13.1

Procedural and Object-Oriented Programming

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 the functions that operate on it. Objects are instances of ADTs that represent the data and its functions.

Limitations of Procedural Programming

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

Object-Oriented Programming Terminology

- **class**: like 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, in the same way that a variable can be an instance of a struct.

Classes and Objects

- A Class is like a blueprint and objects are like houses built from the blueprint.

Object-Oriented Programming Terminology

- **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 of the object. This allows the object to provide access to some data and functions without sharing its internal details and design, and provides some protection from data corruption

Introduction to Classes

- Objects are created from a class
- **Format**:
  ```c++
  class ClassName
  {
      declaration;
      declaration;
  };
  ```

Class Example

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

Access Specifiers

- Used to control access to members of the class
- **public**: can be accessed by functions outside of the class
- **private**: can only be called by or accessed by functions that are members of the class
Class Example

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

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

Using const With Member Functions

- const appearing after the parentheses in a member function declaration specifies that the function will not change any data in the calling object.

    double getWidth() const;
    double getLength() const;
    double getArea() const;

Defining a Member Function

- When defining a member function:
  - Put prototype in class declaration
  - Define function using class name and scope resolution operator (::)

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

Accessors and Mutators

- Mutator: a member function that stores a value in a private member variable, or changes its value in some way
- Accessor: function that retrieves a value from a private member variable. Accessors do not change an object’s data, so they should be marked const.

13.3

Defining an Instance of a Class
Defining an Instance of a Class

- An object is an instance of a class
- Defined like structure variables:
  
  ```
  Rectangle r;
  ```
- Access members using dot operator:
  
  ```
  r.setWidth(5.2);
  cout << r.getWidth();
  ```
- Compiler error if attempt to access private member using dot operator.

Avoiding Stale Data

- Some data is the result of a calculation.
- In the Rectangle class the area of a rectangle is calculated.
  
  ```
  area = length * width
  ```
- If we were to use an area variable here in the Rectangle class, its value would be dependent on the length and the width.
- If we change length or width without updating area, then area would become stale.
- To avoid stale data, it is best to calculate the value of that data within a member function rather than store it in a variable.
**Pointer to an Object**

- Can define a pointer to an object:
  ```
  Rectangle *rPtr = nullptr;
  ```

- Can access public members via pointer:
  ```
  rPtr = &otherRectangle;
  rPtr->setLength(12.5);
  cout << rPtr->getLength() << endl;
  ```

**Dynamically Allocating an Object**

- We can also use a pointer to dynamically allocate an object:
  ```
  // Define a Rectangle pointer.
  Rectangle *rectPtr = nullptr;
  // Dynamically allocate a rectangle object.
  rectPtr = new Rectangle;
  // Store values in the object's width and length.
  rectPtr->setWidth(10.0);
  rectPtr->setLength(15.0);
  // Delete the object from memory.
  delete rectPtr;
  rectPtr = nullptr;
  ```

**Why Have Private Members?**

- Making data members private provides data protection.

- Data can be accessed only through public functions.

- Public functions define the class's public interface.

**Separating Specification from Implementation**

Code outside the class must use the class's public member functions to interact with the object.
Separating Specification from Implementation

- Place class declaration in a header file that serves as the class specification file. Name the file ClassName.h, for example, Rectangle.h.
- Place member function definitions in ClassName.cpp, for example, Rectangle.cpp. File should #include the class specification file.
- Programs that use the class must #include the class specification file, and be compiled and linked with the member function definitions.

Inline Member Functions

- Member functions can be defined
  - inline: in class declaration
  - after the class declaration
- Inline appropriate for short function bodies:
  ```
  int getWidth() const
  { return width; }
  ```

Tradeoffs – Inline vs. Regular Member Functions

- Regular functions – when called, compiler stores return address of call, allocates memory for local variables, etc.
- Code for an inline function is copied into program in place of call – larger executable program, but no function call overhead, hence faster execution.
Constructors

- Member function that is automatically called when an object is created
- Purpose is to construct an object
- Constructor function name is class name
- Has no return type

Contents of Rectangle.cpp (Version 3)

```cpp
// Implementation file for the Rectangle class.
// This version has a constructor.
#include <iostream> // Needed for cout
#include <cstdlib> // Needed for exit function
using namespace std;

// Constructor initializes width and length to 0.0.
Rectangle::Rectangle() {
    width = 0.0;
    length = 0.0;
}
```

Default Constructors

- A default constructor is a constructor that takes no arguments.
- If you write a class with no constructor at all, C++ will write a default constructor for you, one that does nothing.
- A simple instantiation of a class (with no arguments) calls the default constructor:

```
Rectangle r;
```
13.8
Passing Arguments to Constructors

To create a constructor that takes arguments:
- Indicate parameters in prototype:
  `Rectangle(double, double);`
- Use parameters in the definition:
  ```cpp
  Rectangle::Rectangle(double w, double len)
  {
    width = w;
    length = len;
  }
  ```

You can pass arguments to the constructor when you create an object:
```
Rectangle r(10, 5);
```  

More About Default Constructors

If all of a constructor's parameters have default arguments, then it is a default constructor. For example:
```
Rectangle(double = 0, double = 0);
```
Creating an object and passing no arguments will cause this constructor to execute:
```
Rectangle r;
```  

Classes with No Default Constructor

- When all of a class's constructors require arguments, then the class has NO default constructor.
- When this is the case, you must pass the required arguments to the constructor when creating an object.

13.9
Destructors
Destructors

- Member function automatically called when an object is destroyed
- Destructor name is ~classname, e.g., ~Rectangle
- Has no return type; takes no arguments
- Only one destructor per class, i.e., it cannot be overloaded
- If constructor allocates dynamic memory, destructor should release it

Contents of InventoryItem.h Version 1

```cpp
public:
1. // Constructor.
2. InventoryItem() throw() : itemID(0) {}
3. // Initialize just enough memory for the InventoryItem.
4. ~InventoryItem() { delete [] description; } // Deallocate memory.
5. // Design values to cost and units.
6. void setCost(float c) { cost = c; }
7. void setUnits(int n) { units = n; }
8. // Destructor.
9. ~InventoryItem() { delete [] description; } // Deallocate memory.
10. void getCost(float &c) { c = cost; }
11. void getUnits(int &n) { n = units; }
```

Program 13.12

```cpp
1. // This program demonstrates a class with a destructor.
2. #include <iostream>
3. #include "InventoryItem.h"
4. using namespace std;
5. int main()
6. { 
7. InventoryItem item;
8. // Define a InventoryItem with the following data.
9. item.setCost(555.255);
10. item.setUnits(1);
11. // Display the object's data.
12. cout << "Cost: " << item.getCost() << endl;
13. cout << "Units: " << item.getUnits() << endl;
14. return 0;
15. }
```

Contents of InventoryItem.h (Version 1)

```cpp
1. // Specification file for the InventoryItem class.
2. #ifndef INVENTORYITEM_H
3. #define INVENTORYITEM_H
4. #include <cstring> // Needed for strlen and strcpy
5. // InventoryItem class declaration.
6. class InventoryItem
7. { 
8. private:
9. char *description; // The item description
10. double cost; // The item cost
11. int units; // Number of units on hand
```

Constructors, Destructors, and Dynamically Allocated Objects

- When an object is dynamically allocated with the new operator, its constructor executes:
  ```cpp
  Rectangle *r = new Rectangle(10, 20);
  ```
- When the object is destroyed, its destructor executes:
  ```cpp
  delete r;
  ```

13.10

OverloadingConstructors
Overloading Constructors

- A class can have more than one constructor.
- Overloaded constructors in a class must have different parameter lists:
  
  Rectangle();
  Rectangle(double);
  Rectangle(double, double);

Overloaded constructors in a class must have different parameter lists:

- Rectangle();
- Rectangle(double);
- Rectangle(double, double);

Only One Default Constructor and One Destructor

- Do not provide more than one default constructor for a class: one that takes no arguments and one that has default arguments for all parameters.

  Square();
  Square(int = 0);  // will not compile

- Since a destructor takes no arguments, there can only be one destructor for a class.

Member Function Overloading

- Non-constructor member functions can also be overloaded:
  
  void setCost(double);
  void setCost(char *);

- Must have unique parameter lists as for constructors.

3.11

Using Private Member Functions
Using Private Member Functions

- A private member function can only be called by another member function.
- It is used for internal processing by the class, not for use outside of the class.
- See the createDescription function in ContactInfo.h (Version 2).

Arrays of Objects

- Objects can be the elements of an array:
  
  ```
  InventoryItem inventory[40];
  
  ```
- Default constructor for object is used when array is defined.

Arrays of Objects

- Must use initializer list to invoke constructor that takes arguments:
  
  ```
  InventoryItem inventory[3] = {
    "Hammer", "Wrench", "Pliers"
  };
  
  ```

Arrays of Objects

- If the constructor requires more than one argument, the initializer must take the form of a function call:
  
  ```
  InventoryItem inventory[3] = {
    InventoryItem("Hammer", 6.95, 14),
    InventoryItem("Wrench", 6.75, 20),
    InventoryItem("Pliers", 3.75, 10)
  };
  ```

Arrays of Objects

- It isn't necessary to call the same constructor for each object in an array:
  
  ```
  InventoryItem inventory[3] = {
    "Hammer",
    InventoryItem("Wrench", 6.75, 20),
    "Pliers"
  };
  ```
Accessing Objects in an Array

- Objects in an array are referenced using subscripts.
- Member functions are referenced using dot notation:

```cpp
inventory[2].setUnits(30);
std::cout << inventory[2].getUnits();
```

Program 13-14 (Continued)

```cpp
for (int i = 0; i < NUM_ITEMS; i++)
{
    // Set cost for each item
    cost << setprecision(2) << inventory[i].getCost();
    // Set units on hand for each item
    units << setprecision(2) << inventory[i].getUnits();
}
```

The Unified Modeling Language

- **UML** stands for *Unified Modeling Language*.
- The UML provides a set of standard diagrams for graphically depicting object-oriented systems.

**UML Class Diagram**

- A UML diagram for a class has three main sections.

- Class name goes here
  - Member variables are listed here
  - Member functions are listed here
Example: A Rectangle Class

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

UML Access Specification Notation

- In UML you indicate a private member with a minus (-) and a public member with a plus (+).

UML Data Type Notation

- To indicate the data type of a member variable, place a colon followed by the name of the data type after the name of the variable.

```
- width : double
- length : double
```

UML Parameter Type Notation

- To indicate the data type of a function's parameter variable, place a colon followed by the name of the data type after the name of the variable.

```
+ setWidth(w : double)
```

UML Function Return Type Notation

- To indicate the data type of a function's return value, place a colon followed by the name of the data type after the function's parameter list.

```
+ setWidth(w : double) : void
```
### Showing Constructors and Destructors

No return type listed for constructors or destructors

<table>
<thead>
<tr>
<th>InventoryItem</th>
</tr>
</thead>
<tbody>
<tr>
<td>- description : char*</td>
</tr>
<tr>
<td>- cost : double</td>
</tr>
<tr>
<td>- units : int</td>
</tr>
<tr>
<td>- createDescription(size : int, value : char*) : void</td>
</tr>
</tbody>
</table>

**Constructors**

- InventoryItem() : InventoryItem()
- InventoryItem(desc : char*) : InventoryItem(desc : char*, c : double, u : int) :
- InventoryItem(desc : char*) : void
- setCost(c : double) : void
- setUnits(u : int) : void
- getDescription() : char*
- getCost() : double
- getUnits() : int

**Destructor**