Unit 1 : Principles of object oriented programming

Difference Between Procedure Oriented Programming (POP) & Object Oriented Programming (OOP)

	Procedure Oriented Programming	Object Oriented Programming
Divided Into	In POP, program is divided into small parts called functions.	In OOP, program is divided into parts called objects.
Importance	In POP, Importance is not given to data but to functions as well as sequence of actions to be done.	In OOP, Importance is given to the data rather than procedures or functions because it works as a real world.
Approach	POP follows Top Down approach.	OOP follows Bottom Up approach.
Access Specifiers	POP does not have any access specifier.	OOP has access specifiers named Public, Private, Protected, etc.
Data Moving	In POP, Data can move freely from function to function in the system.	In OOP, objects can move and communicate with each other through member functions.
Expansion	To add new data and function in POP is not so easy.	OOP provides an easy way to add new data and function.
Data Access	In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the system.	In OOP, data can not move easily from function to function, it can be kept public or private so we can control the access of data.
Data Hiding	POP does not have any proper way for hiding data so it is less secure.	OOP provides Data Hiding so provides more security.
Overloading	In POP, Overloading is not possible.	In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.
Examples	Example of POP are : C, VB, FORTRAN, Pascal.	Example of OOP are : C++, JAVA, VB.NET, C#.NET.

Basic concepts of object oriented Programming

Object-oriented programming (OOP) is a programming paradigm that uses “Objects “and their interactions to design applications and computer programs.

There are different types of OOPs are used, they are

1. Object
2. Class
3. Data Abstraction & Encapsulation
4. Inheritance
5. Polymorphism
6. Dynamic Binding
7. Message Passing

1. Objects

Objects are the basic run-time entities in an object-oriented system. Programming problem is analyzed in terms of objects and nature of communication between them. When a program is executed, objects interact with each other by sending messages. Different objects can also interact with each other without knowing the details of their data or code.

2. Classes

A class is a collection of objects of similar type. Once a class is defined, any number of objects can be created which belong to that class.

3. Data Abstraction and Encapsulation

Abstraction refers to the act of representing essential features without including the background details or explanations. Classes use the concept of abstraction and are defined as a list of abstract attributes. Storing data and functions in a single unit (class) is encapsulation. Data cannot be accessible to the outside world and only those functions which are stored in the class can access it.

4. Inheritance

Inheritance is the process by which objects can acquire the properties of objects of other class. In OOP, inheritance provides reusability, like, adding additional features to an existing class without modifying it. This is achieved by deriving a new class from the existing one. The new class will have combined features of both the classes.

- Single level inheritance
- Multiple inheritance
- Multi-level inheritance
- Hybrid inheritance
- Hierarchial inheritance

5. Polymorphism

Polymorphism means the ability to take more than one form. An operation may exhibit different behaviors in different instances. The behavior depends on the data types used in the operation. Polymorphism is extensively used in implementing Inheritance.

6. Dynamic binding

It contains a concept of Inheritance and Polymorphism.

7. Message Passing

It refers to that establishing communication between one place to another.

Benefits of object oriented programming

Some of the advantages of object-oriented programming include:

1. Improved software-development productivity: Object-oriented programming is modular, as it provides separation of duties in object-based program development. It is also extensible, as objects can be extended to include new attributes and behaviors. Objects can also be reused within an across applications. Because of these three factors – modularity, extensibility, and

reusability – object-oriented programming provides improved software-development productivity over traditional procedure-based programming techniques.

2. Improved software maintainability: For the reasons mentioned above, objectoriented software is also easier to maintain. Since the design is modular, part of the system can be updated in case of issues without a need to make large-scale changes.

3. Faster development: Reuse enables faster development. Object-oriented programming languages come with rich libraries of objects, and code developed during projects is also reusable in future projects.

4. Lower cost of development: The reuse of software also lowers the cost of development. Typically, more effort is put into the object-oriented analysis and design, which lowers the overall cost of development.

5. Higher-quality software: Faster development of software and lower cost of development allows more time and resources to be used in the verification of the software. Although quality is dependent upon the experience of the teams, objectoriented programming tends to result in higher-quality software.

Some of the disadvantages of object-oriented programming include:

1. Steep learning curve: The thought process involved in object-oriented programming may not be natural for some people, and it can take time to get used to it. It is complex to create programs based on interaction of objects. Some of the key programming techniques, such as inheritance and polymorphism, can be challenging to comprehend initially.

2. Larger program size: Object-oriented programs typically involve more lines of code than procedural programs.

3. Slower programs: Object-oriented programs are typically slower than procedure based programs, as they typically require more instructions to be executed.

4. Not suitable for all types of problems: There are problems that lend themselves well to functional-programming style, logic-programming style, or procedure-based programming style, and applying object-oriented programming in those situations will not result in efficient programs.

Applications of using OOP

- User interface design such as windows, menu ,…
- Real Time Systems
- Simulation and Modeling
- Object oriented databases
- AI and Expert System
- Neural Networks and parallel programming
- Decision support and office automation system etc

What is C++?

C++ is an Enhanced version of c Language which is developed by Bjarne Stroustrup in 1980 in AT & T's Bell Lab. C++ Inherits many features from C Language and it also Some More Features and This Makes C++ an OOP Language. C++ is used for Making Some Code which May used by another Peoples for Making their Applications. C++ Provides Reusability of Code for Another user. C++ is also Known as Object Oriented Language or Simply OOP Language because it Provides

Capability to either Make Stand alone Program or either Make a Reusable Code for Another Users.

Application of C++?

C++ is a general purpose programming language designed to make programming more enjoyable for the serious programmer. For many uses, C++ is not the ideal language. You might prefer Tcl/Tk for writing a user interface, SQL for relational database queries, Java for network programming, or Yacc for writing a parser. C++ is used because it works well when the ideal language is (for whatever reason) not available, and because it interfaces easily with the libraries and the other languages you use.

It's no accident that you can interface C++ with almost any language interpreter or library you find. You rarely find a big program written all in one language, or without using libraries, so easy integration with other languages and libraries was a key design goal.

Most problems have no specialized language to solve them; some because none has (yet) been worth creating, and others because an interpreter would add too much overhead. When you can't afford a specialized language for part of a problem, a library may suffice. C++ was designed with libraries always in mind, and its most useful features are those that help you write portable, efficient, easy-to-use libraries.

Input/Output Operators in C++

I/O Library Header Files:

There are following header files important to C++ programs:

Header File	Function and Description
<iostream>	This file defines the cin, cout, cerr and clog objects, which correspond to the standard input stream, the standard output stream, the un-buffered standard error stream and the buffered standard error stream, respectively.
<iomanip>	This file declares services useful for performing formatted I/O with so-called parameterized stream manipulators, such as setw and setprecision.
<fstream>	This file declares services for user-controlled file processing. We will discuss about it in detail in File and Stream related chapter.

Standard Output (cout)

By default, the standard output of a program is the screen, and the C++ stream object defined to access it is cout. cout is used in conjunction with the insertion operator, which is written as << (two "less than" signs).

```
cout << "Output sentence"; // prints Output sentence on screen
cout << 120;              // prints number 120 on screen
cout << x;                // prints the content of x on screen
```

The << operator inserts the data that follows it into the stream preceding it. In the examples above it inserted the constant string Output sentence, the numerical constant 120 and variable x into the standard output stream cout. Notice that the sentence in the first instruction is enclosed between double quotes (") because it is a constant string of characters. Whenever we want to use constant strings of characters we must enclose them between double quotes (") so that they can be clearly

distinguished from variable names. For example, these two sentences have very different results:

```
cout << "Hello";  // prints Hello
cout << Hello;    // prints the content of Hello variable
```

The insertion operator (<<) may be used more than once in a single statement:

```
cout << "Hello, " << "I am " << "a C++ statement";
```

This last statement would print the message Hello, I am a C++ statement on the screen. The utility of repeating the insertion operator (<<) is demonstrated when we want to print out a combination of variables and constants or more than one variable:

```
cout << "Hello, I am " << age << " years old and my zipcode is " << zipcode;
```

If we assume the age variable to contain the value 24 and the zipcode variable to contain 90064 the output of the previous statement would be:

```
Hello, I am 24 years old and my zipcode is 90064
```

It is important to notice that cout does not add a line break after its output unless we explicitly indicate it, therefore, the following statements:

```
cout << "This is a sentence.";
cout << "This is another sentence.";
```

will be shown on the screen one following the other without any line break between them:

```
This is a sentence.This is another sentence.
```

even though we had written them in two different insertions into cout. In order to perform a line break on the output we must explicitly insert a new-line character into cout. In C++ a new-line character can be specified as \n(backslash, n):

```
cout << "First sentence.\n";
cout << "Second sentence.\nThird sentence.";
```

This produces the following output:

```
First sentence.
Second sentence.
Third sentence.
```

Additionally, to add a new-line, you may also use the endl manipulator. For example:

```
cout << "First sentence." << endl;
cout << "Second sentence." << endl;
```

would print out:

```
First sentence.
Second sentence.
```

The endl manipulator produces a newline character, exactly as the insertion of '\n' does, but it also

has an additional behavior when it is used with buffered streams: the buffer is flushed. Anyway, cout will be an unbuffered stream in most cases, so you can generally use both the \n escape character and the endl manipulator in order to specify a new line without any difference in its behavior.

Standard Input (cin)

The standard input device is usually the keyboard. Handling the standard input in C++ is done by applying the overloaded operator of extraction (>>) on the cin stream. The operator must be followed by the variable that will store the data that is going to be extracted from the stream. For example:

```
int age;
cin >> age;
```

The first statement declares a variable of type int called age, and the second one waits for an input from cin (the keyboard) in order to store it in this integer variable.

cin can only process the input from the keyboard once the RETURN key has been pressed. Therefore, even if you request a single character, the extraction from cin will not process the input until the user presses RETURN after the character has been introduced.

You must always consider the type of the variable that you are using as a container with cin extractions. If you request an integer you will get an integer, if you request a character you will get a character and if you request a string of characters you will get a string of characters.

```
// i/o example

#include <iostream>
using namespace std;

int main ()
{
  int i;
  cout << "Please enter an integer value: ";
  cin >> i;
  cout << "The value you entered is " << i;
  cout << " and its double is " << i*2 << ".\n";
  return 0;
}
Please enter an integer value: 702
The value you entered is 702 and its double is 1404.
```

The user of a program may be one of the factors that generate errors even in the simplest programs that use cin(like the one we have just seen). Since if you request an integer value and the user introduces a name (which generally is a string of characters), the result may cause your program to misoperate since it is not what we were expecting from the user. So when you use the data input provided by cin extractions you will have to trust that the user of your program will be cooperative and that he/she will not introduce his/her name or something similar when an integer value is requested. A little ahead, when we see the stringstream class we will see a possible solution for the errors that can be caused by this type of user input.

You can also use cin to request more than one datum input from the user:

```
cin >> a >> b;
```

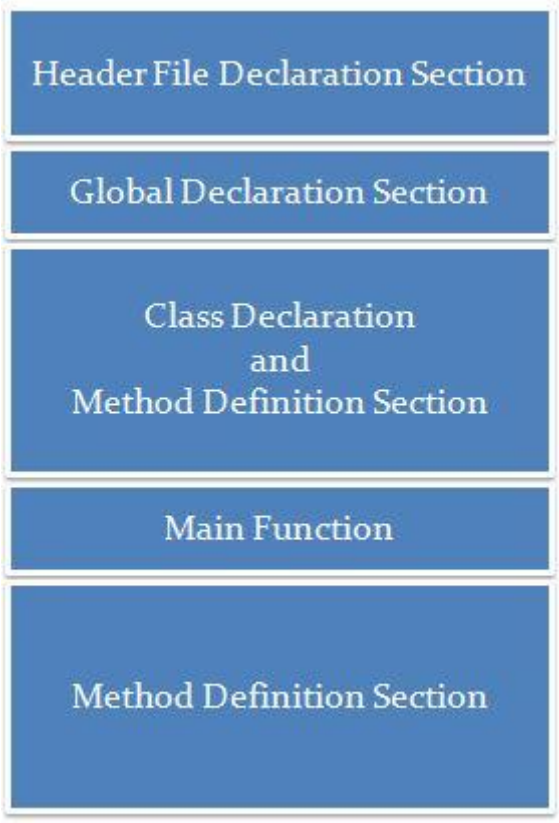
is equivalent to:

```
cin >> a;
cin >> b;
```

In both cases the user must give two data, one for variable a and another one for variable b that may be separated by any valid blank separator: a space, a tab character or a newline.

Structure of C++ Program

C++ Programming language is most popular language after C Programming language. C++ is first Object oriented programming language. We have summarize structure of C++ Program in the following Picture -



Structure of C++ Program

Section 1 : Header File Declaration Section

1. Header files used in the program are listed here.
2. Header File provides **Prototype declaration** for different library functions.
3. We can also include **user define header file**.
4. Basically all preprocessor directives are written in this section.

Section 2 : Global Declaration Section

1. Global Variables are declared here.
2. Global Declaration may include -

- Declaring Structure
- Declaring Class
- Declaring Variable

Section 3 : Class Declaration Section

1. Actually this section can be considered as sub section for the global declaration section.
2. Class declaration and all methods of that class are defined here.

Section 4 : Main Function

1. Each and every C++ program always starts with main function.
2. This is entry point for all the function. Each and every method is called indirectly through main.
3. We can create class objects in the main.
4. Operating system call this function automatically.

Section 5 : Method Definition Section

1. This is optional section . Generally this method was used in C Programming.

Namespaces

Namespaces allow to group entities like classes, objects and functions under a name. This way the global scope can be divided in "sub-scopes", each one with its own name.

The format of namespaces is:

```
namespace identifier
{
entities
}
```

Where identifier is any valid identifier and entities is the set of classes, objects and functions that are included within the namespace. For example:

```
namespace myNamespace
{
  int a, b;
}
```

In this case, the variables a and b are normal variables declared within a namespace called myNamespace. In order to access these variables from outside the myNamespace namespace we have to use the scope operator ::. For example, to access the previous variables from outside myNamespace we can write:

```
myNamespace::a
myNamespace::b
```

The functionality of namespaces is especially useful in the case that there is a possibility that a global object or function uses the same identifier as another one, causing redefinition errors. For example:

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

namespace first
{
  int var = 5;
}

namespace second
{
  double var = 3.1416;
}

int main () {
  cout << first::var << endl;
  cout << second::var << endl;
  return 0;
}
```

```
5
3.1416
```

In this case, there are two global variables with the same name: var. One is defined within the namespace first and the other one in second. No redefinition errors happen thanks to namespaces.