Chapter 13

Introduction to Classes

Procedural and Object-Oriented Programming

- Procedural programming focuses on the process/actions that occur in a program
- Object-Oriented programming is based on the data and functions that operate on it
  - ADTs (Abstract Data Types) represent the data and its function
Problems with Procedural Programming

- If the data structures change, many functions must also be changed
- Programs based on complex function hierarchies are:
  - Difficult to understand and maintain
  - Difficult to modify and extend
  - Easy to break

Rectangle Object

<table>
<thead>
<tr>
<th>Member Variables</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>double width;</td>
<td>double length;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Member Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>void setWidth( double w )</td>
</tr>
<tr>
<td>{ ... function code ... }</td>
</tr>
<tr>
<td>double setLength( double l )</td>
</tr>
<tr>
<td>{ ... function code ... }</td>
</tr>
<tr>
<td>double getWidth( void )</td>
</tr>
<tr>
<td>{ ... function code ... }</td>
</tr>
<tr>
<td>double getLength( void )</td>
</tr>
<tr>
<td>{ ... function code ... }</td>
</tr>
<tr>
<td>double getArea( void )</td>
</tr>
<tr>
<td>{ ... function code ... }</td>
</tr>
</tbody>
</table>
OOP Terminology

- **class**: similar to a struct (allows bundling of related variables),
  - But variables and functions in the class can have different properties than in a struct
- **object**: an instance of a class
  - This is in the same way that a variable can be an instance of a struct
- **attributes**: members of a class
- **methods** or **behaviors**: member functions of a class

More on Objects

- **data hiding**: restricting access to certain members of an object
- **public interface**: members of an object that are available outside the object
  - This allows the object to provide access to some data and functions without sharing its internal detail and design
  - Provides some protection from data corruption
- Objects can be **general-purpose** or **application-specific**
How are Objects Used?

- Although the use of objects is only limited by the programmer’s imagination, they are commonly used to create data types that are either very specific or very general in purpose.
General Purpose Objects

- Creating data types that are improvements on C++’s built-in data types
  - For example, an array object could be created that works like a regular array, but additionally provides bounds checking
- Creating data types that are missing from C++
  - For instance, an object could be designed to process currencies or dates as if they were built-in data types
- Creating objects that perform commonly needed tasks
  - Such as input validation and screen output in a graphical user interface

Application-Specific Objects

- Data types created for a specific application
  - For example, in an inventory program
Introduction to Classes

- Objects are created from a class
- The syntax is:
  ```cpp
class class-name
{
  declaration;
  ...
  declaration;
}; // class-name
```

Class Declaration Example

```cpp
class Rectangle
{
  private:
  double width;
  double length;
  public:
  void setWidth ( double );
  void setLength( double );
  double getWidth ( void ) const;
  double getLength ( void ) const;
  double getArea ( void ) const;
}; // Rectangle
```
Access Specifiers

- Used to control access to members of the class
- `private`: the data/methods can only be accessed/called by functions that are members of the class
- `public`: the data/methods can be accessed/called by functions outside the class

More on Access Specifiers

- Can be listed in any order in a class
- Can appear multiple times in a class
- If not specified, the default is private
Defining an Instance of a Class

- Class objects must be defined after the class is declared
- Defining a class object is called the instantiation of a class
- For example:
  Rectangle box;

Defining Member Functions

- Class member functions are defined similarly to regular functions

```cpp
void Rectangle::setWidth ( double w )
{ width = w;
} // Rectangle::setWidth
```
Set vs. Get Functions

- **Set function**: function that stores a value in a private member variable
- **Get function**: function that retrieves a value from a private member variable
- **Common class design practice**:
  - Make all member variables private
  - Provide public set and get functions for each member variable

---

Rectangle – Program

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

class Rectangle
{
    private:
        double width;
        double length;
    public:
        void setWidth ( double );
        void setLength ( double );
        double getWidth ( void ) const;
        double getLength ( void ) const;
        double getArea ( void ) const;
}; // Rectangle
```

Specifies the function will not change any data stored in the calling object
void Rectangle::setWidth( double w )
{ width = w;
} // setWidth

void Rectangle::setLength( double l )
{ length = l;
} // setLength

double Rectangle::getWidth ( void ) const
{ return width;
} // getWidth

double Rectangle::getLength ( void ) const
{ return length;
} // getLength

double Rectangle::getArea ( void ) const
{ return width * length;
} // getArea
int main ( void )
{
    Rectangle box;
    double rectWidth, rectLength;
    cout << "This program will calculate the area of a\n";
    cout << "rectangle. What is the width? ";
    cin >> rectWidth;
    cout << "What is the length? ";
    cin >> rectLength;
    box.setWidth( rectWidth );
    box.setLength( rectLength );
    cout << "Here is the rectangle's data:\n";
    cout << "Width: " << box.getWidth() << endl;
    cout << "Length: " << box.getLength() << endl;
    cout << "Area: " << box.getArea() << endl;
    return 0;
} // main

Rectangle – Program (cont)

Rectangle – Output

This program will calculate the area of a rectangle. What is the width? 10 [Enter]
What is the length? 5 [Enter]
Here is the rectangle's data:
width: 10
length: 5
area: 50

Rectangle – Program (cont)
Accessing an Object’s Members

- For the previous example, to access a member function you would use the statements:
  
  ```
  area = box.getArea();
  box.setLength(10);
  cout << box.getArea();
  ```

Some Design Considerations

- Usually class declarations are stored in their own header file (.h file)
- Member function definitions are stored in their own .cpp file
- The `#ifndef` directive allows a program to be conditionally compiled
  - This prevents a header file from accidentally being included more than once
Rectangle.h

#ifndef RECTANGLE_H
#define RECTANGLE_H

class Rectangle
{
    private:
        double width;
        double length;
    public:
        void setWidth ( double );
        void setLength ( double );
        double getWidth ( void ) const;
        double getLength ( void ) const;
        double getArea ( void ) const;
    }; // Rectangle

#endif

Rectangle.cpp

#include <iostream>
using namespace std;
#include "Rectangle.h"

void Rectangle::setWidth ( double w )
{
    width = w;
} // setWidth

void Rectangle::setLength ( double l )
{
    length = l;
} // setLength
double Rectangle::getWidth ( void ) const
{ return width;
} // getWidth

double Rectangle::getLength ( void ) const
{ return length;
} // getLength

double Rectangle::getArea ( void ) const
{ return width * length;
} // getArea

#include <iostream>
#include "Rectangle.h"
using namespace std;

int main ( void )
{
    Rectangle box;
    double wide, long;

    cout << "This program calculates the area of a\n";
    cout << "rectangle. What is the width? ";
    cin >> wide;
    cout << "What is the length? ";
    cin >> long;
RectMain.cpp (cont)

```cpp
box.setWidth( wide );
box.setLength( long );
cout << "Here rectangle's data:\n";
cout << "width: " << box.getWidth() << endl;
cout << "length: " << box.getLength() << endl;
cout << "area: " << box.getArea() << endl;
return 0;
}
// main
```

Putting the Files Together

- Need to compile:
  - Rectangle.cpp
  - RectMain.cpp
- The object files
  - Rectangle.obj and
  - RectMain.obj
  are linked together.
- Then execute:
  - RectMain.exe
**Input/Output with Objects**

- Notice that the Rectangle example has no cin or cout
- This is so anyone who writes a program that uses the Rectangle class will not be “locked into” the way the class performs input or output
- Unless a class is specifically designed to perform I/O, operations like user input and output are best left to the person designing the application

---

**Pointers to Objects**

```cpp
Rectangle box;
Rectangle *boxPtr;
boxPtr = &box;
boxPtr->setWidth( 15.0 );
cout << boxPtr->getLength( );
```
Using Private Member Functions

- A private member function may only be called from a function that is a member of the same object.
- It is used for internal processing by the class, not for use outside of the class.

Inline Member Functions

- Member functions can be defined:
  - `inline`: in the class declaration
  - After the class declaration
- Inline is appropriate for short function bodies:
  ```cpp
double getWidth ( void ) { return width; }
```
Tradeoffs – Inline vs. Regular

- Regular functions
  - When called, compiler stores return address of call, allocates memory for local variables, etc.

- Inline function
  - Code is copied into program in place of the call
  - Creates a larger executable program, but there is no function call overhead
  - Faster execution

---

Inline Member Functions – Program

```cpp
#ifndef RECTANGLE_H
#define RECTANGLE_H

class Rectangle
{
    private:
        double width;
        double length;
    public:
        void setWidth ( double w ) { width = w; }
        void setLength ( double l ) { length = l; }
        float getWidth ( void ) const { return width; }
        float getLength ( void ) const { return length; }
        float getArea ( void ) const { return width * length; }
}; // Rectangle
#endif
```
Constructors

- A constructor is a member function that is automatically called when a class object is instantiated (created)
  - Constructors have the same name as the class
  - Constructors must be declared publicly
  - Constructors have no return type

Constructor – Program

#include <iostream>
using namespace std;

class Demo
{
public:
    Demo( void ); // Constructor
}; // Demo

Demo::Demo ( void )
{ cout << "Welcome to the Demo constructor!\n"; }
// Demo constructor

int main ( void )
{ Demo demoObj;
    cout << "This program demonstrates an object\n";
    cout << "with a constructor.\n";
    return 0;
}
// main
Constructor – Output

Welcome to the Demo constructor. This program demonstrates an object with a constructor.

Destructors

- A destructor is a member function that is automatically called when an object is destroyed.
  - Destructors have the same name as the class, preceded by a tilde character (~Rectangle)
  - In the same way that a constructor is called when the object is created, the destructor is automatically called when the object is destroyed
  - In the same way that a constructor sets things up when an object is created, a destructor performs shutdown procedures when an object is destroyed
  - For example, release memory allocated dynamically
Destructors – Program

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

class Demo
c    public:
        Demo ( void ) { cout << "Welcome to the constructor!\n"; }
        ~Demo ( void ) { cout << "The destructor is now running.\n"; }
    ); // Demo

int main ( void )
    { Demo demoObj;
        cout << "This program demonstrates an object \n";
        cout << "with a constructor and destructor.\n";
        return 0;
    } // main
```

Destructors – Output

Welcome to the constructor!
This program demonstrates an object with a constructor and destructor.
The destructor is now running.
Constructor Arguments

- When a constructor does not have any arguments, it is called an object’s **default constructor**
- Like regular functions, constructors may accept arguments, have default arguments, be declared inline, and be overloaded

Constructors with Arguments – Program

```cpp
#include <iostream>
#include <cstring>
using namespace std;
const int CHARSIZE = 51;
class InvItem
{
private:
    char *desc;
    int units;
public:
    InvItem ( void ) { desc = new char [CHARSIZE];
        strcpy( desc, ""); units = 0; }
    InvItem ( char *dscr, int un )
    { desc = new char[CHARSIZE];
        strcpy( desc, dscr ); units = un; }
    ~InvItem ( void ) { delete desc; }
    char *getDesc ( void ) { return desc; }
    int getUnits ( void ) { return units; }
}; // InvItem
```
Constructors with Arguments – Program

```c++
int main( void )
{
    InvItem stock ( "Wrench", 20 );
    InvItem stock1;

    cout << "Item Description: " << stock.getDesc() << " Units on hand: " << stock.getUnits() << endl;
    cout << "Item Description: " << stock1.getDesc() << " Units on hand: " << stock1.getUnits() << endl;
    return 0;
} // main
```

Constructors with Arguments – Output

```
Item Description: Wrench
Units on hand: 20
Item Description:
Units on hand: 0
```
Input Validation Objects

- Object can be designed to validate user input:
  - Acceptable menu choice
  - Test score in range of valid scores
  - etc.

ChRange.h

```cpp
#ifndef CHRANGE_H
#define CHRANGE_H

class ChRange
{
  private:
    char input;
    char lower;
    char upper;
    char *errMsg;
}
#endif
```


ChRange.h

public:
    ChRange( char, char, const char * );
~ChRange( ) { delete errMsg; }
void setLower( char ch ) { lower = ch; }
void setUpper( char ch ) { upper = ch; }
char getLower( void ) { return lower; }
char getUpper( void ) { return upper; }
char getChar ( void );
}; // ChRange
#endif

ChRange.cpp

#include <iostream>
#include <cstring>
#include <cctype>
#include "ChRange.h"
using namespace std;

ChRange::ChRange ( char low, char high, const char *str )
{   lower = toupper( low );
    upper = toupper( high );
    errMsg = new char [strlen( str ) + 1];
    strcpy( errMsg, str );
} // ChRange::ChRange
ChRange.cpp

char ChRange::getChar( void )
{
    cin.get( input );
    cin.ignore( );
    input = toupper( input );
    for ( ; input < lower || input > upper; )
    {
        cout << errMsg << endl;
        cin.get( input );
        cin.ignore( );
        input = toupper( input );
    }
    return input;
} // Chrange::getChar

Test ChRange – Program

#include <iostream>
#include "ChRange.h"
using namespace std;

const char *Msg = "\nOnly enter J, K, L, M, or N\n";

int main( void )
{
    ChRange input( 'J', 'N', Msg );
    cout << "Enter any of the characters J, K, L, M, or N\n";
    cout << "Entering N will stop this program.\n";
    while ( input.getChar() != 'N' )
    {
        return 0;
    }
} // main
Overloaded Constructors

- More than one constructor may be defined for a class
- Overloaded constructors in a class must have different parameters lists:
  Rectangle ( );
  Rectangle ( int );
  Rectangle ( int, int );
Member Function Overloading

- Non-constructor member functions can also be overloaded:
  - void setWidth ( );
  - void setWidth ( int );
- Must have unique parameter lists, the same as constructors

Only One Default Constructor and One Destructor

- A class may only have one default constructor (one without any arguments)
- Since a destructor takes no arguments, there can only be one destructor for a class
Arrays of Objects

- You may declare and work with arrays of class objects.
  
  \[ \text{InvItem inventory[40];} \]

- The default constructor for the object is used when the array is defined.

- Must use an initializer list to invoke the constructor that takes arguments:
  
  \[ \text{InvItem inv[ 3 ] = \{ 5, 7, 11 \};} \]

- More complex initialization if constructor takes more than one argument.

---

InvItem.h

```cpp
#ifndef INVITEM_H
#define INVITEM_H

const int DEFSIZE = 51;

class InvItem
{
  private:
    int *description;
    int units;
  public:
    InvItem() { description = new char [DEFSIZE]; }
    InvItem( char *desc )
      { description = new char [strlen(desc) + 1];
      strcpy( description, desc ); }\n    InvItem( char *desc, int u )
      { description = new char [strlen(desc) + 1];
      strcpy( description, desc ); \text{unit} = u; }
};

#endif
```

---
InvItem.h

~InvItem( ) { delete [] description; }
void setDescription( char *d ) { strcpy( description, d ); }
void setUnits( int u ) { units = u; }
const char *getDescription( void ) { return description; }
int getUnits( void ) { return units; }
}; // InvItem
#endif

#include <iostream>
#include <iomanip>
#include "InvItem.h"
using namespace std;

int main( void )
{
    const int NUMITEMS = 5;
    InvItem inventory[NUMITEMS] =
    { InvItem( "Hammer", 12 ), InvItem("Wrench", 17 ),
      InvItem("Pliers", 40 ), InvItem("Ratchet", 14 ),
      InvItem("Screwdriver", 22 ) };
    cout << "Inventory Item \t\tUnits on Hand\n";
    cout << "-----------------------------------------\n";
testInvItem.cpp

```cpp
for ( int i = 0; i < NUMITEMS; i++ )
{ cout << setw( 14 ) << inventory[i].getDescription();
  cout << setw( 18 ) << inventory[i].getUnits() << endl;
} // for
return 0;
} // main
```

An Object-Oriented System Development Primer

- **Procedural programming:**
  - Program made up of procedures
    - Sets of programming statements that perform tasks

- **Object-Oriented programming:**
  - Program made up of objects
    - Entities that contain data (attributes) and actions (methods)
Benefits of Object-Oriented Programming

- Simplification of software development for graphical (GUI) applications
  - Visual components (menus, text boxes, etc.) modeled as objects
  - Visual components (e.g., windows) may be made up of multiple objects
  - Some objects (e.g., buttons) may have methods associated with them

Benefits of Object-Oriented Programming

- Simplification of software development for non-GUI applications – the problem:
  - Procedural programming enforces separation between code and data
  - Changes in data format require extensive analysis, testing to ensure program functionality
Benefits of Object-Oriented Programming

- Simplifications of software development for non-GUI applications – a solution:
  - OO programming addresses the problem through
    - Encapsulation – combination of data and actions in an object
    - Data hiding – protection of an object's data by its public member functions

Component Reusability

- Component:
  - A software object that performs a well-defined task or that provides a service

- Component Reusability:
  - The ability to use a component in multiple programs without (or with little) modification
Relationships Between Objects

- A program may contain object of different classes
- Object may be related by one of the following:
  - Access – *knows* relationship
  - Ownership – *has a* relationship
  - Inheritance – *is a* relationship

Messages and Polymorphism

- **Message**
  - A request to an object to perform a task
    - Implemented as a member function call
- **Polymorphism**
  - Ability to take may forms
    - Sending the same message to different objects may produce different behaviors
Object Oriented Analysis

- Used to create the logical design of a system
  - What the system is to do
- Usually includes:
  - Examine the problem domain; model the system from with that perspective
  - Identify the objects that exists within the boundaries of that system
  - Identify relationships between the objects
  - Realize that there may be may **right** solutions

Object Oriented Design

- Used to determine how requirements from OO Analysis will be implemented
- Usually includes
  - Determine hierarchical relation between objects
  - Determine object ownership of attributes
  - Implement and test; refine as required