

SUB NAME : OBJECT ORIENTED
PROGRAMMING USING C++
SUB CODE : 231CS34
COURSE TUTOR : Ms.M.JAYAPRIYA AP/CSE

231CS34	OBJECT ORIENTED PROGRAMMING USING C++				L	T	P	C
					3	0	0	3
Programme:	B.E. Computer Science and Engineering	Sem:	3	Category:	PC			
Prerequisites:	231CS21 – Programming in C							
Aim:	To enable the students to tackle complex programming problems, making good use of the object-oriented programming paradigm to simplify the design and implementation process.							
Course Outcomes:	The Students will be able to							
CO1:	Classify object oriented programming and procedural programming.							
CO2:	Apply C++ features such as composition of objects, operator overloads, dynamic memory allocation.							
CO3:	Make use of inheritance and polymorphism concept in developing a C++ projects.							
CO4:	Identify the exceptions and its handling techniques.							
CO5:	Demonstrate the concept of templates and generic functions.							
CO6:	Apply the concept of standard template library.							
BASIC CONCEPTS								9
Object oriented programming concepts – Introduction to C++ - Classes and objects: classes - structures and classes - unions and classes - friend functions – friend classes - inline functions -static class members - scope resolution operator - nested classes - local classes -passing objects to functions - returning objects – object assignment. Arrays, Pointers, References and Dynamic Allocation Operators.								
FUNCTION OVERLOADING AND CONSTRUCTORS								9
Function Overloading – Overloading Constructors – Copy Constructors– Destructors Operator overloading: Creating a member Operator Function – Operator Overloading Using Friend Function – Overloading New and Delete – Overloading Special Operators – Overloading Comma Operator.								
INHERITANCE AND POLYMORPHISM								9
Inheritance: Base-Class Access Control – Inheritance and Protected Members – Inheriting Multiple Base Classes –Virtual Base Classes. Polymorphism: Virtual Functions — Pure Virtual Functions – Using Virtual Functions – Early vs. Late Binding. Run-Time Type ID and Casting Operators: RTTI – Casting Operators –Dynamic Cast.								
TEMPLATES AND EXCEPTION HANDLING								9
Templates: Generic Functions – Applying Generic Functions – Generic Classes – Type name and Export Keywords – Power of Templates. Exception Handling: Fundamentals –Handling Derived Class Exceptions – Exception Handling Options – Understanding terminate() and unexpected() – uncaught exception() Function – Exception and bad exception Classes – Applying Exception Handling.								
I/O STREAMS STANDARD TEMPLATE LIBRARY								9
File I/O-<fstream> and the File Classes-Opening and Closing a File-Reading and Writing Text Files-Unformatted and Binary I/O. Standard Template Library: Overview – Container Classes – General Theory of Operation –Vectors- Lists-Maps – String Class								
Total Periods								45
Text Books:								
1. Robert Lafore, “ <i>Object Oriented Programming in C++</i> ”, Tech media Publication, 4/e, 2008. 2. E. Balagurusamy, “ <i>Object oriented Programming with C++</i> ”, Tata McGraw-Hill, 6/e, 2013. 3. Herbert shield, “ <i>The complete reference C++</i> ”, McGraw Hill Publication, 4/e, 2017.								

References:	
<ol style="list-style-type: none"> 1. Fourth Edition Joyce Farrell , “<i>Object-Oriented Programming Using C++</i>”, 2009, Cengage Learning products are represented in Canada by Nelson Education, Ltd. 2. Ashok N Kamthane, “<i>Programming in C++</i>” , Pearson India, 2/e, 2013 3. B. Trivedi, “<i>Programming with ANSI C++</i>”, Oxford University Press, 2012. 4. K.R Venugopal and Rajkumar Buyya, “<i>Mastering C++</i>”, Tata McGraw Hill, 2/e, 2013. 5. https://onlinecourses.nptel.ac.in/noc24_cs44/preview 	

231CS34		PO	PO	PO	PO	PO	PO	PO	PO	PO	PO	PO	PSO	PSO	PSO
		1	2	3	4	5	6	7	8	9	10	11	12	1	2
		K3	K2	K3	K3								K5	K2	K3
CO1	K2	2	3	2	2								1	3	2
CO2	K3	3	3	3	3								1	3	3
CO3	K3	3	3	3	3								1	3	3
CO4	K3	3	3	3	3								1	3	3
CO5	K2	2	3	2	2								1	3	2
CO6	K3	3	3	3	3								1	3	3

1: Slight (Low) 2: Moderate (Medium) 3: Substantial (High)

UNIT-1

BASIC CONCEPTS

Object oriented programming

- OOP stands for Object-Oriented Programming.
- Object-oriented programming (OOP) is a **computer programming model** that **organizes software design** around data, or objects, rather than functions and logic. An **object can be defined as a data field** that has **unique attributes and behavior**.
- **Procedural programming** is about **writing procedures or functions** that perform operations on the data, while **object-oriented programming** is about **creating objects** that contain both data and functions.
- Object-oriented programming has several **advantages** over procedural programming:
 1. OOP is **faster and easier** to execute
 2. OOP provides a **clear structure** for the programs
 3. OOP helps to keep the C++ code DRY "Don't Repeat Yourself", and makes the code **easier to maintain, modify and debug**
 4. OOP makes it possible to **create full reusable applications** with less code and shorter development time

Object oriented programming concepts

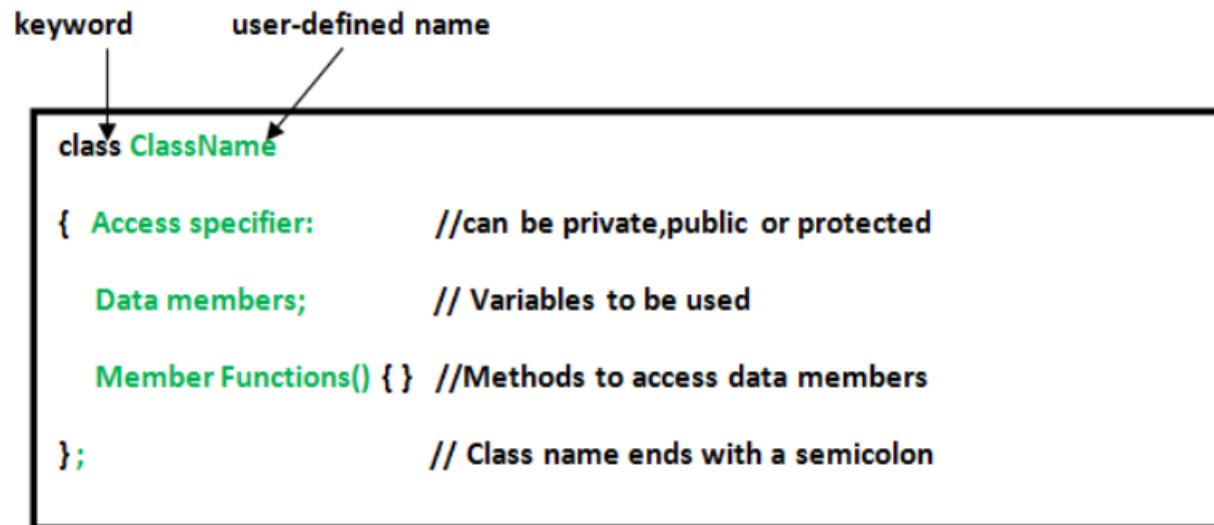
- Object-oriented programming – As the name suggests uses **objects in programming**. Object-oriented programming **aims to implement real-world entities** like inheritance, hiding, polymorphism, etc. in programming.
- The main aim of OOP is to bind together the **data and the functions that operate** on them so that no other part of the **code can access this data except that function**.
- There are some basic concepts that act as the building blocks of OOPs
 1. Class
 2. Objects
 3. Encapsulation
 4. Abstraction
 5. Polymorphism
 6. Inheritance
 7. Dynamic Binding
 8. Message Passing

CLASS

- A class is a **group of similar objects**.
- 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 brands but all of them will share some common properties like all of them will have 4 wheels, Speed Limit, Mileage range, etc.
- So here, the Car is the class, and wheels, speed limits, and mileage are their properties.
- A Class is a user-defined data type that has data members and member functions.
- **Data members are the data variables** and **member functions are the functions** used to **manipulate these variables** together these data members and member functions define the properties and behavior of the objects in a Class.

Defining Class

A class is defined in C++ using the **keyword class** followed by the name of the class. The body of the class is defined inside the curly brackets and terminated by a semicolon at the end.



The diagram shows a C++ class definition with two annotations: 'keyword' pointing to 'class' and 'user-defined name' pointing to 'ClassName'. The class body is enclosed in curly braces and contains three lines of code with comments: an access specifier, data members, and member functions. The class definition ends with a semicolon.

```
class ClassName  
{ Access specifier: //can be private,public or protected  
  Data members; // Variables to be used  
  Member Functions() { } //Methods to access data members  
}; // Class name ends with a semicolon
```

OBJECT

- **Object is an instance of a class** that encapsulates data and functionality pertaining to that data.
- Object is a **real-world entity such as book, car, etc.** **Class is a logical entity.** Object is a **physical entity.**
- An **Object is an identifiable entity** with some characteristics and behavior. 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.**

Declaring Objects

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

Syntax

```
ClassName ObjectName;
```

ENCAPSULATION

- In normal terms, Encapsulation is defined as **wrapping up data and information under a single unit.**
- In Object-Oriented Programming, Encapsulation is defined as **binding together the data and the functions that manipulate them.**
- Encapsulation is a way to **restrict the direct access to some components of an object**, so users cannot access state values for all of the variables of a particular object.
- Encapsulation can be used to **hide both data members and data functions** or methods associated with an instantiated class or object.
- Encapsulation also leads to *data abstraction or data hiding*. Using encapsulation also **hides the data.**
- encapsulation involves **combining similar data and functions into a single unit** called a **class**. By encapsulating these **functions and data**, we **protect** that data from **change**. This concept is also known as **data or information hiding**.
- Encapsulation is defined as binding together the data and the functions that manipulate them.

- Consider a **real-life example of encapsulation**, in a company, there are **different sections like the accounts section, finance section, sales section**, etc. Now,
- The finance section handles all the financial transactions and keeps records of all the data related to finance.
- Similarly, the sales section handles all the sales-related activities and keeps records of all the sales.
- Now there may arise a situation when for some reason an official from the finance section needs all the data about sales in a particular month.
- In this case, he is not allowed to directly access the data of the sales section. He will first have to contact some other officer in the sales section and then request him to give the particular data.
- This is what **Encapsulation** is. Here the data of the sales section and the employees that can manipulate them are wrapped under a single name “sales section”.

ABSTRACTION

- Data abstraction is one of the most essential and important features of object-oriented programming in C++.
- Abstraction means **displaying only essential information** and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, **hiding the background details or implementation.**

POLYMORPHISM

- Polymorphism means "**many forms**", and it occurs when we have **many classes that are related to each other by inheritance**.
- The word polymorphism means having **many forms**. In simple words, we can define polymorphism as the **ability of a message to be displayed in more than one form**.
- A **person** at the same time can have different characteristics. A man at the same time is a **father, a husband, and an employee**. So the same person possesses different behavior in different situations.
- This is called polymorphism. An operation may exhibit different behaviors in different instances. The behavior depends upon the types of data used in the operation. C++ **supports operator overloading and function overloading**.
- *Operator Overloading*: The process of making an **operator exhibit different behaviors in different instances** is known as operator overloading.
- *Function Overloading*: Function overloading is using a **single function name to perform different types of tasks**. Polymorphism is extensively used in implementing inheritance.

INHERITANCE

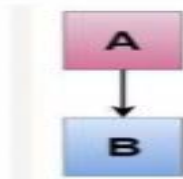
- Inheritance is a mechanism in which **one class acquires the property of another class**. For example, a **child inherits the traits of his/her parents**. With inheritance, we can reuse the fields and methods of the existing class.
- The capability of a class to derive properties and characteristics from another class is called Inheritance. Inheritance is one of the most important features of Object-Oriented Programming.
- **Sub Class**: The class that **inherits properties from another** class is called Sub class or Derived Class.
- **Super Class**: The class whose **properties are inherited by a sub-class** is called Base Class or Superclass.
- **Reusability**: Inheritance supports the concept of “reusability”, i.e. when we want to **create a new class and there is already a class** that includes some of the code that we want, we can derive our new class from the existing class.

TYPES

- ▶ Single Inheritance
- ▶ Multiple Inheritance
- ▶ Multilevel Inheritance
- ▶ Hierarchical Inheritance
- ▶ Hybrid Inheritance

Single Inheritance

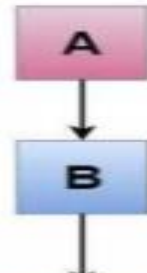
- ▶ **Single inheritance** is defined as the inheritance in which a derived class is inherited from the only one base class.



Where 'A' is the base class, and 'B' is the derived class.

C++ Multilevel Inheritance

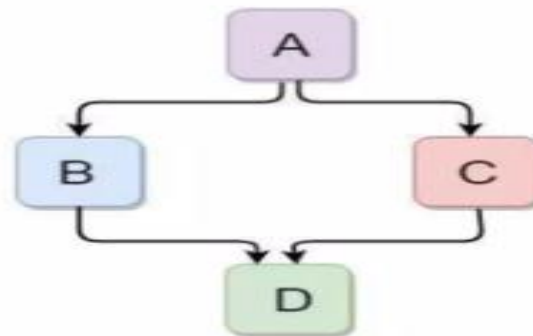
- ▶ **Multilevel inheritance** is a process of deriving a class from another derived class.



When one class inherits another class which is further inherited by another class, it is known as multi level inheritance in C++. Inheritance is transitive so the last derived class acquires all the members of all its base classes.

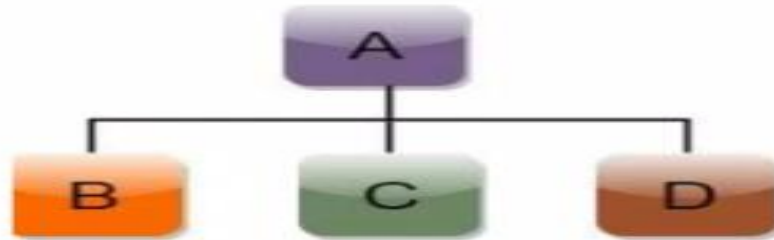
C++ Hybrid Inheritance

- ▶ **Hybrid inheritance** is a combination of more than one type of inheritance.



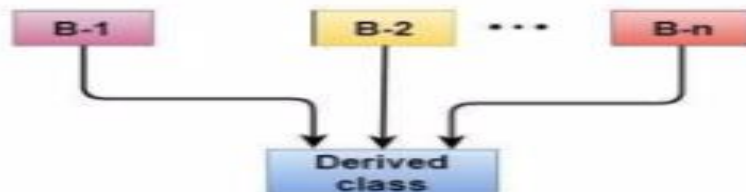
C++ Hierarchical Inheritance

- ▶ Hierarchical inheritance is defined as the process of deriving more than one class from a base class.



C++ Multiple Inheritance

- ▶ Multiple inheritance is the process of deriving a new class that inherits the attributes from two or more classes.



DYNAMIC BINDING

- Dynamic binding is also known as **late binding or runtime binding**. It means that the code that **executes a method or an operator is determined at runtime, not at compile time**.
- In simple terms, Dynamic binding is the **connection between the function declaration and the function call**.
- You can specify that the **compiler match a function call with the correct function** definition at **runtime**; this is called dynamic binding.
- This allows the **same method or operator** to have **different behaviors** depending on the type or state of the objects that invoke or interact with it.
- In dynamic binding, the code to be executed in response to the function call is decided at runtime. C++ has **virtual functions** to support this.
- Because dynamic binding is **flexible, it avoids the drawbacks of static binding**, which connected the function call and definition at build time.

Static Binding Vs Dynamic Binding

Static Binding	Dynamic Binding
It takes place at compile time which is referred to as early binding	It takes place at runtime so it is referred to as late binding
Execution of static binding is faster than dynamic binding because of all the information needed to call a function.	Execution of dynamic binding is slower than static binding because the function call is not resolved until runtime.
It takes place using normal function calls, operator overloading, and function overloading.	It takes place using virtual functions
Real objects never use static binding	Real objects use dynamic binding

MESSAGE PASSING

- Objects **communicate with one another** by sending and receiving information.
- A **message for an object is a request** for the execution of a procedure and therefore will invoke a function in the **receiving object that generates the desired results**.
- Message passing involves specifying the name of the object, the name of the function, and the information to be sent.
- Message passing in C++ is the process by which objects communicate by **exchanging messages**, typically in the form of method or function calls. This mechanism enables **objects to interact and collaborate**, facilitating the accomplishment of desired objectives.
- Message passing in C++ involves **passing objects or data between different parts** of a program **using functions, methods, or message queues**. Objects communicate by sending and receiving data.
- A message for an object is a request for the execution of a procedure, and as such, it will invoke a function in the receiving object that produces the desired results. Message passing entails specifying the object's name, the function's name, and the data to be sent.

C++ Introduction

- ❑ C++ is an object-oriented programming language which allows code to be reused, lowering development costs.
- ❑ C++ was developed by Bjarne Stroustrup, as an extension to the C language.
- ❑ C++ gives programmers a high level of control over system resources and memory.

Characteristics of C++:

- ❑ **Object-Oriented Programming:**
 - It allows the programmer to design applications like a communication between object rather than on a structured sequence of code.
- ❑ **Portability:**
 - We can compile the same C++ code in almost any type of computer & operating system without making any changes.
- ❑ **Modular Programming:**
 - An application's body in C++ can be made up of several source code files that are compiled separately and then linked together saving time.
- ❑ **C Compatibility:**
 - Any code written in C can easily be included in a C++ program without making any changes.

- **Speed:**
 - The resulting code compilation is very efficient due to its duality as high-level and low-level language.
- **Flexibility:**
 - It is highly flexible language and versatility.
- **Wide range of library functions:**
 - It has huge library functions; it reduces the code development time and also reduces cost of software development.
- **System Software Development:**
 - It can be used for developing System Software Viz., Operating system, Compilers, Editors and Database.

General structure of c++ program

- Different programming languages have their own format of coding.
- The basic components of a C++program are:
 - **Comments or Documentation Section**
 - **Pre-processor Directives (Linker Section):**
 - **Definition**
 - **Global Declaration**
 - **main () function**
 - **Declarations**
 - **Statements**

Documentation Section:

- This section comes first and is **used to document the logic of the program** that the programmer going to code.
- It can be also used to write for purpose of the program.
- Whatever **written in the documentation section is the comment** and is **not compiled by the compiler.**
- Documentation Section is **optional**

Linking Section:

- The linking section contains two parts:

Header Files:

- Generally, a program includes various programming elements like **built-in functions, classes, keywords, constants, operators**, etc. that are already defined in the standard C++ library.
- In order to use such **pre-defined elements in a program**, an appropriate **header must be included in the program**.
- Standard headers are specified in a program through the **preprocessor directive #include**. In Figure, the iostream header is used. When the **compiler processes the instruction #include<iostream>**, it includes the contents of the stream in the program. This **enables the programmer to use standard input, output**, and error facilities that are provided only through the standard streams defined in <iostream>.

Syntax: include <*library_name*>

Namespaces:

- A namespace **permits grouping of various entities like classes, objects, functions, and various C++ tokens**, etc. under a single name.
- Any user can **create separate namespaces of its own** and can **use them in any other program**.
- In the below snippets, **namespace std** contains declarations for cout, cin, endl, etc. statements.

Syntax: using namespace std;

Definition Section:

- It is used to **declare some constants and assign them some value.**
- In this section, anyone can define your **own datatype using primitive data types.**
- In **#define** is a **compiler directive** which tells the compiler whenever the message is found to replace it with “Factorial\n”.
- **typedef int INTEGER;** this statement tells the compiler that whenever you will encounter **INTEGER** replace it by int and as you have **declared INTEGER as datatype** you cannot use it as an identifier.

Global Declaration Section:

- Here, the **variables and the class definitions** which are going to be used in the program are **declared to make them global.**
- The **scope of the variable declared** in this section lasts until the **entire program terminates.**
- These variables are accessible within the user-defined functions

Main Function:

- The main function tells the compiler where to **start the execution of the program**. The execution of the program starts with the main function.
- All the statements that are to be **executed are written in the main** function.
- The compiler **executes all the instructions** which are written in the **curly braces {}** which encloses the body of the main function.
- Once all instructions from the main function are executed, control comes out of the main function and the **program terminates and no further execution** occur.

Syntax:

```
int main() {  
  
    ... code ....  
    return 0;  
}
```

General Structure of C++ Program

- Different programming languages have their own format of coding.
- The basic components of a C++ program are:

Simple C++ Program

```
// Hello World program ← comment  
  
#include <iostream.h> ← Allows access to an I/O library  
  
int main() ← Starts definition of special function main()  
{  
    cout << "Hello World\n"; ← output (print) a string  
  
    return 0; ← Program returns a status code (0 means OK)  
}
```

Structure of C++ Program

	1	<code>#include <iostream></code>	Header File
	2	<code>using namespace std;</code>	Standard Namespace
	3	<code>int main()</code>	Main Function
FUNCTION BODY	4	<code>{</code>	
	5	<code>int num1 = 24;</code> <code>int num2 = 34;</code>	Declaration of Variable
	6	<code>int result = num1 + num2;</code>	Expressions
	7	<code>cout << result << endl;</code>	Output
	8	<code>return 0;</code>	Return Statement
	9	<code>}</code>	

Classes and objects

- Class is a **user-defined datatype** that has its own **data members and member functions** whereas an **object is an instance of class** by which we can access the data members and member functions of the class.
- C++ is an object-oriented programming language.
- Everything in C++ is **associated with classes and objects**, along with its attributes and methods. For example: in real life, a car is an **object**. The car has **attributes**, such as weight and color, and **methods**, such as drive and brake.
- Attributes and methods are basically **variables and functions** that belongs to the class. These are often referred to as "**class members**".
- A class is a user-defined data type that we can use in our program, and it works as an object constructor, or a "**blueprint**" for **creating objects**.

Create a Class

To create the class, use the class keyword:

Example:

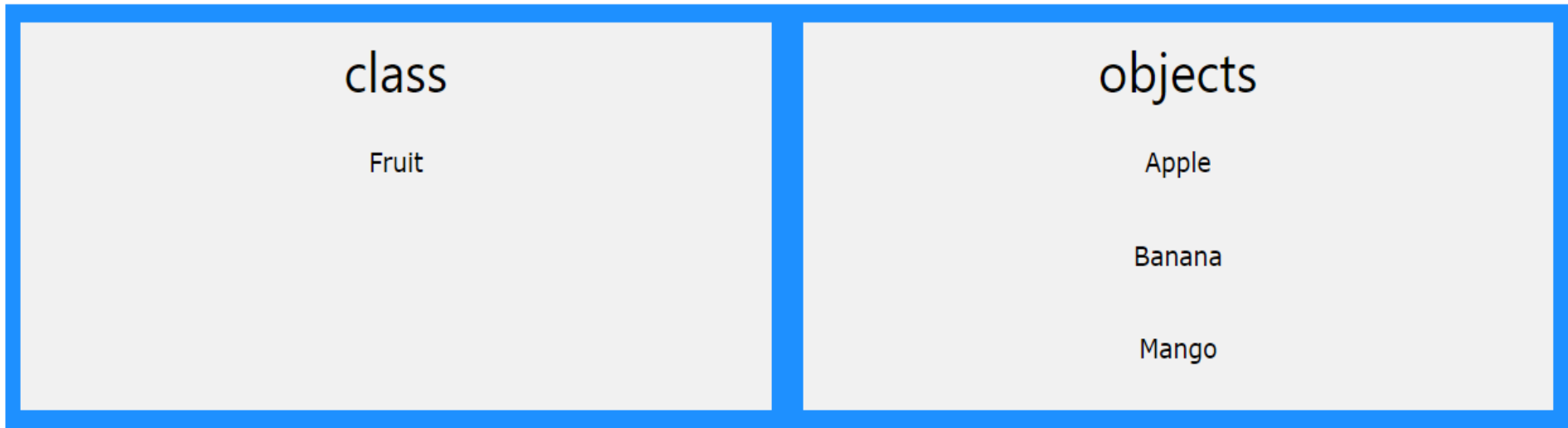
```
class MyClass {           // The class
    public:                // Access specifier
        int myNum;        // Attribute (int variable)
        string myString; // Attribute (string variable)
};
```

- The **class** keyword is used to create a class called **MyClass**.
- The **public** keyword is an **access specifier**, which specifies that members (attributes and methods) of the class are accessible from outside the class.
- Inside the class, there is an integer variable **myNum** and a string variable **myString**.
When variables are declared within a class, they are called **attributes**.
- At last, end the class definition with a semicolon ;

- **A class is a template or a blueprint** that **binds the properties and functions** of an entity. You can put all the **entities or objects having similar attributes** under a single roof, known as a class.
- Classes further implement the core concepts like encapsulation, data hiding, and abstraction.
- In C++, a **class acts as a data type** that can **have multiple objects or instances** of the class type.

Accessing Data Members and Member Functions

- The data members and member functions of the class can be accessed using the dot(‘.’) operator with the object.



Another example:



Syntax to Declare a Class in C++

```
class class_name
{
    // class definition
    access_specifier:    // public, protected, or private
    data_member1; // data members
    data_member2;
    func1(){}    // member functions
    func2(){}
};
```

class: This is the keyword used to declare a class that is followed by the name of the class.

class_name: This is the name of the class which is specified along with the keyword class.

access_specifier: It provides the access specifier before declaring the members of the class. These specifiers control the access of the class members within the class. The specifiers can be public, protected, and private.

data_member: These are the **variables of the class** to store the data values.

member_function: These are the **functions declared** inside the class.

Private and Public Keywords

You must have come across these two keywords. They are access modifiers.

Private:

When the **private keyword** is used to define a function or class, it becomes private. Such are only **accessible from within the class**.

Public:

The **public keyword**, on the other hand, makes data/functions public. These are **accessible from outside the class**.

Protected means data member and member functions can be **used in the same class and its derived class**. Members declared as protected can be accessed within the class and by derived classes.

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**.
- Objects in C++ are **analogous to real-world entities**. There are objects everywhere around you, like trees, birds, chairs, tables, dogs, cars, and the list can go on.
- There are some properties and functions associated with these objects. Similarly, C++ also includes the concept of objects.
- When you **define a class**, it contains **all the information about the objects** of the class type. Once it defines the class, it can **create similar objects sharing that information**, with the class name being the type specifier.

The syntax to create objects in C++:

```
class_name object_name;
```

The object `object_name` once created, can be used to access the data members and member functions of the class `class_name` using the dot operator in the following way:

```
obj.data_member = 10; // accessing data member
```

```
obj.func();          // accessing member function
```

Object Types

1. **Global object** refers to an object defined outside of all function bodies.
2. **Local object** refers to the object specified within the body of a function. A local object is limited to the block in which it is declared.
3. The **static type** is the type of an object variable as declared at **compile time**
4. **Dynamic type** is the type of an object variable determined at **run-time**.

Significance of Class and Object in C++

- **Data hiding:** A class prevents the access of the data from the outside world using access specifiers. It can set permissions to restrict the access of the data.
- **Code Reusability:** You can reduce code redundancy by using reusable code with the help of inheritance. Other classes can inherit similar functionalities and properties, which makes the code clean.
- **Data binding:** The data elements and their associated functionalities are bound under one hood, providing more security to the data.
- **Flexibility:** You can use a class in many forms using the concept of polymorphism. This makes a program flexible and increases its extensibility.

Member Functions in Classes

- A member function of a class is **a function that has its definition or its prototype** within the class definition **like any other variable**.
- It **operates on any object** of the class of which it is a member, and has **access to all the members of a class for that object**. The **member functions** are like the **conventional functions**.
- It **defines these methods inside a class** and has **direct access to all the data members** of its class. When you define a member function, it only **creates and shares one instance of that function** by all the instances of that class.
- The following syntax can be used to declare a **member function inside a class**:

```
class class_name
{
access_specifier:
    return_type member_function_name(data_type arg);
};
```

It specifies the **access specifier before declaring a member function**. It also specifies the **return type and data type** in the same way in which it declares a usual function.

Characteristics of member function

- Different classes have same function name. the “membership label” will resolve their scope.
- Member functions can access the private data of the class .a non member function cannot do this.(friend function can do this.)
- A member function can call another member function directly, without using the dot operator.

Method Definition Outside and Inside of a Class

The following **two ways can define a method** or member functions of a class:

1. Inside class definition
 2. Outside class definition
- The function body remains the same in both approaches to define a member function. The difference lies only in the function's header.

Inside class definition

This approach of defining a member function is generally **preferred for small functions**. It defines a **member function inside a class in the same familiar way** as it defines a conventional function. It specifies the return type of the function, followed by the function name, and it **provides arguments in the function header**. Then it provides the **function body to define the complete function**. The member functions that are defined **inside a class are automatically inline**.

EXAMPLE:

```
#include <iostream>
using namespace std;
// define a class
class my_class
{
public:
    // inside class definition of the function
    void sum(int num1, int num2)          // function header
    {   cout << "The sum of the numbers is: "; // function body
        cout << (num1 + num2) << "\n\n";    } };
int main()
{   // create an object of the class
    my_class obj;
    // call the member function
    obj.sum(5, 10);
    return 0;
}
```

Outside Class Definition

- Use the **scope resolution operator (::)** to define a member function outside its class.
- Even though you define the member function outside the class, it still needs to be **declared first inside the class**. This approach of **defining a member function of a class is the most preferred**.
- The following syntax is used to define a **member function outside its class**:

```
void class_name :: function_name(arguments)
{
// function body
}
```

EXAMPLE:

```
#include <iostream>
```

```
using namespace std;
```

```
// define a class
```

```
class my_class
```

```
{ public:
```

```
    // declare the member function
```

```
    void sum(int num1, int num2); };
```

```
// outside class definition of the function
```

```
void my_class::sum(int num1, int num2) // function header
```

```
{    cout << "The sum of the numbers is: "; // function body
```

```
    cout << (num1 + num2) << "\n\n"; }
```

```
int main()
```

```
{ // create an object of the class
```

```
    my_class obj;
```

```
    // call the member function
```

```
    obj.sum(5, 10);
```

```
    return 0; }
```

Example of Class and Object in C++:

// C++ program to illustrate how create a simple class and object

```
#include <iostream>
```

```
#include <string>
```

```
using namespace std;
```

```
// Define a class named 'Person'
```

```
class Person {
```

```
public:
```

```
    // Data members
```

```
    string name;
```

```
    int age;
```

```
    // Member function to introduce the person
```

```
    void introduce()
```

```
{
```

```
    cout << "Hi, my name is " << name << " and I am "
```

```
        << age << " years old." << endl;
```

```
}
```

```
};
```

```
int main()
{
    // Create an object of the Person class
    Person person1;
    // accessing data members
    person1.name = "Alice";
    person1.age = 30;
    // Call the introduce member method
    person1.introduce();
    return 0;
}
```

OUTPUT:

Hi, my name is Alice and I am 30 years old.

Structures and classes

- A structure is a **collection of variables of different data types** with the same name. A class in C++ is a **single structure that contains a collection of related variables and functions**. The **struct keyword** can be used to declare a structure. The **keyword class** can be used to declare a class.
- A structure is a **grouping** of variables of various **data types** referenced by the same name. A structure declaration serves as a template for creating an instance of the structure.

The syntax of the structure is as follows:

```
Struct Structure_name
```

```
{
```

```
Struct_member1;
```

```
Struct_member2;
```

```
Struct_member3;
```

```
... ..
```

```
Struct_memberN;
```

```
};
```

The **"struct"** keyword indicates to the compiler that a structure has been declared. The **"structure_name"** defines the name of the structure. Since the structure declaration is treated as a statement, so it is often ended by a semicolon.

Example program for Structure:

```
#include <iostream>
using namespace std;
struct Test {
    // x is public
    int x;
};
int main()
{
    Test t;
    t.x = 20;

    // works fine because x is public
    cout << t.x;
}
```

OUTPUT:

20

What is Class in C++?

A class in C++ is similar to a C structure in that it consists of a list of **data members** and a set of operations performed on the class. In other words, a class is the **building block** of Object-Oriented programming. It is a **user-defined object type** with its own set of **data members and member functions** that can be **accessed and used by creating a class instance**. A C++ class is similar to an object's blueprint.

Syntax:

- The structure and the class are syntactically similar. The syntax of class in C++ is as follows:

```
class class_name
{
// private data members and member functions.
Access specifier;
Data member;
Member functions (member list){ . . }
};
```

- The **class** is a **keyword** to indicate the compiler that a class has been declared. OOP's main function is data hiding, which is achieved by having three access specifiers: "**public**", "**private**", and "**safe**". If **no access specifier is specified** in the class when declaring **data members or member functions**, they are all considered **private by default**.
- The **public access specifier** allows others to access program functions or data. A member of that class may reach only the class's private members.
- During **inheritance**, the **safe access specifier is used**. If the access specifier is declared, it **cannot be changed again** in the program.

Main differences between the structure and class

- By default, all the members of the **structure are public**. In contrast, all members of the **class are private**.
- The structure will **automatically initialize** its members. In contrast, **constructors and destructors are used** to initialize the class members.
- When a structure is implemented, **memory allocates on a stack**. In contrast, memory is **allocated on the heap in class**.
- Variables in a structure cannot be initialized during the declaration, but they can be done in a class.
- There can be **no null values** in any structure member. On the other hand, the class variables may **have null values**.
- A structure is a **value type**, while a class is a **reference type**.
- Operators to work on the new data form can be described using a special method.

Features	Structure	Class
Definition	A structure is a grouping of variables of various data types referenced by the same name.	In C++, a class is defined as a collection of related variables and functions contained within a single structure.
Basic	If no access specifier is specified, all members are set to 'public'.	If no access specifier is defined, all members are set to 'private'.
Declaration	<pre> struct structure_name{ type struct_member 1; type struct_member 2; type struct_member 3; . type struct_memberN; }; </pre>	<pre> class class_name{ data member; member function; }; </pre>
Instance	Structure instance is called the 'structure variable'.	A class instance is called 'object'.
Inheritance	It does not support inheritance.	It supports inheritance.

Memory Allocated	Memory is allocated on the stack.	Memory is allocated on the heap.
Nature	Value Type	Reference Type
Purpose	Grouping of data	Data abstraction and further inheritance.
Usage	It is used for smaller amounts of data.	It is used for a huge amount of data.
Null values	Not possible	It may have null values.
Requires constructor and destructor	It may have only parameterized constructor.	It may have all the types of constructors and destructors.

THE SIMILARITIES BETWEEN STRUCT AND CLASS

- You can declare member variable in both.
- You can create member functions in both.
- You can create objects from both class and structure.
- You can create constructor and destructors in both class and structure.
- Inheritance is possible in both class and structure.
- You can inherit a structure from a class and do the reverse.
- You can use access specifiers In both class and structure.

unions and classes

- In C++, a **union is a user-defined datatype** in which we can **define members of different types of data types** just like structures.
- But one thing that makes it **different from structures** is that the **member variables in a union share the same memory location**, unlike a structure that allocates memory separately for each member variable.
- The **size of the union is equal to the size of the largest data type**.
- **Memory space** can be used by **one member variable** at one point in time, which means if we **assign value to one member variable**, it will **automatically deallocate the other member variable** stored in the memory which will lead to loss of data.
- A **union is a user-defined type** in which **all members share the same memory location**. This definition means that at any given time, a union can **contain no more than one object from its list of members**.
- It also means that no matter how many members a union has, it always uses only **enough memory to store the largest member**.

Need of Union in C++

- When the **available memory is limited**, it can be used to achieve memory efficiency.
- It is used to **encapsulate different types of data members**.
- It helps in **optimizing the performance** of applications.

Syntax of Union in C++

Union is defined using the 'union' keyword.

```
union Union_Name {
```

```
// Declaration of data members
```

```
}; union_variables;
```

After defining the union we can also **create an instance** of it the same as we **create an object of the class** or declare any other data type.

```
int main()
```

```
{
```

```
union Union_Name var1;
```

```
Or
```

```
Union_Name var1
```

```
}
```

Example program for union:

```
union geek {  
    // Defining data members  
    int age;  
    char grade;  
    float GPA;  
};  
int main()  
{ // Defining a union variable  
    union geek student1;  
    // Assigning values to data member of union geek and  
    // printing the values of data members  
    student1.age = 25;  
    cout << "Age : " << student1.age << endl;  
    student1.grade = 'B';  
    cout << "Grade : " << student1.grade << endl;  
    student1.GPA = 4.5;  
    cout << "GPA : " << student1.GPA << endl;  
    return 0;  
}
```

OUTPUT: Age : 25

Grade : B

GPA : 4.5

Class

- **Class:** It is a **user-defined datatype** enclosed with variables and functions. It is like a **blueprint for an object**. Class members are **private** by default.
- For Example, the car is an object, its color, design, weight are its attributes whereas the brake, speed limit, etc. are its functions.

Syntax:

```
class <class_name> {  
private: // Data members  
.....  
public: // Data members // Member function  
.....  
protected: // Data members // Member function  
..... };
```

Structure, union and class

	Class	structure	union
Default member visibility	Private	Public	Public
Default inheritance mode	Private	Public	No Inheritance
Constructor	Yes	Yes	Yes
Destructor	Yes	Yes	Yes
Member function	Yes	Yes	Yes
Function overloading	Yes	Yes	Yes
Friend function	Yes	Yes	Yes
Operator overloading	Yes	Yes	Yes
Inheritance	Yes	Yes	No
Polymorphism	Yes	Yes	No
Function overriding	Yes	Yes	No
Usage constraint	No Constraint	No Constraint	Only one member at a time

friend functions

- A friend function is a **function that isn't a member of a class** but has **access to the class's private and protected members**. Friend functions aren't considered class members; they're **normal external functions that are given special access privileges**.
- A friend function can be **granted special access to private and protected members** of a class in C++. They are **not the member functions** of the class but **can access and manipulate the private and protected** members of that class for they are declared as friends.
- A friend function can be:
 - 1.A global function
 - 2.A member function of another class

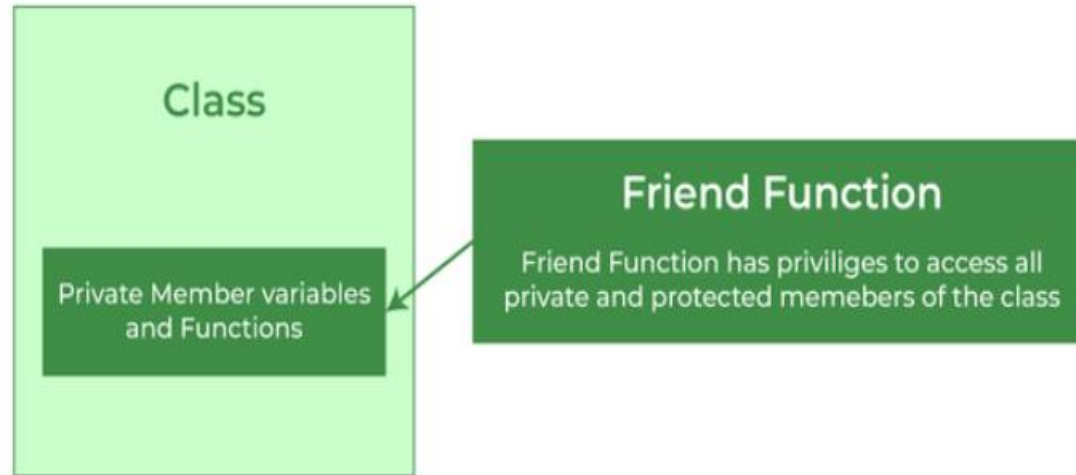
Syntax:

`friend return_type function_name (arguments); // for a global function`

or

`friend return_type class_name::function_name (arguments); // for a member function of another class`

Friend Function



Features of Friend Functions

- A friend function is a **special function** in C++ that in spite of not being a member function of a class has the **privilege to access the private and protected data** of a class.
- A friend function is a **non-member function or ordinary function** of a class, which is declared as a friend using the keyword “**friend**” inside the class. By declaring a function as a friend, all the **access permissions are given to the function**.
- The keyword “**friend**” is **placed only in the function declaration** of the friend function and **not in the function definition or call**.
- A friend function is called like an ordinary function. It **cannot be called using the object name and dot operator**. However, it may accept the object as an argument whose value it wants to access.
- A friend function can be **declared in any section of the class** i.e. public or private or protected.

Advantages of Friend Functions

- A friend function is **able to access members without** the need of **inheriting** the class.
- The friend function acts as a **bridge between two classes** by accessing their private data.
- It can be used to **increase the versatility** of overloaded operators.
- It can be declared **either in the public or private or protected** part of the class.

Disadvantages of Friend Functions

- Friend functions have access to private members of a class from outside the class which **violates the law of data hiding**.
- Friend functions **cannot do any run-time polymorphism** in their members.

Let's see the simple example of C++ friend function used to print the length of a box.

```
#include <iostream>
using namespace std;
class Box
{
    private:
        int length;
    public:
        Box(): length(0) { }
        friend int printLength(Box); //friend function
};
int printLength(Box b)
{
    b.length += 10;
    return b.length;
}
int main()
{
    Box b;
    cout<<"Length of box: "<< printLength(b)<<endl;
    return 0;
}
```

Output:

Length of box: 10

simple example when the function is friendly to two classes.

```
#include <iostream>
using namespace std;
class B;    // forward declaration.
class A
{
    int x;
    public:
    void setdata(int i)
    {
        x=i;
    }
    friend void min(A,B);    // friend function.
};
```

```
class B
{
    int y;
    public:
    void setdata(int i)
    {
        y=i;
    }
    friend void min(A,B);           // friend function
};
void min(A a,B b)
{
    if(a.x<=b.y)
        std::cout << a.x << std::endl;
    else
        std::cout << b.y << std::endl;
}
```

```
int main()
{
    A a;
    B b;
    a.setdata(10);
    b.setdata(20);
    min(a,b);
    return 0;
}
```

Output:

10

Global Function as Friend Function

```
// C++ program to create a global function as a friend function of some class
#include <iostream>
using namespace std;
class base {
private:
    int private_variable;
protected:
    int protected_variable;
public:
    base()
    {
        private_variable = 10;
        protected_variable = 99;
    }
    // friend function declaration
    friend void friendFunction(base& obj);
};
```

```
// friend function definition
void friendFunction(base& obj)
{
    cout << "Private Variable: " << obj.private_variable << endl;
    cout << "Protected Variable: " << obj.protected_variable;
}
// driver code
int main()
{
    base object1;
    friendFunction(object1);
    return 0;
}
```

OUTPUT:

Private Variable: 10

Protected Variable: 99

Member Function of Another Class as Friend Function

```
// C++ program to create a member function of another class as a friend function
#include <iostream>
using namespace std;
class base; // forward definition needed another class in which function is declared
class anotherClass {
public:
    void memberFunction(base& obj);
};
// base class for which friend is declared
class base {
private:
    int private_variable;
protected:
    int protected_variable;
```

public:

```
    base()
    { private_variable = 10;
      protected_variable = 99; }
    // friend function declaration
    friend void anotherClass::memberFunction(base& obj);
};
// friend function definition
void anotherClass::memberFunction(base& obj)
{ cout << "Private Variable: " << obj.private_variable<< endl;
  cout << "Protected Variable: " << obj.protected_variable;
}
int main()
{ base object1;
  anotherClass object2;
  object2.memberFunction(object1);
  return 0;
}
```

Output:

```
Private Variable: 10
Protected Variable: 99
```

Friend class

- A friend class can **access both private and protected members** of the class in which it has been declared as friend.
- A **friend class** can access private and protected members of other classes in which it is declared as a friend. It is sometimes **useful to allow a particular class to access private and protected members** of other classes. For example, a LinkedList class may be allowed to access private members of Node.
- We can declare a friend class in C++ by using the **friend** keyword.

Syntax:

```
friend class class_name; // declared in the base class
```

Use of Friend Class in C++

Friend class has numerous uses and benefits. Some of the primary use cases include:

- Accessing private and protected members of other classes (as you would know by now)
- **Declaring all the functions of a class** as friend functions
- Allowing to **extend storage and access** its part while maintaining encapsulation
- Enabling classes to **share private members' information**
- Widely used where **two or more classes have interrelated** data members.
- Friend class in C++ helps to **access the data of private members**.
- The use of the friend class helps to **write readable code**.
- Friend class helps you in **testing and debugging the errors as data** of private class members are accessible.
- The Friend class is mostly used when operator overloading is involved in classes. To access private member data, operator overloading takes input and output, and the friend **class helps in the easy access of private members**.
- Friend classes are also used when you need to keep your **interface of a class clean**.

example of a friend class:

```
#include <iostream>
```

```
using namespace std;
```

```
class A
```

```
{
```

```
    int x =5;
```

```
    friend class B;    // friend class.
```

```
};
```

```
class B
```

```
{
```

```
    public:
```

```
    void display(A &a) //specifies that their corresponding arguments are to be passed by  
                        reference instead of by value.
```

```
{
```

```
    cout<<"value of x is : "<<a.x;
```

```
} };
```

```
int main()
{
    A a;
    B b;
    b.display(a);
    return 0;
}
```

Output:

value of x is : 5

Difference between a Friend Class and a Friend Function

Feature	Friend Function	Friend Class
Access Control	Grants access to specific functions	Grants access to all members of a class
Declaration Syntax	Declared using friend keyword inside class	Declared using friend keyword before class
Granularity of Access	Can specify specific functions to be friends	Grants access to all members of the class
Usage	Suitable for granting access to individual functions	Suitable for granting access to entire class

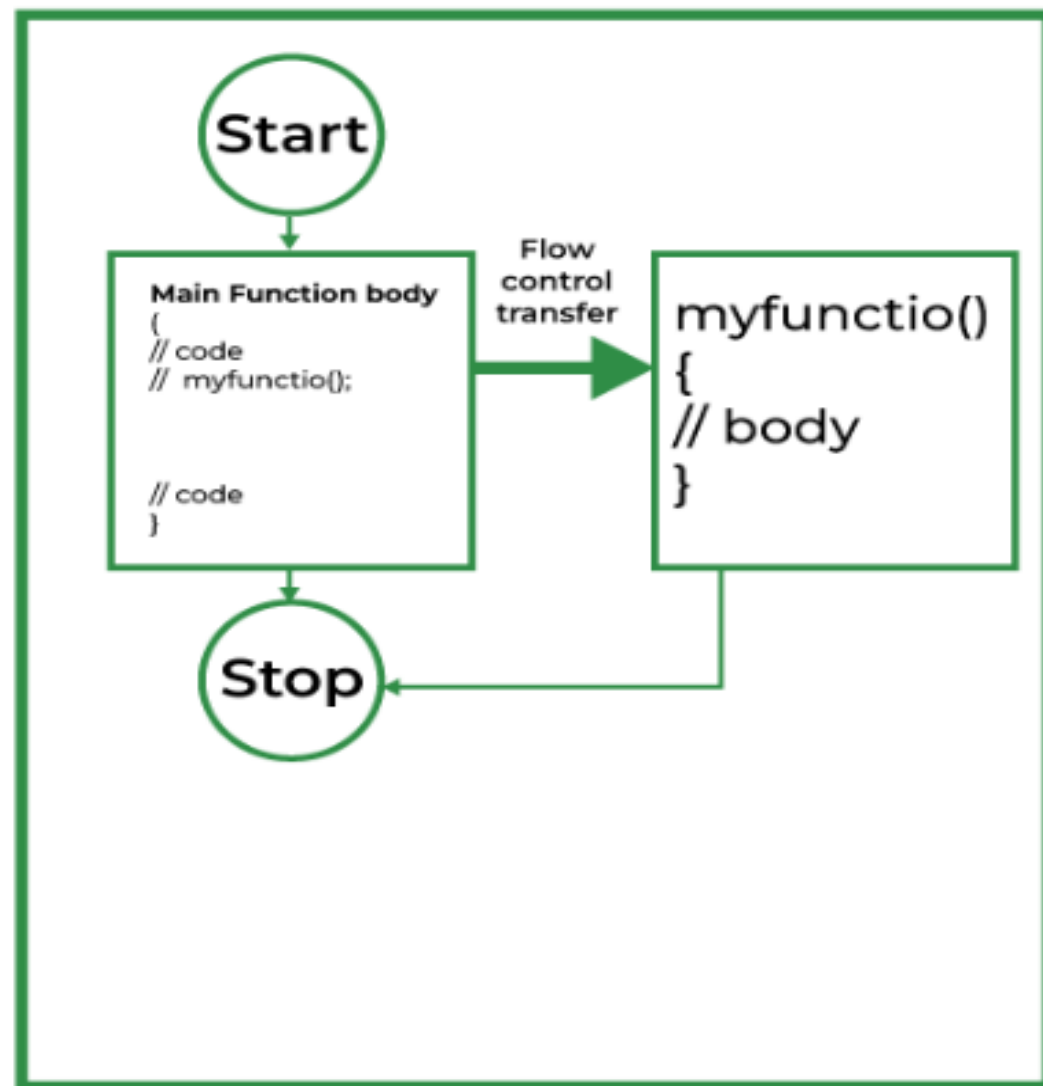
Inline Functions

- C++ provides inline functions to **reduce the function call overhead**. An inline function is a **function that is expanded in line when it is called**.
- When the **inline function is called** whole code of the inline function **gets inserted or substituted at the point of the inline** function call.
- This substitution is **performed by the C++ compiler at compile time**. An inline function may **increase efficiency if it is small**.

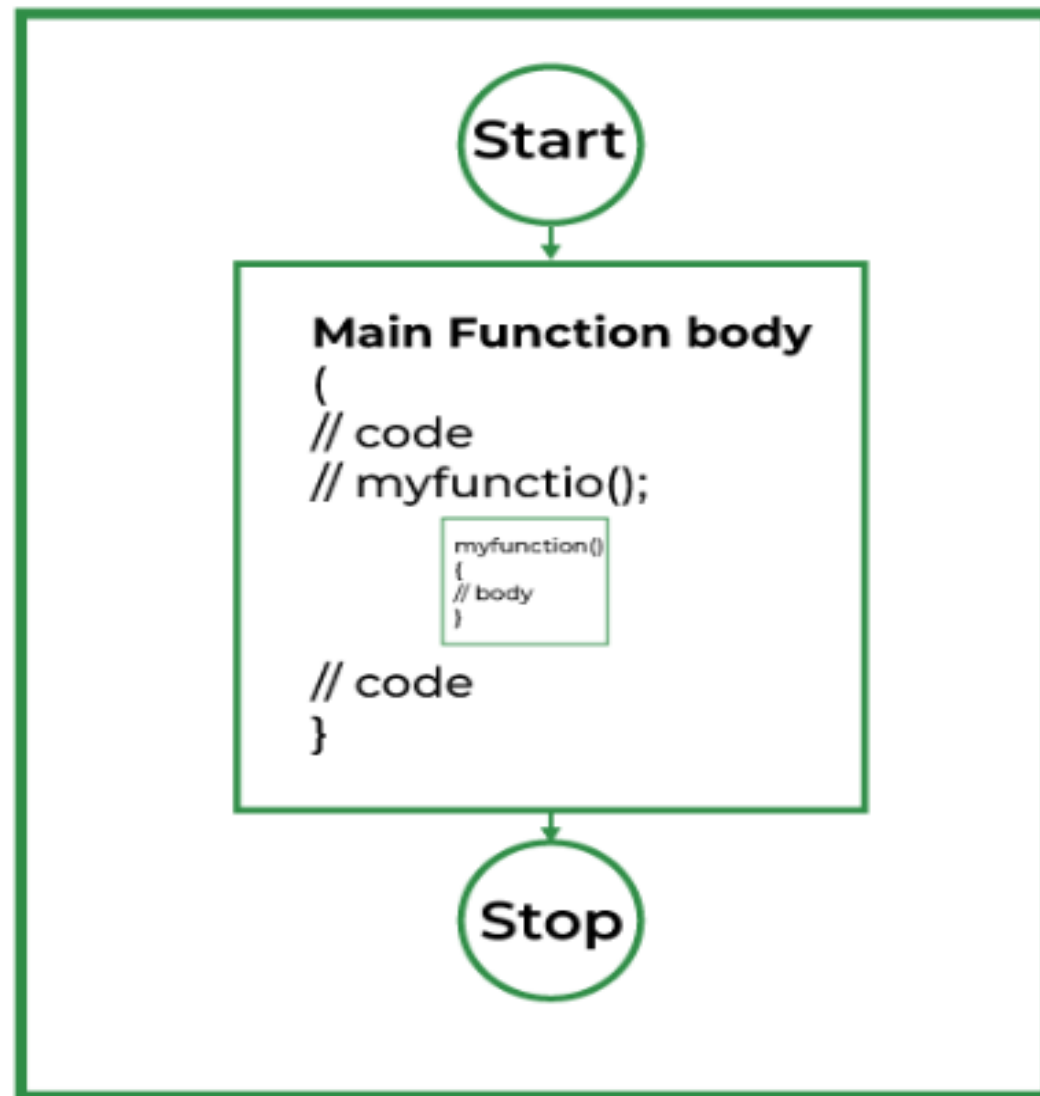
Syntax:

```
inline return-type function-name(parameters)
{
// function code
}
```


Normal Function



Inline Function



Inline functions Advantages

- **Function call overhead doesn't occur.**
- It also **saves the overhead of push/pop variables on the stack** when a function is called.
- It also **saves the overhead of a return call** from a function.
- When you inline a function, you may **enable the compiler to perform context-specific optimization** on the body of the function. Such **optimizations are not possible for normal function calls**. Other optimizations can be obtained by considering the **flows of the calling context and the called context**.
- An inline function may be useful (if it is small) for **embedded systems** because inline can yield **less code than the function called** preamble and return.

Inline function Disadvantages

- The **added variables** from the inlined function **consume additional registers**, After the in-lining function if the variable number which is going to **use the register increases then they may create** overhead on register variable resource utilization. This means that when the inline function body is substituted at the point of the function call, the **total number of variables used by the function also gets inserted**. So the number of registers going to be used for the variables will also get increased. So if after **function inlining variable numbers increase drastically** then it would surely cause overhead on register utilization.
- If you **use too many inline functions** then the **size of the binary executable file will be large**, because of the **duplication of the same code**.
- Too much inlining can also **reduce your instruction cache hit rate**, thus **reducing the speed of instruction** fetch from that of **cache memory to that of primary memory**.
- The inline function may **increase compile time overhead** if someone changes the code inside the inline function then all the calling location has to be **recompiled because the compiler would be required to replace all the code once again** to reflect the changes, otherwise it will continue with old functionality.
- Inline functions may **not be useful for many embedded systems**. Because in embedded systems **code size is more important than speed**.
- Inline functions might cause **thrashing because inlining might increase the size of the binary** executable file. **Thrashing in memory causes the performance** of the computer to **degrade**. The following program demonstrates the use of the inline function.

EXAMPLE

```
#include <iostream>
using namespace std;
inline int add(int a, int b)
{
    return(a+b);
}
int main()
{ cout<<"Addition of 'a' and 'b' is:"<<add(2,3);
  return 0;
}
```

OUTPUT:

Addition of 'a' and 'b' is:5

Example 2

```
// define an inline function that prints the sum of 2 integers
inline void printSum(int num1,int num2) {
    cout << num1 + num2 << "\n";
}
int main() {
    // call the inline function
    // first call
    printSum(10, 20);
    // second call
    printSum(2, 5);
    // third call
    printSum(100, 400);
    return 0;
}
```

Example:

```
#include<iostream>
using namespace std;
// define an inline function using the keyword "inline"
inline int setNum()
{
    return 10; // definition of inline function
}
int main()
{
    int num ;
    // call the inline function
    num = setNum();
    // setNum() will be replaced by definition of inline function
    cout << " The inline function returned: " << num ;
    cout << "\n\n";
    return 0 ;
}
```

OUTPUT:

The inline function returned: 10

static class members

Static data members are **class members that are declared** using **static keywords**.

A static member has **certain special characteristics** which are as follows:

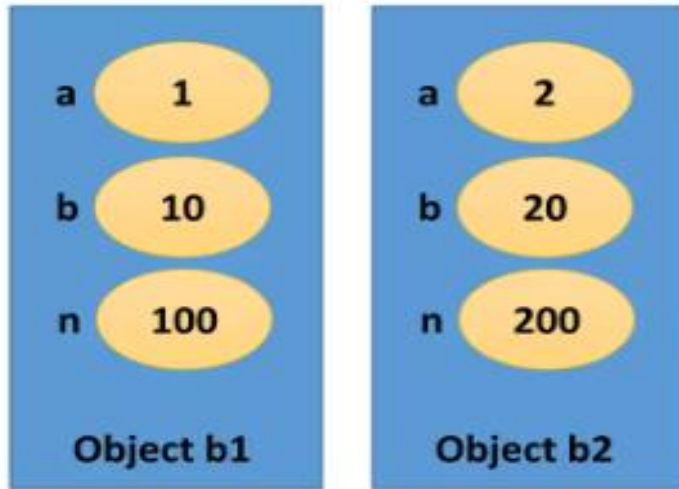
- Only **one copy of that member is created** for the **entire class and is shared** by all the objects of that class, no matter how many objects are created.
- It is **initialized before any object** of this class is created, even before the main starts outside the class itself.
- It is **visible can be controlled** with the **class access specifiers**.
- Its lifetime is the entire program.

Syntax

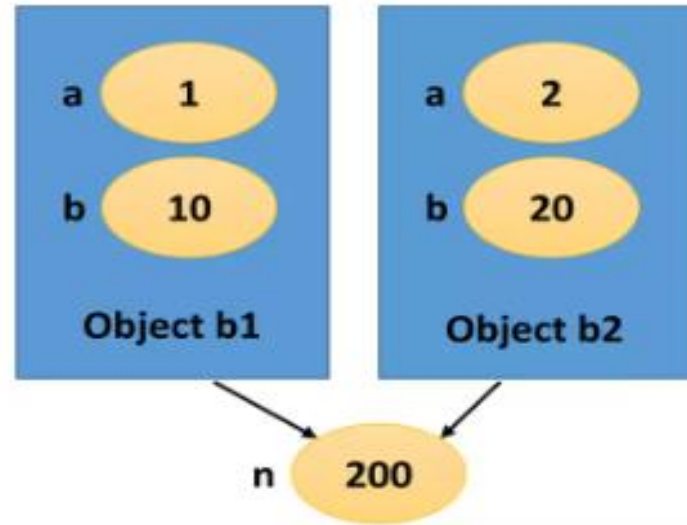
```
className {  
static data_type data_member_name;  
..... }
```

Difference Between Normal & Static Data Members

Object with three normal data members



Object with two normal data members and one static data member



EXAMPLE:

```
#include <iostream>
using namespace std;
// class definition
class A {
public:
    // static data member here
    static int x;
    A() { cout << "A's constructor called " << endl; }
};
// we cannot initialize the static data member inside the class due to class rules and the fact that we cannot assign it a value using
constructor
int A::x = 2;
int main()
{
    // accessing the static data member using scope resolution operator
    cout << "Accessing static data member: " << A::x << endl;
    return 0;
}
```

OUTPUT:

Accessing static data member: 2

Defining Static Data Member

- The **static members** are only **declared in the class declaration**. If we try to **access the static data member without an explicit** definition, the compiler will give an error.
- To **access the static data member** of any class we have to define it first and **static data members are defined outside the class definition**.
- The only exception to this are **static const data members of integral type** which can be **initialized in the class declaration**.

Syntax

```
datatype class_name::var_name = value...;
```

Example:

```
int A::x = 10
```

Accessing a Static Member

- We can access the **static data member** without creating the instance of the **class**. Just remember that we **need to initialize** it beforehand. There are **2 ways** of accessing static data members:

1. Accessing static data member using Class Name and Scope Resolution Operator

- The class name and the scope resolution operator can be used to access the static data member even when there are **no instances/objects of the class** present in the scope.

Syntax:

Class_Name :: var_name

Example:

A::x

2. Accessing static data member through Objects

- We can also access the static data member **using the objects of the class** using dot operator.

Syntax

object_name . var_name

Example

obj.x

C++ Program to demonstrate the working of static data member(Verify the Properties of the Static Data Members):

```
#include <iostream>
using namespace std;
// creating a dummy class to define the static data member it will inform when its type of the object will be
created
class stMember {
public:
    int val;
    // constructor to inform when the instance is created
    stMember(int v = 10): val(v) {
        cout << "Static Object Created" << endl;
    } };
// creating a demo class with static data member of type stMember
class A {
public:
    // static data member
    static stMember s;
    A() { cout << "A's Constructor Called " << endl; } };
stMember A::s = stMember(11);
```

```
int main()
{
    // Statement 1: accessing static member without creating the object
    cout << "accessing static member without creating the object: ";
    // this verifies the independency of the static data member from the instances
    cout << A::s.val << endl;
    cout << endl;
    // Statement 2: Creating a single object to verify if the separate instance will be created for each object
    cout << "Creating object now: ";
    A obj1;
    cout << endl;
    // Statement 3: Creating multiple objects to verify that each object will refer the same static member
    cout << "Creating object now: ";
    A obj2;
    cout << "Printing values from each object and classname" << endl;
    cout << "obj1.s.val: " << obj1.s.val << endl;
    cout << "obj2.s.val: " << obj2.s.val << endl;
    cout << "A::s.val: " << A::s.val << endl;
    return 0;
}
```

Output:

Static Object Created

accessing static member without creating the object: 11

Creating object now: A's Constructor Called

Creating object now: A's Constructor Called

Printing values from each object and classname

obj1.s.val: 11

obj2.s.val: 11

A::s.val: 11

scope resolution operator

- The scope resolution operator is used to reference the global variable or member function that is out of scope.
- Therefore, we use the scope resolution operator to access the hidden variable or function of a program. The operator is represented as the double colon (::) symbol.
- For example, when the global and local variable or function has the same name in a program, and when we call the variable, by default it only accesses the inner or local variable without calling the global variable. In this way, it hides the global variable or function.
- To overcome this situation, we use the scope resolution operator to fetch a program's hidden variable or function.

Uses of the scope resolution Operator

- 1.It is used to access the hidden variables or member functions of a program.
- 2.It defines the member function outside of the class using the scope resolution.
- 3.It is used to access the static variable and static function of a class.
- 4.The scope resolution operator is used to override function in the Inheritance.

Program to access the hidden value using the scope resolution (::) operator

```
#include <iostream>
using namespace std;
// declare global variable
int num = 50;
int main ()
{
// declare local variable
int num = 100;
// print the value of the variables
cout << " The value of the local variable num: " << num;
// use scope resolution operator (::) to access the global variable
cout << "\n The value of the global variable num: " << ::num;
return 0;
}
```

OUTPUT:

The value of the local variable num: 100

The value of the global variable num: 50

Program to define the member function outside of the class using the scope resolution (::) operator

```
#include <iostream>
using namespace std;
class Operate
{ public:
    // declaration of the member function
    void fun();
};
// define the member function outside the class.
void Operate::fun() /* return_type Class_Name::function_name */
{
    cout << " It is the member function of the class. ";
}
int main ()
{ // create an object of the class Operate
  Operate op;
  op.fun();
  return 0;
}
```

Program to access the static variable using the scope resolution (::) operator

```
#include <iostream>
```

```
using namespace std;
```

```
class Parent
```

```
{
```

```
static int n1;
```

```
public:
```

```
static int n2;
```

```
// The class member can be accessed using the scope resolution operator.
```

```
void fun1 ( int n1)
```

```
{
```

```
// n1 is accessed by the scope resolution operator (:: )
```

```
cout << " The value of the static integer n1: " << Parent::n1;
```

```
cout << " \n The value of the local variable n1: " << n1;
```

```
}
```

```
};
```

```
// define a static member explicitly using :: operator
int Parent::n1 = 5; // declare the value of the variable n1
int Parent::n2 = 10;
int main ()
{
Parent b;
int n1 = 15;
b.fun1 (n1);
cout << " \n The value of the Base::n2 = " << Parent::n2;
return 0;
}
```

Output:

The value of the static integer n1: 5

The value of the local variable n1: 15

The value of the Base::n2 = 10

Program to demonstrate the standard namespace using the scope resolution (::) operator

```
#include <iostream>
int main ()
{
int num = 0;

// use scope resolution operator with std namespace
std :: cout << " Enter the value of num: ";
std::cin >> num;
std:: cout << " The value of num is: " << num;
}
```

OUTPUT:

Enter the value of num: 50

The value of num is: 50

Program to override the member function using the scope resolution (::) operator

```
#include <iostream>
using namespace std;
class ABC
{
// declare access specifier
public:
void test ()
{
cout << " \n It is the test() function of the ABC class. ";
}
};
// derive the functionality or member function of the base class
class child : public ABC
{
public:
```

```
void test()
{
ABC::test();
cout << " \n It is the test() function of the child class. ";
}
};
int main ()
{
// create object of the derived class
child ch;
ch.test();
return 0;
}
```

OUTPUT:

It is the test() function of the ABC class.
It is the test() function of the child class

Nested classes

- A nested class is a **class which is declared in another enclosing class**. A nested class is a **member** and as such has the **same access rights as any other member**.
- The **members of an enclosing class** have **no special access to members** of a nested class; the **usual access rules shall be obeyed**.
- 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**.
- Nested class in C++ can be accessed **outside the enclosing class using scope resolution operator (::)**.

Syntax

```
Class Parent_Class
```

```
{ //Here, data and functions implement for the parent class.
```

```
class Nested_Class
```

```
{           // we will set data and functions for the nested class.
```

```
}; };
```

Example:nested classes access private members of the parent class.

```
#include <iostream>
using namespace std;
class ParentClass {
private:
    int data = 5;
public:
    class NestedClass {
public:
        void printPrivateMember(const ParentClass& parent) {
            cout << parent.data ;
        } }; };
int main() {
    ParentClass obj;
    ParentClass::NestedClass nestedObj;
    nestedObj.printPrivateMember(obj);
    return 0;
}
```

OUTPUT: 5

Example to understand the concept of defining nested classes outside the scope of enclosing classes.

```
#include <iostream>
using namespace std;
class Enclosing
{ public:
  class Nested
  {
  public:
  void print();
  };};
//Member function defined outside the scope of class
void Enclosing::Nested::print()
{ cout << "Nested class member outside " << endl;
}
int main()
{
  Enclosing::Nested en;
  en.print(); //calling member function of nested class
  return 0;
}
```

OUTPUT: Nested class member outside

local classes

- A **class declared inside a function** becomes **local to that function** and is called Local Class in C++. A **class declared inside a function** is known as a local class in C++ as it is local to that function.
- A local class name can **only be used locally** i.e., inside the function and **not outside it**.
- The **methods of a local class** must be **defined inside** it only.
- A local class can have **static functions** but, **not static data members**.

1) A local class type name can only be used in the enclosing function:Example

```
#include<iostream>
using namespace std;
void func() {
class LocalClass
{          // Body
};
}
int main() {
    return 0; }
```

2) All the methods of Local classes must be defined inside the class only.

```
#include <iostream>
using namespace std;
void fun()
{
class Test // local to fun
{
public:
// Fine as the method is defined inside the local class
void method()
{
cout << "Local Class method() called";
} };
Test t;
t.method(); }
int main()
{
fun();
return 0; }
```

OUTPUT:
Local Class method() called

3) C++ program to demonstrate that a Local class cannot contain static data members

```
#include <iostream>
using namespace std;
void fun()
{
class Test // local to fun
{
public:
static void method()
{
cout << "Local Class method() called";
} };
Test::method();
}
int main()
{
    fun();
    return 0;
}
```

OUTPUT:

Local Class method() called

Example:

```
#include<iostream>
using namespace std;
void func() {
    class LocalClass {
        private:
            int num;
        public:
            void getdata( int n) {
                num = n;  }
            void putdata() {
                cout<<"The number is "<<num;
                }  };
    LocalClass obj;
    obj.getdata(7);
    obj.putdata();
}
int main() {
    cout<<"Demonstration of a local class"<<endl;
    func();
    return 0;
}
```

OUTPUT:

Demonstration of a local class
The number is 7

passing objects to functions

To **pass an object as an argument** we write the **object name as the argument** while calling the function the same way we do it for other variables.

Syntax:

```
function_name(object_name);
```

Example:

```
// C++ program to show passing of objects to a function
```

```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```
class Example {
```

```
public:
```

```
    int a;
```

```
    // This function will take an object as an argument
```

```
    void add(Example E)
```

```
    {
```

```
        a = a + E.a;
```

```
    }
```

```
};
```



```
// Driver Code
```

```
int main()
```

```
{
```

```
    // Create objects
```

```
    Example E1, E2;
```

```
    // Values are initialized for both objects
```

```
    E1.a = 50;
```

```
    E2.a = 100;
```

```
    cout << "Initial Values \n";
```

```
    cout << "Value of object 1: " << E1.a << "\n& object 2: " << E2.a << "\n\n";
```

```
    // Passing object as an argument to function add()
```

```
    E2.add(E1);
```

```
    // Changed values after passing object as argument
```

```
    cout << "New values \n";
```

```
    cout << "Value of object 1: " << E1.a << "\n& object 2: " << E2.a << "\n\n";
```

```
    return 0;
```

```
}
```

OUTPUT:

Initial Values Value of object 1: 50

& object 2: 100

New values

Value of object 1: 50

& object 2: 150

Returning objects

- **An object is an instance of a class.** Memory is **only allocated when an object is created and not when a class is defined.** An object can be **returned by a function using the return keyword.**
- Pass class's objects as arguments and also return them from a function the same way **we pass and return other variables.** No special keyword or header file is required to do so.

Returning Object as argument syntax:

```
object = return object_name;
```

Example 1: C++ program to show passing of objects to a function

```
#include <bits/stdc++.h>
using namespace std;
class Example {
public: int a;
// This function will take object as arguments and return object
Example add(Example Ea, Example Eb)
{
Example Ec;
Ec.a = Ea.a + Eb.a;
// returning the object
return Ec;
} };
int main()
{
Example E1, E2, E3;
// Values are initialized for both objects
E1.a = 50;
E2.a = 100;
E3.a = 0;
```

```
cout << "Initial Values \n";
cout << "Value of object 1: " << E1.a << ", \nobject 2: " << E2.a << ", \nobject 3: " << E3.a << "\n";
// Passing object as an argument to function add()
E3 = E3.add(E1, E2);
// Changed values after passing object as an argument
cout << "New values \n";
cout << "Value of object 1: " << E1.a << ", \nobject 2: " << E2.a << ", \nobject 3: " << E3.a << "\n";
return 0;
}
```

Output:

Initial Values

Value of object 1: 50,

object 2: 100,

object 3: 0

New values

Value of object 1: 50,

object 2: 100,

object 3: 150

Example 2:

```
#include <iostream>
using namespace std;
class Point {
private:
int x;
int y;
public:
Point(int x1 = 0, int y1 = 0) {
    x = x1;
    y = y1;    }
Point addPoint(Point p) {
    Point temp;
    temp.x = x + p.x;
    temp.y = y + p.y;
    return temp;    }
void display() {
    cout<<"x = "<< x <<"\n";
    cout<<"y = "<< y <<"\n";
}};
```

```
int main() {  
    Point p1(5,3);  
    Point p2(12,6);  
    Point p3;  
    cout<<"Point 1\n";  
    p1.display();  
    cout<<"Point 2\n";  
    p2.display();  
    p3 = p1.addPoint(p2);  
    cout<<"The sum of the two points is:\n";  
    p3.display();  
    return 0;  
}
```

OUTPUT:

```
Point 1  
x = 5  
y = 3  
Point 2  
x = 12  
y = 6  
The sum of the two points is:  
x = 17  
y = 9
```

Object assignment

- **Assign one object to another** using the **assignment operator** (=) if the objects have the **same data type**.
- If both objects are of the same type (that is, both are objects of the same class), then one **object may be assigned to another**.
- By default, when **one object is assigned to another**, a bitwise copy of the first object's data is copied to the second.
- By default, **all data from one object is assigned to the other** using a bit-by-bit copy.
- An **exact duplicate** is created.
- It is possible to overload the assignment operator so that customized assignment operations can be defined.
- Assignment of one object to another simply makes the data in those objects identical.
- The two objects are still completely separate.
- Thus, a subsequent modification of one object's data has no effect on that of the other.

Program demonstrates object assignment:

```
// Demonstrate object assignment.
#include <iostream>
using namespace std;
class myclass {
int a, b;
public:
void setab(int i, int j) { a = i, b = j; }
void showab();
};
void myclass::showab(){
cout << "a is " << a << "\n";
cout << "b is " << b << "\n";
}
```

```
int main()
{
myclass ob1, ob2;
ob1.setab(10, 20);
ob2.setab(0, 0);
cout << "ob1 before assignment: \n";
ob1.showab();
cout << "ob2 before assignment: \n";
ob2.showab();
cout << "\n";
ob2 = ob1; // assign ob1 to ob2
cout << "ob1 after assignment: \n";
ob1.showab();
cout << "ob2 after assignment: \n";
ob2.showab();
return 0;
}
```

OUTPUT:

ob1 before assignment:

a is 10

b is 20

ob2 before assignment:

a is 0

b is 0

ob1 after assignment:

a is 10

b is 20

ob2 after assignment:

a is 10

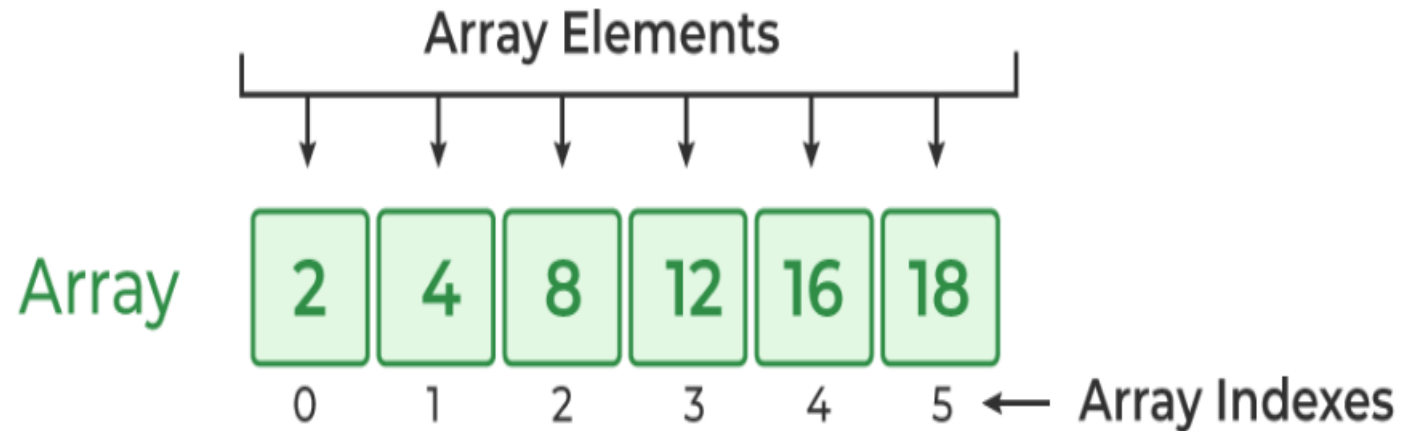
b is 20

Array

- An array is a **data structure** that is **used to store multiple values of similar data types** in a contiguous memory location.
- C++ provides a data structure, the array, which **stores a fixed-size, sequential collection of elements of the same type**.
- An array is used to **store a collection of data**, but it is often more useful to think of an array as a **collection of variables of the same type**.
- Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables.
- A specific element in an array is **accessed by an index**.
- All arrays consist of **contiguous memory locations**.
- The compiler will reserve a **section of the memory to hold all of the data** that is defined for the array.
- The **lowest address** corresponds to the **first element** and the **highest address** to the **last element**.

For example, if we have to **store the marks of 4 or 5 students** then we can easily store them by creating 5 different variables but what if we want to store marks of 100 students or say 500 students then it becomes very challenging to create that numbers of variable and manage them. Now, arrays come into the picture that can do it easily by just creating an array of the required size.

Array in C++



Properties of Arrays in C++

- An Array is a **collection of data of the same data type**, stored at a contiguous memory location.
- Indexing of an array **starts from 0**. It means the **first element is stored at the 0th index**, the **second at 1st**, and so on.
- Elements of an array can be **accessed using their indices**.
- Once an array is declared its size remains constant throughout the program.
- An array can have **multiple dimensions**.
- The **size of the array in bytes** can be determined by the **sizeof operator** using which we can also find the number of elements in the array.
- We can find the **size of the type of elements stored in an array** by **subtracting adjacent addresses**.

Array Declaration

declare an array by simply specifying the data type first and then the name of an array with its size.

Syntax:

```
data_type array_name[Size_of_array];
```

Example:

```
int arr[5];
```

Initialization of Array

initialize an array in **many ways** but we will discuss **some most common ways to initialize an array**. We can **initialize an array at the time of declaration or after declaration**.

1. Initialize Array with Values

```
int arr[5] = { 1, 2, 3, 4, 5 };
```

2. Initialize Array with Values and without Size

```
int arr[] = { 1, 2, 3, 4, 5 };
```

3. Initialize Array after Declaration (Using Loops)

This method is generally used when we want to **take input from the user** or we cant to **assign elements one by one to each index of the array**. We can modify the **loop conditions or change the initialization values** according to requirements.

```
for (int i = 0; i < N; i++) {  
arr[i] = value;  
}
```

4. Initialize an array partially

```
int partialArray[5] = {1, 2};
```

5. Initialize the array with zero

```
int zero_array[5] = {0};
```


Size of an Array

calculate the size of an array using sizeof() operator trick. First, we find the size occupied by the whole array in the memory and then divide it by the size of the type of element stored in the array. This will give us the number of elements stored in the array.

`Data_type size = sizeof(Array_name) /sizeof(Array_name[index]);`

// C++ Program to Illustrate How to Find the Size of an Array

```
#include <iostream>
using namespace std;
int main(){
int arr[] = { 1, 2, 3, 4, 5 }; // Size of one element of an array
    cout << "Size of arr[0]: " << sizeof(arr[0]) << endl; // Size of array 'arr'
cout << "Size of arr: " << sizeof(arr) << endl; // Length of an array
    int n = sizeof(arr) / sizeof(arr[0]);
cout << "Length of an array: " << n << endl;
return 0;}
```

Output:

Size of arr[0]: 4

Size of arr: 20

Length of an array: 5

Types of Arrays in C++

- On the basis of Size

1. Fixed Size Array

2. Dynamic Sized Array

- On the basis of Dimensions

- 1. One-dimensional array (1-D arrays)

Syntax: `data_type array_name[array_size];`

Ex: `int nums[5];`

- 2. Two-dimensional (2D) array

Syntax: `data_type array_name[sizeof_1st_dimension][sizeof_2nd_dimension];`

Ex: `int nums[5][10];`

- 3. Three-dimensional array

Syntax: `data_type array_name[sizeof_1st_dimension][sizeof_2nd_dimension][sizeof_3rd_dimension];`

Ex: `int nums[5][10][2];`

EXAMPLE PROGRAM-1:

A program printing the number of pancakes in a carton/box.

CODE FOR THE PROGRAM:

```
#include <iostream>
using namespace std;
int main()
{
    int box [5] = {10,20,35,15,25};
    cout<< "Box 1 has "<<box[0]<<" pancakes" <<endl
    <<"Box 2 has " <<box[1] <<" pancakes" <<endl
    <<"Box 3 has " <<box[2] <<" pancakes" <<endl
    <<"Box 4 has " <<box[3] <<" pancakes" <<endl
    <<"Box 5 has " <<box[4] <<" pancakes";
    return 0;
}
```

Update Array Element

- To update an element in an array, we can use the index which we want to **update enclosed within the array subscript operator** and assign the new value.

```
arr[i] = new_value;
```

Passing Array to Function in C++

- pass arrays to functions as an argument same as we pass variables to functions but we know that the array name is treated as a pointer using this concept we can **pass the array to functions as an argument and then access all elements of that array using pointer.**
- So ultimately, **arrays are always passed as pointers to the function.** Let's see **3 ways to pass an array to a function**

1. Passing Array as a Pointer

In this method, we simply pass the array name in function call which means we **pass the address to the first element of the array**. In this method, we can modify the array elements within the function.

Syntax:

```
return_type function_name ( data_type *array_name ) {  
// set of statements
```

2. Passing Array as an Unsize Array

In this method, the function accepts the array using a simple array declaration with **no size as an argument**.

Syntax

```
return_type function_name ( data_type array_name[] ) {  
// set of statements
```

3. Passing Array as a Sized Array

In this method, the function accepts the array using a simple array declaration with size as an argument. We use this method by **sizing an array just to indicate the size of an array.**

Syntax

```
return_type function_name(data_type array_name[size_of_array]){  
// set of statements
```

ADVANTAGES OF AN ARRAY IN C/C++:

1. Random access of elements using array index.
2. Use of less line of code as it creates a single array of multiple elements.
3. Easy access to all the elements.
4. Traversal through the array becomes easy using a single loop.
5. Sorting becomes easy as it can be accomplished by writing less line of code.

DISADVANTAGES OF AN ARRAY IN C/C++:

1. Allows a fixed number of elements to be entered which is decided at the time of declaration.
2. Insertion and deletion of elements can be costly since the elements are needed to be managed in accordance with the new memory allocation.

Illustrating the Relationship between Array and Pointers

// C++ Program to Illustrate that Array Name is a Pointer that Points to First Element of the Array

```
#include <iostream>
using namespace std;
int main()
{
    // Defining an array
    int arr[] = { 1, 2, 3, 4 };
    // Define a pointer
    int* ptr = arr;
    // Printing address of the array using array name
    cout << "Memory address of arr: " << &arr << endl;
    // Printing address of the array using ptr
    cout << "Memory address of arr: " << ptr << endl;
    return 0;
}
```

Output:

```
Memory address of arr:
0x7fff2f2cabb0
Memory address of arr:
0x7fff2f2cabb0
```

Pointers

- Pointers are **symbolic representations of addresses**. They enable programs to simulate **call-by-reference** as well as to **create and manipulate dynamic data structures**. Iterating over elements in **arrays or other data structures is one of the main use of pointers**.
- A **pointer** is a variable whose **value is the address of another variable**.
- We can get the **memory address** of a variable by using the **&** operator

Syntax:

```
datatype *var_name;
```

```
int *ptr;           // ptr can point to an address which holds int data
```

How to use a pointer?

- Define a pointer variable
- **Assigning the address of a variable** to a pointer using the **unary operator (&)** which returns the address of that variable.
- **Accessing the value stored in the address** using **unary operator (*)** which returns the value of the variable located at the address specified by its operand.

Advantage of pointer

1. **Pointer reduces the code and improves the performance**, it is used to **retrieving strings, trees** etc. and used with **arrays, structures and functions**.
2. We can return multiple values from function using pointer.
3. It makes you able to **access any memory location** in the computer's memory.

Usage of pointer

There are many usage of pointers in C++ language.

1) Dynamic memory allocation

In c language, we can dynamically **allocate memory using malloc() and calloc()** functions where pointer is used.

2) Arrays, Functions and Structures

Pointers in c language are widely used in arrays, functions and structures. It reduces the code and improves the performance.

Symbols used in pointer

Symbol	Name	Description
& (ampersand sign)	Address operator	Determine the address of a variable.
* (asterisk sign)	Indirection operator	Access the value of an address.

Example:

```
#include <bits/stdc++.h>
using namespace std;
void geeks()
{
    int var = 20; // declare pointer variable
    int* ptr; // note that data type of ptr and var must be same
    ptr = &var; // assign the address of a variable to a pointer
    cout << "Value at ptr = " << ptr << "\n";
    cout << "Value at var = " << var << "\n";
    cout << "Value at *ptr = " << *ptr << "\n";
}
int main()
{
    geeks();
    return 0;
}
```

Output:

```
Value at ptr = 0x7ffc094904a4
Value at var = 20
Value at *ptr = 20
```

Declaring a pointer

The pointer in C++ language can be declared using * (asterisk symbol).

1.int * a; //pointer to int

2.char * c; //pointer to char

What is a void pointer?

- This **unique type of pointer**, which is available in C++, stands in for the lack of a kind.
- Pointers that **point to a value that has no type are known as void pointers** (and thus also an undetermined length and undetermined dereferencing properties).
- This indicates that void pointers are **very flexible** because they can **point to any data type**. This flexibility has benefits.
- **Direct dereference is not possible** with these pointers. Before they may be dereferenced, they must be **converted into another pointer type** that points to a specific data type.

// C++ program to demonstrate the need for void pointer

```
#include <iostream>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    int* ptr;
```

```
    float f = 90.6;
```

```
    ptr = &f; // error
```

```
    cout << "The value of *ptr is : " << *ptr << endl;
```

```
    return 0;
```

```
}
```

Output

error: cannot convert 'float*' to 'int*' in
assignment

```
#include <iostream>
using namespace std;
int main()
{
    // Initializing multiple variables of different data
    // type
    int n = 10;
    float f = 25.25;
    char c = '$';
    // Initializing a void pointer
    void* ptr;
    ptr = &n; // pointing to int
    cout << "The value of n " << n << endl;
    cout << "The Adress of n " << ptr << endl;
    ptr = &f; // pointing to float
    cout << "The value of n " << f << endl;
    cout << "The Adress of n " << ptr << endl;
    ptr = &c; // pointing to char
    cout << "The value of n " << c << endl;
    cout << "The Adress of n " << ptr << endl;
}
```

Output:

The value of n 10

The Adress of n 0x7ffd9f4eb284

The value of n 25.25

The Adress of n 0x7ffd9f4eb280

The value of n \$

The Adress of n 0x7ffd9f4eb27f

What is a invalid pointer?

A pointer must **point to a valid address**, not necessarily to useful items (like for arrays). We refer to these as **incorrect pointers**. Additionally, incorrect pointers are **uninitialized pointers**.

What is a null pointer?

A null pointer is not merely an **incorrect address**; it also points nowhere. Here are two ways to mark a pointer as NULL.

References and Pointers

There are 3 ways to pass C++ arguments to a function:

- Call-By-Value
- Call-By-Reference with a Pointer Argument
- Call-By-Reference with a Reference Argument

// C++ program to illustrate call-by-methods

```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```
// Pass-by-Value
```

```
int square1(int n)
```

```
{
```

```
    // Address of n in square1() is not the same as n1 in main()
```

```
    cout << "address of n1 in square1(): " << &n << "\n";
```

```
    // clone modified inside the function
```

```
    n *= n;
```

```
    return n; }
```

```
// Pass-by-Reference with Pointer Arguments
```

```
void square2(int* n)
```

```
{
```

```
    // Address of n in square2() is the same as n2 in main()
```

```
    cout << "address of n2 in square2(): " << n << "\n";
```

```
    // Explicit de-referencing to get the value pointed-to
```

```
    *n *= *n;
```

```
}
```

```
// Pass-by-Reference with Reference Arguments
```

```
void square3(int& n)
```

```
{
```

```
    // Address of n in square3() is the same as n3 in main()
```

```
    cout << "address of n3 in square3(): " << &n << "\n";
```

```
    // Implicit de-referencing (without '*')
```

```
    n *= n;
```

```
}
```

```
void geeks()
```

```
{
```

```
    // Call-by-Value
```

```
    int n1 = 8;
```

```
    cout << "address of n1 in main(): " << &n1 << "\n";
```

```
    cout << "Square of n1: " << square1(n1) << "\n";
```

```
    cout << "No change in n1: " << n1 << "\n";
```

```
    // Call-by-Reference with Pointer Arguments
```

```
    int n2 = 8;
```

```
    cout << "address of n2 in main(): " << &n2 << "\n";
```

```
    square2(&n2);
```

```
    cout << "Square of n2: " << n2 << "\n";
```

```
cout << "Change reflected in n2: " << n2 << "\n";
// Call-by-Reference with Reference Arguments
int n3 = 8;
cout << "address of n3 in main(): " << &n3 << "\n";
square3(n3);
cout << "Square of n3: " << n3 << "\n";
cout << "Change reflected in n3: " << n3 << "\n";
}
int main() { geeks(); }
```

Output:

```
address of n1 in main(): 0x7ffd71d45b2c
Square of n1: address of n1 in square1(): 0x7ffd71d45b0c
64
No change in n1: 8
address of n2 in main(): 0x7ffd71d45b28
address of n2 in square2(): 0x7ffd71d45b28
Square of n2: 64
Change reflected in n2: 64
address of n3 in main(): 0x7ffd71d45b24
address of n3 in square3(): 0x7ffd71d45b24
Square of n3: 64
Change reflected in n3: 64
```

References

- A reference variable is an **alias**, that is, **another name for an already existing variable**. **Once** a reference is **initialized with a variable**, either the **variable name** or the **reference name** may be used to refer to the variable.
- Reference variable as a **type of variable** that can act as a **reference to another variable**. **'&'** is used for signifying the **address of a variable** or any memory. Variables associated with reference variables can be accessed either by its name or by the reference variable associated with it.
- When a variable is declared as a reference, it becomes an **alternative name for an existing variable**. A variable can be declared as a reference by putting **'&'** in the declaration.

Syntax:

```
data_type &ref = variable;
```

References vs Pointers

References are often confused with pointers but **three major differences between references and pointers** are –

- You cannot have **NULL references**. You must always be able to assume that a reference is connected to a legitimate piece of storage.
- Once a reference is **initialized to an object**, it **cannot be changed** to refer to another object. Pointers can **be pointed to another object** at any time.
- A reference must be initialized when it is created. Pointers can be initialized at any time.

Example:

```
// C++ Program to demonstrate use of references
```

```
#include <iostream>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    int x = 10;
```

```
    // ref is a reference to x.
```

```
    int& ref = x;
```

```
    // Value of x is now changed to 20
```

```
    ref = 20;
```

```
    cout << "x = " << x << "\n";
```

```
    cout << "x = " << &x << "\n";
```

```
    // Value of x is now changed to 30
```

```
    x = 30;
```

```
    cout << "ref = " << ref << "\n";
```

```
    return 0;
```

```
}
```

Output:

```
x = 20
```

```
x = 0x7ffcc492fba4
```

```
ref = 30
```


How to create a reference?

- Reference can be **created by simply using an ampersand (&) operator**. When we create a variable, then it **occupies some memory location**.
- We can create a reference of the variable; therefore, we can **access the original variable by using either name** of the variable or reference.

For example:

```
int a=10;
```

Now, we create the reference variable of the above variable.

```
int &ref=a;
```

C++ provides two types of references:

- References to non-const values
- References as aliases

References to non-const values

- It can be declared by **using & operator** with the reference type variable.

Example:

```
#include <iostream>
using namespace std;
int main()
{
int a=10;
int &value=a;
std::cout << value << std::endl;
return 0;
}
```

References as aliases

- References as aliases is **another name of the variable** which is being referenced.

Example:

```
#include <iostream>
using namespace std;
int main()
{
int a=70; // variable initialization
int &b=a;
int &c=a;
std::cout << "Value of a is :" <<a<< std::endl;
std::cout << "Value of b is :" <<b<< std::endl;
std::cout << "Value of c is :" <<c<< std::endl;
return 0;}

```

Properties of References

1. Initialization

It must be initialized at the time of the declaration.

2. Reassignment

It cannot be reassigned means that the reference variable cannot be modified.

3. References as shortcuts

With the help of references, we can easily access the nested data.

4. Function Parameters

References can also be passed as a function parameter. It does not create a copy of the argument and behaves as an alias for a parameter. It enhances the performance as it does not create a copy of the argument.

Applications of Reference in C++

- There are multiple applications for references in C++, a few of them are mentioned below:

1. Modify the passed parameters in a function

2. Avoiding a copy of large structures

3. In For Each Loop to modify all objects

4. For Each Loop to avoid the copy of objects

Advantages of using References

- 1.Safer:* Since references must be initialized, wild references like wild pointers are unlikely to exist. It is still possible to have references that **don't refer to a valid location**
- 2.Easier to use:* References don't need a **dereferencing operator to access the value**. They can be used like normal variables. The '&' operator is needed only at the time of declaration. Also, **members of an object reference can be accessed with the dot operator ('.')**, unlike pointers where the **arrow operator (->)** is needed to **access members**.

Limitations of References

1. Once a reference is created, it **cannot** be later **made to reference another object**; it cannot be reset. This is often done with pointers.
2. References **cannot be NULL**. Pointers are often made NULL to indicate that they are not pointing to any valid thing.
3. A reference must be initialized when declared. There is **no such restriction with pointers**.

Dynamic Memory Allocation

Memory allocation

Reserving or providing space to a variable is called memory allocation. For storing the data, memory allocation can be done in **two ways** -

- 1. Static allocation or compile-time allocation** - Static memory allocation means **providing space for the variable**. The size and data type of the variable is known, and it remains **constant** throughout the program.
- 2. Dynamic allocation or run-time allocation** - The allocation in which memory is **allocated dynamically**. In this type of allocation, the **exact size of the variable** is not known in advance. Pointers play a major role in dynamic memory allocation.

Dynamic memory allocation in C/C++ refers to performing **memory allocation manually** by a programmer. Dynamically allocated memory is allocated on **Heap**, and non-static and local variables get memory allocated on **Stack**

Why Dynamic memory allocation?

- Dynamically we can **allocate storage** while the program is in a **running state**, but variables cannot be created "on the fly". Thus, there are two criteria for dynamic memory allocation -
- A dynamic space in the memory is needed.
- Storing the address to access the variable from the memory
- Similarly, we do memory de-allocation for the variables in the memory.

In C++, memory is divided into two parts -

- **Stack** - All the variables that are declared inside any function take memory from the stack.
- **Heap** - It is unused memory in the program that is generally used for dynamic memory allocation

Dynamic memory allocation using the new operator

- To allocate the space dynamically, the operator new is used. It means creating a request for memory allocation on the free store. If memory is available, memory is initialized, and the address of that space is returned to a pointer variable.

Syntax

Pointer_variable = new data_type;

The pointer_variable is of pointer data_type. The data type can be int, float, string, char, etc.

Ex: int *m = new int

Initialize memory

We can also initialize memory using new operator. Ex: int *m = new int(20);

- **Allocate a block of memory**
- We can also use a new operator to allocate a block(array) of a particular data type. For example :
int *arr = new int[10]
- Here we have dynamically allocated memory for ten integers which also returns a pointer to the first element of the array. Hence, arr[0] is the first element and so on.

Delete operator

Delete the allocated space in C++ using the **delete** operator.

Syntax

```
delete pointer_variable_name
```

Example:

1. `delete m; // free m that is a variable`
2. `delete [] arr; // Release a block of memory`

Example to demonstrate dynamic memory allocation:

```
#include <iostream>
using namespace std;
int main ()
{
    // Pointer initialization to null
    int* m = NULL;
    // Request memory for the variable using new operator
    m = new(nothrow) int;
    if (!m)
        cout<< "allocation of memory failed\n";
    else
    {
        // Store value at allocated address
        *m=29;
        cout<< "Value of m: " << *m <<endl;
    }
}
```

```
// Request block of memory using new operator
float *f = new float(75.25);
cout<< "Value of f: " << *f <<endl;
// Request block of memory of size
int size = 5;
int *arr = new(nothrow) int[size];
if (!arr)
    cout<< "allocation of memory failed\n";
else
{
    for (int i = 0; i < size; i++)
        arr[i] = i+1;
    cout<< "Value store in block of memory: ";
for (int i = 0; i < size; i++)
    cout<<arr[i] << " ";
}
```

```
// freed the allocated memory
    delete m;
    delete f;
// freed the block of allocated memory
    delete[] arr;
    return 0;
}
```

Output:

Value of m: 29

Value of f: 75.25

Value store in block of memory: 1 2 3 4 5

UNIT-2

FUNCTION OVERLOADING AND CONSTRUCTORS

Function Overloading

- Function overloading is a **feature of object-oriented programming** where **two or more functions can have the same name** but **different parameters**. When a function name is overloaded with different jobs it is called Function Overloading.
- Function Overloading “**Function**” **name should be the same** and the **arguments should be different**. Function overloading can be considered as an example of a **polymorphism** feature in C++.
- If **multiple functions having same name** but **parameters** of the functions should be **different** is known as Function Overloading.
- If we have to **perform only one operation** and **having same name** of the functions increases the readability of the program.
- **function overloading**, multiple functions can have the same name with different parameters.

Example:

```
int myFunction(int x)
```

```
float myFunction(float x)
```

```
double myFunction(double x, double y)
```


- It involves defining **multiple functions in the same scope** with the same name but **different parameter lists**.
- The compiler selects the appropriate function to call based on the number and types of arguments provided during the function call.
- Function overloading is **determined at compile-time**.
- The ability to **declare numerous functions** with the same name but different parameters is known as function overloading in C++.
- In other words, when a function is overloaded with **many different jobs but has the same name** for all, it is overloaded. This is a feature of object-oriented programming and is built on the **idea of polymorphism**.
- It allows using a **single function name** to denote **many behaviors** depending on the **parameters** given.

Example Program:

```
#include <iostream>
using namespace std;
void add(int a, int b)
{ cout << "sum = " << (a + b); }
void add(double a, double b)
{ cout << endl << "sum = " << (a + b); }
// Driver code
int main()
{ add(10, 2);
  add(5.3, 6.2);
  return 0; }
```

OUTPUT:

sum = 12

sum = 11.5

Types of function overloading

There are two types of function overloading

1. **Compile time overloading**– In compile time overloading, **functions are overloaded using different signatures**. Signature of a function includes its return type, number of parameters and types of parameters.
2. **Runtime overloading** -In runtime overloading, functions are **overloaded using a different number of parameters**.

Function Overloading and Ambiguity

- When the compiler is unable to decide which function is to be invoked among the overloaded function, this situation is known as **function overloading**.
- When the **compiler shows the ambiguity error**, the **compiler does not run** the program.

Causes of Function Overloading:

- Type Conversion.
- Function with default arguments.
- Function with pass by reference.

1.Type Conversion

- Function overloading can occur when there is a need to **handle different types of input parameters by automatically converting them to the expected type of parameters.**
- If there is **no exact match**, the compiler looks for compatible standard conversions that can be applied to the arguments to match the parameters of the overloaded functions.
- It tries to find the closest match by converting the arguments to the required parameter types.
- For example, **converting an integer to a floating-point number**, float to double, or converting a string representation of a number to an actual numeric type, are standard type conversion scenarios.

Example:

```
#include <iostream>
int main() {
    int integerNum = 42;
    double doubleNum = static_cast<double>(integerNum); // Explicit casting
    std::cout << "Integer: " << integerNum << std::endl;
    std::cout << "Double: " << doubleNum << std::endl;
    return 0;
}
```

2. Function With Default Arguments

- C++ allows function arguments to **hold a default value**, and the **argument is assigned this value when no value is passed** to the function argument in the function call.
- Often, when functions with a **default value for their arguments are overloaded**, it can lead to **compilation errors**, and we must to **pay extra attention to avoid such cases**.
- For example, consider a function that **calculates the area of a rectangle**. You could **define default values for the width and the height** but still **allow the caller to provide different values** when needed.

Example:

```
#include <iostream>
int calculateArea(int width = 5, int height = 10) {
    return width * height; }
int main() {
    int area1 = calculateArea(); // Uses default values
    int area2 = calculateArea(8); // Uses default height
    int area3 = calculateArea(6, 12); // Uses provided values
    std::cout << "Area 1: " << area1 << std::endl;
    std::cout << "Area 2: " << area2 << std::endl;
    std::cout << "Area 3: " << area3 << std::endl;
    return 0;
}
```

Output:

```
Area 1: 50
Area 2: 80
Area 3: 72
```

3. Function With Pass By Reference

- There are **two ways to create a function call**, i.e., either by **value or by reference**. When parameters to a function are passed by their reference, such calls are referred to as a *call by reference*.
- **Pass-by-reference** can be **more efficient than pass-by-value** (making a copy of the data) for **large data structures** because it **avoids unnecessary data duplication**.

Example:

```
#include <iostream>
void modifyValue(int &num) {
    num *= 2; }
int main() {
    int value = 5;
    std::cout << "Original Value: " << value << std::endl;
    modifyValue(value);
    std::cout << "Modified Value: " << value << std::endl;
    return 0;
}
```

Ways of Function Overloading

1. functions have different parameter type

sum(int a, int b)

sum(double a, double b)

2. functions have a different number of parameters

sum(int a, int b)

sum(int a, int b, int c)

3. functions have a different sequence of parameters

sum(int a, double b)

sum(double a, int b)

Example: functions have different parameter type

```
#include <iostream>
using namespace std;
void add(int a, int b)
{
    cout << "sum = " << (a + b);
}
void add(double a, double b)
{
    cout << endl << "sum = " << (a + b);
}
// Driver code
int main()
{
    add(10, 2);
    add(5.3, 6.2);
    return 0;
}
```

Example: functions have a different number of parameters

```
#include <iostream>
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.
    cout<<C.add(10, 20)<<endl;
    cout<<C.add(12, 20, 23);
    return 0;
}
```

Example: functions have a different sequence of parameters

```
#include<iostream>
using namespace std;
void add(int a, double b)
{
    cout<<"sum = "<<(a+b);
}
void add(double a, int b)
{
    cout<<endl<<"sum = "<<(a+b);
}
// Driver code
int main()
{
    add(10,2.5);
    add(5.5,6);
    return 0;
}
```

Rules Of Function Overloading

some rules or steps to ensure function overloading. They are as follows:

- 1.Function Name:** Overloaded functions must have the **same name**.
- 2.Parameter List:** Overloaded functions must have **different parameter lists**. The parameters can differ in terms of number, type, or both. The order of the parameters is also significant.
- 3.Return Type:** The return type of the functions is not considered during function overloading. Overloaded functions can have the **same or different return types**.
- 4.Function Signature:** The function signature includes the function name and the parameter list. Overloaded functions must **have different function signatures**.
- 5.Ambiguity:** The overloaded functions should be distinguishable by the compiler. If the **compiler cannot determine the best match during function call** resolution due to ambiguous or conflicting overloaded functions, it will result in a compilation error.
- 6.Default Arguments:** Functions with default arguments can be overloaded. The **presence or absence of default arguments** is considered when resolving function calls.

Advantages Of Function Overloading

- Improved **Readability and Maintainability**
- Code **Reusability**
- Polymorphism and Flexibility
- Type Safety
- Compatibility and Scalability

Disadvantages Of Function Overloading

- Ambiguity
- Increased complexity
- Maintenance

constructor

- A constructor in c++ is a **special type of method or member function that is called automatically at the time of object creation**. It refer object as the instance of the class.
- Its main purpose is to **initialize the object with values** that are specified by the **user at the time of creation** and if the **user doesn't specify any value at the time of creation** then it will assign or initialize the object with default values.
- The constructor has the **same name as that of the class name** which is defined or mentioned by the user as this **name is used by the compiler** to have a distinguishing feature between constructor in c++ and other functions of the class.
- **Constructor in C++** is a special method that is **invoked automatically at the time an object of a class is created**. It is used to initialize the data members of new objects generally.
- The constructor in C++ has the **same name as the class or structure**. It constructs the values i.e. provides data for the object which is why it is known as a constructor.

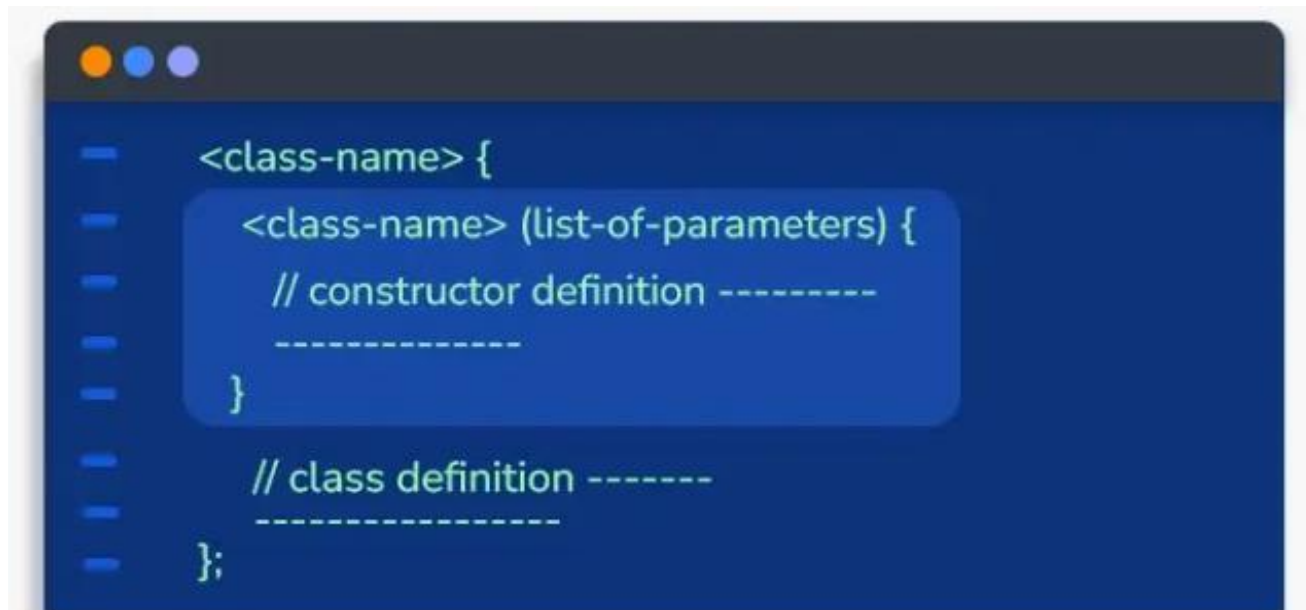
Syntax of Constructors in C++

The prototype of the constructor looks like this:

```
<class-name> () {
```

```
...
```

```
}
```



```
<class-name> {  
    <class-name> (list-of-parameters) {  
        // constructor definition -----  
        -----  
    }  
  
    // class definition -----  
    -----  
};
```

Characteristics of Constructors in C++

- The **name of the constructor is the same** as its class name.
- Constructors are **mostly declared in the public section** of the class though they can be declared in the private section of the class.
- Constructors do **not return values**; hence they do **not have a return type**.
- A constructor gets called automatically when we create the object of the class.

Types of Constructor Definitions in C++

In C++, there are 2 methods by which a constructor can be declared:

1. Defining the Constructor Within the Class

```
<class-name> (list-of-parameters) {  
// constructor definition  
}
```

2. Defining the Constructor Outside the Class

```
<class-name> {  
  
// Declaring the constructor  
// Definition will be provided outside  
<class-name>();  
// Defining remaining class  
}  
<class-name>: :<class-name>(list-of-parameters) {  
// constructor definition  
}
```

Example to show defining the constructor within the class

```
#include <iostream>
using namespace std;
// Class definition
class student {
    int rno;
    char name[50];
    double fee;
public:
    /*Here we will define a constructor inside the same class for which we are creating it. */
    student()
    {
        // Constructor within the class
        cout << "Enter the RollNo:";
        cin >> rno;
        cout << "Enter the Name:";
        cin >> name;
```

```
cout << "Enter the Fee:";
    cin >> fee;
}
// Function to display the data defined via constructor
void display()
{
    cout << endl << rno << "\t" << name << "\t" << fee;
}
};
int main()
{
    student s;
    /* constructor gets called automatically as soon as the object of the class is declared */
    s.display();
    return 0;
}
```

Output:

```
Enter the RollNo:11
Enter the Name: Aman
Enter the Fee:10111
11 Aman 10111
```

defining the constructor outside the class

```
#include <iostream>
```

```
using namespace std;
```

```
class student {
```

```
    int rno;
```

```
    char name[50];
```

```
    double fee;
```

```
public:
```

```
    /* To define a constructor outside the class, we need to declare it within the class first. Then we can define the implementation anywhere. */
```

```
    student();
```

```
    void display();
```

```
};
```

```
/*
```

```
Here we will define a constructor outside the class for which we are creating it. */
```

```
student::student()
{
    // outside definition of constructor
    cout << "Enter the RollNo:";
    cin >> rno;
    cout << "Enter the Name:";
    cin >> name;
    cout << "Enter the Fee:";
    cin >> fee;
}
void student::display()
{
    cout << endl << rno << "\t" << name << "\t" << fee;
}
int main()
{
    student s;
    /* constructor gets called automatically as soon as the object of the class is declared */
    s.display();
    return 0;
}
```

Types of Constructors

There are three types of constructors:-

1. Default Constructor-No parameters. They are used to create an object with default values.

Syntax:

```
className() {  
// body_of_constructor  
}
```

2. Parameterized Constructor-Takes parameters. Used to create an object with specific initial values.

Syntax:

```
className (parameters...) {  
// body  
}
```

3. Copy Constructor-Takes a reference to another object of the same class. Used to create a copy of an object.

Syntax:

```
ClassName (ClassName &obj)  
{  
// body_containing_logic  
}
```

Move Constructor: Takes an rvalue reference to another object. Transfers resources from a temporary object. The move constructor is a recent addition to the family of constructors in C++. It is like a copy constructor that constructs the object from the already existing objects., but instead of copying the object in the new memory, it makes use of move semantics to transfer the ownership of the already created object to the new object without creating extra copies.

Syntax:

```
className (className&& obj) {  
// body of the constructor  
}
```

Default Constructor

```
#include <iostream>
using namespace std;
//Class Name: Default_construct
class Default_construct
{
public:
int a, b;
    // Default Constructor
    Default_construct()
    {
        a = 100;
        b = 200;
    } };
int main() {
    Default_construct con; //Object created
    cout << "Value of a: " << con.a;
        cout<< "Value of b: " << con.b;
    return 0; }
```

OUTPUT:

Value of a: 100

Value of b: 200

Parameterized Constructor: Example

```
#include <iostream>
using namespace std;
class Rectangle {
private:
    double length;
    double breadth;
public:
    // parameterized constructor
    Rectangle(double l, double b) {
        length = l;
        breadth = b; }
    double calculateArea() {
        return length * breadth;
    } };
int main() { // create objects to call constructors
    Rectangle obj1(10,6);
    Rectangle obj2(13,8);
    cout << "Area of Rectangle 1: " << obj1.calculateArea();
    cout << "Area of Rectangle 2: " << obj2.calculateArea();
    return 0; }
```


Copy constructor:Example

```
#include <iostream>
```

```
using namespace std;
```

```
// class name: Rectangle
```

```
class Rectangle {
```

```
private:
```

```
    double length;
```

```
    double breadth;
```

```
public:
```

```
    // parameterized constructor
```

```
    Rectangle(double l, double b) {
```

```
        length = l;
```

```
        breadth = b;
```

```
    }
```

```
    // copy constructor with a Rectangle object as parameter copies data of the obj  
parameter
```

```
Rectangle(Rectangle &obj) {
    length = obj.length;
    breadth = obj.breadth;
}
double calculateArea() {
    return length * breadth;
}
};
int main() {
    // create objects to call constructors
    Rectangle obj1(10,6);
    Rectangle obj2 = obj1; //copy the content using object
    //print areas of rectangles
    cout << "Area of Rectangle 1: " << obj1.calculateArea();
    cout << "Area of Rectangle 2: " << obj2.calculateArea();
    return 0;
}
```

Overloading Constructors

- Constructor overloading can be defined as **having multiple constructors with different parameters** so that **every constructor can perform a different task**.
- The constructor that does **not take any argument** is called the **default constructor**.
- A constructor that **has parameters** is known as a **parameterized constructor**.
- Overloaded constructors essentially **have the same name** (exact name of the class) and **different by number and type of arguments**.
- A constructor is called depending upon the **number and type of arguments passed**.
- While **creating the object**, arguments must be passed to let compiler know, which constructor needs to be called.
- Constructors can be **overloaded in a similar way** as function overloading.
- **Overloaded constructors have the same name** (name of the class) but the **different number of arguments**. Depending upon the number and type of arguments passed, the corresponding constructor is called.

- The use of multiple constructors in the same class is known as **Constructor Overloading**.
- The constructor must follow **one or both of the two rules** below.
- All the constructors in the class should have a **different number of parameters**.
- It is also allowed in a class to have constructors with the **same number of parameters and different data types**.

syntax for the declaration of constructor overloading:

```
class ClassName {  
    public:  
        ClassName() {  
            body; // Constructor with no parameter.  
        }  
        ClassName(int x, int y) {  
            body; // Constructor with two parameters.  
        }  
        ClassName(int x, int y, int z) {  
            body; // Constructor with three parameters.  
        }  
        ClassName(ClassName & object) {  
            body; // Constructor with the same class object as a parameter.  
        }  
        // Other member functions.  
};
```

```
#include <iostream>
using namespace std;
class Person {
private:
    int age;
public:
    // 1. Constructor with no arguments
    Person() {
        age = 20;
    }
    // 2. Constructor with an argument
    Person(int a) {
        age = a;
    }
}
```

```
int getAge() {  
    return age;  
}  
};  
int main() {  
    Person person1, person2(45);  
    cout << "Person1 Age = " << person1.getAge() << endl;  
    cout << "Person2 Age = " << person2.getAge() << endl;  
    return 0;  
}
```

Output:

Person1 Age = 20

Person2 Age = 45

Advantages of Using Constructor Overloading

1. Constructor overloading, inside a class we can **declare multiple constructors with different parameters**, this helps in providing flexibility in terms of how objects of that class can be created.
2. Constructor overloading helps us **reuse the code by invoking one constructor from another constructor of the same class**, this results in the reduction of code duplication and makes the maintenance of code very easier for us.
3. We can optimize the efficiency of our program by **avoiding unnecessary initialization and by minimizing the chances of errors** by implementing constructor overloading.

Disadvantages of using Constructor Overloading

1. Has a **Confusing and complex syntax** which makes it more difficult to understand and maintain the code.
2. Each overloaded **constructor adds more code** to the class which can **increase the size of the executable file** and makes the program run slower.
3. It makes the **debugging process** difficult because **multiple constructors** are used. So, it is kind a **hard to identify the source of the error**

Copy Constructors

- A **copy constructor** is a type of constructor that **initializes an object using another object of the same class**.
- In simple terms, a constructor which **creates an object by initializing it with an object of the same class**, which has been **created previously** is known as a **copy constructor**.
- The **process of initializing members of an object** through a copy constructor is known as **copy initialization**. It is also called **member-wise initialization** because the copy constructor initializes one object with the existing object, both belonging to the same class on a member-by-member copy basis.
- A Copy constructor is an **overloaded** constructor used to declare and initialize an object from another object.

- A **copy constructor** is a type of constructor that initializes an object using another object of the same class.
- In simple terms, a **constructor which creates an object by initializing it with an object of the same class, which has been created previously** is known as a **copy constructor**.
- The process of initializing members of an object through a copy constructor is known as **copy initialization**.
- It is also called **member-wise initialization** because the copy constructor **initializes one object with the existing object**, both belonging to the **same class on a member-by-member copy basis**.
- 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.

Syntax of Copy Constructor

```
className (const className &obj)
```

```
{
```

```
    member1 = obj.member1;
```

```
    // for dynamic resources
```

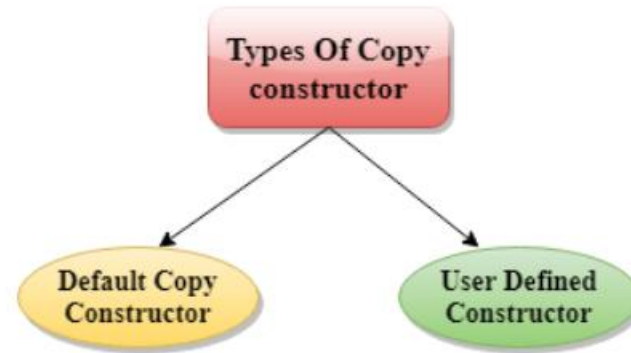
```
    .....
```

```
}
```

Example program for copy constructor:

```
#include <iostream>
using namespace std;
class A
{ public:
  int x;
  A(int a)          // parameterized constructor.
  { x=a; }
  A(A &i)           // copy constructor
  {
    x = i.x;
  } };
int main()
{ A a1(20);        // Calling the parameterized constructor.
  A a2(a1);        // Calling the copy constructor.
  cout<<a2.x;
  return 0; }
```

Copy Constructor is of two types:



Default Copy constructor: The compiler defines the default copy constructor. If the user defines **no copy constructor**, compiler supplies its constructor.

An implicitly defined copy constructor will **copy the bases and members of an object in the same order** that a constructor would **initialize the bases and members of the object**.

User Defined constructor: The programmer defines the user-defined constructor.

Syntax Of User-defined Copy Constructor:

```
Class_name(const class_name &old_object);
```

User Defined Copy Constructor

```
#include <iostream>
#include <string.h>
using namespace std;
class student {
    int rno;
    string name;
    double fee;
public:
    // Parameterized constructor
    student(int, string, double);
    // Copy constructor
    student(student& t) {
        rno = t.rno;
        name = t.name;
        fee = t.fee;
        cout << "Copy Constructor Called" << endl; }
    // Function to display student details
    void display(); };

```

```
// Implementation of the parameterized constructor
student::student(int no, string n, double f)
{
    rno = no;
    name = n;
    fee = f; }
// Implementation of the display function
void student::display()
{
    cout << rno << "\t" << name << "\t" << fee << endl;
}
int main()
{
    // Create student object with parameterized constructor
    student s(1001, "Manjeet", 10000);
    s.display();
    // Create another student object using the copy
    // constructor
    student manjeet(s);
    manjeet.display();
    return 0;
}
```


Default Copy Constructor

```
// Implicit copy constructor Calling
#include <iostream>
using namespace std;
class Sample {
    int id;
public:
    void init(int x) { id = x; }
    void display() { cout << endl << "ID=" << id; }
};
int main()
{
    Sample obj1;
    obj1.init(10);
    obj1.display();
    // Implicit Copy Constructor Calling
    Sample obj2(obj1); // or obj2=obj1;
    obj2.display();
    return 0; }
```

Output:

ID=10

ID=10

When Copy Constructor is called

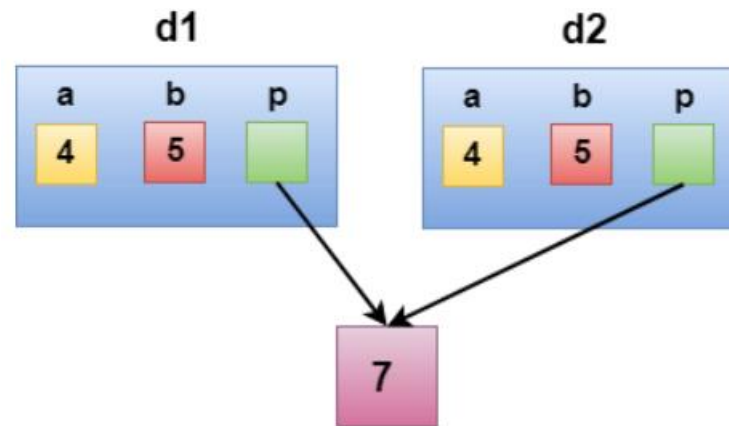
- Copy Constructor is called in the following **scenarios**:
- When we **initialize the object with another existing object of the same class** type. For example, **Student s1 = s2**, where Student is the class.
- When the object of the same class type is **passed by value as an argument**.
- When the **function returns the object of the same class** type by value.

Two types of copies are produced by the constructor:

- Shallow copy
- Deep copy

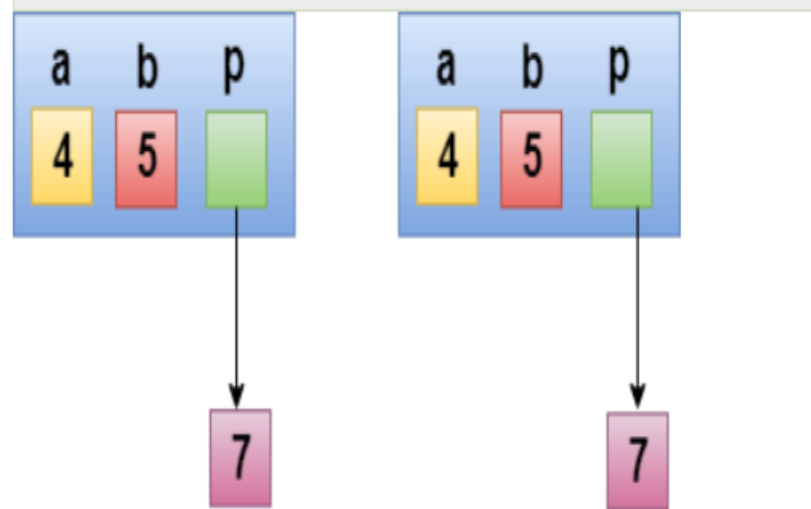
Shallow Copy

- The **default copy constructor** can only **produce the shallow copy**.
- A Shallow copy is defined as the **process of creating the copy of an object by copying data of all the member variables** as it is.



Deep copy

- Deep copy **dynamically allocates the memory** for the copy and then **copies the actual value**, both the source and copy have distinct memory locations.
- In this way, both the source and copy are distinct and will **not share the same memory location**. Deep copy requires us to write the user-defined constructor.



Destructors

- Destructor is an **instance member function that is invoked automatically** whenever an **object is going to be destroyed**. Meaning, a destructor is the **last function that is going to be called before an object is destroyed**.

- A destructor is a **member function that is invoked automatically when the object goes out of scope or is explicitly destroyed by a call to delete or delete[]** . A destructor has the **same name as the class** and is preceded by a **tilde (~)**. For example, the destructor for class String is declared: ~String() .

Syntax to Define Destructor:

The syntax for defining the destructor within the class:

```
~ <class-name>() { // some instructions }
```

Just like any other member function of the class, we can define the **destructor outside the class** too:

```
<class-name> {  
public:  
~<class-name>();  
}
```

```
<class-name> :: ~<class-name>() {  
// some instructions  
}
```

Declaring destructors

- Destructors are functions with the **same name as the class but preceded by a tilde (~)**. Several **rules** govern the **declaration of destructors**. Destructors:
 - Don't accept arguments.
 - Don't return a value (or **void**).
 - Can't be declared as **const**, **volatile**, or **static**. However, they can be invoked for the destruction of objects declared as **const**, **volatile**, or **static**.
 - Can be declared as **virtual**. Using **virtual destructors**, you can **destroy objects without knowing their type**—the correct destructor for the object is invoked using the virtual function mechanism. Destructors can also be **declared as pure virtual functions for abstract classes**.

Characteristics of a Destructor

- A destructor is also a **special member function** like a constructor. Destructor **destroys the class objects** created by the constructor.
- Destructor has the **same name as their class name** preceded by a tilde (~) symbol.
- It is **not possible to define more than one destructor**.
- The destructor is **only one way to destroy the object** created by the constructor. Hence, destructor cannot be overloaded.
- It **cannot** be declared **static or const**.
- Destructor **neither requires any argument** nor returns any value.
- It is **automatically called** when an **object goes out of scope**.
- Destructor **release memory space** occupied by the objects created by the constructor.
- In destructor, **objects are destroyed** in the reverse of an object creation.

// C++ program to demonstrate the execution of constructor and destructor

```
#include <iostream>
using namespace std;
class Test {
public:
    // User-Defined Constructor
    Test() { cout << "\n Constructor executed"; }
    // User-Defined Destructor
    ~Test() { cout << "\nDestructor executed"; }
};
int main()
{
    Test t;
    return 0;
}
```

Output:

Constructor executed

Destructor executed

program demonstrates the number of times constructors and destructors are called:

```
// C++ program to demonstrate the number of times constructor and destructors are called
```

```
#include <iostream>
```

```
using namespace std;
```

```
// It is static so that every class object has the same value
```

```
static int Count = 0;
```

```
class Test {
```

```
public:
```

```
    // User-Defined Constructor
```

```
    Test()
```

```
{
```

```
    // Number of times constructor is called
```

```
    Count++;
```

```
    cout << "No. of Object created: " << Count << endl;
```

```
}
```

```
// User-Defined Destructor
~Test()
{
    // It will print count in decending order
    cout << "No. of Object destroyed: " << Count << endl;
    Count--;
    // Number of times destructor is called
} };
// driver code
int main()
{
    Test t, t1, t2, t3;
    return 0;
}
```

Output:

No. of Object created: 1

No. of Object created: 2

No. of Object created: 3

No. of Object created: 4

No. of Object destroyed: 4

No. of Object destroyed: 3

No. of Object destroyed: 2

No. of Object destroyed: 1

When is the destructor called?

- A destructor function is called automatically when the **object goes out of scope** or is **deleted**. Following are the cases where destructor is called:
 1. Destructor is called when the **function ends**.
 2. Destructor is called when the **program ends**.
 3. Destructor is called when a **block containing local variables** ends.
 4. Destructor is called when a **delete operator is called**.

How to call destructors explicitly?

- **Destructor** can also be called explicitly for an object. We can call the destructors explicitly using the following statement:
 - `object_name.~class_name()`

Operator overloading

- Operator overloading is a **compile-time polymorphism**. It is an **idea of giving special meaning to an existing operator** in C++ without changing its original meaning.
- C++ has the ability to **provide the operators with a special meaning for a data type**, this ability is known as operator overloading.
- Operator overloading is a compile-time polymorphism. For example, we can **overload an operator '+'** in a class like **String** so that we can **concatenate two strings** by just using +.
- 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.
- Other example classes where **arithmetic operators may be overloaded** are **Complex Numbers, Fractional Numbers, Big integers**, etc.

Syntax: The syntax for overloading an operator is similar to that of function with the addition of the operator keyword followed by the operator symbol.

returnType operator symbol (arguments) { }

- returnType - the return type of the function
- operator - a special keyword
- symbol - the operator we want to overload (+, <, -, ++, etc.)
- arguments - the arguments passed to the function

C++ Program to Demonstrate Operator Overloading:

```
#include <iostream>
using namespace std;
class Complex {
private:  int real, imag;
public:
Complex(int r = 0, int i = 0)  {
    real = r;
    imag = i;  }
// This is automatically called when '+' is used with  between two Complex objects
Complex operator+(Complex const& obj)  {
    Complex res;
    res.real = real + obj.real;
    res.imag = imag + obj.imag;
    return res;  }
void print() {
    cout << real << " + i" << imag << "\n"; }};
int main(){
    Complex c1(10, 5), c2(2, 4);
    Complex c3 = c1 + c2;  c3.print();}
```

OUTPUT:12+i9

Operators that can be Overloaded in C++

We can overload

1. Unary operators
2. Binary operators
3. Special operators ([], (), etc)

Operator that cannot be overloaded are as follows:

1. Scope operator (::)
2. Sizeof
3. member selector(.)
4. member pointer selector(*)
5. ternary operator(?:)

Overloadable/Non-overloadable Operators

Following is the list of operators which can be overloaded –

+	-	*	/	%	^
&		~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	
+=	-=	/=	%=	^=	&=
=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

Following is the list of operators, which can not be overloaded –

::	.*	.	?:
----	----	---	----

Rules for Operator Overloading

- **Existing operators** can only be **overloaded**, but the **new operators cannot be overloaded**.
- The overloaded operator contains **atleast 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**.

Operators that can be overloaded	Examples
Binary Arithmetic	+, -, *, /, %
Unary Arithmetic	+, -, ++, —
Assignment	=, +=, *=, /=, -=, %=
Bitwise	& , , << , >> , ~ , ^
De-referencing	(->)
Dynamic memory allocation, De-allocation	New, delete
Subscript	[]
Function call	()
Logical	&, , !
Relational	>, < , = =, <=, >=

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

Syntax of Operator Overloading

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

program to overload the binary operators

```
#include <iostream>
using namespace std;
class A
{
    int x;
    public:
    A(){}
    A(int i)
    {
        x=i;
    }
    void operator+(A);
    void display();
};
void A :: operator+(A a)
{
    int m = x+a.x;
    cout<<"The result of the addition of two objects is : "<<m;
}
int main()
{
    A a1(5);
    A a2(4);
    a1+a2;
    return 0;
}
```

Output:

The result of the addition of
two objects is : 9

Unary Operators Overloading

The unary operators **operate on a single operand** and following are the examples of Unary operators –

- The increment (++) and decrement (--) operators.
- The unary minus (-) operator.
- The logical not (!) operator.

binary operators:

binary operators very frequently like addition (+) operator, subtraction (-) operator and division (/) operator.

relational operators:

relational operators supported by C++ language like (<, >, <=, >=, ==, etc.) which can be **used to compare** C++ built-in data types.

minus (-) operator can be overloaded for prefix as well as postfix:

```
#include <iostream>
using namespace std;
class Distance {
private:
    int feet;        // 0 to infinite
    int inches;     // 0 to 12
public:
    // required constructors
    Distance() {
        feet = 0;
        inches = 0;
    }
    Distance(int f, int i) {
        feet = f;
        inches = i;
    }
}
```

```
// method to display distance
void displayDistance() {
    cout << "F: " << feet << " I:" << inches <<endl;
}
// overloaded minus (-) operator
Distance operator- () {
    feet = -feet;
    inches = -inches;
    return Distance(feet, inches);
}
};
int main() {
    Distance D1(11, 10), D2(-5, 11);
    -D1;           // apply negation
    D1.displayDistance(); // display D1
    -D2;           // apply negation
    D2.displayDistance(); // display D2
    return 0;
}
```

simple example of overloading the binary operators.

```
#Include <Iostream>
```

```
Using Namespace Std;
```

```
Class A
```

```
{   Int X;
```

```
   Public:
```

```
   A(){}
```

```
   A(int I)
```

```
   {   X=i;   }
```

```
   Void Operator+(a);
```

```
   Void Display();  };
```

```
Void A :: Operator+(A a)
```

```
{   int m = x+a.x;
```

```
   cout<<"The result of the addition of two objects is : "<<m;   }
```

```
int main()
```

```
{   A a1(5);
```

```
   A a2(4);
```

```
   a1+a2;
```

```
   return 0;  }
```

OUTPUT:

The result of the addition of two objects is : 9

creating a member operator function

- As a member function, a member operator function **has access to one operand through the 'this' pointer, and the other operand(s) are arguments.**
- A member operator function has **access to the private member variables and functions of its class**, and of the **other operand** if it is of the **same class type**.
- The Operator is handled as a **class member when you overload an operator with a member function**. According to this, a class member that defines the **operator function must be one of the operands**.
- Any **compatible type**, including additional user-defined types, may be used for the other operand.
- The overloaded operator **must be added as a member function** of the left operand.
- The **left operand** becomes the **implicit *this object**
- **All other operands** become **function parameters**.

- The **general syntax for utilizing a member function** to overload an operator is as follows:

```
return_type operator op(parameters) {  
    // Define the behavior of the operator  
    // You can access the current object using 'this' pointer  
}
```

- Here, 'op' represents the Operator you want to overload (e.g., '+', '-', '*', '/'), and 'parameters' represent the Operator's operands. The 'return_type' specifies the type of value the Operator should return.

For example, let's overload the '+' operator for a 'Vector' class using a member function:

```
class Vector {
```

```
Public:
```

```
    Vector(int x, int y) : x(x), y(y) {}
```

```
    // Overloading the '+' operator using a member function
```

```
    Vector operator+(const Vector& other) {
```

```
        return Vector(this->x + other.x, this->y + other.y);
```

```
    }
```

```
Private:
```

```
    int x, y;
```

```
};
```

Operator Overloading with Friend Functions

- Another method for **overloading operators with non-member functions designated as friends inside the class** is friend functions. Although friend functions are **not class members, they can access the class's private members.**
- The **general syntax** for utilizing a friend function to overload an operator is as follows:
- `friend return_type operator op(parameters);`
- The function is declared a **'friend'** of the class with this syntax, enabling it to access its private members. The remaining portions of the grammar resemble member function overloading.

Syntax to use Friend Function in C++ to Overload Operators:

```
friend return_type operator operator symbol(parameters)
{
//Statements;
}
```

Benefits of Overloading Operators with Friend Functions

Symmetry: When overloading **binary operators**, you can achieve **symmetric behaviour using friend functions**. With member functions, you can **only access the data of the currently selected object**, which might **occasionally result in asymmetry**. Since friend functions can **access both operands' private members**, the **behaviour is guaranteed to be symmetric**.

Non-Member Function: Friend functions can overload operators for user-defined types **without altering the class declaration** because they are **not class members**. This is helpful when you need help editing the original class.

Global Functions: Friend functions are **global** because they are **defined outside the class**. This promotes **code reuse** because they can be **used for multiple classes or even different projects**.

Enhanced Readability: When dealing with **complex expressions** involving several objects and operators, **friend functions** sometimes produce more legible code.

Flexibility: When choosing which operators to overload, **friend functions offer flexibility**. Without being constrained by the **member functions of the class**, you can **decide to overload only the operators** that make sense for your class.

Example to Understand Operator Overloading in C++ Using Friend Function

```
#include <iostream>
using namespace std;
class Complex
{
private:
int real;
int img;
public:
Complex (int r = 0, int i = 0)
{
real = r;
img = i;
}
void Display ()
{
cout << real << "+i" << img;
}
```

```
friend Complex operator + (Complex c1, Complex c2);
};
Complex operator + (Complex c1, Complex c2)
{
Complex temp;
temp.real = c1.real + c2.real;
temp.img = c1.img + c2.img;
return temp;
}
int main ()
{
Complex C1(5, 3), C2(10, 5), C3;
C1.Display();
cout << " + ";
C2.Display();
cout << " = ";
C3 = C1 + C2;
C3.Display();
}
```

OUTPUT:
 $5+i3 + 10+i5 = 15+i8$

Example:

// C++ program to show operator overloading using a Friend Function

```
#include <iostream>
```

```
using namespace std;
```

```
class Distance {
```

```
int feet, inch;
```

```
public:  Distance(): feet(0), inch(0) { }
```

```
Distance(int f, int i)  {
```

```
this->feet = f;
```

```
this->inch = i;  }
```

```
    // getter functions
```

```
int getFeet() { return this->feet; }
```

```
int getInch() { return this->inch; }
```

```
// Declaring friend function using friend keyword
```

```
friend Distance operator+(Distance& d1, Distance& d2);
```

```
};
```



```

// Implementing friend function with two parameters Call by reference
Distance operator+(Distance& d1, Distance& d2)
{
    // Create an object to return
    Distance d3;
    d3.feet = d1.feet + d2.feet;
    d3.inch = d1.inch + d2.inch;
    // Return the resulting object
    return d3;}
int main(){
    Distance d1(8, 9);
    Distance d2(10, 2);
    Distance d3;
    // Use overloaded operator
    d3 = d1 + d2;
    cout << "\nTotal Feet & Inches: " << d3.getFeet() << "''" << d3.getInch();
    return 0;}

```

OUTPUT:

Total Feet & Inches: 18'11

Use Cases for Friend Functions to Reduce Operator Overload

Operator overloading with friend functions is **more appropriate in some situations**, even if Operator overloading with member functions is frequently the better option for simple classes:

Mathematical Operations: Operator overloading with buddy functions can **result in more symmetrical and natural behaviour** when dealing with mathematical classes like complex numbers, matrices, or vectors.

Overloading External Types: Because you **cannot change the original class definitions**, buddy functions are required when you need to overload operators for built-in or external types (like 'int', 'double', or 'std::string') to operate with your custom classes.

Cross-Class actions: You can use friend functions to **execute actions between objects of different classes without breaking encapsulation.**

Operators that cannot be overloaded using friend functions

We have seen operators can be **overloaded using member function as well as friend function**. However there are **some operators** which can be **overloaded as member function but cannot be overloaded as friend function**. List of operators which **cannot be overloaded using friend function** is given below:

1. Assignment operator =
2. Function call operator ()
3. Array subscript operator []
4. Access to class member using pointer to object operator ->

Overloading New and Delete operator

- The **new** operator is used to **dynamically allocate memory** on the heap for an object or an array of objects. And **delete** operator is used to **deallocate the memory**.
- The new and delete operators can also be **overloaded like other operators in C++**. New and Delete operators can be **overloaded globally** or they can be **overloaded for specific classes**.
- If these operators are **overloaded using member function** for a class, it means that these operators are overloaded **only for that specific class**.
- If overloading is **done outside a class** (i.e. it is not a member function of a class), the **overloaded 'new' and 'delete' will be called anytime** you make use of these operators (within classes or outside classes). This is **global overloading**.

NEW Operator

- The “new” operator in C++ is used to **allocate memory dynamically** for a **variable or an object at runtime**. This means that the memory is allocated during the execution of the program, as opposed to being **allocated at compile time**.
- When the “new” operator is called, it **reserves a block of memory** that is **large enough to hold the object being created** and then returns a pointer to the first byte of that memory block.

The syntax for overloading the new operator

1. `Pointer_name=new datatype; Ex:int *ptr=new int;`
2. `pointer_variable = new datatype(value); Ex:int *ptr=new int(10);`
3. `pointer_variable = new datatype[size]; Ex:int *ptr=new int[];`
4. `void* operator new(size_t size);`

Syntax for overloading the delete operator

void operator delete(void*);

- The function receives a parameter of type **void*** which has to be **deleted**. Function should **not return anything**.
- Both overloaded new and delete operator functions are **static members** by default. Therefore, they **don't have access to this pointer** .

Overloading new and delete operator for a specific class:

```
#include<iostream>
#include<stdlib.h>
using namespace std;
class student
{
    string name;
    int age;
public:
    student()
    {
        cout<< "Constructor is called\n" ;
    }
    student(string name, int age)
    {
        this->name = name;
        this->age = age;
    }
}
```

```
void display()
{
    cout<< "Name:" << name << endl;
    cout<< "Age:" << age << endl;
}
void * operator new(size_t size)
{
    cout<< "Overloading new operator with size: " << size << endl;
    void * p = ::operator new(size);
    //void * p = malloc(size); will also work fine
    return p;
}
void operator delete(void * p)
{
    cout<< "Overloading delete operator " << endl;
    free(p);
}
};
```



```
int main()
{
    student * p = new student("Yash", 24);
    p->display();
    delete p;
}
```

OUTPUT:

Overloading new operator with size: 40

Name: Yash

Age:24

Overloading delete operator

Global overloading of new and delete operator

```
#include<iostream>
#include<stdlib.h>
using namespace std;
void * operator new(size_t size)
{
    cout << "New operator overloading " << endl;
    void * p = malloc(size);
    return p;
}
void operator delete(void * p)
{
    cout << "Delete operator overloading " << endl;
    free(p);
}
```

```
int main()
{
    int n = 5, i;
    int * p = new int[n];
    for (i = 0; i<n; i++)
        p[i]= i;
    cout << "Array: ";
    for(i = 0; i<n; i++)
        cout << p[i] << " ";
    cout << endl;
    delete [] p;
}
```

Output

New operator overloading
Array: 0 1 2 3 4
Delete operator overloading

Overloading special operators

- C++ is a **strong and flexible programming language** that provides a **large range of operators to modify data and carry out various operations**. Among these operators are the so-called "*special operators*", which are special in their **functionality and necessary for more complex programming jobs**.

Some of the special operators in C++ are as follows:

- `new` – It is used to **allocate the memory dynamically**.
- `Delete` – It is used to **free the memory** dynamically.
- `[]` – It is a subscript operator.
- `->` – It is a member access operators.
- `=` – It is used to assign the values.
- `()` – It is used for function call.
- The operators other than listed above can be **overloaded either as a member or as non-members**. But in general, non-member overloading is recommended.

What Are Special Operators?

- The *special operator* is a subset of the more prevalent arithmetic, assignment, and relational operators in C++. Special operators are used for a variety of applications. They give programmers the means to carry out tasks that **ordinary operators might find challenging**. The **special operators are like**;
 - Ternary conditional operator (?:),
 - The comma operator (,),
 - The scope resolution operator (::),
 - The sizeof operator (sizeof),
 - The pointer-to-member operator (. * and -> *)
 - The member selection operator (. and ->)

1. The Ternary Conditional Operator (? :)

- The "*conditional operator*" or "*ternary operator*", sometimes known as the ternary conditional operator. It is mainly utilized in conditional statements. It enables you to **write condensed conditional statements by assessing a condition and returning one of two values depending on whether the condition is true or false.**

Syntax:

(condition) ? value_if_true : value_if_false;

```
#include <iostream>
using namespace std;
int main() {
    int x = 10, y = 20;
    int max_num = (x > y) ? x : y;
    cout << "The maximum number is: " << max_num << endl;
    return 0;
}
```

O/P:The maximum number is: 20

The Comma Operator (,)

In C++, you can **evaluate several expressions** that are separated by commas from *left* to *right* by using the **comma operator**. It gives back the **rightmost expression's value**.

Syntax: `expr1, expr2, expr3, ..., exprN`


```
#include <iostream>
using namespace std;
int main() {
    int x = 5, y = 10, z = 15;
    int result = (x++, y++, x + y + z);
    cout << "Result: " << result << endl;
    return 0;
}
```

O/P: Result: 30

The Scope Resolution Operator (::):

When *local* and *global variables* conflict, the C++ scope resolution operator is **used to access members of a class or namespace** or to access static members of a class.

Syntax:

```
class_name::member_name;
```

The Sizeof Operator (sizeof):

In C++, the *sizeof operator* is used to calculate an *object's* or data type's size in bytes. When working with arrays or dynamically allocated memory, it is extremely helpful.

Syntax:

```
sizeof(data_type);
```

```
sizeof(object);
```

The Pointer-to-Member Operators (. * and -> *)

In C++, you can access members of a class via *pointers* to *objects* or pointers to member functions by using the *pointer-to-member* operators.

Syntax for . operator:

- It has the following syntax:
- object.*member_pointer;

Syntax for -> operator:

- It has the following syntax:
- pointer_to_object->*member_pointer;

The Member Selection Operators (. and ->)

- If you want to access members of a class or structure, use the *member selection operators* in C++, which include the *dot. operator* and the *arrow -> operator*.

Syntax:

- It has the following syntax:
- object.member;
- pointer_to_object->member;

UNIT -III

INHERITANCE AND POLYMORPHISM

Inheritance

- The capability of a **class to derive properties and characteristics from another class** is called **Inheritance**.
- Inheritance is **one of the most important features** of Object Oriented Programming in C++.

Syntax of Inheritance in C++

```
class derived_class_name : access-specifier base_class_name  
    {  
    // body ....  
    };
```

class: keyword to create a new class

derived_class_name: name of the new class, which will **inherit the base class**

access-specifier: Specifies the access mode which can be **either of private, public or protected**. If neither is specified, **private is taken as default**.

base-class-name: name of the base class.

Example:

```
// Base class
class Vehicle {
public:
    string brand = "Ford";
    void honk() {
        cout << "Tuut, tuut! \n" ;
    } };
// Derived class
class Car: public Vehicle {
public:
    string model = "Mustang";
};
int main() {
    Car myCar;
    myCar.honk();
    cout << myCar.brand + " " + myCar.model;
    return 0;
}
```

Output:

Tuut, tuut!

Ford Mustang

Modes of Inheritance in C++

- Mode of inheritance controls the **access level** of the **inherited members of the base class** in the derived class. In C++, there are **3 modes of inheritance**:
 1. Public Mode
 2. Protected Mode
 3. Private Mode

1. Public Mode: If we derive a subclass 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 the derived class.

Ex: class ABC : public XYZ {...}

2. Protected mode: If we derive a subclass from a Protected base class. Then **both public members and protected members of the base class** will become protected in the **derived class**.

Ex: class ABC : protected XYZ {...}

3. Private mode: If we derive a subclass from a Private base class. Then **both public members and protected members of the base class will become private** in the derived class. They can only be accessed by the member functions of the derived class.

```
class ABC : private XYZ {...}           // private derivation
class ABC: XYZ {...}                   // private derivation by default
```

Types Of Inheritance in C++

- The inheritance can be classified on the basis of the relationship between the derived class and the base class. In C++, we have **5 types of inheritances**:
 1. Single inheritance
 2. Multilevel inheritance
 3. Multiple inheritance
 4. Hierarchical inheritance
 5. Hybrid inheritance

1. Single Inheritance

- In single inheritance, a class is allowed to inherit from **only one class**. i.e. one base class is inherited by **one derived class** only.

Syntax

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

Ex:

```
class A
{
    ... ..
};
class B: public A
{
    ... ..
};
```

2. Multiple Inheritance

- Multiple Inheritance is a feature of C++ where a class can inherit from **more than one class**. i.e one **subclass** is inherited from more than one **base class**.

Syntax

```
class subclass_name : access_mode base_class1, access_mode base_class2, ....  
    {  
    // body of subclass  
    };
```

Ex:

```
class B  
    { ... ..  
    };  
class C  
    {  
... ..  
    };  
class A: public B, public C  
    {  
... ..  
    };
```

3. Multilevel Inheritance

- In this type of inheritance, a **derived class is created from another derived class** and that **derived class can be derived from a base class** or any **other derived class**. There can be any number of levels.

Syntax

```
class derived_class1: access_specifier base_class
{... .. }
class derived_class2: access_specifier derived_class1
{... .. }.....
```

```
Ex:class C
{ ... .. };
class B : public C
{... .. };
class A: public B
{... .. };
```

4. Hierarchical Inheritance

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

Syntax

```
class derived_class1: access_specifier base_class
{
    ... ..
}
class derived_class2: access_specifier base_class
{
    ... ..
}
```

Example:

```
class A
{
    // body of the class A.
}
class B : public A
{
    // body of class B.
}
class C : public A
{
    // body of class C.
}
class D : public A
{
    // body of class D.
}
```

5. Hybrid Inheritance

- Hybrid Inheritance is implemented by **combining more than one type of inheritance**. For example: Combining Hierarchical inheritance and Multiple Inheritance will create hybrid inheritance in C++
- There is **no particular syntax of hybrid inheritance**. We can just combine two of the above inheritance types.

Base-Class Access Control

public – members are accessible from outside the class, and members can be accessed from anywhere.

private – members cannot be accessed (or viewed) from outside the class, i.e members are private to that class only.

protected – members cannot be accessed from outside the class, but, they can be accessed in inherited classes or derived classes.

Accessibility Of Inheritance Access:

Accessibility	Public Members	Protected Members	Private Members
Base Class	Yes	Yes	Yes
Derived Class	Yes	Yes	No

Inheritance Access

public Inheritance

```
// C++ program to demonstrate the working of public inheritance
#include <iostream>
using namespace std;
class Base {
private:
    int pvt = 1;
protected:
    int prot = 2;
public:
    int pub = 3;
    // function to access private member
    int getPVT() { return pvt; }
};
```

```
class PublicDerived : public Base {
public:
    // function to access protected member from Base
    int getProt() { return prot; }
};

int main()
{
    PublicDerived object1;
    cout << "Private = " << object1.getPVT() << endl;
    cout << "Protected = " << object1.getProt() << endl;
    cout << "Public = " << object1.pub << endl;
    return 0;
}
```

Output:

Private = 1

Protected = 2

Public = 3

Protected Inheritance

```
// C++ program to demonstrate the working of protected inheritance
#include <iostream>
using namespace std;
class Base {
private:
    int pvt = 1;
protected:
    int prot = 2;
public:
    int pub = 3;
    // function to access private member
    int getPVT() { return pvt; }
};
class ProtectedDerived : protected Base {
public:
    // function to access protected member from Base
    int getProt() { return prot; }
```

```
// function to access public member from Base
int getPub() { return pub; }

// function to get access to private members from Base
int try_getPVT() {Base::getPVT(); }
};
int main()
{
    ProtectedDerived object1;
    cout << "Private = " << object1.try_getPVT() << endl;
    cout << "Protected = " << object1.getProt() << endl;
    cout << "Public = " << object1.getPub() << endl;
    return 0;
}
```

Output:

Private = 1

Protected = 2

Public = 3

Private Inheritance

```
// C++ program to demonstrate the working of private inheritance
#include <iostream>
using namespace std;
class Base {
private:
    int pvt = 1;
protected:
    int prot = 2;
public:
    int pub = 3;
    // function to access private member
    int getPVT() { return pvt; }
};
class PrivateDerived : private Base {
public:
    // function to access protected member from Base
    int getProt() { return prot; }
```

```
// function to access public member
int getPub() { return pub; }
// function to get access to private members from Base
int try_getPVT() { Base::getPVT(); }
};
int main()
{
    PrivateDerived object1;
    cout << "Private = " << object1.try_getPVT() << endl;
    cout << "Protected = " << object1.getProt() << endl;
    cout << "Public = " << object1.getPub() << endl;
    return 0;
}
```

Output:

Private = 1

Protected = 2

Public = 3

Inheritance and Protected Members

- **private member of a base class cannot be inherited** and therefore its is **not available for the derived class directly**.
- C++ provides a **third visibility modifier, protected** which serves a limited purpose in inheritance. A member declared as **protected** is **accessible by the member functions within its class and any class immediately** derived from it.
- A class can now use all the three visibility modes as illustrated below:

```
class alpha {  
    private:  
    //optional ...  
    //visible to the member functions within its class  
    protected:  
    ... //visible to the member functions of its own and derived class  
    public:  
    ... //visible to all functions in the program  
};
```


- When a **protected member** is inherited in **public mode**, it becomes **protected in the derived class** too, and therefore is **accessible by the member functions of the derived class**. It is also ready for the further inheritance.
- A **protected member**, inherited in the **private mode** derivation, becomes **private in the derived class**. Although, it is **available to the member functions of the derived class**, it is **not available for further inheritance** (since private members cannot be inherited).
- The keywords **private**, **public** and **protected** may appear in any order and in any number of times in the declaration of a class.

For example:

```
class beta {  
protected: ...  
public: ...  
    private: ...  
public: ... }
```

A special mention of Protected Inheritance:

- In previous posts, we mentioned about **defining of derived class by using keywords private and/or public as visibility mode.**
- In addition to that, we can also use the keyword **protected** as visibility mode in the **defining of the derived class.**
- Such inheritance using the visibility mode **protected** is known as **protected inheritance.**
- In **protected inheritance**, **public members of base class** become **protected in derived class** as also the **protected members of the base class** become **protected** in the derived class.
- However, as usual, **private members of a base class** cannot be **inherited.**

Example:

```
class ABC :: protected XYZ //protected derivation
{ members of ABC
};
```

Accessibility in protected Inheritance

Accessibility	private members	protected members	public members
Base Class	Yes	Yes	Yes
Derived Class	No	Yes	Yes (inherited as protected variables)

// C++ program to demonstrate the working of protected inheritance

```
#include <iostream>
```

```
using namespace std;
```

```
class Base {
```

```
    private:
```

```
        int pvt = 1;
```

```
    protected:
```

```
        int prot = 2;
```

```
    public:
```

```
        int pub = 3;
```

```
        // function to access private member
```

```
        int getPVT() {
```

```
            return pvt;
```

```
        } };
```

```
class ProtectedDerived : protected Base {
```

```
    public:
```

```
        // function to access protected member from Base
```

```
        int getProt() {
```

```
            return prot; }
```

```
        // function to access public member from Base
```

```
        int getPub() {
```

```
            return pub;
```

```
        } };
```

```
int main() {  
    ProtectedDerived object1;  
    cout << "Private cannot be accessed." << endl;  
    cout << "Protected = " << object1.getProt() << endl;  
    cout << "Public = " << object1.getPub() << endl;  
    return 0;  
}
```

Output

Private cannot be accessed. Protected = 2

Public = 3

Inheriting Multiple Base Classes

- **Multiple Inheritance** is a feature of C++ where a class **can inherit from more than one classes**. The constructors of **inherited classes are called in the same order** in which they are inherited.
- A class can be derived from more than one base class. In a multiple-inheritance model (where classes are derived from more than one base class), the base classes are specified using the *base-list* grammar element.

Example:

- (i) A CHILD class is derived from FATHER and MOTHER class
- (ii) A PETROL class is derived from LIQUID and FUEL class.

Syntax:

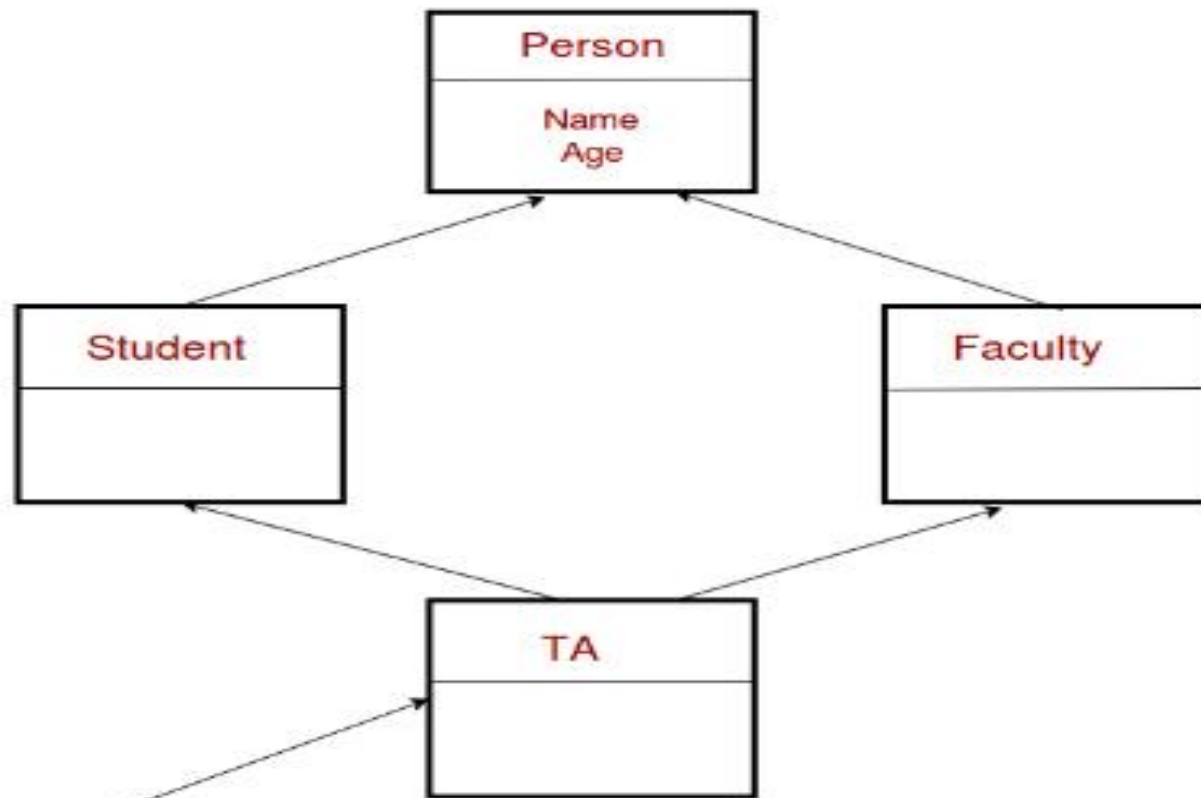
```
class A
{ ... .. }
};
class B
{ ... .. }
};
class C: public A,public B
{ ... .. }
};
```

```
#include<iostream>
using namespace std;
class A
{
public:
A() { cout << "A's constructor called" << endl; }
};
class B
{
public:
B() { cout << "B's constructor called" << endl; }
};
class C: public B, public A // Note the order
{
public:
C() { cout << "C's constructor called" << endl; }
};
int main()
{
    C c;
    return 0;
}
```

Output:

B's constructor called
A's constructor called
C's constructor called

- A class can be **derived from more than one base class**. In a multiple-inheritance model (where classes are derived from more than one base class), the base classes are specified using the *base-list* grammar element.



Name and Age needed only once


```
#include<iostream>
using namespace std;
class Person {
// Data members of person
public:
Person(int x) { cout << "Person::Person(int ) called" << endl; }
};
class Faculty : public Person {
// data members of Faculty
public:
    Faculty(int x):Person(x) {
        cout<<"Faculty::Faculty(int ) called"<< endl;
    } };
class Student : public Person {
// data members of Student
public:
    Student(int x):Person(x) {
        cout<<"Student::Student(int ) called"<< endl;
    } };
class TA : public Faculty, public Student {
public:
    TA(int x):Student(x), Faculty(x) {
        cout<<"TA::TA(int ) called"<< endl;
    } };
int main() {
    TA ta1(30);
}
```

Output:

Person::Person(int) called

Faculty::Faculty(int) called

Person::Person(int) called

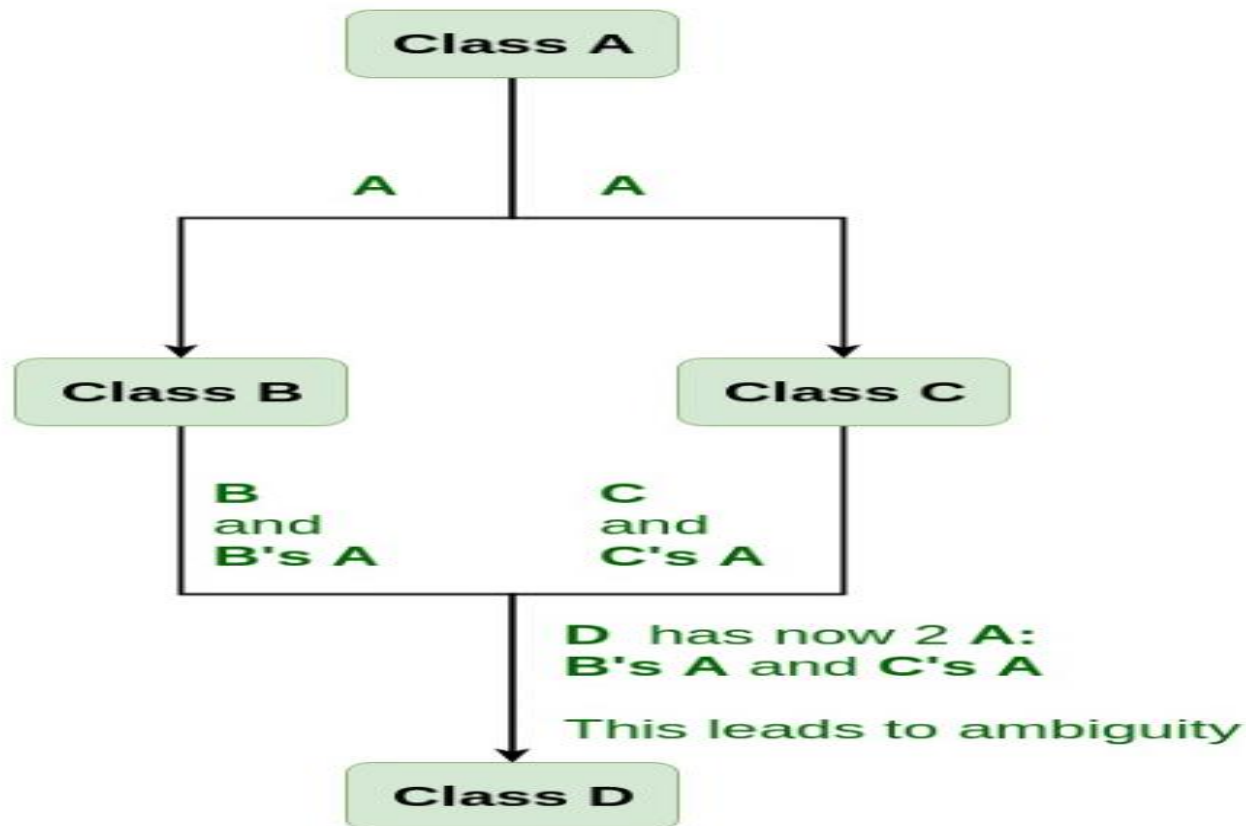
Student::Student(int) called

TA::TA(int) called

Virtual base classes

- Virtual base classes are **used in virtual inheritance** in a way of **preventing multiple “instances”** of a given class appearing in an inheritance hierarchy when using multiple inheritances.
- class can be an **indirect base class** to a **derived class more than once**, C++ provides a way to optimize the way such base classes work. **Virtual base classes offer a way to save space and avoid ambiguities in class hierarchies that use multiple inheritance.**
- Each **nonvirtual object** contains a **copy of the data members defined in the base class.**
- This **duplication wastes space** and requires you to **specify** which **copy of the base class members** you want whenever you access them.
- When a base class is specified as a virtual base, it can act as an **indirect base more than once without duplication** of its data members.
- A **single copy of its data members** is **shared by all the base classes** that use it as a virtual base.
- When declaring a **virtual base class**, the **virtual** keyword appears in the base lists of the derived classes.

- **Need for Virtual Base Classes:** Consider the situation where we have one class **A** . This class **A** is inherited by two other classes **B** and **C**. Both these class are inherited into another in a new class **D** as shown in figure below.



Syntax for Virtual Base Classes:

Syntax 1:

```
class B : virtual public A
{
};
```

Syntax 2:

```
class C : public virtual A
{
};
```

```
#include <iostream>
using namespace std;
class A {
public:
    int a;
    A() // constructor
    {
        a = 10;
    }
};
class B : public virtual A {
};
class C : public virtual A {
};
class D : public B, public C {
};
int main()
{
    D object; // object creation of class d
    cout << "a = " << object.a << endl;
    return 0;
}
```

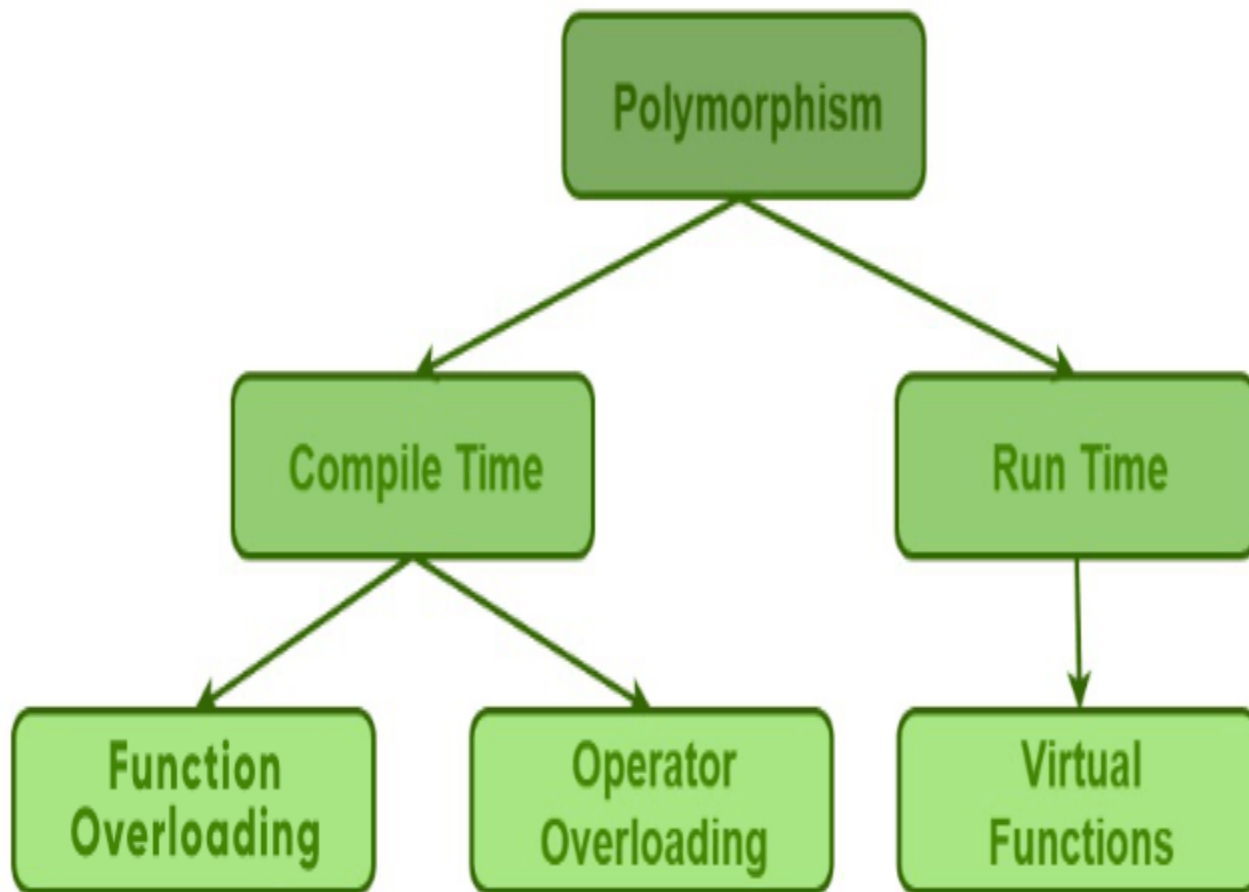
Output:
a = 10

Polymorphism

- The word “**polymorphism**” means having **many forms**. In simple words, we can define polymorphism as the **ability of a message to be displayed in more than one form**.
- A real-life example of polymorphism is a **person who at the same time can have different characteristics**. A man at the **same time is a father, a husband, and an employee**. So the same person exhibits **different behavior in different situations**. This is called polymorphism.
- Polymorphism is considered one of the important features of Object-Oriented Programming.

Types of Polymorphism

1. Compile-time Polymorphism
2. Runtime Polymorphism



1. Compile-Time Polymorphism

- This type of polymorphism is achieved by **function overloading or operator overloading.**

A. Function Overloading

- When there are **multiple functions with the same name but different parameters**, then the functions are said to be **overloaded**, hence this is known as **Function Overloading.**
- Functions can be overloaded by **changing the number of arguments** or/and **changing the type of arguments.**
- In simple terms, it is a feature of object-oriented programming providing many functions that have the same name but distinct parameters when numerous tasks are listed under one function name.
- There are certain Rules of Function Overloading that should be followed while overloading a function.

// C++ program to demonstrate function overloading or Compile-time Polymorphism

```
#include <bits/stdc++.h>
using namespace std;
class Geeks {
public: // Function with 1 int parameter
void func(int x) {
cout << "value of x is " << x << endl; }
// Function with same name but // 1 double parameter
void func(double x) {
cout << "value of x is " << x << endl; }
// Function with same name and // 2 int parameters
void func(int x, int y)
{ cout << "value of x and y is " << x << ", " << y << endl;
};
int main(){
Geeks obj1;
obj1.func(7); // Function being called depends on the parameters passed func() is called with int value
obj1.func(9.132); // func() is called with double value
obj1.func(85, 64); // func() is called with 2 int values
return 0;
}
```

Output:

value of x is 7

value of x is 9.132

value of x and y is 85, 64

B. Operator Overloading

- C++ has the ability to provide the **operators with a special meaning for a data type**, this ability is known as operator overloading.
- For example, we can make use of the **addition operator (+) for string class to concatenate two strings**.
- We know that the task of this operator is to **add two operands**.
- So a **single operator ‘+’**, when placed **between integer operands, adds them** and when **placed between string operands, concatenates them**.

// C++ program to demonstrate Operator Overloading or Compile-Time Polymorphism

```
#include <iostream>
using namespace std;
class Complex {
private:
int real, imag;
public:  Complex(int r = 0, int i = 0)
    {    real = r;    imag = i;  }
    // This is automatically called when '+' is used with between two Complex objects
Complex operator+(Complex const& obj)  {
Complex res;
res.real = real + obj.real;
res.imag = imag + obj.imag;
return res;  }
void print()
{ cout << real << " + i" << imag << endl; };
int main(){
    Complex c1(10, 5), c2(2, 4);
// An example call to "operator+"
Complex c3 = c1 + c2;
c3.print();}
```

Output:

12 + i9

2. Runtime Polymorphism

- This type of polymorphism is achieved by **Function Overriding**.
- **Late binding** and **dynamic polymorphism** are other names for **runtime polymorphism**.
- The **function call is resolved at runtime** in runtime polymorphism.
- In contrast, with compile time polymorphism, the **compiler determines which function call to bind to the object** after deducing it at runtime.

Virtual Functions

- A **virtual function** is a *member function* that is **declared in the *base class*** using the keyword **virtual** and is re-defined (Overridden) in the *derived* class.
- It tells the **compiler to perform late binding** where the **compiler matches the object with the right called function and executes it** during the **runtime**. This technique falls under Runtime Polymorphism.
- The idea is that **virtual functions** are called according to the **type of the object instance pointed to or referenced**, not according to the type of the **pointer or reference**. In other words, virtual functions are resolved late, at runtime.

- **Virtual functions** ensure that the **correct function is called for an object**, regardless of the type of reference (or pointer) used for the function call.
- They are mainly used to achieve Runtime polymorphism.
- Functions are declared with a **virtual** keyword in a base class.
- The resolving of a function call is done at runtime.

Rules for Virtual Functions

The rules for the virtual functions in C++ are as follows:

- Virtual functions **cannot be static**.
- A virtual function can be a **friend function of another class**.
- Virtual functions should be **accessed using a pointer or reference of base class type** to achieve runtime polymorphism.
- The **prototype of virtual functions** should be **the same in the base as well as the derived class**.
- They are **always defined in the base class and overridden in a derived class**. It is **not mandatory for the derived class to override** (or re-define the virtual function), in that case, the base class version of the function is used.
- A class may have a **virtual destructor but it cannot have a virtual constructor**.
- The function call resolving is done at run-time

Limitations of Virtual Functions

- **Slower:** The function call takes slightly longer due to the virtual mechanism and makes it more difficult for the compiler to optimize because it does not know exactly which function is going to be called at compile time.
- **Difficult to Debug:** In a complex system, virtual functions can make it a little more difficult to figure out where a function is being called from.

Compile-Time VS Runtime Behavior of Virtual Functions in C++

A virtual function in C++ exhibits two different types of behavior: compile-time and runtime.

	Compile-time behavior	Runtime behavior
Alternative name	Early binding	Late binding
How is it achieved	The type of pointer	Depending on the location where the pointer is pointing

// C++ program to demonstrate how we will calculate the area of shapes USING VIRTUAL FUNCTION

```
#include <fstream>
```

```
#include <iostream>
```

```
using namespace std;
```

```
// Declaration of Base class
```

```
class Shape {
```

```
public:
```

```
    // Usage of virtual constructor
```

```
    virtual void calculate()
```

```
    {
```

```
        cout << "Area of your Shape ";
```

```
    }
```

```
    // usage of virtual Destuctor to avoid memory leak
```

```
    virtual ~Shape()
```

```
    {
```

```
        cout << "Shape Destuctor Call\n";
```

```
    }
```

```
};
```

```
// Declaration of Derived class
```

```
class Rectangle : public Shape {
```

```
public:
```

```
    int width, height, area;
```

```
void calculate()
{
    cout << "Enter Width of Rectangle: ";
    cin >> width;
    cout << "Enter Height of Rectangle: ";
    cin >> height;
    area = height * width;
    cout << "Area of Rectangle: " << area << "\n";
}
// Virtual Destuctor for every Derived class
virtual ~Rectangle()
{
    cout << "Rectangle Destuctor Call\n";
}
};
```

```

// Declaration of 2nd derived class
class Square : public Shape {
public:
int side, area;
void calculate() {
cout << "Enter one side your of Square: ";
cin >> side;
area = side * side;
cout << "Area of Square: " << area << "\n"; }
// Virtual Destuctor for every Derived class
virtual ~Square(){
cout << "Square Destuctor Call\n";
}};

int main()
{
// base class pointer
Shape* S;
Rectangle r;
// initialization of reference variable
S = &r;
// calling of Rectangle function
S->calculate();
Square sq;
// initialization of reference variable
S = &sq;
// calling of Square function
S->calculate();
// return 0 to tell the program executed
// successfully
return 0; }

```

Output

Enter Width of Rectangle: 10

Enter Height of Rectangle: 20

Area of Rectangle: 200

Enter one side your of Square: 16 Area of
Square: 256

Pure Virtual Functions

- A pure virtual function (or abstract function) in C++ is a **virtual function** for which we can have an **implementation**, **But** we must **override that function in the derived class**, **otherwise, the derived class will also become an abstract class.**
- A pure virtual function is **declared by assigning 0** in the **declaration.**
- An **abstract class is a class** that is **designed** to be specifically **used as a base class.**
- An **abstract class contains at least one pure virtual function.** You declare a pure virtual function by **using a pure specifier (= 0) in the declaration of a virtual member function in the class declaration.**

Characteristics of a pure virtual function

- A pure virtual function is a **"do nothing"** function. Here **"do nothing"** means that it just **provides the template, and derived class implements the function.**
- It can be considered as an **empty function means that the pure virtual function** does not have any definition relative to the base class.
- Programmers need to **redefine the pure virtual function in the derived class** as it has **no definition in the base class.**
- A class having pure virtual function **cannot be used to create direct objects of its own.** It means that the class is **containing any pure virtual function** then we **cannot create the object of that class.** This type of class is known as an **abstract class.**

Example of Pure Virtual Functions

```
// An abstract class
class Test {
    // Data members of class
public:
    // Pure Virtual Function
    virtual void show() = 0;

    /* Other members */
};
```

// C++ Program to illustrate the abstract class and virtual functions

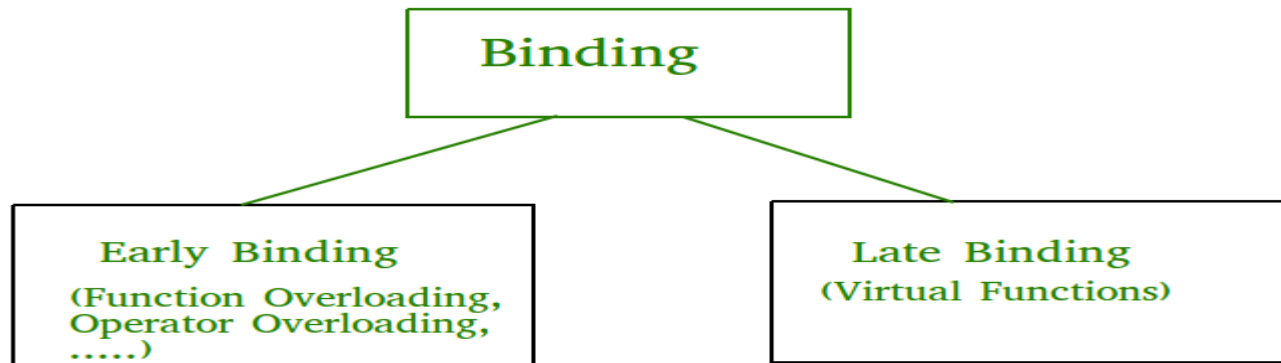
```
#include <iostream>
using namespace std;
class Base {
    // private member variable
    int x;
public:
    // pure virtual function
    virtual void fun() = 0;
    // getter function to access x
    int getX() { return x; } };
// This class inherits from Base and implements fun()
class Derived : public Base {
    // private member variable
    int y;
public:
    // implementation of the pure virtual function
    void fun() { cout << "fun() called"; }
};
int main(void)
{
    // creating an object of Derived class
    Derived d;
    // calling the fun() function of Derived class
    d.fun();
    return 0;
}
```

OUTPUT:
fun() called

Virtual function	Pure virtual function
A virtual function is a member function in a base class that can be redefined in a derived class.	A pure virtual function is a member function in a base class whose declaration is provided in a base class and implemented in a derived class.
The classes which are containing virtual functions are not abstract classes.	The classes which are containing pure virtual function are the abstract classes.
In case of a virtual function, definition of a function is provided in the base class.	In case of a pure virtual function, definition of a function is not provided in the base class.
The base class that contains a virtual function can be instantiated.	The base class that contains a pure virtual function becomes an abstract class, and that cannot be instantiated.
If the derived class will not redefine the virtual function of the base class, then there will be no effect on the compilation.	If the derived class does not define the pure virtual function; it will not throw any error but the derived class becomes an abstract class.
All the derived classes may or may not redefine the virtual function.	All the derived classes must define the pure virtual function.

Early binding and Late binding

- **Binding** refers to the **process of converting identifiers** (such as variable and performance names) into **addresses**. Binding is **done for each variable and functions**.
- For functions, it means that **matching the call with the right function definition** by the compiler. It takes place **either at compile time or at runtime**.
- While **early binding** provides **compile-time checking** of all types so that **no implicit casts** occur, **late binding checks types only when the object is created** or an **action is performed** on the type.



- 1. Early Binding (compile-time time polymorphism)** As the name indicates, **compiler (or linker) directly associate an address to the function call.** It replaces the call with a **machine language instruction** that tells the **mainframe to leap to the address of the function.** For **function overloading** it is an **example** of early binding.
- 2. Late Binding : (Run time polymorphism)** In this, the **compiler adds code that identifies the kind of object at runtime** then **matches** the call with the **right function definition** (Refer this for details). This can be achieved by declaring a **virtual function.**

Example: Early binding

```
#include<iostream>
using namespace std;
class Base {
public: void display() {
    cout<<" In Base class" <<endl;
}};
class Derived: public Base {
public: void display() {
    cout<<"In Derived class" << endl;
}};
int main(void) {
    Base *base_pointer = new Derived;
    base_pointer->display();
    return 0;}
```

OUTPUT:

In Base class

Example: Late binding

```
#include<iostream>
using namespace std;
class Base {
public:  virtual void display() {
    cout<<"In Base class" << endl;
} };
class Derived: public Base {
public:  void display() {
    cout<<"In Derived class" <<endl;
} }; int main() {
    Base *base_pointer = new Derived;
    base_pointer->display();
    return 0; }
```

OUTPUT:

In Derived class

Sr. No.	Early Binding	Late Binding
1.	Early binding is the process of linking a function with an object during the compilation process.	Late binding is a run-time polymorphism with method overriding.
2.	Static binding is another name for early binding.	Dynamic binding is another name for late binding.
3.	Early binding happens at compile time.	Late binding happens at run time.
4.	Execution speed is faster in early binding.	Execution speed is lower in late binding.
5.	The class information is used by Early binding to resolve method calls.	The object is used by Late binding to resolve method calls.

Run-time type information

- Runtime type information (RTTI), also known as **runtime type identification** (RTI), is a feature of **several programming languages** (such as C++, Object Pascal, and Ada) that **makes data about an object's data type available at runtime**.
- RTTI is used to **find out the type of an object at runtime**. This is a **very handy tool for making sure the object type is exactly learned before any assignment is made**. We have already seen the **consequences of assigning a base class object to derived and a derived class object to the base**.
- It is possible for **runtime type information** to be made available to **all kinds or just to those types that have it explicitly** (as is the case with Ada). A **more broad idea called type introspection is specialized into runtime type knowledge**.
- **RTTI (Run-time type information)** is a mechanism that **exposes information about an object's data type at runtime and is available only for the classes which have at least one virtual function**.
- It allows the **type of an object to be determined during program execution**.

Runtime Casts

- The runtime cast, which **checks that the cast is valid**, is the **simplest approach to ascertain the runtime type of an object using a pointer or reference**.
- This is especially beneficial when we **need to cast a pointer from a base class to a derived type**.
- When dealing with the inheritance hierarchy of classes, the **casting of an object is usually required**.
- There are two types of casting:
 1. **Upcasting:** When a **pointer or a reference of a derived class object is treated as a base class pointer**.
 2. **Downcasting:** When a **base class pointer or reference is converted to a derived class pointer**.

What is C++ typeid?

- An **object's class** can be **ascertained at runtime** using the **typeid keyword**. Upon completion of the program, it returns a pointer to the `std::type_info` object.
- In **non-polymorphic contexts** where only the **class information is required**, `typeid` is frequently favoured over **dynamic cast** `class type>` because **typeid is always a constant-time process**.
- In contrast, a **dynamic cast may need to navigate the class derivation lattice of its input at runtime**. `std::type_info::name()`, for example, is implementation-defined and cannot be trusted to remain consistent between compilers.
- When the **unary * operator is used on a null pointer to create the typeid expression**, objects of class `std::bad_typeid` are thrown.
- **Implementation-specific factors determine whether exceptions are raised for additional invalid reference arguments**.
- To put it another way, the **expression must have the form typeid(*p)**, where **p is any expression** that yields a null pointer.

Dynamic cast

- A **reference or pointer** can be **downcast to a more precise type** in the class hierarchy in C++ using the **dynamic cast operator**.
- Unlike the **static cast**, the dynamic cast's **target must be a pointer or reference to a class**.
- A **type safety check** is carried out **at runtime instead of static cast and C-style typecast**, which do their **type checks during compilation**.
- When **dealing with references**, an **exception will be thrown**, or a **null pointer will be returned** if the types are **incompatible** (when dealing with pointers).

Java cast

- Similar to this, a **Java typecast will throw a java. Lang.ClassCastException** instance if the **object being cast is not actually an instance of the target type and cannot be transformed to one by a language-defined method.**

Casting Operators

- Casting operators are used for **type casting in C++**. They are used to **convert one data type to another**. C++ supports **four types of casts**:
 1. `static_cast`
 2. `dynamic_cast`
 3. `const_cast`
 4. `reinterpret_cast`

1. `static_cast`

- The `static_cast` operator is the **most commonly** used casting operator in C++. It performs **compile-time type conversion** and is mainly used for **explicit conversions** that are considered **safe by the compiler**.

Syntax of `static_cast`

`static_cast` <*new_type*> (expression);

where,

expression: Data to be converted.

new_type: Desired type of expression

- The `static_cast` can be used to **convert between related types**, such as **numeric types or pointers** in the **same inheritance hierarchy**.

// C++ program to illustrate the static_cast

```
#include <iostream>
```

```
using namespace std;
```

```
int main() {
```

```
    int j = 41;
```

```
    int v = 4;
```

```
    float m = j/v;
```

```
    float d = static_cast<float>(j)/v;
```

```
    cout << "m = " << m << endl;
```

```
    cout << "d = " << d << endl;
```

```
}
```

OUTPUT:

m = 10

d = 10.25

// C++ program to illustrate the static_cast

```
#include <iostream>
```

```
#include <typeinfo>
```

```
using namespace std;
```

```
int main() { int num = 10;
```

```
// converting int to double
```

```
double numDouble = static_cast<double>(num);
```

```
// printing data type
```

```
cout << typeid(num).name() << endl;
```

```
// typecasting
```

```
cout << typeid(static_cast<double>(num)).name() << endl;
```

```
// printing double type t
```

```
cout << typeid(numDouble).name() << endl;
```

```
return 0; }
```

Output

i

d

d

2. `dynamic_cast`

- The [`dynamic_cast`](#) operator is mainly used to **perform downcasting** (converting a pointer/reference of a base class to a derived class).
- It ensures **type safety by performing a runtime check to verify the validity of the conversion.**

Syntax of `dynamic_cast`

`dynamic_cast` *<new_type>* (expression);

- If the conversion is not possible, **`dynamic_cast`** returns a **null pointer** (for pointer conversions) or throws a **`bad_cast` exception** (for reference conversions).

```
// C++ program to illustrate the dynamic_cast
#include <iostream>
using namespace std;
// Base Class
class Animal {
public: virtual void speak() const {
cout << "Animal speaks." << endl;
} };
// Derived Class
class Dog : public Animal {
public: void speak() const override {
cout << "Dog barks." << endl;
} };
// Derived Class
class Cat : public Animal {
public: void speak() const override {
cout << "Cat meows." << endl;
} };
```

```
int main() {  
    // base class pointer to derived class object  
    Animal* animalPtr = new Dog();  
    // downcasting  
    Dog* dogPtr = dynamic_cast<Dog*>(animalPtr);  
    // checking if the typecasting is successfull  
    if (dogPtr) {  
        dogPtr->speack(); }  
    else {  
        cout << "Failed to cast to Dog." << endl;  
    }  
    // typecasting to other derved class  
    Cat* catPtr = dynamic_cast<Cat*>(animalPtr);  
    if (catPtr) {  
        catPtr->speack(); }  
    else {  
        cout << "Failed to cast to Cat." << endl; }  
    delete animalPtr;  
    return 0; }
```

Output:

Dog barks.

Failed to cast to Cat.

3. `const_cast`

- The `const_cast` operator is used to modify the `const` or `volatile` qualifier of a variable.
- It allows programmers to temporarily remove the constancy of an object and make modifications.
- Caution must be exercised when using `const_cast`, as modifying a `const` object can lead to **undefined behavior**.

Syntax for `const_cast`

`const_cast` <*new_type*> (expression);

// C++ program to illustrate the const_cast

```
#include <iostream>
```

```
using namespace std;
```

```
int main() {
```

```
const int number = 5;
```

```
// Pointer to a const int
```

```
const int* ptr = &number;
```

```
// int* nonConstPtr = ptr; if we use this instead of without using const_cast  
we will get error of invalid conversion
```

```
int*nonConstPtr = const_cast<int*>(ptr);
```

```
*nonConstPtr = 10;
```

```
cout << "Modified number: " << *nonConstPtr;
```

```
return 0;
```

```
}
```

OUTPUT:

Modified number: 10

4. reinterpret_cast

- The [reinterpret_cast operator](#) is used to **convert the pointer to any other type of pointer**. It does **not perform any check whether the pointer converted is of the same type or not**.

Syntax of reinterpret_cast

reinterpret_cast <*new_type*> (expression);

// C++ program to illustrate the reinterpret_cast

```
#include <iostream>
```

```
using namespace std;
```

```
int main() {
```

```
int number = 10;
```

```
// Store the address of number in numberPointer
```

```
int* numberPointer = &number;
```

```
// Reinterpreting the pointer as a char pointer
```

```
char* charPointer = reinterpret_cast<char*>(numberPointer);
```

```
// Printing the memory addresses and values
```

```
cout << "Integer Address: " << numberPointer << endl;
```

```
cout << "Char Address: " << reinterpret_cast<void*>(charPointer) << endl;
```

```
return 0; }
```

Output:

Integer Address: 0x7fff64789f1c

Char Address: 0x7fff64789f1c

UNIT-4

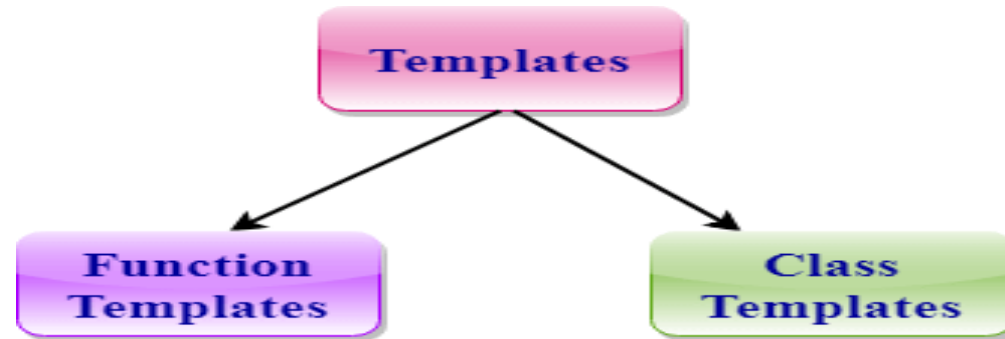
TEMPLATES AND EXCEPTION HANDLING

TEMPLATES

- The **templates** are one of the **most powerful** and **widely used methods** added to C++, **allowing** us to **write generic programs**.
- It allow us to **define generic functions and classes**. It **promote generic programming**, meaning the **programmer does not need to write the same function or method for different parameters**.
- We can define a **template as a blueprint for creating generic classes and functions**.
- The idea behind the templates in C++ is **straightforward**. We **pass the data type as a parameter**, so we **don't need to write the same code for different data types**.

TEMPLATES

- To perform a similar operation on several kinds of data types, a programmer need not write different versions by overloading a function.
- Instead the programmer can write a C++ template based function that will work with all data types.
- There are two types of templates in C++, function templates and class templates.

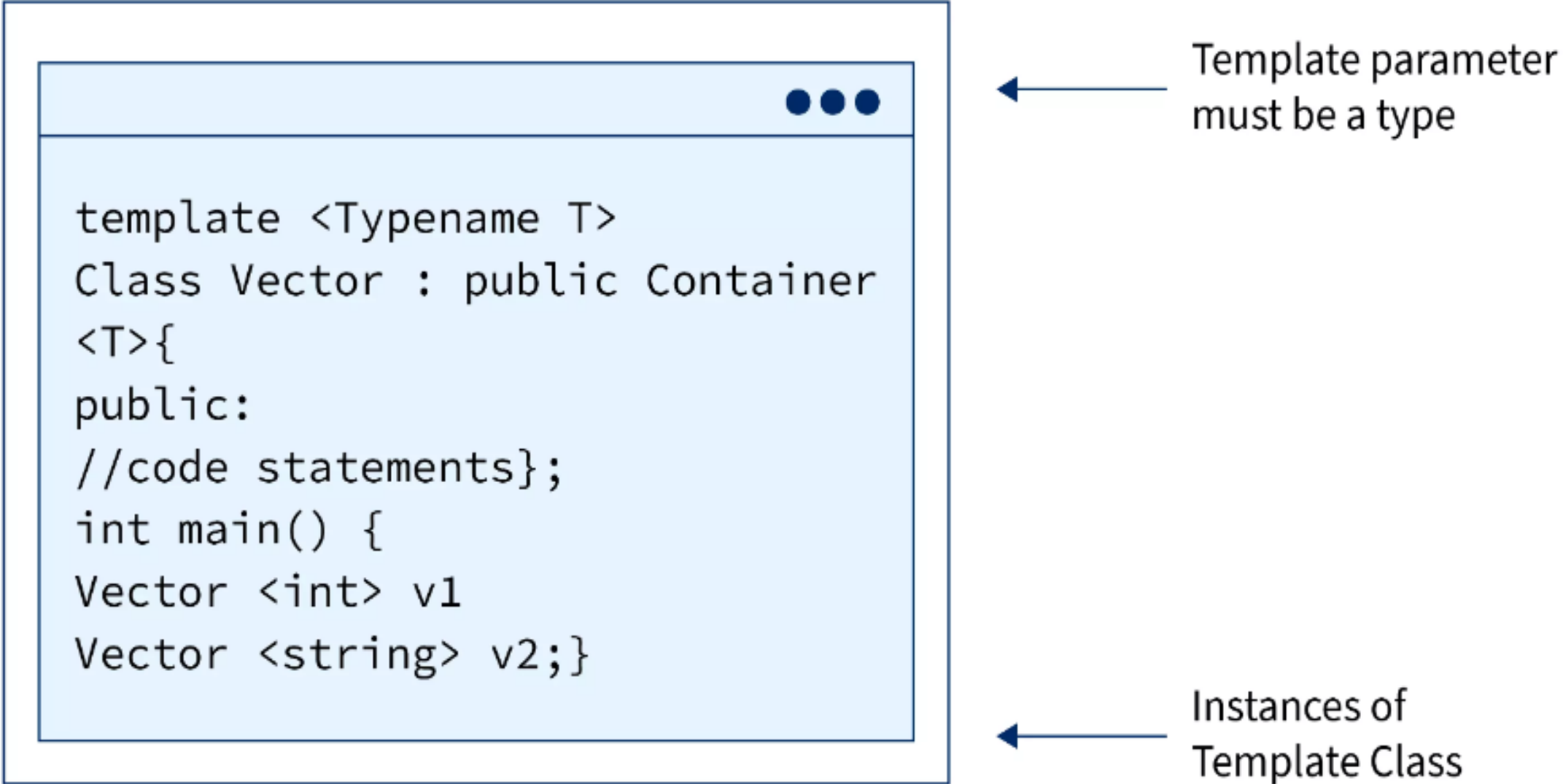


- C++ adds two new keywords to support templates: 'template' and 'typename'. The second keyword can always be replaced by the keyword 'class'.

- A template is a simple yet **very powerful tool** in C++.
- The simple idea is to **pass the data type as a parameter** so that we **don't need to write the same code** for different data types.
- For example, a **software company may need to sort()** for different data types.
- Rather than **writing and maintaining multiple codes**, we can **write one sort() and pass the datatype as a parameter**.

How Do Templates Work?

- Templates are **expanded at compiler time**. This is like macros. The difference is, that the **compiler does type-checking before template expansion**.
- The idea is simple, **source code contains only function/class**, but **compiled code may contain multiple copies of the same function/class**.



```
template <typename T>
Class Vector : public Container
<T>{
public:
//code statements};
int main() {
Vector <int> v1
Vector <string> v2;}
```

← Template parameter
must be a type

← Instances of
Template Class

Function Template

- Function templates are **similar to normal functions**. Normal functions work with **only one data type**, but a **function template code can work on multiple data types**. Hence, we can define function templates in C++ as a single generic function that can work with multiple data types.
- **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, **Quick sorting algorithm** is 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> ret_type func_name(parameter_list)
{
    // body of function.
}
```

- **Ttype:** It is a **place holder 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.

- 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**. Examples of function templates are `sort()`, `max()`, `min()`, `printArray()`.

```
// C++ Program to demonstrate
```

```
// Use of template
```

```
#include <iostream>
```

```
using namespace std;
```

```
// One function works for all data types. This would work
```

```
// even for user defined types if operator '>' is overloaded
```

```
template <typename T> T myMax(T x, T y)
```

```
{
```

```
    return (x > y) ? x : y;
```

```
}
```

```
int main()
```

```
{
```

```
    // Call myMax for int
```

```
    cout << myMax<int>(3, 7) << endl;
```

```
    // call myMax for double
```

```
    cout << myMax<double>(3.0, 7.0) <<
```

```
endl;
```

```
    // call myMax for char
```

```
    cout << myMax<char>('g', 'e') << endl;
```

```
    return 0;
```

```
}
```

Output

7 7 g

// C++ Program to implement Bubble sort using template function

```
#include <iostream>
```

```
using namespace std;
```

```
// A template function to implement bubble sort. We can use this for any data type that supports comparison operator < and swap works for it.
```

```
template <class T> void bubbleSort(T a[], int n)
```

```
{  
    for (int i = 0; i < n - 1; i++)  
        for (int j = n - 1; i < j; j--)  
            if (a[j] < a[j - 1])  
                swap(a[j], a[j - 1]); }  
}
```

```
int main()
```

```
{  
    int a[5] = { 10, 50, 30, 40, 20 };  
    int n = sizeof(a) / sizeof(a[0]);  
    // calls template function  
    bubbleSort<int>(a, n);  
    cout << " Sorted array : ";  
    for (int i = 0; i < n; i++)  
        cout << a[i] << " ";  
    cout << endl;  
    return 0;  
}
```

OUTPUT:

Sorted array : 10 20 30 40 50

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.  
}
```

```
#include <iostream>
using namespace std;
template<class X,class Y> void fun(X a,Y b)
{
    std::cout << "Value of a is : " <<a<< std::endl;
    std::cout << "Value of b is : " <<b<< std::endl;
}
int main()
{
    fun(15,12.3);
    return 0;
}
```

Output:

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

Overloading a Function Template

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

```
#include <iostream>
using namespace std;
template<class X> void fun(X a)
{
    std::cout << "Value of a is : " <<a<< std::endl;
}
template<class X,class Y> void fun(X b ,Y c)
{
    std::cout << "Value of b is : " <<b<< std::endl;
    std::cout << "Value of c is : " <<c<< std::endl;
}
int main()
{
    fun(10);
    fun(20,30.5);
    return 0;
}
```

Output:

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

Restrictions of Generic Functions

- Generic functions **perform the same operation for all the versions of a function except the data type differs**. Let's see a simple example of an **overloaded function which cannot be replaced by the generic function as both the functions have different functionalities**.

```
#include <iostream>
```

```
using namespace std;
```

```
void fun(double a)
```

```
{
```

```
    cout<<"value of a is : "<<a<<"\n";
```

```
}
```

```
void fun(int b)
```

```
{
```

```
    if(b%2==0)
```

```
{
```

```
cout<<"Number is even";  }  
    else  
    {  
        cout<<"Number is odd";  
    } }
```

```
int main()  
{ fun(4.6);  
  fun(6);  
  return 0; }
```

Output:

value of a is : 4.6
Number is even

CLASS TEMPLATE

- **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.

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.

```
#include <iostream>
using namespace std;
template<class T>
class A
{
    public:
    T num1 = 5;
    T num2 = 6;
    void add()
    {
        std::cout << "Addition of num1 and num2 : " << num1+num2<<std::endl;
    }
};
int main()
{
    A<int> d;
    d.add();
    return 0;
}
```

Output:

Addition of num1 and num2 : 11

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.  
}
```

```
#include <iostream>
using namespace std;
template<class T1, class T2>
class A
{ T1 a;
  T2 b;
public:
  A(T1 x,T2 y)
  {
    a = x;
    b = y;
  }
  void display() {
    std::cout << "Values of a and b are : " << a<<" ,"<<b<<std::endl;  }  };
int main()
{
  A<int,float> d(5,6.5);
  d.display();
  return 0;
}
```

Output:

Values of a and b are : 5,6.5

Non type 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**.

```
template<class T, int size>
```

```
class array
```

```
{  
    T arr[size];        // automatic array initialization.  
};
```

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.
```

```
#include <iostream>
using namespace std;
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++)
    {
        std::cout << arr[i] << " ";
    }
}
};
int main()
{
    A<int,10> t1;
    t1.insert();
    t1.display();
    return 0;
}
```

Output:

1 2 3 4 5 6 7 8 9 10

More than one argument for templates

Like normal parameters, we can **pass more than one data type as arguments to templates**. The following example demonstrates the same.

```
// C++ Program to implement Use of template
```

```
#include <iostream>
```

```
using namespace std;
```

```
template <class T, class U> class A {
```

```
    T x;
```

```
    U y;
```

```
public:
```

```
    A() { cout << "Constructor Called" << endl; }
```

```
};
```

```
int main()
```

```
{
```

```
    A<char, char> a;
```

```
    A<int, double> b;
```

```
    return 0;
```

```
}
```

Output:

Constructor Called

Constructor Called

specify a default value for template arguments

like normal parameters, we can specify **default arguments to templates**. The following example demonstrates the same

```
// C++ Program to implement Use of template
#include <iostream>
using namespace std;
template <class T, class U = char> class A {
public:
    T x;
    U y;
    A() { cout << "Constructor Called" << endl; }
};
int main()
{
    // This will call A<char, char>
    A<char> a;
    return 0;
}
```

Output

Constructor Called

Generic Functions

- Generic functions are **functions declared with one or more generic type parameters**. They may be **methods in a class or struct , or standalone functions**. A single generic declaration **implicitly declares a family of functions** that differ only in the substitution of a different actual type for the generic type parameter.
- 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.

Generic Functions using Template:

- Generic function that can be **used for different data types**. Examples of function templates are sort(), max(), min(), printArray()

```
#include <iostream>
```

```
using namespace std;
```

```
// One function works for all data types. This would work even for user defined types if operator '>' is overloaded
```

```
template <typename T>
```

```
T myMax(T x, T y)
```

```
{ return (x > y) ? x : y;
```

```
}
```

```
int main()
```

```
{
```

```
    // Call myMax for int
```

```
    cout << myMax<int>(3, 7) << endl;
```

```
    // call myMax for double
```

```
    cout << myMax<double>(3.0, 7.0) << endl;
```

```
    // call myMax for char
```

```
    cout << myMax<char>('g', 'e') << endl;
```

```
    return 0;
```

```
}
```

Output:

7 7 g

Generic classes

- Classes which are **preceded by a the template keyword** are **automatically turned into generic classes**.
- These classes can have **any number of template parameters**, we're going to concentrate on classes having just one, this being a template type parameter.
- The class definition starts of as follows:

```
template<typename T>  
class Named {  
    T value;  
    std::string name;  
public:  
    // rest of class definition ...
```

- The **unspecified (generic) type T** can be **used within the class body in place of a user-defined or built-in type name** (in this case with the member variable value), and other non-template member variables can be used (as shown with name).
- Strictly speaking, an **instance of class Named should be defined as Named<int>** etc., but in Modern C++ the type detection facilities allow a plain Named in many cases. In older code the use of class instead of typename may sometimes be found;

```
#include <iostream>
#include <string>
#include <string_view>
template<typename T>
class Named {
    T value;
    std::string name;
public:
    Named(const T& value, std::string_view name) : value{value}, name{name} {}
    const T& get() const { return value; }
    std::string_view get_name() const { return name; } };
template<typename T>
std::ostream& operator<<(std::ostream& os, const Named<T>& n) {
    return os << n.get_name() << ':' << n.get(); }
int main() {
    Named i(42, "answer");
    Named p(3.14, "pi");
    std::cout << i << '\n' << p << '\n'; }
```

Output:
answer:42
pi:3.14

Generic Class using Template:

- Like function templates, **class templates are useful when a class defines something that is independent of data type**. Can be useful for classes like LinkedList, binary tree, Stack, Queue, Array, etc.

```
#include <iostream>
```

```
using namespace std;
```

```
template <typename T>
```

```
class Array {
```

```
private:
```

```
    T* ptr;
```

```
    int size;
```

```
public:    Array(T arr[], int s);
```

```
    void print(); };
```

```
template <typename T>
```

```
Array<T>::Array(T arr[], int s)
```

```
{    ptr = new T[s];
```

```
    size = s;
```

```
    for (int i = 0; i < size; i++)
```

```
        ptr[i] = arr[i];
```

```
}
```

```
template <typename T>
```

```
void Array<T>::print()
```

```
{
    for (int i = 0; i < size; i++)
        cout << " " << *(ptr + i);
    cout << endl;
}
```

```
int main()
```

```
{
    int arr[5] = { 1, 2, 3, 4, 5 };
    Array<int> a(arr, 5);
    a.print();
    return 0;
}
```

Output:
1 2 3 4 5

Working with multi-type Generics:

We can pass more than one data types as arguments to templates. The following example demonstrates the same.

```
#include <iostream>
using namespace std;
template <class T, class U>
class A {
    T x;
    U y;
public:
    A()
    {
        cout << "Constructor Called" << endl;
    }
};
int main()
{
    A<char, char> a;
    A<int, double> b;
    return 0;
}
```

Output:
Constructor Called
Constructor Called

Type name

- **Typename keywords** in C++ are used to **tell the compiler** that a **dependent name**(A name that depends on the value of the template parameter) in a **template is a type**. These are **always used alongside template keywords** in C++.
- " **Typename** " is a keyword in the **C++ programming language used when writing templates**. It is used for **specifying that a dependent name in a template definition or declaration is a type**.
- In C++, the “**typename**” is a keyword that was **introduced to declare a type**. It is used for **specifying that the identifier that follows is a type rather than a static member variable**.
- **Typename replaces “class”** in a template keyword in C++ syntax. There is no significant difference between **typename** and **class** for a compiler.

Syntax

```
template<typename X>
```

```
class <class name>{<code>}
```


C++ typename Keyword

The typename keyword is mainly used for two cases:

1. Declaring Template Type Parameters

2. In the Type Declaration

1. Declaring Template Type Parameters

```
cppCopy codetemplate <typename T>
class MyClass {
// Class definition
};
```

2. In the Type Declaration

Inside member functions of class templates, typename is used when referring to nested types from template parameters.

```
cppCopy codetemplate <typename T>
class MyClass {
public:
void myMethod() {
typename T::nested_type variable;
// 'typename' is necessary here to specify that T::nested_type is a type
}
};
```

```

// C++ Program to use the typename keyword
#include <iostream>
#include <vector>
using namespace std;
// function for printing elements of a container passed as parameter
template <typename T> void printElements(const T& container)
{
// loop using the iterator it of container type
    for (typename T::const_iterator it = container.begin(); it != container.end(); ++it)
        // Dereferencing iterator for getting the element and print it
        cout << *it << " ";
    // Print a newline character after all elements are printed
    cout << endl;
}

int main()
{
// Create a vector vec and initialize it
    vector<int> vec = { 1, 2, 3, 4, 5 };
// Use the template function to print elements of the vector
    cout << "Elements: ";
    printElements(vec);
    return 0;
}

```

OUTPUT:
Elements: 1 2 3 4 5

Export Keywords

- Use the **`_Export`** keyword with a function name to declare that it is to be exported (made available to other modules). You must define the function in the same translation unit in which you use the **`_Export`** keyword.

For example: `int _Export anthony(float);`

- Within a module, we can **define declarations** (e.g., variables, functions, classes) that we **want to export to other modules using the ‘export’ keyword**. Exported declarations **become part of the module’s interface** and can be **imported by other modules**.
- `export module module_name; // module declaration`
`export data_type variable_name; // export declaration`
`export return_type function_name(); // export declaration`

Power of templates

- Templates in C++ are a **powerful feature that enables generic programming**, allowing you to **write code that works with any data type**. Here are some key aspects of their power:

1. Code Reusability

- **Generic Functions:** You can **create functions that can operate on any data type**. For example, a template function for finding the maximum of two values can be used with integers, floats, or even custom classes.

Ex:

```
template <typename T>
T maximum(T a, T b) {
    return (a > b) ? a : b;
}
```

2. Type Safety

- **Compile-Time Type Checking:** Templates **allow the compiler to check types at compile time, reducing runtime errors**. This means that if you try to use incompatible types, you'll get an error during compilation.

3. Flexibility

- **Template Classes:** You can create template classes that **can handle any data type**. This is **especially useful for data structures like linked lists, stacks, and queues**.

```
template <typename T>
class Box {
private:
    T value;
public:
    Box(T v) : value(v) {}
    T getValue() const { return value; }
};
```

4. Specialization

- **Template Specialization:** You can **create specific implementations of templates** for particular types, which allows you to optimize performance for certain cases.

```
template <>
class Box<int> {
private:
    int value;
public:
    Box(int v) : value(v) {}
    int getValue() const { return value * 2; } // Example specialization
};
```

5. Metaprogramming

- **Compile-Time Computation:** Templates can be used for **metaprogramming**, where **computations are performed at compile time**, allowing for **more optimized and efficient code**.

6. Type Traits and SFINAE

- **Type Traits:** Using templates with type traits (like `std::enable_if`) allows for **advanced type manipulations and constraints**, enabling **more control over template behavior**.

Ex:

```
template <typename T>
typename std::enable_if<std::is_integral<T>::value, T>::type
add(T a, T b) {
    return a + b;
}
```

7. Standard Template Library (STL)

C++'s Standard Template Library **heavily utilizes templates**, providing a **rich set of generic algorithms and data structures** (like vectors, lists, and maps) that can be used with any data type.

Exception Handling

- Exception Handling in C++ 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 std::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.

- An exception is an **unexpected problem that arises during the execution** of a program our program terminates suddenly with some errors/issues. Exception **occurs during the running of the program** (runtime).

Types of C++ Exception

There are **two types** of exceptions in C++

1. **Synchronous:** Exceptions that happen when something goes wrong because of a **mistake in the input data or when the program is not equipped to handle the current type** of data it's working with, such as dividing a number by zero.
2. **Asynchronous:** Exceptions that are beyond the program's control, such as **disc failure, keyboard interrupts, etc.**

C++ try and catch

C++ provides an **inbuilt feature for Exception Handling**. It can be **done using the following specialized keywords**: try, catch, and throw with each having a different purpose.

Syntax:

```
try {  
    // Code that might throw an exception  
    throw SomeExceptionType("Error message");  
}  
catch( ExceptionName e1 ) {  
    // catch block catches the exception that is thrown from try block  
}
```

1. try in C++

The try keyword represents a block of code that **may throw an exception placed inside the try block**. It's followed by **one or more catch blocks**. If an exception occurs, try **block throws that exception**.

2. catch in C++

The catch statement represents a **block of code that is executed when a particular exception is thrown from the try block**. The code to handle the exception is written inside the catch block.

3. throw in C++

An exception in C++ can be thrown using the **throw keyword**. When a program encounters a throw statement, then it **immediately terminates the current function and starts finding a matching catch block** to handle the thrown exception.

NEED OF EXCEPTION HANDLING

The following are the **main advantages of exception handling** over traditional error handling:

Separation of Error Handling Code from Normal Code: There are **always if-else conditions to handle errors in traditional error handling codes**. These conditions and the code to **handle errors get mixed up with the normal flow**. This makes the **code less readable and maintainable**. With try/catch blocks, the code for error handling becomes separate from the normal flow.

Functions/Methods can handle only the exceptions they choose: A function can throw many exceptions, but may choose to handle some of them. The other exceptions, which are **thrown but not caught**, can be **handled by the caller**. If the caller chooses **not to catch them**, then the **exceptions are handled by the caller of the caller**.

In C++, a function can specify the exceptions that it **throws using the throw keyword**. The caller of this function must handle the exception in some way (either by specifying it again or catching it).

Grouping of Error Types: In C++, **both basic types and objects can be thrown as exceptions**. We can **create a hierarchy of exception objects**, group exceptions in namespaces or classes, and categorize them according to their types.

Examples of Exception Handling in C++

// C++ program to demonstrate the use of try, catch and throw in exception handling.

```
#include <iostream>
```

```
#include <stdexcept>
```

```
using namespace std;
```

```
int main()
```

```
{ // try block
```

```
    try {
```

```
        int numerator = 10;
```

```
        int denominator = 0;
```

```
        int res;
```

```
// check if denominator is 0 then throw runtime
// error.
if (denominator == 0) {
    throw runtime_error(
        "Division by zero not allowed!");
}
// calculate result if no exception occurs
res = numerator / denominator;
//[printing result after division
cout << "Result after division: " << res << endl;
}
// catch block to catch the thrown exception
catch (const exception& e) {
    // print the exception
    cout << "Exception " << e.what() << endl;
}
return 0;
}
```

Output

Exception Division by zero not allowed!

Example 2

```
#include <iostream>
using namespace std;
int main()
{
    int x = -1;
    // Some code
    cout << "Before try \n";
    // try block
    try {
        cout << "Inside try \n";
        if (x < 0) {
            // throwing an exception
```

```
throw x;
    cout << "After throw (Never executed) \n";
} }
// catch block
catch (int x) {
    cout << "Exception Caught \n";
}
cout << "After catch (Will be executed) \n";
return 0;
}
```

Output

Before try

Inside try

Exception Caught

After catch (Will be executed)

Properties of Exception Handling

Property 1: There is a special catch block called the ‘catch-all’ block, written as catch(...), that can be used to catch all types of exceptions. In the following program, an int is thrown as an exception, but there is no catch block for int, so the catch(...) block will be executed.

// C++ program to demonstrate the use of catch all in exception handling.

```
#include <iostream>
```

```
using namespace std;
```

```
int main()
```

```
{ // try block
```

```
    try {
```

```
        // throw
```

```
        throw 10; }
```

```
// catch block
```

```
    catch (char* excp) {
```

```
        cout << "Caught " << excp; }
```

```
// catch all
```

```
    catch (...) {
```

```
        cout << "Default Exception\n"; }
```

```
    return 0; }
```

Output

Default Exception

Property 2:Implicit type conversion doesn't happen for primitive types. In the following program, 'a' is not implicitly converted to int.

// C++ program to demonstate property 2: Implicit type conversion doesn't happen for primitive types. in exception handling.

```
#include <iostream>
using namespace std;
int main()
{
    try {
        throw 'a';
    }
    catch (int x) {
        cout << "Caught " << x;
    }
    catch (...) {
        cout << "Default Exception\n";
    }
    return 0;
}
```

Output

Default Exception

Property 3: If an exception is thrown and not caught anywhere, the program terminates abnormally.

Example

In the following program, a char is thrown, but there is no catch block to catch the char. \

// C++ program to demonstrate property 3: If an exception is thrown and not caught anywhere, the program terminates abnormally in exception handling.

```
#include <iostream>
using namespace std;
int main()
{
    try {
        throw 'a';
    }
    catch (int x) {
        cout << "Caught ";
    }
    return 0;
}
```

Output

terminate called after throwing an instance of 'char'

Property 4: Unlike Java, in C++, all exceptions are unchecked, i.e., the compiler doesn't check whether an exception is caught or not (See this for details). So, it is not necessary to specify all uncaught exceptions in a function declaration. However, exception-handling it's a recommended practice to do so.

Example

- The following program compiles fine, but ideally, the signature of fun() should list the unchecked exceptions.

```
// C++ program to demonstate property 4 in better way
#include <iostream>
using namespace std;
// Here we specify the exceptions that this function throws.
void fun(int* ptr, int x) throw(
    int*, int) // Dynamic Exception specification
{
    if (ptr == NULL)
        throw ptr;
    if (x == 0)
        throw x;
    /* Some functionality */
}
int main()
{
    try {
        fun(NULL, 0);
    }
    catch (...) {
        cout << "Caught exception from fun()";
    }
    return 0;
}
```

Output

Caught exception from fun()

Property 5:In C++, try/catch blocks can be nested. Also, an exception can be re-thrown using “throw;”.

Example

- The following program shows try/catch blocks nesting.

```
// C++ program to demonstrate try/catch blocks can be nested in C++
#include <iostream>
using namespace std;
int main()
{
    // nesting of try/catch
    try {
        try {
            throw 20;
        }
        catch (int n) {
            cout << "Handle Partially ";
            throw; // Re-throwing an exception
        }
    }
    catch (int n) {
        cout << "Handle remaining ";
    }
    return 0;
}
```

Output

Handle Partially Handle remaining

Property 6: When an exception is thrown, all objects created inside the enclosing try block are destroyed before the control is transferred to the catch block.


```
// C++ program to demonstrate
#include <iostream>
using namespace std;
// Define a class named Test
class Test {
public:
    // Constructor of Test
    Test() { cout << "Constructor of Test " << endl; }
    // Destructor of Test
    ~Test() { cout << "Destructor of Test " << endl; }
};
int main()
{
    try {
        // Create an object of class Test
        Test t1;
        // Throw an integer exception with value 10
        throw 10;
    }
    catch (int i) {
        // Catch and handle the integer exception
        cout << "Caught " << i << endl;
    }
}
```

Output

```
Constructor of Test
Destructor of Test
Caught 10
```

Limitations of Exception Handling

- Exceptions may break the structure or flow of the code as multiple invisible exit points are created in the code which makes the code hard to read and debug.
- If exception handling is not done properly can lead to resource leaks as well.
- It's hard to learn how to write Exception code that is safe.
- There is no C++ standard on how to use exception handling, hence many variations in exception-handling practices exist.

Exception handling options

- Exception handling in C++ consist of three keywords: **try** , **throw** and **catch** :
The try statement allows you to define a block of code to be tested for errors while it is being executed. The throw keyword throws an exception when a problem is detected, which lets us create a custom error.
- The try statement allows you to define a block of code to be tested for errors while it is being executed.
- The throw keyword throws an exception when a problem is detected, which lets us create a custom error.
- The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.
- The try and catch keywords come in pairs:

1. try in C++

The try keyword represents a block of code that may throw an exception placed inside the try block. It's followed by one or more catch blocks. If an exception occurs, try block throws that exception.

2. catch in C++

The catch statement represents a block of code that is executed when a particular exception is thrown from the try block. The code to handle the exception is written inside the catch block.

3. throw in C++

An exception in C++ can be thrown using the throw keyword. When a program encounters a throw statement, then it immediately terminates the current function and starts finding a matching catch block to handle the thrown exception.

```
try {  
    // Block of code to try  
    throw exception; // Throw an exception when a problem arise  
}  
catch () {  
    // Block of code to handle errors  
}
```

```
#include <iostream>
using namespace std;
int main() {
    try {
        int age = 15;
        if (age >= 18) {
            cout << "Access granted - you are old enough.";
        } else {
            throw (age);
        }
    }
    catch (int myNum) {
        cout << "Access denied - You must be at least 18 years old.\n";
        cout << "Age is: " << myNum;
    }
    return 0;
}
```

Handle Any Type of Exceptions (...)

- If you do not know the throw **type** used in the try block, you **can use the "three dots" syntax (...)** inside the catch block, which will **handle any type of exception**:

Example

```
#include <iostream>
using namespace std;
int main() {
    try {
        int age = 15;
        if (age >= 18) {
            cout << "Access granted - you are old enough.";
        } else {
            throw 505;
        }
    }
    catch (...) {
        cout << "Access denied - You must be at least 18 years old.\n";
    }
    return 0; }
```

OUTPUT:

Access denied - You must be at
least 18 years old.

Terminate() and unexpected()

- Whenever an **exception arises in C++**, it is **handled as per the behavior defined using the try-catch block**. However, there is often the case when an **exception is thrown but isn't caught** because the **exception handling subsystem fails** to find a **matching catch block** for that particular exception. In that case, the following set of actions takes place:
- The exception handling subsystem calls the function: **unexpected()**. This function, **provided by the default C++ library**, defines the **behavior when an uncaught exception arises**. By default, **unexpected** calls **terminate()**.
- **The terminate function** defines the **actions that are to be performed during process termination**. This, by default, calls **abort()**.
- The process is aborted.

The terminate() function:

- **If no handler at any level catches the exception, the special library function terminate() (declared in the <exception> header) is automatically called**. By default, **terminate()** calls the Standard C library function **abort()**, which abruptly exits the program.

- The `terminate()` and `unexpected()` simply call other functions to actually handle an error. As explained above, `terminate` calls `abort()`, and `unexpected()` calls `terminate()`. Thus, both functions halt the program execution when an exception handling error occurs. However, you can change the way termination occurs.
- To change the terminate handler, the function used is `set_terminate(terminate_handler newhandler)`, which is defined in the header `<exception>`.

The following program demonstrates how to set a custom termination handler:

```
// CPP program to set a new termination handler for uncaught exceptions.
```

```
#include <exception>
```

```
#include <iostream>
```

```
using namespace std;
```

```
// definition of custom termination function
```

```
void myhandler()
```

```
{ cout << "Inside new terminate handler\n";  
  abort(); }
```

```
int main()
```

```
{ // set new terminate handler
```

```
  set_terminate(myhandler);
```

```
  try {
```

```
    cout << "Inside try block\n";
```

```
    throw 100; }
```

```
  catch (char a) // won't catch an int exception
```

```
{
```

```
  cout << "Inside catch block\n"; }
```

```
  return 0; }
```

OUTPUT:

Inside try block

Inside new terminate handler

uncaught exception()

- When the **exception mechanism searches catch handlers for a thrown exception**, the **signature of a catch handler must match the type of the exception**. If a catch handler is **not found as the stack unwinds**, the exception is uncaught.
- If **none of the catch handlers for a try block matches a thrown exception** the **exception moves to the next enclosing try block**.
- If there is **no match in any enclosing try block the exception is uncaught**. An **uncaught exception also occurs if a new exception is thrown before an existing one is handled**. Cleanups may fail to occur with an uncaught exception, so this is an error.
- If an **exception is uncaught the special function terminate()** is called.
- Uncaught exceptions can **always be avoided by enclosing the contents of main in a try block with an ellipsis handler**.

Exception and bad exception Classes

- Standard C++ contains several built-in exception classes, `exception::bad_exception` is one of them. This is an exception thrown by unexpected handler. Below is the syntax for the same:

Header File:

```
include<exception>
```

Syntax:

```
class bad_exception;
```

- **Return:** The `exception::bad_exception` returns a null terminated character that is used to identify the exception.

```
// C++ code for std::bad_exception
#include <bits/stdc++.h>
using namespace std;
void func() {
    throw;
}
void geeksforgeeks() throw(bad_exception) {
    throw runtime_error("test");
}
int main()
{
    set_unexpected(func);
    // try block
    try {
        geeksforgeeks(); }
    // catch block to handle the errors
    catch (const bad_exception& gfg) {
        cout << "Caught exception " << gfg.what() << endl; }
    return 0; }
```

OUTPUT:

Caught exception std::bad_exception

I/O STREAMS STANDARD TEMPLATE LIBRARY

FILE I/O<fstream>AND THE FILE
CLASSES

FILE HANDLING THROUGH C++ CLASSES

- File handling is used to store data permanently in a computer. Using file handling we can store our data in secondary memory(Hard disk).

How to achieve the file handling

For achieving file handling we need to follow the following steps:

STEP 1: Naming a file.

STEP 2: Opening a file.

STEP 3: Writing data into the file.

STEP 4: Reading data from the file.

STEP 5: Closing a file.

FILE I/O IN C++

- The `<fstream>` library provides functionality for file input and output.
- It includes three main classes:
 1. `ifstream` (Input File Stream)
 2. `ofstream` (Output File Stream)
 3. `fstream` (Input/Output File Stream)

FILE I/O IN C++

- `ifstream`:

Used for reading files.

Syntax:

`std::ifstream`

- `ofstream`:

Used for writing files.

Syntax:

`std::ofstream`

- `fstream`:

Used for both reading and writing.

Syntax:

`std::fstream`

DEFAULT OPEN MODES

- ifstream
- ofstream
- fstream
- ios::in
- ios::out
- ios::in|ios::out

EXAMPLE FOR IFSTREAM

```
#include<fstream>
Std::ifstream inFile("filename.txt");
if(!inFile)
{
}
std::string line;
while(std::getline(inFile,line))
{
    std::cout<<line<<std::endl;
}
inFile.close();
```

EXAMPLE FOR OFSTREAM

```
#include<fstream>
std::ofstream outFile("filename.txt");
if(!outFile)
{
}
outFile<<"Hello,World!"<<std::endl;
outFile.close();
```

EXAMPLE FOR FSTREAM

```
#include<fstream>
std::fstream file("filename.txt",std::ios::in | std::ios::out | std::ios::app);
if(!file)
{
}
file<<"Appending this line."<<std::endl;
file.seekg(0);
Std::string line;
while(std::getline(file,line))
{
    std::cout<<line<<std::endl;
}
file.close();
```

BASIC OPERATION

- OPENING A FILE:

You can open a file using the constructor or the `open()` method.

- READING FROM A FILE:

Use the `>>` operator or member functions like `getline()`.

- WRITING TO A FILE:

Use the `<<` operator.

- CLOSING A FILE:

Always close your files with the `close()` method to free resources.

EXAMPLE PROGRAM

```
#include<fstream>
#include<iostream>
#include<string>
int main()
{
    //File output
    std::ofstream outFile("example.txt");
    if(!outFile)
    {
        std::cerr<<"Unable to open file";
        return 1;
    }
}
```

```
Std::cout<<"Enter your name:";
    std::string name;
    std::cin>>name;
    std::cout<<"Enter your age:";
    int age;
    std::cin >>age;
    outFile<<"Name:"<<name<<std::endl;
    outFile<<"Age:"<<age<<std::endl;
    outFile.close();
    //file Input
    std::ifstream inFile("example.txt");
    if(!inFile)
```



```
{
    std::cerr<<"Unable to open file";
    return 1;
}
std::string line;
std::cout<<"File Contents:"<<std::endl;
while(std::getline(inFile,line))
{
    std::cout<<line<<std::endl;
}
inFile.close();
return 0;
}
```

OUTPUT:

Enter your name : kaviya

Enter your age : 18

File Contents:

Name : kaviya

Age : 18

ADVANTAGES

1. Efficient File Handling:

File streams provide efficient file handling capabilities.

2. Flexibility:

Support various file modes (e.g., binary, text).

3. Error Handling:

Built-in error handling mechanisms.

4. Type Safety:

Ensure data type consistency.

DISADVANTAGES

1. Complexity:

Steeper learning curve due to various classes and modes.

2. Performance Overhead:

Additional overhead compared to lower-level file operations.

3. Limited Control:

Less direct control over file operations.

LIMITATIONS

1. Performance Overhead:

File streams can introduce performance overhead due to buffering and synchronization.

2. Limited Control:

File streams provide limited control over file operations, making them less suitable for low-level file manipulation.

3. Error Handling:

While file streams provide built-in error handling, they can be restrictive in handling complex error scenarios.

4. Complexity:

File classes can be complex to use, especially for simple file operations.

5. Platform Dependence:

File classes may exhibit platform-dependent behavior.

THANK YOU

By:

T.Anitha

M.Kaviya

G.Ashika sri