第 4 章　函数与预处理

前面已经介绍了常量、变量、表达式和语句，这些都是组成程序的基本要素。本章将着重介绍程序中的另外一个重要概念——函数。对于一个大型的程序，为便于实现，一般将其分为若干程序模块，每一个模块实现一个特定的功能。在 C++ 语言中，模块的功能由函数来实现。C++ 程序通过函数将上述基本要素结合在一起，完成一定的功能。函数的实现有利于信息的隐藏和数据共享，节省开发时间，增强程序的可靠性。

本章介绍函数的定义、声明、调用以及如何使用函数等内容，此外还将简略介绍有关预处理的概念。

本章内容提要：

- 函数的概念。
- 函数的定义和声明。
- 参数传递。
- 函数的嵌套调用。
- 函数的递归调用。
- 内联函数。
- 函数的重载。
- 有默认参数的函数。
- 变量的存储类别。
- 内部函数和外部函数。
- 预处理命令。

4.1　函数的概念

“函数”这个词是从英文 function 翻译过来的，其实 function 的原意是“功能”。顾名思

义，一个函数就是一个功能。当然，读者可能会联想到数学中的函数。数学中的函数可以根据输入的数值求得一个确定的值。与此类似，C++中的函数也可以根据输入的数据，返回一个结果值。不同的是，C++中的函数涵盖的意义更加广泛，定义也更加灵活，不仅可以求值，还可以执行一组相关的操作。在后文中，如不加说明，所涉及函数都是指C++中的函数。一个函数就是一个语句的集合，这些语句组合在一起完成一项操作。系统按顺序执行多条语句以返回所需要的结果。

4.1.1 使用函数的必要性

客观地说，函数是C++程序的构成基础。从前面所遇到的程序来看，一个C++程序可由一个主函数main和若干子函数构成，而且必须有一个main函数。因为任何一个C++程序都是从主函数(即main)的左花括号开始执行，一直到main函数的右花括号为止。函数也是程序中最小的模块，可以重复使用。

4.1.2 函数的组成部分

函数由函数头和函数体两部分构成，而函数头又由返回值类型、函数名和参数列表构成。函数的一般形式如下。

```
([数据类型]) <函数名> [<形式参数列表>]        //函数头
{
    语句                                      //函数体
}
```

【例4-1】 求出两个整数中的较小值。

```
#include <iostream>
using namespace std;
int min(int x,int y)                          //定义有参函数min
{
    int z;
    z=x<y? x:y;
    return z;                                 //函数返回一个较小的数
}
int main()
{
    int a,b,c;
    cout <<"please input two integers: " <<endl;  //输出提示语句
    cin >>a >>b;
    c=min(a,b);                               //调用min函数
    cout <<"min=" <<c <<endl;
    return 0;
}
```

上述代码中的第 3 行定义了一个函数，它的返回值类型为 int 型，函数名为 min 且参数列表不为空。

程序运行结果如图 4-1 所示。

```
please input two integers:
3
6
min=3
请按任意键继续. . .
```

图 4-1　【例 4-1】运行结果

1）数据类型

数据类型是指函数返回值的类型。例如，【例 4-1】中的 min 函数返回一个 int 型数据（即 int 型 z），就称该函数的返回值类型为 int 型。如果某个函数不返回任何值，则其返回值类型是 void，定义这样的函数的目的不是求一个值，而只是执行一组操作。在此读者只需记住：当函数名前面出现 void 型返回值时，函数体的末尾无 return 语句。

注意： C 语言中规定，如果在定义函数时不指定数据类型，系统会隐式指定数据类型为 int 型，因此，在 C 语言中，int min(int x,int y)可以简写为 min(int x,int y)，但 C++ 取消了这一规定，要求在定义函数时必须指定数据类型，这样做更加严格和安全。

2）函数名

函数名就是函数的名字，即函数的标识符。既然是标识符，就必须遵循标识符的命名规则。由变量的标识符可知，函数的标识符也只能由字母、数字以及下划线组成，并且不能以数字开头。值得注意的是，在给函数命名时，应尽量使函数名体现出该函数的功能或寓意，因为一个准确、有意义的函数名可以提高程序的可读性。

3）形式参数

形式参数简称形参。形参列表是包含在函数名后圆括号中的 0 个或多个以逗号分隔的变量定义。它规定了函数将从调用函数中接收几个数据及它们的类型。当函数形参个数为零时，称为无参函数；函数形参个数不为零时，称为有参函数。之所以称为形参，是因为在定义函数时系统并不为这些参数分配存储空间，只有被调用时，向它传递了实际参数（简称实参），才为形参分配存储空间。当然在调用结束后，形参所占用的存储单元又会被释放。在【例 4-1】中，变量 x、y 是形参，变量 a、b 是实参。

注意： 当形参个数不为 0 时，在定义相同类型的形参时，对于参数列表中的每个参数，都要求明确指定其类型，如 min(int x,int y)不能写成 min(int x,y)。

4）函数体

函数体是一个用一对花括号“{}”括起来的语句序列，它描述了函数实现一个功能的过程。当函数体中有 return 语句时，函数执行完 return 语句后结束；当函数体中无 return 语句，即函数的返回值类型为 void 时，函数执行完最后一条语句遇到右花括号“}”时结束。

在此对 return 语句作一些说明。

（1）一般来说，函数的返回值是通过函数中的 return 语句获得的。return 语句将被调用函数中的一个确定值带回主调函数中，且 return 语句执行后，函数结束。例如：

```
{
    …
    return 0;                        //函数返回
    cout <<"函数结束!" <<endl;      /*这行语句不会执行，因为函数已经返回，
                                       并结束运行*/
}
```

又例如在【例 4-1】中，return(z)将 z 的值返回主调函数中，min 函数结束。

(2)return 语句的一般形式有如下几种。

第一种：

return 表达式；

例如，【例 4-1】中的 min 函数可以改写为：

```
int min(int x,int y)
{
    return x<y? x:y;
}
```

第二种：

return(表达式)；

“return 表达式；”与“return(表达式)；”等价。

例如，“return x<y? x:y;”等价于“return (x<y? x:y);”。

第三种：

return;

(3)如果函数要返回数据，则函数体中至少需要一条 return 语句。

(4)如果函数不需要返回数据，即函数的返回值类型是 void，则在函数体中不需要return 语句。函数执行完最后一条语句，遇到右花括号“}”结束。

(5)一个函数体中可以有多个 return 语句，但每次只能通过一个 return 语句执行返回操作。例如：

```
int max(int x,int y)                //返回两个整数中的较大值
{
    if(x>y)
        return x;
    else
        return y;
}
```

一个函数体中有多个 return 语句时，如【例 4-1】中有两个 return 语句，每个 return 语句的返回值类型都应与函数定义一致。

4.2 函数的定义和声明

函数声明也称为函数原型声明，是指在函数尚未定义的情况下，先将函数的形式告知编译系统，以便编译能够正常进行。函数声明包括返回值类型、函数名和参数列表。值得注意的是，函数声明仅仅是一条语句，因此在函数声明后一定要加上分号。与函数声明不同的是，函数的定义除包含函数头外，还包括函数体，是一个独立完整的函数单位。函数的定义又称函数实现。

4.2.1 函数的定义

函数可以是系统预定义的，也可以是用户自定义的。前者称为系统函数或标准库函数，后者称为用户自定义函数。

1）系统函数

系统函数是由编译系统提供的，用户不必自己定义这些函数就可以直接使用它们。当然在调用时也无须声明，只需用＃include 命令包含相应的头文件即可。例如，＃include ＜cmath＞，其中 cmath 是一个头文件，在 cmath 文件中包括了数学库函数所用到的一些宏定义信息和对函数的声明。

2）用户自定义函数

用户自定义函数时应满足函数的 4 个组成部分：返回值类型、函数名、参数列表和函数体。从函数的形式看，可分为有参函数和无参函数。例如：

```
/* 定义有参函数 f1 */
int f1(int x)      //有参函数，函数头
{                  //函数体
    …
    return 0;      //返回 0 值，函数结束
}
```

```
/* 定义无参函数 f2 */
void f2()          //无参函数，函数头
{                  //函数体
    …
}                  //无返回值，函数结束
```

上面两个程序，都是函数的完整定义。

函数不能嵌套定义，即在函数体中不能定义另外一个函数。下面的情形是不允许出现的。

```
void function1()
{
    int function2()          //错误！函数定义不允许嵌套
    {
        …
        return 0;
    }
}
```

虽然函数不允许嵌套定义，但可以在函数体中声明另外一个函数。下面的情形是可以出现的。

```
void function1()
{
    int function2();          //正确！在函数 function1 中声明函数 function2
}
```

读者不妨自己上机试一试。

4.2.2　函数的参数列表

前面已经提到，在函数的声明和定义中，函数的参数列表可以为空(即无参函数)，也可以不为空(即有参函数)。如果函数为无参函数，则可用空参数列表或带一个 void 关键字的参数列表。例如：

```
int function()
{
    …
    return 0;
}
int function(void)
{
    …
    return 0;
}
```

这两种定义方式是等价的，都表示不接受任何参数。如果函数为有参函数，则在声明函数时，参数列表中的每个参数可以有名字，也可以没有名字。例如：

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

这两种声明方式是等价的，都表示接受两个整型参数。但在定义函数时，参数列表中的每个参数就必须给出名字，形如“int min(int ,int);”在编译时就会出错。此外，此处的参数名也有其特殊的意义：一方面，便于阅读；另一方面，可以提示该参数的含义，开发者可以根据参数的名字传入合适的参数。

4.2.3　函数的声明

C++标准规定：函数在调用之前，必须先声明。在 C++ 中，当函数定义在前，函数调用在后时，调用前可以不必声明，因为编译器已经知道了该函数的全部信息。

【例 4-2】　求两个整数之和。

```
#include <iostream>
using namespace std;
int add(int x,int y)                    //定义求两个整数相加的函数
{
    return (x+y);                       //函数返回两个整数的和
}
int main()                              //主函数
{
    int a,b,c;                          //声明3个整型变量
```

```
    cout <<"please enter two integers: " <<endl;     //输出提示语句
    cin >>a >>b;                                     //输入整数
    c=add(a,b);                                      //调用 add 函数,求和
    cout <<"c=" <<c <<endl;                          //输出相加后的值 c
    return 0;                                        //主函数返回
}
```

程序运行结果如图 4-2 所示。

```
please enter two integers:
4
6
c=10
请按任意键继续. . .
```

图 4-2　【例 4-2】运行结果

当编译器由上而下顺序编译下来,到第 12 行调用函数 add 时,因 add 函数接受的参数类型、参数个数及其返回值都已经明确,所以不需要另外声明。当一个函数的定义在后,调用在前时,在调用前必须声明函数的原型,否则编译将会出错。

【例 4-3】 求两个整数之和。

```
#include <iostream>
using namespace std;
int main()                                           //主函数
{
    int a,b,c;                                       //声明 3 个整型变量
    cout <<"please enter two integers: " <<endl;     //输出提示语句
    cin >>a >>b;                                     //输入整数
    int add(int x,int y);                            //声明求两个整数相加的函数
    c=add(a,b);                                      //调用 add 函数,求和
    cout <<"c=" <<c <<endl;                          //输出相加后的值 c
    return 0;                                        //主函数返回
}
int add(int x,int y)                                 //定义求两个整数相加的函数
{
    return (x+y);                                    //函数返回两个整数的和
}
```

如果输入 a 的值为 4,b 的值为 6,那么程序的运行结果将与【例 4-2】的结果相同。【例 4-3】中的第 8 行就是函数 add 的声明,当编译器编译到这行代码时,就会知道 add 函数的基本信息:函数名为 add,需要接受两个整型参数,返回一个整数。

对比【例 4-2】和【例 4-3】之后,或许读者会提出这样的疑问:编程时都把函数定义写在调用之前,把 main 函数写在最后,岂不是可以省略函数声明这一步了吗?但是,这样做在安排函数顺序时要投入很多精力,在复杂的调用中,一定要考虑好谁先谁后,否则将发生错误。而且,C++程序都是从 main 函数开始执行的,将 main 函数放在程序的开头可提高程序的可读性。因此一般都把 main 函数写在最前面。读者应养成对所有用到的函数作声明的习惯。

下面再对函数的声明作一些说明。

(1)函数声明的位置比较灵活,可以出现在该函数调用前的任何位置,且对函数的原型

声明次数没有限制。

(2)其实可以简单地照写已定义的函数的首部,再加一个分号,就是对函数的声明。

(3)如果一个函数不是被多个函数调用,一般是在调用该函数的前一行代码处声明。例如,【例 4-3】中的第 8 行对 add 函数的声明就是较好的例子。

4.2.4　在头文件中声明函数

C++支持所谓的分别编译,这样程序可以由多个文件组成。由前面的例题可知,对于只有一个源文件的简单程序,可以不需要函数声明,只要保证函数在调用之前定义即可。但用户在以后的学习和工作中,经常面对一些复杂的大型程序,往往有很多源文件,而且一个源文件中定义的函数,往往会被其他源文件使用,因此就需要在其他源文件中声明该函数。在有多个源文件的程序中,往往把函数的声明放在头文件中,而把函数的定义放在源文件中。当其他源文件要声明函数时,只需用#include 命令包含头文件即可。

下面用求两个整数之和的 add 函数来举例说明。

(1)编写一个头文件 function.h,用来声明 add 函数,其内容如下。

```
int add(int x,int y);                 //声明一个求两个整数之和的函数 add
```

(2)编写一个源文件 function.cpp,用来定义 add 函数,其内容如下。

```
int add(int x,int y)                  //定义 add 函数
{ return (x+y); }                     //返回两个数之和
```

(3)在其他源文件中调用 add 函数时,先要包含头文件 function.h,如在主函数中调用该函数。

```
#include "function.h"                 //包含头文件 function.h,即声明函数 add
…
void main()                           //主函数,无返回值
{
    cout <<add(3,5) <<endl;           //调用 add 函数,并输出结果
}                                     //主函数结束
```

注意:C++标准规定,标准头文件括在尖括号(<>)中,用户自定义的头文件放在双引号(" ")中。

4.3　函 数 调 用

定义一个函数的目的是使用它,要想使用函数,就要调用它。调用函数时,开发者应当按照形参列表为函数传入所需的实参。

4.3.1　函数调用的一般形式

函数调用的一般形式如下。

函数名(<实参列表>)

例如：

```
c=add(a,b);          //调用 add(a,b),并将返回值赋给整型变量 c
cout <<add(a,b);     //打印调用函数 add(a,b)后所返回的值
```

以上两种方式是比较常见的，当然也是针对形参列表不为空而言的。如果调用的是无参函数，为了与形参一一对应，实参列表也应为空，但括号不能省略。

【例 4-4】 在 main 函数中调用 print_message 函数。

```
#include <iostream>
using namespace std;
void print_message();                         //函数声明
int main()
{
    print_message();                          //调用 print_message 函数
    return 0;                                 //主函数返回
}
void print_message()                          //定义 print_message 函数
{
    cout <<"********************" <<endl;
    cout <<"  Welcome to C++!" <<endl;
    cout <<"********************" <<endl;
}
```

程序运行结果如图 4-3 所示。

图 4-3 【例 4-4】运行结果

在【例 4-4】中，当程序执行到第 6 行时，程序控制流就会转到被调用的 print_message 函数(即跳转至第 9 行)。当执行到 print_message 函数结尾的花括号(第 14 行)时，又将控制权交还给调用者，即此程序中的 main 函数。

4.3.2 函数的形参和实参

在 C++ 中，形参就是函数定义时的参数，实参是函数调用时传入的参数。在 4.1.2 中也曾提到，在定义函数时系统并不为形参分配存储空间，只有被调用时向它传递了实参，才为形参分配存储空间。在调用结束后，形参所占的存储单元又会被释放。实参才是程序运行时实际存在的参数。例如，在【例 4-3】中，参数 x 和 y 就是形参，而通过调用 add 函数传入的变量 a 和 b 是实参。

函数形参和实参的说明如下。

(1)形参和实参是相对的概念,有时会相互转化。例如:

```
void count_add(int a,int b)            //count 函数,计算两个整数的和
{
    int c=add(a,b);                    //调用 add 函数,求两个整数 a,b 的和
    cout <<"c=" <<c <<endl;            //输出和
}
```

对于 count_add 函数,a 和 b 是形参;对于 add 函数(第 3 行的调用),a 和 b 是实参。

(2)实参的类型与形参的类型应相同或赋值兼容,且实参的个数与形参的个数必须一致,即类型相同或赋值兼容,个数相等。

4.3.3 值传递

在调用有参函数时,实参变量对形参变量的数据传递是一一对应的"值传递",此时编译系统将临时给形参分配存储单元。但应当注意的是,实参单元与形参单元是不同的单元。例如,当【例 4-3】中的程序执行到第 9 行"c=add(a,b);"时,若 a=4,b=6,则将实参 a 和 b 的值 4 和 6 传递给对应的形参 x 和 y,如图 4-4 所示。

调用结束后,形参单元将被释放,实参单元仍保留并维持原值。由于实参与形参之间的值传递方式,使得实参与形参之间的传递是单向的,即只由实参传给形参,而不能由形参回传给实参。例如,若在执行 add 函数的过程中,形参 x 和 y 的值变为 8 和 9,调用结束后,实参 a 和 b 仍为 4 和 6,如图 4-5 所示。

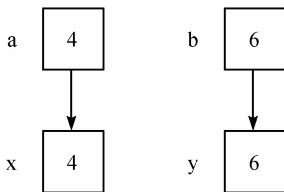

图 4-4　值传递 1

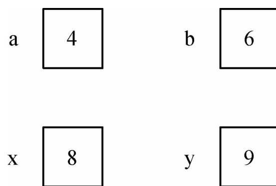

图 4-5　值传递 2

【例 4-5】 交换两个变量的值。

```
#include <iostream>
using namespace std;
void swap(int x,int y);                      //函数声明
int main()
{
    int a=5,b=8;
    swap(a,b);                               //调用 swap 函数
    cout <<"a=" <<a <<endl;
    cout <<"b=" <<b <<endl;
    return 0;
```

```
}
void swap(int x,int y)                              //定义 swap 函数
{
    int temp=x;
    x=y;
    y=temp;
}
```

程序运行结果如图 4-6 所示。

```
a=5
b=8
请按任意键继续. . .
```

图 4-6　【例 4-5】运行结果

程序的目的是输出 a 和 b 交换后的值，即输出“a＝8”和“b＝5”，但实际输出结果却是“a＝5”和“b＝8”。在 swap 函数中，虽然形参 x 和 y 的值交换了，但由于实参与形参之间是单向传递的，形参值的改变不会影响实参的值，所以实参 a 和 b 还是保持其原有的值不变。

4.3.4　参数类型检查

C++ 中的参数类型检查通常是针对有参函数而言的。在调用函数时，编译器会自动检查实参与形参的类型是否相同或赋值兼容。如果两者都不满足，那么编译器会报告错误，编译就无法进行。例如：

```
void add(int x,int y)   //定义两个整数相加的函数
{
    return (x+y);       //函数返回两个整数相加后的值
}
…
int z;                  //定义一个整型变量用以接收 add 函数的返回值
z=add(2,3);             //传入类型，类型相同(即 2、3 和 x、y 都是整型)
z=add(5.4,6.7);   //double 型转换为 int 型，5.4 变为 5 赋给 x，6.7 变为 6 赋给 y
```

4.3.5　使用默认实参

一般情况下，形参的值是靠实参通过数据传递(值传递)获得的。但是，当多次调用同一函数，而其中又有一个或几个参数，它们传递进来的实参值多次都是相同时，就可以在函数定义或声明时赋给形参一个默认值。

默认实参其实就是形参的默认值。指定默认实参的方法类似于初始化一个变量。例如：

```
void add(int x,int y=6);
```

上述 add 函数的声明中，形参 x 是一个普通的参数，而形参 y 则是一个具有默认实参的参数，这个默认实参就是 6。如果在调用 add 函数时没有给形参 y 传递实参值，则形参 y 的值取默认值 6。

由于实参与形参的结合顺序是从左自右进行的，因此指定默认值的参数必须放在形参列表中的最右端，否则会出错。例如：

```
void fun1(int a,int b=2,int c=3);            //正确
void fun2(int a=1,int b=2,int c);            //不正确
void fun3(int a=1,int b,int c=3);            //不正确
```

既可以在函数声明中指定默认值，也可以在函数定义中指定默认实参。通常，应在函数声明中指定默认实参，并将该声明放在合适的头文件中。此外，在一个文件中，一个参数只能被指定一次默认实参。但可以在多次声明中依次向前指定其他参数的默认值。例如，在头文件 method.h 中声明一个名为 f 的函数。

```
int f(int x,int y,int z=10);
```

在源文件 method.cpp 中定义这个函数。

```
#include "method.h"            //包含头文件 method.h
int f(int x,int y,int z=10)    /*错误，在同一文件中形参 z 被重复指定为默认实
                                 参，应去掉“=10”*/
{
    …
}
```

在源文件 main.cpp 中使用函数 f。

```
#include"method.h"             //包含头文件 method.h
int f(int x,int y=12,int z);  //正确，重新声明函数 f 时，向前指定 y 的默认实参 12
```

注意：如果在函数定义的形参列表中指定默认实参，那么只有在包含该函数定义的源文件中调用该函数时，默认实参才有效。

4.3.6 调用有默认参数的函数

指定了默认实参的函数虽然并不普遍，但在多数情况下仍然是适用的。调用有默认参数的函数时，可以省略有默认值的实参。

对于在定义或声明中带有默认实参的函数，如果在调用时没有给相应的形参传递实参值，则使用默认实参作为该形参的值。例如：

```
void output_message(int num1=10,double num2=1.2345)
                              //指定参数 num1 和 num2 的默认实参
{
    cout <<num1 <<endl;    //输出参数的值
    cout <<num2 <<endl;
}
…
```

```
output_message();//调用函数,没有指定实参,函数以默认值 10 和 1.234 5 作为实参
```

上述代码的输出的结果如下。

```
10
1.2345
```

但如果在调用时给形参传递了实参值,则调用者指定的实参将会覆盖形参的默认值,也就是说函数在运行时会按照调用者指定的值运行。例如,调用 output_message 函数。

```
output_message(20,2.3456);            //覆盖 num1 和 num2 的默认实参
output_message(20);
                    //实参数少于形参数,只覆盖 num1 的默认实参,使用 num2 的默认实参
```

说明:

(1)如果函数定义在函数调用之前,则应在函数定义中给出默认值;如果函数定义在函数调用之后,则应在函数调用之前的函数原型声明中给出默认值。是否在函数定义中给出默认值,要根据具体的编译系统而定,因为不同的编译系统有不同的处理规则。

(2)一个函数不能既作为重载函数,又作为有默认参数的函数,否则容易出现二义性,使系统无法执行。因为如果调用函数时少写一个参数,系统无法判定是利用重载函数还是利用有默认参数的函数。

【例 4-6】 求两个或 3 个整数中的最大值,用带有默认参数的函数实现。

```
#include <iostream>
using namespace std;
int max(int ,int ,int =0);                //函数声明,并指定了一个默认实参
int main()
{
    int a,b,c;
    cout <<"please input three integers:" <<endl;   //输出提示语
    cin >>a >>b >>c;
    cout <<"max(a,b)=" <<max(a,b) <<endl;
    //函数调用,实参数少于形参数,使用默认实参
    cout <<"max(a,b,c)=" <<max(a,b,c) <<endl;
    //函数调用,实参数等于形参数,实参 c 将覆盖默认实参
    return 0;
}
int max(int x,int y,int z)
{
    if(x<y)
        x=y;
    if(x<z)
    {
        x=z;
    }
    return x;
```

}

程序运行结果如图 4-7 所示。

```
please input three integers:
36
-17
69
max(a,b)=36
max(a,b,c)=69
请按任意键继续. . .
```

图 4-7 【例 4-6】运行结果

4.3.7 函数的嵌套调用和递归调用

1)函数的嵌套调用

C++ 中的所有函数都是平行的，即在定义函数时是互相独立的。一个函数定义内部包含另一个函数定义，称为嵌套定义。C++ 规定不能嵌套定义函数，但可以嵌套调用函数。所谓嵌套调用，是指在调用一个函数的过程中又调用另一个函数，其示意图如图 4-8 所示。

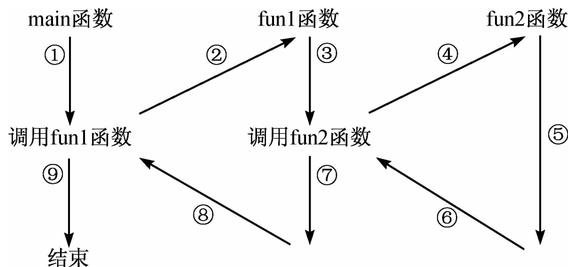

图 4-8 嵌套调用

图 4-8 表示的是两层嵌套(连同 main 函数共 3 层函数)的执行过程示意图，具体过程描述如下。

(1)执行 main 函数的开头部分。

(2)遇到调用 fun1 函数的语句，转去执行 fun1 函数。

(3)执行 fun1 函数的开头部分。

(4)遇到调用 fun2 函数的语句，转去执行 fun2 函数。

(5)执行 fun2 函数的函数体，如果再无其他嵌套的函数，则完成 fun2 函数的全部操作。

(6)返回调用 fun2 函数处，即返回 fun1 函数。

(7)继续执行 fun1 函数尚未执行的部分，直到 fun1 函数结束。

(8)返回 main 函数中原来调用 fun1 函数处。

(9)继续执行 main 函数的剩余部分直到结束。

嵌套调用在 C++ 编程中经常使用。需要注意的是，在程序中实现函数嵌套调用时，如果被调用函数是调用在前，定义在后，那么就需要声明每一个被调用的函数。下面举例说明函数的嵌套调用。

【例 4-7】　编写程序，求两个数 a、b 的最小公倍数和最大公约数。

利用欧几里得算法（又称辗转相除法）实现的过程如下。

（1）求 a 除以 b 的余数 r。

（2）如果余数 r 为 0，则 b 是最大公约数，算法结束；否则执行下一步。

（3）将除数作为新的被除数，余数作为新的除数，即执行“a=b;b=r;”，然后转到步骤(2)。

具体程序如下。

```
#include <iostream>
using namespace std;
int gcd(int a,int b);                        //声明求最大公约数的函数原型
int lcm(int a,int b);                        //声明求最小公倍数的函数原型
void main()
{
    int a,b;
    cout <<"please enter two integers:" <<endl;     //输出提示语
    cin >>a >>b;
    cout <<"the greatest common divisor is:"<<gcd(a,b) <<endl;
                                                     //调用 gcd 函数并输出
    cout <<"the least common multiple is:" <<lcm(a,b) <<endl;
                                                     //调用 lcm 函数并输出
}
int gcd(int a,int b)             //定义 gcd 函数
{
    int r;
    while((r=a%b)! =0)
    {
        a=b; b=r;
    }
    return b;
}
int lcm(int a,int b)           //定义 lcm 函数
{
    int g;
    g=gcd(a,b);                //在求最小公倍数函数中调用求最大公约数函数
    return (a*b/g);
}
```

程序运行结果如图 4-9 所示。

```
please enter two integers :
16
24
the greatest common divisor is:8
the least common multiple is:48
请按任意键继续. . .
```

图 4-9 【例 4-7】运行结果

【例 4-8】 用弦截法求方程 $f(x)=x^3-8x^2+12x+16=0$ 的根。结合数学知识，其解决方法如下。

(1)确定求值区间。输入两个不同的点 x_1,x_2，直到 $f(x_1)$和 $f(x_2)$异号为止。因为一旦 $f(x_1)$和 $f(x_2)$异号，(x_1,x_2)区间内必有一个根。但应注意 x_1,x_2 的值不能相差太大，以保(x_1,x_2)区间内只有一个根。

(2)连接$(x_1,f(x_1))$和$(x_2,f(x_2))$两点，此线(即弦)与 x 轴相交于 x，如图 4-10 所示。其中，x 的坐标可由下式求出。

$$x=\frac{x_1\cdot f(x_2)-x_2\cdot f(x_1)}{f(x_2)-f(x_1)}$$

再由 x 求出 $f(x)$。

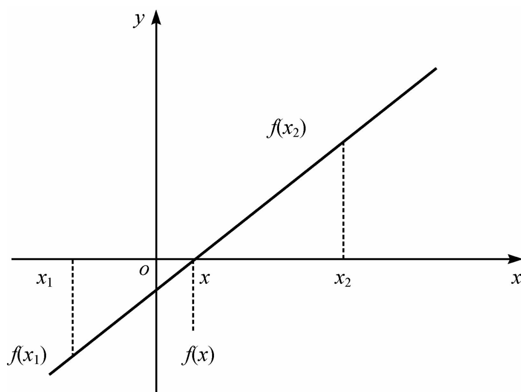

图 4-10 弦截法示意图

(3)若 $f(x)$与 $f(x_1)$同号，则根必在(x_1,x)区间内，此时将 x 作为新的 x_1。如果 $f(x)$与 $f(x_2)$同号，则表示根在(x,x_2)区间内，将 x 作为新的 x_2。重复步骤(2)和步骤(3)，直到 $|f(x)<\varepsilon|$ (其中的 ε 为一个很小的正数，如 10^{-6})为止。此时认为 $f(x)\approx 0$。

这就是弦截法的算法，在程序中分别用以下几个函数来实现各部分的功能。

(1)用 $f(x)$代表 x 的函数：$x^3-8x^2+12x+16$。

(2)用函数 xpoint(x_1,x_2)求$(x_1,f(x_1))$和$(x_2,f(x_2))$两点的连线与 x 轴的交点 x 的坐标。

(3)用函数 root(x_1,x_2)求(x_1,x_2)区间的实根。显然，执行 root 函数的过程中要用到函数 xpoint，而执行 xpoint 函数的过程中要用到 f 函数。

根据以上算法，可编写出如下程序。

```
#include <iostream>
#include <cmath>
#include <iomanip>
```

```
using namespace std;
double f(double);
double xpoint(double,double);
double root(double,double);
void main()
{
    double x1,x2,x,f1,f2;
    do          //输入两个不同点 x1、x2,直到 f(x1)和 f(x2)异号为止
    {
        cout <<"please input x1,x2:" <<endl;
        cin >>x1 >>x2;
        f1=f(x1);
        f2=f(x2);
    }
    while( (f1 * f2) >0);
    x=root(x1,x2);
    cout <<setiosflags(ios::fixed) <<setprecision(6);  //指定输出 6 位小数
    cout <<"a root of equation is:" <<x <<endl;
}
double f(double x)     //定义函数 f,以实现 f(x)=0
{
    return (x * x * x-8 * x * x+12 * x+16);
}
double xpoint(double x1,double x2)      //定义 xpoint 函数,求出弦与 x 轴的交点
{
    double y;
    y=( x1 * f(x2)-x2 * f(x1) )/( f(x2)-f(x1) );  //在 xpoint 函数中调用 f 函数
    return y;
}
double root(double x1,double x2)              //定义 root 函数,求近似根
{
    double x,y,y1;
    y1=f(x1);
    do
    {
        x=xpoint(x1,x2);                     //在 root 函数中调用 xpoint 函数
        y=f(x);                              //在 root 函数中调用 f 函数
        if(y * y1>0)
        {
            y1=y;
```

```
                    x1=x;
                }
    else
                    x2=x;
        }
        while(fabs(y)>=1e-6);
        return x;
    }
```

程序运行结果如图 4-11 所示。

```
please input x1,x2:
3
-4
a root of equation is:-0.828427
请按任意键继续. . .
```

图 4-11 用弦截法求方程根的结果

注意：main 函数可以调用其他函数，各函数间也可以互相调用，但不能调用 main 函数。

2)函数的递归调用

直接或间接调用自己的函数称为递归函数。直接调用自身的函数称为直接递归调用，如在执行函数 fun1 的过程中，又要调用 fun1 函数，如图 4-12 所示。间接调用自身的函数称为间接递归调用，如在执行函数 fun1 的过程中要调用函数 fun2，而在执行函数 fun2 的过程中又要调用函数 fun1，如图 4-13 所示。

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

图 4-12 函数的直接递归调用

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

图 4-13 函数的间接递归调用

下面看一个简单的递归调用例子——用递归调用函数求 n! 的值。

我们知道 5! =5＊4!，也就是说，想要求 5! 的值必须知道 4! 的值，而 4! =4＊3!，同理，要求 4! 的值又必须知道 3! 的值，而 3! =3＊2! ……以此类推，只要知道了 1!（值为 1)，反推回去就求得了 5!，这就是递归思想，该过程的示意图如图 4-14 所示。

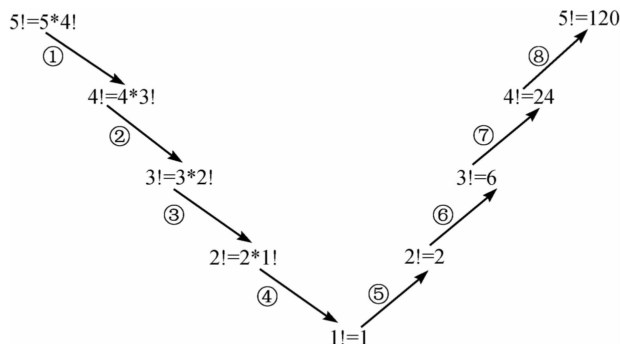

图 4-14 递归调用求 5! 的值示意图

由图 4-14 可知，求解过程分为两个阶段：第 1 阶段（即①～④）是回推，即将 5！表示为 4！的函数，而 4！仍然不知道，还要回推到 3！……直到 1！。此时 1！已知，不必再向前推了。然后开始第 2 阶段（⑤～⑧），采用递推方法，从已知的 1！推算出 2！（值为 2），从 2！再推算出 3！（值为 6）……一直推算出 5！（值为 120）为止。显然，一个递归调用问题可以分为回推和递推两个阶段。

注意：如果要求递归过程不是无限制地进行下去，那么就必须具有一个结束递归过程的条件。通常使用 if 语句来控制。

【例 4-9】 用递归调用函数求 $n!$。其中：

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

编写程序如下。

```
#include <iostream>
using namespace std;
long factorial(int);                  //函数声明，形参为一个整型变量
int main()
{
    int n;                            //n 为需要求阶乘的整数
    long y;
    cout <<"please input an integer:" <<endl;
    cin >>n;
    y=factorial(n);                   //调用递归函数 factorial，将返回值赋给 y
    cout <<n <<"! =" <<y <<endl;
    return 0;
}
long factorial(int n)                 //定义递归函数
{
    long fac;
    if(n<0)                           //如果输入负数，报错并以-1 作为返回值
    {
        cout <<"n<0,wrong number!" <<endl;
        fac=-1;
    }
    else if(n==0||n==1)               //0! 和 1! 相等，都为 1
    {
        fac=1;
    }
    else
        fac=factorial(n-1)*n;   //n>1 时，进行递归调用
```

```
    return fac;                          //将 fac 的值作为函数返回值
}
```

程序运行结果如图 4-15 所示。

```
please input an integer:
5
5! =120
请按任意键继续. . .
```

图 4-15 【例 4-9】运行结果

【例 4-10】 用递归调用函数求两个数的最大公约数。

结合【例 4-7】中介绍的欧几里得算法，求两个数的最大公约数的过程可以描述为如下的式子。

$$\gcd(a,b)=\begin{cases} b & a\%b=0 \\ \gcd(b,a\%b) & a\%b\neq 0 \end{cases}$$

编写程序如下。

```
#include <iostream>
using namespace std;
int gcd(int a,int b);                    //递归函数 gcd 原型声明
void main()
{
    int a,b,g;
    cout <<"please enter two integers:" <<endl;   //输出提示语
    cin >>a >>b;
    g=gcd(a,b);            //调用递归函数并将返回值赋给变量 g
                           //输出最大公约数
    cout <<"the greatest common divisor is:" <<g <<endl;
}
int gcd(int a,int b)
{
    if(a%b==0)             //如果 a、b 的余数为 0,则函数返回 b 的值
        return b;
    else                   //如果 a、b 的余数不为 0,则返回 gcd 函数继续执行
        return gcd(b,a%b);
}
```

程序运行结果如图 4-16 所示。

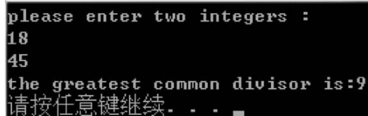

图 4-16 【例 4-10】运行结果

由以上两个例题可以看出，当使用递归调用函数解决问题时，需要满足如下两个条件。

(1)应能减小问题的规模。

(2)应能确定终结条件。

还需说明的是，在实现递归调用时，需要大量的额外开销，执行效率较低。但随着计

算机性能的快速提升，大家首先考虑的往往不再是效率问题，而是程序的可读性问题。用递归调用函数来处理问题，符合人们的思路，程序容易理解。因此，建议读者优先考虑用递归调用函数编程。

4.3.8　内联函数

通常的函数调用是将程序执行顺序转到被调用函数中执行，待被调用函数执行完毕后，再返回调用函数继续执行，见图 4-8。这种转移操作会消耗一定的时间，尤其是当某些函数体代码不是很大，而又需要频繁调用时，就大大降低了程序的执行效率。为了解决这个问题，C++引入了内联函数。

C++规定，如果在函数的定义或声明前加上关键字 inline，就称为内联函数。内联函数的执行机理，是在编译时将所调用函数的函数体代码直接嵌入主调函数中。例如，对于 add 函数，如果其定义为：

```
int add(int x,int y)
{
    return(x+y);
}
```

在 main 函数中声明为：

```
inline int add(int x,int y);
                        //在函数声明前加上关键字 inline,add 函数成为内联函数
```

函数调用为：

```
int c=add(3,5);
```

则程序编译后，实际的代码如下。

```
int c=3+5;
```

即在编译后用 add 函数体的代码（“x+y”的返回值“return(x+y);”）代替“add(3,5)”，同时用实参代替形参。这样，代码“int c=add(3,5);”就被置换成“int c=3 + 5;”。

注意：可以在声明函数和定义函数的同时加上 inline，也可以只在其中一处加上 inline，效果相同，都能按内联函数处理。

【例 4-11】　理解内联函数。

```
#include <iostream>
using namespace std;
inline int min(int ,int ,int);  //声明 min 为内联函数,注意左端有关键字 inline
int main()
{
    int a,b,c,m;
    cout <<"please input three integers:" <<endl;
    cin >>a >>b >>c;
    m=min(a,b,c);              //调用内联函数并将返回值赋给变量 m
    cout <<"min=" <<m <<endl;
```

```
    return 0;
}
int min(int x,int y,int z)
{
    if(y<x)                         //求 a,b,c 中的最小值
        x=y;
    if(z<x)
        x=z;
    return x;
}
```

程序运行结果如图 4-17 所示。

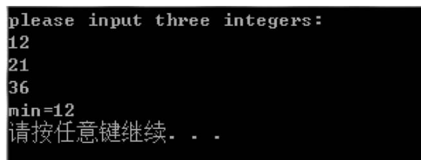

图 4-17 【例 4-3】运行结果

【例 4-11】中源程序的第 3 行，在声明 min 函数时，使用了关键字 inline，指定 min 函数为内联函数。当程序编译执行到第 9 行函数调用 min(a,b,c)时，编译系统就将“m＝min(a,b,c);”替换成下面的代码。

```
if(b<a)
    a=b;
if(c<a)
    a=c;
m=a;
```

使用内联函数如此方便，那可不可以将所有函数都声明为内联函数呢？这显然是不行的。首先，只有规模较小而又被频繁调用的简单函数，才适合声明为内联函数。也就是说，内联函数内不允许出现循环语句、switch 语句及复杂嵌套的 if 语句。其次，对函数作 inline 声明，只是程序设计者对编译系统提出的一个建议，编译器是否作为内联函数来处理由编译器自行决定。

4.4 函数的重载

在 C++程序中允许用同一函数名定义多个函数，这些函数的参数个数和参数类型不同，这就是函数的重载。所谓重载，其实就是“一名多用”，即令一个函数名可以对应多个函数的实现。例如：

```
int function(int);
int function(float);
int function(int,float);
```

虽然上述 3 个函数的名字都是 function，但参数列表不同，所以是重载函数，可以共存于一个程序中。在调用时，只需传入不同的实参，就能调用所需的目标函数，实现不同的功能。

说明：编译系统在判断两个函数是否相同时，是结合函数名和函数的参数列表共同判定的。而函数的返回值类型不能作为区分两个函数的判定依据。

4.4.1　使用重载函数的必要性

其实前面我们早在编写一些算术表达式的程序时，就接触到了重载函数。例如，表达式“2＋4”调用了针对整型操作数的加法操作符，而表达式“2.0＋4.0”调用了另外一个处理浮点操作数的不同的加法操作符。在整个过程中，程序员录入相应的参数后，编译器会根据操作数的类型来区分不同的操作，并应用适当的操作。

表达式“2＋4”和“2.0＋4.0”就应用了重载的加号运算符。类似地，我们可以将运算符的重载推广到的重载函数，这样就能省去为函数起名并记住函数名字的麻烦，从而使程序变得简单，更有助于解决问题。如果不使用重载，那么对于整数的加法和浮点数的加法就要采取两种不同的运算符。整数仍然用“＋”，而浮点数或许可以用“！”表示加法运算，这样做不仅使程序变得很复杂，也无助于解决问题。因此，只有重载才是解决问题的办法。

4.4.2　使用重载函数的时机

定义一组函数，如果目的相似，而参数列表（参数类型或数量）不同，就应当使用函数重载。例如，希望从 3 个数中找出其中的最小值，而每次求最小值时传入的参数不同，程序设计者就会想到图 4-18 所示的两种设计方法。

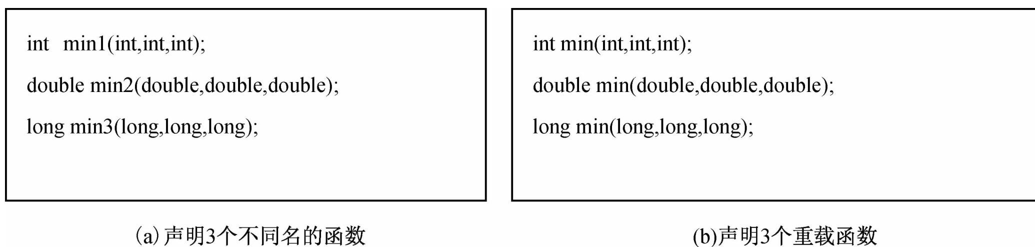

图 4-18　两种设计方法

相信大家都比较倾向于图 4-18(b)所示的处理方法。这样不仅避免了不必要的函数命名（和名字记忆），同时也使程序得到了一定的简化，可读性较高。

注意：在处理一些实际问题时，不要过度使用函数重载。在一些情况下，使用不同的函数名能提供较多的信息，使程序更易于理解。

考虑图 4-19 所示的用于在直角坐标系中移动坐标的两组函数。

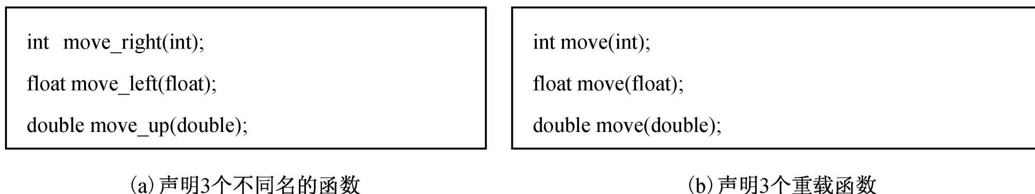

图 4-19　移动坐标的两组函数

显然图 4-19(a)优越于图 4-19(b)，但图 4-19(a)中的函数名能提供移动的方向，而函数重载后会失去原来函数名所包含的信息，如此一来，程序就变得晦涩难懂了。

4.4.3 函数重载解析

当一个函数被重载后，在调用时选择函数的过程就是重载解析。函数重载解析的过程分 3 步进行。

(1)确定实参列表的属性，确定候选函数的集合。

注意：候选函数应与被调用函数同名，并且在调用点上，它的声明可见。

(2)根据实参的类型和个数确定合适的函数(即可行函数)。

(3)选择精确匹配的函数(如果有的话)。

例如：

```
void fun();                        //函数 1
void fun(int);                     //函数 2
void fun(int,int);                 //函数 3
void fun(double,double=2.34);      //函数 4
int main()
{
    …
    fun(1.23);
    return 0;
}
```

按上述步骤，其函数重载解析过程如下。

(1)确定该调用所考虑的候选函数。在这个例子中，有 4 个名为 fun 的候选函数。

(2)根据实参属性从候选函数中选择一个或多个合适的函数，该合适的函数又称为可行函数。可行函数必须满足两个条件：一是函数的形参个数与该调用的实参列表中的参数数目相同；二是每一个实参的类型必须与对应形参的类型匹配，或者它们之间存在隐式转换。

注意：如果函数具有默认实参(参见 4.3.5)，则调用该函数时，所用的实参可能比实际需要的少。

对于函数调用 fun(1.23)有两个候选函数：fun(int)和 fun(double,double=2.34)。

一方面，fun(int)只有一个参数，而且存在从实参类型 double 到形参类型 int 之间的隐式转换。此外，fun(double,double=2.34)虽然有两个参数，但第二个参数给出了默认值，而第一个形参类型是 double，与实参类型精确匹配。

另一方面，fun()和 fun(int,int)这两个函数对于该调用显然是不行的。例子中的调用只有一个实参，而这两个函数分别带有 0 个和 2 个形参。

说明：如果函数重载解析过程的步骤(2)没有找到可行函数，则该调用错误。

(3)选择匹配最佳的可行函数。这个过程应考虑函数调用中的每一个实参，选择对应形参与之最匹配的一个可行函数。最匹配的函数应满足以下条件。

①其每个实参的匹配都不劣于其他可行函数需要的匹配。

②至少有一个实参的匹配优于其他可行函数提供的匹配。

当考虑可行函数 fun(int)时,编译系统需将 double 型的实参转换成 int 型,而这个转换是标准转换。当考虑可行函数 fun(double,double=2.34)时,因可行函数含有默认实参,根据默认实参的参数传递可知,实参与形参是精确匹配的,所以该调用的最佳可行函数是 fun(double,double=2.34)。

说明:如果经过函数重载解析的步骤(3)没有找到最佳匹配的函数,则函数调用是有二义性的。例如,针对上例的调用 fun(2)就存在二义性。

【例 4-12】 求几个操作数之和。

```
#include <iostream>
using namespace std;
int add(int,int);                          //声明两个整数相加的函数
double add(double,double);                 //声明两个双精度数相加的函数
int add(int,int,int);                      //声明 3 个整数相加的函数
int main()
{
    cout <<" 3+5=" <<add(3,5) <<endl;             //调用 add(int,int)函数
    cout <<" 2.0+4.5=" <<add(2.0,4.5) <<endl;/* 调用 add(double,double)
                                                  函数 */
    cout <<" 6+8+10=" <<add(6,8,10) <<endl; /* 调用 add(int,int,int)
                                                  函数 */
    return 0;
}
int add(int x,int y)
{
    return(x+y);
}
double add(double x,double y)
{
    return(x+y);
}
int add(int x,int y,int z)
{
    return(x+y+z);
}
```

程序运行结果如图 4-20 所示。

```
3+5=8
2.0+4.5=6.5
6+8+10=24
请按任意键继续. . .
```

图 4-20 【例 4-12】运行结果

4.5 变量的属性和声明

变量是函数的主要内容，到目前为止，我们所见到的程序中的变量名都是用标识符表示的。一个变量除了具有前面介绍的数据类型(参见2.1节)的属性外，还有作用域、存储期和存储类别3种属性。

这3种属性是有联系的，用户只能声明变量的存储类别，而通过存储类别就可以确定变量的作用域和存储期。后面将会详细介绍。

4.5.1 变量的作用域

每一个变量都有其有效作用范围，这就是变量的作用域。从作用域来分，变量可以分为局部变量和全局变量。

1)局部变量

局部变量是在定义一个函数时，在函数体中声明的变量。它的作用范围仅限于本函数体内，在此函数以外不能访问。例如：

```
void f1()
{
    int a=1;                    //局部变量
}                               //f1 函数结束
int main()                      //主函数
{
    cout <<a <<endl;            //编译错误，标识符 a 没有定义
    return 0;
}
```

上述代码中，虽然在f1函数内定义了一个变量a，但是其作用域仅限于f1函数的函数体内。当在main函数中使用变量a时，由于a是局部变量，编译系统会报错。

同样，在复合语句中声明的变量只在本复合语句范围内有效。例如：

```
void f2()
{
    while(1)
    {
        int i=2;
    }
    cout <<i <<endl;            //错误，标识符 i 的作用域仅限于 while 语句内
}
```

由于局部变量作用域的限制，所以可以在不同函数中使用同名的变量，它们在内存中占

用不同的单元，互不干扰。

注意：形参的作用域也仅限于本函数体内，形参也是局部变量。

2)全局变量

与局部变量相反，全局变量是在函数外部声明的变量，所以又称为外部变量。全局变量的作用域是整个程序，即从声明变量的位置开始到本程序结束，在任何一个函数中都可以访问。例如：

```
int a=0;                        //a 为全局变量
void f1()
{
    cout <<a <<endl;            //在 f1 函数中使用全局变量 a，正确
}
…
int b=1;                        //b 为全局变量
void main()
{
    cout <<a <<b <<endl;  //在 main 函数中使用全局变量 a、b，正确
}
```

上述代码中的 a 和 b 都是全局变量。因为全局变量的作用范围是从变量声明的位置到程序结束，所以可以在 f1 函数中使用 a，可以在 main 函数中使用 a 和 b，但不可以在 f1 函数中使用 b。

注意：全局变量自声明起，在程序中的任何位置都可修改，但全局变量一旦被修改，在其他函数中访问到的值也会随之改变。

局部变量没有默认值，而全局变量的默认值为 0。如果在定义局部变量时没有给出初始化的值，则其值是不确定的；但如果在定义全局变量时没有给出初始化的值，则系统会自动将全局变量初始化为 0。例如：

```
int a;                  //a 为全局变量
void main()
{
    int b;              //b 为局部变量
    cout <<a <<b <<endl;
}
```

虽然在定义全局变量 a 和局部变量 b 时都没有初始化，但程序最终将输出 0 和一个随机值。

注意：在同一个程序中，如果全局变量与局部变量同名，则在局部变量的作用域内，全局变量被屏蔽，只有局部变量可见。

使用全局变量具有以下优势。

(1)使数据可以共享，增加函数间数据联系的渠道。

(2)可以用全局变量作为函数的返回值来保存函数的结果。例如：

```
int a=0;                    //用于保存计算结果的全局变量
```

```
void add(int x,int y)         //定义 add 函数
{
    a=x+y;                     //将求和的结果保存到全局变量 a 中
}
void main()
{
    add(3,5);
    int result=a;              //将全局变量保存的值赋给 result
}
```

使用全部变量的缺点如下。

(1)全局变量在程序的整个执行过程中都占用存储单元。

(2)由于全局变量可以让数据在函数间共享,所以一个函数的逻辑依赖于其他函数的逻辑,函数间的耦合性较高,导致程序复杂化。

(3)全局变量的使用降低了函数的通用性。一般要求把程序中的函数做成一个封闭体,即只依靠“实参—形参”的渠道与外界联系。全局变量的使用会打破这种封闭。

综上所述,不在必要时一般不设全局变量。

4.5.2　变量的存储期和存储类别

1)变量的存储期

变量还有另外一个属性——存储期,即变量在内存中的存在时间。存储期可分为静态存储期和动态存储期。其中,静态存储期是由变量的静态存储方式决定的;动态存储期是由变量的动态存储方式决定的。

顾名思义,静态存储方式就是指在程序运行期间,系统对变量分配固定的、不变的存储空间,例如,全局变量就是采用的这种存储方式。而动态存储方式则是在程序运行期间,系统对变量动态地分配存储空间,系统对函数的形参就是这样处理的。

静态存储区、动态存储区和程序区一起构成了程序在内存中占用的存储空间,如图 4-21 所示。

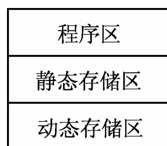

图 4-21　程序在内存中占用的存储空间

程序区用来存放可执行程序的代码。变量一般存放在静态存储区和动态存储区中。分配在静态存储区中的变量为静态变量,如全局变量,在程序开始时给它们分配存储单元,直至程序执行完毕才释放这些空间,且在程序的整个执行过程中占据固定的存储单元。分配在动态存储区中的变量为动态变量,如函数的形参,在函数调用时分配动态存储空间,函数结束时释放这些空间,这种分配和释放是动态的。

2)变量的存储类别

变量的存储类别是指变量在内存中的存储方法。存储方法分为静态存储和动态存储两大类,具体又包含自动(auto)、静态(static)、寄存器(register)和外部(extern)4 种。

在程序中,只有函数的局部变量可以被声明为自动存储类。函数的局部变量和参数一

般属于自动存储类。自动存储类的变量简称为自动变量。自动变量用关键字 auto 作为存储类别的声明。例如：

```
void f(int a)                    //定义函数 f，函数有一个整型形参 a
{
    auto int b=0;                //定义 b 为整型的自动变量
}
```

a 为形参，b 定义为自动变量。它们在内存中都存放在动态存储区，在调用该函数时，系统分别给 a 和 b 动态分配存储空间，函数结束时自动释放这些空间。

声明或定义时，存储类别 auto 和数据类型 int 的顺序任意。关键字 auto 可以省略，如果不写 auto，则系统默认为自动存储类别，它属于动态存储方式。

```
auto int a;
int a;
```

两种声明方式的作用相同。

4.5.3　变量的声明

在函数中声明的变量根据实际情况的需要分为不同的类型。合理地声明函数中的变量才能使数据安全地运算和传输。

1)声明静态局部变量

如果希望函数中局部变量的值在函数调用结束后不消失而保留此原值，即占用的存储单元不被释放，以便下一次调用该函数时，该变量已有值(该值也就是上一次函数调用结束时的保留值)可以声明该局部变量为静态局部变量。声明静态局部变量的格式如下。

```
void f()
{
    static int a;            //定义函数内的局部变量 a 为静态局部变量
}
```

静态局部变量属于静态存储类别，存放在静态存储区内，其生命周期与全局变量一样，在程序的整个运行期间始终存在。

【例 4-13】　观察静态局部变量的值。

```
#include <iostream>
using namespace std;
void f();                        //函数声明
int main()
{
    for(int i=1;i<=3;i++)
    {
        f();                     //调用 f 函数
    }
    return 0;
```

```
}
void f()                    //函数定义
{
    static int a=1;         //定义 a 为静态局部变量
    cout <<"第" <<a-1 <<"次调用 f 函数：a=" <<a++ <<endl;
}
```

程序运行结果如图 4-22 所示。

```
第1次调用f函数: a=1
第2次调用f函数: a=2
第3次调用f函数: a=3
请按任意键继续. . .
```

图 4-22　【例 4-13】运行结果

程序运行后输出的结果是“1，2，3”，而不是“1，1，1”。a 的初值为 1，由于 a 是静态局部变量，在函数调用结束后，系统并不释放变量的存储空间，仍保留上一次调用结束时的值。当第二次调用该函数时，第一次调用结束时保留的值将作为第二次调用时变量的初始值。

静态局部变量的性质与全局变量相似，两者的有效范围都是从声明变量的位置开始到本程序结束，但是任何函数都可以访问全局变量，而静态局部变量只有本函数能够访问，后者更利于控制。例如，对于一些严格的安全系统常常会限制用户登录的次数。这时，可以用静态局部变量来记录用户登录的次数，超过限定次数后就阻止用户继续尝试，从而达到安全保护的效果。

【例 4-14】　用静态局部变量记录用户登录系统的次数。

```
#include <iostream>
using namespace std;
void login();                   //声明登录函数 login
void main()
{
    for(int i=0;i<=3;i++)
    {
        login();                //调用登录函数
    }
}
void login()                    //定义登录函数
{
    static int time=0;          //记录登录次数的静态局部变量
    time++;
    if(time<=3)
        cout <<"登录" <<time <<"次 ：" <<"登录成功。" <<endl;
    else
        cout <<"登录次数已超过 3 次，24 小时内不允许再登录！" <<endl;
}
```

程序运行结果如图 4-23 所示。

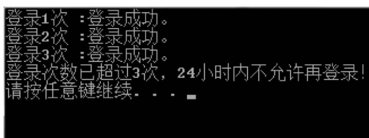

图 4-23　【例 4-14】运行结果

对静态局部变量的几点说明如下。

(1)静态局部变量的初始化仅在程序开始执行时处理一次，在整个程序运行期间它的值始终存在。

(2)静态数值型局部变量的默认初值为 0；静态字符型局部变量的默认初值为空字符'\0'。

(3)虽然静态局部变量在函数调用结束后仍然存在，但其他函数不能使用它。

2)声明寄存器变量

通常，变量的值存放在内存中。当程序用到一个变量的值时，由控制器发出指令将内存中该变量的值送到 CPU 中的运算器进行计算，如果需要保存数据，再从运算器将数据送到内存存放，其过程示意图如图 4-24 所示。

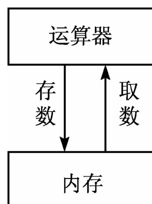

图 4-24　数据存取示意图

如果某个变量使用频繁(如某个 for 循环要执行 5 000 次，每次循环都要使用某个局部变量)，为了节省时间，可以将该变量存放到 CPU 的寄存器中，因为寄存器的存取速度远高于内存的存取速度，这样可以提高执行效率。存放到寄存器中的变量称为寄存器变量，用关键字 register 声明。例如：

```
int function()
{
    register int f=1;      //定义 f 为寄存器变量
    …
    return 0;
}
```

注意：在程序中定义寄存器变量对编译系统只是建议性的，系统会根据具体情况决定是否这样处理。

实际上已经没有必要用 register 声明变量了，因为现今的编译系统能够识别出使用频繁的变量，从而自动将这些变量存放在寄存器中，而不需要用户指定。

3)声明全局变量

可以用关键字 extern 扩展全局变量的作用域。用 extern 声明全局变量有以下两种情况。

(1)在一个源文件的程序中声明全局变量。从 4.5.1 中已经知道，全局变量的作用域是从声明变量的位置开始到本程序的结束。如果某些全局变量没有在文件的开头定义，就在其定义点之前引用这些全局变量，就需要用 extern 声明这些外部变量，表明这些变量是将在下面定义的全局变量。这种声明称为提前引用声明。例如：

```
int a=1;                    //定义全局变量 a
```

```
…
void main()
{
    extern int b;          //对全局变量b进行提前引用声明
    cout <<a <<endl;//输出a
    cout <<b <<endl;//输出b
}
int b=2;                   //定义全局变量b
…
```

在main函数的前面定义了全局变量a,在main函数的后面定义了全局变量b,所以可以在main函数中输出b,但如果想要输出b,则必须对变量b作提前引用声明,否则系统将会报错。一般做法是将全局变量的定义放在引用它的所有函数之前,以避免在函数中再加一个extern声明。

(2)在多个源文件的程序中声明外部变量。当程序由多个源文件组成,而其中某个源文件的外部变量需被其他源文件共享时,在相应源文件的函数使用前还需对该外部变量作如下声明。

```
extern 数据类型 全局变量名;
```

声明前加上关键字extern表明这个全局变量是在其他文件中定义的,但需要在本文件中使用。

例如,现有两个源文件file1.cpp和file2.cpp,其中源文件file1.cpp中已经定义了一些全局变量,如果源文件file2中的函数想要使用这些全局变量,则必须在file2.cpp中的函数使用前声明这些全局变量。

在file1.cpp中定义如下全局变量。

```
/*file1.cpp     源文件1     */
int a=1;
double b=2.0;
```

当在file2.cpp中使用上面两个变量时,必须作如下声明。

```
/*file2.cpp     源文件2     */
extern int a=1;
extern double b=2.0;
```

说明:在全局变量前加上关键字extern只是用来声明全局变量,即告知编译系统该变量是全局变量,而不是定义全局变量。应注意,extern声明要在全局变量定义后,否则编译器会报告"某某全局变量没有定义"的错误。

4)声明静态外部变量

在程序设计过程中如果希望某些外部变量仅限于本文件引用,而不能被其他文件引用,就可以在定义外部变量时加static说明。例如,在file1.cpp中定义静态外部变量a。

```
//file1.cpp     源文件1
static int a=1;     //用static声明静态变量
…
```

在 file2.cpp 中使用变量 a 就会出错。

```
//file2.cpp      源文件 2
extern int a=1;    //用 extern 声明外部变量
…
void f(int b)
{
    a=a+b; /* 错误,a 已在 file1.cpp 中声明为静态外部变量,不能在 file2.cpp
              中引用 */
}
```

可以看到,在 file1.cpp 中用 static 声明了静态外部变量 a 后,a 的作用域就限定在 file1.cpp 中了,再在 file2.cpp 中引用就会出错。

注意:用 extern 关键字声明外部变量后,改变的只是变量的作用域,而变量的存储方式依然不变,即在加关键字前后,该外部变量都采用静态存储方式。

4.6　内部函数和外部函数

一个 C++ 程序可由多个源程序文件组成,根据函数能否被其他源文件中的函数调用,可将函数分为内部函数和外部函数。

4.6.1　内部函数

如果一个函数只能被本文件中的其他函数调用,则称为内部函数(或静态函数)。在定义内部函数时,在函数类型的前面加上 static 即可,格式如下。

static <数据类型>[函数名]<(形参列表)>

例如:

```
static int function(int a,int b)
```

编写大型程序时,为方便程序的设计与调试,往往将一个程序分成若干模块,再由若干人分别完成各个模块。使用内部函数,可以使函数局限于所在文件,即使在不同的文件中有同名的内部函数,也互不干扰。这样,不同的程序员在完成各自的模块时,可以编写不同的函数,而不必担心所使用的函数是否会与其他文件中的函数同名。通常把只能由同一文件使用的函数和全局变量放在一个文件中,在它们的前面加上 static 声明使之局部化,使其他文件不能引用。

4.6.2　外部函数

在一个源文件中定义的函数不仅能在本文件中使用,而且可以在其他文件中共享,这样的函数称为外部函数。同样,外部函数的声明是在进行函数原型声明(或函数定义)时,在函

数类型前加关键字 extern，格式为：

extern <数据类型>［函数名］<(形参列表)>

例如：

```
extern int function(int a,int b)
```

C++语言规定，默认情况下，所有函数都是外部函数。在函数类型前加关键字 extern 修饰，只是为了强调本源文件中调用的函数是在其他源文件中定义的(或者本源文件中定义的函数可以被其他源文件中的函数调用)。例如，现有源文件 file1.cpp 和 file2.cpp，在 file1.cpp 中定义了一个函数 f，要想在 file2.cpp 中调用它，就可以作 extern 声明，具体如下。

在 file1.cpp 中定义函数 f。

```
/* file1.cpp     源文件 1    */
int f(int x,int y)        //定义函数 f
{
    return x*y;
}
…
```

在 file2.cpp 中调用函数 f。

```
/* file2.cpp     源文件 2    */
extern int f(int,int);         //调用前用 extern 声明外部函数
…
z=f(x,y);                      //调用函数 f
```

当然也可以省略关键字 extern，而声明为“int f(int,int);”，效果相同。

说明：由于函数在默认情况下都是外部函数，为方便编程，C++允许在声明函数时省略 extern，而采用函数原型声明。

【例 4-15】 用如下公式计算排列函数，要求用外部函数实现。

$$p(n,k)=\frac{n!}{(n-k)!}$$

解题方法：在源文件 f1.cpp 中定义求阶乘的函数，在源文件 f2.cpp 中调用求阶乘的函数。

```
/* f1.cpp      源文件 1 */
#include <iostream>
using namespace std;
int factorial(int n)           //定义求阶乘函数
{
    if(n<0)                    //n<0 时
    {
        cout <<"n<0,错误!" <<endl;
        return 0;
    }
    else                       //n>0 时
```

```
        {
            int m=1;
            while(n>1)
            {
                m*=n--;
            }
            return m;
        }
    }
    ///////////////////////////////////////////
    /* f2.cpp        源文件 2 */
    #include<iostream>
    using namespace std;
    extern int factorial(int);          //调用前用 extern 声明外部函数
    void main()
    {
        int n,k,f;
        cout<<"请输入 n 的值:"<<endl;
        cin>>n;
        cout<<"请输入 k 的值(k<=n):"<<endl;
        cin>>k;
        f=factorial(n)/factorial(n-k) ;
        cout<<"p("<<n<<","<<k<<")="<<f<<endl;
    }
```

程序运行结果如图 4-25 所示。

```
请输入n的值:
6
请输入k的值（k<=n）:
2
p(6, 2)=30
请按任意键继续. . .
```

图 4-25　【例 4-15】运行结果

在计算机上运行含多个文件的程序时，需要建立一个项目文件，再在该项目文件中包含程序的各个文件。在【例 4-15】中，可先创建 f1. cpp 项目文件，再添加 f2. cpp 项目文件。

4.7　预处理命令

为改进程序设计环境，提高编程效率，C++ 语言允许在程序中用“预处理命令”编写一些命令行。这些预处理命令不是 C++ 语言本身的组成部分，不能直接对它们进行编译(因为编译程序无法识别)。必须在对程序进行通常的编译之前，先对这些特殊的命令进行“预处理”。

预处理是在编译之前对程序内容进行最后的取舍处理，使一些语句参加编译，而另一些

语句不参加编译。预处理行都以“＃”开头，一般写在程序的首部。

C++提供的预处理功能主要有宏定义、文件包含和条件编译 3 种，分别用宏定义命令、文件包含命令和条件编译命令来实现。这里只讨论前两种，第三种条件编译将不作讲解。

注意：预处理命令不是 C++语句，末尾不加分号。

4.7.1 宏定义

宏定义由宏定义命令来实现，宏定义命令用＃define 表示。根据宏名是否带有参数，可将宏定义分为不带参数的宏定义和带参数的宏定义。

1）不带参数的宏定义

不带参数的宏定义就是用一个短的名字（宏名）代表一个长的字符串，它的一般形式为：

```
#define 宏名 宏体
```

其中，宏名是合法的 C++标识符，宏体是一个字符串，不带参数的宏定义其实就是定义符号常量。例如：

```
#define PI 3.1415926
```

它的作用是指定用宏名 PI 来代替字符串 3.1415926，在编译预处理时，将程序中在该命令以后出现的所有的 PI 都代换成 3.1415926。在预编译时将宏名代换成字符串的过程称为“宏代换”。

在书写不带参数的宏定义时，需要注意以下一些事项。

（1）宏定义用宏名代替一个字符串，只作简单的代换，不进行其他任何操作，当然也不进行正确性检查。定义中如果加了分号，则会连分号一起进行代换。例如：

```
#define PI 3.1415926;        //错误，应去掉“;”
area=PI*r*r;
```

经过宏代换后，该语句为：

```
area=3.1415926;*r*r;
```

显然已出现语法错误。

（2）一般习惯用大写字母表示宏名，以便与变量名相区别。

（3）＃define 命令的有效范围为：从定义处开始到包含它的源文件结束。通常将＃define 命令写在文件开头、函数之前。

（4）可以用＃undef 命令终止宏定义的作用域。例如：

```
#define PI 3.1415926
…
void main()
{
    …
}
#undef  PI
…
```

由于#undef 的作用,PI 的作用范围在#undef 行处终止。在#undef 行之后,PI 不再代表 3.1415926。这样可以灵活控制宏定义的作用范围。

2)带参数的宏定义

用#define 命令定义带参数的宏的一般形式为:

#define 宏名(参数表) 宏体

其中,宏名是一个标识符,参数表中可以有一个或多个参数,多个参数之间用逗号分隔。宏体是被代换用的字符序列。例如:

```
#define S(x,y) x*y        //定义宏 S(矩形面积),x、y 为宏的参数
```

如果在程序中出现如下语句:

```
area=S(2,4);
```

则用 2、4 分别代换宏定义中的形参 x、y,即用 2*4 代替 S(2,4)。代换后的实际语句为:

```
area=2*4;
```

可以看出,在代换时,首先进行参数代换,然后再将代换后的字符串进行宏代换。

使用带参数的宏定义的几点说明如下。

(1)带参数的宏定义的字符串应写在一行上,如果需要写在多行上时,应使用续行符(\)。例如:

```
#define MISS "I miss \
my hometown"
```

等价于:

```
#define MISS "I miss my hometown"
```

(2)在书写带参数的宏定义时,宏名与参数列表的左括号之间不能出现空格,否则空格右边的字符都作为替代字符串的一部分。例如:

```
#define S (x,y) x*y
```

这时,宏名 S 将被认为是不带参数的宏定义,它代表字符串“(x,y) x*y”。

使用宏定义虽然可以用宏名代表一个字符串而减少程序中的重复书写,但是 C++ 中可以用 const 语句定义常变量(参见 2.2.5),这比不带参数的宏定义更加优越。此外,C++ 中新增加的内联函数也比用带参数的宏定义更方便,因此#define 除主要用于条件编译以外,在实际中已不多用。

4.7.2 文件包含

文件包含的作用是让编译预处理器把另一个源文件嵌入(包含)当前源文件中的该预处理命令处。经文件包含处理后,一个源文件就把另外的源文件包含到本文件之中。

图 4-26 形象地说明了“文件包含”的含义。如图 4-26 所示,有 file1.cpp 和 file2.h 两个文件,在 file1.cpp 中有一个“#include "file2.h"”命令,以及其他内容(用 A 表示)。文件 file2.h 中的内容用 B 表示。在编译预处理时,系统先对#include 命令进行文件包含处理:将 file2.h 的全部内容复制到“#include "file2.h"”命令处,即把 file2.h 包含到 file1.cpp 中,在随后的编译中,就将包含以后的 file.cpp 作为一个源文件进行编译。

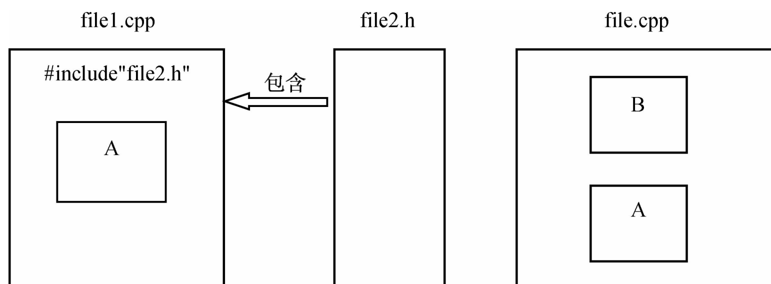

图 4-26　文件包含的含义演示

文件包含命令格式如下。

```
#include <文件名>
```

或

```
#include "文件名"
```

前者的头文件是 C++ 系统提供的，文件名放在一对尖括号内；后者是程序员自定义的头文件，文件名放在一对双引号中。例如：

```
#include <iostream>
```

或

```
#include "file.h"
```

这两种格式在 C++ 中都是认可的。两者的区别是：用尖括号（如<iostream>形式）时，当 C++ 预处理器遇到这条命令后，就自动到该 include 目录中搜寻要包含的文件，并把它嵌入当前文件中，这种搜寻方式称为标准方式。用双引号（如"file. h"形式）时，系统首先在文件所在当前工作目录中搜寻，如果找不到，再按标准方式搜寻。如果文件在用户当前工作目录中，用前一种方式时，系统直接到标准头文件包含目录中去搜寻，因此，这种方式适用于包含系统提供的头文件。用后一种方式时，要先在用户当前工作目录中搜寻，搜寻不到再到系统目录中搜寻，从效率上看，它适用于包含用户建立的头文件。

使用头文件包含时应注意以下几点。

(1)一条文件包含命令只能包含一个文件，若要包含多个文件须用多条包含命令。例如：

```
#include <iostream>
#include <iomanip>
#include <cmath>
…
```

(2)被包含文件还可以使用文件包含命令，即文件包含命令可以嵌套使用。例如，头文件 file1. h 的内容如下。

```
#include "file2.h"
#include "file3.h"
…
```

file2. h 文件内容如下。

```
#include "file3.h"
…
```

需要说明的是，新的 C++ 标准库中的头文件一般不再包括后缀.h，例如：

#include <string>

但为了使大批已有的 C 程序能继续使用，许多 C++ 编译系统保留了 C 的头文件，即提供两种不同的头文件，由程序设计者选用。

习　　题

(1)编写程序，用一个函数判定输入的某个数是否为素数。

(2)从键盘依次输入一列整数(输入 0 结束)，编写函数，统计输入的整数序列中的奇数个数和偶数个数，在主函数中调用该函数并求输入的整数序列中的奇数个数和偶数个数。

(3)定义一个带默认参数值的函数求 $n!$，在主函数中调用两次该函数，一次给出实参，另一次不给实参，体会默认值的意义。

(4)求 $a!+b!+c!$ 的值，要求用两个函数实现：函数 fac 求 $n!$，函数 add 求三者之和。a、b、c 的值由主函数输入，最终得到的值在主函数中输出。

(5)编写一个程序，判定用户输入的年份是否为闰年。闰年的判定用名为 is Leap Year 的 bool 型函数实现。

(6)用递归函数求斐波那契数列中的第 $n(n=1,2,3,\cdots)$ 个数。斐波那契数列的特点是：第 1、2 个数为 1、1，从第 3 个数开始，每个数是其前两个数之和。即

$F_1=1$　　　　　　$(n=1)$

$F_2=1$　　　　　　$(n=2)$

$F_n=F_{n-1}+F_{n-2}$　　　　$(n\geqslant 3)$

(7)写一个函数验证哥德巴赫猜想：一个不小于 6 的偶数可以分解为两个素数之和。要求：在主函数中输入一个偶数 n，判定 n 与 6 的大小。如果 n 不小于 6，则调用函数 gotbaha，在 gotbaha 函数中再调用判定素数的函数 prime，并在主函数中输出“6=3+3”形式的结果。如果 n 小于 6，则程序结束。

(8)使用 add 作为重载函数名，分别定义求 3 个整数之和的函数和求两个浮点数之和的函数，并编程实现。

(9)在主函数中输入两个数 m，n，用内联函数求和，并输出结果。

(10)编写一个程序，将求两个实数中较大值的函数放在一个头文件中，在源程序文件中包含该头文件，并实现输入 3 个实数，求出最大值。

(11)编写一个程序，输入两个整数，求它们的乘积。用带参数的宏实现。

(12)编写一个递归函数，将整数的每个位上的数字按相反的顺序输出。例如，输入 5678，输出 8765。