Chapter 7

INTRODUCTION TO CLASSES

Chapter 7

- **Classes**
- **Information Hiding**
- **Member functions**
- **Dynamic Memory Allocation using *new* and *delete* operators**

Overview

- **Object-oriented programming (OOP) *encapsulates* data (attributes) and functions (behavior) into packages called *classes*.**
- **The data and functions of a class are intimately tied together.**
- **A class is like a blueprint. Out of a blueprint, a builder can build a house. Out of a class, we can create many objects of the same class.**
- **Classes have the property of *information hiding*. Implementation details are hidden within the classes themselves.**

CLASSES

- **In C++ programming, classes are structures that contain variables along with functions for manipulating that data.**
- **The functions and variables defined in a class are referred to as *class members*.**
- **Class variables are referred to as *data members*, while class functions are referred to as *member functions*.**
- **Classes are referred to as user-defined data types because you can work with a class as a single unit, or objects, in the same way you work with variables.**

Class definition

- **The most important feature of C++ programming is class definition with the *class* keyword. You define classes the same way you define structures.**

Example:

```
class Time {
public:
    Time();
    void setTime( int, int, int );
    void printMilitary();
    void printStandard();
private:
     int hour;
     int minute;
      int second;
};
```

Instantiating an object

- **Once the class has been defined, it can be used as a type in object, array and pointer definitions as follows:**

```
Time sunset,          // object of type Times
     ArOfTimes[5],    // array of Times objects
    *ptrTime;         // pointer to a Times objects
```

- **The class name becomes a new type specifier. There may be many objects of a class, just as there may be many variables of a type such as *int*.**
- **The programmer can create new class types as needed.**

INFORMATION HIDING

- **The principle of *information hiding* states that any class members that other programmers do not need to access or know about should be hidden.**

- **Many programmers prefer to make all of their data member "private" in order to *prevent* clients from accidentally assigning the wrong value to a variable or from viewing the internal workings of their programs.**

Access Specifiers

- **Access specifiers control a client's access to data members and member functions. There are four levels of access specifiers: *public*, *private*, *protected*, and *friend*.**

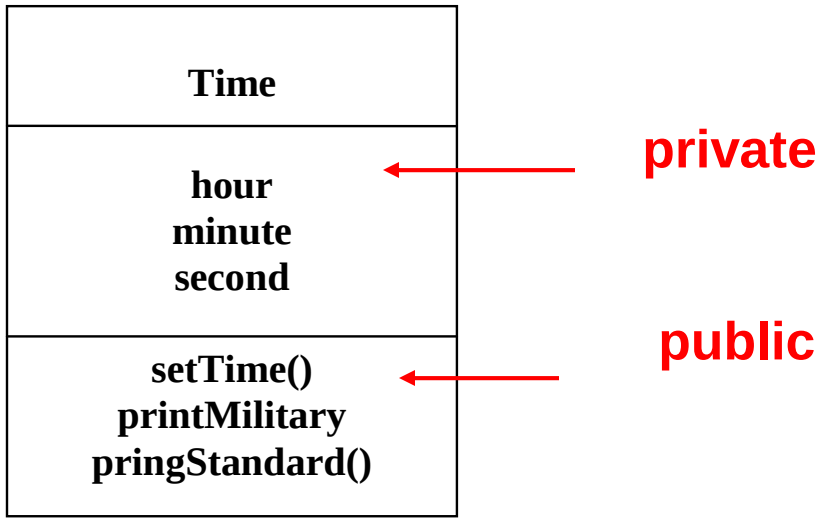
- **The *public access specifier* allows anyone to call a class's function member or to modify a data member.**

- **The *private access specifier* is one of the key elements in information hiding since it prevents clients from calling member functions or accessing data members.**

Note: Class members of both access types are accessible from any of a class's member functions.

Example

```
class Time {
public:
   Time();
   void setTime( int, int, int );
   void printMmilitary();
   void printStandard();
private:
   int hour;
   int minute;
   int second;
};
```



A class' *private* data members are normally not accessible outside the class

Interface and Implementation Files

- **The separation of classes into separate *interface* and *implementation files* is a fundamental software development technique.**

- **The *interface* code refers to the data member and function member declarations inside a class's braces.**

- **The *implementation* code refers to a class's function definitions and any code that assigns values to a class's data members.**

Preventing Multiple Inclusion

- **With large program, you need to ensure that you do not include multiple instances of the same header file.**
- **C++ generates an error if you attempt to compile a program that includes multiple instances of the same header file.**

- **To prevent this kind of error, we use the *#define* preprocessor directive with the *#if* and *#endif* directives in header files.**

- **The *#if* and *#endif* determine which portions of a file to compile depending on the result of a conditional expression.**

- **The syntax for the *#if* and *#endif* preprocessor directives:**

```
#if conditional expression
   statements to compile;
#endif
```

Example:

```
#if !defined(TIME1_H)
#define TIME1_H
class Time {
public:
   Time();
   void setTime( int, int, int );
   void printMilitary();
   void printStandard();
private:
   int hour;
   int minute;
   int second;
};
#endif
```

Note: Common practice when defining a *header file's constant* is to use the header file's name in uppercase letters appended with H.

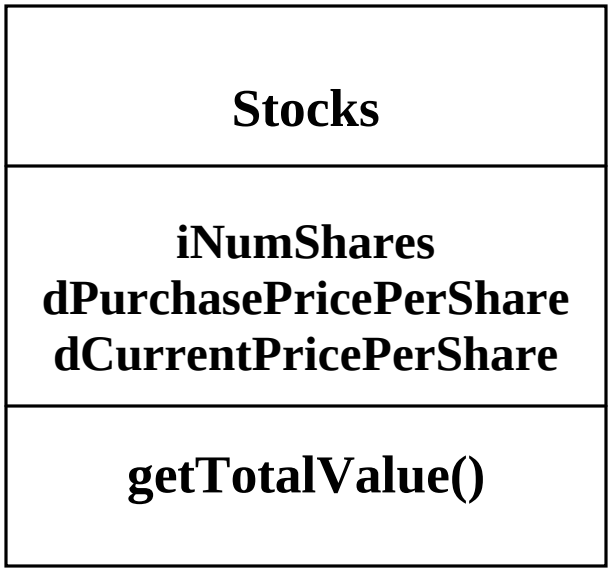
For example, the constant for the *time1.h* header is usually defined as TIME1_H.

MEMBER FUNCTIONS

- **Inline functions**

  Although member functions are usually defined in an implementation file, they can also be defined in an interface file. Functions defined in an interface file are called *inline functions*.

- **Example:**



```
class Stocks {
public:
   double getTotalValue(int iShares, double dCurPrice){
         double dCurrentValue;
    iNumShares = iShares;
    dCurrentPricePerShare = dCurPrice;
    dCurrentValue = iNumShares*dCurrentPricePerShare;
    return dCurrentValue;
    }
private:
    int iNumShares;
   double dPurchasePricePerShare;
   double dCurrentPricePerShare;
};
```

Member functions in Implementation File

Example 7.3.1

```
//stocks.h
#if !defined(STOCKS_H)
#define STOCKS_H
class Stocks{
public:
      double getTotalValue(int iShares, double dCurPrice);
private:
     int iNumShares;
    double dPurchasePricePerShare;
    double dCurrentPricePerShare;
};
#endif
```

```
// stocks.cpp
#include “stocks.h”
#include<iostream.h>
double Stocks::getTotalValue(int iShares, double dCurPrice){
        double dCurrentValue;
    iNumShares = iShares;
    dCurrentPricePerShare = dCurPrice;
    dCurrentValue = iNumShares*dCurrentPricePerShare;
    return dCurrentValue;
}
void main(){
    Stocks stockPick;
    cout << stockPick.getTotalValue(200, 64.25) << endl;
}
```

Output of the above program:

12850

- **The format of member functions included in the implementation section is as follows:**

```
return-type Class-name::functionName(parameter-list)
{
   function body
}
```

- **In order for your class to identify which functions in an implementation section belong to it, you precede the function name in the function header with the class name and the *scope resolution operator* (::).**

Access Functions

- **Access to a class' private data should be controlled by the use of member functions, called *access functions*.**
- **For example, to allow clients to read the value of private data, the class can provide a *get* function.**
- **To enable clients to modify private data, the class can provide a *set* function. A *set* member function can provide data validation capabilities to ensure that the value is set properly.**
- **A *set* function can also translate between the form of data used in the interface and the form used in the implementation.**
- **A *get* function need not expose the data in "raw" format; rather, it can edit data and limit the view of the data the client will see.**

An example of *set* and *get* functions

```
// time1.h
#if !defined(TIME1_H)
#define TIME1_H
class Time {
public:
   Time();                        // constructor
   void setTime( int, int, int ); // set hour, minute, second
   void printMilitary();          // print military time format
   void printStandard();          // print standard time format
private:
   int hour;
   int minute;
   int second;
};
```

```
// time1.cpp
#include “time1.h”
#include <iostream.h>
…..
void Time::setTime( int h, int m, int s ){
   hour = ( h >= 0 && h < 24 ) ? h : 0;
   minute = ( m >= 0 && m < 60 ) ? m : 0;
   second = ( s >= 0 && s < 60 ) ? s : 0;
}
void Time::printMilitary(){
   cout << ( hour < 10 ? "0" : "" ) << hour << ":"
        << ( minute < 10 ? "0" : "" ) << minute;
}
void Time::printStandard(){
   cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
        << ":" << ( minute < 10 ? "0" : "" ) << minute
        << ":" << ( second < 10 ? "0" : "" ) << second
        << ( hour < 12 ? " AM" : " PM" );
}
```

….

Constructor Functions

- **A *constructor function* is a special function with the same name as its class. This function is called automatically when an object from a class is instantiated.**
- **You define and declare constructor functions the same way you define other functions**

- **Example:**

```
class Payroll{
public:
    Payroll( ){    // constructor function
      dFedTax = 0.28;
       dStateTax = 0.05;
   };
private:
    double dFedTax;
    double dStateTax;
}
```

- **You also include just a *function prototype* in the interface file for the constructor function and then create the *function definition* in the implementation file.**

```
Payroll::Payroll( ){    // constructor function
    dFedTax = 0.28;
    dStateTax = 0.05;
};
```

- **Constructor functions do not return values.**

Example 7.3.3

```
#include <iostream.h>
#include <iomanip.h>
// class declaration section
class Date
{
  private:
    int month;
    int day;
    int year;
  public:
    Date(int = 7, int = 4, int = 2001);
// constructor with default values
};
```

```
// implementation section
Date::Date(int mm, int dd, int yyyy)
//constructor
{
  month = mm;
  day = dd;
  year = yyyy;
  cout << "Created a new data object
with data values "
       << month << ", " << day << ", "
       << year << endl;
}
```

```
int main()
{
  Date a;          // declare an object without parameters
  Date b;          // declare an object without parameters
  Date c(4,1,2002);  // declare an object with parameters
  return 0;
}
```

The output of the above program:

Created a new data object with data values 7, 4, 2001
Created a new data object with data values 7, 4, 2001
Created a new data object with data values 4,1, 2001

- ***Default constructor*** **refers to any constructor that does not require any parameters when it is called.**

DYNAMIC MEMORY ALLOCATION WITH OPERATORS new AND delete

- **The *new* and *delete* operators provides a nice means of performing dynamic memory allocation (for any built-in or user-defined type).**

  TypeName *typeNamPtr;

  typeNamePtr = *new* TypeName;

- **The *new* operator automatically creates an object of the proper size, calls the constructor for the object and returns a pointer of the correct type.**

- **To destroy the object and free the space for this object you must use the *delete* operator:**

  ***delete* typeNamePtr;**

- **For built-in data types, we also can use the *new* and *delete* operators.**

- **Example 1:**

```
int *pPointer;
pPointer = new int;
```

- **Example 2:**

```
delete pPointer;
```

- **Example 3: A 10-element integer array can be created and assigned to *arrayPtr* as follows:**

```
int *arrayPtr = new int[10];
```

This array is deleted with the statement

```
delete [] arrayPtr;
```

Stack versus heap

- **A *stack* is a region of memory where applications can store data such as local variables, function calls, and parameters.**
- **The programmers have no control over the stack. C++ automatically handles placing and removing data to and from stack.**
- **The *heap* or free store, is an area of memory that is available to application for storing data whose existence and size are not known until run-time.**
- **Note: When we use *new* operator, we can allocate a piece of memory on the heap and when we use *delete* operator, we can deallocate (free) a piece of memory on the heap.**

Example 7.4.1

```
#include<iostream.h>
void main( )
{
   double* pPrimeInterest = new double;
   *pPrimeInterest = 0.065;
   cout << “The value of pPrimeInterest is: “
          << *pPrimeInterest << endl;
   cout << “The memory address of pPimeInterest is:”
    << &pPrimeInterest << endl;
   delete pPrimeInterest;
   *pPimeInterest = 0.070;
   cout << “The value of pPrimeInterest is: “
         << *pPrimeInterest << endl;
   cout << “The memory address of pPrimeInterest is: “
    << &pPrimeInterest << endl;
}
```

- **The output of the above program:**

The value of pPrimeInterest is: 0.065
The memory address of pPrimeInterest is: 0x0066FD74
The value of pPrimeInterest is: 0.070
The memory address of pPrimeInterest is: 0x0066FD74.

- **<u>Note</u>: You can see that after the *delete* statement executes, the *pPimeInterest* pointer still point to the same memory address!!!**

- **Example 7.4.2**
- **In the following program, we can create some objects of the class *Stocks* on the stack or on the heap and then manipulate them.**

```
#include<iostream.h>
class Stocks{
public:
    int iNumShares;
   double dPurchasePricePerShare;
   double dCurrentPricePerShare;
};
double totalValue(Stocks* pCurStock){
    double dTotalValue;
    dTotalValue = pCurStock->dCurrentPricePerShar*
                                          pCurStock->iNumShares;
    return dTotalValue;
}
```

```
void main( ){
    //allocated on the stack with a pointer to the stack object
    Stocks stockPick;
    Stocks* pStackStock = &stockPick;
    pStackStock->iNumShares = 500;
    pStackStock-> dPurchasePricePerShare = 10.785;
    pStackStock-> dCurrentPricePerShare = 6.5;
    cout << totalValue(pStackStock) << endl;
    //allocated on the heap
    Stocks* pHeapStock = new Stocks;
    pHeapStock->iNumShares = 200;
    pHeapStock-> dPurchasePricePerShare = 32.5;
    pHeapStock-> dCurrentPricePerShare = 48.25;
    cout << totalValue(pHeapStock) << endl;
}
```

The output of the above program:

```
3250
9650
```

Note

- **When declaring and using pointers and references to class objects, follow the same rules as you would when declaring and using pointers and references to structures.**
- **You can use the *indirect member selection operator* (->) to access class members through a pointer to an object either on stack or on the heap.**

- **As we will see, using *new* and *delete* offers other benefits as well. In particular, *new* invokes the constructor and *delete* invokes the class'destructor.**

POINTERS AS CLASS MEMBERS

- **A class can contain any C++ data type. Thus, the inclusion of a pointer variable in a class should not seem surprising.**

Example 7.5.1

```
#include <iostream.h>
#include <string.h>
// class declaration
class Book
{
  private:
    char *title;   // a pointer to a book title
  public:
    Book(char * = NULL);   // constructor with a default value
    void showtitle();      // display the title
};
```

```
// class implementation
Book::Book(char *strng)
{
   title = new char[strlen(strng)+1];  // allocate memory
   strcpy(title,strng);                // store the string
}
void Book::showtitle()
{
   cout << title << endl;
   return;
}
int main()
{
   Book  book1("DOS Primer");   // create 1st title
   Book  book2("A Brief History of Western Civilization");
   book1.showtitle();   // display book1's title
   book2.showtitle();   // display book2's title
  return 0;
}
```

The output of the above program:

DOS Primer
A Brief History of Western Civilization