Chapter 5 introduces the often-used data structure of linked lists. This presentation shows how to implement the most common operations on linked lists.
For this presentation, each node in the linked list is a struct, as shown here.

```c
struct Node {
    typedef int Item;
    Item data;
    Node *link;
};
```
Declarations for Linked Lists

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

```c
typedef int Item;
struct Node {
    Item data;
    Node *link;
};
```
Each Node also contains a link field which is a pointer to another Node.

```c
#define int Item;
Item data;
Node *link;
```
Declarations for Linked Lists

- A program can keep track of the front node by using a pointer variable such as `head_ptr` in this example.
- Notice that `head_ptr` is not a Node -- it is a pointer to a Node.
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.
Inserting a Node at the Front

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

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

1. Create a new node, pointed to by a local variable `insert_ptr`. 
Inserting a Node at the Front

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

1. insert_ptr = new Node;
```
Inserting a Node at the Front

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

1. `insert_ptr = new Node;`
2. Place the data in the new node's data field.
Inserting a Node at the Front

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

1. `insert_ptr = new Node;`
2. `? = entry;`

*What expression appears on the left side of the assignment statement?*
Inserting a Node at the Front

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

1. `insert_ptr = new Node;`
2. `insert_ptr->data = entry;`

*What expression appears on the left side of the assignment statement?*
Inserting a Node at the Front

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

1. `insert_ptr = new Node;`
2. `insert_ptr->data = entry;`
3. Connect the new node to the front of the list.
Inserting a Node at the Front

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

1. `insert_ptr` = new Node;
2. `insert_ptr->data` = entry;
3. `insert_ptr->link` = ?

What expression appears on the right side of the assignment statement?
Inserting a Node at the Front

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

1. `insert_ptr = new Node;`
2. `insert_ptr->data = entry;`
3. `insert_ptr->link = head_ptr;`

*What expression appears on the right side of the assignment statement?*
Inserting a Node at the Front

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

1. `insert_ptr = new Node;`
2. `insert_ptr->data = entry;`
3. `insert_ptr->link = head_ptr;`
4. Make the `head_ptr` point to the new head of the linked list.
Inserting a Node at the Front

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

1. `insert_ptr = new Node;`
2. `insert_ptr->data = entry;`
3. `insert_ptr->link = head_ptr;`
4. `head_ptr = insert_ptr;`
Inserting a Node at the Front

When the function returns, the linked list has a new node at the front, containing 13.

1 insert_ptr = new Node;
2 insert_ptr->data = entry;
3 insert_ptr->link = head_ptr;
4 head_ptr = insert_ptr;
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;
}

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?
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?
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;
}

Inserting a Node at the Front

entry

head_ptr

insert_ptr

13

null
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;
}

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;
}
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;
}

When the function returns, the linked list has one node, containing 13.
Caution!

- Always make sure that your linked list functions work correctly with an empty list.
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. . .
Pseudocode for Inserting Nodes

1. 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);
```
Pseudocode for Inserting Nodes

1. 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);
   ```
Pseudocode for Inserting Nodes

1. 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);
Pseudocode for Inserting Nodes

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

```c
head_insert(head_ptr, entry);
```
Pseudocode for Inserting Nodes

2 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.
Pseudocode for Inserting Nodes

2 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.
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.
Pseudocode for Inserting Nodes

2. 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.

This pointer is called previous_ptr->link

What is the name of this pointer?
\[\text{Pseudocode for Inserting Nodes}\]

1. Otherwise (if the new node will not be first):
   - Start by setting a pointer named \(\text{previous\_ptr}\) to point to the node which is just before the new node's position.

2. Insert the new node into the list by adjusting the links of the nodes around it.

\[\text{previous\_ptr->link = new\_node}\]

\[\text{new\_node->link = next\_node}\]

\[\text{previous\_ptr->next = new\_node}\]
Pseudocode for Inserting Nodes

2. 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.

The new node must be inserted at the front of this small linked list.

Write one C++ statement which will do the insertion.
Pseudocode for Inserting Nodes

2 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.

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

Write one C++ statement which will do the insertion.
Pseudocode for Inserting Nodes

1. Determine whether the new node will be the first node in the linked list. If so, then there is only one step:
   \[
   \text{head_insert(head_ptr, entry)};
   \]

2. 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:
     \[
     \text{head_insert(previous_ptr->link, entry)};
     \]
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.
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.
Removing the Head Node

1. Start by setting up a temporary pointer named `remove_ptr` to the head node.
Removing the Head Node

1. Set up remove_ptr.
2. head_ptr = remove_ptr->link;

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

1. Set up remove_ptr and BeforePtr.
2. head_ptr = remove_ptr->link;
Removing the Head Node

1. Set up remove_ptr and BeforePtr.
2. head_ptr = remove_ptr->link;
3. delete remove_ptr; // Return the node's memory to heap.
Removing the Head Node

Here’s what the linked list looks like after the removal finishes.
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.
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

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.