> individuality　一研为定

算法 内核 编译原理 设计模式
@[toc]

高级第五次 直播 内存 混合运算

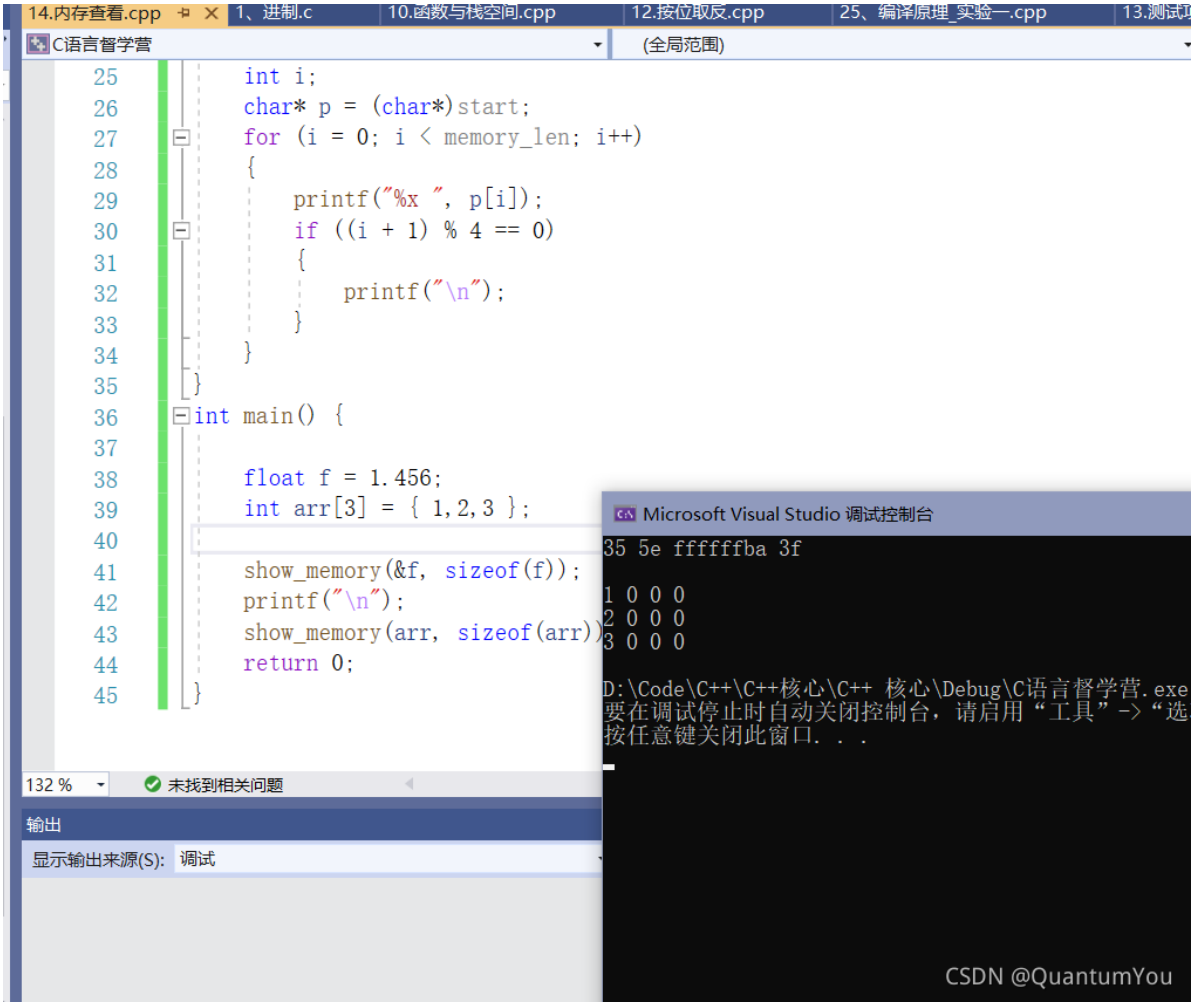
内存查看接口编写

- 位运算的应用实战

```c
#include  <stdio.h>


void show_memory_01(void* start, int memory_len)
{
    int i;
    char* p = (char*)start;
    for (i = 0; i < memory_len; i++)
    {
        if (i % 4 == 0) //输出地址
        {
            printf("0x%p ", p + i);
        }
        printf("%x", (p[i] & 0x000000f0) >> 4);  //输出内存的数据
        printf("%x ", p[i] & 0x0000000f);
        if ((i + 1) % 4 == 0)
        {
            printf("\n");
        }
    }
}

void show_memory(void* start, int memory_len)
{
    int i;
    char* p = (char*)start;
    for (i = 0; i < memory_len; i++)
    {
        printf("%x ", p[i]);
        if ((i + 1) % 4 == 0)
        {
            printf("\n");
        }
    }
}
int main() {

    float f = 1.456;
    int arr[3] = { 1,2,3 };

    show_memory(&f, sizeof(f));
    printf("\n");
    show_memory(arr, sizeof(arr));
```

```
    return 0;
}
```

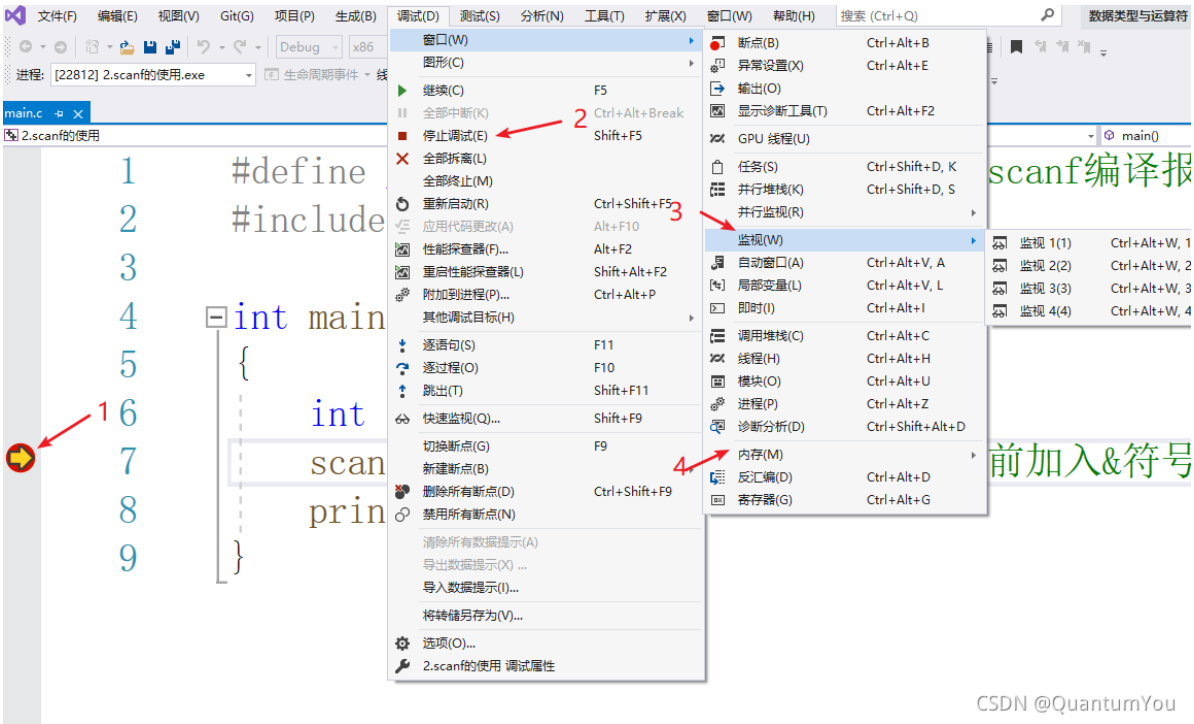
内存查看效果如下，将在后期进行优化

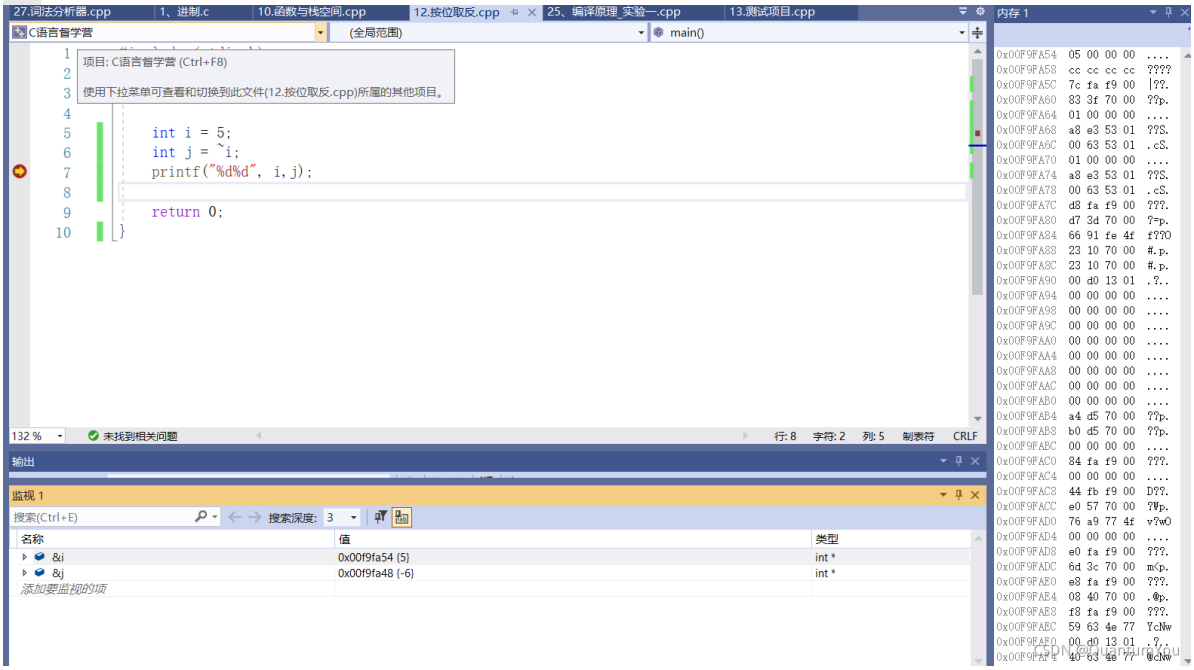


- **注意**：逻辑与 按位与的区别 && 与 & 的区别

参考链接

进行调试测试，注意要进行打断点





- **注意**： 计算机内存中存储的是补码
- `ff ff ff fa` 代表负数，所以先转化为原码，然后转为10进制，就是把`ff ff ff fa`除了符号位各位取反加一得到原码`1000 0000 0000 0000 0000 0000 0000 0110`也就是-6
- 求补码快捷方法：从高位开始按位去反，直到最后一个1（并且最后一个1保持不变）

```c
#include <stdio.h>


void show_memory(void* start, int memory_len)
{
    int i;
    char* p = (char*)start;
    for (i = 0; i < memory_len; i++)
    {
```

```
        if(i%4==0) //输出地址
        {
            printf("0x%p ", p+i);
        }
        printf("%x", (p[i]& 0x000000f0)>>4);  //输出内存的数据
        printf("%x ", p[i]& 0x0000000f);
        if ((i + 1) % 4 == 0)
        {
            printf("\n");
        }
    }
}


int main()
{
    int i = 5, j = 7;
    printf("i & j=%d\n", i & j);
    printf("i | j=%d\n", i | j);
    printf("i ^ j=%d\n", i ^ j);
    printf("~i=%d\n", ~i);
    //位运算实战
    float f = 1.456;
    show_memory(&f, sizeof(f));
    int arr[3] = { 1,2,3 };
    show_memory(arr, sizeof(arr));
    //异或交换两个数
    i = i ^ j;
    j = i ^ j;
    i = i ^ j;
    printf("i=%d,j=%d\n", i, j);
    //找出一个整数最低位为1的那个数
    printf("最低位为1的值 %d\n", 12 & -12);
    return 0;
}
```

1.有两个变量a与b,在不使用第三个量的情况下，通过异或操作来交换这两个变量的值，这种交换相对于之前的加法交换有何优势?

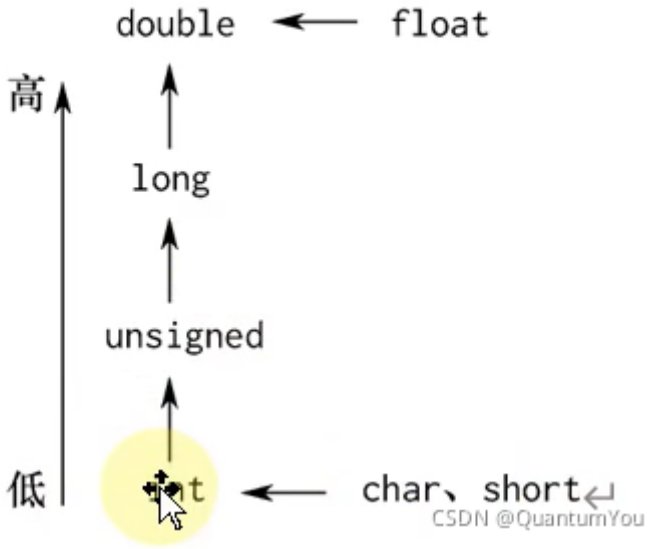
```
i = i^ j ;
j = j ^i ;
i = i ^j ;
```

- 找出一个数最低为位1 的那个数，解决方法：将其与自己的负数相与

0000 0101 => 5
1111 1011 => -5

混合运算

混合运算规则:不同数据类型转换级别



注意事项:

- 例子一：同时左右移位运算与分步左右移运算的区分

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

void big_int_mul()
{
    long long l;
    l = (long long)131072 * 131072;
    printf("%lld\n", l);
}


int main()
{
    char b = 0x93 << 1 >> 1;
    printf("%x\n", b);
    b = 0x93 << 1; //赋值一瞬间发生了丢失
    b = b >> 1;
    printf("%x\n", b);
    big_int_mul();
    return 0;
}
```

整型运算按4个字节进行

> 0x93
> 0000 0000 0000 0000 0000 0000 1001 0011
>
> 0000 0000 0000 0000 0000 0001 0010 0110
>
> b 0010 0110 0001 0011

- 例子二：数据存储 131072 是int 类型再与131072 相乘存储不下

```c
void big_int_mul()
{
    long long l;
    l = (long long)131072 * 131072;
    printf("%lld\n", l);
}
```

- 例子三：浮点型常量默认按8字节运算

```c
#include <stdio.h>
#include <stdlib.h>
//浮点型常量默认按8字节运算
int main()
{
    float f = 12345678900.0 + 1;
    double d = f;
    printf("%f\n", f);    //  12345678848。000000
    printf("%f\n", 12345678900.0 + 1);   //  12345678901.000000
    return 0;
}
```

深入理解 const

参考链接


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

void const_two()
{
    char str[] = "hello world";
    char str1[] = "how do you do";
    char* const ptr = str;//和普通变量一致，代表ptr被修改
    str[0] = 'H';
    puts(ptr);
    ptr[0] = 'n';  //合法
    puts(ptr);
```

```
    //ptr = "world";  //非法，编译错误，error C2166: 左值指定const对象

}

int main()
{
    const int i = 5;//i在下面的代码中不能修改，是常量
    char str[] = "hello world";
    const char* ptr = str;//这里代表ptr指向的空间不能被修改
    str[0] = 'H';  //操作合法
    puts(ptr);
    ptr = "world";
    //ptr[0] = 'n';  //操作非法，编译错误，提示error C2166: 左值指定const对象
    puts(ptr);
    //const修饰指针的第二种情况
    const_two();
    return 0;
}
```

结构体对齐原理

数据类型自身的对齐值如下：

- 对于char型数据，其自身对齐值为1,对于 short型为2,对于int,float, double类型 ， 其自身对齐值为4,单位字节。

高级第六次 直播 汇编讲解

指令格式与常用指令

- **操作码字段**：表征指令的操作特性与功能（指令的唯一标识）不同的指令操作码不能相同
- **地址码字段**：指定参与操作的操作数的地址码

架构：

1、英特尔： （AMD） x86 mov AL BL ；是BL 放到 AL 中
2、龙芯 （Mips）
3、ARM 高通、苹果、华为
4、Powerpc IBM

不同架构间汇编指令差异很大

> 复杂指令集：变长 x86
> 精简指令集：等长 arm

1、C文件预处理后变为 i 文件
2、文件经过编译后变为s文件汇编文件
3、汇编文件经过汇编变为目标文件oj
4、Obj经过链接变为exe

常用指令

- 汇编指令通常可以分为数据传送指令、逻辑计算指令和控制流指令，下面以 Intel格式为例，介绍一些重要的指令。以下用于操作数的标记分别表示寄存器、内存和常数

1、`<reg>`:表示任意寄存器，若其后带有数字，则指定其位数，如reg<32>表示32位 寄存器(eax、ebx、ecx、edx、esi、edi、esp或ebp);表示16位寄存器(ax,bx、cx或dx);表示8位寄存器(ah、al、bh、bl、ch、cl、dh、dl)

2、`<mem>`:表示内存地址(如[eax]、[var+4]或 dword ptr[eax+ebx]).
3、`<con>`:表示8位、16位或32位常数。表示8位常数；`<con16>`表示16位常数；`<con32>`表示32位常数。

- 数据传送指令、算术和逻辑运算指令、控制流指令

[汇编语言学习启航]

理解数组与指针对应的汇编

指令中指定操作数存储位置的字段称为地址码，地址码中可以包含存储器地址，也可包含寄存器编号。
指令中可以有一个、两个或者三个操作数，也可没有操作数，根据一条指令有几个操作数地址，可将指令分为零地址指令、一地址指令、二地址指令、三地址指令。4 个地址码的指令很少被使用

操作码字段	地址码	
操作码	A1 A2 A3	三指令地址
操作码	A1 A2	二指令地址
操作码	A1	一指令地址
操作码		零指令地址

通用寄存器

31 16	15 8	7 0
	AH	AL
	BH	BL
	CH	CL
	DH	DL
ESI		
EDI		
EBP		
ESP		

16bit	32bit	说明
AX	EAX	累加器（Accumulator）
BX	EBX	基地址寄存器（Base Register）
CX	ECX	计数寄存器（Count Register）
DX	EDX	数据寄存器（Data Register）
	ESI	变址寄存器（Index Register）
	EDI	
	EBP	堆栈基指针（Base Pointer）
	ESP	堆栈顶指针（Stack Pointer）

除 EBP 和 ESP 外，其他几个寄存器的用途是比较任意的。

main 去调用子函数是，前后所做的工作

汇编实战

- 在转化为汇编代码时，所有的变量名将不在
- 任何一个函数都是自己独立的栈空间

英特尔CPU 栈顶在低地址，栈底在高地址

（1）**ESP**：栈指针寄存器(extended stack pointer)，其内存放着一个指针，该指针永远指向系统栈最上面一个栈帧的栈顶。

（2）**EBP**：基址指针寄存器(extended base pointer)，其内存放着一个指针，该指针永远指向系统栈最上面一个栈帧的底部。

- 关于 `DWORD PTR` 是什么意思：
  1、`dword` 双字 就是四个字节
  2、`ptr` pointer缩写 即指针
  3、`[]`里的数据是一个地址值，这个地址指向一个双字型数据

比如：mov eax, dword ptr [12345678] 把内存地址12345678中的双字型（32位）数据赋给eax

- `lea`指令的作用：是`DWORD PTR_arrs[ebp]`对应空间的内存地址值放到eax中

```
; 10   :   p=arr;

00063 8d 45 ec   lea   eax, DWORD PTR _arr$[ebp]
00066 89 45 e0   mov   DWORD PTR _p$[ebp], eax
```

条件码

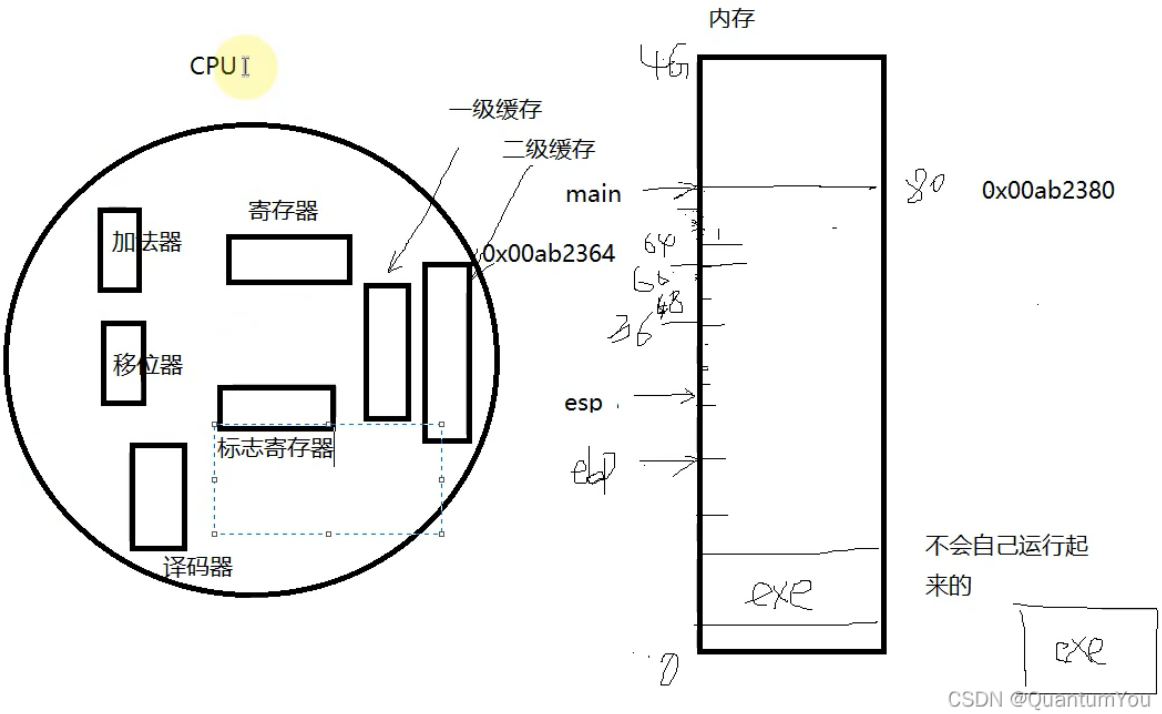
- 编译器通过条件码（标志位）设置指令和各类转移指令来实现程序中的选择结构语句。

(1)条件码（标志位）
除了整数寄存器，CPU还维护着一组条件码（标志位）寄存器，它们描述了最近的算术或逻辑运算操作的属性。可以检测这些寄存器来执行条件分支指令，最常用的条件码有：

- CF:进（借）位标志。最近无符号整数加（减）运算后的进（借）位情况。有进（借）位，CF=1;否则CF=0
- ZF:零标志。最近的操作的运算结算是否为0。若结果为0,ZF=1;否则ZF=0
- SF:符号标志。最近的带符号数运算结果的符号。负数时，SF=1;否则SF=0
- OF:溢出标志。最近带符号数运算的结果是否溢出，若溢出，OF=1;否则OF=0
  可见，**OF和SF对无符号数运算来说没有意义**，而CF对带符号数运算来说没有意义。

常见的算术逻辑运算指令(add、sub、 imul, or、and、shl、inc、dec、not、sal等)会设置条件码。但有两类指令只设置条件码而不改变任何其他寄存器：**cmp指令和sub指令的行为一样，test指令与and指令的行为一样，但它们只设置条件码，而不更新目的寄存器。**

- 之前介绍过的`jcondition`条件转跳指令，就是根据条件码ZF和SF来实现转跳。




```c
#include <stdio.h>
#include <stdlib.h>
int main()
```

```
{
    int arr[3]={1,2,3};
    int *p;
    int i=5;
    int j=10;
    i=arr[2];
    p=arr;
    printf("i=%d\n",i);
    if (i < j)
    {
        printf("i is small\n");
    }
    system("pause");
}
```

主体汇编代如下

```
;   COMDAT _main
_TEXT   SEGMENT
_j$ = -56                       ; size = 4
_i$ = -44                       ; size = 4
_p$ = -32                       ; size = 4
_arr$ = -20                     ; size = 12
__$ArrayPad$ = -4                   ; size = 4
_main   PROC                        ; COMDAT

; 4    : {

  00000 55       push    ebp
  00001 8b ec        mov     ebp, esp
  00003 81 ec fc 00 00
    00       sub     esp, 252       ; 000000fcH
  00009 53       push    ebx
  0000a 56       push    esi
  0000b 57       push    edi
  0000c 8d bd 04 ff ff
    ff       lea     edi, DWORD PTR [ebp-252]
  00012 b9 3f 00 00 00   mov     ecx, 63            ; 0000003fH
  00017 b8 cc cc cc cc   mov     eax, -858993460        ; ccccccccH
  0001c f3 ab        rep stosd
  0001e a1 00 00 00 00   mov     eax, DWORD PTR ___security_cookie
  00023 33 c5        xor     eax, ebp
  00025 89 45 fc     mov     DWORD PTR __$ArrayPad$[ebp], eax
  00028 b9 00 00 00 00   mov     ecx, OFFSET __8B08B150_main@c
  0002d e8 00 00 00 00   call    @__CheckForDebuggerJustMyCode@4

; ---------------------------------------
;  下面代码主要实现 数组为什么数组名为其首地址
; ---------------------------------------
; 5    :    int arr[3]={1,2,3};

  00032 c7 45 ec 01 00
    00 00        mov     DWORD PTR _arr$[ebp], 1
  00039 c7 45 f0 02 00
    00 00        mov     DWORD PTR _arr$[ebp+4], 2
  00040 c7 45 f4 03 00
    00 00        mov     DWORD PTR _arr$[ebp+8], 3
```

```
41
42  ; 6    :    int *p;
43  ; 7    :    int i=5;
44
45    00047 c7 45 d4 05 00
46      00 00        mov     DWORD PTR _i$[ebp], 5
47
48  ; 8    :    int j=10;
49
50    0004e c7 45 c8 0a 00
51      00 00        mov     DWORD PTR _j$[ebp], 10 ; 0000000aH
52
53  ; 9    :    i=arr[2];
54
55    00055 b8 04 00 00 00   mov     eax, 4
56    0005a d1 e0        shl     eax, 1
57    0005c 8b 4c 05 ec  mov     ecx, DWORD PTR _arr$[ebp+eax]
58    00060 89 4d d4     mov     DWORD PTR _i$[ebp], ecx
59
60  ; 10   :    p=arr;
61
62    00063 8d 45 ec     lea     eax, DWORD PTR _arr$[ebp]
63    00066 89 45 e0     mov     DWORD PTR _p$[ebp], eax
64
65  ; 11   :    printf("i=%d\n",i);
66
67    00069 8b 45 d4     mov     eax, DWORD PTR _i$[ebp]
68    0006c 50       push    eax
69    0006d 68 00 00 00 00   push    OFFSET ??_C@_05BKKKKIID@i?$DN?$CFd?6@
70    00072 e8 00 00 00 00   call    _printf
71    00077 83 c4 08     add     esp, 8
72
73  ; 12   :    if (i < j)
74
75    0007a 8b 45 d4     mov     eax, DWORD PTR _i$[ebp]
76    0007d 3b 45 c8     cmp     eax, DWORD PTR _j$[ebp]
77    00080 7d 0d        jge     SHORT $LN2@main
78
79  ; 13   :    {
80  ; 14   :        printf("i is small\n");
81
82    00082 68 00 00 00 00   push    OFFSET ??_C@_0M@KNINEIJI@i?5is?5small?6@
83    00087 e8 00 00 00 00   call    _printf
84    0008c 83 c4 04     add     esp, 4
85  $LN2@main:
86
87  ; 15   :    }
88  ; 16   :    system("pause");
89
90    0008f 8b f4        mov     esi, esp
91    00091 68 00 00 00 00   push    OFFSET ??_C@_05PDJBBECF@pause@
92    00096 ff 15 00 00 00
93      00       call    DWORD PTR __imp__system
94    0009c 83 c4 04     add     esp, 4
95    0009f 3b f4        cmp     esi, esp
96    000a1 e8 00 00 00 00   call    __RTC_CheckEsp
97
98  ; 17   : }
```

函数调用原理（汇编）

关于栈的描述

首先必须明确的一点是，栈是向下生长的。所谓向下生长，是指从内存高地址向低地址的路径延伸。于是，栈就有栈底和栈顶，栈顶的地址要比栈底的低。↵

对 x86 体系的 CPU 而言，寄存器 ebp 可称为帧指针或基址指针（base pointer），寄存器 esp 可称为栈指针（stack pointer）。↵

这里需要说明的几点如下。↵

（1）ebp 在未改变之前始终指向栈帧的开始（也就是栈底），所以 ebp 的用途是在堆栈中寻址（寻址的作用会在下面详细介绍）。↵

（2）esp 会随着数据的入栈和出栈而移动，即 esp 始终指向栈顶。↵

如图 2.2.1 所示，假设函数 A 调用函数 B，称函数 A 为调用者，称函数 B 为被调用者，则函数调用过程可以描述如下：↵

（1）首先将调用者（A）的堆栈的基址（ebp）入栈，以保存之前任务的信息。↵

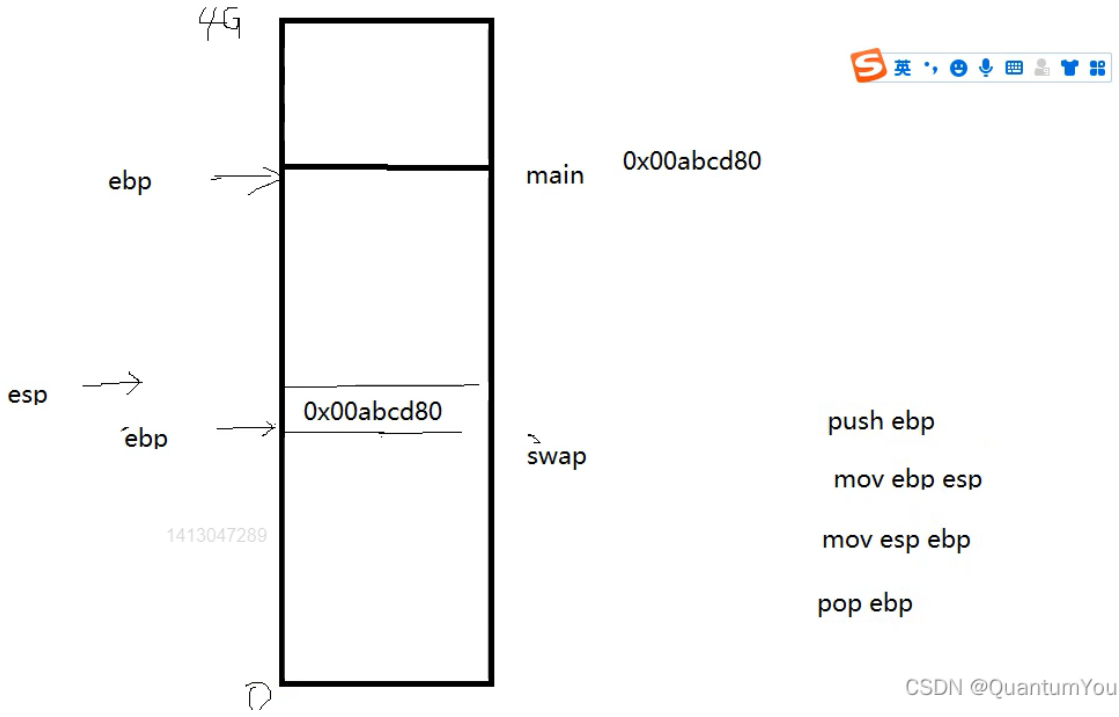
（2）然后将调用者（A）的栈顶指针（esp）的值赋给 ebp，作为新的基址（即被调用者 B 的栈底）。↵

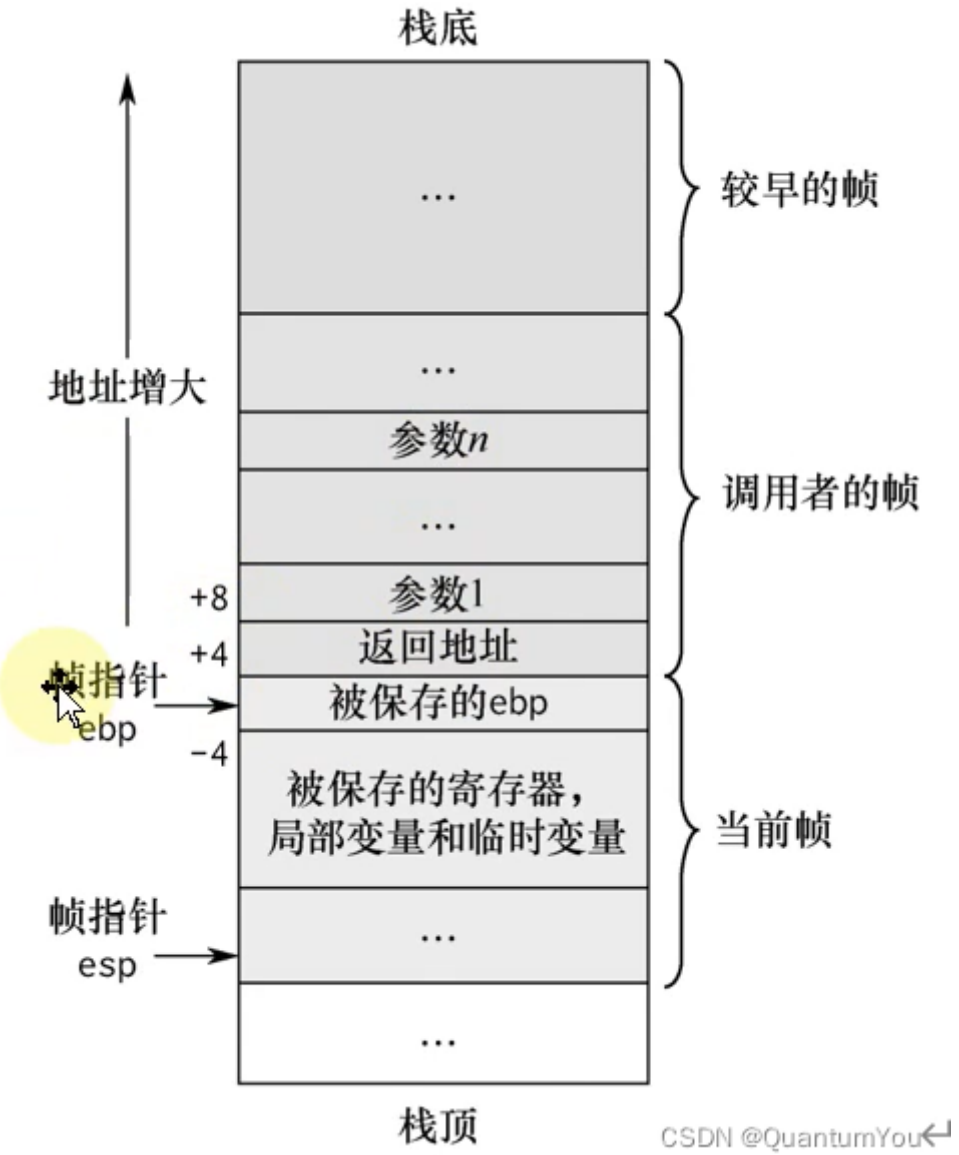
汇编代码如下：

```
  00000 55       push    ebp
  00001 8b ec        mov     ebp, esp
  00003 81 ec cc 00 00
    00       sub     esp, 204        ; 000000ccH
  00009 53       push    ebx
  0000a 56       push    esi
  0000b 57       push    edi
  0000c 8d bd 34 ff ff
    ff       lea     edi, DWORD PTR [ebp-204]
  00012 b9 33 00 00 00   mov     ecx, 51             ; 00000033H
  00017 b8 cc cc cc cc   mov     eax, -858993460         ; ccccccccH
  0001c f3 ab        rep stosd

; 5    :    int c;
; 6    :    c = *a;

  0001e 8b 45 08     mov     eax, DWORD PTR _a$[ebp]
  00021 8b 08        mov     ecx, DWORD PTR [eax]
  00023 89 4d f8     mov     DWORD PTR _c$[ebp], ecx

; 7    :    *a = *b;

  00026 8b 45 08     mov     eax, DWORD PTR _a$[ebp]
  00029 8b 4d 0c     mov     ecx, DWORD PTR _b$[ebp]
  0002c 8b 11        mov     edx, DWORD PTR [ecx]
  0002e 89 10        mov     DWORD PTR [eax], edx

; 8    :    *b = c;

  00030 8b 45 0c     mov     eax, DWORD PTR _b$[ebp]
  00033 8b 4d f8     mov     ecx, DWORD PTR _c$[ebp]
  00036 89 08        mov     DWORD PTR [eax], ecx

; 9    : }

```

```
00038 5f        pop     edi
00039 5e        pop     esi
0003a 5b        pop     ebx
0003b 8b e5         mov     esp, ebp
0003d 5d        pop     ebp
0003e c3        ret     0
```

函数调用不传参





Tips:

- 如下图

```
lea   eax, DWORD PTR _arr$[ebp]
```

lea 指令的作用，是 DWORD PTR _arr$[ebp]对应空间的内存地址值放到 eax 中

函数调用传参

指针的间接访问在汇编的体现

```
1  _TEXT   SEGMENT
2  _ret$ = -32                     ; size = 4
3  _b$ = -20                       ; size = 4
4  _a$ = -8                        ; size = 4
5  _main   PROC                        ; COMDAT
6
7  ; 12   : {
8
9    00000 55       push    ebp
10   00001 8b ec        mov     ebp, esp
11   00003 81 ec e4 00 00
```

```
    00       sub     esp, 228        ; 000000e4H
  00009 53       push    ebx
  0000a 56       push    esi
  0000b 57       push    edi
  0000c 8d bd 1c ff ff
    ff       lea     edi, DWORD PTR [ebp-228]
  00012 b9 39 00 00 00   mov     ecx, 57             ; 00000039H
  00017 b8 cc cc cc cc   mov     eax, -858993460         ; ccccccccH
  0001c f3 ab        rep stosd

; 13   :    int a,b,ret;
; 14   :    a =16;

  0001e c7 45 f8 10 00
    00 00        mov     DWORD PTR _a$[ebp], 16 ; 00000010H

; 15   :    b = 64;

  00025 c7 45 ec 40 00
    00 00        mov     DWORD PTR _b$[ebp], 64 ; 00000040H

; 16   :    ret = 0;

  0002c c7 45 e0 00 00
    00 00        mov     DWORD PTR _ret$[ebp], 0

; 17   :    swap(&a,&b);

  00033 8d 45 ec     lea     eax, DWORD PTR _b$[ebp]
  00036 50       push    eax
  00037 8d 4d f8     lea     ecx, DWORD PTR _a$[ebp]
  0003a 51       push    ecx
  0003b e8 00 00 00 00   call    _swap
  00040 83 c4 08     add     esp, 8

; 18   :    ret = a - b;

  00043 8b 45 f8     mov     eax, DWORD PTR _a$[ebp]
  00046 2b 45 ec     sub     eax, DWORD PTR _b$[ebp]
  00049 89 45 e0     mov     DWORD PTR _ret$[ebp], eax

; 19   :    printf("ret=%d\n",ret);

  0004c 8b f4        mov     esi, esp
  0004e 8b 45 e0     mov     eax, DWORD PTR _ret$[ebp]
  00051 50       push    eax
  00052 68 00 00 00 00   push    OFFSET ??_C@_07EGPJDCKD@ret?$DN?$CFd?6?
$AA@
  00057 ff 15 00 00 00
    00       call    DWORD PTR __imp__printf
  0005d 83 c4 08     add     esp, 8
  00060 3b f4        cmp     esi, esp
  00062 e8 00 00 00 00   call    __RTC_CheckEsp

; 20   :    system("pause");

  00067 8b f4        mov     esi, esp
  00069 68 00 00 00 00   push    OFFSET ??_C@_05PDJBBECF@pause?$AA@
```

```
  0006e ff 15 00 00 00
    00       call    DWORD PTR __imp__system
  00074 83 c4 04     add     esp, 4
  00077 3b f4        cmp     esi, esp
  00079 e8 00 00 00 00   call    __RTC_CheckEsp

; 21   :    return ret;

  0007e 8b 45 e0     mov     eax, DWORD PTR _ret$[ebp]

; 22   : }

  00081 52       push    edx
  00082 8b cd        mov     ecx, ebp
  00084 50       push    eax
  00085 8d 15 00 00 00
    00       lea     edx, DWORD PTR $LN6@main
  0008b e8 00 00 00 00   call    @_RTC_CheckStackVars@8
  00090 58       pop     eax
  00091 5a       pop     edx
  00092 5f       pop     edi
  00093 5e       pop     esi
  00094 5b       pop     ebx
  00095 81 c4 e4 00 00
    00       add     esp, 228       ; 000000e4H
  0009b 3b ec        cmp     ebp, esp
  0009d e8 00 00 00 00   call    __RTC_CheckEsp
  000a2 8b e5        mov     esp, ebp
  000a4 5d       pop     ebp
  000a5 c3       ret     0
  000a6 8b ff        npad    2
$LN6@main:
  000a8 02 00 00 00  DD  2
  000ac 00 00 00 00  DD  $LN5@main
$LN5@main:
  000b0 f8 ff ff ff  DD  -8         ; fffffff8H
  000b4 04 00 00 00  DD  4
  000b8 00 00 00 00  DD  $LN3@main
  000bc ec ff ff ff  DD  -20            ; ffffffecH
  000c0 04 00 00 00  DD  4
  000c4 00 00 00 00  DD  $LN4@main
$LN4@main:
  000c8 62       DB  98          ; 00000062H
  000c9 00       DB  0
$LN3@main:
  000ca 61       DB  97          ; 00000061H
  000cb 00       DB  0
_main   ENDP
_TEXT   ENDS
; Function compile flags: /Odtp /RTCsu /ZI
; File g:\code_2021\»ã±à½²½â\»ã±à½²½â2\main.c
;   COMDAT _swap
_TEXT   SEGMENT
_c$ = -8                        ; size = 4
_a$ = 8                         ; size = 4
_b$ = 12                        ; size = 4
_swap   PROC                        ; COMDAT

```

```
; 4    : {

  00000 55       push    ebp
  00001 8b ec        mov     ebp, esp
  00003 81 ec cc 00 00
    00       sub     esp, 204       ; 000000ccH
  00009 53       push    ebx
  0000a 56       push    esi
  0000b 57       push    edi
  0000c 8d bd 34 ff ff
    ff       lea     edi, DWORD PTR [ebp-204]
  00012 b9 33 00 00 00   mov     ecx, 51            ; 00000033H
  00017 b8 cc cc cc cc   mov     eax, -858993460        ; ccccccccH
  0001c f3 ab        rep stosd

; 5    :    int c;
; 6    :    c = *a;

  0001e 8b 45 08     mov     eax, DWORD PTR _a$[ebp]
  00021 8b 08        mov     ecx, DWORD PTR [eax]
  00023 89 4d f8     mov     DWORD PTR _c$[ebp], ecx

; 7    :    *a = *b;

  00026 8b 45 08     mov     eax, DWORD PTR _a$[ebp]
  00029 8b 4d 0c     mov     ecx, DWORD PTR _b$[ebp]
  0002c 8b 11        mov     edx, DWORD PTR [ecx]
  0002e 89 10        mov     DWORD PTR [eax], edx

; 8    :    *b = c;

  00030 8b 45 0c     mov     eax, DWORD PTR _b$[ebp]
  00033 8b 4d f8     mov     ecx, DWORD PTR _c$[ebp]
  00036 89 08        mov     DWORD PTR [eax], ecx

; 9    : }

  00038 5f       pop     edi
  00039 5e       pop     esi
  0003a 5b       pop     ebx
  0003b 8b e5        mov     esp, ebp
  0003d 5d       pop     ebp
  0003e c3       ret     0
_swap   ENDP
_TEXT   ENDS
END
```