Creating Data Structures in C++

CS 16: Solving Problems with Computers I
Lecture #16

Ziad Matni
Dept. of Computer Science, UCSB

Lecture Outline

Creating Data Structures in C++

- Finishing up Structures
  - **Chapter 10.1**
- Dynamic Arrays
  - Better than arrays … foundation for vectors
  - **Chapter 9.1 and 9.2**
- Linked Lists
  - Using pointers and structures together
  - **Chapter 13.1**

Administrative

- 3 MORE CLASSES TO GO! ☺

M	T	W	Th	F
5/29	5/30 LECTURE 15 HW13 due	5/31 Lab8 issued	6/1 LECTURE 16 HW14 due Lab7 due	6/2
6/5	6/6 LECTURE 17 HW15 due	6/7	6/8 REVIEW HW16 due Lab8 due	6/9 *Last day of the quarter*

Administrative

- New homework #15 issued: Due Tuesday 6/6
- You have one other homework to come!
  - Homework #16 due on Thursday 6/8
- Lab #7 due today
- Lab #8 issued: Due Thursday 6/8

IMPORTANT NOTE!

NO assignment (hwk, lab) will be accepted
to be turned in **AFTER** the **LAST** lecture/class on
THURSDAY 6/8!

("late" assignments policy will not apply –
we simply will not accept them)

Structures

Read Ch. ***10.1*** *in textbook*

Structures in C++

Review

- Example:

```
struct CDAccount
{
      double balance;
      double interest_rate;
      int term;
} ;
```

Remember this semicolon!

- Keyword **struct** begins a structure definition
- In this example,
  CDAccount is the structure ***tag*** – this is the structure's **type**
- Member names are ***identifiers*** declared in the braces

Structures in C++

Review

- Example:

```
struct CDAccount
{
      double balance;
      double interest_rate;
      int term;
} ;
```

- Accessing the member variables is done using the "**dot operator**"

Example:

```
CDAccount MyCDA
MyCDA.balance = 500.00
MyCDA.interest_rate = rateX * 1.5
MyCDA.term = 12
```

Duplicate Names

- Member variable names duplicated between structure types are not a problem

```
struct FertilizerStock
{
    double quantity;
    double nitrogen_content;
};

FertilizerStock  super_grow;
```

```
struct CropYield
{
   int quantity;
   double size;
};

CropYield  apples;
```

- **super_grow.quantity** and **apples.quantity** are different variables stored in different locations

Structures as Arguments

- Recall that:
  Structures can be arguments in function calls
  - The formal parameter can be either **call-by-value** or **call-by-reference**

- Example:
  ```
  void get_data(CDAccount& the_account);
  ```
  - Uses the structure type CDAccount we saw earlier as the type for a call-by-reference parameter

Structures as Return Types

- Structures **can also** be the type of a value ***returned*** by a function

Example:

```
CDAccount shrink_wrap(double the_balance,
                      double the_rate,
                      int the_term)
{
    CDAccount temp;
    temp.balance = the_balance;
    temp.interest_rate = the_rate;
    temp.term = the_term;
    return temp;
}
```

What is this function doing?

Example: Using Function **shrink_wrap**

- **shrink_wrap** builds a complete structure value in **temp**, which is returned by the function
- We can use **shrink_wrap** to give a variable of type ***CDAccount*** a value in this way:

Example:

```
CDAccount  new_account;
new_account = shrink_wrap(1000.00, 5.1, 11);
```

Assignment and Structures

- The assignment operator can be used to assign values to structure types

- Using the CDAccount structure again for example:

```
CDAccount my_account, your_account;
my_account.balance = 1000.00;
my_account.interest_rate = 5.1;
my_account.term = 12;
your_account = my_account;
```

- Note: This last line assigns *all member variables* in **your_account** the corresponding values in **my_account**

Hierarchical Structures

- Structures **can** contain member variables that are **also structures**

```
struct Date
{
    int month;
    int day;
    int year;
};
```

```
struct PersonInfo
{
    double height;
    int weight;
    Date birthday;
};
```

- struct **PersonInfo** contains a **Date** structure

Using **PersonInfo**

*An example on "." **operator** use*

```
struct PersonInfo
{
    double height;
    int weight;
    Date birthday;
};
```

```
struct Date
{
   int month;
   int day;
   int year;
};
```

- A variable of type **PersonInfo** is declared:

```
PersonInfo person1;
```

- To display the birth year of **person1**, first access the birthday member of person1

```
cout <<  person1.birthday…(not complete yet!)
```

- But we want the ***year***, so we now specify the year member of the birthday member

```
cout << person1.birthday.year;
```

Initializing Structures

- A structure can be initialized when declared

Example:

```
struct Date
{
    int month;
    int day;
    int year;
};
```

- Can be initialized in this way – watch for the order!:

```
Date  due_date = {12, 31, 2004};
```

Classes

- A **class** is a data type whose variables are **objects**

- The definition of a class includes
  - Description of the kinds of values of the member variables
  - Description of the member functions

- A class description is
  very much like a structure definition!

Main Differences: **structure** vs **class**

- *Classes* in C++ evolved from the concept of *structures* in C
- Both *classes* and *structures* can have member variables
- Both *classes* and *structures* can have member functions, ALTHOUGH classes are made to be easier to use with member functions
- Classes may not be used when interfacing with C, because C does not have a concept of classes (only structures)

Example of a Class: **DayOfYear** Definition

```
class DayOfYear
    {
      public:
        void output( );
        int month;
        int day;
    };
```

Member Function **Declaration**

Member Variables **Declaration**

public vs private settings for members

***public** means these members can be accessed by a program*

***private** means they are only for use by the class itself (e.g. test code)*

Dynamic Arrays

*Read Ch. **9** (Pointers) in textbook*

Dynamic Arrays

A dynamic array is an array whose size is determined when the program is running, not when you write the program

Is a vector a dynamic array?

Pointer Variables and Array Variables

- Array variables are actually pointer variables that point to the first indexed variable
  - Remember when calling an array in a function?
    - funcA(a) … not … funcA(a[ ])
  - Take, for instance:

```
int  a[10];
typedef int* IntPtr;
IntPtr p;
```

    - <u>NOTE</u>: Variables **a** and **p** are **<u>the same kind of variable!</u>**

- Since **a** is a pointer variable that points to **a[0]**, then issuing: `p = a;` causes **p** to point to the same location as **a**

Pointer Variables As Array Variables

```
int  a[10];
typedef int* IntPtr;
IntPtr p = a;
```

- Continuing with the previous example: Pointer variable **p** can be used as if it were an array variable

- So, p[0], p[1], …p[9] are all legal ways to use p

- *Is there a difference between an array and a pointer?* Variable **a** can be used as a pointer variable BUT the pointer value in **a** cannot be changed
  - So, the following is **not** legal:

```
IntPtr p2;    // p2 is assigned a value
a = p2        // attempt to change a
```

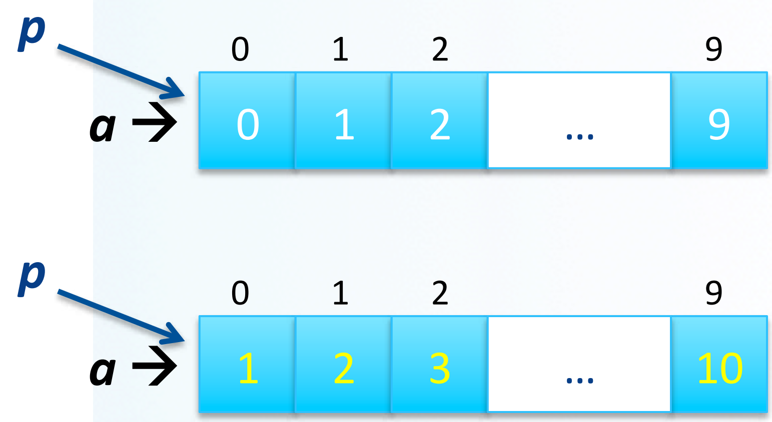
Arrays and Pointer Variables

```
//Program to demonstrate that an array variable is a kind of pointer variable.
#include <iostream>
using namespace std;

typedef int* IntPtr;

int main()
{
    IntPtr p;
    int a[10];
    int index;

    for (index = 0; index < 10; index++)
        a[index] = index;
```

Arrays and Pointer Variables

```
//Program to demonstrate that an array variable is a kind of pointer variable.
#include <iostream>
using namespace std;

typedef int* IntPtr;

int main( )
{
    IntPtr p;
    int a[10];
    int index;

    for (index = 0; index < 10; index++)
        a[index] = index;

    p = a;

    for (index = 0; index < 10; index++)
        cout << p[index] << " ";
    cout << endl;

    for (index = 0; index < 10; index++)
        p[index] = p[index] + 1;

    for (index = 0; index < 10; index++)
        cout << a[index] << " ";
    cout << endl;

    return 0;
}
```

Note that changes to the array p are also changes to the array a.



Output

```
0 1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9 10
```

Creating Dynamic Arrays

- Normal arrays require that the programmer determine the size of the array ***when the program is written***
  - *What if the programmer estimates too large?*
    - Memory is wasted
  - *What if the programmer estimates too small?*
    - The program may not work in some situations
- Dynamic arrays can be created with just the right size ***while the program is running***

Are Dynamic Arrays *aka* Vectors?!

- Not exactly the same…
  - ***vector*** is one *implementation* of dynamic arrays
  - "dynamic arrays" is a bigger (more encompassing) term

- The biggest difference is:
  - Vectors ***automatically*** increase their capacity
  - Dynamic arrays have to be told to do this using **new** and **delete**

- The advantage of vectors is that they are well-defined and you don't have to worry about size changes, capacity adjustments in memory, etc…

Creating Dynamic Arrays

- Dynamic arrays are created using the **new** operator

- Example:
  To create an array of 10 elements of type double:

```
typedef double* DoublePtr;
DoublePtr d;
d = new double[10];
```

d can now be used as if it were an ordinary array!

Dynamic Arrays (cont.)

- Pointer variable d is a pointer to d[0]
- When finished with the array, it should be **delete**d to return memory to the **freestore (heap)**
  - Example: **delete [ ] d;**
  - The brackets tell C++ that a dynamic array is being deleted so it must check the size to know how many indexed variables to remove
  - Do not forget the brackets!
- Display 9.6 in the book has an example of use

Multidimensional Dynamic Arrays

- Example: **Create a 3x4 multidimensional dynamic array**

- Recall: multidimensional arrays are arrays of arrays…
  - So a 3x4 array = 3-element array, each of which is a 4-element array

- First create a one-dimensional dynamic array
  - Start with a new definition:

```
typedef int* IntArrayPtr;
```

  - Now create a dynamic array of pointers named **m**:

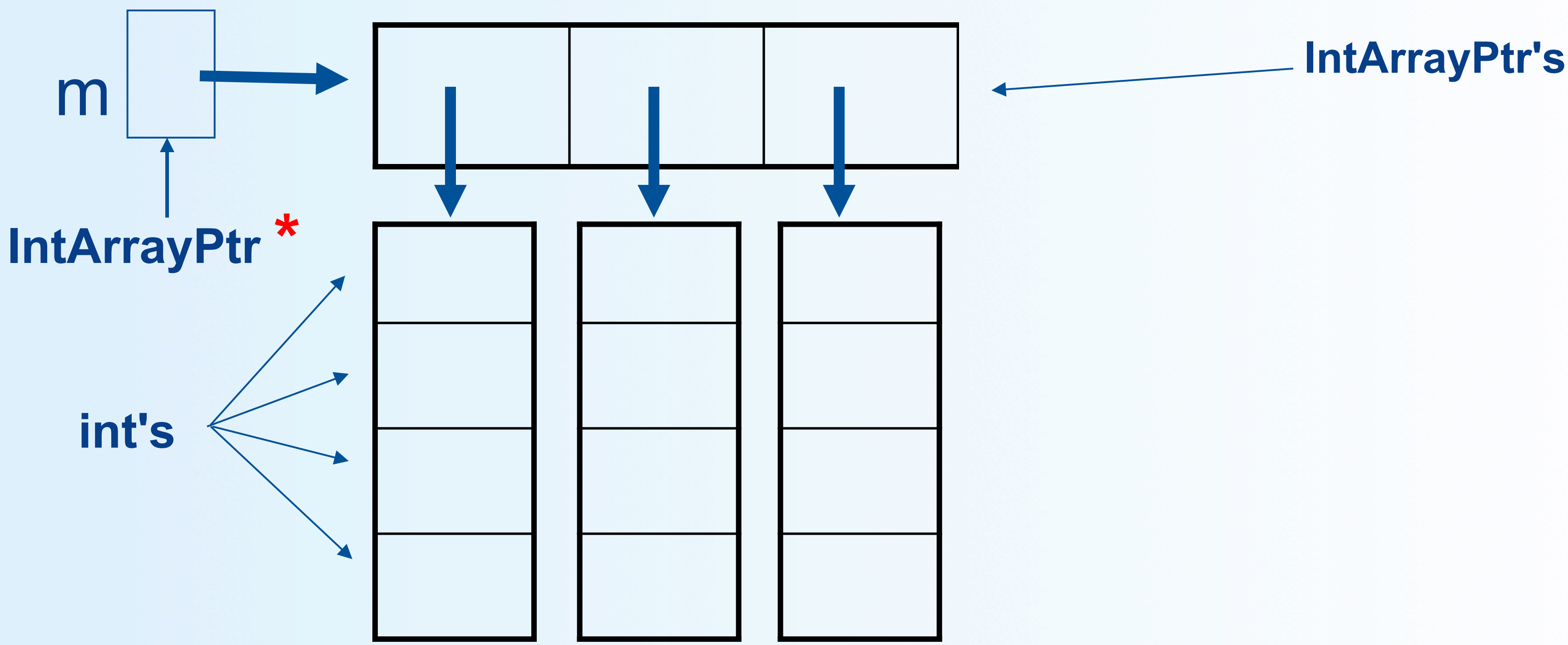
```
IntArrayPtr m = new IntArrayPtr[3];
```

- For each pointer in **m**, create a dynamic array of integers

```
for (int i = 0; i < 3; i++)
        m[i] = new int[4];
```

A Multidimensional Dynamic Array

- The dynamic array created on the previous slide could be visualized like this:



Deleting Multidimensional Arrays

- To delete a multidimensional dynamic array
  - Each call to **new** that created an array must have a corresponding call to **delete[ ]**
  - Example: To delete the dynamic array created on the previous slide:

```
for ( i = 0; i < 3; i++)
     delete [ ] m[i]; //delete the arrays of 4 int's
 delete [ ] m; // delete the array of IntArrayPtr's
```

Linked Lists

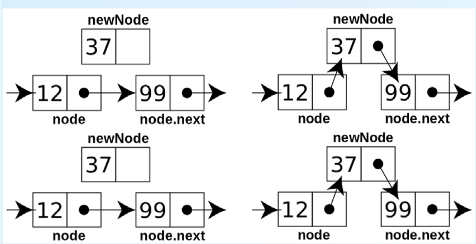
Read Ch. ***13.1*** *in textbook*

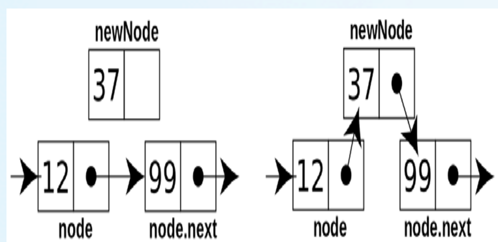
Pointers and Linked Lists

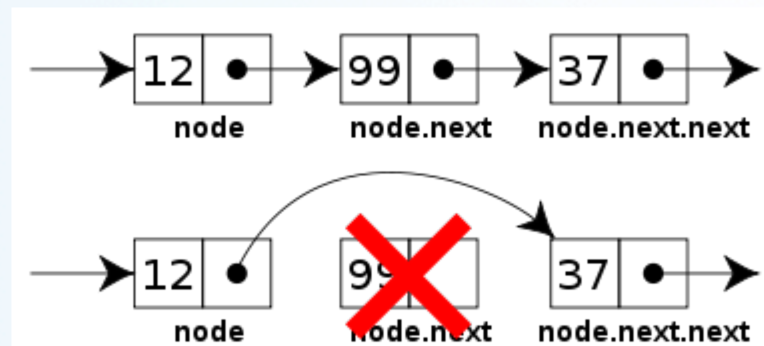
- Pointers are very useful when creating ***linked lists***
- Linear collection of data elements, called ***nodes***, each pointing to the next node by means of a pointer
- List elements can easily be **inserted** or **removed** *without* reorganization of the entire structure (unlike arrays)
- Data items in a linked list do not have to be stored in one large memory block (again, unlike arrays)

Linked Lists

- You can build a list of “nodes” which are made up of variables and pointers to create a chain.
- Adding and deleting nodes in the link can be done by “re-routing” pointer links.







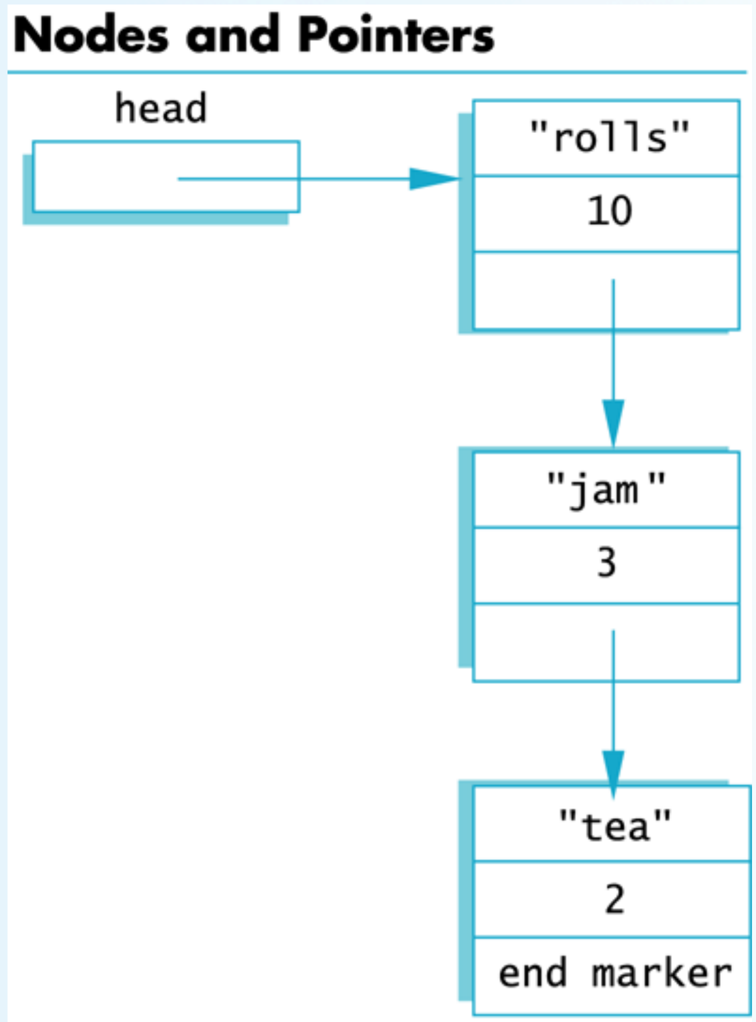
- Chapter 13 in your book explains this further

Nodes

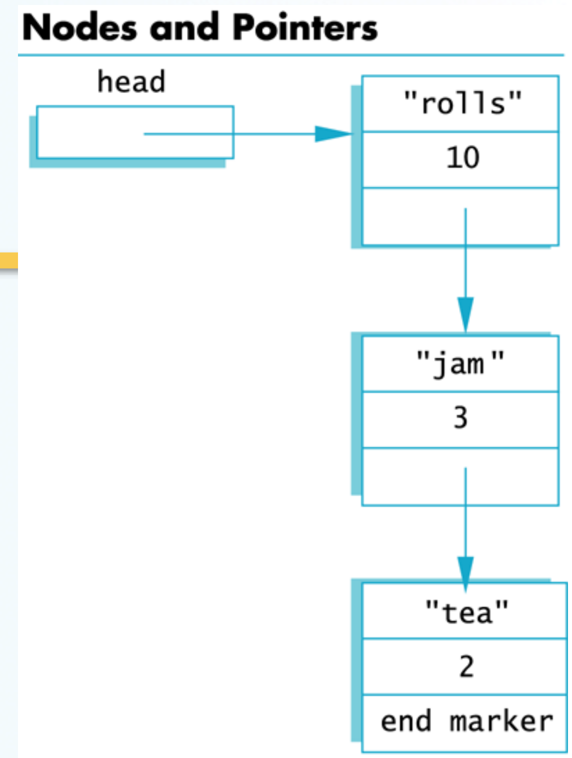
- The boxes in the previous drawing represent the **nodes** of a linked list
  - Nodes contain the data item(s) **and** a pointer that can point to another node of the same type
    - The pointers point to the entire node, not an individual item that might be in the node
- The arrows in the drawing represent pointers

Nodes and Pointers

(shown as Display 13.1 in the textbook)



Implementing Nodes

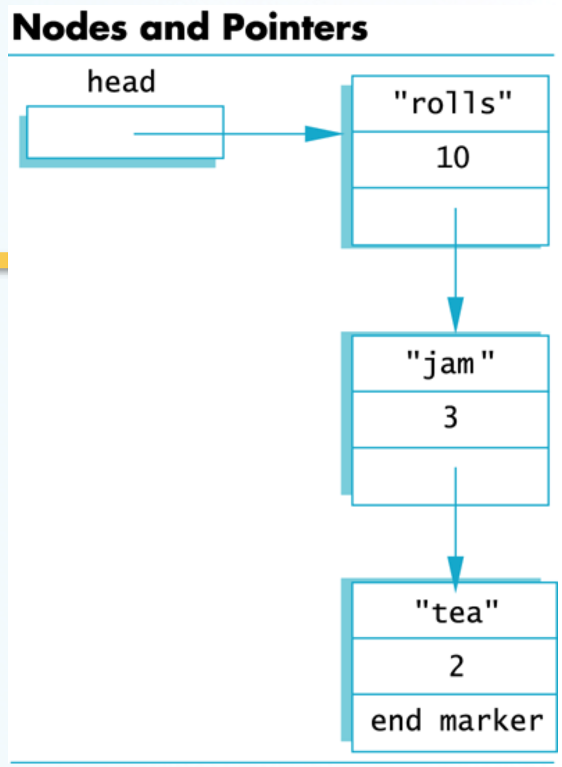


- Nodes are implemented in C++ as **structs** or **classes**
- *Example*: A structure to store two data items and a pointer to another node of the same type, along with a type definition might be:

```
struct ListNode
{
    string item;
    int count;
    ListNode *link;
};

typedef ListNode* ListNodePtr;
```

This circular definition is allowed in C++

The **head** of a List

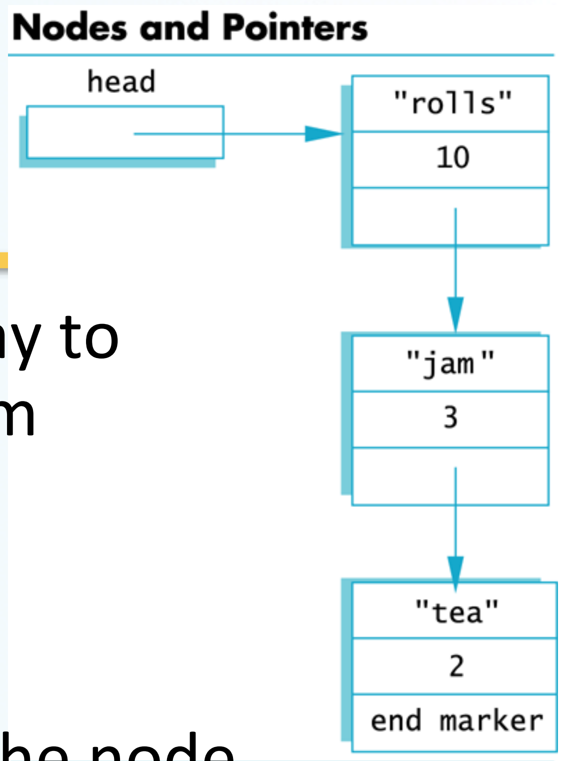


- The box labeled head, in Display 13.1, is not a node, but a **pointer variable** that points to a node

- Pointer variable head is declared as:

```
ListNodePtr head;
```

Accessing Items in a Node



- Using the diagram of 13.1, this is one way to change the number in the first node from 10 to 12:

```
(*head).count = 12;
```

- **head** is a pointer variable so ***head** is the node that **head** points to

- The parentheses are necessary because the dot operator (.) has higher precedence than the dereference operator (*)

The Arrow Operator

- The arrow operator **->** combines the actions of the dereferencing operator * and the dot operator
  - Specifies a member of a struct or object pointed to by a pointer

```
(*head).count = 12;
```

can be written as

```
head->count = 12;
```

- The arrow operator is more commonly used

NULL

- The pre-defined constant NULL is used as an **end marker** for a linked list
  - A program can step through a list of nodes by following the pointers, but when it finds a node containing NULL, it knows it has come to the end of the list
  - The value of a pointer that has nothing to point to

- The value of NULL is 0

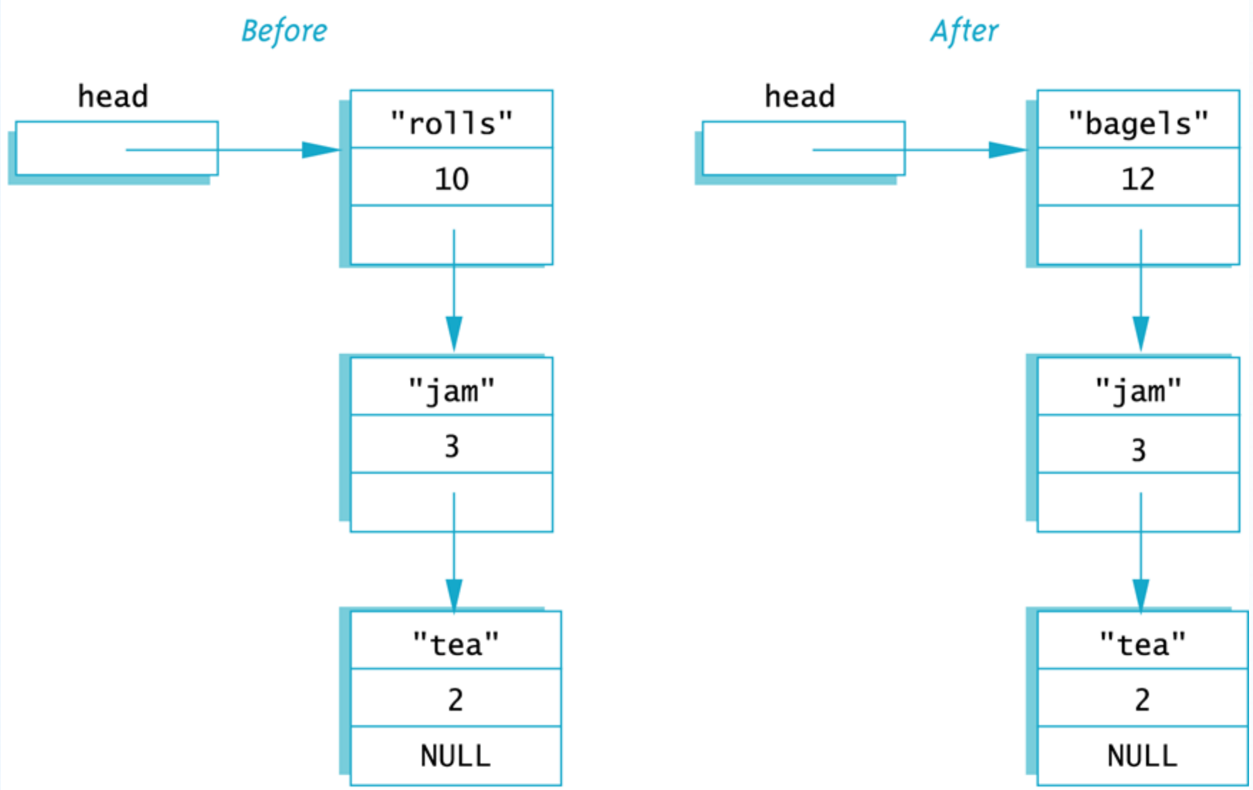
- Any pointer can be assigned the value NULL:

```
double* there = NULL;
```

Accessing Node Data
(shown as Display 13.2 in the textbook)

Accessing Node Data

```
head->count = 12;
head->item = "bagels";
```



Linked Lists in a Nutshell

- The diagram in Display 13.2 depicts a linked list

- A linked list is a list of nodes in which each node has a member variable that is a pointer that points to the next node in the list
  - The first node is called the **head**
  - The pointer variable **head**, points to the first node
    - The pointer named **head** is not the head of the list…it points to the head of the list
  - The last node contains a pointer set to **NULL**

To Use NULL

- A definition of NULL is found in several libraries, including `<iostream>` and `<cstddef>`
- A `using` directive is not needed for NULL

nullptr

- The fact that the constant NULL is actually the number 0 leads to an ambiguity problem.

Consider the overloaded function below:

```
void func(int *p);
void func(int i);
```

- Which function will be invoked if we call `func(NULL)`?

- To avoid this, C++11 has a new constant, **`nullptr`**.
  It is not the integer zero, but a literal constant used to represent a null pointer.

Building a Linked List: The Node Definition

- Let's begin with a simple node definition:

```
struct Node {
    int data;
    Node *link;
};

typedef Node* NodePtr;
```

Building a Linked List: Declaring Pointer Variable head

- With the node defined and a type definition to make or code easier to understand, we can declare the pointer variable head:

```
NodePtr head;
```

Building a Linked List: Creating the First Node

- To create the first node, the operator **new** is used to create a new dynamic variable:

```
head = new Node;
```

- Now head points to the first, and only, node in the list

Building a Linked List: Initializing the Node

- Now that **head** points to a node, we need to give values to the member variables of the node:

```
head->data = 3;
head->link = NULL;
```

- Since this node is the last node, the link is set to NULL

Function **head_insert**

- It would be better to create a function to insert nodes at the head of a list, such as:

```
void head_insert(NodePtr& head, int the_number);
```

  - The first parameter is a **NodePtr** parameter that points to the first node in the linked list
  - The second parameter is the number to store in the list

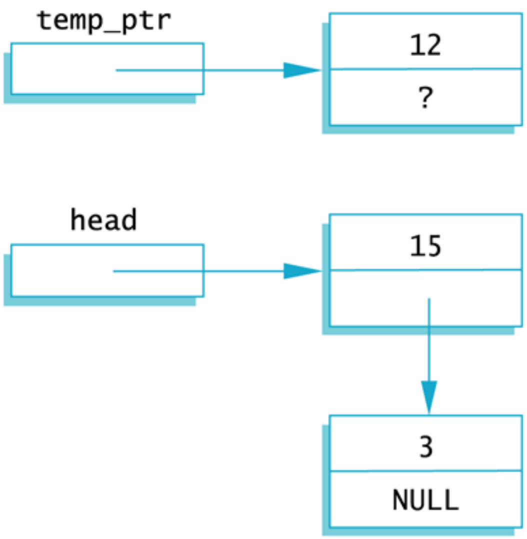
- **head_insert** will create a new node for the number
  - The number will be copied to the new node
  - The new node will be inserted in the list as the new head node

Pseudocode for head_insert

1. Create a new dynamic variable pointed to by **temp_ptr**
2. Place the data in the new node called ***temp_ptr**
3. Make **temp_ptr**'s link variable point to the **head** node
4. Make the head pointer point to **temp_ptr**

Adding a Node to a Linked List

1. Set up new node



Translating **head_insert** to C++

Function to Add a Node at the Head of a Linked List

Function Declaration

```
struct Node
{
    int data;
    Node *link;
};

typedef Node* NodePtr;

void head_insert(NodePtr& head, int the_number);
//Precondition: The pointer variable head points to
//the head of a linked list.
//Postcondition: A new node containing the_number
//has been added at the head of the linked list.
```

Function Definition

```
void head_insert(NodePtr& head, int the_number)
{
    NodePtr temp_ptr;
    temp_ptr = new Node;

    temp_ptr->data = the_number;

    temp_ptr->link = head;
    head = temp_ptr;
}
```

To Dos

- Homework #15 for next Tuesday
- Lab #8 for next week

</LECTURE>