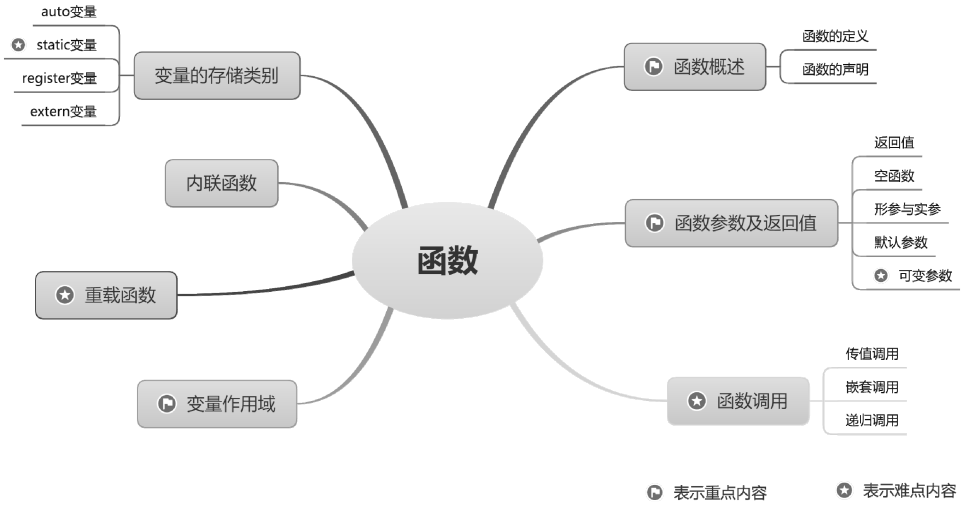
第 6 章

函　　数

程序是由函数组成的，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的结构设计得更合理。

本章知识架构及重难点如下：



6.1　函 数 概 述

函数有别人写好的存放在库里的库函数，也有开发人员自己写的自定义函数。函数根据功能可以分为字符函数、日期函数、数学函数、图形函数、内存函数等。一个程序可以只有一个主函数，但不能没有函数。

6.1.1　函数的定义

定义一个函数的一般形式如下：

```
类型标识符 函数名(形式参数列表)
{
```

```
    变量的声明
    语句
}
```

类型标识符用来标识函数的返回值类型，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。类型标识符可以是整型、字符型、指针型或对象的数据类型。

形式参数列表是由各种类型变量组成的列表，各参数之间用逗号分隔。在进行函数调用时，通过主调函数的实际参数列表对变量进行赋值。

关于函数定义的说明如下：

（1）形式参数列表可以为空。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。

6.1.2 函数的声明

调用一个函数前，必须先声明函数的返回值类型和参数类型。例如：

```
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：

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

下面通过实例介绍如何在程序中声明、定义和调用函数。

【实例 6.1】 编写 3 个函数：做饭，钓鱼，写诗。**（实例位置：资源包\TM\sl\6\1）**

在本实例中，通过定义函数完成某种特定的功能。为了表示函数完成的功能，这里使用输出的信息进行表示。希望读者通过这个实例对函数的定义有一个更为具体的认识。

```
#include<stdio.h>                    /*包含头文件*/
void Cook();                         /*声明函数 Cook*/
void Fish();                         /*声明函数 Fish*/
void Poem();                         /*声明函数 Poem*/
int main()                           /*主函数 main*/
{
        Cook();                      /*调用函数 Cook*/
        Fish();                      /*调用函数 Fish*/
        Poem();                      /*调用函数 Poem*/
        return 0;                    /*程序结束*/
}
void Cook()                          /*自定义 Cook 函数*/
{
```

```
        printf("会做饭\n");
}
void Fish()                                    /*自定义 Fish 函数*/
{
        printf("会钓鱼\n");
}
void Poem()                                    /*自定义 Poem 函数*/
{
        printf("会写诗\n");
}
```

程序运行结果如图 6.1 所示。

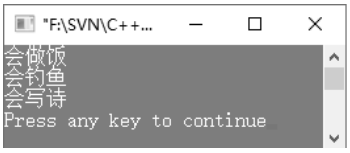


图 6.1　程序运行结果

程序定义和声明了函数 Cook、Fish、Poem，并进行了调用，通过函数中的输出语句进行输出。

编程训练（答案位置：资源包\TM\sl\6\编程训练\）

【训练 1】 定义两数交换函数　编写函数 fun，实现两个数值的交换功能。

【训练 2】 打印新年菜单　新年是家家户户的团圆节日，每家都张灯结彩迎接新年，团圆饭也是必不可少的，在新年前几天就会张罗菜单，准备食材。定义一个函数，功能是打印 2021 年新年菜单，并在主函数中调用该函数，运行结果如下：

```
-----新年菜单如下-------
 1. 凉菜
 2. 红烧鱼
 3. 水煮虾
 4. 酱猪蹄
 5. 红烧排骨
 6. 孜然牛肉
 7. 木须柿子
 8. 水煮肉片
------------------------
```

6.2　函数参数及返回值

6.2.1　返回值

返回值是指函数被调用之后，执行函数体中的程序段取得的并返回给主调函数的值，函数的返回值通过 return 语句返回给主调函数。return 语句的一般形式如下：

```
return (表达式);
```

语句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中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）在函数中允许有多个 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();          //不能进行赋值
SetIndex(ShowIndex);     //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。

6.2.2 空函数

没有参数和返回值，即函数的作用域为空的函数就是空函数。

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

调用空函数时，程序不会执行任何操作。例如，在 main 函数中调用 setWorkSpace 函数时，该函数没有起到任何作用。

```
void setWorkSpace(){ }
void main()
{
  setWorkSpace();
}
```

空函数的存在有什么意义呢？在程序设计中各功能模块通常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则在以后需要时陆续补充。在编写程序的开始阶段，可以在将来准备扩充功能的地方写上一个空函数，这些函数没有开发完成，先占一个位置，以后用一个编好的函数代替它。这样做，程序的结构清楚，可读性好，以后扩充新功能方便，对程序结构影响不大。

误区警示

void 并不意味着函数无类型，void 本身就是 C 语言的一种类型名，只不过这种类型的值为空集。函数类型是指函数名具有的类型，函数返回值类型是指函数调用表达式的类型。而 void 表示的是返回值为空的类型，并非是无类型。

6.2.3　形参与实参

函数定义时如果参数列表为空，说明函数是无参函数；如果不为空，则称为带参函数。带参函数中的参数在函数声明和定义时被称为形式参数，简称形参，在函数被调用时被赋予具体值，具体的值被称为实际参数，简称实参。形参与实参如图 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

实参与形参的个数应相等，类型应一致，并按顺序对应，函数被调用时会一一传递数据。

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项传递，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

实际参数不但可以是常量、变量、数组或指针等，也可以是表达式。

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到调用函数，不妨在定义函数时为参数设置一个默认值。这样在调用函数时可以省略一些参数，此时程序将采用默认值作为函数的实际参数。下面的程序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;                    //输出信息
}
```

【实例 6.2】 展示 2022 冬奥会口号。**（实例位置：资源包\TM\sl\6\2）**

输出两行字符串：一行是使用默认值作为函数实参，一行是将字符串作为函数实参。程序代码如下：

```
#include <iostream>
using namespace std;
void OutputInfo(const char *pchData = "纯洁的冰雪激情的约会")
{
    cout << pchData << endl;                    //输出信息
```

```
}
void main()
{
    OutputInfo();                                   //利用默认值作为函数实际参数
    OutputInfo("Beijing 2022  冬奥会!");            //直接传递实际参数
}
```

程序运行结果如图 6.3 所示。

图 6.3　调用默认参数的函数

在定义函数默认值参数时，如果函数具有多个参数，应保证默认值参数出现在参数列表的右方，没有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。

```
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)                                      //x 与 y 进行比较
        x = y;                                      //赋值
    if (x < z)                                      //x 与 z 进行比较
        x = z;                                      //赋值
    return x;                                       //返回 x
}
```

程序中默认值参数 y 出现在非默认值参数 z 的左方，导致了编译错误，正确的做法是将默认值参数放置在参数列表的右方。例如：

```
int GetMax(int x,int y ,int z=10)                   //定义默认值参数
{
    if (x < y)                                      //x 与 y 进行比较
        x = y;                                      //赋值
    if (x < z)                                      //x 与 z 进行比较
        x = z;                                      //赋值
    return x;                                       //返回 x
}
```

6.2.5　可变参数

库函数 printf 是一个可变参数的函数，它的参数列表中可使用省略号“...”。printf 函数原型的格式如下：

```
_CRTIMP int_cdecl printf(const char *, ...);
```

省略号代表的含义是函数的参数不固定，可以传递一个或多个参数。对于 printf 函数，可以输出一项信息，也可以同时输出多项信息，例如：

```
printf("%d\n",2008);                                //输出一项信息
printf("%s-%s-%s\n","Beijing","2022","冬奥会！");   //输出多项信息
```

声明可变参数的函数和声明普通函数一样，只是参数列表中有一个省略号“...”，例如：

```
void OutputInfo(int num,...)                        //定义省略号参数的函数
```

对于可变参数的函数，在定义函数时需要一一读取用户传递的实际参数。可以使用 va_list 类型和 va_start、va_arg、va_end 3 个宏读取传递到函数中的参数值，同时，使用可变参数需要引用 STDARG.H 头文件。下面以一个具体的示例介绍可变参数的函数的定义及使用。

【实例 6.3】 关灯、开灯。**（实例位置：资源包\TM\sl\6\3）**

设计一个函数，当传递参数为 false 时，函数输出“关灯”；当不传递参数或传递参数为 true 时，函数输出“开灯”。代码如下：

```
#include <iostream>
using namespace std;
void Open(bool bOpen = true)
{
   if(bOpen)
   {
        cout << "开灯" << endl;
   }
   else
   {
        cout << "关灯" << endl;
   }
}
int main(int argc, char* argv[])
{
       Open(false);
       Open();
       Open(true);
       return 0;
}
```

程序运行结果如图 6.4 所示。

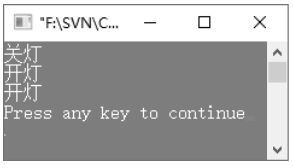


图 6.4　定义省略号形式的函数参数

编程训练（答案位置：资源包\TM\sl\6\编程训练\）

【训练 3】 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 4】 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　函 数 调 用

声明函数后需要在源代码中调用该函数。整个函数的调用过程称为函数调用。标准 C++是一种强制类型检查的语言，在调用函数前，必须把函数的参数类型和返回值类型告知编译器。

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（是库函数或用户自己定义的函数）。

（2）如果使用库函数，还需要将库函数对应的头文件引入，这需要使用预编译指令#include。

（3）如果使用用户自定义函数，一般还应在主调函数中对被调用的函数做声明。

注意

C++和 C 不同，简单地用不同的原型说明同一个函数是无法通过 C++语法检查的。

6.3.1 传值调用

主调函数和被调函数之间存在着数据传递关系。换句话说，主调函数将实参数值复制到被调用函数的形参处，这种调用方式称为传值调用。如果传递的实参是结构体对象，值传递方式的效率是低下的，可以通过传递指针或使用变量的引用来替换传值调用。

传值调用是函数调用的基本方式，下面来看一个例子。

【实例 6.4】 交换两个数值。**（实例位置：资源包\TM\sl\6\4）**

本实例中使用传值调用，实现两个数值的交换。代码如下：

```
#include <iostream.h>
void swap(int a,int b)
{
    int tmp;
    tmp=a;
    a=b;
    b=tmp;
}
void main()
{
    int x,y;
    cout << "输入两个数" << endl;
    cin >> x;
    cin >> y;
    if(x<y)
        swap(x,y);
    cout << "x=" << x <<endl;
    cout << "y=" << y <<endl;
}
```

程序运行结果如图 6.5 所示。程序本意是想实现当 x 小于 y 时交换 x 和 y 的值，但结果并没有实现，为什么呢？原因是调用 swap 函数时复制了变量 x 和 y 的值，而并非变量本身。如果将函数 swap 在调用处展开，x 和 y 就可以实现交换。例如，代码修改如下：

```
#include <iostream>
using namespace std;
void main()
{
    int x,y;
    cout << "输入两个数" << endl;
    cin >> x;
```

```
    cin >> y;
    int tmp;
    if(x<y)
    {
        tmp=x;
        x=y;
        y=tmp;
    }
    cout << "x=" << x <<endl;
    cout << "y=" << y <<endl;
}
```

程序运行结果如图 6.6 所示。

图 6.5　使用传值调用

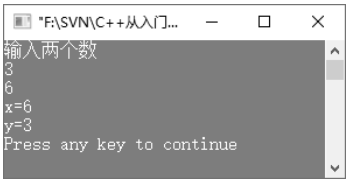


图 6.6　展开函数调用

注意，后一段程序代码是开发人员模拟函数调用时展开 swap 函数，而前一段代码中函数的调用是由编译器来完成的，但不是真的展开 swap 函数，而是移至 swap 函数处执行，执行过程类似于展开。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，形参的值发生改变，却无法再传递回来，因此实参的值不会发生变化。要想通过函数调用的方式实现交换变量的值，可以通过使用指针传递地址和变量引用的方式实现，这在后面的章节会讲解。

6.3.2　嵌套调用

在自定义函数中调用其他自定义函数，这种调用方式称为嵌套调用，例如：

```
#include <iostream>
using namespace std;
void   ShowMessage()                              /*定义函数*/
{
    cout <<"The ShowMessage function" << endl;
}
void   Display()
{
    ShowMessage();                                /*嵌套调用*/
}
void main()
{
    Display();
}
```

在函数嵌套调用时要注意，不要在函数体内定义函数。例如，如下代码是错误的：

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

嵌套调用对调用的层数是没有要求的，但个别的编译器可能会有一些限制，使用时应注意。

6.3.3 递归调用

直接或间接调用自己的函数称为递归函数（recursive funciton）。

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。其缺点是：递归程序的运行效率比较低，无论是从时间角度还是从空间角度都比非递归程序差。对于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 6.5】 汉诺（Hanoi）塔问题。**（实例位置：资源包\TM\sl\6\5）**

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，如图 6.7 所示。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。

分析程序：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。

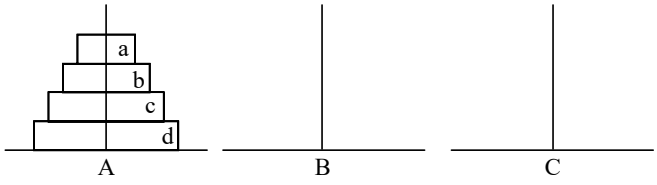


图 6.7　圆盘原始状态

首先，将 a、b 两个圆盘移动到立柱 C 上。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。

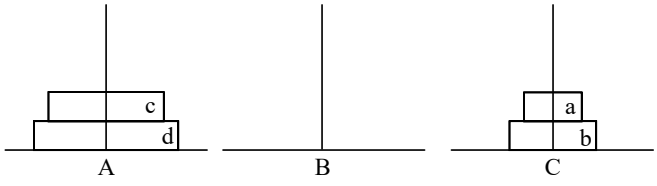


图 6.8　移动两个圆盘到 C

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。

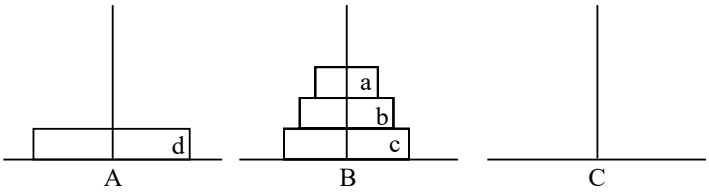


图 6.9　移动 3 个圆盘到 B

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。

总结一下：

将 2 个圆盘移动到指定立柱，需要移动 3 次，分别是 a→B，b→C，a→C。

将 3 个圆盘移动到指定立柱，需要移动 7 次，分别是 a→B，b→C，a→C，c→B，a→A，b→B，a→B。其中，前 3 次移动重复的是将 2 个圆盘移动到指定立柱，后 4 次移动是将第 3 个圆盘移动到指定立体。

将 4 个圆盘移动到指定立柱，总共需要移动 15 次。同样，前 7 次重复的是将 3 个圆盘移动到指定立体，后 8 次是将第 4 个圆盘移动到指定立柱。

当有 n 个圆盘时，要将它们移动到指定立柱，移动次数可以总结为 2^n-1 次。

在移动过程中，可以将 a、b、c 3 个圆盘看成一个，移动 4 个圆盘的过程就像是在移动 2 个圆盘。还可以将 a、b、c 3 个圆盘中的 a、b 2 个圆盘看成一个，移动 3 个圆盘也像是在移动 2 个圆盘。因此，可以使用递归的思路来移动 n 个圆盘。

移动 n 个圆盘的过程可以分成 3 个步骤：

（1）把 A 上的 n−1 个圆盘移动到 B 上。

（2）把 A 上的 1 个圆盘移动到 C 上。

（3）把 B 上的 n−1 个圆盘移动到 C 上。

程序代码如下：

```
#include <iostream>
using namespace std;
long lCount;
void move(int n,char A,char B,char C)                    //将 n 个圆盘从 A 立柱借助 B 立柱，移动到 C 立柱上
{
    if(n==1)
        cout << "Times:" << ++lCount << A << "->" << C << endl;
    else
    {
        move(n-1,x,z,y);
        cout << "Times:" << ++lCount << A << "->" << C <<endl;
        move(n-1,y,x,z);
    }
}
void main()
{
    int n ;
    lCount=0;
```

```
    cout << "please input a number" << endl;
    cin >> n ;
    move(n,'a','b','c');
}
```

程序运行结果如图 6.10 所示。

输入数字 3，表示移动 3 个圆盘，程序打印出移动 3 个圆盘的步骤。可见，与前面的分析过程完全一致。

【实例 6.6】 求 n 的阶乘。**（实例位置：资源包\TM\sl\6\6）**

本实例中，首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，再调用阶乘函数，求用户输入数字的阶乘，最后打印结果。具体代码如下：

```
#include <iostream>
using namespace std;
long Fac(int n)
{
    if(n==0)
        return 1;
    else
        return n*Fac(n-1);
}
void main()
{
    int n ;
    long f;
    cout << "please input a number" << endl;
    cin >> n ;
    f=Fac(n);
    cout << "Result :" << f << endl;
}
```

程序运行结果如图 6.11 所示。

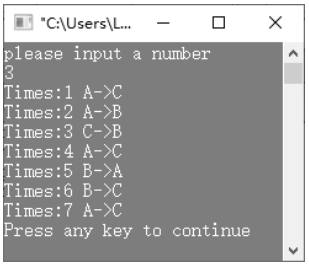


图 6.10　程序运行结果

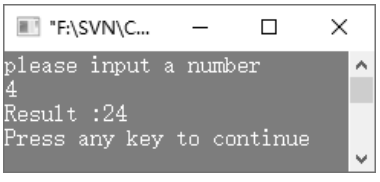


图 6.11　计算阶乘的运行结果

程序中 Fac 函数实现了计算 n 的阶乘。以 n 等于 4 为例，4!等于 4*3!，3!等于 3*2!，…1!等于 1。当计算 4 的阶乘时，只要知道 3 的阶乘，4*3!等于 4!。同理，计算 3 的阶乘，只要知道 2 的阶乘，以此类推。1 的阶乘为 1，知道了 1 的阶乘，就可以计算 2 的阶乘，知道 2 的阶乘就可以计算 3 的阶乘……

在上面的递归函数中，如果传递一个很大的数作为参数，会导致堆栈溢出，因为每调用一个函数，系统会为函数的参数分配堆栈空间。对于上述的递归函数 Fac，完全可以用连续乘积的方式实现。

【实例 6.7】 利用循环求 n 的阶乘。**（实例位置：资源包\TM\sl\6\7）**

在实例 6.6 中，是使用递归调用实现的阶乘。本实例要求定义一个使用循环求阶乘的函数，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
long Fac(const UINT n)                  //定义函数
{
    long ret = 1;                       //定义结果变量
    for(int i=1; i<=n; i++)             //累计乘积
    {
        ret *= i;
    }
    return ret;                         //返回结果
}
void main()
{
    int n ;
    long f;
    cout << "please input a number" << endl;
    cin >> n ;
    f=Fac(n);
    cout << "Result :" << f << endl;
}
```

程序运行结果如图 6.12 所示。

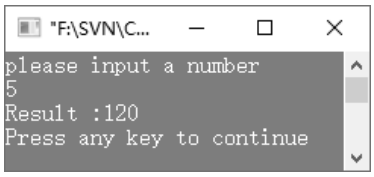


图 6.12　利用循环求 n 的阶乘的运行结果

编程训练（答案位置：资源包\TM\sl\6\编程训练\）

【训练 5】 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 6】 你想如何走楼梯　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　变量作用域

根据变量声明的位置，可以将变量分为局部变量和全局变量。在函数体内定义的变量称为局部变量，在函数体外定义的变量称为全局变量。例如：

```
#include <iostream>
using namespace std;
int iTotalCount;                        //全局变量
int GetCount();
void main()
{
```

```
    int iTotalCount=100;        //局部变量
    cout << iTotalCount << endl;
    cout << GetCount() << endl;
}
int GetCount()
{
    iTotalCount=200;            //给全局变量赋值
    return iTotalCount;
}
```

程序运行结果如图 6.13 所示。

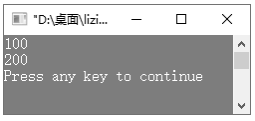


图 6.13　局部变量与全局变量的运行结果

变量都有它的生命期，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；局部变量是在函数调用时创建，并在栈中分配内存，在函数调用结束后销毁并释放。

6.5　重 载 函 数

定义同名的变量，程序会编译出错；定义同名的函数，也会带来冲突的问题。C++中使用了名称重组技术，通过函数的参数类型来识别函数。所谓重载函数，是指多个函数具有相同的函数名，而参数类型或参数个数不同。函数调用时，编译器以参数的类型及个数来区分调用哪个函数。下面的实例定义了重载函数。

【实例 6.8】 使用重载函数。**（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，这两个函数名都是 Add，在主函数中调用这两个函数，输出结果。代码如下：

```
#include <iostream>
using namespace std;
int Add(int x ,int y)                    //定义第一个重载函数
{
    cout << "int add" << endl;           //输出信息
    return x + y;                        //设置函数返回值
}
double Add(double x,double y)            //定义第二个重载函数
{
    cout << "double add" << endl;        //输出信息
    return x + y;                        //设置函数返回值
}
int main()
{
    int ivar = Add(5,2);                 //调用第一个 Add 函数
    float fvar = Add(10.5,11.4);         //调用第二个 Add 函数
    return 0;
}
```

程序运行结果如图 6.14 所示。

程序中定义了两个相同函数名的函数 Add，在 main 调用 Add 函数时实参类型不同，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，编译器可以区分这两个函数，正确调用相应的函数。

图 6.14　使用重载函数的运行结果

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 7】 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 8】 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。

在下面的程序中创建一个内联 IntegerAdd 函数，并进行调用。

```
#include <iostream>
using namespace std;
inline int IntegerAdd(int x,int y);
void 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：

```
void 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（在程序所在文件内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 变量的存储类别

存储类别是变量的属性之一，C++语言中定义了 4 种变量的存储类别，分别是 auto 变量、static 变量、register 变量和 extern 变量。变量存储方式不同会使变量的生存期不同，生存期表示变量存在的时间。生存期和变量作用域是从时间和空间两个不同的角度来描述变量的特性。

静态存储变量通常在变量定义时就分配固定的存储单元并一直保持不变，直至整个程序结束。前面讲过的全局变量即属于此类存储方式，它们存放在静态存储区中。动态存储变量是在程序执行过程中使用它时才分配存储单元，使用完毕立即将该存储单元释放。前面讲过的函数的形式参数，在函数定义时并不给形式参数分配存储单元，只是在函数被调用时才予以分配，调用函数完毕立即释放，此类变量存放在动态存储区中。

6.7.1 auto 变量

auto（自动）变量是C ++语言程序中默认的存储类型。函数内未加存储类型说明的变量均视为 auto 变量，也就是说，auto 变量可省去关键字 auto。例如：

```
{
    int i,j,k;
    ...
}
```

等价于：

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

```
    ...
}
```

auto 变量具有以下特点。

（1）auto 变量的作用域仅限于定义该变量的个体内。在函数中定义的 auto 变量，只在该函数内有效；在复合语句中定义的 auto 变量，只在该复合语句中有效，例如：

```
int Show()
{
    auto int x,y;
    if(true)
    {
        auto char ch;
        cout << ch << endl;          //正确
        cout << x << endl;           //正确
    }
    cout << ch << endl;              //错误
    cout << x << endl;               //正确
}
```

（2）auto 变量属于动态存储方式，变量分配的内存在栈中，当函数调用结束后，自动变量的值被释放。同样在复合语句中定义的 auto 变量，在退出复合语句后也不能再使用，否则将引起错误。

（3）auto 变量的作用域和生存期都局限于定义它的个体内（函数或复合语句内），因此不同的个体中允许使用同名的变量而不会混淆。即使在函数内定义的 auto 变量也可与该函数内部的复合语句中定义的 auto 变量同名。

【实例 6.9】 输出不同生命期的变量值。**（实例位置：资源包\TM\sl\6\9）**

本实例中，定义 auto 变量，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;
void main()
{
    auto int i,j,k;
    cout <<"input the number:" << endl;
    cin >> i >> j;
    k=i+j;
    if( i!=0 && j!=0 )
    {
        auto int k;
        k=i-j;
        cout << "k :" << k << endl;      //输出变量 k 的值
    }
    cout << "k :" <<k << endl;           //输出变量 k 的值
}
```

程序运行结果如图 6.15 所示。

程序两次输出变量 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。虽然变量名都为 k，但其实是两个不同的变量。

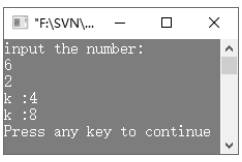


图 6.15　输出不同生命期的变量值

6.7.2 static 变量

在声明变量前加关键字 static，可以将变量声明成 static（静态）变量。静态局部变量的值在函数调用结束后不消失，静态全局变量只能在本源文件中使用。例如，声明变量为 static 变量：

```
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，它具有以下特点。

（1）static 变量在函数内定义，在程序退出时释放，即在程序整个运行期间都不释放，也就是说它的生存期为整个源程序。

（2）static 变量的作用域与 auto 变量相同，在函数内定义在函数内使用，尽管该变量还继续存在，但不能使用。只有再次调用定义它的函数时，才可继续使用。

（3）编译器会为静态局部变量赋予 0 值。

下面通过实例介绍 static 变量的用法。

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

建立函数 click()，用于记录用户点击率。函数中定义 static 变量 sum=0，每次调用此方法，sum 的值都会加 1，并输出此时 sum 的值。调用 5 次 click()，查看此时点击量是多少。具体代码如下：

```
#include <iostream>
using namespace std;
void click()
{
      static int sum = 0;                          /*定义 static 整型变量*/
      sum = sum +1;                                /*变量加 1*/
      cout<<"此时点击率"<<sum<<endl;               /*显示结果*/
}

int main()
     {    /*调用 5 次 click()函数*/
     click();
     click();
     click();
     click();
     click();
      return 0;
}
```

程序运行结果如图 6.16 所示。

程序中 sum 是静态局部变量，每次调用函数 click 时，静态局部变量 sum 都保存了前次被调用后留下的值，所以调用 5 次时，变量 sum 每次都加 1，而不是每次都是 1。

如果去除 static 关键字，则程序运行结果如图 6.17 所示。

当调用 5 次时，变量 sum 每次都是 1。变量 sum 不再使用静态存储区空间，每次调用后变量 sum 的值都被释放，再次调用时 sum 的值为初始值 0。

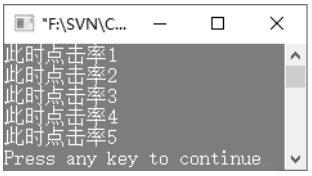


图 6.16　使用 static 变量实现累加

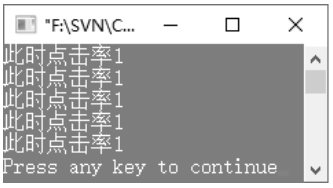


图 6.17　程序运行结果

6.7.3　register 变量

变量的值通常存放在内存中，当需要对一个变量频繁读写时，反复访问内存储器将花费大量的存取时间。为提高效率，C++语言允许将变量声明为 register（寄存器）变量，这种变量将局部变量的值存放在 CPU 中的寄存器中，使用时不需要访问内存，而直接从寄存器中读写。register 变量的说明符是 register。

对 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

在一个源文件中定义的变量和函数只能被本文件中的函数调用。一个 C++程序通常有许多源文件。在使用其他源文件的全局变量时，只需要在本源文件中使用 extern 关键字声明这个变量即可，示例如下。

在 Sample1.cpp 源文件中定义全局变量 a、b、c，程序代码如下：

```
int a,b;                              /*外部变量定义*/
char c;                               /*外部变量定义*/
void main()
{
      cout << a << endl;
      cout << b << endl;
      cout << c << endl;
}
```

在 Sample2.cpp 源文件中要使用 Sample1.cpp 源文件中全局变量 a、b、c，程序代码如下：

```
extern int a,b;                       /*外部变量说明*/
extern char c;                        /*外部变量说明*/
func (int x,y)
{
      cout << a << endl;
      cout << b << endl;
      cout << c << endl;
}
```

在 Sample2.cpp 源文件中，编译系统不再为全局变量 a、b、c 分配内存空间，而是改变全局变量 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 9】 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 10】 数据清洗　定义一个 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
```

6.8 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。

6.9 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 6.18 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 6.19 所示。

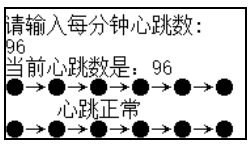


图 6.18　心跳正常

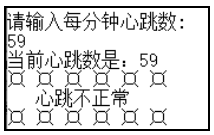


图 6.19　心跳不正常