Structures and Nodes in Linked Lists

Modified from Sections 10.1 & 13.1

PEARSON

Structures

Modified from Section 10.1

Structures

- A structure is similar to a class
  - Contains multiple values of possibly different types
    - The multiple values are logically related as a single item
    - Example: A bank Certificate of Deposit (CD) has the following values: a balance, an interest rate, a term (months to maturity)
  - Contains member functions
- The only difference between a class and a struct in C++ is that structs have default public members and bases and classes have default private members and bases.
- In this course, we will only use structures that have no member functions

The CD Definition

- The Certificate of Deposit structure can be defined as

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

Remember this semicolon!

- Keyword struct begins a structure definition
- CDAccount is the structure tag or the structure’s type
- Member names are identifiers declared in the braces

Using the Structure

- Structure definition is generally placed outside any function definition
  - This makes the structure type available to all code that follows the structure definition

- To declare two variables of type CDAccount:

  ```
  CDAccount  my_account, your_account;
  ```

  - my_account and your_account contain distinct member variables balance, interest_rate, and term

Specifying Member Variables

- Member variables are specific to the structure variable in which they are declared
  - Syntax to specify a member variable:

    Structure_Variable_Name**.**Member_Variable_Name
  - Example: given the declarations

    ```
    CDAccount  my_account, your_account;
    ```
    - Use the dot operator to specify a member variable

      ```
      my_account.balance
      my_account.interest_rate
      my_account.term
      ```

Using Member Variables

- Member variables can be used just as any other variable of the same type
    - my_account.balance = 1000;
      your_account.balance = 2500;
        - Notice that my_account.balance and your_account.balance are different variables!
    - my_account.balance = my_account.balance + interest;

Structures as Arguments

- Structures can be arguments in function calls
  - The formal parameter can be call-by-value
  - The formal parameter can be 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

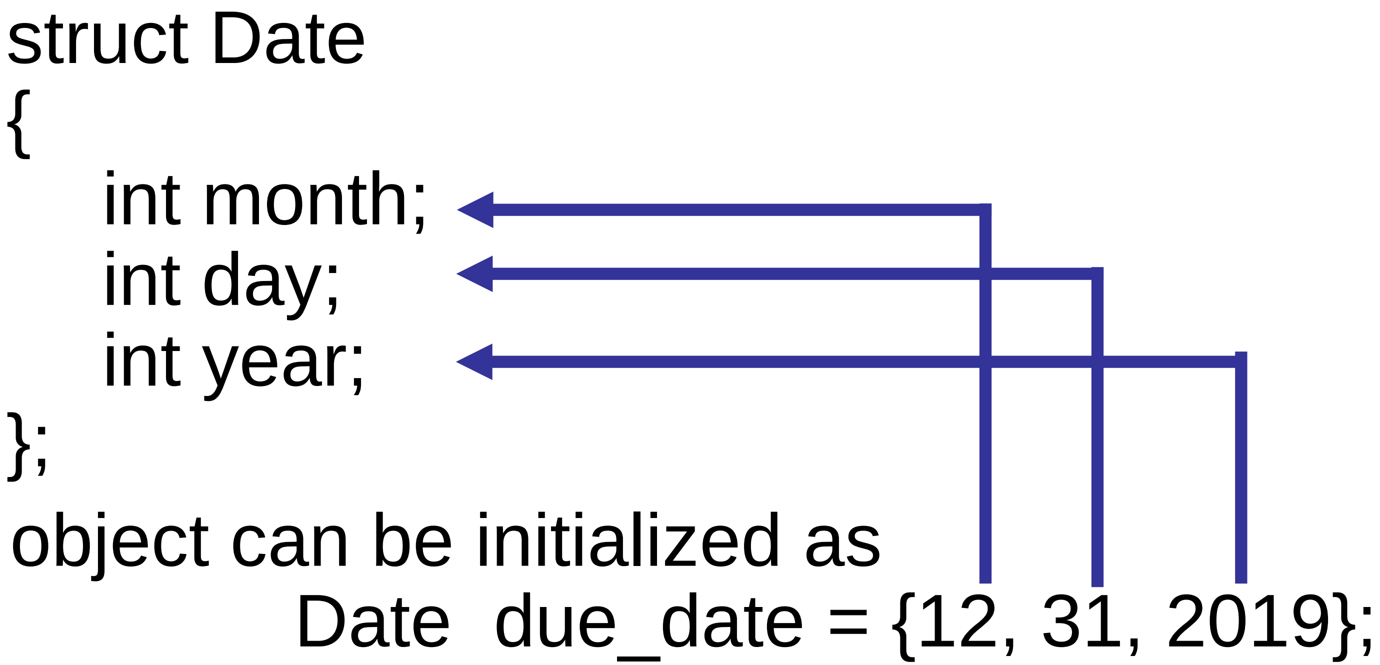
Structures as Return Types

- Structures can 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;
}
```

Initializing Structures

- A structure can be initialized when declared
- Example:



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

- An object can be initialized as

```
Date  due_date = {12, 31, 2019};
```

Initializing Structures

- A structure can be initialized when declared
- Example:

```
struct Date
{
    int month = 12;
    int day = 31;
    int year = 2019;
};
```

  - All Date objects will be initialized as 2019/12/31, unless otherwise specified.

Nodes in Linked Lists

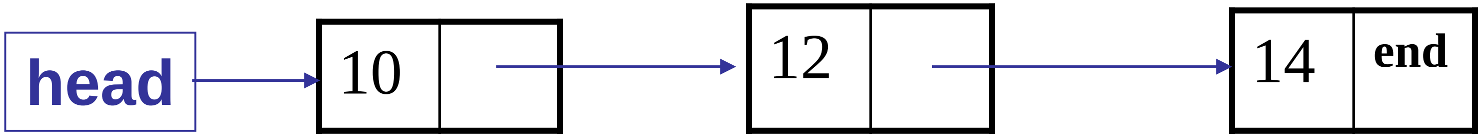
Modified from Section 13.1

Linked Lists

- 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
- A linked list is a list that can grow and shrink while the program is running
- A linked list is constructed using pointers

Nodes and Linked Lists

- A linked list often consists of structs or classes that contain a pointer variable connecting them to other dynamic variables
- A linked list can be visualized as items, drawn as boxes, connected to other items by arrows



Nodes

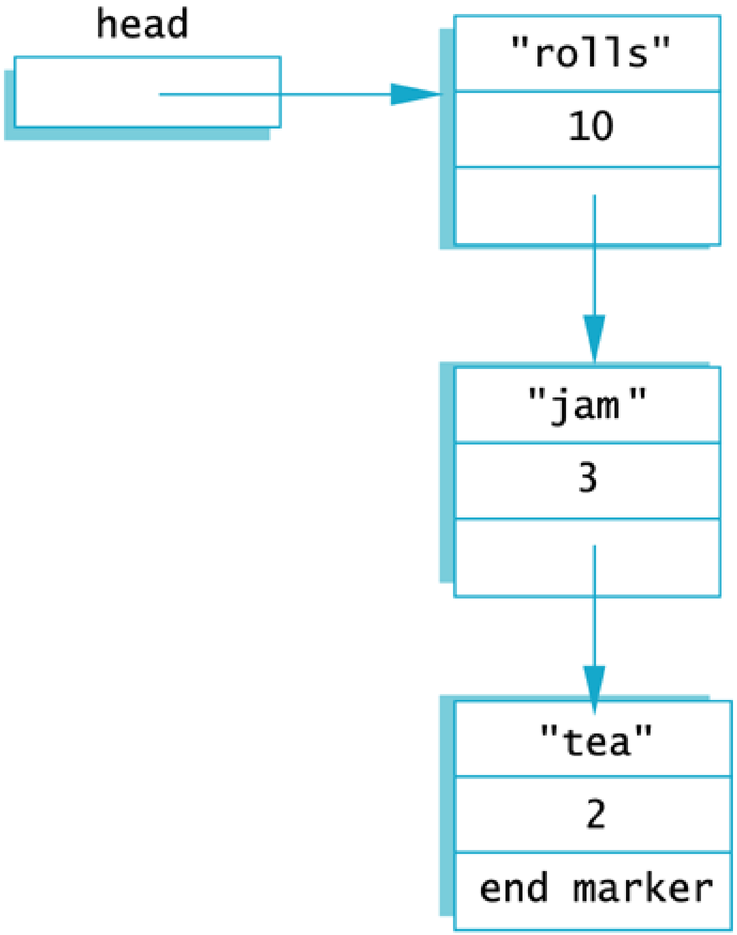
- The boxes in the previous drawing represent the nodes of a linked list
    - Nodes contain the data item(s) and a pointer that can point to another node of the same type
        - The pointers point to the entire node, not an individual item that might be in the node
- The arrows in the drawing represent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

Display 13.1

Display 13.1

Back Next

Nodes and Pointers



Implementing Nodes

- Nodes are implemented 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:

  ListNode* 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 to specify 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

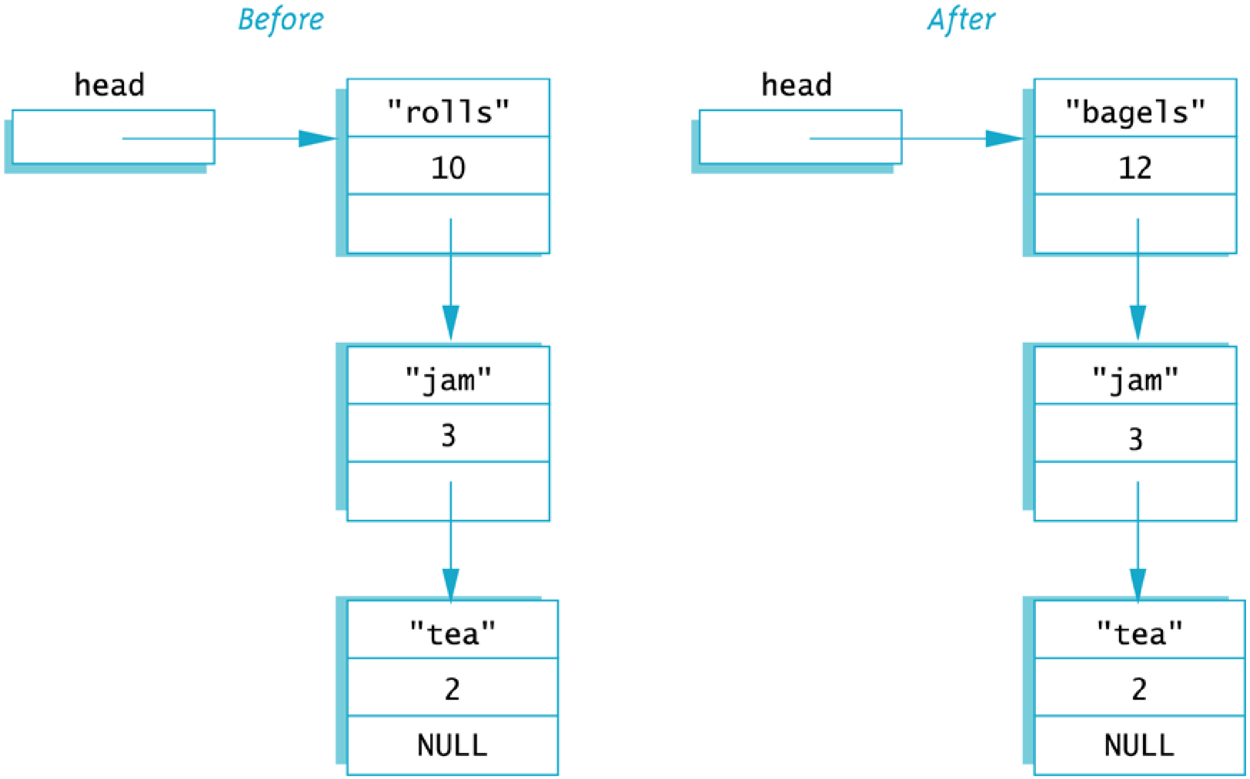
Display 13.2

Display 13.2

Back Next

Accessing Node Data

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



NULL

- The 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;
```

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.