第4章 函数与预处理

4.1 概述

一个较大的程序不可能完全由一个人从头至尾地完成，更不可能把所有的内容都放在一个主函数中。为了便于规划、组织、编程和调试，一般的做法是把一个大的程序划分为若干个程序模块(即程序文件)，每一个模块实现一部分功能。不同的程序模块可以由不同的人来完成。在程序进行编译时，以程序模块为编译单位，即分别对每一个编译单位进行编译。如果发现错误，可以在本程序模块范围内查错并改正。在分别通过编译后，才进行连接，把各模块的目标文件以及系统文件连接在一起形成可执行文件。

在一个程序文件中可以包含若干个函数。无论把一个程序划分为多少个程序模块，只能有一个**main**函数。程序总是从**main**函数开始执行的。在程序运行过程中，由主函数调用其他函数，其他函数也可以互相调用。在**C**语言中没有类和对象，在程序模块中直接定义函数。可以认为，一个**C**程序是由若干个函数组成的，**C**语言被认为是面向函数的语言。**C++**面向过程的程序设计沿用了**C**语言使用函数的方法。在**C++**面向对象的程序设计中，主函数以外的函数大多是被封装在类中的。主函数或其他函数可以通过类对象调用类中的函数。无论是**C**还是**C++**，程序中的各项操作基本上都是由函数来实现的，程序编写者要根据需要编写一个个函数，每个函数用来实现某一功能。因此，读者必须掌握函数的概念以及学会设计和使用函数。

“函数”这个名词是从英文**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。

在实际应用的程序中，主函数写得很简单，它的作用就是调用各个函数，程序各部分的功能全部都是由各函数实现的。主函数相当于总调度，调动各函数依次实现各项功能。

开发商和软件开发人员将一些常用的功能模块编写成函数，放在函数库中供公共选用。程序开发人员要善于利用库函数，以减少重复编写程序段的工作量。

图**4.** 1 是一个程序中函数调用的示意图。

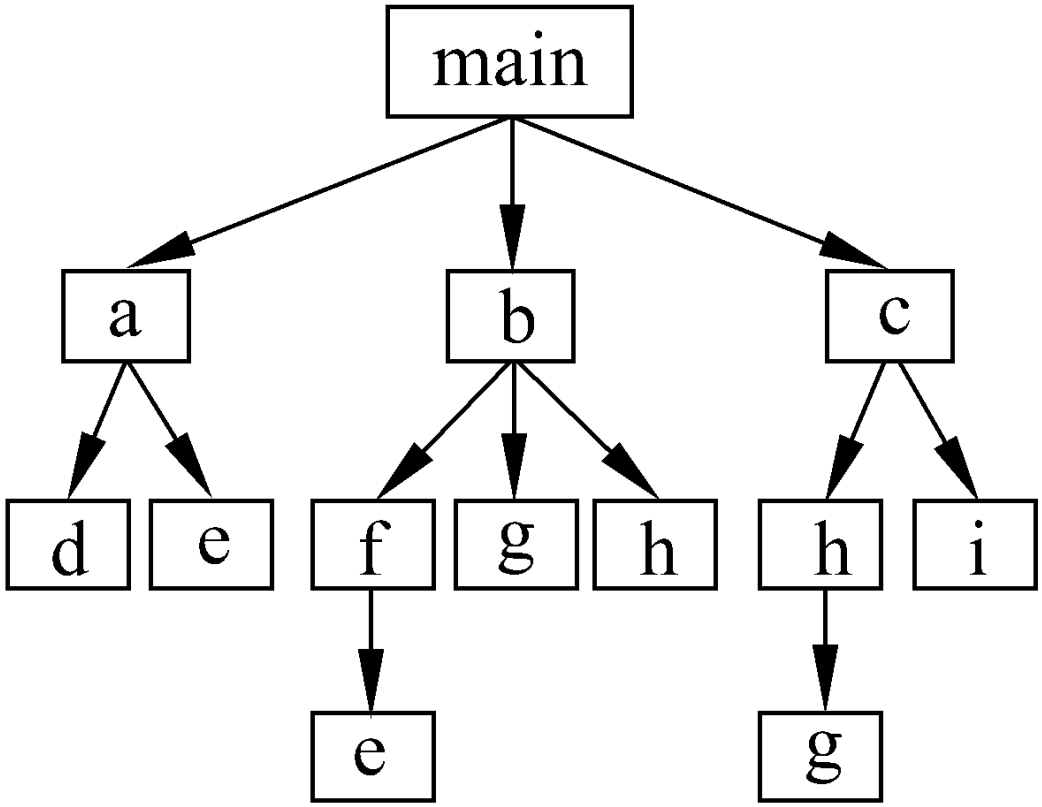


图**4.** 1

例4．1 在主函数中调用其他函数。

```
#include <iostream>
using namespace std;
void printstar(void)                    //定义printstar函数
{
 cout<<"****************************** "<<endl;    //输出30个“*”
}

void print_message(void)                //定义print_message函数
{
 cout<<"     Welcome  to  C++!"<<endl;          //输出一行文字
}

int main(void)
{
printstar( );                    //调用printstar 函数
  print_message( );                    //调用print_message函数
printstar( );                    //调用printstar 函数
return 0;
}
```

运行情况如下：

```
**********************************
    Welcome to C++!
**********************************
```

从用户使用的角度看，函数有两种：

(1) 系统函数，即库函数。这是由编译系统提供的，用户不必自己定义这些函数，可以直接使用它们。

(2) 用户自己定义的函数。用以解决用户的专门需要。

从函数的形式看，函数分两类：

(1) 无参函数。调用函数时不必给出参数。

(2) 有参函数。在调用函数时，要给出参数。在主调函数和被调用函数之间有数据传递。

4.2 定义函数的一般形式

4.2.1 定义无参函数的一般形式

定义无参函数的一般形式为

类型标识符 函数名（［**void**］）

｛声明部分

语句

}

例**4.1**中的**printstar**和**print_message**函数都是无参函数。用类型标识符指定函数的类型，即函数带回来的值的类型。

4.2.2 定义有参函数的一般形式

定义有参函数的一般形式为

类型标识符 函数名（形式参数表列）

｛声明部分

语句

｝

例如：

```
int max(int x，int y)        //函数首部，函数值为整型，有两个整型形参
{int z;                      //函数体中的声明部分
 z=x>y?x：y;                 //将x和y中的大者的值赋给整型变量z
return (z);                  //将z的值作为函数值返回调用点
}
```

C++要求在定义函数时必须指定函数的类型。

4.3 函数参数和函数的值

4.3.1 形式参数和实际参数

在调用函数时，大多数情况下，函数是带参数的。主调函数和被调用函数之间有数据传递关系。前面已提到：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），在主调函数中调用一个函数时，函数名后面括号中的参数(可以是一个表达式)称为实际参数（**actual parameter**，简称实参）。

例4.2 调用函数时的数据传递。

```
#include <iostream>
using namespace std;
int max(int x,int y)              //定义有参函数max
{int z;
 z=x>y?x：y;
 return(z);
}

int main( )
{int a,b,c;
 cout<<"please enter two integer numbers： ";
 cin>>a>>b;
 c=max(a,b);          //调用max函数，给定实参为a,b。函数值赋给c
 cout<<"max="<<c<<endl;
 return 0;
}
```

运行情况如下：

please enter two integer numbers：2 3↙

max=3

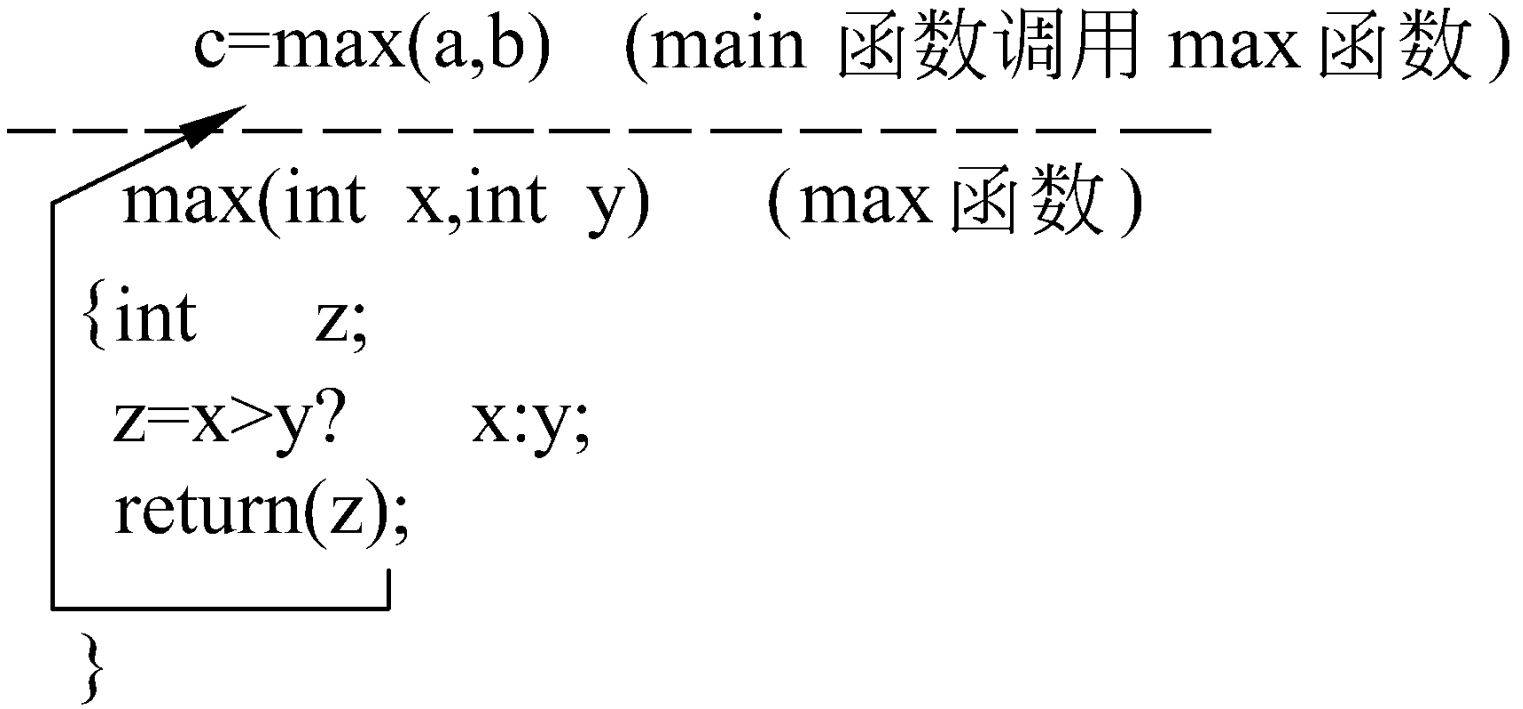


图4.2

有关形参与实参的说明：

(1) 在定义函数时指定的形参，在未出现函数调用时，它们并不占内存中的存储单元，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，只有在发生函数调用时，函数**max**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，形参所占的内存单元也被释放。

(2) 实参可以是常量、变量或表达式，如**max(3, a+b)**；但要求**a**和**b**有确定的值。以便在调用函数时将实参的值赋给形参。

(3) 在定义函数时，必须在函数首部指定形参的类型（见例**4.2**程序第**3**行）。

(4) 实参与形参的类型应相同或赋值兼容。例**4.2**中实参和形参都是整型，这是合法的、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。例如实参**a**的值为**3.5**，而形参**x**为整型，则将**3.5**转换成整数**3**，然后送到形参**b**。字符型与整型可以互相通用。

(5) 实参变量对形参变量的数据传递是**“**值传递**”**，即单向传递，只由实参传给形参，而不能由形参传回来给实参。在调用函数时，编译系统临时给形参分配存储单元。请注意： 实参单元与形参单元是不同的单元。图**4.3**表示将实参 a 和 b 的值**2**和**3**传递给对应的形参 x 和 y 。

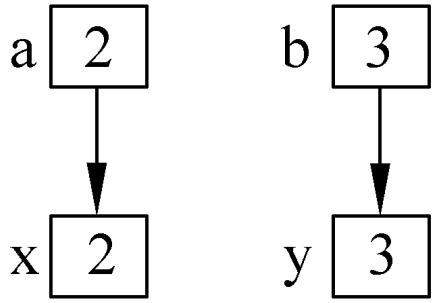


图**4.3**

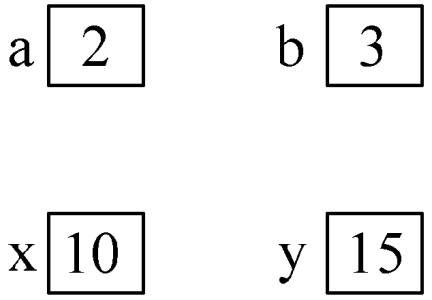


图**4.4**

调用结束后，形参单元被释放，实参单元仍保留并维持原值。因此，在执行一个被调用函数时，形参的值如果发生改变，并不会改变主调函数中实参的值。例如，若在执行**max**函数过程中形参 x 和 y 的值变为**10**和**15**，调用结束后，实参 a 和 b 仍为**2**和**3**，见图**4.4**。

4.3.2 函数的返回值

(1) 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去。

return语句后面的括号可以要，也可以不要。

return后面的值可以是一个表达式。

(2) 函数值的类型。既然函数有返回值，这个值当然应属于某一个确定的类型，应当在定义函数时指定函数值的类型。

(3) 如果函数值的类型和**return**语句中表达式的值不一致，则以函数类型为准，即函数类型决定返回值的类型。对数值型数据，可以自动进行类型转换。

4.4 函数的调用

4.4.1 函数调用的一般形式

函数名（［实参表列］）

如果是调用无参函数，则“实参表列”可以没有，但括号不能省略。如果实参表列包含多个实参，则各参数间用逗号隔开。实参与形参的个数应相等，类型应匹配(相同或赋值兼容)。实参与形参按顺序对应，一对一地传递数据。但应说明，如果实参表列包括多个实参，对实参求值的顺序并不是确定的。

4.4.2 函数调用的方式

按函数在语句中的作用来分，可以有以下3种函数调用方式：

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，只是要求函数完成一定的操作。如例4.1中的**printstar( );**

2. 函数表达式

函数出现在一个表达式中，这时要求函数带回一个确定的值以参加表达式的运算。如**c=2*max(a,b);**

3. 函数参数

函数调用作为一个函数的实参。如

m=max(a,max(b,c)); //**max(b,c)**是函数调用，其值作为外层**max**函数调用的一个实参

4.4.3 对被调用函数的声明和函数原型

在一个函数中调用另一个函数（即被调用函数）需要具备哪些条件呢？

(1) 首先被调用的函数必须是已经存在的函数。

(2) 如果使用库函数，一般还应该在本文件开头用**＃include**命令将有关头文件**“**包含**”**到本文件中来。

(3) 如果使用用户自己定义的函数，而该函数与调用它的函数（即主调函数）在同一个程序单位中，且位置在主调函数之后，则必须在调用此函数之前对被调用的函数作声明。

所谓函数声明**(declare)**，就是在函数尚在未定义的情况下，事先将该函数的有关信息通知编译系统，以便使编译能正常进行。

例**4.3** 对被调用的函数作声明。

```
#include <iostream>
using namespace std;
int main( )
{float add(float x,float y);            //对add函数作声明
 float a,b,c;
cout<<"please enter a,b:  ";
 cin>>a>>b;
 c=add(a,b);
 cout<<"sum="<<c<<endl;
 return 0;
}

float add(float x,float y)              //定义add函数
{float z;
 z=x+y;
 return (z);
}
```

运行情况如下：

please enter a,b：123.68 456.45↙

sum=580.13

注意：对函数的定义和声明不是同一件事情。定义是指对函数功能的确立，包括指定函数名、函数类型、形参及其类型、函数体等，它是一个完整的、独立的函数单位。而声明的作用则是把函数的名字、函数类型以及形参的个数、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，据此对其进行对照检查（例如函数名是否正确，实参与形参的类型和个数是否一致）。

其实，在函数声明中也可以不写形参名，而只写形参的类型，如

float add(float，float);

这种函数声明称为函数原型**(function prototype)**。使用函数原型是**C**和**C++**的一个重要特点。它的作用主要是： 根据函数原型在程序编译阶段对调用函数的合法性进行全面检查。如果发现与函数原型不匹配的函数调用就报告编译出错。它属于语法错误。用户根据屏幕显示的出错信息很容易发现和纠正错误。

函数原型的一般形式为

(1) 函数类型 函数名**(**参数类型**1**，参数类型**2…);**

(2) 函数类型 函数名**(**参数类型**1** 参数名**1**，参数类型**2** 参数名**2…);**

第(1)种形式是基本的形式。为了便于阅读程序，也允许在函数原型中加上参数名，就成了第(2)种形式。但编译系统并不检查参数名。因此参数名是什么都无所谓。上面程序中的声明也可以写成

```
float add(float a，float b);      //参数名不用x、y，而用a、b
```

效果完全相同。

应当保证函数原型与函数首部写法上的一致，即函数类型、函数名、参数个数、参数类型和参数顺序必须相同。在函数调用时函数名、实参类型和实参个数应与函数原型一致。

说明：

(1) 前面已说明，如果被调用函数的定义出现在主调函数之前，可以不必加以声明。因为编译系统已经事先知道了已定义的函数类型，会根据函数首部提供的信息对函数的调用作正确性检查。

有经验的程序编制人员一般都把**main**函数写在最前面，这样对整个程序的结构和作用一目了然，统览全局，然后再具体了解各函数的细节。此外，用函数原型来声明函数，还能减少编写程序时可能出现的错误。由于函数声明的位置与函数调用语句的位置比较近，因此在写程序时便于就近参照函数原型来书写函数调用，不易出错。所以应养成对所有用到的函数作声明的习惯。这是保证程序正确性和可读性的重要环节。

(2) 函数声明的位置可以在调用函数所在的函数中，也可以在函数之外。如果函数声明放在函数的外部，在所有函数定义之前，则在各个主调函数中不必对所调用的函数再作声明。例如：

```
char letter(char,char);   //本行和以下两行函数声明在所有函数之前且在函数外部
float f(float,float);     //因而作用域是整个文件
int i(float, float);
int main( )
 {…}                      //在main函数中不必对它所调用的函数作声明
char letter(char c1,char c2)    //定义letter函数
 {…}
float f(float x,float y)        //定义f函数
 {…}
int i(float j，float k)         //定义i函数
 {…}
```

如果一个函数被多个函数所调用，用这种方法比较好，不必在每个主调函数中重复声明。

*4.5 内置函数

调用函数时需要一定的时间和空间的开销。图4.5表示函数调用的过程：

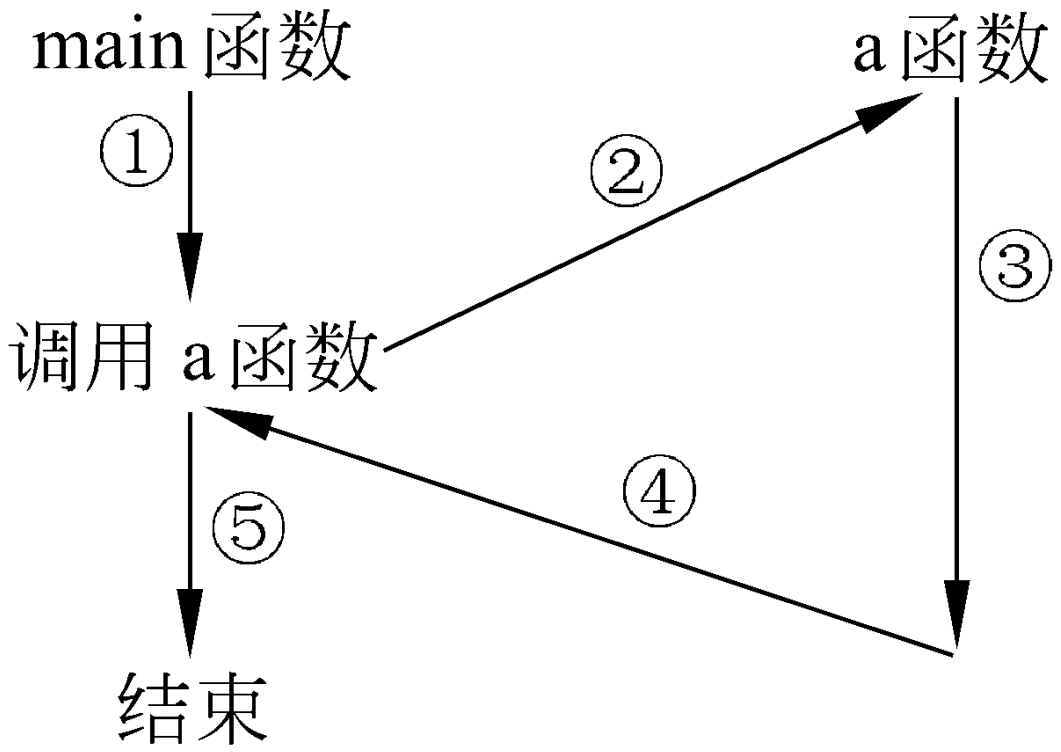


图4.5

C++提供一种提高效率的方法，即在编译时将所调用函数的代码直接嵌入到主调函数中，而不是将流程转出去。这种嵌入到主调函数中的函数称为内置函数**(inline function)**，又称内嵌函数。在有些书中把它译成内联函数。

指定内置函数的方法很简单，只需在函数首行的左端加一个关键字**inline**即可。

例**4.4** 函数指定为内置函数。

```
#include <iostream>
using namespace std;
inline int max(int,int, int);      //声明函数，注意左端有inline
int main( )
{int i=10,j=20,k=30,m;
 m=max(i,j,k);
 cout<<"max="<<m<<endl;
 return 0;
}

inline int max(int a,int b,int c)      //定义max为内置函数
{if(b>a) a=b;                          //求a,b,c中的最大者
 if(c>a) a=c;
 return a;
}
```

由于在定义函数时指定它为内置函数，因此编译系统在遇到函数调用“**max(i,j,k)**”时，就用**max**函数体的代码代替“**max(i,j,k)**”，同时将实参代替形参。这样，程序第**6**行 “**m=max(i,j,k);**”就被置换成

```
if (j>i) i=j;
if(k>i) i=k;
m=i;
```

注意： 可以在声明函数和定义函数时同时写**inline**，也可以只在其中一处声明**inline**，效果相同，都能按内置函数处理。

使用内置函数可以节省运行时间，但却增加了目标程序的长度。因此一般只将规模很小(一般为**5**个语句以下)而使用频繁的函数(如定时采集数据的函数)声明为内置函数。

内置函数中不能包括复杂的控制语句，如循环语句和**switch**语句。

应当说明： 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，也就是说它是建议性的，而不是指令性的。并非一经指定为**inline**，编译系统就必须这样做。编译系统会根据具体情况决定是否这样做。

归纳起来，只有那些规模较小而又被频繁调用的简单函数，才适合于声明为**inline**函数。

*4.6 函数的重载

在编程时，有时我们要实现的是同一类的功能，只是有些细节不同。例如希望从3个数中找出其中的最大者，而每次求最大数时数据的类型不同，可能是3个整数、3个双精度数或3个长整数。程序设计者往往会分别设计出3个不同名的函数，其函数原型为：

```
int max1(int a，int b, int c);              //求3个整数中的最大者
double max2(double a，double b，double c); //求3个双精度数中最大者
long  max3(long a，long b，long c);      //求3个长整数中的最大者
```

C++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(function overloading)。即对一个函数名重新赋予它新的含义，使一个函数名可以多用。

对上面求最大数的问题可以编写如下的C++程序。

例4.5 求3个数中最大的数（分别考虑整数、双精度数、长整数的情况）。

```
#include <iostream>
using namespace std;
int main( )
{int max(int a,int b,int c);                //函数声明
double max(double a,double b,double c);    //函数声明
long max(long a,long b,long c);            //函数声明
 int i1,i2,i3,i;
 cin>>i1>>i2>>i3;                           //输入3个整数
 i=max(i1,i2,i3);                           //求3个整数中的最大者
 cout<<"i_max="<<i<<endl;
 double d1,d2,d3,d;
 cin>>d1>>d2>>d3;                           //输入3个双精度数
 d=max(d1,d2,d3);                           //求3个双精度数中的最大者
 cout<<"d_max="<<d<<endl;
 long g1,g2,g3,g;
 cin>>g1>>g2>>g3;                           //输入3个长整数
```

```
 g=max(g1,g2,g3);                    //求3个长整数中的最大者
 cout<<"g_max="<<g<<endl;
}

int max(int a,int b,int c)                  //定义求3个整数中的最大者的函数
{if(b>a) a=b;
 if(c>a) a=c;
 return a;
}

double max(double a,double b,double c)   //定义求3个双精度数中的最大者的函数
{if(b>a) a=b;
 if(c>a) a=c;
 return a;
}

long max(long a,long b,long c)          //定义求3个长整数中的最大者的函数
{if(b>a) a=b;
 if(c>a) a=c;
 return a;
}
```

运行情况如下：

185 -76 567↙ (输入3个整数)

56.87 90.23 -3214.78↙ (输入3个实数)

67854 -912456 673456↙ (输入3个长整数)

i_max=567 （输出3个整数的最大值）

d_max=90.23 （输出3个双精度数的最大值）

g_max=673456 （输出3个长整数的最大值）

上例3个**max**函数的函数体是相同的，其实重载函数并不要求函数体相同。重载函数除了允许参数类型不同以外，还允许参数的个数不同。

例4.6 编写一个程序，用来求两个整数或3个整数中的最大数。如果输入两个整数，程序就输出这两个整数中的最大数，如果输入3个整数，程序就输出这3个整数中的最大数。

```
#include <iostream>
using namespace std;
int main( )
{int max(int a,int b,int c);            //函数声明
 int max(int a,int b);                  //函数声明
 int a=8,b=-12,c=27;
 cout<<"max(a,b,c)="<<max(a,b,c)<<endl;   //输出3个整数中的最大者
 cout<<"max(a,b)="<<max(a,b)<<endl;       //输出两个整数中的最大者
}

int max(int a,int b,int c)        //此max函数的作用是求3个整数中的最大者
{if(b>a) a=b;
```

```
 if(c>a) a=c;
 return a;
}

int max(int a,int b)              //此max函数的作用是求两个整数中的最大者
{if(a>b) return a;
else return b;
}
```

运行情况如下：

```
max(a,b,c)=27
max(a,b)=8
```

两次调用**max**函数的参数个数不同，系统就根据参数的个数找到与之匹配的函数并调用它。

参数的个数和类型可以都不同。但不能只有函数的类型不同而参数的个数和类型相同。例如：

```
int f(int);          //函数返回值为整型
long f(int);         //函数返回值为长整型
void f(int);         //函数无返回值
```

在函数调用时都是同一形式，如**“f(10)”**。编译系统无法判别应该调用哪一个函数。重载函数的参数个数、参数类型或参数顺序**3**者中必须至少有一种不同，函数返回值类型可以相同也可以不同。

在使用重载函数时，同名函数的功能应当相同或相近，不要用同一函数名去实现完全不相干的功能，虽然程序也能运行，但可读性不好，使人莫名其妙。

*4.7 函数模板

C++提供了函数模板**(function template)**。所谓函数模板，实际上是建立一个通用函数，其函数类型和形参类型不具体指定，用一个虚拟的类型来代表。这个通用函数就称为函数模板。凡是函数体相同的函数都可以用这个模板来代替，不必定义多个函数，只需在模板中定义一次即可。在调用函数时系统会根据实参的类型来取代模板中的虚拟类型，从而实现了不同函数的功能。看下面的例子就清楚了。

例**4.7** 将例**4.6**程序改为通过函数模板来实现。

```
#include <iostream>
using namespace std;
template<typename T>      //模板声明，其中T为类型参数
T max(T a,T b,T c)        //定义一个通用函数，用T作虚拟的类型名
{if(b>a) a=b;
 if(c>a) a=c;
 return a;
}

int main( )
{int i1=185,i2=-76,i3=567,i;
 double d1=56.87,d2=90.23,d3=-3214.78,d;
 long g1=67854,g2=-912456,g3=673456,g;
i=max(i1,i2,i3);          //调用模板函数，此时T被int取代
 d=max(d1,d2,d3);         //调用模板函数，此时T被double取代
g=max(g1,g2,g3);          //调用模板函数，此时T被long取代
 cout<<"i_max="<<i<<endl;
 cout<<"f_max="<<f<<endl;
 cout<<"g_max="<<g<<endl;
 return 0;
}
```

运行结果与例**4.5**相同。为了节省篇幅，数据不用**cin**语句输入，而在变量定义时初始化。

程序第**3~8**行是定义模板。定义函数模板的一般形式为

template < typename T> 或 **template <class T>**

通用函数定义　　通用函数定义

在建立函数模板时，只要将例**4.5**程序中定义的第一个函数首部的**int**改为**T**即可。即用虚拟的类型名**T**代替具体的数据类型。在对程序进行编译时，遇到第**13**行调用函数**max(i1,i2,i3)**，编译系统会将函数名**max**与模板**max**相匹配，将实参的类型取代了函数模板中的虚拟类型**T**。此时相当于已定义了一个函数：

```
int max(int a,int b,int c)
{if(b>a) a=b;
 if(c>a) a=c;
 return a;
}
```

然后调用它。后面两行(14,15行)的情况类似。

类型参数可以不只一个，可以根据需要确定个数。如

```
template <class T1,typename T2>
```

可以看到，用函数模板比函数重载更方便，程序更简洁。但应注意它只适用于函数的参数个数相同而类型不同，且函数体相同的情况，如果参数的个数不同，则不能用函数模板。

*4.8 有默认参数的函数

一般情况下，在函数调用时形参从实参那里取得值，因此实参的个数应与形参相同。有时多次调用同一函数时用同样的实参，C++提供简单的处理办法，给形参一个默认值，这样形参就不必一定要从实参取值了。如有一函数声明

```
float area(float r=6.5);
```

指定r的默认值为6.5，如果在调用此函数时，确认r的值为6.5，则可以不必给出实参的值，如

```
area( );                    //相当于area(6.5);
```

如果不想使形参取此默认值，则通过实参另行给出。如

```
area(7.5);                    //形参得到的值为7.5，而不是6.5
```

这种方法比较灵活，可以简化编程，提高运行效率。

如果有多个形参，可以使每个形参有一个默认值，也可以只对一部分形参指定默认值，另一部分形参不指定默认值。如有一个求圆柱体体积的函数，形参**h**代表圆柱体的高，**r**为圆柱体半径。函数原型如下：

```
float volume(float h,float r=12.5);       //只对形参r指定默认值12.5
```

函数调用可以采用以下形式：

```
volume(45.6);                  //相当于volume(45.6,12.5)
volume(34.2,10.4)              //h的值为34.2，r的值为10.4
```

实参与形参的结合是从左至右顺序进行的。因此指定默认值的参数必须放在形参表列中的最右端，否则出错。例如：

```
void f1(float a，int b=0，int c，char d='a');    //不正确
void f2(float a，int c，int b=0, char d='a');    //正确
```

如果调用上面的**f2**函数，可以采取下面的形式：

```
f2(3.5, 5, 3, 'x')          //形参的值全部从实参得到
f2(3.5, 5, 3)               //最后一个形参的值取默认值'a'
f2(3.5, 5)                  //最后两个形参的值取默认值，b=0,d='a'
```

可以看到，在调用有默认参数的函数时，实参的个数可以与形参的个数不同，实参未给定的，从形参的默认值得到值。利用这一特性，可以使函数的使用更加灵活。例如例**4.7**求**2**个数或**3**个数中的最大数。也可以不用重载函数，而改用带有默认参数的函数。

例4.8 求2个或3个正整数中的最大数，用带有默认参数的函数实现。

```
#include <iostream>
using namespace std;
int main( )
{int max(int a, int b, int c=0);//函数声明,形参c有默认值
int a,b,c;
 cin>>a>>b>>c;
 cout<<"max(a,b,c)="<<max(a,b,c)<<endl;   //输出3个数中的最大者
 cout<<"max(a,b)="<<max(a,b)<<endl;       //输出2个数中的最大者
 return 0;
}

int max(int a,int b,int c)        //函数定义
{if(b>a) a=b;
 if(c>a) a=c;
 return a;
}
```

运行情况如下：

14 -56 135↙

max(a,b,c)=135

max(a,b)=14

在使用带有默认参数的函数时有两点要注意：

(1) 如果函数的定义在函数调用之前，则应在函数定义中给出默认值。如果函数的定义在函数调用之后，则在函数调用之前需要有函数声明，此时必须在函数声明中给出默认值，在函数定义时可以不给出默认值(如例**4.8)**。

(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。因为当调用函数时如果少写一个参数，系统无法判定是利用重载函数还是利用默认参数的函数，出现二义性，系统无法执行。

4.9 函数的嵌套调用

C++不允许对函数作嵌套定义，也就是说在一个函数中不能完整地包含另一个函数。在一个程序中每一个函数的定义都是互相平行和独立的。

虽然C++不能嵌套定义函数，但可以嵌套调用函数，也就是说，在调用一个函数的过程中，又调用另一个函数。见图4.6。

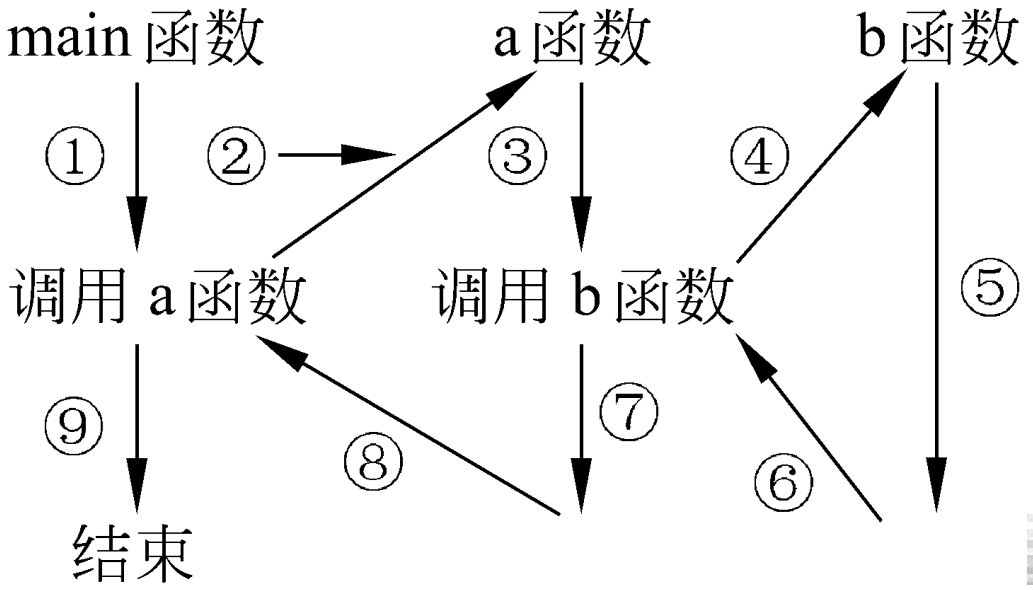


图4.6

在程序中实现函数嵌套调用时，需要注意的是：在调用函数之前，需要对每一个被调用的函数作声明(除非定义在前，调用在后)。

例4.9 用弦截法求方程$f(x)=x^3-5x^2+16x-80=0$的根。

这是一个数值求解问题，需要先分析用弦截法求根的算法。根据数学知识，可以列出以下的解题步骤：

(1) 取两个不同点x_1,x_2,如果f（x_1）和f（x_2）符号相反,则(x_1,x_2)区间内必有一个根。如果f（x_1）与f(x_2)同符号,则应改变x_1,x_2,直到f（x_1）, f(x_2)异号为止。注意x_1、x_2的值不应差太大,以保证(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)交x轴于x,见图4.7。

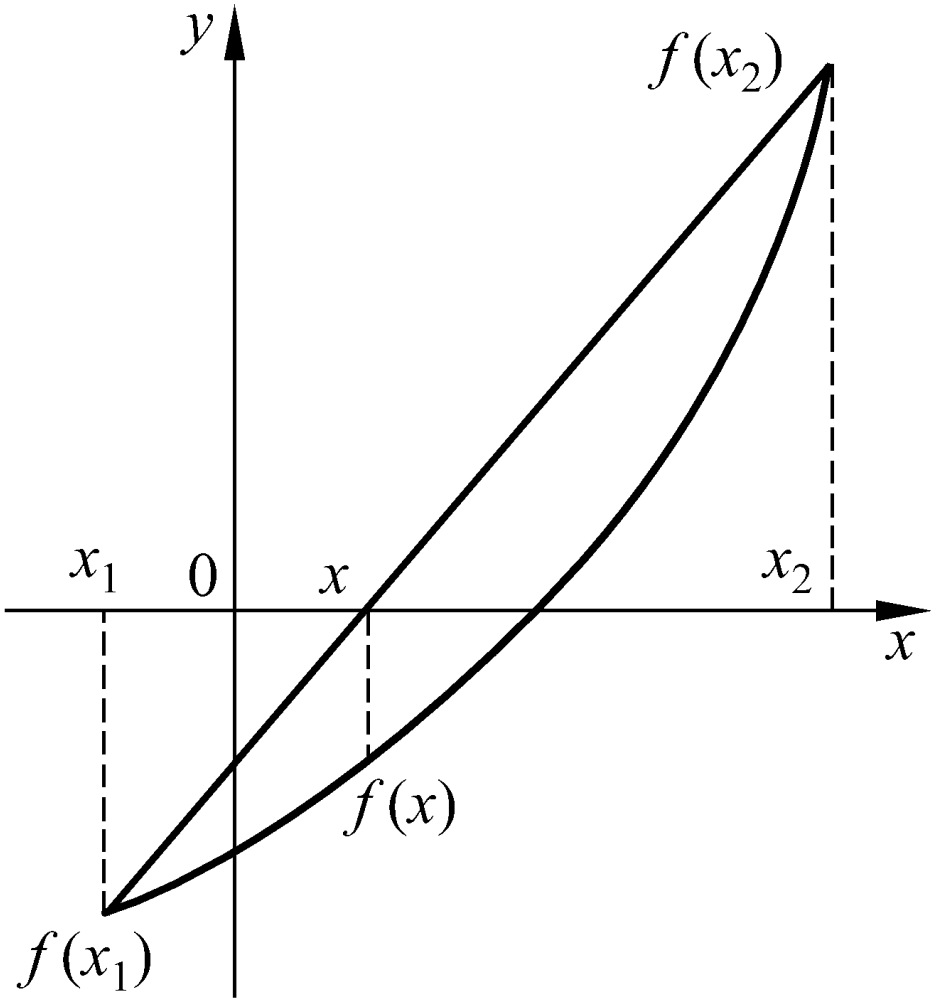


图**4.7**

x点坐标可用下式求出：

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

再从x求出f(x)。

(3) 若f(x)与f（x_1）同符号,则根必在(x, x_2)区间内,此时将x作为新的x_1。如果f(x)与f（x_2）同符号,则表示根在(x_1,x)区间内,将x作为新的x_2。

(4) 重复步骤 (2) 和 (3)， 直到 |f(x)|＜ξ 为止, ξ为一个很小的正数, 例如10^{-6}。此时认为 f(x)≈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以上有关部分功能：

(1) 用函数f(x)代表x的函数：$x^3-5x^2+16x-80$。

(2) 用函数**xpoint** (x_1,x_2)来求(x_1,**f**(x_1))和(x_2,**f**(x_2))的连线与**x**轴的交点**x**的坐标。

(3) 用函数**root**(x_1,x_2)来求(x_1,x_2)区间的那个实根。

显然,执行**root**函数的过程中要用到**xpoint**函数,而执行**xpoint**函数的过程中要用到**f**函数。

根据以上算法，可以编写出下面的程序:

```
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;
double f(double);                  //函数声明
double xpoint(double, double);     //函数声明
double root(double, double);       //函数声明

int main( )
{ double x1,x2,f1,f2,x;
```

```
 do
  {cout<<"input x1,x2： ";
   cin>>x1>>x2;
   f1=f(x1);
   f2=f(x2);
  } while(f1*f2>=0);
 x=root(x1,x2);
 cout<<setiosflags(ios：：fixed)<<setprecision(7);
//指定输出7位小数
 cout<<"A root of equation is "<<x<<endl;
 return 0;
}

double f(double x)              //定义f函数，以实现f(x)
{double y;
 y=x*x*x-5*x*x+16*x-80;
 return y;
}
```

```
double xpoint(double x1, double x2)     //定义xpoint函数，求出弦与 x
轴交点
{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)>=0.00001);
 return x;
}
```

运行情况如下：

input x1,x2：2.5 6.7↙

A root of equation is 5.0000000

对程序的说明：

(1) 在定义函数时，函数名为**f**，**xpoint**和**root**的**3**个函数是互相独立的，并不互相从属。这**3**个函数均定为双精度型。

(2) 3个函数的定义均出现在**main**函数之后，因此在**main**函数的前面对这**3**个函数作声明。

习惯上把本程序中用到的所有函数集中放在最前面声明。

(3) 程序从**main**函数开始执行。函数的嵌套调用见图**4.8**。

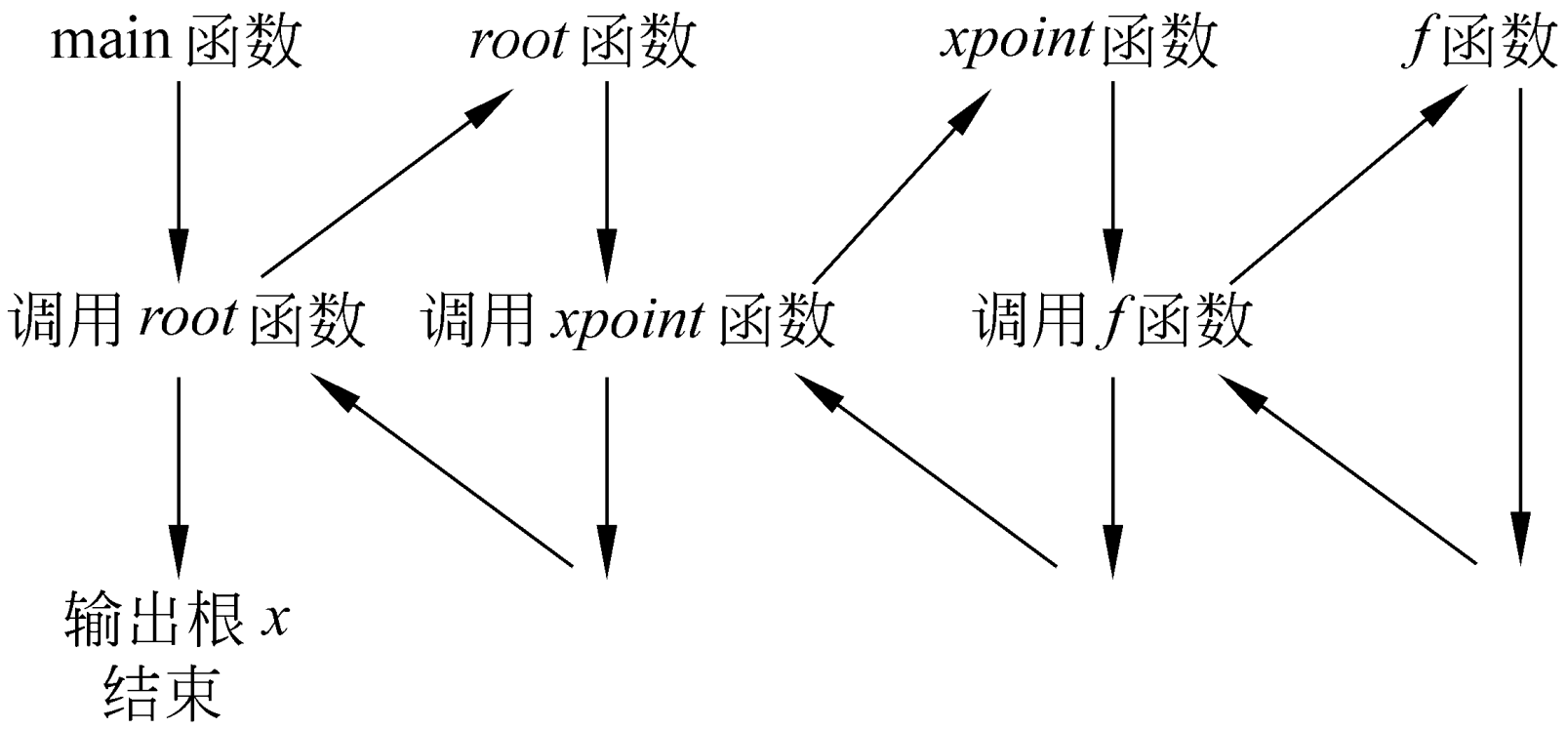


图4.8

(4) 在**root**函数中要用到求绝对值的函数**fabs**，它是对双精度数求绝对值的系统函数。它属于数学函数库，故在文件开头用**#include <cmath>**把有关的头文件包含进来。

4.10 函数的递归调用

在调用一个函数的过程中又出现直接或间接地调用该函数本身，称为函数的递归(recursive)调用。

C++允许函数的递归调用。例如：

```
int f(int x)
{int y,z;
z=f(y);              //在调用函数f的过程中，又要调用f函数
return (2* z );
}
```

以上是直接调用本函数，见图4.9。

图4.10表示的是间接调用本函数。在调用f1函数过程中要调用f2函数，而在调用f2函数过程中又要调用f1函数。

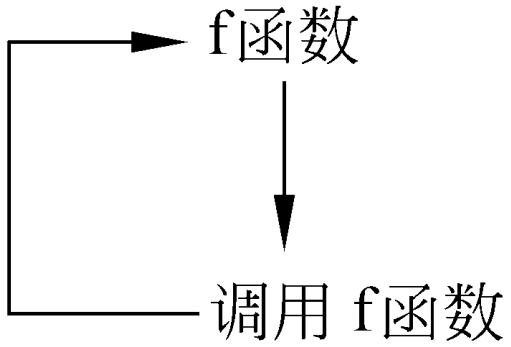


图4.9

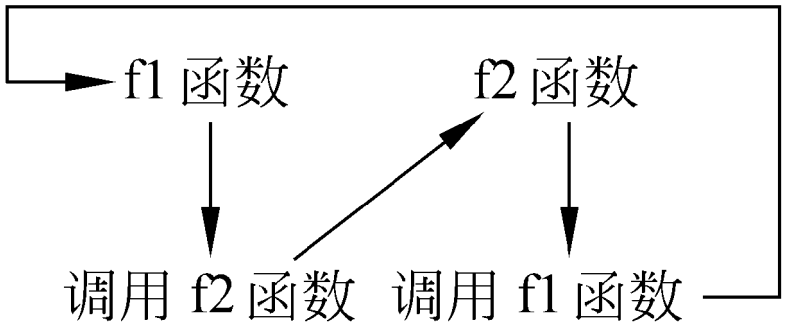


图4.10

从图上可以看到，这两种递归调用都是无终止的自身调用。显然，程序中不应出现这种无终止的递归调用，而只应出现有限次数的、有终止的递归调用，这可以用**if**语句来控制，只有在某一条件成立时才继续执行递归调用，否则就不再继续。

包含递归调用的函数称为递归函数。

例4.10 有5个人坐在一起，问第5个人多少岁？他说比第4个人大两岁。问第4个人岁数，他说比第3个人大两岁。问第3个人，又说比第2个人大两岁。问第2个人，说比第1个人大两岁。最后问第1个人，他说是10岁。请问第5个人多大？

每一个人的年龄都比其前1个人的年龄大两岁。即

age(5)=age(4)+2

age(4)=age(3)+2

age(3)=age(2)+2

age(2)=age(1)+2

age(1)=10

可以用式子表述如下：

age(n)=10　　(n=1)

age(n)=age(n-1)+2　　(n>1)

可以看到，当**n>1**时，求第**n**个人的年龄的公式是相同的。因此可以用一个函数表示上述关系。图**4.11**表示求第**5**个人年龄的过程。

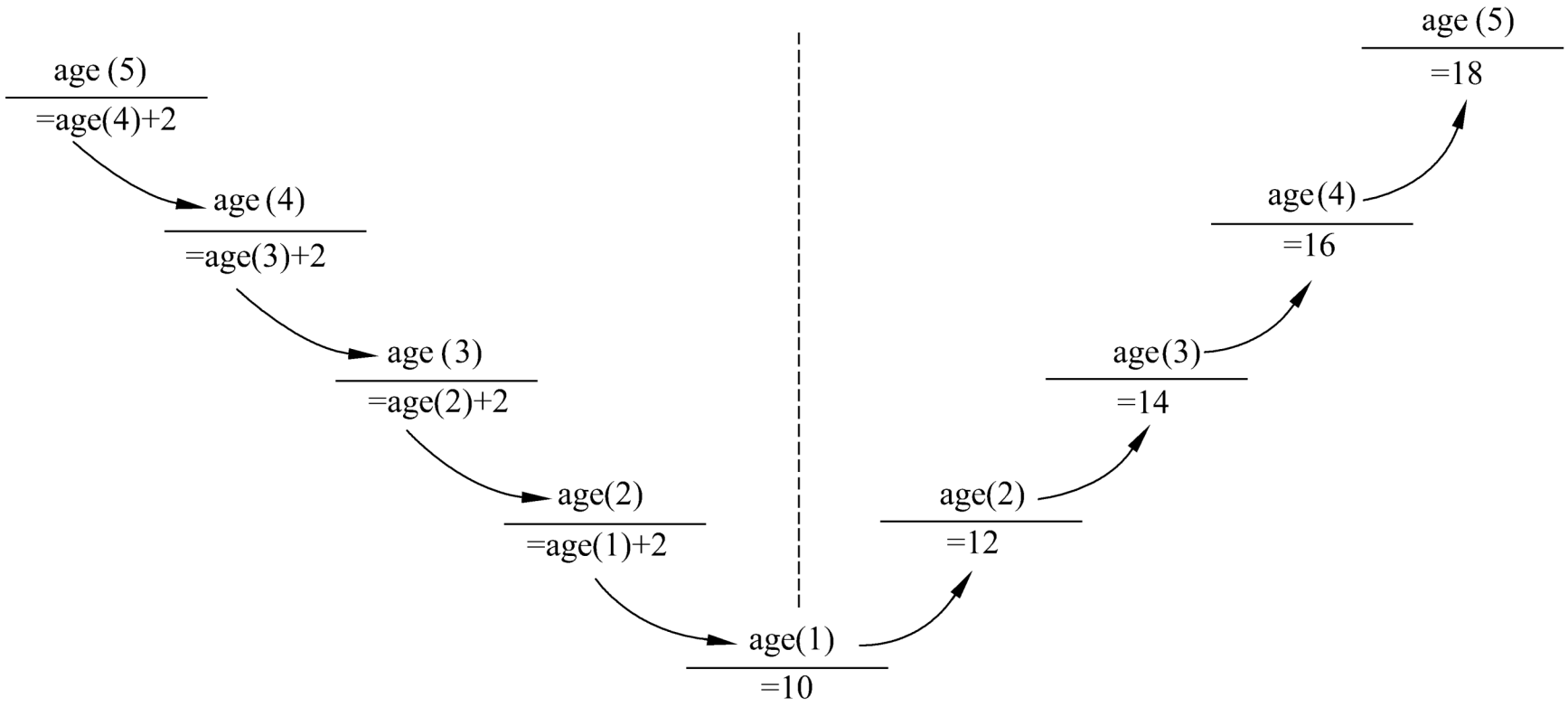


图**4.11**

可以写出以下C++程序，其中的age函数用来实现上述递归过程。

```
#include <iostream>
using namespace std;
int age(int);                    //函数声明
int main( )                      //主函数
 { cout<<age(5)<<endl;
   return 0;
 }

int age(int n)                   //求年龄的递归函数
{int c;                          //用c作为存放年龄的变量
 if(n==1) c=10;                  //当n=1时，年龄为10
 else c=age(n-1)+2;              //当n＞1时，此人年龄是他前一个人的年龄加2
 return c;                       //将年龄值带回主函数
}
```

运行结果如下：

18

函数调用过程如图**4.12**所示。

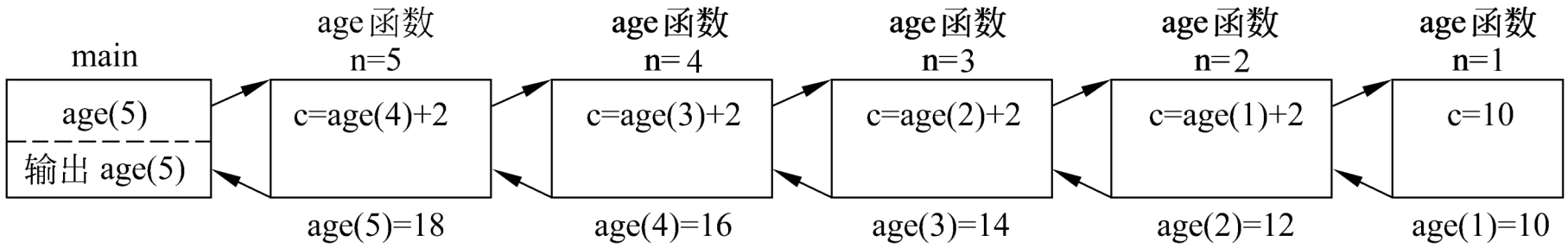


图**4.12**

例4.11 用递归方法求ｎ！。

求n!可以用递推方法，即从1开始，乘2，再乘3……一直乘到n。

求n!也可以用递归方法，即5!=4!×5，而4!＝3!×4,…,1!=１。可用下面的递归公式表示：

$$n! = \begin{cases} 1 & (n=0,1) \\ n\cdot(n-1)! & (n>1) \end{cases}$$

有了例4.10的基础，很容易写出本题的程序：

```
#include <iostream>
using namespace std;
long fac(int);                    //函数声明
int main( )
 {int n;                          //n为需要求阶乘的整数
long y;                           //y为存放n!的变量
```

```
cout<<"please input an integer : ";     //输入的提示
cin>>n;                                 //输入n
 y=fac(n);                              //调用fac函数以求n!
 cout<<n<<"!="<<y<<endl;                //输出n!的值
 return 0;
}

long fac(int n)                         //递归函数
{long f;
 if(n<0)
  {cout<<"n<0,data error!"<<endl;       //如果输入负数，报错并以-1作为
返回值
   f=-1;}
 else if (n==0||n==1) f=1;              //0!和1!的值为1
else f=fac(n-1)*n;                      //n>1时，进行递归调用
 return f;                              //将f的值作为函数值返回
}
```

运行情况如下：

please input an integer：10↙

10!=3628800

许多问题既可以用递归方法来处理，也可以用非递归方法来处理。在实现递归时，在时间和空间上的开销比较大，但符合人们的思路，程序容易理解。

4.11 局部变量和全局变量

4.11.1 局部变量

在一个函数内部定义的变量是内部变量，它只在本函数范围内有效，也就是说只有在本函数内才能使用它们，在此函数以外是不能使用这些变量的。同样，在复合语句中定义的变量只在本复合语句范围内有效。这称为局部变量**(local variable)**。如

```
float f1(int a)                                    //函数f1
{
int b,c;          b、c有效          a有效
 ┆
}
char f2(int  x, int y)                             //函数f2
{int i,j;         i、j有效          x、y有效
 ┆
}
int main( )                                        //主函数
{int m,n;
 ┆
{int p,q;          p、q在复合语句中有效  m、n有效
 ┆
}
}
```

说明：

(1) 主函数**main**中定义的变量**(m,n)**也只在主函数中有效,不会因为在主函数中定义而在整个文件或程序中有效。主函数也不能使用其他函数中定义的变量。

(2) 不同函数中可以使用同名的变量,它们代表不同的对象,互不干扰。例如,在**f1**函数中定义了变量**b**和**c**,倘若在**f2**函数中也定义变量**b**和**c**,它们在内存中占不同的单元,不会混淆。

(3) 可以在一个函数内的复合语句中定义变量,这些变量只在本复合语句中有效,这种复合语句也称为分程序或程序块。

(4) 形式参数也是局部变量。例如**f1**函数中的形参**a**也只在**f1**函数中有效。其他函数不能调用。

(5) 在函数声明中出现的参数名，其作用范围只在本行的括号内。实际上，编译系统对函数声明中的变量名是忽略的，即使在调用函数时也没有为它们分配存储单元。例如

```
int max(int a,int b);              //函数声明中出现a、b
    ┆
int max(int x,int y)               //函数定义，形参是x、y
{ cout<<x<<y<<endl;                  //合法，x、y在函数体中有效
cout<<a<<b<<endl;                  //非法，a、b在函数体中无效
}
```

编译时认为**max**函数体中的**a**和**b**未经定义。

4.11.2 全局变量

前面已介绍,程序的编译单位是源程序文件,一个源文件可以包含一个或若干个函数。在函数内定义的变量是局部变量,而在函数之外定义的变量是外部变量，称为全局变量(**global variable,**也称全程变量)。全局变量的有效范围为从定义变量的位置开始到本源文件结束。如

```
int p=1,q=5;//全局变量
float f1(a)//定义函数f1
int a;
{int b,c;
 ┆
}
char c1,c2;              //全局变量
char f2 (int x, int y)   //定义函数f2
{int i,j;
 ┆
}
main ( )//主函数
{int m,n;
 ┆
}
```

全局变量c1、c2的作用范围

全局变量p、q的作用范围

p、**q**、**c1**、**c2**都是全局变量，但它们的作用范围不同，在**main**函数和**f2**函数中可以使用全局变量**p**、**q**、**c1**、**c2**，但在函数**f1**中只能使用全局变量**p**、**q**，而不能使用**c1**和**c2**。

在一个函数中既可以使用本函数中的局部变量，又可以使用有效的全局变量。

说明：

(1) 设全局变量的作用是增加函数间数据联系的渠道。

(2) 建议不在必要时不要使用全局变量，因为：

① 全局变量在程序的全部执行过程中都占用存储单元，而不是仅在需要时才开辟单元。

② 它使函数的通用性降低了，因为在执行函数时要受到外部变量的影响。如果将一个函数移到另一个文件中，还要将有关的外部变量及其值一起移过去。但若该外部变量与其他文件的变量同名，就会出现问题，降低了程序的可靠性和通用性。在程序设计中，在划分模块时要求模块的内聚性强、与其他模块的耦合性弱。即模块的功能要单一（不要把许多互不相干的功能放到一个模块中），与其他模块的相互影响要尽量少，而用全局变量是不符合这个原则的。

一般要求把程序中的函数做成一个封闭体，除了可以通过“实参——形参”的渠道与外界发生联系外，没有其他渠道。这样的程序移植性好，可读性强。

③ 使用全局变量过多，会降低程序的清晰性。在各个函数执行时都可能改变全局变量的值，程序容易出错。因此，要限制使用全局变量。

(3) 如果在同一个源文件中，全局变量与局部变量同名，则在局部变量的作用范围内，全局变量被屏蔽，即它不起作用。

变量的有效范围称为变量的作用域**(scope)**。归纳起来，变量有**4**种不同的作用域、文件作用域**(file scope)**、函数作用域**(function scope)**、块作用域**(block scope)**和函数原型作用域**(function prototype scope)**。文件作用域是全局的，其他三者是局部的。

除了变量之外，任何以标识符代表的实体都有作用域，概念与变量的作用域相似。

4.12 变量的存储类别

4.12.1 动态存储方式与静态存储方式

上一节已介绍了变量的一种属性——作用域，作用域是从空间的角度来分析的，分为全局变量和局部变量。

变量还有另一种属性——存储期(**storage duration**，也称生命期)。存储期是指变量在内存中的存在期间。这是从变量值存在的时间角度来分析的。存储期可以分为静态存储期(**static storage duration**)和动态存储期(**dynamic storage duration**)。这是由变量的静态存储方式和动态存储方式决定的。

所谓静态存储方式是指在程序运行期间，系统对变量分配固定的存储空间。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。

先看一下内存中的供用户使用的存储空间的情况。这个存储空间可以分为三部分，即：

(1) 程序区

(2) 静态存储区

(3) 动态存储区

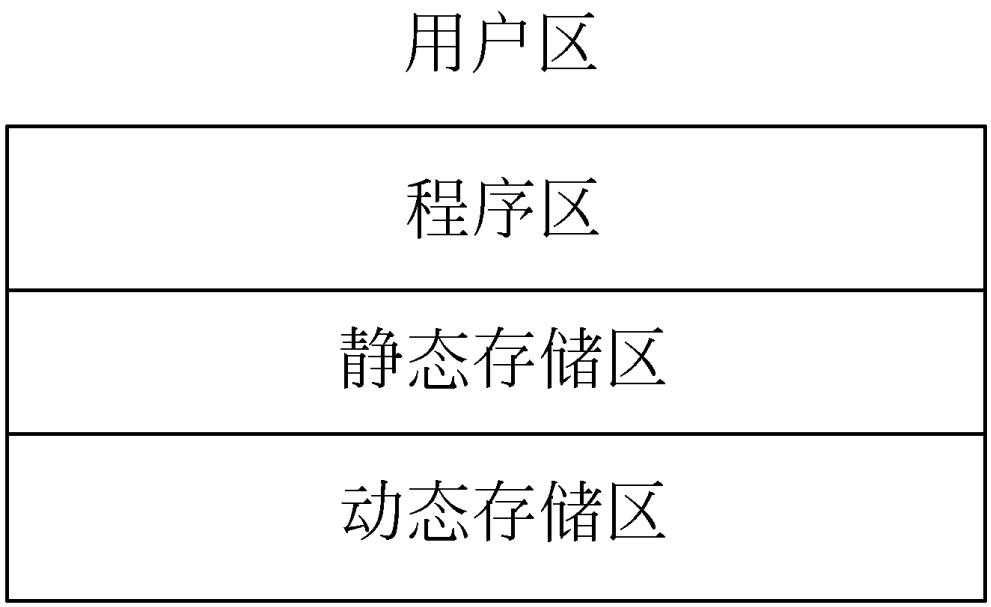


图**4.13**

数据分别存放在静态存储区和动态存储区中。全局变量全部存放在静态存储区中，在程序开始执行时给全局变量分配存储单元，程序执行完毕就释放这些空间。在程序执行过程中它们占据固定的存储单元，而不是动态地进行分配和释放。

在动态存储区中存放以下数据：①函数形式参数。在调用函数时给形参分配存储空间。②函数中的自动变量（未加**static**声明的局部变量，详见后面的介绍）。③函数调用时的现场保护和返回地址等。

对以上这些数据，在函数调用开始时分配动态存储空间，函数结束时释放这些空间。在程序执行过程中，这种分配和释放是动态的，如果在一个程序中两次调用同一函数，则要进行两次分配和释放，而两次分配给此函数中局部变量的存储空间地址可能是不相同的。

如果在一个程序中包含若干个函数，每个函数中的局部变量的存储期并不等于整个程序的执行周期，它只是整个程序执行周期的一部分。根据函数调用的情况，系统对局部变量动态地分配和释放存储空间。

在**C++**中变量除了有数据类型的属性之外，还有存储类别**(storage class)** 的属性。存储类别指的是数据在内存中存储的方法。存储方法分为静态存储和动态存储两大类。具体包含**4**种：自动的**(auto)**、静态的**(static)**、寄存器的**(register)**和外部的**(extern)**。根据变量的存储类别，可以知道变量的作用域和存储期。

4.12.2 自动变量

函数中的局部变量，如果不用关键字**static**加以声明，编译系统对它们是动态地分配存储空间的。函数的形参和在函数中定义的变量(包括在复合语句中定义的变量)都属此类。在调用该函数时，系统给形参和函数中定义的变量分配存储空间，数据存储在动态存储区中。在函数调用结束时就自动释放这些空间。如果是在复合语句中定义的变量，则在变量定义时分配存储空间，在复合语句结束时自动释放空间。因此这类局部变量称为自动变量**(auto variable)**。自动变量用关键字**auto**作存储类别的声明。例如：

```
int f(int a)                    //定义f函数， a 为形参
{auto int b,c=3;                //定义 b 和 c 为整型的自动变量
 ┆
}
```

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**，则系统把它默认为自动存储类别，它属于动态存储方式。程序中大多数变量属于自动变量。本书前面各章所介绍的例子中，在函数中定义的变量都没有声明为**auto**，其实都默认指定为自动变量。在函数体中以下两种写法作用相同：

① **auto int b,c=3;**

② **int b,c=3;**

4.12.3 用static声明静态局部变量

有时希望函数中的局部变量的值在函数调用结束后不消失而保留原值，即其占用的存储单元不释放，在下一次该函数调用时，该变量保留上一次函数调用结束时的值。这时就应该指定该局部变量为静态局部变量**(static local variable)**。

例**4.12** 静态局部变量的值。

```
#include <iostream>
using namespace std;
int f(int a)                          //定义f函数，a为形参
 {auto int  b=0;                       //定义b为自动变量
  static int c=3;                      //定义c为静态局部变量
  b=b+1;
  c=c+1;
  return a+b+c;
 }

 int main( )
 {int a=2,i;
  for(i=0;i<3;i++)
    cout<<f(a)<<" ";
  cout<<endl;
  return 0;
 }
```

运行结果为

7 8 9

先后**3**次调用**f**函数时，**b**和**c**的值如书中表**4.1**所示。

	b	c
第一次调用开始	0	3
第一次调用结束	1	4
第二次调用开始	0	4

图4.14

对静态局部变量的说明：

(1) 静态局部变量在静态存储区内分配存储单元。在程序整个运行期间都不释放。而自动变量（即动态局部变量）属于动态存储类别，存储在动态存储区空间(而不是静态存储区空间)，函数调用结束后即释放。

(2) 为静态局部变量赋初值是在编译时进行值的，即只赋初值一次，在程序运行时它已有初值。以后每次调用函数时不再重新赋初值而只是保留上次函数调用结束时的值。而为自动变量赋初值，不是在编译时进行的，而是在函数调用时进行，每调用一次函数重新给一次初值，相当于执行一次赋值语句。

(3) 如果在定义局部变量时不赋初值的话，对静态局部变量来说，编译时自动赋初值**0**(对数值型变量)或空字符(对字符型变量)。而对自动变量来说，如果不赋初值，则它的值是一个不确定的值。这是由于每次函数调用结束后存储单元已释放，下次调用时又重新另分配存储单元，而所分配的单元中的值是不确定的。

(4) 虽然静态局部变量在函数调用结束后仍然存在，但其他函数是不能引用它的，也就是说，在其他函数中它是“不可见”的。

在什么情况下需要用局部静态变量呢？

(1) 需要保留函数上一次调用结束时的值。例如可以用下例中的方法求ｎ！。

例4.13 输出1～5的阶乘值(即1！,2！,3！,4！,5！)。

```
#include <iostream>
using namespace std;
int fac(int);                    //函数声明
int main( )
 {int i;
  for(i=1;i<=5;i++)
   cout<<i<<"!="<<fac(i)<<endl;
return 0;
 }

int fac(int n)
 {static int f=1;                  //f为静态局部变量，函数结束时f的值不释放
  f=f*n;                          //在f原值基础上乘以n
  return f;
 }
```

运行结果为

1!=1

2!=2

3!=6

4!=24

5!=120

每次调用**fac(i)**，就输出一个**i**，同时保留这个**i**！的值，以便下次再乘**(i+1)**。

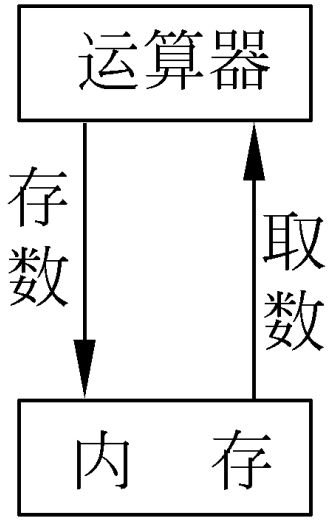
(2) 如果初始化后，变量只被引用而不改变其值，则这时用静态局部变量比较方便，以免每次调用时重新赋值。

但是应该看到，用静态存储要多占内存，而且降低了程序的可读性，当调用次数多时往往弄不清静态局部变量的当前值是什么。因此，如不必要，不要多用静态局部变量。

4.12.4 用register声明寄存器变量

一般情况下，变量的值是存放在内存中的。当程序中用到哪一个变量的值时，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。经过运算器进行运算，如果需要存数，再从运算器将数据送到内存存放。如图4.15所示。

图4.15



为提高执行效率，C++允许将局部变量的值放在CPU中的寄存器中，需要用时直接从寄存器取出参加运算，不必再到内存中去存取。这种变量叫做寄存器变量，用关键字register作声明。例如，可以将例4.14中的fac函数改写如下：

```
int fac(int n)
{register int i,f=1;          //定义i和f是寄存器变量
 for(i=1;i<=n;i++) f=f*i;
 return f;
}
```

定义f和i是存放在寄存器的局部变量，如果n的值大，则能节约许多执行时间。

在程序中定义寄存器变量对编译系统只是建议性(而不是强制性)的。当今的优化编译系统能够识别使用频繁的变量，自动地将这些变量放在寄存器中。

4.12.5 用extern声明外部变量

全局变量(外部变量)是在函数的外部定义的，它的作用域为从变量的定义处开始，到本程序文件的末尾。在此作用域内，全局变量可以为本文件中各个函数所引用。编译时将全局变量分配在静态存储区。

有时需要用**extern**来声明全局变量，以扩展全局变量的作用域。

1. 在一个文件内声明全局变量

如果外部变量不在文件的开头定义，其有效的作用范围只限于定义处到文件终了。如果在定义点之前的函数想引用该全局变量，则应该在引用之前用关键字**extern**对该变量作外部变量声明，表示该变量是一个将在下面定义的全局变量。有了此声明，就可以从声明处起，合法地引用该全局变量，这种声明称为提前引用声明。

例**4.14** 用**extern**对外部变量作提前引用声明，以扩展程序文件中的作用域。

```
#include <iostream>
using namespace std;
int max(int,int);              //函数声明
void main( )
 {extern int a,b;              //对全局变量a,b作提前引用声明
  cout<<max(a,b)<<endl;
 }
int a=15,b=-7;                 //定义全局变量a,b
int max(int x,int y)
 {int z;
  z=x>y?x：  y;
  return z;
 }
```

运行结果如下：

15

在**main**后面定义了全局变量**a**，**b**，但由于全局变量定义的位置在函数**main**之后，因此如果没有程序的第**5**行，在**main**函数中是不能引用全局变量**a**和**b**的。现在我们在**main**函数第**2**行用**extern**对**a**和**b**作了提前引用声明，表示**a**和**b**是将在后面定义的变量。这样在**main**函数中就可以合法地使用全局变量**a**和**b**了。如果不作**extern**声明，编译时会出错，系统认为**a**和**b**未经定义。一般都把全局变量的定义放在引用它的所有函数之前，这样可以避免在函数中多加一个**extern**声明。

2. 在多文件的程序中声明外部变量

如果一个程序包含两个文件，在两个文件中都要用到同一个外部变量**num**，不能分别在两个文件中各自定义一个外部变量**num**。正确的做法是：在任一个文件中定义外部变量**num**，而在另一文件中用**extern**对**num**作外部变量声明。即

extern int num;

编译系统由此知道**num**是一个已在别处定义的外部变量，它先在本文件中找有无外部变量**num**，如果有，则将其作用域扩展到本行开始(如上节所述)，如果本文件中无此外部变量，则在程序连接时从其他文件中找有无外部变量**num**，如果有，则把在另一文件中定义的外部变量**num**的作用域扩展到本文件，在本文件中可以合法地引用该外部变量**num**。

分析下例：

file1.cpp

```
extern int a,b;
int main( )
{cout<<a<<","<<b<<endl;
 return 0;
}
```

file2.cpp

```
int a=3,b=4;
```

用**extern**扩展全局变量的作用域，虽然能为程序设计带来方便，但应十分慎重，因为在执行一个文件中的函数时，可能会改变了该全局变量的值，从而会影响到另一文件中的函数执行结果。

4.12.6 用static声明静态外部变量

有时在程序设计中希望某些外部变量只限于被本文件引用，而不能被其他文件引用。这时可以在定义外部变量时加一个**static**声明。例如：

file1.cpp

```
static int a=3;
int main ( )
{
 ┆

}
```

file2.cpp

```
extern int a;
int fun (int n)
 { ┆
 a=a*n;
   ┆
 }
```

这种加上**static**声明、只能用于本文件的外部变量（全局变量）称为静态外部变量。这就为程序的模块化、通用性提供了方便。如果已知道其他文件不需要引用本文件的全局变量，可以对本文件中的全局变量都加上**static**，成为静态外部变量，以免被其他文件误用。

需要指出，不要误认为用**static**声明的外部变量才采用静态存储方式（存放在静态存储区中），而不加**static**的是动态存储（存放在动态存储区）。实际上，两种形式的外部变量都用静态存储方式，只是作用范围不同而已，都是在编译时分配内存的。

4.13 变量属性小结

一个变量除了数据类型以外，还有3种属性：

(1) 存储类别 C++允许使用auto,static,register和extern 4种存储类别。

(2) 作用域 指程序中可以引用该变量的区域。

(3) 存储期 指变量在内存的存储期限。

以上3种属性是有联系的，程序设计者只能声明变量的存储类别，通过存储类别可以确定变量的作用域和存储期。

要注意存储类别的用法。auto, static和register 3种存储类别只能用于变量的定义语句中，如

auto char c; //字符型自动变量，在函数内定义

static int a; //静态局部整型变量或静态外部整型变量

register int d; //整型寄存器变量，在函数内定义

extern int b; //声明一个已定义的外部整型变量

说明： **extern**只能用来声明已定义的外部变量，而不能用于变量的定义。只要看到**extern**，就可以判定这是变量声明，而不是定义变量的语句。

下面从不同角度分析它们之间的联系。

(1) 从作用域角度分，有局部变量和全局变量。它们采用的存储类别如下：

● 局部变量

自动变量,即动态局部变量(离开函数,值就消失)

静态局部变量(离开函数,值仍保留)

寄存器变量(离开函数,值就消失)

形式参数(可以定义为自动变量或寄存器变量)

● 全局变量

静态外部变量(只限本文件引用)

外部变量(即非静态的外部变量,允许其他文件引用)

(2) 从变量存储期(存在的时间)来区分,有动态存储和静态存储两种类型。静态存储是程序整个运行时间都存在,而动态存储则是在调用函数时临时分配单元。

●动态存储

自动变量(本函数内有效)

寄存器变量(本函数内有效)

形式参数

● 静态存储

静态局部变量(函数内有效)

静态外部变量(本文件内有效)

外部变量(其他文件可引用)

(3) 从变量值存放的位置来区分,可分为

- 内存中静态存储区

静态局部变量

静态外部变量(函数外部静态变量)

外部变量(可为其他文件引用)

- 内存中动态存储区： 自动变量和形式参数
- **CPU** 中的寄存器： 寄存器变量

(4) 关于作用域和存储期的概念。从前面叙述可以知道，对一个变量的性质可以从两个方面分析，一是从变量的作用域，一是从变量值存在时间的长短，即存储期。前者是从空间的角度，后者是从时间的角度。二者有联系但不是同一回事。图**4.16**是作用域的示意图，图**4.17**是存储期的示意图。

```
文件 file1.cpp
int a;
main 函数
{ ⋮
f2( );
 ⋮
f1( );
}
f1 函数
{auto int b;
  ⋮
 f2( );
  ⋮
}
f2 函数
{static int c;
  ⋮
}
```

a作用域

b作用域

c作用域

图4.16

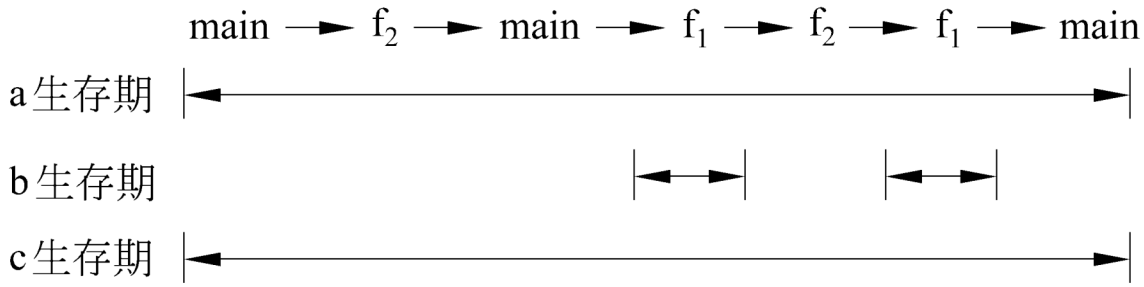


图4.17

如果一个变量在某个文件或函数范围内是有效的，则称该文件或函数为该变量的作用域，在此作用域内可以引用该变量，所以又称变量在此作用域内“可见”，这种性质又称为变量的可见性，例如图**4.16**中变量**a**、**b**在函数**f1**中可见。

如果一个变量值在某一时刻是存在的，则认为这一时刻属于该变量的存储期，或称该变量在此时刻“存在”。书中表**4.** 2 表示各种类型变量的作用域和存在性的情况。

可以看到自动变量和寄存器变量在函数内的可见性和存在性是一致的。在函数外的可见性和存在性也是一致的。静态局部变量在函数外的可见性和存在性不一致。静态外部变量和外部变量的可见性和存在性是一致的。

(5) static声明使变量采用静态存储方式，但它对局部变量和全局变量所起的作用不同。对局部变量来说,**static**使变量由动态存储方式改变为静态存储方式。而对全局变量来说,它使变量局部化(局部于本文件),但仍为静态存储方式。从作用域角度看,凡有**static**声明的,其作用域都是局限的,或者局限于本函数内(静态局部变量),或者局限于本文件内(静态外部变量)。

4.14 关于变量的声明和定义

由第2章已经知道，一个函数一般由两部分组成：(1)声明部分；(2)执行语句。声明部分的作用是对有关的标识符（如变量、函数、结构体、共用体等）的属性进行说明。对于函数，声明和定义的区别是明显的，在本章4.4.3节中已说明，函数的声明是函数的原型，而函数的定义是函数功能的确立。对函数的声明是可以放在声明部分中的，而函数的定义显然不在函数的声明部分范围内，它是一个文件中的独立模块。

对变量而言，声明与定义的关系稍微复杂一些。在声明部分出现的变量有两种情况：一种是需要建立存储空间的(如**int a;)**；另一种是不需要建立存储空间的（如**extern int a**；）。前者称为定义性声明**(defining declaration)**，或简称为定义（**definition**）。后者称为引用性声明**(referenceing declaration)**。广义地说，声明包括定义，但并非所有的声明都是定义。对**“int a;”** 而言，它是定义性声明，既可说是声明，又可说是定义。而对**“extern int a;”** 而言，它是声明而不是定义。一般为了叙述方便，把建立存储空间的声明称为定义，而把不需要建立存储空间的声明称为声明。显然这里指的声明是狭义的，即非定义性声明。例如：

```
int main( )
 {extern int a;        //这是声明不是定义。声明a是一个已定义的外部变量
…
}
int a;                 //是定义，定义a为整型外部变量
```

外部变量定义和外部变量声明的含义是不同的。外部变量的定义只能有一次，它的位置在所有函数之外，而同一文件中的外部变量的声明可以有多次，它的位置可以在函数之内，也可以在函数之外。系统根据外部变量的定义分配存储单元。对外部变量的初始化只能在定义时进行，而不能在声明中进行。所谓声明，其作用是向编译系统发出一个信息，声明该变量是一个在后面定义的外部变量，仅仅是为了提前引用该变量而作的声明。**extern**只用作声明，而不用于定义。

用**static**来声明一个变量的作用有二： **(1)**对局部变量用**static**声明，使该变量在本函数调用结束后不释放，整个程序执行期间始终存在，使其存储期为程序的全过程。**(2)**全局变量用**static**声明，则该变量的作用域只限于本文件模块(即被声明的文件中)。

请注意，用**auto**，**register**，**static**声明变量时，是在定义变量的基础上加上这些关键字，而不能单独使用。如**“static a;”**是不合法的，应写成**“static int a;”**。

4.15 内部函数和外部函数

函数本质上是全局的,因为一个函数要被另外的函数调用,但是,也可以指定函数只能被本文件调用，而不能被其他文件调用。根据函数能否被其他源文件调用,将函数区分为内部函数和外部函数。

4.15.1 内部函数

如果一个函数只能被本文件中其他函数所调用,它称为内部函数。在定义内部函数时,在函数名和函数类型的前面加**static**。函数首部的一般格式为

static 类型标识符 函数名(形参表)

如

static int fun(int a,int b)

内部函数又称静态**(static)**函数。使用内部函数,可以使函数只局限于所在文件。如果在不同的文件中有同名的内部函数,互不干扰。通常把只能由同一文件使用的函数和外部变量放在一个文件中,在它们前面都冠以**static**使之局部化,其他文件不能引用。

4.15.2 外部函数

(1) 在定义函数时,如果在函数首部的最左端冠以关键字**extern,**则表示此函数是外部函数，可供其他文件调用。

如函数首部可以写为

extern int fun (int a, int b)

这样，函数**fun**就可以为其他文件调用。如果在定义函数时省略**extern,**则默认为外部函数。本书前面所用的函数都是外部函数。

(2) 在需要调用此函数的文件中,用**extern**声明所用的函数是外部函数。

例**4.15** 输入两个整数，要求输出其中的大者。用外部函数实现。

file1.cpp（文件１）

```
#include <iostream>
using namespace std;
int main( )
 {extern int max(int,int);  //声明在本函数中将要调用在其他文件中定义的max函数
  int a,b;
  cin>>a>>b;
  cout<<max(a,b)<<endl;
  return 0;
 }
```

file2.cpp（文件2）

```
int max(int x,int y)
{int z;
 z=x>y?x：  y;
 return z;
 }
```

运行情况如下：

7 -34↙

7

在计算机上运行一个含多文件的程序时，需要建立一个项目文件**(project file)**，在该项目文件中包含程序的各个文件。详细情况请参阅本书的配套书《**C++**程序设计习题解答与上机指导》。

通过此例可知：使用**extern**声明就能够在一个文件中调用其他文件中定义的函数，或者说把该函数的作用域扩展到本文件。**extern**声明的形式就是在函数原型基础上加关键字**extern**。由于函数在本质上是外部的，在程序中经常要调用其他文件中的外部函数，为方便编程，**C++**允许在声明函数时省写**extern**。例**4.16**程序**main**函数中的函数声明可写成

int max(int,int);

这就是我们多次用过的函数原型。由此可以进一步理解函数原型的作用。用函数原型能够把函数的作用域扩展到定义该函数的文件之外（不必使用**extern**）。只要在使用该函数的每一个文件中包含该函数的函数原型即可。函数原型通知编译系统：该函数在本文件中稍后定义，或在另一文件中定义。

利用函数原型扩展函数作用域最常见的例子是**#include**命令的应用。在**#include**命令所指定的头文件中包含有调用库函数时所需的信息。例如，在程序中需要调用**sin**函数，但三角函数并不是由用户在本文件中定义的，而是存放在数学函数库中的。按以上的介绍，必须在本文件中写出**sin**函数的原型，否则无法调用**sin**函数。**sin**函数的原型是

double sin(double x);

本来应该由程序设计者在调用库函数时先从手册中查出所用的库函数的原型，并在程序中一一写出来，但这显然是麻烦而困难的。为减少程序设计者的困难，在头文件**cmath**中包括了所有数学函数的原型和其他有关信息，用户只需用以下**#include**命令：

#include <cmath>

即可。这时，在该文件中就能合法地调用各数学库函数了。

4.16 预处理命令

可以在C++源程序中加入一些“预处理命令”(preprocessor directives)，以改进程序设计环境，提高编程效率。预处理命令是C++统一规定的，但是它不是C++语言本身的组成部分，不能直接对它们进行编译（因为编译程序不能识别它们）。

现在使用的C++编译系统都包括了预处理、编译和连接等部分，因此不少用户误认为预处理命令是C++语言的一部分，甚至以为它们是C++语句，这是不对的。必须正确区别预处理命令和C++语句，区别预处理和编译，才能正确使用预处理命令。

C++与其他高级语言的一个重要区别是可以使用预处理命令和具有预处理的功能。

C++提供的预处理功能主要有以下3种：

(1) 宏定义

(2) 文件包含

(3) 条件编译

分别用宏定义命令、文件包含命令、条件编译命令来实现。为了与一般C++语句相区别，这些命令以符号“#”开头，而且末尾不包含分号。

4.16.1 宏定义

可以用**#define**命令将一个指定的标识符（即宏名）来代表一个字符串。定义宏的作用一般是用一个短的名字代表一个长的字符串。它的一般形式为

#define 标识符 字符串

这就是已经介绍过的定义符号常量。如

#define PI 3.1415926

还可以用**#define**命令定义带参数的宏定义。其定义的一般形式为

#define 宏名(参数表) 字符串

如

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

使用的形式如下：

area=S(3,2)

用３、２分别代替宏定义中的形式参数**a**和**b**，即用**3*2**代替**S(3,2)**。因此赋值语句展开为

area=3*2;

由于**C++**增加了内置函数**(inline)**，比用带参数的宏定义更方便，因此在**C++**中基本上已不再用**#define**命令定义宏了，主要用于条件编译中。

4.16.2 “文件包含”处理

1.“文件包含”的作用

所谓“文件包含”处理是指一个源文件可以将另外一个源文件的全部内容包含进来，即将另外的文件包含到本文件之中。**C++**提供了**#include**命令用来实现“文件包含”的操作。如在**file1.cpp**中有以下**#include**命令：

```
#include "file2.cpp"
```

它的作用见图**4.18**示意。

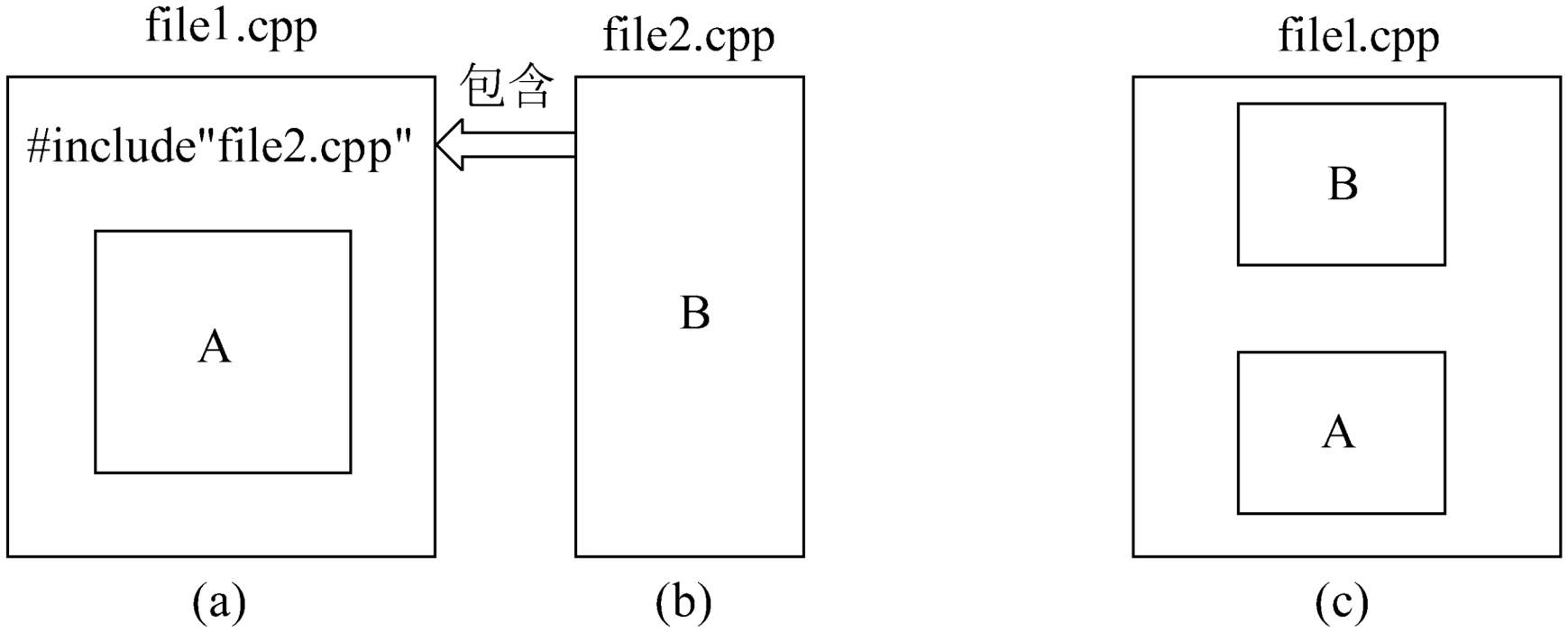


图4.18

“文件包含”命令是很有用的，它可以节省程序设计人员的重复劳动。

#include命令的应用很广泛，绝大多数**C++**程序中都包括**#include**命令。现在，库函数的开发者把这些信息写在一个文件中，用户只需将该文件“包含”进来即可(如调用数学函数的，应包含**cmath**文件)，这就大大简化了程序，写一行**#include**命令的作用相当于写几十行、几百行甚至更多行的内容。这种常用在文件头部的被包含的文件称为“标题文件”或“头部文件”。

头文件一般包含以下几类内容：

(1) 对类型的声明。

(2) 函数声明。

(3) 内置**(inline)**函数的定义。

(4) 宏定义。用**#define**定义的符号常量和用**const**声明的常变量。

(5) 全局变量定义。

(6) 外部变量声明。如**entern int a;**

(7) 还可以根据需要包含其他头文件。

不同的头文件包括以上不同的信息，提供给程序设计者使用，这样，程序设计者不需自己重复书写这些信息，只需用一行**#include**命令就把这些信息包含到本文件了，大大地提高了编程效率。由于有了**#include**命令，就把不同的文件组合在一起，形成一个文件。因此说，头文件是源文件之间的接口。

2. include命令的两种形式

在**#include**命令中，文件名除了可以用尖括号括起来以外，还可以用双撇号括起来。**#include**命令的一般形式为

#include <文件名>

或

#include "文件名"

如

#include <iostream>

或

#include "iostream"

都是合法的。二者的区别是：用尖括号时，系统到系统目录中寻找要包含的文件，如果找不到，编译系统就给出出错信息。

有时被包含的文件不一定在系统目录中，这时应该用双撇号形式，在双撇号中指出文件路径和文件名。

如果在双撇号中没有给出绝对路径，如**#include "file2.c"**则默认指用户当前目录中的文件。系统先在用户当前目录中寻找要包含的文件，若找不到，再按标准方式查找。如果程序中要包含的是用户自己编写的文件，宜用双撇号形式。

对于系统提供的头文件，既可以用尖括号形式，也可以用双撇号形式，都能找到被包含的文件，但显然用尖括号形式更直截了当，效率更高。

3. 关于C++标准库

在C++编译系统中，提供了许多系统函数和宏定义，而对函数的声明则分别存放在不同的头文件中。如果要调用某一个函数，就必须用#include命令将有关的头文件包含进来。C++的库除了保留C的大部分系统函数和宏定义外，还增加了预定义的模板和类。但是不同C++库的内容不完全相同，由各C++编译系统自行决定。不久前推出的C++标准将库的建设也纳入标准，规范化了C++标准库，以便使C++程序能够在不同的C++平台上工作，便于互相移植。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。如

```
#include <iostream.h>          //C形式的头文件
#include <iostream>            //C++形式的头文件
```

效果基本上是一样的。建议尽量用符合C++标准的形式，即在包含C++头文件时一般不用后缀。如果用户自己编写头文件，可以用.h为后缀。

4.16.3 条件编译

一般情况下，在进行编译时对源程序中的每一行都要编译。但是有时希望程序中某一部分内容只在满足一定条件时才进行编译，也就是指定对程序中的一部分内容进行编译的条件。如果不满足这个条件，就不编译这部分内容。这就是“条件编译”。

有时，希望当满足某条件时对一组语句进行编译，而当条件不满足时则编译另一组语句。

条件编译命令常用的有以下形式：

(1)

#ifdef 标识符

程序段 1

e l s e

程序段 2

#endif

它的作用是当所指定的标识符已经被**#define**命令定义过，则在程序编译阶段只编译程序段 1，否则编译程序段 2。**#endif**用来限定**#ifdef**命令的范围。其中**#else**部分也可以没有。

(2)

#if 表达式

　　程序段**1**

#else

　　程序段**2**

#endif

它的作用是当指定的表达式值为真（非零）时就编译程序段１，否则编译程序段２。可以事先给定一定条件，使程序在不同的条件下执行不同的功能。

例**4.16** 在调试程序时，常常希望输出一些所需的信息，而在调试完成后不再输出这些信息。可以在源程序中插入条件编译段。下面是一个简单的示例。

```
#include <iostream>
using namespace std;
#define RUN                          //在调试程序时使之成为注释行
int main( )
{ int x=1,y=2,z=3;
#ifndef RUN                          //本行为条件编译命令
cout<<"x="<< x <<",y="<< y <<",z="<< z ;    //在调试程序时需要输出这些信息
#endif                               //本行为条件编译命令
cout<< "x*y*z=" *y*z<<endl;
}
```

第3行用#define命令的目的不在于用RUN代表一个字符串，而只是表示已定义过RUN，因此RUN后面写什么字符串都无所谓，甚至可以不写字符串。在调试程序时去掉第3行(或在行首加//，使之成为注释行)，由于无此行，故未对RUN定义，第6行据此决定编译第7行，运行时输出x,y,z的值，以便用户分析有关变量当前的值。运行程序输出：

```
x=1,y=2,z=3
x*y*z=6
```

在调试完成后，在运行之前，加上第3行，重新编译，由于此时RUN已被定义过，则该cout语句不被编译，因此在运行时不再输出x,y,z的值。运行情况为：

```
x*y*z=6
```