C++ Note Book

C++ Note Book

Basic Part

Stroge Class

C++ Storage Classes are used to describe the characteristics of a variable/function. It determines the lifetime, visibility, default value, and storage location which helps us to trace the existence of a particular variable during the runtime of a program. Storage class specifiers are used to specify the storage class for a variable.

To specify the storage class for a variable, the following syntax is to be followed:

```
storage_class var_data_type var_name;
```

C++ uses 6 storage classes, which are as follows:

1. **auto Storage Class**
    - **Scope:** Local
    - **Default Value:** Garbage Value
    - **Memory Location:** RAM
    - **Lifetime:** Till the end of its scope
2. **register Storage Class**
    - **Scope:** Local
    - **Default Value:** Garbage Value
    - **Memory Location:** Register in CPU or RAM
    - **Lifetime:** Till the end of its scope

3. **extern Storage Class**
    - **Scope:** Global
    - **Default Value:** Zero
    - **Memory Location:** RAM
    - **Lifetime:** Till the end of the program.
4. **static Storage Class**
    - **Scope:** Local
    - **Default Value:** Zero
    - **Memory Location:** RAM
    - **Lifetime:** Till the end of the program

      Note: *Global Static variables can be accessed in any function.*
5. **mutable Storage Class**

   The mutable specifier does not affect the linkage or lifetime of the object. It will be the same as the normal object declared in that place.
6. **thread_local Storage Class**
    - **Memory Location:** RAM
    - **Lifetime:** Till the end of its thread

▼ C++ Operator

- sizeof (expreation) >> The sizeof operator is a unary compile-time operator used to determine the size of variables, data types, and constants in bytes at compile time. It can also determine the size of classes, structures, and unions

  summerise talk check byte ??

```
int x[]={1,2,3};
int length1=sizeof(x)/sizeof(x[0]);
/*hear total size=12 and number of one
```

```
element size=4 So the length is >> 12/4=3
*/
```

- Scope resolution >> : :

1. **To access a global variable when there is a local variable with same**

```
int x=10;
int main(){
int x=20;
cout<<x;
cout<<::x; //access global variable
}
```

2. **To define a function outside a class**

```
class A{
    int func();
};
int A ::func(){data}
```

3. **To access a class's static variables.**

```
class A{
    static int x;
};
int A::x=10; //static variable data store
```

4. **In case of multiple Inheritance:** If the same variable name exists in two ancestor classes, we can use scope resolution operator to distinguish.

```
class A{
public:
int x;
};

class B:public A{
```

```
public:
int x;
};
class C:public A,Public B{
public:
int x;
cout<<A::x<<B::x; //only public variable accessabl

};
```

5. **For namespace** If a class having the same name exists inside two namespaces we can use the namespace name with the scope resolution operator to refer that class without any conflicts

```
#include <bits/stdc++.h>
#include <iostream>
using namespace std;
#define nline "\n"

// Global Declarations

string name1 = "GFG";
string favlang = "python";
string companyName = "GFG_2.0";

// You can also do the same thing in classes as we
// our struct example

class Developer {
public:
    string name = "krishna";
    string favLang = "c++";
    string company = "GFG";

    // Accessing Global Declarations

    Developer(string favlang, string company)
        : favLang(favlang)
```

```
        , company(companyName)
    {
    }
};

int main()
{
    Developer obj = Developer("python", "GFG");
    cout << "favourite Language : " << obj.favLang
    cout << "company Name : " << obj.company << nl
}
```

6. **Refer to a class inside another class:** If a class exists inside another class we can use the nesting class to refer the nested class using the scope resolution operator

```
// Use of scope resolution class inside another cl
#include <iostream>
using namespace std;

class outside {
public:
    int x;
    class inside {
    public:
        int x;
        static int y;
        int foo();
    };
};
int outside::inside::y = 5;

int main()
{
    outside A;
    outside::inside B;
}
```

▼ Data type Range

```
Data Type                                   Range
Macro for min value              Macro for max value
char                                      -128 to +127
CHAR_MIN                         CHAR_MAX
short char                                -128 to +127
SCHAR_MIN                        SCHAR_MAX
unsigned char                               0  to 255
0                          UCHAR_MAX

short int                                -32768 to +32767
SHRT_MIN                         SHRT_MAX
unsigned short int                          0  to  65535
0                          USHRT_MAX
int                                  -2147483648 to +21474836
47                    INT_MIN                          INT
_MAX
unsigned int                                0  to  42949672
95                       0                           UINT
_MAX
long int              -9223372036854775808 to +92233720
36854775807         LONG_MIN                        LONG
_MAX
unsigned long int                           0  to  18446744
073709551615             0                         ULONG
_MAX
long long int         -9223372036854775808 to +92233720
36854775807        LLONG_MIN                        LLONG
_MAX
unsigned long long int                      0  to  18446744
073709551615             0                           ULLONG
_MAX

float                            1.17549e-38 to  3.40282e
+38                   FLT_MIN                          FLT
_MAX
float(negative)                 -1.17549e-38 to -3.40282e
```

```
+38                -FLT_MIN                        -FLT
_MAX
double                      2.22507e-308 to  1.79769e
+308                DBL_MIN                         DBL
_MAX
double(negative)           -2.22507e-308 to -1.79769e
+308               -DBL_MIN                        -DBL
_MAX
```

▼ C++ Modifer

1. **signed Modifier**

   Signed variables can store positive, negative integers, and zero.

   ```
   signed int a = 45;
   signed int b = -67;
   signed int c = 0;
   ```

   Note: *The int datatype is signed by default. So, int can be directly be used instead of signed int.*

2. **unsigned Modifier**

   Unsigned variables can store only non-negative integer values.

   ```
   unsigned int a = 9;
   unsigned int b = 0;
   ```

   Merjor Diffrences of Signed and Unsigned

- The sign takes 1 bit extra. So, if the unsigned value is being used then one-bit extra space is used to save the value of a variable.
- The range of values for unsigned types starts from 0. For example, for **unsigned int**, the value range is from **0 to 4,294,967,295**. However, for **signed int**, the value range is from **2,147,483,648 to 2,147,483,647**.

  Note: *signed and unsigned modifiers can only be used with int and char datatypes.*

1. **short Modifier**

The **short** keyword modifies the minimum values that a data type can hold. It is used for small integers that lie in the range of **−32,767 to +32,767**.

```
short int x = 4590;
```

Note: *The short int can be written as short also. They are equivalent.*

2. **long Modifier**

   The **long** keyword modifies the maximum values that a data type can hold. It is used for large integers that are in the range of **-2147483647 to 214748364**

```
long int y = 26936;
```

Note: *The long int can be written as long also. They are equivalent.*

Various Data Types with Modifiers and Their Size in Bytes

1. Character Data Type (char)

Character datatype only allows signed and unsigned modifiers. When only **char** is used instead of signed char or unsigned char, this type is known as plain char. When performing numerical computations, it is preferred to utilize signed char or unsigned char instead of plain char. Character values should only be stored as plain char.

S No.	Modifier	Size(in Bytes)
1.	char	1
2.	signed char	1
3.	unsigned char	1

2. Integer Data Type (int)

S No.	Modifier	Size(in Bytes)
1.	int	4
2.	signed int	4

3.	unsigned int	4
4.	short	2
5.	signed short	2
6.	unsigned short	2
7.	long	8
8.	signed long	8
9.	unsigned long	8

3. Floating Point (float) and Double Precision Floating Point (double)

Double type can be used with the long modifier.

S No.	Modifier	Size(in Bytes)
1.	float	4
2.	double	8
3.	long double	16

Type Qualifiers in C++:

Type qualifiers are used to provide more information about a value while also guaranteeing that the data is used correctly.

1. **const:** Objects of type **const** cannot be altered during execution. Const objects cannot be modified by your program while it is running.
2. **volatile:** The modifier **volatile** tells the compiler that a variable's value can be changed in ways that are not explicitly defined by the program. The compiler is informed by the modifier volatile that a variable's value might change in ways that aren't clearly stated in the program.
3. **restrict:** A pointer qualified by **restricting** is initially the only means by which the object it points to can be accessed. The object restricts links that can only initially be accessed via a pointer qualified by it. Restrict is a new type of qualifier that is only added in C99.

▼ Type Conversion in C++

Implicit Type Conversion

Also known as ‘automatic type conversion’.

- Done by the compiler on its own, without any external trigger from the user.
- Generally takes place when in an expression more than one data type is present. In such condition type conversion (type promotion) takes place to avoid lose of data.
- All the data types of the variables are upgraded to the data type of the variable with largest data type.

```
bool -> char -> short int -> int ->

unsigned int -> long -> unsigned ->

long long -> float -> double -> long double
```

- It is possible for implicit conversions to lose information, signs can be lost (when signed is implicitly converted to unsigned), and overflow can occur (when long long is implicitly converted to float).

Example of Type implict Conversion

```
int x = 10; // integer x
    char y = 'a'; // character c

    // y implicitly converted to int. ASCII
    // value of 'a' is 97
    x = x + y;

    // x is implicitly converted to float
    float z = x + 1.0;

    cout << "x = " << x << endl
         << "y = " << y << endl
         << "z = " << z << endl;
```

Explicit Type Conversion: This process is also called type casting and it is user-defined. Here the user can typecast the result to make it of a particular data type.

In C++, it can be done by two ways:

- **Converting by assignment:** This is done by explicitly defining the required type in front of the expression in parenthesis. This can be also considered as forceful casting.

  Syntax:

  ```
  (type) expression
  ```

  where *type* indicates the data type to which the final result is converted.

  ***** More Away : Conversion using Cast operator**

  which forces one data type to be converted into another data type.

  C++ supports four types of casting:

  1. **Static Cast**
  2. Dynamic Cast
  3. **Const Cast**
  4. **Reinterpret Cast**

▼ Type Cast

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:

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

1. Static cast >> very populer and save

```
int a=10;
    double y=static_cast<double>(a);
    cout<<y;
    float d=100.11;
```

```
    int vv=static_cast<int>(d);
    cout<<vv;
    cout<<typeid(a).name(); // this method show witch
```

next added data

▼ Ranged Based Loop

Range-based for loop in C++ has been added since C++ 11. It executes a for loop over a range. Used as a more readable equivalent to the traditional for loop operating over a range of values, such as all elements in a container.

```
for ( range_declaration : range_expression )
    loop_statement
Parameters :
range_declaration :
a declaration of a named variable, whose type is the
type of the element of the sequence represented by
range_expression, or a reference to that type.
Often uses the auto specifier for automatic type
deduction.
range_expression :
any expression that represents a suitable sequence
or a braced-init-list.
loop_statement :
any statement, typically a compound statement, which
is the body of the loop.
```

```
// Illustration of range-for loop// using CPP code#inclu

// Driverint main()
{
    // Iterating over whole array
    vector<int> v = { 0, 1, 2, 3, 4, 5 };
    for (auto i : v)
        cout << i << ' ';
```

```
    cout << '\n';

    // the initializer may be a braced-init-list
     for (int n : { 0, 1, 2, 3, 4, 5 })
         cout << n << ' ';

    cout << '\n';

    // Iterating over array    int a[] = { 0, 1, 2, 3, 4
    for (int n : a)
        cout << n << ' ';

    cout << '\n';

    // Just running a loop for every array    // element
        cout << "In loop" << ' ';

    cout << '\n';

    // Printing string characters
   string str = "Geeks";
    for (char c : str)
        cout << c << ' ';

    cout << '\n';

    // Printing keys and values of a map
    map<int, int> MAP({ { 1, 1 }, { 2, 2 }, { 3, 3 } });
    for (auto i : MAP)
        cout << '{' << i.first << ", " << i.second << "}
}
```

▼ Manipulators in C++

1. **Manipulators without arguments**: The most important manipulators defined by the **IOStream library** are provided below.
    - **endl**: It is defined in ostream. It is used to enter a new line and after entering a new line it flushes (i.e. it forces all the output written on the screen or in the file) the output stream.
    - **ws**: It is defined in istream and is used to ignore the whitespaces in the string sequence.
    - **ends**: It is also defined in ostream and it inserts a null character into the output stream. It typically works with std::ostrstream, when the associated output buffer needs to be null-terminated to be processed as a C string.
    - **flush**: It is also defined in ostream and it flushes the output stream, i.e. it forces all the output written on the screen or in the file. Without flush, the output would be the same, but may not appear in real-time. **Examples:**

▼ Pointer & References All

Pointer :

Pointers are symbolic representation of addresses. They enable programs to simulate call-by-reference as well as to create and manipulate dynamic data structures. It's general declaration in C/C++ has the format:

Pointer variable is point value.

```
  int v=10;
int *ptr;
ptr=&v; // ptr stored address or location  like 0x7f
                  // and *ptr stored value of v like
cout<<v<<" "<<ptr<<" "<<*ptr;

OR Method:
int x=10;
int *ptr=&x;
cout<<*ptr<<" "<<ptr;
```

Real Life use :

- To pass arguments by reference: Passing by reference serves two purposes
- For accessing array elements: The Compiler internally uses pointers to access array elements.
- To return multiple values: For example in returning square and the square root of numbers.
- Dynamic memory allocation: We can use pointers to dynamically allocate memory. The advantage of dynamically allocated memory is, that it is not deleted until we explicitly delete it.
- To implement data structures.
- To do system-level programming where memory addresses are useful.

```
class A{
    public:
    int x;
    void thiskey(int x){
        this->x=x;
        cout<<this->x;
    }
    or
    public:
    int p;
    void thiskey(int x){
        this->p=x;
        cout<<this->p;
    }
};
int main(){
    A obj;
    obj.thiskey(10);
}
```

This Keyword:

> *when function parameter and member variable name is same use this keyword or other perpouse*

Reference :

```
  int y=20;
    int &myref=y;
    //y=100;
    myref=255;
    cout<<myref<<" "<<y;
```

> *কোন ভেরিয়েবল অন্য কোন ভেরিয়েবল কে রেফারেন্স হিসেবে ব্যবহার করলে,যেকোনো একটি ভ্যারিয়েবল এর ভ্যালু চেঞ্জ করলে উভয় varriable value changed hoy ,reference variable and normal variable both changed*

Note *A pointer can be declared as void but a reference can never be void*
References cannot be NULL. Pointers are often made NULL to indicate that they are not pointing to any valid thing.

```
int *ptr=NULL;
    cout<<ptr; //valid

    int &ref=NULL: //code Error
    cout<<ref;
```

> ***Note*** *NULL vs Uninitialized pointe**r** – An uninitialized pointer stores an undefined value. A null pointer stores a defined value, but one that is defined by the environment to not be a valid address for any member or object.*
>
> *NULL vs Void Pointer – Null pointer is a value, while void pointer is a type*

```
int *ptr = nullptr;

   // Below line compiles
   if (ptr) { cout << "true"; }
   else { cout << "false"; }
}
```

Parameter Passing By Pointer

the memory location (address) of the variables is passed to the parameters in the function, and then the operations are performed. It is also called the **call by pointer** method

```
void swap(int *x,int *y){
    int t=*x;
    *x=*y;
    *y=t;
    cout<<*x<<" "<<*y; //print swap value
    cout<<x<<" "<<y; //print Address
}
int main()
{
    int a=10,b=20;
    swap(&a,&b);
}
```

Parameter Passing By Reference

It allows a function to modify a variable without having to create a copy of it. We have to declare reference variables. The memory location of the passed variable and parameter is the same and therefore, any change to the parameter reflects in the variable as well.

```
void swap(int &x,int &y){
    int t=x;
    x=y;
```

```
    y=t;
    cout<<x<<" "<<y;

}
int main()
{
    int a=10,b=20;
    swap(a,b);
}
```

> *here &x and &y variable value is do not copy from a and b ,direct memory location access*

Which is preferred, Passing by Pointer Vs Passing by Reference in C++?

- References are usually preferred over pointers whenever we don't need "reseating".
- If we want to use NULL in our function arguments, prefer pointers.

Some Diffarence:

1. `p = &x;`

   `p = &y; //Pointer reinitialization allowed`

   `int & r = x;`

   `&r = y; // Compile Error (reinitialization not allowed`)

2. `p = NULL;//null supported`

   `// &r = NULL; //null is not support`

3. arrow operator:

   `demo* q = &d;`

   `demo& qq = d;`

   `q->a = 8;//this is valid;`

`// q.a = 8; // 5. Compile Error`

`qq.a = 8;`

`// qq->a = 8;//not valid;`

the point is (→) operator use pointer and qq.a (.) operator use a reference

When do we pass arguments by pointer?

This allows programmers to change the actual data from the function and also improve the performance of the program.]\

1. To Modify Local Variables of the Function

```
void  fun(int x)
{
    // definition of function
    x = 30;
}

int main()
{
    int x = 20;
    fun(x);
    cout << "x = "<<x;
    return 0;
}
//A pointer allows the called function to modify a lo
```

2. For Passing Large-Sized Arguments

```
include <stdio.h>
#include <string.h>

// Structure to represent Employee
struct Employee {
    char name[50];
    char desig[50];
};
```

```
// Function to print Employee details
void printEmpDetails(const struct Employee* emp)
{
    printf("Name: %s\n", emp->name);
    printf("Designation: %s\n", emp->desig);
}

int main()
{
    // Creating an instance of Employee
    struct Employee emp1;
    strcpy(emp1.name, "geek");
    strcpy(emp1.desig, "Software Engineer");

    printEmpDetails(&emp1);

    return 0;
}
```

Note *This point is valid only for struct as we don't get any efficiency advantage for basic types like int, char, etc.*

3. To Achieve Run Time Polymorphism in a Function

4.To Modify the Content of Dynamically Allocated Memory.

> 4 and 3 more information link : https://www.geeksforgeeks.org/when-do-we-pass-arguments-by-reference-or-pointer/?ref=lbp

Diffarence pointer and reference :

Parameters	Pass by Pointer	Pass by Reference
Passing Arguments	We pass the address of arguments in the function call.	We pass the arguments in the function call.
Accessing Values	The value of the arguments is accessed via the dereferencing operator *	The reference name can be used to implicitly reference a value.

Reassignment	Passed parameters can be moved/reassigned to a different memory location.	Parameters can't be moved/reassigned to another memory address.
Allowed Values	Pointers can contain a NULL value, so a passed argument may point to a NULL or even a garbage value.	References cannot contain a NULL value, so it is guaranteed to have some value.

▼ Array

Array decay:

The loss of type and dimensions of an array is known as decay of an array.This generally occurs when we pass the array into function by value or pointer. What it does is, it sends first address to the array which is a pointer, hence the size of array is not the original one, but the one occupied by the pointer in the memory.

```
void ardecy(int *p){
    cout<<sizeof(p);
}
int main(){
    int a[]={1,2,3,4};
    //here actual array size=16
    cout<<"Actual:"<<sizeof(a)<<endl;
    //here passing array size=8
    cout<<"From ardecy function:";
    ardecy(a);
}
```

prevent Array Decay :

A typical solution to handle decay is to pass size of array also as a parameter and not use sizeof on array parameters . Another way to prevent array decay is to send the array into functions by reference. This prevents conversion of array into a pointer, hence prevents the decay.

```
void ardecy(int (&p)[4]){
    cout<<sizeof(p);
```

```
}
int main(){
    int a[4]={1,2,3,4};
    //here actual array size=16
    cout<<"Actual size:"<<sizeof(a)<<endl;
    //here passing array size= 16
    cout<<"From ardecy function size:";
    ardecy(a);
}
another Process normalLy:
void ary(int *p,size){
    //code
}
main(){
a[7]={data}
ary(a,7);
}
```

> *pass reference and size in called function*

Array Pointer :

```
data_type (*var_name)[size_of_array];
int *ptr[10];
```

Here:

- **data_type** is the type of data that the array holds.
- **var_name** is the name of the pointer variable.
- **size_of_array** is the size of the array to which the pointer will point.

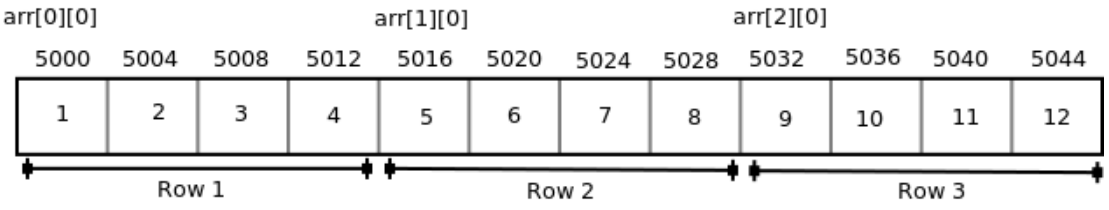
Note : pointer that points to the 0th element of array and the pointer that points to the whole array are totally different.

```
#include<stdio.h>

int main()
```

```
{
    // Pointer to an integer
    int *p;

    // Pointer to an array of 5 integers
    int (*ptr)[5];
    int arr[5];

    // Points to 0th element of the arr.
    p = arr;

    // Points to the whole array arr.
    ptr = &arr;

    printf("p = %p, ptr = %p\n", p, ptr);

    p++;
    ptr++;

    printf("p = %p, ptr = %p\n", p, ptr);
    return 0;
}
```

2D Array in Pointer :

2D array do not understand memory,memory jast understand 1D array,The concept of rows and columns is only theoretical, actually, a 2-D array is stored in row-major order i.e rows are placed next to each other. The following figure shows how the above 2-D array will be stored in memory.

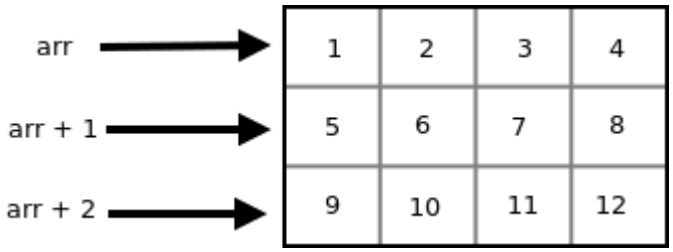


Each row can be considered as a 1-D array, so a two-dimensional array can be considered as a collection of one-dimensional arrays that are placed one after another. In other words, we can say that 2-D

dimensional arrays that are placed one after another. So here *arr* is an array of 3 elements where each element is a 1-D array of 4 integers.

We know that the name of an array is a constant pointer that points to 0th 1-D array and contains address 5000. Since *arr* is a 'pointer to an array of 4 integers', according to pointer arithmetic the expression arr + 1 will represent the address 5016 and expression arr + 2 will represent address 5032.

So we can say that *arr* points to the 0th 1-D array, *arr + 1* points to the 1st 1-D array and *arr + 2* points to the 2nd 1-D array.



arr - Points to 0th element of arr - Points to 0th 1-D array - 5000
arr + 1 - Points to 1th element of arr - Points to 1nd 1-D array - 5016
arr + 2 - Points to 2th element of arr - Points to 2nd 1-D array - 5032

3D and more information Link : https://www.geeksforgeeks.org/pointer-array-array-pointer/?ref=lbp

▼ Structure and Union

Structure :

Structures are used to combine different types of data types, just like an **array** is used to combine the same type of data types.

```
struct check
{
    int i;
    float f;
    string s;
};

int main(){
    // strcture declaration
```

```
    //hear all data transfer check to a variable
      // like inhertace
    struct check a; //structure variable
    a.f=10.10;
    a.i=10;
    a.s="sss";
    cout<<a.f<<"\n";
    cout<<a.i<<"\n";
    cout<<a.s<<"\n";
}
```

Pointer Structure :

*In the above code g is an instance of struct point and **ptr** is the struct pointer because it is storing the address of struct point.*

```
struct point {
    int value;
};
int main()
{
    struct point g;
    // Initialization of the structure pointer
    struct point* ptr = &g;
    return 0;
```

Use Array In Structure:

very usefulll that and very impotant it.

```
 struct person
{
    int age;
    float gpa;
}strucname;

int main(){

    struct person ar[2];
```

```
    int i;

    for (int  i = 0; i < 2; i++)
    {
        cout<<"Age:";
        cin>>ar[i].age;

        cout<<"Gpa:";
        cin>>ar[i].age;
    }

    cout<<"Ouput:"<<"\n";
    for (int  i = 0; i < 2; i++)
    {
        cout<<"Age:";
        cout<<ar[i].age<<"\n";

        cout<<"Gpa:";
        cout<<ar[i].age;
    }

}
```

Structure variable pass to Function :

```
struct person
{
    int age;
    float gpa;
};
void fun(struct person p ){
    cout<<p.age<<"\n";
    cout<<p.gpa<<"\n";
}

int main(){
```

```
    struct person var_name1,var_name2;

    cout<<"Person-1"<<"\n";
    var_name1.age=10;
    var_name1.gpa=3.40;
    fun( var_name1);

  cout<<"Person-2"<<"\n";
    var_name2.age=20;
    var_name2.gpa=3.50;
    fun(var_name2);
}
```

Nested Structure :

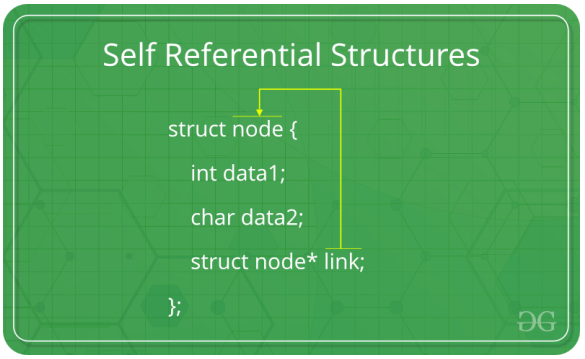
```
struct innerStructName{
     //define inner structure
};
struct outerStructName{
struct innerStructName st; // inner struct as Direct
member
};

OR We can also declare it directly inside the parent
structure:
structouterStructName {
    structinnerStructName {
  //define inner structure
    };
};
```

Code :

```
struct employ{
    int id;
    int age;
};
struct boss{
```

```
    int bossage; //information of boss
    struct employ em_information;
    //hear boss hlod all feture in employ
};
void dispaly(struct boss a){
    cout<<a.bossage<<"\n";
    cout<<a.em_information.age<<"\n";
    cout<<a.em_information.id<<"\n";
}
int main()
{
    //Creating an instance of the boss
    struct boss mystruc;
    //then Access all data
    mystruc.bossage=10;
    mystruc.em_information.age=10;
    mystruc.em_information.id=220215;

    dispaly(mystruc);
}
```

Referential Structures:



more information :

https://www.geeksforgeeks.org/self-referential-structures/

Typedef:

The typedef is used to give data type a new name. thats means user at will data type.

```
typedef int a;
    a t=10;
    cout<<t;
more :

typedef int *ptr;
    ptr a,b;
    cout<<sizeof(a)<<"\n";
    cout<<sizeof(b)<<"\n";
```

use typedef structure:

```
typedef struct math
{
    int a;
    float b;
}strucname;

int main(){
        //structure object initilization
    strucname demo;
    demo.a=10;
    demo.b=10.22;
    cout<<demo.a<<" " <<demo.b;
```

Self Referential structures :

Self Referential structures are those **structures** that have one or more pointers which point to the same type of structure, as their member.

```
```

Union :

A union is a type of structure that can be used where the amount of memory used is a key factor

```
union check
{
    int i;
    float f;
    string s;
};

int main(){
    // strcture declaration
   union check a;
    a.f=10.10;
    a.i=10;
    a.s="sss";
    cout<<a.f<<"\n";
    cout<<a.i<<"\n";
    cout<<a.s<<"\n";
}
```

Enum: Enums are user-defined types that consist of named integral constants.

```
#include <bits/stdc++.h>
using namespace std;
// Defining  an enum
enum GeeksforGeeks { Geek1 };
GeeksforGeeks G1 = Geek1;
// Driver Code
int main()
{
    cout << "The numerical value "
         << "assigned to Geek1 : " << G1 << endl;
 }
```

https://www.geeksforgeeks.org/self-referential-structures/

▼ Class and Access Modifire

What is object ?

object is instance of a class. object is represent classes all attribute. object means class of container or box.

```
class A{
int a,b,c;
int math(){};
};

int main(){

    A obj;
    //this obj store or hold or instance or represent A
}
```

class is user-defined data type,its holds its own data members and member functions,

```
#include<bits/stdc++.h>
using namespace std;

class my_class{

    public:
    int a; //this is a Public attribeute
    string s;

    private:
    int b; //this is a Private attribeute


    protected:
    int c; //this is a Protected attribeute

    public : // public method
        int math(){
            int d=10;
            cout<<d<<" ";
            return 0;
```

```
            b=10;
            cout<<b;
        }
    void display(){ //this is a public method
            cout<<a<<" ";
            cout<<s;
                }

    private:
        int math1(){
            int c=10;
            cout<<c;
            return 0;
        }

    protected:
        int math2(){
            int c=10;
            cout<<c;
            return 0;
        }

};

int main(){
  my_class obj;
  obj.a=50; // object value Assgine
  obj.s="direct access bescuse this is public";
  obj.math(); //access direct public method
  obj.math1();
  obj.display();

}
```

Member Functions in Classes:

- Inside class definition
- Outside class definition

```
class my_class{
    //member function use inside
    public:
        int math1(){
            cout<<"Inside function Access"<<"\n";
            return 0;
        }
    //jast member function defination and access
    // outside in class
    public:
    int math2();
};
int my_class::math2(){
    cout<<"outside function Access";
    return 0;
}

int main(){
  my_class obj;
  obj.math1();
  obj.math2();
}
```

Access Modifiier :

Real Life Example :

The Research and Analysis Wing (R&AW), having 10 core members, has come into possession of sensitive confidential information regarding national security. Now we can correlate these core members to data members or member functions of a class, which in turn can be correlated to the R&A Wing. These 10 members can directly access the confidential information from their wing (the class), but anyone apart from these 10 members can’t access this information directly, i.e., outside functions other than those prevalent in the class itself can’t access the information (that is not entitled to them) without having either assigned privileges (such as those possessed by a friend class or an inherited class, as will be seen in this article ahead) or access to one of these 10 members who is allowed direct access to the confidential

information (similar to how private members of a class can be accessed in the outside world through public member functions of the class that have direct access to private members). This is what data hiding is in practice.

Access Modifiers or Access Specifiers in a **class** are used to assign the accessibility to the class members, i.e., they set some restrictions on the class members so that they can't be directly accessed by the outside functions.
Attributes can have different access specifiers:

- `private`: Attributes declared as private can only be accessed by member functions in this class. They are not accessible from outside the class. when i went access this attribute use **friend functions**

  However, we can access the private data members of a class indirectly using the public member functions of the class.

- `protected`: Attributes declared as protected can be accessed by member functions of the same class and its derived classes.

  Note: This access through inheritance can alter the access modifier of the elements of base class in derived class depending on the **mode of Inheritance**.

  use inheritace

- `public`: Atributes declared as public are accessible from anywhere in the program.

> ***Note**: If we do not specify any access modifiers for the members inside the class, then by default the access modifier for the members will be **Private**.*

Code Hear :

```
// Private Modifier use example useing member function:

class my_class{
    private:
    int private_variable=10;
```

```
    void A(){
        cout<<"private function"<<"\n";
    }
    public:
    int x=200;
    int math(){
        int public_variable=20;
        cout<<public_variable<<"\n";
        //here private attribute Access
        //becuse this is member function
        cout<<private_variable<<"\n";
        A();
        return 0;
    }
};

int main(){
  my_class obj;
  //public Attribute data access
  obj.math();
}

//

//Protecteted modifire by using inheritence Example:
class base{
    protected:
    int protecd_data=10;
};

class drive:public base{
    public:
    int drive_data=20;
    void dis(){
        cout<<drive_data<<"\n";
        /*
        use inheritance and access
        protected data
```

```
        */
        cout<<protecd_data<<"\n";
    }
};
int main(){
  drive obj;
  obj.dis();
}
```

Nested Class :

When declear Class inside another class,that is nested class.must be inerclass declear in public.

Class object create in main function

> Outer_Class_name : : iner_classname object_name

```
#include<iostream>
using namespace std;
class A {
   public:
   class B {
      private:
      int num;
      public:
      void getdata(int n) {
         num = n;
      }
      void putdata() {
         cout<<"The number is "<<num;
      }
   };
};
int main() {
   cout<<"Nested classes in C++"<< endl;
   A :: B obj;
   obj.getdata(9);
```

```
   obj.putdata();
   return 0;
}
```

Local Class :

Function inside class declaration, that is local class.

```
void fun(){
    class A{

    };
}
```

- 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.

Fact :

1.A local class type name can only be used in the enclosing function.

```
void fun(){
    class A{
    public :
    int x;
    //data
    };
    A obj; //fine
    obj.x; //fine
}
int main(){
   A obj; // error
}
```

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

```
void fun(){
    class A{
    public :
    int x;
    //data
    };
   void A::x=10; // error must be this work class inside
}
```

3.A Local class cannot contain static data members. It may contain static functions though

```
void fun(){
    class A{
    public :
    static int x; //error
    static void dis(){ //supported
    cout<<"ok";
    }
    //data
    };
   void A::dis();
}
int main(){
fun();
}
```

4.Member methods of the local class can only access static and enum variables of the enclosing function. Non-static variables of the enclosing function are not accessible inside local classes.

```
void fun(){
     int x; //not accessible for Local class
     static int y=10; //accessible local class
     enum { i = 1, j = 2 };//accessible local class
    class A{
    public :
     void dis(){
```

```
     cout<<y;
     cout<<i<<j;
     cout<<x; //error
    }
    };

}
int main(){
fun();
}
```

5.Local classes can access global types, variables, and functions. Also, local classes can access other local classes of the same function. For example, the following program works fine.

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

int x;

void fun()
{

    // First Local class
    class Test1 {
    public:
        Test1() { cout << "Test1::Test1()" << endl; }
    };

    // Second Local class
    class Test2 {
        // Fine: A local class can use other local class
        // of same function
        Test1 t1;

    public:
        void method()
        {
```

```
            // Fine: Local class member methods can acce
            // global variables.
            cout << "x = " << x << endl;
        }
    };

    Test2 t;
    t.method();
}

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

Enum Classes in C++ and Their Advantage over Enum DataType :

```
#include <iostream>
using namespace std;
enum roll_no {
    satya = 70,
    aakanskah = 73,
    sanket = 31,
    aniket = 05,
    avinash = 68,
    shreya = 47,
    nikita = 69,
};
int main()
{
    enum roll_no obj;
    obj = avinash;
    cout << "The roll no  of avinash=" << obj;
}
```

Class and stricture difference:

Class	Structure
1. Members of a class are private by default.	1. Members of a structure are public by default.
2. An instance of a class is called an ‘object’.	2. An instance of structure is called the ‘structure variable’.
3. Member classes/structures of a class are private by default but not all programming languages have this default behavior eg Java etc.	3. Member classes/structures of a structure are public by default.
4. It is declared using the **class** keyword.	4. It is declared using the **struct** keyword.
5. It is normally used for data abstraction and further inheritance.	5. It is normally used for the grouping of data
6. NULL values are possible in Class.	6. NULL values are not possible.

7. **Syntax:**
class class_name{
data_member;
member_function;
};
	7. **Syntax:**
	struct structure_name{
	type structure_member1;
	type structure_member2;
	};

Syntax:

```
class ClassName {
     int x; //by default private
private:
    member1;
    member2;

public:
    member3;
    .
    .
    memberN;
};
```

Syntax:

```
struct StructureName {
    int x; //by default public
    member1;
    member2;
    .
    .
    .
    memberN;
};
```

Ex-1:

```
#include <iostream>

using namespace std;

class Test {
    // x is private
    int x;
};
struct Testi{
    int x;
    // x is public
};


int main()
{
    Test t;
    // compiler error because x
    // is private
    t.x = 20;

    Testi t; //structure object
    t.x=20; // valid
```

```
}
```

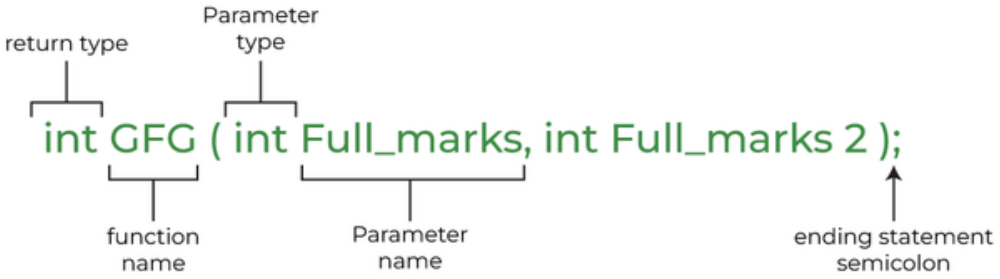
Inheritance of structure :

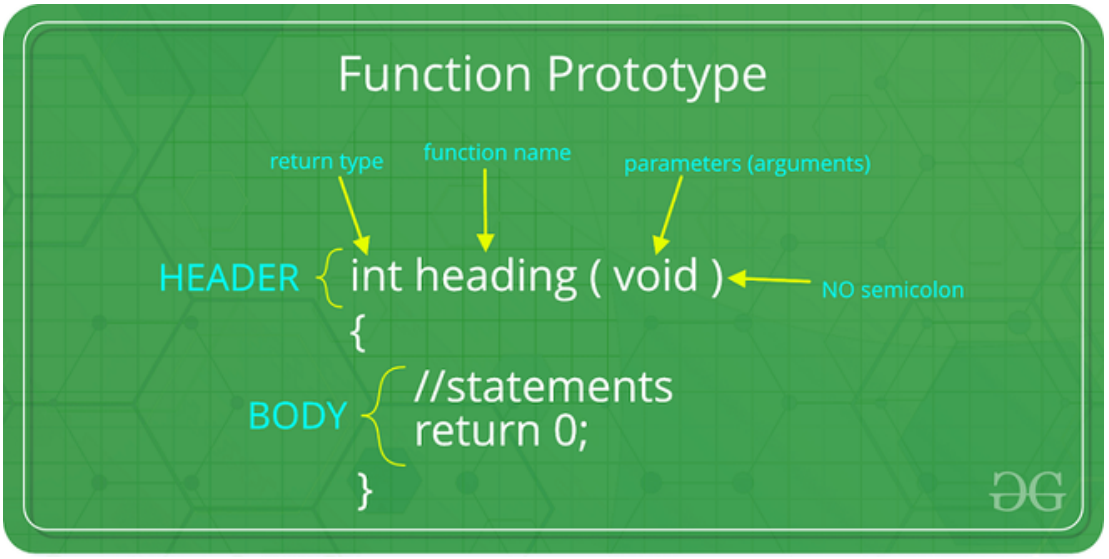
```
using namespace std;

struct Base {
public:
    int x;
};

// is equivalent to
// struct Derived : public Base {}
struct Derived : Base {
public:
    int y;
};

int main()
{
    Derived d;

    // Works fine because inheritance
    // is public.
    d.x = 20;
    cout << d.x;
    cin.get();
    return 0;
}
```

▼ OOP Main Part

▼ Function Importanat data

Function Declear

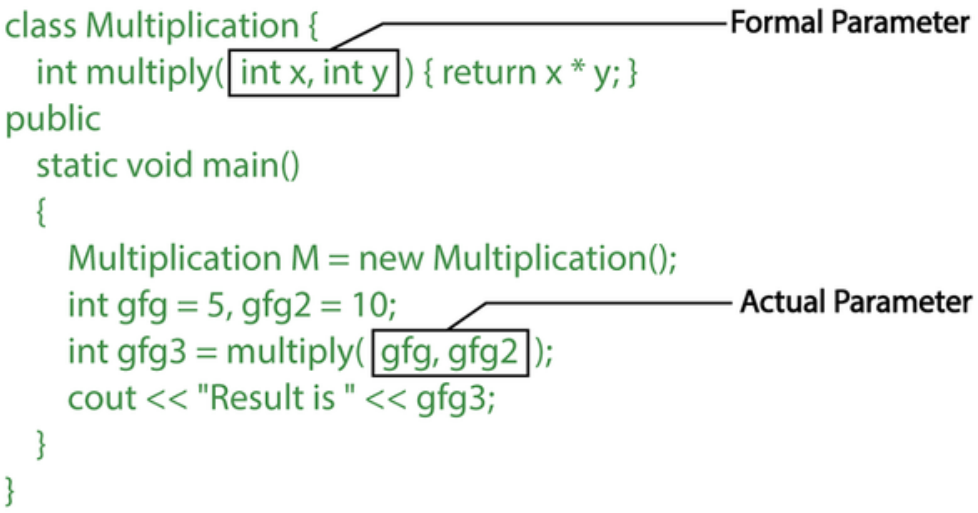




Parameter Passing to Functions

The parameters passed to the function are called ***actual parameters***. For example, in the program below, 5 and 10 are actual parameters.

The parameters received by the function are called ***formal parameters***. For example, in the above program x and y are formal parameters.



Default Argument :

> *When default argument data pass from main function then user defined function default argument data overridden.*

```
fun(a,b,c=0,d=0){return a+b+c+d}
main(){
fun(10,20);//output :30
fun(10,20,15); // c value is overridden
.               so output is : 45
}
}
```

Called function is callee and caller function is caller

Pass by result :

```
int add(int x,int y){
return x+y;
}
int main(){
int resut=add(1,20); // result is hlod add function retu
}
```

Call by Value and Call bty reference

There are two most popular ways to pass parameters:

1. ***Pass by Value:*** In this parameter passing method, values of actual parameters are copied to the function's formal parameters. The actual and formal parameters are stored in different memory locations so any changes made in the functions are not reflected in the actual parameters of the caller.

```
void  fun(int x)
{
    // definition of function
    x = 30;
}

int main()
{
    int x = 20;
    fun(x);
    cout << "x = "<<x;
    return 0;
}
```

2. ***Pass by Reference:*** Both actual and formal parameters refer to the same locations, so any changes made inside the function are reflected in the actual parameters of the caller.

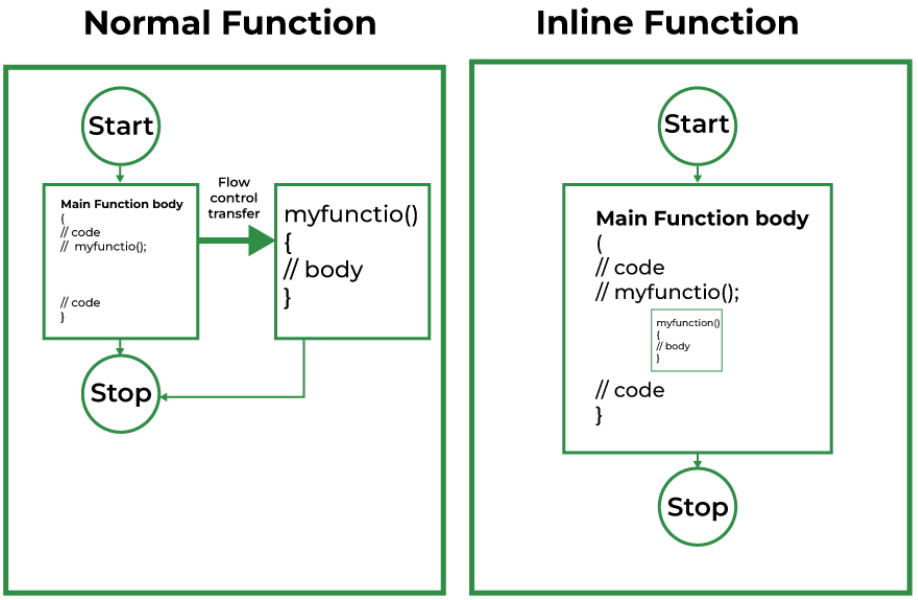
```
void fun(int* ptr){
    *ptr=30;
}

int main(){
    int x=20;
    fun(&x);
    cout<<"x :"<<x;
}
```

> *So Mejor diffrence is call by value >> diffrent memory location formal parameter and actual parameter / callee and caller function. and call by reference >> both value is same location so reflected actual parameter /* caller function ,so calle function any changed thats direct reflect calller function,.

Conclusion:

In conclusion, Call by Value means passing values as copies to the function, so that the original data is preserved and any changes made inside the function are not reflected in the original data, whereas Call by Reference means passing references to the memory locations of variables (in C we pass pointers to achieve call by reference), hence changes made inside the function are directly modified in the original values. The choice between the two completely depends on the particular requirements and considerations of a program.

Inline Functions :



The compiler may not perform inlining in such circumstances as:

1. If a function contains a loop. (*for, while and do-while*)
2. If a function contains static variables.
3. If a function is recursive.
4. If a function return type is other than void, and the return statement doesn't exist in a function body.
5. If a function contains a switch or goto statement

> *function switch is very bad effect in programm . so this bad effect overcame use inline function . no needed jump main function to target function. sumerize*

```
inline int math(){
    int x=y+z;
}
```

Lamda Expretion :

Lambda expressions and closures are powerful features of many modern programming languages, such as Python, JavaScript, and Ruby. They **allow you to write concise, expressive, and flexible code that can handle complex tasks and scenarios**

ল্যাম্বডা এক্সপ্রেশন কি কাজে লাগে?

সংক্ষিপ্ততা: ল্যাম্বডা এক্সপ্রেশনগুলি প্রায়শই প্রচলিত পদ্ধতির ঘোষণার চেয়ে আরও সংক্ষিপ্ত হয়, **যা কোডটিকে পড়তে এবং লিখতে সহজ করে তোলে** । পঠনযোগ্যতা: তারা কোড পঠনযোগ্যতা উন্নত করে, বিশেষ করে যখন LINQ প্রশ্ন এবং কার্যকরী নির্মাণে ব্যবহৃত হয়।

see Bangla tutorial more information >>

https://www.youtube.com/watch?v=hnzIF6Hx-To&ab_channel=as

```
[ capture clause ] (parameters) -> return-type{
definition of method}
```

▼ **Static Member and Function**

1. **Static Variables:** Variables in a function, Variables in a class
2. **Static Members of Class:** Class objects and Functions in a class
3. static variable defult vlaue is zero

Static mermber value do not destroy any momment,this variable value all time stored last updated value.

more :

Static Data Members:

- 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.
- It is visible only within the class, but its lifetime is the entire program.

Syntax:

```
static data_type data_member_name;
```

Static variables in a Function:

When a variable is declared as static, space for **it gets allocated for the lifetime of the program**. Even if the function is called multiple times, space for the static variable is allocated only once and the value of the variable in the previous call gets carried through the next function call. This is useful for implementing **coroutines in C/C++** or any other application where the previous state of function needs to be stored.

> *Satatic varible value fixed and its life time hole programm.and store last update ted value and save it.*

```
#include<bits/stdc++.h>
using namespace std;
int math(){
    int a=0; //normal variable in function
     a++;
    cout<<a<<" ";
    return 0;
}
int math1(){
   static int a=0; //static variable in function
     a++;
    cout<<a<<" ";
    return 0;
}

int main(){
     cout<<"Normal:"<<"\n";
    for (int i = 0; i < 5; i++)
    {
          math(); //when fc called a value is resest
    }
```

```
    cout<<"\n";
    cout<<"Static:"<<"\n";
    for (int i = 0; i < 5; i++)
    {
          math1(); //when fc called a value is stored
          last updated value.
    }

}
```

Static variables in a class:

As the variables declared as static are initialized only once as they are allocated space in separate static storage so, the static variables **in a class are shared by the objects.** There can not be multiple copies of the same static variables for different objects. Also because of this reason static variables can not be initialized using constructors

> A static variable inside a class should be initialized explicitly by the user using the class name and scope resolution operator outside the class . otherise error.example >>> int class_name :: var_name=0;

```
#include<bits/stdc++.h>
using namespace std;

class sta{
    public:
    static int a;
    void dis(){
        a++;
    cout<<a<<" ";
    }

    //normal function
    public:
      int b=0;
    void dis1(){
```

```
        b++;
    cout<<b<<" ";
    }

};
int sta::a=0;


int main(){
    sta obj;
      cout<<"Static:"<<"\n";
    for (size_t i = 0; i < 5; i++)
    {
    obj.dis();
    }

    cout<<"\n";
    cout<<"normal"<<"\n";

    for (size_t i = 0; i < 5; i++)
    {
    obj.dis1();
    }

    //hear nomal and static is same behaveur mejor
     //differce is declaration in satatic var >> int sta


}
```

More example :

```
#include<bits/stdc++.h>
using namespace std;

class abc{
    public:
```

```
    static int a;
    int b;

    abc(){b=0;} //set b value use construnctor
    void fun(){cout<<"a="<<++a<<" "<<"b="<<++b<<"\n";}

};
int abc::a;

int main(){
    abc obj1,obj2;
    obj1.fun(); //a and b value is 1
    obj1.fun(); //a and b previous value is 1,
                            // then increment so both va
    obj2.fun(); /*
                a previous value is 2 then update value
                no reset a value becuse a is static,
                b previous value is 2 then update value
                b value reset becuse when create or
                or use new object instantaneously call c
                contrutor method inside b value =0
                 */
}
```

Class objects as static:

Just like variables, objects also when declared as static have a scope till the lifetime of the program. Consider the below program where the object is non-static.

```
// CPP program to illustrate
// class objects as static
#include <iostream>
using namespace std;

class GfG {
    int i = 0;

public:
```

```
    GfG()
    {
        i = 0;
        cout << "Inside Constructor\n";
    }

    ~GfG() { cout << "Inside Destructor\n"; }
};

int main()
{
    int x = 0;
    if (x == 0) {
        static GfG obj;
    }
    cout << "End of main\n";
}
```

Static functions in a class:

just like the static data members or static variables inside the class, static member functions also do not depend on the object of the class.**Static member functions are allowed to access only the static data members or other static member functions**, they can not access the non-static data members or member functions of the class.

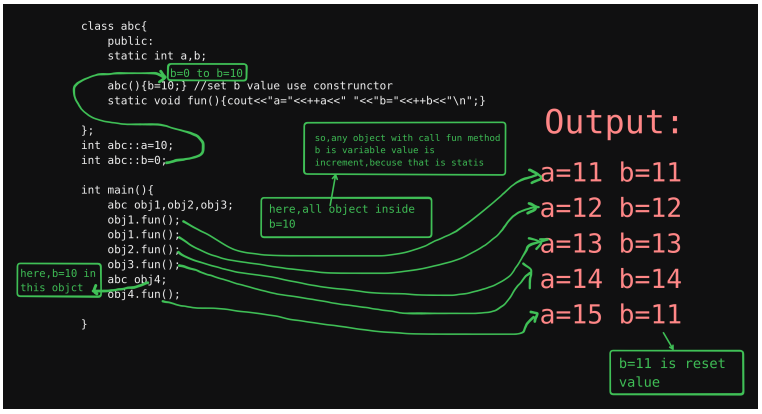
> function call rules >> class_name :: fc_name();

```
#include<bits/stdc++.h>
using namespace std;
class A{
    public:
    // static member function
    static void dis(){
        cout<<"This is statis function";
    }
};
int main(){
```

```
    A obj;
    obj.dis(); //thats work it normal
    cout<<"\n";
    // invoking a static member function
    A ::dis();
}
```

All variable must be static inside static member function in class.

```
#include<bits/stdc++.h>
using namespace std;

class abc{
    public:
    static int a,b;

    abc(){b=0;} //set b value use construnctor
    static void fun(){cout<<"a="<<++a<<" "<<"b="<<++b<<"

};
int abc::a;
int abc::b;
int main(){
    abc obj1,obj2;
    obj1.fun(); //a and b value is 1
    obj1.fun(); //a and b previous value is 1 then incre
    obj2.fun(); /*
                a previous value is 2 then update value
                b previous value is 2 then update value
                when create new object obj2,instantly ca
                vaiable b is static.
                 */
}
```

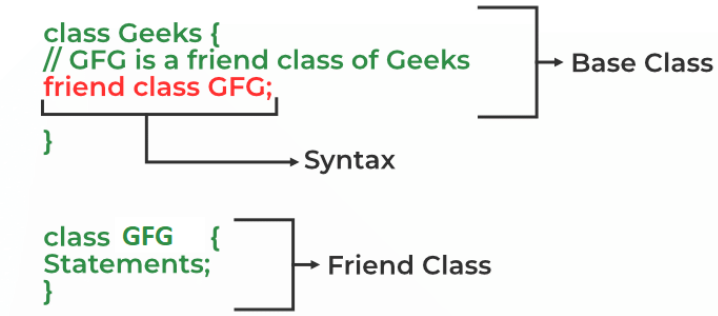


Real life Ex: counting create object in class. or count class

▼ Friend Class and Function

Friend Class :

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.




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

> ***Note:*** *We can declare friend class or function anywhere in the base class body whether its private,*

> *protected or public block. It works all the same. must declear base class inside friend class*

Normal Function use:

```
#include<bits/stdc++.h>
using namespace std;

class my_class{
    //private attribute
    private:
         int pvt_value=10;
        string pvt_name="hasan";

    public:
        //declear friend class
        friend class my_friend;
};

//friend class can access my_classs private attribute
class my_friend{
    //
    public:
    void dis(my_class obj){
        cout<<obj.pvt_value<<"\n";\
        cout<<obj.pvt_name;
    }
};
int main(){
    my_class ob1;
    my_friend ob2;
    /*
    on my_classs object created
    so main function my_class object pass
    */
    ob2.dis(ob1); //this object store all attribute my c
```

```
                                        //that means full class
}
```

Use Contructor Function :

```
#include<bits/stdc++.h>
using namespace std;

class my_class{
    //private attribute
    private:
         int pvt_id;
        string pvt_name;

    public:
        my_class(){
            pvt_name="Pervez";
            pvt_id=202;
        }
        //declear friend class
        friend class my_friend;
};
//friend class can access my_classs private attribute
class my_friend{
    public:
    void dis(my_class obj){
        cout<<obj.pvt_id<<"\n";
        cout<<obj.pvt_name<<"\n";
    }
};

int main(){
    my_class ob1;
    my_friend ob2;
    /*
    on my_classs object created
    so main function my_class object pass
    */
```

```
    ob2.dis(ob1);
}
```

One class another class use friend function :

```
class num1{
    private:
    int n1;

    public:
    num1(){
        n1=50;
    }


    friend class num2;
};

class num2{
    private:
    int n2;

    public:
    num2(){
        //value set n2
        n2=65;
    }

    int avg(){
       //num2 friend class of num1 class
       //num1 class all data is access is permited
       //can be all data access num2 from num1
       //restriction: is all time create object this cla
        num1 obj;
        return (obj.n1+n2)/2;
    }
```

```
    };

    int main(){
        num2 ob2;
        cout<<ob2.avg();
    }
```

can not solved Problem :

```
// Do not solved;
 //set and get fun() not solved
```

> *when friend class use private data from main class ,must be create main class object inside parameter*

Friend Function:

Friend Function Like a friend class. friend function use non member function and member function both case use

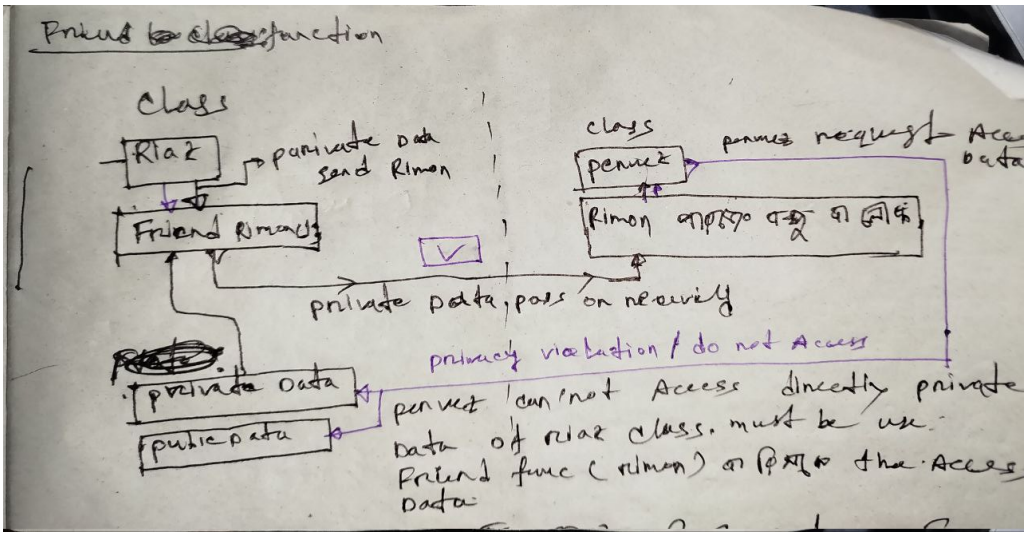
Key Points:

- A friend function is not a member of the class.
- It is declared inside the class but defined outside the class.
- It has access to all private and protected members of the class.
- It can be a regular function or even a member of another class.

When to Use Friend Functions:

- When you need to allow non-member functions or other classes to access the private members of a class.
- It provides controlled access to private data while keeping encapsulation intact.

```
friend datatype funcname(argument);
friend int gc(claas obj);
```



A friend function can be:

1. A global function
2. A member function of another class

USE CASE :

- A member function of another class as use as friend function

```
class base;
class anotherclass{
    public:
    //this is member function of another class or thi
    void memberfunc(base obj);
    /* Note:
         that  means another class can access,
        the data of the friend function inside the ba
        do not access all data, jast access friend fu
    */

};

class base{
```

```
    private:
    int pvt;

    protected:
    int prot;

    public:
    int a=20;
    base(){
        //value assign or set value
        pvt=10;
        prot=20;
    }

    //declear friend function
   void friend  anotherclass::memberfunc(base obj);

   /*
    now we access all attribute from class base
   becuse memberfunction is friend funucton of base c
   so all data access memberfunction inside base clas
   */

};

//friend function defination
void anotherclass::memberfunc(base obj)
{
    cout<<obj.prot<<"\n";
    cout<<obj.pvt;
}


int main(){
    base obj1;
    anotherclass obj2;

    //call member function,this is friends function
```

```
    obj2.memberfunc(obj1);
    /* agument obj1 pass becuse friend function
     argument is base class object.
    */
}
```

- A Function Friendly to Multiple Classes (use non member function). multiple class and and one friend function use

```
class B;
class A{
    private:
    int x;

    public:
    void assign(int i){
        x=i;
    }

    friend void product(A,B);
};

class B{
    private:
    int y;
    public:
    void assign(int i){
        y=i;
    }
    friend void product(A,B);
};

// non member function or public function
void product(A obja,B objb){
    cout<<obja.x*objb.y;
}

int main(){
```

```
    A ob1;
    ob1.assign(45); // value assign of x variable of
                    // this is a setter function so v
    B ob2;
    ob2.assign(45); // y value assign of class B,that

    //output print so call product method.
    //product function like getX() function.
    product(ob1,ob2);

}
```

Global function use friends function:

```
// global function here
class my_class{
    private:
    int salary;
    //friend function Declaration
    friend int friendfunc(my_class s);
};

//friend function defination
//this is global function
int friendfunc(my_class s)
{
    s.salary=1000;
    return s.salary;
}


int main(){
    my_class obj;
    cout<<friendfunc(obj);
}
```

Features of Friend Functions :

- A friend function is a non-member function or ordinary function of a class
- A friend function can be declared in any section of the class i.e. public or private or protected.

Advantage:

- 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.

Disadvantage:

- 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.
- 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.

Important Points About Friend Functions and Classes

1. Friends should be used only for limited purposes. Too many functions or external classes are declared as friends of a class with protected or private data access lessens the value of encapsulation of separate classes in object-oriented programming.
2. Friendship is **not mutual**. If class A is a friend of B, then B doesn't become a friend of A automatically.
3. Friendship is not inherited. (See **this** for more details)
4. The concept of friends is not in Java.

▼ Some important information

- when parivate variable data set must use public function.this function inside to private variable data set
-

▼ Enumeration

is a user defind data type,that consists of integral costants.

enum like structure.

usecase : when work more constant in program,use enum.

Example:

```
enum roll_no{
   pervez=102023, //pervez value is 102023
   hasan, //hasan value is 102024
   khan //hasan value is 102025 here,
         // hasan and khan auto store roll no,and increm

};

int main(){
   roll_no obj,obj1;
   obj=hasan;
   obj1=khan;
   cout<<obj<<" "<<obj1;
}
```

More Example:

```
enum captain{
   jak,mak,pak
};

int main(){
   captain cap=jak;

   if(cap==mak){
       cout<<"he is captain";
   }
   else{
      cout<<"he is not captain";
   }
return 0;
}
```

```
int main(){
   enum color{
   red,green,blue
   };
   color select=green;

   switch(select){
      case red:
         cout<<"that color is red"<<"\n";
         break;
      case green:
         cout<<"that color is green"<<green<<"\n";
         break;
      default:
         cout<<"blue";
   }
}
```

Enumeration class :

is a user-defined data type that can be assigned some limited values. These values are defined by the programmer at the time of declaring the enumerated type.

Syntax:

```
// Declaration
enum class EnumName{ Value1, Value2, ... ValueN};

// Initialisation
EnumName ObjectName = EnumName::Value;
```

Example:

```
// Declaration
enum class Color{ Red, Green, Blue};

// Initialisation
Color col = Color::Red;
```

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

int main()
{

    enum class Color { Red,
                       Green,
                       Blue };
    enum class Color2 { Red,
                        Black,
                        White };
    enum class People { Good,
                        Bad };

    // An enum value can now be used
    // to create variables
    int Green = 10;

    // Instantiating the Enum Class
    Color x = Color::Green;

    // Comparison now is completely type-safe
    if (x == Color::Red)
        cout << "It's Red\n";
    else
        cout << "It's not Red\n";

    People p = People::Good;

    if (p == People::Bad)
```

```
        cout << "Bad people\n";
    else
        cout << "Good people\n";

    // gives an error
    // if(x == p)
    // cout<<"red is equal to good";

    // won't work as there is no
    // implicit conversion to int
    // cout<< x;

    cout << int(x);

    return 0;
}
```

limitation

▼ Function Overloading

constructor overloading

- 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.

```
// C++ program to illustrate
// Constructor overloading
#include <iostream>
using namespace std;

class construct
{

public:
```

```
    float area;

    // Constructor with no parameters
    construct()
    {
        area = 0;
    }

    // Constructor with two parameters
    construct(int a, int b)
    {
        area = a * b;
    }

    void disp()
    {
        cout<< area<< endl;
    }
};

int main()
{
    // Constructor Overloading
    // with two different constructors
    // of class name
    construct o;
    construct o2( 10, 20);

    o.disp();
    o2.disp();
    return 1;
}
```

Functions that cannot be overloaded

- Function declarations that differ only in the return type. For example, the following program fails in compilation.

```
#include<iostream>
int foo() {
return 10;
}

char foo() {
return 'a';
}

int main()
{
char x = foo();
getchar();
return 0;
}
```

- Member function declarations with the same name and the name parameter-type-list cannot be overloaded if any of them is a static member function declaration. For example, following program fails in compilation.

```
#include<iostream>
class Test {
static void fun(int i) {}
void fun(int i) {}
};

int main()
{
Test t;
getchar();
return 0;
}
```

- Parameter declarations that differ only in a pointer * versus an array [] are equivalent. That is, the array declaration is adjusted to become a pointer declaration. Only the second and subsequent array

dimensions are significant in parameter types. For example, following two function declarations are equivalent.

```
int fun(int *ptr);
int fun(int ptr[]); // redeclaration of fun(int *ptr)
```

- Parameter declarations that differ only in that one is a function type and the other is a pointer to the same function type are equivalent.

```
void h(int ());
void h(int (*)()); // redeclaration of h(int())
```

- Parameter declarations that differ only in the presence or absence of const and/or volatile are equivalent. That is, the const and volatile type-specifiers for each parameter type are ignored when determining which function is being declared, defined, or called. For example, following program fails in compilation with error *“redefinition of `int f(int)’ “*

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

using namespace std;

int f ( int x) {
    return x+10;
}

int f ( const int x) {
    return x+10;
}

int main() {
getchar();
return 0;
}
```

Only the const and volatile type-specifiers at the outermost level of the parameter type specification are ignored in this fashion; const and volatile type-specifiers buried within a parameter type specification are significant and can be used to distinguish overloaded function declarations. In particular, for any type T, “pointer to T,” “pointer to const T,” and “pointer to volatile T” are considered distinct parameter types, as are “reference to T,” “reference to const T,” and “reference to volatile T.” For example, see the example in **this comment** posted by Venki. 6) Two parameter declarations that differ only in their default arguments are equivalent. For example, following program fails in compilation with error *“redefinition of `int f(int, int)’ “*

Function overloading and const keyword

The two methods ‘void fun() const’ and ‘void fun()’ have the same signature except that one is const and the other is not. Also, if we take a closer look at the output, we observe that ‘const void fun()’ is called on the const object, and ‘void fun()’ is called on the non-const object. C++ allows member methods to be overloaded on the basis of const type. Overloading on the basis of const type can be useful when a function returns a reference or pointer. We can make one function const, that returns a const reference or const pointer, and another non-const function, that returns a non-const reference or pointer. See **this** for more details. **What about parameters?** Rules related to const parameters are interesting. Let us first take a look at the following two examples. Program 1 fails in compilation, but program 2 compiles and runs fine.

```
// PROGRAM 1 (Fails in compilation)
#include<iostream>
using namespace std;

void fun(const int i)
{
    cout << "fun(const int) called ";
}
void fun(int i)
{
    cout << "fun(int ) called " ;
```

```
}
int main()
{
    const int i = 10;
    fun(i);
    return 0;
}
```

Output: Error

now code fine :

```
// PROGRAM 2 (Compiles and runs fine)
#include<iostream>
using namespace std;

void fun(char *a)
{
cout << "non-const fun() " << a;
}

void fun(const char *a)
{
cout << "const fun() " << a;
}

int main()
{
const char *ptr = "GeeksforGeeks";
fun(ptr);
return 0;
}
```

The two methods ‘void fun() const’ and ‘void fun()’ have the same signature except that one is const and the other is not. Also, if we take a closer look at the output, we observe that ‘const void fun()’ is called on the const object, and ‘void fun()’ is called on the non-const object. C++ allows member methods to be overloaded on the basis of const type. Overloading on the basis of const type can be useful when a function

returns a reference or pointer. We can make one function const, that returns a const reference or const pointer, and another non-const function, that returns a non-const reference or pointer. See **this** for more details. **What about parameters?** Rules related to const parameters are interesting. Let us first take a look at the following two examples. Program 1 fails in compilation, but program 2 compiles and runs fine.

```
// PROGRAM 1 (Fails in compilation)
#include<iostream>
using namespace std;

void fun(const int i)
{
    cout << "fun(const int) called ";
}
void fun(int i)
{
    cout << "fun(int ) called " ;
}
int main()
{
    const int i = 10;
    fun(i);
    return 0;
}
```

Function Overloading and Return Type

functions can not be overloaded if they differ only in the return type.

Example: if there are two functions: **int sum()** and **float sum()**, these two will generate a ***compile-time error*** as function overloading is not possible here.

```
// CPP Program to demonstrate that function overloading
// fails if only return types are different
#include <iostream>
int fun() { return 10; }
```

```
char fun() { return 'a'; }
// compiler error as it is a new declaration of fun()

// Driver Code
int main()
{
    char x = fun();
    getchar();
    return 0;
}
```

Function Overloading and float

Do not support float:

```
include<iostream>
using namespace std;
void test(float s,float t)
{
    cout << "Function with float called ";
}
void test(int s, int t)
{
    cout << "Function with int called ";
}
int main()
{
    test(3.5, 5.6);
    return 0;
}
//code error
```

Code Error,this problem is **Ambiguity**

• The reason behind the ambiguity in above code is that the floating literals

3.5 and **5.6** are actually treated as double by the compiler. ***As per C++ standard, floating point literals (compile time constants) are treated as double unless explicitly specified by a suffix [See 2.14.4 of C++ standard here)***. Since compiler could not find a function with double argument and got confused if the value should be converted from double to int or float.

```
#include<iostream>
using namespace std;
void test(float s,float t)
{
    cout << "Function with float called ";
}
void test(int s,int t)
{
    cout << "Function with int called ";
}
int main()
{
    test(3.5f, 5.6f); // Added suffix "f" to both values
                      // tell compiler, it's a float valu
    return 0;
}
```

Can main() be overloaded

Yea Porssible ,but use bellow rule otherwise error

```
#include <iostream>
int main()
{
    int main = 10;
    std::cout << main;
    return 0;
}
```

now use class :

```
#include <iostream>
using namespace std;
class Test
{
public:
    int main(int s)
    {
        cout << s << "\n";
        return 0;
    }
    int main(char *s)
    {
        cout << s << endl;
        return 0;
    }
    int main(int s ,int m)
    {
        cout << s << " " << m;
        return 0;
    }
};
int main()
{
    Test obj;
    obj.main(3);
    obj.main("I love C++");
    obj.main(9, 6);
    return 0;
}
```

Advantages and Disadvantages of Function Overloading

Advantages of function overloading are as follows:

- The main advantage of function overloading is that it improves code readability and allows code reusability.

- The use of function overloading is to save memory space, consistency, and readability.
- It speeds up the execution of the program
- Code maintenance also becomes easy.
- Function overloading brings flexibility to code.
- The function can perform different operations and hence it eliminates the use of different function names for the same kind of operations.

Disadvantage of function overloading are as follows:

- Function declarations that differ only in the return type cannot be overloaded

Illustration:

```
int fun();
float fun();
```

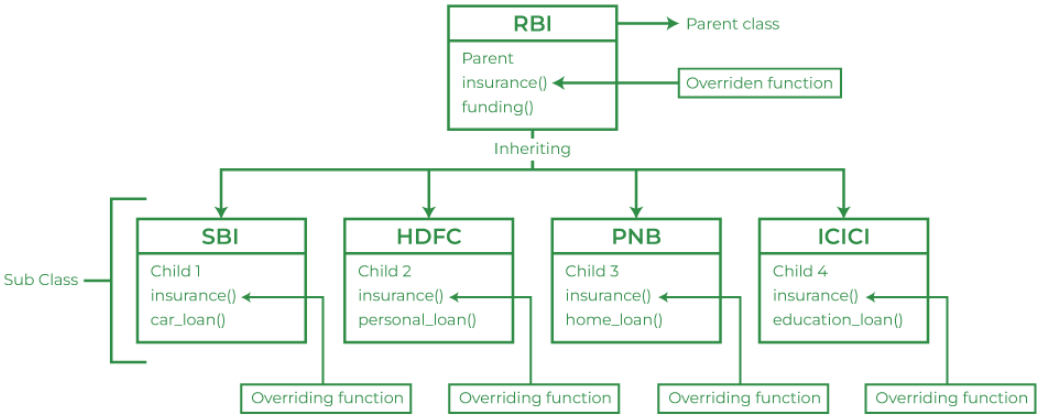
It gives an error because the function cannot be overloaded by return type only.

- Member function declarations with the same name and the same parameter types cannot be overloaded if any of them is a static member function declaration.
- The main disadvantage is that it requires the compiler to perform name mangling on the function name to include information about the argument types.

▼ Function Overriding

Base class and Drive name is same. like

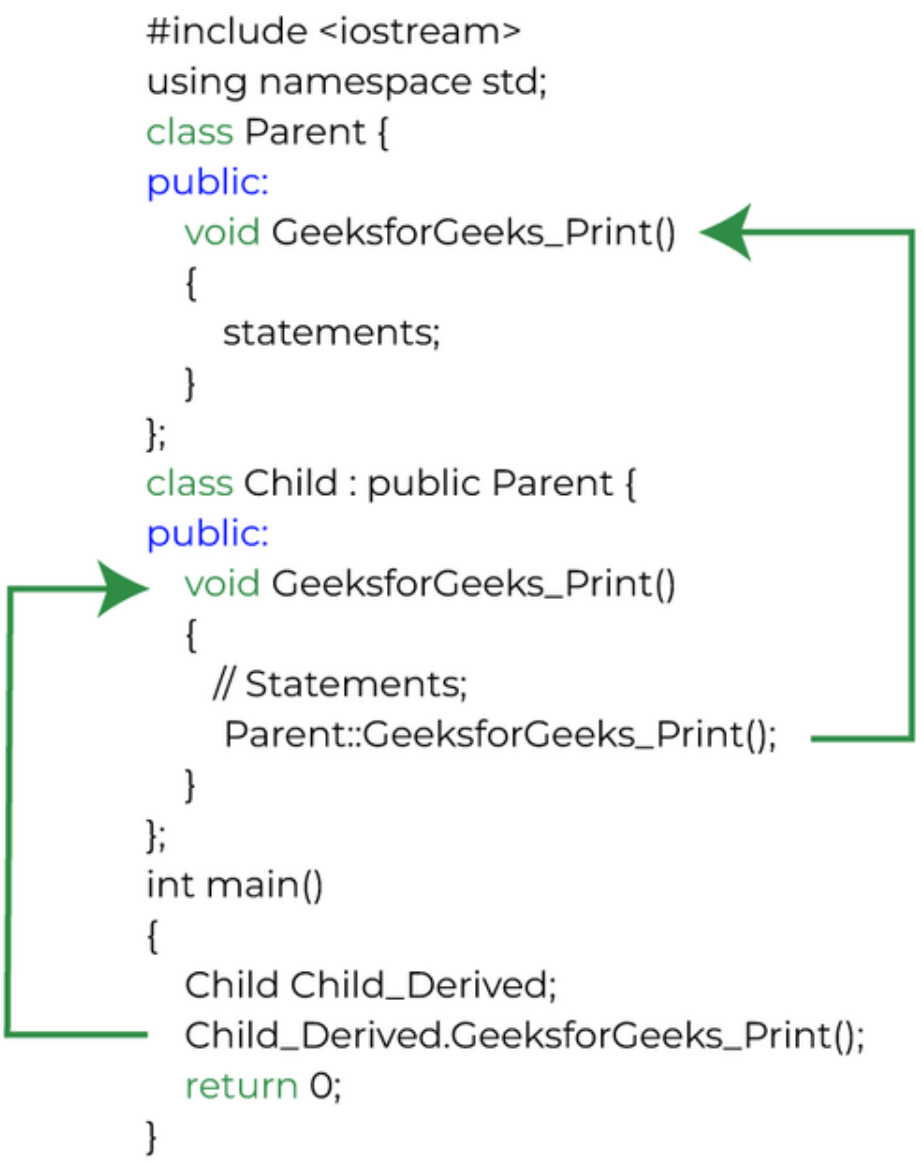
real-life example could be the relationship between RBI(The Reserve Bank of India) and Other state banks like SBI, PNB, ICICI, etc. Where the RBI passes the same regulatory function and others follow it as it is.



Variation Function Overidden:

- Call Overridden Function From Derived Class

```
#include<bits/stdc++.h>
using namespace std;


class base{
   public:
   void  display(){
      cout<<"this is base class"<<"\n";
   }
};

class drive:public base{
   public:
   void display(){
      cout<<"this is drive class"<<"\n";
      base::display(); //call base function
   }
};

int main(){
   drive obj;
   obj.display();
}
```



- Call Overridden Function Using Pointer

  previous study done

- Access of Overridden Function to the Base Class

  `C++ program to access overridden function`

  `main() using the scope resolution operator::`

```
#include<bits/stdc++.h>
using namespace std;


class base{
```

```
   public:
   void  display(){
      cout<<"this is base class"<<"\n";
   }
};

class drive:public base{
   public:
   void display(){
      cout<<"this is drive class"<<"\n";
   }
};

int main(){
   drive obj;
   //Access display function of the Base class
   obj.base::display();
}
```

- ccess to Overridden Function

```
#include<bits/stdc++.h>
using namespace std;


class base{
   public:
   void  display(){
      cout<<"this is base class"<<"\n";
   }
};

class drive:public base{
   public:
   void display(){
      cout<<"this is drive class"<<"\n";
   }
};
```

```
int main(){
   drive obj1,obj2;

   //call drive class into display overdding functior
   obj1.display();
   //call base class overridding function
   obj2.base::display();
}
```

Function Overloading Vs Function Overriding :

Function Overloading	Function Overriding
It falls under Compile-Time polymorphism	It falls under Runtime Polymorphism
A function can be overloaded multiple times as it is resolved at Compile time	A function cannot be overridden multiple times as it is resolved at Run time
Can be executed without inheritance	Cannot be executed without inheritance
They are in the same scope	They are of different scopes.

▼ Operator overloading

দুটি অপরেন্টের উপর ওপারেটর অপারেশন চালানো। আর two অবজেক্ট এর উপর অপারেটর অপারেশন চালানো, দুটি ব্যাপক পার্থক্য। Create data type like builtin datatype useing class. operator overloading means operator কে redefine করা. Redefining the meaning of operators really does not change their original meaning.

operator overloading main concept or main purpose is operation on object.

- normal operator (+,-,*,/) is work bulid in data type like int,float,double and etc. This is because the addition operator “+” is predefined to add variables of built-in data type only.

```
int a,b,c;
c=a+b;
```

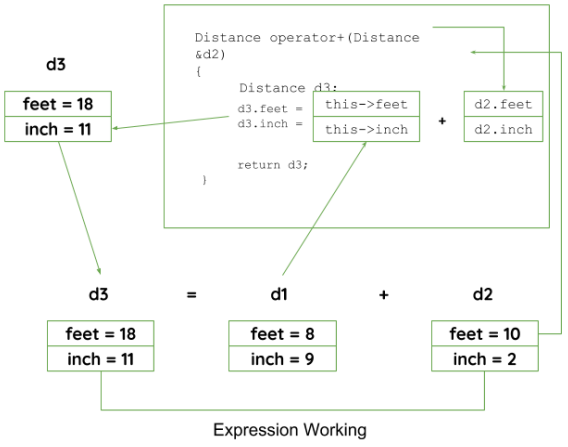
```
//behid the do the work
c=a+(b);
```

- but overloaded operator work in user defind data type like class type data;

```
class A{}; //class is userdefind data type
int main(){
    //create object
    A a,b,c;

    c=a+b; // that is not possible becuse class A is
                //and that is not possible becuse thi
                //because the addition operator “+” i
                //how to adding object a and b,compil
                //this problem resolve comes in opera
            }
```

More example :

d3 = d1 + d2: Here, d1 calls the operator function of its class object and takes d2 as a parameter, by which the operator function returns the object and the result will reflect in the d3 object.



operator overloading is two Type :

- binary operator(+,-,*,)

- Unary oprator(++, - -)
- *Special operators ([ ], (), etc)*

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	>, < , = =, <=, >=

Do not overloaded operator :

```
sizeof
typeid
Scope resolution (::)
Class member access operators (.(dot), .* (po00ver to
member operator))
Ternary or conditional (?:)
```

Example:
Binary operator:

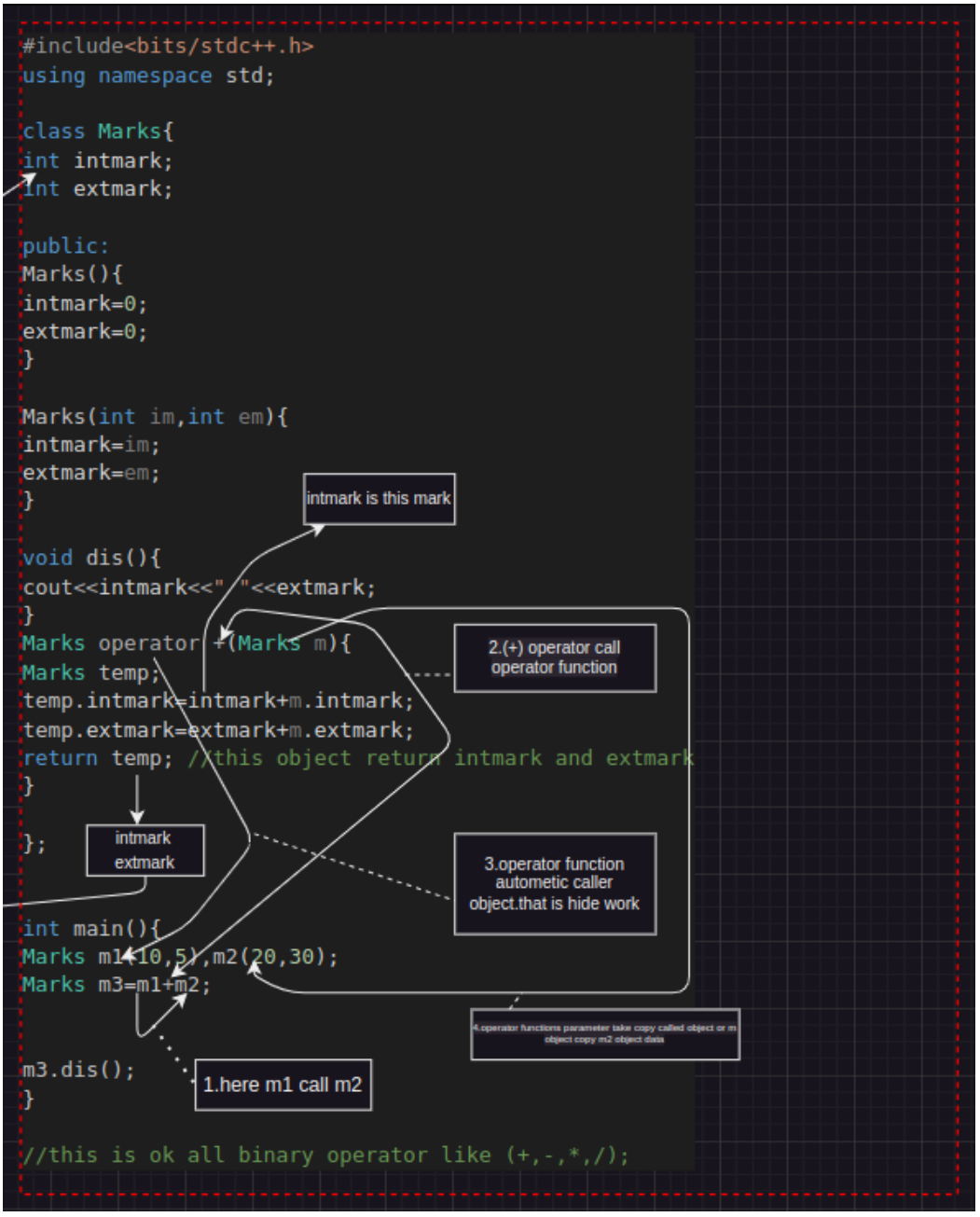
- operator overloading function is inside class Binary operator :

```
#include<bits/stdc++.h>
using namespace std;

class Marks{
    int intmark;
    int extmark;

    public:
```

```
    Marks(){
    }

    Marks(int im,int em){
        intmark=im;
        extmark=em;
    }

    void dis(){
        cout<<intmark<<" "<<extmark;
    }

   Marks operator +(Marks m2){ //here operator function
        Marks m3; //defult constructor object.
       m3.intmark=intmark+m2.intmark;
        m3.extmark=extmark+m2.extmark;
        return m3; //this object return intmark and extm
    }

};


int main(){
    Marks m1(10,5),m2(20,30);
    Marks m3=m1+m2; //here m1 call + operator ,then + op
                                    //m3=m1.operator+(m2

    m3.dis();

}
```

this is ok all binary operator like (+,-,*,/).

▼ image



- outside this class operator function declaration

```
class Marks{
    Marks operator-(Marks m);
};
//class_name Operator_fun_type :: operator+(argument)
Marks Marks :: operator-(Marks m){
        Marks temp;
        temp.intmark=intmark-m.intmark;
```

```
        temp.extmark=extmark-m.extmark;
        return temp;
}
```

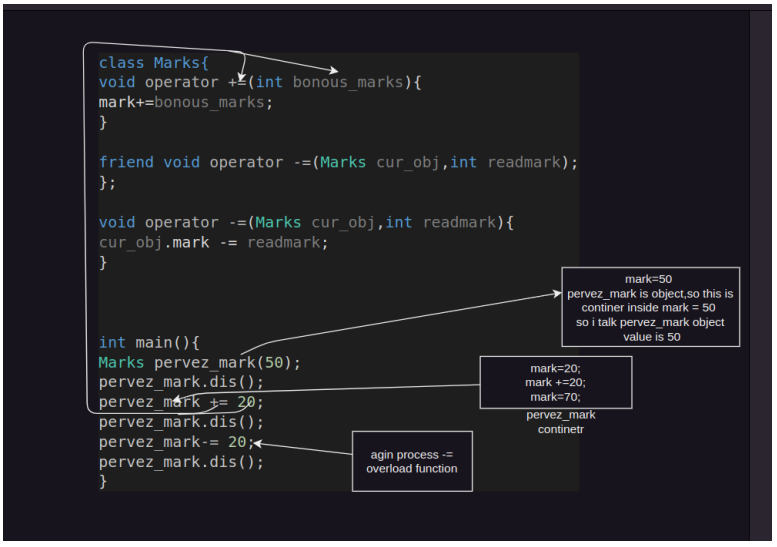
Short hand operator :

all operator is same like *= , /= etc.

```
#include<bits/stdc++.h>
using namespace std;

class Marks{
    int mark; //this is a member
    public:
    Marks(int m=0){
        mark=m;
        }

    void dis(){
        cout<<mark<<" "<<"\n";
        }

      void operator +=(int bonous_marks){
            mark+=bonous_marks;

        }

        friend void operator -=(Marks cur_obj,int readma
};

 void operator -=(Marks cur_obj,int readmark){
    cur_obj.mark -= readmark;
}
int main(){

    Marks pervez_mark(50);
    pervez_mark.dis();
    pervez_mark += 20;
    pervez_mark.dis();
```

```
    pervez_mark-= 20;
    pervez_mark.dis();

    }
```

▼ image



Unary Operator :

Pre Increment and Pre Decrement :

Use case -1 :

```
class Marks{
    int mark;
    public:

    Marks(int x=0){
        mark=x;
    }

    void dis(){
        cout<<mark<<"\n";
    }
```

```
    void operator ++(){
        mark += 1;
    }

    friend void operator--(Marks &m);
};

void operator --(Marks &m){ //when use decrement must
                            //use  reference
    m.mark -= 1;
}

int main(){
    Marks p_mark(50);
    p_mark.dis();

    ++ p_mark;
    p_mark.dis();

    -- p_mark;
    p_mark.dis();
}
```

Use case -2 : useing this pointer

```
previous data is same jast diffrence representation:
class Marks{
Marks operator ++(){
        mark += 1;

        return *this;
    }

    friend Marks operator--(Marks &m);
};

Marks operator --(Marks &m){ //when use decrement must
                           //use  reference
```

```
    m.mark -= 1;
    return m;
}

int main(){
    Marks p_mark(50);
    p_mark.dis();

    ++ p_mark;
    p_mark.dis();

    -- p_mark;
    p_mark.dis();

    //thats line is short
    (-- p_mark).dis();

}
```

Pre and Post fix increment and decrement :

```
#include<bits/stdc++.h>
using namespace std;

class Marks{
    int mark;
    public:

    Marks(int x=0){
        mark=x;
    }

    void dis(){
        cout<<mark<<"\n";
    }


    Marks operator ++(int){
```

```
        Marks duplicate(*this);
        mark += 1;
        return duplicate;
    }

    friend Marks operator --(Marks &m,int);
};

Marks operator --(Marks &m,int){
    Marks duplicate(m);
    m.mark -= 1 ;
    return duplicate;
}


int main(){

   Marks pmark(50);
   pmark.dis();

  (pmark++).dis(); //this object return is duplicate val
  pmark.dis(); //mark = 51
  (pmark++).dis(); //mark is exits 52 after increment.
                   //but display mark=51 [the rule of pc
                   //a++; this line a output is=0
                   //this line a output is = 1 ]

    pmark.dis(); //mark display 52
    (pmark--).dis(); //display 52
    pmark.dis(); //display 51
}
```

Array subscript Operator [] :

```
#include<bits/stdc++.h>
using namespace std;
```

```
class Marks{
    int subject[3];

    public:
    Marks(int sub1,int sub2,int sub3){
        subject[0]=sub1;
        subject[1]=sub2;
        subject[2]=sub3;
    }

    int operator[](int position){
        return subject[position];
    }

};

int main(){
    Marks mobj(80,90,70);

    cout<<mobj[0]<<"\n";
    cout<<mobj[1]<<"\n";
    cout<<mobj[2]<<"\n";
}
```

first braket () :

```
#include<bits/stdc++.h>
using namespace std;

class Marks{
    int mark;
  public:
  Marks(int m){
    cout<<"constructor function is called"<<"\n";
    mark=m;
  }
```

```
    void dis(){
        cout<<"the value is="<<mark<<"\n";
    }

    Marks operator ()(int mk){
        mark=mk;
        cout<<"operator function is called"<<"\n";
        return *this;

    }
};

int main(){
    Marks pmark(50); //this object call contructor funct
    pmark.dis();

    pmark(60); //this object call operator function
                //when use this object auto call operato
    pmark.dis();

}
```

Arrow operator (- >) :

```
#include<bits/stdc++.h>
using namespace std;

class Marks{
    int mark;
  public:

  Marks(int m){
    mark=m;
  }

   void dis(){
    cout<<mark<<"\n";
   }
```

```
    Marks *operator ->(){
        return this;
    }

};

int main(){
    Marks pmark(50);
    pmark.dis();

    pmark->dis(); //pmark object is call operator functo
                //and dot (.) operator replace (->) arro
}
```

New and Delete operator :

```
#include<bits/stdc++.h>
#include<stdexcept>
using namespace std;
class Student{
    string name;
    int age;
public:
    Student(){ //defult constructor
        name="hasan";
        age=23;
    }

    Student(string name,int age){ //parameterise constru
        this->name=name;
        this->age=age;
    }

    void dis(){
        cout<<"my name is "<<name<<"\n";
        cout<<"my age is "<<age<<"\n";
```

```
    }

    void *operator new (size_t size){ //allocet memory
        void *pointer;
        cout<<"Overloaded new.size is="<<size<<"\n";
        pointer= malloc(size);

        if(!pointer){
            bad_alloc ba;
            throw ba;
        }
        return pointer;
    }
    void operator delete(void *pointer){
        cout<<"Overloaded deleted"<<"\n";
        free(pointer);
    }
};

int main(){
    Student *pervezptr;
    try
    {
        pervezptr=new Student("Khan",23);
        pervezptr ->dis();
        delete pervezptr;
    }
    catch(bad_alloc ba)
    {
        cout<<ba.what()<<"\n";
    }

}
```

Insertion and Extersion operator:

use a friend function.

```
#include<bits/stdc++.h>
using namespace std;

class person{
    string name;
    int age;


    person(){
        name="hasan";
        age=20;
    }

    friend ostream &operator << (ostream &output,person
    friend ostream &operator >> (ostream &input,person &
};

ostream &operator <<(ostream &output,person &p){
    output<<"Name is"<<p.name<<"age is"<<p.age;
    return output;

}
  ostream &operator >> (ostream &input,person &p){
    input >> p.name >> p.age;
    return input;
  }

  int main(){
    person obj;

    cout<<"Enter name and age:"<<"\n";
    cin >>obj;
    cout<<obj;


  }
```

```
//do not run code
```

if any more information.

Criteria/Rules :

1. In the case of a **non-static member function**, the binary operator should have only one argument and the unary should not have an argument.
2. In the case of a **friend function**, the binary operator should have only two arguments and the unary should have only one argument.
3. Operators that cannot be overloaded are **.* :: ?:**
4. Operators that cannot be overloaded when declaring that function as friend function are **= () [] >**.
5. The operator function must be either a non-static (member function), global free function or a friend function.

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

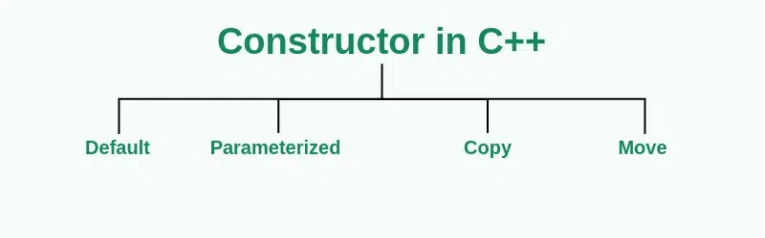
▼ **Constructor / destructor**

following are some main characteristics of the 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.
- Constructors can be overloaded.

- A constructor can not be declared virtual.
- A constructor cannot be inherited.
- The addresses of the Constructor cannot be referred to.
- The constructor makes implicit calls to **new** and **delete** operators during memory allocation.

Constructor Type:



All Tyepe real life Example :

Let us understand the types of constructors in C++ by taking a real-world example. Suppose you went to a shop to buy a marker. When you want to buy a marker, what are the options? The first one you go to a shop and say give me a marker. So just saying give me a marker means that you did not set which brand name and which color, you didn't mention anything just say you want a marker. So when we said just I want a marker whatever the frequently sold marker is there in the market or his shop he will simply hand over that. And this is what a default constructor is!

The second method is you go to a shop and say I want a marker red in color and XYZ brand. So you are mentioning this and he will give you that marker. So in this case you have given the parameters. And this is what a parameterized constructor is!

Then the third one you go to a shop and say I want a marker like this(a physical marker on your hand). So the shopkeeper will see that marker. Okay, and he will give a new marker for you. So copy of that marker. And that's what a copy constructor is!

Now, assume that you don't to buy a new marker but instead take ownership of your friend's marker. It means taking ownership of already

present resources instead of getting a new one. That’s what a move constructor is!

Example:
Defult constructor

```
class A{
int a,c;
public:
    A(){
        int a=20;
        int c=20;
        cout<<a<<b;
    }
};
```

Parametersize constructor:

```
class A{
int a,c;
public:
    A(int x,int y){
        int a=x;
        int c=y;
        cout<<a<<c;
    }
};
```

Defining Parameterized Constructor Outside the Class:

```
class A{
int a;
public:
    A(int a,int b);
};

A::A(int a,int b){
    int x,y;
```

```
    x=a;
    y=b;
    cout<<a<<b;
}
//The parameterized constructors can be called explicitl
```

parameterized constructor with default values

```
class A{
int a;
public:
    A(int x=0){

        int a=x;
        cout<<a;
    }
};

int main(){
A obj1; //output = 0
A obj2(25); //output = 25
}
```

Copy Constructor :

A copy constructor is a member function that initializes an object using another object of the same class.

Copy constructor takes a reference to an object of the same class as an argument.

Uses of Copy Constructor

- Constructs a new object by copying values from an existing object.
- Can be used to perform deep copy.
- Modify specific attributes during the copy process if needed.

In C++, a Copy Constructor may be called for the following cases:

1) When an object of the class is returned by value.

2) When an object of the class is passed (to a function) by value as an argument.

3) When an object is constructed based on another object of the same class.

4) When an object is constructed using initialization lists with braces

5) When the compiler generates a temporary object.

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

  Syntax :

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

Copy constructor work two way:

Implicit Copy Constructor :

```
class A{
    int a;
    public:
    A(int x){
        a=x;
    }

    void dis(){cout<<"ID="<<a<<"\n";}
};
int main(){
    A obj1(20);
    obj1.dis();

    // creating an object of type Sample from the obj
    A obj2(obj1); // or obj2=obj1;
```

```
    obj2.dis();
}
```

Explicit Copy Constructor :

```
class Sample{
    int a;
    public:
    Sample(){} //defult constructor
    Sample(int x){ //parameter consructor
        a=x;
    }
  Sample(Sample &obj){ // copy constructor
      a=obj.a;
    }
    void dis(){cout<<"ID="<<a<<"\n";}
};

int main(){
    Sample obj1(10);
    obj1.dis();

    // creating an object of type Sample from the obj
    Sample obj2(obj1); // or obj2=obj1;
    obj2.dis();
}
```

Initialization with another object of the same type:

```
// CPP Program to demonstrate the use of copy constructo
#include <iostream>
#include <stdio.h>
using namespace std;

class storeVal {
public:
    // Constructor
    storeVal() {}
```

```
    // Copy Constructor
    storeVal(const storeVal& s)
    {
        cout << "Copy constructor has been called " << e
    }
};

// function that returns the object
storeVal foo()
{
    storeVal obj;
    return obj;
}

// function that takes argument of object type
void foo2(storeVal& obj) { return; }

// Driver code
int main()
{
    storeVal obj1;

    cout << "Case 1: ";
    foo();
    cout << endl;

    cout << "Case 2: ";
    foo2(obj1);
    cout << endl;

    cout << "Case 3: ";
    storeVal obj2 = obj1;

    return 0;
}
```

Defining of Explicit Copy Constructor with Parameterized Constructor

```
// C++ program to demonstrate copy construction along wi
// parameterized constructor
#include <iostream>
#include <string.h>
using namespace std;

// class definition
class student {
    int rno;
    char name[50];
    double fee;

public:
    student(int, char[], double);
    student(student& t) // copy constructor
    {
        rno = t.rno;
        strcpy(name, t.name);
        fee = t.fee;
    }

    void display();
};

student::student(int no, char n[], double f)
{
    rno = no;
    strcpy(name, n);
    fee = f;
}

void student::display()
{
    cout << endl << rno << "\t" << name << "\t" << fee;
}

int main()
{
```

```
    student s(1001, "Manjeet", 10000);
    s.display();

    student manjeet(s); // copy constructor called
    manjeet.display();

    return 0;
}
```

Move Constructor:
transfers the ownership of this object to the newly created object.

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

```
??????????????????????
```

Uses of Move Constructor

- Instead of making copies, move constructors efficiently transfer ownership of these resources.
- This prevents unnecessary memory copying and reduces overhead.
- You can define your own move constructor to handle specific resource transfers

RValue and Normal Value Reference :

Rvalue reference use two perpouse:
1. Moving Semantics

2.Perfect Forwarding

here, Compiler auto dected Rvalue and Normal / Lvaue reference value .

```
variable declaration:

Normal value reference:
```

```
int &j=24; //is not allowed,this is lvalue
const int &j=24; //allowed

RValue reference:
 int &&j=24; //allowed that is a Rvalue



 #include<bits/stdc++.h>
using namespace std;

void dis(int & reference){cout<<"Normal value references
void dis(int && reference){cout<<"RValue references";}
int main(){

    int i=10;
    dis(i); //normal, Lvalue reference(int &reference)

    dis(100); //RValue (int &&reference)
    return 0;
}
```

Destructor:

Destructors release memory space occupied by the objects created by the constructor. In **destructor**, objects are destroyed in the reverse of object creation.

```
#include<bits/stdc++.h>
using namespace std;

class my_class{

    public:
    my_class(){  //constructor function
        cout<<"This is constructor function call,";
        cout<<"becuse object is created,";
        cout<<"and this object for allocates memory"<<"\
```

```
    }


    char character(){ // normal function
    char a;
    cout<<"push chacter:"<<"\n";
    cin>>a;
    return a;
   }

   ~my_class(){ //this is distructor function
    cout<<"This is distructor function,and destroy memor
   }
};

int main(){
    my_class obj;  //when object crated auto call constr
    obj.character();
}
```

Characteristics of Destructors in C++

The following are some main characteristics of destructors in C++:

- Destructor is invoked automatically by the compiler when its corresponding constructor goes out of scope and releases the memory space that is no longer required by the program.
- Destructor neither requires any argument nor returns any value therefore it cannot be overloaded.
- Destructor cannot be declared as static and const;
- Destructor should be declared in the public section of the program.
- Destructor is called in the reverse order of its constructor invocation

Private Destructor:

leran link : https://www.geeksforgeeks.org/private-destructor-in-cpp/

Can We Make the Constructors Private?

Yes, in C++, constructors can be made private. This means that no external code can directly create an object of that class.

Note: *We can make the constructor defined outside the class as* ***inline*** *to make it equivalent to the in class definition. But note that* ***inline*** *is not an instruction to the compiler, it is only the request which compiler may or may not implement depending on the circumtances.*

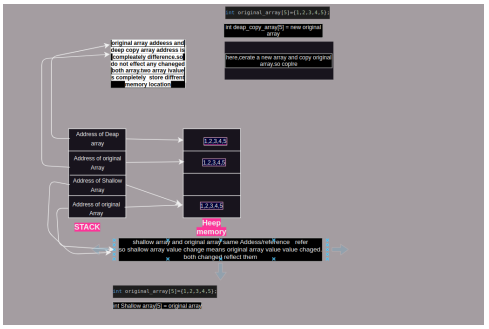
Important Note: *Whenever we define one or more non-default constructors(with parameters) for a class, a default constructor(without parameters) should also be explicitly defined as the compiler will not provide a default constructor in this case. However, it is not necessary but it's considered to be the best practice to always define a default constructor.*

Note: Technically, a constructor or a destructor can perform any type of operation. The code
within these functions does not have to initialize or reset anything related to the class for which
they are defined. For example, a constructor for the preceding examples could have computed pi
to 100 places. However, having a constructor or destructor perform actions not directly related
to the initialization or orderly destruction of an object makes for very poor programming style
and should be avoided.

▼ Shallow copy and Deep copy

when we create variable or work value type, this variable store Stack
when use variable reference,this reference store Stack and value store Heep memory.

concept is :



Static memory Allocation:

```
#include<bits/stdc++.h>
using namespace std;

class Student
{
private:
    char* name;
public:
    //constructor method
    Student(char *n){
        name=n;
    }

    //copy constructor method
    Student(Student &ob){
        name=ob.name;
    }

    //setter method
    void setname(char* n){
        name=n;
    }

    //geter method
    char* getname(){
        return name;
    }
```

```
};

int main(){
    Student s1("Pervez");
    Student s2=s1;

    //name is same s1 and s2
    cout<<"Main constructor:"<<s1.getname()<<"\n";
    cout<<"Copy constructor:"<<s2.getname()<<"\n";

    //then s1 object name changed and print s2 object na
    s1.setname("Hasan");
    s2.setname("Khan");
    cout<<"now changed s1:"<<s1.getname()<<"\n";
    cout<<"now changed s2:"<<s2.getname();

return 0;

}
```

Shallow copy :

this is use same memory location.hense any changed effect both object.

```
#include<bits/stdc++.h>
using namespace std;

class Student
{
private:
    char* name;
public:
    //constructor method
    Student(char *n){
        name=new char[strlen(n)+1]; //1=null pointer,str
        strcpy(name,n);             //and this is a char
        }
```

```
    //copy constructor method
    Student(Student &ob){
        name=ob.name;
    }

    //setter method
    void setname(char* n){
        strcpy(name,n);
    }

    //geter method
    char* getname(){
        return name;
    }


};


int main(){
    Student s1("Pervez");
    Student s2=s1;

    //name is same s1 and s2
    cout<<"Main constructor:"<<s1.getname()<<"\n"<<"Copy
    cout<<endl;

    cout<<"Shallow copy:"<<"\n";
    //then s2 object name changed so s1 objects name is
    s2.setname("Hasan khan");
    cout<<s1.getname()<<"(original name)"<<"\n"<<s2.getr

return 0;

}
```

Some explain:

name=new char[strlen(n)+1]

- `name`: This is likely a variable name. It represents a pointer to a character array (string).
- `new`: In C++, the `new` keyword is used to dynamically allocate memory on the heap. In this case, it's creating a new character array.
- `char`: Indicates that the data type being allocated is a character (i.e., a single character).
- `strlen(n)`: The `strlen` function calculates the length of the string stored in the variable `n`.
- `+1`: Adding 1 to the length accounts for the null terminator character ('\0') required at the end of C-style strings.

Deep Copy :

this is use Diffrent memory location.hense any changed not effect both object.thats object is independent.

```
#include<bits/stdc++.h>
using namespace std;

class Student
{
private:
    char* name;
public:
    //constructor method
    Student(char *n){
        name=new char[strlen(n)+1]; //1=null pointer,str
        strcpy(name,n);             //and this is a char
        }+33

    //copy constructor method
    Student(Student &ob){
        name=new char[strlen(ob.name)+1];
        strcpy(name,ob.name);
    }
```

```
    //setter method
    void setname(char* n){
        strcpy(name,n);
    }

    //geter method
    char* getname(){
        return name;
    }


};


int main(){
    Student s1("Pervez");
    Student s2=s1;

    //name is same s1 and s2
    cout<<"Main constructor:"<<s1.getname()<<"\n"<<"Copy
    cout<<endl;

    cout<<"Deep copy:"<<"\n";
    //s2 object name changed so s1 objects name is no ch
    //this is completely diffrence
    s2.setname("Hasan khan");
    cout<<s1.getname()<<" "<<"(original name)"<<"\n"<<s2

return 0;

}
```

Diffrence between two copy:

	Shallow Copy	**Deep copy**

1.	When we create a copy of object by copying data of all member variables as it is, then it is called shallow copy	When we create an object by copying data of another object along with the values of memory resources that reside outside the object, then it is called a deep copy
2.	A shallow copy of an object copies all of the member field values.	Deep copy is performed by implementing our own copy constructor.
3.	In shallow copy, the two objects are not independent	It copies all fields, and makes copies of dynamically allocated memory pointed to by the fields
4.	It also creates a copy of the dynamically allocated objects	If we do not create the deep copy in a rightful way then the copy will point to the original, with disastrous consequences.

▼ More Important information Constructor Part

▼ When should we write our own copy constructor?

A copy constructor is a member function that initializes an object using another object of the same class. C++ compiler provides a default copy constructor, but it copies members of the class one by one in the target object. This can lead to memory leaks and dangling pointers. It is necessary to define a copy constructor only if an object has pointers or any runtime allocation of the resource. The default constructor does only shallow copy, and deep copy is possible only with a user-defined copy constructor.

The default ***constructor does only shallow copy.***

Deep copy is possible only with a user-defined copy constructor. In a user-defined copy constructor, we make sure that pointers (or references) of copied objects point to new memory locations.

▼ Does C++ compiler create default constructor when we write our own?

No, the C++ compiler doesn't create a default constructor when we initialize our own, the compiler by default creates a default constructor for every class; But, if we define our own constructor, the compiler doesn't create the default constructor. This is so because the default constructor does not take any argument and if two default constructors are created, it is difficult for the compiler which default constructor should be called.

▼ Varatise

1. **Copy constructor vs Assignment Operator**

Copy constructor	**Assignment operator**
It is called when a new object is created from an existing object, as a copy of the existing object	This operator is called when an already initialized object is assigned a new value from another existing object.
It creates a separate memory block for the new object.	It does not create a separate memory block or new memory space.
It is an overloaded constructor.	It is a bitwise operator.
C++ compiler implicitly provides a copy constructor, if no copy constructor is defined in the class.	A bitwise copy gets created, if the Assignment operator is not overloaded.

Syntax:
className(const className &obj) {
// body
}
	Syntax:
	className obj1, obj2;
	obj2 = obj1;

Copy Constructor and assinment operator example:

```
#include <iostream>
#include <stdio.h>
using namespace std;

class Test {
public:
    Test() {}
    Test(const Test& t)
    {
        cout << "Copy constructor called " << endl;
```

```
    }

    Test& operator=(const Test& t) //operator overloa
    {
        cout << "Assignment operator called " << endl
        return *this;
    }
};

// Driver code
int main()
{
    Test t1, t2;
     t2 = t1; //call assignment operator
     Test t3 = t1; //call copy constrctor
     Test p1,p2;
     p2=p1;

    return 0;
}
```

Explanation: Here, **t2 = t1;** calls the **assignment operator**, same as **t2.operator=(t1);** and **Test t3 = t1;** calls the **copy constructor**, same as **Test t3(t1);**

2 .

▼ Encapsulation and Abstraction

Encapsulation :

Encapsulation in C++ is defined as the wrapping up of data and information in a single unit. In Object Oriented Programming,

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”.

Two Important property of Encapsulation

1. **Data Protection:** Encapsulation protects the internal state of an object by keeping its data members private. Access to and modification of these data members is restricted to the class’s public methods, ensuring controlled and secure data manipulation.
2. **Information Hiding:** Encapsulation hides the internal implementation details of a class from external code. Only the public interface of the class is accessible, providing abstraction and simplifying the usage of the class while allowing the internal implementation to be modified without impacting external code.

Features of Encapsulation :

Below are the features of encapsulation:

1. We can not access any function from the class directly. We need an object to access that function that is using the member variables of that class.
2. The function which we are making inside the class must use only member variables, only then it is called *encapsulation*.
3. If we don’t make a function inside the class which is using the member variable of the class then we don’t call it encapsulation.
4. Encapsulation improves readability, maintainability, and security by grouping data and methods together.

5. It helps to control the modification of our data members.

Example-1:

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

class Encapsulation {
private:
    // Data hidden from outside world
    int x;

public:
    // Function to set value of
    // variable x
    void set(int a) { x = a; }

    // Function to return value of
    // variable x
    int get() { return x; }
};

// Driver code
int main()
{
    Encapsulation obj;
    obj.set(5);
    cout << obj.get();
    return 0;
}
```

Explanation: In the above program, the variable **x** is made private. This variable can be accessed and manipulated only using the functions get() and set() which are present inside the class. Thus we can say that here, the variable x and the functions get() and set() are bound together which is nothing but encapsulation.

Points to Consider

As we have seen in the above example, access specifiers play an important role in implementing encapsulation in C++. The process of implementing encapsulation can be sub-divided into two steps:

1. Creating a class to encapsulate all the data and methods into a single unit.
2. Hiding relevant data using access specifiers.

More away Learn :

Class is Encaptulation example.becuse, say class is pakage,inside class store variable and function.

```
Example -1.
//encaptulation achieved use use set and get method
for private variable.
```

Abstruction:

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.

or Abstruction is the process hideing implementation details,and show functionlity details in user.

Consider a ***real-life example of a man driving a car***. The man only knows that pressing the accelerator will increase the speed of the car or applying brakes will stop the car but he does not know how on pressing the accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of the accelerator, brakes, etc in the car. This is what abstraction is.

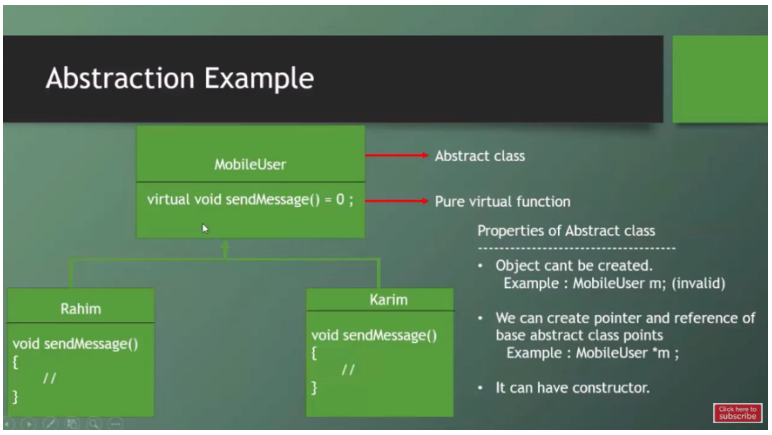
More Example :

when i sent message,i see jast send message but this process hide for me,that is abstruction

আমরা শুধু ততটুকুই দেখাবো যতটা ব্যবহারকারীকে দেখাতে হবে.

if declear pure virtual function,then we can talk abstruct class.otherwise talk to no abstruct class.

so must use pure virtual function.

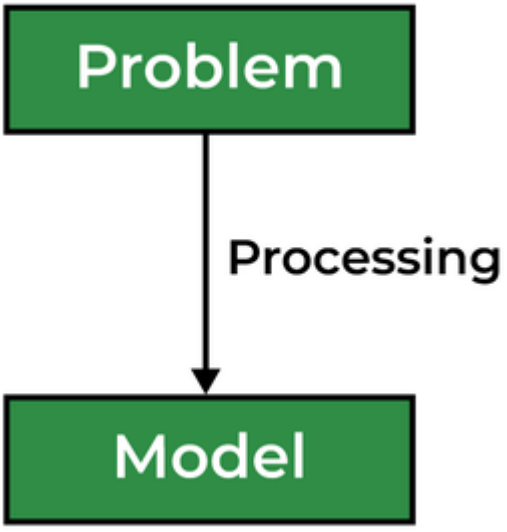


Types of Abstraction:

1. **Data abstraction –** This type only shows the required information about the data and hides the unnecessary data.
2. **Control Abstraction –** This type only shows the required information about the implementation and hides unnecessary information.

```
#include<bits/stdc++.h>
using namespace std;


class mob_user{ //that is abstruct class
   public:
   void call(){
      cout<<"hello"<<"\n";
   }
   virtual void send_message()=0;
};
class Rahim:public mob_user{
   public:
   void send_message(){
      cout<<"hi,i am Rahim"<<"\n";
   }
};
```

```cpp
class Karim:public mob_user{
   public:
   void send_message(){
      cout<<"hi,i am Karim"<<"\n";
   }
};

int main(){
   //mob_user m; //error
   mob_user *m; //fine

   Rahim r;
   Karim k;

   //k.send_message(); //thats way is work no problem
   m =&r;
   m ->send_message(); //rahim send message informati

   m =&k;
   m ->send_message(); //karim send message informatc
   m->call();
}
```



Abstraction using Classes :

We can implement Abstraction in C++ using classes. The class helps us to group data members and member functions using available access specifiers. A Class can decide which data member will be visible to the outside world and which is not.

Abstraction in Header files :

One more type of abstraction in C++ can be header files. For example, consider the pow() method present in math.h header file. Whenever we need to calculate the power of a number, we simply call the function pow() present in the math.h header file and pass the numbers as arguments without knowing the underlying algorithm according to which the function is actually calculating the power of numbers.

Advantages of Data Abstraction

- Helps the user to avoid writing the low-level code
- Avoids code duplication and increases reusability.
- Can change the internal implementation of the class independently without affecting the user.
- Helps to increase the security of an application or program as only important details are provided to the user.
- It reduces the complexity as well as the redundancy of the code, therefore increasing the readability.

Abstraction using Access Specifiers

Access specifiers are the main pillar of implementing abstraction in C++. We can use access specifiers to enforce restrictions on class members. For example:

- Members declared as **public** in a class can be accessed from anywhere in the program.
- Members declared as **private** in a class, can be accessed only from within the class. They are not allowed to be accessed from any part of the code outside the class.

We can easily implement abstraction using the above two features provided by access specifiers. Say, the members that define the

internal implementation can be marked as private in a class. And the important information needed to be given to the outside world can be marked as public. And these public members can access the private members as they are inside the class.

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

class implementAbstraction {
private:
    int a, b;

public:
    // method to set values of
    // private members
    void set(int x, int y)
    {
        a = x;
        b = y;
    }

    void display()
    {
        cout << "a = " << a << endl;
        cout << "b = " << b << endl;
    }
};

int main()
{
    implementAbstraction obj;
    obj.set(10, 20);
    obj.display();
    return 0;
```

You can see in the above program we are not allowed to access the variables a and b directly, however, one can call the function set() to

set the values in a and b and the function display() to display the values of a and b.

Difference between Abstraction and Encapsulation

Abstraction in OOPs :

- Extracts necessary information in a simple way.
- Unnotices less significant components.
- Shows necessary information to users.
- Reduces program complexity by hiding implementation complexities.

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

class Summation {
private:
    // private variables
    int a, b, c;
public:
    void sum(int x, int y)
    {
        a = x;
        b = y;
        c = a + b;
        cout<<"Sum of the two number is : "<<c<<endl;
    }
};
int main()
{
    Summation s;
    s.sum(5, 4);
    return 0;
}
```

Explanation :
In the this example, we can see that abstraction has achieved by using class. The class ‘Summation’ holds the private members a, b and c, which are only accessible by the member functions of that class.

Encaptulation :

Encapsulation Overview

- Process of containing information.
- Hides data in a single entity.
- Protects information from outside world.
- Encapsulates data and function within a class.

Difference between Abstraction and Encapsulation:

S.NO	Abstraction	Encapsulation
1.	Abstraction is the process or method of gaining the information.	While encapsulation is the process or method to contain the information.
2.	In abstraction, problems are solved at the design or interface level.	While in encapsulation, problems are solved at the implementation level.
3.	Abstraction is the method of hiding the unwanted information.	Whereas encapsulation is a method to hide the data in a single entity or unit along with a method to protect information from outside.
4.	We can implement abstraction using abstract class and interfaces.	Whereas encapsulation can be implemented using by access modifier i.e. private, protected and public.
5.	In abstraction, implementation complexities are hidden using abstract classes and interfaces.	While in encapsulation, the data is hidden using methods of getters and setters.

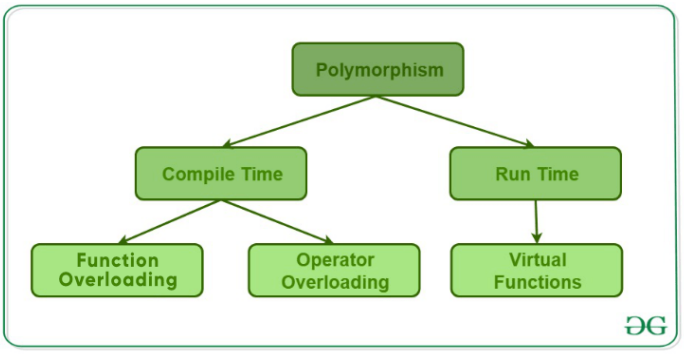
6.	The objects that help to perform abstraction are encapsulated.	Whereas the objects that result in encapsulation need not be abstracted.

▼ Polymorphism

The word “polymorphism” means having many froms. like 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.

Types of Polymorphism

- **Compile-time Polymorphism**
- **Runtime Polymorphism**



- **Compile-Time Polymorphism :**

  When compailation program inside functon, instant compailer detect কোন ফাংশনের কি কাজ, or কোন ফাংশনটি কোন রূপ ধারণ করবে,that is compail time polymorphm.

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

  Compile Time Polymorphism: Whenever an object is bound with its functionality at the compile time, this is known as the compile-time polymorphism. At compile-time, java knows which method to call by checking the method signatures. So this is called compile-time polymorphism or static or early binding. Compile-time polymorphism

is achieved through **method overloading**. Method Overloading says you can have more than one function with the same name in one class having a different prototype. Function overloading is one of the ways to achieve polymorphism but it depends on technology and which type of polymorphism we adopt. In java, we achieve function overloading at compile-Time. The following is an example where compile-time polymorphism can be observed.

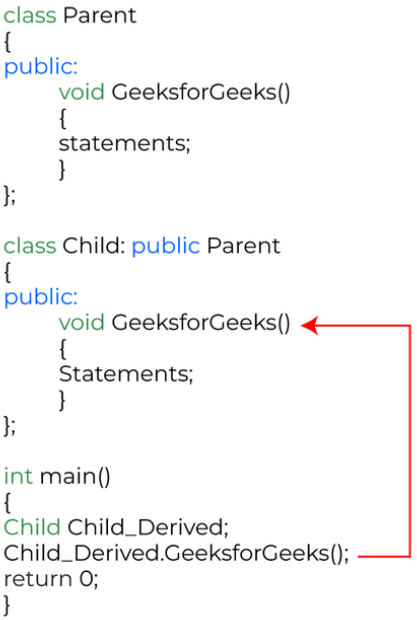
```
#inclue<bits/stdc++.h>
using namespace std;

class A{
    void math(){}
    void math(int a){}
    void math(int a,int b){}
};

int main(){
    A obj;
    obj.math();
    obj.math(10);
    obj.math(20,30);
    //compailer detect witch function witch work
}
```

- **Runtime Polymorphism:**

  when program, then dectect function witch function witch work.

  Run-Time Polymorphism: Whenever an object is bound with the functionality at run time, this is known as runtime polymorphism. The runtime polymorphism can be achieved by **method overriding**. **Java virtual machine** determines the proper method to call at the runtime, not at the compile time. It is also called dynamic or late binding. Method overriding says the child class has the same method as declared in the parent class. It means if the child class provides the specific implementation of the method that has been provided by one of its parent classes then it is known as method overriding. The following is an example where runtime polymorphism can be observed.

Example :



Runtime Polymorphism cannot be achieved by data members in C++.

```
// C++ program for function overriding with data members
#include <bits/stdc++.h>
using namespace std;

// base class declaration.
class Animal {
public:
    string color = "Black";
};

// inheriting Animal class.
class Dog : public Animal {
public:
    string color = "Grey";
};

// Driver code
int main()
{
    Animal d = Dog(); // accessing the field by referenc
```

```
                    // variable which refers to derived
    //cout << d.color;
   d=Animal();
   cout<<d.color;
}
```

When i create derive class object. and call drive class function.and at the same time same name function call base classes.compalir no dectect witch function witch classes,this problem resolved techniqe is use base class virtual functon.

Example Function Overide :

```
#include<bits/stdc++.h>
using namespace std;


class base{
   public:
   void display(){
      cout<<"this is base class"<<"\n";
   }
};

class drive:public base{ //inherite base class
   public:
   void display(){
      cout<<"this is drive class"<<"\n";
   }
};


int main(){
   drive obj;
   obj.display(); //work run time polymorphm
}
```

We can access base class function use virtual function

or i went to access base calss display function,must be use virtual function

```
#include<bits/stdc++.h>
using namespace std;


class base{
   public:
   virtual void  display(){
      cout<<"this is base class"<<"\n";
   }

   void print(){
      cout<<"Print Function base class"<<"\n";
   }
};

class drive:public base{ //inherite base class
   public:
   void display(){
      cout<<"this is drive class"<<"\n";
   }
};


int main(){
   base *ptrobj;
   drive obj;

   ptrobj=&obj;

   /*
   Virtual function, binded at
   runtime (Runtime polymorphism)
   when program compailion skip virtual function.
   and this function detect runtime
   */
```

```
    //This will call the virtual function.
    ptrobj->base::display();
    //non virtual function call in base class
    ptrobj->print();

    //this is drive class
    obj.display();

}
```

The following table demonstrates the difference between runtime polymorphism and compile-time polymorphism:

Compile Time Polymorphism	Run time Polymorphism
In Compile time Polymorphism, the call is resolved by the compiler.	In Run time Polymorphism, the call is not resolved by the compiler.
It is also known as Static binding, Early binding and overloading as well.	It is also known as Dynamic binding, Late binding and overriding as well.
Method overloading is the compile-time polymorphism where more than one methods share the same name with different parameters or signature and different return type.	Method overriding is the runtime polymorphism having the same method with same parameters or signature but associated withcompared, different classes.
It is achieved by function overloading and operator overloading.	It is achieved by virtual functions and pointers.
It provides fast execution because the method that needs to be executed is known early at the compile time.	It provides slow execution as compare to early binding because the method that needs to be executed is known at the runtime.
Compile time polymorphism is less flexible as all things execute at compile time.	Run time polymorphism is more flexible as all things execute at run time.
Inheritance is not involved.	Inheritance is involved.

Difference between Inheritance and Polymorphism:

S.NO	Inheritance	Polymorphism

1.	Inheritance is one in which a new class is created (derived class) that inherits the features from the already existing class(Base class).	Whereas polymorphism is that which can be defined in multiple forms.
2.	It is basically applied to classes.	Whereas it is basically applied to functions or methods.
3.	Inheritance supports the concept of reusability and reduces code length in object-oriented programming.	Polymorphism allows the object to decide which form of the function to implement at compile-time (overloading) as well as run-time (overriding).
4.	Inheritance can be single, hybrid, multiple, hierarchical and multilevel inheritance.	Whereas it can be compiled-time polymorphism (overload) as well as run-time polymorphism (overriding).
5.	It is used in pattern designing.	While it is also used in pattern designing.
6.	**Example :** The class bike can be inherit from the class of two-wheel vehicles, which is turn could be a subclass of vehicles.	**Example :** The class bike can have method name set_color(), which changes the bike's color based on the name of color you have entered.

▼ Virtual Functions and Runtime Polymorphism

Note

> *Note: In C++ what calling a virtual functions means is that; if we call a member function then it could cause a different function to be executed instead depending on what type of object invoked it. Because overriding from derived classes hasn't happened yet, the virtual call mechanism is disallowed in constructors. Also to mention that objects are built from the ground up or follows a bottom to top approach.*

In other words, virtual functions are resolved late, at runtime.

What is the use?

Virtual functions allow us to create a list of base class pointers and call methods of any of the derived classes without even knowing the kind of derived class objec

---

Re learning:

Real-Life Example to Understand the Implementation of Virtual Function

Consider employee management software for an organization.

Let the code has a simple base class *Employee*, the class contains virtual functions like *raiseSalary()*, *transfer()*, *promote()*, etc. Different types of employees like *Managers*, *Engineers*, etc., may have their own implementations of the virtual functions present in base class *Employee*.

In our complete software, we just need to pass a list of employees everywhere and call appropriate functions without even knowing the type of employee. For example, we can easily raise the salary of all employees by iterating through the list of employees. Every type of employee may have its own logic in its class, but we don't need to worry about them because if *raiseSalary()* is present for a specific employee type, only that function would be called.

```
// C++ program to demonstrate how a virtual function
// is used in a real life scenario

class Employee {
public:
    virtual void raiseSalary()
    {
        // common raise salary code
    }

    virtual void promote()
    {
        // common promote code
    }
};
```

```
class Manager : public Employee {
    virtual void raiseSalary()
    {
        // Manager specific raise salary code, may conta
        // increment of manager specific incentives
    }

    virtual void promote()
    {
        // Manager specific promote
    }
};

// Similarly, there may be other types of employees

// We need a very simple function
// to increment the salary of all employees
// Note that emp[] is an array of pointers
// and actual pointed objects can
// be any type of employees.
// This function should ideally
// be in a class like Organization,
// we have made it global to keep things simple
void globalRaiseSalary(Employee* emp[], int n)
{
    for (int i = 0; i < n; i++) {
        // Polymorphic Call: Calls raiseSalary()
        // according to the actual object, not
        // according to the type of pointer
        emp[i]->raiseSalary();
    }
}
```

like the globalRaiseSalary()*' function* there can be many other operations that can be performed on a list of employees without even knowing the type of the object instance.

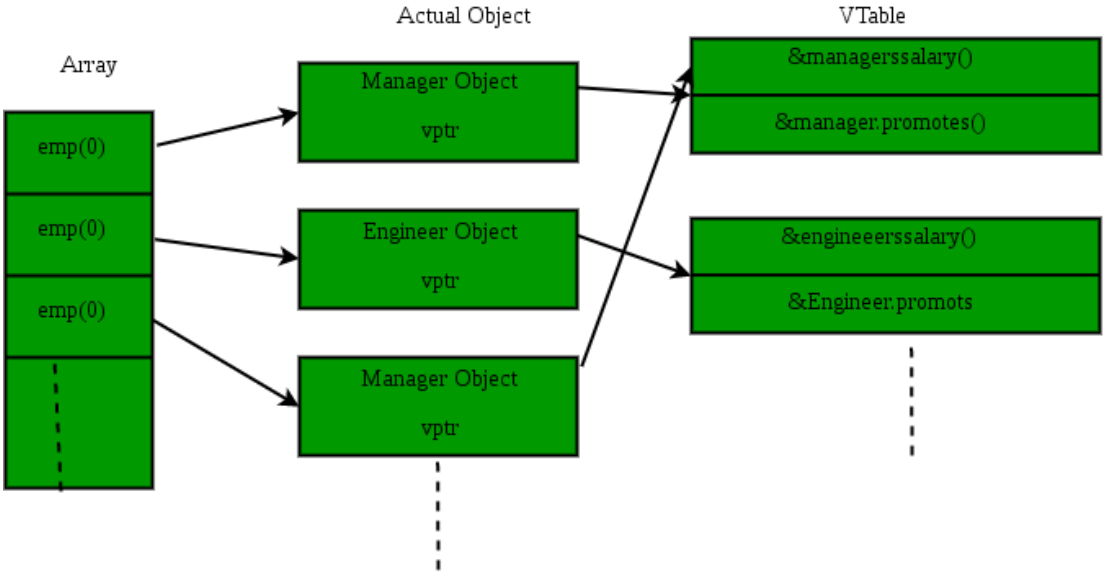
Virtual functions are so useful that later languages like

Java keep all methods virtual by default

How does the compiler perform runtime resolution?

The compiler maintains two things to serve this purpose:

- ***vtable:*** A table of function pointers, maintained per class.
- ***vptr:*** A pointer to vtable, maintained per object instance (see **this** for an example).



The compiler adds additional code at two places to maintain and use *vptr*.

1. Code in every constructor. This code sets the *vptr* of the object being created. This code sets *vptr* to point to the *vtable* of the class.

2. Code with polymorphic function call (e.g. *bp→show()* in above code). Wherever a polymorphic call is made, the compiler inserts code to first look for *vptr* using a base class pointer or reference (In the above example, since the pointed or referred object is of a derived type, *vptr* of a derived class is accessed). Once *vptr* is fetched, *vtable* of derived class can be accessed. Using *vtable*, the address of the derived class function *show()* is accessed and called.

Is this a standard way for implementation of run-time polymorphism in C++?

The C++ standards do not mandate exactly how runtime polymorphism must be implemented, but compilers generally use minor variations on the same basic model.

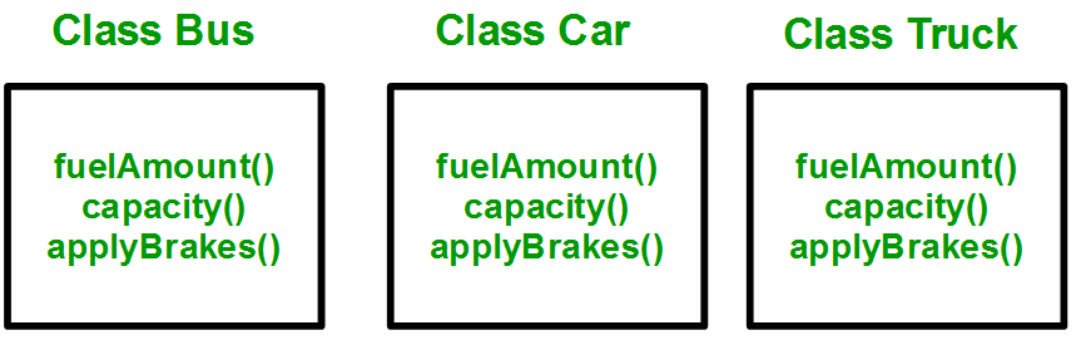
▼ Inheritance

Basic :

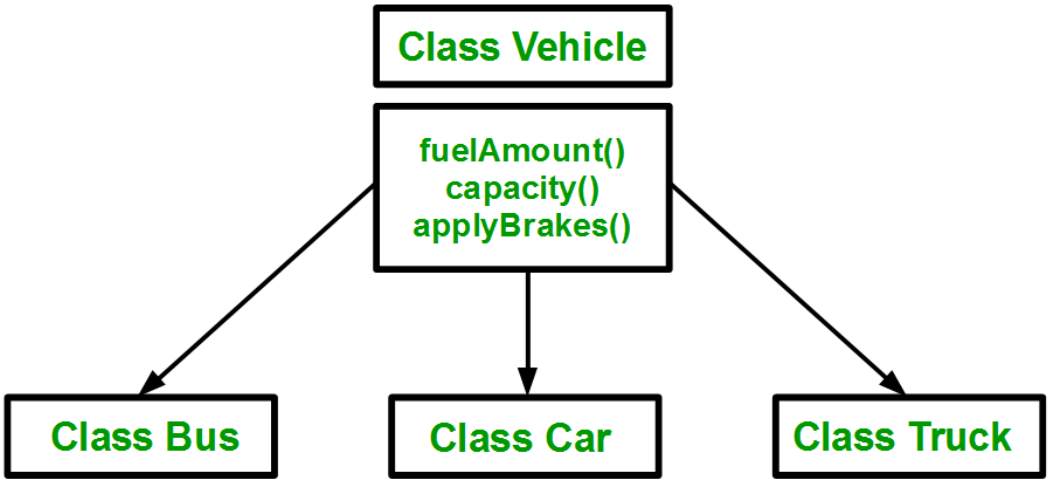
The capability of a **class** to derive properties and characteristics from another class is called **Inheritance**

Why and when to use inheritance?

Consider a group of vehicles. You need to create classes for Bus, Car, and Truck. The methods fuelAmount(), capacity(), applyBrakes() will be the same for all three classes. If we create these classes avoiding inheritance then we have to write all of these functions in each of the three classes as shown below figure:



You can clearly see that the above process results in duplication of the same code 3 times. This increases the chances of error and data redundancy. To avoid this type of situation, inheritance is used. If we create a class Vehicle and write these three functions in it and inherit the rest of the classes from the vehicle class, then we can simply avoid the duplication of data and increase re-usability. Look at the below diagram in which the three classes are inherited from vehicle class:



Using inheritance, we have to write the functions only one time instead of three times as we have inherited the rest of the three classes from the base class (Vehicle).

Note: A derived class doesn't inherit ***access*** to private data members. However, it does inherit a full parent object, which contains any private members which that class declares.

Example:

1. class ABC : private XYZ //private derivation

{ }

2. class ABC : public XYZ //public derivation

{ }

3. class ABC : protected XYZ //protected derivation

{ }

4. class ABC: XYZ //private derivation by default

{ }

Note:

o When a base class is privately inherited by the derived class, public members of the base class becomes the private members of the derived class and therefore, the public members of the base class can only be accessed by the member functions of the derived class. They are inaccessible to the objects of the derived class.

o On the other hand, when the base class is publicly inherited by the derived class, public members of the base class also become the public members of the derived class. Therefore, the public members of the base class are accessible by the objects of the derived class as well as by the member functions of the derived class.

Modes of Inheritance: There are 3 modes of inheritance.

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.
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.
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.

Note: The private members in the base class cannot be directly accessed in the derived class, while protected members can be directly accessed. For example, Classes B, C, and D all contain the variables x, y, and z in the below example. It is just a question of access.

```
/ C++ Implementation to show that a derived class
// doesn’t inherit access to private data members.
// However, it does inherit a full parent object.
class A {
public:
    int x;

protected:
    int y;

private:
    int z;
};
```

```
class B : public A {
    // x is public
    // y is protected
    // z is not accessible from B
};

class C : protected A {
    // x is protected
    // y is protected
    // z is not accessible from C
};

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

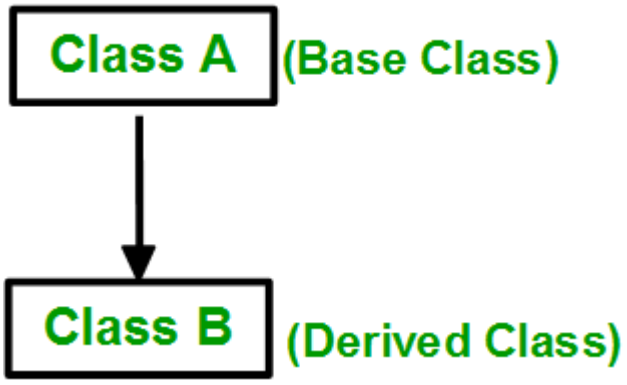
The below table summarizes the above three modes and shows the access specifier of the members of the base class in the subclass when derived in public, protected and private modes:

Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

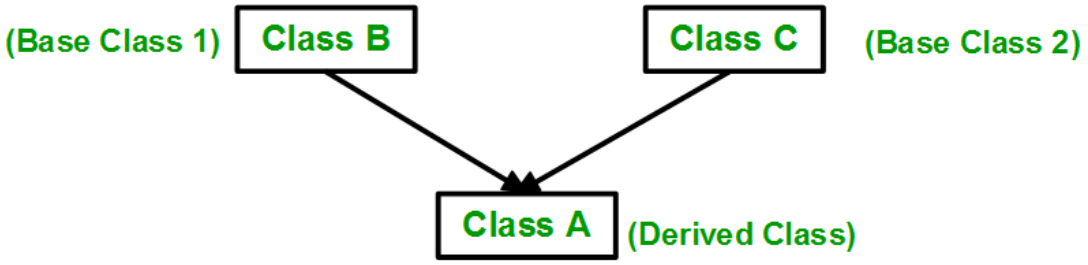
Types Of Inheritance:-

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

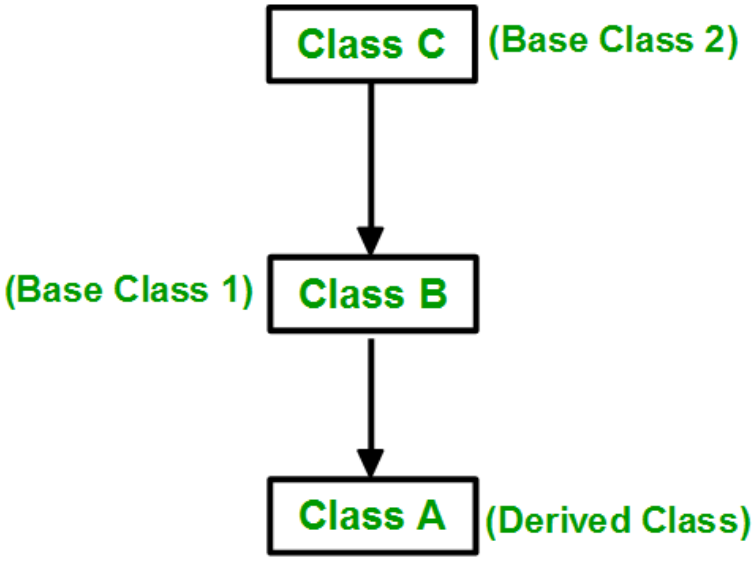
1. Single Inheritance: In single inheritance, a class is allowed to inherit from only one class. i.e. one subclass is inherited by one base class only.



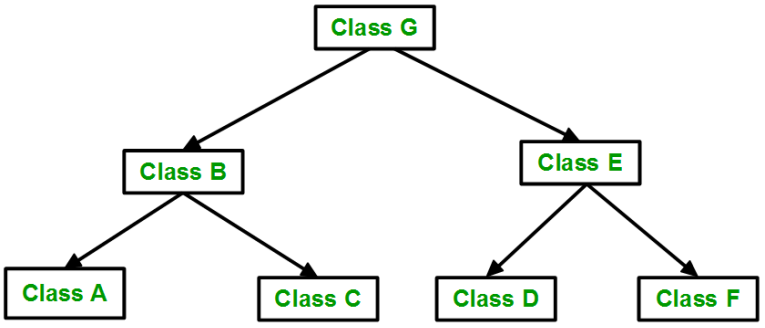
2. Multiple Inheritance: Multiple Inheritance is a feature of C++ where a class can inherit from more than one class. i.e one **subclass** is inherited from more than one **base class**.



3. Multilevel Inheritance: In this type of inheritance, a derived class is created from another derived class.



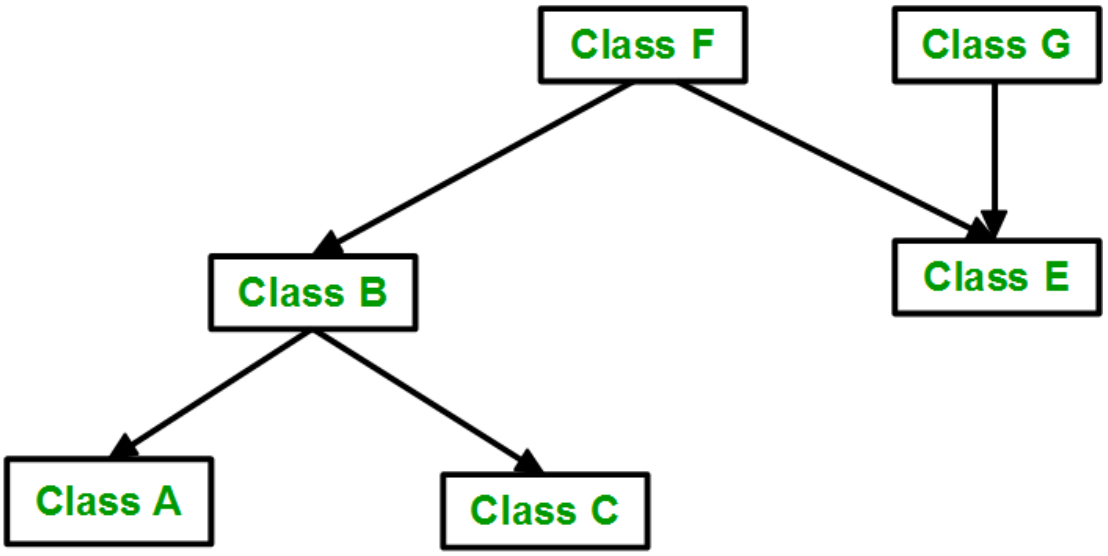
4. Hierarchical Inheritance: In this type of inheritance, more than one subclass is inherited from a single base class. i.e. more than one derived class is created from a single base class.



5. Hybrid (Virtual) Inheritance

: Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance.

Below image shows the combination of hierarchical and multiple inheritances:



6. A special case of hybrid inheritance: Multipath inheritance

A derived class with two base classes and these two base classes have one common base class is called multipath inheritance. Ambiguity can arise in this type of inheritance.

NOW Explanation :

Inheritance Access

public, protected, and private inheritance have the following features:

public inheritance: *makes public members of the base class public in the derived class, and the protected members of the base class remain protected in the derived class.*

drive class only for Access base class public Attribute.if i went use protected attribute must be use function.

```
   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 << object1.getPVT();
    cout << Pr object1.getProt();
    //Direct access public data
    cout << object1.pub << endl;
    return 0;
}
```

protected inheritance: *makes the public and protected members of the base class protected in the derived class.*

drive class do not Access base class public and protected Attribute. if i went use protected attribute and public must be use function.

```
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;
}
```

private inheritance: *makes the public and protected members of the base class private in the derived class.*

drive class do not Access base class public and protected Attribute. if i went use protected attribute and public must be use function.

```
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; }
};

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

private members cannot be accessed from the Derived class. These members cannot be directly accessed from outside the class.

Accessibility Of Inheritance Access:

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

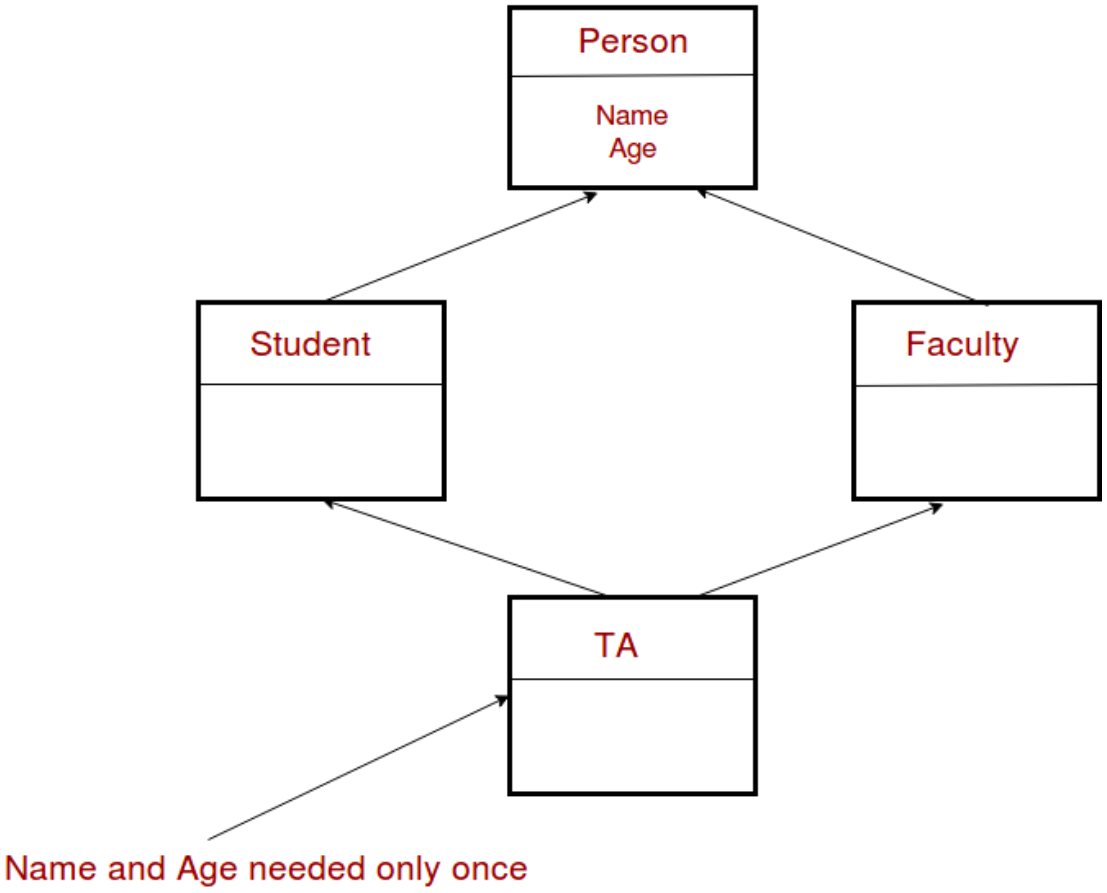
Multiple Inheritance

Java not Allow multiple inheritance :

Syntax:

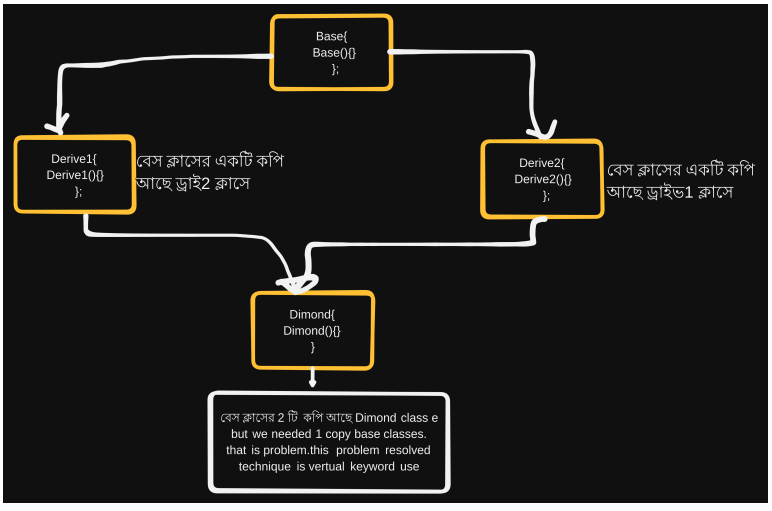
```
class A
{
... .. ...
};
class B
{
... .. ...
};
class C: public A,public B
{
... ... ...
};
```

The diamond problem The diamond problem occurs when two superclasses of a class have a common base class. For example, in the following diagram, the TA class gets two copies of all attributes of Person class, this causes ambiguities.



Multiple inhertance create problem ambiguities.

Now show this problem:




```
class Base{
      //base class attribute
```

```
      public:
      Base(){
         cout<<"Base class Constructor"<<"\n";
         }
   };

   class derive1:public Base{
      //derive class attribute
      public:
      derive1(){
         cout<<"Drive1 class contructor"<<"\n";
         }
   };

   class derive2:public Base{
      //derive class attribute
      public:
      derive2(){
         cout<<"Drive2 class contructor"<<"\n";
         }
   };

   class dimond:public derive1,public derive2{
      //derive class attribute
      public:
      dimond(){
         cout<<"Dimond class contructor"<<"\n";
         }
   };

   int main(){
      dimond obj;
   }
 /*
OutPut:
Base class Constructor
Drive1 class contructor
Base class Constructor
```

```
Drive2 class contructor
Dimond class contructor
 */
```

Resolved Problem:

```
 class Base{
      //base class attribute
      public:
      Base(){
         cout<<"Base class Constructor"<<"\n";
         }
   };

   class derive1:virtual public Base{
      //derive class attribute
      public:
      derive1(){
         cout<<"Drive1 class contructor"<<"\n";
         }
   };

   class derive2: virtual public Base{
      //derive class attribute
      public:
      derive2(){
         cout<<"Drive2 class contructor"<<"\n";
         }
   };

   class dimond:public derive1,public derive2{
      //derive class attribute
      public:
      dimond(){
         cout<<"Dimond class contructor"<<"\n";
         }
```

```
   };

   int main(){
      dimond obj;
   }
 /*
OutPut:
Base class Constructor
Drive1 class contructor
Drive2 class contructor
Dimond class contructor
 */
```

How to call the parameterized constructor of the Base class?

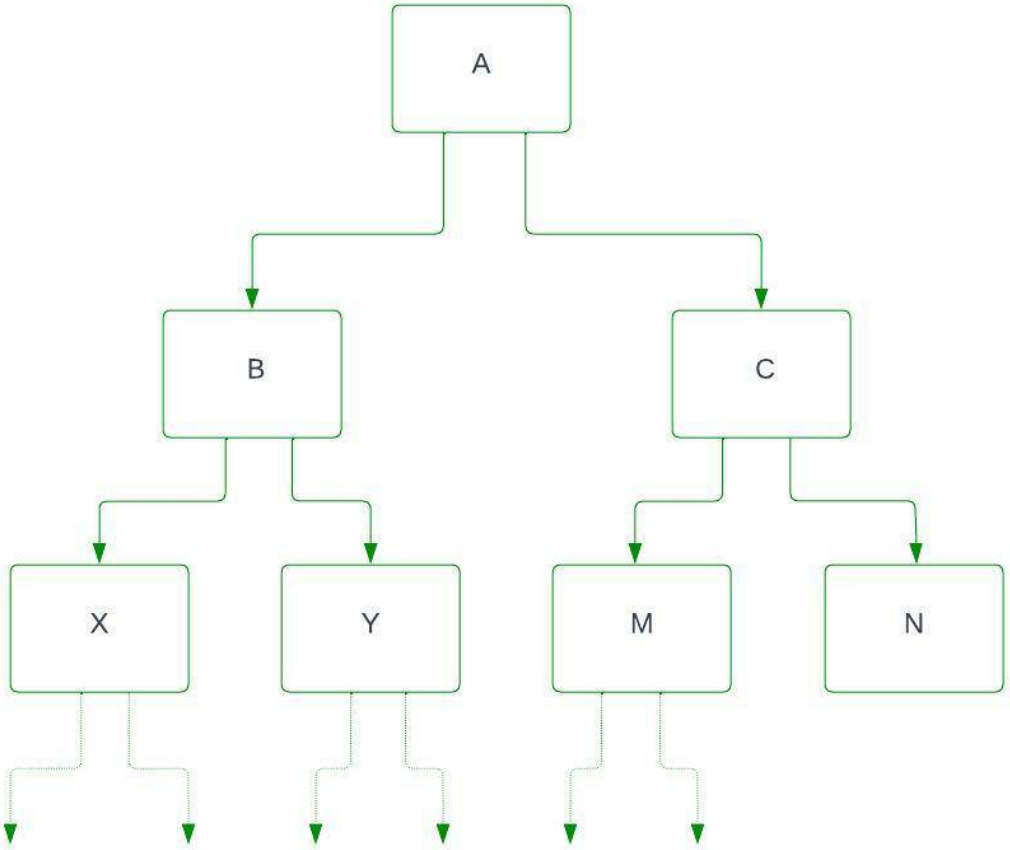
```
#include<iostream>
using namespace std;

class Person {
// Data members of person
public:
    //person class constructor
    Person(int x) { cout << "Person class constructor" <
};


class Faculty :virtual public Person {
// data members of Faculty
public:
    Faculty(int x):Person(x) {
    cout<<"Faculty class constructor"<< endl;
    }
};


class Student : virtual public Person {
// data members of Student
public:
```

```
    Student(int x):Person(x) {
        cout<<"Student class constructor"<< endl;
    }
};

class TA : public Faculty, public Student {
public:
    TA(int x):Student(x), Faculty(x) {
        cout<<"TA class constructor"<< endl;
    }
};

int main() {
    TA ta1(30);
}
```

Hierarchical Inheritance

n Hierarchical inheritance, more than one sub-class inherits the property of a single base class. There is one base class and multiple derived classes. Several other classes inherit the derived classes as well. Hierarchical structures thus form a tree-like structure. It is similar to that, mango and apple both are fruits; both inherit the property of fruit. Fruit will be the Base class, and mango and apple are sub-classes.

The below diagram shows, Class A is a Base class, B is a subclass inherited from class A, and C is a subclass it also inherits from class A.

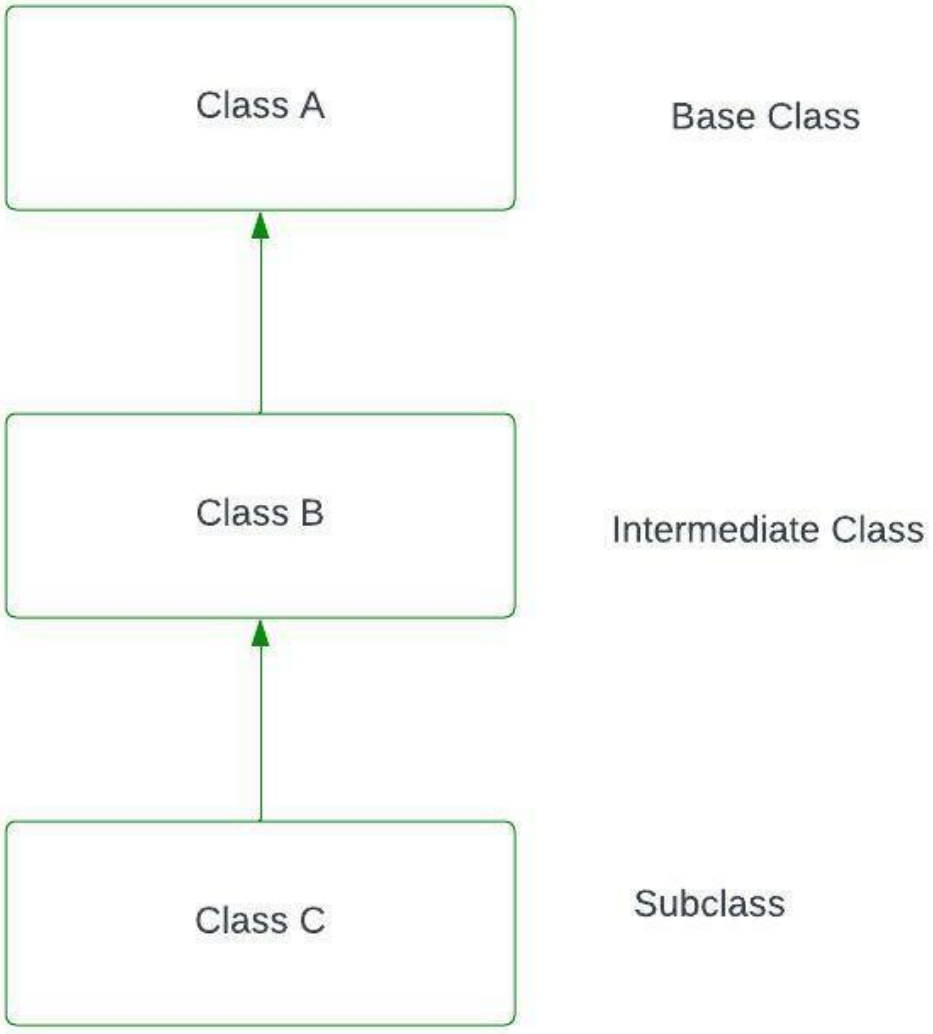


Multilevel Inheritance

Multilevel Inheritance in C++ is the process of deriving a class from another derived class. When one class inherits another class it is further inherited by another class. It is known as multi-level inheritance.

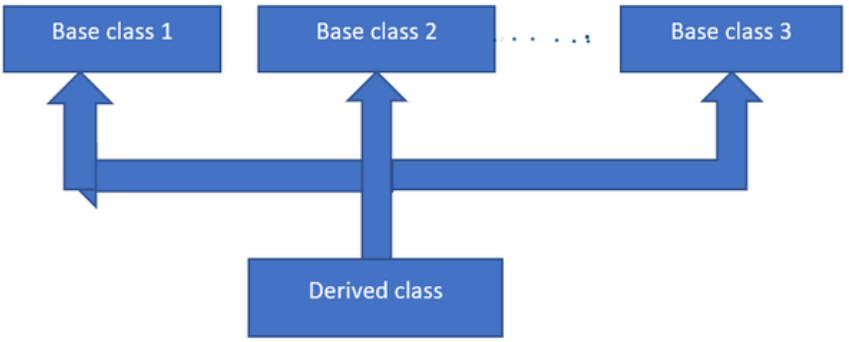
Like :

```
Base class-> Wood, Intermediate class-> furniture, subclass-> table
```

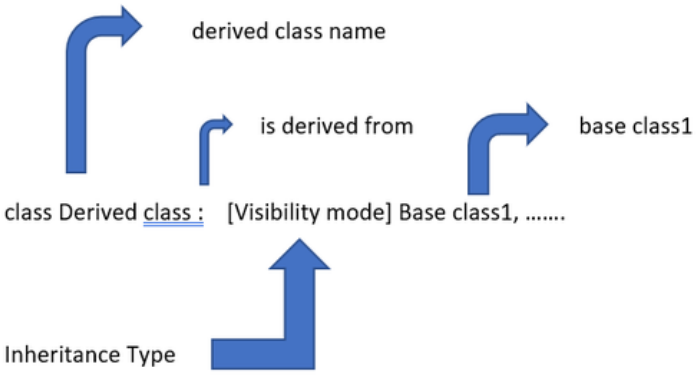


Constructor in Multiple Inheritance

Constructor is a class member function with the same name as the class. The main job of the constructor is to allocate memory for class objects. Constructor is automatically called when the object is created.



Syntax of Multiple Inheritance:



SyntaX :

> *class S: public A1, virtual A2{....};*
> *Here,A2(): virtual base constructor and A1(): base constructor and S(): derived constructor*

Example :

```
#include<iostream>
using namespace std;
class A1
{
```

```
  public:
  A1()
  {
    cout << "Constructor of the base class A1 \n";
  }

};

class A2
{
  public:
  A2()
  {
    cout << "Constructor of the base class A2 \n";
  }

};

class S: public A1, virtual A2
{
  public:
  S(): A1(), A2()
  {
    cout << "Constructor of the derived class S \n";
  }
};

// Driver code
int main()
{
  S obj;
  return 0;
}
```

Inheritance and Friendship

Inheritance in C++: This is an **OOPS** concept. It allows creating classes that are derived from other classes so that they automatically include some of the functionality of its base class and some functionality of its own. (See **this article** for reference)

Friendship in C++: Usually, private and protected members of a class cannot be accessed from outside the same class in which they are declared. However, a friend class has the access to the protected and private members of the first one. Classes that are 'friends' can access not just the public members, but the private and protected members too. (See **this article** for reference)

Difference between Inheritance and Friendship in C++: In C++, friendship is not inherited. If a base class has a friend function, then the function doesn't become a friend of the derived class(es).

For example, the following program prints an error because the ***show()*** which is a friend of base class *A* tries to access private data of derived class *B*.

```
CPP Program to demonstrate the relation between
// Inheritance and Friendship
#include <iostream>
using namespace std;

// Parent Class
class A {
protected:
    int x;

public:
    A() { x = 0; }
    friend void show();
};

// Child Class
class B : public A {
private:
    int y;
```

```
public:
    B() { y = 0; }
};

void show()
{
    B b;
    cout << "The default value of A::x = " << b.x;

    // Can't access private member declared in class 'B'
    cout << "The default value of B::y = " << b.y;
}

int main()
{
    show();
    getchar();
    return 0;
}
```

Output : error

Do not run Program,becuse friends function is show function, Friend function do not inherited without Friend class.

Does overloading work with Inheritance

If we have a function in base class and another function with the same name in derived class, can the base class function be called from derived class object? This is an interesting question and as an experiment, predict the output of the following **C++** program:

```
#include <iostream>
using namespace std;
class Base
{
```

```
public:
    int f(int i)
    {
        cout << "f(int): ";
        return i+3;
    }
};
class Derived : public Base
{
public:
    double f(double d)
    {
        cout << "f(double): ";
        return d+3.3;
    }
};
int main()
{
    Derived* dp = new Derived;
    cout << dp->f(3) << '\n';
    cout << dp->f(3.3) << '\n';
    delete dp;
    return 0;
}
```

The output of this program is:

```
f(double): 6.3
f(double): 6.6
```

Instead of the supposed output:

```
f(int): 6
f(double): 6.6
```

```
#include <iostream>
using namespace std;
class Base
```

```
{
public:
    int f(int i)
    {
        cout << "f(int): ";
        return i+3;
    }
};
class Derived : public Base
{
public:
    double f(double d)
    {
        cout << "f(double): ";
        return d+3.3;
    }
};
int main()
{
    Derived* dp = new Derived;
    cout << dp->f(3) << '\n';
    cout << dp->f(3.3) << '\n';
    delete dp;
    return 0;
}
```

▼ Inheritance Ambiguity

When multiple inheritances two base class function name is same.then drive class inherite 2 base class same function,now crate a problem **Ambiguity**

compailer do not detect witch fucntion witch classes.

```
class Base1{
      public:
      void dis(){
         cout<<"Base1 class Dis";
```

```
      }
   };
   class Base2{
      public:
      void dis(){
         cout<<"Base 2 class Dis";
      }
   };

   class Derive:public Base1,public Base2{
      //now here exits base1 dis and base 2 dis
   };

   int main(){
      Derive D;
      D.dis();
   }
```

Now Resoloved Problem:

Techique 1:

Syntax:

```
ObjectName.ClassName::FunctionName();
```

```
class Base1{
      public:
      void dis(){
         cout<<"Base1 class Dis"<<"\n";
      }
   };
   class Base2{
      public:
      void dis(){
         cout<<"Base 2 class Dis";
      }
   };
```

```
   class Derive:public Base1,public Base2{
      //now here exits base1 dis and base 2 dis
   };

   int main(){
      Derive D;
      D.Base1::dis();
      D.Base2::dis();
   }
```

Techique 2: use virtual Keyword :

Avoiding ambiguity using the virtual base class:

```
#include<iostream>

class ClassA
{
  public:
    int a;
};

class ClassB : virtual public ClassA
{
  public:
    int b;
};

class ClassC : virtual public ClassA
{
  public:
    int c;
};

class ClassD : public ClassB, public ClassC
{
  public:
    int d;
};
```

```
int main()
{
    ClassD obj;

    obj.a = 10;       // Statement 3
    obj.a = 100;      // Statement 4

    obj.b = 20;
    obj.c = 30;
    obj.d = 40;

    cout << "\n a : " << obj.a;
    cout << "\n b : " << obj.b;
    cout << "\n c : " << obj.c;
    cout << "\n d : " << obj.d << '\n';
}
```

According to the above example, Class-D has only one copy of ClassA, therefore, statement 4 will overwrite the value of a, given in statement 3.

▼ Virtual Function basic

My Note:

A base class type reference or pointer can be refer to a drived class object.

```
ase *p=new Derive;
 Derive d;
 Base &r=d;
```

Noramal Function:

```
class Base{
    public:
         void print(){
        cout<<"Base class"<<"\n";
        }
```

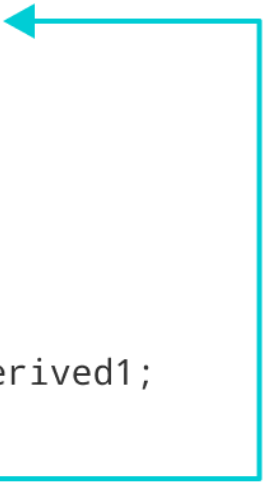
```
};
class Derive:public Base{
    public:
    void print(){
        cout<<"derive derive class"<<"\n";
    }
};

int main(){
//Defenitly print base class
Base b;
b.print();

//definetly print derive class
Derive d;
d.print();


Base *ptr=&d; //the type of pointer Base type.not oop pc
ptr->print(); //ptr point base classes normal function

return 0;
}
```

Virual Function :

```
#include<bits/stdc++.h>
using namespace std;

class Base{
    public:
       virtual void print(){
        cout<<"Base class"<<"\n";
        }

};
```

```
class Derive:public Base{
    public:
    void print(){
        cout<<"derive derive class"<<"\n";
    }
};

int main(){
//Defenitly print base class
Base b;
b.print();

//definetly print derive class
Derive d;
d.print();

Base *ptr=&d; //the type of pointer Base type.
ptr->print(); //point derive class normal function

/*
* ptr point derive classes normal function
  becuse base class Print function is virtual.
* virtual means ভাসা ভাসা functon. তাই derive class এর আ
* Print() ফাংশনটি ওভাররাইড করা হয় এমনকি যখন আমরা বেস টাইপে
  যা Derived অবজেক্ট er d কে নির্দেশ করে।
*/

//Now Access virtual function use derive classes object:
d.Base::print();

return 0;
}
```

```cpp
class Base {
    public:
      virtual void print() {
          // code
      }
};

class Derived : public Base {
    public:
      void print() {
          // code
      }
};

int main() {
    Derived derived1;
    Base* base1 = &derived1;

    base1->print();

    return 0;
}
```



print() of Derived class is called because print() of Base class is virtual

Basic:

A virtual function (also known as virtual methods) is a member function that is declared within a base class and is re-defined (overridden) by a derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the method.

- 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:

1. Virtual functions cannot be static.
2. A virtual function can be a friend function of another class.
3. Virtual functions should be accessed using a pointer or reference of base class type to achieve runtime polymorphism.
4. The prototype of virtual functions should be the same in the base as well as the derived class.
5. 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.
6. A class may have a **virtual destructor** but it cannot have a virtual constructor.

Compile time (early binding) VS runtime (late binding) behavior of Virtual Functions

```
// C++ program to illustrate
// concept of Virtual Functions
```

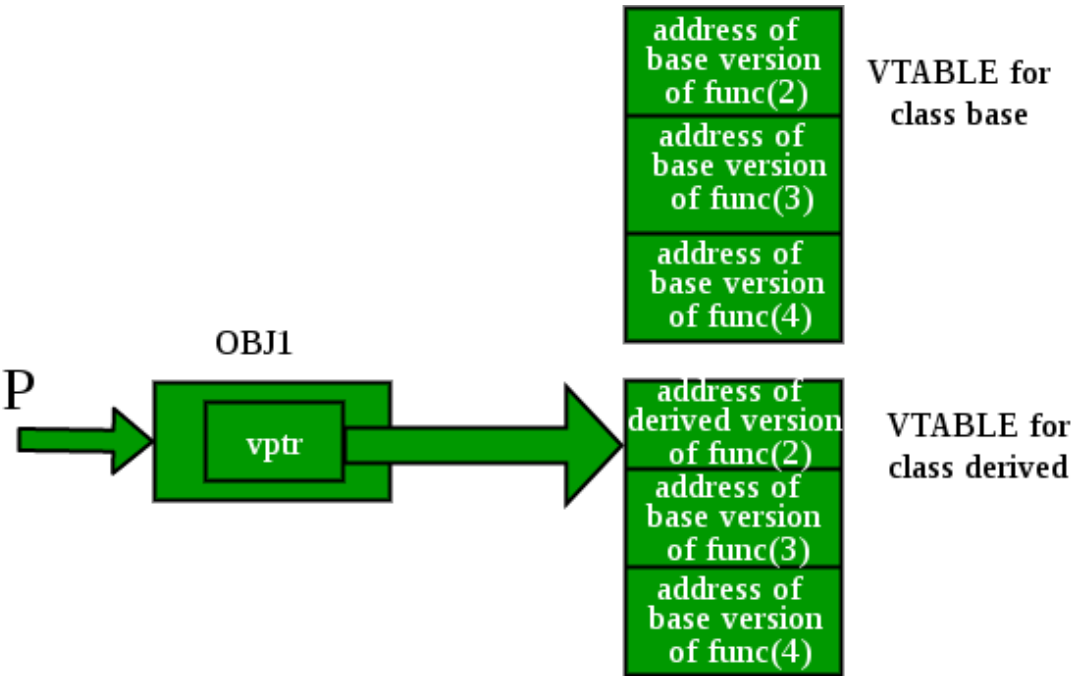
```
#include <iostream>
using namespace std;

class base {
public:
    virtual void print() { cout << "print base class\n";

    void show() { cout << "show base class\n"; }
};

class derived : public base {
public:
    void print() { cout << "print derived class\n"; }

    void show() { cout << "show derived class\n"; }
};

int main()
{
    base* bptr;
    derived d;
    bptr = &d;

    // Virtual function, binded at runtime
    bptr->print();

    // Non-virtual function, binded at compile time
    bptr->show();

    return 0;
}
```

Note: *If we have created a virtual function in the base class and it is being overridden in the derived class then we don't need a virtual keyword in the derived class, functions are automatically considered virtual functions in the derived class.*

Working of Virtual Functions (concept of VTABLE and VPTR)

As discussed **here**, if a class contains a virtual function then the compiler itself does two things.

1. If an object of that class is created then a **virtual pointer (VPTR)** is inserted as a data member of the class to point to the VTABLE of that class. For each new object created, a new virtual pointer is inserted as a data member of that class.
2. Irrespective of whether the object is created or not, the class contains as a member **a static array of function pointers called VTABLE**. Cells of this table store the address of each virtual function contained in that class.




```
// C++ program to illustrate
// working of Virtual Functions
#include <iostream>
using namespace std;

class base {
public:
    void fun_1() { cout << "base-1\n"; }
```

```
    virtual void fun_2() { cout << "base-2\n"; }
    virtual void fun_3() { cout << "base-3\n"; }
    virtual void fun_4() { cout << "base-4\n"; }
};

class derived : public base {
public:
    void fun_1() { cout << "derived-1\n"; }
    void fun_2() { cout << "derived-2\n"; }
    void fun_4(int x) { cout << "derived-4\n"; }
};

int main()
{
    base* p;
    derived obj1;
    p = &obj1;

    // Early binding because fun1() is non-virtual
    // in base
    p->fun_1();

    // Late binding (RTP)
    p->fun_2();

    // Late binding (RTP)
    p->fun_3();

    // Late binding (RTP)
    p->fun_4();

    // Early binding but this function call is
    // illegal (produces error) because pointer
    // is of base type and function is of
    // derived class
    // p->fun_4(5);
```

```
    return 0;
}
```

Output

```
base-1
derived-2
base-3
base-4
```

Explanation: Initially, we create a pointer of the type base class and initialize it with the address of the derived class object. When we create an object of the derived class, the compiler creates a pointer as a data member of the class containing the address of VTABLE of the derived class.

A similar concept of **Late and Early Binding** is used as in the above example. For the fun_1() function call, the base class version of the function is called, fun_2() is overridden in the derived class so the derived class version is called, fun_3() is not overridden in the derived class and is a virtual function so the base class version is called, similarly fun_4() is not overridden so base class version is called.

> Note: fun_4(int) in the derived class is different from the virtual function fun_4() in the base class as prototypes of both functions are different.

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.

▼ Virtual function more information

Virtual Functions in Derived Classes

Last Updated : 16 May, 2022

In C++, once a member function is declared as a virtual function in a base class, it becomes virtual in every class derived from that base class. In other words, it is not necessary to use the keyword virtual in the derived class while declaring redefined versions of the virtual base class function.

For example, the following program prints "***C::fun() called***" as "***B::fun()***" becomes virtual automatically.

```
// C++ Program to demonstrate Virtual
// functions in derived classes
#include <iostream>
using namespace std;

class A {
public:
    virtual void fun() { cout << "\n A::fun() called ";
};

class B : public A {
public:
    void fun() { cout << "\n B::fun() called "; }
};

class C : public B {
            //c class exits b class data
public:
    void fun() { cout << "\n C::fun() called "; }
};

int main()
{
```

```
    // An object of class C
    C c;

    // A pointer of class B pointing
    // to memory location of c
    B* b = &c;

    // this line prints "C::fun() called"
    b->fun();

    getchar(); // to get the next character
    return 0;
}
```

Note: Never call a virtual function from
a **CONSTRUCTOR** or **DESTRUCTOR**

Now me thinking:

```
class A {
public:
    virtual void fun() { cout << "\n A::fun() virtual ca
};

class B : public A {
public:
    void fun() { cout << "\n B::fun() called "; }
};

class C : public B {
public:
    void fun() { cout << "\n C::fun() called "; }
    void dis(){cout<<"ok";}
};

int main()
{
```

```
    // An object of class C
    //all data is transfer c class.
    C c;

    //from C class
    c.A::fun(); //A class virtual function call

    //Class C function call
    c.fun();

    //Class B function call]
    c.B::fun();



    // A pointer of class B pointing
    // to memory location of c
    B* b = &c;
    b->fun(); //class c function call

    return 0;
}
```

Dont understand

Can Virtual Functions be Inlined

By default, all the functions defined inside the class are implicitly or automatically considered as inline except virtual functions.

> *Note: inline is a request to the compiler and its compilers choice to do inlining or not.*

Whenever a virtual function is called using a base class reference or pointer it cannot be inlined because the call is resolved at runtime, but whenever called using the object (without reference or pointer) of that class, can be inlined because the compiler knows the exact class of the object at compile time.

```
class Base {
public:
    virtual void who() { cout << "I am Base\n"; }
};
class Derived : public Base {
public:
    void who() { cout << "I am Derived\n"; }
};

int main()
{
    // Part 1
    //call compail time so can be inlined
    Base b;
    b.who();

    // Part 2
    //call run time, so cannot be inlined
    Base* ptr = new Derived();
    ptr->who();
}
```

Virtual Destructor

When a derived class involves dynamic memory allocation, we have to deallocate the memory in its destructor.

Why i use virtual destructor :

Potential Issues:

- If the class `base` has a non-virtual destructor and is intended to be used polymorphically, there's a risk of undefined behavior.
- When deleting an object via a pointer to its base class, if the base class's destructor isn't virtual, only the base class's destructor will be called. This might result in resources not being properly released for derived classes.

- To ensure proper cleanup when deleting through a base class pointer, make the base class destructor virtual:

```
cppCopy code
class base {
public:
    virtual ~base() {} // Virtual destructor
};
```

Some Informaton :

```
   base *b=new base;
   delete b;
   /*here,
   *b = base type pointer variable
   new base = bynamically mermory allocate for base typ
   delete = frees the memory previously allocated for b
   */
```

Implementation virtual destructor function:

```
class base{
    public:
    base(){cout<<"Constructor base"<<"\n";}
    ~base(){cout<<"Destructor base"<<"\n";}
    };
  class derive:public base{
    public:
    derive(){
        m_array=new int[5];
        cout<<"Constructor derive"<<"\n";
        }
    ~derive(){
        delete[] m_array;
        cout<<"Destrucor derive"<<"\n";
    }
    private:
```

```
    int* m_array;
    };


    int main(){
    cout<<"------------------------"<<"\n";

    base* poly=new derive; //base classes constructor fu
                                                    //,memory
    delete poly;

    }
```

Output:
Constructor base
Constructor derive
Destructor base

[now that is problem] messing destructor functon

Explain:
here, m_array allocate memory inside constructor method.summary = memory allocate

and delete[] m_array; work destroy inside destructor function.
summary= delete memory space inside destructor

here,
call derive class constructor method,but dont call derive class destructor method, when call constructor method instant allocate memory in this function, memroy space space loss for dont call destructor method.

this problem resolved technique:use virtual destructor,below code

```
class base{
public:
base(){cout<<"Constructor base"<<"\n";}
```

```
virtual ~base(){cout<<"Destructor base"<<"\n";}
};
class derive:public base{
public:
derive(){
m_array=new int[5];
cout<<"Constructor derive"<<"\n";
}
~derive(){
delete[] m_array;
cout<<"Destrucor derive"<<"\n";
}
private:
int* m_array;
};

int main(){
cout<<"------------------------"<<"\n";
base* poly=new derive;
delete poly;
}
```

Ouput:

Constructor base
Constructor derive
Destrucor derive
Destructor base [Note : this line changed] now that is fine.

Virtual Constructor

object creation and object type are tightly coupled which forces modifications to extended. The objective of the virtual constructor is to decouple object creation from its type.

Do not understand ???

see :https://www.youtube.com/watch?v=dPNT5u1D_PE&ab_channel=Jaya'sNetworks

Virtual Copy Costructor

Dot understand

See agin

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.

Pure virtual functions are used

- if a function doesn't have any use in the base class
- but the function must be implemented by all its derived classes

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

    /* Other members */
};
```

Code :

```
class base{
    public:
   virtual void show()=0; //declaration
};
```

```
class derive:public base{
    public:
    void show(){ //defination virtual function
        cout<<"this is derive class"<<"\n";
    }
};

int main(){
    derive d;
    d.show(); //That is fine code
}
```

Some Interesting Facts :

- **We can have pointers and references of abstract class type.**

```
class base{
    public:
   virtual void show()=0; //declaration
};

class derive:public base{
    public:
    void show(){ //defination virtual function
        cout<<"this is derive class"<<"\n";
    }
};

int main(){

    // creating a pointer of type
    // Base pointing to an object
    // of type Derived
    base* p=new derive;
    p->show(); //call pure virtual function
}
```

- **If we do not override the pure virtual function in the derived class, then the derived class also becomes an abstract class.**

```
class Base {
public:
    // pure virtual function
    virtual void show() = 0;
};

class Derived : public Base {
};

int main(void)
{
    // creating an object of Derived class
    Derived d;

    return 0;
}
```

Output

```
Compiler Error: cannot declare variable 'd' to be
of abstract type
'Derived'  because the following virtual functions
are pure within
'Derived': virtual void Base::show()
```

- **An abstract class can have constructors**

```
class Base {
protected:
    int x;
public:
    // pure virtual function
    virtual void show()= 0;

    // constructor of Base class
    Base(int i)
```

```
    {
        x = i;
        cout << "Constructor of base called\n";
    }
};

class Derived : public Base {
    private:
    int y;

public:
    // calling the constructor of Base class
    Derived(int i, int j):Base(i){ //Derive construct
                                    //Initialization
                                    //This line initi
                                     //It's calling a
                                      //This is how y
        y=j;
    }




    // implementation of pure virtual function
    void show()
    {
        cout << "x = " << x << ", y = " << y << '\n';
    }
};

int main(void)
{
    // creating an object of Derived class
    Derived d(4,54);

    // calling the show() function of Derived class
    d.show();
```

```
    // creating an object of Derived class using
    // a pointer of the Base class
    Base* ptr = new Derived(6, 7);

    // calling the show() function using the
    // pointer
    ptr->show();

    return 0;
}
```

Pure Virtual Destructor

a pure virtual destructor, you must specify a destructor body.

> Note: Only Destructors can be Virtual. Constructors cannot be declared as virtual, this is because if you try to override a constructor by declaring it in a base/super class and call it in the derived/sub class with same functionalities it will always give an error as overriding means a feature that lets us to use a method from the parent class in the child class which is not possible.

```
// C++ program to demonstrate if the value of
// of pure virtual destructor are provided
// then the program compiles & runs fine.

#include <iostream>
class Base {
public:
    virtual ~Base() = 0; // Pure virtual destructor

};
Base::~Base() // Explicit destructor call
{
```

```
    std::cout << "Pure virtual destructor is called";
}

// Initialization of derived class
class Derived : public Base {
public:
    ~Derived() { std::cout << "~Derived() is executed\n"
};

int main()
{
    // Calling of derived member function
    Base* b = new Derived();
    delete b;
    return 0;
}
```

How did the above code work MAGICALLY?

This basically works because the destructors will be called recursively bottom to up if and only if the value is passed in the virtual destructor. So ***vtable*** is a table containing pointers of all virtual functions that the class defines, and the compiler provides ***vptr*** to the class as a '*hidden pointer*' that points to the ideal vtable, so the compiler makes use of an accurate or correct index, calculated at compile-time, to the vtable which will dispatch the correct virtual function at runtime.

It is ***important to note*** that a class becomes an abstract class(at least a function that has no definition) when it contains a pure virtual destructor.

Can Static Functions Be Virtua

In C++, a static member function of a class cannot be virtual. Virtual functions are invoked when you have a pointer or reference to an instance of a class. Static functions aren't tied to the instance of a class but they are tied to the class. C++ doesn't have pointers-to-class, so there is no scenario in which you could invoke a static function virtually.

```
// CPP Program to demonstrate Virtual member functions
// cannot be static
#include <iostream>

using namespace std;

class Test {
public:
    virtual static void fun() {}
};
```

Also, static member function cannot be **const** and **volatile**. Following code also fails in compilation,

```
// CPP Program to demonstrate Static member function car
// be const
#include <iostream>

using namespace std;

class Test {
public:
    static void fun() const {}
};
```

RTTI (Run-Time Type Information) in C++

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:

- **Upcasting:** When a pointer or a reference of a derived class object is treated as a base class pointer.
- **Downcasting:** When a base class pointer or reference is converted to a derived class pointer.

Using *'dynamic_cast'*: In an inheritance hierarchy, it is used for downcasting a base class pointer to a child class. On successful casting, it returns a pointer of the converted type and, however, it fails if we try to cast an invalid type such as an object pointer that is not of the type of the desired subclass.

For example, dynamic_cast uses RTTI and the following program fails with the error "***cannot dynamic_cast `b' (of type `class B*') to type `class D*' (source type is not polymorphic)*** " because there is no virtual function in the base class B.

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

// initialization of base class
class B {};

// initialization of derived class
class D : public B {};

// Driver Code
int main()
{
    B* b = new D; // Base class pointer
    D* d = dynamic_cast<D*>(b); // Derived class pointer
    if (d != NULL)
        cout << "works";
    else
        cout << "cannot cast B* to D*";
    getchar(); // to get the next character
```

```
    return 0;
}
```

Adding a **virtual function** to the base class B makes it work.

```
/ C++ program to demonstrate
// Run Time Type Identification successfully
// With virtual function

#include <iostream>
using namespace std;

// Initialization of base class
class B {
    virtual void fun() {}
};

// Initialization of Derived class
class D : public B {
};

// Driver Code
int main()
{
    B* b = new D; // Base class pointer
    D* d = dynamic_cast<D*>(b); // Derived class pointer
    if (d != NULL)
        cout << "works";
    else
        cout << "cannot cast B* to D*";
    getchar();
    return 0;
}
```

▼ This Pointer and Scope Resulation

This Pointer:

To repersent an object that invokes the member function (member ফাংশন আহ্বান করে তার object কে প্রতিনিধিত্ব করতে)

বা একই ক্লাসের মধ্যে সেম ভেরিয়েবল এর এড্রেস কে আলাদাভাবে আইডেন্টিফাই করে।

it automaticly points to the the object for which the object is being called.

```
//syntax for referring instance variable :

this -> class_variable=value ;

//Syntax for referring current objective class :

this* ;
```

Working on this pointer :

This keyword represents the address of the current instance of the class.

The this pointer acts an implicit (not in a Direct way) parameter all to the member functions and it is automatically passed to the Member function when it is called.

- **When local variable’s name is same as member’s name**

```
class A{
    private:
    int x;
    public:
    void math(int x){
        // The 'this' pointer is used to retrieve the
       // hidden by the local variable 'x'
       this->x=x;
    }
    void dis(){
        cout<<x;
    }
```

```
};
```

- Chain method is very useful Feture a this pointer,here must be use reference

  When a reference to a local object is returned, the returned reference can be used to chain function calls on a single object.

```
class Test{

    private:
    int x,y;
    int num;

    public:
    Test(int x=0,int y=0){
        this->x=x;
        this->x=y;
    }

    Test &setX(int a){x=a; return *this;}
    Test &setY(int b){y=b; return *this;}

    void dis(){
        cout<<x<<" "<<y;
    }
};

int main(){
    Test obj(10,20);
     // Chained function calls.  All calls modify the
  // as the same object is returned by reference
    obj.setX(200).setY(400).dis();
// C++ program to show use of this to access member w
// there is a local variable with same name
#include <iostream>
using namespace std;
```

```
class Test {
    int a;

public:
    Test() { a = 1; }
    // Local parameter 'a' hides object's member
    // 'a', but we can access it using this.
    void func(int a) {

     cout << this->a<<"\n";
     cout<<a<<"\n";
     }
};

// Driver code
int main()
{
    Test obj;
    int k = 3;
    obj.func(k);
    return 0;
}
}
```

- same time data receive and change and return

```
class Test{
    int num;
    public:
    Test & assign(int num){
        //Reference to the calling object can be retu
        this->num=num;
        return *this;
        //object value recive and return.
    }

    void dis(){
```

```
        cout<<num;
    }
};

int main(){
    Test obj;
    obj.assign(110).dis();
}
```

- Use function

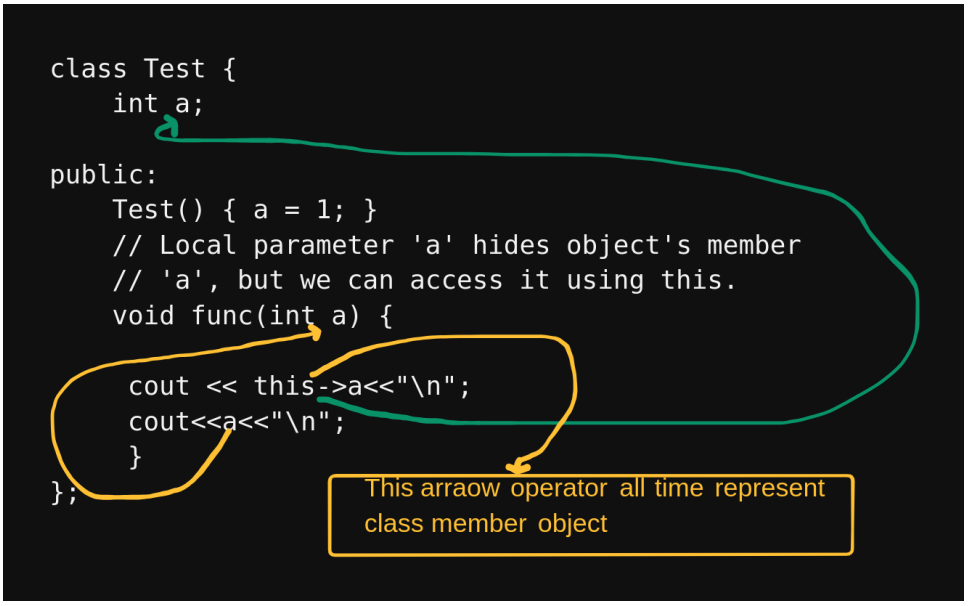
```
class king{
    public:
    void dis();
    void text(){
        //here text function is call dis function
        this->dis();
    }
};
void king::dis(){
    cout<<"A"<<"\n";
    cout<<"B"<<"\n";
    cout<<"C"<<"\n";
}

int main(){
    king obj;
    obj.text();
}
```

More information add:

This (- >) operator:

```
// C++ program to show use of this to access member wher
// there is a local variable with same name
#include <iostream>
using namespace std;
class Test {
```

```
    int a;

public:
    Test() { a = 1; }
    // Local parameter 'a' hides object's member
    // 'a', but we can access it using this.
    void func(int a) {

     cout << this->a<<"\n";
     cout<<a<<"\n";
     }
};

// Driver code
int main()
{
    Test obj;
    int k = 3;
    obj.func(k);
    return 0;
}
```



Scope resulation (: :) operator :

```
class Test {
    static int a;

public:
    // Local parameter 'a' hides class member
    // 'a', but we can access it using ::
    void func(int a) {
        cout << Test::a<<"\n";
        cout<<a;
    }
};

// In C++, static members must be explicitly defined
// like this
int Test::a = 1;

// Driver code
int main()
{
    Test obj;
    int k = 3;
    obj.func(k);
    return 0;
}
```

this operator and scope resulation operator is sam

▼ Exception Handling -1

Basic

What is a C++ Exception?

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).

and sunddely crush programm.

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.(human mistake)
2. **Asynchronous**: Exceptions that are beyond the program's control, such as disc failure, keyboard interrupts, etc.(computer mistakes)

Syntax of try-catch in C++

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

Multiple catch Statements:

n C++, we can use multiple `catch` statements for different kinds of exceptions that can result from a single block of code.

```
try {
    // code
}
catch (exception1) {
    // code
}
catch (exception2) {
    // code
}
catch (...) {
    // code
}
```

Here, our program catches `exception1` if that exception occurs. If not, it will catch `exception2` if it occurs.

If there is an error that is neither `exception1` nor `exception2`, then the code inside of `catch (...) {}` is executed.

Notes:

- `catch (...) {}` should always be the final block in our `try...catch` statement. This is because this block catches all possible exceptions and acts as the default `catch` block.
- It is not compulsory to include the default `catch` block in our code.

throw keywoed:

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.

> *Note: Multiple catch statements can be used to catch different type of exceptions thrown by try block.*

The try and catch keywords come in pairs: We use the try block to test some code and If the code throws an exception we will handle it in our catch block.

Why do we need Exception Handling in C++?

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

1. **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.
2. **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).

3. **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.

Limitations of Exception Handling in C++

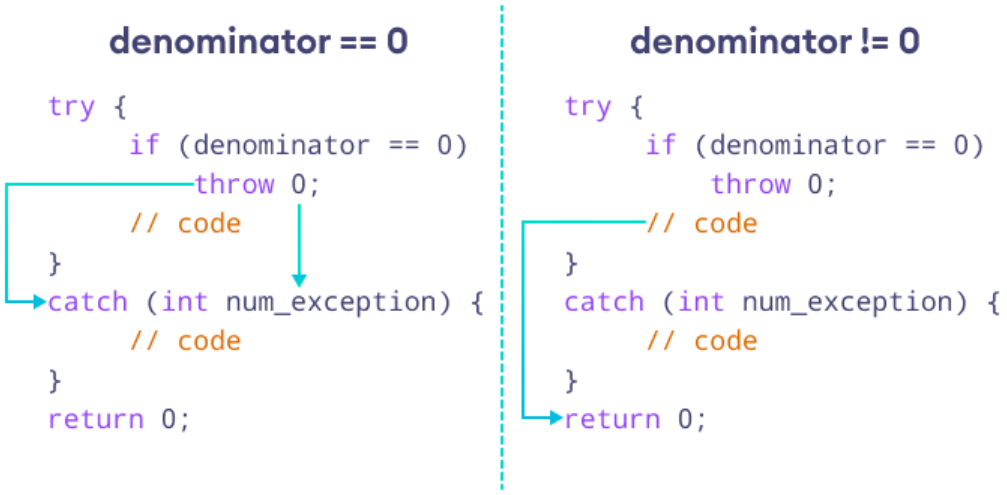
The exception handling in C++ also have few limitations:

- 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.

Conclusion

Exception handling in C++ is used to handle unexpected happening using “try” and “catch” blocks to manage the problem efficiently. This exception handling makes our programs more reliable as errors at runtime can be handled separately and it also helps prevent the program from crashing and abrupt termination of the program when error is encountered.

NOW CODE:



The following examples demonstrate how to use a try-catch block to handle exceptions in C++.

```
int main()
{
    int numerator = 10;
        int denominator = 0;
        int res;

    try {
        // check if denominator is 0 then throw runtime
        if (denominator == 0) {
            throw runtime_error("Division by zero not al
            //runtime_error is library exception type.
        }

        // 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) { //runtime_error type re
```

```
        // print the exception
        cout << "Exception " << e.what() << endl;
        // e.what() returns a C-style string describing
    }

    return 0;
}
```

```
// C
#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;
}
```

Properties of Exception Handling in C++

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.

```
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.

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

int main()
{
    try {
        throw 'a';
    }
    catch (int x) {
        cout << "Caught " << x;
    }
    catch (...) {
        cout << "Default Exception\n";
    }
    return 0;
}
```

Property 3

> If an exception is thrown and not caught anywhere, the program terminates abnormally.

```
int main()
{
    try {
        throw 'a';
    }
    catch (int x) {
        cout << "Caught ";
    }
    return 0;
}
```

Output

```
terminate called after throwing an instance of 'char'
```

We can change this abnormal termination behavior by writing our unexpected function.

> Note: A derived class exception should be caught before a base class exception.

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.

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

// This function signature is fine by the compiler, but
// recommended. Ideally, the function should specify all
// uncaught exceptions and function signature should be
// "void fun(int *ptr, int x) throw (int *, int)"
void fun(int* ptr, int x)
{
    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()
```

A better way to write the above code:

```
#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()
```

> Note: The use of Dynamic Exception Specification has been deprecated since C++11. One of the reasons for it may be that it can randomly abort your program. This can happen when you throw an exception of another type which is not mentioned in the dynamic exception specification. Your program will abort itself because in that scenario, it calls (indirectly) terminate(), which by default calls abort().

Property 5

> In C++, try/catch blocks can be nested. Also, an exception can be re-thrown using “throw; “.

```
// C++ program to demonstrate try/catch blocks can be ne
// 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
```

A function can also re-throw a function using the same “throw; ” syntax.
A function can handle a part and ask the caller to handle the 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.

```
#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
```

Exception Handling using classes or exception throw inside class

Problem and soloution:

Problem Statement:

- Create a class Numbers which has two data members **a** and **b**.
- Write iterative functions to find the **GCD of two numbers**.

- Write an iterative function to check whether any given number is prime or not. If found to be **true**, then throws an exception to class **MyPrimeException**.
- Define your own **MyPrimeException** class.

Solution:

- Define a class named **Number** which has two private data members as **a** and **b**.
- Define two member functions as:
  - **int gcd():** to **calculate the HCF of the two numbers**.
  - **bool isPrime():** to **check if the given number is prime or not**.
- Use **constructor** which is used to initialize the data members.
- Take another class named **Temporary** which will be called when an exception is thrown.

```
#include<bits/stdc++.h>
using namespace std;

class Number{
    private:
    int a,b;

    public:
    //number insert in constructor method
    Number(int x,int y){
        a=x;
        b=y;
    }

    //find gcd
    int gcd(){
       while (a!=b)
       {

        if(a>b){
            a=a-b; //update value of a
```

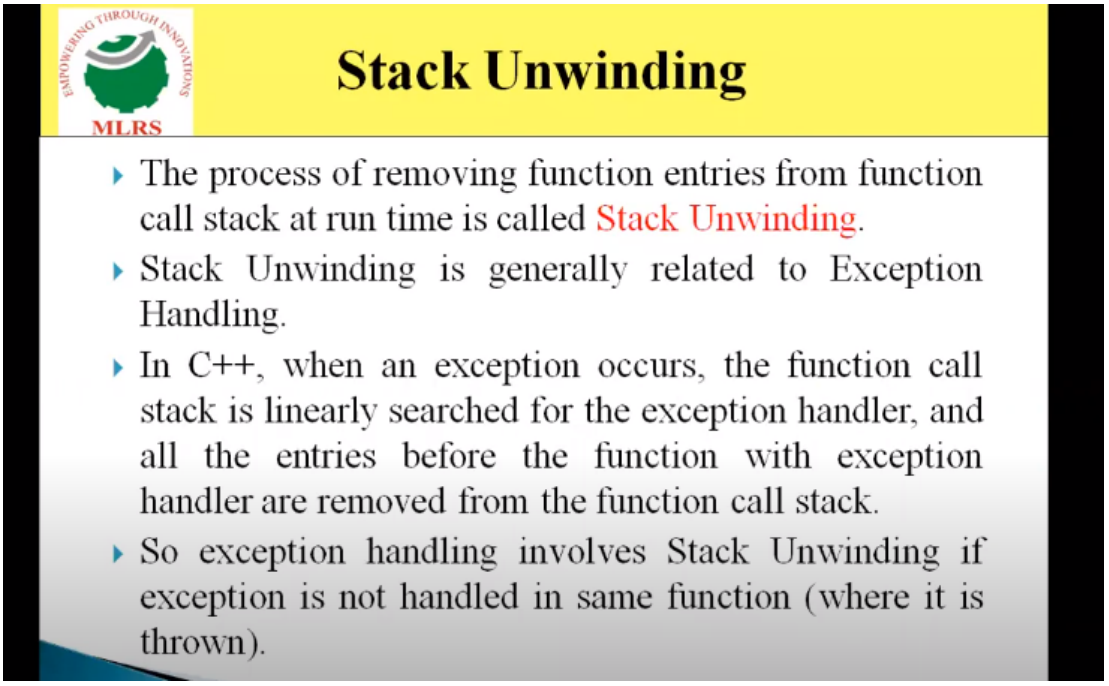
```
        }
        else
            b=b-a;
       }//end while
        return a;
    }

    //check prime
    bool check_prime(int n){
        if(n<1) return false;
        else if(n>1){

        }
        for (int i = 2; i<n; i++)
        {
            if (n%i==0)
            {
                return false;
            }

        }

        return true;
    }
};

//create empty class
class myprimeexception{};

int main(){

    int x,y;
    cout<<"Enter value:"<<"\n";
    cin>>x>>y;

    Number obj(x,y);

    //call and print gcd value
```

```
    obj.gcd();
    cout<<"Gcd value is="<<obj.gcd()<<"\n";

    //x is check weather prime or not prime
    if(obj.check_prime(x))cout<<x<<" "<<"is prime"<<"\n"
    //y is check weather prime or not prime
    if(obj.check_prime(y))cout<<y<<" "<<"is prime"<<"\n"

    //now hundle exception
    //here,one prime number is between x and y,instant w
    if(obj.check_prime(x)||obj.check_prime(y)){
        try{
            //throw exception in class
            throw myprimeexception();
        }
        catch(myprimeexception t){
            cout<<"Exception is cought"<<"\n";
            cout<<"here x or y is prime number"<<"\n";
        }
        }

    return 0;
}
```

▼ Exception Handling -2

Stack Unwinding

Stack Unwinding is the process of removing function entries from function call stack at run time. The local objects are destroyed in reverse order in which they were constructed.




```
// CPP Program to demonstrate Stack Unwinding
#include <iostream>
using namespace std;

// A sample function f1() that throws an int exception
void f1() throw(int)
{
    cout << "\n f1() Start ";
    throw 100;
    cout << "\n f1() End ";
}

// Another sample function f2() that calls f1()
void f2() throw(int)
{
    cout << "\n f2() Start ";
    f1();
    cout << "\n f2() End ";
}

// Another sample function f3() that calls f2() and hanc
// exception thrown by f1()
void f3()
```

```
{
    cout << "\n f3() Start ";
    try {
        f2();
    }
    catch (int i) {
        cout << "\n Caught Exception: " << i;
    }
    cout << "\n f3() End";
}

// Driver Code
int main()
{
    f3();

    getchar();
    return 0;
}
```

Explanation:

- When f1() throws exception, its entry is removed from the function call stack, because f1() doesn't contain exception handler for the thrown exception, then next entry in call stack is looked for exception handler.
- The next entry is f2(). Since f2() also doesn't have a handler, its entry is also removed from the function call stack.
- The next entry in the function call stack is f3(). Since f3() contains an exception handler, the catch block inside f3() is executed, and finally, the code after the catch block is executed.

Note that the following lines inside f1() and f2() are not executed at all.

```
 cout<<"\n f1() End ";  // inside f1()

 cout<<"\n f2() End ";  // inside f2()
```

If there were some local class objects inside f1() and f2(), destructors for those local objects would have been called in the Stack Unwinding process.

> Note: Stack Unwinding also happens in Java when exception is not handled in same function.

Multiple catch Statements

In C++, we can use multiple `catch` statements for different kinds of exceptions that can result from a single block of code.

```
try {
    // code
}
catch (exception1) {
    // code
}
catch (exception2) {
    // code
}
catch (...) { //received anytype exception
    // code
}
```

throw exception from try block,যে টাইপের এক্সেপশন, সেই টাইপের catch block সেটি রিসিভ করবে.

```
#include<bits/stdc++.h>
using namespace std;
int main(){

    double num1,num2,arr[4]={};//0.0,0.0,0.0,0.0
    int index;

    cout<<"Enter index:";
    cin>>index;
```

```
    try{
        //check ranged of array
        if(index>=4){
            throw "Error:out of array renged";
        }
        /*Note:
        if executed if condition upper,instant exit
        try block,that means when true condition instant

        */

        //insert value
        cout<<"Value of number1:";
        cin>>num1;
        cout<<"Value of number2:";
        cin>>num2;

        if(num2==0){
            throw 0;
        }

        //annswer is stored in array index and print out
        arr[index]=num1/num2;
        cout<<"Divide answer:"<<arr[index];
    }//colosed tryblock

    //catched throwing messaged.
    catch(const char* messages){
        cout<<messages<<"\n";
    }
    //catch o value
    catch(int num){
        cout<<"Error:can not devided by"<<num<<"\n";
    }
    //catch any other exception
    catch(...){
        cout<<"Unexcepted exception"<<"\n";
```

```
    }

    return 0;
    }
```

USER DEFIND EXCEPTION HUNDLEING

- Program to implement exception handling with single class

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

class demo {
};

int main()
{
    try {
        throw demo();
    }

    catch (demo d) {
        cout << "Caught exception of demo class \n";
    }
}
```

- Program to implement exception handling with two class

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

class demo1 {
};

class demo2 {
};
```

```
int main()
{
    for (int i = 1; i <= 2; i++) {
        try {
            if (i == 1)
                throw demo1();

            else if (i == 2)
                throw demo2();
        }

        catch (demo1 d1) {
            cout << "Caught exception of demo1 class
        }

        catch (demo2 d2) {
            cout << "Caught exception of demo2 class
        }
    }
}
```

- Use Inheritance

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

class demo1 {
};

class demo2 : public demo1 {
};

int main()
{
    for (int i = 1; i <= 2; i++) {
        try {
            if (i == 1)
                throw demo1();
```

```
            else if (i == 2)
                throw demo2();
        }

        catch (demo1 d1) {
            cout << "Caught exception of demo1 class
        }
        catch (demo2 d2) {
            cout << "Caught exception of demo2 class
        }
    }
}
```

- Use Constructor

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

class demo {
    int num;

public:
    demo(int x)
    {
        try {

            if (x == 0)
                // catch block would be called
                throw "Zero not allowed ";

            num = x;
            show();
        }

        catch (const char* exp) {
            cout << "Exception caught \n ";
```

```
            cout << exp << endl;
        }
    }

    void show()
    {
        cout << "Num = " << num << endl;
    }
};

int main()
{

    // constructor will be called
    demo(0);
    cout << "Again creating object \n";
    demo(1);
}
```

C++ Standard Exception

C++ has provided us with a number of standard exceptions that we can use in our exception handling. Some of them are shown in the table below.

Exception	Description
`std::exception`	The parent class of all C++ exceptions.
`std::bad_alloc`	Thrown when a dynamic memory allocation fails.
`std::bad_cast`	Thrown by C++ when an attempt is made to perform a `dynamic_cast` to an invalid type.
`std::bad_exception`	Typically thrown when an exception is thrown and it cannot be rethrown.

▼ Template

Basic

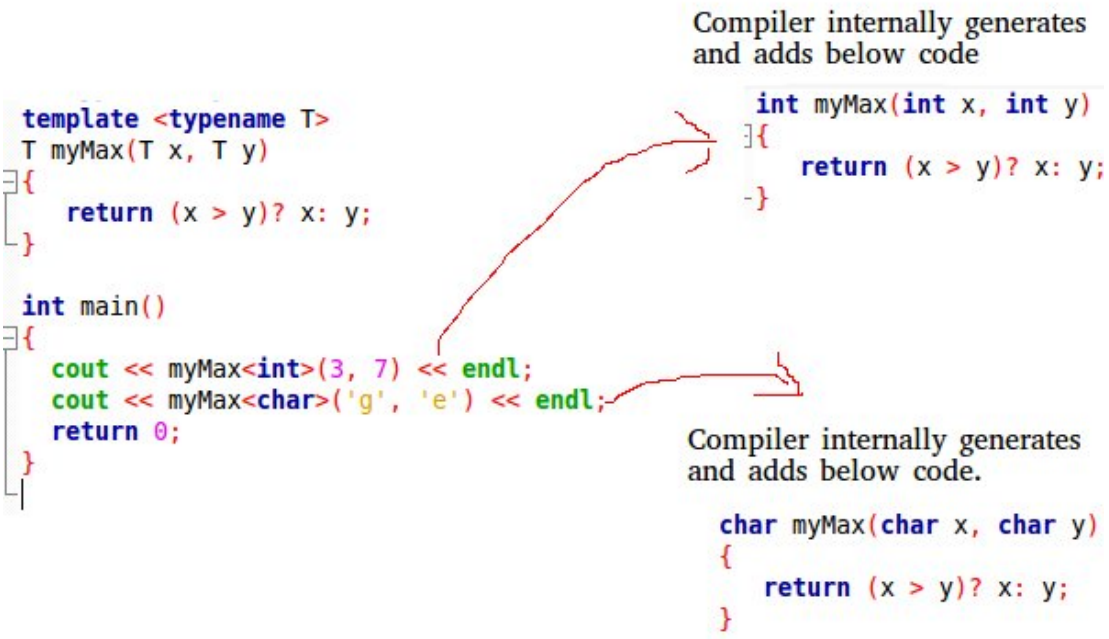
A **template** is a simple yet very powerful tool in C++. The simple idea is to pass the data type as a parameter so that we don’t need to write the

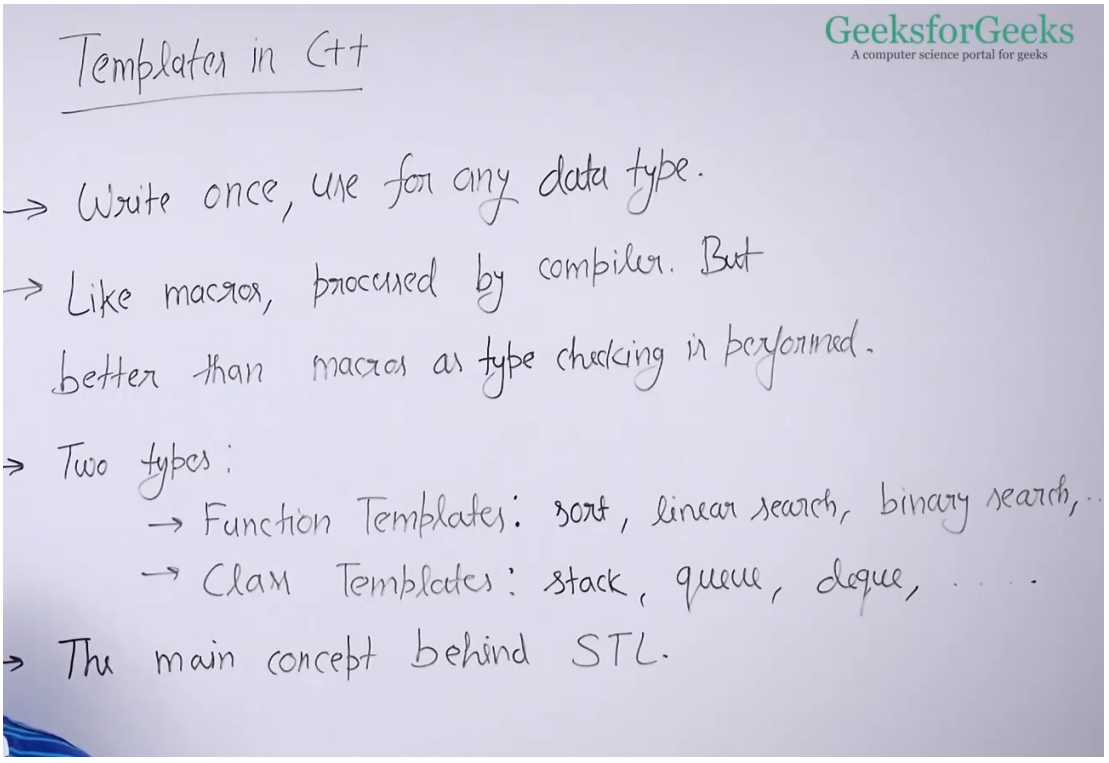
same code for different data types. For example, a software company may need to sort() for different data types. Rather than writing and maintaining multiple codes, we can write one sort() and pass the datatype as a parameter.

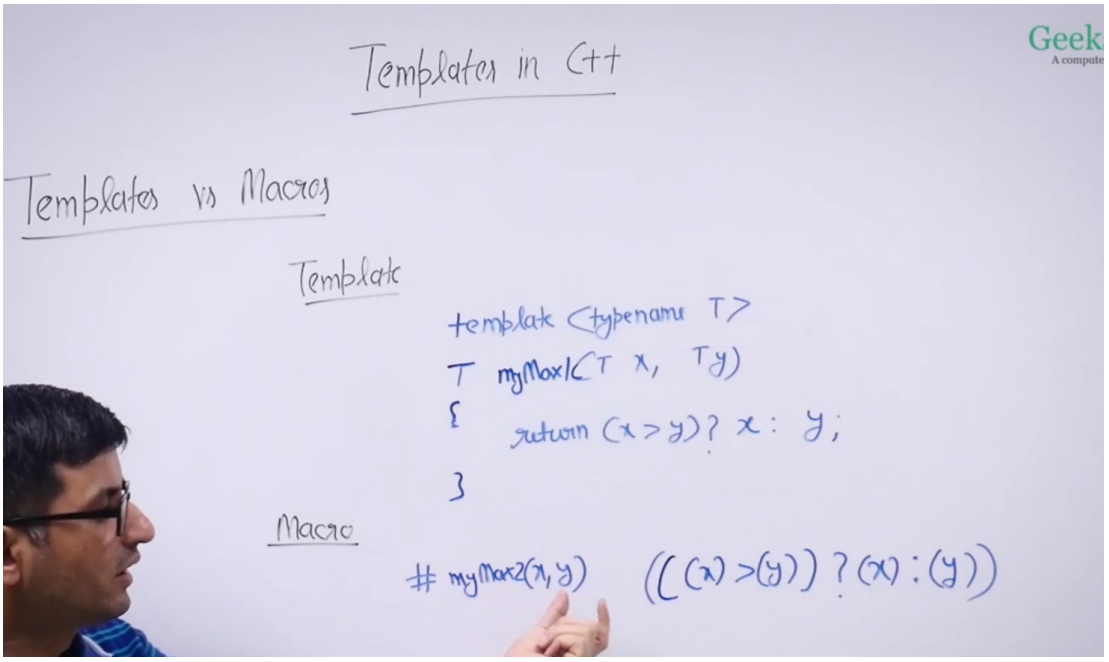
C++ adds two new keywords to support templates: ***'template'*** and ***'typename'***. The second keyword can always be replaced by the keyword **'class'**.

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.







Function Templates

We write a generic function that can be used for different data types. Examples of function templates are sort(), max(), min(), printArray()

```
#include<bits/stdc++.h>
using namespace std;
```

```
//templete declraration
template<typename T>
//functon declaration
T mysum(T a,T b){
    T sum=a+b;
    return sum;
}

int main(){
    //call int type data in template functon
    cout<<mysum<int>(3,7)<<"\n";
    //call by float type
    cout<<mysum<float>(1.7,9.6)<<"\n";
    //call by double type
    cout<<mysum<double>(1.75,9.36)<<"\n";
}
```

Example: Implementing **Bubble Sort** using templates in C++

```
// 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]);
}

// Driver Code
```

```
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;
}
```

Class Templates

Class templates like function templates, class templates are useful when a class defines something that is independent of the data type. Can be useful for classes like LinkedList, BinaryTree, Stack, Queue, Array, etc.

```
#include<bits/stdc++.h>
using namespace std;

//class type template
template<typename T>
class Array{
    private:
    T* ptr; //pointer variable
    int size;//array size

    public:
    //constructor method and array declaration
    Array(T arr[],int s);
    void display();
};
 //constructor method and array defination
 //template<typename T>: this is template
```

```
// Array<T> :T type class

template<typename T>Array<T>::Array(T arr[],int s){
   ptr=new T[s];//dynamic memory allocation pointer typ
   size=s;//array size

   for (int i = 0; i < size; i++)
   {
       ptr[i]=arr[i]; //normal array value store in poi
   }
}
template<typename T> void Array<T>::display(){
   for (int i = 0; i < size; i++)
   {
       cout<<*(ptr+i)<<" "; //dont understand line
   }
}

int main(){
   int a[5]={1,2,3,4,5}; //create array in of user

   //create object of Array class
   Array<int>obj(a,5);
   obj.display();
}
```

Can there be more than one argument for templates?

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

```
#include<bits/stdc++.h>
using namespace std;

template<class U,class V> class Myclass{
   private:
   U x;
   V y;
   public:
```

```
    Myclass(){
        cout<<"constructor method is called"<<"\n";
    }
};
int main(){
    //pass multiple argumnet in template
    Myclass<double,char>obj1;
    Myclass<float,int>obj2;
}
```

Can we specify a default value for template arguments?

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

```
#include<bits/stdc++.h>
using namespace std;
                      // specify a default value for te
template<class U,class V=char> class Myclass{
    private:
    U x;
    V y;
    public:
    Myclass(){
        cout<<"constructor method is called"<<"\n";
    }
};
int main(){
    //pass multiple argumnet in template

    //bleow line act is Myclass<double,char>obj1;
    Myclass<double>obj1;

}
```

What is the difference between function overloading and templates?

Both function overloading and templates are examples of polymorphism features of OOP. Function overloading is used when multiple functions do quite similar (not identical) operations, templates are used when multiple functions do identical operations.

What happens when there is a static member in a template class/function?

Each instance of a template contains its own static variable.
See **Templates and Static variables** for more details.

What is template specialization?

Can we pass non-type parameters to templates?

We can pass non-type arguments to templates. Non-type parameters are mainly used for specifying max or min values or any other constant value for a particular instance of a template. The important thing to note about non-type parameters is, that they must be const. The compiler must know the value of non-type parameters at compile time. Because the compiler needs to create functions/classes for a specified non-type value at compile time. In the below program, if we replace 10000 or 25 with a variable, we get a compiler error.

```
// C++ program to demonstrate
// working of non-type parameters
// to templates in C++
#include <iostream>
using namespace std;

template <class T, int max> int arrMin(T arr[], int n)
{
    int m = max;
    for (int i = 0; i < n; i++)
        if (arr[i] < m)
            m = arr[i];

    return m;
```

```
}

int main()
{
    int arr1[] = { 10, 20, 15, 12 };
    int n1 = sizeof(arr1) / sizeof(arr1[0]);

    char arr2[] = { 1, 2, 3 };
    int n2 = sizeof(arr2) / sizeof(arr2[0]);

    // Second template parameter
    // to arrMin must be a
    // constant
    cout << arrMin<int, 10000>(arr1, n1) << endl;
    cout << arrMin<char, 256>(arr2, n2);

    return 0;
}
```

Output

```
10
1
```

Here is an example of a C++ program to show different data types using a constructor and template. We will perform a few actions

- passing integer value by creating an object in the main() function.
- passing character value by creating an object in the main() function.
- passing float value by creating an object in the main() function.

```
// C++ program to show different data types using a
// constructor and template.
#include <iostream>
using namespace std;

// defining a class template
```

```
template <class T> class info {
public:
    // constructor of type template
    info(T A)
    {
        cout << "\n"
             << "A = " << A
             << " size of data in bytes:" << sizeof(A);
    }
    // end of info()
}; // end of class

// Main Function
int main()
{
    // clrscr();

    // passing character value by creating an objects
    info<char> p('x');

    // passing integer value by creating an object
    info<int> q(22);

    // passing float value by creating an object
    info<float> r(2.25);

    return 0;
}
```

▼ Generics in C++

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

The method of Generic Programming is implemented to increase the efficiency of the code. Generic Programming enables the programmer to write a general algorithm which will work with all data types. It eliminates

the need to create different algorithms if the data type is an integer, string or a character.

The advantages of Generic Programming are

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

Generics can be implemented in C++ using **Templates**. Template is a simple and yet very powerful tool in C++. The simple idea is to pass 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 sort() for different data types. Rather than writing and maintaining the multiple codes, we can write one sort() and pass data type as a parameter.

Generic Functions using Template:

We write a 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 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.

Following is a simple example of template Array class.

```
#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
```

▼ Some Technique VVI

- parameter object pass :

```
class drive{
    public:
    int math1(int a){
        return a;
    }
    int math2(int b){
        return b;
    }
    int math3(int c){
        return c;
    }
};
int main(){
```

```
  drive obj;
  cout<<obj.math1(obj.math2(obj.math3(20)));
}
```

- Function one kind of variable

```

```

- Inside parameter create a class object

```
class A{
int a;
int math(A obj){
    obj.a;
}
};
```

- Create object without main function
- Data inilization method formate

```
// Initialize x
math(int val) {
    x = val;
}
// another way
math(int val) : x(val) {}
```

- Reference fact :

```
int math(int &f){
x=f; // x and f is same
    //becuse reference variable f contains xe
}

//Example:
-------------
int x;
void math(int &f){
x=f;
```

```
cout<<x<<" "<<f;
//here x and f is name
}
int main(){
    int p=10;
    math(p);

}
```

- Some technique

```
variable declaration:

Normal value reference:
int &j=24; //is not allowed
const int &j=24; //allowed

RValue reference:
 int &&j=24; //allowed


 #include<bits/stdc++.h>
using namespace std;

void dis(int & reference){cout<<"Normal value reference
void dis(int && reference){cout<<"RValue references";}
int main(){

    int i=10;
    dis(i);

    dis(100);
    return 0;
}
```

- p
- p

- p
- p
-

V.V.I Question and Answer ?

- multilevel inheritance same variable name,now i indentification witch variable witch classes and access data ?

  Use constructor method :

```
class A{
    public:
    int x;
    A(){
        x=30;
    }
};

class B:public A{
    public:
    int x;
    //Same variable name as in class A and B
    B(){
        x=10;
    }
};

class C:public B{
    public:
    int x;
    C(){
        x=20;
    }
    void disply(){
        cout<<A::x<<"\n"; //Accessing x from class A
        cout<<B::x<<"\n"; //same
        cout<<x<<"\n"; //same
    }
```

```
};

int main(){
    C obj;
    obj.disply();

}
```

- Functor not function ??

▼ **this read imaditely**

https://www.geeksforgeeks.org/inheritance-ambiguity-in-cpp/

Vtable and vtr in Virtual function??

▼ **Do’nt understand ??**

- **Virtual Functions in Derived Classes in C++ ??**
- **Virtual Constructor**
- copy **Virtual Constructor**

▼ function contains static variables.