

## 目录

|                            |     |
|----------------------------|-----|
| <b>第1篇 基本知识</b> .....      | 2   |
| 第1章 C++的初步知识 .....         | 2   |
| 第2章 数据类型与表达式 .....         | 8   |
| <b>第2篇 面向过程的程序设计</b> ..... | 22  |
| 第3章 程序设计初步 .....           | 23  |
| 第4章 函数与预处理 .....           | 134 |
| 第5章 数组 .....               | 167 |
| 第6章 指针 .....               | 187 |
| 第7章 自定义数据类型 .....          | 225 |
| <b>第3篇 基于对象的程序设计</b> ..... | 248 |
| 第8章 类和对象 .....             | 248 |
| 第9章 关于类和对象的进一步讨论 .....     | 267 |
| 第10章 运算符重载 .....           | 297 |
| <b>第4篇 面向对象的程序设计</b> ..... | 317 |
| 第11章 继承与派生 .....           | 317 |
| 第12章 多态性与虚函数 .....         | 342 |
| 第13章 输入输出流 .....           | 354 |
| 第14章 C++工具 .....           | 376 |

# 第1篇 基本知识

## 第1章 C++的初步知识

### \*1.1 从C到C++

#### \*1.2 最简单的C++程序

##### 1.3 C++程序的构成和书写形式

##### 1.4 C++程序的编写和实现

##### 1.5 关于C++上机实践

### \*1.1 从C到C++

计算机诞生初期,人们要使用计算机必须用机器语言或汇编语言编写程序。世界上第一种计算机高级语言诞生于 1954 年,它是 FORTRAN 语言。先后出现了多种计算机高级语言。其中使用最广泛、影响最大的当推 BASIC 语言和 C 语言。

BASIC 语言是 1964 年在 FORTRAN 语言的基础上简化而成的,它是为初学者设计的小型高级语言。

C 语言是 1972 年由美国贝尔实验室的 D.M.Ritchie 研制成功的。它不是为初学者设计的,而是为计算机专业人员设计的。大多数系统软件和许多应用软件都是用 C 语言编写的。

但是随着软件规模的增大,用 C 语言编写程序渐渐显得有些吃力了。

C++是由 AT&T Bell(贝尔)实验室的 Bjarne Stroustrup 博士及其同事于 20 世纪 80 年代初在 C 语言的基础上开发成功的。C++保留了 C 语言原有的所有优点,增加了面向对象的机制。

C++是由 C 发展而来的,与 C 兼容。用 C 语言写的程序基本上可以不加修改地用于 C++。从 C++的名字可以看出它是 C 的超集。C++既可用于面向过程的结构化程序设计,又可用于面向对象的程序设计,是一种功能强大的混合型的程序设计语言。

C++对 C 的“增强”,表现在两个方面:

(1) 在原来面向过程的机制基础上,对 C 语言的功能做了不少扩充。

(2) 增加了面向对象的机制。

面向对象程序设计,是针对开发较大规模的程序而提出来的,目的是提高软件开发的效率。

不要把面向对象和面向过程对立起来,面向对象和面向过程不是矛盾的,而是各有用途、互为补充的。

学习 C++,既要会利用 C++进行面向过程的结构化程序设计,也要会利用 C++进行面向对象的程序设计。本书既介绍 C++在面向过程程序设计中的应用,也介绍 C++在面向对象程序设计中的应用。

### \*1.2 最简单的 C++程序

例 1.1 输出一行字符:“This is a C++ program.”。

程序如下:

```
#include <iostream> //包含头文件 iostream
using namespace std; //使用命名空间 std
int main()
{
    cout<<" This is a C++ program." ;
    return 0;
}
```

在运行时会在屏幕上输出以下一行信息:

This is a C++ program.

用 main 代表“主函数”的名字。每一个 C++程序都必须有一个 main 函数。main 前面的 int 的作用是声明函数

的类型为整型。程序第 6 行的作用是向操作系统返回一个零值。如果程序不能正常执行,则会自动向操作系统返回一个非零值,一般为-1。

函数体是由大括号{ }括起来的。本例中主函数内只有一个以cout开头的语句。注意C++所有语句最后都应当有一个分号。

再看程序的第 1 行“#include <iostream>”,这不是 C++的语句,而是 C++的一个预处理命令,它以“#”开头以与 C++语句相区别,行的末尾没有分号。#include <iostream>是一个“包含命令”,它的作用是将文件 iostream 的内容包含到该命令所在的程序文件中,代替该命令行。文件 iostream 的作用是向程序提供输入或输出时所需要的一些信息。iostream 是 i-o-stream 3 个词的组合,从它的形式就可以知道它代表“输入输出流”的意思,由于这类文件都放在程序单元的开头,所以称为“头文件”(head file)。在程序进行编译时,先对所有的预处理命令进行处理,将头文件的具体内容代替 #include 命令行,然后再对该程序单元进行整体编译。

程序的第 2 行“using namespace std;”的意思是“使用命名空间 std”。C++标准库中的类和函数是在命名空间 std 中声明的,因此程序中如果需要用到 C++标准库(此时就需要用#include 命令行),就需要用“using namespace std;”作声明,表示要用到命名空间 std 中的内容。

在初学 C++时,对本程序中的第 1,2 行可以不必深究,只需知道: 如果程序有输入或输出时,必须使用“#include <iostream>”命令以提供必要的信息,同时要用“using namespace std;”,使程序能够使用这些信息,否则程序编译时将出错。

例 1.2 求 a 和 b 两个数之和。

可以写出以下程序:

```
// 求两数之和 (本行是注释行)
#include <iostream> //预处理命令
using namespace std; //使用命名空间 std
int main() //主函数首部
{ //函数体开始
    int a,b,sum; //定义变量
    cin>>a>>b; //输入语句
    sum=a+b; //赋值语句
    cout<<" a+b=" <<sum<<endl; //输出语句
    return 0; //如程序正常结束,向操作系统返回一个零值
} //函数结束
```

本程序的作用是求两个整数 a 和 b 之和 sum。第 1 行“//求两数之和”是一个注释行,C++规定在一行中如果出现“//”,则从它开始到本行末尾之间的全部内容都作为注释。

如果在运行时从键盘输入

123 456↵

则输出为

a+b=579

例 1.3 给两个数 x 和 y,求两数中的大者。

在本例中包含两个函数。

```
#include <iostream> //预处理命令
using namespace std;
int max(int x,int y) //定义 max 函数,函数值为整型,形式参数 x,y 为整型
{ //max 函数体开始
    int z; //变量声明,定义本函数中用到的变量 z 为整型
    if(x>y) z=x; //if 语句,如果 x>y,则将 x 的值赋给 z
    else z=y; //否则,将 y 的值赋给 z
    return(z); //将 z 的值返回,通过 max 带回调用处
} //max 函数结束
```

```

int main() //主函数
{ //主函数体开始
    int a,b,m; //变量声明
    cin>>a>>b; //输入变量 a 和 b 的值
    m=max(a,b); //调用 max 函数,将得到的值赋给 m
    cout<<" max=" <<m<<' \n' ; //输出大数 m 的值
    return 0; //如程序正常结束,向操作系统返回一个零值
} //主函数结束

```

本程序包括两个函数:主函数 main 和被调用的函数 max。

程序运行情况如下:

18 25 ↵ (输入 18 和 25 给 a 和 b)

max=25 (输出 m 的值)

注意输入的两个数据间用一个或多个空格间隔,不能以逗号或其他符号间隔。

在上面的程序中,max 函数出现在 main 函数之前,因此在 main 函数中调用 max 函数时,编译系统能识别 max 是已定义的函数名。如果把两个函数的位置对换一下,即先写 main 函数,后写 max 函数,这时在编译 main 函数遇到 max 时,编译系统无法知道 max 代表什么含义,因而无法编译,按出错处理。

为了解决这个问题,在主函数中需要对被调用函数作声明。上面的程序可以改写如下:

```

#include <iostream>
using namespace std;
int main()
{ int max(int x,int y); //对 max 函数作声明
    int a,b,c;
    cin>>a>>b;
    c=max(a,b); //调用 max 函数
    cout<<" max=" <<c<<endl;
    return 0;
}

```

```

int max(int x,int y) //定义 max 函数
{ int z;
    if(x>y) z=x;
    else z=y;
    return(z);
}

```

只要在被调用函数的首部的末尾加一个分号,就成为对该函数的函数声明。函数声明的位置应当在函数调用之前。

下面举一个包含类(class)和对象(object)的 C++ 程序,目的是使读者初步了解 C++ 是怎样体现面向对象程序设计方法的。

例 1.4 包含类的 C++ 程序。

```

#include <iostream> // 预处理命令
using namespace std;
class Student // 声明一个类,类名为 Student
{private: // 以下为类中的私有部分
    int num; // 私有变量 num
    int score; // 私有变量 score
public: // 以下为类中的公用部分
    void setdata() // 定义公用函数 setdata

```

```

{cin>>num; // 输入 num 的值
cin>>score; // 输入 score 的值
}

void display() // 定义公用函数 display
{cout<<" num=" <<num<<endl; // 输出 num 的值
cout<<" score=" <<score<<endl; // 输出 score 的值
};
}; // 类的声明结束
Student stud1,stud2; // 定义 stud1 和 stud2 为 Student 类的变量,称为对象
int main() // 主函数首部
{stud1.setdata(); // 调用对象 stud1 的 setdata 函数
stud2.setdata(); // 调用对象 stud2 的 setdata 函数
stud1.display(); // 调用对象 stud1 的 display 函数
stud2.display(); // 调用对象 stud2 的 display 函数
return 0;
}

```

在一个类中包含两种成员：数据和函数，分别称为数据成员和成员函数。在 C++ 中把一组数据和有权调用这些数据的函数封装在一起，组成一种称为“类(class)”的数据结构。在上面的程序中，数据成员 num, score 和成员函数 setdata, display 组成了一个名为 Student 的“类”类型。成员函数是用来对数据成员进行操作的。也就是说，一个类是由一批数据以及对其操作的函数组成的。

类可以体现数据的封装性和信息隐蔽。在上面的程序中，在声明 Student 类时，把类中的数据和函数分为两大类：private(私有的)和 public(公用的)。把全部数据(num, score)指定为私有的，把全部函数(setdata, display)指定为公用的。在大多数情况下，会把所有数据指定为私有，以实现信息隐蔽。

具有“类”类型特征的变量称为“对象”(object)。

程序中第 18~24 行是主函数。

程序运行情况如下：

1001 98.5 ✓ (输入学生 1 的学号和成绩)

1002 76.5 ✓ (输入学生 2 的学号和成绩)

num=1001 (输出学生 1 的学号)

score=98.5 (输出学生 1 的成绩)

num=1002 (输出学生 2 的学号)

score=76.5 (输出学生 2 的成绩)

### 1.3 C++ 程序的构成和书写形式

C++ 程序的结构和书写格式归纳如下：

(1) 一个 C++ 程序可以由一个程序单位或多个程序单位构成。每一个程序单位作为一个文件。在程序编译时，编译系统分别对各个文件进行编译，因此，一个文件是一个编译单元。

(2) 在一个程序单位中，可以包括以下几个部分：

① 预处理命令。上节 4 个程序中都包括 #include 命令。

② 全局声明部分(在函数外的声明部分)。在这部分中包括对用户自己定义的数据类型的声明和程序中所用到的变量的定义。

③ 函数。函数是实现操作的部分，因此函数是程序中必须有的和最基本的组成部分。每一个程序必须包括一个或多个函数，其中必须有一个(而且只能有一个)主函数(main 函数)。

但是并不要求每一个程序文件都必须具有以上 3 个部分，可以缺少某些部分(包括函数)。

(3) 一个函数由两部分组成：

① 函数首部，即函数的第一行。包括函数名、函数类型、函数属性、函数参数(形参)名、参数类型。

一个函数名后面必须跟一对圆括号，函数参数可以缺省，如 int main()。

② 函数体，即函数首部下面的大括号内的部分。如果在一个函数中有多个大括号，则最外层的一对 { } 为函数

体的范围。

**函数体一般包括:**

局部**声明部分** (在函数内的声明部分)。包括对本函数中所用到的类型、函数的声明和变量的定义。

对数据的声明既可以放在函数之外(其作用范围是全局的),也可以放在函数内(其作用范围是局部的,只在本函数内有效)。

**执行部分**。由若干个执行语句组成,用来进行有关的操作,以实现函数的功能。

(4) **语句包括两类**。一类是**声明语句**,另一类是**执行语句**。C++对每一种语句赋予一种特定的功能。语句是实现操作的基本成分,显然,没有语句的函数是没有意义的。C++语句必须**以分号结束**。

(5) 一个C++程序总是从**main 函数开始执行的**,而不论 main 函数在整个程序中的位置如何。

(6) **类(class)**是 C++**新增加的重要的数据类型**,是 C++对 C 的最重要的发展。有了类,就可以实现面向对象程序设计方法中的**封装、信息隐蔽、继承、派生、多态**等功能。在一个类中可以包括数据成员和成员函数,他们可以被指定为私有的(**private**)和公用的(**public**)属性。私有的数据成员和成员函数只能被本类的成员函数所调用。

(7)C++程序**书写格式自由**,一行内可以写几个语句,一个语句可以分写在多行上。C++程序没有行号,也不像 FORTRAN 或 COBOL 那样严格规定书写格式(语句必须从某一列开始书写)。

(8) 一个好的、有使用价值的源程序都应当加上**必要的注释**,以增加程序的可读性。C++还保留了 C 语言的注释形式,可以用“/\*.....\*/”对 C++程序中的任何部分作注释。在“/\*”和“\*/”之间的全部内容作为注释。

**用“/\*”作注释时**,有效范围只有一行,即本行有效,不能跨行。而用“/\*.....\*/”作注释时有效范围为多行。只要在开始处有一个“/\*”,在最后一行结束处有一个“\*/”即可。因此,一般习惯是:内容较少的简单注释常用“/\*”,内容较长的常用“/\*.....\*/”。

#### 1.4 C++程序的编写和实现

一个程序从编写到最后得到运行结果要**经历以下一些步骤**。

##### 1. 用 C++语言编写程序

用高级语言编写的程序称为“**源程序**”(source program)。C++的源程序是以.cpp 作为后缀的(cpp 是 c plus plus 的缩写)。

##### 2. 对源程序进行编译

为了使计算机能执行高级语言源程序,必须先用一种称为“**编译器(compiler)**”的软件(也称编译程序或编译系统),把源程序翻译成二进制形式的“**目标程序(object program)**”。

编译是以源程序文件为单位分别编译的。目标程序一般以.obj 或.o 作为后缀(object 的缩写)。编译的作用是对源程序进行词法检查和语法检查。编译时对文件中的全部内容进行检查,编译结束后会显示出所有的编译出错信息。一般编译系统给出的出错信息分为两种,一种是错误(error);一种是警告(warning)。

##### 3. 将目标文件连接

在改正所有的错误并全部通过编译后,得到一个或多个目标文件。此时要用系统提供的“**连接程序(linker)**”将一个程序的所有目标程序和系统的库文件以及系统提供的其他信息连接起来,最终形成一个可执行的二进制文件,它的后缀是.exe,是可以直接执行的。

##### 4. 运行程序

**运行**最终形成的可执行的二进制文件(.exe 文件),得到运行结果。

##### 5. 分析运行结果

如果运行结果不正确,应**检查**程序或算法是否有问题。

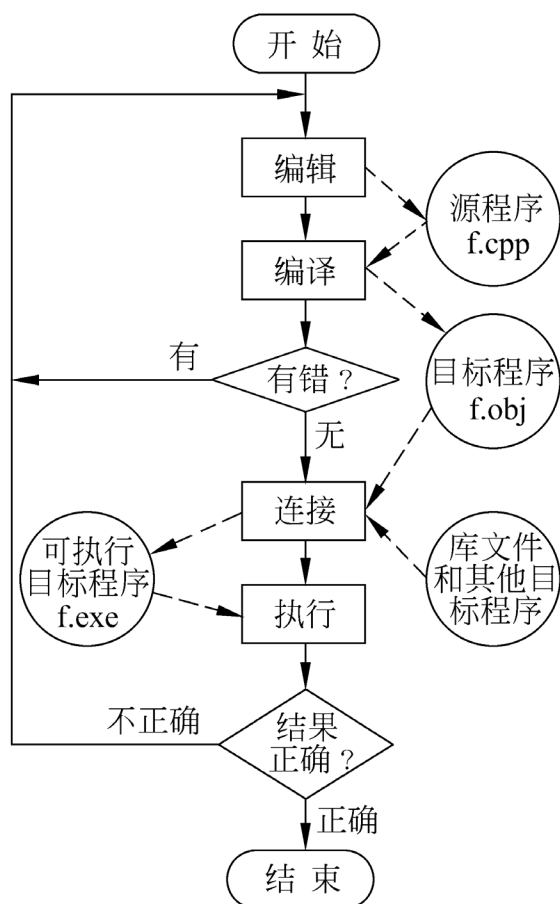


图 1.1

### 1.5 关于 C++上机实践

在了解了 C++语言的初步知识后,读者最好尽快在计算机上编译和运行 C++程序,以加深对 C++程序的认识,并初步掌握 C++的上机操作。

读者可以使用不同的 C++编译系统,在不同的环境下编译和运行一个 C++程序。但是需要强调的是,我们学习的是 C++程序设计,应当掌握的是标准 C++,而不应该只了解某一种“方言化”的 C++。不应当只会使用一种 C++编译系统,只能在一种环境下工作,而应当能在不同的 C++环境下运行自己的程序,并且了解不同的 C++编译系统的特点和使用方法,在需要时能将自己的程序方便地移植到不同的平台上。

在本书的参考书《C++程序设计题解与上机指导》一书中简单介绍了在 Visual C++ 6.0 和 GCC 两种典型的环境下运行 C++程序的方法。

请读者选择一种(如能做到两种更好) C++编译系统,在该环境下输入和运行习题中的程序,掌握上机的方法和步骤。



## 第 2 章 数据类型与表达式

### 2.1 C++的数据类型

### 2.2 常量

### 2.3 变量

### 2.4 C++的运算符

### 2.5 算术运算符与算术表达式

### 2.6 赋值运算符与赋值表达式

### 2.7 逗号运算符与逗号表达式

#### 2.1 C++的数据类型

计算机处理的对象是数据,而数据是以某种特定的形式存在的(例如整数、浮点数、字符等形式)。不同的数据之间往往还存在某些联系(例如由若干个整数组成一个整数数组)。数据结构指的是数据的组织形式。例如,数组就是一种数据结构。不同的计算机语言所允许使用的数据结构是不同的。处理同一类问题,如果数据结构不同,算法也会不同。例如,对 10 个整数排序和对包含 10 个元素的整型数组排序的算法是不同的。

C++可以使用的数据类型如下:





布尔型就是逻辑型，空类型就是无值型。

C++的数据包括常量与变量,常量与变量都具有类型。由以上这些数据类型还可以构成更复杂的数据结构。例如利用指针和结构体类型可以构成表、树、栈等复杂的数据结构。

C++并没有统一规定各类数据的精度、数值范围和在内存中所占的字节数,各 C++编译系统根据自己的情况作出安排。书中表 2.1 列出了 Visual C++数值型和字符型数据的情况。

说明:

- (1) 整型数据分为长整型(long int)、一般整型(int)和短整型(short int)。在 int 前面加 long 和 short 分别表示长整型和短整型。
- (2) 整型数据的存储方式为按二进制数形式存储,例如十进制整数 85 的二进制形式为 1010101,则在内存中的存储形式如图 2.1 所示。

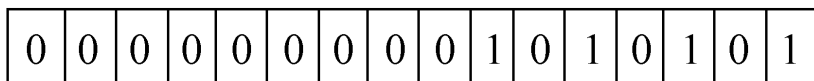


图 2.1

- (3) 在整型符号 int 和字符型符号 char 的前面,可以加修饰符 signed(表示“有符号”)或 unsigned(表示“无符号”)。如果指定为 signed,则数值以补码形式存放,存储单元中的最高位(bit)用来表示数值的符号。如果指定为 unsigned,则数值没有符号,全部二进制位都用来表示数值本身。例如短整型数据占两个字节,见图 2.2。

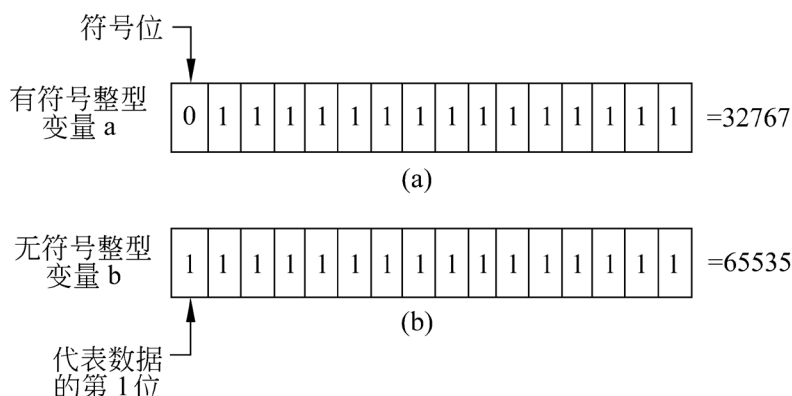


图 2.2

有符号时,能存储的最大值为  $2^{15}-1$ ,即 32767,最小值为-32768。无符号时,能存储的最大值为  $2^{16}-1$ ,即 65535,最小值为 0。有些数据是没有负值的,可以使用 unsigned,它存储正数的范围比用 signed 时要大一倍。

- (4) 浮点型(又称实型)数据分为单精度(float)、双精度(double)和长双精度(long double)3 种,在 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。

(5) 表中类型标识符一栏中,方括号[]包含的部分可以省写,如 short 和 short int 等效,unsigned int 和 unsigned 等效。

常量的值是不能改变的,一般从其字面形式即可判别是否为常量。常量包括两大类,即数值型常量(即常数)和字符型常量。如 12,0,-3 为整型常量,4.6,-1.23 为实型常量,包含在两个单撇号之间的字符为字符常量,如'a','x'。这种从字面形式即可识别的常量称为“字面常量”或“直接常量”。

## 2.2 常量

### 2.2.1 什么是常量

#### 2.2.2 数值常量

数值常量就是通常所说的常数。在 C++中,数值常量是区分类型的,从字面形式即可识别其类型。

##### 1. 整型常量(整数)的类型

在上一节中已知道:整型数据可分为 int,short int,long int 以及 unsigned int,unsigned short,unsigned long 等类别。整型常量也分为以上类别。为什么将数值常量区分为不同的类别呢?因为在进行赋值或函数的参数虚实结合时要求数据类型匹配。

那么,一个整型常量怎样从字面上区分为以上的类别呢?

(1) 一个整数,如果其值在-32768~+32767 范围内,认为它是 short int 型,它可以赋值给 short int 型、int 型和 long int 型变量。

(2) 一个整数,如果其值超过了上述范围,而在-2147483648~+2147483647 范围内,则认为它是 long int 型,可以将它赋值给一个 int 或 long int 型变量。

(3) 如果某一计算机系统的 C++版本(例如 Visual C++)确定 int 与 long int 型数据在内存中占据的长度相同,则它们能够表示的数值的范围相同。因此,一个 int 型的常量也同时是一个 long int 型常量,可以赋给 int 型或 long int 型变量。

(4) 常量无 unsigned 型。但一个非负值的整数可以赋值给 unsigned 整型变量,只要它的范围不超过变量的取值范围即可。

一个整型常量可以用 3 种不同的方式表示:

(1) 十进制整数。如 1357,-432,0 等。在一个整型常量后面加一个字母 l 或 L,则认为是 long int 型常量。例如 123L,421L,0L 等,这往往用于函数调用中。如果函数的形参为 long int,则要求实参也为 long int 型,此时用 123 作实参不行,而要用 123L 作实参。

(2) 八进制整数。在常数的开头加一个数字 0,就表示这是以八进制数形式表示的常数。如 020 表示这是八进制数 20,即 $(20)_8$ ,它相当于十进制数 16。

(3) 十六进制整数。在常数的开头加一个数字 0 和一个英文字母 X(或 x),就表示这是以十六进制数形式表示的常数。如 0X20 表示这是十六进制数 20,即 $(20)_{16}$ ,它相当于十进制数 32。

##### 2. 浮点数的表示方法

一个浮点数可以用两种不同的方式表示:

(1) 十进制小数形式。如 21.456,-7.98 等。它一般由整数部分和小数部分组成,可以省略其中之一(如 78.或.06,0),但不能二者皆省略。C++编译系统把用这种形式表示的浮点数一律按双精度常量处理,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,-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),在 GCC 中占 12 个字节,在 Visual C++ 6.0 中占 8 个字节。

(2) 指数形式(即浮点形式)

一个浮点数可以写成指数形式,如 3.14159 可以表示为  $0.314159 \times 10^1$ ,  $3.14159 \times 10^0$ ,  $31.4159 \times 10^{-1}$ ,

$314.159 \times 10^{-2}$  等形式。在程序中应表示为: 0.314159e1, 3.14159e0, 31.4159e-1, 314.159e-2, 用字母 e 表示其后的

数是以 10 为底的幂,如 e12 表示  $10^{12}$ 。

其一般形式为

**数符** **数字部分** **指数部分**

上面各数据中的 0.314159,3.14159, 31.4159, 314.159 等就是其中的数字部分。可以看到：由于指数部分的存在,使得同一个浮点数可以用不同的指数形式来表示,数字部分中小数点的位置是浮动的。例如：

a=0.314159e1;

a=3.14159e0;

a=31.4159e-1;

a=314.159e-2;

以上 4 个赋值语句中,用了不同形式的浮点数,但其作用是相同的。

在程序中不论把浮点数写成小数形式还是指数形式,在内存中都是以指数形式(即浮点形式)存储的。例如不论在程序中写成 314.159 或 314.159e0,31.4159e1,3.14159e2, 0.314159e3 等形式,在内存中都是以规范化的指数形式存放,如图 2.3 所示。

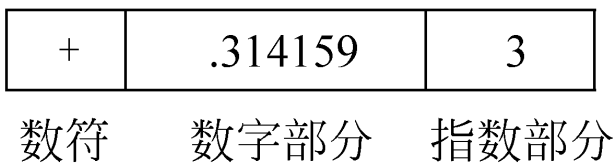


图 2.3

数字部分必须小于 1,同时,小数点后面第一个数字必须是一个非 0 数字,例如不能是 0.0314159。因此 314.159 和 314.159e0,31.4159e1,3.14159e2, 0.314159e3 在内存中表示成  $0.314159 \times 10^3$ 。存储单元分为两部分,一部分用来存放数字部分,一部分用来存放指数部分。为便于理解,在图 2.3 中是用十进制表示的,实际上在存储单元中是用二进制数来表示小数部分,用 2 的幂次来表示指数部分的。

对于以指数形式表示的数值常量,也都**作为双精度常量**处理。

### 2.2.3 字符常量

#### 1. 普通的字符常量

用**单撇号**括起来的一个字符就是字符型常量。如'a','#','%','D'都是合法的字符常量,在内存中占一个字节。

注意：

- ①字符常量只能包括一个字符,如'AB' 是不合法的。
- ②字符常量区分大小写字母,如'A'和'a'是两个不同的字符常量。
- ③撇号(')是定界符,而不属于字符常量的一部分。如 cout<<'a';输出的是一个字母“a”,而不是 3 个字符“'a'”。

#### 2. 转义字符常量

除了以上形式的字符常量外,C++还允许用一种特殊形式的字符常量,就是以“\”开头的字符序列。例如,'\n'代表一个“换行”符。“cout<<'\n'; ” 将输出一个换行,其作用与“cout<<endl; ” 相同。这种“控制字符”,在屏幕上是不能显示的。在程序中也无法用一个一般形式的字符表示,只能采用特殊形式来表示。

常用的以“\”开头的特殊字符见书中表 2.2。

#### 3. 字符数据在内存中的存储形式及其使用方法

将一个字符常量存放到内存单元时,实际上并不是将该字符本身放到内存单元中去,而是将该字符相应的 ASCII 代码放到存储单元中。如果字符变量 c1 的值为'a',c2 的值为'b',则在变量中存放的是'a'的 ASCII 码 97,'b' 的 ASCII 码 98,如图 2.4(a)所示,实际上在内存中是以二进制形式存放的,如图 2.4(b)所示。

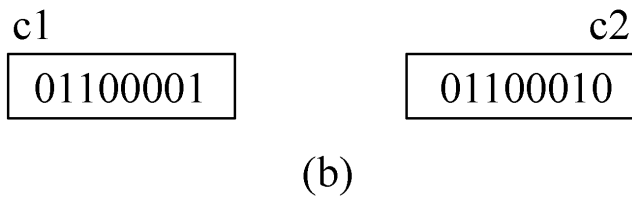
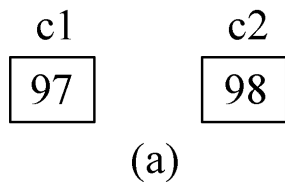


图 2.4

既然字符数据是以 ASCII 码存储的,它的存储形式就与整数的存储形式类似。这样,在 C++ 中字符型数据和整型数据之间就可以通用。一个字符数据可以赋给一个整型变量,反之,一个整型数据也可以赋给一个字符变量。也可以对字符数据进行算术运算,此时相当于对它们的 ASCII 码进行算术运算。

例 2.1 将字符赋给整型变量。

```
#include <iostream>
using namespace std;
int main( )
{
    int i,j; //i 和 j 是整型变量
    i='A'; //将一个字符常量赋给整型变量 i
    j='B'; //将一个字符常量赋给整型变量 j
    cout<<i<<' '<<j<<'\n'; //输出整型变量 i 和 j 的值,'n' 是换行符
    return 0;
}
```

执行时输出

65 66

i 和 j 被指定为整型变量。但在第 5 和第 6 行中,将字符'A'和'B'分别赋给 i 和 j,它的作用相当于以下两个赋值语句:

```
i=65;j=66;
```

因为'A'和'B'的 ASCII 码为 65 和 66。在程序的第 5 和第 6 行是把 65 和 66 直接存放到 i 和 j 的内存单元中。因此输出 65 和 66。

可以看到:在一定条件下,字符型数据和整型数据是可以通用的。但是应注意字符数据只占一个字节,它只能存放 0~255 范围内的整数。

例 2.2 字符数据与整数进行算术运算。下面程序的作用是将小写字母转换为大写字母。

```
#include <iostream>
using namespace std;
int main( )
{
    char c1,c2;
    c1='a';
    c2='b';
    c1=c1-32;
    c2=c2-32;
    cout<<c1<<' '<<c2<<endl;
    return 0;
}
```

运行结果为

A B

'a'的 ASCII 码为 97,而'A'的 ASCII 码为 65,'b'为 98,'B'为 66。从 ASCII 代码表中可以看到每一个小写字母比它相应的大写字母的 ASCII 代码大 32。C++符数据与数值直接进行算术运算,'a'-32 得到整数 65,'b'-32 得到整数 66。将 65 和 66 存放在 c1,c2 中,由于 c1,c2 是字符变量,因此用 cout 输出 c1,c2 时,得到字符 A 和 B(A 的 ASCII 码为 65,B 的 ASCII 码为 66)。

#### 4. 字符串常量

用双撇号括起来的部分就是字符串常量,如"abc","Hello!","a+b","Li ping" 都是字符串常量。字符串常量"abc"在内存中占 4 个字节(而不是 3 个字节),见图 2.5。

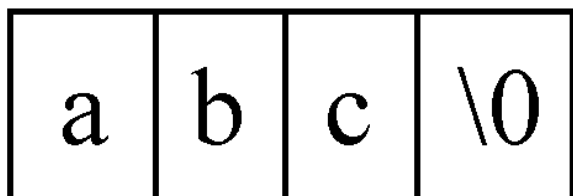


图 2.5

编译系统会在字符串最后自动加一个'\0'作为字符串结束标志。但'\0'并不是字符串的一部分,它只作为字符串的结束标志。如

```
cout<<"abc"<<endl;
```

输出 3 个字符 abc,而不包括'\0'。

注意: "a"和'a'代表不同的含义,"a"是字符串常量,'a' 是字符常量。前者占两个字节,后者占 1 个字节。请分析下面的程序片段:

```
char c;//定义一个字符变量
```

```
c='a';//正确
```

```
c="a"; //错误,c 只能容纳一个字符
```

字符串常量要用字符数组来存放,见第 5 章。

请思考: 字符串常量"abc\n"包含几个字符?不是 5 个而是 4 个字符,其中"\n"是一个转义字符。但它在内存中占 5 个字节(包括一个"\0"字符)。编译系统遇到"\\"时就会把它认作转义字符的标志,把它和其后的字符一起作为一个转义字符。

如果"\\"后面的字符不能与"\\"组成一个合法的转义字符(如"\\c"),则在编译时显示出错信息。如果希望将"\\"字符也作为字符串中的一个字符,则应写为"abc\\n",此时字符包括 5 个字符,即 a,b,c,\\,n。如果有以下输出语句:

```
cout<<"abc\\n"<<endl;
```

则会输出: abc\\,然后换行。同理执行

```
cout<<"I say \\\"Thank you!\\\"\\n";
```

的输出是: I say "Thank you!"

如果在一个字符串中最后一个字符为"\\",则表示它是续行符,下一行的字符是该字符串的一部分,且在两行字符串间无空格。如

```
cout<<"We must study C\\本行最后的\\"后面的空格和换行均不起作用
```

```
++ hard!";//本行的字符紧连在上一行最后的\\"前面字符之后
```

则输出:

```
We must study C++ hard!
```

#### 2.2.4 符号常量

为了编程和阅读的方便,在 C++程序设计中,常用一个符号名代表一个常量,称为符号常量,即以标识符形式出现的常量。

例 2.3 符号常量的使用。

```
#define PRICE 30 //注意这不是语句,末尾不要加分号
```

```
int main ()
```

```

{ int num,total;
num=10;
total=num * PRICE;
cout<<"total="<<total<<endl;
return 0;
}

```

程序中用预处理命令#define 指定 PRICE 在本程序单位中代表常量 30,此后凡在本程序单位中出现的 PRICE 都代表 30,可以和常量一样进行运算,程序运行结果为

total=300

请注意符号常量虽然有名字,但它不是变量。它的值在其作用域(在本例中为主函数)内是不能改变的,也不能被赋值。如用赋值语句“PRICE=40;”给 PRICE 赋值是错误的。使用符号常量的好处是:

- (1) 含义清楚。
- (2) 在需要改变一个常量时能做到“一改全改”。如

```
#define PRICE 35
```

## 2.3 变量

### 2.3.1 什么是变量

其实在前面的例子中已经多次用到了变量。在程序运行期间其值可以改变的量称为变量。一个变量应该有一个名字,并在内存中占据一定的存储单元,在该存储单元中存放变量的值。请注意区分变量名和变量值这两个不同的概念,见图 2.6。

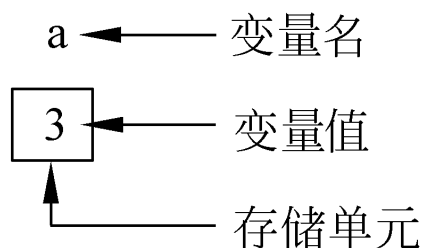


图 2.6

### 2.3.2 变量名规则

先介绍标识符的概念。和其他高级语言一样,用来标识变量、符号常量、函数、数组、类型等实体名字的有效字符序列称为标识符(identifier)。简单地说,标识符就是一个名字。变量名是标识符的一种,变量的名字必须遵循标识符的命名规则。

C++规定标识符只能由字母、数字和下划线 3 种字符组成,且第一个字符必须为字母或下划线。下面列出的是合法的标识符,也是合法的变量名:

sum, average, total, day, month, Student\_name, tan, BASIC, li\_ling

下面是不合法的标识符和变量名:

M.D.John, \$123, #33, 3G64, Ling li, C++, Zhang-ling, U.S.A.

注意,在 C++中,大写字母和小写字母被认为是两个不同的字符。因此,sum 和 SUM 是两个不同的变量名。一般地,变量名用小写字母表示,与人们日常习惯一致,以增加可读性。应注意变量名不能与 C++的关键字、系统函数名和类名相同。在国外软件开发工作中,常习惯在变量前面加一个字母以表示该变量的类型,如 iCount 表示这是一个整型变量,cSex 表示这是一个字符型变量。

C++没有规定标识符的长度(字符个数),但各个具体的 C 编译系统都有自己的规定。有的系统取 32 个字符,超过的字符不被识别。

### 2.3.3 定义变量

在 C++语言中,要求对所有用到的变量作强制定义,也就是必须“先定义,后使用”,如例 2.2 和例 2.3 那样。定义变量的一般形式是

变量类型 变量名表列;

变量名表列指的是一个或多个变量名的序列。如



```
float a,b,c,d,e;
```

定义 a,b,c,d,e 为单精度型变量,注意各变量间以逗号分隔,最后是分号。

可以在定义变量时指定它的初值。如

```
float a=83.5,b,c=64.5,d=81.2,e; //对变量a,c,d 指定了初值,b 和 d 未指定初值
```

C 语言要求变量的定义应该放在所有的执行语句之前,而 C++则放松了限制,只要求在第一次使用该变量之前进行定义即可。也就是说,它可以出现在语句的中间,如

```
int a; //定义变量a(在使用a 之前定义)
```

```
a=3; //执行语句,对a 赋值
```

```
float b; //定义变量b(在使用b 之前定义)
```

```
b=4.67; //执行语句,对b 赋值
```

```
char c; //定义变量c(在使用c 之前定义)
```

```
c='A'; //执行语句,对c 赋值
```

C++要求对变量作强制定义的目的是:

(1) 凡未被事先定义的,不作为变量名,这就能保证程序中变量名使用得正确。例如,如果在声明部分写了

```
int student;
```

而在执行语句中错写成 statent。如

```
statent=30;
```

在编译时检查出 statent 未经定义,作为错误处理。输出“变量 statent 未经声明”的信息,便于用户发现错误,避免变量名使用时出错。

(2) 每一个变量被指定为一确定类型,在编译时就能为其分配相应的存储单元。如指定 a 和 b 为 int 型,一般的编译系统对其各分配 4 个字节,并按整数方式存储数据。

(3) 指定每一变量属于一个特定的类型,这就便于在编译时,据此检查该变量所进行的运算是否合法。例如,整型变量 a 和 b,可以进行求余运算:

```
a%b
```

%是“求余”(见 2.4 节),得到 a/b 的余数。如果将 a 和 b 指定为实型变量,则不允许进行“求余”运算,在编译时会给出有关的出错信息。

### 2.3.4 为变量赋初值

允许在定义变量时对它赋予一个初值,这称为变量初始化。初值可以是常量,也可以是一个有确定值的表达式。如

```
float a,b=5.78*3.5,c=2*sin(2.0);
```

表示定义了 a,b,c 为单精度浮点型变量,对 b 初始化为 5.78\*3, 对 c 初始化为 2\*sin(2.0),在编译连接后,从标准函数库得到正弦函数 sin(2.0)的值,因此变量 c 有确定的初值。变量 a 未初始化。

如果对变量未赋初值,则该变量的初值是一个不可预测的值,即该存储单元中当时的内容是不知道的。例如,若未对 a 和 b 赋值,执行输出语句

```
cout<<a<<" "<<b<<" "<<c<<endl;
```

输出结果可能为

1.48544e-38 15 1.81858(各次运行情况可能不同)

初始化不是在编译阶段完成的(只有在第 4 章中介绍的静态存储变量和外部变量的初始化是在编译阶段完成的),而是在程序运行时执行本函数时赋予初值的,相当于执行一个赋值语句。例如,

```
int a=3;
```

相当于以下两个语句 :

```
int a; // 指定a 为整型变量
```

```
a=3; // 赋值语句,将3 赋给a
```

对多个变量赋予同一初值,必须分别指定,不能写成

```
float a=b=c=4.5;
```

而应写成

```
float a=4.5,b=4.5,c=4.5;
```



或

```
float a,b,c=4.5;
```

```
a=b=c;
```

### 2.3.5 常变量

在定义变量时,如果加上关键字 `const`,则变量的值在程序运行期间不能改变,这种变量称为常变量(constant variable)。例如,

```
const int a=3; //用 const 来声明这种变量的值不能改变,指定其值始终为3
```

在定义常变量时必须同时对它初始化(即指定其值),此后它的值不能再改变。常变量不能出现在赋值号的左边。例如上面一行不能写成

```
const int a;
```

```
a=3; //常变量不能被赋值
```

可以用表达式对常变量初始化,如

```
const int b=3+6,c=3*cos(1.5); //b 的值被指定为9,c 的值被指定为3*cos(1.5)
```

但应注意,由于使用了系统标准数学函数 `cos`,必须将包含该函数有关的信息的头文件“`cmath`”(或 `math.h`)包含到本程序单位中来,可以在本程序单位的开头加上以下 `#include` 命令:

```
#include <cmath> 或 #include <math.h>
```

变量的值应该是可以变化的,怎么值是固定的量也称变量呢?其实,从计算机实现的角度看,变量的特征是存在一个以变量名命名的存储单元,在一般情况下,存储单元中的内容是可以变化的。对常变量来说,无非在此变量的基础上加上一个限定:存储单元中的值不允许变化。因此常变量又称为只读变量(read-only-variable)。

请区别用 `#define` 命令定义的符号常量和用 `const` 定义的常变量。符号常量只是用一个符号代替一个字符串,在预编译时把所有符号常量替换为所指定的字符串,它没有类型,在内存中并不存在以符号常量命名的存储单元。而常变量具有变量的特征,它具有类型,在内存中存在着以它命名的存储单元,可以用 `sizeof` 运算符测出其长度。与一般变量惟一的不同是指定变量的值不能改变。用 `#define` 命令定义符号常量是 C 语言所采用的方法,C++把它保留下来是为了和 C 兼容。C++的程序员一般喜欢用 `const` 定义常变量。虽然二者实现的方法不同,但从使用的角度看,都可以认为用了一个标识符代表了一个常量。有些书上把用 `const` 定义的常变量也称为定义常量,但读者应该了解它和符号常量的区别。

### 2.4 C++的运算符

C++的运算符十分丰富,使得 C++的运算十分灵活方便。例如把赋值号(=)也作为运算符处理,这样,a=b=c=4就是合法的表达式,这是与其他语言不同的。C++提供了以下运算符:

#### (1) 算术运算符

+(加)-(减)\*(乘)/(除) %(整除求余)++(自加)--(自减)

#### (2) 关系运算符

>(大于)<(小于) ==(等于)>=(大于或等于)<=(小于或等于) !=(不等于)

#### (3) 逻辑运算符

&&(逻辑与) ||(逻辑或) !(逻辑非)

#### (4) 位运算符

<<(按位左移) >>(按位右移) &(按位与) |(按位或) ^(按位异或) ~(按位取反)

#### (5) 赋值运算符 (=及其扩展赋值运算符)

#### (6) 条件运算符 (?:)

#### (7) 逗号运算符 (,)

#### (8) 指针运算符 (\*)

#### (9) 引用运算符和地址运算符 (&)

#### (10) 求字节数运算符(sizeof)

#### (11) 强制类型转换运算符( (类型) 或类型())

#### (12) 成员运算符 (.)

#### (13) 指向成员的运算符 (->)

#### (14) 下标运算符 ([])

#### (15) 其他 (如函数调用运算符())

### 2.5 算术运算符与算术表达式

#### 2.5.1 基本的算术运算符

在本章中主要介绍算术运算符与算术表达式,赋值运算符与赋值表达式,逗号运算符与逗号表达式,其他运算符将在以后各章中陆续介绍。

+(加法运算符,或正值运算符。如  $3+5,+3$ )

-(减法运算符,或负值运算符。如  $5-2,-3$ )

\*(乘法运算符。如  $3*5$ )

/(除法运算符。如  $5/3$ )

%(模运算符,或称求余运算符,%两侧均应为整型数据,如  $7\%4$  的值为 3)。

需要说明,两个整数相除的结果为整数,如  $5/3$  的结果值为 1,舍去小数部分。但是,如果除数或被除数中有一个为负值,则舍入的方向是不固定的。例如, $-5/3$  在有的 C++系统上得到结果-1,有的 C++系统则给出结果-2。多数编译系统采取“向零取整”的方法,即  $5/3$  的值等于 1, $-5/3$  的值等于-1,取整后向零靠拢。

如果参加 $+, -, *, /$ 运算的两个数中有一个数为 float 型数据,则运算的结果是 double 型,因为 C++在运算时对所有 float 型数据都按 double 型数据处理。

#### 2.5.2 算术表达式和运算符的优先级与结合性

用算术运算符和括号将运算对象(也称操作数)连接起来的、符合 C++语法规则的式子,称 C++算术表达式。运算对象包括常量、变量、函数等。例如,下面是一个合法的 C++算术表达式:

`a*b/c-1.5+'a'`

C++语言规定了运算符的优先级和结合性。在求解表达式时,先按运算符的优先级别高低次序执行,例如先乘除后加减。如有表达式 `a-b*c`,b 的左侧为减号,右侧为乘号,而乘号优先于减号,因此,相当于 `a-(b*c)`。如果在一个运算对象两侧的运算符的优先级别相同,如 `a-b+c`,则按规定的“结合方向”处理。

C++规定了各种运算符的结合方向(结合性),算术运算符的结合方向为“自左至右”,即先左后右,因此 b 先与减号结合,执行 `a-b` 的运算,再执行加 c 的运算。“自左至右的结合方向”又称“左结合性”,即运算对象先与左面的运算符结合。以后可以看到有些运算符的结合方向为“自右至左”,即右结合性(例如赋值运算符)。关于“结合性”的概念在其他一些高级语言中是没有的,是 C 和 C++的特点之一,希望能弄清楚。附录 B 列出了所有运算符以及它们的优先级别和结合性。

#### 2.5.3 表达式中各类数值型数据间的混合运算

在表达式中常遇到不同类型数据之间进行运算,如

`10+'a'+1.5-8765.1234*'b'`

在进行运算时,不同类型的数据要先转换成同一类型,然后进行运算。转换的规则按图 2.7 所示。

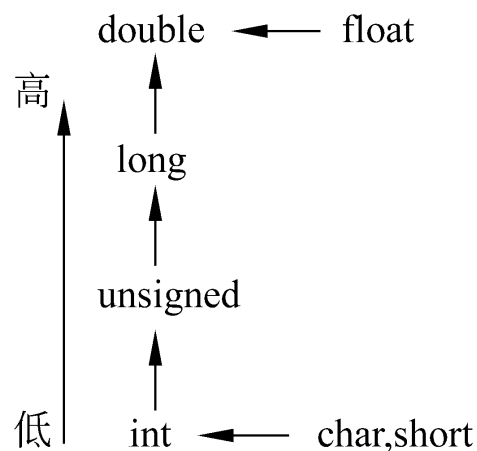


图 2.7

假设已指定 i 为整型变量,f 为 float 变量,d 为 double 型变量,e 为 long 型,有下面表达式:

10+'a'+i\*f-d/e

运算次序为: ①进行 10+'a'的运算,先将'a'转换成整数 97,运算结果为 107。②进行 i\*f 的运算。先将 i 与 f 都转换成 double 型,运算结果为 double 型。③整数 107 与 i\*f 的积相加。先将整数 107 转换成双精度数(小数点后加若干个 0,即 107.000...00),结果为 double 型。④将变量 e 转换成 double 型,d/e 结果为 double 型。⑤将 10+'a'+i\*f 的结果与 d/e 的商相减,结果为 double 型。

上述的类型转换是由系统自动进行的。

#### 2.5.4 自增和自减运算符

在 C 和 C++中,常在表达式中使用自增(++)和自减(--)运算符,他们的作用是使变量的值增 1 或减 1,如

++i(在使用 i 之前,先使 i 的值加 1,如果 i 的原值为 3,则执行 j=++i 后,j 的值为 4)

--i(在使用 i 之前,先使 i 的值减 1,如果 i 的原值为 3,则执行 j=--i 后,j 的值为 2)

i++(在使用 i 之后,使 i 的值加 1,如果 i 的原值为 3,则执行 j=i++后,j 的值为 3,然后 i 变为 4)

i--(在使用 i 之后,使 i 的值减 1,如果 i 的原值为 3,则执行 j=i--后,j 的值为 3,然后 i 变为 2)

++i 是先执行 i=i+1 后,再使用 i 的值;而 i++是先使用 i 的值后,再执行 i=i+1。

正确地使用++和--,可以使程序简洁、清晰、高效。

请注意:

(1) 自增运算符(++)和自减运算符(--)只能用于变量,而不能用于常量或表达式。

(2) ++和--的结合方向是“自右至左”,见附录 B。

(3) 自增运算符(++)和自减运算符(--)使用十分灵活,但在很多情况下可能出现歧义性,产生“意想不到”的副作用。

(4) 自增(减)运算符在 C++程序中是经常见到的,常用于循环语句中,使循环变量自动加 1。也用于指针变量,使指针指向下一个地址。

#### 2.5.5 强制类型转换运算符

在表达式中不同类型的数据会自动地转换类型,以进行运算。有时程序编制者还可以利用强制类型转换运算符将一个表达式转换成所需类型。例如:

(double)a (将 a 转换成 double 类型)

(int)(x+y) (将 x+y 的值转换成整型)

(float)(5%3) (将 5%3 的值转换成 float 型)

强制类型转换的一般形式为

(类型名)(表达式)

注意: 如果要进行强制类型转换的对象是一个变量,该变量可以不用括号括起来。如果要进行强制类型转换的对象是一个包含多项的表达式,则表达式应该用括号括起来。如果写成

(int)x+y

则只将 x 转换成整型,然后与 y 相加。

以上强制类型转换的形式是原来 C 语言使用的形式,C++把它保留了下来,以利于兼容。C++还增加了以下形式:

类型名(表达式)

如 int(x) 或 int(x+y)

类型名不加括号,而变量或表达式用括号括起来。这种形式类似于函数调用。但许多人仍习惯于用第一种形式,把类型名包在括号内,这样比较清楚。

需要说明的是在强制类型转换时,得到一个所需类型的中间变量,但原来变量的类型未发生变化。例如:

(int)x

如果 x 原指定为 float 型,值为 3.6,进行强制类型运算后得到一个 int 型的中间变量,它的值等于 3,而 x 原来的类型和值都不变。

例 2.4 强制类型转换。

```
#include <iostream>
```

```
using namespace std;
```

```
int main( )
```

```

{ float x;
  int i;
  x=3.6;
  i=(int)x;
  cout<<"x="<<x<<" ,i="<<i<<endl;
  return 0;
}

```

运行结果如下:

x=3.6,i=3

x 的型仍为 float 型,值仍等于 3.6。

由上可知,有两种类型转换,一种是在运算时不必用户指定,系统自动进行的类型转换,如 3+6.5。第二种是强制类型转换。当自动类型转换不能实现目的时,可以用强制类型转换。此外,在函数调用时,有时为了使实参与形参类型一致,可以用强制类型转换运算符得到一个所需类型的参数。

## 2.6 赋值运算符与赋值表达式

### 2.6.1 赋值运算符

赋值符号“=”就是赋值运算符,它的作用是将一个数据赋给一个变量。如“a=3”的作用是执行一次赋值操作(或称赋值运算)。把常量 3 赋给变量 a。也可以将一个表达式的值赋给一个变量。

### 2.6.2 赋值过程中的类型转换

如果赋值运算符两侧的类型不一致,但都是数值型或字符型时,在赋值时会自动进行类型转换。

- (1) 将浮点型数据(包括单、双精度)赋给整型变量时,舍弃其小数部分。
- (2) 将整型数据赋给浮点型变量时,数值不变,但以指数形式存储到变量中。
- (3) 将一个 double 型数据赋给 float 变量时,要注意数值范围不能溢出。
- (4) 字符型数据赋给整型变量,将字符的 ASCII 码赋给整型变量。
- (5) 将一个 int、short 或 long 型数据赋给一个 char 型变量,只将其低 8 位原封不动地送到 char 型变量(发生截断)。例如

```
short int i=289;
```

```
char c;
```

```
c=i; //将一个 int 型数据赋给一个 char 型变量
```

赋值情况见图 2.8。为方便起见,以一个 int 型数据占两个字节(16 位)的情况来说明。

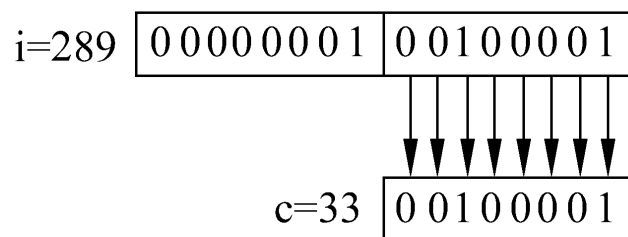


图 2.8

- (6) 将 signed(有符号)型数据赋给长度相同的 unsigned(无符号)型变量,将存储单元内容原样照搬(连原有的符号位也作为数值一起传送)。

例 2.5 将有符号数据传送给无符号变量。

```
#include <iostream>
```

```
using namespace std;
```

```
int main( )
```

```
{ unsigned short a;
```

```
  short int b=-1;
```

```
  a=b;
```

```
  cout<<"a="<<a<<endl;
```

```

return 0;
}

```

运行结果为

65535

赋给 b 的值是-1,怎么会得到 65535 呢?请看图 2.9 所示的赋值情况。

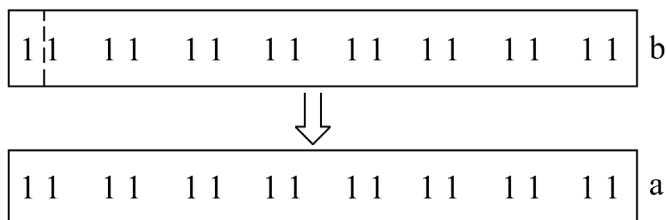


图 2.9

-1 的补码形式为 1111111111111111(即全部 16 个二进制位均为 1),将它传送给 a,而 a 是无符号型变量,16 个位全 1 是十进制的 65535。如果 b 为正值,且在 0~32767 之间,则赋值后数值不变。

不同类型的整型数据间的赋值归根结底就是一条:按存储单元中的存储形式直接传送。

C 和 C++使用灵活,在不同类型数据之间赋值时,常常会出现意想不到的结果,而编译系统并不提示出错,全靠程序员的经验来找出问题。这就要求编程人员对出现问题的原因有所了解,以便迅速排除故障。

### 2.6.3 复合的赋值运算符

在赋值符“=”之前加上其他运算符,可以构成复合的运算符。如果在“=”前加一个“+”运算符就成了复合运算符“+=”。例如,可以有

a+=3 等价于 a=a+3

x\*=y+8 等价于 x=x\*(y+8)

x%=3 等价于 x=x%3

以“a+=3”为例来说明,它相当于使 a 进行一次自加 3 的操作。即先使 a 加 3,再赋给 a。同样,“x\*=y+8”的作用是使 x 乘以(y+8),再赋给 x。

为便于记忆,可以这样理解:

① a+= b(其中 a 为变量,b 为表达式)

② a+= b(将下划线的“a+”移到“=”右侧)

③ a = a + b (在“=”左侧补上变量名 a)

注意,如果 b 是包含若干项的表达式,则相当于它有括号。如

① x %= y+3

② x %=(y+3)

③ x = x%(y+3)(不要错认为 x=x%y+3)





凡是二元(二目)运算符,都可以与赋值符一起组合成复合赋值符。C++可以使用以下几种复合赋值运算符:

`+=, -=, *=, /=, %=, <<=, >>=, &=, ^=, |=`

其中后 5 种是有关位运算的。

C++之所以采用这种复合运算符,一是为了简化程序,使程序精炼,二是为了提高编译效率(这样写法与“逆波兰”式一致,有利于编译,能产生质量较高的目标代码)。专业的程序员在程序中常用复合运算符,初学者可能不习惯,也可以不用或少用。

#### 2.6.4 赋值表达式

由赋值运算符将一个变量和一个表达式连接起来的式子称为“赋值表达式”。

它的一般形式为

`<变量> <赋值运算符> <表达式>`

如“`a=5`”是一个赋值表达式。对赋值表达式求解的过程是:先求赋值运算符右侧的“表达式”的值,然后赋给赋值运算符左侧的变量。一个表达式应该有一个值。赋值运算符左侧的标识符称为“左值”(left value,简写为 lvalue)。并不是任何对象都可以作为左值的,变量可以作为左值,而表达式 `a+b` 就不能作为左值,常变量也不能作为左值,因为常变量不能被赋值。

出现在赋值运算符右侧的表达式称为“右值”(right value,简写为 rvalue)。显然左值也可以出现在赋值运算符右侧,因而左值都可以作为右值。如

```
int a=3,b,c;
```

```
b=a; // b 是左值
```

```
c=b; // b 也是右值
```

赋值表达式中的“表达式”,又可以是一个赋值表达式。如

```
a=(b=5)
```

下面是赋值表达式的例子:

```
a=b=c=5 (赋值表达式值为 5,a,b,c 值均为 5)
```

```
a=5+(c=6)(表达式值为 11,a 值为 11,c 值为 6)
```

```
a=(b=4)+(c=6) (表达式值为 10,a 值为 10,b 等于 4,c 等于 6)
```

```
a=(b=10)/(c=2) (表达式值为 5,a 等于 5,b 等于 10,c 等于 2)
```

请分析下面的赋值表达式:

```
(a=3*5)=4*3
```

赋值表达式作为左值时应加括号,如果写成下面这样就会出现语法错误:

```
a=3*5=4*3
```

因为 `3*5` 不是左值,不能出现在赋值运算符的左侧。

赋值表达式也可以包含复合的赋值运算符。如

```
a+=a-=a*a
```

也是一个赋值表达式。如果 `a` 的初值为 12,此赋值表达式的求解步骤如下:

① 先进行“`a-=a*a`”的运算,它相当于 `a=a-a*a=12-144=-132`。

② 再进行“`a+=-132`”的运算,它相当于 `a=a+(-132)=-132-132=-264`。

#### 2.7 逗号运算符与逗号表达式

C++将赋值表达式作为表达式的一种,使赋值操作不仅可以出现在赋值语句中,而且可以以表达式形式出现在其他语句(如输出语句、循环语句等)中。这是 C++语言灵活性的一种表现。



请注意,用 `cout` 语句输出一个赋值表达式的值时,要将该赋值表达式用括号括起来,如果写成“`cout<<a=b;`”将会出现编译错误。

C++提供一种特殊的运算符——逗号运算符。用它将两个表达式连接起来。如

`3+5,6+8`

称为逗号表达式,又称为“顺序求值运算符”。逗号表达式的一般形式为

**表达式 1,表达式 2**

逗号表达式的求解过程是:先求解表达式 1,再求解表达式 2。整个逗号表达式的值是表达式 2 的值。如,逗号表达式

`a=3*5,a*4`

从附录 B 可知:赋值运算符的优先级别高于逗号运算符,因此应先求解 `a=3*5`(也就是把“`a=3*5`”作为一个表达式)。经计算和赋值后得到 `a` 的值为 15,然后求解 `a*4`,得 60。整个逗号表达式的值为 60。

一个逗号表达式又可以与另一个表达式组成一个新的逗号表达式,如

`(a=3*5,a*4),a+5`

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`

它的值为表达式 `n` 的值。

从附录 B 可知,逗号运算符是所有运算符中级别最低的。因此,下面两个表达式的作用是不同的:

① `x=(a=3,6*3)`

② `x=a=3,6*a`

其实,逗号表达式无非是把若干个表达式“串联”起来。在许多情况下,使用逗号表达式的目的只是想分别得到各个表达式的值,而并非一定需要得到和使用整个逗号表达式的值,逗号表达式最常用于循环语句(`for` 语句)中,详见第 3 章。

在用 `cout` 输出一个逗号表达式的值时,要将该逗号表达式用括号括起来,如

`cout<<(3*5,43-6*5,67/3)<<endl;`

C 和 C++语言表达能力强,其中一个重要方面就在于它的表达式类型丰富,运算符功能强,因而使用灵活,适应性强。

## 第 2 篇 面向过程的程序设计

第 3 章 程序设计初步

第 4 章 函数与预处理

第 5 章 数组

第 6 章 指针

第 7 章 自定义数据类型



## 第 3 章 程序设计初步

- 3.1 面向过程的程序设计和算法
- 3.2 C++程序和语句
- 3.3 赋值语句
- 3.4 C++的输入与输出
- 3.5 编写顺序结构的程序
- 3.6 关系运算和逻辑运算
- 3.7 选择结构和 if 语句
- 3.8 条件运算符和条件表达式
- 3.9 多分支选择结构和 switch 语句
- 3.10 编写选择结构的程序
- 3.11 循环结构和循环语句
- 3.12 循环的嵌套
- 3.13 break 语句和 continue 语句
- 3.14 编写循环结构的程序

### 3.1 面向过程的程序设计和算法

在面向过程的程序设计中,程序设计者必须指定计算机执行的具体步骤,程序设计者不仅要考虑程序要“做什么”,还要解决“怎么做”的问题,根据程序要“做什么”的要求,写出一个个语句,安排好它们的执行顺序。怎样设计这些步骤,怎样保证它的正确性和具有较高的效率,这就是算法需要解决的问题。

### 3.1.1 算法的概念

一个面向过程的程序应包括以下两方面内容:

- (1) 对数据的描述。在程序中要指定数据的类型和数据的组织形式,即数据结构(data structure)。
- (2) 对操作的描述。即操作步骤,也就是算法(algorithm)。

对于面向过程的程序,可以用下面的公式表示:

程序=算法+数据结构

作为程序设计人员,必须认真考虑和设计数据结构和操作步骤(即算法)。

算法是处理问题的一系列的步骤。算法必须具体地指出在执行时每一步应当怎样做。

不要认为只有“计算”的问题才有算法。广义地说,为解决一个问题而采取的方法和步骤,就称为“算法”。计算机算法可分为两大类别:数值算法和非数值算法。数值算法的目的是求数值解。非数值算法包括的面十分广泛,最常见的是用于事务管理领域。目前,计算机在非数值方面的应用远远超过了在数值方面的应用。C++既支持面向过程的程序设计,又支持面向对象的程序设计。无论面向过程的程序设计还是面向对象的程序设计,都离不开算法设计。

### 3.1.2 算法的表示

#### 1. 自然语言

用中文或英文等自然语言描述算法。但容易产生歧义性,在程序设计中一般不用自然语言表示算法。

#### 2. 流程图

可以用传统的流程图或结构化流程图。用图的形式表示算法,比较形象直观,但修改算法时显得不大方便。

#### 3. 伪代码(pseudo code)

伪代码是用介于自然语言和计算机语言之间的文字和符号来描述算法。如

**if x is positive then**

**print x**

**else**

**print-x**

用伪代码写算法并无固定的、严格的语法规则,只需把意思表达清楚,并且书写的格式要写成清晰易读的形式。它不用图形符号,因此书写方便、格式紧凑,容易修改,便于向计算机语言算法(即程序)过渡。

#### 4. 用计算机语言表示算法

用一种计算机语言去描述算法,这就是计算机程序。

### 3.2 C++程序和语句

由第 1 章已知,一个程序包含一个或多个程序单位(每个程序单位构成一个程序文件)。每一个程序单位由以下几个部分组成:

- (1) 预处理命令。如`#include` 命令和`#define` 命令。
- (2) 声明部分。例如对数据类型和函数的声明,以及对变量的定义。
- (3) 函数。包括函数头部和函数体,在函数体中可以包含若干声明语句和执行语句。

如下面是一个完整的 C++程序:



```

#include <iostream> //预处理命令
using namespace std; //在函数之外的声明部分
int a=3; //在函数之外的声明部分
int main() //函数首部
{ float b; //函数内的声明部分
b=4.5; //执行语句
cout<<a<<b; //执行语句
return 0; //执行语句
}

```

如果一个变量在函数之外进行声明,此变量是全局变量,它的有效范围是从该行开始到本程序单位结束。如果一个变量在函数内声明,此变量是局部变量,它的有效范围是从该行开始到本函数结束。C++程序结构可以用图 3.1 表示。

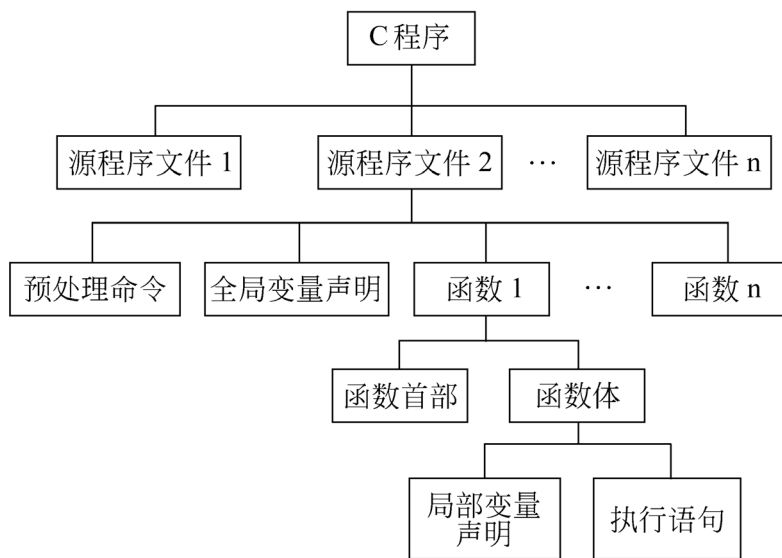


图 3.1

程序应该包括数据描述(由声明语句来实现)和数据操作(由执行语句来实现)。数据描述主要包括数据类型的声明、函数和变量的定义、变量的初始化等。数据操作的任务是对已提供的数据进行加工。

C++程序中最小的独立单位是语句(statement)。它相当于一篇文章中的一个句子。句子是用句号结束的。语句一般是用分号结束的(复合语句是以右花括号结束的)。

C++语句可以分为以下 4 种:

## 1. 声明语句

如 `int a,b;`在 C 语言中,只有产生实际操作的才称为语句,对变量的定义不作为语句,而且要求对变量的定义必须出现在本块中所有程序语句之前。因此 C 程序员已经养成了一个习惯: 在函数或块的开头位置定义全部变量。在 C++中,对变量(以及其他对象)的定义被认为是一条语句,并且可以出现在函数中的任何行,即可以放在其他程序语句可以出现的地方,也可以放在函数之外。这样更加灵活,可以很方便地实现变量的局部化(变量的作用范围从声明语句开始到本函数或本块结束)。

## 2. 执行语句

通知计算机完成一定的操作。执行语句包括：

(1) 控制语句,完成一定的控制功能。C++有 9 种控制语句,即

- ① **if()**~**else**~ (条件语句)
- ② **for()**~ (循环语句)
- ③ **while()**~ (循环语句)
- ④ **do**~**while()** (循环语句)
- ⑤ **continue**(结束本次循环语句)
- ⑥ **break**(中止执行 **switch** 或循环语句)
- ⑦ **switch** (多分支选择语句)
- ⑧ **goto** (转向语句)
- ⑨ **return** (从函数返回语句)

(2) 函数和流对象调用语句。函数调用语句由一次函数调用加一个分号构成一个语句,例如

`sort(x,y,z);` //假设已定义了 `sort` 函数,它有 3 个参数

`cout<<x<<endl;` //流对象调用语句

(3) 表达式语句。由一个表达式加一个分号构成一个语句。最典型的是: 由赋值表达式构成一个赋值语句。

`i=i+1` //是一个赋值表达式

`i=i+1;` //是一个赋值语句

任何一个表达式的最后加一个分号都可以成为一个语句。一个语句必须在最后出现分号。

表达式能构成语句是 C 和 C++ 语言的一个重要特色。C++ 程序中大多数语句是表达式语句(包括函数调用语句)。

### 3. 空语句

下面是一个空语句:

```
;
```

即只有一个分号的语句,它什么也不做。有时用来做被转向点,或循环语句中的循环体。



#### 4. 复合语句

可以用 {}把一些语句括起来成为复合语句。如下面是一个复合语句。

```
{ z=x+y;  
if(z>100) z=z-100;  
cout<<z;  
}
```

注意:复合语句中最后一个语句中最后的分号不能省略。

在本章中将介绍几种顺序执行的语句,在执行这些语句的过程中不会发生流程的控制转移。

### 3.3 赋值语句

前面已介绍,赋值语句是由赋值表达式加上一个分号构成。

(1)C++的赋值语句具有其他高级语言的赋值语句的功能。但不同的是: C++中的赋值号“=”是一个运算符,可以写成

**a=b=c=d;**

而在其他大多数语言中赋值号不是运算符,上面的写法是不合法的。

(2) 关于赋值表达式与赋值语句的概念。在 C++ 中,赋值表达式可以包括在其他表达式之中,例如

```
if((a=b)>0) cout<<"a>0"<<endl;
```

按语法规则 if 后面的()内是一个条件。现在在 x 的位置上换上一个赋值表达式“a=b”,其作用是:先进行赋值运算(将 b 的值赋给 a),然后判断 a 是否大于 0,如大于 0,执行 cout<<"a>0"<<endl;。在 if 语句中的“a=b”不是赋值语句而是赋值表达式,这样写是合法的。不能写成

```
if((a=b;)>0) cout<<"a>0"<<endl;
```

因为在 if 的条件中不能包含赋值语句。C++ 把赋值语句和赋值表达式区别开来,增加了表达式的种类,能实现其他语言中难以实现的功能。

### 3.4 C++的输入与输出

输入和输出并不是 C++语言中的正式组成成分。C 和 C++本身都没有为输入和输出提供专门的语句结构。输入输出不是由 C++本身定义的,而是在编译系统提供的 I/O 库中定义的。

C++的输出和输入是用“流”(stream)的方式实现的。图 3.2 和图 3.3 表示 C++通过流进行输入输出的过程。

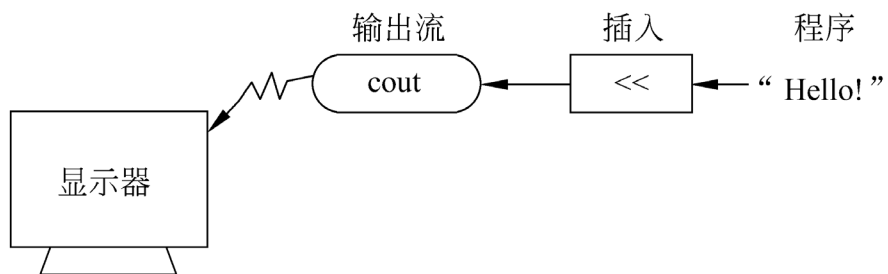


图 3.2

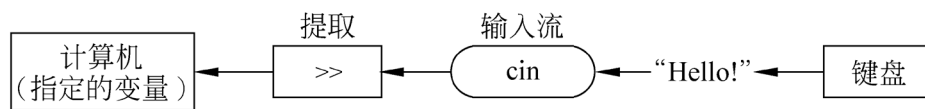


图 3.3

有关流对象 `cin`、`cout` 和流运算符的定义等信息是存放在 C++ 的输入输出流库中的,因此如果在程序中使用 `cin`、`cout` 和流运算符,就必须使用预处理命令把头文件 `stream` 包含到本文件中:

```
#include <iostream>
```

尽管 `cin` 和 `cout` 不是 C++ 本身提供的语句,但是在不致混淆的情况下,为了叙述方便,常常把由 `cin` 和流提取运算符“>>”实现输入的语句称为输入语句或 `cin` 语句,把由 `cout` 和流插入运算符“<<”实现输出的语句称为输出语句或 `cout` 语句。根据 C++ 的语法,凡是能实现某种操作而且最后以分号结束的都是语句。

### \*3.4.1 输入流与输出流的基本操作

cout 语句的一般格式为

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

cin 语句的一般格式为

```
cin>>变量 1>>变量 2>>.....>>变量 n;
```

在定义流对象时,系统会在内存中开辟一段缓冲区,用来暂存输入输出流的数据。在执行 cout 语句时,先把插入的数据顺序存放在输出缓冲区中,直到输出缓冲区满或遇到 cout 语句中的 endl(或'\n',ends,flush)为止,此时将缓冲区中已有的数据一起输出,并清空缓冲区。输出流中的数据在系统默认的设备(一般为显示器)输出。

。

一个 cout 语句可以分写成若干行。如

```
cout<<"This is a simple C++ program."<<endl;
```

可以写成

```
cout<<"This is " //注意行末尾无分号
```

```
<<"a C++ "
```

```
<<"program."
```

```
<<endl; //语句最后有分号
```

也可写成多个 cout 语句,即

```
cout<<"This is "; //语句末尾有分号
```

```
cout <<"a C++ ";
```

```
cout <<"program.";
```

```
cout<<endl;
```

以上 3 种情况的输出均为



This is a simple C++ program.

注意 不能用一个插入运算符“<<”插入多个输出项:

```
cout<<a,b,c; //错误,不能一次插入多项
```

```
cout<<a+b+c; //正确,这是一个表达式,作为一项
```

在用 cout 输出时,用户不必通知计算机按何种类型输出,系统会自动判别输出数据的类型,使输出的数据按相应的类型输出。如已定义 a 为 int 型,b 为 float 型,c 为 char 型,则

```
cout<<a<<' '<<b<<' '<<c<<endl;
```

会以下面的形式输出:

```
4 345.789 a
```

与 cout 类似,一个 cin 语句可以分写成若干行。如

```
cin>>a>>b>>c>>d;
```

可以写成

```
cin>>a //注意行末尾无分号  
>>b //这样写可能看起来清晰些  
>>c  
>>d;
```

也可以写成

```
cin>>a;  
cin>>b;  
cin>>c;  
cin>>d;
```

以上 3 种情况均可以从键盘输入: 1 2 3 4 ↵

也可以分多行输入数据:

1 ✓

2 3 ✓

4 ✓

在用 cin 输入时,系统也会根据变量的类型从输入流中提取相应长度的字节。如有

```
char c1,c2;
```

```
int a;
```

```
float b;
```

```
cin>>c1>>c2>>a>>b;
```

如果输入

1234 56.78 ✓

注意: 34 后面应该有空格以便和 56.78 分隔开。也可以按下面格式输入:

1 2 34 56.78 ✓ (在 1 和 2 之间有空格)

不能用 `cin` 语句把空格字符和回车换行符作为字符输入给字符变量,它们将被跳过。如果想将空格字符或回车换行符(或任何其他键盘上的字符)输入给字符变量,可以用 3.4.3 节介绍的 `getchar` 函数。

在组织输入流数据时,要仔细分析 `cin` 语句中变量的类型,按照相应的格式输入,否则容易出错。

#### \*3.4.2 在输入流与输出流中使用控制符

上面介绍的是使用 `cout` 和 `cin` 时的默认格式。但有时人们在输入输出时有一些特殊的要求,如在输出实数时规定字段宽度,只保留两位小数,数据向左或向右对齐等。C++提供了在输入输出流中使用的控制符(有的书中称为操纵符),见书中表 3.1。

需要注意的是:如果使用了控制符,在程序单位的开头除了要加 `iostream` 头文件外,还要加 `iomanip` 头文件。  
举例:输出双精度数。

**double** a=123.456789012345;对 a 赋初值

(1) `cout<<a;`输出: 123.456

(2) `cout<<setprecision(9)<<a;`输出: 123.456789

(3) `cout<<setprecision(6);`恢复默认格式(精度为 6)

(4) `cout<< setiosflags(ios::fixed);`输出: 123.456789

(5) `cout<<setiosflags(ios::fixed)<<setprecision(8)<<a;`输出: 123.45678901

(6) `cout<<setiosflags(ios::scientific)<<a;`输出: 1.234568e+02

(7) `cout<<setiosflags(ios::scientific)<<setprecision(4)<<a;` 输出: 1.2346e02

下面是整数输出的例子:

**int** b=123456;对 b 赋初值

(1) `cout<<b;`输出: 123456

(2) `cout<<hex<<b;` 输出: 1e240

(3) `cout<<setiosflags(ios::uppercase)<<b;`输出: 1E240

(4) `cout<<setw(10)<<b<<','<<b;` 输出: 123456,123456

(5) `cout<<setfill('*')<<setw(10)<<b;`输出: \*\*\*\* 123456

(6) `cout<<setiosflags(ios::showpos)<<b;`输出: +123456

如果在多个 `cout` 语句中使用相同的 `setw(n)`,并使用 `setiosflags(ios::right)`,可以实现各行数据右对齐,如果指定相同的精度,可以实现上下小数点对齐。

例 3.1 各行小数点对齐。

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    double a=123.456,b=3.14159,c=-3214.67;
    cout<<setiosflags(ios::fixed)<<setiosflags(ios::right)<<setprecision(2);
    cout<<setw(10)<<a<<endl;
    cout<<setw(10)<<b<<endl;
    cout<<setw(10)<<c<<endl;
    return 0;
}
```

输出如下:

123.46 (字段宽度为 10,右对齐,取两位小数)

3.14

-3214.67

先统一设置定点形式输出、取两位小数、右对齐。这些设置对其后的输出均有效(除非重新设置),而 `setw` 只对其后一个输出项有效,因此必须在输出 `a,b,c` 之前都要写 `setw(10)`。



### 3.4.3 用 `getchar` 和 `putchar` 函数进行字符的输入和输出

C++还保留了 C 语言中用于输入和输出单个字符的函数,使用很方便。其中最常用的有 `getchar` 函数和 `putchar` 函数。

#### 1. `putchar` 函数(字符输出函数)

`putchar` 函数的作用是向终端输出一个字符。例如

```
putchar(c);
```

它输出字符变量 `c` 的值。

例 3.2 输出单个字符。

```
#include <iostream> //或者包含头文件 stdio.h: #include <stdio.h>
using namespace std;
int main( )
{char a,b,c;
 a='B';b='O';c='Y';
 putchar(a);putchar(b);putchar(c);putchar('\n');
 putchar(66);putchar(79);putchar(89);putchar(10);
 return 0;
}
```

运行结果为

BOY

BOY

可以看到: 用 `putchar` 可以输出转义字符,`putchar('\n')`的作用是输出一个换行符,使输出的当前位置移到下一行的开头。`putchar(66)`的作用是将 66 作为 ASCII 码转换为字符输出,66 是字母'B'的 ASCII 码,因此 `putchar(66)`输出字母'B'。其余类似。`putchar(10)`中的 10 是换行符的 ASCII 码,`putchar(10)`输出一个换行符,作用与 `putchar('\n')` 相同。

也可以输出其他转义字符,如

`putchar('\101')`(输出字符'A',八进制的 101 是'A'的 ASCII 码)

`putchar('\')` (输出单引号字符')

`putchar('\015')`(输出回车,不换行,使输出的当前位置移到本行开头)

## 2. getchar 函数(字符输入函数)

此函数的作用是从终端(或系统隐含指定的输入设备)输入一个字符。getchar 函数没有参数,其一般形式为 getchar( )函数的值就是从输入设备得到的字符。

例 3.3 输入单个字符。

```
#include <iostream>
using namespace std;
int main( )
{char c;
  c=getchar( ); putchar(c+32); putchar('\n');
  return 0;
}
```

在运行时,如果从键盘输入大写字母'A'并按回车键,就会在屏幕上输出小写字母'a'。

请注意, `getchar()` 只能接收一个字符。 `getchar` 函数得到的字符可以赋给一个字符变量或整型变量,也可以不赋给任何变量,作为表达式的一部分。例如,例 3.3 第 5 行可以用下面一行代替:

```
putchar(getchar()+32);putchar('\n');
```

因为 `getchar()` 读入的值为 'A', 'A'+32 是小写字母 'a' 的 ASCII 码,因此 `putchar` 函数输出 'a'。此时不必定义变量 `c`。

也可用 `cout` 输出 `getchar` 函数得到字符的 ASCII 的值:

```
cout<<getchar();
```

这时输出的是整数 97,因为用 `getchar()` 读入的实际上是字符的 ASCII 码,现在并未把它赋给一个字符变量, `cout` 就按整数形式输出。如果改成

```
cout<<(c=getchar( )); //设 c 已定义为字符变量
```

则输出为字母'a',因为要求输出字符变量 c 的值。

可以看到用 putchar 和 getchar 函数输出和输入字符十分灵活方便,由于它们是函数所以可以出现在表达式中,例如

```
cout<<(c=getchar()+32);
```

#### 3.4.4 用 scanf 和 printf 函数进行输入和输出

在 C 语言中是用 printf 函数进行输出,用 scanf 函数进行输入的。C++保留了 C 语言的这一用法。在此只作很简单的介绍。

scanf 函数一般格式是

scanf(格式控制,输出表列)

printf 函数的一般格式是

printf(格式控制,输出表列)

例 3.4 用 scanf 和 printf 函数进行输入和输出。

```
#include <iostream>
using namespace std;
int main( )
{int a; float b; char c;
 scanf("%d %c %f",&a,&c,&b); //注意在变量名前要加地址运算符&
 printf("a=%d,b=%f,c=%c\n",a,b,c);
 return 0;
}
```

运行情况如下:

12 A 67.98✓ (本行为输入,输入的 3 个数据间以空格相间)

a=12,b=67.980003,c=A(本行为输出)

输入的整数 12 送给整型变量 a,字符'A'送给字符变量 c,67.98 送给单精度变量 b。



### 3.5 编写顺序结构的程序

例 3.5 求一元二次方程式  $ax^2+bx+c=0$  的根。a,b,c 的值在运行时由键盘输入,它们的值满足  $b^2-4ac \geq 0$ 。

根据求  $x_1, x_2$  的算法。它可以编写出以下 C++ 程序:

```
#include <iostream>
#include <cmath> //由于程序要用到数学函数 sqrt, 故应包含头文件 cmath
using namespace std;
int main( )
{float  a,b,c,x1,x2;
  cin>>a>>b>>c;
  x1=(-b+sqrt(b*b-4*a*c))/(2*a);
  x2=(-b-sqrt(b*b-4*a*c))/(2*a);
  cout<<"x1="<<x1<<endl;
  cout<<"x2="<<x2<<endl;
  return 0;
}
```

运行情况如下:

4.5 8.8 2.4 ✓

x1=-0.327612

x2=-1.17794

如果程序中要用到数学函数,都要包含头文件 `cmath`(也可以用老形式的头文件 `math.h`,但提倡使用 C++新形式的头文件,请参阅第 14 章 14.3 节)。在写程序时,一定要注意将数学表达式正确地转换成合法的 C++表达式。

可以看到:顺序结构的程序中的各执行语句是顺序执行的。这种程序最简单,最容易理解。

### 3.6 关系运算和逻辑运算

往往要求根据某个指定的条件是否满足来决定执行的内容。例如,购物在 1000 元以下的打九五折,1000 元及以上的打九折。

C++提供 if 语句来实现这种条件选择。如

`if amount<1000 tax=0.95; //amount 代表购物总额,tax 代表折扣`

`else tax=0.9; //若 amount<1000,条件满足,tax=0.95,否则 tax=0.9`

`pay=amount*tax; //pay 为实付款`

流程可以用图 3.4 表示。

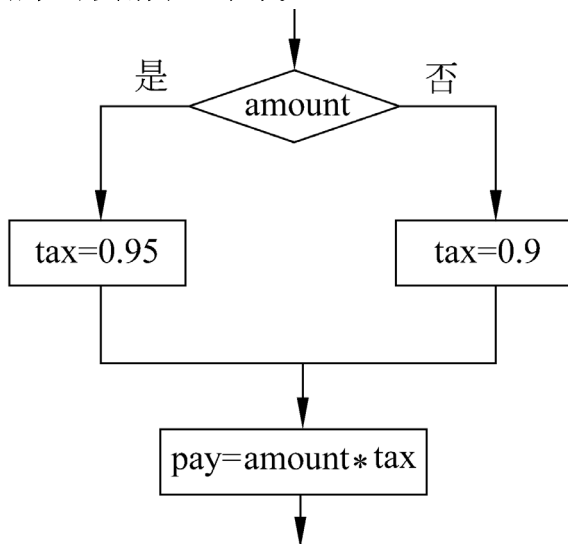


图 3.4

### 3.6.1 关系运算和关系表达式

上面 if 语句中的“amount<1000”实现的不是算术运算,而是关系运算。实际上是比较运算,将两个数据进行比较,判断比较的结果。“amount<1000”就是一个比较式,在高级语言中称为关系表达式,其中“>”是一个比较符,称为关系运算符。

C++的关系运算符有:

- ① <    (小于)
- ② <=   (小于或等于)      优先级相同 (高)
- ③ >    (大于)
- ④ >=   (大于或等于)
- ⑤ ==   (等于)
- ⑥ !=   (不等于)          优先级相同 (低)

关于优先次序:

- ① 前 4 种关系运算符(<,<=,>,>=)的优先级别相同,后两种也相同。前 4 种高于后两种。例如,“>”优先于“==”。而“>”与“<”优先级相同。
- ② 关系运算符的优先级低于算术运算符。
- ③ 关系运算符的优先级高于赋值运算符。

例如:

$c > a + b$  等效于  $c > (a + b)$

$a > b == c$  等效于  $(a > b) == c$

$a == b < c$  等效于  $a == (b < c)$

$a = b > c$  等效于  $a = (b > c)$

用关系运算符将两个表达式连接起来的式子,称为关系表达式。关系表达式的一般形式可以表示为

表达式 关系运算符 表达式

其中的“表达式”可以是算术表达式或关系表达式、逻辑表达式、赋值表达式、字符表达式。例如,下面都是合法的关系表达式:

`a>b, a+b>b+c,(a==3)>(b==5), 'a'<'b', (a>b)>(b<c)`

关系表达式的值是一个逻辑值,即“真”或“假”。例如,关系表达式“`5==3`”的值为“假”,“`5>=0`”的值为“真”。在 C 和 C++中都用数值 1 代表“真”,用 0 代表“假”。

如果有以下赋值表达式:

`d=a>b` 则 d 得到的值为 1

`f=a>b>c` f 得到的值为 0

### 3.6.2 逻辑常量和逻辑变量

C 语言没有提供逻辑型数据,关系表达式的值(真或假)分别用数值 1 和 0 代表。C++增加了逻辑型数据。逻辑型常量只有两个,即 false(假)和 true(真)。

逻辑型变量要用类型标识符 bool 来定义,它的值只能是 true 和 false 之一。如

```
bool found,flag=false; //定义逻辑变量 found 和 flag,并使 flag 的初值为 false
```

```
found=true; //将逻辑常量 true 赋给逻辑变量 found
```

由于逻辑变量是用关键字 bool 来定义的,因此又称为布尔变量。逻辑型常量又称为布尔常量。所谓逻辑型,就是布尔型。

设立逻辑类型的目的是为了看程序时直观易懂。

在编译系统处理逻辑型数据时,将 false 处理为 0,将 true 处理为 1。因此,逻辑型数据可以与数值型数据进行算术运算。

如果将一个非零的整数赋给逻辑型变量,则按“真”处理,如

```
flag=123; //赋值后 flag 的值为 true
```

```
cout<<flag;
```

输出为数值 1。



### 3.6.3 逻辑运算和逻辑表达式

有时只用一个关系表达式还不能正确表示所指定的条件。

C++提供 3 种逻辑运算符：

(1) `&&` 逻辑与(相当于其他语言中的 AND)

(2) `||` 逻辑或(相当于其他语言中的 OR)

(3) `!` 逻辑非(相当于其他语言中的 NOT)

逻辑运算举例如下：

`a && b` 若 `a, b` 为真,则 `a && b` 为真。

`a || b` 若 `a, b` 之一为真,则 `a || b` 为真。

`!a` 若 `a` 为真,则 `!a` 为假。

书中表 3.2 为逻辑运算的“真值表”。

在一个逻辑表达式中如果包含多个逻辑运算符,按以下的优先次序:

- (1) !(非)→ &&(与)→ (或),即“!”为三者中最高的。
- (2) 逻辑运算符中的“&&”和“||”低于关系运算符,“!”高于算术运算符。

例如:

$(a > b) \&\& (x > y)$  可写成  $a > b \&\& x > y$

$(a == b) || (x == y)$  可写成  $a == b || x == y$

$(!a) || (a > b)$  可写成  $!a || a > b$

将两个关系表达式用逻辑运算符连接起来就成为一个逻辑表达式,上面几个式子就是逻辑表达式。逻辑表达式的一般形式可以表示为

## 表达式 逻辑运算符 表达式

逻辑表达式的值是一个逻辑量“真”或“假”。前面已说明,在给出逻辑运算结果时,以数值 1 代表“真”,以 0 代表“假”,但在判断一个逻辑量是否为“真”时,采取的标准是: 如果其值是 0 就认为是“假”,如果其值是非 0 就认为是“真”。例如:

- (1) 若  $a=4$ ,则  $!a$  的值为 0。因为  $a$  的值为非 0,被认作“真”,对它进行“非”运算,得“假”,“假”以 0 代表。
- (2) 若  $a=4, b=5$ ,则  $a \&\& b$  的值为 1。因为  $a$  和  $b$  均为非 0,被认为是“真”。
- (3)  $a, b$  值同前,  $a-b \parallel a+b$  的值为 1。因为  $a-b$  和  $a+b$  的值都为非零值。
- (4)  $a, b$  值同前,  $!a \parallel b$  的值为 1。
- (5)  $4 \&\& 0 \parallel 2$  的值为 1。

在 C++ 中,整型数据可以出现在逻辑表达式中,在进行逻辑运算时,根据整型数据的值是 0 或非 0,把它作为逻辑量假或真,然后参加逻辑运算。

通过这几个例子可以看出:逻辑运算结果不是 0 就是 1,不可能是其他数值。而在逻辑表达式中作为参加逻辑运算的运算对象可以是 0(“假”)或任何非 0 的数值(按“真”对待)。如果在一个表达式中的不同位置上出现数值,应区分哪些是作为数值运算或关系运算的对象,哪些作为逻辑运算的对象。

实际上,逻辑运算符两侧的表达式不但可以是关系表达式或整数(0 和非 0),也可以是任何类型的数据,如字符型、浮点型或指针型等。系统最终以 0 和非 0 来判定它们属于“真”或“假”。例如 'c' && 'd' 的值为 1。

可以将表 3.2 改写成书中表 3.3 形式。

熟练掌握 C++ 的关系运算符和逻辑运算符后,可以巧妙地用一个逻辑表达式来表示一个复杂的条件。例如,要判别某一年(year)是否为闰年。闰年的条件是符合下面两者之一: ①能被 4 整除,但不能被 100 整除。②能被 100 整除,又能被 400 整除。例如 2004、2000 年是闰年,2005、2100 年不是闰年。

可以用一个逻辑表达式来表示:

```
(year % 4 == 0 && year % 100 != 0) || year % 400 == 0
```

当给定 year 为某一整数值时,如果上述表达式值为真(1),则 year 为闰年;否则 year 为非闰年。可以加一个“!”用来判别非闰年:

```
!((year % 4 == 0 && year % 100 != 0) || year % 400 == 0)
```

若表达式值为真(1),year 为非闰年。也可以用下面的逻辑表达式判别非闰年:

`(year % 4 != 0) || (year % 100 == 0 && year % 400 != 0)`

若表达式值为真,year 为非闰年。请注意表达式中右面的括号内的不同运算符(% ,! ,&& ,==)的运算优先次序。

### 3.7 选择结构和 if 语句

if 语句是用来判定所给定的条件是否满足,根据判定的结果(真或假)决定执行给出的两种操作之一。

### 3.7.1 if 语句的 3 种形式

#### 1. if(表达式)语句

例如:

```
if(x>y) cout<<x<<endl;
```

这种 if 语句的执行过程见图 3.5(a)。

#### 2. if(表达式)语句 1 else 语句 2

例如:

```
if (x>y) cout<<x;
```

```
else  cout<<y;
```

见图 3.5(b)。



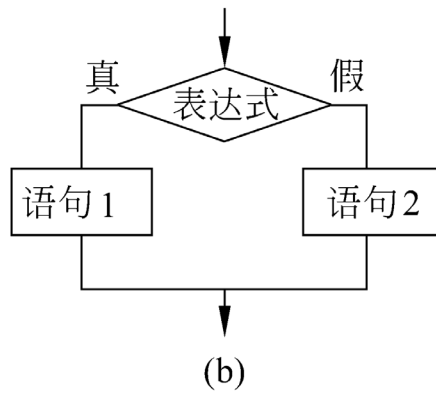
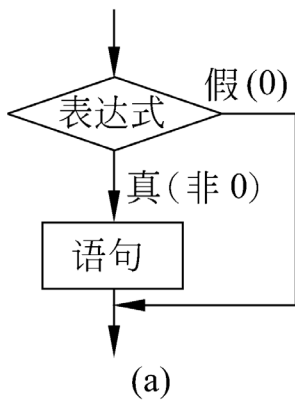


图 3.5

3. **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 流程图见图 3.6。

例如:

```
if (number>500) cost=0.15;  
else if(number>300) cost=0.10;  
else if(number>100) cost=0.075;  
else if(number>50) cost=0.05;  
else cost=0;
```

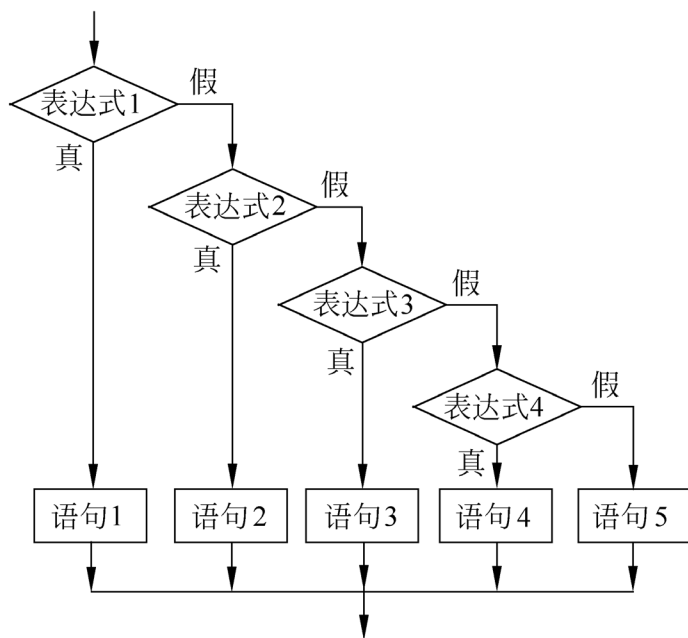


图 3.6

说明:

- (1) 从图 3.5 和图 3.6 可以看到: 3 种形式的 if 语句都是由一个入口进来,经过对“表达式”的判断,分别执行相应的语句,最后归到一个共同的出口。这种形式的程序结构称为选择结构。在 C++ 中 if 语句是实现选择结构主要的语句。
- (2) 3 种形式的 if 语句中在 if 后面都有一个用括号括起来的表达式,它是程序编写者要求程序判断的“条件”,一般是逻辑表达式或关系表达式。
- (3) 第 2、第 3 种形式的 if 语句中,在每个 else 前面有一分号,整个语句结束处有一分号。
- (4) 在 if 和 else 后面可以只含一个内嵌的操作语句(如上例),也可以有多个操作语句,此时用花括号“{”将几个语句括起来成为一个复合语句。

例 3.6 求三角形的面积。

```
#include <iostream>
#include <cmath> //使用数学函数时要包含头文件 cmath
#include <iomanip> //使用 I/O 流控制符要包含头文件 iomanip
using namespace std;
int main( )
{
    double a,b,c;
    cout<<"please enter a,b,c:";
    cin>>a>>b>>c;
    if (a+b>c && b+c>a && c+a>b)
    { //复合语句开始
        double s,area;//在复合语句内定义变量
        s=(a+b+c)/2;
        area=sqrt(s*(s-a)*(s-b)*(s-c));
        cout<<setiosflags(ios::fixed)<<setprecision(4); //指定输出的数包含 4 位小数
        cout<<"area="<<area<<endl; //在复合语句内输出局部变量的值
    } //复合语句结束
    else cout<<"it is not a trilateral!"<<endl;
    return 0;
}
```

运行情况如下:

please enter a,b,c:2.45 3.67 4.89↵

area=4.3565

变量 `s` 和 `area` 只在复合语句内用得到,因此在复合语句内定义,它的作用范围为从定义变量开始到复合语句结束。如果在复合语句外使用 `s` 和 `area`,则会在编译时出错,系统认为这两个变量未经定义。将某些变量局限在某一范围内,与外界隔离,可以避免在其他地方被误调用。

### 3.7.2 if 语句的嵌套

在 if 语句中又包含一个或多个 if 语句称为 if 语句的嵌套。一般形式如下：

```
if()  
if()语句 1  
else 语句 2    内嵌 if  
else  
if()语句 3  
else 语句 4      内嵌 if
```

应当注意 if 与 else 的配对关系。else 总是与它上面最近的、且未配对的 if 配对。假如写成

if()

if()语句 1

else

if()语句 2

else 语句 3          内嵌 if

程序员把第一个 else 写在与第一个 if(外层 if)同一列上,希望 else 与第一个 if 对应,但实际上 else 是与第二个 if 配对,因为它们相距最近,而且第二个 if 并未与任何 else 配对。为了避免误用,最好使每一层内嵌的 if 语句都包含 else 子句(如本节开头列出的形式),这样 if 的数目和 else 的数目相同,从内层到外层一一对应,不致出错。



如果 if 与 else 的数目不一样,为实现程序设计者的企图,可以加花括号来确定配对关系。例如:

```
if()
```

```
{ if () 语句 1} //这个语句是上一行 if 语句的内嵌 if
```

```
else 语句 2//本行与第一个 if 配对
```

这时{}限定了内嵌 if 语句的范围,{}外的 else 不会与{}内的 if 配对。关系清楚,不易出错。

### 3.8 条件运算符和条件表达式

若在 if 语句中,当被判别的表达式的值为“真”或“假”时,都执行一个赋值语句且给同一个变量赋值时,可以用简单的条件运算符来处理。例如,若有以下 if 语句:

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

```
else max=b;
```

可以用条件运算符(? :)来处理:

```
max=(a>b)?a:b;
```

其中“(a>b)?a:b”是一个“条件表达式”。它是这样执行的:如果(a>b)条件为真,则条件表达式的值就取“?”后面的值,即条件表达式的值为 a,否则条件表达式的值为“:”后面的值,即 b。

条件运算符要求有 3 个操作对象,称三目(元)运算符,它是 C++中惟一的一个三目运算符。条件表达式的一般形式为

表达式 1 ? 表达式 2 : 表达式 3

条件运算符的执行顺序是:先求解表达式 1,若为非 0(真)则求解表达式 2,此时表达式 2 的值就作为整个条件表达式的值。若表达式 1 的值为 0(假),则求解表达式 3,表达式 3 的值就是整个条件表达式的值。

“`max=(a>b)?a:b`”的执行结果是将条件表达式的值赋给 `max`。也就是将 `a` 和 `b` 二者中的大者赋给 `max`。条件运算符优先于赋值运算符,因此上面赋值表达式的求解过程是先求解条件表达式,再将它的值赋给 `max`。

条件表达式中,表达式 1 的类型可以与表达式 2 和表达式 3 的类型不同。如

`x? 'a' : 'b'`

如果已定义 `x` 为整型变量,若 `x=0`,则条件表达式的值为字符'b'的 ASCII 码。表达式 2 和表达式 3 的类型也可以不同,此时条件表达式的值的类型为二者中较高的类型。如有条件表达式 `x>y?1:1.5`,如果 `x≤y`,则条件表达式的值为 1.5,若 `x>y`,值应为 1,由于 C++把 1.5 按双精度数处理,双精度的类型比整型高,因此,将 1 转换成双精度数,以此作为表达式的值。

例3.7 输入一个字符,判别它是否为大写字母,如果是,将它转换成小写字母;如果不是,不转换。然后输出最后得到的字符。

```
#include <iostream>
using namespace std;
int main( )
{
    char ch;
    cin>>ch;
    ch=(ch>='A' && ch<='Z')?(ch+32):ch; //判别ch是否大写字母,是则转换
    cout<<ch<<endl;
    return 0;
}
```

### 3.9 多分支选择结构和 **switch** 语句

**switch** 语句是多分支选择语句,用来实现多分支选择结构。

它的一般形式如下:

**switch**(表达式)

{**case** 常量表达式 1:语句 1

**case** 常量表达式 2:语句 2

...

**case** 常量表达式 n:语句 n

**default**:语句 n+1

}

例如,要求按照考试成绩的等级打印出百分制分数段,可以用 switch 语句实现:

```
switch(grade)
{case 'A': cout<<"85~100\n";
case 'B': cout<<"70~84\n";
case 'C': cout<<"60~69\n";
case 'D': cout<<"<60\n";
default: cout<<"error\n";
}
```

说明:

- (1) switch 后面括号内的“表达式”,允许为任何类型。
- (2) 当 switch 表达式的值与某一个 case 子句中的常量表达式的值相匹配时,就执行此 case 子句中的内嵌语句,若所有的 case 子句中的常量表达式的值都不能与 switch 表达式的值匹配,就执行 default 子句的内嵌语句。

- (3) 每一个 case 表达式的值必须互不相同,否则就会出现互相矛盾的现象(对表达式的同一个值,有两种或多种执行方案)。
- (4) 各个 case 和 default 的出现次序不影响执行结果。例如,可以先出现“default:...”,再出现“case 'D':...”,然后是“case 'A':...”。
- (5) 执行完一个 case 子句后,流程控制转移到下一个 case 子句继续执行。“case 常量表达式”只是起语句标号作用,并不是在该处进行条件判断。在执行 switch 语句时,根据 switch 表达式的值找到与之匹配的 case 子句,从此 case 子句开始执行下去,不再进行判断。例如,上面的例子中,若 grade 的值等于'A',则将连续输出:



85~100

70~84

60~69

<60

error

因此,应该在执行一个 case 子句后,使流程跳出 switch 结构,即终止 switch 语句的执行。可以用一个 break 语句来达到此目的。将上面的 switch 结构改写如下:

```
switch(grade)
{
    case 'A': cout<<"85~100\n";break;
    case 'B': cout<<"70~84\n";break;
    case 'C': cout<<"60~69\n";break;
    case 'D': cout<<"<60\n";break;
    default: cout<<"error\n";break;
}
```

最后一个子句(**default**)可以不加 `break` 语句。如果 `grade` 的值为'B',则只输出“70~84”。流程图见图 3.7。

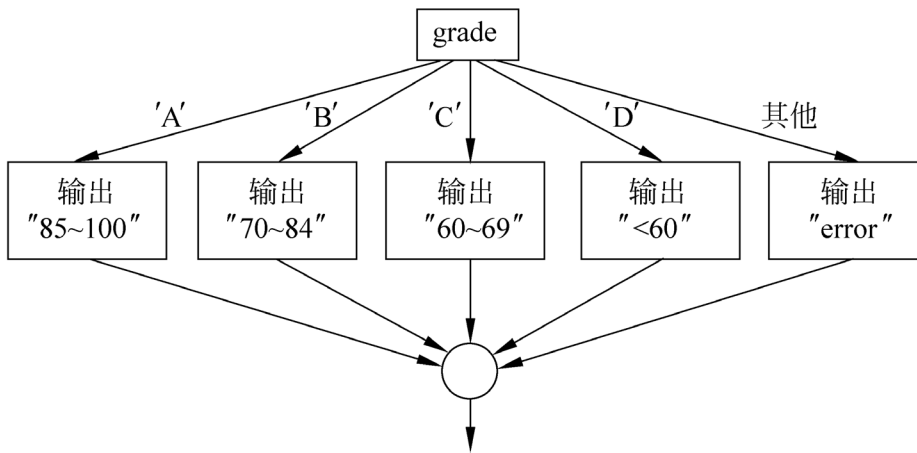


图 3.7

在 case 子句中虽然包含一个以上执行语句,但可以不必用花括号括起来,会自动顺序执行本 case 子句中所有的执行语句。

(6) 多个 case 可以共用一组执行语句,如

... :

case 'A':

case 'B':

case 'C': cout<<">60\n";break;

...

当 grade 的值为'A'、'B'或'C'时都执行同一组语句。

### 3.10 编写选择结构的程序

例 3.8 编写程序,判断某一年是否为闰年。

```
#include <iostream>
using namespace std;
int main( )
{ int year;
  bool leap;
  cout<<"please enter year:";//输出提示
  cin>>year; //输入年份
  if (year%4==0) //年份能被 4 整除
  {if(year%100==0)//年份能被 4 整除又能被 100 整除
  {if (year%400==0)//年份能被 4 整除又能被 400 整除
  leap=true;//闰年,令 leap=true(真)
  else leap=false;} //非闰年,令 leap=false(假)
  else //年份能被 4 整除但不能被 100 整除肯定是闰年
  leap=true;} //是闰年,令 leap=true
```

```

    else //年份不能被4整除肯定不是闰年
    leap=false; //若为非闰年,令 leap=false
    if (leap)
        cout<<year<<" is "; //若 leap 为真,就输出年份和“是”
    else
        cout<<year<<" is not "; //若 leap 为真,就输出年份和“不是”
    cout<<" a leap year."<<endl; //输出“闰年”
    return 0;
}

```

运行情况如下:

① 2005✓

2005 is not a leap year.

② 1900✓

1900 is npt a leap year.

也可以将程序中第 8~16 行改写成以下的 if 语句:

```
if(year%4!=0)
    leap=false;
else if(year%100!=0)
    leap=true;
else if(year%400!=0)
    leap=false;
else
    leap=true;
```

也可以用一个逻辑表达式包含所有的闰年条件,将上述 if 语句用下面的 if 语句代替:

```
if((year%4 == 0 && year%100 !=0) || (year%400 == 0)) leap=true;
else leap=false;
```

例 3.9 运输公司对用户计算运费。路程(s)越远,每公里运费越低。标准如下:

$s < 250\text{km}$  没有折扣

$250 \leq s < 500$  2%折扣

$500 \leq s < 1000$  5%折扣

$1000 \leq s < 2000$  8%折扣

$2000 \leq s < 3000$  10%折扣

$3000 \leq s$  15%折扣

设每公里每吨货物的基本运费为 p(price 的缩写),货物重为 w(wright 的缩写),距离为 s,折扣为 d(discount 的缩写),则总运费 f(freight 的缩写)的计算公式为

$$f = p * w * s * (1 - d)$$

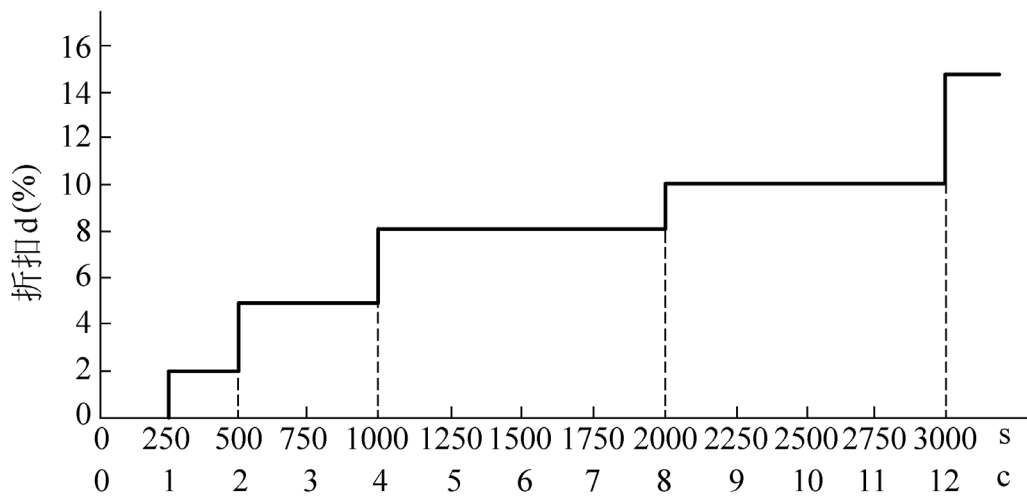


图 3.8



据此写出程序如下:

```
#include <iostream>
using namespace std;
int main( )
{int c,s;
 float p,w,d,f;
 cout<<"please enter p,w,s:";
 cin>>p>>w>>s;
 if(s>=3000) c=12;
 else c=s/250;
 switch (c)
 { case 0:d=0;break;
 case 1:d=2;break;
 case 2:
 case 3:d=5;break;
 case 4:
```

```
case 5:
case 6:
case 7:d=8;break;
case 8:
case 9:
case 10:
case 11:d=10;break;
case 12:d=15;break;
}
f=p*w*s*(1-d/100.0);
cout<<"freight="<<f<<endl;
return 0;
}
```

运行情况如下:

```
please enter p,w,s:100 20 300↵
freight=588000
```

### 3.11 循环结构和循环语句

在人们所要处理的问题中常常遇到需要反复执行某一操作的情况。这就需要用到循环控制。许多应用程序都包含循环。顺序结构、选择结构和循环结构是结构化程序设计的 3 种基本结构,是各种复杂程序的基本构造单元。因此程序设计者必须熟练掌握选择结构和循环结构的概念及使用方法。

### 3.11.1 用 while 语句构成循环

while 语句的一般形式如下：

**while** (表达式) 语句

其作用是：当指定的条件为真(表达式为非0)时,执行 while 语句中的内嵌语句。其流程图见图 3.9。其特点是：先判断表达式,后执行语句。while 循环称为当型循环。

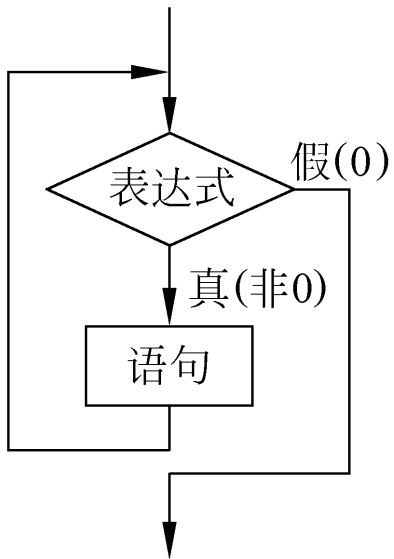


图 3.9

例 3.10 求  $1+2+3+\dots+100$ 。  
用流程图表示算法,见图 3.10。  
根据流程图写出程序:

```
#include <iostream>
using namespace std;
int main( )
{int i=1,sum=0;
while (i<=100)
{ sum=sum+i;
i++;
}
cout<<"sum="<<sum<<endl;
}
```

运行结果为

sum=5050

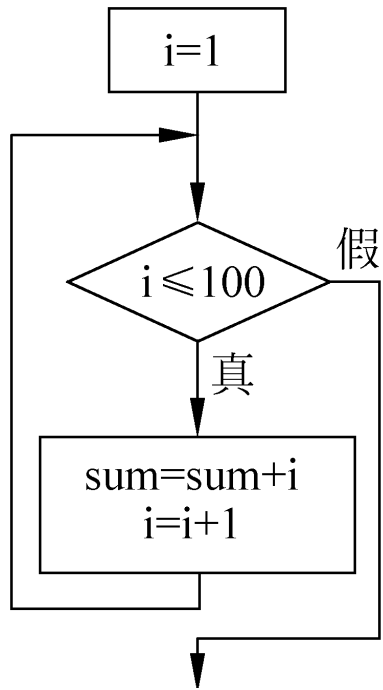


图 3.10

需要注意:

- (1) 循环体如果包含一个以上的语句,应该用花括号括起来,以复合语句形式出现。如果不加花括号,则 `while` 语句的范围只到 `while` 后面第一个分号处。
- (2) 在循环体中应有使循环趋向于结束的语句。

### 3.11.2 用 do-while 语句构成循环

do-while 语句的特点是先执行循环体,然后判断循环条件是否成立。其一般形式为

**do**

语句

**while** (表达式);

它是这样执行的:先执行一次指定的语句(即循环体),然后判别表达式,当表达式的值为非零(“真”) 时,返回重新执行循环体语句,如此反复,直到表达式的值等于 0 为止,此时循环结束。可以用图 3.11 表示其流程。

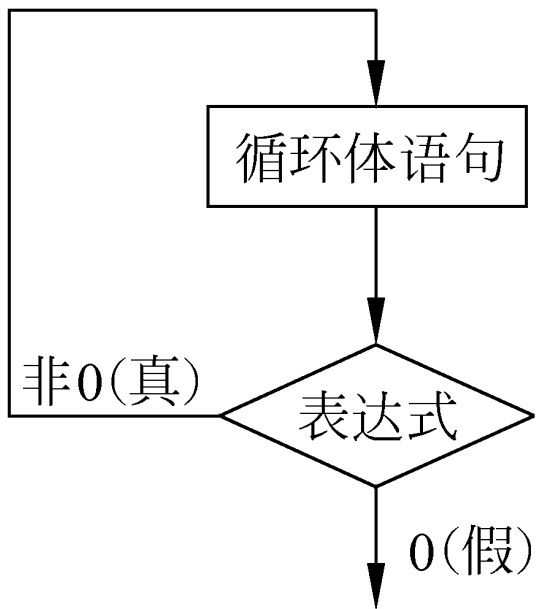


图 3.11

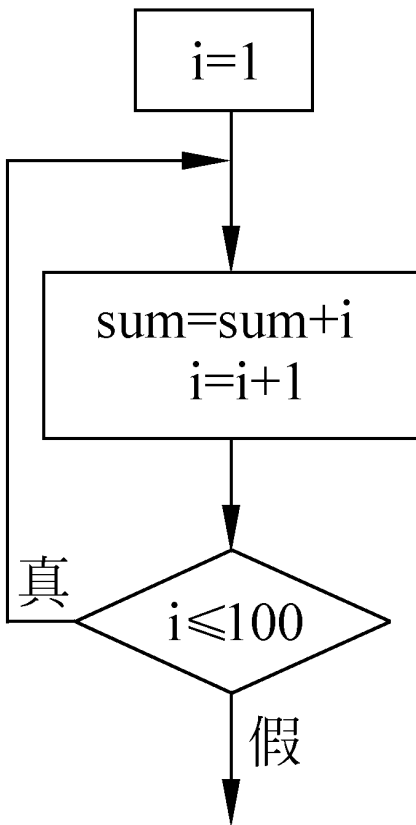


图 3.12



例 3.11 用 do-while 语句求  $1+2+3+\dots+100$ 。

先画出流程图,见图 3.12。

可编写出下面的程序:

```
#include <iostream>
using namespace std;
int main( )
{int i=1,sum=0;
  do
  { sum=sum+i;
    i++;
  }while (i<=100);
  cout<<"sum="<<sum<<endl;
  return 0;
}
```

运行结果与例 3.10 相同。

可以看到:对同一个问题可以用 **while** 语句处理,也可以用 **do while** 语句处理。**do while** 语句结构可以转换成 **while** 结构。图 3.11 可以改画成图 3.13 的形式,二者完全等价。而图 3.13 中虚线框部分就是一个 **while** 结构。

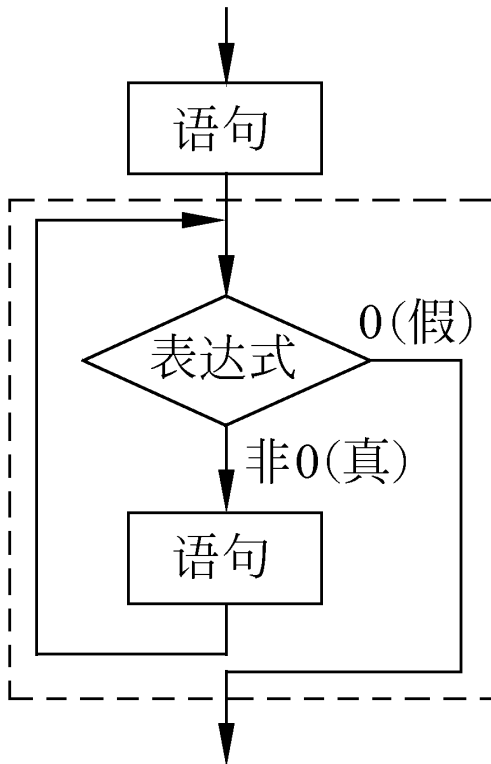


图 3.13

### 3.11.3 用 for 语句构成循环

C++中的 for 语句使用最为广泛和灵活,不仅可以用于循环次数已经确定的情况,而且可以用于循环次数不确定而只给出循环结束条件的情况,它完全可以代替 while 语句。

for 语句的一般格式为

**for**(表达式 1;表达式 2;表达式 3) 语句

它的执行过程如下:

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

- (3) 求解表达式 3。
  - (4) 转回上面第(2)步骤继续执行。
  - (5) 循环结束,执行 for 语句下面的一个语句。
- 可以用图 3.14 来表示 for 语句的执行过程。

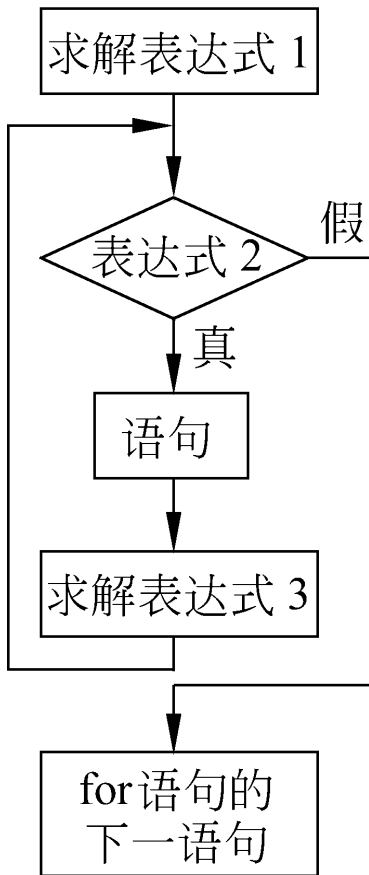


图 3.14

for 语句最简单的形式也是最容易理解的格式如下:

**for**(循环变量赋初值;循环条件;循环变量增值) 语句  
例如

```
for(i=1;i<=100;i++) sum=sum+i;
```

它的执行过程与图 3.10 完全一样。它相当于以下语句:

```
    i=1;  
while(i<=100)  
{sum=sum+i;  
  i++;  
}
```

显然,用 for 语句简单、方便。

for 语句的使用有许多技巧,如果熟练地掌握和运用 for 语句,可以使程序精炼简洁。

说明:

- (1) for 语句的一般格式中的“表达式 1”可以省略,此时应在 for 语句之前给循环变量赋初值。
- (2) 如果表达式 2 省略,即不判断循环条件,循环无终止地进行下去。也就是认为表达式 2 始终为真。(3) 表达式 3 也可以省略,但此时程序设计者应另外设法保证循环能正常结束。
- (4) 可以省略表达式 1 和表达式 3,只有表达式 2,即只给循环条件。
- (5) 3 个表达式都可省略。
- (6) 表达式 1 可以是设置循环变量初值的赋值表达式,也可以是与循环变量无关的其他表达式。

(7) 表达式一般是关系表达式(如 `i<=100`)或逻辑表达式(如 `a<b && x<y`),但也可以是数值表达式或字符表达式,只要其值为非零,就执行循环体。

C++中的 `for` 语句比其他语言中的循环语句功能强得多。可以把循环体和一些与循环控制无关的操作也作为表达式 1 或表达式 3 出现,这样程序可以短小简洁。但过分地利用这一特点会使 `for` 语句显得杂乱,可读性降低,建议不要把与循环控制无关的内容放到 `for` 语句中。

#### 3.11.4 几种循环的比较

(1) 3 种循环都可以用来处理同一问题,一般情况下它们可以互相代替。

(2) `while` 和 `do-while` 循环,是在 `while` 后面指定循环条件的,在循环体中应包含使循环趋于结束的语句(如 `i++`, 或 `i=i+1` 等)。

`for` 循环可以在表达式 3 中包含使循环趋于结束的操作,甚至可以将循环体中的操作全部放到表达式 3 中。因此 `for` 语句的功能更强,凡用 `while` 循环能完成的,用 `for` 循环都能实现。

(3) 用 `while` 和 `do-while` 循环时,循环变量初始化的操作应在 `while` 和 `do-while` 语句之前完成。而 `for` 语句可以在表达式 1 中实现循环变量的初始化。



### 3.12 循环的嵌套

一个循环体内又包含另一个完整的循环结构,称为循环的嵌套。内嵌的循环中还可以嵌套循环,这就是多层循环。

3 种循环(`while` 循环、`do while` 循环和 `for` 循环)可以互相嵌套。例如,下面几种都是合法的形式:

(1)

```
while()  
{  
  |  
  while()  
  {...}  
}
```

(2)

**do**

{ |

**do**

{...}**while**();

}**while**();

(3)

**for**(;;)

{ |

**for**(;;)

{...}

}

(4)

**while()**

{ |

**do**

{...}**while()**;

|

}

(5)

**for(;; )**

{ |

**while()**

{...}

|

}

(6)

**do**

{ |

**for** (;;)

{...}

**}while**();

### 3.13 break 语句和 continue 语句

在 3.9 节中已经介绍过用 **break** 语句可以使流程跳出 **switch** 结构,继续执行 **switch** 语句下面的一个语句。实际上,**break** 语句还可以用于循环体内。

**break** 语句的一般格式为

**break;**

其作用为使流程从循环体内跳出循环体,即提前结束循环,接着执行循环体下面的语句。**break** 语句只能用于循环语句和 **switch** 语句内,不能单独使用或用于其他语句中。

continue 语句的一般格式为

**continue;**

其作用为结束本次循环,即跳过循环体中下面尚未执行的语句,接着进行下一次是否执行循环的判定。

continue 语句和 break 语句的区别是:continue 语句只结束本次循环,而不是终止整个循环的执行。而 break 语句则是结束整个循环过程,不再判断执行循环的条件是否成立。如果有以下两个循环结构:

(1)

**while**(表达式 1)

{  
|

**if**(表达式 2)**break**

|

}

(2)

```
while(表达式 1 )  
{  
  |  
  if(表达式 2) continue;  
  |  
  |  
}
```

程序(1)的流程图如图 3.18 所示,而程序(2)的流程如图 3.19 所示。请注意图 3.18 和图 3.19 中当“表达式 2”为真时流程的转向。

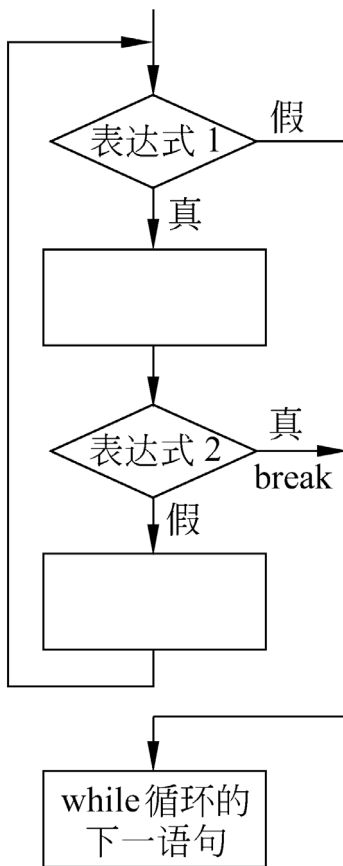


图 3.18

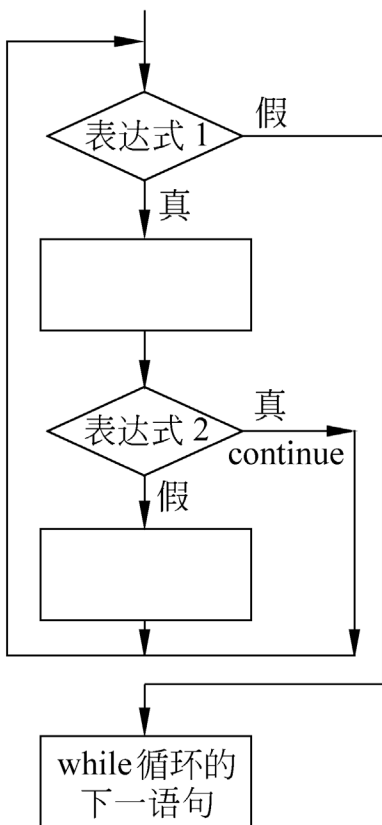


图 3.19



### 3.14 编写循环结构的程序

例 3.12 用下面公式求  $\pi$  的近似值。 $\pi/4 \approx 1 - 1/3 + 1/5 - 1/7 + \dots$  直到最后一项的绝对值小于  $10^{-7}$  为止。

根据给定的算法很容易编写程序如下：

```
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;
int main()
{int s=1;
 double n=1,t=1,pi=0;
 while(fabs(t)>1e-7)
 {pi=pi+t;
  n=n+2;
```

```
s=s;  
t=s/n;  
}  
pi=pi*4;  
cout<<"pi="<<setiosflags(ios::fixed)<<setprecision(6)<<pi<<endl;  
return 0;  
}
```

运行结果为

**pi=3.141592**

注意：不要把 **n** 定义为整型变量,否则在执行“**t=s/n;**”时,得到 **t** 的值为 **0**(原因是两个整数相除)。

例 3.13 求 Fibonacci 数列前 40 个数。这个数列有如下特点:第 1、2 个数为 1、1。从第 3 个数开始,每个数是其前面两个数之和。即

$$F_1=1 \quad (n=1)$$

$$F_2=1 \quad (n=2)$$

$$F_n=F_{n-1}+F_{n-2} \quad (n \geq 3)$$

这是一个有趣的古典数学问题:有一对兔子,从出生后第 3 个月起每个月都生一对兔子。小兔子长到第 3 个月后每个月又生一对兔子。假设所有兔子都不死,问每个月的兔子总数为多少?

可以从书中表 3.4 看出兔子数的规律。

根据给出的每月兔子总数的关系,可编写程序如下:

```

#include <iostream>
#include <iomanip>
using namespace std;
int main( )
{long f1,f2;
  int i;
  f1=f2=1;
  for(i=1;i<=20;i++)
  {cout<<setw(12)<<f1<<setw(12)<<f2;
    //设备输出字段宽度为12,每次输出两个数
    if(i%2==0) cout<<endl;
    //每输出完4个数后换行,使每行输出4个数
    f1=f1+f2;
    //左边的f1代表第3个数,是第1、2个数之和
    f2=f2+f1;
    //左边的f2代表第4个数,是第2、3个数之和
  }
  return 0;
}

```

例 3.14 找出 100~200 间的全部素数。

编写程序如下：

```
#include <iostream>
#include <cmath>
#include <iomanip>
using namespace std;
int main()
{int m,k,i,n=0;
  bool prime;//定义布尔变量 prime
  for(m=101;m<=200;m=m+2) //判别 m 是否为素数,m 由 101 变化到 200,增量为 2
  {prime=true;//循环开始时设 prime 为真,即先认为 m 为素数
   k=int(sqrt(m)); //用 k 代表根号 m 的整数部分
   for(i=2;i<=k;i++) //此循环作用是将 m 被 2~根号 m 除,检查是否能整除
   if(m%i==0) //如果能整除,表示 m 不是素数
```

```

{ prime=false; //使 prime 变为假
break; //终止执行本循环
}
if (prime)//如果 m 为素数
{cout<<setw(5)<<m; //输出素数 m, 字段宽度为 5
n=n+1; //n 用来累计输出素数的个数
}
if(n%10==0) cout<<endl; //输出 10 个数后换行
}
cout<<endl; //最后执行一次换行
return 0;
}

```

例 3.15 译密码。为使电文保密,往往按一定规律将电文转换成密码,收报人再按约定的规律将其译回原文。例如,可以按以下规律将电文变成密码:将字母 A 变成字母 E,a 变成 e,即变成其后的第 4 个字母,W 变成 A,X 变成 B,Y 变成 C,Z 变成 D。见图 3.20。字母按上述规律转换,非字母字符不变。如"Wonderful!"转换为"Asrhivjyp!"。输入一行字符,要求输出其相应的密码。

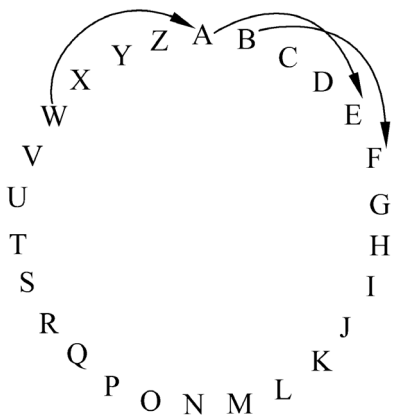


图 3.20

程序如下:

```
#include <iostream>
using namespace std;
int main( )
{char c;
  while ((c=getchar( ))!='\n')
{if((c>='a' && c<='z') || (c>='A' && c<='Z'))
{c=c+4;
  if(c>'Z' && c<='Z'+4 || c>'z') c=c-26;
}
  cout<<c;
}
  cout<<endl;
  return 0;
}
```



运行结果如下:

I am going to Beijing!✓

M eq ksmrk xs Fimnmrk!

while 语句中括号内的表达式有 3 个作用: ①从键盘读入一个字符,这是用 `getchar` 函数实现的; ②将读入的字符赋给字符变量 `c`; ③判别这个字符是否为 `'\n'`(即换行符)。如果是换行符就执行 while 语句中的复合语句(即花括号内的语句),对输入的非换行符的字符进行转换处理。

按前面分析的思路对输入的字符进行处理。有一点请读者注意:内嵌的 if 语句不能写成

`if (c>'Z' || c>'z') c=c-26;`

因为所有小写字母都满足“`c>'Z'`”条件,从而也执行“`c=c-26;`”语句,这就会出错。因此必须限制其范围为“`c>'Z' && c<='Z'+4`”,即原字母为'W'到'Z',在此范围以外的不是原大写字母 W~Z,不应按此规律转换。请考虑:为什么对小写字母不按此处理,即写成 `c>'z' && c<='z'+4` 而只须写成“`c>'z'`”即可。

## 第4章 函数与预处理

### 4.1 概述

### 4.2 定义函数的一般形式

### 4.3 函数参数和函数的值

### 4.4 函数的调用

### \*4.5 内置函数

### \*4.6 函数的重载

### \*4.7 函数模板

### \*4.8 有默认参数的函数

### 4.9 函数的嵌套调用

### 4.10 函数的递归调用

### 4.11 局部变量和全局变量

### 4.12 变量的存储类别

### 4.13 变量属性小结

### 4.14 关于变量的声明和定义

### 4.15 内部函数和外部函数

### 4.16 预处理命令

### 4.1 概述

一个较大的程序不可能完全由一个人从头至尾地完成,更不可能把所有的内容都放在一个主函数中。为了便于规划、组织、编程和调试,一般的做法是把一个大的程序划分为若干个程序模块(即程序文件),每一个模块实现一部分功能。不同的程序模块可以由不同的人来完成。在程序进行编译时,以程序模块为编译单位,即分别对每一个编译单位进行编译。如果发现错误,可以在本程序模块范围内查错并改正。在分别通过编译后,才进行连接,把各模块的目标文件以及系统文件连接在一起形成可执行文件。

在一个程序文件中可以包含若干个函数。无论把一个程序划分为多少个程序模块,只能有一个 `main` 函数。程序总是从 `main` 函数开始执行的。在程序运行过程中,由主函数调用其他函数,其他函数也可以互相调用。在 C 语言中没有类和对象,在程序模块中直接定义函数。可以认为,一个 C 程序是由若干个函数组成的,C 语言被认为是面向函数的语言。C++面向过程的程序设计沿用了 C 语言使用函数的方法。在 C++面向对象的程序设计中,主函数以外的函数大多是被封装在类中的。主函数或其他函数可以通过类对象调用类中的函数。无论是 C 还是 C++,程序中的各项操作基本上都是由函数来实现的,程序编写者要根据需要编写一个个函数,每个函数用来实现某一功能。因此,读者必须掌握函数的概念以及学会设计和使用函数。

“函数”这个名词是从英文 `function` 翻译过来的,其实 `function` 的原意是“功能”。顾名思义,一个函数就是一个功能。

在实际应用的程序中,主函数写得很简单,它的作用就是调用各个函数,程序各部分的功能全部都是由各函数实现的。主函数相当于总调度,调动各函数依次实现各项功能。

开发商和软件开发人员将一些常用的功能模块编写成函数,放在函数库中供公共选用。程序开发人员要善于利用库函数,以减少重复编写程序段的工作量。

图 4.1 是一个程序中函数调用的示意图。

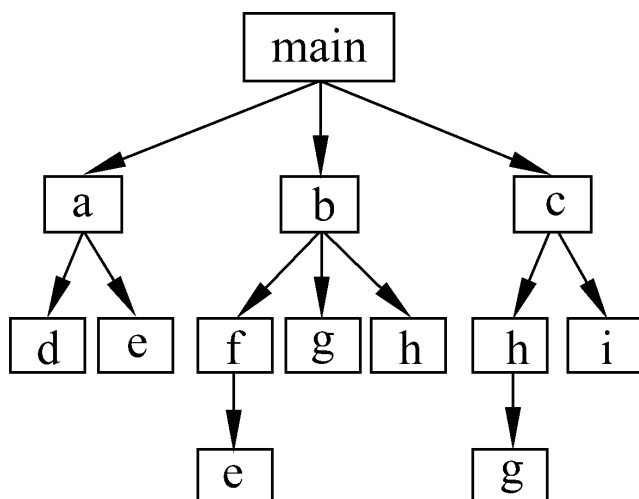


图 4.1

例 4.1 在主函数中调用其他函数。

```
#include <iostream>
using namespace std;
void printstar(void) //定义printstar 函数
{
    cout<<" ***** " <<endl; //输出 30 个“*”
}

void print_message(void) //定义print_message 函数
{
    cout<<" Welcome to C++!" <<endl; //输出一行文字
}

int main(void)
{
    printstar(); //调用printstar 函数
    print_message(); //调用print_message 函数
    printstar(); //调用printstar 函数
    return 0;
}
```

运行情况如下：

```
*****
Welcome to C++!
*****
```

从用户使用的角度看,函数有两种：

- (1) 系统函数,即库函数。这是由编译系统提供的,用户不必自己定义这些函数,可以直接使用它们。
- (2) 用户自己定义的函数。用以解决用户的专门需要。

从函数的形式看,函数分两类：

- (1) 无参函数。调用函数时不必给出参数。

(2) 有参函数。在调用函数时,要给出参数。在主调函数和被调用函数之间有数据传递。

## 4.2 定义函数的一般形式

### 4.2.1 定义无参函数的一般形式

定义无参函数的一般形式为

类型标识符 函数名([void])

{声明部分

语句

}

例 4.1 中的 `printstar` 和 `print_message` 函数都是无参函数。用类型标识符指定函数的类型,即函数带回来的值的类型。

### 4.2.2 定义有参函数的一般形式

定义有参函数的一般形式为

类型标识符 函数名(形式参数表列)

{声明部分

语句

}

例如:

```
int max(int x,int y) //函数首部,函数值为整型,有两个整型形参
{int z; //函数体中的声明部分
  z=x>y?x:y; //将x和y中的大者的值赋给整型变量z
  return (z); //将z的值作为函数值返回调用点
}
```

C++要求在定义函数时必须指定函数的类型。

## 4.3 函数参数和函数的值

### 4.3.1 形式参数和实际参数

在调用函数时,大多数情况下,函数是带参数的。主调函数和被调用函数之间有数据传递关系。前面已提到:在定义函数时函数名后面括号中的变量名称为形式参数(formal parameter,简称形参),在主调函数中调用一个函数时,函数名后面括号中的参数(可以是一个表达式)称为实际参数(actual parameter,简称实参)。

例 4.2 调用函数时的数据传递。

```
#include <iostream>
using namespace std;
int max(int x,int y) //定义有参函数 max
{int z;
  z=x>y?x:y;
  return(z);
}

int main( )
{int a,b,c;
  cout<<" please enter two integer numbers:" ;
  cin>>a>>b;
  c=max(a,b); //调用 max 函数,给定实参为 a,b。函数值赋给 c
  cout<<" max=" <<c<<endl;
```

```

return 0;
}

```

运行情况如下:

please enter two integer numbers: 2 3 ✓

max=3

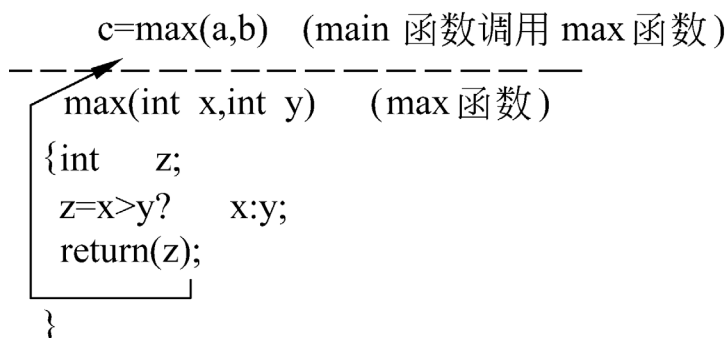
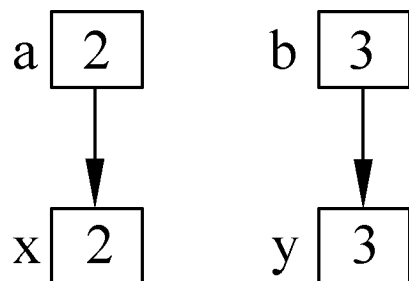


图 4.2

有关形参与实参的说明:

- (1) 在定义函数时指定的形参,在未出现函数调用时,它们并不占内存中的存储单元,因此称它们是形式参数或虚拟参数,表示它们并不是实际存在的数据,只有在发生函数调用时,函数 `max` 中的形参才被分配内存单元,以便接收从实参传来的数据。在调用结束后,形参所占的内存单元也被释放。
- (2) 实参可以是常量、变量或表达式,如 `max(3, a+b)`;但要求 `a` 和 `b` 有确定的值。以便在调用函数时将实参的值赋给形参。
- (3) 在定义函数时,必须在函数首部指定形参的类型(见例 4.2 程序第 3 行)。
- (4) 实参与形参的类型应相同或赋值兼容。例 4.2 中实参和形参都是整型,这是合法的、正确的。如果实参为整型而形参为实型,或者相反,则按不同类型数值的赋值规则进行转换。例如实参 `a` 的值为 3.5,而形参 `x` 为整型,则将 3.5 转换成整数 3,然后送到形参 `b`。字符型与整型可以互相通用。
- (5) 实参变量对形参变量的数据传递是“值传递”,即单向传递,只由实参传给形参,而不能由形参传回来给实参。在调用函数时,编译系统临时给形参分配存储单元。请注意: 实参单元与形参单元是不同的单元。图 4.3 表示将实参 `a` 和 `b` 的值 2 和 3 传递给对应的形参 `x` 和 `y`。



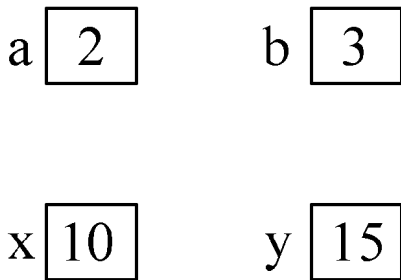


图 4.3 图 4.4

调用结束后,形参单元被释放,实参单元仍保留并维持原值。因此,在执行一个被调用函数时,形参的值如果发生改变,并不会改变主调函数中实参的值。例如,若在执行 `max` 函数过程中形参 `x` 和 `y` 的值变为 10 和 15,调用结束后,实参 `a` 和 `b` 仍为 2 和 3,见图 4.4。

#### 4.3.2 函数的返回值

(1) 函数的返回值是通过函数中的 `return` 语句获得的。`return` 语句将被调用函数中的一个确定值带回主调函数中去。

`return` 语句后面的括号可以要,也可以不要。`return` 后面的值可以是一个表达式。

(2) 函数值的类型。既然函数有返回值,这个值当然应属于某一个确定的类型,应当在定义函数时指定函数值的类型。

(3) 如果函数值的类型和 `return` 语句中表达式的值不一致,则以函数类型为准,即函数类型决定返回值的类型。对数值型数据,可以自动进行类型转换。

### 4.4 函数的调用

#### 4.4.1 函数调用的一般形式

函数名([实参表列])

如果是调用无参函数,则“实参表列”可以没有,但括号不能省略。如果实参表列包含多个实参,则各参数间用逗号隔开。实参与形参的个数应相等,类型应匹配(相同或赋值兼容)。实参与形参按顺序对应,一对一地传递数据。但应说明,如果实参表列包括多个实参,对实参求值的顺序并不是确定的。

#### 4.4.2 函数调用的方式

按函数在语句中的作用来分,可以有以下 3 种函数调用方式:

##### 1. 函数语句

把函数调用单独作为一个语句,并不要求函数带回一个值,只是要求函数完成一定的操作。如例 4.1 中的 `printstar()`;

##### 2. 函数表达式

函数出现在一个表达式中,这时要求函数带回一个确定的值以参加表达式的运算。如 `c=2*max(a,b)`;

##### 3. 函数参数

函数调用作为一个函数的实参。如

`m=max(a,max(b,c));` //*max(b,c)是函数调用,其值作为外层 max 函数调用的一个实参*

#### 4.4.3 对被调用函数的声明和函数原型

在一个函数中调用另一个函数(即被调用函数)需要具备哪些条件呢?

(1) 首先被调用的函数必须是已经存在的函数。

(2) 如果使用库函数,一般还应该在本文件开头用 `#include` 命令将有关头文件“包含”到本文件中来。

(3) 如果使用用户自己定义的函数,而该函数与调用它的函数(即主调函数)在同一个程序单位中,且位置在

主调函数之后,则必须在调用此函数之前对被调用的函数作声明。

所谓函数声明(declare),就是在函数尚在未定义的情况下,事先将该函数的有关信息通知编译系统,以便使编译能正常进行。

例 4.3 对被调用的函数作声明。

```
#include <iostream>
using namespace std;
int main( )
{float add(float x,float y); //对add 函数作声明
  float a,b,c;
  cout<<" please enter a,b: " ;
  cin>>a>>b;
  c=add(a,b);
  cout<<" sum=" <<c<<endl;
  return 0;
}

float add(float x,float y)//定义add 函数
{float z;
  z=x+y;
  return (z);
}
```

运行情况如下:

```
please enter a,b:123.68 456.45✓
sum=580.13
```

注意:对函数的定义和声明不是同一件事情。定义是指对函数功能的确立,包括指定函数名、函数类型、形参及其类型、函数体等,它是一个完整的、独立的函数单位。而声明的作用则是把函数的名字、函数类型以及形参的个数、类型和顺序(注意,不包括函数体)通知编译系统,以便在对包含函数调用的语句进行编译时,据此对其进行对照检查(例如函数名是否正确,实参与形参的类型和个数是否一致)。

其实,在函数声明中也可以不写形参名,而只写形参的类型,如

```
float add(float,float);
```

这种函数声明称为函数原型(function prototype)。使用函数原型是 C 和 C++ 的一个重要特点。它的作用主要是:根据函数原型在程序编译阶段对调用函数的合法性进行全面检查。如果发现与函数原型不匹配的函数调用就报告编译出错。它属于语法错误。用户根据屏幕显示的出错信息很容易发现和纠正错误。

函数原型的一般形式为

- (1) 函数类型 函数名(参数类型 1,参数类型 2...);
- (2) 函数类型 函数名(参数类型 1 参数名 1,参数类型 2 参数名 2...);

第(1)种形式是基本的形式。为了便于阅读程序,也允许在函数原型中加上参数名,就成了第(2)种形式。但编译系统并不检查参数名。因此参数名是什么都无所谓。上面程序中的声明也可以写成

```
float add(float a,float b); //参数名不用x,y,而用a,b
```

效果完全相同。

应当保证函数原型与函数首部写法上的一致,即函数类型、函数名、参数个数、参数类型和参数顺序必须相同。在函数调用时函数名、实参类型和实参个数应与函数原型一致。

说明:

(1) 前面已说明,如果被调用函数的定义出现在主调函数之前,可以不必加以声明。因为编译系统已经事先知道了已定义的函数类型,会根据函数首部提供的信息对函数的调用作正确性检查。

有经验的程序编制人员一般都把 `main` 函数写在最前面,这样对整个程序的结构和作用一目了然,统览全局,然后再具体了解各函数的细节。此外,用函数原型来声明函数,还能减少编写程序时可能出现的错误。由于函数声明的位置与函数调用语句的位置比较近,因此在写程序时便于就近参照函数原型来书写函数调用,不易出错。所以应养成对所有用到的函数作声明的习惯。这是保证程序正确性和可读性的重要环节。

(2) 函数声明的位置可以在调用函数所在的函数中,也可以在函数之外。如果函数声明放在函数的外部,在所有函数定义之前,则在各个主调函数中不必对它所调用的函数再作声明。例如:

```
char letter(char,char); //本行和以下两行函数声明在所有函数之前且在函数外部
float f(float,float); //因而作用域是整个文件
int i(float, float);
int main( )
{...} //在 main 函数中不必对它所调用的函数作声明
char letter(char c1,char c2) //定义 letter 函数
{...}
float f(float x,float y) //定义 f 函数
{...}
int i(float j,float k) //定义 i 函数
{...}
```

如果一个函数被多个函数所调用,用这种方法比较好,不必在每个主调函数中重复声明。

#### \*4.5 内置函数

调用函数时需要一定的时间和空间的开销。图 4.5 表示函数调用的过程:

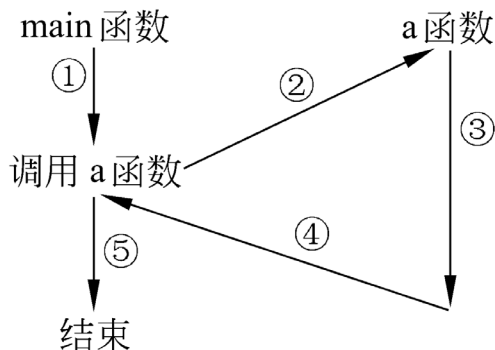


图 4.5

C++ 提供一种提高效率的方法,即在编译时将所调用函数的代码直接嵌入到主调函数中,而不是将流程转出去。这种嵌入到主调函数中的函数称为内置函数(`inline function`),又称内嵌函数。在有些书中把它译成内联函数。

指定内置函数的方法很简单,只需在函数首行的左端加一个关键字 `inline` 即可。

例 4.4 函数指定为内置函数。

```
#include <iostream>
using namespace std;
inline int max(int,int, int); //声明函数,注意左端有 inline
```



```

int main( )
{int i=10,j=20,k=30,m;
  m=max(i,j,k);
  cout<<" max=" <<m<<endl;
  return 0;
}

inline int max(int a,int b,int c) //定义 max 为内置函数
{if(b>a) a=b;//求 a,b,c 中的最大者
  if(c>a) a=c;
  return a;
}

```

由于在定义函数时指定它为内置函数,因此编译系统在遇到函数调用“max(i,j,k)”时,就用 max 函数体的代码代替“max(i,j,k)”,同时将实参代替形参。这样,程序第 6 行 “m=max(i,j,k);”就被置换成

```

if (j>i) i=j;
if (k>i) i=k;
m=i;

```

注意: 可以在声明函数和定义函数时同时写 inline,也可以只在其中一处声明 inline,效果相同,都能按内置函数处理。

使用内置函数可以节省运行时间,但却增加了目标程序的长度。因此一般只将规模很小(一般为 5 个语句以下)而使用频繁的函数(如定时采集数据的函数)声明为内置函数。

内置函数中不能包括复杂的控制语句,如循环语句和 switch 语句。

应当说明: 对函数作 inline 声明,只是程序设计者对编译系统提出的一个建议,也就是说它是建议性的,而不是指令性的。并非一经指定为 inline,编译系统就必须这样做。编译系统会根据具体情况决定是否这样做。

归纳起来,只有那些规模较小而又被频繁调用的简单函数,才适合于声明为 inline 函数。

#### \*4.6 函数的重载

在编程时,有时我们要实现的是同一类的功能,只是有些细节不同。例如希望从 3 个数中找出其中的最大者,而每次求最大数时数据的类型不同,可能是 3 个整数、3 个双精度数或 3 个长整数。程序设计者往往会分别设计出 3 个不同名的函数,其函数原型为:

```

int max1(int a,int b, int c); //求 3 个整数中的最大者
double max2(double a,double b,double c); //求 3 个双精度数中最大者
long max3(long a,long b,long c); //求 3 个长整数中的最大者

```

C++ 允许用同一函数名定义多个函数,这些函数的参数个数和参数类型不同。这就是函数的重载(function overloading)。即对一个函数名重新赋予它新的含义,使一个函数名可以多用。

对上面求最大数的问题可以编写如下的 C++ 程序。

例 4.5 求 3 个数中最大的数(分别考虑整数、双精度数、长整数的情况)。

```

#include <iostream>
using namespace std;
int main( )
{int max(int a,int b,int c); //函数声明
double max(double a,double b,double c); //函数声明
long max(long a,long b,long c); //函数声明
  int i1,i2,i3,i;

```

```

cin>>i1>>i2>>i3; //输入 3 个整数
i=max(i1,i2,i3); //求 3 个整数中的最大者
cout<<" i_max=" <<i<<endl;
double d1,d2,d3,d;
cin>>d1>>d2>>d3; //输入 3 个双精度数
d=max(d1,d2,d3); //求 3 个双精度数中的最大者
cout<<" d_max=" <<d<<endl;
long g1,g2,g3,g;
cin>>g1>>g2>>g3; //输入 3 个长整数

g=max(g1,g2,g3); //求 3 个长整数中的最大者
cout<<" g_max=" <<g<<endl;
}

int max(int a,int b,int c) //定义求 3 个整数中的最大者的函数
{if(b>a) a=b;
 if(c>a) a=c;
 return a;
}

double max(double a,double b,double c) //定义求 3 个双精度数中的最大者的函数
{if(b>a) a=b;
 if(c>a) a=c;
 return a;
}

long max(long a,long b,long c) //定义求 3 个长整数中的最大者的函数
{if(b>a) a=b;
 if(c>a) a=c;
 return a;
}

```

运行情况如下:

```

185-76567✓ (输入 3 个整数)
56.87 90.23 -3214.78✓ (输入 3 个实数)
67854 -912456 673456✓ (输入 3 个长整数)
i_max=567 (输出 3 个整数的最大值)
d_max=90.23 (输出 3 个双精度数的最大值)
g_max=673456 (输出 3 个长整数的最大值)

```

上例 3 个 max 函数的函数体是相同的,其实重载函数并不要求函数体相同。重载函数除了允许参数类型不同以外,还允许参数的个数不同。

例 4.6 编写一个程序,用来求两个整数或 3 个整数中的最大数。如果输入两个整数,程序就输出这两个整数中的最大数,如果输入 3 个整数,程序就输出这 3 个整数中的最大数。

```

#include <iostream>
using namespace std;
int main( )

```

```

{int max(int a,int b,int c); //函数声明
  int max(int a,int b); //函数声明
  int a=8,b=-12,c=27;
  cout<<" max(a,b,c)=" <<max(a,b,c)<<endl;//输出 3 个整数中的最大者
  cout<<" max(a,b)=" <<max(a,b)<<endl; //输出两个整数中的最大者
}

```

```

int max(int a,int b,int c)//此 max 函数的作用是求 3 个整数中的最大者
{if(b>a) a=b;

if(c>a) a=c;
  return a;
}

```

```

int max(int a,int b)//此 max 函数的作用是求两个整数中的最大者
{if(a>b) return a;
else return b;
}

```

运行情况如下:

max(a,b,c)=27

max(a,b)=8

两次调用 max 函数的参数个数不同,系统就根据参数的个数找到与之匹配的函数并调用它。

参数的个数和类型可以都不同。但不能只有函数的类型不同而参数的个数和类型相同。例如:

```

int f(int); //函数返回值为整型
long f(int); //函数返回值为长整型
void f(int); //函数无返回值

```

在函数调用时都是同一形式,如“f(10)”。编译系统无法判别应该调用哪一个函数。重载函数的参数个数、参数类型或参数顺序 3 者中必须至少有一种不同,函数返回值类型可以相同也可以不同。

在使用重载函数时,同名函数的功能应当相同或相近,不要用同一函数名去实现完全不相干的功能,虽然程序也能运行,但可读性不好,使人莫名其妙。

#### \*4.7 函数模板

C++提供了函数模板(function template)。所谓函数模板,实际上是建立一个通用函数,其函数类型和形参类型不具体指定,用一个虚拟的类型来代表。这个通用函数就称为函数模板。凡是函数体相同的函数都可以用这个模板来代替,不必定义多个函数,只需在模板中定义一次即可。在调用函数时系统会根据实参的类型来取代模板中的虚拟类型,从而实现了不同函数的功能。看下面的例子就清楚了。

例 4.7 将例 4.6 程序改为通过函数模板来实现。

```

#include <iostream>
using namespace std;
template<typename T> //模板声明,其中 T 为类型参数
T max(T a,T b,T c) //定义一个通用函数,用 T 作虚拟的类型名
{if(b>a) a=b;
  if(c>a) a=c;
  return a;
}

```

```

int main( )
{int i1=185,i2=-76,i3=567,i;
 double d1=56.87,d2=90.23,d3=-3214.78,d;
 long g1=67854,g2=-912456,g3=673456,g;
 i=max(i1,i2,i3); //调用模板函数,此时 T 被 int 取代
 d=max(d1,d2,d3); //调用模板函数,此时 T 被 double 取代
 g=max(g1,g2,g3); //调用模板函数,此时 T 被 long 取代
 cout<<" i_max=" <<i<<endl;
 cout<<" f_max=" <<f<<endl;
 cout<<" g_max=" <<g<<endl;
 return 0;
}

```

运行结果与例 4.5 相同。为了节省篇幅,数据不用 cin 语句输入,而在变量定义时初始化。

程序第 3~8 行是定义模板。定义函数模板的一般形式为

template <typename T> 或 template <class T>

通用函数定义 通用函数定义

在建立函数模板时,只要将例 4.5 程序中定义的第一个函数首部的 int 改为 T 即可。即用虚拟的类型名 T 代替具体的数据类型。在对程序进行编译时,遇到第 13 行调用函数 max(i1,i2,i3),编译系统会将函数名 max 与模板 max 相匹配,将实参的类型取代了函数模板中的虚拟类型 T。此时相当于已定义了一个函数:

```

int max(int a,int b,int c)
{if(b>a) a=b;
 if(c>a) a=c;
 return a;
}

```

然后调用它。后面两行(14,15 行)的情况类似。

类型参数可以不只一个,可以根据需要确定个数。如

template <class T1,typename T2>

可以看到,用函数模板比函数重载更方便,程序更简洁。但应注意它只适用于函数的参数个数相同而类型不同,且函数体相同的情况,如果参数的个数不同,则不能用函数模板。

#### \*4.8 有默认参数的函数

一般情况下,在函数调用时形参从实参那里取得值,因此实参的个数应与形参相同。有时多次调用同一函数时用同样的实参,C++提供简单的处理办法,给形参一个默认值,这样形参就不必一定要从实参取值了。如有一函数声明

```
float area(float r=6.5);
```

指定 r 的默认值为 6.5,如果在调用此函数时,确认 r 的值为 6.5,则可以不必给出实参的值,如

```
area(); //相当于 area(6.5);
```

如果不想使形参取此默认值,则通过实参另行给出。如

```
area(7.5); //形参得到的值为 7.5,而不是 6.5
```

这种方法比较灵活,可以简化编程,提高运行效率。

如果有多个形参,可以使每个形参有一个默认值,也可以只对一部分形参指定默认值,另一部分形参不指定默认值。如有一个求圆柱体体积的函数,形参 h 代表圆柱体的高,r 为圆柱体半径。函数原型如下:

```
float volume(float h,float r=12.5); //只对形参 r 指定默认值 12.5
```

函数调用可以采用以下形式:

```
volume(45.6); //相当于 volume(45.6,12.5)
```

```
volume(34.2,10.4) //h 的值为 34.2,r 的值为 10.4
```

实参与形参的结合是从左至右顺序进行的。因此指定默认值的参数必须放在形参表列中的最右端,否则出错。例如:

```
void f1(float a,int b=0,int c,char d=' a' ); //不正确
```

```
void f2(float a,int c,int b=0, char d=' a' ); //正确
```

如果调用上面的 f2 函数,可以采取下面的形式:

```
f2(3.5, 5, 3, ' x' )//形参的值全部从实参得到
```

```
f2(3.5, 5, 3) //最后一个形参的值取默认值 ' a' 
```

```
f2(3.5, 5) //最后两个形参的值取默认值,b=0,d=' a' 
```

可以看到,在调用有默认参数的函数时,实参的个数可以与形参的个数不同,实参未给定的,从形参的默认值得到值。利用这一特性,可以使函数的使用更加灵活。例如例 4.7 求 2 个数或 3 个数中的最大数。也可以不用重载函数,而改用带有默认参数的函数。

例 4.8 求 2 个或 3 个正整数中的最大数,用带有默认参数的函数实现。

```
#include <iostream>
```

```
using namespace std;
```

```
int main( )
```

```
{int max(int a, int b, int c=0);//函数声明,形参 c 有默认值
```

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

```
cin>>a>>b>>c;
```

```
cout<<" max(a,b,c)=" <<max(a,b,c)<<endl;//输出 3 个数中的最大者
```

```
cout<<" max(a,b)=" <<max(a,b)<<endl; //输出 2 个数中的最大者
```

```
return 0;
```

```
}
```

```
int max(int a,int b,int c) //函数定义
```

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

```
if(c>a) a=c;
```

```
return a;
```

```
}
```

运行情况如下:

```
14 -56 135✓
```

```
max(a,b,c)=135
```

```
max(a,b)=14
```

在使用带有默认参数的函数时有两点要注意:

(1) 如果函数的定义在函数调用之前,则应在函数定义中给出默认值。如果函数的定义在函数调用之后,则在函数调用之前需要有函数声明,此时必须在函数声明中给出默认值,在函数定义时可以不给出默认值(如例 4.8)。

(2) 一个函数不能既作为重载函数,又作为有默认参数的函数。因为当调用函数时如果少写一个参数,系统无法判定是利用重载函数还是利用默认参数的函数,出现二义性,系统无法执行。

## 4.9 函数的嵌套调用

C++不允许对函数作嵌套定义,也就是说在一个函数中不能完整地包含另一个函数。在一个程序中每一个函

数的定义都是互相平行和独立的。  
虽然 C++不能嵌套定义函数,但可以嵌套调用函数,也就是说,在调用一个函数的过程中,又调用另一个函数。  
见图 4.6。

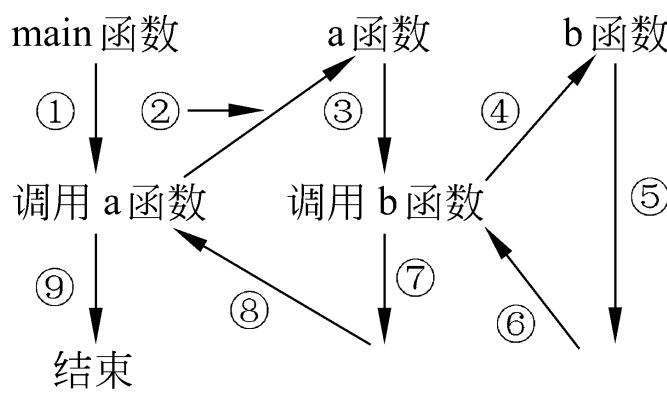


图 4.6

在程序中实现函数嵌套调用时,需要注意的是: 在调用函数之前,需要对每一个被调用的函数作声明(除非定义在前,调用在后)。

例 4.9 用弦截法求方程  $f(x)=x^3-5x^2+16x-80=0$  的根。

这是一个数值求解问题,需要先分析用弦截法求根的算法。根据数学知识,可以列出以下的解题步骤:

- (1) 取两个不同点  $x_1, x_2$ , 如果  $f(x_1)$  和  $f(x_2)$  符号相反, 则  $(x_1, x_2)$  区间内必有一个根。如果  $f(x_1)$  与  $f(x_2)$  同符号, 则应改变  $x_1, x_2$ , 直到  $f(x_1), f(x_2)$  异号为止。注意  $x_1, x_2$  的值不应差太大, 以保证  $(x_1, x_2)$  区间内只有一个根。
- (2) 连接  $(x_1, f(x_1))$  和  $(x_2, f(x_2))$  两点, 此线(即弦)交  $x$  轴于  $x$ , 见图 4.7。

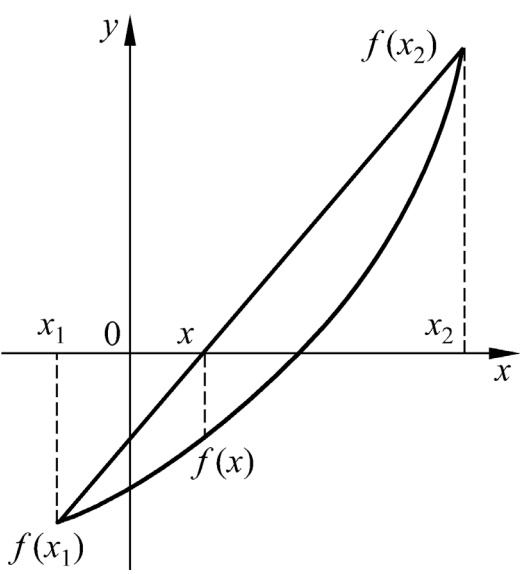


图 4.7

$x$  点坐标可用下式求出:

$$x = x_1 \cdot f(x_2) - x_2 \cdot f(x_1)$$

$$f(x_2) - f(x_1)$$

再从  $x$  求出  $f(x)$ 。

(3) 若  $f(x)$  与  $f(x_1)$  同符号,则根必在  $(x, x_2)$  区间内,此时将  $x$  作为新的  $x_1$ 。如果  $f(x)$  与  $f(x_2)$  同符号,则表示根在  $(x_1, x)$  区间内,将  $x$  作为新的  $x_2$ 。

(4) 重复步骤 (2) 和 (3), 直到  $|f(x)| < \xi$  为止,  $\xi$  为一个很小的正数, 例如  $10^{-6}$ 。此时认为  $f(x) \approx 0$ 。

这就是弦截法的算法,在程序中分别用以下几个函数来实现以上有关部分功能:

(1) 用函数  $f(x)$  代表  $x$  的函数:  $x^3 - 5x^2 + 16x - 80$ 。

(2) 用函数  $xpoint(x_1, x_2)$  来求  $(x_1, f(x_1))$  和  $(x_2, f(x_2))$  的连线与  $x$  轴的交点  $x$  的坐标。

(3) 用函数  $root(x_1, x_2)$  来求  $(x_1, x_2)$  区间的那个实根。显然,执行  $root$  函数的过程中要用到  $xpoint$  函数,而执行  $xpoint$  函数的过程中要用到  $f$  函数。

根据以上算法,可以编写出下面的程序:

```
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;
double f(double); //函数声明
double xpoint(double, double); //函数声明
double root(double, double); //函数声明

int main( )
{ double x1,x2,f1,f2,x;

do
    {cout<<" input x1,x2: " ;
cin>>x1>>x2;
f1=f(x1);
f2=f(x2);
    } while(f1*f2>=0);
x=root(x1,x2);
cout<<setiosflags(ios::fixed)<<setprecision(7);
//指定输出 7 位小数
cout<<" A root of equation is " <<x<<endl;
return 0;
}

double f(double x) //定义 f 函数,以实现 f(x)
{double y;
```

```

y=x*x*x-5*x*x+16*x-80;
return y;
}

```

```

double xpoint(double x1, double x2) //定义 xpoint 函数,求出弦与 x 轴交点
{double y;
 y=(x1*f(x2)-x2*f(x1))/(f(x2)-f(x1)); //在 xpoint 函数中调用 f 函数
 return y;
}

```

```

double root(double x1, double x2) //定义 root 函数,求近似根
{double x,y,y1;
 y1=f(x1);
 do
 {x=xpoint(x1,x2); //在 root 函数中调用 xpoint 函数
 y=f(x); //在 root 函数中调用 f 函数
 if (y*y1>0)
 {y1=y;
 x1=x;
 }
 else
 x2=x;
 }while(fabs(y)>=0.00001);
 return x;
}

```

运行情况如下:

input x1,x2:2.5 6.7↵

A root of equation is 5.0000000

对程序的说明:

- (1) 在定义函数时,函数名为 f,xpoint 和 root 的 3 个函数是互相独立的,并不互相从属。这 3 个函数均定为双精度型。
- (2) 3 个函数的定义均出现在 main 函数之后,因此在 main 函数的前面对这 3 个函数作声明。习惯上把本程序中用到的所有函数集中放在最前面声明。
- (3) 程序从 main 函数开始执行。函数的嵌套调用见图 4.8。

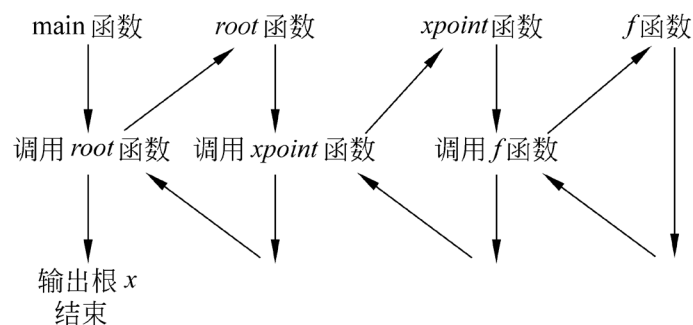


图 4.8

- (4) 在 root 函数中要用到求绝对值的函数 fabs,它是对双精度数求绝对值的系统函数。它属于数学函数库,



故在文件开头用#include <cmath>把有关的头文件包含进来。

#### 4.10 函数的递归调用

在调用一个函数的过程中又出现直接或间接地调用该函数本身,称为函数的递归(recursive)调用。C++允许函数的递归调用。例如:

```
int f(int x)
{int y,z;
z=f(y); //在调用函数f的过程中,又要调用f函数
return (2*z);
}
```

以上是直接调用本函数,见图 4.9。

图 4.10 表示的是间接调用本函数。在调用 f1 函数过程中要调用 f2 函数,而在调用 f2 函数过程中又要调用 f1 函数。

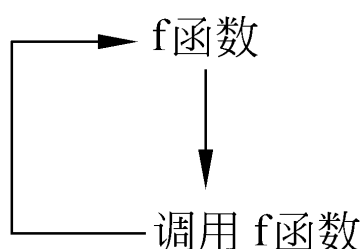
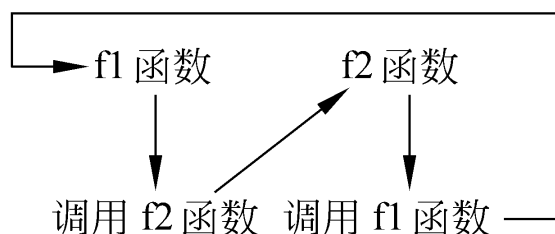


图 4.9 图 4.10



从图上可以看到,这两种递归调用都是无终止的自身调用。显然,程序中不应出现这种无终止的递归调用,而只应出现有限次数的、有终止的递归调用,这可以用 if 语句来控制,只有在某一条件成立时才继续执行递归调用,否则就不再继续。

包含递归调用的函数称为递归函数。

例 4.10 有 5 个人坐在一起,问第 5 个人多少岁?他说比第 4 个人大两岁。问第 4 个人岁数,他说比第 3 个人大两岁。问第 3 个人,又说比第 2 个人大两岁。问第 2 个人,说比第 1 个人大两岁。最后问第 1 个人,他说是 10 岁。请问第 5 个人多大?

每一个人的年龄都比其前 1 个人的年龄大两岁。即

age(5)=age(4)+2

age(4)=age(3)+2

age(3)=age(2)+2

age(2)=age(1)+2

age(1)=10

可以用式子表述如下:

age(n)=10 (n=1)

age(n)=age(n-1)+2 (n>1)

可以看到,当  $n>1$  时,求第  $n$  个人的年龄的公式是相同的。因此可以用一个函数表示上述关系。图 4.11 表示求第 5 个人年龄的过程。

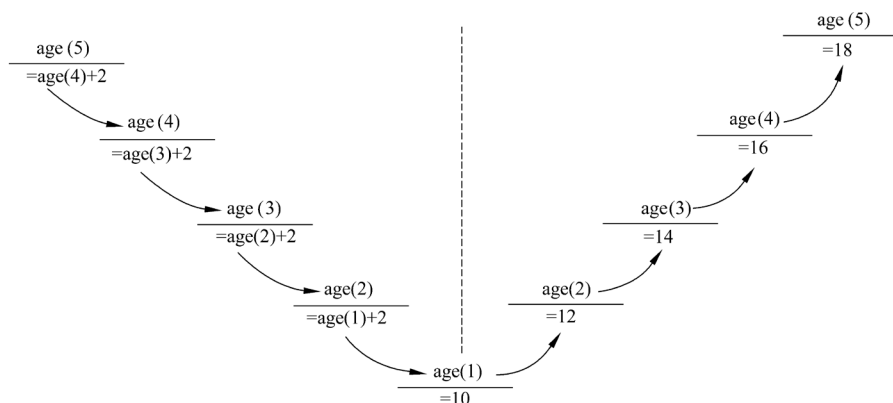


图 4.11

可以写出以下 C++ 程序,其中的 `age` 函数用来实现上述递归过程。

```
#include <iostream>
using namespace std;
int age(int); // 函数声明
int main() // 主函数
{ cout << age(5) << endl;
  return 0;
}

int age(int n) // 求年龄的递归函数
{ int c; // 用 c 作为存放年龄的变量
  if (n == 1) c = 10; // 当 n=1 时, 年龄为 10
  else c = age(n-1) + 2; // 当 n>1 时, 此人年龄是他前一个人的年龄加 2
  return c; // 将年龄值带回主函数
}
```

运行结果如下:

18

函数调用过程如图 4.12 所示。

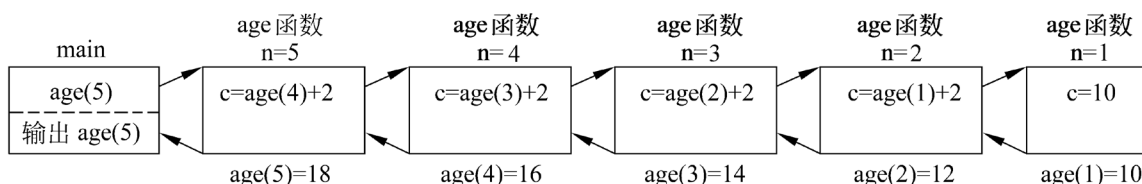


图 4.12

例 4.11 用递归方法求  $n!$ 。

求  $n!$  可以用递推方法,即从 1 开始,乘 2,再乘 3.....一直乘到  $n$ 。

求  $n!$  也可以用递归方法,即  $5! = 4! \times 5$ , 而  $4! = 3! \times 4$ , ...,  $1! = 1$ 。可用下面的递归公式表示:

$$n! = 1 \quad (n=0,1)$$

$$n \cdot (n-1)! \quad (n > 1)$$

有了例 4.10 的基础,很容易写出本题的程序:

```
#include <iostream>
using namespace std;
long fac(int); //函数声明
int main( )
{int n; //n 为需要求阶乘的整数
long y; //y 为存放 n! 的变量

cout<<" please input an integer : " ; //输入的提示
cin>>n; //输入 n
y=fac(n); //调用 fac 函数以求 n!
cout<<n<<" != " <<y<<endl; //输出 n! 的值
return 0;
}

long fac(int n) //递归函数
{long f;
if(n<0)
{cout<<" n<0,data error!" <<endl; //如果输入负数,报错并以-1 作为返回值
f=-1;}
else if (n==0||n==1) f=1; //0! 和 1! 的值为 1
else f=fac(n-1)*n; //n>1 时,进行递归调用
return f; //将 f 的值作为函数值返回
}
```

运行情况如下:

```
please input an integer:10↵
10!=3628800
```

许多问题既可以用递归方法来处理,也可以用非递归方法来处理。在实现递归时,在时间和空间上的开销比较大,但符合人们的思路,程序容易理解。

## 4.11 局部变量和全局变量

### 4.11.1 局部变量

在一个函数内部定义的变量是内部变量,它只在本函数范围内有效,也就是说只有在本函数内才能使用它们,在此函数以外是不能使用这些变量的。同样,在复合语句中定义的变量只在本复合语句范围内有效。这称为局部变量(local variable)。如

```
float f1(int a) //函数 f1
{
int b,c; b、c 有效 a 有效
:
}
char f2(int x, int y) //函数 f2
{int i,j; i、j 有效 x、y 有效
:
}
```

```
int main( )           //主函数
{int m,n;
  :
  {int p,q;  p、q 在复合语句中有效    m、n 有效
  :
  }
}
```

说明:

- (1) 主函数 main 中定义的变量(m,n)也只在主函数中有效,不会因为在主函数中定义而在整个文件或程序中有效。主函数也不能使用其他函数中定义的变量。
- (2) 不同函数中可以使用同名的变量,它们代表不同的对象,互不干扰。例如,在 f1 函数中定义了变量 b 和 c,倘若在 f2 函数中也定义变量 b 和 c,它们在内存中占不同的单元,不会混淆。
- (3) 可以在一个函数内的复合语句中定义变量,这些变量只在本复合语句中有效,这种复合语句也称为分程序或程序块。
- (4) 形式参数也是局部变量。例如 f1 函数中的形参 a 也只在 f1 函数中有效。其他函数不能调用。

(5) 在函数声明中出现的参数名,其作用范围只在本行的括号内。实际上,编译系统对函数声明中的变量名是忽略的,即使在调用函数时也没有为它们分配存储单元。例如

```
int max(int a,int b);//函数声明中出现 a、b
:
int max(int x,int y) //函数定义,形参是 x、y
{ cout<<x<<y<<endl; //合法,x、y 在函数体中有效
cout<<a<<b<<endl;  //非法,a、b 在函数体中无效
}
```

编译时认为 max 函数体中的 a 和 b 未经定义。

#### 4.11.2 全局变量

前面已介绍,程序的编译单位是源程序文件,一个源文件可以包含一个或若干个函数。在函数内定义的变量是局部变量,而在函数之外定义的变量是外部变量,称为全局变量(global variable,也称全程变量)。全局变量的有效范围为从定义变量的位置开始到本源文件结束。如

```
int p=1,q=5;//全局变量      全局变量 c1、c2 的作用范围
float f1(a)//定义函数 f1
int a;
{int b,c;
  :
}
char c1,c2; //全局变量      全局变量 p、q 的作用范围
char f2 (int x, int y) //定义函数 f2
{int i,j;
  :
}
main ( )//主函数
{int m,n;
  :
}
```

p、q、c1、c2 都是全局变量,但它们的作用范围不同,在 main 函数和 f2 函数中可以使用全局变量 p、q、c1、c2,但在函数 f1 中只能使用全局变量 p、q,而不能使用 c1 和 c2。

在一个函数中既可以使用本函数中的局部变量,又可以使用有效的全局变量。

说明:

(1) 设全局变量的作用是增加函数间数据联系的渠道。

(2) 建议不在必要时不要使用全局变量,因为:

① 全局变量在程序的全部执行过程中都占用存储单元,而不是仅在需要时才开辟单元。

② 它使函数的通用性降低了,因为在执行函数时要受到外部变量的影响。如果将一个函数移到另一个文件中,还要将有关的外部变量及其值一起移过去。但若该外部变量与其他文件的变量同名,就会出现问題,降低了程序的可靠性和通用性。在程序设计中,在划分模块时要求模块的内聚性强、与其他模块的耦合性弱。即模块的功能要单一(不要把许多互不相干的功能放到一个模块中),与其他模块的相互影响要尽量少,而用全局变量是不符合这个原则的。

一般要求把程序中的函数做成一个封闭体,除了可以通过“实参——形参”的渠道与外界发生联系外,没有其他渠道。这样的程序移植性好,可读性强。

③ 使用全局变量过多,会降低程序的清晰性。在各个函数执行时都可能改变全局变量的值,程序容易出错。因此,要限制使用全局变量。

(3) 如果在同一个源文件中,全局变量与局部变量同名,则在局部变量的作用范围内,全局变量被屏蔽,即它不起作用。

变量的有效范围称为变量的作用域(scope)。归纳起来,变量有 4 种不同的作用域、文件作用域(file scope)、函数作用域(function scope)、块作用域(block scope)和函数原型作用域(function prototype scope)。文件作用域是全局的,其他三者是局部的。

除了变量之外,任何以标识符代表的实体都有作用域,概念与变量的作用域相似。

## 4.12 变量的存储类别

### 4.12.1 动态存储方式与静态存储方式

上一节已介绍了变量的一种属性——作用域,作用域是从空间的角度来分析的,分为全局变量和局部变量。变量还有另一种属性——存储期(storage duration,也称生命期)。存储期是指变量在内存中的存在期间。这是从变量值存在的时间角度来分析的。存储期可以分为静态存储期(static storage duration)和动态存储期(dynamic storage duration)。这是由变量的静态存储方式和动态存储方式决定的。

所谓静态存储方式是指在程序运行期间,系统对变量分配固定的存储空间。而动态存储方式则是在程序运行期间,系统对变量动态地分配存储空间。

先看一下内存中的供用户使用的存储空间的情况。这个存储空间可以分为三部分,即:

(1) 程序区

(2) 静态存储区

(3) 动态存储区

## 用户区

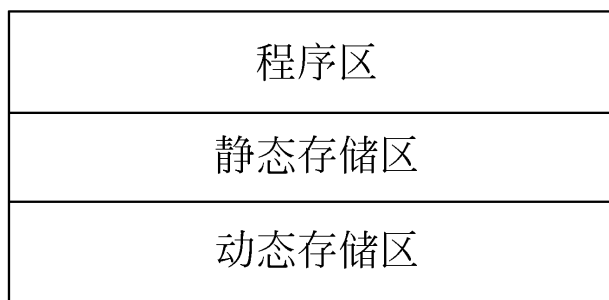


图 4.13

数据分别存放在静态存储区和动态存储区中。全局变量全部存放在静态存储区中,在程序开始执行时给全局变量分配存储单元,程序执行完毕就释放这些空间。在程序执行过程中它们占据固定的存储单元,而不是动态地进行分配和释放。

在动态存储区中存放以下数据:①函数形式参数。在调用函数时给形参分配存储空间。②函数中的自动变量(未加 `static` 声明的局部变量,详见后面的介绍)。③函数调用时的现场保护和返回地址等。

对以上这些数据,在函数调用开始时分配动态存储空间,函数结束时释放这些空间。在程序执行过程中,这种分配和释放是动态的,如果在一个程序中两次调用同一函数,则要进行两次分配和释放,而两次分配给此函数中局部变量的存储空间地址可能是不相同的。

如果在一个程序中包含若干个函数,每个函数中的局部变量的存储期并不等于整个程序的执行周期,它只是整个程序执行周期的一部分。根据函数调用的情况,系统对局部变量动态地分配和释放存储空间。

在 C++ 中变量除了有数据类型的属性之外,还有存储类别(storage class) 的属性。存储类别指的是数据在内存中存储的方法。存储方法分为静态存储和动态存储两大类。具体包含 4 种:自动的(auto)、静态的(static)、寄存器的(register)和外部的(extern)。根据变量的存储类别,可以知道变量的作用域和存储期。

### 4.12.2 自动变量

函数中的局部变量,如果不用关键字 `static` 加以声明,编译系统对它们是动态地分配存储空间的。函数的形参和在函数中定义的变量(包括在复合语句中定义的变量)都属此类。在调用该函数时,系统给形参和函数中定义的变量分配存储空间,数据存储在动态存储区中。在函数调用结束时就自动释放这些空间。如果是在复合语句中定义的变量,则在变量定义时分配存储空间,在复合语句结束时自动释放空间。因此这类局部变量称为自动变量(auto variable)。自动变量用关键字 `auto` 作存储类别的声明。例如:

```
int f(int a) //定义f函数,a为形参
{auto int b,c=3; //定义b和c为整型的自动变量
  ;
}
```

存储类别 `auto` 和数据类型 `int` 的顺序任意。关键字 `auto` 可以省略,如果不写 `auto`,则系统把它默认为自动存储类别,它属于动态存储方式。程序中大多数变量属于自动变量。本书前面各章所介绍的例子中,在函数中定义的变量都没有声明为 `auto`,其实都默认指定为自动变量。在函数体中以下两种写法作用相同:

- ① `auto int b,c=3;`
- ② `int b,c=3;`

### 4.12.3 用 `static` 声明静态局部变量

有时希望函数中的局部变量的值在函数调用结束后不消失而保留原值,即其占用的存储单元不释放,在下一

次该函数调用时,该变量保留上一次函数调用结束时的值。这时就应该指定该局部变量为静态局部变量 (static local variable)。

例 4.12 静态局部变量的值。

```
#include <iostream>
using namespace std;
int f(int a) //定义f函数,a为形参
{
    auto int b=0; //定义b为自动变量
    static int c=3; //定义c为静态局部变量
    b=b+1;
    c=c+1;
    return a+b+c;
}

int main()
{
    int a=2,i;
    for(i=0;i<3;i++)
        cout<<f(a)<<" ";
    cout<<endl;
    return 0;
}
```

运行结果为

7 8 9

先后 3 次调用 f 函数时,b 和 c 的值如书中表 4.1 所示。

|         | b | c |
|---------|---|---|
| 第一次调用开始 | 0 | 3 |
| 第一次调用结束 | 1 | 4 |
| 第二次调用开始 | 0 | 4 |

图 4.14

对静态局部变量的说明:

- (1) 静态局部变量在静态存储区内分配存储单元。在程序整个运行期间都不释放。而自动变量(即动态局部变量)属于动态存储类别,存储在动态存储区空间(而不是静态存储区空间),函数调用结束后即释放。
- (2) 为静态局部变量赋初值是在编译时进行值的,即只赋初值一次,在程序运行时它已有初值。以后每次调用函数时不再重新赋初值而只是保留上次函数调用结束时的值。而为自动变量赋初值,不是在编译时进行的,而是在函数调用时进行,每调用一次函数重新给一次初值,相当于执行一次赋值语句。
- (3) 如果在定义局部变量时不赋初值的话,对静态局部变量来说,编译时自动赋初值 0(对数值型变量)或空字符(对字符型变量)。而对自动变量来说,如果不赋初值,则它的值是一个不确定的值。这是由于每次函数调用



结束后存储单元已释放,下次调用时又重新另分配存储单元,而所分配的单元中的值是不确定的。

(4) 虽然静态局部变量在函数调用结束后仍然存在,但其他函数是不能引用它的,也就是说,在其他函数中它是“不可见”的。

在什么情况下需要用局部静态变量呢?

(1) 需要保留函数上一次调用结束时的值。例如可以用下例中的方法求  $n!$ 。

例 4.13 输出 1~5 的阶乘值(即  $1!, 2!, 3!, 4!, 5!$ )。

```
#include <iostream>
using namespace std;
int fac(int); //函数声明
int main( )
{
    int i;
    for(i=1;i<=5;i++)
        cout<<i<<" != " <<fac(i)<<endl;
    return 0;
}

int fac(int n)
{
    static int f=1; //f 为静态局部变量,函数结束时 f 的值不释放
    f=f*n; //在 f 原值基础上乘以 n
    return f;
}
```

运行结果为

```
1!=1
2!=2
3!=6
4!=24
5!=120
```

每次调用 `fac(i)`,就输出一个 `i`,同时保留这个 `i!` 的值,以便下次再乘  $(i+1)$ 。

(2) 如果初始化后,变量只被引用而不改变其值,则这时用静态局部变量比较方便,以免每次调用时重新赋值。

但是应该看到,用静态存储要多占内存,而且降低了程序的可读性,当调用次数多时往往弄不清静态局部变量的当前值是什么。因此,如不必要,不要多用静态局部变量。

#### 4.12.4 用 register 声明寄存器变量

一般情况下,变量的值是存放在内存中的。当程序中用到哪一个变量的值时,由控制器发出指令将内存中该变量的值送到 CPU 中的运算器。经过运算器进行运算,如果需要存数,再从运算器将数据送到内存存放。如图 4.15 所示。



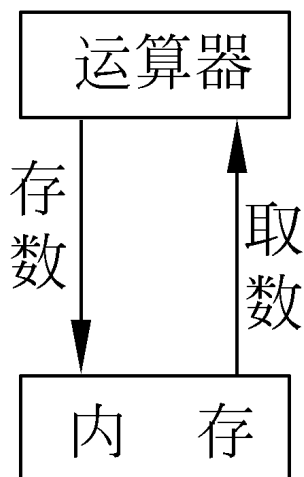


图 4.15

为提高执行效率,C++允许将局部变量的值放在 CPU 中的寄存器中,需要用时直接从寄存器取出参加运算,不必再到内存中去存取。这种变量叫做寄存器变量,用关键字 `register` 作声明。例如,可以将例 4.14 中的 `fac` 函数改写如下:

```

int fac(int n)
{register int i,f=1; //定义 i 和 f 是寄存器变量
  for(i=1;i<=n;i++) f=f*i;
  return f;
}
  
```

定义 `f` 和 `i` 是存放在寄存器的局部变量,如果 `n` 的值大,则能节约许多执行时间。

在程序中定义寄存器变量对编译系统只是建议性(而不是强制性)的。当今的优化编译系统能够识别使用频繁的变量,自动地将这些变量放在寄存器中。

#### 4.12.5 用 `extern` 声明外部变量

全局变量(外部变量)是在函数的外部定义的,它的作用域为从变量的定义处开始,到本程序文件的末尾。在此作用域内,全局变量可以为本文件中各个函数所引用。编译时将全局变量分配在静态存储区。

有时需要用 `extern` 来声明全局变量,以扩展全局变量的作用域。

##### 1. 在一个文件内声明全局变量

如果外部变量不在文件的开头定义,其有效的作用范围只限于定义处到文件终了。如果在定义点之前的函数想引用该全局变量,则应该在引用之前用关键字 `extern` 对该变量作外部变量声明,表示该变量是一个将在下面定义的全局变量。有了此声明,就可以从声明处起,合法地引用该全局变量,这种声明称为提前引用声明。

例 4.14 用 `extern` 对外部变量作提前引用声明,以扩展程序文件中的作用域。

```

#include <iostream>
using namespace std;
int max(int,int); //函数声明
void main()
{extern int a,b; //对全局变量 a,b 作提前引用声明
  cout<<max(a,b)<<endl;
}
int a=15,b=-7; //定义全局变量 a,b
  
```

```
int max(int x,int y)
{int z;
 z=x>y?x:y;
 return z;
}
```

运行结果如下:

15

在 main 后面定义了全局变量 a,b,但由于全局变量定义的位置在函数 main 之后,因此如果没有程序的第 5 行,在 main 函数中是不能引用全局变量 a 和 b 的。现在我们在 main 函数第 2 行用 extern 对 a 和 b 作了提前引用声明,表示 a 和 b 是将在后面定义的变量。这样在 main 函数中就可以合法地使用全局变量 a 和 b 了。如果不作 extern 声明,编译时会出错,系统认为 a 和 b 未经定义。一般都把全局变量的定义放在引用它的所有函数之前,这样可以避免在函数中多加一个 extern 声明。

## 2. 在多文件的程序中声明外部变量

如果一个程序包含两个文件,在两个文件中都要用到同一个外部变量 num,不能分别在两个文件中各自定义一个外部变量 num。正确的做法是:在任一个文件中定义外部变量 num,而在另一文件中用 extern 对 num 作外部变量声明。即

```
extern int num;
```

编译系统由此知道 num 是一个已在别处定义的外部变量,它先在本文件中找有无外部变量 num,如果有,则将其作用域扩展到本行开始(如上节所述),如果本文件中无此外部变量,则在程序连接时从其他文件中找有无外部变量 num,如果有,则把在另一文件中定义的外部变量 num 的作用域扩展到本文件,在本文件中可以合法地引用该外部变量 num。

分析下例:

file1.cppfile2.cpp

```
extern int a,b; int a=3,b=4;
int main() {
{cout<<a<<" , " <<b<<endl;
 return 0;
}
```

用 extern 扩展全局变量的作用域,虽然能为程序设计带来方便,但应十分慎重,因为在执行一个文件中的函数时,可能会改变了该全局变量的值,从而会影响到另一文件中的函数执行结果。

## 4.12.6 用 static 声明静态外部变量

有时在程序设计中希望某些外部变量只限于被本文件引用,而不能被其他文件引用。这时可以在定义外部变量时加一个 static 声明。例如:

file1.cppfile2.cpp

```
static int a=3;extern int a;
int main() int fun(int n)
{{
| a=a*n;
|
}}
```

这种加上 static 声明、只能用于本文件的外部变量(全局变量)称为静态外部变量。这就为程序的模块化、通用性提供了方便。如果已知道其他文件不需要引用本文件的全局变量,可以对本文件中的全局变量都加上

static,成为静态外部变量,以免被其他文件误用。

需要指出,不要误认为用 static 声明的外部变量才采用静态存储方式(存放在静态存储区中),而不加 static 的是动态存储(存放在动态存储区)。实际上,两种形式的外部变量都用静态存储方式,只是作用范围不同而已,都是在编译时分配内存的。

#### 4.13 变量属性小结

一个变量除了数据类型以外,还有 3 种属性:

- (1) 存储类别 C++允许使用 auto,static,register 和 extern 4 种存储类别。
- (2) 作用域 指程序中可以引用该变量的区域。
- (3) 存储期 指变量在内存的存储期限。

以上 3 种属性是有联系的,程序设计者只能声明变量的存储类别,通过存储类别可以确定变量的作用域和存储期。

要注意存储类别的用法。auto, static 和 register 3 种存储类别只能用于变量的定义语句中,如

```
auto char c; //字符型自动变量,在函数内定义
static int a; //静态局部整型变量或静态外部整型变量
register int d; //整型寄存器变量,在函数内定义
extern int b; //声明一个已定义的外部整型变量
```

说明: extern 只能用来声明已定义的外部变量,而不能用于变量的定义。只要看到 extern,就可以判定这是变量声明,而不是定义变量的语句。

下面从不同角度分析它们之间的联系。

- (1) 从作用域角度分,有局部变量和全局变量。它们采用的存储类别如下:

- 局部变量

自动变量,即动态局部变量(离开函数,值就消失)

静态局部变量(离开函数,值仍保留)

寄存器变量(离开函数,值就消失)

形式参数(可以定义为自动变量或寄存器变量)

- 全局变量

静态外部变量(只限本文件引用)

外部变量(即非静态的外部变量,允许其他文件引用)

- (2) 从变量存储期(存在的时间)来区分,有动态存储和静态存储两种类型。静态存储是程序整个运行时间都存在,而动态存储则是在调用函数时临时分配单元。

- 动态存储

自动变量(本函数内有效)

寄存器变量(本函数内有效)

形式参数

- 静态存储

静态局部变量(函数内有效)

静态外部变量(本文件内有效)

外部变量(其他文件可引用)

- (3) 从变量值存放的位置来区分,可分为

- 内存中静态存储区

静态局部变量

静态外部变量(函数外部静态变量)

外部变量(可为其他文件引用)

● 内存中动态存储区: 自动变量和形式参数

● CPU 中的寄存器: 寄存器变量

(4) 关于作用域和存储期的概念。从前面叙述可以知道,对一个变量的性质可以从两个方面分析,一是从变量的作用域,一是从变量值存在时间的长短,即存储期。前者是从空间的角度,后者是从时间的角度。二者有联系但不是同一回事。图 4.16 是作用域的示意图,图 4.17 是存储期的示意图。

文件 file1.cpp

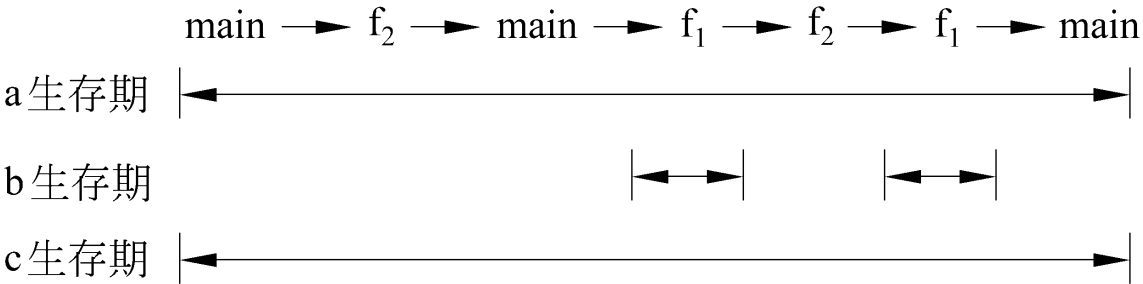
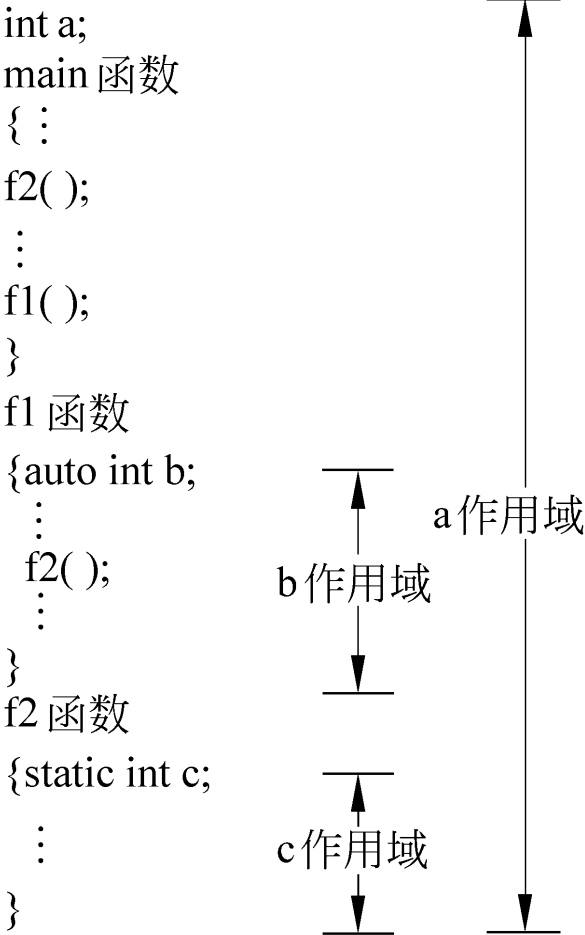


图 4.16

图 4.17

如果一个变量在某个文件或函数范围内是有效的,则称该文件或函数为该变量的作用域,在此作用域内可以引用该变量,所以又称变量在此作用域内“可见”,这种性质又称为变量的可见性,例如图 4.16 中变量 a、b 在函

数 f1 中可见。

如果一个变量值在某一时刻是存在的,则认为这一时刻属于该变量的存储期,或称该变量在此时刻“存在”。书中表 4.2 表示各种类型变量的作用域和存在性的情况。

可以看到自动变量和寄存器变量在函数内的可见性和存在性是一致的。在函数外的可见性和存在性也是一致的。静态局部变量在函数外的可见性和存在性不一致。静态外部变量和外部变量的可见性和存在性是一致的。

如果一个变量在某个文件或函数范围内是有效的,则称该文件或函数为该变量的作用域,在此作用域内可以引用该变量,所以又称变量在此作用域内“可见”,这种性质又称为变量的可见性,例如图 4.16 中变量 a、b 在函数 f1 中可见。

如果一个变量值在某一时刻是存在的,则认为这一时刻属于该变量的存储期,或称该变量在此时刻“存在”。书中表 4.2 表示各种类型变量的作用域和存在性的情况。

可以看到自动变量和寄存器变量在函数内的可见性和存在性是一致的。在函数外的可见性和存在性也是一致的。静态局部变量在函数外的可见性和存在性不一致。静态外部变量和外部变量的可见性和存在性是一致的。

(5) static 声明使变量采用静态存储方式,但它对局部变量和全局变量所起的作用不同。对局部变量来说,static 使变量由动态存储方式改变为静态存储方式。而对全局变量来说,它使变量局部化(局部于本文件),但仍为静态存储方式。从作用域角度看,凡有 static 声明的,其作用域都是局限的,或者局限于本函数内(静态局部变量),或者局限于本文件内(静态外部变量)。

#### 4.14 关于变量的声明和定义

由第 2 章已经知道,一个函数一般由两部分组成: (1)声明部分; (2)执行语句。声明部分的作用是对有关的标识符(如变量、函数、结构体、共用体等)的属性进行说明。对于函数,声明和定义的区别是明显的,在本章 4.4.3 节中已说明,函数的声明是函数的原型,而函数的定义是函数功能的确立。对函数的声明是可以放在声明部分中的,而函数的定义显然不在函数的声明部分范围内,它是一个文件中的独立模块。

对变量而言,声明与定义的关系稍微复杂一些。在声明部分出现的变量有两种情况:一种是需要建立存储空间的(如 `int a;`);另一种是不需要建立存储空间的(如 `extern int a;`)。前者称为定义性声明(defining declaration),或简称为定义(definition)。后者称为引用性声明(referenceing declaration)。广义地说,声明包括定义,但并非所有的声明都是定义。对“`int a;`”而言,它是定义性声明,既可说是声明,又可说是定义。而对“`extern int a;`”而言,它是声明而不是定义。一般为了叙述方便,把建立存储空间的声明称为定义,而把不需要建立存储空间的声明称为声明。显然这里指的声明是狭义的,即非定义性声明。例如:

```
int main()  
{extern int a; //这是声明不是定义。声明 a 是一个已定义的外部变量  
...  
}  
int a; //是定义,定义 a 为整型外部变量
```

外部变量定义和外部变量声明的含义是不同的。外部变量的定义只能有一次,它的位置在所有函数之外,而同一文件中的外部变量的声明可以有多次,它的位置可以在函数之内,也可以在函数之外。系统根据外部变量的定义分配存储单元。对外部变量的初始化只能在定义时进行,而不能在声明中进行。所谓声明,其作用是向编译系统发出一个信息,声明该变量是一个在后面定义的外部变量,仅仅是为了提前引用该变量而作的声明。extern 只用作声明,而不用作定义。

用 static 来声明一个变量的作用有二: (1)对局部变量用 static 声明,使该变量在本函数调用结束后不释放,整个程序执行期间始终存在,使其存储期为程序的全过程。(2)全局变量用 static 声明,则该变量的作用域只限于本文件模块(即被声明的文件中)。

请注意,用 `auto`,`register`,`static` 声明变量时,是在定义变量的基础上加上这些关键字,而不能单独使用。如“`static a;`”是不合法的,应写成“`static int a;`”。

#### 4.15 内部函数和外部函数

函数本质上是全局的,因为一个函数要被另外的函数调用,但是,也可以指定函数只能被本文件调用,而不能被其他文件调用。根据函数能否被其他源文件调用,将函数区分为内部函数和外部函数。

##### 4.15.1 内部函数

如果一个函数只能被本文件中其他函数所调用,它称为内部函数。在定义内部函数时,在函数名和函数类型的前面加 `static`。函数首部的一般格式为

`static` 类型标识符 函数名(形参表)

如

`static int fun(int a,int b)`

内部函数又称静态(`static`)函数。使用内部函数,可以使函数只局限于所在文件。如果在不同的文件中有同名的内部函数,互不干扰。通常把只能由同一文件使用的函数和外部变量放在一个文件中,在它们前面都冠以 `static` 使之局部化,其他文件不能引用。

##### 4.15.2 外部函数

(1) 在定义函数时,如果在函数首部的最左端冠以关键字 `extern`,则表示此函数是外部函数,可供其他文件调用。

如函数首部可以写为

`extern int fun (int a, int b)`

这样,函数 `fun` 就可以为其他文件调用。如果在定义函数时省略 `extern`,则默认为外部函数。本书前面所用的函数都是外部函数。

(2) 在需要调用此函数的文件中,用 `extern` 声明所用的函数是外部函数。

例 4.15 输入两个整数,要求输出其中的大者。用外部函数实现。

file1.cpp(文件 1)

```
#include <iostream>
```

```
using namespace std;
```

```
int main( )
```

```
{extern int max(int,int); //声明在本函数中将要调用在其他文件中定义的 max 函数
```

```
int a,b;
```

```
cin>>a>>b;
```

```
cout<<max(a,b)<<endl;
```

```
return 0;
```

```
}
```

file2.cpp(文件 2)

```
int max(int x,int y)
```

```
{int z;
```

```
z=x>y?x:y;
```

```
return z;
```

```
}
```

运行情况如下:

7 -34✓

7



在计算机上运行一个含多文件的程序时,需要建立一个项目文件(project file),在该项目文件中包含程序的各个文件。详细情况请参阅本书的配套书《C++程序设计习题解答与上机指导》。

通过此例可知:使用 `extern` 声明就能够在文件中调用其他文件中定义的函数,或者说把该函数的作用域扩展到本文件。`extern` 声明的形式就是在函数原型基础上加关键字 `extern`。由于函数在本质上是外部的,在程序中经常要调用其他文件中的外部函数,为方便编程,C++允许在声明函数时省写 `extern`。例 4.16 程序 `main` 函数中的函数声明可写成

```
int max(int,int);
```

这就是我们多次用过的函数原型。由此可以进一步理解函数原型的作用。用函数原型能够把函数的作用域扩展到定义该函数的文件之外(不必使用 `extern`)。只要在使用该函数的每一个文件中包含该函数的函数原型即可。函数原型通知编译系统:该函数在本文件中稍后定义,或在另一文件中定义。

利用函数原型扩展函数作用域最常见的例子是`#include` 命令的应用。在`#include` 命令所指定的头文件中包含有调用库函数时所需的信息。例如,在程序中需要调用 `sin` 函数,但三角函数并不是由用户在本文件中定义的,而是存放在数学函数库中的。按以上的介绍,必须在本文件中写出 `sin` 函数的原型,否则无法调用 `sin` 函数。`sin` 函数的原型是

```
double sin(double x);
```

本来应该由程序设计者在调用库函数时先从手册中查出所用的库函数的原型,并在程序中一一写出来,但这显然是麻烦而困难的。为减少程序设计者的困难,在头文件 `cmath` 中包括了所有数学函数的原型和其他有关信息,用户只需用以下`#include` 命令:

```
#include <cmath>
```

即可。这时,在该文件中就能合法地调用各数学库函数了。

#### 4.16 预处理命令

可以在 C++源程序中加入一些“预处理命令”(preprocessor directives),以改进程序设计环境,提高编程效率。预处理命令是 C++统一规定的,但是它不是 C++语言本身的组成部分,不能直接对它们进行编译(因为编译程序不能识别它们)。

现在使用的 C++编译系统都包括了预处理、编译和连接等部分,因此不少用户误认为预处理命令是 C++语言的一部分,甚至以为它们是 C++语句,这是不对的。必须正确区别预处理命令和 C++语句,区别预处理和编译,才能正确使用预处理命令。C++与其他高级语言的一个重要区别是可以使用预处理命令和具有预处理的功能。

C++提供的预处理功能主要有以下 3 种:

- (1) 宏定义
- (2) 文件包含
- (3) 条件编译

分别用宏定义命令、文件包含命令、条件编译命令来实现。为了与一般 C++语句相区别,这些命令以符号“`#`”开头,而且末尾不包含分号。

##### 4.16.1 宏定义

可以用`#define` 命令将一个指定的标识符(即宏名)来代表一个字符串。定义宏的作用一般是用一个短的名字代表一个长的字符串。它的一般形式为

```
#define 标识符 字符串
```

这就是已经介绍过的定义符号常量。如

```
#define PI 3.1415926
```

还可以用`#define` 命令定义带参数的宏定义。其定义的一般形式为

```
#define 宏名(参数表) 字符串
```

如

`#define S(a,b) a*b`//定义宏 S(矩形面积),a、b 为宏的参数

使用的形式如下:

`area=S(3,2)`

用 3、2 分别代替宏定义中的形式参数 a 和 b,即用 3\*2 代替 S(3,2)。因此赋值语句展开为

`area=3*2;`

由于 C++增加了内置函数(`inline`),比用带参数的宏定义更方便,因此在 C++中基本上已不再用`#define` 命令定义宏了,主要用于条件编译中。

#### 4.16.2 “文件包含”处理

##### 1.“文件包含”的作用

所谓“文件包含”处理是指一个源文件可以将另外一个源文件的全部内容包含进来,即将另外的文件包含到本文件之中。C++提供了`#include` 命令用来实现“文件包含”的操作。如在 `file1.cpp` 中有以下`#include` 命令:

`#include "file2.cpp"`

它的作用见图 4.18 示意。

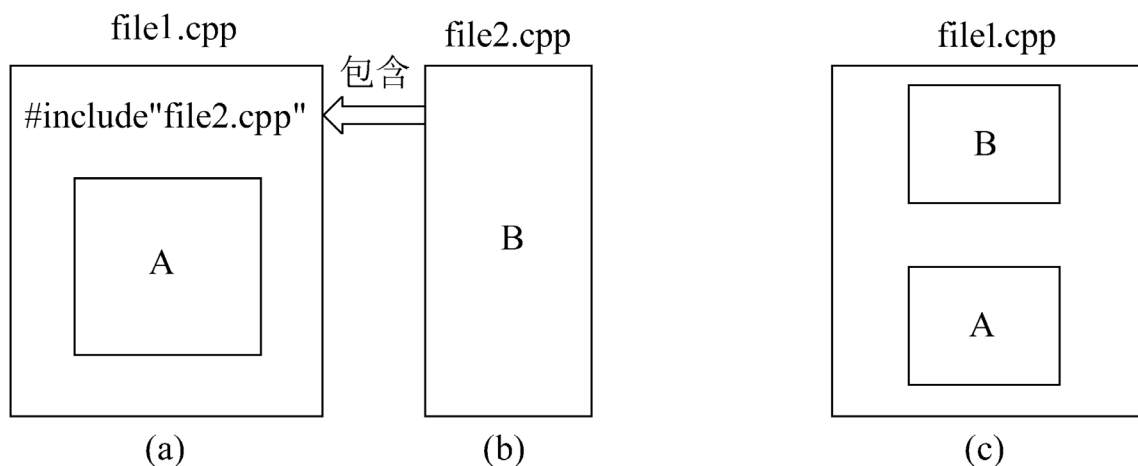


图 4.18

“文件包含”命令是很有用的,它可以节省程序设计人员的重复劳动。

`#include` 命令的应用很广泛,绝大多数 C++程序中都包括`#include` 命令。现在,库函数的开发者把这些信息写在一个文件中,用户只需将该文件“包含”进来即可(如调用数学函数的,应包含 `cmath` 文件),这就大大简化了程序,写一行`#include` 命令的作用相当于写几十行、几百行甚至更多行的内容。这种常用在文件头部的被包含的文件称为“标题文件”或“头部文件”。

头文件一般包含以下几类内容:

- (1) 对类型的声明。
- (2) 函数声明。
- (3) 内置(`inline`)函数的定义。
- (4) 宏定义。用`#define` 定义的符号常量和用 `const` 声明的常变量。
- (5) 全局变量定义。
- (6) 外部变量声明。如 `extern int a;`
- (7) 还可以根据需要包含其他头文件。

不同的头文件包括以上不同的信息,提供给程序设计者使用,这样,程序设计者不需自己重复书写这些信息,



只需用一行#include 命令就把这些信息包含到本文件了,大大地提高了编程效率。由于有了#include 命令,就把不同的文件组合在一起,形成一个文件。因此说,头文件是源文件之间的接口。

## 2. include 命令的两种形式

在#include 命令中,文件名除了可以用尖括号括起来以外,还可以用双撇号括起来。#include 命令的一般形式为

```
#include <文件名>
```

或

```
#include " 文件名 "
```

如

```
#include <iostream>
```

或

```
#include " iostream "
```

都是合法的。二者的区别是:用尖括号时,系统到系统目录中寻找要包含的文件,如果找不到,编译系统就给出出错信息。

有时被包含的文件不一定在系统目录中,这时应该用双撇号形式,在双撇号中指出文件路径和文件名。

如果在双撇号中没有给出绝对路径,如#include " file2.c" 则默认指用户当前目录中的文件。系统先在用户当前目录中寻找要包含的文件,若找不到,再按标准方式查找。如果程序中要包含的是用户自己编写的文件,宜用双撇号形式。

对于系统提供的头文件,既可以用尖括号形式,也可以用双撇号形式,都能找到被包含的文件,但显然用尖括号形式更直截了当,效率更高。

## 3. 关于 C++标准库

在 C++编译系统中,提供了许多系统函数和宏定义,而对函数的声明则分别存放在不同的头文件中。如果要调用某一个函数,就必须用#include 命令将有关的头文件包含进来。C++的库除了保留 C 的大部分系统函数和宏定义外,还增加了预定义的模板和类。但是不同 C++库的内容不完全相同,由各 C++编译系统自行决定。不久前推出的 C++标准将库的建设也纳入标准,规范化了 C++标准库,以便使 C++程序能够在不同的 C++平台上工作,便于互相移植。新的 C++标准库中的头文件一般不再包括后缀.h,例如

```
#include <string>
```

但为了使大批已有的 C 程序能继续使用,许多 C++编译系统保留了 C 的头文件,即提供两种不同的头文件,由程序设计者选用。如

```
#include <iostream.h> //C 形式的头文件
```

```
#include <iostream> //C++形式的头文件
```

效果基本上是一样的。建议尽量用符合 C++标准的形式,即在包含 C++头文件时一般不用后缀。如果用户自己编写头文件,可以用.h 为后缀。

### 4.16.3 条件编译

一般情况下,在进行编译时对源程序中的每一行都要编译。但是有时希望程序中某一部分内容只在满足一定条件时才进行编译,也就是指定对程序中的一部分内容进行编译的条件。如果不满足这个条件,就不编译这部分内容。这就是“条件编译”。

有时,希望当满足某条件时对一组语句进行编译,而当条件不满足时则编译另一组语句。

条件编译命令常用的有以下形式:

(1)

```
#ifdef 标识符
```

程序段 1

`#else`

程序段 2

`#endif`

它的作用是当所指定的标识符已经被`#define` 命令定义过,则在程序编译阶段只编译程序段 1,否则编译程序段 2。`#endif` 用来限定`#ifdef` 命令的范围。其中`#else` 部分也可以没有。

(2)

`#if` 表达式

程序段 1

`#else`

程序段 2

`#endif`

它的作用是当指定的表达式值为真(非零)时就编译程序段 1,否则编译程序段 2。可以事先给定一定条件,使程序在不同的条件下执行不同的功能。

例 4.16 在调试程序时,常常希望输出一些所需的信息,而在调试完成后不再输出这些信息。可以在源程序中插入条件编译段。下面是一个简单的示例。

```
#include <iostream>
```

```
using namespace std;
```

```
#define RUN //在调试程序时使之成为注释行
```

```
int main()
```

```
{ int x=1,y=2,z=3;
```

```
#ifndef RUN //本行为条件编译命令
```

```
cout<<" x=" <<x<<" ,y=" <<y<<" ,z=" <<z; //在调试程序时需要输出这些信息
```

```
#endif //本行为条件编译命令
```

```
cout<<" x*y*z=" *y*z<<endl;
```

```
}
```

第 3 行用`#define` 命令的目的不在于用 `RUN` 代表一个字符串,而只是表示已定义过 `RUN`,因此 `RUN` 后面写什么字符串都无所谓,甚至可以不写字符串。在调试程序时去掉第 3 行(或在行首加`//`,使之成为注释行),由于无此行,故未对 `RUN` 定义,第 6 行据此决定编译第 7 行,运行时输出 `x,y,z` 的值,以便用户分析有关变量当前的值。运行程序输出:

```
x=1,y=2,z=3
```

```
x*y*z=6
```

在调试完成后,在运行之前,加上第 3 行,重新编译,由于此时 `RUN` 已被定义过,则该 `cout` 语句不被编译,因此在运行时不再输出 `x,y,z` 的值。运行情况为:

```
x*y*z=6
```

# 第 5 章 数组

- 5.1 数组的概念
- 5.2 一维数组的定义和引用
- 5.3 二维数组的定义和引用
- 5.4 用数组名作函数参数
- 5.5 字符数组
- \*5.6 C++处理字符串的方法——字符串类与字符串变量

## 第 5 章 数组

### 5.1 数组的概念

概括地说：数组是有序数据的集合。要寻找一个数组中的某一个元素必须给出两个要素,即数组名和下标。数组名和下标惟一地标识一个数组中的一个元素。

数组是有类型属性的。同一数组中的每一个元素都必须属于同一数据类型。一个数组在内存中占一片连续的存储单元。如果有一个整型数组 `a`,假设数组的起始地址为 2000,则该数组在内存中的存储情况如图 5.1 所示。

## 第 5 章 数组

图 5.1

| 数组 a |      |
|------|------|
| 2000 | a[0] |
| 2002 | a[1] |
| 2004 | a[2] |
| 2006 | a[3] |
| 2008 | a[4] |
| 2010 | a[5] |
| 2012 | a[6] |
| 2014 | a[7] |
| 2016 | a[8] |
| 2018 | a[9] |

## 第 5 章 数组

引入数组就不需要在程序中定义大量的变量,大大减少程序中变量的数量,使程序精炼,而且数组含义清楚,使用方便,明确地反映了数据间的联系。许多好的算法都与数组有关。熟练地利用数组,可以大大地提高编程和解题的效率,加强了程序的可读性。

C++用方括号来表示下标,如用 `s[1],s[2],s[3]` 分别代表 `s1,s2,s3`。

## 第 5 章 数组

### 5.2 一维数组的定义和引用

#### 5.2.1 定义一维数组

定义一维数组的一般格式为

类型标识符 数组名[常量表达式];

例如

```
int a[10];
```

它表示数组名为 a,此数组为整型,有 10 个元素。

说明:

(1) 数组名定名规则和变量名相同,遵循标识符定名规则。

## 第 5 章 数组

(2) 用方括号括起来的常量表达式表示下标值,如下面的写法是合法的:

```
int a[10];
```

```
int a[2*5];
```

```
int a[n*2]; //假设前面已定义了 n 为常量
```

(3) 常量表达式的值表示元素的个数,即数组长度。例如,在“`int a[10];`”中,10 表示 a 数组有 10 个元素,下标从 0 开始,这 10 个元素是: `a[0]`,`a[1]`,`a[2]`,`a[3]`,`a[4]`,`a[5]`,`a[6]`,`a[7]`, `a[8]`,`a[9]`。注意最后一个元素是 `a[9]`而不是 `a[10]`。

## 第 5 章 数组

(4) 常量表达式中可以包括常量、常变量和符号常量,但不能包含变量。也就是说,C++不允许对数组的大小作动态定义,即数组的大小不依赖于程序运行过程中变量的值。例如,下面这样定义数组是不行的:

```
int n;
```

```
cin>>n; //输入 a 数组的长度
```

```
int a[n]; //企图根据 n 的值决定数组的长度
```

如果把第 1,2 行改为下面一行就合法了:

```
const int n=5;
```

## 第 5 章 数组

### 5.2.2 引用一维数组的元素

数组必须先定义,然后使用。只能逐个引用数组元素的值而不能一次引用整个数组中的全部元素的值。

数组元素的表示形式为

数组名[下标]

下标可以是整型常量或整型表达式。例如

```
a[0]= a[5]+ a[7]- a[2*3]
```

## 第 5 章 数组

例 5.1 数组元素的引用。

```
#include <iostream>
```

```
using namespace std;
```

```
int main( )
```

```
{int i,a[10];
```

```
    for (i=0;i<=9;i++)
```

```
a[i]=i;
```

```
    for (i=9;i>=0;i--)
```

```
cout<<a[i]<<"  " ;
```

```
    cout<<endl;
```

```
    return 0;
```

```
}
```

运行结果如下:

```
9 8 7 6 5 4 3 2 1 0
```

程序使 `a[0]~a[9]`的值为 0~9,然后按逆序输出。

## 第 5 章 数组

### 5.2.3 一维数组的初始化

(1) 在定义数组时分别对数组元素赋予初值。例如

```
int a[10]={0,1,2,3,4,5,6,7,8,9};
```

(2) 可以只给一部分元素赋值。例如

```
int a[10]={0,1,2,3,4};
```

(3) 如果想使一个数组中全部元素值为 1,可以写成

```
int a[10]={1,1,1,1,1,1,1,1,1,1};
```

不能写成 `int a[10]={1*10};`

不能给数组整体赋初值。

(4) 在对全部数组元素赋初值时,可以不指定数组长度。例如

```
int a[5]={1,2,3,4,5};
```

可以写成 `int a[]={1,2,3,4,5};`

## 第 5 章 数组

### 5.2.4 一维数组程序举例

例 5.2 用数组来处理求 Fibonacci 数列问题。

可以用 20 个元素代表数列中的 20 个数,从第 3 个数开始,可以直接用表达式  $f[i]=f[i-2]+f[i-1]$  求出各数。

程序如下:

```
#include <iostream>
```

```
#include <iomanip>
```

```
using namespace std;
```

```
int main( )
```

```
{ int i;
```

```
int f[20]={1,1};//f[0]=1,f[1]=1
```

```
for(i=2;i<20;i++)
```

## 第 5 章 数组

```
f[i]=f[i-2]+f[i-1];//在 i 的值为 2 时,f[2]=f[0]+f[1],依此类推
```

```
for(i=0;i<20;i++) //此循环的作用是输出 20 个数
```

```
{if(i%5==0) cout<<endl;//控制换行,每行输出 5 个数据
```

```
cout<<setw(8)<<f[i];//每个数据输出时占 8 列宽度
```

```
}
```

```
cout<<endl; //最后执行一次换行
```

```
return 0;
```

```
}
```

运行结果如下:

(空一行)

1 1 2 3 5

8 13 21 34 55

89 144 233 377 610

9871597258441816765

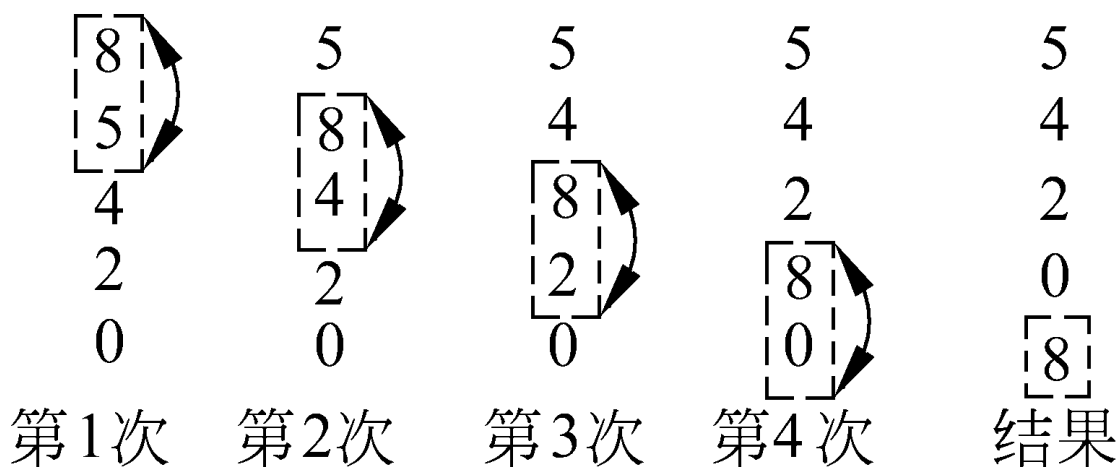
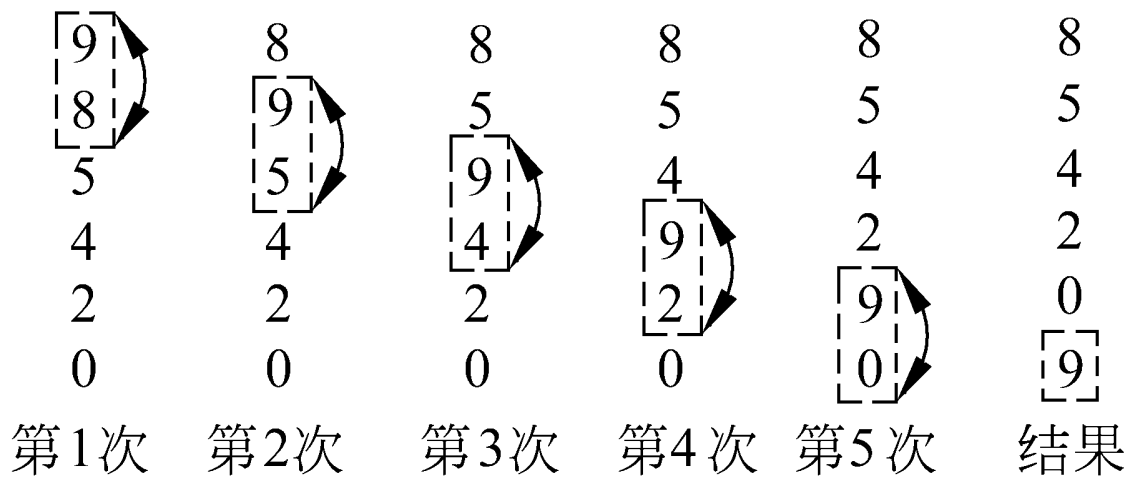
## 第 5 章 数组

例 5.3 编写程序,用起泡法对 10 个数排序(按由小到大顺序)。

起泡法的思路是:将相邻两个数比较,将小的调到前头。见图 5.2。

然后进行第 2 趟比较,对余下的前面 5 个数按上法进行比较,见图 5.3。

图 5.2 图 5.3



## 第5章 数组

可以推知,如果有  $n$  个数,则要进行  $n-1$  趟比较(和交换)。在第 1 趟中要进行  $n-1$  次两两比较,在第  $j$  趟中要进行  $n-j$  次两两比较。

根据以上思路写出程序,今设  $n=10$ ,本例定义数组长度为 11, $a[0]$ 不用,只用  $a[1]\sim a[10]$ ,以符合人们的习惯。从前面的叙述可知,应该进行 9 趟比较和交换。

```
#include <iostream>
using namespace std;
int main()
{
    int a[11];
    int i,j,t;
    cout<<" input 10 numbers : " <<endl;
    for (i=1;i<11;i++) //输入 a[1]~a[10]
```

## 第5章 数组

```
    cin>>a[i];
    cout<<endl;
    for (j=1;j<=9;j++) //共进行9趟比较
    for(i=1;i<=10-j;i++)//在每趟中要进行(10-j)次两两比较
    if (a[i]>a[i+1]) //如果前面的数大于后面的数
    {t=a[i];a[i]=a[i+1];a[i+1]=t;} //交换两个数的位置,使小数上浮
    cout<<" the sorted numbers : " <<endl;
```

```

    for(i=1;i<11;i++) //输出 10 个数
    cout<<a[i]<<" ";
    cout<<endl;
    return 0;
}

```

运行情况如下:

input 10 numbers:

3 5 9 11 33 6 -9 -76 100 123↵

the sorted numbers:

-76 -9 3 5 6 9 11 33 100 123

## 第 5 章 数组

### 5.3 二维数组的定义和引用

具有两个下标的数组称为二维数组。有些数据要依赖于两个因素才能唯一地确定,例如有 3 个学生,每个学生有 4 门课的成绩,显然,成绩数据是一个二维表,如书中表 5.1 所示。

想表示第 3 个学生第 4 门课的成绩,就需要指出学生的序号和课程的序号两个因素,在数学上以  $s_{3,4}$  表示。

在 C++ 中以 `s[3][4]` 表示,它代表数据 73。

## 第 5 章 数组

### 5.3.1 定义二维数组

定义二维数组的一般形式为

类型标识符 数组名[常量表达式][常量表达式]

例如

```
float a[3][4],b[5][10];
```

定义 `a` 为  $3 \times 4$  (3 行 4 列) 的单精度数组, `b` 为  $5 \times 10$  (5 行 10 列) 的单精度数组。注意不能写成“`float a[3,4],b[5,10];`”。C++ 对二维数组采用这样的定义方式,使我们可以把二维数组看作是一种特殊的一维数组:它的元素又是一个一维数组。例如,可以把 `a` 看作是一个一维数组,它有 3 个元素:`a[0]`,`a[1]`,`a[2]`,每个元素又是一个包含 4 个元素的一维数组,见图 5.4。`a[0]`,`a[1]`,`a[2]` 是 3 个一维数组的名字。

## 第 5 章 数组

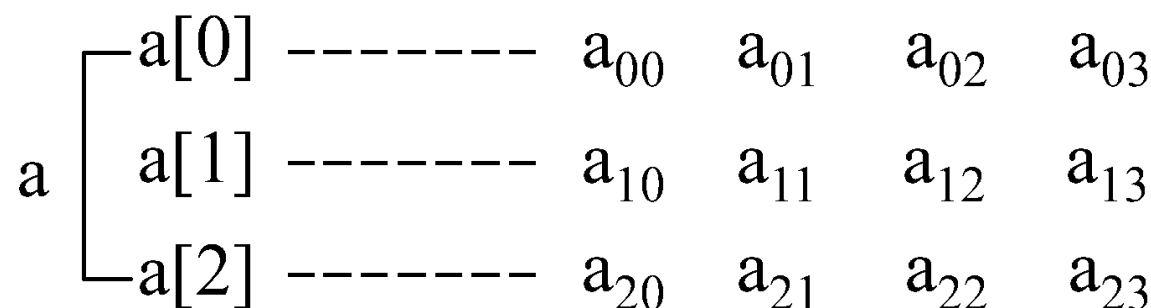


图 5.4

上面定义的二维数组可以理解为定义了 3 个一维数组,即相当于

```
float a[0][4],a[1][4],a[2][4]
```

此处把 `a[0]`,`a[1]`,`a[2]` 作一维数组名。C++ 的这种处理方法在数组初始化和用指针表示时显得很方便,这在以后会体会到。

## 第 5 章 数组

C++ 中,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一行的元素,再存放第二行的元素。图 5.5 表示对 `a[3][4]` 数组存放的顺序。

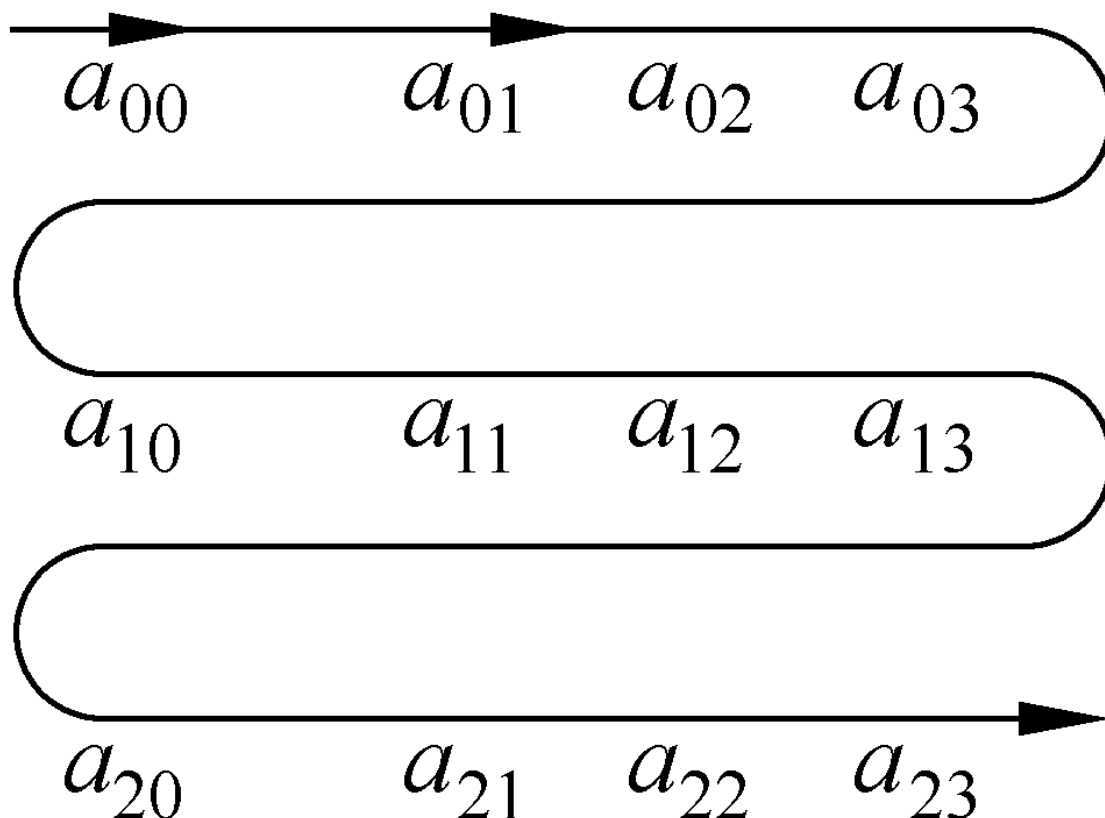


图 5.5

C++ 允许使用多维数组。有了二维数组的基础,再掌握多维数组是不困难的。例如,定义三维数组的方法是

## 第 5 章 数组

**float** a[2][3][4];

定义 **float** 型三维数组 **a**,它有  $2 \times 3 \times 4 = 24$  个元素。多维数组元素在内存中的排列顺序:第一维的下标变化最慢,最右边的下标变化最快。例如,上述三维数组的元素排列顺序为

$a[0][0][0] \rightarrow a[0][0][1] \rightarrow a[0][0][2] \rightarrow a[0][0][3] \rightarrow a[0][1][0] \rightarrow a[0][1][1] \rightarrow a[0][1][2] \rightarrow a[0][1][3] \rightarrow a[0][2][0] \rightarrow a[0][2][1] \rightarrow a[0][2][2] \rightarrow a[0][2][3] \rightarrow a[1][0][0] \rightarrow a[1][0][1] \rightarrow a[1][0][2] \rightarrow a[1][0][3] \rightarrow a[1][1][0] \rightarrow a[1][1][1] \rightarrow a[1][1][2] \rightarrow a[1][1][3] \rightarrow a[1][2][0] \rightarrow a[1][2][1] \rightarrow a[1][2][2] \rightarrow a[1][2][3]$

## 第 5 章 数组

### 5.3.2 二维数组的引用

二维数组的元素的表示形式为

数组名 [下标][下标]

如  $a[2][3]$ 。下标可以是整型表达式,如  $a[2-1][2*2-1]$ 。不要写成  $a[2,3]$ , $a[2-1,2*2-1]$  形式。

数组元素是左值,可以出现在表达式中,也可以被赋值,例如

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

在使用数组元素时,应该注意下标值应在已定义的数组大小的范围内。常出现的错误是

## 第 5 章 数组

**int** a[3][4]; //定义 3 行 4 列的数组

⋮

$a[3][4] = 15;$  //引用  $a[3][4]$  元素

定义 **a** 为  $3 \times 4$  的数组,它可用的行下标值最大为 2,列坐标值最大为 3。最多可以用到  $a[2][3]$ , $a[3][4]$  就超过了数组的范围。

请严格区分在定义数组时用的  $a[3][4]$  和引用元素时的  $a[3][4]$  的区别。前者  $a[3][4]$  用来定义数组的维数和各



维的大小,后者 a[3][4]中的 3 和 4 是下标值,a[3][4]代表某一个元素。

## 第 5 章 数组

### 5.3.3 二维数组的初始化

可以用下面的方法对二维数组初始化:

(1) 分行给二维数组赋初值。如

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

这种赋初值方法比较直观,把第 1 个花括号内的数据赋给第 1 行的元素,第 2 个花括号内的数据赋给第 2 行的元素.....即按行赋初值。

(2) 可以将所有数据写在一个花括号内,按数组排列的顺序对各元素赋初值。如

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

效果与前相同。但以第 1 种方法为好,一行对一行,界限清楚。用第 2 种方法如果数据多,写成一大片,容易遗漏,也不易检查。

## 第 5 章 数组

(3) 可以对部分元素赋初值。如

```
int a[3][4]={ {1},{5},{9}};
```

它的作用是只对各行第 1 列的元素赋初值,其余元素值自动置为 0。赋初值后数组各元素为

```
1  0  0  0
```

```
5  0  0  0
```

```
9  0  0  0
```

也可以对各行中的某一元素赋初值:

```
int a[3][4]={ {1},{0,6},{0,0,11}};
```

初始化后的数组元素如下:

```
1  0  0  0
```

```
0  6  0  0
```

```
0  0 11  0
```

## 第 5 章 数组

这种方法对非 0 元素少时比较方便,不必将所有的 0 都写出来,只需输入少量数据。也可以只对某几行元素赋初值:

```
int a[3][4]={ {1},{5,6}};
```

数组元素为

```
1  0  0  0
```

```
5  6  0  0
```

```
0  0  0  0
```

第 3 行不赋初值。也可以对第 2 行不赋初值:

```
int a[3][4]={ {1},{},{9}};
```

(4) 如果对全部元素都赋初值(即提供全部初始数据),则定义数组时对第一维的长度可以不指定,但第二维的长度不能省。如

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

## 第 5 章 数组

可以写成

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

系统会根据数据总个数分配存储空间,一共 12 个数据,每行 4 列,当然可确定为 3 行。

在定义时也可以只对部分元素赋初值而省略第一维的长度,但应分行赋初值。如

```
int a[][4]={ {0,0,3},{},{0,10}};
```

这样的写法,能通知编译系统: 数组共有 3 行。数组各元素为

```
0  0  3  0
```

```
0  0  0  0
```

0 10 0 0

C++在定义数组和表示数组元素时采用 `a[][]`这种两个方括号的方式,对数组初始化时十分有用,它使概念清楚,使用方便,不易出错。

## 第5章 数组

### 5.3.4 二维数组程序举例

例 5.4 将一个二维数组行和列元素互换,存到另一个二维数组中。例如

```
a=  1   2   3 b= 1   4
    4   5   6   2   5
                3   6
```

程序如下:

```
#include <iostream>
using namespace std;
int main( )
{
    int a[2][3]={ {1,2,3},{4,5,6}};
    int b[3][2],i,j;
    cout<<" array a:" <<endl;
    for (i=0;i<=1;i++)
    {
        for (j=0;j<=2;j++)
        {   cout<<a[i][j]<<"   ";
            b[j][i]=a[i][j];
        }
        cout<<endl;
    }
    cout<<" array b:" <<endl;
    for (i=0;i<=2;i++)
    {
        for(j=0;j<=1;j++)
        cout<<b[i][j]<<"   ";
        cout<<endl;
    }
    return 0;
}
```

## 第5章 数组

运行结果如下:

```
array a:
1  2  3
4  5  6
array b:
1  4
2  5
3  6
```

## 第5章 数组

例 5.5 有一个 3×4 的矩阵,要求程序求出其中值最大的那个元素的值,以及其所在的行号和列号。

开始时把 `a[0][0]`的值赋给变量 `max`,然后让下一个元素与它比较,将二者中值大者保存在 `max` 中,然后再让下

一个元素与新的 max 比,直到最后一个元素比完为止。max 最后的值就是数组所有元素中的最大值。  
程序如下:

```
#include <iostream>
using namespace std;
int main( )
{ int i,j,row=0,column=0,max;
  int a[3][4]={{5,12,23,56},{19,28,37,46},{-12,-34,6,8}};
  max=a[0][0];//使max开始时取a[0][0]的值
第5章 数组
  for (i=0;i<=2;i++)//从第0行~第2行
    for (j=0;j<=3;j++) //从第0列~第3列
      if (a[i][j]>max) //如果某元素大于max
        {max=a[i][j]; //max将取该元素的值
        row=i; //记下该元素的行号i
        column=j;//记下该元素的列号j
        }
    cout<<" max=" <<max<<" ,row=" <<row<<" ,column=" <<column<<endl;
  return 0;
}
```

输出结果为

max=56,row=0,column=3

## 第5章 数组

### 5.4 用数组名作函数参数

常量和变量可以用作函数实参,同样数组元素也可以作函数实参,其用法与变量相同。数组名也可以作实参和形参,传递的是数组的起始地址。

#### 1. 用数组元素作函数实参

由于实参可以是表达式,而数组元素可以是表达式的组成部分,因此数组元素当然可以作为函数的实参,与用变量作实参一样,将数组元素的值传送给形参变量。

## 第5章 数组

例 5.6 用函数处理例 5.5。

算法和例 5.5 是一样的,今设一函数 max\_value,用来进行比较并返回结果。可编写程序如下:

```
#include <iostream>
using namespace std;
int main( )
{ int max_value(int x,int max); //函数声明
  int i,j,row=0,column=0,max;
  int a[3][4]={{5,12,23,56},{19,28,37,46},{-12,-34,6,8}}; //数组初始化
  max=a[0][0];
  for (i=0;i<=2;i++)
    for (j=0;j<=3;j++)
      { max=max_value(a[i][j],max); //调用max_value函数
      if(max==a[i][j])//如果函数返回的是a[i][j]的值
        {row=i; //记下该元素行号i
        column=j; //记下该元素列号j
        }
      }
}
```

```

cout<<" max=" <<max<<" ,row=" <<row<<" ,column=" <<column<<endl;
}
int max_value(int x,int max)//定义 max_value 函数
{if(x>max) return x;//如果 x>max,函数返回值为x
else return max;//如果 x≤max,函数返回值为max
}

```

## 第 5 章 数组

### 2. 用数组名作函数参数

可以用数组名作函数参数,此时实参与形参都用数组名(也可以用指针变量,见第 6 章)。

例 5.7 用选择法对数组中 10 个整数按由小到大排序。

所谓选择法就是先将 10 个数中最小的数与 a[0]对换;再将 a[1]到 a[9]中最小的数与 a[1]对换.....每比较一轮,找出一个未经排序的数中最小的一个。共比较 9 轮。

根据此思路编写程序如下:

```

#include <iostream>
using namespace std;
int main()
{void select_sort(int array[],int n); //函数声明
  int a[10],i;
  cout<<" enter the originl array:" <<endl;
  for(i=0;i<10;i++) //输入 10 个数
    cin>>a[i];
  cout<<endl;
  select_sort(a,10); //函数调用,数组名作实参
  cout<<" the sorted array:" <<endl;
  for(i=0;i<10;i++) //输出 10 个已排好序的数
    cout<<a[i]<<" ";
  cout<<endl;
  return 0;
}
void select_sort(int array[],int n) //形参 array 是数组名
{int i,j,k,t;
  for(i=0;i<n-1;i++)
  {k=i;
   for(j=i+1;j<n;j++)
   if(array[j]<array[k]) k=j;
   t=array[k];array[k]=array[i];array[i]=t;
  }
}

```

## 第 5 章 数组

运行情况如下:

enter the originl array:

6 9 -2 56 87 11 -54 3 0 77↵ //输入 10 个数

the sorted array:

-54 -2 0 3 6 9 11 56 77 87

关于用数组名作函数参数有两点要说明:

(1) 如果函数实参是数组名,形参也应为数组名(或指针变量,关于指针见第 5 章),形参不能声明为普通变量

(如 `int array;`)。实参数组与形参数组类型应一致(现都为 `int` 型),如不一致,结果将出错。

(2) 需要特别说明的是: 数组名代表数组首元素的地址,并不代表数组中的全部元素。因此用数组名作函数实参时,不是把实参数组的值传递给形参,而只是将实参数组首元素的地址传递给形参。

### 第 5 章 数组

形参可以是数组名,也可以是指针变量,它们用来接收实参传来的地址。如果形参是数组名,它代表的是形参数组首元素的地址。在调用函数时,将实参数组首元素的地址传递给形参数组名。这样,实参数组和形参数组就共占同一段内存单元。见图 5.6。

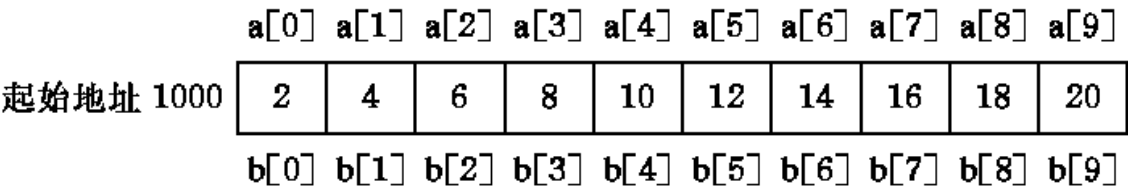


图 5.6

### 第 5 章 数组

在用变量作函数参数时,只能将实参变量的值传给形参变量,在调用函数过程中如果改变了形参的值,对实参没有影响,即实参的值不因形参的值改变而改变。而用数组名作函数实参时,改变形参数组元素的值将同时改变实参数组元素的值。在程序设计中往往有意识地利用这一特点改变实参数组元素的值。

实际上,声明形参数组并不意味着真正建立一个包含若干元素的数组,在调用函数时也不对它分配存储单元,只是用 `array[]` 这样的形式表示 `array` 是一维数组名,以接收实参传来的地址。因此 `array[]` 中方括号内的数值并无实际作用,编译系统对一维数组方括号内的内容不予处理。形参一维数组的声明中可以写元素个数,也可以不写。

### 第 5 章 数组

函数首部的下面几种写法都合法,作用相同。

`void select_sort(int array[10],int n)` //指定元素个数与实参数组相同

`void select_sort(int array[],int n)`//不指定元素个数

`void select_sort(int array[5],int n)` //指定元素个数与实参数组不同

在学习第 6 章时可以进一步知道,C++实际上只把形参数组名作为一个指针变量来处理,用来接收从实参传过来的地址。前面提到的一些现象都是由此而产生的。

### 第 5 章 数组

#### 3. 用多维数组名作函数参数

如果用二维数组名作为实参和形参,在对形参数组声明时,必须指定第二维(即列)的大小,且应与实参的第二维的大小相同。第一维的大小可以指定,也可以不指定。如

`int array[3][10];` //形参数组的两个维都指定

或 `int array[][10];`//第一维大小省略

二者都合法而且等价。但是不能把第二维的大小省略。下面的形参数组写法不合法:

`int array[][];` //不能确定数组的每一行有多少列元素

`int array[3][];`//不指定列数就无法确定数组的结构

在第二维大小相同的前提下,形参数组的第一维可以与实参数组不同。例如,实参数组定义为

`int score[5][10];`

### 第 5 章 数组

而形参数组可以声明为

`int array[3][10];` //列数与实参数组相同,行数不同

`int array[8][10];`

这时形参二维数组与实参二维数组都是由相同类型和大小的一维数组组成的,实参数组名 `score` 代表其首元素(即第一行)的起始地址,系统不检查第一维的大小。

如果是三维或更多维的数组,处理方法是类似的。

## 第 5 章 数组

例 5.8 有一个 3×4 的矩阵,求矩阵中所有元素中的最大值。要求用函数处理。

解此题的算法已在例 5.5 中介绍。

程序如下:

```
#include <iostream>
using namespace std;
int main( )
{int max_value(int array[][4]);
  int a[3][4]={ {11,32,45,67},{22,44,66,88},{15,72,43,37}};
  cout<<" max value is " <<max_value(a)<<endl;
  return 0;
}
```

```
int max_value(int array[][4])
```

```
{int i,j,max;
```

## 第 5 章 数组

```
max=array[0][0];
```

```
for( i=0;i<3;i++)
```

```
for(j=0;j<4;j++)
```

```
if(array[i][j]>max) max=array[i][j];
```

```
return max;
```

```
}
```

运行结果如下:

max value is 88

读者可以将 max\_value 函数的首部改为以下几种情况,观察编译情况:

```
int max_value(int array[][])
```

```
int max_value(int array[3][])
```

```
int max_value(int array[3][4])
```

```
int max_value(int array[10][10])
```

```
int max_value(int array[12])
```

## 第 5 章 数组

### 5.5 字符数组

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++专门它为它提供了许多方便的用法和函数。

## 第 5 章 数组

### 5.5.1 字符数组的定义和初始化

定义字符数组的方法与前面介绍的类似。例如

```
char c[10];
```

```
c[0]=' I' ;c[1]=' ' ;c[2]=' a' ;c[3]=' m' ;c[4]=' ' ;c[5]=' h' ;c[6]=' a' ;c[7]=' p' ;c[8]=' p' ;
c[9]=' y' ;
```

上面定义了 c 为字符数组,包含 10 个元素。在赋值以后数组的状态如图 5.7 所示。

c[0] c[1] c[2] c[3] c[4] c[5] c[6] c[7] c[8] c[9]

|   |  |   |   |  |   |   |   |   |   |
|---|--|---|---|--|---|---|---|---|---|
| I |  | a | m |  | h | a | p | p | y |
|---|--|---|---|--|---|---|---|---|---|

图 5.7

## 第 5 章 数组

对字符数组进行初始化,最容易理解的方式是逐个字符赋给数组中各元素。如

```
char c[10]={ ' I' , ' ' , ' a' , ' m' , ' ' , ' h' , ' a' , ' p' , ' p' , ' y' };
```

把 10 个字符分别赋给 c[0]~c[9]这 10 个元素。

如果花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,其余的元素自动定为空字符。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。如

```
char c[]={ ' I' , ' ' , ' a' , ' m' , ' ' , ' h' , ' a' , ' p' , ' p' , ' y' };
```

也可以定义和初始化一个二维字符数组,如

```
char diamond[5][5]={{' ' , ' ' , ' *' },{' ' , ' *' , ' ' , ' *' },{' *' , ' ' , ' ' , ' ' , ' ' , ' *' },{' ' , ' *' , ' ' , ' ' , ' *' },{' ' , ' ' , ' ' , ' *' }};
```

## 第 5 章 数组

### 5.5.2 字符数组的赋值与引用

只能对字符数组的元素赋值,而不能用赋值语句对整个数组赋值。如

```
char c[5];
```

```
c={' C' , ' h' , ' i' , ' n' , ' a' }; //错误,不能对整个数组一次赋值
```

```
c[0]=' C' ; c[1]=' h' ; c[2]=' i' ; c[3]=' n' ; c[4]=' a' ; //对数组元素赋值,正确
```

如果已定义了 a 和 b 是具有相同类型和长度的数组,且 b 数组已被初始化,请分析:

```
a=b; //错误,不能对整个数组整体赋值
```

```
a[0]=b[0]; //正确,引用数组元素
```

## 第 5 章 数组

例 5.9 设计和输出一个钻石图形。

```
#include <iostream>
```

```
using namespace std;
```

```
void main()
```

```
{char diamond[][5]={{' ' , ' ' , ' *' },{' ' , ' *' , ' ' , ' *' },{' *' , ' ' , ' ' , ' ' , ' *' },{' ' , ' *' , ' ' , ' ' , ' *' },{' ' , ' ' , ' ' , ' *' }};
```

```
int i,j;
```

```
for (i=0;i<5;i++)
```

```
{for (j=0;j<5;j++)
```

```
cout<<diamond[i][j]; //逐个引用数组元素,每次输出一个字符
```

```
cout<<endl;
```

```
}
```

```
}
```

## 第 5 章 数组

运行结果为

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

## 第 5 章 数组

### 5.5.3 字符串和字符串结束标志

用一个字符数组可以存放一个字符串中的字符。如

```
char str[12]={ ' I' , ' ' , ' a' , ' m' , ' ' , ' h' , ' a' , ' p' , ' p' , ' y' };
```

用一维字符数组 str 来存放一个字符串 " I am happy " 中的字符。字符串的实际长度(10)与数组长度(12)不相

等,在存放上面 10 个字符之外,系统对字符数组最后两元素自动填补空字符' \0' 。  
为了测定字符串的实际长度,C++规定了一个“字符串结束标志”,以字符' \0' 代表。在上面的数组中,第 11 个字符为' \0' ,就表明字符串的有效字符为其前面的 10 个字符。也就是说,遇到字符' \0' 就表示字符串到此结束,由它前面的字符组成字符串。

第 5 章 数组

对一个字符串常量,系统会自动在所有字符的后面加一个' \0' 作为结束符。例如字符串" I am happy" 共有 10 个字符,但在内存中它共占 11 个字节,最后一个字节' \0' 是由系统自动加上的。  
在程序中往往依靠检测' \0' 的位置来判定字符串是否结束,而不是根据数组的长度来决定字符串长度。当然,在定义字符数组时应估计实际字符串长度,保证数组长度始终大于字符串实际长度。如果在一个字符数组中先后存放多个不同长度的字符串,则应使数组长度大于最长的字符串的长度。  
说明: ' \0' 只是一个供辨别的标志。  
如果用以下语句输出一个字符串:

```
cout<<"  
How do you do?" ;
```

第 5 章 数组

系统在执行此语句时逐个地输出字符,那么它怎么判断应该输出到哪个字符就停止了呢?  
下面再对字符数组初始化补充一种方法: 用字符串常量来初始化字符数组。例如

```
char str[]{" I am happy" };  
也可以省略花括号,直接写成
```

```
char str[]=" I am happy" ;
```

不是用单个字符作为初值,而是用一个字符串(注意字符串的两端是用双撇号而不是单撇号括起来的)作为初值。显然,这种方法直观,方便,符合人们的习惯。注意: 数组 str 的长度不是 10,而是 11(因为字符串常量的最后由系统加上一个' \0' )。因此,上面的初始化与下面的初始化等价:

```
char str[]={ ' I' , ' ' , ' a' , ' m' , ' ' , ' h' , ' a' , ' p' , ' p' , ' y' , ' \0' };
```

第 5 章 数组

而不与下面的等价:  

```
char str[]={ ' I' , ' ' , ' a' , ' m' , ' ' , ' h' , ' a' , ' p' , ' p' , ' y' };
```

前者的长度为 11,后者的长度为 10。如果有

```
char str[10]=" China" ;
```

数组 str 的前 5 个元素为' C' , ' h' , ' i' , ' n' , ' a' ,第 6 个元素为' \0' ,后 4 个元素为空字符。见图 5.8。

图 5.8

|   |   |   |   |   |    |    |    |    |    |
|---|---|---|---|---|----|----|----|----|----|
| C | h | i | n | a | \0 | \0 | \0 | \0 | \0 |
|---|---|---|---|---|----|----|----|----|----|

第 5 章 数组

需要说明的是:字符数组并不要求它的最后一个字符为' \0' ,甚至可以不包含' \0' 。如以下这样写完全是合法的:

```
char str[5]={ ' C' , ' h' , ' i' , ' n' , ' a' };
```

是否需要加' \0' ,完全根据需要决定。但是由于 C++编译系统对字符串常量自动加一个' \0' 。因此,人们为了使处理方法一致,便于测定字符串的实际长度,以及在程序中作相应的处理,在字符数组中有效字符的后面也人为地加上一个' \0' 。如

```
char str[6]={ ' C' , ' h' , ' i' , ' n' , ' a' , ' \0' };
```

第 5 章 数组

5.5.4 字符数组的输入输出



字符数组的输入输出可以有两种方法:

(1) 逐个字符输入输出,如例 5.9。

(2) 将整个字符串一次输入或输出。例如有以下程序段:

```
char str[20];
```

```
cin>>str; //用字符数组名输入字符串
```

```
cout<<str; //用字符数组名输出字符串
```

在运行时输入一个字符串,如

China✓

在内存中,数组 str 的状态如图 5.9 所示,在 5 个字符的后面自动加了一个结束符 ' \0' 。

## 第 5 章 数组

图 5.9

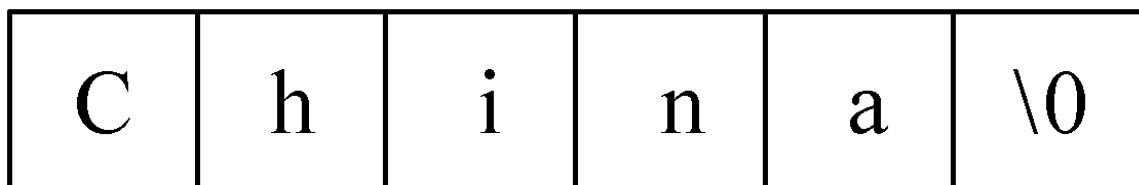
输出时,逐个输出字符直到遇结束符 ' \0' ,就停止输出。输出结果为

China

如前所述,字符数组名 str 代表字符数组第一个元素的地址,执行“cout<<str;”的过程是从 str 所指向的数组第一个元素开始逐个输出字符,直到遇到 ' \0' 为止。

请注意:

(1) 输出的字符不包括结束符 ' \0' 。



## 第 5 章 数组

(2) 输出字符串时,cout 流中用字符数组名,而不是数组元素名。

(3) 如果数组长度大于字符串实际长度,也只输出到遇 ' \0' 结束。

(4) 如果一个字符数组中包含一个以上 ' \0' ,则遇第一个 ' \0' 时输出就结束。

(5) 用 cin 从键盘向计算机输入一个字符串时,从键盘输入的字符串应短于已定义的字符数组的长度,否则会出现问题。

C++提供了 cin 流中的 getline 函数,用于读入一行字符(或一行字符中前若干个字符),使用安全又方便,请参阅第 13 章 13.3.2 节。

## 第 5 章 数组

### 5.5.5 字符串处理函数

由于字符串使用广泛,C 和 C++提供了一些字符串函数,使得用户能很方便地对字符串进行处理。几乎所有版本的 C++都提供下面这些函数,它们是放在函数库中的,在 string 和 string.h 头文件中定义。如果程序中使用这些字符串函数,应该用#include 命令把 string.h 或 string 头文件包含到本文件中。下面介绍几种常用的函数。

## 第 5 章 数组

### 1. 字符串连接函数 strcat

其函数原型为

```
strcat(char[],const char[]);
```

strcat 是 string concatenate(字符串连接)的缩写。该函数有两个字符数组的参数,函数的作用是:将第二个字符数组中的字符串连接到前面字符数组的字符串的后面。第二个字符数组被指定为 const,以保证该数组中的内容不会在函数调用期间修改。连接后的字符串放在第一个字符数组中,函数调用后得到的函数值,就是第一个字符数组的地址。例如

```
char str1[30]=" People' s Republic of " ;
```

```
char str2[]=" China" ;
```

```
cout<<strcat(str1,str2); //调用 strcat 函数
```

第 5 章 数组

输出:

People' s Republic of China

连接前后的状况如图 5.10 所示。

|      |   |   |   |   |   |    |   |   |  |   |   |   |   |   |   |   |   |  |   |   |  |    |    |    |    |    |    |
|------|---|---|---|---|---|----|---|---|--|---|---|---|---|---|---|---|---|--|---|---|--|----|----|----|----|----|----|
| tr1: | P | e | o | p | l | e  | ' | s |  | R | e | p | u | b | l | i | c |  | o | f |  | \0 | \0 | \0 | \0 | \0 | \0 |
| tr2: | C | h | i | n | a | \0 |   |   |  |   |   |   |   |   |   |   |   |  |   |   |  |    |    |    |    |    |    |
| tr3: | P | e | o | p | l | e  | ' | s |  | R | e | p | u | b | l | i | c |  | o | f |  | C  | h  | i  | n  | a  | \0 |

图 5.10

第 5 章 数组

2. 字符串复制函数 strcpy

其函数原型为

strcpy(char[],const char[]);

strcpy 是 string copy(字符串复制)的缩写。它的作用是将第二个字符数组中的字符串复制到第一个字符数组中去,将第一个字符数组中的相应字符覆盖。例如

char str1[10],str2[]=" China" ;

strcpy(str1,str2);

执行后,str2 中的 5 个字符" China" 和' \0' (共 6 个字符)复制到数组 str1 中。

第 5 章 数组

说明:

- (1) 在调用 strcpy 函数时,第一个参数必须是数组名(如 str1),第二个参数可以是字符数组名,也可以是一个字符串常量。
- (2) 可以用 strcpy 函数将一个字符串中前若干个字符复制到字符数组中去。
- (3) 只能通过调用 strcpy 函数来实现将一个字符串赋给一个字符数组,而不能用赋值语句将一个字符串常量或字符数组直接赋给一个字符数组。

第 5 章 数组

3. 字符串比较函数 strcmp

其函数原型为

strcmp(const char[],const char[]);

strcmp 是 string compare(字符串比较)的缩写。作用是比较两个字符串。由于这两个字符数组只参加比较而不应改变其内容,因此两个参数都加上 const 声明。以下写法是合法的:

strcmp(str1,str2);

strcmp(" China" ," Korea" );

strcmp(str1," Beijing" );

比较的结果由函数值带回。

- (1) 如果字符串 1=字符串 2,函数值为 0。
- (2) 如果字符串 1>字符串 2,函数值为一正整数。

第 5 章 数组

- (3) 如果字符串 1<字符串 2,函数值为一负整数。

字符串比较的规则与其他语言中的规则相同,即对两个字符串自左至右逐个字符相比(按 ASCII 码值大小比较),直到出现不同的字符或遇到' \0' 为止。如全部字符相同,则认为相等;若出现不相同的字符,则以第一个

不相同的字符的比较结果为准。

注意:对两个字符串比较,不能用以下形式:

```
if(str1>str2) cout<<" yes" ;
```

字符数组名 `str1` 和 `str2` 代表数组地址,上面写法表示将两个数组地址进行比较,而不是对数组中的字符串进行比较。对两个字符串比较应该用

```
if(strcmp(str1,str2)>0) cout<<" yes" ;
```

## 第 5 章 数组

### 4. 字符串长度函数 `strlen`

函数原型为

```
strlen(const char[]);
```

`strlen` 是 `string length`(字符串长度)的缩写。它是测试字符串长度的函数。其函数的值为字符串中的实际长度,不包括 `'\0'` 在内。如

```
char str[10]=" China" ;
```

```
cout<<strlen(str);
```

输出结果不是 10,也不是 6,而是 5。

以上是几种常用的字符串处理函数,除此之外还有其他一些函数。

## 第 5 章 数组

### 5.5.6 字符数组应用举例

例 5.10 有 3 个字符串,要求找出其中最大者。要求用函数调用。

程序如下:

```
#include <iostream>
```

```
#include <string>
```

```
using namespace std;
```

```
int main()
```

```
{ void max_string(char str[][30],int i); //函数声明
```

```
int i;
```

```
char country_name[3][30];
```

```
for(i=0;i<3;i++)
```

```
cin>>country_name[i]; //输入 3 个国家名
```

```
max_string(country_name,3); //调用 max_string 函数
```

```
return 0;
```

## 第 5 章 数组

```
}
```

```
void max_string(char str[][30],int n)
```

```
{
```

```
int i;
```

```
char string[30];
```

```
strcpy(string,str[0]); //使 string 的值为 str[0] 的值
```

```
for(i=0;i<n;i++)
```

```
if(strcmp(str[i],string)>0) //如果 str[i]>string
```

```
strcpy(string,str[i]); //将 str[i] 中的字符串复制到 string
```

```
cout<<endl<<" the largest string is: " <<string<<endl;
```

```
}
```

运行结果如下:

CHINA✓

GERMANY✓

FRANCH✓

the largest string is: GERMANY

## 第5章 数组

### \*5.6 C++处理字符串的方法——字符串类与字符串变量

用字符数组来存放字符串并不是最理想和最安全的方法。

C++提供了一种新的数据类型——字符串类型(string 类型),在使用方法上,它和 char、int 类型一样,可以用来定义变量,这就是字符串变量——用一个名字代表一个字符序列。

实际上,string 并不是 C++语言本身具有的基本类型,它是在 C++标准库中声明的一个字符串类,用这种类可以定义对象。每一个字符串变量都是 string 类的一个对象。

## 第5章 数组

### 5.6.1 字符串变量的定义和引用

#### 1. 定义字符串变量

和其他类型变量一样,字符串变量必须先定义后使用,定义字符串变量要用类名 string。如

```
string string1; //定义 string1 为字符串变量
```

```
string string2=" China" ;//定义 string2 同时对其初始化
```

应当注意: 要使用 string 类的功能时,必须在本文件的开头将 C++标准库中的 string 头文件包含进来,即应加上

```
#include <string> //注意头文件名不是 string.h
```

## 第5章 数组

#### 2. 对字符串变量的赋值

在定义了字符串变量后,可以用赋值语句对它赋予一个字符串常量,如

```
string1=" Canada" ;
```

既可以用字符串常量给字符串变量赋值,也可以用一个字符串变量给另一个字符串变量赋值。如

```
string2=string1; //假设 string2 和 string1 均已定义为字符串变量
```

不要求 string2 和 string1 长度相同,假如 string2 原来是" China",string1 原来是" Canada",赋值后 string2 也变成" Canada"。在定义字符串变量时不需指定长度,长度随其中的字符串长度而改变。

可以对字符串变量中某一字符进行操作,如

```
string word=" Then" ; //定义并初始化字符串变量 word
```

```
word[2]=' a' ; //修改序号为2 的字符,修改后 word 的值为" Than "
```

## 第5章 数组

#### 3. 字符串变量的输入输出

可以在输入输出语句中用字符串变量名,输入输出字符串,如

```
cin>> string1; //从键盘输入一个字符串给字符串变量 string1
```

```
cout<< string2; //将字符串 string2 输出
```

## 第5章 数组

### 5.6.2 字符串变量的运算

在上一节中可以看到: 在以字符数组存放字符串时,字符串的运算要用字符串函数,如 strcat(连接)、strcmp(比较)、strcpy(复制),而对 string 类对象,可以不用这些函数,而直接用简单的运算符。

#### (1) 字符串复制用赋值号

```
string1=string2;
```

其作用与"strcpy(string1,string2);"相同。

#### (2) 字符串连接用加号

```
string string1=" C++" ;//定义 string1 并赋初值
```

```
string string2=" Language" ;//定义 string2 并赋初值
```

```
string1=string1 + string2;//连接 string1 和 string2
```

连接后 string1 为" C++ Language"。

## 第5章 数组

#### (3) 字符串比较直接用关系运算符

可以直接用 ==(等于)、>(大于)、<(小于)、!=(不等于)、>=(大于或等于)、<=(小于或等于)等关系运算符来进行字符串的比较。

使用这些运算符比使用 5.5.5 节中介绍的字符串函数直观而方便。

第 5 章 数组

5.6.3 字符串数组

不仅可以用 string 定义字符串变量,也可以用 string 定义字符串数组。如

string name[5]; //定义一个字符串数组,它包含 5 个字符串元素

string name[5]={ " Zhang" ," Li" ," Fun" ," Wang" ," Tan" };

//定义一个字符串数组并初始化

此时 name 数组的状况如图 5.11 所示。

|         |   |   |   |   |   |
|---------|---|---|---|---|---|
| name[0] | Z | h | a | n | g |
| name[1] | L | i |   |   |   |
| name[2] | F | u | n |   |   |
| name[3] | W | a | n | g |   |
| name[4] | T | a | n |   |   |

图 5.11

第 5 章 数组

可以看到:

- (1) 在一个字符串数组中包含若干个(现为 5 个)元素,每个元素相当于一个字符串变量。
- (2) 并不要求每个字符串元素具有相同的长度,即使对同一个元素而言,它的长度也是可以变化的,当向某一个元素重新赋值,其长度就可能发生变化。
- (3) 在字符串数组的每一个元素中存放一个字符串,而不是一个字符,这是字符串数组与字符数组的区别。如果用字符数组存放字符串,一个元素只能存放一个字符,用一个一维字符数组存放一个字符串。
- (4) 每一个字符串元素中只包含字符串本身的字符而不包括' \0' 。

第 5 章 数组

可见用字符串数组存放字符串以及对字符串进行处理是很方便的。

在定义字符串数组时怎样给数组分配存储空间呢?实际上,编译系统为每一个字符串变量分配 4 个字节,在这个存储单元中,并不是直接存放字符串本身,而是存放字符串的地址。在本例中,就是把字符串" Zhang" 的地址存放在 name[0],把字符串" Li" 的地址存放在 name[1],把字符串" Fun" 的地址存放在 name[2].....图 5.11 只是一个示意图。在字符串变量中存放的是字符串的指针(字符串的地址)。

第 5 章 数组

5.6.4 字符串运算举例

例 5.11 输入 3 个字符串,要求将字母按由小到大的顺序输出。

#include <iostream>

#include <string>

using namespace std;

int main( )

```
{string string1,string2,string3,temp;
cout<<" please input three strings:" ;//这是对用户输入的提示
cin>>string1>>string2>>string3; //输入 3 个字符串
if(string2>string3) {temp=string2;string2=string3;string3=temp;}
//使串 2 ≤ 串 3
if(string1≤string2) cout<<string1<<" " <<string2<<" " <<string3<<endl;
//如果串 1 ≤ 串 2,则串 1 ≤ 串 2 ≤ 串 3
```

## 第 5 章 数组

```
else if(string1≤string3) cout<<string2<<" " <<string1<<" " <<string3<<endl;
//如果串 1 > 串 2,且串 1 ≤ 串 3,则串 2 < 串 1 ≤ 串 3
else cout<<string2<<" " <<string3<<" " <<string1<<endl;
//如果串 1 > 串 2,且串 1 > 串 3,则串 2 < 串 3 < 串 1
}
```

运行情况如下:

```
please input three strings: China U.S.A. Germany↵
China Germany U.S.A.
```

## 第 5 章 数组

例 5.12 一个班有  $n$  个学生,需要把每个学生的简单材料(姓名和学号)输入计算机保存。然后通过输入某一学生的姓名查找其有关资料。当输入一个姓名后,程序就查找该班中有无此学生,如果有,则输出他的姓名和学号,如果查不到,则输出“本班无此人”。

为解此问题,可以分别编写两个函数,函数 `input_data` 用来输入  $n$  个学生的姓名和学号,函数 `search` 用来查找要找的学生是否在本班。

程序可编写如下:

```
#include <iostream>
#include <string>
using namespace std;
string name[50],num[50]; //定义两个字符串数组,分别存放姓名和学号
int n; //n 是实际的学生数
第 5 章 数组
int main()
{void input_data(); //函数声明
void search(string find_name); //函数声明
string find_name;//定义字符串变量,find_name 是要找的学生
cout<<" please input number of this class:" ; //输入提示: 请输入本班学生的人数
cin>>n; //输入学生数
input_data();//调用 input_data 函数,输入学生数据
cout<<" please input name you want find:" ; //输入提示: 请输入你要找的学生名
cin>>find_name; //输入要找的学生的姓名
search(find_name); //调用 search 函数,寻找该学生姓名
return 0;
}
void input_data() //函数首部
{int i;
for (i=0;i<n;i++)
{cout<<" input name and NO. of student " <<i+1<<" : " ;
//输入提示
```

```

    cin>>name[i]>>num[i];} //输入 n 个学生的姓名和学号
}
第 5 章 数组
void search(string find_name) //函数首部
{int i;
  bool flag=false;
  for(i=0;i<n;i++)
  if(name[i]==find_name)//如果要找的姓名与本班某一学生姓名相同
    { cout<<name[i]<<" has been found, his number is " <<num[i]<<endl;
    //输出姓名与学号
    flag=true;
    break;
  }
  if(flag==false) cout<<" can' t find this name" ;//如找不到,输出“找不到”的信息
}

```

运行情况如下:

```

please input number of this class:5✓
input name and number of student 1:Li 1001✓
input name and number of student 2:Zhang 1002✓
input name and number of student 3:Wang 1003✓
input name and number of student 4:Tan 1004✓
input name and number of student 5:Fun 1005✓
please input name you want find:Wang✓
Wang has been found,his number is 1003

```

## 第 5 章 数组

请考虑:

- (1) 程序第 3 行定义全局变量时,数组的大小不指定为 50,而用变量 n,即 string name[n],num[n];n 在运行时输入,行不行?为什么?
- (2) search 函数 for 循环中最后有一个 break 语句,它起什么作用?不要行不行?
- (3) 如果不使用全局变量,把变量 n 和数组 name,num 都作为局部变量,通过虚实结合的方法在函数间传递数据,这样行不行?请思考并上机试一下。

## 第 5 章 数组

通过以上两个例子可以看到,用 string 定义字符串变量,简化了操作,把原来复杂的问题简单化了,这是 C++对 C 的一个发展。

归纳起来,C++对字符串的处理有两种方法:一种是用字符数组的方法,这是 C 语言采取的方法,一般称为 C string 方法;一种是用 string 类定义字符串变量,称为 string 方法。显然,string 方法概念清楚,使用方便,最好采用这种方法。C++保留 C-string 方法主要是为了与 C 兼容,使以前用 C 写的程序能用于 C++环境。

# 第 6 章 指针

## 6.1 指针的概念



- 6.2 变量与指针
- 6.3 数组与指针
- 6.4 字符串与指针
- 6.5 函数与指针
- 6.6 返回指针值的函数
- 6.7 指针数组和指向指针的指针
- 6.8 有关指针的数据类型和指针运算的小结
- \*6.9 引用

第 6 章 指针

6.1 指针的概念

为了说清楚什么是指针,必须弄清楚数据在内存中是如何存储的,又是如何读取的。  
如果在程序中定义了一个变量,在编译时就给这个变量分配内存单元。系统根据程序中定义的变量类型,分配一定长度的空间。例如,C++编译系统一般为整型变量分配 4 个字节,为单精度浮点型变量分配 4 个字节,为字符型变量分配 1 个字节。内存区的每一个字节有一个编号,这图 6.1 就是“地址”。

第 6 章 指针

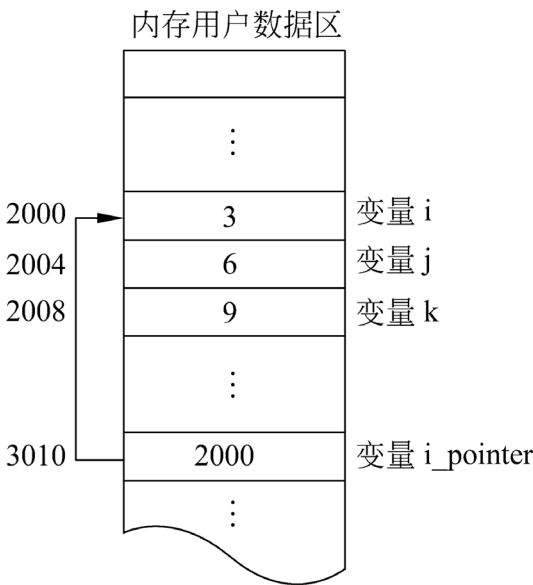


图 6.1

第 6 章 指针

请务必弄清楚一个内存单元的地址与内存单元的内容这两个概念的区别。在程序中一般是通过变量名来对内存单元进行存取操作的。其实程序经过编译以后已经将变量名转换为变量的地址,对变量值的存取都是通过地址进行的。这种按变量地址存取变量值的方式称为直接存取方式,或直接访问方式。  
还可以采用另一种称为间接存取(间接访问)的方式。可以在程序中定义这样一种特殊的变量,它是专门用来存放地址的。

图 6.2 是直接访问和间接访问的示意图。为了将数值 3 送到变量中,可以有两种方法:

- (1) 直接将数 3 送到整型变量 i 所标识的单元中。见图 6.2(a)。

第 6 章 指针

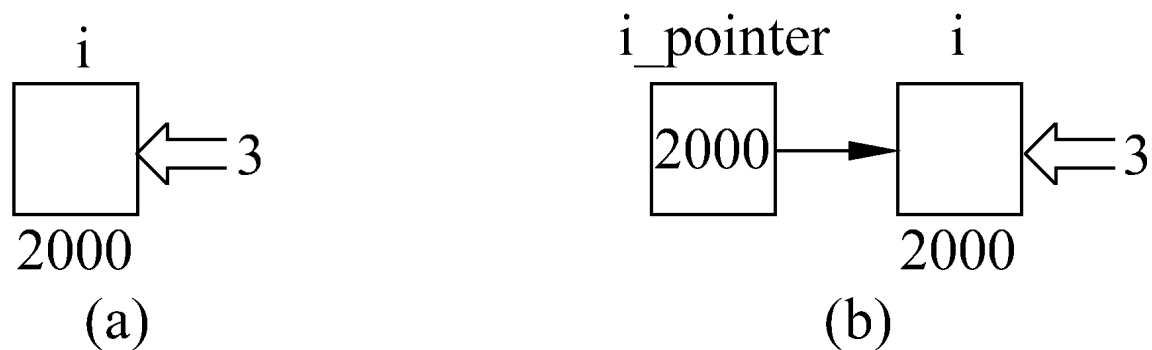
- (2) 将 3 送到指针变量 i\_pointer 所指向的单元(这就是变量 i 所标识的单元)中。见图 6.2(b)。

图 6.2

所谓指向,就是通过地址来体现的。



由于通过地址能找到所需的变量单元,因此可以说,地址指向该变量单元。因此将地址形象化地称为“指针”。一个变量的地址称为该变量的指针。  
如果有一个变量是专门用来存放另一变量地址(即指针)的,则它称为指针变量。指针变量的值(即指针变量中存放的值)是地址(即指针)。



第 6 章 指针

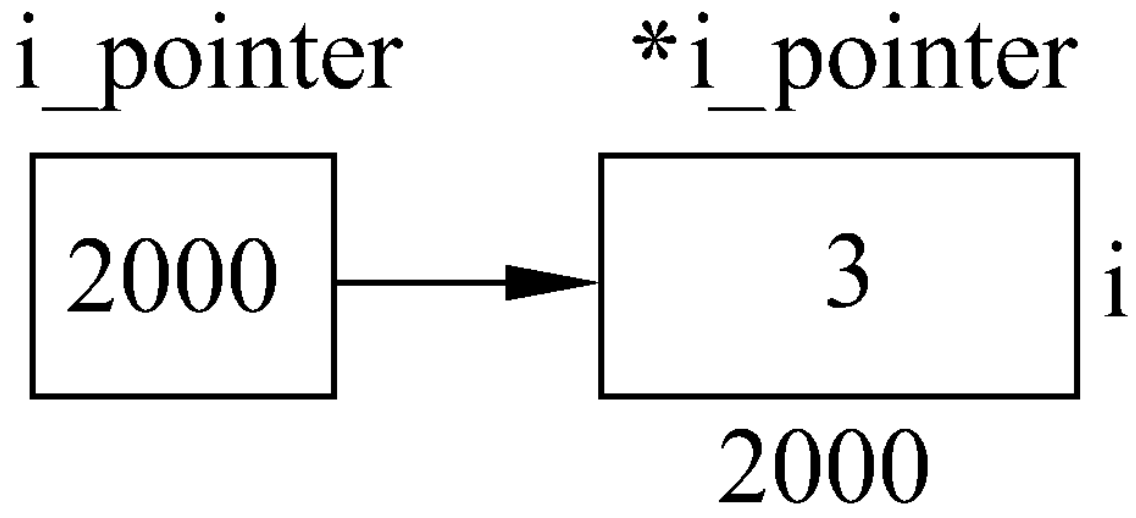
6.2 变量与指针

指针变量是一种特殊的变量,它和以前学过的其他类型的变量的不同之处是: 用它来指向另一个变量。为了表示指针变量和它所指向的变量之间的联系,在 C++中用“\*”符号表示指向,例如,`i_pointer` 是一个指针变量,而 `*i_pointer` 表示 `i_pointer` 所指向的变量,见图 6.3。

下面两个语句作用相同:

- ① `i=3;`
- ② `*i_pointer=3;`

图 6.3



第 6 章 指针

6.2.1 定义指针变量

C++规定所有变量在使用前必须先定义,即指定其类型。在编译时按变量类型分配存储空间。对指针变量必须将它定义为指针类型。先看一个具体例子:

```
int i,j; //定义整型变量 i,j
int *pointer_1,*pointer_2; //定义指针变量 *pointer_1,*pointer_2
```

第 2 行开头的 `int` 是指: 所定义的指针变量是指向整型数据的指针变量。也就是说,指针变量 `pointer_1` 和 `pointer_2` 只能用来指向整型数据(例如 `i` 和 `j`),而不能指向浮点型变量 `a` 和 `b`。这个 `int` 就是指针变量的基类型。指针变量的基类型用来指定该指针变量可以指向的变量的类型。

定义指针变量的一般形式为

第 6 章 指针

基类型 \*指针变量名;

下面都是合法的定义:

```
float *pointer_3;           //pointer_3 是指向单精度型数据的指针变量
```

```
char *pointer_4;           //pointer_4 是指向字符型数据的指针变量
```

请注意: 指针变量名是 `pointer_3` 和 `pointer_4`, 而不是 `*pointer_3` 和 `*pointer_4`, 即 “\*” 不是指针变量名的一部分, 在定义变量时在变量名前加一个 “\*” 表示该变量是指针变量。

那么, 怎样使一个指针变量指向另一个变量呢? 只需要把被指向的变量的地址赋给指针变量即可。例如:

```
pointer_1=&i;               //将变量i 的地址存放到指针变量pointer_1 中
```

```
pointer_2=&j;               //将变量j 的地址存放到指针变量pointer_2 中
```

## 第 6 章 指针

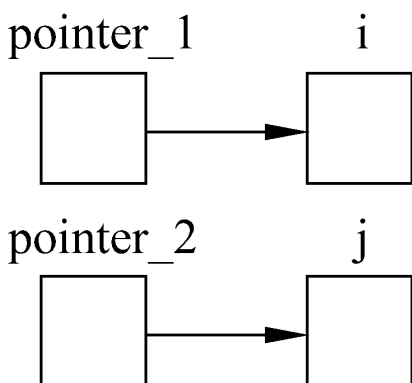
这样, `pointer_1` 就指向了变量 `i`, `pointer_2` 就指向了变量 `j`。见图 6.4。

图 6.4

一般的 C++ 编译系统为每一个指针变量分配 4 个字节的存储单元, 用来存放变量的地址。

在定义指针变量时要注意:

- (1) 不能用一个整数给一个指针变量赋初值。
- (2) 在定义指针变量时必须指定基类型。



## 第 6 章 指针

### 6.2.2 引用指针变量

有两个与指针变量有关的运算符:

- (1) `&` 取地址运算符。
- (2) `*` 指针运算符 (或称间接访问运算符)。

例如: `&a` 为变量 `a` 的地址, `*p` 为指针变量 `p` 所指向的存储单元。

## 第 6 章 指针

例 6.1 通过指针变量访问整型变量。

```
#include <iostream>
```

```
using namespace std;
```

```
int main( )
```

```
{int a,b;
```

```
//定义整型变量 a,b
```

```
int *pointer_1,*pointer_2;
```

```
//定义指针变量*pointer_1,*pointer_2
```

```
a=100;b=10;
```

```
//对 a,b 赋值
```

```
pointer_1=&a;
```

```
//把变量 a 的地址赋给 pointer_1
```

```
pointer_2=&b;
```

```
//把变量 b 的地址赋给 pointer_2
```

```
cout<<a<<" " <<b<<endl;
```

```
//输出 a 和 b 的值
```

```
cout<<*pointer_1<<" " <<*pointer_2<<endl; //输出*pointer_1 和*pointer_2 的值
```

```
return 0;
}
```

第 6 章 指针

运行结果为

100 10

100 10

请对照图 6.5 分析。

(a 和 b 的值)

(\*pointer\_1 和 \*pointer\_2 的值)

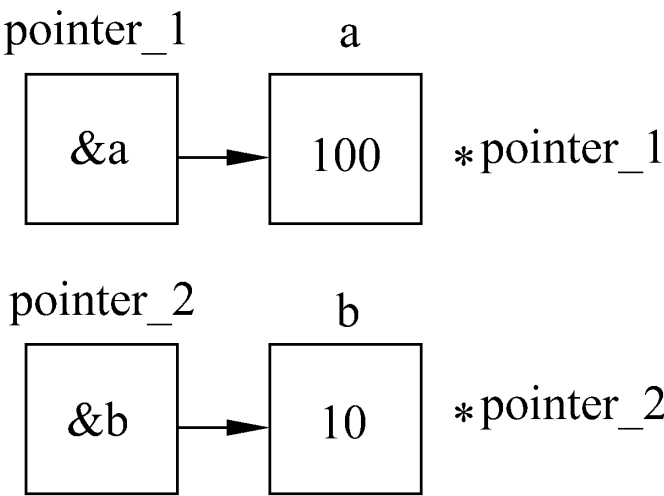


图 6.5

第 6 章 指针

下面对“&”和“\*”运算符再做些说明:

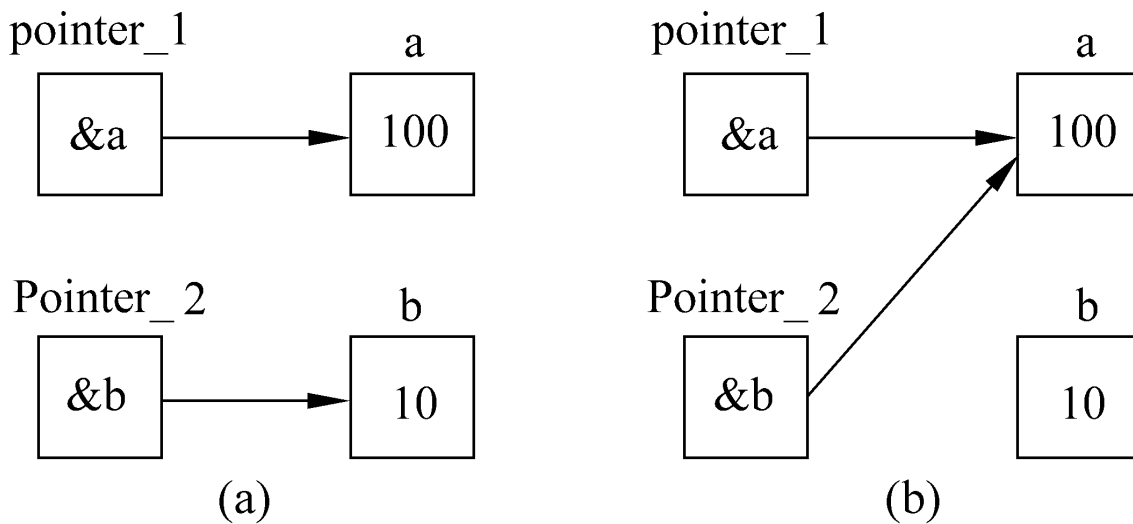
(1) 如果已执行了“pointer\_1=&a;”语句,请问\*&pointer\_1 的含义是什么?“&”和“\*”两个运算符的优先级别相同,但按自右至左方向结合,因此先进行\*&pointer\_1 的运算,它就是变量 a,再执行&运算。因此,&\*&pointer\_1 与&a 相同,即变量 a 的地址。

如果有 pointer\_2=&\*&pointer\_1;它的作用是将&a(a 的地址)赋给 pointer\_2,如果 pointer\_2 原来指向 b,经过重新赋值后它已不再指向 b 了,而也指向了 a,见图 6.6。图 6.6(a)是原来的情况,图 6.6(b)是执行上述赋值语句后的情况。

第 6 章 指针

图 6.6

(2) \*&a 的含义是什么?先进行&a 的运算,得 a 的地址,再进行\*运算,即&a 所指向的变量,\*&a 和\*pointer\_1 的作用是一样的(假设已执行了“pointer\_1=&a;”),它们等价于变量 a。即\*&a 与 a 等价,见图 6.7。

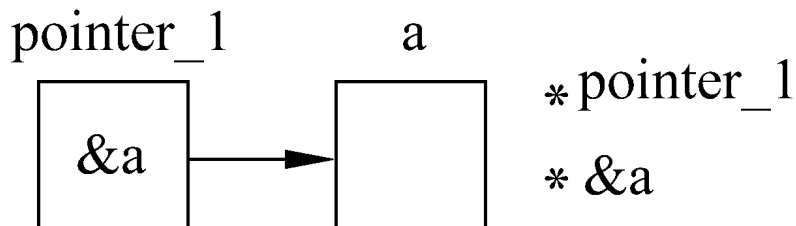


## 第 6 章 指针

图 6.7

例 6.2 输入 `a` 和 `b` 两个整数,按先大后小的顺序输出 `a` 和 `b`(用指针变量处理)。

解此题的思路是: 设两个指针变量 `p1` 和 `p2`,使它们分别指向 `a` 和 `b`。使 `p1` 指向 `a` 和 `b` 中的大者,`p2` 指向小者,顺序输出 `*p1,*p2` 就实现了按先大后小的顺序输出 `a` 和 `b`。按此思路编写程序如下:



## 第 6 章 指针

```
#include <iostream>
using namespace std;
int main( )
{
    int *p1,*p2,*p,a,b;
    cin>>a>>b;                //输入两个整数
    p1=&a;                      //使 p1 指向 a
    p2=&b;                      //使 p2 指向 b
    if(a<b)                    //如果 a<b 就使 p1 与 p2 的值交换
    {p=p1;p1=p2;p2=p;}        //将 p1 的指向与 p2 的指向交换
    cout<<" a=" <<a<<" b=" <<b<<endl;
    cout<<" max=" <<*p1<<" min=" <<*p2<<endl;
    return 0;
}
```

运行情况如下:

## 第 6 章 指针

4578✓

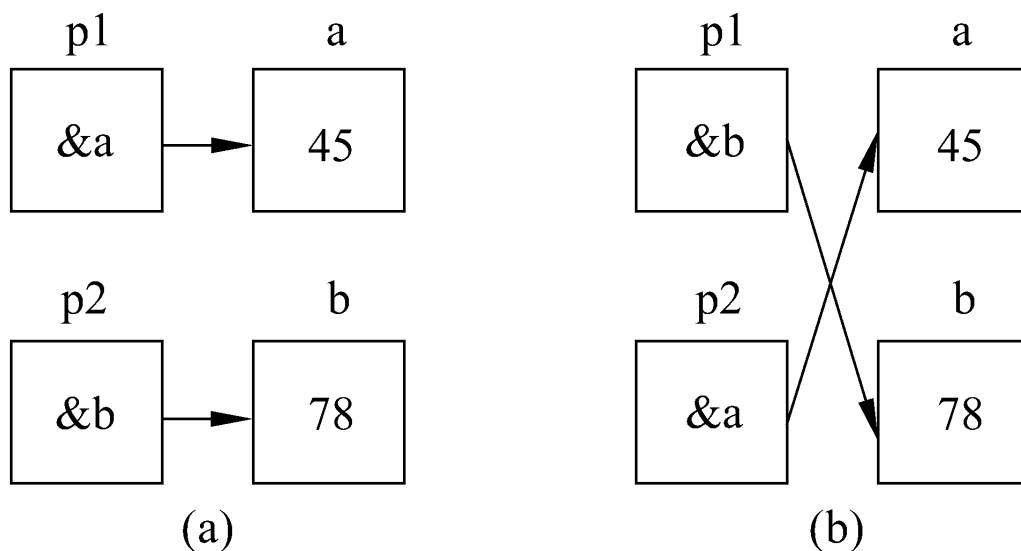
a=45 b=78

max=78 min=45

输入 a 的值 45,b 的值 78,由于  $a < b$ ,将 p1 的值和 p2 的值交换,即将 p1 的指向与 p2 的指向交换。交换前的情况见图 6.8(a),交换后的情况见图 6.8(b)。

图 6.8

请注意,这个问题的算法是不交换整型变量的值,而是交换两个指针变量的值。



## 第 6 章 指针

### 6.2.3 指针作为函数参数

函数的参数不仅可以是整型、浮点型、字符型等数据,还可以是指针类型。它的作用是将一个变量的地址传送给被调用函数的形参。

例 6.3 题目同例 6.2,即对输入的两个整数按大小顺序输出。

这里用函数处理,而且用指针类型的数据作函数参数。

程序如下:

## 第 6 章 指针

```
#include <iostream>
using namespace std;
int main( )
{ void swap(int *p1,int *p2);    //函数声明
  int *pointer_1,*pointer_2,a,b; //定义指针变量pointer_1,pointer_2,整型变量a,b
  cin>>a>>b;
  pointer_1=&a;                  //使pointer_1 指向a
  pointer_2=&b;                  //使pointer_2 指向b
  if(a<b) swap(pointer_1,pointer_2); //如果a<b,使*pointer_1 和*pointer_2 互换
  cout<<" max=" <<a<<" min=" <<b<<endl; //a 已是大数,b 是小数
  return 0;
}
```

```
void swap(int *p1,int *p2)    //函数的作用是将*p1 的值与*p2 的值交换
{ int temp;
```

```
temp=*p1;
*p1=*p2;
*p2=temp;
}
```

## 第 6 章 指针

运行情况如下:

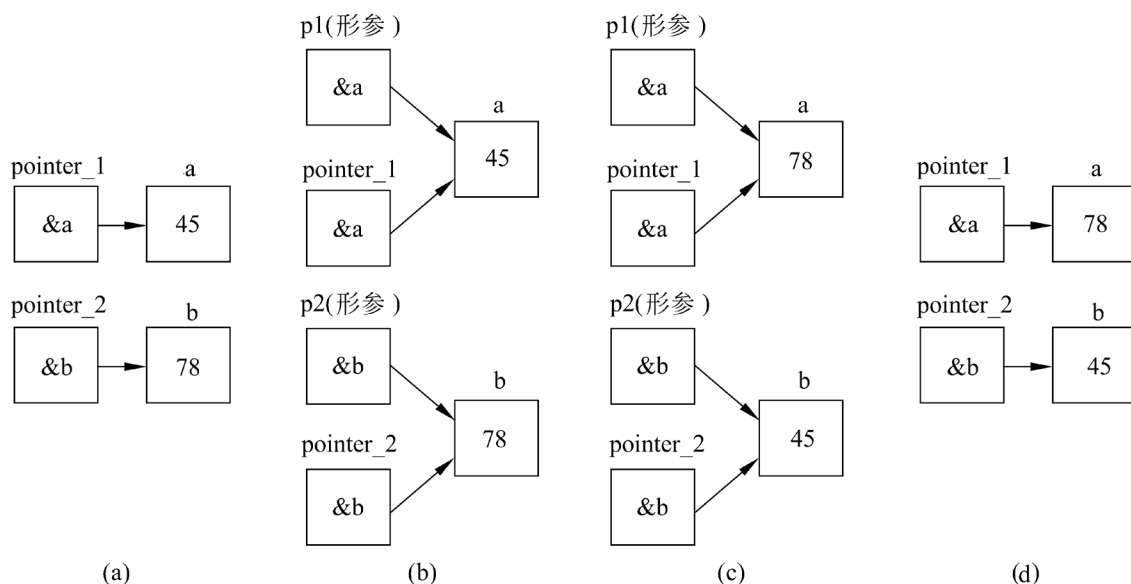
45 78

max=78 min=45

请注意: 不要将 main 函数中的 swap 函数调用写成

**if(a<b) swap(\*pointer\_1,\*pointer\_2);**

图 6.9



## 第 6 章 指针

请注意交换 `*p1` 和 `*p2` 的值是如何实现的。如果写成以下这样就有问题了:

```
void swap(int *p1,int *p2)
```

```
{int *temp;
```

```
*temp=*p1; //此语句有问题
```

```
*p1=*p2;
```

```
*p2=*temp;
```

```
}
```

本例采取的方法是交换 `a` 和 `b` 的值,而 `p1` 和 `p2` 的值不变。这恰和例 6.2 相反。

可以看到,在执行 `swap` 函数后,主函数中的变量 `a` 和 `b` 的值改变了。这个改变不是通过将形参值传回实参来实现的。请读者考虑一下能否通过调用下面的函数实现 `a` 和 `b` 互换。

## 第 6 章 指针

```
void swap(int x,int y)
```

```
{int temp;
```

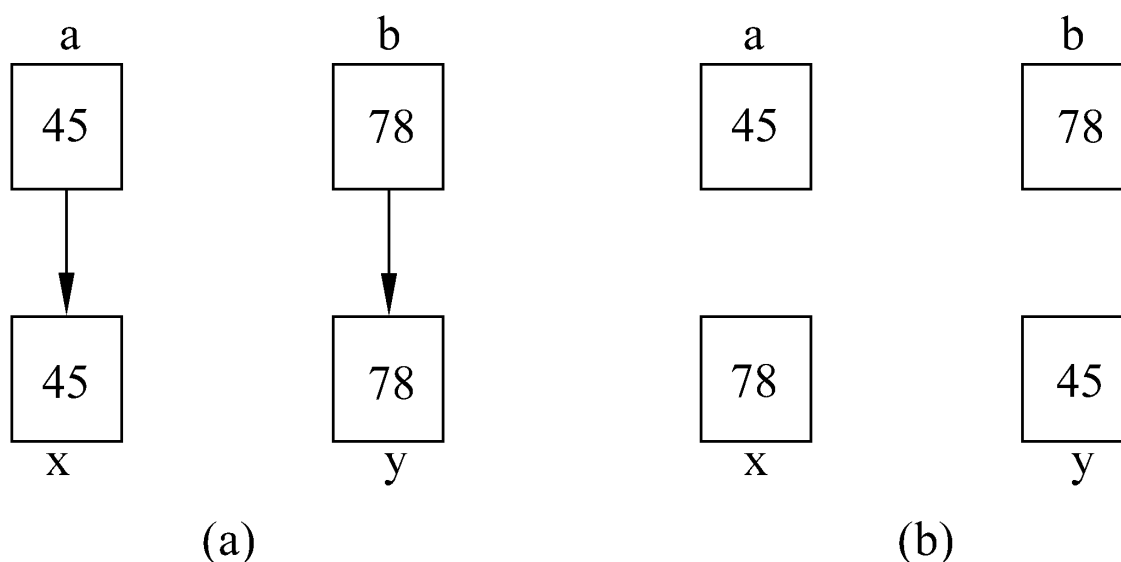
```
temp=x;
x=y;
y=temp;
}
```

在 main 函数中用“swap(a,b);”调用 swap 函数,会有什么结果呢?在函数调用时,a 的值传送给 x,b 的值传送给 y,如图 6.10(a)所示。执行完 swap 函数最后一个语句后,x 和 y 的值是互换了,但 main 函数中的 a 和 b 并未互换,如图 6.10(b)所示。也就是说由于虚实结合是采取单向的“值传递”方式,只能从实参向形参传数据,形参值的改变无法回传给实参。

## 第 6 章 指针

图 6.10

为了使在函数中改变了的变量值能被 main 函数所用,不能采取把要改变值的变量作为参数的办法,而应该用指针变量作为函数参数。在函数执行过程中使指针变量所指向的变量值发生变化,函数调用结束后,这些变量值的变化依然保留下来,这样就实现了“通过调用函数使变量的值发生变化,在主调函数中使用这些改变了的值”的目的。



## 第 6 章 指针

如果想通过函数调用得到 n 个要改变的变量,可以采取下面的步骤: ①在主调函数中设 n 个变量,用 n 个指针变量指向它们;②编写被调函数,其形参为 n 个指针变量,这些形参指针变量应当与主调函数中的 n 个指针变量具有相同的基类型;③在主调函数中将 n 个指针变量作实参,将它们的值(是地址值)传给所调用函数的 n 个形参指针变量,这样,形参指针变量也指向这 n 个变量;④通过形参指针变量的指向,改变该 n 个变量的值;⑤在主调函数中就可以使用这些改变了值的变量。

请注意,不能企图通过改变形参指针变量的值而使实参指针变量的值改变。请分析下面程序:

## 第 6 章 指针

```
#include <iostream>
using namespace std;
int main( )
{ void swap(int *p1,int *p2);
  int *pointer_1,*pointer_2,a,b;
```

```

cin>>a>>b;
pointer_1=&a;
pointer_2=&b;
if(a<b) swap(pointer_1,pointer_2);
cout<<" max=" <<a<<" min=" <<b<<endl;
return 0;
}

```

```

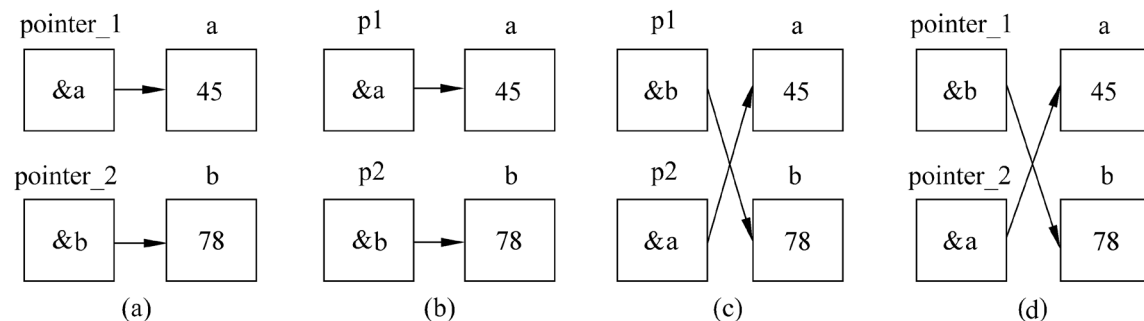
void swap(int *p1,int *p2)
{ int *temp;
temp=p1;
p1=p2;
p2=temp;
}

```

## 第 6 章 指针

图 6.11

实参变量和形参变量之间的数据传递是单向的“值传递”方式。指针变量作函数参数也要遵循这一规则。调用函数时不会改变实参指针变量的值,但可以改变实参指针变量所指向变量的值。



## 第 6 章 指针

函数的调用可以(而且只可以)得到一个返回值(即函数值),而使用指针变量作函数参数,就可以通过指针变量改变主调函数中变量的值,相当于通过函数调用从被调用的函数中得到多个值。如果不用指针变量是难以做到这一点的。

例 6.4 输入 a,b,c 3 个整数,按由大到小的顺序输出。

用上面介绍的方法,用 3 个指针变量指向 3 个整型变量,然后用 swap 函数来实现互换 3 个整型变量的值。

程序如下:

```

#include <iostream>
using namespace std;
int main( )
{ void exchange(int *,int *,int *); //对 exchange 函数的声明
int a,b,c,*p1,*p2,*p3;
cin>>a>>b>>c; //输入 3 个整数

```

## 第 6 章 指针



```
p1=&a;p2=&b;p3=&c;           //指向3个整型变量
exchange(p1,p2,p3);          //交换p1,p2,p3指向的3个整型变量的值
cout<<a<<" " <<b<<" " <<c<<endl;    //按由大到小的顺序输出3个整数
}
```

```
void exchange(int *q1,int *q2,int *q3)
{void swap(int *,int *);      //对swap函数的声明
if(*q1<*q2) swap(q1,q2);      //调用swap,将q1与q2所指向的变量的值互换
if(*q1<*q3) swap(q1,q3);      //调用swap,将q1与q3所指向的变量的值互换
if(*q2<*q3) swap(q2,q3);      //调用swap,将q2与q3所指向的变量的值互换
}
```

```
void swap(int *pt1,int *pt2)   //将pt1与pt2所指向的变量的值互换
{int temp;
temp=*pt1;
*pt1=*pt2;
*pt2=temp;
}
```

运行情况如下:

12 -56 87✓

87 12 -56

## 第6章 指针

### 6.3 数组与指针

#### 6.3.1 指向数组元素的指针

一个变量有地址,一个数组包含若干元素,每个数组元素都在内存中占用存储单元,它们都有相应的地址。指针变量既然可以指向变量,当然也可以指向数组元素(把某一元素的地址放到一个指针变量中)。所谓数组元素的指针就是数组元素的地址。

```
int a[10];           //定义一个整型数组a,它有10个元素
int *p;              //定义一个基类型为整型的指针变量p
p=&a[0];              //将元素a[0]的地址赋给指针变量p,使p指向a[0]
```

在C++中,数组名代表数组中第一个元素(即序号为0的元素)的地址。因此,下面两个语句等价:

## 第6章 指针

```
p=&a[0];
```

```
p=a;
```

在定义指针变量时可以给它赋初值:

```
int *p=&a[0];          //p的初值为a[0]的地址
```

也可以写成

```
int *p=a;              //作用与前一行相同
```

可以通过指针引用数组元素。假设p已定义为一个基类型为整型的指针变量,并已将一个整型数组元素的地址赋给了它,使它指向某一个数组元素。如果有以下赋值语句:

```
*p=1;                  //对p当前所指向的数组元素赋予数值1
```

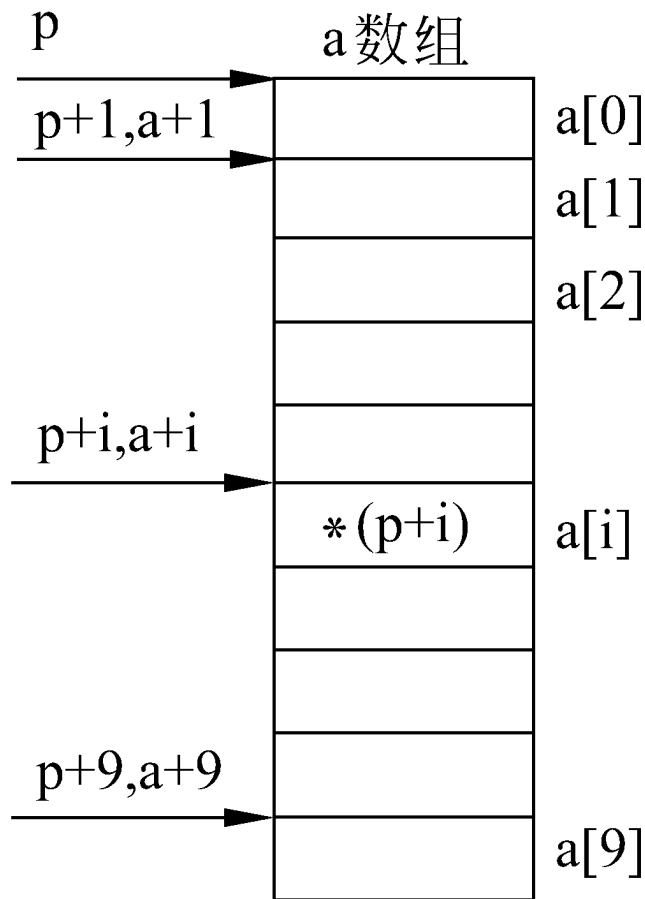
如果指针变量p已指向数组中的一个元素,则p+1指向同一数组中的下一个元素。

## 第6章 指针

如果p的初值为&a[0],则:

(1) p+i 和 a+i 就是 a[i]的地址,或者说,它们指向a数组的第i个元素,见图6.12。

图 6.12



第 6 章 指针

(2)  $*(p+i)$ 或 $*(a+i)$ 是  $p+i$  或  $a+i$  所指向的数组元素,即  $a[i]$ 。

可以看出, $[]$ 实际上是变址运算符。对  $a[i]$ 的求解过程是:先按  $a+i \times d$  计算数组元素的地址,然后找出此地址所指向的单元中的值。

(3) 指向数组元素的指针变量也可以带下标,如  $p[i]$ 与 $*(p+i)$ 等价。

根据以上叙述,引用一个数组元素,可用以下方法:

- (1) 下标法,如  $a[i]$ 形式;
- (2) 指针法,如 $*(a+i)$ 或 $*(p+i)$ 。其中  $a$  是数组名, $p$  是指向数组元素的指针变量。如果已使  $p$  的值为  $a$ ,则 $*(p+i)$ 就是  $a[i]$ 。可以通过指向数组元素的指针找到所需的元素。使用指针法能使目标程序质量高。

第 6 章 指针

例 6.5 输出数组中的全部元素。

假设有一个整型数组  $a$ ,有 10 个元素。要输出各元素的值有 3 种方法:

(1) 下标法

```
#include <iostream>
using namespace std;
int main()
{ int a[10];
```

```

int i;
for(i=0;i<10;i++)
    cin>>a[i];           //引用数组元素a[i]
cout<<endl;
for(i=0;i<10;i++)
    cout<<a[i]<<" ";    //引用数组元素a[i]
cout<<endl;
return 0;
}

```

## 第6章 指针

运行情况如下:

9 8 7 6 5 4 3 2 1 0 (输入 10 个元素的值)

9 8 7 6 5 4 3 2 1 0 (输出 10 个元素的值)

### (2) 指针法

将上面程序第 7 行和第 10 行的“a[i]”改为“\*(a+i)”,运行情况与(1)相同。

### (3) 用指针变量指向数组元素

```

#include <iostream>
using namespace std;
int main( )
{ int a[10];
  int i,*p=a;           //指针变量p 指向数组a 的首元素a[0]
  for(i=0;i<10;i++)
    cin>>*(p+i);        //输入a[0]~a[9]共10个元素
  cout<<endl;

```

## 第6章 指针

```

for(p=a;p<(a+10);p++)
  cout<<*p<<" ";      //p 先后指向a[0]~a[9]
cout<<endl;
return 0;
}

```

运行情况与前相同。请仔细分析 p 值的变化和\*p 的值。

对 3 种方法的比较:

方法(1)和(2)的执行效率是相同的。第(3)种方法比方法(1)、(2)快。这种方法能提高执行效率。

用下标法比较直观,能直接知道是第几个元素。用地址法或指针变量的方法都不太直观,难以很快地判断出当前处理的是哪一个元素。

## 第6章 指针

在用指针变量指向数组元素时要注意: 指针变量 p 可以指向有效的数组元素,实际上也可以指向数组以后的内存单元。如果有

```

int a[10],*p=a;           //指针变量p 的初值为&a[0]
cout<<*(p+10);           //要输出a[10]的值

```

在使用指针变量指向数组元素时,应切实保证指向数组中有效的元素。

指向数组元素的指针的运算比较灵活,务必小心谨慎。下面举几个例子:

如果先使 p 指向数组 a 的首元素(即 p=a),则:

(1) p++(或 p+=1)。使 p 指向下一元素,即 a[1]。如果用\*p,得到下一个元素 a[1]的值。

## 第6章 指针

(2) \*p++。由于++和\*同优先级,结合方向为自右而左,因此它等价于\*(p++)。作用是: 先得到 p 指向的变量的值(即\*p),然后再使 p 的值加 1。例 6.5(3)程序中最后一个 for 语句:

```
for(p=a;p<a+10;p++)
```

```
cout<<*p;
```

可以改写为

```
for(p=a;p<a+10;)
```

```
cout<<*p++;
```

(3)  $*(p++)$ 与 $*(++p)$ 作用不同。前者是先取 $*p$ 值,然后使 $p$ 加1。后者是先使 $p$ 加1,再取 $*p$ 。若 $p$ 的初值为 $a$ (即 $\&a[0]$ ),输出 $*(p++)$ 得到 $a[0]$ 的值,而输出 $*(++p)$ 则得到 $a[1]$ 的值。

## 第6章 指针

(4)  $(*p)++$ 表示 $p$ 所指向的元素值加1,即 $(a[0])++$ ,如果 $a[0]=3$ ,则 $(a[0])++$ 的值为4。注意:是元素值加1,而不是指针值加1。

(5) 如果 $p$ 当前指向 $a[i]$ ,则

$*(p--)$  先对 $p$ 进行“ $--$ ”运算,得到 $a[i]$ ,再使 $p$ 减1, $p$ 指向 $a[i-1]$ 。

$*(++p)$  先使 $p$ 自加1,再作 $*$ 运算,得到 $a[i+1]$ 。

$*(-p)$  先使 $p$ 自减1,再作 $*$ 运算,得到 $a[i-1]$ 。

将 $++$ 和 $--$ 运算符用于指向数组元素的指针变量十分有效,可以使指针变量自动向前或向后移动,指向下一个或上一个数组元素。例如,想输出 $a$ 数组100个元素,可以用以下语句:

## 第6章 指针

```
p=a;
```

```
while(p<a+100)
```

或

```
p=a;
```

```
while(p<a+100)
```

```
cout<<*p++;
```

```
{cout<<*p;
```

```
p++;}
```

在用 $*p++$ 形式的运算时,很容易弄错,一定要十分小心,弄清楚先取 $p$ 值还是先使 $p$ 加1。

## 第6章 指针

### 6.3.2 用指针变量作函数参数接收数组地址

在第5章5.4节中介绍过可以用数组名作函数的参数。前面已经多次强调:数组名代表数组首元素的地址。用数组名作函数的参数,传递的是数组首元素的地址。很容易推想:用指针变量作函数形参,同样可以接收从实参传递来的数组首元素的地址(此时,实参是数组名)。下面将第5章5.4节中的例5.7程序改写,用指针变量作函数形参。

例6.6 将10个整数按由小到大的顺序排列。

在例5.7程序的基础上,将形参改为指针变量。

## 第6章 指针

```
#include <iostream>
```

```
using namespace std;
```

```
int main( )
```

```
{void select_sort(int *p,int n);
```

//函数声明

```
int a[10],i;
```

```
cout<<" enter the originl array:" <<endl;
```

```
for(i=0;i<10;i++)
```

//输入10个数

```
cin>>a[i];
```

```
cout<<endl;
```

```
select_sort(a,10);
```

//函数调用,数组名作实参

```
cout<<" the sorted array:" <<endl;
```

```
for(i=0;i<10;i++)
```

//输出10个已排好序的数

```
cout<<a[i]<<" " ;
```

```
cout<<endl;
```

```
return 0;
```

```
}void select_sort(int *p,int n)
```

//用指针变量作形参

```

{int i,j,k,t;
for(i=0;i<n-1;i++)
{k=i;
  for(j=i+1;j<n;j++)
if(*(p+j)<*(p+k)) k=j;          //用指针法访问数组元素
  t=*(p+k);*(p+k)=*(p+i);*(p+i)=t;
}
}

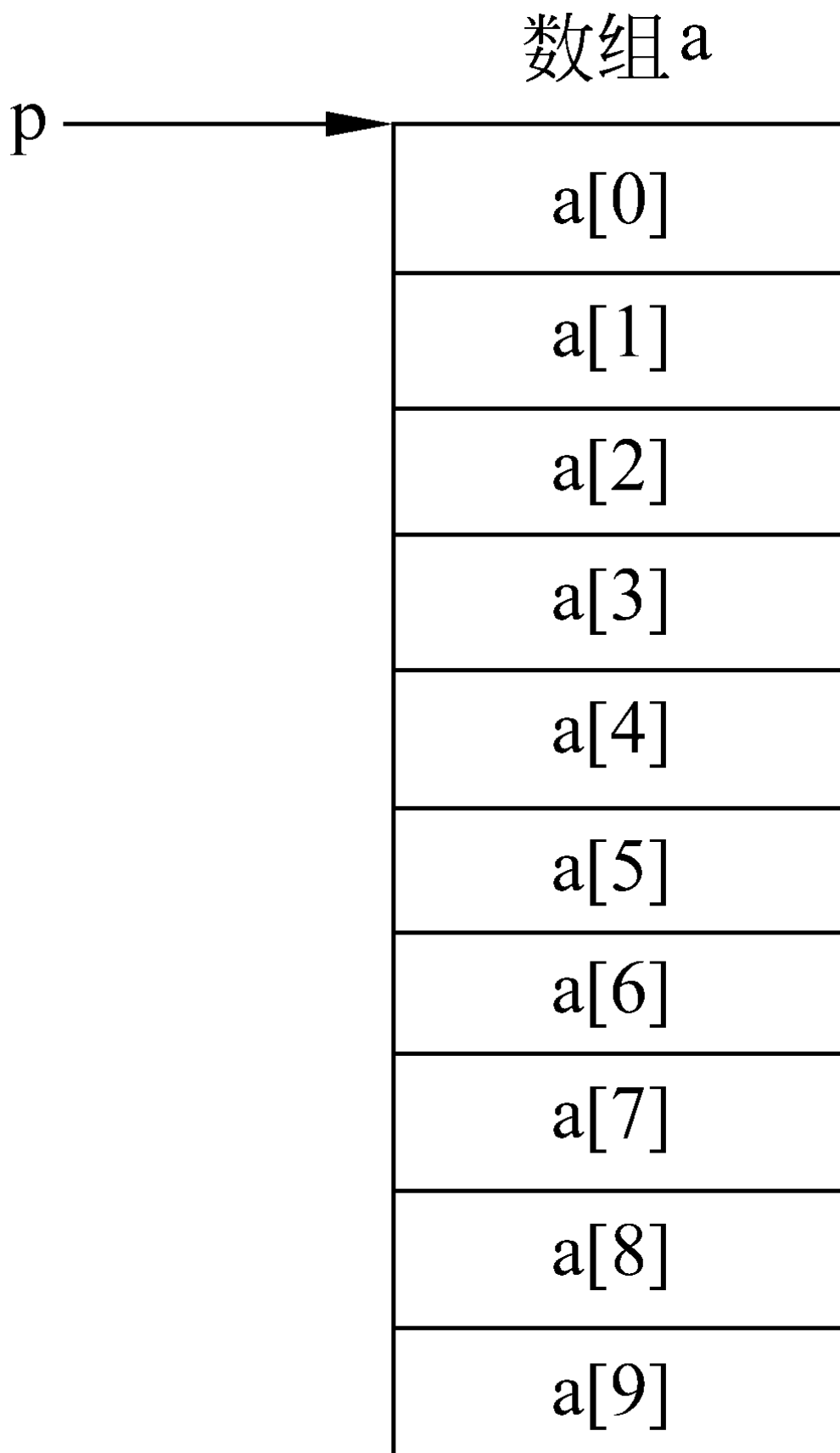
```

## 第 6 章 指针

运行情况与例 5.7 相同。

图 6.13

本例与例 5.7 在程序的表现形式上虽然有不同,但实际上,两个程序在编译以后是完全相同的。C++编译系统将形参数组名一律作为指针变量来处理。



## 第 6 章 指针

实际上在函数调用时并不存在一个占有存储空间的形参数组,只有指针变量。

实参与形参的结合,有以下 4 种形式:

实 参          形 参

数组名                  数组名 (如例 5.7)

数组名                      指针变量      (如例 6.6)

指针变量                  数组名

指针变量                  指针变量

在此基础上,还要说明一个问题: 实参数组名 `a` 代表一个固定的地址,或者说是指针型常量,因此要改变 `a` 的值是不可能的。如

```
a++; //语法错误,a 是常量,不能改变
```

### 第 6 章 指针

而形参数组名是指针变量,并不是一个固定的地址值。它的值是可以改变的。在函数调用开始时,它接收了实参数组首元素的地址,但在函数执行期间,它可以再被赋值。如

```
f(array[],int n)
{ cout<<array; //输出 array[0] 的值
  array=array+3; //指针变量 array 的值改变了,指向 array[3]
  cout<<*arr<<endl; //输出 array[3] 的值
}
```

### 第 6 章 指针

#### 6.3.3 多维数组与指针

用指针变量可以指向一维数组中的元素,也可以指向多维数组中的元素。

##### 1. 多维数组元素的地址

设有一个二维数组 `a`,它有 3 行 4 列。它的定义为

```
int a[3][4]={1,3,5,7},{9,11,13,15},{17,18,21,23}};
```

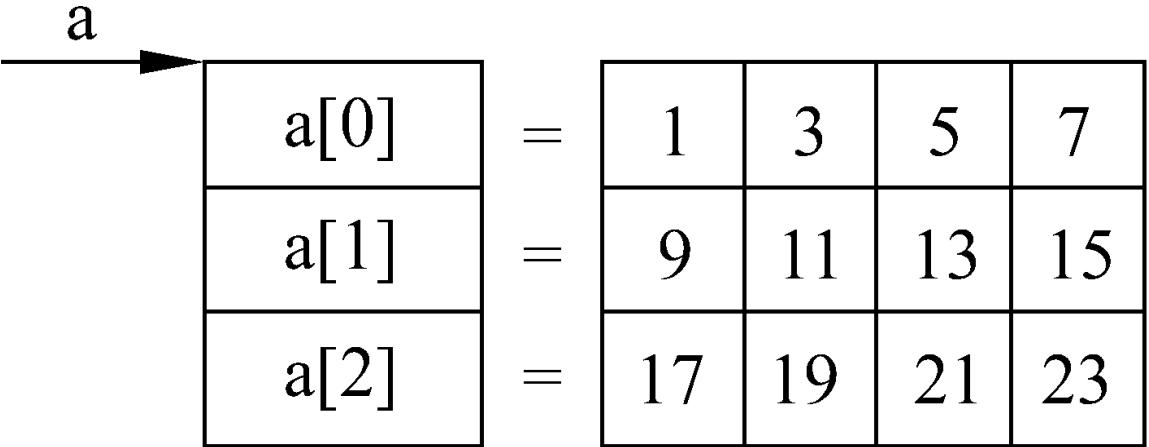
`a` 是一个数组名。`a` 数组包含 3 行,即 3 个元素:`a[0]`,`a[1]`,`a[2]`。而每一元素又是一个一维数组,它包含 4 个元素(即 4 个列元素),例如,`a[0]`所代表的一维数组又包含 4 个元素: `a[0][0]`, `a[0][1]`, `a[0][2]`, `a[0][3]`,见图 6.14。可以认为二维数组是“数组的数组”,即数组 `a` 是由 3 个一维数组所组成的。

### 第 6 章 指针

图 6.14

从二维数组的角度来看,`a` 代表二维数组首元素的地址,现在的首元素不是一个整型变量,而是由 4 个整型元素所组成的一维数组,因此 `a` 代表的是首行的起始地址(即第 0 行的起始地址,`&a[0]`),`a+1` 代表 `a[1]` 行的首地址,即`&a[1]`。

`a[0]`,`a[1]`,`a[2]`既然是一维数组名,而 C++又规定了数组名代表数组首元素地址,因此 `a[0]`代表一维数组 `a[0]` 中 0 列元素的地址,即`&a[0][0]`。`a[1]`的值是`&a[1][0]`,`a[2]`的值是`&a[2][0]`。



### 第 6 章 指针

图 6.15

0 行 1 列元素的地址可以直接写为 $\&a[0][1]$ ,也可以用指针法表示。 $a[0]$ 为一维数组名,该一维数组中序号为 1 的元素显然可以用  $a[0]+1$  来表示,见图 6.16。

欲得到  $a[0][1]$  的值,用地址法怎么表示呢?既然  $a[0]+1$  是  $a[0][1]$  元素的地址,那么,  $*(a[0]+1)$  就是  $a[0][1]$  元素的值。而  $a[0]$  又是和  $*(a+0)$  无条件等价的,因此也可以用  $*(*(a+0)+1)$  表示  $a[0][1]$  元素的值。依此类推,  $*(a[i]+j)$  或  $*(*(a+i)+j)$  是  $a[i][j]$  的值。

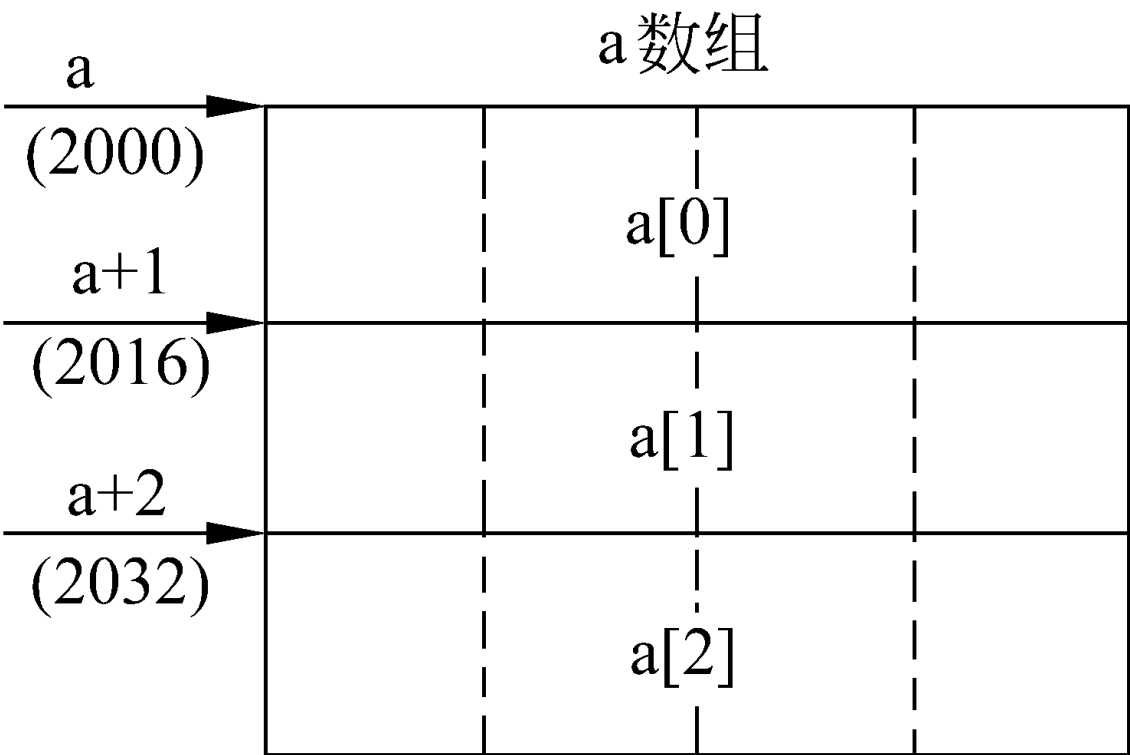


图 6.16



|     | a[0] | a[0]+1 | a[0]+2 | a[0]+3 |
|-----|------|--------|--------|--------|
| a   | 2000 | 2004   | 2008   | 2012   |
| a+1 | 1    | 3      | 5      | 7      |
| a+2 | 2016 | 2020   | 2024   | 2028   |
|     | 9    | 11     | 13     | 15     |
|     | 2032 | 2036   | 2040   | 2044   |
|     | 17   | 19     | 21     | 23     |

## 第 6 章 指针

### 2. 指向多维数组元素的指针变量

#### (1) 指向数组元素的指针变量

例 6.7 输出二维数组各元素的值。

这里采用的方法是用基类型为整型的指针变量先后指向各元素,逐个输出它们的值。

```
#include <iostream>
using namespace std;
int main()
{ int a[3][4]={1,3,5,7,9,11,13,15,17,19,21,23};
  int *p; //p 是基类型为整型的指针变量
  for(p=a[0];p<a[0]+12;p++)
    cout<<*p<<" ";
  cout<<endl;
  return 0;
}
```

## 第 6 章 指针

运行结果如下:

1 3 5 7 9 11 13 15 17 19 21 23

说明:

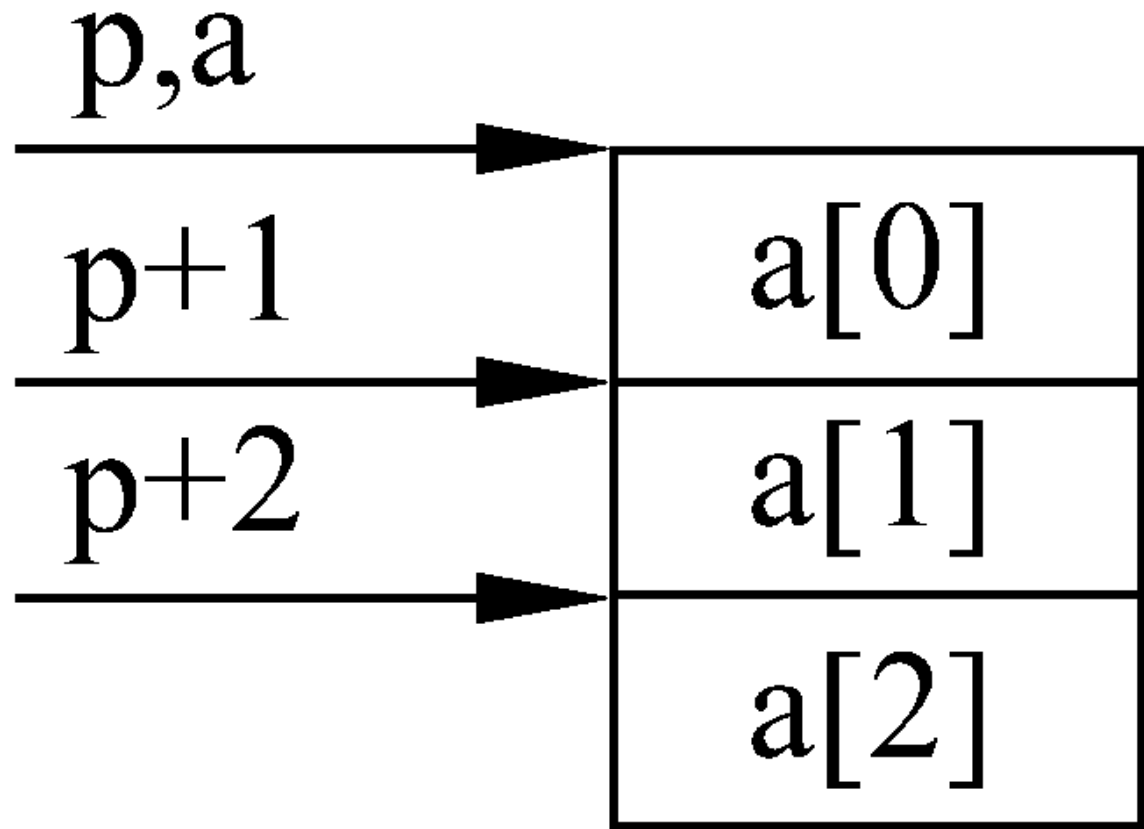
- ① p 是指向整型数据的指针变量,在 for 语句中对 p 赋初值 a[0],也可以写成“p=&a[0][0]”。
- ② 循环结束的条件是“p<a[0]+12”,只要满足 p<a[0]+12,就继续执行循环体。
- ③ 执行“cout<<\*p;”输出 p 当前所指的列元素的值,然后执行 p++,使 p 指向下一个列元素。

## 第 6 章 指针

(2) 指向由 m 个元素组成的一维数组的指针变量

可以定义一个指针变量,它不是指向一个整型元素,而是指向一个包含 m 个元素的一维数组。这时,如果指针变量 p 先指向 a[0](即 p=&a[0]),则 p+1 不是指向 a[0][1],而是指向 a[1],p 的增值以一维数组的长度为单位,见图 6.17。

图 6.17



第 6 章 指针

例 6.8 输出二维数组任一行任一列元素的值。

```
#include <iostream>
using namespace std;
int main( )
{ int a[3][4]={1,3,5,7,9,11,13,15,17,19,21,23};
  int (*p)[4],i,j;
  cin>>i>>j;
  p=a;
  cout<<*((p+i)+j)<<endl;
  return 0;
}
```

运行情况如下:

2 3↵

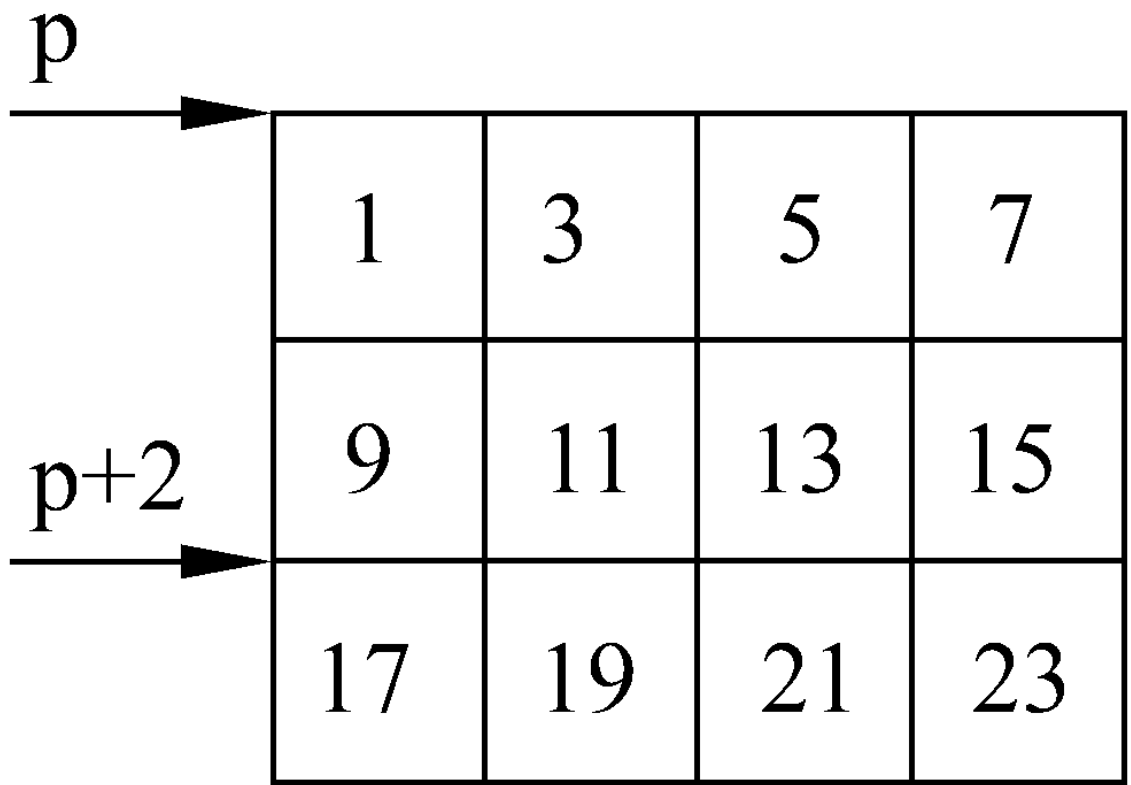
23

第 6 章 指针

由于执行了“p=a”,使 p 指向 a[0]。因此 p+2 是二维数组 a 中序号为 2 的行的起始地址(由于 p 是指向一维数

组的指针变量,因此 p 加 1,就指向下一个一维数组),见图 6.18。\*(p+2)+3 是 a 数组 2 行 3 列元素地址。  
\*(\*(p+2)+3)是 a[2][3]的值。

图 6.18



第 6 章 指针

3. 用指向数组的指针作函数参数

一维数组名可以作为函数参数传递,多维数组名也可作函数参数传递。

例 6.9 输出二维数组各元素的值。

题目与例 6.7 相同,但本题用一个函数实现输出,用多维数组名作函数参数。

```
#include <iostream>
using namespace std;
int main( )
{ void output(int (*p)[4]);           //函数声明
int a[3][4]={1,3,5,7,9,11,13,15,17,19,21,23};
output(a);                           //多维数组名作函数参数
return 0;
}
```

第 6 章 指针

```
void output(int (*p)[4])              //形参是指向一维数组的指针变量
{ int i,j;
for(i=0;i<3;i++)
for(j=0;j<4;j++)
```

```
cout<<*(p+i+j)<<" ";
cout<<endl;
}
```

运行情况如下:

```
1 3 5 7 9 11 13 15 17 19 21 23
```

## 第 6 章 指针

### 6.4 字符串与指针

在 C++中可以用 3 种方法访问一个字符串(在第 5 章介绍了前两种方法)。

#### 1. 用字符数组存放一个字符串

例 6.10 定义一个字符数组并初始化,然后输出其中的字符串。

```
#include <iostream>
using namespace std;
int main( )
{ char str[]=" I love CHINA!" ;
cout<<str<<endl;
return 0;
}
```

## 第 6 章 指针

运行时输出:

```
I love CHINA!
```

#### 2. 用字符串变量存放字符串

例 6.11 定义一个字符串变量并初始化,然后输出其中的字符串。

```
#include <string>
#include <iostream>
using namespace std;
int main( )
{ string str=" I love CHINA!" ;
cout<<str<<endl;
return 0;
}
```

## 第 6 章 指针

#### 3. 用字符指针指向一个字符串

例 6.12 定义一个字符指针变量并初始化,然后输出它指向的字符串。

```
#include <iostream>
using namespace std;
int main( )
{ char *str=" I love CHINA!" ;
cout<<str<<endl;
return 0;
}
```

对字符串中字符的存取,可以用下标方法,也可以用指针方法。

例 6.13 将字符串 str1 复制为字符串 str2。

## 第 6 章 指针

定义两个字符数组 str1 和 str2,再设两个指针变量 p1 和 p2,分别指向两个字符数组中的有关字符,通过改变指针变量的值使它们指向字符串中的不同的字符,以实现字符的复制。

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

```

int main( )
{ char str1[]=" I love CHINA!" ,str2[20],*p1,*p2;
p1=str1;p2=str2;
for(;*p1!=' \0' ;p1++,p2++)
    *p2=*p1;
*p2=' \0' ;
p1=str1;p2=str2;
cout<<" str1 is: " <<p1<<endl;
cout<<" str2 is: " <<p2<<endl;
return 0;
}

```

## 第 6 章 指针

运行结果为

```

str1 is: I love CHINA!
str2 is: I love CHINA!

```

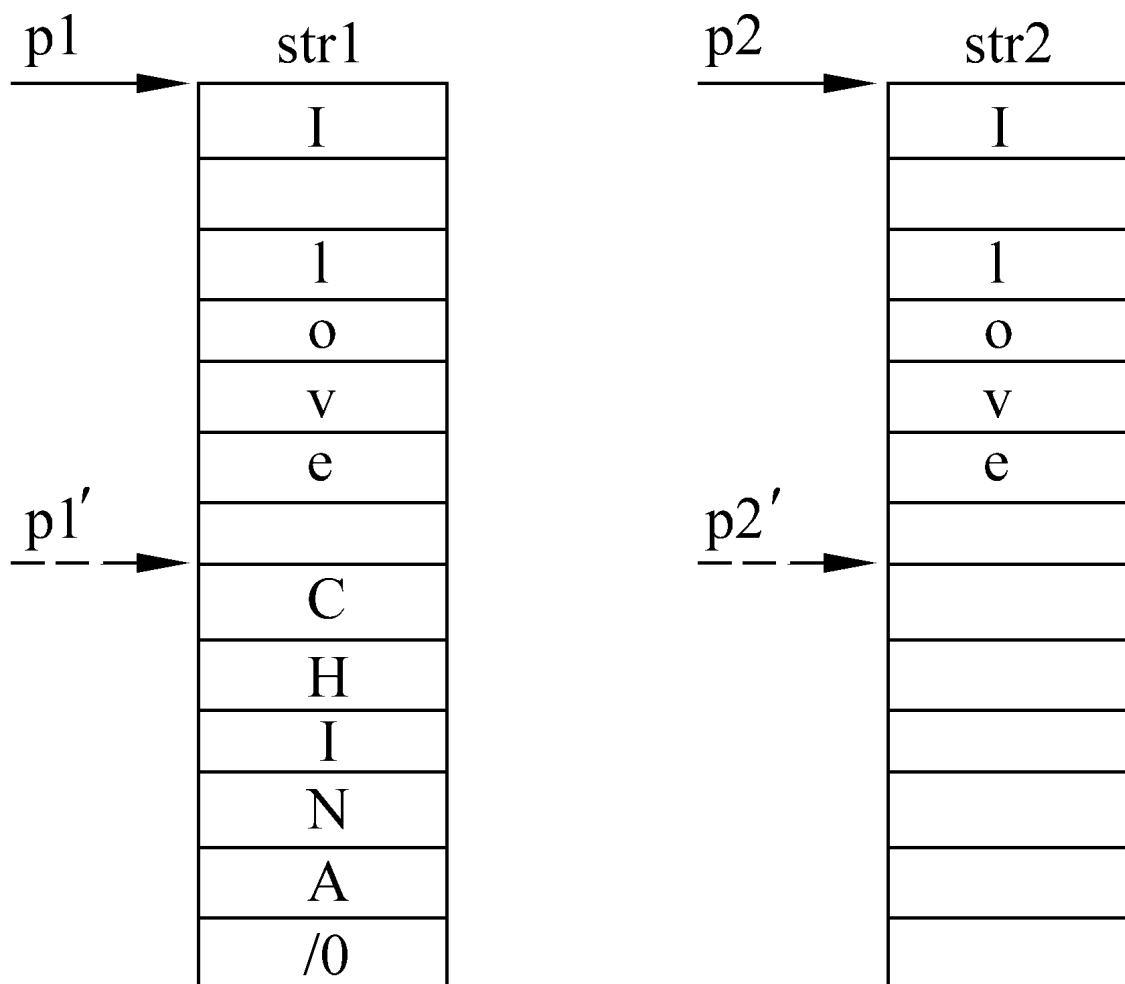
图 6.19

这个例子用来说明怎样使用字符指针。其实,对例 6.13 来说,用 string 变量来处理是十分简单的:

```

string str1=" I love CHINA!" ,str2;    //定义 string 变量
str2=str1;                            //将 str1 复制到 str2

```



## 第 6 章 指针

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。

例 6.14 求 a 和 b 中的大者。

先按一般方法写程序:

```
#include <iostream>
using namespace std;
int main()
{int max(int x,int y);           //函数声明
```

### 6.5 函数与指针

#### 6.5.1 用函数指针变量调用函数

## 第 6 章 指针

```
int a,b,m;
cin>>a>>b;
m=max(a,b);           //调用函数 max,求出最大值,赋给 m
cout<<" max=" <<m<<endl;
return 0;
}
```

```
int max(int x,int y)
{int z;
if(x>y) z=x;
else z=y;
```

```
return(z);
}
```

## 第 6 章 指针

可以用一个指针变量指向 `max` 函数,然后通过该指针变量调用此函数。定义指向 `max` 函数的指针变量的方法是:

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

`p` 所指向的函数的形参类型

`p` 是指向函数的指针变量

指针变量 `p` 指向的函数的类型

请将它和函数 `max` 的原型作比较

```
int max(int,int);           //max 函数原型
```

可以看出: 只是用 `(*p)` 取代了 `max`,其他都一样。现在将上面程序的主函数修改如下:

## 第 6 章 指针

```
#include <iostream>
```

```
using namespace std;
```

```
int main( )
```

```
{int max(int x,int y);           //函数声明
```

```
int (*p)(int,int);             //定义指向函数的指针变量p
```

```
int a,b,m;
```

```
p=max;                        //使p 指向函数max
```

```
cin>>a>>b;
```

```
m=p(a,b);
```

```
cout<<" max=" <<m<<endl;
```

```
return 0;
```

```
}
```

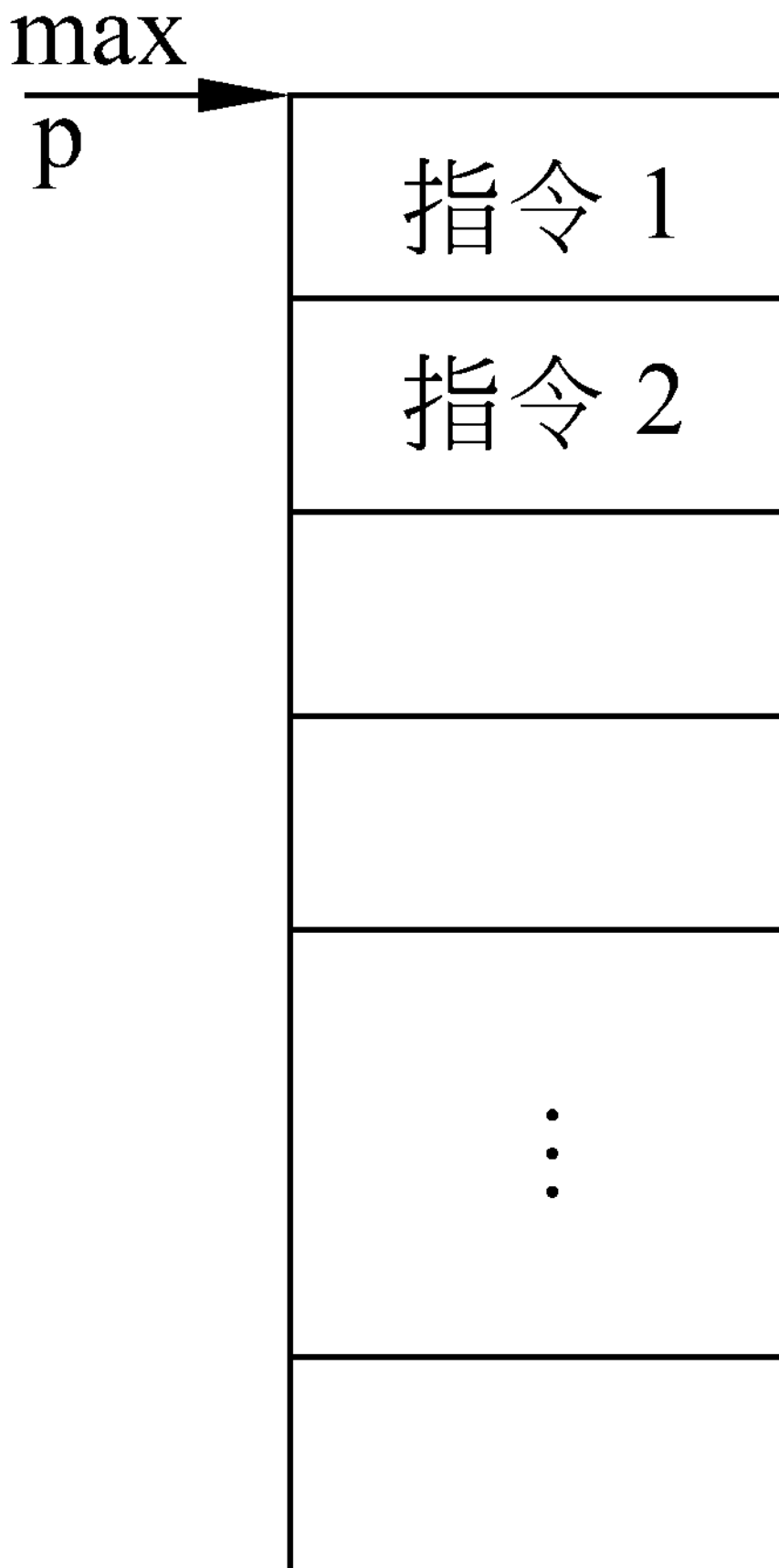
请注意第 7 行的赋值语句“`p=max;`”。此语句千万不要漏写,它的作用是将函数 `max` 的入口地址赋给指针变量 `p`。这时,`p` 才指向函数 `max`。见图 6.20。

## 第 6 章 指针

图 6.20

指向函数的指针变量的一般定义形式为

函数类型 (\*指针变量名)(函数形参表);



## 第 6 章 指针

### 6.5.2 用指向函数的指针作函数参数

在 C 语言中,函数指针变量常见的用途之一是作为函数的参数,将函数名传给其他函数的形参。这样就可以在调用一个函数的过程中根据给定的不同实参调用不同的函数。

例如,利用这种方法可以编写一个求定积分的通用函数,用它分别求 5 个函数的定积分:每次需要求定积分的



函数是不一样的。可以编写一个求定积分的通用函数 `integral`, 它有 3 个形参: 下限 `a`、上限 `b`, 以及指向函数的指针变量 `fun`。函数原型可写为

```
double integral (double a,double b,double (*fun)(double));
```

## 第 6 章 指针

分别编写 5 个函数 `f1`, `f2`, `f3`, `f4`, `f5`, 用来求上面 5 个函数的值。然后先后调用 `integral` 函数 5 次, 每次调用时把 `a`, `b` 以及 `f1`, `f2`, `f3`, `f4`, `f5` 之一作为实参, 即把上限、下限以及有关函数的入口地址传送给形参 `fun`。在执行 `integral` 函数过程中求出各函数定积分的值。

在面向对象的 C++ 程序设计中, 这种用法就比较少了。有兴趣的读者可参阅作者所著的《C 程序设计(第二版)》一书中的有关章节。

## 第 6 章 指针

### 6.6 返回指针值的函数

一个函数可以带回一个整型值、字符值、实型值等, 也可以带回指针型的数据, 即地址。其概念与以前类似, 只是带回的值的类型是指针类型而已。返回指针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

```
int *a(int x,int y);
```

## 第 6 章 指针

### 6.7 指针数组和指向指针的指针

#### 6.7.1 指针数组的概念

如果一个数组, 其元素均为指针类型数据, 该数组称为指针数组, 也就是说, 指针数组中的每一个元素相当于一个指针变量, 它的值都是地址。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];
```

可以用指针数组中各个元素分别指向若干个字符串, 使字符串处理更加方便灵活。

## 第 6 章 指针

例 6.15 若干字符串按字母顺序(由小到大)输出。

```
#include <iostream>
```

```
using namespace std;
```

```
int main( )
```

```
{ void sort(char *name[],int n); //声明函数
```

```
void print(char *name[],int n); //声明函数
```

```
char *name[]={ " BASIC" ," FORTRAN" ," C++" ," Pascal" ," COBOL" }; //定义指针数组
```

```
int n=5;
```

```
sort(name,n);
```

```
print(name,n);
```

```
return 0;
```

```
}
```

```
void sort(char *name[],int n)
```

```
{ char *temp;
```

```
int i,j,k;
```

```
for(i=0;i<n-1;i++)
```

```
{k=i;
```

## 第 6 章 指针

```
for(j=i+1;j<n;j++)
```

```
if(strcmp(name[k],name[j])>0) k=j;
```

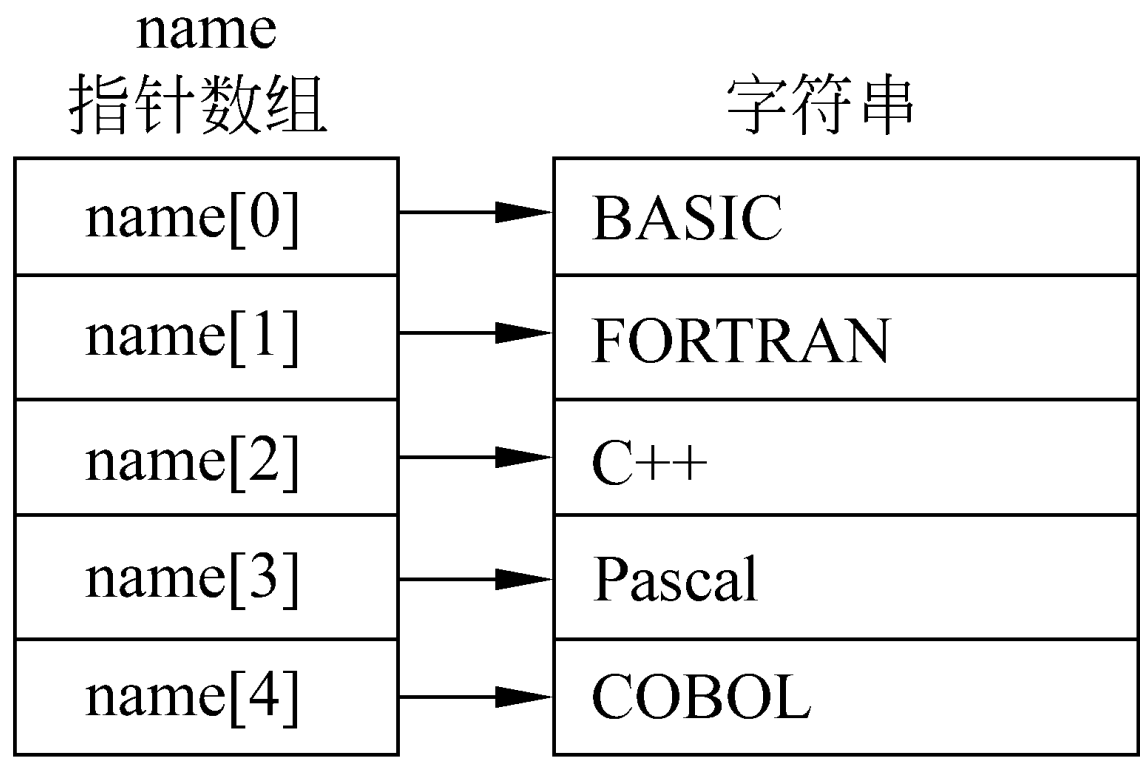
```
if(k!=i)
{ temp=name[i];name[i]=name[k];name[k]=temp;}
}
}
```

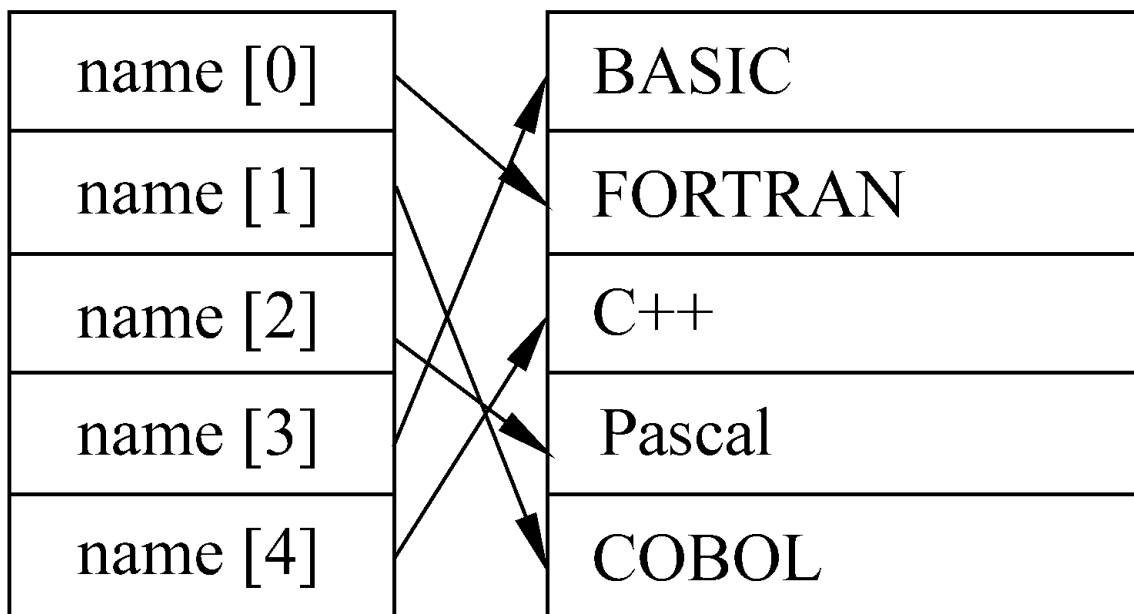
```
void print(char *name[],int n)
{ int i;
for(i=0;i<n;i++)
cout<<name[i]<<endl;
}
```

运行结果为  
BASIC  
COBOL  
C++  
FORTRAN  
Pascal  
第 6 章 指针

图 6.21

图 6.22





## 第 6 章 指针

`print` 函数的作用是输出各字符串。`name[0]~name[4]`分别是各字符串的首地址。`print` 函数也可改写为以下形式:

```
void print(char *name[],int n)
{ int i=0
char *p;
p=name[0];
while(i<n)
{p=*(name+i++);
cout<<p<<endl;
}
}
```

其中“`*(name+i++)`”表示先求`*(name+i)`的值,即 `name[i]`(它是一个地址)。将它赋给 `p`,然后 `i` 加 1。最后输出以 `p` 地址开始的字符串。

## 第 6 章 指针

### 6.7.2 指向指针的指针

在掌握了指针数组的概念的基础上,下面介绍指向指针数据的指针,简称为指向指针的指针。从图 6.22 可以看到,`name` 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),分别指向不同的字符串。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(见图 6.23)。`p` 就是指向指针型数据的指针变量。

## 第 6 章 指针

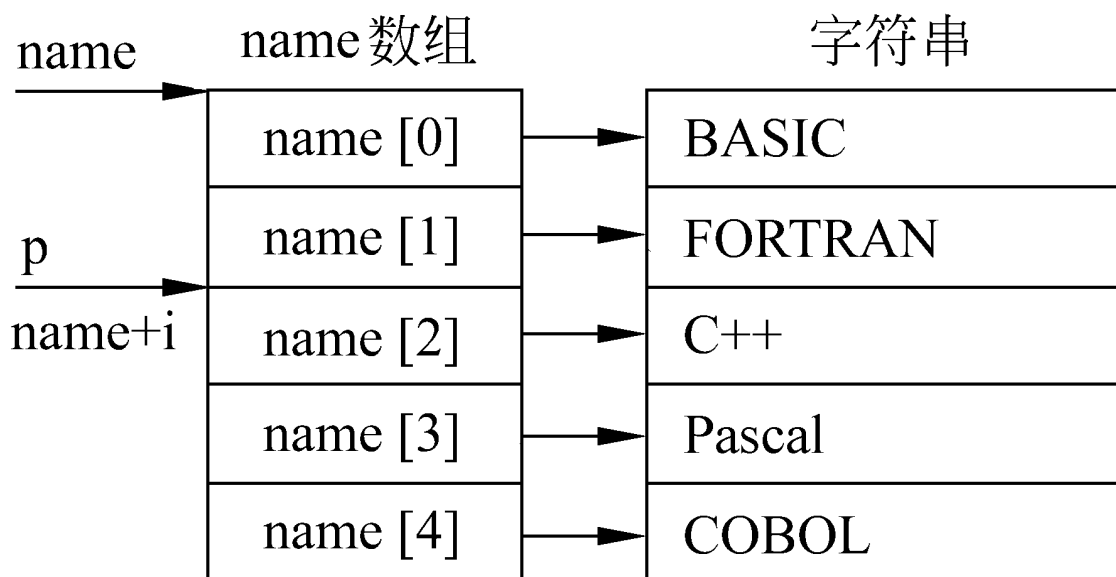
图 6.23

怎样定义一个指向指针数据的指针变量呢?如下:

```
char *(*p);
```

从附录 B 可以知道,\*运算符的结合性是从右到左,因此“`char *(*p);`”可写成

`char **p;`



## 第 6 章 指针

例 6.16 指向字符型数据的指针变量。

```
#include <iostream>
using namespace std;
int main()
{ char **p; //定义指向字符指针数据的指针变量p
  char *name[]={ " BASIC" ," FORTRAN" ," C++" ," Pascal" ," COBOL" };
  p=name+2; //见图 6.23 中 p 的指向
  cout<<*p<<endl; //输出 name[2]指向的字符串
  cout<<**p<<endl; //输出 name[2]指向的字符串中的第一个字符
}
```

运行结果为

C++

C

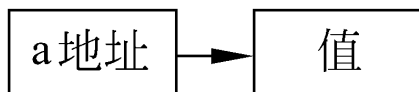
指针数组的元素也可以不指向字符串,而指向整型数据或单精度型数据等。

## 第 6 章 指针

在本章开头已经提到了“间接访问”一个变量的方式。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”,见图 6.24(a)。指向指针的指针用的是“二级间址”方法。见图 6.24(b)。从理论上说,间址方法可以延伸到更多的级,见图 6.24(c)。但实际上在程序中很少有超过二级间址的。

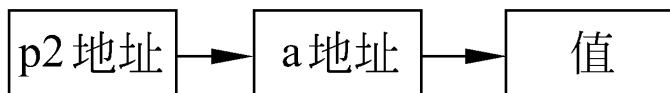
图 6.24

指针变量 p    变量 a



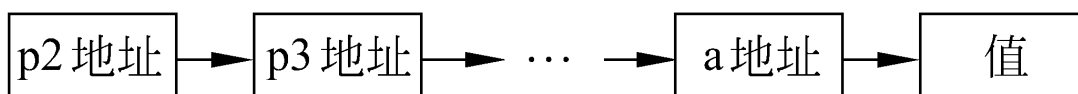
(a)

指针变量 p1    指针变量 p2    变量 a



(b)

指针变量 p1    指针变量 p2                      指针变量 pn    变量 a



(c)

## 第 6 章 指针

### 6.8 有关指针的数据类型和指针运算的小结

#### 6.8.1 有关指针的数据类型的小结

表 6.1 有关指针的数据类型

## 第 6 章 指针

### 6.8.2 指针运算小结

前面已用过一些指针运算(如  $p++$ ,  $p+i$  等),现在把全部的指针运算列出如下。

#### (1) 指针变量加/减 一个整数

例如:  $p++$ ,  $p--$ ,  $p+i$ ,  $p-i$ ,  $p+=i$ ,  $p-=i$  等。

C++规定,一个指针变量加/减一个整数是将该指针变量的原值(是一个地址)和它指向的变量所占用的内存单元字节数相加或相减。如  $p+i$  代表这样的地址计算:  $p+i*d$ ,  $d$  为  $p$  所指向的变量单元所占用的字节数。这样才能保证  $p+i$  指向  $p$  下面的第  $i$  个元素。

#### (2) 指针变量赋值

将一个变量地址赋给一个指针变量。如

## 第 6 章 指针

```
p=&a;           //将变量 a 的地址赋给 p
p=array;        //将数组 array 首元素的地址赋给 p
p=&array[i];     //将数组 array 第 i 个元素的地址赋给 p
p=max;          //max 为已定义的函数,将 max 的入口地址赋给 p
p1=p2;          //p1 和 p2 都是同类型的指针变量,将 p2 的值赋给 p1
```

#### (3) 指针变量可以有空值,即该指针变量不指向任何变量,可以这样表示:

```
p=NULL;
```

实际上 NULL 代表整数 0,也就是使  $p$  指向地址为 0 的单元。这样可以使指针不指向任何有效的单元。实际上系统已先定义了

```
NULL: #define NULL 0
```

在 `iostream` 头文件中就包括了以上的 NULL 定义,NULL 是一个符号常量。应注意, $p$  的值等于 NULL 和  $p$  未被赋值是两个不同的概念。

## 第 6 章 指针

任何指针变量或地址都可以与 NULL 作相等或不相等的比较,如

```
if(p==NULL) p=p1;
```

(4) 两个指针变量可以相减

如果两个指针变量指向同一个数组的元素,则两个指针变量值之差是两个指针之间的元素个数,见图 6.25。

假如 p1 指向 a[1],p2 指向 a[4],则  $p2-p1=(a+4)-(a+1)=4-1=3$ 。

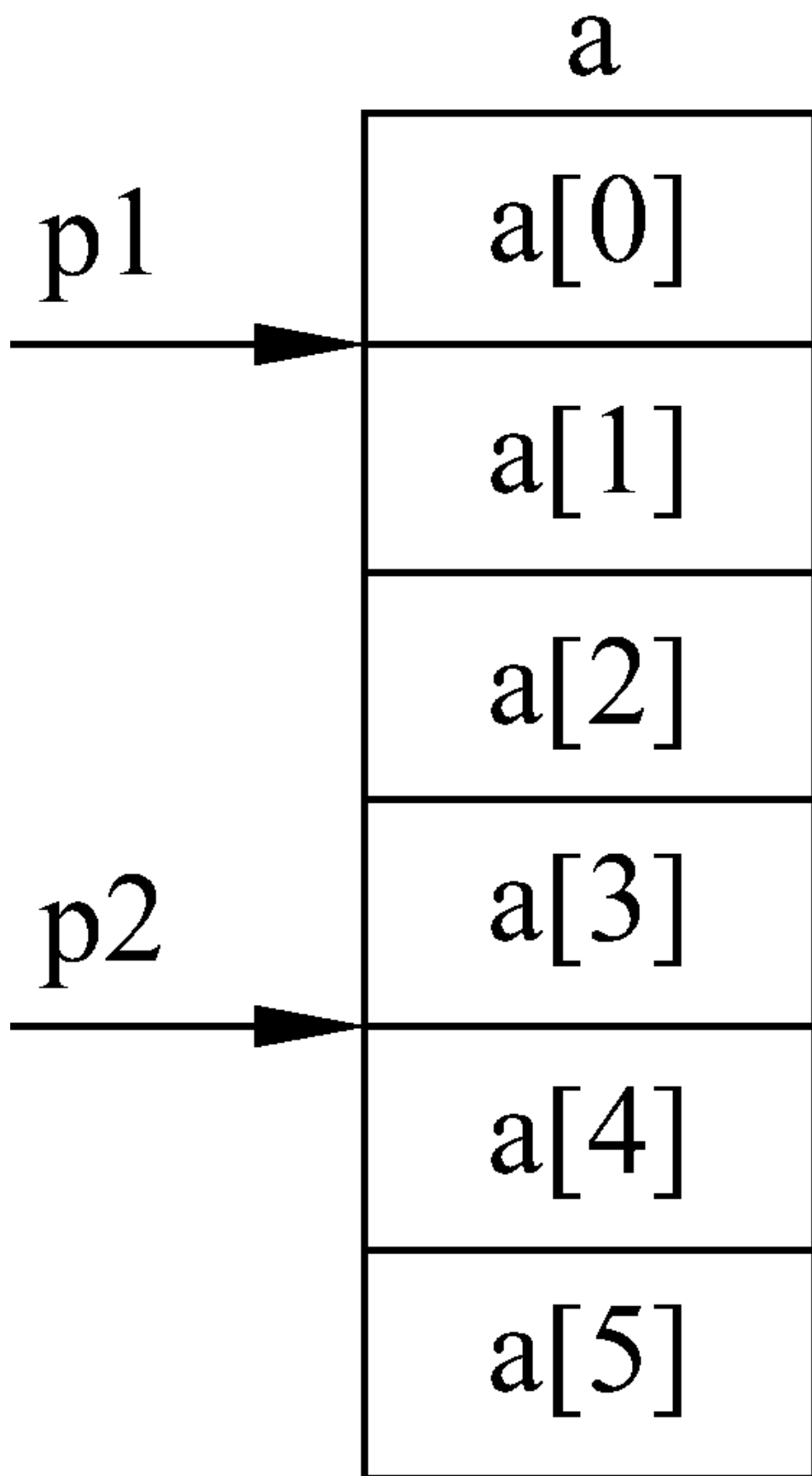
但 p1+p2 并无实际意义。

#### 第 6 章 指针

(5) 两个指针变量比较

若两个指针指向同一个数组的元素,则可以进行比较。指向前面的元素的指针变量小于指向后面元素的指针变量。如图 6.25 中, $p1 < p2$ ,或者说,表达式“ $p1 < p2$ ”的值为真,而“ $p2 < p1$ ”的值为假。注意,如果 p1 和 p2 不指向同一数组则比较无意义。

图 6.25



## 第 6 章 指针

(6) 对指针变量的赋值应注意类型问题。

在本章前几节中介绍了指针的基本概念和初步应用。应该说明,指针是 C 和 C++ 中重要的概念,是 C 和 C++ 的一个特色。使用指针的优点是: ①提高程序效率;②在调用函数时,如果改变被调用函数中某些变量的值,这些值能为主调函数使用,即可以通过函数的调用,得到多个可改变的;③可以实现动态存储分配。

但是同时应该看到,指针使用实在太灵活,对熟练的程序人员来说,可以利用它编写出颇有特色的、质量优良的程序,实现许多用其他高级语言难以实现的功能,但也十分容易出错,而且这种错误往往难以发现。

## 第 6 章 指针

### \*6.9 引用

#### 6.9.1 什么是变量的引用

对一个数据可以使用“引用”(reference),这是 C++对 C 的一个重要扩充,引用是一种新的变量类型,它的作用是为一个变量起一个别名。假如有一个变量 a,想给它起一个别名 b,可以这样写:

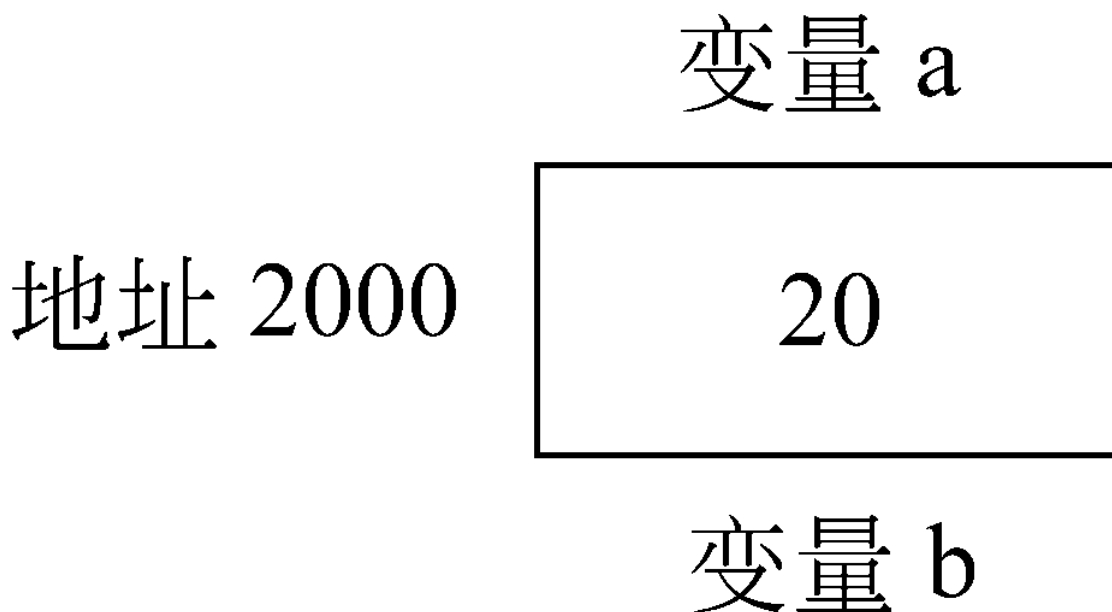
```
int a;           //定义 a 是整型变量
int &b=a;         //声明 b 是 a 的引用
```

以上语句声明了 b 是 a 的引用,即 b 是 a 的别名。经过这样的声明后,a 或 b 的作用相同,都代表同一变量。

## 第 6 章 指针

注意:在上述声明中,&是引用声明符,并不代表地址。不要理解为“把 a 的值赋给 b 的地址”。声明变量 b 为引用类型,并不需要另外开辟内存单元来存放 b 的值。b 和 a 占内存中的同一个存储单元,它们具有同一地址。声明 b 是 a 的引用,可以理解为:使变量 b 具有变量 a 的地址。见图 6.26,如果 a 的值是 20,则 b 的值也是 20。

图 6.26



## 第 6 章 指针

在声明一个引用类型变量时,必须同时使之初始化,即声明它代表哪一个变量。在声明变量 b 是变量 a 的引用后,在它们所在函数执行期间,该引用类型变量 b 始终与其代表的变量 a 相联系,不能再作为其他变量的引用(别名)。下面的用法不对:

```
int a1,a2;
int &b=a1;
int &b=a2;           //企图使 b 又变成 a2 的引用(别名)是不行的
```

## 第 6 章 指针

### 6.9.2 引用的简单使用

例 6.17 引用和变量的关系。

```
#include <iostream>
```



```

#include <iomanip>
using namespace std;
int main( )
{ int a=10;
  int &b=a;           //声明 b 是 a 的引用
  a=a*a;             //a 的值变化了,b 的值也应一起变化
  cout<<a<<setw(6)<<b<<endl;
  b=b/5;             //b 的值变化了,a 的值也应一起变化
  cout<<b<<setw(6)<<a<<endl;
  return 0;
}

```

## 第 6 章 指针

a 的值开始为 10,b 是 a 的引用,它的值当然也应该是 10,当 a 的值变为 100(a\*a 的值)时,b 的值也随之变为 100。在输出 a 和 b 的值后,b 的值变为 20,显然 a 的值也应为 20。

运行记录如下:

```

100 100           (a 和 b 的值都是 100)
20  20           (a 和 b 的值都是 20)

```

## 第 6 章 指针

### 6.9.3 引用作为函数参数

有了变量名,为什么还需要一个别名呢?C++之所以增加引用类型,主要是把它作为函数参数,以扩充函数传递数据的功能。

到目前为止,本书介绍过函数参数传递的两种情况。

(1) 将变量名作为实参和形参。这时传给形参的是变量的值,传递是单向的。如果在执行函数期间形参的值发生变化,并不传回给实参。因为在调用函数时,形参和实参不是同一个存储单元。

例 6.18 要求将变量 i 和 j 的值互换。下面的程序无法实现此要求。

## 第 6 章 指针

```

#include <iostream>
using namespace std;
int main( )
{ void swap(int,int);           //函数声明
  int i=3,j=5;
  swap(i,j);                   //调用函数 swap
  cout<<i<<" " <<j<<endl;      //i 和 j 的值未互换
  return 0;
}

void swap(int a,int b)          //企图通过形参 a 和 b 的值互换,实现实参 i 和 j 的值互换
{ int temp;
  temp=a;                      //以下 3 行用来实现 a 和 b 的值互换
  a=b;
  b=temp;
}

```

运行时输出 3 5 i 和 j 的值并未互换。见图 6.27 示意。

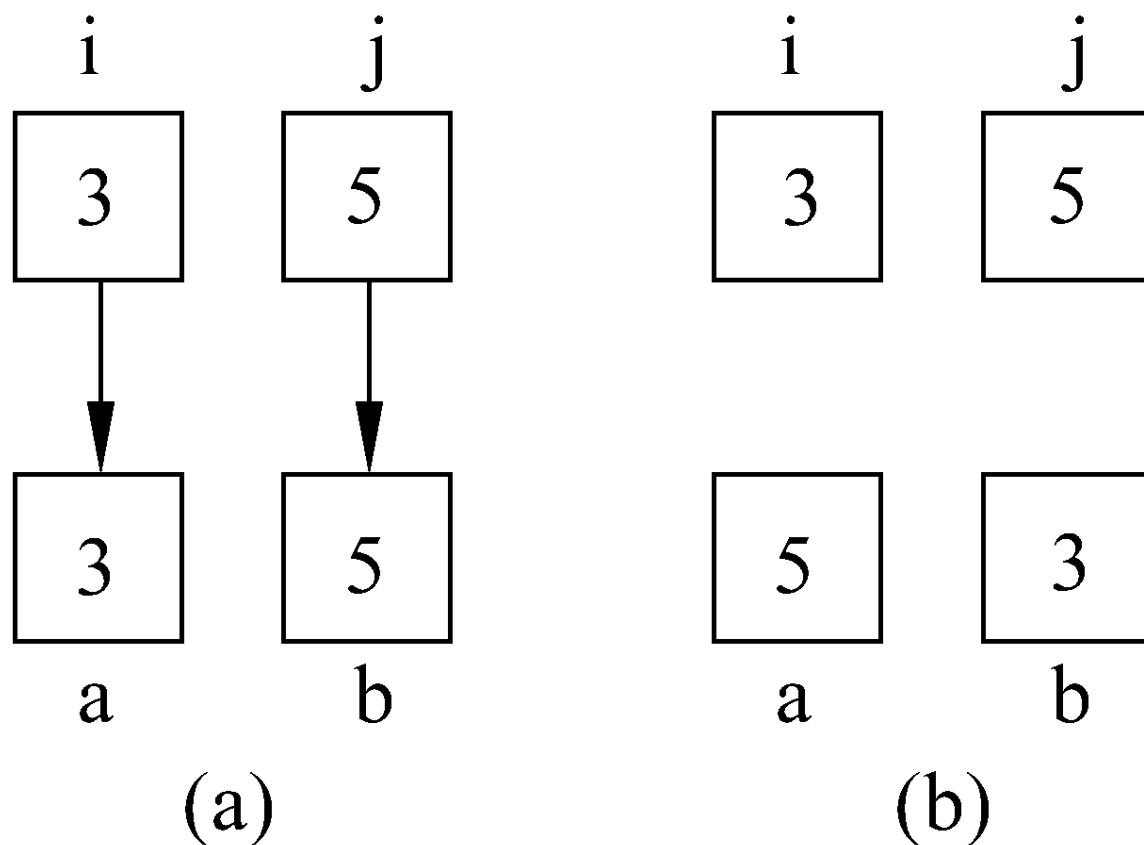
为了解决这个问题,采用传递变量地址的方法。

## 第 6 章 指针

图 6.27

(2) 传递变量的指针。形参是指针变量,实参是一个变量的地址,调用函数时,形参(指针变量)指向实参变量单元。程序见例 6.19。

例 6.19 使用指针变量作形参,实现两个变量的值互换。



## 第 6 章 指针

```
#include <iostream>
```

```
using namespace std;
```

```
int main( )
```

```
{ void swap(int *,int *);
```

```
int i=3,j=5;
```

```
swap(&i,&j);
```

```
cout<<i<<" " <<j<<endl;
```

```
return 0;
```

```
}
```

//实参是变量的地址

//i 和j 的值已互换

```
void swap(int *p1,int *p2)
```

//形参是指针变量

```
{ int temp;
```

```
temp=*p1;
```

//以下3 行用来实现i 和j 的值互换

```
*p1=*p2;
```

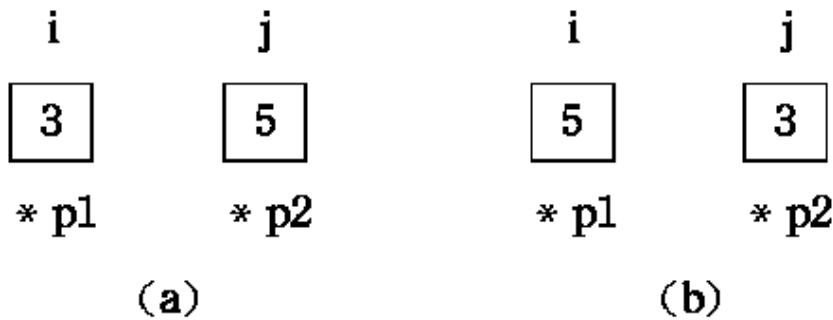
```
*p2=temp;
```

```
}
```

## 第 6 章 指针

形参与实参的结合见图 6.28 示意。

图 6.28  
这种虚实结合的方法仍然是“值传递”方式,只是实参的值是变量的地址而已。通过形参指针变量访问主函数中的变量(i 和 j),并改变它们的值。这样就能得到正确结果,但是在概念上却是兜了一个圈子,不那么直截了当。



第 6 章 指针

在 Pascal 语言中有“值形参”和“变量形参”(即 var 形参),对应两种不同的传递方式,前者采用值传递方式,后者采用地址传递方式。在 C 语言中,只有“值形参”而无“变量形参”,全部采用值传递方式。C++把引用型变量作为函数形参,就弥补了这个不足。

C++提供了向函数传递数据的第(3)种方法,即传送变量的别名。

例 6.20 利用“引用形参”实现两个变量的值互换。

```
#include <iostream>
using namespace std;
int main( )
{ void swap(int &,int &);
int i=3,j=5;
第 6 章 指针
swap(i,j);
cout<<" i=" <<i<<"    " <<" j=" <<j<<endl;
return 0;
}

void swap(int &a,int &b)           //形参是引用类型
{ int temp;
temp=a;
a=b;
b=temp;
}
```

输出结果为

i=5 j=3

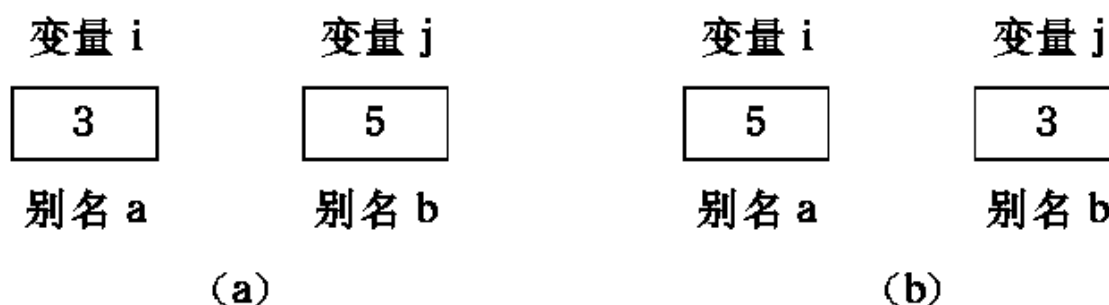
在 swap 函数的形参表列中声明 a 和 b 是整型变量的引用。

第 6 章 指针

图 6.29

实际上,在虚实结合时是把实参 *i* 的地址传到形参 *a*,使形参 *a* 的地址取实参 *i* 的地址,从而使 *a* 和 *i* 共享同一单元。同样,将实参 *j* 的地址传到形参 *b*,使形参 *b* 的地址取实参 *j* 的地址,从而使 *b* 和 *j* 共享同一单元。这就是地址传递方式。为便于理解,可以通俗地说:把变量 *i* 的名字传给引用变量 *a*,使 *a* 成为 *i* 的别名。

请思考:这种传递方式和使用指针变量作形参时有何不同?分析例 6.20(对比例 6.19),可以发现:



## 第 6 章 指针

① 使用引用类型就不必在 `swap` 函数中声明形参是指针变量。指针变量要另外开辟内存单元,其内容是地址。而引用变量不是一个独立的变量,不单独占内存单元,在例 6.20 中引用变量 *a* 和 *b* 的值的数据类型与实参相同,都是整型。

② 在 `main` 函数中调用 `swap` 函数时,实参不必用变量的地址(在变量名的前面加`&`),而直接用变量名。系统向形参传送的是实参的地址而不是实参的值。

这种传递方式相当于 Pascal 语言中的“变量形参”,显然,这种用法比使用指针变量简单、直观、方便。使用变量的引用,可以部分代替指针的操作。有些过去只能用指针来处理的问题,现在可以用引用来代替,从而降低了程序设计的难度。

## 第 6 章 指针

例 6.21 对 3 个变量按由小到大的顺序排序。

```
#include <iostream>
using namespace std;
int main( )
{ void sort(int &,int &,int &); //函数声明,形参是引用类型
  int a,b,c; //a,b,c 是需排序的变量
  int a1,b1,c1; //a1,b1,c1 最终的值是已排好序的数列
  cout<<" Please enter 3 integers: " ;
  cin>>a>>b>>c; //输入 a,b,c
  a1=a;b1=b;c1=c;
  sort(a1,b1,c1); //调用 sort 函数,以 a1,b1,c1 为实参
  cout<<" sorted order is " <<a1<<" " <<b1<<" " <<c1<<endl; //此时 a1,b1,c1 已排好序
  return 0;
}
void sort(int &i,int &j,int &k) //对 i,j,k 3 个数排序
{ void change(int &,int &); //函数声明,形参是引用类型
  if (i>j) change (i,j); //使 i<=j
  if (i>k) change (i,k); //使 i<=k
  if (j>k) change (j,k); //使 j<=k
}
void change (int &x,int &y) //使 x 和 y 互换
{ int temp;
```

```
temp=x;
x=y;
y=temp;
}
```

运行情况如下:

Please enter 3 integers: 23 12 -345✓

sorted order is -345 12 23

## 第 6 章 指针

可以看到: 这个程序很容易理解,不易出错。由于在调用 `sort` 函数时虚实结合使形参 `i,j,k` 成为实参 `a1,b1,c1` 的引用,因此通过调用函数 `sort(a1,b1,c1)` 既实现了对 `i,j,k` 排序,也就同时实现了对 `a1,b1,c1` 排序。同样,执行 `change(i,j)` 函数,可以实现对实参 `i` 和 `j` 的互换。

引用不仅可以用于变量,也可以用于对象。例如实参可以是一个对象名,在虚实结合时传递对象的起始地址。这会在以后介绍。

当看到 `&a` 这样的形式时,怎样区别是声明引用变量还是取地址的操作呢?当 `&a` 的前面有类型符时(如 `int &a`),它必然是对引用的声明;如果前面无类型符(如 `cout<<&a`),则是取变量的地址。

# 第 7 章 自定义数据类型

## 7.1 结构体类型

## 7.2 共用体

## 7.3 枚举类型

## 7.4 用 `typedef` 声明类型

## 7.1 结构体类型

C++提供了许多种基本的数据类型(如 `int`、`float`、`double`、`char` 等)供用户使用。但是由于程序需要处理的问题往往比较复杂,而且呈多样化,已有的数据类型显得不能满足使用要求。因此 C++允许用户根据需要自己声明一些类型,例如第 5 章介绍的数组就是用户自己声明的数据类型。此外,用户可以自己声明的类型还有结构体(structure)类型、共用体(union)类型、枚举(enum)类型、类(class)类型等,这些统称为用户自定义类型(user-defined type,UDT)。本章介绍结构体类型、共用体类型和枚举类型,第 8 章将介绍类类型。

**7.1 结构体类型** **7.1.1 结构体概述**有时需要将不同类型的数据组合成一个有机的整体,以供用户方便地使用。这些组合在一个整体中的数据是互相联系的。例如,一个学生的学号、姓名、性别、年龄、成绩、家庭地址等项,都是这个学生的属性。见图 7.1。

图 7.1

在一个组合项中包含若干个类型不同(当然也可以相同)的数据项。C 和 C++允许用户自己指定这样一种数据类型,它称为结构体。它相当于其他高级语言中的记录(record)。例如,可以通过下面的声明来建立如图 7.1 所示的数据类型。

```
struct Student//声明一个结构体类型 Student
{
    int num;//包括一个整型变量 num
    char name[20];//包括一个字符数组 name,可以容纳 20 个字符
    char sex;//包括一个字符变量 sex
    int age;//包括一个整型变量 age
    float score;//包括一个单精度型变量
    char addr[30];//包括一个字符数组 addr,可以容纳 30 个字符
};//最后有一个分号
```

这样,程序设计者就声明了一个新的结构体类型 `Student`(`struct` 是声明结构体类型时所必须使用的关键字,不能省略),它向编译系统声明:这是一种结构体类型,它包括 `num`, `name`, `sex`, `age`, `score`, `addr` 等不同类型的数据项。应当说明 `Student` 是一个类型名,它和系统提供的标准类型(如 `int`、`char`、`float`、`double` 一样,都可以用来定义变量,只不过结构体类型需要事先由用户自己声明而已。

声明一个结构体类型的一般形式为

```
struct 结构体类型名 {成员表列};
```

结构体类型名用来作结构体类型的标志。上面的声明中 `Student` 就是结构体类型名。大括号内是该结构体中的全部成员(member),由它们组成一个特定的结构体。上例中的 `num`, `name`, `sex`, `score` 等都是结构体中的成员。在声明一个结构体类型时必须对各成员都进行类型声明即类型名 成员名;每一个成员也称为结构体中的一个域(field)。成员表列又称为域表。

成员名的定名规则与变量名的定名规则相

声明结构体类型的位置一般在文件的开头,在所有函数(包括 `main` 函数)之前,以便本文件中所有的函数都能利用它来定义变量。当然也可以在函数中声明结构体类型。

在 C 语言中,结构体的成员只能是数据(如上面例子中所表示的那样)。

C++对此加以扩充,结构体的成员既可以包括数据(即数据成员),又可以包括函数(即函数成员),以适应面向对象的程序设计。

但是由于 C++提供了类(class)类型,一般情况下,不必使用带函数的结构体,因此在本章中只介绍只含数据成员的结构体,有关包含函数成员的结构体将在第 8 章介绍类类型时一并介绍。

### 7.1.2 结构体类型变量的定义方法及其初始化

前面只是指定了一种结构体类型，它相当于一个模型，但其中并无具体数据，系统也不为之分配实际的内存单元为了能在程序中使用结构体类型的数据，应当定义结构体类型的变量，并在其中存放具体的数据。

1. 定义结构体类型变量的方法可以采取以下 3 种方法定义结构体类型的变量。

(1) 先声明结构体类型再定义变量名如上面已定义了一个结构体类型 `Student`，可以用它来定义结构体变量。如

```
Student student1, student2;
```

以上定义了 `student1` 和 `student2` 为结构体类型 `Student` 的变量，即它们具有 `Student` 类型的结构。如图 7.2 所示。图 7.2 在定义了结构体变量后，系统会为之分配内存单元。例如 `student1` 和 `student2` 在内存中各占 63 个字节

( $4+20+1+4+4+30=63$ )。

(2) 在声明类型的同时定义变量

例如：

```
struct Student//声明结构体类型 Student
{
int num;
char name[20];
char sex;
int age;
float score;
char addr[30];
} student1,student2;//定义两个结构体类型 Student 的变量 student1,student2
```

这种形式的定义的一般形式为

```
struct 结构体名
{
成员表列
}变量名表列;
```

(3) 直接定义结构体类型变量

其一般形式为

```
struct//注意没有结构体类型名
{
成员表列
} 变量名表列;
```

这种方法虽然合法，但很少使用。提倡先定义类型后定义变量的第(1)种方法。

在程序比较简单，结构体类型只在本文件中使用的情况下，也可以用第(2)种方法。

关于结构体类型，有几点要说明：

(1) 不要误认为凡是结构体类型都有相同的结构。实际上，每一种结构体类型都有自己的结构，可以定义出许多种具体的结构体类型。

(2) 类型与变量是不同的概念，不要混淆。只能对结构体变量中的成员赋值，而不能对结构体类型赋值。

在编译时，是不会为类型分配空间的，只为变量分配空间。

(3) 对结构体中的成员（即“域”），可以单独使用，它的作用与地位相当于普通变量。关于对成员的引用方法见 7.3 节。

(4) 成员也可以是一个结构体变量。

如

```
struct Date //声明一个结构体类型 Date
{
int month;
int day;
int year;
};
```

```
struct Student
//声明一个结构体类型 Student
{
int num;
char name[20];
char sex;
int age;
Date birthday;
char addr[30];
}student1,student2;
```

//定义 student1 和 student2 为结构体类型 Student 的变量 Student 的结构见图 7.3 所示。图 7.3

(5) 结构体中的成员名可以与程序中的变量名相同,但二者没有关系。



例如,程序中可以另定义一个整型变量 num,它与 student 中的 num 是两回事,互不影响。

## 2. 结构体变量的初始化

和其他类型变量一样,对结构体变量可以在定义时指定初始值。如

```
struct Student
{
    int num;
    char name[20];
    char sex;
    int age;
    float score;
    char addr[30];
} student1={10001,"Zhang Xin",'M',19,90.5,"Shanghai"};
```

这样,变量 student1 中的数据如图 7.2 中所示。也可以采取声明类型与定义变量分开的形式,在定义变量时进行初始化:

```
student2= student1;
```

### 7.1.3 结构体变量的引用

在定义了结构体变量以后,当然可以引用这个变量。

(1) 可以将一个结构体变量的值赋给另一个具有相同结构的结构体变量。

如上面的 student1 和 student2 都是 student 类型的变量,可以这样赋值:

```
student1= student2;
```

(2) 可以引用一个结构体变量中的一个成员的值。

例如, student1.num 表示结构体变量 student1 中的成员的值,如果 student1 的值如图 7.2 所示,

则 student1.num 的值为 10001。

引用结构体变量中成员的一般方式为

结构体变量名.成员名

例如可以这样对变量的成员赋值:

```
student1.num=10010;
```

(3) 如果成员本身也是一个结构体类型,则要用若干个成员运算符,一级一级地找到最低一级的成员。

例如,对上面定义的结构体变量 `student1`, 可以这样访问各成员:

`student1.num` (引用结构体变量 `student1` 中的 `num` 成员)

如果想引用 `student1` 变量中的 `birthday` 成员中的 `month` 成员, 不能写成 `student1.month`, 必须逐级引用, 即

```
student1.birthday.month=12;
```

(引用结构体变量 `student1` 中的 `birthday` 成员中的 `month` 成员)

(4) 不能将一个结构体变量作为一个整体进行输入和输出。

例如,已定义 `student1` 和 `student2` 为结构体变量,并且它们已有值。不能企图这样输出结构体变量中的各成员的值:

```
cin>>student1;
```

只能对结构体变量中的各个成员分别进行输入和输出。

(5) 对结构体变量的成员可以像普通变量一样进行各种运算(根据其类型决定可以进行的运算种类)。例如

```
student2.score=student1.score;
```

```
sum=student1.score+student2.score;
```

```
student1.age++;
```

```
++student1.age;
```

由于“.”运算符的优先级最高, `student1.age++` 相当于 `(student1.age)++`。++是对 `student1.age` 进行自加运算, 而不是先对 `age` 进行自加运算。

(6) 可以引用结构体变量成员的地址, 也可以引用结构体变量的地址。如

```
cout<<&student1;//输出 student1 的首地址
```

```
cout<<&student1.age;//输出 student1.age 的地址
```

结构体变量的地址主要用作函数参数, 将结构体变量的地址传递给形参。

例 7.1 引用结构体变量中的成员。

```
#include <iostream>
```

```
using namespace std;
```

```

struct Date//声明结构体类型 Date
{int month;
int day;
int year;
};
struct Student//声明结构体类型 Student
{int num;
char name[20];
char sex;
Date birthday;//声明 birthday 为 Date 类型的成员
float score;
}student1,student2={10002,"Wang Li",'f',5,23,1982,89.5};
//定义 Student 类型的变量 student1,student2, 并对 student2 初始化
int main()
{ student1=student2;//将 student2 各成员的值赋予 student1 的相应成员
cout<<student1.num<<endl;//输出 student1 中的 num 成员的值
cout<<student1.name<<endl;//输出 student1 中的 name 成员的值

cout<<student1.sex<<endl;// 输 出 student1 中 的 sex 成 员 的 值
cout<<student1.birthday.month<<"/"<<student1.birthday.day<<"/" <<student1.birthday.year<<endl;//输出
student1 中的 birthday 各成员的值
cout<<student1.score<<endl;
return 0; }

```

运行结果如下：

```
10002 Wang Li f 5/23/1982 89.5 7.1.4
```

#### 7.1.4 结构体数组

一个结构体变量中可以存放一组数据（如一个学生的学号、姓名、成绩等数据）。如果有 10 个学生的数据需要参加运算，显然应该用数组，这就是结构体数组。结构体数组与以前介绍过的数值型数组的不同之处在于：

每个数组元素都是一个结构体类型的数据，它们都分别包括各个成员项。

1. 定义结构体数组和定义结构体变量的方法相仿，定义结构体数组时只需声明其为数组即可。如

```
struct Student//声明结构体类型 Student
```

```

{ int num;
char name[20];
char sex;
int age;
float score;
char addr[30];
};

```

```
Student stu[3];//定义 Student 类型的数组 stu
```

也可以直接定义一个结构体数组，如

```
struct Student
{ int num;
  char name[20];
  char sex;
  int age;
  float score;
  char addr[30];
}stu[3];
```

或

```
struct
{ int num;
  char name[20];
  char sex;
  int age;
  float score;
  char addr[30];
}stu[3];
```

见图 7.4。数组各元素在内存中连续存放，见图 7.5 示意。

图 7.4 图 7.5 2.

结构体数组的初始化与其他类型的数组一样，对结构体数组可以初始化。如

```
struct Student
{
  int num;
  char name[20];
  char sex;
  int age;
  float score;
  char addr[30];
}stu[3]={
{10101,"Li Lin", 'M', 18,87.5, "103 Beijing Road"},
{10102,"Zhang Fun",'M', 19,99, "130 Shanghai Road"},
{10104,"Wang Min",'F', 20,78.5, "1010 Zhongshan Road"}};
```

定义数组 stu 时，也可以不指定元素个数，即写成以下形式：

```
stu[]={ {...}, {...}, {...} };
```

编译时，系统会根据给出初值的结构体常量的个数来确定数组元素的个数。

一个结构体常量应包括结构体中全部成员的值。当然，数组的初始化也可以用以下形式：

```
Student stu[] = {{...},{...},{...}}; //已事先声明了结构体类型 Student
```

由上可以看到，结构体数组初始化的一般形式是在所定义的数组名的后面加上 = {初值表};

### 3. 结构体数组应用举例

下面举一个简单的例子来说明结构体数组的定义和引用。

例 7.2 对候选人得票的统计程序。设有 3 个候选人，最终只能有 1 人当选为领导。今有 10 个人参加投票，从键盘先后输入这 10 个人所投的候选人的名字，要求最后输出这 3 个候选人的得票结果。可以定义一个候选人结构体数组，包括 3 个元素，在每个元素中存放有关的数据。

程序如下：

```
#include <iostream>

struct Person //声明结构体类型 Person
{
    char name[20];
    int count;
};

int main()
{
    Person leader[3]={"Li",0,"Zhang",0,"Fun",0};

    //定义 Person 类型的数组，内容为 3 个候选人的姓名和当前的得票数
    int i,j;
    char leader_name[20];
    //leader_name 为投票人所选的人的姓名
    for(i=0;i<10;i++) {cin>>leader_name;
    //先后输入 10 张票上所写的姓名
    for(j=0;j<3;j++)//将票上姓名与 3 个候选人的姓名比较
    if(strcmp(leader_name,leader[j].name)==0) leader[j].count++;
    //如果与某一候选人的姓名相同，就给他加一票
    }
    cout<<endl;

    for(i=0;i<3;i++)//输出 3 个候选人的姓名与最后得票数
    {cout<<leader[i].name<<": "<<leader[i].count<<endl;}

    return 0;
}

运行情况如下：
Zhang✓ (每次输入一个候选人的姓名)
Li✓
Fun✓
Li✓
```

Zhang✓

Li✓

Zhang✓

Li✓

Fun✓

Wang✓

Li:4 (输出 3 个候选人的姓名与最后得票数)

Zhang:3

Fun:2

程序定义一个全局的结构体数组 leader,它有 3 个元素,每一元素包含两个成员,即 name(姓名)和 count(得票数)。在定义数组时使之初始化,使 3 位候选人的票数都先置零。见图 7.6。

图 7.6

在这个例子中,也可以不用字符数组而用 string 方法的字符串变量来存放姓名数据,程序可修改如下:

```
#include <iostream>
```

```
#include <string>
```

```
using namespace std;
```

```
struct Person
```

```
{
```

```
string name;//成员 name 为字符串变量
```

```
int count;
```

```
};
```

```
int main()
```

```
{
```

```
Person leader [3] ={"Li",0,"Zhang",0,"Fun",0};
```

```
int i,j;
```

```
string leader_name;// leader_name 为字符串变量
```

```
for(i=0;i<10;i++)
```

```
{
```

```
cin>>leader_name;
```

```
for(j=0;j<3;j++)
```

```
if(leader_name==leader[j].name) leader[j].count++
```

```
//用“==”进行比较
```

```
}
```

```
cout<<endl;
```

```
for(i=0;i<3;i++)
```

```
{cout<<leader[i].name<<": "<<leader[i].count<<endl;}
```

```
return 0;
```

```
}
```

### 7.1.5 指向结构体变量的指针

一个结构体变量的指针就是该变量所占据的内存段的起始地址。可以设一个指针变量,用来指向一个结构

体变量，此时该指针变量的值是结构体变量的起始地址。

指针变量也可以用来指向结构体数组中的元素。

### 1. 通过指向结构体变量的指针引用结构体变量中的成员

下面通过一个简单例子来说明指向结构体变量的指针变量的应用。

#### 例 7.3 指向结构体变量的指针的应用。

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    struct Student
    //声明结构体类型 student
    { int num;
      string name;
      char sex;
      float score;
    };
    Student stu;
    //定义 Student 类型的变量 stu
    Student *p=&stu;
    //定义 p 为指向 Student 类型数据的指针变量并指向 stu
    stu.num=10301;
    //对 stu 中的成员赋值
    stu.name="Wang Fun";
    //对 string 变量可以直接赋值
    stu.sex='f';
    stu.score=89.5;

    cout<<stu.num<<" "<<stu.name<<" "<<stu.sex<<" "<<
    stu.score<<endl;

    cout<<p->num<<" "<<(*p).name<<" "<<(*p).sex<<" "<<
    (*p).score<<endl;
```

```
return 0;
}
```

图 7.7

程序运行结果如下：

10301 Wang Fun f 89.5 (通过结构体变量名引用成员)

10301 Wang Fun f 89.5 (通过指针引用结构体变量中的成员)

两个 cout 语句输出的结果是相同的。

为了使用方便和使之直观，C++提供了指向结构体变量的运算符->，

例如 p->num 表示指针 p 当前指向的结构体变量中的成员 num。

p->num 和(\*p).num 等价。

同样，p->name 等价于(\*p).name。

也就是说，以下 3 种形式等价：

- ① 结构体变量.成员名。如 stu.num。
- ② (\*p).成员名。如(\*p).num。
- ③ p->成员名。如 p->num。

“->”称为指向运算符。

请分析以下几种运算：

p->n 得到 p 指向的结构体变量中的成员 n 的值。

p->n++ 得到 p 指向的结构体变量中的成员 n 的值，用完该值后使它加 1。

++p->n 得到 p 指向的结构体变量中的成员 n 的值，并使之加 1，然后再使用它。

## 2. 用结构体变量和指向结构体变量的指针构成链表

链表是一种常见的重要的数据结构。

图 7.8 表示最简单的一种链表（单向链表）的结构。

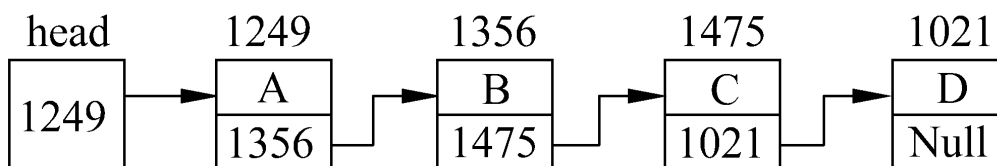


图 7.8



链表有一个“头指针”变量，图中以 head 表示，它存放一个地址。该地址指向一个元素。链表中的每一个元素称为“结点”，每个结点都应包括两个部分：

- 一是用户需要用的实际数据，
- 二是下一个结点的地址。

可以看到链表中各元素在内存中的存储单元可以是不连续的。要找某一元素，可以先找到上一个元素，根据它提供的下一元素地址找到下一个元素。

可以看到，这种链表的数据结构，必须利用结构体变量和指针才能实现。

可以声明一个结构体类型，包含两种成员，一种是需要用的实际数据，另一种是用来存放下一结点地址的指针变量。

例如，可以设计这样一个结构体类型：

```
struct Student
{
    int num;
    float score;
    Student *next; //next 指向 Student 结构体变量
};
```

其中成员 num 和 score 是用户需要用到的数据，相当于图 7.8 结点中的 A,B,C,D。next 是指针类型的成员，它指向 Student 类型数据（就是 next 所在的结构体类型）。用这种方法就可以建立链表。见图 7.9。

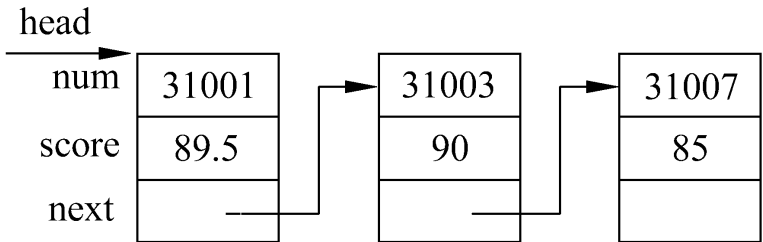


图 7.9

图中每一个结点都属于 Student 类型，在它的成员 next 中存放下一个结点的地址，程序设计者不必知道各结点的具体地址，只要保证能将下一个结点的地址放到前一结点的成员 next 中即可。

下面通过一个例子来说明如何建立和输出一个简单链表。

例 7.4 建立一个如图 7.9 所示的简单链表，它由 3 个学生数据的结点组成。输出各结点中的数据。

```
#define NULL 0
#include <iostream>
struct Student
{
    long num;
    float score;
    struct Student *next;
};

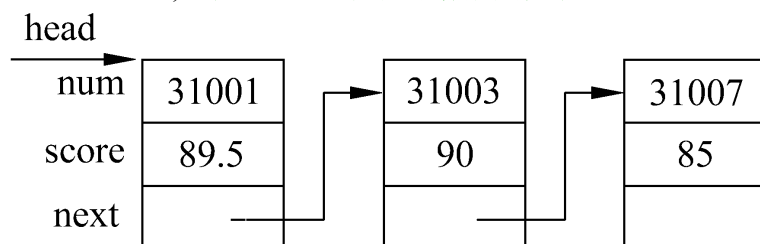
int main()
```

```

{
Student a,b,c,*head,*p;
a. num=31001;
a.score=89.5;//对结点 a 的 num 和 score 成员赋值
b. num=31003;
b.score=90;//对结点 b 的 num 和 score 成员赋值
c. num=31007;
c.score=85;//对结点 c 的 num 和 score 成员赋值

head=&a;//将结点 a 的起始地址赋给头指针 head
a.next=&b; //将结点 b 的起始地址赋给 a 结点的 next 成员
b.next=&c; //将结点 c 的起始地址赋给 b 结点的 next 成员
c.next=NULL; //结点的 next 成员不存放其他结点地址

```



```

p=head; //使 p 指针指向 a 结点
do
{
cout<<p->num<<" "<<p->score<<endl; //输出 p 指向的结点的数据
p=p->next; //使 p 指向下一个结点
} while (p!=NULL); //输出完 c 结点后 p 的值为 NULL
return 0;
}

```

请读者考虑：

- ①各个结点是怎样构成链表的。
- ②p 起什么作用？

本例是比较简单的，所有结点(结构体变量)都是在程序中定义的，不是临时开辟的，也不能用完后释放，这种链表称为静态链表。对各结点既可以通过上一个结点的 next 指针去访问，也可以直接通过结构体变量名 a,b,c 去访问。

动态链表则是指各结点是可以随时插入和删除的，这些结点并没有变量名，只能先找到上一个结点，才能根据它提供的下一结点的地址找到下一个结点。只有提供第一个结点的地址，即头指针 head，才能访问整个链表。如同一条铁链一样，一环扣一环，中间是不能断开的。

建立动态链表，要用到下面 7.1.7 小节介绍的动态分配内存的运算符 new 和动态撤销内存的运算符 delete。

### 7.1.6 结构体类型数据作为函数参数

将一个结构体变量中的数据传递给另一个函数，有下列 3 种方法：

- (1) 用结构体变量名作参数。一般较少用这种方法。
- (2) 用指向结构体变量的指针作实参，将结构体变量的地址传给形参。
- (3) 用结构体变量的引用变量作函数参数。

下面通过一个简单的例子来说明，并对它们进行比较。

例 7.5 有一个结构体变量 stu，内含学生学号、姓名和 3 门课的成绩。要求在 main 函数中为各成员赋值，在另一函数 print 中将它们的值输出。

(1) 用结构体变量作函数参数

```
#include <iostream>
#include <string>
using namespace std;
struct Student//声明结构体类型 Student
{ int num;
  char name[20];
  float score[3];
};
int main()
{
  void print(Student);//函数声明，形参类型为结构体 Student
  Student stu;//定义结构体变量
```

```
  stu.num=12345;//以下 5 行对结构体变量各成员赋值
  stu.name="Li Fung";
  stu.score[0]=67.5;
  stu.score[1]=89;
  stu.score[2]=78.5;
  print(stu);//调用 print 函数，输出 stu 各成员的值
  return 0;
}
```

```
void print(Student st)
{
  cout<<st.num<<" "<<st.name<<" "<<st.score[0]
  <<" "<<st.score[1]<<" "<<st.score[2]<<endl;
}
```

运行结果为

12345 Li Fung 67.5 89 78.5 (2)

用指向结构体变量的指针作实参在上面程序的基础上稍作修改即可。

```
#include <iostream>
#include <string>
using namespace std;
struct Student
{
  int num; string name;//用 string 类型定义字符串变量
  float score[3];
}stu={12345,"Li Fung",67.5,89,78.5};//定义结构体 student 变量 stu 并赋初值
int main()
```

```

{
void print(Student *); //函数声明，形参为指向 Student 类型数据的指针变量
Student *pt=&stu; //定义基类型为 Student 的指针变量 pt，并指向 stu
print(pt); //实参为指向 Student 类数据的指针变量
return 0;
}

```

//定义函数，形参 p 是基类型为 Student 的指针变量

```

void print(Student *p)
{
cout<<p->num<<" "<<p->name<<" "<<p->score[0]<<" " <<
p->score[1]<<" "<<p->score[2]<<endl;
}

```

调用 print 函数时，实参指针变量 pt 将 stu 的起始地址传送给形参 p (p 也是基类型为 student 的指针变量)。这样形参 p 也就指向 stu，见图 7.10。

在 print 函数中输出 p 所指向的结构体变量的各个成员值，它们也就是 stu 的成员值。在 main 函数中也可以不定义指针变量 pt，而在调用 print 函数时以 &stu 作为实参，把 stu 的起始地址传给实参 p。

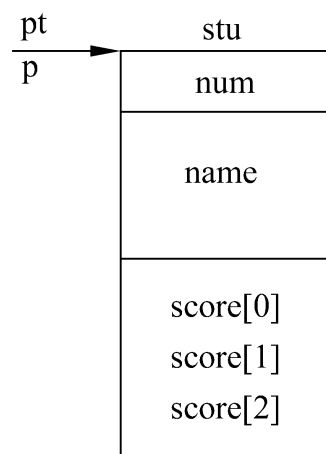


图 7.10

(3) 用结构体变量的引用作函数参数

```

#include <iostream>
#include <string>
using namespace std;
struct Student
{
int num;
string name;
float score[3];
}stu={12345,"Li Li",67.5,89,78.5};

void main()
{
void print(Student &);
//函数声明，形参为 Student 类型变量的引用
print(stu);
//实参为结构体 Student 变量

```

```
}
```

//函数定义，形参为结构体 Student 变量的引用

```
void print(Student &stud)
{
    cout<<stud.num<<" "<<stud.name<<" "<<stud.score[0]
    <<" "<<stud.score[1]<<" "<<stud.score[2]<<endl;
}
```

程序(1)用结构体变量作实参和形参，程序直观易懂，效率是不高的。

程序(2)采用指针变量作为实参和形参，空间和时间的开销都很小，效率较高。但程序(2)不如程序(1)那样直接。

程序(3)的实参是结构体 Student 类型变量，而形参用 Student 类型的引用，虚实结合时传递的是 stu 的地址，因而效率较高。它兼有(1)和(2)的优点。

引用变量主要用作函数参数，它可以提高效率，而且保持程序良好的可读性。在本例中用了 string 方法定义字符串变量，在某些 C++系统中目前不能运行这些程序，读者可以修改程序，使之能在自己所用的系统中运行。

### \*7.1.7 动态分配和撤销内存的运算符 new 和 delete

在软件开发过程中，常常需要动态地分配和撤销内存空间，例如对动态链表中结点的插入与删除。

在 C 语言中是利用库函数 malloc 和 free 来分配和撤销内存空间的。

#### sizeof

C++提供了较简便而功能较强的运算符 new 和 delete 来取代 malloc 和 free 函数。

注意：new 和 delete 是运算符，不是函数，因此执行效率高。

虽然为了与 C 语言兼容，C++仍保留 malloc 和 free 函数，但建议用户不用 malloc 和 free 函数，而用 new 和 delete 运算符。

new 运算符的例子：

```
new int; //开辟一个存放整数的存储空间，返回一个指向该存储空间的地址(即指针)
new int(100); //开辟一个存放整数的空间，并指定该整数的初值为 100，返回一个指向该存储空间的地址
new char[10]; //开辟一个存放字符数组(包括 10 个元素)的空间，返回首元素的地址
new int[5][4]; //开辟一个存放二维整型数组(大小为 5*4)的空间，返回首元素的地址
float *p=new float(3.14159); //开辟一个存放单精度数的空间，并指定该实数的初值为 3.14159，将返回的该空间的地址赋给指针变量 p
```

new 运算符使用的一般格式为

new 类型 [初值]

用 new 分配数组空间时不能指定初值。如果由于内存不足等原因而无法分配空间，则 new 会返回一个空指针 NULL，用户可以根据该指针的值判断分配空间是否成功。

delete 运算符使用的一般格式为

`delete []` 指针变量

例如要撤销上面用 `new` 开辟的存放单精度数的空间(上面第 5 个例子), 应该用

`delete p;`

前面用“`new char[10];`”开辟的字符数组空间, 如果把 `new` 返回的指针赋给了指针变量 `pt`, 则应该用以下形式的 `delete` 运算符撤销该空间:

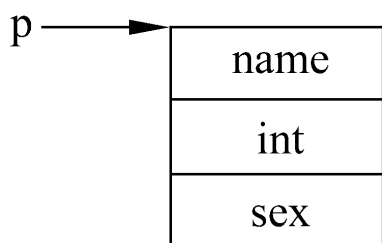
`delete [] pt;` //在指针变量前面加一对方括号, 表示是对数组空间的操作

例 7.6 开辟空间以存放一个结构体变量。

```
#include <iostream>
#include <string>
using namespace std;
struct Student           //声明结构体类型 Student
{ string name;
  int num;
  char sex;
};
int main( )
{ Student *p; //定义指向结构体类型 Student 的数据的指针变量
  p=new Student; //用 new 运算符开辟一个存放 Student 型数据的空间
  p->name=" Wang Fun" ;    //向结构体变量的成员赋值
  p->num=10123;
  p->sex='m';
  cout<<p->name<<endl<<p->num
  <<endl<<p->sex<<endl; //输出各成员的值
  delete p; //撤销该空间
  return 0;
}
```

运行结果为

Wang Fun 10123 m



用 new student  
开辟的空间

图 7.11

在动态分配/撤销空间时, 往往将这两个运算符和结构体结合使用, 是很有效的。可以看到:

要访问用 new 所开辟的结构体空间，无法直接通过变量名进行，只能通过指针 p 进行访问。如果要建立一个动态链表，必须从第一个结点开始，逐个地开辟结点并输入各结点数据，通过指针建立起前后相链的关系。

7.2 共用体

7.2.1 共用体的概念

有时需要使几种不同类型的变量存放到同一段内存单元中。例如，可把一个整型变量、一个字符型变量、一个双精度型变量放在同一个地址开始的内存单元中（见图 7.12）。

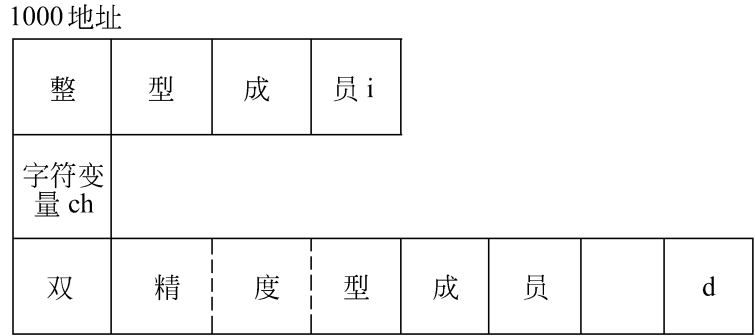


图 7.12 以上 3 个变量在内存中占的字节数不同，但都从同一地址开始存放。也就是使用覆盖技术，几个变量互相覆盖。这种使几个不同的变量共占同一段内存的结构，称为共用体(union)类型的结构(有些书译为联合)。

声明共用体类型的一般形式为  
union 共用体类型名 {成员表列};

定义共用体变量的一般形式为  
共用体类型名 共用体变量名;

当然也可在声明共用体类型的同时定义共用体变量，也可没有共用体类型名而直接定义共用体变量。例如可以看到，“共用体”与“结构体”的定义形式相似。但它们的含义是不同的。结构体变量所占内存长度是各成员占的内存长度之和。每个成员分别占有其自己的内存单元。共用体变量所占的内存长度等于最长的成员的长度。

7.2.2 对共用体变量的访问方式

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，下面的引用方式是正确的：

- a.i （引用共用体变量中的整型成员 i）
- a.ch（引用共用体变量中的字符型成员 ch）
- a.f （引用共用体变量中的双精度型成员 d）

不能只引用共用体变量，例如

cout<<a; 是错误的，  
应该写成  
cout<<a.i;或 cout<<a.ch;等。

7.2.3 共用体类型数据的特点

(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。但请注意：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它初始化;不能用共用体变量名作为函数参数。

例 7.7 设有若干个人的数据,其中有学生和教师。学生的数据中包括: 姓名、号码、性别、职业、年级。教师的数据包括: 姓名、号码、性别、职业、职务。可以看出,学生和教师所包含的数据是不同的。现要求把它们放在同一表格中,见图 7.13。

图 7.13

如果 job 项为 s (学生),则第 5 项为 grade (年级)。即 Li 是 3 年级的。如果 job 项是 t (教师),则第 5 项为 position (职务)。**Wang 是 prof(教授)**。显然对第 5 项可以用共用体来处理(将 class 和 position 放在同一段内存中)。要求输入人员的数据,然后再输出。为简化起见,只设两个人(一个学生、一个教师)。程序如下:

```
#include <iostream>
#include <string>
#include <iomanip> //因为在输出流中使用了控制符 setw
using namespace std;
struct
{
    int num;
    char name[10];
    char sex;
    char job;
    union P //声明共用体类型
    {
        int grade; //年级
        char position[10]; //职务
    } category; //成员 category 为共用体变量
    } person[2]; //定义共用体数组 person, 含两个元素
int main( )
{
    int i;
    for(i=0;i<2;i++) //输入两个学生的数据
    {
        cin>>person[i].num>>person[i].name
        >>person[i].sex>>person[i].job;
        if(person[i].job=='s')
            cin>>person[i].category.grade; //若是学生则输入年级
        else if (person[i].job=='t')
            cin>>person[i].category.position; //若是教师则输入职务
    }
    cout<<endl<<"No. Name sex job grade/position"<<endl;

    for(i=0;i<2;i++)
    {
        if (person[i].job=='s') cout<<person[i].num<<setw(6)
        <<person[i].name<<" "<<person[i].sex <<" "
        <<person[i].job<<setw(10)<<person[i].category.grade<<endl;
        else cout<<person[i].num<<setw(6)<<person[i].name
```



```
<<" "<<person[i].sex <<" "<<person[i].job
<<setw(10)<<person[i].category.position<<endl;
}
return 0;
}
```

运行情况如下：

**101** Li fs **3**✓ (注意在输入的字母 f 和 s 之间无空格)

**102** Wang mt prof✓ (注意在输入的字母 m 和 t 之间无空格)

No. Name sex job grade/position **101** Li f s **3** **102** Wang m t prof

为了使输出结果上下对齐，在 cout 语句中用了 setw 控制符和插入空格。往往需要试验多次。

### 7.3 枚举类型

如果一个变量只有几种可能的值，**可以定义为枚举(enumeration)类型**。所谓“枚举”是指将变量的值一一列举出来，变量的值只能在列举出来的值的范围内。声明枚举类型用 enum 开头。

例如

```
enum weekday{sun, mon, tue, wed, thu, fri, sat};
```

上面声明了一个枚举类型 weekday，花括号中 sun,mon,...,sat 等称为枚举元素或枚举常量。

表示这个类型的变量的值只能是以上 7 个值之一。它们是用户自己定义的标识符。

声明枚举类型的一般形式为

```
enum 枚举类型名 {枚举常量表列};
```

在声明了枚举类型之后，可以用它来定义变量。

如

```
weekday workday,week_end;
```

这样，workday 和 week\_end 被定义为枚举类型 weekday 的变量。

在 C 语言中，枚举类型名包括关键字 enum，以上的定义可以写为

```
enum weekday workday,week_end;
```

在 C++中允许不写 enum，一般也不写 enum，但保留了 C 的用法。根据以上对枚举类型 weekday 的声明，枚举变量的值只能是 sun 到 sat 之一。

例如

```
workday=mon; week_end=sun; 是正确的。
```

也可以直接定义枚举变量，如

```
enum{sun, mon, tue, wed, thu, fri, sat} workday,week_end;
```

这些标识符并不自动地代表什么含义。

**说明：**

(1) 对枚举元素按常量处理，故称枚举常量。

(2) 枚举元素作为常量，它们是有值的，C++编译按定义时的顺序对它们赋值为 0,1,2,3,...。也可以在声明枚举类型时另行指定枚举元素的值。

(3) 枚举值可以用来做判断比较。

(4) 一个整数不能直接赋给一个枚举变量。

**例 7.8** 口袋中有红、黄、蓝、白、黑 5 种颜色的球若干个。每次从口袋中任意取出 3 个球，问得到 3 种不同颜色的球的可能取法，输出每种排列的情况。

```
#include <iostream>
```

```
#include <iomanip> //在输出时要用到 setw 控制符
```

```
using namespace std;
```

```

int main()
{ enum color {red,yellow,blue,white,black}; //声明枚举类型 color
  color pri; //定义 color 类型的变量 pri
  int i,j,k,n=0,loop; //n 是累计不同颜色的组合数
  for (i=red;i<=black;i++) //当 i 为某一颜色时
  for (j=red;j<=black;j++) //当 j 为某一颜色时
  if (i!=j) //若前两个球的颜色不同
  { for (k=red;k<=black;k++) //只有前两个球的颜色不同，才需要检查第 3 个球的颜色
    if ((k!=i) && (k!=j)) //3 个球的颜色都不同
    {n=n+1; //使累计值 n 加 1
     cout<<setw(3)<<n; //输出当前的 n 值，字段宽度为 3
     for (loop=1;loop<=3;loop++) //先后对 3 个球作处理
     {switch (loop) //loop 的值先后为 1,2,3
      {case 1: pri=color(i);break; //color(i)是强制类型转换，使 pri 的值为 i
       case 2: pri=color(j);break; //使 pri 的值为 j
       case 3: pri=color(k);break; //使 pri 的值为 k
       default :break;
      }
     }
    }
  }

  switch (pri) //判断 pri 的值，输出相应的“颜色”
  {case red: cout<<setw(8)<<"red"; break;
   case yellow: cout<<setw(8)<<"yellow";break;
   case blue:cout<<setw(8)<<"blue"; break;
   case white:cout<<setw(8)<<"white"; break;
   case black:cout<<setw(8)<<"black"; break;
   default : break;
  }
}

cout<<endl;
}
}

cout<<"total:"<<n<<endl; //输出符合条件的组合的个数
return 0;
}

```

运行结果如下:

```

1 red yellow blue 2 red yellow white 3 red yellow black
:
:
:
58 black white red
59 black white yellow
60 black white blue
total:60

```

不用枚举常量,而用常数 0 代表“红”，1 代表“黄”……也可以。但显然用枚举变量更直观，因为枚举元素都选用了令人“见名知意”的标识符，而且枚举变量的值限制在定义时规定的几个枚举元素范围如果赋予它一

个其他的值，就会出现出错信息，便于检查。

#### 7.4 用 typedef 声明类型

除了用以上方法声明结构体、共用体、枚举等类型外，还可以用 typedef 声明一个新的类型名来代替已有的类型如

**typedef int** INTEGER; //指定用标识符 **INTEGER** 代表 **int** 类型

**typedef float** REAL; //指定用 **REAL** 代表 **float** 类型

这样，以下两行等价：

① **int i,j; float a,b;**

② **INTEGER i,j; REAL a,b;** 这样可以使熟悉 FORTRAN 的人能用 **INTEGER** 和 **REAL** 定义变量，以适应他们的习惯。

如果在一个程序中，整型变量是专门用来计数的，可以用 **COUNT** 来作为整型类型名：

**typedef int** COUNT; //指定用 **COUNT** 代表 **int** 型

**COUNT i,j;** //将变量 **i,j** 定义为 **COUNT** 类型，

即 **int** 类型在程序中将变量 **i,j** 定义为 **COUNT** 类型，可以使人更一目了然地知道它们是用于计数的。

也可以声明结构体类型：

**typedef struct** //注意在 **struct** 之前用了关键字 **typedef**，表示是声明新名

{ **int** month; **int** day; **int** year; } **DATE;** //注意 **DATE** 是新类型名，而不是结构体变量名

所声明的新类型名 **DATE** 代表上面指定的一个结构体类型。这样就可以用 **DATE** 定义变量：

**DATE** birthday; **DATE \*p;** //p 为指向此结构体类型数据的指针

还可以进一步：

① **typedef int** NUM[100]; //声明 **NUM** 为整型数组类型，包含 100 个元素 **NUM n;** //定义 **n** 为包含 100 个整型元素的数组

② **typedef char \*STRING;** //声明 **STRING** 为字符指针类型 **STRING p,s[10];** //p 为字符指针变量，s 为指针数组(有 10 个元素)

③ **typedef int (\*POINTER)();** //声明 **POINTER** 为指向函数的指针类型，函数返回整型值 **POINTER p1, p2;** //p1, p2 为 **POINTER** 类型的指针变量

归纳起来，声明一个新的类型名的方法是：

① 先按定义变量的方法写出定义语句（如 **int i;**）。

② 将变量名换成新类型名（如将 **i** 换成 **COUNT**）。

③ 在最前面加 **typedef** (如 **typedef int COUNT**)。

④ 然后可以用新类型名去定义变量。

再以声明上述的数组类型为例来说明：

① 先按定义数组形式书写：**int n[100];**

② 将变量名 **n** 换成自己指定的类型名：

**int** NUM[100];

③ 在前面加上 **typedef**，得到

**typedef int** NUM[100];

④ 用来定义变量：**NUM n;** (n 是包含 100 个整型元素的数组)。

习惯上常把用 typedef 声明的类型名用大写字母表示，以便与系统提供的标准类型标识符相区别。

说明：

(1) typedef 可以声明各种类型名，但不能用来定义变量。用 typedef 可以声明数组类型、字符串类型，使用比较方便。

(2) 用 typedef 只是对已经存在的类型增加一个类型名，而没有创造新的类型。

(3) 当在不同源文件中用到同一类型数据（尤其是像数组、指针、结构体、共用体等类型数据）时，常用 typedef 声明一些数据类型，把它们单独放在一个头文件中，然后在需要用到它们的文件中用 #include 命令把它们包含进来，以提高编程效率。

(4) 使用 typedef 有利于程序的通用与移植。有时程序会依赖于硬件特性，用 typedef 便于移植。

## 第 3 篇基于对象的程序设计

第 8 章 类和对象

第 9 章 关于类和对象的进一步讨论

第 10 章 运算符重载

## 第 8 章 类和对象

8.1 面向对象程序设计方法概述

8.2 类的声明和对象的定义

8.3 类的成员函数

8.4 对象成员的引用

8.5 类的封装性和信息隐蔽

8.6 类和对象的简单应用举例

8.7 面向对象程序设计方法概述

8.1 面向对象程序设计方法概述

到目前为止，我们介绍的是 C++ 在面向过程的程序设计中的应用。对于规模比较小的程序，编程者可以直接编写出一个面向过程的程序，详细地描述每一瞬时的数据结构及对其的操作过程。但是当程序规模较大时，就显得力不从心了。C++ 就是为了解决编写大程序过程中的困难而产生的。

8.1.1 什么是面向对象的程序设计

面向对象的程序设计的思路 and 人们日常生活中处理问题的思路是相似的。在自然世界和社会生活中，一个复杂的事物总是由许多部分组成的。

当人们生产汽车时，分别设计和制造发动机、底盘、车身和轮子，最后把它们组装在一起。在组装时，各部分之间有一定的联系，以便协调工作。这就是面向对象的程序设计的基本思路。

为了进一步说明问题，下面先讨论几个有关的概念。

### 1. 对象

客观世界中任何一个事物都可以看成一个对象(object)。对象可大可小。对象是构成系统的基本单位。

任何一个对象都应当具有这两个要素，即属性(attribute)和行为(behavior)，它能根据外界给的信息进行相应的操作。

一个对象往往是由一组属性和一组行为构成的。一般来说，凡是具备属性和行为这两种要素的，都可以作为对象。在一个系统中的多个对象之间通过一定的渠道相互联系，如图 8.1 示意。要使某一个对象实现某一种行为(即操作)，应当向它传送相应的消息。对象之间就是这样通过发送和接收消息互相联系的。

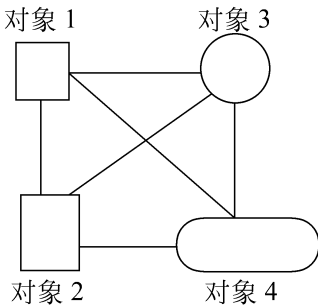


图 8.1

面向对象的程序设计采用了以上人们所熟悉的这种思路。使用面向对象的程序设计方法设计一个复杂的软件系统时，首要的问题是确定该系统是由哪些对象组成的，并且设计这些对象。在 C++ 中，每个对象都是由数据和函数(即操作代码)这两部分组成的，见图 8.2。

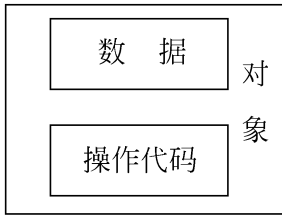


图 8.2

数据体现了前面提到的“属性”，如一个三角形对象，它的 3 个边长就是它的属性。函数是用来对数据进行操作的，以便实现某些功能，例如可以通过边长计算出三角形的面积，并且输出三角形的边长和面积。计算三角形面积和输出有关数据就是前面提到的行为，在程序设计方法中也称为方法(method)。调用对象中的函数就是向该对象传送一个消息(message)，要求该对象实现某一行(功能)。

### 2. 封装与信息隐蔽

可以对一个对象进行封装处理，把它的一部分属性和功能对外界屏蔽，也就是说从外界是看不到的，甚至是不可知的。这样做的好处是大大降低了操作对象的复杂程度。

面向对象程序设计方法的一个重要特点就是“封装性”(encapsulation)，所谓“封装”，指两方面的含义：

一是将有关的数据和操作代码封装在一个对象中，形成一个基本单位，各个对象之间相对独立，互不干扰。二是将对象中某些部分对外隐蔽，即隐蔽其内部细节，只留下少量接口，以便与外界联系，接收外界的消息。这种对外界隐蔽的做法称为信息隐蔽(imformation hiding)。

信息隐蔽还有利于数据安全，防止无关的人了解和修改数据。

C++ 的对象中的函数名就是对象的对外接口，外界可以通过函数名来调用这些函数来实现某些行为(功能)。这些将在以后详细介绍。

### 3. 抽象

在程序设计方法中，常用到抽象(abstraction)这一名词。抽象的过程是将有关事物的共性归纳、集中的过

程。抽象的作用是表示同一类事物的本质。

C 和 C++ 中的数据类型就是对一批具体的数的抽象。对象是具体存在的，如一个三角形可以作为一个对象，10 个不同尺寸的三角形是 10 个对象。如果这 10 个三角形对象有相同的属性和行为，可以将它们抽象为一种类型，称为三角形类型。

在 C++ 中，这种类型就称为“类(class)”。这 10 个三角形就是属于同一“类”的对象。类是对象的抽象，而对象则是类的特例，或者说是类的具体表现形式。

#### 4. 继承与重用

如果在软件开发中已经建立了一个名为 A 的“类”，又想另外建立一个名为 B 的“类”，而后者与前者内容基本相同，只是在前者的基础上增加一些属性和行为，只需在类 A 的基础上增加一些新内容即可。这就是面向对象程序设计中的继承机制。

利用继承可以简化程序设计的步骤。

“白马”继承了“马”的基本特征，又增加了新的特征(颜色)，“马”是父类，或称为基类，“白马”是从“马”派生出来的，称为子类或派生类。

C++ 提供了继承机制，采用继承的方法可以很方便地利用一个已有的类建立一个新的类。这就是常说的“软件重用”(software reusability) 的思想。

#### 5. 多态性

如果有几个相似而不完全相同的对象，有时人们要求在向它们发出同一个消息时，它们的反应各不相同，分别执行不同的操作。这种情况就是多态现象。

如，在 Windows 环境下，用鼠标双击一个文件对象(这就是向对象传送一个消息)，如果对象是一个可执行文件，则会执行此程序，如果对象是一个文本文件，则启动文本编辑器并打开该文件。

在 C++ 中，所谓多态性(polymorphism)是指：由继承而产生的相关的不同的类，其对象对同一消息会作出不同的响应。多态性是面向对象程序设计的一个重要特征，能增加程序的灵活性。

#### 8.1.2 面向对象程序设计的特点

传统的面向过程程序设计是围绕功能进行的，用一个函数实现一个功能。所有的数据都是公用的，一个函数可以使用任何一组数据，而一组数据又能被多个函数所使用（见图 8.3）。

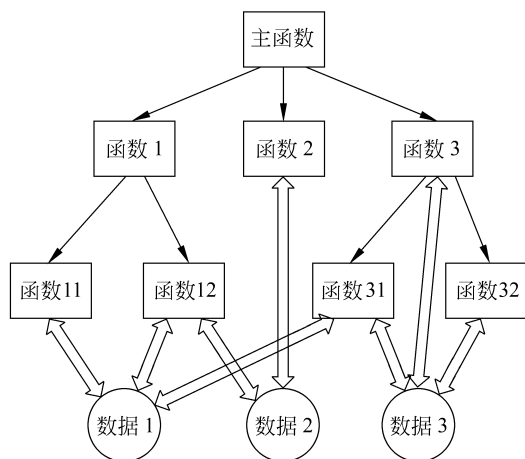


图 8.3

面向对象程序设计采取的是另外一种思路。它面对的是一个对象。实际上，每一组数据都是有特定的用途的，是某种操作的对象。也就是说，一组操作调用一组数据。

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。

这时他如同一个总调度，不断地向各个对象发出命令，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，减少出错机会。



### 8.1.3 类和对象的作用

类是 C++ 中十分重要的概念，它是实现面向对象程序设计的基础。类是所有面向对象的语言的共同特征，所有面向对象的语言都提供了这种类型。一个有一定规模的 C++ 程序是由许多类所构成的。

C++ 支持面向过程的程序设计，也支持基于对象的设计，又支持面向对象的设计。在本章到第 10 章将介绍基于对象的设计。包括类和对象的概念、类的机制和声明、类对象的定义与使用等。这是面向对象的设计的基础。基于对象就是基于类。与面向过程的程序不同，基于对象的程序是以类和对象为基础的，程序的操作是围绕对象进行的。

在此基础上利用了继承机制和多态性，就成为面向对象的设计(有时不细分基于对象程序设计和面向对象程序设计，而把二者合称为面向对象的设计)。

基于对象程序设计所面对的是一个对象。所有的数据分别属于不同的对象。在面向过程的结构化程序设计中，人们常使用这样的公式来表述程序：

程序 = 算法 + 数据结构

算法和数据结构两者是互相独立、分开设计的，面向过程的设计是以算法为主体的。在实践中人们逐渐认识到算法和数据结构是互相紧密联系不可分的，应当以一个算法对应一组数据结构，而不宜提倡一个算法对应多组数据结构，以及一组数据结构对应多个算法。基于对象和面向对象程序设计就是把一个算法和一组数据结构封装在一个对象中。因此，就形成了新的观念：

对象 = 算法 + 数据结构

程序 = (对象 + 对象 + 对象 + ...) + 消息

或：

程序 = 对象 s + 消息

“对象 s”表示多个对象。消息的作用就是对对象的控制。程序设计的关键是设计好每一个对象，及确定向这些对象发出的命令，使各对象完成相应操作。

### 8.1.4 面向对象的软件开发

随着软件规模的迅速增大，软件人员面临的问题十分复杂。需要规范整个软件开发过程，明确软件开发过程中每个阶段的任务，在保证前一个阶段工作的正确性的情况下，再进行下一阶段的工作。这就是软件工程需要研究和解决的问题。面向对象的软件工程包括以下几个部分：

#### 1. 面向对象分析(object oriented analysis, OOA)

软件工程中的系统分析阶段，系统分析员要和用户结合在一起，对用户的要求作出精确的分析和明确的描述，从宏观的角度概括出系统应该做什么(而不是怎么做)。面向对象的分析，要按照面向对象的概念和方法，在对任务的分析中，从客观存在的事物和事物之间的关系，归纳出有关的对象(包括对象的属性和行为)以及对象之间的联系，并将具有相同属性和行为的对象用一个类(class)来表示。建立一个能反映真实工作情况的需求模型。

#### 2. 面向对象设计(object oriented design, OOD)

根据面向对象分析阶段形成的需求模型，对每一部分分别进行具体的设计，

首先是进行类的设计，类的设计可能包含多个层次(利用继承与派生)。

然后以这些类为基础提出程序设计的思路和方法，包括对算法的设计。

在设计阶段，并不牵涉某一种具体的计算机语言，而是用一种更通用的描述工具(如伪代码或流程图)来描述。

#### 3. 面向对象编程(object oriented programming, OOP)

根据面向对象设计的结果，用一种计算机语言把它写成程序，显然应当选用面向对象的计算机语言(例如 C++)，否则无法实现面向对象设计的要求。

#### 4. 面向对象测试(object oriented test, OOT)

在写好程序后交给用户使用前，必须对程序进行严格的测试。测试的目的是发现程序中的错误并改正它。

面向对象测试是用面向对象的方法进行测试，以类作为测试的基本单元。

#### 5. 面向对象维护(object oriented soft maintenance, OOSM)

因为对象的封装性，修改一个对象对其它对象影响很小。利用面向对象的方法维护程序，大大提高了软件维护的效率。现在设计一个大的软件，是严格按照面向对象软件工程的 5 个阶段进行的，这 5 个阶段的工

作不是由一个人从头到尾完成的，而是由不同的人分别完成的。这样，OOP 阶段的任务就比较简单了，程序编写者只需要根据 OOD 提出的思路用面向对象语言编写出程序即可。在一个大型软件的开发中，OOP 只是面向对象开发过程中的一个很小的部分。如果所处理的是一个较简单的问题，可以不必严格按照以上 5 个阶段进行，往往由程序设计者按照面向对象的方法进行程序设计，包括类的设计(或选用已有的类)和程序的设计。

## 8.2 类的声明和对象的定义

### 8.2.1 类和对象的关系

每一个实体都是对象。有一些对象是具有相同的结构和特性的。每个对象都属于一个特定的类型。在 C++ 中对象的类型称为类(class)。类代表了某一批对象的共性和特征。前面已说明：

类是对象的抽象，而对象是类的具体实例(instance)。

正如同结构体类型和结构体变量的关系一样，人们先声明一个结构体类型，然后用它去定义结构体变量。同一个结构体类型可以定义出多个不同的结构体变量。

在 C++ 中也是先声明一个类类型，然后用它去定义若干个同类型的对象。对象就是类类型的一个变量。可以说类是对象的模板，是用来定义对象的一种抽象类型。

类是抽象的，不占用内存，而对象是具体的，占用存储空间。

在一开始时弄清对象和类的关系是十分重要的。

### 8.2.2 声明类类型

类是用户自己指定的类型。如果程序中要用到类类型，必须自己根据需要进行声明，或者使用别人已设计好的类。C++ 标准本身并不提供现成的类的名称、结构和内容。

在 C++ 中声明一个类类型和声明一个结构体类型是相似的。

下面是声明一个结构体类型的方法：

struct Student //声明了一个名为 Student 的结构体类型

```
{
int num;
char name[20];
char sex;
};
```

Student stud1, stud2;

//定义了两个结构体变量 stud1 和 stud2

它只包括数据，没有包括操作。

现在声明一个类：

class Student //以 class 开头

```
{
int num;
char name[20];
char sex; //以上 3 行是数据成员
void display() //这是成员函数
{cout<<"num:"<<num<<endl;
cout<<"name:"<<name<<endl;
cout<<"sex:"<<sex<<endl;
//以上 4 行是函数中的操作语句}
};
```

Student stud1,stud2; //定义了两个 Student 类的对象 stud1 和 stud2

可以看到声明类的方法是由声明结构体类型的方法发展而来的。



可以看到，类(class)就是对象的类型。实际上，类是一种广义的数据类型。类这种数据类型中的数据既包含数据，也包含操作数据的函数。

不能把类中的全部成员与外界隔离，一般是把数据隐蔽起来，而把成员函数作为对外界的接口。

可以将上面类的声明改为

```
class Student //声明类类型
{private : //声明以下部分为私有的
int num;
char name[20];
char sex;
```

```
public : //声明以下部分为公用的
```

```
void display()
{cout<<"num:"<<num<<endl;
cout<<"name:"<<name<<endl;
cout<<"sex:"<<sex<<endl;}
};
```

```
Student stud1,stud2;//定义了两个 Student 类的对象
```

如果在类的定义中既不指定 private，也不指定 public，则系统就默认为私有的。归纳以上对类类型的声明，

可得到其一般形式如下：

```
class 类名
{
private : 私有的数据和成员函数;
public : 公用的数据和成员函数;
};
```

private 和 public 称为成员访问限定符(member access specifier)。除了 private 和 public 之外，还有一种成员访问限定符 protected(受保护的)，用 protected 声明的成员称为受保护的成员，它不能被类外访问(这点与私有成员类似)，但可以被派生类的成员函数访问。

在声明类类型时，声明为 private 的成员和声明为 public 的成员的次序任意，既可以先出现 private 部分，也可以先出现 public 部分。如果在类体中既不写关键字 private，又不写 public，就默认为 private。

在一个类体中，关键字 private 和 public 可以分别出现多次。每个部分的有效范围到出现另一个访问限定符或类体结束时(最后一个右花括号)为止。

但是为了使程序清晰，应该养成这样的习惯：使每一种成员访问限定符在类定义体中只出现一次。

在以前的 C++ 程序中，常先出现 private 部分，后出现 public 部分，如上面所示。

现在的 C++ 程序多数先写 public 部分，把 private 部分放在类体的后部。

这样可以使用户将注意力集中在能被外界调用的成员上，使阅读者的思路更清晰一些。

在 C++ 程序中，经常可以看到类。为了用户方便，常用的 C++ 编译系统往往向用户提供类库(但不属于 C++ 语言的组成部分)，内装常用的基本的类，供用户使用。不少用户也把自己或本单位经常用到的类放在一个专门的类库中，需要用时直接调用，这样就减少了程序设计的工作量。

### 8.2.3 定义对象的方法

8.2.2 节的程序段中，最后一行用已声明的 Student 类来定义对象，这种方法是很容易理解的。经过定义后，stud1 和 stud2 就成为具有 Student 类特征的对象。stud1 和 stud2 这两个对象都分别包括 Student 类中定义的数据和函数。

定义对象也可以有几种方法。

1. 先声明类类型，然后再定义对象前面用的就是这种方法，如 `Student stud1, stud2;`//`Student` 是已经声明的类类型在 C++ 中，声明了类类型后，定义对象有两种形式。

(1) `class` 类名 对象名

如

```
class Student stud1,stud2;
```

把 `class` 和 `Student` 合起来作为一个类名，用来定义对象。

(2) 类名 对象名

如 `Student stud1, stud2;`

直接用类名定义对象。

这两种方法是等效的。第 1 种方法是从 C 语言继承下来的，第 2 种方法是 C++ 的特色，显然第 2 种方法更为简捷方便。

2. 在声明类类型的同时定义对象

```
class Student
```

```
//声明类类型
```

```
{public :
```

```
//先声明公用部分
```

```
void display()
```

```
{cout<<"num:"<<num<<endl;
```

```
cout<<"name:"<<name<<endl;
```

```
cout<<"sex:"<<sex<<endl;}
```

```
private :
```

```
//后声明私有部分
```

```
int num;
```

```
char name[20];
```

```
char sex;
```

```
}stud1, stud2;//定义了两个 Student 类的对象
```

在定义 `Student` 类的同时，定义了两个 `Student` 类的对象。

3. 不出现类名，直接定义对象

```
class //无类名
```

```
{private : //声明以下部分为私有的
```

```
⋮
```

```
public : //声明以下部分为公用的
```

```
⋮
```

```
}stud1, stud2;//定义了两个无类名的类对象
```

直接定义对象，在 C++ 中是合法的、允许的，但却很少用，也不提倡用。在实际的程序开发中，一般都采用上面 3 种方法中的第 1 种方法。在小型程序中或所声明的类只用于本程序时，也可以用第 2 种方法。在定义一个对象时，编译系统会为这个对象分配存储空间，以存放对象中的成员。

#### 8.2.4 类和结构体类型的异同

C++ 增加了 `class` 类型后，仍保留了结构体类型(`struct`)，而且它的功能也扩展了。C++ 允许用 `struct` 来定义一个类型。如可以将前面用关键字 `class` 声明的类类型改为用关键字 `struct`：

```
struct Student //用关键字 struct 来声明一个类类型
```

```
{private : //声明以下部分为私有的
```

```
int num;//以下 3 行为数据成员
```

```
char name[20];char sex;
```

```
public: //声明以下部分为公用的
```

```
void display() //成员函数
```

```
{cout<<"num:"<<num<<endl;cout<<"name:"<<name<<endl;cout<<"sex:"<<sex<<endl;}
};
```

Student stud1, stud2;//定义了两个 Student 类的对象

为了使结构体类型也具有封装的特征，C++不是简单地继承 C 的结构体，而是使它也具有类的特点，以便于用于面向对象程序设计。

用 struct 声明的结构体类型实际上也就是类。用 struct 声明的类，如果对其成员不作 private 或 public 的声明，系统将其默认为 public。

如果想分别指定私有成员和公用成员，则应用 private 或 public 作显式声明。

而用 class 定义的类，如果不作 private 或 public 声明，系统将其成员默认为 private，在需要时也可以自己用显式声明改变。

如果希望成员是公用的，使用 struct 比较方便，如果希望部分成员是私有的，宜用 class。建议尽量使用 class 来建立类，写出完全体现 C++风格的程序。

## 8.3 类的成员函数

### 8.3.1 成员函数的性质

类的成员函数(简称类函数)是函数的一种，它的用法和作用和第 4 章介绍过的函数基本上是一样的，它也有返回值和函数类型，

它与一般函数的区别只是：

它是属于一个类的成员，出现在类体中。

它可以被指定为 private(私有的)、public(公用的)或 protected(受保护的)。

在使用类函数时，要注意调用它的权限(它能否被调用)以及它的作用域(函数能使用什么范围中的数据 and 函数)。

例如私有的成员函数只能被本类中的其它成员函数所调用，而不能被类外调用。

成员函数可以访问本类中任何成员(包括私有的和公用的)，可以引用在本作用域中有效的数据。

一般的做法是将需要被外界调用的成员函数指定为 public，它们是类的对外接口。

但应注意，并非要求把所有成员函数都指定为 public。有的函数并不是准备为外界调用的，而是为本类中的成员函数所调用的，就应该将它们指定为 private。

这种函数的作用是支持其它函数的操作，是类中其它成员的工具函数(utility function)，类外用户不能调用这些私有的工具函数。

类的成员函数是类体中十分重要的部分。如果一个类中不包含成员函数，就等同于 C 语言中的结构体了，体现不出类在面向对象程序设计中的作用。

### 8.3.2 在类外定义成员函数

在前面已经看到成员函数是在类体中定义的。也可以在类体中只写成员函数的声明，而在类的外面进行函数定义。如

```
class Student
{ public :
void display();
//公用成员函数原型声明
private :
int num;
string name;
char sex;
//以上 3 行是私有数据成员
};
void Student::display()
//在类外定义 display 类函数
{cout<<"num:"<<num<<endl;
```

//函数体

```
cout<<"name:"<<name<<endl;
cout<<"sex:"<<sex<<endl;
}
```

Student stud1,stud2;

//定义两个类对象

注意：在类体中直接定义函数时，不需要在函数名前面加上类名，因为函数属于哪一个类是不言而喻的。但成员函数在类外定义时，必须在函数名前面加上类名，予以限定(qualified)，**“::”是作用域限定符(field qualifier)**或称作用域运算符，用它声明函数是属于哪个类的。

如果在作用域运算符**“::”**的前面没有类名，或者函数名前面既无类名又无作用域运算符**“::”**，

如 **::display()** 或 **display()**

则表示 **display** 函数不属于任何类，这个函数不是成员函数，而是全局函数，即非成员函数的一般普通函数。类函数必须先**在类体中作原型声明**，然后在类外定义，也就是说类体的位置应在函数定义之前，否则编译时会出错。

虽然函数在类的外部定义，但在调用成员函数时会根据在**类中声明的函数原型找到函数的定义**（函数代码），从而执行该函数。

在类的内部对成员函数作声明，而在类体外定义成员函数，这是程序设计的一种良好习惯。

如果一个函数，其函数体只有 2-3 行，一般可在声明类时在类体中定义。多于 3 行的函数，一般在类体内声明，在类外定义。

### 8.3.3 inline 成员函数

关于**内置(inline)**函数，已在第 4 章第 4.5 节中作过介绍。

类的成员函数也可以指定为内置函数。

在类体中定义的成员函数的规模一般都很小，而系统调用函数的过程所花费的时间开销相对是比较大的。调用一个函数的时间开销远远大于小规模函数体中全部语句的执行时间。

为了减少时间开销，如果在**类体中定义的成员函数**中不包括**循环**等控制结构，C++系统会自动将它们作为**内置(inline)**函数来处理。

也就是说，在程序调用这些成员函数时，并不是真正地**执行函数的调用过程**(如保留返回地址等处理)，而是把函数代码嵌入程序的调用点。

这样可以大大**减少调用成员函数**的时间开销。C++要求对一般的内置函数要用关键字 **inline** 声明，但对**类内定义的成员函数，可以省略 inline**，因为这些成员函数已被**隐含地指定为内置函数**。如

```
class Student
{public :
void display()
{cout<<"num:"<<num<<endl;cout<<"name:"
<<name<<endl;cout<<"sex:"<<sex<<endl;}
private :
int num;
string name;
char sex;
};
```

其中第 3 行

**void display()** 也可以写成

**inline void display()**

将 **display** 函数**显式**地声明为内置函数。

以上两种写法是等效的。对在**类体内定义的函数，一般都省写 inline**。

应该注意的是：如果成员函数不在类体内定义，而在**类体外**定义，系统并不把它默认为**内置(inline)**函数，

调用这些成员函数的过程和调用一般函数的过程是相同的。如果想将这些成员函数指定为内置函数，应当用 `inline` 作显式声明。如

```
class Student
{
public : inline void display();//声明此成员函数为内置函数
private :
int num;
string name;
char sex;
};

inline void Student::display() // 在类外定义 display 函数为内置函数
{cout<<"num:"<<num<<endl;cout<<"name:"<<name<<endl;cout<<"sex:"<<sex<<endl;}
```

在第 4 章第 4.5 节曾提到过，在函数的声明或函数的定义两者之一作 `inline` 声明即可。  
值得注意的是：如果在类体外定义 `inline` 函数，则必须将类定义和成员函数的定义都放在同一个头文件中(或者写在同一个源文件中)，否则编译时无法进行置换(将函数代码的拷贝嵌入到函数调用点)。  
但是这样做，不利于类的接口与类的实现分离，不利于信息隐蔽。虽然程序的执行效率提高了，但从软件工程质量的角度来看，这样做并不是好的办法。  
只有在类外定义的成员函数规模很小而调用频率较高时，才将此成员函数指定为内置函数。

8.3.4 成员函数的存储方式

用类去定义对象时，系统会为每一个对象分配存储空间。  
如果一个类包括了数据和函数，要分别为数据和函数的代码分配存储空间。  
按理说，如果用同一个类定义了 10 个对象，那么就需要分别为 10 个对象的数据和函数代码分配存储单元，如图 8.4 所示。

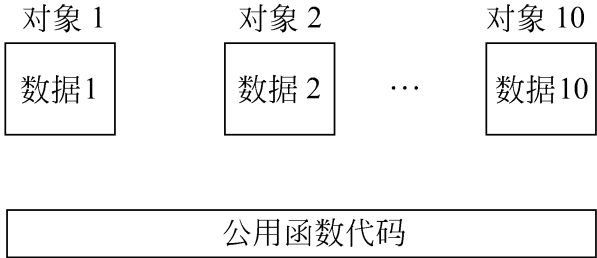


图 8.4  
能否只用一段空间来存放这个共同的函数代码段，在调用各对象的函数时，都去调用这个公用的函数代码。如图 8.5 所示。

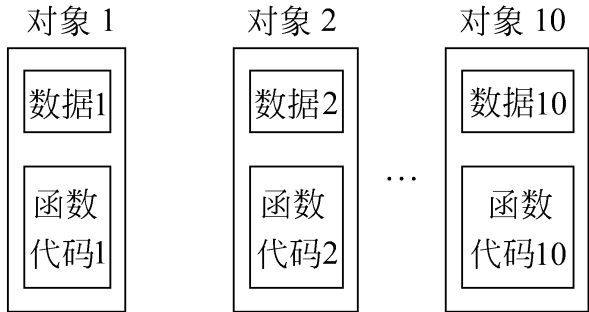


图 8.5  
显然，这样做会大大节约存储空间。C++编译系统正是这样做的，因此每个对象所占用的存储空间只是该对象的数据部分所占用的存储空间，而不包括函数代码所占用的存储空间。如果声明了一个类：

```
class Time
```

```
{
public:
int hour;
int minute;
int sec;
void set( )
{cin>>a>>b>>c;}
};
```

可以用下面的语句来输出该类对象所占用的字节数：

```
cout<<sizeof(Time)<<endl;
```

输出的值是 12。

这就证明了一个对象所占的空间大小只取决于该对象中数据成员所占的空间，而与成员函数无关。

函数代码是存储在对象空间之外的。如果对同一个类定义了 10 个对象，这些对象的成员函数对应的是同一个函数代码段，而不是 10 个不同的函数代码段。

需要注意的是：虽然调用不同对象的成员函数时都是执行同一段函数代码，但是执行结果一般是不相同的。

不同的对象使用的是同一个函数代码段，它怎么能够分别对不同对象中的数据进行操作呢？

原来 C++ 为此专门设立了一个名为 **this** 的指针，用来指向不同的对象。需要说明：

- (1) 不论成员函数在类内定义还是在类外定义，成员函数的代码段都用同一种方式存储。
- (2) 不要将成员函数的这种存储方式和 **inline**(内置)函数的概念混淆。
- (3) 应当说明：常说的“某某对象的成员函数”，是从逻辑的角度而言的，而成员函数的存储方式，是从物理的角度而言的，二者是不矛盾的。

## 8.4 对象成员的引用

在程序中经常需要访问对象中的成员。访问对象中的成员可以有 3 种方法：

通过对象名和成员运算符访问对象中的成员；

通过指向对象的指针访问对象中的成员；

通过对象的引用变量访问对象中的成员。

### 8.4.1 通过对象名和成员运算符访问对象中的成员

例如在程序中可以写出以下语句：

```
stud1.num=1001;//假设 num 已定义为公用的整型数据成员
```

表示将整数 1001 赋给对象 stud1 中的数据成员 num。

其中“.”是成员运算符，用来对成员进行限定，指明所访问的是哪一个对象中的成员。

注意不能只写成员名而忽略对象名。

访问对象中成员的一般形式为

对象名.成员名

不仅可以在类外引用对象的公用数据成员，而且还可以调用对象的公用成员函数，但同样必须指出对象名，如

```
stud1.display();//正确，调用对象 stud1 的公用成员函数
```

```
display();//错误，没有指明是哪一个对象的 display 函数
```

由于没有指明对象名，编译时把 display 作为普通函数处理。

应该注意所访问的成员是公用的(public)还是私有的(private)。只能访问 public 成员，而不能访问 private 成员，如果已定义 num 为私有数据成员，下面的语句是错误的：

```
stud1.num=10101;//num 是私有数据成员，不能被外界引用
```

在类外只能调用公用的成员函数。在一个类中应当至少有一个公用的成员函数，作为对外的接口，否则就无法对对象进行任何操作。



### 8.4.2 通过指向对象的指针访问对象中的成员

在第 7 章第 7.1.5 节中介绍了指向结构体变量的指针，可以通过指针引用结构体中的成员。用指针访问对象中的成员的方法与此类似。如果有以下程序段：

```
class Time
{
public : //数据成员是公用的
int hour;
int minute;
}; Time t, *p; //定义对象 t 和指针变量 p
p=&t; //使 p 指向对象 t
cout<<p->hour; //输出 p 指向的对象中的成员 hour
在 p 指向 t 的前提下，p->hour, (*p).hour 和 t.hour 三者等价。
```

### 8.4.3 通过对象的引用变量来访问对象中的成员

如果为一个对象定义了一个引用变量，它们是共占同一段存储单元的，实际上它们是同一个对象，只是用不同的名字表示而已。

因此完全可以通过引用变量来访问对象中的成员。

如果已声明了 Time 类，并有以下定义语句：

```
Time t1; //定义对象 t1
Time &t2=t1; //定义 Time 类引用变量 t2，并使之初始化为 t1
cout<<t2.hour; //输出对象 t1 中的成员 hour
```

由于 t2 与 t1 共占同一段存储单元(即 t2 是 t1 的别名)，因此 t2.hour 就是 t1.hour。

本章第 8.6 节的例 8.2 中的程序(b)，介绍的是引用变量作为形参的情况，读者可以参考。

## 8.5 类的封装性和信息隐蔽

### 8.5.1 公用接口与私有实现的分离

从前面的介绍已知：C++通过类来实现封装性，把数据和与这些数据有关的操作封装在一个类中，或者说，类的作用是把数据和算法封装在用户声明的抽象数据类型中。

在声明了一个类以后，用户主要是通过调用公用的成员函数来实现类提供的功能(例如对数据成员设置值，显示数据成员的值，对数据进行加工等)。

因此，公用成员函数是用户使用类的公用接口(public interface)，或者说是类的对外接口。

当然并不一定要把所有成员函数都指定为 public(公用)的，但这时这些成员函数就不是公用接口了。

在类外虽然不能直接访问私有数据成员，但可以通过调用公用成员函数来引用甚至修改私有数据成员。

用户可以调用公用成员函数来实现某些功能，而这些功能是在声明类时已指定的，用户可以使用它们而不应改变它们。

实际上用户往往并不关心这些功能是如何实现的细节，而只需知道调用哪个函数会得到什么结果，能实现什么功能即可。

通过成员函数对数据成员进行操作称为类的实现，为了防止用户任意修改公用成员函数，改变对数据进行的的操作，往往不让用户看到公用成员函数的源代码，显然更不能修改它，用户只能接触到公用成员函数的目标代码(详见 8.5.2 节)。

可以看到：类中被操作的数据是私有的，实现的细节对用户是隐蔽的，这种实现称为私有实现(private implementation)。

这种“类的公用接口与私有实现的分离”形成了信息隐蔽。

软件工程的一个最基本的原则就是将接口与实现分离，信息隐蔽是软件工程中一个非常重要的概念。它的好处在于：

- (1) 如果想修改或扩充类的功能，只需修改本类中有关的数据成员和与它有关的成员函数，程序中类外的部分可以不必修改。
- (2) 如果在编译时发现类中的数据读写有错，不必检查整个程序，只需检查本类中访问这些数据的少数成

员函数。

### 8.5.2 类声明和成员函数定义的分离

在面向对象的程序开发中，**一般做法是将类的声明**(其中包含成员函数的声明)放在指定的**头文件**中，用户如果想用该类，只要把有关的**头文件包含**进来即可，不必在程序中重复书写类的声明，以减少工作量，节省篇幅，提高编程的效率。

由于在头文件中包含了类的声明，因此在程序中**就可以用该类来定义对象**。由于在类体中包含了对成员函数的声明，在程序中就可以**调用这些对象的公用成员函数**。

为了实现上一节所叙述的信息隐蔽，对**类成员函数的定义**一般不放在头文件中，而另外放在一个文件中。

例如，可以分别**写两个文件**：

**//student.h (这是头文件，在此文件中进行类的声明)**

```
class Student //类声明
{ public :
void display(); //公用成员函数原型声明
private :
int num;
char name[20];
char sex;};
```

**//student.cpp**

**//在此文件中进行函数的定义**

```
#include <iostream>
```

```
#include "student.h" //不要漏写此行，否则编译通不过
```

```
void Student::display() //在类外定义 display 类函数
```

```
{cout<<"num:"<<num<<endl;
```

```
cout<<"name:"<<name<<endl;
```

```
cout<<"sex:"<<sex<<endl;
```

```
}
```

为了组成一个完整的源程序，还应当有包括主函数的源文件：

**//main.cpp 主函数模块**

```
#include <iostream>
```

```
#include "student.h" //将类声明头文件包含进来
```

```
int main( )
```

```
{Student stud; //定义对象
```

```
stud.display(); //执行 stud 对象的 display 函数
```

```
return 0;
```

```
}
```

这是一个包括**3个文件**的程序，组成两个文件模块：

一个是主模块 main.cpp，

一个是 student.cpp。

在主模块中又包含头文件 student.h。

在预编译时会将头文件 student.h 中的内容取代#include "student.h"行。

请注意： 由于将头文件 student.h 放在用户当前目录中，因此在文件名两侧用**双撇号**包起来("student.h")而**不用尖括号**(<student.h>)，否则编译时会找不到此文件。



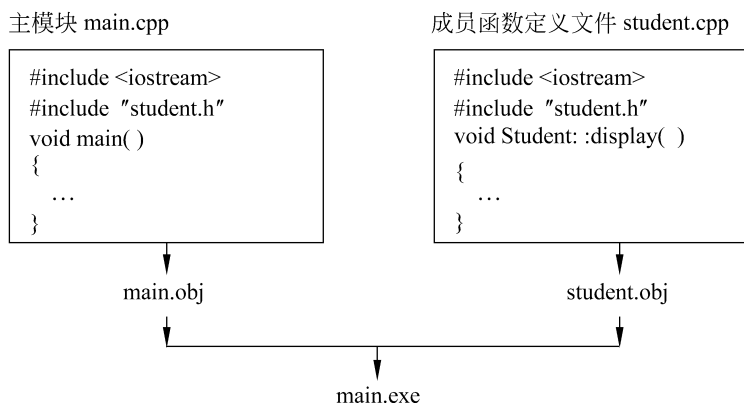


图 8.6。

在运行程序时调用 stud 中的 display 函数，输出各数据成员的值。

如果一个类声明多次被不同的程序所选用，**每次都要对包含成员函数定义的源文件**(如上面的 student.cpp)进行编译，这是否可以改进呢？

的确，可以不必每次都对它重复进行编译，而只需编译一次即可。

把第一次编译后所形成的目标文件保存起来，以后在需要时把它调出来直接与程序的目标文件相连接即可。这和使用函数库中的函数是类似的。这也是把成员函数的定义不放在头文件中的一个好处。

在实际工作中，并不是将一个类声明做成一个头文件，而是将若干个常用的功能相近的类声明集中在一起，形成类库。

类库有两种：

一种是 C++ 编译系统提供的标准类库；

一种是用户根据自己的需要做成的用户类库，提供给自己和自己授权的人使用，这称为自定义类库。

在程序开发工作中，类库是很有用的，它可以减少用户自己对类和成员函数进行定义的工作量。

类库包括两个组成部分：

(1) 类声明头文件；

(2) 已经过编译的成员函数的定义，它是目标文件。

**用户只需把类库装入到自己的计算机系统中**(一般装到 C++ 编译系统所在的子目录下)，并在程序中用 #include 命令行将有关的类声明的头文件包含到程序中，就可以使用这些类和其中的成员函数，顺利地运行程序。

这和在程序中使用 C++ 系统提供的标准函数的方法是一样的，例如用户在调用 sin 函数时只需将包含声明此函数的头文件包含到程序中，即可调用该库函数，而不必了解 sin 函数是怎么**实现的**(函数值是怎样计算出来的)。

当然，前提是系统已装了标准函数库。在用户源文件经过编译后，**与系统库**(是目标文件)相连接。

在用户程序中包含类声明头文件，类声明头文件就成为用户使用类的**公用接口**，在头文件的类体中还提供了成员函数的函数原型声明，用户只有通过头文件才能使用有关的类。

用户看得见和接触到的是这个头文件，任何要使用这个类的用户**只需包含这个头文件即可**。包含成员函数定义的文件就是类的实现。

请特别注意：**类声明和函数定义一般是分别放在两个文本中的**。由于要求接口与实现分离，为软件开发商向用户提供类库创造了很好的条件。

开发商把用户所需的各种类的声明按类放在不同的头文件中，同时对包含成员函数定义的源文件进行编译，得到成员函数定义的**目标代码**。

**软件商向用户提供这些头文件和类的实现的目标代码**(不提供函数定义的源代码)。用户在使用类库中的类时，只需将有关头文件包含到自己的程序中，并且在编译后连接成员函数定义的目标代码即可。

由于类库的出现，用户可以**像使用零件一样**方便地使用在实践中积累的通用的或专用的类，这就大大减少了程序设计的工作量，有效地提高了工作效率。

### 8.5.3 面向对象程序设计中的几个名词

类的成员函数在面向对象程序理论中被称为“方法”(method),“方法”是指对数据的操作。一个“方法”对应一种操作。

显然,只有被声明为公用的方法(成员函数)才能被对象外界所激活。

外界是通过发“消息”来激活有关方法的。所谓“消息”,其实就是一个命令,由程序语句来实现。前面的 `stud.display()` 就是向对象 `stud` 发出的一个“消息”,通知它执行其中的 `display`“方法”(即 `display` 函数)。

上面这个语句涉及 3 个术语:对象、方法和消息。

`stud` 是对象,

`display()` 是方法,

语句“`stud.display()`”是消息。

## 8.6 类和对象的简单应用举例

例 8.1 最简单的例子。

```
#include <iostream>
using namespace std;
class Time //定义 Time 类
{public : //数据成员为公用的
int hour;
int minute;
int sec;};
int main()
{
Time t1;//定义 t1 为 Time 类对象
cin>>t1.hour;//输入设定的时间
cin>>t1.minute;
cin>>t1.sec;
//输出时间:
cout<<t1.hour<<":"<<t1.minute<<":"<<t1.sec<<endl;
return 0;
}
```

运行情况如下: 1232 43 ✓

12:32:43

注意:

(1) 在引用数据成员 `hour`, `minute`, `sec` 时不要忘记在前面指定对象名。

(2) 不要错写为类名,

如写成

`Time.hour`, `Time.minute`, `Time.sec` 是不对的。因为类是一种抽象的数据类型,并不是一个实体,也不占存储空间,而对象是实际存在的实体,是占存储空间的,其数据成员是有值的,可以被引用的。

(3) 如果删去主函数的 3 个输入语句,即不向这些数据成员赋值,则它们的值是不可预知的。

```
Time t1;//定义对象 t1
Time &t2=t1;//定义 Time 类引用变量 t2,并初始化为 t1
cout<<t2.hour;//输出对象 t1 中的成员 hour
```

例 8.2 引用多个对象的成员。

(1) 程序(a)

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

```

class Time
{public :
int hour;
int minute;
int sec;
};
int main()
{Time t1;//定义对象 t1
cin>>t1.hour;//向 t1 的数据成员输入数据
cin>>t1.minute;
cin>>t1.sec;

cout<<t1.hour<<":"<<t1.minute<<":"<<t1.sec<<endl;//输出 t1 中数据成员的值
Time t2;//定义对象 t2
cin>>t2.hour;//向 t2 的数据成员输入数据
cin>>t2.minute;
cin>>t2.sec;
cout<<t2.hour<<":"<<t2.minute<<":"<<t2.sec<<endl;//输出 t2 中数据成员的值
return 0;
}

```

运行情况如下：

```

1032 43 ✓
10:32:43
22 32 43 ✓
22:32:43

```

程序是清晰易懂的，但是在主函数中对不同的对象一一写出有关操作，会使程序冗长。为了解决这个问题，可以使用函数来进行输入和输出。见程序(b)。

## (2) 程序(b)

```

#include <iostream>
using namespace std;
class Time
{public :
int hour;
int minute;
int sec;
};
int main()
{
void set_time(Time&);//函数声明
void show_time(Time&);//函数声明
Time t1;//定义 t1 为 Time 类对象
set_time(t1);//调用 set_time 函数，向 t1 对象中的数据成员输入数据
show_time(t1);//调用 show_time 函数，输出 t1 对象中的数据

Time t2;//定义 t2 为 Time 类对象
set_time(t2);//调用 set_time 函数，向 t2 对象中的数据成员输入数据
show_time(t2);//调用 show_time 函数，输出 t2 对象中的数据
}

```

```
return 0;
}
```

```
void set_time(Time& t) //定义函数 set_time，形参 t 是引用变量
{
    cin>>t.hour;//输入设定的时间
    cin>>t.minute;
    cin>>t.sec;
}
```

```
void show_time(Time& t) //定义函数 show_time，形参 t 是引用变量
{
    cout<<t.hour<<":"<<t.minute<<":"<<t.sec<<endl;//输出对象中的数据
}
```

运行情况与程序(a)相同。

### (3) 程序(c)

可以对上面的程序作一些修改，数据成员的值不再由键盘输入，而在调用函数时由实参给出，并在函数中使用默认参数。将程序(b)第 8 行以下部分改为

```
int main()
{
    void set_time(Time&,int hour=0,int
minute=0,int sec=0);//函数声明
    void show_time(Time&);//函数声明
    Time t1;
    set_time(t1,12,23,34);//通过实参传递时、分、秒的值
    show_time(t1);
    Time t2;
    set_time(t2);//使用默认的时、分、秒的值
    show_time(t2);
    return 0;
}
void set_time(Time& t,int hour,int minute,int sec)
{
    t.hour=hour;
    t.minute=minute;
    t.sec=sec;
}
void show_time(Time& t)
{
    cout<<t.hour<<":"<<t.minute<<":"<<t.sec<<endl;
}
```

程序运行时的输出为

12:23:34 (t1 中的时、分、秒)

0:0:0 (t2 中的时、分、秒)

以上两个程序中定义类都只有数据成员，没有成员函数，这显然没有体现出使用类的优越性。在下面的例子中，类体中就包含了成员函数。

例 8.3 将例 8.2 的程序改用含成员函数的类来处理。

```
#include <iostream>
using namespace std;
class Time
{public :
void set_time();//公用成员函数
void show_time();//公用成员函数
private : //数据成员为私有
int hour;
int minute;
int sec;
};
int main()
{
Time t1;//定义对象 t1
t1.set_time();//调用对象 t1 的成员函数 set_time, 向 t1 的数据成员输入数据
t1.show_time();//调用对象 t1 的成员函数 show_time, 输出 t1 的数据成员的值
Time t2;//定义对象 t2
t2.set_time();//调用对象 t2 的成员函数 set_time, 向 t2 的数据成员输入数据
t2.show_time();//调用对象 t2 的成员函数 show_time, 输出 t2 的数据成员的值
return 0;
}
void Time::set_time() //在类外定义 set_time 函数
{
cin>>hour;
cin>>minute;
cin>>sec;
}
void Time::show_time() //在类外定义 show_time 函数
{
cout<< hour<<":"<< minute<<":"<< sec<< endl;
}
```

运行情况与例 8.2 中的程序(a)相同。

注意:

- (1) 在主函数中调用两个成员函数时, 应指明对象名(t1,t2)。表示调用的是哪一个对象的成员函数。
- (2) 在类外定义函数时, 应指明函数的作用域(如 void Time::set\_time())。在成员函数引用本对象的数据成员时, 只需直接写数据成员名, 这时 C++系统会把它默认为本对象的数据成员。也可以显式地写出类名并使用域运算符。
- (3) 应注意区分什么场合用域运算符“::”, 什么场合用成员运算符“.”, 不要搞混。

例 8.4 找出一个整型数组中的元素的最大值。这个问题可以不用类的方法来解决, 现在用类来处理, 读者可以比较不同方法的特点。

```
#include <iostream>
using namespace std;
class Array_max //声明类
{public : //以下 3 行为成员函数原型声明
```

```

void set_value(); //对数组元素设置值
void max_value(); //找出数组中的最大元素
void show_value(); //输出最大值
private :
int array[10]; //整型数组
int max; //max 用来存放最大值
};
void Array_max::set_value() //成员函数定义，向数组元素输入数值
{ int i;
for (i=0;i<10;i++)
cin>> array[i];
}
void Array_max::max_value() //成员函数定义，找数组元素中的最大值
{int i;
max=array[0];
for (i=1;i<10;i++)
if(array[i]> max) max=array[i];
}
void Array_max::show_value() //成员函数定义，输出最大值
{cout<< "max="<< max;}
int main()
{Array_max arrmax; //定义对象 arrmax
arrmax.set_value(); //调用 arrmax 的 set_value 函数，向数组元素输入数值
arrmax.max_value(); //调用 arrmax 的 max_value 函数，找出数组元素中的最大值
arrmax.show_value(); //调用 arrmax 的 show_value 函数，输出数组元素中的最大值
return 0;
}

```

运行结果如下：

```

12 12 39 -34 17 134 045 -91 76 ✓ (输入 10 个元素的值)
max=134 (输入 10 个元素中的最大值)

```

请注意成员函数定义与调用成员函数的关系，定义成员函数只是设计了一组操作代码，并未实际执行，只有在被调用时才真正地执行这一组操作。

可以看出：主函数很简单，语句很少，只是调用有关对象的成员函数，去完成相应的操作。

在大多数情况下，主函数中甚至不出现控制结构(判断结构和循环结构)，而在成员函数中使用控制结构。

在面向对象的程序设计中，最关键的工作是类的设计。所有的数据和对数据的操作都体现在类中。

只要把类定义好，编写程序的工作就显得很简单了。

## 第 9 章 关于类和对象的进一步讨论

- 9.1 构造函数
- 9.2 析构函数
- 9.3 调用构造函数和析构函数的顺序
- 9.4 对象数组
- 9.5 对象指针
- 9.6 共用数据的保护
- 9.7 对象的动态建立和释放
- 9.8 对象的赋值和复制
- 9.9 静态成员
- 9.10 友元
- 9.11 类模板
- 9.1 构造函数

**9.1.1** 对象的初始化在建立一个对象时，常常需要作某些初始化的工作，例如对数据成员赋初值。

如果一个数据成员未被赋值，则它的值是不可预知的，因为在系统为它分配内存时，保留了这些存储单元的原状，这就成为了这些数据成员的初始值。

这种状况显然是与人们的要求不相符的，对象是一个实体，它反映了客观事物的属性(例如时钟的时、分、秒的值)，是应该有确定的值的。

注意： 类的数据成员是不能在声明类时初始化的。

如果一个类中所有的成员都是公用的，则可以在定义对象时对数据成员进行初始化。

如

```
class Time
{public : //声明为公用成员
    hour;
    minute;
    sec;
};
```

Time t1={14,56,30}; //将 t1 初始化为 14:56:30

这种情况和结构体变量的初始化是差不多的，在一个花括号内顺序列出各公用数据成员的值，两个值之间用逗号分隔。

但是，如果数据成员是私有的，或者类中有 private 或 protected 的成员，就不能用这种方法初始化。

在第 8 章的几个例子中，是用成员函数来对对象中的数据成员赋初值的(例如例 8.3 中的 set\_time 函数)。

从例 8.3 中可以看到，用户在主函数中调用 set\_time 函数来为数据成员赋值。

如果对一个类定义了多个对象，而且类中的数据成员比较多，那么，程序就显得非常臃肿烦琐。

### 9.1.2 构造函数的作用

为了解决这个问题，C++提供了构造函数(constructor)来处理对象的初始化。

构造函数是一种特殊的成员函数，与其他成员函数不同，不需要用户来调用它，而是在建立对象时自动执行。

构造函数的名字必须与类名同名，而不能由用户任意命名，以便编译系统能识别它并把它作为构造函数处理。

它不具有任何类型，不返回任何值。

构造函数的功能是由用户定义的，用户根据初始化的要求设计函数体和函数参数。

例 9.1 在例 8.3 基础上定义构造成员函数。

```
#include <iostream>
using namespace std;
class Time
{public :
    Time()
    {hour=0;
    minute=0;
    sec=0;
    }
    void set_time( );
    void show_time( );
private :
    int hour;
    int minute;
    int sec;
};
```



```

void Time::set_time()
{cin>>hour;
cin>>minute;
cin>>sec;
}
void Time::show_time()
{
cout<<hour<<":"<<minute<<":"<<sec<<endl;
}
int main()
{
Time t1;
t1.set_time();
t1.show_time();
Time t2;
t2.show_time();
return 0;
}

```

程序运行的情况为：

10 25 54 ✓

10:25:54

0:0:0

(从键盘输入新值赋给 t1 的数据成员)(输出 t1 的时、分、秒值)(输出 t2 的时、分、秒值)

上面是在类内定义构造函数的，也可以只在类内对构造函数进行声明而在类外定义构造函数。

将程序中的第 4~7 行改为下面一行：

**Time();** //对构造函数进行声明

在类外定义构造函数：

**Time::Time()** //在类外定义构造成员函数，要加上类名 **Time** 和域限定符“::”

```

{hour=0;
minute=0;
sec=0;
}

```

有关构造函数的使用，有以下说明：

- (1) 在类对象进入其作用域时调用构造函数。
- (2) 构造函数没有返回值，因此也不需要定义构造函数时声明类型，这是它和一般函数的一个重要的不同点。
- (3) 构造函数不需用户调用，也不能被用户调用。
- (4) 在构造函数的函数体中不仅可以对数据成员赋初值，而且可以包含其他语句。但是一般不提倡在构造函数中加入与初始化无关的内容，以保持程序的清晰。
- (5) 如果用户自己没有定义构造函数，则 C++ 系统会自动生成一个构造函数，只是这个构造函数的函数体是空的，也没有参数，不执行初始化操作。

### 9.1.3 带参数的构造函数

在例 9.1 中构造函数不带参数，在函数体中对数据成员赋初值。

这种方式使该类的每一个对象都得到同一组初值(例如例 9.1 中各数据成员的初值均为 0)。

但是有时用户希望对不同的对象赋予不同的初值。

可以采用带参数的构造函数，在调用不同对象的构造函数时，从外面将不同的数据传递给构造函数，以实现不同的初始化。

构造函数首部的一般格式为**构造函数名**(类型 1 形参 1, 类型 2 形参 2, ...) 前面已说明: 用户是不能调用构造函数的, 因此无法采用常规的调用函数的方法给出实参。

实参是在定义对象时给出的。

定义对象的一般格式为 类名 **对象名**(实参 1, 实参 2, ...);

例 9.2 有两个长方柱, 其长、宽、高分别为:

(1)12,20,25;

(2)10,14,20。

求它们的体积。

编一个基于对象的程序, 在类中用带参数的构造函数。

```
#include <iostream>
using namespace std;
class Box
{public :
    Box(int,int,int);
    int volume( );
private :
    int height;
    int width;
    int length;
};
//声明带参数的构造函数//声明计算体积的函数
Box::Box(int h,int w,int len) //在类外定义带参数的构造函数
{height=h;
width=w;
length=len;
}
int Box::volume( ) //定义计算体积的函数
{
    return (height*width*length);
}
int main()
{
    Box box1(12,25,30); //建立对象 box1, 并指定 box1 长、宽、高的值
    cout<<"The volume of box1 is "<<box1.volume()<<endl;
    Box box2(15,30,21); //建立对象 box2, 并指定 box2 长、宽、高的值
    cout<<"The volume of box2 is "<<box2.volume()<<endl;
    return 0;
}
```

程序运行结果如下:

The volume of box1 is 9000

The volume of box2 is 9450

可以知道:

- (1) 带参数的构造函数中的形参, 其对应的实参在定义对象时给定。
- (2) 用这种方法可以方便地实现对不同的对象进行不同的初始化。

#### 9.1.4 用参数初始化表对数据成员初始化

在 9.1.3 节中介绍的是在构造函数的函数体内通过赋值语句对数据成员实现初始化。

C++还提供另一种**初始化数据成员的方法**——**参数初始化表**来实现对数据成员的初始化。

这种方法不在函数体内对数据成员初始化，而是在函数首部实现。

例如例 9.2 中定义构造函数可以改用以下形式：

```
Box::Box(int h,int w,int len):height(h), width(w), length(len){ }
```

这种写法方便、简练，尤其当需要初始化的数据成员较多时更显其优越性。

甚至可以直接在类体中(而不是在类外)定义构造函数。

### 9.1.5 构造函数的重载

在一个类中可以定义多个构造函数，以便对类对象提供不同的初始化的方法，供用户选用。

这些构造函数具有相同的名字，而参数的个数或参数的类型不相同。

这称为构造函数的重载。

在第 4 章第 4.6 节中所介绍的函数重载的知识也适用于构造函数。

通过下面的例子可以了解怎样应用构造函数的重载。

例 9.3 在例 9.2 的基础上，定义两个构造函数，其中一个无参数，一个有参数。

```
#include <iostream>
using namespace std;
class Box
{public : Box( ); //声明一个无参的构造函数
  Box(int h,int w,int len):height(h),width(w),length(len){ } //声明一个有参的构造函数，用参数的初始化表对数据成员初始化
  int volume( );
private :
  int height;
  int width;
  int length;
};
Box::Box() //定义一个无参的构造函数
{
  height=10; width=10; length=10; } int Box::volume( ) {return (height*width*length);
}
int main( )
{
  Box box1; //建立对象 box1,不指定实参
  cout<<"The volume of box1 is "<<box1.volume( )<<endl;
  Box box2(15,30,25); //建立对象 box2,指定 3 个实参
  cout<<"The volume of box2 is "<<box2.volume( )<<endl;
  return 0;
}
```

在本程序中定义了两个重载的构造函数，其实还可以定义其他重载构造函数，其原型声明可以为

```
Box::Box(int h); //有 1 个参数的构造函数
```

```
Box::Box(int h,int w); //有两个参数的构造函数
```

在建立对象时分别给定 1 个参数和 2 个参数。

说明：

(1) 调用构造函数时不必给出实参的构造函数，称为默认构造函数(default constructor)。

显然，无参的构造函数属于默认构造函数。

一个类只能有一个默认构造函数。

(2) 如果在建立对象时选用的是无参构造函数，应注意正确书写定义对象的语句。

(3) 尽管在一个类中可以包含多个构造函数，但是对于每一个对象来说，建立对象时只执行其中一个构造

函数，并非每个构造函数都被执行。

### 9.1.6 使用默认参数的构造函数

构造函数中参数的值既可以通过实参传递，也可以指定为某些默认值，即如果用户不指定实参值，编译系统就使形参取默认值。

在第 4 章第 4.8 节中介绍过在函数中可以使用有默认值的参数。

在构造函数中也可以采用这样的方法来实现初始化。

例 9.3 的问题也可以使用包含默认参数的构造函数来处理。

例 9.4 将例 9.3 程序中的构造函数改用含默认值的参数，长、宽、高的默认值均为 10。

在例 9.3 程序的基础上改写如下：

```
#include <iostream>
using namespace std;
class Box
{public :
Box(int h=10,int w=10,int len=10); //在声明构造函数时指定默认参数
int volume( );
private :
int height;
int width;
int length;
};
Box::Box(int h,int w,int len) //在定义函数时可以不指定默认参数
{height=h;
width=w;
length=len;
}
int Box::volume( )
{return (height*width*length);
}
int main()
{
Box box1; //没有给实参
cout<<"The volume of box1 is "<<box1.volume()<<endl;
Box box2(15); //只给定一个实参
cout<<"The volume of box2 is "<<box2.volume()<<endl;
Box box3(15,30); //只给定 2 个实参
cout<<"The volume of box3 is "<<box3.volume()<<endl;
Box box4(15,30,20); //给定 3 个实参
cout<<"The volume of box4 is "<<box4.volume()<<endl;
return 0;
}
```

程序运行结果为

The volume of box1 is 1000

The volume of box2 is 1500

The volume of box3 is 4500

The volume of box4 is 9000

程序中对构造函数的定义(第 12-16 行)也可以改写成参数初始化表的形式:

```
Box::Box(int h,int w,int len):height(h),width(w),length(len){ }
```

可以看到: 在构造函数中使用默认参数是方便而有效的, 它提供了建立对象时的多种选择, 它的作用相当于好几个重载的构造函数。

它的好处是: 即使在调用构造函数时没有提供实参值, 不仅不会出错, 而且还确保按照默认的参数值对对象进行初始化。

尤其在希望对每一个对象都有同样的初始化状况时用这种方法更为方便。

说明:

- (1) 应该在声明构造函数时指定默认值, 而不能只在定义构造函数时指定默认值。
- (2) 程序第 5 行在声明构造函数时, 形参名可以省略。
- (3) 如果构造函数的全部参数都指定了默认值, 则在定义对象时可以给一个或几个实参, 也可以不给出实参。
- (4) 在一个类中定义了全部是默认参数的构造函数后, 不能再定义重载构造函数。

## 9.2 析构函数

析构函数(destructor)也是一个特殊的成员函数, 它的作用与构造函数相反, 它的名字是类名的前面加一个“~”符号。

在 C++ 中“~”是位取反运算符, 从这点也可以想到:

析构函数是与构造函数作用相反的函数。

当对象的生命期结束时, 会自动执行析构函数。

具体地说如果出现以下几种情况, 程序就会执行析构函数:

- ①如果在一个函数中定义了一个对象(它是自动局部对象), 当这个函数被调用结束时, 对象应该释放, 在对象释放前自动执行析构函数。
- ②static 局部对象在函数调用结束时对象并不释放, 因此也不调用析构函数, 只在 main 函数结束或调用 exit 函数结束程序时, 才调用 static 局部对象的析构函数。
- ③如果定义了一个全局对象, 则在程序的流程离开其作用域时(如 main 函数结束或调用 exit 函数) 时, 调用该全局对象的析构函数。
- ④如果用 new 运算符动态地建立了一个对象, 当用 delete 运算符释放该对象时, 先调用该对象的析构函数。析构函数的作用并不是删除对象, 而是在撤销对象占用的内存之前完成一些清理工作, 使这部分内存可以被程序分配给新对象使用。

程序设计者事先设计好析构函数, 以完成所需的功能, 只要对象的生命期结束, 程序就自动执行析构函数来完成这些工作。

析构函数不返回任何值, 没有函数类型, 也没有函数参数。

因此它不能被重载。

一个类可以有多个构造函数, 但只能有一个析构函数。

实际上, 析构函数的作用并不仅限于释放资源方面, 它还可以被用来执行“用户希望在最后一次使用对象之后所执行的任何操作”, 例如输出有关的信息。

这里说的用户是指类的设计者, 因为, 析构函数是在声明类的时候定义的。

也就是说, 析构函数可以完成类的设计者所指定的任何操作。

一般情况下, 类的设计者应当在声明类的同时定义析构函数, 以指定如何完成“清理”的工作。

如果用户没有定义析构函数, C++ 编译系统会自动生成一个析构函数, 但它只是徒有析构函数的名称和形式, 实际上什么操作都不进行。

想让析构函数完成任何工作, 都必须在定义的析构函数中指定。

例 9.5 包含构造函数和析构函数的 C++ 程序。

```
#include<string>
#include<iostream>
using namespace std;
class Student //声明 Student 类
```

```

{public :
student(int n,string nam,char s ) //定义构造函数
{num=n;
name=nam;
sex=s;
cout<<"Constructor called."<<endl; //输出有关信息
}
~Student() //定义析构函数
{cout<<"Destructor called."<<endl;} //输出有关信息
void display() //定义成员函数
{cout<<"num: "<<num<<endl;
cout<<"name: "<<name<<endl;
cout<<"sex: "<<sex<<endl<<endl; }
private :
int num;
char name[10];
char sex;
};
int main()
{Student stud1(10010,"Wang_li",'f'); //建立对象 stud1
stud1.display(); //输出学生 1 的数据
Student stud2(10011,"Zhang_fun",'m'); //定义对象 stud2
stud2.display(); //输出学生 2 的数据
return 0;
}

```

程序运行结果如下：

Constructor called.

num: 10010

name:Wang\_li

sex: f

Constructor called.

num: 10011

name:Zhang\_fun

sex:m

Destructor called.

Destructor called.

(执行 stud1 的构造函数) (执行 stud1 的 display 函数)

(执行 stud2 的构造函数) (执行 stud2 的 display 函数)

(执行 stud2 的析构函数) (执行 stud1 的析构函数)

### 9.3 调用构造函数和析构函数的顺序

在使用构造函数和析构函数时，需要特别注意对它们的调用时间和调用顺序。

在一般情况下，调用析构函数的次序正好与调用构造函数的次序相反：最先被调用的构造函数，其对应的(同一对象中的)析构函数最后被调用，而最后被调用的构造函数，其对应的析构函数最先被调用。

如图 9.1 示意。

图 9.1

但是，并不是在任何情况下都是按这一原则处理的。

在第 4 章第 4.11 和 4.12 节中曾介绍过作用域和存储类别的概念，这些概念对于对象也是适用的。



对象可以在不同的作用域中定义，可以有不同的存储类别。

这些会影响调用构造函数和析构函数的时机。

下面归纳一下什么时候调用构造函数和析构函数：

(1) **在全局范围中定义的对象**(即在所有函数之外定义的对象)，它的构造函数在文件中的**所有函数**(包括 main 函数)执行之前调用。

但如果一个程序中有多个文件，而不同的文件中都定义了全局对象，则这些对象的构造函数的执行顺序是不确定的。

当 **main 函数执行完毕或调用 exit 函数时**(此时程序终止)，调用析构函数。

(2) **如果定义的是局部自动对象**(例如在函数中定义对象)，则在建立对象时调用其构造函数。

如果函数被多次调用，则在每次建立对象时都要调用构造函数。

在函数调用结束、对象释放时先调用析构函数。

(3) **如果在函数中定义静态(static)局部对象**，则只在程序第一次调用此函数建立对象时调用构造函数一次，在调用结束时对象并不释放，因此也不调用析构函数，只在 main 函数结束或调用 exit 函数结束程序时，才调用析构函数。

构造函数和析构函数在面向对象的程序设计中是相当重要的。

以上介绍了最基本的、使用最多的普通构造函数，在本章第 9.8 节中将会介绍复制构造函数，在第 10 章第 10.7 节中还要介绍转换构造函数。

**9.4 对象数组**数组不仅可以由简单变量组成(例如整型数组的每一个元素都是整型变量)，也可以由对象组成(对象数组的每一个元素都是同类的对象)。

在日常生活中，有许多实体的属性是共同的，只是属性的具体内容不同。

例如一个班有 50 个学生，每个学生的属性包括姓名、性别、年龄、成绩等。

如果为每一个学生建立一个对象，需要分别取 50 个对象名。

用程序处理很不方便。

这时可以定义一个“学生类”对象数组，每一个数组元素是一个“学生类”对象。

例如

```
Student stud[50];
```

//假设已声明了 Student 类，定义 stud 数组，有 50 个元素

在建立数组时，同样要调用构造函数。

如果有 50 个元素，需要调用 50 次构造函数。

在需要时可以在定义数组时提供实参以实现初始化。

如果构造函数只有一个参数，在定义数组时可以直接在等号后面的花括号内提供实参。

如

```
Student stud[3]={60,70,78};
```

//合法，3 个实参分别传递给 3 个数组元素的构造函数

如果构造函数有多个参数，则不能用在定义数组时直接提供所有实参的方法，因为一个数组有多个元素，对每个元素要提供多个实参，如果再考虑到构造函数有默认参数的情况，很容易造成实参与形参的对应关系不清晰，出现歧义性。

例如，类 Student 的构造函数有多个参数，且为默认参数：

```
Student:: Student(int=1001,int=18,int=60);
```

//定义构造函数，有多个参数，且为默认参数

如果定义对象数组的语句为

```
Student stud[3]={1005,60,70};
```

在程序中最好不要采用这种容易引起歧义性的方法。

编译系统只为每个对象元素的构造函数传递一个实参，所以在定义数组时提供的实参个数不能超过数组元素个数，如

```
Student stud[3]={60,70,78,45};
```

//不合法，实参个数超过对象数组元素个数

那么，如果构造函数有多个参数，在定义对象数组时应当怎样实现初始化呢？回答是：在花括号中分别写出构造函数并指定实参。

如果构造函数有 3 个参数，分别代表学号、年龄、成绩。

则可以这样定义对象数组：

```
Student Stud[3]={ //定义对象数组
```

```
Student(1001,18,87), //调用第 1 个元素的构造函数，为它提供 3 个实参
```

```
Student(1002,19,76), //调用第 2 个元素的构造函数，为它提供 3 个实参
```

```
Student(1003,18,72) //调用第 3 个元素的构造函数，为它提供 3 个实参
```

```
};
```

在建立对象数组时，分别调用构造函数，对每个元素初始化。

每一个元素的实参分别用括号包起来，对应构造函数的一组形参，不会混淆。

例 9.6 对象数组的使用方法。

```
#include <iostream>
```

```
using namespace std;
```

```
class Box
```

```
{
```

```
public :
```

```
Box(int h=10,int w=12,int len=15): height(h),width(w),length(len){ } //声明有默认参数的构造函数，用参数初始化表对数据成员初始化
```

```
int volume( );
```

```
private :
```

```
int height;
```

```
int width;
```

```
int length;
```

```
}; int Box::volume( ) {return (height*width*length); }
```

```
int main( )
```

```
{ Box a[3]={ //定义对象数组
```

```
Box(10,12,15), //调用构造函数 Box，提供第 1 个元素的实参
```

```
Box(15,18,20), //调用构造函数 Box，提供第 2 个元素的实参
```

```
Box(16,20,26) //调用构造函数 Box，提供第 3 个元素的实参
```

```
};
```

```
cout<<"volume of a[0] is "<<a[0].volume( )<<endl;
```

```
cout<<"volume of a[1] is "<<a[1].volume( )<<endl;
```

```
cout<<"volume of a[2] is "<<a[2].volume( )<<endl;
```

```
}
```

运行结果如下：

```
volume of a[0] is 1800
```

```
volume of a[1] is 5400
```

```
volume of a[2] is 8320
```

## 9.5 对象指针

### 9.5.1 指向对象的指针

在建立对象时，编译系统会为每一个对象分配一定的存储空间，以存放其成员。

对象空间的起始地址就是对象的指针。

可以定义一个指针变量，用来存放对象的指针。

如果有一个类：

```
class Time
```

```
{public :
```



```
int hour;
int minute;
int sec;
void get_time();
};
void Time::get_time()
{cout<<hour<<":"<<minute<<":"<<sec<<endl;}
```

在此基础上有以下语句：

Time \*pt; //定义 pt 为指向 Time 类对象的指针变量

Time t1; //定义 t1 为 Time 类对象

pt=&t1; //将 t1 的起始地址赋给 pt

这样，pt 就是指向 Time 类对象的指针变量，它指向对象 t1。

定义指向类对象的指针变量的一般形式为

**类名 \*对象指针名；**

可以通过对象指针访问对象和对象的成员。

如

(\*pt).hour

pt->hour

(\*pt).get\_time()

pt->get\_time()

pt 所指向的对象，即 t1。

pt 所指向的对象中的 hour 成员，即 t1.hour

pt 所指向的对象中的 hour 成员，即 t1.hour 调用 pt 所指向的对象中的 get\_time 函数，即 t1.get\_time 调用

pt 所指向的对象中的 get\_time 函数，即 t1.get\_time

### 9.5.2 指向对象成员的指针

对象有地址，存放对象初始地址的指针变量就是指向对象的指针变量。

对象中的成员也有地址，存放对象成员地址的指针变量就是指向对象成员的指针变量。

**1.** 指向对象数据成员的指针定义指向对象数据成员的指针变量的方法和定义指向普通变量的指针变量方法相同。

例如

int \*p1; //定义指向整型数据的指针变量

定义指向对象数据成员的指针变量的一般形式为数据类型名 \*指针变量名；如果 Time 类的数据成员 hour 为公用的整型数据，则可以在类外通过指向对象数据成员的指针变量访问对象数据成员 hour。

p1=&t1.hour;

//将对象 t1 的数据成员 hour 的地址赋给 p1，p1 指向 t1.hour

cout<<\*p1<<endl;

//输出 t1.hour 的值

**2.** 指向对象成员函数的指针需要提醒读者注意： 定义指向对象成员函数的指针变量的方法和定义指向普通函数的指针变量方法有所不同。

成员函数与普通函数有一个最根本的区别： 它是类中的一个成员。

编译系统要求上面的赋值语句中，指针变量的类型必须与赋值号右侧函数的类型相匹配，要求在以下 3 方面都要匹配：

函数参数的类型和参数个数；

②函数返回值的类型；

③所属的类。

定义指向成员函数的指针变量应该采用下面的形式：

```
void (Time::*p2)();
```

//定义 p2 为指向 Time 类中公用成员函数的指针变量

定义指向公用成员函数的指针变量的一般形式为

**数据类型名 (类名::\*指针变量名)(参数表列);**

可以让它指向一个公用成员函数，只需把公用成员函数的入口地址赋给一个指向公用成员函数的指针变量即可。

如

```
p2=&Time::get_time;
```

使指针变量指向一个公用成员函数的一般形式为

**指针变量名=&类名::成员函数名;**

例 9.7 有关对象指针的使用方法。

```
#include <iostream>
```

```
using namespace std;
```

```
class Time
```

```
{public:
```

```
Time(int,int,int);
```

```
int hour;
```

```
int minute;
```

```
int sec;
```

```
void get_time( );
```

```
};
```

```
Time::Time(int h,int m,int s)
```

```
{hour=h;
```

```
minute=m;
```

```
sec=s;
```

```
}
```

```
void Time::get_time( ) //声明公有成员函数
```

```
//定义公有成员函数
```

```
{cout<<hour<<":"<<minute<<":"<<sec<<endl;}
```

```
int main()
```

```
{Time t1(10,13,56); //定义 Time 类对象 t1
```

```
int *p1=&t1.hour; //定义指向整型数据的指针变量 p1，并使 p1 指向 t1.hour
```

```
cout<<* p1<<endl; //输出 p1 所指的数据成员 t1.hour
```

```
t1.get_time( ); //调用对象 t1 的成员函数 get_time
```

```
Time *p2=&t1; //定义指向 Time 类对象的指针变量 p2，并使 p2 指向 t1
```

```
p2->get_time( ); //调用 p2 所指向对象(即 t1)的 get_time 函数
```

```
void (Time::*p3)(); //定义指向 Time 类公用成员函数的指针变量 p3
```

```
p3=&Time::get_time; //使 p3 指向 Time 类公用成员函数 get_time
```

```
(t1.*p3)(); //调用对象 t1 中 p3 所指的成员函数(即 t1.get_time())
```

```
}
```

程序运行结果为

10 (main 函数第 4 行的输出)

10:13:56 (main 函数第 5 行的输出)

10:13:56 (main 函数第 7 行的输出)

10:13:56 (main 函数第 10 行的输出)

可以看到为了输出 t1 中 hour,minute 和 sec 的值，可以采用 3 种不同的方法。

说明：

(1) 从 main 函数第 9 行可以看出： 成员函数的入口地址的正确写法是： &类名::成员函数名。

(2) main 函数第 8、9 两行可以合写为一行：

```
void (Time::*p3)()=&Time::get_time; //定义指针变量时指定其指向
```

### 9.5.3 this 指针

在第 8 章中曾经提到过： 每个对象中的数据成员都分别占有存储空间，如果对同一个类定义了 n 个对象，则有 n 组同样大小的空间以存放 n 个对象中的数据成员。

但是，不同对象都调用同一个函数代码段。

那么，当不同对象的成员函数引用数据成员时，怎么能保证引用的是所指定的对象的数据成员呢？假如，对于例 9.6 程序中定义的 Box 类，定义了 3 个同类对象 a,b,c。

如果有 a.volume()，应该是引用对象 a 中的 height, width 和 length，计算出长方体 a 的体积。

如果有 b.volume()，应该是引用对象 b 中的 height, width 和 length，计算出长方体 b 的体积。

而现在都用同一个函数段，系统怎样使它分别引用 a 或 b 中的数据成员呢？

在每一个成员函数中都包含一个特殊的指针，这个指针的名字是固定的，称为 this。

它是指向本类对象的指针，它的值是当前被调用的成员函数所在的对象的起始地址。

例如，当调用成员函数 a.volume 时，编译系统就把对象 a 的起始地址赋给 this 指针，于是在成员函数引用数据成员时，就按照 this 的指向找到对象 a 的数据成员。

例如 volume 函数要计算 height\*width\*length 的值，实际上是执行：

```
(this->height)*(this->width)*(this->length)
```

由于当前 this 指向 a，因此相当于执行：

```
(a.height)*(a.width)*( a.length)
```

这就计算出长方体 a 的体积。

同样如果有 b.volume()，编译系统就把对象 b 的起始地址赋给成员函数 volume 的 this 指针，显然计算出来的是长方体 b 的体积。

this 指针是隐式使用的，它是作为参数被传递给成员函数的。

本来，成员函数 volume 的定义如下：

```
int Box::volume()  
{return (height*width*length);  
}
```

C++把它处理为

```
int Box::volume(Box *this)  
{return (this->height * this->width * this->length); } 即在成员函数的形参表列中增加一个 this 指针。
```

在调用该成员函数时，实际上是用以下方式调用的：

```
a.volume(&a);
```

将对象 a 的地址传给形参 this 指针。

然后按 this 的指向去引用其他成员。

需要说明： 这些都是编译系统自动实现的，编程者不必人为地在形参中增加 this 指针，也不必将对象 a 的地址传给 this 指针。

在需要时也可以显式地使用 this 指针。

例如在 Box 类的 volume 函数中，下面两种表示方法都是合法的、相互等价的。

```
return (height * width * length); //隐含使用 this 指针
```

```
return (this->height * this->width * this->length); //显式使用 this 指针
```

可以用\*this 表示被调用的成员函数所在的对象，\*this 就是 this 所指向的对象，即当前的对象。

例如在成员函数 a.volume()的函数体中，如果出现\*this，它就是本对象 a。

上面的 return 语句也可写成

```
return((*this).height * (*this).width * (*this).length);
```

注意\*this 两侧的括号不能省略，不能写成\*this.height。

所谓“调用对象 a 的成员函数 f”，实际上是在调用成员函数 f 时使 this 指针指向对象 a，从而访问对象 a 的成员。

在使用“调用对象 a 的成员函数 f”时，应当对它的含义有正确的理解。

## 9.6 共用数据的保护

C++ 虽然采取了不少有效的措施(如设 private 保护)以增加数据的安全性，但是有些数据却往往是共享的，人们可以在不同的场合通过不同的途径访问同一个数据对象。

有时在无意之中的误操作会改变有关数据的状况，而这是人们所不希望出现的。

既要使数据能在一定范围内共享，又要保证它不被任意修改，这时可以使用 const，即把有关的数据定义为常量。

### 9.6.1 常对象

在定义对象时指定对象为常对象。

常对象必须要有初值，如

```
Time const t1(12,34,46); //t1 是常对象
```

这样，在所有的场合中，对象 t1 中的所有成员的值都不能被修改。

凡希望保证数据成员不被改变的对象，可以声明为常对象。

定义常对象的一般形式为

```
类名 const 对象名[(实参表列)];
```

也可以把 const 写在最左面：

```
const 类名 对象名[(实参表列)];
```

二者等价。

如果一个对象被声明为常对象，则不能调用该对象的非 const 型的成员函数(除了由系统自动调用的隐式的构造函数和析构函数)。

例如，对于例 9.7 中已定义的 Time 类，如果有

```
const Time t1(10,15,36); //定义常对象 t1
```

```
t1.get_time(); //企图调用常对象 t1 中的非 const 型成员函数，非法
```

这是为了防止这些函数会修改常对象中数据成员的值。

不能仅依靠编程者的细心来保证程序不出错，编译系统充分考虑到可能出现的情况，对不安全的因素予以拦截。

现在，编译系统只检查函数的声明，只要发现调用了常对象的成员函数，而且该函数未被声明为 const，就报错，提请编程者注意。

引用常对象中的数据成员很简单，只需将该成员函数声明为 const 即可。

如

```
void get_time() const; //将函数声明为 const
```

这表示 get\_time 是一个 const 型函数，即常成员函数。

常成员函数可以访问常对象中的数据成员，但仍然不允许修改常对象中数据成员的值。

有时在编程时有要求，一定要修改常对象中的某个数据成员的值，ANSI C++ 考虑到实际编程时的需要，对此作了特殊的处理，对该数据成员声明为 mutable，如

```
mutable int count;
```

把 count 声明为可变的数据成员，这样就可以用声明为 const 的成员函数来修改它的值。

### 9.6.2 常对象成员可以将对象的成员声明为 const，包括常数据成员和常成员函数。

1. 常数据成员其作用和用法与一般常变量相似，用关键字 const 来声明常数据成员。

常数据成员的值是不能改变的。

有一点要注意：只能通过构造函数的参数初始化表对常数据成员进行初始化。

如在类体中定义了常数据成员 hour：

```
const int hour; //声明 hour 为常数据成员
```

不能采用在构造函数中对常数据成员赋初值的方法。

在类外定义构造函数，应写成以下形式：

Time::Time(int h):hour(h){} //通过参数初始化表对常数据成员 hour 初始化

常对象的数据成员都是常数据成员，因此常对象的构造函数只能用参数初始化表对常数据成员进行初始化。

2. 常成员函数 前面已提到：一般的成员函数可以引用本类中的非 const 数据成员，也可以修改它们。

如果将成员函数声明为常成员函数，则只能引用本类中的数据成员，而不能修改它们，例如只用于输出数据等。

如

void get\_time() const; //注意 const 的位置在函数名和括号之后

const 是函数类型的一部分，在声明函数和定义函数时都要有 const 关键字，在调用时不必加 const。

常成员函数可以引用 const 数据成员，也可以引用非 const 的数据成员。

const 数据成员可以被 const 成员函数引用，也可以被非 const 的成员函数引用。

具体情况可以用书中表 9.1 表示。

怎样利用常成员函数呢？

(1) 如果在一个类中，有些数据成员的值允许改变，另一些数据成员的值不允许改变，则可以将一部分数据成员声明为 const，以保证其值不被改变，可以用非 const 的成员函数引用这些数据成员的值，并修改非 const 数据成员的值。

(2) 如果要求所有的数据成员的值都不允许改变，则可以将所有的数据成员声明为 const，或将对象声明为 const(常对象)，然后用 const 成员函数引用数据成员，这样起到“双保险”的作用，切实保证

(3) 如果已定义了一个常对象，只能调用其中的 const 成员函数，而不能调用非 const 成员函数(不论这些函数是否会修改对象中的数据)。

这是为了保证数据的安全。

如果需要访问对象中的数据成员，可将常对象中所有成员函数都声明为 const 成员函数，但应确保在函数中不修改对象中的数据成员。

不要误认为常对象中的成员函数都是常成员函数。

常对象只保证其数据成员是常数据成员，其值不被修改。

如果在常对象中的成员函数未加 const 声明，编译系统把它作为非 const 成员函数处理。

还有一点要指出：常成员函数不能调用另一个非 const 成员函数。

### 9.6.3 指向对象的常指针

将指针变量声明为 const 型，这样指针值始终保持为其初值，不能改变。

如 Time t1(10,12,15),t2; //定义对象

Time \* const ptr1; //const 位置在指针变量名前面，规定 ptr1 的值是常值

ptr1=&t1; //ptr1 指向对象 t1，此后不能再改变指向

ptr1=&t2; //错误，ptr1 不能改变指向

定义指向对象的常指针的一般形式为类名 \* const 指针变量名；也可以在定义指针变量时使之初始化，如将上面第 2,3 行合并为

Time \* const ptr1=&t1; //指定 ptr1 指向 t1

请注意：指向对象的常指针变量的值不能改变，即始终指向同一个对象，但可以改变其所指向对象(如 t1)的值。

如果想将一个指针变量固定地与一个对象相联系(即该指针变量始终指向一个对象)，可以将它指定为 const 型指针变量。

往往用常指针作为函数的形参，目的是不允许在函数执行过程中改变指针变量的值，使其始终指向原来的对象。

### 9.6.4 指向常对象的指针变量

为了更容易理解指向常对象的指针变量的概念和使用，首先了解指向常变量的指针变量，然后再进一步研究指向常对象的指针变量。

下面定义了一个指向常变量的指针变量

ptr: const char \*ptr;



注意 `const` 的位置在最左侧，它与类型名 `char` 紧连，表示指针变量 `ptr` 指向的 `char` 变量是常变量，不能通过 `ptr` 来改变其值的。

定义指向常变量的指针变量的一般形式为 `const` 类型名 \*指针变量名；说明：

(1) 如果一个变量已被声明为常变量，只能用指向常变量的指针变量指向它，**而不能用一般的**(指向非 `const` 型变量的)指针变量去指向它。

(2) 指向常变量的指针变量除了可以指向常变量外，还可以指向未被声明为 `const` 的变量。

此时不能通过此指针变量改变该变量的值。

如果希望在任何情况下都不能改变 `c1` 的值，则应把它定义为 `const` 型。

(3) 如果函数的形参是指向非 `const` 型变量的指针，实参只能用指向非 `const` 变量的指针，而不能用指向 `const` 变量的指针，这样，在执行函数的过程中可以改变形参指针变量**所指向的变量**(也就是实参指针所指向的变量)的值。

如果函数的形参是指向 `const` 型变量的指针，在执行函数过程中显然不能改变指针变量所指向的变量的值，因此允许实参是指向 `const` 变量的指针，或指向非 `const` 变量的指针。

使用形参和实参的对应关系见书中表 9.2。

**以上的对应关系与在(2)中所介绍的指针变量和其所指向的变量的关系是一致的：** 指向常变量的指针变量可以指向 `const` 和非 `const` 型的变量，而指向非 `const` 型变量的指针变量只能指向非 `const` 的变量。

以上介绍的是指向常变量的指针变量，指向常对象的指针变量的概念和使用是与此类似的，只要将“变量”换成“对象”即可。

(1) 如果一个对象已被声明为常对象，只能用指向常对象的指针变量指向它，**而不能用一般的**(指向非 `const` 型对象的)指针变量去指向它。

(2) 如果定义了一个指向常对象的指针变量，并使它指向一个非 `const` 的对象，则其指向的对象是不能通过指针来改变的。

如果希望在任何情况下 `t1` 的值都不能改变，则应把它定义为 `const` 型。

(3) 指向常对象的指针最常用于函数的形参，目的是在保护形参指针所指向的对象，使它在函数执行过程中不被修改。

请记住这样一条规则： 当希望在调用函数时对象的值不被修改，就应当把形参定义为指向常对象的指针变量，**同时用对象的地址作实参**(对象可以是 `const` 或非 `const` 型)。

如果要求该对象不仅在调用函数过程中不被改变，而且要求它在程序执行过程中都不改变，则应把它定义为 `const` 型。

(4) 如果定义了一个指向常对象的指针变量，是不能通过它改变所指向的对象的值的，但是指针变量本身的值是可以改变的。

#### 9.6.5 对象的常引用

过去曾介绍： 一个变量的引用就是变量的别名。

实质上，变量名和引用名都指向同一段内存单元。

如果形参为变量的引用名，实参为变量名，则在调用函数进行虚实结合时，**并不是为形参另外开辟一个存储空间**(常称为建立实参的一个拷贝)，**而是把实参变量的地址传给形参**(引用名)，这样引用名也指向实参变量。

例 9.8 对象的常引用。

```
#include <iostream>
using namespace std;
class Time
{public:
    Time(int,int,int);
    int hour;
    int minute;
    int sec;
};
```

```
Time::Time(int h,int m,int s) //定义构造函数
```

```
{hour=h;
```

```
minute=m;
```

```
sec=s;
```

```
}
```

```
void fun(Time &t)
```

```
{t.hour=18;}
```

```
int main( )
```

```
{Time t1(10,13,56);
```

```
fun(t1);
```

```
cout<<t1.hour<<endl;
```

```
return 0;
```

```
}
```

如果不希望在函数中修改实参 t1 的值，可以把引用变量 t 声明为 const(常引用)，函数原型为 void fun(const Time &t); 则在函数中不能改变 t 的值，也就是不能改变其对应的实参 t1 的值。

在 C++ 面向对象程序设计中，经常用常指针和常引用作函数参数。

这样既能保证数据安全，使数据不能被随意修改，在调用函数时又不必建立实参的拷贝。

用常指针和常引用作函数参数，可以提高程序运行效率。

### 9.6.6 const 型数据的小结表 9.3

## 9.7 对象的动态建立和释放

用前面介绍的方法定义的对象是静态的，在程序运行过程中，对象所占的空间是不能随时释放的。

但有时人们希望在需要用到对象时才建立对象，在不需要用该对象时就撤销它，释放它所占的内存空间以供别的数据使用。

这样可提高内存空间的利用率。

在第 7 章 7.1.7 节中介绍了用 new 运算符动态地分配内存，用 delete 运算符释放这些内存空间。

这也适用于对象，可以用 new 运算符动态建立对象，用 delete 运算符撤销对象。

如果已经定义了一个 Box 类，可以用下面的方法动态地建立一个对象：

new Box; 编译系统开辟了一段内存空间，并在此内存空间中存放一个 Box 类对象，同时调用该类的构造函数，以使该对象初始化(如果已对构造函数赋予此功能的话)。

但是此时用户还无法访问这个对象，因为这个对象既没有对象名，用户也不知道它的地址。

这种对象称为无名对象，它确实是存在的，但它没有名字。

用 new 运算符动态地分配内存后，将返回一个指向新对象的指针的值，即所分配的内存空间的起始地址。

用户可以获得这个地址，并通过这个地址来访问这个对象。

需要定义一个指向本类的对象的指针变量来存放该地址。

如 Box \*pt; //定义一个指向 Box 类对象的指针变量 pt

pt=new Box; //在 pt 中存放了新建对象的起始地址

在程序中就可以通过 pt 访问这个新建的对象。

如

cout<<pt->height; //输出该对象的 height 成员

cout<<pt->volume( ); //调用该对象的 volume 函数，计算并输出体积

C++ 还允许在执行 new 时，对新建立的对象进行初始化。

如

Box \*pt=new Box(12,15,18);

这种写法是把上面两个语句(定义指针变量和用 new 建立新对象)合并为一个语句，并指定初值。

这样更精炼。

新对象中的 height, width 和 length 分别获得初值 12,15,18。

调用对象既可以通过对象名, 也可以通过指针。

用 new 建立的动态对象一般是不用对象名的, 是通过指针访问的, 它主要应用于动态的数据结构, 如链表。访问链表中的结点, 并不需要通过对象名, 而是在上一个结点中存放下一个结点的地址, 从而由上一个结点找到下一个结点, 构成链接的关系。

在执行 new 运算时, 如果内存量不足, 无法开辟所需的内存空间, 目前大多数 C++编译系统都使 new 返回一个 0 指针值。

只要检测返回值是否为 0, 就可判断分配内存是否成功。

ANSI C++标准提出, 在执行 new 出现故障时, 就“抛出”一个“异常”, 用户可根据异常进行有关处理。

但 C++标准仍然允许在出现 new 故障时返回 0 指针值。

当前, 不同的编译系统对 new 故障的处理方法是不同的。

在不再需要使用由 new 建立的对象时, 可以用 delete 运算符予以释放。

如

```
delete pt; //释放 pt 指向的内存空间
```

这就撤销了 pt 指向的对象。

此后程序不能再使用该对象。

如果用一个指针变量 pt 先后指向不同的动态对象, 应注意指针变量的当前指向, 以免删错了对象。

在执行 delete 运算符时, 在释放内存空间之前, 自动调用析构函数, 完成有关善后清理工作。

## 9.8 对象的赋值和复制

### 9.8.1 对象的赋值

如果对一个类定义了两个或多个对象, 则这些同类的对象之间可以互相赋值, 或者说, 一个对象的值可以赋给另一个同类的对象。

这里所指的对象的值是指对象中所有数据成员的值。

对象之间的赋值也是通过赋值运算符“=”进行的。

本来, 赋值运算符“=”只能用来对单个的变量赋值, 现在被扩展为两个同类对象之间的赋值, 这是通过对赋值运算符的重载实现的。

实际这个过程是通过成员复制来完成的, 即将一个对象的成员值一一复制给另一对象的对应成员。

对象赋值的一般形式为

对象名 1 = 对象名 2;

注意对象名 1 和对象名 2 必须属于同一个类。

例如

```
Student stud1,stud2; //定义两个同类的对象
```

```
⋮
```

```
stud2=stud1; //将 stud1 赋给 stud2
```

通过下面的例子可以了解怎样进行对象的赋值。

例 9.9 对象的赋值。

```
#include <iostream>
using namespace std;
class Box
{public :
Box(int =10,int =10,int =10); //声明有默认参数的构造函数
int volume( );
private :
int height;
int width;
int length;
};
```



```

Box::Box(int h,int w,int len)
{height=h;
width=w;
length=len;
}
int Box::volume( )
{return (height*width*length); //返回体积
}
int main()
{Box box1(15,30,25),box2; //定义两个对象 box1 和 box2
cout<<"The volume of box1 is "<<box1.volume()<<endl;
box2=box1; //将 box1 的值赋给 box2
cout<<"The volume of box2 is "<<box2.volume()<<endl; return 0;
}

```

运行结果如下：

The volume of box1 is 11250

The volume of box2 is 11250

说明： (1) 对象的赋值只对其中的数据成员赋值，而不对成员函数赋值。

(2) 类的数据成员中不能包括动态分配的数据，否则在赋值时可能出现严重后果。

### 9.8.2 对象的复制

有时需要用到多个完全相同的对象。

此外，有时需要将对象在某一瞬时的状态保留下来。

这就是对象的复制机制。

用一个已有的对象快速地复制出多个完全相同的对象。

如 Box box2(box1); 其作用是用已有的对象 box1 去克隆出一个新对象 box2。

其一般形式为

类名 对象 2(对象 1);

用对象 1 复制出对象 2。

可以看到： 它与前面介绍过的定义对象方式类似，但是括号中给出的参数不是一般的变量，而是对象。

在建立对象时调用一个特殊的构造函数——复制构造函数(copy constructor)。

这个函数的形式是这样的：

//The copy constructor definition.

```

Box::Box(const Box& b)
{height=b.height; width=b.width; length=b.length; }

```

复制构造函数也是构造函数，但它只有一个参数，这个参数是本类的对象(不能是其他类的对象)，而且采用对象的引用的形式(一般约定加 const 声明，使参数值不能改变，以免在调用此函数时因不慎而使对象值被修改)。

此复制构造函数的作用就是将实参对象的各成员值一一赋给新的对象中对应的成员。

回顾复制对象的语句 Box box2(box1); 这实际上也是建立对象的语句，建立一个新对象 box2。

由于在括号内给定的实参是对象，因此编译系统就调用复制构造函数(它的形参也是对象)，而不会去调用其他构造函数。

实参 box1 的地址传递给形参 b(b 是 box1 的引用)，因此执行复制构造函数的函数体时，将 box1 对象中各数据成员的值赋给 box2 中各数据成员。

如果用户自己未定义复制构造函数，则编译系统会自动提供一个默认的复制构造函数，其作用只是简单地复制类中每个数据成员。

C++还提供另一种方便用户的复制形式，用赋值号代替括号，如

```
Box box2=box1; //用 box1 初始化 box2
```

其一般形式为

类名 对象名 1 = 对象名 2;

可以在一个语句中进行多个对象的复制。

如

```
Box box2=box1,box3=box2;
```

按 box1 来复制 box2 和 box3。

可以看出： 这种形式与变量初始化语句类似，请与下面定义变量的语句作比较：

```
int a=4,b=a;
```

这种形式看起来很直观，用起来很方便。

但是其作用都是调用复制构造函数。

请注意对象的复制和 9.8.1 节介绍的对象的赋值在概念上和语法上的不同。

对象的赋值是对一个已经存在的对象赋值，因此必须先定义被赋值的对象，才能进行赋值。

而对象的复制则是从无到有地建立一个新对象，并使它与一个已有的对象完全相同(包括对象的结构和成员的值)。

可以对例 9.7 程序中的主函数作一些修改：

```
int main()  
{Box box1(15,30,25); //定义 box1  
cout<<"The volume of box1 is "<<box1.volume()<<endl; Box box2=box1,box3=box2; //按 box1 来复制 box2,box3  
cout<<"The volume of box2 is "<<box2.volume()<<endl;  
    cout<<"The volume of box3 is "<<box3.volume()<<endl;  
}
```

执行完第 3 行后，3 个对象的状态完全相同。

请注意普通构造函数和复制构造函数的区别。

(1) 在形式上类名(形参表列); //普通构造函数的声明，如 Box(int h,int w,int len);

类名(类名& 对象名); //复制构造函数的声明，如 Box(Box &b);

(2) 在建立对象时，实参类型不同。

系统会根据实参的类型决定调用普通构造函数或复制构造函数。

如

```
Box box1(12,15,16); //实参为整数，调用普通构造函数
```

```
Box box2(box1); //实参是对象名，调用复制构造函数
```

(3) 在什么情况下被调用

普通构造函数在程序中建立对象时被调用。

复制构造函数在用已有对象复制一个新对象时被调用，在以下 3 种情况下需要克隆对象：

① 程序中需要新建一个对象，并用另一个同类的对象对它初始化，如前面介绍的那样。

② 当函数的参数为类的对象时。

在调用函数时需要将实参对象完整地传递给形参，也就是需要建立一个实参的拷贝，这就是按实参复制一个形参，系统是通过调用复制构造函数来实现的，这样能保证形参具有和实参完全相同的值。

如

```
void fun(Box b) //形参是类的对象
```

```
{}
```

```
int main()
```

```
{Box box1(12,15,18);
```

```
fun(box1); //实参是类的对象，调用函数时将复制一个新对象 b
```

```
return 0;
```

```
}
```

③ 函数的返回值是类的对象。

在函数调用完毕将返回值带回函数调用处时。

此时需要将函数中的对象复制一个临时对象并传给该函数的调用处。

如

```
Box f() //函数 f 的类型为 Box 类类型
```

```
{Box box1(12,15,18);
```

```
return box1; //返回值是 Box 类的对象
```

```
}
```

```
int main()
```

```
{Box box2; //定义 Box 类的对象 box2
```

```
box2=f(); //调用 f 函数，返回 Box 类的临时对象，并将它赋值给 box2
```

```
}
```

以上几种调用复制构造函数都是由编译系统自动实现的，不必由用户自己去调用，读者只要知道在这些情况下需要调用复制构造函数就可以了。

**9.9 静态成员**如果有  $n$  个同类的对象，那么每一个对象都分别有自己的数据成员，不同对象的数据成员各自有值，互不相干。

但是有时人们希望有某一个或几个数据成员为所有对象所共有。

这样可以实现数据共享。

在第 7 章中曾介绍过全局变量，它能够实现数据共享。

如果在一个程序文件中有多个函数，在每一个函数中都可以改变全局变量的值，全局变量的值为各函数共享。

但是用全局变量的安全性得不到保证，由于在各处都可以自由地修改全局变量的值，很有可能偶一失误，全局变量的值就被修改，导致程序的失败。

因此在实际工作中很少使用全局变量。

如果想在同类的多个对象之间实现数据共享，也不用全局对象，可以用静态的数据成员。

#### 9.9.1 静态数据成员

静态数据成员是一种特殊的数据成员。

它以关键字 `static` 开头。

例如

```
class Box {public : int volume( ); private : static int height; //把 height 定义为静态的数据成员
```

```
int width; int length;
```

```
};
```

如果希望各对象中的 `height` 的值是一样的，就可以把它定义为静态数据成员，这样它就为各对象所共有，而不只属于某个对象的成员，所有对象都可以引用它。

静态的数据成员在内存中只占一份空间。

每个对象都可以引用这个静态数据成员。

静态数据成员的值对所有对象都是一样的。

如果改变它的值，则在各对象中这个数据成员的值都同时改变了。

这样可以节约空间，提高效率。

说明：(1) 在第 8 章中曾强调：如果只声明了类而未定义对象，则类的一般数据成员是不占内存空间的，只有在定义对象时，才为对象的数据成员分配空间。

但是静态数据成员不属于某一个对象，在为对象所分配的空间中不包括静态数据成员所占的空间。

静态数据成员是在所有对象之外单独开辟空间。

只要在类中定义了静态数据成员，即使不定义对象，也为静态数据成员分配空间，它可以被引用。

在一个类中可以有一个或多个静态数据成员，所有的对象共享这些静态数据成员，都可以引用它。

(2) 在第7章中曾介绍了静态变量的概念： 如果在一个函数中定义了静态变量，在函数结束时该静态变量并不释放，仍然存在并保留其值。

现在讨论的静态数据成员也是类似的，它不随对象的建立而分配空间，**也不随对象的撤销而释放**(一般数据成员是在对象建立时分配空间，在对象撤销时释放)。

静态数据成员是在程序编译时被分配空间的，到程序结束时才释放空间。

(3) 静态数据成员可以初始化，但只能在类体外进行初始化。

如

`int Box::height=10;` //表示对 `Box` 类中的数据成员初始化

其一般形式为

数据类型类名::静态数据成员名=初值;

不必在初始化语句中加 `static`。

注意： 不能用参数初始化表对静态数据成员初始化。

如在定义 `Box` 类中这样定义构造函数是错误的：

`Box(int h,int w,int len):height(h){ }` //错误，`height` 是静态数据成员

如果未对静态数据成员赋初值，则编译系统会自动赋予初值 0。

(4) 静态数据成员既可以通过对象名引用，也可以通过类名来引用。

请观察下面的程序。

例 9.10 引用静态数据成员。

```
cout<<a.height<<endl; //通过对象名 a 引用静态数据成员
cout<<b.height<<endl; //通过对象名 b 引用静态数据成员
cout<<Box::height<<endl; //通过类名引用静态数据成员
cout<<a.volume( )<<endl; //调用 volume 函数，计算体积，输出结果;
```

上面 3 个输出语句的输出结果相同(都是 10)。

这就验证了所有对象的静态数据成员实际上是同一个数据成员。

请注意： 在上面的程序中将 `height` 定义为公用的静态数据成员，所以在类外可以直接引用。

可以看到在类外可以通过对象名引用公用的静态数据成员，也可以通过类名引用静态数据成员。

即使没有定义类对象，也可以通过类名引用静态数据成员。

这说明静态数据成员并不是属于对象的，而是属于类的，但类的对象可以引用它。

如果静态数据成员被定义为私有的，则不能在类外直接引用，而必须通过公用的成员函数引用。

(5) 有了静态数据成员，各对象之间的数据有了沟通的渠道，实现数据共享，因此可以不使用全局变量。

全局变量破坏了封装的原则，不符合面向对象程序的要求。

但是也要注意公用静态数据成员与全局变量的不同，**静态数据成员的作用域只限于定义该类的作用域内**(如果是在一个函数中定义类，那么其中静态数据成员的作用域就是此函数内)。

在此作用域内，可以通过类名和域运算符“::”引用静态数据成员，而不论类对象是否存在。

### 9.9.2 静态成员函数成员

函数也可以定义为静态的，在类中声明函数的前面加 `static` 就成了静态成员函数。

如 `static int volume( )`; 和静态数据成员一样，静态成员函数是类的一部分，而不是对象的一部分。

如果要在类外调用公用的静态成员函数，要用类名和域运算符“::”。

如 `Box::volume( )`; 实际上也允许通过对象名调用静态成员函数，如 `a.volume( )`; 但这并不意味着此函数是属于对象 `a` 的，而只是用 `a` 的类型而已。

与静态数据成员不同，静态成员函数的作用不是为了对象之间的沟通，而是为了能处理静态数据成员。

前面曾指出： **当调用一个对象的成员函数**(非静态成员函数)时，系统会把该对象的起始地址赋给成员函数的 `this` 指针。

而静态成员函数并不属于某一对象，它与任何对象都无关，因此静态成员函数没有 `this` 指针。

既然它没有指向某一对象，**就无法对一个对象中的非静态成员进行默认访问**(即在引用数据成员时不指定对象名可以说，静态成员函数与非静态成员函数的根本区别是： 非静态成员函数有 `this` 指针，而静态成员函数没有 `this` 指针。

由此决定了静态成员函数不能访问本类中的非静态成员。

静态成员函数可以直接引用本类中的静态数据成员，因为静态成员同样是属于类的，可以直接引用。

在 C++ 程序中，静态成员函数主要用来访问静态数据成员，而不访问非静态成员。

假如在一个静态成员函数中有以下语句：`cout<<height<<endl;` //若 `height` 已声明为 `static`，则引用本类中的静态成员，合法 `cout<<width<<endl;` //若 `width` 是非静态数据成员，不合法

但是，并不是绝对不能引用本类中的非静态成员，只是不能进行默认访问，因为无法知道应该去找哪个对象。

如果一定要引用本类的非静态成员，应该加对象名和成员运算符“.”。

如

`cout<<a.width<<endl;` //引用本类对象 `a` 中的非静态成员

假设 `a` 已定义为 `Box` 类对象，且在当前作用域内有效，则此语句合法。

通过例 9.11 可以具体了解有关引用非静态成员的具体方法。

例 9.11 静态成员函数的应用。

```
#include <iostream>
using namespace std;
class Student //定义 Student 类
{public :
    Student(int n,int a,float s):num(n),age(a),score(s){ } //定义构造函数 void total(); static float average(); //声明静态成员函数
private :
    int num;
    int age;
    float score;
    static float sum; //静态数据成员
    static int count; //静态数据成员
};
void Student::total() //定义非静态成员函数
{sum+=score; //累加总分
count++; //累计已统计的人数}
float Student::average() //定义静态成员函数
{return (sum/count); }
float Student::sum=0; //对静态数据成员初始化
int Student::count=0; //对静态数据成员初始化
int main()
{
    Student stud[3]={ //定义对象数组并初始化
        Student(1001,18,70), Student(1002,19,78), Student(1005,20,98) };
    int n;
    cout<<"please input the number of students:";
    cin>>n; //输入要求前面多少名学生的平均成绩
    for(int i=0;i<n;i++) //调用 3 次 total 函数
        stud[i].total();
    cout<<"the average score of "<<n<<" students is "<<Student::average()<<endl; //调用静态成员函数
    return 0;
}
```

运行结果为

please input the number of students:3 ✓



the average score of 3 students is 82.3333

说明:

(1) 在主函数中定义了 stud 对象数组, 为了使程序简练, 只定义它含 3 个元素, 分别存放 3 个学生的数据。程序的作用是先求用户指定的 n 名学生的总分, 然后求平均成绩(n 由用户输入)。

(2) 在 Student 类中定义了两个静态数据成员 sum(总分)和 count(累计需要统计的学生人数), 这是由于这两个数据成员的值是需要进行累加的, 它们并不是只属于某一个对象元素, 而是由各对象元素共享的, 可以看出: 它们的值是在不断变化的, 而且无论对哪个对象元素而言, 都是相同的, 而且始终不释放内存空间。

(3) total 是公有的成员函数, 其作用是将一个学生的成绩累加到 sum 中。

公有的成员函数可以引用本对象中的一般数据成员(非静态数据成员), 也可以引用类中的静态数据成员。score 是非静态数据成员, sum 和 count 是静态数据成员。

(4) average 是静态成员函数, 它可以直接引用私有的静态数据成员(不必加类名或对象名), 函数返回成绩的平均值。

(5) 在 main 函数中, 引用 total 函数要加对象名(今用对象数组元素名), 引用静态成员函数 average 函数要用类名或对象名。

(6) 请思考: 如果不将 average 函数定义为静态成员函数行不行? 程序能否通过编译? 需要作什么修改? 为什么要用静态成员函数? 请分析其理由。

(7) 如果想在 average 函数中引用 stud[1]的非静态数据成员 score, 应该怎样处理? 以上是在例 9.11 的基础上顺便说明静态成员函数引用非静态数据成员的方法, 以帮助读者理解。

但是在 C++ 程序中最好养成这样的习惯: 只用静态成员函数引用静态数据成员, 而不引用非静态数据成员。

这样思路清晰, 逻辑清楚, 不易出错。

9.10 友元在一个类中可以有公用的(public)成员和私有的(private)成员。

在类外可以访问公用成员, 只有本类中的函数可以访问本类的私有成员。

现在, 我们来补充介绍一个例外——友元(friend)。

友元可以访问与其有好友关系的类中的私有成员。

友元包括友元函数和友元类。

9.10.1 友元函数如果在类本类以外的其他地方定义了一个函数(这个函数可以是不属于任何类的非成员函数, 也可以是其他类的成员函数), 在类体中用 friend 对其进行声明, 此函数就称为本类的友元函数。

友元函数可以访问这个类中的私有成员。

1. 将普通函数声明为友元函数通过下面的例子可以了解友元函数的性质和作用。

```
} void display(Time& t) //这是友元函数, 形参 t 是 Time 类对象的引用
```

```
{cout<<t.hour<<":"<<t.minute<<":"<<t.sec<<endl;}
```

```
int main()
```

```
{
```

```
Time t1(10,13,56);
```

```
display(t1); //调用 display 函数, 实参 t1 是 Time 类对象
```

```
return 0;
```

```
}
```

程序输出结果如下:

10:13:56

由于声明了 display 是 Time 类的 friend 函数, 所以 display 函数可以引用 Time 中的私有成员 hour, minute, sec。但注意在引用这些私有数据成员时, 必须加上对象名, 不能写成

```
cout<<hour<<":"<<minute<<":"<<sec<<endl;
```

因为 display 函数不是 Time 类的成员函数, 不能默认引用 Time 类的数据成员, 必须指定要访问的对象。

2. 友元成员函数 friend 函数不仅可以是一般函数(非成员函数), 而且可以是另一个类中的成员函数。

见例 9.13。

例 9.13 友元成员函数的简单应用。

在本例中除了介绍有关友元成员函数的简单应用外，还将用到类的提前引用声明，请读者注意。

```
#include <iostream>
using namespace std;
class Date; //对 Date 类的提前引用声明
class Time //定义 Time 类
{public :
    Time(int,int,int);
    void display(Date &); //display 是成员函数，形参是 Date 类对象的引用
private :
    int hour;
    int minute;
    int sec;
};
class Date //声明 Date 类
{public :
    Date(int,int,int);
    friend void Time::display(Date &); //声明 Time 中的 display 函数为友元成员函数
private :
    int month;
    int day;
    int year;
};
{cout<<d.month<<"/"<<d.day<<"/"<<d.year<<endl; //引用 Date 类对象中的私有数据
cout<<hour<<":"<<minute<<":"<<sec<<endl; //引用本类对象中的私有数据
}
Date::Date(int m,int d,int y) //类 Date 的构造函数
{month=m; day=d; year=y; }
int main()
{
    Time t1(10,13,56); //定义 Time 类对象 t1
    Date d1(12,25,2004); //定义 Date 类对象 d1
    t1.display(d1);
    //调用 t1 中的 display 函数，实参是 Date 类对象 d1
    return 0;
}
```

运行时输出：

12/25/2004 (输出 Date 类对象 d1 中的私有数据)

10:13:56 (输出 Time 类对象 t1 中的私有数据)

在本例中定义了两个类 Time 和 Date。

程序第 3 行是对 Date 类的声明，因为在第 7 行和第 16 行中对 display 函数的声明和定义中要用到类名 Date，而对 Date 类的定义却在其后面。

能否将 Date 类的声明提到前面来呢？也不行，因为在 Date 类中的第 4 行又用到了 Time 类，也要求先声明 Time 类才能使用它。

为了解决这个问题，C++允许对类作“提前引用”的声明，即在正式声明一个类之前，先声明一个类名，表示此类将在稍后声明。

程序第 3 行就是提前引用声明，它只包含类名，不包括类体。

如果没有第 3 行，程序编译就会出错。

在这里简要介绍有关对象提前引用的知识。

在一般情况下，对象必须先声明，然后才能使用它。

但是在特殊情况下(如上面例子所示的那样)，在正式声明类之前，需要使用该类名。

但是应当注意：类的提前声明的使用范围是有限的。

只有在正式声明一个类以后才能用它去定义类对象。

如果在上面程序第 3 行后面增加一行：`Date d1; //企图定义一个对象`会在编译时出错。

因为在定义对象时是要为这些对象分配存储空间的，在正式声明类之前，编译系统无法确定应为对象分配多大的空间。

编译系统只有在“见到”类体后，才能确定应该为对象预留多大的空间。

在对一个类作了提前引用声明后，可以用该类的名字去定义指向该类型对象的指针变量或对象的引用变量(如在本例中，定义了 `Date` 类对象的引用变量)。

这是因为指针变量和引用变量本身的大小是固定的，与它所指向的类对象的大小无关。

请注意程序是在定义 `Time::display` 函数之前正式声明 `Date` 类的。

如果将对 `Date` 类的声明的位置(程序 13~21 行)改到定义 `Time::display` 函数之后，编译就会出错，因为在 `Time::display` 函数体中要用到 `Date` 类的成员 `month, day, year`。

如果不事先声明 `Date` 类，编译系统无法识别成员 `month, day, year` 等成员。

在一般情况下，两个不同的类是互不相干的。

在本例中，由于在 `Date` 类中声明了 `Time` 类中的 `display` 成员函数是 `Date` 类的“朋友”，因此该函数可以引用 `Date` 类中所有的数据。

请注意在本程序中调用友元函数访问有关类的私有数据方法：

- (1) 在函数名 `display` 的前面要加 `display` 所在的对象名(`t1`)；
- (2) `display` 成员函数的实参是 `Date` 类对象 `d1`，否则就不能访问对象 `d1` 中的私有数据；
- (3) 在 `Time::display` 函数中引用 `Date` 类私有数据时必须加上对象名，如 `d.month`。

3. 一个函数(包括普通函数和成员函数)可以被多个类声明为“朋友”，这样就可以引用多个类中的私有数据例如，可以将例 9.13 程序中的 `display` 函数不放在 `Time` 类中，而作为类外的普通函数，然后分别在 `Time` 和 `Date` 类中将 `display` 声明为朋友。

在主函数中调用 `display` 函数，`display` 函数分别引用 `Time` 和 `Date` 两个类的对象的私有数据，输出年、月、日和时、分、秒。

### 9.10.2 友元类

不仅可以将一个函数声明为一个类的“朋友”，而且可以将一个类(例如 `B` 类)声明为另一个类(例如 `A` 类)的“朋友”。

这时 `B` 类就是 `A` 类的友元类。

友元类 `B` 中的所有函数都是 `A` 类的友元函数，可以访问 `A` 类中的所有成员。

在 `A` 类的定义体中用以下语句声明 `B` 类为其友元类：`friend B;`

声明友元类的一般形式为

`friend` 类名;

关于友元，有两点需要说明：

- (1) 友元的关系是单向的而不是双向的。
- (2) 友元的关系不能传递。

在实际工作中，除非确有必要，一般并不把整个类声明为友元类，而只将确实有需要的成员函数声明为友元函数，这样更安全一些。

关于友元利弊的分析：面向对象程序设计的一个基本原则是封装性和信息隐蔽，而友元却可以访问其他类中的私有成员，不能不说这是对封装原则的一个小的破坏。

但是它能有助于数据共享，能提高程序的效率，在使用友元时，要注意到它的副作用，不要过多地使用友元，只有在使用它能使程序精炼，并能大大提高程序的效率时才用友元。



```

#include <iostream>
using namespace std;
class Box
{public :
Box(int ,int);
int volume( );
static int height; //把 height 定义为公用的静态的数据成员
int width;
int length;
};
Box::Box(int w,int len) //通过构造函数对 width 和 length 赋初值
{width=w;
length=len;
}
int Box::volume( )
{return (height*width*length);
}
int Box::height=10; //对静态数据成员 height 初始化
1244
1245
int main( )
{
Box a(15,20),b(20,30);

```

例 9.12 友元函数的简单例子。

```

#include <iostream>
using namespace std;
class Time
{public :
Time(int ,int,int );
friend void display(Time &); //声明 display 函数为 Time 类的友元函数
private : //以下数据是私有数据成员
int hour;
int minute;
int sec;
};

Time::Time(int h,int m,int s) //构造函数，给 hour,minute,sec 赋初值
{hour=h;

minute=m;
sec=s;

```

```
Time::Time(int h,int m,int s) //类 Time 的构造函数
```

```
{hour=h;  
minute=m;  
sec=s;  
}
```

```
void Time::display(Date &d) //display 的作用是输出年、月、日和时、分、秒
```

### 9.11 类模板

有时，有两个或多个类，其功能是相同的，仅仅是数据类型不同，如下面语句声明了一个类：

```
class Compare_int  
{public :  
Compare(int a,int b)  
{x=a;y=b;}  
int max()  
{return (x>y)?x:y;}  
int min()  
{return (x<y)?x:y;}  
private :  
int x,y;  
};
```

其作用是对两个整数作比较，可以通过调用成员函数 max 和 min 得到两个整数中的大者和小者。

如果想对两个浮点数(float 型)作比较，需要另外声明一个类：

```
class Compare_float  
{public :  
Compare(float a,float b)  
{x=a;y=b;}  
float max()  
{return (x>y)?x:y;}  
float min()  
{return (x<y)?x:y;}  
private :  
float x,y;  
}
```

显然这基本上是重复性的工作，应该想办法减少重复的工作。

C++在发展的后期增加了模板(template)的功能，提供了解决这类问题的途径。

可以声明一个通用的类模板，它可以有一个或多个虚拟的类型参数，

如对以上两个类可以综合写出以下的类模板：

```
template <class numtype> //声明一个模板，虚拟类型名为 numtype  
class Compare //类模板名为 Compare  
{public :  
Compare(numtype a,numtype b)  
{x=a;y=b;}  
numtype max()  
{return (x>y)?x:y;}  
numtype min()  
}
```

```
{return (x<y)?x:y;}
private :
numtype x,y;
};
```

请将此类模板和前面第一个 Compare\_int 类作一比较，可以看到有两处不同：

(1) 声明类模板时要增加一行

template <class 类型参数名>

(2) 原有的类型名 int 换成虚拟类型参数名 numtype。

在建立类对象时，如果将实际类型指定为 int 型，编译系统就会用 int 取代所有的 numtype，如果指定为 float 型，就用 float 取代所有的 numtype。

这样就能实现“一类多用”。

由于类模板包含类型参数，因此又称为参数化的类。

如果说类是对象的抽象，对象是类的实例，则类模板是类的抽象，类是类模板的实例。

利用类模板可以建立含各种数据类型的类。

在声明了一个类模板后，怎样使用它？怎样使它变成一个实际的类？

先回顾一下用类来定义对象的方法：

Compare\_int cmp1(4,7); // Compare\_int 是已声明的类

用类模板定义对象的方法与此相似，但是不能直接写成

Compare cmp(4,7); // Compare 是类模板名

Compare 是类模板名，而不是一个具体的类，类模板体中的类型 numtype 并不是一个实际的类型，只是一个虚拟的类型，无法用它去定义对象。

必须用实际类型名去取代虚拟的类型，

具体的做法是：

Compare <int> cmp(4,7);

即在类模板名之后在尖括号内指定实际的类型名，在进行编译时，编译系统就用 int 取代类模板中的类型参数 numtype，这样就把类模板具体化了，或者说实例化了。

这时 Compare<int>就相当于前面介绍的 Compare\_int 类。

例 9.14 是一个完整的例子。

例 9.14 声明一个类模板，利用它分别实现两个整数、浮点数和字符的比较，求出大数和小数。

```
#include <iostream>
using namespace std;
template <class numtype>
//定义类模板
class Compare
{public :
Compare(numtype a,numtype b)
{x=a;y=b;}
numtype max( )
{return (x>y)?x:y;}
numtype min( )
{return (x<y)?x:y;}
private :
numtype x,y;
};
int main( )
```

```

{Compare<int> cmp1(3,7);
//定义对象 cmp1, 用于两个整数的比较
cout<<cmp1.max()<<" is the Maximum of two integer numbers."<<endl;
cout<<cmp1.min()<<" is the Minimum of two integer numbers."<<endl<<endl;
Compare<float> cmp2(45.78,93.6); //定义对象 cmp2, 用于两个浮点数的比较
cout<<cmp2.max()<<" is the Maximum of two float numbers."<<endl;
cout<<cmp2.min()<<" is the Minimum of two float numbers."<<endl<<endl;
Compare<char> cmp3('a','A'); //定义对象 cmp3, 用于两个字符的比较
cout<<cmp3.max()<<" is the Maximum of two characters."<<endl;
cout<<cmp3.min()<<" is the Minimum of two characters."<<endl;
return 0;
}

```

运行结果如下:

```

7 is the Maximum of two integers.
3 is the Minimum of two integers.
93.6 is the Maximum of two float numbers.
45.78 is the Minimum of two float numbers.
a is the Maximum of two characters.
A is the Minimum of two characters.

```

还有一个问题要说明: 上面列出的类模板中的成员函数是在类模板内定义的。

如果改为在类模板外定义, 不能用一般定义类成员函数的形式:

```
numtype Compare::max() {...} //不能这样定义类模板中的成员函数
```

而应当写成类模板的形式:

```

template <class numtype>
numtype Compare<numtype>::max( )
{ {return (x>y)?x:y;}

```

归纳以上的介绍, 可以这样声明和使用类模板:

(1) 先写出一个实际的类。

由于其语义明确, 含义清楚, 一般不会出错。

(2) 将此类中准备改变的类型名(如 int 要改变为 float 或 char)

改用一个自己指定的虚拟类型名(如上例中的 numtype)。

(3) 在类声明前面加入一行, 格式为

```

template <class 虚拟类型参数>, 如
template <class numtype> //注意本行末尾无分号

```

```
class Compare
```

```
{...}; //类体
```

(4) 用类模板定义对象时用以下形式:

类模板名<实际类型名> 对象名;

类模板名<实际类型名> 对象名(实参表列);

如

```
Compare<int> cmp;
```

```
Compare<int> cmp(3,7);
```

(5) 如果在类模板外定义成员函数, 应写成类模板形式:

```
template <class 虚拟类型参数>
```

```
函数类型 类模板名<虚拟类型参数>::成员函数名(函数形参表列) {...}
```

说明:

(1) 类模板的类型参数可以有一个或多个, 每个类型前面都必须加 class, 如

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

```
class someclass
```

```
{...};
```

在定义对象时分别代入实际的类型名，如

```
someclass<int,double> obj;
```

(2) 和使用类一样，使用类模板时要注意其作用域，只能在其有效作用域内用它定义对象。

(3) 模板可以有层次，一个类模板可以作为基类，派生出派生模板类。

## 第 10 章 运算符重载

### 10.1 什么是运算符重载

### 10.2 运算符重载的方法

### 10.3 重载运算符的规则

### 10.4 运算符重载函数作为类成员函数和友元函数

### 10.5 重载双目运算符

### 10.6 重载单目运算符

### 10.7 重载流插入运算符和流提取运算符

### 10.8 不同类型数据间的转换

### 第 10 章 运算符重载

所谓重载,就是重新赋予新的含义。函数重载就是对一个已有的函数赋予新的含义,使之实现新功能。

运算符也可以重载。实际上,我们已经在不知不觉之中使用了运算符重载。

现在要讨论的问题是:用户能否根据自己的需要对 C++已提供的运算符进行重载,赋予它们新的含义,使之一名多用。譬如,能否用“+”号进行两个复数的相加。在 C++中不能在程序中直接用运算符“+”对复数进行相加运算。用户必须自己设法实现复数相加。例如用户可以通过定义一个专门的函数来实现复数相加。见例 10.1。

### 10.1 什么是运算符重载

### 第 10 章 运算符重载

例 10.1 通过函数来实现复数相加。

```
#include <iostream>
```

```
using namespace std;
```

```

class Complex //定义 Complex 类
{public:
Complex() {real=0;imag=0;} //定义构造函数
Complex(double r,double i){real=r;imag=i;} //构造函数重载
Complex complex_add(Complex &c2); //声明复数相加函数
void display(); //声明输出函数
private:
double real; //实部
double imag; //虚部
};

```

```

Complex Complex::complex_add(Complex &c2)

```

```

{Complex c;
c.real=real+c2.real;
第 10 章 运算符重载
c.imag=imag+c2.imag;
return c;}

```

```

void Complex::display() //定义输出函数
{cout<<" (" <<real<<" , " <<imag<<" i)" <<endl;}

```

```

int main()
{Complex c1(3,4),c2(5,-10),c3;//定义 3 个复数对象
c3=c1.complex_add(c2); //调用复数相加函数
cout<<" c1=" ; c1.display(); //输出 c1 的值
cout<<" c2=" ; c2.display(); //输出 c2 的值
cout<<" c1+c2=" ; c3.display(); //输出 c3 的值
return 0;
}

```

运行结果如下:

```

c1=(3+4i)
c2=(5-10i)
c1+c2=(8,-6i)

```

### 第 10 章 运算符重载

结果无疑是正确的,但调用方式不直观、太烦琐,使人感到很不方便。能否也和整数的加法运算一样,直接用加号“+”来实现复数运算呢?如

```

c3=c1+c2;

```

编译系统就会自动完成 c1 和 c2 两个复数相加的运算。如果能做到,就为对象的运算提供了很大的方便。这就需要运算符“+”进行重载。

### 第 10 章 运算符重载

运算符重载的方法是定义一个重载运算符的函数,在需要执行被重载的运算符时,系统就自动调用该函数,以实现相应的运算。也就是说,运算符重载是通过定义函数实现的。运算符重载实质上是函数的重载。

重载运算符的函数一般格式如下:

函数类型 operator 运算符名称 (形参表列)

{ 对运算符的重载处理 }

例如,想将“+”用于 Complex 类(复数)的加法运算,函数的原型可以是这样的:

```

Complex operator+ (Complex& c1,Complex& c2);

```

## 10.2 运算符重载的方法

### 第 10 章 运算符重载

在定义了重载运算符的函数后,可以说: 函数 `operator+` 重载了运算符`+`。

为了说明在运算符重载后,执行表达式就是调用函数的过程,可以把两个整数相加也想像为调用下面的函数:

```
int operator + (int a,int b)
```

```
{return (a+b);}
```

如果有表达式 `5+8`,就调用此函数,将 5 和 8 作为调用函数时的实参,函数的返回值为 13。这就是用函数的方法理解运算符。

可以在例 10.1 程序的基础上重载运算符`+`,使之用于复数相加。

### 第 10 章 运算符重载

例 10.2 改写例 10.1,重载运算符`+`,使之能用于两个复数相加。

```
#include <iostream>
```

```
using namespace std;
```

```
class Complex
```

```
{public:
```

```
Complex(){real=0;imag=0;}
```

```
Complex(double r,double i){real=r;imag=i;}
```

```
Complex operator+(Complex &c2);//声明重载运算符的函数
```

```
void display();
```

```
private:
```

```
double real;
```

```
double imag;
```

```
};
```

```
Complex Complex::operator+(Complex &c2) //定义重载运算符的函数
```

```
{ Complex c;
```

```
c.real=real+c2.real;
```

```
c.imag=imag+c2.imag;
```

### 第 10 章 运算符重载

```
return c;}
```

```
void Complex::display( )
```

```
{ cout<<" (" <<real<<" , " <<imag<<" i)" <<endl;}
```

```
int main( )
```

```
{ Complex c1(3,4),c2(5,-10),c3;
```

```
c3=c1+c2; //运算符+用于复数运算
```

```
cout<<" c1=" ;c1.display( );
```

```
cout<<" c2=" ;c2.display( );
```

```
cout<<" c1+c2=" ;c3.display( );
```

```
return 0;
```

```
}
```

运行结果与例 10.1 相同:

```
c1=(3+4i)
```

```
c2=(5-10i)
```

```
c1+c2=(8,-6i)
```

### 第 10 章 运算符重载



请比较例 10.1 和例 10.2,只有两处不同:

(1) 在例 10.2 中以 `operator+` 函数取代了例 10.1 中的 `complex_add` 函数,而且只是函数名不同,函数体和函数返回值的类型都是相同的。

(2) 在 `main` 函数中,以“`c3=c1+c2;`”取代了例 10.1 中的“`c3=c1.complex_add(c2);`”。在将运算符+重载为类的成员函数后,C++编译系统将程序中的表达式 `c1+c2` 解释为

`c1.operator+(c2)` //其中 `c1` 和 `c2` 是 `Complex` 类的对象

即以 `c2` 为实参调用 `c1` 的运算符重载函数 `operator+(Complex &c2)`,进行求值,得到两个复数之和。

## 第 10 章 运算符重载

虽然重载运算符所实现的功能完全可以用函数实现,但是使用运算符重载能使用户程序易于编写、阅读和维护。在实际工作中,类的声明和类的使用往往是分离的。假如在声明 `Complex` 类时,对运算符`+`、`-`、`*`、`/`都进行了重载,那么使用这个类的用户在编程时可以完全不考虑函数是怎么实现的,放心大胆地直接使用`+`、`-`、`*`、`/`进行复数的运算即可,十分方便。

对上面的运算符重载函数 `operator+` 还可以改写得简练一些:

```
Complex Complex::operator + (Complex &c2)
```

```
{return Complex(real+c2.real, imag+c2.imag);}
```

需要说明的是:运算符被重载后,其原有的功能仍然保留,没有丧失或改变。

## 第 10 章 运算符重载

通过运算符重载,扩大了 C++已有运算符的作用范围,使之能用于类对象。

运算符重载对 C++有重要的意义,把运算符重载和类结合起来,可以在 C++程序中定义出很有实用意义而使用方便的新的数据类型。运算符重载使 C++具有更强大的功能、更好的可扩充性和适应性,这是 C++最吸引人的特点之一。

## 第 10 章 运算符重载

(1) C++不允许用户自己定义新的运算符,只能对已有的 C++运算符进行重载。

(2) C++允许重载的运算符

C++中绝大部分的运算符允许重载。具体规定见书中表 10.1。

不能重载的运算符只有 5 个:

· (成员访问运算符)

·\* (成员指针访问运算符)

:: (域运算符)

sizeof(长度运算符)

?: (条件运算符)

## 10.3 重载运算符的规则

## 第 10 章 运算符重载

前两个运算符不能重载是为了保证访问成员的功能不能被改变,域运算符和 `sizeof` 运算符的运算对象是类型而不是变量或一般表达式,不具重载的特征。

(3) 重载不能改变运算符运算对象(即操作数)的个数。

(4) 重载不能改变运算符的优先级别。

(5) 重载不能改变运算符的结合性。

(6) 重载运算符的函数不能有默认的参数,否则就改变了运算符参数的个数,与前面第(3)点矛盾。

(7) 重载的运算符必须和用户定义的自定义类型的对象一起使用,其参数至少应有一个是类对象(或类对象的引用)。也就是说,参数不能全部是 C++的标准类型,以防止用户修改用于标准类型数据的运算符的性质。

## 第 10 章 运算符重载

(8) 用于类对象的运算符一般必须重载,但有两个例外,运算符“=”和“&”不必用户重载。

① 赋值运算符(=)可以用于每一个类对象,可以利用它在同类对象之间相互赋值。

② 地址运算符&也不必重载,它能返回类对象在内存中的起始地址。

(9) 应当使重载运算符的功能类似于该运算符作用于标准类型数据时所实现的功能。

(10) 运算符重载函数可以是类的成员函数(如例 10.2),也可以是类的友元函数,还可以是既非类的成员函数



也不是友元函数的普通函数。

## 第 10 章 运算符重载

在本章例 10.2 程序中对运算符“+”进行了重载,使之能用于两个复数的相加。在该例中运算符重载函数 `operator+` 作为 `Complex` 类中的成员函数。

“+”是双目运算符,为什么在例 10.2 程序中的重载函数中只有一个参数呢?实际上,运算符重载函数有两个参数,由于重载函数是 `Complex` 类中的成员函数,有一个参数是隐含的,运算符函数是用 `this` 指针隐式地访问类对象的成员。

### 10.4 运算符重载函数作为类成员函数和友元函数

## 第 10 章 运算符重载

可以看到,重载函数 `operator+` 访问了两个对象中的成员,一个是 `this` 指针指向的对象中的成员,一个是形参对象中的成员。如 `this->real+c2.real`, `this->imag` 就是 `c1.real`。

在 10.2 节中已说明,在将运算符函数重载为成员函数后,如果出现含该运算符的表达式,如 `c1+c2`,编译系统把它解释为

`c1.operator+(c2)`

即通过对象 `c1` 调用运算符重载函数,并以表达式中第二个参数(运算符右侧的类对象 `c2`)作为函数实参。运算符重载函数的返回值是 `Complex` 类型,返回值是复数 `c1` 和 `c2` 之和 (`Complex(c1.real + c2.real, c1.imag+c2.imag)`)。

## 第 10 章 运算符重载

运算符重载函数除了可以作为类的成员函数外,还可以是非成员函数。可以将例 10.2 改写为例 10.3。

例 10.3 将运算符“+”重载为适用于复数加法,重载函数不作为成员函数,而放在类外,作为 `Complex` 类的友元函数。

```
#include <iostream>
using namespace std;
class Complex
{public:
    Complex(){real=0;imag=0;}
    Complex(double r,double i){real=r;imag=i;}
    friend Complex operator + (Complex &c1,Complex &c2); //重载函数作为友元函数
    void display();
    private:
    double real;
    double imag;
};

第 10 章 运算符重载
```

```
Complex operator + (Complex &c1,Complex &c2) //定义作为友元函数的重载函数
{return Complex(c1.real+c2.real, c1.imag+c2.imag);}
```

```
void Complex::display()
{cout<<" (" <<real<<" , " <<imag<<" i)" <<endl;}
int main()
{Complex c1(3,4),c2(5,-10),c3;
c3=c1+c2;
cout<<" c1=" ; c1.display();
cout<<" c2=" ; c2.display();
cout<<" c1+c2=" ; c3.display();
}
```

## 第 10 章 运算符重载

与例 10.2 相比较,只作了一处改动,将运算符函数不作为成员函数,而把它放在类外,在 `Complex` 类中声明它为友元函数。同时将运算符函数改为有两个参数。在将运算符“+”重载为非成员函数后,C++编译系统将程序中的表达式 `c1+c2` 解释为

```
operator+(c1,c2)
```

即执行 `c1+c2` 相当于调用以下函数:

```
Complex operator + (Complex &c1,Complex &c2)
{return Complex(c1.real+c2.real, c1.imag+c2.imag);}
```

求出两个复数之和。运行结果同例 10.2。

为什么把运算符函数作为友元函数呢?因为运算符函数要访问 `Complex` 类对象中的成员。如果运算符函数不是 `Complex` 类的友元函数,而是一个普通的函数,它是没有权利访问 `Complex` 类的私有成员的。

## 第 10 章 运算符重载

在 10.2 节中曾提到过:运算符重载函数可以是类的成员函数,也可以是类的友元函数,还可以是既非类的成员函数也不是友元函数的普通函数。现在分别讨论这 3 种情况。

首先,只有在极少数的情况下才使用既不是类的成员函数也不是友元函数的普通函数,原因是上面提到的,普通函数不能直接访问类的私有成员。

在剩下的两种方式中,什么时候应该用成员函数方式,什么时候应该用友元函数方式?二者有何区别呢?如果将运算符重载函数作为成员函数,它可以通过 `this` 指针自由地访问本类的数据成员,因此可以少写一个函数的参数。但必须要求运算表达式第一个参数(即运算符左侧的操作数)是一个类对象,

## 第 10 章 运算符重载

而且与运算符函数的类型相同。因为必须通过类的对象去调用该类的成员函数,而且只有运算符重载函数返回值与该对象同类型,运算结果才有意义。在例 10.2 中,表达式 `c1+c2` 中第一个参数 `c1` 是 `Complex` 类对象,运算符函数返回值的类型也是 `Complex`,这是正确的。如果 `c1` 不是 `Complex` 类,它就无法通过隐式 `this` 指针访问 `Complex` 类的成员了。如果函数返回值不是 `Complex` 类复数,显然这种运算是没有实际意义的。

如想将一个复数和一个整数相加,如 `c1+i`,可以将运算符重载函数作为成员函数,如下面的形式:

```
Complex Complex::operator+(int &i) //运算符重载函数作为Complex类的成员函数
{return Complex(real+i,imag);}
```

## 第 10 章 运算符重载

注意在表达式中重载的运算符“+”左侧应为 `Complex` 类的对象,如

```
c3=c2+i;
```

不能写成

```
c3=i+c2; //运算符“+”的左侧不是类对象,编译出错
```

如果出于某种考虑,要求在使用重载运算符时运算符左侧的操作数是整型量(如表达式 `i+c2`,运算符左侧的操作数 `i` 是整数),这时是无法利用前面定义的重载运算符的,因为无法调用 `i.operator+` 函数。可想而知,如果运算符左侧的操作数属于 C++标准类型(如 `int`)或是一个其他类的对象,则运算符重载函数不能作为成员函数,只能作为非成员函数。如果函数需要访问类的私有成员,则必须声明为友元函数。可以在 `Complex` 类中声明:

## 第 10 章 运算符重载

```
friend Complex operator+(int &i,Complex &c); //第一个参数可以不是类对象
```

在类外定义友元函数:

```
Complex operator+(int &i, Complex &c) //运算符重载函数不是成员函数
{return Complex(i+c.real,c.imag);}
```

将双目运算符重载为友元函数时,在函数的形参表列中必须有两个参数,不能省略,形参的顺序任意,不要求第一个参数必须为类对象。但在使用运算符的表达式中,要求运算符左侧的操作数与函数第一个参数对应,运算符右侧的操作数与函数的第二个参数对应。如

```
c3=i+c2; //正确,类型匹配
```

```
c3=c2+i; //错误,类型不匹配
```

## 第 10 章 运算符重载

请注意,数学上的交换律在此不适用。如果希望适用交换律,则应再重载一次运算符“+”。如

**Complex operator+(Complex &c, int &i) //此时第一个参数为类对象**

**{return Complex(i+c.real,c.imag);}**

这样,使用表达式 `i+c2` 和 `c2+i` 都合法,编译系统会根据表达式形式选择调用与之匹配的运算符重载函数。可以将以上两个运算符重载函数都作为友元函数,也可以将一个运算符重载函数(运算符左侧为对象名的)作为成员函数,另一个(运算符左侧不是对象名的)作为友元函数。但不可能将两个都作为成员函数,原因是显然的。

C++规定,有的运算符(如赋值运算符、下标运算符、函数调用运算符)必须定义为类的成员函数,有的运算符则不能定义为类的成员函数(如流插入“<<”和流提取运算符“>>”、类型转换运算符)。

## 第 10 章 运算符重载

由于友元的使用会破坏类的封装,因此从原则上说,要尽量将运算符函数作为成员函数。但考虑到各方面的因素,一般将单目运算符重载为成员函数,将双目运算符重载为友元函数。在学习了本章第 10.7 节例 10.9 的讨论后,读者对此会有更深入的认识。

说明: 有的 C++编译系统(如 Visual C++ 6.0)没有完全实现 C++标准,它所提供不带后缀.h 的头文件不支持把成员函数重载为友元函数。上面例 10.3 程序在 GCC 中能正常运行,而在 Visual C++ 6.0 中会编译出错。但是 Visual C++所提供的老形式的带后缀.h 的头文件可以支持此项功能,因此可以将程序头两行修改如下,即可顺利运行:

**#include <iostream.h>**

以后如遇到类似情况,亦可照此办理。

## 第 10 章 运算符重载

双目运算符(或称二元运算符)是 C++中最常用的运算符。双目运算符有两个操作数,通常在运算符的左右两侧,如 `3+5`,`a=b`,`i<10` 等。在重载双目运算符时,不言而喻在函数中应该有两个参数。下面再举一个例子说明重载双目运算符的应用。

### 10.5 重载双目运算符

## 第 10 章 运算符重载

例 10.4 定义一个字符串类 `String`,用来存放不定长的字符串,重载运算符“==”,“<”和“>”,用于两个字符串的等于、小于和大于的比较运算。

为了使读者便于理解程序,同时也使读者了解建立程序的步骤,下面分几步来介绍编程过程。

(1) 先建立一个 `String` 类:

**#include <iostream>**

**using namespace std;**

**class String**

**{public:**

**String(){p=NULL;} //默认构造函数**

**String(char \*str); //构造函数**

**void display();**

**private:**

**char \*p; //字符型指针,用于指向字符串**

**};**

## 第 10 章 运算符重载

**String::String(char \*str) //定义构造函数**

**{p=str;} //使 p 指向实参字符串**

**void String::display() //输出 p 所指向的字符串**

**{cout<<p;}**

```
int main( )
{String string1(" Hello" ),string2(" Book" );
string1.display();
cout<<endl;
string2.display();
return 0;
}
```

运行结果为

Hello

Book

## 第 10 章 运算符重载

(2) 有了这个基础后,再增加其他必要的内容。现在增加对运算符重载的部分。为便于编写和调试,先重载一个运算符“>”。程序如下:

```
#include <iostream>
#include <string>
using namespace std;
class String
{public:
String(){p=NULL;}
String(char *str);
friend bool operator>(String &string1,String &string2);//声明运算符函数为友元函数
void display();
private:
char *p;//字符型指针,用于指向字符串
};
String::String(char *str)
{p=str;}
```

## 第 10 章 运算符重载

```
void String::display() //输出 p 所指向的字符串
{cout<<p;}
bool operator>(String &string1,String &string2)//定义运算符重载函数
{if(strcmp(string1.p,string2.p)>0)
return true;
else return false;
}
```

```
int main( )
{String string1(" Hello" ),string2(" Book" );
cout<<(string1>string2)<<endl;
}
```

程序运行结果为 1。

这只是一个并不很完善的程序,但是,已经完成了实质性的工作了,运算符重载成功了。其他两个运算符的重载如法炮制即可。

## 第 10 章 运算符重载

(3) 扩展到对 3 个运算符重载。

在 String 类体中声明 3 个成员函数:

```

friend bool operator> (String &string1, String &string2);
friend bool operator< (String &string1, String &string2);
friend bool operator==(String &string1, String& string2);
在类外分别定义 3 个运算符重载函数:
bool operator>(String &string1,String &string2) //对运算符">"重载
{if(strcmp(string1.p,string2.p)>0)
return true;
else
return false;
}

```

```

bool operator<(String &string1,String &string2) //对运算符"<"重载
{if(strcmp(string1.p,string2.p)<0)
return true;
else
第 10 章 运算符重载
return false;
}

```

```

bool operator==(String &string1,String &string2) //对运算符"=="重载
{if(strcmp(string1.p,string2.p)==0)
return true;
else
return false;
}

```

再修改主函数:

```

int main( )
{String string1(" Hello" ),string2(" Book" ),string3(" Computer" );
cout<<(string1>string2)<<endl;//比较结果应该为 true
cout<<(string1<string3)<<endl;//比较结果应该为 false
cout<<(string1==string2)<<endl; //比较结果应该为 false
return 0;
}

```

第 10 章 运算符重载

运行结果为

```

1
0
0

```

结果显然是对的。到此为止,主要任务基本完成。

(4) 再进一步修饰完善,使输出结果更直观。下面给出最后的程序。

```

#include <iostream>
using namespace std;
class String
{public:
String( ){p=NULL;}
String(char *str);
friend bool operator>(String &string1,String &string2);

```

```
friend bool operator<(String &string1,String &string2);
friend bool operator==(String &string1,String &string2);
```

第 10 章 运算符重载

```
void display();
private:
char *p;
};
String::String(char *str)
{p=str;}
```

```
void String::display() //输出 p 所指向的字符串
{cout<<p;}
```

```
bool operator>(String &string1,String &string2)
{if(strcmp(string1.p,string2.p)>0)
return true;
else
return false;
}
```

```
bool operator<(String &string1,String &string2)
第 10 章 运算符重载
{if(strcmp(string1.p,string2.p)<0)
return true;
else
return false;
}
```

```
bool operator==(String &string1,String &string2)
{if(strcmp(string1.p,string2.p)==0)
return true;
else
return false;
}
```

```
void compare(String &string1,String &string2)
{if(operator>(string1,string2)==1)
{string1.display();cout<<" >" ;string2.display();}
else
if(operator<(string1,string2)==1)
第 10 章 运算符重载
{string1.display();cout<<" <" ;string2.display();}
else
if(operator==(string1,string2)==1)
{string1.display();cout<<" =" ;string2.display();}
cout<<endl;
}
```



```
int main( )
{String string1(" Hello" ),string2(" Book" ),string3(" Computer" ),string4(" Hello" );
compare(string1,string2);
compare(string2,string3);
compare(string1,string4);
return 0;
}
```

运行结果为

Hello>Book

Book<Computer

Hello==Hello

## 第 10 章 运算符重载

增加了一个 compare 函数,用来对两个字符串进行比较,并输出相应的信息。这样可以减轻主函数的负担,使主函数简明易读。

通过这个例子,不仅可以学习到有关双目运算符重载的知识,而且还可以学习怎样去编写 C++ 程序。

这种方法的指导思想是:先搭框架,逐步扩充,由简到繁,最后完善。边编程,边调试,边扩充。千万不要企图在一开始时就解决所有的细节。类是可扩充的,可以一步一步地扩充它的功能。最好直接在计算机上写程序,每一步都要上机调试,调试通过了前面一步再做下一步,步步为营。这样编程和调试的效率是比较高的。读者可以试验一下。

## 第 10 章 运算符重载

单目运算符只有一个操作数,如!a,-b,&c,\*p,还有最常用的++i 和--i 等。重载单目运算符的方法与重载双目运算符的方法是类似的。但由于单目运算符只有一个操作数,因此运算符重载函数只有一个参数,如果运算符重载函数作为成员函数,则还可省略此参数。

下面以自增运算符“++”为例,介绍单目运算符的重载。

### 10.6 重载单目运算符

## 第 10 章 运算符重载

例 10.5 有一个 Time 类,包含数据成员 minute(分)和 sec(秒),模拟秒表,每次走一秒,满 60 秒进一分钟,此时秒又从 0 开始算。要求输出分和秒的值。

```
#include <iostream>
using namespace std;
class Time
{public:
Time() {minute=0;sec=0;} //默认构造函数
Time(int m,int s):minute(m),sec(s){} //构造函数重载
Time operator++(); //声明运算符重载函数
void display() {cout<<"minute<<" :<<" <<sec<<endl;} //定义输出时间函数
private:
int minute;
int sec;
};
Time Time::operator++() //定义运算符重载函数
{if(++sec>=60)
第 10 章 运算符重载
{sec-=60; //满 60 秒进 1 分钟
++minute;}
return *this; //返回当前对象值
}
```

```

int main( )
{Time time1(34,0);
for (int i=0;i<61;i++)
{++time1;
time1.display( );}
return 0;
}

```

运行情况如下:

```

34:1
34:2
|
34:59
35:0
35:1 (共输出 61 行)

```

## 第 10 章 运算符重载

可以看到: 在程序中对运算符“++”进行了重载,使它能用于 Time 类对象。“++”和“-”运算符有两种使用方式,前置自增运算符和后置自增运算符,它们的作用是不一样的,在重载时怎样区别这二者呢?

针对“++”和“-”这一特点,C++约定: 在自增(自减)运算符重载函数中,增加一个 int 型形参,就是后置自增(自减)运算符函数。

例 10.6 在例 10.5 程序的基础上增加对后置自增运算符的重载。修改后的程序如下:

```

#include <iostream>
using namespace std;
class Time
{public:
Time( ){minute=0;sec=0;}
Time(int m,int s):minute(m),sec(s){}

```

## 第 10 章 运算符重载

```

Time operator++( );//声明前置自增运算符“++”重载函数
Time operator++(int);//声明后置自增运算符“++”重载函数
void display( ){cout<<minute<<" : " <<sec<<endl;}
private:
int minute;
int sec;
};

```

```

Time Time::operator++( )//定义前置自增运算符“++”重载函数
{if(++sec>=60)
{sec-=60;
++minute;}
return *this;//返回自加后的当前对象
}

```

```

Time Time::operator++(int)//定义后置自增运算符“++”重载函数
{Time temp(*this);
sec++;

```

## 第 10 章 运算符重载

```

if(sec>=60)

```



```

{sec-=60;
++minute;}
return temp; //返回的是自加前的对象
}

int main( )
{Time time1(34,59),time2;
cout<<" time1 : " ;
time1.display();
++time1;
cout<<" ++time1: " ;
time1.display();
time2=time1++; //将自加前的对象的值赋给 time2
cout<<" time1++: " ;
time1.display();
cout<<" time2 : " ;
time2.display(); //输出 time2 对象的值
}

```

## 第 10 章 运算符重载

请注意前置自增运算符“++”和后置自增运算符“++”二者作用的区别。前者是先自加,返回的是修改后的对象本身。后者返回的是自加前的对象,然后对象自加。请仔细分析后置自增运算符重载函数。

运行结果如下:

```

time1 : 34:59(time1 原值)
++time1: 35:0 (执行++time1 后 time1 的值)
time1++: 35:1 (再执行 time1++后 time1 的值)
time2 : 35:0 (time2 保存的是执行 time1++前 time1 的值)

```

可以看到:重载后置自增运算符时,多了一个 int 型的参数,增加这个参数只是为了与前置自增运算符重载函数有所区别,此外没有任何作用。编译系统在遇到重载后置自增运算符时,会自动调用此函数。

## 第 10 章 运算符重载

C++的流插入运算符“<<”和流提取运算符“>>”是 C++在类库中提供的,所有 C++编译系统都在类库中提供输入流类 istream 和输出流类 ostream。cin 和 cout 分别是 istream 类和 ostream 类的对象。在类库提供的头文件中已经对“<<”和“>>”进行了重载,使之作为流插入运算符和流提取运算符,能用来输出和输入 C++标准类型的数据。因此,在本书前面几章中,凡是用“cout<<”和“cin>>”对标准类型数据进行输入输出的,都要用 #include <iostream>把头文件包含到本程序文件中。

### 10.7 重载流插入运算符和流提取运算符

## 第 10 章 运算符重载

用户自己定义的类型的数据,是不能直接用“<<”和“>>”来输出和输入的。如果想用它们输出和输入自己声明的类型的的数据,必须对它们重载。

对“<<”和“>>”重载的函数形式如下:

```

istream & operator >> (istream &,自定义类 &);
ostream & operator << (ostream &,自定义类 &);

```

即重载运算符“>>”的函数的第一个参数和函数的类型都必须是 istream&类型,第二个参数是要进行输入操作的类。重载“<<”的函数的第一个参数和函数的类型都必须是 ostream&类型,第二个参数是要进行输出操作的类。因此,只能将重载“>>”和“<<”的函数作为友元函数或普通的函数,而不能将它们定义为成员函数。

## 第 10 章 运算符重载

在程序中,人们希望能用插入运算符“<<”来输出用户自己声明的类的对象的信息,这就需要重载流插入运算符“<<”。

例 10.7 在例 10.2 的基础上,用重载的“<<”输出复数。

```
#include <iostream>
using namespace std;
class Complex
{public:
Complex() {real=0;imag=0;}
Complex(double r,double i){real=r;imag=i;}
Complex operator + (Complex &c2); //运算符“+”重载为成员函数
friend ostream& operator << (ostream&,Complex&); //运算符“<<”重载为友元函数
private:
```

10.7.1 重载流插入运算符“<<”

## 第 10 章 运算符重载

```
double real;
double imag;
};
```

```
Complex Complex::operator + (Complex &c2)//定义运算符“+”重载函数
{return Complex(real+c2.real,imag+c2.imag);}
ostream& operator << (ostream& output,Complex& c) //定义运算符“<<”重载函数
{output<<" (" <<c.real<<" + " <<c.imag<<" i)" <<endl;
return output;
}
```

```
int main()
{Complex c1(2,4),c2(6,10),c3;
c3=c1+c2;
cout<<c3;
return 0;
}
```

## 第 10 章 运算符重载

(在 Visual C++ 6.0 环境下运行时,需将第一行改为#include <iostream.h>,并删去第 2 行。)

运行结果为

(8+14i)

可以看到在对运算符“<<”重载后,在程序中用“<<”不仅能输出标准类型数据,而且可以输出用户自己定义的类对象。用“cout<<c3”即能以复数形式输出复数对象 c3 的值。形式直观,可读性好,易于使用。

下面对怎样实现运算符重载作一些说明。程序中重载了运算符“<<”,运算符重载函数中的形参 output 是 ostream 类对象的引用,形参名 output 是用户任意起的。分析 main 函数最后第二行:

```
cout<<c3;
```

## 第 10 章 运算符重载

运算符“<<”的左面是 cout,前面已提到 cout 是 ostream 类对象。“<<”的右面是 c3,它是 Complex 类对象。由于已将运算符“<<”的重载函数声明为 Complex 类的友元函数,编译系统把“cout<<c3”解释为

operator<<(cout,c3)

即以 cout 和 c3 作为实参,调用下面的 operator<<函数:

```
ostream& operator<<(ostream& output,Complex& c)
{output<<" (" <<c.real<<" + " <<c.imag<<" i)" <<endl;
return output;}
```

调用函数时,形参 output 成为 cout 的引用,形参 c 成为 c3 的引用。因此调用函数的过程相当于执行:

```
cout<<" (" <<c3.real<<" + " <<c3.imag<<" i)" <<endl; return cout;
```

## 第 10 章 运算符重载

请注意：上一行中的“<<”是 C++ 预定义的流插入符，因为它右侧的操作数是字符串常量和 double 类型数据。执行 cout 语句输出复数形式的信息。然后执行 return 语句。请思考：return output 的作用是什么？回答是能连续向输出流插入信息。output 是 ostream 类的对象，它是实参 cout 的引用，也就是 cout 通过传送地址给 output，使它们二者共享同一段存储单元，或者说 output 是 cout 的别名。因此，return output 就是 return cout，将输出流 cout 的现状返回，即保留输出流的现状。

请问返回到哪里？刚才是在执行

```
cout<<c3;
```

在已知 cout<<c3 的返回值是 cout 的当前值。如果有以下输出：

## 第 10 章 运算符重载

```
cout<<c3<<c2;
```

先处理 cout<<c3，即

```
(cout<<c3)<<c2;
```

而执行(cout<<c3)得到的结果就是具有新内容的流对象 cout，因此，(cout<<c3)<<c2 相当于 cout(新值)<<c2。运算符“<<”左侧是 ostream 类对象 cout，右侧是 Complex 类对象 c2，则再次调用运算符“<<”重载函数，接着向输出流插入 c2 的数据。现在可以理解了为什么 C++ 规定运算符“<<”重载函数的第一个参数和函数的类型都必须是 ostream 类型的引用，就是为了返回 cout 的当前值以便连续输出。

请读者注意区分什么情况下的“<<”是标准类型数据的流插入符，什么情况下的“<<”是重载的流插入符。如

## 第 10 章 运算符重载

```
cout<<c3<<5<<endl;
```

有下划线的是调用重载的流插入符，后面两个“<<”不是重载的流插入符，因为它的右侧不是 Complex 类对象而是标准类型的数据，是用预定义的流插入符处理的。

还有一点要说明：在本程序中，在 Complex 类中定义了运算符“<<”重载函数为友元函数，因此只有在输出 Complex 类对象时才能使用重载的运算符，对其他类型的对象是无效的。如

```
cout<<time1; //time1 是 Time 类对象，不能使用用于 Complex 类的重载运算符
```

## 第 10 章 运算符重载

C++ 预定义的运算符“>>”的作用是从一个输入流中提取数据，如“cin>>i;”表示从输入流中提取一个整数赋给变量 i（假设已定义 i 为 int 型）。重载流提取运算符的目的是希望将“>>”用于输入自定义类型的对象的信息。

### 10.7.2 重载流提取运算符“>>”

## 第 10 章 运算符重载

例 10.8 在例 10.7 的基础上，增加重载流提取运算符“>>”，用“cin>>”输入复数，用“cout<<”输出复数。

```
#include <iostream>
```

```
using namespace std;
```

```
class Complex
```

```
{public:
```

```
friend ostream& operator << (ostream&,Complex&); //声明重载运算符“<<”
```

```
friend istream& operator >> (istream&,Complex&); //声明重载运算符“>>”
```

```
private:
```

```
double real;
```

```
double imag;
```

```
};
```

```
ostream& operator << (ostream& output,Complex& c) //定义重载运算符“<<”
```

```
{output<<" (" <<c.real<<" + " <<c.imag<<" i)" ;
```

```
return output;
```

```
}
```

```
istream& operator >> (istream& input,Complex& c) //定义重载运算符">>"
```

## 第 10 章 运算符重载

```
{cout<<" input real part and imaginary part of complex number:" ;  
input>>c.real>>c.imag;  
return input;  
}  
int main( )  
{Complex c1,c2;  
cin>>c1>>c2;  
cout<<" c1=" <<c1<<endl;  
cout<<" c2=" <<c2<<endl;  
return 0;  
}
```

运行情况如下:

```
input real part and imaginary part of complex number:3 6✓  
input real part and imaginary part of complex number:4 10✓  
c1=(3+6i)  
c2=(4+10i)
```

## 第 10 章 运算符重载

以上运行结果无疑是正确的,但并不完善。在输入复数的虚部为正值时,输出的结果是没有问题的,但是虚部如果是负数,就不理想,请观察输出结果。

```
input real part and imaginary part of complex number:3 6✓  
input real part and imaginary part of complex number:4 -10✓  
c1=(3+6i)  
c2=(4+-10i)
```

根据先调试通过,最后完善的原则,可对程序作必要的修改。将重载运算符“<<”函数修改如下:

```
ostream& operator << (ostream& output,Complex& c)  
{output<<" (" <<c.real;  
if(c.imag>=0) output<<" + " ;//虚部为正数时,在虚部前加"+"号  
output<<c.imag<<" i)" <<endl; //虚部为负数时,在虚部前不加"+"号  
return output;  
}
```

这样,运行时输出的最后一行为 c2=(4-10i) 。

## 第 10 章 运算符重载

通过本章前面几节的讨论,可以看到: 在 C++中,运算符重载是很重要的、很有实用意义的。它使类的设计更加丰富多彩,扩大了类的功能和使用范围,使程序易于理解,易于对对象进行操作,它体现了为用户着想、方便用户使用的思想。有了运算符重载,在声明了类之后,人们就可以像使用标准类型一样来使用自己声明的类。类的声明往往是一劳永逸的,有了好的类,用户在程序中就不必定义许多成员函数去完成某些运算和输入输出的功能,使主函数更加简单易读。好的运算符重载能体现面向对象程序设计思想。

## 第 10 章 运算符重载

在本章的例子中读者应当注意到,在运算符重载中使用引用(reference)的重要性。利用引用作为函数的形参可以在调用函数的过程中不是用传递值的方式进行虚实结合,而是通过传址方式使形参成为实参的别名,因此不生成临时变量(实参的副本),减少了时间和空间的开销。此外,如果重载函数的返回值是对象的引用时,返回的不是常量,而是引用所代表的对象,它可以出现在赋值号的左侧而成为左值(left value),可以被赋值或参与其他操作(如保留 cout 流的当前值以便能连续使用“<<”输出)。但使用引用时要特别小心,因为修改了引用就等于修改了它所代表的对象。

## 第 10 章 运算符重载

在 C++ 中,某些不同类型数据之间可以自动转换,例如

```
int i = 6;  
i = 7.5 + i;
```

编译系统对 7.5 是作为 double 型数处理的,在求解表达式时,先将 6 转换成 double 型,然后与 7.5 相加,得到和为 13.5,在向整型变量 i 赋值时,将 13.5 转换为整数 13,然后赋给 i。这种转换是由 C++ 编译系统自动完成的,用户不需干预。这种转换称为隐式类型转换。

## 10.8 不同类型数据间的转换

### 10.8.1 标准类型数据间的转换

#### 第 10 章 运算符重载

C++ 还提供显式类型转换,程序人员在程序中指定将一种指定的数据转换成另一指定的类型,其形式为 类型名(数据)

如

```
int(89.5)
```

其作用是将 89.5 转换为整型数 89。

对于用户自己声明的类型,编译系统并不知道怎样进行转换。解决这个问题的关键是让编译系统知道怎样去进行这些转换,需要定义专门的函数来处理。

#### 第 10 章 运算符重载

转换构造函数(conversion constructor function) 的作用是将一个其他类型的数据转换成一个类的对象。

先回顾一下以前学习过的几种构造函数:

默认构造函数。以 Complex 类为例,函数原型的形式为

```
Complex(); //没有参数
```

用于初始化的构造函数。函数原型的形式为

```
Complex(double r,double i); //形参表列中一般有两个以上参数
```

用于复制对象的复制构造函数。函数原型的形式为

### 10.8.2 转换构造函数

#### 第 10 章 运算符重载

```
Complex (Complex &c); //形参是本类对象的引用
```

现在又要介绍一种新的构造函数——转换构造函数。

转换构造函数只有一个形参,如

```
Complex(double r) {real=r;imag=0;}
```

其作用是将 double 型的参数 r 转换成 Complex 类的对象,将 r 作为复数的实部,虚部为 0。用户可以根据需要定义转换构造函数,在函数体中告诉编译系统怎样去进行转换。

在类体中,可以有转换构造函数,也可以没有转换构造函数,视需要而定。以上几种构造函数可以同时出现在同一个类中,它们是构造函数的重载。编译系统会根据建立对象时给出的实参的个数与类型选择形参与之匹配的构造函数。

#### 第 10 章 运算符重载

使用转换构造函数将一个指定的数据转换为类对象的方法如下:

- (1) 先声明一个类。
- (2) 在这个类中定义一个只有一个参数的构造函数,参数的类型是需要转换的类型,在函数体中指定转换的方法。
- (3) 在该类的作用域内可以用以下形式进行类型转换:

类名(指定类型的数据)

就可以将指定类型的数据转换为此类的对象。

#### 第 10 章 运算符重载

不仅可以将一个标准类型数据转换成类对象,也可以将另一个类的对象转换成转换构造函数所在的类对象。如可以将一个学生类对象转换为教师类对象,可以在 Teacher 类中写出下面的转换构造函数:

```
Teacher(Student& s){num=s.num;strcpy(name,s.name);sex=s.sex;}
```



但应注意: 对象 s 中的 num,name,sex 必须是公用成员,否则不能被类外引用。

## 第 10 章 运算符重载

用转换构造函数可以将一个指定类型的数据转换为类的对象。但是不能反过来将一个类的对象转换为一个其他类型的数据(例如将一个 Complex 类对象转换成 double 类型数据)。

C++ 提供类型转换函数(type conversion function)来解决这个问题。类型转换函数的作用是将一个类的对象转换成另一类型的数据。如果已声明了一个 Complex 类,可以在 Complex 类中这样定义类型转换函数:

```
operator double()
```

```
{return real;}
```

类型转换函数的一般形式为

### 10.8.3 类型转换函数

## 第 10 章 运算符重载

```
operator 类型名()
```

```
{实现转换的语句}
```

在函数名前面不能指定函数类型,函数没有参数。其返回值的类型是由函数名中指定的类型名来确定的。类型转换函数只能作为成员函数,因为转换的主体是本类的对象。不能作为友元函数或普通函数。

从函数形式可以看到,它与运算符重载函数相似,都是用关键字 operator 开头,只是被重载的是类型名。double 类型经过重载后,除了原有的含义外,还获得新的含义(将一个 Complex 类对象转换为 double 类型数据,并指定了转换方法)。这样,编译系统不仅能识别原有的 double 型数据,而且还会把 Complex 类对象作为 double 型数据处理。

## 第 10 章 运算符重载

那么程序中的 Complex 类对具有双重身份,既是 Complex 类对象,又可作为 double 类型数据。Complex 类对象只有在需要时才进行转换,要根据表达式的上下文来决定。

转换构造函数和类型转换运算符有一个共同的功能: 当需要的时候,编译系统会自动调用这些函数,建立一个无名的临时对象(或临时变量)。

## 第 10 章 运算符重载

例 10.9 使用类型转换函数的简单例子。

```
#include <iostream>
```

```
using namespace std;
```

```
class Complex
```

```
{public:
```

```
Complex(){real=0;imag=0;}
```

```
Complex(double r,double i){real=r;imag=i;}
```

```
operator double() {return real;} //类型转换函数
```

```
private:
```

```
double real;
```

```
double imag;
```

```
};
```

```
int main( )
```

```
{Complex c1(3,4),c2(5,-10),c3;
```

```
double d;
```

```
d=2.5+c1; //要求将一个 double 数据与 Complex 类数据相加
```

```
cout<<d<<endl;
```

```
return 0;
```

```
}
```

## 第 10 章 运算符重载

分析:

(1) 如果在 Complex 类中没有定义类型转换函数 operator double,程序编译将出错。

(2) 如果在 main 函数中加一个语句:

```
c3=c2;
```

由于赋值号两侧都是同一类的数据,是可以合法进行赋值的,没有必要把 c2 转换为 double 型数据。

(3) 如果在 Complex 类中声明了重载运算符“+”函数作为友元函数:

```
Complex operator+ (Complex c1,Complex c2)//定义运算符“+”重载函数
```

```
{return Complex(c1.real+c2.real, c1.imag+c2.imag);}
```

若在 main 函数中有语句

## 第 10 章 运算符重载

```
c3=c1+c2;
```

由于已对运算符“+”重载,使之能用于两个 Complex 类对象的相加,因此将 c1 和 c2 按 Complex 类对象处理,相加后赋值给同类对象 c3。

如果改为

```
d=c1+c2; //d 为 double 型变量
```

将 c1 与 c2 两个类对象相加,得到一个临时的 Complex 类对象,由于它不能赋值给 double 型变量,而又有对 double 的重载函数,于是调用此函数,把临时类对象转换为 double 数据,然后赋给 d。

从前面的介绍可知:对类型的重载和本章开头所介绍的对运算符的重载的概念和方法都是相似的。重载函数都使用关键字 operator。

## 第 10 章 运算符重载

因此,通常把类型转换函数也称为类型转换运算符函数,由于它也是重载函数,因此也称为类型转换运算符重载函数(或称强制类型转换运算符重载函数)。

假如程序中需要对一个 Complex 类对象和一个 double 型变量进行+,-,\*,/等算术运算,以及关系运算和逻辑运算,如果不用类型转换函数,就要对多种运算符进行重载,以便能进行各种运算。这样,是十分麻烦的,工作量较大,程序显得冗长。如果用类型转换函数对 double 进行重载(使 Complex 类对象转换为 double 型数据),就不必对各种运算符进行重载,因为 Complex 类对象可以被自动地转换为 double 型数据,而标准类型的数据的运算,是可以使用系统提供的各种运算符的。

## 第 10 章 运算符重载

例 10.10 包含转换构造函数、运算符重载函数和类型转换函数的程序。

先阅读以下程序,在这个程序中只包含转换构造函数和运算符重载函数。

```
#include <iostream>
```

```
using namespace std;
```

```
class Complex
```

```
{public:
```

```
Complex(){real=0;imag=0;} //默认构造函数
```

```
Complex(double r){real=r;imag=0;}//转换构造函数
```

```
Complex(double r,double i){real=r;imag=i;}//实现初始化的构造函数
```

```
friend Complex operator + (Complex c1,Complex c2); //重载运算符“+”的友元函数
```

```
void display();
```

```
private:
```

```
double real;
```

```
double imag;
```

```
};
```

## 第 10 章 运算符重载

```
Complex operator + (Complex c1,Complex c2)//定义运算符“+”重载函数
```

```
{return Complex(c1.real+c2.real, c1.imag+c2.imag);}
```

```
void Complex::display()
{cout<<" (" <<real<<" , " <<imag<<" i)" <<endl;}
```

```
int main()
{Complex c1(3,4),c2(5,-10),c3;
c3=c1+2.5; //复数与double 数据相加
c3.display();
return 0;
}
```

对程序的分析:

(1) 如果没有定义转换构造函数,则此程序编译出错。

(2) 现在,在类 Complex 中定义了转换构造函数,并具体规定了怎样构成一个复数。由于已重载了算符“+”,在处理表达式  $c1+2.5$  时,编译系统把它解释为

### 第 10 章 运算符重载

```
operator+(c1,2.5)
```

由于 2.5 不是 Complex 类对象,系统先调用转换构造函数  $\text{Complex}(2.5)$ ,建立一个临时的 Complex 类对象,其值为  $(2.5+0i)$ 。上面的函数调用相当于

```
operator+(c1,Complex(2.5))
```

将  $c1$  与  $(2.5+0i)$  相加,赋给  $c3$ 。运行结果为

$(5.5+4i)$

(3) 如果把“ $c3=c1+2.5;$ ”改为  $c3=2.5+c1;$  程序可以通过编译和正常运行。过程与前相同。

从中得到一个重要结论: 在已定义了相应的转换构造函数情况下,将运算符“+”函数重载为友元函数,在进行两个复数相加时,可以用交换律。

### 第 10 章 运算符重载

如果运算符函数重载为成员函数,它的第一个参数必须是本类的对象。当第一个操作数不是类对象时,不能将运算符函数重载为成员函数。如果将运算符“+”函数重载为类的成员函数,交换律不适用。

由于这个原因,一般情况下将双目运算符函数重载为友元函数。单目运算符则多重载为成员函数。

(4) 如果一定要将运算符函数重载为成员函数,而第一个操作数又不是类对象时,只有一个办法能够解决,再重载一个运算符“+”函数,其第一个参数为 double 型。当然此函数只能是友元函数,函数原型为

```
friend operator+(double,Complex &);
```

显然这样做不太方便,还是将双目运算符函数重载为友元函数方便些。

### 第 10 章 运算符重载

(5) 在上面程序的基础上增加类型转换函数:

```
operator double(){return real;}
```

此时 Complex 类的公用部分为

```
public:
```

```
Complex(){real=0;imag=0;}
```

```
Complex(double r){real=r;imag=0;} //转换构造函数
```

```
Complex(double r,double i){real=r;imag=i;}
```

```
operator double(){return real;} //类型转换函数
```

```
friend Complex operator+ (Complex c1,Complex c2); //重载运算符“+”
```

```
void display();
```

其余部分不变。程序在编译时出错,原因是出现二义性。



## 第 4 篇 面向对象的程序设计

第 11 章 继承与派生

第 12 章 多态性与虚函数

第 13 章 输入输出流

第 14 章 C++工具

# 第 11 章 继承与派生

11.1 继承与派生的概念

11.2 派生类的声明方式

11.3 派生类的构成

11.4 派生类成员的访问属性

11.5 派生类的构造函数和析构函数

11.6 多重继承

11.7 基类与派生类的转换

11.8 继承与组合

11.9 继承在软件开发中的重要意义

面向对象程序设计有 4 个主要特点：抽象、封装、继承和多态性。

要较好地进行面向对象程序设计，还必须了解面向对象程序设计另外两个重要特征——继承性和多态性。

在本章中主要介绍有关继承的知识，在第 12 章中将介绍多态性。

面向对象技术强调软件的可重用性(software reusability)。

C++语言提供了类的继承机制，解决了软件重用问题。

11.1 继承与派生的概念

在 C++中可重用性是通过继承(inheritance)这一机制来实现的。

继承是 C++的一个重要组成部分。

一个类中包含了若干数据成员和成员函数。

在不同的类中，数据成员和成员函数是不相同的。

但有时两个类的内容基本相同或有一部分相同。

利用原来声明的类 Student 作为基础，再加上新的内容即可，以减少重复的工作量。

C++提供的继承机制就是为了解决这个问题。

在第 8 章已举了马的例子来说明继承的概念。

见图 11.1 示意。

在 C++中，所谓“继承”就是在一个已存在的类的基础上建立一个新的类。

已存在的类(例如“马”)称为“基类(base class)”或“父类(father class)”。

新建的类(例如“公马”)称为“派生类(derived class)”或“子类(son class)”。

见图 11.2 示意。

图 11.1

图 11.2

一个新类从已有的类那里获得其已有特性，这种现象称为类的继承。

通过继承，一个新建子类从已有的父类那里获得父类的特性。

从另一角度说，从已有的类(父类)产生一个新的子类，称为类的派生。

类的继承是用已有的类来建立专用类的编程技术。

派生类继承了基类的所有数据成员和成员函数，并可以对成员作必要的增加或调整。

一个基类可以派生出多个派生类，每一个派生类又可以作为基类再派生出新的派生类，因此基类和派生类是相对而言的。

以上介绍的是最简单的情况：一个派生类只从一个基类派生，这称为单继承(single inheritance)，这种继承关系所形成的层次是一个树形结构，可以用图 11.3 表示。

图 11.3

请注意图中箭头的方向，在本书中约定，箭头表示继承的方向，从派生类指向基类。

一个派生类不仅可以从一个基类派生，也可以从多个基类派生。

一个派生类有两个或多个基类的称为多重继承(multiple inheritance)，这种继承关系所形成的结构如图 11.4 所示。

图 11.4

关于基类和派生类的关系，可以表述为：派生类是基类的具体化，而基类则是派生类的抽象。

图 11.5

## 11.2 派生类的声明方式

假设已经声明了一个基类 Student，在此基础上通过单继承建立一个派生类 Student1:

```
class Student1: public Student//声明基类是 Student
{public:
void display_1()//新增加的成员函数
{cout<<"age: "<<age<<endl;
cout<<"address: "<<addr<<endl;}
private :
int age;//新增加的数据成员
string addr;//新增加的数据成员
};
```

基类名前面有 public 的称为“公用继承(public inheritance)”。

声明派生类的一般形式为

class 派生类名: [继承方式] 基类名{派生类新增加的成员};

继承方式包括: public (公用的),private (私有的)和 protected(受保护的)，此项是可选的，如果不写此项，则默认为 private(私有的)。

## 11.3 派生类的构成

派生类中的成员包括从基类继承过来的成员和自己增加的成员两大部分。

在基类中包括数据成员和成员函数(或称数据与方法)两部分，派生类分为两大部分：一部分是从基类继承来的成员，另一部分是在声明派生类时增加的部分。

每一部分均分别包括数据成员和成员函数。

图 11.6

实际上，并不是把基类的成员和派生类自己增加的成员简单地加在一起就成为派生类。

构造一个派生类包括以下 3 部分工作:

(1) 从基类接收成员。

派生类把基类全部的成员(不包括构造函数和析构函数)接收过来，也就是说是没有选择的，不能选择接收其中一部分成员，而舍弃另一部分成员。

要求我们根据派生类的需要慎重选择基类，使冗余量最小。

事实上，有些类是专门作为基类而设计的，在设计时充分考虑到派生类的要求。

(2) **调整**从基类接收的成员。

接收基类成员是程序人员不能选择的，但是程序人员可以对这些成员作某些调整。

(3) 在声明派生类时**增加的成员**。

这部分内容是很重要的，它体现了派生类对基类功能的扩展。

要根据需要仔细考虑应当增加哪些成员，精心设计。

此外，在声明派生类时，一般还应当自己定义**派生类的构造函数和析构函数**，因为**构造函数和析构函数是不能从基类继承的**。

派生类是基类定义的延续。

可以先声明一个基类，在此基类中只提供某些最基本的功能，而另外有些功能并未实现，然后在声明派生类时加入某些具体的功能，形成适用于某一特定应用的派生类。

通过对基类声明的延续，将一个抽象的基类转化成具体的派生类。

因此，**派生类是抽象基类的具体实现**。

#### 11.4 派生类成员的访问属性

既然派生类中包含基类成员和派生类自己增加的成员，就产生了这两部分成员的关系和访问属性的问题。在建立派生类的时候，并不是简单地把基类的私有成员直接作为派生类的私有成员，把基类的公用成员直接作为派生类的公用成员。

实际上，对基类成员和派生类自己增加的成员是按不同的原则处理的。

具体说，在讨论访问属性时，要考虑以下几种情况：

- (1) 基类的成员函数访问基类成员。
- (2) 派生类的成员函数访问派生类自己增加的成员。
- (3) 基类的成员函数访问派生类的成员。
- (4) 派生类的成员函数访问基类的成员。
- (5) 在派生类外访问派生类的成员。
- (6) 在派生类外访问基类的成员。

对于第(1)和第(2)种情况，比较简单，按第8章介绍过的规则处理，即：基类的成员函数可以访问基类成员，派生类的成员函数可以访问派生类成员。

私有数据成员只能被同一类中的成员函数访问，公用成员可以被外界访问。

第(3)种情况也比较明确，基类的成员函数只能访问基类的成员，而不能访问派生类的成员。

第(5)种情况也比较明确，在派生类外可以访问派生类的公用成员，而不能访问派生类的私有成员。

对于第(4)和第(6)种情况，就稍微复杂一些，也容易混淆。

这些牵涉到如何确定基类的成员在派生类中的访问属性的问题，不仅要考虑对基类成员所声明的访问属性，还要考虑派生类所声明的对基类的继承方式，根据这两个因素共同决定基类成员在派生类中的访问属性。

前面已提到：在派生类中，对基类的继承方式可以有 **public**(公用的)，**private**(私有的)和 **protected**(保护的)3种。

不同的继承方式决定了**基类成员在派生类中的访问属性**。

简单地说：

(1) **公用继承(public inheritance)**

基类的**公用成员**和**保护成员**在派生类中保持原有访问属性，其**私有成员**仍为基类私有。

(2) **私有继承(private inheritance)**

基类的**公用成员**和**保护成员**在派生类中成了**私有成员**。

其**私有成员**仍为基类私有。

(3) **受保护的继承(protected inheritance)**

基类的**公用成员**和**保护成员**在派生类中成了**保护成员**，其**私有成员**仍为基类私有。

**保护成员**的意思是：不能被外界引用，但可以被派生类的成员引用，具体的用法将在稍后介绍。

#### 11.4.1 公用继承

在定义一个派生类时将基类的继承方式指定为 public 的，称为公用继承，用公用继承方式建立的派生类称为公用派生类(public derived class)，其基类称为公用基类(public base class)。

采用公用继承方式时，基类的公用成员和保护成员在派生类中仍然保持其公用成员和保护成员的属性，而基类的私有成员在派生类中并没有成为派生类的私有成员，它仍然是基类的私有成员，只有基类的成员函数可以引用它，而不能被派生类的成员函数引用，因此就成为派生类中的不可访问的成员。

公用基类的成员在派生类中的访问属性见书中表 11.1。

例 11.1 访问公有基类的成员。

下面写出类的声明部分:

```
Class Student//声明基类
{public : //基类公用成员
void get_value( )
{cin>>num>>name>>sex;}
void display()
{cout<<" num: "<<num<<endl;
cout<<" name: "<<name<<endl;
cout<<" sex: "<<sex<<endl;}
private : //基类私有成员
int num;
    string name;
char sex;
};
```

```
class Student1: public Student //以 public 方式声明派生类 Student1
{public:
void display_1( )
{cout<<" num: "<<num<<endl; //企图引用基类的私有成员，错误
cout<<" name: "<<name<<endl; //企图引用基类的私有成员，错误
cout<<" sex: "<<sex<<endl; //企图引用基类的私有成员，错误
cout<<" age: "<<age<<endl; //引用派生类的私有成员，正确
cout<<" address: "<<addr<<endl;} //引用派生类的私有成员，正确
private:
int age;
string addr;
};
```

由于基类的私有成员对派生类来说是不可访问的，因此在派生类中的 display\_1 函数中直接引用基类的私有数据成员 num，name 和 sex 是不允许的。

只能通过基类的公用成员函数来引用基类的私有数据成员。

可以将派生类 Student1 的声明改为

```
class Student1: public Student//以 public 方式声明派生类 Student1
{public:
void display_1( ) {cout<<" age: "<<age<<endl; //引用派生类的私有成员，正确 cout<<" address: "<<addr<<endl;
//引用派生类的私有成员，正确}
private:
```

```
int age; string addr;
};
```

然后在 main 函数中分别调用基类的 display 函数和派生类中的 display\_1 函数，先后输出 5 个数据。  
可以这样写 main 函数（假设对象 stud 中已有数据）：

```
int main()
{Student1 stud;//定义派生类 Student1 的对象 stud
:
stud.display();//调用基类的公用成员函数，输出基类中 3 个数据成员的值 stud.display_1();//调用派生类公用
成员函数，输出派生类中两个数据成员的值
return 0;
}
```

请根据上面的分析，写出完整的程序，程序中应包括输入数据的函数。

实际上，程序还可以改进，在派生类的 display\_1 函数中调用基类的 display 函数，在主函数中只要写一行:stud.display\_1(); 即可输出 5 个数据。

#### 11.4.2 私有继承

在声明一个派生类时将基类的继承方式指定为 private 的，称为私有继承，用私有继承方式建立的派生类称为私有派生类(private derived class)，其基类称为私有基类(private base class)。

私有基类的公用成员和保护成员在派生类中的访问属性相当于派生类中的私有成员，即派生类的成员函数能访问它们，而在派生类外不能访问它们。

私有基类的私有成员在派生类中成为不可访问的成员，只有基类的成员函数可以引用它们。

一个基类成员在基类中的访问属性和在派生类中的访问属性可能是不同的。

私有基类的成员在私有派生类中的访问属性见书中表 11.2。

图 11.7

图 11.7 表示了各成员在派生类中的访问属性。

对表 11.2 的规定不必死记，只需理解：既然声明为私有继承，就表示将原来能被外界引用的成员隐藏起来，不让外界引用，因此私有基类的公用成员和保护成员理所当然地成为派生类中的私有成员。

私有基类的私有成员按规定只能被基类的成员函数引用，在基类外当然不能访问它们，因此它们在派生类中是隐蔽的，不可访问的。

对于不需要再往下继承的类的功能可以用私有继承方式把它隐蔽起来，这样，下一层的派生类无法访问它的任何成员。

**可以知道：**一个成员在不同的派生层次中的访问属性可能是不同的。

它与继承方式有关。

**例 11.2** 将例 11.1 中的公用继承方式改为用私有继承方式(基类 Student 不改)。

**可以写出私有派生类如下：**

```
class Student1: private Student//用私有继承方式声明派生类 Student1
{public:
void display_1() //输出两个数据成员的值
{cout<<"age: "<<age<<endl; //引用派生类的私有成员，正确
cout<<"address: "<<addr<<endl;} //引用派生类的私有成员，正确
private:
int age; string addr;
};
```

请分析下面的主函数：

```
int main()
{Student1 stud1;//定义一个 Student1 类的对象 stud1
stud1.display();//错误，私有基类的公用成员函数在派生类中是私有函数 stud1.display_1();//正确。Display_1
函数是 Student1 类的公用函数
```



```
stud1.age=18; //错误。外界不能引用派生类的私有成员
return 0;
}
```

可以看到:

- (1) 不能通过派生类对象(如 stud1)引用从私有基类继承过来的任何成员(如 stud1.display()或 stud1.num)。
- (2) 派生类的成员函数不能访问私有基类的私有成员, 但可以访问私有基类的公用成员(如 stud1.display\_1 函数可以调用基类的公用成员函数 display, 但不能引用基类的私有成员 num)。

有没有办法调用私有基类的公用成员函数, 从而引用私有基类的私有成员呢? 有。

**应当注意到:** 虽然在派生类外不能通过派生类对象调用私有基类的公用成员函数, 但可以通过派生类的成员函数调用私有基类的公用成员函数(此时它是派生类中的私有成员函数, 可以被派生类的任何成员函数调用)。

可将上面的私有派生类的成员函数定义改写为

```
void display_1()//输出 5 个数据成员的值
{display(): //调用基类的公用成员函数, 输出 3 个数据成员的值
cout<<"age: "<<age<<endl; //输出派生类的私有数据成员
cout<<"address: "<<addr<<endl;} //输出派生类的私有数据成员
main 函数可改写为
```

```
int main()
{
Student1 stud1;
stud1.display_1();//display_1 函数是派生类 Student1 类的公用函数
return 0;
}
```

这样就能正确地引用私有基类的私有成员。

可以看到, 本例采用的方法是:

- ① 在 main 函数中调用派生类中的公用成员函数 stud1.display\_1;
- ② 通过该公用成员函数调用基类的公用成员函数 display(它在派生类中是私有函数, 可以被派生类中的任何成员函数调用);
- ③ 通过基类的公用成员函数 display 引用基类中的数据成员。

请根据上面的要求, 补充和完善上面的程序, 使之成为完整、正确的程序。

程序中应包括输入数据的函数。

由于私有派生类限制太多, 使用不方便, 一般不常使用。

### 11.4.3 保护成员和保护继承

由 protected 声明的成员称为“受保护的成员”, 或简称“保护成员”。

从类的用户角度来看, 保护成员等价于私有成员。

但有一点与私有成员不同, 保护成员可以被派生类的成员函数引用。

见图 11.8 示意。

图 11.8

如果基类声明了私有成员, 那么任何派生类都是不能访问它们的, 若希望在派生类中能访问它们, 应当把它们声明为保护成员。

如果在一个类中声明了保护成员, 就意味着该类可能要用作基类, 在它的派生类中会访问这些成员。

在定义一个派生类时将基类的继承方式指定为 protected 的, 称为保护继承, 用保护继承方式建立的派生类称为保护派生类(protected derived class), 其基类称为受保护的基类(protected base class), 简称保护基类。

**保护继承的特点是:** 保护基类的公用成员和保护成员在派生类中都成了保护成员, 其私有成员仍为基类私有。

也就是把基类原有的公用成员也保护起来, 不让类外任意访问。

将表 11.1 和表 11.2 综合起来，并增加保护继承的内容，见书中表 11.3。

保护基类的所有成员在派生类中都被保护起来，类外不能访问，其公用成员和保护成员可以被其派生类的成员函数访问。

**比较一下私有继承和保护继承**(也就是比较在私有派生类中和在保护派生类中的访问属性)，可以发现，在直接派生类中，以上两种继承方式的作用实际上是相同的：在类外不能访问任何成员，而在派生类中可以通过成员函数访问基类中的公用成员和保护成员。

但是如果继续派生，在新的派生类中，两种继承方式的作用就不同了。

例如，如果以公用继承方式派生出一个新派生类，原来私有基类中的成员在新派生类中都成为不可访问的成员，无论在派生类内或外都不能访问，而原来保护基类中的公用成员和保护成员在新派生类中为保护成员，可以被新派生类的成员函数访问。

从表 11.3 可知：基类的私有成员被派生类继承后变为不可访问的成员，派生类中的一切成员均无法访问它们。

如果需要在派生类中引用基类的某些成员，应当将基类的这些成员声明为 **protected**，而不要声明为 **private**。

如果善于利用保护成员，可以在类的层次结构中找到数据共享与成员隐蔽之间的结合点。

既可实现某些成员的隐蔽，又可方便地继承，能实现代码重用与扩充。

通过以上的介绍，可以知道：

(1) 在派生类中，成员有 4 种不同的访问属性：

- ① 公用的，派生类内和派生类外都可以访问。
- ② 受保护的，派生类内可以访问，派生类外不能访问，其下一层的派生类可以访问。
- ③ 私有的，派生类内可以访问，派生类外不能访问。
- ④ 不可访问的，派生类内和派生类外都不能访问。

可以用书中表 11.4 表示。

**需要说明的是：**

- ① 这里所列出的成员的访问属性是指在派生类中所获得的访问属性。
- ② 所谓在派生类外部，是指在建立派生类对象的模块中，在派生类范围之外。
- ③ 如果本派生类继续派生，则在不同的继承方式下，成员所获得的访问属性是不同的，在本表中只列出在下一层公用派生类中的情况，如果是私有继承或保护继承，读者可以从表 11.3 中找到答案。

(2) 类的成员在不同作用域中有不同的访问属性，对这一点要十分清楚。

下面通过一个例子说明怎样访问保护成员。

例 11.3 在派生类中引用保护成员。

```
#include <iostream>
#include <string>
using namespace std;
class Student//声明基类
{public :
//基类公用成员
void display();
protected:
//基类保护成员
int num;
string name;
char sex;
};
//定义基类成员函数
void Student::display()
{cout<<"num: "<<num<<endl;
cout<<"name: "<<name<<endl;
```

```

cout<<"sex: "<<sex<<endl;
}
class Student1: protected
Student //用 protected 方式声明派生类 Student1
{public :
void display1(); //派生类公用成员函数
private :
int age; //派生类私有数据成员
string addr; //派生类私有数据成员
};
void Student1::display1() //定义派生类公用成员函数
{cout<<"num: "<<num<<endl; //引用基类的保护成员, 合法
cout<<"name: "<<name<<endl; //引用基类的保护成员, 合法
cout<<"sex: "<<sex<<endl; //引用基类的保护成员, 合法
cout<<"age: "<<age<<endl; //引用派生类的私有成员, 合法
cout<<"address: "<<addr<<endl; //引用派生类的私有成员, 合法}
int main()
{Student1 stud1; //stud1 是派生类 Student1 类的对象
stud1.display1(); //合法, display1 是派生类中的公用成员函数
stud1.num=10023; //错误, 外界不能访问保护成员
return 0;
}

```

在派生类的成员函数中引用基类的保护成员是合法的。

保护成员和私有成员不同之处, 在于把保护成员的访问范围扩展到派生类中。

**注意:** 在程序中通过派生类 Student1 的对象 stud1 的公用成员函数 display1 去访问基类的保护成员 num.name 和 sex, 不要误认为可以通过派生类对象名去访问基类的保护成员。

请补充、修改上面的程序, 使之能正常运行。

程序中应包括输入数据的函数。

私有继承和保护继承方式在使用时需要十分小心, 很容易搞错, 一般不常用。

#### 11.4.4 多级派生时的访问属性

图 11.9

如果有图 11.9 所示的派生关系: 类 A 为基类, 类 B 是类 A 的派生类, 类 C 是类 B 的派生类, 则类 C 也是类 A 的派生类。

类 B 称为类 A 的直接派生类, 类 C 称为类 A 的间接派生类。

类 A 是类 B 的直接基类, 是类 C 的间接基类。

在多级派生的情况下, 各成员的访问属性仍按以上原则确定。

例 11.4 多级派生的访问属性。

如果声明了以下的类:

```

class A //基类
{public:
int i;
protected :
void f2();
int j;
private :
int k;

```



```

};
class B: public A//public 方式
{public :
void f3();
protected :
void f4();
private :
int m;
};
class C: protected B //protected 方式
{public :
void f5();
private :
int n;
};

```

类 A 是类 B 的公用基类，类 B 是类 C 的保护基类。

**各成员在不同类中的访问属性如下：**无论哪一种继承方式，在派生类中是不能访问基类的私有成员的，私有成员只能被本类的成员函数所访问，毕竟派生类与基类不是同一个类。

如果在多级派生时都采用公用继承方式，那么直到最后一级派生类都能访问基类的公用成员和保护成员。

如果采用私有继承方式，经过若干次派生之后，基类的所有的成员已经变成不可访问的了。

如果采用保护继承方式，在派生类外是无法访问派生类中的任何成员的。

而且经过多次派生后，人们很难清楚地记住哪些成员可以访问，哪些成员不能访问，很容易出错。

因此，在实际中，常用的是公用继承。

### 11.5 派生类的构造函数和析构函数

用户在声明类时可以不定义构造函数，系统会自动设置一个默认的构造函数，在定义类对象时会自动调用这个默认的构造函数。

这个构造函数实际上是一个空函数，不执行任何操作。

如果需要对类中的数据成员初始化，应自己定义构造函数。

构造函数的主要作用是对数据成员初始化。

在设计派生类的构造函数时，不仅要考虑派生类所增加的数据成员的初始化，还应当考虑基类的数据成员初始化。

也就是说，希望在执行派生类的构造函数时，使派生类的数据成员和基类的数据成员同时都被初始化。

**解决这个问题的思路是：**在执行派生类的构造函数时，调用基类的构造函数。

#### 11.5.1 简单的派生类的构造函数

任何派生类都包含基类的成员，简单的派生类只有一个基类，**而且只有一级派生**(只有直接派生类，没有间接派生类)，**在派生类的数据成员中不包含基类的对象**(即子对象)。

例 11.5 简单的派生类的构造函数。

```

#include <iostream>
#include<string>
using namespace std;
class Student//声明基类 Student
{public:
Student(int n,string nam,char s) //基类构造函数
{num=n;
name=nam;
sex=s; }
~Student(){} //基类析构函数

```

```

protected : //保护部分
int num;
string name;
char sex ;
};
class Student1: public Student //声明派生类 Student1
{public : //派生类的公用部分
Student1(int n,string nam,char s,int a,string ad):Student(n,nam,s)//派生类构造函数
{age=a; //在函数体中只对派生类新增的数据成员初始化
addr=ad; }
void show( )
{cout<<"num: "<<num<<endl;
cout<<"name: "<<name<<endl;
cout<<"sex: "<<sex<<endl;
cout<<"age: "<<age<<endl;
cout<<"address: "<<addr<<endl<<endl;
}
~Student1(){ } //派生类析构函数
private : //派生类的私有部分
int age;
string addr;
};

int main()
{Student1 stud1(10010,"Wang-li",'f',19,"115 Beijing Road,Shanghai");
Student1 stud2(10011,"Zhang-fun",'m',21,"213 Shanghai Road,Beijing");
stud1.show(); //输出第一个学生的数据
stud2.show(); //输出第二个学生的数据
return 0;
}

```

运行结果为

```

num:10010
name:Wang-li
sex:f
address: 115
Beijing Road,Shanghai
num:10011
name:Zhang-fun
sex:m
address: 213
Shanghai Road,Beijing

```

请注意派生类构造函数首行的写法: `Student1(int n, string nam, char s, int a, string ad):Student(n, nam, s)`  
 其一般形式为

派生类构造函数名（总参数表列）: 基类构造函数名（参数表列）{派生类中新增数据成员初始化语句}

在 main 函数中，建立对象 stud1 时指定了 5 个实参。

它们按顺序传递给派生类构造函数 Student1 的形参。

然后，派生类构造函数将前面 3 个传递给基类构造函数的形参。

见图 11.10。

图 11.10

通过 `Student(n, nam, s)` 把 3 个值再传给基类构造函数的形参，见图 11.11。

```
Student(n, nam, s)
```

↓↓↓

```
Student(int n, string nam, char s) //这是基类构造函数的首部
```

图 11.11

在上例中也可以将派生类构造函数在类外面定义，而在类体中只写该函数的声明：`Student1(int n, string nam, char s, int a, string ad);`

在类的外面定义派生类构造函数：

```
Student1::Student1(int n,string nam,char s,int a,string ad):Student(n,nam,s){age=a;
addr=ad; }
```

请注意：在类中对派生类构造函数作声明时，不包括基类构造函数名及其参数表列（即 `Student(n, nam, s)`）。只在定义函数时才将它列出。

在以上的例子中，调用基类构造函数时的实参是从派生类构造函数的总参数表中得到的，也可以不从派生类构造函数的总参数表中传递过来，而直接使用常量或全局变量。

例如，派生类构造函数首行可以写成以下形式：

```
Student1(string nam,char s,int a,string ad):Student(10010,nam,s)
```

即基类构造函数 3 个实参中，有一个是常量 10010，另外两个从派生类构造函数的总参数表传递过来。

请回顾一下在第 9 章介绍过的构造函数初始化表的例子：

```
Box::Box(int h,int w,int len):height(h), width(w), length(len) { }
```

它也有一个冒号，在冒号后面的是对数据成员的初始化表。

实际上，本章介绍的在派生类构造函数中对基类成员初始化，就是在第 9 章介绍的构造函数初始化表。

也就是说，不仅可以利用初始化表对构造函数的数据成员初始化，而且可以利用初始化表调用派生类的基类构造函数，实现对基类数据成员的初始化。

也可以在同一个构造函数的定义中同时实现这两种功能。

例如，例 11.5 中派生类的基类构造函数的定义采用了下面的形式：

```
Student1(int n, string nam,char s,int a, string ad):Student(n,nam,s)
{age=a; //在函数体中对派生类数据成员初始化
  addr=ad;
}
```

可以将对 `age` 和 `addr` 的初始化也用初始化表处理，将构造函数改写为以下形式：`Student1(int n, string nam,char s,int a, string ad):Student(n,nam,s),age(a),addr(ad){}` 这样函数体为空，更显得简单和方便。

在建立一个对象时，执行构造函数的顺序是：

- ①派生类构造函数先调用基类构造函数；
- ②再执行派生类构造函数本身(即派生类构造函数的函数体)。

对上例来说，先初始化 `num`，`name`，`sex`，然后再初始化 `age` 和 `addr`。

在派生类对象释放时，先执行派生类析构函数 `~Student1()`，再执行其基类析构函数 `~Student()`。

### 11.5.2 有子对象的派生类的构造函数

类的数据成员中还可以包含类对象，如可以在声明一个类时包含这样的数据成员：

```
Student s1; // Student 是已声明的类名，s1 是 Student 类的对象
```

这时，`s1` 就是类对象中的内嵌对象，称为子对象(subobject)，即对象中的对象。

通过例子来说明问题。

例 11.6 包含子对象的派生类的构造函数。

为了简化程序以易于阅读，这里设基类 `Student` 的数据成员只有两个，即 `num` 和 `name`。

```

#include <iostream>
#include <string>
using namespace std;
class Student//声明基类
{public : //公用部分
Student(int n, string nam ) //基类构造函数，与例 11.5 相同
{num=n;
name=nam;
}
void display() //成员函数，输出基类数据成员
{cout<<"num:"<<num<<endl<<"name:"<<name<<endl;}
protected : //保护部分
int num;
string name;
};
class Student1: public Student //声明公用派生类 Student1
{public:
Student1(int n, string nam,int n1, string nam1,int a, string ad):Student(n,nam),monitor(n1,nam1) //派生类构造函数
{age=a;
addr=ad;
}
void show( )
{cout<<"This student is:"<<endl;
display(); //输出 num 和 name
cout<<"age: "<<age<<endl; //输出 age
cout<<"address: "<<addr<<endl<<endl; //输出 addr
}
void show_monitor() //成员函数，输出子对象
{cout<<endl<<"Class monitor is:"<<endl;
monitor.display(); //调用基类成员函数
}
private : //派生类的私有数据
Student monitor; //定义子对象(班长)
int age;
string addr;
};
int main( )
{
Student1 stud1(10010,"Wang-li",10001,"Li-sun",19,"115 Beijing Road,Shanghai");
stud1.show(); //输出学生的数据
stud1.show_monitor(); //输出子对象的数据
return 0;
}

```

运行时的输出如下:

This student is:

num: 10010

name: Wang-li  
age: 19  
address: 115 Beijing Road, Shanghai  
Class monitor is:  
num: 10001  
name: Li-sun

派生类构造函数的任务应该包括 3 个部分:

- (1) 对基类数据成员初始化;
- (2) 对子对象数据成员初始化;
- (3) 对派生类数据成员初始化。

程序中派生类构造函数首部如下:

**Student1**(int n, string nam, int n1, string nam1, int a, string ad): **Student**(n, nam), **monitor**(n1, nam1)

在上面的构造函数中有 6 个形参, 前两个作为基类构造函数的参数, 第 3、第 4 个作为子对象构造函数的参数, 第 5、第 6 个是用作派生类数据成员初始化的。

见图 11.12。

图 11.12

归纳起来, 定义派生类构造函数的一般形式为

派生类构造函数名(总参数表列): 基类构造函数名(参数表列), 子对象名(参数表列){派生类中新增数据成员初始化语句}

执行派生类构造函数的顺序是:

- ① 调用基类构造函数, 对基类数据成员初始化;
- ② 调用子对象构造函数, 对子对象数据成员初始化;
- ③ 再执行派生类构造函数本身, 对派生类数据成员初始化。

派生类构造函数的总参数表列中的参数, 应当包括基类构造函数和子对象的参数表列中的参数。

基类构造函数和子对象的次序可以是任意的, 如上面的派生类构造函数首部可以写成

**Student1**(int n, string nam, int n1, string nam1, int a, string ad): **monitor**(n1, nam1), **Student**(n, nam)

编译系统是根据相同的参数名(而不是根据参数的顺序)来确立它们的传递关系的。

但是习惯上一般先写基类构造函数。

如果有多个子对象, 派生类构造函数的写法依此类推, 应列出每一个子对象名及其参数表列。

### 11.5.3 多层派生时的构造函数

一个类不仅可以派生出一个派生类, 派生类还可以继续派生, 形成派生的层次结构。

在上面叙述的基础上, 不难写出在多级派生情况下派生类的构造函数。

例 11.7 多级派生情况下派生类的构造函数。

```
#include <iostream>
#include <string>
using namespace std;
class Student//声明基类
{public ://公用部分
    Student(int n, string nam)//基类构造函数
    {num=n;
    name=nam;
    }
    void display()//输出基类数据成员
    {cout<<"num:"<<num<<endl;
    cout<<"name:"<<name<<endl;
    }
```

```

protected ://保护部分
int num;//基类有两个数据成员
string name;
};
class Student1: public Student//声明公用派生类 Student1
{public :
Student1(int n,char nam[10],int a):Student(n,nam)//派生类构造函数
{age=a; }//在此处只对派生类新增的数据成员初始化
void show() //输出 num, name 和 age
{display(); //输出 num 和 name
cout<<"age: "<<age<<endl; }
private ://派生类的私有数据
int age; //增加一个数据成员
};
class Student2:public Student1 //声明间接公用派生类 Student2
{public://下面是间接派生类构造函数
Student2(int n, string nam,int a,int s):Student1(n,nam,a) {score=s;}
void show_all() //输出全部数据成员
{show(); //输出 num 和 name
cout<<"score:"<<score<<endl; //输出 age}
private:
int score; //增加一个数据成员
};
int main()
{Student2 stud(10010,"Li",17,89);
stud.show_all(); //输出学生的全部数据
return 0;
}

```

运行时的输出如下:

```

num:10010
name:Li
age:17
score:89

```

其派生关系如图 11.13 所示。

图 11.13

请注意基类和两个派生类的构造函数的写法:

基类的构造函数首部:

```
Student(int n, string nam)
```

派生类 Student1 的构造函数首部:

```
Student1(int n, string nam,int a):Student(n,nam)
```

派生类 Student2 的构造函数首部:

```
Student2(int n, string nam,int a,int s):Student1(n,nam,a)
```

在声明 Student2 类对象时, 调用 Student2 构造函数; 在执行 Student2 构造函数时, 先调用 Student1 构造函数; 在执行 Student1 构造函数时, 先调用基类 Student 构造函数。

初始化的顺序是:

① 先初始化基类的数据成员 num 和 name。



- ② 再初始化 Student1 的数据成员 age。
- ③ 最后再初始化 Student2 的数据成员 score。

#### 11.5.4 派生类构造函数的特殊形式

在使用派生类构造函数时，有以下特殊的形式：

(1) 当不需要对派生类新增的成员进行任何初始化操作时，派生类构造函数的函数体可以为空，即构造函数是空函数，如例 11.6 程序中派生类 Student1 构造函数可以改写为

```
Student1(int n, strin nam, int n1, strin nam1):Student(n,nam),  
monitor(n1,nam1) { }
```

此派生类构造函数的作用只是为了将参数传递给基类构造函数和子对象，并在执行派生类构造函数时调用基类构造函数和子对象构造函数。

在实际工作中常见这种用法。

(2) 如果在基类中没有定义构造函数，或定义了没有参数的构造函数，那么在定义派生类构造函数时可不写基类构造函数。

因为此时派生类构造函数没有向基类构造函数传递参数的任务。

调用派生类构造函数时系统会自动首先调用基类的默认构造函数。

如果在基类和子对象类型的声明中都没有定义带参数的构造函数，而且也不需对派生类自己的数据成员初始化，则可以不必显式地定义派生类构造函数。

因为此时派生类构造函数既没有向基类构造函数和子对象构造函数传递参数的任务，也没有对派生类数据成员初始化的任务。

在建立派生类对象时，系统会自动调用系统提供的派生类的默认构造函数，并在执行派生类默认构造函数的过程中，调用基类的默认构造函数和子对象类型默认构造函数。

如果在基类或子对象类型的声明中定义了带参数的构造函数，那么就必须显式地定义派生类构造函数，并在派生类构造函数中写出基类或子对象类型的构造函数及其参数表。

如果在基类中既定义无参的构造函数，又定义了有参的构造函数(构造函数重载)，则在定义派生类构造函数时，既可以包含基类构造函数及其参数，也可以不包含基类构造函数。

在调用派生类构造函数时，根据构造函数的内容决定调用基类的有参的构造函数还是无参的构造函数。

编程者可以根据派生类的需要决定采用哪一种方式。

#### 11.5.5 派生类的析构函数

在派生时，派生类是不能继承基类的析构函数的，也需要通过派生类的析构函数去调用基类的析构函数。

在派生类中可以根据需要定义自己的析构函数，用来对派生类中所增加的成员进行清理工作。

基类的清理工作仍然由基类的析构函数负责。

在执行派生类的析构函数时，系统会自动调用基类的析构函数和子对象的析构函数，对基类和子对象进行清理。

调用的顺序与构造函数正好相反：先执行派生类自己的析构函数，对派生类新增加的成员进行清理，然后调用子对象的析构函数，对子对象进行清理，最后调用基类的析构函数，对基类进行清理。

#### 11.6 多重继承

前面讨论的是单继承，即一个类是从一个基类派生而来的。

实际上，常常有这样的情况：一个派生类有两个或多个基类，派生类从两个或多个基类中继承所需的属性。

C++为了适应这种情况，允许一个派生类同时继承多个基类，这种行为称为多重继承(multiple inheritance)。

##### 11.6.1 声明多重继承的方法

如果已声明了类 A、类 B 和类 C，可以声明多重继承的派生类 D：

```
class D:public A,private B,protected C {类 D 新增加的成员}
```

D 是多重继承的派生类，它以公用继承方式继承 A 类，以私有继承方式继承 B 类，以保护继承方式继承 C

类。

D 按不同的继承方式的规则继承 A,B,C 的属性，确定各基类的成员在派生类中的访问权限。

### 11.6.2 多重继承派生类的构造函数

多重继承派生类的构造函数形式与单继承时的构造函数形式基本相同，只是在初始表中包含多个基类构造函数。

如

派生类构造函数名（总参数表列）：基类 1 构造函数（参数表列），基类 2 构造函数（参数表列），基类 3 构造函数（参数表列）{派生类中新增数成员据成员初始化语句}

各基类的排列顺序任意。

派生类构造函数的执行顺序同样为：先调用基类的构造函数，再执行派生类构造函数的函数体。

调用基类构造函数的顺序是按照声明派生类时基类出现的顺序。

例 11.8 声明一个教师(Teacher)类和一个学生(Student)类，用多重继承的方式声明一个研究生(Graduate)派生类。

教师类中包括数据成员 name(姓名)、age(年龄)、title(职称)。

学生类中包括数据成员 name1(姓名)、age(性别)、score(成绩)。

在定义派生类对象时给出初始化的数据，然后输出这些数据。

```
#include <iostream>
#include <string>
using namespace std;
class Teacher//声明类 Teacher(教师)
{public ://公用部分
Teacher(string nam,int a, string t)//构造函数
{name=nam;
age=a;
title=t;}
void display()//输出教师有关数据
{cout<<"name:"<<name<<endl;
cout<<"age"<<age<<endl;
cout<<"title:"<<title<<endl;
}
protected ://保护部分
string name;
int age;
string title;//职称
};
class Student//定义类 Student(学生)
{public :
Student(char nam[],char s,float sco)
{strcpy(name1,nam);
sex=s;
score=sco;}//构造函数
void display1()//输出学生有关数据
{cout<<"name:"<<name1<<endl;
cout<<"sex:"<<sex<<endl;
cout<<"score:"<<score<<endl;
}
```



```

protected ://保护部分
string name1;
char sex;
float
score;//成绩
};

class Graduate:public Teacher,public Student //声明多重继承的派生类 Graduate
{public:
    Graduate(string nam,int a,char s, string t,float sco,float w):Teacher(nam,a,t),Student(nam,s,sco),wage(w) { }
    void show() //输出研究生的有关数据
    {cout<<"name:"<<name<<endl;
      cout<<"age:"<<age<<endl;
      cout<<"sex:"<<sex<<endl;
      cout<<"score:"<<score<<endl;
      cout<<"title:"<<title<<endl;
      cout<<"wages:"<<wage<<endl;
    }
private: float wage; //工资
};

int main()
{Graduate grad1("Wang-li",24,'f',"assistant",89.5,1234.5);
grad1.show();
return 0;
}

```

程序运行结果如下:

```

name: Wang-li
age: 24
sex:f
score: 89.5
title: assistance
wages: 1234.5

```

在两个基类中分别用 name 和 name1 来代表姓名,其实这是同一个人的名字,从 Graduate 类的构造函数中可以看到总参数表中的参数 nam 分别传递给两个基类的构造函数,作为基类构造函数的实参。

解决这个问题有一个好方法:

在两个基类中可以都使用同一个数据成员名 name,而在 show 函数中引用数据成员时指明其作用域,如

```
cout<<"name:"<<Teacher::name<<endl;
```

这就是惟一的,不致引起二义性,能通过编译,正常运行。

通过这个程序还可以发现一个问题:在多重继承时,从不同的基类中会继承一些重复的数据。

如果有多个基类,问题会更突出。

在设计派生类时要细致考虑其数据成员,尽量减少数据冗余。

### 11.6.3 多重继承引起的二义性问题

多重继承可以反映现实生活中的情况,能够有效地处理一些较复杂的问题,使编写程序具有灵活性,但是多重继承也引起了一些值得注意的问题,它增加了程序的复杂度,使程序的编写和维护变得相对困难,容易出错。

其中最常见的问题就是继承的成员同名而产生的二义性(ambiguous)问题。

在上一节中已经初步地接触到这个问题了。

现在作进一步的讨论。

如果类 A 和类 B 中都有成员函数 `display` 和数据成员 `a`，类 C 是类 A 和类 B 的直接派生类。

分别讨论下列 3 种情况：

两个基类有同名成员。

如图 11.14 所示。

图 11.14

```
class A
{public:
int a; void display(); };
class B
{public:
int a;
void display(); };
class C :public A,public B {public :
int b;
void show();
};
```

如果在 `main` 函数中定义 C 类对象 `c1`，并调用数据成员 `a` 和成员函数 `display`：

```
C c1;
```

```
c1.a=3; c1.display();
```

由于基类 A 和基类 B 都有数据成员 `a` 和成员函数 `display`，编译系统无法判别要访问的是哪一基类的成员，因此，程序编译出错。

可以用基类名来限定：`c1.A::a=3;` //引用 `c1` 对象中的基类 A 的数据成员 `a`

`c1.A::display();` //调用 `c1` 对象中的基类 A 的成员函数 `display`

如果是在派生类 C 中通过派生类成员函数 `show` 访问基类 A 的 `display` 和 `a`，可以不必写对象名而直接写 `A::a=3;` //指当前对象

```
A::display();
```

如同上一节最后所介绍的那样。

为清楚起见，图 11.14 应改用图 11.15 的形式表示。

图 11.15

(2) 两个基类和派生类三者都有同名成员。

将上面的 C 类声明改为

```
class C :public A,public B
{int a;
void display();};
```

如图 11.16 所示。

即有 3 个 `a`，3 个 `display` 函数。

图 11.16

如果在 `main` 函数中定义 C 类对象 `c1`，并调用数据成员 `a` 和成员函数 `display`：

```
C c1;
```

```
c1.a=3;
```

```
c1.display();
```

程序能通过编译，也可正常运行。

访问的是派生类 C 中的成员。

规则是：基类的同名成员在派生类中被屏蔽，成为“不可见”的，或者说，派生类新增加的同名成员覆盖了基类中的同名成员。

因此如果在定义派生类对象的模块中通过对象名访问同名的成员，则访问的是派生类的成员。

请注意：不同的成员函数，只有在函数名和参数个数相同、类型相匹配的情况下才发生同名覆盖，如果只有函数名相同而参数不同，不会发生同名覆盖，而属于函数重载。

要在派生类外访问基类 A 中的成员，应指明作用域 A，写成以下形式：

`c1.A::a=3;` //表示是派生类对象 c1 中的基类 A 中的数据成员 a

`c1.A::display();` //表示是派生类对象 c1 中的基类 A 中的成员函数 display

如果类 A 和类 B 是从同一个基类派生的，如图 11.18 所示。

图 11.18

```
class N
{public:
int a;
void display(){cout<<"A::a="<<a<<endl;}}
};
class A:public N
{public:
int a1;
};
class B:public N
{public:
int a2;
};
class C :public A,public B
{public :
int a3;
void show(){cout<<"a3="<<a3<<endl;}}; int main() {C c1; //定义 C 类对象 c1
;
}
```

图 11.19 和图 11.20 表示了派生类 C 中成员的情况。

图 11.19 图 11.20

怎样才能访问类 A 中从基类 N 继承下来的成员呢？显然不能用 `c1.a=3;c1.display();` 或

`c1.N::a=3; c1.N::display();`

因为这样依然无法区别是类 A 中从基类 N 继承下来的成员，还是类 B 中从基类 N 继承下来的成员。应当通过类 N 的直接派生类名来指出要访问的是类 N 的哪一个派生类中的基类成员。

如

`c1.A::a=3; c1.A::display();` //要访问的是类 N 的派生类 A 中的基类成员

#### 11.6.4 虚基类

1. 虚基类的作用从上面的介绍可知：如果一个派生类有多个直接基类，而这些直接基类又有一个共同的基类，则在最终的派生类中会保留该间接共同基类数据成员的多份同名成员。

如图 11.19 和图 11.20 所示。

在引用这些同名的成员时，必须在派生类对象名后增加直接基类名，以避免产生二义性，使其惟一地标识一个成员，如

c1.A::display()。

在一个类中保留间接共同基类的多份同名成员，这种现象是人们不希望出现的。

C++提供虚基类(virtual base class)的方法，使得在继承间接共同基类时只保留一份成员。

图 11.21 图 11.22

现在，将类 A 声明为虚基类，方法如下：

```
class A//声明基类 A
{...};
class B:virtual public A//声明类 B 是类 A 的公用派生类，A 是 B 的虚基类
{...};
class C:virtual public A//声明类 C 是类 A 的公用派生类，A 是 C 的虚基类
{...};
```

注意：虚基类并不是在声明基类时声明的，而是在声明派生类时，指定继承方式时声明的。

因为一个基类可以在生成一个派生类时作为虚基类，而在生成另一个派生类时不作为虚基类。

声明虚基类的一般形式为

class 派生类名: virtual 继承方式 基类名

经过这样的声明后，当基类通过多条派生路径被一个派生类继承时，该派生类只继承该基类一次。

在派生类 B 和 C 中作了上面的虚基类声明后，派生类 D 中的成员如图 11.23 所示。

需要注意：为了保证虚基类在派生类中只继承一次，应当在该基类的所有直接派生类中声明为虚基类。

否则仍然会出现对基类的多次继承。

如果像图 11.24 所示的那样，在派生类 B 和 C 中将类 A 声明为虚基类，而在派生类 D 中没有将类 A 声明为虚基类，则在派生类 E 中，虽然从类 B 和 C 路径派生的部分只保留一份基类成员，但从类 D 路径派生的部分还保留一份基类成员。

图 11.23 图 11.24

2. 虚基类的初始化如果在虚基类中定义了带参数的构造函数，而且没有定义默认构造函数，则在其所有派生类(包括直接派生或间接派生的派生类)中，通过构造函数的初始化表对虚基类进行初始化。

例如

```
class A//定义基类 A
{A(int i){ } //基类构造函数，有一个参数};
class B:virtual public A //A 作为 B 的虚基类
{B(int n):A(n){ } //B 类构造函数，在初始化表中对虚基类初始化};
class C:virtual public A //A 作为 C 的虚基类
{C(int n):A(n){ } //C 类构造函数，在初始化表中对虚基类初始化};
class D:public B,public C //类 D 的构造函数，在初始化表中对所有基类初始化
{D(int n):A(n),B(n),C(n){ } };
```

注意：在定义类 D 的构造函数时，与以往使用的方法有所不同。

规定：在最后的派生类中不仅要负责对其直接基类进行初始化，还要负责对虚基类初始化。

C++编译系统只执行最后的派生类对虚基类的构造函数的调用，而忽略虚基类的其他派生类(如类 B 和类 C)对虚基类的构造函数的调用，这就保证了虚基类的数据成员不会被多次初始化。

### 3. 虚基类的简单应用举例

例 11.9 在例 11.8 的基础上，在 Teacher 类和 Student 类之上增加一个共同的基类 Person，如图 11.25 所示。作为人员的一些基本数据都放在 Person 中，在 Teacher 类和 Student 类中再增加一些必要的的数据。

图 11.25

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

```

using namespace std;
//声明公共基类 Person
class Person
{public:
Person(string nam,char s,int a)//构造函数
{name=nam;sex=s;age=a;}
protected :
//保护成员
string name;
char sex;
int age;
};
//声明 Person 的直接派生类 Teacher
class Teacher:virtual public Person //声明 Person 为公用继承的虚基类
{public :
Teacher(string nam,char s,int a, string t):Person(nam,s,a)//构造函数
{title=t;
}
protected : //保护成员
string title; //职称
};
//声明 Person 的直接派生类 Student
class Student:virtual public Person //声明 Person 为公用继承的虚基类
{public:
Student(string nam,char s,int a,float sco) //构造函数
:Person(nam,s,a),score(sco){ } //初始化表
protected : //保护成员
float score; //成绩
};
//声明多重继承的派生类 Graduate
class Graduate:public Teacher,public Student //Teacher 和 Student 为直接基类{public:
Graduate(string nam,char s,int a, string t,float sco,float w)//构造函数
:Person(nam,s,a),Teacher(nam,s,a,t),Student(nam,s,a,sco),wage(w){} //初始化表
void show() //输出研究生的有关数据
{cout<<"name:"<<name<<endl;
cout<<"age:"<<age<<endl;
cout<<"sex:"<<sex<<endl;
cout<<"score:"<<score<<endl;
cout<<"title:"<<title<<endl;
cout<<"wages:"<<wage<<endl;
}
private:
float wage; //工资
};
//主函数
int main()
{

```

```
Graduate grad1("Wang-li",'f',24,"assistant",89.5,1234.5); grad1.show();
return 0;
}
```

运行结果为

```
name: Wang-li
age:24
sex:f
score:89.5
title:assistant
wages:1234.5
```

可以看到：使用多重继承时要十分小心，经常会出现二义性问题。

许多专业人员认为：不要提倡在程序中使用多重继承，只有在比较简单和不易出现二义性的情况或实在必要时才使用多重继承，能用单一继承解决的问题就不要使用多重继承。

也是由于这个原因，有些面向对象的程序设计语言(如 Java，Smalltalk)并不支持多重继承。

**11.7 基类与派生类的转换**只有公用派生类才是基类真正的子类型，它完整地继承了基类的功能。

基类与派生类对象之间有赋值兼容关系，由于派生类中包含从基类继承的成员，因此可以将派生类的值赋给基类对象，在用到基类对象的时候可以用其子类对象代替。

具体表现在以下几个方面：

(1) 派生类对象可以向基类对象赋值。

可以用子类(即公用派生类)对象对其基类对象赋值。

如

```
A a1; //定义基类 A 对象 a1
B b1; //定义类 A 的公用派生类 B 的对象 b1
a1=b1; //用派生类 B 对象 b1 对基类对象 a1 赋值
```

在赋值时舍弃派生类自己的成员。

如图 11.26 示意。

实际上，所谓赋值只是对数据成员赋值，对成员函数不存在赋值问题。

请注意：赋值后不能企图通过对象 a1 去访问派生类对象 b1 的成员，因为 b1 的成员与 a1 的成员是不同的。

假设 age 是派生类 B 中增加的公用数据成员，分析下面的用法：

图 11.26

```
a1.age=23; //错误，a1 中不包含派生类中增加的成员
```

```
b1.age=21; //正确，b1 中包含派生类中增加的成员
```

应当注意，子类型关系是单向的、不可逆的。

B 是 A 的子类型，不能说 A 是 B 的子类型。

只能用子类对象对其基类对象赋值，而不能用基类对象对其子类对象赋值，理由是显然的，因为基类对象不包含派生类的成员，无法对派生类的成员赋值。

同理，同一基类的不同派生类对象之间也不能赋值。

(2) 派生类对象可以替代基类对象向基类对象的引用进行赋值或初始化。

如已定义了基类 A 对象 a1，可以定义 a1 的引用变量：

```
A a1; //定义基类 A 对象 a1
B b1; //定义公用派生类 B 对象 b1
A& r=a1; //定义基类 A 对象的引用变量 r，并用 a1 对其初始化
```

这时，引用变量 r 是 a1 的别名，r 和 a1 共享同一段存储单元。

也可以用子类对象初始化引用变量 r，将上面最后一行改为

```
A& r=b1; //定义基类 A 对象的引用变量 r，并用派生类 B 对象 b1 对其初始化
```

或者保留上面第 3 行“A& r=a1;”，而对 r 重新赋值：

`r=b1;`//用派生类 B 对象 b1 对 a1 的引用变量 r 赋值

注意: 此时 r 并不是 b1 的别名, 也不与 b1 共享同一段存储单元。

它只是 b1 中基类部分的别名, r 与 b1 中基类部分共享同一段存储单元, r 与 b1 具有相同的起始地址。  
见图 11.27。

图 11.27

(3) 如果函数的参数是基类对象或基类对象的引用, 相应的实参可以用子类对象。

如有一函数

`fun: void fun(A& r)`//形参是类 A 的对象的引用变量

`{cout<<r.num<<endl;}` //输出该引用变量的数据成员 num

函数的形参是类 A 的对象的引用变量, 本来实参应该为 A 类的对象。

由于子类对象与派生类对象赋值兼容, 派生类对象能自动转换类型, 在调用 fun 函数时可以用派生类 B 的对象 b1 作实参: `fun(b1)`; 输出类 B 的对象 b1 的基类数据成员 num 的值。

与前相同, 在 fun 函数中只能输出派生类中基类成员的值。

(4) 派生类对象的地址可以赋给指向基类对象的指针变量, 也就是说, 指向基类对象的指针变量也可以指向派生类对象。

例 11.10 定义一个基类 Student(学生), 再定义 Student 类的公用派生类 Graduate(研究生), 用指向基类对象的指针输出数据。

本例主要是说明用指向基类对象的指针指向派生类对象, 为了减少程序长度, 在每个类中只设很少成员。学生类只设 num(学号), name(名字)和 score(成绩)3 个数据成员, Graduate 类只增加一个数据成员 pay(工资)。

程序如下:

```
#include <iostream>
```

```
#include <string>
```

```
Graduate::Graduate(int n, string nam, float s, float p):Student(n,nam,s),pay(p){ }
```

```
using namespace std;
```

```
class Student//声明 Student 类
```

```
{public :
```

```
Student(int, string, float );//声明构造函数
```

```
void display( );//声明输出函数
```

```
private :
```

```
int num;
```

```
string name;
```

```
float score;
```

```
};
```

```
Student::Student(int n, string nam, float s) //定义构造函数
```

```
{num=n;
```

```
name=nam;
```

```
score=s;
```

```
}
```

```
void Student::display( )//定义输出函数
```

```
{cout<<endl<<"num:"<<num<<endl;
```

```
cout<<"name:"<<name<<endl;
```

```
cout<<"score:"<<score<<endl;
```

```
}
```

```
class Graduate:public Student//声明公用派生类 Graduate
```

```
{public :
```

```
Graduate(int, string, float, float );//声明构造函数
```



```

void display();//声明输出函数
private :
float pay;//工资
};
//定义构造函数
void Graduate::display() //定义输出函数
{Student::display(); //调用 Student 类的 display 函数
cout<<"pay="<<pay<<endl; }
int main()
{Student stud1(1001,"Li",87.5); //定义 Student 类对象 stud1
  Graduate grad1(2001,"Wang",98.5,563.5); //定义 Graduate 类对象 grad1
  Student *pt=&stud1; //定义指向 Student 类对象的指针并指向 stud1
  pt->display(); //调用 stud1.display 函数
  pt=&grad1; //指针指向 grad1
  pt->display(); //调用 grad1.display 函数
}

```

很多读者会认为: 在派生类中有两个同名的 display 成员函数, 根据同名覆盖的规则, 被调用的应当是派生类 Graduate 对象的 display 函数, 在执行 Graduate::display 函数过程中调用 Student::display 函数, 输出 num,name,score, 然后再输出 pay 的值。

事实上这种推论是错误的, 先看看程序的输出结果:

```

num:1001
name:Li
score:87.5
num:2001
name:wang
score:98.5

```

并没有输出 pay 的值。

问题在于 pt 是指向 Student 类对象的指针变量, 即使让它指向了 grad1, 但实际上 pt 指向的是 grad1 中从基类继承的部分。

通过指向基类对象的指针, 只能访问派生类中的基类成员, 而不能访问派生类增加的成员。

所以 pt->display()调用的不是派生类 Graduate 对象所增加的 display 函数, 而是基类的 display 函数, 所以只输出研究生 grad1 的 num,name,score3 个数据。

如果想通过指针输出研究生 grad1 的 pay, 可以另设一个指向派生类对象的指针变量 ptr, 使它指向 grad1, 然后用 ptr->display()调用派生类对象的 display 函数。

但这不大方便。

通过本例可以看到: 用指向基类对象的指针变量指向子类对象是合法的、安全的, 不会出现编译上的错误。但在应用上却不能完全满足人们的希望, 人们有时希望通过使用基类指针能够调用基类和子类对象的成员。

在下一章就要解决这个问题。

办法是使用虚函数和多态性。

## 11.8 继承与组合

在本章 11.5.2 节中已经说明: 在一个类中可以用类对象作为数据成员, 即子对象。

在 11.5.2 节中的例 11.6 中, 对象成员的类型是基类。

实际上, 对象成员的类型可以是本派生类的基类, 也可以是另外一个已定义的类。

在一个类中以另一个类的对象作为数据成员的, 称为类的组合(composition)。

例如, 声明 Professor(教授)类是 Teacher(教师)类的派生类, 另有一个类 BirthDate(生日), 包含 year,month,day 等数据成员。



可以将教授生日的信息加入到 Professor 类的声明中。

如

```
class Teacher//教师类
{public:
  |
private:
  int num;
  string name;
  char sex;
};
class BirthDate //生日类
{ public :
  |
private:
  int year;
  int month;
  int day;
};
class Professor:public Teacher //教授类
{public:
  |
private:
  BirthDate birthday; //BirthDate 类的对象作为数据成员
};
```

类的组合和继承一样，是软件重用的重要方式。

组合和继承都是有效地利用已有类的资源。

但二者的概念和用法不同。

Professor 类通过继承，从 Teacher 类得到了 num,name,age,sex 等数据成员，通过组合，从 BirthDate 类得到了 year,month,day 等数据成员。

继承是纵向的，组合是横向的。

如果定义了 Professor 对象 prof1，显然 prof1 包含了生日的信息。

通过这种方法有效地组织和利用现有的类，大大减少了工作量。

如果有

```
void fun1(Teacher &);
```

```
void fun2(BirthDate &);
```

在 main 函数中调用这两个函数:

```
fun1(prof1);//正确，形参为 Teacher 类对象的引用，实参为 Teacher 类的子类对象，与之赋值兼容
```

```
fun2(prof1.birthday);//正确，实参与形参类型相同，都是 BirthDate 类对象 fun2(prof1);//错误，形参要求是 BirthDate 类对象，而 prof1 是 Professor 类型，不匹配
```

对象成员的初始化的方法已在 11.5.2 节中作过介绍。

如果修改了成员类的部分内容，只要成员类的公用接口(如头文件名)不变，如无必要，组合类可以不修改。但组合类需要重新编译。

**11.9** 继承在软件开发中的重要意义有了继承，使软件的重用成为可能。

继承是 C++和 C 的最重要的区别之一。

由于 C++提供了继承的机制，这就吸引了许多厂商开发各类实用的类库。

用户将它们作为基类去建立适合自己的类(即派生类)，并在此基础上设计自己的应用程序。

类库的出现使得软件的重用更加方便，现在有一些类库是随着 C++编译系统卖给用户的。

读者不要认为类库是 C++ 编译系统的一部分。

不同的 C++ 编译系统提供的由不同厂商开发的类库一般是不同的。

对类库中类的声明一般放在头文件中，**类的实现**(函数的定义部分)是单独编译的，以目标代码形式存放在系统某一目录下。

用户使用类库时，不需要了解源代码，**但必须知道头文件的使用方法和怎样去连接这些目标代码**(在哪个子目录下)，以便源程序在编译后与之连接。

由于基类是单独编译的，在程序编译时只需对派生类新增的功能进行编译，这就大大提高了调试程序的效率。

如果在必要时修改了基类，只要基类的公用接口不变，派生类不必修改，但基类需要重新编译，派生类也必须重新编译，否则不起作用。

人们为什么这么看重继承，要求在软件开发中使用继承机制，尽可能地通过继承建立一批新的类？为什么不是将已有的类加以修改，使之满足自己应用的要求呢？

- (1) 有许多基类是被程序的其他部分或其他程序使用的，这些程序要求保留原有的基类不受破坏。
- (2) 用户往往得不到基类的源代码。
- (3) 在类库中，一个基类可能已被指定与用户所需的多种组件建立了某种关系，因此在类库中的基类是不容许修改的。
- (4) 实际上，许多基类并不是从已有的其他程序中选取来的，而是专门作为基类设计的。
- (5) 在面向对象程序设计中，需要设计类的层次结构，从最初的抽象类出发，每一层派生类的建立都逐步地向着目标的具体实现前进。

## 第 12 章 多态性与虚函数

### 12.1 多态性的概念

### 12.2 一个典型的例子

### 12.3 虚函数

### 12.4 纯虚函数与抽象类

### 12.1 多态性的概念

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++ 程序设计中，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。

在面向对象方法中一般是这样表述多态性的:向不同的对象发送同一个消息，不同的对象在接收时会产生不同的行为(即方法)。

也就是说，每个对象可以用自己的方式去响应共同的消息。

在 C++ 程序设计中，在不同的类中定义了其响应消息的方法，那么使用这些类时，不必考虑它们是什么类型，只要发布消息即可。

从系统实现的角度看，多态性分为两类：静态多态性和动态多态性。

以前学过的函数重载和运算符重载实现的多态性属于静态多态性，在程序编译时系统就能决定调用的是哪个函数，因此静态多态性又称编译时的多态性。

静态多态性是通过函数的重载实现的(运算符重载实质上也是函数重载)。

动态多态性是在程序运行过程中才动态地确定操作所针对的对象。

它又称运行时的多态性。

动态多态性是通过虚函数(virtual function)实现的。

有关静态多态性的应用已经介绍过了，在本章中主要介绍动态多态性和虚函数。

要研究的问题是：当一个基类被继承为不同的派生类时，各派生类可以使用与基类成员相同的成员名，如果在运行时用同一个成员名调用类对象的成员，会调用哪个对象的成员？也就是说，通过继承而产生了相关的不同的派生类，与基类成员同名的成员在不同的派生类中有不同的含义。

也可以说，多态性是“一个接口，多种方法”。

## 12.2 一个典型的例子

下面是一个承上启下的例子。

一方面它是有关继承和运算符重载内容的综合应用的例子，通过这个例子可以进一步融会贯通前面所学的内容，另一方面又是作为讨论多态性的一个基础用例。

例 12.1 先建立一个 Point(点)类，包含数据成员 x,y(坐标点)。

以它为基类，派生出一个 Circle(圆)类，增加数据成员 r(半径)，再以 Circle 类为直接基类，派生出一个 Cylinder(圆柱体)类，再增加数据成员 h(高)。

要求编写程序，重载运算符“<<”和“>>”，使之能用于输出以上类对象。

对于一个比较大的程序，应当分成若干步骤进行。

先声明基类，再声明派生类，逐级进行，分步调试。

声明基类 Point 类可写出声明基类 Point 的部分如下：

```
#include <iostream>
//声明类 Point
class Point
{
public:
    Point(float x=0,float y=0); //有默认参数的构造函数
    void setPoint(float ,float); //设置坐标值
    float getX()const {return x;} //读 x 坐标
    float getY()const {return y;} //读 y 坐标
    friend ostream & operator <<(ostream &,const Point &); //重载运算符“<<”
protected: //受保护成员
    float x, y;
};
//下面定义 Point 类的成员函数
Point::Point(float a,float b) //Point 的构造函数
{x=a;y=b;} //对 x,y 初始化
void Point::setPoint(float a,float b) //设置 x 和 y 的坐标值
{x=a;y=b;} //为 x,y 赋新值
ostream & operator <<(ostream &output, const Point &p){output<< "["<<p.x<< ","<<p.y<< "]"<<endl; return
output;} //重载运算符“<<”，使之能输出点的坐标
```

以上完成了基类 Point 类的声明。

现在要对上面写的基类声明进行调试，检查它是否有错，为此要写出 main 函数。

实际上它是一个测试程序。

```
int main()
{
    Point p(3.5,6.4);//建立 Point 类对象 p
    cout<<"x="<<p.getX()<<"y="<<p.getY()<<endl;//输出 p 的坐标值
    p.setPoint(8.5,6.8);//重新设置 p 的坐标值
    cout<<"p(new):"<<p<<endl;//用重载运算符"<<"输出 p 点坐标
}
```

程序编译通过，运行结果为

x=3.5,y=6.4

p(new):[8.5,6.8]

测试程序检查了基类中各函数的功能，以及运算符重载的作用，证明程序是正确的。

(2)声明派生类 Circle 在

上面的基础上，再写出声明派生类 Circle 的部分：

```
class Circle:public Point//circle 是 Point 类的公用派生类
{public:
    Circle(float x=0,float y=0,float r=0);//构造函数
    void setRadius(float );//设置半径值
    float getRadius()const;//读取半径值
    float area()const;//计算圆面积
    friend ostream &operator <<(ostream &,const Circle &);//重载运算符"<<"
private:
    float radius;
};
//定义构造函数，对圆心坐标和半径初始化
Circle::Circle(float a,float b,float r):Point(a,b),radius(r){}
//设置半径值
void Circle::setRadius(float r){radius=r;}
//读取半径值
float Circle::getRadius()const {return radius;}
//计算圆面积
float Circle::area()const
{return 3.14159*radius*radius;}
//重载运算符"<<"，使之按规定的形式输出圆的信息
ostream &operator <<(ostream &output,const Circle
&c){output<<"Center=["<<c.x<<" "<<c.y<<"],r="<<c.radius<<" ,area="<<c.area()<<endl;
return output;
}
```

为了测试以上 Circle 类的定义，可以写出下面的主函数：

```
int main() {Circle c(3.5,6.4,5.2);//建立 Circle 类对象 c，并给定圆心坐标和半径
cout<<"original circle:\nx="<<c.getX()<<" y="<<c.getY()<<" r="<<c.getRadius()<<"
area="<<c.area()<<endl;//输出圆心坐标、半径和面积
c.setRadius(7.5);//设置半径值
c.setPoint(5,5);//设置圆心坐标值 x,y
cout<<"new circle:\n"<<c;//用重载运算符"<<"输出圆对象的信息
```

Point &pRef=c;//pRef 是 Point 类的引用变量，被 c 初始化

cout<<"pRef:"<<pRef;//输出 pRef 的信息

return 0;

}

程序编译通过，运行结果为

original circle:(输出原来的圆的的数据)

x=3.5, y=6.4, r=5.2, area=84.9486

new circle:(输出修改后的圆的的数据)

Center=[5,5], r=7.5, area=176.714

pRef:[5,5] (输出圆的圆心“点”的数据)

(3)声明 Circle 的派生类 Cylinder 前面已从基类 Point 派生出 Circle 类，现在再从 Circle 派生出 Cylinder 类。

class Cylinder:public Circle// Cylinder 是 Circle 的公用派生类

{public :

Cylinder (float x=0,float y=0,float r=0,float h=0);//构造函数

void setHeight(float );//设置圆柱高

float getHeight( )const;//读取圆柱高

loat area( )const;//计算圆表面积

float volume( )const;//计算圆柱体积

friend ostream& operator <<(ostream&,const Cylinder&);//重载运算符<<"

protected :

float height;//圆柱高

};

//定义构造函数

Cylinder::Cylinder(float a,float b,float r,float h):Circle(a,b,r),height(h){}

//设置圆柱高

void Cylinder::setHeight(float h){height=h;}

//读取圆柱高

float Cylinder::getHeight( )const {return height;}

//计算圆表面积

float Cylinder::area( )const { return 2\*Circle::area( )+2\*3.14159\*radius\*height;}

//计算圆柱体积

float Cylinder::volume()const {return Circle::area()\*height;}

ostream &operator <<(ostream &output,const Cylinder&

cy){output<<"Center=[ "<<cy.x<<" "<<cy.y<<"",r=" "<<cy.radius<<"",h=" "<<cy.height <<"\narea=" "<<cy.area( )<<"",

volume=" "<<cy.volume( )<<endl;

return output;} //重载运算符"<<"

可以写出下面的主函数:

int main( )

{

Cylinder cy1(3.5,6.4,5.2,10);//定义 Cylinder 类对象 cy1

cout<<"\noriginal cylinder:\nx="<<cy1.getX( )<<" y="<<cy1.getY( )<<" r="

<<cy1.getRadius( )<<" h="<<cy1.getHeight( )<<"\narea="<<cy1.area()

<<" volume="<<cy1.volume()<<endl;//用系统定义的运算符"<<"输出 cy1 的数据

cy1.setHeight(15);//设置圆柱高

cy1.setRadius(7.5);//设置圆半径

cy1.setPoint(5,5);//设置圆心坐标值 x,y

cout<<"\nnew cylinder:\n"<<cy1;//用重载运算符"<<"输出 cy1 的数据

```

Point &pRef=cy1;//pRef 是 Point 类对象的引用变量
cout<<"\npRef as a Point:"<<pRef;//pRef 作为一个“点”输出
Circle &cRef=cy1;//cRef 是 Circle 类对象的引用变量
cout<<"\ncRef as a Circle:"<<cRef;//cRef 作为一个“圆”输出
return 0;
}

```

运行结果如下:

```

original cylinder:(输出 cy1 的初始值)
x=3.5, y=6.4, r=5.2, h=10 (圆心坐标 x,y。
半径 r, 高 h)
area=496.623, volume=849.486 (圆柱表面积 area 和体积 volume)
new cylinder: (输出 cy1 的新值)
Center=[5,5], r=7.5, h=15 (以[5,5]形式输出圆心坐标)
area=1060.29, volume=2650.72(圆柱表面积 area 和体积 volume)
pRef as a Point:[5,5] (pRef 作为一个“点”输出)
cRef as a Circle:Center=[5,5], r=7.5, area=176.714(cRef 作为一个“圆”输出)

```

在本例中存在静态多态性，这是运算符重载引起的。  
 可以看到，在编译时编译系统即可以判定应调用哪个重载运算符函数。  
 稍后将在此基础上讨论动态多态性问题。

## 12.3 虚函数

### 12.3.1 虚函数的作用

在类的继承层次结构中，在不同的层次中可以出现名字相同、参数个数和类型都相同而功能不同的函数。  
 编译系统按照同名覆盖的原则决定调用的对象。

在例 12.1 程序中用 `cy1.area()` 调用的是派生类 `Cylinder` 中的成员函数 `area`。

如果想调用 `cy1` 中的直接基类 `Circle` 的 `area` 函数，应当表示为`cy1.Circle::area()`。

用这种方法来区分两个同名的函数。

但是这样做很不方便。

人们提出这样的设想，能否用同一个调用形式，既能调用派生类又能调用基类的同名函数。

在程序中不是通过不同的对象名去调用不同派生层次中的同名函数，而是通过指针调用它们。

例如，用同一个语句“`pt->display();`”可以调用不同派生层次中的 `display` 函数，只需在调用前给指针变量 `pt` 赋以不同的值(使之指向不同的类对象)即可。

C++中的虚函数就是用来解决这个问题的。

虚函数的作用是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。

请分析例 12.2。

这个例子开始时没有使用虚函数，然后再讨论使用虚函数的情况。

例 12.2 基类与派生类中有同名函数。

在下面的程序中 `Student` 是基类，`Graduate` 是派生类，它们都有 `display` 这个同名的函数。

```

#include <iostream>
#include <string>
using namespace std;
//声明基类 Student
class Student
{public:Student(int, string,float);//声明构造函数 void display();//声明输出函数
protected ://受保护成员，派生类可以访问
int num;
string name;

```



```

float score;
};
//Student 类成员函数的实现
Student::Student(int n, string nam,float s)//定义构造函数 {num=n;name=nam;score=s;}
void Student::display( )// 定 义 输 出 函 数
{cout<<"num:"<<num<<"\\nname:"<<name<<"\\nscore:"<<score<<"\\n\\n";}
//声明公用派生类 Graduate
class Graduate:public Student
{public:Graduate(int, string, float, float);//声明构造函数 void display( );//声明输出函数
private:float pay;};
// Graduate 类成员函数的实现
void Graduate::display( )// 定 义 输 出 函 数
{cout<<"num:"<<num<<"\\nname:"<<name<<"\\nscore:"<<score<<"\\npay="<<pay<<endl;}
Graduate::Graduate(int n, string nam,float s,float p):Student(n,nam,s),pay(p){}
//主函数
int main()
{Student stud1(1001,"Li",87.5);//定义 Student 类对象 stud1
Graduate grad1(2001,"Wang",98.5,563.5);//定义 Graduate 类对象 grad1
Student *pt=&stud1;//定义指向基类对象的指针变量 pt
pt->display( );
pt=&grad1;
pt->display( );
return 0;
}

```

运行结果如下，请仔细分析。

```

num:1001(stud1 的数据)
name:Li
score:87.5
num:2001 (grad1 中基类部分的数据)
name:wang
score:98.5

```

下面对程序作一点修改，在 Student 类中声明 display 函数时，在最左面加一个关键字 virtual，即

```
virtual void display( );
```

这样就把 Student 类的 display 函数声明为虚函数。

程序其他部分都不改动。

再编译和运行程序，请注意分析运行结果：

```

num:1001(stud1 的数据)
name:Li
score:87.5
num:2001 (grad1 中基类部分的数据)
name:wang
score:98.5

```

pay=1200 (这一项以前是没有的)

由虚函数实现的动态多态性就是：同一类族中不同类的对象，对同一函数调用作出不同的响应。

虚函数的使用方法是：

(1)在基类用 **virtual** 声明成员函数为虚函数。

这样就可以在派生类中重新定义此函数，为它赋予新的功能，并能方便地被调用。

在类外定义虚函数时，不必再加 `virtual`。

(2)在派生类中重新定义此函数，要求函数名、函数类型、函数参数个数和类型全部与基类的虚函数相同，并根据派生类的需要重新定义函数体。

C++规定，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。

因此在派生类重新声明该虚函数时，可以加 `virtual`，也可以不加，但习惯上一般在每一层声明该函数时都加 `virtual`，使程序更加清晰。

如果在派生类中没有对基类的虚函数重新定义，则派生类简单地继承其直接基类的虚函数。

(3)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。

(4)通过该指针变量调用此虚函数，此时调用的就是指针变量指向的对象的同名函数。

通过虚函数与指向基类对象的指针变量的配合使用，就能方便地调用同一类族中不同类的同名函数，只要先用基类指针指向即可。

如果指针不断地指向同一类族中不同类的对象，就能不断地调用这些对象中的同名函数。

这如同前面说的，不断地告诉出租车司机要去的目的地，然后司机把你送到你要去的地方。

需要说明：有时在基类中定义的非虚函数会在派生类中被重新定义(如例 12.1 中的 `area` 函数)，如果用基类指针调用该成员函数，则系统会调用对象中基类部分的成员函数；如果用派生类指针调用该成员函数，则系统会调用派生类对象中的成员函数，这并不是多态性行为(使用的是不同类型的指针)，没有用到虚函数的功能。

以前介绍的函数重载处理的是同一层次上的同名函数问题，而虚函数处理的是不同派生层次上的同名函数问题，前者是横向重载，后者可以理解为纵向重载。

但与重载不同的是：同一类族的虚函数的首部是相同的，而函数重载时函数的首部是不同的(参数个数或类型不同)。

### 12.3.2 静态关联与动态关联

编译系统要根据已有的信息，对同名函数的调用作出判断。

对于调用同一类族中的虚函数，应当在调用时用一定的方式告诉编译系统，你要调用的是哪个类对象中的函数。

这样编译系统在对程序进行编译时，即能确定调用的是哪个类对象中的函数。

确定调用的具体对象的过程称为关联(binding)。

在这里是指把一个函数名与一个类对象捆绑在一起，建立关联。

一般地说，关联指把一个标识符和一个存储地址联系起来。

前面所提到的函数重载和通过对象名调用的虚函数，在编译时即可确定其调用的虚函数属于哪一个类，其过程称为静态关联(static binding)，由于是在运行前进行关联的，故又称为早期关联(early binding)。函数重载属静态关联。

在上一小节程序中看到了怎样使用虚函数，在调用虚函数时并没有指定对象名，那么系统是怎样确定关联的呢？是通过基类指针与虚函数的结合来实现多态性的。

先定义了一个指向基类的指针变量，并使它指向相应的类对象，然后通过这个基类指针去调用虚函数(例如“`pt->display()`”)。

显然，对这样的调用方式，编译系统在编译该行时是无法确定调用哪一个类对象的虚函数的。

因为编译只作静态的语法检查，光从语句形式是无法确定调用对象的。

在这样的情况下，编译系统把它放到运行阶段处理，在运行阶段确定关联关系。

在运行阶段，基类指针变量先指向了某一个类对象，然后通过此指针变量调用该对象中的函数。

此时调用哪一个对象的函数无疑是确定的。

例如，先使 `pt` 指向 `grad1`，再执行“`pt->display()`”，当然是调用 `grad1` 中的 `display` 函数。

由于是在运行阶段把虚函数和类对象“绑定”在一起的，因此，此过程称为动态关联(dynamic binding)。

这种多态性是动态的多态性，即运行阶段的多态性。

在运行阶段，指针可以先后指向不同的类对象，从而调用同一类族中不同类的虚函数。

由于动态关联是在编译以后的运行阶段进行的，因此也称为滞后关联(late binding)。

### 12.3.3 在什么情况下应当声明虚函数



使用虚函数时，有两点要注意：

(1)只能用 **virtual** 声明类的成员函数，使它成为虚函数，而不能将类外的普通函数声明为虚函数。

因为虚函数的作用是允许在派生类中对基类的虚函数重新定义。

显然，它只能用于类的继承层次结构中。

(2)一个成员函数被声明为虚函数后，在同一类族中的类就**不能再定义**一个非 **virtual** 的但与该虚函数具有相同的参数(包括个数和类型)和函数返回值类型的同名函数。

根据什么考虑是否把一个成员函数声明为虚函数呢？主要考虑以下几点：

(1)首先看成员函数所在的类**是否会作为基类**。

然后看成员函数在类的**继承后有无可能被更改功能**，如果希望更改其功能的，一般应该将它声明为虚函数。

(2)如果成员函数在类被**继承后功能不需修改**，或派生类用不到该函数，则不要把它声明为虚函数。

不要仅仅考虑到要作为基类而把类中的所有成员函数都声明为虚函数。

(3)应考虑对成员函数的调用**是通过对象名还是通过基类指针或引用去访问**，如果是通过基类指针或引用去访问的，则应当声明为虚函数。

(4)有时，在定义虚函数时，并不定义其函数体，即函数体是空的。

它的作用只是**定义了一个虚函数名，具体功能留给派生类去添加**。

在 12.4 节中将详细讨论此问题。

**需要说明的是：**使用虚函数，系统要有一定的空间开销。

当一个类带有虚函数时，编译系统会为该类构造一个**虚函数表(virtual function table，简称 vtable)**，它是一个**指针数组，存放每个虚函数的入口地址**。

系统在进行动态关联时的时间开销是很少的，因此，**多态性是高效的**。

#### 12.3.4 虚析构函数

析构函数的作用**是在对象撤销之前做必要的“清理现场”的工作**。

当派生类的对象从内存中撤销时一般先调用派生类的析构函数，然后再调用基类的析构函数。

但是，如果用 **new** 运算符建立了临时对象，若基类中有析构函数，并且定义了一个指向该基类的指针变量。在程序用带指针参数的 **delete** 运算符撤销对象时，会发生一个情况：系统会**只执行基类的析构函数**，而不执行派生类的析构函数。

例 12.3 基类中有非虚析构函数时的执行情况。

为简化程序，只列出最必要的部分。

```
#include <iostream>
using namespace std;
class Point//定义基类 Point 类
{public:
    Point(){}//Point 类构造函数
    ~Point(){cout<<"executing Point destructor"<<endl;}//Point 类析构函数
};
class Circle:public Point//定义派生类 Circle 类
{public:
    Circle(){}//Circle 类构造函数
    ~Circle(){cout<<"executing Circle destructor"<<endl;}//Circle 类析构函数
private:
    int radius;
};
int main()
{
    Point *p=new Circle;//用 new 开辟动态存储空间
    delete p;//用 delete 释放动态存储空间
    return 0;
```

```
}
```

这只是一个示意的程序。

p 是指向基类的指针变量，指向 new 开辟的动态存储空间，希望用 delete 释放 p 所指向的空间。

但运行结果为：

executing Point destructor

表示只执行了基类 Point 的析构函数，而没有执行派生类 Circle 的析构函数。

原因是以前介绍过的。

如果希望能执行派生类 Circle 的析构函数，可以将基类的析构函数声明为虚析构函数，如

```
virtual ~Point(){cout<<"executing Point destructor"<<endl;}
```

程序其他部分不改动，再运行程序，结果为

executing Circle destructor

executing Point destructor

先调用了派生类的析构函数，再调用了基类的析构函数，符合人们的愿望。

当基类的析构函数为虚函数时，无论指针指的是同一类族中的哪一个类对象，系统会采用动态关联，调用相应的析构函数，对该对象进行清理工作。

如果将基类的析构函数声明为虚函数时，由该基类所派生的所有派生类的析构函数也都自动成为虚函数，即使派生类的析构函数与基类的析构函数名字不相同。

最好把基类的析构函数声明为虚函数。

这将使所有派生类的析构函数自动成为虚函数。

这样，如果程序中显式地用了 delete 运算符准备删除一个对象，而 delete 运算符的操作对象用了指向派生类对象的基类指针，则系统会调用相应类的析构函数。

虚析构函数的概念和用法很简单，但它在面向对象程序设计中却是很重要的技巧。

专业人员一般都习惯声明虚析构函数，即使基类并不需要析构函数，也显式地定义一个函数体为空的虚析构函数，以保证在撤销动态分配空间时能得到正确的处理。

构造函数不能声明为虚函数。

这是因为在执行构造函数时类对象还未完成建立过程，当然谈不上函数与类对象的绑定。

## 12.4 纯虚函数与抽象类

### 12.4.1 纯虚函数

有时在基类中将某一成员函数定为虚函数，并不是基类本身的要求，而是考虑到派生类的需要，在基类中预留了一个函数名，具体功能留给派生类根据需要去定义。

例如在本章的例 12.1 程序中，基类 Point 中没有求面积的 area 函数，因为“点”是没有面积的，也就是说，基类本身不需要这个函数，所以在例 12.1 程序中的 Point 类中没有定义 area 函数。

但是，在其直接派生类 Circle 和间接派生类 Cylinder 中都需要有 area 函数，而且这两个 area 函数的功能不同，一个是求圆面积，一个是求圆柱体表面积。

有的读者自然会想到，在这种情况下应当将 area 声明为虚函数。

可以在基类 Point 中加一个 area 函数，并声明为虚函数：

```
virtual float area()const {return 0;}
```

其返回值为 0，表示“点”是没有面积的。

其实，在基类中并不使用这个函数，其返回值也是没有意义的。

为简化，可以不写出这种无意义的函数体，只给出函数的原型，并在后面加上“=0”，如

```
virtual float area()const =0; //纯虚函数
```

这就将 area 声明为一个纯虚函数(pure virtual function)。

纯虚函数是在声明虚函数时被“初始化”为 0 的函数。

声明纯虚函数的一般形式是

```
virtual 函数类型 函数名 (参数表列)=0;
```

注意：

①纯虚函数没有函数体；

②最后面的“=0”并不表示函数返回值为0，它只起形式上的作用，告诉编译系统“这是纯虚函数”；

③这是一个声明语句，最后应有分号。

纯虚函数只有函数的名字而不具备函数的功能，不能被调用。

它只是通知编译系统：“在这里声明一个虚函数，留待派生类中定义”。

在派生类中对此函数提供定义后，它才能具备函数的功能，可被调用。

纯虚函数的作用是在基类中为其派生类保留一个函数的名字，以便派生类根据需要对它进行定义。

如果在基类中没有保留函数名字，则无法实现多态性。

如果在一个类中声明了纯虚函数，而在其派生类中没有对该函数定义，则该虚函数在派生类中仍然为纯虚函数。

#### 12.4.2 抽象类

如果声明了一个类，一般可以用它定义对象。

但是在面向对象程序设计中，往往有一些类，它们不用来生成对象。

定义这些类的唯一目的是用它作为基类去建立派生类。

它们作为一种基本类型提供给用户，用户在这个基础上根据自己的需要定义出功能各异的派生类。

用这些派生类去建立对象。

一个优秀的软件工作者在开发一个大的软件时，决不会从头到尾都由自己编写程序代码，他会充分利用已有资源(例如类库)作为自己工作的基础。

这种不用来定义对象而只作为一种基本类型用作继承的类，称为抽象类(abstract class)，由于它常用作基类，通常称为抽象基类(abstract base class)。

凡是包含纯虚函数的类都是抽象类。

因为纯虚函数是不能被调用的，包含纯虚函数的类是无法建立对象的。

抽象类的作用是作为一个类族的共同基类，或者说，为一个类族提供一个公共接口。

一个类层次结构中当然也可不包含任何抽象类，每一层次的类都是实际可用的，可以用来建立对象的。

但是，许多好的面向对象的系统，其层次结构的顶部是一个抽象类，甚至顶部有好几层都是抽象类。

如果在抽象类所派生出的新类中对基类的所有纯虚函数进行了定义，那么这些函数就被赋予了功能，可以被调用。

这个派生类就不是抽象类，而是可以用来定义对象的具体类(concrete class)。

如果在派生类中没有对所有纯虚函数进行定义，则此派生类仍然是抽象类，不能用来定义对象。

虽然抽象类不能定义对象(或者说抽象类不能实例化)，但是可以定义指向抽象类数据的指针变量。

当派生类成为具体类之后，就可以用这种指针指向派生类对象，然后通过该指针调用虚函数，实现多态性的操作。

#### 12.4.3 应用实例

例 12.4 虚函数和抽象基类的应用。

在本章例 12.1 介绍了以 Point 为基类的点—圆—圆柱体类的层次结构。

现在要对它进行改写，在程序中使用虚函数和抽象基类。

类的层次结构的顶层是抽象基类 Shape(形状)。

Point(点), Circle(圆), Cylinder(圆柱体)都是 Shape 类的直接派生类和间接派生类。

下面是一个完整的程序，为了便于阅读，分段插入了一些文字说明。

程序如下：

第(1)部分

```
#include <iostream>
using namespace std;
//声明抽象基类 Shape
class Shape
{public:
virtual float area()const {return 0.0;}//虚函数
virtual float volume()const {return 0.0;}//虚函数
```

```
virtual void shapeName()const =0;//纯虚函数
```

```
};
```

第(2)部分

//声明 Point 类

```
class Point:public Shape//Point 是 Shape 的公用派生类
```

```
{public:
```

```
Point(float=0,float=0);
```

```
void setPoint(float ,float ); float getX( )const {return x;}float getY( )const {return y;}
```

```
virtual void shapeName( )const {cout<<"Point:";}//对虚函数进行再定义
```

```
friend ostream & operator <<(ostream &,const Point &);
```

```
protected:
```

```
float x,y;
```

```
};
```

//定义 Point 类成员函数

```
Point::Point(float a,float b)
```

```
{x=a;y=b;}
```

```
void Point::setPoint(float a,float b)
```

```
{x=a;y=b;}
```

```
ostream & operator <<(ostream &output,const Point &p)
```

```
{output<<"["<<p.x<<","<<p.y<<"]";
```

```
return output;
```

```
}
```

第(3)部分

//声明 Circle 类

```
class Circle:public Point
```

```
{public:
```

```
Circle(float x=0,float y=0,float r=0);
```

```
void setRadius(float );
```

```
float getRadius( )const;
```

```
virtual float area( )const;
```

```
virtual void shapeName( )const {cout<<"Circle:";}//对虚函数进行再定义
```

```
friend ostream &operator <<(ostream &,const Circle &);
```

```
protected :
```

```
float radius;
```

```
};
```

//声明 Circle 类成员函数

```
Circle::Circle(float a,float b,float r):Point(a,b),radius(r){}
```

```
void Circle::setRadius(float r):radius(r){}
```

```
float Circle::getRadius( )const {return radius;}
```

```
float Circle::area( )const {return 3.14159*radius*radius;}
```

```
ostream &operator <<(ostream &output,const Circle &c){output<<"["<<c.x<<","<<c.y<<"], r="<<c.radius;
```

```
return output;}
```

第(4)部分

//声明 Cylinder 类

```
class Cylinder:public Circle
```

```
{public:
```

```
Cylinder (float x=0,float y=0,float r=0,float h=0);
```

```

void setHeight(float );
virtual float area( )const;
virtual float volume( )const;
virtual void shapeName( )const {cout<<"Cylinder:";}//对虚函数进行再定义
friend ostream& operator <<(ostream&,const Cylinder&);
protected:
float height;
};
//定义 Cylinder 类成员函数
Cylinder::Cylinder(float a,float b,float r,float h):Circle(a,b,r),height(h){}
void Cylinder::setHeight(float h){height=h;}
float Cylinder::area( )const{return 2*Circle::area( )+2*3.14159*radius*height;}
float Cylinder::volume( )const{return Circle::area( )*height;}
ostream &operator <<(ostream &output,const Cylinder& cy){output<<"["<<cy.x<<","<<cy.y<<"],
r="<<cy.radius<<","h="<<cy.height; return output;}

```

第(5)部分

//main 函数

```

int main( )
{Point point(3.2,4.5);//建立 Point 类对象 point
Circle circle(2.4,1.2,5.6);
//建立 Circle 类对象 circle
Cylinder cylinder(3.5,6.4,5.2,10.5);
//建立 Cylinder 类对象 cylinder
point.shapeName();
//静态关联
cout<<point<<endl;
circle.shapeName();//静态关联
cout<<circle<<endl;
cylinder.shapeName();//静态关联
cout<<cylinder<<endl<<endl;
Shape *pt;//定义基类指针
pt=&point;//指针指向 Point 类对象
pt->shapeName( )//动态关联
cout<<"x="<<point.getX( )<<","y="<<point.getY( )<<"\narea="<<pt->area( )
<<"\nvolume="<<pt->volume( )<<"\n\n";
pt=&circle;//指针指向 Circle 类对象
pt->shapeName( )//动态关联
cout<<"x="<<circle.getX( )<<","y="<<circle.getY( )<<"\narea="<<pt->area( )
<<"\nvolume="<<pt->volume( )<<"\n\n";
pt=&cylinder;//指针指向 Cylinder 类对象
pt->shapeName( )//动态关联
cout<<"x="<<cylinder.getX( )<<","y="<<cylinder.getY( )<<"\narea="<<pt->area( )
<<"\nvolume="<<pt->volume( )<<"\n\n";
return 0;}

```

程序运行结果如下。

请读者对照程序分析。

Point:[3.2,4.5](Point 类对象 point 的数据:点的坐标)



Circle:[2.4,1.2], r=5.6 (Circle 类对象 circle 的数据:圆心和半径)

Cylinder:[3.5,6.4], r=5.5, h=10.5 (Cylinder 类对象 cylinder 的数据: 圆心、半径和高)

Point:x=3.2,y=4.5 (输出 Point 类对象 point 的数据:点的坐标)

area=0 (点的面积)

volume=0 (点的体积)

Circle:x=2.4,y=1.2 (输出 Circle 类对象 circle 的数据:圆心坐标)

area=98.5203 (圆的面积)

volume=0 (圆的体积)

Cylinder:x=3.5,y=6.4 (输出 Cylinder 类对象 cylinder 的数据:圆心坐标)

area=512.595 (圆的面积)

volume=891.96 (圆柱的体积)

从本例可以进一步明确以下结论:

(1)一个基类如果包含一个或一个以上纯虚函数,就是**抽象基类**。

抽象基类不能也不必要定义对象。

(2)抽象基类与普通基类不同,它一般并不是现实存在的对象的抽象(例如圆形(Circle)就是千千万万个实际的圆的抽象),它可以没有任何物理上的或其他实际意义方面的含义。

(3)在类的层次结构中,顶层或最上面的几层可以是**抽象基类**。

抽象基类体现了**本类族中各类的共性**,把各类中共有的成员函数集中在抽象基类中声明。

(4)抽象基类是本类族的**公共接口**。

或者说,从同一基类派生出的多个类有同一接口。

(5)区别**静态关联**和**动态关联**。

(6)如果在基类声明了虚函数,则在派生类中凡是与该函数有相同的函数名、函数类型、参数个数和类型的函数,均为**虚函数(不论在派生类中是否用 virtual 声明)**。

(7)使用虚函数提高了程序的**可扩充性**。

把类的声明与类的使用分离。

这对于设计类库的软件开发商来说尤为重要。

开发商设计了各种各样的类,但不向用户提供源代码,用户可以不知道类是怎样声明的,但是可以使用这些类来派生出自己的类。

利用虚函数和多态性,程序员的注意力集中在处理普遍性,而让执行环境处理特殊性。

**多态性把操作的细节留给类的设计者**(他们多为专业人员)去完成,而让**程序人员**(类的使用者)只需要做一些宏观性的工作,告诉系统做什么,而不必考虑怎么做,极大地简化了应用程序的编码工作,大大减轻了程序员的负担,也降低了学习和使用 C++编程的难度,使更多的人能更快地进入 C++程序设计的大门。

## 第 13 章 输入输出流

### 13.1 C++的输入和输出

### 13.2 标准输出流

### 13.3 标准输入流

### 13.4 文件操作与文件流

### 13.5 字符串流

以前所用到的输入和输出,都是以终端为对象的,即从键盘输入数据,运行结果输出到显示器屏幕上。从操作系统的角度看,每一个与主机相连的输入输出设备都被看作一个文件。除了以终端为对象进行输入和输出外,还经常用磁盘(光盘)作为输入输出对象,磁盘文件既可以作为输入文件,也可以作为输出文件。

程序的输入指的是从输入文件将数据传送给程序,程序的输出指的是从程序将数据传送给输出文件。C++的输入与输出包括以下方面的内容:

### 13.1 C++的输入和输出

#### 13.1.1 输入输出的含义

- (1) 对系统指定的标准设备的输入和输出。即从键盘输入数据,输出到显示器屏幕。这种输入输出称为标准的输入输出,简称标准 I/O。
- (2) 以外存磁盘文件为对象进行输入和输出,即从磁盘文件输入数据,数据输出到磁盘文件。以外存文件为对象的输入输出称为文件的输入输出,简称文件 I/O。
- (3) 对内存中指定的空间进行输入和输出。通常指定一个字符数组作为存储空间(实际上可以利用该空间存储任何信息)。这种输入和输出称为字符串输入输出,简称串 I/O。

C++采取不同的方法来实现以上种输入输出。

为了实现数据的有效流动,C++系统提供了庞大的 I/O 类库,调用不同的类去实现不同的功能。

在 C 语言中,用 `printf` 和 `scanf` 进行输入输出,往往不能保证所输入输出的数据是可靠的安全的。

在 C++的输入输出中,编译系统对数据类型进行严格的检查,凡是类型不正确的数据都不可能通过编译。因此 C++的 I/O 操作是类型安全(type safe)的。

C++的 I/O 操作是可扩展的,不仅可以用来输入输出标准类型的数据,也可以用于用户自定义类型的数据。

C++对标准类型的数据和对用户声明类型数据的输入输出,采用同样的方法处理。

C++通过 I/O 类库来实现丰富的 I/O 功能。C++的输入输出优于 C 语言中的 `printf` 和 `scanf`,但是比较复杂,要掌握许多细节。

#### 13.1.2 C++的 I/O 对 C 的发展——类型安全和

可扩展性

C++的输入输出流是指由若干字节组成的字节序列,这些字节中的数据按顺序从一个对象传送到另一对象。流表示了信息从源到目的端的流动。在输入操作时,字节流从输入设备(如键盘、磁盘)流向内存,在输出操作时,字节流从内存流向输出设备(如屏幕、打印机、磁盘等)。流中的内容可以是 ASCII 字符、二进制形式的数据、图形图像、数字音频视频或其他形式的信息。

实际上,在内存中为每一个数据流开辟一个内存缓冲区,用来存放流中的数据。流是与内存缓冲区相对应的,或者说,缓冲区中的数据就是流。

#### 13.1.3 C++的输入输出

在 C++中,输入输出流被定义为类。C++的 I/O 库中的类称为流类(stream class)。用流类定义的对象称为流对象。

`cout` 和 `cin` 并不是 C++语言中提供的语句,它们是 `iostream` 类的对象,在未学习类和对象时,在不致引起误解的前提下,为叙述方便,把它们称为 `cout` 语句和 `cin` 语句。

在学习了类和对象后,我们对 C++的输入输出应当有更深刻的认识。

##### 1. `iostream` 类库中有关的类

C++编译系统提供了用于输入输出的 `iostream` 类库。`iostream` 这个单词是由个部分组成的,即 i-o-stream,意为输入输出流。在 `iostream` 类库中包含许多用于输入输出的类。常用的见书中表.1。

`ios` 是抽象基类,由它派生出 `istream` 类和 `ostream` 类,两个类名中第个字母 i 和 o 分别代表输入(input)和输出(output)。`istream` 类支持输入操作,`ostream` 类支持输出操作,`iostream` 类支持输入输出操作。`iostream` 类是从 `istream` 类和 `ostream` 类通过多重继承而派生的类。其继承层次见图.1 表示。

图.1

C++对文件的输入输出需要用 `ifstream` 和 `ofstream` 类,两个类名中第个字母 i 和 o 分别代表输入和输出,第个字母 f 代表文件(file)。`ifstream` 支持对文件的输入操作,`ofstream` 支持对文件的输出操作。类 `ifstream` 继承了类 `istream`,类 `ofstream` 继承了类 `ostream`,类 `fstream` 继承了类 `iostream`。见图.2。

图.2

I/O 类库中还有其他类,见图。

图

## 2. 与 iostream 类库有关的头文件

iostream 类库中不同的类的声明被放在不同的头文件中,用户在自己的程序中用#include 命令包含了有关的头文件就相当于在本程序中声明了所需要用到的类。可以换一种说法:头文件是程序与类库的接口,iostream 类库的接口分别由不同的头文件来实现。常用的有

iostream 包含了对输入输出流进行操作所需的基本信息。

fstream 用于用户管理的文件的 I/O 操作。

strstream 用于字符串流 I/O。

stdiostream 用于混合使用 C 和 C++的 I/O 机制时。

iomanip 在使用格式化 I/O 时应包含此头文件。

## 3. 在 iostream 头文件中定义的流对象

在 iostream 头文件中定义的类有 ios,istream,ostream,iostream,istream \_withassign, ostream \_withassign,iostream \_withassign 等。

iostream.h 包含了对输入输出流进行操作所需的基本信息。因此大多数 C++程序都包括 iostream.h。在 iostream.h 头文件中不仅定义了有关的类,还定义了种流对象,见书中表.2。

cin 是 istream 的派生类 istream \_withassign 的对象,它是从标准输入设备(键盘)输入到内存的数据流,称为 cin 流或标准输入流。cout 是 ostream 的派生类 ostream \_withassign 的对象,它是从内存输入到标准输出设备(显示器)的数据流,称为 cout 流或标准输出流。

cerr 和 clog 作用相似,均为向输出设备(显示器)输出出错信息。因此用键盘输入时用 cin 流,向显示器输出时用 cout 流。向显示器输出出错信息时用 cerr 和 clog 流。

在 iostream 头文件中定义以上个流对象用以下的形式(以 cout 为例):

```
ostream cout (stdout);
```

在定义 cout 为 ostream 流类对象时,把标准输出设备 stdout 作为参数,这样它就与标准输出设备(显示器)联系起来,如果有

```
cout<<3;
```

就会在显示器的屏幕上输出。

## 4. 在 iostream 头文件中重载运算符

“<<”和“>>”本来在 C++中是被定义为左位移运算符和右位移运算符的,由于在 iostream 头文件对它们进行了重载,使它们能用作标准类型数据的输入和输出运算符。所以,在用它们的程序中必须用#include 命令把 iostream 包含到程序中。

```
#include <iostream>
```

在 istream 和 ostream 类(这两个类都是在 iostream 中声明的)中分别有一组成员函数对位移运算符“<<”和“>>”进行重载,以便能用它输入或输出各种标准数据类型的数据。对于不同的标准数据类型要分别进行重载,如

```
ostream operator << (int );//用于向输出流插入一个 int 数据
```

```
ostream operator << (float ); //用于向输出流插入一个 float 数据
```

```
ostream operator << (char ); //用于向输出流插入一个 char 数据
```

```
ostream operator << (char *); //用于向输出流插入一个字符串数据等。
```

如果在程序中有下面的表达式:

```
cout<<"C++";
```

根据第4章所介绍的知识,上面的表达式相当于

```
cout.operator<<("C++")
```

"C++"的值是其首字节地址,是字符型指针(char\*)类型,因此选择调用上面最后一个运算符重载函数,通过重载函数的函数体,将字符串插入到 cout 流中,函数返回流对象 cout。

在 istream 类中已将运算符“>>”重载为对以下标准类型的提取运算符: char,signed char,unsigned char,short,unsigned short,int,unsigned int,

long,unsigned long,float, double,long double, char\*,signed char\*,unsigned char\*等。



在 `ostream` 类中将“<<”重载为插入运算符,其适用类型除了以上的标准类型外,还增加了一个 `void*` 类型。如果想将“<<”和“>>”用于自己声明的类型的数,就不能简单地采用包含 `iostream` 头文件来解决,必须自己用第章的方法对“<<”和“>>”进行重载。

怎样理解运算符“<<”和“>>”的作用呢?它们指出了数据移动的方向,例如

```
>>a
```

箭头方向表示把数据放入 `a` 中。而

```
<<a
```

箭头方向表示从 `a` 中拿出数据。++

标准输出流是流向标准输出设备(显示器)的数据。

### 13.2 标准输出流

`ostream` 类定义了一个输出流对象,即 `cout`,`cerr`,`clog`。分述如下。

#### 1. `cout` 流对象

`cout` 是 `console output` 的缩写,意为在控制台(终端显示器)的输出。

(1) `cout` 不是 C++ 预定义的关键字,它是 `ostream` 流类的对象,在 `iostream` 中定义。

(2) 用“`cout<<`”输出基本类型的数时,可以不必考虑数是什么类型,系统会判断数的类型,并根据其类型选择调用与之匹配的运算符重载函数。

##### 13.2.1

(3) `cout` 流在内存中对应开辟了一个缓冲区,用来存放流中的数,当向 `cout` 流插入一个 `endl` 时,不论缓冲区是否已满,都立即输出流中所有数,然后插入一个换行符,并刷新流(清空缓冲区)。

(4) 在 `iostream` 中只对“<<”和“>>”运算符用于标准类型数的输入输出进行了重载,但未对用户声明的类型的输入输出进行重载。

#### 2. `cerr` 流对象

`cerr` 流对象是标准错误流。`cerr` 流已被指定为与显示器关联。`cerr` 的作用是向标准错误设备(standard error device)输出有关出错信息。`cerr` 与标准输出流 `cout` 的作用和用法差不多。但有一点不同: `cout` 流通常是传送到显示器输出,但也可以被重定向输出到磁盘文件,而 `cerr` 流中的信息只能在显示器输出。当调试程序时,往往不希望程序运行时的出错信息被送到其他文件,而要求在显示器上及时输出,这时应该用 `cerr`。`cerr` 流中的信息是用户根据需要指定的。

例.1 有一元二次方程  $ax^2+bx+c=0$ ,其一般解为

$x_{1,2}=(-b\pm\sqrt{b^2-4ac})/2a$ ,但若  $a=0$ ,或  $b^2-4ac<0$  时,用此公式出错。

编程序,从键盘输入  $a,b,c$  的值,求  $x_1$  和  $x_2$ 。如果  $a=0$  或  $b^2-4ac<0$ ,输出出错信息。

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{float a,b,c,disc;
  cout<<"please input a,b,c:";
  cin>>a>>b>>c;
  if (a==0)
    cerr<<"a is equal to zero,error!"<<endl;
  //将有关出错信息插入 cerr 流,在屏幕输出
  else
    if ((disc=b*b-4*a*c)<0)
      cerr<<"disc=b*b-4*a*c<0"<<endl; //将有关出错信息插入 cerr 流,在屏幕输出
    else
      {cout<<"x1="<<(-b+sqrt(disc))/(2*a)<<endl;
        cout<<"x2="<<(-b-sqrt(disc))/(2*a)<<endl;
      }
}
```

```
return 0;
}
```

运行情况如下:

①

please input a,b,c: 0 2 3 ✓

a is equal to zero,error!

②please input a,b,c: 5 2 3 ✓

sc=b\*b-4\*a\*c<0

③please input a,b,c: 1 2.5 1.5 ✓

x1=-1

x2=-1.5

### 3. clog 流对象

clog 流对象也是标准错误流,它是 console log 的缩写。它的作用和 cerr 相同,都是在终端显示器上显示出错信息。区别: cerr 是不经过缓冲区,直接向显示器上输出有关信息,而 clog 中的信息存放在缓冲区中,缓冲区满后或遇 endl 时向显示器输出。

在输出数据时,有时希望数据按指定的格式输出。有两种方法可以达到此目的。一种是第章已介绍过的使用控制符的方法;第种是使用流对象的有关成员函数。

#### 1. 使用控制符控制输出格式

输出数据的控制符见书中表.3。

应当注意,这些控制符是在头文件 iomanip 中定义的,因而程序中应当包含 iomanip。

#### 13.2.2

例.2 用控制符控制输出格式。

```
#include <iostream>
```

```
#include <iomanip> //不要忘记包含此头文件
```

```
using namespace std;
```

```
int main()
```

```
{int a;
```

```
cout<<"input a:";
```

```
cin>>a;
```

```
cout<<"dec:"<<dec<<a<<endl; //以十进制形式输出整数
```

```
cout<<"hex:"<<hex<<a<<endl; //以十六进制形式输出整数 a
```

```
cout<<"oct:"<<setbase(8)<<a<<endl; //以八进制形式输出整数 a
```

```
char *pt="China"; //pt 指向字符串"China"
```

```
cout<<setw(10)<<pt<<endl; //指定域宽为,输出字符串
```

```
cout<<setfill('*')<<setw(10)<<pt<<endl; //指定域宽,输出字符串,空白处以'*'填充
```

```
double pi=22.0/7.0; //计算 pi 值
```

```
cout<<setiosflags(ios::scientific)<<setprecision(8); //按指数形式输出,8 位小数
```

```
cout<<"pi="<<pi<<endl; //输出 pi 值
```

```
cout<<"pi="<<setprecision(4)<<pi<<endl; //改为位小数
```

```
cout<<"pi="<<setiosflags(ios::fixed)<<pi<<endl; //改为小数形式输出
```

```
return 0;
```

```
}
```

运行结果如下:

input a:34 ✓ (输入 a 的值)

dec:34 (十进制形式)

hex:22 (十六进制形式)

oct:42 (八进制形式)

|                   |                 |
|-------------------|-----------------|
| China             | (域宽为)           |
| *****China        | (域宽为,空白处以'*'填充) |
| pi=3.14285714e+00 | (指数形式输出,8 位小数)  |
| pi=3.1429e+00     | (指数形式输出,4 位小数)  |
| pi=3.143          | (小数形式输出,精度仍为)   |

## 2. 用流对象的成员函数控制输出格式

除了可以用控制符来控制输出格式外,还可以通过调用流对象 `cout` 中用于控制输出格式的成员函数来控制输出格式。用于控制输出格式的常用的成员函数见书中表.4。

流成员函数 `setf` 和控制符 `setiosflags` 括号中的参数表示格式状态,它是通过格式标志来指定的。格式标志在类 `ios` 中被定义为枚举值。因此在引用这些格式标志时要在前面加上类名 `ios` 和域运算符“`::`”。格式标志见书中表.5。

例.3 用流控制成员函数输出数据。

```
#include <iostream>
using namespace std;
int main( )
{int a=21
  cout.setf(ios::showbase);//显示基数符号(0x 或)
  cout<<"dec:"<<a<<endl;          //默认以十进制形式输出 a
  cout.unsetf(ios::dec);           //终止十进制的格式设置
  cout.setf(ios::hex);             //设置以十六进制输出的状态
  cout<<"hex:"<<a<<endl;          //以十六进制形式输出 a
  cout.unsetf(ios::hex);           //终止十六进制的格式设置
  cout.setf(ios::oct);             //设置以八进制输出的状态
  cout<<"oct:"<<a<<endl;          //以八进制形式输出 a
  cout.unsetf(ios::oct);
  char *pt="China";               //pt 指向字符串"China"
  cout.width(10);                  //指定域宽为
  cout<<pt<<endl;                 //输出字符串
  cout.width(10);                  //指定域宽为
  cout.fill('*');                  //指定空白处以'*'填充
  cout<<pt<<endl;                 //输出字符串
  double pi=22.0/7.0;              //输出 pi 值
  cout.setf(ios::scientific);      //指定用科学记数法输出
  cout<<"pi=";                     //输出"pi="
  cout.width(14);                  //指定域宽为
  cout<<pi<<endl;                 //输出 pi 值
  cout.unsetf(ios::scientific);    //终止科学记数法状态
  cout.setf(ios::fixed);           //指定用定点形式输出
  cout.width(12);                  //指定域宽为
  cout.setf(ios::showpos);         //正数输出“+”号
  cout.setf(ios::internal);        //数符出现在左侧
  cout.precision(6);               //保留位小数
  cout<<pi<<endl;                 //输出 pi,注意数符“+”的位置
  return 0;
}
```

运行情况如下:

dec:21(十进制形式)

hex:0x15 (十六进制形式,以 x 开头)

oct:025 (八进制形式,以开头)

China (域宽为)

\*\*\*\*\*China (域宽为,空白处以 '\*' 填充)

pi=\*\*3.142857e+00 (指数形式输出,域宽,默认位小数)

+\*\*\*3.142857 (小数形式输出,精度为,最左侧输出数符“+”)

ostream 类除了提供上面介绍过的用于格式控制的成员函数外,还提供了专用于输出单个字符的成员函数 put。如

```
cout.put('a');
```

调用该函数的结果是在屏幕上显示一个字符 a。put 函数的参数可以是字符或字符的 ASCII 代码(也可以是一个整型表达式)。如

```
cout.put(65+32);
```

也显示字符 a,因为是字符 a 的 ASCII 代码。

可以在一个语句中连续调用 put 函数。如

```
cout.put(71).put(79).put(79).put(68).put("\n");
```

在屏幕上显示 GOOD。

### 13.2.3 用流成员函数 put 输出字符

例.4 有一个字符串"BASIC",要求把它们按相反的顺序输出。

```
#include <iostream>
```

```
using namespace std;
```

```
int main( )
```

```
{char *a="BASIC";//字符指针指向'B'
```

```
for(int i=4;i>=0;i--)
```

```
    cout.put(*(a+i));           //从最后一个字符开始输出
```

```
    cout.put("\n");
```

```
    return 0;
```

```
}
```

运行时在屏幕上输出:

CISAB

还可以用 putchar 函数输出一个字符。putchar 函数是 C 语言中使用的,在 stdio.h 头文件中定义。C++ 保留了这个函数,在 iostream 头文件中定义。

例.4 也可以改用 putchar 函数实现。

```
#include <iostream>//也可以用#include <stdio.h>,同时不要下一行
```

```
using namespace std;
```

```
int main( )
```

```
{char *a="BASIC";
```

```
for(int i=4;i>=0;i--)
```

```
    putchar(*(a+i));
```

```
    putchar("\n");
```

```
}
```

运行结果与前相同。

成员函数 put 不仅可以用 cout 流对象来调用,而且也可以用 ostream 类的其他流对象调用。

标准输入流是从标准输入设备(键盘)流向程序的数据。

### 13.3 标准输入流

在上一节中已知,在头文件 iostream.h 中定义了 cin,cout,cerr,clog 4 个流对象,cin 是输入流,cout,cerr,clog 是输出流。

cin 是 istream 类的对象,它从标准输入设备(键盘)获取数据,程序中的变量通过流提取符“>>”从流中提取数

据。流提取符“>>”从流中提取数据时通常跳过输入流中的空格、tab 键、换行符等空白字符。注意：只有在输入完数据再按回车键后,该行数据才被送入键盘缓冲区,形成输入流,提取运算符“>>”才能从中提取数据。需要注意保证从流中读取数据能正常进行。

### 13.3

例.5 通过测试 cin 的真值,判断流对象是否处于正常状态。

```
#include <iostream>
using namespace std;
int main( )
{float grade;
  cout<<"enter grade:";
  while(cin>>grade)//能从 cin 流读取数据
  {if(grade>=85) cout<<grade<<"GOOD!"<<endl;
  if(grade<60) cout<<grade<<"fail!"<<endl;
  cout<<"enter grade:";
  }
  cout<<"The end."<<endl;
  return 0;
}
```

运行情况如下:

enter grade: 67✓

enter grade: 89✓

89 GOOD!

enter grade: 56✓

56 fail!

enter grade: 100✓

100 GOOD!

enter grade: ^Z✓//键入文件结束符

The end.

如果某次输入的数据为

enter grade: 100/2✓

输出"The end."。

在不同的 C++ 系统下运行此程序,在最后的处理上有些不同。以上是在 GCC 环境下运行程序的结果,如果在 VC++ 环境下运行此程序,在键入 Ctrl+Z 时,程序运行马上结束,不输出"The end."。

除了可以用 cin 输入标准类型的数据外,还可以用 istream 类流对象的一些成员函数,实现字符的输入。

#### 1. 用 get 函数读入一个字符

流成员函数 get 有种形式: 无参数的,有一个参数的,有个参数的。

##### (1) 不带参数的 get 函数

其调用形式为

```
cin.get()
```

用来从指定的输入流中提取一个字符,函数的返回值就是读入的字符。若遇到输入流中的文件结束符,则函数值返回文件结束标志 EOF(End Of File) 。

### 13.3.2 用于字符输入的流成员函数

例.6 用 get 函数读入字符。

```
#include <iostream>
int main( )
{int c;
  cout<<"enter a sentence:"<<endl;
```

```

while((c=cin.get())!=EOF)
    cout.put(c);
return 0;
}

```

运行情况如下:

enter a sentence:

I study C++ very hard.↵ (输入一行字符)

I study C++ very hard. (输出该行字符)

^Z↵ (程序结束)

C 语言中的 `getchar` 函数与流成员函数 `cin.get()` 的功能相同,C++保留了 C 的这种用法。

(2) 有一个参数的 `get` 函数

其调用形式为

```
cin.get(ch)
```

其作用是从输入流中读取一个字符,赋给字符变量 `ch`。如果读取成功则函数返回非值(真),如失败(遇文件结束符) 则函数返回值(假)。例.6 可以改写如下:

```
#include <iostream>
```

```
int main( )
```

```
{char c;
```

```
    cout<<"enter a sentence:"<<endl;
```

```
    while(cin.get(c)) //读取一个字符赋给字符变量 c,如果读取成功,cin.get(c)为真
```

```
{cout.put(c);}
```

```
    cout<<"end"<<endl;
```

```
    return 0;
```

```
}
```

(3) 有个参数的 `get` 函数

其调用形式为

```
cin.get(字符数组,字符个数 n,终止字符)
```

或

```
cin.get(字符指针,字符个数 n,终止字符)
```

其作用是从输入流中读取 `n-1` 个字符,赋给指定的字符数组(或字符指针指向的数组),如果在读取 `n-1` 个字符之前遇到指定的终止字符,则提前结束读取。如果读取成功则函数返回非值(真),如失败(遇文件结束符) 则函数返回值(假)。再将例.6 改写如下:

```
#include <iostream>
```

```
using namespace std;
```

```
int main( )
```

```
{char ch[20];
```

```
    cout<<"enter a sentence:"<<endl;
```

```
    cin.get(ch,10,'\n');//指定换行符为终止字符
```

```
    cout<<ch<<endl;
```

```
    return 0;
```

```
}
```

运行情况如下:

enter a sentence:

I study C++ very hard.↵

I study

`get` 函数中第个参数可以省写,此时默认为 `'\n'`。下面两行等价:

```
cin.get(ch,10,'\n');
```

```
cin.get(ch,10);
```

终止字符也可以用其他字符。如

```
cin.get(ch,10,'x');
```

## 2. 用成员函数 getline 函数读入一行字符

getline 函数的作用是从输入流中读取一行字符,其用法与带个参数的 get 函数类似。即

cin.getline(字符数组(或字符指针),字符个数 n,终止标志字符)

例.7 用 getline 函数读入一行字符。

```
#include <iostream>
```

```
using namespace std;
```

```
int main( )
```

```
{char ch[20];
```

```
cout<<"enter a sentence:"<<endl;
```

```
cin>>ch;
```

```
cout<<"The string read with cin is:"<<ch<<endl;
```

```
cin.getline(ch,20,'/');//读个字符或遇'/'结束
```

```
cout<<"The second part is:"<<ch<<endl;
```

```
cin.getline(ch,20);
```

//读个字符或遇'\n'结束

```
cout<<"The third part is:"<<ch<<endl;
```

```
return 0;
```

```
}
```

程序运行情况如下:

```
enter a sentence: I like C++./I study C++./I am happy.✓
```

```
The string read with cin is:I
```

```
The second part is: like C++.
```

```
The third part is:I study C++./I am h
```

除了以上介绍的用于读取数据的成员函数外,istream 类还有其他在输入数据时用得着的一些成员函数。常用的有以下几种:

### 1. eof 函数

eof 是 end of file 的缩写,表示“文件结束”。从输入流读取数据,如果到达文件末尾(遇文件结束符),eof 函数值为非零值(表示真),否则为(假)。

### 13.3.3 istream 类的其他成员

例.8 逐个读入一行字符,将其中的非空格字符输出。

```
#include <iostream>
```

```
using namespace std;
```

```
int main( )
```

```
{char c;
```

```
while(!cin.eof())//eof( )为假表示未遇到文件结束符
```

```
if((c=cin.get( ))!=' ') //检查读入的字符是否为空格字符
```

```
cout.put(c);
```

```
return 0;
```

```
}
```

运行情况如下:

```
C++ is very interesting.✓
```

```
C++isveryinteresting.
```

```
^Z(结束)
```

### 2. peek 函数

peek 是“观察”的意思,peek 函数的作用是观测下一个字符。其调用形式为



```
c=cin.peek();cin.peek
```

函数的返回值是指针指向的当前字符,但它只是观测,指针仍停留在当前位置,并不后移。如果要访问的字符是文件结束符,则函数值是 EOF(-1)。

### 3. putback 函数

其调用形式为

```
cin.putback(ch);
```

其作用是将前面用 get 或 getline 函数从输入流中读取的字符 ch 返回到输入流,插入到当前指针位置,以供后面读取。

例.9 peek 函数和 putback 函数的用法。

```
#include <iostream>
using namespace std;
int main( )
{char c[20];
 int ch;
 cout<<"please enter a sentence:"<<endl;
 cin.getline(c,15,'/');
 cout<<"The first part is:"<<c<<endl;
 ch=cin.peek();//观看当前字符
 cout<<"The next character(ASCII code) is:"<<ch<<endl;
 cin.putback(c[0]);           //将T插入到指针所指处
 cin.getline(c,15,'/');
 cout<<"The second part is:"<<c<<endl;
 return 0;
}
```

运行情况如下:

please enter a sentence:

I am a boy./ am a student./ ✓

The first part is:I am a boy.

The next character(ASCII code) is:32(下一个字符是空格)

The second part is:I am a student

图.4

### 4. ignore 函数

其调用形式为

```
cin.ignore(n, 终止字符)
```

函数作用是跳过输入流中 n 个字符,或在遇到指定的终止字符时提前结束(此时跳过包括终止字符在内的若干字符)。如

```
ignore(5, 'A')//跳过输入流中5个字符,遇'A'后就不再跳了
```

也可以不带参数或只带一个参数。如

```
ignore( )(n 默认值为,终止字符默认为 EOF)
```

相当于

ignore(1,EOF) 例.10 用 ignore 函数跳过输入流中的字符。

先看不用 ignore 函数的情况:

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



```

{char ch[20];
cin.get(ch,20,'/');
cout<<"The first part is:"<<ch<<endl;
cin.get(ch,20,'/');
cout<<"The second part is:"<<ch<<endl;
return 0;
}

```

运行结果如下:

I like C++./I study C++./I am happy.✓

The first part is:I like C++.

The second part is:(字符数组 ch 中没有从输入流中读取有效字符)

如果希望第二个 cin.get 函数能读取 "I study C++.", 就应该设法跳过输入流中第一个 '/', 可以用 ignore 函数来实现此目的, 将程序改为

```

#include <iostream>
using namespace std;
int main()
{char ch[20];
cin.get(ch,20,'/');
cout<<"The first part is:"<<ch<<endl;
cin.ignore( );//跳过输入流中一个字符
cin.get(ch,20,'/');
cout<<"The second part is:"<<ch<<endl;
return 0;
}

```

运行结果如下:

I like C++./I study C++./I am happy.✓

The first part is:I like C++.

The second part is:I study C++.

以上介绍的各个成员函数,不仅可以用 cin 流对象来调用,而且也可以用 istream 类的其他流对象调用。

迄今为止,我们讨论的输入输出是以系统指定的标准设备(输入设备为键盘,输出设备为显示器)为对象的。在实际应用中,常以磁盘文件作为对象。即从磁盘文件读取数据,将数据输出到磁盘文件。

所谓“文件”,一般指存储在外部介质上数据的集合。一批数据是以文件的形式存放在外部介质上的。操作系统是以文件为单位对数据进行管理的。要向外部介质上存储数据也必须先建立一个文件(以文件名标识),才能向它输出数据。

## 13.4 文件操作与文件流

### 13.4.1 文件的

外存文件包括磁盘文件、光盘文件和 U 盘文件。目前使用最广泛的是磁盘文件。

对用户来说,常用到的文件有两大类,一类是程序文件(program file)。一类是数据文件(data file)。程序中的输入和输出的对象就是数据文件。

根据文件中数据的组织形式,可分为 ASCII 文件和二进制文件。

对于字符信息,在内存中是以 ASCII 代码形式存放的,因此,无论用 ASCII 文件输出还是用二进制文件输出,其数据形式是一样的。但是对于数值数据,二者是不同的。例如有一个长整数,在内存中占个字节,如果按内部格式直接输出,在磁盘文件中占个字节,如果将它转换为 ASCII 码形式输出,则要占个字节,见图.5。

图.5

C++ 提供低级的 I/O 功能和高级的 I/O 功能。高级的 I/O 功能是把若干个字节组合为一个有意义的单位,然后以 ASCII 字符形式输入和输出。传输大容量的文件时由于数据格式转换,速度较慢,效率不高。

所谓低级的 I/O 功能是以字节为单位输入和输出的,在输入和输出时不进行数据格式的转换。这种输入输出

速度快、效率高,一般大容量的文件传输用无格式转换的 I/O。但使用时会感到不太方便。

文件流是以外存文件为输入输出对象的数据流。输出文件流是从内存流向外存文件的数据,输入文件流是从外存文件流向内存的数据。每一个文件流都有一个内存缓冲区与之对应。

请区分文件流与文件的概念。文件流本身不是文件,而只是以文件为输入输出对象的流。若要对磁盘文件输入输出,就必须通过文件流来实现。

在 C++ 的 I/O 类库中定义了几种文件类,专门用于对磁盘文件的输入输出操作。在图.2 中可以看到除了已介绍过的标准输入输出流类 `istream`,`ostream` 和 `iostream` 类外,还有个用于文件操作的文件类:

#### 13.4.2 文件流类与文件流

(1) `ifstream` 类,它是从 `istream` 类派生的。用来支持从磁盘文件的输入。

(2) `ofstream` 类,它是从 `ostream` 类派生的。用来支持向磁盘文件的输出。

(3) `fstream` 类,它是从 `iostream` 类派生的。用来支持对磁盘文件的输入输出。

要以磁盘文件为对象进行输入输出,必须定义一个文件流类的对象,通过文件流对象将数据从内存输出到磁盘文件,或者通过文件流对象从磁盘文件将数据输入到内存。

其实在用标准设备为对象的输入输出中,也是要定义流对象的,如 `cin`,`cout` 就是流对象,C++ 是通过流对象进行输入输出的。

由于 `cin`,`cout` 已在 `iostream.h` 中事先定义,所以用户不需自己定义。在用磁盘文件时,由于情况各异,无法事先统一定义,必须由用户自己定义。此外,对磁盘文件的操作是通过文件流对象(而不是 `cin` 和 `cout`)实现的。文件流对象是用文件流类定义的,而不是用 `istream` 和 `ostream` 类来定义的。

可以用下面的方法建立一个输出文件流对象: `ofstream outfile;`

现在在程序中定义了 `outfile` 为 `ofstream` 类(输出文件流类)的对象。但是有一个问题还未解决: 在定义 `cout` 时已将它和标准输出设备建立关联,而现在虽然建立了一个输出文件流对象,但是还未指定它向哪一个磁盘文件输出,需要在使用时加以指定。

##### 1. 打开磁盘文件

打开文件是指在文件读写之前做必要的准备工作,包括:

(1) 为文件流对象和指定的磁盘文件建立关联,以便使文件流流向指定的磁盘文件。

(2) 指定文件的工作方式。

以上工作可以通过两种不同的方法实现。

(1) 调用文件流的成员函数 `open`。如

```
ofstream outfile; //定义 ofstream 类(输出文件流类)对象 outfile
outfile.open("fl.dat",ios::out); //使文件流与 fl.dat 文件建立关联
```

调用成员函数 `open` 的一般形式为

#### 13.4.3 文件的打开与

文件流对象.`open`(磁盘文件名,输入输出方式);

磁盘文件名可以包括路径,如 `"c:\\new\\fl.dat"`,如缺省路径,则默认为当前目录下的文件。

(2) 在定义文件流对象时指定参数

在声明文件流类时定义了带参数的构造函数,其中包含了打开磁盘文件的功能。因此,可以在定义文件流对象时指定参数,调用文件流类的构造函数来实现打开文件的功能。如

```
ostream outfile("fl.dat",ios::out);
```

一般多用此形式,比较方便。作用与 `open` 函数相同。

输入输出方式是在 `ios` 类中定义的,它们是枚举常量,有多种选择,见书中表.6。

说明:

①新版本的 I/O 类库中不提供 `ios::nocreate` 和 `ios::noreplace`。

②每一个打开的文件都有一个文件指针。

③可以用“位或”运算符“|”对输入输出方式进行组合。

④如果打开操作失败,`open` 函数的返回值为(假),如果是用调用构造函数的方式打开文件的,则流对象的值为。

##### 2. 关闭磁盘文件

在对已打开的磁盘文件的读写操作完成后,应关闭该文件。关闭文件用成员函数 `close`。如

`outfile.close();` //将输出文件流所关联的磁盘文件关闭

所谓关闭,实际上是解除该磁盘文件与文件流的关联,原来设置的工作方式也失效,这样,就不能再通过文件流对该文件进行输入或输出。此时可以将文件流与其他磁盘文件建立关联,通过文件流对新的文件进行输入或输出。如

`outfile.open("f2.dat",ios::app|ios::nocreate);`

此时文件流 `outfile` 与 `f2.dat` 建立关联,并指定了 `f2.dat` 的工作方式。

如果文件的每一个字节中均以 ASCII 代码形式存放数据,即一个字节存放一个字符,这个文件就是 ASCII 文件(或称字符文件)。程序可以从 ASCII 文件中读入若干个字符,也可以向它输出一些字符。

对 ASCII 文件的读写操作可以用以下两种方法:

(1) 用流插入运算符“<<”和流提取运算符“>>”输入输出标准类型的数据。

(2) 用本章.2.3 节和.3.2 节中介绍的文件流的 `put`,`get`,`getline` 等成员函数进行字符的输入输出。

#### 13.4.4 对 ASCII 文件

例.11 有一个整型数组,含个元素,从键盘输入个整数给数组,将此数组送到磁盘文件中存放。

```
#include <fstream>
using namespace std;
int main()
{int a[10];
  ofstream outfile("f1.dat",ios::out); //定义文件流对象,打开磁盘文件"f1.dat"
  if(!outfile)                          //如果打开失败,outfile 返回值
  {cerr<<"open error!"<<endl;
    exit(1);
  }
  cout<<"enter 10 integer numbers:"<<endl;
  for(int i=0;i<10;i++)
  {cin>>a[i];
    outfile<<a[i]<<" ";                //向磁盘文件"f1.dat"输出数据
  }
  outfile.close();                      //关闭磁盘文件"f1.dat"
  return 0;
}
```

运行情况如下:

enter 10 integer numbers:

1 3 5 2 4 6 10 8 7 9 ✓

请注意: 在向磁盘文件输出一个数据后,要输出一个(或几个)空格或换行符,以作为数据间的分隔,否则以后从磁盘文件读数据时,10 个整数的数字连成一片无法区分。

例.12 从例.11 建立的数据文件 `f1.dat` 中读入个整数放在数组中,找出并输出个数中的最大者和它在数组中的序号。

```
#include <fstream>
int main()
{int a[10],max,i,order;
  ifstream infile("f1.dat",ios::in|ios::nocreate);
  //定义输入文件流对象,以输入方式打开磁盘文件 f1.dat
  if(!infile)
  {cerr<<"open error!"<<endl;
    exit(1);
  }
  for(i=0;i<10;i++)
```

```

    {infile>>a[i];//从磁盘文件读入个整数,顺序存放在 a 数组中
    cout<<a[i]<<" ";}          //在显示器上顺序显示个数
    cout<<endl;
    max=a[0];
    order=0;
    for(i=1;i<10;i++)
        if(a[i]>max)
        {max=a[i];              //将当前最大值放在 max 中
        order=i;                //将当前最大值的元素序号放在 order 中
        }
    cout<<"max="<<max<<endl<<"order="<<order<<endl;
    infile.close();
    return 0;
}

```

运行情况如下:

1 3 5 2 4 6 10 8 7 9(在磁盘文件中存放的个数)

max=10 (最大值为)

order=6 (最大值是数组中序号为的元素)

例.13 从键盘读入一行字符,把其中的字母字符依次存放在磁盘文件 f2.dat 中。再把它从磁盘文件读入程序,将其中的小写字母改为大写字母,再存入磁盘文件 f3.dat。

```

#include <fstream>
using namespace std;
// save_to_file 函数从键盘读入一行字符,并将其中的字母存入磁盘文件
void save_to_file()
{ofstream outfile("f2.dat");
//定义输出文件流对象 outfile,以输出方式打开磁盘文件 f2.dat
if(!outfile)
    {cerr<<"open f2.dat error!"<<endl;
    exit(1);
    }
    char c[80];
    cin.getline(c,80);//从键盘读入一行字符
    for(int i=0;c[i]!='\0';i++) //对字符逐个处理,直到遇'\0'为止
    if(c[i]>=65 && c[i]<=90||c[i]>=97 && c[i]<=122)//如果是字母字符
    {outfile.put(c[i]);          //将字母字符存入磁盘文件 f2.dat
    cout<<c[i];                 //同时送显示器显示
    cout<<endl;
    outfile.close();            //关闭 f2.dat
    }

//从磁盘文件 f2.dat 读入字母字符,将其中的小写字母改为大写字母,再存入 f3.dat
void get_from_file()
{char ch;
    ifstream infile("f2.dat",ios::in|ios::nocreate);
//定义输入文件流 outfile,以输入方式打开磁盘文件 f2.dat
if(!infile)
    {cerr<<"open f2.dat error!"<<endl;

```

```

        exit(1);
    }
ofstream outfile("f3.dat");
    //定义输出文件流 outfile,以输出方式打开磁盘文件 f3.dat
if(!outfile)
    {cerr<<"open f3.dat error!"<<endl;
    exit(1);
    }
while(infile.get(ch))//当读取字符成功时执行下面的复合语句
    {if(ch>=97 && ch<=122)        //判断 ch 是否为小写字母
    ch=ch-32;                        //将小写字母变为大写字母
    outfile.put(ch);                //将该大写字母存入磁盘文件 f3.dat
    cout<<ch;                      //同时在显示器输出
    }
cout<<endl;
infile.close( );                  //关闭磁盘文件 f2.dat
outfile.close();                  //关闭磁盘文件 f3.dat
}
int main( )
{save_to_file( );                 //调用 save_to_file( ),从键盘读入一行字符并将其中的字母存入磁盘文件 f2.dat
get_from_file( );
    //调用 get_from_file(),从 f2.dat 读入字母字符,改为大写字母,再存入 f3.dat
return 0;
}

```

运行情况如下:

New Beijing, Great Olypic, 2008, China.✓

NewBeijingGreatOlypicChina(将字母写入磁盘文件 f2.dat,同时在屏幕显示)

NEWBEIJINGGREATOLYPICCHINA (改为大写字母)

磁盘文件 f3.dat 的内容虽然是 ASCII 字符,但人们是不能直接看到的,如果想从显示器上观看磁盘上 ASCII 文件的内容,可以采用以下两个方法:

(1) 在 DOS 环境下用 TYPE 命令,如

D:\C++>TYPE f3.dat✓(假设当前目录是 D:\C++)

在显示屏上会输出

NEWBEIJINGGREATOLYPICCHINA

如果用 GCC 编译环境,可选择 File 菜单中的 DOS Shell 菜单项,即可进入 DOS 环境。想从 DOS 返回 GCC 主窗口,从键盘输入 exit 即可。

(2) 编一程序将磁盘文件内容读入内存,然后输出到显示器。可以编一个专用函数。

```

#include <fstream>
using namespace std;
void display_file(char *filename)
{ifstream infile(filename,ios::in|ios::nocreate);
if(!infile)
    {cerr<<"open error!"<<endl;
    exit(1);}
char ch;
while(infile.get(ch))
cout.put(ch);
}

```

```
cout<<endl;
infile.close();
}
```

然后在调用时给出文件名即可:

```
int main( )
{display_file("f3.dat");//将 f3.dat 的入口地址传给形参 filename
return 0;
}
```

运行时输出 f3.dat 中的字符:

NEWBEIJINGGREATOLYPICCHINA

二进制文件不是以 ASCII 代码存放数据的,它将内存中数据存储形式不加转换地传送到磁盘文件,因此它又称为内存数据的映像文件。因为文件中的信息不是字符数据,而是字节中的二进制形式的信息,因此它又称为字节文件。

对二进制文件的操作也需要先打开文件,用完后要关闭文件。在打开时要用 `ios::binary` 指定为以二进制形式传送和存储。二进制文件除了可以作为输入文件或输出文件外,还可以是既能输入又能输出的文件。这是和 ASCII 文件不同的地方。

#### 13.4.5 对二进制文件的操作

##### 1. 用成员函数 read 和 write 读写二进制文件

对二进制文件的读写主要用 `istream` 类的成员函数 `read` 和 `write` 来实现。这两个成员函数的原型为

```
istream& read(char *buffer,int len);
```

```
ostream& write(const char * buffer,int len);
```

字符指针 `buffer` 指向内存中一段存储空间。`len` 是读写的字节数。调用的方式为

a. `write(p1,50);`

b. `read(p2,30);`

例.14 将一批数据以二进制形式存放在磁盘文件中。

```
#include <fstream>
using namespace std;
struct student
{char name[20];
int num;
int age;
char sex;
};
int main( )
{student stud[3]={"Li",1001,18,'f',"Fun",1002,19,'m',"Wang",1004,17,'f'};
ofstream outfile("stud.dat",ios::binary);
if(!outfile)
{cerr<<"open error!"<<endl;
abort( );//退出程序
}
for(int i=0;i<3;i++)
outfile.write((char*)&stud[i],sizeof(stud[i]));
outfile.close( );
return 0;
}
```

其实可以一次输出结构体数组的个元素,将 `for` 循环的两行改为以下一行:

```
outfile.write((char*)&stud[0],sizeof(stud));
```

执行一次 write 函数即输出了结构体数组的全部数据。

可以看到,用这种方法一次可以输出一批数据,效率较高。在输出的数据之间不必加入空格,在一次输出之后也不必加回车换行符。在以后从该文件读入数据时不是靠空格作为数据的间隔,而是用字节数来控制。

例.15 将刚才以二进制形式存放在磁盘文件中的数据读入内存并在显示器上显示。

```
#include <fstream>
using namespace std;
struct student
{string name;
 int num;
 int age;
 char sex;
};
int main( )
{student stud[3];
 int i;
 ifstream infile("stud.dat",ios::binary);
 if(!infile)
 {cerr<<"open error!"<<endl;
 abort( );
 }
 for(i=0;i<3;i++)
 infile.read((char*)&stud[i],sizeof(stud[i]));
 infile.close( );
 for(i=0;i<3;i++)
 {cout<<"NO."<<i+1<<endl;
 cout<<"name:"<<stud[i].name<<endl;
 cout<<"num:"<<stud[i].num<<endl;;
 cout<<"age:"<<stud[i].age<<endl;
 cout<<"sex:"<<stud[i].sex<<endl<<endl;
 }
 return 0;
}
```

运行时在显示器上显示:

NO.1

name: Li  
num: 1001  
age: 18  
sex: f

NO.2

name: Fun  
num: 1001  
age: 19  
sex: m

NO.3

name: Wang



num: 1004

age: 17

sex: f

请思考: 能否一次读入文件中的全部数据,如

```
infile.read((char*)&stud[0],sizeof(stud));
```

## 2. 与文件指针有关的流成员函数

在磁盘文件中有一个文件指针,用来指明当前应进行读写的位置。对于二进制文件,允许对指针进行控制,使它按用户的意图移动到所需的位置,以便在该位置上进行读写。文件流提供一些有关文件指针的成员函数。为了查阅方便,将它们归纳为书中表.7,并作必要的说明。

说明:

(1) 这些函数名的第一个字母或最后一个字母不是 g 就是 p。

(2) 函数参数中的“文件中的位置”和“位移量”已被指定为 long 型整数,以字节为单位。“参照位置”可以是下面三者之一:

ios::beg 文件开头(beg 是 begin 的缩写),这是默认值。

ios::cur 指针当前的位置(cur 是 current 的缩写)。

ios::end 文件末尾。

它们是在 ios 类中定义的枚举常量。

举例如下:

```
infile.seekg(100);//输入文件中的指针向前移到字节位置
```

```
infile.seekg(-50,ios::cur);    //输入文件中的指针从当前位置后移字节
```

```
outfile.seekp(-75,ios::end);    //输出文件中的指针从文件尾后移字节
```

## 3. 随机访问二进制数据文件

一般情况下读写是顺序进行的,即逐个字节进行读写。但是对于二进制数据文件来说,可以利用上面的成员函数移动指针,随机地访问文件中任一位置上的数据,还可以修改文件中的内容。

例.16 有个学生的数据,要求:

- (1) 把它们存到磁盘文件中;
- (2) 将磁盘文件中的第3,5 个学生数据读入程序,并显示出来;
- (3) 将第个学生的数据修改后存回磁盘文件中的原有位置。
- (4) 从磁盘文件读入修改后的个学生的数据并显示出来。

要实现以上要求,需要解决个问题:

- (1) 由于同一磁盘文件在程序中需要频繁地进行输入和输出,因此可将文件的工作方式指定为输入输出文件,即 ios::in|ios::out|ios::binary。
- (2) 正确计算好每次访问时指针的定位,即正确使用 seekg 或 seekp 函数。
- (3) 正确进行文件中数据的重写(更新)。

可写出以下程序:

```
#include <fstream>
using namespace std;
struct student
{int num;
 char name[20];
 float score;
};
int main( )
{student stud[5]={1001,"Li",85,1002,"Fun",97.5,1004,"Wang",54,
                  1006,"Tan",76.5,1010,"ling",96};
 fstream iofile("stud.dat",ios::in|ios::out|ios::binary);
//用 fstream 类定义输入输出二进制文件流对象 iofile
```



```

if(!iofile)
{cerr<<"open error!"<<endl;
  abort( );
}
for(int i=0;i<5;i++)//向磁盘文件输出个学生的数据
  iofile.write((char*)&stud[i],sizeof(stud[i]));
student stud1[5];           //用来存放从磁盘文件读入的数据
for(int i=0;i<5;i=i+2)
  {iofile.seekg(i*sizeof(stud[i]),ios::beg); //定位于第2,4 学生数据开头
    iofile.read((char*)&stud1[i/2],sizeof(stud1[0]));
//先后读入个学生的数据,存放在 stud1[0],stud1[1]和 stud1[2]中
    cout<<stud1[i/2].num<<" "<<stud1[i/2].name<<" "<<stud1[i/2].score<<endl;
//输出 stud1[0],stud1[1]和 stud1[2]各成员的值
  }
cout<<endl;
stud[2].num=1012;           //修改第个学生(序号为)的数据
strcpy(stud[2].name,"Wu");
stud[2].score=60;
iofile.seekp(2*sizeof(stud[0]),ios::beg); //定位于第个学生数据的开头
iofile.write((char*)&stud[2],sizeof(stud[2])); //更新第个学生数据
iofile.seekg(0,ios::beg); //重新定位于文件开头
for(int i=0;i<5;i++)
  {iofile.read((char*)&stud[i],sizeof(stud[i])); //读入个学生的数据
    cout<<stud[i].num<<" "<<stud[i].name<<" "<<stud[i].score<<endl;
  }
iofile.close( );
return 0;
}

```

运行情况如下:

```

1001 Li 85(第个学生数据)
1004 Wang 54      (第个学生数据)
1010 ling 96      (第个学生数据)

1001 Li 85        (输出修改后个学生数据)
1002 Fun 97.5
1012 Wu 60        (已修改的第个学生数据)
1006 Tan 76.5
1010 ling 96

```

本程序将磁盘文件 stud.dat 指定为输入输出型的二进制文件。这样,不仅可以向文件添加新的数据或读入数据,还可以修改(更新)数据。利用这些功能,可以实现比较复杂的输入输出任务。

请注意,不能用 ifstream 或 ofstream 类定义输入输出的二进制文件流对象,而应当用 fstream 类。

文件流是以外存文件为输入输出对象的数据流,字符串流不是以外存文件为输入输出的对象,而以内存中用户定义的字符数组(字符串)为输入输出的对象,即将数据输出到内存中的字符数组,或者从字符数组(字符串)将数据读入。字符串流也称为内存流。

字符串流也有相应的缓冲区,开始时流缓冲区是空的。如果向字符数组存入数据,随着向流插入数据,流缓冲

区中的数据不断增加,待缓冲区满了(或遇换行符),一起存入字符数组。如果是从字符数组读数据,先将字符数组中的数据送到流缓冲区,然后从缓冲区中提取数据赋给有关变量。

### 13.5 字符串流

在字符数组中可以存放字符,也可以存放整数、浮点数以及其他类型的数据。在向字符数组存入数据之前,要先将数据从二进制形式转换为 ASCII 代码,然后存放在缓冲区,再从缓冲区送到字符数组。从字符数组读数据时,先将字符数组中的数据送到缓冲区,在赋给变量前要先将 ASCII 代码转换为二进制形式。总之,流缓冲区中的数据格式与字符数组相同。

文件流类有 `ifstream`, `ofstream` 和 `fstream`,而字符串流类有 `istream`, `ostream` 和 `stringstream`。文件流类和字符串流类都是 `ostream`, `istream` 和 `iostream` 类的派生类,因此对它们的操作方法是基本相同的。向内存中的一个字符数组写数据就如同向文件写数据一样,但有点不同:

(1) 输出时数据不是流向外存文件,而是流向内存中的一个存储空间。输入时从内存中的存储空间读取数据。

(2) 字符串流对象关联的不是文件,而是内存中的一个字符数组,因此不需要打开和关闭文件。

(3) 每个文件的最后都有一个文件结束符,表示文件的结束。而字符串流所关联的字符数组中没有相应的结束标志,用户要指定一个特殊字符作为结束符,在向字符数组写入全部数据后要写入此字符。

字符串流类没有 `open` 成员函数,因此要在建立字符串流对象时通过给定参数来确立字符串流与字符数组的关联。即通过调用构造函数来解决此问题。建立字符串流对象的方法与含义如下:

#### 1. 建立输出字符串流对象

`ostream` 类提供的构造函数的原型为

```
ostream::ostream(char *buffer, int n, int mode=ios::out);
```

`buffer` 是指向字符数组首元素的指针, `n` 为指定的流缓冲区的大小(一般选与字符数组的大小相同,也可以不同),第个参数是可选的,默认为 `ios::out` 方式。可以用以下语句建立输出字符串流对象并与字符数组建立关联:

```
ostream strout(ch1,20);
```

作用是建立输出字符串流对象 `strout`,并使 `strout` 与字符数组 `ch1` 关联(通过字符串流将数据输出到字符数组 `ch1`),流缓冲区大小为。

#### 2. 建立输入字符串流对象

`istream` 类提供了两个带参的构造函数,原型为

```
istream::istream(char *buffer);
```

```
istream::istream(char *buffer, int n);
```

`buffer` 是指向字符数组首元素的指针,用它来初始化流对象(使流对象与字符数组建立关联)。可以用以下语句建立输入字符串流对象:

```
istream strin(ch2);
```

作用是建立输入字符串流对象 `strin`,将字符数组 `ch2` 中的全部数据作为输入字符串流的内容。

```
istream strin(ch2,20);
```

流缓冲区大小为,因此只将字符数组 `ch2` 中的前个字符作为输入字符串流的内容。

#### 3. 建立输入输出字符串流对象

`stringstream` 类提供的构造函数的原型为

```
stringstream::stringstream(char *buffer, int n, int mode);
```

可以用以下语句建立输入输出字符串流对象:

```
stringstream strio(ch3, sizeof(ch3), ios::in|ios::out);
```

作用是建立输入输出字符串流对象,以字符数组 `ch3` 为输入输出对象,流缓冲区大小与数组 `ch3` 相同。

以上个字符串流类是在头文件 `stringstream` 中定义的,因此程序中在用到 `istream`, `ostream` 和 `stringstream` 类时应包含头文件 `stringstream`(在 GCC 中,用头文件 `stringstream`)。

例.17 将一组数据保存在字符数组中。

```
#include <stringstream>
```

```
using namespace std;
```

```
struct student
```

```

{int num;
 char name[20];
 float score;
}
int main( )
{student stud[3]={1001,"Li",78,1002,"Wang",89.5,1004,"Fun",90};
 char c[50];//用户定义的字符数组
 ostream strout(c,30); //建立输出字符串流,与数组 c 建立关联,缓冲区长
 for(int i=0;i<3;i++) //向字符数组 c 写个学生的数据
   strout<<stud[i].num<<stud[i].name<<stud[i].score;
 strout<<ends; //ends 是 C++的 I/O 操作符,插入一个'\0'
 cout<<"array c:"<<c<<endl; //显示字符数组 c 中的字符
}

```

运行时在显示器上的输出如下:

array c:

1001Li781002Wang89.51004Fun90

以上就是字符数组 c 中的字符。可以看到:

- (1) 字符数组 c 中的数据全部是以 ASCII 代码形式存放的字符,而不是以二进制形式表示的数据。
- (2) 一般都把流缓冲区的大小指定与字符数组的大小相同。
- (3) 字符数组 c 中的数据之间没有空格,连成一片,这是由输出的方式决定的。如果以后想将这些数据读回赋给程序中相应的变量,就会出现问题,因为无法分隔两个相邻的数据。为解决此问题,可在输出时人为地加入空格。如

```

for(int i=0;i<3;i++)
  strout<<" "<<stud[i].num<<" "<<stud[i].name<<" "<<stud[i].score;

```

同时应修改流缓冲区的大小,以便能容纳全部内容,今改为字节。这样,运行时将输出

1001 Li 78 1002 Wang 89.5 1004 Fun 90

再读入时就能清楚地将数据分隔开。

例.18 在一个字符数组 c 中存放了个整数,以空格相间隔,要求将它们放到整型数组中,再按大小排序,然后再存放回字符数组 c 中。

```

#include <sstream>
using namespace std;
int main( )
{char c[50]="12 34 65 -23 -32 33 61 99 321 32";
 int a[10],i,j,t;
 cout<<"array c:"<<c<<endl;//显示字符数组中的字符串
 istream strin(c,sizeof(c)); //建立输入串流对象 strin 并与字符数组 c 关联
 for(i=0;i<10;i++)
   strin>>a[i]; //从字符数组 c 读入个整数赋给整型数组 a
 cout<<"array a:";
 for(i=0;i<10;i++)
   cout<<a[i]<<" "; //显示整型数组 a 各元素
 cout<<endl;
 for(i=0;i<9;i++) //用起泡法对数组 a 排序
 for(j=0;j<9-i;j++)
   if(a[j]>a[j+1])
 {t=a[j];a[j]=a[j+1];a[j+1]=t;}
 ostream strout(c,sizeof(c)); //建立输出串流对象 strout 并与字符数组 c 关联

```

```

for(i=0;i<10;i++)
    strout<<a[i]<<" ";           //将个整数存放在字符数组 c
strout<<ends;                    //加入'\0'
cout<<"array c:"<<c<<endl;      //显示字符数组 c
return 0;
}

```

运行结果如下:

array c: 12 34 65 -23 -32 33 61 99 321 32(字符数组 c 原来的内容)

array a: 12 34 65 -23 -32 33 61 99 321 32 (整型数组 a 的内容)

array c: -32 -12 32 33 34 61 65 99 321 (字符数组 c 最后的内容)

可以看到:

- (1) 用字符串流时不需要打开和关闭文件。
- (2) 通过字符串流从字符数组读数据就如同从键盘读数据一样,可以从字符数组读入字符数据,也可以读入整数、浮点数或其他类型数据。
- (3) 程序中先后建立了两个字符串流 `strin` 和 `strout`,与字符数组 `c` 关联。`strin` 从字符数组 `c` 中获取数据,`strout` 将数据传送给字符数组。分别对同一字符数组进行操作。甚至可以对字符数组交叉进行读写。
- (4) 用输出字符串流向字符数组 `c` 写数据时,是从数组的首地址开始的,因此更新了数组的内容。
- (5) 字符串流关联的字符数组并不一定是专为字符串流而定义的数组,它与一般的字符数组无异,可以对该数组进行其他各种操作。

与字符串流关联的字符数组相当于内存中的临时仓库,可以用来存放各种类型的数据(以 ASCII 形式存放),在需要时再从中读回来。它的用法相当于标准设备(显示器与键盘),但标准设备不能保存数据,而字符数组中的内容可以随时用 ASCII 字符输出。它比外存文件使用方便,不必建立文件(不需打开与关闭),存取速度快。但它的生命周期与其所在的模块(如主函数)相同,该模块的生命周期结束后,字符数组也不存在了。因此只能作为临时存储空间。

## 第 14 章 C++工具

### 14.1 异常处理

### 14.2 命名空间

### 14.3 使用早期的函数库

在 C++发展的后期,有时 C++编译系统根据实际工作的需要,增加了一些功能,作为工具来使用,其中主要有模板(包括函数模板和类模板)、异常处理、命名空间和运行时类型识别,以帮助程序设计人员更方便地进行程序的设计和调试工作。年 ANSI C++委员会将它们纳入了 ANSI C++标准,建议所有的 C++编译系

统都能实现这些功能。这些工具是非常有用的，C++的使用者应当尽量使用这些工具。

## 14.1 异常处理

### 14.1.1 异常处理的任务

程序编制者不仅要考虑程序没有错误的理想情况，更要考虑程序存在错误时的情况，应该能够尽快地发现错误，消除错误。

程序中常见的错误有两大类：语法错误和运行错误。在编译时，编译系统能发现程序中的语法错误。

有的程序虽然能通过编译，也能投入运行。但是在运行过程中会出现异常，得不到正确的运行结果，甚至导致程序不正常终止，或出现死机现象。这类错误比较隐蔽，不易被发现，往往耗费许多时间和精力。这成为程序调试中的一个难点。

在设计程序时，应当事先分析程序运行时可能出现的各种意外的情况，并且分别制订出相应的处理方法，这就是程序的异常处理的任务。

在运行没有异常处理的程序时，如果运行情况出现异常，由于程序本身不能处理，程序只能终止运行。如果在程序中设置了异常处理机制，则在运行情况出现异常时，由于程序本身已规定了处理的方法，于是程序的流程就转到异常处理代码段处理。用户可以指定进行任何的处理。

需要说明，只要出现与人们期望的情况不同，都可以认为是异常，并对它进行异常处理。因此，所谓异常处理指的是对运行时出现的差错以及其他例外情况的处理。

### 14.1.2 异常处理的方法

在一个小的程序中，可以用比较简单的方法处理异常。但是在一个大的系统中，如果在每一个函数中都设置处理异常的代码段，会使程序过于复杂和庞大。因此，C++采取的办法是：如果在执行一个函数过程中出现异常，可以不在本函数中立即处理，而是发出一个信息，传给它的上一级(即调用它的函数)，它的上级捕捉到这个信息后进行处理。如果上一级的函数也不能处理，就再传给其上一级，由其上一级处理。如此逐级上送，如果到最高一级还无法处理，最后只好异常终止程序的执行。

这样做使异常的发现与处理不由同一函数来完成。好处是使底层的函数专门用于解决实际任务，而不必再承担处理异常的任务，以减轻底层函数的负担，而把处理异常的任务上移到某一层去处理。这样可以提高效率。

C++处理异常的机制是由个部分组成的，即检查(**try**)、抛出(**throw**)和捕捉(**catch**)。把需要检查的语句放在 **try** 块中，**throw** 用来当出现异常时发出一个异常信息，而 **catch** 则用来捕捉异常信息，如果捕捉到了异常信息，就处理它。

例.1 给出三角形的三边  $a, b, c$ ，求三角形的面积。只有  $a+b>c, b+c>a, c+a>b$  时才能构成三角形。设置异常处理，对不符合三角形条件的输出警告信息，不予计算。

先写出没有异常处理时的程序：

```
#include <iostream>
#include <cmath>
using namespace std;
int main( )
{double triangle(double,double,double);
 double a,b,c;
 cin>>a>>b>>c;
 while(a>0 && b>0 && c>0)
 {cout<<triangle(a,b,c)<<endl;
  cin>>a>>b>>c;
 }
 return 0;
```

```
}
```

```
double triangle(double a,double b,double c)
{double area;
 double s=(a+b+c)/2;
 area=sqrt(s*(s-a)*(s-b)*(s-c));
 return area;
}
```

运行情况如下:

```
6 5 4✓ (输入 a,b,c 的值)
9.92157      (输出三角形的面积)
1 1.5 2✓    (输入 a,b,c 的值)
0.726184    (输出三角形的面积)
1 2 1✓      (输入 a,b,c 的值)
0           (输出三角形的面积, 此结果显然不对,因为不是三角形)
1 0 6✓      (输入 a,b,c 的值)
(结束)
```

修改程序, 在函数 `triangle` 中对三角形条件进行检查, 如果不符合三角形条件, 就抛出一个异常信息, 在主函数中的 `try-catch` 块中调用 `triangle` 函数, 检测有无异常信息, 并作相应处理。修改后的程序如下:

```
#include <iostream>
#include <cmath>
using namespace std;
void main()
{double triangle(double,double,double);
 double a,b,c;
 cin>>a>>b>>c;
 try//在 try 块中包含要检查的函数
 {while(a>0 && b>0 && c>0)
 {cout<<triangle(a,b,c)<<endl;
 cin>>a>>b>>c;
 }
 }

 catch(double) //用 catch 捕捉异常信息并作相应处理
 {cout<<"a="<<a<<","b="<<b<<","c="<<c<<","that is not a triangle!"<<endl;}
 cout<<"end"<<endl;
 }

 double triangle(double a,double b,double c) //计算三角形的面积的函数
 {double s=(a+b+c)/2;
 if (a+b<=c||b+c<=a||c+a<=b) throw a; //当不符合三角形条件抛出异常信息
 return sqrt(s*(s-a)*(s-b)*(s-c));
 }
```

程序运行结果如下:

```
6 5 4✓ (输入 a,b,c 的值)
9.92157      (计算出三角形的面积)
```



```

1 1.5 2✓      (输入 a,b,c 的值)
0.726184      (计算出三角形的面积)
1 2 1✓      (输入 a,b,c 的值)
a=1,b=2,c=1, that is not a triangle! (异常处理)
end

```

现在结合程序分析怎样进行异常处理。

- (1) 首先把可能出现异常的、需要检查的语句或程序段放在 `try` 后面的花括号中。
- (2) 程序开始运行后，按正常的顺序执行到 `try` 块，开始执行 `try` 块中花括号内的语句。如果在执行 `try` 块内的语句过程中没有发生异常，则 `catch` 子句不起作用，流程转到 `catch` 子句后面的语句继续执行。
- (3) 如果在执行 `try` 块内的语句(包括其所调用的函数)过程中发生异常，则 `throw` 运算符抛出一个异常信息。`throw` 抛出异常信息后，流程立即离开本函数，转到其上一级的函数(main 函数)。`throw` 抛出什么样的数据由程序设计者自定，可以是任何类型的数据。

- (4) 这个异常信息提供给 `try-catch` 结构，系统会寻找与之匹配的 `catch` 子句。
  - (5) 在进行异常处理后，程序并不会自动终止，继续执行 `catch` 子句后面的语句。
- 由于 `catch` 子句是用来处理异常信息的，往往被称为 `catch` 异常处理块或 `catch` 异常处理器。

下面讲述异常处理的语法。

`throw` 语句一般是由 `throw` 运算符和一个数据组成的，其形式为

`throw` 表达式;

`try-catch` 的结构为

```

try
    {被检查的语句}
catch(异常信息类型[变量名])
    {进行异常处理的语句}

```

说明:

- (1) 被检测的函数必须放在 `try` 块中，否则不起作用。
- (2) `try` 块和 `catch` 块作为一个整体出现，`catch` 块是 `try-catch` 结构中的一部分，必须紧跟在 `try` 块之后，不能单独使用，在二者之间也不能插入其他语句。但是在一个 `try-catch` 结构中，可以只有 `try` 块而无 `catch` 块。即在本函数中只检查而不处理，把 `catch` 处理块放在其他函数中。
- (3) `try` 和 `catch` 块中必须有用花括号括起来的复合语句，即使花括号内只有一个语句，也不能省略花括号。
- (4) 一个 `try-catch` 结构中只能有一个 `try` 块，但却可以有多个 `catch` 块，以便与不同的异常信息匹配。

- (5) `catch` 后面的圆括号中，一般只写异常信息的类型名，如

`catch(double)`

`catch` 只检查所捕获异常信息的类型，而不检查它们的值。因此如果需要检测多个不同的异常信息，应当由 `throw` 抛出不同类型的异常信息。

异常信息可以是 C++ 系统预定义的标准类型，也可以是用户自定义的类型(如结构体或类)。如果由 `throw` 抛出的信息属于该类型或其子类型，则 `catch` 与 `throw` 二者匹配，`catch` 捕获该异常信息。

`catch` 还可以有另外一种写法，即除了指定类型名外，还指定变量名，如

`catch(double d)`

此时如果 `throw` 抛出的异常信息是 `double` 型的变量 `a`，则 `catch` 在捕获异常信息 `a` 的同时，还使 `d` 获得 `a` 的值，或者说 `d` 得到 `a` 的一个拷贝。什么时候需要这样做呢？有时希望在捕获异常信息时，还能利用 `throw` 抛出的值，如

```
catch(double d)
```

```
{cout<<"throw "<<d;}
```

这时会输出 d 的值(也就是 a 值)。当抛出的是类对象时,有时希望在 catch 块中显示该对象中的某些信息。这时就需要在 catch 的参数中写出变量名(类对象名)。

(6) 如果在 catch 子句中没有指定异常信息的类型,而用了删节号“...”,则表示它可以捕捉任何类型的异常信息,如

```
catch(...) {cout<<"OK"<<endl;}
```

它能捕捉所有类型的异常信息,并输出“OK”。

这种 catch 子句应放在 try catch 结构中的最后,相当于“其他”。如果把它作为第一个 catch 子句,则后面的 catch 子句都不起作用。

(7) try catch 结构可以与 throw 出现在同一个函数中,也可以不在同一函数中。当 throw 抛出异常信息后,首先在本函数中寻找与之匹配的 catch,如果在本函数中无 try catch 结构或找不到与之匹配的 catch,就转到离开出现异常最近的 try catch 结构去处理。

(8) 在某些情况下,在 throw 语句中可以不包括表达式,如

```
throw;
```

表示“我不处理这个异常,请上级处理”。

(9) 如果 throw 抛出的异常信息找不到与之匹配的 catch 块,那么系统就会调用一个系统函数 terminate,使程序终止运行。

例.2 在函数嵌套的情况下检测异常处理。

这是一个简单的例子,用来说明在 try 块中有函数嵌套调用的情况下抛出异常和捕捉异常的情况。请自己先分析以下程序。

```
#include <iostream>
```

```
using namespace std;
```

```
int main( )
```

```
{void f1( );
```

```
try
```

```
{f1();} //调用 f1( )
```

```
catch(double)
```

```
{cout<<"OK0!"<<endl;}
```

```
cout<<"end0"<<endl;
```

```
return 0;
```

```
}
```

```
void f1( )
```

```
{void f2( );
```

```
try
```

```
{f2();} //调用 f2( )
```

```
catch(char)
```

```
{cout<<"OK1!"<<endl;}
```

```
cout<<"end1"<<endl;
```

```
}
```

```
void f2( )
```

```
{void f3( );
```



```

try
{f3();}                //调用 f3()
catch(int)
{cout<<"Ok2!"<<endl;}
cout<<"end2"<<endl;
}
void f3()
{double a=0;
try
{throw a;}            //抛出 double 类型异常信息
catch(float)
{cout<<"OK3!"<<endl;}
cout<<"end3"<<endl;
}

```

分种情况分析运行情况:

( ) 执行上面的程序。图.1 为有函数嵌套时异常处理示意图。

图.1

程序运行结果如下:

OK0!(在主函数中捕获异常)

end0 (执行主函数中最后一个语句时的输出)

(2) 如果将 f3 函数中的 catch 子句改为 catch(double), 而程序中其他部分不变, 则程序运行结果如下:

OK3!(在 f3 函数中捕获异常)

end3 (执行 f3 函数中最后一个语句时的输出)

end2 (执行 f2 函数中最后一个语句时的输出)

end1 (执行 f1 函数中最后一个语句时的输出)

end0 (执行主函数中最后一个语句时的输出)

(3) 如果在此基础上再将 f3 函数中的 catch 块改为

```

catch(double)
{cout<<"OK3!"<<endl;throw;}

```

程序运行结果如下:

OK3!(在 f3 函数中捕获异常)

OK0! (在主函数中捕获异常)

end0 (执行主函数中最后一个语句时的输出)

#### 14.1.3 在函数声明中进行异常情况指定

为便于阅读程序, 使用户在看程序时能够知道所用的函数是否会抛出异常信息以及异常信息可能的类型, C++允许在声明函数时列出可能抛出的异常类型, 如可以将例.1 中第二个程序的第行改写为

```
double triangle(double,double,double) throw(double);
```

表示 triangle 函数只能抛出 double 类型的异常信息。如果写成

```
double triangle(double,double,double) throw(int,double,float,char);
```

则表示 triangle 函数可以抛出 int,double,float 或 char 类型的异常信息。异常指定是函数声明的一部分, 必须

同时出现在函数声明和函数定义的首行中，否则在进行函数的另一次声明时，编译系统会报告“类型不匹配”。

如果在声明函数时未列出可能抛出的异常类型，则该函数可以抛出任何类型的异常信息。如例.1 中第个程序中所表示的那样。

如果想声明一个不能抛出异常的函数，可以写成以下形式：

```
double triangle(double,double,double) throw();//throw 无参数
```

这时即使在函数执行过程中出现了 throw 语句，实际上也并不执行 throw 语句，并不抛出任何异常信息，程序将非正常终止。

#### 14.1.4 在异常处理中处理析构函数

如果在 try 块(或 try 块中调用的函数)中定义了类对象，在建立该对象时要调用构造函数。在执行 try 块(包括在 try 块中调用其他函数)的过程中如果发生了异常，此时流程立即离开 try 块。这样流程就有可能离开该对象的作用域而转到其他函数，因而应当事先做好结束对象前的清理工作，C++的异常处理机制会在 throw 抛出异常信息被 catch 捕获时，对有关的局部对象进行析构(调用类对象的析构函数)，析构对象的顺序与构造的顺序相反，然后执行与异常信息匹配的 catch 块中的语句。

例.3 在异常处理中处理析构函数。

这是一个为说明在异常处理中调用析构函数的示例，为了清晰地表示流程，程序中加入了一些 cout 语句，输出有关的信息，以便对照结果分析程序。

```
#include <iostream>
#include <string>
using namespace std;
class Student
{public:
    Student(int n,string nam)//定义构造函数
    {cout<<"constructor-"<<n<<endl;
    num=n;name=nam;}
    ~Student() {cout<<"destructor-"<<num<<endl;} //定义析构函数
    void get_data(); //成员函数声明
private:
    int num;
    string name;
};

void Student::get_data() //定义成员函数
{if(num==0) throw num; //如 num=0,抛出 int 型变量 num
 else cout<<num<<" "<<name<<endl; //若 num≠, 输出 num,name
 cout<<"in get_data()"<<endl; //输出信息，表示目前在 get_data 函数中
}

void fun()
{Student stud1(1101,"Tan"); //建立对象 stud1
stud1.get_data(); //调用 stud1 的 get_data 函数
Student stud2(0,"Li"); //建立对象 stud2
stud2.get_data(); //调用 stud2 的 get_data 函数
}
```

```

int main()
{cout<<"main begin"<<endl;           //表示主函数开始了
cout<<"call fun()"<<endl;           //表示调用 fun 函数
try
    {fun();}                          //调用 fun 函数

catch(int n)
    {cout<<"num="<<n<<"error!"<<endl;} //表示 num=0 出错
cout<<"main end"<<endl;           //表示主函数结束
return 0;
}

```

程序运行结果如下:

```

main begin
call fun()
constructor-1101
1101 tan
in get_data()
constructor-0
destructor-0
destructor-1101
num=0,error!
main end

```

## 14.2 命名空间

在学习本书前面各章时，已经多次看到在程序中用了以下语句:

```
using namespace std;
```

这就是使用了命名空间 `std`。在本节中将对它作较详细的介绍。

### 14.2.1 为什么需要命名空间

命名空间是 ANSI C++引入的可以由用户命名的作用域，用来处理程序中常见的同名冲突。

在 C 语言中定义了几个层次的作用域，即文件(编译单元)、函数和复合语句。C++又引入了类作用域，类是出现在文件内的。在不同的作用域中可以定义相同名字的变量，互不干扰，系统能够区别它们。

下面先简单分析一下作用域的作用，然后讨论命名空间的作用。

如果在文件中定义了两个类，在这两个类中可以有同名的函数。在引用时，为了区别，应该加上类名作为限定，如

```

class A//声明 A 类
{public:
void fun1();           //声明 A 类中的 fun1 函数
private:
int i;
};
void A::fun1()          //定义 A 类中的 fun1 函数
{
//
}
class B                 //声明 B 类
{public:
void fun1();           //B 类中也有 fun1 函数
}

```

```

void fun2( );
};
void B::fun1( )           //定义 B 类中的 fun1 函数
{
    //
}

```

这样不会发生混淆。

在文件中可以定义全局变量(global variable)，它的作用域是整个程序。如果在文件 A 中定义了一个变量 a

```
int a=3;
```

在文件 B 中可以再定义一个变量 a

```
int a=5;
```

在分别对文件 A 和文件 B 进行编译时不会有问题。但是，如果一个程序包括文件 A 和文件 B，那么在进行连接时，会报告出错，因为在同一个程序中有两个同名的变量，认为是对变量的重复定义。问题在于全局变量的作用域是整个程序，在同一作用域中不应有两个或多个同名的实体(entity)，包括变量、函数和类等。

可以通过 extern 声明同一程序中的两个文件中的同名变量是同一个变量。如果在文件 B 中有以下声明：

```
extern int a;
```

表示文件 B 中的变量 a 是在其他文件中已定义的变量。由于有此声明，在程序编译和连接后，文件 A 的变量 a 的作用域扩展到了文件 B。如果在文件 B 中不再对 a 赋值，则在文件 B 中用以下语句输出的是文件 A 中变量 a 的值：

```
cout<<a;//得到 a 的值为
```

在简单的程序设计中，只要人们小心注意，可以争取不发生错误。但是，一个大型的应用软件，往往不是由一个人独立完成的，假如不同的人分别定义了类，放在不同的头文件中，在主文件(包含主函数的文件)需要用这些类时，

就用#include 命令行将这些头文件包含进来。由于各头文件是由不同的人设计的，有可能在不同的头文件中用了相同的名字来命名所定义的类或函数。这样在程序中就会出现名字冲突。

例.4 名字冲突。

程序员甲在头文件 header1.h 中定义了类 Student 和函数 fun。

```
//header1.h (头文件，设其文件名为 cc14-4-h1.h)
```

```

#include <string>
#include <cmath>
using namespace std;
class Student//声明 Student 类
{public:
    Student(int n,string nam,char s)
{num=n;name=nam;sex=s;}
    void get_data( );
private:

int num;
    string name;
    char sex;
};
void Student::get_data( )           //成员函数定义

```

```

{cout<<num<<" "<<name<<" "<<sex<<endl;
}
double fun(double a,double b)           //定义全局函数(即外部函数)
{return sqrt(a+b);}
在 main 函数所在的文件中包含头文件 header1.h:
#include <iostream>
#include "cc14-4-h1.h"    //注意要用双引号, 因为文件一般是放在用户目录中的
using namespace std;
int main()
{Student stud1(101,"Wang",18); //定义类对象 stud1
  stud1.get_data();
  cout<<fun(5,3)<<endl;
  return 0;
}

```

程序能正常运行, 输出为

```

101 Wang 18
2.82843

```

如果程序员乙写了头文件 header2.h, 在其中除了定义其他类以外, 还定义了类 Student 和函数 fun, 但其内容与头文件 header1.h 中的 Student 和函数 fun 有所不同。

```

//header2.h (头文件, 设其文件名为 cc14-4-h2.h)
#include <string>
#include <cmath>
using namespace std;
class Student//声明 Student 类
{public:
  Student(int n,string nam,char s)    //参数与 header1 中的 student 不同
  {num=n;name=nam;sex=s;}
  void get_data();

private:
  int num;
  string name;
  char sex;                          //此项与 header1 不同
};

```

```

void Student::get_data()              //成员函数定义
{cout<<num<<" "<<name<<" "<<sex<<endl;
}
double fun(double a,double b)         //定义全局函数
{return sqrt(a-b);}                   //返回值与 header1 中的 fun 函数不同

```

//头文件中可能还有其他内容

假如主程序员在其程序中要用到 header1.h 中的 Student 和函数 fun, 因而在程序中包含了头文件 header1.h, 同时要用到头文件 header2.h 中的一些内容, 因而在程序中又包含了头文件 header2.h。如果主文件(包含主函数的文件)如下:

```

//main file

```

```

#include <iostream>
#include "cc14-4-h1.h"//包含头文件
#include "cc14-4-h2.h"    //包含头文件
using namespace std;
int main( )
{Student stud1(101,"Wang",18);
stud1.get_data();
cout<<fun(5,3)<<endl;
return 0;
}

```

这时程序编译就会出错。因为在预编译后，头文件中的内容取代了对应的#include 命令行，这样就在同一个程序文件中出现了两个 Student 类和两个 fun 函数，显然是重复定义，这就是名字冲突，即在同一个作用域中有两个或多个同名的实体。

不仅如此，在程序中还往往需要引用一些库，为此需要包含有关的头文件。如果在这些库中包含有与程序的全局实体同名的实体，或者不同的库中有相同的实体名，则在编译时就会出现名字冲突。

为了避免这类问题的出现，人们提出了许多方法，例如：将实体的名字写得长一些；把名字起得特殊一些，包括一些特殊的字符；由编译系统提供的内部全局标识符都用下划线作为前缀，如\_complex()，以避免与用户命名的实体同名；由软件开发商提供的实体的名字用特定的字符作为前缀。但是这样的效果并不理想，而且增加了阅读程序的难度，可读性降低了。

C 语言和早期的 C++语言没有提供有效的机制来解决这个问题，没有使库的提供者能够建立自己的命名空间的工具。人们希望 ANSI C++标准能够解决这个问题，提供一种机制、一种工具，使由库的设计者命名的全局标识符能够和程序的全局实体名以及其他库的全局标识符区别开来。

#### 14.2.2 什么是命名空间

为了解决上面这个问题，ANSI C++增加了命名空间(namespace)。所谓命名空间，实际上就是一个由程序设计者命名的内存区域。程序设计者可以根据需要指定一些有名字的空间域，把一些全局实体分别放在各个命名空间中，从而与其他全局实体分隔开来。如

```

namespace ns1//指定命名空间 ns1
{int a;
double b;
}

```

现在命名空间成员包括变量 a 和 b，注意 a 和 b 仍然是全局变量，仅仅是把它们隐藏在指定的命名空间中而已。

如果在程序中要使用变量 a 和 b，必须加上命名空间名和作用域分辨符“::”，如 ns1::a, ns1::b。这种用法称为命名空间限定(qualified)，这些名字(如 ns1::a)称为被限定名(qualified name)。C++中命名空间的作用类似于操作系统中的目录和文件的关系。

命名空间的作用是建立一些互相分隔的作用域，把一些全局实体分隔开来，以免产生名字冲突。

可以根据需要设置许多个命名空间，每个命名空间名代表一个不同的命名空间域，不同的命名空间不能同名。这样，可以把不同的库中的实体放到不同的命名空间中。过去我们用的全局变量可以理解为全局命名空间，独立于所有有名的命名空间之外，它是不需要用 namespace 声明的，实际上是由系统隐式声明的，存在于每个程序之中。

在声明一个命名空间时，花括号内不仅可以包括变量，而且还可以包括以下类型：

- ? 变量(可以带有初始化);
- ? 常量;

- ? 函数(可以是定义或声明);
- ? 结构体;
- ? 类;
- ? 模板;
- ? 命名空间(在一个命名空间中又定义一个命名空间, 即嵌套的命名空间)。

例如

```
namespace ns1
{const int RATE=0.08;//常量
double pay;                //变量
double tax( )               //函数
{return a*RATE;}
namespace ns2               //嵌套的命名空间
{int age;}
}
```

如果想输出命名空间 ns1 中成员的数据, 可以采用下面的方法:

```
cout<<ns1::RATE<<endl;
cout<<ns1::pay<<endl;
cout<<ns1::tax()<<endl;
cout<<ns1::ns2::age<<endl;//需要指定外层的和内层的命名空间名
```

#### 14.2.3 使用命名空间解决名字冲突

现在, 对例.4 程序进行修改, 使之能正确运行。

例.5 利用命名空间来解决例.4 程序名字冲突问题。

修改两个头文件, 把在头文件中声明的类分别放在两个不同的命名空间中。

//header1.h (头文件)

```
#include <string>
#include <cmath>
using namespace std;
namespace ns1//声明命名空间 ns1
{class Student          //在命名空间 ns1 内声明 Student 类
{public:
Student(int n,string nam,int a)

{num=n;name=nam;age=a;}
void get_data( );
private:
int num;
string name;
int age;
};
void Student::get_data() //定义成员函数
{cout<<num<<" "<<name<<" "<<age<<endl;
}

double fun(double a,double b) //在命名空间 ns1 内定义 fun 函数
{return sqrt(a+b);}
}
```

```

//header2.h ((头文件)
#include <string>
#include <cmath>

using namespace std;
namespace ns2                //声明命名空间 ns2
{
    class Student
    {
    public:
        Student(int n,string nam,char s)
        {num=n;name=nam;sex=s;}
        void get_data( );
    private:
        int num;
        char name[20];
        char sex;
    };
    void Student::get_data( )
    {cout<<num<<" "<<name<<" "<<sex<<endl;
    }
    double fun(double a,double b)
    {return sqrt(a-b);}
    }

//main file (主文件)
#include <iostream>
#include "cc14-5-h1.h"        //包含头文件
#include "cc14-5-h2.h"        //包含头文件
using namespace std;
int main( )
{
    ns1::Student stud1(101,"Wang",18); //用命名空间 ns1 中声明的 Student 类定义 stud1
    stud1.get_data( );                 //不要写成 ns1::stud1.get_data( );
    cout<<ns1::fun(5,3)<<endl;         //调用命名空间 ns1 中的 fun 函数
    ns2::Student stud2(102,"Li",'f'); //用命名空间 ns2 中声明的 Student 类定义 stud2
    stud2.get_data( );
    cout<<ns2::fun(5,3)<<endl;         //调用命名空间 ns2 中的 fun 函数
    return 0;
}

```

程序能顺利通过编译，并得到以下运行结果：

```

101 Wang 18(对象 stud1 中的数据)
2.82843          (5+3 的开方值)
102 Li f         (对象 stud2 中的数据)
1.41421          (5-3 的开方值)

```

#### 14.2.4 使用命名空间成员的方法

在引用命名空间成员时，要用命名空间名和作用域分辨符对命名空间成员进行限定，以区别不同的命名空



间中的同名标识符。即

命名空间名::命名空间成员名

这种方法是有效的,能保证所引用的实体有惟一的名字。但是如果命名空间名字比较长,尤其在有命名空间嵌套的情况下,为引用一个实体,需要写很长的名字。在一个程序中可能要多次引用命名空间成员,就会感到很不方便。

为此,C++提供了一些机制,能简化使用命名空间成员的手续。

#### ( ) 使用命名空间别名

可以为命名空间起一个别名(namespace alias),用来代替较长的命名空间名。如

```
namespace Television//声明命名空间,名为 Television
```

```
{...}
```

可以用一个较短而易记的别名代替它。如

```
namespace TV = Television;//别名 TV 与原名 Television 等价
```

#### ( ) 使用 using 命名空间成员名

using 后面的命名空间成员名必须是由命名空间限定的名字。例如

```
using ns1::Student;
```

using 声明的有效范围是从 using 语句开始到 using 所在的作用域结束。如果在以上的 using 语句之后有以下语句:

```
Student stud1(101,"Wang",18);//此处的 Student 相当于 ns1::Student
```

上面的语句相当于

```
ns1::Student stud1(101,"Wang",18);
```

又如

```
using ns1::fun;//声明其后出现的 fun 是属于命名空间 ns1 中的 fun
```

```
cout<<fun(5,3)<<endl; //此处的 fun 函数相当于 ns1::fun(5,3)
```

显然,这可以避免在每一次引用命名空间成员时都用命名空间限定,使得引用命名空间成员方便易用。

但是要注意:在同一作用域中用 using 声明的不同命名空间的成员中不能有同名的成员。例如

```
using ns1::Student;//声明其后出现的 Student 是命名空间 ns1 中的 Student
```

```
using ns2::Student; //声明其后出现的 Student 是命名空间 ns2 中的 Student
```

```
Student stud1; //请问此处的 Student 是哪个命名空间中的 Student?
```

产生了二义性,编译出错。

#### ( ) 使用 using namespace 命名空间名

能否在程序中用一个语句就能一次声明一个命名空间中的全部成员呢?

C++提供了 using namespace 语句来实现这一目的。using namespace 语句的一般格式为

```
using namespace 命名空间名;
```

例如

```
using namespace ns1;
```

声明了在本作用域中要用到命名空间 ns1 中的成员,在使用该命名空间的任何成员时都不必用命名空间限定。如果在作了上面的声明后有以下语句:

```
Student stud1(101,"Wang",18);//Student 隐含指命名空间 ns1 中的 Student
```

```
cout<<fun(5,3)<<endl; //这里的 fun 函数是命名空间 ns1 中的 fun 函数
```

在用 using namespace 声明的作用域中,命名空间 ns1 的成员就好像在全局域声明的一样。因此可以不必用命名空间限定。显然这样的处理对写程序比较方便。但是如果同时用 using namespace 声明多个命名空间时,往往容易出错。因此只有在使用命名空间数量很少,以及确保这些命名空间中没有同名成员时才用 using namespace 语句。

### 14.2.5 无名的命名空间

以上介绍的是有名字的命名空间，C++还允许使用没有名字的命名空间，如在文件 A 中声明了以下的无名命名空间：

`namespace//命名空间没有名字`

```
{void fun()                      //定义命名空间成员
    {cout<<"OK."<<endl;}
}
```

由于命名空间没有名字，在其他文件中显然无法引用，它只在本文件的作用域内有效。无名命名空间的成员 `fun` 函数的作用域为文件 A。在文件 A 中使用无名命名空间的成员，不必(也无法)用命名空间名限定。如果在文件 A 中有以下语句：

```
fun();
```

则执行无名命名空间中的成员 `fun` 函数，输出“OK.”。

在本程序中的其他文件中也无法使用该 `fun` 函数，也就是把 `fun` 函数的作用域限制在本文件范围中。

### 14.2.6 标准命名空间 `std`

为了解决 C++标准库中的标识符与程序中的全局标识符之间以及不同库中的标识符之间的同名冲突，应该将不同库的标识符在不同的命名空间中定义(或声明)。标准 C++库的所有的标识符都是在一个名为 `std` 的命名空间中定义的，或者说标准头文件(如 `iostream`)中函数、类、对象和类模板是在命名空间 `std` 中定义的。这样，在程序中用到 C++标准库时，需要使用 `std` 作为限定。如

```
std::cout<<"OK."<<endl; //声明 cout 是在命名空间 std 中定义的流对象
```

在大多数的 C++程序中常用 `using namespace` 语句对命名空间 `std` 进行声明，这样可以不必对每个命名空间成员一一进行处理，在文件的开头加入以下 `using namespace` 声明：

```
using namespace std;
```

这样，在 `std` 中定义和声明的所有标识符在本文件中都可以作为全局量来使用。但是应当绝对保证在程序中不出现与命名空间 `std` 的成员同名的标识符。由于在命名空间 `std` 中定义的实体实在太多，有时程序设计人员也弄不清哪些标识符已在命名空间 `std` 中定义过，为减少出错机会，有的专业人员喜欢用若干个“`using 命名空间成员`”声明来代替“`using namespace 命名空间`”声明，如

```
using std::string;
```

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

```
using std::cin;
```

等。为了减少在每一个程序中都要重复书写以上的 `using` 声明，程序开发者往往把编写应用程序时经常会用到的命名空间 `std` 成员的 `using` 声明组成一个头文件，然后在程序中包含此头文件即可。

### 14.3 使用早期的函数库

C 语言程序中各种功能基本上都是由函数来实现的，在 C 语言的发展过程中建立了功能丰富的函数库，C++从 C 语言继承了这份宝贵的财富。在 C++程序中可以使用 C 语言的函数库。

如果要用函数库中的函数，就必须在程序文件中包含有关的头文件，在不同的头文件中，包含了不同的函数的声明。

在 C++中使用这些头文件有两种方法。

(1) 用 C 语言的传统方法。头文件名包括后缀 `.h`，如 `stdio.h`，`math.h` 等。由于 C 语言没有命名空间，头文件并不存放在命名空间中，因此在 C++程序文件中如果用到带后缀 `.h` 的头文件时，不必用命名空间。只需在文件中包含所用的头文件即可。如

```
#include <math.h>
```

(2) 用 C++的新方法。C++标准要求系统提供的头文件不包括后缀 `.h`，例如 `iostream`、`string`。为了表示与 C 语言的头文件有联系又有区别，C++所用的头文件名是在 C 语言的相应的头文件名(但不包括后缀 `.h`)之前加一字母 `c`。

此外，由于这些函数都是在命名空间 `std` 中声明的，因此在程序中要对命名空间 `std` 作声明。如

```
#include <cstdio>
#include <cmath>
using namespace std;
```

目前所用的大多数 C++ 编译系统既保留了 C 的用法，又提供了 C++ 的新方法。下面两种用法等价，可以任选。

C 传统方法

```
#include <stdio.h>
#include <math.h>
#include <string.h>
```

C++ 新方法

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

```
using namespace std;
```

可以使用传统的 C 方法，但应当提倡使用 C++ 的新方法。