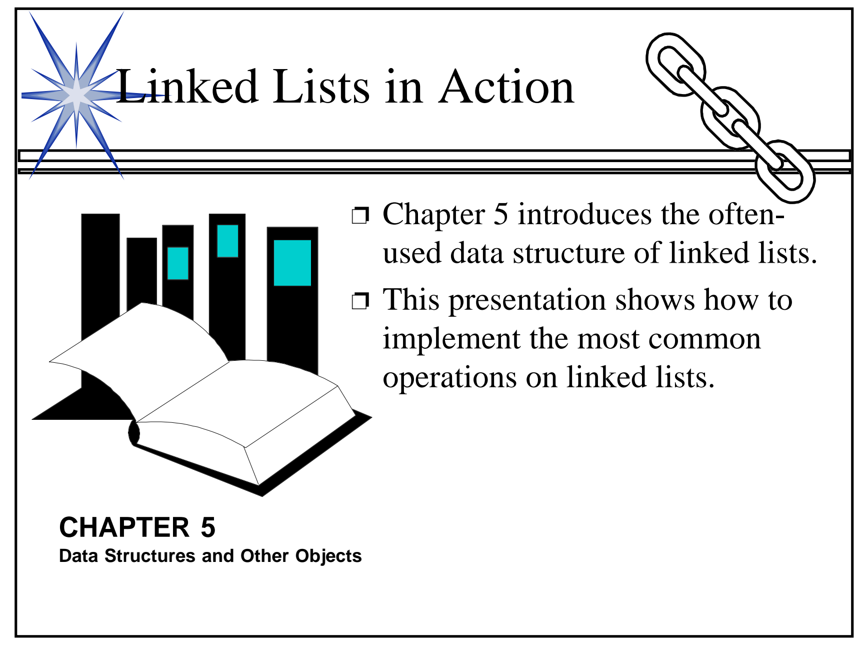


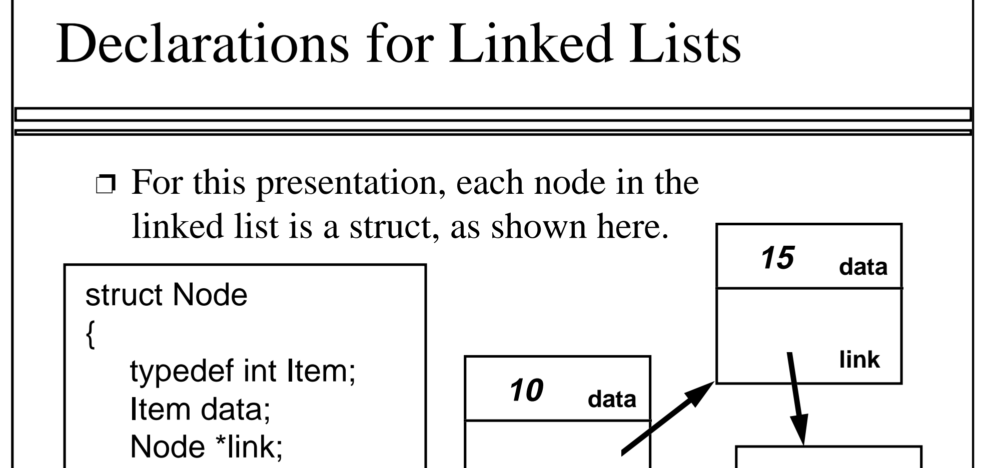
This lecture shows three linked list operation in detail. The operations are:

1. Adding a new node at the head of a linked list.

2. Adding a new node in the middle of a linked list.

3. Removing a node from a linked list.

The best time for this lecture is just after the students have been introduced to linked lists (Section 5.1), and before the complete linked list toolkit has been covered (Section 5.2).

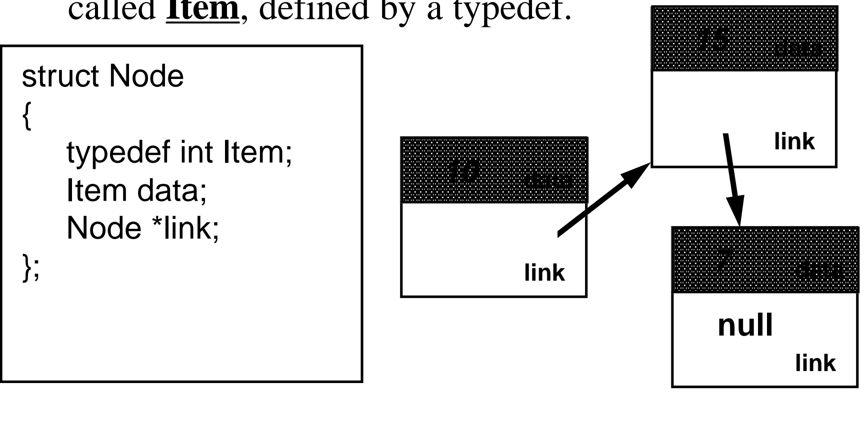


Here is a typical struct declaration that can be used to implement a linked list of integers, as described in Section 5.1 of the text.

Note that we have used a struct rather than a class. In a struct, members are public unless you specify otherwise. C++ programmers tend to use a struct only when all the members are public.

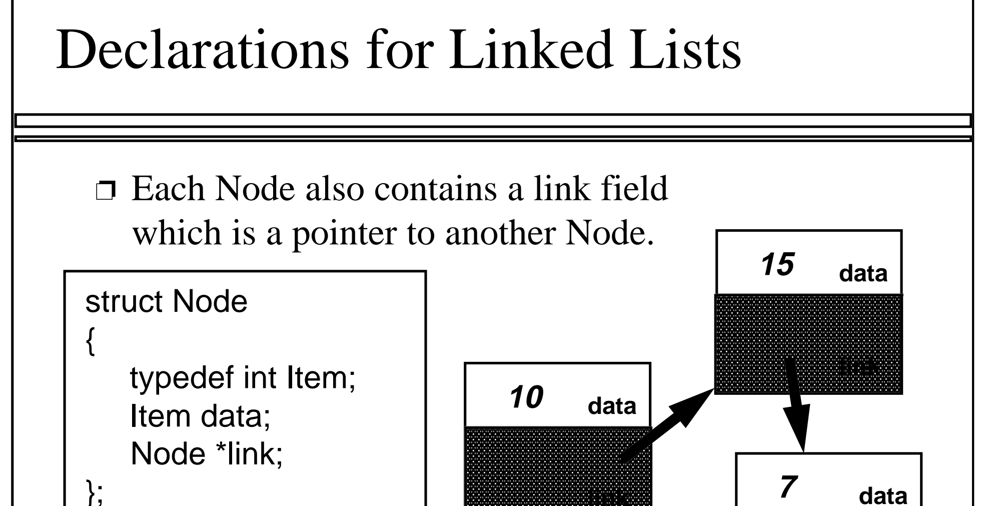
Declarations for Linked Lists

❒ The data portion of each node is a type called **Item**, defined by a typedef.



Within the struct, there is a type definition for a type called called "Item". The purpose of the Item data type is to tell us what kind of data resides in each of the nodes on the linked list. For example, if we want to create a linked list of real numbers, then we would declare typedef double Item; In this example, we want a linked list of integers, so we have declared typedef int Item;

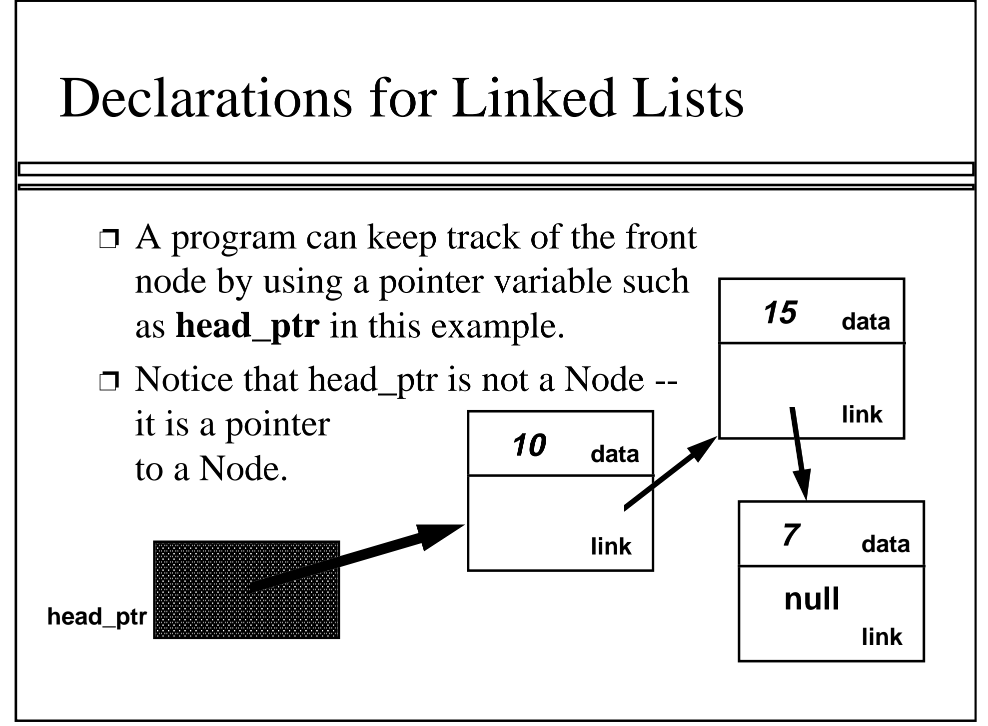
As you can see, the Item data type is used in the node declaration as the data type of a member variable called data. In the picture, I have drawn three nodes which might be part of a linked list of integers. The data portions of these nodes contain the integers 10, 15 and 7.



Also inside each Node is a second member variable called link. The purpose of the link member variable is to contain a pointer to the next Node in the sequence of nodes.

Question: What appears in the link field of the final node in a linked list such as this?

Answer: A special pointer value called null, which means "This pointer doesn't point to anything." This makes sense because there is no node after the final node.

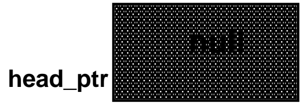


When a linked list is implemented, we generally keep track of the first node by using a special pointer called the "head pointer". The head pointer points to the first node in the linked list.

It is important to remember that head_ptr is not actually a node. It is just a single pointer, which points to the first actual node.

Declarations for Linked Lists

- A program can keep track of the front node by using a pointer variable such as **head_ptr**.
- Notice that head_ptr is not a Node -- it is a pointer to a Node.
- We represent the empty list by storing **null** in the Head pointer.



Also remember that it is possible for a linked list to have no nodes at all. This situation is called the "empty list".

Even with the empty list, we still have a head pointer, but the head pointer has no nodes to point to. So, instead of pointing to something, the head pointer of the empty list contains that special value, null.

Inserting a Node at the Front

```
void list_head_insert(Node*& head_ptr, const Node::Item& entry);
```

We want to add a new entry, 13, to the **front** of the linked list shown here.

In this lecture, I'll show you the workings of three functions to manipulate linked lists. The first function is called head_insert. The purpose of the function is to insert a new entry at the front of a linked list. The first parameter is the head point of the linked list declared here as:

Node*& head_ptr

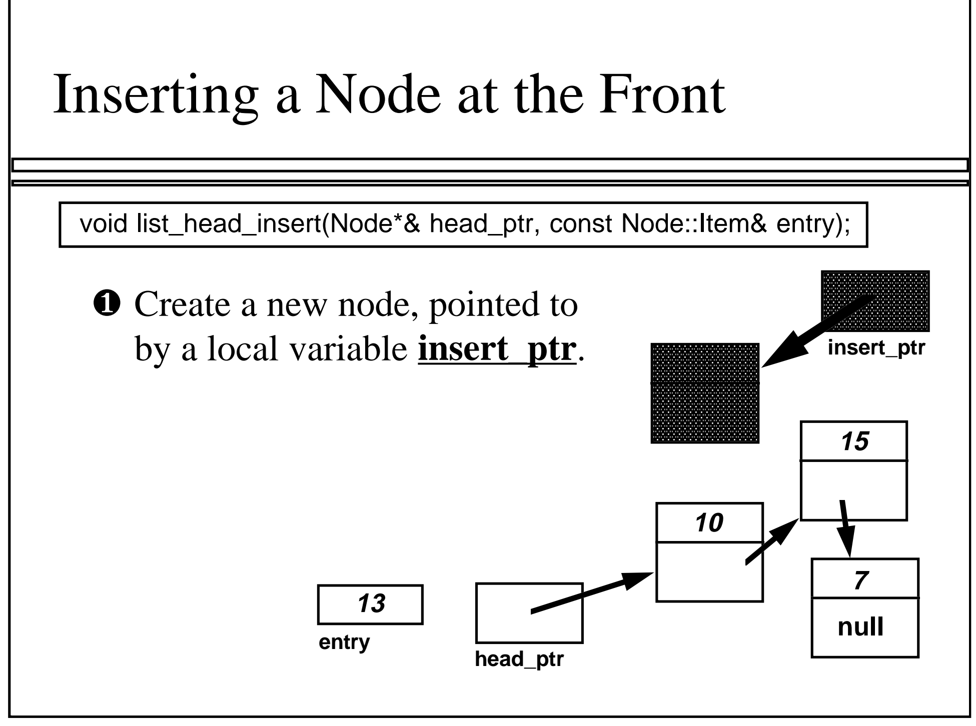
Notice that this is a pointer to a Node (Node*), but it is also a reference parameter (indicated by the "&"). The reason for the reference parameter is that the function will make the head pointer point to a new node (containing the new entry)--and we want the change to affect the actual head pointer back in the calling program.

The new entry itself is the the second parameter, declared here as:

const Node::Item& entry

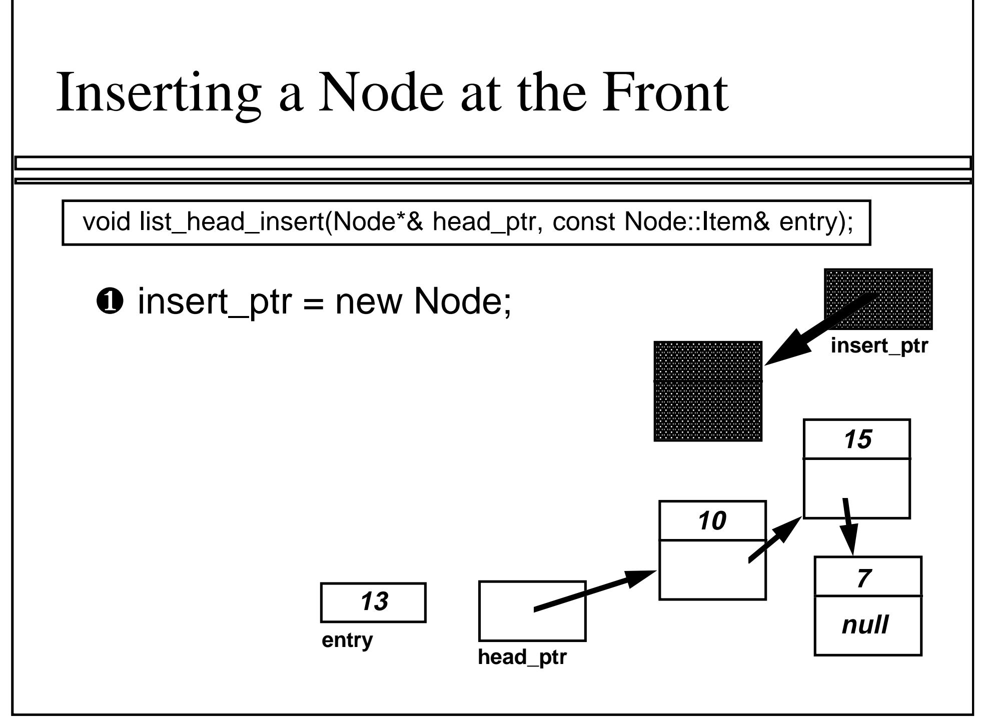
This is a const reference parameter because the function uses the new entry (for the data part of a new node), but the function does not change the value of the parameter. In a prototype of a function such as this, we must use the entire data type (Node::Item) and not just Item.

In order to develop the function, we'll trace through a small example. In the example the new entry (number 13) is being added to a list which already has three entries (10, 15 and 7).



The first step of the head_insert function is to create a new node. In order to create the new node, we need a local variable which I have called insert_ptr, which is a pointer to a node. Think of insert_ptr as being a "temporary insertion pointer" because we won't need it any more once the new node is actually placed in the list.

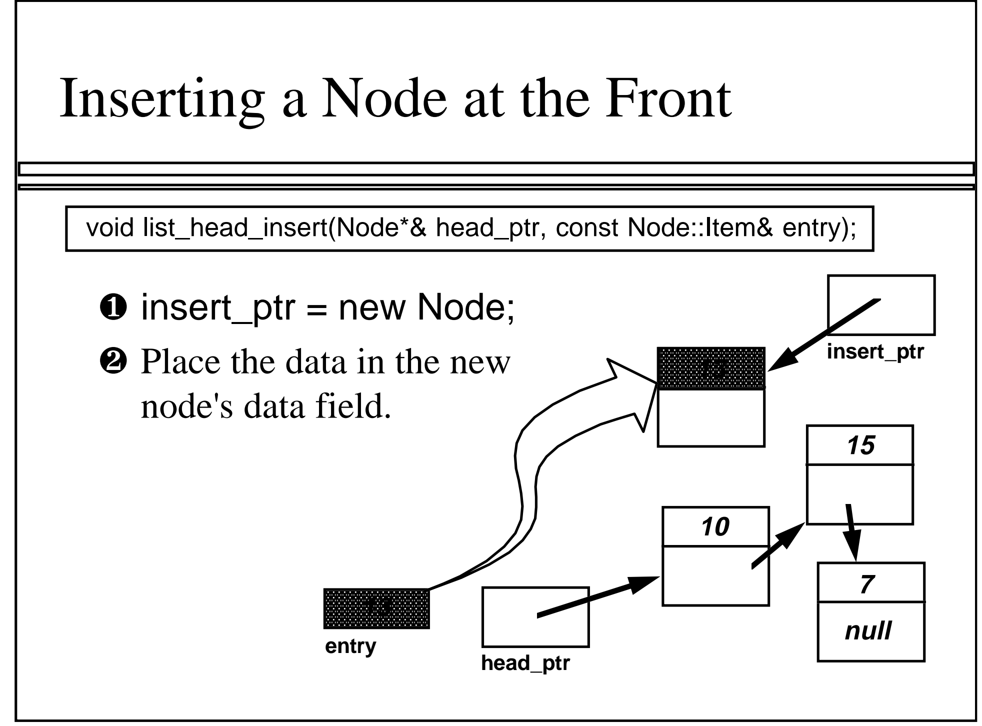
The idea is to get insert_ptr to point to the newly-created node when this node is first created. Do you remember enough from Chapter 4 to know the C++ statement which will create the new node and point insert_ptr at this new node?



Here is the C++ statement:

```
insert_ptr = new Node;
```

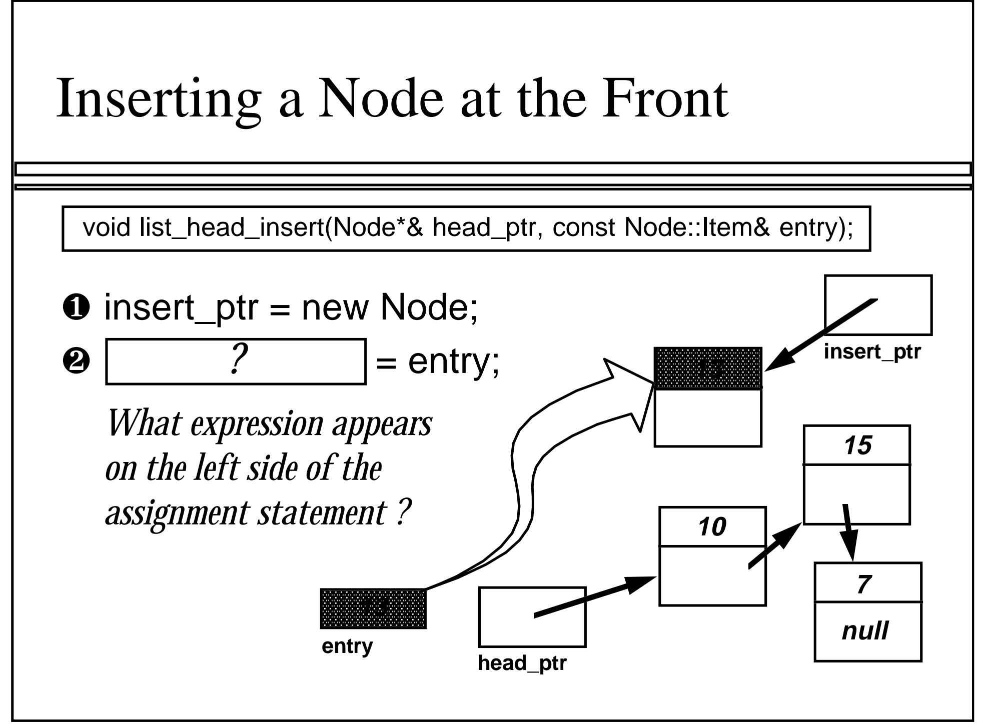
The statement creates a new node, and points the local variable insert_ptr at this newly-created node.



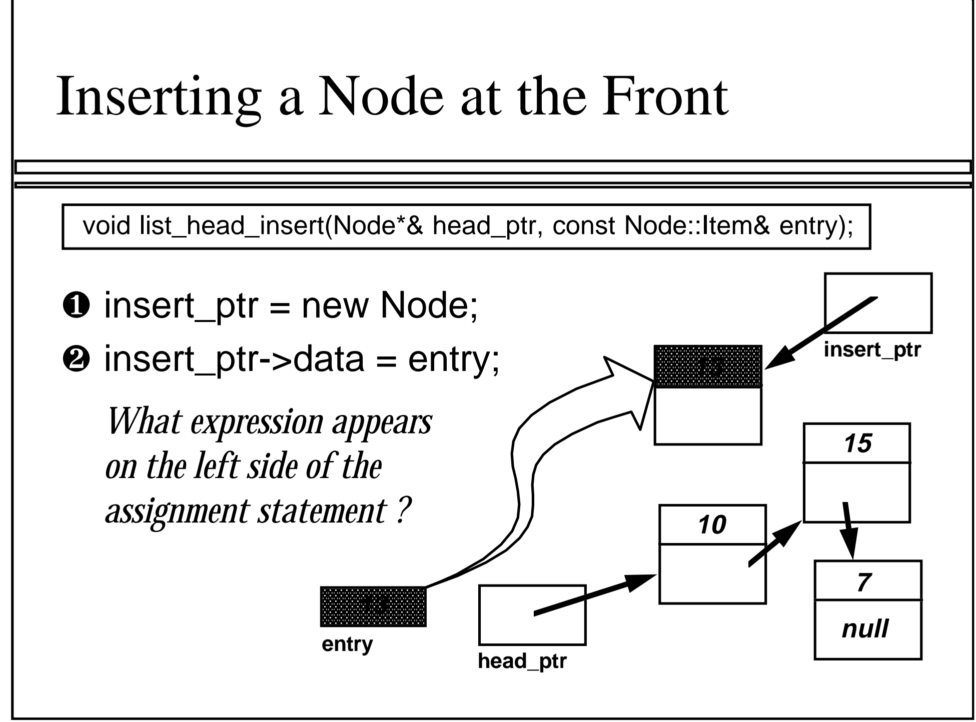
Once the new node has been created, we need to place values in the new node's member variables. The first member variable is the new node's data field, where we are supposed to place a copy of the entry.

Another question: Can you write the C++ statement which will copy the value of entry into the data field of the new node?

The answer is on the next slide...



...or at least most of the answer is on this slide! The assignment statement written here is copying entry into some location. What goes on the left side of the assignment statement? Your answer should refer to the data member variable of the newly-created node.



Here is the full answer. The expression on the left of the assignment statement is:

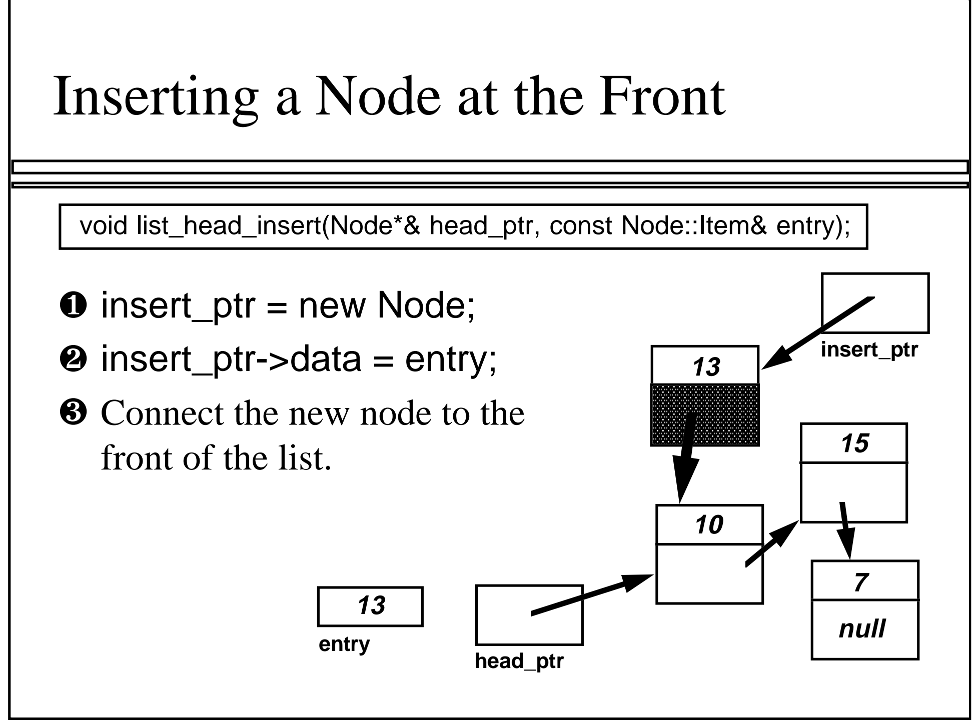
insert_ptr->data

You can read an expression like this from left-to-right. The expression means

1. Start at the pointer named insert_ptr;

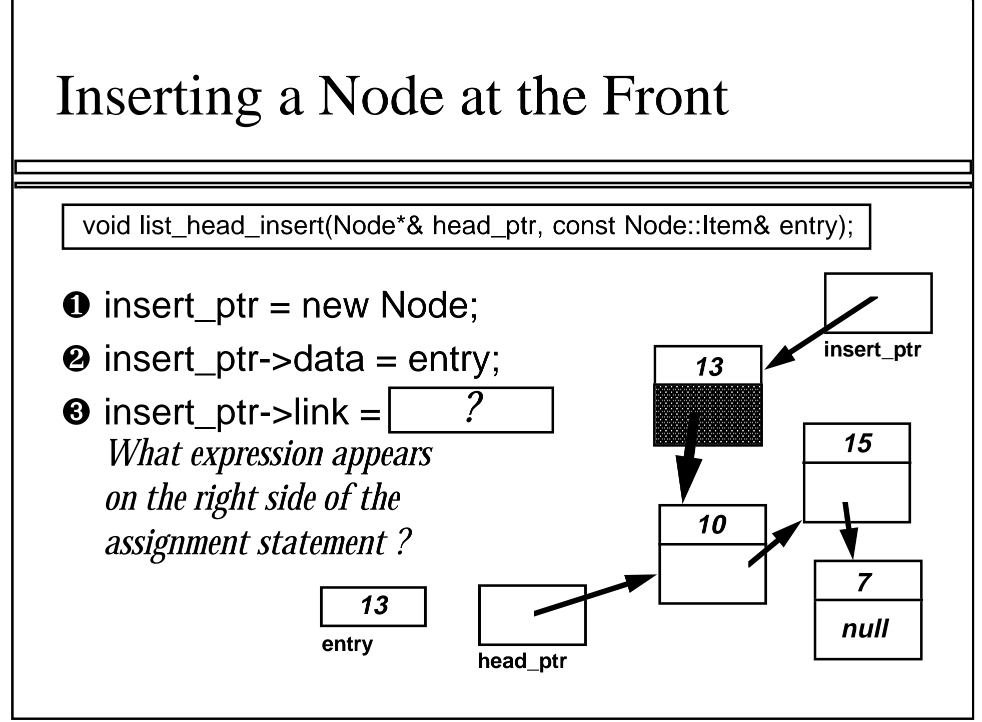
2. The -> is the “member selection operator,” used to select a member of the object that insert_ptr points to.

3. The particular member we selected is the member called data.



The third step of head_insert is to connect the new node to the rest of the existing list. In other words, we will place a pointer into the link member variable of the new node.

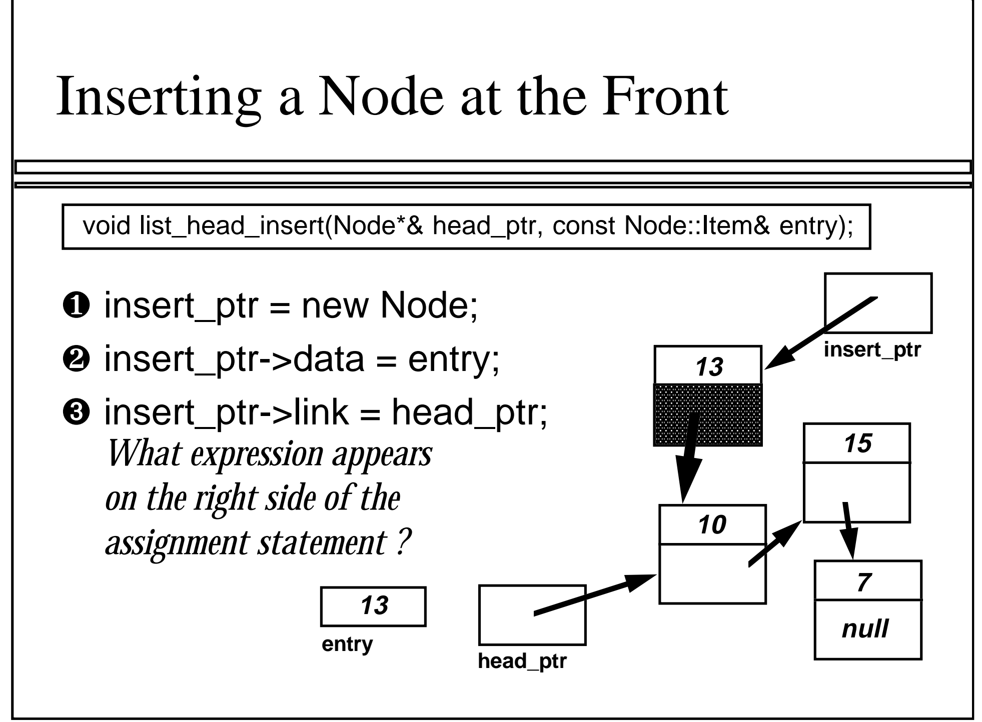
Another question: Can you write the assignment statement to place this pointer into the link member variable of the newly-created node?



Here is half of the assignment statement:

insert_ptr->link = ____________

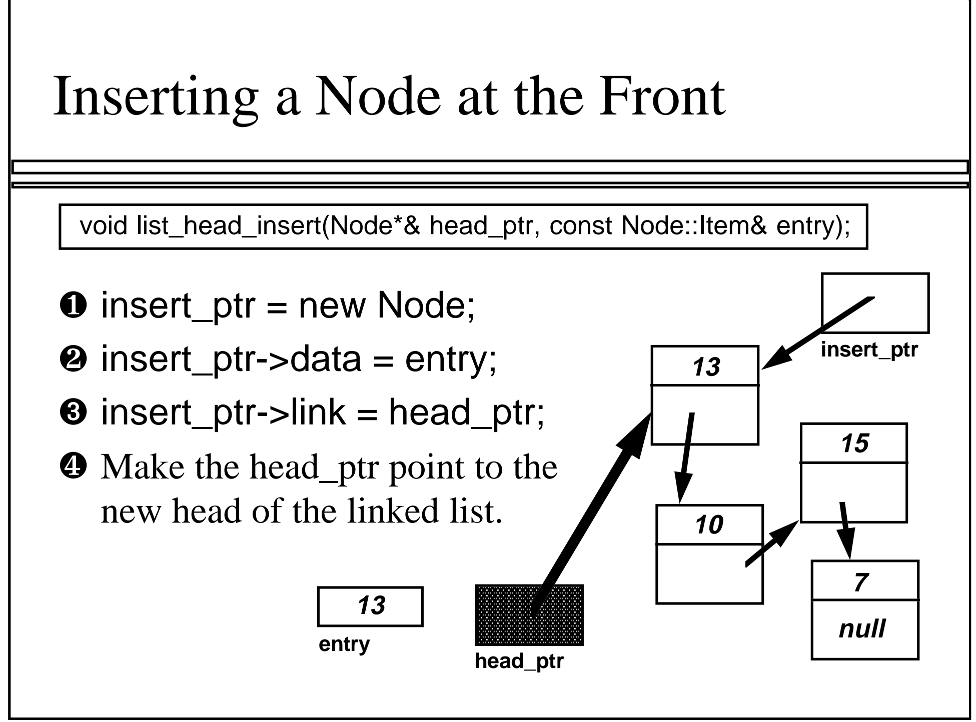
How do you fill in this blank? We want the pointer to be a pointer to the first node of the old linked list. And head_ptr is already a pointer to the first node of the old linked list. So...



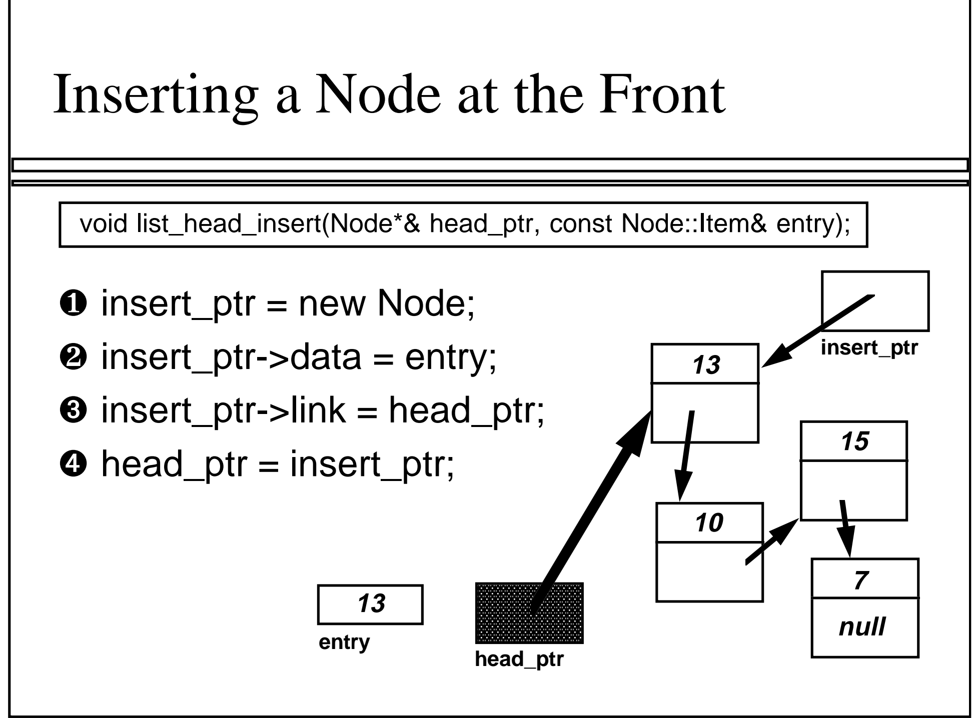
...here is the complete assignment statement:

insert_ptr->link = head_ptr;

In English you can say "Make the link member variable of the new node point to the same thing that the head pointer is pointing to."



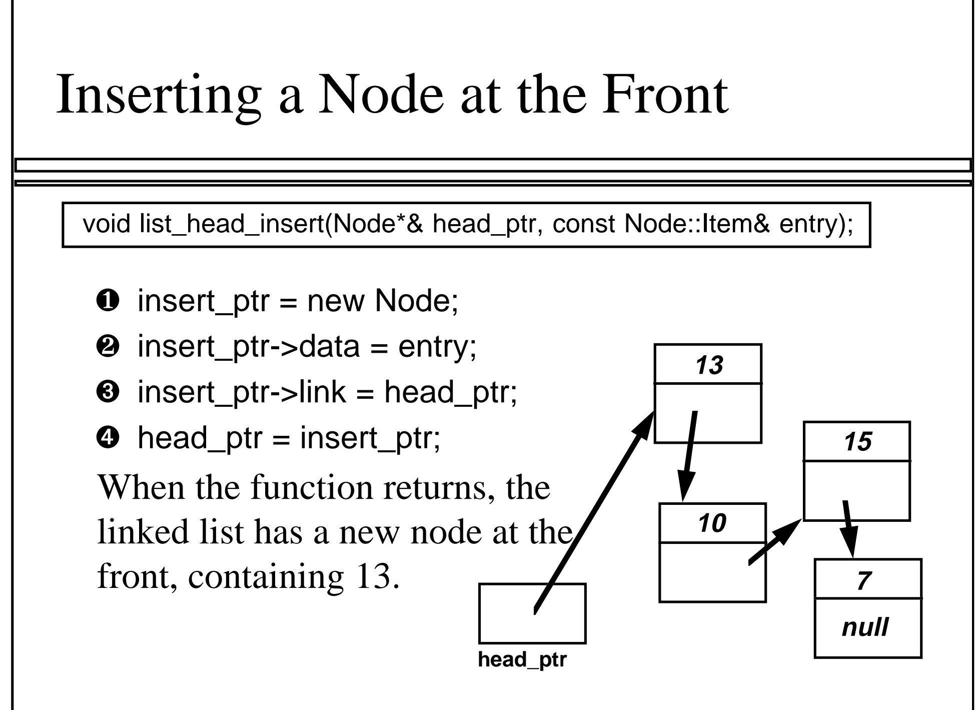
The head_insert function needs one last step: Making the head pointer point to the newly-created node. Again, this will be an assignment statement. Can you write the statement before I move to the next slide?



The assignment statement is:

head_ptr = insert_ptr;

In English this means: "Make the head pointer point to the same place that insert_ptr is pointing." Or in other words: "Make the head pointer point to the newly-created node."



When the head_insert function returns, the linked list will have a new node (containing 13).

Notice that when the function finishes, the local variable named insert_ptr no longer exists. But the newly created node does still exist, and the head pointer points to the newly created node.

One warning: You may have read about the dispose function in C++, which is used to destroy nodes. Never call dispose when you are adding a new node to a linked list! The only time that dispose is called is when you want to have fewer nodes, not when you want to have more nodes.

Inserting a Node at the Front

```
void list_head_insert(Node*& head_ptr, const Node::Item& entry)
{
    Node *insert_ptr;

    insert_ptr = new Node;
    insert_ptr->data = entry;
    insert_ptr->link = head_ptr;
    head_ptr = insert_ptr;
}
```

Here's the complete declaration of the head_insert function. Can you find these things:

-- the declaration of the local variable, insert_ptr

-- the place where the new node is actually created?

By the way, the word "created" is probably too strong a word. No new memory is actually created. What does happen is that a portion of memory is set aside for the new operator to use. This portion of memory is called the "heap". Whenever the new operator is called, part of the heap is provided for your new node.

Inserting a Node at the Front

```
void list_head_insert(Node*& head_ptr, const Node::Item& entry)
{
    Node *insert_ptr;

    insert_ptr = new Node;
    insert_ptr->data = entry;
    insert_ptr->link = head_ptr;
    head_ptr = insert_ptr;
}
```

Does the function work correctly for the empty list ?

Before we finish the example, we should check one last thing. Does the function work correctly if the list is initially empty? In other words, if the head pointer is null, will the function manage to correctly add the first node of the list?

We can check with an example...

Inserting a Node at the Front

```
void list_head_insert(Node*& head_ptr, const Node::Item& entry)
{
    Node *insert_ptr;

    insert_ptr = new Node;
    insert_ptr->data = entry;
    insert_ptr->link = head_ptr;
    head_ptr = insert_ptr;
}
```

Does the function work correctly for the empty list ?

13
entry

null
head_ptr

...as shown here. In this example, the head pointer is null, and the new entry is 13.

Inserting a Node at the Front

```
void list_head_insert(Node*& head_ptr, const Node::Item& entry)
{
    Node *insert_ptr;

    insert_ptr = new Node;
    insert_ptr->data = entry;
    insert_ptr->link = head_ptr;
    head_ptr = insert_ptr;
}
```

13
entry
null
head_ptr
insert_ptr

The first two statements of the function allocate a new node, and place the number 13 in the data field of the new node.

Inserting a Node at the Front

```
void list_head_insert(Node*& head_ptr, const Node::Item& entry)
{
    Node *insert_ptr;

    insert_ptr = new Node;
    insert_ptr->data = entry;
    insert_ptr->link = head_ptr;
    head_ptr = insert_ptr;
}
```


The third statement of the function copies the value null from the head pointer to the link field of the new node, as shown in this slide.

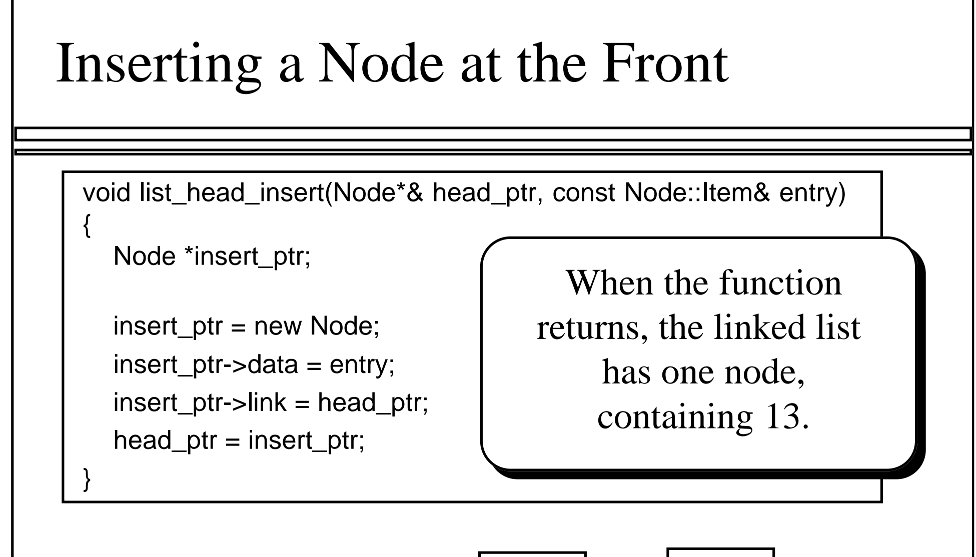
Inserting a Node at the Front

```
void list_head_insert(Node*& head_ptr, const Node::Item& entry)
{
    Node *insert_ptr;

    insert_ptr = new Node;
    insert_ptr->data = entry;
    insert_ptr->link = head_ptr;
    head_ptr = insert_ptr;
}
```

13
entry
head_ptr
13
null
insert_ptr

The function's fourth statement makes the head pointer point to the new node.



So, as you can see, when the function finally returns, there is one node in the linked list, and the head pointer correctly points to the new node. In other words, our function works just fine, even if we are adding the first node to a previously empty list.

Caution!

- Always make sure that your linked list functions work correctly with an empty list.

This slide is just a warning: Always make sure that your linked list functions work sensibly with the empty list. If you run into a function that fails for the empty list, then you will need to modify the function by adding some special code to deal with a null head pointer.

Pseudocode for Inserting Nodes

- Nodes are often inserted at places other than the front of a linked list.
- There is a general pseudocode that you can follow for any insertion function. . .

That's the end of our head_insert function. There are two more linked list functions that we'll look at today. Actually, we'll just look at typical pseudocode for the next two functions. The first of these functions is a function for inserting a new node at a place other than the front of a linked list.

Actually, the pattern that we'll follow will be capable of inserting a new node at any location in a linked list: Maybe at the front, maybe in the middle, maybe at the end. For example, you might be keeping the nodes sorted from smallest integer to largest integer. Or perhaps there is some other method to your insertions. The pseudocode that I'll describe will work for any method that you might think of. Really!

Pseudocode for Inserting Nodes

❶ Determine whether the new node will be the first node in the linked list. If so, then there is only one step:

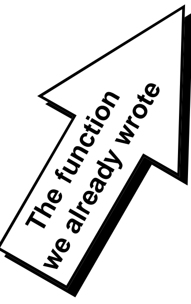
```
head_insert(head_ptr, entry);
```

The first step of the pseudocode is to determine whether the new node will be inserted as the first node of the linked list. If so, then you can simply call the head_insert function which we already wrote, as shown here.

Pseudocode for Inserting Nodes

❶ Determine whether the new node will be the first node in the linked list. If so, then there is only one step:

```
head_insert(head_ptr, entry);
```

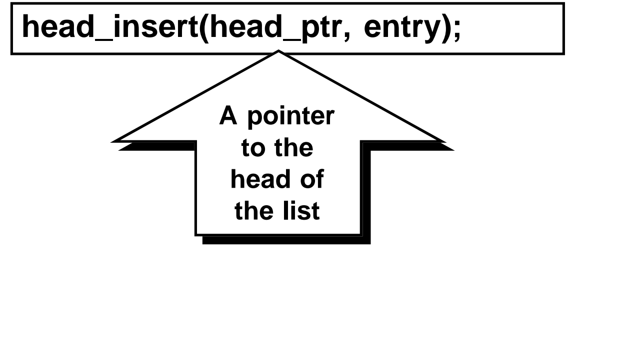


There are three parts to calling the function we already wrote. First, of course, is the function name: head_insert.

Pseudocode for Inserting Nodes

❶ Determine whether the new node will be the first node in the linked list. If so, then there is only one step:

```
head_insert(head_ptr, entry);
```



Next is the first argument, which is the pointer to the head of the linked list.

Pseudocode for Inserting Nodes

❶ Determine whether the new node will be the first node in the linked list. If so, then there is only one step:

```
head_insert(head_ptr, entry);
```

The data to put in the new node

And finally is the data for the new node.

So, this function call will add the new data at the front of an existing linked list.

Pseudocode for Inserting Nodes

❷ Otherwise (if the new node will not be first):

- ❒ Start by setting a pointer named **previous_ptr** to point to the node which is just **before** the new node's position.

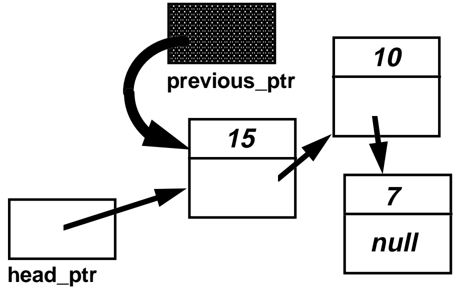
On the other hand, if the new data does not belong at the front of the linked list, then you will start by setting up a pointer that I call previous_ptr. This pointer must be set up to point to the node which is just before the new node's position.

Pseudocode for Inserting Nodes

❷ Otherwise (if the new node will not be first):

- ❐ Start by setting a pointer named **previous_ptr** to point to the node which is just **before** the new node's position.

In this example, the new node will be the second node



As an example, suppose that we want to add 13 to this list, and we want to keep all the entries in order from largest to smallest. We don't want to put 13 first on this list, because 13 is smaller than 15. So, we proceed to set up the previous_ptr to point to the node which is just before the position where 13 should be inserted.

In this example, previous_ptr would be set up to point to the first node of the linked list (containing 15). The 13 will be inserted as the new second node of the list (after the 15, but before the 10).

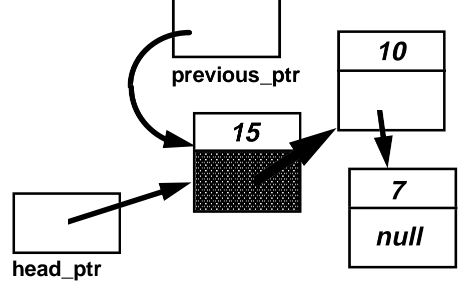
Pseudocode for Inserting Nodes

❷ Otherwise (if the new node will not be first):

❒ Start by setting a pointer named previous_ptr to point to the node which is just before the new node's position

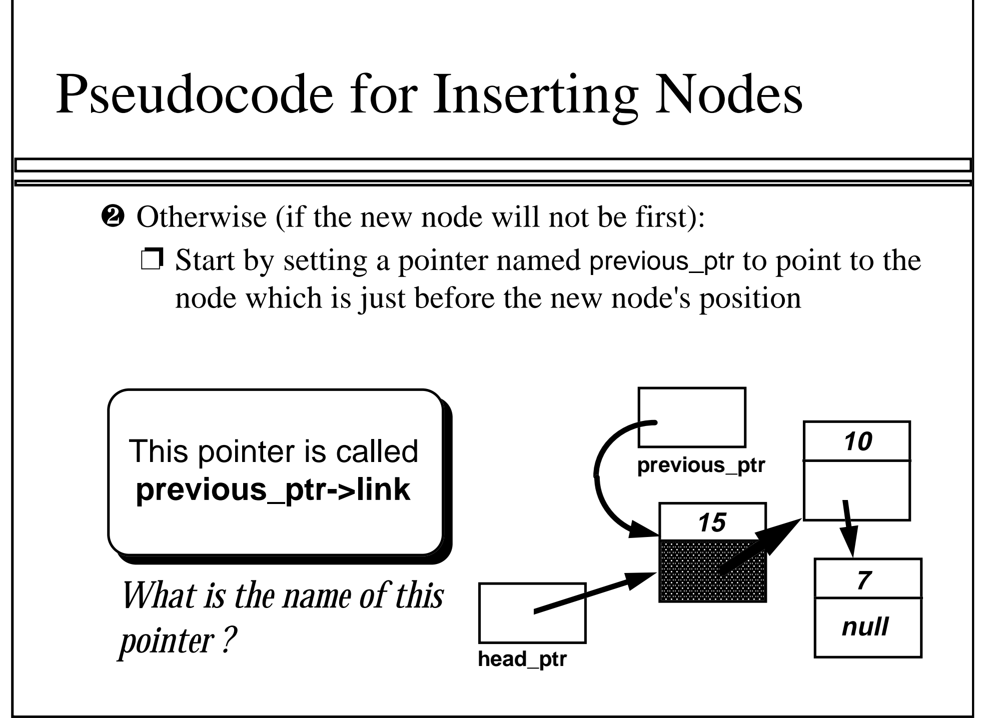
Look at the pointer which is **in the node** previous_ptr^

What is the name of this pointer ?



In order to insert the new node at the proper place, we need to examine the link field which is in the node pointed to by previous_ptr. This link field is highlighted in the picture.

Question: What is the name of this link field? Can you write the answer before I move to the next slide?



As you see, the highlighted pointer is called previous_ptr->link.

In other words: Start at previous_ptr, follow the pointer, and select the link field from the record.

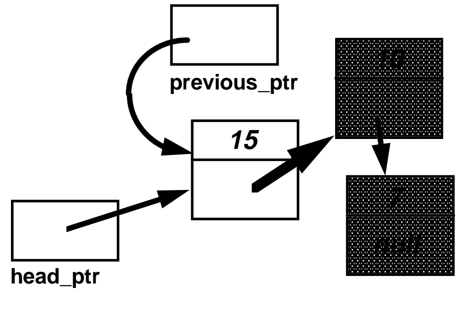
There is a reason that previous_ptr->link is important to us...

Pseudocode for Inserting Nodes

❷ Otherwise (if the new node will not be first):

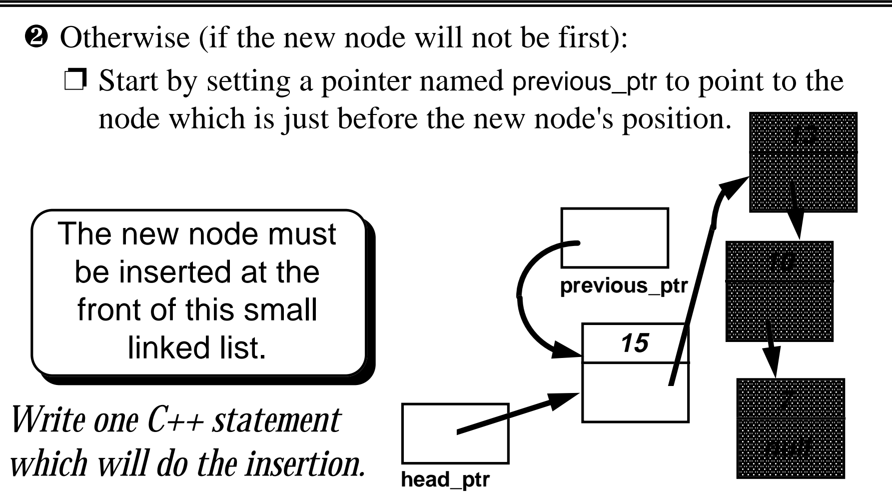
- ❐ Start by setting a pointer named previous_ptr to point to the node which is just before the new node's position

previous_ptr->link points to the head of a small linked list, with 10 and 7



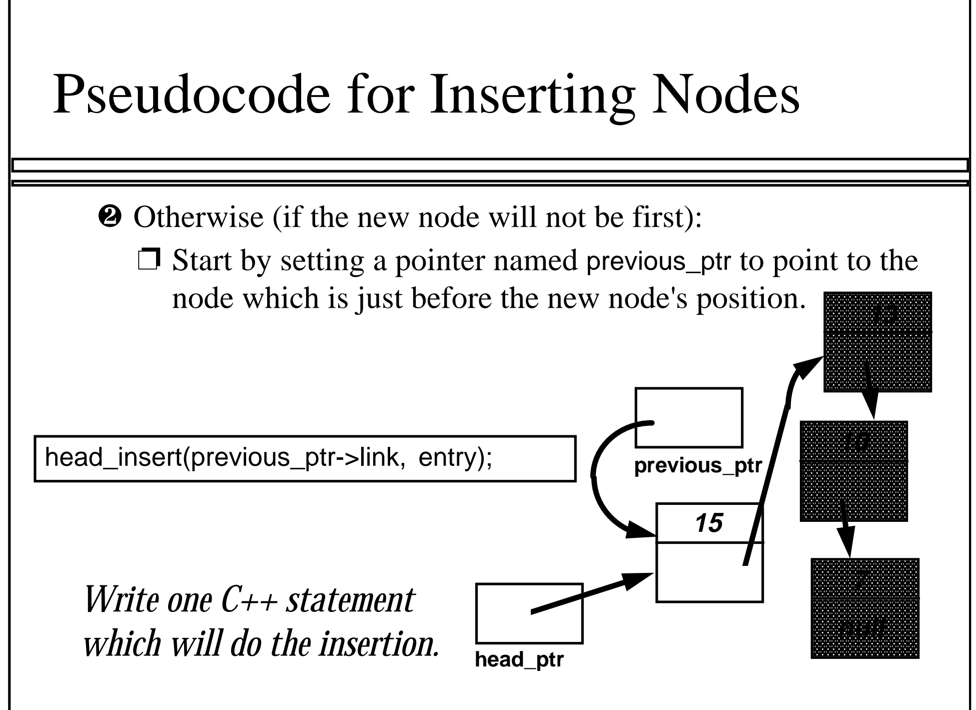
...as you can see here, previous_ptr->link actually points to the first node of a small linked list. The small linked list contains 10 and 7 -- and more important, we want to add the new entry at the front of this small linked list.

Pseudocode for Inserting Nodes



Here is a small challenge: Can you write just one C++ statement which will insert the new node at the front of the small linked list? Use the name entry for the name of the data that you are placing in the new node...

...think about what functions you already have...



Did anyone come up with this solution? By calling the head_insert function, we can insert entry at the front of the small linked list. The key is that we have the pointer

```
previous_ptr->link
```

which points to the head of the small linked list.

Pseudocode for Inserting Nodes

❶ Determine whether the new node will be the first node in the linked list. If so, then there is only one step:

```
head_insert(head_ptr, entry);
```

❷ Otherwise (if the new node will not be first):

- ❐ Set a pointer named previous_ptr to point to the node which is just before the new node's position.
- ❐ Make the function call:

```
head_insert(previous_ptr->link, entry);
```

Here is the complete pseudocode for adding a new node. Remember: this pseudocode can be used for many different methods of inserting, such as keeping nodes in order from largest to smallest.

Pseudocode for Inserting Nodes

- The process of adding a new node in the middle of a list can also be incorporated as a separate function. This function is called list_insert in the linked list toolkit of Section 5.2.

Section 5.2 gives a collection of function for manipulating linked lists. One of these functions, called list_insert, places a new node in the middle of a linked list.

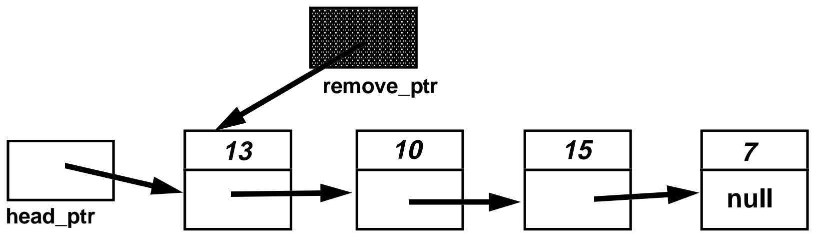
Pseudocode for Removing Nodes

- ❒ Nodes often need to be removed from a linked list.
- ❒ As with insertion, there is a technique for removing a node from the front of a list, and a technique for removing a node from elsewhere.
- ❒ We'll look at the pseudocode for removing a node from the front of a linked list.

Do you have energy for one more function? Or at least for some pseudocode. I'll just show you how to remove a node from the front of a linked list. You can figure out the code for removing other nodes, or you can read the solution in Section 5.2.

Removing the Head Node

❶ Start by setting up a temporary pointer named **remove_ptr** to the head node.



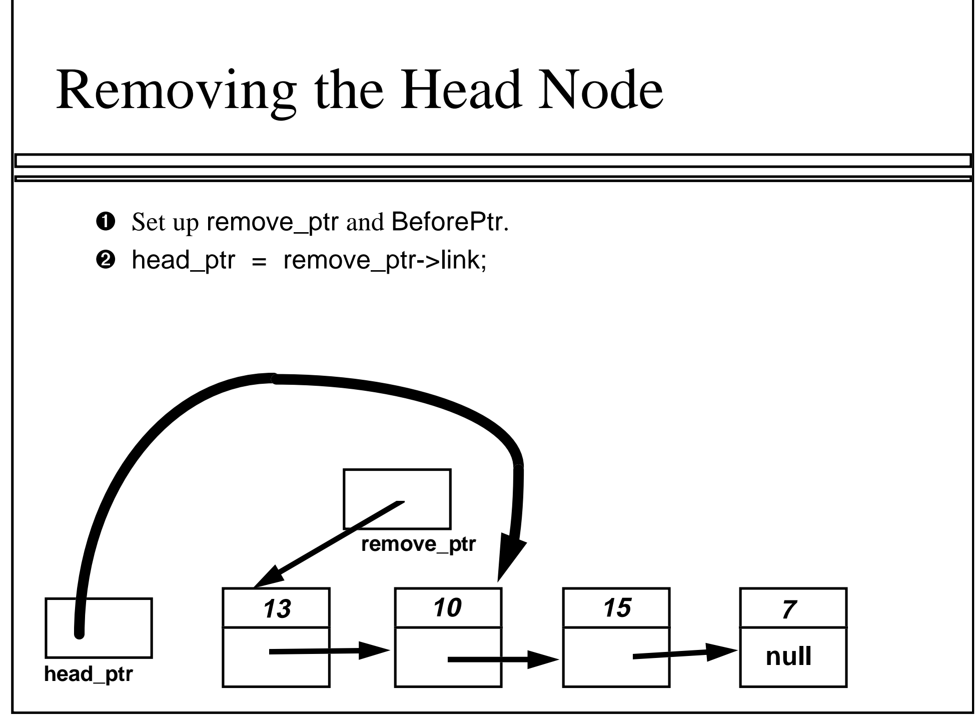
As we did before, we'll use an example to illustrate the pseudocode. In the example, we want to delete the head node which contains 13. The first part of the pseudocode sets up a local variable named remove_ptr to point to the node that we want to remove.

Removing the Head Node

❶ Set up remove_ptr.
❷ head_ptr = remove_ptr->link;

Draw the change that this statement will make to the linked list.

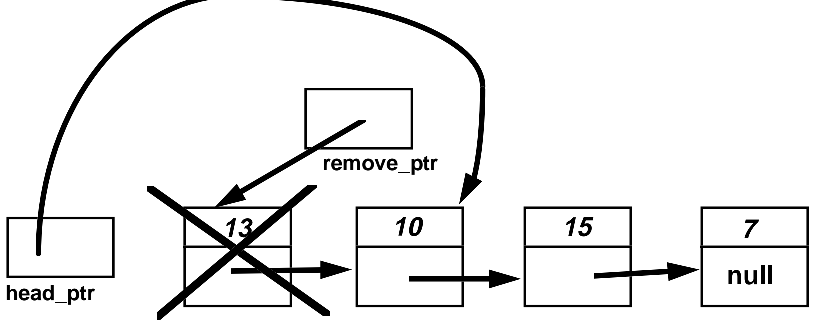
Since we are removing the first node of the list, we need to change the place where head_ptr is pointing. Can you tell me where the head_ptr will point after the assignment statement that is shown here? Be quick, because I am about to change the slide...



As you can see, the assignment statement makes the head pointer point to the second node of the list.

Removing the Head Node

❶ Set up remove_ptr and BeforePtr.
❷ head_ptr = remove_ptr->link;
❸ delete remove_ptr; // Return the node's memory to heap.

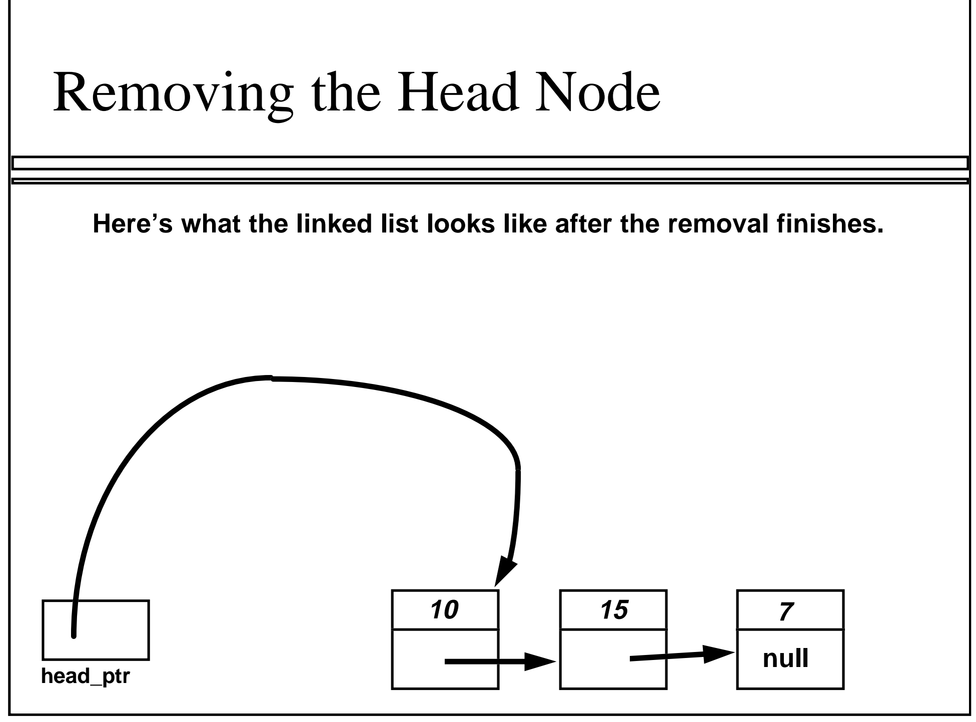


There is one last statement that the removal function should execute:

```
delete remove_ptr;
```

This takes the node which is pointed to by remove_ptr and returns it to the heap, so that the memory can be reused at some later date. If you forget this step, then the memory won't be able to be reused -- a situation that's called a "heap leak".

Remember this rule of thumb: If you are adding a node to a linked list, then make sure that new ... is called exactly once. If you are removing a node from a linked list, then make sure that delete ... is called exactly once.



Notice that the local variable, remove_ptr, is no longer around.

Summary

- It is easy to insert a node at the front of a list.
- The linked list toolkit also provides a function for inserting a new node elsewhere
- It is easy to remove a node at the front of a list.
- The linked list toolkit also provides a function for removing a node elsewhere--you should read about this function and the other functions of the toolkit.

A quick summary . . .

Presentation copyright 1997, Addison Wesley Longman,
For use with *Data Structures and Other Objects Using C++*
by Michael Main and Walter Savitch.

Some artwork in the presentation is used with permission from Presentation Task Force (copyright New Vision Technologies Inc) and Corel Gallery Clipart Catalog (copyright Corel Corporation, 3G Graphics Inc, Archive Arts, Cartesia Software, Image Club Graphics Inc, One Mile Up Inc, TechPool Studios, Totem Graphics Inc).

Students and instructors who use *Data Structures and Other Objects Using C++* are welcome to use this presentation however they see fit, so long as this copyright notice remains intact.

THE END

Feel free to send your ideas to:

Michael Main
main@colorado.edu