第 3 章

函　数

C++ 语言继承了 C 语言的全部语法，也包括函数的定义与使用方法。在面向过程的结构化程序设计中，函数是模块划分的基本单位，是对处理问题过程的一种抽象。在面向对象的程序设计中，函数同样有着重要的作用，它是面向对象程序设计中对功能的抽象。

一个较为复杂的系统往往需要划分为若干子系统，然后对这些子系统分别进行开发和调试。高级语言中的子程序就是用来实现这种模块划分的。C 和 C++ 语言中的子程序体现为函数。通常将相对独立的、经常使用的功能抽象为函数。函数编写好以后，可以被重复使用，使用时可以只关心函数的功能和使用方法而不必关心函数功能的具体实现。这样有利于代码重用，可以提高开发效率、增强程序的可靠性，也便于分工合作和修改维护。

3.1　函数的定义与使用

第 2 章例题中出现的 main 就是一个函数，它是 C++ 程序的主函数。一个 C++ 程序可以由一个主函数和若干子函数构成。主函数是程序执行的开始点。由主函数调用子函数，子函数还可以再调用其他子函数。

调用其他函数的被称为**主调函数**，被其他函数调用的称为被**调函数**。一个函数很可能既调用别的函数又被另外的函数调用，这样它可能在某一个调用与被调用关系中充当主调函数，而在另一个调用与被调用关系中充当被调函数。

3.1.1　函数的定义

1. 函数定义的语法形式

```
类型说明符　函数名(含类型说明的形式参数表)
{
    语句序列
}
```

2. 形式参数

形式参数(简称形参)表的内容如下：

```
type1 name1, type2 name2, …, typen namen
```

type1、type2、……、typen 是类型标识符，表示形参的类型。name1、name2、……、namen 是形参名。形参的作用是实现主调函数与被调函数之间的联系，通常将函数所处理的数据、影响函数功能的因素或者函数处理的结果作为形参。

如果一个函数的形参表为空，则表示它没有任何形参，例如第 2 章例题中的 main 函数都没有形参。main 函数也可以有形参，其形参也称命令行参数，由操作系统在启动程序时初始化。不过命令行参数的数量和类型有特殊要求，请读者参考学生用书中本章的实验指导，尝试编写带命令行参数的程序。

函数在没有被调用时是静止的，此时的形参只是一个符号，它标志着在形参出现的位置应该有一个什么类型的数据。函数在被调用时才执行，也是在被调用时才由主调函数将实际参数（简称实参）赋予形参。这与数学中的函数概念相似，例如在数学中我们都熟悉这样的函数形式：

$$f(x)=x^2+x+1$$

这样的函数只有当自变量被赋值以后，才能计算出函数的值。

3. 函数的返回值和返回值类型

函数可以有一个返回值，函数的返回值是需要返回给主调函数的处理结果。类型说明符规定了函数返回值的类型。函数的返回值由 return 语句给出，格式如下：

```
return 表达式;
```

除了指定函数的返回值外，return 语句还有一个作用，就是结束当前函数的执行。

例如，主函数 main 的返回值类型是 int，主函数中的 return 0 语句用来将 0 作为返回值，并且结束 main 函数的执行。main 函数的返回值最终传递给操作系统。

一个函数也可以不将任何值返回给主调函数，这时它的类型标识符为 void，可以不写 return 语句，但也可以写一个不带表达式的 return 语句，用于结束当前函数的调用，格式如下：

```
return;
```

3.1.2　函数的调用

1. 函数的调用形式

在 2.2.3 小节曾经提到过，变量在使用之前需要首先声明，类似地，函数在调用之前也需要声明。函数的定义就属于函数的声明，因此，在定义了一个函数之后，可以直接调用这个函数。但如果希望在定义一个函数前调用它，则需要在调用函数之前添加该函数的**函数原型**声明。函数原型声明的形式如下：

```
类型说明符　函数名（含类型说明的形参表）;
```

与变量的声明和定义类似，声明一个函数只是将函数的有关信息（函数名、参数表、返回值类型等）告诉编译器，此时并不产生任何代码；定义一个函数时除了同样要给出函数的有关信息外，主要是要写出函数的代码。2.5 节讲解过使用 decltype 来获取某个变量或表达式的类型，对于函数返回值的使用方法类似，以简化函数返回值类型定义：

```
int a=10, b=5;
decltype(a) myMax(decltype(a) lhs, decltype(a) rhs){    //返回值类型与 a 保持一致
    return lhs>rhs? lhs:rhs;
}
```

如上定义了函数返回值和形参类型与变量 a 类型一致的取最大值函数。

细节 声明函数时,形参表只要包含完整的类型信息即可,形参名可以省略,也就是说,原型声明的形参表可以按照下面的格式书写:

```
type1, type2, ..., typen
```

但这并不是值得推荐的写法,因为形参名可以向编程者提示每个参数的含义。

如果是在所有函数之前声明了函数原型,那么该函数原型在本程序文件中任何地方都有效。也就是说,在本程序文件中任何地方都可以依照该原型调用相应的函数。如果是在某个主调函数内部声明了被调函数原型,那么该原型就只能在这个函数内部有效。

声明了函数原型之后,便可以按如下形式调用子函数:

```
函数名(实参列表)
```

实参列表中应给出与函数原型形参个数相同、类型相符的实参,每个实参都是一个表达式。函数调用可以作为一条语句,这时函数可以没有返回值。函数调用也可以出现在表达式中,这时就必须有一个明确的返回值。

调用一个函数时,首先计算函数的实参列表中各个表达式的值,然后主调函数暂停执行,开始执行被调函数,被调函数中形参的初值就是主调函数中实参表达式的求值结果。当被调函数执行到 return 语句,或执行到函数末尾时,被调函数执行完毕,继续执行主调函数。

例 3-1 编写一个求 x 的 n 次方的函数。

```
//3_1.cpp
#include <iostream>
using namespace std;

//计算 x 的 n 次方
double power(double x, int n) {
    double val=1.0;
    while (n--)
        val *=x;
    return val;
}

int main() {
    cout<<"5 to the power 2 is "<<power(5, 2)<<endl;
    //函数调用作为一个表达式出现在输出语句中
    return 0;
}
```

运行结果:

```
5 to the power 2 is 25
```

本程序中,由于函数 power 的定义位于调用之前,所以无须再对函数原型加以声明。

例 3-2　输入一个 8 位二进制数，将其转换为十进制数输出。

分析：将二进制转换为十进制，只要将二进制数的每一位乘以该位的权然后相加。例如：$(00001101)_2=0\times(2^7)+0\times(2^6)+0\times(2^5)+0\times(2^4)+1\times(2^3)+1\times(2^2)+0\times(2^1)+1\times(2^0)=(13)_{10}$，所以，如果输入 1101，则应输出 13。

这里我们调用例 3-1 中的函数 power 来求 2^n。

源程序：

```
//3_2.cpp
#include <iostream>
using namespace std;

//计算 x 的 n 次方
double power(double x, int n);

int main() {
    int value=0;

    cout<<"Enter an 8 bit binary number: ";
    for (int i=7; i>=0; i--) {
        char ch;
        cin>>ch;
        if (ch=='1')
            value+=static_cast<int>(power(2, i));
    }
    cout<<"Decimal value is "<<value<<endl;
    return 0;
}

double power (double x, int n) {
    double val=1.0;
    while (n--)
        val *=x;
    return val;
}
```

运行结果：

```
Enter an 8 bit binary number: 01101001
Decimal value is 105
```

本程序中，由于 power 函数的定义位于它的调用之后，因此要事先声明 power 函数的原型。

例 3-3　编写程序求 π 的值，公式如下：

$$\pi=16\arctan\left(\frac{1}{5}\right)-4\arctan\left(\frac{1}{239}\right)$$

其中 arctan 用如下形式的级数计算：

$$\arctan x = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \cdots$$

直到级数某项绝对值不大于 10^{-15} 为止；π 和 x 均为 double 型。

源程序：

```
//3_3.cpp
#include <iostream>
using namespace std;

double arctan(double x) {
    double sqr=x*x;
    double e=x;
    double r=0;
    int i=1;
    while (e/i>1e-15) {
        double f=e/i;
        r=(i%4==1) ? r+f : r-f;
        e=e*sqr;
        i+=2;
    }
    return r;
}

int main() {
    double a=16.0*arctan(1/5.0);
    double b=4.0*arctan(1/239.0);
    //注意：因为整数相除结果取整,如果参数写 1/5,1/239,结果就都是 0
    cout<<"PI="<<a-b<<endl;
    return 0;
}
```

运行结果：

```
PI=3.14159
```

例 3-4 寻找并输出 11～999 的数 m，它满足 m、m^2 和 m^3 均为回文数。

所谓回文数是指其各位数字左右对称的整数。例如：121、676、94249 等。满足上述条件的数如 $m=11$，$m^2=121$，$m^3=1331$。

分析：判断一个数是否回文，可以用除以 10 取余的方法，从最低位开始，依次取出该数的各位数字，然后用最低位充当最高位，按反序重新构成新的数，比较与原数是否相等，若相等，则原数为回文。

源程序：

```
//3_4.cpp
#include <iostream>
```

```
using namespace std;

//判断 n 是否为回文数
bool symm(unsigned n) {
    unsigned i=n;
    unsigned m=0;
    while (i>0) {
        m=m * 10+i%10;
        i /=10;
    }
    return m==n;
}

int main() {
    for (unsigned m=11; m <1000; m++)
        if (symm(m) && symm(m * m) && symm(m * m * m)) {
            cout<<"m="<<m;
            cout<<"  m * m="<<m * m;
            cout<<"  m * m * m="<<m * m * m<<endl;
        }
    return 0;
}
```

运行结果：

```
m=11  m * m=121  m * m * m=1331
m=101  m * m=10201  m * m * m=1030301
m=111  m * m=12321  m * m * m=1367631
```

例 3-5　计算如下公式，并输出结果。

$$k=\begin{cases}\sqrt{\sin^2 r+\sin^2 s} & (r^2 \leqslant s^2)\\ \dfrac{1}{2}\sin(rs) & (r^2 > s^2)\end{cases}$$

其中 r、s 的值由键盘输入。$\sin x$ 的近似值按如下公式计算：

$$\sin x=\frac{x}{1!}-\frac{x^3}{3!}+\frac{x^5}{5!}-\frac{x^7}{7!}+\cdots=\sum_{n=1}^{\infty}(-1)^{n-1}\frac{x^{2n-1}}{(2n-1)!}$$

计算精度为 10^{-10}，当某项的绝对值小于计算精度时，停止累加，累加和即为该精度下的 $\sin x$ 的近似值。

源程序：

```
//3_5.cpp
#include <iostream>
#include <cmath>  //头文件 cmath 中具有对 C++标准库中数学函数的说明
using namespace std;

const double TINY_VALUE=1e-10;
```

```
double tsin(double x) {
    double g=0;
    double t=x;
    int n=1;
    do {
        g+=t;
        n++;
        t=-t*x*x/(2*n-1)/(2*n-2);
    } while (fabs(t)>=TINY_VALUE);
    return g;
}

int main() {
    double k, r, s;
    cout<<"r=";
    cin>>r;
    cout<<"s=";
    cin>>s;
    if (r*r<=s*s)
        k=sqrt(tsin(r)*tsin(r)+tsin(s)*tsin(s));
    else
        k=tsin(r*s)/2;
    cout<<k<<endl;
    return 0;
}
```

运行结果:

```
r=5
s=8
1.37781
```

本程序中用到了两个标准 C++ 的系统函数——绝对值函数 double fabs(double x)和平方根函数 double sqrt(double x),它们的原型都在 cmath 头文件中定义。3.5 节将专门介绍标准 C++ 的系统函数。

例 3-6 投骰子的随机游戏。

游戏规则是:每个骰子有 6 面,点数分别为 1、2、3、4、5、6。游戏者在程序开始时输入一个无符号整数,作为产生随机数的种子。

每轮投两次骰子,第一轮如果和数为 7 或 11 则为胜,游戏结束;和数为 2、3 或 12 则为负,游戏结束;和数为其他值则将此值作为自己的点数,继续第二轮、第三轮……直到某轮的和数等于点数则取胜,若在此前出现和数为 7 则为负。

由 rollDice 函数负责模拟投骰子、计算和数并输出和数。

提示 系统函数 int rand(void)的功能是产生一个伪随机数,伪随机数并不是真正随机的。这个函数自己不能产生真正的随机数。如果在程序中连续调用 rand,期望由此可以产

生一个随机数序列，你会发现每次运行这个程序时产生的序列都是相同的，这称为伪随机数序列。这是因为函数 rand 需要一个称为“种子”的初始值，种子不同，产生的伪随机数也就不同。因此只要每次运行时给予不同的种子，然后连续调用 rand 便可以产生不同的随机数序列。如果不设置种子，rand 总是默认种子为 1。不过设置种子的方法比较特殊，不是通过函数的参数，而是在调用它之前，需要首先调用另外一个函数 void srand(unsigned int seed) 为其设置种子，其中的参数 seed 便是种子。

源程序：

```
//3_6.cpp
#include <iostream>
#include <cstdlib>
using namespace std;

//投骰子,计算和数,输出和数
int rollDice() {
    int die1=1+rand()%6;
    int die2=1+rand()%6;
    int sum=die1+die2;
    cout<<"player rolled "<<die1<<"+"<<die2<<"="<<sum<<endl;
    return sum;
}

enum GameStatus {WIN, LOSE, PLAYING};

int main() {
    int sum, myPoint;
    GameStatus status;

    unsigned seed;
    cout<<"Please enter an unsigned integer: ";
    cin>>seed;      //输入随机数种子
    srand(seed);    //将种子传递给 rand()

    sum=rollDice(); //第一轮投骰子、计算和数
    switch (sum) {
    case 7:         //如果和数为 7 或 11 则为胜,状态为 WIN
    case 11:
        status=WIN;
        break;
    case 2:         //和数为 2、3 或 12 则为负,状态为 LOSE
    case 3:
    case 12:
        status=LOSE;
        break;
    default:        //其他情况,游戏尚无结果,状态为 PLAYING,记下点数,为下一轮做准备
```

```
        status=PLAYING;
        myPoint=sum;
        cout<<"point is "<<myPoint<<endl;
        break;
    }

    while (status==PLAYING) {     //只要状态仍为 PLAYING,就继续进行下一轮
        sum=rollDice();
        if (sum==myPoint)         //某轮的和数等于点数则取胜,状态置为 WIN
            status=WIN;
        else if (sum==7)          //出现和数为 7 则为负,状态置为 LOSE
            status=LOSE;
    }

    //当状态不为 PLAYING 时上面的循环结束,以下程序段输出游戏结果
    if (status==WIN)
        cout<<"player wins"<<endl;
    else
        cout<<"player loses"<<endl;

    return 0;
}
```

运行结果 1：

```
Please enter an unsigned integer:8
player rolled 5+1=6
point is 6
player rolled 6+6=12
player rolled 6+4=10
player rolled 6+6=12
player rolled 6+6=12
player rolled 3+2=5
player rolled 2+2=4
player rolled 3+4=7
player loses
```

运行结果 2：

```
Please enter an unsigned integer:23
player rolled 6+3=9
point is 9
player rolled 5+4=9
player wins
```

2. 嵌套调用

函数允许嵌套调用。如果函数 1 调用了函数 2,函数 2 再调用函数 3,便形成了函数的

嵌套调用。

例 3-7　输入两个整数，求它们的平方和。

分析：虽然这个问题很简单，但是为了说明函数的嵌套调用问题，在这里设计两个函数：求平方和函数 fun1 和求一个整数的平方函数 fun2。由主函数调用 fun1，fun1 又调用 fun2。

源程序：

```
//3_7.cpp
#include <iostream>
using namespace std;

int fun2(int m) {
    return m * m;
}

int fun1(int x,int y) {
    return fun2(x)+fun2(y);
}

int main() {
    int a, b;
    cout<<"Please enter two integers(a and b): ";
    cin>>a>>b;
    cout<<"The sum of square of a and b: "<<fun1(a, b)<<endl;
    return 0;
}
```

运行结果：

```
Please enter two integers(a and b):3 4
The sum of square of a and b:25
```

图 3-1 说明了例 3-7 函数的调用过程，图中标号标明了执行顺序。

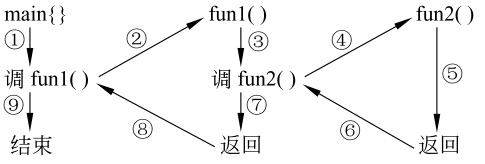

图 3-1　例 3-7 函数的调用过程

3. 递归调用

函数可以直接或间接地调用自身，称为递归调用。

所谓直接调用自身，就是指在一个函数的函数体中出现了对自身的调用表达式，例如：

```
void fun1() {
   …
```

```
    fun1();          //调用 fun1 自身
    …
}
```

就是函数直接调用自身的例子。

而下面的情况是函数间接调用自身:

```
void fun1() {
    …
    fun2();
    …
}
void fun2() {
    …
    fun1();
    …
}
```

这里 fun1 调用了 fun2,而 fun2 又调用了 fun1,于是构成了递归。

递归算法的实质是将原有的问题分解为新的问题,而解决新问题时又用到了原有问题的解法。按照这一原则分解下去,每次出现的新问题都是原有问题的简化的子集,而最终分解出来的问题,是一个已知解的问题。这便是有限的递归调用。只有有限的递归调用才是有意义的,无限的递归调用永远得不到解,没有实际意义。

递归的过程有两个阶段。

第一阶段:递推。将原问题不断分解为新的子问题,逐渐从未知向已知推进,最终达到已知的条件,即递归结束的条件,这时递推阶段结束。

例如,求 5!,可以这样分解:

$5!=5\times4!\ \rightarrow 4!=4\times3!\ \rightarrow 3!=3\times2!\ \rightarrow 2!=2\times1!\ \rightarrow 1!=1\times0!\ \rightarrow 0!=1$

未知——————————————————————→已知

第二阶段:回归。从已知的条件出发,按照递推的逆过程,逐一求值回归,最后达到递推的开始处,结束回归阶段,完成递归调用。

例如,求 5!的回归阶段如下:

$5!=5\times4!=120\leftarrow 4!=4\times3!=24\leftarrow 3!=3\times2!=6\leftarrow 2!=2\times1!=2\leftarrow 1!=1\times0!=1\leftarrow 0!=1$

未知←——————————————————————已知

例 3-8 求 $n!$。

分析:计算 $n!$的公式如下:

$$n!=\begin{cases}1 & (n=0)\\ n(n-1)! & (n>0)\end{cases}$$

这是一个递归形式的公式,在描述“阶乘”算法时又用到了“阶乘”这一概念,因而编程时也自然采用递归算法。递归的结束条件是 $n=0$。

源程序:

```
//3_8.cpp
```

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

//计算 n 的阶乘
unsigned fac(unsigned n) {
    unsigned f;
    if (n==0)
        f=1;
    else
        f=fac(n-1) * n;
    return f;
}

int main() {
    unsigned n;
    cout<<"Enter a positive integer: ";
    cin>>n;
    unsigned y=fac(n);
    cout<<n<<"!="<<y<<endl;
    return 0;
}
```

运行结果：

```
Enter a positive integer: 8
8!=40320
```

注意　对同一个函数的多次不同调用中，编译器会为函数的形参和局部变量分配不同的空间，它们互不影响。例如，在执行 fac(2)时调用 fac(1)时，会用 1 为该调用中作为被调函数的 fac 函数的形参 n 初始化，但这不会改变作为主调函数的 fac 中的形参 n 的值，也就是说当 fac(1)返回后，读取 n 的值时仍然能够得到 2 这一结果。对于 fac 函数中定义的变量 f 也一样。这一问题涉及变量的生存期，会在第 5 章详细讨论。

例 3-9　用递归法计算从 n 个人中选择 k 个人组成一个委员会的不同组合数。

分析：由 n 个人里选 k 个人的组合数

＝由 $n-1$ 个人里选 k 个人的组合数＋由 $n-1$ 个人里选 $k-1$ 个人的组合数

由于计算公式本身是递归的，因此可以编写一个递归函数来完成这一功能，递推的结束条件是 $n=k$ 或 $k=0$，这时的组合数为 1，然后开始回归。

源程序：

```
//3_9.cpp
#include <iostream>
using namespace std;

//计算从 n 个人里选 k 个人的组合数
int comm(int n, int k) {
```

```
    if (k>n)
        return 0;
    else if (n==k || k==0)
        return 1;
    else
        return comm(n-1, k)+comm(n-1, k-1);
}

int main() {
    int n, k;
    cout<<"Please enter two integers n and k: ";
    cin>>n>>k;
    cout<<"C(n, k)="<<comm(n, k)<<endl;
    return 0;
}
```

运行结果：

```
Please enter two integers n and k: 18 5
C(n, k)=8568
```

例 3-10 汉诺塔问题。

有三根针 A、B、C。A 针上有 n 个盘子，盘子大小不等，大的在下，小的在上，如图 3-2 所示。要求把这 n 个盘子从 A 针移到 C 针，在移动过程中可以借助 B 针，每次只允许移动一个盘，且在移动过程中在三根针上都保持大盘在下，小盘在上。

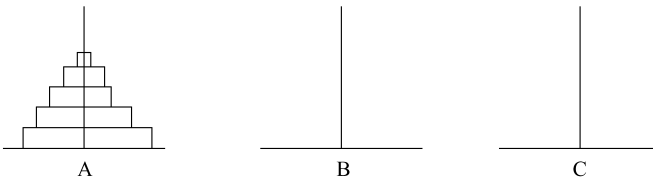

图 3-2 汉诺塔问题示意图

分析：将 n 个盘子从 A 针移到 C 针可以分解为下面三个步骤。

(1) 将 A 上 $n-1$ 个盘子移到 B 针上(借助 C 针)。

(2) 把 A 针上剩下的一个盘子移到 C 针上。

(3) 将 $n-1$ 个盘子从 B 针移到 C 针上(借助 A 针)。

事实上，上面 3 个步骤包含两种操作：

(1) 将多个盘子从一个针移到另一个针上，这是一个递归的过程。

(2) 将 1 个盘子从一个针上移到另一针上。

于是用两个函数分别实现上面两种操作，用 hanoi 函数实现第(1)种操作，用 move 函数实现第(2)种操作。

源程序：

```
//3_10.cpp
```

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

//把 src 针的最上面一个盘子移动到 dest 针上
void move(char src, char dest) {
    cout<<src<<" -->"<<dest<<endl;
}

//把 n 个盘子从 src 针移动到 dest 针,以 medium 针作为中介
void hanoi(int n, char src, char medium, char dest) {
    if (n==1)
        move(src, dest);
    else {
        hanoi(n-1, src, dest, medium);
        move(src, dest);
        hanoi(n-1, medium, src, dest);
    }
}

int main() {
    int m;
    cout<<"Enter the number of disks: ";
    cin>>m;
    cout<<"the steps to move "<<m<<" disks:"<<endl;
    hanoi(m, 'A', 'B', 'C');
    return 0;
}
```

运行结果：

```
Enter the number of disks:3
the steps to move 3 disks:
A -->C
A -->B
C -->B
A -->C
B -->A
B -->C
A -->C
```

3.1.3　函数的参数传递

在函数未被调用时，函数的形参并不占有实际的内存空间，也没有实际的值。只有在函数被调用时才为形参分配存储单元，并将实参与形参结合。每个实参都是一个表达式，其类型必须与形参相符。函数的参数传递指的就是形参与实参结合（简称形实结合）的过程，形实结合的方式有值传递和引用传递。

1. 值传递

值传递是指当发生函数调用时,给形参分配内存空间,并用实参来初始化形参(直接将实参的值传递给形参)。这一过程是参数值的单向传递过程,一旦形参获得了值便与实参脱离关系,此后无论形参发生了怎样的改变,都不会影响实参。

例 3-11 将两个整数交换次序后输出。

```
//3_11.cpp
#include <iostream>
using namespace std;

void swap(int a, int b) {
    int t=a;
    a=b;
    b=t;
}

int main() {
    int x=5, y=10;
    cout<<"x="<<x<<"    y="<<y<<endl;
    swap(x, y);
    cout<<"x="<<x<<"    y="<<y<<endl;
    return 0;
}
```

运行结果:

```
x=5      y=10
x=5      y=10
```

分析:从上面的运行结果可以看出,并没有达到交换的目的。这是因为,采用的是值传递,函数调用时传递的是实参的值,是单向传递过程。形参值的改变对实参不起作用。图 3-3是程序执行时变量的情况。

2. 引用传递

我们已经看到,值传递时参数是单向传递,那么如何使在子函数中对形参做的更改对主函数中的实参有效呢?这就需要使用引用传递。

引用是一种特殊类型的变量,可以被认为是另一个变量的别名,通过引用名与通过被引用的变量名访问变量的效果是一样的,例如:

```
int i, j;
int &ri=i;       //建立一个 int 型的引用 ri,并将其初始化为变量 i 的一个别名
j=10;
ri=j;            //相当于 i=j;
```

使用引用时必须注意下列问题。

- 声明一个引用时,必须同时对它进行初始化,使它指向一个已存在的对象。
- 一旦一个引用被初始化后,就不能改为指向其他对象。

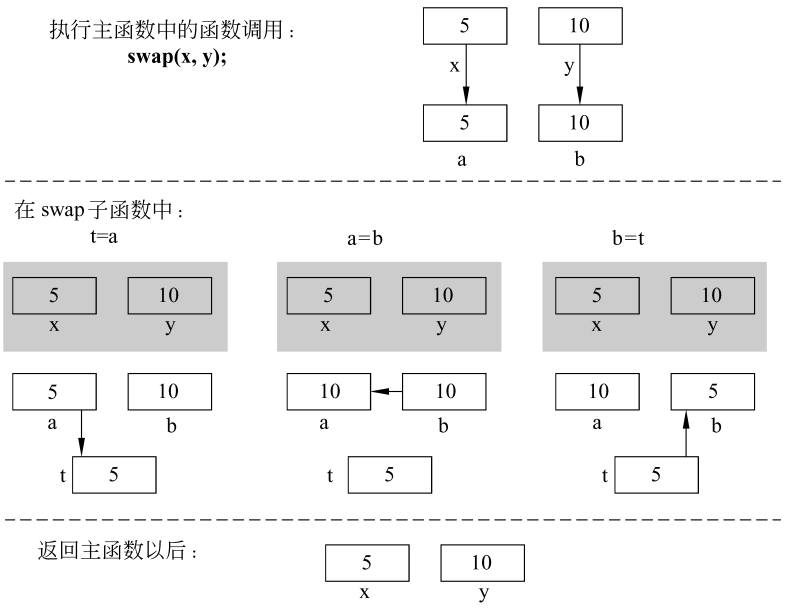

图 3-3　例 3-11 程序执行时变量的情况

也就是说，一个引用，从它诞生之时起，就必须确定是哪个变量的别名，而且始终只能作为这一个变量的别名，不能另作他用。

引用也可以作为形参，如果将引用作为形参，情况便稍有不同。这是因为，形参的初始化不在类型说明时进行，而是在执行主调函数中的调用表达式时，才为形参分配内存空间，同时用实参来初始化形参。这样引用类型的形参就通过形实结合，成为实参的一个别名，对形参的任何操作也就会直接作用于实参。

用引用作为形参，在函数调用时发生的参数传递，称为引用传递。

例 3-12　使用引用传递改写例 3-11 的程序，使两整数成功地进行交换。

```
//3_12.cpp
#include <iostream>
using namespace std;

void swap(int &a, int &b) {
    int t=a;
    a=b;
    b=t;
}

int main() {
    int x=5, y=10;
    cout<<"x="<<x<<"    y="<<y<<endl;
    swap(x, y);
    cout<<"x="<<x<<"    y="<<y<<endl;
    return 0;
```

```
}
```

运行结果：

```
x=5    y=10
x=10   y=5
```

分析：从运行结果可以看出，改用引用传递后成功地实现了交换。引用传递与值传递的区别只是函数的形参写法不同。主调函数中的调用表达式是完全一样的。图 3-4 是程序执行时变量的情况。

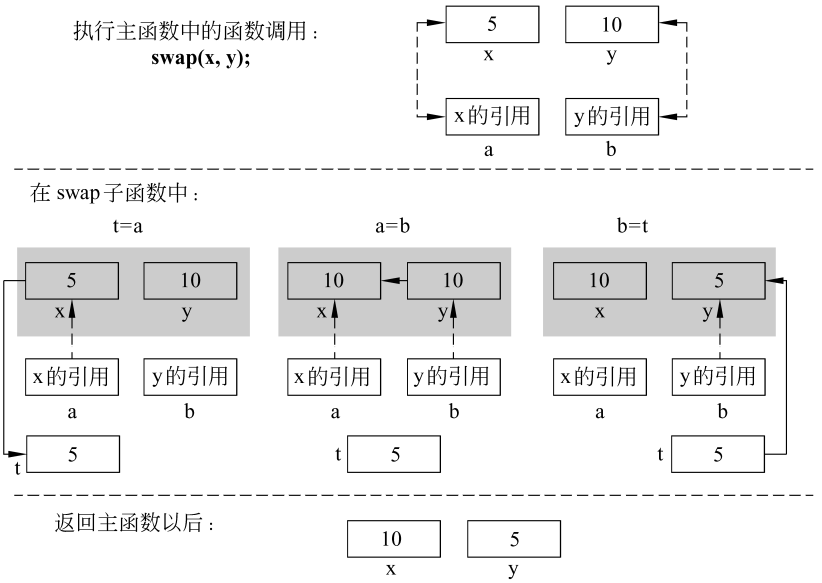

图 3-4　例 3-12 程序执行时变量的情况

例 3-13　值传递与引用传递的比较。

```
//3_13.cpp
#include <iostream>
#include <iomanip>
using namespace std;

void fiddle(int in1, int &in2) {
    in1=in1+100;
    in2=in2+100;
    cout<<"The values are ";
    cout<<setw(5)<<in1;
    cout<<setw(5)<<in2<<endl;
}

int main() {
    int v1=7, v2=12;
    cout<<"The values are ";
```

```
    cout<<setw(5)<<v1;
    cout<<setw(5)<<v2<<endl;
    fiddle(v1, v2);
    cout<<"The values are ";
    cout<<setw(5)<<v1;
    cout<<setw(5)<<v2<<endl;
    return 0;
}
```

运行结果：

```
The values are    7  12
The values are  107 112
The values are    7 112
```

分析：子函数 fiddle 的第一个参数 in1 是普通的 int 型，被调用时传递的是实参 v1 的值，第二个参数 in2 是引用，被调用时由实参 v2 初始化后成为 v2 的一个别名。于是在子函数中对参数 in1 的改变不影响实参，而对形参 in2 的改变实质上就是对主函数中变量 v2 的改变。因而返回主函数后，v1 值没有变化，而 v2 值发生了变化。

3. 含有可变数量形参的函数

当无法提前预知应该向函数传递几个实参时，例如，在编写代码输出程序产生错误信息时，最好统一用一个函数实现该功能，使得对所有错误的处理能够整齐划一。然而错误信息的种类不同，调用错误信息输出函数时传递的参数也会各不相同。

为了编写能处理不同数量实参的函数，C++ 标准中提供了两种主要的方法：如果所有的实参类型相同，可以传递一个名为 initializer_list 的标准库类型；如果实参的类型不同，可以编写可变参数模板的类，关于它的细节将在 9.5 节进行介绍。

initializer_list 是一种标准库类型，用于表示某种特定类型的值的数组，该类型定义在同名的头文件中。

initializer_list 提供的操作：

initializer_list＜T＞ lst：默认初始化；T 类型元素的空列表。

initializer_list＜T＞ lst{a，b，c…}：lst 的元素数量和初始值一样多；lst 的元素是对应初始值的副本；列表中的元素是 const。

lst2(lst)，lst2＝lst：复制或者赋值一个 initializer_list 对象但不复制列表中的元素；复制后原始列表和副本共享元素。

lst.size()：列表中的元素数量。

lst.begin()：返回指向 lst 首元素的指针。

lst.end()：返回指向 lst 尾元素下一位置的指针。

initializer_list 是一个类模板，模板相关内容会在第 9 章详细介绍，在这里先试着使用它。

使用模板时，需要在模板名字后面跟一对尖括号，括号内给出类型参数。例如：

```
initializer_list<string> ls;    //initializer_list 的元素类型是 string
initializer_list<int>li;        //initializer_list 的元素类型是 int
```

initializer_list 比较特殊的一点是,其对象中的元素永远是常量值,人们无法改变 initializer_list 对象中元素的值。

接下来使用 initializer_list 编写一个错误信息输出函数,使其可以作用于可变数量的形参:

```
void print_err(initializer_list<string>lst) {
    for (auto beg=lst.begin(); beg !=lst.end();++beg)
        cout<< *beg<<' ';
    cout<<endl;
}
```

如果想向 initializer_list 形参中传递一个值的序列,则必须把序列放在一对花括号内,例如:

```
//expected 和 actual 是 string 对象
if (expected !=actual)
    print_err( {"return", actual, ",", expected, "is", "expected"} );
else
    print_err( {"function", "right"} );
```

例 3-13 中调用了同一个函数 print_err,但是两次调用传递的参数数量不同:第一次传入了 6 个值,第二次只传入了 2 个值。值得注意的是,例 3-13 函数中含有 initializer_list 类对象 lst 是函数的一个形参,它也可以同时拥有其他类型的任意个形参,如添加 int 类型的错误代码形参时:

```
void print_err(initializer_list<string>lst, int error_code); //第二个形参为 int 类型
```

3.2 内联函数

在本章的开头提到,使用函数有利于代码重用,可以提高开发效率、增强程序的可靠性,也便于分工合作,便于修改维护。但是,函数调用也会降低程序的执行效率,增加时间和空间方面的开销。因此,对于一些功能简单、规模较小又使用频繁的函数,可以设计为内联函数。**内联函数不是在调用时发生控制转移,而是在编译时将函数体嵌入在每一个调用处**。这样就节省了参数传递、控制转移等开销。

内联函数在定义与普通函数的定义方式几乎一样,只是需要使用关键字 inline,其语法形式如下:

```
inline  类型说明符  函数名(含类型说明的形参表)
{
    语句序列
}
```

需要注意的是,inline 关键字只是表示一个要求,编译器并不承诺将 inline 修饰的函数作为内联函数。而在现代编译器中,没有用 inline 修饰的函数也可能被编译为内联函数。通常内联函数应该是比较简单的函数,结构简单、语句少,调用频繁。如果将一个复杂的函

数定义为内联函数，反而会造成代码膨胀，增大开销。这种情况下，多数编译器都会自动将其转换为普通函数来处理。到底什么样的函数会被认为太复杂呢？不同的编译器处理起来是不同的。此外，有些函数是肯定无法以内联方式处理的，例如存在对自身的直接递归调用的函数。

例 3-14　内联函数应用举例。

```
//3_14.cpp
#include <iostream>
using namespace std;

const double PI=3.14159265358979;

//内联函数,根据圆的半径计算其面积
inline double calArea(double radius) {
    return PI * radius * radius;
}

int main() {
    double r=3.0;     //r 是圆的半径
    //调用内联函数求圆的面积,编译时此处被替换为 CalArea 函数体语句,
    //展开为 area=PI * radius * radius;
    double area=calArea(r);
    cout<<area<<endl;
    return 0;
}
```

运行结果：

```
28.2743
```

* 3.3　constexpr 函数

constexpr 函数是指能用于常量表达式的函数。定义 constexpr 函数的方法与其他函数类似，但要遵循几项约定：函数的返回类型以及所有的形参类型必须是常量，而且函数体中必须有且仅有一条 return 语句：

```
constexpr int get_size() {return 20;}
constexpr int foo=get_size();          //正确: foo 是一个常量表达式
```

把 get_size 定义成无参数的 constexpr 函数，编译器能在程序编译时验证 get_size 函数返回的是常量表达式，因此可以用 get_size 函数初始化 constexpr 类型的变量。执行初始化任务时，编译器把对 constexpr 函数的调用替换成其结果值，为了能在编译过程中随时展开，constexpr 函数被隐式地指定为内联函数。constexpr 函数体内也可以包含其他语句，只要这些语句在运行时不执行任何操作就行。例如，constexpr 函数中可以有空语句，类型别名以及 using 声明。