Chapter 1

Object Oriented Programming

1.1 OOP

Definition 1.1.1 (**OOP Object Oriented Programming**)
OOP stands for **Object-Oriented Programming**. Object-oriented programming is about *creating objects, which can hold data and functions that work on that data.*

Advantages:

- OOP provides a *clear structure* to programs
- Makes *code easier to maintain, reuse, and debug*
- Helps keep your code DRY (Don't Repeat Yourself)
- Makes it *possible to create full reusable applications with less code and shorter development time*

1.1.1 Classes and Objects

Classes and **objects** are the two main aspects of object-oriented programming.

Definition 1.1.2 (**Class**)
A **class** *defines what an object should look like, and an object is created based on that class.*

Class	Objects
Fruit	Apple, Banana, Mango
Car	Volvo, Audi, Toyota

Table 1.1:

Observation 1.1.1 (**Functions Inside a Class**)
When you create an object from a class, it inherits all the variables and functions defined inside that class.

Also, we say that an object *apple* is an instance of the class *fruit*.

1.1.2 Procedural vs Object-Oriented Programming

We have two programming focuses:

- **Procedural programming** is about writing functions that operate on data.
- **Object-oriented programming (OOP)** is about creating objects that contain both the data and the functions.

1.2 C++ Classes/Objects

C++ is an object-oriented programming language. Everything in C++ is associated with classes and objects, along with its attributes and methods.

Example 1.2.1
For example: in real life, a car is an object. The car has attributes, such as weight and color, and methods, such as drive and brake.

Definition 1.2.1 (**Attributes and Methods**)
Attributes and **methods** are *basically variables and functions that belongs to the class. These are often referred to as class members.*

A class is a user-defined data type that we can use in our program, and it works as an object constructor, or a blueprint for creating objects.

Example 1.2.2 (**Example of a Class**)
To create a Class in C++ we do the following:

```
class MyClass {        // The class
  public:              // Access specifier
    int myNum;         // Attribute (int variable)
    string myString;   // Attribute (string variable)
};
```

Code 1.1: caption

- The `class` keyword is used to create a `class` called `MyClass`.
- The `public` keyword is an **access specifier**, which *specifies that members (attributes and methods) of the class are accessible from outside the class.* You will learn more about access specifiers later.
- Inside the `class`, there is an integer variable `myNum` and a string variable `myString`. When variables are declared within a class, they are called **attributes**.
- At last, *end the class definition with a semicolon* `;`.

1.2.1 Create an Object

In C++, an object is created from a `class`. We have already created the `class` named `MyClass`, so now we can use this to create objects.

To create an object of `MyClass`, specify the class name, followed by the object name.

To access the class attributes (`myNum` and `myString`), use the dot syntax (`.`) on the object:

Example 1.2.3 (**Creation of an Object**)
Create an object called `myObj` and access the attributes:

```
class MyClass {        // The class
  public:              // Access specifier
    int myNum;         // Attribute (int variable)
    string myString;   // Attribute (string variable)
};

```

```cpp
int main() {
  MyClass myObj;  // Create an object of MyClass

  // Access attributes and set values
  myObj.myNum = 15;
  myObj.myString = "Some text";

  // Print attribute values
  cout << myObj.myNum << "\n";
  cout << myObj.myString;
  return 0;
}
```

Code 1.2: caption

Observation 1.2.1 (**Several Objects**)
Also, we can create several objects from a same class.

1.3 Class Methods

Definition 1.3.1 (**Methods**)
Methods are *functions that belongs to the class.*

There are two ways to define functions that belongs to a class:

- Inside class definition
- Outside class definition

1.3.1 Define a Method Inside the Class

In the following example, we define a function inside the class, and we name it `myMethod`.

Idea 1.3.1 (**Note**)
You access methods just like you access attributes; by creating an object of the class and using the dot syntax (.):

Example 1.3.1 (**Method Creation**)
Example of a method inside a class:

```cpp
class MyClass {        // The class
  public:              // Access specifier
    void myMethod() {  // Method/function defined inside the
      class
      cout << "Hello World!";
    }
};

int main() {
  MyClass myObj;     // Create an object of MyClass
  myObj.myMethod();  // Call the method
```

```
  return 0;
}
```

Code 1.3: caption

1.3.2 Define a Method Outside the Class

Sometimes it is *better to declare the method in the class and define it later* (especially in large programs).

This is done by specifiying the name of the class, followed the scope resolution `::` operator, followed by the name of the function:

```
class MyClass {        // The class
  public:              // Access specifier
    void myMethod();   // Method/function declaration
};

// Method/function definition outside the class
void MyClass::myMethod() {
  cout << "Hello World!";
}

int main() {
  MyClass myObj;     // Create an object of MyClass
  myObj.myMethod();  // Call the method
  return 0;
}
```

Code 1.4: caption

1.3.3 Parameters

You can also pass values to methods just like regular functions:

Example 1.3.2 (**Pass Values Through Methods**)
To pass some value to a method we do the following:

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

class Car {
  public:
    int speed(int maxSpeed);
};

int Car::speed(int maxSpeed) {
  return maxSpeed;
}

int main() {
  Car myObj; // Create an object of Car
  cout << myObj.speed(200); // Call the method with an
     argument
  return 0;
```

```
17 }
```

Code 1.5: caption

1.4 Constructors

Definition 1.4.1 (**Constructor**)
A **constructor** is a *special method that is automatically called when an object of a class is created.*
To create a constructor, use the same name as the class, followed by parentheses `()`:

Creation of a constructor:

```
1  class MyClass {       // The class
2    public:             // Access specifier
3      MyClass() {       // Constructor
4        cout << "Hello World!";
5      }
6  };
7
8  int main() {
9    MyClass myObj;     // Create an object of MyClass (this will
         call the constructor)
10   return 0;
11 }
```

Code 1.6: caption

Observation 1.4.1 (**Constructor Rules**)
When *creating a constructor*, we must have in mind the following rules:

- The constructor *has the same name as the class.*
- It *has no return type* (not even void).
- It is *usually declared public.*
- It is *automatically called when an object is created.*

1.4.1 Constructor with Parameters

Definition 1.4.2 (**Constructor with Parameters**)
Constructors *can also take parameters* (just like regular functions), which *can be useful for setting initial values for attributes.*

The following class has `brand`, `model` and `year` attributes, and a constructor with different parameters. Inside the constructor we set the attributes equal to the constructor parameters (`brand=x`, etc).

When we call the constructor (by creating an object of the class), we pass parameters to the constructor, which will set the value of the corresponding attributes to the same:

```
1  class Car {         // The class
```

```
  public:           // Access specifier
    string brand;   // Attribute
    string model;   // Attribute
    int year;       // Attribute
    Car(string x, string y, int z) { // Constructor with
       parameters
      brand = x;
      model = y;
      year = z;
    }
};

int main() {
  // Create Car objects and call the constructor with different
     values
  Car carObj1("BMW", "X5", 1999);
  Car carObj2("Ford", "Mustang", 1969);

  // Print values
  cout << carObj1.brand << " " << carObj1.model << " " <<
     carObj1.year << "\n";
  cout << carObj2.brand << " " << carObj2.model << " " <<
     carObj2.year << "\n";
  return 0;
}
```

Code 1.7: caption

1.4.2 Constructor Defined Outside the Class

You can also define the constructor outside the class using the scope resolution operator ::.

```
class Car {         // The class
  public:           // Access specifier
    string brand;   // Attribute
    string model;   // Attribute
    int year;       // Attribute
    Car(string x, string y, int z); // Constructor declaration
};

// Constructor definition outside the class
Car::Car(string x, string y, int z) {
  brand = x;
  model = y;
  year = z;
}

int main() {
  // Create Car objects and call the constructor with different
     values
  Car carObj1("BMW", "X5", 1999);
  Car carObj2("Ford", "Mustang", 1969);

  // Print values
```

```
22    cout << carObj1.brand << " " << carObj1.model << " " <<
        carObj1.year << "\n";
23    cout << carObj2.brand << " " << carObj2.model << " " <<
        carObj2.year << "\n";
24    return 0;
25  }
```

Code 1.8: caption

1.4.3 Why Using Constructors

Constructors **run by themselves when you create an object**. They **set things up so everything is ready right away**.

Think of it like this: When you order a pizza (object), the constructor is the chef who adds the sauce, cheese, and toppings before it gets to you - you don't have to do it yourself!

1.4.4 C++ Constructor Overloading

In C++, you can have more than one constructor in the same class. This is called **constructor overloading**.

Observation 1.4.2
Each constructor *must have a different number or type of parameters, so the compiler knows which one to use when you create an object.*

Idea 1.4.1 (Why Use Constructor Overloading?)
We use constructor overloading to:

- To give flexibility when creating objects
- To set default or custom values
- To reduce repetitive code

Example 1.4.2 (Example with Two Constructors)
This class has two constructors: one without parameters, and one with parameters:

```
1  class Car {
2    public:
3      string brand;
4      string model;
5
6      Car() {
7        brand = "Unknown";
8        model = "Unknown";
9      }
10
11     Car(string b, string m) {
12       brand = b;
13       model = m;
14     }
15 };
16
17 int main() {
```

```cpp
    Car car1;
    Car car2("BMW", "X5");
    Car car3("Ford", "Mustang");

    cout << "Car1: " << car1.brand << " " << car1.model << "\n";
    cout << "Car2: " << car2.brand << " " << car2.model << "\n";
    cout << "Car3: " << car3.brand << " " << car3.model;
    return 0;
}
```

Code 1.9: caption

The result of executing the following code is this:

```
Car1: Unknown Unknown
Car2: BMW X5
Car3: Ford Mustang
```

Code 1.10: caption

1.5 C++ Access Specifiers

Definition 1.5.1 (**Access Specifiers**)
Access specifiers *control how the members (attributes and methods) of a class can be accessed.*

They **help protect data** and **organize code** so that only the right parts can be seen or changed.

1.5.1 Using Access Specifiers

The `public` keyword is an access specifier.

Example 1.5.1 (**Use of public Keyword**)
In the code below, the *members are public* - which means that *they can be accessed and modified from outside the code*:

```cpp
class MyClass {  // The class
  public:        // Access specifier
    // class members goes here
};
```

Code 1.11: caption

In C++, there are three access specifiers:

- `public` - members are *accessible from outside the class.*
- `private` - members *cannot be accessed (or viewed) from outside the class.*
- `protected` - members *cannot be accessed from outside the class, however, they can be accessed in inherited classes.*

1.5.2 Private

Members declared as `private` cannot be accessed from outside the class.

Example 1.5.2 (**Differences Between public and private**)
In the following code, we demonstrate the *differences between public and private members*:

```
class MyClass {
  public:    // Public access specifier
    int x;   // Public attribute
  private:   // Private access specifier
    int y;   // Private attribute
};

int main() {
  MyClass myObj;
  myObj.x = 25;  // Allowed (public)
  myObj.y = 50;  // Not allowed (private)
  return 0;
}
```

Code 1.12: caption

Observation 1.5.1 (**Note on private**)
It is *possible to access private members of a class using a public method inside the same class.* See the next chapter (Encapsulation) on how to do this.

Idea 1.5.1 (**Tip**)
It is *considered good practice to declare your class attributes as private (as often as you can).* This will reduce the possibility of yourself (or others) to mess up the code. This is also the main ingredient of the **Encapsulation** concept, which you will learn more about in the next chapter.

By default, all members of a class are `private` if you don't specify an access specifier:

```
class MyClass {
  int x;   // Private attribute
  int y;   // Private attribute
};
```

Code 1.13: caption

1.5.3 Protected

Members declared as `protected` cannot be accessed from outside the class, but they can be accessed in child classes.

1.6 C++ Encapsulation

Definition 1.6.1 (**Encapsulation**)
The meaning of **Encapsulation**, is to make sure that *sensitive data is hidden from users.*

To achieve this, you must *declare class variables/attributes as private (cannot be accessed from outside the class).*

If you want others to read or modify the value of a private member, you *can provide public get and set methods.*

Example 1.6.1 (**Employee's Salary**)
Think of an employee's salary:

- The **salary is private** - the employee can't change it directly
- Only their manager can update it or share it when appropriate

Encapsulation works the same way. The **data is hidden, and only trusted methods can access or modify it**.

1.6.1 Access Private Members

To access a private attribute, use public `get` and `set` methods:

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

class Employee {
  private:
    // Private attribute
    int salary;

  public:
    // Setter
    void setSalary(int s) {
      salary = s;
    }
    // Getter
    int getSalary() {
      return salary;
    }
};

int main() {
  Employee myObj;
  myObj.setSalary(50000);
  cout << myObj.getSalary();
  return 0;
}
```

Code 1.14: caption

Observation 1.6.1 (**Explanation of Encapsulation**)
Example explained

- `salary` is `private` - it cannot be accessed directly
- `setSalary()` sets the value
- `getSalary()` returns the value

We use `myObj.setSalary(50000)` to assign a value, and `myObj.getSalary()` to print it.

Observation 1.6.2 (**Why Encapsulation?**)
We use encapsulation in order to:

- It is considered good practice to declare your class attributes as private (as often as you can). Encapsulation ensures better control of your data, because you (or others) can change one part of the code without affecting other parts
- Increased security of data

1.7 C++ The Friend Keyword

Normally, private members of a class can only be accessed using public methods like getters and setters. But in some cases, you *can use a special function called a friend function to access them directly.*

A **friend** function is *not a member of the class, but it is allowed to access the class's private data* as in the following example:

```
1  class Employee {
2    private:
3      int salary;
4
5    public:
6      Employee(int s) {
7        salary = s;
8      }
9
10     // Declare friend function
11     friend void displaySalary(Employee emp);
12 };
13
14 void displaySalary(Employee emp) {
15   cout << "Salary: " << emp.salary;
16 }
17
18 int main() {
19   Employee myEmp(50000);
20   displaySalary(myEmp);
21   return 0;
22 }
```

Code 1.15: caption

- The `friend` function `displaySalary()` is declared inside the `Employee class` but defined outside of it.
- Even though `displaySalary()` is not a member of the class, it can still access the private member `salary`.
- In the `main()` function, we create an `Employee` object and call the `friend` function to print its salary.

1.8 Inheritance

Definition 1.8.1 (**Inheritance**)
Inheritance *allows one class to reuse attributes and methods from another class.* It helps you write cleaner, more efficient code by avoiding duplication.

We group the *inheritance concept* into two categories:

- **derived class (child)** - the *class that inherits from another class*
- **base class (parent)** - the *class being inherited from*

To inherit from a class, use the : symbol.

Example 1.8.1 (**Example of Inheritance**)
In the code below, the `Car` class (child) inherits the attributes and methods from the `Vehicle` class (parent):

```
// Base class
class Vehicle {
  public:
    string brand = "Ford";
    void honk() {
      cout << "Tuut, tuut! \n" ;
    }
};

// Derived class
class Car: public Vehicle {
  public:
    string model = "Mustang";
};

int main() {
  Car myCar;
  myCar.honk();
  cout << myCar.brand + " " + myCar.model;
  return 0;
}
```

Code 1.16: caption

1.8.1 Multilevel Inheritance

A class can also be derived from one class, which is already derived from another class.

Example 1.8.2 (**Example of Multilevel Inheritance**)
In the following example, `MyGrandChild` is derived from class `MyChild` (which is derived from `MyClass`):

```
// Base class (parent)
class MyClass {
  public:
    void myFunction() {
      cout << "Some content in parent class." ;
    }
};

// Derived class (child)
class MyChild: public MyClass {
};
```

```
12
13 // Derived class (grandchild)
14 class MyGrandChild: public MyChild {
15 };
16
17 int main() {
18   MyGrandChild myObj;
19   myObj.myFunction();
20   return 0;
21 }
```

Code 1.17: caption