第 4 章　函数

针对一个规模比较大的实际问题编写程序前，通常要按问题的逻辑关系将问题合理划分成功能相对独立的模块。功能模块在程序设计中被称为函数。函数有一定的定义格式和调用格式。所以，要编写好函数，必须清楚函数如何定义；要用好函数，则必须把握函数的调用机制。本章介绍 C++函数的编程机制以及函数的应用。

函数的定义与调用
函数的参数传递
函数的嵌套调用与递归调用
内联函数
变量和函数的属性
函数模板
C++常用系统函数

实际应用问题的程序一般是比较复杂的且代码较长，这样的大程序不可能完全由一个人从头至尾地完成，更不能把所有的程序代码都放在一个主函数中。通常是把这个大的程序分割成一些功能相对独立而且便于管理和阅读的小块程序，甚至这些小块程序又可分成若干更细的小块，逐步细化。这就把原来大问题的求解分解成若干小问题的求解，使整个问题的程序从功能上看结构清晰，便于解决问题。同时带来的好处是方便了程序编制员和阅读者。

把实现某一特定功能的相关语句按某种格式组织在一起形成一个程序块，并给程序块起一个相应的名称，这样的程序块就称为函数（function），而给程序块所起的相应名称称作函数名。函数有时也被称作例程或过程。

一个函数被使用时就是这个函数被调用的时候，函数调用通过调用语句实现，调用语句需要指定被调用函数的名字和调用函数所需要的信息（参数）。调用语句所在的函数称调用函数，又称主调函数；被调用的函数简称被调函数。

C++程序就是由许多函数组成的。一个 C++程序虽然可能包含许多函数；但一个程序必须有且只有一个主函数，主函数名字固定为 main()且地位特殊。之所以说主函数地位特殊，是因为程序总是从主函数开始执行，然后主函数通过调用一系列其他函数来完成整个程序的任务。而程序中的其他函数是不能调用主函数的。只有操作系统可以调用主函数，程序启动时是操作系统调用程序中的主函数使程序开始执行。程序中这种调用和被调用的层次关系可以用图 4.1

表达。图中每一根线有双向箭头，分别代表调用函数调用谁和被调函数返回信息给谁。

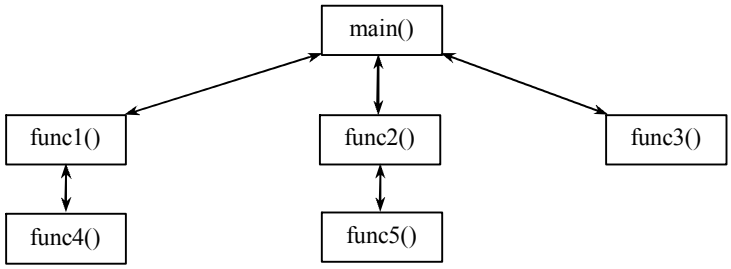

图 4.1　调用和被调用函数的层次关系

4.1　函数的定义与调用

4.1.1　函数的定义

函数定义的一般形式如下：

```
类型标识符 函数名([形式参数列表])
{ 声明语句
  执行语句
}
```

说明：

（1）函数的第一行（函数首部）最前面的“类型标识符”是一个数据类型关键字，用来定义该函数返回值的数据类型。“函数名”是程序员自己为函数取的名字，可以是任何合法的标识符，函数的取名一般要尽量反映该函数的功能。接下来一对花括号里面的内容是函数的函数体，主要包括两类语句：一是应先出现的声明语句，它们主要是声明本函数中要使用的变量以及将要调用的函数；二是语句，就是实现本函数功能的执行语句，这类语句是函数体的主要内容。

（2）在每个函数第一行的函数名后面括号中的内容称作函数的参数，是形式参数。如果在函数首部函数名后面的括号里面省略形式参数或者括号内填写 void 关键字，表示该函数是一个无参数函数。如果在函数首部函数名后面的括号里面填写了形式参数，则它是有参数函数。有参函数通过参数列表列出每个参数的数据类型、参数变量名。若参数不止一个，则每个参数之间用逗号分隔。

下面举一个简单的函数调用例子。

【例 4.1】主函数调用 comp(n)函数计算前 n 个自然数之和，又调用 print_word 函数打印一行字符。

程序代码如下：

```
//*****ex4_1.cpp*****
#include <iostream>
using namespace std;
int comp(int n)                    //定义 comp 函数
{     int k,s;
```

```
        s=0;
        for (k=1;k<=n;k++)     s=s+k;
        return s;                                    //返回计算结果
    }
    void print_word(void)                  //定义 print_word 函数
    { cout<<"This is an example for C++ function!"<<endl; }
    void main(void)
    {     int n;
        cin>>n;
        if (n>1)
            cout<<"The sum is:   "<< comp(n)<<endl;      //调用 comp 函数
        print_word();                       //调用 print_word 函数
    }
```

程序的运行结果如下：

```
10
The sum is:  55
This is an example for C++ function!
```

说明：

（1）例 4.1 中包含 3 个函数：main 函数、comp 函数和 print_word 函数。其中 comp 函数和 print_word 函数是用户自定义的函数，comp 函数的功能是求 1～n 的和；print_word 函数的功能是输出一行文字信息。

（2）形式参数为 void 关键字的表示该函数没有函数参数，因此在调用该函数时也不能给它参数；而 comp 函数中的内容(int n)表示该函数有一个整数类型的参数，因此在调用此函数时应给它一个整数参数。

（3）在每个函数第一行的函数名前面出现的关键字一定是一个数据类型关键字，用于指定函数值类型。其中，该关键字若为 void，则表示该函数不返回函数值；若为其他关键字，则返回相应类型的函数值。例如 comp 函数前面出现关键字 int，表示它是返回整数值的函数。在 C 语言中该关键字也可以省略，这时默认函数的返回值为整型；但 C++中不能省略该关键字，这样做更加严格和安全。

（4）函数调用语句中函数名后面括号里的内容是调用者提供的信息，叫做实际参数，如主函数中 comp(n)语句里面的 n 就是实际参数。这里的实际参数应该是已知的常数，或是有具体值的变量名，或是有具体值的表达式。

（5）对于有函数值的函数，在它的函数内容中必须有一个 return 语句，利用 return 语句将后面的值作为函数值返回给调用者。

（6）程序是首先执行主函数，在主函数中遇到调用函数语句时流程转到被调函数内执行，当被调函数内的语句执行完毕后，程序返回到主调函数调用语句的下一句继续执行。

C++程序就是由一系列函数组成的。函数可以分为下列两类：

（1）系统函数，即库函数。这是由 C++编译系统提供的，用户不必自己编写，可以在需要的地方直接调用它们。值得说明的是，不同的 C++编译系统提供的库函数的数量和功能不同，当然，有一些基本的函数是共同的。

如果用户在程序中需要使用某些库函数，那么，必须在这个程序的开头部分用 include 语

句把与这些库函数有关的头文件包含到用户程序中。而调用这些库函数的方法与调用自己编写的函数是类似的。因此，事先只要求用户了解欲使用的库函数名称、功能、所需参数个数、参数类型、函数值类型；至于库函数里面的内容是如何写的不需要用户关心。

（2）用户自定义函数。这就是用户根据专门需要而自己编写的函数，自己编写函数叫做自定义函数，所谓“定义”，作为动词的意思在这里就等同于“编写”。

之所以 C++给用户提供自定义函数的机会，是因为库函数不可能将解决现实世界任何问题的方法都包罗进去。

4.1.2 函数的声明

声明的作用实际上是提前通知编译系统有什么函数将被调用。在一个函数中调用另一个函数之前，一般先要对被调函数提出声明。只有在下列特殊情况下可以不需要声明：被调函数是自定义函数，而且该自定义函数出现在同一文件的主调函数之前。因为被调函数出现在前，后面的主调函数已经知道它的存在，所以不要声明。对于被调函数出现在主调语句之后的情况，还是需要提前声明。

（1）对库函数的声明。其实，对库函数的声明语句已经写在有关包含文件中了，因此只要在程序文件头用 include 语句将这些包含文件包含到本程序中，就完成了对库函数的声明。

（2）对自定义函数的声明。必须在调用某自定义函数的语句之前写上如下声明语句：

函数类型关键字 函数名([参数 1 类型,参数 1 名称][,参数 2 类型,参数 2 名称][,……]);

也可以在声明语句中略去参数的名称，或写一个任意名称，这叫做函数原型（function prototype），即声明函数原型。函数原型有下列两种表示形式：

函数类型关键字 函数名([参数 1 类型][,参数 2 类型][……]);

函数类型关键字 函数名([参数 1 类型,标识符 1][,参数 2 类型,标识符 2][……]);

C++中的函数原型说明了函数的类型、函数名、函数各形式参数类型。使用函数原型是 C 和 C++的一个重要特点。

下面的例子在主函数中要调用 3 个自定义函数 add、sub 和 mult，它们都出现在主函数之后，因此在调用之前都要提出声明；声明语句分别使用了上述三种格式。

【例 4.2】 声明被调函数的示例。

程序代码如下：

```
//*****ex4_2.cpp*****
#include <iostream>
using namespace std;
float add(float x, float y);            //声明 add 函数
float sub(float, float);                //声明 sub 函数
double mult(float p, float q);          //声明 mult 函数
void main()
{    float a,b;
     cout <<"Input a, b: ";
     cin >>a>>b;
     cout << add(a, b)<<", "<< sub(a, b)<<", "<< mult(a, b)<<endl;
}
float add(float x, float y)             //定义 add 函数
```

```
{     float z;
      z=x+y;
      return (z);
}
float sub(float x, float y)          //定义 sub 函数
{     float z;
      z=x-y;
      return (z);
}
double mult(float x, float y)        //定义 mult 函数
{     double z;
      z=x*y;
      return (z);
}
```

程序的运行结果如下：

```
Input a, b: 20 10
30, 10, 200
```

说明：

（1）函数的定义和函数的声明是两回事，不要混淆。定义函数是建立一个函数，实现函数的功能，包括指定函数名、函数类型、形式参数类型及其名字、函数体等，它是一个完整的、独立的函数单位。而声明的作用是把函数的名字、函数的类型，以及形式参数的个数、类型和顺序（注意，不包括函数体）通知编译系统，以便在对包含函数调用的语句进行编译时，根据声明内容对函数进行对照检查（例如调用语句使用的函数名是否正确，提供的参数与形式参数的类型和个数是否一致）。相似点就是函数的声明与函数定义中的函数首部基本上是相同的。因此，可以简单地照写已定义函数的首行，再加上一个分号，就成为对函数的声明语句；注意函数定义的第 1 行末尾是没有分号的，因此它不是一个独立语句。

（2）之所以函数原型中可以省略形式参数的名称，是因为形式参数的名称是无关紧要的，且在调用前形参并不存在。重要的是形式参数的类型、个数和顺序。即使在函数原型声明中也可以写上形式参数名，参数名称也是可以随便写的。因为编译系统不检查形式参数名。

（3）函数声明语句的位置。函数声明语句可以放在主调函数中，也可放在函数外面，只要出现在调用语句之前即声明有效。如果函数声明语句放在函数外面，且处在所有函数定义之前，则各主调函数中就可以都不必对所调用的函数再做声明。例如，例 4.2 的三个函数声明都在任何函数定义之前，所以任何函数要做主调函数来调用声明的函数，都不要再声明了。如果一个函数被多个函数调用，用这种方式声明比较好，它可以避免在每个主调函数中重复声明。

4.1.3 函数的返回值

有了函数的定义、声明之后，函数就可以被调用了。但调用一个函数之前应该对该函数的返回值与函数类型有所了解。函数的返回值就是该函数的函数值。

关于函数的返回值需要注意以下几个要点：

（1）关于 return 语句。函数的返回值是通过函数定义中的 return 语句返回到主调函数的。如果一个函数需要有返回值，那么该函数的定义中必须有一个 return 语句；如果不需要函数返

回值，则函数的定义中可以不要 return 语句。

return 语句的写法：

return (表达式);

return 语句的作用不仅可以将函数的值返回到主调函数，同时也是结束被调函数运行的语句。一个函数中可以有一个以上的 return 语句，但执行到哪个 return 语句，哪个语句起作用。

return 语句后面的括号可要可不要，如“return z;”与“return (z);”两种写法的语句是等价的。return 语句后面的表达式表示返回值，如“return (x+y);”。

（2）函数值的类型。既然函数值也是一个值，就必然属于某种数据类型，一个有函数值的函数，它返回的函数值的类型是确定的，这个类型唯一由函数首部的数据类型关键字所决定。人们也把函数首部该关键字的类型称作是函数的类型。因此，只要看函数定义的首部，就能知道该函数的返回值类型。

（3）如果 return 语句后面的表达式值类型与函数的类型不一致，则返回的值类型以函数的类型为准，即函数类型决定返回值类型。对数值型数据，可以自动进行类型转换。

这种转换有时可能会使得函数返回值与 return 语句后面的值之间出现一些误差，即，如果函数的类型精确度比 return 语句后面表达式值的精确度低（比如函数是 int 型，return 语句后面的表达式值是 float 型），则返回的值与 return 语句后面的表达式值就会存在差异。

4.1.4 函数的调用

函数调用的一般格式为：

函数名([实际参数列表])

参数列表中的一对方括号表示该项内容可有可无。如果调用的是无参函数，则没有“实际参数列表”，但括号不能省略，例如例 4.1 中 print_word()函数的调用。如果调用的函数是有参数的，则调用必须提供实际参数，如果有多个参数，则提供的多个实际参数之间用逗号分隔。注意，在提供的实际参数列表中，只要列出每个实际参数即可（可以是常量、有值的变量名或表达式），而不需要加上数据类型关键字。这一点是函数调用格式与函数定义格式在参数写法上的区别。实际参数必须与函数定义首部中描述的形式参数在个数、类型、顺序各方面严格一致。实际参数将与形式参数按顺序对应，一对一地传递数据。但有一点值得说明，如果实际参数表包括多个参数，且参数中出现表达式，而表达式的值又与参数求值顺序有关，这时传到函数形式参数的值就与实际参数求值顺序有关了。例如，若实际参数变量 n 的值为 3，有以下函数调用：

```
fun(n,++n);
```

如果实际参数的求值顺序是自左至右，则函数调用相当于 fun(3,4)，传给形式参数的两个数据依次为 3、4；如果实际参数的求值顺序是自右至左，则函数调用相当于 fun(4,4)，这时传给形式参数的两个数据依次为 4、4。参数求值的顺序是与编译系统有关的，许多 C++系统（如 Visual C++）是按自右至左的顺序求值的。

函数调用能够以下列三种形式出现：

（1）函数语句。把函数调用单独作为一个语句，并不要求函数带回一个值或者不需要函数的值，只是要求把函数执行一遍以完成函数中的操作。调用形式为：

函数名([实际参数列表]);

最后的分号不可少。

（2）函数表达式。函数调用出现在一个表达式中，将函数值作为一个参与表达式运算的数据，这时要求函数带回一个确定的值以参加表达式的运算。如下面的函数调用 max(a,b)出现在一个乘法表达式中，该赋值语句右边是一个函数表达式：

```
c=3*max(a,b);          //赋值语句右边是典型的函数表达式
```

（3）函数参数。函数调用得到的值作为另一次函数调用的实际参数。如：

```
m=max(a,max(b,c));   // max(b,c)是典型的函数参数
```

这里，max(b,c)是函数调用，其函数值作为另一次函数调用（外层 max 函数）的一个实参出现，外层函数的另一实参是 a。如果 max(x,y)的功能是求 x、y 中的大数，那么上面的示例语句 m 得到的就是 a、b、c 三数中的最大数。由此可见，对有函数参数情况，C++是先进行参数中的函数调用，再进行外层函数调用。

【例 4.3】 测试编译程序的参数计算顺序。

程序代码如下：

```
//*****ex4_3.cpp*****
#include <iostream>
using namespace std;
void fun(int x, int y)
{     cout<<"x="<<x<<"\ty="<<y<<endl;     }
void main()
{
      int a=10, b=20;
      cout<<"a="<<a<<"\tb="<<b<<endl;
      cout<<"fun(a, ++a): ";
      fun(a, ++a);
      cout<<"fun(b, b++): ";
      fun(b, b++);
}
```

程序的运行结果如下：

```
a=10    b=20
fun(a, ++a): x=11       y=11
fun(b, b++): x=20       y=20
```

第 2 行的输出结果表明：Visual C++计算参数是从右至左，所以形参 x 和 y 的值都是 11。前缀运算的结果影响了左边参数的值。而第 3 行的输出表明：后缀运算的结果并不影响左边参数的值。

4.2 函数的参数传递

大多数函数是带参数的，只有透彻理解有关概念和参数的传递机制，才能灵活运用函数。

形参：在定义函数时，函数名后面括号中的变量名称为形式参数（formal parameter），简称形参。形参是无内存单元（因而不存在）的任何合法标识符。

实参：在调用一个函数时，调用语句的函数名后面括号中的参数称为实际参数（actual parameter），简称实参。实参是实际存在（因而有特定值）的常量、变量或表达式。

有关形参和实参的说明：

（1）在定义函数时指定的形参，在未被调用时它们并不占内存中的存储单元，实际上也就是不存在的，因此称它们是形式参数或虚拟参数，表示它们并不是实际存在的数据。只有在函数调用发生时，被调函数中的形参才被分配内存单元，以便接收从实参传来的数据。在调用结束后（即函数值返回后），形参所占用的内存单元也被释放（即形参又不存在了）。

（2）实参可以是常量、变量或表达式，但作为实参表达式中的变量必须有确定值，以便整个实参表达式有确定的值。

（3）在定义函数时，必须在函数首部指定形参的类型，至于形参使用何名字可随意。

（4）实参与形参的类型应相同或赋值兼容。两种参数类型完全一致无疑是完全合法、正确的。如果实参为整型而形参为实型，或者相反，则按不同类型数值的赋值规则进行转换，原则上不出现语法错误，但结果可能带来某些非期望的误差。例如实参 a 的值是 3.5，而被调函数的形参 x 为整型，则调用该函数时会将 3.5 转化为整数 3，然后送到形参 x，故 x 得到的是 3，由 3 计算出的函数值与人们期望由 3.5 计算出的函数值一般是有差别的。但字符型与整型可以互相通用。

4.2.1 参数的值传递

函数的参数传递最常见的方式是值传递方式。实际上，值传递参数的实现是系统将实参拷贝一个副本给形参，拷贝后两者就断开关系。在被调函数中，形参可以被改变，但这只影响副本中的形参值，而不影响调用函数的实参值。所以这类函数有对原始数据保护的作用。换一句话说，这种参数传递机制是单向影响，即只能由实参将值传给形参（实参影响形参）；而形参在函数中的值如果发生变化，不会反过来影响对应的实参。

【例 4.4】 参数值传递的示例。

程序代码如下：

```
//*****ex4_4.cpp*****
#include <iostream>
using namespace std;
int max(int x, int y)              //定义有参函数 max 求两数最大值，x 和 y 是形参
{     float m;
      cout<<"The initial x,y: "<<x<<","<<y<<endl;
      m=x>y?x:y;
      x=2*x; y=y+1;
      cout<<"The new x,y: "<<x<<","<<y<<endl;
      return (m);
}
void main()
{     float a,b,c;
      cout<<"Input two integer numbers: ";
      cin >> a >> b;
      c=max(a,b);              //调用函数 max，a 和 b 是实参，函数值赋给变量 c
      cout << "The maxinum is: "<<c<<endl;
      cout << "After calling function,a,b: "<<a<<","<<b<<endl;
}
```

程序的运行结果如下：

```
Input two integer numbers: 17 18
The initial x,y: 17,18
The new x,y: 34,19
The maxinum is: 18
After calling function,a,b: 17,18
```

例 4.4 的运行结果表明，虽然被调函数在执行过程中形参变量 x 和 y 的值发生了变化，但不能反过来影响其原来对应的实参 a 和 b，即 a 和 b 的值还是原来的值。

4.2.2 参数的地址传递

除了上一小节介绍的值传递参数方式外，函数调用还有一种特殊的值传递形式，即传递的值不是一般的数值，而是一些内存单元地址编号（即地址），这时，一般称之为参数的地址传递。在这种参数传递形式中，无论在函数的定义中出现的形参还是在调用语句中出现的实参，都是代表一些内存单元地址，而不是一般的数值。

C++中的参数地址传递情况一般有如下几种：实参可以是一个有确定值的普通变量的地址，或者是一个已经初始化的指针变量（后面章会学到，指针变量的值是代表一个内存单元的地址），或者是一个初始化的数组名（后面章会学到，数组名的值也是代表一个内存单元地址），或者是一个具体的函数名（后面章会学到，函数名的值也是代表一个内存单元地址编号，即函数的入口地址）。而形参可以是一个任意普通变量的地址，或是一个任意指针变量，或是一个任意的数组名，或是一个指向函数的指针变量（对应于实参是具体函数名）。

实际上，这种参数传递机制就是在函数调用时把一个内存单元地址传递给形参，使形参也具有实参的内存单元地址（即两者对应同一个内存单元），称作形参和实参地址结合，两者合二为一。这样一来，任何时候形参的值等于实参的值；而实参的值也等于形参的值。因此，形参在函数中发生变化后，也会引起实参跟着变化（因为它们是捆绑在一起的，一体化的）。这就意味着按地址传递的方式，在调用刚开始时实参的值影响了形参；而在被调函数执行过程中形参值若发生了变化，它也会影响实参的值变化。即机制是双向影响，这与值传递方式的单向影响机制形成对比。

【例 4.5】 地址传递的示例。

程序代码如下：

```
//*****ex4_5.cpp*****
#include <iostream>
using namespace std;
void fun(int *p);
void main()
{
    int a=1;
    cout<<"a 的初值: "<<a<<endl;
    fun(&a);          //用变量 a 的地址作为实参
    cout<<"调用函数 fun 后，a 的值: "<<a<<endl;
}
void fun(int *p)      //p 是一个指针变量，可以指向整型类型变量
{                     //通过实参与形参的地址结合，让 p 指向变量 a
```

```
    *p=5;              //将 p 指向的变量的内容设置为 5
}
```

程序的运行结果如下：

```
a的初值: 1
调用函数fun后，a的值: 5
```

在例 4.5 中，因为采用地址传递参数的办法，形参值的改变影响了实参变量 a 的值。请读者注意，在函数调用处实参的写法不再是基本类型的变量，而是一个地址值。在函数定义处的形参也不再是基本类型的变量，而是一个指针变量。

还要提示的是：地址传递的方式不仅可以让形参影响实参，还可以让函数的返回值有多个。有关用地址、指针作为函数参数的例题，将会在后面“数组与指针”一章中遇到。

4.3 函数的嵌套调用与递归调用

虽然 C++函数不能嵌套定义，但 C++函数是可以嵌套调用的，即允许在一个函数的函数体中调用另一个函数。甚至可以递归调用，即允许在一个函数的函数体中直接或间接调用函数本身，这就是函数的递归调用。

4.3.1 函数的嵌套调用

下面的 main 函数调用了 f1 函数，而 f1 函数在执行中又调用 f2 函数，代码具有如下形式：

```
void main()
{…
  f1();              //主函数中调用 f1 函数
}
  int f1()           //定义 f1 函数
  {…
    f2();            // f1 中调用 f2 函数
  }
    int f2()         //定义 f2 函数
  {…
  }
```

这就是嵌套调用，上述嵌套调用可用图 4.2 示意。

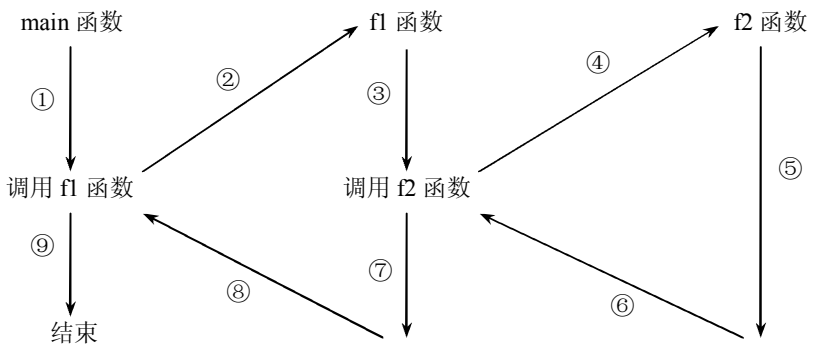

图 4.2　函数的嵌套调用

图 4.2 表示的示例是两层嵌套（算上 main 函数共 3 层函数），图中每一根带箭头的线代表一类操作步骤，共总结出 3 类操作。其中竖直向下的箭头线表示执行同一函数内的语句，从左下往右上的箭头线表示函数调用，从右下往左上的箭头线表示函数返回。每根线旁边标注的数字代表执行的顺序，即：第 1 步首先从主函数的先前代码开始执行，第 2 步执行主函数调用 f1 函数的操作，……，最后第 9 步执行主函数的末尾代码，直到整个程序运行结束。发现嵌套调用有这样一个规律：最先执行的主函数最后结束，而最后被调用的 f2 函数却最先结束。这在计算机学科中称作“后进先出”（称栈或堆栈）规则。

【例 4.6】 编程求组合 $C_m^n = \frac{m!}{n!(m-n)!}$，其中求组合的功能要求用函数完成。

分析：根据组合的计算公式，知组合函数有两个形参：m 和 n，可以用自定义函数 comb(int n,int m)表示求组合。而在 comb 函数中需要 3 次计算阶乘，如果定义函数 fac(k)求 k 的阶乘，然后在 comb 函数中调用 fac 函数，可以使程序代码简单，只要在 comb 函数中写一个语句“c=fac(m)/(fac(n)*fac(m-n));”即可求出组合值。

程序代码如下：

```
//*****ex4_6.cpp*****
#include <iostream>
using namespace std;
long fac(int k)                                    //定义求阶乘的函数
{    long f=1; int i;
     for(i=1;i<=k;i++) f = f*i;
     return f;
}
long comb(int n, int m)                            //定义组合函数
{
     return fac(m)/(fac(n)*fac(m-n));              //嵌套调用阶乘函数
}
void main()
{    int n,m; long c;
     cout<<"Input two integer numbers: m,n: ";
     cin>>m>>n;
     c=comb(n,m);                                  //调用组合函数 comb
     cout<<"c="<<c<<endl;
}
```

程序的运行结果如下：

```
Input two integer numbers: m,n: 10 6
c=210
```

主函数调用 comb 函数；comb 函数在执行过程中又调用了 fac 函数。fac 函数的调用被嵌套在 comb 函数的调用中。

4.3.2 函数的递归调用

在调用一个函数的过程中（函数的函数体中）又出现直接或间接地调用该函数本身，这种调用现象称为函数的递归（recursive）调用。

直接递归调用的代码形式如下：

```
int fun1()                  //fun1 函数的定义
{
    …                       //函数其他部分
    z=fun1();               //直接调用自身
    …                       //函数其他部分
}
```

以上是在 fun1 函数中直接调用了 fun1 函数，直接递归调用过程如图 4.3 所示。

间接递归调用可以表现为如下形式：

```
int fun2()                  //fun2 函数的定义
{
    …                       //fun2 函数的其他部分
    x=fun3();               //调用 fun3 函数
    …                       //fun2 函数的其他部分
}
int fun3()                  //fun3 函数的定义
{
    …                       //fun3 函数的其他部分
    y=fun2();               //调用 fun2 函数
    …                       //fun3 函数的其他部分
}
```

fun2 函数中调用了 fun3 函数，而 fun3 函数中又调用了 fun2 函数，相当于 fun2 函数间接地调用了 fun2 函数。这种调用称为间接递归调用，调用过程如图 4.4 所示。

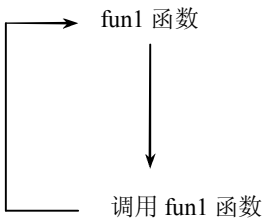

图 4.3　函数的直接递归调用

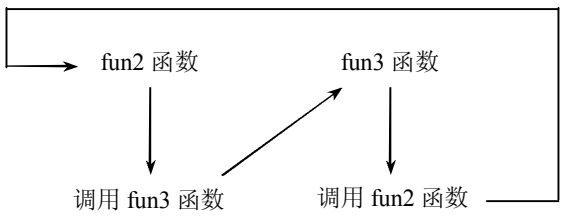

图 4.4　函数的间接递归调用

可以看到，递归调用是一种特殊的嵌套调用。按上面所述（参看图 4.3 和图 4.4）会发现这两种递归调用似乎都是无限地调用自身（形成无限循环）。显然，这样的程序将出现类似于“死循环”的问题。然而，实际上应当只能允许出现有限次的递归调用，当达到某种条件时应该使递归调用终止。通常这种条件可以用 if 语句来控制，所以使用递归时特别要注意确定终止递归的条件。

递归在解决某些问题时是一个十分有用的方法。其一，因为有的问题它本身就是递归定义的；其二，因为它可以使某些看起来不易解决的问题变得容易描述和容易解决，使一个蕴含递归关系且结构复杂的程序变得简洁精炼，程序可读性强。

【例 4.7】 用递归计算 n!。

分析：n! 本身就是以递归的形式定义的：

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

求 n!，应先求(n-1)!；而求(n-1)!，又需要先求(n-2)!；而求(n-2)!，又可以变成求(n-3)!；……；如此继续，直到最后变成求 1!的问题。而根据公式有 1！=1（这就是本问题的递归终止条件）。由终止条件得到 1!的结果后，再反过来依次求出 2!，3!，……，直到最后求出 n!。

设求 n!的函数为 fac(n)，函数体内求 n!，只要 n>1，可用 n*fac(n-1)表示，即 fac(n)函数体内将递归调用 fac()自身；但一旦参数 n 为 1，则终止调用函数自身并给出函数值 1。

程序代码如下：

```
//*****ex4_7.cpp*****
#include <iostream>
using namespace std;
long fac(int n)
{    long f;
     if (n==1)
          f=1;
     else
          f=n*fac(n-1);          //递归调用，求(n-1)!
     return f;
}
void main()
{    long y; int n;
     cout<<"Input a integer n: ";
     cin >> n;
     y=fac(n);                   //调用 fac(n)求 n!
     cout<<"n="<<n<<", "<<"y="<<y<<endl;
}
```

程序的运行结果如下：

```
Input a integer n: 10
n=10, y=3628800
```

从求 n!的递归程序中可以看出，递归定义有两个要素：

（1）递归终止条件。也就是所描述问题的最简单情况，它本身不再使用递归的定义，即程序必须终止。如例 4.7，当 n=1 时，fac(n)=1，不再使用 f(n-1)来定义。

（2）递归定义使问题向终止条件转化的规则。递归定义必须能使问题越来越简单，即参数越来越接近终止条件的参数；达到终止条件参数时函数有确定的值。如例 4.7，fac(n)由 fac(n-1)定义，越来越靠近 fac(1)，即参数越来越接近终止条件参数 1；达到终止条件参数时函数有确定的值，即 fac(1)=1。

采用递归调用的程序结构清楚；但是采用递归调用的程序效率往往很低，比较费时且耗用较多内存空间。因为在递归调用的过程当中，系统需要为每一层的返回点、局部量等开辟栈来存储。递归次数过多容易造成栈溢出等。

4.4 内联函数

内联函数也称内置函数、内嵌函数。引入内联函数的目的是为了提高程序中函数调用的效率。

4.4.1 内联函数的引入

先看下面的例子。

【例 4.8】 下面程序中的 is_number 函数反复被调用，用于判断用户从键盘输入的系列字符是数字字符还是其他字符。

程序代码如下：

```
//*****ex4_8.cpp*****
#include <iostream>
using namespace std;
int is_number(char);            //函数声明
void main()
{    char c;
     while ((c=cin.get())!='\n')
     {
          if (is_number(c))        //调用一个小函数
               cout<<"you enter a digit \n";
          else
               cout<<"you enter a non-digit \n";
     }
}
int is_number(char ch)              //函数定义
{ return (ch>='0'&& ch<='9')?1:0; }
```

程序中不断到设备中读取数据，频繁调用 is_number 函数。为了避免频繁调用函数，提高执行效率，可以将例 4.8 的程序改为：

```
//*****ex4_8_2.cpp*****
#include <iostream>
using namespace std;
void main()
{    char c;
     while ((c=cin.get())!='\n')
     {
          if ((c>='0'&& c<='9')?1:0)   //修改处：直接计算表达式
               cout<<"you enter a digit \n";
          else
               cout<<"you enter a non-digit \n";
     }
}
```

修改后的程序在 if 语句中用表达式替换了函数调用。在程序运行上，提高了一些执行效率，因为免去了大量的函数调用开销。

但是，由于 is_number 函数比相应的表达式可读性好，所以修改后的代码可读性降低，尤其若程序中多处出现 is_number 的替换时，会大大降低可读性。

我们希望既要用函数调用来体现其结构化和可读性，又要使效率尽可能的高。为了尽量做到两全其美，C++中引入内联函数这个方法。

4.4.2 内联函数的定义与调用

定义内联函数的方法很简单，只要在函数定义的首部加上关键字 inline 就可以了，函数定义的具体格式如下：

```
inline 函数名(形参列表)
{
  …
}
```

内联函数的声明也是用 inline，具体声明格式如下：

```
inline 函数名(形参类型表);
```

编译器看到 inline 后，为该函数创建一段代码，以便在后面每次碰到该函数的调用都用相应的一段代码来替换。其实，内联函数只要在开头一次性声明为 inline 即可，而后面的函数定义仍可写成一般函数定义的形式，编译器也会将函数视为内联函数。如对于例 4.8 的问题，下面使用内联函数来解决，代码可以写成下列形式：

```
//*****ex4_8_3.cpp*****
#include <iostream>
using namespace std;
inline int is_number(char);          //inline 函数声明
void main()
{    char c;
     while ((c=cin.get())!='\n')
     {
          if (is_number(c))          //调用一个小函数
               cout<<"you enter a digit \n";
          else
               cout<<"you enter a non-digit \n";
     }
}
int is_number(char ch)               //此处无 inline，却被本程序视为 inline
{ return (ch>='0'&& ch<='9')?1:0; }
```

关于内联函数的说明：

（1）内联函数与一般函数的区别在于函数调用的处理。一般函数进行调用时，要将程序执行到被调用函数中，然后返回到主调函数中；而内联函数在调用时，是将调用部分用内联函数体来替换。

（2）内联函数必须先声明后调用。因为程序编译时要对内联函数替换，所以在内联函数调用之前必须声明是内联的（inline），否则将会像一般函数那样产生调用而不是进行替换操作。下面内联函数的声明就是达不到目的的。

```
#include <iostream>
using namespace std;
int is_number(char ch);              //此处没有声明 is_number 是内联函数
void main()
{    char c;
     while ((c=cin.get())!='\n')
```

```
    {    if (is_number(c))              //将按普通函数调用 is_number
         cout<<"you enter a digit \n";
         else
         cout<<"you enter a non-digit \n";
    }
}
inline int is_number(char ch)           //此处被本程序视为普通函数
{ return (ch>='0'&& ch<='9')?1:0; }
```

（3）在内联函数中，不能含有复杂的结构控制语句，如 switch、for 和 while。如果内联函数有这些语句，则编译器将该函数视同一般函数那样产生函数调用。

（4）递归函数不能用作内联函数。

（5）以后讲到的类中，所有定义在说明内部的函数都是内联函数。

4.5 变量和函数的属性

每一个变量除了有数据类型属性外，还有空间、时间两大属性。

所谓变量的空间属性是指变量都有其有效的作用范围，这个范围叫做变量的作用域。只有在变量的作用域内程序才能访问到该变量；而在它的作用域外是不能访问这些变量的。

所谓时间属性是指变量都有其存储的时间段，这个时间段叫做变量的生存期。只有在变量的生存期间才能访问到该变量；而在它的生存期之外的时间段是不能访问这些变量的（因为此时该变量已经不被存储了）。关于变量的作用域和生存期请参阅表 2.2。

每一个函数除了有数据类型属性外，也有空间属性，即函数可以被使用的范围。

4.5.1 变量的作用域

首先将变量按作用域大小分类，可以分为局部变量和全局变量。

C++中并不是所有的变量对任何函数都是可知的。一些变量在整个程序或文件中都是可见的，这些变量作用域大，被称为全局变量。一些变量只能在一个函数或块中可见，这些变量作用域小，被称为局部变量。

变量作用域的大小和它们的存储区域有关。全局变量存储在全局数据区（也称为静态存储区），它在程序运行的时候被分配存储空间，当程序结束时释放存储空间。局部变量存储在堆栈数据区，当程序执行到该变量声明的函数（或程序块）时才开辟存储空间，当函数（或程序块）执行完毕时释放存储空间。而决定变量作用域大小的根源是它的定义位置。

1．局部变量

局部变量是指定义在函数或程序块内的变量，它们的作用域分别在所定义的函数体或程序块内。例如，前面所有函数中定义的形参和函数内定义的变量都是局部变量。

下面举例说明局部变量的实例：

```
void main()
{int x;                  //x 为定义在 main 函数中的局部变量，其作用域为 main 函数内
    …
    {   int y;          //y 为定义在块内的局部变量，其作用域为块内
        …
```

```
        }
        …
}
void fun()
{       char ch;            //ch 为定义在 fun 函数的局部变量，其作用域为 fun 函数内
        …
}
```

由于局部变量只在其定义的函数（或程序块）内有效，所以不同函数内命名相同的变量是可以的，它们不会相互干扰。这个性质为多函数的程序设计提供了方便，使得项目管理者只需要为程序员指定编写函数的参数和功能；而程序员则不必担心自己编写的函数中是否有变量与其他程序员编写函数中的变量同名，只需专心致力于编写函数的功能就可以了。

【例 4.9】 局部变量的使用。

程序代码如下：

```
//*****ex4_9.cpp*****
#include <iostream>
using namespace std;
double fun1(double a, double b)          //fun1 函数中有 3 个局部变量 a、b、c
{       double c;
        c=a+b;
        return c;
}
double fun2(double a, double b)          //fun2 函数中有 3 个局部变量 a、b、c
{       double c;
        c=a*b;
        return c;
}
void main()       //main 函数中有 3 个局部变量，也分别取名 a、b、c
{       double a,b,c;
        cout<<"Input two numbers: ";
        cin>>a>>b;
        c=fun1(a,b);
        cout<<"a+b="<<c<<endl;
        c=fun2(a,b);
        cout<<"a*b="<<c<<endl;
}
```

程序的运行结果如下：

```
Input two numbers: 5 15
a+b=20
a*b=75
```

可见，每个函数中都定义了相同名称的局部变量，比如有 3 个 a；但这个范围中定义的 a 与另一个范围中定义的 a 并不是同一个对象，即“此 a 非彼 a”。

2．全局变量

全局变量是定义在函数以外的变量（一般的全局变量也称外部变量，而一类特殊的全局变量则叫静态全局变量），它们原则上对整个文件的函数都是可见的，甚至对本程序的其他文件的函数也可见；但通常条件下的全局变量作用域是从定义变量的位置到该文件的结束。

【例 4.10】 全局变量的使用。

程序代码如下：

```
//*****ex4_10.cpp*****
#include <iostream>
using namespace std;
int a;                          //a 的作用域为整个文件
void fun1();                    //声明 fun1 函数
void main()
{    cout<<a<<endl;             //main 函数中使用了全局变量 a
     fun1();                    //调用 fun1 函数
     cout<<a<<endl;             //main 函数中再次使用全局变量 a
}
void fun1()
{ a=5; }                        //fun1 函数中使用全局变量 a
```

程序的运行结果如下：

```
0
5
```

说明：

（1）在例 4.10 中，主函数中第一次输出变量 a 时，a 还没有赋值，但是执行结果显示 0。这是因为当程序执行时，全局变量开辟存储空间的同时被系统初始化为 0（定义全局变量时，专门初始化除外）。调用 fun1 函数后，a 在 fun1 函数中被赋值为 5，所以主函数中第二次输出变量 a 时结果为 5。

（2）全局变量可以定义在任何位置，但其作用域是从定义的位置开始到文件的末尾。一般来说，全局变量定义在所有函数上面，这样所有的函数就可以使用该全局变量了。而定义在文件中间的全局变量就只能被其下面的函数所使用，全局变量定义之前的所有函数不会知道该变量。例如：

```
#include <iostream>
using namespace std;
void fun1();
void main()
{    x=4;     //不能使用全局变量 x，尚未看见 x 的定义，编译时认为 x 没有定义
     cout<<x<<endl;             //所以本程序不能编译执行
}
int x;                          //定义全局变量 x
void fun1()
{    cout<<x<<endl;             //可以使用全局变量 x
}
```

（3）全局变量为函数间数据的传递提供了通道。由于全局变量可以被其定义后的函数所使用，所以可以使用全局变量进行函数间数据的传递。而且这种传递数据的方法可以传递多个数据的值。下面就是利用全局变量传递数据的例子。

【例 4.11】 分别编写两个函数，求给定两个数的最大公约数和最小公倍数。其中，要求用全局变量存放最大公约数和最小公倍数，而不用函数值返回。

分析：由于求最小公倍数要依赖于最大公约数，所以应先求最大公约数。故应将求最大

公约数的函数写在前面，求最小公倍数的函数放在后面。

程序代码如下：

```
//*****ex4_11.cpp*****
#include <iostream>
using namespace std;
int greatest_common_divisor(int,int);          //声明求最大公约数的函数
int lease_common_multiple(int,int);            //声明求最小公倍数的函数
int max, min;          //全局变量分别存放最大公约数、最小公倍数
void main()
{    int a, b;
     cout<<"Input a b: ";
     cin>>a>>b;
     greatest_common_divisor(a,b);
     lease_common_multiple(a,b);
     cout<<"The greatest common divisor: "<<max<<endl;
     cout<<"The lease common multiple: "<<min<<endl;
}
int greatest_common_divisor(int x, int y)
{    int t, r;
     if (x<y)
     { t=x; x=y; y=t;}
     r=x%y;
     while(r!=0)
     {    x=y; y=r; r=x%y;    }
          max=y;
          return max;
     }
     int lease_common_multiple(int x,int y)
     {
       min=x*y/max;
       return min;
}
```

程序的运行结果如下：

```
Input a b: 48 18
The greatest common divisor: 6
The lease common multiple: 144
```

本例中利用全局变量 max 和 min 存储最大公约数和最小公倍数。最大公约数是在 greatest_common_divisor 函数中赋给 max 的；最小公倍数是在 lease_common_multiple 函数中赋给 min 的，而 max 和 min 又在 main 函数中使用。本程序的整个过程就是利用全局变量 max 和 min 由 greatest_common_divisor 函数和 lease_common_multiple 函数向 main 函数传递数据实现的。

（4）其他文件也可以使用文件外定义的全局变量，但要求该变量是外部变量类型的全局变量，而且要求在使用该变量的文件中要有对该变量的声明。外部变量跨文件使用的例子，请参考本章后面的例 4.14。

（5）全局变量降低了函数的通用性，建议不是必须不要使用全局变量。因为如果函数在执行的时候使用了全局变量，那么其他程序使用该函数时也必须将该全局变量一起移过去。另外，全局变量在程序执行的全部过程都占用存储空间，而不是需要时才开辟存储空间。

3．重名局部变量和全局变量作用域规则

在 C++中，虽然不允许相同作用域的变量同名，但允许不同作用域的变量同名。即允许在函数内或程序块内定义与全局变量同名的变量。

【例 4.12】函数内变量、程序块内变量与全局变量同名示例。

```
//*****ex4_12.cpp*****
#include <iostream>
using namespace std;
int a=10;                                          //全局变量 a
void fun1()
{    cout<<"全局变量 a="<<a<<endl; }               //输出 10
void main()
{    int a=1;                                      //函数内局部变量 a
     cout<<"函数内局部变量 a="<<a<<endl;           //输出 1
     {    int a=2;                                 //程序块内局部变量 a
          cout<<"程序块内局部变量 a="<<a<<endl;    //输出 2
     }
     cout<<"函数内局部变量 a="<<a<<endl;           //输出 1
     fun1();
}
```

程序的运行结果如下：

```
函数内局部变量a=1
程序块内局部变量a=2
函数内局部变量a=1
全局变量a=10
```

在例 4.12 中，三个同名变量 a 定义在三个不同的作用域，是三个不同的变量，对应存储单元不同。

前面介绍过全局变量的作用域是整个文件中其定义后的部分，局部变量的作用域是其定义所在的函数或程序块。那么在一个特定的位置引用到某个有重名的变量时，到底是使用哪个变量呢？这由如下重名变量作用域规则决定：在某个作用域范围内定义的变量在该范围的子范围内可以定义重名变量，这时原定义的变量在子范围内是不可见的，但是它还存在，只是在子范围内由于出现了重名的变量而被暂时隐藏起来，过了子范围后，它又是可见的。

在上述例 4.12 中，全局变量 a 在 main 函数内是不可见的，因为 main 函数中定义了以 a 命名的变量，因此第一个输出语句输出的也是函数局部变量 a 的值 1。但全局变量 a 可以在其他函数中可见。同时，在程序块中全局变量 a 和函数局部变量 a 都是不可见的，因为程序块中又定义了以 a 命名的变量；因此第二个输出语句输出的是程序块内局部变量 a 的值 2。但在 main 函数中程序块以外的函数中定义的变量 a 是可见的，因此第三个输出语句输出的是函数局部变量 a 的值 1。全局变量 a 在 fun1 函数中是可见的，因此第四个输出语句输出的是全局变量 a 的值 10。

4.5.2 变量的生存期

从变量的空间属性考虑变量有作用域的概念。那么，从变量的时间属性考虑变量还有存

储期（storage duration，也称生存期）的概念，存储期即变量值在内存中存在的时间。而变量的存储期是由变量的存储类别决定的，有两种存储类别：动态存储方式与静态存储方式。

1．短生存期变量——动态存储方式

所谓动态存储方式，是指在程序运行期间动态地分配存储空间给变量的方式。这类变量存储在动态存储空间（堆或栈），执行其所在函数或程序块时开辟存储空间，函数或程序块结束时释放存储空间，生存期为函数或程序块的运行期间，主要有：函数的形参和函数或程序块中定义的局部变量（未用 static 声明）。

使用动态存储方式的变量有两种：自动变量和寄存器类变量。

（1）自动变量：函数中的局部变量默认是自动变量，存储在动态数据存储区。自动变量可以用关键字 auto 作为存储类别的声明。自动变量的生存期为函数或程序块执行期间，作用域也是其所在函数或程序块。例如：

```
int fun()
{     auto int a;   }     //a 为自动类变量
```

实际编程过程中，关键字“auto”可以省略。例如上述自动变量也可声明为下面形式：

```
int fun()
{     int a; }
```

（2）寄存器变量：寄存器变量也是动态变量，可以用 register 作为存储类别的声明。寄存器变量存储在 CPU 的通用寄存器中，由于 CPU 读写寄存器中的数据比读写内存中的数据要快，因此可以提高程序运行效率。寄存器变量的生存期和作用域为其定义所在的函数或程序块。一般情况下，将局部最常用到的变量声明为寄存器变量，如循环变量。

下面 main 函数中的循环变量 i 就使用了寄存器变量：

```
#include <iostream>
using namespace std;
void main()
{     register int i;
      int sum=0;
      for(i=1;i<=100;i++)     sum+=i;
      cout<<"1+2+...+100="<<sum<<endl;
}
```

寄存器变量的使用应注意以下问题：

（1）寄存器变量不宜定义过多。计算机中寄存器数量是有限的，不能允许所有的变量都为寄存器变量。如果寄存器变量过多或通用寄存器被其他数据使用，那么系统将自动把寄存器变量转换成自动变量。

（2）寄存器变量的数据长度与通用寄存器的长度相当。一般是 char 型和 int 型变量。

2．长生存期变量——静态存储方式

所谓静态存储方式，是指在程序运行期间分配固定的存储空间给变量的方式。这类变量存储在全局数据区，当程序运行时开辟存储空间，程序结束时释放存储空间，生存期为程序运行期间。采用静态存储方式的变量有：全局变量（含外部变量、静态全局变量）和静态局部变量。

（1）外部变量：外部变量就是只用数据类型关键字而未用 static 关键字定义的全局变量。存储在全局数据区，生存期为程序执行期间。如果不对外部变量另加声明，则它的作用域是从定义点到所在文件的末尾。其实，外部变量不管在何处定义，可以通过用 extern 关键字加以声

明后将其作用域扩展到整个程序，声明语句格式为：

extern 外部变量名;

这种扩展作用域的声明语句有两种情况用到：①对文件内容后面位置所定义的外部变量x，将其作用域扩展到本文件前面的函数。这时只要在该文件的前面用 extern 对该变量 x 声明即可，这叫做提前引用声明。②对本程序的另一个源文件 B 中定义的外部变量 y，要想扩展到本文件 A 中使用，这时只要在本文件 A 的前面用 extern 对该变量 y 声明即可，不妨称此为跨文件引用声明。下面的例 4.13 和例 4.14 分别对这两种情况加以示例。

【例 4.13】对定义在同一文件中的外部变量，作提前引用声明以扩展其使用范围到文件前面。

程序代码如下：

```
//*****ex4_13.cpp*****
#include <iostream>
using namespace std;
extern x;                    //提前引用声明
void main()
{     x=4; cout<<x<<endl;   }
int x;                 //外部变量 x 的定义
```

在例 4.13 中，通过对外部变量 x 作提前引用声明，提前告诉编译系统，该变量是本文件后面定义的外部变量，或者是本程序另一个文件中定义的外部变量。避免在函数中引用时出现“变量 x 未定义”的错误。

【例 4.14】对定义在另一文件中的外部变量，作跨文件引用声明以扩展其作用域到本文件。

程序代码如下：

```
//*****ex4_14.cpp*****      （文件 ex4_14.cpp 的内容）
#include <iostream>
using namespace std;
extern w;                                  //跨文件引用声明
void main()
{     cout<<w<<endl;        }          //使用 ex4_14_2.cpp 文件中定义的变量 w
//*****ex4_14_2.cpp*****          （文件 ex4_14_2.cpp 的内容）
int w=10;                                  //外部变量 w 的定义
int fun(int x,int y)
{     return (x+y);       }
```

程序的运行结果如下：

```
10
```

例 4.14 中 ex4_14.cpp 文件使用了在 ex4_14_2.cpp 中定义的变量 w。

无论是提前引用声明还是跨文件引用声明，编译系统看到 extern 声明语句时，首先是在本文件的后面查找该变量是否是本文件的外部变量；本文件没有时才到别的文件中查找。

（2）静态变量：静态变量存储在全局数据区，使用 static 声明。静态变量有两种：静态局部变量和静态全局变量。

① 静态局部变量是在定义局部变量时开头再添加一个 static 关键字。静态局部变量的特点是：程序执行时，为其开辟存储空间直到程序结束，但只能被其定义所在的函数或程序块所使用。所以静态局部变量的生命周期为程序执行期间，作用域为其定义所在的函数或程序块内。如果没有定义静态局部变量的初始值，系统将自动初始化为 0。

【例 4.15】 静态局部变量的使用。

程序代码如下：

```
//*****ex4_15.cpp*****
#include <iostream>
using namespace std;
void fun();
void main()
{    int i;
     for(i=0;i<3;i++)   fun();
}
void fun()
{    int a=0;
     static int b=0;          //定义静态局部变量
     a=a+1; b=b+1;
     cout<<a<<","<<b<<endl;
}
```

程序的运行结果如下：

```
1,1
1,2
1,3
```

在例 4.15 中，main 函数调用 fun 函数三次，fun 函数中定义了自动局部变量 a 和静态局部变量 b；fun 函数每次被调用结束时输出 a 和 b 的值。静态局部变量在程序执行开始就被存储在全局数据区并初始化为 0，值得特别指出的是：静态局部变量只被初始化一次。以后每次调用 fun 函数时，都在相同的存储单元存取数据，前一次调用后拥有的值可以被保存，所以三次输出的 b 分别是 1、2 和 3。a 存储在动态数据区，每次使用时开辟存储空间，fun 函数结束时释放存储空间，值不能被保存，所以每次调用函数时重新初始化为 0，故每次输出的 a 值都是 1。

在程序调试阶段可以利用静态局部变量统计一个函数执行的次数。

② 静态全局变量是在定义全局变量时开头再添加一个 static 关键字。静态全局变量的特点是：程序执行时，为其在全局数据区开辟存储空间并初始化为 0，在存储期仍与外部变量一样一直延续到程序结束。但其只能被其定义所在的文件使用，而不能借助前面所讲的跨文件引用声明扩展到文件外。

【例 4.16】 静态全局变量的使用。

程序代码如下：

```
//*****ex4_16.cpp*****   （文件 ex4_16.cpp 的内容）
#include <iostream>
using namespace std;
extern u;                          //试图对 u 作跨文件引用声明，此时行不通
void fun();
void main()
{    fun();
     cout<<u<<endl;                //出现“变量 u 未定义”错误
}
//*****ex4_16_2.cpp*****   （文件 ex4_16_2.cpp 的内容）
#include <iostream>
```

```
using namespace std;
static int u=10;                    //定义静态全局变量
//本例不能运行成功！去掉 static 后，程序可正常运行！
void fun()
{     cout<<"本例不能运行成功！去掉 static 后，程序可正常运行！"<<endl;     }
```

在例 4.16 中，文件 ex4_16.cpp 中定义的全局变量 u 加了 static 关键字，成为静态全局变量；因此不能扩展作用域到文件 ex4_16_2.cpp。去掉文件 ex4_16.cpp 中第二句前的 static 关键字，程序就正确了。

static 作用总结：static 关键字加在局部变量前是延长局部变量的存储期（但不改变其作用域）；static 关键字加在全局变量前是限制全局变量的作用域（但不改变其存储期）。

4.5.3 内部函数和外部函数

函数按其存储类别也可以分为两类：内部函数和外部函数。

1．内部函数

内部函数只能在定义它的文件中被调用，而在同一程序中的其他文件中不可调用。内部函数定义时，在函数类型前加 static，所以也称为静态函数，定义格式如下：

```
static  函数类型  函数名([参数列表])
{
    函数体
}
```

内部函数的作用域只限于定义它的文件，所以在同一个程序的不同文件中可以有相同名称的内部函数，它们互不干扰。

【例 4.17】 静态函数的例子。

程序代码如下：

```
//*****ex4_17.cpp*****     （文件 ex4_17.cpp 中的内容）
#include <iostream>
using namespace std;
static void fun();
void main()
{     fun();   }
static void fun()  //文件 ex4_17.cpp 中定义静态函数 fun
{ cout<<"fun is in ex4_17.cpp not in ex4_17_2.cpp!"<<endl; }
//*****ex4_17_2.cpp*****     （文件 ex4_17_2.cpp 中的内容）
#include <iostream>
using namespace std;
static void fun()  //文件 ex4_17_2.cpp 中定义静态函数 fun
{ cout<<"fun is in ex4_17_2.cpp!"<<endl; }
```

程序的运行结果如下：

```
fun is in ex4_17.cpp not in ex4_17_2.cpp!
```

程序表明，两个文件中都可出现同名 fun 函数，它们互不干扰。主函数 main 引用的是同一文件中的 fun 函数。

2．外部函数

外部函数是可以在整个程序各个文件中被调用的函数。

（1）外部函数的定义。在函数类型前加存储类型关键字 extern，或缺省存储类型关键字 extern，定义格式如下：

```
[extern] 函数类型 函数名([参数列表])
{
 函数体
}
```

（2）外部函数的声明。文件 A 在需要调用文件 B 中所定义的外部函数时，需要在文件 A 中用关键字 extern 对被调函数提出声明，声明格式如下：

```
extern 函数类型 函数名(参数类型列表)
```

【例 4.18】 文件 ex4_18.cpp 利用文件 ex4_18_2.cpp 中的外部函数实现求阶乘。

程序代码如下：

```
//*****ex4_18.cpp*****     （文件 ex4_18.cpp 中的内容）
#include <iostream>
using namespace std;
extern long fac(int);            //声明将要调用在其他文件中定义的 fac 函数
void main()
{    int n;
     cout<<"Input a integer to n: ";
     cin>>n;
     cout<<n<<"!="<<fac(n)<<endl;
}
//*****ex4_18_2.cpp*****     （文件 ex4_18_2.cpp 中的内容）
#include <iostream>
using namespace std;
extern long fac(int m)                //定义 fac 函数
{    int n; double s=1;
     for(n=1;n<=m; n++) s=s*n;
     return s;
}
```

程序的运行结果如下：

```
Input a integer to n: 10
10!=3628800
```

在文件 ex4_18_2.cpp 中可以将 extern 省略，定义为：

```
long fac(int m) { }
```

4.6 函数模板

模板是 C++支持参数化多态性的工具，具体讲是用来解决代码重用的一种工具。代码重用就是要求重用的代码有通用性，不受所使用的数据类型的影响。这种程序设计方法也称为参数化程序设计。

函数模板是对函数参数类型进行参数化，以获得具有相同形式的函数体。

模板有函数模板和类模板两种。类模板将在后续章节中介绍，本节内容只针对函数模板。

4.6.1 函数模板与模板函数

函数模板是一个通用函数，它适应一定范围内不同类型对象的操作。函数模板代表不同类型的一组函数，它们使用相同的代码，这样可以实现代码重用，避免了重复劳动。

1．函数模板的定义

函数模板的定义如下：

```
template  <参数化类型名表>
    类型标识符  函数名([形式参数列表])
{
    函数体
}
```

函数模板的定义还可以有如下形式：

```
template  <参数化类型名表> 类型标识符  函数名([形式参数列表])
{
    函数体
}
```

其中，参数化类型名表又称模板参数表，两边的尖括号是必要的。有多个参数（模板参数）时，之间用逗号分开。该表的格式如下：

```
class 标识符 1,class 标识符 2,……
```

其中，class 是类标识关键字，它的意义将在后续章节描述。

下面是一个函数模板的例子，该模板的功能是用来交换两个参数的值。

```
template <class T>
T swap(T a, T b)        //注意参数的写法，与前面函数定义的不同处
{
    T t;
    t=a; a=b; b=t;
}
```

该函数模板所允许的类型范围是对赋值运算符有意义的所有类型。

2．模板函数

函数模板定义了一类可以重载的模板函数，模板函数是函数模板的一个实例。也就是说一个函数模板可以生成多个可重载的模板函数。例如在上面的例子中，使用 int 代替模板参数 T，可以生成一个模板函数：

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

当然还可以用 float、double、char 等类型符号代替模板参数 T，生成不同类型的模板函数。可见，一个函数模板对应多个模板函数，各模板函数之间仅有参数类型不同的区别。

4.6.2 函数模板的使用

下面通过实例介绍函数模板的定义与使用。

【例 4.19】 交换两个变量值的通用函数。

程序代码如下：

```
#include <iostream>
using namespace std;
template <class T>
void swap(T *x, T *y)                    //形式参数使用传地址形式参数
{
    T t;
    t=*x; *x=*y; *y=t;                   //“*x”表示取变量 x 的内容（地址）的内容
}
void main()
{
    int a=1, b=2;
    double c=100.55, d=200.11;
    char e='A', f='B';
    cout<<"a="<<a<<"   b="<<b<<endl;
    swap(&a,&b);
    cout<<"a="<<a<<"   b="<<b<<endl;
    cout<<"c="<<c<<"   d="<<d<<endl;
    swap(&c,&d);
    cout<<"c="<<c<<"   d="<<d<<endl;
    cout<<"e="<<e<<"   f="<<f<<endl;
    swap(&e,&f);
    cout<<"e="<<e<<"   f="<<f<<endl;
}
```

程序的运行结果如下：

```
a=1   b=2
a=2   b=1
c=100.55   d=200.11
c=200.11   d=100.55
e=A   f=B
e=B   f=A
```

例 4.19 定义了一个函数模板 swap，有一个模板参数 T。主函数中使用了三个模板函数，它们分别以 int 型、double 型和 char 型替换模板参数 T。

要注意的是，例中的函数参数使用传地址参数形式，另外，swap 函数的函数名前使用了 void 型，通常情况下，其类型应该是函数值要求的类型，当然可以是模板参数的类型。

4.7 C++常用系统函数

C++编译系统提供了几百个函数供编程用户使用。本节将介绍一些常用的系统函数的使用方法。

使用系统函数要注意以下两点：

（1）C++编译系统提供的系统函数是按类分别存放在不同的.h 文件（头文件）中的。例如像平方根函数、绝对值函数、指数函数、对数函数、三角函数等常用数学函数放在 math.h 文件中（Visual C++放在 cmath.h 中）；有关字符串处理的函数放在 string.h 中；判断字母、数字、大小写字母、输入输出流的函数放在 iostream.h 中；有关格式控制的函数放在 iomanip.h 中。另外，不同版本的编译系统或不同厂商提供的编译系统提供的系统函数可能不尽相同，也可能放在不同的头文件中。因此，在使用系统函数之前，要阅读编译系统的使用手册，或通过联机帮助系统了解某个系统函数是否提供？它被存放在哪个头文件中？调用前要将包含被调用的系统函数所在的头文件用 include 语句包含进去。如果把头文件用错了，将导致编译不成功。

（2）调用系统函数之前，要了解该函数的功能、参数类型、返回值类型等信息，便于正确使用该函数。

4.7.1 常用数学函数

常用的数学函数包括：

（1）int abs(int num)：求绝对值。

（2）double acos(double arg)：求反余弦。

（3）double asin(double arg)：求反正弦。

（4）double atan(double arg)：求反正切。

（5）double cos(double arg)：求余弦。

（6）double exp(double arg)：求 e 的幂。

（7）double fabs(double arg)：求浮点数的绝对值。

（8）long labs(long num)：求长整型数的绝对值。

（9）double log(double num)：自然对数。

（10）double log10(double num)：以 10 为底的自然对数。

（11）double pow(double base, double exp)：求幂。

（12）double sin(double arg)：求正弦。

（13）double sqrt(double num)：求平方根。

（14）double tan(double arg)：求正切。

常用的数学函数在标准 C++文档中定义在 math.h 文件中，而 Visual C++ 6.0 系统中定义在 cmath.h 文件中。

【例 4.20】 常用数学函数使用示例。

程序代码如下：

```
#include <iostream>
#include <cmath>
using namespace std;
void main()
{
    int a=3, b=4;
    cout<<sqrt(a*a+b*b)<<endl;
```

```
    cout<<pow(sqrt(a*a+b*b),2)<<endl;
    cout<<exp(1)<<'\0'<<exp(1.5)<<endl;
}
```

程序的运行结果如下：

```
5
25
2.71828 4.48169
```

4.7.2 常用字符串处理函数

C++兼容标准 C 的字符串处理函数。本小节介绍其中几个比较常用的字符串处理函数。C++字符串处理函数定义在 string.h 文件中。

以下给出的函数调用形式中的参数 str、str1 和 str2，除特别声明外，均是指字符串常量、字符数组名或字符串存储空间开始地址（字符串指针），后两项内容将在后续章节介绍。

1．连接函数 strcat

函数原型：char *strcat(char *str1, const char *str2);

调用格式：strcat(str1,str2);

函数功能：将字符串 str2 连接到 strl 的后面，str2 的值不变。

2．字符串拷贝函数 strcpy 和 strncpy

函数原型：char *strcpy(char *str1,char *str2);

调用格式：strcpy(str1,str2);

函数功能：将字符串 str2 整个复制到字符数组 str1 中，str2 的值不变。

调用该函数时，一般 str1 是字符数组，且 str1 定义得足够大，以便能容纳被拷贝的 str2 的内容。例如：

```
strcpy(str1,"Changsha");
```

在某些应用中，需要将一个字符串的前面一部分拷贝，其余部分不拷贝。调用函数 strncpy 可实现这个要求。函数调用 strncpy(str1,str2,n)的作用是将 str2 中的前 n 个字符拷贝到 str1（附加'\0'）。其中 n 是整型表达式，指明欲拷贝的字符个数。如果 str2 中的字符个数不多于 n，则该函数调用等价于 strcpy(str1,str2)。

3．求字符串长度函数 strlen

函数原型：int strlen(char *str);

调用格式：strlen(str);

函数功能：求字符串的实际长度（不包括'\0'），由函数值返回。例如：strlen("good")函数值为 4。

4．字符串比较函数 strcmp

函数原型：int strcmp(char *str1,char *str2);

调用格式：strcmp(str1,str2);

函数调用 strcmp(str1,str2)比较两个字符串大小。对两个字符串自左至右逐个字符相比较（按字符的 ASCII 代码值的大小），直至出现不同的字符或遇到'\0'为止。如果全部字符都相同，则认为相等，函数返回 0 值；若出现不相同的字符，则以第一个不相同的字符比较结果为准。若 str1 的某个不相同字符小于 str2 的相应字符，函数返回一个负整数；反之，返回一个正整数。

注意，对字符串不允许进行相等“==”和不相等“!=”运算，必须用字符串比较函数对字符串作比较。

5．字符串大写字母转换成小写字母函数 strlwr

函数调用 strlwr(str)将 str 中的大写字母转换成小写字母，要求 str 不能是字符串常量。

6．字符串小写字母转换成大写字母函数 strupr

函数调用 strupr(str)将 str 中的小写字母转换成大写字母。要求 str 不能是字符串常量。

7．子串查找函数 strstr

函数调用 strstr(str1,str2)，在 str1 中查找是否包含子串 str2，若找到，则返回第一次出现 str2 的位置，否则，返回空指针 NULL。

8．int atoi(char *str)

将 str 字符串转换成整数。

9．long atol(char *str)

将 str 字符串转换成长整数。

10．double atof(char *str)

将 str 字符串转换成 double 型数值。

【例 4.21】 字符串处理函数使用示例。

程序代码如下：

```
#include <iostream>
#include <string>
using namespace std;
void main()
{
    char s[10]="First", s1[10]="";
    cout<<strlen(s)<<endl;
    cout<<strcat(s,"Name")<<endl;
    cout<<strcpy(s1,"Changsha")<<endl;
    cout<<strupr(s1)<<endl;
    cout<<atof("123.789")<<endl;
}
```

程序的运行结果如下：

```
5
FirstName
Changsha
CHANGSHA
123.789
```

一、选择题

1．以下有关函数定义和调用的说法中，正确的是（　　）。

A．函数的定义就是指“函数”这个名词的概念

B．函数的声明就是提前写出函数定义的首部

C．一个函数只能调用除自己以外的函数

D．函数的返回值只能返回给它的直接调用者

2．以下有关函数参数的叙述不正确的是（　　）。

A．函数的形参命名可以任意，只要符合标识符规则

B．实参只能是常数

C．形参的值与实参的值不一定时刻保持一致

D．函数参数的值也可以是内存单元地址

3．以下有关变量的时间与空间属性的叙述不正确的是（　　）。

A．变量的时间属性是指变量有一定的生存期

B．变量的空间属性是指变量有一定的作用域

C．static 关键字加在变量定义语句前只能使变量的生存期延长

D．静态全局变量是指其作用域最多只局限于本文件范围

4．以下有关函数存储类别的叙述不正确的是（　　）。

A．内部函数又叫做静态函数，定义时用到 static 关键字

B．内部函数不能被定义它的文件外的语句调用

C．必须加 extern 关键字定义的函数才是外部函数

D．外部函数可以被定义它的文件之外语句调用，只是调用前需要用 extern 声明

二、填空题

1．函数的声明语句类似于函数定义中的_____；但它们之间有如下区别：函数的声明语句中，形参名称可以_____，函数定义中的形参名称不可以_____；函数声明语句末尾_____，函数定义中的部分的末尾_____。

2．若有函数定义：float f(int x,char y){…}。将该函数声明为内联函数的语句为_____。

3．全局变量定义在_____位置，包括_____和_____两种。其中，后者的作用域不超出定义它的文件范围，且后者的定义比前者的定义要多一个_____关键字；而前者的作用域原则上可以扩展到程序所有_____中的所有_____，前提是只要在使用它的文件开头写形如_____格式的声明语句。

4．static 加在局部变量定义前，改变局部变量的_____但不改变它的_____；static 加在全局变量定义前，改变全局变量的_____但不改变它的_____。static 加在函数定义首部时，是限制函数的_____，使函数最多只能在_____范围内可以被调用。

三、程序阅读题

程序 1：

```
#include <iostream.h>
long f1(int p)
{
    long f2(int);
    long c=1,s;
    int i;
    for(i=1;i<=p;i++)
```

```
            c=c*i;
        s=f2(c);
        return s;
    }
    long f2(int q)
    {
        long r;
        r=q*q;
        return r;
    }
    void main()
    {
        long s;
        s=f1(2)+f1(3);
        cout<<"\ns="<<s<<endl;
    }
```

程序 2：若程序运行时输入：5↙

```
    #include <iostream.h>
    long fun(int n)
    {
        long f;
        if(n<0) cout<<"n<0,input error";
        else if(n==0) f=1;
        else f=fun(n-1)*(n-1)+n;
        return(f);
    }
    void main()
    {
        int n;
        long y;
        cout<<"\ninput a integer number:\n";
        cin>>n;
        y=fun(n);
        cout<<"y="<<y<<endl;
    }
```

程序 3：

```
    #include <iostream.h>
    int a=3,b=5;        //a、b 为外部变量
    max(int a,int b)   //a、b 为外部变量
    {       int c;
            c=a>b?a:b;
            return(c);
    }
    void main()
    {       int a=8;
            cout<<"max="<<max(a,b)<<endl;
    }
```

四、程序设计题

1．对任意给定的两个正整数 m、n，用函数实现求 s=m!+n!。要求编写 2 个自定义函数，其一为求两数之和，函数原型为：float add(int x, int y)；其二为求某数阶乘，函数原型为：float fac(int n)。主函数中给出实参，调用两者得到最终结果。

2．编写求[1,100]中所有素数之和的程序。其中，判断某个数 n 是否为素数用函数实现，函数原型可为：int isprime(int n)。

3．试分别用非递归与递归函数方式编写求 a 的 n 次幂的函数，函数原型为：float pow(float a,int n)，限定 n≥0。

4．试分别用非递归与递归函数方式编写求 s=1+2+…+n 的函数，函数原型为：float sum(int n)，限定 n≥1。

5．编程实现对用户输入的英文字符进行输出，其中，判断输入字符为英文字符的功能用内联函数实现。

6．将一个求阶乘的函数 fac（函数原型同第 1 题）专门写在一个文件 file1.cpp 中，定义为外部函数；然后在另一文件 file2.cpp 中计算 $y=a!/b^n$，其中，调用 fac 计算阶乘，调用本文件中的求幂运算函数 pow（函数原型同第 3 题）。