```
  new_node->value = n;
  new_node->next = list;
  return new_node;
}
```

Note that add_to_list doesn't modify the list pointer. Instead, it returns a pointer to the newly created node (now at the beginning of the list). When we call add_to_list, we'll need to store its return value into first:

```
first = add_to_list(first, 10);
first = add_to_list(first, 20);
```

These statements add nodes containing 10 and 20 to the list pointed to by first. Getting add_to_list to update first directly, rather than return a new value for first, turns out to be tricky. We'll return to this issue in Section 17.6.

The following function uses add_to_list to create a linked list containing numbers entered by the user:

```
struct node *read_numbers(void)
{
  struct node *first = NULL;
  int n;

  printf("Enter a series of integers (0 to terminate): ");
  for (;;) {
    scanf("%d", &n);
    if (n == 0)
      return first;
    first = add_to_list(first, n);
  }
}
```

The numbers will be in reverse order within the list, since first always points to the node containing the last number entered.

Searching a Linked List

Once we've created a linked list, we may need to search it for a particular piece of data. Although a while loop can be used to search a list, the for statement is often superior. We're accustomed to using the for statement when writing loops that involve counting, but its flexibility makes the for statement suitable for other tasks as well, including operations on linked lists. Here's the customary way to visit the nodes in a linked list, using a pointer variable p to keep track of the "current" node:

idiom

```
for (p = first; p != NULL; p = p->next)
  ...
```

The assignment

```
p = p->next
```