CSC340: Abstract Data Type in C++

Main Topics:

1. **Motivations behind abstract data types (ADTs)**
2. **Specifying and implementing ADTs in C++**
3. **Common class implementation constructs: initialization list, const, explicit and inline**
4. **Example: an ADT for sorted lists with exception handling.**

Readings: Chapter 3 (Version 5)

Hui Yang
Computer Science Department
San Francisco State University
http://www.cs.sfsu.edu/~huiyang/

Abstract Data Types

- **An ADT is composed of**
  - A collection of data members
  - A set of operations on these data members
- **Specifications of an ADT tell**
  - **What** the ADT operations do
  - **Not how** to implement them
  - The header file: *.h
- **Implementation of an ADT**
  - Includes choosing a particular data structure
  - The implementation file: *.cpp

Abstract Data Types

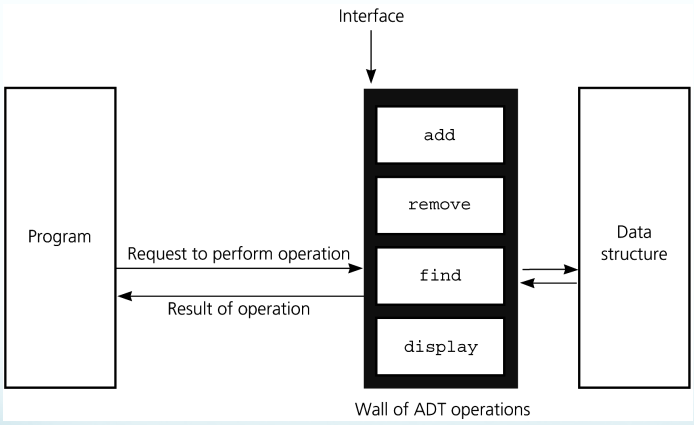

Figure 3-4

A wall of ADT operations isolates a data structure from the program that uses it

Example: complex numbers

- **Note: C++ contains a <complex> library, which is a template**
- **We are going to implement our own Complex ADT**
- **Design questions:**
  - What will be the data members?
  - What operations (or methods, or member functions) do we need to include?
  - Any constraints?
  - What are the most frequently used operations?

Implementing ADTs

- **Classes in C++**
  - Similar to Java classes
- **Choosing the data structure to represent the ADT's data is a part of implementation**
  - Choice of a data structure depends on
    - Details of the ADT's operations
    - Context in which the operations will be used
- **Separate compilation**
  - Header file (*.h): the interface
  - Implementation file (*.cpp): implementation of each member function

C++ class: syntax

```
class Class_Name
{
  private:
        Member_Specification_n+1
        Member_Specification_n+2
        ...
   public:
        Member_Specification_1
        Member_Specification_2
        ...
        Member_Specification_3

};
```

Syntactic differences

C++

```
class Person{
 private:
    string fname;
    string lname;
    int      ssn; //social security number
    int      age;
    double salary;
 public:
    Person(); //default constructor
    Person(string fn, string ln, int thessn, int theage,
    int thesal);
    string get_fname(); //an accessor, returns fname
    //a set of accessors
    //a set of mutators
};
```

Java

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

    public Person();
    public Person(string fn, string ln, int thessn, int
                  theage, int thesal);
    public string get_fname();
    //a set of accessors
    //a set of mutators
}
```


Separate Compilation

- **Header file (*.h)**
  - Function declarations
  - Declaration of a class (ADT)
- **Implementation file (*.cpp)**
  - Implementation of each function
    - Member function
    - Non-member function
- **The main() file (*.cpp)**
  - Include routines that use the functions or classes declared in *.h

Using #ifndef in the Header file

- **Consider this code in the interface file person.h**

```
#ifndef PERSON_H
 #define PERSON_H
 //< The Person class >
#endif
```

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

Include the header file

- **The implementation file and the main() file need to include the following directive:**

```
#include “Person.h”
```

  - NOT

```
#include <myheader.h>  //incorrect
```

Why separate compilation?

- **The header file can be reused by many programs.**
- **Changing the implementation file does not require changing the program using the header file.**
- **In summary**
  - Facilitate code sharing
  - Improve code reusability
  - Reduce the workload of a programmer
  - Reduce unnecessary interaction between designers and users

Why ADTs (1)?

- **Modularity**
  - Keeps the complexity of a large program manageable by systematically controlling the interaction of its components
  - Isolates errors
  - Eliminates redundancies
  - Benefits: easy to read/write/maintain
- **Functional abstraction**
  - Separates the purpose and use of a module from its implementation
  - A module's specifications should
    - Detail how the module behaves
    - Be independent of the module's implementation

Why ADTs ? (2)

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

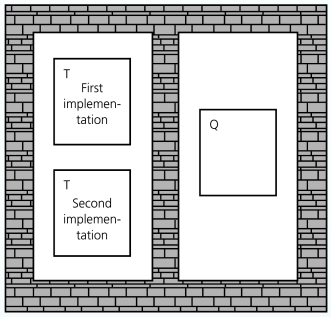

Figure 3-1 Isolated tasks: the implementation of task *T* does not affect task *Q*

C++ Classes: Constructors

- **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

C++ Classes: Constructors

- **The implementation of a method qualifies its name with the scope resolution operator ::**
- **The implementation of a constructor**
  - Sets data members to initial values
    - Can use an initializer

```
Person::Person() : age(20)
{
} // end default constructor
```

  - Cannot use return to return a value

'::' and '.'

- **'::' used with classes or namespaces to identify a member**

```
string Person::get_fname()
{
    return fname;
 }
```

- **'.'used with objects to identify a member**

```
Person john;
john.get_fname();
```

Initialization Sections

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

```
Person::Person( ): fname("na"),lname("na"),ssn(-
1), age(-1), salary(0.0)
{
   // No code needed in this example
}
```

- **The values in parenthesis are the initial values for the member variables listed**

The 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 class. Otherwise:**
  - A default constructor will be called first
  - An = operator is called next to overwrite the initial value

When to Use an Initialization List?

- **If a data member does not have 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 if the value of the data member needs validation.

Calling A Constructor

- **A constructor is called automatically at the object declaration**

```
Person john("john", "adams", 11, 20, 20000);
```

- **The above example create a Person object by calling the corresponding constructor, which also initializes all the members using the specified values.**

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( double newsalary)
{
    salary = newsalary;
}
//in another function
Person John = 20.00;  //this is legal
```

- **To avoid implicit type conversion**

```
class Person{
        public:
                explicit  Person( double );
        // other components
};
```

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

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: members functions that do not promise not to change the state of an object**

Implementing a Member Function

- Similar to constructor implementation
- Member functions are declared in the class declaration
- Member function definitions identify the class in which the function is a member

```
string Person::get_fname() const
{
        return fname;
}
```


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{
        private:
                ...
        public:
                ...
                string get_age() const; //return age
    };
```

- 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

Function Definitions With const

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


C++ Classes: Destructors

- **Destructor**
  - 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
    - For now, we will use the compiler's destructor

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 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

Inline Functions (1)

- **For non-member functions:**
  - Use keyword *inline* in function declaration and function heading
  - Declaration:

```
void f(int i, char c);
```

  - Inline implementation

```
inline
void f(int i, char c)
{
  …
}
```

Inline Functions (2)

- **For class member functions:**
  - Place implementation (code) for function IN class definition → automatically inline
- **Use for very short functions only**
- **Code actually inserted in place of call**
  - Eliminates overhead
  - More efficient, but only when short!

Inline Member Functions

- **Member function definitions**
  - Typically defined separately, in different file
  - Can be defined IN class definition
    - Makes function "in-line"
- **Again: use for very short functions only**
- **More efficient**
  - If too long → actually less efficient!

Namespaces

- **A mechanism for logically grouping declarations and definitions into a common declarative region**
- **Similar to packages in Java**
- **Using namespace namespace1;**
  - Equivalent to the import directive ending with .* in Java
  - namespace1::classA;

```
namespace namespace1
{
    class classA {
    };
}
```

- **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. Such being the case, they can only be accessed within the compilation unit (e.g., *.cpp).**

Nested Classes (1)

- **A class can be declared within another class**

```
class Outer{
        private:
                class Inner{
                        public:
                                int member1;
                };
                Inner data1;
        public:
                //
};
```

- **A nested class in C++ behaves in a manner similar to a static nested class in Java**
- **If *member1* is private, *Outer* cannot access it.**
- **Private members in *Outer* are not accessible by *Inner*.**

Nested Classes (2)

- 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 limited usage.
- C++ does not allow anonymous classes.

Circular Class References

- Class A refers to B; class B refers to A
- Solution: pre-declaration

```
class B;

Class A
{
   ...
   B data1;
   ...
};
```

```
Class B
{
   ...
   A data1;
   ...
};
```

Summary

- **Abstract data types: why?**
- **Designing an ADT**
- **Implementing an ADT**
  - Separate compilation
  - C++ classes vs. Java classes
  - Constructors: implicit vs. explicit
  - Initialization list
  - Const
  - Inline
- **Exception handling revisited**