3 结构化的程序设计

郑海永
zhenghaiyong@ouc.edu.cn

中国海洋大学 电子工程学院

目录 I

内容提要 I

内容提要 I

函数

```
1 y=f(x)
```

- 已知一个数，求其平方根：r=sqrt(100.0);
- 已知底数x，幂指数y，求x^y：k=pow(x,y);
- 求一个字符串的长度：i=strlen(str1);
- 比较两个字符串的大小：v=strcmp(str1,str2);
- 把字符串转换为相应的整数：n=atoi(str1);

什么是函数

```
1  #include<iostream>
2  using namespace std;
3  int absolute(int n) //形式参数 n
4  {
5      if(n<0) return (-n);
6      else return n;
7  }
8  int main()
9  {
10     int m=-123,result=0;
11     result=absolute(m); //实际参数 m
12     cout<<result;
13     return 0;
14 }
```

什么是函数

```
1  #include<iostream>
2  using namespace std;
3  float max(float a, float b)
4  {
5      if(a>b) return a;
6      else return b;
7  }
8  int main()
9  {
10     int m=3,n=4;
11     float result=0;
12     result=max(m,n);
13     cout<<result;
14     return 0;
15 }
```

什么是函数

```
1  #include<iostream>
2  using namespace std;
3  float max(float a, float b)
4  {
5      if(a>b) return a;
6      else return b;
7  }
8  int main()
9  {
10     cout<<max(3,4);
11     return 0;
12 }
```

什么是函数

```
1  #include<iostream>
2  using namespace std;
3  int get_int()
4  {
5      int n=0;
6      cout<<"Please input an integer:"<<endl;
7      cin>>n;
8      return n;
9  }
10 int main()
11 {
12     int result=0;
13     result=get_int();
14     cout<<result;
15     return 0;
16 }
```

什么是函数

```
1  #include<iostream>
2  using namespace std;
3  void delay(int n)
4  {
5      for(int i=0;i<n*100000;i++)
6      return;
7  }
8  int main()
9  {
10     for(int j=0;j<100;j++)
11     {
12         cout<<j<<endl;
13         delay(1000);
14     }
15     return 0;
16 }
```

什么是函数

```
#include<iostream>
using namespace std;
void show()
{
    cout<<"**********************************"<<endl;
    cout<<"*    System error has occurred.    *"<<endl;
    cout<<"* Please contact the administrator *"<<endl;
    cout<<"*   Sorry for the inconvenience.   *"<<endl;
    cout<<"**********************************"<<endl;
}
int main()
{
    show();
    return 0;
}
```

什么是函数

```
1 #include<iostream>
2 using namespace std;
3 int main()
4 {
5
6     return 0;
7 }
```

- 函数是 C 程序的基本构成单位
  - 一个 C 程序由一个或多个源程序文件组成。
  - 一个源程序文件可以由一个或多个函数组成。
- 函数都是有“类型”的
  - 函数的类型是指：函数的返回值的数据类型

函数调用的方式

以函数在程序中的出现位置和形式来看，函数的调用方式可分为三种：

1. 函数调用作为独立语句，例如：

```
1 stringprint();
```

   调用函数完成某项功能，没有任何的返回值（void）。

2. 函数作为表达式的一部分，例如：

```
1 number=max(numA,numB)/2;
```

3. 以实参形式出现在其它函数的调用中，例如：

```
1 number=min(sum(-5,100),numC);
```

函数的声明

函数的原型 = 返回值类型 + 函数名 + 参数类型

```cpp
#include<iostream>
using namespace std;
float max(float a,float b)
{
    if(a>b) return a;
    else return b;
}
int main()
{
    cout<<max(3,4);
    return 0;
}
```

```cpp
#include<iostream>
using namespace std;
float max(float,float);//函数声明
int main()
{
    cout<<max(3,4);
    return 0;
}
float max(float a,float b)
{
    if(a>b) return a;
    esle return b;
}
```

函数原型及函数声明

函数原型

- 由函数的返回类型、函数名以及参数表构成的一个符号串，其中参数可不写名字。

```
1 bool checkPrime(int);
```

- 函数的原型又称为函数的“Signature”。

函数声明

- 函数在使用前都要声明，除非被调用函数的定义部分已经出现在主调函数之前。
- 在 C 语言中，函数声明就是函数原型。

函数声明

max.h

```
1 float max(float a,float b)
2 {
3     if(a>b)
4         return a;
5     else
6         return b;
7 }
```

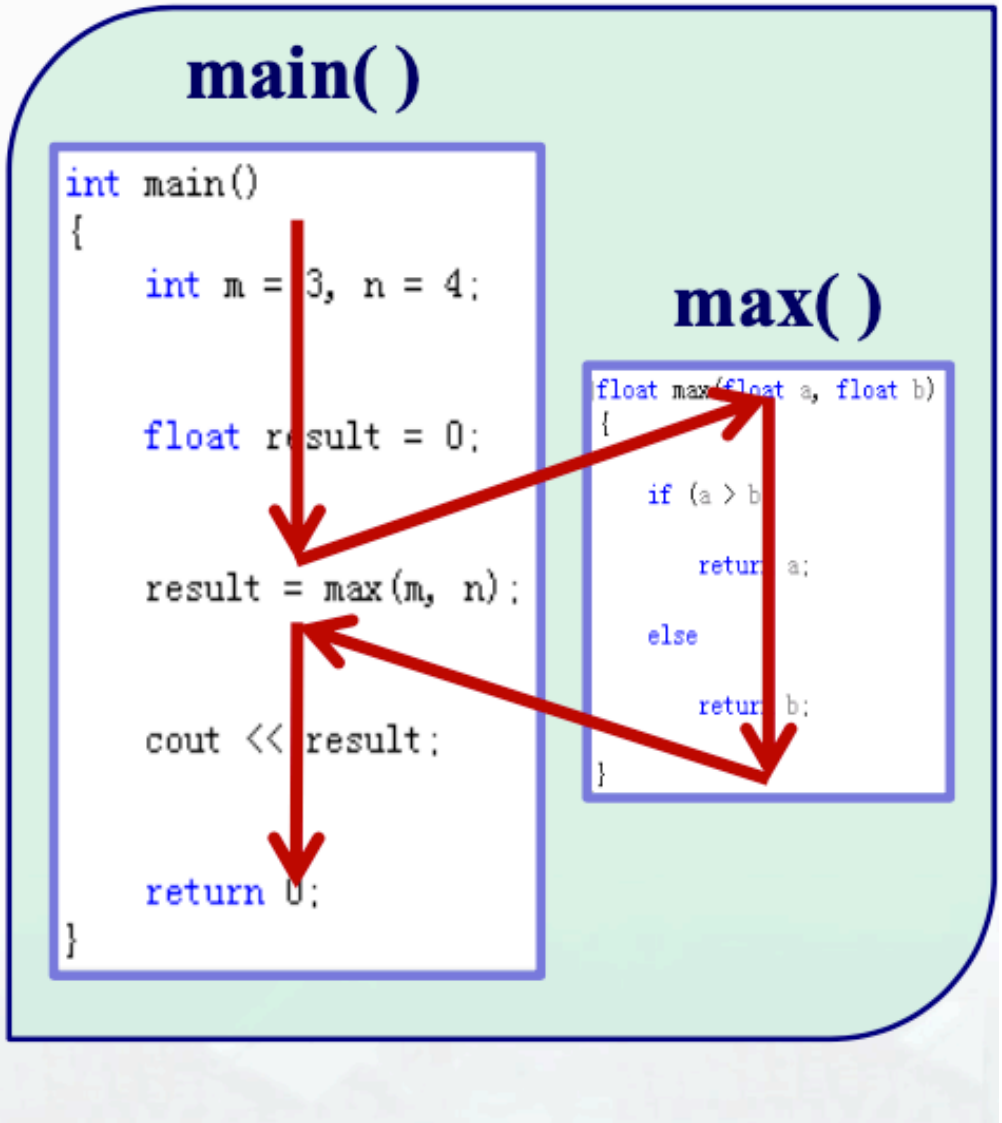
compare.cpp

```
1 #include<iostream>
2 #include "max.h"
3 using namespace std;
4 int main()
5 {
6     cout<<max(3,4);
7     return 0;
8 }
```

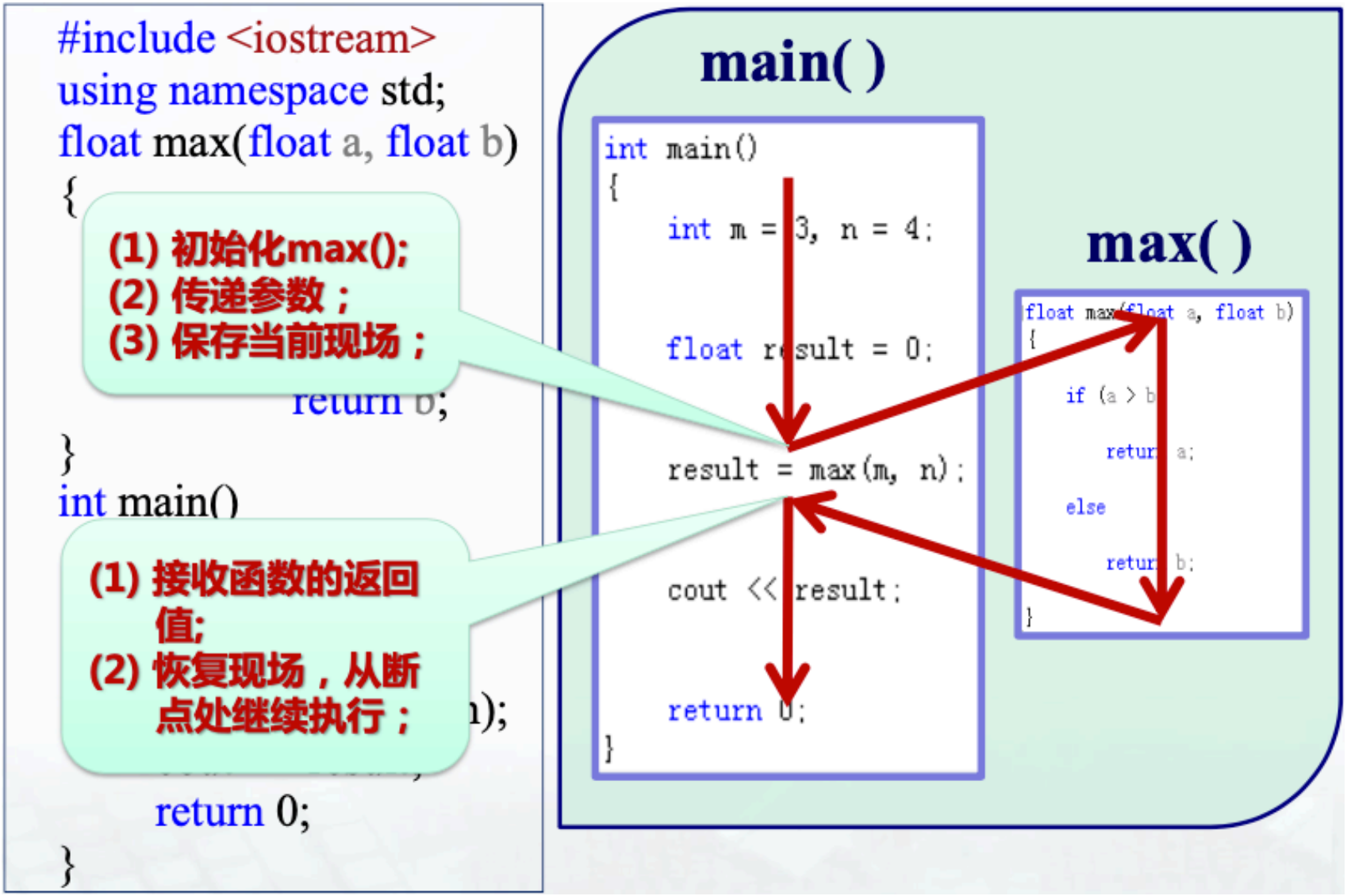
内容提要 I

函数的执行

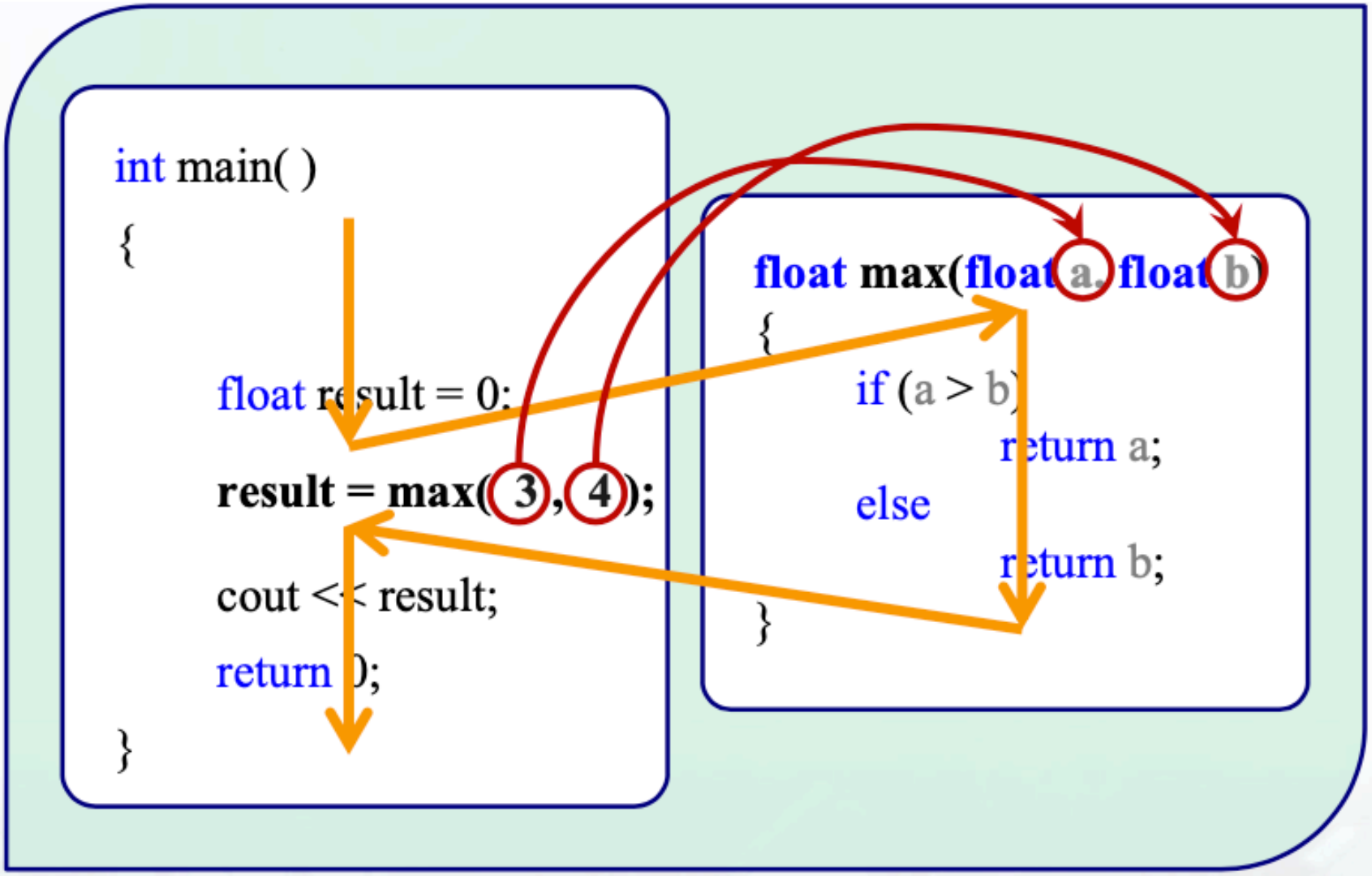
```
#include <iostream>
using namespace std;
float max(float a, float b)
{
        if (a > b)
                return a;
        else
                return b;
}
int main()
{
        int m = 3, n = 4;
        float result = 0;
        result = max(m, n);
        cout << result;
        return 0;
}
```



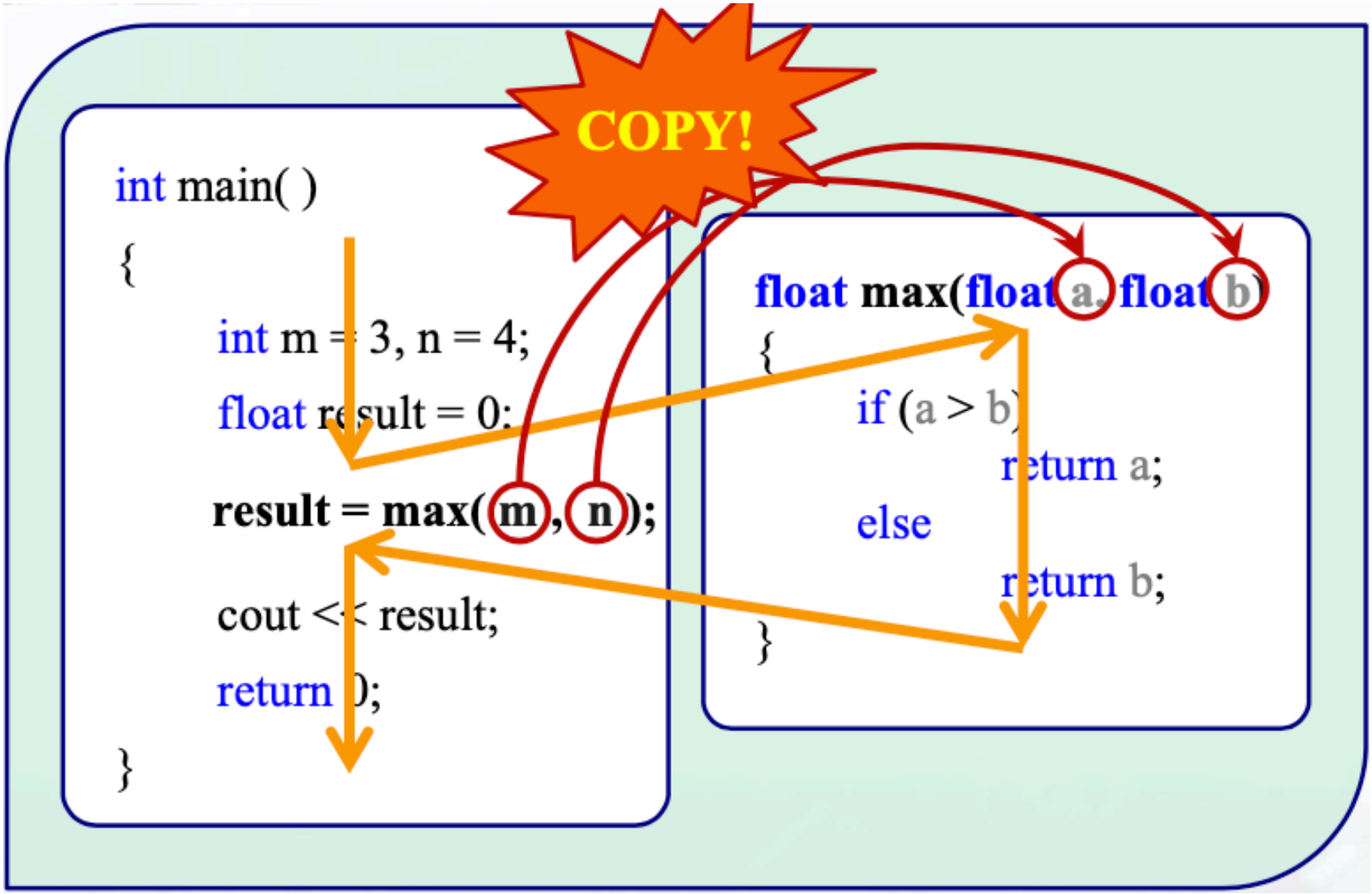
函数的执行


#include <iostream>
using namespace std;
float max(float a, float b)
{
return b;
}
int main()
return 0;
}
(1) 初始化max();
(2) 传递参数；
(3) 保存当前现场；
(1) 接收函数的返回值;
(2) 恢复现场，从断点处继续执行；
main( )
int main()
{
int m = 3, n = 4;
float result = 0;
result = max(m, n);
cout << result;
return 0;
}
max( )
float max(float a, float b)
{
if (a > b
retur a;
else
retur b;
}

函数参数的传递


int main( )
{
float result = 0;
result = max(3 , 4);
cout << result;
return 0;
}
float max(float a, float b)
{
if (a > b)
return a;
else
return b;
}

函数参数的传递


COPY!
int main( )
{
int m = 3, n = 4;
float result = 0;
result = max(m, n);
cout << result;
return 0;
}
float max(float a, float b)
{
if (a > b)
return a;
else
return b;
}

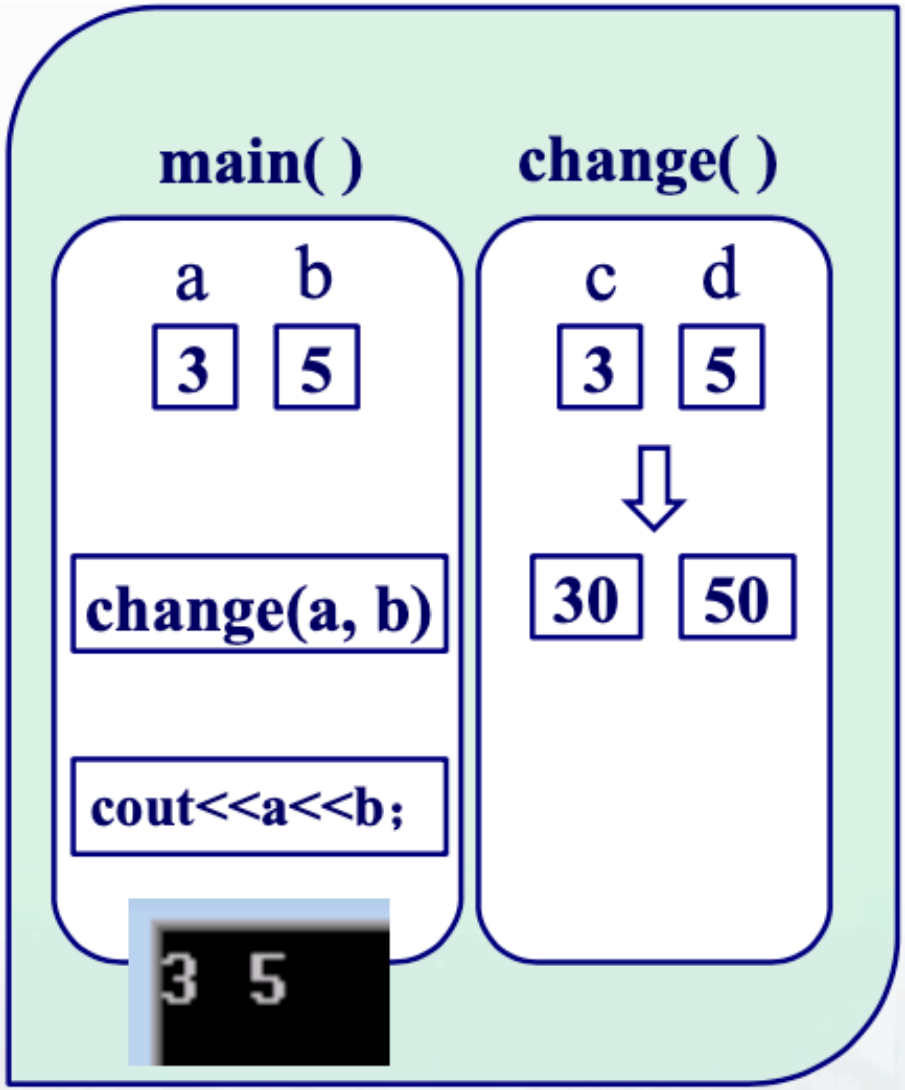
函数参数的传递

参数的传递

- 实参与形参具有不同的存储单元，实参与形参变量的数据传递是“值传递”。
- 函数调用时，系统给形参分配存储单元，并将实参对应的值传递给形参。
- 实参与形参的类型必须相同或可以兼容。

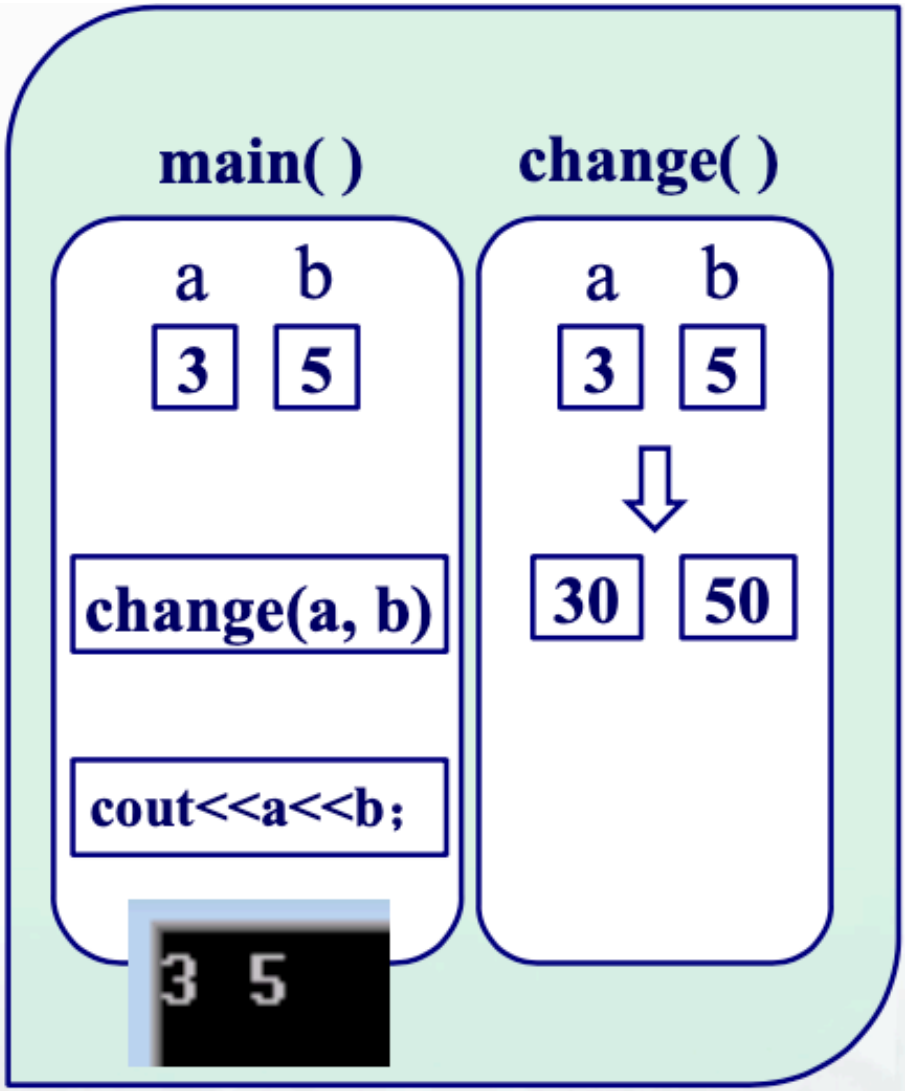
思考题

```
#include <iostream>
using namespace std;
void change(int c, int d)
{
        c = 30; d = 50;
}
int main()
{
        int a = 3, b = 5;
        change(a, b);
       cout<<a<<" "<<b;
        return 0;
}
```



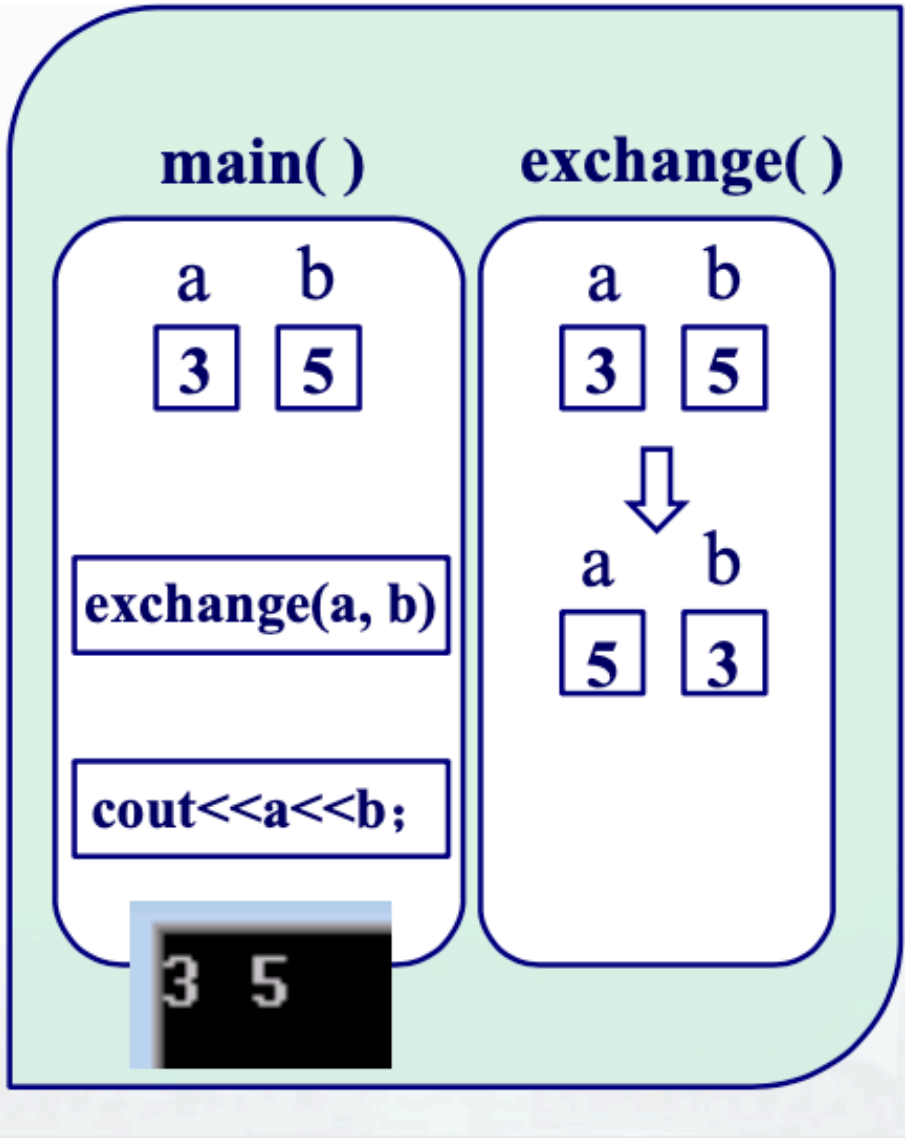
思考题

```
#include <iostream>
using namespace std;
void change(int a, int b)
{
        a = 30; b = 50;
}
int main()
{
        int a = 3, b = 5;
        change(a, b);
       cout<<a<<" "<<b;
        return 0;
}
```



思考题

```
#include<iostream>
using namespace std;
void exchange(int a, int b)
{
    int p;
    if (a < b)
    {
        p = a; a = b; b = p;
    }
}
int main()
{
    int a = 3, b = 5;
    exchange(a, b);
    cout<<a<<" "<<b<< endl;
    return 0;
}
```



内容提要 I

局部变量与全局变量

根据变量在程序中作用范围的不同，可以将变量分为：

- **局部变量**：在函数内或块内定义，只在这个函数或块内起作用。
- **全局变量**：在所有函数外定义，作用域从定义变量的位置开始到本程序文件结束。

局部变量

```
#include<iostream>
using namespace std;
void exchange(int a, int b)
{
    int p;
    if (a < b)
    {
        p = a; a = b; b = p;
    }
}
int main()
{
    int a = 3, b = 5;
    exchange(a, b);
    cout<<a<<" "<<b<< endl;
    return 0;
}
```

a, b 的势力范围

形参是局部变量

a, b 的势力范围

局部变量

```
int excel_number = 0;
void excel_count( float score )
{
        if (score > 85)
        {
                excel_number++;
        }
}
int main()
{
        float score = 0;
        for (int i = 0; i < 100; i++)
        {
                cin >> score;
                excel_count(score);
        }
        cout << excel_count << endl;
        return 0;
}
```

局部变量 score 的作用范围

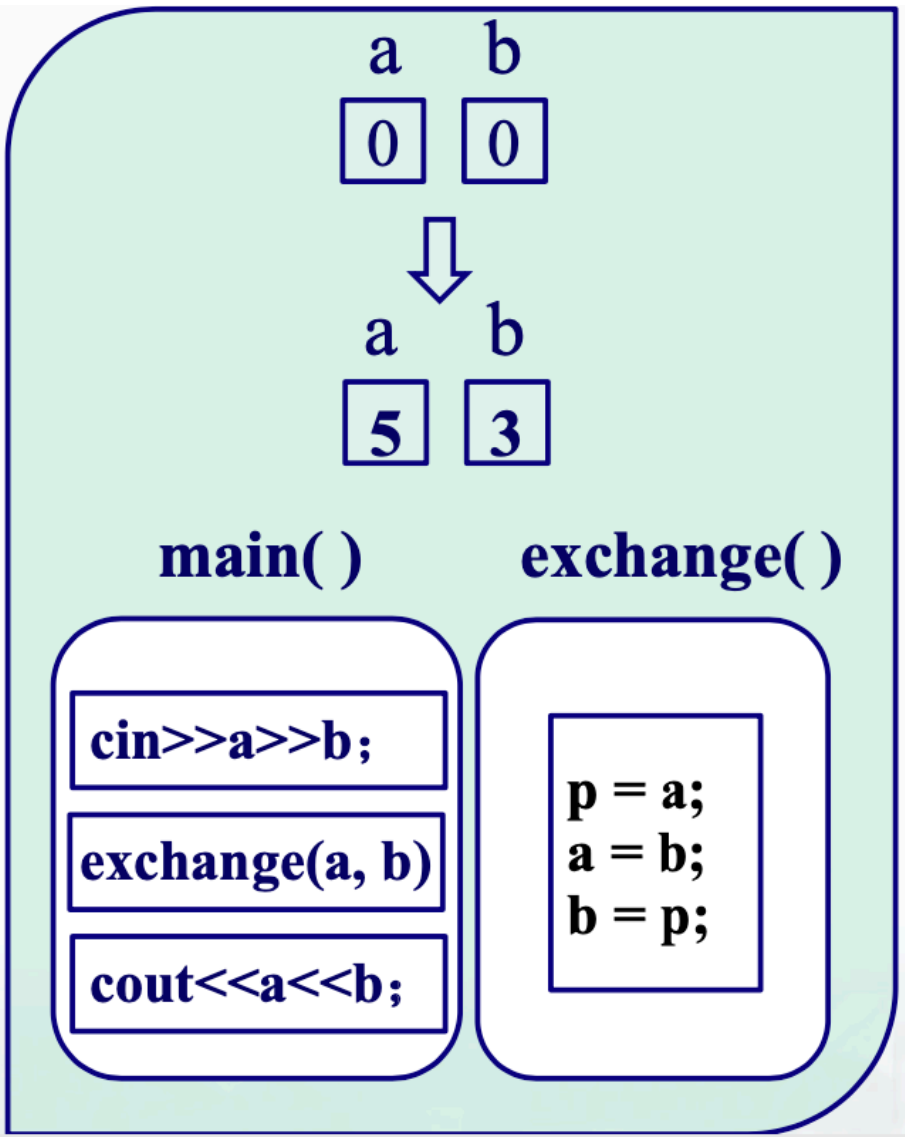
全局变量 excel_number 的作用范围

局部变量 i 的作用范围

局部变量 score 的作用范围

局部变量 vs. 全局变量

```
#include<iostream>
using namespace std;
int a = 0, b = 0;
void exchange( )
{
    int p;
    if (a < b)
    {
        p = a; a = b; b = p;
    }
}
int main()
{
    cin >> a >> b;
    exchange( );
    cout<<a<<" "<<b<< endl;
    return 0;
}
```



局部变量 vs. 全局变量

```
#include<iostream>
using namespace std;
int a = 0, b = 0;
void exchange(int a, int b)
{
    int p;
    if (a < b)
    {
        p = a; a = b; b = p;
    }
}
int main()
{
    cin>>a>>b;
    exchange(a, b);
    cout<<a<<" "<<b<< endl;
    return 0;
}
```

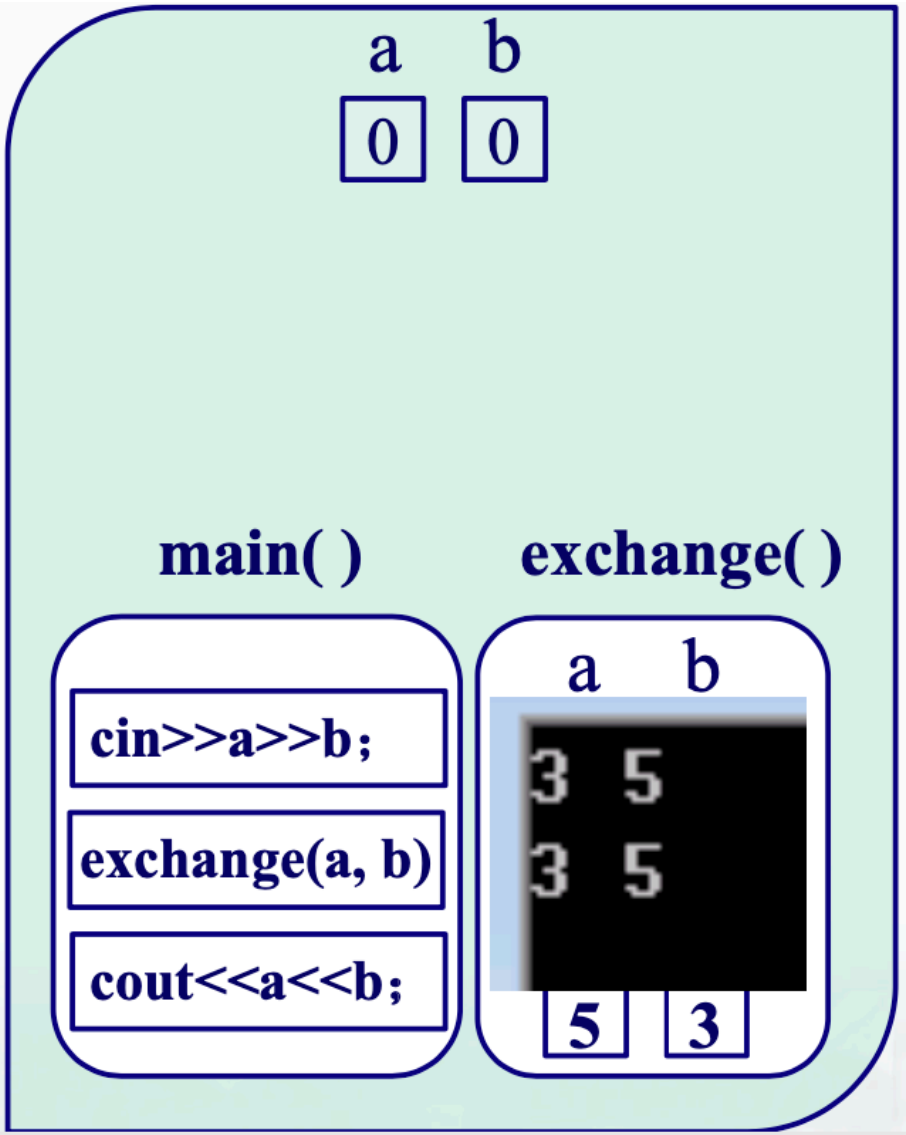
局部变量 a, b 的作用范围

全局变量 a, b 的作用范围

- 当全局变量与局部变量同名时，局部变量将在自己作用域内有效，它将屏蔽同名的全局变量。

局部变量 vs. 全局变量

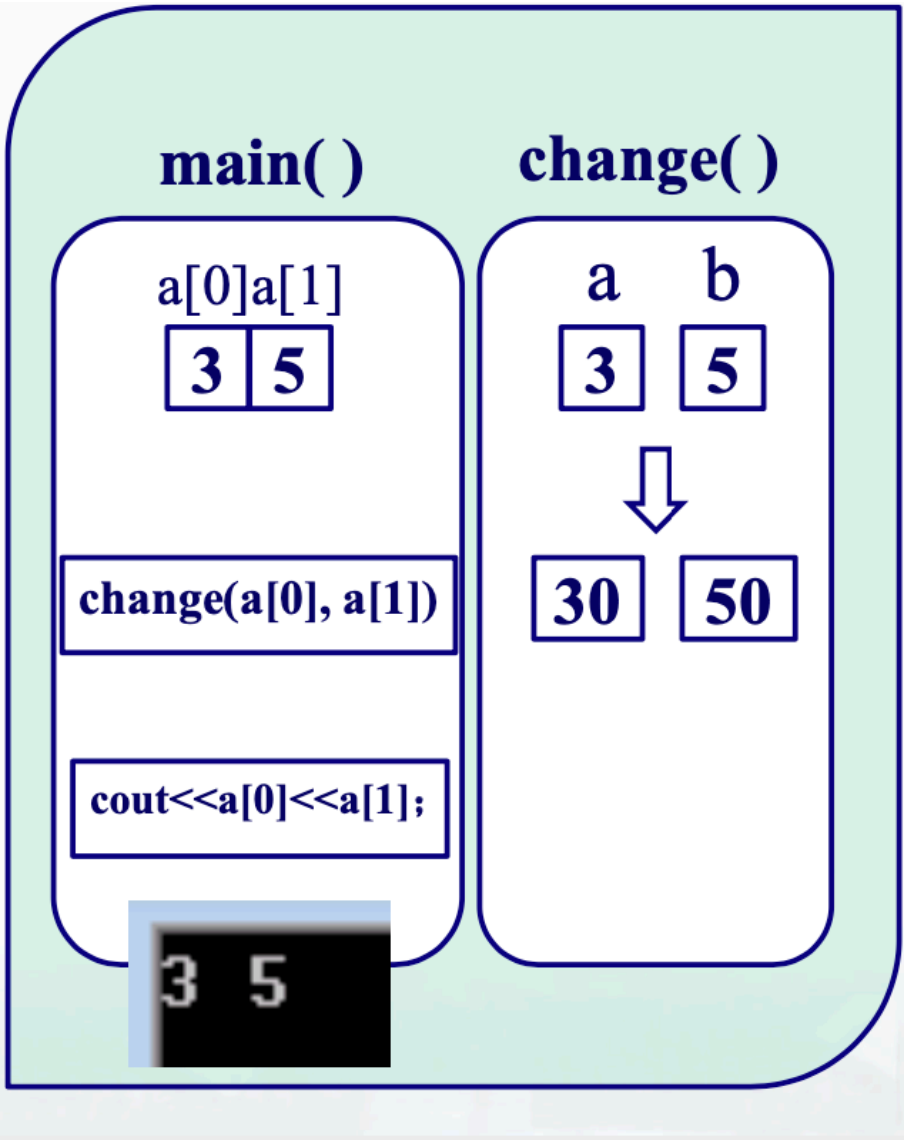
```
#include<iostream>
using namespace std;
int a = 0, b = 0;
void exchange(int a, int b)
{
    int p;
    if (a < b)
    {
        p = a; a = b; b = p;
    }
}
int main()
{
    cin>>a>>b;
    exchange(a, b);
    cout<<a<<" "<<b<< endl;
    return 0;
}
```



内容提要 I

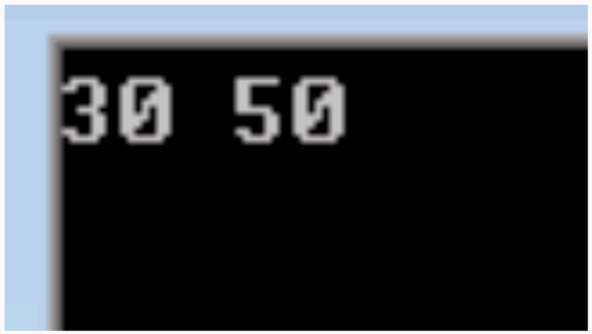
数组元素做函数参数

```
#include <iostream>
using namespace std;
void change(int a, int b)
{
        a = 30; b = 50;
}
int main()
{
        int a[2] = {3, 5};
        change(a[0], a[1]);
        cout<<a[0]<<" "
                <<a[1]<<endl;
        return 0;
}
```

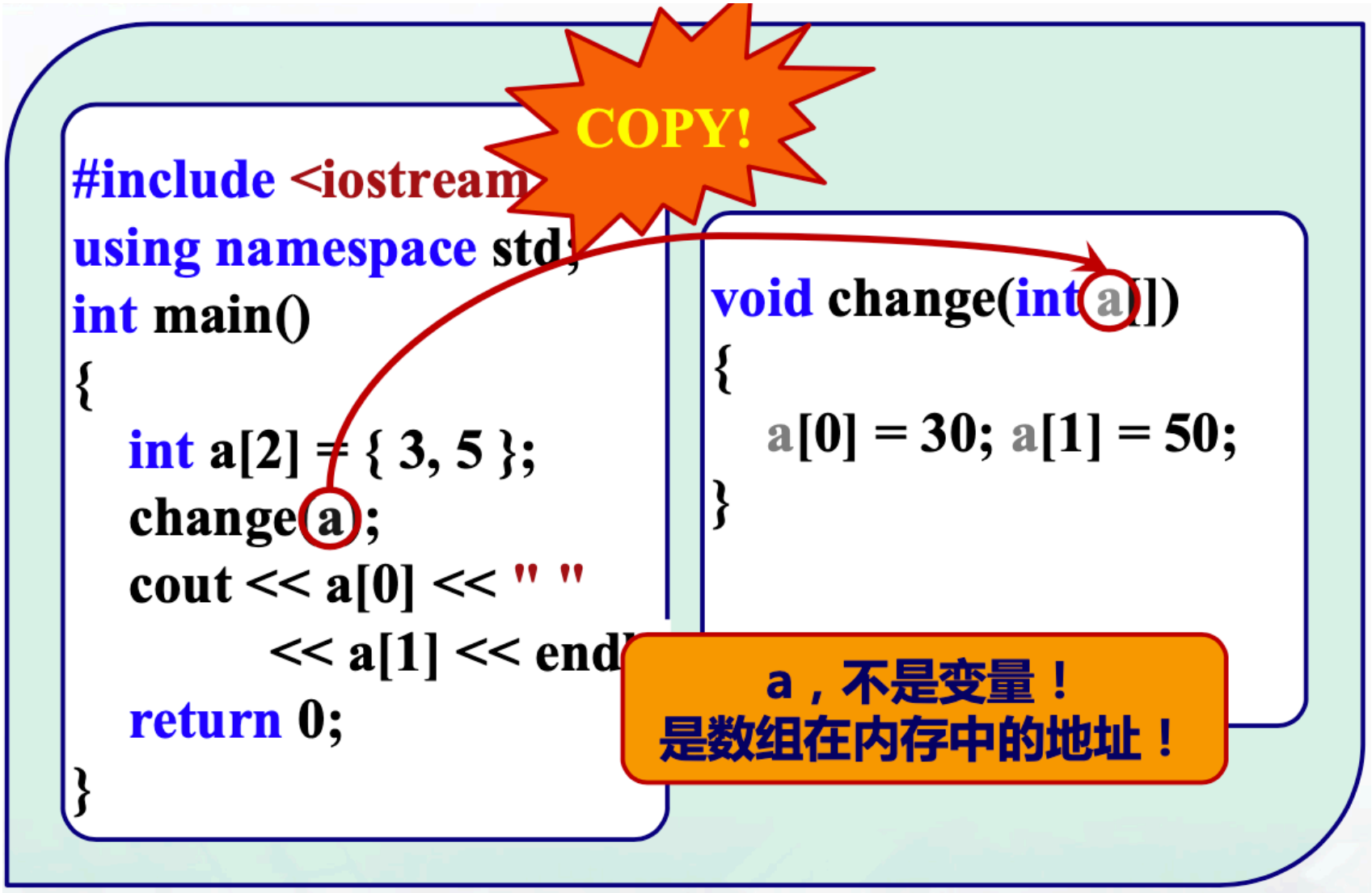


数组元素做函数参数

```
#include <iostream>
using namespace std;
void change(int a[])
{
    a[0] = 30; a[1] = 50;
}
int main()
{
    int a[2] = { 3, 5 };
    change(a);
    cout << a[0] << " "
         << a[1] << endl;
    return 0;
}
```

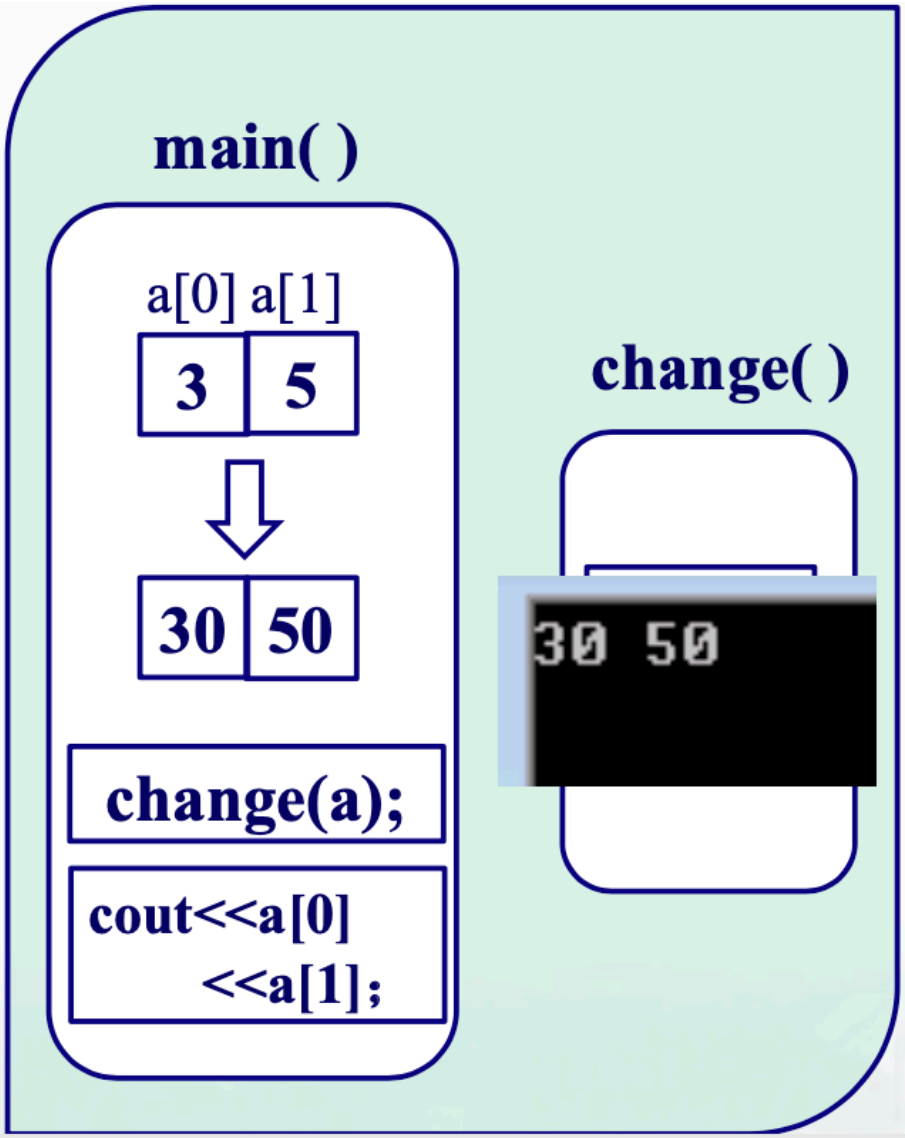


数组名做函数参数


COPY!
#include <iostream
using namespace std;
int main()
{
int a[2] = { 3, 5 };
change(a);
cout << a[0] << " "
<< a[1] << endl;
return 0;
}
void change(int a[])
{
a[0] = 30; a[1] = 50;
}
a，不是变量！
是数组在内存中的地址！

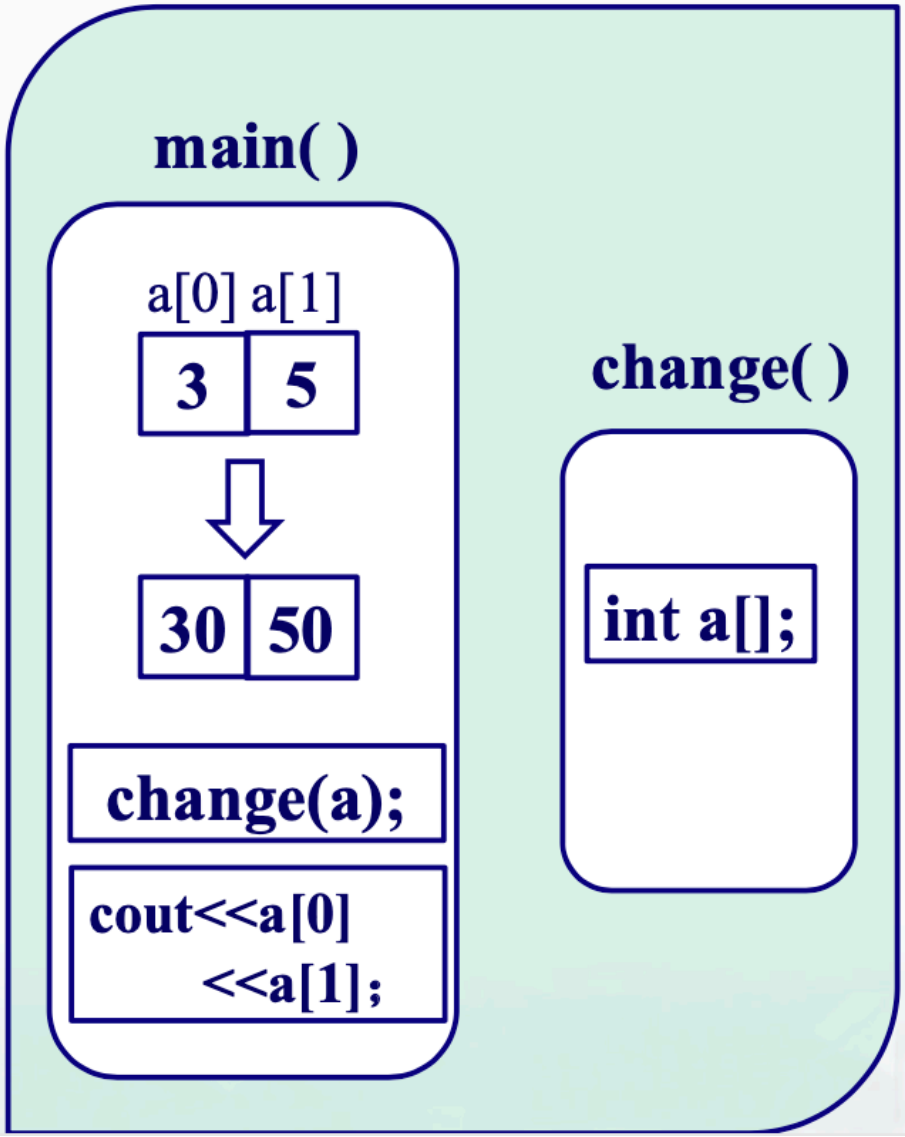
数组名做函数参数

```
#include <iostream>
using namespace std;
void change(int a[])
{
    a[0] = 30; a[1] = 50;
}
int main()
{
    int a[2] = { 3, 5 };
    change(a);
    cout << a[0] << " "
         << a[1] << endl;
    return 0;
}
```



数组名做函数参数

```
#include <iostream>
using namespace std;
void change(int a[])
{
    a[0] = 30; a[1] = 50;
}
int main()
{
    int a[2] = { 3, 5 };
    change(a);
    cout << a[0] << " "
         << a[1] << endl;
    return 0;
}
```



内容提要 I

日历问题

问题描述：

- 给定从公元 2000 年 1 月 1 日开始逝去的天数，请编写程序给出这一天是哪年哪月哪日星期几。
  - 注意闰年：闰年被定义为能被 4 整除的年份，但是能被 100 整除而不能被 400 整除的年是例外，它们不是闰年。
  - 例如：1700、1800、1900 和 2100 不是闰年，而 1600、2000 和 2400 是闰年。
  - 注意每个月的天数不一样！

日历问题

输入输出要求：

- 输入多组数据，每组一个正整数，表示从2000年1月1日开始已经过去的天数。
- 对输入的每个天数，输出一行，该行包含对应的日期和星期几。格式为：

```
1 "YYYY-MM-DD DayOfWeek"
```

其中“DayOfWeek”必须是：Sunday，Monday，Tuesday，Wednesday，Thursday，Saturday。

- 输入最后一行是-1，不必处理。可以假设结果的年份不会超过9999。

日历问题


■ 思路
days
计算出星期几
减掉每年的天数
year
减掉每月的天数
month
day
1、判断是否闰年
2、减掉相应天数
直到剩下天数不够一年
减掉每个月的天数
直到剩下天数不够一月

日历问题

① 计算星期几

```
1 int get_dayofweek()
2 {
3     int dayofweek;
4     dayofweek = days%7;
5     return dayofweek;
6 }
7 char week[7][10] =
  ↪  {"Saturday","Sunday","Monday","Tuesday",
8         "Wednesday","Thursday","Friday"};
```

日历问题

② 计算年数

```
1  int get_year()
2  {
3      int i=2000,leap_year;
4      while(1)
5      {
6          leay_year=(i%4==0&&i%100!=0||i%400==0);
7          if(leap_year==1&&days>=366)
8          { days = days-366; i++; continue; }
9          else if(leap_year==0&&days>=365)
10         { days = days-365; i++; continue; }
11         else break;
12     }
13     return i;
14 }
```

日历问题

③ 计算月份

```
1  int get_month(int leap_year)
2  {
3      int pmonth[12]={31,28,31,30,31,30,31,31,30,31,30,31};
4      int rmonth[12]={31,29,31,30,31,30,31,31,30,31,30,31};
5      int j=0;
6      while(1)
7      {
8          if(leap_year==1&&days>=rmonth[j])
9          { days=days-rmonth[j]; j++; }
10         else if(leap_year==0&&days>=pmonth[j])
11         { days=days-pmonth[j]; j++; }
12         else break;
13     }
14     return ++j;
15 }
```

```
#include<iostream>
using namespace std;
int days;
int get_dayofweek();
int get_year();
int get_month(int);
int main()
{
    int year,month,dayofweek,leap_year;
    char week[7][10] = {"Saturday","Sunday","Monday","Tuesday",
        "Wednesday","Thursday","Friday"};
    while((cin>>days)&&days!=-1) {
        dayofweek=get_dayofweek();
        year=get_year();
        leap_year=(year%4==0&&year%100!=0||year%400==0);
        month=get_month(leap_year);
        cout<<year<<"-"<<month<<"-"<<++days<<" "<<week[dayofweek];
    }
    return 0;
}
```

关于全局变量

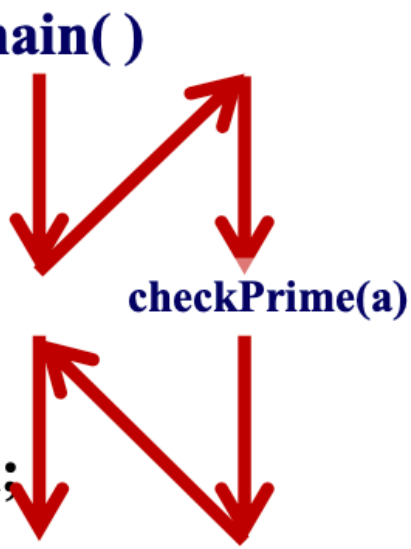
- 破坏了函数的“相对独立性”；
- 增加了函数之间的“耦合性”；
- 函数之间的交互不够清晰；
- 不在非常必要的情况下，不要使用全局变量。

内容提要 I

内容提要 I

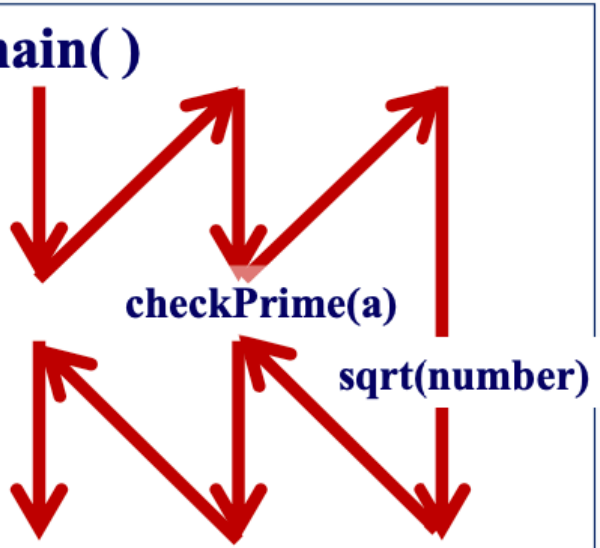
函数的嵌套调用

```
#include <iostream>
#include <cmath>
using namespace std;
bool checkPrime(int);
int main()
{
        int a;
        cout << "请输入一个整数" << endl;
        while (cin >> a)
        {
                if (checkPrime(a))
                        cout << "是质数" << endl;
                else
                        cout << "不是质数" << endl;
        }
        return 0;
}
```



函数的嵌套调用

```
bool checkPrime(int number)
{
        int i, k;
        k = sqrt(number);
        for (i = 2; i <= k; i++)
        {
                if (number % i == 0)
                                        //只要有一个数被除尽
                        return 0;       //则不是素数
        }
        return 1;           //走到这一步，说明没能被除尽
}
```



思路整理

- **函数不能嵌套定义**：所有函数一律平等。
- **函数可以嵌套调用**：无论嵌套多少层，原理都一样。

- 一个函数能调用“它自己”吗？

求 $n!$

```
1  #include<iostream>
2  using namespace std;
3  int fact(int n)
4  {
5      if(n==1)
6          return 1;
7      else
8          return n*fact(n-1);
9  }
10 int main()
11 {
12     cout<<fact(4)<<endl;
13     return 0;
14 }
```

- 问题：已知 n，求 $n!$

  $n! = (n-1)! * n$

  $(n-1)! = (n-2)! * (n-1)$

  ……

  $3! = 2! * 3$

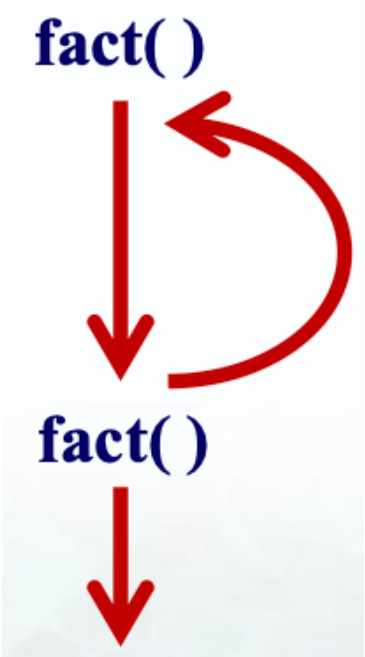
  $2! = 1! * 2$

  $1! = 1$
- 可知：
  - fact(n)的值等于fact(n-1)*n；
  - fact(1)=1。

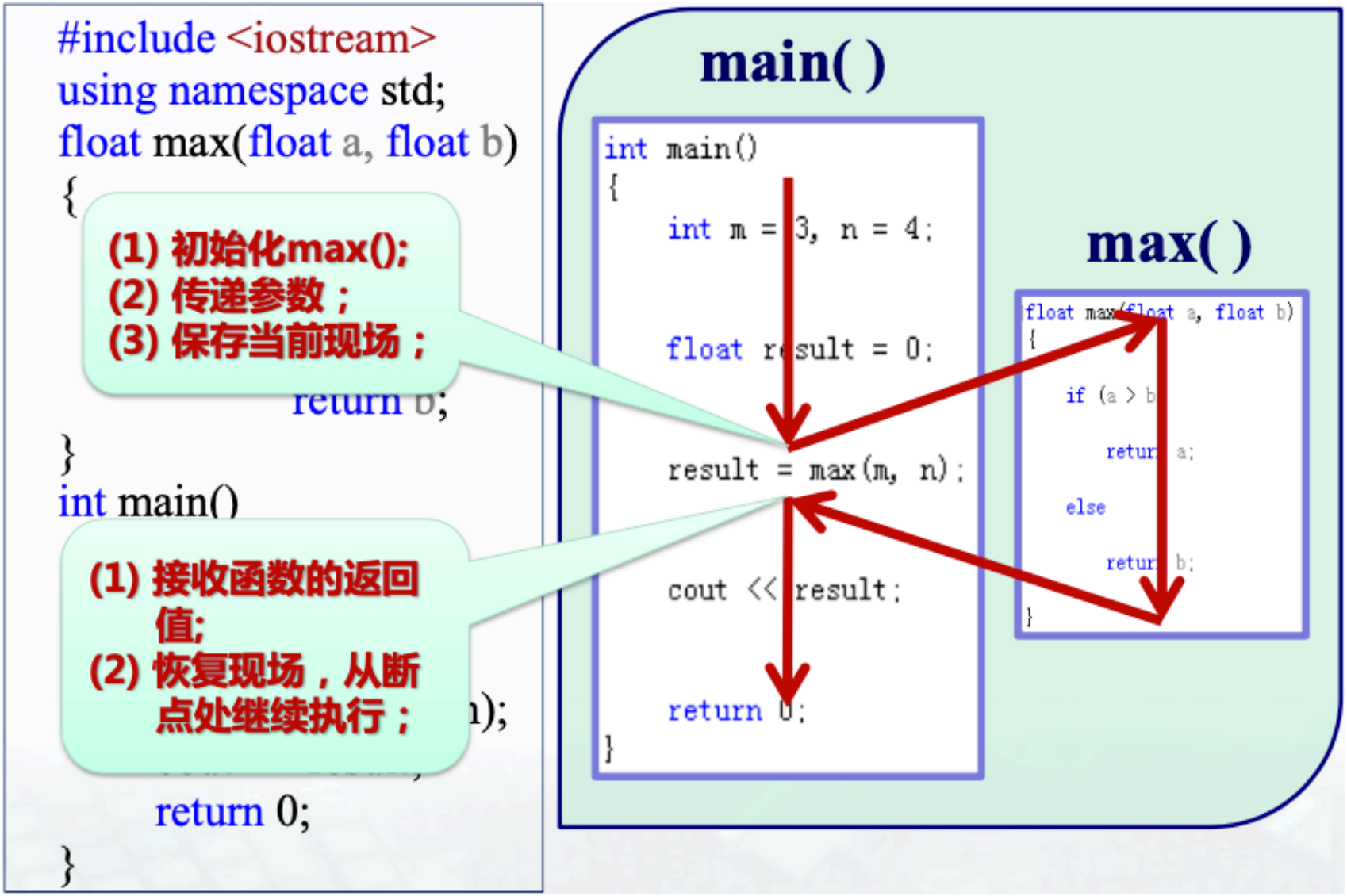
思考一下

```
1  #include<iostream>
2  using namespace std;
3  int fact(int n)
4  {
5      if(n==1)
6          return 1;
7      else
8          return n*fact(n-1);
9  }
10 int main()
11 {
12     cout<<fact(4)<<endl;
13     return 0;
14 }
```

- 这个称为“**递归调用**”
- 什么是递归？
  - 一个函数在其定义中直接或间接调用自身的一种方法。



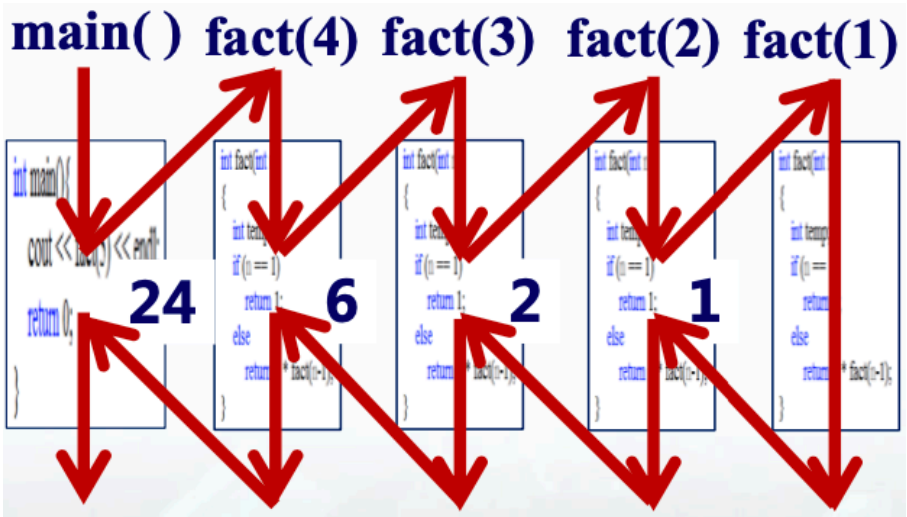
函数的执行


#include <iostream>
using namespace std;
float max(float a, float b)
{
return b;
}
int main()
return 0;
}
(1) 初始化max();
(2) 传递参数；
(3) 保存当前现场；
(1) 接收函数的返回值;
(2) 恢复现场，从断点处继续执行；
main( )
int main()
{
int m = 3, n = 4;
float result = 0;
result = max(m, n);
cout << result;
return 0;
}
max( )
float max(float a, float b)
{
if (a > b
retur a;
else
retur b;
}

递归调用

```
1  #include<iostream>
2  using namespace std;
3  int fact(int n)
4  {
5      if(n==1)
6          return 1;
7      else
8          return n*fact(n-1);
9  }
10 int main()
11 {
12     cout<<fact(4)<<endl;
13     return 0;
14 }
```

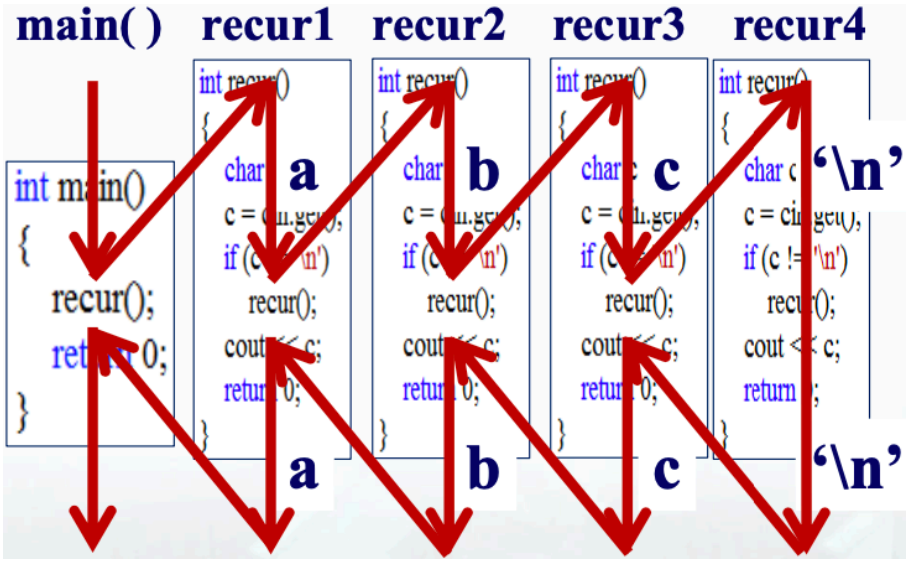
递归调用与普通调用没有区别！



内容提要 I

深入理解递归过程

```
1  #include<iostream>
2  using namespace std;
3  int recur()
4  {
5      char c;
6      c=cin.get();
7      if(c!='\n')
8          recur();
9      cout<<c;
10     return 0;
11 }
12 int main()
13 {
14     recur();
15     return 0;
16 }
```

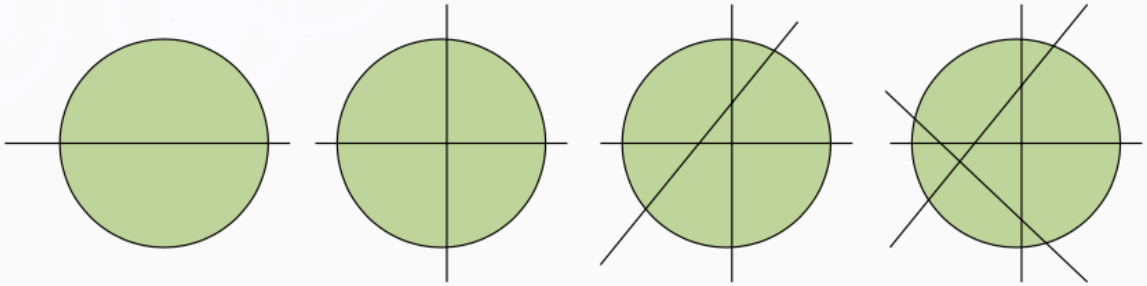



```
1  abc
```

内容提要 I

用递归来完成递推

从这个例子开始说起



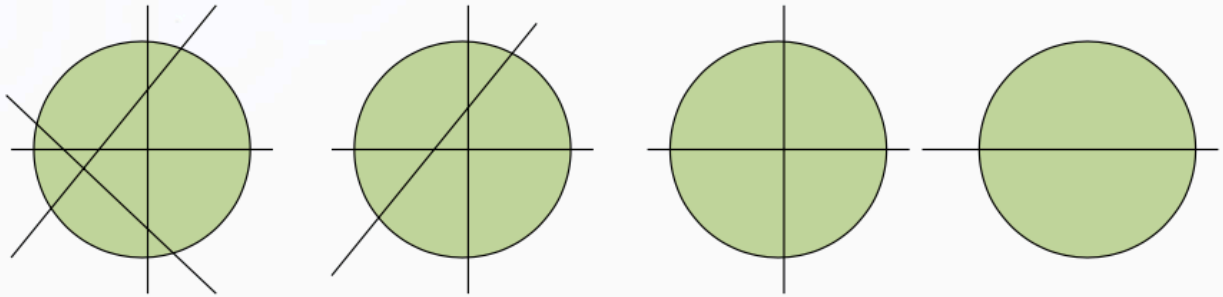
- q(0) = 1;
- q(1) = 1+1=2
- q(2) = 1+1+2=4;
- q(3) = 1+1+2+3=7;
- q(4) = 1+1+2+3+4=11;

… …

- q(n) = q(n-1)+n;

```
#include <iostream>
using namespace std;
int main()
{
    int blockCount = 0;
    int i = 0, N = 0;
    cin >> N;
    blockCount = 1;
    for (i = 1; i <= N; i++)
        blockCount = blockCount + i;
    cout << "最多可切" << blockCount << "块" << endl;
    return 0;
}
```

从这个例子开始说起



- q(n) = q(n-1)+n;
- q(n-1) = q(n-2)+n-1
- q(n-2) = q(n-3)+n-2

... ...

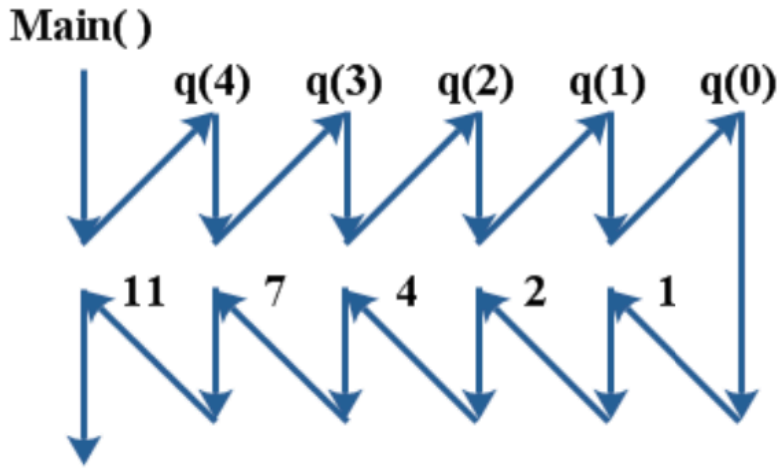
- q(2) = q(1)+2=4;
- q(1) = q(0)+1=2
- q(0) = 1

```
int q(int n)
{
    if (n == 0)
        return 1;
    else
        return(n + q(n - 1));
}
```

从这个例子开始说起

```
#include<iostream>
using namespace std;
int q(int n){
    if (n == 0)
        return 1;
    else
        return(n + q(n - 1));
}

int main(){
    cout << q(4) << endl;
    return 0;
}
```



递归与递推

不同

- 递推的关注点放在起始点条件
- 递归的关注点放在求解目标上

相同

- 重在表现第 i 次与第 $i+1$ 次的关系

用递归实现递推

优点

- 让程序变得简明

方法

- 把关注点放在要求解的目标上
- 找到第 n 次做与第 $n-1$ 次做之间的关系
- 确定第 1 次的返回结果

斐波那契数列

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144…

- **fab(n)= fab(n-1) + fab(n-2) ;**
- **fab(1)=1, fab(2)=1;**

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

模拟连续发生的动作

进制转换

- 将123转换成等值的二进制数：

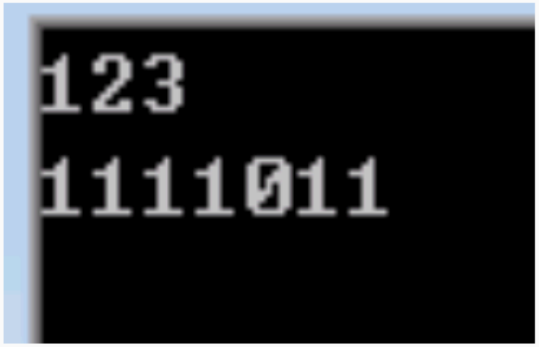
除以2的商（取整）	余数
123/2 = 61	1
61/2 = 30	1
30/2 = 15	0
15/2 = 7	1
7/2 = 3	1
3/2 = 1	1
1/2 = 0	1

- 自下而上收集余数：1111011

```
void convert(int x)
{
    if ((x / 2) != 0)
    {
        convert(x / 2);
        cout << x % 2;
    }
    else
        cout << x;
}
```

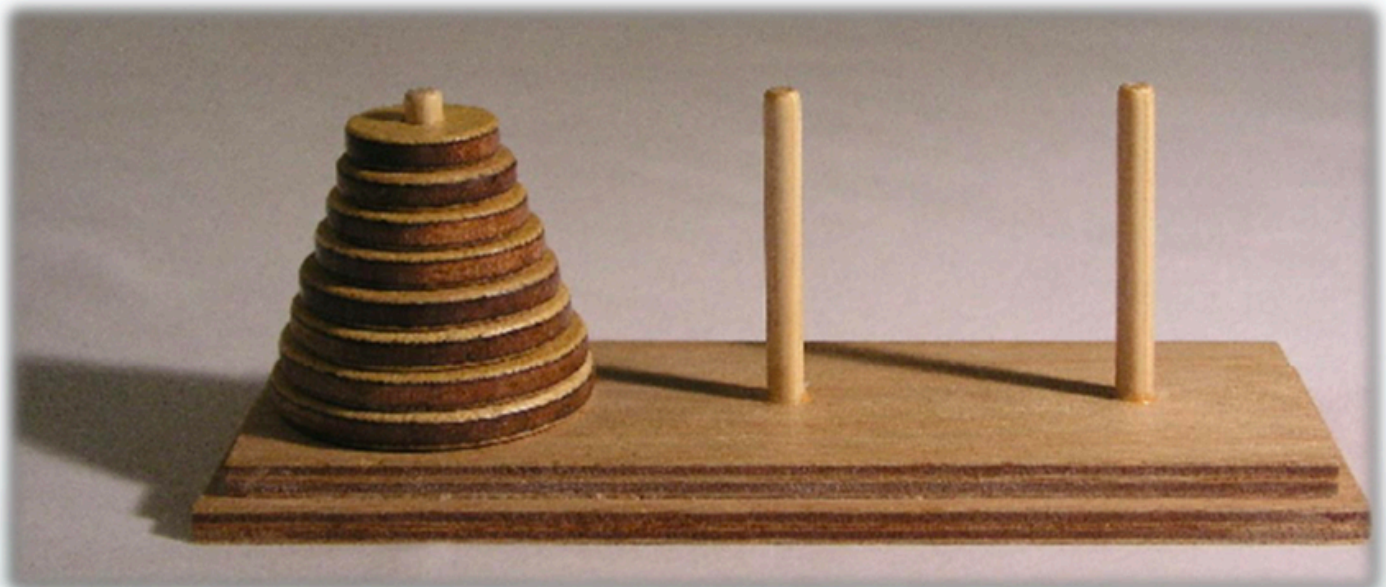
进制转换

```
#include<iostream>
using namespace std;
void convert(int x)
{
    if ((x / 2) != 0)
    {
        convert(x / 2);
        cout << x % 2;
    }
    else
        cout << x;
}
int main()
{
    int x;
    cin >> x;
    convert(x);
    return 0;
}
```

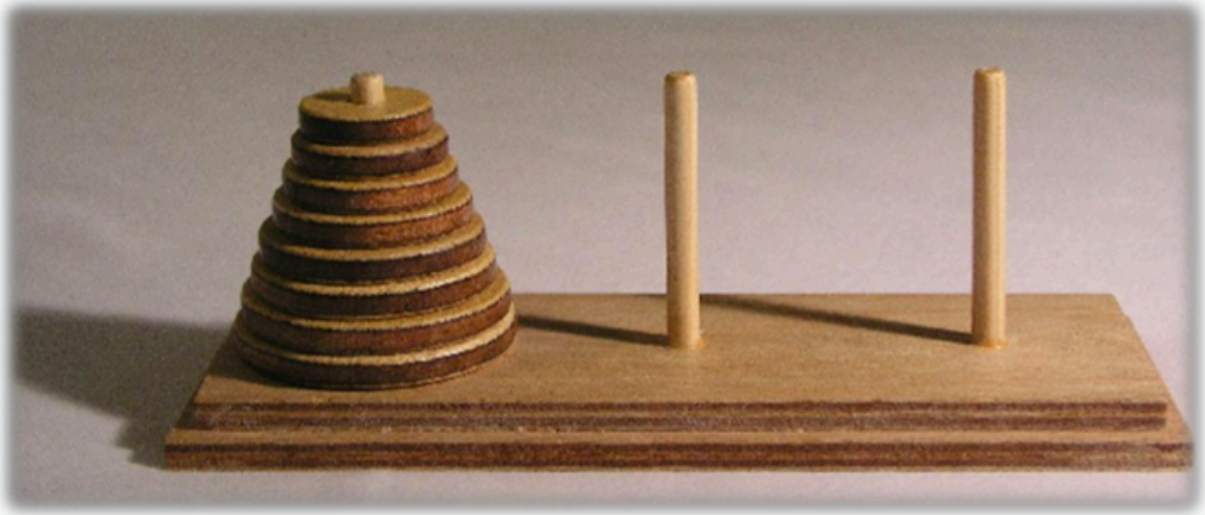


汉诺塔问题

- 故事：相传在古代印度的Bramah庙中，有位僧人整天把三根柱子上的金盘倒来倒去，他想把64个一个比一个小的金盘从一根柱子上移到另一根柱子上去。移动过程中恪守下述规则：每次只允许移动一只盘，且大盘不得落在小盘上面。
- 有人会觉得这很简单，真的动手移盘就会发现，如以每秒移动一只盘子的话，按照上述规则将64只盘子从一个柱子移至另一个柱子上，所需时间约为5800亿年。



汉诺塔问题



```
请输入盘数n = 4
在3根柱子上移4只盘的步骤为:
把一个盘子从 a 移动到 b
把一个盘子从 a 移动到 c
把一个盘子从 b 移动到 c
把一个盘子从 a 移动到 b
把一个盘子从 c 移动到 a
把一个盘子从 c 移动到 b
把一个盘子从 a 移动到 b
把一个盘子从 a 移动到 c
把一个盘子从 b 移动到 c
把一个盘子从 b 移动到 a
把一个盘子从 c 移动到 a
把一个盘子从 b 移动到 c
把一个盘子从 a 移动到 b
把一个盘子从 a 移动到 c
把一个盘子从 b 移动到 c
```

汉诺塔问题

要实现：move(n, A, B, C)

需进行：

move(n-1, A, C, B)
move from A to C
move(n-1, B, A, C)

```
void  move(int m, char x, char y, char z) //将m个盘子从A经过B移动到C
{
   if (m == 1)
   {
      cout << "把一个盘子从 " << x << " 移动到 " << z << endl;
   }
   else
   {
      move(m - 1, x, z, y);
      cout << "把一个盘子从 " << x << " 移动到 " << z << endl;
      move(m - 1, y, x, z);
   }
}
```

汉诺塔问题

```
#include<iostream>
using namespace std;
void  move(int m, char x, char y, char z)
{
   if (m == 1)
   {
      cout << "把一个盘子从 " << x << " 移动到 " << z << endl;
   }
   else
   {
      move(m - 1, x, z, y);
      cout << "把一个盘子从 " << x << " 移动到 " << z << endl;
      move(m - 1, y, x, z);
   }
}
int main() {
   int n;
   cout << "请输入盘数n = ";
   cin >> n;
   cout << "在3根柱子上移" << n << "只盘的步骤为:" << endl;
   move(n, 'A', 'B', 'C');
   return 0;
}
```

```
请输入盘数n = 4
在3根柱子上移4只盘的步骤为:
把一个盘子从 A 移动到 B
把一个盘子从 A 移动到 C
把一个盘子从 B 移动到 C
把一个盘子从 A 移动到 B
把一个盘子从 C 移动到 A
把一个盘子从 C 移动到 B
把一个盘子从 A 移动到 B
把一个盘子从 A 移动到 C
把一个盘子从 B 移动到 C
把一个盘子从 B 移动到 A
把一个盘子从 C 移动到 A
把一个盘子从 B 移动到 C
把一个盘子从 A 移动到 B
把一个盘子从 A 移动到 C
把一个盘子从 B 移动到 C
```

模拟连续发生的动作

- 搞清楚连续发生的动作是什么

```
1 void move(int m, char x, char y, char z)
```

- 搞清楚不同次动作之间的关系

```
1 move(m-1,x,z,y);
2 cout<<" 把一个盘子从"<<x<<" 移动到"<<z<<endl;
3 move(m-1,y,x,z);
```

- 搞清楚边界条件是什么

```
1 if(m==1) {
2     cout<<" 把一个盘子从"<<x<<" 移动到"<<z<<endl;
3 }
```

进行“自动的分析”

逆波兰表达式

题目描述

- 逆波兰表达式是一种把运算符前置的算术表达式：
  - 2+3的逆波兰表示法为+ 2 3
  - (2+3)*4的逆波兰表示法为* + 2 3 4
- 编写程序求解任一仅包含+ - * /四个运算符的逆波兰表达式的值。

输入

```
1 * + 11.0 12.0 + 24.0 35.0
```

输出

```
1 1357.0
```

逆波兰表达式

× ÷ + 12 36 + 1 3 − 15 8

[notation( )]

× [notation()][notation()]

÷ [notation()][notation()]

+[notation()][notation()]

× → notation( ) × notation( )

÷ → notation( ) ÷ notation( )

+ → notation( ) + notation( )

− → notation( ) − notation( )

其他字符：将读到的字符串解释为 数；

```
double notation()
{
   char str[10];
   cin >> str;
   switch (str[0])
   {
   case '+': return notation() + notation();
   case '-': return notation() - notation();
   case '*': return notation()* notation();
   case '/': return notation() / notation();
   default: return atof(str);
   }
}
```

逆波兰表达式

```
#include<iostream>
using namespace std;
double notation()
{
    char str[10];
    cin >> str;
    switch (str[0])
    {
    case '+': return notation() + notation();
    case '-': return notation() - notation();
    case '*': return notation()* notation();
    case '/': return notation() / notation();
    default: return atof(str);
    }
}
int main()
{
    cout << notation();
    return 0;
}
```

$$\times \div + 12\ 36 + 1\ 3 - 15\ 8$$

$$((12+36)\div(1+3))\times(15-8)$$

```
* / + 12 36 + 1 3 - 15 8
84
```

放苹果

题目描述

- 把M个同样的苹果放在N个同样的盘子里，允许有的盘子空着不放，问共有多少种不同的分法？
- 注意：5,1,1和1,5,1是同一种分法
- 输入：7 3
- 输出：8

问题

- M个苹果放入N个盘子，多少种放法？

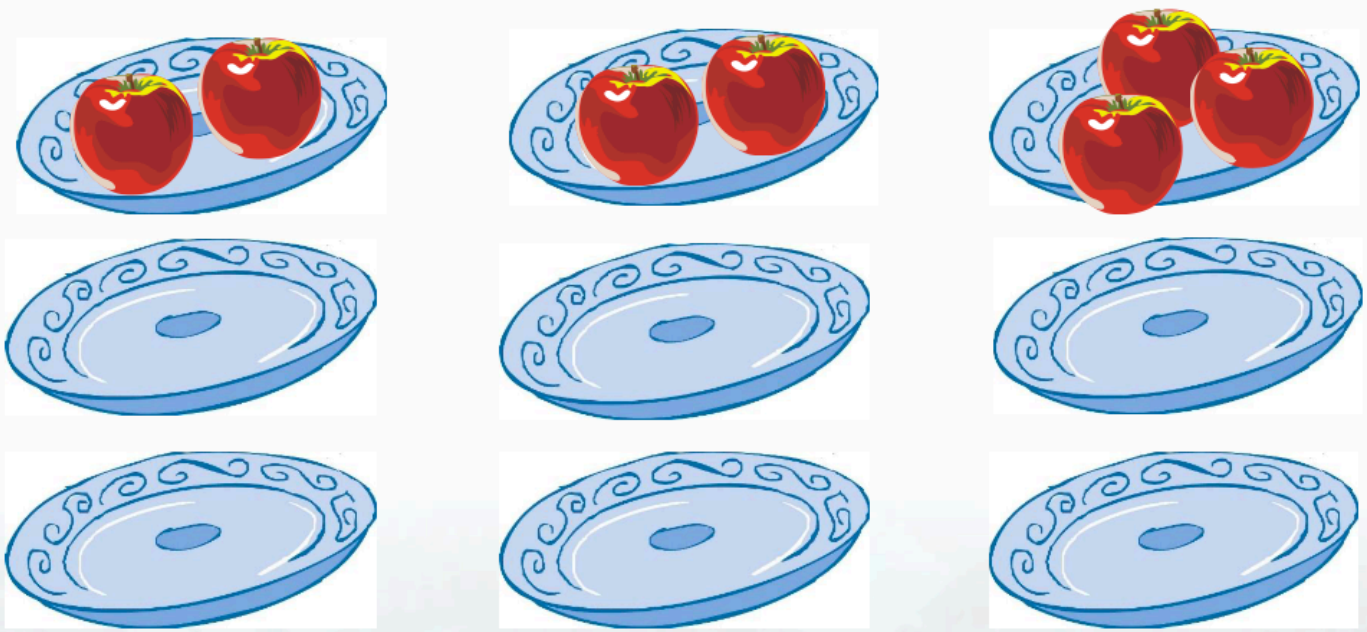
假设

- 有一个函数f(m,n)能告诉我答案

放苹果

如果：n/盘子数 > M/苹果数

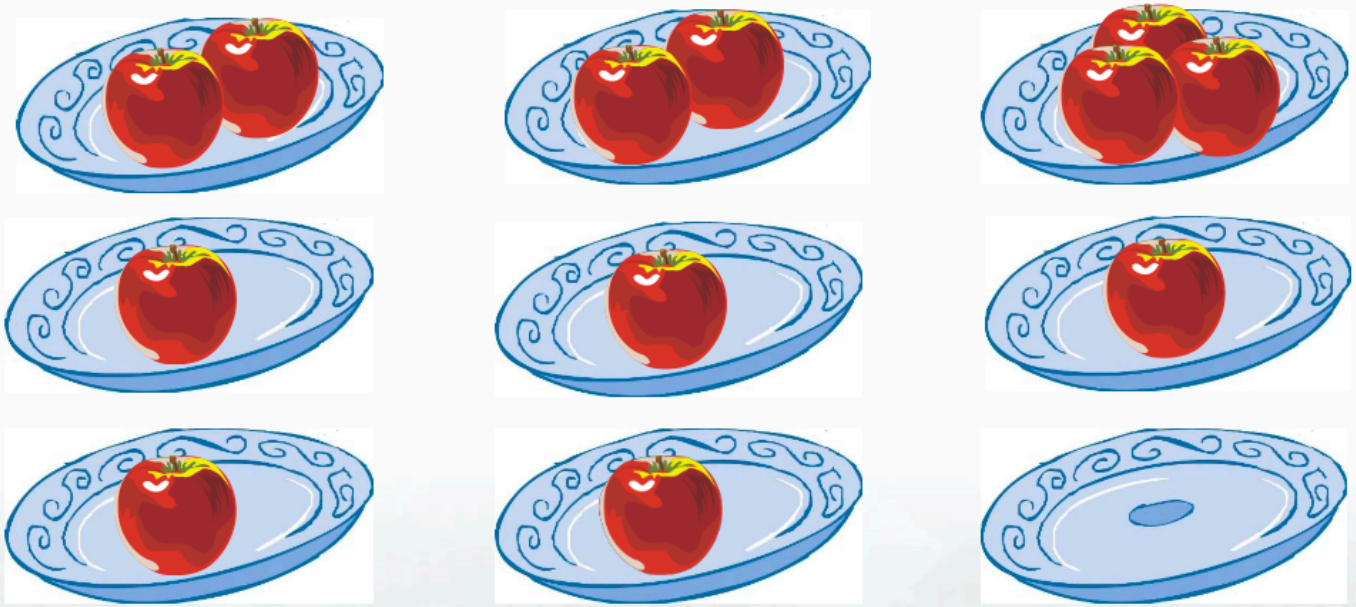
$$if(n>m)\ f(m,n) = f(m,m)$$



放苹果

如果：M/苹果数 >= n/盘子数

$$\begin{cases} \text{有盘子空着：} & f(m, n) = f(m, n-1) \\ \text{没盘子空着：} & f(m, n) = f(m-n, n) \end{cases}$$



放苹果

```
#include<iostream>
using namespace std;
int count(int m, int n)
{
    if(m==0||n==1) return 1;
    if(m<n) return count(m,m);
    else return (countm,n-1)+count(m-n,n);
}
int main()
{
    int apples,plates;
    cin>>apples>>plates;
    cout<<count(apples,plates);
    return 0;
}
```

利用递归进行“自动分析”

方法

- 先假设有一个函数能给出答案
  `notation() count(int m,int n)`
- 在利用这个函数的前提下，分析如何解决问题

```
'+': return notation() + notation();
'-': return notation() - notation();
'*': return notation()* notation();
'/': return notation() / notation();
```

```
if (m < n)
    return count(m, m);
else
    return count(m, n - 1) + count(m - n, n);
```

- 搞清楚最简单的情况下答案是什么
  `default: return atof(str); if(m<=1||n<=1) return 1;`

总结

递归的作用

- 用递归来完成递推
  - 把关注点放在要求解的目标上
  - 找到第 n 次与第 $n-1$ 次执行之间的关系
  - 确定第 1 次的返回结果
- 模拟连续发生的动作
- 进行“自动的分析”

总结

递归的作用

- 用递归来完成递推
- 模拟连续发生的动作
  - 搞清楚连续发生的动作是什么
  - 搞清楚不同次动作之间的关系
  - 搞清楚边界条件是什么
- 进行“自动的分析”

总结

递归的作用

- 用递归来完成递推
- 模拟连续发生的动作
- 进行“自动的分析”
  - 先假设有一个函数能给出答案
  - 在利用这个函数的前提下，分析如何解决问题
  - 搞清楚最简单的情况下答案是什么