# Module 4: Classes and Objects

## Lesson 1: Introduction to classes

### Learning Outcomes

- ➢ LO 4.1.1 Create classes and implement private, public and protected members of that class.
- ➢ LO 4.1.2 Use function to access the private members of the class.
- ➢ LO 4.1.3 Examine how object constructors and destructors are implemented.

The previous chapter covers grouping of related data together using C++ structures. However, these programming is inclined to procedure oriented approach, since the focus is to break down a task to specific functions that would accomplish it. The problem solving approach of object-oriented design (OOD) is to identify the necessary components (called objects) to complete the given task. The object must be a complete package of both data and function. The combination of both data a function is called a class.

#### Declaring and using a `class` instance

The `class` is a reserved word in C++. It defines a new `class` instance so no memory is allocated yet. The definition would start with a `class` identifier followed by an open curly brace, then the `class` members, then finally terminated by a close curly brace and a semicolon. The syntax and `class` definition example is given below.

**Syntax**

```
class classID
{
   classMembers
}
```

**Example**

```
class point
{
public:
    void setX(double);
    void setY(double);
    double getX();
    double getY();
private:
    double x;
    double y;
};
```

The example shows that the member of the class point are variables and functions. When declaring a variable you must declare it like any other variable without an initialization. On the other hand, when a function is declared, it is done similar to a function prototype. Take note also that all the functions and variables within the class can directly be accessed by a function within the class.

The members of a class can be the following:

- `public` these are members that can be accessed by outside the class.
- `private` these are members that can only be accessed within the class.
- `protected` the accessibility of this members are in between private and public. That is, only a derived (inherited) class can access the protected variables on the base.

#### Implementing the accessor and mutator functions in a `class`

The function(s) that are defined in a class are just function prototypes. For them to work properly a related algorithm must be implemented for those functions.

However, it would make the class definition too lengthy and difficult to comprehend. Moreover, by using function prototypes, function definitions related to that class can be hidden from the user. Information hiding will be discussed later, but for the meanwhile, defining a function from the class is done by using the resolution operator (double colon ::). See the syntax and example:

```
Syntax                     Example
dataType                   void point::setX(double varX)
classID::function          {
{                              x = varX;
   //function              }
implementation             void point::setY(double varY)
}                          {
                               y = varY;
                           }
                           double point::getX()
                           {
                               return x;
                           }
                           double point::getY()
                           {
                               return y;
                           }
```

The member function on the `class point` `void` which is `setX(double)` would initialize a value set by the user for the member variable `x`. This function that would access to the `private` variable in a class is called the ***mutator*** function. On the other hand the function that would only access the value found on the class (i.e `double getX()`) is called the ***accessor*** function.

## Declaring and accessing class members

After the class is defined, together with the appropriate implementation of their member function an object can be declared from it. An object is an instance of a class, think of it as the class being the template of a product while the object is the actual product. The syntax and example is shown below.

| **Syntax** | **Example** |
|---|---|
| `classID objectID` | `point p1;` |

After declaring, different values can be allocated for variable `x` and `y`. Now, accessing the members of the class is done by using the dot (.) operator. Refer to the syntax and example below.

| **Syntax** | **Example** |
|---|---|
| `objectID.classMember` | `p1.setX(5);` |

## Constructors and destructors

When the variable within the class is uninitialized, the output for this variable would be some strange number. This is because C++ does not automatically initialize the variables while they are in a class. So to ensure that they are initialized you must use the constructor. A constructor is characterized by the following.

- It is done by creating a function same as the class name.
- It has no data type, not even void.
- A class may have many constructors of the same name but must have different formal parameter.

- Constructors execute automatically since they have no types.
- When there are many constructors, the one that is called during the creation of the object will be the one to initialize the variables of the object.

Refer to the syntax and example.

| Syntax | Example |
|---|---|
| `classID();` | `point();` |
| … | … |
| `classID::classID()` | `//default constructor` |
| `{` | `point::point()` |
| `//initialize  values` | `{` |
| `here` | `    x=0;` |
| `}` | `    y=0;` |
|  | `}` |

A default constructor is invoked immediately when an object is created, however when customized constructor is made you must specify the value of the formal parameter. Take for example.

**Example**

```
point(double);
…
point p2(5);//initialize only variable x
…
//customized constructor
point::point(double varX)
{
    x = varX;
    y=0;
}
```

Similar to constructors, destructors are functions without any type. These functions are neither value-returning nor void function. Moreover, only one destructor is available per class. It is implemented by placing a tilde character (~), before the name of the class. See the syntax and example below.

| Syntax | Example |
|---|---|
| `~classID();` | `~point();` |
| … | … |
| `classID::~classID()` | `point::~point()` |
| `{` | `{` |
| `}` | `}` |

When the object would go out of scope, the destructor would automatically execute.

**Classes and Functions**

The objects that are created from classes can be passed to a function the same way as primitive variabless are passed. These are the rules that exist between function and classes.

- Objects can be parameters to a function and can also be return values from functions.
- Passing can be done by reference or by value.

- If passed by value a copy of the object is sent to the function.
- If passed by reference only the address of the object is passed. This process is more efficient especially if the object has many member variables.

**Example**

```
void dispXY(point p);…
…
dispXY(p1);
…
void dispXY(point p)
{
    cout<<p.getX();
    cout<<p.getY();
}
```

## Lesson 2: Data Abstraction

### Learning Outcome:

- LO 4.2.1 Make separate files for the class, implementation and user interface.

When using day to day objects such as a pen, there is no need to get concerned with the complexity on how the pen would work. Most of us just want to write by using the pen. So by separating the design of the pen's ballpoint mechanism with the usability of the pen would make the user more focused on actually using the pen. Data abstraction in C++ works on the same manner by separating the logical properties of the class with its implementation details. A logical property such as using the pen is separated from the implementation details such as mechanism of the gravity fed ballpoint.

#### Hiding information

The previous section the definition of `class point`, the implementation for the function of that `class point` and finally the function main to make use of `class point` are all on the same file. This may look a very straightforward implementation but this is not a good practice as the code base would grow in size. Take for instance, if the user has direct access to the function definition and member function it is possible that the user would modify the function according to the user's preference. So, if a team of programmers have direct access to the internal parts of the object, there is no guarantee that a standardized class definition would be followed. Moreover, by hiding the implementation, the burden of having an extra piece of code every time an implementation is made is freed from the user. Lastly, if the implementation code is written, debugged and tested appropriately then it would seem unnecessary to keep on repeating the same process.

Hiding the implementation is usually done by creating a header file that would contain the class definition. For uniformity the `class point` would be used for this example.

**Example**

```
//header point.h
#ifndef POINT_H_INCLUDED
#define POINT_H_INCLUDED
class point
{
public:
    point();
    point(double);
    void setX(double);
    void setY(double);
    double getX();
    double getY();

    ~point();
private:
    double x;
    double y;
};

#endif // POINT_H_INCLUDED
```

Since the implementation of the header would be too long, it is a good practice to create another `cpp` file that would handle implementation details. This file would include the class definition. Take note that when including a user defined header file, double quotation is used instead of angular brackets.

**Example**

```
//pointImp.cpp
#include "point.h"
using namespace std;

void point::setX(double varX)
{
    x = varX;
}
void point::setY(double varY)
{
    y = varY;
}
double point::getX()
{
    return x;
}
double point::getY()
{
    return y;
}
point::point()
{
    x=0;
    y=0;
}
point::point(double varX)
{
    x = varX;
    y=0;
};
point::~point()
{
}
```

It is a good practice that when defining a class, one would have to add the needed comments that describes operations of the function. Moreover, it is recommend to specify whether a statement must be true before a statement is called (precondition) or what is the expected result after that function is called (postcondition).

**Separate compilation**

When making use of the class definition and function implementation, a separate program must be able to access it. The client's program is the file with the main() function. As previously indicated, the file must include the header in which your class is defined. Refer to the example below.

**Example**

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


int main()
{
   //add user's code here
}
```

The Code::Blocks IDE place all the features of an editor, compiler and linker in one package. So to create a multiple-file program you must begin by starting a new project. Out of that project, you can add the necessary headers, implementation program and user's program. Please check the user documentation or web support on how to implement this operation.

## Lesson 3: `class` inheritance

### Learning Outcome:

- LO 4.3.1 Use inheritance and for derived and base classes

Consider the following scenario, you are tasked in creating another version of the `class point` that would be applicable on a three-dimensional space. The main features that are necessary to accomplish this task is already present on the `class point` such as the variables x and y. Does this mean that your new class named point3D would be started from scratch? Well, creating a new class it would result to a time consuming task. That is why a good practice is to extent the existing `class point` so that it would be applicable for the `point3D` definition. This feature in C++ is called inheritance or in other words, allowing you to create new classes from existing classes. The new class is called the derived class while the exiting class is called the base class.

Since inheritance would allow you to take advantage of existing properties of the base class, software complexity would be greatly reduced. Then, the derived class would be a future base class giving better scalability of your code base. The principle of inheritance can also be viewed as an "is a" relationship. For example every 3D point is a point. The syntax and example is shown below.

| Syntax | Example |
|---|---|
| `class classID: accessSpecifier baseClassID`<br>`{`<br><br>`};` | `//point3D.h`<br>`#include "point.h"`<br>`#ifndef POINT3D_H_INCLUDED`<br>`#define POINT3D_H_INCLUDED`<br>`class point3D: public point`<br>`{`<br>`public:` |

```
    point3D();
    void setZ();
    void getZ();
    ~point3D();
private:
    double z;
};
#endif // POINT3D_H_INCLUDED
```

When doing class inheritance remember the following facts.

1. The private members of the base class cannot be accessed directly by the derived class.
2. The public members of the base class can be inherited as public or private members of the derived class.
3. The derived class can have its own data and function definitions.
4. The derived class can redefine the public function of the base class.
5. All member functions and variables of the base class is also members of the derived class. Refer to the #1 for access privileges.

Given that the derived class has the following implementation.

**Example**

```
//from point3DImp.cpp
point3D::point3D()
{
    z=0;
}
void point3D::setZ(double varZ)
{
    z=varZ;
}

double point3D::getZ()
{
    return z;
}

point3D::~point3D()
{
}
```

Refer to the example below in creating an object and using its features in function main.

**Example**

```
#include "point3D.h"
…
    point3D p2;
    p2.setX(3);
    p2.setY(5);
    p2.setZ(7);
```

**Overriding member functions of base class**

Suppose that the base class and the derived class has a member function of the same name, type of parameter, number of parameter and return type. The derived class can override the operation of the base class so that it would be applicable to the derived class. Take for example,

**Example**

```
//from class point
void print();

//from class point3D
void print();

//from pointImp.cpp
#include <iostream>
…
void point::print()
{
    cout<<"x:"<<x<<" y:"<<y;
}

//from point3DImp.cpp
#include <iostream>
…
void point3D::print()
{
    point::print();
    cout<<" z:"<<z;
}

//from function main()
p2.print(); //output: x:3 y:5 z:7
```

The example shows that the both classes (base class and derived class) as having a function of the same name but the derived class override the existing function of the base class. Now the overridden function would display variable z that is exclusive to the derived class. If the function of the derived class has the same name but of different parameters from the base class, this type of operation is called ***overloading***.

**Constructors and destructors from derived class**

When an object from the derived class is created it would inherit all the public constructors from the base class. However since the derived class cannot access the private variables of the base class, the default constructor must execute in order to initialize the variables of the base class. Moreover, since the derived has its own private variables it can also have its own constructors. Typically this constructor would initialize the value for its member variables. Take for example,

**Example**

```
point3D::point3D()
{
    z=0;
}
```

On the other hand, when declaring a customized constructor from the derived class you must be sure to include the public constructor from the base class then integrate it to the customized constructor on the derived class. For example

**Example**

```
//from class point3D
point3D(double varX,double varZ);
…
//from pointImp.cpp
point3D::point3D(double varX,double varZ)
:point(varX)
{
    z = varZ;
}
…
//from function main
point3D p2(4,7);
```

Recall that a destructor is used to deallocate the memory of an object it would go out of scope. So given that a derived class would go out of scope it would first execute its own destructor then afterwards the destructor that is present on the base class. This can otherwise be represented as destructors executing on reverse order.

**Including multiple header files**

When one header would contain one class, it would be possible that there will be many header to be included on a source file. Recall that before a program is compiled the preprocessor must be executed first. So, when a derived class would include the header of the base class, then the same base class header is included on the source file causing a compilation error. This is because, the identifiers that are declared are declared all over again. To avoid multiple inclusion enclose the header operation with the commands below.

| **Syntax** | **Example** |
|---|---|
| `//point3D.h` | `//point3D.h` |
| `#include "point.h"` | `#include "point.h"` |
| `#ifndef ClassID_H_INCLUDED` | `#ifndef` |
| `#define ClassID _H_INCLUDED` | `POINT3D_H_INCLUDED` |
| `//` | `#define` |
| `#endif                   //` | `POINT3D_H_INCLUDED` |
| `POINT3D_H_INCLUDED` | `//header contents here` |
| | `#endif                //` |
| | `POINT3D_H_INCLUDED` |

- `#ifndef POINT3D_H_INCLUDED – if not defined point3D.h`
- `#define POINT3D_H_INCLUDED – define point3D.h`
- `#endif – end definition`

This means that when point3D.h is not defined then it would be defined, otherwise do not define it. All headers that are created using the Code::blocks IDE would automatically have these pieces of code.

**Protected members of the class**

The `public` members of the class can be accessed directly by any function outside the class. While the `private` members can only be accessed within the class. However, there may be instances in which the derived class needs to efficiently access the variables on the base class. So for the base class to give direct access to its variables but restrict its access to outside function, you would use the `protected` member access specifier. Therefore the accessibility of a `protected` class is somewhere in between private and public.

## Inheritance through `private`, `public` or `protected`

Given that you have two classes, `class y` which inherits `class x` then used a public access specifier. Then we can say that all private variables of `class x` are inaccessible by `class y` while all `public` variables of `class x` are directly accessible by `class y`. How about if the access specifier is `private` and `protected`? This section covers the fundamental rules for this operation.

| Member access specifier of class y | public variables and functions on class x | private variables and functions on class x | protected variables and functions on class x |
|---|---|---|---|
| public | public | private | protected |
| private | private | private | private |
| protected | protected | private | protected |

## `class` composition

Composition is defined simply as a "has a" relationship between two objects. That is, an object is made up of another object. For instance, a line has a starting and ending point. By having this statement we can build a `class line` that uses the point as member variables. For example every line has a starting and ending point.

**Example**
```
#include "point.h"
#ifndef LINE_H_INCLUDED
#define LINE_H_INCLUDED
class line
{
public:
    void setStartPt(double,double);
    void setEndPt(double,double);
    point getStartPt();
    point getEndPt();
    double distance();
    double angle();
    line();
    ~line();
private:
    point startPt;
    point endPt;
};
#endif // LINE_H_INCLUDED
```

The implementation of the class is as follows

**Example**
```
#include "line.h"
#include <iostream>
#include <cmath>
using namespace std;

line::line()
```

```
{
    startPt.setX(0);
    startPt.setY(0);
    endPt.setX(0);
    endPt.setY(0);
}
void line::setStartPt(double x, double y)
{
    startPt.setX(x);
    startPt.setY(y);
}

void line::setEndPt(double x, double y)
{
    endPt.setX(x);
    endPt.setY(y);
}

point line::getStartPt()
{
    return startPt;
}
point line::getEndPt()
{
    return endPt;
}

double line::distance()
{
    double x1 = startPt.getX();
    double y1 = startPt.getY();
    double x2 = endPt.getX();
    double y2 = endPt.getY();
    return      sqrt(pow((x2-x1),2)+(pow((y2-
y1),2)));
}

double line::angle()
{
    double x1 = startPt.getX();
    double y1 = startPt.getY();
    double x2 = endPt.getX();
    double y2 = endPt.getY();
   return atan((y2-y1)/(x2-x1))*57.2958;
}
line::~line()
{
}
```

You can use the function that is found on the implementation on function main. An example is shown below.

**Example**

```
 //from function main
    line l;
    l.setStartPt(1,2);
    l.setEndPt(13,8);
```

```
        cout<<l.distance()<<endl;
        cout<<l.angle();
```