Object Oriented Programming with C++ Day 2

Compiled by
M S Anand

Department of Computer Science

Introduction

Text Book(s):

1. “C++ Primer”, Stanley Lippman, Josee Lajoie, Barbara E Moo, Addison-Wesley Professional, 5th Edition..

Structure

User defined data type

used to store group of items of non-similar data types.

General syntax:

```
struct  structure_name
{
        member1;
        member2;

        memberN;
};
```

Structure

Two types of members:

Data Member: These members are normal C++ variables. We can create a structure with variables of different data types in C++.

Member Functions: These members are normal C++ functions. Along with variables, we can also include functions inside a structure declaration.

Sample program SimpleStruct.cpp

In C++, a structure is the same as a class except for a few differences. The most important of them is security. A Structure is not secure and cannot hide its implementation details from the end user while a class is secure and can hide its programming and designing details.

Structure

Size of structures – Word aligned

Padding

Packing

A program to demonstrate this: sstruct.cpp

Sample program - structure with member functions and constructor is StructWithMethods.cpp

OOP with C++

Classes and Objects

Class is the building block that leads to Object-Oriented programming. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class.

A C++ class is like a blueprint for an object.

Example: Consider the Class of **Cars**.
There may be many cars with different names and brands but all of them will share some common properties - all of them will have *4 wheels*, a steering wheel, brakes, *Speed Limit*, *Mileage range,* etc.

So here, Car is the class, and wheels, speed limits, and mileage are their properties.

Classes and Objects

A Class is a user-defined data type that has data members and member functions.

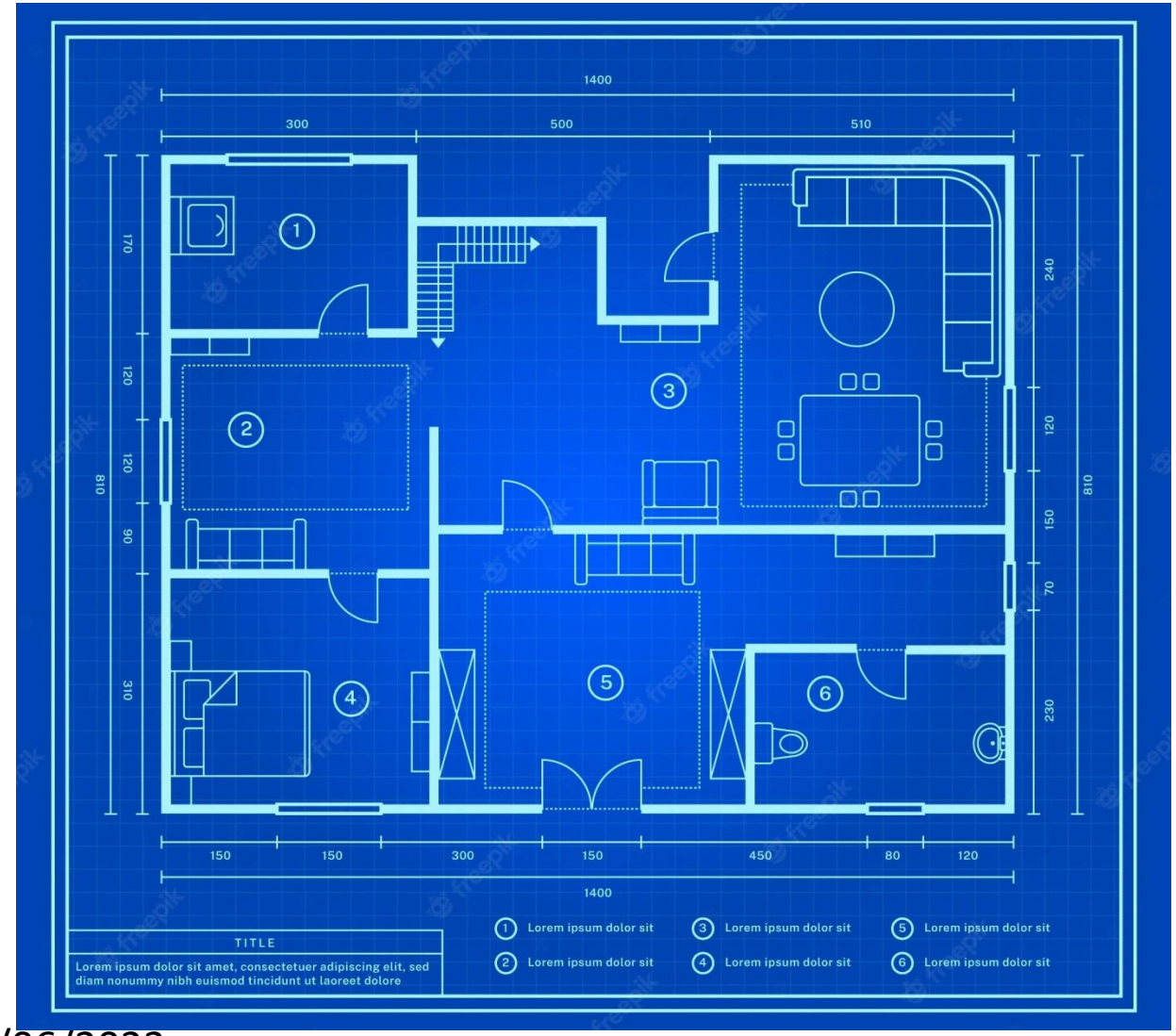
Data members are the data variables and member functions are the functions used to manipulate these variables together, these data members and member functions define the properties and behavior of the objects in a Class.

In the above example of class *Car*, the data members are *speed limit*, *mileage,* etc, and member functions can be *applying brakes*, *increasing speed,* etc.

An **Object** is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

OOP with C++

Classes and Objects



Classes and Objects

Syntax

```
class class_name
{
        Access specifier:     // can be public, private or protected

        Data members;         // Variables to be used

        Member functions ()  // Methods to access data members

};      // Ends with a semicolon
```

Classes and Objects

Declaring Objects

When a class is defined, only the specification for the object is defined; no memory or storage is allocated. To use the data and access functions defined in the class, we need to create objects.

Syntax

ClassName ObjectName;

Accessing data members and member functions: The data members and member functions of the class can be accessed using the dot('.') operator with the object. For example, if the name of the object is *obj* and you want to access the member function with the name *printName()* then you will have to write *obj.printName()*.

Accessing Data Members

The public data members are also accessed in the same way. However, the private data members are not allowed to be accessed directly by the object. Accessing a data member depends solely on the access control of that data member.

This access control is given by Access modifiers in C++.

There are three access modifiers: **public, private, and protected**.

Classes and Objects

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.

Program to show the effect of access modifiers

A simple example SimpleClass.cpp

Where can we define member functions?

1. Inside the class
2. Outside the class

Sample program MemberOutsideClass.cpp

Note: It is possible to access private members of a class using a public method inside the same class.

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/

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

Classes and Objects

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.

Access Private Members

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

A sample program SetGet.cpp

Constructor and Destructor

Constructor and Destructor are special member functions of the class which are created by the C++ compiler or can be defined by the user.

Constructor is used to initialize the object of the class while destructor is called by the compiler when the object is destroyed.

Constructor

A constructor is a special member function of a class and shares the same name as the class.

Constructor is called by the compiler whenever the object of the class is created; it allocates the memory to the object and initializes class data members to default values or values passed by the user while creating an object.

Constructors don’t have any return type because their work is to just create and initialize an object.

Basic syntax of Constructor

```
class class_name{
  private:
  // private members

  public:

  // declaring constructor
  class_name({parameters})
  {
    // constructor body
  }

};
```

Key points about Constructor

Access specifiers

Constructors can be defined as public, protected, or private as per the requirements. By default or default constructors, which are created by the compiler are declared as public. If the constructor is created as private, then we are not able to create its object.

When does someone define constructor as private?

When there is no requirement of the object of a class (in a situation when all class members are static) we can define its constructor as private.

Usually, constructors are defined as public because constructors are used to create an object and initialize the class data members for the object. An object is always created from outside of class, which justifies making constructors public.

Inheritance

As a derived class can access all the public properties of the base class, it can call its constructor also if it is not declared as private. Also, the constructor's address cannot be referenced.

Virtual

Constructor in C++ cannot be declared as virtual.

Types of Constructors

1. Default constructor
2. Parameterized constructor
3. Copy Constructor
4. Dynamic Constructor

Default Constructor

Default constructor is also known as a zero-argument constructor, as it doesn't take any parameters. It can be defined by the user; if not, then the compiler creates it on its own. Default constructor always initializes data members of the class with the same value they were defined.

```
class class_name{
  private:
  // private members

  public:

  // declaring default constructor
  class_name()
  {
    // constructor body
  }
  };
```

A sample program DefaultConstructor.cpp

Parameterized Constructor

Parameterized constructor is used to initialize data members with the values provided by the user. We can define more than one parameterized constructor according to the need of the user, but we have to follow the rules of the function overloading; a different set of arguments must be there for each constructor.

```
class class_name{
  private:
  // private members

  public:

  // declaring parameterized constructor
  class_name(parameter1, parameter2,...)
  {
    // constructor body
  }
  };
```

A sample program is ParameterizedConstructor.cpp

Copy Constructor

If we have an object of a class and want to create its copy in a new declared object of the same class, then a copy constructor is used. The compiler provides each class a default copy constructor; users can define it also. It takes a single argument which is an object of the same class.

```
class class_name{
  private:
  // private members

  public:

  // declaring copy constructor
  class_name(const class_name& obj)
  {
    // constructor body
  }
};
```

A sample program: CopyConstructor.cpp

Dynamic Constructor

When memory is allocated dynamically to the **data members** at the runtime using a new operator, the constructor is known as the dynamic constructor. This constructor is similar to the default or parameterized constructor; the only difference is it uses a new operator to allocate the memory.

```
class class_name{
  private:
  // private members

  public:

  // declaring dynamic constructor
  class_name({parameters})
  {
    // constructor body where data members are initialized using new operator
  }
};
```

A sample program is DynamicConstructor.cpp

Delegating Constructor

Sometimes it is useful for a constructor to be able to call another constructor of the same class. This feature, called **Constructor Delegation**,

Sample program: Delegate.cpp

Destructor

A destructor is called by the compiler when the object is destroyed and its main function is to deallocate the memory of the object. The object may be destroyed when the program ends, or local objects of the function get out of scope when the function ends or in any other case.

Destructor has the same name as the class with prefix tilde(~) operator and it cannot be overloaded . Destructors take no argument and have no return type and return value.

Destructor

```
class class_name{
  private:
  // private members

  public:

  // declaring destructor
  ~class_name()
  {
    // destructor body
  }

}
```

A sample program ShowDestructor.cpp

Initializer list

Initializer List is used in initializing the data members of a class. The list of members to be initialized is indicated with constructor as a comma-separated list followed by a colon.

An example: Initialization.cpp

The above code is just an example for syntax of the Initializer list. In the above code, x and y can also be easily initialized inside the constructor. But there are situations where initialization of data members inside constructor doesn't work and Initializer list must be used.

Some scenarios where Initialization needs to be used

For initialization of non-static const data members:

const data members must be initialized using Initializer List. In the following example, “t” is a const data member of Test class and is initialized using Initializer List.

Reason for initializing the const data member in the initializer list is because no memory is allocated separately for const data member, it is folded in the symbol table due to which we need to initialize it in the initializer list.

Also, it is a Parameterized constructor and we don’t need to call the assignment operator which means we are avoiding one extra operation.

Example program: Initialize2.cpp

For initialization of reference members:
Reference members must be initialized using Initializer List. In the following example, “t” is a reference member of Test class and is initialized using Initializer List.

Sample program: Initialize3.cpp

For initialization of member objects which do not have default constructor:

In the following example, an object “a” of class “A” is data member of class “B”, and “A” doesn’t have default constructor. Initializer List must be used to initialize “a”.

Sample program: Initializer4.cpp

For initialization of base class members : Like the previous point, the parameterized constructor of the base class can only be called using Initializer List.

Sample program: Initializer5.cpp

When constructor's parameter name is same as data member
If constructor's parameter name is same as data member name then the data member must be initialized either using this pointer or Initializer List. In the following example, both member name and parameter name for A() is "i".

Sample program: Initializer6.cpp

For Performance reasons:
It is better to initialize all class variables in Initializer List instead of assigning values inside body.

Sample program: Find out.

Dynamic memory management using Constructor and Destructor

Because classes have complicated internal structures, including data and functions, object initialization and cleanup for classes is much more complicated than it is for simple data structures.

Constructors and destructors are special member functions of classes that are used to construct and destroy class objects. Construction may involve memory allocation and initialization for objects. Destruction may involve cleanup and deallocation of memory for objects.

Classes and Objects

Use of the **new** operator signifies a request for the memory allocation on the heap. If sufficient memory is available, it initializes the memory and returns its address to the pointer variable.

The **new** operator should only be used if the data object should remain in memory until **delete** is called. Otherwise if the **new** operator is not used, the object is automatically destroyed when it goes out of scope.

In other words, the objects using **new** are to be cleaned up manually while other objects are automatically cleaned when they go out of scope.

So, whenever a class has members which need memory allocation, allocate the same inside the constructor using “malloc” and the same has to be deallocated using “free” in the destructor.

A simple example: DelegateConstruct.cpp

Copy constructor (Some more info)

Copy constructors are the member functions of a class that initialize the data members of the class using another object of the same class. It copies the values of the data variables of one object of a class to the data members of another object of the same class. A copy constructor can be defined as follows:

```
class Class_name {
    Class_name(Class_name &old_object){     //copy constructor
        Var_name = old_object.Var_name;
                ….
                ….
                        }
        }
```

A copy constructor in C++ is further categorized into two types:

1. Default Copy Constructor
2. User-defined Copy Constructor

Default Copy Constructors: When a copy constructor is not defined, the C++ compiler automatically supplies with its self-generated constructor that copies the values of the object to the new object.

Sample Program: CopyConstructor1.cpp

User-defined Copy Constructors: In case of a user-defined copy constructor, the values of the parameterised object of a class are copied to the member variables of the newly created class object. The initialization or copying of the values to the member variables is done as per the definition of the copy constructor.

Sample program: CopyConstructor2.cpp

Copy Assignment Operator

Unlike other object-oriented languages like Java, C++ has robust support for object deep-copying and assignment. You can choose whether to pass objects to functions by reference or by value, and can assign objects to one another as though they were primitive data types.

C++ handles object copying and assignment through two functions called copy constructors and assignment operators.

While C++ will automatically provide these functions if you don't explicitly define them, in many cases you'll need to manually control how your objects are duplicated.

Crucial difference between assignment and initialization. Consider the following code snippet:

```
MyClass one;
MyClass two = one;
```

Here, the variable two is initialized to one because it is created as a copy of another variable. When two is created, it will go from containing garbage data directly to holding a copy of the value of one with no intermediate step.

However, if we rewrite the code as

```
MyClass one, two;
two = one;
```

Then two is assigned the value of one. Note that before we reach the line two = one, two already contains a value.

What is the difference between assignment and initialization ?

When a variable is created to hold a specified value, it is being initialized, whereas when an existing variable is set to hold a new value, it is being assigned.

How can you be sure about this?
If you're ever confused about when variables are assigned and when they're initialized, you can check by sticking a const declaration in front of the variable. If the code still compiles, the object is being initialized. Otherwise, it's being assigned a new value.

Always remember that the assignment operator is called only when assigning an existing object a new value. Otherwise, you're using the copy constructor.

OOP with C++

Copy Assignment operator

Sample program: CopyAssignment.cpp

The above program shows the difference between a simple assignment v/s Copy Assignment.

The rvalue references

The Right Time for Rvalue References

Rvalue references are a new category of reference variables that can bind to *rvalues*. Rvalues are slippery entities, such as **temporaries** and literal values;

Technically, an rvalue is an unnamed value that exists only during the evaluation of an expression. For example, the following expression produces an rvalue:

x+(y*z); // A C++ expression that produces a temporary

C++ creates a temporary (an rvalue) that stores the result of y*z, and then adds it to x. Conceptually, this rvalue evaporates by the time you reach the semicolon at the end of the full expression.

A declaration of an rvalue reference looks like this:

std::**string&& rrstr**; //C++11 rvalue reference variable

The traditional reference variables of C++ e.g.,

std::string& ref; are now called *lvalue references*.

Move semantics

In C++03, there were costly and unnecessary deep copies which happened implicitly when objects were passed by value.

In C++11, the resources of the objects can be moved from one object to another rather than copying the whole data of the object to another. This can be done by using “move semantics” in C++11.

Move semantics points the other object to the already existing object in the memory. It avoids the instantiation of unnecessary temporary copies of the objects by giving the resources of the already existing object to the new one and safely taking from the existing one. Taking resources from the old existing object is necessary to prevent more than one object from having the same resources.

Move semantics uses move constructor and r-value references.

Move semantics

The following example shows the way the move operation works in contrast to the copy operation (with move constructor and copy constructor)

Source program: MoveExample1.cpp

The working of the “move assignment” operator is shown in the following example

Move_Assignment.cpp

THANK YOU

M S Anand
Department of Computer Science Engineering

anandms@yahoo.com