Functions in C++

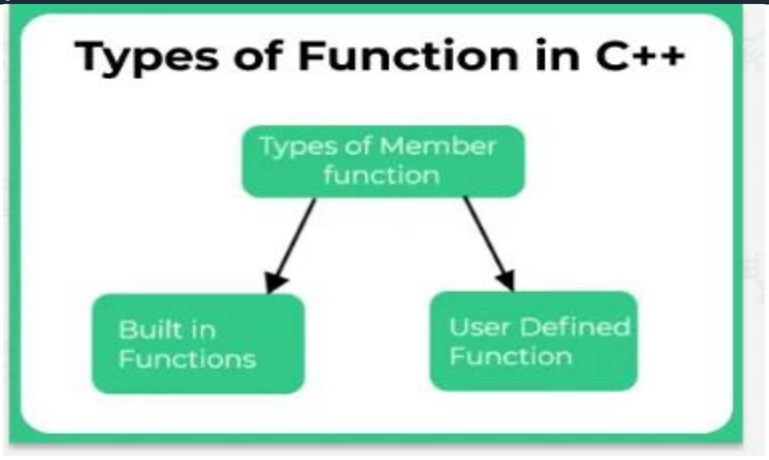
Functions in C++
Below is the syntax of the Functions in C++ with example

Syntax:

```
return_type func_name(args)//function definition
{
---
stmts;//fun_body
}
```

Example:

```
int add()//func_def
{
int a=2,b=3;//fun_body
return a+b;
}
```



Functions in C++ are defined in 2 ways:
1.Predefined functions

- These functions are been ***already defined by the designer*** and placed in the library
- We can make use of these library functions to get the pre-defined output instead of writing our own code to get those outputs.

- ex: *pow() in math.h ,printf() in stdio.h*

2.User defined functions

- These are ***defined and customized by the user*** per the requirement
- ex: add()

Function components

To write the programs on function on any language we should understand the following 8 components

1.Function declaration(prototype)

- Function declaration tells the compiler the *return type of the functions, number of arguments, function return type of the arguments and order of arguments*
- A function declaration can be inside or outside the main() but the before function call;

Syntax:

```
return_type func_name(arg1,arg2..argn);
```

ex:

```
int sum(int,int);
```

The above function takes 2 inputs of type int and returns the output of type int

2.Function definition

- The set of statements which ***defines the task of the function***
- The return type must be mentioned in function definition which is the same given in function prototype

```
int sum(int x,int y)// function defination
{
return x+y;
}
```

3.Function call

- The function call is a ***request to execute the function*** defined , till then function will not execute
- No return-type should be specified at the function call
- A user can call function any number of time and anywhere

```
int add(2,3);//invalid
add(2,3);//valid
```

4.Formal Parameters

- The ***parameters present in the function definition*** are formal parameters
- Formal parameters will receive value from the actual parameter
- Formal parameters must be variable type only

```
int sum(int x,int y)//func_defination
{
return x+y;
// x and y are formal parameters
}
```

5.Actual parameters

- ***Parameters passed during function call*** are called formal parameters
- Actual parameters make a communication to formal parameters
- They can be the variable, constant or expression type

```
add(2,3);//constant type actual parametres
add(x,y);//variable type actual parametres
add(2*3,7-2);//expression type actual parametres
```

6.Return statement

- Return type specifies the function output and allocates some space to function
- Return type also depends on arguments type

```
int sum(int,int);
float sum(int,float);
long fact(int);
```

- Generally, it is used at the function definition end, because it makes to exit the function module
- Another major purpose of the return statement, it ***exits from the function module and gives control back to the caller***

```
#include <iostream>
using namespace std;
int demo()//fun definition
{
cout<<"control acquired from main\n";
• return 1;
• //return0;error:multiple return types not allowed
• }
• int main()
• {
• cout<<"before call control under main\n";
• cout<<"controltranmsferreed from main to demo\n"<<demo();
• cout<<"control return back from demo to main()\n";
• return 0;
• }
```

- here function call at line demo() transfers function call to from main() to demo(), “return 1” transfers control back to main()
- Only one return statement is allowed in a function definition
- void return type function cannot use the return statement

```
void add()
{
return a+b;//error:void cannot return
}
```

7.Function call

It is the function in which a function call is made

```
main()//fun call
{
add(2,3);
}
```

- here main() is calle for add() function, main is requesting add() to execute

8.Function signature

It is function name excluding return type in the function definition

```
int add(int x,int y)
{
return a+b;
//add(int x,int y) is function signature
}
```

Functions in C++ can be classified in 4 ways

No arguments passed and no return value

```
#include <iostream>
using namespace std;
void prime();//func_protoytype
int main()
{
// No argument is passed to prime()
prime();//func_call
return 0;
}
// Return type of function is void because value is not returned.
void prime()
{
int num, i, flag = 0;
cout << "Enter a positive integer enter to check: "; cin >> num;
for(i = 2; i <= num/2; ++i)
{
if(num % i == 0)
{
flag = 1;
break;
}
}
if (flag == 1)
cout << num << " is not a prime number.";
else
cout << num << " is a prime number.";

}
```

- In the above program, prime() is called from the main() with no arguments.
- prime() takes the positive number from the user and checks whether a number is a prime number or not.
- Since return type of prime() is void, no value is returned from the function.

No arguments passed but a return value

```
#include <iostream>
using namespace std;
int prime();//func_protoytype
int main()
{
int num, i, flag = 0;
// No argument is passed to prime()
num = prime();//func_call
for (i = 2; i <= num/2; ++i)
{
if (num%i == 0)
{
flag = 1;
break;
}
}
if (flag == 1)
cout<<num<<" is not a prime number.";
else
cout<<num<<" is a prime number.";
return 0;
}
// Return type of function is int
int prime()//fun_def
{
int n;
printf("Enter a positive integer to check: ");
cin >> n;

return n;
}
```

- In the above program, prime() function is called from the main() with no arguments.prime() takes a positive integer from the user. Since return type of the function
- is an int, it returns the inputted number from the user back to the calling main() function.
- Then, whether the number is prime or not is checked in the main() itself and printed onto the screen.

Arguments passed but no return value

```
#include <iostream>
using namespace std;
void prime(int n);//fun_dec
int main()
{
int num;
cout << "Enter a positive integer to check: "; cin >> num;
// Argument num is passed to the function prime()
prime(num);//func_call
return 0;
}
/ There is no return value to calling function. /*Hence, return type of function is void. */
void prime(int n)//func_def
{
```

```
int i, flag = 0;
for (i = 2; i <= n/2; ++i)
{
if (n%i == 0)
{
flag = 1;
break;
}
}
if (flag == 1)
{
cout << n << " is not a prime number.";
}
else {
cout << n << " is a prime number.";
}
}
```

- In the above program, a positive number is first asked from the user which is stored in the variable num.
- Then, num is passed to the prime() function where, whether the number is prime or not is checked and printed.
- Since the return type of prime() is a void, no value is returned from the function.

Arguments passed and a return value.

```
#include <iostream>
using namespace std;
int prime(int n);//func_dec
int main()
{
int num, flag = 0;
cout << "Enter positive integer to check: "; cin >> num;
// Argument num is passed to check() function
flag = prime(num);//func_call
if(flag == 1)
cout << num << " is not a prime number.";
else
cout<< num << " is a prime number.";
return 0;
}
/* This function returns integer value. */
int prime(int n)//func_def
{
int i;
for(i = 2; i <= n/2; ++i)
{
```

```
if(n % i == 0)
return 1;
}
return 0;
}
```

- In the above program, a positive integer is asked from the user and stored in the variable num.
- Then, num is passed to the function prime() where, whether the number is prime or not is checked.
- Since the return type of prime() is an int, 1 or 0 is returned to the main() calling function. If the number is a prime number, 1 is returned. If not, 0 is returned.
- Back in the main() function, the returned 1 or 0 is stored in the variable flag, and the corresponding text is printed onto the screen.

Which style of functions should be used?

- All four programs above gives the same output and all are technically correct program.
- There is no hard and fast rule on which method should be chosen.
- The particular method is chosen depending upon the situation and how you want to solve a problem.

Advantages of Functions

- *Debugging*: Debugging becomes easier as error sticks to that particular module(function)
- *Sourcecode Reusability*: Once a function is called from any other module, without rewriting code again

Is main() user-defined or predefined function?

main()is a user-defined function

- The body of the main() is defined by the user as per the requirement
- The main() body can be customized as per the requirement at any time by the user
- The return type of main can be changed

Infinite Loop in Function

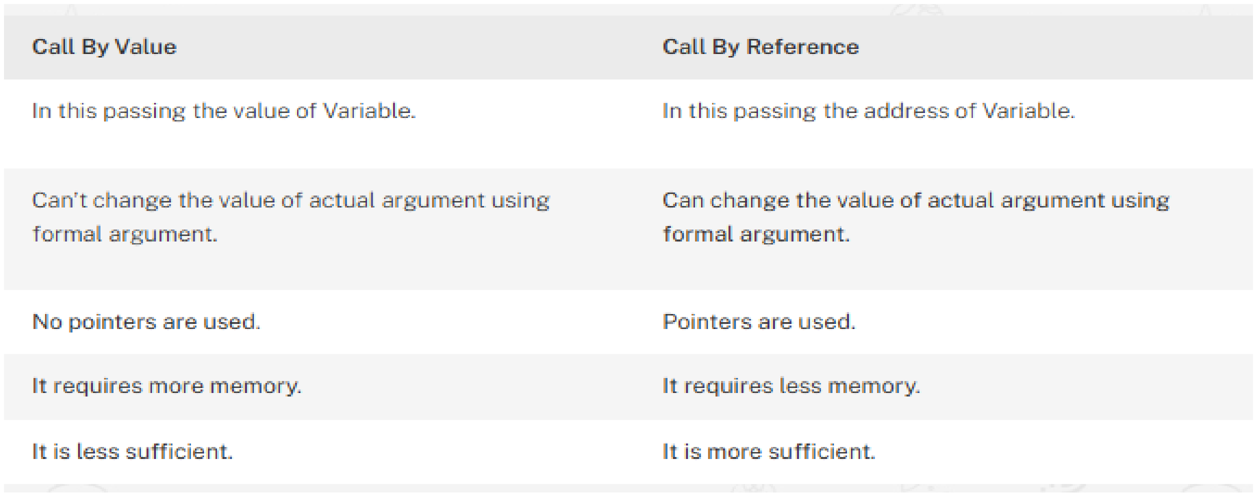
- This is very interesting to know that a function can cause an infinite loop
- This happens when ***recursive calls occur without any bounds*** i.e no tets condition that stops the function call

```
void display();//prototype
int main()
```

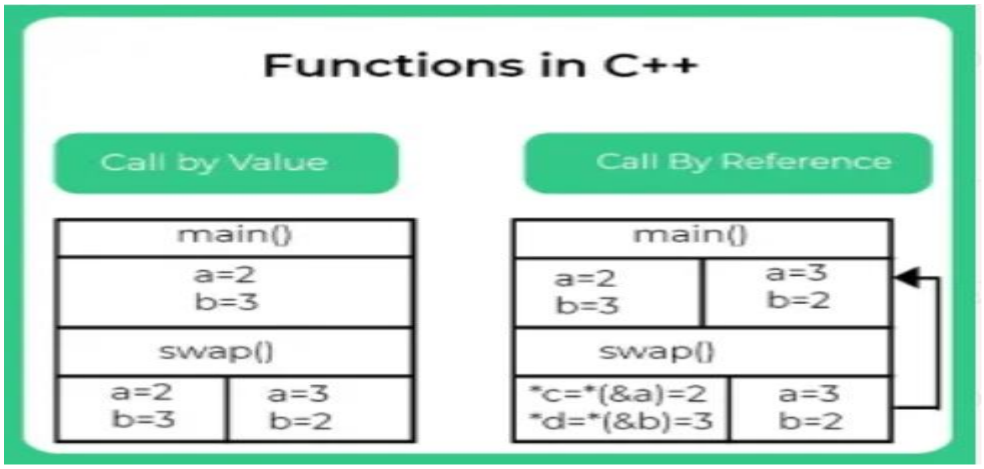
```
{
display();//call

}
void display()//function_def
{
cout<<"This will create infinite loop-3shaank";
display();//call itself
}
```

Difference Between Call by Value And Call by Reference



Call By Value	Call By Reference
In this passing the value of Variable.	In this passing the address of Variable.
Can't change the value of actual argument using formal argument.	Can change the value of actual argument using formal argument.
No pointers are used.	Pointers are used.
It requires more memory.	It requires less memory.
It is less sufficient.	It is more sufficient.



Call by Value

Example

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

```
void swap (int, int);
int main ()                                         //caller function
{
  int a = 2, b = 3;
  swap (a, b);                              //fun_call
  cout << "In calle:after swapping a=" << a << " b=" << b;
}

void
swap (int a, int b)
{
  b = (a + b) - (a = b);          //swap logic in a single line

  cout << "In called:after swapping a=" << a << " b=" << b;
  cout << endl;
}
```

call by address

- Sending the address of variables from caller to calle function is called call-by-address
- Any changes made in function definition will also affect the change in the caller function

Example

```
#include <iostream>
using namespace std;
void swap (int *x, int *y)
{
  int swap;
  swap = *x;
  *x = *y;
  *y = swap;
}

int main ()
{
  int x = 2, y = 3;
  cout << "Value of x before swap is: " << x << endl;
  cout << "Value of y before swap is: " << y << endl;
  swap (&x, &y);
  cout << "Value of x is: " << x << endl;
  cout << "Value of y is: " << y << endl;
  return 0;
}
```

Member Functions in C++

Member Functions in C++

It operates on an object of the class of which it is a member, and has access to all the members of a class for that object.

Lets have a look at member functions below –

DefinitionA member function of a class is a function that has its definition or its prototype within the class definition like any other variable. It operates on any object of the class of which it is a member, and has access to all the members of a class for that object.

Member Function inside the class

Lets look at an example below

```
class Cube {
    public:
        // member variables
        double len;
        double breadth;
        double height;

        // member function declared and defined
        // inside class
        double getVolume(){
            return len * breadth * height;
        }
};
```

Problem with function definition inside:

If a function defined inside the class, it becomes inline and implicit expansion takes place, hence performance decreases and recommended to define the function outside the class.

Member Function outside the class

Member functions are defined outside the class using scope resolution operator(::) as shown below –

```
return_Data_type class_name::function_name()
```

Let's look at an example below

```
class Cube {
    public:
        // member variables
        double len;
        double breadth;
        double height;
        // member function declared inside class
        // but defined outside
        double getVolume();
};

// Member function definitioned outside
// scope resolutor :: is used
double Cube::getVolume() {
   return length * breadth * height;
}
```

Accessing private member function of a class

- When a member function is public we can access that member fun directly by using (.) membership operator anywhere
    - Example cube_obj.Len will be valid if len is public
- However, a private member variable can not be accessed using dot(.) membership operator
    - Example : cube_obj.breadth will be invalid if breadth private

Hence A private member function is accessed by only another public function of the same class within the body of the class

NoteDefault access in C++ is private, which is visible only within the class and cannot be accessed outside the class

Example:-

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

class Cube {
        // member variables
        double len;
        double breadth;

    public:
        double height;
        // member function declared and defined
        // inside class
        double getVolume(){
            return len * breadth * height;
        }

        // setters : to set values declared and defined
        // inside class
        void setLen(int l){
            len = l;
        }
        void setBreadth(int b){
            breadth = b;
        }
        void setHeight(int h){
            height = h;
        }

        // getter functions created to get values
        int getLen(){
            return len;
        }
        int getBreadth(){
            return breadth;
        }
};

int main(){
    Cube cube_obj1;

    cube_obj1.setLen(10.0);
    cube_obj1.setBreadth(10.0);
    cube_obj1.setHeight(10.0);

    cout << "The Volume : " << cube_obj1.getVolume() << endl;

    cout << "The height : " << cube_obj1.height << endl;

    // will be able to access private variables
    // using these public member functions (getters)
    cout << "The breadth : " << cube_obj1.getBreadth() << endl;
    cout << "The Length : " << cube_obj1.getLen() << endl;
}
```

Static Keyword in C++

Static Data Members (Variables)

Inside a function

The static variable (data member), has a lifetime throughout 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.

Example

Here we have created a static variable count inside a function, value is carried over in every function call

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

void myFunction()
{
    // static variable,
    // will only get initialised in first function call
    static int count = 0;

    cout << count << " ";

    // value from previous function call is retained
    // will be carried to next function calls
    count++;
}

int main()
{
    for (int i = 0; i <= 10; i++)
        myFunction();

    return 0;
}
```

Inside a class

Generally, separate copies & memories of variables (data members) are created for each object for any given class.

However, when a variable (data member) is declared as static only a **single copy of memory is created for all the objects** & each object shares the same copy for the static variable.

Note – Static data members are class members, not object member

Basic Properties of Static data members

1. Static data members are initialized with zero automatically

2. Static data members can be called directly using the class name

Static Member functions

Definition: A static member function is **designed only for accessing static data members**, other static member functions and any other functions from outside the class.

- A static member function can be called ***even if no objects of the class exist***
- Static member functions have a class scope and do not have access to this pointer of the class.
- The static function can only access static members of the class

Static Objects

Very similar to static variables whose scope becomes till the lifetime of the program.

Similarly, for static objects, their scope becomes till just before the program terminates.

We can verify this with the following two programs below –

- **Example 1:** Without a static object
- **Example 2:** With a Static object

Example 1

Constructors/Destructor functioning without static objects

- When the object is not static, the object is destroyed as soon as its scope ends.
- In the example below destructor is called as soon as the scope of the object ends
- Signalling object being destroyed as soon as scope ended

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

class PrepInsta
{
    public:
    // constructor
    PrepInsta() {
        cout << "PrepInsta Constructed\n";
    }
•
•       // destructor
•       ~PrepInsta() {
•           cout << "PrepInsta Destructed\n";
•       }
•   };
•
•   int main()
•   {
•       if(1) {
•           // scope of obj starts here (Constructor called)
•           PrepInsta obj;
•
•       }
•       // scope of obj ends here (Destructor Called)
•       // since the object was not static
•
•       cout << "End of main function\n";
•   }
```

Example 2

Constructors/Destructors functioning with static objects

- When the object is static, the object is destroyed only when the program ends
- In the example below destructor is called just before program terminates

```
•   #include <iostream>
•   using namespace std;
•
•   class PrepInsta
•   {
•       public:
•       // constructor
•       PrepInsta() {
•           cout << "PrepInsta Constructed\n";
•       }
•
•       // destructor
•       ~PrepInsta() {
•           cout << "PrepInsta Destructed\n";
•       }
•   };
•
•   int main()
•   {
•       if(1) {
// scope of obj starts here (Constructor called)
static PrepInsta obj;

}

cout << "End of main function\n";

// scope of obj ends here (Destructor Called)
// just before the program terminates
// since the object was static
}
```

Facts about static members / methods

- Static data members can be accessed by other methods too
- Overloading static methods is possible
- Non-static data members can not be accessed by static methods.
- Static methods can only access static members (data and methods)

Const keyword in C++

Constant Variables

A variable declared as constant, will **not allow any modifications after its initialization.** Constant variables are created using the keyword const

For example –

```
#include <iostream>
using namespace std;
int main()
{
  int a1=10; //normal variable
  const int a2=20; //const variable
  //works fine as normal variable
  a1++;
  //this would give error as
  //we are trying to change value in const variable
  a2++;
  return 0;
}
```

Constant Member Function of Class

- A constmember function ***cannot change any data members of the class*** and it also cannot call any non-const function
- It is a ***read-only function***
- To make any member function const, we add the *const* keyword after the list of the parameters after the function name.

```
class test
{
  public: int x;
  void func() const
  {
     x = 0; // this will give compilation error
   }
};
```

- The above code will give us compilation error because 'func()' is a const member function of class test and we are trying to assign a value to its data member 'x'.
- A const object can only call a const member function, this is because a const object cannot change the value of the data members and a const member function also cannot change the value of the data member of its class. So, a const object would like to call a function which does not violate its rule.
- We cannot make constructors const Generally, const objects initialize the values of their data members through constructors and if we make the constructor const, then the constructor would not change the values of the data members and the object would remain uninitialized.

Constant Pointer

- If we make a pointer const, Its address cannot be changed after initialization
- This means that the pointer will always point to the same address.
- Here address of const pointer cant be changed but the value that is pointed can be changed

Example:

```
int main()
{
  int a = 4,b=5;
  int* const ptr = &a; // const pointer p pointing to the variable a
  //ptr=&b;//invalid :const pointer address cannot be changed
  *ptr=6;//valid,value at ptrs address is non_const
}
```

Note that we cannot change the pointer p but can change the value of a.

Pointer to Constant Variable
Here, the ***value that is pointed by the pointer cannot be modified***

Example:

```
int main()
{
  int a = 4,b=5;
  const int *p=&a;//a becomes constable variable
  //*p=5;//inavlid:a is const
  p=&b;//valid: address pointed by p is non_const
}
```

Here, p is a pointer which is pointing to a const int variable, this means that we cannot change the value of that int variable.

Constant objects

- We cannot modify the data members of a const object, ***only one initialization is allowed for all the copies of that object***
- const objects are ***reads only objects***

We make an object const by writing the const keyword at the beginning of the object declaration as shown below.

```
const test t(5);
```

- We made the object a of class test const by writing the const keyword before the class name at the time of its declaration.
- A const class object is initialized through the constructor.

Example:

```
class test
{
  public: int x;
  test()
  {
    x = 0;
  }
};
main()
{
  const test t; // declaring const object 'a' of class 'A'
  // t.x = 10; // compilation error
}
```

- The above program will give a compilation error.
- Since we declared ‘t’ as a const object of the class test, we cannot change its data members.
- When we created ‘t’, its constructor got called assigning a value 0 to its data member ‘x’. Since ‘x’ is a data member of a const object, we cannot change its value further in the program. So when we tried to change its value, we got a compilation error.

Constant Data Members

- ***Once initialized, a const data member may never be modified***, not even in the constructor or destructor.
- The const data member must be initialized by the constructor using an initialization list.

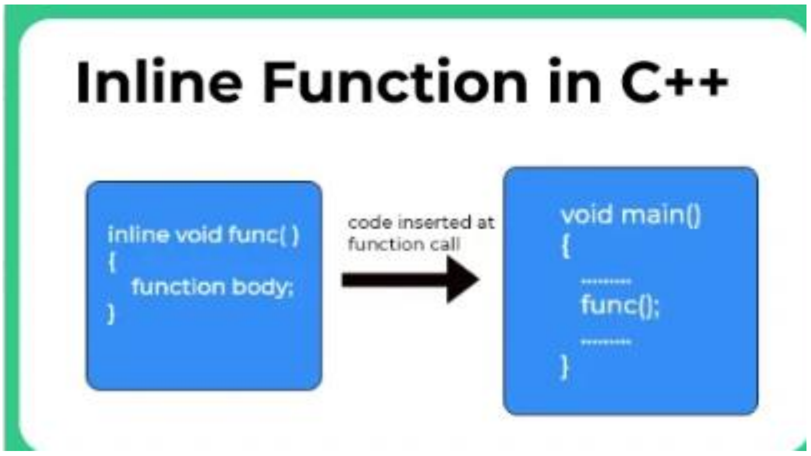
Example:

```
class test
{
  const int x;//const_data member
  public:
  test(int y)
  {
    x = y;
  }
  test(float z)
  {
    //x = z;//const memeber x cannot be reassigned
   }
};
main()
{
  A a(5);
  //A b(2.3);//invalid :assignment of read_only_object
}
```

Inline Function in C++

Inline functions and Classes

The inline function may also be defined inside a class.In actuality, all of the class’s declared functions are implicitly inlined.All inline function limitations are thus also applicable here.Simply declare the function inside the class and define it outside the class using the inline keyword if you need to explicitly declare an inline function in the class.



Syntax for Inline Function:

For creating Inline Function, use the **'inline'** keyword.

```
inline returnType function(parameters)
{
  // code
}
```

Example 1:

```
#include<iostream>
using namespace std;
inline int Min(int a, int b) {
return (a < b)? a : b;
}
// Main function
int main() {
cout << "Min (30,15): " << Min(30,15) << endl;
cout << "Min (50,400): " << Min(50,400) << endl;
cout << "Min (500,2000): " << Min(500,2000) << endl;
return 0;
}
```

Advantages of Inline Function:

- The compiler does not comply with the programmer's request to make a function with a return statement that is marked as inline and without returning any value an inline function.
- If a function requests to be inlined but contains switch or go-to statements, the compiler rejects the request.
- It is not possible to convert a function with static variables into an inline function.

Disadvantages of Inline Function:

- You must recompile the application after making any changes to the inline function code to verify that it is current.
- The size of the binary executable program has grown as a result of code expansion.
- Due to the larger executable, an increase in page faults results in subpar application performance.

Are Macros and Inline functions both the same?

- No, usage of macros is completely discouraged in C++ and unnecessary .they cannot access the private data of a class.so, inlining is recommended than macros. inline functions are capable of type checking and perform casting if required for arguments whereas macros just replace the code
- Inlining is under the control of compiler whereas macros are under the control of preprocessor

Can Virtual functions be Inlined?

- No, virtual function calls are resolved during the execution i.e it gets clarity about the function to be executed in such case how the function can be inlined as it doesn’ t to know what code to be replaced at compile time
- Inline functions are complied timed and virtual functions are run timed

Are Inline Functions really helpful despite many complexities?

- Yes, even though failure rate of an inline function is high, they can be really powerful in case of taking certain measures to enjoy the fruitful result such as, keeping the number of inline functions in a program to the minimum and avoid inlining of large functions.

Friend Function in C++

Friend Function in C++

Basic Properties of Friend Function in C++

- A function can be declared as a friend to any number of classes.
- A friend function tough not a member function has full rights and access to the members of the class

Creating a Friend Function

```
#include <iostream>
using namespace std;
class demo
{
  private:
  int a=5;
  friend void display(demo);//friend function declaration
};
void display(demo d)//friend function defination
{
  cout<<"a: "<<d.a<<endl;//access of private data
}
int main()
{
  demo d;
  display(d);//calling a friend function
return 0;
}
```

In the above code display() function is made friend with demo class so that display() function can access the private data of the class demo.

Friend class

- A class declared with a friend with another class is able to access private/protected data members of that class
- When a class is made a friend class, all the member functions of that class becomes friend functions to that class

Creating friend class

- In this program, all member functions of class test2 will be friend functions of class test1.
- Thus, any member function of class test2 can access the private and protected data of class test1.
- But, member functions of class test1 cannot access the data of class test2.

```
• #include <iostream>
• using namespace std;
• class test1
• {
```

```
•     int a,b;
•     public:
•     void get()
•     {
•      cout<<"enter 2 integers";
       cin>>a>>b;
•     }
•     friend class test2;//defining friend class
•   };
•   class test2
•   {
•      test1 t1;
•      public:
•      void put()
•     {
•         t1.get();
•         cout<<"a: "<<t1.a<<endl;
•          cout<<"b: "<<t1.b<<endl;
•      }
•   };
•   int main()
•   {
•       test2 t2;
•       t2.put();
•   }
```

Advantages

Code reusability: instead of creating separate member functions to access private data of each class ,a single member function can access private data of many classes

Disadvantages

Security: friend functions should be used only when absolutely necessary because friend mechanism alters the security of private data which is against data hiding.

Real-time applications of Friend

- Device integration is possible only by friend

  ex: GPRS, Satellite communication

- Friend class is also used in APIs(Application Programming Interface)

  ex: money transfer from one bank to another

Note: Java and Unix System never support Friend Mechanism

Virtual Function in C++

Virtual Function in C++

Virtual functions give programmer capability to call a member function of a different class by the same function call based on a different situation.

- In other words, a virtual function is a function which gets override in a derived class and instructs c++ compiler for executing late binding on that function
- A function call is resolved at runtime in late binding and so that compiler determines the type of an object at runtime.

Virtual function syntax

```
class class_name
{
  access:
  virtual return fucntion_name(arg...)
  {
     ---;
  }
};
```

Advantage of virtual functions

Ambiguity in function calls is reduced because It can call all the required functions from parent and child class in a different context

Some points to note about virtual functions

- Members of some class virtual functions cannot be static
- The virtual function can be defined in the parent class even if it is not used
- The declaration of the virtual function of parent class in the child class must be identical, if the prototype is not identical c++ will consider them as overloaded functions
- A virtual constructor is not possible whereas a virtual destructor is possible

C++ program to demonstrate the usage of virtual keyword

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

class parent//parent class
{
  public:
  virtual void show()
  {
    cout << "Base class\n";
  }
};

class child:public parent//child class
{
  public:  void show()
   {
     cout << "Derived Class";
   }
};

int main()
{
  parent* p; //Base class pointer
  child c; //Derived class object
  p = &c;
  c.show(); //Late Binding Occurs
}
```

When we have used virtual keyword with the base class function dynamic binding takes place and the child version of the function will be called because parent class points to Child class object The output of the above program without using the virtual keyboard.

- Here we have declared parent class and assigned child class object but still, base class pointer reference will point the base class function show()
- To overcome this problem we will use the virtual keyword.

Function Overriding in C++

What is Function Overriding?

- Using inheritance a child class can inherit features (data members/member functions) from a parent class. Imagine if, the same function is defined in both the parent class and the child class.
- Now if we call this function using the object of the child class, the function of the child class is executed.
- This is known as **function overriding** in C++. The function in child class overrides the inherited function in the Parent class
- **There are two methods to do function overriding –**

1. Generic function overriding
2. Virtual function based overriding

Advantages of function overriding in C++

As we have discussed above what is function overriding, now let's look at some of its advantages –

- The child class having the same function as the parent class, can even have its independent implementation of that function.
- Helps in saving memory space.
- Maintain consistency, and readability of our code.
- Helps in making code re-usability easy.
- The child class can access the function of the parent class as well.

Disadvantages of function Overriding in C++

After learning things about function overriding lets have a look on its disadvantage:-

- Function overriding cannot be done within a same class. So, we need to do implement inheritance.
- Static methods can never be overridden.

Code to show Function Overriding (Method 1)

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

class Parent {
   public:
    void display() {
        cout << "Parent Function" << endl;
    }
};

class Child : public Parent {
   public:
    void display() {
        cout << "Child Function" << endl;
    }
};
```

```
int main() {
    Child childObj;
    childObj.display();

    return 0;
}
```

Using Virtual Function to Achieve Function Overriding (Method 2)

For a function display() which is defined in both parent & child classes. If we create a pointer of Parent type and point it to an object of the child class. It will –

- Always call the parent class function
- Will never override parent class function

As demonstrated in the example given below –

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

class Parent
{
public:
  void print ()
  {
    cout<<"Parent Class"<<endl;
  }
};


class Child:public Parent
{
public:

  void print ()
  {
    cout<<"Child Class"<<endl;
}};
int main ()
{
  Child c2;
  Parent &p2 = c2;
  p2.print ();
  return 0;
}
```

ReasonThe reason is because of how C++ was written by its founder Bjarne Stroustrup in 1979. What exactly happens here is called static resolution. How C++ is written forces by default makes a static call at the time compilation itself and forces early binding i.e. binding to parent class function itself.

How to solve this?

We use a virtual function in the parent class in order to ensure that the parent function is **overridden.** It is done by adding virtual keyword before function declaration in the parent class.

```
#include <iostream>
using namespace std;
class Parent
{
public:

  virtual void print ()
  {
    cout << "Parent Class" << endl;
  }
};

class Child:public Parent
{
public:

  void print ()
  {
    cout << "Child Class" << endl;
  }
};
int main ()
{
  Child c2;
  Parent &p2 = c2;

  p2.print ();

  return 0;
}
```

Function overloading in C++

Function Overloading in C++ Language

- A function is a set of statements that allow us to structure program in segments of code to perform individual tasks.

- For example a function created as ”int addition(int a, int b)” can be used anywhere in our program where we need to add two integer type numbers.
- Function overloading is the process in which we use the same name for two or more functions defined in the class.
- The only difference between these functions is that they have different types of parameters or a different number of parameters.
- We have to use different type or different number of arguments because it is the only way by which compiler can differentiate between two or more overloaded function.
- It is done at compile time hence it is also known as compile time polymorphism.

Advantages of function overloading in C++

As we have discussed above what is function overloading, now lets look at some of its advantage :-

- Using the function overloading concept, we can make more than one function with the same name
- Function overloading helps us to save the memory space, consistency, and readability of our code.
- Function overloading speeds up the execution of our code.
- Function overloading helps the application to load the class method based on the type of parameter.
- Function overloading makes code re-usability easy, thus it also helps to save memory.
- Function overloading makes code maintenance easy.

C++ programming code to show function overloading

```
#include <iostream>
using namespace std;
int area(int s)
{
   cout<<"Area of square="<<s*s <<endl;

   return 0;
}
```

```
int area(int l, int b)//Second definition.
{
   cout<<"Area of reactangle=" <<l*b<<endl;
   return 0;
}


int main()
{
   area(12);//Passing value for first defination.

   area(20,10);//Passing value for second defination.

}
```

In the above program, we have defined area() function two times. First function to calculate area of square using one integers value as parameters , second function to calculate area of rectangle using two integer value as parameters. In this way we have overloaded area() function by using different number of parameters.

Disadvantages of function Overloading in C++

After learning things about function overloading lets look at some of its disadvantage:-

- Function overloading process cannot be applied to the function whose declarations differ only by its return type

```
void area()
int area(); // error
```

- Member function declared with the same name and the same parameter types cannot be overloaded if any one of them is a static member function declaration.

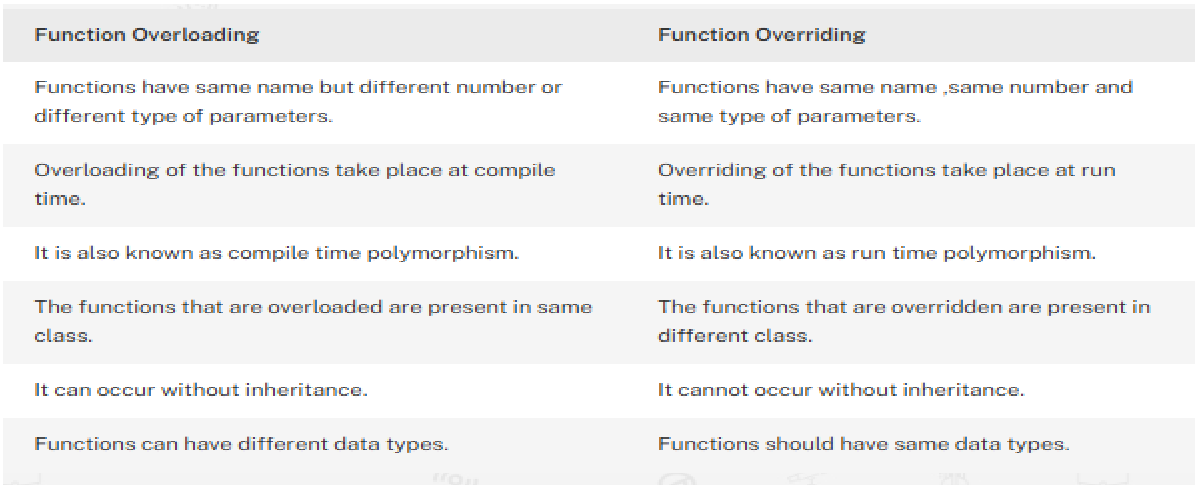
```
static void area();
void area(); // error
```

Function Overloading and Function Overriding Difference in C ++

Differences between Function Overloading and Overriding in C++

- **Function overloading** is a concept using which we define two or more functions in the same class with the same name with a condition that the parameters of the function should differ by its number or type.

- **Function overriding** is a concept using which we define two functions with the same name and same parameters with a condition that one function must present in a base class and other function in a derived class.



Function Overloading	Function Overriding
Functions have same name but different number or different type of parameters.	Functions have same name ,same number and same type of parameters.
Overloading of the functions take place at compile time.	Overriding of the functions take place at run time.
It is also known as compile time polymorphism.	It is also known as run time polymorphism.
The functions that are overloaded are present in same class.	The functions that are overridden are present in different class.
It can occur without inheritance.	It cannot occur without inheritance.
Functions can have different data types.	Functions should have same data types.

Recursion in C++

What is Recursion in C++?

Recursion is a method of solving a problem where the solution depends on solutions to smaller instances of the same problem. A recursive function is a function that calls itself with a smaller input until it reaches a base case, at which point it returns a result.

Example of a recursive function in C++ that calculates the factorial of a given number:

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

int factorial(int n)
{
  if (n == 0)
    return 1;
  else
    return n * factorial(n - 1);
}

int main()
{
  int num;
  cout << "Enter a number: "; cin >> num;
  int result = factorial(num);
  cout << "The factorial of " << num << " is " << result << endl;

  return 0;
}
```

Types of Recursion

There are two main types of recursion in C++:

1. Tail recursion
2. Non-tail recursion

Tail Recursion

- This is a type of recursion where the recursive call is the last thing the function does before returning a value.
- Tail-recursive functions are generally preferred over non-tail-recursive functions because they can be optimized by the compiler.

```
int factorial(int n) {
  if (n == 0) {
    return 1;
  }
  return n * factorial(n - 1);
}
```

Non Tail Recursion

- This is a type of recursion where the recursive call is not the last thing the function does before returning a value.
- Non-tail-recursive functions are generally less efficient than tail-recursive functions because they cannot be optimized by the compiler.

```
int factorial(int n) {
  if (n == 0) {
    return 1;
  }
  int result = factorial(n - 1);
  return n * result;
}
```

Return Reference in C++

Return by Reference in C++

In C++, a reference is a way to refer to an object indirectly. A reference is like an alias, or a nickname, for an existing object. To return a reference, the function definition must include an ampersand (&) after the type of the returned value.

Returning a reference can be useful when you want to avoid making a copy of a large object, or when you want to modify an object that was passed to a function by reference. It also allows you to chain function calls, since the return value of the function can be used as the input to another function.

Syntax

```
data_type func_name(data_type &var_name){
   // code
   return var_name;
}
```

Here,

- data_type is the return type of the function.
- func_name is the name of the function.
- var_name is the parameter which is passed as argument to it.

Algorithm

Step 1: Declare the function to return a reference by including an & symbol in the function's return type.
Step 2: Inside the function, return a reference to the desired object or variable by using the & symbol before its name.
Step 3: When calling the function, you can use the return value as a reference, which allows you to modify the original object or variable.

Note: Keep in mind that returning a reference can make your code more difficult to understand and debug, so you should use this feature with caution.

Examples

Returning a reference to a global variable:

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

int &max(int &x, int &y) {
  if (x > y) {
    return x;
  }
  return y;
}

int main() {
  int a = 10;
  int b = 20;
  int &c = max(a, b);  // c is a reference to the larger of a and b

  cout << "c: " << c << endl;

  c = 30;  // This sets the larger of a and b to 30
  cout << "a: " << a << endl;
  cout << "b: " << b << endl;

  return 0;
}
```

Returning a reference to an element of an array:

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

int &getElement(int *arr, int index) {
  return arr[index];
}

int main() {
  int arr[5] = {1, 2, 3, 4, 5};
  getElement(arr, 3) = 10;
  cout << arr[3] << endl;
  return 0;
}
```

Note: When returning a reference, it is important to ensure that the reference remains valid. For example, if the reference is to a local variable, the variable will no longer be in scope once the function returns, and the reference will be invalid.

Function floor() in C++

floor() Function in C++

In C++ programming language the floor function is included in cmath header file.

The floor function can handle values of any finite or infinite magnitude within the range of the double type in C++. The datatype of parameter can be double/ float/ long double only.

Note: If you try to pass a value outside of the range of the double type to floor, the behavior is undefined and may lead to unexpected results or a runtime error.

Declaration of floor function

```
double floor(double x)
```

Parameters of floor function

The round function takes a single parameter x.

Parameter	Description
x	It is the number for which you want to calculate the floor.

Return value of floor function

The floor function in C++ returns the greatest integer that is less than or equal to a given number. The return value is of type double, even if the input is an integer.

Implementation of floor function in C++

Example 1:

The following code shows the use of floor function.

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

int main() {

  double result;

  // Initializing parameter x
  int x = 20;
  result = floor(x);

  cout << "Floor of " << x << " = " << result;

  return 0;
}
```

Note: The floor function only works with real numbers. If you try to pass an integer or other type of value to floor, you may need to cast it to a double first.

Function ceil() in C++

ceil() Function in C++

In C++ programming language the ceil function is included in cmath header file.

It is used to round a number up to the next integer and the range of the ceil function is the set of all real numbers. The datatype of parameter can be double/ float/ long double only.

Declaration of ceil function

```
double ceil(double x)
```

Parameters of ceil function

The ceil function takes a single argument of type double.

Parameter	Description
x	It is the number that you want to round up to the next integer.

Return value of ceil function

The ceil function returns a value of type double, which is the smallest integer that is greater than or equal to the input number.

mplementation of ceil function in C++

Example 1:

The following code shows the use of ceil function.

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

int main() {

  // Initializing parameter
  double x = 4.6;
  double y = ceil(x);
  cout << "The smallest integer greater than or equal to " << x << " is " << y << endl;

  return 0;
}
```

Note: The ceil function rounds up to the next integer, so if the input number is already an integer, the function will return the same value.

User defined functions in C++

User defined functions in C++

In C++, a user-defined function is a block of code that performs a specific task and can be called multiple times in a program.

Syntax:

```
return_type function_name(parameter list) {
   // function body
}
```

- The return_type specifies the data type of the value that the function returns. If the function does not return a value, the return_type should be void.
- The function_name is the name by which the function can be called.
- The parameter list specifies the input arguments that are passed to the function.
- The function body contains the code that is executed when the function is called.

User-defined functions Types

In C++, there are 4 different types of user-defined functions which are:

1) Function with no parameter and no return value

A user-defined function in C++ that has no parameters and no return value, also known as a void function, is a block of code that performs a specific task but does not take any input and does not return any output.

Example 1:

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

void print_message() {
  cout << "PrepInsta Prime" << endl;
}

int main() {
  print_message(); // Calling the function
  return 0;
}
```

2) Function with no parameter but return value

A user-defined function in C++ that has no parameters but has a return value, is a block of code that performs a specific task and returns a value of a specific data type but doesn’t take any input.

Example 2:

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

int get_random_number() {
  return rand() % 10 + 1;
}

int main() {
  cout << "Random number: " << get_random_number() << endl;
  return 0;
}
```

3) Function with parameter but no return value

A user-defined function in C++ that has parameters and no return value, also known as a void function, is a block of code that performs a specific task and takes input in the form of function parameters, but does not return any output.

Example 3:

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

void print_string(string input) {
   cout << "Input: " << input << endl;
}

int main() {
   string message = "PrepInsta Prime";
   print_string(message); // Calling the function
   return 0;
}
```

4) Function with parameter and return value

A user-defined function in C++ that has parameters and a return value is a block of code that performs a specific task, takes input in the form of function parameters, and returns a value of a specific data type.

Example 4:

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

int multiply(int x, int y) {
   return x * y;
}

int main() {
   int a = 2, b = 3;
   int result = multiply(a, b);
   cout << "Result: " << result << endl;
   return 0;
}
```

ote: Even if the function doesn't take any input or return any value, it can still use variables or constants defined outside the function or passed to it via other means (e.g. global variables, pointers etc.).

Structures in C++ Language

Structure in C++ Language

Once you create a structure, then it becomes a data type. Now you can create any variables of this data type. And you can also create an array of this data type. But when you initialize the value of this variable, you have to initialize the value of all the variables defined in that structure. We can define the structure before the main method.

Defining the Structure

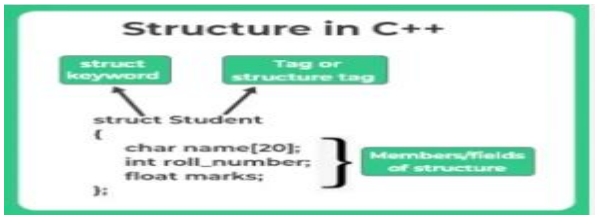
Struct keyword is used to define the structure. After this keyword the structure is given the unique name. After this, variables are created in curly braces and semicolon is applied after the ending curly bracket.

```
struct struct_Name
{
   Structure member 1;
   Structure member 2;
    ...
  Structure member N;
};
```

Let’s take an example :

```
struct Student
{
   int stu_id;
   char stu_name[10];
   int stu_age;
   char stu_branch;
};
```

Here, the name of the structure is taken in the name “Student” and there are four members of the structure,which have two data types. An integer for a ‘stu_id’ and ‘stu_age’ and character for a ‘stu_name’ and ‘stu_branch’.



```
struct Student
{
    char name[20];
    int roll_number;
    float marks;
};
```

Structure Variable Declaration

Structure variables can be declared in two ways:-

1. Structure variable is written even when the definition of structure is written.
2. The variable of the structure is also done inside the main() function.

Syntax for Structure variable outside of structure definition

```
struct structure_name
{
   data_type member 1;
   data_type member 2;
      ......
   data_type member n;
}structure_variable(s);
```

Example:-

```
struct Student
{
   int stu_id;
   int stu_age;
   char stu_name[10];
   char stu_branch[10];
}info;
```

Syntax for structure variable in main() function.

```
struct structure_name
{
   data_type member 1;
   data_type member 2;
            ......
   data_type member n;
}structure_variable(s);
int main()
{
struct structure_name structure_variable_name;
}
```

Example:-

```
struct Student
{
   int stu_id;
   int stu_age;
   char stu_name[10];
   char stu_branch[10];
};
int main()
{
struct Student info;
}
```

Accessing Structure Members

It can be assigned to values in many ways. Without the structure the members of the structure have no personal meaning. To provide a value to a member of any structure, the member name must be connected with the structure variables using dot(.). The operator is also called the term or member access operator.

Syntax for Accessing Structure members:-

```
structure_variable_name.member_of _structure =value(optional);
```

Example:-

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

struct Student
{
  char stu_name[10];
  int stu_age;
  char stu_branch;
  int stu_id;
};
int main ()
{
  struct Student s1;              // s1 is a variable of Student type and
  s1.stu_age = 17;                //age is a member of student

  // using string function to add name
  strcpy (s1.stu_name, "Soumya");

  cout << "Name of student 1:" << s1.stu_name << endl;
  cout << "Age of student 1:" << s1.stu_age << endl;

  return 0;
}
```

Structure Initialization

The structure initialization is of two type.

We can also initialize the structure variable at compile time.

TYPE 1:-

```
struct Student
{
  float height;
  int weight;
  int age;
};
struct Student s1 = { 180.75 , 73, 23 };        //initialization
```

TYPE 2:-

```
struct Student s1;
s1.height = 182.5 ;         //initialization of each member separately
s1.weight = 65;
```

```
s1.age = 20;
```

Array of Structure

We can create an array of structure like other primitive data types. In which each element of array will represent a structure variable.

Example :

```
 struct student stu[5];
```

The below program defines an array **stu of size 5.**

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

struct Student
{
  char name[10];
  int id;
};
struct Student stu[5];
int i, j;
void ask ()
{
  for (i = 0; i < 3; i++)
    {
      cout << "enter student record: " << i + 1;
      cout << "Student name: "; cin >> stu[i].name;
      cout << endl;
      cout << "enter id: "; cin >> stu[i].id;
    }
  cout << "displaying Student record:" << endl;;
  for (i = 0; i < 3; i++)
    {
      cout << "Student name is " << stu[i].name << endl;
      cout << "id is " << stu[i].id << endl;
    }
}

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

Structure and Function in C++

Structure and Function in C++

In C++, you can define functions inside a structure as member functions. These functions have access to the member variables of the structure and can be used to manipulate them.

Here's an example of a structure with member functions in C++:

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

struct Employee {
  string name;
  int age;
  float salary;

  // Define a member function that prints the employee's details
  void printDetails() {
    cout << "Name: " << name << endl;
    cout << "Age: " << age << endl;
    cout << "Salary: " << salary << endl;
  }
};

int main() {
  Employee emp = {"John Smith", 30, 45000.0};

  // Call the member function to print the employee's details
  emp.printDetails();

  return 0;
}
```

here are several ways to pass structure members to functions in C++. Here are two common approaches:

Pass the structure as an argument to the function:

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

struct Point {
    int x;
    int y;
};

void printPoint(struct Point p) {
    cout << "\nDisplaying coordinates of a point." << endl;
    cout << "(" << p.x << " , " ;
    cout  << p.y << ")";

}

int main() {

    struct Point p = {1, 2};
    printPoint(p);
    return 0;
}
```

Return struct from a function

To return a struct from a function in C++, you can simply use the return statement and specify the struct as the return value. Here's an example:

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

struct Point {
    int x;
    int y;
};

struct Point createPoint(int x, int y) {
    struct Point p = {x, y};
    return p;
}

int main() {

    struct Point p = createPoint(1, 2);
    cout << "\nDisplaying coordinates of a point." << endl;
    cout << "(" << p.x << " , " ;
    cout  << p.y << ")";

    return 0;
}
```

Note: It's important to note that when a struct is returned from a function, it is typically copied. This means that any changes made to the struct within the function will not be reflected in the original struct. If you want to modify a struct in a function and have those changes persist, you will need to pass a pointer to the struct to the function and modify the original struct through the pointer.

Pointers to Structures in C++

Pointer to Structures in C++

Here is an example of how you might declare a structure and a pointer to it in C++:

```
struct Student {
char name[30];
int age;
float grade;
};

struct Student *ptr;
```

To access the members of a structure using a pointer, you can use the "arrow" operator (->). For example:

```
ptr->name;   // accesses the "name" member of the structure pointed to by "ptr"
ptr->age;    // accesses the "age" member of the structure pointed to by "ptr"
ptr->grade;  // accesses the "grade" member of the structure pointed to by "ptr"
```

You can also use the "dot" operator (.) to access the members of a structure, but this requires that you have the actual structure variable rather than just a pointer to it. For example:

```
struct Student s;

s.name;   // accesses the "name" member of the structure "s"
s.age;    // accesses the "age" member of the structure "s"
s.grade;  // accesses the "grade" member of the structure "s"
```

Pointers to structures can be useful when you want to pass a structure to a function by reference (i.e., modify the structure within the function) or when you want to create an array of structures and access the elements using pointers.

<u>Pass a pointer to the structure:</u>

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

struct Employee {
  string name;
  int age;
  double salary;
};
```

```
int main() {

  // create a Employee structure variable
  Employee e;

  // accessing the members of Employee structure
  e.name = "John";
  e.age = 30;
  e.salary = 50000.0;

  // create a pointer to the Employee structure
  Employee *p_ptr = &e;

  // access the structure's members using the pointer
  cout << "Name: " << p_ptr->name << endl;
  cout << "Age: " << p_ptr->age << endl;
  cout << "Salary: " << p_ptr->salary << endl;

  cout<<endl;

  // assign values to the structure's members using the pointer
  p_ptr->name = "Sahil";
  p_ptr->age = 35;
  p_ptr->salary = 55000.0;

  // access the structure's members using the pointer
  cout << "Name: " << e.name << endl;
  cout << "Age: " << e.age << endl;
  cout << "Salary: " << e.salary << endl;

  return 0;
}
```

Note: Using pointers to access the members of a structure can be more efficient in some cases, as it allows you to access the members directly in memory rather than through a copy of the structure. However, it's important to be careful when using pointers, as they can be more difficult to work with than regular variables and can lead to runtime errors if not used correctly.

Enumeration in C++

Enumeration used in C++

Enum is almost similar to structure, as it can hold any data-type, which the programmer may want, this is advantageous when the program gets complicated or when more than one programmer would be working on it, for the sake of better understanding.

Declaration

Declaring a enum is almost similar to declare a structure.
We can declare a enum using the keyword "*enum*" followed by the enum name, and than stating its elements with the curly braces "{}", separated by commas.

```
enum color {
red,
green,
blue
};
```

- The first part of the above declaration, declares the data type and specifies its possible values. These values are called "enumerators".
- The second part declares variables of this data-types.

Working of Enumeration

The variables declared within the enum, are treated as integers, i.e. each variable of the enumerator can take its corresponding integral value, starting from zero(0). That means in our above discussed example,

- red = 0
- green = 1
- blue = 2

The main purpose of using enumerated variables is to ease the understanding of the program. For example, if we need to employee departments in a payroll program, it'll make the listing easier to read if we use values like Assembly, Production, Accounts, etc, rather than using 1,2 and 3.

Let's take one more example for understanding this:

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

int main()
  {
      enum emp_dept    //declaring enum emp_dept and its variables
         {
             assembly,
             manufacturing,
             accounts,
             stores
         };
      struct employee   //declaring srtucture
```

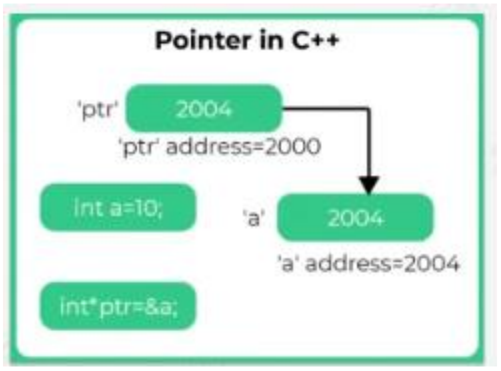
```
        {
          char name[30];
          int age;
          float bs;
          enum emp_dept department;
        };
      struct employee e;
    strcpy(e.name, "Nitin Kumar");   //intializing employee details
    e.age=22;
    e.bs=12500;
    e.department=stores;

    printf("\n Name = %s",e.name);
    printf("\n Age = %d",e.age);
    printf("\n Basic Salary = %f",e.bs);
    printf("\n Department = %d",e.department);

      if(e.department==accounts)
          printf("\n %s is an accountant",e.name);
      else
          printf("\n %s is not an accountant",e.name);

  return 0;
  }
```

Pointers in C++



A pointer is a variable that stores the address of another variable. Each variable is assigned with a unique memory location(16bit) by the compiler for access and identification. The values stored in this pointer can be the address of an ordinary variable, array variable or another pointer variable.

The address and pointers work by using two operators Addressof (&) and indirection or dereference (*) operator

Declaring and initializing pointers

```
data_type *var_name;
*var_name=<&any_var>;
```

Example:-

```
int k=5;
int *a;//declaring pointer
a=&k;//initialising pointer
```

C++ program to demonstrate pointers and address

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

  int a = 20;

  int *ptr;                              //pointer declartion
  ptr = &a;                                        //pointer initilisation
  cout << "value of a:" << a << endl;   // gives 20 i.e a's value
  cout << "address of a:" << &a << endl;           //a's address
  cout << "value of *ptr:" << *ptr << endl;        //pointing to address of 'a' i.e gives value at a's
adress
  cout << "value of *&a:" << *(&a) << endl;     //* ptr is interpreteed as *&a
  cout << "value of ptr:" << ptr << endl;           // ptr gives a's adress as it is initialised with a's
adress
  cout << "address of ptr:" << &ptr << endl;      //gives ptr adress
  return 0;

}
```

Advantages of pointer:

- Dynamic memory allocation
- Improves execution speed
- handle arrays, strings, structures in functions
- To manage data structures
- They play a key role in the design of Operating systems

Disadvantages:

They are not secured as they are directly working on address

Pointers to pointers:

- In order to store the ***address of the pointer variable itself***, we require another a pointer called double pointer
- To ***store the address of double pointer*** We store it in another pointer variable called triple pointer and so on, this can be extended any level

Creating Double and Triple Pointers

```
#include <iostream>
using namespace std;
int main()
{
  int a = 25;

  int *p = &a;                                    //normal pointer
  int **p2 = &p;                   //double pointer stores pointer address i.e p's addres
  int ***p3 = &p2;                 //triple pointer stores address of double pointer i.e p3's address
  cout << "a value is :" << a << endl;

  cout << "a value using p:" << *p << endl;

  cout << "a value using p2:" << **p2 << endl;

  cout << "a value using p3:" << ***p3 << endl;

return 0;
}
```

<u>How it works:</u>

- given a=25, assume a’s and p’s addresses are 2000 and 3000 respectively.
- *p=*(&a)=*(2000)=25
- **p2=*(*p2)=*(*(&p))=*(*(3000))=*((2000))=25

as p points a’s address, p2 contains the address of p, pointing to p2 gives P’s’address, again pointing to P’s address gives a’s’ address and finally pointing a’s address gives a’s value

Using Pointers in Arrays

By assigning the ***base cell address*** of an array to a pointer we can access the array elements using the following notation

*(baseadress+index*sizeof(array_type))

Example:

```
int a[5]={9,7,3,5,2}
int *ptr=&a[0]//assigning staring adress and incerementing
```

- Assume base address is 2000

  a[0]=*(2000+0*4)=*(2000+0)=*(2000)=9

  a[1]=*(2 array 000+1*4)=*(2000+4)=*(2004)=7

  Suppose, the pointer needs to point to the fourth element of an array, that is, hold the address of fourth array element in the above case.

- Since ptr points to the third element in the above example, ptr + 1 will point to the fourth element.

- You may think, ptr + 1 gives you the address of next byte to the ptr. But it’s not correct.This is because pointer ptr is a pointer to an int and size of it is fixed for an operating system (size of it is 4 byte of the 64-bit operating system).

- Hence, the address between ***ptr and ptr + 1 differs by 4 bytes***. If pointer ptr was a pointer to char then, the address between ptr and ptr + 1 would have differed by 1 byte since the size of a character is 1 byte.

C++ Program to display address of elements of an array using both array and pointers

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

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

int *ptr;

cout << "Displaying address using arrays: " << endl;

for (int i = 0; i < 5; ++i)

   {
   cout << "&arr[" << i << "] = " << &arr[i] << endl;
   }
// ptr = &arr[0]
  ptr = arr;                                          //assiing base address to pointer
  cout << "\nDisplaying address using pointers: " << endl;

for (int i = 0; i < 5; ++i)

    {
cout << "ptr + " << ptr + i << " = " << *(ptr + i) << endl; //access by incerementing address
    }
return 0;

}
```

Wild/Bad pointers

- A pointer which just declared but not initialized with any address is called wild pointer
- They point to some arbitrary memory location that and may be used by the system and cause a program to crash or behave badly.
- This problem can be solved using null pointers

Example

```
void main()
{
int *p;//declartion but no initialisation
cout<<"*p="<<*p<<"\n";//junk
cout<<"p="<<p<<"\n";//gives an arbitraryaddresss that is reserved already
}
```

Null Pointer

- It is always a good practice to assign the *pointer NULL to a pointer variable at the declaration case you do not have an address to be assigned.*
- A pointer that is ***assigned NULL*** is called a null pointer.
- The NULL pointer is a constant with a value of zero, defined in several standard libraries, including iostream.
- Many times, uninitialized variables hold some junk values and it becomes difficult to debug the program.

Example

```
int main ()
{
int *ptr = NULL;//inilialising with NULL
cout << "The value of ptr is " << ptr ;
return 0;
}
```

- On most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system.
- But by convention, if a pointer contains the null (zero) value, it is assumed to point to nothing.

Void Pointer

- Generally, Pointers are type specific i.e a pointer of integer type can hold the address of an integer variable only, It some times may cause problems if by mistake if we assign a pointer variable to a different datatype
- Hence a Void pointer is a ***Generic Pointer that can point to any datatype object***, provided it must be typecasted while accessing so as compiler should the required number of bytes for that value
- If a pointer's type is void *, the pointer can point to any variable that is not declared with the const or volatile keyword.

Example

```
int main()
{
int a=10;
float b=1.5;
void *p=&b;//pointing to float variable address
//int *c=&b//invalid types c must be float pointer
cout<<"b value:"<<*(float *)p;
}
```

Dangling pointer

- A pointer that is still ***pointing to a memory address that has been deleted*** is called dangling pointer
- To over this null pointers can be used

Example:

```
int main()
{
int *p=new int(100);//allocating memory
cout<<"*p="<<*p<<endl;
cout<<"p="<<p<<endl;
delete p;// adress in p is no more
cout<<"p value="<<*p<<endl;//garbage value
p=NULL;//point no where
cout<<"p value="<<*p<<endl;
}
```

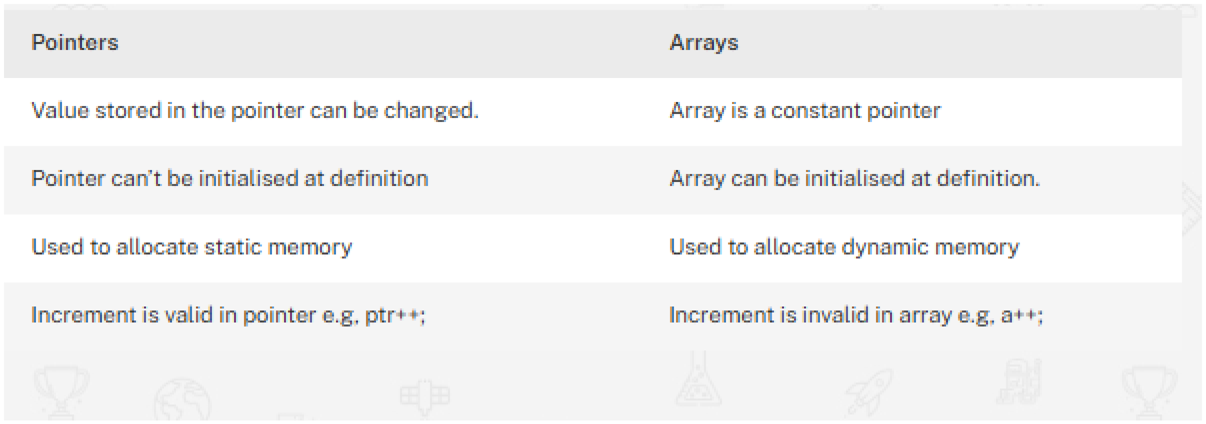
Pointer and Array In C++

A pointer is a type of variable that keeps as its value the memory address of another variable while an array is defined as the consecutive block of sequence which can store similar type of data.

Basic Definition

Array Definition : Array is defined as the continuous block of memory in which a similar data type is stored. Array indexing starts with size 0 and ends with size -1. The size of the array is fixed and it is stored in static memory.

Pointer Definition : Pointers in C++ language are is nothing but sort of like a variable which works like a locator/indicator to an address values (hence the name pointer).It helps in reducing the code and returning multiple values from a function.



Pointers	Arrays
Value stored in the pointer can be changed.	Array is a constant pointer
Pointer can't be initialised at definition	Array can be initialised at definition.
Used to allocate static memory	Used to allocate dynamic memory
Increment is valid in pointer e.g, ptr++;	Increment is invalid in array e.g, a++;

Null Pointer in C++

Null Pointers in C++ language are is nothing but sort of like a variable which works like a locator/indicator to an address values (hence the name pointer).It helps in reducing the code and returning multiple values from a function.

Syntax Of Pointers In C++:

```
datatype * var_name;
```

The * operator creates a pointer variable, which points to a data type (like an int) of the same type.The pointer is given the address of the variable you are working with.

Example :
In the below example, pointer is the address of the variable while *pointer and val print the value of the variable

```
#include <iostream>
using namespace std;
int main()
{
   int var1 = 3;
   int var2 = 24;
   int var3 = 17;
   cout << "Address of var1 = "<< &var1 << endl;
   cout << "Address of var2 = " << &var2 << endl;
   cout << "Address of var3 = " << &var3 << endl;
}
```

Uses of Null Pointers :

- To pass references for arguments
- In order to access array items
- Numerous values to be returned
- Allocating memory dynamically
- Data structures can be implemented easily.
- Very easy to access memory address by Pointers.

Null Pointer Arithmetic :
Arithmetic of Pointers is different from integer. The following arithmetic of pointer are :

- Increment/Decrement of Pointer.
- Addition/ subtraction of integer to a Pointer.

<u>Advantages of Pointers in C++ :</u>

- Arrays can be easily accessed by pointers.
- Memory locations can be easily accessed by pointers.

- Complex data structures can be easily built by the pointers.

Disadvantages of Pointers in C++ :

- Pointers are hard to understand.
- Pointers are slower than variable in C++.

Memory Management in C++

Memory Management : C++

When memory is allocated dynamically and not deallocated, it leads to memory leaks, which can cause your program to run out of memory and crash.

C++11 also introduce smart pointers, like std::unique_ptr, std::shared_ptr and std::weak_ptr which helps managing memory by handling the deletion automatically, reducing the risk of memory leaks and dangling pointers.

It is important to use delete and delete[] correctly to avoid undefined behavior and memory leaks, when you are allocating memory using new and new[].

Additionally, C++ also provides the new and delete operators with matching placement versions, new (p) T and delete (p), that allows to use a pre-allocated buffer to create objects, this is called placement new and placement delete.

C++ new Operator

In C++, the new operator is used to dynamically allocate memory for a variable or an array of variables at runtime. The operator returns a pointer to the start of the allocated memory, which can then be used to access the memory.

The basic syntax for using the new operator to allocate memory for a single variable is:

```
Type* ptr = new Type;
```

Here Type is the data type of the variable, and ptr is a pointer that will store the address of the allocated memory.

You can also use the new operator to allocate memory for an array of variables:

```
Type* ptr = new Type[size];
```

Here Type is the data type of the array elements, size is the number of elements in the array, and ptr is a pointer that will store the address of the allocated memory.

C++ delete Operator

In C++, the delete operator is used to deallocate memory that was previously allocated by the new operator. When you no longer need the memory, it should be deallocated to prevent memory leaks.

Once you're done using the memory allocated with the new operator, you should release the memory using the delete operator:

```
delete ptr;
```

Or when allocating an array

```
delete[] ptr;
```

It is important to match the delete operator with the allocation method, using delete with memory allocated with new[] and vice versa will cause undefined behavior.

C++ also provides the new operator with matching placement versions, new (p) T and delete (p) which allows to use preallocated buffer to create objects, this is called placement new and placement delete.

C++ new and delete Operator for Arrays

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

int main() {

int num;
cout << "Enter total number of aspirants: ";
cin >> num;
float* ptr;

// memory allocation of num number of floats
ptr = new float[num];

cout << "Enter marks of aspirants." << endl;
for (int i = 0; i < num; ++i) {
cout << "aspirant" << i + 1 << ": ";
cin >> *(ptr + i);
}

cout << "\nDisplaying marks of aspirants." << endl;
for (int i = 0; i < num; ++i) {
cout << "aspirant" << i + 1 << ": " << *(ptr + i) << endl;
}
// ptr memory is released
delete[] ptr;

return 0;
}
```

References in C++

Syntax

```
data_type &refernce_variabe=value_variable;
```

Example:

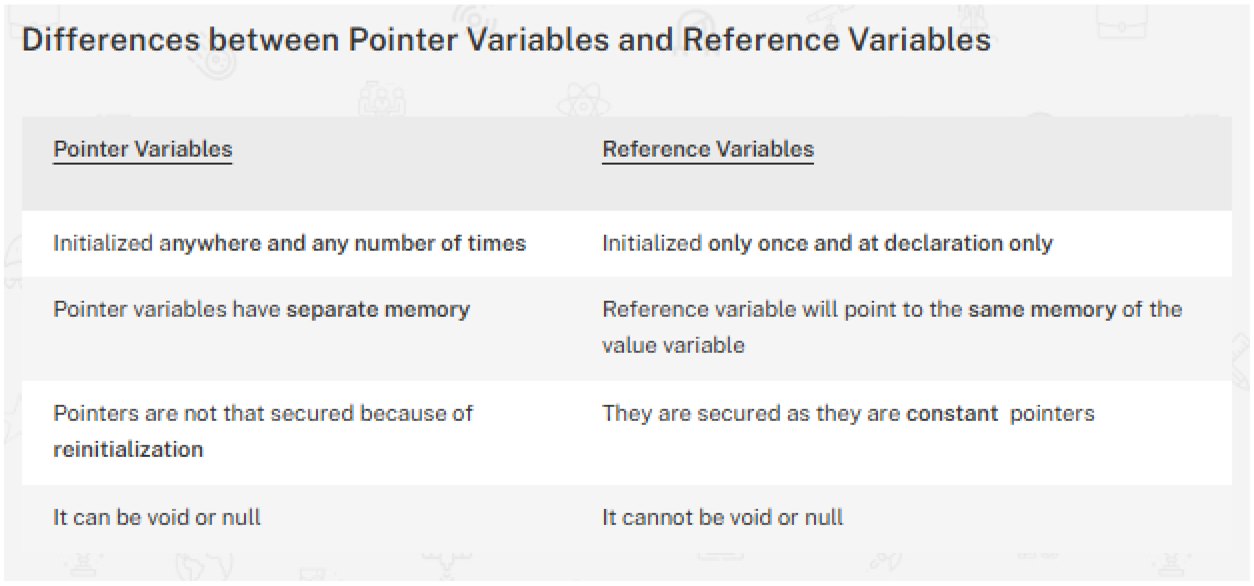
```
int a=5;
int &z=a;//now z and a have same address
```

- In the above example, variables ‘a’ and ‘z’ are ***pointing to the same memory address***
- So that ‘z’ becomes the alias name for ‘a’
- Any change in ‘a’ will also reflect change int ‘z’ and vice-versa

Program to demonstate reference variables

```
#include <iostream>
using namespace std;
int main()
{
  int a=10;
  int &b=a;//b is alias of a
  int &c=b;//c is alias of b
  cout<<"a:"<<a<<" b:"<<b<<" c:"<<c<<endl;
  c=20;//a,b also becomes 20
  cout<<"a:"<<a<<" b:"<<b<<" c:"<<c<<endl;
  cout<<"&a:"<<&a<<" &b:"<<&b<<" &c:"<<&c<<endl;//a,b,c has same address
  return 0;
}
```

- Initially ‘b’ is made as an alias of ‘a’ by giving the address of ‘a’ to b now ‘c’ is made an alias of ‘b’, as ‘b’ is pointing a’s address ‘c’ also takes the address of ‘a’.
- Hence ‘a’, ‘b’, ‘c’ are having the same address and become an alias to each other.
- Now, ‘b’ is changed to 20, as a result, ‘a’ and ‘b’ also becomes 20 because they are having a ***common memory space.***



Differences between Pointer Variables and Reference Variables

Pointer Variables	Reference Variables
Initialized **anywhere and any number of times**	Initialized **only once and at declaration only**
Pointer variables have **separate memory**	Reference variable will point to the **same memory** of the value variable
Pointers are not that secured because of **reinitialization**	They are secured as they are **constant** pointers
It can be void or null	It cannot be void or null

Advantage

When the reference variable is lost it can be accessed through another alias, but if the pointer is lost, the whole security comes into question.

Applications

- The internet architecture is developed by C++ under call- by- reference concept
- Many other applications like Chat dialogue box, we can observe gtalk box used in video conferencing, teleconferencing and live telecast etc.

Call by reference in C++

```
#include <iostream>
using namespace std;
void swap(int& x, int& y)                    //aliasing a and b
{
  int temp = x;

  x = y;
  y = temp;

  cout << "x:" << x << " y:" << y << endl;
}
int main()
{
  int a = 2, b = 3;

  swap (a, b);                          //func_call
  cout << "a:" << a << " b:" << b << endl;
return 0;

}
```

- Here instead of passing address through pointers ‘x’, ‘y’ are made aliases to ‘a’, ‘b’
- As ‘x’ and ‘a’, ‘y’ and ‘b’ both are made to have the same address, change in one affects others

Pointers vs References in C++

Pointers Vs References in C++

Pointers in C++

In C++, a pointer is a variable that stores the memory address of another variable. Pointers are used to store the addresses of memory locations used by programs, and can be used to access those locations.

Here is an example of declaring a pointer in C++:

```
int x = 5;
int *ptr;
ptr = &x;
```

In this example, ptr is a pointer to an int and it stores the memory address of x. The & operator is used to get the memory address of a variable.

To access the value stored at the memory location pointed to by a pointer, you can use the * operator, like this:

```
int y = *ptr;
```

In this example, y will be assigned the value of x, which is 5.

Fact about Pointers Pointers can be used to pass arguments to functions by reference, rather than by value, allowing the function to modify the original argument. Pointers can also be used to dynamically allocate memory at runtime

References in C++

In C++, a reference is a way to refer to an already existing variable, rather than creating a new one. It is similar to a pointer, but with a few key differences.

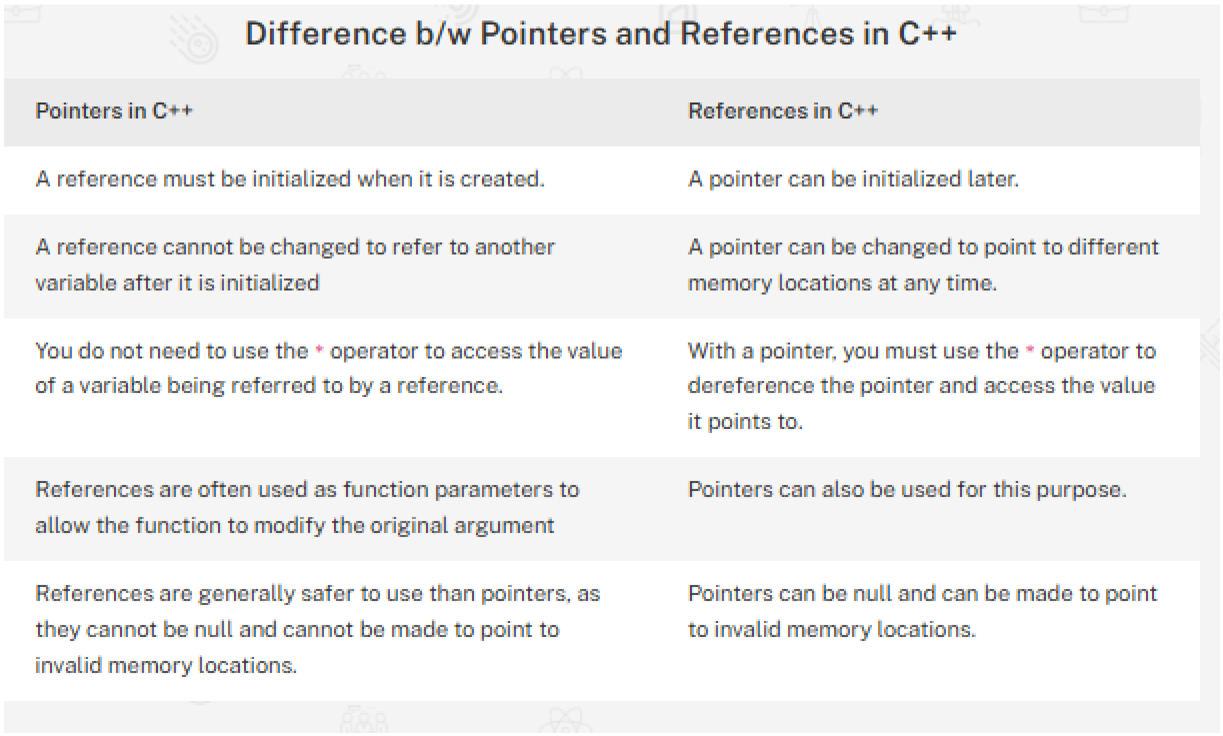
Here is an example of declaring a reference in C++:

```
int x = 5;
int &y = x;
```

In this example, y is a reference to x. The & operator is used to declare a reference, just as it is used to declare a pointer. However, unlike a pointer, you do not need to use the * operator to access the value of the variable being referred to. You can use the reference itself, like this:

```
cout << y; // Outputs 5
y = 10; // Changes the value of x to 10
```

Fact about References References are often used as function parameters to allow the function to modify the original argument.
They are also used as an alternative to pointers when you want to avoid the use of pointers, or when you want to make it clear that the variable being referred to will not be null.



Difference b/w Pointers and References in C++

Pointers in C++	References in C++
A reference must be initialized when it is created.	A pointer can be initialized later.
A reference cannot be changed to refer to another variable after it is initialized	A pointer can be changed to point to different memory locations at any time.
You do not need to use the * operator to access the value of a variable being referred to by a reference.	With a pointer, you must use the * operator to dereference the pointer and access the value it points to.
References are often used as function parameters to allow the function to modify the original argument	Pointers can also be used for this purpose.
References are generally safer to use than pointers, as they cannot be null and cannot be made to point to invalid memory locations.	Pointers can be null and can be made to point to invalid memory locations.

Use of Pointers in C++

Pointers are useful in C++ for a number of reasons. Here are some common scenarios where pointers are used:

1. To pass arguments to functions by reference, allowing the function to modify the original argument.
2. To dynamically allocate memory at runtime, using the new operator.
3. To store the address of an element in an array, allowing you to iterate over the array using pointer arithmetic.
4. To implement data structures such as linked lists and trees.

5. To access elements of an array or a string using array notation, even if the array is passed to a function as a pointer.
6. To pass large structures or arrays to functions more efficiently, by passing a pointer to the structure instead of the entire structure.

Use of References in C++

References are a useful feature of C++, and can be used in a number of situations. Here are some common scenarios where references are used:

1. As function parameters, to allow the function to modify the original argument. This is often more convenient and easier to read than using pointers for this purpose.
2. As a more convenient and readable alternative to pointers, especially when you want to avoid the use of pointers or when you want to make it clear that the variable being referred to will not be null.
3. To improve performance when passing large structures or arrays to functions, by passing a reference to the structure instead of the entire structure.
4. To create aliases for variables. For example, you might use a reference to give an alternate name to a complex or unwieldy variable, making it easier to work with in your code.

Passing by Pointer Vs Passing by Reference

Pointer Vs Reference in C++

Passing by reference means that if you change the value of the argument inside the function, the change will be reflected outside the function as well. Whereas passing by pointer means inside the function, you can dereference the pointer to change the value of the argument, but the change will not be reflected outside the function unless you pass the pointer back as a return value.

Passing by pointer vs passing by reference in C++

Here is an example to illustrate the difference:

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

void incrementByPointer(int* x) {
  (*x)++;
}

void incrementByReference(int& x) {
  x++;
}
```

```
int main() {
  int a = 5;
  int b = 5;

  incrementByPointer(&a);
  incrementByReference(b);

  std::cout << "a: " << a << std::endl;
  std::cout << "b: " << b << std::endl;

  return 0;
}
```

Object and Class in C++

Object and Class in C++

- In C++ programming language, Here is an example of a simple class definition in C++.
- To create an object of this class, you would use the following syntax:

```
Point p;
```

This class, called “Point”, has two data members (x and y) and two member functions (setX and setY).

```
class Point {
  public:
    int x;
    int y;
    void setX(int newX) { x = newX; }
    void setY(int newY) { y = newY; }
};
```

This class, called “Point”, has two data members (x and y) and two member functions (setX and setY). The public: keyword indicates that the data members and member functions following it are accessible from outside the class.

You can then access the object’s data members and call its member functions like this:

```
p.x = 10;
p.setY(20);
```

You can also define multiple objects of the same class:

```
Point p1, p2;
```

Each object has its own separate instance of the class’s data members and can have different values for those data members.

Member Functions in Classes in C++

```
// C++ program to demonstrate function

#include <iostream>
using namespace std;
class Rectangle {
   private:
      int length;
      int width;
   public:
      void setLength(int len) {
         length = len;
      }
      void setWidth(int wid) {
         width = wid;
      }
      int getArea() {
         return length * width;
      }
};
int main() {
   Rectangle rect;
   rect.setLength(5);
   rect.setWidth(3);
   cout << "The area of the rectangle is " << rect.getArea() << endl;
   return 0;
}
```

Note: Each object has its own separate instance of the class's data members and can have different values for those data members.

Constructors in C++

Constructors in C++ Language

Syntax

Defining Internally to the class

```
class Demo {
  char c;
  public:
     Demo(){
       // Constructor declaration here
       // User defined assignments for initialisation
  }
};
```

Defining Externally to the class

```
class Demo
{
 char c;
 public:
   Demo();
     //Constructor Declared here
};

Demo::Demo()
// Constructor definition here
{
    // initialisation of values by the user
    char = "PrepInsta";
}
```

Why Constructors are used?

- When an object is created, the memory is assigned the object entities and some garbage values are assigned to those.
- These garbage values may cause hinderance in the programs, as it may be needed to only assign very specific values to entities/data.
- Constructors do not have any return argument.
- They have the same name as the name of the class and are enclosed within the class
- C++ has its own version of constructor which assigns some garbage or random values to object entities.

Types of Constructors –

The following are the types of constructors in C++ –

- Default
- Parameterized

- Copy

Default Constructors

- **Highlights –** No Passing Arguments in the constructor.

For Example –

```
#include <iostream>
using namespace std;
class Demo
{
   public:
        int i;
        Demo(){
            i=8000;
          }
};

int main()
{
  Demo  demo;
  cout << demo.i;

return 0;
}
```

Parameterised Constructors

- **Highlights –** Arguments are passed with the constructor.

For example –

```
#include <iostream>
using namespace std;
class Demo
{
  public:
    int i;
    Demo(int val)
    {
       i=val;
    }
};
int main()
```

```
{
    //setting parameterised values
    Demo demo(6000);
    Demo demo2(12000);
    cout << demo.i<<endl;;
    cout << demo2.i;

return 0;
}
```

Copy Constructors

- **Highlights –** Arguments are passed as objects and object is copied as instantiated as a new object.
- The object can be passed in multiple ways
    - Demo d2 = d1;
    - Demo d3(d1);
- While the following doesn't call the copy constructor but only assigns the value with the help of assignment operator
    - Demo d4;
    - d4 = d1

For example –

```
#include <iostream>
using namespace std;
class Demo
{
   private:
    int a, b;
    public:
    Demo()
    { }
    Demo(int a1, int b1) {
     a = a1;
     b = b1;
    }
    // Copy constructor example here
    Demo(const Demo &d2)
    {
        a = d2.a;
        b = d2.b;
   }

    int getA()
    {
        return a;

    }
    int getB()
    {
        return b;

    }
};

int main()
{
    Demo d1(5000, 6000);

    // We have called a copy constructor
    Demo d2 = d1;

    // We have called a copy constructor
    Demo d3(d1);
    Demo d4;

    // Assignment operation happens there is no call to copy constructor
    d4 = d1;

    // displaying values for both constructors
    cout << "d1.a = " << d1.getA() << ", d1.b = " << d1.getB();
    cout << "\nd2.a = " << d2.getA() << ", d2.b = " << d2.getB();

    cout << "\nd3.a = " << d3.getA() << ", d3.b = " << d3.getB();
    cout << "\nd4.a = " << d4.getA() << ", d4.b = " << d4.getB();
    return 0;
}
```

Objects and functions in C++

Object and Function in C++

In C++, it is possible to define a class, its member functions, and create an object of that class all in one place. This is known as the “combined definition” style.

Here is an example of how Pass Object from a Function in C++:

```
#include <iostream>

using namespace std;

class MyClass {
  public:
    int x;
    void setX(int val) {
      x = val;
    }
    int getX() {
      return x;
    }
};

int main() {
  MyClass obj;
  obj.setX(5);
  cout << obj.getX() << endl;
  return 0;
}
```

This program defines a class MyClass with a data member x and two member functions setX() and getX(). It then creates an object of that class called obj and uses the member functions to set and get the value of x.

Objects in C++ are a key part of object-oriented programming (OOP). They allow you to encapsulate data and behavior into a single, self-contained entity.

Fact about Objects Objects makes it easier to design and work with complex programs, as you can think of each object as a separate, modular component of the overall program.

Return Object from a Function in C++

```
#include <iostream>

using namespace std;

class MyClass {
  public:
    int x;
    MyClass(int val) {
      x = val;
    }
    int getX() {
```

```
    return x;
  }
};

int main() {
MyClass obj(5);
cout << obj.getX() << endl;
return 0;
}
```

In this example, the class MyClass is defined with a data member x and a constructor function that initializes x to a specified value. The object obj is created using the constructor function and the value 5. The getX() member function is then used to retrieve the value of x and print it to the console.

This style can be useful for simple programs where you only need to create a single object of a given class. It can make the code more compact and easier to read. However, for larger and more complex programs, it is often more appropriate to use the separate function definition style, where the class definition and the object definitions are kept in separate parts of the code.