

为了管理变长栈帧，x86-64 代码使用寄存器 `%rbp` 作为帧指针 (frame pointer) (有时称为基指针 (base pointer)，这也是 `%rbp` 中 `bp` 两个字母的由来)。当使用帧指针时，栈帧的组织结构与图 3-44 中函数 `vframe` 的情况一样。可以看到代码必须把 `%rbp` 之前的值保存到栈中，因为它是一个被调用者保存寄存器。然后在函数的整个执行过程中，都使得 `%rbp` 指向那个时刻栈的位置，然后用固定长度的局部变量 (例如 `i`) 相对于 `%rbp` 的偏移量来引用它们。

图 3-43b 是 GCC 为函数 `vframe` 生成的部分代码。在函数的开始，代码建立栈帧，并为数组 `p` 分配空间。首先把 `%rbp` 的当前值压入栈中，将 `%rbp` 设置为指向当前的栈位置 (第 2~3 行)。然后，在栈上分配 16 个字节，其中前 8 个字节用于存储局部变量 `i`，而后 8 个字节是未被使用的。接着，为数组 `p` 分配空间 (第 5~11 行)。练习题 3.49 探讨了分配多少空间以及将 `p` 放在这段空间的什么位置。当程序到第 11 行的时候，已经 (1) 在栈上分配了 $8n$ 字节，并 (2) 在已分配的区域内放置好数组 `p`，至少有 $8n$ 字节可供其使用。


初始化循环的代码展示了如何引用局部变量 `i` 和 `p` 的例子。第 13 行表明数组元素 `p[i]` 被设置为 `q`。该指令用寄存器 `%rcx` 中的值作为 `p` 的起始地址。我们可以看到修改局部变量 `i` (第 15 行) 和读局部变量 (第 17 行) 的例子。`i` 的地址是引用 `-8(%rbp)`，也就是相对于帧指针偏移量为 `-8` 的地方。

在函数的结尾，`leave` 指令将帧指针恢复到它之前的值 (第 20 行)。这条指令不需要参数，等价于执行下面两条指令：

```
movq %rbp, %rsp    Set stack pointer to beginning of frame
popq %rbp          Restore saved %rbp and set stack ptr
                    to end of caller's frame
```

也就是，首先把栈指针设置为保存 `%rbp` 值的位置，然后把该值从栈中弹出到 `%rbp`。这个指令组合具有释放整个栈帧的效果。

在较早版本的 x86 代码中，每个函数调用都使用了帧指针。而现在，只在栈帧长可变的情况下才使用，就像函数 `vframe` 的情况一样。历史上，大多数编译器在生成 IA32 代码时会使用帧指针。最近的 GCC 版本放弃了这个惯例。可以看到把使用帧指针的代码和不使用帧指针的代码混在一起是可以的，只要所有的函数都把 `%rbp` 当做被调用者保存寄存器来处理即可。

 **练习题 3.49** 在这道题中，我们要探究图 3-43b 第 5~11 行代码背后的逻辑，它分配了变长大小的数组 `p`。正如代码的注释表明， s_1 表示执行第 4 行的 `subq` 指令之后栈指针的地址。这条指令为局部变量 `i` 分配空间。 s_2 表示执行第 7 行的 `subq` 指令之后栈指针的值。这条指令为局部数组 `p` 分配存储。最后， p 表示第 10~11 行的指令赋给寄存器 `%r8` 和 `%rcx` 的值。这两个寄存器都用来引用数组 `p`。

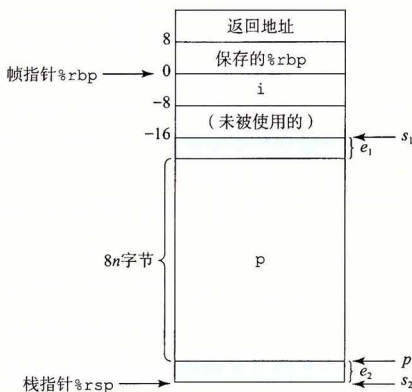


图 3-44 函数 `vframe` 的栈帧结构 (该函数使用寄存器 `%rbp` 作为帧指针。图右边的注释供练习题 3.49 所用)