

对于call指令会做的几件事：

1~减少栈指针，指的是控制程序运行阶段的那个栈。由于这个栈是有高位生长的，所以这个减少栈指针可以看做是将这个栈指针指向了下一条要执行的语句，如果是在程序中，我们可以视作是其指向了调用语句的下一条语句（地址）

这里需要注意的是，对于这个操作的意义·，设计到之前提到的rsp寄存器，这个寄存器是一个栈顶指针，起着PC的功能，在我们对栈进行压入时这个寄存器也会跟着进行改变

2~接下来程序**不会**根据PC的值来跳转到对应的位置，但是PC还是会加的，也就是说在执行对应函数体时，对应的栈指针指向的其实就是对应调用函数的下一条语句了，在这里，对于call指令需要注意，这里是由cpu直接对PC的值进行修改，而不是通过栈指针的逻辑等它自动指向下一个，这里的cpu直接将PC中的地址修改为对应调用函数的地址，记住，rsp只存了一个值，这里就实现了从调用处到目标函数的跳转。而直到ret之后，PC才会自动+1子类的跳转到原调用处的下一条语句

对于ret指令会做的几件事：

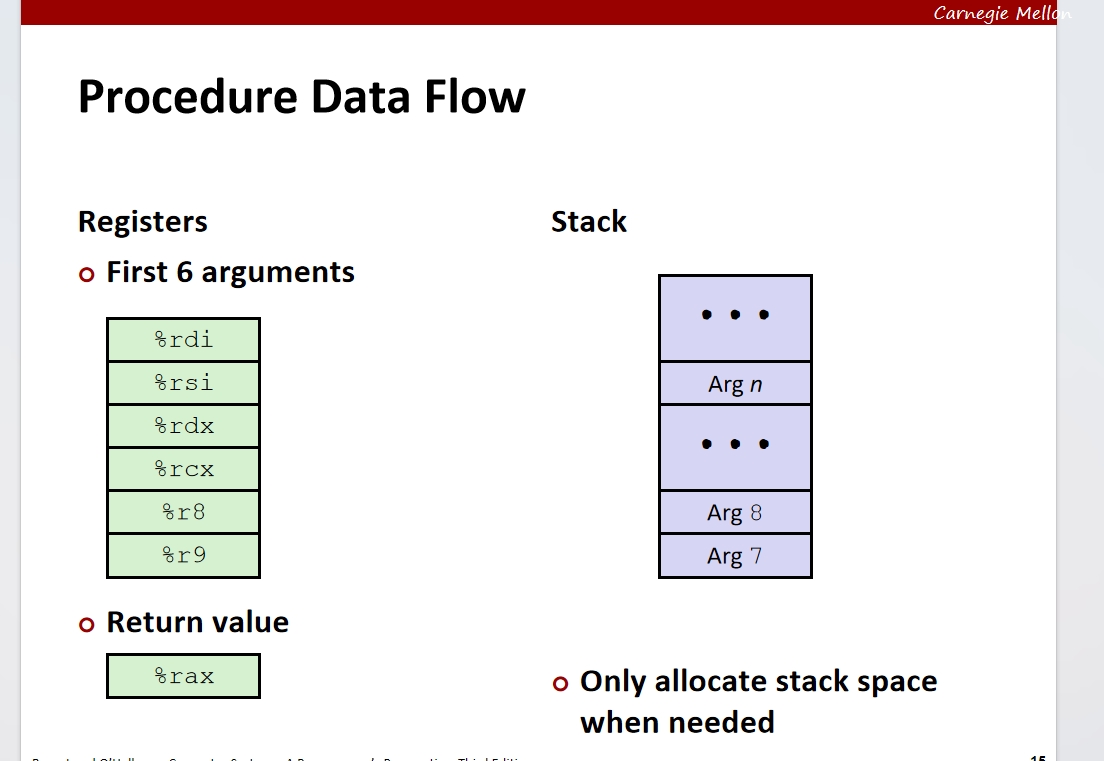
1~对栈进行pop操作，这个操作会弹出当前的栈顶指针，而前面说过，这个指针指向的其实就是之前调用call语句的下一条语句了，

2~PC会被更新为这个被弹出来的地址，此时就意味着程序已经跳回到了调用后的第一条语句，接下来会接着继续执行

对这里的调用返回进行一些拙劣的解释：

1~首先明白一点，无论实在那个函数体内，PC的值都是自动+1的，而对于PC逻辑的控制，都是有CPU决定的，包括main函数和被调用函数，在执行期间，PC的值都自动加1；

2~在函数被调用时，PC自动+1，使得程序正常运行，当遇到call语句时，CPU会额外进行一步，它会将call语句中的函数地址(程序自动找到的)替换当前PC的值，使得程序转到想要的执行函数体中去，接下来，PC的值还是自动+1，知道遇到ret函数，此时会将函数栈中的元素压出并将PC的值替换为这个值，这是就实现了函数的返回



在传递参数中，约定前6个参数会有寄存器来传递，而不是由栈来传递，而这样做的原因就是为了提高效率，至于哪几个寄存器，就是上图中左边的6个寄存器

一般来说，如果参数不过6个，其寄存器的选择遵守上面给出的顺序

但**特别**需要注意一点，即使在传递过程中，可能使用寄存器来传递参数，但是在一个函数对应栈帧的构建中，其是包含了所有参数的数据的，这是为了保证数据的不丢失

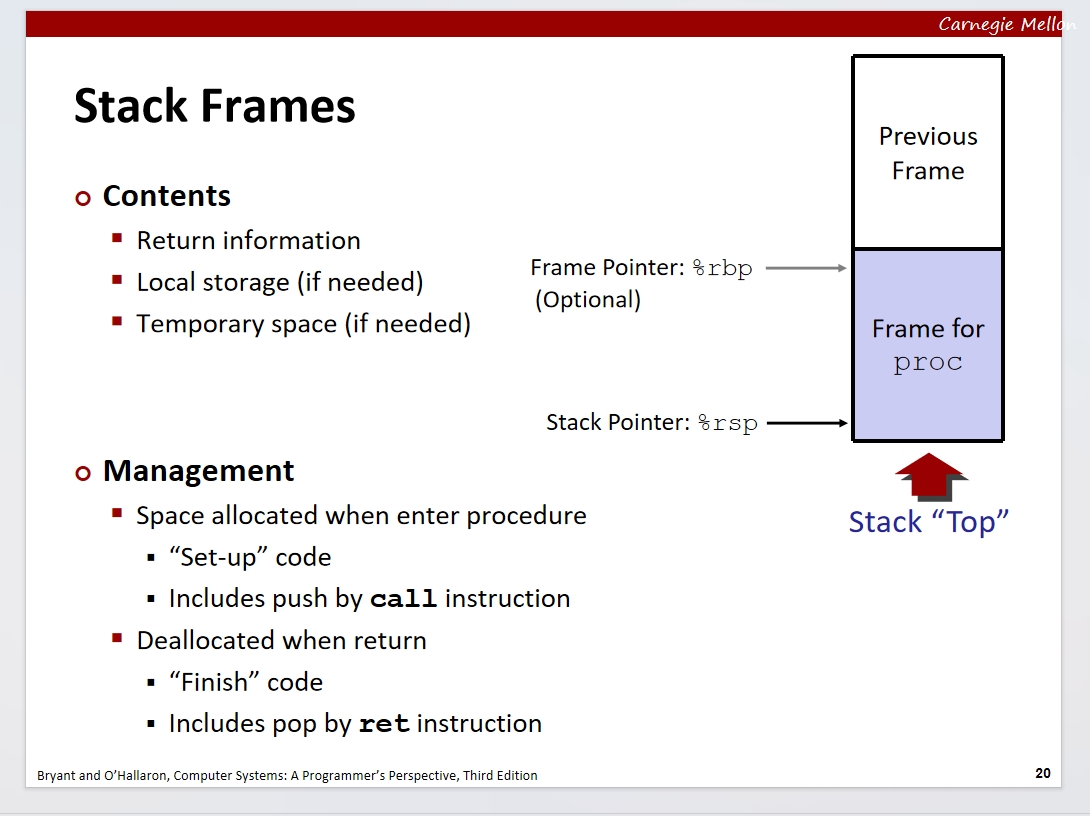
在函数开始之前，所有的参数数据都保存在了栈帧中，而函数的执行可以抽象的认为是在栈帧中进行的，所以可以保证参数传递的准确性

在返回参数中，要求使用rax寄存器，需要注意的是，这里的返回值只允许整形或者指针类型，至于其他的类型，一般也有对应的寄存器规定，并且是可能使用到栈来传递返回数据的

而之所以使用寄存器而不使用栈的目的其实也很明显，就是说寄存器的速度比栈的速度快很多

规定把栈上用于特点call的每个内存块视为栈帧

有一个关键点需要注意的，在栈中我们需要的只是每个被调用且还未被返回的过程保留一个栈帧



在栈中，每个栈帧通常由俩个指针分割，一个是熟悉的栈指针rsp，一个是额外的帧指针，如上图的右边

需要注意的是，栈帧所在的栈是与之前函数调用的栈同一个的，或者说，栈帧与函数调用总是在一起的，一般来说，栈帧总是先于函数地址的创建和压入，在栈帧中，其储存了多种数据，包括但不限于 函数的返回地址，临时变量，栈指针，参数，局部变量，保存的寄存器状态等，也就是说，所有在函数体内部创建的局部变量，其的生命周期也跟栈帧的生命周期相绑定。而在这可以看出栈帧的大小其实是与这些多有的元素都相关的，

关于这个，可以举出一个数组的例子来说

对于一个在函数体内正常声明的数组，其在帧内的体现就是一个完整的数组节后，而如果声明的是一个动态数组，帧内的只会是该数组在堆中的地址，真正的数组会在堆中进行分配

一般来说，程序都限定了栈的最大深度，也就是说，不能无限的创建栈帧，这会导致栈溢出

同时由这一点也能推断出，栈帧的数量大小一定程度上是与栈帧的内存大小相关的

接下来讨论下**caller**和**callee**的情况

这个机制是为了保存caller的状态，使得在callee在执行过程中不会因为自身的逻辑而对caller的一些变量进行修改而改变返回后caller的状态。一般储存的这些变量都会是caller中声明的callee中将会使用的，并且不希望对其进行修改的，会放到一些寄存器中去，在返回时通过这些寄存器对数据进行恢复来进行实现

