


Q(y) 的值。在函数的开头，把这两个寄存器的值保存到栈中(第 2~3 行)。在第一次调用 Q 之前，把参数 x 复制到 %rbp(第 5 行)。在第二次调用 Q 之前，把这次调用的结果复制到 %rbx(第 8 行)。在函数的结尾，(第 13~14 行)，把它们从栈中弹出，恢复这两个被调用者保存寄存器的值。注意它们的弹出顺序与压入顺序相反，说明了栈的后进先出规则。

 **练习题 3.34** 一个函数 P 生成名为 a0~a7 的局部变量，然后调用函数 Q，没有参数。

GCC 为 P 的第一部分产生如下代码：

```

long P(long x)
x in %rdi
1  P:
2      pushq   %r15
3      pushq   %r14
4      pushq   %r13
5      pushq   %r12
6      pushq   %rbp
7      pushq   %rbx
8      subq    $24, %rsp
9      movq    %rdi, %rbx
10     leaq    1(%rdi), %r15
11     leaq    2(%rdi), %r14
12     leaq    3(%rdi), %r13
13     leaq    4(%rdi), %r12
14     leaq    5(%rdi), %rbp
15     leaq    6(%rdi), %rax
16     movq    %rax, (%rsp)
17     leaq    7(%rdi), %rdx
18     movq    %rdx, 8(%rsp)
19     movl    $0, %eax
20     call    Q

```

- A. 确定哪些局部值存储在被调用者保存寄存器中。
- B. 确定哪些局部变量存储在栈上。
- C. 解释为什么不能把所有的局部值都存储在被调用者保存寄存器中。

3.7.6 递归过程

前面已经描述的寄存器和栈的惯例使得 x86-64 过程能够递归地调用它们自身。每个过程调用在栈中都有它自己的私有空间，因此多个未完成调用的局部变量不会相互影响。此外，栈的原则很自然地就提供了适当的策略，当过程被调用时分配局部存储，当返回时释放存储。

图 3-35 给出了递归的阶乘函数的 C 代码和生成的汇编代码。可以看到汇编代码使用寄存器 %rbx 来保存参数 n，先把已有的值保存在栈上(第 2 行)，随后在返回前恢复该值(第 11 行)。根据栈的使用特性和寄存器保存规则，可以保证当递归调用 rfact(n-1) 返回时(第 9 行)，(1)该次调用的结果会保存在寄存器 %rax 中，(2)参数 n 的值仍然在寄存器 %rbx 中。把这两个值相乘就能得到期望的结果。

从这个例子我们可以看到，递归调用一个函数本身与调用其他函数是一样的。栈规则提供了一种机制，每次函数调用都有它自己私有的状态信息(保存的返回位置和被调用者保存寄存器的值)存储空间。如果需要，它还可以提供局部变量的存储。栈分配和释放的