C++

Naresh Limba

Table of Contents

Introduction

C++ is a general purpose, case-sensitive, free-form programming language that supports object-oriented, procedural and generic programming.

C++ is a middle-level language, because it encapsulates both high and low-level language features.

C++ programming language was developed in 1980 by **Bjarne Stroustrup** at AT&T (American Telephone & Telegraph) Bell Laboratories, located in U.S.A. Bjarne Stroustrup is known as the founder of C++ language.

C++ was developed for adding a feature of OOP (Object Oriented Programming) in C without significantly changing the C component. C++ programming is a superset of C, so we can say that any valid C program is also a valid C++ program.

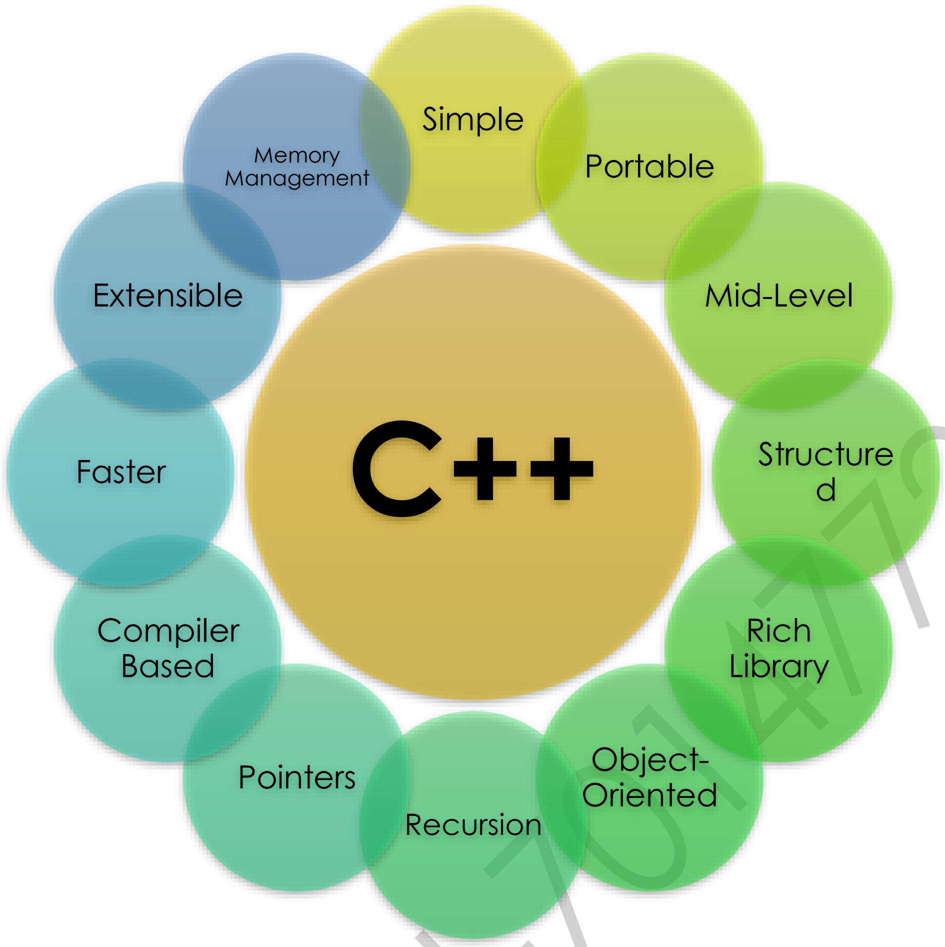
Difference between C and C++

No.	C	C++
1	C is procedural oriented programming language.	C++ is multi-paradigm. It supports both procedural and object oriented.
2	Less security of data in C.	In C++, we can use modifiers for class members to make it inaccessible for outside users.
3	C follows the top-down approach.	C++ follows the bottom-up approach.
4	C does not support **function overloading.**	C++ supports function overloading.
5	In C, you can't use functions in structure.	In C++, you can use functions in structure.
6	C does not support reference variables.	C++ supports reference variables.
7	Operator overloading is not possible in C.	Operator overloading is possible in C++.
8	C programs are divided into procedures and modules	C++ programs are divided into functions and classes.
9	C does not provide the feature of **namespace**.	C++ supports the feature of **namespace**.
10	Exception handling is not easy in C. It has to perform using other functions.	C++ provides exception handling using Try and Catch block.

Table: Differences between C and C++

Feature of C++

As we know C++ is object-oriented programming language so it supports a lot of features as given below-



1) Simple

C++ is a simple language. It provides structured approach (to break the problem into parts), rich set of library functions, data types etc.

2) Machine Independent or Portable

C programs can be executed in many machines with little bit or no change. But it is not platform-independent.

3) Mid-Level Programming Language

C++ is also used to do low level programming. It is used to develop system applications such as kernel, driver etc. It also supports the feature of high-level language. That is why it is known as mid-level language.

4) Structured Programming Language

C++ is a structured programming language in the sense that we can break the program into parts using functions. So, it is easy to understand and modify.

5) Rich Library

C++ provides a lot of inbuilt functions that makes the development fast.

6) Memory Management

It supports the feature of dynamic memory allocation. In C++ language, we can free the allocated memory at any time by calling the free() function.

7) Fast Speed

The compilation and execution time of C++ language is fast.

8) Pointers

C++ provides the feature of pointers. We can directly interact with the memory by using the pointers. We can use pointers for memory, structures, functions, array etc.

9) Recursion

In C++, we can call the function within the function. It provides code reusability for every function.

10) Extensible

C++ language is extensible because it can easily adopt new features.

11) Object Oriented

C++ is object-oriented programming language. OOPs makes development and maintenance easier where as in Procedure-oriented programming language it is not easy to manage if code grows as project size grows.

12) Compiler Based

C++ is a compiler-based programming language, it means without compilation no C++ program can be executed. First, we need to compile our program using compiler and then we can execute our program.

Program in C++ and it's structure

```
#include<iostream.h>
#include<conio.h>
void main()
{
      clrscr();
      cout<<"First Program in C++ Programming";
      getch();
}
```

Output:

```
First Program in C++ Programming
```

#include<iostream.h>

Includes the standard input output library functions. It provides cin and cout methods for reading from input and writing to output respectively.

#include<conio.h>

includes the console input output library functions. The **getch()** and **clrscr()** function is defined in **conio.h** file.

`void main()`

The main() function is the entry point of every program in C++ language. The void keyword specifies that it returns no value.

`cout <<" First Program in C++ Programming";`

It is use to print the data "Welcome to C++ Programming." on the console.

`getch()`

The getch() function asks for a single character. Until you press any key, it blocks the output screen.

Basic Input/Output in C++

C++ I/O operation is using the stream concept. Stream is the sequence of bytes or flow of data. It makes the performance fast.

If bytes flow from main memory to device like printer, display screen, or a network connection, etc., this is called as **output operation**.

If bytes flow from device like printer, display screen, or a network connection, etc. to main memory, this is called as **input operation**.

Standard Output Stream (cout)

The **cout** is a predefined object of ostream class. It is connected with the standard output device, which is usually a display screen. The **cout** is used with **stream insertion operator** (<<) to display the output on a console-

```
#include<iostream.h>
void main()
{
	int a;
	a=10;
	cout<<"Value of a is: "<<a<<endl;
}
```

Output:

```
Value of a is: 10
```

Standard Input Stream (cin)

The **cin** is a predefined object of **istream** class. It is connected with the standard input device, which is usually a keyboard. The **cin** is used with **stream extraction operator** (>>) to read the input from a console-

```
#include<iostream.h>
void main()
{
	int a;
	cout<<"Enter a value :";
	cin>>a;
	cout<<"Value of a is: "<<a<<endl;
}
```

Output:

```
Enter a value: 55
Value of a is: 55
```

Standard end line (endl)

The **endl** is a predefined object of ostream class. It is used to insert a new line characters and flushes the stream.

```
#include<iostream.h>
void main()
{
	cout<<"Learning C++ ";
	cout<<"is very interesting."<<endl;
	cout<<"Thanks"<<endl;
}
```

Output:

```
Learning C++ is very interesting.
Thanks

_
```

Variable in C++

A variable is a name of memory location. It is used to store data. Its value can be changed and it can be reused many times. It is a way to represent memory location through symbol so that it can be easily identified.

Syntax for variable declaration: `type variable(s);`

For Example:

```
int a;
float b;
char c;
```

Here! a, b, c are variables and int, float, char are data types.

Rules to declare Variables:

- ✓ A variable can have alphabets, digits and underscore.
- ✓ A variable name can start with alphabet and underscore only. It can't start with digit.
- ✓ No white space is allowed within variable name.
- ✓ A variable name must not be any reserved word or keyword e.g. char, float etc.

Valid variable names:

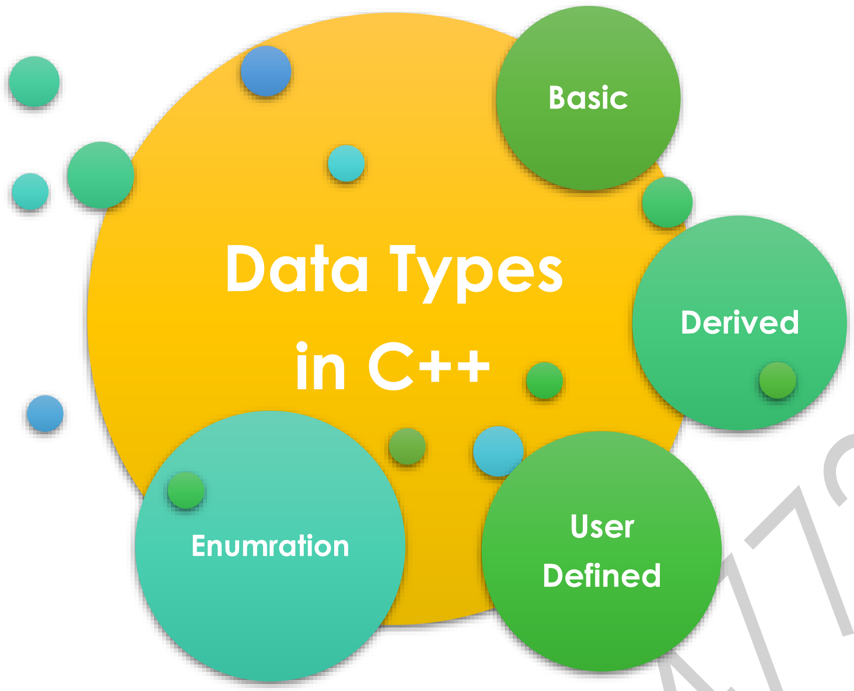
```
int a;
int _ab;
int a30;
```

Invalid variable names:

```
int 4;
int x y;
int double;
```

Data Types C++

A data type specifies the type of data that a variable can store such as integer, floating, character etc.



There are 4 types of data types in C++ language-

Type	Data Type
Basic Data Type	int, char, float, short, long, double etc.
Derived Data Type	array, pointer
Enumeration Data Type	enum
User Defined Data Type	structure, union, class

Table: Data Types

Keywords in C++

A keyword is a reserved word. You cannot use it as a variable name, constant name etc. A list of 32 Keywords in C++ Language which are also available in C language are given below.

`auto`	`break`	`case`	`char`	`const`	`continue`	`default`	`Do`
`double`	`else`	`enum`	`extern`	`float`	`for`	`goto`	`If`
`int`	`long`	`register`	`return`	`short`	`signed`	`sizeof`	`Static`
`struct`	`switch`	`typedef`	`union`	`unsigned`	`void`	`volatile`	`While`

A list of 30 Keywords in C++ Language which are not available in C language are given below.

asm	dynamic_cast	namespace	reinterpret_cast	bool
explicit	new	static_cast	false	catch
operator	template	friend	private	class
this	inline	public	throw	const_cast
delete	mutable	protected	true	try
typeid	typename	using	virtual	wchar_t

Operators in C++

An operator is simply a symbol that is used to perform operations. There can be many types of operations like arithmetic, logical, bitwise etc. There are following types of operators to perform different types of operations in C language.

1) Arithmetic Operators
2) Relational Operators
3) Logical Operators
4) Bitwise Operators
5) Assignment Operator
6) Unary operator
7) Ternary or Conditional Operator
8) Misc Operator

Binary Operator	+, -, *, /, %	Arithmetic Operator
	<, <=, >, >=, ==, !=	Relational Operator
	&&, | |, !	Conditional Operator
	&, |, <<, >>, ~, ^	Bitwise Operator
	=, +=, -=, *=, /=, %=	Assignment Operator
Unary Operator	++, --	Unary Operator
Ternary Operator	? :	Ternary Operator

++ if-else

C++ switch

C++ Array

C++ For Loop

C++ While Loop

C++ Do-While Loop

C++ Functions

C++ Break Statement

C++ Continue Statement

C++ Goto Statement

C++ Comments

Please refer to reference of C to recover all these topics.

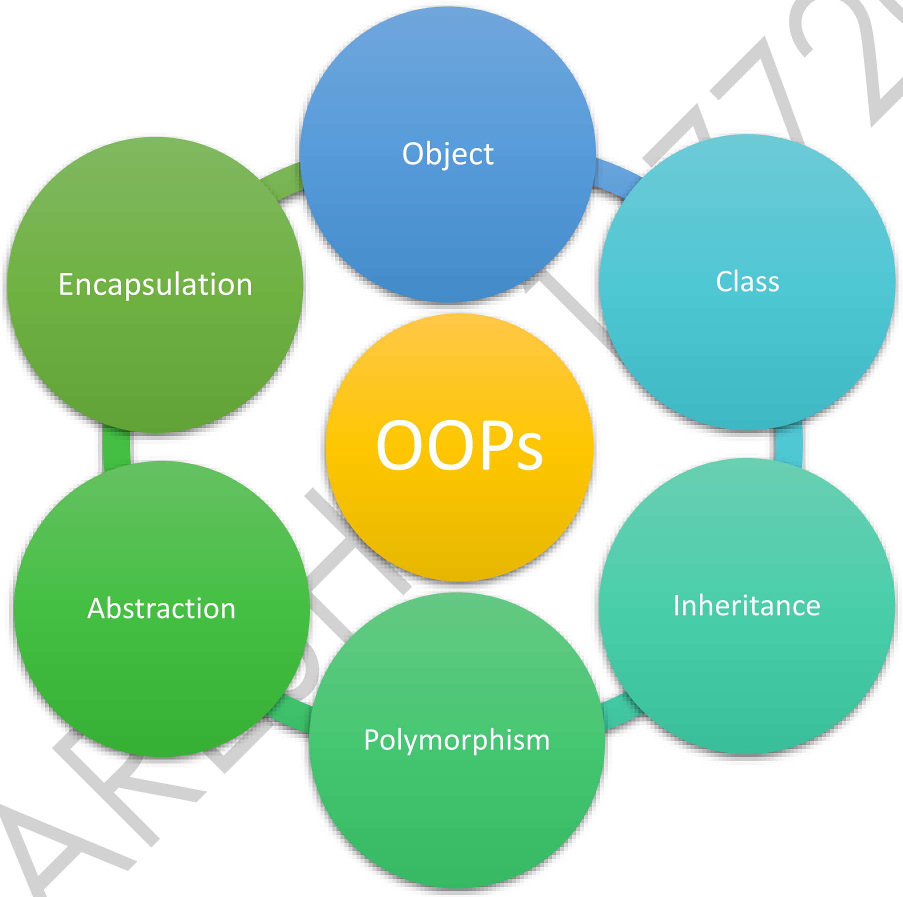
OOPs Concepts in C++

The major purpose of C++ programming is to introduce the concept of object orientation to the C programming language. Object Oriented Programming provides many concepts such as inheritance, data binding, polymorphism etc.

The programming paradigm where everything is represented as an object is known as truly object-oriented programming language. Smalltalk is considered as the first truly object-oriented programming language.

OOPs (Object Oriented Programming System)

Object means a real word entity such as pen, chair, table etc. Object-Oriented Programming is a methodology or paradigm to design a program using classes and objects. It simplifies the software development and maintenance by providing some concepts:



1) Object

Any entity that has **state** and **behavior** is known as an object. For example: chair, pen, table, keyboard, bike etc. It can be **physical and logical**.

2) Class

Collection of objects is called class. It is a ***logical entity***.

3) Inheritance

When one object acquires all the properties and behaviors of parent object is known as inheritance. It provides ***code reusability***. It is used to achieve ***runtime polymorphism***.

4) Polymorphism

When one task is performed by **different ways** is known as polymorphism. In C++, we use ***Function overloading*** and ***Function overriding*** to achieve polymorphism.

5) Abstraction

Hiding internal details and showing functionality is known as abstraction. For example: phone call, we don't know the internal processing.

In C++, we use abstract class to achieve abstraction.

6) Encapsulation

Binding (or wrapping) code and data together ***into a single unit*** is known as encapsulation. For example: capsule, it is wrapped with different medicines.

Advantage of OOPs over POP language

OOPs	POPs
OOPs makes development and maintenance easier.	In *Procedure-oriented programming* language it is not easy to manage if code grows as project size grows.
OOPs provide data hiding	In Procedure-oriented programming language a global data can be accessed from anywhere.
We can provide the solution of real word problem using OOPs.	It is not possible to solve real life problems using POPs.

Table: OOPs vs POPs

Functions in C++

A function is a collection of statements that can perform a task together. Every C++ program has at least one function, which is ***main()***, and we can define additional functions as over requirement. We can divide up our code into separate functions. We can define functions according to our choice to perform a task or a set of different tasks.

A **function declaration** tells the compiler about a function's name, return type, and parameters. A **function definition** provides the actual body of the function. There are two types of function are available in **C++,** these are following:

1) In-built/Library Functions
2) User define Functions

In-built/Library Functions

These functions are declared in the C++ header files such as getline(), get(), put(), write() etc.

User Define Function

User define function are defined by programmer. We define the function to reduce the complexity of program and to reuse the code that is required many times in a program.

In C++, there are following 3 important part of any function:

1) Declaration
2) Definition
3) Calling

1) Declaration

A function declaration tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately. A function declaration has the following parts –

```
return_type function_name(parameter_list);
```

2) Definition

A C++ function definition consists of a function header and a function body. Here are all the parts of a function –

Return Type – A function may return a value. The `return_type` is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the `return_type` is the keyword `void`.

Function Name – This is the actual name of the function. The **function name** and the parameter list together construct the function signature.

Parameters – A parameter is like a placeholder. When a function is invoked, we pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

Function Body – The function body contains a collection of statements that define what the function does.

3) Calling

While creating a C++ function, we give a definition of what the function has to do. To use a function, we will have to **call or invoke that function**.
When a program calls a function, program control is transferred to the called function. A called function performs defined task and when it's return statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program.

For Example: returning max number from given values.

```
#include <iostream.h>
int max(int n, int m);          //Function Declaration
int max(int n, int m)           //Function Definition
{
      if(n>m)
      {
            return n;
      }
      else
      {
            return m;
      }
}
void main()
{
      int a,b,r;
      a=12;
      b=33;
      r=max(a,b);           //Function Calling with passing two arguments
      cout<<"Max number is : "<<r;
}
```

Output:

```
Max number is : 33_
```

Default Argument

It means make one or more argument of a function default. So that in case of missing argument in function call, default arguments will automatically take place for those missed arguments.

NOTE: We can make default arguments default from right to left because arguments can be absent from right to left.

For Example:

```
#include<iostream.h>
class Demo
{
      private:
      int p;
      float q;
      public:
      void setDat(int a=0, float x=0.0)       //Assigning Default Arguments
      {
            p=a;
            q=x;
      }
      void dispDat()
      {
            cout<<"\nValue of P: "<<p;
            cout<<"\nValue of Q: "<<q;
      }
};
void main()
{
      Demo A,B;
      A.setDat();
      A.dispDat();
      B.setDat(2,5.99);
      B.dispDat();
}
```

We missed passing arguments in this function call, so in this situation default arguments will be treated as function arguments.

Output:

```
Value of P: 0
Value of Q: 0.0
Value of P: 2
Value of Q: 5.99
```

Objects & Classes

The main purpose of C++ programming is to add object orientation to the C programming language and classes are the central feature of C++ that supports object-oriented programming and are often called ***user-defined*** types. A ***class*** is used to specify the form of an ***object*** and it combines data representation and methods for manipulating that data into one package. The data and functions within a class are called members of the class.

Class in C++

A class is an abstract data type similar to 'C structure'. The Class representation of objects and the sets of operations that can be applied to such objects. The class consists of **Data members** and **methods**.

NOTE: private, public, protected are the visibility labels.

Members of a class: *Data and functions are member.*

Syntax:

```
class <class_name>
{
      variable/ Data Member declaration;
      Methods;
}
```

For Example:

```
#include<iostream.h>
class Demo
{
      private:
      int a,b;
      public:
      void setVal()
      {
            a=b=10;
      }
      void dispDat()
      {
            cout<<a<<endl<<b;
      }
};
```
Output:

```
10
10
```

C++ Object

A class provides the blueprints for objects, so basically an object is created from a class. Objects are instances of class. We declare objects of a class with exactly the same way as we declare variables of basic types.

Syntax:

```
      class_name object1, object2;
```

For Example:

```
#include<iostream.h>
class Demo
{
      private:
      int a,b;
      public:
      void setVal()
      {
            a=b=10;
      }
      void dispDat()
      {
            cout<<a<<endl<<b;
      }
};
void main()
{
      Demo D;               //Object Declaration of class
      D.setVal();           //calling member function
      D.dispDat();          //calling member function
}
```
Output:

```
10
10_
```

Accessing the Data Members and Member Functions

The public data members of objects of a class can be accessed using the **direct member access operator** (.).

In above example we have access public member function of class **Demo**, later we will learn how to access private and protected data members.

Function Definition Outside of Class

In C++, we can define member function outside of a class also. To do this we have to use scope resolution operator for that member function.

Syntax:

```
return_type class_name :: function_name(parameters)
{
        Statement(s);
}
```

For Example:

```
#include <iostream.h>
class Demo
{
        int id,age;
        public:
        void getDat();
        void dispDat();
};
void Demo :: getDat()
{
        cout<<"Enter id and age : ";
        cin>>id>>age;
}
void Demo :: dispDat()
{
        cout<<"\nID: "<<id<<"\tAge: "<<age;
}
void main()
{
        Demo D;
        D.getDat();
        D.dispDat();
}
```

Output:

```
Enter id and age: 121
22
ID: 121     Age: 22_
```

Passing Object as Function argument

In C++ it is possible to pass an object as function argument to a class. To perform this approach, we can follow the same approach as we pass other variables.

For Example:

```
#include<iostream.h>
class Demo
{
        private:
        int a,b;
        public:
        void setVal()
        {
                a=b=10;
        }
        void copyVal(Demo A)
        {
                a=A.a;
                b=A.b;
        }
        void dispDat()
        {
                Cout<<a<<endl<<b;
        }
};
void main()
{
        Demo A,B;
        A.setVal();
        B.copyVal(A);       //Passing object (A)  as argument to a function.
        cout<<"After copy value from object A to B"<<endl;
        B.dispDat();
}
```

Output:

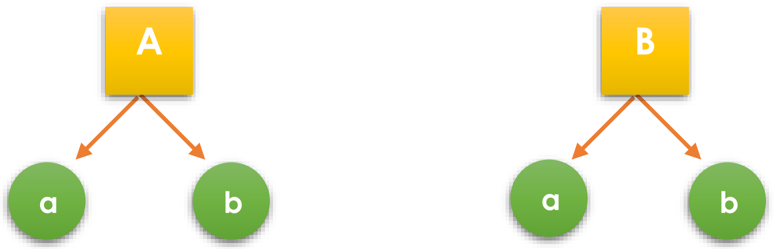
```
After copy value from object A to B
10
10_
```

Some Highlights about Object

- Each object contains its own field/ data member.
- Two different objects may have two different records, and each record is a collection of related fields/ data members.
- Method calling via object is a process used to initialized or access data member of an object.

For Example:

```
#include<iostream.h>
class Demo
{
        private:
        int a,b;
        public:
        void setVal()
        {
                cout<<"Enter value for a and b: ";
                cin>>a>>b;
        }
        void dispDat()
        {
                Cout<<"Values are: "<<a<<" and "<<b;
        }
};
void main()
{
        Demo A,B;
        A.setVal();
        A.dispDat();
        B.setVal();
        B.dispDat();
}
```



Output:

```
Enter values for a and b: 17
15
Values are 17 and 15
Enter values for a and b: 1
5
Values are 1 and 5
```

In above example **A** and **B** are two objects of class **Demo**. Object A have different values for data member a, b and object B have also different values for data member a, b.

Adding values of two objects

We can also perform calculation on values of objects stored by the respective object of the class.

For Example:

```
#include<iostream.h>
class Demo
{
      private:
      int a,b;
      public:
      void setVal()
      {
            cout<<"Enter value for a and b: ";
            cin>>a>>b;
      }
      void sumVal(Demo A, Demo B)
      {
            a=A.a+B.a;
            b=A.b+B.b;
      }
      void dispSum()
      {
            cout<<"Sum of A->a + B->a is :"<<a;
            cout<<"\nSum of A->b + B->b is :"<<b;
      }
};
void main()
{
      Demo A,B,C;
      A.setVal();
      B.setVal();
      C.sumVal(A,B);
      C.dispSum();
}
```

Output:

```
Enter value for a and b: 2
8
Enter value for a and b: 12
4
Sum of A->a + B->a is :14
Sum of A->b + B->b is :12_
```

Returning Object

Let see the following example to figure out how we can return object to its call.

For Example:

```
#include<iostream.h>
class Demo
{
	private:
	int a,b;
	public:
	void setVal()
	{
		cout<<"Enter value for a and b: ";
		cin>>a>>b;
	}
	Demo sumVal(Demo B)
	{
		Demo X;
		X.a=a+B.a;
		X.b=b+B.b;
		return X;
	}
	void dispSum()
	{
		cout<<"Sum of A->a + B->a is :"<<a;
		cout<<"\nSum of A->b + B->b is :"<<b;
	}
};
void main()
{
	Demo A,B,C;
	A.setVal();
	B.setVal();
	C=A.sumVal(B);
	C.dispSum();
}
```

Output:

```
Enter value for a and b: 2
8
Enter value for a and b: 12
4
Sum of A->a + B->a is :14
Sum of A->b + B->b is :12_
```

Array of Object

In C++ we can also declare array of any object same as normal array of other data types.

For Example:

```
#include<iostream.h>
class Demo
{
      private:
      int rn,age;
      public:
      void getDat()
      {
            cout<<"Enter your Roll No: ";
            cin>>rn;
            cout<<"Enter your age: ";
            cin>>age;
      }
      void dispDat()
      {
            cout<<rn<<"\t"<<age;
      }
};
void main()
{
      Demo A[2];              //Array of Object
      int i;
      for(i=0;i<=1;i++)
      {
            A[i].getDat();
      }
      cout<<"RollNo\tAge";
      for(i=0;i<=1;i++)
      {
            A[i].dispDat();
      }
}
```

Output:

```
Enter you Roll No: 11
Enter you age: 20;
Enter you Roll No: 12
Enter you age: 22;
RollNo      Age
11          20
12          22_
```

Static Keyword

In C++, static is a keyword or modifier that belongs to the type not instance. So instance is not required to access the static members. In C++, static can be field, method, constructor, class, properties, operator and event.

Static Variable

All **Static Members** are initialized with zero (0) by default. Each object has separate copy of class members but **Static members** are common for all objects, means all object can access and modify the value of static member.

For Example: 1st Method- (Counting of a function calling)

```
#include<iostream.h>
static int x;
class Demo
{
	private:
	int a,b;
	public:
	void getDat()
	{
		x++;
		cout<<"Enter two values: ";
		cin>>a>>b;
	}
	void dispDat()
	{
		cout<<"Function call: "<<x;
		cout<<"\tValues are: "<<a<<"\t"<<b;
	}
};
void main()
{
	Demo A,B;
	A.getDat();
	A.dispDat();
	B.getDat();
	B.dispDat();
}
```

Output:

```
Enter two values: 2
3
Function call: 1        Values are: 2     3
Enter two values: 4
7
Function call: 2        Values are: 4     7
```

For Example: 2nd Method- (Counting of Objects created)

```
#include<iostream.h>
class Demo
{
      private:
      int a,b;
      static int x;
      public:
      void getDat()
      {
            x++;
            cout<<"Enter two values: ";
            cin>>a>>b;
      }
      void dispDat()
      {
            cout<<"Function call: "<<x;
            cout<<"\tValues are: "<<a<<"\t"<<b;
      }
};
int Demo::x;
void main()
{
      Demo A,B;
      A.getDat();
      A.dispDat();
      B.getDat();
      B.dispDat();
}
```

Output:

```
Enter two values: 2
3
Function call: 1        Values are: 2     3
Enter two values: 4
7
Function call: 2        Values are: 4     7
```

this Pointer

In C++ programming, **this** is a keyword that refers to the current instance of the class. There can be 3 main usage of **this** keyword in C++ are given below-

- It can be used to pass current object as a parameter to another method.
- It can be used to refer current class instance variable.
- It can be used to declare indexers.

For Example:

```
#include <iostream.h>
class Employee
{
	public:
	int id;
	float salary;
	void setDat(int id, float salary)
	{
		this->id = id;
		this->salary = salary;
	}
	void dispDat()
	{
		cout<<"ID\tSalary"<<endl;
		cout<<id<<"\t"<<salary<<endl;
	}
};
void main()
{
	Employee E,F;
	E.setDat(101, 8900);
	F.setDat(102, 9200);
	E.dispDat();
	F.dispDat();
}
```

Output:

```
ID    Salary
101   8900
102   9200
_
```

Friend Function

If a function is defined as a friend function in C++ then the protected and private data of a class can be accessed using the friend function. By using the keyword **friend** compiler knows the given function is a **friend function**.

Points to remember about Friend Function:

- It can be member of one or more than one classes.
- It must receive objects of associated class to work on it.
- We can directly call the function with its name.
- Friend function must be defined outside of the classes.
- It must be declared or signature within all the classes.

For Example:[1]

```
#include <iostream.h>
class Person;
class Employee
{
	int id;
	float sal;
	public:
	void proDat()
	{
		cout<<"Enter Your ID: ";
		cin>>id;
		cout<<"Enter Your Salary: ";
		cin>>sal;
	}
	friend void dispDat(Employee E, Person P);
};
class Person
{
	int a;
	char n[20];
	public:
	void perDat()
	{
		cout<<"Enter Your Name: ";
		cin>>n;
		cout<<"Enter Your Age: ";
		cin>>a;
	}
	friend void dispDat(Employee E, Person P);
};
void dispDat(Employee E, Person P)
{
	cout<<"Name\tAge\tID\tSalary"<<endl;
	cout<<P.n<<"\t"<<P.a<<"\t"<<E.id<<"\t"<<E.sal;
}
void main()
{
	Employee E;
	E.proDat();
	Person P;
	P.perDat();
	dispDat(E,P);
}
```

Output:

```
Enter Your Id: 11
Enter Your Salary: 12000
Enter Your Name: Naresh
Enter Your Age: 24
Name        Age   ID    Salary
Naresh      24    11    12000
```

NOTE: *Using Friend Function, we can also access private data member of a class.*

NOTE: *To input or edit value using friend function we must pass the refence of objects.*

For Example: [2]- Swapping values of two different classes.

```
#include <iostream.h>
class Two;
class One
{
      int a;
      public:
      void setDat(int x)
      {
            a=x;
            cout<<"\nBefore swap a is: "<<a;
      }
      void show()
      {
            cout<<"\nAfter swap a is: "<<a;
      }
      friend void swapVal(One &O, Two &T);
};
class Two
{
      int b;
      public:
      void setData(int x)
      {
            b=x;
            cout<<"\nBefore swap b is: "<<b;
      }
      void disp()
      {
            cout<<"\nAfter swap b is: "<<b;
      }
      friend void swapVal(One &O, Two &T);
};
void swapVal(One &O, Two &T)
{
      int i;
      i=O.a;
      O.a=T.b;
      T.b=i;
}
void main()
{
      One O;
      Two T;
      O.setDat(11);
      T.setDat(66);
      swapVal(O,T);
      O.show();
      T.disp();
}
```

Output:

```
Before swap a is: 11
Before swap b is: 66
After swap a is: 66
After swap b is: 11
```

Constructor

In C++, constructor is a special method which is invoked automatically when the object is created. It is used to initialize the data members of new object generally. The constructor has the same name as class name.

Points to remember about Constructor:

- Constructor is always public.
- Constructor is special member function.
- Constructor name is same as class name.
- It can be overloaded.
- There is no need to invoke constructor because they are invoked automatically when the object is created.
- Constructor has not any return type even void.

For Example:

```
#include<iostream.h>
class Demo
{
      private:
      int rn,age;
      public:
      Demo()
      {
            rn=121;
            age=24;
      }
      void dispDat()
      {
            cout<"Roll No: "<<rn<<"\nAge: "<<age;
      }
};
void main()
{
      Demo A;             //constructor will invoke because object is created.
      A.dispDat();
}
```

Output:

```
Roll No: 121
Age: 24_
```

Types of Constructor:

In C++ programming there are following types of constructor-

1) Default Constructor
2) Parameterized Constructor
3) Copy Constructor

1) Default Constructor

A constructor which has no argument is known as default constructor. It is invoked at the time of creating object.

For Example:

```
#include<iostream.h>
#include<conio.h>
class Employee
{
	public:
	Employee()
	{
		cout<<"Default Constructor Invoked"<<endl;
	}
};
void main()
{
	Employee E;		//creating an object of Employee
	Employee F;
}
```

Output:

```
Default Constructor Invoked
Default Constructor Invoked
```

2) Parameterized Constructor:

A constructor which has parameters is called parameterized constructor. It is used to provide different values to separate objects.

For Example:

```
#include <iostream.h>
class Employee
{
	int id,age;
	public:
	Employee(int i, int j)		//Parameterized Constructor
	{
		id=i;
		age=j;
	}
	void dispDat()
	{
		cout<<"\nID: "<<id<<"\tAge: "<<age;
```

```
        }
};
void main()
{
        Employee E(205,24);          //creating an object of Employee
        E.dispDat();
        Employee F(235,22);          //creating an object of Employee
        F.dispDat();
}
```

Output:

```
ID: 205     Age: 24

ID: 235     Age: 22
```

3) Copy Constructor:

The copy constructor is a constructor which creates an object by initializing it with an object of the same class, which has been created previously. The copy constructor is used to:

- Initialize one object from another of the same type.
- Copy an object to pass it as an argument to a function.
- Copy an object to return it from a function.

If a copy constructor is not defined in a class, the compiler itself defines one. If the class has pointer variables and has some dynamic memory allocations, then it is a must to have a copy constructor.

For Example:

```
#include <iostream.h>
class Employee
{
        int id,age;
        public:
        Employee()                          //Default Constructor
        {
                id=000;
                age=00;
        }
        void getDat()
        {
                cout<<"\nEnter id and age: ";
                cin>>id>>age;
        }
        Employee(Employee &A)                 //Copy Constructor
        {
                id=A.id;
                age=A.age;
        }
        void dispDat()
        {
                cout<<"\nID: "<<id<<"\tAge: "<<age;
        }
};
void main()
{
        Employee E;          //creating an object of Employee
        E.dispDat();        //will display values initialized by default const.
        E.getDat();
        Employee F(E);      //creating an object and passing it to copy constructor
        F.dispDat();        //will display values of copy const.
}
```

Output:

```
ID: 0 Age: 0
Enter id and age: 121
22
ID: 121     Age: 22
```

Dynamic Constructor or Dynamic Array

We can allocate memory during the creation of objects too. The memory is saved as it allocates the right amount of memory for each object.

Allocation of memory to object at the time of their construction is known as **Dynamic Constructor** of objects. In the dynamic constructor, ***new*** operator is used for allocation of memory.

For Example:

```
#include <iostream.h>
class Dynam
{
        int *p,s;
        public:
        Dynam()                         //Default Constructor
        {
                p=NULL;
                s=0;
        }
        Dynam(int t)                    //Dynamic Constructor
        {
                s=t;
                p=new int[s];
        }
        void getDat()
        {
                int i;
                cout<<"\nEnter "<<s<<" values: ";
                for(i=0;i<s;i++)
                {
                        cin>>p[i];
                }
        }
        void dispDat()
        {
                cout<<"\nValues in Array: \t";
                for(int i=0;i<s;i++)
                {
                        cout<<p[i]<<"\t";
                }
        }
};
```

```
void main()
{
	int s;
	cout<<"\nEnter size of array: ";
	cin>>s;
	Dynam D(s);
	D.getDat();
	D.dispDat();
	cout<<"\nEnter size of array: ";
	cin>>s;
	Dynam E(s);
	E.getDat();
	E.dispDat();
}
```

Output:

```
Enter the size of array: 5
Enter 5 values: 1
4
3
2
66
Values in Array:  1     4     3     2     66
Enter the size of array: 2

Enter 2 values: 13
54
Values in Array:  13    54
```

Inheritance

Inheritance is one of the most important concepts in object-oriented programming. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to **reuse the code functionality** and fast implementation time.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base class**, and the new class is referred to as the **derived class**.

Modes of Inheritance

1) Public Mode

If we derive a sub class from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in derived class.

2) Protected mode

If we derive a sub class from a Protected base class. Then both public member and protected members of the base class will become protected in derived class.

3) Private mode

If we derive a sub class from a Private base class. Then both public member and protected members of the base class will become Private in derived class.

NOTE: *The private members in the base class cannot be directly accessed in the derived class, while protected members can be directly accessed.*

For Example: The private members in the base class cannot be directly accessed in the derived class, while protected members can be directly accessed.

```
#include <iostream.h>
class A
{
      public:
      int x;
      protected:
      int y;
      private:
      int z;
};
class B : public A
{
      // x is public
      // y is protected
      // z is not accessible from B
};
class C : protected A
{
      // x is protected
      // y is protected
      // z is not accessible from C
};
```

```
class D : private A    // 'private' is default for classes
{
      // x is private
      // y is private
      // z is not accessible from D
};
```

Access Control and Inheritance

ACCESS	PUBLIC	PROTECTED	PRIVATE
SAME CLASS	YES	YES	YES
DERIVED CLASSES	YES	YES	NO
OUTSIDE CLASSES	YES	NO	NO

Types of inheritance

In C++ there are following types of inheritance -

1) Single Level Inheritance
2) Multi-Level Inheritance
3) Multiple Inheritance
4) Hierarchical Inheritance
5) Hybrid Inheritance

Single Inheritance

When one class inherits another class, it is known as single level inheritance. Let's see the example of single level inheritance which inherits the fields only.

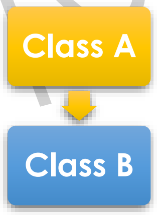


Figure: Single Level Inheritance

For Example:

```
#include <iostream.h>
class Account
{
      public:
      float salary = 60000;
};
class Programmer: public Account
{
      public:
      float bonus = 5000;
};
void main()
{
      Programmer p1;
      cout<<"Salary: "<<p1.salary<<endl;
      cout<<"Bonus: "<<p1.bonus<<endl;
}
```

Output:

```
Salary: 6000
Bonus: 5000
```

Multi-Level Inheritance

When one class inherits another class, which is further inherited by another class, it is known as **multi-level inheritance** in C++. In this type of inheritance last derived class acquires all the members of all its base classes.

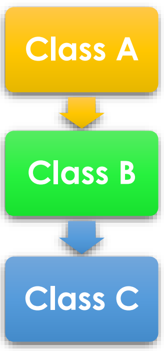


Figure: Multi-Level Inheritance

For Example:

```
#include <iostream.h>
class Animal
{
	public:
	void eat()
	{
		cout<<"Eating..."<<endl;
	}
};
class Dog: public Animal
{
	public:
	void bark()
	{
		cout<<"Barking..."<<endl;
	}
};
class BabyDog: public Dog
{
	public:
	void weep()
	{
		cout<<"Weeping...";
	}
};
void main()
{
	BabyDog d1;
	d1.eat();
	d1.bark();
	d1.weep();
}
```

Output:

```
Eating…
Barking…
Weeping…
```

Multiple Inheritance

Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes.

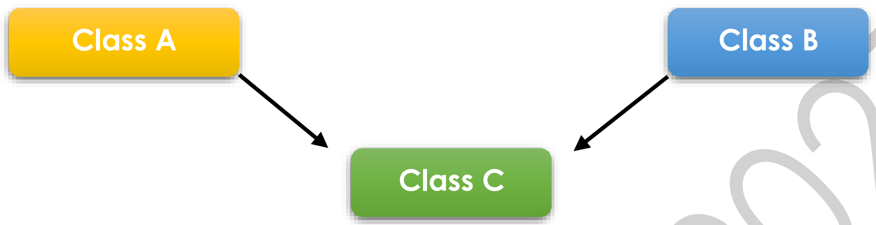


Figure: Multiple Inheritance

For Example:

```
#include <iostream.h>
class Truck
{
      public:
      void heavyMotor()
      {
            cout<<"Truck is a heavy motor vehicle"<<endl;
      }
};
class Car
{
      public:
      void lightMotor()
      {
            cout<<"Car is a light motor vehicle"<<endl;
      }
};
class Vehicle: public Truck, public Car
{
      public:
      void fourWheel()
      {
            cout<<"Both Truck & Car are 4 wheelers."<<endl;
      }
};
void main()
{
      Vehicle Ob;
      Ob.heavyMotor();
      Ob.lightMotor();
      Ob.fourwheel();
}
```

Output:

```
Truck is a heavy motor vehicle
Car is a light motor vehicle
Both Truck & Car are 4 wheelers
```

Hierarchical Inheritance

In this type of inheritance, more than one sub class is inherited from a single base class. i.e. more than one derived class is created from a single base class.

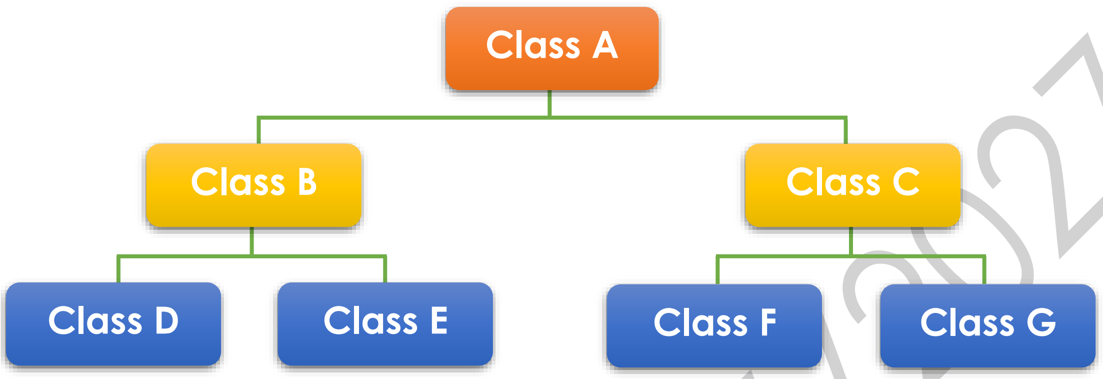


Figure: Hierarchical Inheritance

```
For Example:
#include <iostream.h>
class Vehicle
{
      public:
      void allVehicle()
      {
            cout<<"2 wheelers & 4 wheelers-> ";
      }
};
class TwoWheel: public Vehicle
{
      public:
      void Bike()
      {
            cout<<"Bike is 2 wheeler"<<endl;
      }
};
class FourWheel: public Vehicle
{
      public:
      void Car()
      {
            cout<<"Car is 4 wheeler"<<endl;
      }
};
void main()
{
      TwoWheel TW;
      FourWheel FW;
      TW.allVehicle();
      TW.Bike();
      FW.allVehicle();
      FW.Car();
}
```

Output:

```
2 wheelers & 4 wheelers -> Bike is a 2 wheeler
2 wheelers & 4 wheelers ->Car is 4 wheeler
```

Polymorphism

The term **"Polymorphism"** is the combination of **"poly"** + **"morphs"** which means many forms. It is a **Greek** word. In object-oriented programming, we use 3 main concepts: ***inheritance, encapsulation***, and ***polymorphism***.

Real Life Example of Polymorphism: Let's consider a real-life example of polymorphism. A lady behaves like a teacher in a classroom, mother or daughter in a home and customer in a market. Here, a single person is behaving differently according to the situations. There are two types of polymorphism in C++:

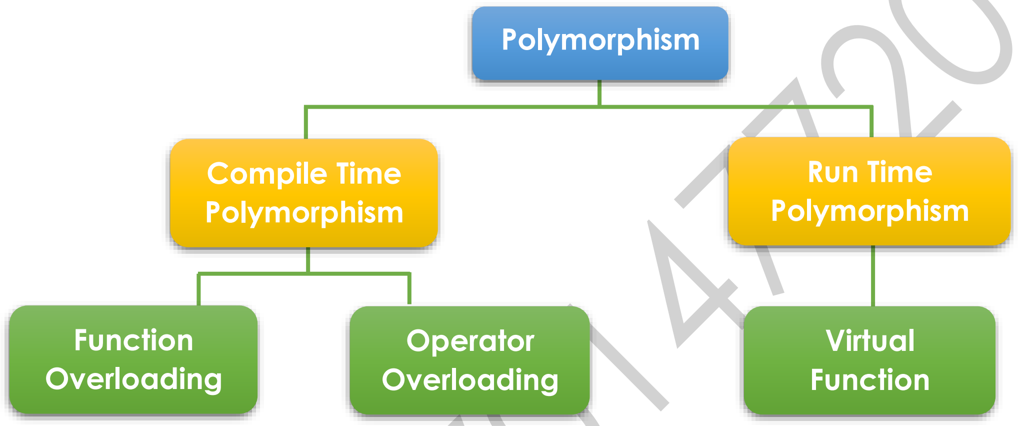


Figure: Types of Polymorphism

1) Compile time polymorphism

The overloaded functions are invoked by matching the type and number of arguments. This information is available at the compile time and, therefore, compiler selects the appropriate function at the compile time. **It is achieved by** function overloading **and** operator overloading which is also known as static binding or early binding. Now, let's consider the case where function name and prototype are same.

- ***For Detailed study about Function Overloading click here.***
- ***For Detailed study about Operator Overloading click here.***

```
class A                                //base class declaration.
{
      int a;
      public:
      void display()
      {
            cout<<"Class A ";
      }
};
class B : public A                     //derived class declaration.
{
      int b;
      public:
      void display()
      {
            cout<<"Class B";
```

```
        }
};
```

In the above case, the prototype of **display()** function is the same in both the **base and derived class**. Therefore, the static binding cannot be applied. It would be great if the appropriate function is selected at the run time. This is known as **run time polymorphism**.

2) Run time polymorphism

Run time polymorphism is achieved when the object's method is invoked at the run time instead of compile time. It is achieved by **method overriding** which is also known as **dynamic binding or late binding**.

- ***For Detailed study about Method Overriding click here.***
- ***For Detailed study about Dynamic Binding using Virtual function, click here.***

Differences b/w compile time and run time polymorphism.

Compile time polymorphism	Run time polymorphism
The function to be invoked is known at the compile time.	The function to be invoked is known at the run time.
It is also known as overloading, early binding and static binding.	It is also known as overriding, Dynamic binding and late binding.
Overloading is a compile time polymorphism where more than one method is having the same name but with the different number of parameters or the type of the parameters.	Overriding is a run time polymorphism where more than one method is having the same name, number of parameters and the type of the parameters.
It is achieved by function overloading and operator overloading.	It is achieved by virtual functions and pointers.
It provides fast execution as it is known at the compile time.	It provides slow execution as it is known at the run time.
It is less flexible as mainly all the things execute at the compile time.	It is more flexible as all the things execute at the run time.

Table: Compile time polymorphism vs Run time polymorphism

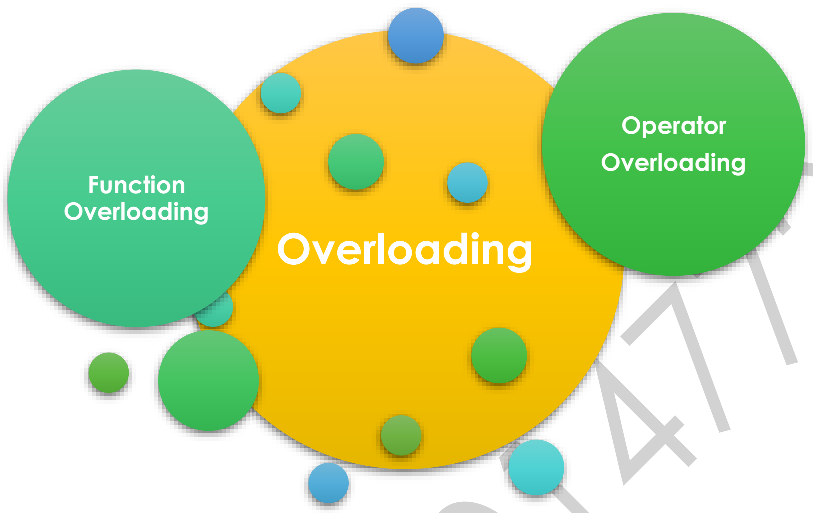
Overloading in C++

If we create two or more members having the same name but different in number or type of parameter, it is known as C++ overloading. In C++, we can overload:

- Methods
- Constructors

Types of Overloading

1) Function overloading
2) Operator overloading



Function Overloading

Another strong concept introduced in C++ is **Function Overloading.** We can overload our function in C++. As we know C++ support **Polymorphism**, so we can overload our functions in C++ program whenever needed. Function overloading is a concept in which we can define different function with same name but having different arguments/parameter list.

In function overloading, the function is redefined by using either different types of arguments or a different number of arguments. It is only through these differences' compiler can differentiate between the functions.

The advantage of Function overloading is that it increases the readability of the program because you don't need to use different names for the same action.

Highlights About Function Overloading

- In function overloading return type of function has no effect on function call of definition.
- Parameter list must be different to achieve function overloading.
- Function Name should be same with respect to above 2nd point.
- Function overloading is achieved in compile time (Early Binding).

Let's see the simple example of function overloading where we are changing number of arguments of add() method.

```
// program of function overloading when number of arguments vary.
#include <iostream.h>
class Cal
{
      public:
      int add(int a,int b)
      {
            return a + b;
      }
      int add(int a, int b, int c)
      {
            return a + b + c;
      }
};
void main()
{
      Cal C;                  // class object declaration.
      cout<<C.add(10, 20)<<endl;
      cout<<C.add(12, 20, 23);
}
```

Output:

```
30

55
```

```
// Program of function overloading with different types of arguments.
#include<iostream>
int mul(int,int);
float mul(float,int);
int mul(int a,int b)
{
      return a*b;
}
float mul(double x, int y)
{
      return x*y;
}
void main()
{
      int r1 = mul(6,7);
      float r2 = mul(0.2,3);
      cout << "r1 is : " <<r1<<endl;
      cout <<"r2 is : "  <<r2<<endl;
}
```

Output:

```
r1 is : 42
r2 is : 0.6
```

Operators Overloading

Operator overloading is a compile-time polymorphism in which the operator is overloaded to provide the special meaning to the user-defined data type. Operator overloading is used to overload or redefines most of the operators available in C++. It is used to perform the operation on the user-defined data type. For example, C++ provides the ability to add the variables of the user-defined data type that is applied to the built-in data types.

The advantage of Operators overloading is to perform different operations on the same operand.

Operator that cannot be overloaded are as follows:

- Scope operator (::)
- Sizeof
- member selector(.)
- ternary operator/Conditional Operator(?:)

Syntax of Operator Overloading

```
return_type class_name  : : operator op(argument_list)
{
     // body of the function.
}
```

Where the ***return type*** is the type of value returned by the function.

class_name is the name of the class.

operator op is an operator function where op is the operator being overloaded, and the operator is the keyword.

Rules for Operator Overloading

- Existing operators can only be overloaded, but the new operators cannot be overloaded.
- The overloaded operator contains at least one operand of the user-defined data type.
- We cannot use friend function to overload certain operators. However, the member function can be used to overload those operators.
- When unary operators are overloaded through a member function take no explicit arguments, but, if they are overloaded by a friend function, takes one argument.
- When binary operators are overloaded through a member function takes one explicit argument, and if they are overloaded through a friend function takes two explicit arguments.

Let's see the simple example of overloading **+ operator**: **Test operator + ()** operator function is defined (inside Test class).

```
// program to overload the addition operator (+)
#include <iostream.h>
class Test
{
      private:
      int num;
      public:
      void get()
      {
            cout<<"Enter a number: ";
            cin>>num;
      }
```

```
	void show()
	{
		cout<<"Sum is: "<<num;
	}
	Test operator +(Test T)
	{
		Test X;
		X.num=num+T.num;
		return X;
	}
};
void main()
{
	Test a,b,c;
	a.get();
	b.get();
	c=a+b;		// calling of a function "operator +"
	c.show();
}
```

Output:

```
Enter a number: 5
Enter a number: 4
Sum is: 9
```

Function Overriding

If derived class defines same function as defined in its base class, it is known as function overriding in C++. It is used to achieve runtime polymorphism. It enables you to provide specific implementation of the function which is already provided by its base class.

Let's see a simple example of Function overriding in C++. In this example, we are overriding the **eat()** function.

```
#include <iostream.h>
using namespace std;
class Animal
{
	public:
	void eat()
	{
		cout<<"Eating...";
	}
};
class Dog: public Animal
{
	public:
	void eat()
	{
		cout<<"Eating bread...";
	}
};
void main()
{
	Dog d = Dog();
	d.eat();
}
```

Output:

```
Eating bread...
```

Virtual Function

A virtual function is a member function in the base class that We redefine in a derived class. It is declared using the **virtual** keyword. It is used to tell the compiler to perform dynamic linkage or late binding on the function.

There is a necessity to use the single pointer to refer to all the objects of the different classes. So, we create the pointer to the base class that refers to all the derived objects. But, when base class pointer contains the address of the derived class object, always executes the base class function. This issue can only be resolved by using the **'virtual'** function. A **'virtual'** is a keyword preceding the normal declaration of a function. When the function is made virtual, C++ determines which function is to be invoked at the runtime based on the type of the object pointed by the base class pointer.

Rules of Virtual Function

- Virtual functions must be members of some class.
- Virtual functions cannot be static members.
- They are accessed through object pointers.
- They can be a friend of another class.
- A virtual function must be defined in the base class, even though it is not used.
- The prototypes of a virtual function of the base class and all the derived classes must be identical. If the two functions with the same name but different prototypes, C++ will consider them as the overloaded functions.
- We cannot have a virtual constructor, but we can have a virtual destructor
- Consider the situation when we don't use the virtual keyword.

For Example:

```
#include <iostream.h>
#include <conio.h>
class A
{
      int x=5;
      public:
      void display()
      {
            cout<<"Value of x is : "<<x<<endl;
      }
};
class B: public A
{
      int y = 10;
      public:
      void display()
      {
            cout << "Value of y is : " <<y<<endl;
      }
};
void main()
{
      A *a;
      B b;
```

```
        a = &b;
        a->display();
        getch();
}
```

Output:

```
Value of x is: 5
```

In the above example, * a is the base class pointer. The pointer can only access the base class members but not the members of the derived class. Although C++ permits the base pointer to point to any object derived from the base class, it cannot directly access the members of the derived class. Therefore, there is a need for virtual function which allows the base pointer to access the members of the derived class.

Let's see the example of virtual function used to invoked the derived class in a program.

```
#include <iostream.h>
class A
{
        public:
        virtual void display()
        {
                cout << "Base class is invoked"<<endl;
        }
};
class B:public A
{
        public:
        void display()
        {
        cout << "Derived Class is invoked"<<endl;
        }
};
void main()
{
        A* a;                   //pointer of base class
        B b;                    //object of derived class
        a = &b;
        a->display();           //Late Binding occurs
}
```

Output:

```
Derived Class is invoked
```

Pure Virtual Function

- A virtual function is not used for performing any task. It only serves as a placeholder.
- When the function has no definition, such function is known as "**do-nothing**" function.
- The "**do-nothing**" function is known as a **pure virtual function**. A pure virtual function is a function declared in the base class that has no definition relative to the base class.
- A class containing the pure virtual function cannot be used to declare the objects of its own, such classes are known as abstract base classes.
- The main objective of the base class is to provide the traits to the derived classes and to create the base pointer used for achieving the runtime polymorphism.

Let's see a simple example:

```
#include <iostream.h>
class Base
{
    public:
    virtual void show() = 0;
};
class Derived : public Base
{
    public:
    void show()
    {
        cout << "Derived class is derived from the base class." <<endl;
    }
};
void main()
{
    Base *bptr;
    //Base b;
    Derived d;
    bptr = &d;
    bptr->show();
}
```

Output:

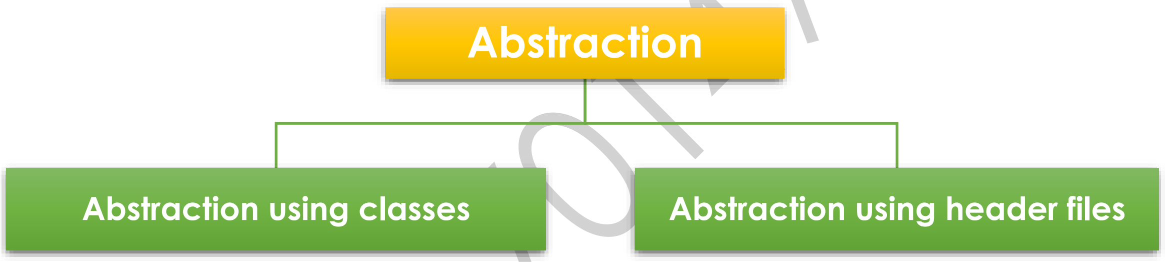
```
Derived class is derived from the base class.
```

In the above example, the base class contains the pure virtual function. Therefore, the base class is an abstract base class. We cannot create the object of the base class.

Data Abstraction

- Data Abstraction is a process of providing only the essential details to the outside world and hiding the internal details, ***for example:*** representing only the essential details in the program.
- **Data Abstraction** is a programming technique that depends on the separation of the interface and implementation details of the program.
- Let's take a real-life example of AC, which can be turned ON or OFF, change the temperature, change the mode, and other external components such as fan, swing. But we don't know the internal details of the AC, now the question is: how it works internally? Thus, we can say that AC separates the implementation details from the external interface.
- C++ provides a great level of abstraction.
- For example, pow() function is used to calculate the power of a number without knowing the algorithm the function follows.
- In C++ program if we implement class with private and public members then it is an example of data abstraction.

Data Abstraction can be achieved in two ways:
1) Abstraction using classes
2) Abstraction in header files.



Abstraction using classes

An abstraction can be achieved using classes. A class is used to group all the data members and member functions into a single unit by using the access specifiers. A class has the responsibility to determine which data member is to be visible outside and which is not.

Abstraction in header files

Another type of abstraction is header file. For example, pow() function available is used to calculate the power of a number without actually knowing which algorithm function uses to calculate the power. Thus, we can say that header files hide all the implementation details from the user.

Access Specifiers Implement Abstraction:

Public specifier

When the members are declared as public, members can be accessed anywhere from the program.

Private specifier

When the members are declared as private, members can only be accessed only by the member functions of the class.

Let's see a simple example of abstraction in header files.

```
// program to calculate the power of a number.
#include <iostream.h>
#include<math.h>
void main()
{
      int n = 4;
      int power = 3;
      int result = pow(n,power);          // pow(n,power) is the  power function
      cout << "Cube of n is : " <<result<<endl;
}
```

Output:

```
Cube of n is : 64
```

In the above example, pow() function is used to calculate 4 raised to the power 3. The pow() function is present in the **math.h** header file in which all the implementation details of the pow() function is hidden.

Let's see a simple example of data abstraction using classes.

```
#include <iostream.h>
class Sum
{
      private:
      int x, y, z; // private variables
      public:
      void add()
      {
            cout<<"Enter two numbers: ";
            cin>>x>>y;
            z= x+y;
            cout<<"Sum of two number is: "<<z<<endl;
      }
};
void main()
{
      Sum sm;
      sm.add();
}
```

Output:

```
Enter two numbers:
3
6
Sum of two number is: 9
```

In the above example, abstraction is achieved using classes. A class 'Sum' contains the private members x, y and z are only accessible by the member functions of the class.

Advantages of Abstraction:

- Implementation details of the class are protected from the inadvertent user level errors.
- A programmer does not need to write the low-level code.
- Data Abstraction avoids the code duplication, **for instance**: programmer does not have to go through the same tasks every time to perform the similar operation.

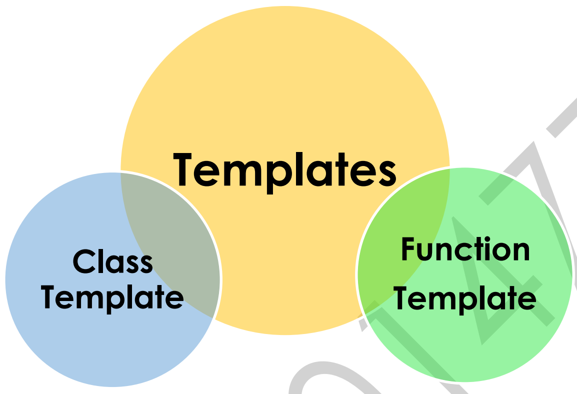
- The main aim of the data abstraction is to reuse the code and the proper partitioning of the code across the classes.
- Internal implementation can be changed without affecting the user level code.

Templates

A C++ template is a powerful feature added to C++. It allows us to define the generic classes and generic functions and thus provides support for generic programming. ***Generic programming is a technique where generic types are used as parameters in algorithms so that they can work for a variety of data types.***

Templates can be represented in two ways:

1) Function templates
2) Class templates



1) Function Templates

We can define a template for a function. For example, if we have an add() function, we can create versions of the add function for adding the int, float or double type values.

Points to Remember

- Generic functions use the concept of a function template. Generic functions define a set of operations that can be applied to the various types of data.
- The type of the data that the function will operate on depends on the type of the data passed as a parameter.
- **For example**: A Sorting algorithm can be implemented using a generic function, it can be implemented to an array of integers or array of floats.
- A Generic function is created by using the keyword **template**. The template defines what function will do.

Syntax of Function Template

```
template<class Ttype> return_type function_name(parameter_list)
{
    // body of function.
}
```

Where **Ttype**: It is a placeholder name for a data type used by the function. It is used within the function definition. It is only a placeholder that the compiler will automatically replace this placeholder with the actual data type.

class: A class keyword is used to specify a generic type in a template declaration.

Let's see a simple example of a function template:

```
#include <iostream.h>
template<class T> int add(T a,T b)
{
      return a+b;
}
void main()
{
      int i =2;
      int j =3;
      float m = 2.3;
      float n = 1.2;
      cout<<"\nAddition of i and j is: "<<add(i,j);
      cout<<"\nAddition of m and n is: "<<add(m,n);
}
Output:
```

```
Addition of i and j is :5
Addition of m and n is :3.5
```

In the above example, we create the function template which can perform the addition operation on any type either it can be integer, float or double.

Function Templates with Multiple Parameters

We can use more than one generic type in the template function by using the comma to separate the list.

Syntax

```
template<class T1, class T2,.....>
return_type function_name (arguments of type T1, T2....)
{
      // body of function.
}
```

In the above syntax, we have seen that the template function can accept any number of arguments of a different type.

Let's see a simple example:

```
#include <iostream.h>
template<class X,class Y> void fun(X a,Y b)
{
      cout<<"Value of a is: "<<a<<endl;
      cout<<"Value of b is: "<<b<<endl;
}
void main()
{
      fun(15,12.3);
}
Output:
```

```
Value of a is: 15
Value of b is: 12.3
```

In the above example, we use two generic types in the template function, i.e., X and Y.

Overloading a Function Template

We can overload the generic function means that the overloaded template functions can differ in the parameter list.

Let's understand this through a simple example:

```
#include <iostream.h>
template<class X> void fun(X a)
{
	cout <<"Value of a is: "<<a<<endl;
}
template<class X,class Y> void fun(X b ,Y c)
{
	cout<<"Value of b is: "<<b<<endl;
	cout<<"Value of c is: "<<c<<endl;
}
void main()
{
	fun(10);
	fun(20,30.5);
}
```

Output:

```
Value of a is : 10
Value of b is : 20
Value of c is : 30.5
```

In the above example, template of fun() function is overloaded.

2) Class Template

We can define a template for a class. For example, a class template can be created for the array class that can accept the array of various types such as int array, float array or double array. **Class Template** can also be defined similarly to the Function Template. When a class uses the concept of Template, then the class is known as generic class.

Syntax

```
template<class Ttype>
class class_name
{
	.
	.
};
```

Ttype is a placeholder name which will be determined when the class is instantiated. We can define more than one generic data type using a comma-separated list. The **Ttype** can be used inside the class body.

Now, we create an instance of a class

```
class_name<type> ob;
```

where:

class_name: It is the name of the class.

type: It is the type of the data that the class is operating on.

ob: It is the name of the object.

Let's see a simple example:

```
#include <iostream.h>
template<class T>
class A
{
	public:
	T a = 5;
	T b = 6;
	void add()
	{
		cout<<"Addition of a+b: "<<a+b<<endl
	}

};
void main()
{
	A<int> d;
	d.add();
}
```

Output:

```
Addition of a+b: 11
```

In the above example, we create a template for class A. Inside the main() method, we create the instance of class A named as, 'd'.

Class Template with Multiple Parameters

We can use more than one generic data type in a class template, and each generic data type is separated by the comma.

Syntax

```
template<class T1, class T2, ......>
class class_name
{
	// Body of the class.
}
```

Let's see a simple example when class template contains two generic data types.

```
#include <iostream.h>
template<class T1, class T2>
class A
{
	T1 a;
	T2 b;
	public:
	A(T1 x,T2 y)
	{
		a = x;
		b = y;
	}
	void display()
	{
		cout<<"Sum of a+b is: "<<a+b<<endl;
	}
};
```

```
void main()
{
      A<int,float> d(5,6.5);
      d.display();
}
Output:
```

```
Values of a and b are : 5,6.5
```

Nontype Template Arguments

The template can contain multiple arguments, and we can also use the non-type arguments In addition to the type T argument, we can also use other types of arguments such as strings, function names, constant expression and built-in types. Let' s see the following example:

```
template<class T, int size>
class array
{
      T arr[size];      // automatic array initialization.
};
```

In the above case, the ***nontype template*** argument is **size** and therefore, template supplies the size of the array as an argument. Arguments are specified when the objects of a class are created:

```
            array<int, 15> t1;            //array of 15 integers.
            array<float, 10> t2;          // array of 10 floats.
            array<char, 4> t3;            // array of 4 chars.
```

For Example:

```
#include<iostream.h>
template<class T, int size>
class A
{
      public:
      T arr[size];
      void insert()
      {
            int i =1;
            for (int j=0;j<size;j++)
            {
                  arr[j] = i;
                  i++;
            }
      }
      void display()
      {
            for(int i=0;i<size;i++)
            {
                  cout<<arr[i]<<"\t";
            }
      }
};
```

```
void main()
{
      A<int,10> t1;
      t1.insert();
      t1.display();
}
Output:
```

```
1     2     3     4     5     6     7     8     9     10
```

In the above example, the class template is created which contains the **nontype** template argument, i.e., size. It is specified when the object of class 'A' is created.

Points to Remember

- C++ supports a powerful feature known as a template to implement the concept of generic programming.
- A template allows us to create a family of classes or family of functions to handle different data types.
- Template classes and functions eliminate the code duplication of different data types and thus makes the development easier and faster.
- Multiple parameters can be used in both class and function template.
- Template functions can also be overloaded.
- We can also use **nontype arguments** such as built-in or derived data types as template arguments.

Nested Classes

A nested class is a class that is declared in another class. The nested class is also a member variable of the enclosing class and has the same access rights as the other members. However, the member functions of the enclosing class have no special access to the members of a nested class.

A program that demonstrates nested classes in C++ is as follows.

```
#include<iostream.h>
class Outer
{
      int a,b;
      public:
      void set()
      {
            a=12;
            b=32;
      }
      void show()
      {
            cout<<"Value of A: "<<a<<"\nValue of B: "<<b;
      }
      class Inner
      {
            int x,y;
            public:
            void put()
            {
                  x=76;
                  y=45;
            }
            void disp()
            {
                  cout<<"Value of X: "<<x<<"\nValue of Y: "<<y;
            }
      };

};
void main()
{
      Outer A;
      A.get();
      A.show();
      Inner B;
      B.put();
      B.disp();
}
```

Output:

```
Value of A: 12
Value of B: 32
Value of X: 76
Value of Y: 45
```

Please do not use DOSBOX/TURBOC compilers for Exception Handling in C++.

Use other C++ IDE like Visual Studio Code/ Code Block etc.

IMPORTANT

Exception Handling

Exception Handling is a process to handle runtime errors. We perform exception handling so the normal flow of the application can be maintained even after runtime errors. In C++, exception is an event or object which is thrown at runtime. All exceptions are derived from exception class. It is a runtime error which can be handled. If we don't handle the exception, it prints exception message and terminates the program.

Errors interrupt normal execution of program. Exception handling is the process of handling errors and exceptions in such a way that they do not hinder normal execution of the system. For example, User divides a number by zero, this will compile successfully but an exception or run time error will occur due to which our applications will be crashed. In order to avoid this, we'll introduce exception handling technics in our code.

Advantage of Exception Handling

It maintains the normal flow of the application. In such case, rest of the code is executed even after exception.

Types of Errors

Let's discuss about types of error first, Errors can be broadly categorized into two types:

1. Compile Time Errors
2. Run Time Errors

1) Compile Time Errors

Errors caught during compiled time is called Compile time errors. Compile time errors include library reference, syntax error or incorrect class import.

2) Run Time Errors

They are also known as exceptions. An exception caught during run time creates serious issues.

In C++, Error handling is done using three keywords:

- try
- catch
- throw

Syntax

```
try
{
    //code
    throw parameter;
}
catch(exceptionname ex)
{
    //code to handle exception
}
```

try block

The code which can throw any exception is kept inside (or enclosed in) a `try` block. Then, when the code will lead to any error, that error/exception will get caught inside the `catch` block.

catch block

`catch` block is intended to catch the error and handle the exception condition. We can have multiple catch blocks to handle different types of exception and perform different actions when the exceptions occur. For example, we can display descriptive messages to explain why any particular exception occurred.

throw statement

It is used to throw exceptions to exception handler i.e. it is used to communicate information about error. A `throw` expression accepts one parameter and that parameter is passed to handler. `throw` statement is used when we explicitly want an exception to occur, then we can use `throw` statement to throw or generate that exception.

Need of Exception Handling

Let's take a simple example to understand the usage of try, catch and throw. Below program compiles successfully but the program fails at runtime, leading to an exception.

```
#include<iostream>
using namespace std;
float division(int x, int y)
{
      return (x/y);
}
int main ()
{
      int i=50;
      int j=0;
```

```
	float k=0;
	k = division(i, j);
	cout<<k<<endl;
	return 0;
}
```

The above program will not run, and will show **runtime error** on screen, because we are trying to divide a number with **0**, which is not possible.

How to handle this situation? We can handle such situations using exception handling and can inform the user that you cannot divide a number by zero, by displaying a message.

Now we will update the above program and include exception handling in it.

```
#include <iostream>
using namespace std;
float division(int x, int y)
{
	try
	{
		if(y==0)
		{
			throw "Exception: Attempted to divide by zero!";
		}
		else
		{
			return (x/y);
		}
	}
	catch (const char* e)
	{
		cout<< e << endl;
	}
}
int main ()
{
	int i = 25;
	int j = 0;
	float k = 0;
	k = division(i, j);
	cout<< k << endl;
	return 0;
}
```

Output:

```
Exception: Attempted to divide by zero!
```

Multiple catch Blocks

Below program contains multiple catch blocks to handle different types of exception in different way.

```
#include <iostream>
using namespace std;
float division(int x, int y)
{
	if(y==0)
	{
		throw "Exception: Attempted to divide by zero!";
	}
	else if(y<-32768 || y>32767 || x<-32768 || x>32767)
	{
		throw "Exception: Number is out of integer value range!";
	}
	else
	{
		return (x/y);
	}
}
int main ()
{
	int i ;
	int j ;
	float k;
	cout<<"Enter two numbers: ";
	cin>>i>>j;
	try
	{
		k = division(i, j);
		cout << k << endl;
	}
	catch (const char* e)
	{
		cout << e << endl;
	}
	catch (const char* e)
	{
		cout<< e << endl;
	}
	return 0;
}
```

Output:

```
Enter two numbers: 21
43123
Exception: Number is out of integer value range!
```

Standard Exceptions in C++

There are some standard exceptions in C++ under <exception> which we can use in our programs. They are arranged in a parent-child class hierarchy which is depicted below:

std::exception - Parent class of all the standard C++ exceptions.

logic_error - Exception happens in the internal logical of a program.

domain_error - Exception due to use of invalid domain.

invalid argument - Exception due to invalid argument.

out_of_range - Exception due to out of range i.e. size requirement exceeds allocation.

length_error - Exception due to length error.

runtime_error - Exception happens during runtime.

range_error - Exception due to range errors in internal computations.

overflow_error - Exception due to arithmetic overflow errors.

underflow_error - Exception due to arithmetic underflow errors

bad_alloc - Exception happens when memory allocation with **new()** fails.

bad_cast - Exception happens when dynamic cast fails.

bad_exception - Exception is specially designed to be listed in the dynamic-exception-specifier.

bad_typeid - Exception thrown by typeid.

User-Defined Exceptions

The new exception can be defined by overriding and inheriting **exception** class functionality. Let's see the simple example of user-defined exception in which **std::exception** class is used to define the exception.

```
#include <iostream>
#include <exception>
using namespace std;
class MyException : public exception
{
	public:
	const char * what() const throw()
	{
		return "Exception: Attempted to divide by zero!";
	}
};
int main()
{
	try
	{
		int x, y;
		cout << "Enter the two numbers: ";
		cin >> x >> y;
		if (y == 0)
		{
			MyException z;
			throw z;
		}
```

```
            else
            {
                  cout << "x / y = " << x/y << endl;
            }
      }
      catch(exception e)
      {
            cout << e.what();
      }
}
```

Output:

```
Enter two numbers: 21
0
Exception: Attempted to divide by zero!
```

File Handling

File represents storage medium for storing data or information. Streams refer to sequence of bytes. In Files we store data i.e. text or binary data permanently and use these data to read or write in the form of input output operations by transferring bytes of data. So, we use the term File Streams/File handling. We use the header file **<fstream>.**

ofstream: It represents output Stream and this is used for writing in files.

ifstream: It represents input Stream and this is used for reading from files.

fstream: It represents both output Stream and input Stream. So, it can read from files and write to files.

Operations in File Handling

1) Creating a file: **`open()`**
2) Reading data: **`read()`**
3) Writing new data: **`write()`**
4) Closing a file: **`close()`**

Creating/Opening a File

We create/open a file by specifying new path of the file and mode of operation. Operations can be reading, writing, appending and truncating.

Syntax for file creation:

FilePointer.open("Path",ios::mode);

Example of file opened for writing: **`st.open("E:\demo.txt",ios::out);`**

Example of file opened for reading: **`st.open("E:\demo.txt ",ios::in);`**

Example of file opened for appending: **`st.open("E:\demo.txt ",ios::app);`**

Example of file opened for truncating: **`st.open("E:\demo.txt ",ios::trunc);`**

```
#include<iostream>
#include <fstream>
using namespace std;
int main()
{
	fstream st;                     // Step 1: Creating object of fstream class
	st.open("E:\demo.txt",ios::out);    // Step 2: Creating new file
	cout<<"File creation successful.";
	st.close();                     // Step 2: Creating new file
	return 0;
}
```

Output:

```
File creation successful.
```

Writing a File

Example: writing a file using *fstream*

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
        fstream st;                            // Step 1: Creating object of fstream class
        st.open("E:\demo.txt",ios::out);       // Step 2: Creating new file
        if(!st)                                // Step 3: Checking whether file/path exist
        {
                cout<<"File creation failed";
        }
        else
        {
                cout<<"New file created..";
                st<<"Hello";                   // Step 4: Writing to file
                cout<<"\nData Saved!!";
                st.close();                    // Step 5: Closing file
        }
        return 0;
}
```

Output:

```
New file created..
Data Saved!!
```

Example: writing a file using ofstream

```
#include <iostream>
#include <fstream>
using namespace std;
int main ()
{
        ofstream fs("demo.txt");
        if (fs.is_open())
        {
                fs<<"Welcome to CPP.\n";
                fs<<"Learning CPP.\n";
                fs.close();
                cout<<"Data is saved!!!";
        }
        else
        {
                cout<<"File opening is failed.";
        }
        return 0;
}
```

Output:

```
Data is saved!!!
```

Reading a File

Example: reading a file using fstream

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
        fstream st;                              // step 1: Creating object of fstream class
        st.open("E:\demo.txt",ios::in);          // Step 2: Creating new file
        if(!st)                                  // Step 3: Checking whether file/path exist
        {
                cout<<"No such file";
        }
        else
        {
                char ch;
                while (!st.eof())
                {
                        st>>ch;                  // Step 4: Reading from file
                        cout<<ch;                // Data from file
                }
                st.close();                      // Step 5: Closing file
        }
        return 0;
}
```

Output:

```
Helloo
```

Example: reading a file using *ifstream*

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
        string srg;
        ifstream fs("demo.txt");
        if (fs.is_open())
        {
                while(getline(fs,srg))
                {
                        cout<<srg<<endl;
                }
                fs.close();
        }
        else
        {
                cout<<"File opening is failed."<<endl;
        }
        return 0;
}
```

Output:

```
Welcome to CPP.
Learning CPP.
```

Append data to file

For Example:

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
	fstream st;
	st.open("Demo.txt",ios::app);
	if(!st)
	{
		cout<<"No such file";
	}
	else
	{
		char n[200];
		int a;
		cout<<"Enter Name: ";
		cin.get(n,200);
		cout<<"Enter Age: ";
		cin<<a;
		st<<endl<<n<<"\t\t"<<a;
		st.close();
		st.open("Demo.txt",ios::in);
		string ch;
		while (getline(st,ch))
		{
			cout<<ch<<endl;
		}
	}
	return 0;
}
```

Output

```
Enter Name: Vikas
Enter Age: 22

Naresh       25
Ravi         22
Vikas        22
```

iostream Member Function

This class inherits all members from its two parent classes istream and ostream, thus it is able to perform both input and output operations. The class relies on a single streambuf object for both the input and output operations.

There are following function can be used for modified output using ***cout.***

1) cout.write()
2) cout.put()
3) cout.width()
4) cout.setf()
5) cout.precision()
6) cout.fill()

1) cout.write()

It is used to write block of data.

Syntax:

```
            cout.write(variable,size);
```

Example:

```
#include<iostream>
using namespace std;
int main()
{
      char a[200]="Hello World!! Testing iostream!!";
      cout.write(a,200);
      return 0;
}
```

Output:

```
Hello World!! Testing iostream!!
```

2) cout.put()

It is used to write a character.

Syntax:

```
      cout.put(variable);
```

Example:

```
      #include<iostream>
      using namespace std;
      int main()
      {
            char a='m';
            cout.put(a);
            return 0;
      }
```

Output:

```
m
```

3) cout.write()

This method determines the minimum number of characters to be written in some output representations.

Syntax:

```
        cout.write(size);
```

Example:

```
#include<iostream>
using namespace std;
int main()
{
        char a[50]="Hello World!! Testing iostream!!";
        cout.width(50);
        cout.write(a,30);
        cout<<"Text after width ends";
        return 0;
}
```

Output:

```
Hello World!! Testing iostream                         Text after width ends
```

4) cout.setf()

It is used to modify the default flag.

Syntax:

```
        cout.setf(ios::right);                    : Set the alignment to right
        cout.setf(ios::left);                     : Set the alignment to left
```

Example:

```
#include<iostream>
using namespace std;
int main()
{
        char a[200]="Hello World!! Testing iostream!!";
        cout.width(200);
        cout.setf(ios::right);
        cout<<a<<endl;
        cout.setf(ios::left);
        cout<<a<<endl;
        return 0;
}
```

Output:

```
                      Hello World!! Testing iostream!!
Hello World!! Testing iostream!!
```

5) cout.precision()

This function is used to control the number of digits of an output stream display of a floating-point value.

Syntax:

```
cout.precision(int n);
```

n: New value for the decimal precision.

Example:

```
#include<iostream>
using namespace std;
int main()
{
	float n= 11.32453244;
	cout.precision(3);
	cout<<n<<endl;
	cout<<fixed;
	cout.precision(3);
	cout<<n;
	return 0;
}
Output:
```

```
11.3
11.324
```

6) cout.fill()

This function used to fill blank width with given character constant.

Syntax:

```
cout.fill(<character_const_to_fill>);
```

Example:

```
#include<iostream>
using namespace std;
int main()
{
	char a[200]="Hello World!! Testing iostream!!";
	cout.width(50);
	cout.fill('*');
	cout<<a<<endl;
	return 0;
}
Output:
```

```
Hello World!! Testing iostream!!******************
```