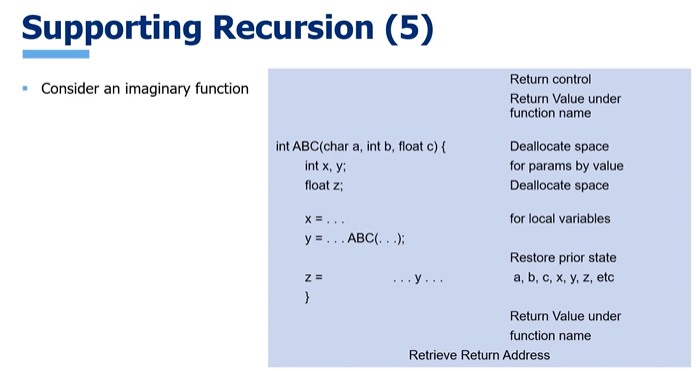


Many compilers simply indiscriminately save everything. Some compilers optimize recursion. One of the things they do when they optimize recursion is they pay more attention to the things that really needs to be saved at this point. So for instance, in this example, nothing happens to Z until after the recursive function call. So there really wouldn't be a lot of value in savings z on the stack. That's one type of choice that the compiler could make if optimizing the recursion, some variables aren't going to be used again. X might qualify in this instance. So there might be no point in saving X. We save the current state so that when we return later from the recursive call, we can recreate the current state and continue executing the rest of this function. (pass control)



But again, we have one extra step because this time it's a recursive call. And we're going to pop the stack and take all of those variables off of it to restore the prior state. And then we keep executing. So the way it's going to work is we come in, we do the work, we make a recursive call, saving the recurrent state. When we return from this recursive call, we restore the current state off the stack and keep executing until we hit that scope closure. So that's an overview of how we support recursion at the level of the compiler and the operating system. Sometimes, as we've discussed, you may have a problem that's naturally recursive, but you may not be in a situation to actually use recursion. We talked about using that as a basis for developing an iterative solution. Another thing that you might want to do is simulate recursion. It uses stacks, which we've actually already discussed. Start by writing your own stacks and you store items on the stack.