- There are three points worth noting here:
  • a function can be **called from more than one location**
  • a function call can **appear in procedure**. This **allows the nesting of functions** to an arbitrary depth
  • each function call is **matched by a return** in the called program

- We know that CALL instruction can branch from the current location to the beginning of a function to be executed. But how does the RETURN instruction know the return address that is the start of the remaining instructions in the procedure where it left.

---

**Stack**

- The way to save the return address is to use a stack.
- As we know, a stack is a contiguous section of memory following the LIFO strategy.
- When a function executes a call, it places the return address on the stack. When it execute return, it uses the address on the stack.
- Still use the above diagram to illustrate how to use stack to store return addresses. The stack of a computer when executing the main program looks like

![Stack Diagram]

- When the CALL instruction is executed, the return address, which is the address of the instruction after the current CALL instruction is pushed into the stack.
- When the RETURN instruction is executed, the top return address is popped and used by the RETURN instruction to find where the execution sequence should go.

- In addition to providing a return address, it is also often necessary to pass parameters, as functions usually have local variables and parameters.
- To better store those data for each function call, the stack pointer, frame pointer, and old frame pointer are used.
  • **Stack pointer**: point to the **top of the stack**
• **Frame pointer**: point to the *beginning of the current stack frame*. It’s fixed for the duration of the current function stack frame and convenient when a function is likely to move the stack pointer several times throughout the course of running the function.

• **Old frame pointer**: address of the *beginning of the previous frame*. It’s needed if the number of parameters to be stacked for each function is variable.

  Let’s use an example to illustrate. Assume $x_1$ and $x_2$ are the local variables in function $P$, and $y_1$ and $y_2$ are the local variables in function $Q$. $P$ calls $Q$ at some point in its function body.

- Generally speaking, when execute a function
  - Push space for the return value(s)
  - Push parameters
  - CALL: push return address (PC’s value) on the stack
  - Push old frame pointer (current value of the frame pointer)
  - Set the frame pointer (a register) to the stack pointer
  - Push local variables, if necessary
  - Push the values of any registers (e.g., CR) to be used
  - Execute the function
  - Pop the values of any registers used
  - Pop the local variables, if any
  - Pop the old stack frame and restore it (let the frame pointer have it)
  - RETURN: pops the return address to the PC
  - Pop the arguments
  - The top of the stack is the return value