1

C++ ADTs and Classes

CSC 340 - Credit to Hui Yang

February 24, 2016

2

Overview

- **ADTs**
- Separate Compilation
- Classes

3

Abstract Data Types

- This should be review…
- An ADT is composed of
  - A collection of data members
  - A set of operations on these data members
- Specifications of an ADT tell you
  - **What** the ADT operations do
  - **Not how** to implement them
  - In C++: the header file (*.h)

Abstract Data Types

- Implementation of an ADT
  - Includes choosing a particular data structure (this is an implementation detail of the given implementation - not the only way to implement an ADT)
  - In C++: the implementation file (*.cpp)

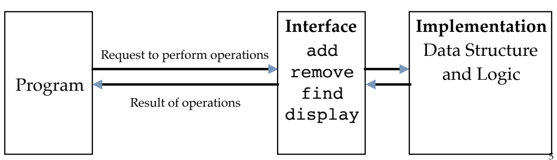
4

4

Examples: Linked List implementation can be with dynamically created "node" objects, or with a dynamic array. Implementation doesn't matter, just the behavior that is exposed by the specification.

Abstract Data Types

- We call the specification the interface
- The interface isolates the underlying implementation (data structures, logic, etc.) from the program that uses it



5

5

Example

- Lists
  - Except for the first and last items in a list, each item has a unique predecessor and a unique successor
    - Head (or front) does not have a predecessor
    - Tail (or end) does not have a successor
- Sorted Lists
  - Maintains items in sorted order
  - Inserts and deletes items by their values, not their positions

6

6

How is this implemented? (Hint: It's a trick question)
If we wrote the interface for this, what would it look like? UML? C++?

Designing an ADT

- The design of an ADT should evolve naturally during the problem solving process
- Questions to ask when designing an ADT
  - What **data** does a problem require?
  - Any constraints? (For example, maximum number of items in a list?)
  - What **operations** does a problem require?
  - What are the most frequently used operations?

7

7

You'll see this as the project continues to evolve throughout the semester - implementation decisions or domain models will change as we evolve the system….

Sorted Lists: Interface

```
Sorted List
-----------------------------------------------------------------
+ isEmpty() : boolean
+ getLength() : integer
+ insert( in newItem : ListItemType, out success : boolean )
+ remove( in index : integer, out success : boolean )
+ retrieve( in index : integer, out dataItem : ListItemType,
  out success : boolean )
+ position( in item : ListItemType, out isPresent : boolean )
  : integer
```

8

8

Side note - how do we get in and out parameters in C++?

Interface == a contract

We're using UML to show the interface

Top - members (why are there none here?) - we could define as private with - if we wanted

Bottom - operations/behavior/methods

Why ADTs?

- Modularity
  - Keeps the complexity of a large program manageable by systematically controlling the interaction of its components
  - Isolates errors
  - Eliminates redundancies

9

9

10

Why ADTs?

- Functional Abstraction
  - Separates the purpose and use of a module from its implementation (we'll do this in an upcoming assignment)
  - A module's specification should
    - Detail how the module behaves
    - Be independent of the module's implementation (also in that upcoming assignment)

11

Why ADTs?

- Information hiding
  - Hides certain implementation details within a module
  - Makes those details inaccessible from outside the module

12

Implementing ADTs

- Classes in C++
  - Many similarities with Java classes
- Choosing the data structure to represent the ADT's is part of implementation
  - Depends on details of the ADT's operation
  - Depends on the context in which the operations will be used

13

Overview

- ADTs
- **Separate Compilation**
- Classes

13

14

Separate Compilation

- For remaining projects (after the assignment we begin tonight), we will use separate compilation units to create our modules
- Header file (*.h) is the interface
- Implementation file (*.cpp) is the implementation of each member function
- Our makefiles will help us manage the complexity of compilation

14

15

Separate Compilation

- Header Files (*.h)
  - Function declarations
  - Declaration of a class (ADT)
- Implementation Files (*.cpp)
  - Implementation of each function (both member and non-member/friends)
- The main() file (main.cpp for us)
  - Include routines that use the functions or classes declared in the headers

15

Main can have a header too, but if we do it correctly, it won't need it

16

Compiling in C++

- Java took care of a lot of compilation detail - finding the compilation units, ensuring they were included once.
- C++ does not, so we must take care to ensure things are declared only once during compilation

17

Using #ifndef in the Header file

- Preprocessor directive `#ifndef` tells compiler if a unique constant has not been defined, include the code specified between the `#define` and `#endif` directives

```
#ifndef __PERSON_H__
#define __PERSON_H__
// Person class declaration
// NOT Implementation!!
#endif
```


18

Using #ifndef in the Header file

- The first time a `#include "person.h"` is found, `PERSON_H` and the class are defined
- The next time a `#include "person.h"` is found, all lines between `#ifndef` and `#endif` are skipped

```
#ifndef PERSON_H
#define PERSON_H
// Person class declaration
// NOT Implementation!!
#endif
```

19

Including Header Files

- The implementation file for a given class, and anywhere that class is used, must include the header file:
  `#include “person.h”`
- NOT angle brackets! (this refers to a system library - does not look in current directory)
  `#include <person.h>`

19

Also - ONLY THE IMPLEMENTATION FILE INCLUDES THE HEADER, not the other way around!!

20

Why Separate Compilation?

- Reuse: The header file can be reused by main programs
- Changing the implementation file does not require changing the program using the header file
- Summary:
  - Facilitates code sharing
  - Improves code reusability
  - Reduces the workload of the programmer
  - Reduces unnecessary interaction between designers and users

20

21

Overview

- ADTs
- Separate Compilation
- **<u>Classes</u>**

21

C++ Class Syntax

```
class ClassName
{
  private:
    member_one_private_specification;
    …
    member_n_private_specification;
  public:
    member_one_public_specification;
    …
    member_n_public_specification;
};
```


22

What does a member specification look like?
Either data member or method declarations

Syntactic Differences

C++

```
class Person
{
  private:
    string fname;
    string lname;
    int age;
  public:
    // default constructor
    Person();
    Person(string, string, int);
    // accessor
    string get_fname();
};
```

Java

```
public class Person
{
  private string fname;
  private string lname;
  private int age;

  public Person();
  public Person(string, string, i
  public string get_fname();
};
```


23

Semicolon ends declaration in C++
private and public explicit for each member in Java

C++ Constructors

- Create and initialize new instances of a class
  - Invoked when you declare an instance of the class
- Have the same name as the class
- Have no return type, not even `void`
- A class can have several constructors
  - A default constructor has no arguments
  - The compiler will generate a default constructor if you do not define any constructors

24

25

C++ Constructors

- The implementation of a method (in the *.cpp file) qualifies its name with the scope resolution operator ::
- The implementation of a constructor
  - Sets data members to initial values (and can use an initializer)

    ```
    Person::Person() : age(20)
    {}
    ```
  - Cannot use return to return a value

26

C++ Class Operators . and ::

- :: used with ***classes*** to identify a member

  ```
  string Person::get_first_name()
  {
      return first_name;
  }
  ```
- . used with ***variables (instances)*** to identify a member

  ```
  Person john;
  john.get_first_name();
  ```


27

Initialization Sections

- An initialization section in a function definition provides an alternative way to initialize member variables

  ```
  Person::Person() :
    first_name("N/A"), last_name("N/A"), ssn(-1), age(-1)
  {
    // No code needed - everything init'ed in init
  section
  }
  ```
- The values in the parentheses are the initial values assigned to the data member variables listed

28

Initialization List Semantics

- Data members on the list are initialized immediately
- Does not make a huge difference for primitive data types
- Save CPU cycles when a data member is an object of another type. Otherwise:
  - A default constructor will be called first
  - An = operator is called next to overwrite the initial value

29

When to Use an Initialization List?

- If a data member's type does not provide a default constructor
- If a superclass does not have a default constructor (inheritance)
- Constant members
- If the value of the data member has no restrictions
  - Otherwise use a set method (mutator) if the value of the data member needs validation

30

Calling a Constructor

- A constructor is called automatically at the object declaration:
  `Person john("john", "roberts", 37, …);`
- The above example creates a Person object by calling the corresponding constructor, which also initializes all the members using the specific value (note that this is ***not*** the default constructor)

31

The explicit Keyword

- C++ automatically invokes a one-parameter constructor for implicit type conversion
- Example: assume the Person class has the following constructor:

```
Person::Person( int new_age )
{
    age = new_age;
}

// This would be legal
Person John = 20;
```


32

The explicit Keyword

- To avoid implicit type conversion

```
class Person
{
  public:
    explicit Person(int);
}
```

- A general rule: one-parameter constructor should be made `explicit`

33

Accessors and Mutators

- Accessors: Read-only member functions
  - Place the keyword `const` at the end of a function declaration
  - Not supported in Java
  - A nice feature in C++ for performance optimization
- Mutators: member functions that do not promise not to change the state of an object

34

Implementing a Member Function

- Similar to constructor implementation
- Member functions are *declared* in the class declaration (in the *.h file)
- Member function definitions identify the class in which the function is a member (in the .cpp file)

```
string Person::get_first_name() const
{
  return first_name;
}
```

34

35

Please do not copy this code format - for fitting on a slide only

const Modifies Functions

- If a constant parameter makes a member function call, the member function being called must be marked so the compiler knows it will not change the parameter

```
class Person {
  public:
    int get_age() const;
}
```

- `const` is used to mark functions that will not change the value of an object
- `const` is used in the function declaration and the function definition

```
int Person::get_age() const { return age; }
```

35

36

Use const Consistently

- Once a parameter is modified by using `const` to make it a constant parameter
  - Any member functions that are called by the parameter must also be modified using `const` to tell the compiler that they will not change the parameter
  - It is a good idea to modify, with `const`, every member function that does not change a member variable

36

C++ Destructors

- Destroys an instance of an object when the object's lifetime ends
- Each class has one destructor
  - For many classes, you can omit the destructor
  - The compiler will generate a destructor if you do not define one

37

37

Why are these useful?

Inline Functions

- Inlining causes invocations of a function to be replaced by their definition
  - Eliminates overhead
  - More efficient, but only when short
- For non-member functions (not belonging to a class)
  - Use keyword inline in function declaration and `function heading`

```
// Declaration
inline void f(int, char);
// Implementation
inline void(int i, char c) { … }
```

38

38

no stack frames

Inline Functions

- For class member functions
  - Place implementation for function IN class definition (*.h) - this automatically inlines
  - Use for very short functions only

39

39

40

Namespaces

- A mechanism for logically grouping declarations and definitions into a common declarative region
- Similar to packages in Java
- Using statement is equivalent to import directive ending in .* in Java
  `using namespace some_namespace;`
- Example

```
namespace some_namespace
{
  class classA {
  };
}
```


41

Namespaces

- In Java, classes can be declared as public or package visible. C++ does not allow this
- Classes, functions, and objects that are declared outside of any namespace are considered to be in the global namespace (`::classA`)
- Classes, functions, and objects can also be declared within an anonymous class. When this happens, they can only be accessed within the compilation unit (*.cpp)

Examples on next slide

42

Nested Classes

- A class can be declared within another class

```
class Outer {
  private:
    class Inner {
      public:
        int member_one;
    }
}
```

- A nested class in C++ behaves in a manner similar to a static nested class in Java
- If `member_one` is private, `Outer` can not access it
- Private members in `Outer` are not accessible by `Inner`

43

Nested Classes

- C++ does not support Java-style inner classes in which instances of the inner class are constructed with the hidden reference to an outer object that caused its creation
- C++ allows local classes in which a class is declared inside a function, but this is discouraged due to its limited usage
- C++ does not allow anonymous classes

43

44

Circular Class References

- Class A refers to B and Class B refers to A
- Solution: pre-declaration

```
class B;

class A         class B
{               {
  // …            // …
  B data;         A data;
}               }
```

44

45

Conclusion

- More C++ user-defined types
- Review of ADT
- Separate compilation
- C++ Classes

45