

```
(*new_node).value = 10;
```

The `->` operator is a combination of the `*` and `.` operators; it performs indirection on `new_node` to locate the structure that it points to, then selects the `value` member of the structure.

lvalues ► 4.2

The `->` operator produces an lvalue, so we can use it wherever an ordinary variable would be allowed. We've just seen an example in which `new_node->value` appears on the left side of an assignment. It could just as easily appear in a call of `scanf`:

```
scanf("%d", &new_node->value);
```

Notice that the `&` operator is still required, even though `new_node` is a pointer. Without the `&`, we'd be passing `scanf` the *value* of `new_node->value`, which has type `int`.

Inserting a Node at the Beginning of a Linked List

One of the advantages of a linked list is that nodes can be added at any point in the list: at the beginning, at the end, or anywhere in the middle. The beginning of a list is the easiest place to insert a node, however, so let's focus on that case.

If `new_node` is pointing to the node to be inserted, and `first` is pointing to the first node in the linked list, then we'll need two statements to insert the node into the list. First, we'll modify the new node's `next` member to point to the node that was previously at the beginning of the list:

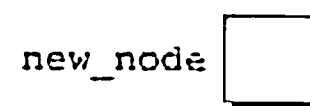
```
new_node->next = first;
```

Second, we'll make `first` point to the new node:

```
first = new_node;
```

Will these statements work if the list is empty when we insert a node? Yes, fortunately. To make sure this is true, let's trace the process of inserting two nodes into an empty list. We'll insert a node containing the number 10 first, followed by a node containing 20. In the figures that follow, null pointers are shown as diagonal lines.

```
first = NULL;
```



```
new_node = malloc(sizeof(struct node));
```

