C++高级语言程序设计

- 王晨宇

- 北京邮电大学网络空间安全学院

第2章 函数

2.1 函数的定义

- 函数：功能独立的语句块。需先定义后使用。
- 函数的优势：①易于实现 ②便于调用 ③简化程序
  ④支持复杂问题的功能分解和模块化程序设计。
- 函数分为：库函数和自定义函数。
- main函数：系统约定，用户定义，有且仅有一个，程序执行的唯一入口，由操作系统调用。main函数通过调用库函数或自定义函数实现程序的功能。
- 库函数(预定义函数)：C++编译系统、操作系统或其它系统为方便用户程序设计而预定义的函数。使用库函数可简化程序，提高编程效率。
- 用户自定义函数：用户编写、完成特定功能的函数。

函数定义格式

函数返回值的类型，可为任何已定义数据类型。若省略，则默认是int型。若无返回值，则其类型应用void说明。

- 指明每个参数的类型和名称，参数间用逗号分隔。若没有参数，则参数列表可为空或用void说明。即使参数列表为空，其两侧的一对圆括号“( )”也不可省。
- 参数列表是函数调用时数据传递的接口。
- 函数定义时的参数称为形式参数，函数调用时的参数称为实际参数。
- 函数定义时,圆括号后不能加分号“；”。

[<返回值类型>]<函数名>（<参数列表>）//函数头

{

<语句序列>

}

//函数体

函数的名字，应见名知意。

- 函数体：用一对花括号“{ }”括起的块语句。即使函数体内没有语句，花括号“{ }”也不可省。
- 函数体内不能包含其它函数的定义。

函数定义举例

- 有参且有返回值。如：

```
int max(int a,int b){  return (a>b)?a:b;  }
```

- 有参但无返回值。如：

```
void swap(int x,int y){  int t=x;x=y;y=t;  }
```

- 无参但有返回值。如：

```
char getc( )
{   char x;
    cin>>x;
    return x;
}
```

- 若函数有返回值，则必须使用return语句返回其值。

- 无参也无返回值。如：

```
void mess( ){  cout<<"你好，欢迎学习C++!";  }
```

常用运算符种类:

- 算术运算符 + - * / % + --
- 关系运算符 ==、!=、<、>、<=、>=
- 逻辑运算符 !、&&、||
- 条件运算符 e1？e2：e3
- 赋值运算符 =、+=、-=、*=、/=、

... ...

相同目数的各运算符的优先顺序为:

算术、关系、逻辑、条件、赋值

2.2 函数的参数

- 函数的形式参数简称形参。

①当被调函数有参时，主调函数和被调函数之间通过形参实现数据传递。

②函数的形参仅在函数被调用时，才由系统分配内存，用于接收主调函数传递来的实际参数。

- 函数的实际参数简称实参。

①函数调用时实参的类型应与形参的类型一一对应。

②实参应有确定值，可为常量、变量或表达式。

③形参与实参占用不同的内存，即使形参和实参同名也不会混淆。函数调用结束时，形参所占内存即被释放。

2.3 函数的返回值

- 函数的返回值：由被调函数计算处理后向主调函数返回的计算结果，最多只能一个，用return实现。
- 无返回值的函数其返回值类型应说明为void类型，否则将返回一个不确定的值。
- 在执行被调函数时，遇到return语句就结束函数的执行，返回到主调函数。若函数中无return语句，会执行到函数体最后的”}”为止，返回到主调函数。

- return语句的格式：

①return <表达式>;//用于带有返回值的函数

作用：先计算<表达式>的值。若<表达式>的值的类型与该函数的类型不同，则将<表达式>的类型强制转换为该函数的类型。再将<表达式>的值返回给调用函数并将程序的流程由被调用函数转给调用函数。

②return;　　//用于无返回值的函数

作用：将程序的流程由被调用函数转给调用函数。

对于无返回值的函数，若函数没有return语句，则执行完最后一条语句后将返回到调用函数。

- 若函数有多个返回分支，则应保证每个分支均有确定的返回值，否则可能出现逻辑错误。例如：

```
char toLower(char c)
{
  if(c>='A'&&c<='Z')
    return c+'a'-'A';
}
```

现象：编译该函数时将出现一个警告，指出未使每个执行分支均有确定的返回值。

原因：若c是小写字母，则该函数无确定的返回值。

改正：

```
char toLower(char c)
{
  if(c>='A'&&c<='Z')
    return c+'a'-'A';
  return c;
  //或：else return c;
}
```

2.4 函数原型

- 函数后定义先使用是常见现象，如：

①自顶向下、逐步求精的程序设计方法，使程序员习惯将main函数作为程序的第一个函数。这样在main函数中可能调用其后定义的许多函数。

②程序由多个文件组成时，若一个文件中的函数要调用另一个文件中的函数时，也会出现类似问题。

③使用库函数。

- 函数原型的作用：使函数能后定义先使用。
- 函数原型的含义：对定义在后或库中的函数，在使用前做声明，包括函数名、返回类型及参数类型。

- 函数原型的格式:
  <返回类型> <函数名>(<形参列表>);

①函数原型与函数定义在返回类型、函数名和参数类型方面必须一致。

②函数原型是语句，必须以分号“;”结束，而函数定义时的头部之后不能有分号。

- 库函数的原型通常在头文件中声明，在编程时，若要使用某个头文件中的库函数，则必须先将这个头文件包含到程序中。例如:

```
#include<cmath>
   …
double x=sqrt(2.0);
```

- 函数原型举例：

```
#include<iostream>
using namespace std;
int dec(int a,int b); //函数调用前做原型说明
//或：  int dec(int,int);

void main(void)
{
   cout<<"两数之差="<<dec(20,3);//函数先调用
}
int dec(int a,int b)          //函数后定义
{
   return a-b;
}
```

- 函数原型功能:

- 编译器正确处理函数的返回值;
- 编译器检查使用的参数数目是否正确;
- 编译器检查使用的参数类型是否正确。如果不正确,则转换为正确的类型（如果可能的话)

2.5 函数的调用

- 函数定义后，并不能自动执行，必须通过函数调用来实现函数的功能。
- 函数调用，即控制执行某个函数。
- C++中，主函数可以调用其它子函数，而其它函数之间也可以相互调用。
- 在本节中，我们将介绍一下内容：
  - ✓函数调用的格式
  - ✓参数的传递方式
  - ✓函数的递归调用

2.5 函数的调用

- **函数调用的一般格式：**

  <函数名>（<实际参数表>）//有参调用

  或 **<函数名>（）**//无参调用

其中：

- **<函数名>**为要使用的函数的名字。
- **<实际参数表>**是以逗号分隔的实参列表，必须放在一对圆括号中。

  <实参表>与<形参表>中参数的个数、类型和次序应保持一致。
- 当调用无参函数时，函数名后的圆括号不能省略。

2.5 函数的调用

1. 实参的几种形式

- 形参为简单类型变量，对应的实参可以是：常量，变量及表达式。
- 形参为数组，对应的实参为数组（名）。
- 形参为结构类型，对应的实参为结构类型变量。

如：调用已知三边求三角形面积的函数Area。

```
double Area(double,double,double); //函数声明
cout<<Area(4.0,5.0,6.0)<<endl; //常量作实参
cout<<Area(a,b,c)<<endl; //变量作实参
cout<<Area(a+1,b+1,c+2)<<endl; //表达式作实参
```

2.5 函数的调用

2. 函数调用的形式

(1)函数调用作为一个独立的语句（用于无返回值的函数）

调用的形式为:

函数名（实参表）； 或 **函数名（）；**

- 函数调用举例：语句调用。

```
 #include<iostream>
using namespace std;
void mess(char * msg)
{
    cout << msg << endl;
}
int main(void)
{
 char cook[17] = "同学们，你们好！ ";
mess(cook); //语句调用
 return 0;
}
```

程序运行结果：
同学们，你们好!

- 无返回值的函数只能用函数调用语句来调用。
- 在不需要利用返回值时，有返回值的函数也可用函数调用语句来调用。

2.5 函数的调用

2. 函数调用的形式

(2)函数调用出现在表达式中（适于有返回值的函数调用形式）

如： 函数max()求两个数的最大值。函数原型如下：

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

该函数有返回值，调用时应出现在表达式中。

判断以下语句完成的功能：

- ✓ c=max(a,b); //函数调用出现在赋值运算符右边的表达式中
- ✓ d=max(c,max(a,b));//函数调用同时出现在实参表达式中
- ✓ cout<<max(a,b)<<endl;//输出一个函数值

- 函数调用举例：表达式调用。

```
#include<iostream>
using namespace std;
int max(int m,int n)
{
   return m>n?m:n;
}

void main(void)
{  int a, b;
   cout<<"输入两个整数： ";
   cin>>a>>b;
   cout<<"两数之大数： "<<max(a,b)<<'\n';
}
```

程序运行结果：

输入两个整数：156 201↙

两个数中的大数为：201

- 若函数有返回值，则函数调用可出现在表达式中，使函数的返回值参与表达式的运算。

- 函数是独立完成某个功能的模块，函数与函数之间主要通过参数和返回值来联系。
- 函数的参数和返回值是该函数对内、对外联系的窗口，称为接口。
- 从函数调用角度看，可将函数看作是一个“黑盒”，除了接口外，其他不必关心。
- “黑盒”函数的接口用函数原型描述。例如，大量的库函数都是“黑盒”函数。
- 函数调用时，系统为形参分配相应的存储单元，用于接收实参传递的数据。注意：函数调用期间，形参和实参各自拥有独立的存储单元。
- 函数调用结束，系统回收分配给形参的存储单元。

- 函数调用时，实参向形参传递参数的方式有三种：

①值传递，也称传值。

形式：形参为普通变量，实参为表达式，实参向形参赋值。

特点：参数传递后，实参和形参不再有任何联系。

注意：实参是表达式，故形参不可能给实参赋值。

②引用传递，也称传引用。

形式：形参为引用型变量，实参为变量，实参为引用型形参初始化。

特点：参数传递后，形参是实参的别名，彼此关联。

③指针传递，也称传指针。

形式：形参为指针变量，实参为指针表达式。

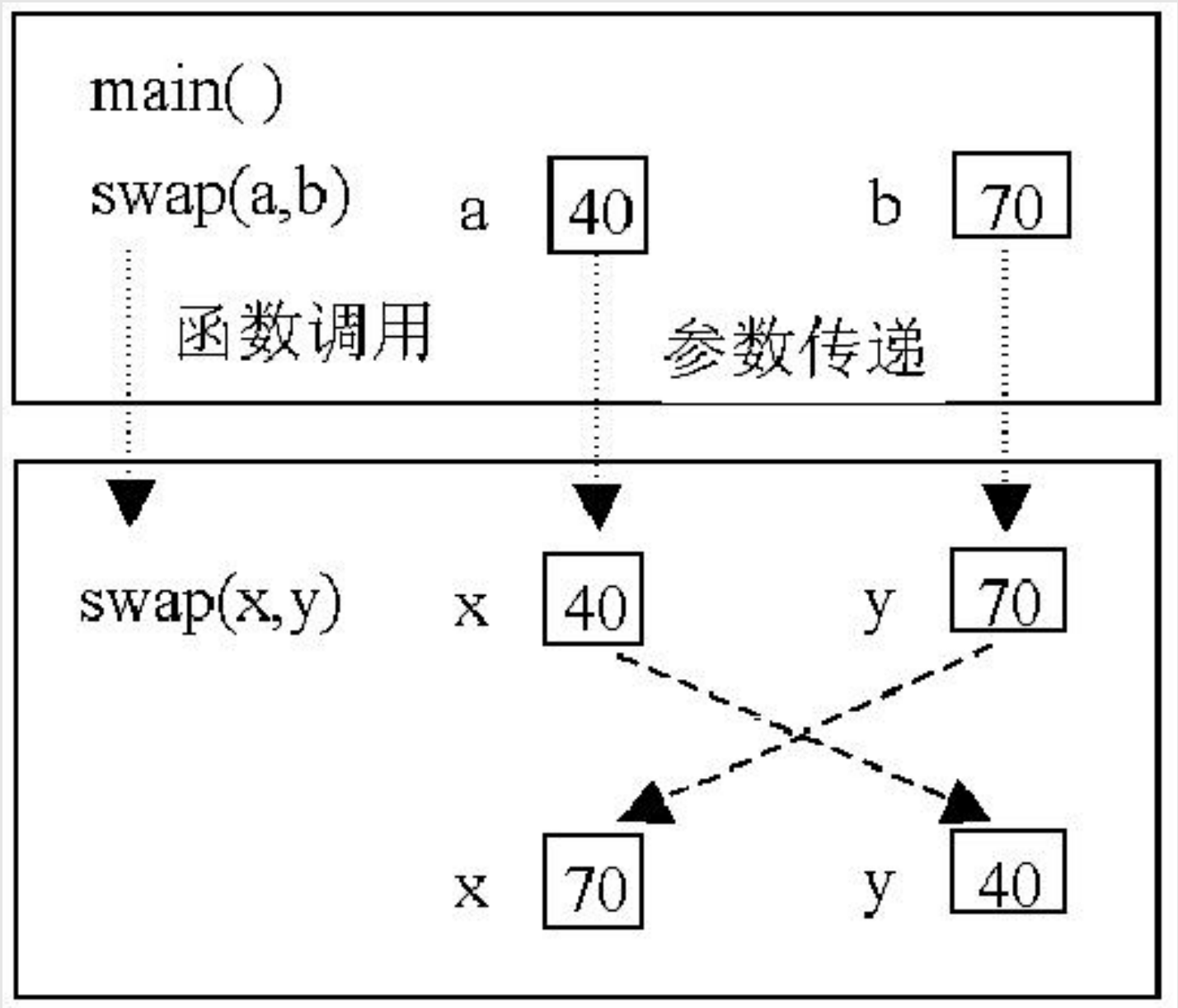
特点：参数传递后，形参可读写实参所指的存储空间。

2.5.1函数的传值调用

- 函数的传值调用分析:

```
#include<iostream>
using namespace std;
void swap(float x,float y)//仅交换形参x和y
{  float t=x;x=y;y=t;
   cout<<"x="<<x<<"\ty="<<y<<'\n';
}
void main(void)
{  float a=40,b=70;
   swap(a,b);              //未交换实参a和b
   cout<<"a="<<a<<"\tb="<<b<<'\n';
}
```

输出结果:
x=70 y=40
a=40 b=70


main( )
swap(a,b)
a
40
b
70
函数调用
参数传递
swap(x,y)
x
40
y
70
x
70
y
40

- 传值调用的优点：函数调用对其外界的变量无影响，最多只能用return返回一个值，函数独立性强。

- 举例：输入一个正整数，判断该数是否为素数。

```
#include<iostream>
#include<cmath>
using namespace std;
int prime(int m)
{  for(int i=2;i<=sqrt(m);i++)
     if(m%i==0) return 0;
   return 1;
}
```

```
void main(void)
{  int m;

   cout<<"请输入一个正整数:  ";
   cin>>m;

   if(prime(m)) cout<<m<<"是素数\n";
   else cout<<m<<"不是素数\n";
}
```

运行结果:

请输入一个数: 23↙

23是素数.

2.5.2函数的引用调用

- 引用：引用是一种特殊的变量，它被认为是一个变量的别名。对别名的访问就是对别名所关联变量的访问，反之亦然。
- 引用定义的格式如下：
  - <数据类型> &<引用名>=<目标变量名>;
- 其中：&为引用（变量）的标志符号，<引用名>是一个标识符。

<数据类型>为<目标变量>的类型

例如：int a,&b=a;

b=15;

- 该例说明了a是一个整型变量，b是一个引用整型变量a的引用，即b是a变量的一个别名。这时，使用a与使用b是等价的。

2.5.2函数的引用调用

- 使用引用应注意：

① 定义引用时，应同时对它初始化，使它与一个类型相同的已有变量关联；

② 一个引用与某变量关联，就不能再与其它变量关联。

③ 引用主要用作函数的形参和返回值。

④ 引用并不分配独立的内存空间，它与目标变量共用其内存空间。

⑤ 引用只能引用变量，不能引用常量和表达式。

- 若引用做函数的形参，则函数调用时，实参用变量名。实参为形参初始化，即声明形参是实参的别名。这样，在被调函数中，对形参的操作就是对实参的操作。
- 用引用调用，使两实参做互换。

```
void swap(float &x, float &y)
{ float t=x;  x=y;  y=t;  }
int main(void)
{ float x=40,y=70;
  cout<<"x="<<x<<"\ty="<<y<<'\n';
  swap(x,y); //独立语句调用
  cout<<"x="<<x<<"\ty="<<y<<'\n';
return 0;
}
```

此时swap函数中的形参x和y分别是main函数中的实参x和y的别名。

程序输出结果：

x=70， y=40

- 下面程序片段的输出结果?

```
int a=1,b=2;
int &r=a;
r=b;
r=7;
cout<<a<<endl;
```

A. 1

B. 2

C. 7

D. 都不是

2.5.3 函数调用过程分析

- 执行程序时，若遇到调用函数f()，则暂停当前函数的执行，保存断点(即返回地址和现场)，然后转去执行f()函数。当遇到return语句或者函数结束时，则恢复先前保存的现场，并从先前保存的返回地址开始继续执行。

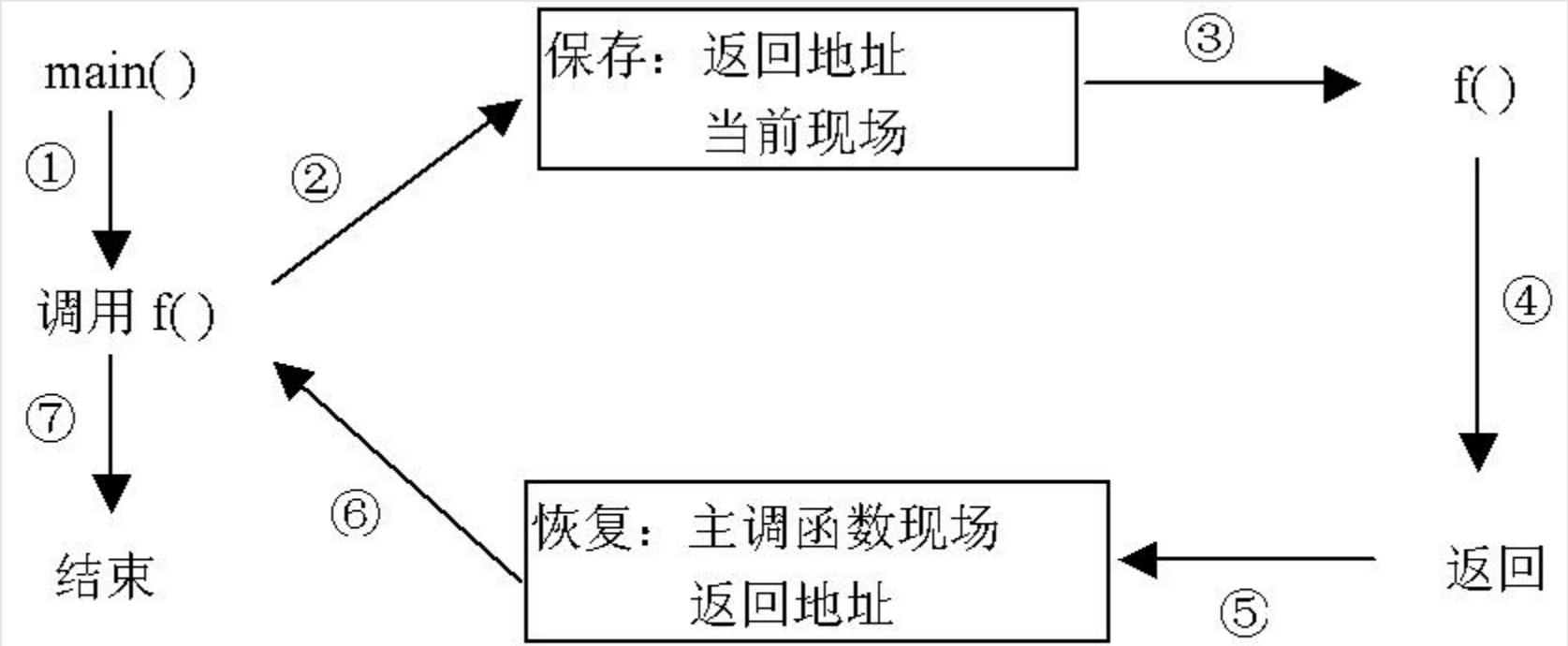


图 6-2 函数调用和返回的示意图

2.5.4函数的嵌套调用

- C++语言不允许在一个函数的定义中再定义另一个函数，即不允许函数的嵌套定义。但允许在一个函数的定义中调用另一个函数，即允许函数的嵌套调用。
- 例2.7 计算$1^k+2^k+3^k+\cdots+n^k$。

```
#include<iostream.h>
int sump(int,int);    //函数原型声明
int powers(int,int);
void main(void)
{  int k=4,n=6;
   cout<<"从1到"<<n<<"的"<<k<<"次幂="
       <<sump(k,n)<<endl;
}
```

```
int powers(int n,int k)//计算n^k
{
   for(int i=1,product=1;i<=k;i++)
      product*=n;
   return product;
}

int sump(int k,int n)  //计算1^k+2^k+3^k+…+n^k
{
   for(int i=1,sum=0;i<=n;i++)
      sum+=powers(i,k);
   return sum;
}
```

程序运行结果:
从1到6的4次幂之和=2275

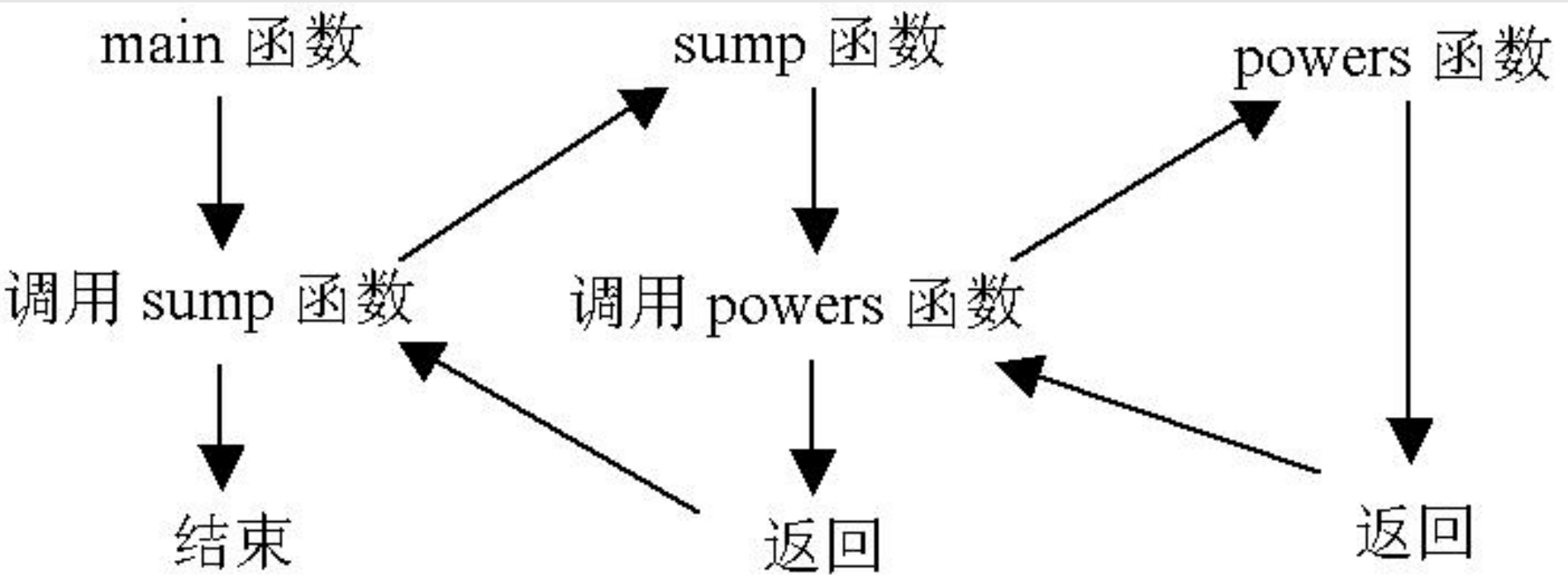


图 **6-3** 例 6.7 函数的嵌套调用

(3) 作业

- 用递归函数实现任意个随机数的累计乘积

2.5.5函数的递归调用

- 函数的递归调用：一个函数直接或间接调用本身。
- 递归举例：求4!。

分析：通常有两种求法。

①递推法：

由已知开始，逐步递推求解，最终求得未知。

由1!=1，求得2!=1!*2=1*2=2；再由2!=2，求得3!=2!*3=2*3=6；最后，由3!=6，求得4!=3!*4=6*4=24。

②递归法：

将n!转换为：

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

直接求4!：由于无法一下子求得结果，把它转换成4*3!，只要能求得3!，4!就能求得。

接着求3!：转换成3*2!。

再求2!：转换成2*1!。

直到变成求1!的问题为止(因1!为1)，递推过程结束。

然后，再逐级回归，依次得到2!=2*1=2，3!=3*2=6和4!=4*6=24。这种方法包括递推和回归两个阶段，递归结束条件是结束递推、开始回归的转折点。

- 求4!的递归程序:

```
#include<iostream>
using namespace std;
int f(int n)
{   if(n==1) return 1;   //A
    else return n*f(n-1);//B
}

int main(void)
{   cout<<"4!="<<f(4)<<'\n';
    return 0;
}
```

- 递归调用本质上是嵌套调用，只不过是嵌套调用自身。
- f(4)函数调用时，系统自动为它的形参n分配内存，用于接收实参，递推时向嵌套调用的函数传递参数，回归时要向主调函数返回值。嵌套调用f(3)、f(2)、f(1)函数时，系统也分别自动为它们的形参n分配独立内存。

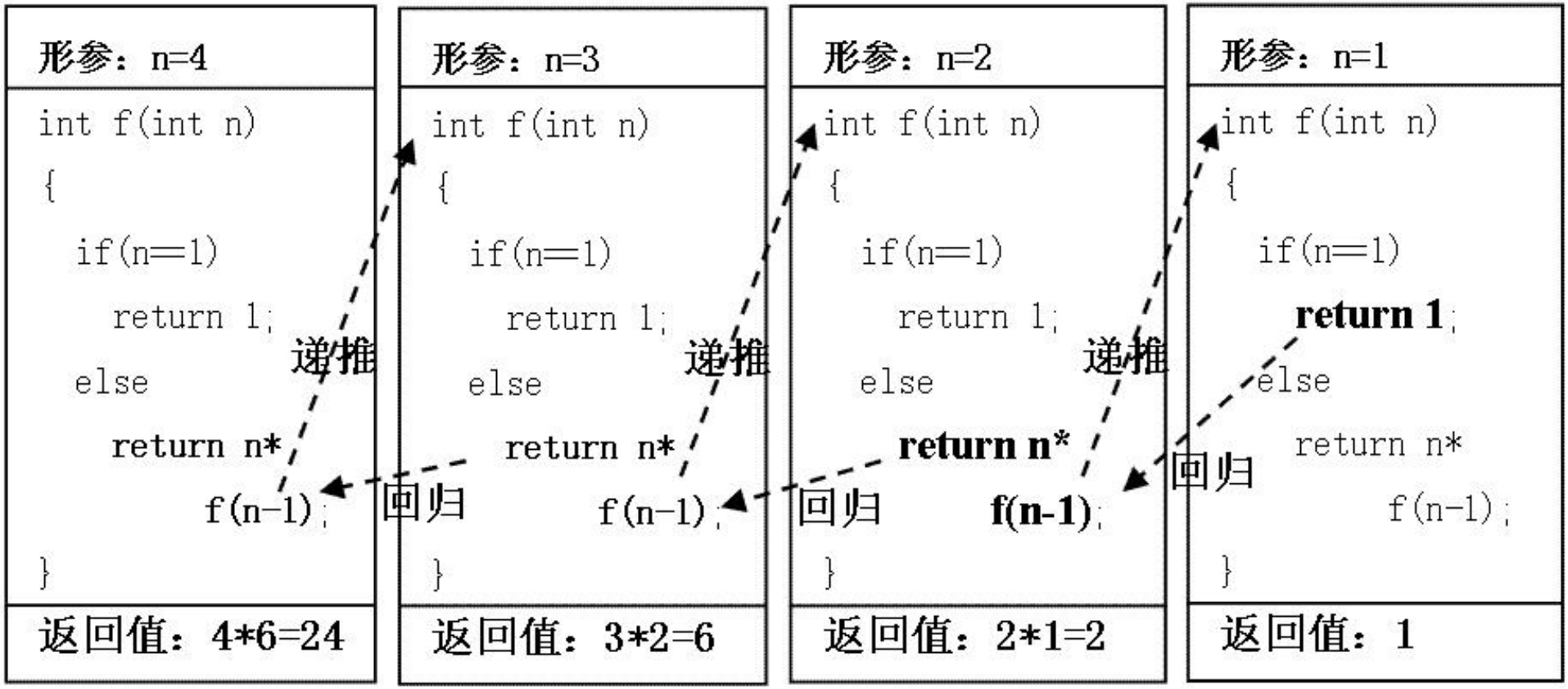
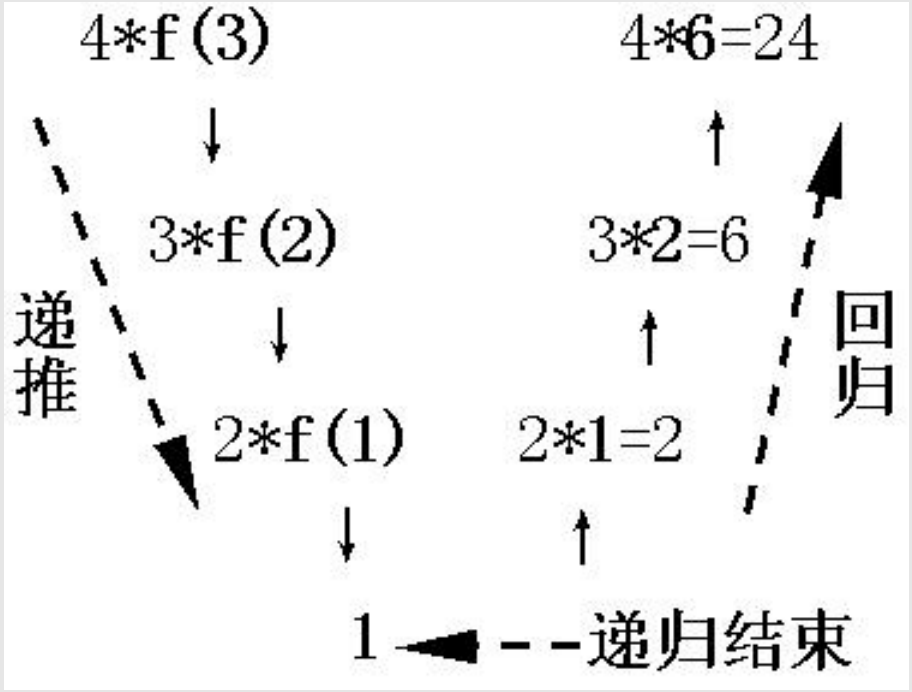


图　　f(4)函数调用的代码执行过程

- 递归函数的执行包括递推和回归两个过程。一旦调用递归函数，这两个过程由系统自动执行。
- 以f(4)函数调用为例说明递归函数的执行。因f(4)中实参不为1，故执行B行语句，即成为4*f(3)。



f(3)成为3*f(2)，…，直到出现f(1)函数调用时，才执行A行语句，将值1返回。如图左边所示。当出现函数调用f(1)时，递归结束，转入回归过程。将返回值1与2相乘后的结果作为f(2)的返回值，与3相乘后，结果值6作为f(3)的返回值，依次回归。如图右边所示。

- 从递归函数f的代码和其执行过程看，尽管函数体只使用if分支结构，代码非常简单，但通过递归调用隐式实现了循环。正是通过递归调用将循环隐藏起来，大大简化了算法的表达。
- 递归法解决问题的关键：

(1)善于归纳问题的递归特征：将原问题转化为一个新问题，而这个新问题与原问题有相同的解决方法。继续这种转化，直到转化出来的问题是一个有已知解的问题为止。例如，n!可转化为n*(n-1)!，最终1!=1。

(2)善于分析和总结问题的递归求解结束条件：只有有限次递归才有实际意义。例如，求n!时，n为1是递归结束条件。

- 例2.10 输入一个整数，逐位正序和反序输出。如输入3456，则输出3456和6543。分别设计两个函数，一个实现正序输出，另一个实现反序输出。
- 分析：求各位数可转换成将给定的整数重复除以10求余，求出的余数就是对应位上的数值，直到商为0为止。如3456%10的余数为6，即先得到最低位上的数值为6，商为345；345%10的余数为5，商为34；34%10的余数为4，商为3；3%10的余数为3，商为0，至此结束。
- 将问题转换为求余问题，可用递归方法解决：

$$f(n)=\begin{cases} n\%10 & \\ f(n/10) & n/10!=0 \end{cases}$$

```
#include<iostream>
using namespace std;
void fzx(int n)//先递归，后输出余数，则为正序
{
   if(n/10!=0) fzx(n/10);
   cout<<n%10<<'\t';
}

void ffx(int n)//先输出余数，后递归，则为反序
{
   cout<<n%10<<'\t';
   if(n/10!=0) ffx(n/10);
}
```

```
int main(void)
{  int m;
   cout<<"请输入一个整数：  ";
   cin>>m;
   cout<<"正序输出：  ";
   fzx(m);
   cout<<"\n反序输出：  ";
   ffx(m);
   return 0;
}
```

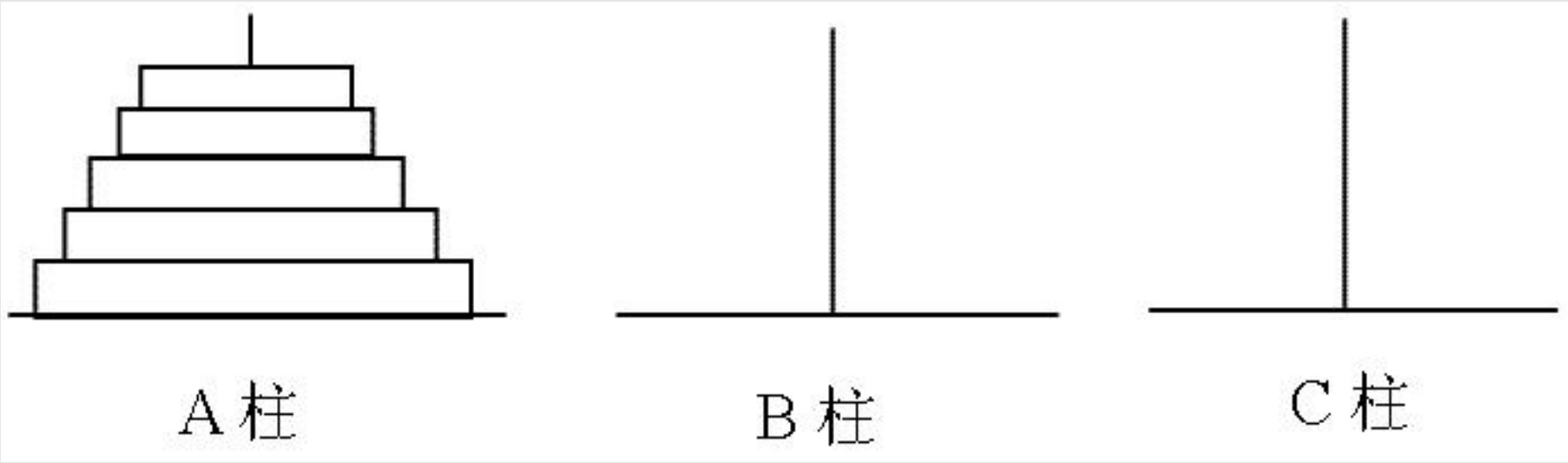
程序运行结果:

请输入一个整数：3456↙

正序输出：3 4 5 6

反序输出：6 5 4 3

- 因递归调用大大简化了算法的表达，故递归成为设计和描述算法的有力工具，常用于描述复杂算法。
- 例2.11 汉诺塔问题。有三根柱子A、B、C。设A柱上有n个盘子，盘子的大小不等，大的盘子在下，小的盘子在上，如下图所示。要求将A柱上的n个盘子移到C上，每一次只能移一个盘子。在移动过程中，可以借助于任一根柱子，但必须保证三根柱子上的盘子都是大的盘子在下，小的盘子在上。要求编一个程序打印出移动盘子的步骤。



- 分析：若用常规方法解决这个问题，很快发现这个问题太复杂。但用递归方法，就可将移动n个盘子的问题简化为移动n-1个盘子的问题，即将n个盘子从A柱移到C柱可分解为下面三个步骤：

(1)将A柱上的n-1个盘子借助于C柱移到B柱上；

(2)将A柱上的最后一个盘子移到C柱上；

(3)再将B柱上的n-1盘子借助于A柱移到C柱上。

这种分解可一直递推下去，直到变成移动一个盘子，递推结束。其实以上三个步骤只包含两种操作：

(1)将多个盘子从一根柱子移到另一根柱子上，是一个递归函数。 用hanoi(int n,char A,char B,char C)函数把A柱上的n个盘子借助于B柱移到C柱上。

(2)将一个盘子从一根柱子移到另一根柱子。用函数move(char x,char y)将1个盘子从x柱移到y柱，并输出移动盘子的提示信息。

```
#include<iostream>
using namespace std;
void move(char,char);
void hanoi(int,char,char,char);

void main(void)
{  int n;
   cout<<"Enter the number of diskes:";
   cin>>n;
   hanoi(n,'A','B','C');
}

void move(char x,char y)//将1个盘子从x柱移到y柱
{  cout<<x<<"→"<<y<<endl;  }
```

```
//把A柱上的n个盘子借助于B柱移到C柱上
void hanoi(int n,char A,char B,char C)
{
   if(n==1) move(A,C);
   else{
      //将A柱上的n-1个盘子借助于C柱移到B柱上
      hanoi(n-1,A,C,B);
      //将A柱上的最后一个盘子移到C柱上
      move(A,C);
      //再将B柱上的n-1盘子借助于A柱移到C柱上
      hanoi(n-1,B,A,C);
   }
}
```

2.6 作用域和存储类型

- 作用域（可见性）：程序中说明的标识符的有效区域（空间概念）。分为：

(1)块作用域（局部作用域)

(2)文件作用域（全局作用域)

(3)函数原型作用域

(4)函数作用域

(5)类作用域：后续章节介绍。

(6)命名空间作用域：后续章节介绍。

- 生存期：变量在什么时间内存在（时间概念）。

2.6.1作用域

1. 块作用域

- 块：用一对花括号“{}”括起来的部分程序。
- 块作用域：在块内说明的标识符，其作用域始于标识符的说明处，止于块的结尾处。只能在该块内引用。

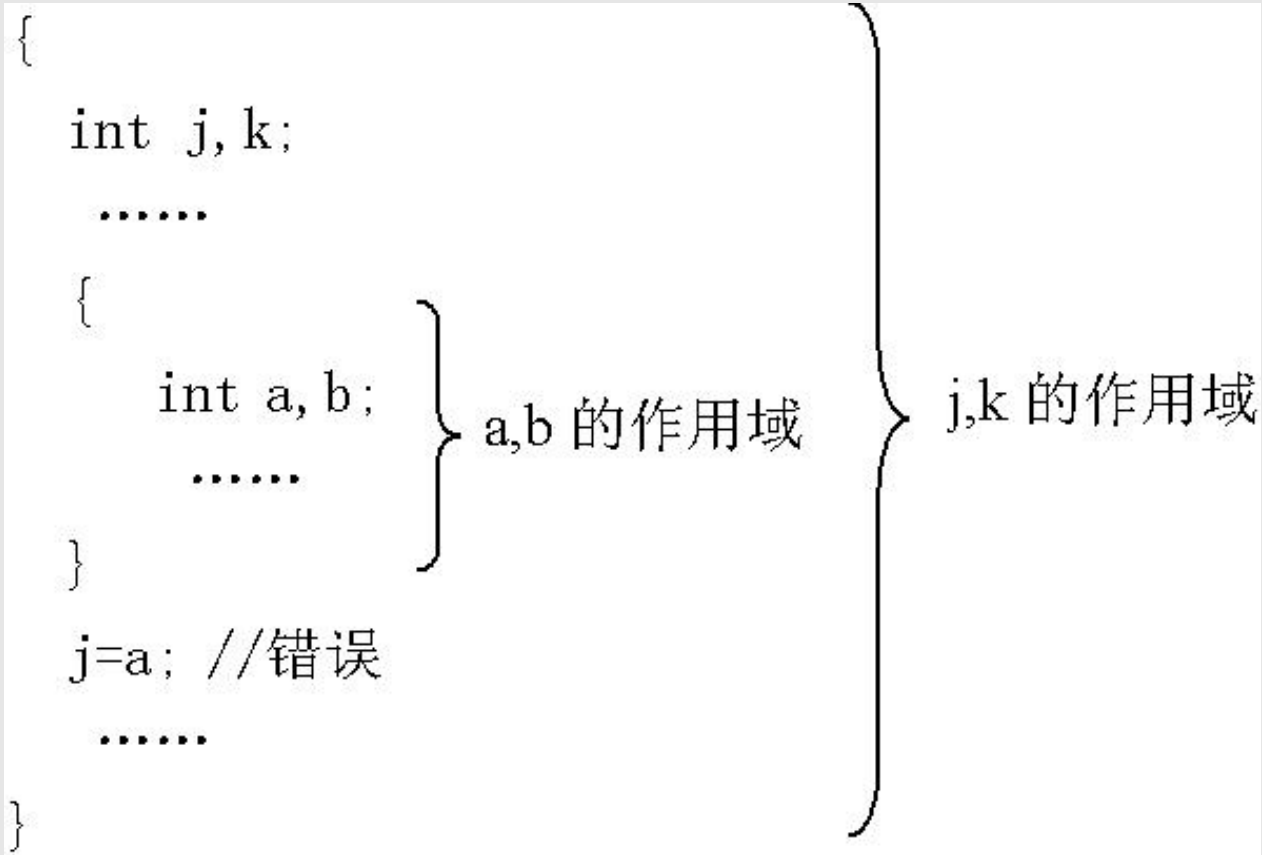


图 6-7 块作用域

局部变量

- 局部变量：在函数内或在块内定义的变量。
- 局部变量的内存：程序执行到该块时，系统自动在栈内为局部变量分配内存，在退出该块时，系统自动回收该块的局部变量占用的内存。
- 局部变量仅在其作用域范围内有效。
- 举例如图6-7所示：

(1)j、k、a、b均为局部变量。

(2)作用域可以覆盖。如j、k变量的作用域覆盖了a、b变量的作用域。

不同作用域的局部变量同名问题

- 标识符同名：同一作用域不允许，不同作用域允许。
- 不同作用域的局部变量的同名问题。

```
#include<iostream>
using namespace std;
int main(void)
{  int i=10,j=20,k=i+j;
   cout<<i<<','<<j<<','<<k<<'\n';
   {  int i=50,j=60;
      k=i+j;
      cout<<i<<','<<j<<'\n';
   }
   cout<<i<<','<<j<<','<<k<<'\n';
   return 0;
}
```

- 规则：内层标识符在其作用域内，将屏蔽其外层作用块的同名标识符。

for语句中说明的变量具有块作用域

- 在for语句中说明的变量具有块作用域。在标准C++中，其作用域仅作用于for语句；但VC++中，其作用域为包含for语句的那个内层块，并非仅作用于for语句。
- 例如：

```
{  for(int i=0;i<10;i++) cout<<i*i<<'\t';
   cout<<i;  //VC++允许，标准C++不允许
}
```

在VC + + 中，这段程序等同于：

```
{  int i;
   for(i=0;i<10;i++) cout<<i*i<<'\t';
   cout<<i;
}
```

2. 文件作用域

- 全局变量(标识符)：在函数外定义的变量(标识符)或用extern说明的变量(标识符)。
- 文件作用域：全局变量(标识符)的作用域。从定义位置开始到该源程序文件结束。
- 当全局变量出现使用在前而说明在后时，要先对全局变量作外部说明，其方法在后面介绍。
- 通过全局变量，增加了函数之间传递数据的途径，但也使函数的独立性降低。由于一个函数对外部变量的修改，直接影响到其他引用这个变量的函数，因此，外部变量的过多使用，会导致函数间依赖性增强。

全部变量示例

```
#include <iostream>
using namespace std;
int m=10;
void f1(int n)
{ n=2*n;m=m/3; }
int n;
void f2()
{ n=5;m++;n++; }
int main()
{ int n=2;
  f1(n);  f2();
cout<<m<<"  "<<n<<endl;
system("pause");  return 0;
}
```

定义了全局变量m

定义了全局变量n

定义了局部变量n

全局变量m的作用域

全局变量n的作用域

局部变量n的作用域

问题一：

全局变量m，n的作用域是什么？

问题二：

main函数中局部变量n的作用域是什么？

问题三：

main函数既是全局变量n的作用域，也是局部变量n的作用域，main函数输出语句中的n是哪一个n？

注意:

当全局变量与局部变量同名且作用域有重叠时，在局部变量的作用域内，起作用的是局部变量，而全局变量被“屏蔽”掉。

全部变量示例

全局变量 m

形参n(局部变量)

全局变量n

局部变量n

```
#include <iostream>
using namespace std;
int m=10;
void f1(int n)
{ n=2*n;m=m/3; }
int n;
void f2()
{ n=5;m++;n++; }
int main()
{ int n=2;
  f1(n);  f2();
  cout<<m<<" "<<n<<endl;
system("pause");  return 0;
}
```

//m=3,n=4（形参n）

//m=4,n=6（全局变量n）

//全局变量m，局部变量n

问题四：

程序的输出结果是什么？

局部变量与全局变量同名问题

```
#include<iostream>
using namespace std;
int i=10;              //全局变量i
void main(void)
{  int i=19,j=50;
   ::i=::i+2;          //全局变量i
   j=::i + i ;         //全局变量i、局部变量i和j
   cout<<"::i="<<::i;//全局变量i
   cout<<",i="<<i<<",j="<<j<<'\n';//局部变量i和j
   return 0;
}
```

- 规则1：局部变量在其作用域内，将屏蔽与其同名的全局变量。

- 规则2：在同名局部变量作用域内，用作用域运算符可访问同名的全局变量。

程序运行结果：

::i=12,i=19,j=31

3. 函数原型作用域

- 函数原型作用域：函数原型参数表中说明的标识符，其作用域始于说明处，止于函数原型说明的结束处。
- 函数原型中说明的标识符仅有形式上的意义：

(1)函数原型中说明的标识符可与其定义不同，例如：

```
float f(int x,float y);//f( )的原型说明
float f(int a,float b) //f( )的定义
{  return a+b;  }
```

(2)函数原型中说明的标识符也可省略，例如：

```
float f(int,float);
```

4. 函数作用域

- 函数作用域：只有标号具有函数作用域，即在一个函数中定义的标号，在其整个函数内均可以引用。
- 同一函数内不允许标号同名，而在不同函数内允许标号同名。
- 不允许在一个函数内用goto语句转移到另一个函数内的某一个语句去执行。

2.6.2 存储类型

- 程序执行时，系统为它分配内存并将它装入内存。
- 程序占用内存分为：

(1)程序区：存放程序的可执行代码。
(2)静态存储区：存放程序中定义的静态变量。
(3)动态存储区：存放程序中定义的动态变量。

- 动态变量：当程序执行到动态变量的作用域的开始处时，才为它分配内存；而执行到它的作用域的结束处时，收回为它分配的内存。该变量的生命期仅在变量的作用域内。

- 静态变量：在程序开始执行时就为其分配内存的变量，直到程序执行结束时，才收回为变量分配的内存。静态变量的生命期为程序的执行期，即静态变量一直占用所分配的内存，不管是否处在它的作用域。

- 存储类型：在变量说明时指定何时为变量分配内存，何时收回分给变量的内存，反映了变量占用内存的期限。引入存储类型是为了提高内存使用效率。
- 变量的存储类型：自动类型、寄存器类型、静态类型和外部类型。

1. 自动类型变量

- 自动类型变量：在说明局部变量时，用auto修饰。可缺省。
- 作用域：从定义点开始到所在的分程序结束
- 生存期：所在分程序执行期间
- 初始化：自动类型变量为动态变量，可以初始化，缺省值为随机值。
- 举例：

```
void f(void)
{  int x;  //默认为：  auto int x;
   auto int y;
}
```

2. 静态类型变量

- 静态类型变量：用static修饰的变量，为静态变量。在说明静态类型变量时，若没有指定初值，则编译器将其初值置为0。
- 作用域：从定义点到所在分程序结束。
- 生存期：程序的整个执行周期。
- 例如：

```
static float y=4.5f;//静态全局变量y，初值4.5
static char s;      //静态全局变量s，初值0
void f(void)
{  static int x;    //静态局部变量x，初值0
   cout<<x<<','<<y<<','<<s<<endl;
}
```

静态局部变量举例

```
#include<iostream>
using namespace std;
int f(int i)
{  static int r=1;
   r*=i;
   return r;
}
int main(void)
{  for(int i=1;i<5;i++)
     cout<<i<<"!="<<f(i)<<'\n';
   return 0;
}
```

静态局部变量r
- 生存期：与程序的执行期相同。
- 作用域：局限于f( )函数内。
- 初始化：仅在函数f( )首次调用时。

程序运行结果：

```
1!=1
2!=2
3!=6
4!=24
```

静态全局变量

- 全局变量：静态存储类型，其缺省初值为0。

- 静态全局变量：在说明全局变量时，用static修饰。表示所说明的变量仅限于本程序文件内使用。
- 若一个程序仅由一个文件组成，在说明全局变量时，有无static修饰，并无区别。

- 对于多文件构成的程序来说，如果将仅局限于一个文件中使用的全局变量加static修饰，则能有效避免全局变量的重名问题。

3.寄存器类型变量

- 寄存器类型变量：用register修饰的局部变量。register指示编译器为这类变量尽可能直接分配CPU中的寄存器，以提高存取速度。
- 例如：

  register int i,j;
- 说明：

(1)寄存器类型变量主要用作循环变量，存放临时值。
(2)静态变量和全局变量不能定义为寄存器类型变量。
(3)有的编译系统把寄存器变量作为自动变量来处理，有的编译系统限制了定义寄存器变量的个数。

4.外部类型变量

- 外部类型变量：在说明全局变量时，用extern修饰。
  主要用于下列两种情况：

(1)同一文件中，全局变量使用在前，定义在后，例如：

```
#include<iostream.h>
void f(int i)
{    extern int x; //引用性说明：x 为外部变量
     x+=i;         //使用全局变量 x
}
int x=10;          //定义性说明：x 为全局变量
void main(void)
{    f(5);
     cout<<x<<'\n';//此时 x 的值为 15
}
```

扩展全局变量x的作用域至f函数

全局变量x的原始作用域

(2)多文件组成一个程序时，一个源程序文件中定义的全局变量要被其他若干个源程序文件引用时，例如：

```
//c1.cpp
#include<iostream>
using namespace std;
float x;
extern float f( );  //外部函数原型声明
void main(void){  cout<<f( )<<'\n';  }

//c2.cpp
float f( )
{  //引用性说明外部变量x，使其作用域扩至f函数
   extern float x; //问：这样声明有何优点？
   return x+=2;
}
```

存储类型小结

- 基于变量访问的安全性以及内存利用效率等方面的考虑，在实际编程时，优先使用自动变量，严格限制使用静态变量和外部变量，基本不用寄存器变量。
- 不得不使用静态变量时，优先使用静态局部变量。
- 不得不使用全局变量时，优先使用静态全局变量。
- 不得不使用外部变量时，优先采用块作用域对外部变量的作用域做扩展。

2.7 C++增加的函数特性

- C++对函数的声明、定义、调用方式和实现机制做了进一步完善和改进，增加了：

(1)内联函数

(2)带缺省参数值的函数

(3)重载函数

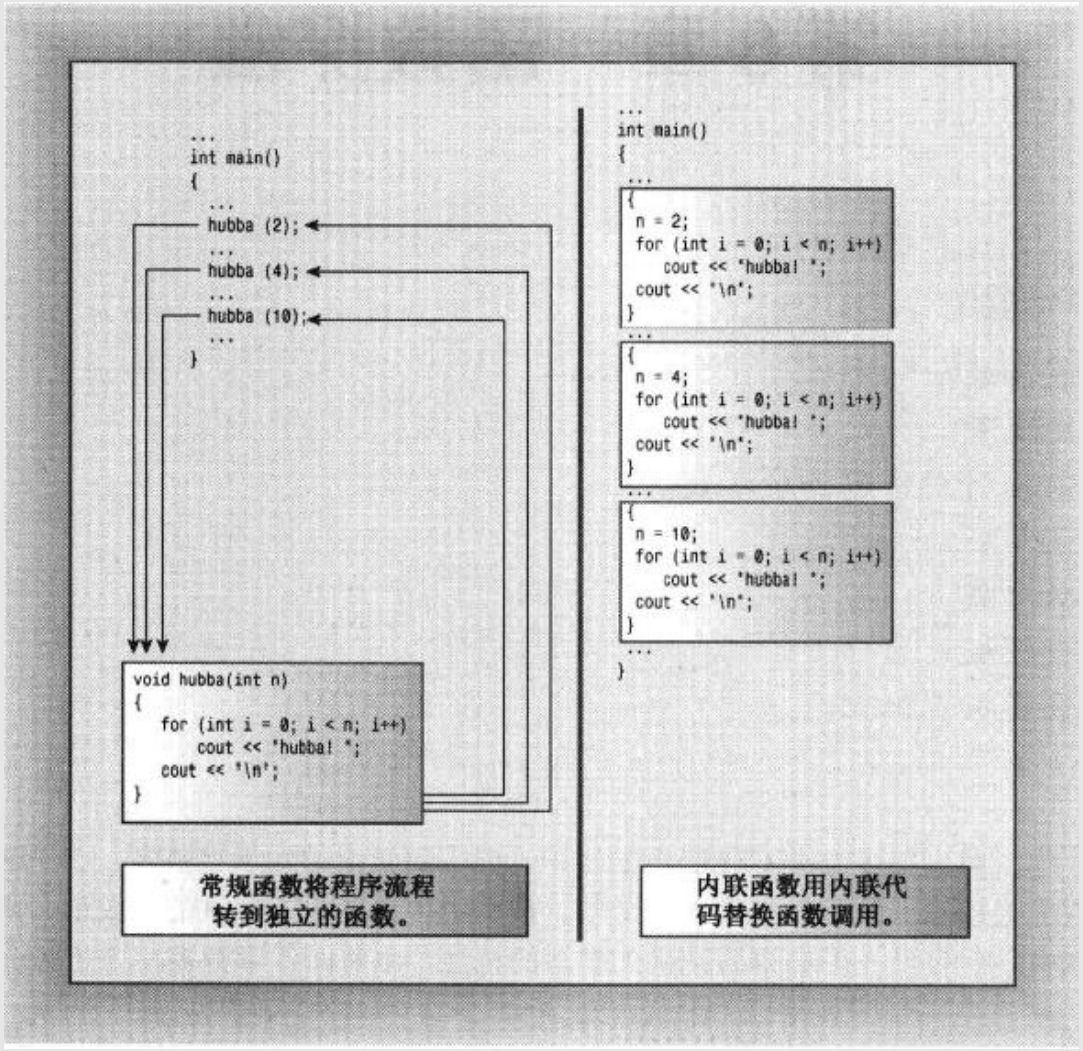
(4)函数模板：15.2节介绍。

2.7.1 内联函数

- 背景：函数调用时都会产生一些额外的时间开销，主要是系统栈的保护、参数的传递、系统栈的恢复等。对于那些函数体很小、执行时间很短但又频繁使用的函数来说，这种额外时间开销很可观。
- 内联函数机制：不是在调用时发生转移，而是在编译时将函数体嵌入到每个内联函数调用处。这样就省去了参数传递、系统栈的保护与恢复等的时间开销。
- 内联函数的定义：

```
inline <类型标识符><函数名> (形参表)
{
    函数体
}
```

- 只有简单、频繁调用的函数才有必要说明为内联的。
- 内联函数的本质：以增加程序代码的存储开销为代价来减少程序执行的时间开销。



- 内联函数举例：

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

- 内联函数说明：

(1)内联函数一般不能含有循环语句和switch语句。

(2)内联函数的定义必须出现在第一次被调用之前。

(3)内联函数不能指定抛掷异常的类型。

(4)函数用inline修饰只是向编译器提出了内联请求，编译器是否作为内联函数来处理由编译器决定。

2.7.2 缺省参数值的函数

- 当函数的参数在多数情况下使用相同的值时，为这些参数设定缺省值可以简化函数的调用。
- 例2.17 具有缺省参数值的延时函数

```
#include<iostream>
using namespace std;
void delay(int n=1000){  for( ;n>0;n--);  }
int main(void)
{  cout<<"延时1000个单位时间…\n";
   delay( );
   cout<<"延时800个单位时间…\n";
   delay(800);
   return 0;
}
```

使用具有缺省参数的函数时，应注意：

(1)缺省参数的说明必须出现在函数调用之前,说明方法有两种：①具有缺省参数值的函数的定义在函数调用之前说明，如例2.17所示。②先给出函数的原型说明，并在原型说明中依次列出参数的缺省值，但在后面的函数定义中，不能再指定函数参数的缺省值，如例2.18所示。

(2)参数的缺省值可以是表达式，但应有确定的值。

(3)函数的缺省参数可有多个，但缺省参数应从参数表的最右边依次向左设定。如例2.18中A行不能写为：

```
float Area(float x = 10,float y);
```

(4)同一函数在相同的作用域内，默认形参值的说明应保持唯一；在不同作用域，可用函数原型声明方式提供不同的缺省参数值。

```
int add(int x=5, int y=6);
void main( )
{
    int add(int x=7,int y=8);
    int ret = add();  //实现7+8
}
void func( ){ add( );} //实现5+6
int add(int x, int y){ return x+y;}
```

- 例2.18 输入长方形的长和宽，计算长方形的面积。

```
#include<iostream>
float Area(float x, float y=10);//A
using namespace std;
void main(void)
{  float x, y;
   cout<<"输入第一个长方形的长和宽：  ";
   cin>>x>>y;
   cout<<"第一个长方形的面积="<<Area(x,y);
   cout<<"\n输入第二个长方形的长：  ";
   cin>>x;
   cout<<"第二个长方形的面积="<<Area(x);
}
float Area(float l,float w){ return l*w; }
```

2.7.3 函数的重载

- 函数重载：两个或两个以上的函数同名，但形参的类型或形参的个数有所不同。仅返回值不同，不能定义为重载函数。
- 函数重载的原则：只有功能相近的函数才有必要重载。互不相干的函数使用函数重载，只会造成混乱，降低程序的可读性。
- 函数重载的好处：合理使用函数重载可以减轻用户对函数名的记忆负担，方便用户对函数的调用，提高程序的可读性。

- 两个以上的函数，具有相同的函数名，但是形参的个数或者类型不同，编译器会根据实参的类型及个数的最佳匹配来自动确定调用哪一个函数。

```
//形参列表不同
int add(int x, int y){…..};
float add(float x, float
y){…..};  int add(int x, int y,
int z){…..};
float add(float x, float y, float
z){…..};
```

```
void
    main()
    {  int a ,
    b ,c;  float     //int add(int x, int y);
    f1, f2;          //float add(float x, float y);
    add(a, b);        //int add(int x, int y,int z);
    add(f1, f2);
    add(a, b, c);
}
```

- **注意：不能以形参名字或函数返回类型的不同来区分函数。**

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

//错误！ 编译器不以形参名来区分函数

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

//错误！ 编译器不以返回值来区分函数

函数重载举例

- 例2.19 重载求平方的函数，实现求整数、单精度浮点数和双精度浮点数的平方值。
- #include<iostream>

```
using namespace std;
int Square(int x){  return x*x;  }      //A
float Square(float x){  return x*x;  }  //B
double Square(double x){  return x*x; } //C
int main(void)
{  cout<<Square(2)          //调用A行函数
       <<','<<Square(3.0f) //调用B行函数
       <<','<<Square(4.0); //调用C行函数
   return 0;
}
```