Chapter 3: Data Abstraction

• Abstraction, modularity, information hiding
• Abstract data types
• Example-1: List ADT
• Example-2: Sorted list ADT
• C++ Classes
• C++ Namespaces
• C++ Exceptions
Modularity and Abstraction

• Important when developing large programs.
• Divide program in small *manageable* modules
  – each module understood individually
  – easier to write, understand, modify, and debug
• Modules communicate using *well-defined* interfaces
  – different module implementations use same interface
  – provide a different and easier interface to communicating modules – abstraction
Fundamental Concepts

• Modularity
  – manages complexity of large programs
  – isolates errors
  – eliminates redundancies
  – program is easier to read, write, and modify

• Information hiding
  – hides certain implementation details within a module
  – makes these details inaccessible from outside the module
Abstraction

- **Functional abstraction**
  - separates the purpose and use of a module from its implementation
  - module’s specifications only details its behavior, independent of the module’s implementation

- **Data abstraction**
  - asks you to think what you can do to a collection of data independently of how you do it
  - allows you to develop each data structure in relative isolation from the rest of the solution
Isolated Tasks

T
First implementation

T
Second implementation

Q
Isolation of Modules is Not Total

• A function’s specification, or contract, governs how it interacts with other modules

![Diagram showing a program that uses method S and the implementation of method S with a slit in the wall representing the interaction](image)

*Figure 3-2* A slit in the wall
Abstract Data Type (ADT)

- An ADT is composed of
  - collection of data
  - set of operations on that data

- Specifications of an ADT indicate
  - what the ADT operations do, not how to implement them

- Implementation of an ADT
  - includes choosing a particular data structure
Abstract Data Types

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

Figure 3-4
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?
  – What operations does a problem require?
List ADT Example

• ADT for a list of items: grocery list, TO-DO list
• What *operations* do we perform on/with a list?
  – add item, delete item, find item, read, etc.
  – cannot think of everything?
    • should refine iteratively!
• How to store the data
  – implementation detail hidden from users of the list
  – arrays or linked lists
List ADT Example – Properties

• Except for the first and last items, each item has a unique predecessor and successor
• Items are referenced by their position in the list
• Specifications of the ADT operations
  – Define an operation contract for the ADT list
  – Do not specify how to store the list or how to perform the operations
• ADT operations can be used in an application without the knowledge of how the operations will be implemented
List ADT Example – Operations

• Create an empty list
• Destroy a list
• Determine whether a list is empty
• Determine the number of items in a list
• Insert an item at a given position in the list
• Delete the item at a given position in the list
• Retrieve the item at a given position in the list
List ADT – Operation Contract

• `createList()`
• `destroyList()`
• `isEmpty()`: boolean {query}
• `getLength()`: integer {query}
• `insert(in index: integer, in newItem: ListItemType, out success: boolean)`
• `remove(in index: integer, out success: boolean)`
• `retrieve(in index: integer, dItem: ListItemType, out success: boolean) {query}`

see Table on pages 128-129
List ADT Example – Operations

• Create the list -- milk, eggs, butter
  – aList.createList()
  – aList.insert(1, milk, success)
  – aList.insert(2, eggs, success)
  – aList.insert(3, butter, success)

• Insert bread after milk
  – aList.insert(2, bread, success)
milk, bread, eggs, butter

• Insert juice at end of list
  – aList.insert(5, juice, success)
milk, bread, eggs, butter, juice
List ADT Example – Operations

• Remove eggs
  – aList.remove(3, success)
  – milk, bread, butter, juice

• Insert apples at beginning of list
  – aList.insert(1, apples, success)
  – apples, milk, bread, butter, juice
List ADT Example -- Operations

- Algorithm description independent of list implementation, as long as each item has an index
- Pseudocode function that displays a list

```pseudocode
displayList(in aList:List)
{
    for (position=1 to aList.getLength()){
        aList.retrieve(position, dataItem, success)
        display dataItem
    }
}
```
List ADT Example -- Implementation

- How to implement the List ADT?
- A list’s $k^{th}$ item is stored in items[$k-1$]
- To insert an item, make room in the array
List ADT Example -- Implementation

- To delete an item, remove gap in array

Figure 3-13 (a) Deletion causes a gap; (b) fill gap by shifting
List ADT – Options

• Many other design options are possible
  – retrieve items by name, instead of by index
  – sort items by name or some other factor
  – display list in some sorted order

• Several data structures can be used during implementation
  – arrays, linked lists, trees, hash-tables, etc.
  – different advantages, restrictions, and costs
ADT Sorted List -- Properties

• Maintains items in sorted order
• Inserts and deletes items by their values, not their positions
ADT Sorted List – Operation Contract

- sortedIsEmpty(): boolean
- sortedGetLength(): integer
- sortedInsert(in nltem: ListItemType, out success: boolean)
- sortedRemove(in index: integer, out success : boolean)
- sortedRetrieve(in index: integer, out dltem: ListItemType, out success : boolean)
- locatePosition(in anltem: ListItemType, out isPresent: boolean): integer
Implementing ADTs

• 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

• Implementation details should be hidden behind a wall of ADT operations
  – A program (client) should only be able to access the data structure by using the ADT operations
Hiding Data Structures and Code

ADT operations provide access to a data structure

Figure 3-8
ADT operations provide access to a data structure
Violating Information Hiding

**Figure 3-9** Violating the wall of ADT operations
C++ Classes

• Encapsulation combines an ADT’s data with its operations to form an object
  – an object is an instance of a class
  – a class defines a new data type
  – a class contains data members and methods (member functions)
  – by default, all data members in a class are private
    • but, can specify them as public
    • can only be accessed by other class members
  – some member functions have to be public
  – encapsulation hides implementation details
C++ Classes

Figure 3-10
An object’s data and methods are encapsulated.
C++ Classes

• Each class definition is placed in a header file
  – Classname . h

• The implementation of a class’s methods are placed in an implementation file
  – Classname . cpp
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
  – 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

    Sphere::Sphere() : theRadius(1.0)
    {
    }
    // end default constructor

  – cannot use return to return a value
C++ Classes: Destructors

• Destructor
  – destroys an instance of an object when the object’s lifetime ends
  – called automatically for local variables on subroutine exit
  – called explicitly by `delete` operator
  – primary duty is to de-allocate dynamic memory

• Each class has one destructor
  – for many classes, you can omit the destructor
    • if they do not allocate any memory
  – the compiler will generate a destructor if you do not define one
**C++ Classes: The header file**

```cpp
/** @file Sphere.h */
const double PI = 3.14159;
class Sphere
{
    public:
        Sphere();                     // Default constructor
        Sphere(double initialRadius); // Constructor
        void setRadius(double newRadius);
        double getRadius() const;    // can’t change data members
        double getDiameter() const;
        double getCircumference() const;
        double getArea() const;
        double getVolume() const;
        void displayStatistics() const;

    private:
        double theRadius;       // data members should be private
};  // end Sphere
```
/** @file Sphere.cpp */
#include <iostream>
#include "Sphere.h"  // header file
using namespace std;
Sphere::Sphere() : theRadius(1.0)
{
}  // end default constructor

Sphere::Sphere(double initialRadius)
{
    if (initialRadius > 0)
        theRadius = initialRadius;
    else
        theRadius = 1.0;
}  // end constructor
void Sphere::setRadius(double newRadius) {
    if (newRadius > 0)
        theRadius = newRadius;
    else
        theRadius = 1.0;
} // end setRadius

• The constructor could call setRadius
double Sphere::getRadius() const
{
    return theRadius;
} // end getRadius

... 

double Sphere::getArea() const
{
    return 4.0 * PI * theRadius * theRadius;
} // end getArea

...
```cpp
#include <iostream>
#include "Sphere.h" // header file
using namespace std;
int main() // the client
{
    Sphere unitSphere;
    Sphere mySphere(5.1);
    cout << mySphere.getDiameter() << endl;
    . . .
} // end main
```
Inheritance in C++

• Inheritance is a way to reuse the code (and behavior) of existing classes
  • existing class is called the base or super or parent class
  • the new class is called derived or sub class

• Derived class inherits any of the publicly defined methods or data members of a base class
  • public members are accessible by any function
  • protected members are accessible only in base and derived classes
Inheritance in C++

• Derived classes can add new data members and member functions
  – methods with the same prototype (name as well as number and types of arguments) in the derived class override base class methods
  – distinct from overloading – same function name but different set of parameters
• An instance of a derived class is considered to also be an instance of the base class
  – can be used anywhere an instance of the base class can be used
• An instance of a derived class can invoke public methods of the base class
#include "Sphere.h"

enum Color {RED, BLUE, GREEN, YELLOW};

class ColoredSphere: public Sphere
{
    public:
    ...
    Color getColor() const;
    ...

    private:
    Color c;
}  // end ColoredSphere
C++ Namespaces

• Mechanism for logically grouping declarations and definitions into one declarative region

• The contents of the namespace can be accessed by code inside or outside the namespace
  – use the scope resolution operator (::) to access elements from outside the namespace
  – alternatively, the using declaration allows the names of the elements to be used directly
C++ Namespaces

• Creating a namespace

```cpp
namespace smallNamespace
{
    int count = 0;
    void abc();
} // end smallNamespace
```

• Using a namespace

```cpp
using namespace smallNamespace;
count += 1;
abc();
```
C++ Exceptions

• Mechanism for handling errors at runtime
  – pre-defined as well as user-defined
  – default action is often to kill the program
• A function can indicate that an error has occurred by throwing an exception
• Code that deals with the exception is said to handle it
  – uses a try block and catch blocks
C++ Exceptions

• Place a statement that might throw an exception within a try block
  try { statement(s); }  

• Write a catch block for each type of exception handled
  – order is not important
  catch(ExceptionClass identifier) {
    statement(s);
  }
  catch(ExceptionClass identifier2) {
    statement(s);
  }
C++ Exceptions

- When a statement in a try block causes an exception
  - rest of try block is ignored
    - destructors of objects local to the block are called
  - control passes to catch block corresponding to the exception
  - after a catch block executes, control passes to statement after last catch block associated with the try block
  - if a catch block for the exception is not found, the program typically aborts

see C3-exceptions.cpp
C++ Exceptions

• Throwing exceptions
  – A throw statement throws an exception
  – Methods that throw an exception have a throw clause
    ```
    void myMethod(int x) throw(MyException) {
        if (. . .)
            throw MyException("MyException: ...");
        . . .
    } // end myMethod
    ```

• You can use an exception class in the C++ Standard Library or define your own
List Implemented Using Exceptions

• We define two exception classes

```
#include <stdexcept>
#include <string>
using namespace std;

class ListIndexOutOfRangeException :
    public out_of_range
{
    public:
        ListIndexOutOfRangeException(const string & message = "")
        : out_of_range(message.c_str())
        {}
}; // end ListException
```
List Implemented Using Exceptions

```
#include <stdexcept>
#include <string>
using namespace std;
class ListException : public logic_error
{
 public:
   ListException(const string & message = "")
   : logic_error(message.c_str())
   {}
}; // end ListException
```
List Implemented Using Exceptions

/** @file ListAexcept.h */
#include "ListException.h"
#include "ListIndexOutOfRangeException.h"
 . . .
class List
{
public:
 . . .
    void insert(int index,
                    const ListItemType& newItem)
        throw(ListIndexOutOfRangeException,
                ListException);
 . . .
} // end List
/** @file ListAexcept.cpp */

void List::insert(int index,
                  const ListItemType& newItem)
    throw(ListIndexOutOfRangeException,
          ListException);
{
    if (size > MAX_LIST)
        throw ListException("ListException: "+
                             "List full on insert");
    ...
} // end insert
Summary

• Data abstraction controls the interaction between a program and its data structures
• Abstract data type (ADT): a set of data-management operations together with the data values upon which they operate
• Define an ADT fully before making any decisions about an implementation
• C++ classes used to implement ADT
  – encapsulates both data and operations
Summary

• Members of a class are private by default
  – data members are typically private
  – public methods can be provided to access them
• Namespace: a mechanism to group classes, functions, variables, types, and constants
• You can throw an exception if you detect an error during program execution
  – use try and catch blocks to handle exceptions