有一个地址，一般用其所占内存空间的第一个存储单元的地址表示，也就是首地址。

6.1.2 指针

指针是一个变量，因此它具有变量的几个要素：名称、类型、值。指针的名称与普通变量的命名规则相同；指针的类型取决于指针所指向的变量的数据类型，如用于存放 `char` 类型变量的指针就是 `char` 类型指针；指针的值就是它所指向的变量在内存中所处的地址的值。

指针可以指向任何类型的数据、数组、函数、对象，甚至指向指针。

§6.2 指针的声明及运算

6.2.1 指针变量的声明

指针变量的声明语句格式：

<类型指示符> *指针变量名



有关 * 运算符在指针的运算中会进一步解释，现在可以先记住指针是这么写的。

看几个例子：

```
char *str;//声明了指向字符类型的指针str
char *s="Hello!" //声明了指向常数字符“Hello!”的指针s
int *i//声明了指向整数类型的指针i
```

在 C 中，预处理器宏 NULL 被定义为空指针常量，可被转换为 void* 类型（指向 void 的指针）。在 C++ 中继承了 C 的 NULL 宏定义，但空指针常量被优先解释为整型 0。

6.2.2 * 和 & 运算符

首先，取地址运算符 &：使用 & 运算符可以取出一个变量的地址（甚至是指针变量的地址）。

然后，指针运算符 *：使用 * 运算符可以取出一个地址指向的内容，“* 一个指针”就代表着这个指针所指向的对象。

```
//example6-1 指针变量的运算
#include<iostream>
using namespace std;
int main(){
    int x=1;
    int *px;
    px=&x;
    cout<<"x="<<x<<endl<<"px="<<px<<endl<<"*px="<<*px<<endl;
    *px+=1;
    cout<<"*px adds 1\n";
    cout<<"x="<<x<<endl<<"px="<<px<<endl<<"*px="<<*px<<endl;
    x++;
    cout<<"x adds 1 again\n";
    cout<<"x="<<x<<endl<<"px="<<px<<endl<<"*px="<<*px<<endl;
    return 0;
}
```



6.2.3 指针的赋值运算

将一个指针的值赋给另一个指针，结果是两个指针指向同一个地址。

```
//example6-2 指针赋值运算
#include<iostream>
using namespace std;
int main(){
    int x=1;
    int *px1,*px2;
    px1=&x;
    px2=px1;// 指针赋值运算
    cout<<" x=" <<x<<" \npx1=" <<px1<<" \npx2=" <<px2;
    cout<<" \n*px1=" <<*px1<<" \n*px2=" <<*px2;
    return 0;
}
```

6.2.4 指针的关系运算

在关系表达式中允许指针的比较运算，但一般情况下不同类型的指针之间的比较是没有意义的。

```
//example6-3 指针比较运算
#include <iostream>
using namespace std;
int main(){
    float a,b,*i,*j;
    i=&a;
    j=&b;
    cout<<"变量a的地址: "<<i<<endl;
    cout<<"变量b的地址: "<<j<<endl;
    cout<<"i>j的结果: "<<(i>j);
    return 0;
}
```



6.2.5 指针的算术运算

指针变量只有加法和减法两种算术运算。

- (1) 自增 ++、自减-运算；
- (2) 加、减整型数据；
- (3) 指向同一个数组的不同数组元素的指针之间的减法。

对指针变量进行乘法、除法、两个指针相加、指针类型与其他类型的加减运算都毫无意义。

```
//example6-4 指针的算术运算
#include <iostream>
using namespace std;
int main(){
    int a,b,*ip,*jp;
    int p;
    ip=&a;
    jp=&b;
    p=ip-jp;
    cout<<"变量a的地址："<<ip<<endl;
    cout<<"变量b的地址："<<jp<<endl;
    cout<<"ip-jp="<<p<<endl;
    cout<<"ip>jp的结果："<<(ip>jp)<<endl;
    cout<<"ip==jp的结果："<<(ip==jp)<<endl;
    jp=jp+p;
    cout<<"ip==jp的结果："<<(ip==jp)<<endl;
    return 0;
}
```

需要注意的是，指针的加减法并不是让地址加了数值 1，而是让指针后移了一个单位。

§6.3 指针与数组

6.3.1 数组

在 C++ 中，指针与数组关系十分密切。



C++ 规定单写数组名表示数组的首地址，例如：

```
int a[10];  
int *pa=a;//也就是int *pa=&a[0];
```

实际上数组名本身就是一个指针所指地址不变的指针，即指针常量。而数组名所指向的地址就是数组首元素的地址，在上例中，`a` 就相当于 `&a[0]`。

由于数组中元素在内存中是连续排列存放的，所以可以通过指针来实现对数组中元素的使用。例如，在上例中：`a[i]`，`*(a+i)`，`*(pa+i)`，`pa[i]` 均等价。其中，写成 `a[i]` 或 `pa[i]` 的形式编译器会先翻译成 `*(a+i)` 的形式，所以，在运行时 `*(a+i)` 的形式会更加高效快捷。这里对指针的加法运算服从 6.2.5 节指针的算术运算，即指针增 1 是让指针后移了一个单位，而不是指向地址的下一个。

6.3.2 指针与字符数组

字符串也是一种特殊的数组，它全部由字符组成。所以对字符串的操作也可以用指针来实现。字符数组与字符串的区别在于字符串的最后一个元素必须是 `'\0'`，字符数组则没有此限制。

我们来通过一个例子来理解指针对字符串的操作和指针对数组的操作无异。

```
//example6-5 整数数组的逆序  
#include<iostream>  
#include<cstdlib>  
using namespace std;  
void nixu(int *a,int n){  
    int t,*i,*j;  
    i=a;  
    j=a+n-1;  
    for(;i<j;i++,j--){  
        t=*i;  
        *i=*j;  
        *j=t;  
    }  
}
```




```

int main(){
    int i,a[10];
    for(i=0;i<10;i++) a[i]=rand()%10;
    cout<<"逆序前:";
    for (i=0;i<10;i++)  cout<<a[i]<<" ";
    cout<<endl;
    nixu(a,10);
    cout<<"逆序后:";
    for (i=0;i<10;i++)  cout<<a[i]<<" ";
    return 0;
}
//example6-6字符串的逆序
#include<iostream>
#include<cstdlib>
#include<cstring>
using namespace std;
void mystrev_1(char s[]){
    int n=strlen(s);
    char tmp;
    for(int m=0;m<n/2;m++){
        tmp=s[m];
        s[m]=s[n-m-1];
        s[n-m-1]=tmp;
    }
}
void mystrev_2(char *s){
    int n=strlen(s);
    char tmp;
    for(int m=0;m<n/2;m++){
        tmp=s[m];
        s[m]=s[n-m-1];
        s[n-m-1]=tmp;
    }
}
}

```



```

int main(){
    char s1[10],s2[10],s3[10];
    for(int i=0;i<9;i++) s1[i]=s2[i]=s3[i]=rand()%(126-32)+32;
    s1[9]=s2[9]=s3[9]='\0';
    cout<<"反转前的字符为："<<s1<<endl;
    mystrrrev_1(s1);
    mystrrrev_2(s2);
    strrev(s3);
    cout<<"自定义函数mystrrrev_1反转后的字符为："<<s1<<endl;
    cout<<"自定义函数mystrrrev_2反转后的字符为："<<s2<<endl;
    cout<<"系统函数反转后的字符为："<<s3;
    return 0;
}

```

其中，`strlen` 函数返回到‘\0’前的所有字符数。

§6.4 动态存储分配

程序中我们使用的变量、数组、函数等需要事先声明，声明的时候编译器会开辟相应所需的内存空间，因此数据所占内存是确定的（事先开辟好了）。这种方法叫做静态存储分配。这么做有它的缺点，就是程序无法根据具体运行情况灵活调整内存的分配，造成内存分配过大的浪费或是内存不足导致的数据缺失。

动态存储分配就是用来克服这种不便的。动态存储分配需要使用两个新的运算符 `new` 和 `delete`。

运算符 `new` 用来申请所需内存，使用格式如下：

< 指针 >=`new` < 类型 >; 或 < 指针 >=`new` < 类型 >{初值};

这样你就申请了一个变量的内存，并将其地址存放在指针中了。如果没有申请成功，指针的值就会被赋为 `NULL`，即空指针，表示内存已满，申请失败。初值表示为申请的变量赋初值。

申请数组内存的格式如下：

< 指针 >=`new` < 类型 >[< 元素个数 >];

这样你就申请了指定个数大小的一个一维数组。

俗话说：“有借有还，再借不难。”我们会借东西了（申请），就必须会还东西（释放）。



运算符 **delete** 用于释放已申请的内存，使用格式如下：

delete < 指针 >;

这样就完成了一个变量内存的释放。释放数组内存的格式如下：

delete[]< 指针 >;//在指针名和 **delete** 之间的中括号 **[]** 表示释放的是数组的内存

```
//example6-7利用动态数组求斐波那契数列的前n项
#include <iostream>
using namespace std;
int main(){
    int n;
    cout<<"输入项数：";
    cin>>n;
    int *p =new int[n]; //申请数组空间
    if(p==0||n<=0){//如果没有申请到内存或数据输入有误，则返回
        cout<<"Error!"<<endl;
        return -1;
    }
    p[0]=0;
    p[1]=1;
    cout<<p[0]<<endl;
    cout<<p[1]<<endl;
    for(int i=2; i<n; i++){
        p[i]=p[i-2]+p[i-1];
        cout<<p[i]<<endl;
    }
    delete []p;//释放数组空间
    return 0;
}
```

需要注意的是，一定要确认内存分配成功后再使用！否则可能会造成严重后果。并且在分配完成后，不要变动指针的值，否则释放时会出现错误。采用 **new** 申请的空间不会自动释放，必须通过 **delete** 来释放；也就是说，如果你在某个函数中申请了一些内存，那么必须在这个函数中释放，否则当你退出后，局部变量失效，你就再也找不到这些内存的地址了，而其所对应的内存空间则会无法释放，长此以往将无内存可申请。



§6.5 补充内容

6.5.1 动态申请二维数组

代码并不复杂并且写了注释相信读者可以看懂

```
#include<iostream>
#include<cstdlib>
using namespace std;
int main(){
    //定义行数列数变量M,N
    int M,N;//M行N列
    cout<<"输入行数与列数：";
    cin>>M>>N;
    //动态申请二维数组
    int **a;
    a=new int *[M];
    for(int i=0;i<M;i++) a[i] = new int[N];
    //使用二维数组
    for(int i=0;i<M;i++)
        for(int j=0;j<N;j++)
            a[i][j]=rand();
    for(int i=0;i<M;i++){
        for(int j=0;j<N;j++) cout<<a[i][j]<<'\\t';
        cout<<endl;
    }
    //释放空间
    for(int i=0;i<M;i++) delete[]a[i];
    delete[]a;
    return 0;
}
```

6.5.2 指针和引用

还记得 swap 函数吗？



//example5-8失败的交换两个变量的值的函数

```
void swap(int x,int y){  
    int tmp;  
    tmp=x;  
    x=y;  
    y=x;  
}
```

//example5-9利用引用交换两个变量的值的函数

```
void swap(int &x,int &y){  
    int tmp;  
    tmp=x;  
    x=y;  
    y=x;  
}
```

同样的，我们可以用指针来实现。

//example6-8利用指针交换两个变量的值的函数

```
void swap(int *x,int *y){  
    int tmp;  
    tmp=*x;  
    *x=*y;  
    *y=*x;  
}
```

通过传入两个待交换变量的地址来达到在函数体内部可以访问变量的目的。

在这个例子中，指针与引用都能达到相同的目的，不过他们也有一些不同：指针指向一块内存，引用仅仅是某块内存的别名；指针是可以改变的，而引用只能在定义时被初始化一次；指针可以是空指针，引用不能为空；从内存分配上看，程序为指针分配内存，却不会为引用分配内存。总之，引用就是一个别名，指针是真实的地址。（但是函数参数写成引用的形式会简化函数体中变量调用的书写形式）



6.5.3 指针数组

指针数组就是数组元素全为指向同一种数据类型的指针的数组。使用方式与普通数组的使用方式相同。主要功能为将一系列相关字符串存储在同一个数组中；当然你可以用二维数组存储每个字符，但是你也可以选择存放指针。

6.5.4 指向指针的指针

指针也是变量，也有地址，它的地址当然也能被 & 取出。能够存放指针的地址的变量也是指针，就是“指向指针的指针”。其定义方法为：

<数据类型> **<指针变量名>

例如：

```
//example6-9 指向指针的指针
int a=1;
int *pa,**ppa;
pa=&a;
ppa=&pa;
```

在上例中，ppa 为指向地址 pa 的指针，与指针的运算相同，*ppa 就是 pa，也就是 a 的地址，**ppa 就是 *pa，也就是变量 a。

指向指针的指针有什么用呢？

最主要的就是用于指针数组的处理。指针数组的元素是指向字符类型变量的指针，所以该数组名也就是指针数组首元素的地址，亦即指向指针的指针。

例如：假设 pc[10] 是一个指向字符类型的指针数组，那么用 char **ppc=&pc[i] 就可以取出其中第 i 个元素的地址。

6.5.5 指针与结构体

因为结构体也是存储在内存单元中的，所以它也是有地址的，同样可以用 & 运算符取得结构体类型变量的地址，自然可以声明指向结构体的指针。

声明方法如下：



```
//example6-10指向结构体变量的指针
struct S{
int x;
char c;
};
S ss;
S *pss=&ss;
```

在上例中, `ss` 是一个结构体变量, `pss` 就是一个指向结构体变量 `ss` 的指针, 其值为 `ss` 的地址。如果想要访问 `ss` 的成员变量, 则要用到箭头操作符 “`->`”, 例如:

```
pss->x=10;//通过pss访问ss结构内的x变量并将其赋值为10
```

6.5.6 指针数组的初始化

指针数组在声明的同时可以进行初始化, 例如:

```
char *season[]={ "spring", "summer", "autumn", "winter"};
```

6.5.7 void 和 const 类型的指针

指向 `void` 的指针是通用型指针, 可以指向任何变量。但在求指针的对象变量的内容或进行指针运算时必须强制类型转换成对应的数据类型。例如:

```
//example6-11void类型的指针
int x,y;
void *p;
p=&x;//指向void的指针可以存入任何类型变量的地址
y=*((int*)p);//使用前必须强制类型转换
```

用关键字 `const` 修饰一个指针时, 位置不同有不同的含义: 可以简单记为: `const` 写在 `*` 前表示对象是 `const`, 写在 `*` 后表示指针是 `const`。例如写在 `*` 后:

```
char* const p="abc";//指针p是const
*p="c";//OK, 指针所指的對象不是const
p++;//ERROR, 指针本身是const, 不能运算或复制
```

写在 `*` 前:



```
char* s="abc";  
const char* p=s;//对象 (*p) 是const  
p++;//OK, 指针并不是const  
*p="b";//ERROR, 不能通过p来修改*p  
s="b";//OK, 但是原对象本身可以不是const
```

请读者仔细体会上面的例子，理解 **const** 类型的指针的含义。

体会完了之后，笔者在这里想告诉你，没事别把指针定义成 **const** 或者 **void**，很凌乱。在给一个函数传参数的时候，不希望函数内部修改指针所指向的对象，这时候可以加一个 **const**；**void** 可以用在你不知道这个指针将来会指向哪种类型的变量，但又必须先定义一个的时候。



第七章 函数和指针程序设计

§7.1 指针作为函数的参数

在 6.5.2 小节指针与引用中，我们已经见过了指针作为函数的参数的例子。

指针可以作为函数的参数，当以指针作为函数的参数时，其实在调用过程中发生的是使传入的实参和函数的形参指向同一块地址，这样在函数内部修改变量也会影响到函数外部。

§7.2 指针作为函数返回值

函数返回值也可以是一个地址。在说明返回值为地址的函数时，要使用指针类型说明符。例如：char * strstr(char* string1, char* string2);

```
//example7-1 月份翻译函数
char* month_trans(int n){
    static char* month_name[]={
        "IllegalMonth",
        "January", "February", "March",
        "April", "May", "June",
        "July", "August", "September",
        "October", "November", "December"
    };
    return (n>=1&& n<=12)?month[n]:month[0];
}
```



§7.3 指向函数的指针

函数本身作为一段程序，其代码在内存中也占有一片存储区域。函数的第一个字节的地址，称为首地址。首地址是函数的入口地址，调用函数时就是让程序转移到函数的入口地址处开始执行。与数组类似，函数名单独出现就表示函数的首地址。指向函数的指针就是指向了函数的首地址。说明格式为：

< 函数返回类型说明符 > (*< 指针变量名 >)(< 参数说明表 >);

例如：

```
int(*p)();  
float(*q)(float,int);
```

需要注意的是，指针变量名前后的括号不可少。若缺省，则会与函数原型混淆，错误。

如果已经将某个函数的地址赋给了一个指向函数的指针，就可以通过指针来调用函数。

```
//example7-2用梯形积分法求解定积分  
#include<iostream>  
#include<cmath>  
using namespace std;  
double integral(double a,double b,double (*fun)(double),int n){  
    //double (*fun)(double)即需要传入一个返回值与之对应的函数的首地址  
    double h=(b-a)/n;  
    double sum=(((*fun)(a)+(*fun)(b))/2;  
    int i;  
    for(i=1;i<n;i++) sum+=(*fun)(a+i*h);  
    sum*=h;  
    return sum;  
}  
int main(){  
    double (*fp)(double);  
    fp=cos;  
    cout<<integral(0.0,1.0,sin,1000)<<endl;  
    cout<<integral(0.0,1.0,fp,1000);  
    return 0;  
}
```



§7.4 函数庶事

7.4.1 递归函数

如果某个函数的函数体内有调用其自身的语句，则称该函数为递归函数。

递归是一种非常重要的编程手段。其思想核心是分治，也就是将某个大问题分解成规模较小的子问题加以求解，在分解时需要注意，子问题应与原问题具有相同的结构但规模却不断缩小。这么说估计你没懂，看个例子吧：

```
//example7-3 利用递归求1~n的和
int sum(int n){
    if(n==1) return 1;
    else return n+sum(n-1);
}
```

如果你看懂了，就跳过这一段。如上所示，我们假设想要计算15的和值，在第一次调用时初始传入5，此时 $n==5$ ，返回 $5+sum(4)$ ； $sum(4)$ 中返回 $4+sum(3)$ ； $sum(3)$ 中返回 $3+sum(2)$ ； $sum(2)$ 中返回 $2+sum(1)$ ； $sum(1)$ 返回1。如此所示，逐次将返回值回代，你就得到最终返回了 $5+4+3+2+1=15$ 的值。

可以看出，迭代其实就是通过递归实现的，如下例所示：

```
//example7-4 求斐波那契函数第n项的值
int f(int n){
    if(n==1) return 1;
    if(n==2) return 1;
    return f(n-1)+f(n-2);
}
```

利用递归，我们可以求出斐波那契函数各项的值。

有关递归，最后要说的一点是，不知道你有没有发现，循环可以用递归实现啊！只要你通过全局变量或静态局部变量控制一下循环次数再或者直接通过结束时的条件来控制循环就行了！当然万物都有两面性：递归使用函数实现的，编译器难以预先估计存储量，循环次数过多导致中间状态数目过于庞大，（容易爆栈）；而且，递归函数的运行速度也比较慢。

使用递归，要注意控制使函数在合适的时候回代、退出，不然会出现死循环的情况，程序就会超时最后爆栈死掉。



7.4.2 重载函数

函数重载就是指一组参数与返回值不完全相同的函数共用一个函数名 (有点像多义词)。通过重载, 我们的函数可以拥有更强大的功能! 看下面这个例子:

```
//example7-5重载加法函数
int plus(int a,int b){
    return a+b;
}
double plus(double a,double b){
    return a+b;
}
```

在上面这个例子中, 我们通过重载加法函数使之可以应对两个整数的加法与两个双精度浮点数的加法 (虽然看起来没啥用), 我们的 **plus** 函数的功能更加强大了!

咱们来看一个有点用的例子:

```
//example7-6重载求绝对值函数
int abs(int x){
    return x>=0x:-x;
}
double abs(double x){
    return x>=0x:-x;
}
```

通过重载, 我们可以使用 **abs** 函数对整型和浮点型数据取绝对值, 而不需要使用 **fabs** 了, **abs** 函数的功能得到了强化!

上述例子仅展示了参数与返回值的数据类型不同的重载, 其实完全可以将参数表中的数据个数进行重载。重载过后, 在调用函数时, 编译器会自动判断你想要调用哪个函数, 并调用之。

牢记重载时函数名不可改变, 其他所有东西均可根据需要进行改变且必须不同, 不然会报错。仅有返回值类型不同的几个函数是不能够重载的, 因为编译器会不知道你想返回一个什么值而不知道你想调用哪个函数。



7.4.3 带参数的 main () 函数

main 函数的函数原型为：int main(int argc,char *argv[]);

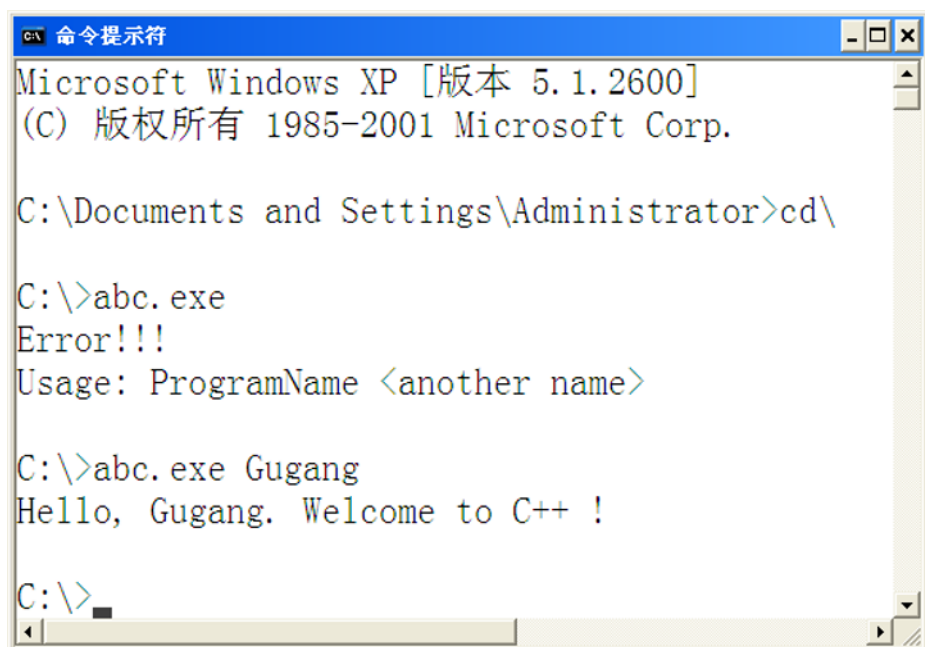
第一个整型参数用以指明以命令行方式执行本程序是所带的参数个数（包括程序名本身，所以，argc 的值至少为 1）；第二个参数为一个字符型指针数组，其第一个变量 argv[0] 指向本程序名的字符串，接下来的 argc-1 个参数 argv[1]、argv[2]……分别指向命令行传递给程序的各个参数，用来存放命令行中命令字及各个参数的字符串。

举例如下：

```
//example7-7带参数的main函数的使用，假设文件名设为abc.cpp
#include<iostream>
using namespace std;
int main(int a, char *ar[]) {
    if(a!=2){
        cout<<"Error!!!"<<endl;
        cout<<"Usage: ProgramName <sb' s name>"<<endl;
        return 1;
    }
    cout<<"Hello, "<<ar[1]<<". Welcome to C++ !"<<endl;
    return 0;
}
```

调试运行步骤：1. 输入带参数 main 函数的源程序 2. 按 abc.cpp 保存，然后编译成功 3. 退出 DEV，返回到 WINDOWS 界面 4. 将 abc.exe 复制到 C 盘根目录 5. 按序点击：“开始” → “附件” → “命令提示符”





```
命令提示符
Microsoft Windows XP [版本 5.1.2600]
(C) 版权所有 1985-2001 Microsoft Corp.

C:\Documents and Settings\Administrator>cd\

C:\>abc.exe
Error!!!
Usage: ProgramName <another name>

C:\>abc.exe Gugang
Hello, Gugang. Welcome to C++ !

C:\>
```



第八章 结构体与枚举

§8.1 结构体

在实际使用中，仅有基本数据类型常常是不够用的。面对数据类型混乱，而彼此间又有相互联系的复杂数据时，可以考虑使用结构体说明这种数据。

定义的一般形式为：

```
struct <结构体类型名>
{
<结构体类型的成员变量说明语句表>
};
```

注意不要忘记最后的分号。

例如：

```
struct Date{
    int year;
    int month;
    int day;
};
```

这样，我们就有了一个 **Date** 的数据类型，用它声明或定义变量的方式与基本数据类型相同，如：

```
Date today;
```

today 变量中含有三个整型数据，想要访问或赋值的话需要用到 “.” 运算符，使用方式为：

```
today.year=2020;
```



§8.2 枚举

枚举类型是 C++ 的一种自定义数据类型，是由用户定义的若干枚举常量的组合，关键字为 **enum**，定义格式为：

enum < 类型名 > < 枚举常量表 >;

枚举的常量若没有指定，则为从 0 开始逐次加一的常量，也可以指定某个常量的值，其后的常量从此逐个加一。例如：

```
//example
enum color{RED,BLACK,BLUE=1,YELLOW}
//RED=0,BLACK=1,BLUE=1,YELLOW=2
enum week {Sun=7, Mon=1, Tue, Wed, Thu, Fri, Sat};
//Sun,Mon,Tue,Wed,Thu,Fri,Sat的值分别为7、1、2、3、4、5、6
```

注意：枚举类型定义的只是常量，与 **const** 作用相同。枚举常量只能用标识符表示，与变量命名规则相同。



第九章 类和对象

§9.1 类与对象的声明

9.1.1 类的声明

C++ 中的类是由 C 语言中的结构体演变而来的，所以类与结构体十分类似。但是他们也有区别：结构体只有数据成员，而类除了数据之外，还包含可以操作数据的一组函数。但是本质上讲，他们都是一类抽象数据类型。

类的定义以关键字 **class** 开始，定义体放在大括号之间，最后用分号表示结束，一般格式如下：

```
//example
class <类名>{
    private:
        <私有成员>;
    protected:
        <保护成员>;
    public:
        <公有成员>;
}
```

上面出现了三种不同的成员属性，这表示类成员的访问控制权限。

私有部分 (**private**) 和保护部分 (**protected**) 的数据成员和成员函数只能在类的范围内或被本类的成员函数访问，其区别在于继承后对派生类的影响不同，公有部分 (**public**) 的成员既可以被本类成员访问，也可以在类外被该类成员访问。

声明时的顺序是任意的，也可以不包含所有的部分。如果没有显示指定，默认为私有部分。



例如:

```
//example
class Book{
    char* content;//未显示指定，为私有部分
    public://公有部分
        int TotalPage;
        char* catalog;
    private://私有部分
        char author;
        char edition;
}
```

9.1.2 对象的定义

对象的定义方式与基本数据类型类似，一般形式为：

<类名><对象 1>,<对象 2>;

可见，类是某种抽象出来的集合，代表了某一批对象的共性，是一种类型，而满足这种类型的实体才是对象。

§9.2 成员函数

类中数据的定义与结构体相同，类中的函数称为成员函数。定义的一般格式为：

```
<类型> <类名>::<函数名>(<参数表>){
    <函数体>
}
```

这与一般的函数定义基本相同，只是多了作用域运算符“::”，表明了函数是该类下的函数，访问要收到一定的限制。

下面举一个成员函数定义的例子



```
//example
class Book{
    public://公有部分
        int TotalPage;
        void outPage(void){//类内定义
            cout<< TotalPage<<endl;
        }
        void outAuthor(void);//函数声明
    private://私有部分
        char author;
}
void Book::outAuthor(void){//类外定义
    cout<< author<<endl;
}
```

可见，类外定义需要先在类内声明这个函数所在的位置（确定访问限制类型），之后再类的定义外写下函数体，此时需要使用二元作用域运算符“::”，而在类内定义则不需要。

§9.3 对象的访问

(1) 在类的作用域内，可以直接访问同类中的数据成员或调用同类中的成员函数；

(2) 在类的作用域外，对象只能访问本类的公有数据或公有函数，需要用到“.”运算符，例如：

```
//example
Book book1;
...
book1.TotalPage=1;
book1.outpage(void);
```

(3) 同类对象间可以整体赋值，所有参数都被完全复制。



§9.4 静态成员

每当声明一个对象时，就会为该对象分配一块内存来存放所有成员，即每个对象都拥有属于自己的所有成员，对象之间的成员不会互相干扰。但是在某些情况下，我们希望同一类的不同对象共享某个成员，对此，较好地解决办法是将其定义成类的静态成员。

使用关键字 **static** 修饰的成员叫做静态成员，可为数据也可为函数。例如：

```
//example
class Book{
    public:
        int TotalPage;//普通数据成员
        void outPage(void){//普通函数成员
            cout<< TotalPage<<endl;
        }
        static void outEdition(void);//静态函数成员
    private:
        static char* edition;//静态数据成员
}

static void Book::outEdition(void){
    cout<< edition<<endl;
}
```

§9.5 结构体与类的区别与联系

结构体类型是 C++ 从 C 语言那里继承下来的东西。在 C 中，结构体的使用方式比较简单，仅有如第八章所说的成员变量；而在 C++ 中，结构体的功能被大大强化，与类一样，可以有成员函数也有访问限制。C++ 中的结构体与类的唯一区别就是：未明确说明的情况下，结构体的成员是公有的，类的成员是私有的。

但是，尽管结构体与类在 C++ 中基本相同，习惯上还是在结构体定义中仅包含数据，涉及成员函数时尽量使用类的定义。



第十章 类的构造与析构

§10.1 知识要点

10.1.1 构造函数

构造函数是用来给对象初始化的函数。理论上来说构造函数体内可以做任何事情，但一般只专用做成员变量的初始化。构造函数在对象创建之时自动调用，任何对构造函数的显示调用都是不被允许的。构造函数没有返回类型，函数名固定为类名，但是允许按照重载规则给出不同的参数类型的构造函数。在没有给任何构造函数时，系统会自动给出缺省构造函数，里面什么也不做。

在一些 OOP（面向对象）语言比如 Java 当中，编译器会自动给类里的成员变量按照统一的规则进行初始化；然而 C++ 语言当中，如果没有定义构造函数，它是**不会**自动给类里面的成员变量初始化的。这是 C++ 为了提高编译效率而考量的。因此如果在对类里的成员进行操作之前没有进行初始化，那么成员变量里的值将会是**不确定**。

除了在构造函数的函数体内进行初始化之外，C++ 还允许在构造函数头部列一个参数表来进行初始化。

10.1.2 析构函数

析构函数常常用来做对象消亡之前的扫尾工作，常常是用来对 new 出来的空间做 delete。析构函数没有返回类型，函数名是加上类名，不带参数，不能重载。析构函数在对象消亡之时自动调用，在任何地方显示调用析构函数也是不对的。

编写析构函数体现了一个程序设计者的良好素质

虽然按照顾刚老师的要求，只要程序不出错，不编写析构函数也不会扣分。但是为了培养自己“善始善终”的意识，即使不需要在析构函数里做任何



事情，也建议写一个空的析构函数。如下例：

```
1 class A{  
2     private:  
3         int a;  
4     public:  
5         A(int n) {a = n;}  
6         ~A() {}  
7 };
```

当这个类采用 `new` 运算符动态申请内存空间时，则必须编写析构函数并在析构函数体中写相应的 `delete` 语句。否则就会产生内存垃圾：即被 `new` 申请的内存变量用完之后没有还给操作系统，那么这块内存就无人管理，无法再次利用。如果申请的内存比较多，而且程序运行很多次，那么逐渐增多的内存垃圾就会占满整个内存空间从而引起系统崩溃。用 `new` 而不写析构函数，或者没有做 `delete` 是一定会被扣分的。

10.1.3 对象和指针

可以定义指向对象的指针。通过该指针访问对象成员时要用 `->` 运算符。每一个对象都含有一个指向本类的 `this` 指针。

10.1.4 string 类的成员函数

`string` 类提供了一些非常方便的对于字符串的操作。使用这些函数不仅方便快捷，而且增强了代码的鲁棒性（robust）即可靠性。但是要熟练应用这些成员函数不仅需要牢记函数原型，还需要一定量的练习。在我们现阶段的学习中暂时没有必要记住。加上顾刚老师也不喜欢记忆这些东西，作业和考试很少涉及。

§10.2 重难点分析

10.2.1 什么是缺省构造函数？

Default constructor(缺省构造函数)：是指程序员编写的不带参数的构造函数。按照函数重载的规则，缺省构造函数最多只能有一个。

(1) 当程序员没有写任何的构造函数时，编译器会自动给出 `default constructor`；



(2) 当程序员编写了构造函数时 (无论是带参的还是不带参的), 编译器就不再给 default constructor 了;

(3) 当程序员只给了带参数的构造函数, 但是初始化时却调用的是缺省构造函数, 编译器会报错:

```
1 #include<cstdio>
2 class A{
3 private:
4     int a;
5 public:
6     A(int n) { a = n;}
7     void Print(){
8         printf("I am a function\n");
9     }
10    ~A() {printf("I am a destructor!\n");}
11 };
12 int main(){
13     A a1;
14     a1.Print();
15     return 0;
16 }
```

Line	Col	File	Message
		C:\Users\lenovo\Documents\C++\cpp\nil_constructor...	In function 'int main()':
13	4	C:\Users\lenovo\Documents\C++\cpp\nil_constructor.c...	[Error] no matching function for call to 'A::A()'
13	4	C:\Users\lenovo\Documents\C++\cpp\nil_constructor.c...	[Note] candidates are:
6	2	C:\Users\lenovo\Documents\C++\cpp\nil_constructor.c...	[Note] A::A(int)
6	2	C:\Users\lenovo\Documents\C++\cpp\nil_constructor.c...	[Note] candidate expects 1 argument, 0 provided
2	7	C:\Users\lenovo\Documents\C++\cpp\nil_constructor.c...	[Note] A::A(const A&)
2	7	C:\Users\lenovo\Documents\C++\cpp\nil_constructor.c...	[Note] candidate expects 1 argument, 0 provided

此时只要加上缺省构造函数就可以了:

```
1 #include<cstdio>
2 class A{
3 private:
4     int a;
5 public:
6     A() {printf("I am a default constructor!\n");}
7     A(int n) { a = n;}
8     void Print(){
9         printf("I am a function\n");
10    }
11    ~A() {printf("I am a destructor!\n");}
12 };
13 int main(){
14     A a1;
15     a1.Print();
16     return 0;
17 }
```

```
I am a default constructor!
I am a function
I am a destructor!

Process exited after 0.4286 seconds with return value 0
请按任意键继续. . .
```

```
Compilation results...
-----
- Errors: 0
- Warnings: 0
- Output Filename: C:\Users\lenovo\Documents\C++\cpp\nil_
- Output Size: 256.0400390625 KiB
- Compilation Time: 0.94s
```

10.2.2 拷贝构造函数

拷贝构造函数是对象的又一种初始化方法, 常常会在作业和考试中遇到, 所以建议读者跟顾老师多多探讨探讨。

(1) 在 C++ 中, initialization(初始化) 和 assignment(赋值) 是完全不同的两个概念。具体如下:

```
#include<iostream>
using namespace std;
class A{
    int a;
public:
    A(int n){
        a = n;
        cout<<"I am a constructor!"<<endl;
    }
    void Print() { cout << a << endl; }
    A(){
        a = 0;
    }
    ~A(){}
};
int main(){
    //C++当中, initialization和assignment是两个完全不同的概念
    A a1(10), a2;
    a2 = a1;//assignment
    A a3 = a1;//initialization
    cout<<"a1: ";
    a1.Print();
    cout<<"a2: ";
    a2.Print();
    cout<<"a3: ";
    a3.Print();
    return 0;
}
```




```
选择C:\Users\lenovo\Documents\C++\cpp\difference.exe
I am a constructor!
a1: 10
a2: 10
a3: 10

-----
Process exited after 0.3848 seconds with return value 0
请按任意键继续. . .
```

在以上的例子当中，创建三个对象，但是带参的构造函数仅仅被调用了一次。而 `a2 = a1` 是赋值操作，不会再调用构造函数。也就是说，`a3` 的初始化并没有经过我们给出的构造函数。

(2) 在初始化时，用圆括号和用 `=` 是等价的

```
#include<iostream>
using namespace std;
class A{
    int a;
public:
    A(int n){
        a = n;
        cout<<"I am a constructor!"<<endl;
    }
    void Print() { cout << a << endl; }
    A(){
        a = 0;
    }
    ~A(){}
};

int main(){
    A a1(10); //C++中初始化时，用圆括号和用赋值号等价
    A a2 = 10; //这里实际上调用了构造函数
    cout << "a1是: ";
    a1.Print();
    cout << "a2是: ";
    a2.Print();
    return 0;
}
```



```

C:\Users\lenovo\Documents\C++\cpp\ini_asi.exe
I am a constructor!
I am a constructor!
a1是: 10
a2是: 10

-----
Process exited after 0.4168 seconds with return value 0
请按任意键继续. . .

```

(3) 拷贝构造函数: 如果我们有一个构造函数, 这个构造函数的参数是本类的一个 **const reference** (常量引用), 那么这样的构造函数就能够实现用本类的对象的初始化。这个构造函数叫做拷贝构造函数 (**copy constructor**) 函数原型是

```
T::T(const T&);
```

例如:

```

A(const A& a1) { //copy constructor
    a = a1.a;
}

```

下面看一个用拷贝构造进行初始化的例子

```

#include<iostream>
using namespace std;
class A{
    int a;
public:
    A(int n){
        a = n;
        cout<<"I am a constructor!"<<endl;
    }
    A(const A &a1) { //copy constructor
        a = a1.a;
        cout<<"I am a copy constructor"<<endl;
    }
    void Print() { cout << a << endl; }
    A(){
        a = 0;
    }
}

```



```

    }
    ~A() {}
};

int main() {
    A a1(10); // constructor
    cout << "a1是: ";
    a1.Print();
    A a2 = a1; // copy constructor
    cout << "a2是: ";
    a2.Print();
    return 0;
}

```

```

C:\Users\lenovo\Documents\C++\cpp\difference.exe
I am a constructor!
a1是: 10
I am a copy constructor
a2是: 10

-----
Process exited after 0.4121 seconds with return value 0
请按任意键继续. . .

```

(4) 如果不写成 `const reference` 的形式，而直接传进去一个对象，这样在传值的过程中也会发生拷贝构造。这样将会发生无穷递归，是编译器所不允许的。如下例：

```

1  #include<iostream>
2  using namespace std;
3  class A{
4      int a;
5  public:
6      A(int n){
7          a = n;
8          cout << "I am a constructor!" << endl;
9      }
10     A(A a1) { // copy constructor
11         // 1/7 public constructor A::A ()
12     }
13     void Print() { cout << a << endl; }
14     A(){
15         a = 0;
16     }
17     ~A(){}
18 };

```

Message
[Error] invalid constructor; you probably meant 'A (const A&)'

(5) 如果我们没有给出拷贝构造函数，C++ 会自动给出一个拷贝构造函数。在这个默认的拷贝构造函数中，它会拷贝每一个成员变量。

- 1 如果成员变量当中还含有其他类的对象，那么拷贝构造会递归下去。
- 2 如果成员变量里面含有指针或引用的话，那么指针和引用也会被拷贝过



去。这就意味着会有两个指针或引用指向同一块内存空间！这是非常危险的！

(6) 何时拷贝构造函数会被调用？

- 1 当某个函数的参数是对象（而不是对象的引用和指针）时；
- 2 当函数的返回值是对象时。



第十一章 继承

§11.1 知识要点

11.1.1 继承的基本概念

C++ 中的继承机制是实现软件重用的一种手段。它使得人们能够从一个已有的类派生出另一个类。前者成为基类或者父类 (parent class); 后者成为派生类或者子类 (children class)。

11.1.2 派生类

派生类声明的一般格式为:

```
class 派生类名:继承方式 基类名1, 继承方式 基类名2, ...,  
            继承方式 基类名n  
{  
    新增加的成员声明;  
};
```

派生类既可以由一个基类继承而来,也可以由多个基类继承而来。派生类的成员包括三种情况:

- (1) 从基类继承而来;
- (2) 对基类进行扩充的新增成员;
- (3) 对基类成员的改造。

11.1.3 派生类的继承方式和访问属性

访问权限的设计可以采取以下规则:

- (1) 让所有的数据成员都是 private;



(2) 把让所有人都用的东西放在 `public` 里;

(3) 留一些 `protected` 的接口给子类。

基类成员在派生类中的访问属性总结可以参看课本 P192 表 9-1

11.1.4 派生类的构造函数和析构函数

派生类的构造函数跟基类的构造函数紧密关联。在以下两种情况:

(1) 派生类本身需要构造函数

(2) 基类的构造函数带有参数

时必须定义派生类的构造函数。派生类的构造函数执行次序如下:

基类构造函数 → 内嵌对象的构造函数 → 派生类构造函数体

派生类的析构函数与基类的析构函数没有什么联系,二者各自做自己所属对象的内存释放工作。在派生类本身需要做内存释放工作时必须定义派生类的析构函数。执行次序如下:

派生类的析构函数 → 内嵌对象的析构函数 → 基类的析构函数

§11.2 重难点分析

11.2.1 软件重用的办法

在 C++ 当中,软件重用一般有两种方式。其一是对象组合 (composition),比如把一个类的对象作为另一个类的成员变量;其二就是继承。

对象组合。

对象组合又有两种方式, `Fully` 和 `By Reference` .

`Fully` 是指直接把一个类的对象作为另一个类的成员变量; `By Reference` 是指将一个对象的指针或者引用作为另一个类的成员变量。

以下是 `Fully` 的一个例子:



```

#include<iostream>
using namespace std;
//肝脏
class Liver {
public:
    //器官功能
    void LiverF() {
        cout << "I am a liver!\n";
    }
};
//心脏
class Heart {
public:
    //器官功能
    void HeartF() {
        cout << "I am a heart!\n";
    }
};
//人类
class Human {
    //器官
    Heart myHeart;
    Liver myLiver;
    //...其他各种器官
public:
    //人的功能
    void myHearF() { //人的心脏的功能
        myHeart.HeartF();
    }
    void myLiverF() { //人的肝脏的功能
        myLiver.LiverF();
    }
}

```



```

void Study() {
    cout << "I am a human, I can study!\n";
}
void StudyAndStudy() {
    cout << "I am a human, I can study and study!\n";
}
void StudyAndStudyAndStudy() {
    cout << "I am a human, I can study and study and study!\n";
}
//...人的其他功能
};
int main(){
    Human yyl;
    yyl.myHearF();
    yyl.myLiverF();
    yyl.Study();
    return 0;
}

```

C:\Users\lenovo\Documents\C++\cpp\Fully.exe

```

I am a heart!
I am a liver!
I am a human, I can study!

```

Process exited after 0.6768 seconds with return value 0
请按任意键继续. . .

分析：以上是构造一个人类的例子。要构造一个人，首先构造人的各种器官，然后用各种器官组成一个人。

以下是 By Reference 的一个例子：




```

//以下是树的链式存储结构
class BinaryTreeNode{//二叉树结点类
private:
    int data; //数据域
    BinaryTreeNode * left;//左儿子指针
    BinaryTreeNode * right;//右儿子指针
public:
    //... (一些函数)
};

class BinaryTree{//二叉树类
protected:
    BinaryTreeNode * root; //二叉树根结点指针
public:
    //... (一些函数)
};

```

分析：以上是二叉树类的一个例子。我们构造这个二叉树类的方法是先构造一个二叉树结点类，然后再用这个结点类来构造二叉树类。就好比在画一棵树之前先画好叶子，然后再用一个一个的叶子来组成一棵树。根据二叉树的定义，二叉树有且仅有一个结点叫做根节点。因此二叉树类里面就包含了一个二叉树结点的指针，以此来作为根结点的指针。具体如何用根结点指针来构造出整棵二叉树来，有它具体的算法，这里暂时不需要知道。



第十二章 多态性

§12.1 知识要点

12.1.1 多态性概述

多态性是 C++ 面向对象程序设计的几大属性之一（抽象、封装、继承、多态）。多态性要求某一个类的不同对象在面对相同消息时可以做出不同响应。

具体地说，多态性是指：在由继承形成的家族的各个类中，可以存在具有相同名称，相同参数表的方法（函数），这些同名的方法在不同类中的具体行为各不相同；当使用基类的引用变量或指针引用不同的派生类对象时，如果为他们发送相同的消息，请求它们执行同名的方法，则根据对象实际所属的类，在程序运行时动态的选用在该类中定义的方法，不同的对象相应消息的具体行为各不相同。

12.1.2 派生类对象替换基类对象

派生类对象替换基类对象的原则是：凡是基类对象出现的场合都可以用公有派生类对象来替换，包括以下三种情况：

- (1) 派生类对象给基类对象赋值
- (2) 派生类对象初始化基类对象的引用
- (3) 派生类对象的指针指向基类对象

但是不论是哪种情况，派生类对象在替换基类对象之后只能当作基类对象来使用。为了实现多态性，必须使用虚函数。

12.1.3 虚函数

`virtual` 使得基类和派生类的同名函数之间产生了某种联系。虽然基类和派生类的析构函数的名称一定不一样，但是有时也要用 `virtual`



只要在继承树上有一个祖先的某个函数被定义成了虚函数，那么从该祖先开始之后的派生类子子孙孙中，不论加不加 **virtual** 这个函数都是虚函数。但是仍然建议写上 **virtual** 以加强代码可读性。

如何才能产生动态绑定的效果？

(1) 只有使用派生类对象的指针或者引用来调用虚函数才会产生动态绑定的效果。直接用派生类对象给基类对象赋值，所得到的对象仍然只能当作基类对象来使用；

(2) 直接以对象名调用虚函数 (**obj.Speak();**) 不会产生动态绑定；

(3) 析构函数一般也要写成虚函数。因为如果没有把析构函数写成虚函数，那么派生类对象在被 **delete** 的时候发生的是静态绑定，调用的是基类的析构函数。而派生类的析构函数则没有被调用，不起作用。因此只要类里面有至少一个是虚函数，那么析构函数必须也是虚函数。

12.1.4 运算符重载

C++ 运算符和函数一样也可以重载。运算符重载一般是为了扩大运算符的适用范围。比如 **+** 原来可用于实数范围，经过重载之后可以对复数对象使用。理论上运算符重载的函数体内可以写任何内容，但是脱离运算符原来语义的函数体是不被推荐的。

有哪些运算符可以被重载？

```
+ - * / % ^ & | ~
= < > += -= *= /= %=
^= &= |= << >> >>= <<= ==
!= <= >= ! && || ++ --
, ->* -> () []
operator new      operator delete
operator new[]    operator delete[]
```

注释：**new** 和 **delete** 运算符也是可以重载的。这样可以允许我们自己来定义动态申请内存空间的操作。

有哪些运算符不可以被重载？



```
.      *      ::      ?:  
sizeof  typeid  
static_cast dynamic_cast const_cast  
reinterpret_cast
```

(1) 除此之外，只有已经存在的运算符可以被重载，自创的运算符不能被重载。

(2) 运算符重载前后操作数，优先级和结合性不会变。

(3) **const** 关键字在运算符重载当中的妙用：当某个运算符不会改变操作数本身的值时，就要在重载的时候把他设置成相应的 **const** 型。比如 “+” 运算符不会修改左右两个操作数的值，那么重载函数的参数和返回值都应该是 **const**。如下例：

```
#include<iostream>  
using namespace std;  
class Complex{//复数类  
    double Re, Im;//实部和虚部  
public:  
    Complex(int r, int i) { Re = r; Im = i; }  
    //复数类加法运算符重载，这里有几个问题下面详细叙述  
    const Complex operator+(const Complex &c) const  
    { return Complex(c.Re + this->Re, c.Im + this->Im); }  
    void Print() { cout << Re << "+" << Im << "i" ; }  
};  
int main(){  
    Complex c1(1,2), c2(2,2);  
    Complex c3 = c1 + c2;  
    c1.Print();  
    cout<<"+";  
    c2.Print();  
    cout<<"=";  
    c3.Print();  
    return 0;  
}
```



}

```
C:\Users\lenovo\Documents\C++\cpp\test_overload.exe
1+2i+2+2i=3+4i
-----
Process exited after 0.4049 seconds with return value 0
请按任意键继续. . .
```

分析:

(1) 正如 `+` 运算所要求的那样, 两个复数对象相加产生第三个复数对象, 而且不允许修改两个操作数的值。因此参数和返回类型都是 `const Complex`

(2) 重载 `+` 的函数也写要成 `const`, 如果不这样做, 就意味着 `c1+c2` 的结果可以做左值。但在现阶段不写成 `const` 也一般不会出现太大问题。

(3) 在一些原本就修改操作数的运算符比如 `++`, `-`, 则不用加 `const`

下面来看一些常用的运算符重载函数的原型:

- `+ - * / % ^ & | ~`
- `const T operatorX(const T& l, const T& r) const;`
- `! && || < <= == >= >`
- `bool operatorX(const T& l, const T& r) const;`
- `[]`
- `T& T::operator[](int index);`

关于 `++` 和 `-`: `++` 运算符要区分前缀和后缀, 即 `++a` (先 `++` 后引用) 和 `a++` (先引用后 `++`) 的效果是不一样的。前者是先改变 `a` 的值, 再在 `++a` 所在表达式当中引用 `a` 的值; 后者是先在 `a++` 所在表达式当中引用 `a` 的值, 引用完之后再改变 `a` 的值。相应地, `++` 运算符重载也要有前缀 (`prefix`) 和后缀 (`postfix`) 之分。



```

class Integer {
public:
    ...
    const Integer& operator++(); //prefix++
    const Integer operator++(int); //postfix++
    const Integer& operator--(); //prefix--
    const Integer operator--(int); //postfix--
    ...
};

```

且看下例:

```

#include<iostream>
using namespace std;
class Complex{
    double Re, Im;
public:
    Complex(int r, int i) { Re = r; Im = i; }
    const Complex operator+(const Complex &c)
    { return Complex(c.Re + this->Re, c.Im + this->Im); }
    //复数加法：实部与虚部分别相加
    //前缀++, 复数的++定义为实部++
    const Complex& operator++() { Re++; return *this; }
    //后缀++, 复数的++定义为实部++
    const Complex operator++(int)
    { Complex temp = *this; Re++; return temp; }
    void Print() { cout << Re << "+" << Im << "i" ; }
};

int main(){
    Complex c1(1,2), c2(1,2);
    cout<<"c1 = ";
    c1.Print();
    cout<<"    c2 = ";
    c2.Print();
    Complex c3 = c2 + (c1++); //先引用后++, c3的结果应该是2+4i
    cout<<endl<<"c3 = c2 + (c1++) = ";
}

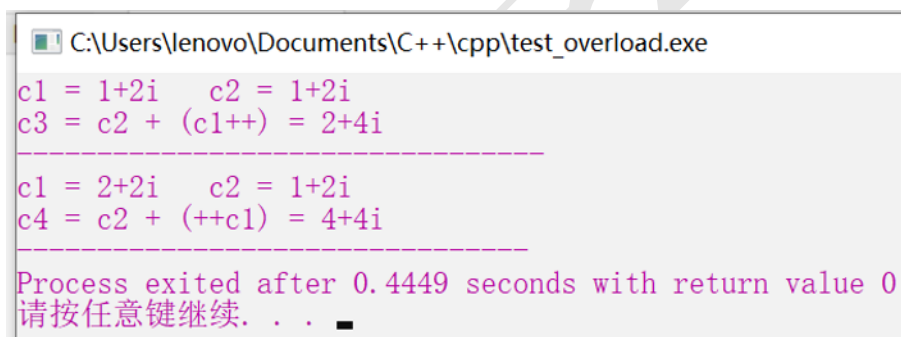
```



```

c3.Print();
cout<<endl<<"-----"<<endl;
cout<<"c1 = ";
c1.Print();
cout<<"    c2 = ";
c2.Print();
Complex c4 = c2 + (++c1); //先++后引用, c4的结果应该是4+4i
cout<<endl<<"c4 = c2 + (++c1) = ";
c4.Print();
return 0;
}

```



```

C:\Users\lenovo\Documents\C++\cpp\test_overload.exe
c1 = 1+2i    c2 = 1+2i
c3 = c2 + (c1++) = 2+4i
-----
c1 = 2+2i    c2 = 1+2i
c4 = c2 + (++c1) = 4+4i
-----
Process exited after 0.4449 seconds with return value 0
请按任意键继续. . .

```

§12.2 重难点分析

12.2.1 const 修饰符

在 C++ 当中，一旦变量被 `const` 修饰，一经初始化后，编译器就会保证它永远不可以被更改。`const` 常常用来在全局当中定义一些公用常量，比如圆周率 `pi`，最大数组长度 `maxSize` 等等。相比宏定义，`const` 变量的好处在于它不会像宏定义那样需要加很多括号。



```
1 #include<iostream>
2 using namespace std;
3 int main(){
4     const int num = 1;
5     num = 2;
6     return 0;
7 }
```

Message

In function 'int main()':
[Error] assignment of read-only variable 'num'

(1) 但要注意, 被 `const` 修饰的变量仍然是变量, 在内存中占有相应空间, 遵循变量的作用域规则, 和常量有本质区别。

(2) `const` 在定义 (definition, 要分清楚定义和声明 (declaration) 的区别) 时就必须初始化, 否则编译报错

```
7 int main(){
8     const int num;
9     cout<<num<<endl;
10    return 0;
11 }
```

Message

In function 'int main()':
[Error] uninitialized const 'num' [-fpermissive]

(3) 在 `const` 前面再以 `extern` 修饰, 这是告诉编译器此处声明 (declaration)



了一个位于当前作用域（scope）之外的一个全局 const 变量，不可以修改它。
extern 与初始化同时出现是不对的：

```
1  #include<iostream>
2  using namespace std;
3  int f(int n){
4      extern const int num = 1;
5      return 1;
6  }
7  int main(){
8      const int num = 2;
9      cout<<num<<endl;
10     return 0;
11 }
```

Message

In function 'int f(int)':
[Error] 'num' has both 'extern' and initializer

因为 extern 是声明而不是定义，正确用法是用 extern 声明之后，在别处定义这个 const 变量：

```
testconst.cpp
1  #include<iostream>
2  using namespace std;
3  int f(int n){
4      extern const int num;
5      return 1;
6  }
7  int main(){
8      const int num = 2;
9      cout<<num<<endl;
10     return 0;
11 }
```

C:\Users\lenovo\Documents\C++\cpp\testco
2

Process exited after 0.6001 seconds
请按任意键继续. . .

Compilation results...

- Errors: 0
- Warnings: 0



这种由编译器保证的 `const` 叫做 **compile time constants**

(4) 用 `const` 变量做数组大小时，一定要保证在数组定义之前 `const` 变量的值就已经经过初始化确定了，否则编译器是不知道该分配多大的内存空间的。

(5) `const` 与指针：

```
• char * const q = "abc"; // q is const
  *q = 'c'; // OK
  q++;      // ERROR
• const char *p = "ABCD";
  // (*p) is a const char
  *p = 'b'; // ERROR! (*p) is the const
```

第一行是定义了一个 `const` 指针，这个指针今后不能指向其他内存空间，但是该内存空间的值是可以更改的；

第二行是一个指针指向了一个 `const` 变量，这个指针还可以指向其他变量，但是该指针目前所指内存空间里的值不能更改。

(6) `const` 与对象：

```
Person p1( "Fred", 200 );
const Person* p = &p1;
Person const* p = &p1;
Person *const p = &p1;
```

第一行：对象是 `const`

第三行：指针是 `const`

第二行：对象是 `const`

判断的标准是 `const` 在 * 的前面还是后面，在 * 前面是对象为 `const`；反之指针是 `const`

还可以写成：

```
Person const* const p = &p1;
```

这样对象和指针都是 `const`。

用 `const` 修饰整个对象时，整个对象里面的数据都不能被修改。这样的规定下就会带来一个问题：当调用该对象的成员函数时，有可能成员函数会修改对象当中的数据。因此为了保证被 `const` 修饰对象的数据不被修改，所有修改数据的成员函数都不能被访问；而不会修改成员数据的函数可以访问，只要在该函数的后面加上 `const` 就可以了。修改数据的函数不能以 `const` 修饰。



12.2.2 赋值运算符重载

如果类的对象中存在指针类型的数据成员时，当使用一个已经存在的对象去初始化另一个正在创建的对象时，就会发生两个指针指向相同内存地址的情况，这种现象成为对象的浅拷贝。浅拷贝增加了程序模块之间的耦合性，在程序运行的过程中容易引发异常。消除浅拷贝的办法之一是创建类的拷贝构造函数（见重难点分析），但是它并不能彻底解决对象的浅拷贝问题。解决问题的办法是：为类重载赋值运算符，再进行对象赋值时，赋值运算符重载函数将被调用。

但是还要注意一个问题：当类的成员变量有指针时，在赋值运算符重载函数一开始要先判断 `this != &that`；如果 `this == &that`，那么直接返回 `*this` 就可以了，且看下例：

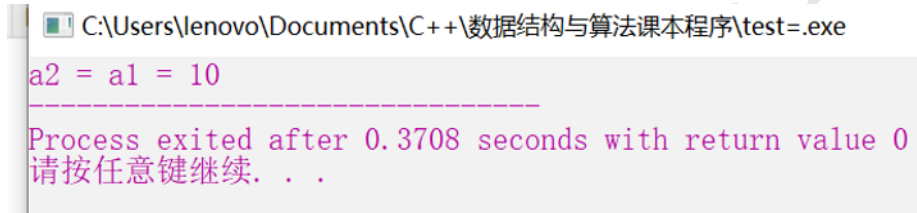
```
#include<iostream>
using namespace std;
class A{
private:
    int *numptr;
public:
    A() { numptr = new int(1); }
    A(int num) { numptr = new int(num); }
    //A(A&); //声明拷贝构造函数
    ~A() { delete numptr; }
    A& operator = (const A&); //声明赋值运算符重载
    int getValue() { return *numptr; }
};

A&A::operator = (const A &a){
    if (this != &a){
        delete numptr;
        numptr = new int(*a.numptr);
    }
    return *this;
}

int main(){
    A a1(10);
```



```
A a2;  
a2 = a1;//a2被赋值为10  
cout<<"a2 = a1 = "<<a2.getValue();  
return 0;  
}
```



```
C:\Users\lenovo\Documents\C++\数据结构与算法课本程序\test=.exe  
a2 = a1 = 10  
-----  
Process exited after 0.3708 seconds with return value 0  
请按任意键继续. . .
```

分析: 在上面的赋值运算符重载中, 如果没有判 `if (this != &a)`, 由于 `numptr` 是动态申请的空间, 要先 `delete` 掉。可是如果 `this == &a`, 那么 `numptr` 被 `delete` 掉之后就变成空指针了, 显然会出错。因此如果 `this == &a`, 直接返回 `*this` 就可以了。



第十三章 标准库和输入输出流

§13.1 知识要点

13.1.1 C++ 流概述

C++ 用流来做输入输出。相比 C 语言的 `printf` 和 `scanf`, C++ 流输入输出能够提供更好的类型安全性。而且也能够实现基于面向对象的一些高级操作。但是它也有一些缺点,比如比较啰嗦,通常也比较慢。

在 C++ 程序里面, C 的一切输入输出库函数都是可以使用的。C++ 的流运算符可以重载,这是它相对于 C 的又一优势。

流的概念:流是一维、单方向的。这意味着只需要一个参数就可以确定流中任意一个位置。而对于 C 语言来说,文件读入是可以随机存取,可以任意读写。而 C++ 流不一样,只能向前走,不能任意读写。也就是“一个人从来也不能两次跨入相同的河流”。

本章内容的重点是文件读写。考试也会有一道题考到。

§13.2 重难点分析

13.2.1 文件读写

磁盘文件的输入/输出操作由以下 3 步构成:

- (1) 创建文件输入/输出流对象
- (2) 打开磁盘文件,并将其和上一步创建的输入/输出流对象相关联
- (3) 使用各种输入/输出操作符、格式化标志和函数完成输入/输出操作。

以下举一个例子:用带参数 `main` 函数和文件读写来实现文件复制功能
源代码如下:



```

#include<fstream>
#include<cstring>
#include<iostream>
using namespace std;
//复制文件
int main(int argc, char *argv[]){
    if (argc != 2){
        //argv[0]是文件名，之后argv[1]是我们设置的参数文件名。
        //复制文件只需要1个文件就够了，因此总共2个参数
        cout<<"Usage:<filename>\n";//提示用户本程序的用法
    }
    //创建文件输入流，以二进制方式读写可以使程序适用范围更广
    ifstream in(argv[1], ios::in | ios::binary);
    if (!in){//排查错误
        cout<<"Cannot open the file "<<argv[1]<<endl;
        return 1;
    }
    char outFile[strlen(argv[1]) + 10];
    strcpy(outFile,argv[1]);
    strcat(outFile,".mycopy");//做出输出文件名，最后加上后缀.copy
    ofstream out(outFile, ios::out | ios::binary);//创建输出流
    if (!out){
        cout<<"Cannot create the copy file\n";
        return 1;
    }
    char ch;
    while (in.get(ch)){//向目标文件写入字符
        out<<ch;
    }
    cout<<"output: "<<outFile<<endl;//提示用户输出文件的文件名
    in.close();
    out.close();
    return 0;
}

```



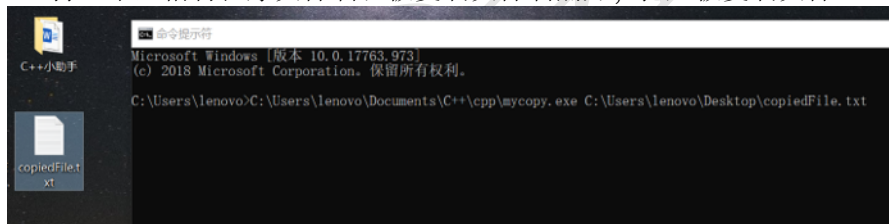
调试:

1、在 windows 中查找 cmd



2、找到编译生成的可执行程序文件.exe 所在位置，并将其拖入命令行

3、打一个空格将程序文件名和被复制文件名隔开，拖入被复制文件



4、回车，运行程序，得到输出文件



5、按照程序提示的输出文件路径找到得到的复制文件

