School of Computing and Informatics

BCS 362 – Generic Programming with C++

Chapter 7 – Classes and Structures

1 Introduction

A class in C++ is a user defined type or data structure declared with keyword class that has data and functions (also called methods) as its members whose access is governed by the three access specifiers private, protected or public (by default access to members of a class is private). The private members are not accessible outside the class; they can be accessed only through methods of the class. The public members form an interface to the class and are accessible outside the class.

1.1 Class declaration

To create a class, we use the keyword class followed by a class-name, followed by a pair of curl brackets and a semicolon. The definition of a class comes in-between the curl brackets. Listing 1 shows basic prototype of a class.

```cpp
class ClassName
{
  //class body
};
```

Listing 1: Basic Class Prototype

1.2 Class interface

A class interface consist of a class public services and class data members declarations. It is important to define a class interface and separate it from class definition. Code outside a class that wishes to use code within a class only needs to know what services the class is providing but not how those services are provided. It is important to separate class interface from class definition so that:-

1. The class is reusable
2. The clients of the class know what member functions the class provide, how to call them and what return type to expect from them
3. The clients do not know how the class member functions are implemented

The class interface is often defined in a header file (.h or .hpp). It is important to use an include guard in the header file to prevent the header code from being included into the same source code file more than once. Since a class can only be defined once, using preprocessor include guard directive prevents multiple-definition errors.

After a class header has been defined, member function definitions are placed in a source code file (.cpp) with the same name as the name of the header file its implementing.

1.3 Class definition

Consider the class definition in Listing 2

```cpp
 1 //prevent multiple inclusions of the header file
 2 #ifndef TIME_H
 3 #define TIME_H
 4 //#if 0
 5 /*comment this code*/
 6 class Time
 7 {
 8 public:
 9   Time();
10   Time(int h, int m, int s);
11   void setSec(int);
12   void setMin(int);
13   void setHour(int);
14 
15   int getHour() const;
16   int getMin() const;
17   int getSec() const;
18 
19   void printTime() const;
20 
21 private:
22   int hour; // 0 - 23
23   int min; // 0 - 59
24   int sec; // 0 - 59
25 };
26 /*code above will be commented*/
27 //#endif
28 #endif
```

Listing 2: Time class definition

The include guard (line 2, 3) prevents the code between #ifndef and #endif from being included if the the name TIME_H has been define. When time.hpp is included the first time, the name TIME_H is not defined. Therefore, the #define directive defines TIME_H and the preprocessor includes Time.hpp code into the .cpp file. If the header is included again, TIME_H is already defined and hence the code between #ifndef and #endif is ignored.

1.4 Class member functions

Class member functions are defined in a file with the same name as the name of the header file in which member functions were declared.

For constant functions, the const keyword must appear in both function prototype and function definition. Listing 3 shows the member function definitions for functions declared in Listing 2

```cpp
#include <iostream>
#include <stdexcept>
#include "Time.hpp"
using namespace std;

Time::Time()
{
  hour = 0;
  min = 0;
  sec = 0;
}
Time::Time(int h, int m, int s)
{
  hour = h;
  min = m;
  sec = s;
}
void Time::setHour(int h)
{
  if ((h >= 0 && h < 24))
  {
    hour = h;
  }
  else
    throw invalid_argument("Hour Invalid");
}

void Time::setMin(int m)
{
  if ((m >= 0 && m < 60))
  {
    min = m;
  }
  else
    throw invalid_argument("Minute Invalid");
}
void Time::setSec(int s)
{
  if ((s >= 0 && s < 24))
  {
    sec = s;
  }
  else
    throw invalid_argument("Second Invalid");
}
void Time::printTime() const
{
  cout << hour << ":" << min << ":" << sec << endl;
}
int Time::getHour() const
{
  return hour;
}
int Time::getMin() const
{
  return min;
}
int Time::getSec() const
{
  return sec;
}
```

Listing 3: Time class member function definition

1.5 Scope resolution operator (::)

You will notice that, in Listing 3, each member function name is preceded by the class name followed by scope resolution operator (::). Time:: tells the compiler that each member function is within the class scope and its name is know to other class members.

Without Time:: preceding each function name, these functions would not be recognized by the compiler as Time member functions. The compiler will consider them as global functions like main().

1.6 Using class Time

Once a class has been declared and its member functions defined, the class can be used as a user defined data type. Listing 4 shows how Time class can be used to define objects and references

```
Time time; //object of type Time. Non constant objects can be used to call both constant and non constant functions
array<Time, 5> arrays; // Array of 5 time objects. can be used to call both constant and non constant functions
Time& ref = time; //Reference to a time object. can be used to call both constant and non constant functions
Time* timePtr = &time; //pointer to a Time object. can be used to call both constant and non constant functions
const Time conTime; //constant object. Can only be used to call constant functions
```

Listing 4: How to use Time class

See the code in Listing 5 to see how we can use the Time class to create objects, references and pointers to objects. Read comments carefully to understand what is going on, copy past all the codes here to your preferred IDE and play around with them to see how the objects behave.

```
#include <iostream>
#include "Time.hpp"
#include <array>
using namespace std;

int main()
{
  Time time; //object of type Time. Non constant objects can be used to call both constant and non constant functions
  array<Time, 5> arrays; // Array of 5 time objects. can be used to call both constant and non constant functions
  Time& ref = time; //Reference to a time object. can be used to call both constant and non constant functions
  Time* timePtr = &time; //pointer to a Time object. can be used to call both constant and non constant functions
  const Time conTime; //constant object. Can only be used to call constant functions

  Time sunset;
  const Time& mem = sunset; // constant reference. Can only be used to call constant functions

  cout << "Object lives in " << &time << endl;
  cout << "Object lives in " << timePtr << endl;
  cout << "Const Object lives in " << &conTime << endl;
  time.getHour(); //use object with dot operator to call functions
  time.setHour(16);
  time.setMin(45);
  time.setSec(52);
  cout << "Time as set ";
  time.printTime();
  cout << "Time as seen by smart pointer ";
  timePtr->printTime(); //we can now use a smart pointer and arrow selection operator like this to access member
     functions
  cout << "Pointer is pointing to " << timePtr << endl;
  cout << "Hour as seen by constant object " << conTime.getHour() << endl; //you can use const object to access const
     member function

  cout << "Objects in my array live in the following memory addresses " << endl;
  for (size_t i = 0; i < arrays.size(); i++)
  {
    cout << i + 1 << "\t" << &arrays.at(i) << endl;
  }
  cout << "Lets use a reference to initialize time " << endl;
  ref.setHour(23); //when using a reference to call functions, use a dot operator
  cout << "What is the time now ";
  ref.printTime();
  //Lets initialize the time for the 3rd Time object in the array of objects arrays
  arrays.at(2).setHour(20);
  arrays.at(2).setMin(57);
  arrays.at(2).setSec(15);
  cout << "What is the time in 3rd object in the array of objects arrays " << endl;
  arrays.at(2).printTime();
```

```

    //It  illegal  to  call  non−constant functions using a constant object
    cout << "sunset object reference " << &sunset << endl;
    cout << "mem object reference " << &mem << endl;
    //sunset and mem point to same object. but
    sunset.setHour(14); //This is  legal  (it  works fine,  no error)
    //mem.setHour(14); //This is illegal. Doesnt work, throws an error. wht?
    cout << "Time as seen by sunset ";
    sunset.printTime();
    cout << "Time as seen by mem ";
    mem.printTime(); //notice that mem can see what sunset modified (updated). Remember they are pointing to same object
    cin.get();
}
```

Listing 5: How to use Time class

1.7 Access functions and utility functions

1. Access functions can read or display data, but not modify the data. Access function can also be used to test truth or falsity of conditions. Such functions are called predicate functions.

2. Utility functions (also called helper functions) is a private member function that supports operations of a class's other member functions. They are declared private because they are not intended for use by class's clients. Can be used to hold a common code that would rather be duplicated in several functions.

1.8 Constructor with default arguments

Just like other functions, a constructor can have default arguments. A constructor is function that has the same name as the name of the class and it does not return a value, not even void. It is used to create objects when classes are instantiated.

A default constructor for class Time can be defined as

```
class  Time
{
    Time(); //default constructor
}
```

Listing 6: Default constructor

A default constructor with default arguments can also be explicitly defined as

```
class  Time
{
    explicit  Time(int = 0, int  = 0, int  = 0); //default  constructor
}
```

Listing 7: Default constructor

Line 3 of Listing 7 shows how to explicitly define a default constructor using the keyword explicit. An explicit default constructor with default arguments for all data members of a class is same as an implicit default constructor shown in Listing 6. A class should have one, and only one default constructor. A compiler will give an error if the constructors in Listing 6 and Listing 7 cannnot be in the same same class because both are default constructors.

1.9 Overloaded constructors

The default constructor in Listing 7 had default argument for each parameter. We could have define that constructor as four overloaded constructors with following prototypes

```
class Time
{
  Time(); // initialize all values to zero
  explicit Time(int); // initialize hour, default min and sec
  Time(int, int, int); // initialize hour, min and sec
  Time(int, int); // initialize hour, min and default sec to 0
};
```

Listing 8: Overloaded constructors

Just as constructors can call other member functions, constructors can call other constructors in the same class. The calling constructor is called a delegating constructor.

This is important when overloaded constructors have common code that could be defined in a private utility function and called by all other constructors.

The first 3 of the 4 Time constructors in Listing 8 can delegate work to one with three int arguments, passing 0 as the default value for the extra parameters. To do so, you use a member initializer with the name of the class as follows

```

  //delegate to Time(int, int, int)
  Time::Time() : Time{0, 0, 0} {}
  //delegate to Time(int, int, int)
  Time::Time(int hour) : Time{hour, 0, 0} {}
  //delegate to Time(int, int, int)
  Time::Time(int hour, int min) : Time{hour, min, 0}{}
```

Listing 9: Constructor delegation

The constructors in Listing 3 can be defined as

```
//definition of default constructor that delegates to Time(int, int, int)
Time::Time() : Time{ 0, 0, 0 }{}

//definition of one parameter constructor that delegates to Time(int, int, int)
Time::Time(int hour) : Time{ hour, 0, 0 }{}

//definition of two parameter constructor that delegates to Time(int, int, int)
Time::Time(int hour, int min) : Time{ hour, min, 0 }{}

Time::Time(int h, int m, int s)
{
  hour = h;
  min = m;
  sec = s;
}
```

Listing 10: Constructor delegation

1.10 Destructors

A destructor is another type of special member function. The name of a destructor for a class is the tilde character ( ) followed by the name of the class. A class destructor is called implicitly when an object is destroyed. However, the destructor itself does not actually release memory, it does termination house keeping, (like closing a file) before object's memory is reclaimed. A class has got exactly one destructor. If you do not explicitly define a destructor, the compiler provides a default empty destructor.

1.11 Default memberwise assignment

We can use the assignment operator (=) to assign an object to another object of the same class. Such assignment is done by memberwise (or copy) assignment.

```
Time t(15, 45, 54);
Time tt(20, 12, 24);
t = tt; //contents of tt are copied to respective location of t
t.print(); //display 20:12:24
tt.print(); //dispaly 20:12:24
t.setHour(18);
t.print(); //display 18:12:24
tt.print(); //display 20:12:24
```

Listing 11: Memberwise assignment

1.12 toString() definition

```
#include <sstream>
//std::string toString() const; //function declared as this in .hpp file
string Time::toString() const
{
  ostringstream output;
  output << hour << ":" << min << ":" << sec;

  return output.str();

}
```

Listing 12: toString() definition