advances the p pointer from one node to the next. An assignment of this form is invariably used in C when writing a loop that traverses a linked list.

Let's write a function named `search_list` that searches a list (pointed to by the parameter `list`) for an integer `n`. If it finds `n`, `search_list` will return a pointer to the node containing `n`; otherwise, it will return a null pointer. Our first version of `search_list` relies on the "list-traversal" idiom:

```
struct node *search_list(struct node *list, int n)
{
  struct node *p;

  for (p = list; p != NULL; p = p->next)
    if (p->value == n)
      return p;
  return NULL;
}
```

Of course, there are many other ways to write `search_list`. One alternative would be to eliminate the `p` variable, instead using `list` itself to keep track of the current node:

```
struct node *search_list(struct node *list, int n)
{
  for (; list != NULL; list = list->next)
    if (list->value == n)
      return list;
  return NULL;
}
```

Since `list` is a copy of the original list pointer, there's no harm in changing it within the function.

Another alternative is to combine the `list->value == n` test with the `list != NULL` test:

```
struct node *search_list(struct node *list, int n)
{
  for (; list != NULL && list->value != n; list = list->next)
    ;
  return list;
}
```

Since `list` is `NULL` if we reach the end of the list, returning `list` is correct even if we don't find `n`. This version of `search_list` might be a bit clearer if we used a `while` statement:

```
struct node *search_list(struct node *list, int n)
{
  while (list != NULL && list->value != n)
    list = list->next;
  return list;
}
```