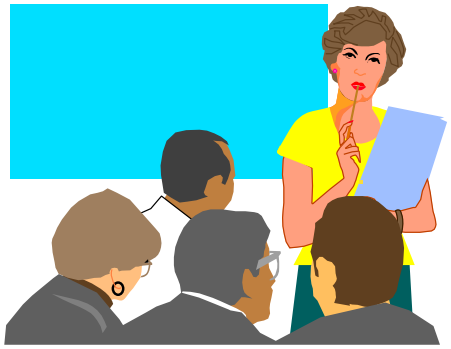
编译技术



胡春明
hucm@buaa.edu.cn

2019.9-2019.12

编译过程

编译过程是指将高级语言程序翻译为等价的目标程序的过程。

习惯上是将编译过程划分为5个基本阶段：

理解程序在运行时的存储与组织

程序设计语言中包含“高级语言”的结构

- 函数调用
- 对象、结构体（Struct）
- 丰富的数据类型
- 数组

……

目标环境只包含有限的“低级”结构

- 数据环境： fixed (8bit, 16bit, 32bit…, signed/unsigned), float
- 控制跳转： jmp

“运行时存储组织与管理”要回答

- 数据结构在内存里是如何表示的
- 函数在内存中是如何表示的
- 他们在内存中如何放置，如何管理

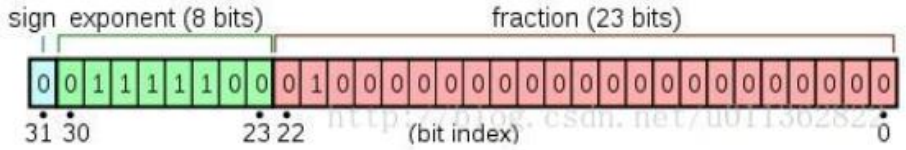
“运行时存储组织与管理”要回答

- 数据结构在内存里是如何表示的
- 函数在内存中是如何表示的
- 他们在内存中如何放置，如何管理

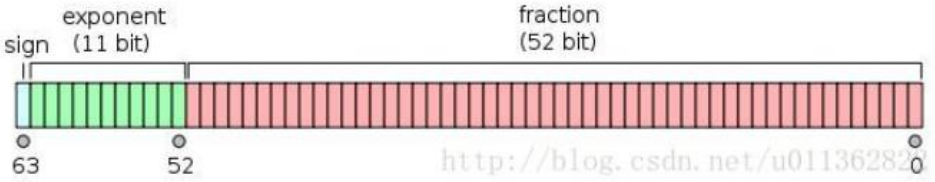
基本数据类型（Windows 32系统中）

类型	大小（字节）
bool	1
char	1
short	2
int	4
long	4
float	4
double	8

float



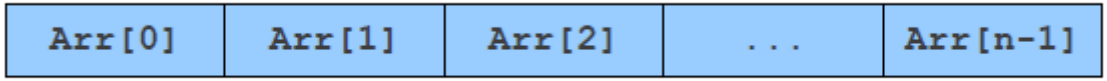
double



“运行时存储组织与管理”要回答

数组

C-style



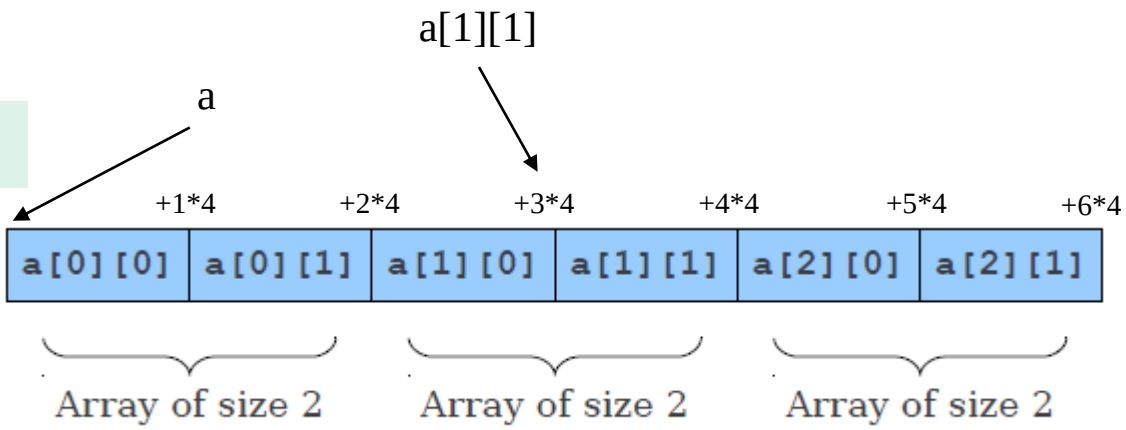
Java-style

Source: Stanford CS143 (2012)

“运行时存储组织与管理”要回答

多维数组

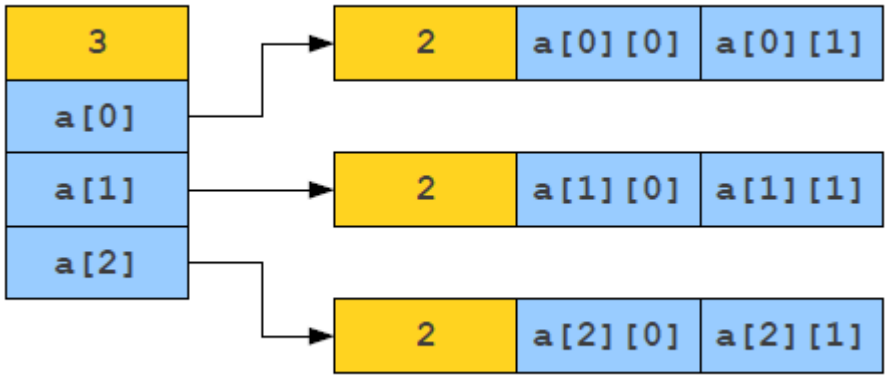
```
int a[1][1];
```



C-style

```
int[][] a = new int [3][2];
```

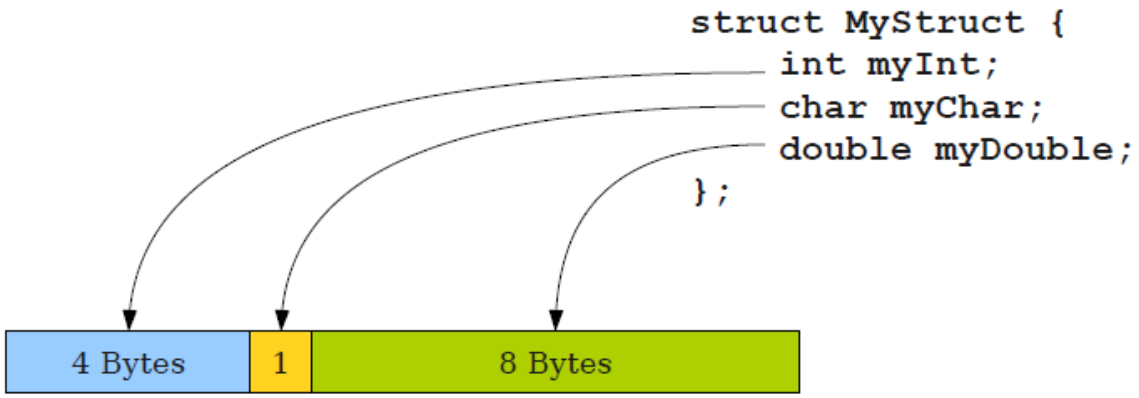
Java-style

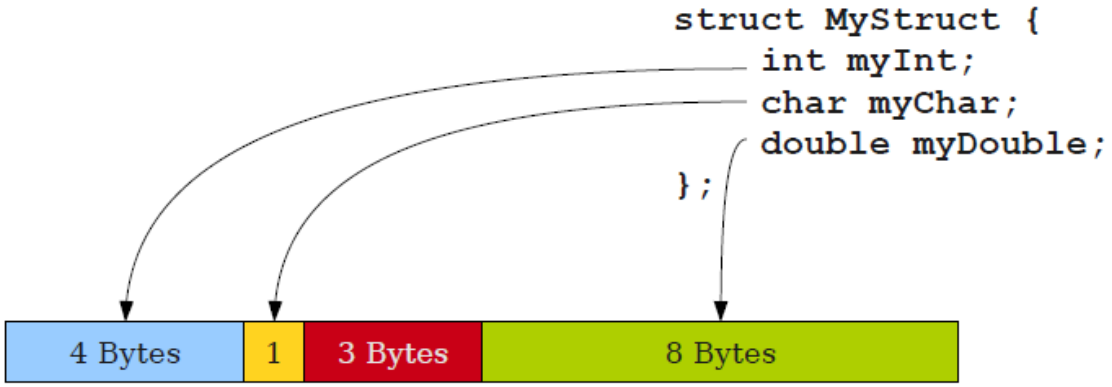


Source: Stanford CS143 (2012)

“运行时存储组织与管理”要回答

结构体

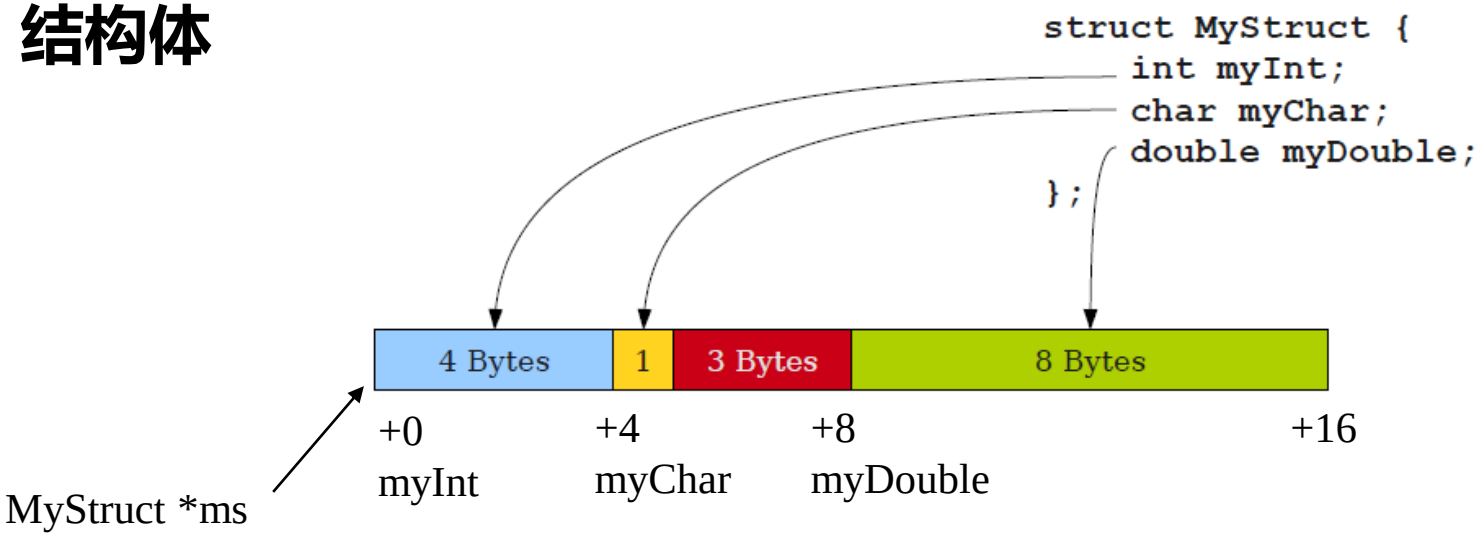




Source: Stanford CS143 (2012)

“运行时存储组织与管理”要回答

结构体



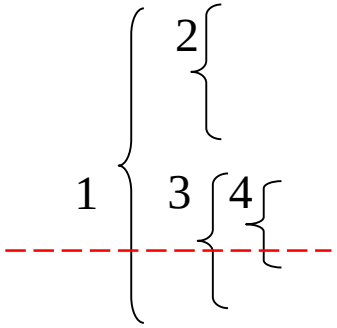

```
MyStruct *ms = new MyStruct;
ms -> myInt = 137;
ms -> myChar = ‘A’;
ms -> myDouble = 2.71
```

Source: Stanford CS143 (2012)

“运行时存储组织与管理”要回答

```
5  using namespace std;
6
7  int x=137;
8  int y=42;
9
10 void Function1() {
11     cout << x+y << endl;
12     return;
13 }
14
15 void Function2() {
16     int x=0;
17     Function1();
18 }
19
20 void Function3() {
21     int y=0;
22     Function2();
23 }
24
25 int main() {
26     Function1();
27     Function2();
28     Function3();
29     return 0;
30 }
```

函数及函数调用



需求1：需要记住调用前的状态，以便在调用结束后返回

需求2：需要提供一种办法，在调用函数和被调函数之间传递参数，返回值

需求23：满足 Scope 的要求，能够找到所有的数据结构

第六章 运行时的存储组织及管理

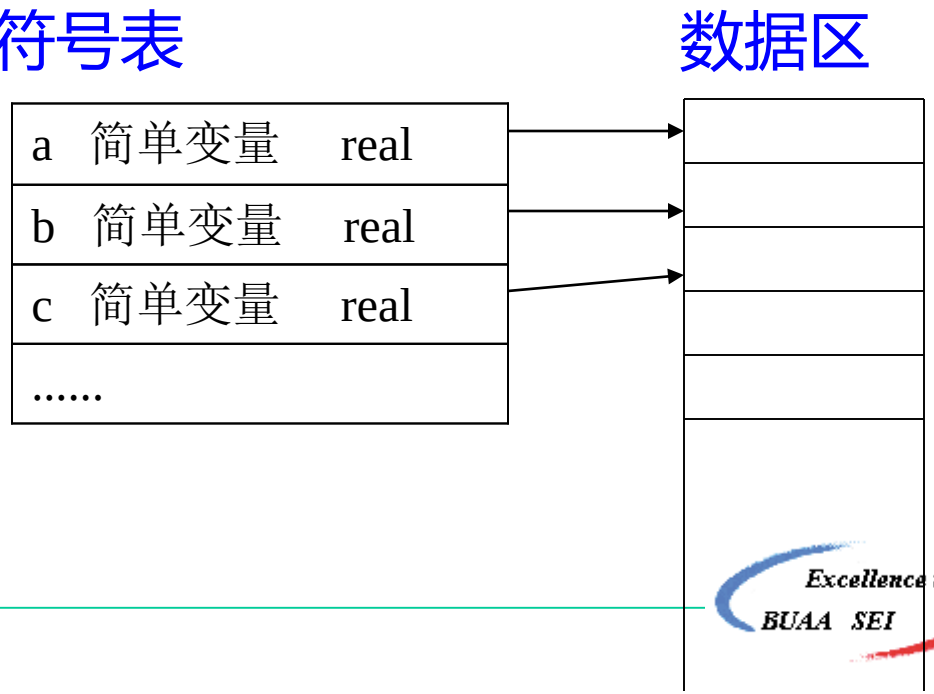
- 概述
- 静态存储分配
- 动态存储分配

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。

静态存储分配

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。

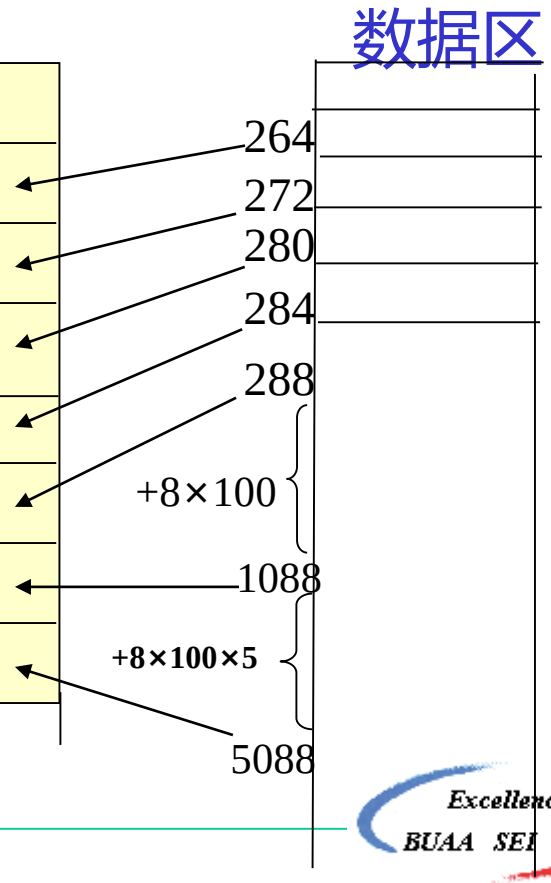
- 开辟一数据区。（首地址在加载时定）
- 按编译顺序给每个模块分配存储空间。
- 在模块内部按顺序给模块的变量分配存储，一般用相对地址，所占数据区的大小由变量类型决定。
- 目标地址填入变量的符号表中。

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：

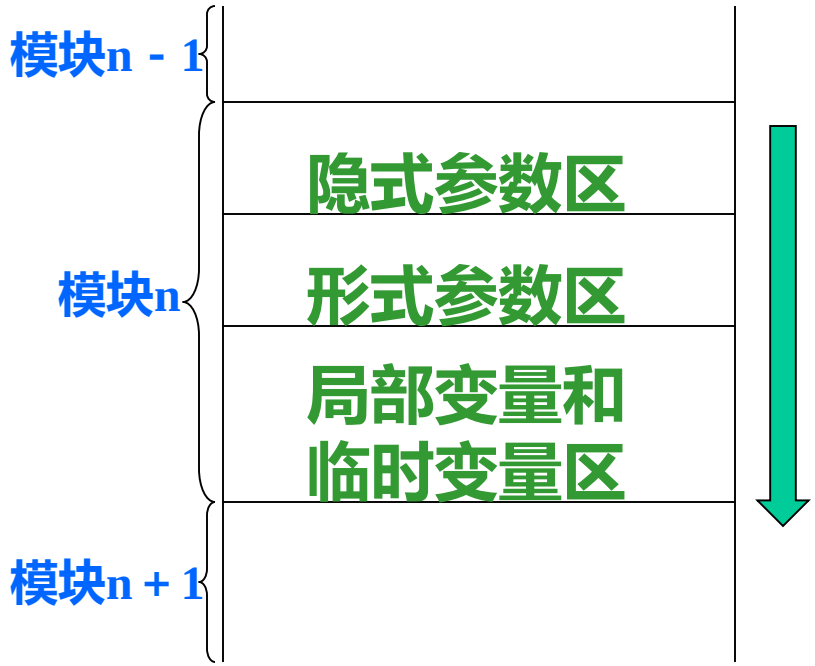
名字	类型	维数	地址
MAXPRN	r	0	264
RATE	r	0	272
IND1	i	0	280
IND2	i	0	284
PRINT	r	1	288
YPRINT	r	2	1088
TOTINT	r	0	5088



(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)

动态存储分配

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)

(g)

```
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
```

6.3.1 活动记录

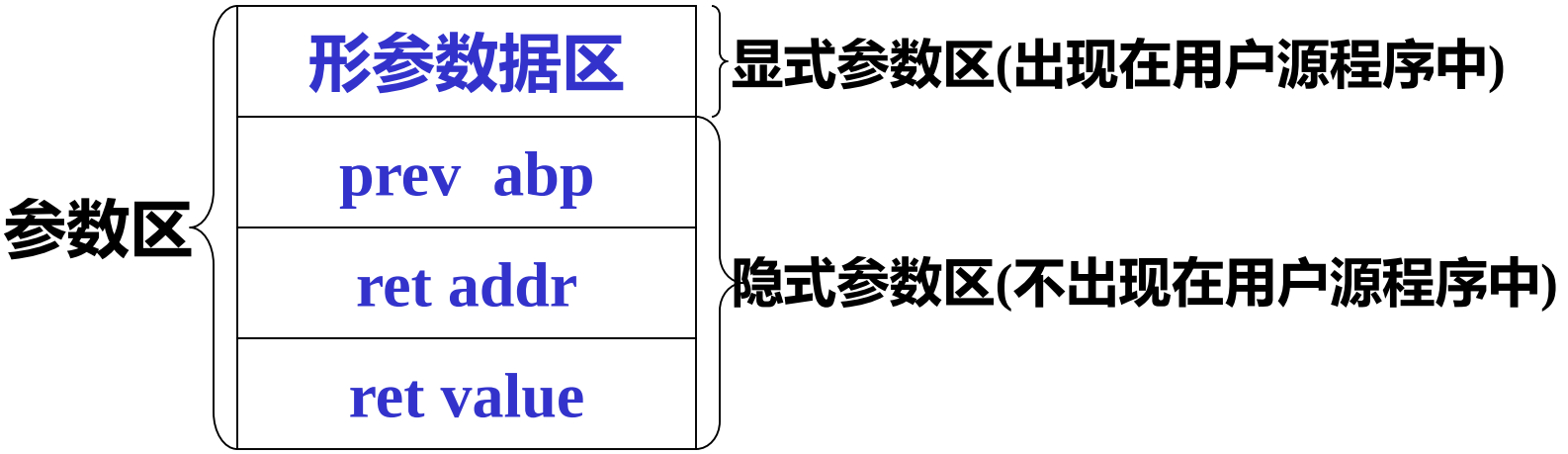
一个典型的活动记录可以分为三部分：

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。



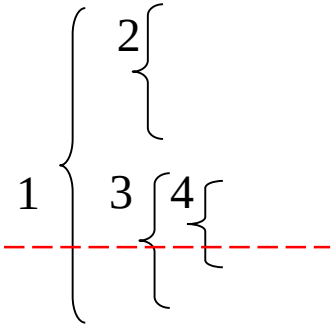
prev abp： 存放调用模块记录基地址,函数执行完时,释放其数据区，数据区指针指向调用前的位置

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）

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。

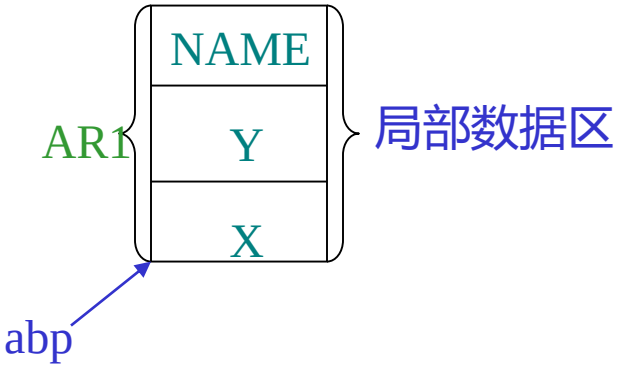
在M1的display区中可找到程序块1的活动记录基地址, 加上Y在数据区的相对地址就可以求得Y的绝对地址。

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

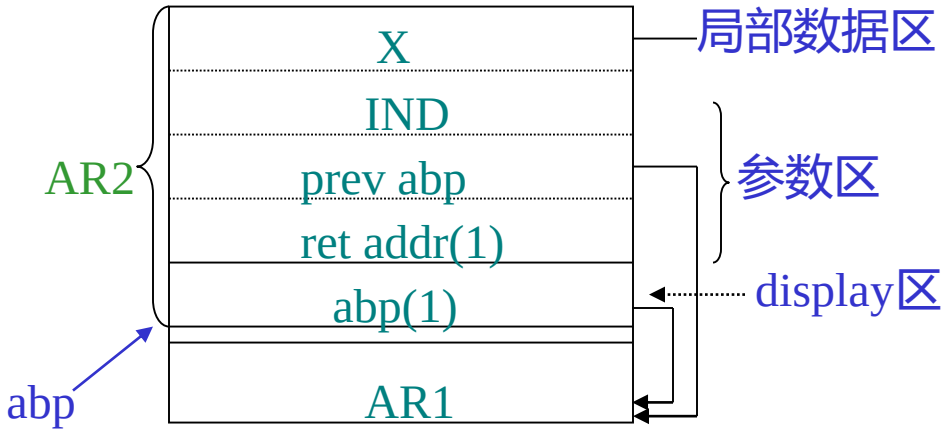
```
1 BBLOCK;
    REAL X,Y;  STRING  NAME;
  M1: PBLOCK(INTEGER  IND);
2       INTEGER   X;
            CALL  M2(IND+1);
  END  M1;
  M2:  PBLOCK(INTEGER J);
3        BBLOCK;
4          ARRAY  INTEGER F(J);
           LOGICAL TEST1;
         END ;
  END  M2;
    CALL  M1(X/Y);
END
```

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
```

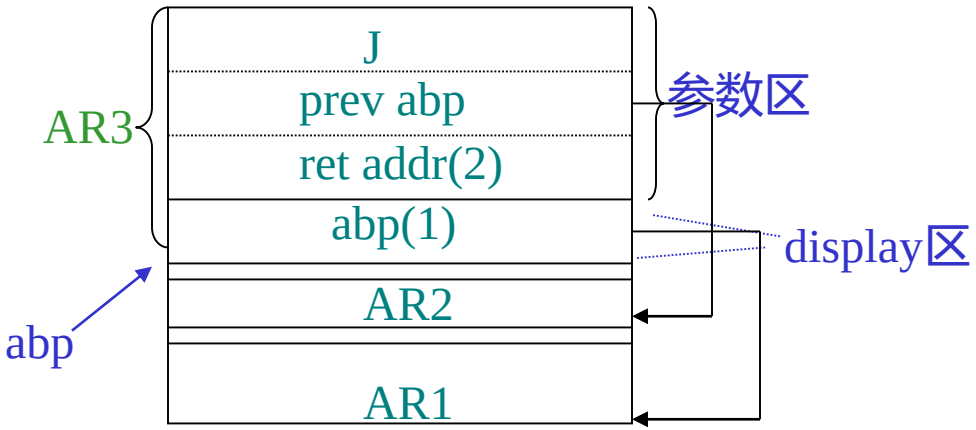


(a) 进入模块1

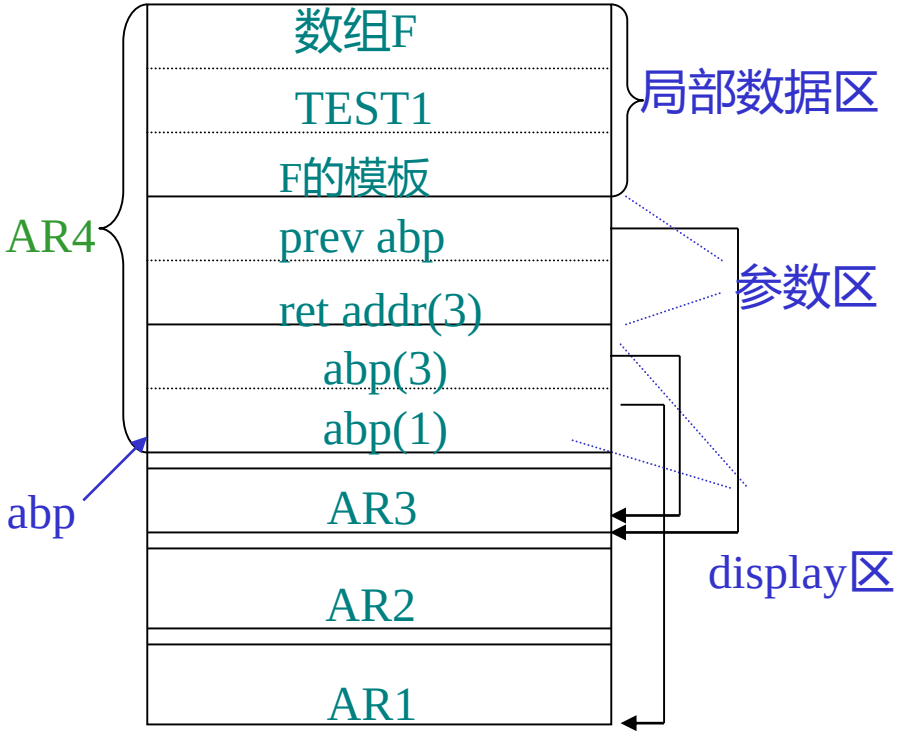


(b) M1被调用

```
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）

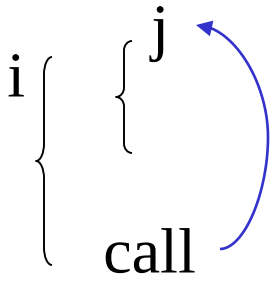
（g）当M1执行完，则abp:=prev abp，这样abp恢复到进入M1时的情况，运行栈情况如（a）

（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

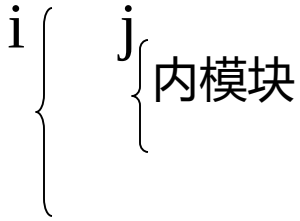
6.3.2 建造display区的规则

从i层模块进入(调用)j层模块，则：

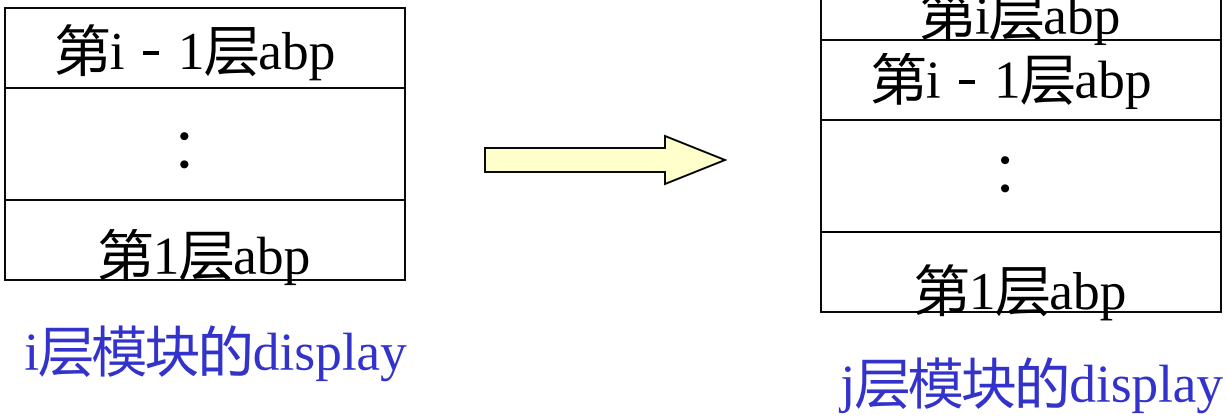
(1) 若j = i + 1



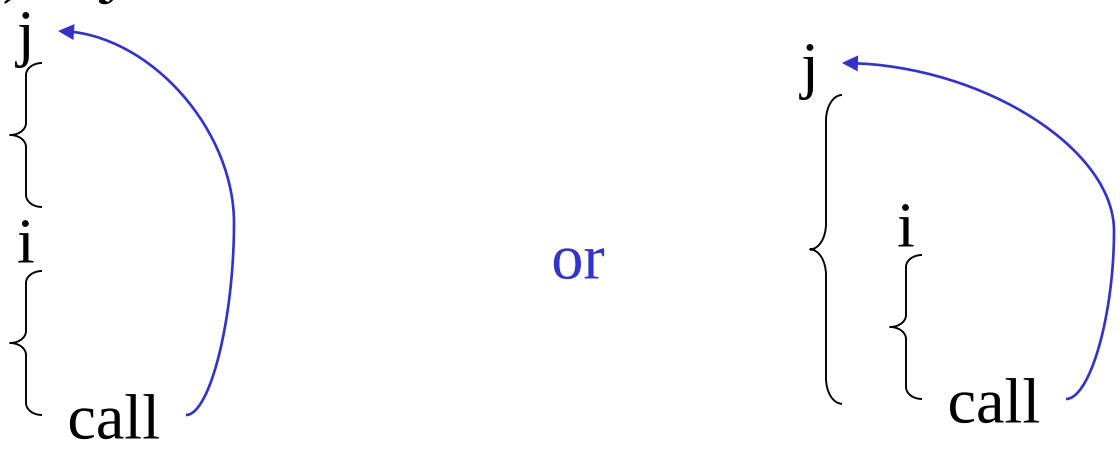
or



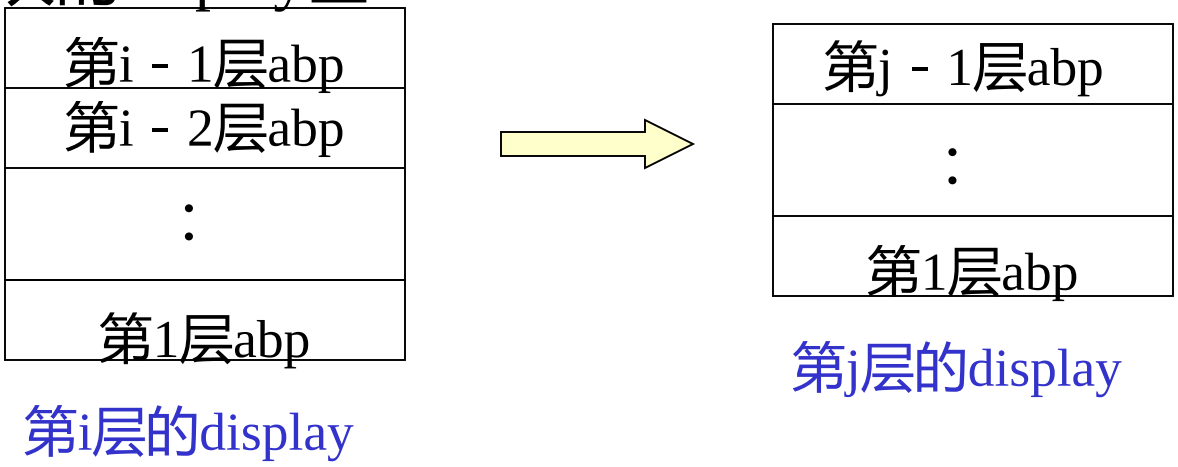
复制i层的display，然后增加一个指向i层模块记录基地址的指针



(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

```
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
```

1 X: (BL=1, ON=0)
Y: (BL=1, ON=1)
NAME: (BL=1, ON=2)

2 IND: (BL=2, ON=0)
X: (BL=2, ON=1)

3 J: (BL=2, ON=0)

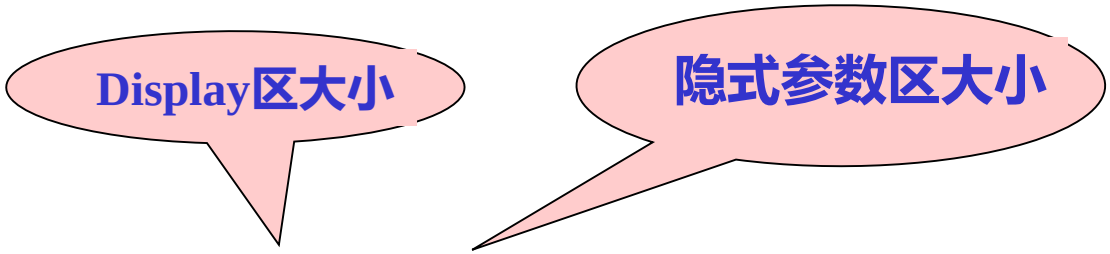
4 F: (BL=3, ON=0)
TEST1: (BL=3, ON=1)

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)

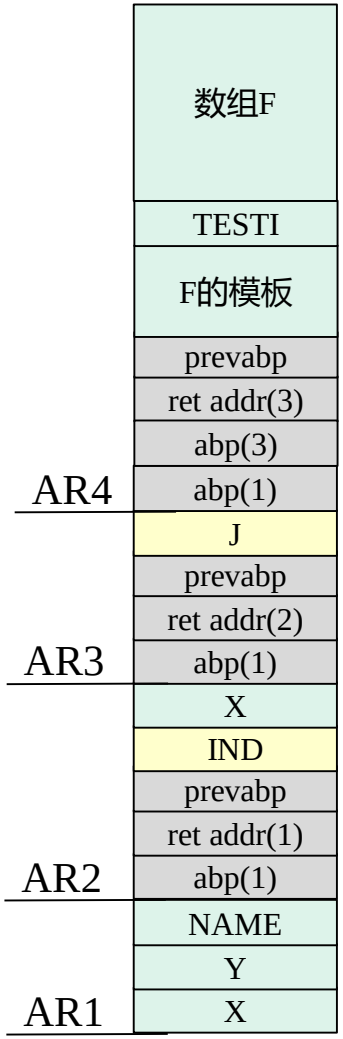
该变量在LEV层模块中引用

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")
```

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)

BL：嵌套深度　ON：变量在本层的编号

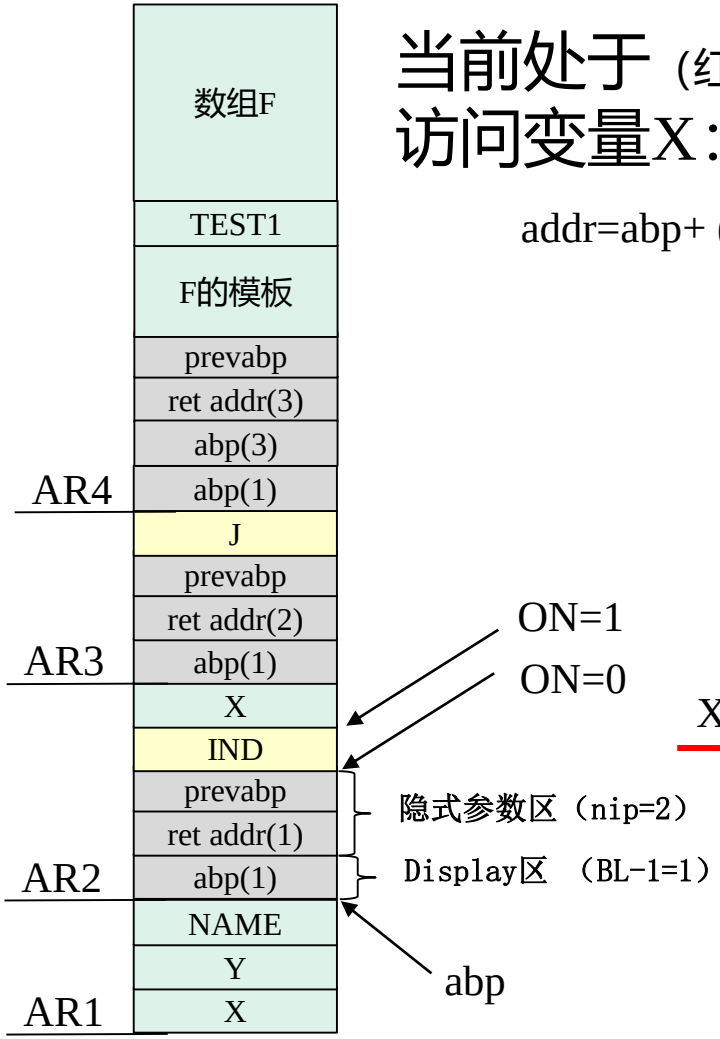
地址计算公式：

Display区大小　隐式参数区大小

```
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")
```

```
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
```

6.3.3 运行时的地址计算



当前处于（红线位置），当前嵌套深度LEV=2

访问变量X： （BL=2，ON=1)

addr=abp+ (BL-1)+nip+ON

地址计算公式:

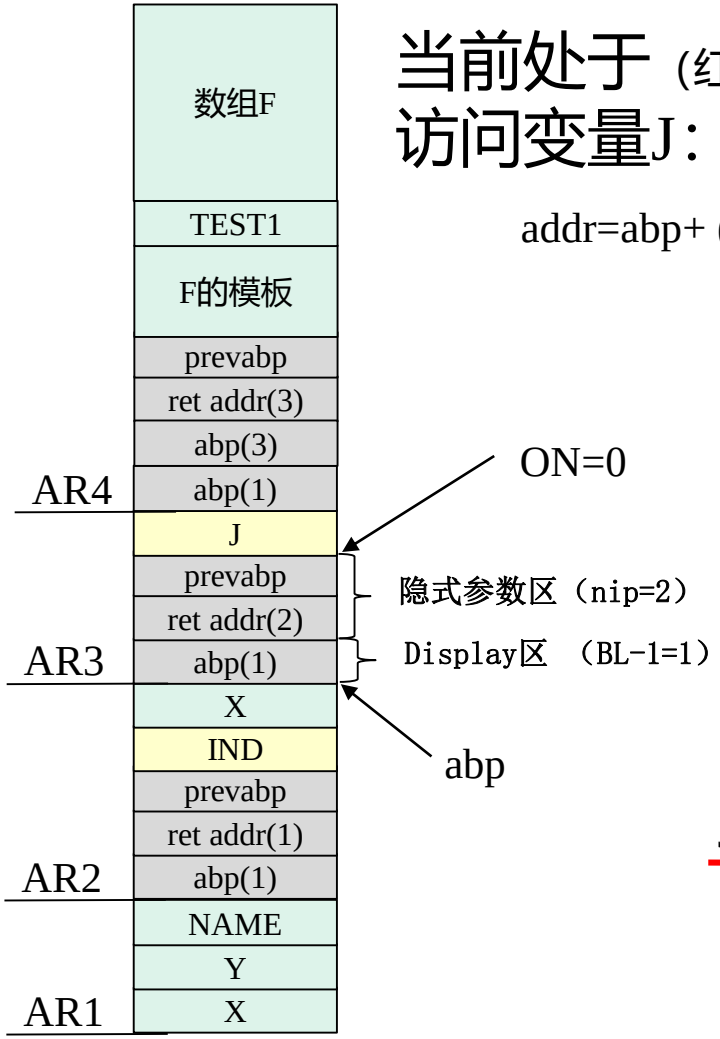
Display区大小　隐式参数区大小

```
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")
```

X: (BL=2, ON=1)

```
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
```

6.3.3 运行时的地址计算



当前处于（红线位置），当前嵌套深度LEV=3

访问变量J： （BL=2，ON=0)

addr=abp+ (BL-1)+nip+ON

地址计算公式:

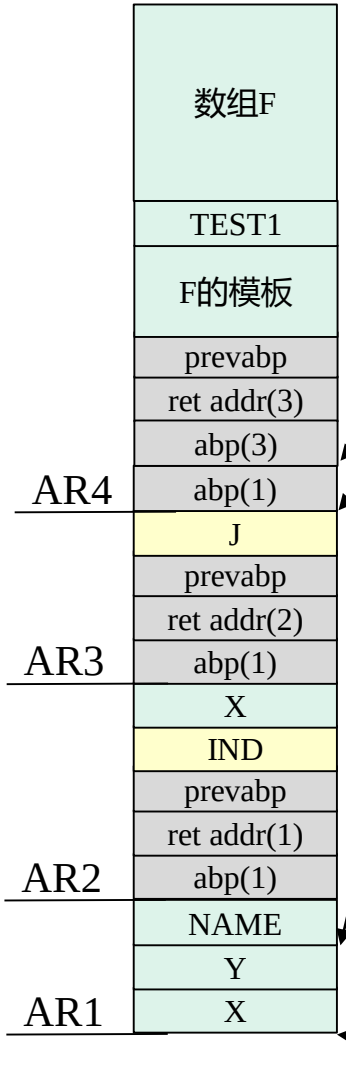
Display区大小　隐式参数区大小

```
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")
```

J: (BL=2, ON=0)

```
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
```

6.3.3 运行时的地址计算



当前处于Block 4（红线位置），当前BL=3
访问变量NAME： （BL=1，ON=2)

地址计算公式：

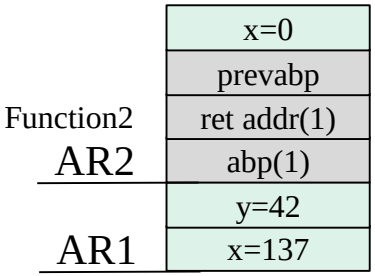
Display区大小　隐式参数区大小

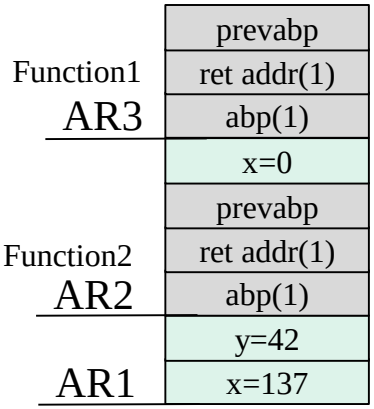
```
if BL = LEV then
        addr := abp + (BL-1) + nip + ON
else if BL < LEV then
        addr := display[BL] + (BL-1) + nip + ON
else
    write("地址错，不合法的模块层次")
```

```
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
```

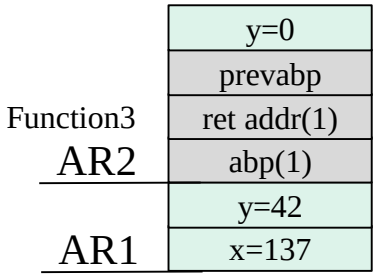
处理递归调用 (p.126)

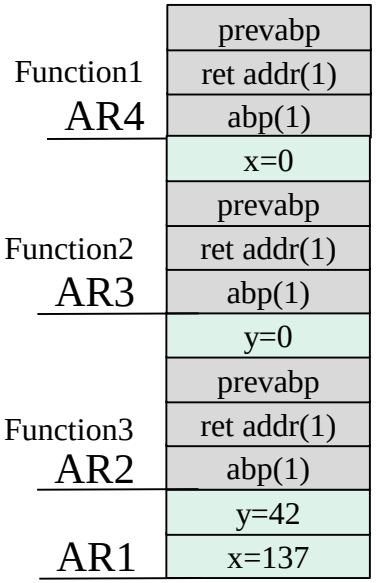
```
5   using namespace std;
6
7   int x=137;
8   int y=42;
9
10  void Function1() {
11      cout << x+y << endl;
12      return;
13  }
14
15  void Function2() {
16      int x=0;
17      Function1();
18  }
19
20  void Function3() {
21      int y=0;
22      Function2();
23  }
24
25  int main() {
26      Function1();
27      Function2();
28      Function3();
29      return 0;
30  }
```





```
5   using namespace std;
6
7   int x=137;
8   int y=42;
9
10  void Function1() {
11      cout << x+y << endl;
12      return;
13  }
14
15  void Function2() {
16      int x=0;
17      Function1();
18  }
19
20  void Function3() {
21      int y=0;
22      Function2();
23  }
24
25  int main() {
26      Function1();
27      Function2();
28      Function3();
29      return 0;
30  }
```

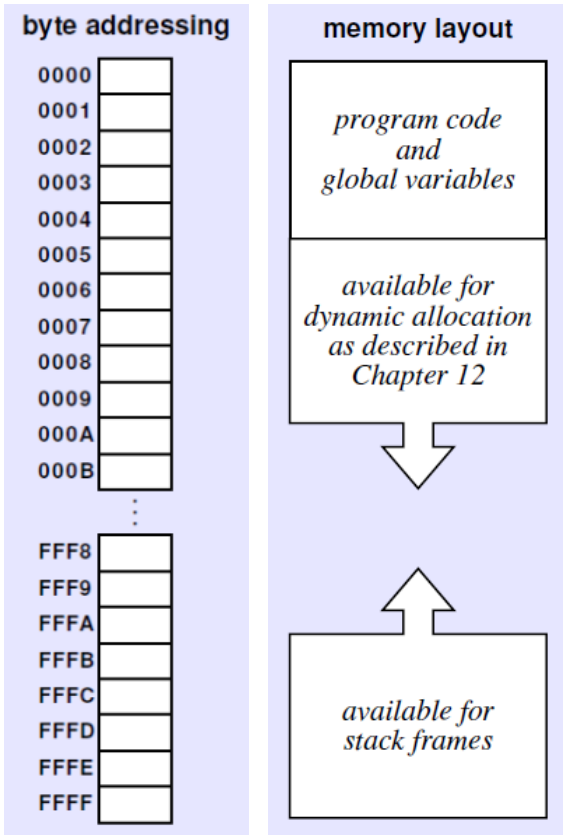


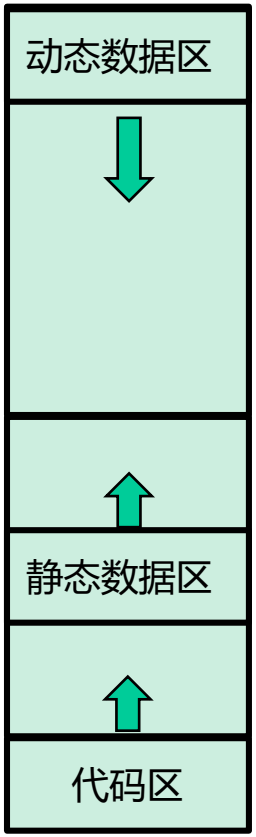


C语言的处理

例：C语言的运行时存储管理

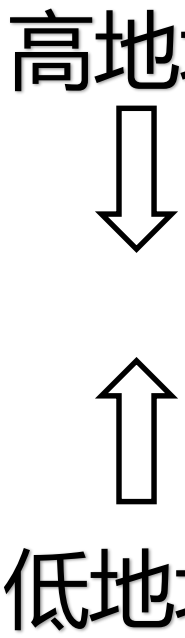
C语言不采用 Display 区记录





例：C语言的运行时存储管理

C语言不采用 Display 区记录



栈区
- 向下增长
- 保存局部变量

堆区
- 向上增长
- 保存由malloc系列函数或new操作符分配的内存

静态区
- 未初始化全局变量
- 已初始化全局变量、静态变量、常量

代码区
- 可执行代码

例：C语言的运行时存储管理

全局变量和静态变量

```
int global_c = 0 ;

void foo(int a)
{
    static int s_c = 0 ;
    s_c += a ;
    global_c = s_c ;
}
```

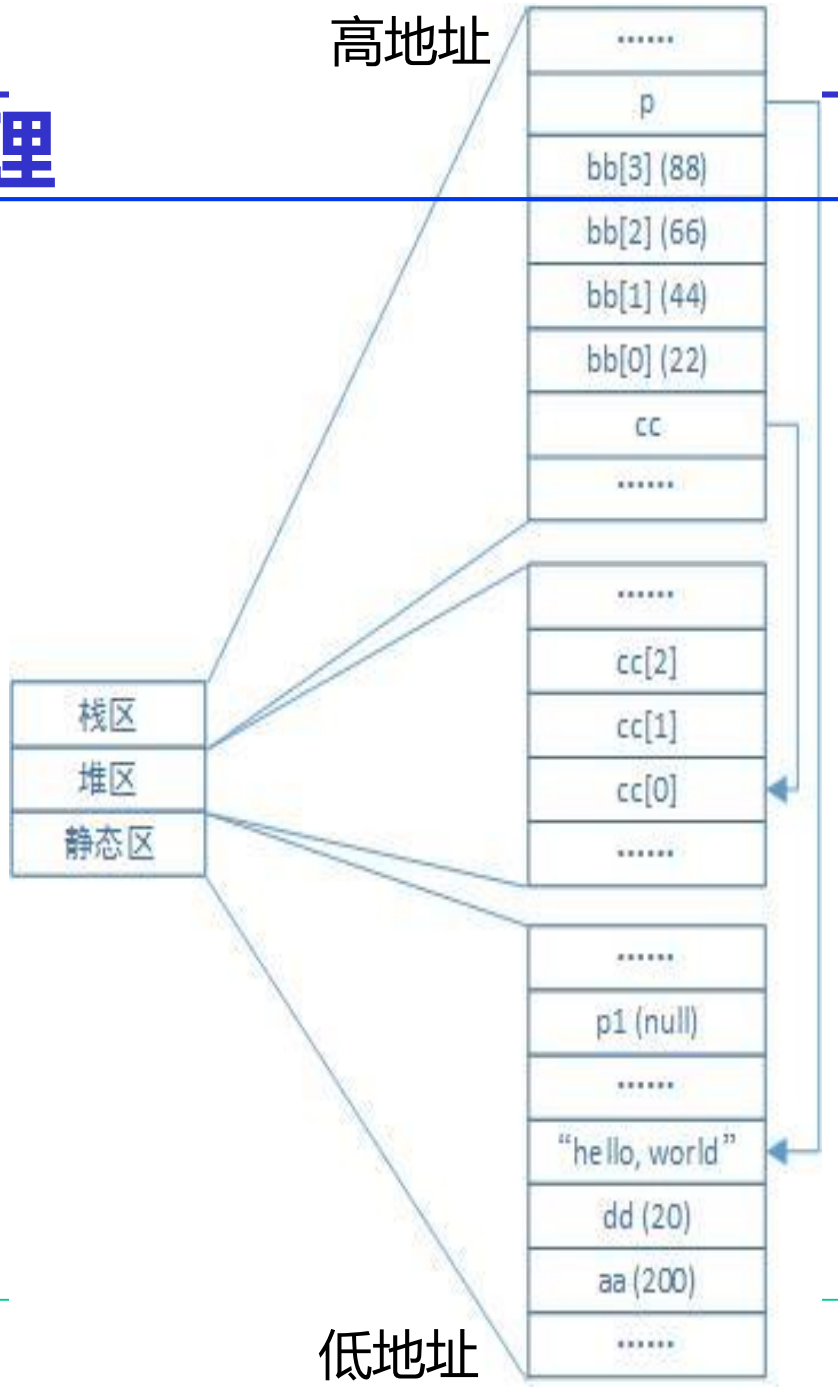
```
00427e34        global_c
00427e38        s_c
…
```

```
…
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
…
```

例：C语言的运行时存储管理

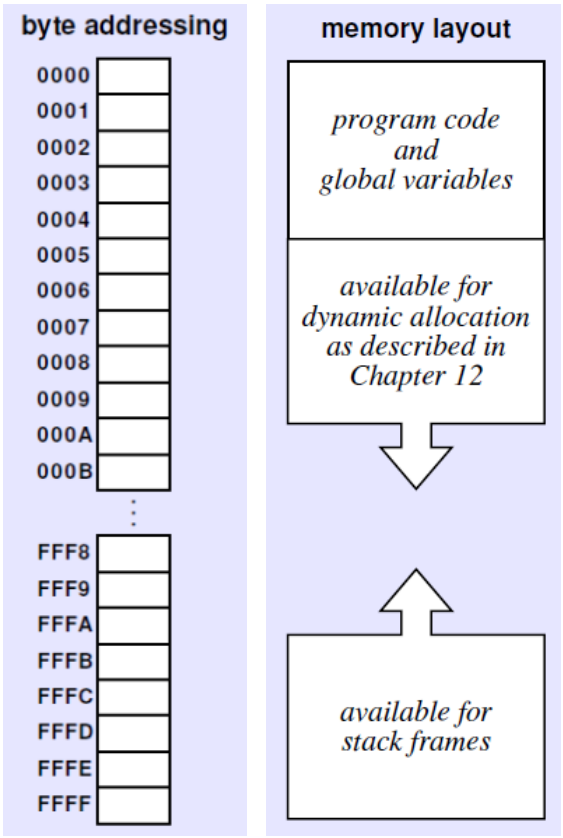
C语言不采用 Display 区记录

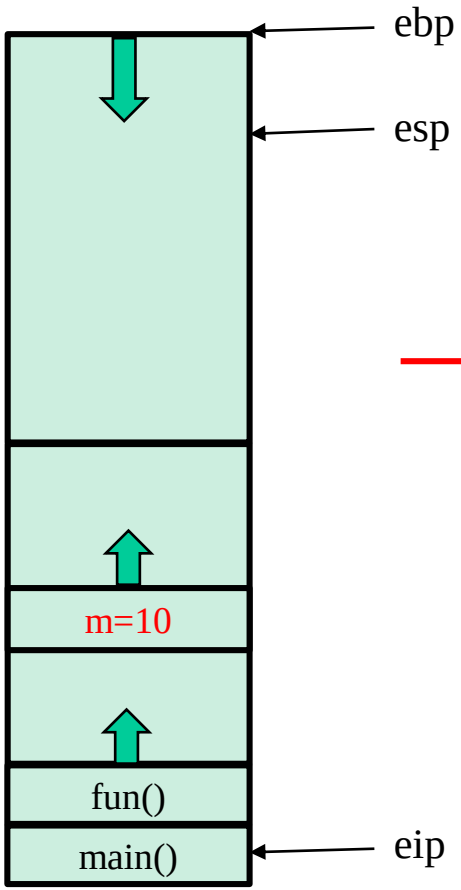
```
#include <iostream.h>
int dd=20;     //已初始化全局变量，在静态区
char *p1;      //未初始化全局变量，在静态区
int main()
{
    static int aa = 200;  //全局静态变量，在静态区
    char *p = "hello, world";  //p在栈区，"hello, world"在静态区
    int bb[] = {22, 44, 66, 88}; //bb在栈区，占四个字节
    int *cc;   //局部变量，在栈区
    cc = new int[3];  //由new动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



例：C语言的运行时存储管理

C语言不采用 Display 区记录

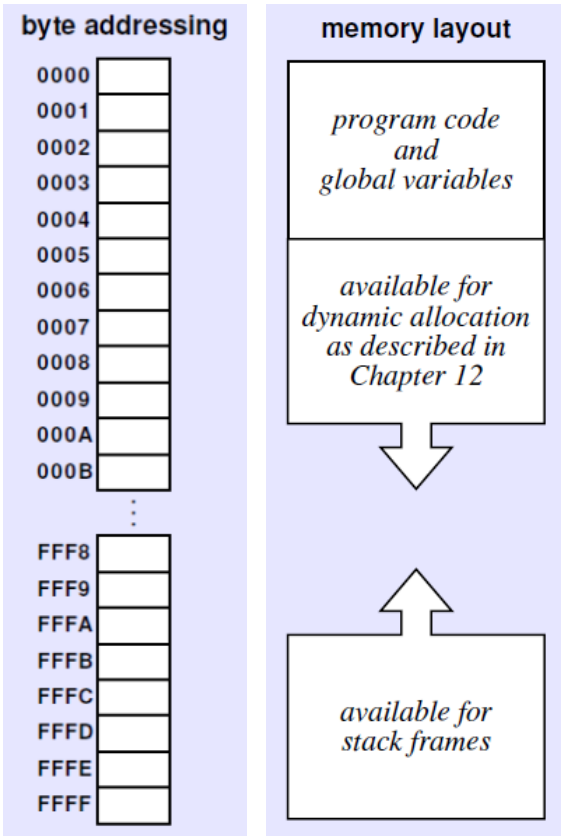


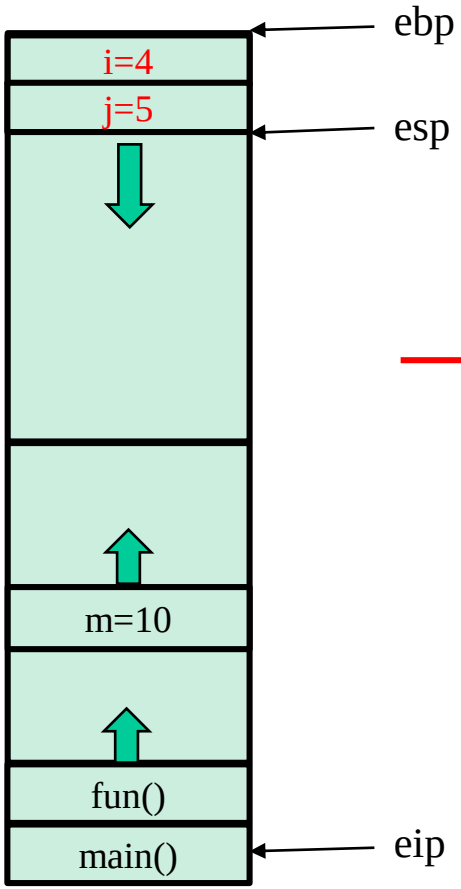



```
 6  int fun(int a, int b);
 7  int m=10;
 8  int main()
 9  {
10      int i=4;
11      int j=5;
12      m = fun(i,j);
13      return 0;
14  }
15
16  int fun(int a, int b) {
17      int c=0;
18      c=a+b;
19      return c;
20  }
```

例：C语言的运行时存储管理

C语言不采用 Display 区记录

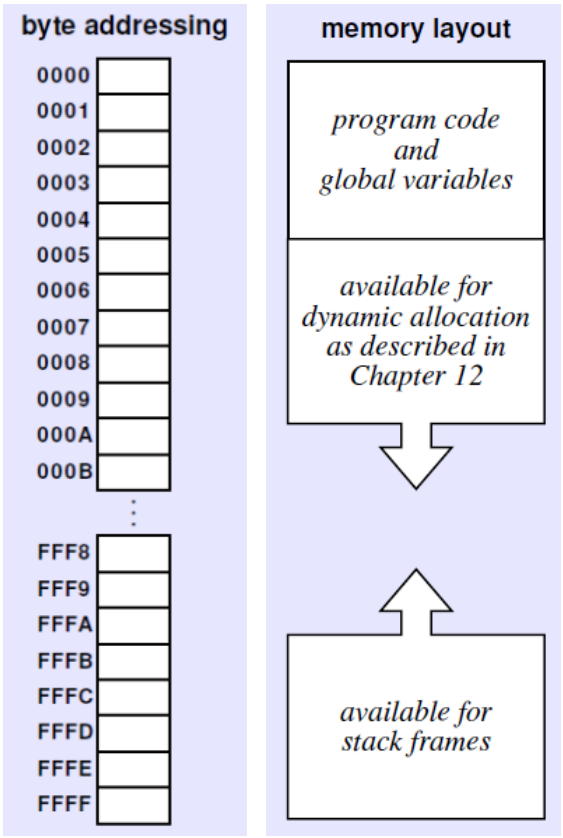


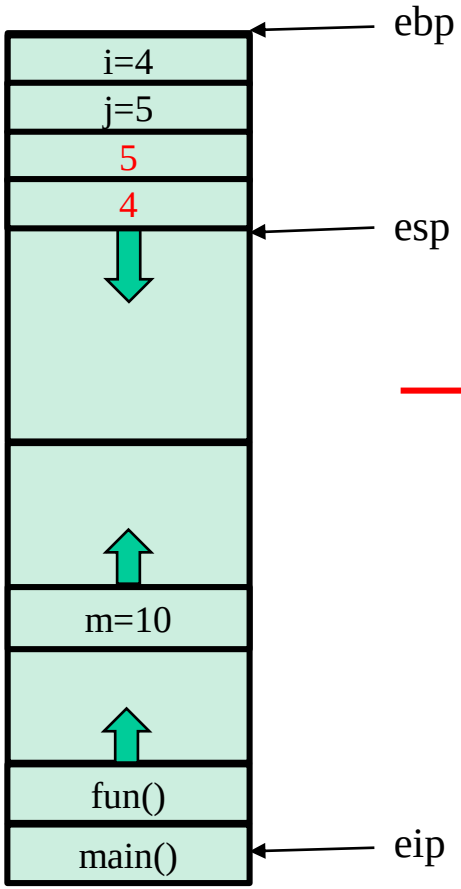



```
 6  int fun(int a, int b);
 7  int m=10;
 8  int main()
 9  {
10      int i=4;
11      int j=5;
12      m = fun(i,j);
13      return 0;
14  }
15
16  int fun(int a, int b) {
17      int c=0;
18      c=a+b;
19      return c;
20  }
```

例：C语言的运行时存储管理

C语言不采用 Display 区记录

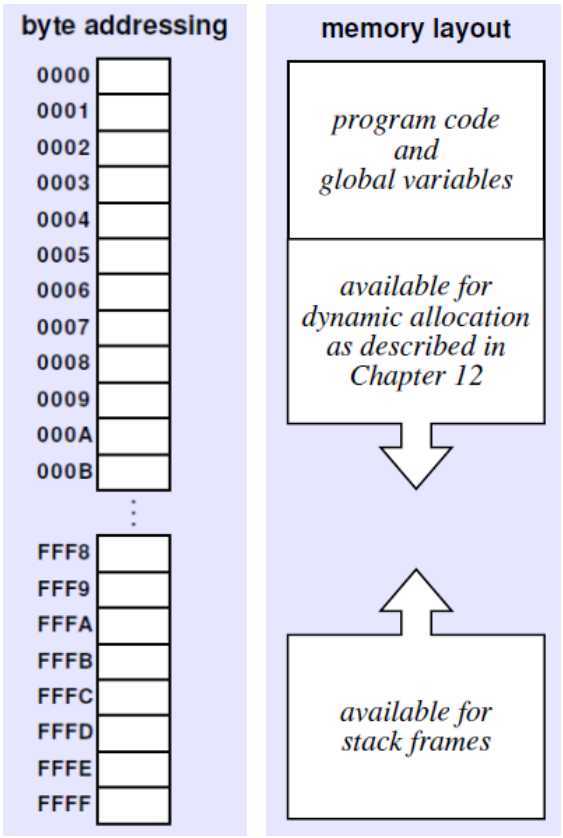


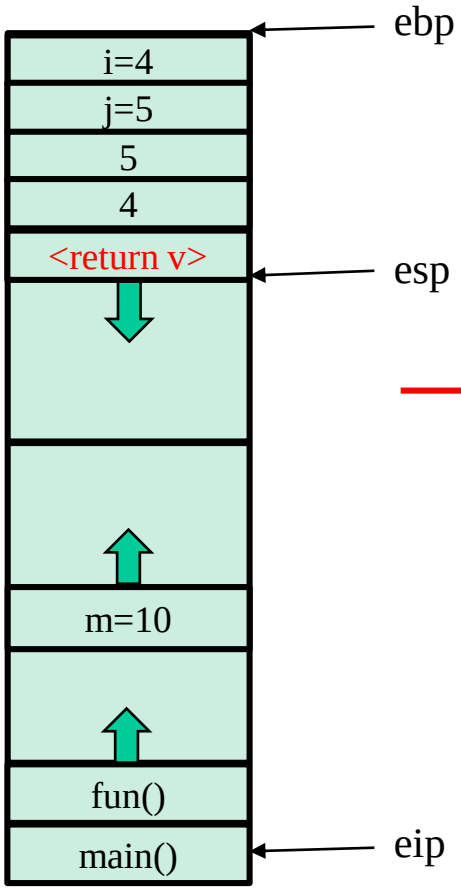



```
 6  int fun(int a, int b);
 7  int m=10;
 8  int main()
 9  {
10      int i=4;
11      int j=5;
12      m = fun(i,j);
13      return 0;
14  }
15
16  int fun(int a, int b) {
17      int c=0;
18      c=a+b;
19      return c;
20  }
```

例：C语言的运行时存储管理

C语言不采用 Display 区记录

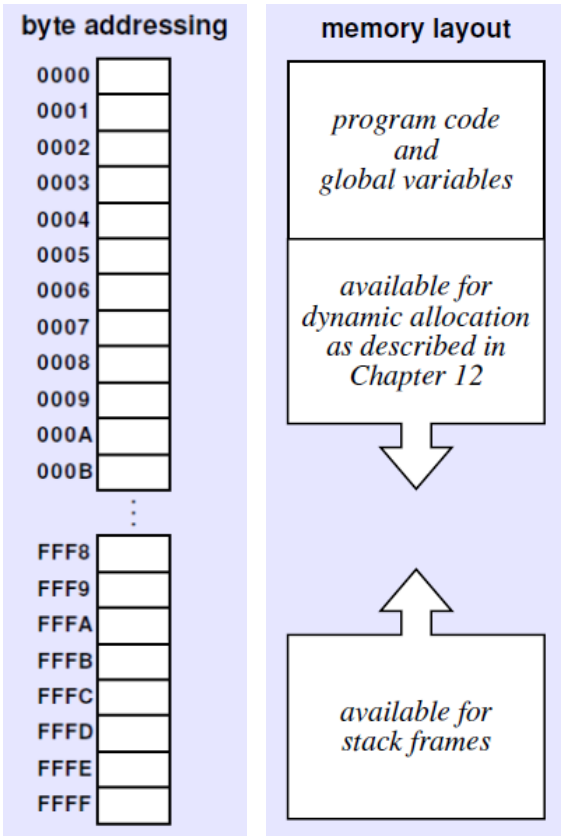


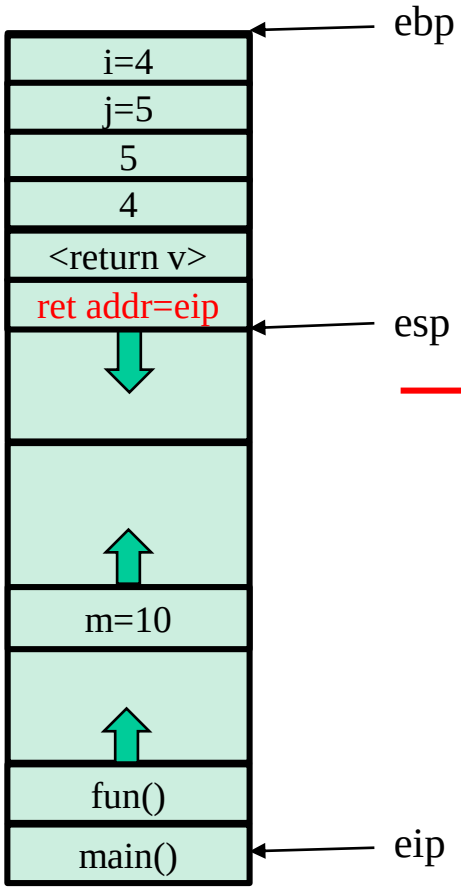



```
 6  int fun(int a, int b);
 7  int m=10;
 8  int main()
 9  {
10      int i=4;
11      int j=5;
12      m = fun(i,j);
13      return 0;
14  }
15
16  int fun(int a, int b) {
17      int c=0;
18      c=a+b;
19      return c;
20  }
```

例：C语言的运行时存储管理

C语言不采用 Display 区记录

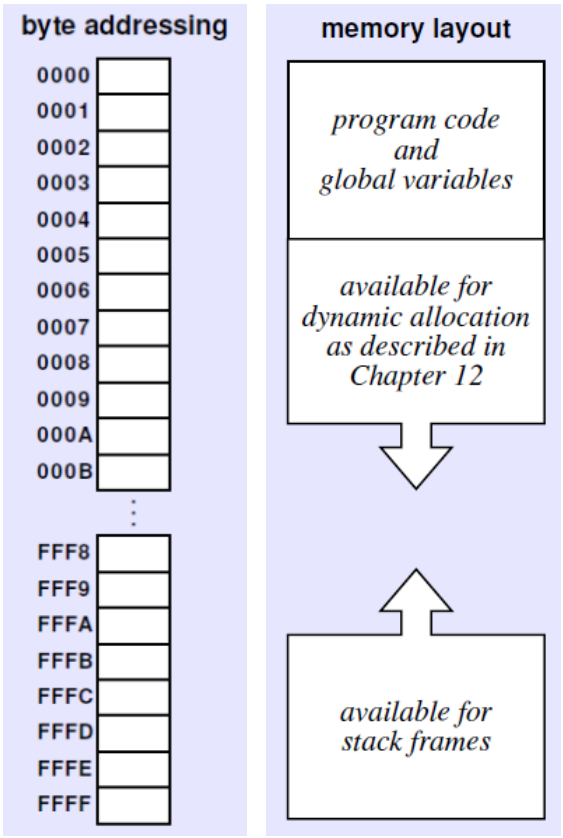


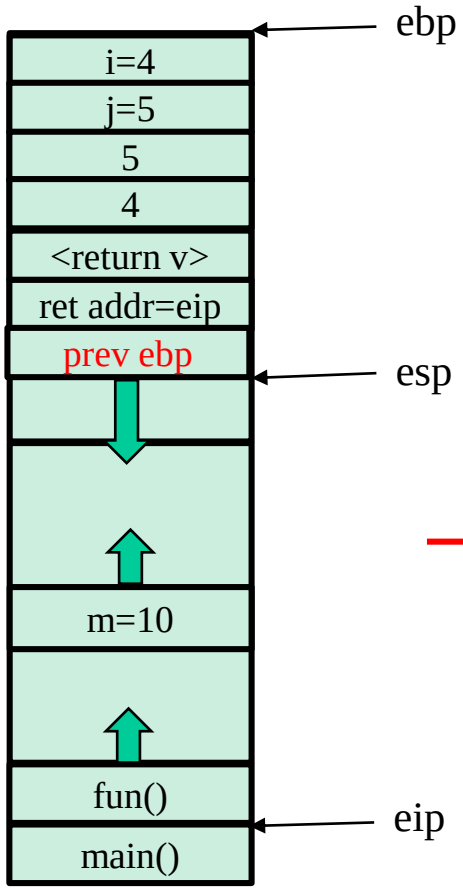



```
 6  int fun(int a, int b);
 7  int m=10;
 8  int main()
 9  {
10      int i=4;
11      int j=5;
12      m = fun(i,j);
13      return 0;
14  }
15
16  int fun(int a, int b) {
17      int c=0;
18      c=a+b;
19      return c;
20  }
```

例：C语言的运行时存储管理

C语言不采用 Display 区记录

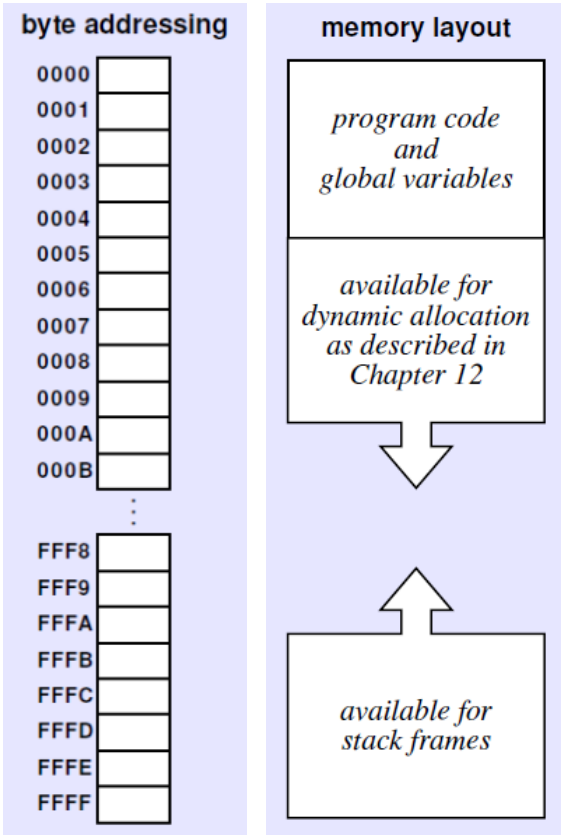


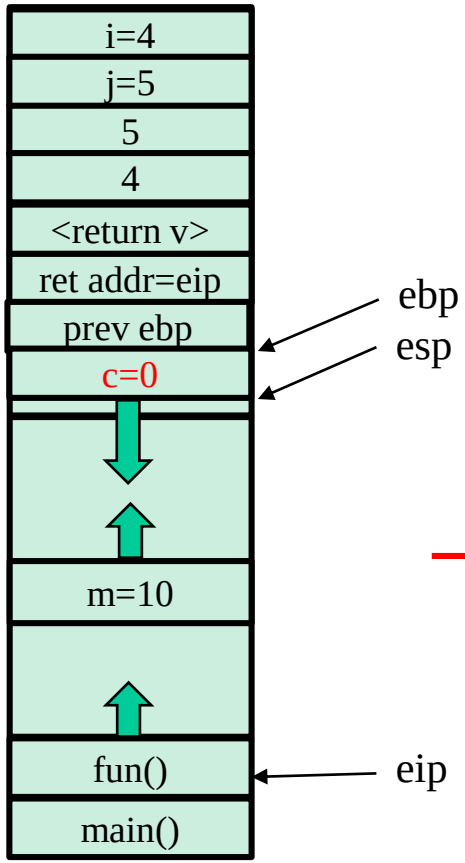



```
 6  int fun(int a, int b);
 7  int m=10;
 8  int main()
 9  {
10      int i=4;
11      int j=5;
12      m = fun(i,j);
13      return 0;
14  }
15
16  int fun(int a, int b) {
17      int c=0;
18      c=a+b;
19      return c;
20  }
```

例：C语言的运行时存储管理

C语言不采用 Display 区记录

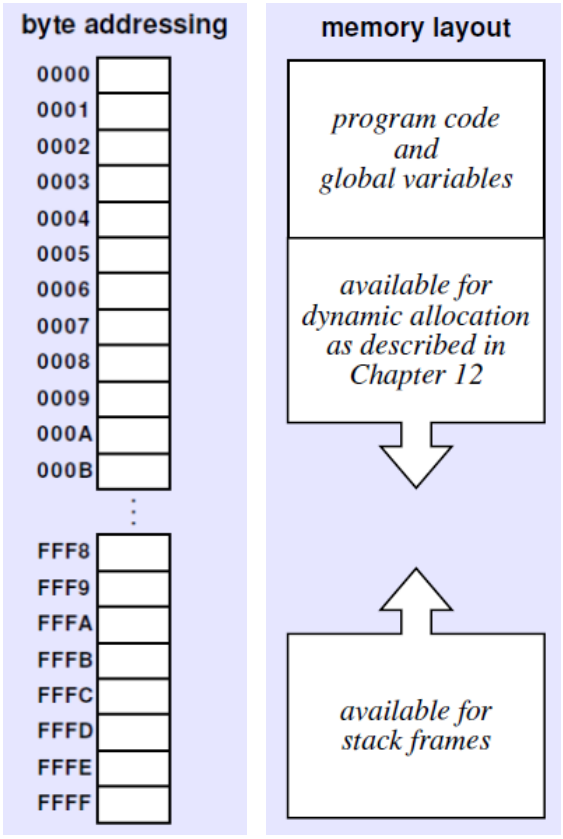


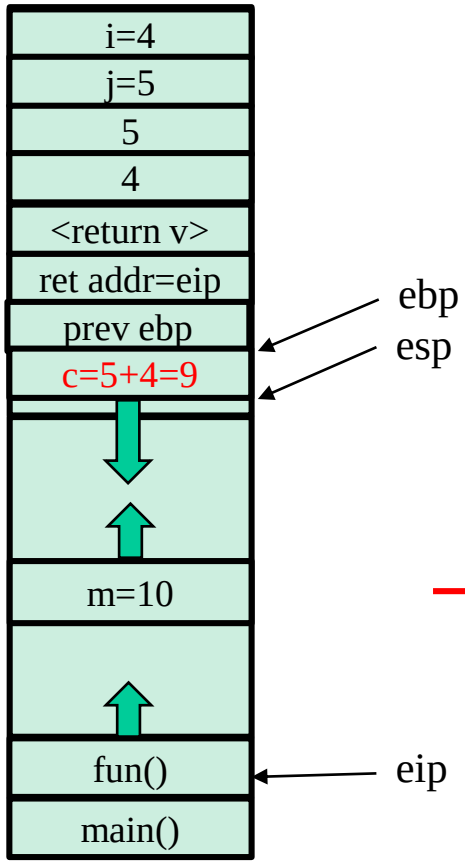



```
 6  int fun(int a, int b);
 7  int m=10;
 8  int main()
 9  {
10      int i=4;
11      int j=5;
12      m = fun(i,j);
13      return 0;
14  }
15
16  int fun(int a, int b) {
17      int c=0;
18      c=a+b;
19      return c;
20  }
```

例：C语言的运行时存储管理

C语言不采用 Display 区记录

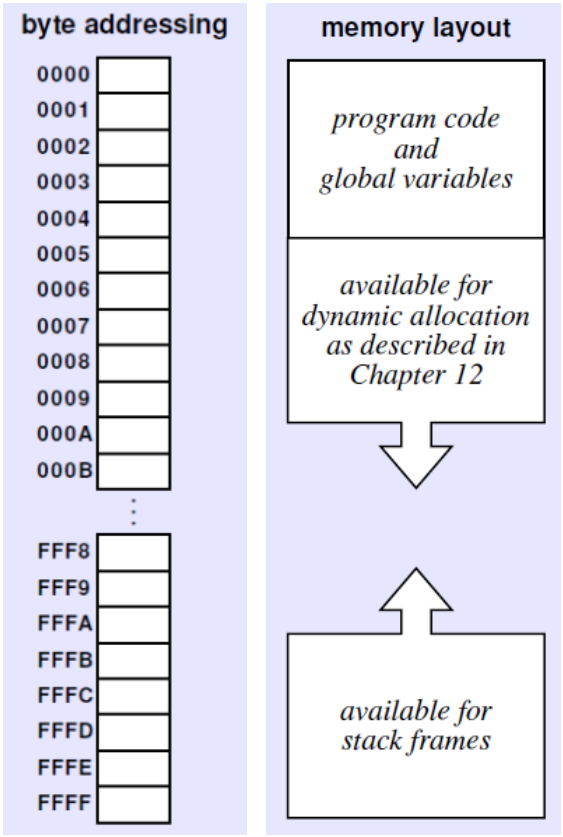


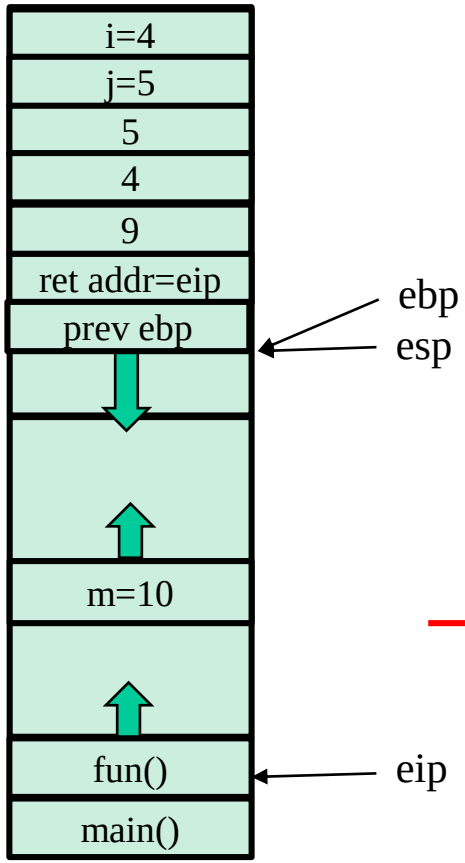



```
 6  int fun(int a, int b);
 7  int m=10;
 8  int main()
 9  {
10      int i=4;
11      int j=5;
12      m = fun(i,j);
13      return 0;
14  }
15
16  int fun(int a, int b) {
17      int c=0;
18      c=a+b;
19      return c;
20  }
```

例：C语言的运行时存储管理

C语言不采用 Display 区记录

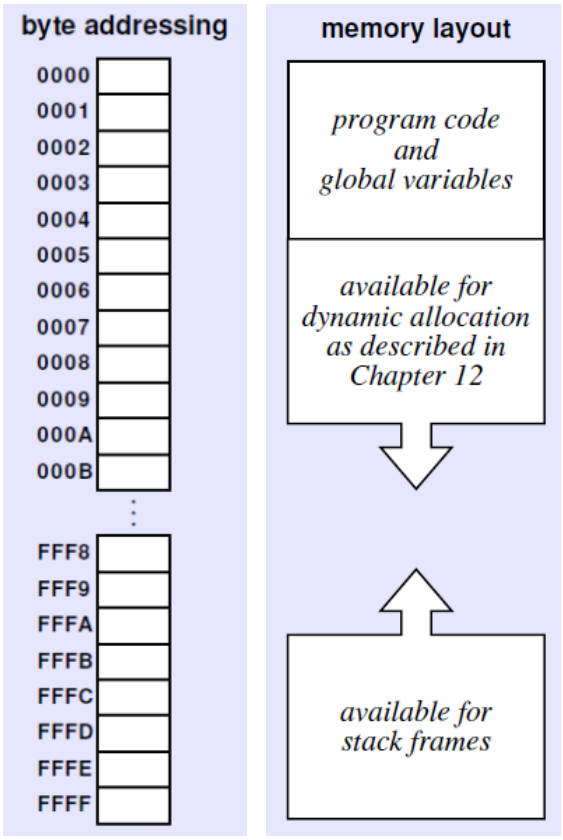


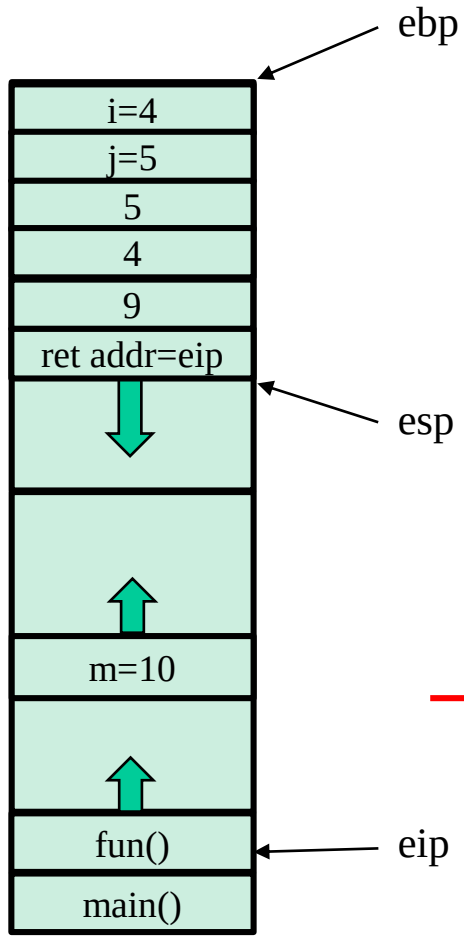



```
 6  int fun(int a, int b);
 7  int m=10;
 8  int main()
 9  {
10      int i=4;
11      int j=5;
12      m = fun(i,j);
13      return 0;
14  }
15
16  int fun(int a, int b) {
17      int c=0;
18      c=a+b;
19      return c;
20  }
```

例：C语言的运行时存储管理

C语言不采用 Display 区记录

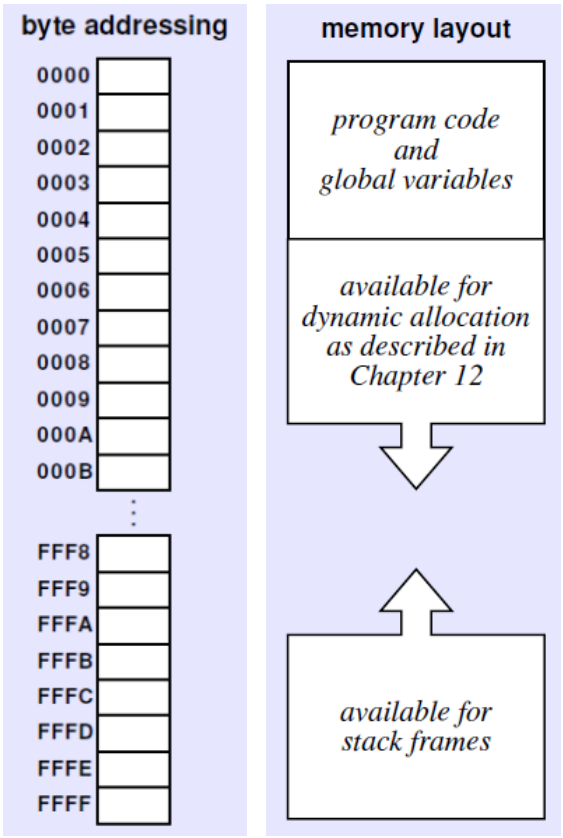


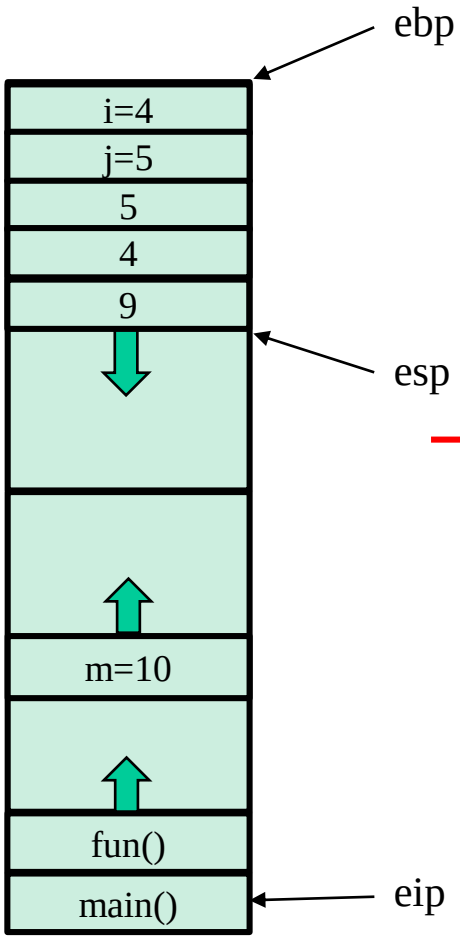



```
 6  int fun(int a, int b);
 7  int m=10;
 8  int main()
 9  {
10      int i=4;
11      int j=5;
12      m = fun(i,j);
13      return 0;
14  }
15
16  int fun(int a, int b) {
17      int c=0;
18      c=a+b;
19      return c;
20  }
```

例：C语言的运行时存储管理

C语言不采用 Display 区记录

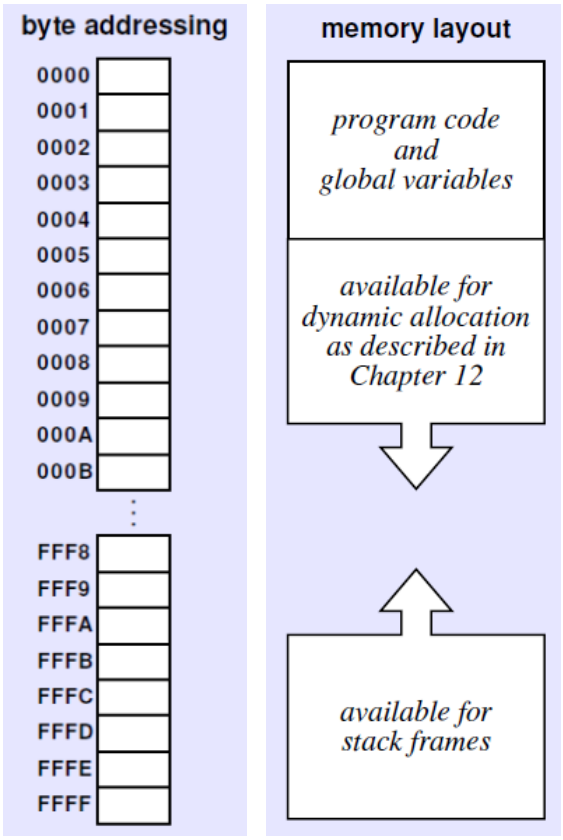


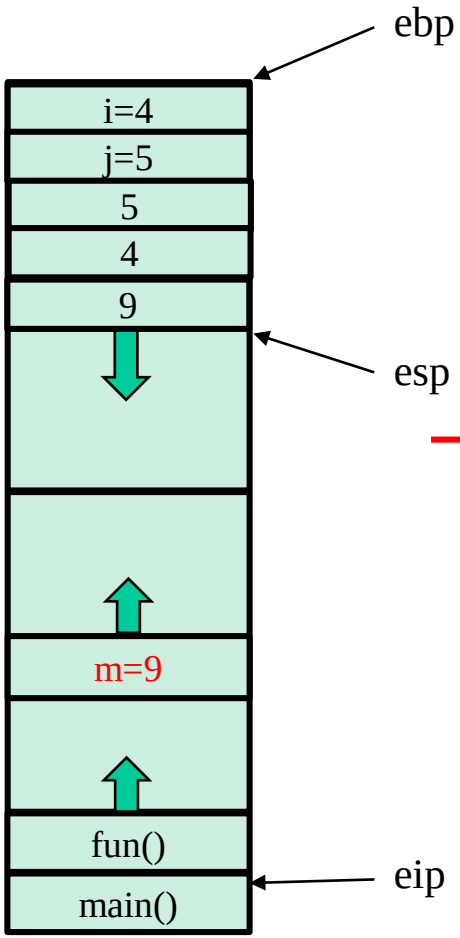



```
6   int fun(int a, int b);
7   int m=10;
8   int main()
9   {
10      int i=4;
11      int j=5;
12      m = fun(i,j);
13      return 0;
14  }
15
16  int fun(int a, int b) {
17      int c=0;
18      c=a+b;
19      return c;
20  }
```

例：C语言的运行时存储管理

C语言不采用 Display 区记录

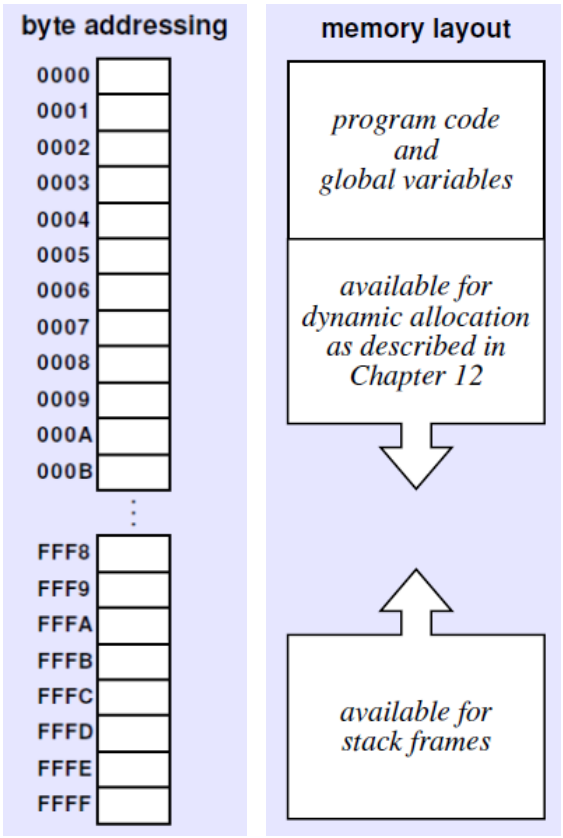


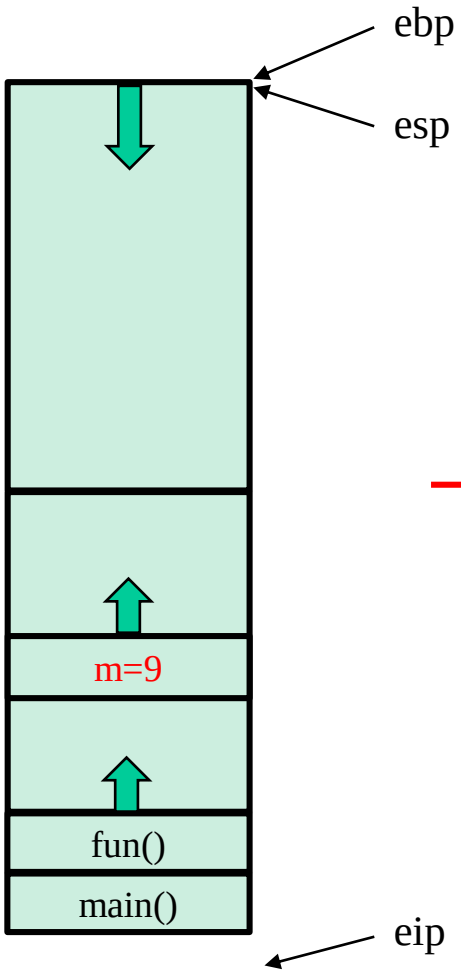



```
6   int fun(int a, int b);
7   int m=10;
8   int main()
9   {
10      int i=4;
11      int j=5;
12      m = fun(i,j);
13      return 0;
14  }
15
16  int fun(int a, int b) {
17      int c=0;
18      c=a+b;
19      return c;
20  }
```

例：C语言的运行时存储管理

C语言不采用 Display 区记录






```
 6  int fun(int a, int b);
 7  int m=10;
 8  int main()
 9  {
10      int i=4;
11      int j=5;
12      m = fun(i,j);
13      return 0;
14  }
15
16  int fun(int a, int b) {
17      int c=0;
18      c=a+b;
19      return c;
20  }
```

垃圾回收

例：C语言的运行时存储管理

堆和栈的分配区别

栈	堆
解决了函数递归调用等问题	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

例：C语言的运行时存储管理

堆分配引入的新问题——垃圾回收

垃圾 = 不再使用的内存对象

垃圾回收是一种自动内存管理机制，可以对动态分配的内存空间进行自动回收。

垃圾回收是运行时的一部分，但可以从编译器中收集信息。

```
int main() {
    Object x, y;
    x = new Object();
    y = new Object();
    /* Point A */

    x.doSomething();
    y.doSomething();
    /* Point B */

    y = new Object();
    /* Point C */
}
```

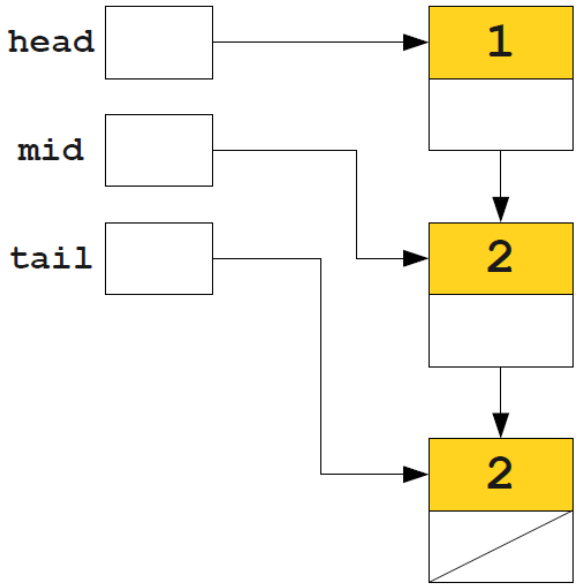
Source: Stanford CS143 (2012)

例：C语言的运行时存储管理

基本方法：引用计数（Ref Count）

记录指向内存对象的指针个数，是一种“增量方法”

```
class LinkedList {
    LinkedList next;
}

int main() {
    LinkedList head = new LinkedList;
    LinkedList mid = new LinkedList;
    LinkedList tail = new LinkedList;

    head.next = mid;
    mid.next = tail;

    mid = tail = null;

    head.next.next = null;

    head = null;
}
```



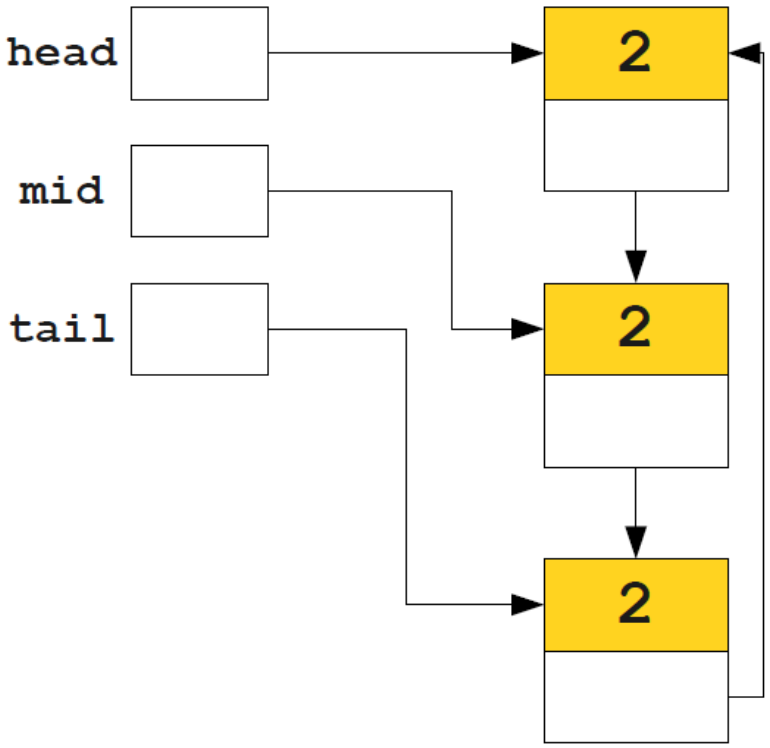
Source: Stanford CS143 (2012)

例：C语言的运行时存储管理

基本方法：引用计数（Ref Count）

```
class LinkedList {
    LinkedList next;
}

int main() {
    LinkedList head = new LinkedList;
    LinkedList mid = new LinkedList;
    LinkedList tail = new LinkedList;

    head.next = mid;
    mid.next = tail;
    tail.next = head;

    head = null;
    mid = null;
    tail = null;
}
```

循环引用



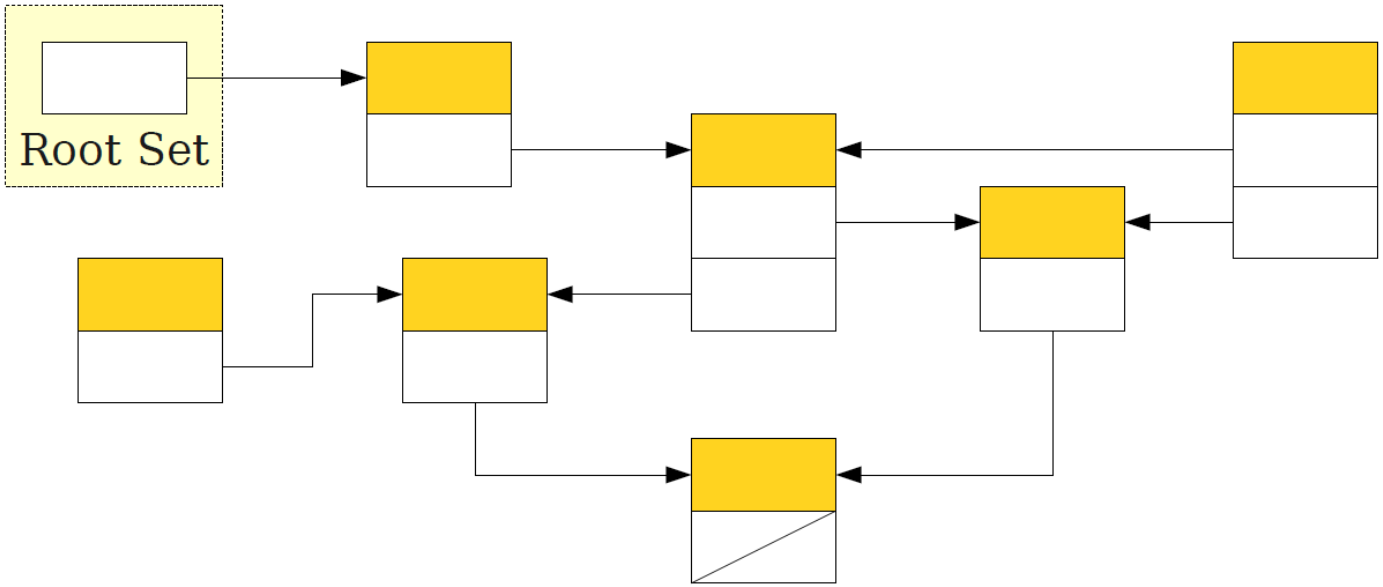
Source: Stanford CS143 (2012)

例：C语言的运行时存储管理

基本方法2：标记和清除（Mark & Sweep）

基本思想：区分可达对象和不可达对象

从一组对象（Root Set）出发遍历对象引用



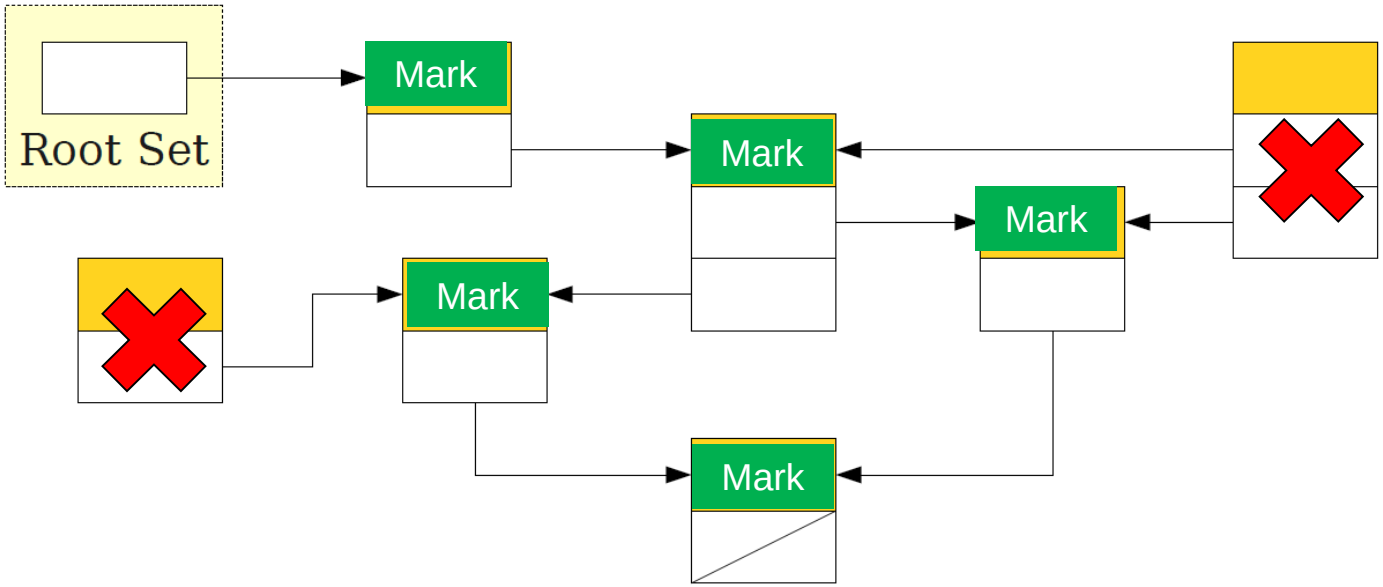
Source: Stanford CS143 (2012)

例：C语言的运行时存储管理

基本方法2：标记和清除（Mark & Sweep）

基本思想：区分可达对象和不可达对象

从一组对象（Root Set）出发遍历对象引用



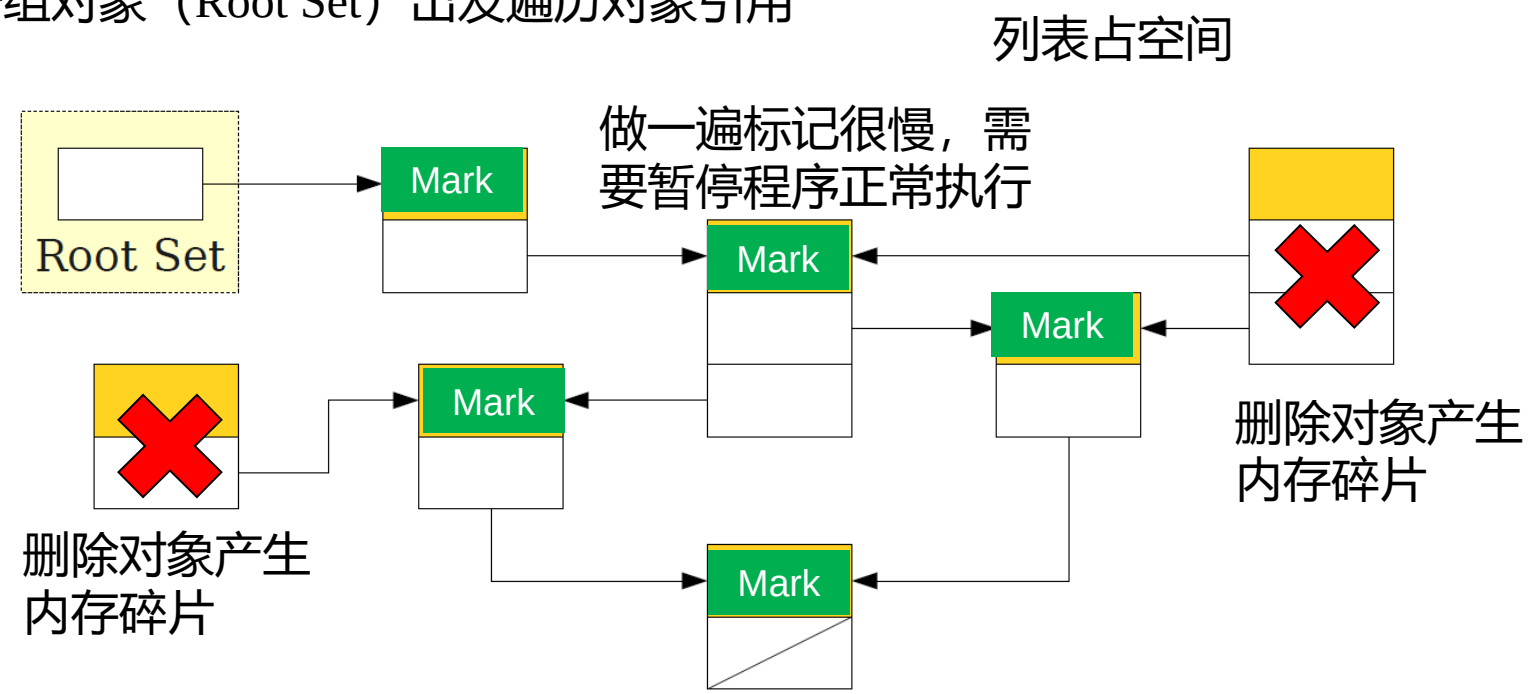
Source: Stanford CS143 (2012)

例：C语言的运行时存储管理

基本方法2：标记和清除（Mark & Sweep）

基本思想：区分可达对象和不可达对象

从一组对象（Root Set）出发遍历对象引用



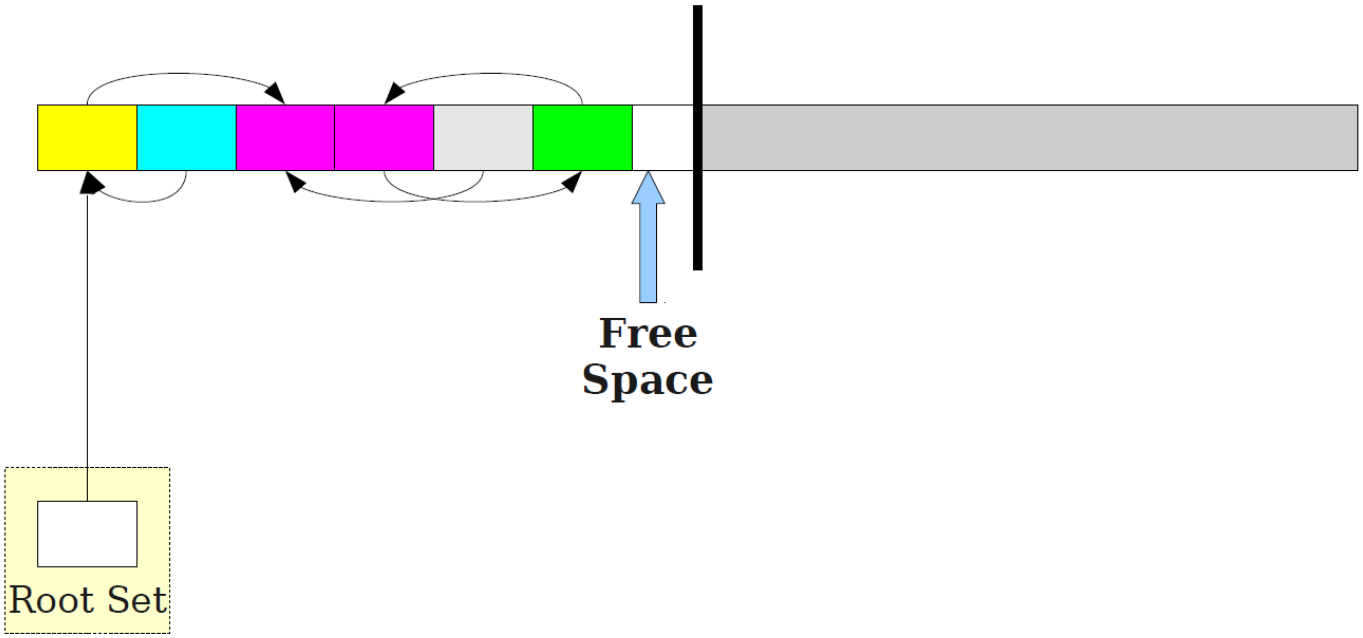
Source: Stanford CS143 (2012)

例：C语言的运行时存储管理

基本方法3：拷贝回收（Stop & Copy）

基本思想：解决空洞问题，用拷贝的办法来清除碎片（压缩：compaction）

同时对应的更新指针



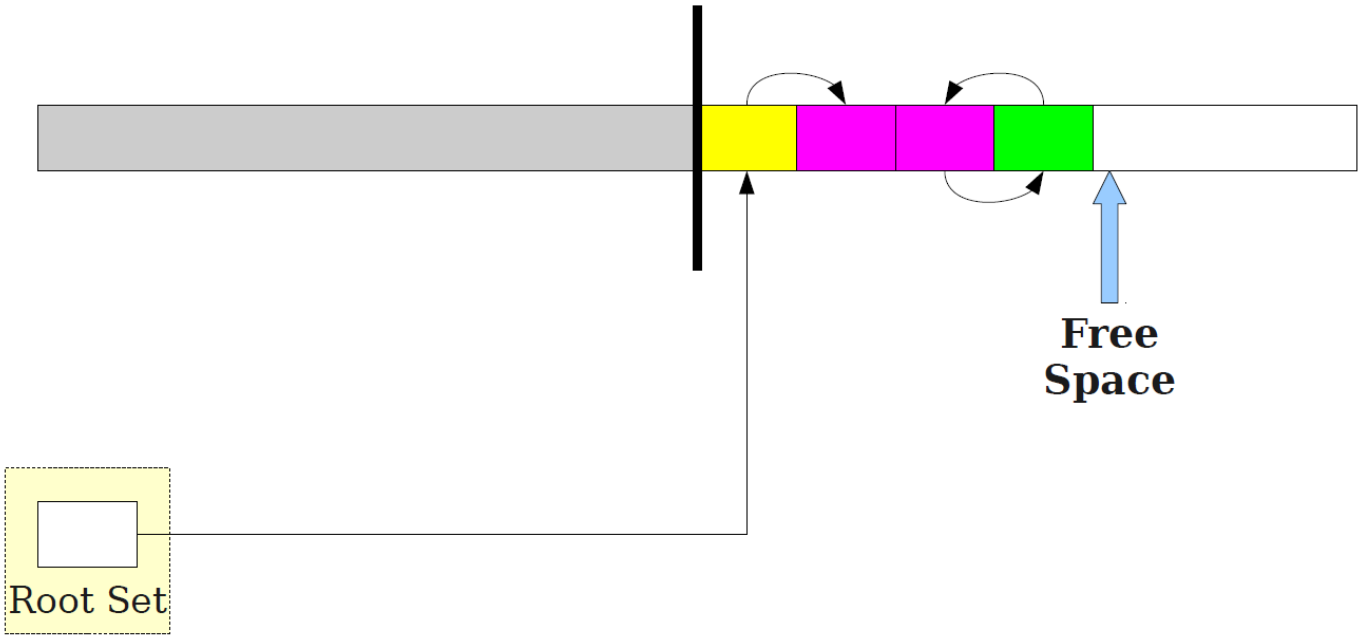
Source: Stanford CS143 (2012)

例：C语言的运行时存储管理

基本方法3：拷贝回收（Stop & Copy）

基本思想：解决空洞问题，用拷贝的办法来清除碎片（压缩：compaction）

同时对应的更新指针



Source: Stanford CS143 (2012)

例：C语言的运行时存储管理

基本方法4：混合策略（Hybrid）：分代垃圾回收

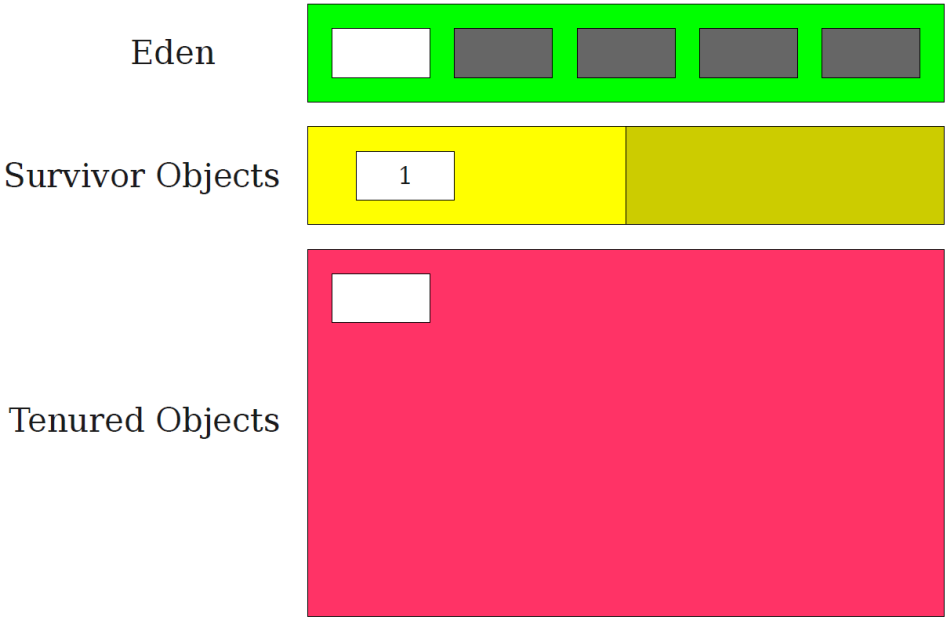
基本思想：综合利用前面的各种策略

利用局部性原理

短期对象：Stop & Copy

中期对象：Stop & Copy

中期对象：Mark & Sweap



Source: Stanford CS143 (2012)

小结：“运行时存储组织与管理”要回答

- 数据结构在内存里是如何表示的
- 函数在内存中是如何表示的
- 他们在内存中如何放置，如何管理

作业：p119 1
p133 2,3

谢谢！