1. ***Write a program using classes to multiply two complex numbers. Include multiple constructors as necessary. Also incorporate the concept of returning objects.***

Input: $a_1 = 4, b_1 = 8$
$a_2 = 5, b_2 = 7$
Output: Sum = 9 + i15
Explanation:
(4 + i8) + (5 + i7)
= (4 + 5) + i(8 + 7)
= 9 + i15

Input: $a_1 = 9, b_1 = 3$
$a_2 = 6, b_2 = 1$
Output: Sum = 15 + i4

```
// C++ program to Add two complex numbers
#include<bits/stdc++.h>
using namespace std;

// User Defined Complex class
class Complex {

	// Declaring variables
	public:
		int real, imaginary;

	// Empty Constructor
	Complex()
	{
	}

	// Constructor to accept
	// real and imaginary part
	Complex(int tempReal, int tempImaginary)
	{
		real = tempReal;
		imaginary = tempImaginary;
	}
```

```
        // Defining addComp() method
        // for adding two complex number
        Complex addComp(Complex C1, Complex C2)
        {
                // creating temporary variable
                Complex temp;

                // adding real part of complex numbers
                temp.real = C1.real + C2.real;

                // adding Imaginary part of complex numbers
                temp.imaginary = C1.imaginary + C2.imaginary;

                // returning the sum
                return temp;
        }
};

// Main Class
int main()
{

        // First Complex number
        Complex C1(3, 2);

        // printing first complex number
        cout<<"Complex number 1 : "<< C1.real
                                << " + i"<< C1.imaginary<<endl;

        // Second Complex number
        Complex C2(9, 5);

        // printing second complex number
        cout<<"Complex number 2 : "<< C2.real
                                << " + i"<< C2.imaginary<<endl;
```

```
        // for Storing the sum
        Complex C3;

        // calling addComp() method
        C3 = C3.addComp(C1, C2);

        // printing the sum
        cout<<"Sum of complex number : "
                                    << C3.real << " + i"
                                    << C3.imaginary;
}

// This code is contributed by chitranayal
```

2. ***Write down the syntax of defining the member function outside the class specification. How these functions can be made inline?***

Defining a member function outside a class requires the function declaration (function prototype) to be provided inside the class definition. The member function is declared inside the class like a normal function. This declaration informs the compiler that the function is a member of the class and that it has been defined outside the class. After a member function is declared inside the class, it must be defined (outside the class) in the program.

The definition of member function outside the class differs from normal function definition, as the function name in the function header is preceded by the class name and the scope resolution operator (: :). The scope resolution operator informs the compiler what class the member belongs to. The syntax for defining a member function outside the class is

```
1
2    Return_type class_name :: function_name (parameter_list) {
3
       // body of the member function

     }
```

To understand the concept of defining a member function outside a class, consider this example.

Example : Definition of member function outside the class

```
class book {

  // body of the class

};

 void book :: getdata(char a[],float b) {

    // defining member function outside the class

    strcpy(title,a):

    price = b:

}

  void book :: putdata () {

    cout<<"\nTitle of Book: "<<title;

    cout<<"\nPrice of Book: "<<price;

}
```

Note that the member functions of the class can access all the data members and other member functions of the same class (private, public or protected) directly by using their names. In addition, different classes can use the same function name.

Inside the Class: A member function of a class can also be defined inside the class. However, when a member function is defined inside the class, the class name and the scope resolution operator are not specified in the function header. Moreover, the member functions defined inside a class definition are by default inline functions.

To understand the concept of defining a member function inside a class, consider this example.

Example : Definition of a member function inside a class

```
class book {

  char title[30];

  float price;

  public:
```

```
8
9          void getdata(char [],float); II declaration

           void putdata()//definition inside the class {

           cout<<"\nTitle of Book: "<<title;

           cout<<"\nPrice of Book: "<<price;

      } ;
```

In this example, the member function putdata() is defined inside the class book. Hence, putdata() is by default an inline function.

Note that the functions defined outside the class can be explicitly made inline by prefixing the keyword inline before the return type of the function in the function header. For example, consider the definition of the function getdata().

```
1
2     inline void book ::getdata (char a [],float b) {
3
        body of the function

      }
```

3. *Explain the procedure to overload the pre-increment and post-increment operators with suitable examples.*

This C++ program overloads the pre-increment and post-increment operators for user-defined objects. The pre-increment operator is an operation where the value attribute of the object is incremented and the reference to resulting object
returned whereas in the post-increment operator, a local copy of the object is saved, the value attribute of the object is incremented and the reference to the local copy of the object is returned.
Here is the source code of the C++ program which overloads the pre-increment and post-increment operators for user-defined objects. The C++ program is successfully compiled and run on a Linux system. The program output is also shown below.

```
1. /*
2.  * C++ Program to overload pre-increment and post-increment operator
3.  */
4. #include <iostream>
5. using namespace std;
6.
7. class Integer {
```

```
    private:
        int value;
    public:
        Integer(int v) : value(v) { }
        Integer operator++();
        Integer operator++(int);
        int getValue() {
            return value;
        }
};

// Pre-increment Operator
Integer Integer::operator++()
{
   value++;
   return *this;
}

// Post-increment Operator
Integer Integer::operator++(int)
{
   const Integer old(*this);
   ++(*this);
   return old;
}

int main()
{
   Integer i(10);

   cout << "Post Increment Operator" << endl;
   cout << "Integer++ : " << (i++).getValue() << endl;
   cout << "Pre Increment Operator" << endl;
   cout << "++Integer : " << (++i).getValue() << endl;
}
```

```
$ a.out
Post Increment Operator
Integer++ : 10
Pre Increment Operator
Integer++ : 12
```

4. ***Write a function to return the division of two complex numbers, The real part and imaginary pat could be int, float or double.***

#include<iostream.h>

#include<conio.h>

```
#include<math.h>

struct complex

{

float rel;

float img;

}s1,s2;

void main()

{

clrscr();

float a,b;

cout<<“Enter real and imaginary part of 1st complex number:”;

cin>>s1.rel>>s1.img;

cout<<“Enter real and imaginary part of 2nd complex number:”;

cin>>s2.rel>>s2.img;

//Addition

a=(s1.rel)+(s2.rel);

b=(s1.img)+(s2.img);

cout<<“nAddition: “<<“(“<<a<<“)”<<“+”<<“(“<<b<<“)”<<“i”;

//Subtraction
```

```
a=(s1.rel)-(s2.rel);

b=(s1.img)-(s2.img);

cout<<“nSubtraction: “<<“(“<<a<<“)”<<“+”<<“(“<<b<<“)”<<“i”;

//Multiplication

a=((s1.rel)*(s2.rel))-((s1.img)*(s2.img));

b=((s1.rel)*(s2.img))+((s2.rel)*(s1.img));

cout<<“nMultiplication: “<<“(“<<a<<“)”<<“+”<<“(“<<b<<“)”<<“i”;

//Division

a=(((s1.rel)*(s2.rel))+((s1.img)*(s2.img)))/(pow(s2.rel,2)+pow(s2.img,2));

b=(((s2.rel)*(s1.img))-((s1.rel)*(s2.img)))/(pow(s2.rel,2)+pow(s2.img,2));

cout<<“nDivision: “<<“(“<<a<<“)”<<“+”<<“(“<<b<<“)”<<“i”;

getch();

}
```

5. *Write the characteristics of inline functions. List the differences between inline functions and macros.*

Inline function is the optimization technique used by the compilers. One can simply prepend inline keyword to function prototype to make a function inline. Inline function instruct compiler to insert complete body of the function wherever that function got used in code.

Advantages :- 1) It does not require function calling overhead.
2) It also save overhead of variables push/pop on the stack, while function calling.
3) It also save overhead of return call from a function.
4) It increases locality of reference by utilizing instruction cache.

5) After in-lining compiler can also apply intraprocedural optmization if specified. This is the most important one, in this way compiler can now focus on dead code elimination, can give more stress on branch prediction, induction variable elimination etc..

Disadvantages :-
1) May increase function size so that it may not fit on the cache, causing lots of cahce miss.
2) After in-lining function if variables number which are going to use register increases than they may create overhead on register variable resource utilization.
3) It may cause compilation overhead as if some body changes code inside inline function than all calling location will also be compiled.
4) If used in header file, it will make your header file size large and may also make it unreadable.
5) If somebody used too many inline function resultant in a larger code size than it may cause thrashing in memory. More and more number of page fault bringing down your program performance.
6) Its not useful for embeded system where large binary size is not preferred at all due to memory size constraints.

1. Inline :

An inline function is a normal function that is defined by the **inline** keyword. An inline function is a short function that is expanded by the compiler. And its arguments are evaluated only once. An inline functions are the short length functions that are automatically made the inline functions without using the **inline** keyword inside the class.

Syntax of an Inline function:

```
inline return_type function_name ( parameters )
{
 // inline function code
}
```

Example of an Inline function:

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

// Inline function
inline int Maximum(int a, int b)
{
    return (a > b) ? a : b;
}

// Main function for the program
int main()
{
    cout << "Max (100, 1000):" << Maximum(100, 1000)
<< endl;
    cout << "Max (20, 0): " << Maximum(20, 0) <<
endl;

    return 0;
}
```

Output:

```
Max (100, 1000): 1000
Max (20, 0): 20
```

2. Macro :

It is also called **preprocessors directive**. The macros are defined by the **#define** keyword. Before the program compilation, the preprocessor examines the program whenever the preprocessor detects the macros then preprocessor replaces the macro by the macro definition.

Syntax of Macro:

```
#define MACRO_NAME Macro_definition
```

Example of Macro:

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

// macro with parameter
#define MAXIMUM(a, b) (a > b) ? a
: b

// Main function for the program
int main()
{
    cout << "Max (100, 1000):";
    int k = MAXIMUM(100, 1000);
    cout << k << endl;

    cout << "Max (20, 0):";
    int k1 = MAXIMUM(20, 0);
    cout << k1;

    return 0;
}
```

Output:

```
Max (100, 1000):1000
Max (20, 0):20
```

Difference between Inline and Macro in C++ :

S.NO	Inline	Macro

1.	An inline function is defined by the **inline** keyword.	Whereas the macros are defined by the **#define** keyword.
2.	Through inline function, the class's data members can be accessed.	Whereas macro can't access the class's data members.
3.	In the case of inline function, the program can be easily debugged.	Whereas in the case of macros, the program can't be easily debugged.
4.	In the case of inline, the arguments are evaluated only once.	Whereas in the case of macro, the arguments are evaluated every time whenever macro is used in the program.
5.	In C++, inline may be defined either inside the class or outside the class.	Whereas the macro is all the time defined at the beginning of the program.
6.	In C++, inside the class, the short length functions are	While the macro is specifically defined.

	automatically made the inline functions.	
7.	Inline is not as widely used as macros.	While the macro is widely used.
8.	Inline is not used in competitive programming.	While the macro is very much used in competitive programming.
9.	Inline function is terminated by the curly brace at the end.	While the macro is not terminated by any symbol, it is terminated by a new line.

6. What are static data members? Explain and give a suitable example.

Static data members are class members that are declared using the static keyword. There is only one copy of the static data member in the class, even if there are many class objects. This is because all the objects share the static data member. The static data member is always initialized to zero when the first class object is created.

The syntax of the static data members is given as follows −

```
static data_type data_member_name;
```

In the above syntax, static keyword is used. The data_type is the C++ data type such as int, float etc. The data_member_name is the name provided to the data member.

A program that demonstrates the static data members in C++ is given as follows −

Example

```
#include <iostream>
```

```cpp
#include<string.h>

using namespace std;
class Student {
   private:
   int rollNo;
   char name[10];
   int marks;
   public:
   static int objectCount;
   Student() {
      objectCount++;
   }

   void getdata() {
      cout << "Enter roll number: "<<endl;
      cin >> rollNo;
      cout << "Enter name: "<<endl;
      cin >> name;
      cout << "Enter marks: "<<endl;
      cin >> marks;
   }

   void putdata() {
      cout<<"Roll Number = "<< rollNo <<endl;
      cout<<"Name = "<< name <<endl;
      cout<<"Marks = "<< marks <<endl;
      cout<<endl;
   }
};
int Student::objectCount = 0;
int main(void) {
   Student s1;
   s1.getdata();
   s1.putdata();
   Student s2;

   s2.getdata();
   s2.putdata();
   Student s3;

   s3.getdata();
   s3.putdata();
   cout << "Total objects created = " << Student::objectCount << endl;
   return 0;

}
```

Output

The output of the above program is as follows −

```
Enter roll number: 1
Enter name: Mark
Enter marks: 78
Roll Number = 1
Name = Mark
Marks = 78

Enter roll number: 2
Enter name: Nancy
Enter marks: 55
Roll Number = 2
Name = Nancy
Marks = 55

Enter roll number: 3
Enter name: Susan
Enter marks: 90
Roll Number = 3
Name = Susan
Marks = 90
Total objects created = 3
```

In the above program, the class student has three data members denoting the student roll number, name and marks. The objectCount data member is a static data member that contains the number of objects created of class Student. Student() is a constructor that increments objectCount each time a new class object is created.

There are 2 member functions in class. The function getdata() obtains the data from the user and putdata() displays the data. The code snippet for this is as follows −

```
class Student {
   private:
   int rollNo;
   char name[10];
   int marks;
   public:
   static int objectCount;
   Student() {
      objectCount++;
   }

   void getdata() {
      cout << "Enter roll number: "<<endl;
      cin >> rollNo;
```

```
      cout << "Enter name: "<<endl;
      cin >> name;
      cout << "Enter marks: "<<endl;
      cin >> marks;
   }

   void putdata() {
      cout<<"Roll Number = "<< rollNo <<endl;
      cout<<"Name = "<< name <<endl;
      cout<<"Marks = "<< marks <<endl;
      cout<<endl;
   }

};
```

In the function main(), there are three objects of class Student i.e. s1, s2 and s3. For each of these objects getdata() and putdata() are called. At the end, the value of objectCount is displayed. This is given below −

```
int main(void) {
   Student s1;
   s1.getdata();
   s1.putdata();

   Student s2;
   s2.getdata();
   s2.putdata();

   Student s3;
   s3.getdata();
   s3.putdata();

   cout << "Total objects created = " << Student::objectCount << endl;

   return 0;
}
```

7. ***Give the advantages of the reference variables over the pointer variables. Under which situation they should be avoided?***

Pointers: A pointer is a variable that holds memory address of another variable. A pointer needs to be dereferenced with * operator to access the memory location it points to.

References : A reference variable is an alias, that is, another name for an already existing variable. A reference, like a pointer, is also implemented by storing the address of an object.

A reference can be thought of as a constant pointer (not to be confused with a pointer to a constant value!) with automatic indirection, i.e the compiler will apply the * operator for you.

```
int i = 3;

// A pointer to variable i (or stores
// address of i)
int *ptr = &i;

// A reference (or alias) for i.
int &ref = i;
```

Differences :

1. Initialization: A pointer can be initialized in this way:

```
int a = 10;
  int *p = &a;
         OR
     int *p;
   p = &a;
we can declare and initialize pointer at same step or in
multiple line.
```

2. While in references,

```
int a=10;
int &p=a;  //it is correct
   but
int &p;
 p=a;    // it is incorrect as we should declare and
initialize references at single step.
```

3. NOTE: This differences may vary from compiler to compiler.The above differences is with respect to turbo IDE.

4. Reassignment: A pointer can be re-assigned. This property is useful for implementation of data structures like linked list, tree, etc. See the following examples:

```
int a = 5;
int b = 6;
int *p;
p =  &a;
p = &b;
```

5. On the other hand, a reference cannot be re-assigned, and must be assigned at initialization.

```
int a = 5;
int b = 6;
int &p = a;
int &p = b;  //At this line it will show error as "multiple
declaration is not allowed".

However it is valid statement,
int &q=p;
```

6. Memory Address: A pointer has its own memory address and size on the stack whereas a reference shares the same memory address (with the original variable) but also takes up some space on the stack.

```
int &p = a;
 cout << &p << endl << &a;
```

7. NULL value: Pointer can be assigned NULL directly, whereas reference cannot. The constraints associated with references (no NULL, no reassignment) ensure that the underlying operations do not run into exception situation.

8. Indirection: You can have pointers to pointers offering extra levels of indirection. Whereas references only offer one level of indirection.I.e,

```
In Pointers,
int a = 10;
int *p;
int **q;  //it is valid.
p = &a;
```

```
q = &p;

Whereas in references,

int &p = a;
int &&q = p; //it is reference to reference, so it is an
error.
```

9. Arithmetic operations: Various arithmetic operations can be performed on pointers whereas there is no such thing called Reference Arithmetic.(but you can take the address of an object pointed by a reference and do pointer arithmetics on it as in &obj + 5).)

When to use What

The performances are exactly the same, as references are implemented internally as pointers. But still you can keep some points in your mind to decide when to use what :

- Use references
    - In function parameters and return types.
- Use pointers:
    - Use pointers if pointer arithmetic or passing NULL-pointer is needed. For example for arrays (Note that array access is implemented using pointer arithmetic).
    - To implement data structures like linked list, tree, etc and their algorithms because to point different cell, we have to use the concept of pointers.

Quoted in C++ FAQ Lite : Use references when you can, and pointers when you have to. References are usually preferred over pointers whenever you don't need "reseating". This usually means that references are most useful in a class's public interface. References typically appear on the skin of an object, and pointers on the inside.

The exception to the above is where a function's parameter or return value needs a "sentinel" reference — a reference that does not refer to an object. This is usually best done by returning/taking a pointer, and giving the NULL pointer this special significance (references must always alias objects, not a dereferenced null pointer).

8. ***How garbage collection does takes place in C++? Illustrate with an example.***
9. ***Differentiate between compile time binding and run-time binding.***

Dynamic Binding :

The address of the functions are determined at runtime rather than @ compile time. This is also known as "Late Binding".

Static Binding :

The address of the functions are determined at compile time rather than @ run time. This is also known as "Early Binding"

Ex - 1: What is the output?

```
func(a,b)

int a,b;

{

 return( a= (a==b) );

}

main()

{

int process(),func();

printf("The value of process is %d !\n ",process(func,3,6));

}

process(pf,val1,val2)

int (*pf) ();

int val1,val2;

{

return((*pf) (val1,val2));

 }
```

Ans:

The value if process is 0 !

Explanation:

The function 'process' has 3 parameters - 1, a pointer to another function 2 and 3, integers. When this function is invoked from main, the following substitutions for formal parameters take place: func for pf, 3 for val1 and 6 for val2. This function returns the result of the operation performed by the function 'func'. The function func has two integer parameters. The formal parameters are substituted as 3 for a and 6 for b. since 3 is not equal to 6, a==b returns 0. therefore the function returns 0 which in turn is returned by the function 'process'.

Ex - 2: What is the output?

```
void main()
{
        static int i=5;
        if(--i){
                main();
                printf("%d ",i);
        }
}
```

Ans:

0 0 0 0

Explanation:

The variable "I" is declared as static, hence memory for I will be allocated for only once, as it encounters the statement. The function main() will be called recursively unless I

becomes equal to 0, and since main() is recursively called, so the value of static I ie., 0 will be printed every time the control is returned.

10. How virtual functions are implemented in C++?

Virtual functions in C++ used to create a list of base class pointers and call methods of any of the derived classes without even knowing kind of derived class object. Virtual functions are resolved late, at runtime.

Here is an implementation of virtual function in C++ program −

Example

```
#include <iostream>
using namespace std;
class B {
   public:
      virtual void s() { //virtual function
         cout<<" In Base \n";
      }
};
class D: public B {
   public:
      void s() {
         cout<<"In Derived \n";
      }
};
int main(void) {
   D d; // An object of class D
   B *b= &d; // A pointer variable of type B* pointing to d
   b->s(); // prints"D::s() called"
   return 0;

}
```

Output

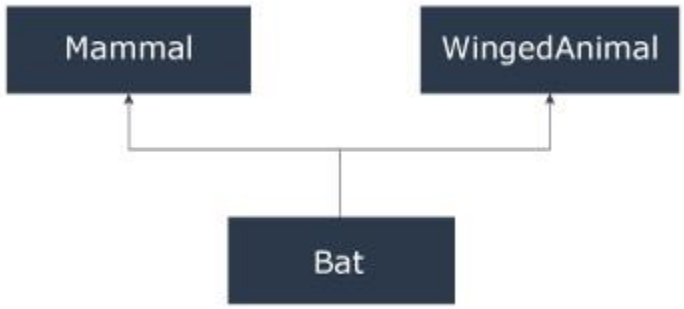
```
In Derived
```

11. What are the characteristics of inline functions? List the difference between inline and macros.

12. Discuss ambiguity in multiple inheritance. How ambiguity can be resolved in multiple inheritance? Illustrate with a proper example.

C++ Multiple Inheritance

In C++ programming, a class can be derived from more than one parents. For example: A class `Bat` is derived from base classes `Mammal` and `WingedAnimal`. It makes sense because bat is a mammal as well as a winged animal.



Example 2: Multiple Inheritance in C++ Programming

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

class Mammal {
  public:
    Mammal()
    {
      cout << "Mammals can give direct birth." << endl;
    }
};

class WingedAnimal {
  public:
    WingedAnimal()
    {
      cout << "Winged animal can flap." << endl;
    }
};

class Bat: public Mammal, public WingedAnimal {

};

int main()
```

```
{
    Bat b1;
    return 0;
}
```

Output

```
Mammals can give direct birth.
Winged animal can flap.
```

Ambiguity in Multiple Inheritance

The most obvious problem with multiple inheritance occurs during function overriding.

Suppose, two base classes have a same function which is not overridden in derived class.

If you try to call the function using the object of the derived class, compiler shows error. It's because compiler doesn't know which function to call. For example,

```
class base1
{
  public:
     void someFunction( )
     { .... ... .... }
};
class base2
{
    void someFunction( )
     { .... ... .... }
};
class derived : public base1, public base2
{

};
```

```
int main()
{
    derived obj;

    obj.someFunction() // Error!
}
```

This problem can be solved using scope resolution function to specify which function to class either `base1or base2`

```
int main()
{
    obj.base1::someFunction( );  // Function of base1 class is called
    obj.base2::someFunction();   // Function of base2 class is called.
}
```

13. Discuss the methods to overload an operator in C++. Write a C++ programs to overload unary minus(-) with each methods.

The unary operators operate on a single operand and following are the examples of Unary operators −

- The increment (++) and decrement (--) operators.
- The unary minus (-) operator.
- The logical not (!) operator.

The unary operators operate on the object for which they were called and normally, this operator appears on the left side of the object, as in !obj, -obj, and ++obj but sometime they can be used as postfix as well like obj++ or obj--.

Following example explain how minus (-) operator can be overloaded for prefix as well as postfix usage.

Live Demo

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

class Distance {
   private:
      int feet;             // 0 to infinite
```

```
      int inches;            // 0 to 12

   public:
      // required constructors
      Distance() {
         feet = 0;
         inches = 0;
      }
      Distance(int f, int i) {
         feet = f;
         inches = i;
      }

      // method to display distance
      void displayDistance() {
         cout << "F: " << feet << " I:" << inches <<endl;
      }

      // overloaded minus (-) operator
      Distance operator- () {
         feet = -feet;
         inches = -inches;
         return Distance(feet, inches);
      }
};

int main() {
   Distance D1(11, 10), D2(-5, 11);

   -D1;                       // apply negation
   D1.displayDistance();      // display D1

   -D2;                       // apply negation
   D2.displayDistance();      // display D2

   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
F: -11 I:-10
F: 5 I:-11
```

14. Write short note on pure virtual functions. Also discuss the need of pure virtual functions.

Virtual Function in C++

A virtual function a member function which is declared within a base class and is re-defined(Overriden) by a derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.

Pure Virtual Functions in C++

A pure virtual function (or abstract function) in C++ is a virtual function for which we don't have an implementation, we only declare it. A pure virtual function is declared by assigning 0 in the declaration.

Similarities between virtual function and pure virtual function

1. These are the concepts of Run-time polymorphism.
2. Prototype i.e. Declaration of both the functions remains the same throughout the program.
3. These functions can't be global or static.

Difference between virtual function and pure virtual function in C++

Virtual function	Pure virtual function
A virtual function is a member function of base class which can be redefined by derived class.	A pure virtual function is a member function of base class whose only declaration is provided in base class and should be defined in derived class otherwise derived class also becomes abstract.

Classes having virtual functions are not abstract.

Syntax:

filter_none

brightness_4

```
virtual<func_type><func_n
ame>()
{
    // code
}
```

Definition is given in base class.

Base class having virtual function can be instantiated i.e. its object can be made.

Base class containing pure virtual function becomes abstract.

Syntax:

filter_none

brightness_4

```
virtual<func_type><func_n
ame>()
    = 0;
```

No definition is given in base class.

Base class having pure virtual function becomes abstract i.e. it cannot be instantiated.

If derived class do not redefine virtual function of base class, then it does not affect compilation.	If derived class do not redefine virtual function of base class, then no compilation error but derived class also becomes abstract just like the base class.
All derived class may or may not redefine virtual function of base class.	All derived class must redefine pure virtual function of base class otherwise derived class also becomes abstract just like base class.

Pure virtual functions are used to build abstract classes. These classes cannot be instantiated. This means we cannot create objects of their type. This feature comes very much in handy when building *interfaces*, such as APIs.

Interfaces usually encapsulate an *abstract level of generality* for your code. This means you are aware that you'll need a type of object in your app, but you don't yet know what its functionalities will be.

Let's say you need you build a GUI. This means you'll need buttons, thus you'll make a `Button` class. However, you don't know how many types of buttons you'll need, like `RadioButton`, `CheckBox`, `ClickButton` etc. All that you know is that all these button have similar characteristics, as well as specific ones.

You can now encapsulate the existence of those similarities inside the `Button` class by using pure virtual functions. Now you cannot create a button per se, but you can provide the other button classes with the base methods they all need, such as `createButton`, `changePosition` or `addText` etc. Thus, pure abstract functions help from the point of view of the codes architecture.

15. ***Write short note on composition v/s classification hierarchies.***
16. ***What is the difference between base class and derived class? How data members and member functions of a base class can be accessed by derived class?***

Base Class**:** A base class is a class in Object-Oriented Programming language, from which other classes are derived. The class which inherits the base class has all members of a base class as well as can also have some additional properties. The Base class members and member functions are inherited to Object of the derived class. A base class is also called **parent class** or **superclass**.

Derived Class**:** A class that is created from an existing class. The derived class inherits all members and member functions of a base class. The derived class can have more functionality with respect to the Base class and can easily access the Base class. A Derived class is also called a **child class** or **subclass**.

Syntax for creating Derive Class:

```
class BaseClass{
  // members....
  // member function
}

class DerivedClass : public BaseClass{
  // members....
  // member function
}
```

Difference between Base Class and Derived Class:

S.No.	BASE CLASS	DERIVED CLASS
1.	A class from which properties are inherited.	A class from which is inherited from the base class.

2.	It is also known as parent class or superclass.	It is also known as child class subclass.
3.	It cannot inherit properties and methods of Derived Class.	It can inherit properties and methods of Base Class.
4.	**Syntax:** Class base_classname{ … }.	**Syntax:** Class derived_classname : access_mode base_class_name { … }.

A derived class can access all the non-private members of its base class. Thus base-class members that should not be accessible to the member functions of derived classes should be declared private in the base class.

We can summarize the different access types according to - who can access them in the following way −

Access	public	protected	private
Same class	yes	yes	yes
Derived classes	yes	yes	no
Outside classes	yes	no	no

A derived class inherits all base class methods with the following exceptions −

- Constructors, destructors and copy constructors of the base class.

- Overloaded operators of the base class.
- The friend functions of the base class.