

■ 问题的提出

源程序最终需要运行。因此必须了解：与源程序等价的目标程序如何在内存中运行，为了程序的正确运行需要什么样的支持。

函数

不同的源语言结构，所需的运行环境和支持不同。本章仅以最简单的、基于过程的、顺序执行的程序为前提讨论，即源程序的基本结构是顺序执行的过程，过程与过程之间仅通过子程序调用的方式进行控制流的转移。

■ 过程、活动、生存期

过程的每一次运行称为一次活动(activation)。
活动是一个动态的概念，它有有限的生存期(life time)。

活动的生存期是指从进入活动的第一条指令执行到离开此活动前的最后一条指令执行的这段时间，其中包括调用其它过程时其它活动的生存期。

运行时为名字**X**分配存储空间**S**，这一过程称为**绑定 (binding)**。

X是一个对象：

- * 既可以是数据对象，如变量，与之结合的是一个存储单元；
- * 也可以是操作对象，如过程。与之结合的是可执行的代码。

讨论仅限于X是一个数据对象。

■ 静态与动态

名字的声明与名字的绑定均需要有对应的存储空间，而存储空间的对应方式，一个是静态的，一个是动态的。

声明时关心的是声明的**作用域**，即当一个名字被引用时，在不同的作用域中与该名字的不同声明结合；

绑定时关心的是绑定的**生存期**，即当一个名字在运行时被实际分配的存储单元，名字与存储单元结合的这段时间被称为绑定的生存期，显然此生存期应该和名字的生存期一致。

■ 静态与动态(续)

- **静态：** 如果一个名字的性质通过说明语句或隐或显规则而定义，则称这种性质是“静态”确定的。
- **动态：** 如果名字的性质只有在程序运行时才能知道，则称这种性质为“动态”确定的。

■ 静态与动态(续)

静 态

过程的定义

名字的声明

声明的作用域

符号表

动 态

过程的活动

名字的绑定

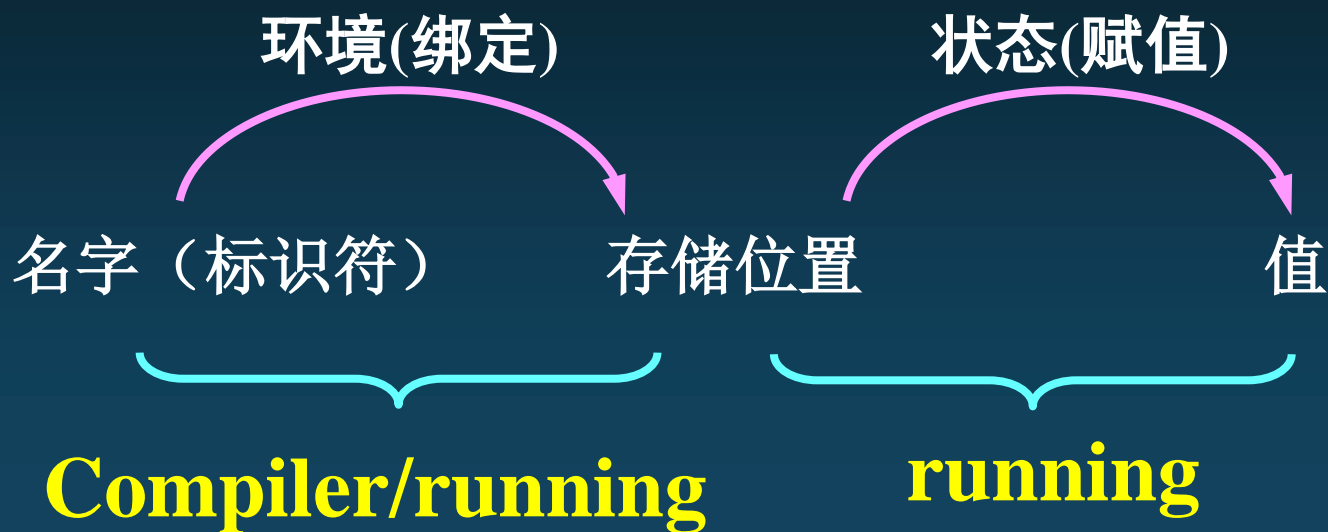
绑定的生存期

活动记录

■ 运行环境

指目标计算机的寄存器及存储器结构，用来管理存储器并保存执行过程所需的信息。

■ 变量与值的两步映射

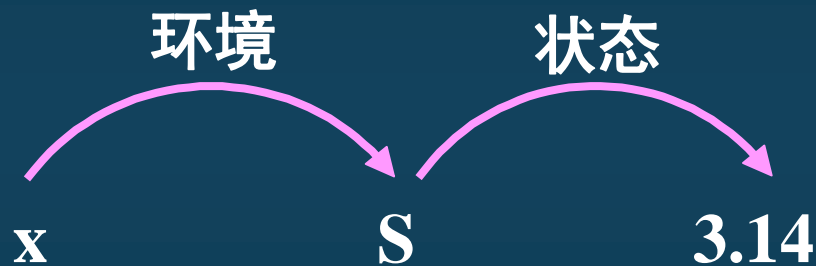


变量名字的映射

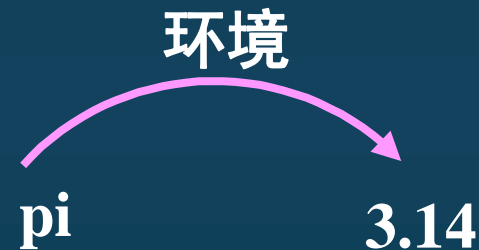


常量名字的映射

例7.1 若有变量声明 **float x ;**
常量声明 **const pi=3.14;**
则赋值句中变量和常量的映射关系:



x= 3.14的映射



pi=3.14的映射

常量没有左值（存储空间），所以不能被赋值。

🔥 注意:

* 环境、状态的区别:

赋值改变状态，但不改变环境。

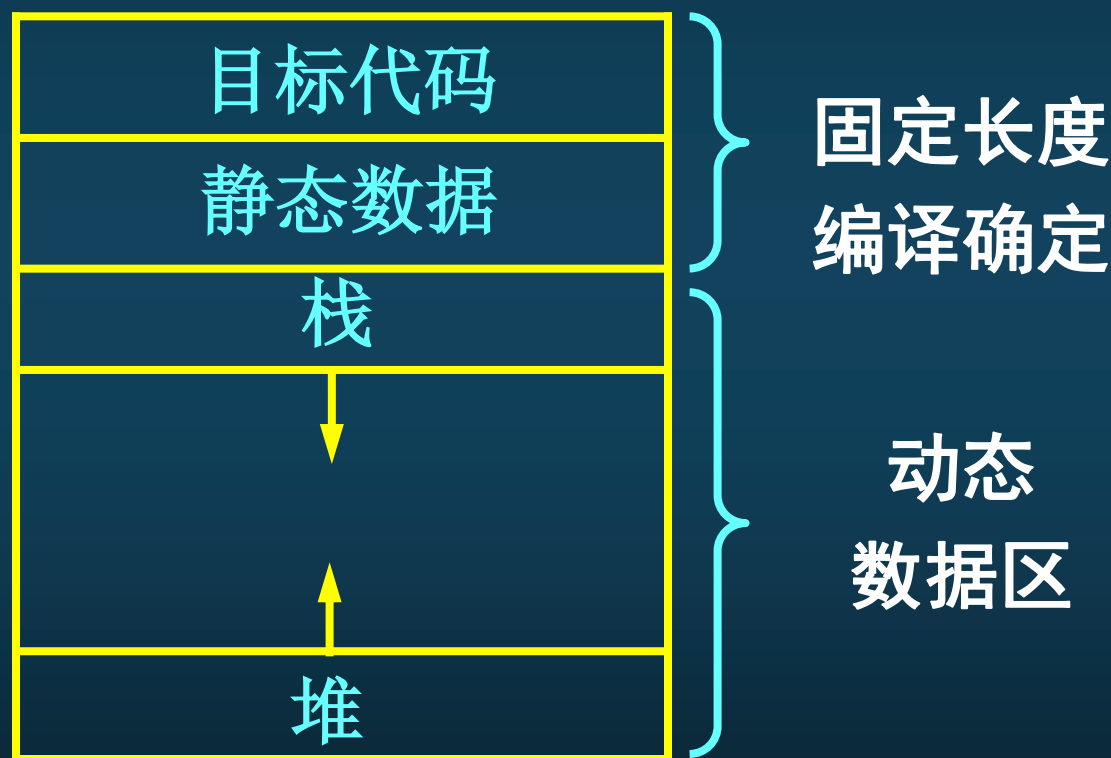
* 环境与语言特性相关:

FORTRAN完全静态环境;

C, C++, PASCAL, Ada基于栈的环境;

LISP完全动态环境。

运行时, 系统为目标程序分配的存储空间
按用途可划分为下面几个部分:



存储分配策略

静态分配策略

动态分配策略

栈式方案

堆式方案

■ 静态存储分配

在编译时能够确定目标程序运行时所需的全部数据空间的大小，即在编译时就可以将程序中的名字关联到存储单元，确定其存储位置，这种分配策略称为静态存储分配。

■ 动态存储分配

在编译时不能确定目标程序运行时所需的全部数据空间的大小，而是在目标程序运行时动态确定的。

■ 栈式动态存储分配

在内存中开辟一个栈区，按栈的特性进行存储分配。程序运行时，每当进入一个函数或过程，该函数所需的存储空间动态地分配于栈顶，函数返回时，释放所占用的空间。

■ 堆式动态存储分配

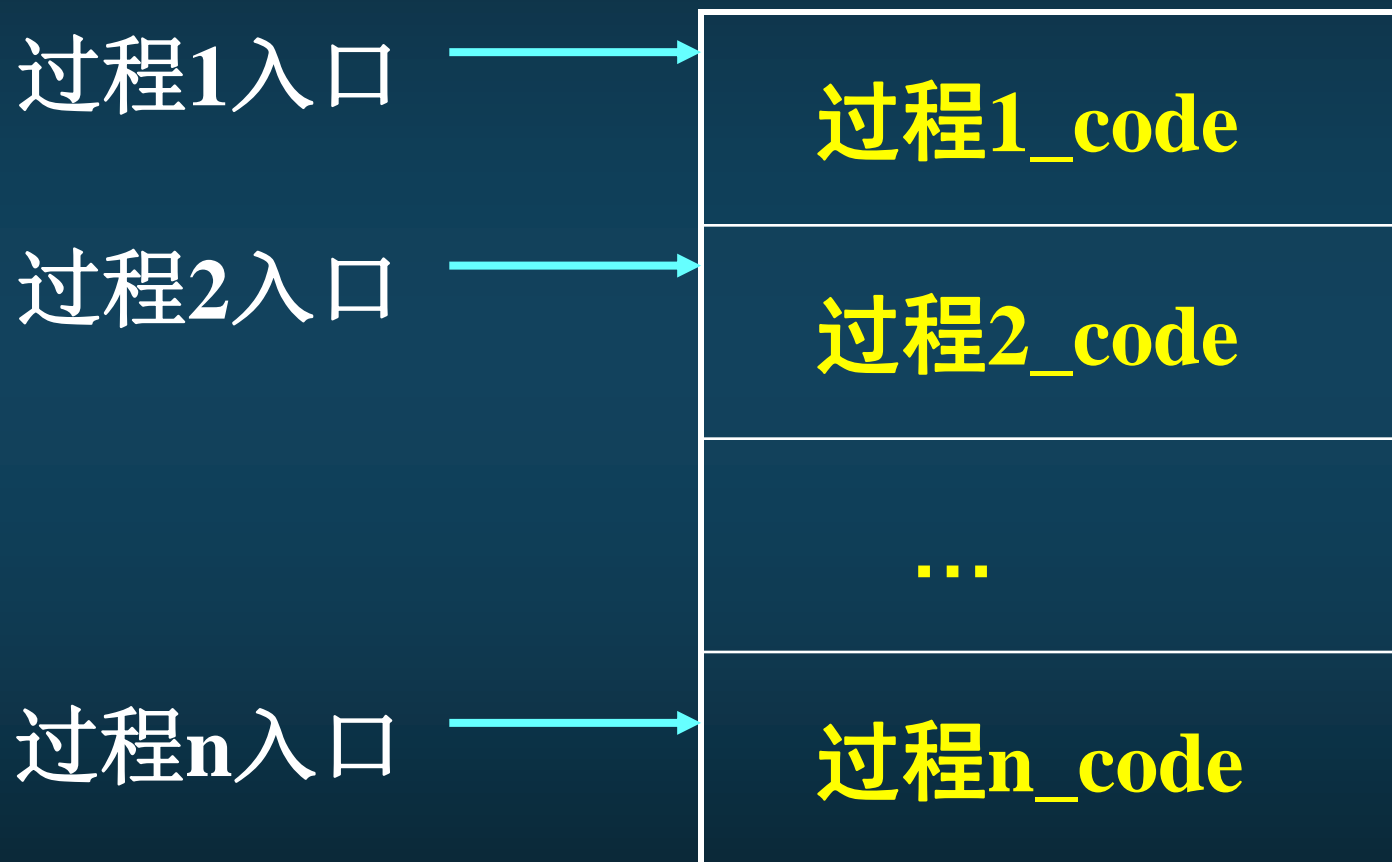
在内存中开辟一个称为堆的存储区，程序运行每当需要（申请）时就按照某种分配原则在堆的自由区（可占用区）中，分配能满足其需要的存储空间给它，使用后需要释放操作，再将不再占用的存储空间归还给堆的自由区。

💧 运行时存储分配的一个原则是，尽可能对数据对象进行静态分配。

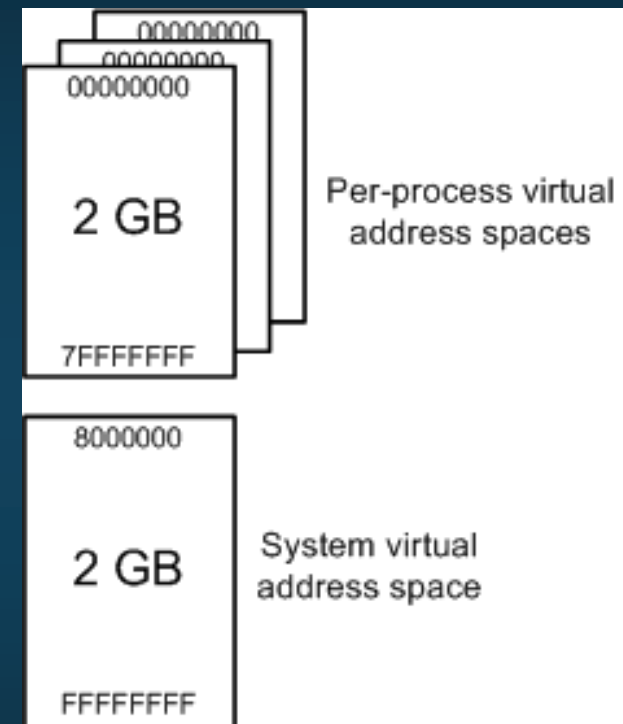
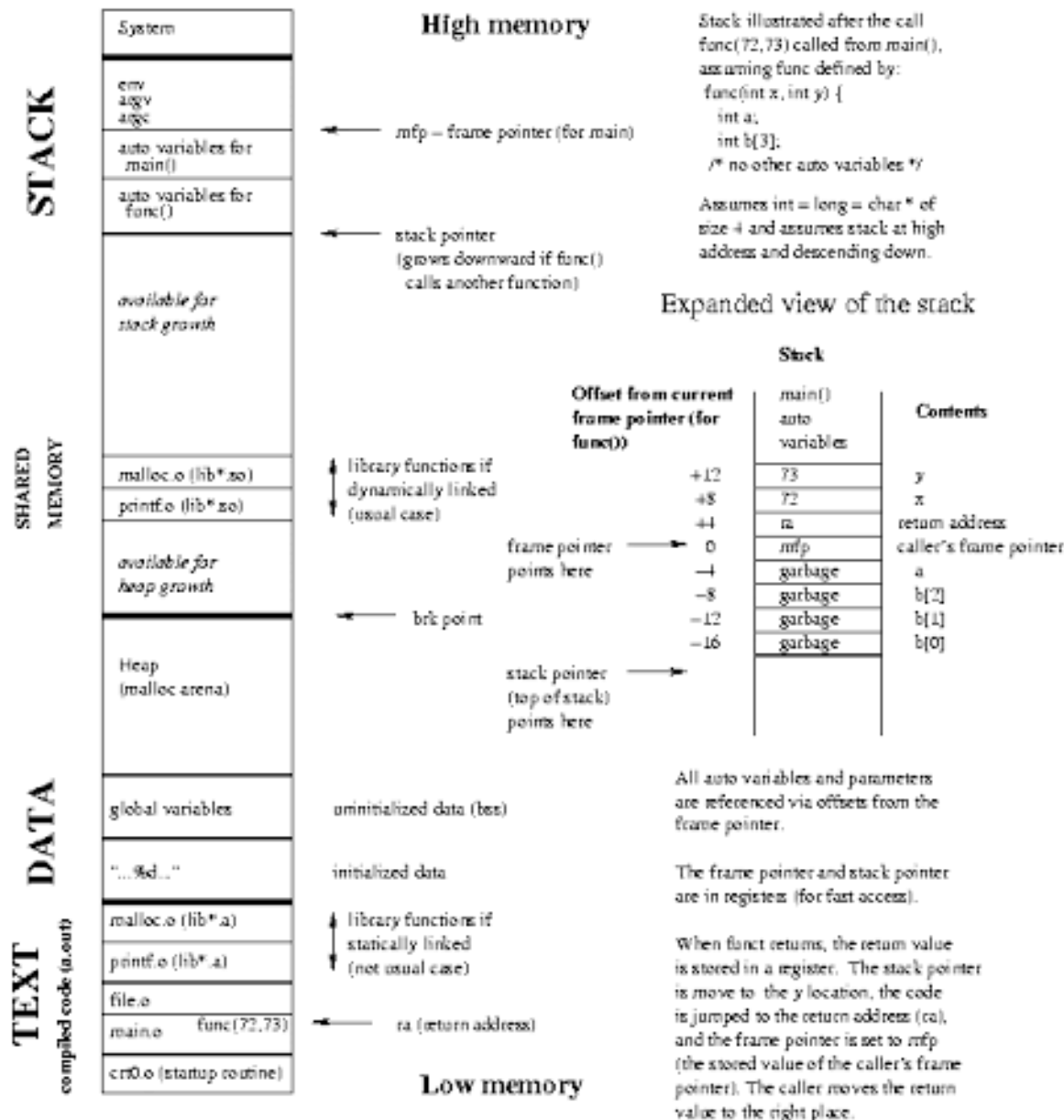
■ 存储区保存的对象

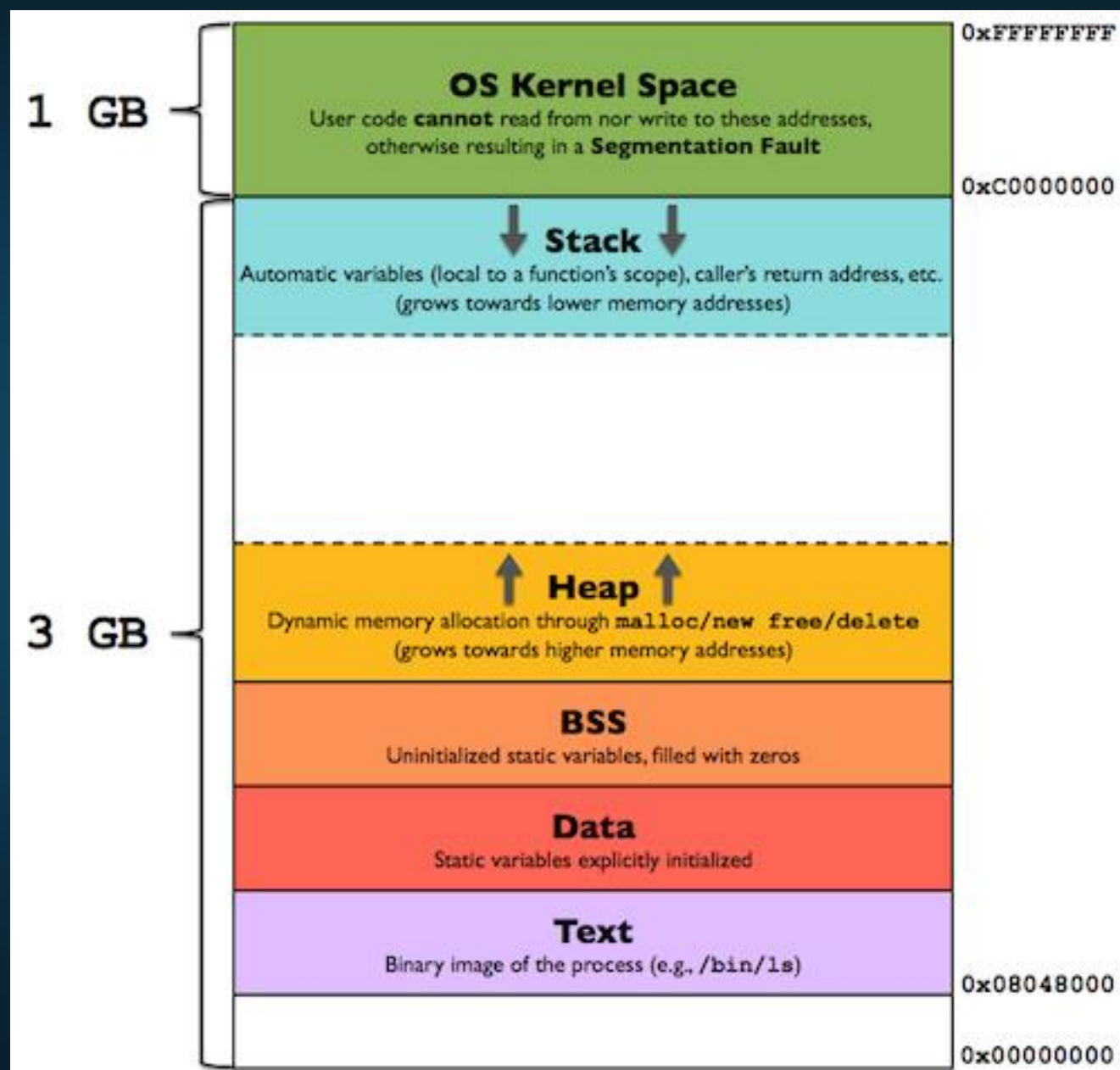
- (1) 生成的目标代码。(Code区)
- (2) 目标代码运行时的数据空间。包括用户定义的各种类型的数据对象，作为保存中间结果和传递参数临时工作单元，组织输入、输出所需的缓冲区等。
- (3) 记录过程活动的控制栈。

■ 代码区



Memory Layout (Virtual address space of a C process)





- 存储分配的重要单元—过程的活动记录AR
(Procedure Activation Record)

一块连续的存储区，被用来存放过程或函数的一次调用执行所需要的信息。

活动记录的组织依赖于目标机体系结构，被编译的语言特性等。



■ 活动记录基本内容

自变量实参空间

局部数据空间

局部临时变量空间

保存的机器状态信息

返回地址 **RA**

[存取链 **SL**]

[控制链 **DL**]

■ 静态存储分配

在编译时能够确定目标程序运行时所需的全部数据空间的大小，即在编译时就可以将程序中的名字关联到存储单元，确定其存储位置，这种分配策略称为静态存储分配。

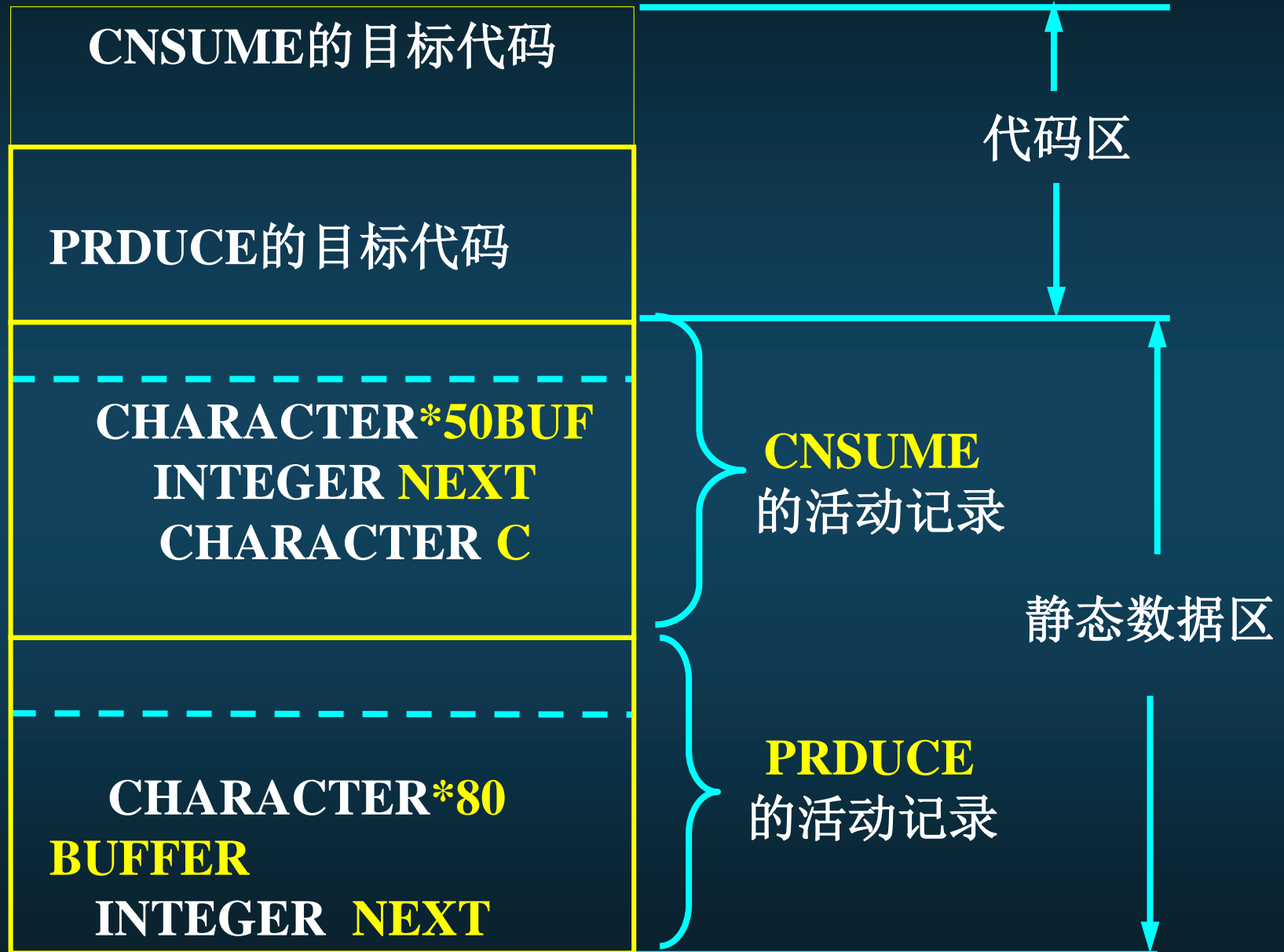
特点：绑定是1对1的映射。名字在程序编译时与存储空间结合，每次过程活动时，它的名字映射到同一存储单元。程序运行时不再有对存储空间的分配。

例7.2 给出FORTRAN程序如下。

```
(1)      PROGRAM CNSUME
(2)      CHARACTER *50 BUF
(3)      INTEGER NEXT
(4)      CHARACTER C, PRDUCE
(5)      DATA NEXT / 1 / ,BUF/ ' ' /
(6)      6   C=PRDUCE( )
(7)      BUF(NEXT:NEXT)=C
(8)      NEXT=NEXT+1
(9)      IF(C.NE.' ') GOTO 6
(10)     WRITE(* , '(A)' ) BUF
(11)     END
```

例7.2 给出FORTRAN程序如下。

```
(12)      CHARACTER FUNCTION PRDUCE()
(13)      CHARACTER * 80 BUFFER
(14)      INTEGER NEXT
(15)      SAVE BUFFER , NEXT
(16)      DATA NEXT /81/
(17)      IF (NEXT.GT.80) THEN
(18)      READ ( * , '(A)' ) BUFFER
(19)      NEXT=1
(20)      END IF
(21)      PRDUCE=BUFFER(NEXT:NEXT)
(22)      NEXT=NEXT+1
(23)      END
```



■ 静态分配对语言的限制

(1) 数据对象的长度和它在内存中的位置的限制必须在编译时知道。

(2) 不允许递归过程，因为一个过程的所有活动使用同样的局部名字结合。

(3) 数据结构不能动态建立，因为没有运行时的存储分配机制。

■ 动态存储分配

在编译时不能确定目标程序运行时所需的全部数据空间的大小，而是在目标程序运行时动态确定的。

■ 栈式动态存储分配

在内存中开辟一个栈区，按栈的特性进行存储分配。程序运行时，每当进入一个函数或过程，该函数所需的存储空间动态地分配于栈顶，函数返回时，就释放这部分空间。

函数所需的数据空间

生存期在本过程本次活动中的数据对象 (局部变量、临时变量...)

管理过程活动的记录信息 (过程调用中断时当前机器的状态信息)

■ 嵌套过程语言的栈式分配 (主要特点)

(语言) 一个过程可以引用包围它的任一外层过程所定义的标识符 (如变量, 数组或过程等)。

(实现) 一个过程可以引用它的任一外层过程的最新活动记录中的某些数据。

- 存储分配的重要单元—过程的活动记录AR
(Procedure Activation Record)

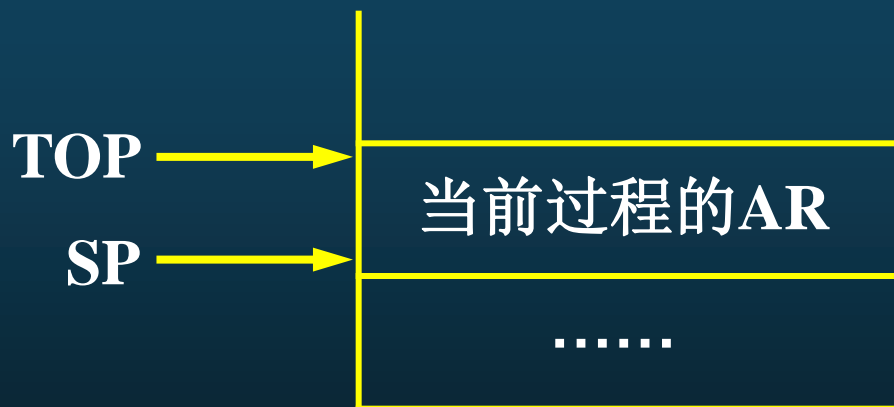
一块连续的存储区，被用来存放过程或函数的一次调用执行所需要的信息。

活动记录的组织依赖于目标机体系结构，被编译的语言特性等。




■ 活动记录的栈

称为运行时栈或调用栈。它随着程序执行时发生的调用链生长或缩小。每个过程每次在调用栈上可以有不同的活动记录，每个都代表一个不同的调用。

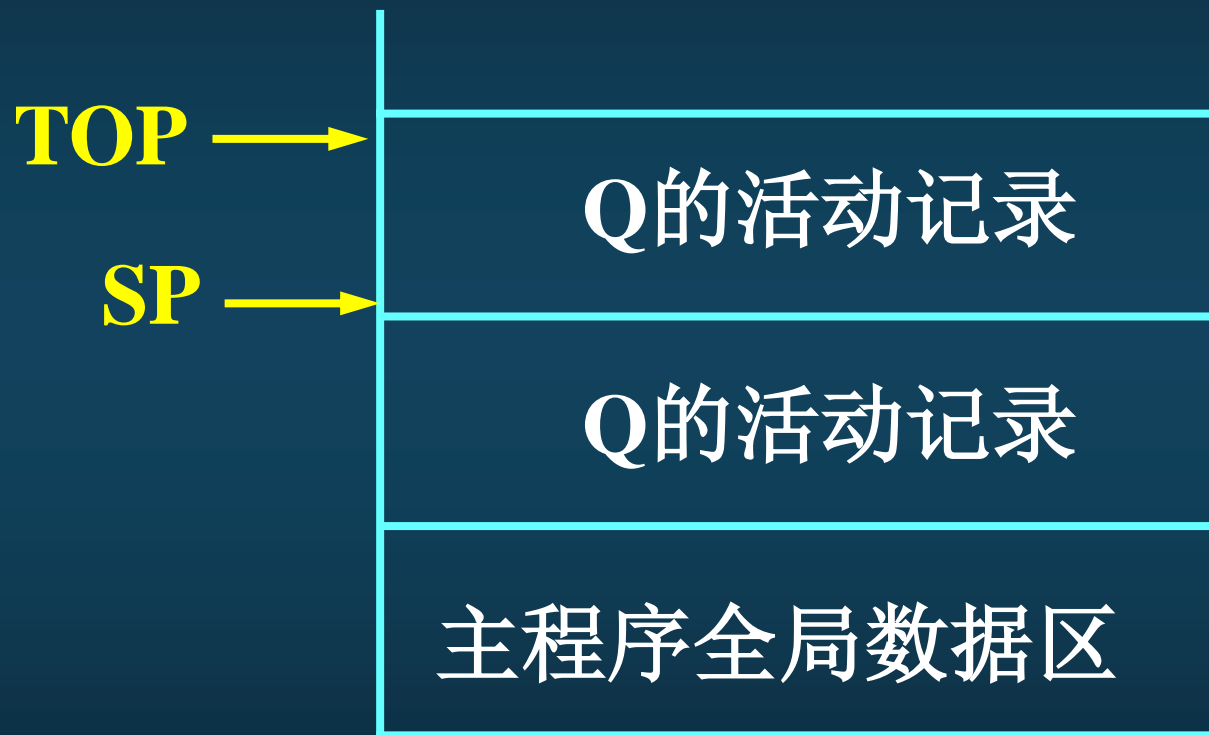


例7.3 设有如下程序

```
main
  全局变量的说明
    proc  R
      .....
    end R;
    proc  Q
      .....
    end Q;
  主程序执行语句
end main
```



Main → Q → Q



■ 关键技术:

解决对非局部量的引用（存取）。

设法跟踪每个外层过程的最新活动记录AR的位置。

■ 跟踪办法:

1. 用静态链（如PL/0的SL）。
2. 用DISPLAY表。

AR

局部变量
机器状态信息
存取链
控制链
实参
返回地址



SL:存取链。指向定义该过程的直接外层过程运行时最新活动记录的基地址

SL

SP/DL

DL: 控制链。指向调用该过程前正在运行过程的活动记录基地址。

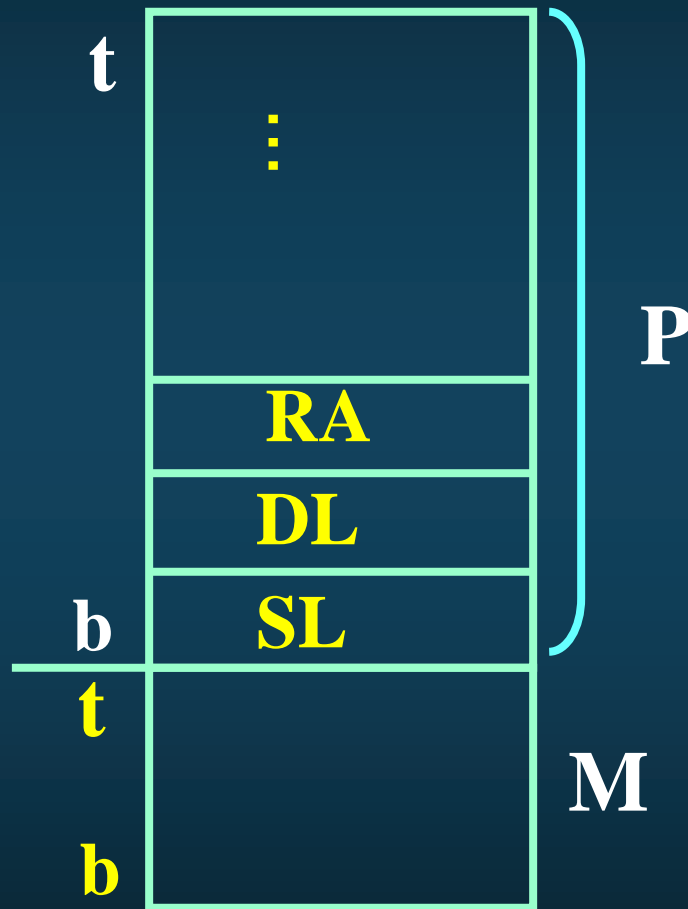

```

const a=10;
var b,c;
procedure p;
begin
    c:=b+a;
end;
begin
    read(b);
    while b#0 do
    begin
        call p;
        write(2*c);
        read(b);
    end
end.

```

(0) jmp 0 8 转向主程序入口
 (1) jmp 0 2 转向过程p入口
 (2) int 0 3 过程p入口,为过程p开辟空间
 (3) lod 1 3 取变量b的值到栈顶
 (4) lit 0 10 取常数10到栈顶
 (5) opr 0 2 次栈顶与栈顶相加
 (6) sto 1 4 栈顶值送变量c中
 (7) opr 0 0 退栈并返回调用点(16)
 (8) int 0 5 主程序入口开辟5个栈空间
 (9) opr 0 16 从命令行读入值置于栈顶
 (10) sto 0 3 将栈顶值存入变量b中
 (11) lod 0 3 将变量b的值取至栈顶
 (12) lit 0 0 将常数值0进栈
 (13) opr 0 9 次栈顶与栈顶是否不等
 (14) jpc 0 24 等时转(24) (条件不满足转)
 (15) cal 0 2 调用过程p
 (16) lit 0 2 常数值2进栈
 (17) lod 0 4 将变量c的值取至栈顶
 (18) opr 0 4 次栈顶与栈顶相乘(2*c)
 (19) opr 0 14 栈顶值输出至屏幕
 (20) opr 0 15 换行
 (21) opr 0 16 从命令行读取值到栈顶
 (22) sto 0 3 栈顶值送变量b中
 (23) jmp 0 11 无条件转到循环入口(11)
 (24) opr 0 0 结束退栈

■ M调用过程P



■ 解决对非局部量的引用：用Display表

Display表 — 嵌套层次显示表

当前激活过程的层次为**K**，它的Display表含有**K+1**个单元，自第 **K+1**、**K**、...、**0** 个单元依次存放着现行层，直接外层...直至最外层的每一过程的最新活动记录的基地址。

主程序设为第0层

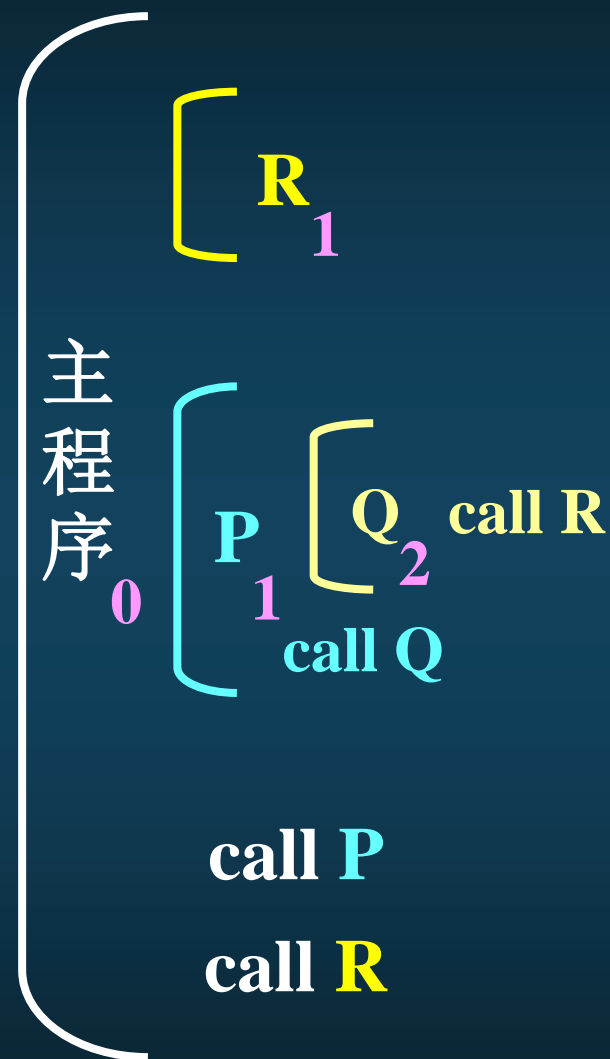
例7.4

```

program main( i,0) ;
    .....
    proc R(c,d) ;
        .....
    end /*R*/
    proc P(a);
        .....
        proc Q(b);
            .....
            R(x,y);
        end /* Q*/
        .....
        Q(z);
    end /* P*/
    .....
    P(W);
    .....
    R(U,V);
    .....
end /* main*/

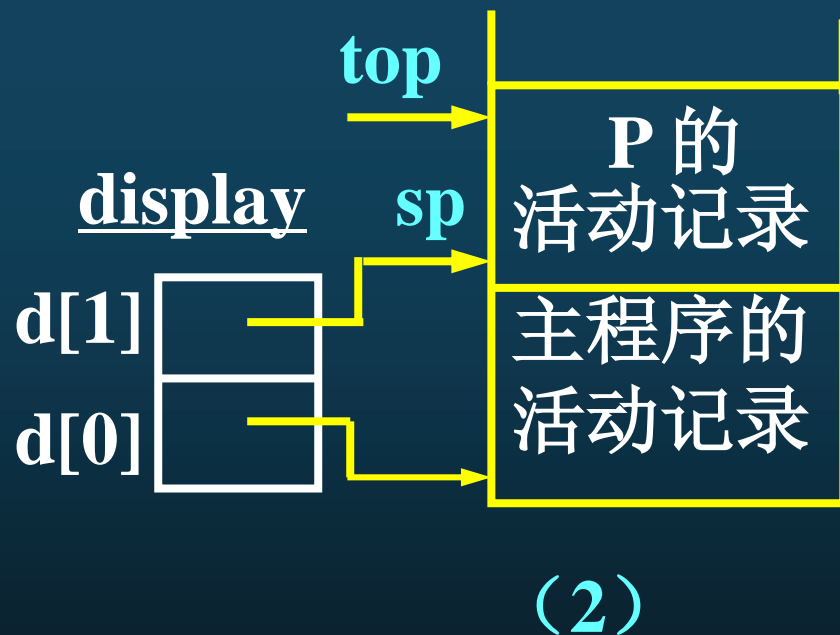
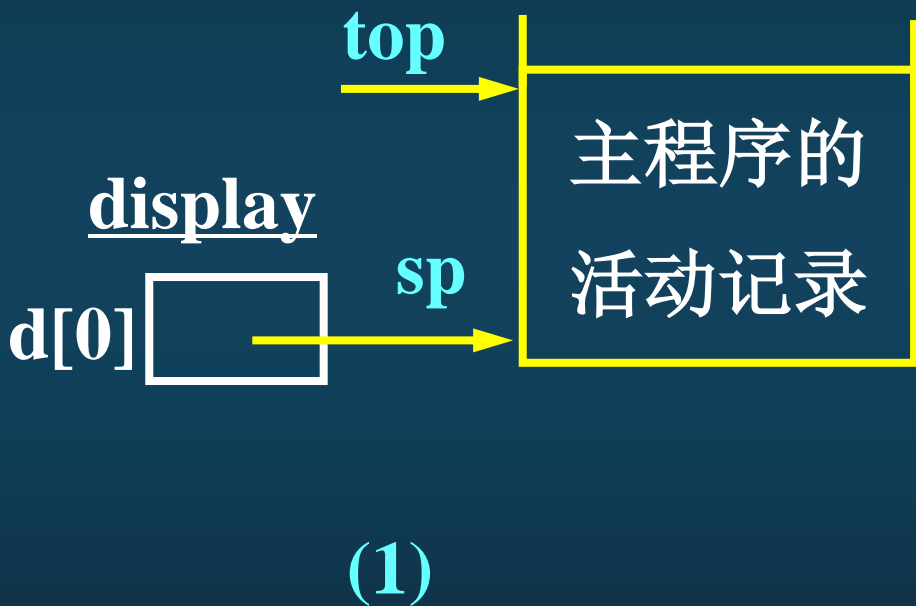
```

程序结构图

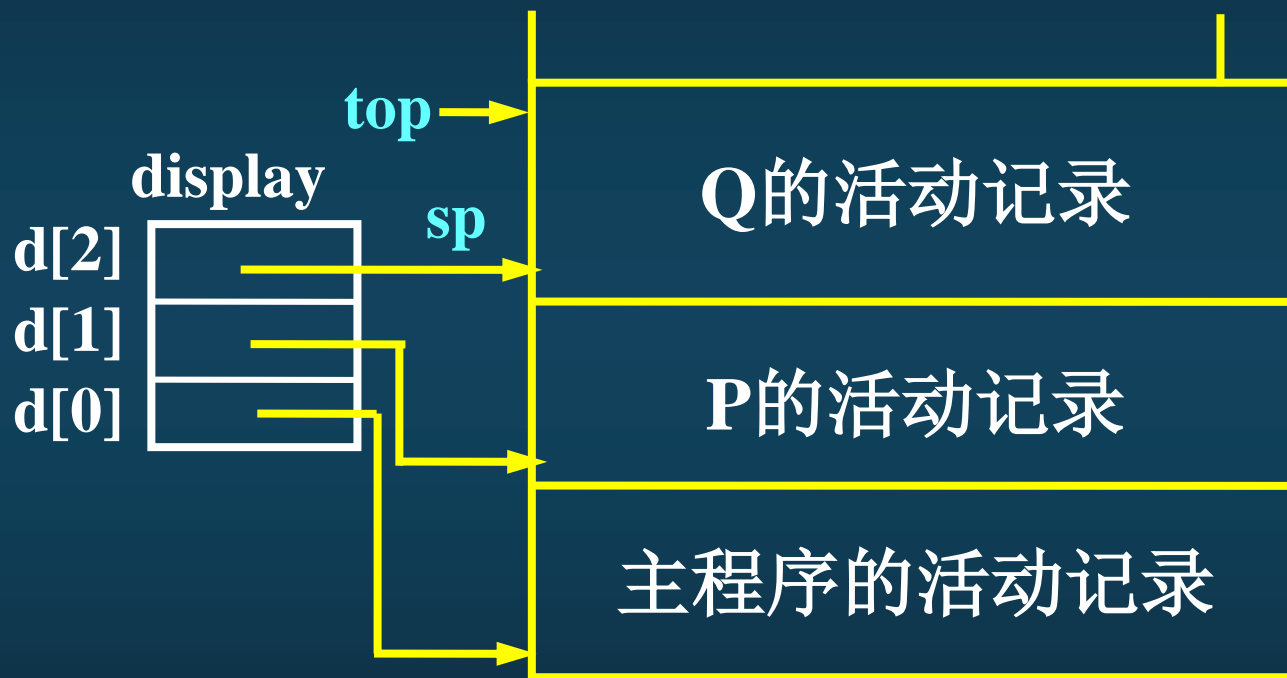


■ 用Display表的方案

主程序 (1) \rightarrow P(2) \rightarrow Q(3) \rightarrow R(4)

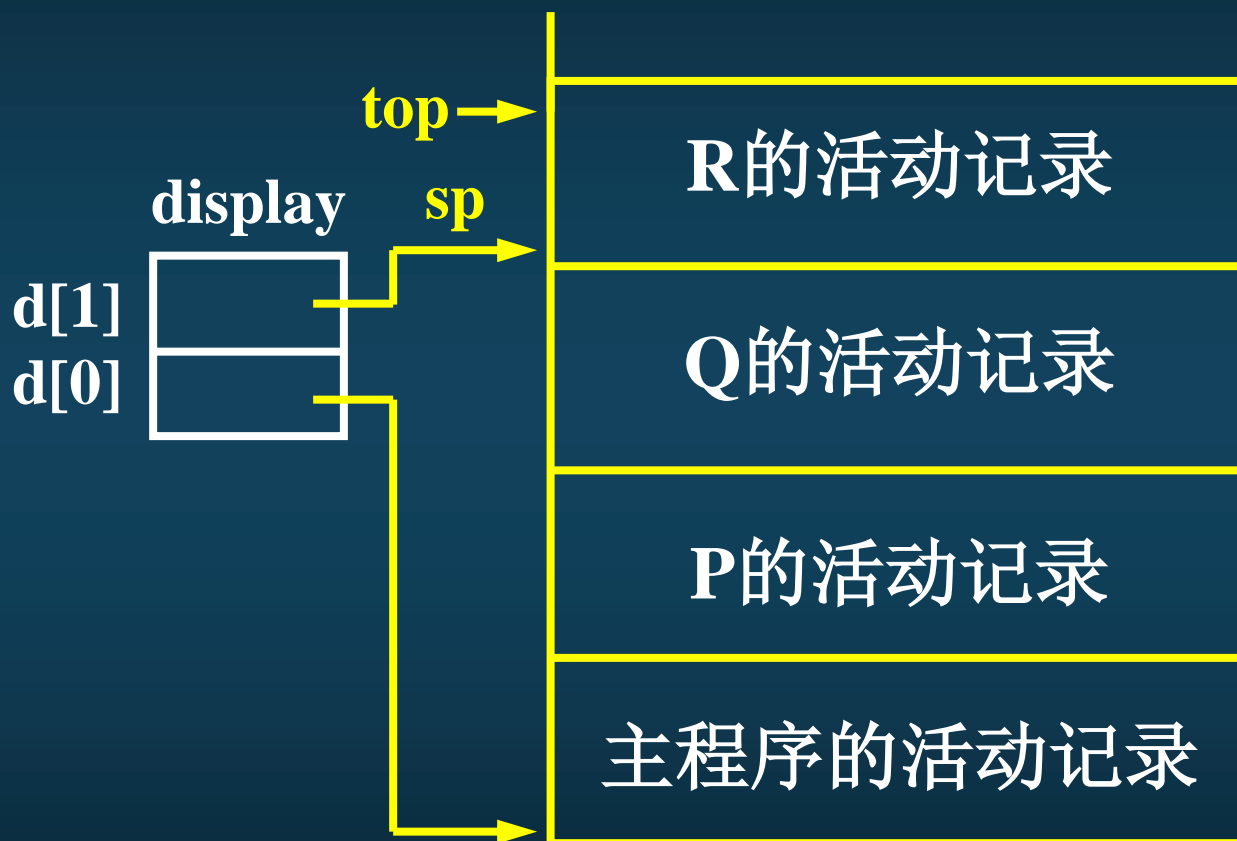


主程序 (1) → P(2) → Q(3) → R(4)



(3)

主程序 (1) → P(2) → Q(3) → R(4)



(4)

例7.5 设有C程序

gcd(15,10)



gcd(10,5)



gcd(5,0)

```
#include <stdio.h>
```

```
int x,y ;
```

```
int gcd(int u,int v)
```

```
{ if (v==0) return u;
```

```
    else return gcd(v, u % v);
```

```
}
```

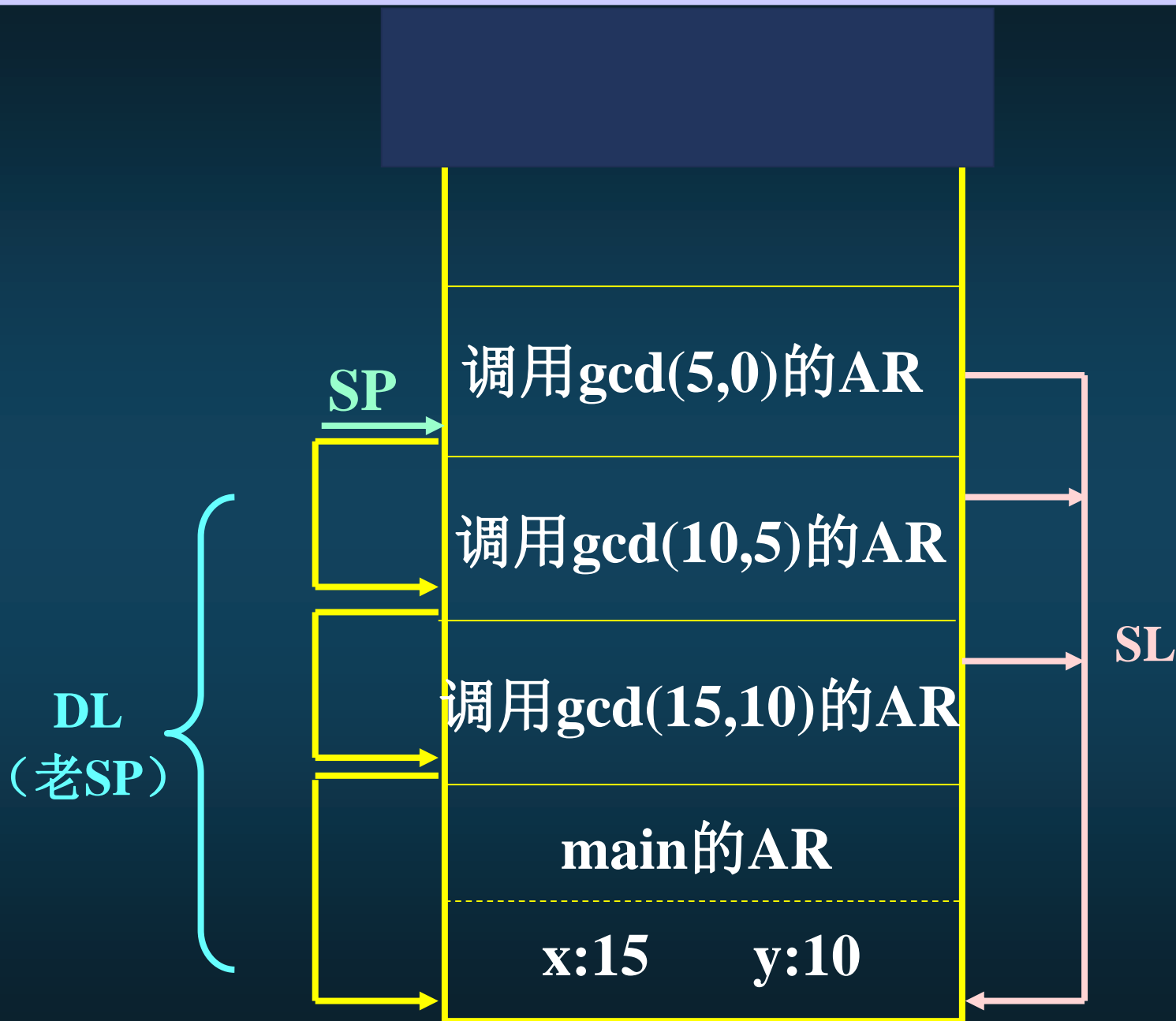
```
main()
```

```
{ scanf("%d%d",&x,&y);
```

```
    printf("%d\n",gcd(x,y));
```

```
    return;
```

```
}
```

例7.6 P204_例6.2

main_block () **/* P1 */**

float **x,y**; string **name**;

.....

.....

end (* main_block *)

p2_block (int id);

int **x**;

.....

call **p3_block(id+1)**;

.....

end (* p2_block *)

p3_block (int j);

.....

p4_block ();

int array **f[j]**; logical **test**;

.....

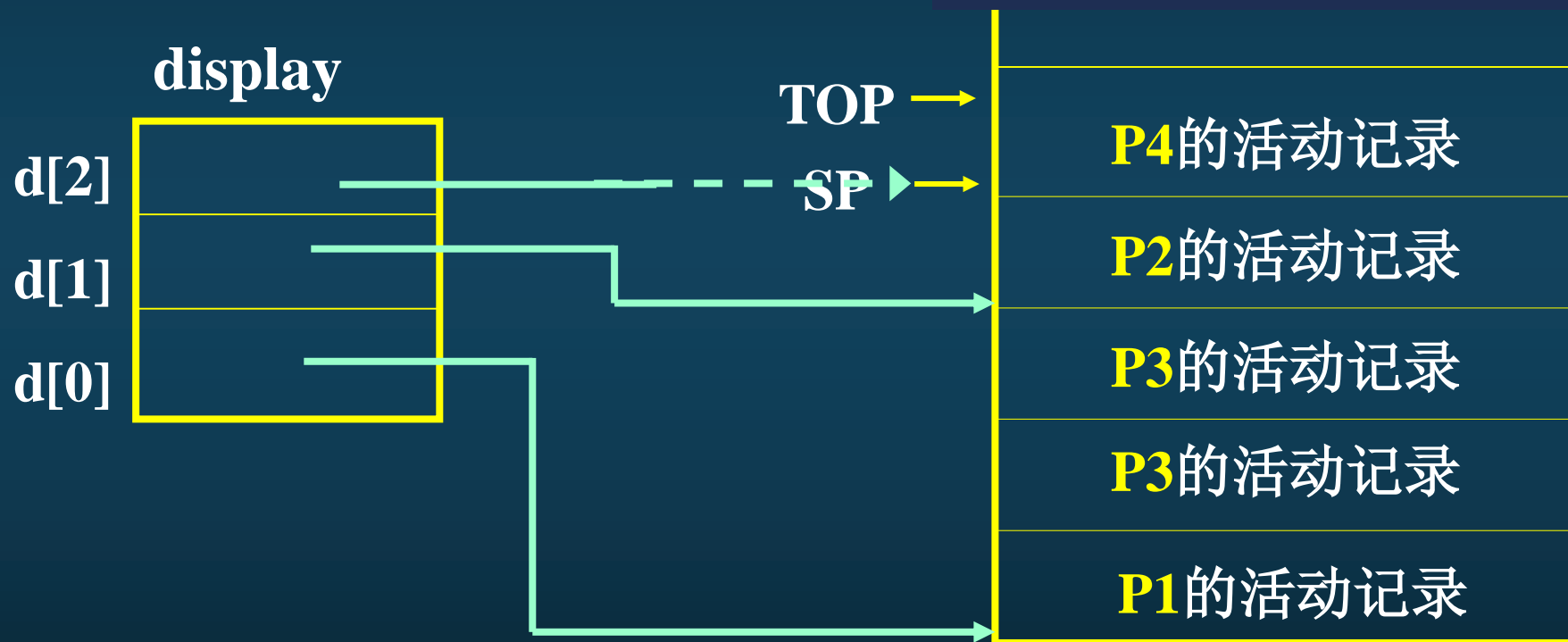
end (* p4_block *)

end (* p3_block *)

.....

call **p2_block(x/y)**;

main(P1) → P3 → P3 → P2 → P4



■ 堆分配策略

特点： 可任意分配和撤消数据； 对程序设计语言没有限制；

■ 堆分配的基本思想：



(a) 堆分配的存储空间



(b) 堆分配的可用存储空间链