第3章 函 数

3.1 函数的概述

在程序设计中，函数是程序的基本组成单位，是功能相对独立的、使用频率相对较高的子程序或子模块。通常，人们在解决一个复杂问题时，总是将大问题分解成几个较容易的小问题，然后分别求解。那么，当程序员设计一个复杂的应用程序时，往往也是把整个程序划分为若干个功能较单一的程序模块，然后分别予以实现，这些模块就是一个个函数。

因此，函数的编写可以降低问题的难度。同时，函数的编写也有利于实现代码的重用，提高代码开发的效率，便于开发人员之间的分工合作，便于程序的修改维护。

3.2 函数的定义和调用

变量在使用时，必须先定义，后使用。与之类似，函数在使用时，也要经过函数的定义、函数的声明和函数的调用这三个步骤。对这三方面的统一协调和正确处理是编写和使用函数、实现函数功能的关键。

3.2.1 函数的定义

函数由函数头和函数体两部分构成，函数定义的语法形式如下。

```
[函数返回值类型标识符] <函数名>(类型标识符 [形式参数 1],…)
{
  函数体
}
```

例如：求两数中最小者的函数 min()的定义形式如下。

```
int min(int x,int y)
{ int m;
  if(x<y)
    m=x;
  else
    m=y;
  return m;
}
```

1. 函数头

(1) 函数头由函数返回值类型标识符、函数名和形式参数列表组成。函数头不是语句，函数头后面不能够有分号。

(2) [函数返回值类型标识符]是用来说明该函数返回值的类型。如果函数没有返回

值，则函数的返回值类型说明为 void。

(3) <函数名>是用来标识该函数的。函数名的命名应该符合标识符的命名规则，由字母、数字和下划线组成，并且不能以数字开头。

(4) 函数名后边的形式参数列表中给出该函数的各个参数名及参数类型说明。该列表中可以没有参数，也可以有一个参数，还可以有多个参数。当有多个参数时，参数之间用逗号分隔。不管是否有参数，函数名后面的圆括号是不能省略的。

2. 函数体

函数体是一个语句序列，用来描述函数的功能。函数体需要用{}括起来，其中包括声明语句和执行语句。

函数体中一条语句也没有的函数称为空函数。例如：

```
void empty()
{    }
```

空函数被调用时，不执行任何操作。但空函数往往是为以后方便扩展程序的新功能而设计的，在程序设计中也是常常用到的。

需要注意的是，所有函数的定义都是独立的、平行的。也就是说，在一个函数的定义体中，不允许出现另外一个函数的定义体，但可以出现函数的调用语句。

3.2.2 函数的声明

如果函数的调用出现在函数定义之前，那么调用函数之前，需要进行函数的声明。函数的声明就是事先将函数的名字、函数类型以及形参的个数、类型和顺序通知编译系统，以便在编译时可以根据声明的内容对被调用的函数进行检查，保证编译的正常进行。函数声明也被称做函数的原型说明。

标准库函数的函数原型都放在头文件中，程序可以用“#include”预处理指令包含这些原型文件。对于用户自定义的函数，程序员需要在程序中声明函数原型。声明的格式如下。

```
[函数返回值类型标识符] <函数名>(类型标识符[形式参数 1],…);
```

例如，函数 min()的原型声明格式如下。

```
int min(int x,int y);
```

由于在函数声明语句中，形式参数名会被编译系统忽略，所以函数声明语句中可以省略形式参数的名称，只包括形式参数的类型。例如：

```
int min(int ,int);
```

函数声明语句与函数定义非常相似，它们的差别如下。

(1) 函数原型声明没有函数体，而函数定义则必须具有函数体。

(2) 函数原型声明是一条语句，必须用分号结尾。而函数定义则是一个程序模块。

在程序中，先定义而后调用的函数，可以省略函数声明语句；但是，后定义而先调用的函数，在调用前必须说明。通常情况下，为了不考虑函数定义的顺序，在程序的开头将该程序中被调用函数都作说明，这样可以避免一些不必要的错误。

函数声明时要与函数定义时一致，否则将出现错误信息。

【例 3-1】 比较两名学生总成绩的高低。

```
/*3-1.cpp*/
 1. #include <iostream.h>
 2. void main()
 3. {
 4.     int score1,score2,maxScore;
 5.     cout<<"请输入两名学生的成绩："<<endl;
 6.     cin>>score1>>score2;
 7.     int max(int,int);                    /*在 max 函数被调用之前进行函数声明*/
 8.     maxScore=max(score1,score2);         /*函数 max 的调用语句*/
 9.     cout<<maxScore<<endl;
10. }
11. int max(int i,int j)
12. { int m;
13.     m=i>j?i:j;
14.     return m;
15. }
```

【运行结果】

运行结果如图 3-1 所示。

图 3-1 【例 3-1】的运行结果

【分析与思考】

(1) 程序第 7 行为函数 max 的原型声明语句，分号结尾。

(2) 程序第 8 行为函数 max 的首次调用语句，该语句出现在函数定义之前，所以需要在程序中做函数的原型声明。

(3) 程序第 11～15 行为函数 max 的定义部分。

3.2.3 函数的参数

函数的参数包括形式参数与实际参数。

形式参数与实际参数是主调函数与被调函数之间数据流通通道的两个端点，确立了主调函数与被调函数之间的数据传递关系。

函数的形式参数和实际参数是依据它们在程序中出现的位置不同而确定的。在函数的定义语句和函数的声明语句中出现的参数称为形式参数(简称形参)；在函数的调用语句中出现的参数称为实际参数(简称实参)。每个形参的类型可以是任一种数据类型，包括基本类型、指针类型、数组类型、引用类型等；实参可以是常量、变量或表达式。实际参数的定义要求与形式参数的类型兼容。

【例 3-2】 求两个整数的平方和。

```
/*3-2.cpp*/
 1. #include <iostream.h>
 2. void main()
```

```
3. {
4.     int a=3,b=4,c;
5.     int sum(int,int);
6.     c=sum(a,b);
7.     cout<<c <<endl;
8. }
9. int sum(int i,int j)
10. {
11.     return(i * i+j * j);
12. }
```

【运行结果】

运行结果如图 3-2 所示。

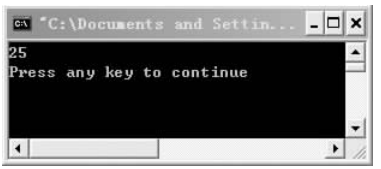

图 3-2 【例 3-2】的运行结果

【分析与思考】

(1) 程序第 5 行是主函数 main 在调用函数 sum 之前对函数 sum 的声明语句,需要有分号结尾。

(2) 程序第 6 行是一个函数调用语句,sum 后括号内的 a 和 b 是实参,它们是 main 函数中定义的整型变量。

(3) 程序中第 9～12 行是函数定义,其中第 9 行函数名 sum 后面括号中指定了两个形参 i、j 及其类型,注意这行的末尾没有分号。

3.2.4 函数的返回值

函数的返回值是函数的运行结果。函数的功能多种多样,有些函数具有返回值,有些函数没有返回值。在带有返回值的函数中,需要使用 return 语句来返回一个表达式的值。如果没有返回值带回给主调函数,则可以省略 return 语句。

return 语句的形式如下。

```
return [(<表达式>)];
```

其中,return 是返回语句的关键字,〈表达式〉的值是要返回的值,圆括号也可以省略。

例如,在【例 3-2】中,函数 sum()中的下述语句:

```
return(i * i+j * j);
```

就是一个带返回值的返回语句。它将 i * i+j * j 的值返回给主调函数。

返回语句 return 也是一条转向语句,它的作用是将语句的执行顺序返回给调用该函数的语句,然后去执行调用函数语句下面的语句。在有返回值时,还需将返回值传递给调用函数。

带有返回值的 return 语句的实现机制如下。

(1) 执行带有返回值的 return 语句时,先计算 return 语句后面的表达式的值。

(2) 根据该函数的类型来确定表达式的类型,如果表达式的类型与函数的类型不一致,则强行将表达式类型转换为函数类型,这里可能出现精度受损失的问题。

(3) 将表达式值作为函数的返回值传递给调用函数。

(4) 将程序的执行流程返回到调用函数语句,接着执行调用函数语句下面的语句。

如果没有返回值，return 语句执行起来比较简单，只返回语句执行的控制权。格式如下。

```
return;
```

在一个函数中，如果不出现 return 语句，则该函数是无返回值的，并且执行完所有函数体内的语句时，将自动返回主调函数。实际上，函数体定界符的右花括号也具有 return 功能。一个函数中也可出现多个返回语句，执行到哪一个 return 语句，哪一个语句起作用。这种情况多出现在选择结构中。

【例 3-3】 无返回值的函数举例。

```
/*3-3.cpp*/
 1. #include <iostream.h>
 2. void main()
 3. {
 4.     void coutline();
 5.     coutline();
 6.     cout<<" 欢迎登录学生选课系统 "<<endl;
 7.     coutline();
 8. }
 9. void coutline(int i,int j)
10. {
11.     printf("************************");
12. }
```

【运行结果】

运行结果如图 3-3 所示。

图 3-3 【例 3-3】的运行结果

【分析与思考】

(1) 程序第 4 行是无返回值函数 coutline 的函数声明语句。

(2) 程序第 5～7 行是函数 coutline 的调用语句。

(3) 程序第 9～12 行是函数 coutline 的定义部分。

【例 3-4】 返回值的类型与函数类型不同的函数举例。

```
/*3-4.cpp*/
 1. #include <iostream.h>
 2. void main()
 3. {
 4.     float a=3.2,b=4.4,c;
 5.     int sum(float,float);
 6.     c=sum(a,b);
 7.     cout<<c <<endl;
 8. }
 9. int sum(float i,float j)
10. {
11.     return(i*i+j*j);
12. }
```

【运行结果】

运行结果如图 3-4 所示。

图 3-4 【例 3-4】的运行结果

【分析与思考】

当 return 语句中表达式值的类型与函数值的定义类型不一致时，系统会以函数类型为准，自动进行类型转换。

3.2.5 函数的调用

1. 函数的调用方式

函数只有被调用才会发挥作用。在 C++ 中，除了主函数 main 由系统自动调用外，其他函数都是由主调函数直接或间接调用的。函数调用的语法格式如下。

```
<函数名>([实参表])
```

如果是调用无参函数，则“实参表”可以没有，但括号不能省略。如果实参表由多个实参组成，则参数间要用逗号隔开，实参和形参个数相等，类型匹配兼容。

常见的函数调用方式有两种，一种是函数语句，另一种是函数表达式。

(1) 函数语句

将函数调用作为一条独立语句使用，只要求函数完成一定的操作，而不使用其返回值。若函数调用带有返回值，则这个值将会自动丢失。例如定义如下函数：

```
void func()
{
    cout<<"\n)";
}
```

以一条独立语句调用该函数如下：

```
void main()
{
  func();
  cout<<"C++Program."
  func();
}
```

(2) 函数表达式。

对于具有返回值的函数来说，把函数调用语句看做语句的一部分，使用函数的返回值参与相应的运算或执行相应的操作。例如，在【例 3-2】的主函数中，下列语句

```
c=sum(a,b);
```

就是对函数 sum()的表达式调用形式。

2. 函数的嵌套调用

在一个函数的调用过程中又出现了另外一个函数的调用语句，这种现象称做函数的嵌套调用。例如主函数 main 调用了函数 f1()，函数 f1()中又调用了函数 f2()。这称

为函数的嵌套调用。嵌套层数理论上没有限制，函数 f2()可以继续调用其他函数 f3()等。

【例 3-5】 求两整数的平方差。

```
/*3-5.cpp*/
#include <iostream.h>
int func2 (int x)
{
    return x*x;
}
int func1 (int m,int n)
{
    return func2(m)-func2(n);        /*函数 func1 嵌套调用函数 func2*/
}
void main()
{
    int a,b;
    cout<<"Please enter two integers:";
    cin>>a>>b;
    cout<<"The result is:"<<func1(a,b)<<endl;
}
```

【运行结果】

运行结果如图 3-5 所示。

【分析与思考】

如图 3-6 所示，图 3-6 中方向线表示程序的执行顺序，按照方向线上数字的递增顺序依次执行。

图 3-5 【例 3-5】的运行结果

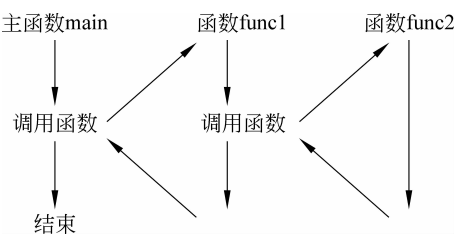

图 3-6 【例 3-5】函数的调用过程

3. 函数的递归调用

在 C++ 语言程序中，主函数可以调用其他任何函数，任意函数又可以调用除主函数之外的其他任何函数。较为特殊的一种情况，是一个函数还可以直接或间接地调用它自己本身，这种情况称为递归调用。所谓直接递归，是指在一个函数体中使用调用本函数的函数调用表达式。而间接递归是指在一个函数中调用另一个函数，而在另一个函数中又反过来调用这个函数。这里举一个直接递归调用的例子。

【例 3-6】 利用递归方法求解 n 的阶乘(n!)的值。

```
/*3-6.cpp*/
#include <iostream.h>
int f (int n)
{
    if(n==0||n==1)
        return 1;
    else
        return n*f(n-1);          /*函数 f 调用自身,即递归调用*/
}
void main()
{
    int x,res;
    cout<<"Please enter x:";
    cin>>x;
    res=f(x);
    cout<<"The result is:"<<res<<endl;
}
```

图 3-7 【例 3-6】的运行结果

【运行结果】

运行结果如图 3-7 所示。

【分析与思考】

设用函数 f(n)表示 n!。由数学知识可知,n 阶乘的递归定义为:它等于 n 乘以 n－1 的阶乘。当 n 等于 0 或 1 时,函数值为 1,用数学公式表示为:

$$f(n)=\begin{cases}1 & (n=0 \text{ 或 } n=1)\\ n*f(n-1) & (n>1)\end{cases}$$

在这里,n 等于 0 或 1 是递归终止的条件,得到的函数值为 1。当 n 大于 1 时,需要向下递归,先求出 f(n－1)的值后,再乘以 n,才能得到 f(n)的值。计算 f(n)的递归函数如下。

```
int f(int n)
{
    if(n==0 || n==1) return 1;
    else return n*f(n-1);
}
```

假定用 f(5)去调用 f(n)函数,该函数返回 5*f(4)的值,因返回表达式中包含有函数 f(4)表达式,所以接着进行递归调用,返回 4*f(3)的值。依此类推,当最后进行 f(1)递归调用,返回函数值 1 后,结束本次递归调用,返回到调用函数 f(1)的位置,从而计算出 2*f(1)的值 2,即 2*f(1)＝2*1＝2,作为调用函数 f(2)的返回值,返回到 3*f(2)表达式中,计算出值 6,作为 f(3)函数的返回值,接着返回到 4*(3)表达式中,计算出值 24,作为 f(4)函数的返回值,再接着返回到 5*f(4)表达式中,计算出 f(5)的返回值 120,从而结束整个调用过程,返回到调用函数 f(5)的位置,继续向下执行。

3.3 函数的参数传递

主调函数和被调函数之间交换信息的方法主要有两种。一种是由被调函数把返回值返回给主调函数，另一种是通过参数交换信息。用参数交换信息的方法有三种，传值调用、传地址调用和引用参数。前两种是从 C 语言继承过来的，本节作为回顾作一介绍，后一种将在下节介绍。

3.3.1 传值调用

传值调用要求实参是表达式，形参是变量。当函数调用发生时，系统才为形参变量分配存储单元，然后主调函数将实参的值对应地传递给形参变量。调用期间实参变量和对应的形参变量占用的是各自不同的存储单元，形参变量从对应实参变量获得初始值后，若被调函数执行过程中改变了形参变量的值，不会影响形参所对应的实参变量的值。这一传递过程是单向的值传递过程，也就是说，在被调函数中改变形参的值不会同时改变实参的值。被调函数只能通过 return 语句来给主调函数传递返回值。

实参变量和形参变量可使用相同的名字，但它们是同名的不同变量。

【例 3-7】 分析下列程序的输出结果，观察形参的改变是否影响实参。

```
/* 3-7.cpp */
#include <iostream.h>
void swap(int a, int b);              /* swap 函数声明语句 */
void main()
{
    int x,y;
    cin>>x>>y;
    swap(x,y);
    cout<<" x="<<x<<", y="<<y<<endl;
}
void swap(int a, int b)
{
    int t;
    t=a;
    a=b;
    b=t;
    cout<<"a="<<a<<", b="<<b<<endl;
}
```

【运行结果】

运行结果如图 3-8 所示。

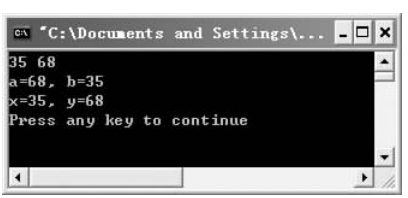

图 3-8 【例 3-7】的运行结果

【分析与思考】

从上面程序可以看出，在函数 swap() 被调用执行时，编译系统为形参 a 和 b 分配存储单元。然后，实参 x、y 的值赋给相应的形参 a、b。函

数调用结束后，形参 a、b 的值已经改变。但返回调用函数 main()后，实参 x、y 的值没有改变。函数 swap()在被调用过程中实参与形参的变化情况如图 3-9 所示。

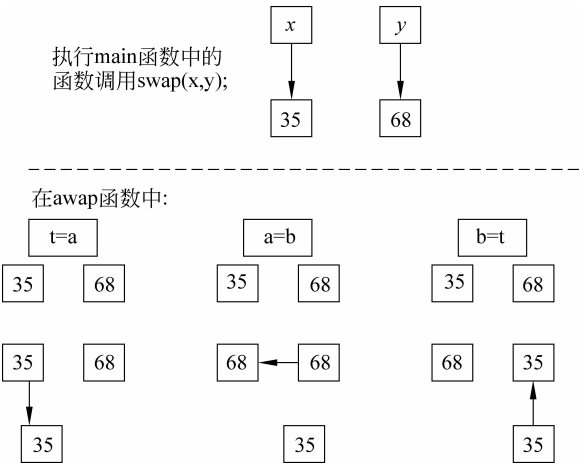

图 3-9 【例 3-7】执行时形参与实参的变化情况

3.3.2 传地址调用

传地址调用方式要求主调函数的实参用变量地址，被调函数的对应形参用指针变量。指针是一种用来存放某个变量地址值的变量。一个指针存放了哪个变量的地址值，该指针就指向哪个变量。

传址调用方式中，将实参变量的地址值传送给形参指针后，就使形参指针指向了该实参变量。因此，传址调用方式不是把实参值传送给形参，而是使形参指针直接指向实参变量。

【例 3-8】 分析下面程序中数据传送的特点。

```
/*3-8.cpp*/
#include <iostream.h>
void swap(int *a, int *b);                    /*形参为整型指针*/
void main()
{
    int x,y;
    cin>>x>>y;
    swap(&x,&y);                              /*实参为整型变量地址*/
    cout<<"x="<<x<<", y="<<y<<endl;
}
void swap(int *a, int *b)
{
    int t;
    t=*a;
    *a=*b;
    *b=t;
    cout<<"*a="<<*a<<", *b="<<*b<<endl;
}
```

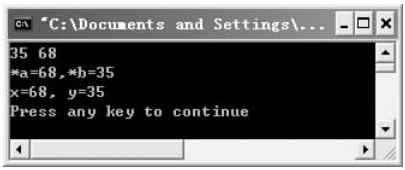

图 3-10 【例 3-8】的运行结果

【运行结果】

运行结果如图 3-10 所示。

【分析与思考】

(1) 从上面程序可以看出,被调函数 swap 执行后,实参 x、y 的值已经交换。

(2) 传址调用中可以通过形参指针来改变主调函数中的实参变量值,以达到传出数据的目的。

3.4 引用在函数中的应用

3.4.1 引用作为函数参数

引用调用是指将被调函数的形参定义为引用变量,主调函数里的实参用变量名。函数调用发生时,用实参来初始化形参,这样引用类型的形参就通过形参与实参的结合成为了实参变量的别名。因此,被调函数执行过程中对形参的改变,实质上就是直接通过引用即实参变量的别名来改变实参变量的值。

用引用类型作为形参的函数调用,称为引用调用。在函数调用语句中,实参只能是与形参引用同类型的变量名,不能是指针变量名、数组名、表达式、数字常量和符号常量等。

【例 3-9】 交换两个整数。

```
/*3-9.cpp*/
#include <iostream.h>
void swap(int &a,int &b)                    /*形参为整型引用变量*/
{
    int t;
    t=a;
    a=b;
    b=t;
    return;
}
void main()
{
    int x,y;
    cin>>x>>y;
    cout<<"Before swapping:x="<<x<<",y="<<y<<endl;
    swap(x,y);
    cout<<"After swapping:x="<<x<<",y="<<y<<endl;
}
```

【运行结果】

运行结果如图 3-11 所示。

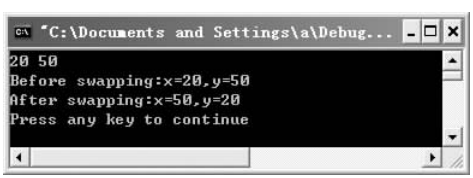

图 3-11 【例 3-9】的运行结果

【分析与思考】

该程序采用了引用调用的参数传递方式,

用实际参数 x 和 y，对形参引用进行初始化，函数体中形参 a、b 成为实参 x 和 y 的别名。也就是说，形参 a、b 不占用临时存储空间，它们分别对应实参 x 和 y 的存储单元。所以，对形式参数 a、b 的修改即是对实际参数 x 和 y 的修改。

3.4.2 引用作为函数返回值

在 C 语言中，函数返回值常常放在赋值号右边，作为右值表达式，或作其他用途，但不能放在赋值号左边被再赋值。在 C++ 中，如果函数返回值为引用类型，它可以成为左值表达式，被再赋值。原因是这时函数返回的不是某个值，而是某个变量的引用。

【例 3-10】 作为返回值的引用举例。

```
/*3-10.cpp*/
#include <iostream.h>
int& max(int& a,int& b)                    /*函数返回值定义为引用类型*/
{
    return(a>b?a:b);
}
void main()
{
    int i=12,j=34;
    cout<<"i="<<i<<",j="<<j<<endl;
    cout<<"The larger is "<<max(i,j)<<endl;
    max(i,j)=0;                            /*函数返回值作为赋值号左值*/
    cout<<"i="<<i<<",j="<<j<<endl;
}
```

【运行结果】

运行结果如图 3-12 所示。

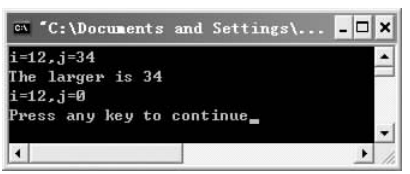

图 3-12 【例 3-10】的运行结果

【分析与思考】

主函数 main()对函数 max()进行了两次调用。第一次调用计算出了 i 和 j 的最大值。第二次调用使用的语句是 max(i,j)＝0，此次调用返回的是 j 变量的引用，因此语句 max(i,j)＝0 实际上是对 j 进行赋值为 0 的操作。

3.5 默认参数的函数

一般情况下，实参个数应与形参个数相同。但是在 C++ 中，允许实参个数与形参个数不同，方法是在形参列表中对一个或几个形参指定缺省值(或称默认值)。像这样在函数说明或定义中为形参指定默认值的函数，称为缺省参数的函数或带默认形参值的函数。

函数调用时，如果指定了形式参数所对应的实际参数，则形式参数使用实际参数的值。如果未指定相应的实际参数，则形式参数使用缺省值。

当函数有多个形式参数时，缺省参数必须从右向左依次定义，即在一个缺省参数的右边不能还有未指定的缺省值的参数。例如函数原型：

```
int fun1(int a=4,char b,int c=7);
```

上述缺省参数定义是不合法的，因为在缺省参数 a 的右边存在未指定缺省值的参数 b。

需要注意的是，如果在函数的声明语句中设置了形参的默认值，那么在函数定义的头部就不能够再重复设置了，否则编译时将出现错误信息。

【例 3-11】 求两点之间的距离。

```
/*3-11.cpp*/
 1. #include <iostream.h>
 2. #include <math.h>
 3. double distance(double x1,double y1,double x2=0.0,double y2=0.0);
 4. int main()
 5. {
 6.     cout<<"(3,4)--->(0,0)distance:"<<distance(3,4)<<endl;
 7.     cout<<"(4,3)--->(0,0)distance:"<<distance(4,3)<<endl;
 8.     cout<<"(4,3)--->(1,-1)distance:"<<distance(4,3,1,-1)<<endl;
 9.     return 0;
10. }
11. double distance(double x1,double y1,double x2,double y2)
12. {
13.     double x,y;
14.     x=x2-x1;
15.     y=y2-y1;
16.     return sqrt((x*x)+(y*y));
17. }
```

【运行结果】

运行结果如图 3-13 所示。

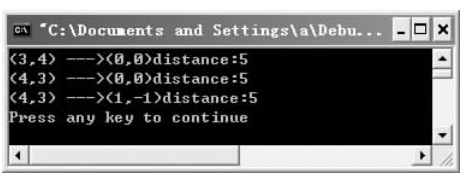

图 3-13 【例 3-11】的运行结果

【分析与思考】

(1) 程序第 3 行声明 distance()函数，带有两个默认形参值，distance()成为缺省参数的函数。

(2) 程序第 6 行和第 7 行两次调用函数 distance()，并采用默认形参值。

(3) 程序第 8 行调用函数 distance()，调用语句提供 4 个实参值，默认形参值不被采用。

(4) 程序第 11～17 行是缺省参数的函数 distance()的定义部分，函数头部不再重复设置形式参数的默认值。

从以上例子可知，当需要多次调用一个函数，并且多数情况下是给这个函数传递某个同样的参数值时，可以考虑用缺省参数。如在上例中，distance()函数用于计算两点之间的距离，但在多数情况下可能需要计算某点到原点的距离，所以设置 distance()函数中的后两个参数缺省为 0。

3.6 内联函数

虽然函数的使用有利于代码重用,便于程序的修改和维护,可以提高开发效率,但函数调用也会降低函数的执行效率,增加时间和空间上的耗费。因为调用函数时,系统会将程序执行流程转移到函数体在内存中的存放地址,将函数体内容执行完闭后,再返回调用函数继续执行。这种转移操作要求在流程转移前要保护现场,并记忆执行的地址,返回后先要恢复现场,并按原来保存地址继续执行。因此,函数调用要有一定时间和空间方面的开销,会导致时间效率与空间效率的降低。特别是对于函数体代码不是很大,但被调用次数较为频繁的函数来讲,解决其效率问题更为重要。

内联函数可以很好地解决上述问题。内联函数不是在函数调用时发生控制转移,而是在程序编译时,编译器将程序中出现的内联函数的调用表达式用内联函数的函数体来代替。显然,这种做法不会产生由于函数调用而带来的时间与空间效率降低的问题。但是由于在编译时将函数体中的代码替代到程序中,因此会增加目标程序代码量,进而增加空间开销,而在时间开销上不像函数调用时那么大,可见它是以目标代码的增加为代价换取时间的节省。

定义内联函数的方法很简单。只要在函数定义的头部前面加上关键字 inline 即可,其他与函数定义相同。内联函数定义的格式如下。

```
inline 类型 函数名(参数表)
{
    函数体
}
```

与其他函数一样,内联函数必须先声明后使用。如果要说明一个内联函数原型,则也必须加上声明关键字 inline。例如:

```
inline int add_int(int x,int y,int z);              //函数原型
```

【例 3-12】 内联函数举例。

```
/*3-12.cpp*/
#include <iostream.h>
inline int power_int(int x);
void main()
{
    for (int i=1;i<=10;i++)
    {
        int p=power_int(i);
        cout<<i<<'*'<<i<<'='<<p<<endl;
    }
}
inline int power_int(int x)
{
```

```
13.     return (x) * (x);
14. }
```

【运行结果】

运行结果如图 3-14 所示。

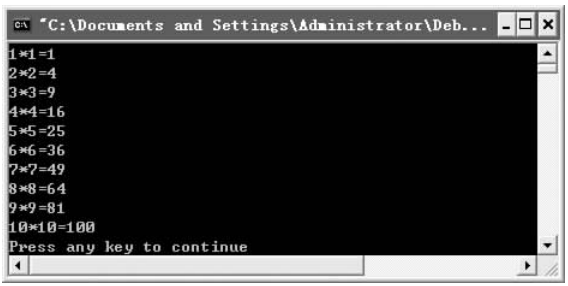

图 3-14 【例 3-12】的运行结果

【分析与思考】

(1) 程序第 2 行是内联函数 power_int()声明语句。程序第 11～14 行是内联函数 power_int()的定义体。这两部分都要使用 inline 关键字。

(2) 函数 power_int()在程序中被大量调用,而其代码又比较短,将其定义为内联函数,大大提高了程序的效率。

在对内联函数的实际处理过程中,C++ 编译程序根据内联函数的具体情况决定是否对其使用内联功能。如果函数的代码很长很复杂,函数体内出现循环语句或开关语句,或者出现了将一个递归函数用做内联函数的情况,编译程序会把这个定义成内联的函数作为一般函数来处理。

3.7 重 载 函 数

用 C 语言编程时,例如要从 3 个数中找出其中最大者,而这 3 个数的类型事先不确定,可以是整数、实数或长整数。程序设计者必须分别设计出 3 个函数,其原型如下。

```
int max1(int a,int b,int c);           (求 3 个整数中的最大者)
float max2(float a,float b,float c);   (求 3 个实数中的最大者)
long max3(long a,long b,long c);       (求 3 个长整数中的最大者)
```

C 语言规定在同一作用域中不能有同名的函数,因此 3 个函数的名字各不相同。

C++ 允许在同一作用域中用同一函数名定义多个函数,要求这些函数的参数个数或参数类型不同。这样的函数被称做重载函数。对于上述问题,可以声明重载函数如下。

```
int max(int a,int b,int c);            (求 3 个整数中的最大者)
float max(float a,float b,float c);    (求 3 个实数中的最大者)
long max(long a,long b,long c);        (求 3 个长整数中的最大者)
```

重载函数要求当函数调用发生时,编译器能够唯一地确定调用应执行哪个函数代码,即采用哪个函数实现。编译系统根据实际参数与形式参数的最佳匹配原则确定函数实现。也

就是说，进行函数重载时，要求同名函数的参数类型或个数必须不同，或者两者都不同。否则将无法实现重载。

需要注意的是，如果同名函数仅仅是返回值的类型不相同，而参数的个数和类型都相同，那么后面的函数会被看做是第一个函数的错误重复声明。例如：

```
int fun(int,int);
float fun(int,int);                    //fun 函数的错误重复声明
```

如果同名函数的形参列表中只有缺省参数不同，那么后一个函数会被看做是第一个同名函数的重复声明。例如：

```
int fun(int x,int y);
int fun(int x,int y=10);               //fun 函数重复声明错误
```

下面举一个在参数类型上不同的重载函数的例子。

【例 3-13】 求 3 个数中最大的数(分别考虑整数、实数、长整数的情况)。

```
/*3-13.cpp*/
#include <iostream.h>
int max(int a,int b,int c)             //求 3 个整数中的最大者
{
    if (b>a) a=b;
    if (c>a) a=c;
    return a;
}
float max(float a,float b,float c)     //求 3 个实数中的最大者
{
    if (b>a) a=b;
    if (c>a) a=c;
    return a;
}
long max(long a,long b,long c)         //求 3 个长整数中的最大者
{
    if (b>a) a=b;
    if (c>a) a=c;
    return a;
}
void main()
{
    int a,b,c;
    float d,e,f;
    long g,h,i;
    cin>>a>>b>>c;
    cin>>d>>e>>f;
    cin>>g>>h>>i;
    int m;
    m=max(a,b,c);                      //函数值为整型
```

```
30.     cout <<"max-int="<<m<<endl;
31.     float n;
32.     n=max(d,e,f);                          //函数值为实型
33.     cout<<"max-float="<<n<<endl;
34.     long int p;
35.     p=max(g,h,i);                          //函数值为长整型
36.     cout<<"max-long="<<p<<endl;
37. }
```

【运行结果】

运行结果如图 3-15 所示。

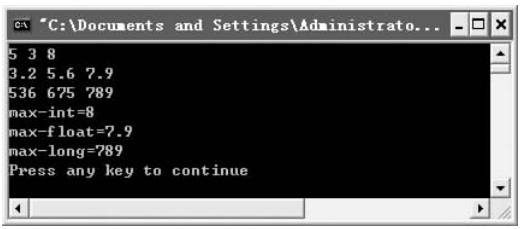

图 3-15 【例 3-13】的运行结果

【分析与思考】

(1) 参数类型不同的 3 个 max()函数为重载函数。

(2) 程序第 29 行、第 32 行和第 35 行 3 次调用 max 函数,每次调用实参的类型不同。系统会根据实参的类型找到与之最佳匹配的 max 函数,然后调用该函数。

下面再举一个在参数个数上不相同的函数重载的例子。

【例 3-14】 请按照要求输出不同形式的个人参加工作的时间。

```
/*3-14.cpp*/
 1. #include <iostream.h>
 2. void workdate(int a, int b);
 3. void workdate(int a, int b, int c);
 4. void main()
 5. {
 6.     int y,m,d;
 7.     cout<<"请输入个人参加工作年份:"<<endl;
 8.     cin>>y;
 9.     cout<<"请输入个人参加工作月份:"<<endl;
10.     cin>>m;
11.     cout<<"请输入个人参加工作日:"<<endl;
12.     cin>>d;
13.     cout<<"您参加工作的时间是:"<<endl;
14.     workdate(y,m);
15.     cout<<"您参加工作的具体时间是:"<<endl;
16.     workdate(y,m,d);
17. }
```

```
18. void workdate(int a, int b)
19. {
20.     cout<<a<<"年"<<b<<"月"<<endl;
21. }
22. void workdate(int a, int b, int c)
23. {
24.     cout<<a<<"年"<<b<<"月"<<c<<"日"<<endl;
25. }
```

【运行结果】

运行结果如图 3-16 所示。

图 3-16 【例 3-14】的运行结果

【分析与思考】

该程序中出现了函数重载,函数名 workdate 对应有两个不同的实现,函数区分的依据是参数个数不相同。在这两个重载函数中,参数个数分别为 2 个和 3 个,调用函数时,根据实参的个数来选取不同的函数实现。

需要说明的是,函数重载的参数个数或类型必须至少有一个不同,函数返回值类型可以相同,也可以不同。但不允许参数个数和类型都相同而只有返回值类型不同,因为系统无法从函数的调用形式上判断哪一个函数与之匹配。

3.8 变量的作用域与存储类别

3.8.1 变量的作用域

变量的有效范围称为变量的作用域。变量的作用域有 4 种类别:全局作用域、文件作用域、函数作用域和块作用域。具有全局作用域的变量称为全局变量,具有函数作用域和块作用域的变量称为局部变量。

1. 全局作用域

当一个变量在一个程序文件的所有函数定义之外(并且通常在所有函数定义之前)定义时,则该变量具有全局作用域,即该变量在整个程序包括的所有文件中都有效,都是可见的,都是可以访问的。当一个全局变量不是在本程序文件中定义时,若要在本程序文件中使用,则必须在本文件开始时进行声明,声明格式如下。

```
extern <类型名><变量名>, <变量名>,...;
```

它与变量定义语句格式类似,其区别是:不能对变量进行初始化,并且要在整个语句前加上 extern 保留字。

当用户定义一个全局变量时,若没有对其初始化,则编译时会自动把它初始化为 0。

2. 文件作用域

当一个变量定义语句出现在一个程序文件中的所有函数定义之外,并且该语句前带有 static 保留字时,则该语句定义的所有变量都具有文件作用域,即在整个程序文件中有效,但在其他文件中是无效的,不可见的。

若在定义文件作用域变量时没有初始化,则编译时会自动把它初始化为 0。

3. 函数作用域

当变量在函数内部被定义时,则具有函数作用域,这些变量的作用域在它所定义的函数体内,从定义语句开始,到函数体结束为止。

4. 块作用域

当一个变量在复合语句内定义时,则称它具有块作用域,其作用域范围是从定义语句开始,直到该复合语句结束(即复合语句的右花括号)为止。

具有函数作用域和块作用域的变量称为局部变量,若局部变量没有被初始化,则系统也不会对它初始化,它的初值是不确定的。对于在函数体中使用的变量定义语句,若在其前面加上 static 保留字,则称所定义的变量为静态局部变量,若静态局部变量没有被初始化,则编译时会被自动初始化为 0。

对于非静态局部变量,每次执行到它的定义语句时,都会为它分配对应的存储空间,并对带初值表达式的变量进行初始化;而对于静态局部变量,只是在整个程序执行过程中第一次执行到它的定义语句时为其分配对应的存储空间,并进行初始化,以后再执行到它时,什么都不会做,相当于第一次执行后就删除了该语句。

在 C++ 程序中定义的符号常量也同变量一样,具有全局、文件和局部这 3 种作用域。当符号常量定义语句出现在所有函数定义之外,并在前面带有 extern 保留字时,则所定义的常量具有全局作用域,若在前面带有 static 关键字,或什么都没有,则所定义的常量具有文件作用域。若符号常量定义语句出现在一个函数体内,则定义的符号常量具有局部作用域。

具有同一作用域的任何标识符,不管它表示什么对象(如常量、变量、函数、类型等),都不允许重名,若重名,则系统就无法唯一确定它的含义了。

由于每一个复合语句就是一个块,所以在不同复合语句中定义的对象具有不同的块作用域,也称为具有不同的作用域,其对象名允许重名,因为系统能够区分它们。

【例 3-15】 局部变量的作用域举例。

```
/*3-15.cpp*/
 1. #include <iostream.h>
 2. void main()
 3. {
 4.     int a=15;          //a 的作用域在整个 main()函数内部
 5.     if(a<50)
```

```
 6.     {
 7.         int b=20;       //b 的作用域在 if 语句块内
 8.         cout<<"a="<<a<<"b="<<b<<endl;
 9.         a=b/2;
10.     }
11.     b=100;              //错误!超出了 b 的作用域
12.     cout<<"a="<<a<<"b="<<b<<endl;
13.     a=c*2;              //错误!超出了 c 的作用域
14.     int c=10;           //c 的作用域从定义它的位置开始一直到 main()函数结束
15.     cout<<"a="<<a<<"b="<<b<<"c="<<c<<endl;
16. }
```

【分析与思考】

变量 a 的作用域在整个 main()函数体内,可以被函数体内的所有语句使用;变量 b 的作用域在 if 语句块内,离开这个语句块,变量 b 就不能够再被使用;变量 c 的作用域从定义它的位置开始一直到 main()函数结束,在 c 变量被定义之前的任何位置是不能使用它的。

【例 3-16】 全局变量的作用域举例。

```
/*3-16.cpp*/
 1. #include <iostream.h>
 2. void fun();
 3. int x,y;
 4. void main()
 5. {
 6.     cin>>x>>y;
 7.     cout<<"x="<<x<<"y="<<y<<endl;
 8.     fun();
 9.     cout<<"x="<<x<<",y="<<y<<endl;
10. }
11. void fun()
12. {
13.     x=x+10;
14.     y=y+10;
15.     cout<<"x="<<x<<",y="<<y<<endl;
16. }
```

【运行结果】

运行结果如图 3-17 所示。

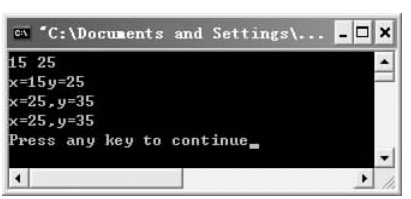

图 3-17 【例 3-16】的运行结果

【分析与思考】

x、y 在函数外声明,x、y 为全局变量,在 main()函数和 fun()函数中都可以访问。在 fun()函数中引起的全局变量的变化能够在 fun()函数运行结束后传递给 main()函数。