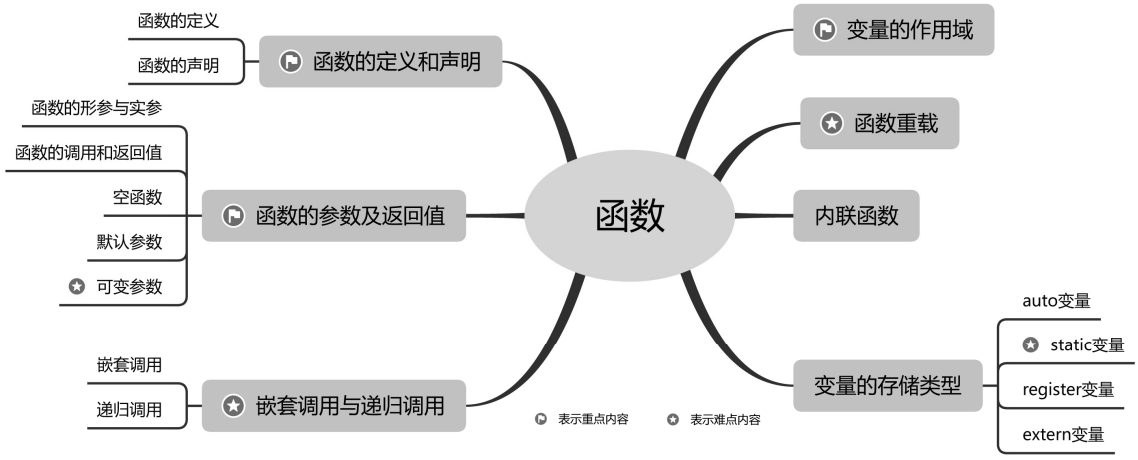
第 6 章

函数

函数是能够实现特定功能的程序模块，它可以是只有一条语句的简单函数，也可以是包含许多子函数的复杂函数。函数之间可以相互调用，可以将联系密切的语句都放到一个函数内，也可以将复杂的函数分解成多个子函数。熟练掌握函数可以将程序结构设计得更加合理。

本章知识架构及重难点如下：



6.1 函数的定义和声明

C++程序的入口和出口都位于 main()函数中。除此以外，开发者可以调用系统内置的库函数，还可以自行定义很多函数。main()函数可以调用其他函数，其他函数间也可以相互调用。main()函数就像总工程师一样，控制着整体程序的推进和执行。

6.1.1 函数的定义

一个函数应包括函数头和函数体。定义一个函数的语法格式如下：

```
返回值类型 函数名(参数列表)
{
    变量声明
    语句块
}
```

函数头是函数的入口，包括返回值类型、函数名和形式参数列表，标志着一段函数代码的开始。

其中，返回值类型可以是整型、字符型、指针型或对象，通过返回值可以判断函数的执行情况，也可以获取想要的数据；函数名是开发者为函数起的名称，其命名应符合标识符命名规则；参数列表是由各种类型变量组成的列表，参数间用逗号分隔。

根据功能，可将函数分为字符函数、日期函数、数学函数、图形函数、内存函数等。

关于函数定义的说明如下：

（1）形式参数列表可以为空，此时表明并不需要通过函数返回某个值。例如：

```
int ShowMessage()
{
    int i=0;
    cout << i << endl;
    return 0;
}
```

上述代码中，函数 ShowMessage 为无参函数，作用是通过 cout 流输出变量 i 的值。

（2）函数名后的大括号表示函数体，在函数体内进行变量声明和添加实现语句。

（3）函数体内不能再包含其他函数的定义。

6.1.2　函数的声明

调用函数前，要先对函数进行声明。函数定义是为了让编译器知道函数的功能，而函数声明是为了让编译器预先知道有这么一个函数（并不需要知道具体功能），以及函数的名称、参数、返回值类型等信息。声明函数的代码和定义函数时的函数头基本一致，但末尾要添加分号“;”。

函数声明的一般形式如下：

```
返回值类型 函数名(参数列表);
```

例如，声明一个函数的代码如下：

```
int SetIndex(int i);
```

函数声明也称为函数原型，函数声明时可以省略变量名。例如，上述函数声明也可写为：

```
int SetIndex(int );
```

【实例 6.1】编写一个简单的函数。**（实例位置：资源包\TM\sl\6\1）**

本实例中，定义 Poem()函数，并在主函数中调用它。注意函数声明和函数定义的用法。

```
#include<stdio.h>                    //包含头文件
void Poem();                         //声明 Poem()函数
int main()
{
    Poem();                          //调用 Poem()函数
    return 0;
}
void Poem()                          //定义 Poem()函数，输出“会写诗”
{
    printf("会写诗\n");
}
```

程序运行结果如图 6.1 所示。Poem()是一个无参函数，函数功能很简单，调用后输出了“会写诗”。

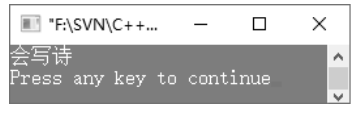


图 6.1　程序运行结果

6.2 函数的参数及返回值

声明函数后，可以在后续的代码中调用该函数，并通过参数和返回值实现数据间的传递。标准 C++是一种强制类型检查的语言，调用函数前必须把函数的参数类型和返回值类型告知编译器。

6.2.1 函数的形参与实参

声明和定义函数时，函数名后面括号中的参数称为形式参数，简称形参。这些参数只是定义了类型，在实际参数传入前并没有实际意义。调用函数的过程就是真正使用函数的过程，因此调用函数会传递一些要实际参与运算的参数给被调用函数，简称实参。

实参与形参的个数应相等，类型应一致，并按顺序对应，调用时一对一地传递数据，如图 6.2 所示。

```
                形参    形参
int function(int a,int b);
void main()
{
                实参   实参
    function( 3 , 4 );

    cout << "the loop end" << endl;
}
int function(int a,int b)
{
    return a+b;
}
```

图 6.2 形参与实参

形参与实参的区别如下：

（1）定义函数时指定的形参，在未出现函数调用前，并不占用内存中的存储单元。只有在发生函数调用时，函数的形参才被分配内存单元。在调用结束后，其所占的内存单元将被释放。

（2）实参是确定的值。在调用时将实参的值赋给形参，如果形参是指针类型，就将地址值传递给形参。

（3）实参与形参的类型相同。

（4）实参与形参之间是单向传递，只能由实参传递给形参，而不能由形参传回给实参。

实参与形参之间存在一个分配空间和参数值传递的过程，这个过程是在函数调用时发生的，C++支持引用型变量，但没有值传递的过程，这将在后文讲解。

说明

实参可以是常量、变量、数组或指针等，还可以是表达式。

6.2.2 函数的调用与返回值

1. 函数的调用

函数调用发生时，会立即暂停主调函数，转而执行被调用函数中的程序段。直到被调用函数执行完毕，才会再次回到主调函数，继续执行后面的语句。实例 6.1 中，我们已经接触过了函数调用。

函数调用的说明如下：

（1）被调用的函数必须是已经存在的函数（库函数或用户自定义函数）。

（2）如果使用库函数，需在主函数中使用#include 预编译指令引入对应的头文件。

（3）如果使用用户自定义函数，需在主调函数中声明被调用函数。

函数的调用方式有 3 种，分别是语句调用、表达式调用和函数参数调用。

（1）函数可以作为一个独立语句出现，这是最常用的函数调用方式。语句调用中，函数可以有返回值，也可以没有返回值。例如：

```
AddTwoNum(3,5);                    //通过语句调用 AddTwoNum()函数
```

（2）函数调用可以出现在表达式中，此时函数必须返回一个确定的值，作为表达式运算的一部分。例如：

```
iResult=iNum*AddTwoNum(3,5);       //函数调用出现在表达式中，返回值参与乘法运算
```

（3）函数调用可以出现在函数参数中，此时函数的返回值作为实际参数使用。例如：

```
iResult=AddTwoNum(10,Max(3,5));    //函数调用出现在函数参数中
```

2．函数的返回值

返回值是函数被调用时，执行函数体中程序段后得到并返回给主调函数的值。

函数的返回值通过 return 语句返回，其一般形式如下：

```
return (表达式);
```

return 语句可将表达式的值返回给主调函数。关于返回值的说明如下：

（1）函数返回值的类型和函数定义时的类型标识符应保持一致。如果两者不一致，则以函数类型为准，自动进行类型转换。

（2）如函数值为整型，在函数定义时可以省去类型标识符。

（3）函数中允许有多个 return 语句，但每次调用只能有一个被执行，因此只能返回一个函数值。

（4）不需要返回函数值的函数，可以明确定义为“空类型”，类型标识符为 void。例如：

```
void ShowIndex()
{
    int iIndex=10;
    cout << "Index is :" << iIndex << endl;
}
```

（5）类型标识符为 void 的函数不能进行赋值运算及值传递。例如：

```
i= ShowIndex();                    //错误，空类型函数不能进行赋值
SetIndex(ShowIndex);               //错误，空类型函数不能进行值传递
```

说明

为了降低程序出错的概率，凡不要求返回值的函数都应定义为 void 空类型。

3．传值调用

主调函数和被调函数之间存在着数据传递关系，也就是说，主调函数需要将实参传递给被调函数的形参，这种调用方式称为传值调用。传值调用是函数调用的基本方式。需要注意的是，传值调用是单向的，只能把实参的值传递给形参，却无法把形参的值传回实参。

【实例 6.2】认识传值调用。**（实例位置：资源包\TM\sl\6\2）**

本实例中，在主函数中输入两个数，先调用 add()函数，实现两个数的求和；然后判断 x 和 y 的大

小，当 x 小于 y 时调用 swap()函数，交换两个数值。程序代码如下：

```
#include <iostream.h>

int add(int a,int b);                          //声明 add()函数
void swap(int a,int b);                        //声明 swap 函数
int main()
{
        int x,y,z;
        cout << "输入两个数" << endl;
        cin >> x;                              //输入 x
        cin >> y;                              //输入 y
        z=add(x,y);                            //调用 add()函数，计算两个数的和，返回值赋给 z
        cout << "x+y=" <<z <<endl;             //输出 z
        if(x<y)                                //如果 x<y
                swap(x,y);                     //调用 swap()函数，交换两个数值
        cout << "x=" << x <<endl;              //输出 x
        cout << "y=" << y <<endl;              //输出 y
}
int add(int a,int b)                           //定义 add()函数，可正常传值调用
{
        int sum;
        sum=a+b;
        return sum;
}
void swap(int a,int b)                         //定义 swap()函数，无法将交换后的数值传回
{
        int tmp;
        tmp=a;
        a=b;
        b=tmp;
}
```

程序运行结果如图 6.3 所示，可见 x、y 进行了求和，但并未进行交换。这是为什么呢？原因就是调用函数时的值传递过程是单向的，只能把实参的值传递给形参（本质是复制了一份实参值给形参），形参的值即便发生了改变，也无法再传递回来。

图 6.3　传值调用

add()函数中，实参 x、y 的值传递给形参 a、b，进行加法计算后，通过 return 语句返回两数之和。swap()函数中，实参 x 和 y 的值传递给形参 a、b，进行两数交换，但交换后的值却无法再传回给实参。

说明

C++中，参数传递方式有两种：值传递和引用传递。值传递时，程序会将实参值复制一份副本，传递给函数形参，改变副本的值不会影响实参；引用传递时，程序会将实参的内存地址作为参数传递给函数形参，改变地址内存放的内容，会同时影响到实参。

要想通过函数调用交换变量的值，必须通过指针和引用传递方式来实现，第 7 章中会详细讲解。

6.2.3　空函数

空函数就是没有参数和返回值，函数作用域为空的函数。例如：

```
void setWorkSpace()
{  }
```

调用空函数时，程序不会执行任何操作。

空函数的存在有什么意义呢？实际开发中，各功能模块需要由不同的函数来实现，第一阶段只设计最基本的模块，其他一些次要功能或锦上添花的功能会在以后陆续补充。此时，就需要在将来会扩充功能的地方先写上一个空函数，占一个位置，后续再用编好的函数代替它。

6.2.4　默认参数

调用带参函数时，如果经常需要传递同一个值，不妨在定义该函数时设置一个默认参数值。

设置默认参数的好处是：调用函数时如果省略参数，表示用默认值作为函数的实际参数；如果不省略参数，表示传递实际参数到函数形参处。

例如，下面的代码定义了一个带有默认参数值的函数。

```
void OutputInfo(const char *pchData = "One world,one dream!")
{
    cout << pchData << endl;          //输出默认信息
}
```

【实例 6.3】展示某公司口号。**（实例位置：资源包\TM\sl\6\3）**

输出两行字符串：一行使用默认值作为函数实参，一行将字符串作为函数实参。代码如下：

```
#include <iostream>
using namespace std;
void OutputInfo(const char *pchData = "GO UP OR GO HOME")
{
    cout << pchData << endl;          //输出默认信息
}
int main()
{
    OutputInfo();                     //无参调用，默认值作为函数实际参数
    OutputInfo("口号!");              //有参调用，传递实际参数
}
```

程序运行结果如图 6.4 所示。

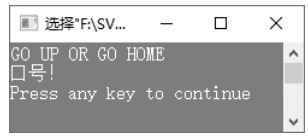


图 6.4　调用默认参数的函数

注意，如果函数有多个参数，应保证有默认值的参数出现在参数列表的右方，无默认值的参数出现在参数列表的左方，即默认值参数不能出现在非默认值参数的左方。

例如，下面的函数定义是非法的，默认值参数 y 出现在非默认值参数 z 的左方，导致编译错误。

```
int GetMax(int x,int y=10 ,int z)     //非法的函数定义，默认参数 y 出现在参数 z 的左方
{
    if (x < y)                        //如果 x 小于 y
        x = y;
    if (x < z)                        //如果 x 小于 z
        x = z;
    return x;                         //返回 x
}
```

正确的做法是将默认值参数放置在参数列表的右方，将函数头改为：

```
int GetMax(int x,int y ,int z=10)            //定义默认值参数
```

编程训练（答案位置：资源包\TM\sl\6\编程训练\）

【训练 1】种瓜得瓜　设计一个函数，该函数接受一个字符串参数，当传递参数时，输出该参数的值；当不传递参数时，输出“什么也不说，祖国知道我!”

【训练 2】电梯是否超重　编写电梯测重函数，函数使用可变参数，传递电梯中所有乘客的体重。如果电梯中的质量超过 1000kg，则返回 0，否则返回 1。

6.3　嵌套调用与递归调用

6.3.1　嵌套调用

在一个函数体内可以调用另外一个函数，这种调用方式称为嵌套调用。例如：

```
#include <iostream>
using namespace std;
void   ShowMessage()                         //定义 ShowMessage()函数
{
       cout <<"The ShowMessage function" << endl;
}
void   Display()                             //定义 Display()函数
{
       ShowMessage();                        //在 Display()函数中调用 ShowMessage()函数
}
int main()
{
       Display();                            //在主函数中调用 Display()函数
}
```

注意，C++中不允许函数进行嵌套定义，因此不能在一个函数体内定义另一个函数。例如：

```
int main()
{
       void   Display()                      /*错误!!! 不能在函数内定义另外一个函数*/
       {
              cout << "I want to show the Nesting function" << endl;
       }
       return 0;
}
```

嵌套调用的层数一般是没有限制的，但个别编译器可能会有一些限制，使用时应注意。

6.3.2　递归调用

所谓递归调用，就是函数自己调用自己。从定义中可以看出，函数递归调用是函数嵌套调用的一种特殊形式。

使用递归方法解决问题的优点是：问题描述清楚，代码可读性强，结构清晰，代码量比使用非递归方法少。缺点是：递归程序的运行效率较低，无论是从时间角度还是从空间角度都比非递归程序差。对时间复杂度和空间复杂度要求较高的程序，使用递归函数调用时要慎重。

递归函数必须定义一个停止条件，否则函数将永远递归下去。

【实例 6.4】利用递归调用求解 n 的阶乘。**（实例位置：资源包\TM\sl\6\4）**

阶乘的计算公式为：n! =1×2×3×…×n。以 n 等于 4 为例，4!等于 4*3!，3!等于 3*2!，2!等于 2*1!，1!等于 1。当计算 4!时，只要知道 3!；计算 3!时，只要知道 2!，以此类推。现在 1!为 1，可据此计算 2!，根据 2!可以计算 3!……根据上述算法，可以通过递归调用求解 n!。

本实例中，先定义一个利用递归调用求阶乘的函数，然后在主函数中定义一个变量保存用户输入的值，调用阶乘函数，计算用户输入数字的阶乘，最后输出结果。代码如下：

```
#include <iostream>
using namespace std;
long Fac(int n)                          //定义 Fac()函数，计算 n 的阶乘
{
    if(n==0)                             //如果 n 为 0，返回 1
        return 1;
    else                                 //如果 n 不为 0
        return n*Fac(n-1);               //返回当前数与前一个数的乘积，此处递归调用了函数自身
}
int main()
{
    int n ;
    long f;
    cout << "please input a number" << endl;
    cin >> n ;                           //输入一个自然数
    f=Fac(n);                            //调用 Fac()函数，求输入数的阶乘
    cout << "Result :" << f << endl;
}
```

程序中通过递归调用 Fac()函数计算 n 的阶乘。程序运行结果如图 6.5 所示。

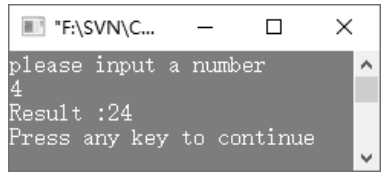


图 6.5　利用递归求解 n!

在上面的递归函数中，如果传递的参数很大，会导致堆栈溢出。因为每调用一次函数，系统都需要为函数参数分配一个堆栈空间。为避免内存溢出，建议用循环乘积的方式求解大数的阶乘 n!。

【实例 6.5】利用 for 循环求解 n 的阶乘。**（实例位置：资源包\TM\sl\6\5）**

本实例中，定义一个使用 for 循环求解阶乘的函数，在主函数中调用此函数。代码如下：

```
#include <iostream>
using namespace std;
typedef unsigned int UINT;               //自定义类型 UINT，用于表示自然数（即无符号整型数）
long Fac(UINT n)                         //定义 Fac()函数，计算 n 的阶乘
{
    long ret = 1;                        //定义 ret，表示阶乘计算结果
    for(int i=1; i<=n; i++)              //for 循环，累计乘积，计算阶乘
    {
        ret *= i;
    }
    return ret;                          //返回结果
```

```
}
int main()
{
        int n ;
        long f;
        cout << "please input a number" << endl;
        cin >> n ;                              //输入一个自然数
        f=Fac(n);                               //调用 Fac()函数，求输入数的阶乘
        cout << "Result :" << f << endl;
}
```

程序运行结果如图 6.6 所示。

下面来解决一个经典的数学问题——汉诺塔问题。该例题的算法过程有些复杂，但弄懂算法后，编写的代码却非常简单。这就是用程序解决复杂问题的魅力之所在。

【实例 6.6】汉诺（Hanoi）塔问题。**（实例位置：资源包\TM\sl\6\6）**

有 3 个立柱垂直矗立在地面，给这 3 个立柱分别命名为 A、B、C。开始时立柱 A 上有 64 个圆盘，大小不一，并且按从小到大的顺序依次摆放在立柱 A 上。现在要将立柱 A 上的 64 个圆盘移到立柱 C 上，并且每次只允许移动一个圆盘，在移动过程中始终保持大盘在下，小盘在上。

64 个圆盘的移动过于复杂，因此我们先来简化一下问题。假设要移动的圆盘只有 4 个，它们初始都位于立柱 A 上，圆盘按由上到下的顺序分别命名为 a、b、c、d，如图 6.7 所示。

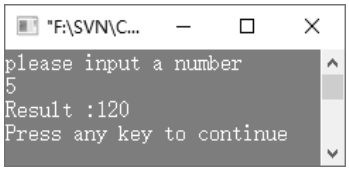


图 6.6　利用循环求解 n!

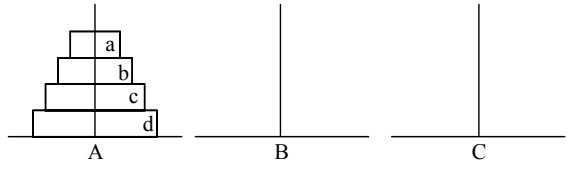


图 6.7　圆盘初始状态

首先，将 a、b 两个圆盘移动到立柱 C 上。移动过程需要借助立柱 B，移动顺序是 a→B，b→C，a→C，移动次数为 3 次。移动结果如图 6.8 所示。

接下来，将 a、b、c 3 个圆盘移动到立柱 B 上。移动顺序是 c→B，a→A，b→B，a→B，移动次数为 4 次，移动结果如图 6.9 所示。

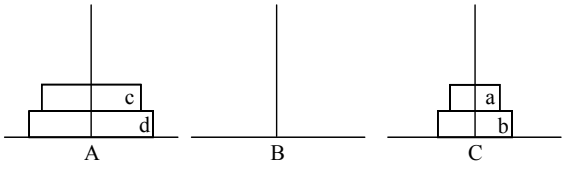


图 6.8　移动两个圆盘到立柱 C

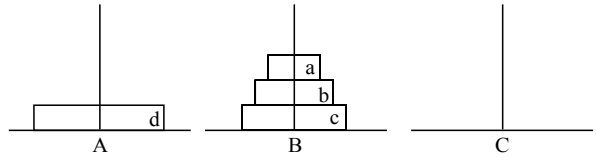


图 6.9　移动 3 个圆盘到立柱 B

最后，将 4 个圆盘都移动到立柱 C 上。移动顺序是 d→C，a→C，b→A，a→A，c→C，a→B，b→C，a→C。

先来总结一下移动规律：

（1）将 2 个圆盘移动到指定立柱，需要移动 3 次（即 2^2-1 次）。例如，将 a、b 圆盘由立柱 A 移到立柱 C 上，移动顺序为 a→B，b→C，a→C。

（2）将 3 个圆盘移动到指定立柱，需要移动 7 次（即 2^3-1 次）。例如，将 a、b、c 圆盘由立柱 A

移到立柱 B 上，移动顺序为 a→B，b→C，a→C，c→B，a→A，b→B，a→B。其中，前 3 次重复的是将 2 个圆盘移动到指定立柱的操作，后 4 次是将第 3 个圆盘移动到指定立柱的操作。

（3）将 4 个圆盘移动到指定立柱，需要移动 15 次（即 2^4-1 次）。同样，前 7 次重复的是将 3 个圆盘移动到指定立柱的操作，后 8 次是将第 4 个圆盘移动到指定立柱的操作。

（4）以此类推，当有 n 个圆盘时，要将它们移动到指定立柱，需要移动 2^n-1 次。

继续思考：移动过程中，如果将 a、b、c 3 个圆盘看成一个整体，则移动 4 个圆盘的过程和移动 2 个圆盘一样。同理，将 a、b 2 个圆盘看成一个整体，则移动 3 个圆盘的过程也像是在移动 2 个圆盘。因此，可以使用递归的思路解决 n 个圆盘的移动问题，整个移动过程分为 3 步。

（1）把立柱 A 上的 n−1 个圆盘移动到立柱 B 上。

（2）把立柱 A 上的 1 个圆盘移动到立柱 C 上。

（3）把立柱 B 上的 n−1 个圆盘移动到立柱 C 上。

现在我们试着用上述递归思路编写程序，具体代码如下：

```
#include <iostream>
using namespace std;
long lCount;
void move(int n,char a,char b,char c)              //定义 move()函数，将 n 个圆盘从立柱 a 移到立柱 c
{
      if(n==1)                                     //如果只有 1 个圆盘
            cout << "Times:" << ++lCount <<' '<< a << "->" << c << endl;
      else                                         //如果有多个圆盘
      {
            move(n-1,a,c,b);                       //递归调用 move()函数
            cout << "Times:" << ++lCount <<' '<< a << "->" << c <<endl;
            move(n-1,b,a,c);                       //递归调用 move()函数
      }
}
int main()
{
      int n ;
      lCount=0;                                    //lCount 表示移动次数
      cout << "please input a number" << endl;
        cin >> n ;                                 //输入圆盘数量
      move(n,'A','B','C');                         //调用 move()函数，将 n 个圆盘从立柱 A 移到立柱 C
}
```

程序运行结果如图 6.10 所示。这里输入数字 3，表示要移动 3 个圆盘，程序将输出 3 个圆盘的移动步骤，与我们前面分析的移动过程完全一致。

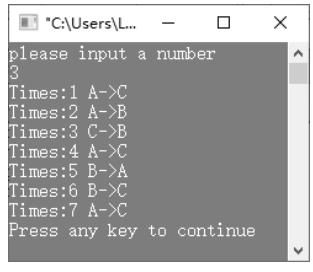


图 6.10　汉诺塔程序运行结果

编程训练（答案位置：资源包\TM\sl\6\编程训练\）

【训练 3】兔子繁殖问题　设一对大兔子每月生一对小兔子，每对新生兔在出生一个月后又可以生小兔子。假若兔子都不死亡，问：一对兔子一年后可变成多少对兔子？

【训练 4】走楼梯问题　楼梯有 10 阶台阶，上楼可以一步上 1 阶，也可以一步上 2 阶。编写程序，计算共有多少种不同的走法。

6.4 变量的作用域

作用域就是程序中变量的作用范围。局部变量的作用域是局部的，如函数体内；全局变量的作用域则是整个程序。

我们前面接触过的变量基本都是局部变量，这些变量在函数体内声明，无法被其他函数所使用。函数的形参也属于局部变量，作用范围仅限于函数内部的语句块。除此之外，在各类语句块（如 if 语句、while 语句、for 语句等）中定义的变量也是局部变量，作用范围仅限于语句块内部。

全局变量在函数体外声明，因此不属于某个函数，而属于整个 C++源文件，可在程序的任何位置访问它。全局变量可增加函数间的数据联系。例如，同一文件中的所有函数都能引用全局变量的值，因此如果在一个函数中改变了全局变量的值，就能影响到其他函数，相当于函数间有了一个数据通道。

说明

当内部作用域与外部作用域声明了同名变量时，内部作用域中的变量会屏蔽外部作用域中的变量。

【实例 6.7】变量的作用域。**（实例位置：资源包\TM\sl\6\7）**

本程序中，变量 iTotalCount 被定义两次，第一次定义出现在所有函数外，表示全局变量；第二次定义出现在 main()函数中，表示局部变量。细心体会两者的作用域，以及局部变量对全局变量的屏蔽作用。代码如下：

```
#include <iostream>
using namespace std;
int iTotalCount;                        //定义全局变量 iTotalCount
int GetCount();                         //声明 GetCount()函数
int main()
{
    int iTotalCount=100;                //定义局部变量 iTotalCount
    cout << iTotalCount << endl;        //输出局部变量 iTotalCount 的值
    cout << GetCount() << endl;         //输出调用 GetCount()函数后的返回值
}
int GetCount()                          //定义函数 GetCount()
{
    iTotalCount=200;                    //给全局变量赋值
    return iTotalCount;
}
```

程序运行结果如图 6.11 所示。main()函数中第一次输出的是局部变量 iTotalCount 的值 100，第二次输出的是调用 GetCount()函数后的返回值 200。

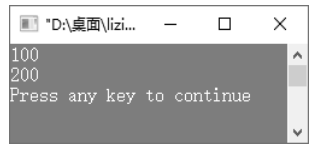


图 6.11 局部变量与全局变量

每个变量都有一定的生命周期。局部变量在函数调用时创建，在栈中分配内存，函数调用结束后销毁并释放。全局变量在程序开始时创建并分配空间，在程序结束时释放内存并销毁。

6.5　函 数 重 载

C++中，同一作用域内不能定义同名的变量，否则程序会编译出错；也不能定义同名的函数，否则会带来冲突问题。但实际开发中，经常需要处理功能几乎相似，仅传入参数不同（参数类型或参数个数不同）的问题，编写大量的函数并分别命名非常麻烦。为了提高代码的复用性和可读性，C++允许通过函数参数列表来识别同名函数，这就是函数重载。

所谓函数重载，是指多个函数具有相同的函数名，但参数列表不同，函数调用时，编译器根据传入的参数类型及参数个数来区分调用的是哪个函数。

函数重载是 C++中非常重要的特性，它可以提高程序的效率，使代码更加简洁、直观、高效。

【实例 6.8】使用重载函数实现两个数相加。**（实例位置：资源包\TM\sl\6\8）**

本实例中，定义 int Add(int x ,int y)函数，再定义 double Add(double x,double y)函数，这两个函数的名称都是 Add，仅参数类型不同。在主函数中调用这两个函数，输出两数相加结果。代码如下：

```
#include <iostream>
using namespace std;
int Add(int x ,int y)                          //定义函数 add()，返回两个整数的和
{
    cout << "调用 int add()函数" << endl;
    return x + y;
}
double Add(double x,double y)                  //重载函数 add()，返回两个双精度数的和
{
    cout << "调用 double add()函数" << endl;
    return x + y;
}
int main()
{
    int ivar = Add(5,2);                       //调用 int Add()函数
    cout <<"sum = " << ivar << endl;
    float fvar = Add(10.5,11.4);               //调用 double Add()函数
    cout <<"sum = " << fvar << endl;
    return 0;
}
```

程序运行结果如图 6.12 所示。程序中重载了函数 Add()，在 main()中调用时给出不同的实参类型，由编译器进行区分。语句“int ivar = Add(5,2);”的实参类型是整型，语句“float fvar = Add(10.5,11.4);”的实参类型是双精度，编译器可以区分它们，正确调用相应的函数。

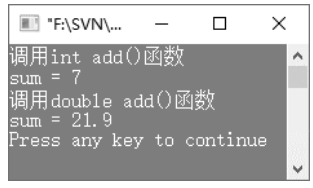


图 6.12　使用重载函数

在定义重载函数时，应注意函数的返回值类型不作为区分重载函数的一部分。例如，下面的重载函数是非法的。

```
int Add(int x ,int y)                          //定义一个重载函数
{
    return x + y;
}
double Add(int x,int y)                        //定义第二个重载函数
{
```

```
    return x + y;
}
```

编程训练（答案位置：资源包\TM\sl\6\编程训练\）

【训练 5】警匪大片中嫌疑人可选择的状态　电影中，当嫌疑人被警方抓捕时，警方都会对嫌疑人说“你有权保持沉默，但你说的每一句话都会成为呈堂证供”。使用方法的重载，在控制台上输出嫌疑人可选择的状态。

【训练 6】选择相同类型的数据　利用重载函数特性定义函数 concat，该函数接受两个相同类型（int、short、long、char*）的参数，然后打印它们的连接形式。例如：

```
concat(1,2); 输出 12
concat(“I miss”, “you”); 输出 I miss you
```

6.6 内 联 函 数

通过 inline 关键字可以把函数定义为内联函数，编译器会在每个调用该函数的地方展开一个函数的副本。

例如，创建一个内联函数 IntegerAdd 并进行调用，代码如下：

```
#include <iostream>
using namespace std;
inline int IntegerAdd(int x,int y);
int main()
{
    int a;
    int b;
    int iresult=IntegerAdd(a,b);
}
inline int IntegerAdd(int x,int y)
{
    return x+y;
}
```

IntegerAdd 函数被定义为内联函数，其在计算机中的执行过程如下：

```
int main()
{
    int a;
    int b;
    int iresult= a+b;
}
```

使用内联函数可以减少函数调用带来的开销（即程序文件中移动指针寻找调用函数地址带来的开销），但它只是一种解决方案，编译器可以忽略内联的声明。

建议在函数实现代码很简短或者调用该函数次数相对较少的情况下将函数定义为内联函数，内联函数通常定义一条返回语句，不能包含循环或者 switch 语句。例如，一个递归函数不能在调用时完全展开，一个 1000 行代码的函数也不可能在调用时展开，内联函数只能在优化程序时使用。在抽象数据类设计中，它对支持信息隐藏起主要作用。

如果某个内联函数要作为外部全局函数，即它将被多个源代码文件使用，那么就把它定义在头文件里，在每个调用该内联函数的源文件中包含该头文件，这种方法保证对每个内联函数只有一个定义，以防止在程序的生命期中引起无意的不匹配。

6.7　变量的存储类型

第 2 章中我们学习过变量的数据类型。除此之外，变量还分为 4 种存储类型，分别是 auto、static、register 和 extern。存储方式不同，变量的作用域和生存期也不同。生存期和作用域，从时间和空间两个不同的维度描述了一个变量。

6.7.1　auto 变量

auto 变量是动态存储变量，只在程序执行到它时才从栈区分配存储单元，使用完毕后立刻释放该存储单元。例如，函数形参即为 auto 变量，函数定义时并不会为其分配存储单元，函数调用时才会分配，调用完毕后会立即释放存储单元。auto 变量存放在动态存储区中。

auto 是 C++默认的存储类型。也就是说，凡未加存储类型说明的变量均为 auto 变量，且 auto 关键字可以省略。因此：

```
int i,j,k;
```

等价于：

```
auto int i,j,k;
```

auto 变量具有以下特点。

（1）auto 变量的作用域和生存期仅限于定义该变量的个体内。也就是说，在函数中定义的 auto 变量，只在该函数内有效，无法在函数外使用；在复合语句中定义的 auto 变量，只在该复合语句中有效，无法在复合语句外使用。例如：

```
int Show()
{
    auto int x,y;                    //变量 x, y 的作用域为整个 Show()函数
    if(true)
    {
        auto char ch;                //变量 ch 的作用域为 if 语句块
        cout << ch << endl;          //正确
        cout << x << endl;           //正确
    }
    cout << ch << endl;              //不可在 if 语句块外使用变量 ch，错误
    cout << x << endl;               //正确
}
```

（2）不同的个体（函数或复合语句）中允许使用同名变量而不会混淆。例如，函数内定义的 auto 变量可以与复合语句中定义的 auto 变量同名。

【实例 6.9】同名变量的“屏蔽”作用。**（实例位置：资源包\TM\sl\6\9）**

本实例中，定义两次 auto 变量 k，分别在 if 语句内外使用此变量进行计算，观察结果，思考 auto

变量的作用。代码如下：

```
#include <iostream>
using namespace std;
int main()
{
	auto int i,j,k;                          //第一次定义 k，作用域为整个 main()函数
	cout <<"input the number:" << endl;
	cin >> i >> j;
	k=i+j;                                   //令 k 等于 i、j 之和
	if( i!=0 && j!=0 )                       //if 语句
	{
		auto int k;                          //定义同名变量 k，作用域仅限 if 语句
		k=i-j;                               //令 k 等于 i、j 之差
		cout << "k :" << k << endl;          //输出 k 的值
	}
	cout << "k :" <<k << endl;               //退出 if 语句，返回 main()函数，输出变量 k 的值
}
```

程序运行结果如图 6.13 所示。k 为 auto 变量，第一次输出的是 i−j 的值，第二次输出的是 i+j 的值。

本例中，虽然变量名都为 k，但由于位于不同的作用域和拥有不同的生存期，其实是两个不同的变量。表面看来，就好像 if 语句内的变量“屏蔽”了外部的同名变量。

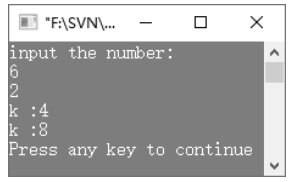


图 6.13　输出不同生存期的变量值

6.7.2　static 变量

static 变量是静态存储变量，定义变量时系统就会为其分配固定的存储单元，直至整个程序运行结束。6.4 节中我们接触过的全局变量即为 static 变量，它们存放在静态存储区中。

使用 static 关键字，可将变量声明成 static 变量。例如：

```
static int a,b;
static float x,y;
static int a[3]={0,1,2};
```

static 变量属于静态存储方式，具有以下特点。

（1）无论是静态全局变量，还是静态局部变量，其生存期均为整个 C++源程序运行期间。

（2）静态全局变量在函数外定义，作用域是整个 C++源程序，即可在程序任意位置使用它。

（3）静态局部变量的作用域与 auto 变量类似，在函数内定义，只能在函数中使用。离开函数后，尽管其值一直存在，但无法被使用。当再次调用函数时，可以继续使用它。

（4）编译器会为静态局部变量赋予 0 值。

【实例 6.10】记录点击量。**（实例位置：资源包\TM\sl\6\10）**

建立函数 click()，用于记录用户点击量。函数中定义一个静态变量 sum，用于记录点击次数。调用 5 次 click()，查看此时点击量是多少。代码如下：

```
#include <iostream>
using namespace std;
void click()                                 //定义 click()函数，记录用户点击量
{
	static int sum = 0;                      //定义静态局部变量 sum，初始点击次数为 0
```

```
    sum = sum +1;                          //点击次数加 1
    cout<<"此时点击量"<<sum<<endl;          //输出点击量
}

int main()
{   //调用 5 次 click()函数，模拟 5 次点击行为
    click();
    click();
    click();
    click();
    click();
    return 0;
}
```

程序运行结果如图 6.14 所示。sum 是静态局部变量，其值自定义起，会一直存在。所以 5 次调用 click()函数时，变量 sum 每次都在原来的数值上加 1。

第 5 行代码中，如果去除 static 关键字，则程序运行结果如图 6.15 所示。此时变量 sum 为动态存储形式，每次调用 click()函数后，sum 的值都被释放，再次调用时重新被赋初值 0。因此，5 次调用 click()函数，输出的结果都是 1。

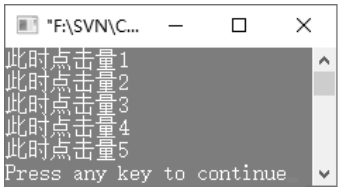


图 6.14　使用 static 变量实现累加

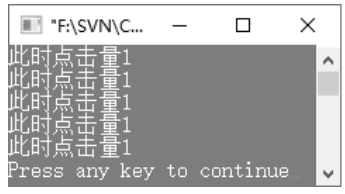


图 6.15　程序运行结果

6.7.3　register 变量

变量通常存储在内存中，当某个变量需要高频读写时，就需要重复访问内存。为了提高效率，C++允许将变量声明为 register（寄存器）类型，这种变量将局部变量的值存放在 CPU 寄存器中，使用时不再访问内存，而直接从寄存器中读写。

register 变量的说明如下。

（1）register 变量属于动态存储方式，凡需要采用静态存储方式的量不能定义为 register 变量。

（2）编译程序会自动决定哪个变量使用寄存器存储。register 变量起到程序优化的作用。

6.7.4　extern 变量

一个 C++程序通常包含多个源文件。由于 C++文件中定义的变量和函数，只能被本文件中的函数调用。所以，要想调用其他源文件中的某个全局变量，就需要使用 extern 关键字声明该变量。

例如，在 Sample1.cpp 源文件中定义了 3 个全局变量 a、b、c，在 Sample2.cpp 源文件中想使用 Sample1.cpp 源文件中的全局变量 a、b、c，就需要使用 extern 关键字进行声明，程序代码如下：

```
extern int a,b;                            //声明外部变量 a,b
extern char c;                             //声明外部变量 c
func (int x,y)
{
```

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

在 Sample2.cpp 源文件中，编译系统不会再为全局变量 a、b、c 分配内存空间。在 Sample2.cpp 源文件中改变全局变量 a、b、c 的值，在 Sample1.cpp 源文件中输出值也会发生变化。

编程训练（答案位置：资源包\TM\sl\6\编程训练\）

【训练 7】停车场还剩多少停车位　创建函数 park()，用于记录停车场中停车位的数量，每进入一辆车，则执行此方法一次。停车场共有 30 个停车位，进入 4 辆车之后，计算停车场还剩多少个停车位。

【训练 8】数据清洗　定义 mul()函数，在函数中使用 auto 变量修饰变量 a，并对此变量每次都乘 2，调用 5 次，查看结果，发现每次都得到同一个结果，说明每次都将数据进行了清洗重置。结果如下：

```
第 1 次调用：6
第 2 次调用：6
第 3 次调用：6
第 4 次调用：6
第 5 次调用：6
```

本章小结

本章主要介绍函数的使用，调用函数要了解函数的返回值、参数以及调用方式。变量的作用域和函数有关，函数的递归调用可以帮助开发人员设计思路明晰的程序，内联函数可以提高程序的运算效率，函数重载则解决了代码复用中函数名冲突的问题。

6.8　实践与练习

答案位置：（资源包\TM\sl\6\实践与练习\）

综合练习 1：模拟 12306 抢票系统　12306 抢票系统，每售出一张票，全国各地的系统显示都会同步减少一张票。利用全局变量模拟 12306 抢票系统，输出效果如下：

```
始发地：上海　目的地：长春　时间：2021 年 4 月 10 日 16：20 出发
3 个城市剩余的票数分别为：
上海的 12306 系统剩余票数: 99 张
北京的 12306 系统剩余票数: 99 张
深圳的 12306 系统剩余票数: 99 张
我抢到一张票之后剩余票数：98 张
我抢到一张票之后 3 个城市剩余的票数分别为：
上海的 12306 系统剩余票数: 98 张
北京的 12306 系统剩余票数: 98 张
深圳的 12306 系统剩余票数: 98 张
```

综合练习 2：为和尚写诗　自定义一个 poetry()函数，为和尚写一首诗，诗句如下：

```
空门有路不知处
头白齿黄犹念经
何年饮着声闻酒
迄至如今醉未醒
```

综合练习 3：确定女主角　某导演有一个剧本，需要找演员来演对应的角色。利用函数的实参和形参编写代码，实现为剧本选女主角的功能。效果如下：

```
导演选定女主角是：Lucy
→*→*→*→*→*→*→*→*→*→*→*→*
     Lucy 开始参演李美丽角色
→*→*→*→*→*→*→*→*→*→*→*→*
```

综合练习 4：递归求年龄　使用递归调用求年龄。某一天，甲乙丙丁戊 5 个人坐在一起聊天，大家猜戊的年龄，他说比丁大 2 岁；问丁的年龄，他说比丙大 2 岁；问丙的年龄，他说比乙大 2 岁；问乙的年龄，他说比甲大 2 岁；问甲的年龄，他说他 10 岁。编写程序，求戊的年龄。效果如下：

```
--------------------------------
   戊的年龄是：18 岁
--------------------------------
```

综合练习 5：你的心跳正常吗？　把函数作为参数使用，编写程序，判断输入的心跳数是否是正常。当心跳数在 60～100 次/min，显示心跳正常，运行结果如图 6.16 所示；当心跳数大于 100 次/min 或小于 60 次/min 时，显示心跳不正常。运行结果如图 6.17 所示。

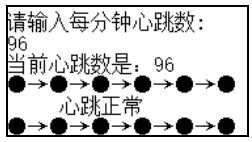


图 6.16　心跳正常

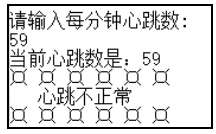


图 6.17　心跳不正常