MACHINE PROG:PROCEDURES

程序的机器级表示：过程

- 运行时栈
- 转移控制
- 数据传送
- 栈上的局部存储
- 寄存器中的局部存储
- 递归过程

回顾：

一、C语言中局部变量的生存周期

C中的局部变量只在创建该局部变量的函数内部有效，而在该函数结束（返回）时析构

- 比如for(int i=0;i<n;i++){
  ……
  }其中的i只在循环体内有效

二、C语言的函数调用

参数的传递方式多样：传值（int i）、传指针（int* a）、传引用（int& a）等，这三者中，第一种不会改变源变量的值，而后两者则可以，*a可以改变内存中的值，引用形式的传值可以直接改变源变量。

当调用的函数结束（返回）时，会回到紧跟着 此次调用函数的语句 后面的语句。

C语言中代码段、全局、局部变量在（虚拟）地址中的位置

例如bomblab的phase_1中将一个预先设定好的字符串的地址赋给%rsi

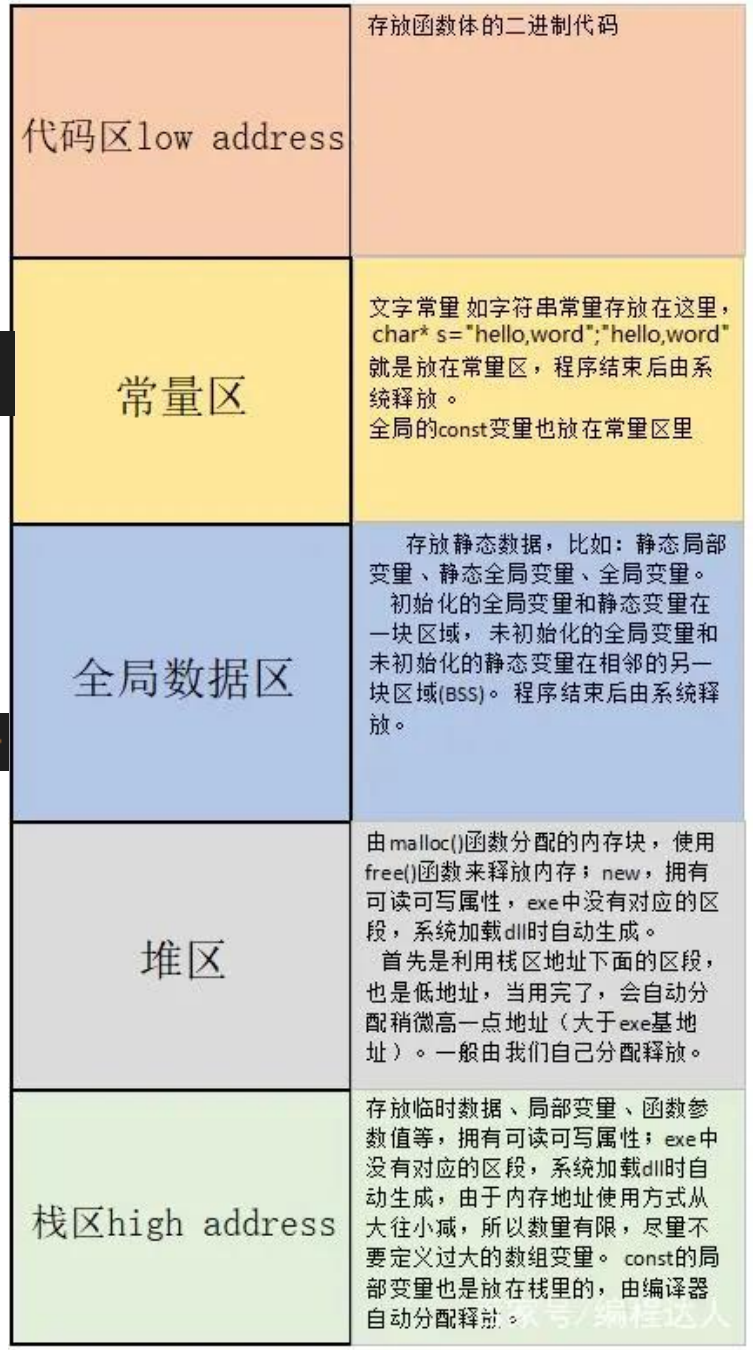
```
17b5:   48 8d 35 54 2a 00 00    lea    0x2a54(%rip),%rsi
```

其中0x2a54(%rip)就是预先设定好的字符串的地址

又例如phase_6中将一个预设好的链表的头节点的地址赋给%rdx

```
1b10:   48 8d 15 39 65 00 00    lea    0x6539(%rip),%rdx        # 8050 <node1>
```

这两个例子中的字符串、链表都是作为全局数据，存在其对应的位置



改变局部变量和全局变量的汇编代码

以bomblab中的一些代码为例

例：改变局部变量

右图中(%rsp,%rcx,4)是栈帧中的局部变量，代码段的功能是把这些值变为7-(%rsp,%rcx,4)

有时是改变寄存器中的局部变量

```
1ad5:   ba 07 00 00 00          mov    $0x7,%edx
1ada:   2b 14 8c                sub    (%rsp,%rcx,4),%edx
1add:   89 14 8c                mov    %edx,(%rsp,%rcx,4)
```

例：改变全局变量

右图中的0x20(%rsp,%rdx,8)、%rcx是一个链表中的节点的地址，代码段可以按一定规则改变链表的连接顺序

```
1b28:   48 63 d0                movslq %eax,%rdx
1b2b:   48 8b 54 d4 20          mov    0x20(%rsp,%rdx,8),%rdx
1b30:   48 89 51 08             mov    %rdx,0x8(%rcx)
1b34:   83 c0 01                add    $0x1,%eax
1b37:   48 89 d1                mov    %rdx,%rcx
1b3a:   83 f8 05                cmp    $0x5,%eax
1b3d:   7e e9                   jle    1b28 <phase_6+0xcb>
```

运行时栈

当x86-64过程需要的存储空间超出寄存器可存放的大小时，会在栈上分配空间，称之为过程的栈帧

注意：栈帧不是必要的，寄存器不够用时，才会在栈上分配空间
回顾：x86中栈的增长实际上是地址的减小，先进入栈的地址大，后进入栈的地址小，%rsp是栈顶指针

栈帧中以返回地址为调用者、被调用者的分界线
注意：返回地址属于调用者的栈帧，其中存放的是此次调用结束后应返回到调用者中的位置的地址

C语言中进出栈的机制与进入、退出函数的机制很相似（进入函数就有新的数据入栈，退出函数时把刚才入栈的数据弹出），函数没有结束等价于对应的栈帧还在栈内（因为被调用的函数可能还会调用别的函数，创建新的栈帧）

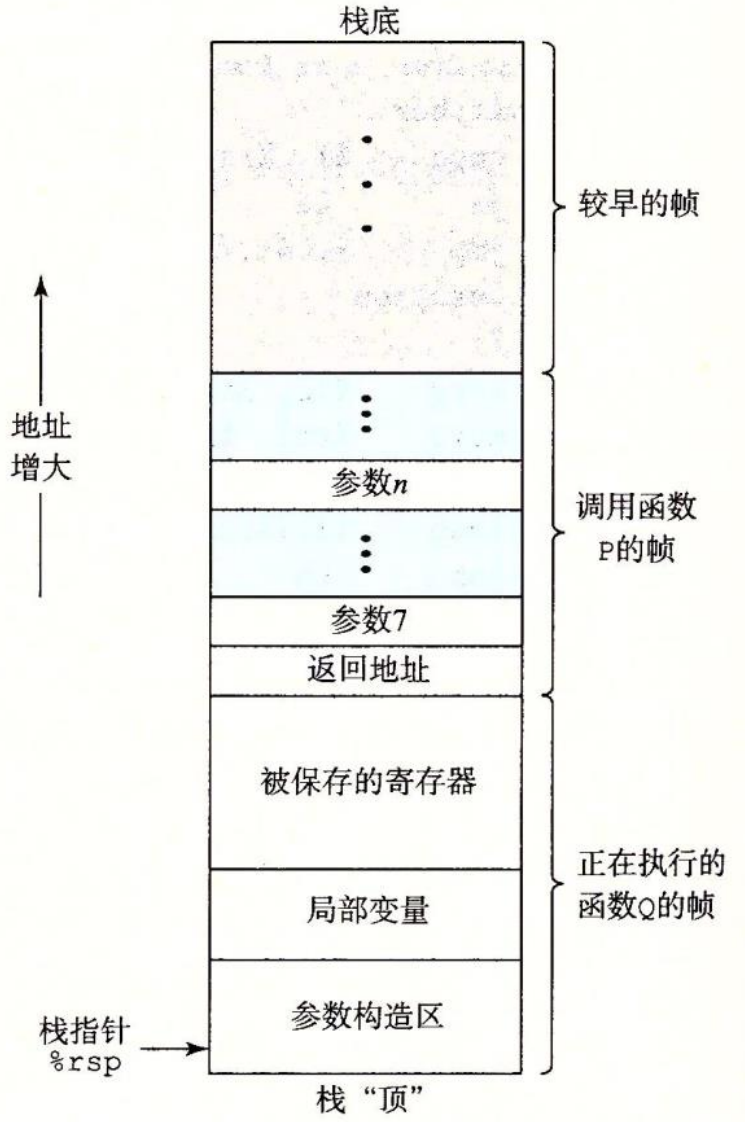


图 3-25 通用的栈帧结构(栈用来传递参数、存储返回信息、保存寄存器，以及局部存储。省略了不必要的部分)

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）

(　　)13.(补充题,不计分 命题人:王星博)下列说法中正确的是____.

A. pushq指令的功能把数据压入到栈上，pushq S 的意义是将寄存器S中的数据保存到寄存器%rsp中

B. 形如 $Imm(r_b, r_i, s)$ 的寻址格式为比例变址寻址，例如$10(\%\mathrm{eax}, \%\mathrm{ebx}, 4)$

C. cltq指令是符号扩展指令，不需要操作数

D. 将一个四字值弹出栈后需要将栈指针加8，因此，弹出一个四字的操作包括 $\mathrm{popq}\ \ \%\mathrm{rax}; \mathrm{addq}\ \ \$8, \%\mathrm{rsp}.$

答案：13.C(A: pushq S的意义是将栈指针移动后，将寄存器S中的数据保存到寄存器%rsp指向的内存地址B: 比例变址寻址的基址和变址寄存器都必须是64位寄存器C: 正确D: popq 包含了栈指针加8的操作，弹出一个四字的操作等同于 $movq\ (\%rsp), \%rax; addq\ \$8, \%rsp.$

转移控制

实际是程序计数器PC的转移

- 当控制从P转移至Q时，需要将程序计数器(PC)的值设为被调用者Q的起始地址，并把稍后返回时，需在P中继续执行的代码的地址压入栈中，对应的指令是call

指令		描述
call	*Label*	过程调用
call	**Operand*	过程调用
ret		从过程调用中返回

- 与jmp指令类似，有直接与间接两种形式的call指令，直接调用接的是被调用者的代码地址

```
1b64:   e8 99 05 00 00          call   2102 <explode_bomb>
```

- 间接调用则是以*接操作数指示符的形式跳转

```
call    *%rdx
```

转移控制

- 直接跳转call指令的编码，也与jmp指令类似，是PC相对的，右图中call指令的目标地址为0x2102，而其下一条指令的地址为0x1b69，用0x2102-0x1b69，得到0x0599，再依据小端法得到call指令的编码应为99 50 00 00

```
1b64:   e8 99 05 00 00          call   2102 <explode_bomb>
1b69:   eb e3                   jmp    1b4e <phase_6+0xf1>
```

- 右图中用同样的方法0x1b89-0x1c0e得到-(0x85)，即0xffffff7b，再依据小端法得到call指令的编码 7b ff ff ff

```
1c09:   e8 7b ff ff ff          call   1b89 <emulate_fsm>
1c0e:   44 39 e0                cmp    %r12d,%eax
```

- 被调用者返回时，应将控制转回调用者，对应的指令是ret，应返回的地址就是之前压入调用者栈帧的栈顶的返回地址

转移控制

- jmp类指令与call指令的区别

- jmp指令只是简单地将控制转移至目标处，并一直进行下去

- call指令与ret成对出现，call相当于push+jmp，而ret相当于pop+jmp，push和pop的内容都是返回地址

- 这意味着call指令一定会返回至调用者处，再继续执行代码，而jmp则是跳到哪就在哪一直执行下去

转移控制

- 我们都知道返回地址应当储存在调用者的栈帧的栈顶，那么有没有其他方式来存放返回地址呢？
- 在x64中没有

- 放在寄存器中？

- 可用的寄存器只有16个，并且都有各自的用途，没有空余的寄存器用来存放返回地址
- 此外，如果有个寄存器叫做ra，在过程a调用过程b时存放a的返回地址，那么如果过程b调用了过程c，要么把ra覆盖掉（显然不行），要么再占用一个新的寄存器（这就回到了使用栈的起点）

转移控制

- 间接跳转在什么时候可能被用到？

```c
volatile long g;

#include <stdio.h>

void f() {
    printf("f\n");
}

void g() {
    printf("g\n");
}

int main() {
    void (*function)() = f;
    (*function)();
    return 0;
}
```

```
14          .string "g"
15  g:
16          pushq   %rbp
17          movq    %rsp, %rbp
18          movl    $.LC1, %edi
19          call    puts
20          nop
21          popq    %rbp
22          ret
23  main:
24          pushq   %rbp
25          movq    %rsp, %rbp
26          subq    $16, %rsp
27          movq    $f, -8(%rbp)
28          movq    -8(%rbp), %rdx
29          movl    $0, %eax
30          call    *%rdx
31          movl    $0, %eax
32          leave
33          ret
```

- 函数指针

6. 有如下代码段：

```
int func(int x, int y);
int (*p) (int a, int b);
p = func;
p(0,0);
```

对应的下列 x86-64 过程调用正确的是：

A. call *%rax　　B. call (%rax)

C. call *(%rax)　　D. call func

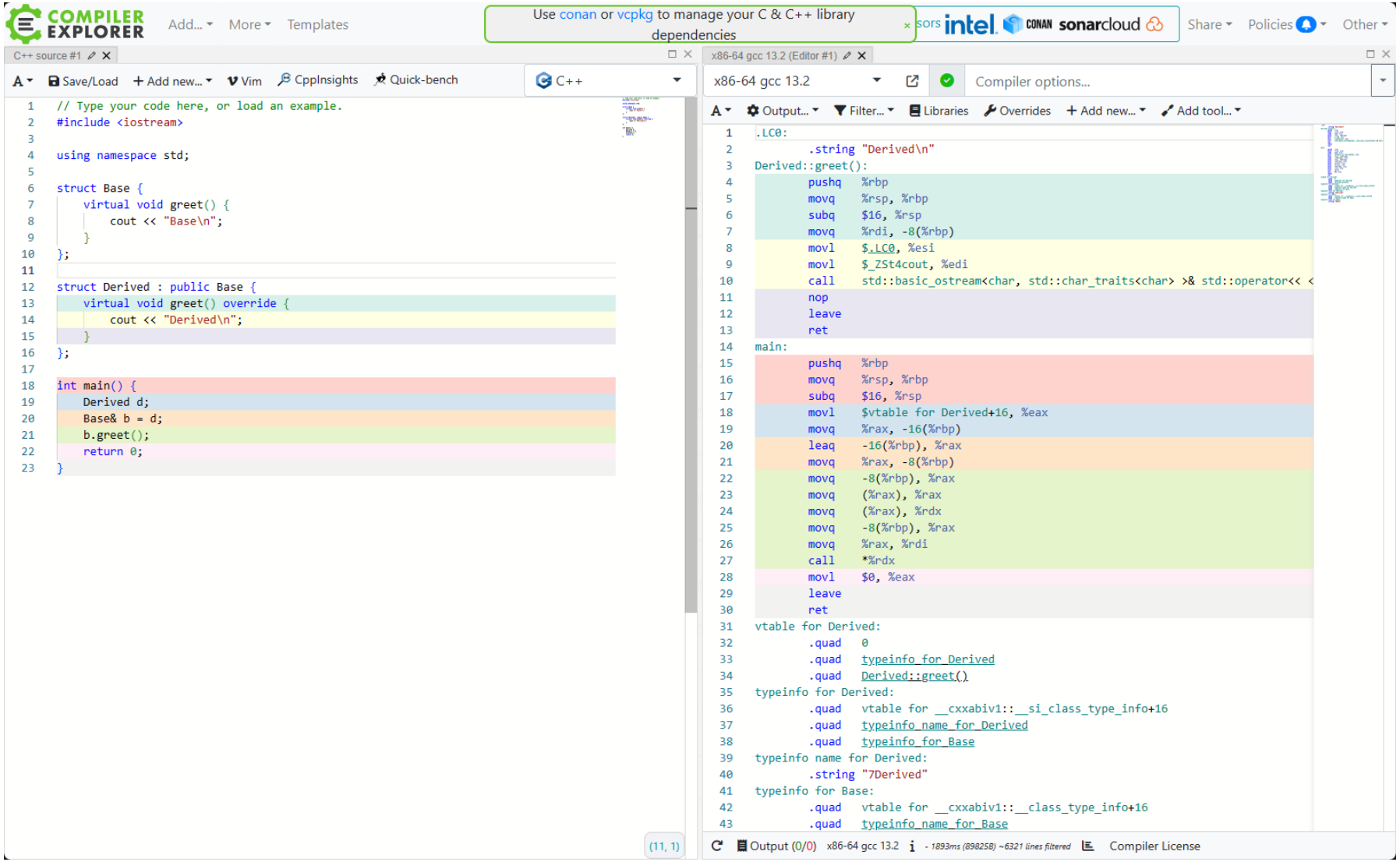
答案：A/D 都对

解析：用 O0 编译可以得到 A，用 Og 编译可以得到 D。

转移控制

（接上页）

- 虚函数的多态性
- 右图中调用的是子类的greet函数：Derived::greet()



数据传送

- 在过程调用中传递参数时，前6个参数在寄存器中传递，而从第7个开始在栈帧中传送

- 参数在寄存器中的顺序：第1至第6个参数：%rdi、%rsi、%rdx、%rcx、%r8、%r9

- 参数在栈帧中的顺序：第7个参数在栈顶（低地址），剩下参数的地址依次递增，并向8字节对齐（每8个字节为1个单位，放得下就继续放，放不下时再使用一个新的8字节）

- 返回值的传递：一般由被调用者在返回前存入%rax

```
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;
}
```

a）C代码

```
    void proc(a1, a1p, a2, a2p, a3, a3p, a4, a4p)
    Arguments passed as follows:
      a1  in %rdi          (64 bits)
      a1p in %rsi          (64 bits)
      a2  in %edx          (32 bits)
      a2p in %rcx          (64 bits)
      a3  in %r8w          (16 bits)
      a3p in %r9           (64 bits)
      a4  at %rsp+8        ( 8 bits)
      a4p at %rsp+16       (64 bits)
1   proc:
2     movq    16(%rsp), %rax    Fetch a4p    (64 bits)
3     addq    %rdi, (%rsi)      *a1p += a1   (64 bits)
4     addl    %edx, (%rcx)      *a2p += a2   (32 bits)
5     addw    %r8w, (%r9)       *a3p += a3   (16 bits)
6     movl    8(%rsp), %edx     Fetch a4     ( 8 bits)
7     addb    %dl, (%rax)       *a4p += a4   ( 8 bits)
8     ret                       Return
```

b）生成的汇编代码

各种数据的传送方式

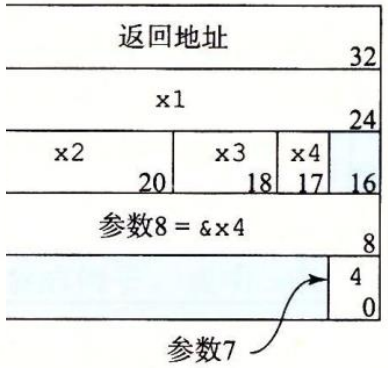
- int型：直接将值传入寄存器或栈中
- 指针：指针可以看作是一个64位的整数，也可以直接传入寄存器或栈中
- 结构体(struct)：若传结构体指针，则与上述方式一样，只需传结构体的首地址；若结构体拷贝，则会在栈中根据传入的结构体依次生成其中的变量拷贝，生成一个完全一样的结构体

```
struct st
{
    int number;char ch;
};
function1(st s1){…}//传拷贝
function2(st* s2){…}//传指针
```

栈上的局部存储

- 在一些情况下，局部数据必须在内存中存储
  - 寄存器不足以存放时
  - 对数据使用了&，取其地址，它就必须拥有一个地址
  - 数据类型为数组或结构
- 注意：此时数据的延展方向与构造参数时相反，先存储的值地址高，后存储的地址低

```
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);
}
```




```
    long call_proc()
1   call_proc:
      Set up arguments to proc
2     subq    $32, %rsp          Allocate 32-byte stack frame
3     movq    $1, 24(%rsp)       Store 1 in &x1
4     movl    $2, 20(%rsp)       Store 2 in &x2
5     movw    $3, 18(%rsp)       Store 3 in &x3
6     movb    $4, 17(%rsp)       Store 4 in &x4
7     leaq    17(%rsp), %rax     Create &x4
8     movq    %rax, 8(%rsp)      Store &x4 as argument 8
9     movl    $4, (%rsp)         Store 4 as argument 7
10    leaq    18(%rsp), %r9      Pass &x3 as argument 6
11    movl    $3, %r8d           Pass 3 as argument 5
12    leaq    20(%rsp), %rcx     Pass &x2 as argument 4
13    movl    $2, %edx           Pass 2 as argument 3
14    leaq    24(%rsp), %rsi     Pass &x1 as argument 2
15    movl    $1, %edi           Pass 1 as argument 1
      Call proc
16    call    proc
      Retrieve changes to memory
17    movslq  20(%rsp), %rdx     Get x2 and convert to long
18    addq    24(%rsp), %rdx     Compute x1+x2
19    movswl  18(%rsp), %eax     Get x3 and convert to int
20    movsbl  17(%rsp), %ecx     Get x4 and convert to int
21    subl    %ecx, %eax         Compute x3-x4
22    cltq                       Convert to long
23    imulq   %rdx, %rax         Compute (x1+x2) * (x3-x4)
24    addq    $32, %rsp          Deallocate stack frame
25    ret                        Return
```

栈上的局部存储

右图是phase_2中执行的一些代码，其中通过read_six_numbers读取的6个数字依次存储在栈中:(%rsp) $0x4(%rsp) $0x8(%rsp) $0xc(%rsp) $0x10(%rsp) $0x14(%rsp),从下面调用的过程中也可以看出，比如设%rsp为int* ptr,%rax为int x,那么(%rsp,%rax,4)表示*(ptr+x)

```
17ed:   e8 96 09 00 00          call   2188 <read_six_numbers>
17f2:   83 3c 24 00             cmpl   $0x0,(%rsp)
17f6:   75 07                   jne    17ff <phase_2+0x2e>
17f8:   83 7c 24 04 01          cmpl   $0x1,0x4(%rsp)
17fd:   74 05                   je     1804 <phase_2+0x33>
17ff:   e8 fe 08 00 00          call   2102 <explode_bomb>
1804:   bb 02 00 00 00          mov    $0x2,%ebx
1809:   eb 08                   jmp    1813 <phase_2+0x42>
180b:   e8 f2 08 00 00          call   2102 <explode_bomb>
1810:   83 c3 01                add    $0x1,%ebx
1813:   83 fb 05                cmp    $0x5,%ebx
1816:   7f 1b                   jg     1833 <phase_2+0x62>
1818:   48 63 d3                movslq %ebx,%rdx
181b:   8d 4b fe                lea    -0x2(%rbx),%ecx
181e:   48 63 c9                movslq %ecx,%rcx
1821:   8d 43 ff                lea    -0x1(%rbx),%eax
1824:   48 98                   cltq
1826:   8b 04 84                mov    (%rsp,%rax,4),%eax
1829:   03 04 8c                add    (%rsp,%rcx,4),%eax
182c:   39 04 94                cmp    %eax,(%rsp,%rdx,4)
182f:   74 df                   je     1810 <phase_2+0x3f>
1831:   eb d8                   jmp    180b <phase_2+0x3a>
```

寄存器中的局部存储

- 被调用者保存寄存器
  - 是：被调用者（callee）负责保存
  - 不是：被　调用者（caller）保存
  - 包括%rbx, %rbp, %r12~%r15
- 常见的例子：

```
1a63:    55                              push   %rbp
```

  - 将%rbp存入栈，就相当于为push之前的%rbp内的值做出保障，之后可以随意改动%rbp，只要返回之前将栈中保存的值pop回到%rbp中即可
- 除了被调用者保存、栈指针%rsp以外的寄存器都被定义为调用者保存寄存器，可以理解为这是P调用的子过程需要用到的值，所以保存这个值的责任由P承担

ASLR

- ASLR：Address Space Layout Randomization地址空间布局随机化是一项针对缓冲区溢出的安全保护技术。
- 借助ASLR，每次运行程序时相应进程的栈以及堆的初始地址都会随机变化（加上一个很大的随机数），从而加大黑客利用缓冲区溢出进行攻击的难度

```
0x1b03 <phase_6+166>    add    $0x1,%esi
0x1b06 <phase_6+169>    cmp    $0x5,%esi
0x1b09 <phase_6+172>    jg     0x1b19 <phase_6+188>
0x1b0b <phase_6+174>    mov    $0x1,%eax
0x1b10 <phase_6+179>    lea    0x6539(%rip),%rdx        # 0x8050 <node1>
0x1b17 <phase_6+186>    jmp    0x1af6 <phase_6+153>
0x1b19 <phase_6+188>    mov    0x20(%rsp),%rbx
0x1b1e <phase_6+193>    mov    %rbx,%rcx
0x1b21 <phase_6+196>    mov    $0x1,%eax
0x1b26 <phase_6+201>    jmp    0x1b3a <phase_6+221>
0x1b28 <phase_6+203>    movslq %eax,%rdx
0x1b2b <phase_6+206>    mov    0x20(%rsp,%rdx,8),%rdx
0x1b30 <phase_6+211>    mov    %rdx,0x8(%rcx)
0x1b34 <phase_6+215>    add    $0x1,%eax
0x1b37 <phase_6+218>    mov    %rdx,%rcx
0x1b3a <phase_6+221>    cmp    $0x5,%eax
0x1b3d <phase_6+224>    jle    0x1b28 <phase_6+203>
0x1b3f <phase_6+226>    movq   $0x0,0x8(%rcx)
0x1b47 <phase_6+234>    mov    $0x0,%ebp
0x1b4c <phase_6+239>    jmp    0x1b55 <phase_6+248>
0x1b4e <phase_6+241>    mov    0x8(%rbx),%rbx
0x1b52 <phase_6+245>    add    $0x1,%ebp
0x1b55 <phase_6+248>    cmp    $0x4,%ebp
0x1b58 <phase_6+251>    jg     0x1b6b <phase_6+270>
0x1b5a <phase_6+253>    mov    0x8(%rbx),%rax
0x1b5e <phase_6+257>    mov    (%rax),%eax
```

```
0x56143bd114c9 <main>           endbr64
0x56143bd114cd <main+4>         push   %rbx
0x56143bd114ce <main+5>         cmp    $0x1,%edi
0x56143bd114d1 <main+8>         je     0x56143bd11524 <main+91>
0x56143bd114d3 <main+10>        mov    %rsi,%rbx
0x56143bd114d6 <main+13>        cmp    $0x2,%edi
0x56143bd114d9 <main+16>        jne    0x56143bd11556 <main+141>
0x56143bd114db <main+18>        mov    0x8(%rsi),%rdi
0x56143bd114df <main+22>        lea    0x33a2(%rip),%rsi        # 0x56143bd14888
0x56143bd114e6 <main+29>        call   0x56143bd11350 <fopen@plt>
0x56143bd114eb <main+34>        mov    %rax,0x6f9e(%rip)        # 0x56143bd18490 <infile>
0x56143bd114f2 <main+41>        test   %rax,%rax
0x56143bd114f5 <main+44>        je     0x56143bd11534 <main+107>
0x56143bd114f7 <main+46>        call   0x56143bd11df0 <initialize_bomb>
0x56143bd114fc <main+51>        mov    %rax,%rbx
0x56143bd114ff <main+54>        cmpl   $0x2023fa11,(%rax)
0x56143bd11505 <main+60>        je     0x56143bd11579 <main+176>
0x56143bd11507 <main+62>        lea    0x2b7a(%rip),%rsi        # 0x56143bd14088
0x56143bd1150e <main+69>        mov    $0x1,%edi
0x56143bd11513 <main+74>        mov    $0x0,%eax
```

递归过程

递归调用自身与调用其他函数的差别不大，但其结构类似于循环结构，只是把负责循环执行的jmp类指令换为了call指令

妙处在于每次深层递归的执行结束后，返回值在%rax中，可以便捷地直接被继续使用

```
long rfact(long n)
{
    long result;
    if (n <= 1)
        result = 1;
    else
        result = n * rfact(n-1);
    return result;
}
```

a）C代码

```
    long rfact(long n)
    n in %rdi
1   rfact:
2     pushq   %rbx              Save %rbx
3     movq    %rdi, %rbx        Store n in callee-saved register
4     movl    $1, %eax          Set return value = 1
5     cmpq    $1, %rdi          Compare n:1
6     jle     .L35              If <=, goto done
7     leaq    -1(%rdi), %rdi    Compute n-1
8     call    rfact             Call rfact(n-1)
9     imulq   %rbx, %rax        Multiply result by n
10  .L35:                     done:
11    popq    %rbx              Restore %rbx
12    ret                       Return
```

b）生成的汇编代码

形象看待递归调用的过程

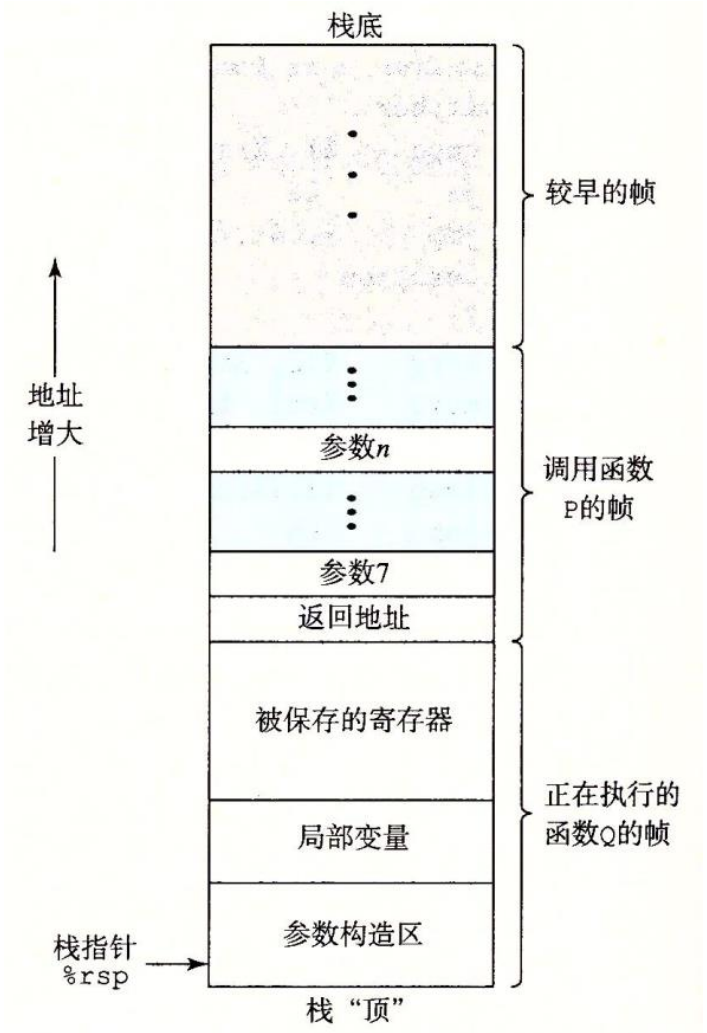
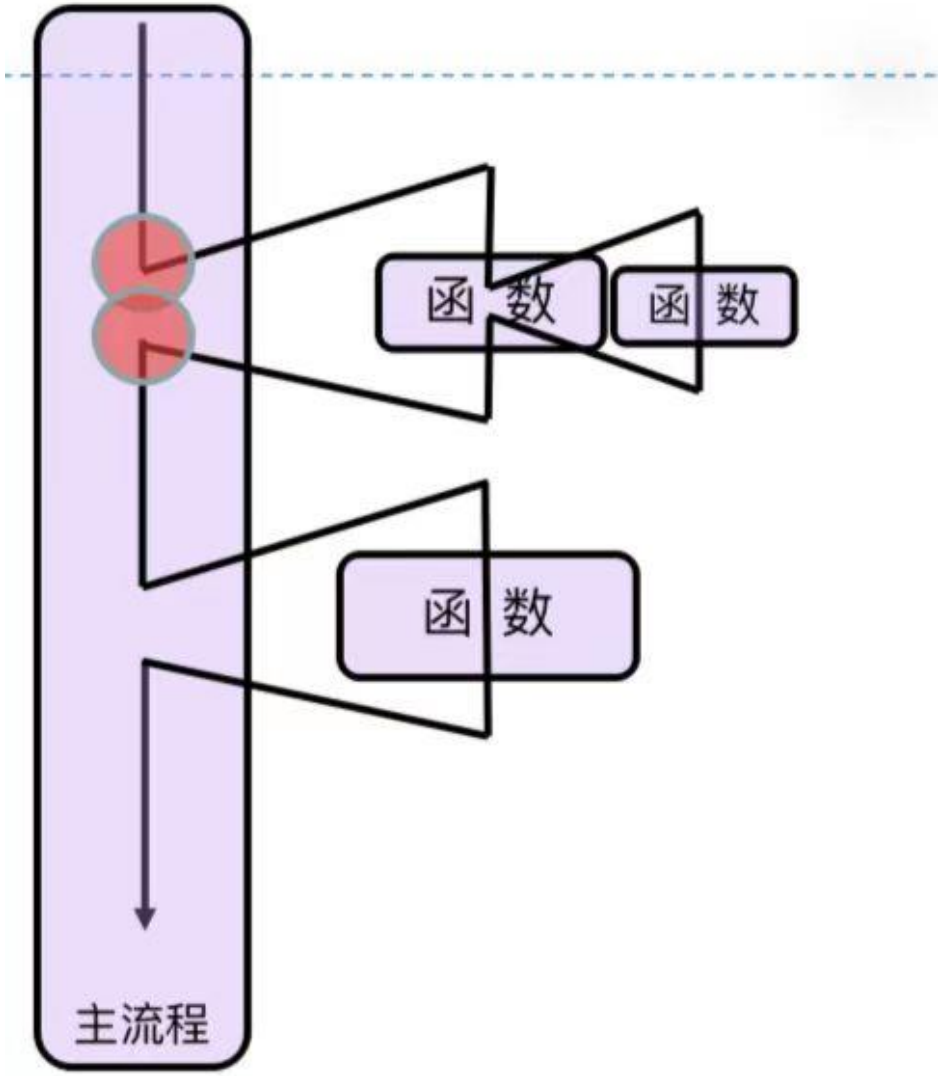


图 3-25 通用的栈帧结构(栈用来传递参数、存储返回信息、保存寄存器，以及局部存储。省略了不必要的部分)



（ C ）3. 下列关于通用目的寄存器和条件码的说法中，正确的是____.

A. %rbx, %rbp, %r12-r15是被调用者保存寄存器，其他寄存器是调用者保存寄存器.

B. xorq %rax %rax可以用于清空%rax；cmpq %rax, %rax可以用于判断%rax的值是否为零.

C. 调用一个过程时，最多可以传递6个整型参数，依次存储在(假设参数都是64位的)%rdi, %rsi, %rdx, %rcx, %r8, %r9中. 当需要更多的参数时，调用者过程可以在自己的栈帧里按照地址由低到高的顺序依次存储这些参数.

D. leaq指令不改变条件码；逻辑操作(如XOR)会将进位标志和溢出标志都设置为零；移位操作会把溢出标志设置为最后一个被移出(shift out)的位，而把进位标志设置为零.