西北工業大學

NORTHWESTERN POLYTECHNICAL UNIVERSITY

程序设计基础
Programming in C++

U10G13027/U10G13015

主讲：魏英，计算机学院

- 4.10 对象初始化
- 4.11 声明与定义
- 4.12 变量修饰小结
- 4.13 程序组织结构
- 4.14 函数应用程序举例

※ 我想设计一个程序完成下面的计算：

$$\frac{m!\ n!}{(m-n)!}$$ 其中，m、n为整数且m>n

该如何设计程序最有效？

- 函数定义的一般形式为：

```
返回类型  函数名(形式参数列表)
{
        函数体声明部分
        函数体执行语句
}
```

- 函数定义本质上就是函数的实现，包括：
- ①确定函数名；②确定形式参数列表；
- ③确定返回类型；④编写函数体代码。

- 1. 函数名
  - 实现函数需要确定函数名，以便使用函数时能够按名引用。

- 2. 形式参数列表
  - 形式参数列表是函数与调用者进行数据交换的途径，一般形式为：

类型1 参数名1,类型2 参数名2, 类型3 参数名3 , ...

- 多个参数用逗号（，）分隔，且每个参数都要有自己的类型说明，即使类型相同的参数也是如此。例如：

```
int fun(int x, int y, double m)
{
    return m>12.5 ? x : y;
}
```

- 函数可以没有形式参数，定义形式为：

```
返回类型  函数名()
{
        函数体声明部分
        函数体执行语句
}
```

```
返回类型  函数名(void)
{
        函数体声明部分
        函数体执行语句
}
```

3. 返回类型

- 返回类型可以是C++除数组之外的内置数据类型或自定义类型。

- 函数可以不返回数据，此时返回类型应写成void，表示没有返回值，其形式为：

```
void  函数名(形式参数列表)
{
      函数体声明部分
      函数体执行语句
}
```

- 4. 函数体
  - 函数体（function body）包含声明部分和执行语句，是一组能实现特定功能的语句序列的集合。
  - 编写函数体是为了实现函数功能。故称函数定义为函数实现，简称实现。
  - 而函数头简称接口。

例4.1

```
1 #include <iostream>
2 using namespace std;
3 int IsPrime(int m) //求素数函数
4 {   //枚举法求m是否素数
5     int i;
6     for (i=2 ; i<=m-1 ; i++)
7         if (m % i==0) return 0;
8     return 1; //是素数返回1
9 }
```

第3行为函数头，称为IsPrime函数的接口。

第4～9行为函数体，称为IsPrime函数的实现。

例4.1

```
10 int main()
11 {
12     int m;
13     cin>>m;
14     if (IsPrime(m)) cout<<"Yes"<<endl;
15     else cout<<"No"<<endl; //不是素数输出No
16     return 0;
17 }
```

1. 无返回值函数

- 当函数的返回类型是void时，表明函数无返回值。

- 没有返回值的函数，在调用处是不能按表达式来调用函数的，只能按语句形式调用函数。

- 如：m=max(a,b);　**//以表达式的方式调用max函数**
  print( );　**//以语句的形式调用print函数**

2. 有返回值函数

- 当函数的返回类型不是void，表明函数有返回值。如果函数体内没有return语句，此时函数返回的值是与返回类型相同但内容却是随机的一个值。

- 如果要用return语句结束，即return必须返回值。此时函数返回的值是与返回类型相同、由表达式计算出来的一个值。

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

 }
```

- 形式参数
- 函数定义中的形式参数列表（parameters），简称形参。例如：

```
1 int max(int a, int b)
2 {
3     return a>b ? a : b;
4 }
```

第1行a和b就是形参。

- 函数定义时指定的形参，在未进行函数调用前，并不实际占用内存中的存储单元。
- 只有在发生函数调用时，形参才分配实际的内存单元，接受从主调函数传来的数据。
- 当函数调用结束后，形参占用的内存单元被自动释放。

- 实际参数
- 函数调用时提供给被调函数的参数称为实际参数（arguments），简称实参。
- 实参必须有确定的值，因为调用函数会将它们传递给形参。实参可以是常量、变量或表达式，还可以是函数的返回值。例如：

```
x = max(a,b); //max函数调用，实参为a,b
y = max(a+3,128); //max函数调用，实参为a+3,128
z = max(max(a,b),c);
   //max函数调用，实参为max(a,b),c
```

- 实参的类型、次序和数目要与形参一致。
- 如果参数数目不一致，则出现编译错误；
- 如果参数次序不一致，则传递到被调函数中的数据就不合逻辑，难有正确的程序结果；
- 如果参数类型不一致时，则函数调用时按形参类型隐式类型转换实参；

- 程序通常有两种函数参数传递机制：
- 值传递和引用传递。
- 值传递（pass-by-value）过程中，形参作为被调函数的内部变量来处理，即开辟内存空间以存放由主调函数复制过来的实参的值，从而成为实参的一个副本。
- 值传递的特点是被调函数对形参的任何操作都是对内部变量进行，不会影响到主调函数的实参变量的值。

```
void fun(int x, int y, int m)
{
    m = x>y ? x : y; //仅修改函数内部的m
}
void caller( ) //主调函数，调用者
{
    int a=10, b=5, k=1;
    fun(a,b,k); //实参值传递
}
```

- 值传递时，实参数据传递给形参是单向传递，即只能由实参传递给形参，而不能由形参传回给实参。

1. 函数声明

- 当要调用函数时，C++语言规定在调用一个函数之前必须有该函数的声明。

```
4  double root(double x, double y) //root函数求x-y的平方根
5  {   if (x>=y) return sqrt(x-y); //只有在x大于等于y时计算x-y的平方根
6      else return 0; //否则返回0
7  }
8  int main() //主函数
9  {       double a, b; //定义两个浮点型变量
10         cin>>a>>b; //输入两个数
11         cout<<root(a,b)<<endl; //输出a-b的平方根
12         return 0; //主函数正常结束返回0
13 }
```

4.3.1 函数声明和函数原型

- 定义是函数实现，一个函数只能定义一次。
- C++规定函数的定义既是函数定义，也是函数声明。换言之，只要函数调用是写在函数定义的后面，就自然有了函数声明。
- 声明的作用是程序向编译器提供函数的接口信息，因而多次提供接口信息是允许的，但不能提供相互矛盾、语义不一致的接口信息。
- 如果被调函数写在主调函数后面，可以在调用点前面写上函数的原型声明。

- 2. 函数原型
- 函数原型（function prototype）的作用是提供函数调用所必须的接口信息，使编译器能够检查函数调用中可能存在的问题，有两种形式：
- ①第一种形式：

```
返回类型 函数名(类型1 形参1,类型2 形参2,.....);
```

- ②第二种形式：

```
返回类型 函数名(类型1 ,类型2 ,.....);
```

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

- 标准库求平方根的函数原型，表示调用它需要：
- ①包含头文件cmath，因为sqrt函数原型在cmath中。

- ②sqrt函数须提供一个double型的实参，返回值也是double型。

例4.3

```
1  #include <iostream>
2  using namespace std;
3  int gcd(int,int);
4  int main()
5  {
6      int m,n;
7      cin>>m>>n;
8      cout<<gcd(m,n)<<endl;
9      return 0;
10 }
```

例4.3

```
11 int gcd(int m, int n)
12 {   //求最大公约数，gcd函数实现在后
13     int r;
14     while (n!=0) { //欧几里德算法
15         r = m % n ;
16         m = n ;
17         n = r ;
18     }
19     return m;
20 }
```

- 编译器在编译时，无论它们哪个在前，均以第一次“看到”的函数接口为准,，如果后面的与这个函数接口不一致，就会出现编译错误，所以函数原型要与函数定义匹配。

上节课知识回顾

- 函数的定义
- 函数的返回
- 函数的参数
- 值传递

```
#include<iostream>
using namespace std;
int max(int x, int y)
{ int z;
   z=(x>y)? x:y;
   return(z);
}
int main()
{ int a,b,c;
   cin>>a>>b;
   c=max(a,b);
   cout<<"max is"<<c<<endl;
   return 0;
}
```

- 3. 函数调用
- 有了函数声明，就可以调用函数，有参数函数调用的形式为：

```
函数名(实参列表)
```

- 实参可以是常量、变量、表达式和函数调用，各实参之间用逗号（，）分隔。实参的类型、次序、个数应与形参一致。

- 无参数函数调用的形式为：

```
函数名()
```

- 函数名后面的括号（）必须有，括号内不能有任何参数。

- 在C++语言中，可以用以下几种方式调用函数。
- （1）函数表达式。

```
z = max(x,y);
```

- （2）函数调用语句。

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

- （3）函数实参

```
m=max(max(x,y),z);
```

上节课知识回顾

- 函数的定义
- 函数的返回
- 函数的参数
- 值传递
- 函数原型
- 函数调用

```
#include<iostream>
using namespace std;
int max(int x, int y)
{ int z;
   z=(x>y)? x:y;
   return (z);
}
int main()
{ int a,b,c;
   cin>>a>>b;
   c=max(a,b);
   cout<<"max is"<<c<<endl;
   return 0;
}
```

- C++提供一种提高函数效率的方法，即在编译时将被调函数的代码直接嵌入到主调函数中，取消调用这个环节。这种嵌入到主调函数中的函数称为内联函数（inline function）。

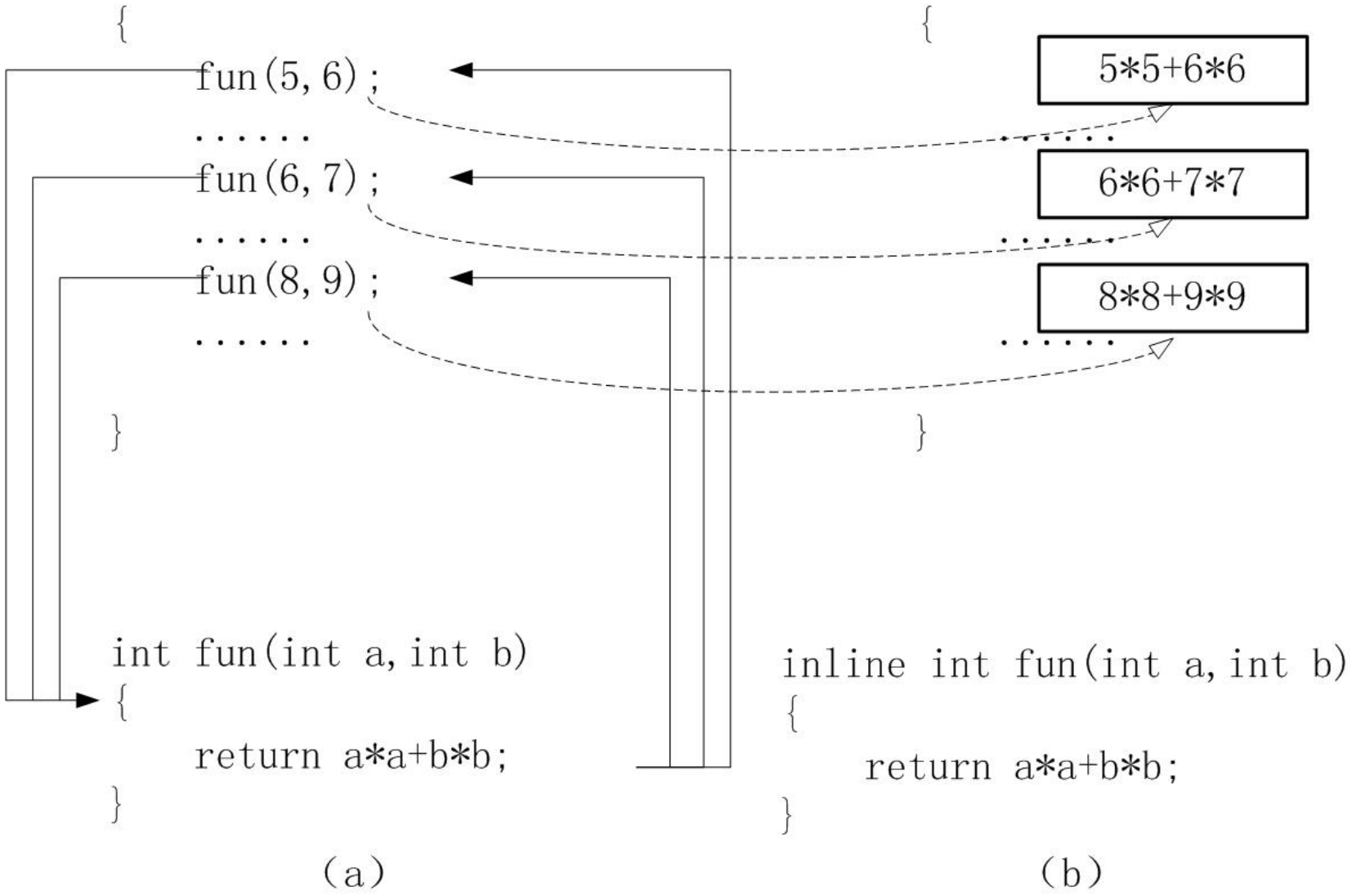
- 内联函数的声明是在函数定义的类型前加上inline修饰符，定义形式为：

```
inline 返回类型  函数名(形式参数列表)
{
        函数体
}
```

例4.6

```
1  #include <iostream>
2  using namespace std;
3  inline int fun(int a,int b) //内联函数
4  {
5      return a*a+b*b;
6  }
7  int main()
8  {
9      int n=5,m=8,k;
10     k = fun(n,m); //调用点嵌入 a*a+b*b 代码
11     cout<<"k="<<k<<endl;
12     return 0;
13 }
```

图4.2 内联函数调用示意



- 内联函数中不允许用循环语句和switch语句。
- 内联函数的声明必须出现在内联函数第一次被调用之前。

- C++允许在函数定义或函数声明时，为形参指定默认值，这样的参数称为默认参数（default argument），一般形式为：

```
返回类型 函数名(...,类型 默认参数名=默认值)
{       函数体       }
```

```
int add(int x=5,int y=6)
{
    return x+y;
}
```

```
int main()
{   add(10,20);  //10+20
    add(10);  //10+6
    add();  //5+6
    return 0;
}
```

- （1）如果在函数定义时设置了默认参数，那么就不能在函数声明中再次设置，反之亦然。

```
int volume(int L, int W , int H=1);
int volume(int L, int W , int H=1)
//定义再次设置，错误
{
    return length * width * height;
}
```

```
int volume(int L, int W , int H=1)
{
    return length * width * height;
}
int volume(int L, int W , int H=1);
//声明再次设置，错误
```

```
int volume(int L, int W , int H=1);
int volume(int L, int W , int H) // 正确
{
    return length * width * height;
}
```

- （2）可以设置多个默认参数，设置的顺序为自右向左，换言之，要为某个参数设置默认值，则它右边的所有参数必须都是默认参数。

```
int volume(int L, int W , int H=1);
//H为默认参数，正确
int volume(int L, int W=1 , int H=1);
//W,H为默认参数，正确
int volume(int L=1, int W=1 , int H=1);
//L,W,H为默认参数，正确
int volume(int L=1, int W , int H); // 错误
int volume(int L, int W=1 , int H); // 错误
```

- （3）默认值可以是常量、全局变量，甚至是一个函数调用（调用实参必须是常量或全局变量的表达式），不可以是局部变量。

```
int p1=2 , p2=10 ;
int max(int a,int b)
{
   return a>b ? a : b;
}
int volume(int L, int W , int H=1); //允许常量
int volume(int L, int W , int H=p1+p2);
//允许全局变量及表达式
int volume(int L, int W , int H=max(5,6));
//允许函数调用
int volume(int L, int W , int H=max(p1,p2))
//允许全局变量函数调用
```

- 默认参数函数的调用
- 默认参数本质上是编译器根据函数声明或函数定义时的默认参数设置，对函数调用中没有给出来的实参自动用默认值表达式“补齐”再进行编译。

例4.7

```
#include <iostream>
using namespace std;
int p1=2 , p2=10 ;
int max(int a,int b)
{
    return a>b ? a : b;
}
    int   volume(int   L=1,   int   W=p1+p2   ,   int
H=max(p1,p2))
{
    return L * W * H ;
}
```

例4.7

```
12 int main()
13 {
14     cout << "v0=" << volume() <<endl;
15     cout << "v1=" << volume(5) <<endl;
16     cout << "v2=" << volume(5,10) <<endl;
17     cout<<"v3="<<volume(5,10,15) <<endl;
18     return 0;
19 }
```

- 函数重载（function overloading）是在同一个域中用同一个函数名来定义多个函数，但函数参数列表应彼此有不同，或者是参数个数不同，或者是参数类型不同，或者两者均有不同。

```
int add(int a, int b);
double add(double a, double b);
```

形参类型不同

```
int add(int a, int b);
int add(int a, int b, int c);
```

形参个数不同

- 需要注意函数重载与重复声明是有区别的。
- （1）如果两个函数声明的返回类型和形参表完全一样，则将第二个函数声明视为第一个的重复声明；如果两个函数的形参列表完全相同，但返回类型不同，则第二个声明是错误的。

```
int max(int a,int b);
double max(int a,int b);  // 声明错误
```

- 即函数重载不能仅仅是返回类型不同。

- （2）函数原型的两种形式写在一起是重复声明。

```
int max(int a,int b);
int max(int ,int );   //重复声明
```

- （3）形参列表只有默认参数不同（默认参数没有改变形参的个数）。

```
int max(int a,int b);
int max(int a,int b=10); //重复声明
```

- （4）非引用型形参列表的区别仅仅是const限定的差异。

```
int max(int a,int b);
int max(const int a,const int b); //重复声明
```

例4.8

```
1  #include <iostream>
2  using namespace std;
3  int max(int a, int b) //整型版本
4  {
5      return (a>b ? a : b);
6  }
7  double max(double a, double b) //双精度版本
8  {
9      return (a>b ? a : b);
10 }
11 long max(long a, long b) //长整型版本
12 {
13     return (a>b ? a : b);
14 }
```

例4.8

```
15 int main()
16 {
17     int i=12 , j=-12 , k;
18     k = max(i,j) ; //调用整型版本max
19     cout << "int max=" << k << endl;
20     double x=123.4 , y=65.43 , z;
21     z = max(x,y) ; //调用双精度版本max
22     cout << "double max=" << z << endl;
23     long a=7654321 , b=1234567, c;
24     c = max(a,b) ; //调用长整型版本max
25     cout << "long max=" << c << endl;
26     return 0;
27 }
```

例4.9

```
1  #include <iostream>
2  using namespace std;
3  int max(int a, int b) //两个参数版本
4  {
5      return (a>b ? a : b);
6  }
7  int max(int a, int b, int c) //三个参数版本
8  {
9      a = a>b ? a : b ;
10     a = a>c ? a : c ;
11     return (a);
12 }
```

例4.9

```
13 int main()
14 {
15     int a , b, i=10, j=8 , k=12;
16     a = max(i,j) ; //调用两个参数版本max
17     cout << "max(i,j)=" << a << endl;
18     b = max(i,j,k) ; //调用三个参数版本max
19     cout << "max(i,j,k)=" << b << endl;
20     return 0;
21 }
```

重载函数的注意事项：

- 重载函数的形参必须不同（个数不同或类型不同）。
- 编译程序将根据实参和形参的类型及个数的最佳匹配来选择调用哪一个函数。
- 不要将不同功能的函数声明为重载函数，以免出现调用结果的误解、混淆。

```
int add(int x, int y);
int add(int a, int b);
```

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

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

```
double add(double x, double y);
{  return x-y; }
```

- 函数模板（function template）是一个独立于类型的函数，可作为一种模式，产生函数的特定类型版本。

```
int abs( int x )
{  return x<0?-x:x;  }
```

```
double abs( double x )
{  return x<0?-x:x;  }
```

- 使用函数模板可以设计通用型的函数，这些函数与类型无关并且只在需要时自动实例化，从而形成“批量型”的编程方式。

- 1. 函数模板的定义
- 函数模板定义的语法形式为：

```
template<模板形参表>返回类型 函数名(形参列表)
{
    函数体
}
```

```
template<typename T>
T abs(T x)
{  return x<0?-x:x; }
```

```
int main()
{  int n=-5;   double d=-5.5;
   cout<<abs(n)<<',<<abs(d)<<endl;
   return 0;
}
```

- 模板形参表（template parameter list）是用一对尖括号括< >括起来的一个或多个模板形参的列表，不允许为空，形参之间以逗号分隔，其形式有两种。
- ①第一种形式

typename 类型参数名1,typename 类型参数名2,.....

- ②第二种形式

class 类型参数名1, class 类型参数名2,.....

例4.11

```
1  #include <iostream>
2  using namespace std;
3  template <class T> T add(T a,T b)
4  {
5      return a+b;
6  }
7  int main()
8  {
9      cout<<"int_add="<<add(10,20)<<endl;
10     cout<<"double_add="<<add(10.2,20.5)<<endl;
11     cout<<"char_add="<<add('A','\2')<<endl;
12     cout<<"int_add="<<add(100,200)<<endl;
13     return 0;
14 }
```

- 嵌套调用
- 在调用一个函数的过程中，又调用另一个函数，称为函数的嵌套调用，C++允许函数多层嵌套调用，只要在函数调用前有函数声明即可。

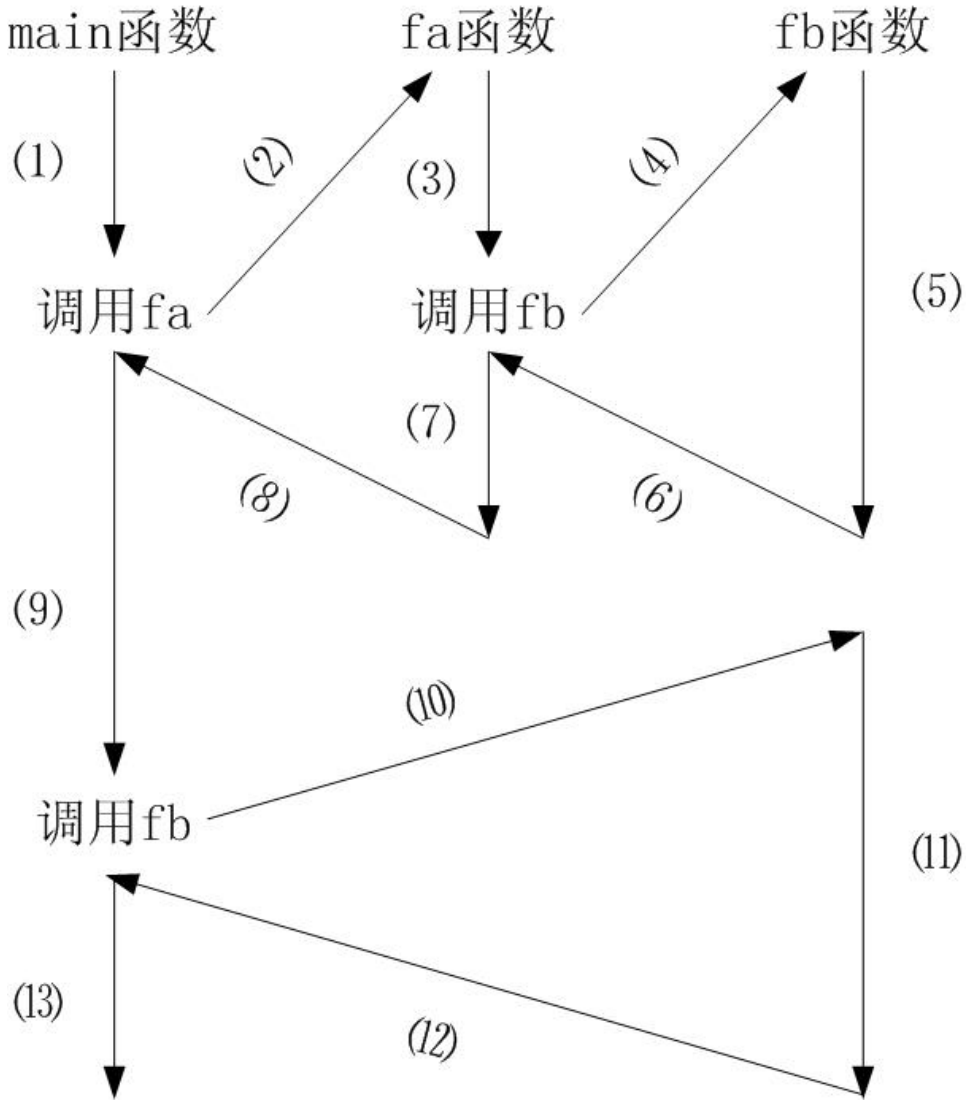
例4.12

```
1  #include <iostream>
2  using namespace std;
3  int fa(int a,int b); //fa函数原型
4  int fb(int x); //fb函数原型
5  int main()
6  {
7      int a=5,b=10,c;
8      c = fa(a,b); cout<<c<<endl;
9      c = fb(a+b); cout<<c<<endl;
10      return 0;
11 }
```

例4.12

```
12 int fa(int a,int b)
13 {
14     int z;
15     z= fb(a*b);
16     return z;
17 }
18 int fb(int x)
19 {
20     int a=15,b=20,c;
21     c=a+b+x;
22     return c;
23 }
```

图4.4 嵌套调用示意

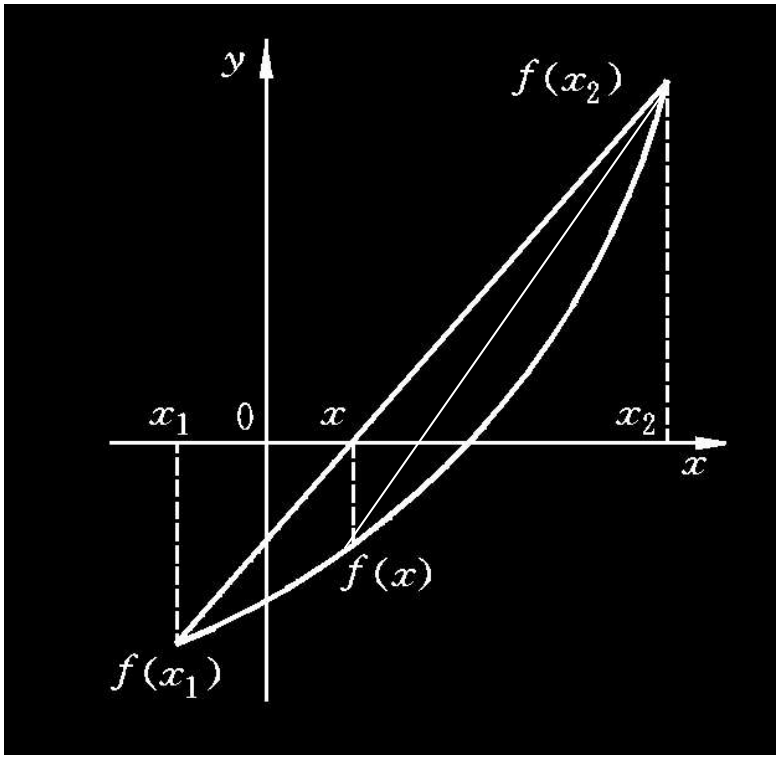


- 【例4.8】 用弦截法求

$$f(x)=x^3-5x^2+16x-80$$

- 方程的根，精度 $\varepsilon=10^{-6}$ 。

$x^3 - 5x^2 + 16x - 80 = 0$



算法分析：

对任意的 f(x)=0，

1、给定两个x1、x2，满足x1<x2且f(x1)和f(x2)的符号相反

2、过f(x1)和f(x2)两点做直线（弦）交x轴于x，其中

$$x = \frac{x1 \cdot f(x2) - x2 \cdot f(x1)}{f(x2) - f(x1)}$$

3、求f(x);若f(x)与f(x1)同符号，则根必在（x，x2）区间内，令x1=x；反之，则根必在（x1，x）区间内，令x2=x

4、重复2和3，直到f(x) <ε（ε为一个很小的数，如10^{-6}）为止。此时有f(x) ≈0

程序说明：

1、函数$f(x)=x^3-5x^2+16x-80$

2、用函数xpoint(x1,x2)求弦截点x的坐标

$$x=\frac{x1\cdot f(x2)-x2\cdot f(x1)}{f(x2)-f(x1)}$$

3、用函数root(x1,x2)求区间(x1,x2)上的根。

例4.13

```
1  #include <iostream>
2  #include <cmath>
3  using namespace std;
4  double f(double x)
5  {    //所要求解的函数公式，可改为其他公式
6       return x*x*x-5*x*x+16*x-80;
7  }
8  double xpoint(double a,double b)
9  {   //求解弦与x轴的交点
10      return (a*f(b)-b*f(a))/(f(b)-f(a));
11  }
```

例4.13

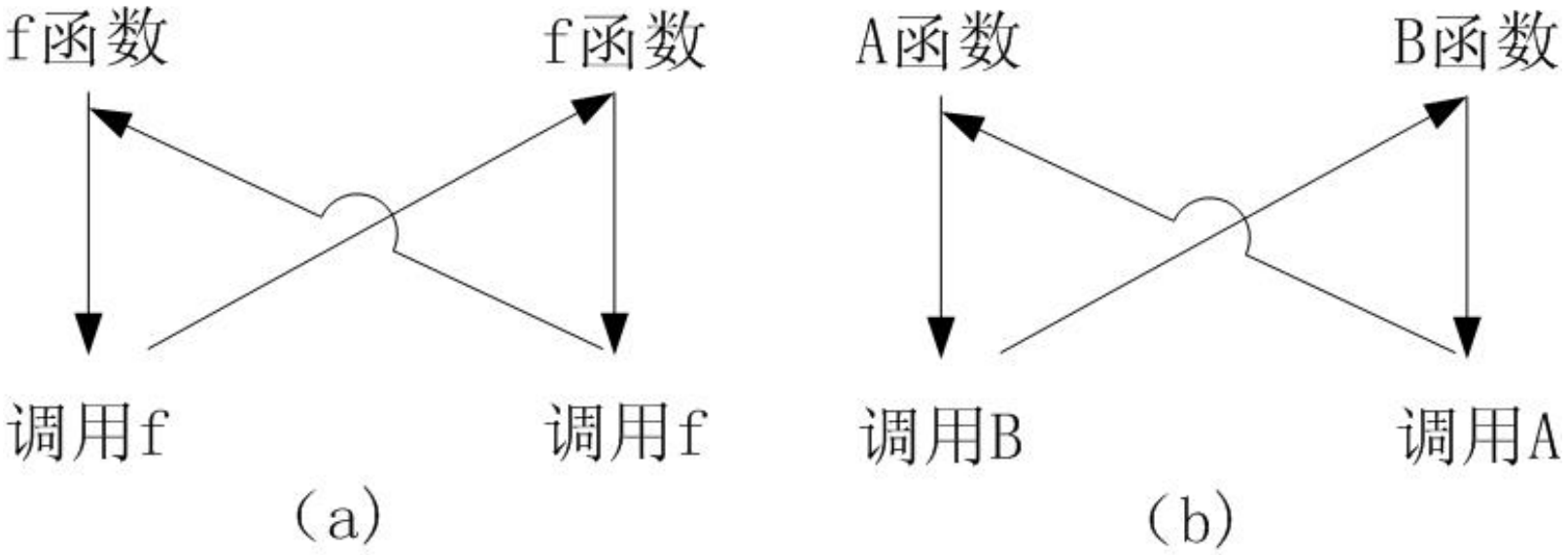
```
12 double root(double a, double b)
13 {   //弦截法求方程[a,b]区间的根
14     double x,y,y1;
15     y1=f(a);
16     do {
17         x=xpoint(a,b); //求交点x坐标
18         y=f(x);//求y
19         if (y*y1>0) y1=y, a=x;
20         else b=x;
21     } while (fabs(y)>=0.00001);//计算精度E
22     return x;
23 }
```

例4.13

```
24 int main()
25 {
26     double a,b;
27     cin>>a>>b;
28     cout<<"root="<<root(a,b)<<endl;
29     return 0;
30 }
```

- 4.5.2 递归调用
- 函数直接或间接调用自己称为递归调用。C++语言允许函数递归调用，如图4.7（a）所示为直接递归调用，如图4.7（b）所示为间接递归调用。

图4.7 递归调用示意



例4.14 编写求n的阶乘的函数

```
1  #include <iostream>
2  using namespace std;
3  int f(int n)
4  {
5      if (n>1) return f(n-1)*n; //递归调用
6      return 1;
7  }
8  int main()
9  {
10     cout<<f(5)<<endl;
11     return 0;
12 }
```

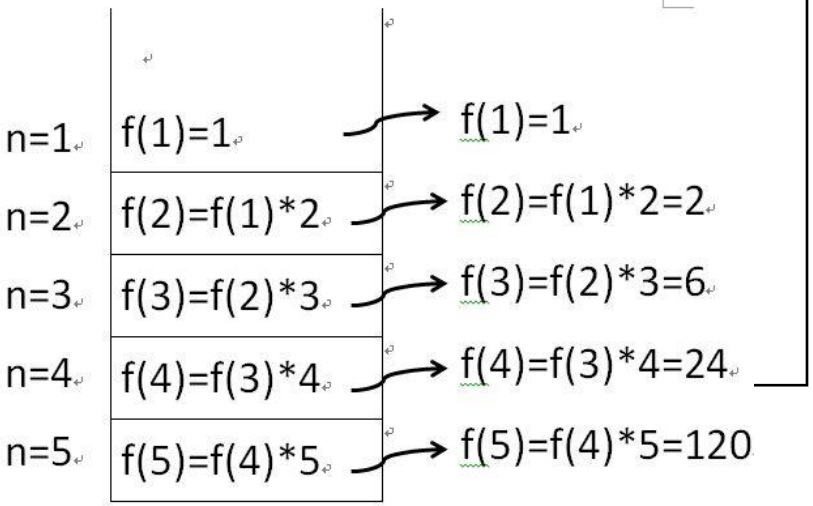
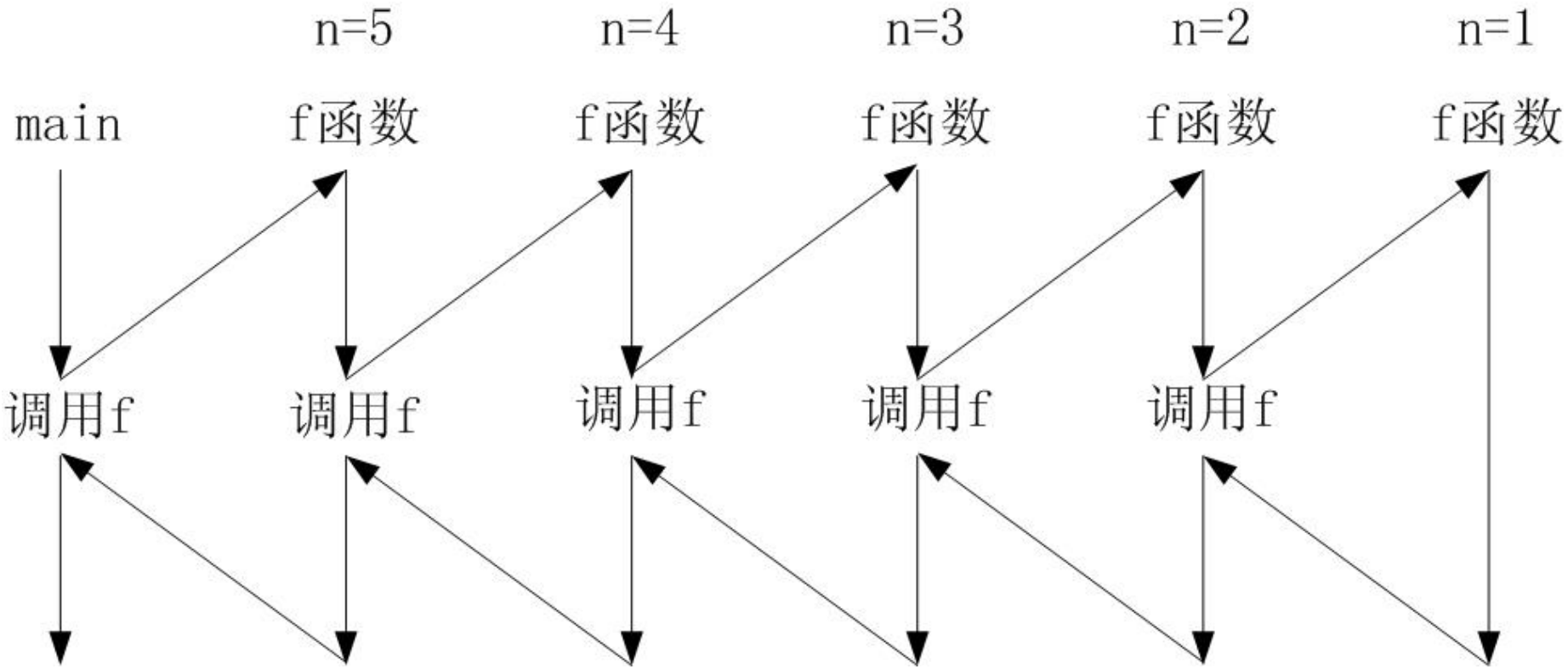


图4.8 递归调用过程



- 本节讨论（设计函数的重要内容）

- 作用域（scope）
- 生命期（lifetimes）

- 局部变量
- 在函数内部或复合语句中（简称区域）定义的变量，称为局部变量（local variable），又称为内部变量。

下列变量是局部变量：

1、在一个函数内部定义的变量

2、函数的形式参数

3、在某个复合语句中定义的变量

4.9.1 局部变量

```
1    int f1(int x,int y) // f1 函数
2    {
3        int a,b,m=100;
4        if (x>y) {
5            int a,t;
             a=100;
20       }
         ⋮
35   }
36   int main() // main 函数
37   {
38       int a=15,b=10,c,n=200;
39       c = f1(a,b);
             ⋮
```

a,t 有效

a,b,m 有效

x,y 有效

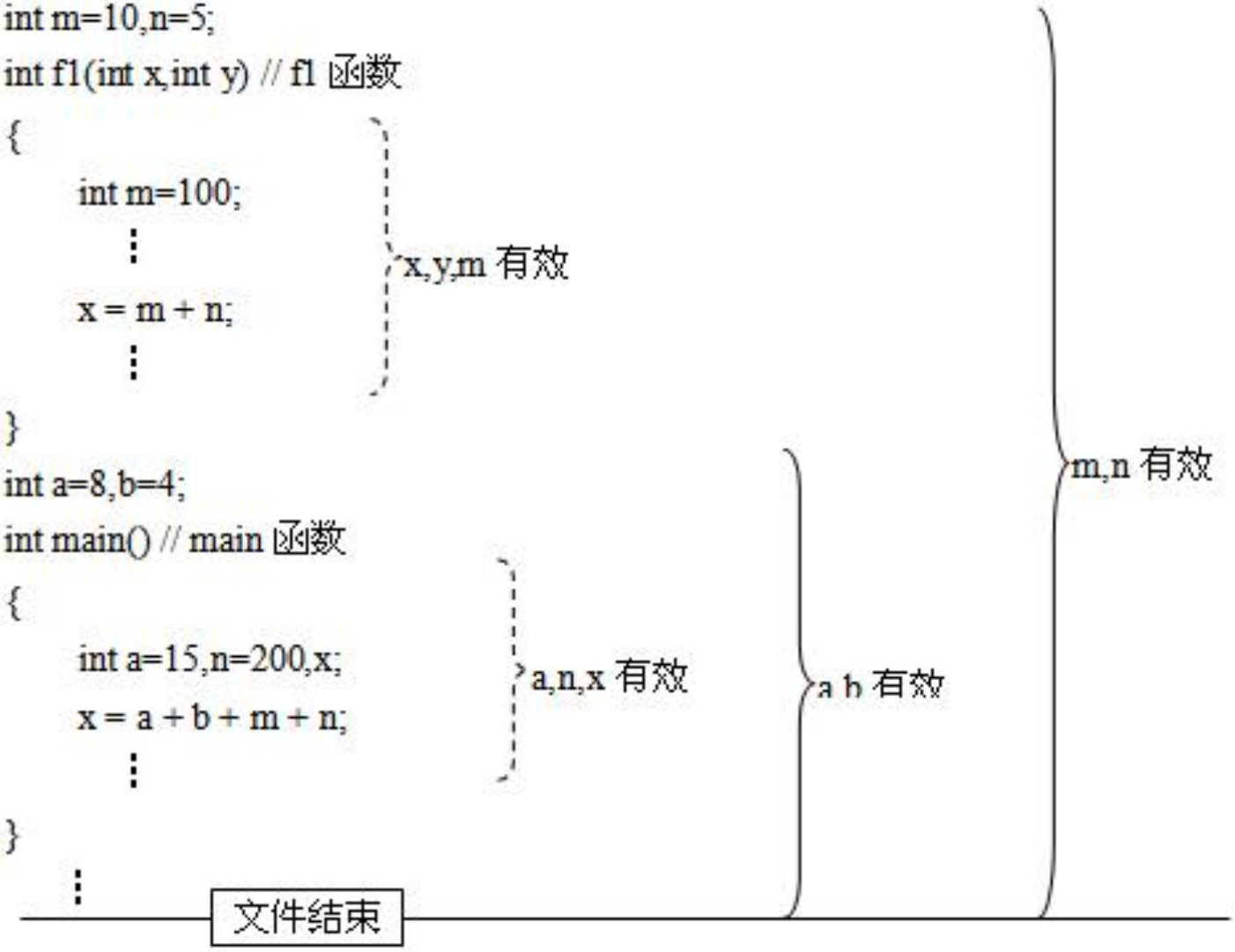
a,b,c 有效

- 局部变量的说明。
- （1）局部变量只能在定义它的区域及其子区域中使用。
- （2）在同一个区域中不能定义相同名字的变量。
- （3）在不同区域中允许定义相同名字的变量，但本质上它们是不同的变量
- （4）如果一个变量所处区域的子区域中有同名的变量，则该变量在子区域无效，有效的是子区域的变量，称为定义屏蔽。

- 全局变量
- 在源文件中，但在函数外部定义的变量，称为全局变量（global variable），全局变量的有效区域是从定义变量的位置开始到源文件结束。

4.9.2 全局变量

```
1    int m=10,n=5;
2    int f1(int x,int y) // f1 函数
3    {
4        int m=100;
             ⋮
10       x = m + n;
             ⋮
20   }
21   int a=8,b=4;
22   int main() // main 函数
23   {
24       int a=15,n=200,x;
25       x = a + b + m + n;
             ⋮
30   }
         ⋮
```



- 函数之间数据传递尽管可以利用全局变量，但这样一来也导致两个函数彼此分不开，违背模块化的原则，所以结构化程序设计

- 提倡少用或不用全局变量。

例：编写一个函数swap用于交换两个整数的值
如，a=3，b=4，调用函数swap后，得a=4，b=3

方法1：

```
void swap(int x,int y)
{  int t;
   t=x; x=y; y=t;
}
int main( )
{  int a,b;
   cin>>a>>b;
   swap(a,b);
   cout<<"a="<<a<<",b="
                <<b<<endl;
   return 0;
}
```

```
3 4
a=3,b=4
```

方法2：

```
int a,b;
void swap()
{  int t;
   t=a; a=b; b=t;  }
int main( )
{
   cin>>a>>b;
   swap();
   cout<<"a="<<a<<",b="
                <<b<<endl;
   return 0;
}
```

正确!

```
3 4
a=4,b=3
```

```
int i=0;
fun1(int i)
{  i=(i%i)*((i*i)/(2*i)+4);
    cout<<"i="<<i<<endl;
    return(i);
}
fun2(int i)
{ i=i<=2?5:0;
   return(i);
}
main()
{ int i=5;
  fun2(i/2);  cout<<"i="<<i<<endl;
  fun2(i=i/2);  cout<<"i="<<i<<endl;
  fun2(i/2);  cout<<"i="<<i<<endl;
  fun1(i/2);  cout<<"i="<<i<<endl;
}
```

```
i=5
i=2
i=2
i=0
i=2
```

- C++的实体通常有三类：
- ①变量或对象。例如基本类型变量、数组对象、指针对象、结构体对象等；
- ②函数；
- ③类型。包含结构体类型、共用体类型、类类型。

- 作用域是程序中的一段区域。在同一个作用域上，C++程序中每个名字都与唯一的实体对应；如果在不同的作用域上，程序中可以多次使用同一个名字，对应不同作用域中的不同实体。

- 实体在作用域内可以使用称为可见（visible），又称有效。可见的含义是指实体在作用域上可以使用。
- 下面给出C++实体可见规则。
- （1）规则一。同一个作用域内不允许有相同名字的实体，不同作用域的实体可以有相同名字。

- （2）规则二。实体在包含它的作用域内，从定义或声明的位置开始，按文件行的顺序往后（往下）直到该作用域结束均是可见的，包含作用域内的所有子区域及其嵌套。

- （3）规则三。若实体A在包含它的作用域内的子区域中出现了相同名字的实体B，则实体A被屏蔽。

```
int main()
{  int a,b,c;
    c=2;
     {  int c;
        ……
        c=a+b;
        ……
     }
    c=c+1;
    return 0;
}
```

- （4）规则四。可以使用extern声明将变量或函数实体的可见区域往前延伸，称为前置声明（forward declaration）。

- （5）规则五。在全局作用域中，变量或函数实体若使用static修饰，则该实体对于其他源文件是屏蔽的，称为私有的（private）。

- ①extern声明变量实体的形式为：

```
extern 类型 变量名, ……
```

- extern声明函数原型的形式为：

```
extern 返回类型 函数名(类型1 参数名1, ……);
extern 返回类型 函数名(类型1,……);
```

- ②static修饰变量实体的形式为：

```
static 类型 变量名[=初值], ……
```

- static修饰函数原型的形式为：

```
static 返回类型 函数名(类型1 参数名1, ……);
static 返回类型 函数名(类型1, ……) ;
```

```
1  // FILE1.CPP 全局作用域
2  int a=1 , b=2;    //全局变量
3  int c=10 , d=11;    //全局变量
4  void f1(int n,int m)    //f1函数作用域
5  {
6      int x=21, y=22,z=23;  //f1局部变量
7      extern int h,k;//正确，h=60 k=61 规则四
8      n = n + t ;    //错误，t 违反规则二
```

```
 9      if (n>100) {  //块作用域
10          int x=31,t=20; //复合语句局部变量
11          n=x+y; //正确，n=31+22 规则二 规则三
12          if (m>10) {  //嵌套块作用域
13           int y=41;//嵌套的复合语句局部变量
14           n=x+y;//正确，n=31+41 规则二 规则三
15          }
16      }
17      n = a + x ;    //正确，n=1+21  规则二
18      m = e + f ;    //错误，e,f 违反规则二
19      n = h + k ;    //正确，n=60+61 规则四
20  }
```

```
21  int e=50 , f=51;    //全局变量
22  int h=60 , k=61;    //全局变量
23  void f2(int n,int m)    //f2函数作用域
24  {
25      n=a+b+e+f;  //正确，n=1+2+50+51 规则二
26      m=z;        //错误，z 违反规则一
27  }
28  int f3(int n,int m)    //f3函数作用域
29  {
30      return n+m;    //正确，规则二
31  }
```

```
32  int f4(int n,int m)    //f4函数作用域
33  {
34      return n-m;    //正确，规则二
35  }
36  // FILE1.CPP 文件结束
```

```
1  // FILE2.CPP 全局作用域
2  int a=201 , b=202;
   //错误，连接时与FILE1.CPP的同名 违反规则一
3  void f1(int n,int m)
   //错误，连接时与FILE1.CPP的同名 违反规则一
4  {
5      n=n*m;
6  }
7  static int c=210 , d=212;  //正确，规则五
8  static void f2(int n,int m) //正确，规则五
9  {
10     n=n/m;
11 }
```

```
12 extern int h , k;    //正确，规则四
13 extern int f4(int n,int m); //正确，规则四
14 int main()
15 {
16     int p,q,r;//main函数局部变量
17     p = c + d;//正确，p=210+212 规则二
18     f2(-1,-2);//正确，不是FILE1.CPP的 规则二
19     q=e+f;//错误，试图使用FILE1.CPP违反规则一
20     f3(-10,-12); //错误
21     r = h + k;    //正确，r=60+61 规则四
22     f4(-20,-22);    //正确，规则四
23     return 0;
24 }
25 // FILE2.CPP 文件结束
```

- C++中，每个名字都有作用域，即可以使用名字的区域，而每个对象都有生命期（lifetimes），即在程序执行过程中对象存在的时间。

- 1．动态存储
- 动态存储（dynamic storage duration）是指在程序运行期间，系统为对象动态地分配存储空间。动态存储的特点是存储空间的分配和释放是动态的，要么由函数调用来自动分配释放，要么由程序指令来人工分配释放，这个生命期是整个程序运行期的一部分。
- 动态存储的优点是对象不持久地占有存储空间，释放后让出空闲空间给其他对象的分配。

- 动态存储在分配和释放的形式有两种，一种是由函数调用来自动完成的，称为自动存储（automatic storage），一种是由程序员通过指令的方式来人工完成的，称为自由存储（free storage）。

- 2. 静态存储
- 静态存储（static storage duration）是指对象在整个程序运行期持久占有存储空间，其生命期与程序运行期相同。静态存储的特点是对象的数据可以在程序运行期始终保持直到修改为止，或者程序结束为止，静态存储的分配和释放在编译完成时就决定好了。

- 现代程序设计的观点是，除非有必要尽量少地使用静态存储。

- 3．自动对象
- 默认情况下，函数或复合语句中的对象（包含形参）称为自动对象（automatic objects），其存储方式是自动存储，程序中大多数对象是自动存储。

auto 类型 变量名[=初值], ……

- 4. 寄存器变量
- C++语言允许用CPU的寄存器来存放局部变量，称为寄存器变量。在局部变量前加上register存储类别修饰来定义的，其形式为：

register 类型 变量名[=初值], ……

- 5．静态局部对象
- 在局部对象的前面加上static存储类别修饰用来指明对象是静态局部对象（static local object），一般形式为：

static 类型 变量名[=初值]，……

例4.15

```
1  #include <iostream>
2  using namespace std;
3  int fun()
4  {
5      static int cnt=0;//静态局部变量会保持其值
6      cnt++;
7      return cnt;
8  }
9  int main()
10 {
11     int i,c;
12     for (i=1;i<=10;i++) c=fun();
13     cout<<c<<endl;
14     return 0;
15 }
```


程序设计