

# UNIT 2 OOPS

By Deepti Loharikar Talnikar

## Pointers and Arrays

Pointers are variables that hold addresses of other variables. Not only can a pointer store the address of a single variable, it can also store the address of cells of an array.

```
int *ptr;  
int arr[5];  
  
// store the address of the first  
// element of arr in ptr  
ptr = arr;
```

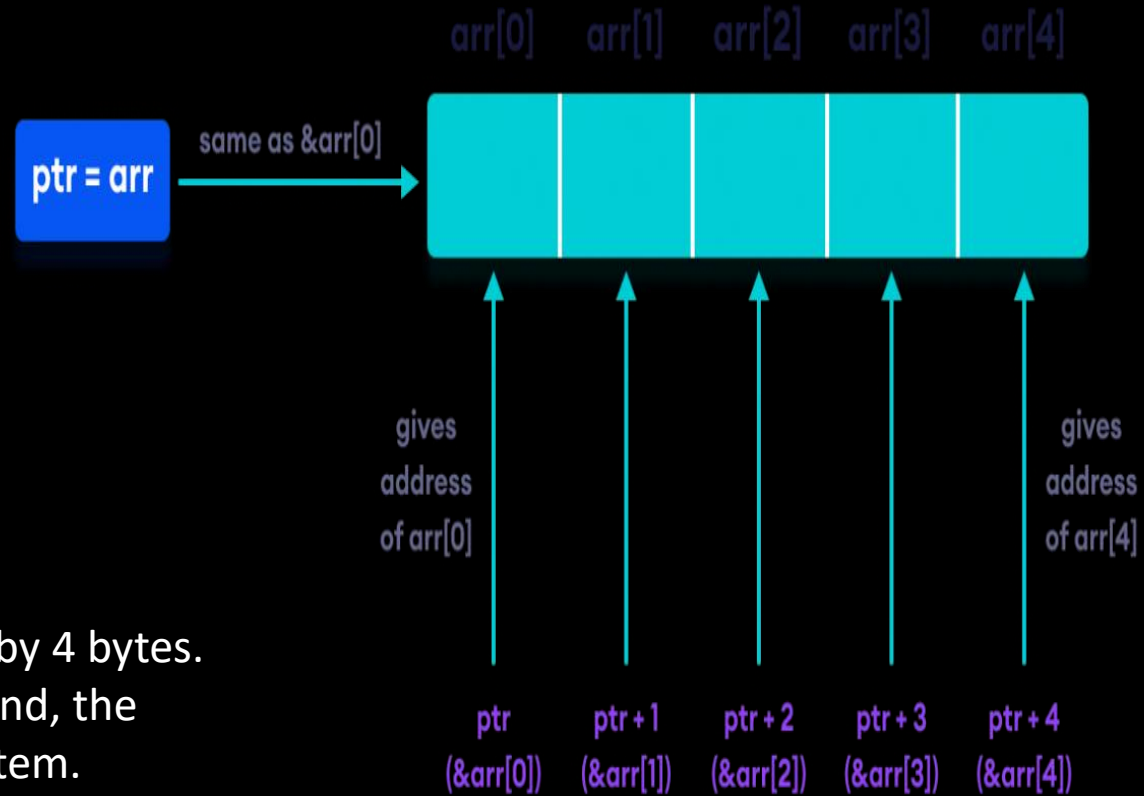
ptr is a pointer variable while arr is an int array. The code `ptr = arr;` stores the address of the first element of the array in variable ptr.

Notice that we have used arr instead of `&arr[0]`. This is because both are the same.

```
int *ptr;  
int arr[5];  
ptr = &arr[0];
```

we can access the elements using the single pointer.  
For example,

```
// use dereference operator
*ptr == arr[0];
*(ptr + 1) is equivalent to arr[1];
*(ptr + 2) is equivalent to arr[2];
*(ptr + 3) is equivalent to arr[3];
*(ptr + 4) is equivalent to arr[4];
```



The address between `ptr` and `ptr + 1` differs by 4 bytes. It is because `ptr` is a pointer to an `int` data. And, the size of `int` is 4 bytes in a 64-bit operating system.

Similarly, if pointer `ptr` is pointing to `char` type data, then the address between `ptr` and `ptr + 1` is 1 byte. It is because the size of a character is 1 byte.

```
// C++ Program to display address of each element of an array
```

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

```
int main()
{
    float arr[3];

    // declare pointer variable
    float *ptr;

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

    // use for loop to print addresses of all array elements
    for (int i = 0; i < 3; ++i)
    {
        cout << "&arr[" << i << "] = " << &arr[i] << endl;
    }
}
```

```
// ptr = &arr[0]
ptr = arr;

cout<<"\nDisplaying address using pointers: "<< endl;
// use for loop to print addresses of all array elements
// using pointer notation
for (int i = 0; i < 3; ++i)
{
    cout << "ptr + " << i << " = " << ptr + i << endl;
}
return 0;
}
```

O/P

Displaying address using arrays:

&arr[0] = 0x61fef0

&arr[1] = 0x61fef4

&arr[2] = 0x61fef8

Displaying address using pointers:

ptr + 0 = 0x61fef0

ptr + 1 = 0x61fef4

ptr + 2 = 0x61fef8

we first simply printed the addresses of the array elements without using the pointer variable ptr.

Then, we used the pointer ptr to point to the address of a[0], ptr + 1 to point to the address of a[1], and so on.

## Call by reference

passing arguments to a function where the actual values of arguments are not passed. Instead, the reference to values is passed.

// function that takes value as parameter

```
void func1(int numVal) {  
    // code  
}
```

// function that takes reference as parameter

// notice the & before the parameter

```
void func2(int &numRef) {  
    // code  
}
```

```
int main() {  
    int num = 5;
```

```
    // pass by value  
    func1(num);
```

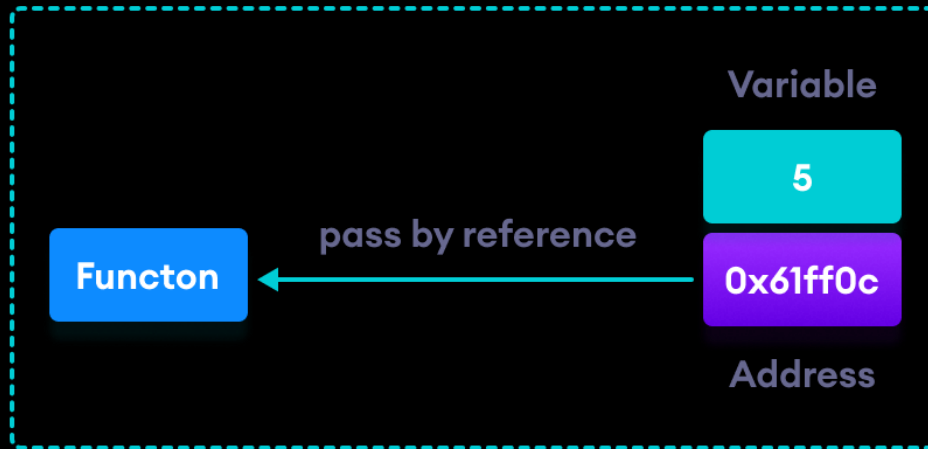
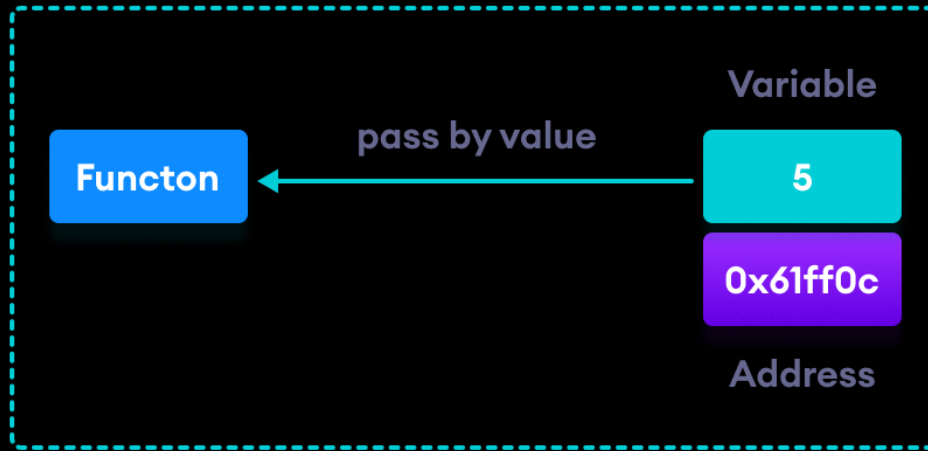
```
    // pass by reference  
    func2(num);
```

```
    return 0;
```

```
}
```

& in void func2(int &numRef). This denotes that we are using the address of the variable as our parameter.

So, when we call the func2() function in main() by passing the variable num as an argument, we are actually passing the address of num variable instead of the value 5.



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

// function prototype with pointer as parameters
void swap(int*, int*);

int main()
{

    // initialize variables
    int a = 1, b = 2;

    cout << "Before swapping" << endl;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;

    // call function by passing variable addresses
    swap(&a, &b);

    cout << "\nAfter swapping" << endl;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    return 0;
}
```

```
// function definition to swap numbers
void swap(int* n1, int* n2) {
    int temp;
    temp = *n1;
    *n1 = *n2;
    *n2 = temp;
}
```



o/p

Before swapping

a = 1

b = 2

After swapping

a = 2

b = 1

// &a is address of a

// &b is address of b

swap(&a, &b);

Here, the address of the variable is passed during the function call rather than the variable.

the address is passed instead of value, a dereference operator \* must be used to access the value stored in that address.

temp = \*n1;

\*n1 = \*n2;

\*n2 = temp;

\*n1 and \*n2 gives the value stored at address n1 and n2 respectively.

Since n1 and n2 contain the addresses of a and b, anything is done to \*n1 and \*n2 will change the actual values of a and b.

Hence, when we print the values of a and b in the main() function, the values are changed.

## C++ Inheritance

It allows us to create a new class (derived class) from an existing class (base class).

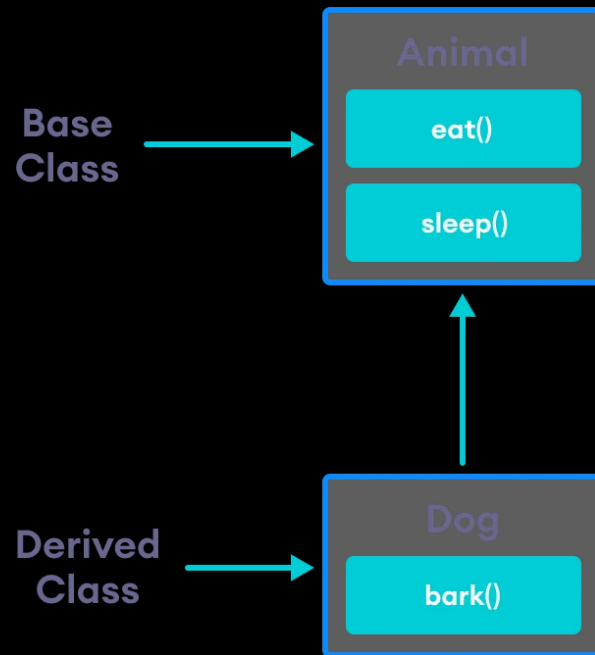
The derived class inherits the features from the base class and can have additional features of its own.

```
class Animal {  
    // eat() function  
    // sleep() function  
};
```

```
class Dog : public Animal {  
    // bark() function  
};
```

### SYNTAX

```
class Dog : public Animal {...};
```



is-a relationship

Inheritance is an is-a relationship. We use inheritance only if an is-a relationship is present between the two classes.

Here are some examples:

A car is a vehicle.

Orange is a fruit.

A surgeon is a doctor.

A dog is an animal.

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

```
// base class
class Animal {
```

```
    public:
    void eat() {
        cout << "I can eat!" << endl;
    }
```

```
    void sleep() {
        cout << "I can sleep!" << endl;
    }
};
```

```
// derived class
class Dog : public Animal {
    public:
    void bark() {
        cout << "I can bark! Woof woof!!"
        << endl;
    }
};

int main() {
    // Create object of the Dog class
    Dog dog1;
    // Calling members of the base class
    dog1.eat();
    dog1.sleep();
    // Calling member of the derived class
    dog1.bark();

    return 0;
}
```

o/p

I can eat!

I can sleep!

I can bark! Woof woof!!

dog1 (the object of derived class Dog) can access members of the base class Animal. It's because Dog is inherited from Animal.

// Calling members of the Animal class

dog1.eat();

dog1.sleep();

## Access Specifier

The various ways we can derive classes are known as access modes. These access modes have the following effect:

**public:** If a derived class is declared in public mode, then the members of the base class are inherited by the derived class just as they are.

**private:** In this case, all the members of the base class become private members in the derived class.

**protected:** The public members of the base class become protected members in the derived class.

The private members of the base class are always private in the derived class.

## Pointers to class type

```
class Simple
{
    public:
    int a;
};

int main()
{
    Simple obj;
    Simple* ptr; // Pointer of class type
    ptr = &obj;

    cout << obj.a;
    cout << ptr->a; // Accessing member with pointer
}
```

we have declared a pointer of class type which points to class's object. We can access data members and member functions using pointer name with arrow -> symbol.

## Pointers to data members of Class

### SYNTAX FOR DECLARATION

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

### SYNTAX FOR ASSIGNMENT

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

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

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

### POINTER TO MEMBER FUNCTIONS

```
return_type (class_name::*ptr_name) (argument_type) = &class_name::function_name;
```

## Pointers to objects

You can access an object either directly, or by using a pointer to the object.

To access an element of an object when using the actual object itself, use the dot operator.

To access a specific element of an object when using a pointer to the object, you must use the arrow operator.

### SYNTAX

`Object.*pointerToMember`

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

`ObjectPointer->*pointerToMember`



```
// A simple example using an object pointer.
```

```
#include <iostream>
```

```
using namespace std;
```

```
class My_Class {
```

```
    int num;
```

```
public:
```

```
    void set_num(int val) {num = val;}
```

```
    void show_num();
```

```
};
```

```
void My_Class::show_num()
```

```
{
```

```
    cout << num << "\n";
```

```
}
```

```
int main()
```

```
{
```

```
    My_Class ob, *p; // declare an object and pointer to  
    it
```

```
    ob.set_num(1); // access ob directly
```

```
    ob.show_num();
```

```
    p = &ob; // assign p the address of ob
```

```
    p->show_num(); // access ob using pointer
```

```
    return 0;
```

```
}
```

## Operator overloading

Operator overloading is a compile-time polymorphism in which the operator is overloaded to provide the special meaning to the user-defined data type. Operator overloading is used to overload or redefines most of the operators available in C++. It is used to perform the operation on the user-defined data type.

The advantage of Operators overloading is to perform different operations on the same operand.

Operator that cannot be overloaded are as follows:

Scope operator (::)

Sizeof

member selector (.)

member pointer selector (\*)

ternary operator (?:)

### SYNTAX

```
return_type class_name :: operator op(argument_list)
{
    // body of the function.
}
```

Where the return type is the type of value returned by the function.

class\_name is the name of the class.

operator op is an operator function where op is the operator being overloaded, and the operator is the keyword.

## Example of Operator overloading:

```
class Box {
public:
    double getVolume(void) {
        return length * breadth * height;
    }
    void setLength( double len ) {
        length = len;
    }
    void setBreadth( double bre ) {
        breadth = bre;
    }
    void setHeight( double hei ) {
        height = hei;
    }

    // Overload + operator to add two Box objects.
    Box operator+(const Box& b) {
        Box box;
        box.length = this->length + b.length;
        box.breadth = this->breadth + b.breadth;
        box.height = this->height + b.height;
        return box;
    }

private:
    double length;    // Length of a box
    double breadth;   // Breadth of a box
    double height;    // Height of a box
};
```

```
// Main function for the program
int main() {
    Box Box1;        // Declare Box1 of type Box
    Box Box2;        // Declare Box2 of type Box
    Box Box3;        // Declare Box3 of type Box
    double volume = 0.0; // Store the volume of a box here

    // box 1 specification
    Box1.setLength(6.0);
    Box1.setBreadth(7.0);
    Box1.setHeight(5.0);

    // box 2 specification
    Box2.setLength(12.0);
    Box2.setBreadth(13.0);
    Box2.setHeight(10.0);

    // volume of box 1
    volume = Box1.getVolume();
    cout << "Volume of Box1 : " << volume << endl;
    // volume of box 2
    volume = Box2.getVolume();
    cout << "Volume of Box2 : " << volume << endl;
    // Add two object as follows:
    Box3 = Box1 + Box2;
    // volume of box 3
    volume = Box3.getVolume();
    cout << "Volume of Box3 : " << volume << endl;

    return 0;
}
```

## THIS POINTER

This is a keyword that holds the address of current object or points to the current object of the class. . 3 main usage of this keyword in C++.

It can be used to pass current object as a parameter to another method.

It can be used to refer current class instance variable.

It can be used to declare indexers.

```
class Demo {
private:
    int num;
    char ch;
public:
    void setMyValues(int num, char ch){
        this->num =num;
        this->ch=ch;
    }
    void displayMyValues(){
        cout<<num<<endl;
        cout<<ch;
    }
};

int main(){
    Demo obj;
    obj.setMyValues(100, 'A');
    obj.displayMyValues();
    return 0;
}
```

O/P  
100  
A

## Copy constructor

The copy constructor in C++ is used to copy data of one object to another.

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

```
// declare a class
```

```
class Wall {
private:
    double length;
    double height;
```

```
public:
```

```
// initialize variables with parameterized constructor
```

```
Wall(double len, double hgt) {
    length = len;
    height = hgt;
}
```

```
// copy constructor with a Wall object as parameter
```

```
// copies data of the obj parameter
```

```
Wall(Wall &obj) {
    length = obj.length;
    height = obj.height;
```

```
}
double calculateArea() {
    return length * height;
}
```

```
};
```

```
int main() {
```

```
// create an object of Wall class
```

```
Wall wall1(10.5, 8.6);
```

```
// copy contents of wall1 to wall2
```

```
Wall wall2 = wall1;
```

```
// print areas of wall1 and wall2
```

```
cout << "Area of Wall 1: " << wall1.calculateArea() << endl;
```

```
cout << "Area of Wall 2: " << wall2.calculateArea();
```

```
return 0;
```

```
}
```

O/P

Area of Wall 1: 90.3

Area of Wall 2: 90.3

we have used a copy constructor to copy the contents of one object of the Wall class to another. The code of the copy constructor is:

```
Wall(Wall &obj) {  
    length = obj.length;  
    height = obj.height;  
}
```

Notice that the parameter of this constructor has the address of an object of the Wall class.

We then assign the values of the variables of the obj object to the corresponding variables of the object calling the copy constructor. This is how the contents of the object are copied.

In main(), we then create two objects wall1 and wall2 and then copy the contents of wall1 to wall2:

```
// copy contents of wall1 to wall2
```

```
Wall wall2 = wall1;
```

Here, the wall2 object calls its copy constructor by passing the address of the wall1 object as its argument i.e.

```
&obj = &wall1
```

## 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.

Meaning each object gets its own copy of data members and all objects share a single copy of member functions.

If only one copy of each member function exists and is used by multiple objects, how are the proper data members are accessed and updated?

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.

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



## 2) To return reference to the calling object

```
/* Reference to the calling object can be returned */
Test& Test::func ()
{
    // Some processing
    return *this;
}
```

When a reference to a local object is returned, the returned reference can be used to chain function calls on a single object.

```
#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;
}
```

Output:

x = 10 y = 20

## OPERATOR OVERLOADING

```
class className {  
    ... ..  
    public  
        returnType operator symbol (arguments) {  
            ... ..  
        }  
    ... ..  
};
```

operator is overloaded to provide the special meaning to the user-defined data type.

Operator that cannot be overloaded are as follows:

Scope operator (::)

Sizeof

member selector (.)

member pointer selector (\*)

ternary operator (?:)

## ++ Operator (Unary Operator) Overloading

// Overload ++ when used as prefix

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

```
class Count {
private:
    int value;
```

```
public:
```

```
    // Constructor to initialize count to 5
    Count() : value(5) {}
```

```
    // Overload ++ when used as prefix
    void operator ++ () {
        ++value;
    }
```

```
    void display() {
        cout << "Count: " << value << endl;
    }
```

```
};
```

```
int main() {
    Count count1;
```

```
    // Call the "void operator ++ ()" function
    ++count1;
```

```
    count1.display();
    return 0;
}
```

O/P

Count: 6

we use ++count1;, the void operator ++ () is called. This increases the value attribute for the object count1 by 1.

When we overload operators, we can use it to work in any way we like. For example, we could have used ++ to increase value by 100.

However, this makes our code confusing

## Operator Overloading in Binary Operators

Binary operators work on two operands.

When we overload the binary operator for user-defined types by using the code:

```
obj3 = obj1 + obj2;
```

The operator function is called using the obj1 object and obj2 is passed as an argument to the function.

```
// C++ program to overload the binary operator +
// This program adds two complex numbers
```

```
class Complex {
private:
    float real;
    float imag;
public:
    // Constructor to initialize real and imag to 0
    Complex() : real(0), imag(0) {}

    void input() {
        cout << "Enter real and imaginary parts
respectively: ";
        cin >> real;
        cin >> imag;
    }
    // Overload the + operator
    Complex operator + (Complex obj) {
        Complex temp;
        temp.real = real + obj.real;
        temp.imag = imag + obj.imag;
        return temp;
    }
}
```

```
void output() {
    if (imag < 0)
        cout << "Output Complex number: " << real << imag << "i";
    else
        cout << "Output Complex number: " << real << "+" << imag << "i";
    }
};

int main() {
    Complex complex1, complex2, result;

    cout << "Enter first complex number:\n";
    complex1.input();

    cout << "Enter second complex number:\n";
    complex2.input();

    // complex1 calls the operator function
    // complex2 is passed as an argument to the function
    result = complex1 + complex2;
    result.output();
    return 0;
}
```

O/P

Enter first complex number:

Enter real and imaginary parts respectively: 9 5

Enter second complex number:

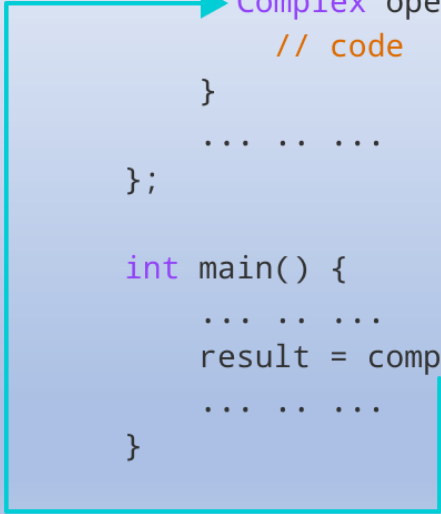
Enter real and imaginary parts respectively: 7 6

Output Complex number: 16+11i

using & makes our code efficient by referencing the complex2 object instead of making a duplicate object inside the operator function.

using const is considered a good practice because it prevents the operator function from modifying complex2.

```
class Complex {  
    ... ..  
public:  
    ... ..  
    Complex operator +(const Complex& obj) {  
        // code  
    }  
    ... ..  
};  
  
int main() {  
    ... ..  
    result = complex1 + complex2;  
    ... ..  
}
```



The diagram illustrates a function call from the `main` function to the `operator +` function of the `Complex` class. A red arrow originates from the `complex1` variable in the `main` function and points to the `Complex` parameter in the `operator +` function signature. Another red arrow points from the `complex2` variable in the `main` function to the `obj` parameter in the `operator +` function signature. A red box highlights the `Complex` parameter in the function signature and the `complex1` variable in the `main` function.

function call from complex1

## SELF REFERENTIAL CLASS

It is a special type of class. It is basically created for linked list and tree based implementation in C++. If a class contains the data member as pointer to object of similar class, then it is called a self-referential class.

```
class node {  
private:  
    int data;  
    node *next; //pointer to object of same type  
  
public:  
    //Member functions.  
};
```

In this declaration, the statement `node *next;` represents the self-referential class declaration, `node` is the name of same class and `next` the pointer to class (object of class).

## INHERITANCE

Inheritance is one of four pillars of Object-Oriented Programming (OOPs). It is a feature that enables a class to acquire properties and characteristics of another class. Inheritance allows you to reuse your code

```
class parent_class
```

```
{
```

```
    //class definition of the parent class
```

```
};
```

```
class child_class : visibility_mode parent_class
```

```
{
```

```
    //class definition of the child class
```

```
};
```



## TYPE OF INHERITANCE

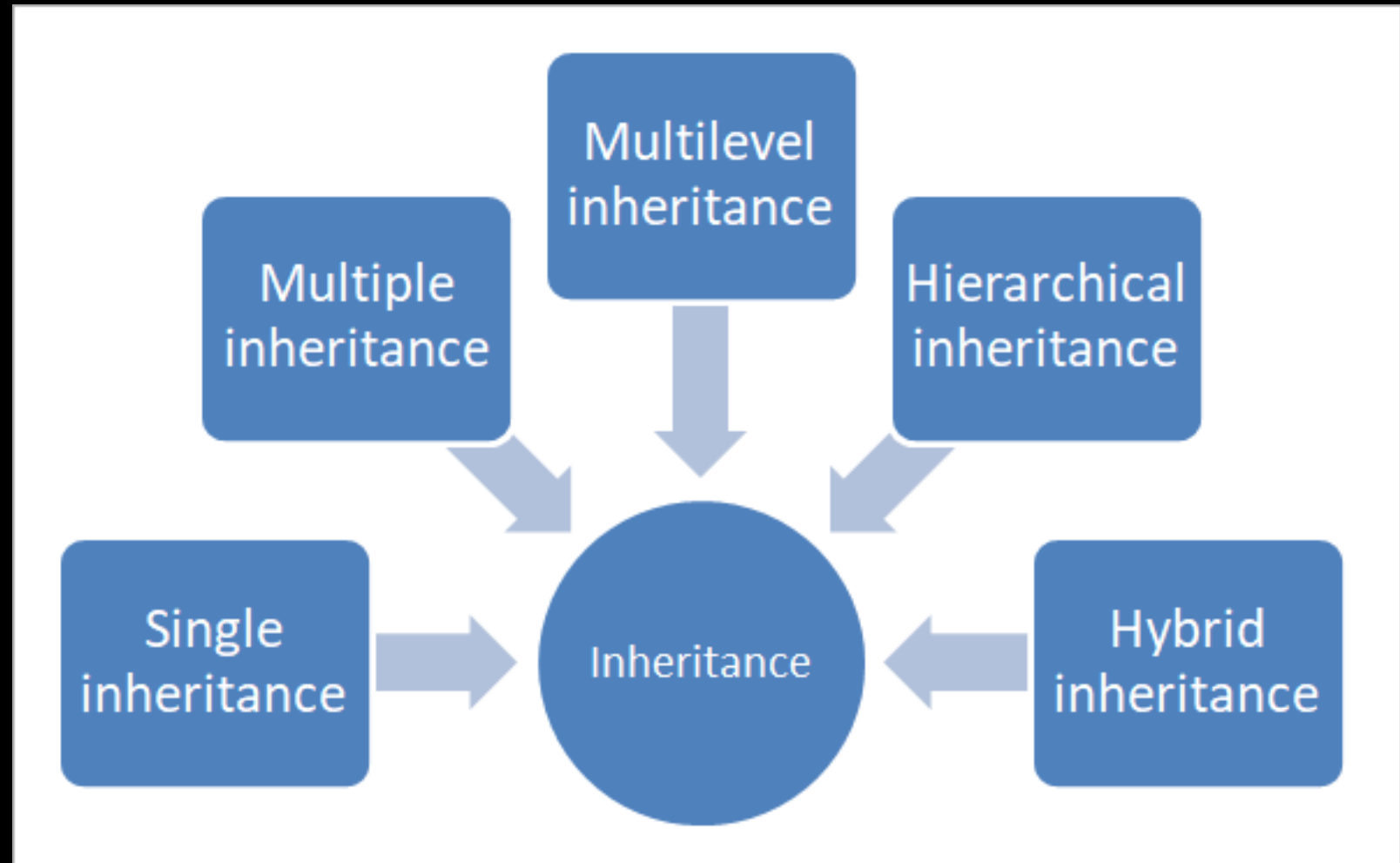
Single Inheritance

Multiple Inheritance

Multilevel Inheritance

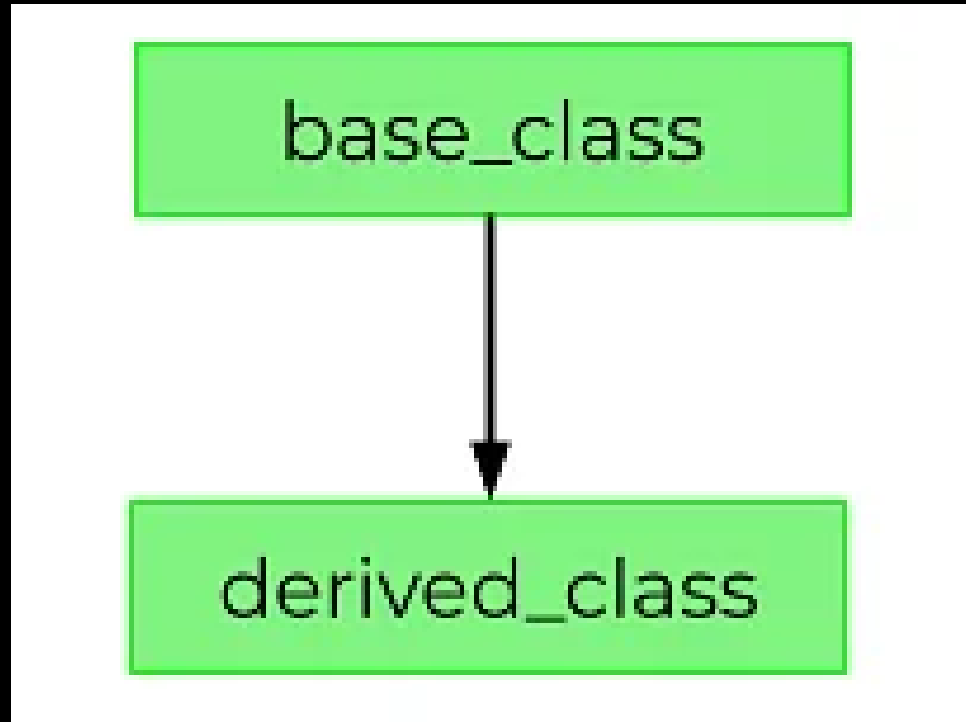
Hierarchical Inheritance

Hybrid Inheritance



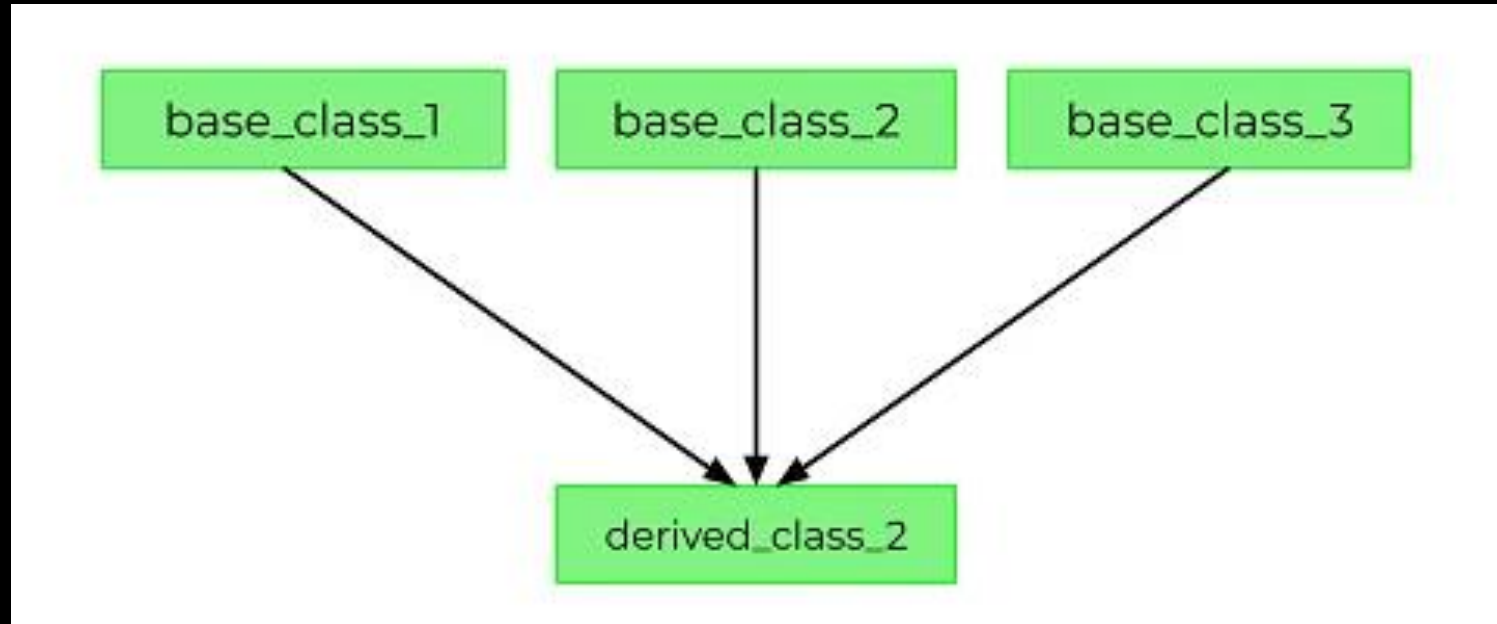
## Single Inheritance

In this inheritance, a single class inherits the properties of a base class.



## MULTIPLE INHERITANCE

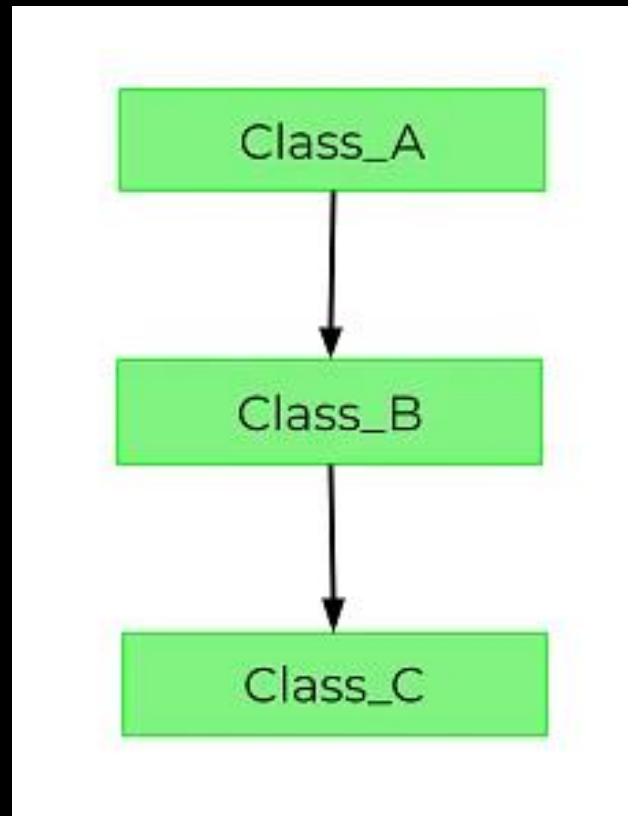
The inheritance in which a class can inherit or derive the characteristics of multiple classes, or a derived class can have over one base class, is known as Multiple Inheritance.



## MULTILEVEL INHERITANCE

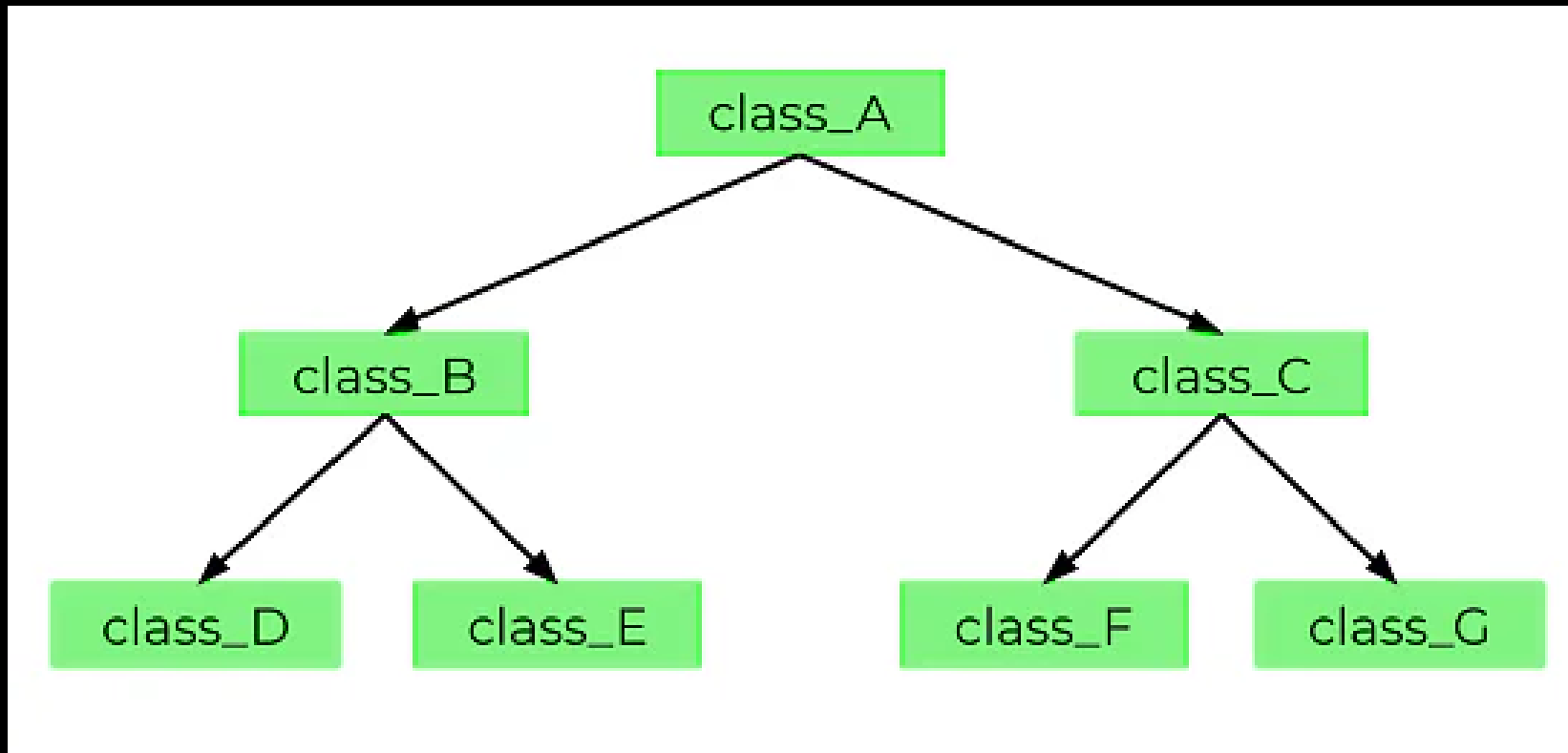
The inheritance in which a class can be derived from another derived class is known as Multilevel Inheritance.

There are three classes A, B, and C. A is the base class that derives from class B. So, B is the derived class of A. Now, C is the class that is derived from class B. This makes class B, the base class for class C but is the derived class of class A. This scenario is known as the Multilevel Inheritance.



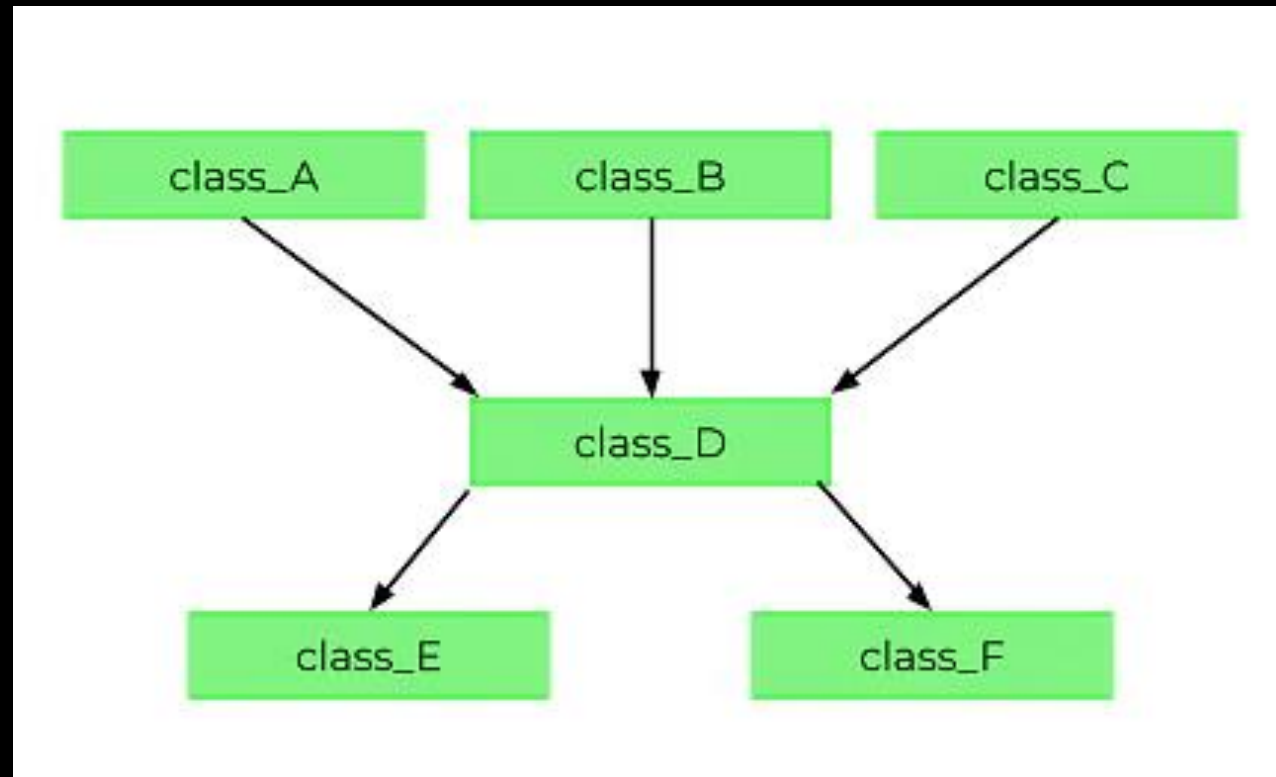
## HIERARCHICAL INHERITANCE

The inheritance in which a single base class inherits multiple derived classes is known as the Hierarchical Inheritance. This inheritance has a tree-like structure since every class acts as a base class for one or more child classes.



## HYBRID INHERITANCE

Hybrid Inheritance, is the combination of two or over two types of inheritances. For example, the classes in a program are in such an arrangement that they show both single inheritance and hierarchical inheritance at the same time. Such an arrangement is known as the Hybrid Inheritance.



```
class class_A
```

```
{
```

```
    // class definition
```

```
};
```

```
class class_B
```

```
{
```

```
    // class definition
```

```
};
```

```
class class_C: visibility_mode class_A, visibility_mode class_B
```

```
{
```

```
    // class definition
```

```
};
```

```
class class_D: visibility_mode class_C
```

```
{
```

```
    // class definition
```

```
};
```

```
class class_E: visibility_mode class_C
```

```
{
```

```
    // class definition
```

```
};
```

The derived class class\_C inherits two base classes that are, class\_A and class\_B. This is the structure of Multiple Inheritance. And two subclasses class\_D and class\_E, further inherit class\_C. This is the structure of Hierarchical Inheritance. The overall structure of Hybrid Inheritance includes more than one type of inheritance.



## DYNAMIC OBJECTS

In C++, dynamic memory allocation is done by using the new and delete operators. There is a similar feature in C using malloc(), calloc(), and deallocation using the free() functions. Note that these are functions. This means that they are supported by an external library. C++, however, imbibed the idea of dynamic memory allocation into the core of the language feature by using the new and delete operators. Unlike C's dynamic memory allocation and deallocation functions, new and delete in C++ are operators and are a part of the list of keywords used in C++.

Although, in both C and C++, dynamic memory is allocated in the heap area and gets a pointer returned to the object created, C++ is stricter in the construction norms. C++ never gives the memory of the allocated element direct access to initialize, but a constructor is used instead so as to ensure that:

- Initialization is guaranteed.

- There is no accidental access to an uninitialized object.

- A wrong sized object can't be returned.

Prior to the use of object created, proper initialization is vital. This one thing can help in avoiding many programming problems. Also note that the operators in C++ actually performs twin functions—it dynamically allocates the memory and also invokes the constructor for proper initialization.

The cleanup procedure with the delete operator also automatically calls the destructor.

## DYNAMIC ALLOCATION

Suppose you want to put a toy in a box, but you only have an approximate idea of its size. For that, you would require a box whose size is equal to the approximate size of the toy.

The new operator is used to allocate memory at runtime. The memory is allocated in bytes.

To allocate a variable dynamically.

```
int *ptr = new int;
```

By writing new int, we allocated the space in memory required by an integer. Then we assigned the address of that memory to an integer pointer ptr.

Suppose we declared the array with the array size 30 as follows.

```
char name[30];
```

And if the user enters the name having only 12 characters, then the rest of the memory space which was allocated to the array at the time of its declaration would become waste, thus unnecessary consuming the memory.

we will be using the new operator to dynamically allocate the memory at runtime.  
We use the new operator as follows.

```
char *arr = new char[length];
```

```
#include <iostream>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    int length, sum = 0;
```

```
    cout << "Enter the number of  
students in the group" << endl;
```

```
    cin >> length;
```

```
    int *marks = new int[length];
```

```
    cout << "Enter the marks of the  
students" << endl;
```

```
    for( int i = 0; i < length; i++ )        // entering marks of students
```

```
    {
```

```
        cin >> *(marks+i);
```

```
    }
```

```
    for( int i = 0; i < length; i++ )        // calculating sum
```

```
    {
```

```
        sum += *(marks+i);
```

```
    }
```

```
    cout << "sum is " << sum << endl;
```

```
    return 0;
```

```
}
```

## DELETE

Different groups have a different number of students, therefore we are asking the number of students ( i.e.the size of the array ) every time we are running the program. In this example, the user entered the size as 4.

`int *marks = new int[length];` - We declared an array of integer and allocated it some space in memory dynamically equal to the size which would be occupied by length number of integers. Thus it is allocated a space equal to 'length \* (size of 1 integer)' and assigned the address of the assigned memory to the pointer marks.

We call this array dynamic because it is being assigned memory when the program runs. We made this possible by using the new operator. This dynamic array is being allocated memory from heap unlike other fixed arrays which are provided memory from stack. We can give any size to these dynamic arrays and there is no limitation to it.

Now what if the variable to which we dynamically allocated some memory is not required anymore?

In that case, we use the delete operator.

To delete the memory assigned to a variable, we simply need to write the following code.

```
delete ptr;
```

```
int main()
{
    int length, sum = 0;
    cout << "Enter the number of students in the group" << endl;
    cin >> length;
    int *marks = new int[length];
    cout << "Enter the marks of the students" << endl;
    for( int i = 0; i < length; i++ )        // entering marks of students
    {
        cin >> *(marks+i);
    }
    for( int i = 0; i < length; i++ )        // calculating sum
    {
        sum += *(marks+i);
    }
    cout << "sum is " << sum << endl;
    delete[] marks;
    return 0;
}
```