```
Setting up jump table access
1       movl 8(%ebp),%ebx       Get x
2       addl $2,%eax
3       cmpl $6,%eax
4       ja .L10
5       jmp *.L11(,%eax,4)
```

```
Jump table for switch2
1       .L11:
2               .long .L4
3               .long .L10
4               .long .L5
5               .long .L6
6               .long .L8
7               .long .L8
8               .long .L9
```

Use the foregoing information to answer the following questions:

A. What were the values of the case labels in the switch statement body?

B. What cases had multiple labels in the C code?

3.7 Procedures

A procedure call involves passing both data (in the form of procedure parameters and return values) and control from one part of the code to another. In addition, it must allocate space for the local variables of the procedure on entry and deallocate them on exit. Most machines, including IA32, provide only simple instructions for transferring control to and from procedures. The passing of data and the allocation and deallocation of local variables is handled by manipulating the program stack.

3.7.1 Stack Frame Structure

IA32 programs make use of the program stack to support procedure calls. The stack is used to pass procedure arguments, to store return information, to save registers for later restoration, and for local storage. The portion of the stack allocated for a single procedure call is called a *stack frame*. Figure 3.17 diagrams the general structure of a stack frame. The topmost stack frame is delimited by two pointers, with register `%ebp` serving as the *frame pointer*, and register `%esp` serving as the *stack pointer*. The stack pointer can move while the procedure is executing, and hence most information is accessed relative to the frame pointer.

Suppose procedure `P` (the *caller*) calls procedure `Q` (the *callee*). The arguments to `Q` are contained within the stack frame for `P`. In addition, when `P` calls `Q`, the *return address* within `P` where the program should resume execution when it returns from `Q` is pushed on the stack, forming the end of `P`'s stack frame. The stack frame for `Q` starts with the saved value of the frame pointer (i.e., `%ebp`). followed by copies of any other saved register values.

Procedure `Q` also uses the stack for any local variables that cannot be stored in registers. This can occur for the following reasons:

- There are not enough registers to hold all of the local data.
- Some of the local variables are arrays or structures and hence must be accessed by array or structure references.

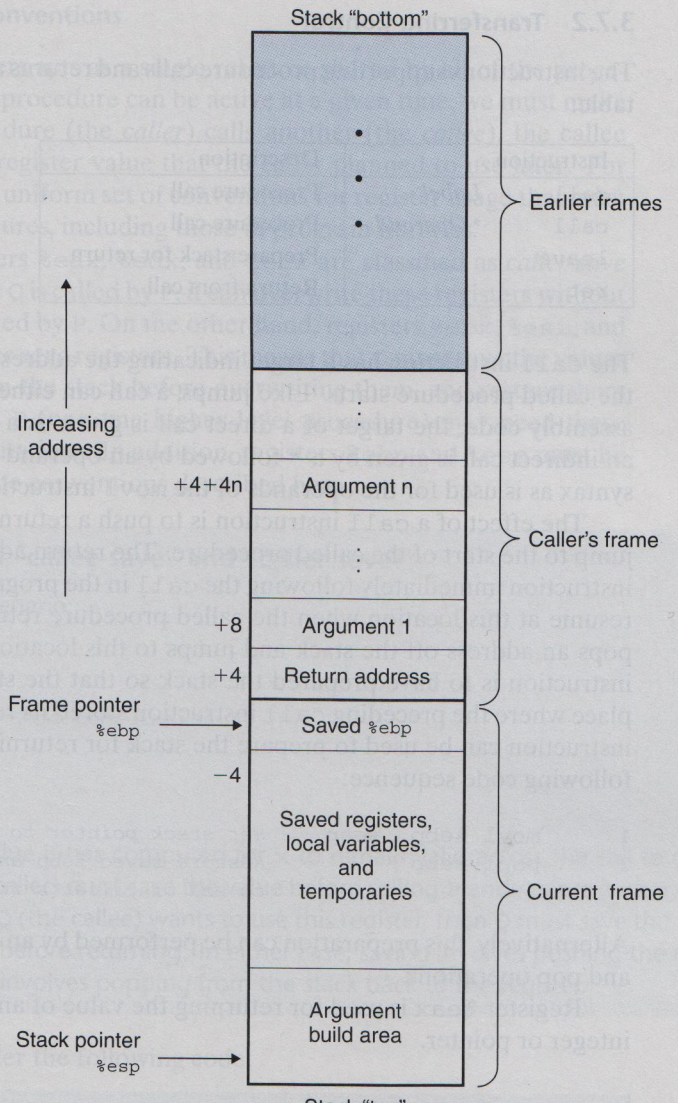

Figure 3.17
Stack frame structure. The stack is used for passing arguments, for storing return information, for saving registers, and for local storage.

- The address operator '`&`' is applied to one of the local variables, and hence we must be able to generate an address for it.

Finally, `Q` will use the stack frame for storing arguments to any procedures it calls.

As described earlier, the stack grows toward lower addresses and the stack pointer `%esp` points to the top element of the stack. Data can be stored on and retrieved from the stack using the `pushl` and `popl` instructions. Space for data with no specified initial value can be allocated on the stack by simply decrementing the stack pointer by an appropriate amount. Similarly, space can be deallocated by incrementing the stack pointer.