Question 2

（1）sizeof(a)=0，因为数组元素的大小是数组中元素个数乘以每个元素大小

（2）140727892865392

编译器gcc9.3.0 Ubuntu20.04

加上注释的汇编代码

```
main:
.LFB0:
    .cfi_startproc
    endbr64
    pushq   %rbp ;保存旧的栈帧指针
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp ;获取新的栈帧指针
    .cfi_def_cfa_register 6
    subq    $32, %rsp ;分配栈存储空间
    movq    %fs:40, %rax ;存储返回值到返回值寄存器中
    movq    %rax, -8(%rbp) ;返回值压栈
    xorl    %eax, %eax ;xor this is to clear %eax
    movq    $4, -32(%rbp) ;load the value of i into stack
    movq    $8, -24(%rbp) ;load the value of j into stack
    movq    -16(%rbp), %rax ;-16(%rbq)store the array first element
    movq    %rax, %rdx ;The Third arguement of printf
    movl    $0, %esi ;The second arguement of printf
    leaq    .LC0(%rip), %rdi ;The first parameter of printf
    movl    $0, %eax ;%eax store the return value
    call    printf@PLT
    movl    $0, %eax
    movq    -8(%rbp), %rcx
    xorq    %fs:40, %rcx
    je  .L3
    call    __stack_chk_fail@PLT
```

Question 3

```
        .file   "ex7-9.c"
        .text
        .globl  main
        .type   main, @function
    main:
    .LFB0:
    ;-8(%rbp) i
    ;-4(%rbp) j
        pushq   %rbp ;将老的堆栈指针压栈
        movq    %rsp, %rbp ;获取新的堆栈指针
        jmp     .L2 ;无条件跳转到L2
    .L5:
        movl    -4(%rbp), %eax ;%eax=j
        movl    %eax, -8(%rbp) ;i=%eax
    .L2:
```

```
        cmpl    $0, -8(%rbp) ;compare and judge the value i || j
        jne     .L3 ;if i==true than judge whether j>5
        cmpl    $0, -4(%rbp) ;comapre j and 0
        je      .L4 ;if j==0 than the condition is false
    .L3:
        cmpl    $5, -4(%rbp); ;comapre j and 5
        jg      .L5; j>5
    .L4:
        movl    $0, %eax ;load 0 to %eax(store the return value)
        popq    %rbp ;恢复现场
        ret
    .LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu 7.5.0-3ubuntu1~16.04) 7.5.0"
```

对于条件判断

```
while ( (i || j) && (j > 5) )
```

先判断i是否为真，为真则条件成立，否则进入下一个判断

接着判断j是否为假，为假则条件为假，返回

j为假，再来判断j>5是否成立

这就是短路判断的过程

Question 4

(1)可能的原因是，对于某些编译器，将程序中定义的常量放在的内存区域并不是标记为不可修改的，而且在这种例子中，两个指针指向的内存区域正好重叠，strcpy其实是内存之间值的拷贝，这样就覆盖了cp2指向的内存空间

(2)编译器将两个字符串的内容存放在常量数据区，修改常量内存单元的内容会报错

Question 5

第六行的代码打印出的是函数funcold形式参数的地址，下一行打印的是局部变量的地址

第12行打印的是函数形参的地址，下一行是局部变量的地址

还有地址的问题，在Linux/x86_64系统上，short一般是2个字节，float是4个字节，这点从7、13行打印函数内部局部变量的地址就可以看出来

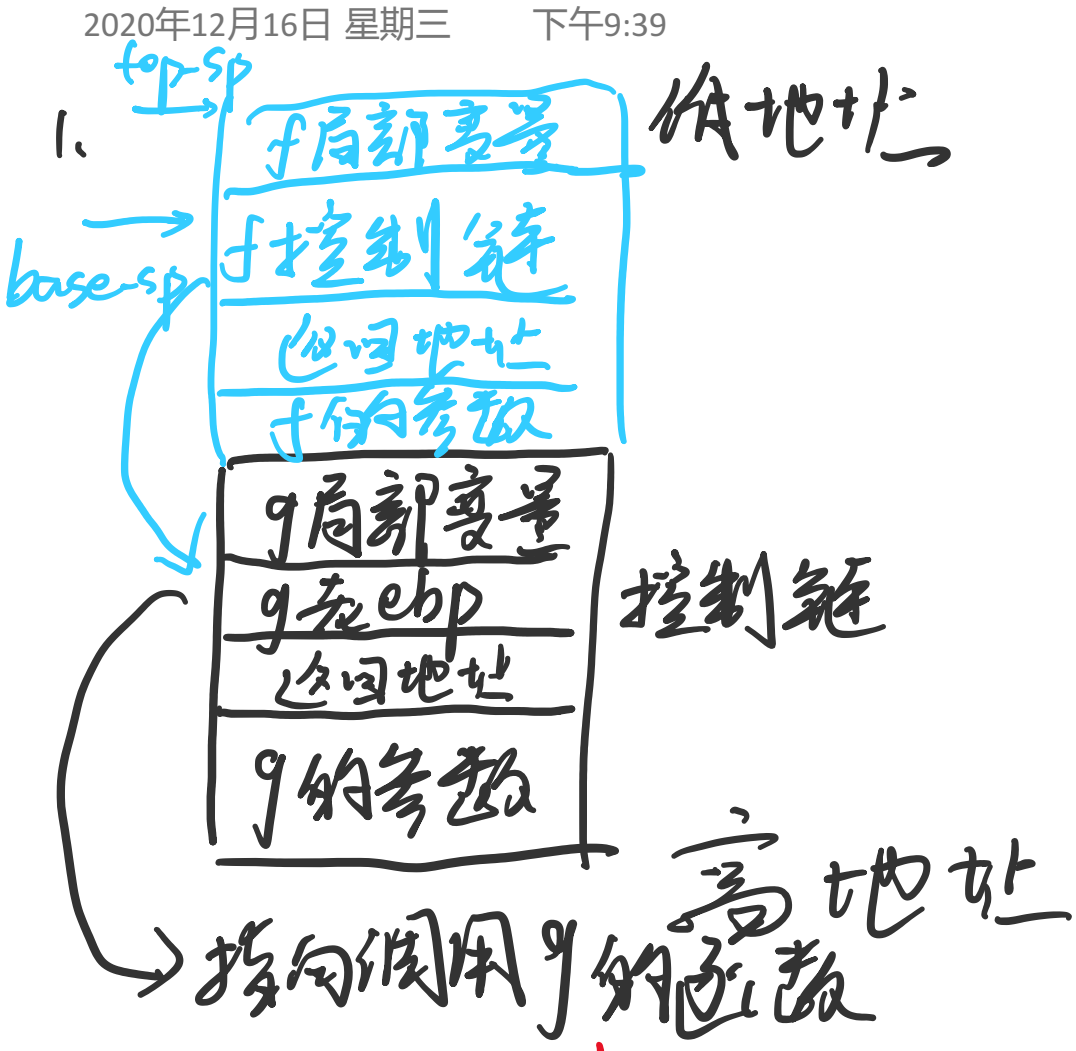
但是传递参数的时候，我们可以看到，`funcold`中short形参占据了4个字节，float也占用了4个字节

这是因为按照`funcold`定义函数的时候，没有指明参数的类型，所以函数传参数的时候就传递k个字节的数据，k是参数的个数

第一题和第五题内存布局在后面统一说明

HW8运行时空间组织

2020年12月16日 星期三　　下午9:39

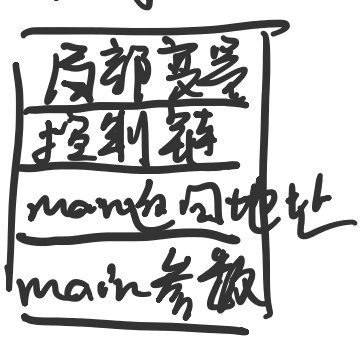


低地址　g调用f

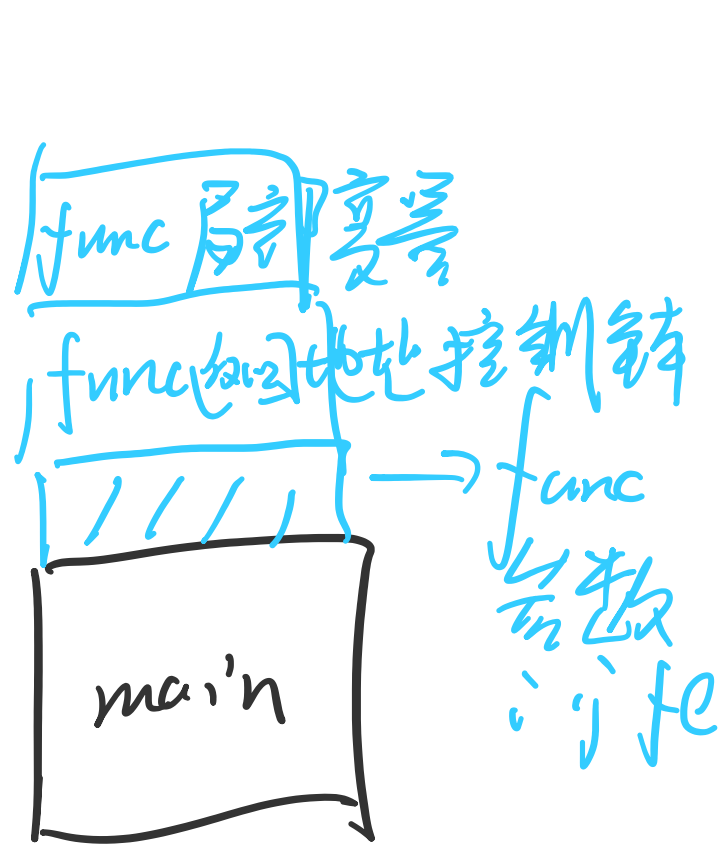
Question 5 内存布局

调用 func 之前

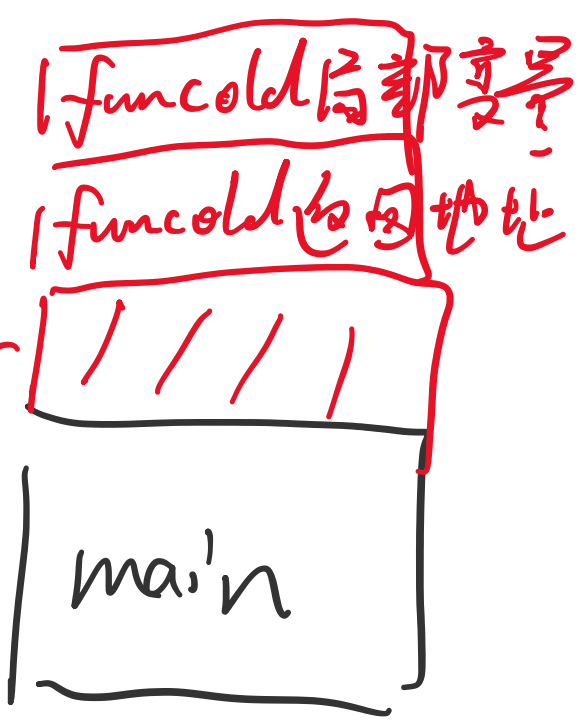
与调用 func old 之前



调用 func



调用 funcold



funcold参数

因为 func funcold参数

位置相同