# 第六章　运行时的存储组织及管理

- 概述
- 静态存储分配
- 动态存储分配

# 6.1 概述

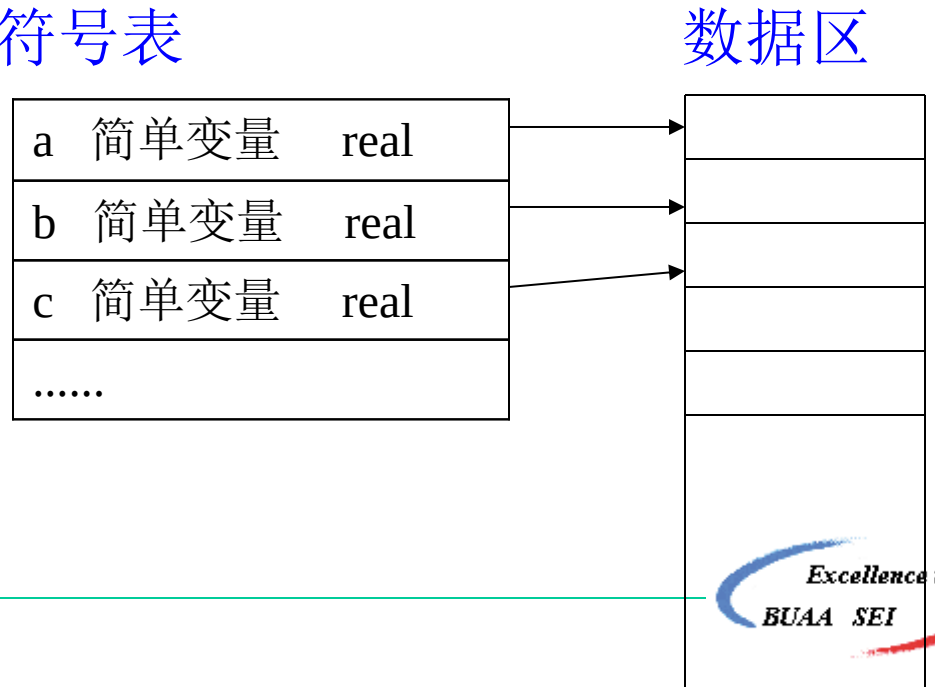
## (1) 运行时的存储组织及管理

目标程序运行时所需存储空间的组织与管理以及源程序中变量存储空间的分配。

例：real a, b, c ;
...
a := b*c ;

⇩

取b
* c
送a



## (2) 静态存储分配和动态存储分配

### 静态存储分配

在编译阶段由编译程序实现对存储空间的管理和为源程序中的变量分配存储的方法。

### 条 件

如果在编译时能够确定源程序中变量在运行时的数据空间大小，且运行时不改变，那么就可以采用静态存储分配方法。

**但是并不是所有数据空间大小都能在编译过程中确定**

## 动态存储分配

在目标程序运行阶段由目标程序实现对存储空间的组织与管理，和为源程序中的变量分配存储的方法。

## 特　点

- 在目标程序运行时进行变量的存储分配。
- 编译时要生成进行动态分配的目标指令。

# 6.2 静态存储分配

## (1) 分配策略

由于每个变量所需空间的大小在编译时已知，因此可以用简单的方法给变量分配目标地址。

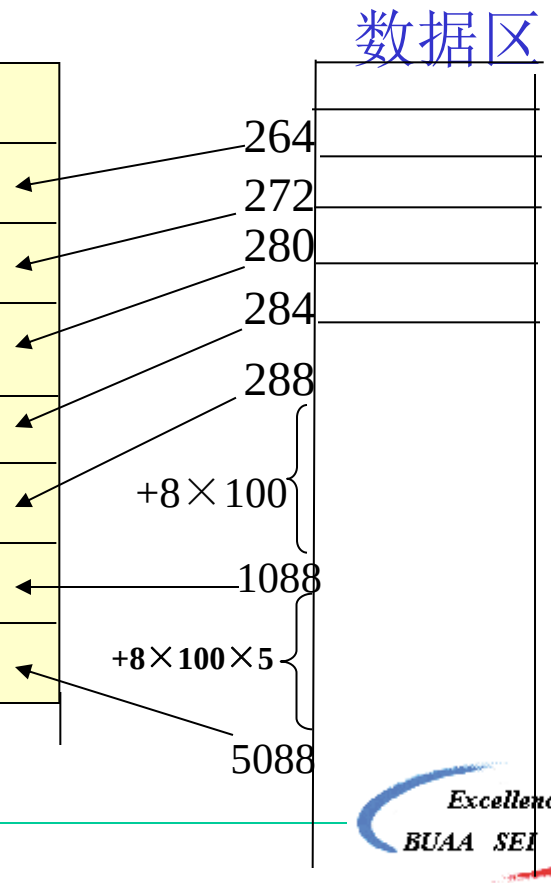
- 开辟一数据区。（首地址在加载时定）
- 按编译顺序给每个模块分配存储空间。
- 在模块内部按顺序给模块的变量分配存储，一般用相对地址，所占数据区的大小由变量类型决定。
- 目标地址填入变量的符号表中。

例：有下列FORTRAN 程序段

```
real        MAXPRN, RATE
integer     IND1, IND2
real        PRINT(100),YPRINT(5,100),TOTINT
```

假设整数占4个字节大小，实数占8个字节大小，则符号表中各变量在数据区中所分配的地址为：

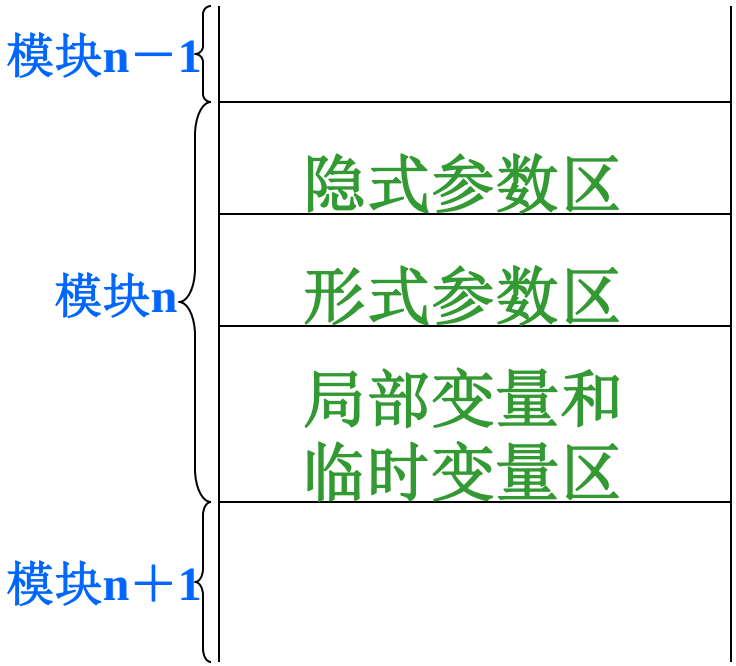
| 名字 | 类型 | 维数 | 地址 |
|---|---|---|---|
| MAXPRN | r | 0 | 264 |
| RATE | r | 0 | 272 |
| IND1 | i | 0 | 280 |
| IND2 | i | 0 | 284 |
| PRINT | r | 1 | 288 |
| YPRINT | r | 2 | 1088 |
| TOTINT | r | 0 | 5088 |



## (2) 模块(FORTRAN子程序)的完整数据区

- 变量
- 返回地址
- 形式参数
- 临时变量

# FORTRAN子程序的典型数据区




隐式参数区:返回地址
函数返回值

形式参数区:存放相应实参信息(值或地址)

# 6.3 动态存储分配

- 编译时不能具体确定程序所需数据空间
- 编译程序生成有关存储分配的目标代码
- 实际上的分配要在目标程序运行时进行

分程序结构，且允许递归调用的语言：

栈式动态存储分配

分配策略： 整个数据区为一个堆栈，

(1) 当进入一个过程时，在栈顶为其分配一个数据区。

(2) 退出时，撤消过程数据区。

(1)当进入一个过程时，在栈顶为其分配一个数据区。
(2)退出时，撤消过程数据区。

例1:

```
1  BBLOCK;                                   (1)
       REAL  X, Y;  STRING   NAME;
2      M1: PBLOCK(INTEGER IND) ;             (2)
           INTEGER   X;
                                             (3)
           CALL  M2(IND+1);
                                             (7)
       EDN  M1;
3      M2:  PBLOCK(INTEGER J);               (3)
4          BBLOCK;                           (4)
               ARRAY  INTEGER F(J);
               LOGICAL TEST1;
                                             (5)
           END
       END  M2;                              (6)
                                             (2)
       CALL  M1(X/Y);
END;                                         (8)
```

AR4 F，TEST1数据区

# 运行中数据区的分配情况：


(a)


(b)


(c)


(d)


(e)


(f)

```
1 BBLOCK;
    REAL X,Y;  STRING  NAME;
  M1: PBLOCK(INTEGER  IND);
2       INTEGER  X;
            CALL  M2(IND+1);
  END  M1;
  M2:  PBLOCK(INTEGER J);
3       BBLOCK;
4           ARRAY  INTEGER F(J);
            LOGICAL TEST1;
          END ;
  END  M2;
    CALL  M1(X/Y);
END
```


(g)

# 6.3.1 活动记录

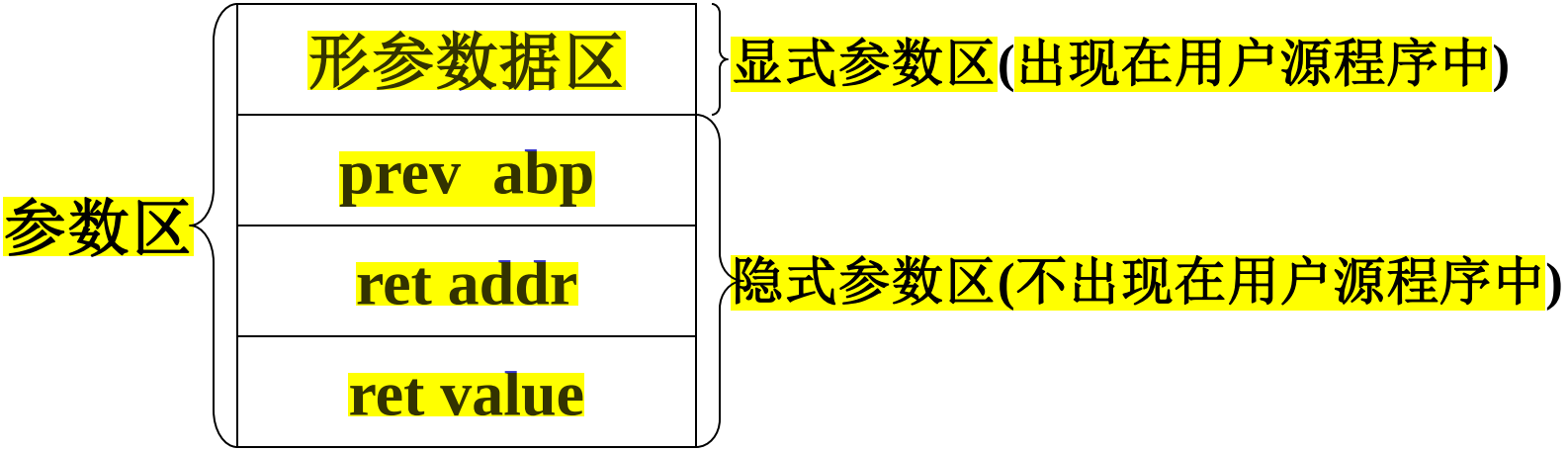
一个典型的活动记录可以分为三部分：

| 局部数据区 |
| --- |
| 参数区 |
| display区 |

(1) 局部数据区：

存放模块中定义的各个局部变量。

(2) 参数区： 存放隐式参数和显式参数。



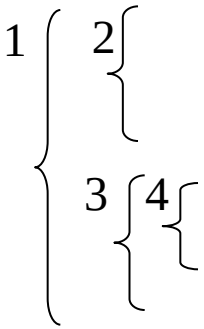

prev abp ： 存放调用模块记录基地址,函数执行完时,释放其数据区，数据区指针指向调用前的位置

ret addr：返回地址，即调用语句的下一条执行指令地址

ret value ： 函数返回值(无值则空)

形参数据区： 每一形参都要分配数据空间,形参单元中存放实参值或者实参地址

(3) display区：存放各外层模块活动记录的基地址。




对于例1中所举的程序段，模块4可以引用模块1和模块3中所定义的变量，故在模块4的display，应包括AR1和AR3的基地址。

变量二元地址(BL、ON)

BL：变量声明所在的层次。

可得到该层数据区开始地址

并列过程具有相同层次

ON：相对于显式参数区的开始位置的位移。

相对地址

例如：程序块1
X：（1，0）
Y：（1，1）
NAME：（1，2）

过程块M1
IND：（2，0）
X：（2，1）

```
BBLOCK;
    REAL X,Y;  STRING  NAME;
    M1: PBLOCK(INTEGER  IND);
        INTEGER  X;
        CALL  M2(IND+1);
    END  M1;
    M2:  PBLOCK(INTEGER J);
        BBLOCK;
            ARRAY  INTEGER F(J);
            LOGICAL TEST1;
        END ;
    END  M2;
    CALL  M1(X/Y);
END
```

(1) (2) (3) (4)

高层(内层)模块可以引用低层(外层)模块中的变量，例如在M1中可引用外层模块中定义的变量Y。

在M1的display区中可找到程序块1的活动记录基地址, 加上Y在数据区的相对地址就可以求得Y的绝对地址。

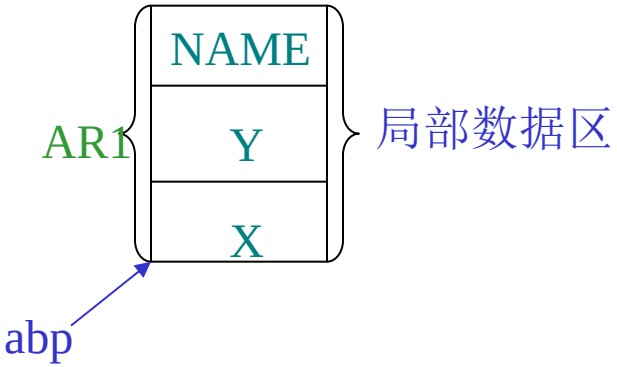
```
1 BBLOCK;
    REAL X,Y;  STRING  NAME;
  M1: PBLOCK(INTEGER  IND);
2       INTEGER   X;
            CALL  M2(IND+1);
  END  M1;
  M2:  PBLOCK(INTEGER J);
3         BBLOCK;
4           ARRAY  INTEGER F(J);
            LOGICAL TEST1;
          END ;
  END  M2;
    CALL  M1(X/Y);
END
```

程序的目标程序运行时，
栈)的跟踪情况：

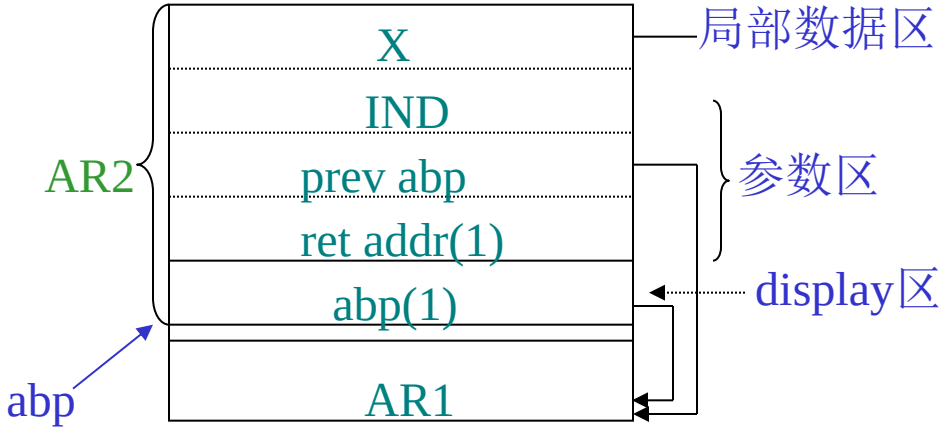
```
1 X, Y,   NAME;

  2 M1: ( IND) ;
      X;
    CALL  M2;

  3 M2: ( J);
      4 ARRAY  F(J);
        TESTI;
CALL  M1
```




(a) 进入模块1

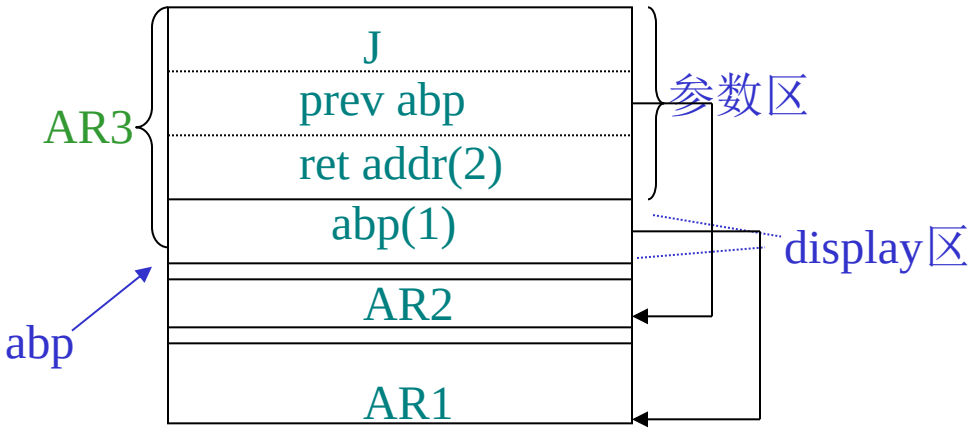



(b) M1被调用

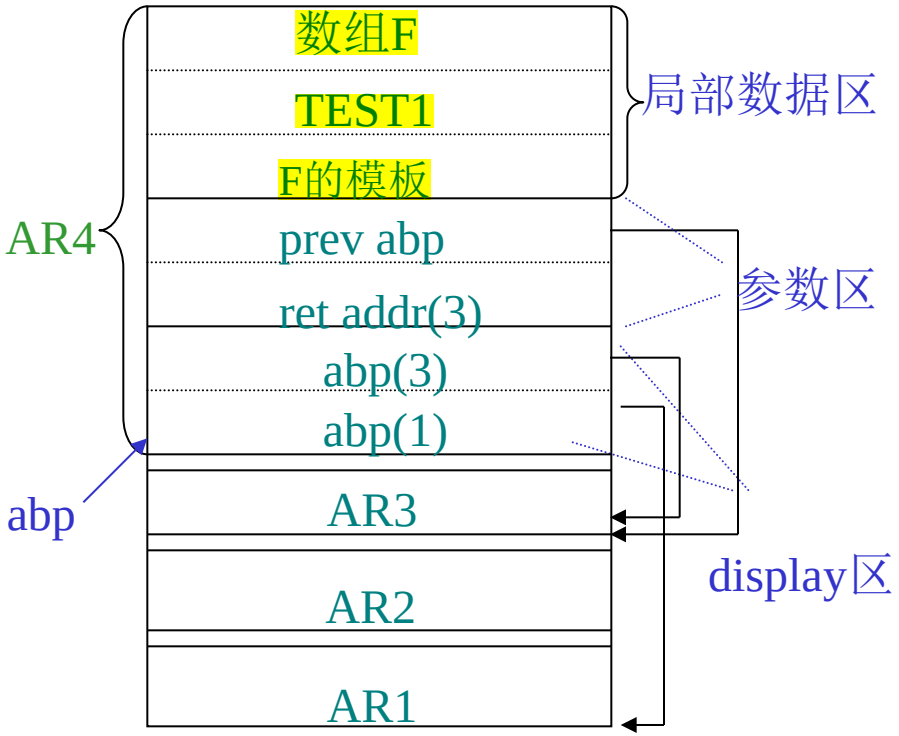
```
1  X, Y,   NAME;

   2  M1: ( IND) ;
        X;
      CALL  M2;

   3  M2: ( J);
        4  ARRAY  F(J);
           TESTI;
   CALL  M1
```




(c)M2被调用




(d) 进入内模块4

（e）当模块4执行完，则abp:=prev abp，这样abp恢复到进入模块4时的情况，运行栈情况如（c）

（f）当M2执行完，则abp:=prev abp，这样abp恢复到进入M2时的情况，运行栈情况如（b）

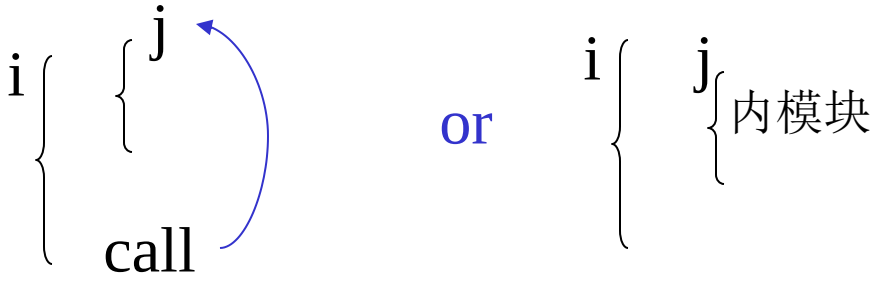
（g）当M1执行完，则abp:=prev abp，这样abp恢复到进入M1时的情况，运行栈情况如（a）

（h）当最外层模块执行完，运行栈恢复到进入模块时的情况，运行栈空

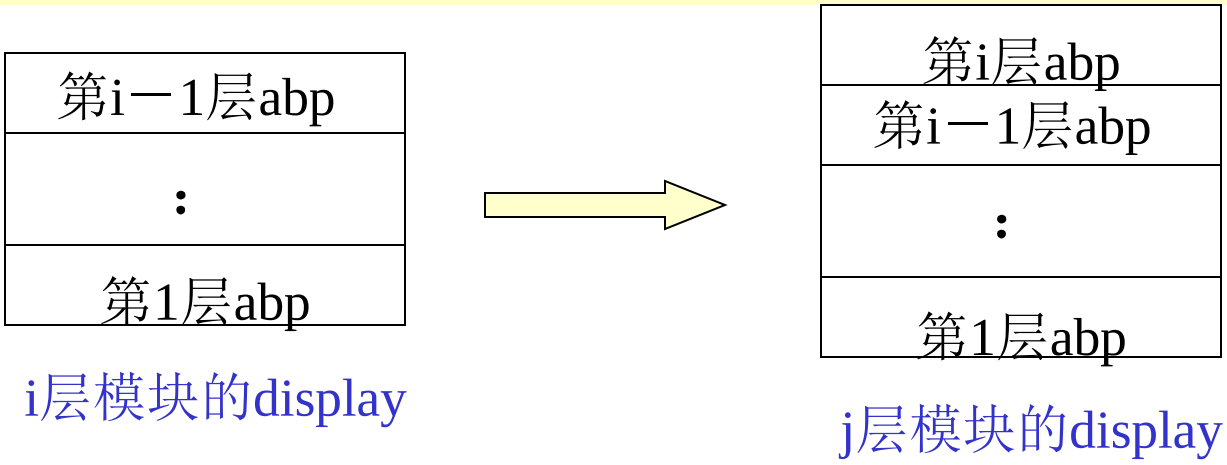
# 6.3.2 建造display区的规则

从i层模块进入(调用)j层模块，则：

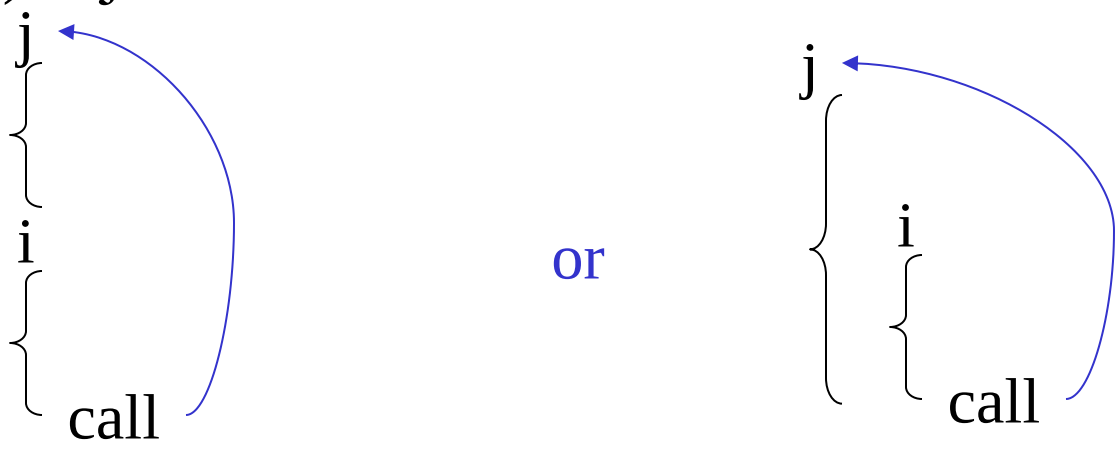
(1) 若j＝i＋1



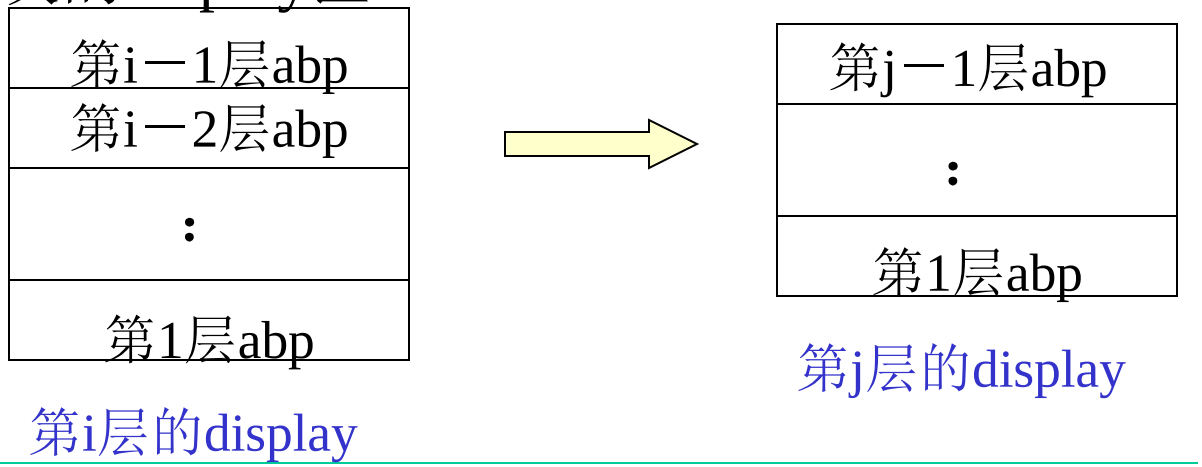

复制i层的display，然后增加一个指向i层模块记录基地址的指针



(2) 若j≤i 即调用外层模块或同层模块




将i层模块的display区中的前面j－1个入口复制到第j层模块的display区

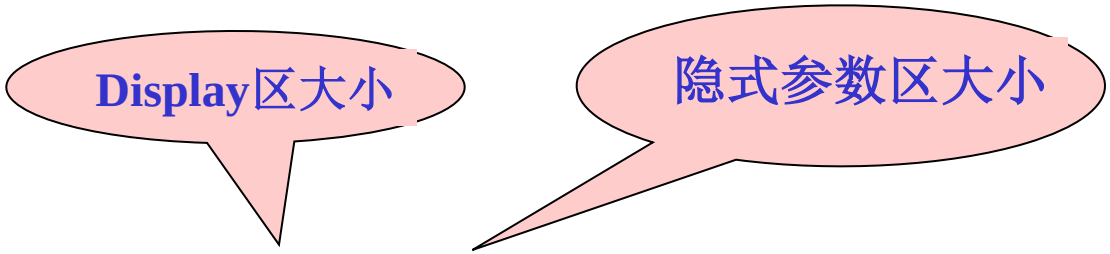


# 6.3.3 运行时的地址计算

设要访问的变量的二元地址为：（BL，ON)

该变量在LEV层模块中引用

地址计算公式：




```
if  BL = LEV  then
            addr := abp + (BL-1) + nip + ON
else  if  BL < LEV  then
            addr := display[BL] + (BL-1) + nip + ON
else
     write(“地址错，不合法的模块层次”）
```

作业：p119　1

P133 2,3

# C语言的运行时存储管理

# C语言的运行时存储管理

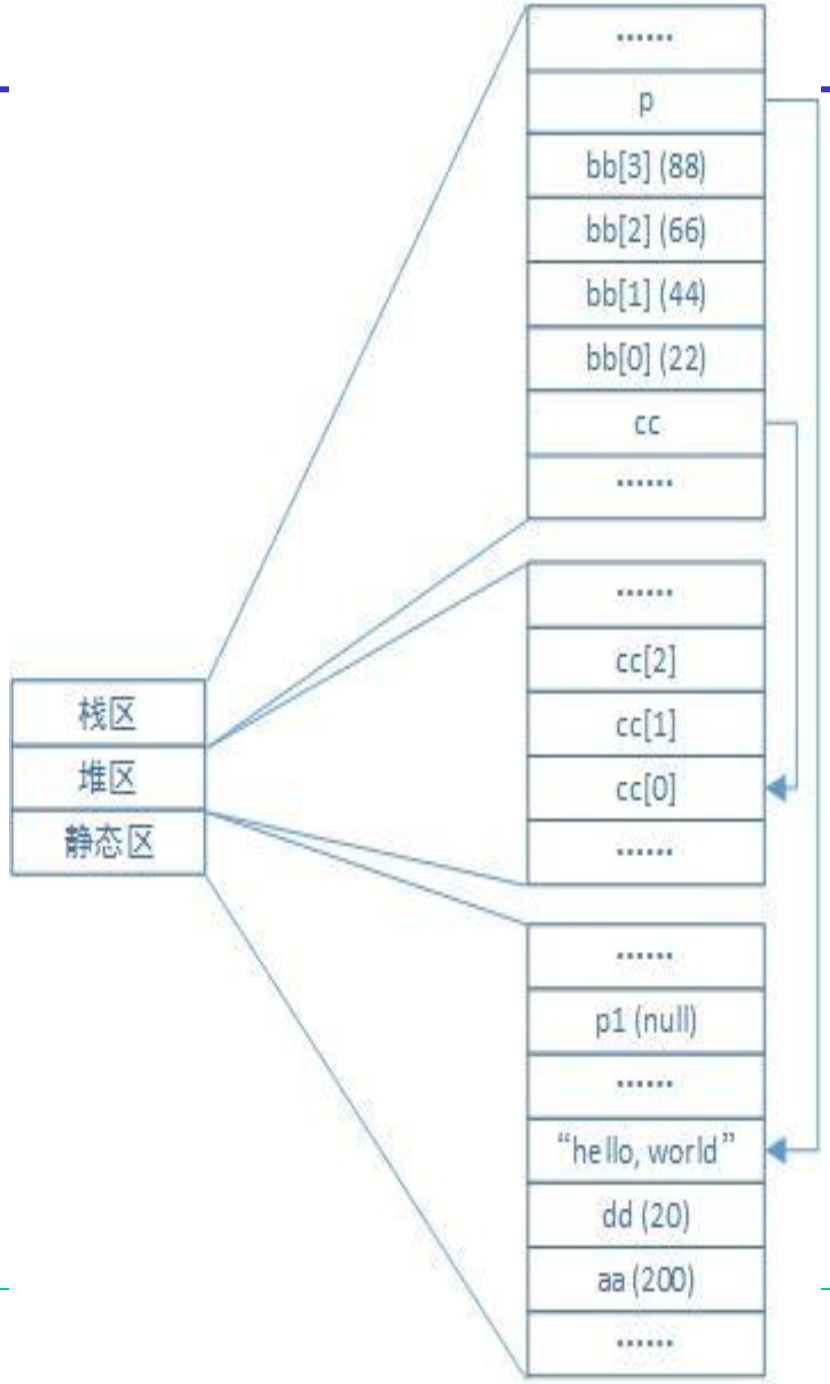
```
#include <iostream.h>
int dd=20;     //已初始化全局变量，在静态区
char *p1;      //未初始化全局变量，在静态区
int main()
{
    static int aa = 200;  //全局静态变量，在静态区
    char *p = "hello, world";  //p在栈区，"hello, world"在静态区
    int bb[] = {22, 44, 66, 88}; //bb在栈区，占四个字节
    int *cc;   //局部变量，在栈区
    cc = new int[3];  //由new动态分配的内存在堆区，三个字节
}
```

P159 1,2; P166 2,3

上面的代码在运行时的存储管理如右图所示。

图中，上方为高地址，
下方为低地址。

```
#include <iostream.h>
int dd=20;     //已初始化全局变量，在静态区
char *p1;      //未初始化全局变量，在静态区
int main()
{
    static int aa = 200;  //全局静态变量，在静态区
    char *p = "hello, world";  //p在栈区，“hello, world”在静态区
    int bb[] = {22, 44, 66, 88}; //bb在栈区，占四个字节
    int *cc;   //局部变量，在栈区
    cc = new int[3];  //由new动态分配的内存在堆区，三个字节
}
```



# 栈式分配和堆式分配的比较

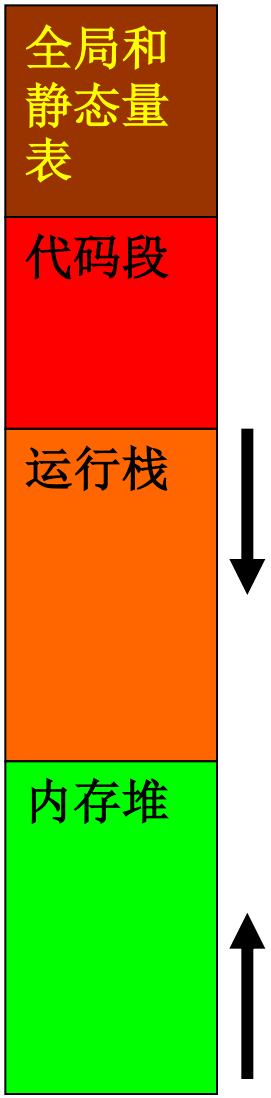
| 栈 | 堆 |
| --- | --- |
| 解决了函数的递归调用等问题 | 解决了动态申请空间的问题 |
| 由编译器自动管理 | 由程序员控制空间的申请和释放工作 |
| 向内存地址减少的方向增长 | 向内存地址增加的方向增长 |
| 不会产生碎片 | 会产生碎片 |
| 计算机底层支持，分配效率高 | C函数库支持，分配效率低 |

# 补充：运行时的存储管理

- 全局和静态量表
- 代码段
- 运行栈
- 内存堆

# 运行时的存储管理

- 以MS-WIN为例，从高地址到低地址，自上而下的是：
  - 全局和静态量表
  - 代码段
  - 运行栈
  - 内存堆



# 全局和静态量表

```
int global_c = 0 ;

void foo(int a)
{
    static int s_c = 0 ;

    s_c += a ;

    global_c = s_c ;
}
```

```
00427e34        global_c
00427e38        s_c
…
```

```
…
12:     s_c += a ;
00401028  mov     eax,[global_c+4 (00427e38)]
0040102D  add     eax,dword ptr [ebp+8]
00401030  mov     [global_c+4 (00427e38)],eax
13:
14:     global_c = s_c ;
00401035  mov     ecx,dword ptr [global_c+4 (00427e38)]
0040103B  mov     dword ptr [global_c (00427e34)],ecx
…
```

# 运行栈

- 子程序/函数运行时所需的基本空间
- 进入子程序/函数时分配，地址空间向下生长（从高地址到低地址）
- 从子程序/函数返回时，当前运行栈将被废弃
- 递归调用的同一个子程序/函数，每次调用都将获得不同的运行栈空间

# 运行栈

- 一个典型的运行栈包括
  - 函数的返回地址
  - 全局寄存器的保存区
  - 临时变量的保存区
  - 未分配到全局寄存器的局部变量的保存区
  - 为调用函数预留的参数传递区
  - 其他辅助信息的保存区
    - 例，PASCAL类语言的DISPLAY区

# 一个XScale上的Java/C/C++函数运行栈的示意图

# 内存堆

- 内存堆用来存放哪些数据？
  - 函数/子程序活动结束时仍需保持的数据
  - 程序运行前无法得知所需空间大小的数据
  - 并非全局量或者静态量

# 内存堆：例

```
25:             p = (char*)malloc(len) ;
004010A9    mov         eax,dword ptr [ebp+8]
004010AC    push        eax
004010AD    call        malloc (00401150)
004010B2    add         esp,4
004010B5    mov         dword ptr [ebp-4],eax
26:
27:             return p ;
004010B8    mov         eax,dword ptr [ebp-4]
```

**eax = 0x00031000**

# 垃圾收集GC——Java核心技术之一

**Java**的垃圾回收机制是**Java虚拟机**提供的能力，用于在空闲时间以**不定时**的方式动态回收无任何引用的**对象占据的内存空间**

目的：清除不再使用的对象

原理：通过确定对象**是否被活动对象引用**来确定是否收集该对象

主要步骤：

- 判断对象是否可以收集
- 垃圾回收后的内存组织

# 判断对象是否可回收

## 1. 引用计数——早期策略

技术原理：堆中的每个对象都有一个引用计数

运行机制：

① 当对象创建时且分配给一个变量时，该变量计数设置为1

② 任何变量被复制为该对象的引用时，计数+1

③ 当该对象某个引用超过生命周期或者被设置为新值时，计数-1

④ 任何引用计数为0的对象可以被当做垃圾收集

⑤ 当一个对象被垃圾收集时，它引用的任何对象计数-1

优点：引用计数可以很快的执行，对程序不被长时间打断的实时环境比较有利。

缺点：无法检测出循环引用。例如父子相互引用，其引用计数永远不可能为0.

# 判断对象是否可回收

## 2. 跟踪——目前常用策略

运行机制：对象引用遍历

从一组对象开始，沿着整个对象图上的每条链接，递归确定可到达的对象。如某对象不能从这些跟对象中的至少一个到达，则将它作为垃圾回收。在遍历阶段，GC需要标记对象。

# 垃圾回收后的内存组织

## 1. 标记-清除收集

直接清除，产生大量内存碎片，浪费内存

## 2. 标记-压缩收集

将标记对象复制到堆栈的新域中压缩堆栈，需要停止内存的其他操作。

## 3. 复制收集

将堆栈分为两个域，一个进行操作，一个进行垃圾回收后的复制。持续复制长生存期对象导致效率降低，而且使用半空间方式，空间利用率不高。

## 4. 增量收集

将堆栈分为多个域，每次仅从一个域收集垃圾。降低每次垃圾收集对应用程序的中断时间。

## 5. 分代收集

将堆栈分为两个或者多个域，用以存放不同寿命的对象。Java虚拟机生成的新对象一般放在其中的某个域中。过一段时间，继续存在的对象将获得使用期并转入更长寿命的域中。不同域使用不同的算法以优化性能。