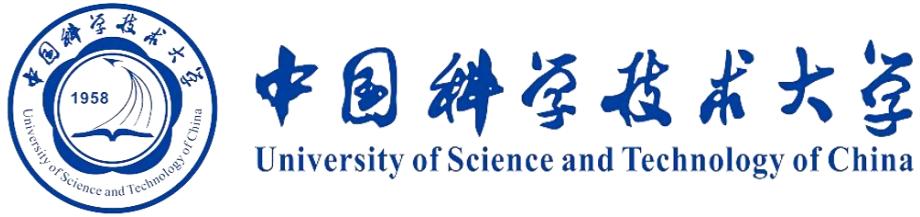


中国科学技术大学
University of Science and Technology of China

实用算法设计

➢ 栈、队列与哈希

主讲：娄文启

目录

1. 栈和队列

2. 哈希表

顺序表——动态定义（Vector）

1

```
#include<iostream>
class Object{
public:
// 构造函数
Object(int v_):v(v_){
    std::cout << "Construct Object" << std::endl;}
    // 拷贝构造函数
    Object(const Object& other):v(other.v){
    std::cout << "Copy Object" << std::endl;}
    private:
    int v;
};
```

2

```
int fun1(Object b){
    std::cout<<"fun1"<<std::endl;  return 0;}

int fun2(Object &b){
    std::cout<<"fun2"<<std::endl;  return 0;}
```

3

```
int main(){
    Object obj1(1);
    Object obj2(obj1); cout<<endl;
    Object obj3=obj1; cout<<endl;
    fun1(obj1); cout<<endl;
    fun2(obj1);}
```

输出：？

```
Construct Object
Copy Object

Copy Object

Copy Object
fun1

fun2
```

顺序表——动态定义（Vector）


```
std::vector<Object> v;
//向vector添加一个Object
Object obj1(1);
v.push_back(obj1);
```

1

输出?

```
Construct Object
Copy Object
```

思考:

为什么不是两次拷贝,
(**传参**拷贝构造 + Vector内部**拷贝构造**)?

```
void push_back(const value_type& __x)
```

```
std::vector<Object> v;
Object obj1(1);
v.push_back(obj1);
v.push_back(obj1);
```

2

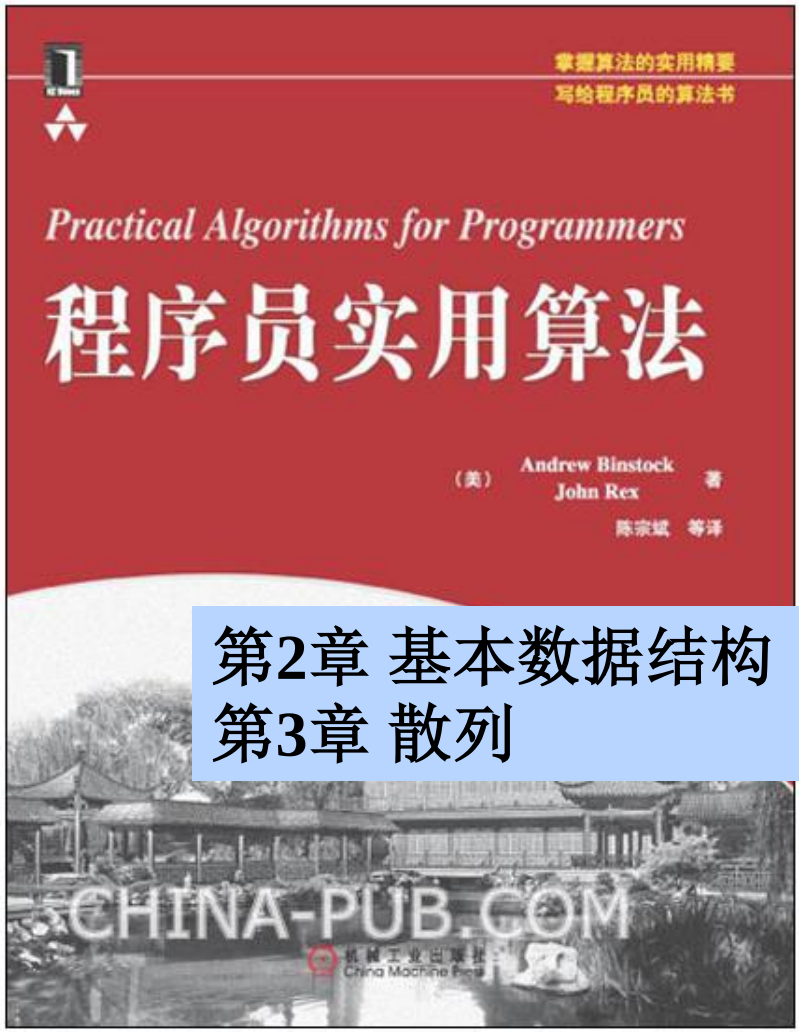
输出?

```
Construct Object
Copy Object
Copy Object
Copy Object
```

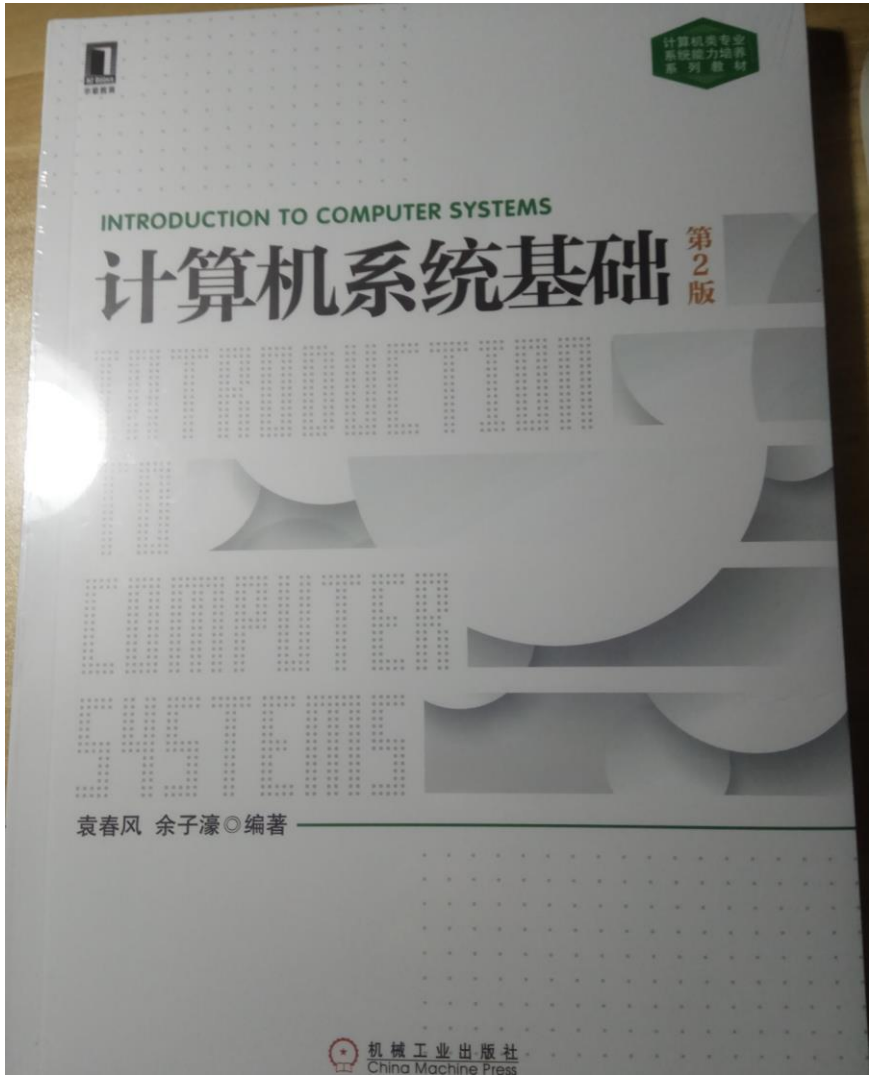
思考:

为什么多拷贝了一次,
(明明添加了两次)?

参考内容



第2章 基本数据结构
第3章 散列



4.1 栈和队列

- 理解：
  - 栈和队列的特殊性（与一般的线性表对比）
  - 栈和队列的定义的实现
  - 栈和队列的原子操作的实现
    - 判空？判满？
    - 入栈，出栈；入队，出队
  - 顺序栈为什么更常用
  - 循环队列的设计初衷
- 会利用栈和队列的原子操作来解决实际问题
- 了解 函数调用过程中，到底发生了什么？（函数调用是栈的应用）
  - eip
  - 栈框架：ebp 和 esp

4.1 栈和队列

4.1 栈

- **顺序栈的定义及实现**
- 链栈的定义及实现
- 栈的应用：案例分析——函数调用

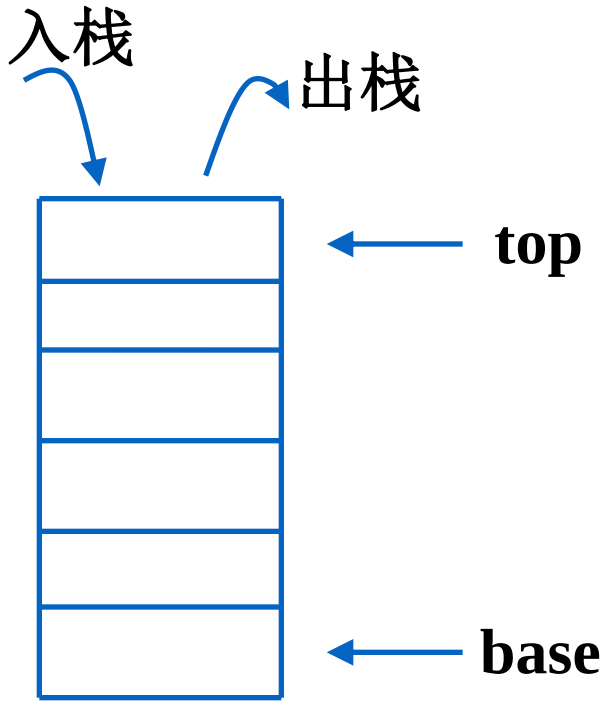
4.2 队列

- 循环队列的定义及实现
- 链队列的定义及实现
- 队列的应用：案例分析

顺序栈的定义及实现

- 什么是栈?
  - 是操作受限的特殊线性表。
    - 线性表：有限个数据元素组成的序列，记作(a1,a2, … , an)
  - 是限定在一端(top)进行插入或删除操作的线性表;
  - 栈顶(top)：允许插入或删除的一端;
  - 栈底（base/bottom）

后进先出！



- 存储结构：
  - 顺序栈：顺序方式存储
  - 链栈：链式存储
- 原子操作:
  - 创建空栈
  - 销毁已有栈
  - 查找直接后继和直接前驱
  - 入栈
  - 出栈
- 栈的应用举例：函数调用;

顺序栈的定义及实现

- 顺序栈：利用动态定义的顺序表来实现栈。
- C 定义：

```
struct stack_struct {
    ElemType  *base;        /* point to base of stack */
    int              stack_size;  /* number of elements */
    int              min_stack;   /* bottom-most element */
    int              max_stack;  /* last possible element */
    int              top ;        /* current top */
};
typedef struct stack_struct Stack;
```

顺序栈的定义及实现——原子操作

- Stack * this_stack;
- 创建空栈（初始化栈）：
  - 将this_stack->base指向分配成功的连续内存空间的开始处; this_stack->top=-1（注意不是0,若为0则意味着栈中已有一个元素了）；
- 销毁已有栈:
  - this_stack->top=-1;
  - free(this_stack->base);
- 入栈：
  - 若栈未满，则让栈顶下标top递增，然后将要入栈的数复制到栈顶处。（分析：插入位置为栈顶元素的下一个，无须判断位置的合法性；上溢即栈满的条件需要判断，由于是增量式分配，故栈满时需要重新申请空间）；
- 出栈：
  - 若栈未空，则将栈顶元素复制到一个指定的目的地；并让栈顶下标top递减以指向栈上的下一个元素：this_stack->top -= 1
- 问题：如何判定栈满？如何判定栈空？

顺序栈的定义及实现——原子操作

```
Stack *CreateStack ( int how_many )
{
    Stack *pstk;
    assert ( how_many > 0 );    /* make sure the size is legal */
    pstk = (Stack *) malloc ( sizeof ( Stack ));
    if ( pstk == NULL )
        return ( NULL );
    pstk->stack_size = how_many;
    pstk->base = ( struct StkElement * )
        malloc ( how_many * sizeof ( struct StkElement ));
    if ( pstk->base == NULL ) /* error in allocating stack */
        return ( NULL );
   pstk->min_stack = 0;
   pstk->top = -1;
   pstk->max_stack = how_many - 1;
   ClearStack ( pstk );
   return ( pstk );
}
```

顺序栈的定义及实现——原子操作

```
void ClearStack ( Stack *this_stack )
{
    this_stack->top = -1;
}
```

顺序栈的定义及实现——原子操作

```
void DestroyStack ( Stack *this_stack )
{
    ClearStack (this_stack);
    free(this_stack->base);
    this_stack->base=Null;
}
```

顺序栈的定义及实现——原子操作

```
int PushElement ( Stack *this_stack, struct StkElement *
  to_push )
{
    /* is stack full? */
    if ( this_stack->top == this_stack->max_stack )
        return ( 0 );
    this_stack->top += 1;
    memmove ( &(( this_stack->base )[this_stack->top] ),
  to_push, sizeof ( struct StkElement ));
    return ( 1 );
}
```

顺序栈的定义及实现——原子操作

```
int PopElement ( Stack *this_stack, struct StkElement *
  destination )
{
    if ( this_stack->top == -1 ) /* stack empty, return error */
        return ( 0 );
    memmove ( destination,&(( this_stack->base) [this_stack-
  >top] ), sizeof(struct StkElement ));
    this_stack->top -= 1;
    return ( 1 );
}
```

顺序栈的定义及实现——原子操作

栈空？栈满？

栈空：s->top == -1；

栈满：stack->top == this_stack->max_stack

入栈和出栈的时间复杂度：都为O(1)。

4.1 栈和队列

4.1 栈

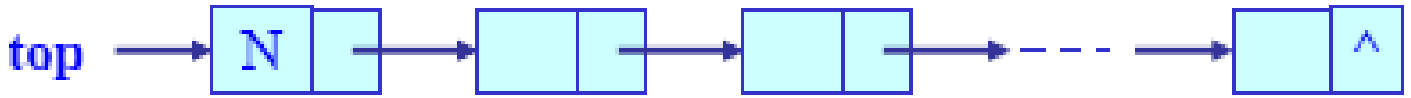
- 顺序栈的定义及实现
- **链栈的定义及实现**
- 栈的应用：案例分析——函数调用

4.2 队列

- 循环队列的定义及实现
- 链队列的定义及实现
- 队列的应用：案例分析

链栈的定义及实现

- 链栈：用链表来实现栈。
  - 栈顶(top)：用链表的头指针来表示。
  - 栈底(base)：无需额外表示。(只在栈顶top中进行操作)



- 特点：
  - 无栈满问题：内存可扩充（除非是内存不足）；

后进先出！

- 问题：
  1）如何判断栈空?
  2）在链栈下的入栈、出栈操作如何实现?
  3）需引入头结点？（无需）
  4）顺序栈中为何需要定义base?

总结——栈的存储结构

- 推荐使用顺序栈
  - 实现简单 & 随机存取
  - 栈的受限操作的特性正好屏蔽了顺序表的弱势
    - 顺序栈中添加和删除数据都是在表尾进行的。

4.1 栈和队列

4.1 栈

- 顺序栈的定义及实现
- 链栈的定义及实现
- **栈的应用：案例分析——函数调用**

4.2 队列

- 循环队列的定义及实现
- 链队列的定义及实现
- 队列的应用：案例分析

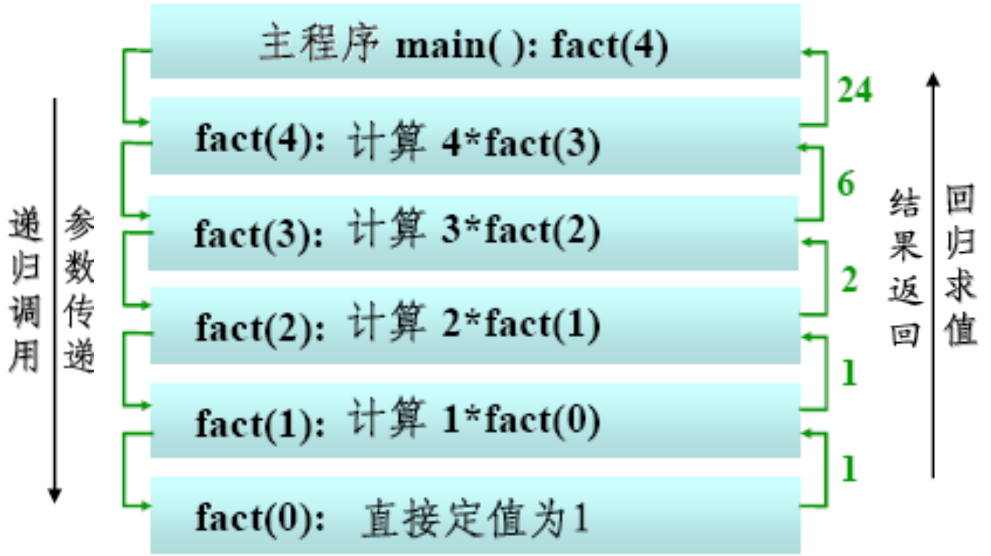
栈的应用：案例分析

- 例：递归的应用：求解n!
  - n!=n*(n-1)! (n>0)
  - 0!=1

栈的应用：案例分析

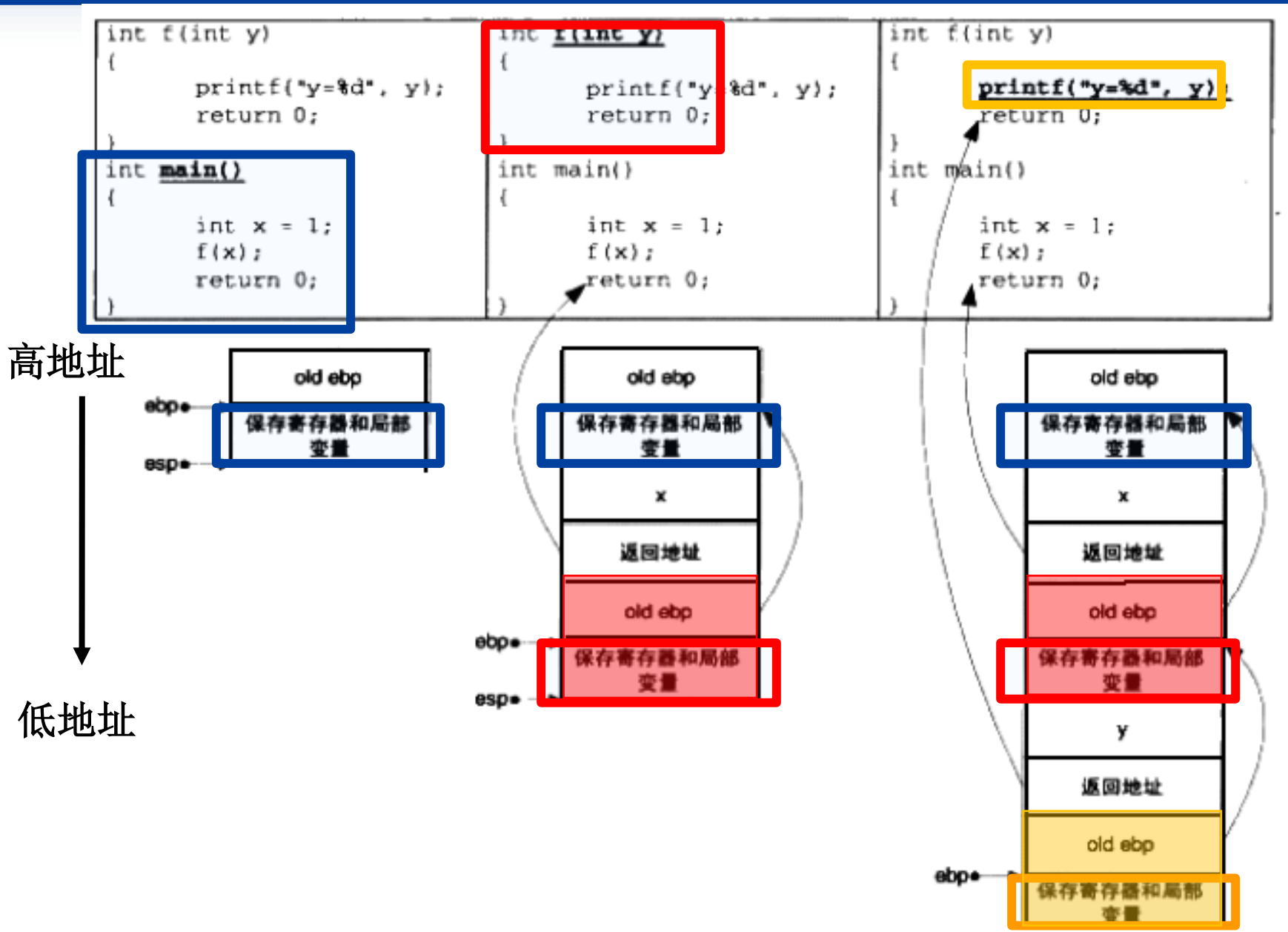
- 例：递归的应用：求解n!

```
long fact( long n)
{
    if (n==0) return 1; //递归结束条件
    else return n*fact(n-1);// 递归的规则
}
```



调用的顺序和返回的顺序是相反的，正好符合了“栈”的特点

栈的应用：案例分析


int f(int y)
{
printf("y=%d", y);
return 0;
}
int main()
{
int x = 1;
f(x);
return 0;
}
高地址
低地址
old ebp
保存寄存器和局部变量
ebp
esp
x
返回地址
y

递归的实现总结

- 调用前：
  - 现场保护（参数、返回地址、旧基址 入栈），被调用函数的局部变量的空间分配，控制转移至被调用的函数入口。
- 调用后：
  - 保存计算结果，释放被调函数的数据区，控制转移回调用处。
- 实现——栈
  - “后调用先返回”。系统利用递归工作栈记录各层调用的现场信息。

堆栈

- 堆栈是C语言程序运行时必须的一个记录调用路径和参数的空间
  - 函数调用框架
  - 传递参数
  - 保存返回地址
  - 提供局部变量空间
  - ……
- C语言编译器对堆栈的使用有一套的规则
- 了解堆栈存在的目的和编译器对堆栈使用的规则是理解操作系统一些关键性代码的基础

- 以x86体系结构为例

堆栈寄存器和堆栈操作

- 堆栈相关的寄存器
  - esp，堆栈指针（stack pointer）
  - ebp，基址指针（base pointer）
- 堆栈操作
  - push <寄存器/内存地址>

    PUSH EAX //将 EAX 寄存器的内容压入堆栈;

    PUSH 0x12345678 //将立即数 0x12345678 压入堆栈
  - pop <寄存器/内存地址>

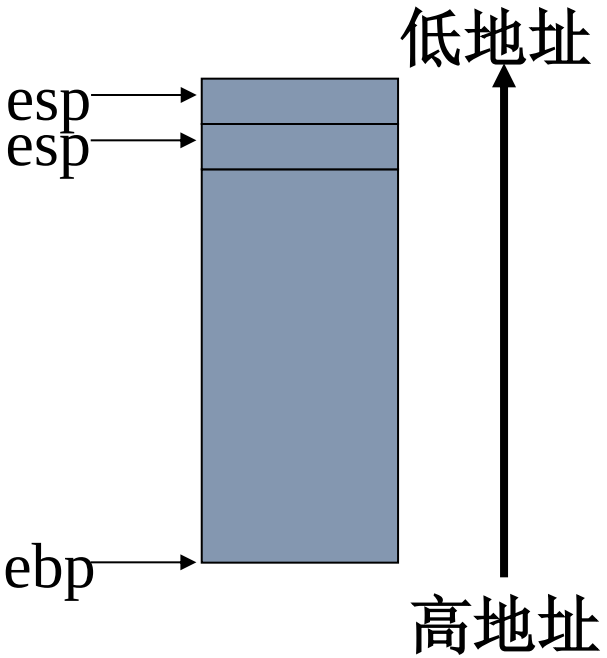
    PUSH EAX //堆栈中弹出一个值(字/双字/四字)到 EAX 寄存器中

    POP DWORD PTR [var] //堆栈顶的数据弹出到内存中的某个变量

栈顶指针esp的变化?

堆栈寄存器和堆栈操作

- 堆栈相关的寄存器
  - **esp，堆栈指针（stack pointer）**
  - **ebp，基址指针（base pointer）**
- 堆栈操作
  - **push <寄存器/内存地址>**
    栈顶地址减少4个字节（32位）
  - **pop <寄存器/内存地址>**
    栈顶地址增加4个字节



对比之前写的stack数据结构，pop/push 汇编命令是如何实现的?

IA32中的pop和push

pop/push 指令是由CPU内部的微指令集自动执行的一个复杂过程。

以**pop destination**汇编指令为例，CPU会执行以下步骤来完成这个操作：

① CPU首先检查 ESP 寄存器当前指向的堆栈顶部地址。

② 然后，CPU从堆栈顶部读取数据（在32位模式下通常是4个字节）并将其放入指定的寄存器或内存位置。

③ 接着，CPU将 ESP 寄存器的值增加相应的大小（在32位模式下增加4个字节），因为数据已经从堆栈中移除。

④最后，ESP 寄存器更新为新的地址，这个新地址是原始 ESP 值加上数据大小后的结果。

这个过程是完全自动化的，不需要程序员进行任何额外的操作。pop 指令简化了从堆栈中检索数据的过程，确保了函数返回和局部变量的管理能够在堆栈上有效进行。

IA-32 寄存器

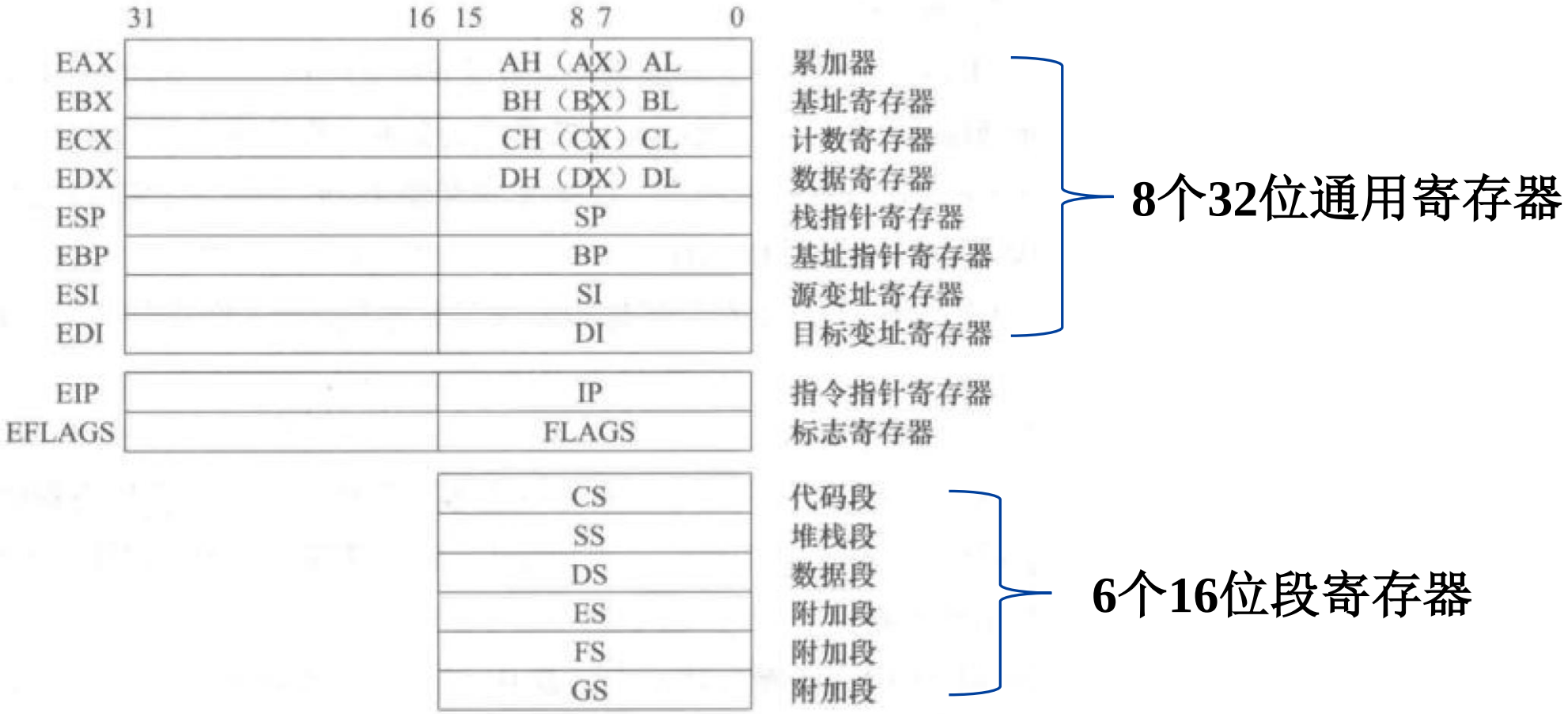


图 3.2　IA-32 的定点寄存器组

MIPS和Intel 架构

- IA-32架构（Intel Architecture 32-bi
  x86架构
  首次应用在Intel 80386微处理器中。
  数据操作数，从1985年的80386到P
  IA-32是一个复杂指令集（CISC）架
  程模型，支持内存管理、软件模块保
  处理等功能。

- MIPS架构是一种精简指令集计算机（
  司于1981年开发。
  它以简洁性、高性能、可扩展性和跨
  域。MIPS通过简化指令集和流水线
  和计算能力。MIPS架构的算术和逻
  允许编译器优化复杂的表达式

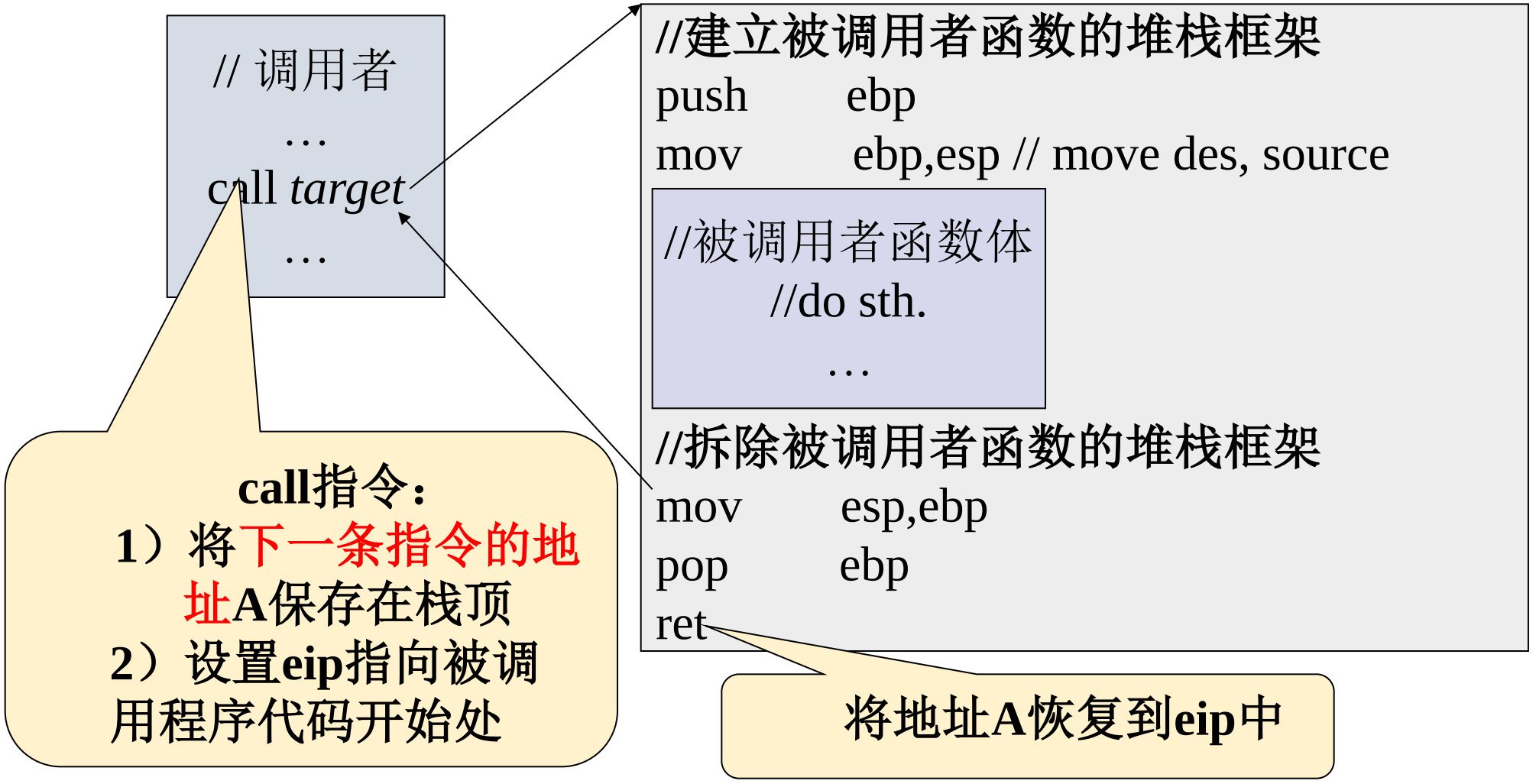
0	`zero`	Always returns 0
1	`at`	(assembly temporary) Reserved for use by assembly
2–3	`v0, v1`	Value returned by subroutine
4–7	`a0–a3`	(arguments) First few parameters for a subroutine
8–15	`t0–t7`	(temporaries) Subroutines can use without saving
24, 25	`t8, t9`	
16–23	`s0–s7`	Subroutine register variables; a subroutine that writes one of these must save the old value and restore it before it exits, so the *calling* routine sees the values preserved
26, 27	`k0, k1`	Reserved for use by interrupt/trap handler; may change under your feet
28	`gp`	Global pointer; some runtime systems maintain this to give easy access to (some) `static` or `extern` variables
29	`sp`	Stack pointer
30	`s8/fp`	Ninth register variable; subroutines that need one can use this as a frame pointer
31	`ra`	Return address for subroutine

利用堆栈实现函数调用和返回

- ebp在C语言中用作记录当前函数调用基址
- 其他关键寄存器
  - cs：eip：总是指向下一条的指令地址
    - 顺序执行：总是指向地址连续的下一条指令
    - 跳转/分支：执行这样的指令的时候，cs：eip的值会根据程序需要被修改
    - call：将当前cs：eip的值压入栈顶，cs：eip指向被调用函数的入口地址
    - ret：从栈顶弹出原来保存在这里的cs：eip的值，放入cs：eip中

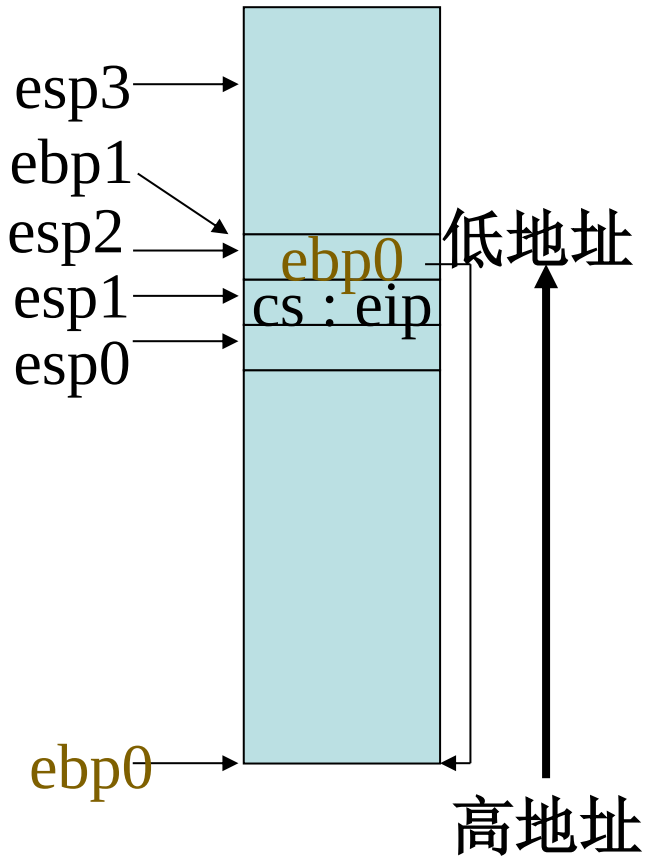
CS (Code Segment): 这是代码段寄存器
EIP (Instruction Pointer): 这是指令指针寄存器

利用堆栈实现函数调用和返回

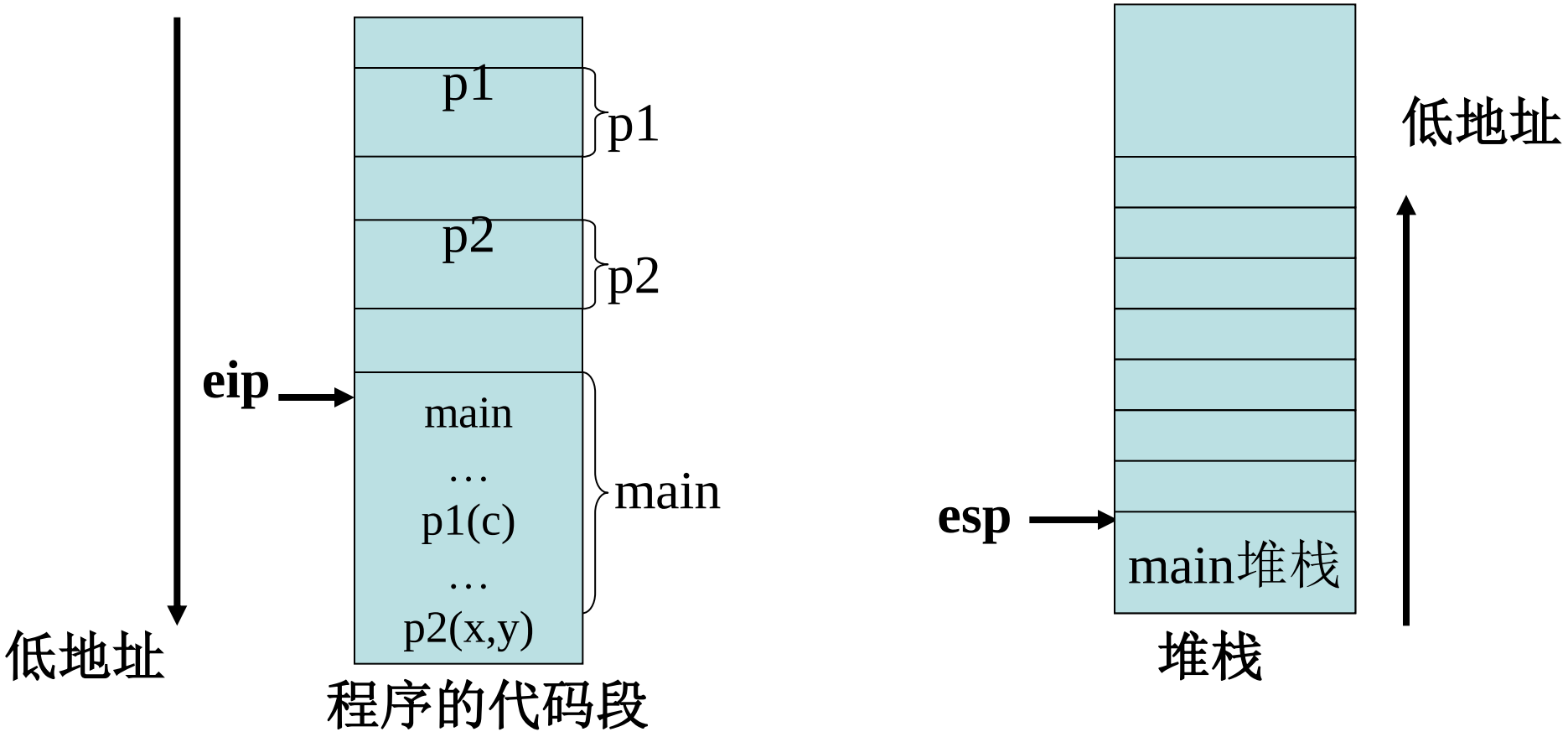


利用堆栈实现函数调用和返回

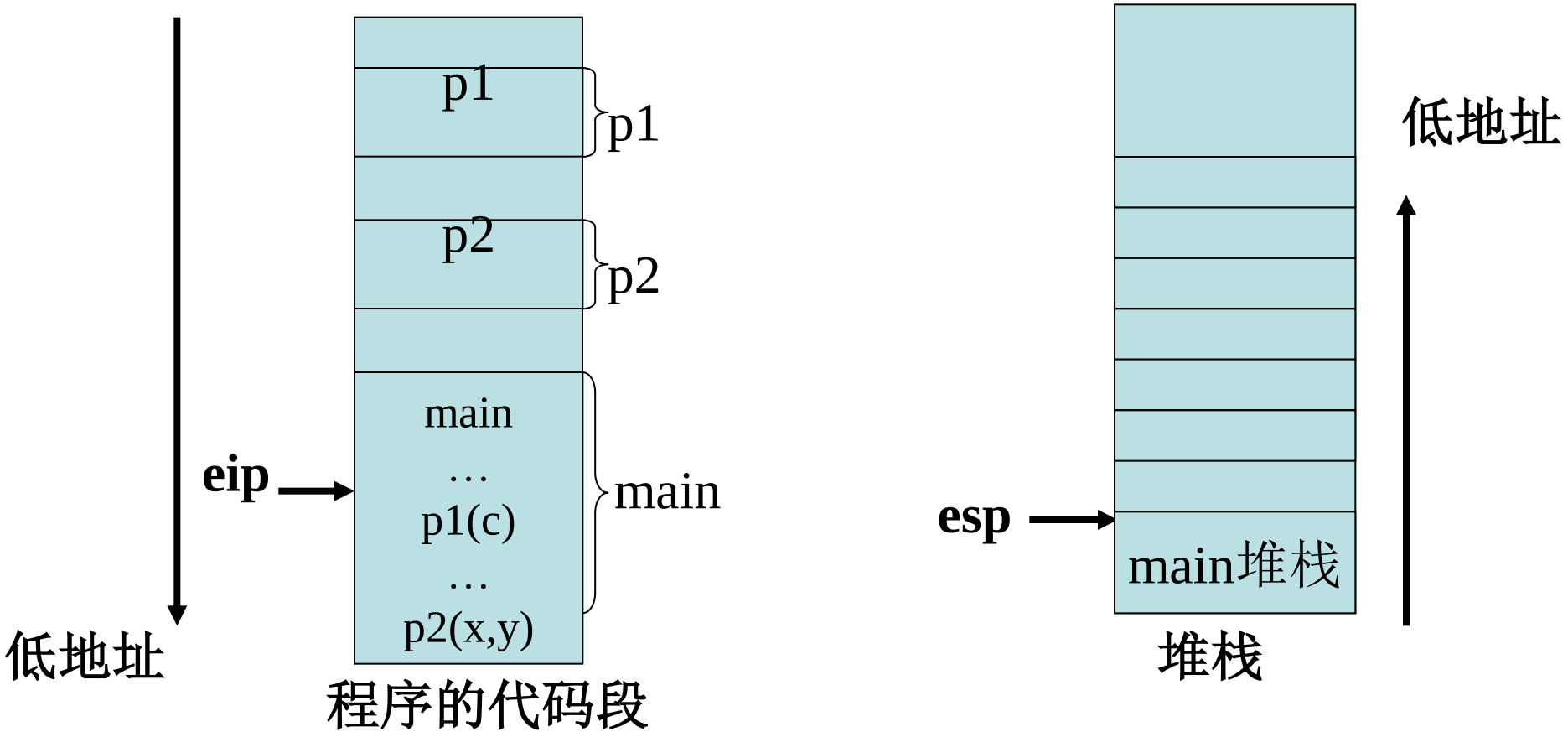
- **call xxx**
  - 执行call之前
  - 执行call时，cs：eip原来的值指向call的下一条指令，该值被保存到栈顶，然后cs：eip的值指向xxx的入口地址
- **进入xxx**
  - 第一条指令： push ebp
  - 第二条指令： mov ebp,esp
  - 函数体中的常规操作，可能会压栈、出栈
- **退出xxx**
  - mov esp,ebp
  - pop ebp
  - ret



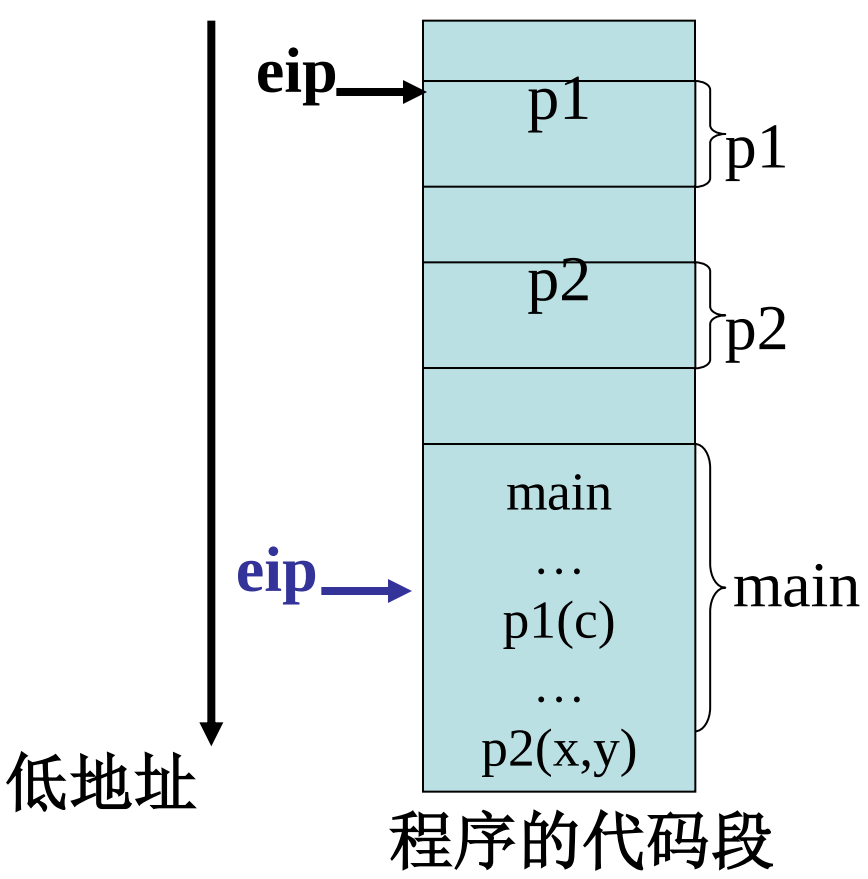
观察程序运行时堆栈的变化(1)

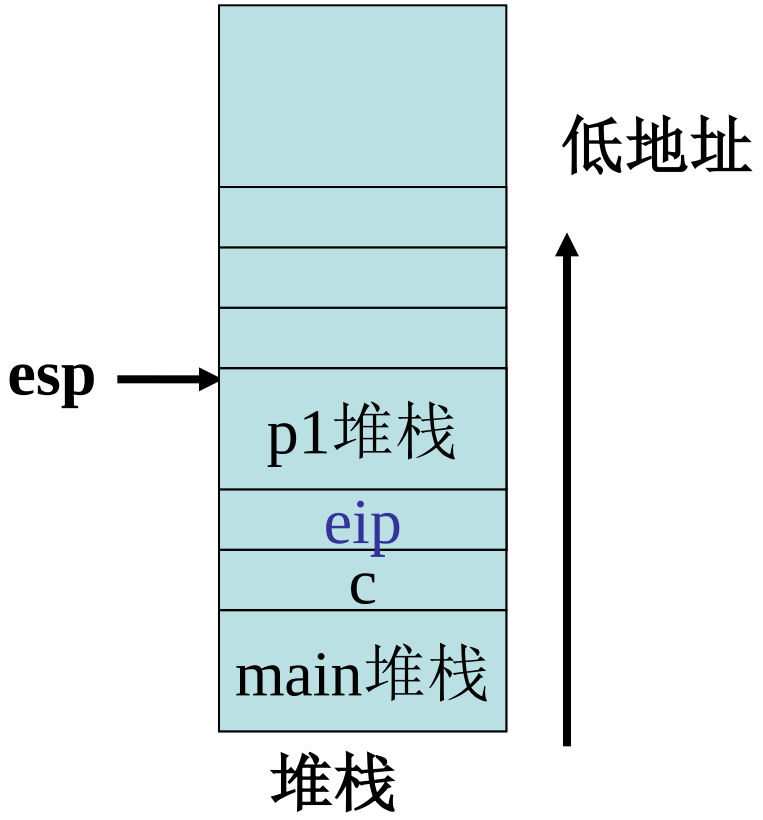


观察程序运行时堆栈的变化(1)

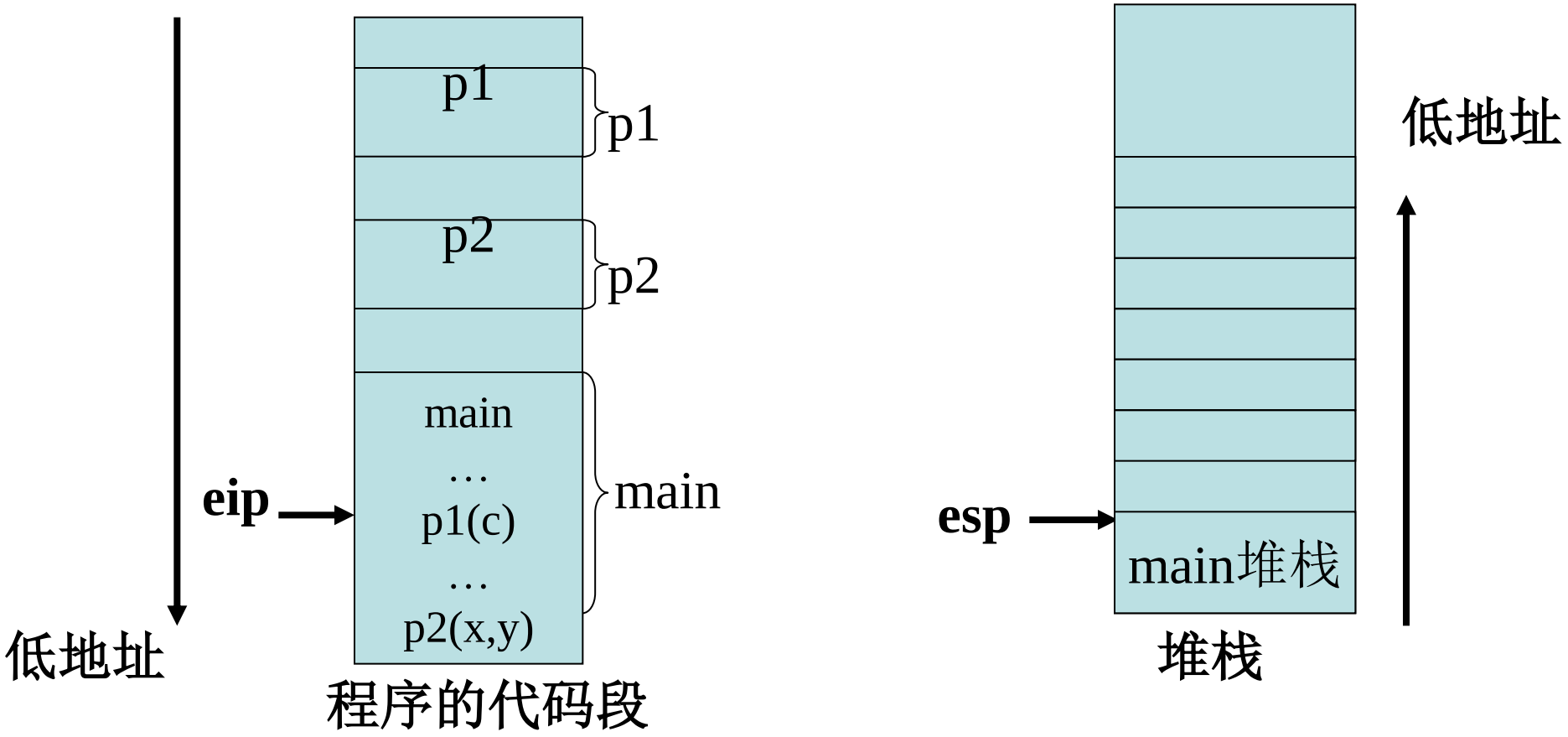


观察程序运行时堆栈的变化(1)

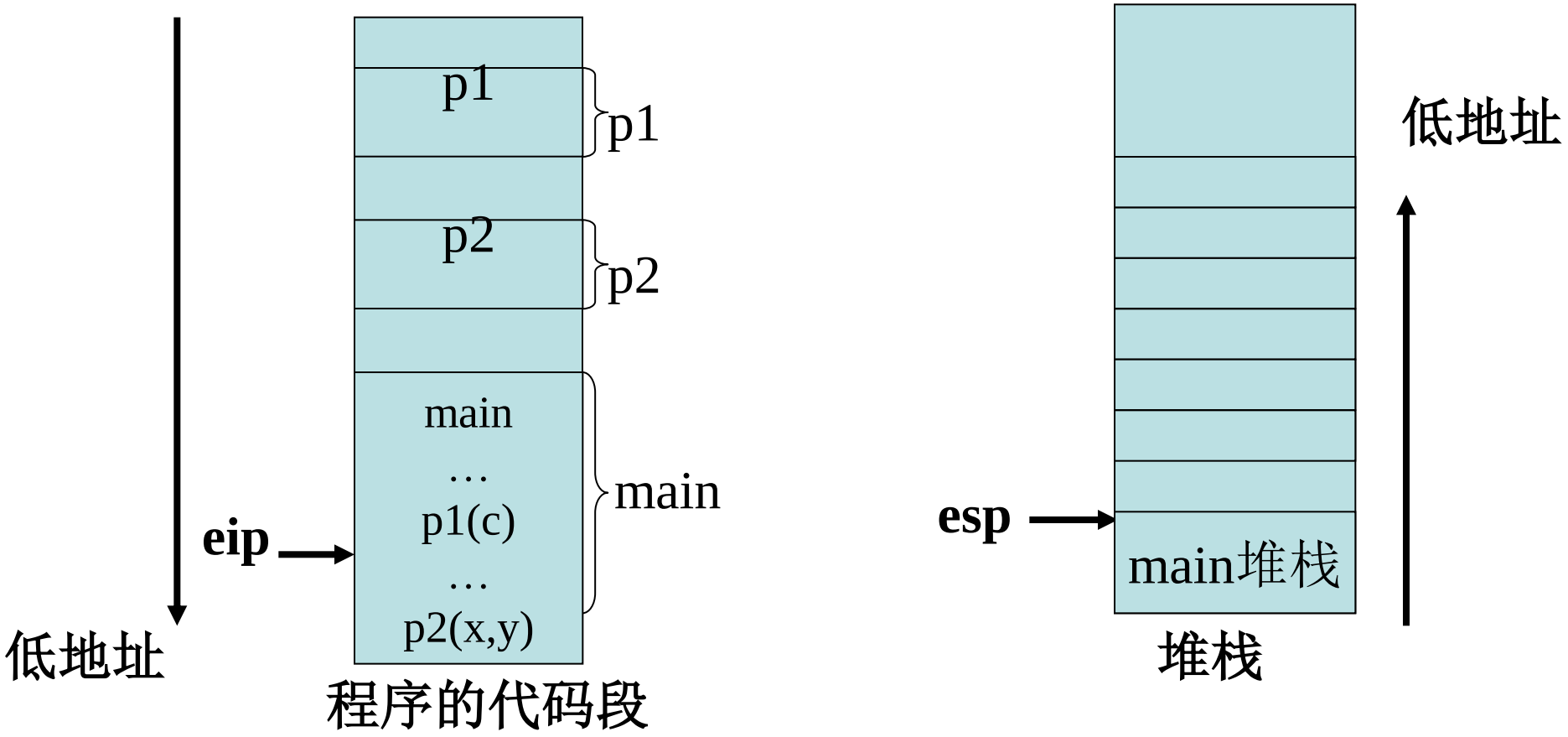




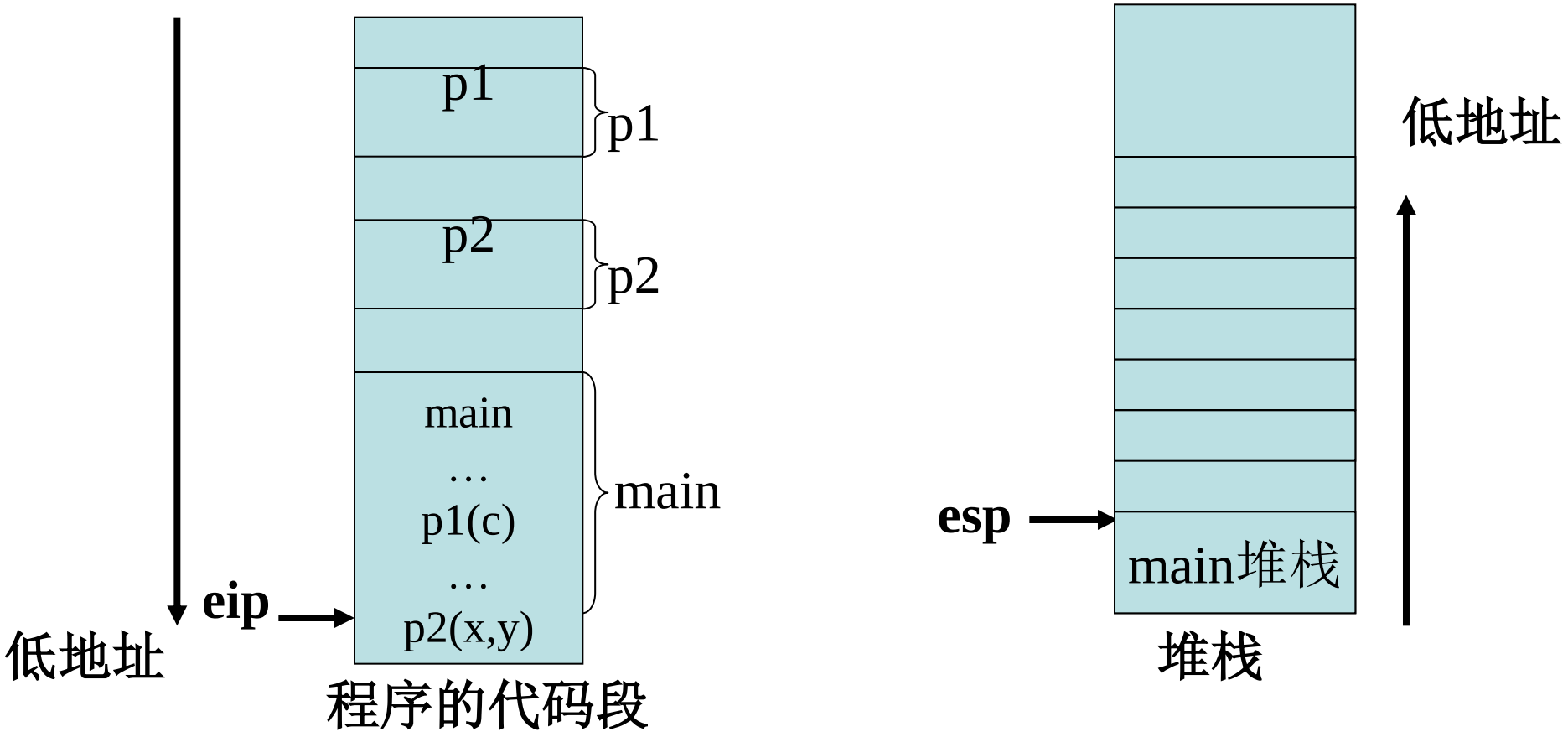
观察程序运行时堆栈的变化(1)



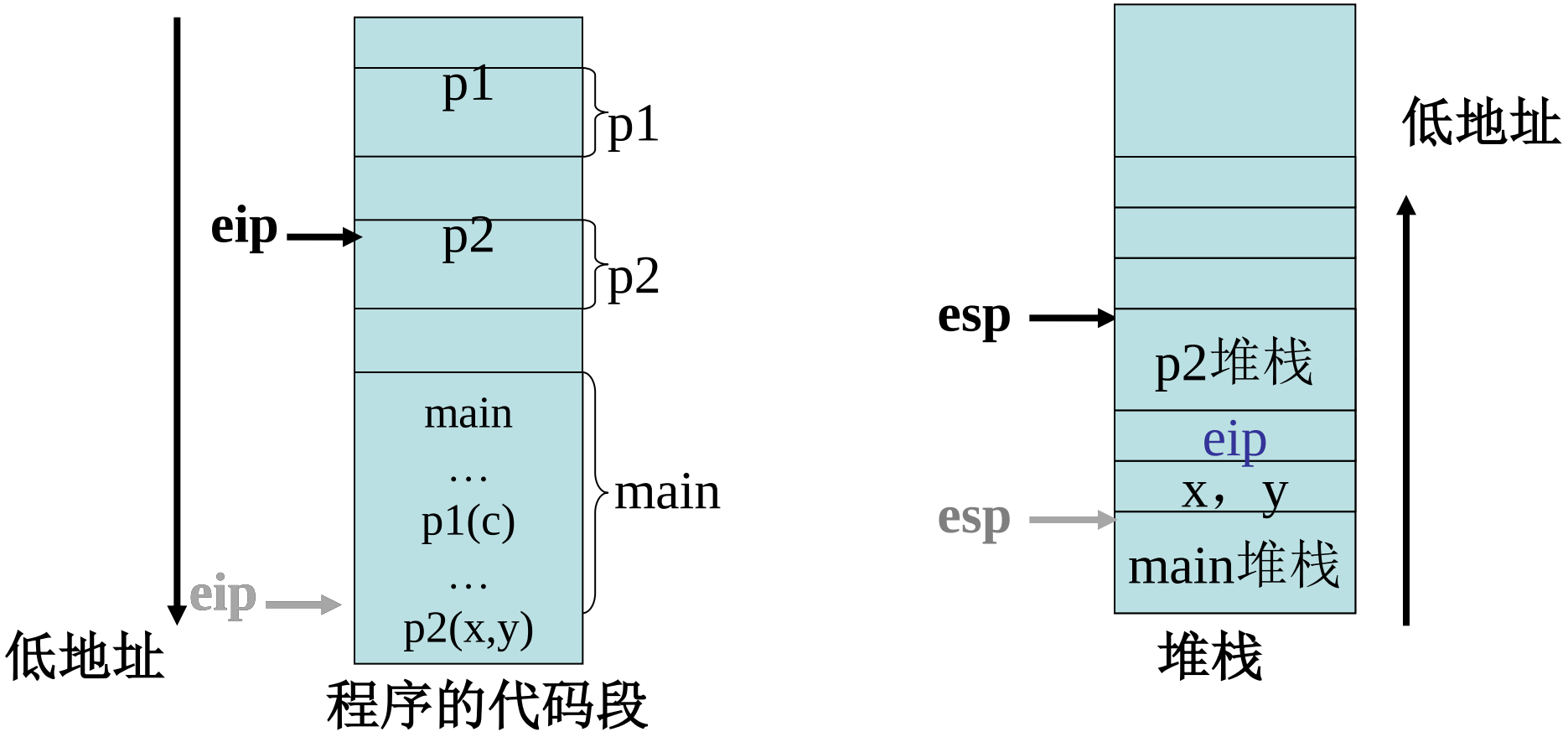
观察程序运行时堆栈的变化(1)



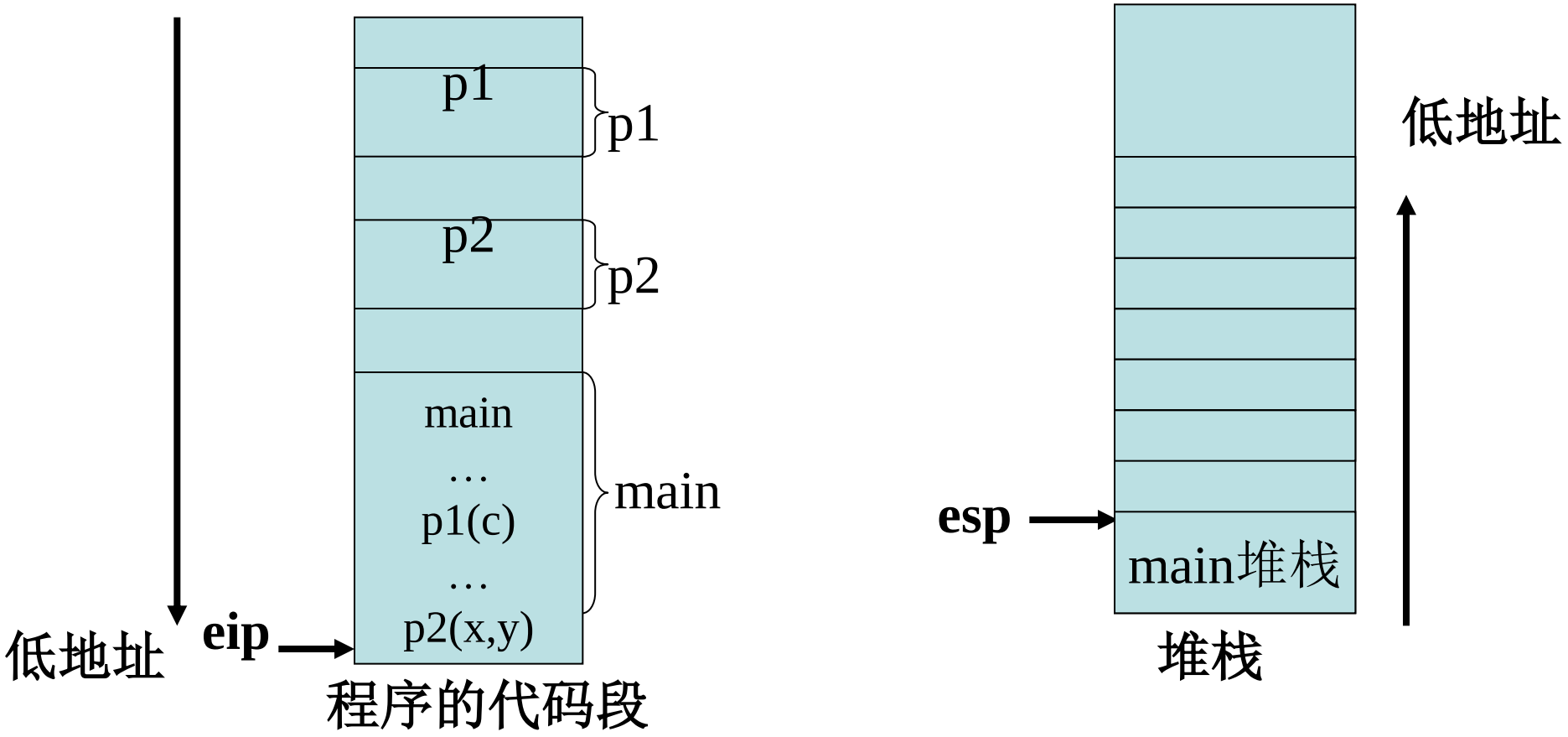
观察程序运行时堆栈的变化(1)



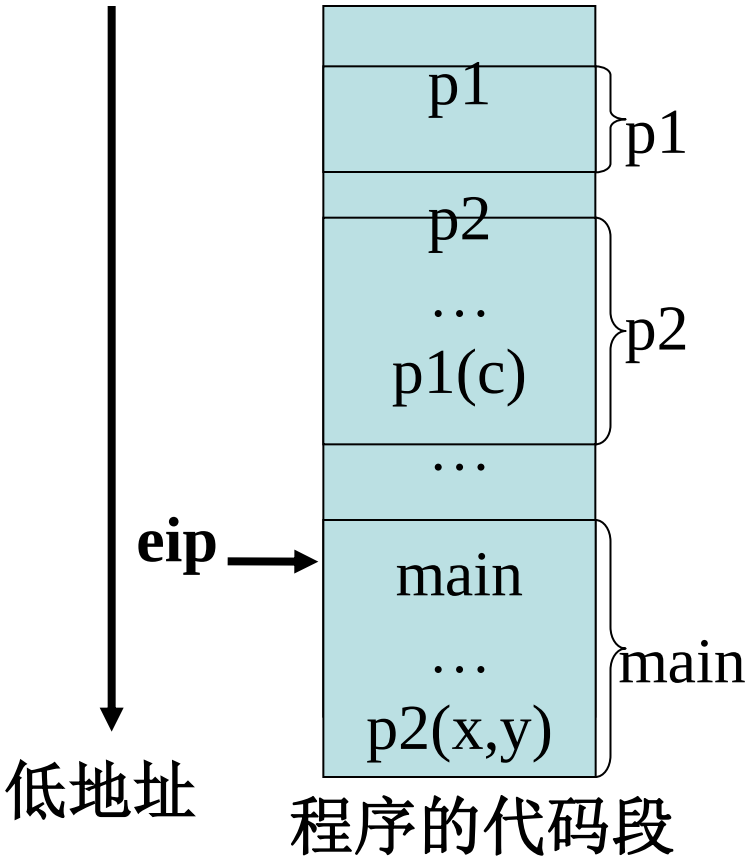
观察程序运行时堆栈的变化(1)

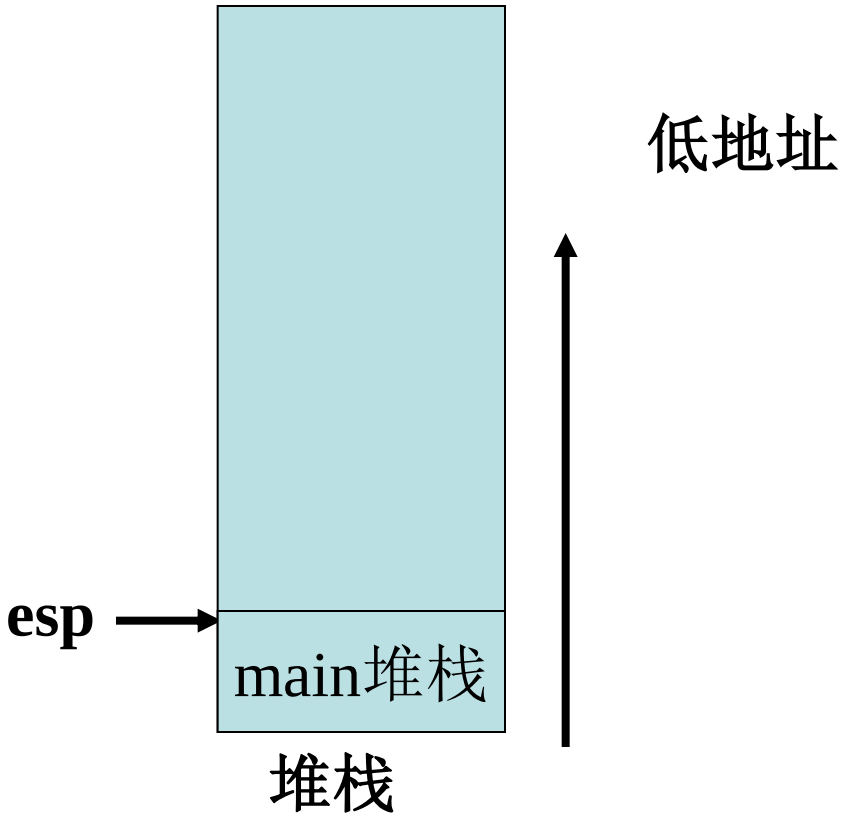


观察程序运行时堆栈的变化(1)

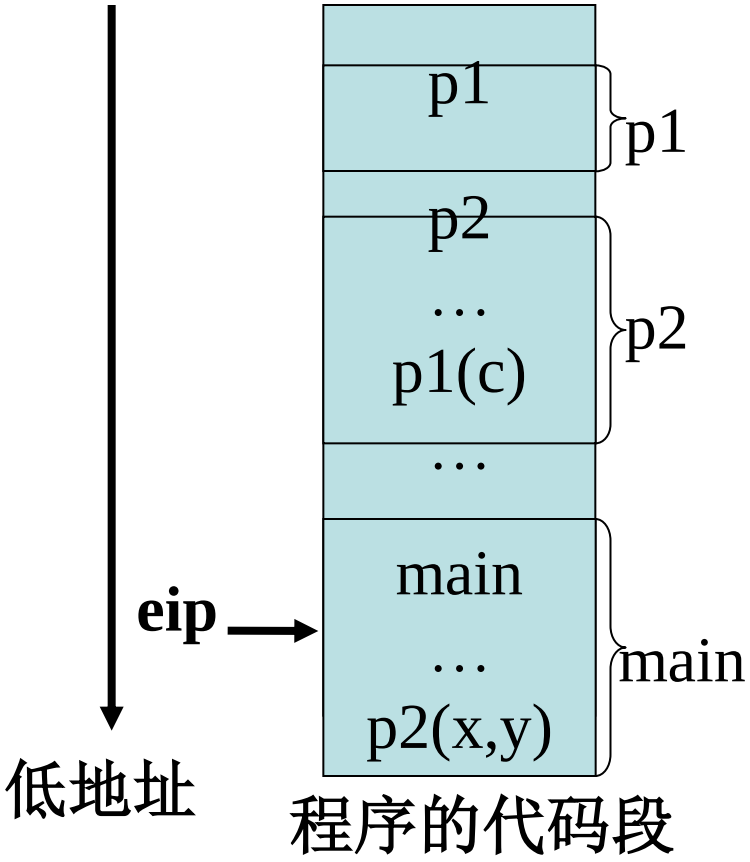


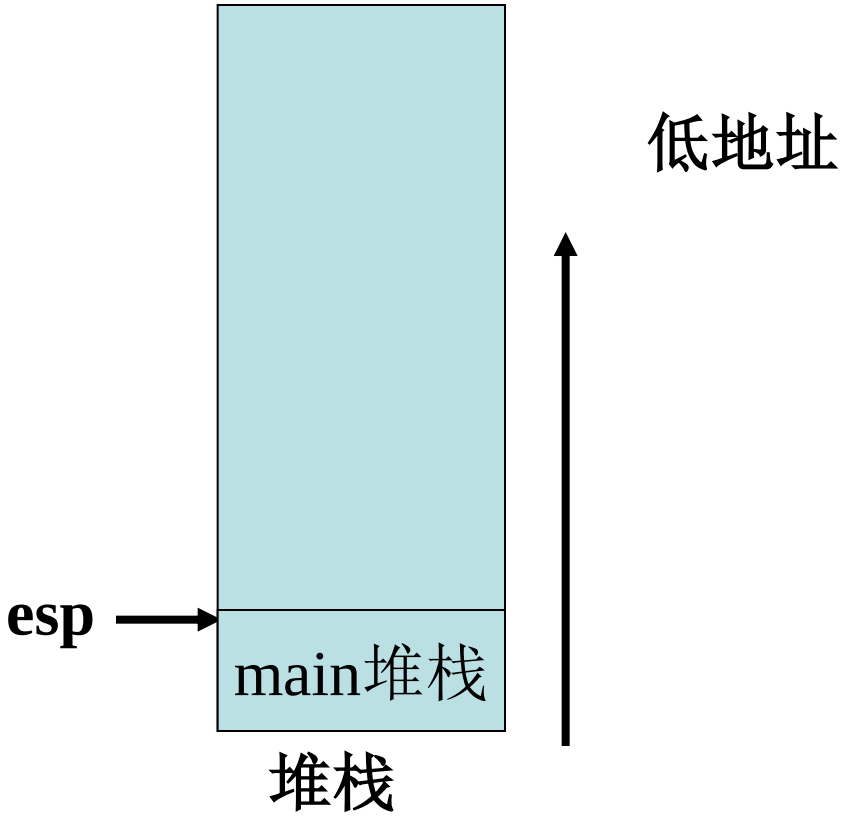
观察程序运行时堆栈的变化(2)



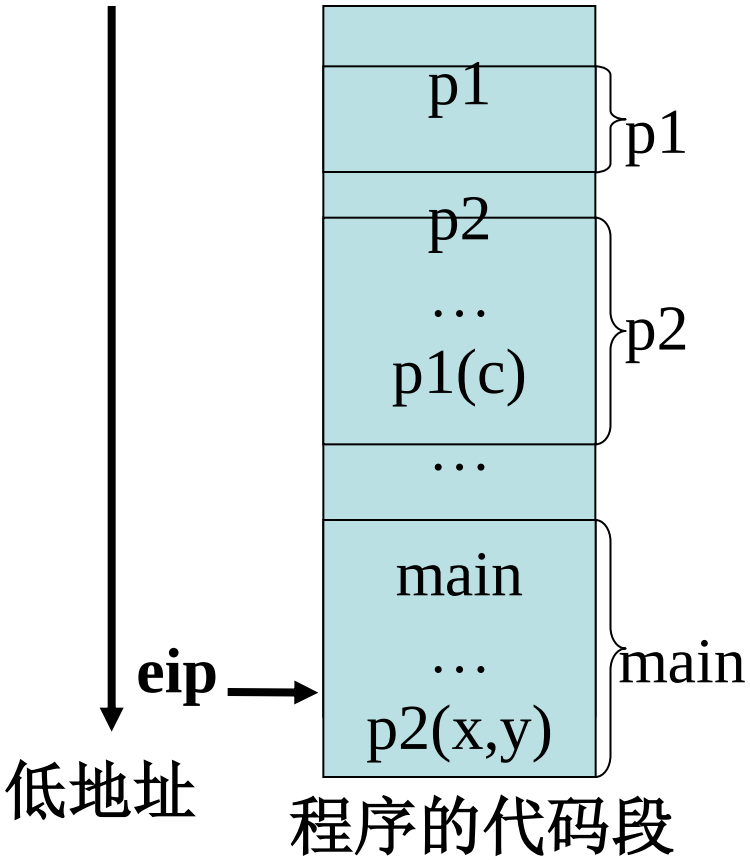


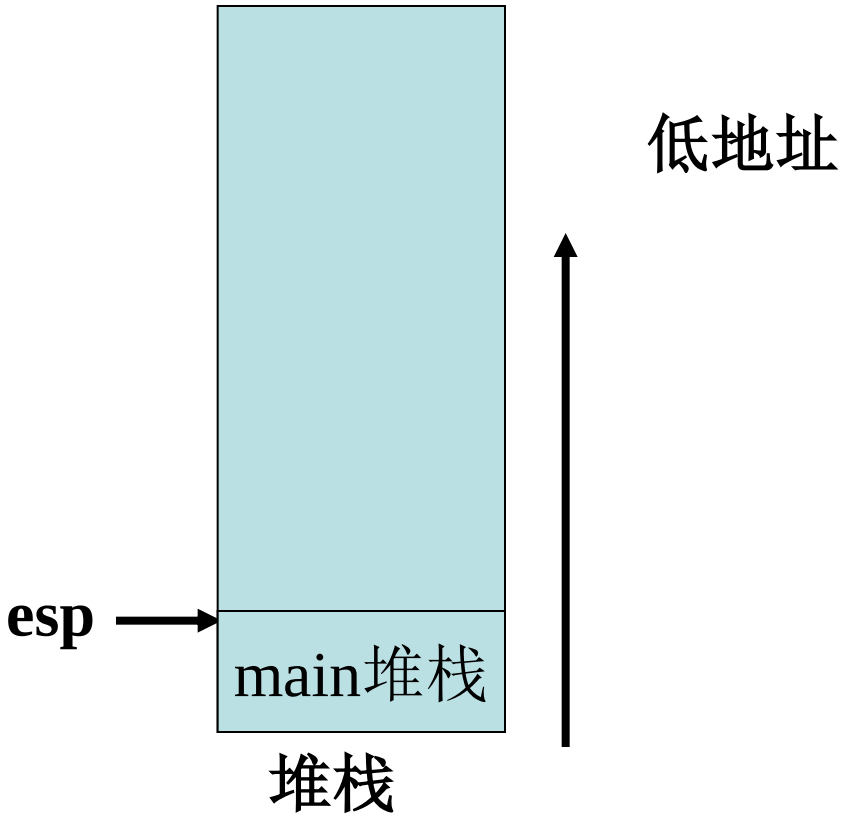
观察程序运行时堆栈的变化(2)



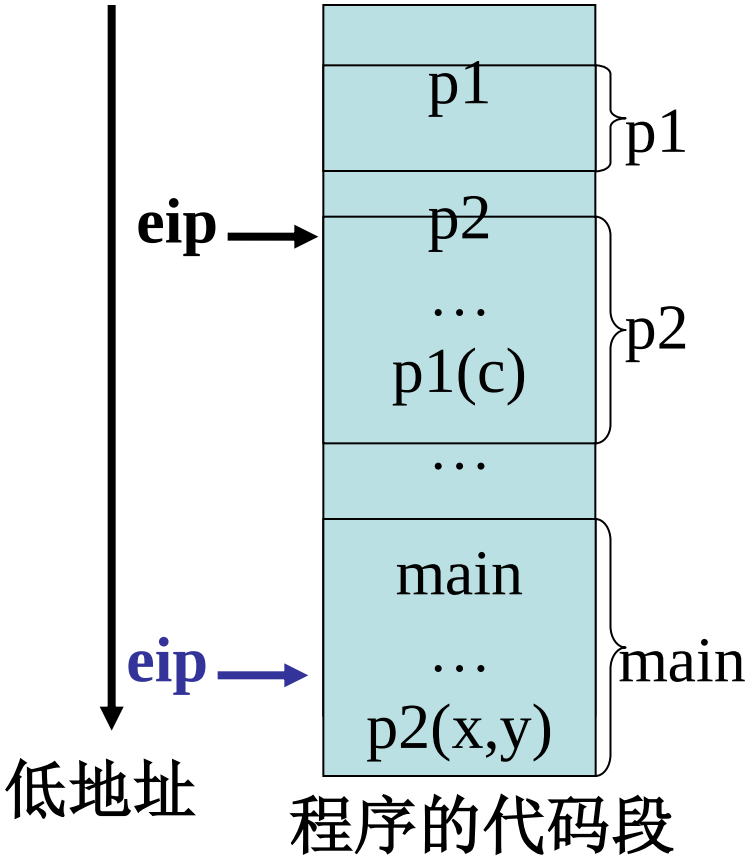


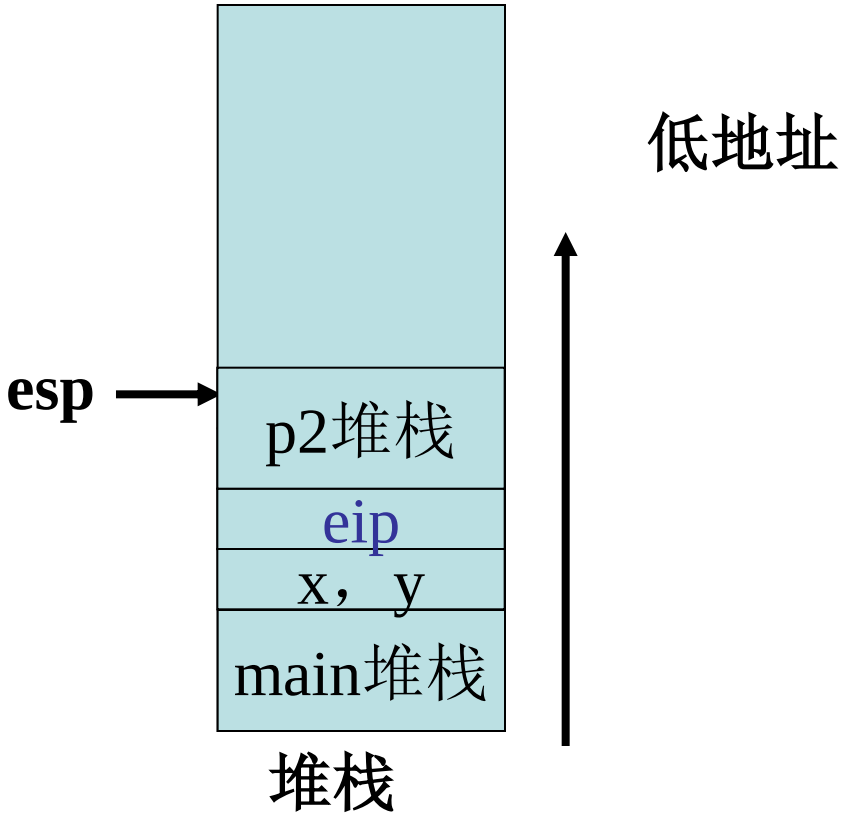
观察程序运行时堆栈的变化(2)



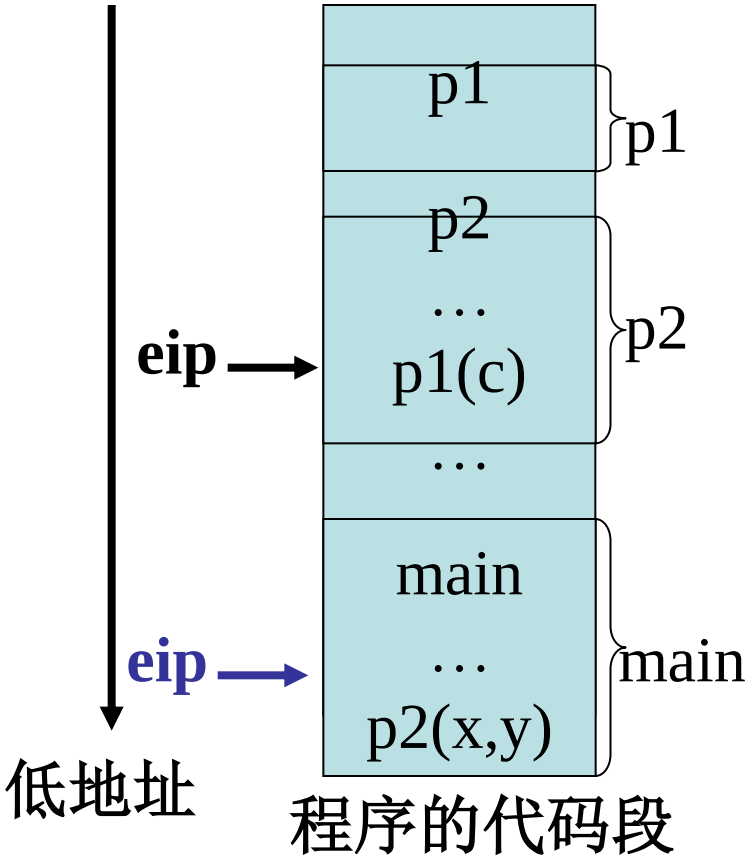


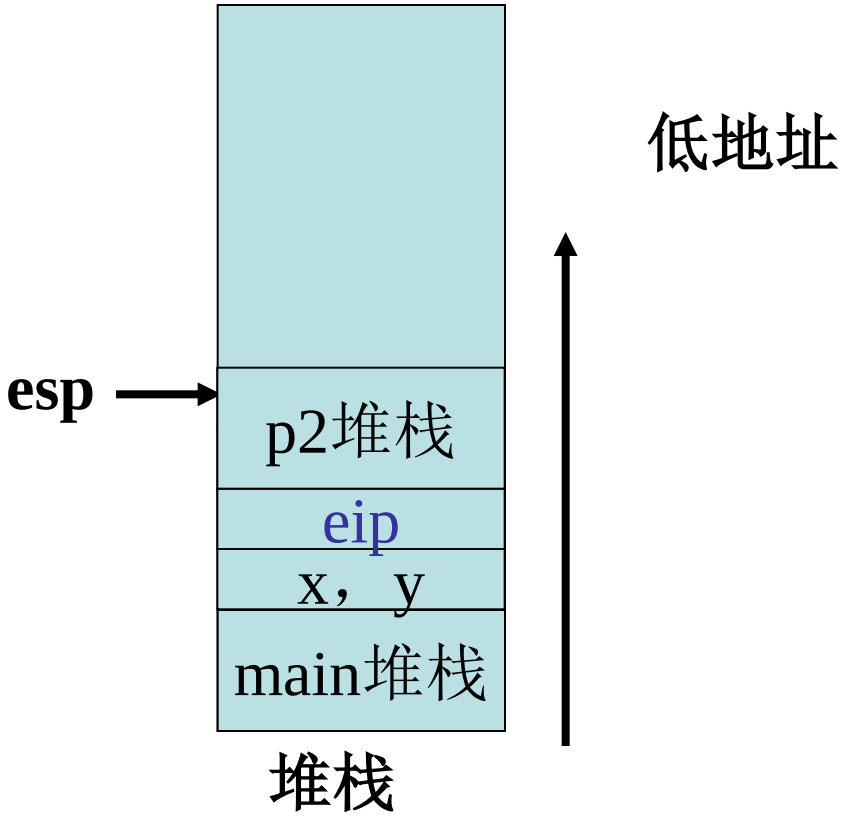
观察程序运行时堆栈的变化(2)



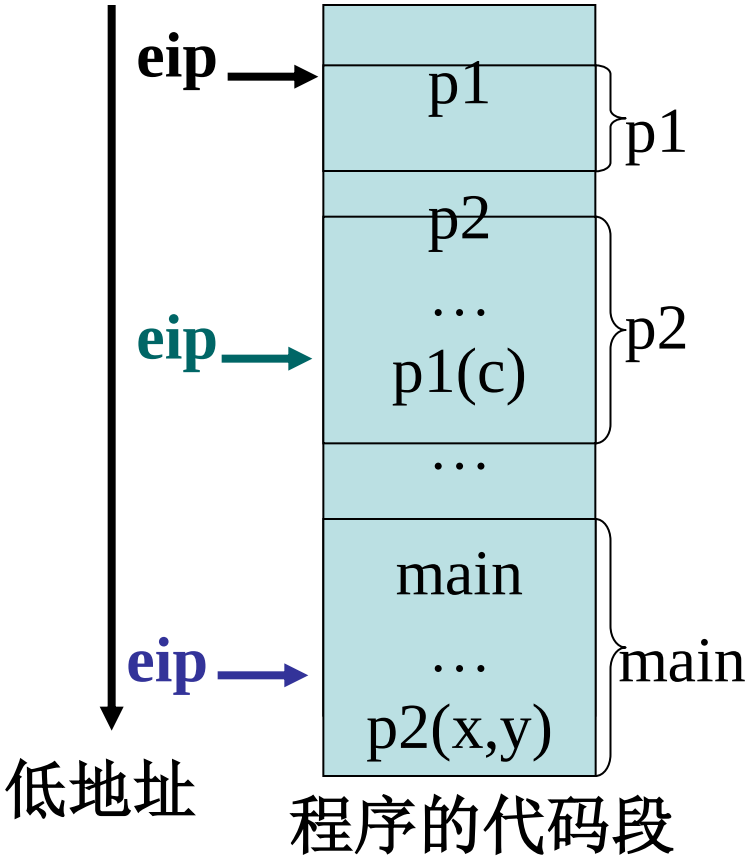


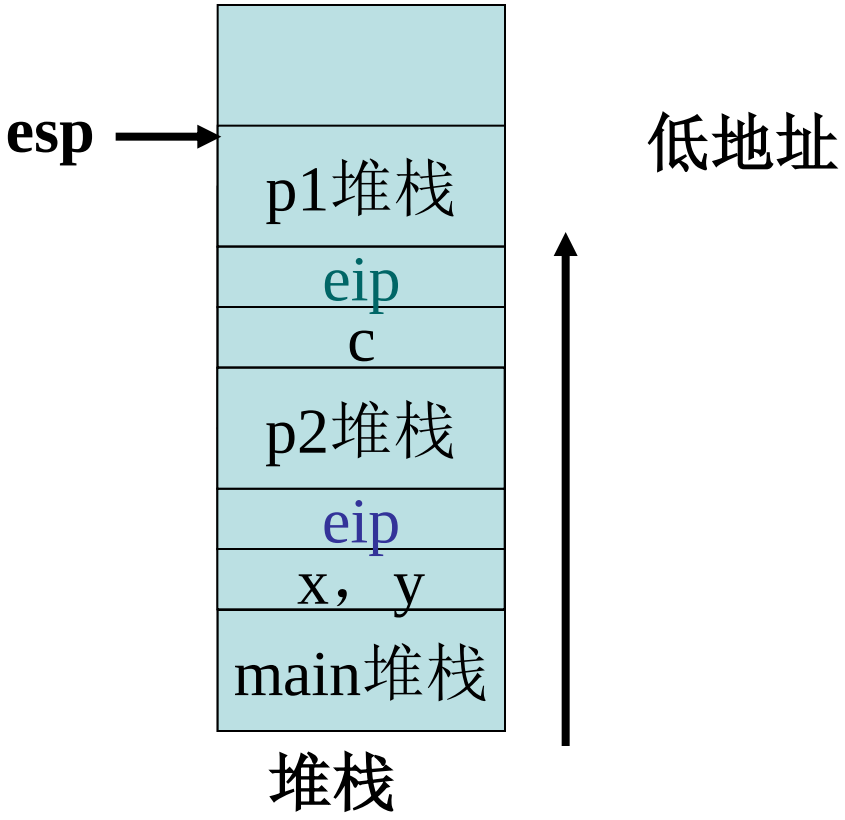
观察程序运行时堆栈的变化(2)



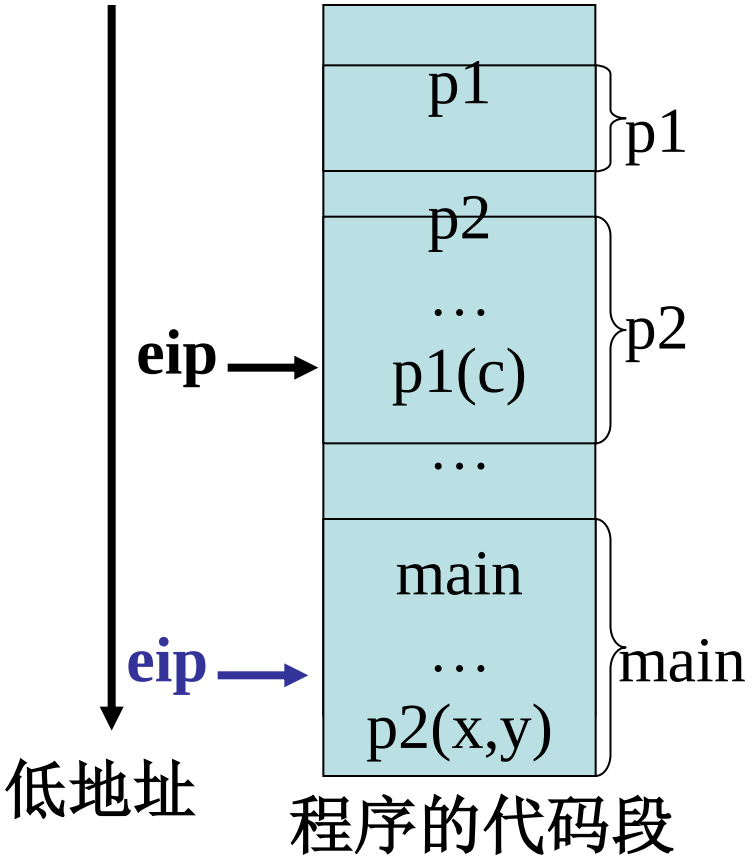


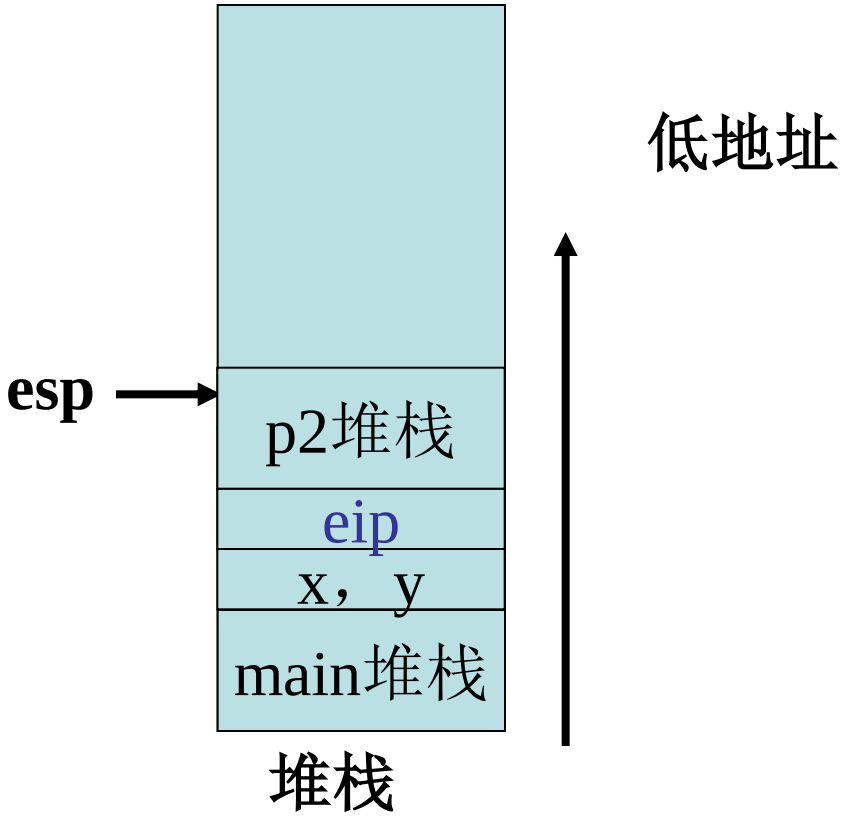
观察程序运行时堆栈的变化(2)



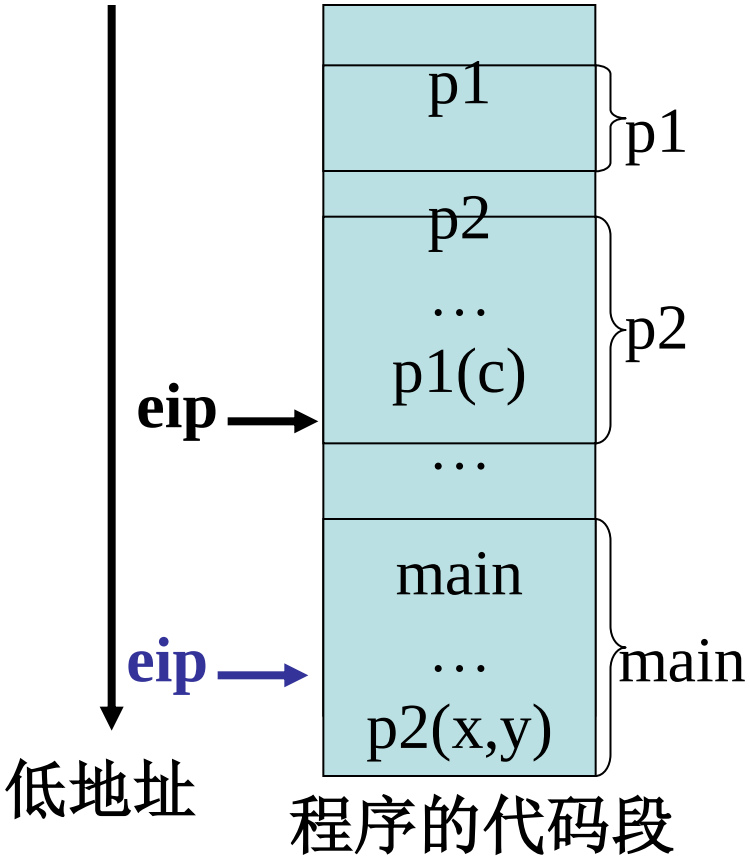


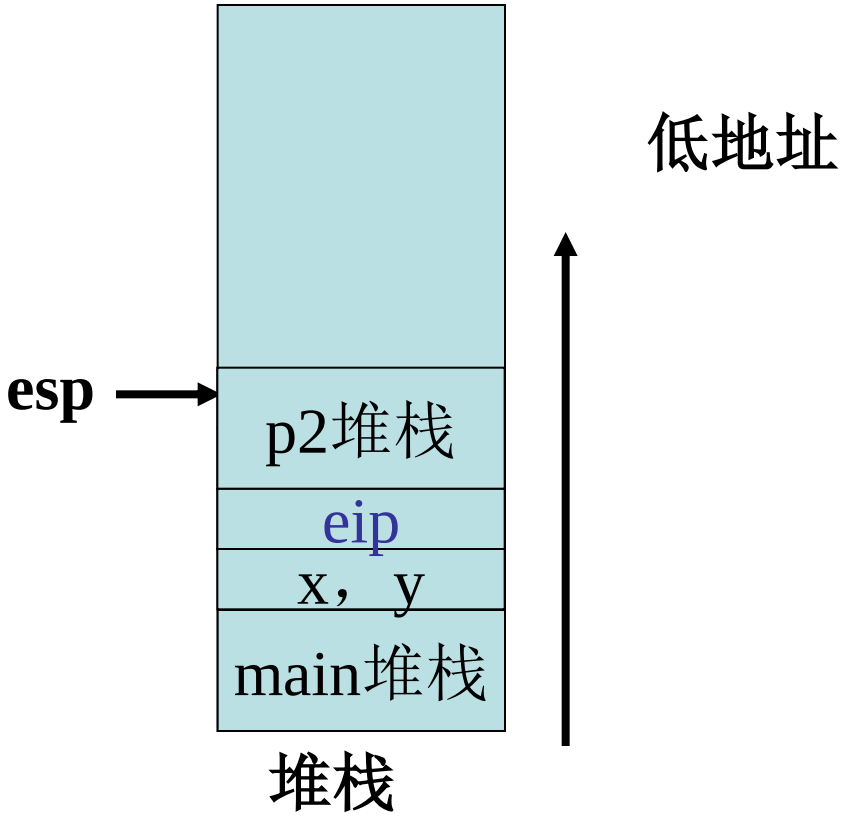
观察程序运行时堆栈的变化(2)



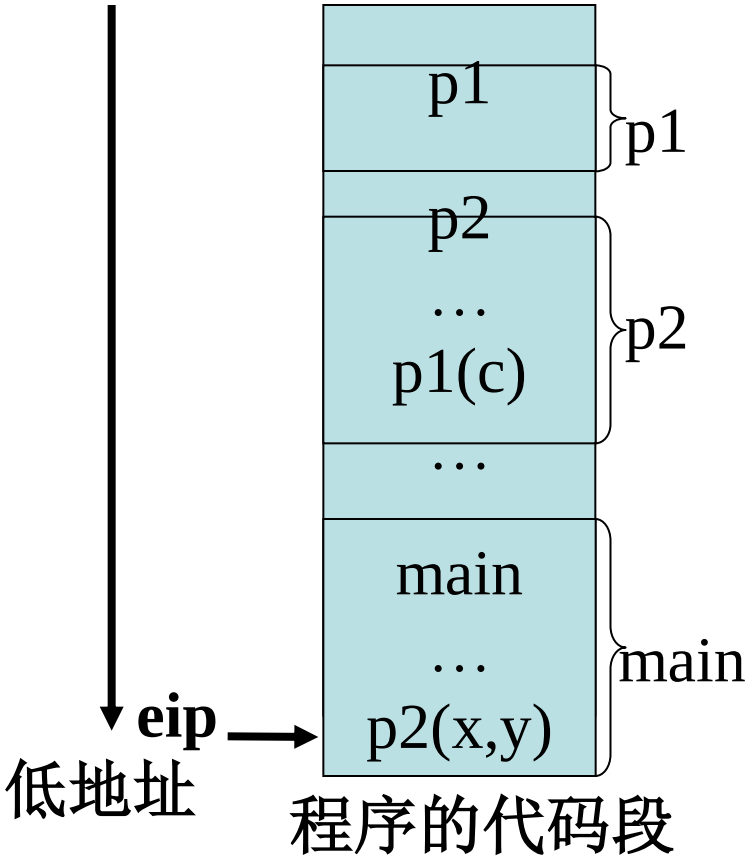


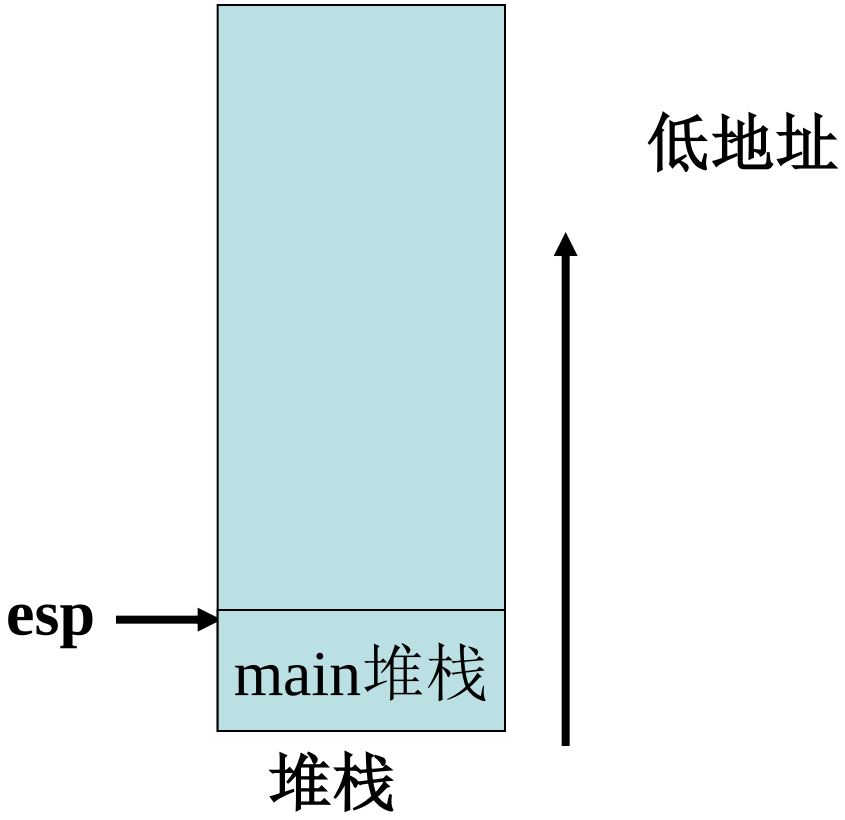
观察程序运行时堆栈的变化(2)





观察程序运行时堆栈的变化(2)

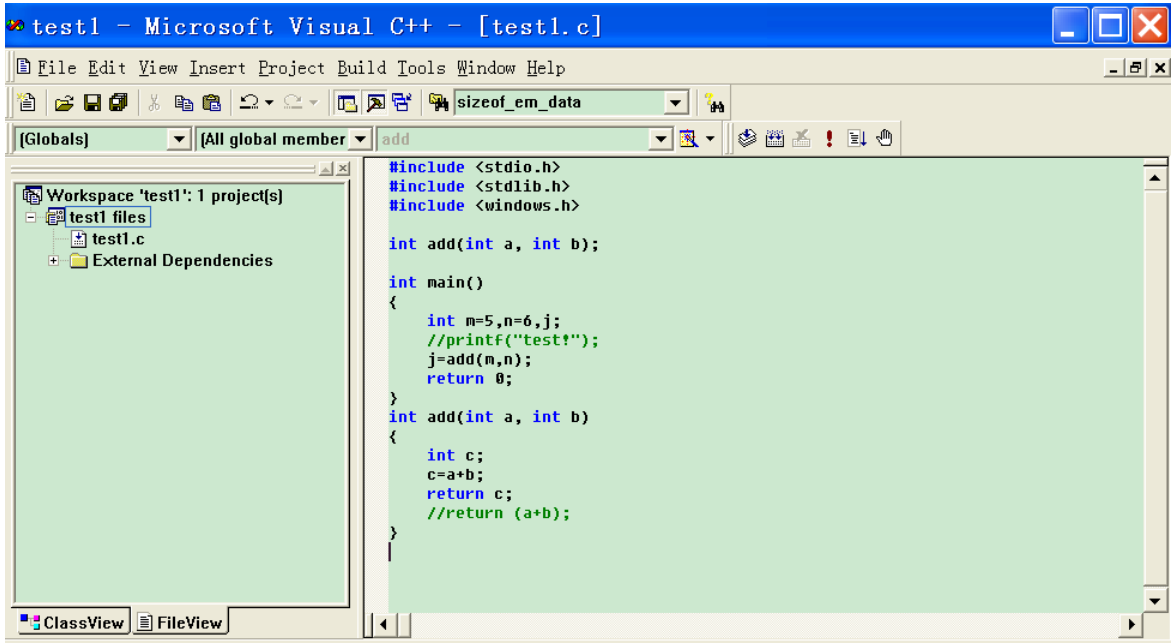




反汇编可执行文件

中国科学技术大学 University of Science and Technology of China

Windows下：利用dumpbin反汇编exe文件



注意，这里 mov指令是 mov src，des 汇编代码风格不同

edi, esi, edx, eax?

linux下：利用objdump反汇编

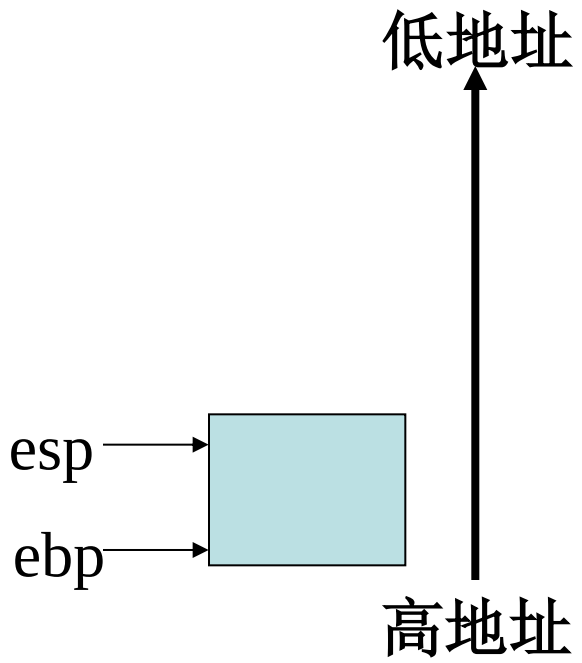
```
test.o:     file format elf64-x86-64

Disassembly of section .text:
0000000000000000 <_Z3addii>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   89 7d ec                mov    %edi,-0x14(%rbp)
   7:   89 75 e8                mov    %esi,-0x18(%rbp)
   a:   8b 55 ec                mov    -0x14(%rbp),%edx
   d:   8b 45 e8                mov    -0x18(%rbp),%eax
  10:   01 d0                   add    %edx,%eax
  12:   89 45 fc                mov    %eax,-0x4(%rbp)
  15:   8b 45 fc                mov    -0x4(%rbp),%eax
  18:   5d                      pop    %rbp
  19:   c3                      retq
```

观察main函数的堆栈框架


```
_main:
  00401020: 55                  push        ebp
  00401021: 8B EC               mov         ebp,esp
  00401023: 83 EC 4C            sub         esp,4Ch
  00401026: 53                  push        ebx
  00401027: 56                  push        esi
  00401028: 57                  push        edi
  00401029: 8D 7D B4            lea         edi,[ebp-4Ch]
  0040102C: B9 13 00 00 00      mov         ecx,13h
  00401031: B8 CC CC CC CC      mov         eax,0CCCCCCCCh
  00401036: F3 AB               rep stos    dword ptr es:[edi]
  00401038: C7 45 FC 05 00 00   mov         dword ptr [ebp-4],5
            00
  0040103F: C7 45 F8 06 00 00   mov         dword ptr [ebp-8],6
            00
  00401046: 8B 45 F8            mov         eax,dword ptr [ebp-8]
  00401049: 50                  push        eax
  0040104A: 8B 4D FC            mov         ecx,dword ptr [ebp-4]
  0040104D: 51                  push        ecx
  0040104E: E8 B2 FF FF FF      call        @ILT+0(_add)
  00401053: 83 C4 08            add         esp,8
  00401056: 89 45 F4            mov         dword ptr [ebp-0Ch],eax
  00401059: 33 C0               xor         eax,eax
  0040105B: 5F                  pop         edi
  0040105C: 5E                  pop         esi
  0040105D: 5B                  pop         ebx
  0040105E: 83 C4 4C            add         esp,4Ch
  00401061: 3B EC               cmp         ebp,esp
  00401063: E8 58 00 00 00      call          chkesp
  00401068: 8B E5               mov         esp,ebp
  0040106A: 5D                  pop         ebp
  0040106B: C3                  ret
```

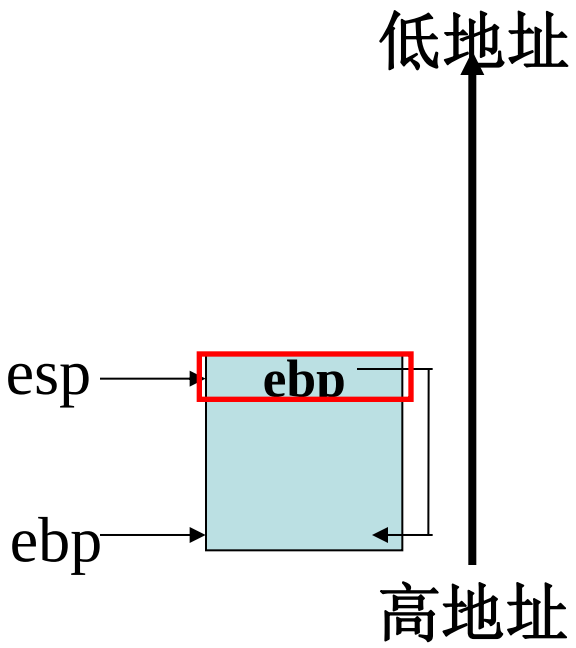
```
int main()
{
    int m=5,n=6,j;
    //printf("test!");
    j=add(m,n);
    return 0;
}
```



观察main函数的堆栈框架


```
_main:
  00401020: 55                  push        ebp
  00401021: 8B EC               mov         ebp,esp
  00401023: 83 EC 4C            sub         esp,4Ch
  00401026: 53                  push        ebx
  00401027: 56                  push        esi
  00401028: 57                  push        edi
  00401029: 8D 7D B4            lea         edi,[ebp-4Ch]
  0040102C: B9 13 00 00 00      mov         ecx,13h
  00401031: B8 CC CC CC CC      mov         eax,0CCCCCCCCh
  00401036: F3 AB               rep stos    dword ptr es:[edi]
  00401038: C7 45 FC 05 00 00   mov         dword ptr [ebp-4],5
            00
  0040103F: C7 45 F8 06 00 00   mov         dword ptr [ebp-8],6
            00
  00401046: 8B 45 F8            mov         eax,dword ptr [ebp-8]
  00401049: 50                  push        eax
  0040104A: 8B 4D FC            mov         ecx,dword ptr [ebp-4]
  0040104D: 51                  push        ecx
  0040104E: E8 B2 FF FF FF      call        @ILT+0(_add)
  00401053: 83 C4 08            add         esp,8
  00401056: 89 45 F4            mov         dword ptr [ebp-0Ch],eax
  00401059: 33 C0               xor         eax,eax
  0040105B: 5F                  pop         edi
  0040105C: 5E                  pop         esi
  0040105D: 5B                  pop         ebx
  0040105E: 83 C4 4C            add         esp,4Ch
  00401061: 3B EC               cmp         ebp,esp
  00401063: E8 58 00 00 00      call          chkesp
  00401068: 8B E5               mov         esp,ebp
  0040106A: 5D                  pop         ebp
  0040106B: C3                  ret
```

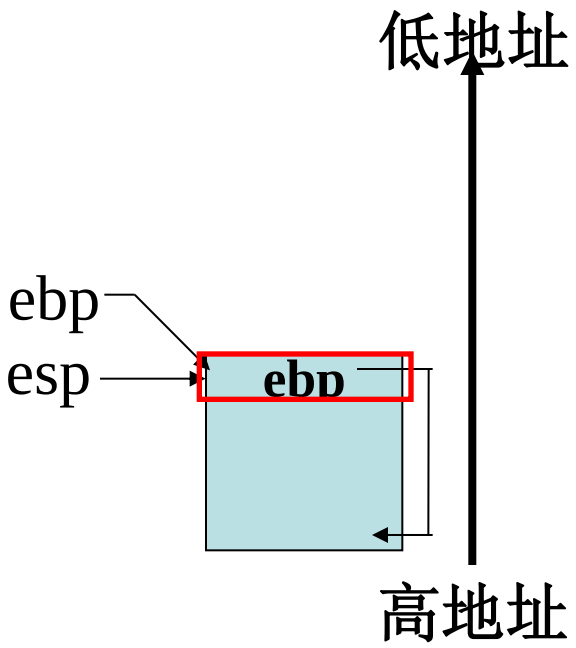
```
int main()
{
    int m=5,n=6,j;
    //printf("test!");
    j=add(m,n);
    return 0;
}
```



观察main函数的堆栈框架


```
_main:
  00401020: 55                 push        ebp
  00401021: 8B EC              mov         ebp,esp
  00401023: 83 EC 4C           sub         esp,4Ch
  00401026: 53                 push        ebx
  00401027: 56                 push        esi
  00401028: 57                 push        edi
  00401029: 8D 7D B4           lea         edi,[ebp-4Ch]
  0040102C: B9 13 00 00 00     mov         ecx,13h
  00401031: B8 CC CC CC CC     mov         eax,0CCCCCCCCh
  00401036: F3 AB              rep stos    dword ptr es:[edi]
  00401038: C7 45 FC 05 00 00  mov         dword ptr [ebp-4],5
            00
  0040103F: C7 45 F8 06 00 00  mov         dword ptr [ebp-8],6
            00
  00401046: 8B 45 F8           mov         eax,dword ptr [ebp-8]
  00401049: 50                 push        eax
  0040104A: 8B 4D FC           mov         ecx,dword ptr [ebp-4]
  0040104D: 51                 push        ecx
  0040104E: E8 B2 FF FF FF     call        @ILT+0(_add)
  00401053: 83 C4 08           add         esp,8
  00401056: 89 45 F4           mov         dword ptr [ebp-0Ch],eax
  00401059: 33 C0              xor         eax,eax
  0040105B: 5F                 pop         edi
  0040105C: 5E                 pop         esi
  0040105D: 5B                 pop         ebx
  0040105E: 83 C4 4C           add         esp,4Ch
  00401061: 3B EC              cmp         ebp,esp
  00401063: E8 58 00 00 00     call          chkesp
  00401068: 8B E5              mov         esp,ebp
  0040106A: 5D                 pop         ebp
  0040106B: C3                 ret
```

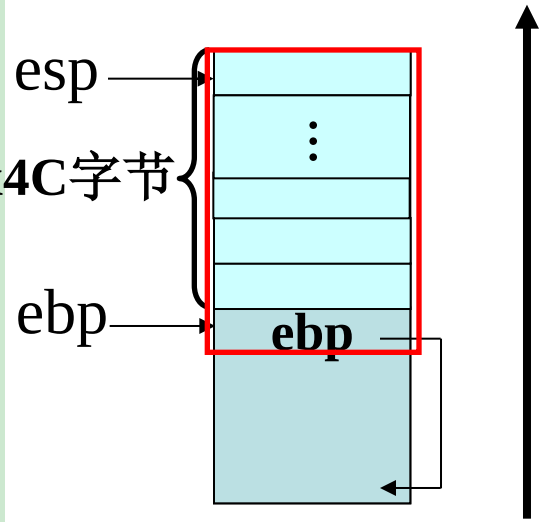
```
int main()
{
    int m=5,n=6,j;
    //printf("test!");
    j=add(m,n);
    return 0;
}
```



观察main函数的堆栈框架


```
_main:
  00401020: 55                  push        ebp
  00401021: 8B EC               mov         ebp,esp
  00401023: 83 EC 4C            sub         esp,4Ch
  00401026: 53                  push        ebx
  00401027: 56                  push        esi
  00401028: 57                  push        edi
  00401029: 8D 7D B4            lea         edi,[ebp-4Ch]
  0040102C: B9 13 00 00 00      mov         ecx,13h
  00401031: B8 CC CC CC CC      mov         eax,0CCCCCCCCh
  00401036: F3 AB               rep stos    dword ptr es:[edi]
  00401038: C7 45 FC 05 00 00   mov         dword ptr [ebp-4],5
            00
  0040103F: C7 45 F8 06 00 00   mov         dword ptr [ebp-8],6
            00
  00401046: 8B 45 F8            mov         eax,dword ptr [ebp-8]
  00401049: 50                  push        eax
  0040104A: 8B 4D FC            mov         ecx,dword ptr [ebp-4]
  0040104D: 51                  push        ecx
  0040104E: E8 B2 FF FF FF      call        @ILT+0(_add)
  00401053: 83 C4 08            add         esp,8
  00401056: 89 45 F4            mov         dword ptr [ebp-0Ch],eax
  00401059: 33 C0               xor         eax,eax
  0040105B: 5F                  pop         edi
  0040105C: 5E                  pop         esi
  0040105D: 5B                  pop         ebx
  0040105E: 83 C4 4C            add         esp,4Ch
  00401061: 3B EC               cmp         ebp,esp
  00401063: E8 58 00 00 00      call          chkesp
  00401068: 8B E5               mov         esp,ebp
  0040106A: 5D                  pop         ebp
  0040106B: C3                  ret
```

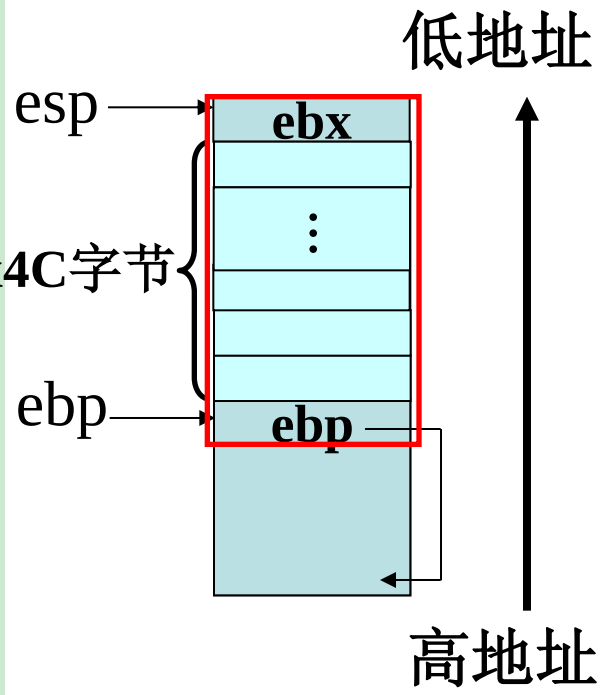
```
int main()
{
    int m=5,n=6,j;
    //printf("test!");
    j=add(m,n);
    return 0;
}
```



观察main函数的堆栈框架


```
_main:
  00401020: 55                  push        ebp
  00401021: 8B EC               mov         ebp,esp
  00401023: 83 EC 4C            sub         esp,4Ch
  00401026: 53                  push        ebx
  00401027: 56                  push        esi
  00401028: 57                  push        edi
  00401029: 8D 7D B4            lea         edi,[ebp-4Ch]
  0040102C: B9 13 00 00 00      mov         ecx,13h
  00401031: B8 CC CC CC CC      mov         eax,0CCCCCCCCh
  00401036: F3 AB               rep stos    dword ptr es:[edi]
  00401038: C7 45 FC 05 00 00   mov         dword ptr [ebp-4],5
            00
  0040103F: C7 45 F8 06 00 00   mov         dword ptr [ebp-8],6
            00
  00401046: 8B 45 F8            mov         eax,dword ptr [ebp-8]
  00401049: 50                  push        eax
  0040104A: 8B 4D FC            mov         ecx,dword ptr [ebp-4]
  0040104D: 51                  push        ecx
  0040104E: E8 B2 FF FF FF      call        @ILT+0(_add)
  00401053: 83 C4 08            add         esp,8
  00401056: 89 45 F4            mov         dword ptr [ebp-0Ch],eax
  00401059: 33 C0               xor         eax,eax
  0040105B: 5F                  pop         edi
  0040105C: 5E                  pop         esi
  0040105D: 5B                  pop         ebx
  0040105E: 83 C4 4C            add         esp,4Ch
  00401061: 3B EC               cmp         ebp,esp
  00401063: E8 58 00 00 00      call          chkesp
  00401068: 8B E5               mov         esp,ebp
  0040106A: 5D                  pop         ebp
  0040106B: C3                  ret
```

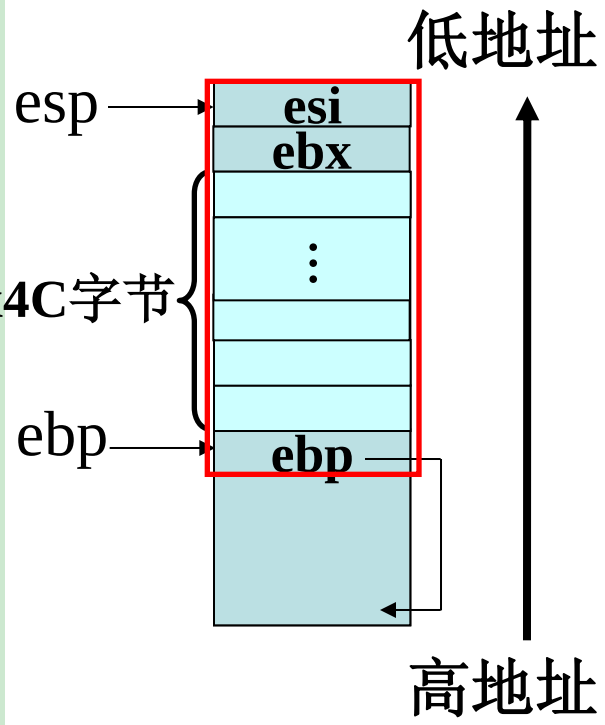
```
int main()
{
    int m=5,n=6,j;
    //printf("test!");
    j=add(m,n);
    return 0;
}
```



观察main函数的堆栈框架


```
_main:
  00401020: 55                  push        ebp
  00401021: 8B EC               mov         ebp,esp
  00401023: 83 EC 4C            sub         esp,4Ch
  00401026: 53                  push        ebx
  00401027: 56                  push        esi
  00401028: 57                  push        edi
  00401029: 8D 7D B4            lea         edi,[ebp-4Ch]
  0040102C: B9 13 00 00 00      mov         ecx,13h
  00401031: B8 CC CC CC CC      mov         eax,0CCCCCCCCh
  00401036: F3 AB               rep stos    dword ptr es:[edi]
  00401038: C7 45 FC 05 00 00   mov         dword ptr [ebp-4],5
            00
  0040103F: C7 45 F8 06 00 00   mov         dword ptr [ebp-8],6
            00
  00401046: 8B 45 F8            mov         eax,dword ptr [ebp-8]
  00401049: 50                  push        eax
  0040104A: 8B 4D FC            mov         ecx,dword ptr [ebp-4]
  0040104D: 51                  push        ecx
  0040104E: E8 B2 FF FF FF      call        @ILT+0(_add)
  00401053: 83 C4 08            add         esp,8
  00401056: 89 45 F4            mov         dword ptr [ebp-0Ch],eax
  00401059: 33 C0               xor         eax,eax
  0040105B: 5F                  pop         edi
  0040105C: 5E                  pop         esi
  0040105D: 5B                  pop         ebx
  0040105E: 83 C4 4C            add         esp,4Ch
  00401061: 3B EC               cmp         ebp,esp
  00401063: E8 58 00 00 00      call          chkesp
  00401068: 8B E5               mov         esp,ebp
  0040106A: 5D                  pop         ebp
  0040106B: C3                  ret
```

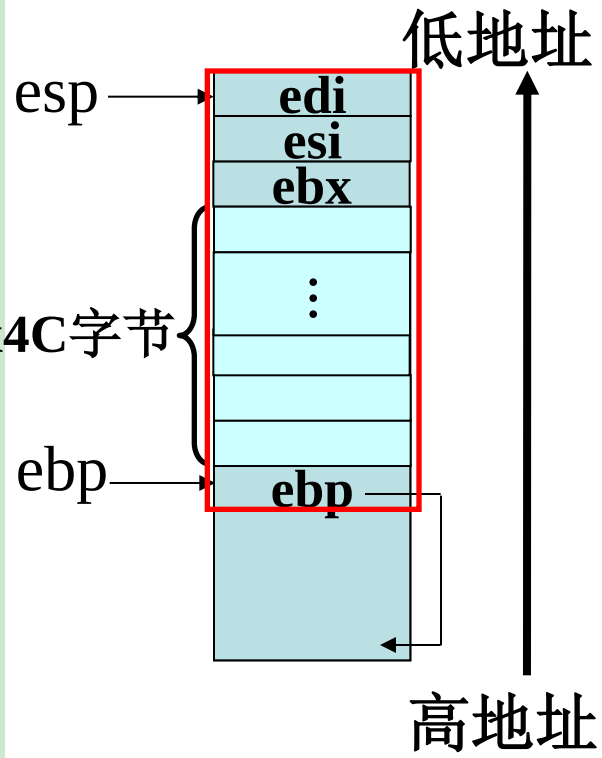
```
int main()
{
    int m=5,n=6,j;
    //printf("test!");
    j=add(m,n);
    return 0;
}
```



观察main函数的堆栈框架


```
_main:
  00401020: 55                  push        ebp
  00401021: 8B EC               mov         ebp,esp
  00401023: 83 EC 4C            sub         esp,4Ch
  00401026: 53                  push        ebx
  00401027: 56                  push        esi
  00401028: 57                  push        edi
  00401029: 8D 7D B4            lea         edi,[ebp-4Ch]
  0040102C: B9 13 00 00 00      mov         ecx,13h
  00401031: B8 CC CC CC CC      mov         eax,0CCCCCCCCh
  00401036: F3 AB               rep stos    dword ptr es:[edi]
  00401038: C7 45 FC 05 00 00   mov         dword ptr [ebp-4],5
            00
  0040103F: C7 45 F8 06 00 00   mov         dword ptr [ebp-8],6
            00
  00401046: 8B 45 F8            mov         eax,dword ptr [ebp-8]
  00401049: 50                  push        eax
  0040104A: 8B 4D FC            mov         ecx,dword ptr [ebp-4]
  0040104D: 51                  push        ecx
  0040104E: E8 B2 FF FF FF      call        @ILT+0(_add)
  00401053: 83 C4 08            add         esp,8
  00401056: 89 45 F4            mov         dword ptr [ebp-0Ch],eax
  00401059: 33 C0               xor         eax,eax
  0040105B: 5F                  pop         edi
  0040105C: 5E                  pop         esi
  0040105D: 5B                  pop         ebx
  0040105E: 83 C4 4C            add         esp,4Ch
  00401061: 3B EC               cmp         ebp,esp
  00401063: E8 58 00 00 00      call          chkesp
  00401068: 8B E5               mov         esp,ebp
  0040106A: 5D                  pop         ebp
  0040106B: C3                  ret
```

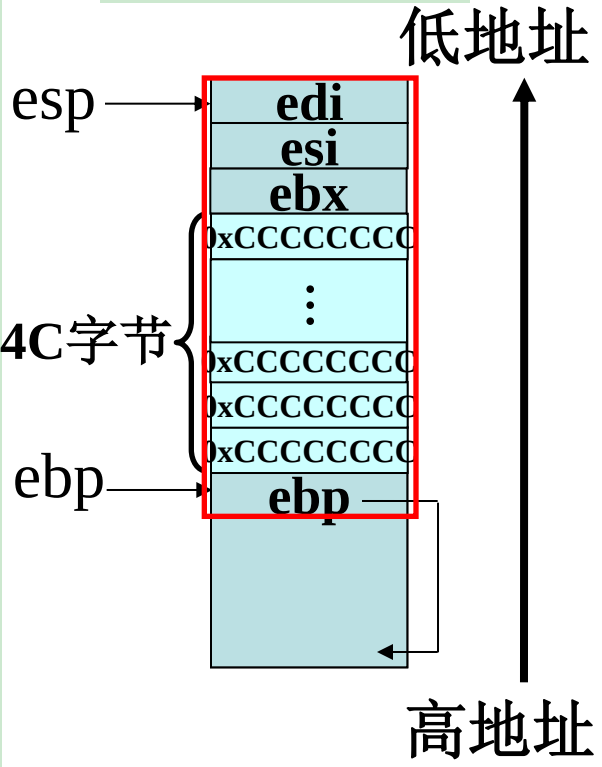
```
int main()
{
    int m=5,n=6,j;
    //printf("test!");
    j=add(m,n);
    return 0;
}
```



观察main函数的堆栈框架


```
_main:
  00401020: 55                  push        ebp
  00401021: 8B EC               mov         ebp,esp
  00401023: 83 EC 4C            sub         esp,4Ch
  00401026: 53                  push        ebx
  00401027: 56                  push        esi
  00401028: 57                  push        edi
  00401029: 8D 7D B4            lea         edi,[ebp-4Ch]
  0040102C: B9 13 00 00 00      mov         ecx,13h
  00401031: B8 CC CC CC CC      mov         eax,0CCCCCCCCh
  00401036: F3 AB               rep stos    dword ptr es:[edi]
  00401038: C7 45 FC 05 00 00   mov         dword ptr [ebp-4],5
            00
  0040103F: C7 45 F8 06 00 00   mov         dword ptr [ebp-8],6
            00
  00401046: 8B 45 F8            mov         eax,dword ptr [ebp-8]
  00401049: 50                  push        eax
  0040104A: 8B 4D FC            mov         ecx,dword ptr [ebp-4]
  0040104D: 51                  push        ecx
  0040104E: E8 B2 FF FF FF      call        @ILT+0(_add)
  00401053: 83 C4 08            add         esp,8
  00401056: 89 45 F4            mov         dword ptr [ebp-0Ch],eax
  00401059: 33 C0               xor         eax,eax
  0040105B: 5F                  pop         edi
  0040105C: 5E                  pop         esi
  0040105D: 5B                  pop         ebx
  0040105E: 83 C4 4C            add         esp,4Ch
  00401061: 3B EC               cmp         ebp,esp
  00401063: E8 58 00 00 00      call          chkesp
  00401068: 8B E5               mov         esp,ebp
  0040106A: 5D                  pop         ebp
  0040106B: C3                  ret
```

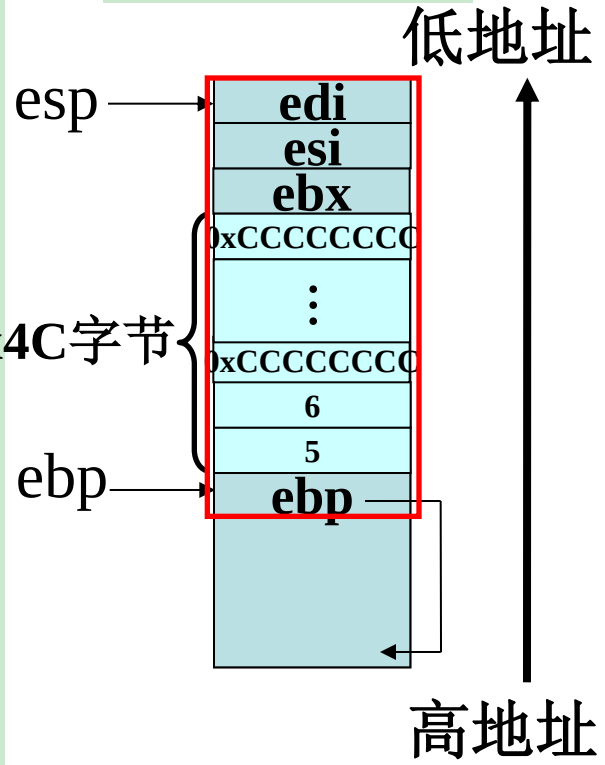
```
int main()
{
    int m=5,n=6,j;
    //printf("test!");
    j=add(m,n);
    return 0;
}
```



观察main函数的堆栈框架


```
_main:
  00401020: 55                  push        ebp
  00401021: 8B EC               mov         ebp,esp
  00401023: 83 EC 4C            sub         esp,4Ch
  00401026: 53                  push        ebx
  00401027: 56                  push        esi
  00401028: 57                  push        edi
  00401029: 8D 7D B4            lea         edi,[ebp-4Ch]
  0040102C: B9 13 00 00 00      mov         ecx,13h
  00401031: B8 CC CC CC CC      mov         eax,0CCCCCCCCh
  00401036: F3 AB               rep stos    dword ptr es:[edi]
  00401038: C7 45 FC 05 00 00   mov         dword ptr [ebp-4],5
            00
  0040103F: C7 45 F8 06 00 00   mov         dword ptr [ebp-8],6
            00
  00401046: 8B 45 F8            mov         eax,dword ptr [ebp-8]
  00401049: 50                  push        eax
  0040104A: 8B 4D FC            mov         ecx,dword ptr [ebp-4]
  0040104D: 51                  push        ecx
  0040104E: E8 B2 FF FF FF      call        @ILT+0(_add)
  00401053: 83 C4 08            add         esp,8
  00401056: 89 45 F4            mov         dword ptr [ebp-0Ch],eax
  00401059: 33 C0               xor         eax,eax
  0040105B: 5F                  pop         edi
  0040105C: 5E                  pop         esi
  0040105D: 5B                  pop         ebx
  0040105E: 83 C4 4C            add         esp,4Ch
  00401061: 3B EC               cmp         ebp,esp
  00401063: E8 58 00 00 00      call          chkesp
  00401068: 8B E5               mov         esp,ebp
  0040106A: 5D                  pop         ebp
  0040106B: C3                  ret
```

```
int main()
{
    int m=5,n=6,j;
    //printf("test!");
    j=add(m,n);
    return 0;
}
```

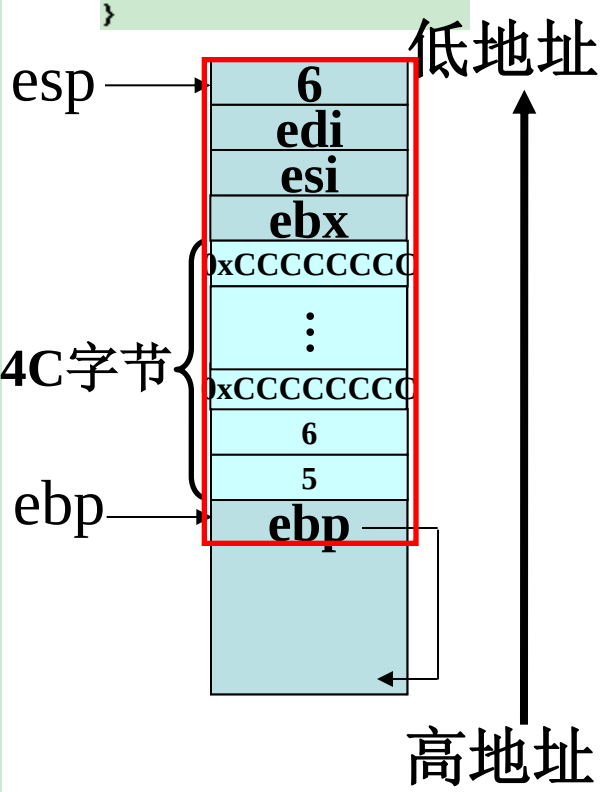


将temp m,n压栈。
这里5,6从哪来的?

观察main函数的堆栈框架

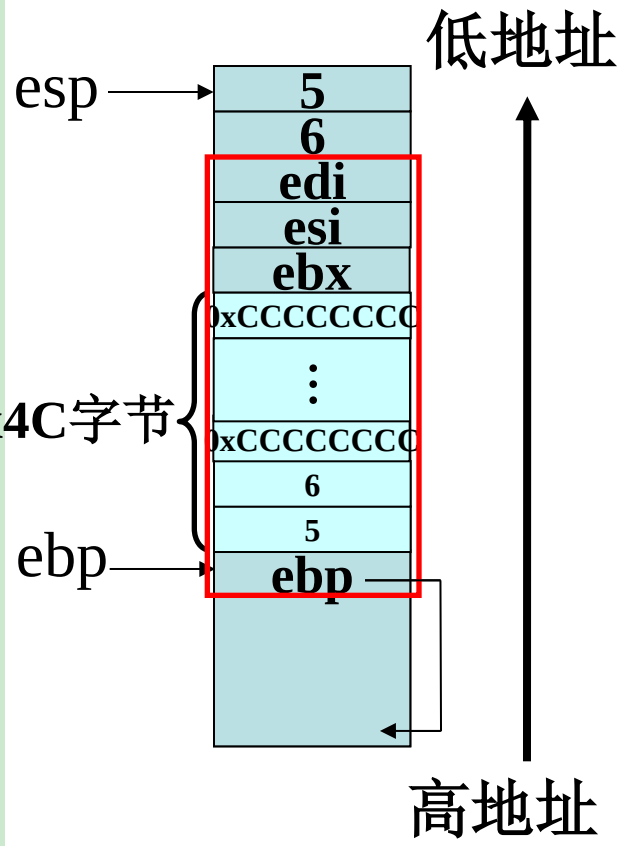

```
_main:
  00401020: 55                  push        ebp
  00401021: 8B EC               mov         ebp,esp
  00401023: 83 EC 4C            sub         esp,4Ch
  00401026: 53                  push        ebx
  00401027: 56                  push        esi
  00401028: 57                  push        edi
  00401029: 8D 7D B4            lea         edi,[ebp-4Ch]
  0040102C: B9 13 00 00 00      mov         ecx,13h
  00401031: B8 CC CC CC CC      mov         eax,0CCCCCCCCh
  00401036: F3 AB               rep stos    dword ptr es:[edi]
  00401038: C7 45 FC 05 00 00   mov         dword ptr [ebp-4],5
            00
  0040103F: C7 45 F8 06 00 00   mov         dword ptr [ebp-8],6
            00
  00401046: 8B 45 F8            mov         eax,dword ptr [ebp-8]
  00401049: 50                  push        eax
  0040104A: 8B 4D FC            mov         ecx,dword ptr [ebp-4]
  0040104D: 51                  push        ecx
  0040104E: E8 B2 FF FF FF      call        @ILT+0(_add)
  00401053: 83 C4 08            add         esp,8
  00401056: 89 45 F4            mov         dword ptr [ebp-0Ch],eax
  00401059: 33 C0               xor         eax,eax
  0040105B: 5F                  pop         edi
  0040105C: 5E                  pop         esi
  0040105D: 5B                  pop         ebx
  0040105E: 83 C4 4C            add         esp,4Ch
  00401061: 3B EC               cmp         ebp,esp
  00401063: E8 58 00 00 00      call          chkesp
  00401068: 8B E5               mov         esp,ebp
  0040106A: 5D                  pop         ebp
  0040106B: C3                  ret
```

```
int main()
{
    int m=5,n=6,j;
    //printf("test!");
    j=add(m,n);
    return 0;
}
```



观察main函数的堆栈框架

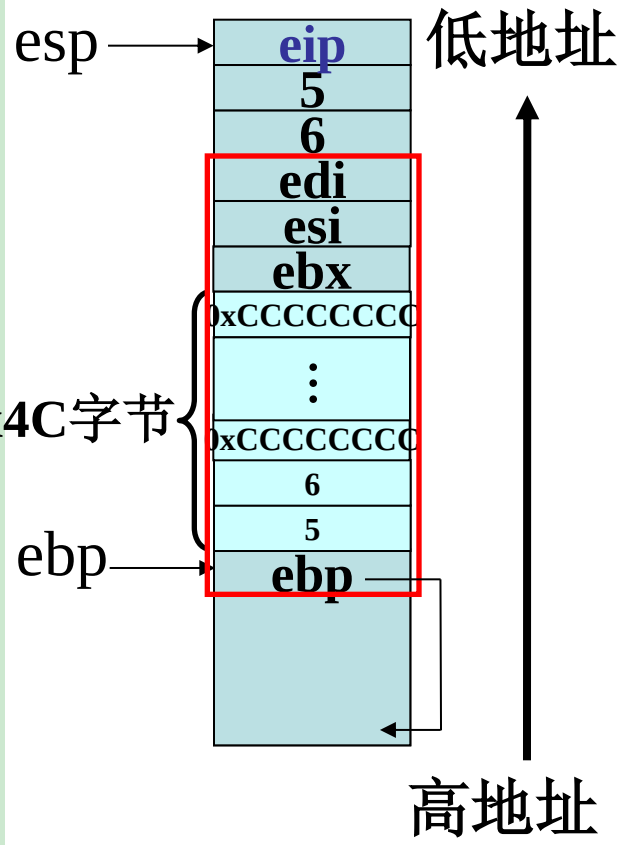

```
_main:
00401020: 55                 push        ebp
00401021: 8B EC              mov         ebp,esp
00401023: 83 EC 4C           sub         esp,4Ch
00401026: 53                 push        ebx
00401027: 56                 push        esi
00401028: 57                 push        edi
00401029: 8D 7D B4           lea         edi,[ebp-4Ch]
0040102C: B9 13 00 00 00     mov         ecx,13h
00401031: B8 CC CC CC CC     mov         eax,0CCCCCCCCh
00401036: F3 AB              rep stos    dword ptr es:[edi]
00401038: C7 45 FC 05 00 00  mov         dword ptr [ebp-4],5
          00
0040103F: C7 45 F8 06 00 00  mov         dword ptr [ebp-8],6
          00
00401046: 8B 45 F8           mov         eax,dword ptr [ebp-8]
00401049: 50                 push        eax
0040104A: 8B 4D FC           mov         ecx,dword ptr [ebp-4]
0040104D: 51                 push        ecx
0040104E: E8 B2 FF FF FF     call        @ILT+0(_add)
00401053: 83 C4 08           add         esp,8
00401056: 89 45 F4           mov         dword ptr [ebp-0Ch],eax
00401059: 33 C0              xor         eax,eax
0040105B: 5F                 pop         edi
0040105C: 5E                 pop         esi
0040105D: 5B                 pop         ebx
0040105E: 83 C4 4C           add         esp,4Ch
00401061: 3B EC              cmp         ebp,esp
00401063: E8 58 00 00 00     call          chkesp
00401068: 8B E5              mov         esp,ebp
0040106A: 5D                 pop         ebp
0040106B: C3                 ret
```




```
int main()
{
    int m=5,n=6,j;
    //printf("test!");
    j=add(m,n);
    return 0;
}
```

观察main函数的堆栈框架


```
_main:
00401020: 55                 push        ebp
00401021: 8B EC              mov         ebp,esp
00401023: 83 EC 4C           sub         esp,4Ch
00401026: 53                 push        ebx
00401027: 56                 push        esi
00401028: 57                 push        edi
00401029: 8D 7D B4           lea         edi,[ebp-4Ch]
0040102C: B9 13 00 00 00     mov         ecx,13h
00401031: B8 CC CC CC CC     mov         eax,0CCCCCCCCh
00401036: F3 AB              rep stos    dword ptr es:[edi]
00401038: C7 45 FC 05 00 00  mov         dword ptr [ebp-4],5
          00
0040103F: C7 45 F8 06 00 00  mov         dword ptr [ebp-8],6
          00
00401046: 8B 45 F8           mov         eax,dword ptr [ebp-8]
00401049: 50                 push        eax
0040104A: 8B 4D FC           mov         ecx,dword ptr [ebp-4]
0040104D: 51                 push        ecx
0040104E: E8 B2 FF FF FF     call        @ILT+0(_add)
00401053: 83 C4 08           add         esp,8
00401056: 89 45 F4           mov         dword ptr [ebp-0Ch],eax
00401059: 33 C0              xor         eax,eax
0040105B: 5F                 pop         edi
0040105C: 5E                 pop         esi
0040105D: 5B                 pop         ebx
0040105E: 83 C4 4C           add         esp,4Ch
00401061: 3B EC              cmp         ebp,esp
00401063: E8 58 00 00 00     call          chkesp
00401068: 8B E5              mov         esp,ebp
0040106A: 5D                 pop         ebp
0040106B: C3                 ret
```



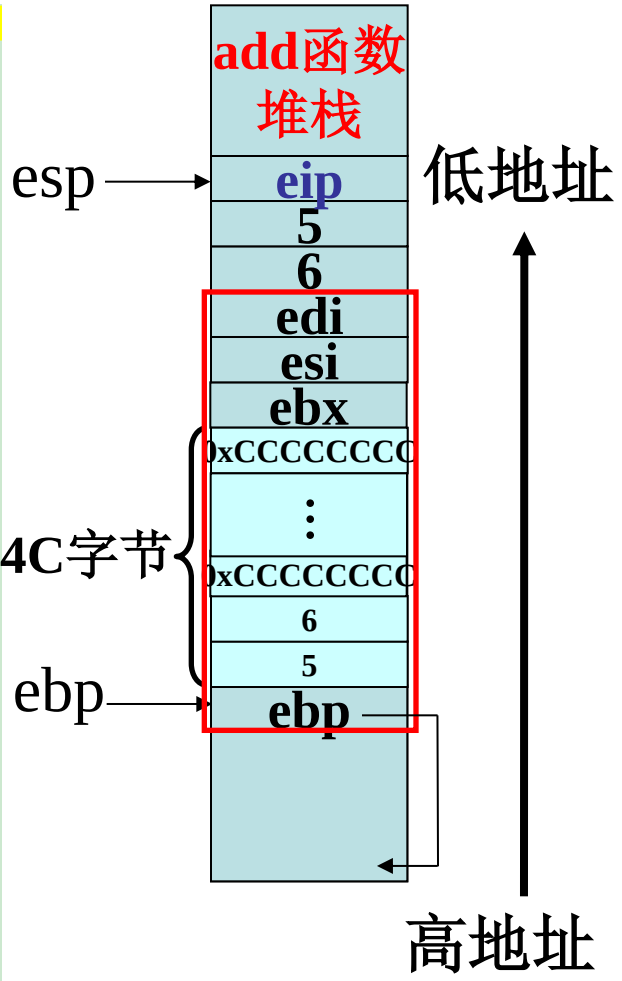

```
int main()
{
    int m=5,n=6,j;
    //printf("test!");
    j=add(m,n);
    return 0;
}
```

为接下来的函数调用做准备，值传递。

为什么不直接添加立即数到低地址中？

观察main函数的堆栈框架


```
_main:
00401020: 55                 push        ebp
00401021: 8B EC              mov         ebp,esp
00401023: 83 EC 4C           sub         esp,4Ch
00401026: 53                 push        ebx
00401027: 56                 push        esi
00401028: 57                 push        edi
00401029: 8D 7D B4           lea         edi,[ebp-4Ch]
0040102C: B9 13 00 00 00     mov         ecx,13h
00401031: B8 CC CC CC CC     mov         eax,0CCCCCCCCh
00401036: F3 AB              rep stos    dword ptr es:[edi]
00401038: C7 45 FC 05 00 00  mov         dword ptr [ebp-4],5
          00
0040103F: C7 45 F8 06 00 00  mov         dword ptr [ebp-8],6
          00
00401046: 8B 45 F8           mov         eax,dword ptr [ebp-8]
00401049: 50                 push        eax
0040104A: 8B 4D FC           mov         ecx,dword ptr [ebp-4]
0040104D: 51                 push        ecx
0040104E: E8 B2 FF FF FF     call        @ILT+0(_add)
00401053: 83 C4 08           add         esp,8
00401056: 89 45 F4           mov         dword ptr [ebp-0Ch],eax
00401059: 33 C0              xor         eax,eax
0040105B: 5F                 pop         edi
0040105C: 5E                 pop         esi
0040105D: 5B                 pop         ebx
0040105E: 83 C4 4C           add         esp,4Ch
00401061: 3B EC              cmp         ebp,esp
00401063: E8 58 00 00 00     call          chkesp
00401068: 8B E5              mov         esp,ebp
0040106A: 5D                 pop         ebp
0040106B: C3                 ret
```

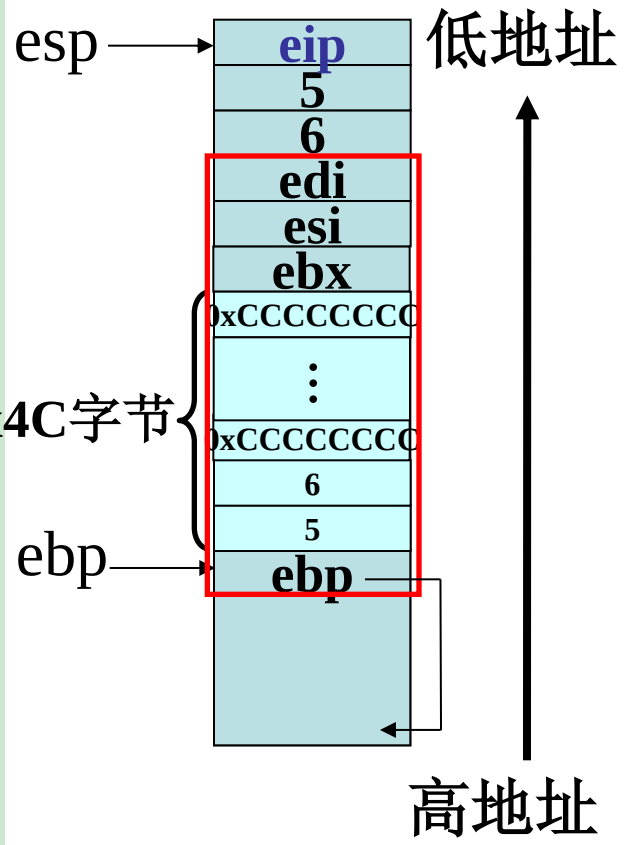



```
int main()
{
    int m=5,n=6,j;
    //printf("test!");
    j=add(m,n);
    return 0;
}
```

调用add函数，
其返回值默认保
存在eax寄存器

观察main函数的堆栈框架

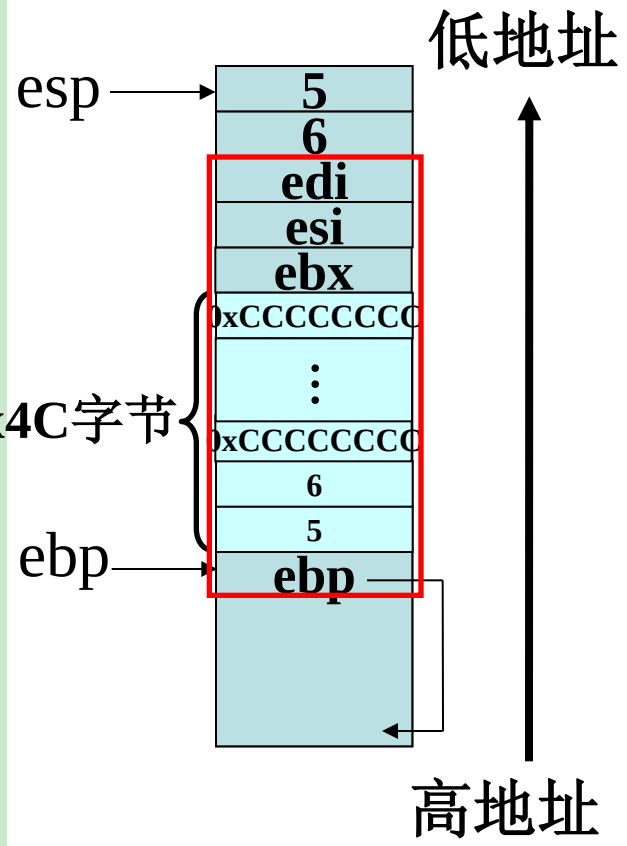

```
_main:
  00401020: 55                  push        ebp
  00401021: 8B EC               mov         ebp,esp
  00401023: 83 EC 4C            sub         esp,4Ch
  00401026: 53                  push        ebx
  00401027: 56                  push        esi
  00401028: 57                  push        edi
  00401029: 8D 7D B4            lea         edi,[ebp-4Ch]
  0040102C: B9 13 00 00 00      mov         ecx,13h
  00401031: B8 CC CC CC CC      mov         eax,0CCCCCCCCh
  00401036: F3 AB               rep stos    dword ptr es:[edi]
  00401038: C7 45 FC 05 00 00   mov         dword ptr [ebp-4],5
            00
  0040103F: C7 45 F8 06 00 00   mov         dword ptr [ebp-8],6
            00
  00401046: 8B 45 F8            mov         eax,dword ptr [ebp-8]
  00401049: 50                  push        eax
  0040104A: 8B 4D FC            mov         ecx,dword ptr [ebp-4]
  0040104D: 51                  push        ecx
  0040104E: E8 B2 FF FF FF      call        @ILT+0(_add)
  00401053: 83 C4 08            add         esp,8
  00401056: 89 45 F4            mov         dword ptr [ebp-0Ch],eax
  00401059: 33 C0               xor         eax,eax
  0040105B: 5F                  pop         edi
  0040105C: 5E                  pop         esi
  0040105D: 5B                  pop         ebx
  0040105E: 83 C4 4C            add         esp,4Ch
  00401061: 3B EC               cmp         ebp,esp
  00401063: E8 58 00 00 00      call          chkesp
  00401068: 8B E5               mov         esp,ebp
  0040106A: 5D                  pop         ebp
  0040106B: C3                  ret
```




```
int main()
{
    int m=5,n=6,j;
    //printf("test!");
    j=add(m,n);
    return 0;
}
```

观察main函数的堆栈框架

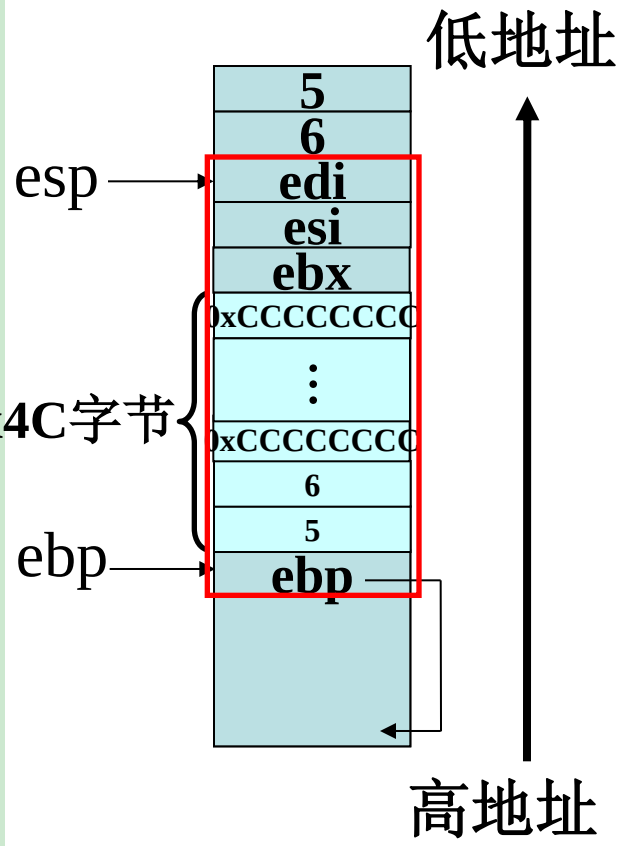

```
_main:
  00401020: 55                  push        ebp
  00401021: 8B EC               mov         ebp,esp
  00401023: 83 EC 4C            sub         esp,4Ch
  00401026: 53                  push        ebx
  00401027: 56                  push        esi
  00401028: 57                  push        edi
  00401029: 8D 7D B4            lea         edi,[ebp-4Ch]
  0040102C: B9 13 00 00 00      mov         ecx,13h
  00401031: B8 CC CC CC CC      mov         eax,0CCCCCCCCh
  00401036: F3 AB               rep stos    dword ptr es:[edi]
  00401038: C7 45 FC 05 00 00   mov         dword ptr [ebp-4],5
            00
  0040103F: C7 45 F8 06 00 00   mov         dword ptr [ebp-8],6
            00
  00401046: 8B 45 F8            mov         eax,dword ptr [ebp-8]
  00401049: 50                  push        eax
  0040104A: 8B 4D FC            mov         ecx,dword ptr [ebp-4]
  0040104D: 51                  push        ecx
  0040104E: E8 B2 FF FF FF      call        @ILT+0(_add)
  00401053: 83 C4 08            add         esp,8
  00401056: 89 45 F4            mov         dword ptr [ebp-0Ch],eax
  00401059: 33 C0               xor         eax,eax
  0040105B: 5F                  pop         edi
  0040105C: 5E                  pop         esi
  0040105D: 5B                  pop         ebx
  0040105E: 83 C4 4C            add         esp,4Ch
  00401061: 3B EC               cmp         ebp,esp
  00401063: E8 58 00 00 00      call          chkesp
  00401068: 8B E5               mov         esp,ebp
  0040106A: 5D                  pop         ebp
  0040106B: C3                  ret
```




```
int main()
{
    int m=5,n=6,j;
    //printf("test!");
    j=add(m,n);
    return 0;
}
```

观察main函数的堆栈框架

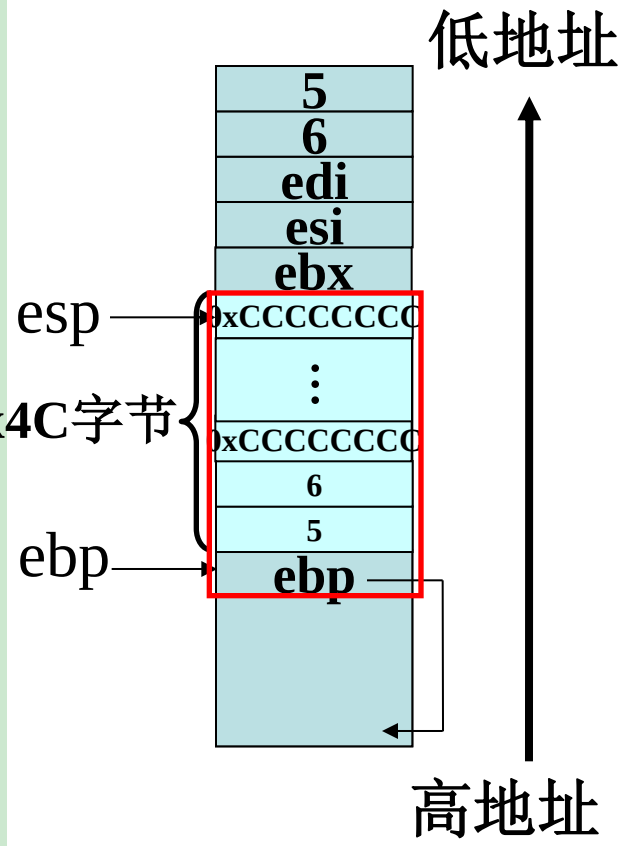

```
_main:
00401020: 55                 push        ebp
00401021: 8B EC              mov         ebp,esp
00401023: 83 EC 4C           sub         esp,4Ch
00401026: 53                 push        ebx
00401027: 56                 push        esi
00401028: 57                 push        edi
00401029: 8D 7D B4           lea         edi,[ebp-4Ch]
0040102C: B9 13 00 00 00     mov         ecx,13h
00401031: B8 CC CC CC CC     mov         eax,0CCCCCCCCh
00401036: F3 AB              rep stos    dword ptr es:[edi]
00401038: C7 45 FC 05 00 00  mov         dword ptr [ebp-4],5
          00
0040103F: C7 45 F8 06 00 00  mov         dword ptr [ebp-8],6
          00
00401046: 8B 45 F8           mov         eax,dword ptr [ebp-8]
00401049: 50                 push        eax
0040104A: 8B 4D FC           mov         ecx,dword ptr [ebp-4]
0040104D: 51                 push        ecx
0040104E: E8 B2 FF FF FF     call        @ILT+0(_add)
00401053: 83 C4 08           add         esp,8
00401056: 89 45 F4           mov         dword ptr [ebp-0Ch],eax
00401059: 33 C0              xor         eax,eax
0040105B: 5F                 pop         edi
0040105C: 5E                 pop         esi
0040105D: 5B                 pop         ebx
0040105E: 83 C4 4C           add         esp,4Ch
00401061: 3B EC              cmp         ebp,esp
00401063: E8 58 00 00 00     call          chkesp
00401068: 8B E5              mov         esp,ebp
0040106A: 5D                 pop         ebp
0040106B: C3                 ret
```




```
int main()
{
    int m=5,n=6,j;
    //printf("test!");
    j=add(m,n);
    return 0;
}
```

观察main函数的堆栈框架

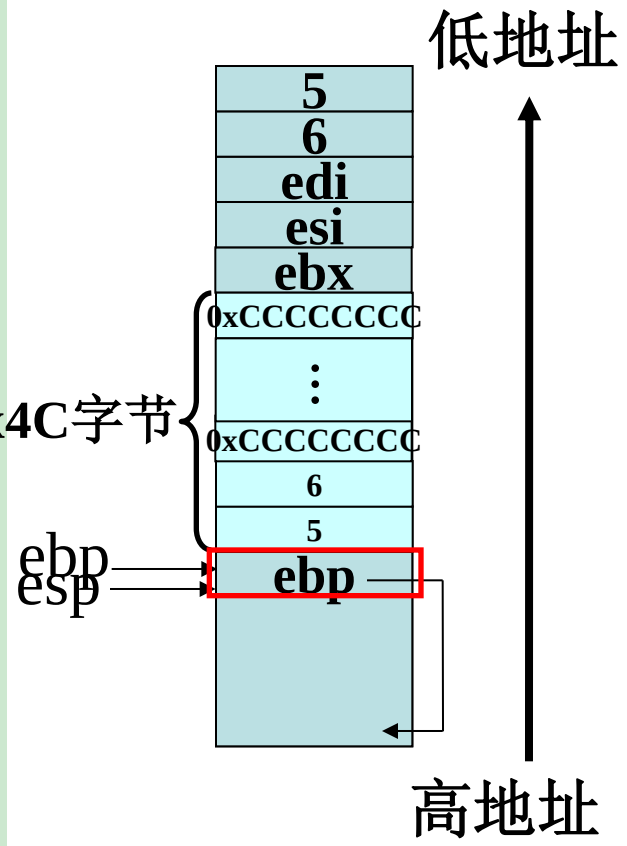

```
_main:
  00401020: 55                  push        ebp
  00401021: 8B EC               mov         ebp,esp
  00401023: 83 EC 4C            sub         esp,4Ch
  00401026: 53                  push        ebx
  00401027: 56                  push        esi
  00401028: 57                  push        edi
  00401029: 8D 7D B4            lea         edi,[ebp-4Ch]
  0040102C: B9 13 00 00 00      mov         ecx,13h
  00401031: B8 CC CC CC CC      mov         eax,0CCCCCCCCh
  00401036: F3 AB               rep stos    dword ptr es:[edi]
  00401038: C7 45 FC 05 00 00   mov         dword ptr [ebp-4],5
            00
  0040103F: C7 45 F8 06 00 00   mov         dword ptr [ebp-8],6
            00
  00401046: 8B 45 F8            mov         eax,dword ptr [ebp-8]
  00401049: 50                  push        eax
  0040104A: 8B 4D FC            mov         ecx,dword ptr [ebp-4]
  0040104D: 51                  push        ecx
  0040104E: E8 B2 FF FF FF      call        @ILT+0(_add)
  00401053: 83 C4 08            add         esp,8
  00401056: 89 45 F4            mov         dword ptr [ebp-0Ch],eax
  00401059: 33 C0               xor         eax,eax
  0040105B: 5F                  pop         edi
  0040105C: 5E                  pop         esi
  0040105D: 5B                  pop         ebx
  0040105E: 83 C4 4C            add         esp,4Ch
  00401061: 3B EC               cmp         ebp,esp
  00401063: E8 58 00 00 00      call          chkesp
  00401068: 8B E5               mov         esp,ebp
  0040106A: 5D                  pop         ebp
  0040106B: C3                  ret
```




```
int main()
{
    int m=5,n=6,j;
    //printf("test!");
    j=add(m,n);
    return 0;
}
```

观察main函数的堆栈框架

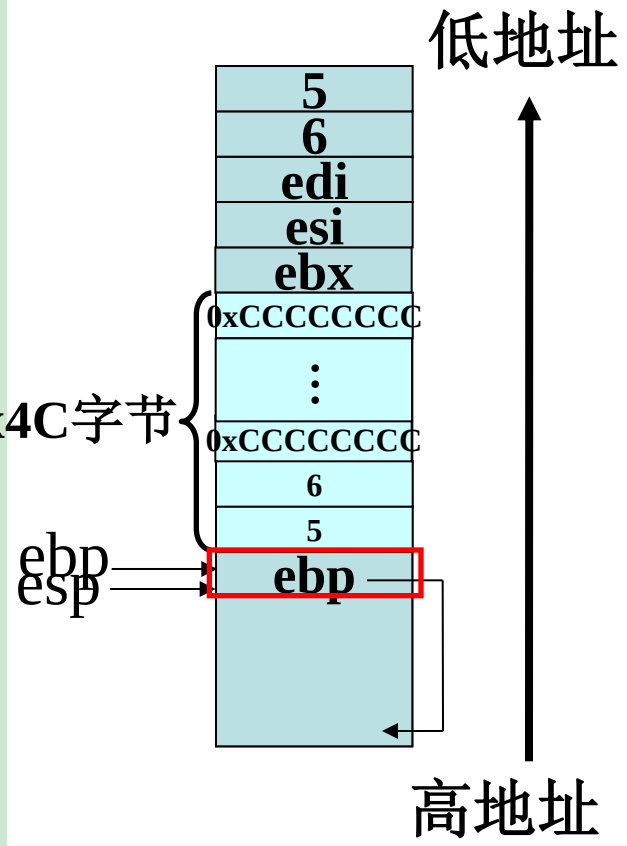

```
_main:
00401020: 55                 push        ebp
00401021: 8B EC              mov         ebp,esp
00401023: 83 EC 4C           sub         esp,4Ch
00401026: 53                 push        ebx
00401027: 56                 push        esi
00401028: 57                 push        edi
00401029: 8D 7D B4           lea         edi,[ebp-4Ch]
0040102C: B9 13 00 00 00     mov         ecx,13h
00401031: B8 CC CC CC CC     mov         eax,0CCCCCCCCh
00401036: F3 AB              rep stos    dword ptr es:[edi]
00401038: C7 45 FC 05 00 00  mov         dword ptr [ebp-4],5
          00
0040103F: C7 45 F8 06 00 00  mov         dword ptr [ebp-8],6
          00
00401046: 8B 45 F8           mov         eax,dword ptr [ebp-8]
00401049: 50                 push        eax
0040104A: 8B 4D FC           mov         ecx,dword ptr [ebp-4]
0040104D: 51                 push        ecx
0040104E: E8 B2 FF FF FF     call        @ILT+0(_add)
00401053: 83 C4 08           add         esp,8
00401056: 89 45 F4           mov         dword ptr [ebp-0Ch],eax
00401059: 33 C0              xor         eax,eax
0040105B: 5F                 pop         edi
0040105C: 5E                 pop         esi
0040105D: 5B                 pop         ebx
0040105E: 83 C4 4C           add         esp,4Ch
00401061: 3B EC              cmp         ebp,esp
00401063: E8 58 00 00 00     call          chkesp
00401068: 8B E5              mov         esp,ebp
0040106A: 5D                 pop         ebp
0040106B: C3                 ret
```




```
int main()
{
    int m=5,n=6,j;
    //printf("test!");
    j=add(m,n);
    return 0;
}
```

观察main函数的堆栈框架


```
_main:
00401020: 55                 push        ebp
00401021: 8B EC              mov         ebp,esp
00401023: 83 EC 4C           sub         esp,4Ch
00401026: 53                 push        ebx
00401027: 56                 push        esi
00401028: 57                 push        edi
00401029: 8D 7D B4           lea         edi,[ebp-4Ch]
0040102C: B9 13 00 00 00     mov         ecx,13h
00401031: B8 CC CC CC CC     mov         eax,0CCCCCCCCh
00401036: F3 AB              rep stos    dword ptr es:[edi]
00401038: C7 45 FC 05 00 00  mov         dword ptr [ebp-4],5
          00
0040103F: C7 45 F8 06 00 00  mov         dword ptr [ebp-8],6
          00
00401046: 8B 45 F8           mov         eax,dword ptr [ebp-8]
00401049: 50                 push        eax
0040104A: 8B 4D FC           mov         ecx,dword ptr [ebp-4]
0040104D: 51                 push        ecx
0040104E: E8 B2 FF FF FF     call        @ILT+0(_add)
00401053: 83 C4 08           add         esp,8
00401056: 89 45 F4           mov         dword ptr [ebp-0Ch],eax
00401059: 33 C0              xor         eax,eax
0040105B: 5F                 pop         edi
0040105C: 5E                 pop         esi
0040105D: 5B                 pop         ebx
0040105E: 83 C4 4C           add         esp,4Ch
00401061: 3B EC              cmp         ebp,esp
00401063: E8 58 00 00 00     call          chkesp
00401068: 8B E5              mov         esp,ebp
0040106A: 5D                 pop         ebp
0040106B: C3                 ret
```




```
int main()
{
    int m=5,n=6,j;
    //printf("test!");
    j=add(m,n);
    return 0;
}
```

观察子函数add的堆栈框架


```
_add:
 00401080: 55                   push    ebp
 00401081: 8B EC                mov     ebp,esp
 00401083: 83 EC 44             sub     esp,44h
 00401086: 53                   push    ebx
 00401087: 56                   push    esi
 00401088: 57                   push    edi
 00401089: 8D 7D BC             lea     edi,[ebp-44h]
 0040108C: B9 11 00 00 00       mov     ecx,11h
 00401091: B8 CC CC CC CC       mov     eax,0CCCCCCCCh
 00401096: F3 AB                rep stos    dword ptr es:[edi]
 00401098: 8B 45 08             mov     eax,dword ptr [ebp+8]
 0040109B: 03 45 0C             add     eax,dword ptr [ebp+0Ch]
 0040109E: 89 45 FC             mov     dword ptr [ebp-4],eax
 004010A1: 8B 45 FC             mov     eax,dword ptr [ebp-4]
 004010A4: 5F                   pop     edi
 004010A5: 5E                   pop     esi
 004010A6: 5B                   pop     ebx
 004010A7: 8B E5                mov     esp,ebp
 004010A9: 5D                   pop     ebp
 004010AA: C3
```

```
int add(int a, int b)
{
    int c;
    c=a+b;
    return c;
    //return (a+b);
}
```

案例1

问题：读取一个C程序文件来查明其中是否有任何不匹配的圆括号、方括号和大括号。如“(1+2*(3+5)”。

解题思路：

Step1：判断输入参数的数量是否正确。

Step2：依据文件名称，打开指定的文件并将内容进行读取，若不能正常打开，则报错；

Step3：若能正常读取文件，则*创建一个空栈*。若创建失败，需报错；

Step4：创建一个栈元素结点。

Step5：逐行读取文件，对于每一行，依次处理该行中所包含的各类括号

1）若是(或[或{，则将该括号及对应的行号保存到新建的结点中，并将该结点*入栈*；

2）若是)或]或}，则将当前的栈顶元素*出栈*，并查看是否与当前括号匹配。若匹配，则继续检查后续括号，否则报错，退出程序。

Step6：若文件读取完毕，栈中仍有元素，则打印出所有不匹配信息。

Step7：关闭文件。

案例2

问题：给定一个整数数组 temperatures ， 表示每天的温度，返回一个数组 answer ， 其中 answer[i] 是指对于第 i 天，下一个更高温度出现在几天后。如果气温在这之后都不会升高，请在该位置用 0 来代替。

示例 1:
输入: temperatures = [73,74,75,71,69,72,76,73]
输出: [1,1,4,2,1,1,0,0]

示例 2:
输入: temperatures = [30,40,50,60]
输出: [1,1,1,0]

案例2

示例 1: 输入: temperatures = [73,74,75,71,69,72,76,73]
输出: [1,1,4,2,1,1,0,0]

当 *i*=0 时，栈为空，将 0号元素进栈。

- *stack*=[0(73)]
- *ans*=[0,0,0,0,0,0,0,0

当 *i*=1 时，由于 74 大于 73，移除栈顶 0号，赋值 *ans*[0]:=1−0，栈顶为1号元素。

- *stack*=[1(74)]
- *ans*=[1,0,0,0,0,0,0,0]

当 *i*=2 时，由于 75 大于 74，因此移除栈顶元素 1，赋值 *ans*[1]:=2−1，栈顶为2号元素。

- *stack*=[2(75)]
- *ans*=[1,1,0,0,0,0,0,0]

当 *i*=3 时，由于 71 小于 75，栈顶为3号元素

- *stack*=[2(75),3(71)]
- *ans*=[1,1,0,0,0,0,0,0]

当 *i*=4 时，由于 69 小于 71，栈顶为4号元素

- *stack*=[2(75),3(71),4(69)]
- *ans*=[1,1,0,0,0,0,0,0]

当 *i*=5 时，由于 72 大于 69 和 71，因此依次移除栈顶元素 4 和 3，赋值 *ans*[4]:=5−4 和 *ans*[3]:=5−3，将 5 进栈

案例2


```
class Solution {
public:
    vector<int> dailyTemperatures(vector<int>& temperatures) {
        int n = temperatures.size();
        vector<int> ans(n);
        stack<int> s;
        for (int i = 0; i < n; ++i) {
            while (!s.empty() && temperatures[i] > temperatures[s.top()])
{
                int previousIndex = s.top();
                ans[previousIndex] = i - previousIndex;
                s.pop();
            }
            s.push(i);
        }
        return ans;
    }};
```

4.1 栈和队列

4.1 栈

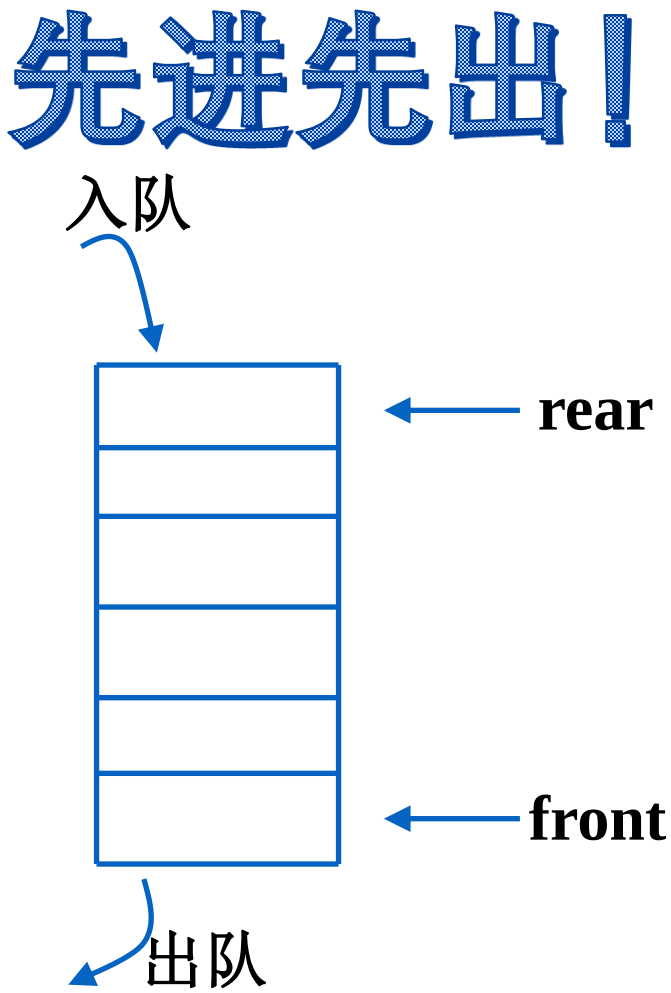
- 顺序栈的定义及实现
- 链栈的定义及实现
- 栈的应用：案例分析——函数调用

4.2 队列

- **循环队列的定义及实现**
- 链队列的定义及实现
- 队列的应用：案例分析

队列的定义及实现

- 队列：
  - 是操作受限的特殊线性表。
  - 是限定在一端进行插入，而在另一端进行删除的线性表；
  - 队头(front)：允许删除的一端；
  - 队尾(rear)：允许插入的一端
- 队列的应用举例：操作系统的作业排队；
- 存储结构：
  - 循环队列：顺序存储
  - 链队列：链式存储
- 原子操作:
  - 创建空队列
  - 销毁已有队列
  - 查找直接后继和直接前驱
  - 入队
  - 出队



循环队列的定义及实现

- 利用顺序表来实现队列。约定front指向队列头元素，rear 指向队尾元素的下一位置。

- C定义：

```
#define MAXQSIZE   100    /* 最大队列长度       */
typedef struct{
    ElemType    *base;                  /* 存储空间   */
    int             front;          /* 头指针，指向队列的头元素 */
    int             rear;     /* 尾指针，指向队尾元素的下一个位置*/
} SqQueue;      /* 非增量式的空间分配 */
```

顺序队列——原子操作的实现(分析)

- SqQueue Q;
- 创建空队列：
  - 将this_stack->base指向分配成功的连续内存空间的开始处; Q.front=Q.rear=0;;
- 入队：
  - 若队列未满， Q.rear位置放新插入的元素，Q.rear++（可避免大量移动） （分析：插入位置为Q.rear，无须判断位置的合法性；上溢即队列满的条件需要判断，由于是增量式分配，故队列满时需要重新申请空间）；
- 出队：
  - 若队列未空，则Q.front位置为待删除的元素，Q.front++（可避免大量移动）

- **问题：**

1）如何判定队空？如何判定队满？

队空条件：Q.front == Q.rear

队满条件：Q.rear == MAXQSIZE

2）为何要使用循环队列，而不用普通的顺序式队列？

存在假上溢 (由于出队操作，队列空间的上部可能存在空闲空间)

循环队列

- 假上溢的解决
  - 将队列假想为首尾相接的环，即循环队列。
    - 入队：……，Q.rear = (Q.rear+1)%MAXQSIZE
    - 出队：……，Q.front = (Q.front+1)%MAXQSIZE
    - 队空条件：Q.front == Q.rear，由于出队Q.front追上了Q.rear
    - 队满条件：Q.front == Q.rear，由于入队Q.rear追上了Q.front

- 问题：队空和队满的判断条件一样

循环队列

- **如何区分队空和队满?**
  - **方案1**：设标志位：不足在于需要额外对标志位的判断及维护
  - **方案2**：在队列的结构中引入长度成员，在初始化队列、入队、出队操作中维护这个成员。
  - **方案3**：少用一个元素空间，即队满的条件如下：

    (Q.rear+1)% MAXQSIZE == Q.front

- **进一步思考：**
  - **对比上面所列的3种方法在队列操作中处理的不同。**

循环队列

– 方案1： isFull 来区分循环队列的满和非满状态

布尔型标志位，插入、删除需要更新

- **初始化：**front 和 rear 指针指向通常是0，isFull 标志设置为 false，
- **插入操作 (Enqueue)**
  - 在向队列中添加新元素之前，检查是否队列已满：
  - 如果 isFull 为 true，则表示队列已满，不能插入新元素。
  - 如果 isFull 为 false ，若插入后 front == rear， isFull 设置为 true。否则，直接插入元素，不需要更改 isFull 的值。
- **删除操作 (Dequeue)**
  - 在从队列中移除元素之前，检查是否队列为空：
  - 如果 front == rear 且 isFull 为 false，则表示队列为空，不能进行删除操作。
  - 如果 front == rear 但 isFull 为 true，那么删除后队列将不再满，因此在删除元素之后需要将 isFull 设置为 false。
  - 否则，直接删除元素，不需要更改 isFull 的值

4.1 栈和队列

4.1 栈

- 顺序栈的定义及实现
- 链栈的定义及实现
- 栈的应用：案例分析——函数调用

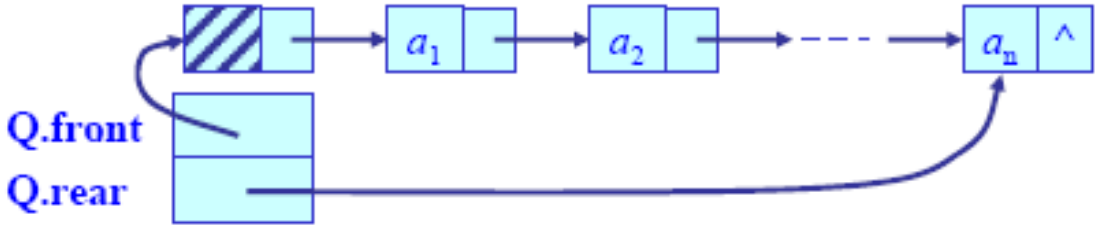
4.2 队列

- 循环队列的定义及实现
- **链队列的定义及实现**
- 队列的应用：案例分析

链队列的定义及实现

- 链队列：用双向链表来实现队列。
  - 队头(front)：用链表的头指针来表示；队尾(rear)：用链表的尾指针来表示

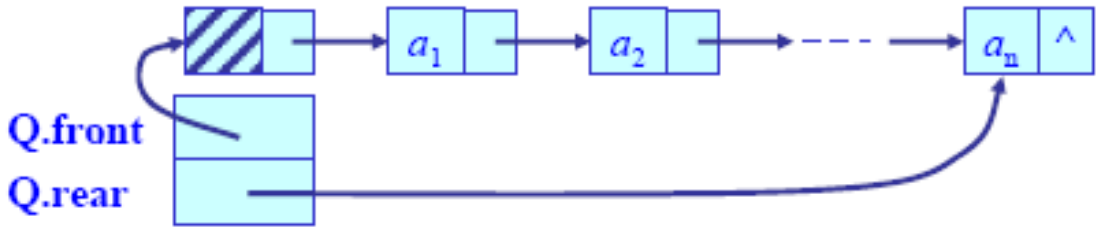
- 特点：
  - 无队列满问题：内存可扩充（除非是内存不足）；



- 问题：
  1）是否需引入头结点？（需要。特殊：对于空队列的入队）
  2）在链队列下的队列初始化、入队、出队算法如何实现？
  3）如何判断队空？

链队列的定义及实现

```
struct Node {
    struct Node     *prev;  /* link to previous node */
    struct Node     *next;  /* link to next node */
    void            *pdata; /* generic pointer to data *
};
typedef struct Node *Link;
/* a linked list data structure */
struct List {
    Link            LHead;              /* 队头指针，指向头元素 */
    Link            LTail;              /* 队尾指针，指向队尾元素 */
    unsigned int    LCount;
    void * ( * LCreateData )    ( void * );
    int    ( * LDeleteData )    ( void * );
    int    ( * LDuplicatedNode ) ( Link, Link );
    int    ( * LNodeDataCmp )   ( void *, void * );
};
```



队列的存储结构

- 循环队列：（难）
  - 需额外区分队空和队满

- 链队列：（推荐）
  - （直接利用带头结点的双向链表来实现）

案例

问题：读取一个C程序文件来查明其中是否有任何不匹配的圆括号、方括号和大括号。如“(1+2*(3+5)”。

解题思路：

Step1：判断输入参数的数量是否正确。

Step2：依据文件名称，打开指定的文件并将内容进行读取，若不能正常打开，则报错；

Step3：若能正常读取文件，则*创建一个空栈*。若创建失败，需报错；

Step4：创建一个栈元素结点。

Step5：逐行读取文件，对于每一行，依次处理该行中所包含的各类括号

1）若是(或[或{，则将该括号及对应的行号保存到新建的结点中，并将该结点*入栈*；

2）若是)或]或}，则将将当前的栈顶元素*出栈*，并查看是否与当前括号匹配。若匹配，则继续检查后续括号，否则报错，退出程序。

Step6：若文件读取完毕，栈中仍有元素，则打印出所有不匹配信息。

Step7：关闭文件。

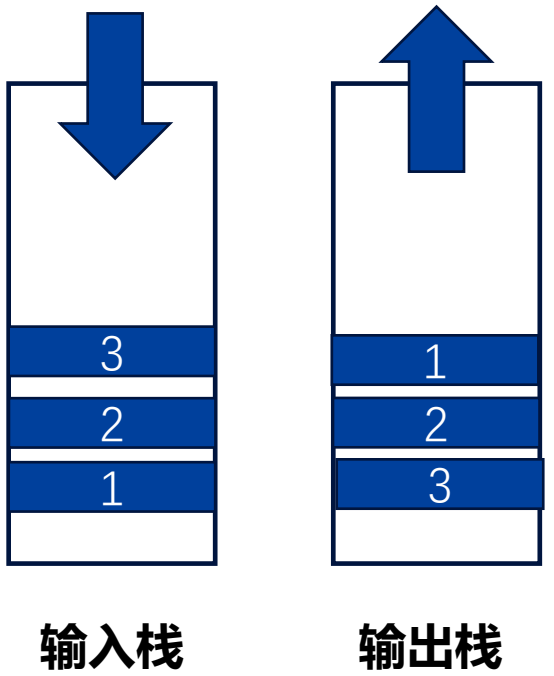
栈实现队列

- 栈的标准操作 **（先进后出：记作负）**
  - **Push to top, pop from top, size, is_empty**

- 需要实现的队列操作 **(先进先出：记作正)**
  - 入队：**push(x)**
  - 移除队首元素：**pop()**
  - 返回队首元素: **peek()**
  - 返回队列是否为空：**empty()**

仅用一个栈来模拟队列，是不行的

需要两个栈 一个输入栈，一个输出栈
（负负得正）



栈实现队列

```
    void push(int x) {
        stIn.push(x);
    }
```

```
    int pop() {
        // 当stOut为空的时候，从stIn里导入数据（导入stIn全部数据）
        if (stOut.empty()) {
            // 从stIn导入数据直到stIn为空
            while(!stIn.empty()) {
                stOut.push(stIn.top());
                stIn.pop();
            }
        }
        int result = stOut.top();
        stOut.pop();
        return result;
    }
```

```
int peek() {
    int res = this->pop(); // 直接使用pop函数
    stOut.push(res); // 弹出后添加回去
    return res;
}

bool empty() {
    return stIn.empty() && stOut.empty();
}
```

- 时间复杂度: push和empty为O(1), pop和peek为O(n)
- 空间复杂度: O(n)

队列实现栈

- 思考：如何用队列实现栈呢？

一个输入队列，一个输出队列，来模拟栈的功能？ 正正得负？

```
int pop() {
    int size = que.size();
    size--;
    while (size--) {
        que.push(que.front());
        que.pop();
    }
    int result = que.front();
    que.pop();
    return result;
}
```

```
int top() {
    return que.back();
}

bool empty() {
    return que.empty();
}
```

入栈操作 $O(n)$，其余操作都是 $O(1)$

栈的应用

题目描述：根据 逆波兰表示法，求表达式的值。有效的运算符包括 + ，- ，* ，/ 。每个运算对象可以是整数，也可以是另一个逆波兰表达式。

说明：
整数除法只保留整数部分。 给定逆波兰表达式总是有效的。换句话说，表达式总会得出有效数值且不存在除数为 0 的情况。

示例 1：

输入: ["2", "1", "+", "3", " * "]
输出: 9
解释: 中缀算术表达式为：((2 + 1) * 3) = 9

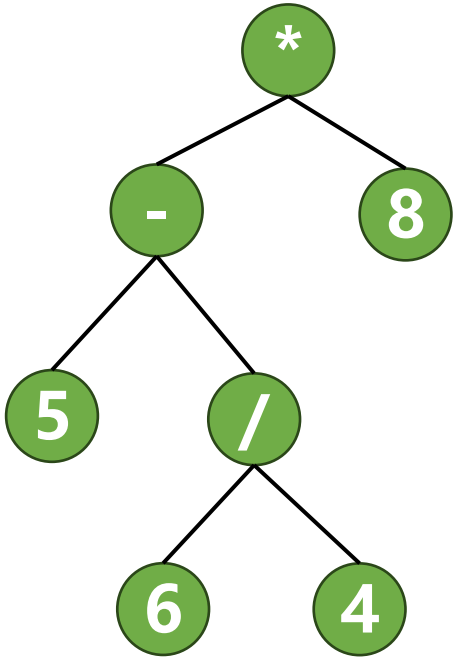
示例 2：

输入: ["4", "13", "5", "/", "+"]
输出: 6
解释: 中缀算术表达式为：(4 + (13 / 5)) = 6

栈的应用

逆波兰表达式，也叫做后缀表达式: "操作数① 操作数③ 运算符②"的顺序

平时经常遇到的是中缀表达式，是其对应的语法树**中序遍历**;
后缀表达式是其对应的语法树的**后序遍历**;



中序：(5-6/4) *8

后序：5 6 4 / - 8 *

对于计算机而言，后缀表达式求值更简单（可以不需要括号，适合用栈来实现）

栈的应用

题目描述：根据 逆波兰表示法，求表达式的值。有效的运算符包括 +，-，*，/ 。每个运算对象可以是整数，也可以是另一个逆波兰表达式。

逆波兰表达式求值的过程是：

1. 如果遇到数字就进栈；
2. 如果遇到操作符，就从栈顶弹出两个数字分别为 num2（栈顶）、num1（栈中的第二个元素）；
3. 计算 num1 运算 num2，后将结果压栈。

栈的应用

```
int evalRPN(vector<string>& tokens) {
        stack<int> stk;
        int n = tokens.size();
        for (int i = 0; i < n; i++) {
            string& token = tokens[i];
            if (isNumber(token)) {
                stk.push(atoi(token.c_str()));  // stk.push(std::stoi(token))
            } else {
                int num2 = stk.top();stk.pop();
                int num1 = stk.top();stk.pop();
                switch (token[0]) {
                    case '+':stk.push(num1 + num2); break;
                    case '-':stk.push(num1 - num2);break;
                    case '*': stk.push(num1 * num2); break;
                    case '/': stk.push(num1 / num2); break; }
            }
        }
        return stk.top();}
```

哈希表

重点：理解Hash表的特点及其实现；会构造Hash表；会使用外部拉链法来解决冲突。
难点：针对具体应用合理选择key；Hash表的查找性能分析。

基础：C语言编程；顺序表和双向链表的定义及基本操作的实现。

5.1 什么是哈希表

5.2 哈希表的构造

5.3 冲突解决方法

5.4 哈希表的查找及性能分析

什么是哈希表

- Hash表（哈希表）是一种线性结构。
  - 有限个数据项组成的序列，记作(a1,a2, … , an)
- 可以建立数据项的关键字和其逻辑存储位置之间的对应关系。
- 即：HashKey = H (key)
  - Key（关键字）：是数据项（或记录）中某个分量的值，它可以用来标识一个数据元素（或记录）。
  - 主关键字：唯一标识一个记录的关键字
  - HashKey （hash键 ）：也称为槽。它是关键字的“像”，是关键字在该表中的逻辑存储位置。 （必须合法，∈ [0,hash表长-1]）
  - h（Hash函数）：是一个映像，它将一组关键字映像到一个有限的、地址连续的地址区间上。（h : Key -> HashKey）。

- Hash表的构造过程，是将关键字映像到其逻辑存储位置（或hash键 ）的过程。
- 冲突：两个不同的数据项映像到同一个HashKey上。即：Key1≠Key2，但H(Key1)= H(Key2)。

基本哈希表的实现

- 利用动态定义的顺序表来实现基本hash表。

```
#define Table_Size 100/*分配空间的大小*/
Typedef HashTable_Struct{
    ElemType *elem;       /*顺序表的存储空间*/
    int len;              /*实际长度*/
    int TableSize ; /*当前分配的空间大小*/
};
typedef struct HashTable_Struct HashTable;
```

哈希表的适用场景

- 要求在内存中存储具有线性结构的数据集合
- 集合中的数据项的数量预先无法确定
- 要求能快速、近似随机的访问数据项（按值查找）

哈希表的构造

- 散列函数的通用形式：
  - Hashkey = calculated-key(key) % Table_Size

- 完美散列函数：不同数据项，对应的HashKey也不同。（永远不会出现冲突）。

```
bool isAnagram(string s, string t) {
        int record[26] = {0};
        for (int i = 0; i < s.size(); i++) {
                    record[s[i] - 'a']++;}
        for (int i = 0; i < t.size(); i++) {
            record[t[i] - 'a']--;}
        for (int i = 0; i < 26; i++) {
            if (record[i] != 0) {
                // record数组如果有的元素不为零0，说明字符串s和t 一定是谁多了字符或者谁少了字符。
                return false;}}
        // record数组所有元素都为零0，说明字符串s和t是字母异位词
        return true;
```

哈希表的构造

- 实际：几乎不可能构造出完全散列函数，因此应选择良好的通用算法。
- 良好的散列函数：**HashPJW, ElfHash**

  1）计算快速，Hashkey分布均匀;

  2）必须弥补可能出现在输入数据中的聚集。

```
unsigned long ElfHash ( const unsigned char *name )
{
    unsigned long   h = 0, g;
    while ( *name ){
        h = ( h << 4 ) + *name++;
        if ( g = h & 0xF0000000 )
            h ^= g >> 24;
        h &= ~g;
    }
    return h;}
```

“冲突”≠“聚集”

冲突解决方法

- Q1：什么是冲突？
- Q2：什么情况下需要处理冲突？
- Q3：如何处理冲突？

- “处理冲突”：为产生冲突的地址寻找下一个哈希地址。
  具体地，为产生冲突的地址H(key) 求得一个地址序列：H0, H1, H2, …, Hs 1≤s≤m-1
  - Hi = (H(key) + di) MOD m，其中： i=1, 2, …, s
    - H(key)为哈希函数;
    - m为哈希表长;
    - di为增量序列;

冲突解决方法

- 线性再散列法
  - di 为线性的
- 非线性再散列法
  - di 为非线性的
- 外部拉链法
  - 将散列表看做一个链表数组

线性再散列法

- di 为线性的，思想：从冲突位置开始，以顺序方式遍历散列表，来查找一个可用的槽。
- di可以是1，3，5等与表大小互质的数即可。
- 可能有3种结果：1）该元素已经在hash表中了； 2）找到一个空槽；3）hash表满了。

di为何要选择质数?

显然di=2/4并不是一个好选择，而di为质数可以保证表中的每个槽都会被检查到

证明:

线性再散列法

缺点：

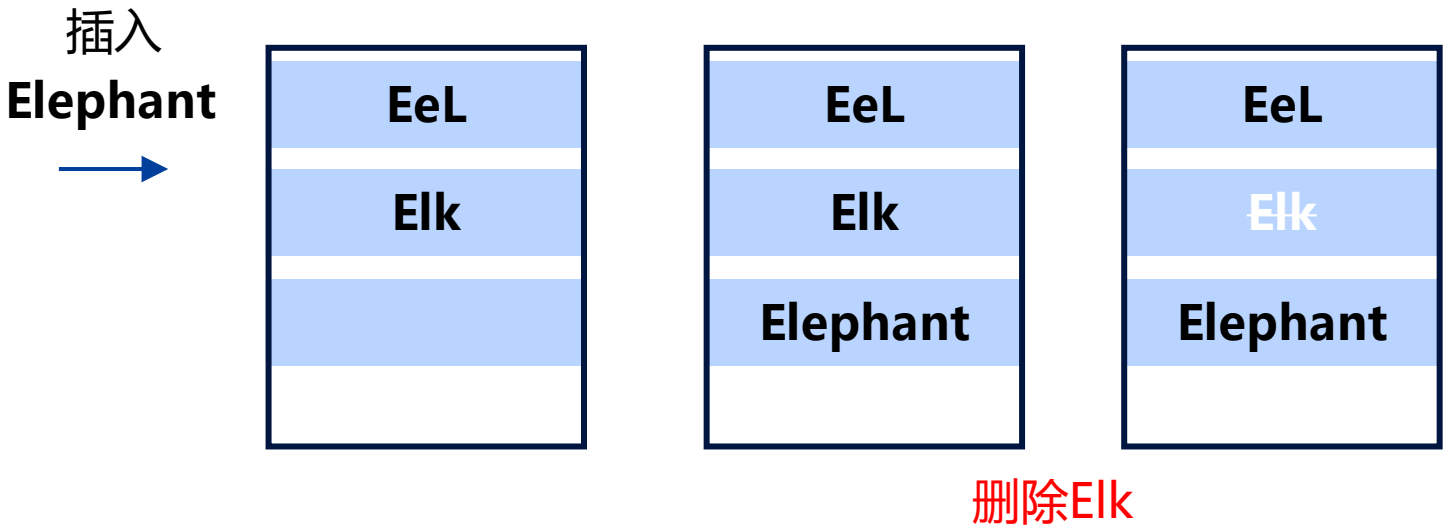
1）不能从hash表中删除数据。

2）当表被填满时性能下降明显；

对于缺点1：

对策：把使用过的槽标记为**无效**，才能执行删除。若查找时遇到无效的槽，则应继续执行查找。

例如，假定有一个包含 **26 个元素的散列表**，我们首先将其用于保存单词，并基于它们的**首字母**对它们进行散列



如果查找停止在第一个空槽上，则根本不能从表中执行删除操作

非线性再散列法

- **di 为非线性的**
  - 比如，**di =** 以固定的数作为种子所生成的随机数序列。
- 可以避免相似数据项的聚集。

- 负载因子**α：hash**表中的数据项个数**(n)**除以可用槽的总数**(HashTable_Size)**。
  - 负载因子越大，冲突概率越大。

总结：再散列法

- 优点：
  - 容易进行动态编码；
  - 负载因子较低并且不太可能执行删除操作的情况下，它的速度足够快；
  - 通常认为，负载因子$\alpha>0.5$时，再散列将不是一种切实可行的解决方案。

- 适用场景：
  - 只应该在快速而又随性的情况下，或者在快速原型化的环境中使用再散列法解决冲突。
  - 若不满足上述需求，则应该使用外部拉链法。

外部拉链法

- 将散列表看做一个链表数组。
  - Hash表中的每个槽要么为空，要么指向一个链表。
- 可以通过将数据项添加到链表中的方法来解决冲突：将所有hashkey相同的数据项存储在同一链表中。（“聚集”的效果）
- 解决冲突的代价：
  - 不会超过向链表中添加一个结点(采用“头插法”)
  - 无需执行再散列。

- 与前面的2种再散列法不同之处：
  - 外部拉链法可以容纳的元素只取决于可用的内存大小；
  - 而再散列法中hash表的最大表项取决于表的大小。

用外部拉链法解决冲突时哈希表的实现

- 利用链表数组来描述。

```
#define Table_Size 100/*分配空间的大小*/
Typedef HashTable_Struct{
        Link *elem;       /*顺序表的存储空间*/
        int len;          /*实际长度*/
        int TableSize ; /*当前分配的空间大小*/
};
typedef struct HashTable_Struct HashTable;
```

总结

- 优点：
  - 平均查找时间=链表长度/2+1（链表非空时）；
- 缺点：
  - 需要多一些的存储空间，因为每次探查时都需要添加结点，而不仅仅是数据项。但是，在硬件便宜的现在，可以忽略不计，故该方法现在用得最多。

例题

- 给定关键字Key的集合{19,01,23,14,55,68,11,82,36}
- 假设H(key)=key mod 11（即：表长=11），
- 要求：依据不同的冲突解决方法，分别构造出Hash表

1）采用线性再散列法解决冲突:di=1,2,3, …

0	1	2	3	4	5	6	7	8	9	10
55	01	23	14	68	11	82	36	19		
1	1	2	1	3	6	2	5	1		

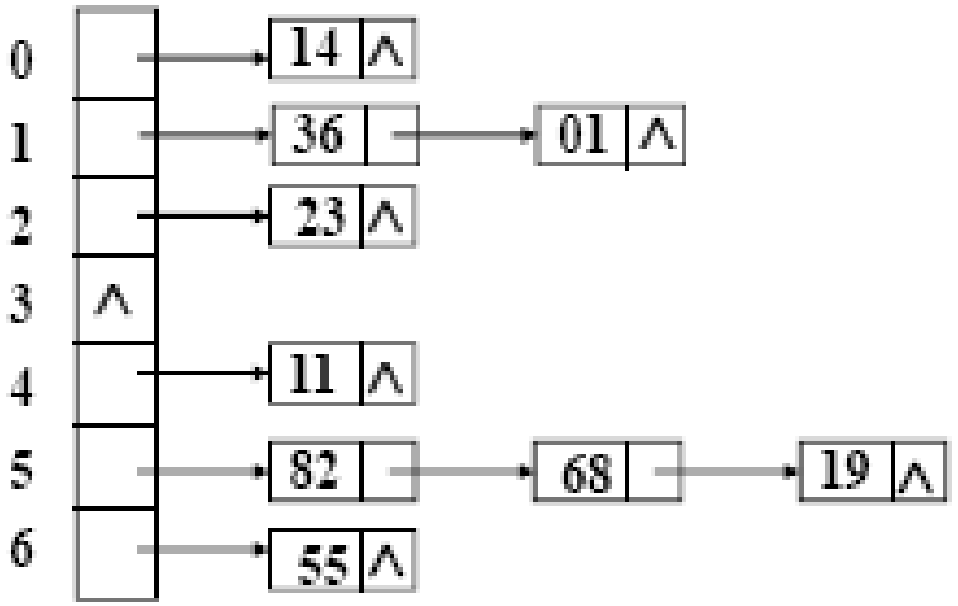
关注：探查次数

2）采用非线性再散列法解决冲突，假设di=1^2,-1^2, 2^2,-2^2, 3^2,-3^2,…

0	1	2	3	4	5	6	7	8	9	10
55	01	23	14	36	82	68		19		11
探查次数: 1	1	2	1	2	1	4		1		3

例题

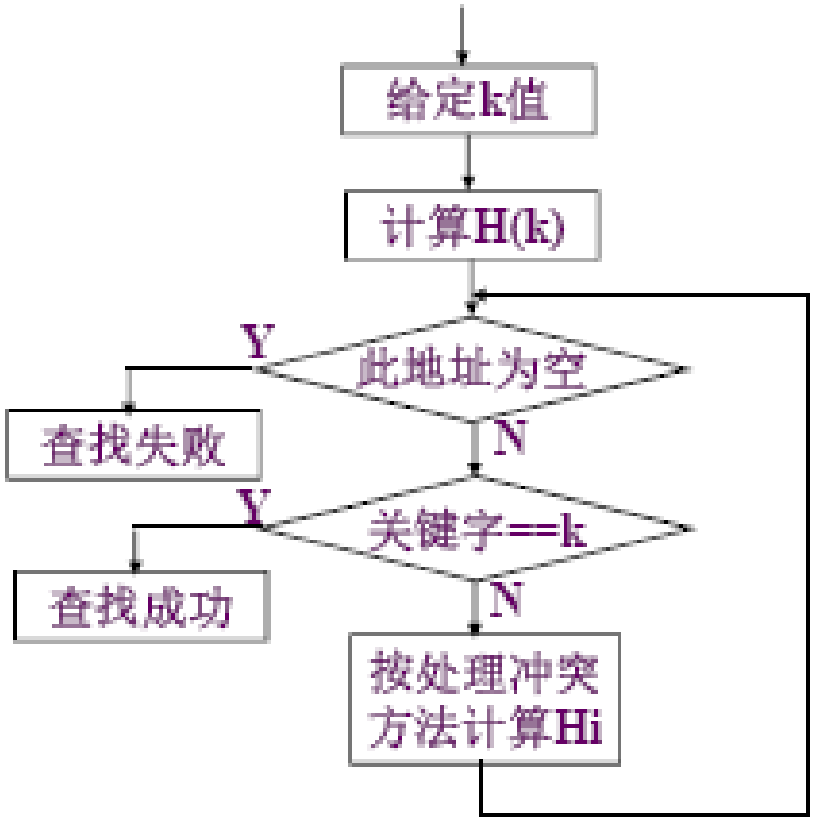
- 给定关键字集合{19,01,23,14,55,68,11,82,36}
- 假设H(key)=key mod 7（表长=7），
- 要求：采用外部拉链法解决冲突，并构造出Hash表



关注：探查次数

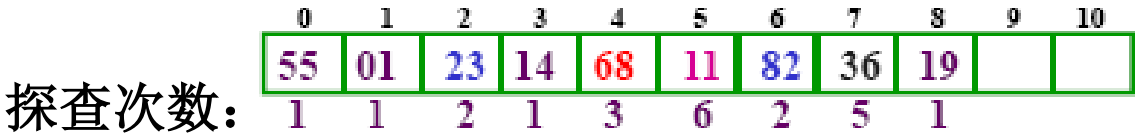
哈希表的查找及性能分析

- 如何查找？（对比：如何构造Hash表？）



哈希表的查找及性能分析

- 采用ASL (Avarage Search Length)来衡量Hash表的性能。
  Case 1：采用线性再散列法解决冲突:di=1,2,3, …



- 查找成功时的平均查找长度(查找概率相等):

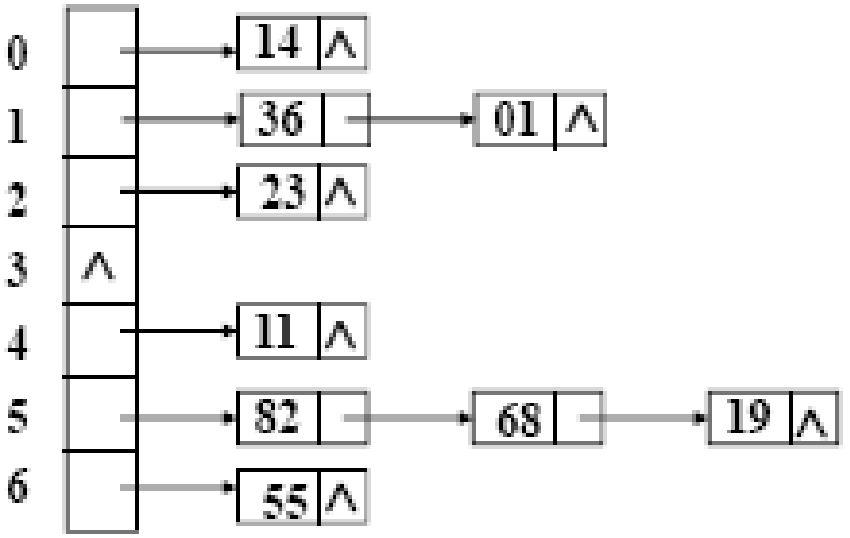
  ASL=(1·4+2·2+3·1+5·1+6·1)/9=22/9

- 查找不成功时的平均查找长度(查找概率相等):

  ASL=(10+9+8+7+6+5+4+3+2+1·2)/11=56/11

哈希表的查找及性能分析

Case 2:



关注：探查次数

- 查找成功(查找概率相等)时：
  ASL=(1·6+2·2+3·1)/9=13/9
- 查找不成功时的平均查找长度(查找概率相等):
  ASL=(2+3+2+1+2+4+2)/7=16/7

两数之和

- 给定一个整数数组 **nums** 和一个目标值 **target**，请你在该数组中找出和为目标值的那 两个 整数，并返回他们的数组下标。
  - 你可以假设每种输入只会对应一个答案。但是，数组中同一个元素不能使用两遍。
  - 示例:

    给定 **nums = [2, 7, 11, 15], target = 9**
    因为 **nums[0] + nums[1] = 2 + 7 = 9**
    所以返回 **[0, 1]**

两数之和

- 暴力解法：两层for循环，时间复杂度$O(n^2)$
- 使用哈希法：

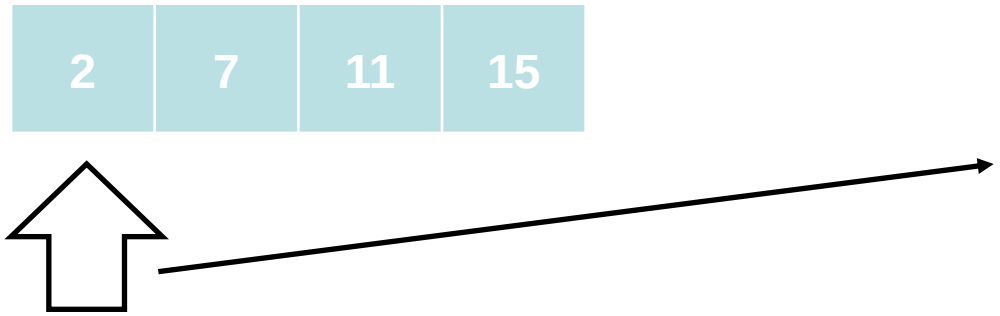
  - **Map**：

  {key：数据元素，value：数组元素对应的下标}

两数之和

- 暴力解法：两层for循环，时间复杂度$O(n^2)$
- 使用哈希法：

Target：9

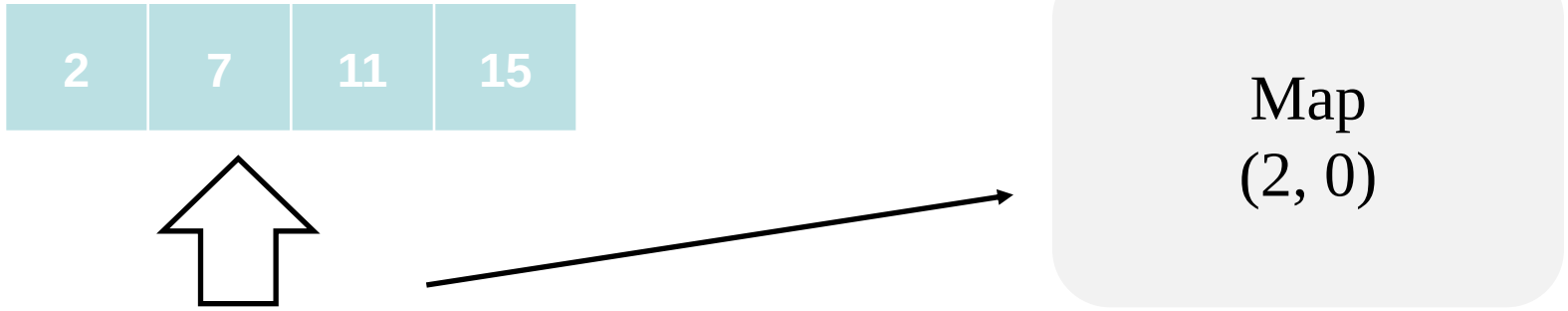


寻找7，不在map中，填入(2,0)

两数之和

- 暴力解法：两层for循环，时间复杂度$O(n^2)$
- 使用哈希法：

Target：9



寻找2，在map中，找到元素{2,7},
对应下标{0, 1}

两数之和

```
vector<int> twoSum(vector<int>& nums, int target) {
        unordered_map<int, int> hashtable;
        for (int i = 0; i < nums.size(); ++i) {
            auto it = hashtable.find(target - nums[i]);
            if (it != hashtable.end()) {
                return {it->second, i};
            }
            hashtable[nums[i]] = i;
        }
```