UNIT I
FUNDAMENTALS

Object–Oriented Programming Concepts:

The important concept of OOPs are:

- Objects
- Classes
- Inheritance
- Data Abstraction
- Data Encapsulation
- Polymorphism
- Overloading
- Reusability

Objects:

Object is the basic unit of object-oriented programming. Objects are identified by its unique name. An object represents a particular instance of a class. There can be more than one instance of an object. Each instance of an object can hold its own relevant data.

An Object is a collection of data members and associated member functions also known as methods.

Classes:

Classes are data types based on which objects are created. Objects with similar properties and methods are grouped together to form a Class. Thus a Class represent a set of individual objects. Characteristics of an object are represented in a class as **Properties**. The actions that can be performed by objects becomes functions of the class and is referred to as **Methods**.

No memory is allocated when a class is created. Memory is allocated only when an object is created, i.e., when an instance of a class is created.

Inheritance:

Inheritance is the process of forming a new class from an existing class or *base class*. The base class is also known as *parent class* or *super class*, The new class that is formed is called *derived class*. Derived class is also known as a *child class* or *sub class*. Inheritance helps in reducing the overall code size of the program, which is an important concept in object-oriented programming.

Data Abstraction:

Data Abstraction increases the power of programming language by creating user defined data types. Data Abstraction also represents the needed information in the program without presenting the details.

Data Encapsulation:

Data Encapsulation combines data and functions into a single unit called Class. When using Data Encapsulation, data is not accessed directly; it is only accessible through the functions present inside the class. Data Encapsulation enables the important concept of data hiding possible.

Polymorphism:
Polymorphism allows routines to use variables of different types at different times. An operator or function can be given different meanings or functions. Polymorphism refers to a single function or multi-functioning operator performing in different ways.

Overloading:
Overloading is one type of Polymorphism. It allows an object to have different meanings, depending on its context. When an exiting operator or function begins to operate on new data type, or class, it is understood to be overloaded.

Reusability:
This term refers to the ability for multiple programmers to use the same written and debugged existing class of data. This is a time saving device and adds code efficiency to the language. Additionally, the programmer can incorporate new features to the existing class, further developing the application and allowing users to achieve increased performance. This time saving feature optimizes code, helps in gaining secured applications and facilitates easier maintenance on the application.

```
#include <iostream>
class employee // Class Declaration
{
private:
char empname[50];
int empno;
public:
void getvalue()
{
cout<<"INPUT Employee Name:";
cin>>empname;
cout<<"INPUT Employee Number:";
cin>>empno;
}
void displayvalue()
{
cout<<"Employee Name:"<<empname<<endl;
cout<<"Employee Number:"<<empno<<endl;
}
};
main()
{
```

```
employee e1; // Creation of Object
e1.getvalue();
e1.displayvalue();
}
```

2.Programming Concepts:

Encapsulation

It is a mechanism that associates the code and the data it manipulates into a single unit to and keeps them safe from external interference and misuse. In C++ this is supported by construct called class. An instance of an object is known as object which represents a real world entity.

Data Abstraction

A data abstraction is a simplified view of an object that includes only features one is interested in while hides away the unnecessary details. In programming languages, a data abstraction becomes an abstract data type or a user-defined type. In OOP, it is implemented as a class.

Inheritance:

Inheritance is a means of specifying hierarchical relationships between types C++ classes can inherit both data and function members from other (parent) classes. Terminology: "the child (or derived) class inherits (or is derived from) the parent (or base) class".

Polymorphism:

Polymorphism is in short the ability to call different functions by just using one type of function call. It is a lot useful since it can group classes and their functions together. Polymorphism means that the same thing can exist in two forms. This is an important characteristic of true object oriented design - which means that one could develop good OO design with data abstraction and inheritance, but the real power of object oriented design seems to surface when polymorphism is used.

Multiple Inheritance

The mechanism by which a class is derived from more than one base class is known as multiple inheritance. Instances of classes with multiple inheritance have instance variables for each of the inherited base classes.

Programming Elements:

A Program Composed of Elements called Tokens which are collections of characters that form the basic vocabulary the compiler recognizes.

There are five kinds of tokens. They are,

- Keywords
  - Reserved Words.
  - Ex. Float, int, if ,while.
- Identifiers
  - Sequence of letters, digits and underscore.
  - Identifier cannot begin with digit.

- Literals
  - Literals are Constant values such as 1 or 3.124
  - Character literals are usually given as symbol such as ‘a’, ‘\n’
- Operators
  - Used in expressions and are meaningful when given appropriate arguments.
- Punctuators
  - It includes parantheses, braces,commas and colons and are used to structure elements of a program.

Program Structure:

Basic c++:

- A class definition begins with the keyword class.

The body of the class is contained within a set of braces, { } ; (notice the semi-*colon).*

```
class class_name
        {
        ………..
        };
```

- Within the body, the keywords *private:* and *public:* specify the access level of the members of the class.

– the default is private.

- Usually, the data members of a class are declared in the *private:* section of the class and the member functions are in *public:* section.

```
class class_name
{
private:
………
public:
………
};
```

Example:

- This class example shows how we can encapsulate (gather) a circle information into one package (unit or class)

```
class Circle
{
private:
```

```
double radius;
public:
void setRadius(double r);
double getDiameter();
double getArea();
double getCircumference();
}
```

Member access specifiers:

Access specifiers are used to identify access rights for the data and member functions of the class. There are three main types of access specifiers in C++ programming language:

private
public
protected

A *private* member within a class denotes that only members of the same class have accessibility. The *private* member is inaccessible from outside the class.

Public members are accessible from outside the class. .

A protected access specifier is a stage between *private* and *public* access. If member functions defined in a class are *protected*, they cannot be accessed from outside the class but can be accessed from the derived class.

Data Members :

Data members include members that are declared with any of the fundamental types, as well as other types, including pointer, reference, array types, bit fields, and user-defined types. You can declare a data member the same way as a variable, except that explicit initializers are not allowed inside the class definition. However, a const static data member of integral or enumeration type may have an explicit initializer.
A class X cannot have a member that is of type X, but it can contain pointers to X, references to X, and static objects of X. Member functions of X can take arguments of type X and have a return type of X.

Enumeration Types:

The keyword enum is used to declare a distinct integer type with a set of named integer constants called enumerators.
Consider the declaration:
Enum suit{clubs, diamonds,hearts,spades}

This creates an integer type with the four suit names as named integer constants. These enumerators are identifiers whose values are 0,1,2 and 3 respectively. These values are assigned by default.

Enumerators can be defined and initialized to arbitrary integer constants.

Enum suit{clubs=5, diamonds,hearts=21,spades=23}

Functions and pointers:

Function invocation:

A C++ program is made up of one or more functions, one of which is main(). Program execution always begin with main(). When program control encounters a function name, the function is invoked or called. This means the program control passes to the function.

Ex:

```
#include<iostream.h>

Void ring()

{

cout<<"Bell";

}

int main()

{

ring();

}
```

Default arguments:

```
#include<iostream.h>
 #include<iomanip.h>
long int sum(int n,int diff=1,int first_term=1 )
{
long sum=0;
 for(int i=0;i<n;i++)
{
cout<<setw(5)<<first_term+ diff*i;
 sum+=first_term+diff*i;
 }
return sum;
}
int main()
{
cout<<endl<<\"Sum=\"<<setw(7)< <sum(10)<<endl;
cout<<endl<<\"Sum=\"<<setw(7)< <sum(6,3,2)<<endl;
return 1;
}
```

all the parameters with default values should lie to the right in the signature list i.e. the default arguments should be the trailing arguments—those at the end of the list. when a function with default arguments is called, the first argument in the call statement is assigned to the first argument in the definition, the 2nd to 2nd and so on.

<u>Overloading functions:</u>

*C++ supports writing more than one function with the same name but different argument lists. This could include:

– different data types
– different number of arguments

*The advantage is that the same apparent function can be called to perform similar but different tasks. The following will show an example of this.

```
void swap (int *a, int *b) ;
void swap (float *c, float *d) ;
void swap (char *p, char *q) ;
int main ( )
{
int a = 4, b = 6 ;
float c = 16.7, d = -7.89 ;
char p = 'M' , q = 'n' ;
swap (&a, &b) ;
swap (&c, &d) ;
swap (&p, &q) ;
}
void swap (int *a, int *b)
```

```
{
int temp;
temp = *a;
*a = *b;
*b = temp;
}
void swap (float *c, float *d)
{
float temp;
temp = *c;
*c = *d;
*d = temp;
}
void swap (char *p, char *q)
{
char temp;
temp = *p;
*p = *q;
 *q = temp;
}
```

Scope and storage class:

The kernel language has two principal forms of scope

- Local Scope
  - Local scope is scoped to a block.
  - Ex. Function body
- File Scope
  - It has names that are external (global).

Storage Class:

Storage class defined for a variable determines the accessibility and longevity of the variable. The accessibility of the variable relates to the portion of the program that has access to the variable. The longevity of the variable refers to the length of time the variable exists within the program.

Types of Storage Class Variables in C++:

- Automatic
- External
- Static
- Register

Automatic:

Variables defined within the function body are called automatic variables. Auto is the keyword used to declare automatic variables. By default and without the use of a keyword, the variables defined inside a function are automatic variables.

External:

External variables are also called global variables. External variables are defined outside any function, memory is set aside once it has been declared and remains until the end of the program. These variables are accessible by any function. This is mainly utilized when a programmer wants to make use of a variable and access the variable among different function calls.

Static:

The static automatic variables, as with local variables, are accessible only within the function in which it is defined. Static automatic variables exist until the program ends in the same manner as external variables. In order to maintain value between function calls, the static variable takes its presence.

Register:

The register variable are stored in a high speed registers. It is a request to the compiler.

Ex:

```
Void main()

{

Auto int a;

Static int b,c;

Register float x=3.5;

Extern pi=3.1;

}
```

Pointer types:

A pointer is a variable that is used to store a memory address. The address is the location of the variable in the memory. Pointers help in allocating memory dynamically. Pointers improve execution time and saves space. Pointer points to a particular data type. The general form of declaring pointer is:-

```
type *variable_name;
```

type is the base type of the pointer and variable_name is the name of the variable of the pointer. For example,

```
int *x;
```

x is the variable name and it is the pointer of type integer.

Pointer Operators

There are two important pointer operators such as '*' and '&'. The '&' is a unary operator. The unary operator returns the address of the memory where a variable is located. For example,

```
int x*;
int c;
x=&c;
```

variable x is the pointer of the type integer and it points to location of the variable c. When the statement

```
x=&c;
```

is executed, '&' operator returns the memory address of the variable c and as a result x will point to the memory location of variable c.

The '*' operator is called the indirection operator. It returns the contents of the memory location pointed to. The indirection operator is also called deference operator. For example,

```
int x*;
int c=100;
int p;
```

```
x=&c;

p=*x;
```

variable x is the pointer of integer type. It points to the address of the location of the variable c. The pointer x will contain the contents of the memory location of variable c. It will contain value 100. When statement

```
p=*x;
```

is executed, '*' operator returns the content of the pointer x and variable p will contain value 100 as the pointer x contain value 100 at its memory location. Here is a program which illustrates the working of pointers.

```
#include<iostream.h>

int main ()

{

        int *x;

        int c=200;

        int p;

        x=&c;

        p=*x;

        cout << " The address of the memory location of x : " << x << endl;

        cout << " The contents of the pointer x : " << *x << endl;

        cout << " The contents of the variable p : " << p << endl;

        return(0);

}
```

Arrays and pointers:

An array name is really a pointer to the first element of the array. For example, the following is legal.

```
int b[100];    // b is an array of 100 ints.
```

```
int* p;         // p is a pointer to an int.
p = b;          // Assigns the address of first element of b to p.
p = &b[0];      // Exactly the same assignment as above.
```

Array name is a const pointer

When you declare an array, the name is a *pointer*, which cannot be altered. In the previous example, you could never make this assignment.

```
   p = b;    // Legal -- p is not a constant.
   b = p;    // ILLEGAL because b is a constant, although the correct type.
```

Pointer addition and element size

When you add an integer to a pointer, the integer is *multiplied by the element size* of the type that the pointer points to.

```
// Assume sizeof(int) is 4.
int b[100];  // b is an array of 100 ints.
int* p;      // p is a a pointer to an int.
p = b;       // Assigns address of first element of b. Ie, &b[0]
p = p + 1;   // Adds 4 to p (4 == 1 * sizeof(int)). Ie, &b[1]
```

Equivalence of subscription and dereference

Because of the way C/C++ uses pointers and arrays, you can reference an array element either by subscription or * (the unary dereference operator).

```
int b[100];  // b is an array of 100 ints.
int* p;      // p is a a pointer to an int.
p = b;       // Assigns address of first element of b. Ie, &b[0]
*p = 14;     // Same as b[0] = 14
p = p + 1;   // Adds 4 to p (4 == 1 * sizeof(int)). Ie, &b[1]
*p = 22;     // Same as b[1] = 22;
```

Example - Two ways to add numbers in an array

The first uses subscripts, the second pointers. They are equivalent.

```
int a[100];
. . .
int sum = 0;
for (int i=0; i<100; i++) {
```

```
int a[100];
. . .
int sum = 0;
for (int* p=a; p<a+100; p++) {
    sum += a[i];
}
```

```
    sum += *p;
}
```

Call-by-reference:

The **call by reference** method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument.

To pass the value by reference, argument pointers are passed to the functions just like any other value. So accordingly you need to declare the function parameters as pointer types as in the following function **swap()**, which exchanges the values of the two integer variables pointed to by its arguments.

```
// function definition to swap the values.
void swap(int *x, int *y)
{
   int temp;
   temp = *x; /* save the value at address x */
   *x = *y; /* put y into x */
   *y = temp; /* put x into y */

   return;

}

#include <iostream.h>

// function declaration
void swap(int *x, int *y);

int main ()
{
   // local variable declaration:
   int a = 100;
   int b = 200;

   cout << "Before swap, value of a :" << a << endl;
   cout << "Before swap, value of b :" << b << endl;

   /* calling a function to swap the values.
    * &a indicates pointer to a ie. address of variable a and
    * &b indicates pointer to b ie. address of variable b.
    */
   swap(&a, &b);

   cout << "After swap, value of a :" << a << endl;
   cout << "After swap, value of b :" << b << endl;
   return 0;
}
```

Assertions

Asserts are to catch implementation errors. The developer can document all the assumptions made on the his/her program, by using ASSERTs. If you have used asserts in your code and if it triggers, then you can know, there is a problem in your code and needs to be solved. And thus ASSERTs are to find bugs in your code.

When to use ASSERTs

Developers should make it a practice to use Asserts wherever it is needed to test :
1. Validity of function arguments before using them
2. Validity of a pointer before using it
3. Testing any condition that you think, Must be True i.e. checking the values of a variable at some point in code, checking the expected range at some point etc.

Syntax:

```
void assert (expression);
```

Expression to be evaluated. If this expression evaluates to `0`, this causes an *assertion failure* that terminates the program.

```
/* assert example */
#include <stdio.h>
#include <assert.h>

void print_number(int* myInt) {
  assert (myInt!=NULL);
  printf ("%d\n",*myInt);
}

int main ()
{
  int a=10;
  int * b = NULL;
  int * c = NULL;

  b=&a;

  print_number (b);
  print_number (c);
  return 0;
}
```

Standard Template Library:

The **Standard Template Library** (**STL**) is a C++ software library which heavily influenced many parts of the C++ Standard Library. It provides four components called *algorithms*, *containers*, *functional*, and *iterators*.

The STL provides a ready-made set of common classes for C++, such as containers and associative arrays, that can be used with any built-in type and with any user-defined type that supports some elementary operations (such as copying and assignment). STL algorithms are independent of containers, which significantly reduces the complexity of the library.

Containers

The STL contains sequence containers and associative containers. The standard sequence containers include `vector`, `deque`, and `list`. The standard associative containers are `set`, `multiset`, `map`, and `multimap`. There are also *container adaptors* `queue`, `priority_queue`, and `stack`, that are containers with specific interface, using other containers as implementation.

Iterators

The STL implements five different types of iterators. These are *input iterators* (that can only be used to read a sequence of values), *output iterators* (that can only be used to write a sequence of values), *forward iterators* (that can be read, written to, and move forward), *bidirectional iterators* (that are like forward iterators, but can also move backwards) and *random access iterators* (that can move freely any number of steps in one operation).

It is possible to have bidirectional iterators act like random access iterators, as moving forward ten steps could be done by simply moving forward a step at a time a total of ten times. However, having distinct random access iterators offers efficiency advantages. For example, a vector would have a random access iterator, but a list only a bidirectional iterator.

Algorithms

A large number of algorithms to perform operations such as searching and sorting are provided in the STL, each implemented to require a certain level of iterator (and therefore will work on any container that provides an interface by iterators).