CS 211 – Data Structures

"Bad programmers worry about the code. Good programmers worry about data structures and their relationships."

-- Linus Torvalds

"Good data structures make code easier to design and maintain. It makes software more reliable, systems more understandable, and code more readable. When designing any software, application logic often follows the data model. Treating the data model as an afterthought results in more work down the line. The inverse is true also - having a well-thought out data model makes migrations and building on top of complex systems easier down the line."

-- Engineer's Codex

1

1

Abstract Data Types

- An Abstract Data Type (ADT) is "a data type where programmers who use it do not have access to the details of how the values and operations are implemented"
- There are two ways to interpret what ADT means:
  - In one, it's an object class with public and private parts, where methods like constructors, accessors, mutators, etc. are used to interact with objects
  - In the other, it's a structure that contains such objects as its elements. That is the focus of this course!
- Typical design methodologies focus on developing models of solutions before implementing them
- These models emphasize structure and function of the algorithms used
- Initially our attention is on **what** needs to be done, not **how** it is done

2

2

Abstract Data Types and Data Structures

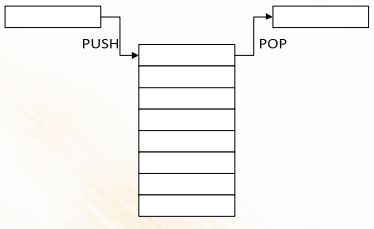
- So we define program behavior in terms of operations to be performed on data
- The details will emerge as we refine the definitions of the operations
- Only then will implementation of those operations be carried out
- As part of this implementation, we must choose appropriate data structures
- A **data structure** is a technique of storing and organizing data so it can be used efficiently
- The online text lists the data structures we will study in CS 211 – there are many others besides that we won't get to!
- A data structure can be independent of the data types it contains. For example, an array can hold integers, or doubles, or strings, etc., as its elements.

3

3

Abstract Data Types (continued)

- ADTs are by necessity defined abstractly, in terms of operations to be performed, rather than in terms of their actual inner structure
- ADTs can then be implemented through class definitions in an object-oriented language
- For example, consider a stack ADT, one of the :



A Simple Stack Representation

4

4

ADT Example – a Stack

- A stack is also called a **last-in first-out (LIFO)** linear structure where items can only be added and removed from one end
- Operations on this stack ADT might include:
  - PUSH – add an item to the stack
  - POP – remove the item at the top of the stack
  - TOP – return the value of the item at the top of the stack
  - EMPTY – determine if the stack is empty
  - CREATE – create a new empty stack
- Notice these simply describe what one does with stacks, not how they are done. If the stack is stored in an array, or a linked list, or some other way… well, the user doesn't need to know what's going on behind the scenes!

5

5

The Use of the C++ Language

First, we'll start with a review of some concepts and their implementations in C++:

- Classes and Objects
- Encapsulation
- Inheritance
- Polymorphism
- Pointers

6

6

Objects, Classes, and Methods

- Fundamental to object-oriented programming (OOP) is the notion of an object
- An ***object*** is a data structure, combined with the operations pertinent to that structure
- Most object-oriented languages (OOLs) define objects through the use of a class
- A ***class*** is a template which implements the ADT defining the objects the class creates
- Within a class, the data elements are called ***data members***, and the operations are called ***methods***

7

7

Data Encapsulation

- The combination of data members and methods in a class is referred to as ***data encapsulation***
- An object, then, can be defined as: "the instantiation of a class, creating an entity that can be used in a program" – this concept is a very powerful and useful tool in modern programming (as you learned in CS 112)
- Deencapsulation binds the data structure and its operations together in the class. The programmer can then focus on the manipulation of objects through their associated methods

8

8

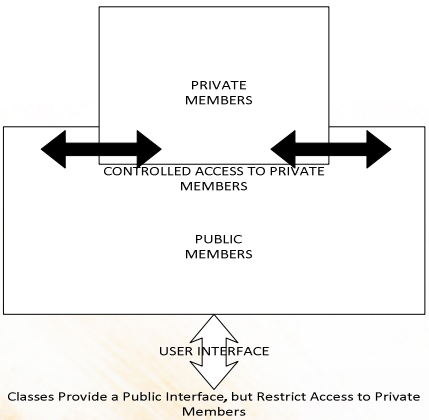
Encapsulation and Information Hiding

- This approach has several advantages:
  - The strong link between data and operations better mimics real-world behavior, on which program models are based
  - Errors in implementation are confined to the methods of a particular class in which they occur, making them easier to detect and correct
  - Details of the implementation of the object can be hidden from other objects to prevent side effects from occurring
- This last point illustrates the principle of ***information-hiding***
- Our use of an object is based on what it does for us, not how it goes about doing it
- So an object can be looked at as a black box, with specific user-available methods and a well-defined behavior

9

9

Encapsulation Illustrated



10

10

Inheritance

- ***Inheritance*** is a technique of reusing existing class definitions to derive new classes
- These new classes (called ***derived classes*** or ***subclasses*** or ***child classes***) can inherit the attributes and behavior of the pre-existing classes (called ***base classes*** or ***superclasses*** or ***parent classes***)
- This relationship of classes through inheritance forms a hierarchy, which can be diagrammed
- Information hiding can be extended through this hierarchy by the access the base class allows the derived class(es)
- Derived classes can then add, delete, & modify methods and data members in their own definitions (known as **overriding**)

11

11

Polymorphism

- Polymorhpism ("many forms") is the general term used to describe the use of overloading and overriding methods in a derived class
- This can be done at compile time, where the C++ compiler recognizes which member function (method) or operator method is to be used depending on context (i.e., the data type(s) of the parameter(s) in the function call or expression) by examining the syntax of the code
- It can also be done at runtime, with the use of virtual methods, where an object's data type is determined dynamically during execution, which then decides which method to call

12

12

Pointers

- A variable defined in a program has two important attributes
  - Its content or value (what it stores)
  - Its location or address (where it is)
- Normally, we access a variable's contents by specifying the variable's name in an operation
- However, it is possible to store a variable's address in another variable
- This new variable is called a pointer; it allows us access to the original variable's value through its address
- A ***pointer*** is a variable whose value is the address of another variable in memory

13

13

Pointers

- To assign a variable's address to a pointer, the **reference** operator (&) is placed before the variable
- For pointer `p_ptr` to point to variable `i`, we write

```
p_ptr = &i;
```

- This pointer is then said to **point to** that variable
- To access the value a pointer points to, we have to **dereference** the pointer, using the dereference operator, an asterisk (*)
- So `*p_ptr` refers to the value stored at `p_ptr`, the address of `i`

14

14

Pointers

- In addition to variables, pointers can be used to access dynamically created locations
- These are created during runtime using the memory manager
- Two functions are used to handle dynamic memory
  - To allocate memory, `new` is used; it returns the address of the allocated memory, which can be assigned to a pointer

```
p_ptr = new int;
```

  - To release the memory pointed at, `delete` is used

```
delete p_ptr;
```

- Care must be exercised when using this type of memory operation!

15

15

Pointers

- One problem can arise when an object is deleted without modifying the value of the pointer
- The pointer still points to the memory location of the deallocated memory
- This creates the ***dangling reference problem***
- Attempting to dereference the pointer will cause an error (the infamous "segmentation violation," as trying to access memory location 0 on the system violates security!)
- To avoid this, after deleting the object, the pointer should be set to a known address or `NULL`, which is equivalent to 0

`p_ptr = NULL;` or `p_ptr = 0;`

16

16

Pointers

- The second type of problem that can occur with dynamic memory usage is the ***memory leak***
- This occurs when the same pointer is used in consecutive allocations, such as:

```
p_ptr = new int;
p_ptr = new int;
```

- Since the second allocation occurs without deleting the first, the memory from the first allocation becomes inaccessible
- Depending on code structure, this leak can accumulate until no memory is available for further allocation
- To avoid this, memory needs to be deallocated when no longer in use. The burden is on the programmer in C++!

17

17

Pointers (continued)

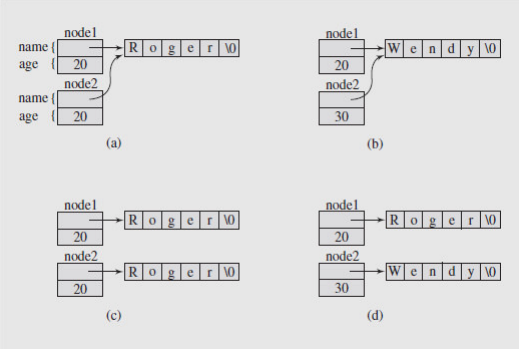
- Pointers and Copy Constructors
  - A potential problem can arise when copying data from one object to another if one of the data members is a pointer
  - The default behavior is to copy the items member by member
  - Because the value of a pointer is an address, this address is copied to the new object
  - Consequently the new object's pointer points to the same data as the old object's pointer, instead of being distinct
  - To correct this, the user must create a **copy constructor** which will copy not only the pointer, but the object the pointer points to
  - This conflict is illustrated on the next slide

18

18

Pointers (continued)

- Pointers and Copy Constructors (continued)



Illustrating the necessity of using a copy constructor for objects with pointer members

Pointers (continued)

- Pointers and Reference Variables
  - There is a close correspondence between pointers and reference variables
  - This is because reference variables are implemented as constant pointers
  - Given the declarations:

```
int n = 5,p_ptr = &n,&r_ref = n;
```

    a change to the value of `n` via any of the three will be reflected in the other two
  - Thus `n = 7` or `*p_ptr = 7`, or `r_ref = 7` accomplishes the same result; the value of `n` is now 7
  - So we can dereference a pointer, or use a reference directly to access the original object's value

Pointers

- Pointers and Reference Variables (continued)
  - We do have to exercise care in declaring pointers, though, because

    ```
    int *const p_ptr;
    ```

    declares a constant pointer to an integer (whose value can be changed!), while

    ```
    const int * p_ptr;
    ```

    declares a pointer to a constant integer
  - The latter can cause errors if we attempt to assign a value through a dereferenced pointer
  - Reference variables are valuable in working with functions, as they allow us to modify the values of arguments
  - They can also be returned from a function
  - While pointers can be used instead, they have to be dereferenced

21