

高级语言C++程序设计

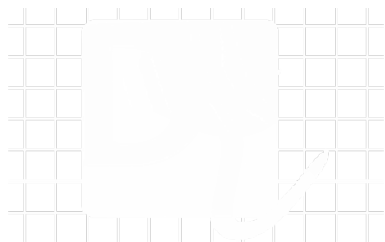
Lecture 6 函数

李雨森

南开大学 计算机学院

2021

函数的引入



函数的引入

问题：已知 a ，用迭代法求 $x = \sqrt[3]{a}$

解析：求立方根的迭代公式为

$$x_{i+1} = \frac{2}{3}x_i + \frac{a}{3x_i^2}$$

设 x 的初始值为 a ，迭代到 $|x_{i+1} - x_i| < \varepsilon = 10^{-5}$ 为止

```
float x, root, croot = a;  
do{  
    root = croot;  
    croot = (2*root + a/(root*root))/3;  
}while(fabs(croot-root) > eps);  
x = croot;
```

函数的引入

问题：分别求a, b的立方根

```
float x1, x2, root, croot;  
croot = a;  
do{  
    root = croot;  
    croot = (2*root + a/(root*root))/3;  
}while(fabs(croot-root) > eps);  
x1 = croot;
```

```
croot = b;  
do{  
    root = croot;  
    croot = (2*root + b/(root*root))/3;  
}while(fabs(croot-root) > eps);  
x2 = croot;
```

程序中包含两段类似的代码，
完成相同的功能

函数的引入

问题：分别求a, b的立方根

```
float x1, x2;  
x1 = cuberoot(a);  
  
x2 = cuberoot(b);
```

```
float cuberoot(float x) {  
    float root , croot;  
    const float eps=1e-6;  
    croot=x;  
    do{  
        root=croot;  
        croot=(2*root+x/(root*root))/3;  
    }while(fabs(croot - root)>eps);  
    return croot;  
}
```

将功能相同的模块抽象成**函数(function)**，简化程序

函数的引入

程序的功能相对独立，用来解决某个问题
具有明显的入口和出口

- 入口: 参数
- 出口: 返回值

```
float cuberoot(float x) {  
    float root , croot;  
    const float eps=1e-6;  
    croot=x;  
    do{  
        root=crout;  
        croot=(2*root+x/(root*root))/3;  
    }while(fabs(croot - root)>eps);  
    return croot;  
}
```

函数的作用

函数的作用

- ❑ 实现程序功能的模块化
- ❑ 实现程序结构的简化
- ❑ 实现程序代码的重用

函数的应用场景

- ❑ 包含多处功能相同的代码
 - 处理数据的类型、处理过程相同或相似
- ❑ 代码段具有代表性或特殊含义

函数的简单使用

用函数的思想实现最简单的C++程序

```
#include<iostream>
using namespace std;
void printString() {
    cout<<"Hello!"<<endl;
    return;
}
```

主调函数

```
int main() {
    printString();
    return 0;
}
```

被调函数

主调函数和被调函数是相对的概念，一个函数既可以调用其它函数(包括该函数自身)，亦可以被其它函数调用

函数的简单使用

用函数的思想实现最简单的C++程序

```
#include<iostream>
using namespace std;
void printString(){//函数定义
    cout<<"Hello!"<<endl;
    return;
}
int main(){
    printString();//函数调用
    return 0;
}
```

函数定义必须出现在
调用函数之前

函数的简单使用

用函数的思想实现最简单的C++程序

```
#include<iostream>
using namespace std;
void printString(); //函数原型(函数声明)
int main() {
    printString(); //调用函数
    return 0;
}
void printString() { //函数定义
    cout<<"Hello!"<<endl;
    return;
}
```

也可以先声明函数，
后定义

函数的分类

标准库函数

- 程序中可直接使用(调用)系统预定义的标准库函数, 但要求在调用前使用编译预处理指令include将对应的头文件包含进来

用户自定义函数

- 由用户自定义的函数与系统预定义的标准库函数的不同点在于, 自定义函数的函数名、参数个数、函数返回值类型以及函数所实现的功能等都完全由用户程序来规定(指定)

函数的分类

```
#include<cmath>
using namespace std;
```

用户自定义函数



```
float cuberoot(float x) {
    float root , croot;
    const float eps=1e-6;
    croot=x;
    do{
        root=croot;
        croot=(2*root+x/(root*root))/3;
    }while(fabs(croot - root)>eps);
    return croot;
}
```

标准库函数, 求绝对值



```
int main() {
    cuberoot(1.1); //调用自定义函数
    return 0;
}
```

函数的分类

```
#include<iostream>
#include<string.h>
#include<stdio.h>
using namespace std;
```

```
int main()
{
    char a[] = "abcde";
    char b[] = "nankai";
    cout<<strlen(a)<<endl;
    cout<<strcmp(a, b)<<endl;
    gets(a);
    cout<<a<<endl;
    return 0;
}
```

标准库函数



函数的分类

无参函数

- 调用它们时不需要提供实际参数
- 函数定义的一般形式 `<返回值类型><函数名>() {<函数体>}`
- 通常用来实现某种特定的功能
 - 不需要进行数据的传递
 - 处理的数据通常与主调函数无关

```
void printStar() { //打印10个“*”的无参函数
    for(int i=0;i<10;i++)
        cout<<"*";
    cout<<endl;
}
```

函数的分类

有参函数（带有参数的用户自定义函数）

- 进行调用时，必须提供所需个数的且具有相匹配数据类型的实际参数
- 定义的一般形式

<返回值类型> <函数名> (<以逗号分割的形参类型及名字表>)
{<函数体>}

- 通过调用处提供的不同实参值来计算出其对应的函数值、或实现某种与传递过来的那些不同值有关的某种功能
-

函数的分类

有参函数

- 定义一个函数，输出k个“*”

```
void printStar(int k) {  
    for(int i=0;i<k;i++)  
        cout<<"*";  
    cout<<endl;  
}
```


练习题

函数的作用包括：

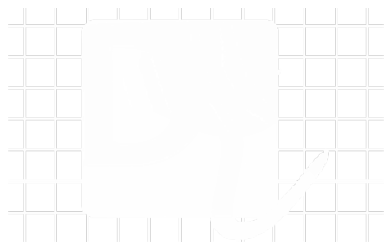
- A 实现程序结构的简化
- B 实现程序代码的重用
- C 加快程序执行速度

练习题

关于函数调用说法正确的是：

- A 函数定义必须放在调用函数前面
- B 函数声明和定义必须同时放在调用函数前面
- C 函数声明可以省略
- D 函数定义可以放在调用函数后面

函数定义



函数定义

函数名(符合标识符命名规则)

```
float cuberoot(float x) {  
    float root , croot;  
    const float eps=1e-6;  
    croot=x;  
    do{  
        root=crout;  
        croot=(2*root+x/ (root*root) ) /3;  
    }while (fabs (croot - root)>eps) ;  
    return croot;  
}
```

函数定义

参数列表

```
float cuberoot(float x){  
    float root , croot;  
    const float eps=1e-6;  
    croot=x;  
    do{  
        root=crout;  
        croot=(2*root+x/(root*root))/3;  
    }while(fabs(croot - root)>eps);  
    return croot;  
}
```

■ 空参数表

```
void printroot(){}  
void printroot(void){}
```

■ 多个参数 (逗号隔开)

```
int sum(int a, int b){}
```

函数定义

```
float cuberoot(float x) {  
    float root , croot;  
    const float eps=1e-6;  
    croot=x;  
    do{  
        root=crout;  
        croot=(2*root+x/(root*root))/3;  
    }while(fabs(croot - root)>eps);  
    return croot;  
}
```

- 是复合语句，即程序块，由完成函数功能所需的全部语句构成
- 可以是空语句 { ; }
- 可以没有任何语句 { }

函数体
(花括号部分)

函数定义

函数返回值类型

```
float cuberoot(float x) {  
    float root, croot;  
    const float eps=1e-6;  
    croot=x;  
    do{  
        root=crout;  
        croot=(2*root+x/(root*root))/3;  
    }while(fabs(croot - root)>eps);  
    return croot;  
}
```

两者类型
必须相同

函数返回

函数定义

函数定义的格式为

[<属性说明>] <返回值类型> <函数名> ([<参数表>])
{<函数体>}

属性说明：可缺省，一般可以是下面的关键字之一

- **inline**: 表示该函数为内联函数
 - **static**: 表示该函数为静态函数
 - **virtual**: 表示该函数为虚函数
 - **friend**: 表示该函数为某类(class)的友元函数
-

函数定义

```
void printroot() {}  
static void printroot(int a) {a++; return;}  
void printroot(int a, char b) {a+b;}  
int printroot(int a, int b) {return a+b;}  
void printroot(char c[8]) {c[0]=c[1];}
```

函数声明

用来指明函数的名称、参数以及返回值类型

函数原型格式为：

[<属性说明>] <返回值类型> <函数名> ([<参数表>]);

例如

```
int add(int a, int b);  
void swap(float &s, float &t);  
void print(char [8]);
```

函数声明

```
#include<iostream>
using namespace std;
void printString(); //函数原型(函数声明)
int main() {
    printString(); //调用函数
    return 0;
}
void printString() { //函数定义
    cout<<"Hello!"<<endl;
    return;
}
```

函数声明与函数定义

如果同时存在函数声明和函数定义，函数定义可以出现在函数调用之后；如果只有函数定义，必须出现在调用之前

```
#include<iostream>
using namespace std;
void printString(); //函数声明
int main() {
    printString(); //调用函数
    return 0;
}
void printString() { //函数定义
    cout<<"Hello!"<<endl;
    return;
}
```

```
#include<iostream>
using namespace std;
void printString() { //函数定义
    cout<<"Hello!"<<endl;
    return;
}
int main() {
    printString(); //调用函数
    return 0;
}
```

函数声明与函数定义

函数声明的参数表中, 参数名可以省略
函数定义的参数表中, 必须给出参数名

```
int add(int, int); //函数声明
```

```
int add(int a, int b) { //函数定义  
    return a+b;  
}
```

函数声明与函数定义

函数原型的参数表后面加分号“;”

函数定义的参数表后面是函数体，即花括号{ }

```
#include<iostream>
using namespace std;
void printString(); //函数声明
int main() {
    printString(); //调用函数
    return 0;
}
void printString() { //函数定义
    cout<<"Hello!"<<endl;
    return;
}
```

函数声明与函数定义

函数定义不能出现在任何函数体中
函数原型可以出现在其它函数体中

```
int main() {  
    void printString(); //调用声明  
    printString();  
    return 0;  
}  
void printString() { //函数定义  
    cout<<"Hello!"<<endl;  
    return;  
}
```

函数调用

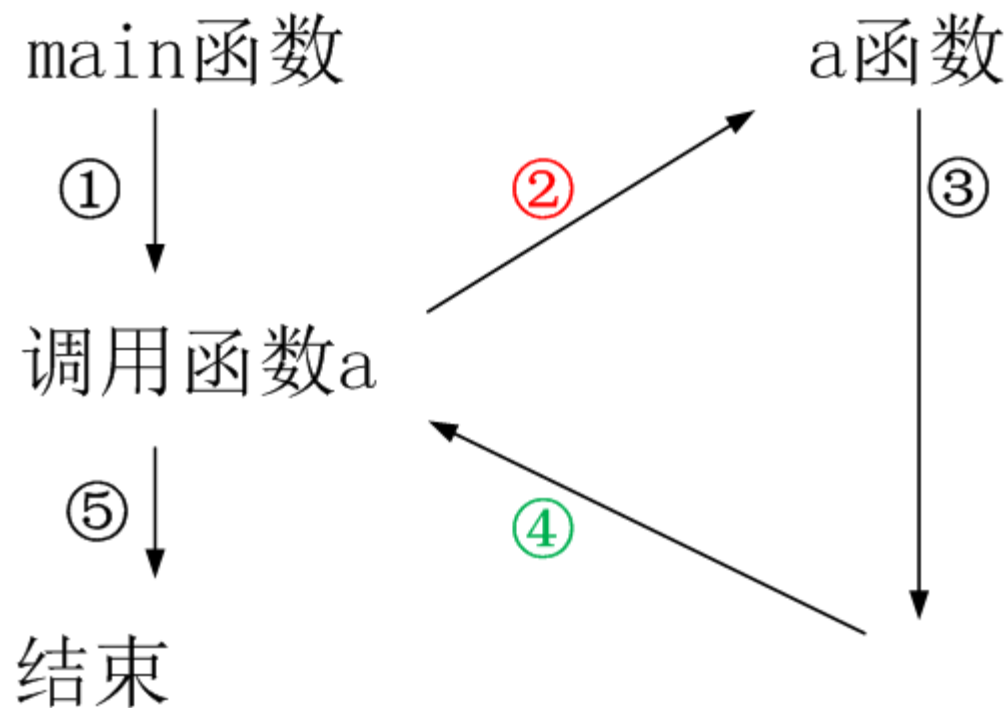
函数调用是已定义函数的一次实际运行，调用一个函数就是去执行该函数之函数体的过程

```
#include<iostream>
using namespace std;
void printString(); //函数声明
int main() {
    printString(); //调用函数
    return 0;
}
void printString() { //函数定义
    cout<<"Hello!"<<endl;
    return;
}
```

在C++程序中，除main函数外，其它任一函数的执行都是通过主函数中直接或间接地调用该函数而引发的

函数调用

函数调用过程




函数调用

函数调用的执行顺序

- ❑ 根据调用语句中的函数名在整个程序中搜索同名函数定义；
 - ❑ 对实参数的参数个数，类型，顺序进行核对，判定是否与函数定义中的形参表对应一致
 - ❑ 根据参数的类型(值参数或引用参数)进行值参数的值传递或引用参数的换名
 - ❑ 运行函数体代码
 - ❑ 返回调用点，并返回所要求的函数值
-

函数返回

```
double f (double x) {  
    double y;  
    y=(x*x+x+1)/2-5.5;  
    return y;  函数返回  
}
```

- 函数的返回表示函数执行结束，将执行结果（无论是否有具体的数据）返回到调用函数的地方
 - 函数返回用return语句表示，返回值（如果有）则为return后面的表达式的值
-

函数返回

返回值类型

```
void printString() {  
    cout<<"Hello!"<<endl;  
    return;  
}
```

如果函数无值返回, 应说明为`void` 类型, `return` 可省略

函数返回

返回值类型

```
double f (double x) {  
    double y;  
    y = (x*x+x+1) / 2 - 5.5;  
    return y;  
}
```

值型：返回一个具有类型的值，包括int、float、char、bool等，函数只能返回一个值

函数返回

返回值类型

```
double & f (double &x) {  
    x++;  
    return x;  
}
```

复合类型: 指针、引用、类类型等

函数返回

- ❑ 空型(void)
 - 如果函数无值返回, 应说明为void 类型。未作类型说明的函数, 系统认为是int 类型函数, 应返回一整型值
 - ❑ 值型: 返回一个具有类型的值, 包括int、float、char、bool等
 - 当函数要返回的值不止一个时, 情况比较复杂, 一般它可以以结构或类的形式, 也可以以指针类型方式实现
 - ❑ 复合类型
 - 指针、引用、类类型等
 - ❑ 如果函数中包含多个分支, 则需要保证每一条路径都能够带有一个return语句
-

函数返回

```
#include<iostream>
using namespace std;
double f (double x){return x*x;} //函数f的定义

int main(){
    double z,a;
    z=(f(2.5)+2*f(6))/f(4.3); //调用自定义函数f
    cout<<"z="<<z<<endl;
    cout<<"Input a="; //提示用户输入
    cin>>a;
    cout<<"f(a)="<<f(a)<<endl; //算出f(a)并输出
    f(a) = 1; //Error! f(a)非左值
    return 0;
}
```

练习

以下**函数定义**正确的是：

A `void _func() {}`

B `void sum(int a, int b) {return a+b;}`

C `int sum(int a, int b) {a+b;}`

D `int sum(int, int);`

E `int, double f(int a, double b) {
 return a, b; }`

练习

以下程序有什么错误？

```
#include<iostream>
```

```
using namespace std;
```

(1)无函数声明

```
int main() {
```

```
    _sum(1, 2, 3);
```

```
    return 0;
```

```
}
```

```
int _sum(int a, int b) {
```

```
    return a + b;
```

```
}
```

(2)参数个数错误

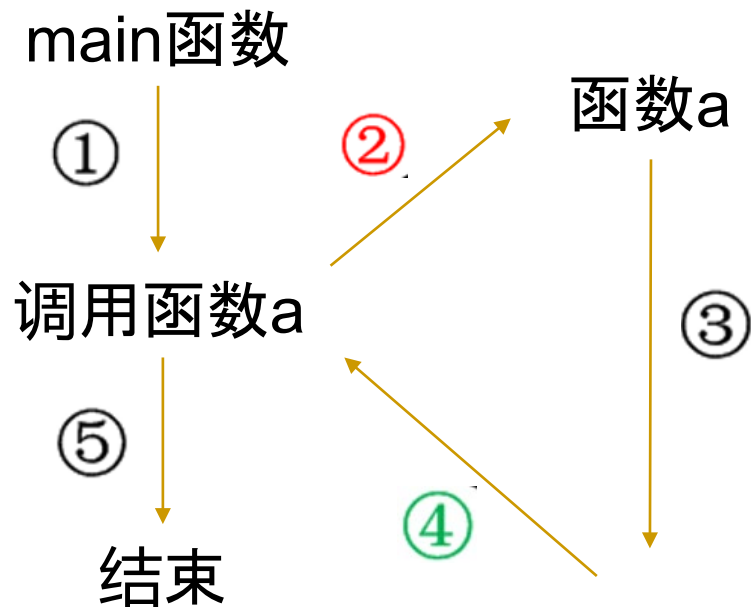
内联函数

可在一般的函数说明前冠以关键字**inline**，称这样的函数为**内联函数**

```
inline double f (double x) {  
    double y;  
    y = (x*x+x+1) / 2 - 5.5;  
    return y;  
}
```

内联函数

普通函数调用过程



“控制转移”带来额外开销

内联函数

内联函数调用过程



在编译过程中，系统把内联函数的执行代码插入到每个调用点处(取代函数调用)，从而使程序执行过程中，每次对该函数调用时不需控制转移，可节省执行时间

内联函数

main函数
执行代码



嵌入a函数之
后的main函数
执行代码

函数a的执行
代码

由于每个调用点处均出现那一函数的执行代码拷贝，相对来说使用内联函数后会**扩大其代码空间**

- 内联函数的函数体一般讲不宜过大，以**1--5**行为宜

内联函数

```
#include <iostream>
using namespace std;
inline int max(int x, int y) { //内联函数max
    return (x>y?x:y);
}
void main()
{
    int a,b;
    cout<<"Input a,b:";
    cin>>a>>b;
    cout<<"max (a,b) ="<<max (a,b) <<endl;
    //对内联函数max的调用
}
```

练习题

已知函数 f, 下面赋值正确的是: **A, B**

A `double a = f(10);`

B `int a = f(10);`

C `f(10) = 2;`

```
double f (double x) {  
    double y;  
    y=(x*x+x+1)/2-5.5;  
    return y;  
}
```

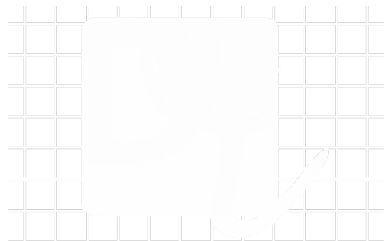

练习题

指出下面程序中的错误

```
double f (double x, double y) {  
    if (x > 0) {  
        return -1;  
    } else if (y > 0) {  
        return 1;  
    }  
}
```

不能保证每条分支都带return

函数的参数



函数参数

无参函数

```
void printString() {  
    cout<<"Hello!"<<endl;  
    return;  
}
```

函数参数

一个参数

```
double f (double x) {  
    double y;  
    y=(x*x+x+1)/2-5.5;  
    return y;  
}
```

函数参数

多个参数


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

函数参数

```
#include <iostream>
using namespace std;
```

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

函数原型或定义中的参数称为**形式参数(形参)**



```
void main()
{
    int a,b;
    cout<<"Input a,b:";
    cin>>a>>b;
    cout<<"max (a,b) ="<<max(a,b)<<endl;
}
```

函数调用表达式中的参数称为**实际参数(实参)**

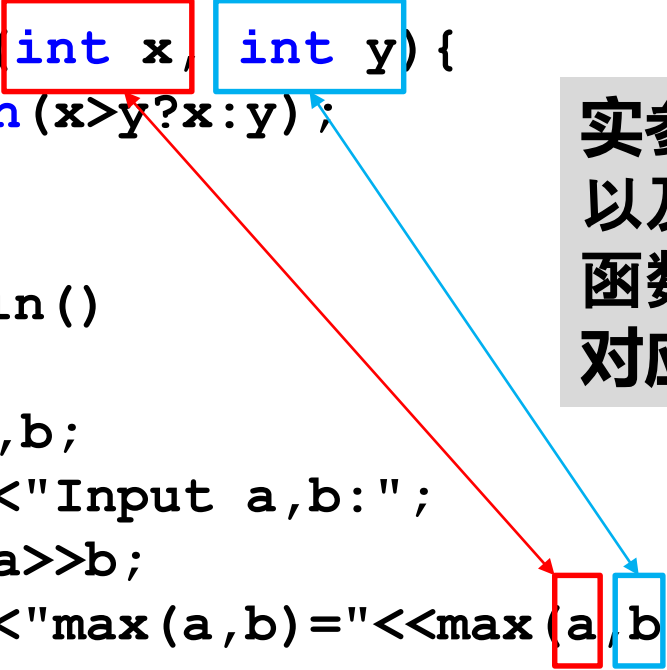


函数参数

```
#include <iostream>
using namespace std;

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

void main()
{
    int a,b;
    cout<<"Input a,b:";
    cin>>a>>b;
    cout<<"max (a,b) ="<<max(a,b)<<endl;
}
```



The diagram illustrates the correspondence between function arguments and parameters. A red box highlights the parameter 'x' in the 'max' function signature, and a blue box highlights the parameter 'y'. In the 'main' function, a red box highlights the variable 'a' and a blue box highlights the variable 'b' in the 'max(a,b)' call. A red arrow points from the 'a' box in 'main' to the 'x' box in 'max', and a blue arrow points from the 'b' box in 'main' to the 'y' box in 'max'.

实参表在参数个数、参数顺序、以及参数类型等方面要与被调函数的形参表之间有一个一一对应的相互匹配关系

函数参数

一般写法

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


函数参数

省略参数名(无名参数)

- 函数定义中，只有类型，没有名称的参数

```
int f(int a, int b) {return a+b*b;}
```

```
int f(int a, int b, int) {return a*a+b;}
```

- 两个不同的函数同名，但由于第二个函数包含一无名参数，使得在调用时能够被区分，`f(x, y)` 是第一个函数的调用，`f(x, y, 0)` 是第二个函数的调用

函数参数

可缺省参数（参数的默认值）

- 允许在函数定义处为其中最后面的连续若干个参数设置默认值(也称缺省值)

```
#include<iostream>
using namespace std;
void func(int a=11, int b=22, int c=33) {
    //为参数a、b、c设置了默认值11、22与33
    cout<<"a="<<a<<" , b="<<b<<" , c="<<c<<endl;
}
```

函数参数

可缺省参数（参数的默认值）

- 若调用处缺省了某个或某些实参的情况下，系统将自动使用那些在函数定义处给定的参数默认值

```
int main() {  
    func();    //调用时缺省了3个实参，将使用  
    func(55); //调用时缺省了后2个实参，将使用  
    func(77, 99); //调用时缺省了最后1个实参，将使用  
    func(8, 88, 888); //调用时没缺省任一个实参，  
    return 0;  
}
```

函数参数

只能为函数最后面的**连续**若干个参数设置默认值
，且在调用处也只能缺省后面的**连续**若干个实参

□ 例如：

```
void func(int a, int b=2, int c=3);    //OK!  
void func(int a=1, int b, int c=3);    //ERROR!  
void func(int a=1, int b=2, int c);    //ERROR!
```

对第一个函数说明，采用如下的调用语句：

```
func(1, 22, 333);    //OK!    调用时给出所有实参  
func();              // ERROR!    参数a没有默认值  
func(10,20);         //OK!    参数c默认为3  
func(5, ,9); //ERROR!调用处也只能缺省后面的连续若干个实参
```

练习题

下面程序段中，两个函数定义可以共存的是？

C

A

```
void func(int a, int b, int c=3) {}  
void func(int a, int b, int) {}
```

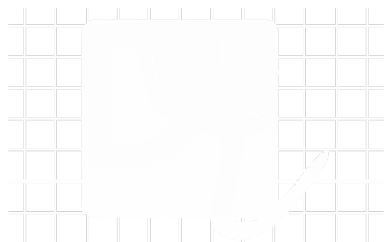
B

```
void func(int a, int b, int c=3) {}  
void func(int a, int b) {}
```

C

```
void func(int a, int b, int) {}  
void func(int a, int b) {}
```

函数的参数传递




参数传递

函数调用过程中的参数传递

一般传递过程（赋值形参传递）

a b

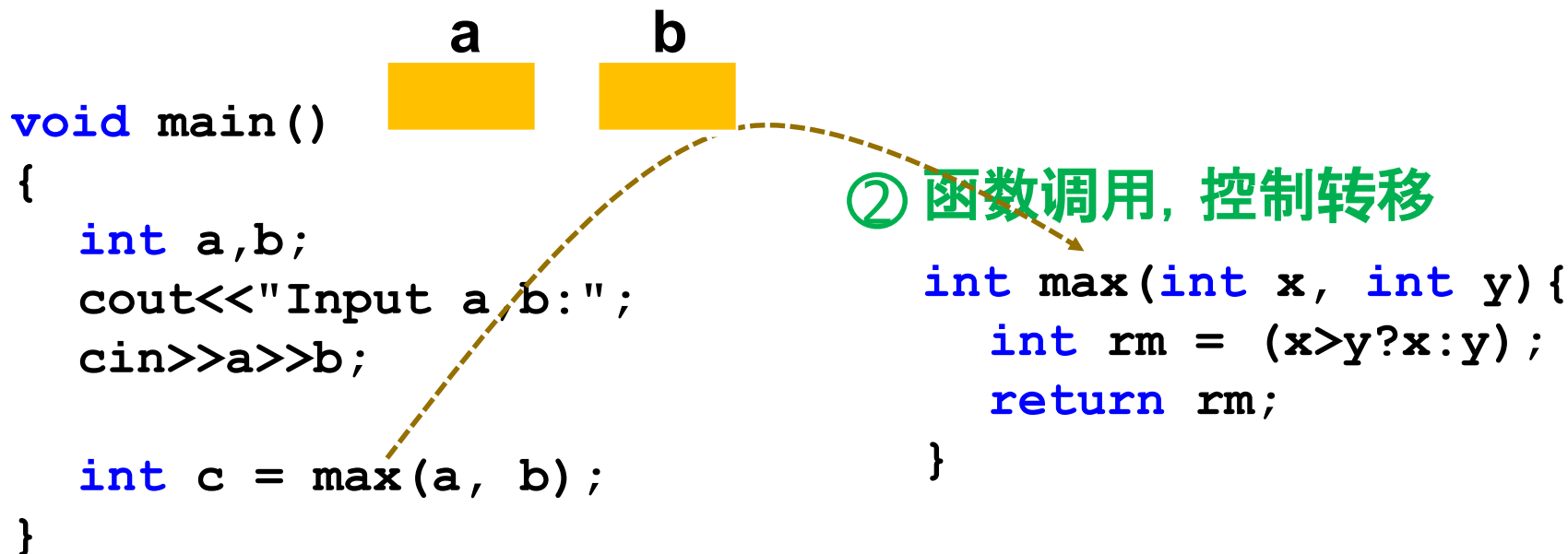


```
void main()  
{  
    ① 生成main函数局部变量a,b  
    int a,b;  
    cout<<"Input a,b:";  
    cin>>a>>b;  
  
    int c = max(a, b);  
}  
  
int max(int x, int y){  
    int rm = (x>y?x:y);  
    return rm;  
}
```

参数传递

函数调用过程中的参数传递

一般传递过程（赋值形参传递）



参数传递

函数调用过程中的参数传递

一般传递过程（赋值形参传递）

```
void main()  
{  
    int a,b;  
    cout<<"Input a b:";  
    cin>>a>>b;  
  
    int c = max(a, b);  
}
```



③ 生成max函数局部变量x,y

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

a,b和x,y是不同的变量, 存在不同的地址

参数传递

函数调用过程中的参数传递

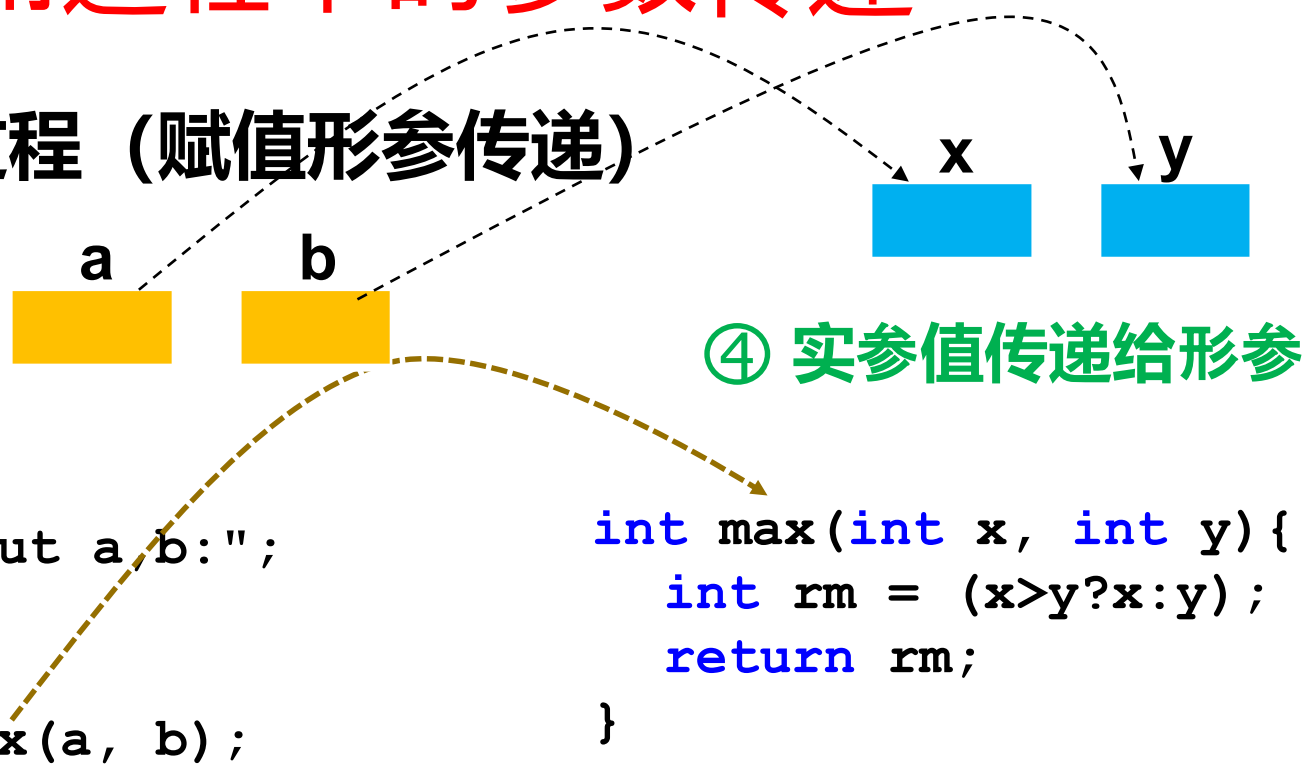
一般传递过程（赋值形参传递）

```
void main()  
{  
    int a,b;  
    cout<<"Input a b:";  
    cin>>a>>b;  
  
    int c = max(a, b);  
}
```



④ 实参值传递给形参

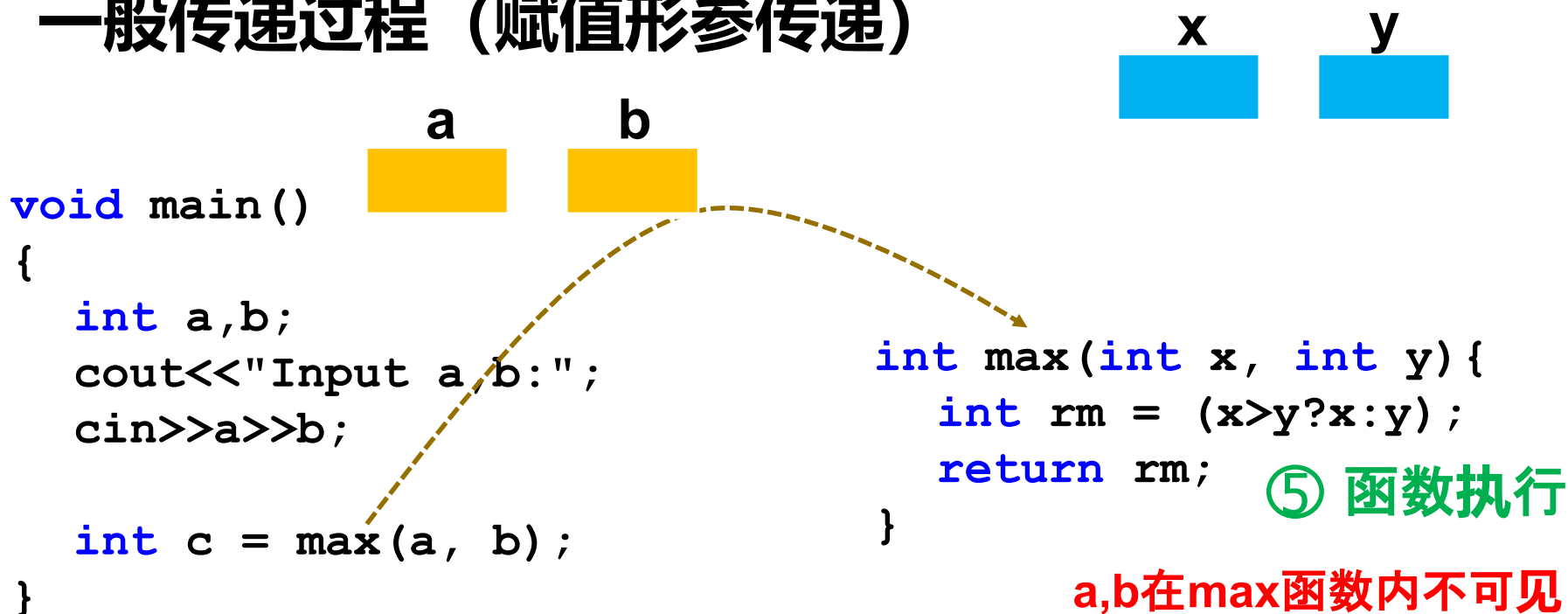
```
int max(int x, int y){  
    int rm = (x>y?x:y);  
    return rm;  
}
```



参数传递

函数调用过程中的参数传递

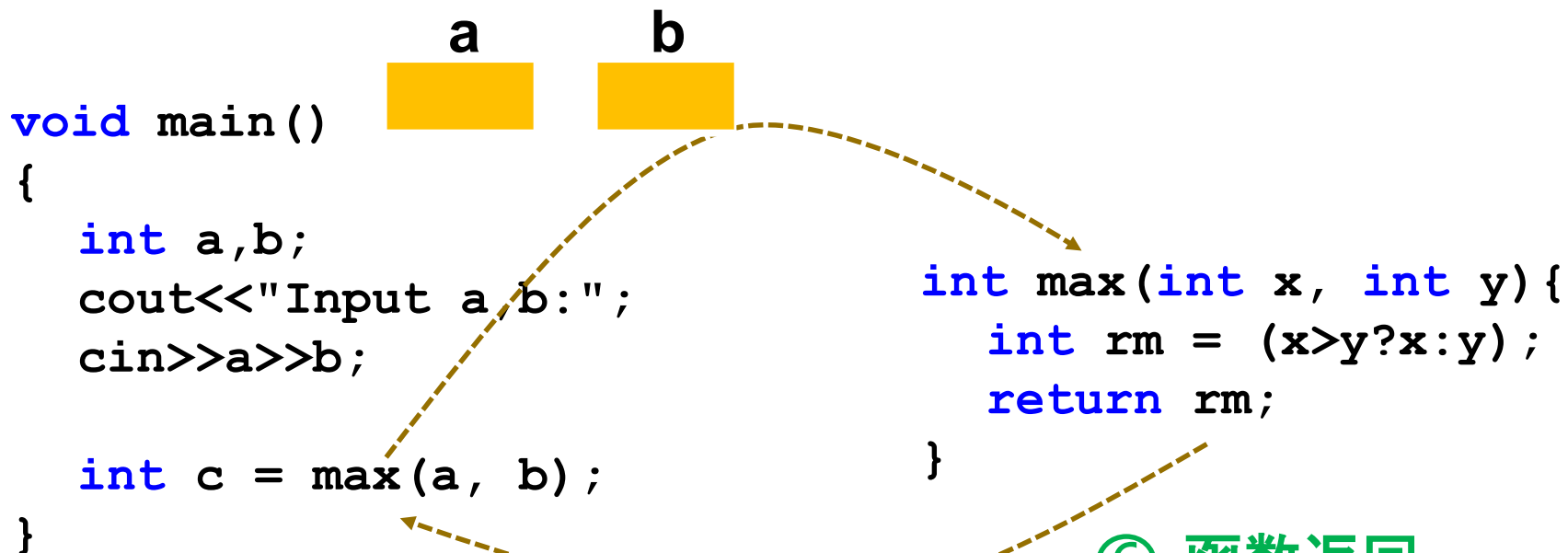
一般传递过程（赋值形参传递）



参数传递

函数调用过程中的参数传递

一般传递过程（赋值形参传递）



⑥ 函数返回

返回值拷贝给c, max函数局部变量消失

参数传递

赋值形参

- 函数调用表达式中的实参(**实参表达式**), 可以是指定类型的常量、变量或表达式

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

```
int main(){  
    int x = 5;  
    int c=add(1, 4*x+2); //实参为表达式  
    .....  
}
```

参数传递

赋值形参

- 凡是赋值形参，在函数的每次调用时，都必须为每一个赋值形参创建一个新的参数变量

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

```
int main() {  
    int x = 5;  
    int c=add(1, 4*x+2);  
    int d=add(2, 6);  
}
```

每次调用会生成不同的a,b变量

参数传递

如果参数是数组, 如何传递?

先生成数组x, 再将a的每个元素copy到x?

```
void main()  
{  
    int a[100], b;  
    cin>>b;  
  
    int c = search(a, b);  
}
```

a
↓
int search(int x[100], int y) {
 return x[y];
}

参数传递

如果参数是数组, 如何传递?

按地址传递过程

```
void main()  
{  
    int a[100], b;  
    cin>>b;  
  
    int c = search(a, b);  
}
```

数组a



传递实参数组的首地址
, 并不创建新的数组

```
int search(int x[100], int y){  
    return x[y];  
}
```

在函数体中, 根据形参数组**首地址**和下标指示的**偏移量**访问数组元素

参数传递

如果参数是数组, 如何传递?

按地址传递过程

```
void main()  
{  
    int a[100], b;  
    cin>>b;  
  
    int c = search(a, b);  
}
```

数组a



传递实参数组的首地址
, 并不创建新的数组

```
int search(int x[100], int y){  
    return x[y];  
}
```

节省空间, 提高效率

参数传递

如果参数是数组, 如何传递?

按地址传递过程

```
void main()  
{  
    int a[100], b;  
    cin>>b;  
  
    int c = search(a, b);  
}
```

数组a



传递实参数组的地址
，并不创建新的数组

```
int search(int x[100], int y){  
    x[0] = -1;  
    return x[y];  
}
```

函数体对内存空间进行的修改将保留,
对主调函数仍然有效

参数传递

一维数组作为参数

一维数组作为函数的参数，在形参表中将参数说明为数组，数组大小可以指定，也可以不指定

```
int searchArray(int [],int);  
int searchArray(int [10],int);
```

```
int searchArray(int a[],int b){  
    return a[b];  
}  
int searchArray(int a[10],int b){  
    return a[b];  
}
```

参数传递

一维数组作为参数

```
#include<iostream>
using namespace std;
void strcat(char s[],char ct[])
{
    int i=0,j=0;
    while (s[i]!=0)
        i++;
    while (ct[j]!=0)
        s[i++]=ct[j++];
    s[i]='\0';
}
```

字符串连接

参数传递

一维数组作为参数

字符串连接

```
int main (void) {  
    char a[40]="李明";  
    char b[20]="是东南大学学生";  
    strcat(a,b); //实参为数组名  
    cout<<a<<endl; //打印字符数组a  
    return 0;  
}
```

参数传递

多维数组作为参数

多维数组（以二维数组为例）作为函数的参数，参数表的数组参数可以写为

```
int a[][10];  
int a[10][10];
```

行可以省略

但不可以写为：

```
int a[][];  
int a[20][];
```

列一定要给定

参数传递

一维数组以及多维数组的**第一维大小**，**形参、实参**可以**不对应**

```
int a[10];  
int b[5][5];  
  
void f1(int x[6], int y[4][5]) {  
}  
void f2(int x[], int y[4][4]) {  
}
```

```
f1(a, b); //OK  
f2(a, b); //第二个参数错误, b的列是5, y的列是4
```

参数传递

多维数组作为参数

矩阵转置

```
void inverse(int matrix1[3][6], int middle[6][3])
{
    int i, j;
    for (i=0; i<3; i++)
        for (j=0; j<6; j++)
            middle[j][i]=matrix1[i][j];
    return;
}
```

参数传递

多维数组作为参数

矩阵相乘

```
void multi(int middle[6][3], int matrix2[3][4],  
           int result[6][4]) {  
    int i, j, k;  
    for (i=0; i<6; i++) {  
        for (j=0; j<4; j++) {  
            result[i][j] = 0;  
            for (k=0; k<3; k++)  
                result[i][j] +=  
                    middle[i][k] * matrix2[k][j];  
        }  
    }  
    return;  
}
```

参数传递

多维数组作为参数

矩阵输出

```
void output(int result[6][4]) {  
    cout <<"result"<<' \n' ;  
    int i,j;  
    for (i=0;i<6;i++) {  
        for (j=0;j<4;j++)  
            cout <<setw(4)<<result[i][j]<<"    ";  
        cout<<' \n' ;  
    }  
    return;  
}
```

参数传递

多维数组作为参数

完整例子

```
#include<iostream>
using namespace std;
void inverse(int [3][6], int [6][3]); //转置矩阵
void multi(int [6][3], int [3][4], int [6][4]); //矩阵乘法
void output(int [6][4]); //矩阵输出
int main(){
    int middle[6][3], result[6][4];
    int matrix1[3][6]={8,10,12,23,1,3,5,7,9,2,4,6,
        34,45,56,2,4,6};
    int matrix2[3][4]={3,2,1,0,-1,-2,9,8,7,6,5,4};
    inverse(matrix1,middle); //实参为数组名
    multi(middle,matrix2,result);
    output(result);
    return 0;
}
```

练习

下面程序的输出是？

```
#include<iostream>
using namespace std;
void increase(int x) {
    x++;
}
```

```
int c = 0; //全局变量
void main()
{
    c ++;
    increase(c) ;
    cout<<c<<endl;
}
```

函数increase(c)仅改变形参x的值
，不会改变实参c的值

练习

下面程序是否正确？

```
#include<iostream>
using namespace std;
```

```
int f(int x[100]) {
    return x[99];
}
```

函数体访问的数据位置超过实参数组范围

```
void main()
{
    int a[10];
    cout<<f(a)<<endl;
}
```

练习

下面程序是否正确？

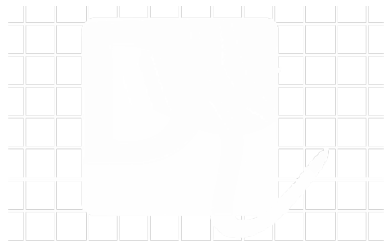
```
#include<iostream>
using namespace std;
```

```
void f(int x[100], int n) {
    for(int i=0; i<n; i++)
        x[i]++;
}
```

此种情况通常需要有二个参数，第二个参数说明数组大小

```
void main()
{
    int a[20];
    f(a,20);
}
```

引用

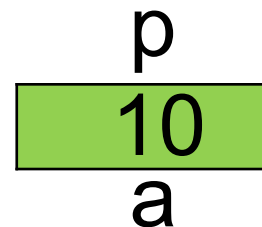


引用

变量的别名

```
int a;  
int & p = a;
```

引用数据类型 & 引用名 = 变量名



变量有两个名字 a, p,
但只有一个空间！

- 引用在声明的同时必须初始化
- 一个引用所引用的对象在初始化后不能改变
- 操作引用和操作所引用的对象一样

引用

变量的别名

```
int a=0, b=1;  
int & p = a; //OK  
int & q; //Error! 必须初始化  
a ++;  
p ++;  
p = b; //OK, 但p仍然是a的引用  
p ++;  
cout<<a<<b;
```

引用类型的函数参数

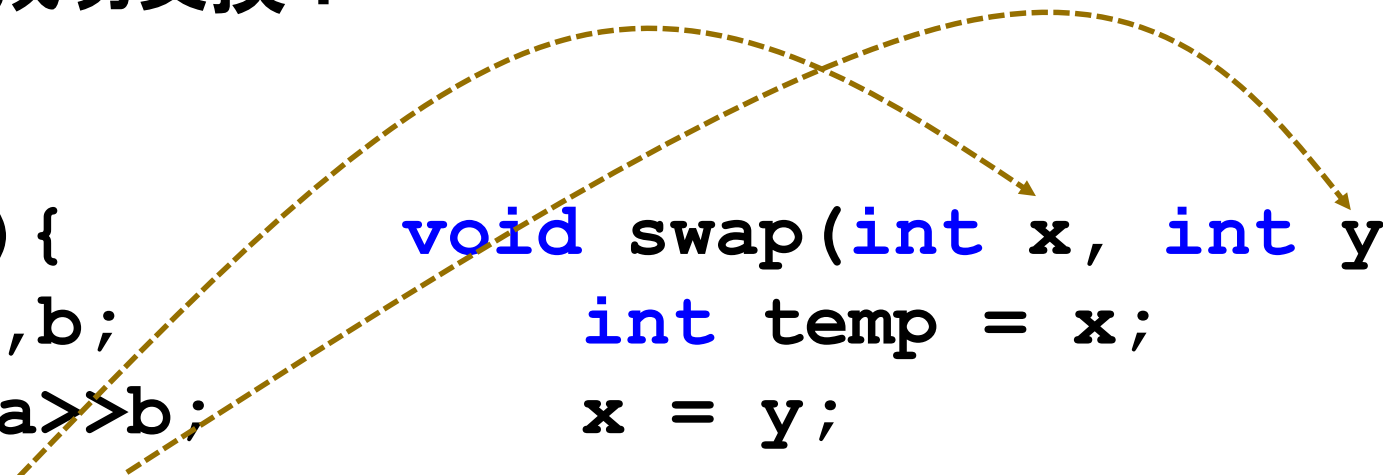
编写函数，实现两个整数的交换

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

引用类型的函数参数

a, b是否能成功交换？

```
int main() {  
    int a, b;  
    cin >> a >> b;  
    swap(a, b);  
    return 0;  
}  
  
void swap(int x, int y) {  
    int temp = x;  
    x = y;  
    y = temp;  
}
```



a, b和x,y是不同的变量, 占用不同的存储空间,
x,y进行了交换, 但a, b没有

引用类型的函数参数

引用做函数的参数

引用形参

- 函数定义的参数表中，名字前加上符号 & 的参数为引用形参。例如

```
void swap(int &a, int &b) ;
```

引用类型的函数参数

引用形参传递过程

```
int main() {  
    int a,b;  
    cin>>a>>b;  
    swap(a,b);  
    return 0;  
}
```

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

引用类型的函数参数

引用形参传递过程

① 生成局部变量a,b

a

b



```
int main() {  
    int a,b;  
    cin>>a>>b;  
    swap(a,b);  
    return 0;  
}
```

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

引用类型的函数参数

引用形参传递过程

a

b



② 控制转移

```
int main() {  
    int a,b;  
    cin>>a>>b;  
    swap(a,b);  
    return 0;  
}
```

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

}

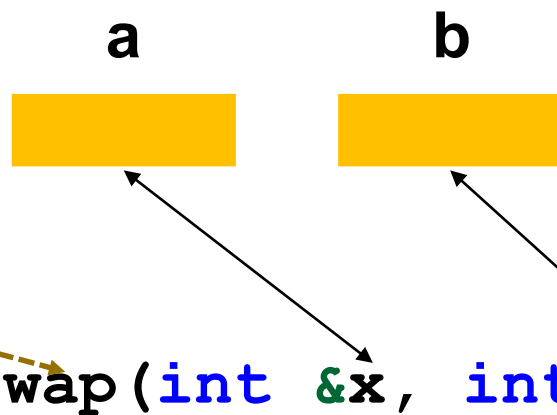
引用类型的函数参数

引用形参传递过程

③ 为变量a,b生成别名x,y

```
int main() {  
    int a,b;  
    cin>>a>>b;  
    swap(a,b);  
    return 0;  
}
```

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



引用类型的函数参数

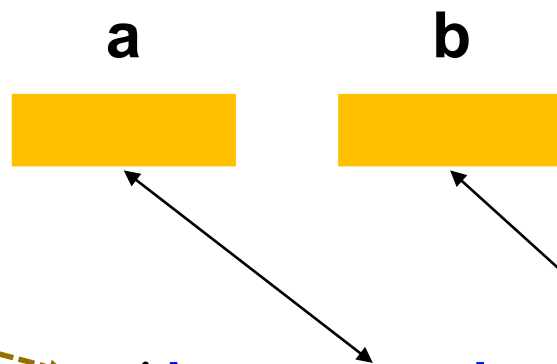
引用形参传递过程

```
int main() {  
    int a,b;  
    cin>>a>>b;  
    swap(a,b);  
    return 0;  
}
```

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

④ 函数执行

对x,y的操作相当于对a,b的操作



引用类型的函数参数

引用形参

- 函数的调用语句中对应于引用形参的实参必须是同一类型的变量，非变量的表达式则不允许

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

```
int main() {  
    int x = 5;  
    int c=add(1, 4*x+2) ; //实参为非变量，不允许！  
    .....  
}
```

引用类型的函数参数

```
void printStar(int k, int n);  
//它所用的两个参数均为赋值参数
```

```
void swap(int &x, int &y);  
//它所用的两个参数均为引用参数
```

```
int myFunc(int a, float &b);  
//它所用的第一个参数为赋值参数，另一个为引用参数
```

返回值为引用类型

函数返回类型为普通类型

```
int a = 0;
int main() {
    int b = increment();
    increment() = 1; // Error!
    return 0;
}
```

```
int increment () {
    a++;
    return a;
}
```

函数返回的是一个数值(a的值), 不能作为左值

2 = 1 合理吗?

返回值为引用类型

函数返回类型为引用

```
int a = 0;
int main() {
    int b = increment();
    increment() = 1; // OK!
    return 0;
}
```

```
int & increment () {
    a++;
    return a;
}
```

返回的就是a本身，可以作为左值！

a = 1 合理吧

引用举例

```
int a = 0;  
int main() {  
    a++;  
    ++a;
```

后++操作符返回的是
加之前变量的值

前++操作符返回的是
加完之后变量的引用

`a++ = 1;` // Error, 相当于 `2=1`

`++a = 1;` // OK, 相当于 `a=1`

`++(a++);` // Error, 相当于 `++1`

`(++a)++;` // OK, 相当于 `a++`

`return 0;`

}

练习

下面程序输出结果是？

```
void increment(int &a) {  
    a++;  
}
```

```
int main() {  
    int x = 5;  
    increment(x);  
    cout<<x<<endl;  
    increment(x);  
    cout<<x<<endl;  
}
```

函数每次调用，都相当于对x
进行操作

练习

下面程序是否正确？

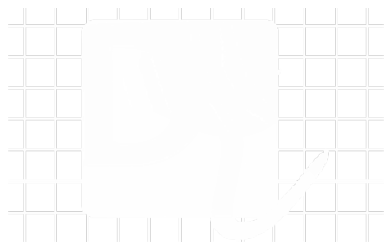
```
int & f() {  
    int a = 0;  
    a ++;  
    return a;  
}
```

```
int main() {  
    int x = 5;  
    f() = x;  
    cout<<x<<endl;  
}
```

a是局部变量，函数f()执行后局部变量会消失，因此不能作为左值



函数的嵌套与递归



函数嵌套

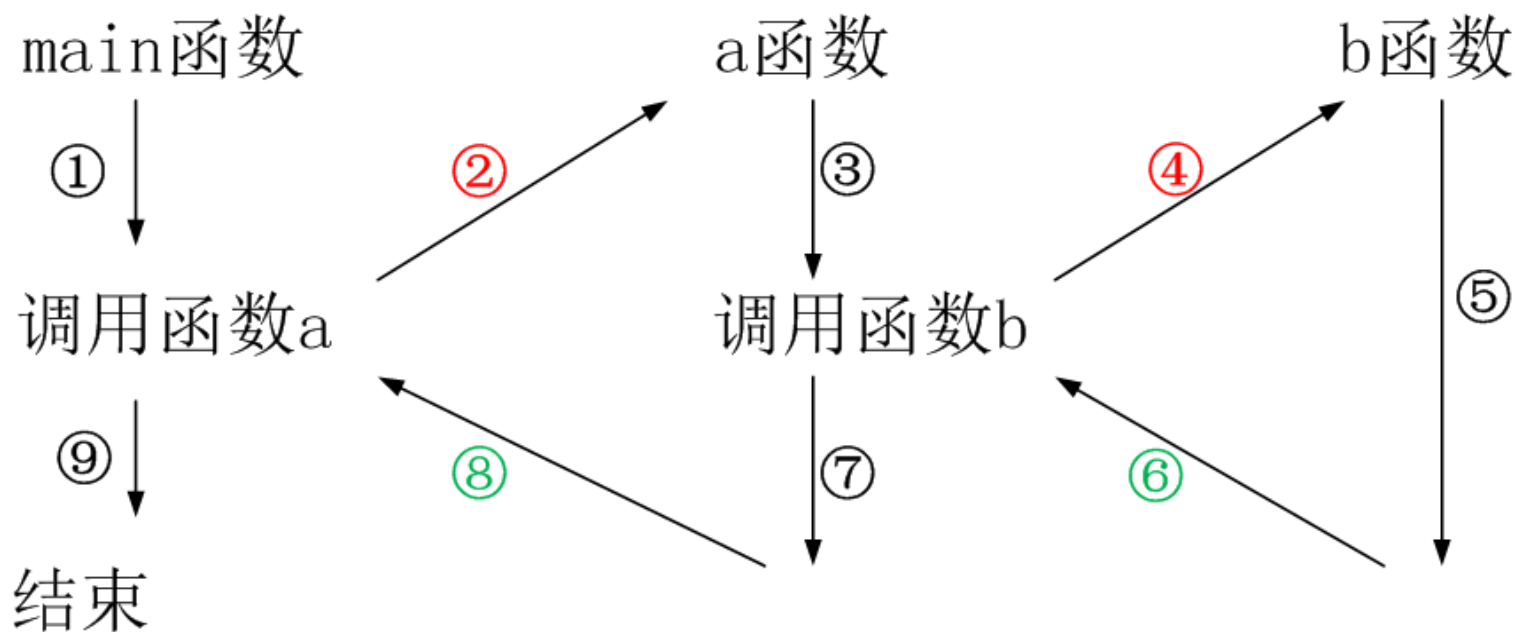
main函数调用a, a又调用b

```
int b(int y){  
    return y+1;  
}  
  
int a(int x){  
    int x1 = b(x);  
    return x1;  
}
```

```
int main() {  
    int m;  
    m = a(3);  
    return 0;  
}
```

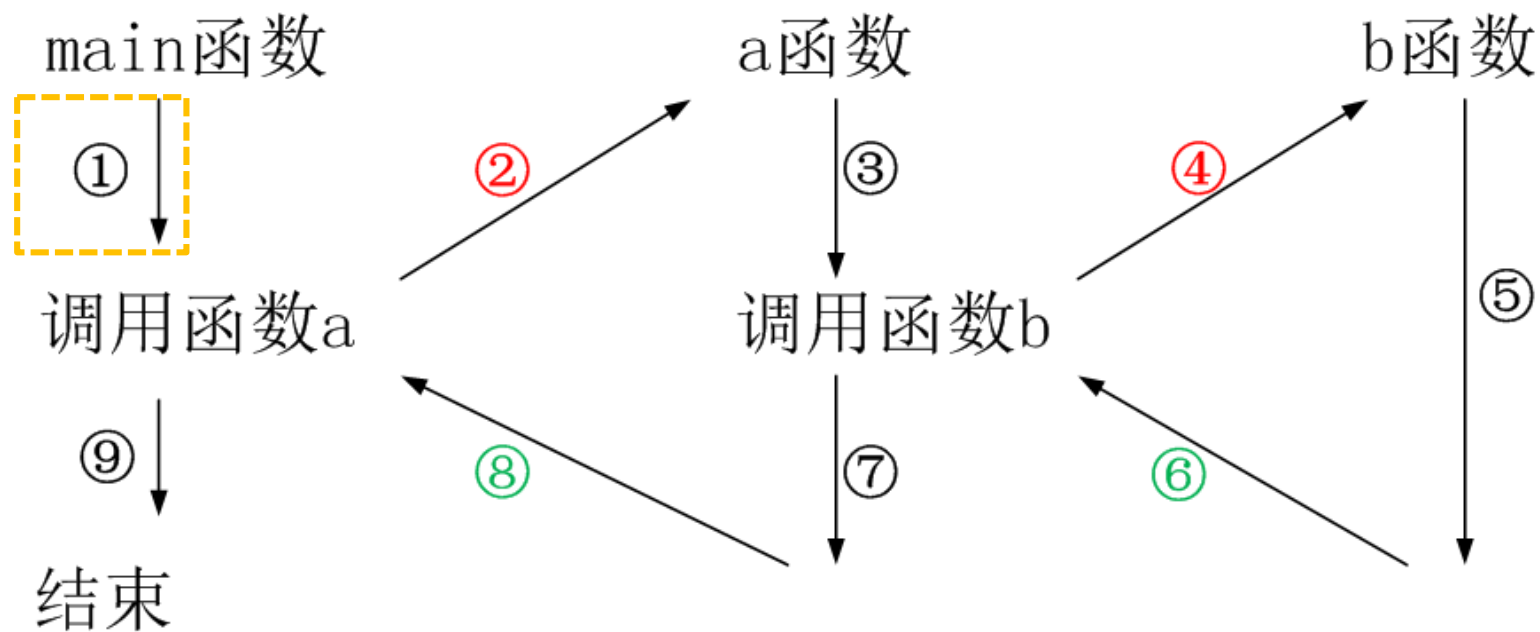
函数嵌套

main函数调用a, a又调用b



每一次嵌套都要控制转移, 需要保护“现场”、占用内存空间, 因此嵌套层数不能太多

函数嵌套



函数嵌套

main函数开始执行



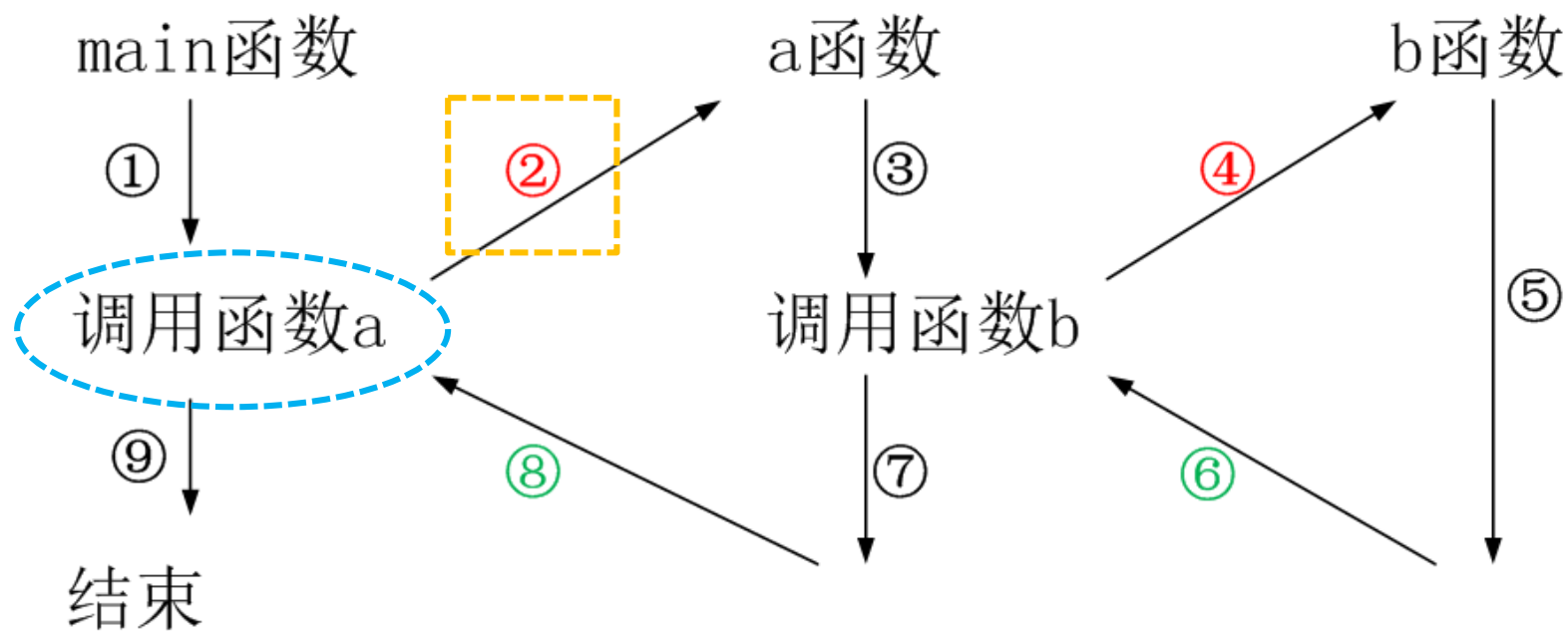
```
int main() {  
    int m;  
    m = a(3);  
    return 0;  
}
```

内存状态

main函数的变量

main函数参数(无)

函数嵌套



函数嵌套

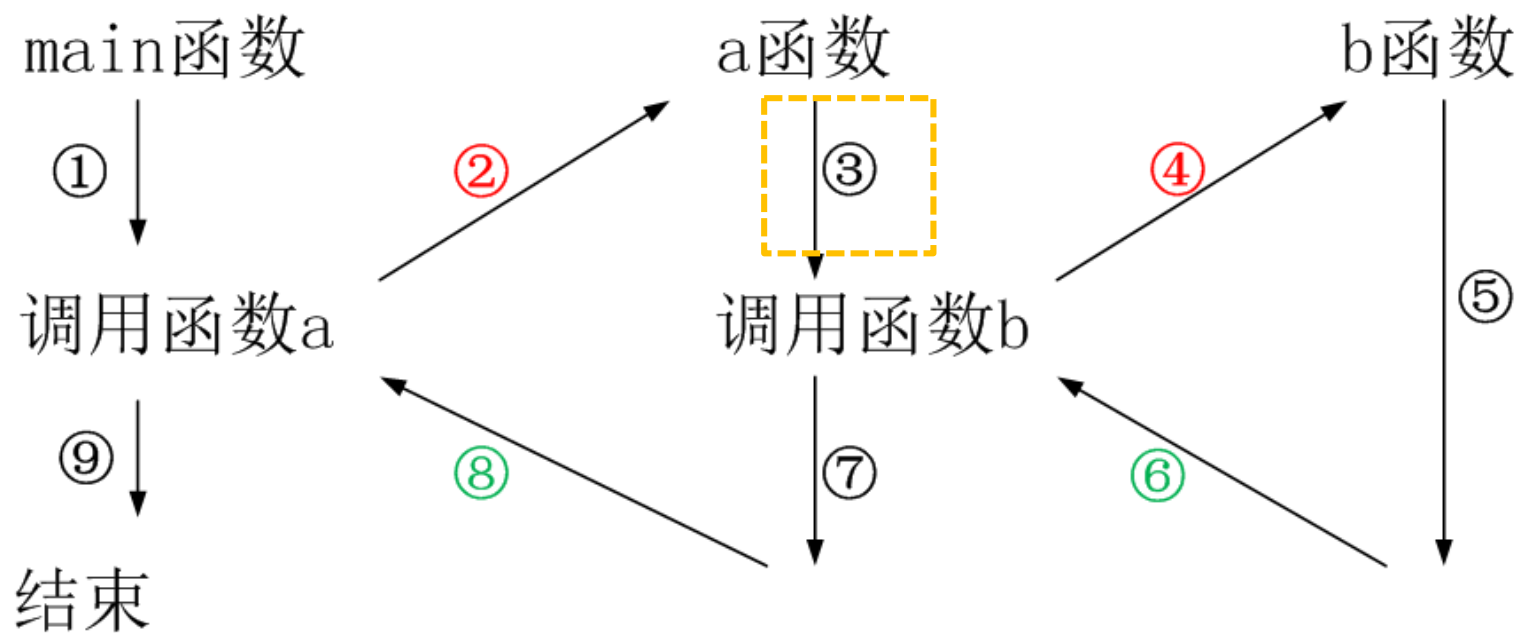
调用函数a

```
int main() {  
    int m;  
    m = a(3);  
    return 0;  
}
```

内存状态

函数a的返回地址
main函数的现场 (CPU寄存器等)
main函数的变量
main函数参数(无)

函数嵌套



函数嵌套

执行函数a



```
int a(int x) {  
    int x1 = b(x);  
    return x1;  
}
```

内存状态

函数a的变量 (x1)

函数a的参数 (x)

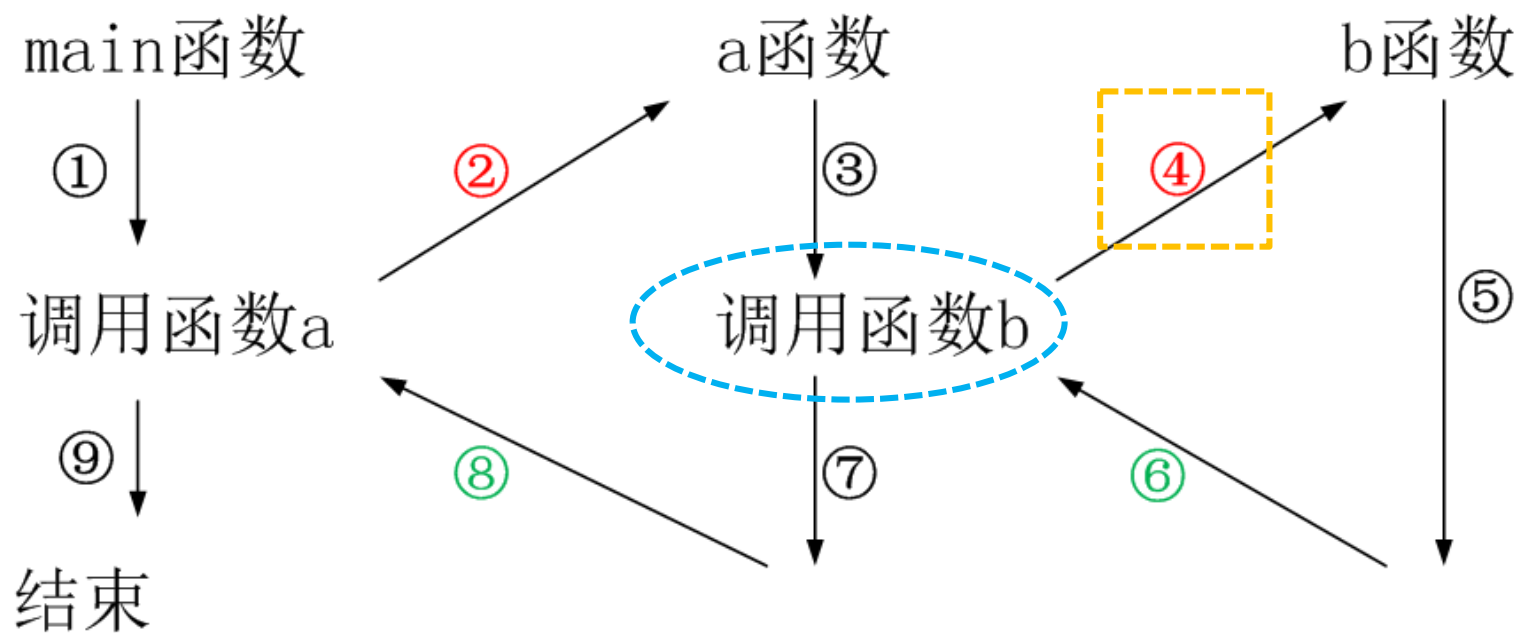
函数a的返回地址

main函数的现场
(CPU寄存器等)

main函数的变量

main函数参数(无)

函数嵌套



函数嵌套

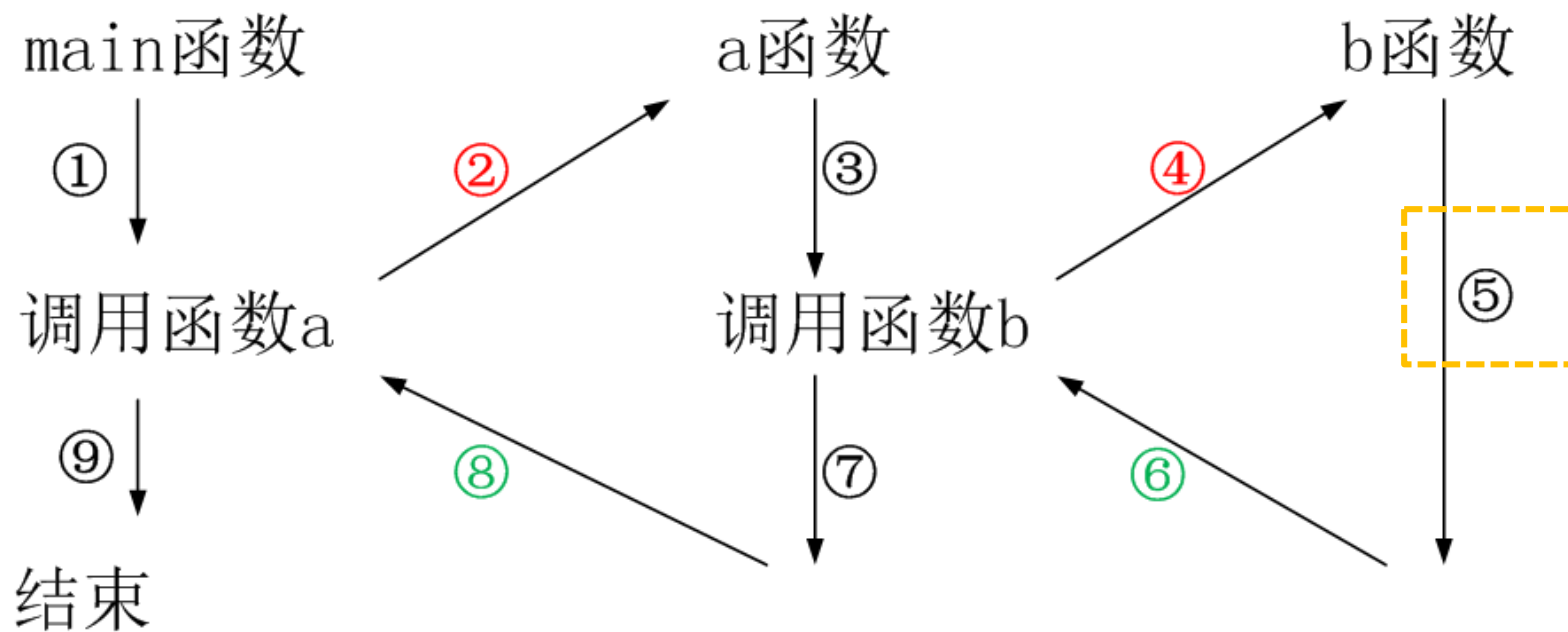
调用函数b

```
int a(int x) {  
    int x1 = b(x);  
    return x1;  
}
```

内存状态

函数b的返回地址
函数a的现场 (CPU寄存器等)
函数a的变量 (x1)
函数a的参数 (x)
函数a的返回地址
main函数的现场 (CPU寄存器等)
main函数的变量
main函数参数(无)

函数嵌套



函数嵌套

内存状态

执行函数b



```
int b(int y) {  
    return y+1;  
}
```

函数b的变量(无)

函数b的参数(y)

函数b的返回地址

函数a的现场
(CPU寄存器等)

函数a的变量(x1)

函数a的参数(x)

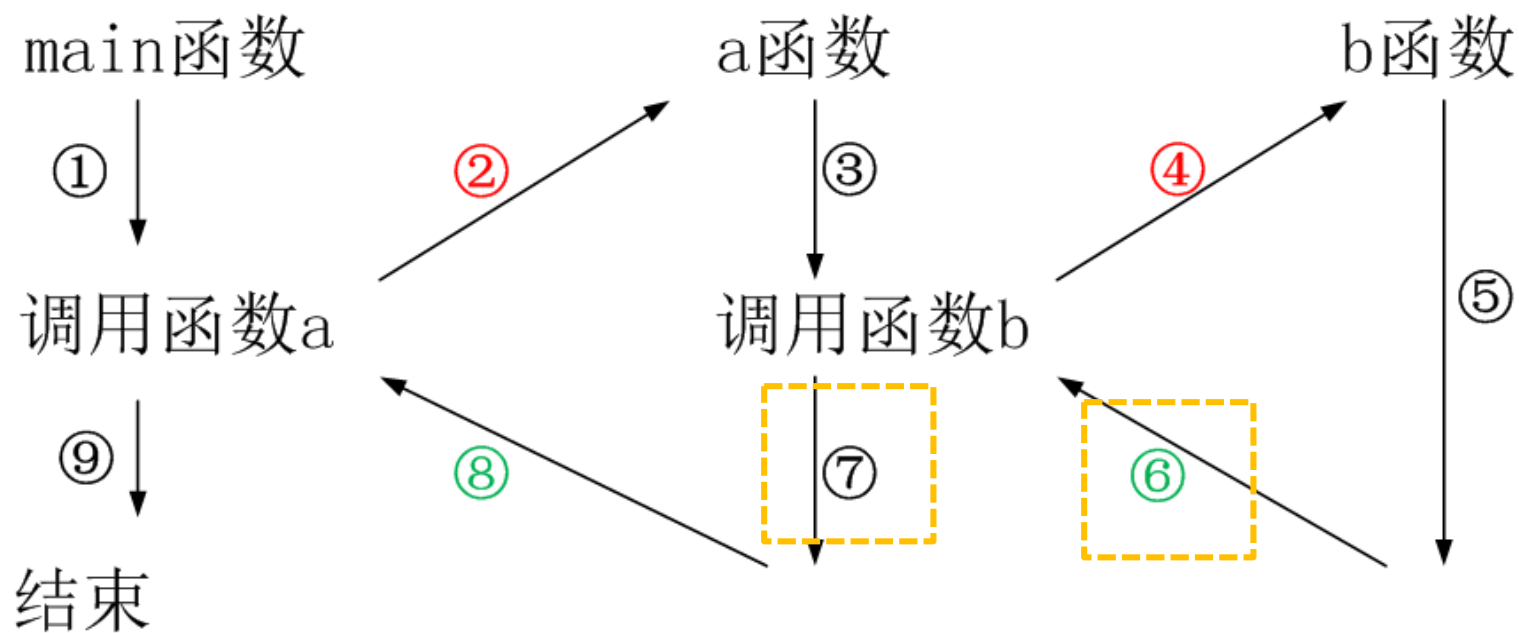
函数a的返回地址

main函数的现场
(CPU寄存器等)

main函数的变量

main函数参数(无)

函数嵌套



函数嵌套

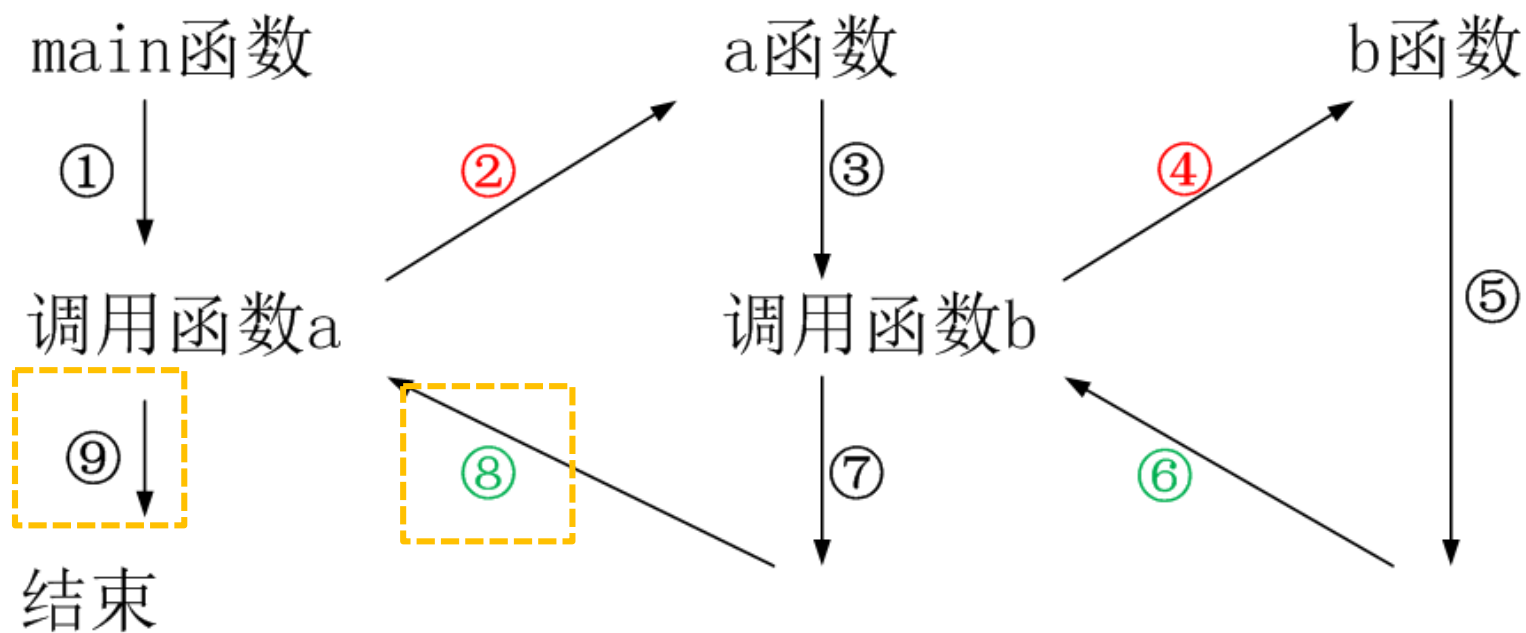
函数b返回

```
int a(int x){  
    int x1 = b(x);  
    return x1;  
}
```

内存状态

函数a的变量 (x1)
函数a的参数 (x)
函数a的返回地址
main函数的现场 (CPU寄存器等)
main函数的变量
main函数参数(无)

函数嵌套



函数嵌套

函数a返回

```
int main() {  
    int m;  
    m = a(3);  
    return 0;  
}
```

内存状态

main函数的变量
main函数参数(无)

函数递归

函数调用自己

```
int factorial(int n) {  
    if (n==0)  
        return 1;  
    else  
        return n*factorial(n-1);  
}
```

函数递归

factorial(4);

```
int factorial(int n){  
    if (n==0)  
        return 1;  
    else  
        return n*factorial(n-1);  
}
```

函数递归

```
int factorial(int n){  
    if (n==0)  
        return 1;  
    else  
        return n*factorial(n-1);  
}
```

factorial(4);

n=4

Returns 4*factorial(3)

函数递归

```
int factorial(int n){  
    if (n==0)  
        return 1;  
    else  
        return n*factorial(n-1);  
}
```

factorial(4);

n=4

Returns 4*factorial(3)

n=3

Returns 3*factorial(2)

函数递归

```
int factorial(int n){  
    if (n==0)  
        return 1;  
    else  
        return n*factorial(n-1);  
}
```

factorial(4);

n=4

Returns 4*factorial(3)

n=3

Returns 3*factorial(2)

n=2

Returns 2*factorial(1)

函数递归

```
int factorial(int n){  
    if (n==0)  
        return 1;  
    else  
        return n*factorial(n-1);  
}
```

factorial(4);

n=4

Returns 4*factorial(3)

n=3

Returns 3*factorial(2)

n=2

Returns 2*factorial(1)

n=1

Returns 1*factorial(0)

函数递归

```
int factorial(int n){  
    if (n==0)  
        return 1;  
    else  
        return n*factorial(n-1);  
}
```

factorial(4);

n=4

Returns 4*factorial(3)

n=3

Returns 3*factorial(2)

n=2

Returns 2*factorial(1)

n=1

Returns 1*factorial(0)

n=0

Returns 1

函数递归

```
int factorial(int n){  
    if (n==0)  
        return 1;  
    else  
        return n*factorial(n-1);  
}
```

factorial(4);

n=4

Returns 4*factorial(3)

n=3

Returns 3*factorial(2)

n=2

Returns 2*factorial(1)

n=1

Returns 1*factorial(0)

1

函数递归

```
int factorial(int n){  
    if (n==0)  
        return 1;  
    else  
        return n*factorial(n-1);  
}
```

factorial(4);

n=4

Returns 4*factorial(3)

n=3

Returns 3*factorial(2)

n=2

Returns 2*factorial(1)

1

函数递归

```
int factorial(int n){  
    if (n==0)  
        return 1;  
    else  
        return n*factorial(n-1);  
}
```

factorial(4);

n=4

Returns 4*factorial(3)

n=3

Returns 3*factorial(2)

2



函数递归

```
int factorial(int n){  
    if (n==0)  
        return 1;  
    else  
        return n*factorial(n-1);  
}
```

factorial(4);

n=4

Returns 4*factorial(3)

6



函数递归

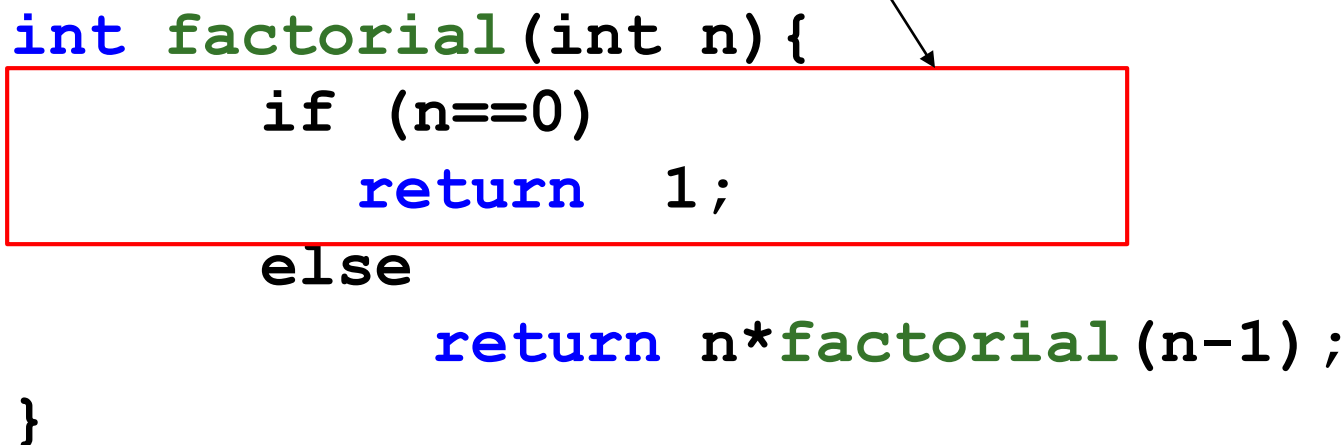
~~factorial(4);~~

24

```
int factorial(int n){  
    if (n==0)  
        return 1;  
    else  
        return n*factorial(n-1);  
}
```

函数递归

必须要有终止条件，否则无限递归



```
int factorial(int n) {  
    if (n==0)  
        return 1;  
    else  
        return n*factorial(n-1);  
}
```

The diagram illustrates a recursive function `factorial`. A red box highlights the base case `if (n==0) return 1;`. An arrow points from the text '必须要有终止条件，否则无限递归' to this base case. The recursive call `return n*factorial(n-1);` is shown below the base case.

递归和嵌套本质相同，层数不能太多

函数递归

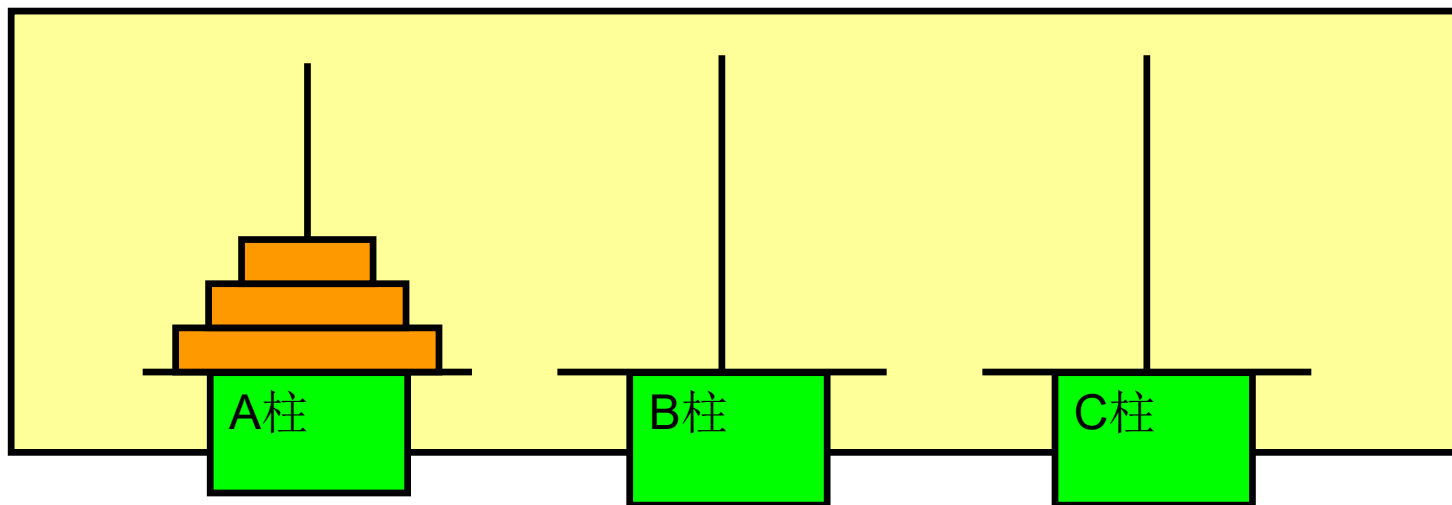
斐波那契数列(从第三个数起, 后一个数等于前面两个数之和): 1、1、2、3、5、8、13、21、34

```
long int fun(int n) {  
    if (n==1 || n==2)  
        return 1;  
    else  
        return fun(n-1) + fun(n-2);  
}
```

函数递归

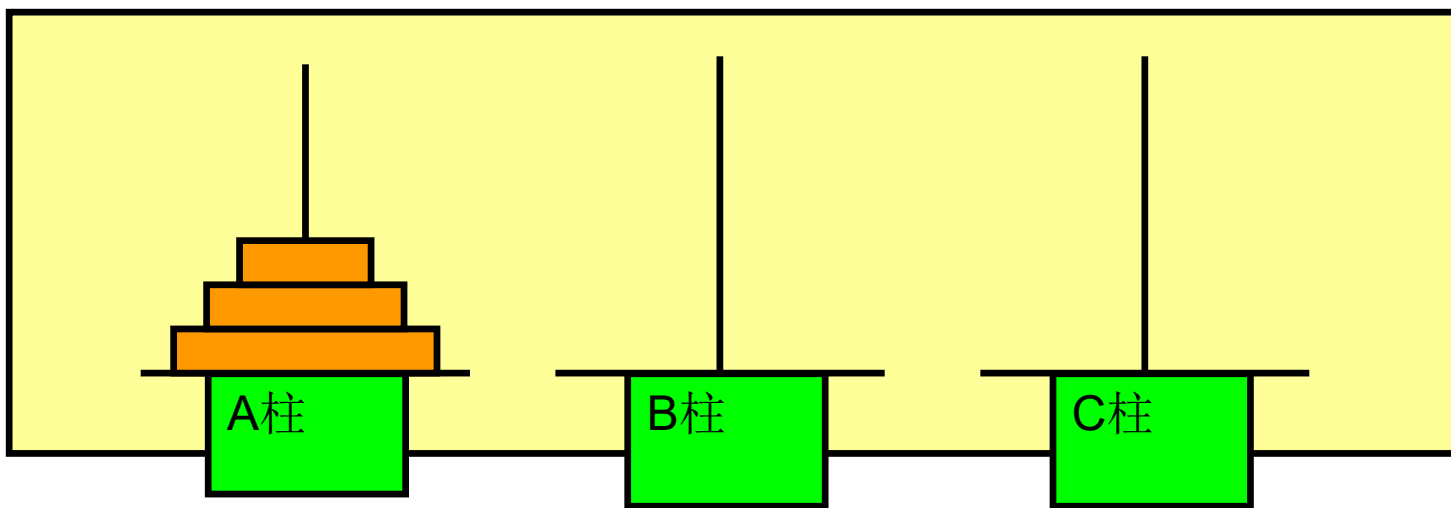
汉诺塔：有三个立柱A、B、C，在A柱上穿有大小不等的圆盘64个，较大的圆盘在下，较小者在上。要求借助于B柱将A柱上的64个圆盘移到C柱，规则为：

- (1) 每次只能把一个柱上最上面的圆盘移至另一个柱的最上面；
- (2) 每个柱上总保持较大的圆盘在下，较小者在上。



函数递归

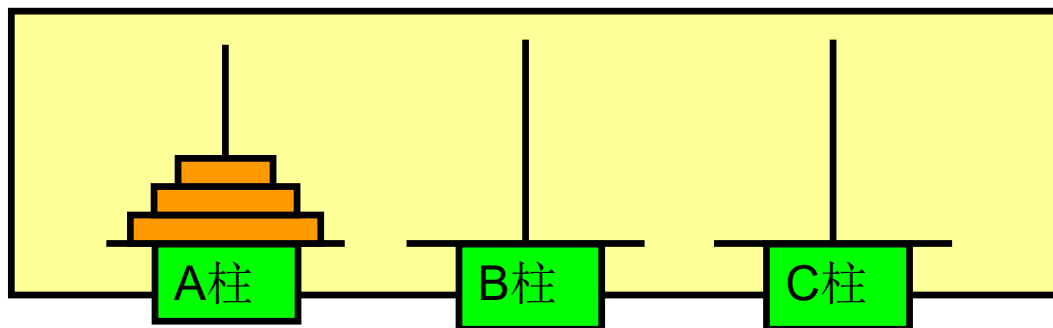
编制程序, 实现将任意 n 个圆盘从A柱借助于B柱移到C柱, 并显示出全部移动过程



函数递归

- ❑ A柱只有一个盘子的情况
 - A柱→C柱
- ❑ A柱有两个盘子的情况
 - 小盘 A柱→B柱, 大盘 A柱→C柱, 小盘 B柱→C柱
- ❑ A柱有n个盘子的情况
 - 看成上面n-1个盘子和最下面第n个盘子的情况
 - ❑ n-1个盘子 A柱→B柱; 第n个盘子A柱→C柱, n-1个盘子 B柱→C柱

问题转化成搬动n-1个盘子的问题, 同样, 将n-1个盘子看成上面n-2个盘子和下面第n-1个盘子的情况, 进一步转化为搬动n-2个盘子的问题,, 类推下去, 一直到最后成为搬动一个盘子的问題



函数递归

- ❑ 总任务(圆盘数为n的任务):
 - 把A柱上的n个圆盘, 借助于B柱, 按规则移到C柱上(移动规则: 一次移一片, 大片不可压小片)
 - 靠调用自定义函数hanoi来完成

hanoi(n,'A','B','C');

//n-1个盘子A->B

hanoi(n-1,'A','C','B');

第n个盘子 A->C

//n-1个盘子B->C

hanoi(n-1,'B','A','C');

//n-2个盘子A->C

hanoi(n-2,'A','B','C');

第n-1个盘子 A->B

//n-2个盘子C->B

hanoi(n-2,'C','A','B');

//n-2个盘子B->A

hanoi(n-2,'B','C','A');

第n-1个盘子 B->C

//n-2个盘子A->C

hanoi(n-2,'A','B','C');

函数递归

递归函数hanoi的定义

```
void hanoi(int n, char source, char temp, char
target) {
    if (n==1)
        move(source, target) ;
    else{
        //将n-1个盘子搬到中间柱
        hanoi (n-1, source, target, temp) ;
        //将最后一个盘子搬到目标柱
        move (source, target) ;
        //将n-1个盘子搬到目标柱
        hanoi (n-1, temp, source, target) ;
    }
}
```

函数递归

//将最后一个盘子搬到目标柱（打印到屏幕）

函数move的定义

```
void move(char source, char target)
{
    cout<<source<<"="<<target<<endl;
}
```

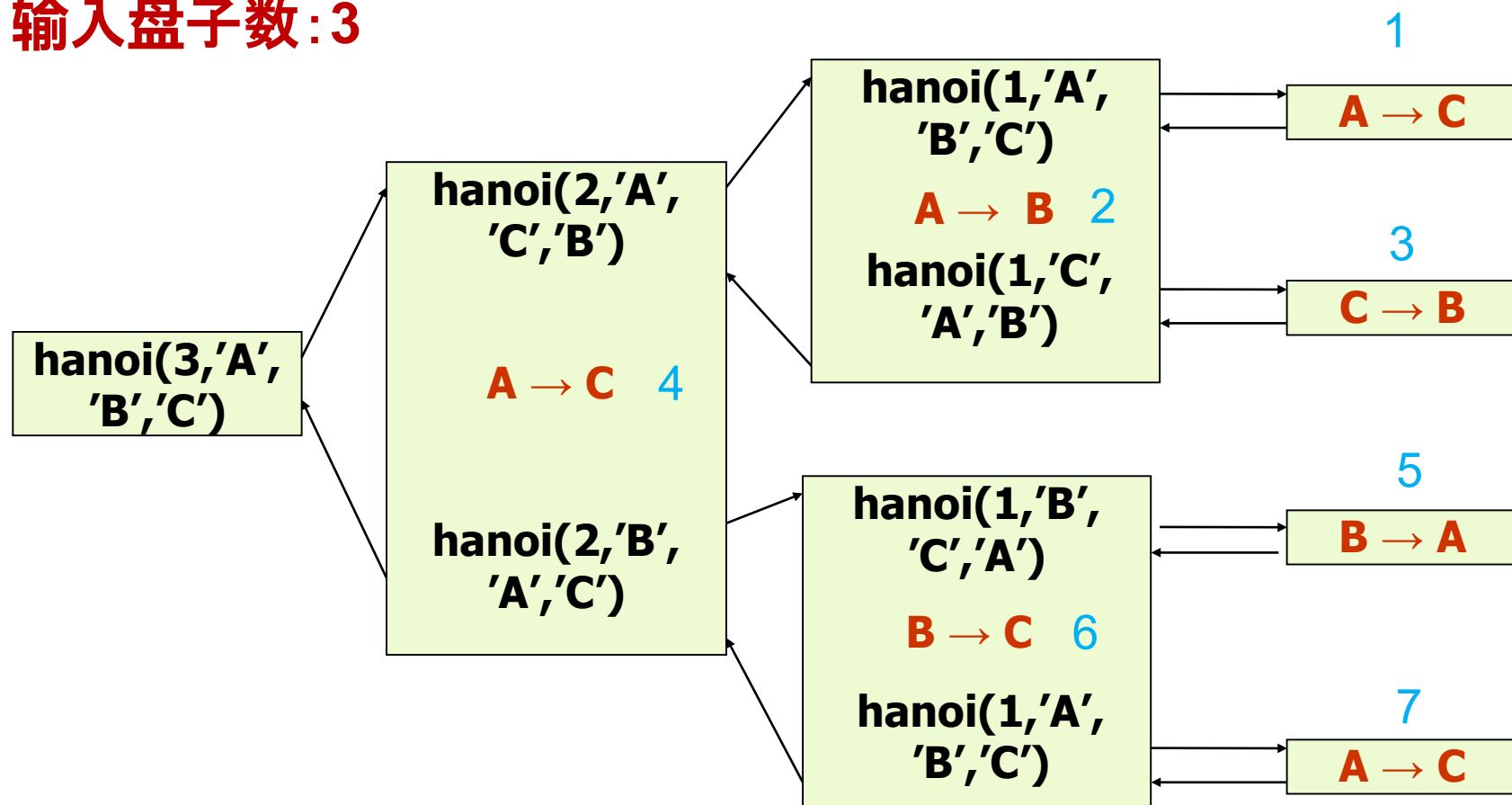
函数递归

```
#include<iostream>
using namespace std;
void move(char, char) ;
void hanoi(int, char, char, char) ;
int main() {
    int n;
    cout<<"输入盘子数: "<<endl;
    cin>>n;
    hanoi(n, 'A', 'B', 'C') ;
    return 0;
}
```

函数递归

汉诺塔程序执行框图

输入盘子数: 3



函数递归

快速排序

基本思想：

- ❑ 枢值归位
 - 枢值是指在排序序列中指定的某个值，一般是排序序列的中间元素的值
 - 在排序过程中，可以多次指定枢值，并放到排序序列（或者子序列）的末尾位置（亦可以放在首位置）
- ❑ 二分有序
 - 将元素序列以枢值为界，分为左右两个部分，左边的值都小于枢值，而右边的值都大于枢值

函数递归

快速排序

(1) 选择中间元素作为枢值

2 1 10 8 7 5 12



(2) 将枢值换到最后一个位置

2 1 10 12 7 5 8



函数递归

快速排序

(3) 交换元素,使得左边元素小于枢值, 右边元素大于枢值

2

1

10

12

7

5

8

左下标

右移左下标, 直到遇到大于枢值的元素

右下标

左移右下标, 直到遇到小于枢值的元素

函数递归

快速排序

(3) 交换元素,使得左边元素小于枢值, 右边元素大于枢值

2

1

10

12

7

5

8



左下标



右下标

右移左下标, 直到遇到大于枢值的元素

左移右下标, 直到遇到小于枢值的元素

函数递归

快速排序

(3) 交换元素,使得左边元素小于枢值, 右边元素大于枢值

2

1

5

12

7

10

8



左下标



右下标

交换左右下标对应的元素

函数递归

快速排序

(3) 交换元素,使得左边元素小于枢值, 右边元素大于枢值

2

1

5

12

7

10

8



左下标 右下标

右移左下标, 直到遇到大于枢值的元素

左移右下标, 直到遇到小于枢值的元素

函数递归

快速排序

(3) 交换元素,使得左边元素小于枢值, 右边元素大于枢值

2

1

5

7

12

10

8



左下标 右下标

交换左右下标对应的元素

函数递归

快速排序

(3) 交换元素,使得左边元素小于枢值, 右边元素大于枢值

2

1

5

7

12

10

8



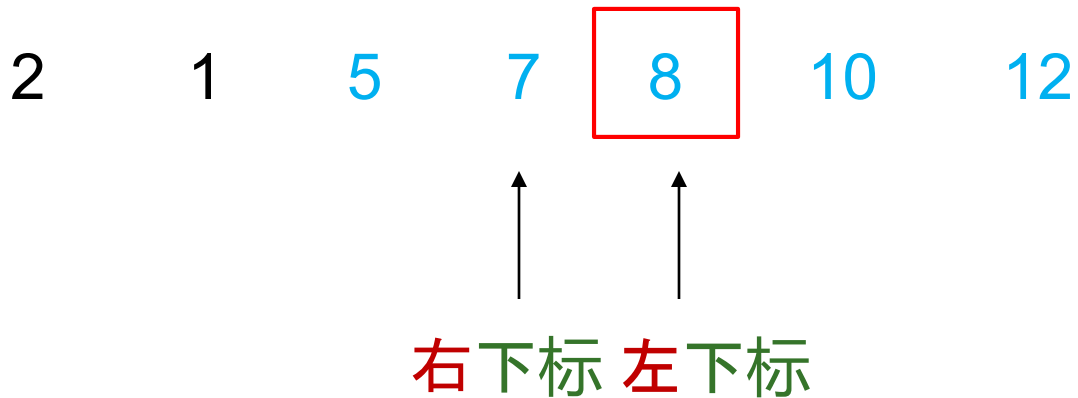
右下标 左下标

左右下标继续移动, 出现交叉, 交换过程停止

函数递归

快速排序

(3) 交换元素,使得左边元素小于枢值, 右边元素大于枢值

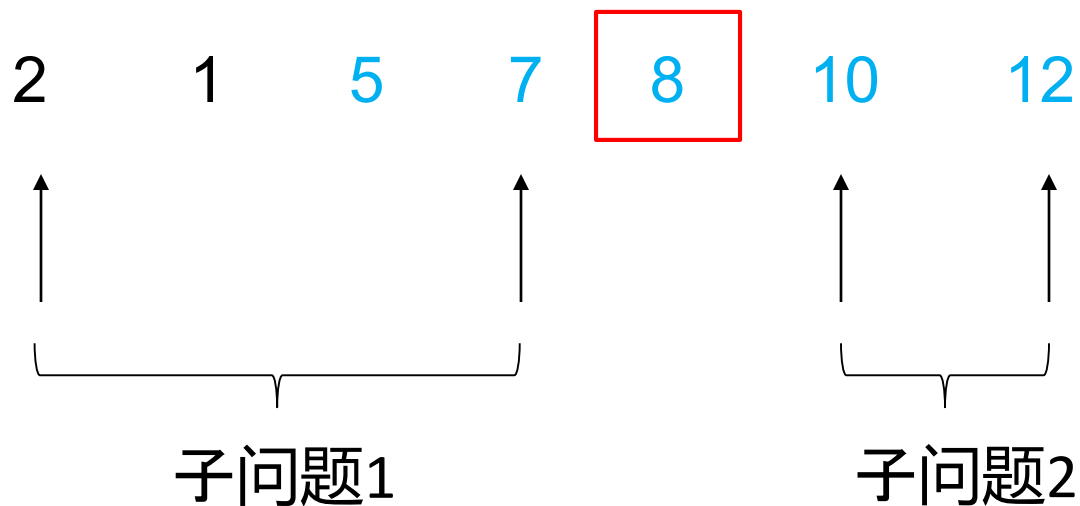


枢值和左下标对应的元素交换, 完成目标(左边的元素都小于枢值, 右边的元素都大于枢值)

函数递归

快速排序

(3) 交换元素,使得左边元素小于枢值, 右边元素大于枢值



以枢值为界限, 将元素分成左、右两组, 递归排序

函数递归

快速排序

```
#include<iostream>
#include<cstdlib>
#include<ctime>
#include<iomanip>
using namespace std;
const int n = 10;
void swap (int &, int &);
int findpivot(int [],int,int);
int partition(int [],int,int,int);
void quickSort(int [],int,int);
```

函数递归

快速排序

```
int main() {
    int b[n];
    srand( (unsigned) time(NULL) );
    cout<<"随机数: "<<endl;
    for(int i=0;i<n;i++){
        b[i] = rand()%100;
        cout<<setw(5)<<b[i];
        if( (i+1)%10==0)
            cout<<endl;
    }
    quickSort(b, 0, n-1);
}
```

函数递归

快速排序

```
    cout<<endl<<"排序后: "<<endl;
    for(int i=0;i<n;i++){
        cout<<setw(5)<<b[i];
        if((i+1)%10==0)
            cout<<endl;
    }
    return 0;
}
```

函数递归

快速排序

函数swap定义

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

函数递归

快速排序

函数findpivot定义

```
int findpivot(int a[], int i, int j)
{ //将中间元素设置为枢值
    return (i+j)/2;
}
```

函数递归

快速排序

函数partition定义

```
int partition(int a[], int l, int r, int
    pivot) //返回枢值的下标
{
    do{
        while (a[++l] < pivot) ;
        while (r && a[--r] > pivot) ;
        swap(a[l], a[r]) ;
    } while (l < r) ;
    swap(a[l], a[r]) ;
    return l ;
}
```

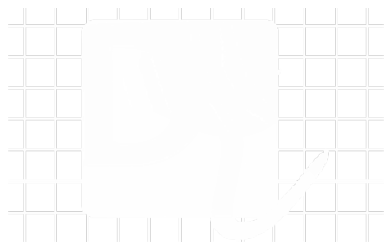
函数递归

快速排序

函数quickSort定义

```
void quickSort(int a[], int i, int j)
{
    int pivotindex = findpivot(a, i, j);
    swap(a[pivotindex], a[j]);
    int k = partition(a, i-1, j, a[j]);
    swap(a[k], a[j]);
    if ((k-i) > 1)
        quickSort(a, i, k-1);
    if ((j-k) > 1)
        quickSort(a, k+1, j);
}
```

函数与运算符重载



函数重载

函数重载实际上是不同的函数采用同一名字

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

```
float abs(float f) {  
    if (f<0)  
        f=-f;  
    return f;  
}
```

```
double abs(double d) {  
    if (d<0)  
        return -d;  
    return d;  
}
```

三个函数都是求绝对值，
采用同一个函数名，更符合人们的习惯

函数重载

函数重载必须满足下列条件之一

- ❑ 参数表中至少有一对参数类型不同
- ❑ 参数表中参数个数不同

```
void printStar();  
void printStar(int);  
void printStar(int, int);
```

符合重载规则

函数重载

函数重载必须满足下列条件之一

- ❑ 参数表中至少有一对参数类型不同
- ❑ 参数表中参数个数不同

```
float add(int ,float);  
int add(int ,float);
```

仅返回类型不同,
不能进行重载

```
add(1 ,2.0);
```

无法区分调用哪个函数

函数重载

函数重载必须满足下列条件之一

- ❑ 参数表中至少有一对参数类型不同
- ❑ 参数表中参数个数不同

```
void print(double);  
void print(double&);
```

引用参数类型不能
作为重载条件

```
double a = 2.0;  
print(a);
```

无法区分调用哪个函数

函数重载

函数重载必须满足下列条件之一

- ❑ 参数表中至少有一对参数类型不同
- ❑ 参数表中参数个数不同

```
int max(int a, int b);  
int max(int a, int b, int c=0);
```

函数重载不能引起二义性

```
max(1, 2);
```

无法区分调用哪个函数

函数重载

函数重载必须满足下列条件之一

- ❑ 参数表中至少有一对参数类型不同
- ❑ 参数表中参数个数不同

```
int abs(int a){return (a>0?a:-1);}  
float abs(float a){return (a>0?a:-1);}
```

函数重载正确, 函数调用也可能产生错误

```
double a = 1.0;  
abs(a);  
abs(3.0);  
abs(3);
```

错误, 无法区分调用哪个函数

正确, 调用 int abs(int a);

运算符重载

C++的运算符只支持基本类型

```
int main() {  
    int i=0;  
    double j=2.0;  
    char c='a';  
    cout<<i<<j<<c<<endl;  
    i++;  
    j+i;  
    c&&'k';  
    return 0;  
}
```

运算符重载

C++的运算符只支持基本类型

```
struct student {  
    char name[20];  
    int age;  
    int height;  
};
```

```
int main() {  
    student s1={"Li", 18, 180};  
    s1+1; //Error!  
    return 0;  
}
```


C++运算符不能作用于student
类型的变量

运算符重载

运算符重载让C++运算符支持**非基本类型**运算

关键字operator, 加上要重载的运算符

两个参数, 左边为 student 类型, 右边为 int 类型



```
student operator + (student a, int b) {  
    a.age += b;  
    return a;  
}
```

对 '+' 运算符进行重载, 使其支持student和int类型相加

运算符重载

运算符重载让C++运算符支持**非基本类型**的变量

```
student operator + (student a, int b) {  
    a.age += b;  
    return a;  
}
```

```
int main() {  
    student s1={"Li", 18, 180};  
    s1+1; //OK!  
    return 0;  
}
```

s1+1将返回一个student类型的变量
，其age为19

运算符重载

可以重载的运算符几乎包含了C++的全部运算符集，C++语言规定，大多数运算符都可以重载

- ❑ 单目运算符

- -, ~, !, ++, --, new, delete

- ❑ 双目运算符

- +, -, *, /, %, &, |, ^, <<, >>, ==, !=, <, >, <=, >=, ^=, &=, |=, >>=, <<= 等

- ❑ 例外的是: 限定符., ::, 条件运算符?:, 取长度运算符 sizeof

不能创建新的运算符

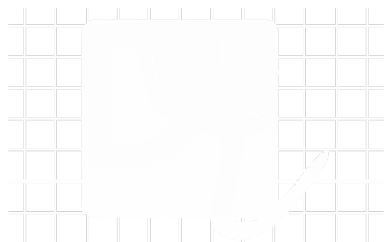
运算符重载

为了区别前缀++和后缀++, C++语言规定, 在后缀++的重载函数的原型参数表中增加一个int 型的无名参数

```
student &operator ++ (student &a) {  
    a.age ++;  
    return a;  
} //前缀++
```

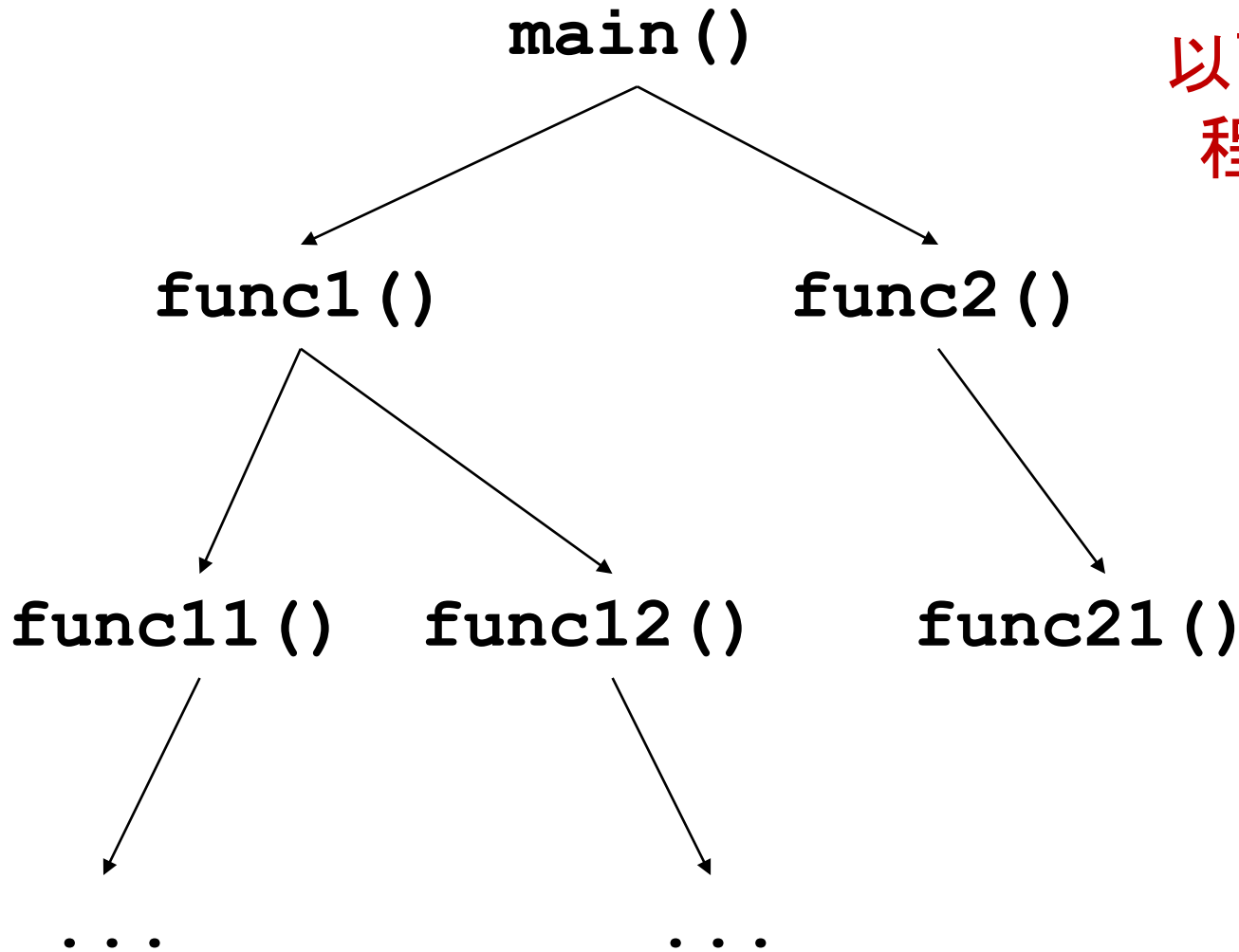
```
student operator ++ (student &a, int) {  
    student s=a;  
    s.age ++;  
    return s;  
} //后缀++
```

结构化程序设计



结构化程序设计

以函数为核心的
程序设计思想



变量作用域和生存期

程序作用域： 程序的作用范围，即所有文件的每个角落

变量作用域： 变量在程序中可见的范围

```
int main() {  
    int a = 2;  
    func1();  
    return 0;  
}
```

a的作用域是整个main()函数

a在func1()中不可见，b在main()中不可见

```
void func1() {  
    int b = 2;  
    cout<<b;  
}
```

b的作用域是整个func1()函数

变量作用域和生存期

程序生存期： 从main开始执行到main函数结束

变量生存期： 程序执行过程中变量存在的时间段

```
int main() {  
    int a = 2;  
    func1();  
    return 0;  
}
```

main() 执行时，a的生存期时
从a的定义开始，到main执行
结束

b的生存期比a的生存期短

```
void func1() {  
    int b = 2;  
    cout<<b;  
}
```

func1() 被调用时，b的生存
期从b的定义开始，到func1
函数结束

变量作用域和生存期

局部变量：函数内部定义的变量（包括函数的形参），作用域是整个函数，生存期从变量定义开始，到函数执行结束

```
int main() {  
    int a = 2;  
    func1();  
    return 0;  
}
```

a是main()函数的局部变量

```
void func1(int c) {  
    int b = 2;  
    cout<<b;  
}
```

b、c是函数func1()的局部变量

变量作用域和生存期

局部变量：函数内部定义的变量（包括函数的形参），作用域是整个函数，生存期从变量定义开始，到函数执行结束

```
int main() {  
    func1();  
    func1();  
    return 0;  
}  
  
void func1(int c) {  
    int b = 2;  
    cout<<b;  
}
```

函数可能被多次调用，每次调用时，局部变量都重新生成！



变量作用域和生存期

全局变量：函数外部定义的变量，作用域是整个程序，生存期从程序开始到程序结束

```
int a = 1;
void func1();
int main() {
    a++;
    func1();
    cout<<a;
    return 0;
}
void func1() {
    a++;
}
```

a是全球变量

a在main()函数可见

全局变量在程序开始时生成，生存期时整个程序执行过程

a在func1()函数也可见

变量作用域和生存期

变量同名：不同作用域的变量名相同，优先访问作用域小的变量

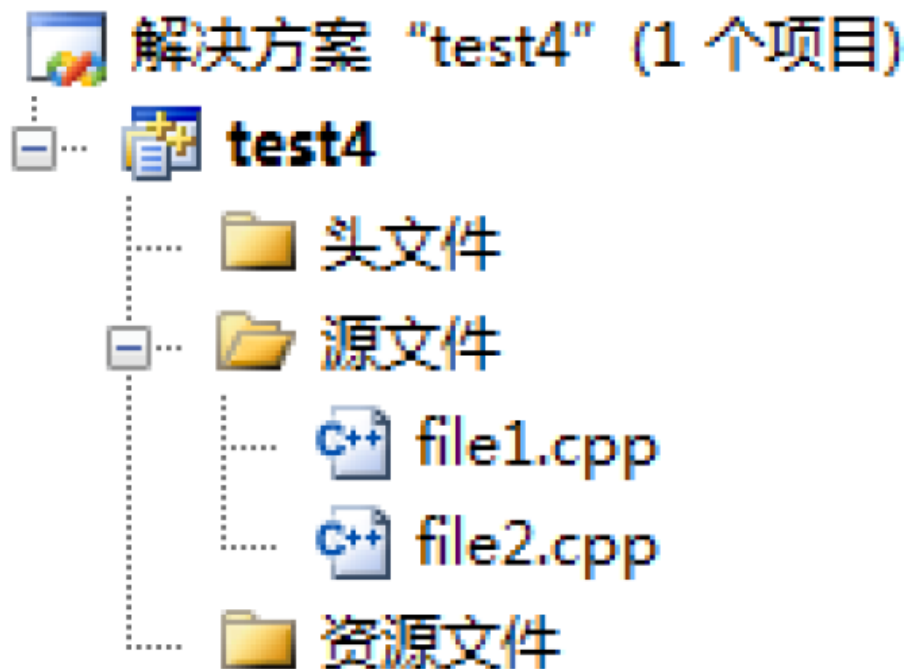
```
int a = 1;
void func1();

int main() {
    cout<<a;
    func1();
    int a = 3;
    cout<<a;
    return 0;
}
```

```
void func1() {
    int a = 2;
    cout<<a;
    for(int a=4; a<5;) {
        cout<<a;
        a++;
    }
}
```

变量作用域和生存期

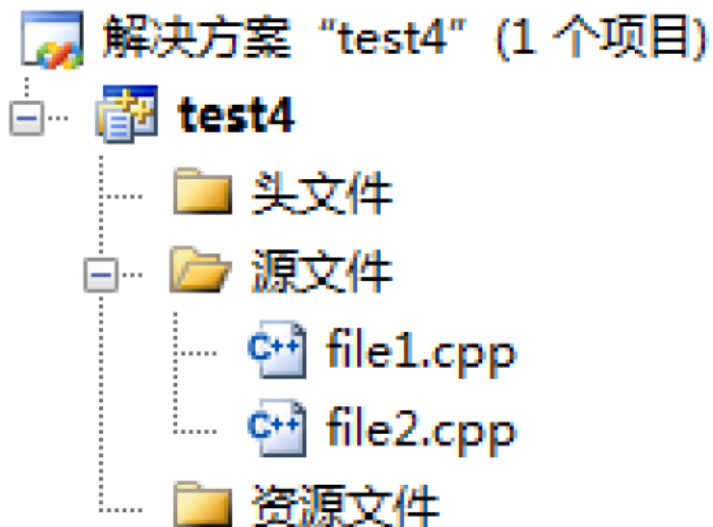
多文件结构：一个程序包含多个源文件、头文件等



多个源文件时，只能有一个文件包含main()函数

变量作用域和生存期

多文件结构：全局变量再每个文件都可见，但使用前要用extern声明



file1.cpp

```
int a = 1; // 变量定义
```

file2.cpp

```
#include <iostream>
using namespace std;
extern int a; // 外部声明
int main() {
    cout<<a<<endl;
}
```

变量作用域和生存期

和全局变量一样，函数也可以在多个文件中定义

file1.cpp

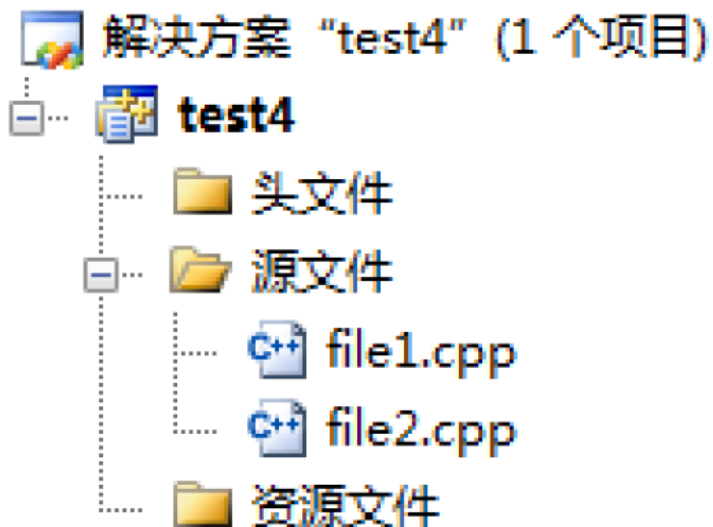
```
int func(int a) {cout<<a;} // 定义函数
```

file2.cpp

```
#include <iostream>
using namespace std;
extern int func(int a) ; // 外部函数声明, extern可省略 !
int main() {
    func(0) ; // 使用外部函数
}
```


变量作用域和生存期

static限定符： 全局变量只在其定义的文件中可见



file1.cpp

```
static int a = 1;
```

file2.cpp

```
#include <iostream>
using namespace std;
extern int a; // 错误
int main() {
    cout<<a<<endl;
}
```

变量作用域和生存期

static限定符：将局部变量的生命期延长到整个程序执行过程

```
#include <iostream>
using namespace std;

int main() {
    cout<<f1();
    cout<<f1();
}

int f1() {
    static int a = 0;
    a++;
    return a;
}
```

a在第一次调用f1()时生成并初始化
(如果没有初始化, 默认为0)

a在之后的调用中不再重新生成和
初始化, 而是保留上次修改的值

变量作用域和生存期

源文件多产生的问题：结构混乱

file1.cpp

```
extern int a2;  
extern int a3;  
extern f2();  
extern f3();
```

```
int a1;  
void f1() {};
```

file2.cpp

```
extern int a1;  
extern int a3;  
extern f1();  
extern f3();
```

```
int a2;  
void f2() {};
```

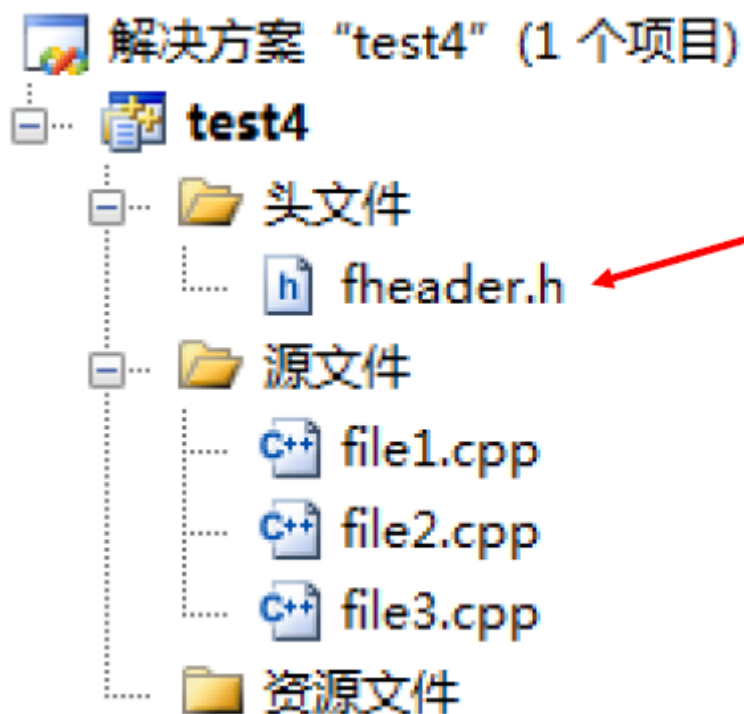
file3.cpp

```
extern int a1;  
extern int a2;  
extern f1();  
extern f2();
```

```
int a3;  
void f3() {};
```

变量作用域和生存期

头文件：以.h结尾，用来放所有外部声明



fhd.h

```
extern int a1;  
extern int a2;  
extern int a3;  
  
extern f1 ();  
extern f2 ();  
extern f3 ();
```

变量作用域和生存期

头文件： 每个源文件只需加上 `#include "fhd.h"`

fhd.h

```
extern int a1;  
extern int a2;  
extern int a3;  
  
extern f1 ();  
extern f2 ();  
extern f3 ();
```

```
#include "fhd.h"
```

```
int a1;  
void f1 ();
```

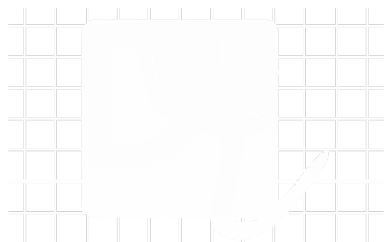
```
#include "fhd.h"
```

```
int a2;  
void f2 ();
```

```
#include "fhd.h"
```

```
int a3;  
void f3 ();
```

函数模板



函数模板

函数重载可以支持多种数据类型，但是冗余度高

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

```
char max (char a, char b){  
    if(a>b)  
        return a;  
    else  
        return b;  
}
```

```
double max (double a, double b){  
    if(a>b)  
        return a;  
    else  
        return b;  
}
```

函数模板

函数模板：“提取”出一个可变化的类型参数，定义一组函数

```
T max (T a, T b) {  
    if (a>b)  
        return a;  
    else  
        return b;  
}
```


函数模板

函数模板的完整定义

template 关键字

模板参数, 用尖括号括起来, 可以是一个或多个, 用“,”分隔

```
template <typename T> T max (T a, T b) {  
    if (a>b)  
        return a;  
    else  
        return b;  
}
```

模板参数类型,
任意取名

函数模板

```
template <typename T> T max (T a, T b) {  
    return (a>b)?a:b;  
}
```

```
void main() {  
    int i1=-11, i2=0;  
    double d1, d2;  
    cout<<max(i1,i2)<<endl; //OK,形参T对应于int  
    cout<<max(23,-56)<<endl;  
    cout<<max('f','k')<<endl; //OK,T对应char  
    cin>>d1>>d2;  
    cout<<max(d1,d2)<<endl; //OK,T对应double  
    cout<<max(23,-5.6)<<endl; //出错!不进行实参到形参  
    类型的自动转换  
}
```

函数模板

```
template <typename T, typename U>
    bool if_max (T a, U b) {
        if(a>b) return true;
        else return false;
    }
```

```
void main() {
    int i1=-11;
    double d1=0.12;
    cout<<max(i1,d1)<<endl;
    //OK,形参T对应于int, U对应于double
}
```

函数模板与函数重载

调用顺序：首先检查是否存在重载函数，若匹配成功则调用该函数，否则再去匹配函数模板

```
template <typename T> T min (T a, T b) {  
    return (a<b?a:b);  
}  
  
char min (char a, char b) {  
    return (a<b)?a:b;  
}
```

函数模板与函数重载

```
template <typename T> T min (T a, T b) {  
    return (a<b?a:b);  
}  
char min (char a, char b) {  
    return (a<b)?a:b;  
}
```

```
void main() {  
    cout<<min(3,-10)<<endl; //使函数模板  
    cout<<min(2.5,99.5)<<endl; //使用函数模板  
    cout<<min('m','c')<<endl; //使用函数重载  
}
```

函数模板重载

定义两个函数模板，都叫做sum，都使用了一个类型参数T，但两者的形参个数不同

```
template <typename T> T sum(T a[], int size ) {  
    T total=0;  
    for (int i=0;i<size;i++)  
        total+=a[i];  
    return total;  
}
```

```
template <typename T>T sum(T a1[], T a2[], int size  
) {  
    T total=0;  
    for (int i=0;i<size;i++)  
        total+=(a1[i]+a2[i]);  
    return total;  
}
```

END

