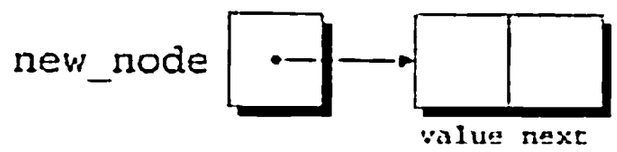
We'll use `malloc` to allocate memory for the new node, saving the return value in `new_node`:

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

`new_node` now points to a block of memory just large enough to hold a `node` structure:



Be careful to give `sizeof` the name of the *type* to be allocated, not the name of a *pointer* to that type:

```
new_node = malloc(sizeof(new_node));    /*** WRONG ***/
```

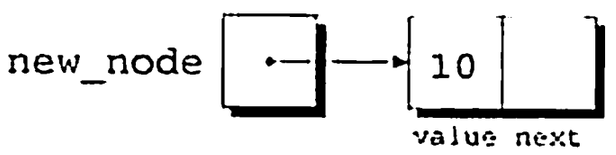
Q&A

The program will still compile, but `malloc` will allocate only enough memory for a *pointer* to a `node` structure. The likely result is a crash later, when the program attempts to store data in the node that `new_node` is presumably pointing to.

Next, we'll store data in the `value` member of the new node:

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

Here's how the picture will look after this assignment:



To access the `value` member of the node, we've applied the indirection operator `*` (to reference the structure to which `new_node` points), then the selection operator `.` (to select a member of the structure). The parentheses around `*new_node` are mandatory because the `.` operator would otherwise take precedence over the `*` operator.

table of operators ➤*Appendix A*

The -> Operator

Before we go on to the next step, inserting a new node into a list, let's take a moment to discuss a useful shortcut. Accessing a member of a structure using a pointer is so common that C provides a special operator just for this purpose. This operator, known as ***right arrow selection,*** is a minus sign followed by >. Using the `->` operator, we can write

```
new_node->value = 10;
```

instead of