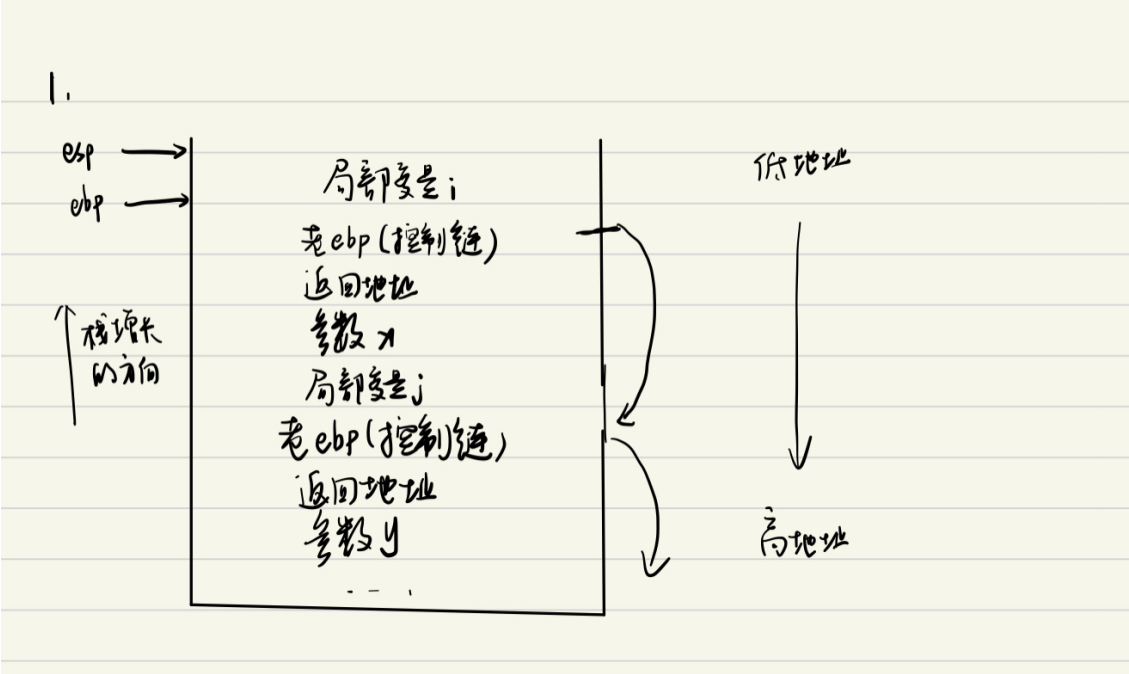
1. 


2. 教材6.18
   - sizeof(a)的值是0，理由：这可以根据数组size的计算公式直接得到，因为`a[0][4]`可以看出第一维维度为0，即$0 * 4 * sizeof(long) = 0$
   - `a[0][0]`的值是随机的。

```
micheallei@micheallei-TUF-GAMING-FX504GE-FX80GE:~/snap/Compiler/arm_lab » ./lear
n
0, -1030308080
micheallei@micheallei-TUF-GAMING-FX504GE-FX80GE:~/snap/Compiler/arm_lab » ./lear
n
0, -537947648
micheallei@micheallei-TUF-GAMING-FX504GE-FX80GE:~/snap/Compiler/arm_lab » ./lear
n
0, 1605025776
micheallei@micheallei-TUF-GAMING-FX504GE-FX80GE:~/snap/Compiler/arm_lab » ./lear
n
0, 277550672
micheallei@micheallei-TUF-GAMING-FX504GE-FX80GE:~/snap/Compiler/arm_lab »
```

     理由：由汇编码可以看出，机器为数组a实际分配了栈空间，`a[0][0]`的地址与a的起始地址是一致的，在-16(%rbp)处。并且该空间的内容并未被初始化，所以每次运行时的值都是一个未知数。

```
    movq    $4, -32(%rbp);i=4
    movq    $8, -24(%rbp);j=8
    movq    -16(%rbp), %rax;此处为a的地址
```

   - gcc版本

     `gcc (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0`

   - 汇编码

```
    .file   "learn.c"
    .text
    .section    .rodata
.LC0:
    .string "%ld, %d\n"
```

```
    .text
    .globl  main
    .type   main, @function
main:
.LFB0:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $32, %rsp
    movq    %fs:40, %rax
    movq    %rax, -8(%rbp)
    xorl    %eax, %eax
    movq    $4, -32(%rbp);i=4
    movq    $8, -24(%rbp);j=8
    movq    -16(%rbp), %rax;此处为a的地址
    movq    %rax, %rdx
    movl    $0, %esi
    leaq    .LC0(%rip), %rdi
    movl    $0, %eax
    call    printf@PLT
    movl    $0, %eax
    movq    -8(%rbp), %rcx
    xorq    %fs:40, %rcx
    je  .L3
    call    __stack_chk_fail@PLT
.L3:
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE0:
    .size   main, .-main
    .ident  "GCC: (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0"
    .section    .note.GNU-stack,"",@progbits
```

3. 教材7.9
   - 汇编码及注释

```
     .file   "ex7-9.c"
     .text
     .globl  main
     .type   main, @function
main:
.LFB0:
     pushq   %rbp
     movq    %rsp, %rbp
     jmp     .L2
.L5:
     movl    -4(%rbp), %eax;j的值装入eax
     movl    %eax, -8(%rbp);i=j，即把eax内值移入i的内存地址
.L2:
     cmpl    $0, -8(%rbp);计算i的布尔值
     jne     .L3;i!=0转L3，不用再计算j的布尔值
```

```
    cmpl    $0, -4(%rbp);计算j的布尔值
    je      .L4;假转L4，此时已确定(i || j)为假，退出循环
.L3:
    cmpl    $5, -4(%rbp);计算j>5的布尔值
    jg      .L5;真转L5，执行while循环体内语句
.L4:
    movl    $0, %eax
    popq    %rbp
    ret
.LFE0:
    .size   main, .-main
    .ident  "GCC: (Ubuntu 7.5.0-3ubuntu1~16.04) 7.5.0"
```

- 由汇编码注释可以得知，确实是用短路计算方式来完成布尔表达式计算的。

4. 对于该c程序

    1. 为什么cp2所指的串被修改了?

        两个字符串常量在内存中(.data区)连续存放，当strcpy把cp2指向的字符串复制给cp1时，超过了cp1原本的大小，导致超出部分"ghij\0"占据了原来cp2对应字符串的前面5个字符。这样输出cp2所指向的字符串时输出的是ghij

    2. 在某些系统上运行会输出段错误，为什么?

        某些系统的编译器会把程序中的字符串常量单独分配在一个段(一般在rodata只读数据段)，与其他常数分开，即该段数据在程序运行时不能被修改，所以在执行串复制strcpy时，会报告段错误

    注:`char *strcpy(char *dest, const char *src);`

    The strcpy() function copies the string pointed to by src, including the terminating null byte('\0'), to the buffer pointed to by dest

5. 首先在本机上编译并运行了代码

    - 注意到funcOld的函数声明是一种古老的K&R风格，与一般的ANSI风格有一定区别。参考汇编码可以发现，funcOld和func函数里第二个printf代码是完全一致的，区别仅仅在第一个printf函数的参数操作。
        - 查资料发现对K&R方式声明的函数，在向其传递参数时，较小类型的参数会被进行隐式类型转换，如char、short被转换为int，float被转换为double。即堆栈中所存储的参数其所占字节数大于实际应该占用的字节。

            其参数在使用时，会先按被隐式类型转换之后的大小，从堆栈中提取出来，然后再按函数定义中的实际类型进行截取。如，char类型变量c实际访问时，先在堆栈中变量c的存储位置，提取出一个int大小的“临时变量”，然后将该“临时变量”截取成一个char变量再进行运算。short及float变量同理。
        - 而对ANSI C的函数声明的函数，在向其传递参数时，不会发生隐式类型转换，堆栈中各个参数所占字节就是各个类型的实际应该占用的字节。
    - 而参考汇编码发现确实如此

        main函数里调用funcOld前对float类型两个变量的类型提升（float到double）

        ```
        cvtss2sd    -4(%rbp), %xmm1
        cvtss2sd    -8(%rbp), %xmm0
        ```

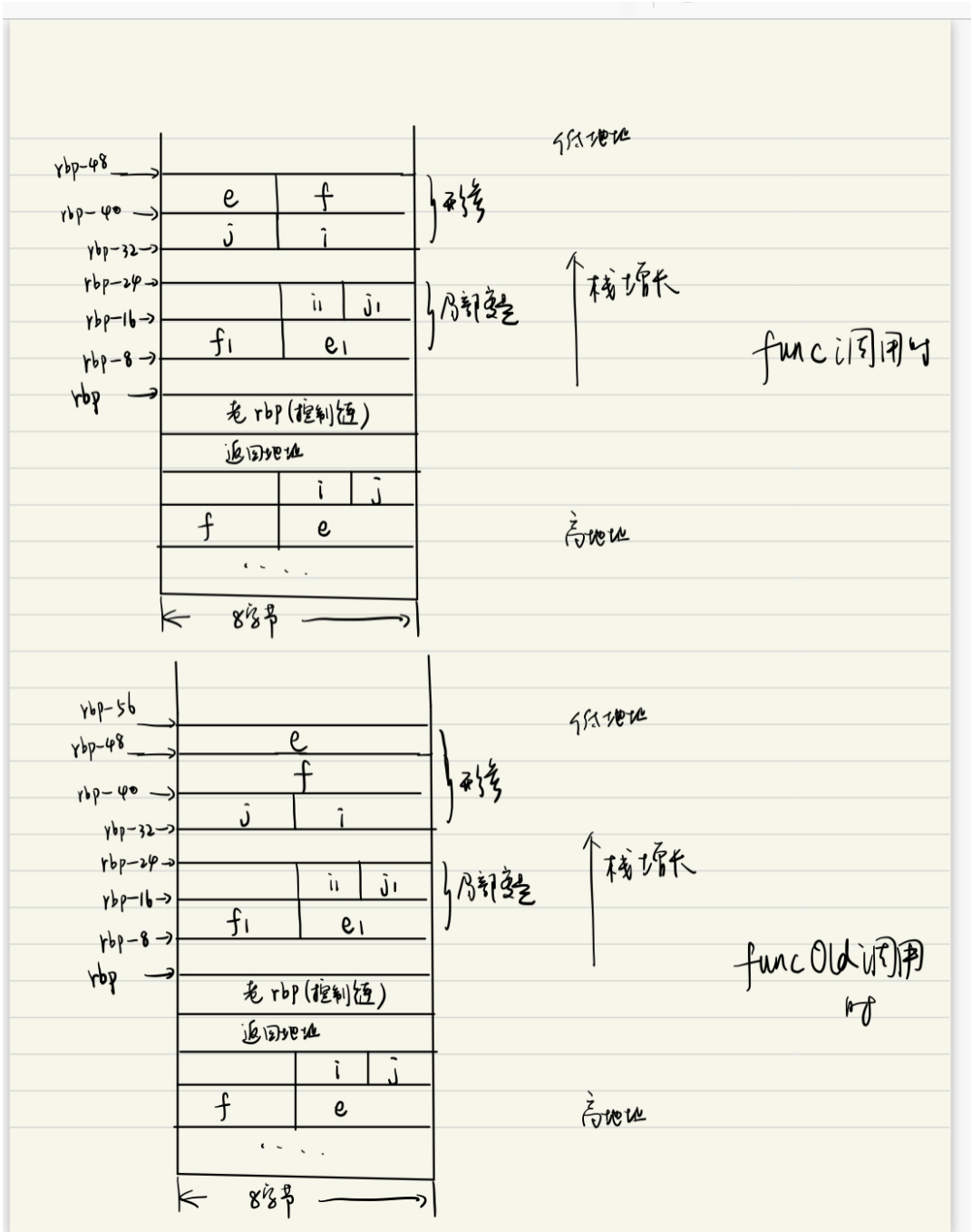
        funcOld里的四个参数的地址如下

```
leaq    -56(%rbp), %rsi   ;8字节
leaq    -48(%rbp), %rcx ;8字节
leaq    -40(%rbp), %rdx  ;4字节
leaq    -36(%rbp), %rax ;4字节
```

func函数里则是均为4字节

```
leaq    -48(%rbp), %rsi
leaq    -44(%rbp), %rcx
leaq    -40(%rbp), %rdx
leaq    -36(%rbp), %rax
```

- 再结合输出的地址信息，可画出各个形参和局部变量的内存布局(由生成的汇编码可发现，main里传参是通过寄存器实现的)




- 从而也可解释程序的输出结果：地址间隔有区别的原因在于
  - 对形参i,j,f,e，在栈上从高地址到低地址依次分配
    - funcOld里形参i,j为4字节；e，f为8字节

        - func里形参均为4字节
    - 对局部变量i1,j1,f1,e1,在栈上是按声明顺序从低地址到高地址分配空间
        - 无论funcOld还是func，为i1,j1(short)分配2字节；为e1,f1(float)分配4字节
    - 所以回到程序输出，发现与分析一致

```
0x7ffeb312ba8c, 0x7ffeb312ba88, 0x7ffeb312ba84, 0x7ffeb312ba80
0x7ffeb312ba9c, 0x7ffeb312ba9e, 0x7ffeb312baa0, 0x7ffeb312baa4
0x7ffeb312ba8c, 0x7ffeb312ba88, 0x7ffeb312ba80, 0x7ffeb312ba78
0x7ffeb312ba9c, 0x7ffeb312ba9e, 0x7ffeb312baa0, 0x7ffeb312baa4
```

        - 第一行地址依次减小，彼此间相差4字节
        - 第二、四行地址依次增大，彼此相差2,2,4字节
        - 第三行地址依次减小，彼此相差4,8,8字节

- 附上汇编码,运行环境见汇编码倒数第二行

```
    .file   "5.c"
    .text
    .section    .rodata
.LC0:
    .string "%p, %p, %p, %p\n"
    .text
    .globl  funcOld
    .type   funcOld, @function
funcOld:
.LFB0:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $64, %rsp
    movl    %edi, %edx
    movl    %esi, %eax
    movw    %dx, -36(%rbp)
    movw    %ax, -40(%rbp)
    cvtsd2ss    %xmm0, %xmm0
    movss   %xmm0, -48(%rbp)
    cvtsd2ss    %xmm1, %xmm0
    movss   %xmm0, -56(%rbp)
    movq    %fs:40, %rax
    movq    %rax, -8(%rbp)
    xorl    %eax, %eax
    leaq    -56(%rbp), %rsi
    leaq    -48(%rbp), %rcx
    leaq    -40(%rbp), %rdx
    leaq    -36(%rbp), %rax
    movq    %rsi, %r8
    movq    %rax, %rsi
    leaq    .LC0(%rip), %rdi
    movl    $0, %eax
    call    printf@PLT
    leaq    -12(%rbp), %rsi
    leaq    -16(%rbp), %rcx
    leaq    -18(%rbp), %rdx
```

```
    leaq    -20(%rbp), %rax
    movq    %rsi, %r8
    movq    %rax, %rsi
    leaq    .LC0(%rip), %rdi
    movl    $0, %eax
    call    printf@PLT
    nop
    movq    -8(%rbp), %rax
    xorq    %fs:40, %rax
    je  .L2
    call    __stack_chk_fail@PLT
.L2:
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE0:
    .size   funcOld, .-funcOld
    .globl  func
    .type   func, @function
func:
.LFB1:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $48, %rsp
    movl    %edi, %edx
    movl    %esi, %eax
    movss   %xmm0, -44(%rbp)
    movss   %xmm1, -48(%rbp)
    movw    %dx, -36(%rbp)
    movw    %ax, -40(%rbp)
    movq    %fs:40, %rax
    movq    %rax, -8(%rbp)
    xorl    %eax, %eax
    leaq    -48(%rbp), %rsi
    leaq    -44(%rbp), %rcx
    leaq    -40(%rbp), %rdx
    leaq    -36(%rbp), %rax
    movq    %rsi, %r8
    movq    %rax, %rsi
    leaq    .LC0(%rip), %rdi
    movl    $0, %eax
    call    printf@PLT
    leaq    -12(%rbp), %rsi
    leaq    -16(%rbp), %rcx
    leaq    -18(%rbp), %rdx
    leaq    -20(%rbp), %rax
    movq    %rsi, %r8
    movq    %rax, %rsi
    leaq    .LC0(%rip), %rdi
    movl    $0, %eax
    call    printf@PLT
    nop
    movq    -8(%rbp), %rax
```

```
    xorq    %fs:40, %rax
    je  .L4
    call    __stack_chk_fail@PLT
.L4:
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE1:
    .size   func, .-func
    .globl  main
    .type   main, @function
main:
.LFB2:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $32, %rsp
    movswl  -12(%rbp), %edx
    movswl  -10(%rbp), %eax
    movss   -4(%rbp), %xmm0
    movl    -8(%rbp), %ecx
    movaps  %xmm0, %xmm1
    movl    %ecx, -20(%rbp)
    movss   -20(%rbp), %xmm0
    movl    %edx, %esi
    movl    %eax, %edi
    call    func
    cvtss2sd    -4(%rbp), %xmm1
    cvtss2sd    -8(%rbp), %xmm0
    movswl  -12(%rbp), %edx
    movswl  -10(%rbp), %eax
    movl    %edx, %esi
    movl    %eax, %edi
    movl    $2, %eax
    call    funcOld
    movl    $0, %eax
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE2:
    .size   main, .-main
    .ident  "GCC: (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0"
    .section    .note.GNU-stack,"",@progbits
```

注:

```
Movswl    传送做了符号扩展字到双字
Movl                                传送双字(32bit)
Movw                               传送字（16bit）
movss                       32位数据的移动
movq                        4字(64bit)
movb             传送8bit

cvtss2sd    单精度转双精度
leaq:  根据括号里的源操作数来计算地址，然后把地址加载到目标寄存器中
```