Address of variable

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

OUTPUT:

Address of var1: 0x7fff5fbff8ac
Address of var2: 0x7fff5fbff8a8
Address of var3: 0x7fff5fbff8a4

Assigning Addresses to Pointers

- Assign addresses to pointers:

```
int var = 5;

int* point_var = &var;
```

Here, 5 is assigned to the variable var. And the address of var is assigned to the point_var pointer with the code point_var = &var.

- To get the value pointed by a pointer, we use the * operator.
- For example: int var = 5;

// assign address of var to point_var

- int* point_var = &var;

// access value pointed by point_var

cout << *point_var << endl; **// Output: 5**

In the above code, the address of var is assigned to point_var. We have used the *point_var to get the value stored in that address.

When * is used with pointers, it's called the dereference operator. It operates on a pointer and gives the value pointed by the address stored in the pointer. That is, *point_var = var.

POINTER

Syntax:

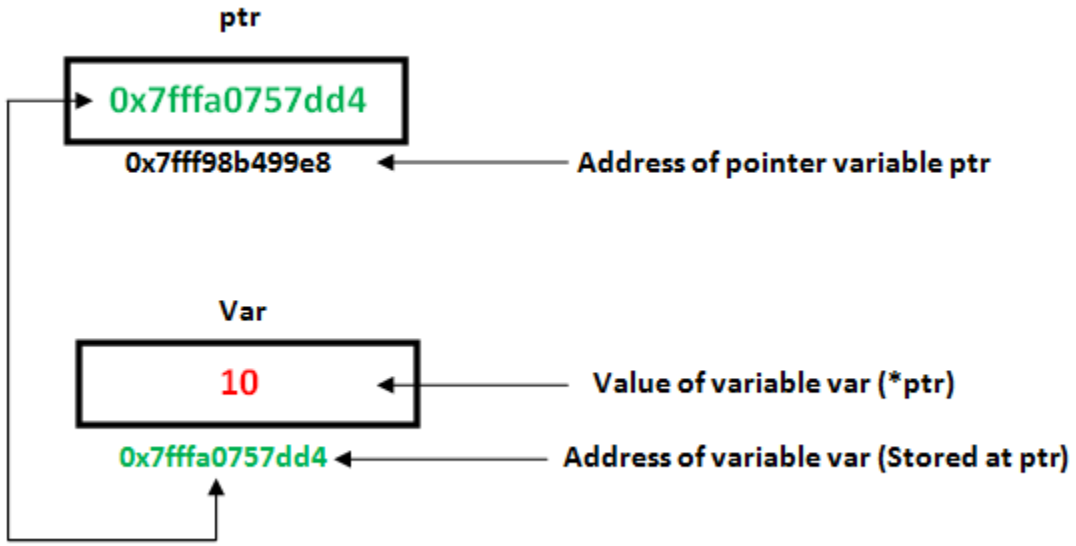
datatype *var_name;

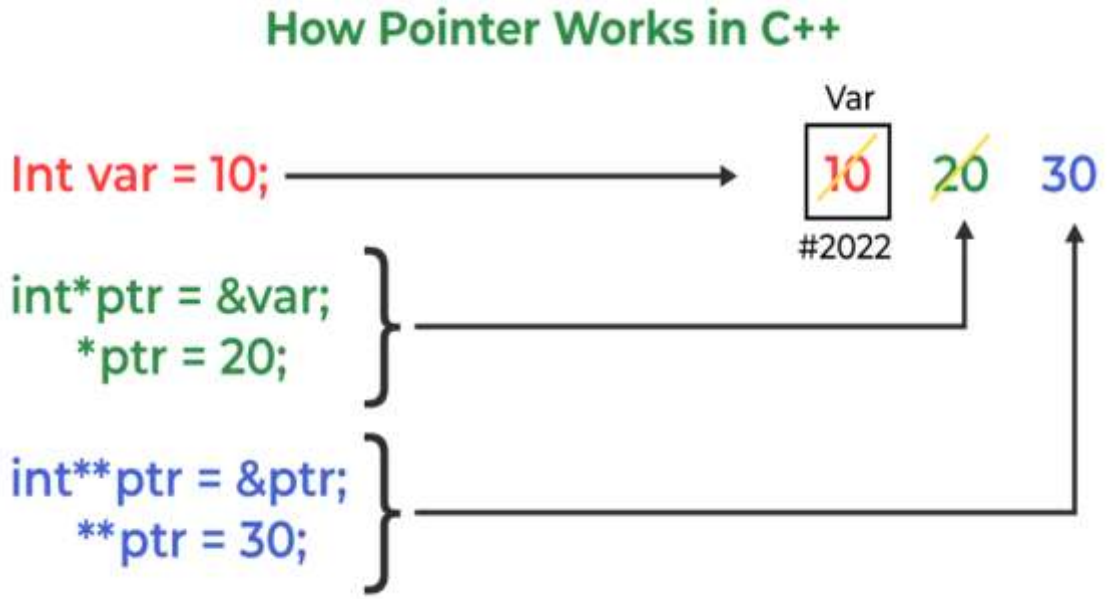
int *ptr; // ptr can point to an address which holds int data

How to use a pointer?

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

The reason we associate data type with a pointer is that it knows how many bytes the data is stored in. When we increment a pointer, we increase the pointer by the size of the data type to which it points.





// C++ program to illustrate Pointers

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

OUTPUT:

Value at ptr = 0x7ffe454c08cc
Value at var = 20
Value at *ptr = 20

Changing Value Pointed by Pointers

```
int var = 5;
int* point_var = *var;
// change value at address point_var
*point_var = 1;
cout << var << endl; // Output: 1
```

Changing Value Pointed by Pointers

```
#include <iostream>
using namespace std;
int main() {
   int var = 5;
// store address of var
   int* point_var = &var;
// print var
   cout << "var = " << var << endl;
 // print *point_var
   cout << "*point_var = " << *point_var << endl
        << endl;
cout << "Changing value of var to 7:" << endl;
 // change value of var to 7
   var = 7;
// print var
   cout << "var = " << var << endl;
// print *point_var
   cout << "*point_var = " << *point_var << endl
cout << "Changing value of *point_var to 16:" << endl;
   // change value of var to 16
   *point_var = 16;
 // print var
   cout << "var = " << var << endl;
  // print *point_var
   cout << "*point_var = " << *point_var << endl;
   return 0;
}
```

OUTPUT:

var = 5
***point_var = 5**
Changing value of var to 7:
var = 7
***point_var = 7**
Changing value of *point_var to 16:
var = 16
***point_var = 16**

Address-of operator (&)

- The Address-of operator (&) is a unary operator that returns the memory address of its operand which means it stores the address of the variable, which depicts that we are only storing the address not the numerical value of the operand. It is spelled as the address of the variable.

Syntax: gfg = &x; // the variable gfg stores the address of the variable x.

```
#include <iostream>
using namespace std;
int main()
{
        int x = 20;
        // Pointer pointing towards x
        int* ptr = &x;
        cout << "The address of the variable x is :- " << ptr;
        return 0;
}
```

References and Pointers

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

1. Call-By-Value
2. Call-By-Reference with a Pointer Argument

Pass by Reference with Pointers

```
// C++ program to implement pass-by-reference with  pointers
#include <iostream>
using namespace std;
void f(int *x)
{
*x = *x - 1;
}
int main()
{
int a = 5;
cout << a << endl;
f(&a);
cout << a << endl;
}
```

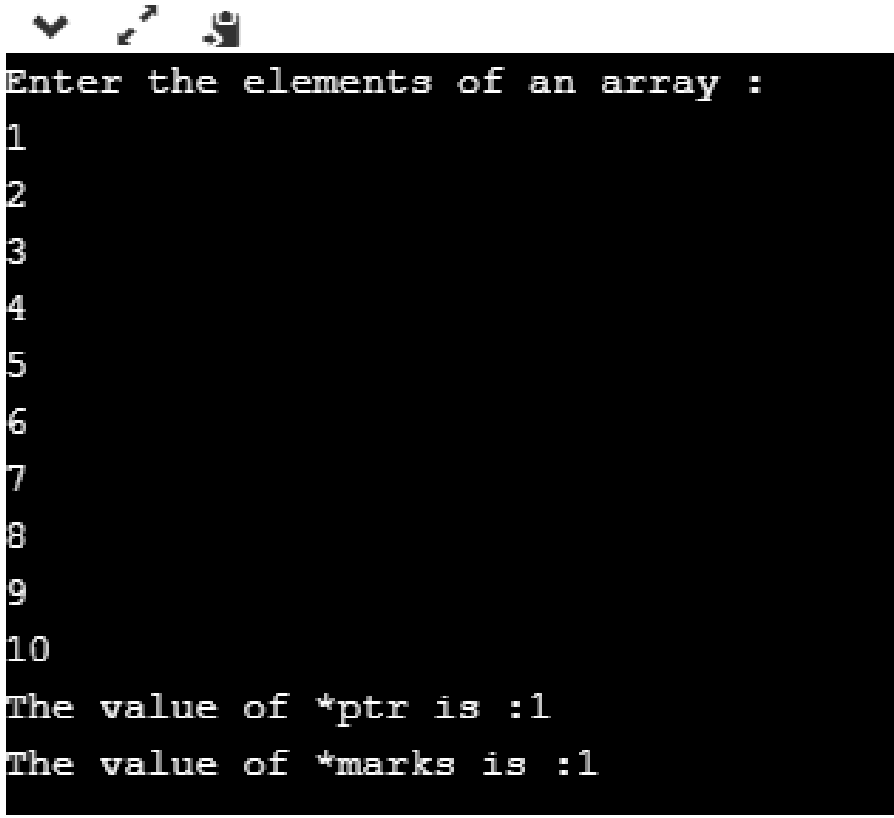
Array Name as Pointers

- An array name contains the address of the first element of the array which acts like a constant pointer. It means, the address stored in the array name can't be changed. For example, if we have an array named val then **val** and **&val[0]** can be used interchangeably.

```
#include <bits/stdc++.h>
using namespace std;
void geeks()
{
    // Declare an array
    int val[3] = { 5, 10, 20 };
// declare pointer variable
    int* ptr;
// Assign the address of val[0] to ptr
// We can use ptr=&val[0];(both are same)
    ptr = val;
    cout << "Elements of the array are: ";
    cout << ptr[0] << " " << ptr[1] << " " << ptr[2];
}
// Driver program
int main() { geeks(); }
```

The pointer contains 1st element of an array

```
#include <iostream>
using namespace std;
int main()
{
   int *ptr;  // integer pointer declaration
   int marks[10]; // marks array declaration
   std::cout << "Enter the elements of an array :" << std::endl;
   for(int i=0;i<10;i++)
   {
      cin>>marks[i];
   }
   ptr=marks; // both marks and ptr pointing to the same element..
   std::cout << "The value of *ptr is :" <<*ptr<< std::endl;
   std::cout << "The value of *marks is :" <<*marks<<std::endl;
}
```



// C++ Program to illustrate how to use array of pointers to strings

```
#include <cstring>
#include <iostream>
using namespace std;
#define SIZE 5
int main()
{
// declaring and initializing array of pointers
char* names[SIZE]
= { "Rahul", "Aman", "Abdul", "Ram", "Pradeep" };
for (int i = 0; i < SIZE; i++) {
   int currentStrLen = strlen(names[i]);
// accessing character
      char lastChar = names[i][currentStrLen - 1];
      cout << lastChar << " ";
   }
   cout << endl;

   // printing whole strings
   for (int i = 0; i < SIZE; i++) {
      cout << names[i] << " ";
   }
   cout << endl;

   // updating element
   names[2] = "Fashil";

   // printing whole strings
   for (int i = 0; i < SIZE; i++) {
      cout << names[i] << " ";
   }
return 0;
}
```

Output

l n l m p

Rahul Aman Abdul Ram Pradeep

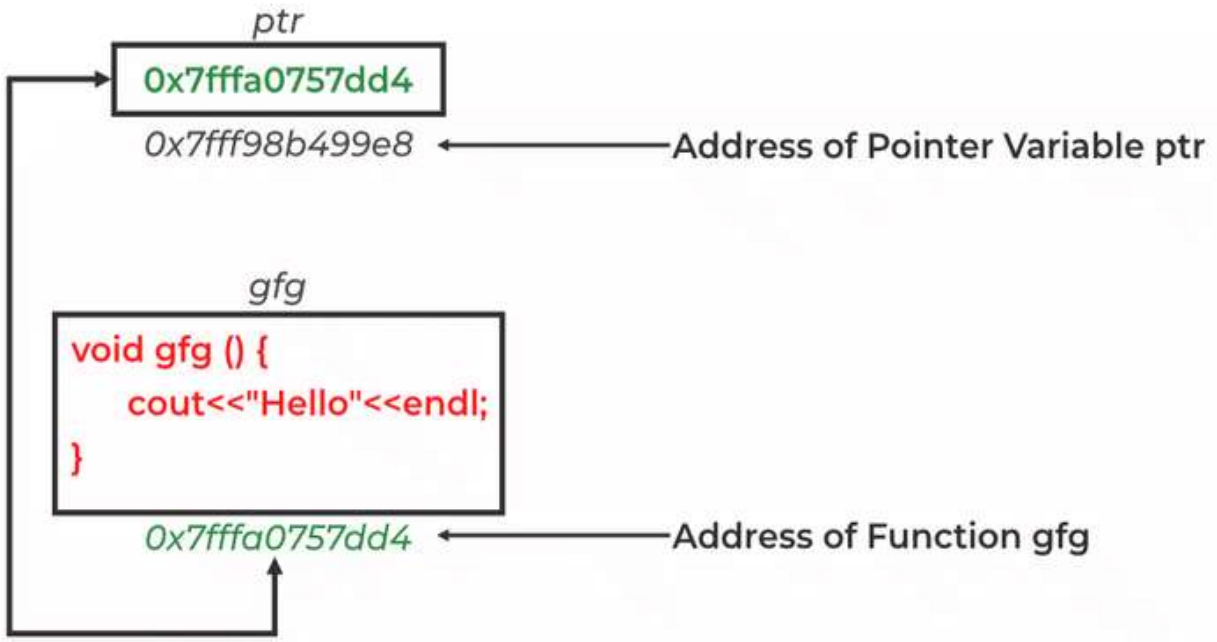
Rahul Aman Fashil Ram Pradeep

Function Pointer in C++

- The function pointer is used to point functions, similarly, the pointers are used to point variables.
- It is utilized to save a function's address.
- The function pointer is either used to call the function or it can be sent as an argument to another function.

Syntax:

return_type (*FuncPtr) (parameter type, ...)



Referencing and Dereferencing of the Function Pointer in C++

Syntax:

// Declaring

return_type (*FuncPtr) (parameter type, ...);

// Referencing

FuncPtr= function_name;

// Dereferencing

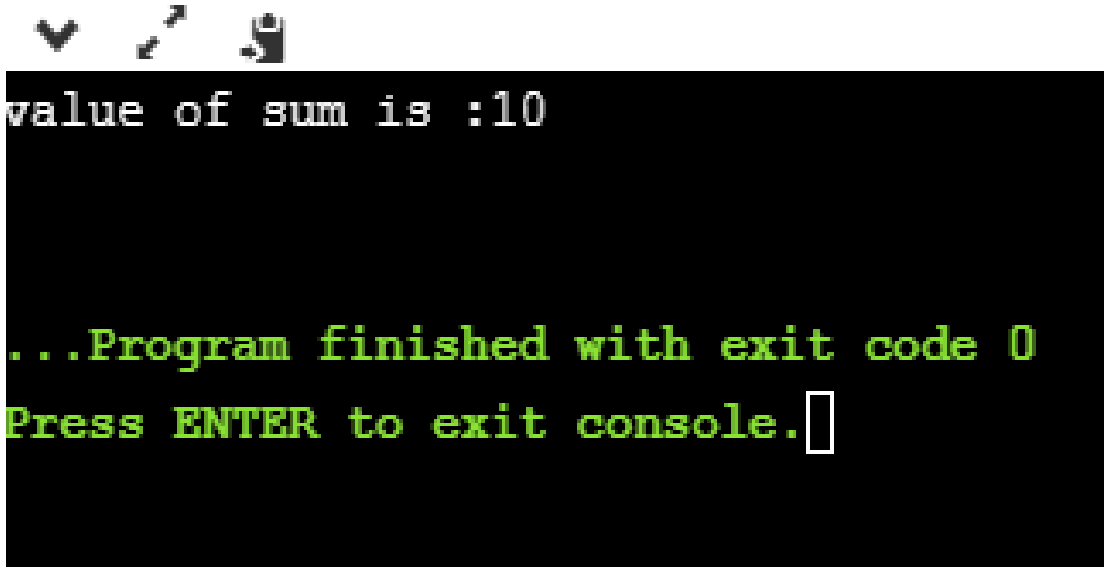
data_type x=*FuncPtr;

Function pointer used to call the function

```
#include <iostream>
using namespace std;
int multiply(int a, int b) { return a * b; }
int main()
{
	int (*func)(int, int);

	// func is pointing to the multiplyTwoValues function

	func = multiply;

	int prod = func(15, 2);
	cout << "The value of the product is: " << prod << endl;
	return 0;
}
```

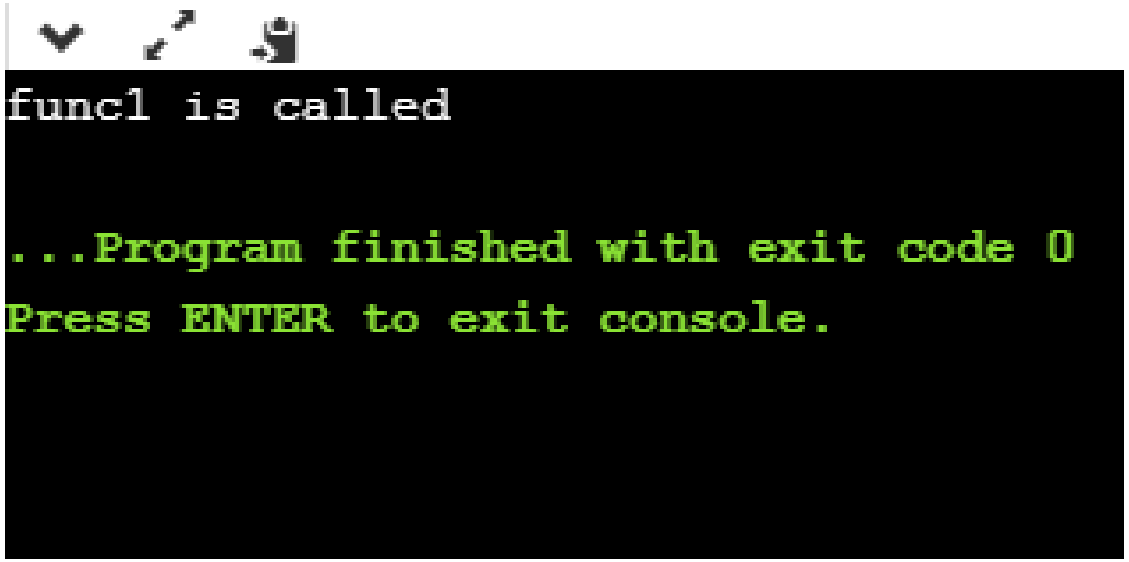
Calling a function indirectly

```
#include <iostream>
using namespace std;
int add(int a , int b)
{
   return a+b;
}
int main()
{
 int (*funcptr)(int,int);          // function pointer declaration
 funcptr=add;                      // funcptr is pointing to the add function
 int sum=funcptr(5,5);
 std::cout << "value of sum is :" <<sum<< std::endl;
  return 0;
}
```



Passing a function pointer as a parameter

```
#include <iostream>
using namespace std;
void func1()
{
   cout<<"func1 is called";
}
void func2(void (*funcptr)())
{
   funcptr();
}
int main()
{
  func2(func1);
  return 0;
}
```



new and delete Operators in C++ For Dynamic Memory

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

- **How is it different from memory allocated to normal variables?**

For normal variables like “int a”, “char str[10]”, etc, memory is automatically allocated and deallocated. For dynamically allocated memory like “int *p = new int[10]”, it is the programmer’s responsibility to deallocate memory when no longer needed. If the programmer doesn’t deallocate memory, it causes a memory leak (memory is not deallocated until the program terminates).

- In C++, memory is divided into two parts -
- **Stack -** All the variables that are declared inside any function take memory from the stack.
- **Heap -** It is unused memory in the program that is generally used for dynamic memory allocation.
- **Example:**

```
int main()
{
  // This memory for 10 integers is allocated on heap.
  int *ptr  = new int[10];
}
```

How is memory allocated/deallocated in C++?

C uses the malloc() and calloc() function to allocate memory dynamically at run time and uses a free() function to free dynamically allocated memory. C++ supports these functions and also has two operators new and delete, that perform the task of allocating and freeing the memory in a better and easier way.

new operator

The new operator denotes a request for memory allocation on the Free Store. If sufficient memory is available, a new operator initializes the memory and returns the address of the newly allocated and initialized memory to the pointer variable.

Syntax to use new operator

pointer-variable = new data-type;

Example:

```
// Pointer initialized with NULL Then request memory for the variable
int *p = NULL;
    p = new int;
```

OR // Combine declaration of pointer and their assignment

```
int *p = new int;
```

```
#include <iostream>
#include <memory>
using namespace std;
int main()
{
   // pointer to store the address returned by
the new
   int* ptr;
   // allocating memory for integer
   ptr = new int;
   // assigning value using dereference operator
   *ptr = 10;

   // printing value and address
   cout << "Address: " << ptr << endl;
   cout << "Value: " << *ptr;

   return 0;
}
```

- **Allocate a block of memory**: a new operator is also used to allocate a block(an array) of memory of type data type.

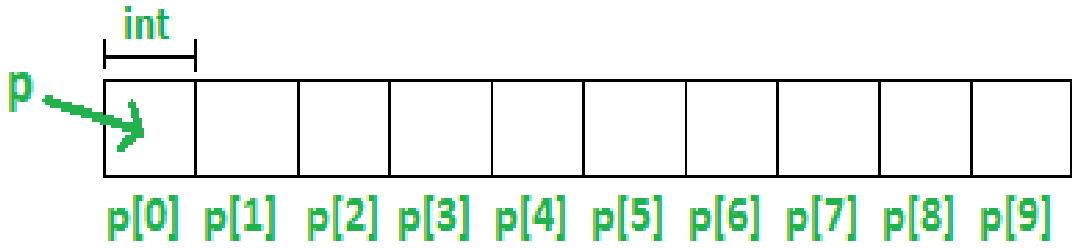
pointer-variable = new data-type[size];

where size(a variable) specifies the number of elements in an array.

Example:

int *p = new int[10]

Dynamically allocates memory for 10 integers continuously of type int and returns a pointer to the first element of the sequence, which is assigned top(a pointer). p[0] refers to the first element, p[1] refers to the second element, and so on.



Normal Array Declaration vs Using new :

There is a difference between declaring a normal array and allocating a block of memory using new. The most important difference is, that normal arrays are deallocated by the compiler (If the array is local, then deallocated when the function returns or completes). However, dynamically allocated arrays always remain there until either they are deallocated by the programmer or the program terminates.

What if enough memory is not available during runtime?

If enough memory is not available in the heap to allocate, the new request indicates failure by throwing an exception of type std::bad_alloc, unless "no throw" is used with the new operator, in which case it returns a NULL pointer Therefore, it may be a good idea to check for the pointer variable produced by the new before using its program.

```
int *p = new(nothrow) int;
if (!p)
{
   cout << "Memory allocation failed\n";
}
```

delete operator

- Since it is the programmer's responsibility to deallocate dynamically allocated memory, programmers are provided delete operator in C++ language.

Syntax:

// Release memory pointed by pointer-variable

delete pointer-variable;

Here, the pointer variable is the pointer that points to the data object created by new.

To free the dynamically allocated array pointed by pointer variable, use the following form of delete:

// Release block of memory pointed by pointer-variable

delete[] pointer-variable;

Example:

// It will free the entire array pointed by p.

delete[] p;

C++ program to illustrate dynamic allocation and deallocation of memory using new and delete

Output:

Value of p: 29

Value of r: 75.25

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

```
#include <iostream>
using namespace std;
int main()
{
   // Pointer initialization to null
   int* p = NULL;
  // Request memory for the variable
   // using new operator
   p = new (nothrow) int;
   if (!p)
      cout << "allocation of memory failed\n";
   else {
      // Store value at allocated address
      *p = 29;
      cout << "Value of p: " << *p << endl;
   }
// Request block of memory using new operator
   float* r = new float(75.25);
cout << "Value of r: " << *r << endl;
 // Request block of memory of size n
   int n = 5;
   int* q = new (nothrow) int[n];
if (!q)
      cout << "allocation of memory failed\n";
   else {
      for (int i = 0; i < n; i++)
         q[i] = i + 1;
cout << "Value store in block of memory: ";
      for (int i = 0; i < n; i++)
         cout << q[i] << " ";
   }
// freed the allocated memory
   delete p;
   delete r;
// freed the block of allocated memory
   delete[] q;
return 0;
}
```

What is a Pointer to an object?

- A pointer to an object in C++ is a variable that contains an object's memory address. Pointers provide indirect access to and control over memory items. They are especially helpful when we need to dynamically allocate memory for objects, build linked lists or trees, or pass objects by reference to methods without making duplicates.
- The behavior of an object pointer is identical to that of a **variable pointer**. But in this case, the **object's address** is kept instead of the **variables**. When a class object is formed in the **main function**, a pointer variable is declared similarly to the variable itself. Using a **data type** for the **pointer** is not recommended when generating a pointer to an object. Instead, we must make use of the object pointer's class name. The **->** symbol must be used to call a class member function using a Pointer in the **main function**.

Syntax:

It has the following syntax:

Declaring a Pointer to an Object:

We declare a pointer to an object using the object's class name followed by an asterisk (*) and the pointer name.

ClassName *pointer name; // Declaration of a pointer to an object

Creating Objects and Pointers:

Objects are created using the **new keyword**, which dynamically allocates memory for the object. After that, the Pointer is assigned the **memory address** of the newly created object.

```
ClassName *objPtr = new ClassName(); // Creating an object and assigning its address to the pointer .
```

Accessing Object Members via Pointer:

We can use the arrow operator (->) to access members (variables and functions) of the object through the pointer.

objects->memberFunction(); // Calling a member function through the pointer

int value = objPtr->memberVariable; // Accessing a member variable through the pointer

- **Dereferencing a Pointer:**

We ***"dereference"*** the ***pointer*** using the ***asterisk (*) operator*** to access the object itself (rather than its members).

ClassNameobj = *objPtr; // Dereferencing the pointer to get the object

Example of pointer to object

```
#include <iostream>
using namespace std;
class My_Class {
  int num;
public:
  void set_number(int value) {num = value;}
  void show_number();
};
void My_Class::show_number()
{
  cout << num << "\n";
}
```

```
int main()
{
  My_Class object, *p; // an object is declared and a pointer
to it

  object.set_number(1); // object is accessed directly
  object.show_number();

  p = &object; //  the address of the object is assigned to p
  p->show_number(); // object is accessed using the pointer

  return 0;
}
```

Object with two dimensional array

```
#include <iostream>
using namespace std;
class My_Class {
  int num;
public:
 void set_number(int val)
{num = val;}
 void show_number();
};
void My_Class::show_number()
{
  cout << num << "\n";
}
int main()
{
 My_Class object[2], *p;
object[0].set_number(10);  // objects is accessed directly

  object[1].set_number(20);

p = &object[0];  // the pointer is obtained to the first element
  p->show_number(); // value of object[0] is shown using pointer

p++;  // advance to the next object
  p->show_number(); // show value of object[1] is shown using
the pointer

  p--;  // retreat to previous object
  p->show_number(); // again value of object[0] is shown
 return 0;
}
```

Object with two dimensional array USING ARROW OPERATOR

```
#include <iostream>
using namespace std;
class My_Class {
int num;  // Private member variable
public:
void set_number(int val) {
num = val
void show_number(){
};
void My_Class::show_number() {
cout << num << "\n";
}
```

Object with two dimensional array USING ARROW OPERATOR

```
int main() {
// Declare an array of My_Class objects and a pointer p
    My_Class object[2], *p;
p = &object[0];  // Point p to the first object
 // Set the value of the first object to 10 using the pointer
 p->set_number(10);
 p->show_number();  // Output: 10
 p++;  // Increment the pointer to point to the next object
// Set the value of the second object to 20 using the pointer
    p->set_number(20);
    p->show_number();  // Output: 20
p--;  // Decrement the pointer to point back to the previous object
    p->show_number();  // Output: 10
 return 0;
}
```

VIRTUAL FUNCTION:

- A C++ virtual function is a member function in the base class that you redefine in a derived class. It is declared using the virtual keyword.
- we create the pointer to the base class that refers to all the derived objects. But, when base class pointer contains the address of the derived class object, always executes the base class function. This issue can only be resolved by using the 'virtual' function.
- A 'virtual' is a keyword preceding the normal declaration of a function.

when we don't use the virtual keyword.

```
#include <iostream>
using namespace std;
class A
{
   int x=5;
    public:
    void display()
    {
        std::cout << "Value of x is : " << x<<std::endl;
    }
};
class B: public A
{
    int y = 10;
public:
    void display()
    {
cout << "Value of y is : " <<y<<endl;
    }
};

int main()
{
    A *a;
    B b;
    a = &b;
   a->display();
    return 0;
}
```

Output:
Value of x is : 5

WITH VIRTUAL FUNCTION

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

Output:

Derived Class is invoked

VIRTUAL FUNCTION

```
#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;
}
```

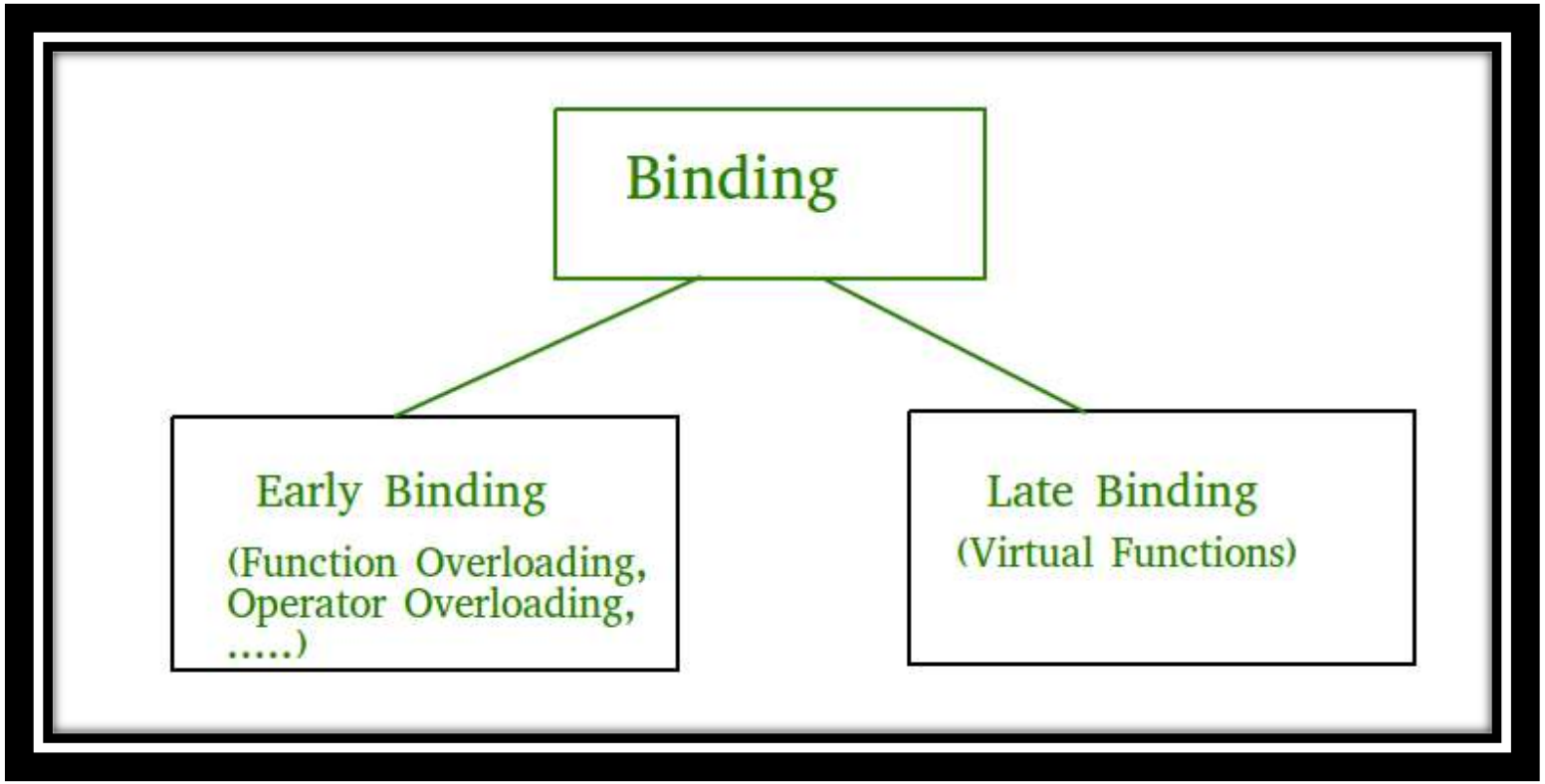
OUTPUT:

print derived class

show base class

Early binding and late binding

```
#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();
 p->fun_4(5);
return 0;
}
```



By default early binding happens in C++. Late binding (discussed below) is achieved with the help of virtual keyword)

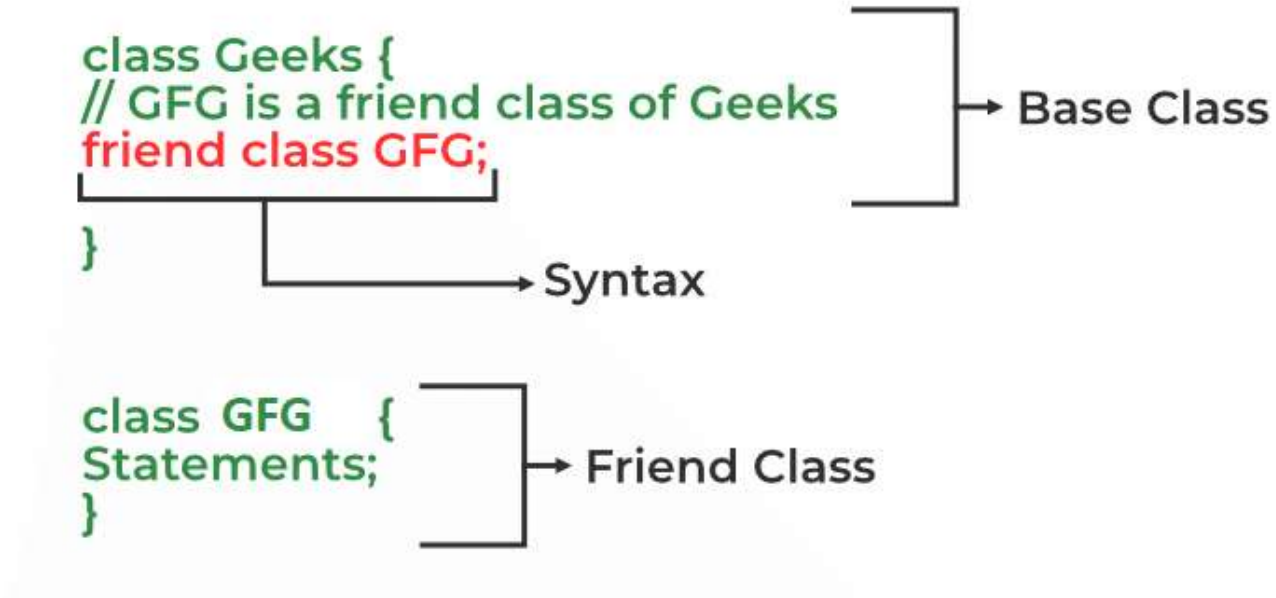
Friend Class

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. For example, a LinkedList class may be allowed to access private members of Node.

We can declare a friend class in C++ by using the **friend** keyword.

Syntax:

friend class class_name; // declared in the base class



```
#include <iostream>
using namespace std;
class GFG {
private:
    int private_variable;
protected:
    int protected_variable;
public:
    GFG()
    {
        private_variable = 10;
        protected_variable = 99;
    }
 // friend class declaration
    friend class F;
};
```

// Here, class F is declared as a friend inside class GFG. Therefore, F is a friend of class GFG. Class F can access the private members of class GFG.

```
class F {
public:
    void display(GFG& t)
    {
        cout << "The value of Private Variable = "<< t.private_variable <<
endl;
        cout << "The value of Protected Variable = "<< t.protected_variable;
    }
};
int main()
{
    GFG g;
    F fri;
    fri.display(g);
    return 0;
}
```

Friend Function

Like a friend class, a friend function can be granted special access to private and protected members of a class in C++. They are not the member functions of the class but can access and manipulate the private and protected members of that class for they are declared as friends.

A friend function can be:

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

Syntax:

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

or

friend return_type class_name::function_name (arguments);

// for a member function of another class

C++ program to create a global function as a friend function of some class

```
#include <iostream>
using namespace std;
class base {
private:
    int private_variable;
protected:
    int protected_variable;
public:
    base()
    {
        private_variable = 10;
        protected_variable = 99;
    }
     // friend function declaration
    friend void friendFunction(base& obj);
};
// friend function definition
void friendFunction(base& obj)
{
   cout << "Private Variable: " << obj.private_variable<< endl;
    cout << "Protected Variable: " << obj.protected_variable;
}
int main()
{
    base object1;
    friendFunction(object1);

    return 0;
}
```

C++ program to create a member function of another classas a friend function

```
#include <iostream>
using namespace std;
class base; // forward definition needed
// another class in which function is declared
class anotherClass {
public:
   void memberFunction(base& obj);
};
// base class for which friend is declared
class base {
private:
   int private_variable;
protected:
   int protected_variable;
public:
   base()
   {
      private_variable = 10;
      protected_variable = 99;
   }
 // friend function declaration
   friend void anotherClass::memberFunction(base&);
};
// friend function definition
void anotherClass::memberFunction(base& obj)
{
   cout << "Private Variable: " << obj.private_variable
        << endl;
   cout << "Protected Variable: " <<
obj.protected_variable;
}
```

```
int main()
{
    base object1;
    anotherClass object2;
    object2.memberFunction(object1);
return 0;
}
```

Output

Private Variable: 10

Protected Variable: 99

Points

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

Static function

- The static keyword is used with a variable to make the memory of the variable static once a static variable is declared its memory can’t be changed.
- Static members of a class are not associated with the objects of the class. Just like a static variable once declared is allocated with memory that can’t be changed every object points to the same memory.

Example:

```
class Person{
    static int index_number;
};
```

Example

```
#include <iostream>
using namespace std;
class Student {
public:
        // static member
        static int total;
// Constructor called
        Student() { total += 1; }
};
int Student::total = 0;
int main()
{
        Student s1;
        cout << "Number of students:" << s1.total << endl;
        Student s2;
        cout << "Number of students:" << s2.total << endl;
        Student s3;
        cout << "Number of students:" << s3.total << endl;
        return 0;
}
```

Output

Number of students:1
Number of students:2
Number of students:3

- Static Member Function in C++

- A static member function is independent of any object of the class.
- A static member function can be called even if no objects of the class exist.
- A static member function can also be accessed using the class name through the scope resolution operator.
- A static member function can access static data members and static member functions inside or outside of the class.
- You can also use a static member function to determine how many objects of the class have been created.

Example

```
#include <iostream>
using namespace std;
class Box
{
private:
static int length;
static int breadth;
static int height;
public:
static void print()
{
cout << "The value of the length is: " << length << endl;
cout << "The value of the breadth is: " << breadth << endl;
cout << "The value of the height is: " << height << endl;
}
};
int Box :: length = 10;
int Box :: breadth = 20;
int Box :: height = 30;
int main()
{
Box b;
cout << "Static member function is called through Object name: \n"
<< endl;
b.print();
cout << "\nStatic member function is called through Class name: \n"
<< endl;
Box::print();
return 0;
}
```

THIS Pointer ->

Output:

101 Sonoo 890000

102 Nakul 59000

```
#include <iostream>
using namespace std;
class Employee {
 public:
int id; //data member (also instance variable)
 string name; //data member(also instance variable)
 float salary;
Employee(int id, string name, float salary)
{
    this->id = id;
     this->name = name;
     this->salary = salary;
 }
void display()
      {
         cout<<id<<"  "<<name<<"  "<<salary
<<endl;
      }
};
int main(void) {
    Employee e1 =Employee(101, "Sonoo", 890000);
//creating an object of Employee
    Employee e2=Employee(102, "Nakul", 59000);
//creating an object of Employee
    e1.display();
    e2.display();
    return 0;
}
```