

# Machine Programming-II

元培数科 王恩博

Let's get started →

2025/10/15



# 小班安排更新

周次	日期	大班课	展示同学	日期	对应大班节数	日期	大班课	展示同学
	周一	5~6节		周三	10~11节	周四	5~6节	
第1周	9月8日	1		9月10日	1	9月11日	2	龚鹤宁 陈泽羽
第2周	9月15日	3	孔令珊 刘筱	9月17日	2	9月18日	4	张哲涵 黎祖含
第3周	9月22日	5	陈俊宇 胡勋炀	9月24日	3/4	9月25日	6	黎宣萱 李準珩
第4周	9月29日	7	陈俊言 胡思成	10月1日	国庆放假	10月2日	国庆放假	
第5周	10月6日	国庆放假		10月8日	国庆放假	10月9日	8	
第6周	10月13日	9	阶段测验1 (第1~8课)	10月15日	5/6/7	10月16日	10	王欣语 曾若天
第7周	10月20日	11	龚鹤宁 陈泽羽	10月22日	10	10月23日	12	孔令珊 刘筱
第8周	10月27日	13	张哲涵 黎祖含	10月29日	11/12	10月30日	14	陈俊宇 胡勋炀
第9周	11月3日	15	黎宣萱 李準珩	11月5日	13/14	11月6日	16	陈俊言 胡思成
第10周	11月10日	17	王欣语 曾若天	11月12日	15/16	11月13日	18	
第11周	11月17日	19	阶段测验2 (第10~18课)	11月19日	17/18	11月20日	20	
第12周	11月24日	21		11月26日	20	11月27日	22	
第13周	12月1日	23		12月3日	21/22	12月4日	24	
第14周	12月8日	25	待定	12月10日	23/24	12月11日	26	
第15周	12月15日	27	LAB测验 (L1~L7)	12月17日	25/26	12月18日	28	待定
第16周	12月22日	29		12月24日	28	12月25日	30	
考试周	12月29日		期末考试 (周一下午)					

# 条件码 CC

- 进位标志 **CF** (carry flag) : 无符号溢出 (无论上溢还是下溢)
- 零标志 **ZF** (zero flag) : 零
- 符号标志 **SF** (sign flag) : 负数
- 溢出标志 **OF** (overflow flag) : 有符号溢出
- leaq 不会设置任何条件码
- **逻辑操作**会置零 CF 和 OF (**无溢出**)
- **移位操作**会置零 OF, 根据最后一个移出的位改变 CF (无论左移还是右移)
- INC 和 DEC 会改变 ZF 和 OF, 但是**不会改变 CF**

CF	$(\text{unsigned}) t < (\text{unsigned}) a$	Unsigned overflow
ZF	$(t == 0)$	Zero
SF	$(t < 0)$	Negative
OF	$(a < 0 == b < 0) \&\& (t < 0 != a < 0)$	Signed overflow

# 设置条件码

## 直接设置条件码

指令	全称	功能
CLC	clear carry flag	CF清零
STC	set carry flag	CF置位1
CMC	complement carry flag	CF取反
CLD	clear direction flag	DF清零
STD	set direction flag	DF置位1
CLI	clear interrupt endable flag	IF清零, 关闭中断
STI	set interrupt endable flag	IF置位1, 打开中断

# 间接设置条件码

- cmp/test 用于设置 CC, 后缀 b/w/l/q, 行为等同 sub/and

- testq %rax %rax (零测试)
- testq %rax \$MASK (掩码测试)

Instruction		Based on	Description
CMP	$S_1, S_2$	$S_2 - S_1$	Compare
cmpb			Compare byte
cmpw			Compare word
cmpl			Compare double word
cmpq			Compare quad word
TEST	$S_1, S_2$	$S_1 \& S_2$	Test
testb			Test byte
testw			Test word
testl			Test double word
testq			Test quad word

# 访问条件码

- 任意后缀对应的条件码要求熟练掌握
- 不用背但要能快速写出【理解】

Instruction	Synonym	Effect	Set condition
sete $D$	setz	$D \leftarrow ZF$	Equal / zero
setne $D$	setnz	$D \leftarrow \sim ZF$	Not equal / not zero
sets $D$		$D \leftarrow SF$	Negative
setns $D$		$D \leftarrow \sim SF$	Nonnegative
setg $D$	setnle	$D \leftarrow \sim (SF \wedge OF) \& \sim ZF$	Greater (signed >)
setge $D$	setnl	$D \leftarrow \sim (SF \wedge OF)$	Greater or equal (signed $\geq$ )
setl $D$	setnge	$D \leftarrow SF \wedge OF$	Less (signed <)
setle $D$	setng	$D \leftarrow (SF \wedge OF) \mid ZF$	Less or equal (signed $\leq$ )
seta $D$	setnbe	$D \leftarrow \sim CF \& \sim ZF$	Above (unsigned >)
setae $D$	setnb	$D \leftarrow \sim CF$	Above or equal (unsigned $\geq$ )
setb $D$	setnae	$D \leftarrow CF$	Below (unsigned <)
setbe $D$	setna	$D \leftarrow CF \mid ZF$	Below or equal (unsigned $\leq$ )

**Figure 3.14 The SET instructions.** Each instruction sets a single byte to 0 or 1 based on some combination of the condition codes. Some instructions have “synonyms,” that is, alternate names for the same machine instruction.

# 条件跳转-jmp

- 直接跳转 jmp .L1
- 间接跳转 jmp \*%rax 或 jmp \*(%rax)
- 条件跳转只能是直接跳转
- jmp 的二进制编码为相对寻址
  - 如右图, jmp 8 的 8 对应的编码为 03, 因为第二行结束时 PC 为 05, 与 08 相差 03, jmp 5 的 5 对应的编码为 f8, 因为在第五行结束时 PC 为 0d, 与 05 相差 f8 (-8 的十六进制编码)
- 绝对寻址 (只在大程序中使用, 如 > 2M)

Instruction	Synonym	Jump condition	Description
jmp <i>Label</i>		1	Direct jump
jmp * <i>Operand</i>		1	Indirect jump
je <i>Label</i>	jz	ZF	Equal / zero
jne <i>Label</i>	jnz	~ZF	Not equal / not zero
js <i>Label</i>		SF	Negative
jns <i>Label</i>		~SF	Nonnegative
jg <i>Label</i>	jnle	~(SF ^ OF) & ~ZF	Greater (signed >)
jge <i>Label</i>	jnl	~(SF ^ OF)	Greater or equal (signed >=)
jl <i>Label</i>	jnge	SF ^ OF	Less (signed <)
jle <i>Label</i>	jng	(SF ^ OF)   ZF	Less or equal (signed <=)
ja <i>Label</i>	jnbe	~CF & ~ZF	Above (unsigned >)
jae <i>Label</i>	jnb	~CF	Above or equal (unsigned >=)
jb <i>Label</i>	jnae	CF	Below (unsigned <)
jbe <i>Label</i>	jna	CF   ZF	Below or equal (unsigned <=)

**Figure 3.15 The jump instructions.** These instructions jump to a labeled destination when the jump condition holds. Some instructions have “synonyms,” alternate names for the same machine instruction.

# 条件传送-cmov

- 不能传送至内存，不支持单字节传送
- 会自动推断传送数据大小
- 无需预测判断结果，更高效
- 条件跳转可能被优化为条件传送，但以下情形不适用：
  - 表达式求值需要大量运算
  - 表达式求值可能导致错误，如  $\text{val} = p ? *p : 0$
  - 表达式求值可能导致副作用，如  $\text{val} = x > 0 ? (x *= 7) : (x += 3)$

Instruction	Synonym	Move condition	Description
cmove $S, R$	cmovez	ZF	Equal / zero
cmovne $S, R$	cmovnz	$\sim ZF$	Not equal / not zero
cmovs $S, R$		SF	Negative
cmovns $S, R$		$\sim SF$	Nonnegative
cmovg $S, R$	cmovnle	$\sim (SF \wedge OF) \& \sim ZF$	Greater (signed $>$ )
cmovge $S, R$	cmovnl	$\sim (SF \wedge OF)$	Greater or equal (signed $\geq$ )
cmovl $S, R$	cmovnge	SF $\wedge$ OF	Less (signed $<$ )
cmovle $S, R$	cmovng	$(SF \wedge OF) \mid ZF$	Less or equal (signed $\leq$ )
cmova $S, R$	cmovnbe	$\sim CF \& \sim ZF$	Above (unsigned $>$ )
cmovae $S, R$	cmovnb	$\sim CF$	Above or equal (Unsigned $\geq$ )
cmovb $S, R$	cmovnae	CF	Below (unsigned $<$ )
cmovbe $S, R$	cmovna	CF $\mid$ ZF	Below or equal (unsigned $\leq$ )

**Figure 3.18** The conditional move instructions. These instructions copy the source value  $S$  to its destination  $R$  when the move condition holds. Some instructions have “synonyms,” alternate names for the same machine instruction.

# 条件分支-switch

- 当 switch 分支数量较多且值的跨度范围较小时会启用跳转表，存放于 .rodata (只读数据区) 中，使用间接跳转
- 不是简单的 if-else，而是空间换时间的策略
- switch 源代码与汇编代码的对应
  - 没有 break 的 case 对应无 jmp 的 label
  - default 有另设 label
  - 多 case 同处理对应同 label 段

(a) Code

```

void switcher(long a, long b, long c, long *dest)
a in %rsi, b in %rdi, c in %rdx, d in %rcx
1    switcher:
2        cmpq    $7, %rdi
3        ja     .L2
4        jmp     *.L4(%rdi,8)
5        .section .rodata
6        .L7:
7        xorq    $15, %rsi
8        movq    %rsi, %rdx
9        .L3:
10       leaq    112(%rdx), %rdi
11       jmp     .L6
12       .L5:
13       leaq    (%rdx,%rsi), %rdi
14       salq    $2, %rdi
15       jmp     .L6
16       .L2:
17       movq    %rsi, %rdi
18       .L6:
19       movq    %rdi, (%rcx)
20       ret

```

(b) Jump table

```

1   .L4:
2   .quad   .L3
3   .quad   .L2
4   .quad   .L5
5   .quad   .L2
6   .quad   .L6
7   .quad   .L7
8   .quad   .L2
9   .quad   .L5

```

Figure 3.24 Assembly code and jump table for Problem 3.31.

# 循环语句

do-while/while/for

- 本质都是 goto 的应用
- for 大致等价于初始化 + while (需要仔细处理 continue)
- while 的两种转化，左边称为 jump to the middle，右边称为 **guarded-do**
  - 二者在语义上等价，不过后者可以优化初始的判断（少一次 jmp）

```

if (test-expr)           init-expr;
    then-statement
else                      while (test-expr) {
    else-statement          body-statement
                            update-expr;
}
t = test-expr;           init-expr;           init-expr;
if (!t)                  goto test;           t = test-expr;
    goto false;           if (!t)
    goto done;            goto done;
false:                   then-statement       loop:
    goto done;           body-statement
    update-expr;          update-expr;         loop:
                           body-statement
else-statement          test:                 body-statement
done:                    t = test-expr;        update-expr;
                        if (t)                t = test-expr;
                        goto loop;           if (t)
                                         goto loop;
                                         done;

```

# 过程

procedure

过程是软件设计中的重要抽象概念，它提供了一种封装代码的方式。通过指定的参数和可选的返回值来实现某个特定功能。

深入到机器层面，过程基于如下机制：

- **传递控制**：在进入过程 Q 的时候，程序计数器必须被设置为 Q 的代码的起始地址，然后在返回时，要把程序计数器设置为 **P 中调用 Q 后面那条指令** 的地址。
- **传递数据**：P 必须能够向 Q 提供一个或多个参数，Q 必须能够向 P 返回一个值。
- **分配和释放内存**：在开始时，Q 可能需要为局部变量分配空间，而在返回前，又必须释放这些存储空间。

# 运行时栈

runtime stack

C 语言过程调用依赖于运行时栈进行数据和指令管理。

- **过程栈帧**: 每次调用会分配一个新帧，保存局部变量、参数和返回地址。
- **栈帧管理**: 调用和返回过程中，栈帧通过压栈 `push` 和出栈 `pop` 操作管理数据。

栈帧不是必要的（只有寄存器不够用时，才会在栈上分配空间）

- **栈顶与栈底**: 通过调整栈顶指针 `%rsp` 和栈底指针 `%rbp` 来管理栈帧。

注意，栈顶在低地址，栈底在高地址，栈是向下增长的。

# x86-64 的栈结构

stack structure in x86-64

## 1. 栈帧布局：

- **参数构造区**: 存放函数构造的参数

看图的上面，参数 7~n 就是 P 的参数构造区，注意顺序

- **局部变量区**: 用于函数临时构造的局部变量。
- **被保存的寄存器**: 用于保存调用过程中使用到的寄存器状态

被调用者保存寄存器: %rbp %rbx %r12 %r13  
%r14 %r15

- **返回地址**: 调用结束时的返回地址

**返回地址属于调用者 P 的栈帧**

- **对齐**: 栈帧的地址必须是 16 的倍数

16 字节对齐，Attacklab 会用到哦

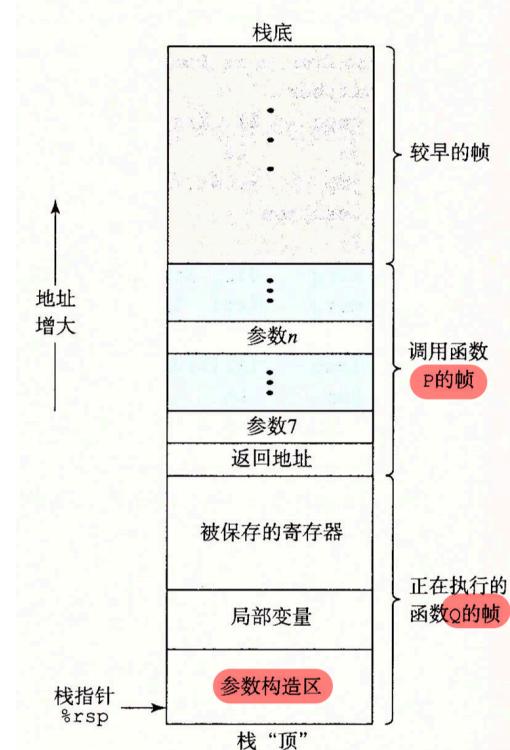


图 3-25 通用的栈帧结构(栈用来传递参数、存储返回信息、保存寄存器，以及局部存储。省略了不必要的部分)

# x86-64 的栈结构

stack structure in x86-64

2. **栈顶管理:** 使用 `push` 和 `pop` 指令进行数据的压栈和出栈管理。
3. **帧指针和栈指针:** 使用寄存器 `%rbp` 和 `%rsp` 定位和管理栈帧。

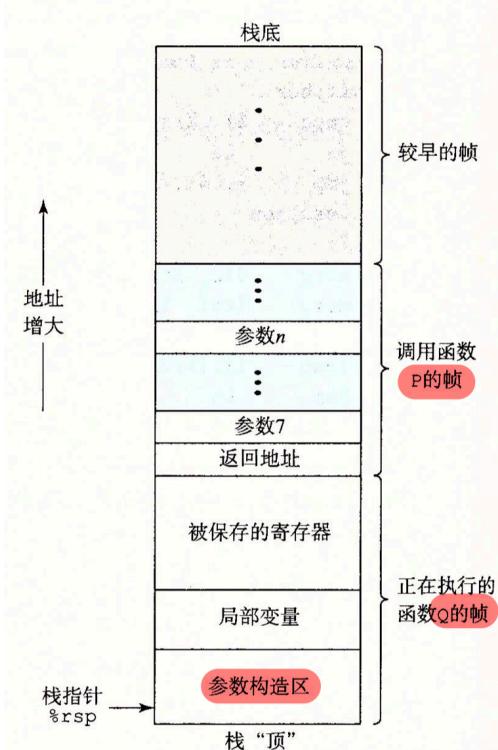


图 3-25 通用的栈帧结构(栈用来传递参数、存储返回信息、保存寄存器，以及局部存储。省略了不必要的部分)

# 转移控制

transfer control

转移控制是将程序的执行流程从一个函数跳转到另一个函数，并在完成任务后返回原函数。

- 指令层面，从函数 P 跳转到函数 Q，只需将程序计数器（PC）设置为 Q 的起始地址。
- 参数/数据层面，则需要通过栈帧 / 寄存器来传递。

# call 和 ret 指令

call and ret instructions

在x86-64体系中，这个转移过程通过指令 call Q 和 ret 来完成：

- call Q : 调用 Q 函数，并将返回地址压入栈，返回地址是 call 指令的下一条指令的地址。
- ret : 从栈中弹出压入的返回地址，并将 PC 设置为该地址。

这样，程序可以在函数间跳转，并能够正确返回。

## call 指令

- call Label : 直接调用，目标为标签地址
- call \*Operand : 间接调用，目标为寄存器或内存中的地址

## ret 指令

- 执行返回，将返回地址从栈中弹出并跳转

# call 和 ret 指令

call and ret instructions

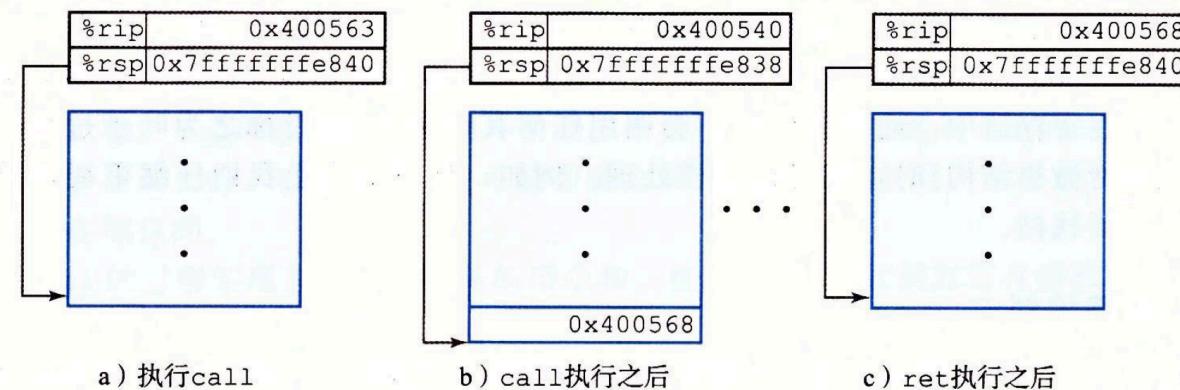


图 3-26 call 和 ret 函数的说明。call 指令将控制转移到一个函数的起始，而 ret 指令返回到这次调用后面的那条指令

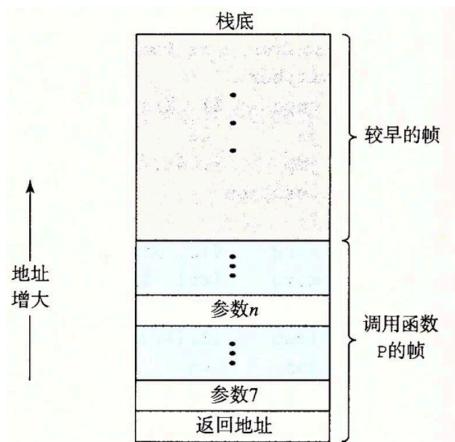
观察：

- 压栈后, %rsp -8, 压入的是 %rip 下一条指令地址
- 弹栈后, %rsp +8, 弹出的是栈帧中的内容, 和当前运行时的 %rip 无关

# 数据传送

data transfer

- 前 6 个参数：通过寄存器传递
  - %rdi %rsi %rdx %rcx %r8 %r9
- 剩余的参数：通过栈传递
  - 参数 7 在栈顶（低地址）
  - 参数构造区向 8 对齐



## C 代码

```
void proc(long a1, long *a1p, int a2, int *a2p,
short a3, short *a3p, char a4, char *a4p) {
    *a1p += a1;
    *a2p += a2;
    *a3p += a3;
    *a4p += a4;
}
```

## 生成的汇编代码

```
proc:
    movq 16(%rsp), %rax    # 取 a4p (64 位)
    addq %rdi, (%rsi)       # *a1p += a1 (64 位)
    addl %edx, (%rcx)       # *a2p += a2 (32 位)
    addw %r8w, (%r9)        # *a3p += a3 (16 位)
    movl 8(%rsp), %edx     # 取 a4 (8 位)
    addb %dl, (%rax)        # *a4p += a4 (8 位)
    ret
```

# 栈上的局部存储

local storage on the stack

有时，局部数据必须在内存中：

- 寄存器不够用
- 对一个局部变量使用地址运算符 `&` (因此必须能够为它产生一个地址，而不能放到寄存器里)
- 是数组或结构 (要求连续、要求能够被引用 `&` 访问到)

注意，生长方向与参数构造区相反！

```
long call_proc() {
    long x1 = 1; int x2 = 2;
    short x3 = 3; char x4 = 4;
    proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
    return (x1 + x2) * (x3 - x4);
}
```

```
movl $3, 18(%rsp)      # 将 3 存储在 &x3
movw $4, 17(%rsp)      # 将 4 存储在 &x4
leaq 17(%rsp), %rax   # 创建 &x4
movq %rax, 8(%rsp)     # 将 &x4 作为参数 8 存储
movl $4, (%rsp)        # 将 4 作为参数 7 存储
leaq 18(%rsp), %r9     # 将 &x3 作为参数 6
movl $3, %r8d           # 将 3 作为参数 5
leaq 20(%rsp), %rcx    # 将 &x2 作为参数 4
movl $2, %edx           # 将 2 作为参数 3
leaq 24(%rsp), %rsi    # 将 &x1 作为参数 2
movl $1, %edi           # 将 1 作为参数 1
```

```
# 调用 proc
call proc
```

```
# 从内存中检索更改
movslq 20(%rsp), %rdx  # 获取 x2 并转换为 long
addq 24(%rsp), %rdx    # 计算 x1 + x2
movswl 18(%rsp), %eax  # 获取 x3 并转换为 int
movsbl 17(%rsp), %ecx  # 获取 x4 并转换为 int
subl %ecx, %eax        # 计算 x3 - x4
cltq                   # 转换为 long
imulq %rdx, %rax       # 计算 (x1 + x2) * (x3 -
addq $32, %rsp          # 释放栈帧
ret                     # 返回
```

# 寄存器上的局部存储

local storage on registers

PRO

被调用者 / 保存

被调用者保存寄存器: %rbx %rbp %r12 %r13 %r14 %r15

其他寄存器, **再除外** %rsp, 均为“调用者保存”寄存器

CON

被 / 调用者保存

# Machine Data

# 指针运算

pointer arithmetic

注意步长！

指针（数组名也是指针）的加减，会乘以步长，其值是指针所代表数据类型的大小。

如 `int*` 加减的步长是 `int` 的大小，即 4

可以计算同一个数据结构中的两个指针之差，值等于两个地址之差 **除以该数据类型的大小**。

看最后一个示例

表达式	类型	值	汇编代码
$E$	<code>int*</code>	$x_E$	<code>movq %rdx, %rax</code>
$E[0]$	<code>int</code>	$M[x_E]$	<code>movl (%rdx), %rax</code>
$E[i]$	<code>int</code>	$M[x_E + 4i]$	<code>movl (%rdx), %rcx, 4, %eax</code>
$\&E[2]$	<code>int*</code>	$x_E + 8$	<code>leaq 8(%rdx), %rax</code>
$E + i - 1$	<code>int*</code>	$x_E + 4i - 4$	<code>leaq -4(%rdx, %rcx, 4), %rax</code>
$*(E + i - 3)$	<code>int</code>	$M[x_E + 4i - 12]$	<code>movl -12(%rdx, %rcx, 4), %eax</code>
$\&E[i] - E$	<code>long</code>	$i$	<code>movq %rcx, %rax</code>

# 数组分配和访问

array allocation and access

数组声明如：

```
T A[N];
```

其中  $T$  为数据类型， $N$  为整数常数。

初始化位置信息：

- 在内存中分配一个  $L \times N$  字节的 **连续区域**， $L$  为数据类型  $T$  的大小（单位为字节）。
- 引入标识符  $A$ ，可以通过指针  $x_A$  访问数组元素。

访问公式：

$$\&A[i] = x_A + L \cdot i$$

数组名是指针常量，指向数组的首地址。

# 数组分配和访问

array allocation and access

如下声明的数组：

```
char A[12];
char B[8];
int C[6];
double D[5];
```

这些声明会生成带有以下参数的数组：

	数组	元素大小	总的大小	起始地址	元素 $x[i]$ 的地址
A	char : 1	12	$x_A$	$x_A + i$	
B	char : 1	8	$x_B$	$x_B + i$	
C	int : 4	24	$x_C$	$x_C + 4i$	
D	double : 8	40	$x_D$	$x_D + 8i$	

# 数组分配和访问

array allocation and access

假设 `E` 是一个 `int` 型数组，其地址存放在寄存器 `%rdx` 中，`i` 存放在寄存器 `%rcx`，那么 `E[i]` 的汇编代码为：

```
movl (%rdx, %rcx, 4), %eax # (Start, Index, Step)
```

特别地，对于数组下标 `A[i]` 的计算，实际上是 `*(A+i)`，即 `A+i` 是一个指针，指向 `A` 的第 `i` 个元素。

## 嵌套数组

```
T D[R][C]; # Row, Column
```

数组元素 `D[i][j]` 的内存地址为（解的时候顺序从左到右）：

$$\&D[i][j] = x_D + L \cdot (C \cdot i + j)$$

# 解码复杂表达式

decode complex expression

1. 从变量名开始
2. 往右读直到读到右括号或到底
3. 往左，忽略读过的
4. 上面的括号均不包括成对的括号
5. `*` 读作“指针，指向”
6. `[x]` 读作“长为 x 的数组，元素类型为”
7. `(...)` 读作“函数，返回值为” + 上面提的成对括号  
(参数列表可选)
8. 如果为匿名类型，手动补充变量名 (参数列表里会出现)

```
int *(*p[2])[3];
```

1. `p` : 变量名
2. `[2]` : 往右读，这是一个长度为 2 的数组，元素类型为...
3. `)` : 往左读，忽略读过的
4. `(*p[2])` : 这是一个指针，指向...
5. `(*p[2])[3]` : 一个长度为 3 的数组
6. `;` : 到底了，往左读
7. `*(*p[2])[3]` : 一个指针，指向...
8. `int *(*p[2])[3]` : `int` 类型

合并：这是一个长度为 2 的数组，元素类型为指针，指向一个长度为 3 的数组，数组元素类型为指针，指向 `int`。

# 解码复杂表达式

decode complex expression

1. 从变量名开始
2. 往右读直到读到右括号或到底
3. 往左，忽略读过的
4. 上面的括号均不包括成对的括号
5. `*` 读作“指针，指向”
6. `[x]` 读作“长为 x 的数组，元素类型为”
7. `(...)` 读作“函数，返回值为” + 上面提的成对括号  
(参数列表可选)
8. 如果为匿名类型，手动补充变量名 (参数列表里会出现)

```
int func();
```

函数，返回值为 `int`

```
void func(int a, float b);
```

函数，返回值为 `void`，参数列表为 `int a` 和 `float b`

```
int* func();
```

函数，返回值为 `int*`

```
int (*func)();
```

函数指针，指向返回值为 `int` 的函数

# 解码复杂表达式

decode complex expression

## Quiz

```
int *(*p[2])[3];
```

你可以在 [cdecl.org](http://cdecl.org) 验证，或者尝试其他表达式。

让我们逐步解码：

1. `p` : 变量名
2. `p[2]` : 一个长度为 2 的数组，元素类型为...
3. `*p[2]` : 一个指针，指向...
4. `(*p[2])[3]` : 一个长度为 3 的数组，元素类型  
为...
5. `*(*p[2])[3]` : 一个指针，指向...
6. `int *(*p[2])[3]` : `int` 类型

合并：声明 `p` 为一个包含 2 个元素的 **数组**，每个元素是 **指向一个包含 3 个元素的数组的指针**，**这些数组的元素是指向 int 类型的指针**。

# 解码复杂表达式

- 指针运算动图：阅读方法：先找变量名，一层一层括号往外读

# 数组名和指针

array name and pointer

数组名在大多数情况下的行为类似于指针。

相同

- 数组名在表达式中会被隐式转换为指向数组第一个元素的指针
- 可以对数组名和指针进行类似的指针算术操作

```
int value = *(arr + 2); // 等价于 arr[2]
```

相异

- 数组名指向的是编译时分配的一块连续内存，而指针可以指向动态分配的内存或其他变量。

```
int arr[5]; // 静态分配的数组
int *p = malloc(5 * sizeof(int)); // 动态分配的内存
```

- 数组名是一个常量指针，不能被修改；而指针是一个变量，可以被修改。

```
int arr[5];
int *p = arr;
p = NULL; // 合法, p 可以重新赋值
arr = NULL; // 非法, arr 不能重新赋值
```

# 数组名和指针

array name and pointer

数组名在大多数情况下的行为类似于指针。

相异

- 数组在声明时必须指定大小或初始化，而指针在声明时可以不初始化。

```
int arr[5] = {1, 2, 3, 4, 5}; // 数组声明并初始化
int *p; // 指针声明，无需初始化
p = arr; // 之后初始化
```

- 对数组名使用 `sizeof` 操作符时，返回的是整个数组的大小，而对指针使用 `sizeof` 操作符时，返回的是指针本身的大小。

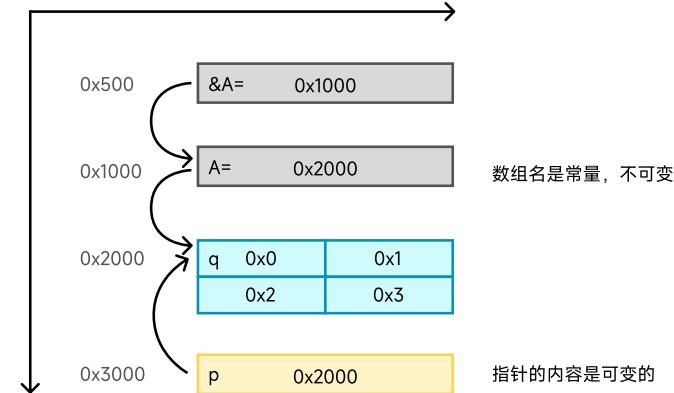
```
int arr[5];
int *p = arr;
// %zu 代表 size_t 类型，通常用于表示 sizeof 操作符的结果
printf("%zu\n", sizeof(arr)); // 输出数组的总大小, 20
printf("%zu\n", sizeof(p)); // 输出指针的大小, 8
printf("%zu\n", sizeof(*p)); // 输出指针所指向的类型的大小, 4
```

# 数组名和指针

array name and pointer

```
sizeof(q) = 4;
sizeof(A) = 16;
int* p = A; // 只是完成了赋值（数据一样了）,
              // 但是没有让他们附带的信息（指向的内容大小）一样
sizeof(p) = 8; // 从而 sizeof 有不同的结果
```

理解为两张照片，像素一模一样，但是元数据（在哪里拍的 / 怎么修的图 → 指向空间大小）不一样



# sizeof 与数组名

sizeof 和 array name

注意，当 A 是数组名，调用 sizeof(A) 时，返回的是整个数组的大小，如下所示：

```
int main() {
    int A[5][3];
    cout << sizeof(&A) << endl; // 8, 因为 A 是数组名，也就是指针，其内容是一串 8 字节地址常量
    // ↑ 所以 sizeof(&A) 是指针类型的大小，即 8 字节
    cout << sizeof(A) << endl; // 60, A 是指针，指向一块 5 * 3 * sizeof(int) 大小的空间，即 int[5][3]
    cout << sizeof(*A) << endl; // 12, *A 是指针，指向一块 3 * sizeof(int) 大小的空间，即 int[3]
    cout << sizeof(A[0]) << endl; // 12, A[0] 等价于 *A，即 int[3]
    cout << sizeof(**A) << endl; // 4, **A 是指针，指向一块 sizeof(int) 大小的空间，即 int
    cout << sizeof(A[0][0]) << endl; // 4, A[0][0] 等价于 **A
    cout << &A << " " << (&A + 1) << endl; // 0x8c 0xc8, 可以看到差值为 0x3c，即 60
}
```

\*此页内容可能存在不严谨之处，能理解、会算就行，考试真的会考

# 指针的大小与类型

pointer size and type

对于一个 `int* p`, `sizeof(p)` 等于?

答: 因为 `p` 本身是一个变量名, 它对应一个 `int*` 类型的变量, 所以 `sizeof(p)` 返回的是指针的大小 (8), 而不是 `int` 的大小 (4)。但是, `sizeof(*p)` 返回的是 `int` 的大小 (4)。

辨析: `int q`

此时, `q` 也是一个变量名, 但它对应一个 `int` 类型的变量, 所以 `sizeof(q)` 返回的是其指向的内容, 也即一个 `int` 的大小 (4)。

- 变量名是内存空间的名字 (好比人的名字), 调用 `sizeof(p)` 时, 返回的是其对应的内容的大小
- 地址是指内存空间的编号 (好比人的身份证号码), 是一个值、一段数据, 调用 `sizeof(p)` 时, 返回的是其指向的内容的大小
- 通过变量名或者地址都能获取这块内存空间的内容 (就好比通过名字或者身份证都能找到这个人)。

# 定长数组

fixed-length array

在处理定长数组时，编译器通过优化，可以尽可能避免开销较大的乘法运算。

原始的 C 代码

```
int fix_prod_ele(fix_matrix A, fix_matrix B, long i, long j;
    int result = 0;
    for (j = 0; j < N; j++) {
        result += A[i][j] * B[j][k];
    }
    return result;
}
```



优化的 C 代码

```
int fix_prod_ele_opt(fix_matrix A, fix_matrix B, long i,
    int *Aptr = &A[i][0];
    int *Bptr = &B[0][k];
    int *Bend = &B[N][k];
    int result = 0;

    do {
        result += *Aptr * *Bptr;
        Aptr++;
        Bptr += N;
    } while (Bptr != Bend);

    return result;
}
```



- 利用指针加速元素访问和乘法操作

- 这是常规的固定矩阵乘法实现
- 迭代访问元素并计算乘积
- 因为使用了 `A[i][j]` 这种形式，所以每次访问一个矩阵元素时，都需要进行一次乘法运算。

# 变长数组

variable-length array

变长数组为灵活的数据存储解决方案。由于数组长度的不确定性，使用单个索引时容易导致性能问题。

初始 C 代码

```
/* 计算变量矩阵乘积的函数 */
int var_prod_ele(long n, int A[n][n], int B[n][n], long
    long j;
    int result = 0;
    for (j = 0; j < n; j++) {
        result += A[i][j] * B[j][k];
    }
    return result;
}
```



- 由于使用了 `A[n][n]` 这种形式，而 `n` 是不能在编译时确定的变量，所以每次访问一个矩阵元素时，都需要进行一次乘法运算。
- 所以在访问变长数组元素时，被迫使用 `imulq`，这可能会导致性能下降。

优化后的 C 代码

```
/* 优化后的变量矩阵乘积计算函数 */
int var_prod_ele_opt(long n, int A[n][n], int B[n][n], l
    int *Arow = A[i];
    int *Bptr = &B[0][k];
    int result = 0;
    long j;

    for (j = 0; j < n; j++) {
        result += Arow[j] * *Bptr;
        Bptr += n; // 向后移动指针，以减少访问时间
    }
    return result;
}
```



- 规律性的访问仍能被优化：通过定位数组指针，避免重复计算索引。

# 数据结构

data structure

struct

所有组分存放在内存中一段连续的区域内。

```
struct S3 {
    char c;
    int i[2];
    double v;
};
```

union

用不同的字段引用相同的内存块。

```
union U3 {
    char c;
    int i[2];
    double v;
};
```

类型	c	i	v	大小
S3	0	4	16	24
U3	0	0	0	8

可以看到，对于 Union，所有字段的偏移都是 0，因为它们共享同一块内存。

# 对齐

alignment

任何  $K$  字节的基本对象的地址必须是  $K$  的倍数。

$K$	类型
-----	----

1	char
---	------

2	short
---	-------

4	int float
---	-----------

8	long double char*
---	-------------------

# 结构体对齐示例

structure alignment example

下面的例子展示了如何对结构体进行对齐：

```
struct S1 {  
    int i;  
    char c;  
    int j;  
};
```

虽然结构体的三个元素总共只占 9 字节，但为了满足变量 `j` 的对齐，内存布局要求填充一个 3 字节的间隙，这样 `j` 的偏移量将是 8，也就导致了整个结构体的大小达到 12 字节。

# 结构体对齐示例

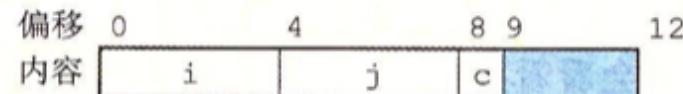
structure alignment example

继续考虑另一个结构体的定义：

```
struct S2 {
    int i;
    int j;
    char c;
};
```

此时，正常排确实可以只需要 9 字节，且同时满足了 `i` 和 `j` 的对齐要求。

但是，因为要考虑可能会有 `S2[]` 这种数组声明，且数组又要求各元素在内存中连续，所以编译器实际上会为结构体分配 12 字节，最后 3 个字节是浪费的空间。



`.align 8` 命令可以确保数据的开始地址满足 8 的倍数。

# 结构体对齐示例

structure alignment example

在 x86-64 平台下，Linux 操作系统中，定义如下的 C 结构体：

## 定义

```
struct A {  
    char CC1[6];  
    int II1;  
    long LL1;  
    char CC2[10];  
    long LL2;  
    int II2;  
};
```

回答以下问题：

1. `sizeof(A)` 为？  $6(2) + 4(4) + 8 + 10(6) + 8 + 4(4) = 56$
2. 若将结构体重排，尽量减少结构体的大小，得到的新结构体大小？

$$6 + 10 + 4 + 4 + 8 + 8 = 40$$

## 技巧：

- 尽量减少结构体的大小：依据数据类型大小排序，从小到大 / 从大到小都可
- 结构体的对齐以其中最大的数据类型为准，对于嵌套的 `union` `struct` 以其内部最大的为准

# 结构体对齐示例

structure alignment example

## 定义

```
typedef union {
    char c[7];
    short h;
} union_e;

typedef struct {
    char d[3]; // 4
    union_e u; // 8
    int i; // 4
} struct_e;

struct_e s;
```

回答以下问题：

1. `s.u.c` 的首地址相对于 `s` 的首地址的偏移量是? 4
2. `sizeof(union_e)` 为? 8
3. `s.i` 的首地址相对于 `s` 的首地址的偏移量是? 12
4. `sizeof(struct_e)` 为? 16
5. 若只将 `i` 的类型改成 `short`，那么 `sizeof(struct_e)` 为? 14
6. 若只将 `h` 的类型改成 `int`，那么 `sizeof(union_e)` 为? 8
7. 若将 `i` 的类型改成 `short`，将 `h` 的类型改成 `int`，那么 `sizeof(union_e)` 为? `sizeof(struct_e)` 为? 8 16
8. 若将 `short h` 的定义删除，那么 (1)~(4) 间的答案分别是? 3 7 12 16

# 强制对齐

force alignment

- 任何内存分配函数（`alloca` `malloc` `calloc` `realloc`）生成的块的起始地址都必须是 16 的倍数
- 大多数函数的栈帧的边界都必须是 16 字节的倍数（这个要求有一些例外）
- 参数构造区向 8 对齐

# 浮点型

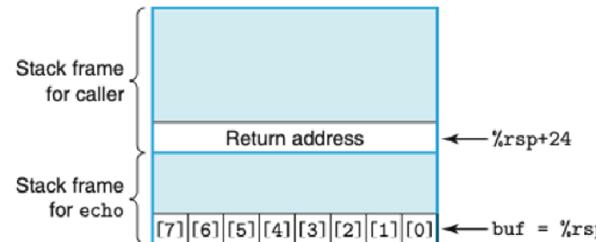
- 浮点数使用的寄存器叫做向量寄存器，共有 16 个，每个有 256 位，最多能保存 4 个 double 类型的数据
- 但在进行标量操作时，引用的寄存器是 `xmm` 而不是 `ymm`，且只会使用寄存器的最低 32 位或 64 位（寄存器的其他部分用于高级的优化技术）

# Machine Advanced

# 内存越界引用与缓冲区溢出

## 攻击手段

- 直接攻击：**让 `gets` 读入的字符串长度超过栈上分配空间的大小，覆盖返回地址，使返回地址被修改为攻击者希望的位置，这个位置可以有任何一个函数，比如可以是执行关机程序的系统调用



Characters typed	Additional corrupted state
0–7	None
9–23	Unused stack space
24–31	Return address
32+	Saved state in caller

# 内存越界引用与缓冲区溢出

## 攻击手段

- 对抗 ASLR: nop sled
  - 在跳转到的函数前放置大段 nop, 只要跳转到 nop 所在区域就能攻击成功
- 对抗 canary
  - 在程序运行过程中直接输出, 暴力获取 canary 值
  - 崩溃后自动重启程序 ASLR 偏移和 canary 值都不变, 暴力破解所有 canary 位
- 对抗 NX: ROP 攻击
  - 利用各类已经存在的函数的 ret 语句 (c3) 前的最后一段机器码 (可能是某个指令的一部分) 帮助实现特定的操作, 从而实现攻击

# 内存越界引用与缓冲区溢出

举个例子：

rtarget 有这样一个函数：

```
void setval_210(unsigned *p)
{
    *p = 3347663060U;
}
```

它的汇编代码字节级表示为：

```
0000000000400f15 <setval_210>:
400f15: c7 07 d4 48 89 c7    movl $0xc78948d4,(%rdi)
400f1b: c3                      retq
```

查表可知，取其中一部分字节序列 48 89 c7 就表示指令 `movq %rax, %rdi`，这整句指令的地址为 `0x400f15`，于是从 `0x400f18` 开始的代码就可以变成下面这样：

```
movq %rax, %rdi
ret
```

# 防御手段

- **栈随机化**: 地址空间布局随机化 (ASLR) 的一部分，每次启动栈的空间都被随机分配，从而攻击者难以指定返回地址的具体位置
- **栈破坏检测**: 金丝雀 **canary**，在返回地址和栈的其余部分之间存储一个金丝雀值，如果函数即将返回时金丝雀值被改动，说明受到了攻击
- **限制可执行代码区域**: NX (No Execute) 位，保证栈上大部分内容不可执行
- 以上所有方法均无法彻底防范栈溢出攻击
- 如果你现在还没明白，没关系，**attacklab** 会让你明白

# Homework Review

# Q1

- P217 3.60 考慮下面的汇编代码.....

## Ans

```
long loop(long x, long n) {
    long result = 0;      // Initial value of result
    long mask;           // Initial value of mask

    // Loop as long as mask is non-zero
    for (mask = 1; mask != 0; mask = mask << n) {
        result |= (x & mask); // Update result with x & mask
    }

    return result; // Return result
}
```

# Q1

## Sol

- 有同学注意到了n的溢出问题，但是到底该如何处理？

- `mask = mask << n`
- `mask = mask << (n & 0xFF)`
- `mask = mask << (n & 0x3F)`

```
long loop(long x, long n) {
    long result = 0;      // Initial value of result
    long mask;           // Initial value of mask

    for (mask=1; mask!=0; mask = mask << (n & 0x3F))
    {
        result |= (x & mask);
    }

    return result; // Return result
}
```

```
C hw.c
class-in > 03-prog > C hw.c > ...
41 }
42 |
43 void main()
44 {
45     long x = 0x1234567812345678;
46     // 打印sizeof(Long)的值
47     printf("Size of long: %ld\n", sizeof(long));
48     // 打印x的值 16进制
49     printf("Value of x: %lx\n", x);
50
51     long result_64 = x << 64;
52     long result_63 = x << 63;
53     printf("Result 64: %lx\n", result_64);
54     printf("Result 63: %lx\n", result_63);
55 }

[Done] exited with code=13 in 0.046 seconds

[Running] cd "/home/ubuntu/ICS/class-in/03-prog/" && gcc hw.c -o hw && "/home/ubuntu/ICS/class-in/03-prog/"/hw
hw.c: In function ‘main’:
hw.c:51:24: warning: left shift count >= width of type [-Wshift-count-overflow]
| 51 |     long result_64 = x << 64;
|     |                                     ^
Size of long: 8
Value of x: 1234567812345678
Result 64: 0
Result 63: 0

[Done] exited with code=13 in 0.05 seconds

[Running] cd "/home/ubuntu/ICS/class-in/03-prog/" && gcc hw.c -o hw && "/home/ubuntu/ICS/class-in/03-prog/"/hw
SSH: 10.129.81.70
```

The screenshot shows a terminal window running on an SSH connection to IP 10.129.81.70. It displays the output of the hw.c program. The program prints the size of a long variable (8 bytes), its value (1234567812345678), and then performs a left shift operation on the value by 64 bits. Due to the nature of the left shift operation, the result is 0 because the value is zeroed out during the shift. The terminal also shows the command used to run the program and the current working directory.

## Q2

- P219 3.63 这个程序给你.....

```
long switch_prob(long x, long n){  
    long result = x;  
    switch (n){  
        case 60:  
        case 62:  
            result *= 8;  
            break;  
        case 61:  
            result += 75; // 0x4b = 75  
            break;  
        case 63:  
            result >>= 3;  
            break;  
        case 64:  
            result = ((result << 4) - x); // (x << 4) - x  
        case 65:  
            result *= result;  
        default:  
            result = result + 75; // 0x4b = 75  
    }  
    return result;  
}
```

## Q3

- P221 3.66 考慮下面的源代码.....

## Ans

```
#define NR(n) (3*(n))
#define NC(n) (4*(n) + 1)
```

- 基于 `leaq 1(%rdi, 4), %r8` 和 `addq %r8, %rcx` 确定 NC
  - `salq $3, %r8`, 注意到了很好, 但注意到了一定更要想清楚
- 基于 `leaq (%rdi,%rdi,2), %rax` 和 `testq %rax, %rax` 确定 NR

# Sol

```
long sum_col(long n, long A[NR(n)][NC(n)], long j)
n in %rdi, A in %rsi, j in %rdx
1 sum_col:
2  leaq 1(%rdi,4), %r8          # r8 = 4*n + 1
3  leaq (%rdi,%rdi,2), %rax    # rax = 3*n
4  movq %rax, %rdi              # rdi = 3*n
5  testq %rax, %rax            # 测试 rax 是否为零
6  jle .L4                      # 如果 3*n <= 0, 跳转到 .L4 (返回 0)
7  salq $3, %r8                # r8 = (4*n + 1) << 3 = (4*n + 1) * 8 = 32*n + 8
8  leaq (%rsi,%rdx,8), %rcx    # rcx = A + j*8
9  movl $0, %eax                # result = 0
10 movl $0, %edx                # i = 0
11 .L3:
12  addq (%rcx), %rax          # result += A[i][j]
13  addq $1, %rdx                # i++
14  addq %r8, %rcx              # rcx += 32*n + 8
15  cmpq %rdi, %rdx              # 比较 i 和 3*n
16  jne .L3                      # 如果 i < 3*n, 继续循环
17  rep; ret
18 .L4:
19  movl $0, %eax                # 返回 0
20  ret
```

# Q4

- P221 3.67 这个作业要查看GCC.....

A. 图示 eval 函数的栈帧，显示在调用 process 之前存储在栈上的值

**答案：**

在调用 process 函数之前，eval 函数的栈帧如下所示（从低地址到高地址）：

```
+-----+ <-- %rsp + 104
|       ...
+-----+ <-- %rsp + 88
|     strB r (64-80(%rsp)) | // 返回值结构体 r 的存储空间
+-----+ <-- %rsp + 32
|       ...
+-----+ <-- %rsp + 24
|     z (24(%rsp))      | // z 的值
+-----+ <-- %rsp + 16
|     &z (16(%rsp))    | // s.p = &z
+-----+ <-- %rsp + 8
|     y (8(%rsp))      | // s.a[1] = y
+-----+ <-- %rsp + 0
|     x (0(%rsp))      | // s.a[0] = x
+-----+
```

## Q4

B. `eval` 在调用 `process` 时传递了什么值？

**答案：**

`eval` 函数在调用 `process` 时，传递了一个指向返回值结构体 `strB r` 的存储地址，即 `64(%rsp)`，通过寄存器 `%rdi` 传递给 `process` 函数。

## Q4

C. `process` 函数如何访问结构体参数 `s` 的各个元素？

**答案：**

`process` 函数通过固定的栈偏移量访问结构体参数 `s` 的各个成员：

- `s.a[0]` 存储在栈偏移 `16(%rsp)`。
- `s.a[1]` 存储在栈偏移 `8(%rsp)`。
- `s.p` 存储在栈偏移 `24(%rsp)`。

## Q4

C. `process` 函数如何访问结构体参数 `s` 的各个元素？

具体对应关系：

- `s.p` (**指向 `z` 的指针**)：
  - 指令 `movq 24(%rsp), %rdx` 从栈偏移 `24(%rsp)` 加载 `s.p` 的值 (即 `&z`) 到寄存器 `%rdx`。
  - 指令 `movq (%rdx), %rdx` 解引用指针，加载 `*s.p` (即 `z` 的值) 到 `%rdx`。
- `s.a[0]` (**即 `x`**)：
  - 指令 `movq 16(%rsp), %rcx` 从栈偏移 `16(%rsp)` 加载 `s.a[0]` 的值到 `%rcx`。
- `s.a[1]` (**即 `y`**)：
  - 指令 `movq 8(%rsp), %rcx` 从栈偏移 `8(%rsp)` 加载 `s.a[1]` 的值到 `%rcx`。

## Q4

D. `process` 函数如何设置结果结构体 `r` 的字段?

**答案:**

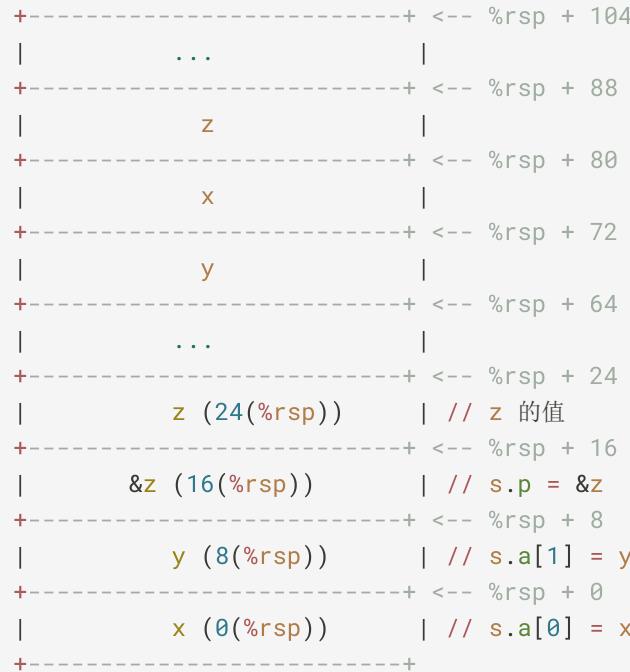
`process` 函数通过返回值地址 `%rdi` 指向的内存位置，依次将 `s.a[0]`，`s.a[1]` 和 `*s.p` 的值赋给 `r.u[0]`，`r.u[1]` 和 `r.q`，具体操作如下：

- `r.u[0] = s.a[0]` : `movq %rcx, (%rdi)`
- `r.u[1] = s.a[1]` : `movq %rcx, 8(%rdi)`
- `r.q = *s.p` : `movq %rdx, 16(%rdi)`

# Q4

E. 完成 eval 函数的栈帧图示，显示 eval 在 process 返回后如何访问结构体 r 的元素

答案：



## Q4

F. 关于结构体作为函数参数传递和作为函数返回值的一般原则有哪些？

**答案：**

**结构体参数传递：**

- **小结构体：** 可能通过寄存器传递。
- **大结构体：** 通过栈传递，成员按顺序存储，使用固定栈偏移量访问。

**结构体返回值：**

- 通常通过隐藏的返回值指针传递，调用者预留内存空间并传递其地址给被调用函数。
- **小结构体：** 在某些情况下，可能通过寄存器返回。

**调用约定：**

- 遵循 x86-64 调用约定，使用寄存器和栈进行参数传递和返回值管理。
- 隐藏参数用于传递返回值地址，优化函数调用效率。

## Q5

- P223 3.68 在下面的代码中.....

## Ans

```
#define A 9  
#define B 5
```

# Q5

## Sol

```
typedef struct {
    int x[A][B]; // 占用 A * B * 4 字节
    long y;      // 占用 8 字节
} str1;
```

```
typedef struct {
    char array[B]; // 占用 B 字节
    int t;         // 占用 4 字节
    short s[A];   // 占用 2A 字节
    long u;        // 占用 8 字节
} str2;
```

1. movslq 8(%rsi), %rax →  $B \in (4, 8]$ , 由于对齐原则无法确定
2. addq 32(%rsi), %rax →  $A \in (6, 10]$ , 由于对齐原则无法确定
3. movq %rax, 184(%rdi) →  $A \times B \in (44, 46]$ , 由于对齐原则无法确定
4. 综上, 有唯一整数解  $A=9$  ,  $B=5$

# Exercises

## E1

7. 在作业题中，我们考察过 GCC 为一个参数和返回值都是结构体的函数产生的汇编代码，此时参数和返回值都是通过栈传递的，以下有关说法错误的是：

- A. 返回的结构体的地址被存放在%rax 寄存器中
- B. 虽然参数是通过栈传递的，但这里依旧使用到了%rdi
- C. 返回的结构体实际上は存放在被调用者的栈帧当中
- D. 如果结构体很小的话，GCC 可能会直接用寄存器来传输参数和返回值

# E1

7. 在作业题中，我们考察过 GCC 为一个参数和返回值都是结构体的函数产生的汇编代码，此时参数和返回值都是通过栈传递的，以下有关说法错误的是：

- A. 返回的结构体的地址被存放在%rax寄存器中
- B. 虽然参数是通过栈传递的，但这里依旧使用到了%rdi
- C. 返回的结构体实际上是存放在被调用者的栈帧当中
- D. 如果结构体很小的话，GCC 可能会直接用寄存器来传输参数和返回值

答案：C。

c. 返回的结构体实际上存放在调用者的栈帧当中，而非被调用者

## E2

7. 考虑以下 C 语言变量声明：

```
int * (*f[3])();
```

那么在一台 x86-64 机器上，`sizeof(f)` 和 `sizeof(*f)` 的值是多少？

- A. 8 24
- B. 24 8
- C. 8 8
- D. 8 不确定

## E2

7. 考虑以下 C 语言变量声明：

```
int * (*f[3])();
```

那么在一台 x86-64 机器上，`sizeof(f)` 和 `sizeof(*f)` 的值是多少？

A. 8 24

B. 24 8

C. 8 8

D. 8 不确定

答案：B 本题考察指针和数组的存储方式，答案为 B。`f` 是一个数组，每个元素都是一个函数指针，指向返回值为 `int *` 的函数。对于 `f` 而言，`sizeof(f)` 返回整个指针数组的大小，为  $3 * 8 = 24$ ；对于 `*f` 而言，它是 `f[0]` 元素，是一个指针变量，大小为 8。

## E3

7. 有 A 的定义: int A[3][2] = {{1,2}, {3,3}, {2,1}};  
那么 A[2] 的值为:
- A. &A+16                  B. A+16                  C. \*A+4                  D. \*A+2

## E3

答案：C

解析：参见书 P177 页表格，机器在计算指针与常数的运算时，会将常数乘以指针指向的元素大小。&A 常数扩大的倍数为 `sizeof(A[3][2]) = 3*2*4`；A 常数扩大的倍数为 `sizeof(A[0])=2*4`；\*A 常数扩大的倍数为 `sizeof(int) = 4`。  
正确的答案应为 A+2 或 \*A+4，故应选择 C。

程序验证如下：

```
int main(){
    int A[3][2] = {{1,2},{3,3},{2,1}};
    cout << "&A+16: " << &A+16 << endl;
    cout << "A+16: " << A+16 << endl;
    cout << "*A+4: " << *A+4 << endl;
    cout << "*A+2: " << *A+2 << endl;
    cout << A[2] << endl;
}
```

```
&A+16: 0xb5fee0
A+16: 0xb5fdde0
*A+4: 0xb5fd70
*A+2: 0xb5fd68
0xb5fd70
```

## E4

6. 下列关于 C 语言中的结构体(struct)以及联合(union)的说法中，正确的是：
- A) 对于任意 struct，将其成员按照其实际占用内存大小从小到大的顺序进行排列  
不一定会使之内存占用最小
  - B) 对于任意 struct，将其成员按照其实际占用内存大小从小到大的顺序进行排列  
一定不会使之内存占用最大
  - C) 对于任意 union，将其成员按照其实际占用内存大小从小到大的顺序进行排列  
不一定会使之内存占用最小
  - D) 对于任意 union，将其成员按照其实际占用内存大小从小到大的顺序进行排列  
一定不会使之内存占用最大

## E4

6. 下列关于 C 语言中的结构体(struct)以及联合(union)的说法中，正确的是：
- A) 对于任意 struct，将其成员按照其实际占用内存大小从小到大的顺序进行排列  
不一定会使之内存占用最小
  - B) 对于任意 struct，将其成员按照其实际占用内存大小从小到大的顺序进行排列  
一定不会使之内存占用最大
  - C) 对于任意 union，将其成员按照其实际占用内存大小从小到大的顺序进行排列  
不一定会使之内存占用最小
  - D) 对于任意 union，将其成员按照其实际占用内存大小从小到大的顺序进行排列  
一定不会使之内存占用最大

答案：A。联合以及只含一个基本数据类型成员的结构体的内存占用与其成员排列方式无关，即任意排列方式都可使得内存占用最小（最大），故 b、c、d 错误。对于 struct，应当将成员按照其存储单元所占内存大小从小到大（或从大到小）的顺序进行排列才能使内存占用最小，故 a 正确。

## E5

8. 在 x86-64 架构下，有如下变量：

```
union {char c[8], int i;} x;
```

在  $x.i=0x41424344$  时， $x.c[2]$  的值为多少（提示： $'A'=0x41$ ）：

- A. 'A'
- B. 'B'
- C. 'C'
- D. 'D'

## E5

8. 在 x86-64 架构下，有如下变量：

```
union {char c[8], int i;} x;
```

在  $x.i=0x41424344$  时， $x.c[2]$  的值为多少（提示：'A'=0x41）：

- A. 'A'
- B. 'B'
- C. 'C'
- D. 'D'

**答案： B**

# E6

5. 以下代码的输出结果是

```
union {
    double d;
    struct {
        int i;
        char c[4];
    } s;
} u;
u.d = 1;
printf("%d\n", u.s.c[2]);
```

- A) 0
- B) -16
- C) 240
- D) 191

## E6

答案：B

c[2]保存的数据是 $(11110000)_2$ , char 保存的数据是有符号数, 所以应该是-16

答案修订：ABC 均正确。

题目未说明大小端, 若为大端法, 选择 A。若为小端法, u. s. c[2]为 $(11110000)_2$ , 由于 char 在不同的编译器上会被实现成 signed char 或 unsigned char, 因此扩展为 int 时-16 和 240 均有可能, B, C 均正确。

## E7

5. 已知下面的数据结构，假设在 Linux/IA32 下要求对齐，这个结构的总的大  
小是多少个字节？如果重新排列其中的字段，最少可以达到多少个字节？

```
struct {  
    char a;  
    double *b;  
    double c;  
    short d;  
    long long e;  
    short f;  
};
```

- A. 32, 28
- B. 36, 32
- C. 28, 26
- D. 26, 26

## E7

5. 已知下面的数据结构，假设在 Linux/IA32 下要求对齐，这个结构的总的大  
小是多少个字节？如果重新排列其中的字段，最少可以达到多少个字节？

```
struct {  
    char a;  
    double *b;  
    double c;  
    short d;  
    long long e;  
    short f;  
};
```

- A. 32, 28
- B. 36, 32
- C. 28, 26
- D. 26, 26

选择 A, Linux/IA32 要求 double 和 long long 在 4 字节边界上对齐，  
其中 a 占 1 字节，后面填充 1 字节，b 占 4 字节，填充 2 个字节，c 占 8 字节，  
d 占 2 字节，填充 2 字节，e 占 8 字节，f 占 2 字节，再填充 2 字节，共 32 字节；  
重新排列时，可以按字节数从大到小排列，最后，最少需要  $8+8+4+2+2+1$   
再加 3 字节满足 4 字节对齐要求，共 28 个字节。

## E8

6、有如下定义的结构，在 x86-64 下，下述结论中错误的是？

```
struct {
    char c;
    union {
        char vc;
        double value;
        int vi;
    } u;
    int i;
} sa;
```

- A.  $\text{sizeof}(sa) == 24$
- B.  $(\&sa.i - \&sa.u.vi) == 8$
- C.  $(\&sa.u.vc - \&sa.c) == 8$
- D. 优化成员变量的顺序，可以做到“ $\text{sizeof}(sa) == 16$ ”

答：( )

## E8

答案：B

由于对齐的需求，在 `x86-64` 下，要保证 `sa.u.value` (`double` 类型变量) 的地址必须 8 字节对齐，而在 `ia32` 的 `Linux` 系统中，可 4 字节对齐。故 `sizeof(sa)` 在 `x86-64` 下要占用三个 8 字节的空间共 24 字节。`sa.u.vc` 与 `sa.c` 之间和 `sa.i` 与 `sa.u.vi` 之间的地址值之差均为 8，但即使不知道 `(&sa.i - &sa.u.vi)` 表示之间可以放置多少个整数（值为 2），也可通过优化成员变量顺序后有 `sizeof(sa) == 16`，用排除法得出正确答案。

## E9

7、关于如何避免缓冲区溢出带来的程序风险，下述错误的做法为？

- A. 编程时定义大的缓冲区数组
- B. 编程时避免使用 gets，而采用 fgets
- C. 程序运行时随机化栈的偏移地址
- D. 在硬件级别引入不可执行代码段的机制

答：( )

## E9

7、关于如何避免缓冲区溢出带来的程序风险，下述错误的做法为？

- A. 编程时定义大的缓冲区数组
- B. 编程时避免使用 gets，而采用 fgets
- C. 程序运行时随机化栈的偏移地址
- D. 在硬件级别引入不可执行代码段的机制

答：( )

答案：**A**

**B、C、D** 均为讲义中所提及的解决缓冲区溢出风险的方法。**A** 策略则无法从根本上解决缓冲区溢出问题，只要输入足够长数据就仍然可以实现缓冲区溢出攻击。

## E10

8. 大多数过程的栈帧是\_\_\_\_\_的，其长度在\_\_\_\_\_时确定。（注：此处的编译指从高级语言转化为汇编语言的过程）
- A. 定长，编译
  - B. 定长，汇编
  - C. 可变长，汇编
  - D. 可变长，运行

## E10

8. 大多数过程的栈帧是\_\_\_\_\_的，其长度在\_\_\_\_\_时确定。（注：此处的编译指从高级语言转化为汇编语言的过程）
- A. 定长，编译
  - B. 定长，汇编
  - C. 可变长，汇编
  - D. 可变长，运行

**答案：A。**

**说明：**大多数过程的栈帧是定长的，在过程开始时通过减小栈指针的方式分配，减小的大小由编译器在编译时计算。大家常常在汇编代码中过程的开头看到“`subq $24, %rsp`”，就是编译器计算出了栈帧大小并写在了汇编代码里。（书 P165）

# E11

9. pushq %rbp 的行为等价于以下()中的两条指令。

- A. subq \$8, %rsp      movq %rbp, (%rdx)
- B. subq \$8, %rsp      movq %rbp, (%rsp)
- C. subq \$8, %rsp      movq %rax, (%rsp)
- D. subq \$8, %rax      movq %rbp, (%rdx)

## E11

9. pushq %rbp 的行为等价于以下()中的两条指令。

- A. subq \$8, %rsp      movq %rbp, (%rdx)
- B. subq \$8, %rsp      movq %rbp, (%rsp)
- C. subq \$8, %rsp      movq %rax, (%rsp)
- D. subq \$8, %rax      movq %rbp, (%rdx)

答案: B。

说明: x86-64 系统中, pushq %rbp 指令将栈指针减 8, 并向其中存入%rbp 寄存器的值 (书 P127)

## E12

5. 已知函数 func 的参数超过 6 个。当 x86-64 机器执行完指令 call func 之后，%rsp 的值为 s。那么 func 的第 k( $k > 6$ ) 个参数的存储地址是？
- A.  $s + 8 * (k - 6)$
  - B.  $s + 8 * (k - 7)$
  - C.  $s - 8 * (k - 6)$
  - D.  $s - 8 * (k - 7)$

## E12

5. 已知函数 func 的参数超过 6 个。当 x86-64 机器执行完指令 call func 之后，%rsp 的值为 s。那么 func 的第 k( $k > 6$ ) 个参数的存储地址是？
- A.  $s + 8 * (k - 6)$
  - B.  $s + 8 * (k - 7)$
  - C.  $s - 8 * (k - 6)$
  - D.  $s - 8 * (k - 7)$
- · ·

答案：A

本题考察 x86-64 运行时栈帧结构，答案为 A。当执行完 call 指令后，s 处存储的是函数的返回地址；再往上依次是第 7、第 8 个函数参数...故第 k 个函数参数存储在  $s + 8 * (k - 6)$  的地址处。

## E13

9. x86 体系结构中，下面哪个说法是正确的？

- A. leal 指令只能够用来计算内存地址
- B. x86\_64 机器可以使用栈来给函数传递参数
- C. 在一个函数内，改变任一寄存器的值之前必须先将其原始数据保存在栈内
- D. 判断两个寄存器中值大小关系，只需要 SF 和 ZF 两个条件码

## E13

9. x86 体系结构中，下面哪个说法是正确的？

- A. leal 指令只能够用来计算内存地址
- B. x86\_64 机器可以使用栈来给函数传递参数
- C. 在一个函数内，改变任一寄存器的值之前必须先将其原始数据保存在栈内
- D. 判断两个寄存器中值大小关系，只需要 SF 和 ZF 两个条件码

答案：B

A. leal 指令做普通算术运算；C. caller saved 寄存器才需要；D. 还需要 OF

# E14

## 第三题 机器级编程 (15 分, 每空 1 分)

下面的 C 程序包含 main(), caller(), callee() 三个函数。本题给出了该程序的部分 C 代码和 X86-64 汇编与机器代码。请分析给出的代码，补全空白处的内容，并回答问题。

注：汇编与机器码中的数字用 16 进制数填写

X86-64 汇编与机器代码：

答案填写处：

```
0000000000400fcd <callers>
4006c6: 55          push    %rbp
4006e1: 48 89 e5    mov     %rsp, %rbp
4006d1: 48 83 ec 50 sub    $0x50, %rsp
4006d5: 48 89 7d b8 mov     %rdi, -0x48(%rbp)
4006d9: 64 48 8b 04 25 28 00 mov     %fs:0x28, %rax
4006e0: 00 00
4006e2: 48 89 45 f8 mov     %rax, -0x8(%rbp)
4006e6: 31 c0 xor    %eax, %eax
4006e8: c6 45 d0 00 movb   $0x0, -0x30(%rbp)
4006ec: c6 45 e0 00 movb   $0x0, [1] _____
4006f0: 48 8b 45 b8 mov     %rax, [2] _____
4006f4: 48 89 c7 mov     %rax, %rdi
4006f7: callq  400510 <strlen@plt>
4006f8: 89 94 cc mov     [3], -0x34(%rbp) [3] _____
4006f9: 83 7d cc 0e cmpl   $0xe, -0x34(%rbp)
400703: 7f [4] jg    400752 <caller+0x85> [4] _____
400705: 83 7d cc 09 cmpl   $0x9, -0x34(%rbp)
400709: jg    400720 <caller+0x53>
40070b: 48 8b 55 b8 mov     -0x48(%rbp), %rdx
40070f: 48 8d 45 d0 lea    [5], %rax [5] _____
400713: 48 89 d6 mov     %rdx, %rsi
400716: 48 89 c7 mov     %rax, %rdi
400719: callq  400500 <strcpy@plt>
40071e: jmp   40073b <caller+0x6e>
400720: 48 8b 45 b8 mov     -0x48(%rbp), %rax
400724: 48 8d 50 0a lea    $0a(%rax), %rdx
400728: 48 8d 45 d0 lea    -0x30(%rbp), %rax
40072c: 48 83 c0 10 add    [6], %rax [6] _____
```

```
400730: 48 89 d6 mov     %rdx, %rsi
400733: 48 89 c7 mov     %rax, %rdi
400736: callq  400500 <strcpy@plt>
40073b: ff 75 e8 pushq  -0x18(%bp)
40073e: ff 75 e0 pushq  -0x20(%bp)
400741: ff 75 d8 pushq  -0x28(%bp)
400744: ff 75 d0 pushq  -0x30(%bp)
400747: e8 [7] callq  400666 <callee> [7] _____
40074c: 48 83 c4 20 add    $0x20, %rsp
400750: jmp   400753 <caller+0x86>
400752: 90 nop
400753: 48 8b 45 f8 mov     [8], %rax [8] _____
400757: 64 48 33 04 25 28 00 xor    %fs:0x28, %rax
400760: 00 00
400762: je    400767 <caller+0x9a>
400767: c9 leaveq
400768: c3 retq
```

C 代码：

答案填写处：

```
#include <stdio.h>
#include "string.h"
#define N [9] _____
#define M [10] _____
```

```
typedef union {char str_u[N]; long l;} union_e;
typedef struct {char str_s[M]; union_e u; long c;} struct_e;
```

```
void callee(struct_e s){
    char buf[M+N];
    strcpy(buf, s.str_s);
    strcat(buf, s.u.str_u);
    printf("%s\n",buf);
}
```

```
void caller(char *str){
    struct_e s;
    s.str_s[0]=\0';
    s.u.str_u[0]='\0';
    int len = strlen(str);
    if(len>= M+N)
        [11];
    else if(len<N){
        strcpy(s.str_s, [12]);
    }
    else{
        strcpy(s.u.str_u, [13]);
    }
    callee(s);
}

int main(int argc, char *argv[]){
    caller("0123456789abcd");
    return 0;
}
```

caller 函数中，变量 s 所占的内存空间为：(14)\_\_\_\_\_  
 该程序运行后，printf 函数是否有输出？输出结果为：(15)\_\_\_\_\_

# E14

## 答案:

机器级编程（15 分，每空 1 分）

下面的 C 程序包含 main(), caller(), callee() 三个函数。本题给出了该程序的部分 C 代码和 X86-64 汇编与机器代码。请分析给出的代码，补全空白处的内容，并回答问题。

注：汇编与机器码中的数字用 16 进制数填写

X86-64 汇编与机器代码：

00000000004006cd <caller>:

4006cd: 55	push	%rbp
4006ce: 48 89 e5	mov	%rsp, %rbp
4006d1: 48 83 ec 50	sub	\$0x50, %rsp
4006d5: 48 89 7d b8	mov	%rdi, -0x48(%rbp)
4006d9: 64 48 8b 04 25 28 00	mov	%fs:0x28, %rax

答案填写处:

# E14

```

4006e0: 00 00
4006e2: 48 89 45 f8    mov    %rax, -0x8(%rbp)
4006e4: 31 c0            xor    %eax, %eax
4006e8: c6 45 d0 00    movb   $0x0, -0x30(%rbp)
4006ec: c6 45 e0 00    movb   $0x0, [1] -0x28(%rbp)
4006f0: 48 8b 45 b8    mov    [2], %rax      [2] -0x48(%rbp)
4006f4: 48 89 c7            mov    %rax, %rdi
4006f7: callq  400510 <strlen@plt>
4006fc: 89 45 cc    mov    [3], -0x34(%rbp) [3] %eax
4006ff: 83 7d cc 0e    cmpl   $0xe, -0x34(%rbp)
400703: ff [4]           jg    400752 <caller+0x85> [4] 4d
400705: 83 7d cc 09    cmpl   $0x9, -0x34(%rbp)
400709:                400720 <caller+0x53>
40070b: 48 8b 55 b8    mov    -0x48(%rbp), %rdx
40070f: 48 8d 45 d0    lea    [5], %rax      [5] -0x30(%rbp)
400713: 48 89 d6            mov    %rdx, %rsi
400716: 48 89 c7            mov    %rax, %rdi
400719: callq  400500 <strcpy@plt>
40071e: jmp   40073b <caller+0xe>
400720: 48 8b 45 b8    mov    -0x48(%rbp), %rax
400724: 48 8d 50 0a    lea    %rax(%rax), %rdx
400728: 48 8d 45 d0    lea    -0x30(%rbp), %rax
40072c: ff 83 c0 10    add    [6], %rax      [6] 0x8
400730: 48 89 d6            mov    %rdx, %rsi
400733: 48 89 c7            mov    %rax, %rdi
400736: callq  400500 <strcpy@plt>
40073b: ff 75 e8    pushq -0x18(%rbp)
40073e: ff 75 e0    pushq -0x20(%rbp)
400741: ff 75 d8    pushq -0x28(%rbp)
400744: ff 75 d0    pushq -0x30(%rbp)
400747: e8 [7]           callq 400666 <callee> [7] 1a ff ff
40074c: 48 83 c4 20    add    $0x20, %rsp
400750: jmp   400753 <caller+0x86>
400752: 90             nop
400753: 48 8b 45 f8    mov    [8], %rax      [8] -0x8(%rbp)
400757: 64 48 33 04 25 28 00  xor    %fs:0x28, %rax
40075e: 00 00

```

```

400760:                je    400767 <caller+0x9a>
400762: callq  400520 <_stack_chk_fail@plt>
400767: c9            leaveq
400768: c3            retq
}

C 代码: 答案填写处:


```

#include <stdio.h>
#include "string.h"
#define N [9]
#define M [10]
[9] 10
[10] 5

typedef union {char str_u[N]; long l;} union_e;
typedef struct {char str_s[M]; union_e u; long c;} struct_e;

void callee(struct_e s){
    char buf[M+N];
    strcpy(buf, s.str_s);
    strcat(buf, s.u.str_u);
    printf("%s\n",buf);
}

void caller(char *str){
    struct_e s;
    s.str_s[0] = '\0';
    s.u.str_u[0] = '\0';
    int len = strlen(str);
    if(len >= M+N)
        [11];
    else if(len < N){
        strcpy(s.str_s, [12]);
        [12] str
    }
    else{
        strcpy(s.u.str_u, [13]);
        [13] str + M
    }
    callee(s);
}

```


```

```

int main(int argc, char *argv[]){
    caller("0123456789abcd");
    return 0;
}

```

caller 函数中，变量 s 所占的内存空间为：(14) 32 字节  
 该程序运行后，printf 函数是否有输出？输出结果为：(15) abcd

# THANKS

Made by WEB-05

webrun@stu.pku.edu.cn

Reference: [WalkerCH]'s and [Arthals]'s presentations.



扫一扫上面的二维码图案，加我为朋友。