讲稿（教学内容、步骤）

第 9 章 运行时存储组织

1. 运行时存储组织的作用与任务

- 作用：
  编译程序在生成目标代码时，应该明确程序的各类对象在逻辑空间内的存储，以及目标代码运行时是如何使用和支配自己的逻辑存储空间的。

- 任务：
  1) 数据对象的表示：明确源语言中各类数据对象在目标机中的表示形式。
  2) 表达式的计算：明确如何正确有效地组织表达式的计算过程。
  3) 存储分配策略：正确有效地分配不同作用域或不同生命周期的数据对象的存储。
  4) 过程实现：实现过程或函数调用以及参数传递。

2. 程序运行时存储空间的布局

- 为方便存储组织与管理，将存储空间从逻辑上划分为“代码区”和“数据区”。

- 不同的区域存放的数据和对应的管理方法不同。

3. 存储分配策略

(1) 静态存储分配

- 如果编译时就能确定目标程序运行中所需的全部数据空间大小，那么在编译时就安排好目标程序运行时的全部数据，并确定每个数据对象的存储位置。

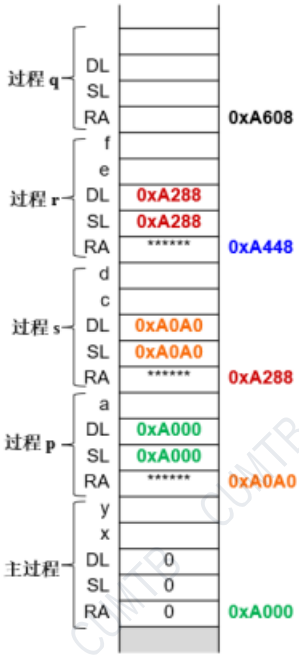
- 特点：
  1) 数据名所需存储空间大小在编译时就能确定
  2) 数据名的性质是完全确定的
  3) 现代程序语言可静态分配的数据对象包括大小固定且在程序执行期间可全程访问的全局变量、静态变量、程序中的常量、class 虚函数表等

(2) 栈式存储分配

- 将整个程序的数据空间设计成一个“运行栈”，运行栈数据空间的存储和管理方式称为“栈式存储分配”。
  - ✓ 每调用一个过程，该过程所需的数据空间分配在栈顶（这部分存储单元称为“活动记录”）
  - ✓ 该过程一结束就释放这部分活动记录数据空间

- 活动记录的数据空间内容包括：
  1) 存放本过程这次活动中的数据对象
     如：局部变量、参数、临时变量等

  2) 用于管理过程活动的记录信息
     如：静态链、动态链、返回地址等



【练习】写出如下程序运行过程的存储分配。

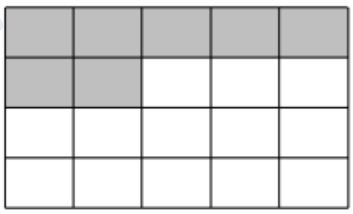
```
int  f(int a){
   static int x;
   x +=  a;
   return x;
}

int main(){
   int x, y;
   x = 1;

   y = f(x);
   cout<<y<<endl;

   y = f(x);
   cout<<y<<endl;
   return 0;
}
```

静态数据区



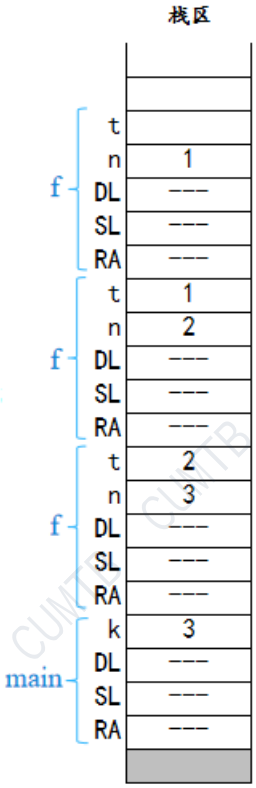
栈区

【练习】

练习：写出程序运行栈空间的存储分配过程。

```
int f(int n){
    int t;
    if(n<=1)
        return 1;
    else{
        t = n - 1;
        t = n * f(t);
        return t;
    }
}

void main(){
    int k=3;
    k=f(k);
    printf("%d", k);
}
```



(3) 堆式存储分配

1) 在堆式存储分配中，可以在任意时刻以任意次序从数据段的堆区分配和释放数据对象的运行时存储空间。
如：C++中的 new 和 delete

2) 空间分配未必服从“先申请后释放”原则。
即可以任意时刻以任意次序分配和释放数据对象的存储空间，因此程序运行一段时间之后堆存储空间可能被划分成许多块，有些被占用，有些空闲。

3) 对于堆存储空间的管理，通常需要好的存储分配算法：
- ✓ 最佳适应算法：选择空间浪费最少的存储块
- ✓ 最先适应算法：选择最先找到的足够大的存储块
- ✓ 循环最先适应算法：起始点不同的最先适应算法

4. 活动记录

(1) 过程活动记录

- 程序运行时，每进入一个过程，就在栈顶为其分配所需的数据空间；过程结束就释放该过程的数据空间。
- 一个过程的一次执行所需要的信息，使用一段连续的存储区来管理，这个区（块）就是一个过程的活动记录 AR（Activation Record）或栈帧（frame）。
- 通常活动记录 AR 中要记录：
  ① 临时工作单元：存放中间结果的临时单元。
  ② 局部变量

③ **实参**
④ **机器状态信息**：保存运行过程前的状态（寄存器值，程序计数器，……）
⑤ 控制链（**动态链**）：可选， 指向调用者的活动记录，释放栈。
⑥ 存取链（**静态链**）：可选， 用于非局部量的引用。
⑦ **返回地址**

(2) 嵌套过程定义中非局部量的访问

- 允许嵌套的过程定义/函数定义的语言
  如 Pascal、ML 等语言。

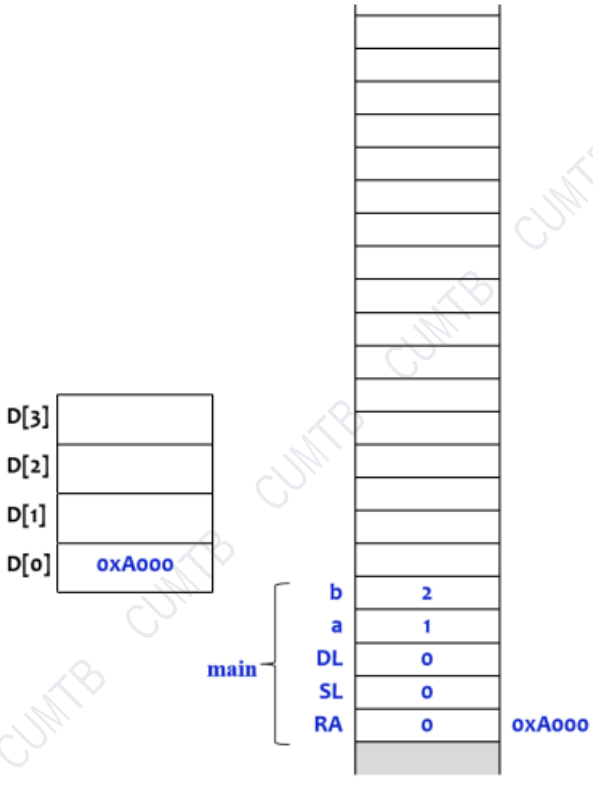
- 主要特点
  内层过程体可以访问包含它的任一外层过程中定义的数据对象（如变量，数组或过程等）。

- 引出的问题
  访问**不在当前活动记录中的数据对象**时如何处理，即“非局部量的访问”。

- 解决方法
  1) **Display 表**：每进入一个过程后，建立活动记录的同时建立一张“嵌套层次表 display”。
  【举例】利用 **Display 表** 解决非局部量的访问
  如：执行 d:=a+b 时的访问情况

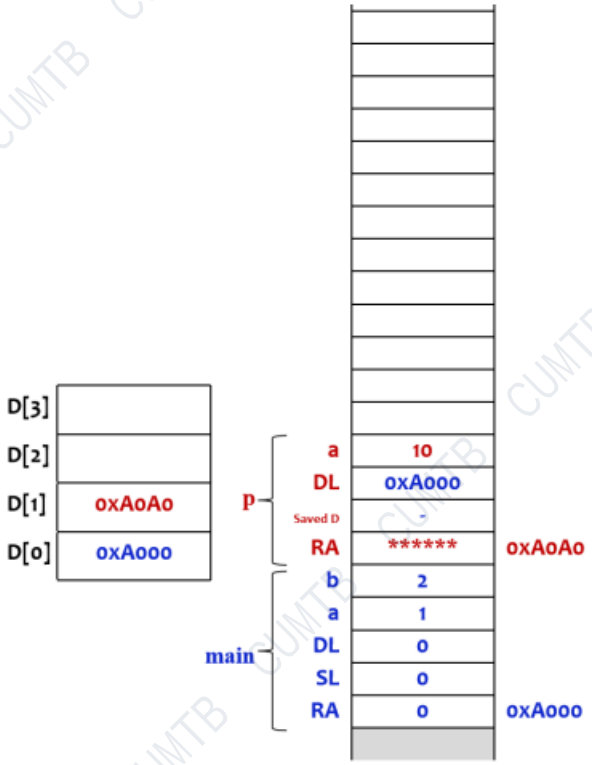
  ✓ 执行主过程的情况：

```
var a,b;
procedure p;
   var a;
    procedure q;
       var d;
       begin
          d:=a+b;
       end;
    procedure s;
       var c,d;
       procedure r;
          var e,f;
          begin
             call q;
          end;
       begin
          call r;
       end;
    begin
       a := 10;
       call s;
    end;
begin
   a := 1;
   b := 2;
    call p;
end.
```



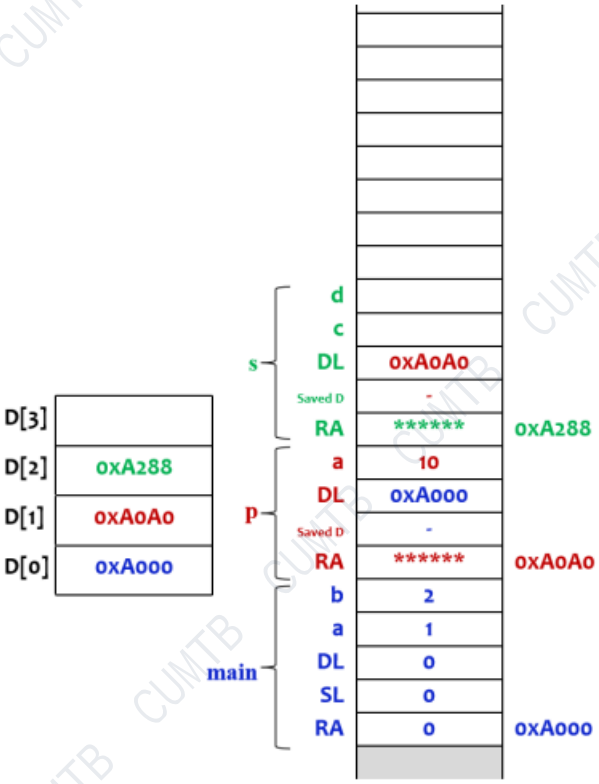
✓ 执行 p 过程的情况：

```
var a,b;
procedure p;
   var a;
    procedure q;
       var d;
       begin
          d:=a+b;
       end;
    procedure s;
       var c,d;
       procedure r;
          var e,f;
          begin
             call q;
          end;
       begin
          call r;
       end;
    begin
       a := 10;
      call s;
    end;
begin
   a := 1;
   b := 2;
    call p;
end.
```



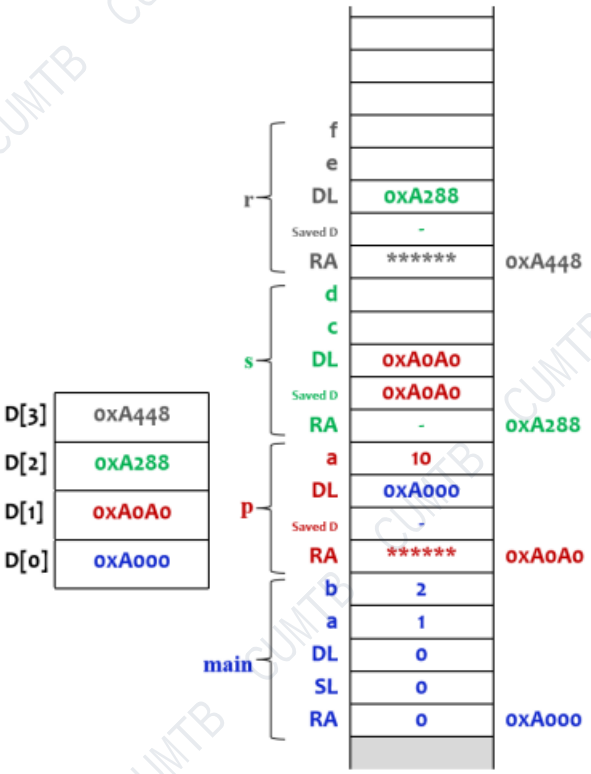
✓ 执行 s 过程的情况：

```
var a,b;
procedure p;
   var a;
    procedure q;
       var d;
       begin
          d:=a+b;
       end;
    procedure s;
       var c,d;
       procedure r;
          var e,f;
          begin
             call q;
          end;
       begin
          call r;
       end;
    begin
       a := 10;
      call s;
    end;
begin
   a := 1;
   b := 2;
    call p;
end.
```



✓ 执行 r 过程的情况：

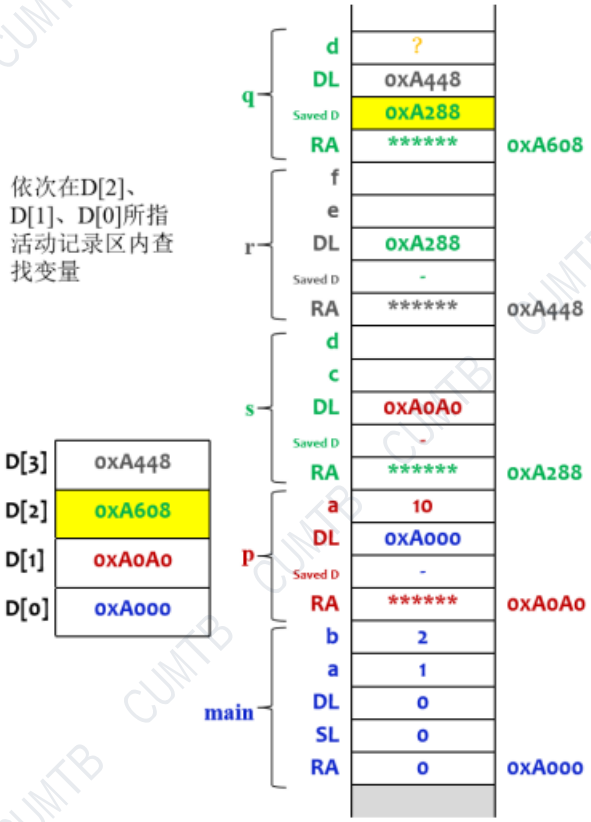
```
var a,b;
procedure p;
   var a;
    procedure q;
       var d;
       begin
          d:=a+b;
       end;
    procedure s;
       var c,d;
       procedure r;
          var e,f;
          begin
             call q;
          end;
       begin
          call r;
       end;
    begin
       a := 10;
      call s;
    end;
begin
   a := 1;
   b := 2;
    call p;
end.
```



✓ 执行 q 过程的情况：

```
var a,b;
procedure p;
   var a;
    procedure q;
       var d;
       begin
          d:=a+b;
       end;
    procedure s;
       var c,d;
       procedure r;
          var e,f;
          begin
             call q;
          end;
       begin
          call r;
       end;
    begin
       a := 10;
      call s;
    end;
begin
   a := 1;
   b := 2;
    call p;
end.
```

依次在D[2]、D[1]、D[0]所指活动记录区内查找变量

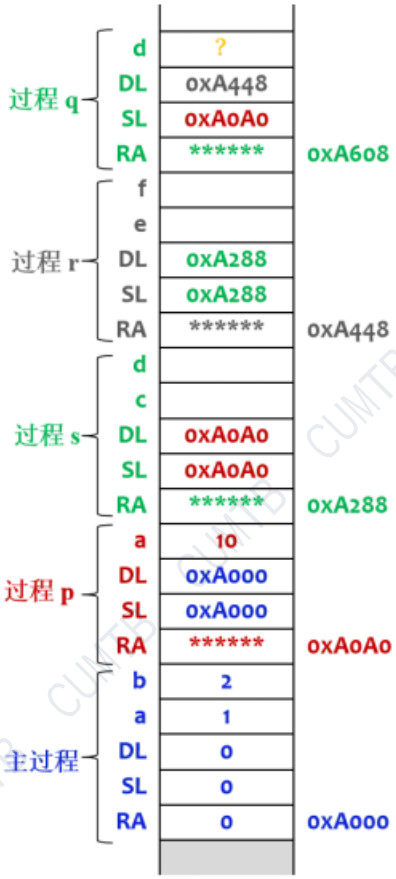


2) **静态链**：指向定义该过程的直接外过程运行时最新数据段的基地址。

【举例】利用 **静态链** 解决非局部量的访问

如：d:=a+b 时的访问情况

```
var a,b;
procedure p;
   var a;
    procedure q;
       var d;
       begin
          d:=a+b;
       end;
    procedure s;
       var c,d;
       procedure r;
          var e,f;
          begin
             call q;
          end;
       begin
          call r;
       end;
    begin
       a := 10;
      call s;
    end;
begin
   a := 1;
   b := 2;
    call p;
end.
```



- 非嵌套定义的情况及处理

  对于 C 语言等**不支持嵌套函数定义**的语言，非局部量只有全局变量，通常分配在静态数据区。

(3) 嵌套程序块的非局部量访问

- 嵌套程序块

  块的内部也允许声明局部变量。

【举例】

```
int p( )
{    int A;
     …
     { int  B, C;
       …
     }
     { int  D, E, F;
       …
       {      int G;
              …
       }
     }
}
```

- 同样存在“非局部量访问”的问题。

- 解决方法
  - ✓ 将每个块看作内嵌的无参过程，为它创建一个新的活动记录“块级活动记录”。
    缺点：代价高。
  - ✓ 借用其所属的过程级记录即可。

5. 过程调用

(1) 过程调用的**初始化**阶段：

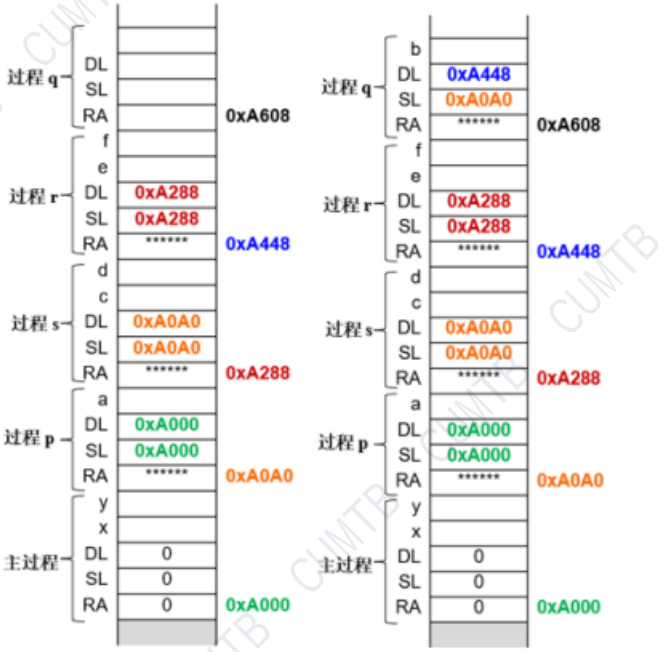
- 实现过程调用的控制信息
  - ✓ 调用程序返回地址：用于保证当前过程或函数运行结束时返回到调用过程继续执行。
  - ✓ 必要的联系单元（动态链、静态链、display 表等）

- 实现过程调用的数据信息
  - ✓ 实参
  - ✓ 寄存器保护区

- 被调过程的活动记录初始化时，需要包含控制信息、数据信息、静态局部数据。

(2) 调用的**起始阶段**需要完成的任务

① 参数传递；
② 分配栈上的存储空间；
③ 保存旧栈的活动记录基址（动态链）；
④ 保存返回地址；
⑤ 保存其他控制信息（静态链、Display 表等）；
⑥ 保存寄存器信息；
⑦ 建立新栈基址；
⑧ 建立新栈顶；
⑨ 转移控制，启动被调过程的执行。



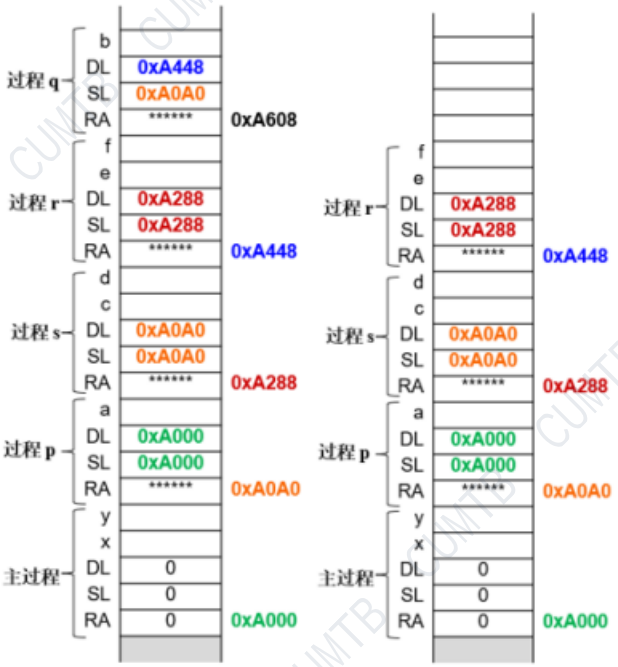
(3) 调用的**收尾阶段**需要完成的任务

① 若被调是函数，则返回值写入专门的寄存器；

② 恢复所有被调过程保存的寄存器；

③ 弹出被调过程的活动记录，恢复旧活动记录；

④ 将控制返回给调用过程。



(4) **参数传递**

- 在实现过程调用时，参数传递方式是很重要的环节。
- 常见参数传递方式：

① 传值（值调用）

✓ 形参被作为函数的局部变量，在被调过程的活动记录中开辟了形参的存储单元。

✓ 调用过程时，把实参的值赋值给对应形参的存储单元。对形参的任何运算**不影响实参的值**。

【举例】

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

② 传址（地址引用）

✓ 形参用于存放实参对应的地址。

✓ 调用过程时，把实参的地址赋值给对应形参的存储单元。对形参的运算**直接影响实参的值**。

【举例】

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

另例：形参是数组，就是把数组的首地址传递给形参

6. PL/0 编译程序的运行时存储组织

【举例】

```
var x,y;
procedure p;
  var a;
  procedure q;
              var b;
              begin
                          b:=10;
              end;
  procedure s;
              var c,d;
              procedure r;
                          var e,f;
                          begin
                            call q;
                          end;
              begin
                          call r;
              end;
  begin
              call s;
  end;
begin
  call p;
end.
```

●动态链（控制链）：指向调用该过程的活动记录的基址。

●静态链（访问链）：指向定义该过程的直接外过程运行时最新的活动记录的基址。

画出执行到b:=10时的运行栈。

【本章小结】

1. 存储组织的任务
   (1) 数据对象的表示
   (2) 表达式的计算
   (3) 存储分配策略
   (4) 过程实现
2. 存储空间的布局

3. 存储分配
   (1) 静态
   (2) 栈式
   (3) 堆式
4. 活动记录 AR
   (1) 过程
   (2) 嵌套过程定义
   (3) 嵌套程序块
5. 过程调用
   (1) 初始化阶段
   (2) 起始阶段
   (3) 收尾阶段
   (4) 参数传递（传值、传址）

【课堂讨论】

(1) 讨论 C 的变量作用域

```
1.  /*  scope.c  */
2.  #include <stdio.h>
3.  int a = 1;

4.  void func(){
5.          a = 2;
6.          //b = 3;      //【讨论】若执行，会怎样？
7.          int a = 3;   //【讨论】允许声明一个同名的局部变量吗？
8.          int b = a;   //【讨论】这个a是哪个？
9.          printf("in func a=%d b=%d \n", a, b);
10. }

11. int b = 4; //b的作用域从这里开始
12. int main(int argc, char **argv){
13.         printf("test1: a=%d b=%d \n", a, b); //【讨论】a、b分别是哪个？

14.         func();
15.         printf("test2: a=%d b=%d \n", a, b);

16.         int a = 5;
17.         int b = 6;
18.         printf("test3: a=%d b=%d \n", a, b); //【讨论】a、b分别是哪个？

19.         if (a > 0){
20.                 int b = 7;
21.                 printf("in if: a=%d b=%d \n", a, b); //【讨论】a、b分别是哪个？
22.         }else{
23.                 int b = 8;
24.                 printf("in else: a=%d b=%d \n", a, b); //【讨论】a、b分别是哪个？
25.         }

26.         printf("test4: a=%d b=%d \n", a, b); //【讨论】a、b分别是哪个？
27. }
```

输出：

```
test1: a=1 b=4
in func: a=3 b=3
test2: a=2 b=4
test3: a=5 b=6
in if: a=5 b=7
test4: a=5 b=6
```

(2) 讨论 java 的变量作用域

```
/* Scope.java */

public class ScopeTest{
        public static void main(String args[]){
                int a = 1;
                int b = 2;

                if (a > 0){
                        int b = 3;  //【讨论】是否允许执行？ 声明的变量不允许与外部变量同名
                        int c = 3;  //【讨论】是否允许执行？ 允许声明新变量
                }
                else{
                        int c = 4;
                }

                int c = 5;  //【讨论】是否允许执行？ 声明的变量允许与之前的内部变量同名
        }
}
```

(3) 讨论 javaScript 的变量作用域

```
/*  Scope.js  */

var x = 5;
var y = 5;
console.log("t1: x=%d y=%d", x, y);

if (x > 0) {
   x = 4;
   console.log("t2: x=%d y=%d", x, y);
   var y = 3;    //【讨论】这里是否声明了一个新变量y?
   console.log("t3: x=%d b=%d", x, y);
}else {
   var y = 4;
   console.log("t4: x=%d y=%d", x, y);
}

console.log("t5: x=%d y=%d", x, y);

for (var y = 0; y < 2; y++){   //【讨论】这里是否声明了一个新变量y?
   console.log("t6: x=%d y=%d", x, y);
}

console.log("t7: x=%d y=%d", x, y);
```

输出：

```
t1: x=5 y=5
t2: x=4 y=5
t3: x=4 y=3
t5: x=4 y=3
t6: x=4 y=0
t6: x=4 y=1
t7: x=4 y=2
```

javascript没有块作用域，这两处看似重新定义了新的变量 y，但实际上还是外部已声明过的变量 y