B.C.A study

Unit-1: introduction to C++

Introduction

“C++ is a statically-typed, free-form, (usually) compiled, multi-paradigm, intermediate-level general-purpose middle-level programming language.”

In simple terms, C++ is a sophisticated, efficient and a general-purpose programming language based on C. It was developed by BjarneStroustrup in 1979.

Many of today’s operating systems, system drivers, browsers and games use C++ as their core language. This makes C++ one of the most popular languages today.

Since it is an enhanced/extended version of C programming language, C and C++ are often denoted together as C/C++.

HISTORY

While BjarneStroustrup was working in AT&T Bell Labs in 1979, he faced difficulties in analyzing UNIX kernel for distributed systems. The current languages were either too slow or too low level. So, he set forward to create a new language.

For building this language, he chose C. Why C? Because it is a general purpose language and is very efficient as well as fast in its operations.

He used his knowledge of object-oriented model from SIMULA and began working on class extensions to C. His aim was to create a language with far higher level of abstraction while retaining the efficiency of C.

This new programming language was named C withClasses, but was later renamed to C++ (++ refers to the increment operator in C).

UPGRADES

C++98

When C++ was first released in 1985, there were no official standards released. It was only until 1998 that C++ was first standardized which was known as C++98.

C++03

In 2003, a new version of C++ standard was published. C++03 wasn't really a new standard altogether but a bug fix release identified with C++98 "to ensure greater consistency and portability".

C++11 (C++0x)

The next major standard for C++ was released in 2011 and it was named C++11. Since, C++ committee was sure this update would be released within 2009, they unofficially named it C++0x. Later, when they didn't, Stroustrup joked that C++0x went hexadecimal – C++0xB (C++11). Nice save.

C++14 (C++1y)

C++14 is the current iteration of C++ released in 2014. Like C++03, it included mainly bug fixes and simple improvements to C++11.

C++17 (C++1z)

The supposedly next iteration to C++ which is planned to be rolled out in 2017. It is expected to have many new features. Most of the features planned for this version are alreadted.

FEATURES OF C++

Being a general-purpose language, C++ is undoubtedly feature-rich. Going through all the features will take you some time but, as a beginner, below are the most important features you should know.

1.C++ is fast

Since, C++ is an extended version of C, the C part of it is very low level.

This offers a huge boost in speed that high level languages like Python, Java don't give you.

2.C++ is statically typed

C++ is a statically typed programming language.

In simple terms, C++ doesn't allow the compiler to make assumptions about the type of data e.g. 10 is different from "10" and you have to let C++ know which one you are talking about.

This helps the compiler catch errors and bugs before execution of the program.

3 .C++ is a multi-paradigm programming language

C++ supports at least 7 different styles of programming and gives developers the freedom to choose one at their will.

Unlike Java and Python, you don't need to use objects to solve every task (if it's not necessary).

You can choose the programming style that fits your use case.

4.Object oriented programming with C++

Object oriented programming helps you solve a complex problem intuitively.

With its use in C++, you are able to divide these complex problems into smaller sets by creating objects.

5.Power of standard library (Standard template library – STL)

The power of C++ extends with the use of standard libraries contained in it.

These libraries contain efficient algorithms that you use extensively while coding.This saves ample amount of programming effort, which otherwise would have been wasted reinventing the wheel.

INTRODUCTION OF OBJECT ORIENTED APPROACH

The prime purpose of C++ programming was to add object orientation to the C programming language, which is in itself one of the most powerful programming languages.

The core of the pure object-oriented programming is to create an object, in code, that has certain properties and methods. While designing C++ modules, we try to see whole world in the form of objects. For example a car is an object which has certain properties such as color, number of doors, and the like. It also has certain methods such as accelerate, brake, and so on.

There are a few principle concepts that form the foundation of object-oriented programming −

Object

This is the basic unit of object oriented programming. That is both data and function that operate on data are bundled as a unit called as object.

Class

When you define a class, you define a blueprint for an object. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

Abstraction

Data abstraction refers to, providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details.

For example, a database system hides certain details of how data is stored and created and maintained. Similar way, C++ classes provides different methods to the outside world without giving internal detail about those methods and data.

Encapsulation

Encapsulation is placing the data and the functions that work on that data in the same place. While working with procedural languages, it is not always clear which functions work on which variables but object-oriented programming provides you framework to place the data and the relevant functions together in the same object.

Inheritance

One of the most useful aspects of object-oriented programming is code reusability. As the name suggests Inheritance is the process of forming a new class from an existing class that is from the existing class called as base class, new class is formed called as derived class.

This is a very important concept of object-oriented programming since this feature helps to reduce the code size.

Polymorphism

The ability to use an operator or function in different ways in other words giving different meaning or functions to the operators or functions is called polymorphism. Poly refers to many. That is a single function or an operator functioning in many ways different upon the usage is called polymorphism.

Overloading

The concept of overloading is also a branch of polymorphism. When the exiting operator or function is made to operate on new data type, it is said to be overloaded.

Review of C

The C language exhibits the following characteristics:

1. There is a small, fixed number of keywords, including a full set of control flow primitives: if/else, for, do/while, while, and switch. User-defined names are not distinguished from keywords by any kind of sigil.
2. There are a large number of arithmetic, bitwise and logic operators: +, +=, ++, &, ||, etc.
3. More than one assignment may be performed in a single statement.
4. Function return values can be ignored when not needed.
5. Typing is static, but weakly enforced; all data has a type, but implicit conversions are possible.
6. Declaration syntax mimics usage context. C has no “define” keyword; instead, a statement beginning with the name of a type is taken as a declaration. There is no “function” keyword; instead, a function is indicated by the parentheses of an argument list.
7. User-defined (typedef) and compound types are possible.
8. Heterogeneous aggregate data types (struct) allow related data elements to be accessed and assigned as a unit.
9. Union is a structure with overlapping members; only the last member stored is valid.
10. Array indexing is a secondary notation, defined in terms of pointer arithmetic. Unlike structs, arrays are not first-class objects: they cannot be assigned or compared using single built-in operators. There is no “array” keyword in use or definition; instead, square brackets indicate arrays syntactically, for example month[11].
11. Enumerated types are possible with the enum keyword. They are freely interconvertible with integers.
12. Strings are not a distinct data type, but are conventionally implemented as null-terminated character arrays.

Relations to other languages

Many later languages have borrowed directly or indirectly from C, including C++, C#, Unix’s C shell, D, Go, Java, JavaScript, Limbo, LPC, Objective-C, Perl, PHP, Python, Rust, Swift, Verilog and SystemVerilog (hardware description languages).[5] These languages have drawn many of their control structures and other basic features from C. Most of them (Python being a dramatic exception) also express highly similar syntax to C, and they tend to combine the recognizable expression and statement syntax of C with underlying type systems, data models, and semantics that can be radically different.

DIFFERENCE BETWEEN C AND C++

C Programming Language

1. C language was developed in 1972 by Dennis Ritchie.

2. C is a Procedural Oriented Programming language.

3. C has Top Down programming approach.

4. In c programming language, a big problem divides into small pieces known as functions; this approach of programming is known as Modular programming.

5. Structure in C does not have function declaration features i.e. we cannot declare a function as member function of structure in C.

C++ Programming Language

1. C++ language was developed in 1980 by BjarneStrostroup.

2. C++ is an Object Oriented Programming language.

3. C++ has Bottom Up programming approach.

4. In c++ programming language, a big problem divides into Classes and Objects.

5. Structure in C++ provide the feature of declare a function as member function of structure.

Cin and Cout

The Standard Output Stream (cout)

The predefined object **cout** is an instance of **ostream** class. The cout object is said to be "connected to" the standard output device, which usually is the display screen. The **cout** is used in conjunction with the stream insertion operator, which is written as << which are two less than signs as shown in the following example.Live Demo

```
#include <iostream>

using namespace std;

int main() {
   char str[] = "Hello C++";

   cout << "Value of str is : " << str << endl;
}
```

When the above code is compiled and executed, it produces the following result −

```
Value of str is : Hello C++
```

The C++ compiler also determines the data type of variable to be output and selects the appropriate stream insertion operator to display the value. The << operator is overloaded to output data items of built-in types integer, float, double, strings and pointer values.

The insertion operator << may be used more than once in a single statement as shown above and **endl** is used to add a new-line at the end of the line.

The Standard Input Stream (cin)

The predefined object **cin** is an instance of **istream** class. The cin object is said to be attached to the standard input device, which usually is the keyboard. The **cin** is used in conjunction with the stream extraction operator, which is written as >> which are two greater than signs as shown in the following example.Live Demo

```
#include <iostream>

using namespace std;

int main() {
   char name[50];

   cout << "Please enter your name: ";
   cin >> name;
   cout << "Your name is: " << name << endl;

}
```

When the above code is compiled and executed, it will prompt you to enter a name. You enter a value and then hit enter to see the following result −

```
Please enter your name: cplusplus
Your name is: cplusplus
```

The C++ compiler also determines the data type of the entered value and selects the appropriate stream extraction operator to extract the value and store it in the given variables.

The stream extraction operator >> may be used more than once in a single statement. To request more than one datum you can use the following −

```
cin >> name >> age;
```

This will be equivalent to the following two statements −

```
cin >> name;
cin >> age;
```

New and delete operator

The new operator

The new operator requests for the memory allocation in heap. If the sufficient memory is available, it initializes the memory to the pointer variable and returns its address.

Here is the syntax of new operator in C++ language,

```
pointer_variable = new datatype;
```

Here is the syntax to initialize the memory,

```
pointer_variable = new datatype(value);
```

Here is the syntax to allocate a block of memory,

```
pointer_variable = new datatype[size];
```

Here is an example of new operator in C++ language,

Example

```
#include <iostream>
using namespace std;
int main () {
   int *ptr1  = NULL;
   ptr1 = new int;
   float *ptr2 = new float(223.324);
   int *ptr3 = new int[28];
   *ptr1 = 28;
   cout << "Value of pointer variable 1 : " << *ptr1 << endl;
   cout << "Value of pointer variable 2 : " << *ptr2 << endl;
   if (!ptr3)
      cout << "Allocation of memory failed\n";
   else {
      for (int i = 10; i < 15; i++)
         ptr3[i] = i+1;
      cout << "Value of store in block of memory: ";
      for (int i = 10; i < 15; i++)
         cout << ptr3[i] << " ";
   }
   return 0;
}
```

Output

```
Value of pointer variable 1 : 28
Value of pointer variable 2 : 223.324
Value of store in block of memory: 11 12 13 14 15
```

The delete operator

The delete operator is used to deallocate the memory. User has privilege to deallocate the created pointer variable by this delete operator.

Here is the syntax of delete operator in C++ language,

```
delete pointer_variable;
```

Here is the syntax to delete the block of allocated memory,

```
delete[ ] pointer_variable;
```

Here is an example of delete operator in C++ language,

Example

```
#include <iostream>
using namespace std;
int main () {
   int *ptr1  = NULL;
   ptr1 = new int;
   float *ptr2 = new float(299.121);
   int *ptr3 = new int[28];
   *ptr1 = 28;
   cout << "Value of pointer variable 1 : " << *ptr1 << endl;
   cout << "Value of pointer variable 2 : " << *ptr2 << endl;
   if (!ptr3)
      cout << "Allocation of memory failed\n";
   else {
      for (int i = 10; i < 15; i++)
         ptr3[i] = i+1;
      cout << "Value of store in block of memory: ";
      for (int i = 10; i < 15; i++)
         cout << ptr3[i] << " ";
   }
   delete ptr1;
   delete ptr2;
   delete[] ptr3;
   return 0;
}
```

Output

```
Value of pointer variable 1 : 28
Value of pointer variable 2 : 299.121
Value of store in block of memory: 11 12 13 14 15
```

OPERATORS

Operators are special type of functions, that takes one or more arguments and produces a new value. For example : addition (+), substraction (-), multiplication (*) etc, are all operators. Operators are used to perform various operations on variables and constants.

Types of operators

1. Assignment Operator

2. Mathematical Operators

3. Relational Operators

4. Logical Operators

5. Bitwise Operators

6. Shift Operators

7. Unary Operators

8. Ternary Operator

9. Comma Operator

Assignment Operator(=)

Operates '=' is used for assignment, it takes the right-hand side (called rvalue) and copy it into the left-hand side (called lvalue). Assignment operator is the only operator which can be overloaded but cannot be inherited.

Mathematical Operators

There are operators used to perform basic mathematical operations. Addition (+) , subtraction (-) , diversion (/) multiplication (*) and modulus (%) are the basic mathematical operators. Modulus operator cannot be used with floating-point numbers.

Relational Operators

These operators establish a relationship between operands. The relational operators are : less than (<) , grater thatn (>) , less than or equal to (<=), greater than equal to (>=), equivalent (==) and not equivalent (!=). You must notice that assignment operator is (=) and there is a relational operator, for equivalent (==). These two are different from each other, the assignment operator assigns the value to any variable, whereas equivalent operator is used to compare values, like in if-else conditions.

Logical Operators

The logical operators are AND (&&) and OR (||). They are used to combine two different expressions together.

If two statement are connected using AND operator, the validity of both statements will be considered, but if they are connected using OR operator, then either one of them must be valid. These operators are mostly used in loops (especially while loop) and in Decision making.

Bitwise Operators

There are used to change individual bits into a number. They work with only integral data types like char, int and long and not with floating point values.

- Bitwise AND operators &
- Bitwise OR operator |
- And bitwise XOR operator ^
- And, bitwise NOT operator ~

They can be used as shorthand notation too = , |= , ^= , ~= etc.

Shift Operators

Shift Operators are used to shift Bits of any variable. It is of three types,

1. Left Shift Operator <<
2. Right Shift Operator >>
3. Unsigned Right Shift Operator >>>

Unary Operators

These are the operators which work on only one operand. There are many unary operators, but increment ++ and decrement — operators are most used.

Other Unary Operators : address of &, dereference *, **new** and **delete**, bitwise not ~, logical not !, unary minus – and unary plus +.

Ternary Operator

The ternary if-else ? : is an operator which has three operands.

Comma Operator

This is used to separate variable names and to separate expressions. In case of expressions, the value of last expression is produced and used.

B.C.A study

Unit-2: Classes and objects

Encapsulation

Encapsulation is a process of combining data members and functions in a single unit called class. This is to prevent the access to the data directly, the access to them is provided through the functions of the class. It is one of the popular feature of Object Oriented Programming(OOPs) that helps in **data hiding**.

How Encapsulation is achieved in a class

To do this:
1) Make all the data members private.
2) Create public setter and getter functions for each data member in such a way that the set function set the value of data member and get function get the value of data member.

Let's see this in an example Program:

Encapsulation Example in C++

Here we have two data members num and ch, we have declared them as private so that they are not accessible outside the class, this way we are hiding the data. The only way to get and set the values of these data members is through the public getter and setter functions.

```
#include<iostream>
using namespace std;
class ExampleEncap{
private:
   int num;
   char ch;
public:
   int getNum() const {
      return num;
   }
   char getCh() const {
      return ch;
   }

  void setNum(int num) {
      this->num = num;
   }
   void setCh(char ch) {
      this->ch = ch;
   }
};
int main(){
   ExampleEncap obj;
   obj.setNum(100);
   obj.setCh('A');
   cout<<obj.getNum()<<endl;
   cout<<obj.getCh()<<endl;
   return 0;
}
```

Output:

```
100
A
```

information hiding

Information hiding or data hiding in programming is about protecting data or information from any inadvertent change throughout the program. Information hiding is a powerful OOP feature. Information hiding is closely associated with encapsulation.

Information hiding definition Information or data hiding is a programming concept which protects the data from direct modification by other parts of the program.

The feature of information hiding is applied using Class in most of the programming languages.

example of information hiding in C++:

```
class student
{
        char name[30];
        int marks;
     public:
        void display();
};

int main()
{
      student S;
      S.marks=50;  // Wrong code!
      getch();
      return 0;
}
```

The accessibility often plays an important role in information hiding. Here the data element *marks* is private element and thus it cannot be accessed by main function or any other function except the member function *display()* of class student. To make it accessible in main function, it should be made a *public* member.

Key Differences Between Data Hiding and Encapsulation

1. Encapsulation deals with hiding the **complexity** of a program. On the other hand, data hiding deals with the **security** of data in a program.

2. Encapsulation focuses on **wrapping** (encapsulating) the complex data in order to present a simpler view for the user. On the other hand, data hiding focuses on **restricting** the use of data, intending to assure the data security.
3. In encapsulation data can be **public or private** but, in data hiding, data must be **private** only.
4. Data hiding is a **process** as well as a technique whereas, encapsulation is **subprocess** in data hiding.

abstract data types

An **abstract data type** (or **ADT**) is a class that has a defined set of operations and values. In other words, you can create the starter motor as an entire **abstract data type**, protecting all of the inner code from the user. When the user wants to start the car, they can just execute the start() function.

In programming, an ADT has the following features:

- An ADT doesn't state how data is organized, and
- It provides only what's needed to execute its operations

An ADT is a prime example of how you can make full use of data abstraction and data hiding. This means that an abstract data type is a huge component of object-oriented programming methodologies: enforcing abstraction, allowing data hiding (protecting information from other parts of the program), and encapsulation (combining elements into a single unit, such as a class).

object and classes

OBJECT– In object-oriented programming languages like C++, the data and functions (procedures to manipulate the data) are bundled together as a self-contained unit called an object . An **Object** is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated

Class: The building block of C++ that leads to Object Oriented programming is a **Class**. It is a user defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object.
For Example: Consider the Class of **Cars**. There may be many cars with different names and brand but all of them will share some common properties like all of them will have *4 wheels*, *Speed Limit*, *Mileage range* etc. So here, Car is the class and wheels, speed limits, mileage are their properties.

- A Class is a user defined data-type which has data members and member functions.
- Data members are the data variables and member functions are the functions used to manipulate these variables and together these data members and member functions defines the properties and behavior of the objects in a Class.

- In the above example of class *Car*, the data member will be *speed limit*, *mileage* etc and member functions can be *apply brakes*, *increase speed* etc.

- 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.

The primary purpose of a class is to hold data/information. This is achieved with attributes which are also known as data members.

The member functions determine the behavior of the class i.e. provide a definition for supporting various operations on data held in form of an object.

Definition /declaration of a class

Syntax:

```
Class class_name
{
    Data Members;
    Methods;
}
```

Example:

```
class A
{
    public:
    double length; // Length of a box
    double breadth; // Breadth of a box
    double height; // Height of a box
}
```

- ***Private***, ***Protected***, ***Public*** is called visibility labels.
- The members that are declared private can be accessed only from within the class.
- Public members can be accessed from outside the class also.
- In C++, data can be hidden by making it private.

class members

Data and ***functions*** are members.

Data Members and methods must be declared within the class definition.Example:

```
Class A
{
    int i; // i is a data member of class A
    int j; // j is a data member of class A
    int i; // Error redefinition of i
}
```

- A member cannot be redeclared within a class.
- No member can be added elsewhere other than in the class definition.

Example:

```
Class A
{
    int i;
    int j;
    void f (int, int);
    int g();
}
```

f and *g* are a member function of ***class A***. They determine the behavior of the objects of ***class A***.

Accessing the Data Members

The public data members are also accessed in the same way given however the private data members are not allowed to be accessed directly by the object. Accessing a data member depends solely on the access control of that data member.
This access control is given by Access modifiers in C++. There are three access modifiers : **public, private and protected**.

state,identity,behaviour of an object

State: Represents data (value) of an object.

Behavior: Represents the behavior (functionality) of an object such as deposit, withdraw etc.

Identity: Object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. But,it is used internally by the JVM to identify each object uniquely.

Constructor in C++

A **Constructor** is a special member method which will be called implicitly (automatically) whenever an object of class is created. In other words, it is a member function which initializes a class which is called automatically whenever a new instance of a class is created.

Features of Constructor

- The same name as the class itself.
- no return type.

Syntax

```
classname()
{
....
}
```

Note: If you do not specify a constructor, the compiler generates a default constructor for you (expects no parameters and has an empty body).

Why use constructor ?

The main use of constructor is placing user defined values in place of default values.

How Constructor eliminate default values ?

Constructor are mainly used for eliminate default values by user defined values, whenever we create an object of any class then its allocate memory for all the data members and initialize there default values. To eliminate these default values by user defined values we use constructor.

Example of Constructor in C++

```
#include<iostream.h>
#include<conio.h>

class sum
{
int a,b,c;
sum()
{
a=10;
b=20;
c=a+b;
cout<<"Sum: "<<c;
}
};

void main()
{
sum s;
getch();
}
```

Output

```
Sum: 30
```

In above example when we create an object of “Sum” class then constructor of this class call and initialize user defined value in a=10 and b=20. And here we no need to call sum() constructor.

Destructor

Destructor is a member function which deletes an object. A destructor function is called automatically when the object goes out of scope:

When destructor call

- when program ends
- when a block containing temporary variables ends
- when a delete operator is called

Features of destructor

- The same name as the class but is preceded by a tilde (~)
- no arguments and return no values

Syntax

```
~classname()
{
......
}
```

Note: If you do not specify a destructor, the compiler generates a default destructor for you.

Example of Destructor in C++

```
#include<iostream.h>
#include<conio.h>

class sum
{
int a,b,c;
sum()
{
a=10;
b=20;
c=a+b;
cout<<"Sum: "<<c;
}
~sum()
{
cout<<<<endl;"call destructor";
}
delay(500);
};

void main()
{
sum s;
cout<<<<endl;"call main";
getch();
}
```

Output

```
Sum: 30
call main
call destructor
```

Explanation: In above example when you create object of class sum auto constructor of class is call and after that control goes inside main and finally before end of program destructor is call.

B.C.A study

Unit 3 Inheritance and polymorphism

Inheritance

In C++, it is possible to inherit attributes and methods from one class to another. We group the "inheritance concept" into two categories:

- **derived class** (child) – the class that inherits from another class
- **base class** (parent) – the class being inherited from

Implementing inheritance in C++: For creating a sub-class which is inherited from the base class we have to follow the below syntax.
Syntax:

```
class subclass_name : access_mode base_class_name
{
  //body of subclass
};
```

Here, **subclass_name** is the name of the sub class, **access_mode** is the mode in which you want to inherit this sub class for example: public, private etc. and **base_class_name** is the name of the base class from which you want to inherit the sub class.
Note: A derived class doesn't inherit ***access*** to private data members. However, it does inherit a full parent object, which contains any private members which that class declares.

class derivation list names one or more base classes and has the form −

```
class derived-class: access-specifier base-class
```

Where access-specifier is one of **public, protected,** or **private**, and base-class is the name of a previously defined class. If the access-specifier is not used, then it is private by default.

Consider a base class **Shape** and its derived class **Rectangle** as follows −

```
#include <iostream>

using namespace std;

// Base class
class Shape {
   public:
      void setWidth(int w)   {
         width = w;
      }
      void setHeight(int h)  {
         height = h;
      }

   protected:
      int width;
      int height;
};

// Derived class
class Rectangle: public Shape {
   public:
      int getArea()          {
     return (width * height);
      }
};

int main(void)               {
   Rectangle Rect;

   Rect.setWidth(5);
   Rect.setHeight(7);
   cout << "Total area: " << Rect.getArea() << endl;

   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Total area: 35
```

Access Control and Inheritance

A derived class can access all the non-private members of its base class. Thus base-class members that should not be accessible to the member functions of derived classes should be declared private in the base class.

We can summarize the different access types according to – who can access them in the following way −

Access	public	protected	private
Same class	yes	yes	yes
Derived classes	yes	yes	no
Outside classes	yes	no	no

A derived class inherits all base class methods with the following exceptions −

- Constructors, destructors and copy constructors of the base class.
- Overloaded operators of the base class.
- The friend functions of the base class

We hardly use **protected** or **private**inheritance, but **public** inheritance is commonly used. While using different type of inheritance, following rules are applied −

- **Public Inheritance** − When deriving a class from a **public** base class, **public**members of the base class become **public** members of the derived class and **protected** members of the base class become **protected** members of the derived class. A base class's **private** members are never accessible directly from a derived class, but can be accessed through calls to the **public** and **protected**members of the base class.
- **Protected Inheritance** − When deriving from a **protected** base class, **public** and **protected** members of the base class become **protected**members of the derived class.
- **Private Inheritance** − When deriving from a **private** base class, **public** and **protected** members of the base class become **private** members of the derived class.

Types of Inheritance in C++

1) Single inheritance
2) Multilevel inheritance
3) Multiple inheritance
4) Hierarchical inheritance
5) Hybrid inheritance

Single inheritance

In Single inheritance one class inherits one class exactly.
For example: Lets say we have class A and B

```
B inherits A
```

Example of Single inheritance:

```
#include <iostream>
using namespace std;
class A {
public:
  A(){
     cout<<"Constructor of A class"<<endl;
  }
};
class B: public A {
public:
  B(){
     cout<<"Constructor of B class";
  }
};
int main() {
   //Creating object of class B
   B obj;
   return 0;
}
```

Output:

```
Constructor of A class
Constructor of B class
```

2)Multilevel Inheritance

In this type of inheritance one class inherits another child class.

```
C inherits B and B inherits A
```

Example of Multilevel inheritance:

```
#include <iostream>
using namespace std;
class A {
public:
  A(){
     cout<<"Constructor of A class"<<endl;
  }
};
class B: public A {
public:
  B(){
     cout<<"Constructor of B class"<<endl;
  }
};
class C: public B {
public:
  C(){
     cout<<"Constructor of C class"<<endl;
  }
};
int main() {
  //Creating object of class C
  C obj;
  return 0;
}
```

Output:

```
Constructor of A class
Constructor of B class
Constructor of C class
```

Multiple Inheritance

In multiple inheritance, a class can inherit more than one class. This means that in this type of inheritance a single child class can have multiple parent classes.
For example:

```
C inherits A and B both
```

Example of Multiple Inheritance:

```
#include <iostream>
using namespace std;
class A {
public:
  A(){
     cout<<"Constructor of A class"<<endl;
  }
};
class B {
public:
  B(){
     cout<<"Constructor of B class"<<endl;
  }
};
class C: public A, public B {
public:
  C(){
     cout<<"Constructor of C class"<<endl;
  }
};
int main() {
   //Creating object of class C
   C obj;
   return 0;
}
```

```
Constructor of A class
Constructor of B class
Constructor of C class
```

4)Hierarchical Inheritance

In this type of inheritance, one parent class has more than one child class. For example:

```
Class B and C inherits class A
```

Example of Hierarchical inheritance:

```
#include <iostream>
using namespace std;
class A {
public:
  A(){
     cout<<"Constructor of A class"<<endl;
  }
};
class B: public A {
public:
  B(){
     cout<<"Constructor of B class"<<endl;
  }
};
class C: public A{
public:
  C(){
     cout<<"Constructor of C class"<<endl;
  }
};
int main() {
   //Creating object of class C
   C obj;
   return 0;
}
```

Output:

```
Constructor of A class
Constructor of C class
```

5) Hybrid Inheritance

Hybrid inheritance is a combination of more than one type of inheritance. For example, A child and parent class relationship that follows multiple and hierarchical inheritance both can be called hybrid inheritance.

Polymorphism in C++

Polymorphism is a feature of OOPSthat allows the object to behave differently in different conditions. In C++ we have two types of polymorphism:
1) Compile time Polymorphism – This is also known as static (or early) binding.
2) Runtime Polymorphism – This is also known as dynamic (or late) binding.

1) Compile time Polymorphism

Function overloading and Operator overloading are perfect example of Compile time polymorphism.

Compile time Polymorphism Example

In this example, we have two functions with same name but different number of arguments. Based on how many parameters we pass during function call determines which function is to be called, this is why it is considered as an example of polymorphism because in different conditions the output is different. Since, the call is determined during compile time thats why it is called compile time polymorphism.

```
#include <iostream>
using namespace std;
class Add {
public:
  int sum(int num1, int num2){
     return num1+num2;
  }
  int sum(int num1, int num2, int num3){
     return num1+num2+num3;
  }
};
int main() {
  Add obj;
  //This will call the first function
  cout<<"Output: "<<obj.sum(10, 20)<<endl;
  //This will call the second function
  cout<<"Output: "<<obj.sum(11, 22, 33);
  return 0;
}
```

Output:

```
Output: 30
Output: 66
```

2) Runtime Polymorphism

Function overriding is an example of Runtime polymorphism.
Function Overriding: When child class declares a method, which is already present in the parent class then this is called function overriding, here child class overrides the parent class.

In case of function overriding we have two definitions of the same function, one is parent class and one in child class. The call to the function is determined at **runtime** to decide which definition of the function is to be called, thats the reason it is called runtime polymorphism.

Example of Runtime Polymorphism

```
#include <iostream>
using namespace std;
class A {
public:
  void disp(){
     cout<<"Super Class Function"<<endl;
  }
};
class B: public A{
public:
  void disp(){
     cout<<"Sub Class Function";
  }
};
int main() {
  //Parent class object
  A obj;
  obj.disp();
  //Child class object
  B obj2;
  obj2.disp();
  return 0;
}
```

Output:

```
Super Class Function
Sub Class Function
```

Function overloading in C++

Function overloading is a C++ programming feature that allows us to have more than one function having same name but different parameter list, when I say parameter list, it means the data type and sequence of the parameters, for example the parameters list of a function `myfuncn(int a, float b)` is `(int, float)` which is different from the function `myfuncn(float a, int b)` parameter list `(float, int)`. Function overloading is a compile time polymorphism
Now that we know what is parameter list lets see the rules of overloading: we can have following functions in the same scope.

```
sum(int num1, int num2)
sum(int num1, int num2, int num3)
sum(int num1, double num2)
```

> *The easiest way to remember this rule is that the parameters should qualify any one or more of the following conditions, they should have different **type**, **number** or **sequence** of parameters.*

For example:
These two functions have different parameter **type**:

```
sum(int num1, int num2)
sum(double num1, double num2)
```

These two have different **number** of parameters:

```
sum(int num1, int num2)
sum(int num1, int num2, int num3)
```

These two have different **sequence** of parameters:

```
sum(int num1, double num2)
sum(double num1, int num2)
```

All of the above three cases are valid case of overloading. We can have any number of functions, just remember that the parameter list should be different. For example:

```
int sum(int, int)
double sum(int, int)
```

This is not allowed as the parameter list is same. Even though they have different return types, its not valid.

Function overloading Example

Lets take an example to understand function overloading in C++.

```
#include <iostream>
using namespace std;
class Addition {
public:
    int sum(int num1,int num2) {
        return num1+num2;
    }
    int sum(int num1,int num2, int num3) {
       return num1+num2+num3;
    }
};
int main(void) {
    Addition obj;
    cout<<obj.sum(20, 15)<<endl;
    cout<<obj.sum(81, 100, 10);
    return 0;
}
```

Output:

```
35
191
```

Advantages of Function overloading

The main advantage of function overloading is to the improve the **code readability** and allows **code reusability**. In the example 1, we have seen how we were able to have more than one function for the same task(addition) with different parameters, this allowed us to add two integer numbers as well as three integer numbers, if we wanted we could have some more functions with same name and four or five arguments.
Imagine if we didn't have function overloading, we either have the limitation to add only two integers or we had to write different name functions for the same task addition, this would reduce the code readability and reusability.

Function Overriding in C++

Function overriding is a feature that allows us to have a same function in child class which is already present in the parent class. A child class inherits the data members and member functions of parent class, but when you want to override a functionality in the child class then you can use function overriding. It is like creating a new version of an old function, in the child class.

Function Overriding Example

To override a function you must have the same signature in child class. By signature I mean the data type and sequence of parameters. Here we don't have any parameter in the parent function so we didn't use any parameter in the child function.

```
#include <iostream>
using namespace std;
class BaseClass {
public:
   void disp(){
      cout<<"Function of Parent Class";
   }
};
class DerivedClass: public BaseClass{
public:
   void disp() {
      cout<<"Function of Child Class";
   }
};
int main() {
   DerivedClass obj = DerivedClass();
   obj.disp();
   return 0;
}
```

Output:

```
Function of Child Class
```

> ***Note:*** *In function overriding, the function in parent class is called the overridden function and function in child class is called overriding function.*

Difference between function overloading and function overriding

Now that we understand what is function overloading and overriding in c++programming lets see the difference between them:

1) Function Overloading happens in the same class when we declare same functions with different arguments in the same class. Function Overriding is happens in the child class when child class overrides parent class function.
2) In function overloading function signature should be different for all the overloaded functions. In function overriding the signature of both the functions (overriding function and overridden function)

should be same.
3) Overloading happens at the compile time thats why it is also known as compile time polymorphism while overriding happens at run time which is why it is known as run time polymorphism.
4) In function overloading we can have any number of overloaded functions. In function overriding we can have only one overriding function in the child class.

Overloading in C++

If we create two or more member of the same class having the same name but different in number or type of parameter,it is known as C++ overloading.

For example, we can overload an operator '+' in a class like String so that we can concatenate two strings by just using +.
Other example classes where arithmetic operators may be overloaded are Complex Number, Fractional Number, Big Integer, etc.

B.C.A study

Unit-4: Generic Function

Generics in C++

Generics is the idea to allow type (Integer, String, … etc and user-defined types) to be a parameter to methods, classes and interfaces. For example, classes like an array, map, etc, which can be used using generics very efficiently. We can use them for any type.

The method of Generic Programming is implemented to increase the efficiency of the code. Generic Programming enables the programmer to write a general algorithm which will work with all data types. It eliminates the need to create different algorithms if the data type is an integer, string or a character.

The advantages of Generic Programming are

1. Code Reusability
2. Avoid Function Overloading
3. Once written it can be used for multiple times and cases

Templates

Templates are powerful features of C++ which allows you to write generic programs. In simple terms, you can create a single function or a class to work with different data types using templates.

Templates are often used in larger codebase for the purpose of code reusability and flexibility of the programs.

The concept of templates can be used in two different ways:

- Function Templates
- Class Templates

Function Templates

A function template works in a similar to a normal function, with one key difference.

A single function template can work with different data types at once but, a single normal function can only work with one set of data types.

Normally, if you need to perform identical operations on two or more types of data, you use function overloading to create two functions with the required function declaration.

However, a better approach would be to use function templates because you can perform the same task writing less and maintainable code.

How to declare a function template?

A function template starts with the keyword **template** followed by template parameter/s inside **< >**which is followed by function declaration.

```
template <class T>
T someFunction(T arg)
{
   ... .. ...
}
```

In the above code T is a template argument that accepts different data types (int, float), and **class** is a keyword.

You can also use keyword **typename**instead of class in the above example.

When, an argument of a data type is passed to `someFunction( )`, compiler generates a new version of `someFunction()` for the given data type.

Class Templates

Like function templates, you can also create class templates for generic class operations.

Sometimes, you need a class implementation that is same for all classes, only the data types used are different.

Normally, you would need to create a different class for each data type OR create different member variables and functions within a single class.

This will unnecessarily bloat your code base and will be hard to maintain, as a change is one class/function should be performed on all classes/functions.

However, class templates make it easy to reuse the same code for all data types.

How to declare a class template?

```
template <class T>
class className
{
   ... .. ...
public:
   T var;
   T someOperation(T arg);
   ... .. ...
};
```

In the above declaration, `T` is the template argument which is a placeholder for the data type used.

Inside the class body, a member variable and a member function `someOperation()` are both of type `T`.

How to create a class template object?

To create a class template object, you need to define the data type inside a `< >` when creation.

```
className<dataType> classObject;
```

For example:

```
className<int> classObject;
className<float> classObject;
className<string> classObject;
```

Function overloading

Creating two or more members that have the same name but are different in number or type of parameter is known as **overloading**.

In C++, we can overload:

- Methods
- Constructors
- Indexed Properties

The process of having two or more functions with the same name, but different parameters, is known as **function overloading**.

The function is redefined by either using different types of arguments or a different number of arguments. It is only through these differences that a compiler can differentiate between functions.

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

Function overloading example

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

#include<iostream.h>.

```
using namespace std;
class Cal {
public:
static int add(int a,int b){
return a + b;
}
static int add(int a, int b, int c)
{
return a + b + c;
}
};
int main(void) {
Cal C; //class object declaration. // class object declaration.
cout<<C.add(10, 20)<<endl;
cout<<C.add(12, 20, 23);
return 0;
}
```

Explanation

- We have created two `add()`methods that have two different numbers of arguments.
- Now, in the `main()` function, we call these two methods by passing different numbers of arguments through the same call `c.add` .
- The compiler will differentiate based on the number of passing arguments, and execute the correct method accordingly.

B.C.A study

Unit-5:File and Exception Handling

File Handling using File Streams in C++

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:

- Creating a file: `open()`
- Reading data: `read()`
- Writing new data: `write()`
- 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:\studytonight.txt",ios::out);`

Example of file opened for reading:

st.open("E:\studytonight.txt",ios::in)

Example of file opened for appending:

st.open("E:\studytonight.txt",ios::app)

Example of file opened for truncating:

st.open("E:\studytonight.txt",ios::trunc)

```
#include<iostream.h> #include<conio.h> #include <fstream.h> using namespace std; int main() {

fstream st; st.open("E\studytonight.txt",ios::out)

if(!st)

{ cout<<"File creation failed";

} else { cout<<"New file created"; st.close();

} getch(); return 0;

}
```

Writing to a File

```
#include <iostream> #include<conio> #include <fstream> using namespace std; int main() { fstream st; st.open("E\studytonight.txt",ios::out); if(!st) { cout<<"File creation failed"; }

else

{

cout<<"New file created";

st<<"Hello";

st.close();

Closing file
```

```
}

getch(); return 0;

}
```

Here we are sending output to a file. So, we use ios::out. As given in the program, information typed inside the quotes after **"FilePointer <<"** will be passed to output file.

Reading from a File

```
#include <iostream> #include<conio> #include <fstream> using namespace std; int main() { fstream st;

st.open("E\studytonight.txt",ios::in)


if(!st)

{ cout<<"No such file";

} else

{ char ch; while (!st.eof()) { st >>ch;

cout << ch;

}

st.close();

}

getch();

return 0;

}
```

Here we are reading input from a file. So, we use ios::in. As given in the program, information from the output file is obtained with the help of following syntax **"FilePointer >>variable"**.

Close a File

It is done by `FilePointer.close()`.

```
#include <iostream> #include<conio>
```

```
#include <fstream>

using namespace std;

int main()

{ fstream st;

st.open("E\studytonight.txt",ios::out)

st.close()

getch();

return 0;

}
```

Special operations in a File

There are few important functions to be used with file streams like:

- `tellp()` – It tells the current position of the put pointer. **Syntax:** filepointer.tellp()
- `tellg()` – It tells the current position of the get pointer. **Syntax:** filepointer.tellg()
- `seekp()` – It moves the put pointer to mentioned location. **Syntax:** filepointer.seekp(no of bytes,reference mode)
- `seekg()` – It moves get pointer(input) to a specified location. **Syntax:** filepointer.seekg((no of bytes,reference point)
- `put()` – It writes a single character to file.
- `get()` – It reads a single character from file.

> ***Note:*** *For seekp and seekg three reference points are passed:*
> ***ios::beg*** *– beginning of the file*
> ***ios::cur*** *– current position in the file*
> ***ios::end*** *– end of the file*