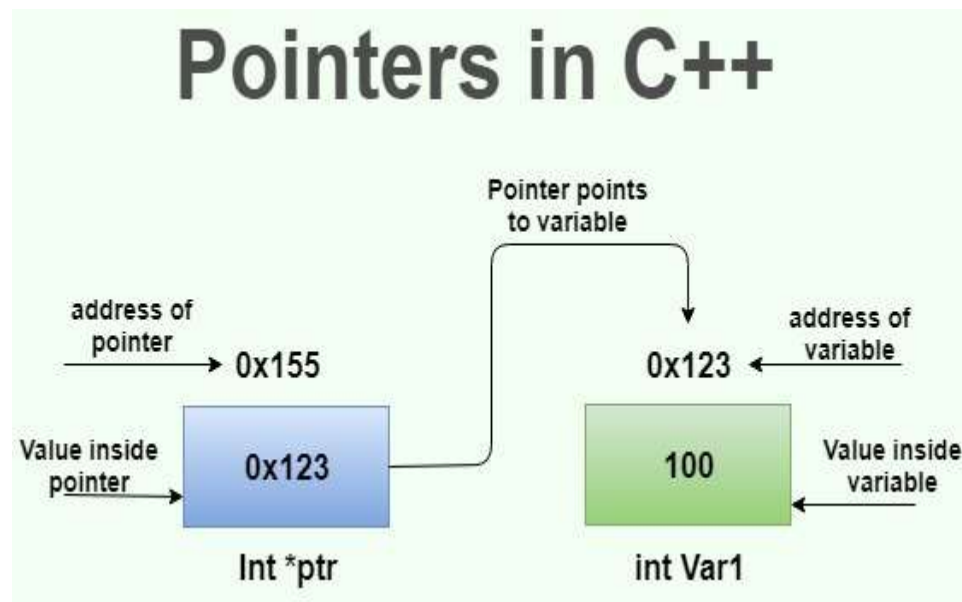


# Pointers in C++

Pointers are symbolic representation of addresses.

Pointers (pointer variables) are special variables that are used to store addresses rather than values.

They enable programs to simulate call-by-reference as well as to create and manipulate dynamic data structures.



## Pointers With Normal data type (Variables)

### Declaring pointer?

#### Syntax

Here is how we can declare pointers.

```
int* p;
```

Here, we have declared a pointer `p` of `int` type.

You can also declare pointers in these ways.

```
int *p1;
```

```
int* p2;
```

Another way...

```
int* p1, p2;
```

Here, we have declared a pointer p1 and a normal variable p2

## Assigning addresses to Pointers

Let's take an example.

```
int *pc, c;
```

```
c = 5;
```

```
pc = &c; //1000
```

Here, 5 is assigned to the c variable. And, the address of c is assigned to the pc pointer.

## Accessing pointed value

```
int* pc, c;
```

```
c = 5;
```

```
pc = &c;
```

```
printf("%d", pc); // Output: 5
```

Here, the address of `c` is assigned to the `pc` pointer. To get the value stored in that address, we used `*pc`.

## Changing Value Pointed by Pointers

Let's take an example.

```
int* pc, c;
```

```
c = 5;
```

```
pc = &c;
```

```
c = 1;
```

```
printf("%d", c); // Output: 1
```

```
printf("%d", *pc); // Ouptut: 1
```

Let's take one more example.

```
int* pc, c, d;
```

```
c = 5;
```

```
d = -15;
```

```
pc = &c; printf("%d", *pc); // Output: 5
```

```
pc = &d; printf("%d", *pc); // Ouptut: -15
```

### **Example:**

```
#include <stdio.h>

int main()
{
    int* pc, c;

    c = 22;

    printf("Address of c: %p\n", &c);
    printf("Value of c: %d\n\n", c);

    pc = &c;

    printf("Address of pointer pc: %p\n", pc);
    printf("Content of pointer pc: %d\n\n", *pc);

    c = 11;

    printf("Address of pointer pc: %p\n", pc);
    printf("Content of pointer pc: %d\n\n", *pc);

    *pc = 2;

    printf("Address of c: %p\n", &c);
    printf("Value of c: %d\n\n", c);

    return 0;}
```

```
Address of c: 0x7ffdbde41eb4
Value of c: 22

Address of pointer pc: 0x7ffdbde41eb4
Content of pointer pc: 22

Address of pointer pc: 0x7ffdbde41eb4
Content of pointer pc: 11

Address of c: 0x7ffdbde41eb4
Value of c: 2
```

## Pointers To objects//object pointer

C++ allows you to have pointers to objects. The pointers pointing to objects are referred to as Object Pointers.

Just like pointers to normal variables and functions, we can have pointers to class member functions and member variables.

### Declaration and Use of Object Pointers

Just like other pointers, the object pointers are declared by placing in front of a object pointer's name.

```
class-name * object-pointer ;
```

where class-name is the name of an already defined class and object-pointer is the pointer to an object of this class type. For example,

```
Sample *optr ; //sample* optr;
```

```
Optr=&obj1;
```

```
Obj1.a // obj1.fun()
```

```
->
```

where Sample is already defined class.

When accessing members of a class using an object pointer, the arrow operator (->) is used instead of dot operator.

```
#include <iostream>
```

```
using namespace std;
```

```
class Box {
```

```
    public:
```

```
        // Constructor definition
```

```
        Box(double l = 2.0, double b = 2.0, double h = 2.0) {
```

```
            cout <<"Constructor called." << endl;
```

```
            length = l;
```

```
            breadth = b;
```

```
            height = h;
```

```
        }
```

```
        double Volume() {
```

```
            return length * breadth * height;
```

```
        }
```

```
    private:
```

```
    double length;    // Length of a box
```

```
    double breadth;  // Breadth of a box
```

```
        double height;    // Height of a box
    };

int main( ) {

    Box Box1(3.3, 1.2, 1.5);    // Declare box1

    Box Box2(8.5, 6.0, 2.0);    // Declare box2

    Box *ptrBox;                // Declare pointer to a class.

    // Save the address of first object

    ptrBox = &Box1;

    // Now try to access a member using member access operator

    cout << "Volume of Box1: " << ptrBox->Volume() << endl;

    // Save the address of second object

    ptrBox = &Box2;

    // Now try to access a member using member access operator

    cout << "Volume of Box2: " << ptrBox->Volume() << endl;

    return 0;

}
```

## Pointer to Data Members of Class

We can use pointer to point to class's data members (Member variables).

### Syntax for Declaration :

```
datatype class_name :: *pointer_name;
```

```
int abc::*ptr
```

```
class abc
```

```
{
```

```
Int a;
```

```
Float f;
```

```
Char ch;
```

```
}
```

- The data-type is the data type of the data member.
- The class-name is the class name of the class of which the data member is a part of.
- The pointer-name is the name of the pointer, pointing to the member function.

### Syntax for Assignment:



```
pointer_name = &class_name :: datamember_name;
```

\*\*\*The data-member-name is the name of the data member being referenced.\*\*\*

Both declaration and assignment can be done in a single statement too.

```
datatype class_name::*pointer_name = &class_name::datamember_name ;
```

```
::*
```

### Using Pointers with Objects:

when we have a **pointer to data member**, we have to dereference that pointer to get what its pointing to,

syntax:

```
Object.*pointerToMember // *ptr
```

and with pointer to object, it can be accessed by writing,

syntax:

```
ObjectPointer->*pointerToMember
```

Dereferencing Operators	Description
<code>::*</code>	This operator allows us to <i>create a <b>pointer to a class member</b></i> , which could be <i>data member</i> or <i>member function</i> .
<code>-&gt;*</code>	This operator uses the <i><b>pointer to the member</b></i> of a class and a <i><b>pointer to the object</b></i> of the same class, to access the member of a class.
<code>.*</code>	This operator uses the <i><b>pointer to the member</b></i> of a class and an <i><b>object</b></i> of the same class, to access member of a class.

### Example (Object.\*pointerToMember)

```
#include<iostream>

using namespace std;

class A
{
public:
int x,y;
};

int main()
```

```

{
//Pointer to member variable x of class A
int A::* p1 = &A :: x;

//Pointer to member variable y of class A
int A::* p2 = &A :: y;


//Creating an object of class A
A ob;
ob.*p1 = 100; // ob.x=100
ob.*p2 = 10;
cout<<"The value of x is : " << ob.*p1 << "\n";
cout<<"The value of y is : " << ob.*p2 << "\n";
}

```

Example 2 (ObjectPointer->\*pointerToMember)

```

#include<iostream>

using namespace std;

class A
{
public:
int x,y;

};

int main()

```

```

{
A ob;

//Pointer to object

A *ptr = &ob;//

int A :: *p1 = &A :: x;

int A :: *p2 = &A :: y;

//Using pointer to object to access data member, pointed by a pointer

ptr->*p1 = 100;

//Using pointer to object to access data member, pointed by a pointer

ptr->*p2 = 30;

cout<<"The value of x is : " << ptr->*p1 << "\n";

cout<<"The value of y is : " << ptr->*p2 << "\n";

}

```

## Pointer to Member Functions of Class

### Syntax: (Assignment & Declaration)

**return\_type (class\_name::\*ptr\_name) (argument\_type) =  
&class\_name::function\_name;**

```
class Data
{
    public:
    int f(float)
    {
        return 1;
    }
};

Data obj1;
Obj1.f();

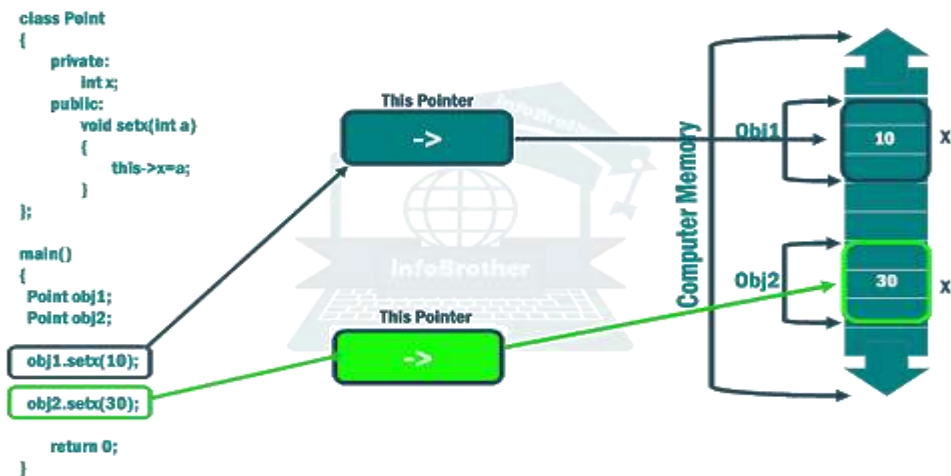
int (Data::*f p1) (float) = &Data::f; // Declaration and assignment
int (Data::*fp2) (float); // Only Declaration

int main()
{
    fp2 = &Data::f; // Assignment inside main()
}
```

## this pointer

To understand 'this' pointer, it is important to know how objects look at functions and data members of a class.

- Each object gets its own copy of the data member.
- All-access the same function definition as present in the code segment.
- The this pointer holds the address of current object, in simple words you can say that this pointer points to the current object of the class.
- The compiler supplies an implicit pointer along with the names of the functions as 'this'
- The 'this' pointer is passed as a hidden argument to all nonstatic member function calls and is available as a local variable within the body of all nonstatic functions.
- 'this' pointer is not available in static member functions as static member functions can be called without any object (with class name)



## Application of “this” pointer

1. To distinguish data member and local variables which is having the same name

Exempl:

```
#include<iostream>
```

```
using namespace std;
```

```
/* local variable is same as a member's name */
```

```
class Test
```

```
{
```

```
private:
```

```
int x;
```

```
public: void setX (int x){
```

```
    // The 'this' pointer is used to retrieve the object's x
```

```

this->x = x;

}

void print() { cout << "x = " << x << endl; }

};

int main()

{

Test obj;

int x = 20;

obj.setX(x);

obj.print();

return 0;

}

```

## 2. To return reference to the calling object

*/\* Reference to the calling object can be returned \*/*

```

Test& Test::func ()

{

    // Some processing

    return *this;

}

```



### 3. In Method Chaining

Once after returning the object from a function, a very useful application would be to chain the methods for ease and cleaner code.

Example: positionObj->setX(15)->setY(15)->setZ(15);

Example:

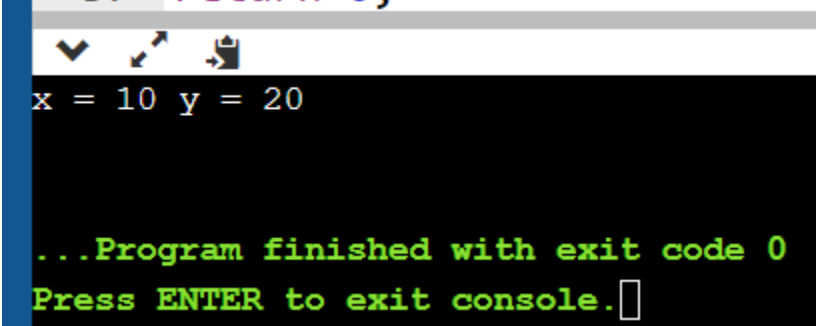
```
#include<iostream>
using namespace std;
```

```
class Test
{
private:
int x;
int y;
public:
Test(int x = 0, int y = 0)
{
this->x = x;
this->y = y;
}
Test &setX(int a)
{ x = a;
return *this;
}
Test &setY(int b)
{ y = b;
return *this;
}
void print()
{
cout << "x = " << x << " y = " << y << endl;
}
};
int main()
```

```
{  
Test obj1(5, 5);
```

```
// Chained function calls. All calls modify the same object  
// as the same object is returned by reference  
obj1.setX(10).setY(20);
```

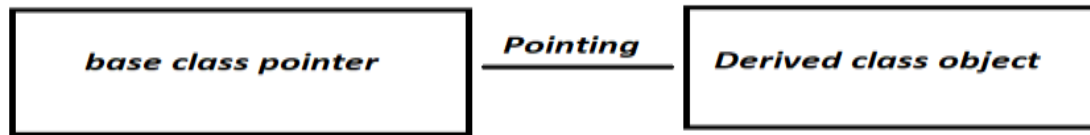
```
obj1.print();  
return 0;  
}
```



```
x = 10 y = 20
```

```
...Program finished with exit code 0  
Press ENTER to exit console.
```

## Pointer to derived class



*Used to point the derived class object and pointer can still use aspects of base class*

- A derived class is a class which takes some properties from its base class.
- It is true that a pointer of one class can point to other class, but classes must be a base and derived class, then it is possible.
- To access the variable of the base class, base class pointer will be used.
- So, a pointer is type of base class, and it can access all, public function and variables of base class since pointer is of base class, this is known as binding pointer.
- By using the pointer of base type **only the inherited properties and methods will be pointed**
- Further we can use **typecasting** or **virtual function** concept to point the additional member and methods of derived class by using the pointer of base class

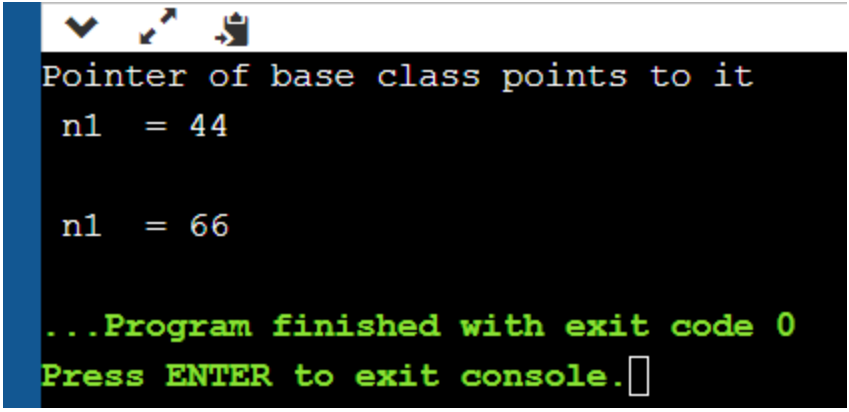
Example 1:

```
#include<iostream>
using namespace std;
class base
{
    public:
    int n1;
```

```

    void show()
    {
        cout<<"\n n1  = "<<n1;
    }
};
class derive : public base
{
    public:
    int n2;
    void show()
    {
        cout<<"\nn1 = " <<n1;
        cout<<"\nn2 = " <<n2;
    }
};
int main()
{
    base b; //object of base class
    base *bptr;    //base pointer
    cout<<"Pointer of base class points to it";
    bptr=&b;        //address of base class
    bptr->n1=44;      //access base class via base pointer
    bptr->show();
    derive d; //object of derive class
    cout<<"\n";
    bptr=&d;        //address of derive class
    bptr->n1=66;      //access derive class via base pointer
    bptr->show();
    return 0;
}

```



```
Pointer of base class points to it
n1  = 44

n1  = 66

...Program finished with exit code 0
Press ENTER to exit console.□
```

### Example 2:

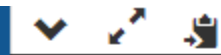
```
#include <iostream>
using namespace std;

class base
{
    public:
    void show()
    {
        cout<<"Base class "<<endl;
    }
};

class derived:public base
{
    public:
    void show()
    {
        cout<<"Derived class : "<<endl;
    }
};

int main()
{
    base b;
```

```
base *ptr;  
derived a;  
ptr=&a;  
ptr->show();  
return 0;  
}
```



Base class

...Program finished with exit code 0  
Press ENTER to exit console.█

## Virtual Function

Basically, a virtual function is used in the base class in order to ensure that the function is overridden. This especially applies to cases where a pointer of base class points to an object of a derived class.

It is declared using the **virtual** keyword.

- It is used to tell the compiler to perform dynamic linkage or late binding on the function.
- There is a necessity to use the single pointer to refer to all the objects of the different classes. So, 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 the function is made virtual, C++ determines which function is to be invoked at the runtime based on the type of the object pointed by the base class pointer.

### Rules of Virtual Function

- Virtual functions must be members of some class.
- Virtual functions cannot be static members.
- They are accessed through object pointers.
- They can be a friend of another class.
- A virtual function must be defined in the base class, even though it is not used.

- The prototypes of a virtual function of the base class and all the derived classes must be identical.

Consider the scenario when function is not using 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()
```

```
    {
```



```
        std::cout << "Value of y is : " <<y<< std::endl;
    }
};

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

Output:

Value of x is : 5

Note:

In the above example, \* a is the base class pointer. The pointer can only access the base class members but not the members of the derived class. Although C++ permits the base pointer to point to any object derived from the base class, it cannot directly access the members of the derived class.

Same above scenario 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(); //Late Binding occurs
}
```

Output:

Derived Class is invoked