第3章 C++在非面向对象方面的常用新特性

本章学习目标：

- C++的输入与输出；
- 用 const 定义常量；
- 函数的重载；
- 带有默认参数的函数；
- 引用；
- 动态分配内存；
- 布尔类型；
- 函数原型；
- 作用域运算符。

C++是 AT&T 公司的贝尔实验室的 Bjarne Stroustrup 博士开发的一种编程语言，它是在 C 语言的基础上增加了面向对象功能和其他一些增强功能，起初被称为“带类的 C”(C with Class)，1983 年正式取名 C++。为了表明它是 C 的增强版，所以在名字中使用了 C 语言中的自增运算符＋＋，从而形成了 C++。

C++是从 C 发展而来的，摒弃了 C 的局限性，增加了一些新的特点，如表 3.1 所示。

表 3.1　C 语言的局限与 C++语言的特点

C 语言的局限	C++语言的特点
类型检测机制比较弱，很多错误在编译阶段不能发现	兼容 C 语言
没有提供代码重用的结构	结构更清晰，可读性强
不适合大型程序的开发	代码质量高，速度快
	可维护性好，重用性高

总之，C++对 C 引入了面向对象的新概念，同时增加了一些非面向对象的新特性，这些特性使得 C++使用起来更加方便与安全，本节将讨论一些常用的新特性。

3.1 C++的输入与输出简介

为了方便用户，C++增加了标准输入输出流对象 cout 和 cin。所谓“流”是指来自设备或输出到设备的一系列字节，这些字节按照进入“流”的顺序排列。cout 代表标准输出流对象，cin 代表标准输入流对象。cout 和 cin 都是在头文件 iostream 中定义的。cin 的输入设备是键盘，cout 的输出设备是屏幕。

3.1.1 用 cout 输出数据流

cout 是标准输出流对象，指的是标准输出设备，通常表示屏幕。运算符“<<”在 C 语言中表示位“左移”操作。在 C++语言中，“<<”除了表示“左移”操作外，还可以用于输出，即将“<<”右边的数据写到标准输出流对象 cout 中，在屏幕上进行显示。cout 通常与“<<”结合使用，“<<”在此处起到插入的作用。cout 的一般形式如下：

```
cout <<表达式 1 <<…<<表达式 n;
```

例如：

```
cout <<"Hello, World!\n";     //用 C++的方法输出一行
```

其作用是将字符串“Hello, World! \n”插入到输出流中，也就是输出在标准输出设备上。在头文件 iostream 中定义了控制符 endl，endl 代表回车换行操作，作用与“\n”相同。

注意：

(1) 允许通过“<<”输出多个数据，例如：

```
cout << x <<' '<< y <<' '<< i <<' '<< j <<' '<<"hello "<<"大家好"<< endl;
```

(2) 允许输出表达式的值，例如：

```
cout << x + y <<' '<< i * j << endl;
```

3.1.2 用 cin 输入数据流

cin 是标准输入流对象，指的是标准输入设备，通常表示键盘。运算符“>>”在 C 语言中表示位“右移”操作。在 C++语言中，“>>”除了表示“右移”操作外，还可以用于输入，即将从标准输入流对象(键盘)中读取的值传送给右边的变量。cin 通常与“>>”结合使用。

cin 是从键盘向内存流动的数据流。用“>>”运算符从输入设备“键盘”取得数据送到标准输入流 cin 中，然后再送到内存。“>>”常称为输入运算符。

cin 应与“>>”配合使用。cin 的一般形式如下：

```
cin >>变量 1 >>…>>变量 n;
```

例如：

```
int m;                  //定义整型变量 m
float x;                //定义浮点型变量 x
cin >> m >> x;          // 输入一个整数和一个实数
```

可以从键盘输入：

```
16 168.98
```

m 和 x 分别获得值 16 和 168.98。

注意：

(1) 运算符“>>”允许用户从键盘上输入多个数据，例如：

```
cin >> x >> y >> i >> j;
```

它按照顺序从键盘输入中提取多个数据，并依次存入相对应的变量中。在进行多个数据的输入时，两个数据之间要使用空格或者用回车、Tab 键进行分隔。

(2) 可以对输入输出数据的格式进行控制。例如可以通过操作符 dec、hex 和 oct 以不同进制的形式显示数据。其中，dec 是将基数设为十进制，hex 是将基数设为十六进制，oct 是将基数设为八进制，默认采用的基数是十进制。

```
int i = 100;
cout << hex << i << ' ' << dec << i << ' ' << oct << i << endl;
```

输出的结果如下：

```
64 100 144
```

(3) 如果在程序中使用 cin 和 cout，必须将头文件包含到本文件中：

```
#include <iostream>
```

(4) cin 或 cout 语句可以写在同一行上，也可以分开写在多行上。如果写在多行上，除最后一行外，行尾不能加分号。

```
cout <<"x = "<< x
     <<"y = "<< y;
```

(5) 利用 cin 和 cout 在输入输出时不必考虑变量或表达式的类型。对于 cout，系统会自动判断正确的类型并进行输出；对于 cin，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。

【例 3.1】 cin 与 cout 使用示例。

```
#include <iostream>                          //编译预处理命令
using namespace std;                         //使用命名空间 std
int main()                                   //主函数 main()
{
    cout <<"请输入你的姓名和年龄 "<< endl;      //输出提示信息
    char name[16];                           //姓名
    int age;                                 //年龄
    cin >> name;                             //输入姓名
    cin >> age;                              //输入年龄
    cout <<"你的姓名是: "<< name << endl;       //输出姓名
    cout <<"你的年龄是: "<< age << endl;        //输出年龄
    system("pause");                         //输出系统提示信息
    return 0;                                //返回值为 0,返回操作系统
}
```

程序的运行结果如图 3.1 所示。

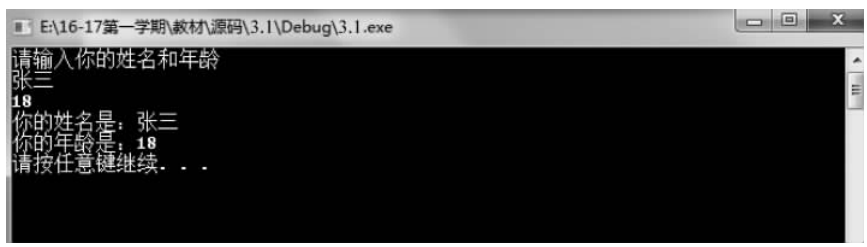

图 3.1 例 3.1 的运行结果

注意：对变量的定义放在执行语句之后。在 C 语言中要求变量的定义必须在执行语句之前。C++允许将变量的定义放在程序的任何位置。

3.2 用 const 定义常量

在 C 语言中常用#define 命令定义符号常量，例如：

```
#define PI 3.14159          //声明符号常量 PI
```

在预编译时进行字符替换，把程序中出现的字符串 PI 全部替换为 3.14159。

C++提供了用 const 定义常量的方法，例如：

```
const float PI = 3.14159;     //定义常量 PI
```

常量 PI 具有数据类型，在编译时要进行类型检查，占用存储单元，在程序运行期间它的值是固定的。

下面介绍 define 宏定义和 const 常变量的区别。

1. 用法区别

(1) define是宏定义，程序在预处理阶段将用define定义的内容进行了替换，因此程序运行时常量表中并没有用define定义的常量，系统不为它分配内存。对于用const定义的常量，程序运行时在常量表中，系统为它分配内存。

(2) 用define定义的常量在预处理时只是直接进行了替换，所以编译时不能进行数据类型检验。用const定义的常量在编译时进行严格的类型检验，可以避免出错。

(3) 用define定义表达式时要注意“边际效应”，例如以下定义：

```
double d1 = 2.0;
#define C1 d1 + d1
#define C2 C1 * C1
```

我们预想的C2＝(d1＋d1)＊(d1＋d1)，编译器将处理成C2＝d1＋d1＊d1＋d1，这就是宏定义的字符串替换的“边际效应”，因此要如下定义：

```
double d1 = 2.0;
const double C1 = d1 + d1;
const double C2 = C1 * C1;
```

用const定义表达式没有上述问题。const定义的常量叫常变量有两个原因，一是const定义常量像变量一样检查类型；二是const可以在任何地方定义常量，编译器对它的处理与变量相似，只是分配内存的地方不同。

2. 实现机制

宏是预处理命令，即在预编译阶段进行字节替换。const常量是变量，在执行时const定义的只读变量在程序运行过程中只有一份副本，因为它是全局的只读变量，存放在静态存储区的只读数据区。根据C/C++语法，当用户声明该量为常量时即告诉程序和编译器不希望此量被修改。对于程序的实现，为了保护常量，特将常量放在受保护的静态存储区内。凡是试图修改这个区域内的值都将被视为非法，并报错。

3. 效果

C++语言用const来定义常量，也可以用#define来定义常量，但是前者比后者有更多的优点：

const常量有数据类型，而宏常量没有数据类型；编译器可以对前者进行类型安全检查，而对后者只进行字符替换，没有类型安全检查，并且在字符替换时可能会产生意想不到的错误(边际效应)。

有些集成化的调试工具可以对const常量进行调试，但是不能对宏常量进行调试。在C++程序中只使用const常量，不使用宏常量，即const常量完全取代宏常量。

【例 3.2】 用 const 定义常量示例。

```
#include <iostream>                    //编译预处理命令
using namespace std;                   //使用命名空间 std
int main()                             //主函数 main()
{
    const float PI = 3.14159;          //定义常量 PI
    float r, s;                        //定义变量
    cout <<"输入半径:";                //输入提示信息
    cin >> r;                          //输入半径 r
    s = PI * r * r;                    //计算面积
    cout <<"面积:"<< s << endl;        //输出面积
    system("pause");                   //输出系统提示信息
    return 0;                          //返回值 0, 返回操作系统
}
```

程序的运行结果如图 3.2 所示。

图 3.2 例 3.2 的运行结果

3.3 函数的重载

C++允许在同一作用域内定义多个同名函数,但要求这些函数参数的类型或个数不相同,这个功能称为函数的重载。在同一个作用域内,函数名相同,参数的类型或个数不同的函数称为重载函数。

函数重载能够在很大程度上方便用户的编程。编译器通过对同名函数的参数类型及参数个数进行分析就能够区分开不同的重载函数。

重载函数的形参个数或类型至少有一个不同,不允许参数个数和类型都相同而返回值类型不同,这是由于系统无法从函数的调用形式判断与哪一个重载函数相匹配。例如下面的代码不能实现函数重载,会出现编译错误。

```
int add(int i, int j);
double add(int i, int j);
```

这是因为在进行函数调用时这两个函数的调用形式是相同的(例如都是“add(a,b);”),编译器无法判断哪一个函数与调用形式相匹配。

【例 3.3】 求两个数中的最小值(分别考虑整数、浮点数的情况)。

```
#include<iostream>
using namespace std;
int Min(int a, int b)                    //求两个整数的最小值
{
    return a<b?a:b;                      //返回 a、b 的最小值
}
float Min(float a, float b)              //求两个浮点数的最小值
{
    return a<b?a:b;                      //返回 a、b 的最小值
}
int main()                               //主函数 main()
{
    int a, b;                            //定义整型变量
    float x, y;                          //定义浮点型变量
    cout <<"输入整数 a,b:";              //输入提示
    cin >> a >> b;                       //输入 a、b
    cout << a <<","<< b <<"的最小值为"<< Min(a, b) << endl;
                                         //输出 a、b 的最小值,调用"int Min(int a, int b)"
    cout << "输入浮点数 x,y:";           //输入提示
    cin >> x >> y;                       //输入 x、y
    cout << x <<","<< y <<"的最小值为"<< Min(x, y) << endl;
                                         //输出 x、y 的最小值,调用"float Min(float a, float b)"
    system("pause");                     //调用库函数 system( ),输出系统提示信息
    return 0;                            //返回值 0, 返回操作系统
}
```

程序的运行结果如图 3.3 所示。

图 3.3　例 3.3 的运行结果

【例 3.4】 用重载函数分别求两个整数或 3 个整数中的最小者。

```
#include<iostream>
using namespace std;
int Min(int a, int b)                    //求两个整数的最小值
{
    return a<b?a:b;                      //返回 a、b 的最小值
}
int Min(int a, int b, int c)             //求 3 个整数的最小值
{
    int t = a<b?a:b;                     //a、b 的最小值
```

```
    t = t < c ? t : c;                    //t、c 的最小值
    return t;                             //返回 a、b、c 的最小值
}
int main()                                //主函数 main()
{
    int a, b, c;                          //定义整型变量
    cout <<"输入整数 a,b,c:";             //输入提示
    cin >> a >> b >> c;                   //输入 a、b、c
    cout << a <<","<< b <<"的最小值为"<< Min(a, b) << endl;
    //输出 a、b 的最小值,调用"int Min(int a, int b)"
    cout << a <<","<< b <<","<< c <<"的最小值为"<< Min(a, b, c) << endl;
    //输出 a、b、c 的最小值,调用"int Min(int a, int b, int c)"
    system("pause");                      //输出系统提示信息
    return 0;                             //返回值 0, 返回操作系统
}
```

程序的运行结果如图 3.4 所示。

图 3.4　例 3.4 的运行结果

3.4　有默认参数的函数

在 C 语言中,在函数调用时形参从实参获得参数值,所以实参的个数应与形参相同。有时多次调用同一函数使用相同的实参值,C++允许给形参提供默认值,这样形参就不一定要从实参取值。例如有一函数声明:

```
float Area(float r = 1.6);        //有默认值的函数声明
```

上面的函数声明指定参数 r 的默认值为 1.6,如果在调用此函数时无实参,则参数 r 的值为 1.6,例如:

```
s = Area();                       //等价于 Area(1.6)
```

C++规定,对于有默认参数的函数:

(1) 默认参数应在函数名第 1 次出现时指定。

(2) 默认参数必须是函数参数表中最右边(尾部)的参数。例如:

```
float Volume(float l = 10.0, float w = 8.0, float h);          //错误
float Volume(float l = 10.0, float w = 8.0, float h = 6.0);  //正确
```

对于上面正确的函数声明，可采用以下形式的函数调用：

```
v = Volume(10.1, 8.2, 6.8);          //形参值全从实参得到，l = 10.1，w = 8.2，h = 6.8
v = Volume(10.1, 8.2);               //最后一个形参的值取默认值，l = 10.1，w = 8.2，h = 6.0
v = Volume(10.1);                    //最后两个形参的值取默认值，l = 10.1，w = 8.0，h = 6.0
v = Volume();                        //形参的值全取默认值，l = 10.0，w = 8.0，h = 6.0
```

【例 3.5】 函数默认参数示例。

```
#include<iostream>
using namespace std;
void Show(char str1[], char str2[] = "", char str3[] = "");
//在声明函数时给出默认值
int main()                                          //主函数 main()
{
    Show("你好!");                                  //str1 值取"你好!"，str2 与 str3 取默认值
    Show("你好，", "欢迎学习 C++!");
    // str1 值取"你好，"，str2 取"欢迎学习 C++!"，str3 取默认值
    Show("你好", "，", "欢迎学习 C++!");
    // str1 值取"你好，"，str2 取"，"，str3 取"欢迎学习 C++!"
    system("pause");                                //输出系统提示信息
    return 0;                                       //返回值 0，返回操作系统
}
void Show(char str1[], char str2[], char str3[])
{
    cout << str1 << str2 << str3 << endl;           //输出 str1、str2、str3
}
```

程序的运行结果如图 3.5 所示。

图 3.5　例 3.5 的运行结果

3.5　引用

引用是 C++的特性。简单来说，引用就是另一个变量的别名；也就是说，引用和它所指的对象是同一个实体。引用的主要用途之一是作为函数的输出参数使用，在作为输出参数方面，它可以起到与指针参数相同的作用，但其使用更简便。

3.5.1 引用的概念

建立“引用”的作用是为一个变量起另一个名字，对一个变量的“引用”的所有操作实际上都是对其所代表的(原来的)变量的操作。

设有一个变量 x，要给它起一个别名 y，可以这样写：

```
float x;          //定义变量 x
float &y = x;     //声明 y 是一个浮点型变量的引用变量,它被初始化为 x
```

声明后，使用 x 或 y 代表同一变量。在上述声明中，& 是“引用声明符”，对变量声明一个引用，并不另开辟内存单元，x 和 y 代表同一变量存储单元。

在声明一个引用时必须同时使之初始化。

在函数中声明一个变量的引用后，在函数执行期间，该引用一直与其代表的变量相联系，不能再作为其他变量的别名。例如：

```
int a, b;            //定义整型变量 a、b
int &c = a;          //使 c 成为变量 a 的引用(别名)
int &c = b;          //又使 c 成为变量 b 的引用(别名)是错误的
```

【例 3.6】 变量的引用使用示例。

```
#include<iostream>
using namespace std;
int main()                                  //主函数 main()
{
    int a = 10;                             //定义变量
    int &b = a;                             // b 为 a 的引用,a 与 b 代表相同变量存储单元 a
    b = b + 2;                              // b 的值自动加 2,a 与 b 的值都为 a12
    cout <<"a 的地址:"<< &a << endl;        //输出 a 的地址
    cout <<"b 的地址:"<< &b << endl;        //输出 b 的地址
    cout <<"a 的值:"<< a << endl;           //输出 a 的值
    cout <<"b 的值:"<< b << endl;           //输出 a 的值
    system("pause");                        //输出系统提示信息
    return 0;                               //返回值 0, 返回操作系统
}
```

程序的运行结果如图 3.6 所示。

图 3.6 例 3.6 的运行结果

3.5.2　将引用作为函数的参数

C++增加“引用”的主要目的是利用它作为函数参数，以便扩充函数传递数据的功能。

在C语言中将变量名作为实参，这时将变量的值传递给形参。传递是单向的，在调用函数时形参和实参不是同一个存储单元。在执行函数期间形参值发生变化并不传回给实参。

【例 3.7】 以变量为实参不能实现交换变量的值的示例。

```
#include<iostream>
using namespace std;
void Swap(int a, int b)                    //不能实现交换实参变量的值
{
    int t = a;
          a = b;
          b = t;                           //交换a、b的值
}
int main()                                 //主函数main()
{
    int m = 6, n = 8;                      //定义整型变量
    Swap(m, n);                            //调用函数Swap()
    cout << m <<""<< n << endl;            //输出m、n的值
    system("pause");                       //输出系统提示信息
    return 0;                              //返回值0，返回操作系统
}
```

将变量 m、n 作为函数 Swap(m,n)的实参，这时将变量的值传递给形参 a、b。传递是单向的，在调用的时候形参和实参不是同一个存储单元，在执行函数的时候形参的值发生变化并不传回给实参，如图 3.7 所示。

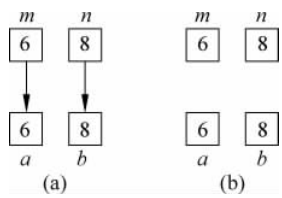

图 3.7　变量作为实参不能实现变量的值

程序的运行结果如图 3.8 所示。

图 3.8　例 3.7 的运行结果

【例 3.8】 用指针变量作为形参实现两个变量的值的互换。

```
#include<iostream>
using namespace std;
void Swap(int *p, int *q)                    //实现交换*p与*q的值
{
    int t = *p;
       *p = *q;
       *q = t;                               //循环赋值交换*p与*q的值
}
int main()                                   //主函数main()
{
    int m = 6, n = 8;                        //定义整型变量
    Swap(&m, &n);                            //调用函数Swap()
    cout<<m<<""<<n<<endl;                    //输出m、n的值
    system("pause");                         //输出系统提示信息
    return 0;                                //返回值0,返回操作系统
}
```

在 C 程序中可以用指针传递变量地址的方法使形参得到一个变量的地址,这时形参指针变量指向实参变量单元,如图 3.9 所示。

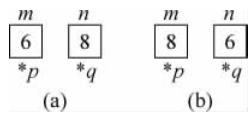

图 3.9 用指针变量作为形参实现两个变量的值的互换

程序的运行结果如图 3.10 所示。

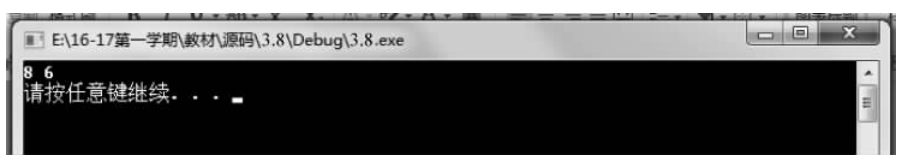

图 3.10 例 3.8 的运行结果

【例 3.9】 利用引用形参实现交换两个变量的值。

```
#include<iostream>
usingnamespace std;
void Swap(int &a, int &b)                    //实现交换实参变量的值
{
    int t = a; a = b; b = t;                 //循环赋值交换a与b的值
}
int main()                                   //主函数main()
{
    int m = 6, n = 8;                        //定义整型变量
    Swap(m, n);                              //调用函数Swap()
```

```
    cout << m <<""<< n << endl;                    //输出 m、n 的值
    system("pause");                               //输出系统提示信息
    return 0;                                      //返回值 0，返回操作系统
}
```

在 C++中把变量的引用作为函数形参，由于形参是实参的引用，也就是形参是实参的别名，这样对形参的操作等价于对实参的操作，如图 3.11 所示。

图 3.11　变量的引用作为函数形参实现交换

程序的运行结果如图 3.12 所示。

图 3.12　例 3.9 的运行结果

3.5.3　引用和指针的区别

引用和指针既有相似之处，也有明显的区别，总结如下：

(1) 引用和指针都可以通过一个变量访问另一个变量，但访问时的语法形式不同。引用采用的是直接访问形式，而指针采用的是间接访问形式。

(2) 当作为函数参数使用时，引用所对应的实参是某个变量的名字，而指针所对应的实参是某个变量的地址。引用在作为函数参数使用时其效果与指针相同，但使用更方便。

(3) 引用在定义时被初始化，其后不能被改变(即不能再成为另一个变量的别名)，而指针可以再通过赋值的方式指向另一个变量。

3.5.4　常指针与常引用

1. 常指针

在用 const 修饰指针时，由于 const 的位置不同而含义不同。

```
char * const ptr1 = "abcd";
```

该语句的作用是定义一个常指针 ptr1，它存放的是字符串"abcd"的首地址，这个地址值是不能改变的。

```
const char * ptr2 = "abcd";
```

该语句的作用是定义一个指向常量的指针变量 ptr2。

```
const char * const ptr3 = "chen";
```

该语句定义了一个指向常量的常指针变量。ptr3 中的地址值不能改变，ptr3 指向的字符串中的内容也不能改变。

2. 常引用

常引用就是用 const 对引用加以限定，被说明的引用为常引用，表示不允许改变该引用的值。

其格式如下：

```
const 类型说明符 &引用名;
```

例如：

```
int a = 6;                    //定义整型变量 a,初值为 6
const int &b = a;             //声明常引用,不允许改变 b 的值
b = 8;                        //改变常引用 b 的值,错误
a = 8;                        //改变 a 的值,正确
```

常引用通常用作函数形参，这样能保证形参的值不被改变。

注意：C++不区分变量的 const 引用和 const 变量的引用。程序不能给引用本身重新赋值，使它指向另一个变量，因此引用总是 const 的。如果对引用应用关键字 const，其作用就是使目标成为 const 变量，即没有：

```
const double const& a = 1;
```

只有：

```
const double& a = 1;
```

【例 3.10】 常引用形参示例。

```
#include <iostream>
using namespace std;
struct Person
{
    char name[20];                          //姓名
    char sex[3];                            //性别
};
void Show(const Person &p)
{
    cout <<"姓名:"<< p.name << endl;        //输出姓名
    cout <<"性别:"<< p.sex << endl;         //输出性别
}
```

```
int main()                          //主函数 main()
{
    Person p = {"李倩", "女"};      //定义结构体变量
    Show(p);                        //输出 p
    system("pause");                //输出系统提示信息
    return 0;                       //返回值 0, 返回操作系统
}
```

程序的运行结果如图 3.13 所示。

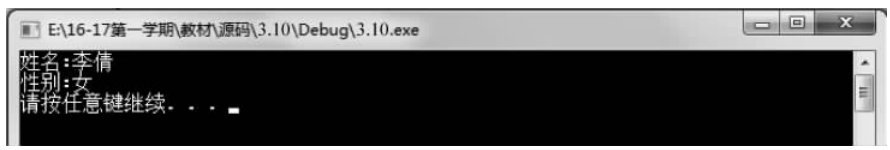

图 3.13　例 3.10 的运行结果

备注：

(1) 结构、联合和枚举名可直接作为类型名。

声明类型时：

```
enum Day{SUN, MON, TUE, WED, THU, FRI, SAT};
union U
{
    char c;
    float f;
    double d;
};
struct Student
{
    char m_strName[20];
    char m_strID[12];
    char m_cSex;
    char m_strMajor[20];
};
```

定义变量时(C 语言中)：

```
enum Day day;
union U u;
struct Student student1;
```

定义变量时(C++语言中)：

```
Day day;
U u;
Student student1;
```

在该程序中用结构名 Person 作为类型来定义变量 p，在 C 语言中不能用结构名来定义结构变量名，必须在结构名前加 struct 才能定义结构变量，即 C 语言应采用以下形式定义：

```
structPerson p = {"李倩", "女"};        //定义结构体变量
```

(2) 可以用常量或表达式对常引用进行初始化，例如：

```
int a = 6;                              //定义变量
const int &b = a + 3;                   //正确，可以用表达式对常引用进行初始化
int &c = a + 3;                         //错误，对非常引用只能用变量进行初始化
```

在用表达式对常引用进行初始化时系统将生成一个临时变量，用于存储表达式的值，引用是临时变量的别名。例如将“const int &b = a + 3;”变换为：

```
int tem = a + 4;                        //将表达式的值存放在临时变量 tem 中
const int &b = tem;                     //声明 b 是 tem 的引用(别名)
```

3.5.5 引用小结

引用是 C++所独有的特性。指针存在种种问题，间接引用指针会使代码的可读性差，编程易出错。

在引用的使用中，单纯取个别名是毫无意义的，引用的主要目的是用在函数的参数传递中解决大对象的传递效率和空间都不如意的问题。

引用能够保证参数传递中不产生副本，从而发挥指针的作用，提高传递的效率，通过使用 const 保证了引用传递的安全性。

引用具有表达清晰的优点。引用传递与值传递在使用方法上唯一的区别在于函数的形式参数声明。

3.6 动态分配内存

在 C 语言中使用函数 malloc( )和 free( )动态分配内存和释放动态分配的内存。C++语言使用能完成动态内存分配和初始化工作的运算符 new 以及一个能完成清理与释放内存工作的运算符 delete 来管理动态内存。

3.6.1 new 关键字

在 C++中 new 运算符用于动态分配一块连续的内存空间，其基本语法形式如下：

```
   new < type >
/ * 从一块自由存储区中分配一块 sizeof(type)字节大小的内存 * /
```

其中，type 可以是 C++支持的所有数据类型名，可以是基本的数据类型，也可以是自定

义数据类型。

new 运算符的使用方法有以下 3 种：

```
new 类型;                  //分配存储单个数据的空间时不指定初始值
new 类型(初值);            //分配单个数据的存储空间时将指定初始值
new 类型[元素个数];        //分配数组存储空间时不指定初始值
```

例如：

```
new int;                   //分配一个存放整数的空间,返回一个指向整型数据的指针
new int(6);                //分配一个存放整数的空间,并且初始化为 6
new char[16];              //分配一个存放字符数组的空间,该数组有 16 个元素
```

new 运算符返回一个指向所分配存储空间的第 1 个单元的指针，如果当前存储器没有足够的内存空间可分配，则返回 NULL。例如：

```
int * p1 = new int;        //分配用于存储一个整型数据的连续区域
                           //并将首地址返回给 p1
```

使用 new 可以为数组动态分配空间，这时需要在类型名后加上数组的大小。

例如：

```
int * p, * q, * r;
p = new int(5);
//分配一个整数内存空间,该空间存储的初始值为 5
q = new int;
//分配一个整数内存空间,但没有进行初始化
r = new int[10];
//分配 10 个整数的内存空间(r[0]～r[9]),并返回该空间的首地址(r = &r[0])
int * p2 = new int[10];
//分配一个整型的一维数组
//大小为 10 个整型数据,并将首地址返回给 p2
int * p3 = new int[4][5];
//分配一个整型数组
//存储一个 4×5 的二维数组,并将首地址返回给 p3
```

注意：在使用 new 动态分配内存的时候，如果没有足够的内存满足分配要求，new 将返回空指针(NULL)，因此通常要对内存的动态分配是否成功进行检查。

3.6.2　delete 运算符

对于动态分配的内存在使用完后一定要及时归还给系统。如果应用程序对有限的内存只取不还，系统很快就会因为内存枯竭而崩溃。利用 new 动态分配的存储空间通常可以利用 delete 运算符进行释放。

delete 运算符的使用方法有下面两种形式：

```
delete 指针变量名
delete [ ]指针变量名
```

第 1 种形式释放用 new 分配的单个数据的存储空间，第 2 种形式用于释放数组对象空间，对应“new 类型名[表达式]”。delete 操作没有返回值，或者说它的返回类型是 void。

在使用 delete 释放内存空间时应注意以下两点：

(1) 利用 new 运算符分配的内存空间只允许使用一次 delete，如果对同一块空间进行多次释放将会导致严重错误。

(2) delete 只能用来释放动态分配的内存空间。

【例 3.11】 new/delete 运算符使用示例。

```
#include<iostream>
using namespace std;
int main()                                          //主函数 main()
{
    int *p;                                         //定义整型指针
    p = new int(16);                                //分配单个整数的存储空间,并初始化为 16
    if (p == NULL)
    {
        cout<<"分配存储空间失败!"<<endl;
        exit(1);                                    //退出程序的运行,并向操作系统返回 1
    }
    cout<< *p<<endl;                                //输出 p 所指向的动态存储空间的值 16
    delete p;                                       //释放存储空间
    p = new int;                                    //分配单个整数的存储空间
    if (p == NULL)
    {
        cout<<"分配存储空间失败!"<<endl;
        exit(2);                                    //退出程序的运行,并向操作系统返回 2
    }
    *p = 8;                                         //将 p 指向的动态存储空间赋值为 8
    cout<< *p<<endl;                                //输出 p 所指向的动态存储空间的值 8
    p = new int[8];                                 //分配整型数组存储空间
    if (p == NULL)
    {
        cout<<"分配存储空间失败!"<<endl;
        exit(3);                                    //退出程序的运行,并向操作系统返回 3
    }
    int i;                                          //定义整型变量
    for (i = 0; i<8; i++)
        p[i] = i;                                   //为数组赋元素值
    for (i = 0; i<8; i++)
        cout<<p[i]<<"";                             //输出数组元素值"0 1 2 3 4 5 6 7"
    cout<<endl;                                     //换行
    delete []p;                                     //释放存储空间
    system("pause");                                //输出系统提示信息
    return 0;                                       //返回值 0, 返回操作系统
}
```

程序的运行结果如图 3.14 所示。

图 3.14　例 3.11 的运行结果

3.7　布尔类型

布尔类型 bool 是 ISO/ANSI(国际标准化组织/美国国家标准化组织)后来增补到 C++ 语言中的。

布尔变量包含两种取值,即 true 和 false。如果在表达式中使用布尔变量,它将把自身取值的 true 或 false 分别转换为 1 或 0。如果将数值转换为布尔类型,如数值是零,布尔变量为 false; 如数值是非零值,布尔变量就为 true。

【例 3.12】 编写判断一个整型是否为质数的函数,并用此函数输出 1～100 的质数,要求编写测试程序。

说明: 一个整型 n 如果大于 1,并且不能被 2～n－1 的整数所整除,那么 n 为质数,由质数的定义很容易实现判断一个整型是否为质数的函数,具体程序实现如下。

```
#include<iostream>
using namespace std;
bool IsPrime(int n)
{
    if (n<= 1) return false;            //质数至少为 2
    for (int p = 2; p<n; p++)
        if (n % p == 0) return false;   //如果 n 能被 p 整除,为合数
    return true;                        //n 不能被 2～n-1 的所有整数整除为质数
}
int main()                              //主函数 main()
{
    for (int n = 1; n<= 100; n++)
        if (IsPrime(n))                 //如果 n 为质数
            cout<<n<<"";                //那么输出 n
    cout<<endl;                         //换行
    system("pause");                    //输出系统提示信息
    return 0;                           //返回值 0, 返回操作系统
}
```

程序的运行结果如图 3.15 所示。

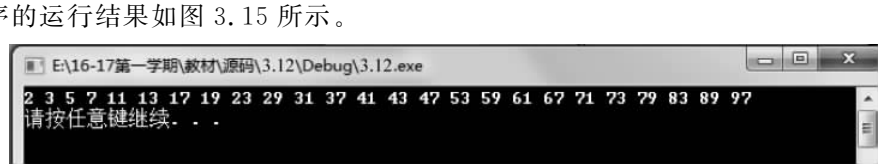

图 3.15　程序 3.12 的运行结果

3.8 函数原型

在C++程序中，如果函数调用的位置出现在函数的定义之前，则必须在调用函数之前给出函数原型，即函数名称、函数参数的类型、函数的返回值。其主要目的是让编译器能够检查调用函数的参数是否与给出的函数原型相符合，从而减少编程时的差错。另外，通过声明函数原型可以使程序的结构更清晰。

关于函数原型的说明有以下几点：

(1) 函数原型声明的格式：

```
函数返回值类型函数名(参数声明列表);
```

在参数声明列表中可以只包含参数的类型，而不包含参数的名字。例如：

```
double Area(double, double);
```

(2) 如果函数的定义在前，调用在后，则不必再给出原型声明，因为这时函数定义的说明部分已经起到了函数原型声明的作用。

(3) 如果函数原型声明(函数说明)中没有给出函数的返回值类型，则其默认类型为int。例如，下面的两条语句是等价的。

```
int Add(int, int);
Add(int, int);
```

(4) 如果一个函数没有返回值，则必须在其函数原型(函数说明)中写出其函数返回类型是void。这时可以在函数定义中省略“return;”语句。例如：

```
void fun1();
```

(5) 在C++语言中，如果在函数原型中没有标明参数，则说明该函数的参数表为空(void)，即该函数不带任何参数。下面的两条语句是等价的。

```
void f();
void f(void);
```

而在C语言中，下面两条函数原型说明语句代表了不同的含义。

```
void f();           //该函数的参数没有给出,它可能带有多个参数
void f(void);       //该函数不带任何参数
```

【例 3.13】 函数原型的示例。

```
#include <iostream>
using namespace std;
const unsigned int ARRAY_SIZE1 = 20;
```

```
const unsigned int ARRAY_SIZE2 = 12;
struct Student                                  //学生结构体
{
    char m_strName[ARRAY_SIZE1];                //姓名
    char m_strID[ARRAY_SIZE2];                  //编号
    char m_cSex;                                //性别：'0'为男,'1'为女
    char m_strMajor[ARRAY_SIZE1];               //专业
};
void PrintInfo(Student student);                //函数原型声明
int main()
{   Student Student1;
    cout<<"输入姓名：\n";
    cin>>Student1.m_strName;
    cout<<"输入学生编号：\n";
    cin>>Student1.m_strID;
    cout<<"输入性别('0':男 '1':女)：\n";
    cin>>Student1.m_cSex;
    cout<<"输入专业：\n";
    cin>>Student1.m_strMajor;
    PrintInfo(Student1);
    return 0;
}
void PrintInfo(Student student)
{   cout<<"该学生的信息为：\n";
    cout<<"姓名："<<student.m_strName<<"  "<<"编号："<<student.m_strID<<"  "
    <<"性别：";
    if(student.m_cSex == '0')
        cout<<"男";
    else
        cout<<"女";
    cout<<"  "<<"专业："<<student.m_strMajor<<endl;
}
```

程序的运行结果如图 3.16 所示。

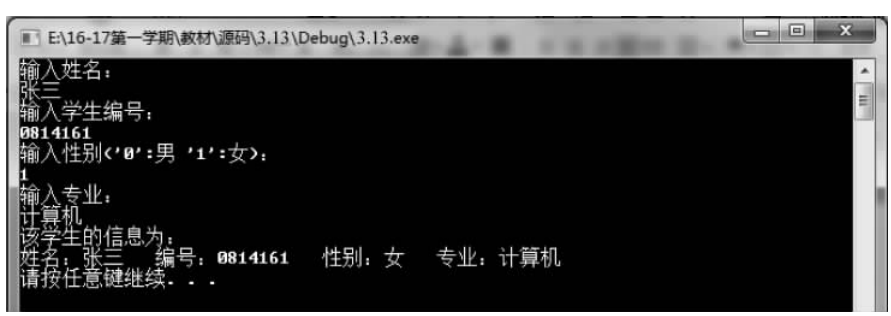

图 3.16　例 3.13 的运行结果

3.9 作用域运算符

如果同名的两个变量中一个是全局的，另一个是局部的，那么在局部变量的作用域内局部变量将屏蔽掉全局变量。如何在局部变量的作用域内使用同名的全局变量呢？可以通过作用域运算符“::”来实现。

【例 3.14】 作用域运算符示例 1。

```
#include<iostream>
using namespace std;
int avar=10;
int main();
{
    int avar;
    avar=25;
    cout<<"avar is "<<avar<<endl;
    return 0;
}
```

程序的输出结果如下：

```
avar is 25
```

这是因为在通常情况下如果有两个同名变量，一个是全局变量，另一个是局部变量，那么局部变量在其作用域内具有较高的优先权，它将屏蔽全局变量。

在 main 函数的输出语句中使用的变量 avar 是 main 函数内定义的局部变量，因此结果为局部变量的值。如果希望在局部变量的作用域内使用同名的全局变量，可以在该变量前加上“::”，此时::avar 代表全局变量，“::”就是作用域运算符。

【例 3.15】 作用域运算符示例 2。

```
#include<iostream>
using namespace std;
int avar;
int main()
{
    int avar;
    avar=25;                    //局部变量
    ::avar=10;                  //全局变量
    cout<<"local avar = "<<avar<<endl;
    cout<<" global avar = "<<::avar<<endl;
    system("pause");
    return 0;
}
```

程序的运行结果如图 3.17 所示。

图 3.17　例 3.15 的运行结果

3.10　内置函数

对于普通函数，在调用时需要经历如图 3.18 所示的过程：

（1）主调函数执行函数调用语句；

（2）系统将主调函数的局部变量和返回地址压入堆栈，并转入函数 max 的入口，传递相应参数；

（3）执行函数 max 中的语句；

（4）从堆栈中弹出主调函数的运行环境，并带回返回值；

（5）执行主调函数中的剩余语句。

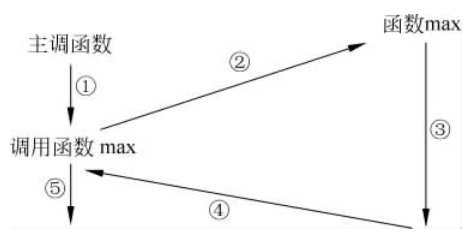

图 3.18　函数调用示意图

C++为避免以上 5 个步骤，提供了 inline 函数。对于 inline 函数，在编译时 C++的编译器将使用函数体中的代码代替函数调用表达式，而不像普通函数那样需要经历调用过程，因而能够获得更快的执行速度。

在函数说明前加上关键字“inline”，该函数就被声明为内置函数。每当程序中出现对该函数的调用时，C++编译器使用函数体中的代码代替函数调用表达式，这样能够加快代码的执行，减少调用开销。

【例 3.16】　内置函数的示例。

```
#include <iostream.h>
inline float area(float r)                                    //内置函数
{
    return 3.1416 * r * r;
}
int main()
{
    for(int i = 1; i <= 3; i++)
```

```
    cout <<"r = "<< i <<" area = "<< area(i) << endl;      //内置函数的调用
    return 0;
}
```

程序的运行结果如图 3.19 所示。

图 3.19　例 3.16 的运行结果

关于内置函数的说明如下：

(1) 内置函数在被调用前必须进行完整的定义，否则编译器无法知道应该插入什么代码。内置函数通常写在主函数的前面。

(2) C++的内置函数具有与 C 中的宏定义 # define 相同的作用和相似的机理，但是消除了 # define 的不安全因素。

所避免的 # define 的不安全因素如下：

(1) 内联函数就像其他 C++函数一样，在调用时编译器会进行正确的类型检查，而预处理器的宏不支持类型检查。

(2) 内联函数不像宏那样在使用不正确时会产生意想不到的副作用。

(3) 内联函数可以使用调试程序调试。

【例 3.17】 使用带参的宏定义完成乘 2 的功能。

```
#include <iostream.h>
#define doub(x) x * 2
int main()
{
    for(int i = 1; i <= 3 ; i++)
    cout << i <<" doubled is"<< doub(i) << endl;
    cout <<"1 + 2 doubled is"<< doub(1 + 2) << endl;
    return 0;
}
```

程序的运行结果如图 3.20 所示。

图 3.20　例 3.17 的运行结果

程序的运行结果并非是我们想要的结果，原因是 define 宏定义出现了边际效应。如果使用 inline 函数就不会出现上述问题。

【例 3.18】 使用内置函数解决上述问题。

```
#include <iostream.h>
inline int doub(int x)
{
    return x * 2;
}
int main()
{
    for(int i = 1; i <= 3 ; i++)
    cout << i <<" doubled is"<< doub(i) << endl;
    cout <<"1 + 2 doubled is"<< doub(1 + 2) << endl;
    system("pause");
    return 0;
}
```

程序的运行结果如图 3.21 所示。

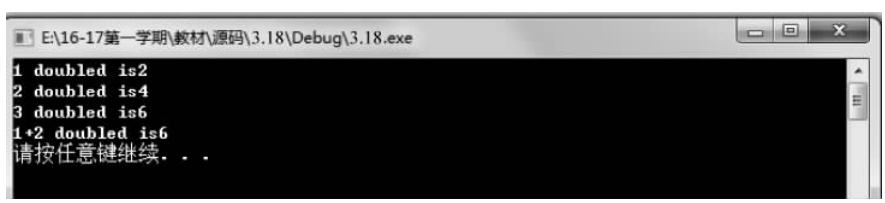

图 3.21　例 3.18 的运行结果

但是这样做有一个问题，就是会产生函数代码的多个副本，并分别插入到程序中每一个调用该函数的位置上，而不是只产生一个副本。

对内联函数所做的任何修改要求使用该函数的所有地方都要重新编译，在程序开发和维护的某些场合这是有意义的。

对编译器而言，限定词 inline 是一个请求(request)，而不是一个命令。如果出于种种原因，编译器不能满足这个请求，那么这个内联函数将被编译成一个普通的函数，即请求无效。

编译器的不同版本对内联函数的限制也是不同的，有些编译器不允许用递归函数和包含 static 变量、循环语句、switch 语句、goto 语句的函数作为内联函数。

3.11　C++的注释

在程序中注释语句的作用主要有以下两个：

(1) 为方便读程序，程序员通常会增加一些说明性的文字。

(2) 在程序中，如果对某(几)条语句暂时不能决定是否需要删除可以暂时将其注释。

在 C++中提供了下面两种语句注释方法。

（1）块注释：即使用/*开始、*/结束的形式，这种形式不允许出现注释嵌套，主要用于多行注释，例如程序开头的功能说明、版权说明等信息。

（2）行注释：即以“//”开始，直到行结尾结束的注释。这种形式多用于注释单行，或在一行的后面添加说明语句。这种注释方式允许嵌套使用。

下面用一个例子说明C++中两种注释语句的用法：

```
/*这是一个说明C++中注释语句的例子
当前使用的块注释方式*/
class MyComplex
{  //这是用C++定义的一个复数类
    private:
      double x;
    double y;
    void display()
};
```

上面例子中的前两句采用了块注释；第3行在行末使用行注释说明本条语句的作用，并且使用了注释嵌套；第4行用行注释注释掉一行。

综合实例

指针作为函数参数和引用作为函数参数的比较。

这里定义函数StudentSat，进行男生和女生人数的统计，并通过指针参数进行统计结果的函数输出。要求程序的输出结果如图3.22所示。

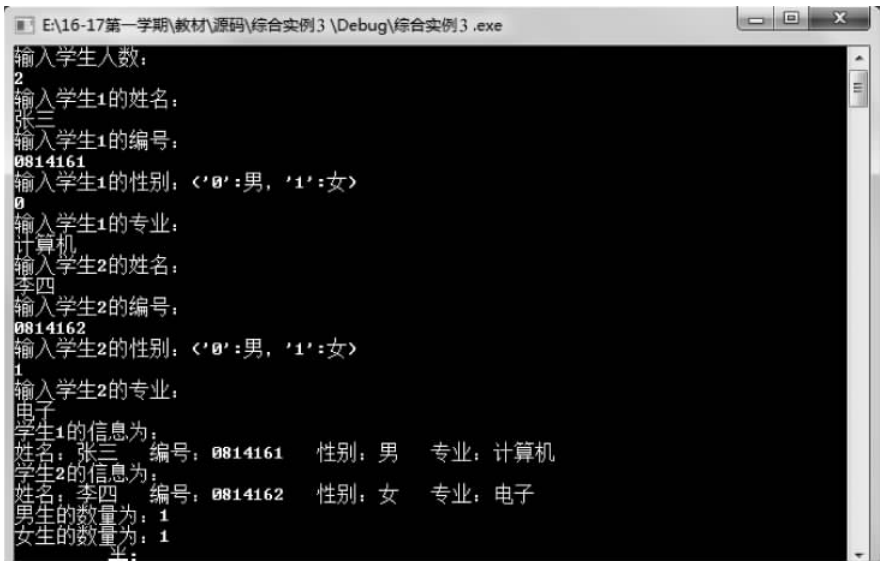

图3.22　综合实例的运行结果

1. 指针作为函数的参数

用指针作为函数的参数的实现代码如下：

```
#include <iostream>
using namespace std;
struct Student                                          //学生结构体
{
    char m_strName[20];                                 //姓名
    char m_strID[12];                                   //编号
    char m_cSex;                                        //性别: '0'为男,'1'为女
    char m_strMajor[20];                                //专业
};
void StudentStat(Student *aStudent, int nNumber, int *pnMaleNumber, int *pnFemaleNumber)
{
    int nMaleNumber, nFemaleNumber;
    nMaleNumber = 0;
    nFemaleNumber = 0;
    for(int i=0; i<nNumber; i++)
    {
        if(aStudent[i].m_cSex == '0')
            nMaleNumber++;
        else
            nFemaleNumber++;
    }
    *pnMaleNumber = nMaleNumber;
    *pnFemaleNumber = nFemaleNumber;
}
int main()
{
    int *pnNumber;
    pnNumber = new int;
    cout<<"输入学生人数: "<<endl;
    cin>>*pnNumber;
    Student *aStudent = new Student[*pnNumber];         //灵活的局部变量说明
    for(int i=0; i<*pnNumber; i++)
    {
        cout<<"输入学生"<<i+1<<"的姓名: "<<'\n';
        cin>>aStudent[i].m_strName;
        cout<<"输入学生"<<i+1<<"的编号: "<<'\n';
        cin>>aStudent[i].m_strID;
        cout<<"输入学生"<<i+1<<"的性别: "<<"('0':男,'1':女)\n";
        cin>>aStudent[i].m_cSex;
        cout<<"输入学生"<<i+1<<"的专业: "<<'\n';
        cin>>aStudent[i].m_strMajor;
    }
for(i=0; i<*pnNumber; i++)
    {cout<<"学生"<<i+1<<"的信息为: "<<'\n';
    cout<<"姓名: "<<aStudent[i].m_strName<<"   "
```

```
    <<"编号: "<< aStudent[i].m_strID <<"  "<<"性别: ";
        if(aStudent[i].m_cSex == '0')
            cout <<"男";
        else
            cout <<"女";
        cout <<"  ";
        cout <<"专业: "<< aStudent[i].m_strMajor << endl;
    }
    int nMaleNumber, nFemaleNumber;
    StudentStat(aStudent, *pnNumber, &nMaleNumber, &nFemaleNumber);
    cout <<"男生的数量为: "<< nMaleNumber <<'\n';
    cout <<"女生的数量为: "<< nFemaleNumber <<'\n';
    delete pnNumber;
    delete []aStudent;
     system("pause");
    return 0;
}
```

2. 引用作为函数的参数

用引用作为函数的参数的实现代码如下：

```
#include <iostream>
using namespace std;
struct Student                                          //学生结构体
{
    char m_strName[20];                                 //姓名
    char m_strID[12];                                   //编号
    char m_cSex;                                        //性别: '0'为男,'1'为女
    char m_strMajor[20];                                //专业
};
void StudentStat(Student *aStudent, int nNumber, int &rnMaleNumber, int &rnFemaleNumber)
{   int nMaleNumber, nFemaleNumber;
    nMaleNumber = 0;
    nFemaleNumber = 0;
    for(int i = 0; i < nNumber; i++)
    {   if(aStudent[i].m_cSex == '0')
            nMaleNumber++;
        else
            nFemaleNumber++;
    }
    rnMaleNumber = nMaleNumber;
    rnFemaleNumber = nFemaleNumber;
}
int main()
{
    int *pnNumber;
    pnNumber = new int;
```

```
    cout <<"输入学生人数: "<< endl;
    cin >> * pnNumber;
    Student * aStudent = new Student[ * pnNumber]; //灵活的局部变量说明
    for(int i = 0; i < * pnNumber; i++)
    {
        cout <<"输入学生"<< i + 1 <<"的姓名: "<<'\n';
        cin >> aStudent[i].m_strName;
        cout <<"输入学生"<< i + 1 <<"的编号: "<<'\n';
        cin >> aStudent[i].m_strID;
        cout <<"输入学生"<< i + 1 <<"的性别: "<<"('0':男,'1':女)\n";
        cin >> aStudent[i].m_cSex;
        cout <<"输入学生"<< i + 1 <<"的专业: "<<'\n';
        cin >> aStudent[i].m_strMajor;
    }
for(i = 0; i < * pnNumber; i++)
    {cout <<"学生"<< i + 1 <<"的信息为: "<<'\n';
    cout <<"姓名: "<< aStudent[i].m_strName <<"  "
    <<"编号: "<< aStudent[i].m_strID <<"  "<<"性别: ";
        if(aStudent[i].m_cSex == '0')
            cout <<"男";
        else
            cout <<"女";
        cout <<"";
        cout <<"专业: "<< aStudent[i].m_strMajor << endl;
    }
    int nMaleNumber, nFemaleNumber;
    StudentStat(aStudent, * pnNumber, nMaleNumber, nFemaleNumber);
    cout <<"男生的数量为: "<< nMaleNumber <<'\n';
    cout <<"女生的数量为: "<< nFemaleNumber <<'\n';
    delete pnNumber;
    delete []aStudent;
     system("pause");
    return 0;
}
```

本章小结

本章主要介绍了C++程序的基本格式和一般编写过程，C++在非面向对象方面的一些特性，如I/O流、内置函数、函数原型、带默认参数的函数、函数重载、const修饰符、new/delete运算符、引用等。

本章的学习目标是通过比较C源程序和C++源程序熟悉C++程序的风格，掌握C++程序的格式、结构特点；掌握C++在非面向对象方面的特点。

习题

一、选择题

1. 适宜采用 inline 定义函数的情况是(　　)。

A. 函数体含有循环语句　　B. 函数体含有递归语句

C. 函数代码少、频繁调用　　D. 函数代码多、不常调用

2. 使用地址作为实参传给形参,下列说法正确的是(　　)。

A. 实参是形参的备份　　B. 实参与形参无联系

C. 形参是实参的备份　　D. 实参与形参是同一对象

3. 在 C++中使用流进行输入与输出,其中(　　)用于屏幕输入。

A. cin　　B. cerr　　C. cout　　D. clog

二、改错题

1. 计算两个数之和。

```
#include <iostream>
int main()
{    int x , y , sum ;
     cout <<"Enter two numbers:"<<'\n';          //提示用户输入两个数
     cin >> x ;                                  //从键盘输入变量 x 的值
     cin >> y ;                                  //从键盘输入变量 y 的值
     sum = add( x , y ) ;
     cout <<"The sum is :"<< sum << '\n';        //输出 sum 的值
     return 0;
}
int add( int a , int b)
{
     int c;
     c = a + b;
     return c;
}
```

2. 输出引用的地址。

```
#include <iostream>
using namespace std;
int main()
{    float f = (float)1.1;
     float &rf = f;
     cout <<"f = "<< f <<"   "<<"rf = "<< rf << endl;
     f = float(2.2);
     cout <<"f = "<< f <<"   "<<"rf = "<< rf << endl;
     rf = float(3.3);
     cout <<"f = "<< f <<"   "<<"rf = "<< rf << endl;
```

```
    cout <<"变量 f 的地址: "<< f << endl;
    cout <<"引用 rf 的地址: "<< rf << endl;
    return 0;
}
```

三、编程题

1. 建立一个被称为 sroot()的函数，返回其参数的二次根。重载 sroot()两次，让它分别返回整数、双精度数的二次根(为了计算二次根，可以使用标准库函数 sqrt())。

2. 编写一个程序动态分配一个浮点空间，输入一个数到该空间中，计算该数为半径的圆的面积，并在屏幕上显示，最后释放该空间。请用 new 和 delete 运算符。

3. 编写一个程序，输入两个整数，将它们按从小到大的顺序输出。要求使用变量的引用。

4. 读入 9 个双精度的数，把它们存放在一个存储块里，然后求出它们的积。要求使用动态分配和指针操作。

四、上机操作题

1. 编写 C++风格的程序，通过键盘输入一个整数、一个字符和一个字符串到相应的变量中，然后在屏幕上输出这些变量的值。

2. 用户通过键盘输入整数的个数 n 以及每个整数的值，将这些整数存入由 new 运算符分配的动态数组中，对这 n 个整数进行排序，并输出排序结果，最后通过 delete 运算符完成相关内存的释放。

3. 编写一个函数，将引用作为函数参数，实现两个复数变量值的交换。提示：首先定义复数结构体。

4. 利用函数重载编写两个分别求整数和双精度数绝对值的函数，要求有输入和输出。