**Introduction**

**Pointers :**

What are Pointers?

A **pointer** is a variable whose value is the address of another variable. Like any variable or constant, you must declare a pointer before you can work with it. **The general form of a pointer variable declaration is –**

**Pointer declarations and initialization :**

type \*var-name;

Here, **type** is the pointer's base type; it must be a valid C++ type and **var-name** is the name of the pointer variable. The asterisk you used to declare a pointer is the same asterisk that you use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Following are the valid pointer declaration −

int \*ip; // pointer to an integer

double \*dp; // pointer to a double

float \*fp; // pointer to a float

char \*ch // pointer to character

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

**Int \*ptr,a; // declaration**

**Ptr=&a; // initialization**

**Manipulation of pointers :**

There are few important operations, which we will do with the pointers very frequently. **(a)** We define a pointer variable. **(b)** Assign the address of a variable to a pointer. **(c)** Finally access the value at the address available in the pointer variable. This is done by using unary operator \* that returns the value of the variable located at the address specified by its operand. Following example makes use of these operations −

#include <iostream>

using namespace std;

int main () {

int var = 20; // actual variable declaration.

int \*ip; // pointer variable

ip = &var; // store address of var in pointer variable

cout << "Value of var variable: ";

cout << var << endl;

// print the address stored in ip pointer variable

cout << "Address stored in ip variable: ";

cout << ip << endl;

// access the value at the address available in pointer

cout << "Value of \*ip variable: ";

cout << \*ip << endl;

return 0;

}

**Output :**

Value of var variable: 20

Address stored in ip variable: 0xbfc601ac

Value of \*ip variable: 20

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**Pointers expressions and pointer arithmetic :**

**Pointer arithmetic**

**The C++ language allows you to perform integer addition or subtraction operations on pointers.**

* If ptr points to an integer, ptr + 1 is the address of the next integer in memory after ptr.
* ptr - 1 is the address of the previous integer before ptr.

Note that ptr + 1 does not return the *memory address* after ptr, but the memory address of the *next object of the type* that ptr points to. If ptr points to an integer (assuming 4 bytes), ptr + 3 means 3 integers (12 bytes) after ptr. If ptr points to a char, which is always 1 byte, ptr + 3 means 3 chars (3 bytes) after ptr.

When calculating the result of a pointer arithmetic expression, the compiler always multiplies the integer operand by the size of the object being pointed to. This is called **scaling**.

**Consider the following program:**

|  |  |
| --- | --- |
|  | #include <iostream>  int main()  {      int value = 7;      int \*ptr = &value;      std::cout << ptr << '\n';      std::cout << ptr+1 << '\n';      std::cout << ptr+2 << '\n';      std::cout << ptr+3 << '\n';      return 0;  } |

**this output:**

0012FF7C

0012FF80

0012FF84

0012FF88

**(OR)**

**Incrementing a Pointer ( + + )**

We prefer using a pointer in our program instead of an array because the variable pointer can be incremented, unlike the array name which cannot be incremented because it is a constant pointer. The following program increments the variable pointer to access each succeeding element of the array −

[Live Demo](http://tpcg.io/g0pNAF)

#include <iostream>

using namespace std;

const int MAX = 3;

int main () {

int var[MAX] = {10, 100, 200};

int \*ptr;

// let us have array address in pointer.

ptr = var;

for (int i = 0; i < MAX; i++) {

cout << "Address of var[" << i << "] = ";

cout << ptr << endl;

cout << "Value of var[" << i << "] = ";

cout << \*ptr << endl;

// point to the next location

ptr++;

}

return 0;

}

**OUTPUT :**

Address of var[0] = 0xbfa088b0

Value of var[0] = 10

Address of var[1] = 0xbfa088b4

Value of var[1] = 100

Address of var[2] = 0xbfa088b8

Value of var[2] = 200

**Decrementing a Pointer (- -)**

The same considerations apply to decrementing a pointer, which decreases its value by the number of bytes of its data type as shown below −

[Live Demo](http://tpcg.io/ujgz9K)

#include <iostream>

using namespace std;

const int MAX = 3;

int main () {

int var[MAX] = {10, 100, 200};

int \*ptr;

// let us have address of the last element in pointer.

ptr = &var[MAX-1];

for (int i = MAX; i > 0; i--) {

cout << "Address of var[" << i << "] = ";

cout << ptr << endl;

cout << "Value of var[" << i << "] = ";

cout << \*ptr << endl;

// point to the previous location

ptr--;

} return 0;

}

**OUTPUT**

Address of var[3] = 0xbfdb70f8

Value of var[3] = 200

Address of var[2] = 0xbfdb70f4

Value of var[2] = 100

Address of var[1] = 0xbfdb70f0

Value of var[1] = 10

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Using pointer with arrays and strings**

* Pointers are useful to allocate arrays dynamically
* We can decide the array size at runtime.
* Pointers do not refer to any section of memory
* Array refers to the block of memory space.

**Pointers with arrays**

#include <iostream>

using namespace std;

const int MAX = 3;

int main () {

int var[MAX] = {10, 100, 200};

for (int i = 0; i < MAX; i++) {

cout << "Value of var[" << i << "] = ";

cout << var[i] << endl;

}

return 0;

}

**OUTPUT** :

Value of var[0] = 10

Value of var[1] = 100

Value of var[2] = 200

There may be a situation, when we want to maintain an array, which can store pointers to an int or char or any other data type available.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Arrays of pointer**

A Situation in an array which can store pointers to an int or char or any data type variable is called “**ARRAY OF POINTERS”**

***Following is the declaration of an array of pointers to an integer −***

**int \*ptr[MAX];**

This declares **ptr** as an array of MAX integer pointers. Thus, each element in ptr, now holds a pointer to an int value. Following example makes use of three integers which will be stored in an array of pointers as follows −

[Live Demo](http://tpcg.io/qgpsAZ)

#include <iostream>

using namespace std;

const int MAX = 3;

int main () {

int var[MAX] = {10, 100, 200};

int \*ptr[MAX];

for (int i = 0; i < MAX; i++) {

ptr[i] = &var[i]; // assign the address of integer.

}

for (int i = 0; i < MAX; i++) {

cout << "Value of var[" << i << "] = ";

cout << \*ptr[i] << endl;

}

return 0;

}

**Output :**

Value of var[0] = 10

Value of var[1] = 100

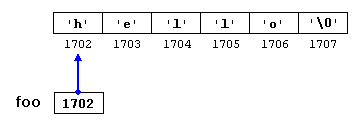
Value of var[2] = 200

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

### Pointers and string literals

String literals are arrays of the proper array type to contain all its characters plus the terminating null-character, with each of the elements being of type **const char** (as literals, they can never be modified). For example:

|  |  |  |
| --- | --- | --- |
|  | const char \* foo = "hello"; |  |

This declares an array with the literal representation for "hello", and then a pointer to its first element is assigned to foo. If we imagine that "hello" is stored at the memory locations that start at address 1702, we can represent the previous declaration as:  
   
Note that here foo is a pointer and contains the value 1702, and not 'h', nor "hello", although 1702 indeed is the address of both of these.  
  
The pointer foo points to a sequence of characters. And because pointers and arrays behave essentially in the same way in expressions, foo can be used to access the characters in the same way arrays of null-terminated character sequences are. For example:

|  |  |
| --- | --- |
| \*(foo+4)  foo[4] |  |

**Both expressions have a value of 'o' (the fifth element of the array)**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**Pointers to functions**

* Pointers to function is known as callback functions
* C++ allows operations with pointers to functions.
* The typical use of this is for passing a function as an argument to another function.
* Pointers to functions are declared with the same syntax as a regular function declaration, except that the name of the function is enclosed between parentheses () and an asterisk (\*) is inserted before the name:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 | // pointer to functions  #include <iostream>  using namespace std;  int addition (int a, int b)  { return (a+b); }  int subtraction (int a, int b)  { return (a-b); }  int operation (int x, int y, int (\*functocall)(int,int))  {  int g;  g = (\*functocall)(x,y);  return (g);  }  int main ()  {  int m,n;  int (\*minus)(int,int) = subtraction;  m = operation (7, 5, addition);  n = operation (20, m, minus);  cout <<n;  return 0;  }  **OUTPUT :**  **8** |  |

In the example above, minus is a pointer to a function that has two parameters of type int. It is directly initialized to point to the function subtraction:

**int (\* minus)(int,int) = subtraction;**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**POINTERS TO OBJECT :**

**A variable that holds an address value is called  a pointer variable or simply pointer.**

**Pointer can point to objects as well as to simple data types and arrays.**

sometimes we dont know, at the time that we write the program , how many objects we want to creat. when this is the case we can usenew to creat  objects while the program is running. new returns a pointer to an unnamed objects. lets see the example of student that wiil clear your idea about this topic

**EXAMPLE :**

#include <iostream>

#include <string>

using namespace std;

class student

{

private:

            int rollno;

            string name;

public:

            student(){

rollno(0),name("")

            }

            student(int r, string n): rollno(r),name (n)

            {}

            void get()

            {

                        cout<<"enter roll no";

                        cin>>rollno;

                        cout<<"enter  name";

                        cin>>name;

            }

            void print()

            {

                        cout<<"roll no is "<<rollno;

                        cout<<"name is "<<name;

            }

};

void main ()

{

            student \*ps=new student;

            (\*ps).get();

            (\*ps).print();

            delete ps;

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**This pointer (-🡪)**

**Every object in C++ has access to its own address through an important pointer called this pointer.**

The this **pointer** is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object. ... Only member functions have a this **pointer**.

**EXAMPLE :**

clas abc

{

-------

Int a;

};

// the private variable ‘a’ can be used directly inside the member function.

a=123;

// we can use this

This->a=123;

**Following are the situations where ‘this’ pointer is used:**

**1) When local variable’s name is same as member’s name**

|  |
| --- |
| #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         // hidden by the local variable 'x'         this->x = x;     }     void print() { cout << "x = " << x << endl; }  };  int main()  {     Test obj;     int x = 20;     obj.setX(x);     obj.print();     return 0;  } |

**Output:**

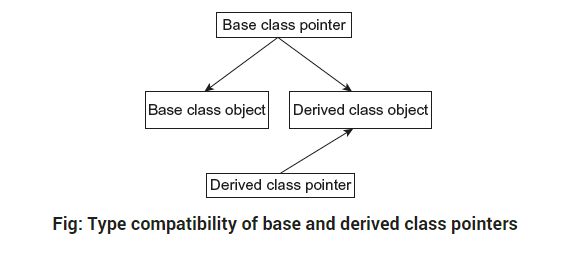
x = 20

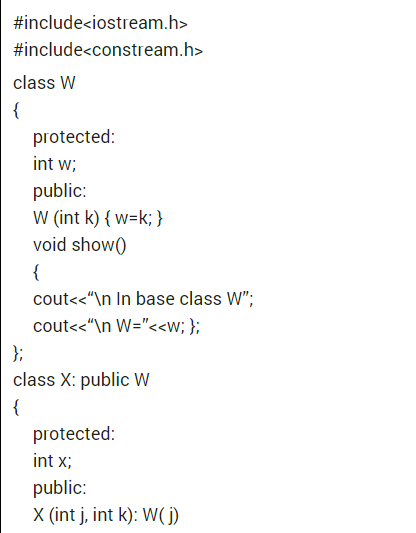
For constructors, [initializer list](https://www.geeksforgeeks.org/archives/13797) can also be used when parameter name is same as member’s name.

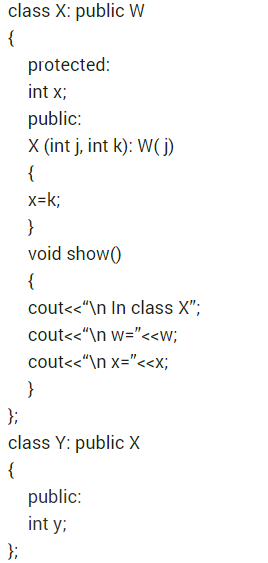
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

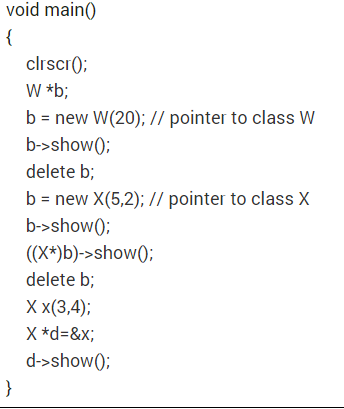
**Pointers to derieved class**

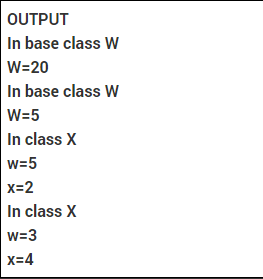
All **derived classes** inherit properties from the common base **class**. **Pointers** can be declared to the point base or **derived class**. **Pointers** to objects of the base **class** are type compatible with **pointers** to objects of the **derived class**. A base **class pointer** can point to objects of both the base and **derived class**.

****

****

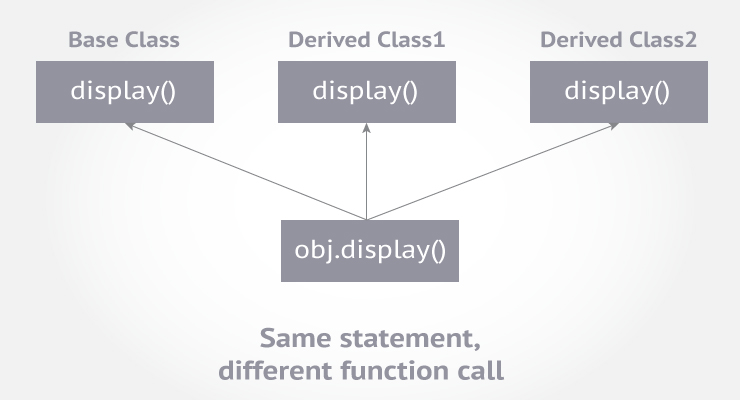
****

****

****

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

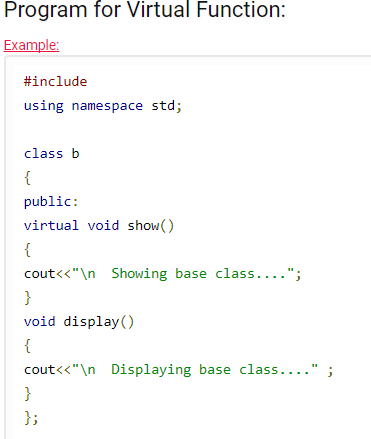
**Virtual functions**

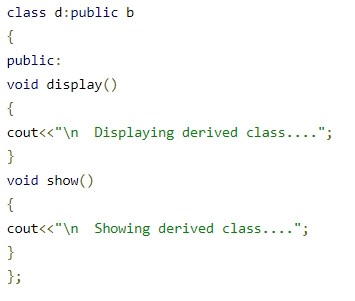
****

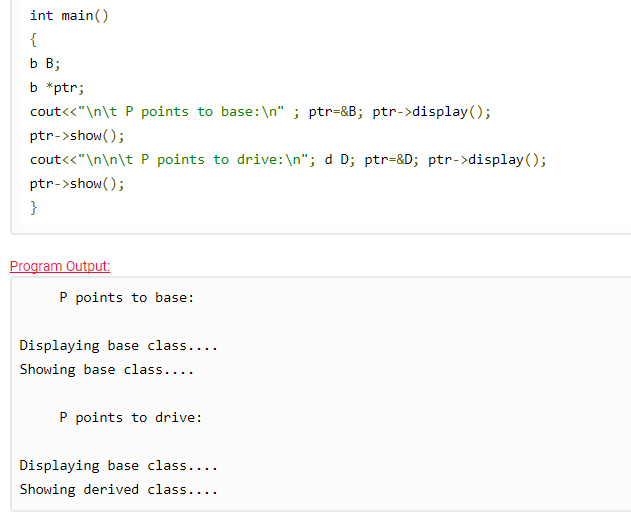
A virtual function is a member function in base class that you expect to redefine in derived classes.

**What is virtual function**

A virtual function is a special form of member function that is declared within a base class and redefined by a derived class. The keyword virtual is used to create a virtual function, precede the function's declaration in the base class. If a class includes a virtual function and if it gets inherited, the virtual class redefines a virtual function to go with its own need. In other words, a virtual function is a function which gets override in the derived class and instructs the C++ compiler for executing late binding on that function. A function call is resolved at runtime in late binding and so compiler determines the type of object at runtime.



****

****

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**Rules of virtual functions**

* Virtual functions must be members of a class.
* Virtual functions must be created in public section so that objects can access them.
* When virtual function is defined outside the class, virtual keyword is required only in the function declaration. Not necessary in the function definition.
* Virtual functions cannot be static members.
* Virtual functions must be accessed using a pointer to the object.
* A virtual function cannot be declared as a friend of another class.
* Virtual functions must be defined in the base class even though it does not have any significance.
* The signature of virtual function in base class and derived class must be same.
* A class must have a virtual destructor but it cannot have a virtual constructor.

We know that when a base class pointer refers to a derived class object, the extra features in derived class are not available. To access the extra features in the derived class, we make the functions in the base class as virtual.

**Syntax for creating a virtual function is as follows:**

|  |  |
| --- | --- |
|  | virtual return–type function–name(params–list)  {  //Body of function  ...  } |

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

### Abstract Class

Abstract Class is a class which contains atleast one Pure Virtual function in it. Abstract classes are used to provide an Interface for its sub classes. Classes inheriting an Abstract Class must provide definition to the pure virtual function, otherwise they will also become abstract class.

#### Characteristics of Abstract Class

1. Abstract class cannot be instantiated, but pointers and refrences of Abstract class type can be created.
2. Abstract class can have normal functions and variables along with a pure virtual function.
3. Abstract classes are mainly used for Upcasting, so that its derived classes can use its interface.
4. Classes inheriting an Abstract Class must implement all pure virtual functions, or else they will become Abstract too

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

#### Pure Virtual Functions

Pure virtual Functions are virtual functions with no definition. They start with **virtual** keyword and ends with = 0. Here is the syntax for a pure virtual function,

“ A do –nothing function “

virtual void f() = 0;

#### Example of Abstract Class

class Base //Abstract base class

{

public:

**virtual void show() = 0;** //Pure Virtual Function

};

class Derived:public Base

{

public:

void **show**()

{ cout << "Implementation of Virtual Function in Derived class"; }

};

int main()

{

Base obj; //Compile Time Error

Base \*b;

Derived d;

b = &d;

b->show();

}

**Output :**

Implementation of Virtual Function in Derived class

In the above example Base class is abstract, with pure virtual **show()** function, hence we cannot create object of base class.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**Virtual constructors and destructors**

* A constructor can not be virtual

**Some reasons are :**

1. Create an object the constructor of the object class mut be the same type as the class.
2. At the time of calling a constructor no any virtual function calls.

**VIRTUAL DESTRUCTORS :**

- : The explicit destroying of object with the use of delete operator to a base class pointer to the object is performed by the destructor of the base-class is invoked on that object.  
- : The above process can be simplified by declaring a virtual base class destructor.   
- : All the derived class destructors are made virtual in spite of having the same name as the base class destructor. In case the object in the hierarchy is destroyed explicitly by using delete operator to the base class pointer to a derived object, the appropriate destructor will be invoked

**Example**

Class A

{

Public:

~A()

{

//BASE CLASS DESTRUCTOR

}

};

Class B:public A

{

Public:

~B()

{

//derived CLASS DESTRUCTOR

}

};

Main()

{

A \*ptr=new B();

...

,...

Delete ptr;

}

// destructor using virtual

Class A

{

Public:

Virtual ~A()

{

//BASE CLASS DESTRUCTOR

}

};

**\*\*\*\*\*\*\*\*\*\*\***\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*