Chapter 3:
Data Abstraction: The Walls

Data Abstraction & Problem Solving with C++
Fifth Edition
by Frank M. Carrano

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 members in a class are private
    - But you can specify them as public
  - Encapsulation hides implementation details

C++ Classes

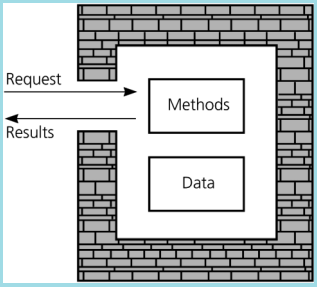

Figure 3-10
An object's data and methods are encapsulated

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: The header file

```
/** @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
```


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

```
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
- 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

C++ Classes: The implementation file

```
/** @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
```


C++ Classes: The implementation file

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

- The constructor could call `setRadius`

C++ Classes: The implementation file

```
double Sphere::getRadius() const
{
   return theRadius;
}  // end getRadius
. . .

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


C++ Classes: Using the class `Sphere`

```
#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++

- A derived class or subclass inherits any of the publicly defined methods or data members of a base class or superclass

```
#include "Sphere.h"
enum Color {RED, BLUE, GREEN, YELLOW};
class ColoredSphere: public Sphere
{
public:
…
  Color getColor() const;
…
private:
   Color c;
}  // end ColoredSphere
```


Inheritance in C++

- 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

C++ Namespaces

- A mechanism for logically grouping declarations and definitions into a common declarative region

```
namespace myNamespace
{
 // Declarations . . .
} //end myNamespace
```


C++ Namespaces

- 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

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

- Using a namespace

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


C++ Namespaces

- Items declared in the C++ Standard Library are declared in the *std* namespace
- You *include* files for several functions declared in the *std* namespace
  - To include input and output functions from the C++ library, write

```
#include <iostream>
using namespace std;
```

An Array-Based ADT List

- Both an array and a list identify their items by number
  - Using an array to represent a list is a natural choice
  - Store a list's items in an array `items`
- Distinguish between the list's length and the array's size
  - Keep track of the list's length

An Array-Based ADT List

- Header file

```
/** @file ListA.h */
const int MAX_LIST = maximum-size-of-list;
typedef desired-type-of-list-item ListItemType;
class List
{
public:
   . . .
private:
   ListItemType items[MAX_LIST];
   int          size;
}  // end List
```


An Array-Based ADT List

- A list's k^{th} item is stored in `items[k-1]`

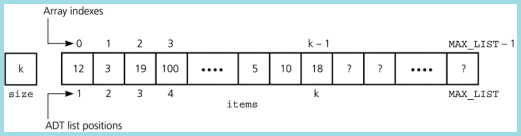

Figure 3-11 An array-based implementation of the ADT list

An Array-Based ADT List

- To insert an item, make room in the array

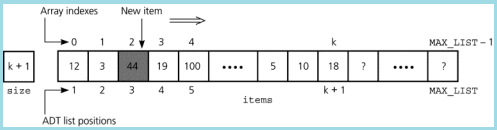

Figure 3-12 Shifting items for insertion at position 3

An Array-Based ADT List

- To delete an item, remove gap in array

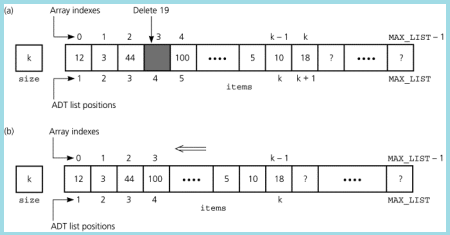

Figure 3-13 (a) Deletion causes a gap; (b) fill gap by shifting

C++ Exceptions

- Exception
  - A mechanism for handling an error during execution
  - A function can indicate that an error has occurred by throwing an exception
  - The code that deals with the exception is said to handle it
    - Uses a `try` block and `catch` blocks

C++ Exceptions

- `try` block
  - Place a statement that might throw an exception within a `try` block

```
try
{
   statement(s);
}
```


C++ Exceptions

- `catch` block
  - Deals with an exception

```
catch (ExceptionClass identifier)
{
   statement(s);
}
```

- Write a `catch` block for each type of exception handled

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

C++ Exceptions

- Throwing exceptions
  - A `throw` statement throws an exception

```
throw ExceptionClass (stringArgument) ;
```

  - 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

An ADT List Implementation 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
```


An ADT List Implementation 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
```

An ADT List Implementation 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
```