Lecture 6.1

Topics

1. Object Orientation – Introduction
2. C++ Class – Introduction

---

1. Object Orientation – Introduction

1.1 Objects – Definition

Generally, Object-Orientation can be of any consideration, abstraction, construction, and manipulation that delivered to some object.

So, what is an OBJECT? An object is a conceptually real or abstract thing that one can use to store data and methods (functions) to manipulate data.

Toward the programming end, an object

- Is identifiable
- Has some lifetime and space
- Provides data & operations that can be used to manipulate & interface with its surrounding environment

Object Type is a description for the creation of some objects; and an **object** is an instance or sample of Object Type.

Example 1

It is fair to say that “Car” is an object type and “Airplane” is another object type. Consider the following:

A Buick is a passenger car and a 300Z is a sport car. Thus, Buick and 300Z are instances of (general) car. A Boeing747 is a jumbo plane and an F15 is a fighter jet. Thus, Boeing747 and F15 are instances of (general) airplane.

Methods:

Each object might have data and methods. Methods specify the way in which data to be manipulated. Methods of an object operate only on the data of the same object and no others.

Encapsulation:

To protect data integrity, object may hide its data from the outside world. That means an object will protect its own internal characteristics.

Request (Message):

To use an object, a request must be sent to this object. The request will invoke some methods to perform the desired operations. The request contains object's identity, specified operation, and necessary parameters.

1.2 OO Analysis (OOA), OO Design (OOD), OO Programming (OOP)

OOA:

OOA is a method of analysis that examines and produces output (description/solution) for a problem constrained by the consideration of objects.

The given inputs will be examined in terms of what, how, and why for data and logic validations. Then the inter-relationship between the problem domain and solution domain will be analyzed using objects as the building blocks.

OOD:

OOD provides a method of designing or constructing the building blocks for solution domain based on object behaviors. The designers must then decide how data and methods should be packaged to form objects.

OOP:

OOP is the method of implementation (coding) in which the objects were created and used to perform the required tasks.

1.3 Object-Oriented Programming (OOP) Paradigm

Putting together the analysis and design into the programming phase we are constrained to follow an OOP paradigm. This programming paradigm has several essential elements: (1) Abstraction, (2) Encapsulation, (3) Modularity, (4) Hierarchy, (5) Concurrency, and (6) Persistence.

Abstraction:

An abstraction is a description for a set of characteristics of an object to make it unique from other objects.

Encapsulation:

Encapsulation is the process of hiding information belonged to an abstraction (object) from the external sources. Object encapsulation preserves its data integrity and structures.

Modularity:

System modularity refers to a decomposition of system into sets of well-defined and cohesive modules. In OOP, a module can generally be thought as a group of objects so that when put all together will interact and make up the system.

Hierarchy:

Hierarchy is a ranking of abstractions or, perhaps, of object types. Hierarchy induces inheritance; and in OOP, inheritance plays an important role. Using inheritance, more objects can be created from a few with ease.

In object orientation, a base abstraction (base class) is the one that can be used to create the new one. This new one is called the derived abstraction (derived class). The derived abstraction will inherit some or the entire base plus some of its own.

Concurrence:

In loose terms, concurrency is the property that distinguishes scope of objects within a system. Objects can be invoked concurrently (that is, active at the same time) or by some interactive order (e.g., nested process). These concurred objects will retain their abstraction, encapsulation, and hierarchical properties.

Persistence:

Persistence is the property of an object through which its existence transcends time (i.e., the object continues to exist after its creator ceases to exist) and/or space (that means object can be relocated to different places in the system, e.g., location in memory space).

Thus, C++ is used in an OOP paradigm to allow the creation of objects through a so-called **`class`**. C++ is also called the class-based object-oriented language and will have the following traits:

Encapsulation:

Encapsulation is found in classes (object types) to provide ways for binding methods and data together and to prevent outside interference. Methods and data of an object can be kept either public or private (or a third kind: protected).

Private members are known to and accessible only by other members of the object. Other members of the object and other parts of the program can access public members.

Polymorphism:

Polymorphism is the property that allows one name (as of method/function name) to be used for two or more related but technically different purposes. For examples, the same function name can be used for many different purposes, this is called function overloading; or different data types can also be used with the same operator, this is called operator overloading.

Inheritance:

Inheritance is the process that allows derived classes to be created from some base classes. Inheritance can either be single or multiple. That means the derived class can be obtained either only from a single base class (i.e., single inheritance) or from a group of two or more bases classes (i.e., multiple inheritance).

2. C++ `class` -- Introduction

Currently, many computer programs are written to run real world simulations and events, which are hard to describe without using terms related to or relevant to objects.

In C++, a **`class`** provides the mechanism to create objects. The syntax of class will allow the user to characterize the definition and its objects. Let's look at **`class.`**

2.1 `Class` -- Definition

*Class is an abstraction used to define a new data **type** through the **`class`** keyword.*

The general syntax of a **`class`** declaration is as follows,

```
class ClassName {
   private:
  //private functions and data members of class
   public:
  //public functions and data members of class
   protected:
  //protected functions and data members
};
```

where

- The keyword **`class`** is used to declare a new data type of `ClassName`; or **`ClassName`** is the name of this new data type.
- Everything above the **`public`** (access) specifier is private to this class. That means it can be accessed ONLY by other parts of this class and none by others.
- All functions and variables declared after the **`public`** specifier are accessible both by other members of this class and by any other parts of the program that contain this class.

- Functions can just be `prototyped` and then their definitions are given later. Each function of a class is `specified` and defined by using the scope resolution operator `::` with correct return type and arguments.

Note that there are three different specifiers that can be used to partition function and variable members: **`private`**, **`public`**, and **`protected`**. There may be data members as well as function members defined in a class definition.

2.2 Specification Groups -- `private`, `public`, and `protected`

The default specification of class members is **`private`** (which is in contrast to that of structured data that means **`struct`**, where the default specification is **`public`**).

A **`private`** member can only be accessed by other class members (as these members are defined within the same class definition). Mostly, the members will be functions defined for the given class.

A **`public`** member can be accessed by all other members within the definition of the class **and** outside of the class definition through instances of this class (or equivalently, the objects of this class). Outside accessing to the members of a given class is done through an object and the dot (**`.`**) operator or a pointer and the arrow (**`->`**) operator.

A **`protected`** member is almost the same as a **`private`** member. In addition, a protected member can also be accessed by the base class and its derived classes in an inheritance structure. Discussion about the **`protected`** members will be given later.

Declaring and defining a class mean to define a new data type (no memory allocated, no object created). This will just provide the mean to create objects. Whenever an object was created from a class then this object is an instance of that class and memory must be allocated for this object to handle values belonging to the data members.

Thus, just declaring a new class type does not give rise to any of its use (i.e., memory allocation and access) until an instance is needed or called for. We will be working with objects for the rest of the semester.

2.3 Class -- Examples

The simplest example of class is an empty class.

Example 1

```
/**
 *Program Name: cis25L0611.cpp
 *Discussion:   Empty class
 */
#include <iostream>
using namespace std;

class ClassName {
private:
  //private functions and data members of class
public:
  //public functions and data members of class
protected:
  //protected functions and data members
};
```

```
class EmptyClass {
  //Nothing is given in the definition body
};

int main( void ) {
  int iX;

  EmptyClass empt1;

  return 0;
}
```

The above example shows that one can certainly and intentionally waste system resource and get away with it.

Example 2

```
/**
 *Program Name:  cis25L0612.cpp
 *Discussion:    Empty Class
 *               Private and unusable class
 */
#include <iostream>
using namespace std;

class EmptyClass {
};

class OnePrivateValue {
  int iValue;
};

int main( void ) {
  int iX;

  EmptyClass empt1;

  OnePrivateValue opv1;

  return 0;
}
```

In the above example, it is a bit more “extravagant” way of wasting system resource. It seems to be OK with having data member (only one) as part of the class definition; but it would be impossible to work or use the object in any meaningful ways.

Note that **`opv1`** is also called a “dead” object, which is the object that you cannot affect or change its value(s).

Example 3

```
/**
 *Program Name:  cis25L0613.cpp
 *Discussion:    Empty Class
 *               Private and unusable class
 *               Public class
```

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

class EmptyClass {
};

class OnePrivateValue {
  int iValue;
};

class OnePublicValue {
public:
  int iValue;
};

int main( void ) {
  int iX;

  EmptyClass empt1;

  OnePrivateValue opr1;

  OnePublicValue op1;

  cout << "\nAccessing op1 -- "
       << "\n\tValue of member iValue : "
       << op1.iValue << endl;

  op1.iValue = 0;

  cout << "\nAccessing op1 -- "
       << "\n\tValue of member iValue : "
       << op1.iValue << endl;

  iX = op1.iValue;

  return 0;
}
```

OUTPUT

```
Accessing op1 --
        Value of member iValue : -858993460

Accessing op1 --
        Value of member iValue : 0
```

In the above example, **`OnePublicValue`** class is defined with a single data member of public specification -- **`iValue`**. As an object **`op1`** was created, this object can access the public data member **`iValue`** through the dot operator.

2.4 Class – Topics of Discussion

There are several topics relevant to classes that will be discussed in the remaining sessions of this class. They are given as follows,

(1) Class definition

(2) Function members and overloading
(3) Operator functions and overloading
(4) Type conversion for classes
(5) Class with dynamic (data) members
(6) Class inheritance
(7) Type conversion in class hierarchies
(8) Polymorphism and virtual functions/methods
(9) Abstract classes
(10) Multiple inheritance
(11) Templates