Module 7a

Classes and Data Abstraction
In this chapter, you will:

• Learn about classes
• Learn about private, protected, and public members of a class
• Implement classes
• Examine constructors and destructors
• Learn about the abstract data type (ADT)
Objectives (cont’d.)

• Use classes to implement ADTs
• Learn about information hiding
• Implement information hiding in C++
• Learn about the static members of a class
Classes

• **Object-oriented design (OOD):** a problem solving methodology
• **Objects:** components of a solution
• **Class:** a collection of a fixed number of components
• **Member:** a component of a class
Classes (cont’d.)

- Class definition:
  - Defines a data type; no memory is allocated
  - Don’t forget the semicolon after the closing brace

- Syntax:

```cpp
class classIdentifier
{
    classMembersList
};
```
Classes (cont’d.)

• Class member can be a variable or a function
• If a member of a class is a variable
  – It is declared like any other variable
  – You cannot initialize a variable when you declare it
• If a member of a class is a function
  – Function prototype is listed
  – Function members can (directly) access any member of the class
Classes (cont’d.)

• Three categories of class members:
  – *private* (default)
    • Member cannot be accessed outside the class
  – *public*
    • Member is accessible outside the class
  – *Protected*
    . Will explain this when we come across
Variable (Object) Declaration

- Once defined, you can declare variables of that class type
  ```
  clockType myClock;
  ```
- A class variable is called a class object or class instance

![Diagram showing objects myClock and yourClock](image-url)
Accessing Class Members

• Once an object is declared, it can access the public members of the class

• Syntax:

\[ \text{classObjectName} \text{. memberName} \]

  – The dot (.) is the member access operator

• If an object is declared in the definition of a member function of the class, it can access the public and private members
Built-in Operations on Classes

• Most of C++’s built-in operations do not apply to classes
  – Arithmetic operators cannot be used on class objects unless the operators are overloaded
  – Cannot use relational operators to compare two class objects for equality
  – We can implement them \( \Rightarrow \) operator overloaded

• Built-in operations that are valid for class objects:
  – Member access (.)
  – Assignment (=)
Assignment Operator and Classes

(a) myClock and yourClock before executing
myClock = yourClock;

(b) myClock and yourClock after executing
myClock = yourClock;

FIGURE 10-3 myClock and yourClock before and after executing the statement
myClock = yourClock;
Class Scope

• An object can be automatic or static
  – Automatic: created when the declaration is reached and destroyed when the surrounding block is exited
  – Static: created when the declaration is reached and destroyed when the program terminates

• Object has the same scope as other variables
Class Scope (cont’d.)

- A member of the class is local to the class.
- Can access a class member outside the class by using the class object name and the member access operator (.)
Functions and Classes

• Objects can be passed as parameters to functions and returned as function values

• As parameters to functions
  – Objects can be passed by value or by reference

• If an object is passed by value
  – Contents of data members of the actual parameter are copied into the corresponding data members of the formal parameter
Reference Parameters and Class Objects (Variables)

• Passing by value might require a large amount of storage space and a considerable amount of computer time to copy the value of the actual parameter into the formal parameter

• If a variable is passed by reference
  – The formal parameter receives only the address of the actual parameter
• Pass by reference is an efficient way to pass a variable as a parameter
  – Problem: when passing by reference, the actual parameter changes when formal parameter changes
  – Solution: use \texttt{const} in the formal parameter declaration
Implementation of Member Functions

- Must write the code for functions defined as function prototypes
- Prototypes are left in the class to keep the class smaller and to hide the implementation
- To access identifiers local to the class, use the scope resolution operator ::
Implementation of Member Functions (cont’d.)

(a) myClock before executing
myClock.setTime(3, 48, 52);

(b) myClock after executing
myClock.setTime(3, 48, 52);

FIGURE 10-4  myClock before and after executing the statement myClock.setTime(3, 48, 52);
Implementation of Member Functions (cont’d.)

• Once a class is properly defined and implemented, it can be used in a program
  – A program that uses/manipulates objects of a class is called a client of that class

• When you declare objects of the class clockType, each object has its own copy of the member variables (hr, min, and sec)
  • Called instance variables of the class
  – Every object has its own instance of the data
Accessor and Mutator Functions

• **Accessor function**: member function that only accesses the value(s) of member variable(s)

• **Mutator function**: member function that modifies the value(s) of member variable(s)

• **Constant function**:
  – Member function that cannot modify member variables
  – Use `const` in function heading
Order of public and private Members of a Class

• C++ has no fixed order in which to declare public and private members
• By default, all members of a class are private
• Use the member access specifier public to make a member available for public access
Constructors

• Use constructors to guarantee that member variables of a class are initialized

• Two types of constructors:
  – With parameters
  – Without parameters (default constructor)
  – Name of a constructor = name of the class
  – A constructor has no type
Constructors (cont’d.)

• A class can have more than one constructor
  – Each must have a different formal parameter list
• Constructors execute automatically when a class object enters its scope
• They cannot be called like other functions
• Which constructor executes depends on the types of values passed to the class object when the class object is declared
Invoking a Constructor

- A constructor is automatically executed when a class variable is declared.
- Because a class may have more than one constructor, you can invoke a specific constructor.
Invoking the Default Constructor

- To invoke the default constructor:

```cpp
className classObjectName;
```

- Example:

```cpp
clockType yourClock;
```
Invoking a Constructor with Parameters

• Syntax:

```cpp
className classObjectName(argument1, argument2, ...);
```

• Number and type of arguments should match the formal parameters (in the order given) of one of the constructors
  – Otherwise, C++ uses type conversion and looks for the best match
  – Any ambiguity causes a compile-time error
Constructors and Default Parameters

• A constructor can have default parameters
  – Rules for declaring formal parameters are the same as for declaring default formal parameters in a function
  – Actual parameters are passed according to same rules for functions

• Default constructor: a constructor with no parameters or with all default parameters
Classes and Constructors: A Precaution

• If a class has no constructor(s), C++ provides the default constructor
  – However, object declared is still uninitialized

• If a class includes constructor(s) with parameter(s), but not the default constructor
  – C++ does not provide the default constructor
Arrays of Class Objects (Variables) and Constructors

- If you declare an array of class objects, the class should have the default constructor

```cpp
clockType arrivalTimeEmp[100];
arrivalTimeEmp[0]
arrivalTimeEmp[1]
arrivalTimeEmp[49]
arrivalTimeEmp[98]
arrivalTimeEmp[99]
```

![Array arrivalTimeEmp](image)
Destructors

- Destructors are functions without any type
- The name of a destructor is the character '\~' followed by class name
  - For example:
    \[ \sim\text{clockType}() \];
- A class can have only one destructor
  - The destructor has no parameters
- Destructor automatically executes when the class object goes out of scope
Data Abstract, Classes, and Abstract Data Types

• Abstraction
  – Separating design details from usage
  – Separating the logical properties from the implementation details

• Abstract data type (ADT): data type that separates the logical properties from the implementation details
A struct Versus a class

- By default, members of a `struct` are `public`.
  - Private specifier can be used in a struct to make a member private.
- By default, the members of a `class` are `private`.
- `classes` and `structs` have the same capabilities.
A struct Versus a class (cont’d.)

- In C++, the definition of a `struct` was expanded to include member functions, constructors, and destructors.
- If all member variables of a `class` are `public` and there are no member functions:
  - Use a `struct`
Information Hiding

• **Information hiding**: hiding the details of the operations on the data

• **Interface (header) file**: contains the specification details
  • File extension is .h

• **Implementation file**: contains the implementation details
  • File extension is .cpp

• In header file, include function prototypes and comments that briefly describe the functions
  – Specify preconditions and/or postconditions
Information Hiding (cont’d.)

• Implementation file must include header file via `include` statement

• In `include` statement:
  – User-defined header files are enclosed in double quotes
  – System-provided header files are enclosed between angular brackets
To use an object in a program
- The program must be able to access the implementation

Visual C++, Visual Studio .NET, C++ Builder, Quincy and DevCpp put the editor, compiler, and linker into a package
- One command (build, rebuild, or make) compiles program and links it with the other necessary files
- These systems also manage multiple file programs in the form of a project
Static Members of a Class

- Use the keyword `static` to declare a function or variable of a class as `static`.
- A `public static function` or member of a class can be accessed using the class name and the scope resolution operator.
- `static` member variables of a class exist even if no object of that class type exists.
Static Members of a Class (cont’d.)

• Multiple objects of a class each have their own copy of non-static member variables
• All objects of a class share any static member of the class
Next topic ....
Objectives

• In this chapter, you will
  – Learn about overloading
  – Become aware of the restrictions on operator overloading
  – Examine the pointer this
  – Learn about friend functions
  – Explore the members and nonmembers of a class
  – Overload various operators
Why Operator Overloading Is Needed

• Consider the following statements:

```cpp
clockType myClock(8, 23, 34);
clockType yourClock(4, 5, 30);
```

• Which of the following would you prefer?

```cpp
myClock.printTime();
myClock.incrementSeconds();
if (myClock.equalTime(yourClock))
    .
    .
    .

cout << myClock << endl
myClock++;
if (myClock == yourClock)
    .
    .
    .
```
Why Operator Overloading Is Needed (cont’d.)

- Assignment and member selection are the only built-in operations on classes
  - Other operators cannot be applied directly to class objects
- Operator overloading: extends definition of an operator to work with a user-defined data type
  - C++ allows you to extend the definitions of most of the operators to work with classes
Operator Overloading

• Most existing C++ operators can be overloaded to manipulate class objects
• **Cannot create new operators**
• **Operator function**: overloads an operator
  – Use reserved word `operator` as the function name
Syntax for Operator Functions

• Syntax of an operator function heading:

```cpp
returnType operator operatorSymbol(formal parameter list)
```

- It is a value-returning function
- `operator` is a reserved word

• To overload an operator for a class:
  - Include operator function declaration in the class definition
  - Write the definition of the operator function
Overloading an Operator: Some Restrictions

- Cannot change precedence or associativity
- Default parameters cannot be used
- Cannot change number of parameters
- Cannot create new operators
- Cannot overload: . .* :: ?: sizeof
- How the operator works with built-in types remains the same
  - Can overload for user-defined objects or for a combination of user-defined and built-in objects
• Every object of a class maintains a (hidden) pointer to itself called `this`
• When an object invokes a member function
  – `this` is referenced by the member function
Friend Functions of Classes

• Friend function (of a class): a nonmember function of the class that has access to all the members of the class

• Use the reserved word `friend` in the function prototype in the class definition

```cpp
class classIllusFriend
{
    friend void two(/*parameters*/);
    .
    .
};
```
Definition of a friend Function

• "friend" doesn’t appear in function definition
• When writing the friend function definition
  – The name of the class and the scope resolution operator are not used

```cpp
void two(/*parameters*/) {
    
    
}
```
Overloading Binary Operators

• If \# represents a binary operator (e.g., + or ==) that is to be overloaded for some className
  – It can be overloaded as either a member function of the class or as a friend function
Overloading the Binary Operators as Member Functions

• Function prototype (included in the class definition):

```cpp
returnType operator#(const className&) const;
```

• Function definition:

```cpp
returnType className::operator#
    (const className& otherObject) const
{
    //algorithm to perform the operation

    return value;
}
```
Overloading the Arithmetic or Relational Operators

- Function prototype (included in class definition):

```cpp
friend returnType operator#(const className&, const className&);
```

- Function definition:

```cpp
returnType operator#(const className& firstObject, const className& secondObject)
{
    //algorithm to perform the operation
    return value;
}
```
Overloading the Stream Insertion (<<) and Extraction (>>) Operators

• Consider the expression:
  
  ```
  cout << myRectangle;
  ```
  
  – Leftmost operand is an `ostream` object, not a `rectangleType` object

• Thus, the operator function that overloads `<<` for `rectangleType` must be a `nonmember` function of the class
  
  – Same applies to the function that overloads `>>`
Overloading the Stream Insertion Operator (<<)

• Function prototype:

```cpp
friend ostream& operator<<(ostream&, const className&);
```

• Function definition:

```cpp
ostream& operator<<(ostream& osObject, const className& cObject)
{
    //local declaration, if any
    //Output the members of cObject.
    //osObject << . . .

    //Return the stream object.
    return osObject;
}
```
Overloading the Stream Extraction Operator (>>)

- Function prototype:

```cpp
friend istream& operator>>(istream&, className&);
```

- Function definition:

```cpp
istream& operator>>(istream& isObject, className& cObject) {
    //local declaration, if any
    //Read the data into cObject.
    //isObject >> . . .

    //Return the stream object.
    return isObject;
}
```
Overloading the Assignment Operator (=)

• Function prototype:

```cpp
const className& operator=(const className&);
```

• Function definition:

```cpp
const className& className::operator=
    (const className& rightObject)
{
    //local declaration, if any

    if (this != &rightObject)    //avoid self-assignment
    {
        //algorithm to copy rightObject into this object
    }

    //Return the object assigned.
    return *this;
}
```
Overloading the Increment (++) and Decrement (––) Operators

• General syntax to overload the pre-increment operator ++ as a member function
  – Function prototype:

```cpp
className operator++();
```

– Function definition:

```cpp
className className::operator++()
{
    //increment the value of the object by 1
    return *this;
}
```
Overloading the Increment (++) and Decrement (---) Operators (cont’d.)

• General syntax to overload the pre-increment operator ++ as a nonmember function:
  – Function prototype:

    ```cpp
    friend className operator++(className&);
    ```

  – Function definition:

    ```cpp
    className operator++(className& incObj)
    {
        //increment incObj by 1
        return incObj;
    }
    ```
Overloading the Increment (++) and Decrement (---) Operators (cont’d.)

• General syntax to overload the post-increment operator ++ as a member function:
  – Function prototype:

```cpp
className operator++(int);```

  – Function definition:

```cpp
className className::operator++(int u)
{
  className temp = *this;  // use this pointer to copy
  // the value of the object
  // increment the object

  return temp;  // return the old value of the object
}
```
Overloading the Increment (++) and Decrement (---) Operators (cont’d.)

- Syntax to overload the **post-increment operator** `++` as a nonmember function:
  - Function prototype:
    ```cpp
    friend className operator++(className&, int);
    ```
  - Function definition:
    ```cpp
    className operator++(className& incObj, int u)
    {
        className temp = incObj; //copy incObj into temp

        //increment incObj

        return temp; //return the old value of the object
    }
    ```
An example of ++, -- operators

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

class ClockType {
    friend ostream& operator<< (ostream&, const ClockType&);
    public:
        ClockType ();
        ClockType (int, int, int);
        ClockType& operator++ (); // pre-increment
        ClockType operator++ (int); // post-increment
        ClockType& operator-- () ; // pre-decrement
    private:
        int hour, minute, second;
};
```
ClockType& ClockType::operator++ ()
{
    second++;  
    
    if (second == 60)
    {
        second = 0;
        minute++;
        
        if (minute == 60)
        {
            minute = 0;
            hour++;
            
            if (hour == 24)
            {
                hour = 0;
            }
        }
    }
    
    return *this;
}

ClockType ClockType::operator++ (int n) // post-increment
{
    ClockType temp = *this;

    second++;  
    
    if (second == 60)
    {
        second = 0;
        minute++;
        
        if (minute == 60)
        {
            minute = 0;
            hour++;
            
            if (hour == 24)
            {
                hour = 0;
            }
        }
    }

    return temp;
}
ClockType& ClockType::operator-- ()
{
    second--;  
    if (second == -1)  
    {
        second = 59;  
        minute--;  
        if (minute == -1)  
        {
            minute = 59;  
            hour--;  
            if (hour == -1)  
            {
                hour = 23;  
            }
        }
    }

    return *this;
}
int main ()
{
    ClockType ct1 (8, 9, 34);
    ++ct1; // work on pre-increment
    cout << ct1 << endl;

    ClockType ct2 (23, 59, 59);
    ++ct2;
    cout << ct2 << endl;

    --ct2;
    cout << ct2 << endl;

    ct1++;  
    cout << ct1 << endl;

}
Operator Overloading: Member Versus Nonmember

• Some operators must be overloaded as member functions and some must be overloaded as nonmember (friend) functions
Function Overloading

• **Overloading a function**: several functions with the same name, but different parameters
  – Parameter list determines which function will execute
  – Must provide the definition of each function
Summary

• An operator that has different meanings with different data types is said to be overloaded

• Operator function: a function that overloads an operator
  – operator is a reserved word
  – Operator functions are value-returning

• Operator overloading provides the same concise notation for user-defined data types as for built-in data types
Summary (cont’d.)

• Only existing operators can be overloaded
• The pointer this refers to the object
• A friend function is a nonmember of a class
• If an operator function is a member of a class
  – The leftmost operand of the operator must be a class object (or a reference to a class object) of that operator’s class
The standard **string** class

- It is a class!!
- We can use `=` operator to assign a value to.
- We can use the `+` sign to concatenate two strings.
- The class **string** has a default *constructor* that initializes a string object to the empty string.
- The class string also has a 2\textsuperscript{nd} *constructor* that takes one argument (a standard C string) and so can be a quoted string.

```cpp
string module1;
module1 = "C++ programming";
string module2 ("C++ programming");
string module3 = module1 + " is difficult";
```
```cpp
#include <iostream>
#include <string>

using namespace std;

int main () {
    string str1, str2;
    str1 = "abc";
    str2 = "xyz";

    cout << "String 1 = " << str1 << endl;
    cout << "String 2 = " << str2 << endl;
    cout << "String 1 + String 2 = " << str1 + str2 << endl;
    cout << "String 2 + String 1 = " << str2 + str1 << endl;

    // Compare strings
    bool alpha = (str1 > str2);
    cout << boolalpha << (str1 > str2) << endl;
    if (str1 > str2)
        cout << "String 1 > String 2" << endl;
    else
        cout << "String 1 <= String 2" << endl;
}
```

Define string literal header file

We can assign string literals to str1 and str2

String 1 = abc
String 2 = xyz
String 1 + String 2 = abcxyz
String 2 + String 1 = xyzabc
false
String 1 <= String 2

String concatenation

We can use >== to compare strings

true or false
#include <cstring> 
#include <string> 
#include <iostream> 
using namespace std;  

main () {
    char a_c_string [] = "This is my C string";
    string string_variable;

    string_variable = a_c_string;
    cout << string_variable << endl;

    string_variable = a_c_string; // Illegal
    cout << string_variable << endl;

    string_variable = string_variable + "", funny leh";
    strcpy (a_c_string, string_variable.c_str());
    cout << a_c_string << endl;
    cout << string_variable << endl;
}
I/O with class `string`

- You can use the insertion operator `<<` and `cout` to output `string` objects.

- The extraction operator `>>` and `cin` works the same for string objects as for other data, **but the extraction operator ignores initial white spaces and stops reading when it encounters more white spaces.**
getline function to read string objects

- To read an entire line of input into a variable of type string, we can use `getline` function.

```cpp
string message;
cout << "Enter a message\n";
ggetline (cin, message);
cout << message << endl;
```

```cpp
char a_cstring[80];
cin.getline(a_cstring, 80);
```
Let us now look at workshop 7 question