
A complete note on,


Object oriented programming

*Along with the problems and solutions:
also available note on : Data Structure and Algorithms*

*By: Er. Bishwas Pokharel(Lecturer)
B.E- Pulchowk campus
MSc.Engg. -Pulchowk Campus
Published On: 2019/01/01*

1. General structure of C++ program

```
#include<iostream>           a
using namespace std;         b
int main(void)                c
{
    cout<<"hello world"<<endl;  d
    return 0;
}
```



a. **#include<iostream>**

This line is a preprocessing directive. All preprocessing directives within C++ source code begin with a # symbol. This one directs the preprocessor to add some predefined source code to our existing source code before the compiler begins to process it. This process is done automatically and is invisible to us.

2. Importance of namespace

b. **Using namespace std;**

The two items our program needs to display a message on the screen, **cout** and **endl**, have longer names: **std::cout** and **std::endl**. This **Using namespace std** directive allows us to omit the **std::** prefix and use their shorter names. name **std** stands for “standard,” and the Using namespace std line indicates that some of the names we use in our program are part of the so-called “standard namespace.”

Let us see examples about the importance of namespace:

```
#include<iostream>
using namespace std;
int main()
{
    int value=10;
    int value=100;
    cout<<value<<endl;
    cout<<value;
    return 0;
}
```

Output:

```
Message
In function 'int main()':
error: redeclaration of 'int value'
error: 'int value' previously declared here
```

This implies that you cannot declare same variable twice in same function

```
#include<iostream>
using namespace std;
int value=10;
int main()
{

    int value=100;
    cout<<value<<endl;
    cout<<value;
    return 0;
}
```

Output:

```
100
100
```

This implies that the value of the global variable is replaced by the value of the local variable.

Q. How can you declare and initialize the same variable in one program?

```
#include<iostream>
using namespace std;
namespace first
{
    int value=10;
}
int main()
{
    int value=100;
    cout<<first::value<<endl;
    cout<<value<<endl;
    return 0;
}
```

Output:

```
10
100
```

namespace allows the use of same variable in two different places but to access the name of variables of namespace you have to use the syntax as **name_of_namespace::variable_name** (first::value as in above example).

Now Observe,

```
#include<iostream>
using namespace std;
namespace first
{
    int value=10;
}
int main()
{
    cout<<first::value<<endl;
    return 0;
}
```

Output: 10

As you see to access the variable inside you have to use first:: value. Each time :: operator is necessary to access the variable value. But if you use using namespace first; you do not have to write first::value each time. Below example shows a clear clarification.

```
#include<iostream>
using namespace std;
namespace first
{
    int value=10;
}
using namespace first;
int main()
{
    cout<<value<<endl;
    return 0;
}
```

Output:10

Similarly, if you don't write using namespace std; then to access the member cout and cin of the namespace std, each time you have to write the syntax:- **name_of_namespace::variable_name** (std::cout, std::cin, std::endl etc) as shown in below..

```
#include<iostream>
namespace first
{
    int value=10;
}
using namespace first;
```

```

int main()
{
    std::cout<<value<<std::endl;
    return 0;
}

```

Output: 10

Finally, so **using namespace std;** allows to mention the member of the **namespace std** without using the syntax each time as:

```

#include<iostream>
namespace first{
int value=10;
}

using namespace first;
using namespace std;

int main()
{
    cout<<value<<endl;
    return 0;
}

```

Output: 10

Problem 1:

Define two namespaces: Square and Cube. In both the namespaces, define an integer variable named "num" and a function named "fun". The "fun" function in "Square" namespace, should return the square of an integer passed as an argument while the "fun" function in "Cube" namespace, should return the cube of an integer passed as an argument. In the main function, set the integer variables "num" of both the namespaces with different values. Then, compute and print the cube of the integer variable "num" of the "Square" namespace using the "fun" function of the "Cube" namespace and the square of the integer variable "num" of the "Cube" namespace using the "fun" function of the "Square" namespace.

Solution:

```

#include<iostream>
namespace Square
{
    int num;
    int func(int x)
    {

```

```

        return (x*x);
    }
}

namespace Cube
{
    int num;
    int func(int x)
    {
        return (x*x*x);
    }
}

using namespace std;
int main(void)
{
    cout<<"Enter two different values for num: "<<endl;
    cin>>Square::num;
    cin>>Cube::num;
    cout<<"The square of num used in namespace Cube:
"<<Square::func((Cube::num))<<endl;
    cout<<"The cube of num used in namespace Square:
"<<Cube::func((Square::num))<<endl;
    return 0;
}

```

Output:

```

Enter two different values for num:
2
3
The square of num used in namespace Cube: 9
The cube of num used in namespace Square: 8

```

- c. **int main(void):** This specifies the real beginning of our program. Here we are declaring a function named **main**.

Q. Why always main? Is it possible to use any other name besides main?

Ans: Compiler has instruction to search for keyword **main**. If compiler is instructed to search for other name as your choice besides **main** then the program must have you search name which you instruct to search for compiler. return value 0 is also not necessary; you can return any integer value. But compiler does not care about any value you return. So it is considered better to return 0.

d. `cout<<"hello world"<<endl`

The body of our **main** function contains only one statement. This statement directs the executing program to print the message **hello world** on the screen. A statement is the fundamental unit of execution in a C++ program. Functions contain statements that the compiler translates into executable machine language instructions. All statements in C++ end with a semicolon(;).

`printf()` in c but `cout` in C++

IN C	IN C++
<pre>#include<stdio.h> int main(void) { int a; printf("Enter number a: "); scanf("%d",&a); printf("The number is %d\n",a); return 0; }</pre>	<pre>#include<iostream> using namespace std; int main(void) { int a; cout<<"Enter the number a:"; cin>>a; cout<<"The number is: " <<a<<endl; return 0; }</pre>
Output: Enter number a: 1 The number is 1	Output: Enter number a: 1 The number is 1

`scanf()` in c but `cin>>` in C++

Note: `cout<<` is used to display on the screen as like `printf()` where as `cin>>` is used to input the value. `endl` is end line it is almost as “\n” in C.

3. Reference (Be careful it is not pointer’s address as you see as: `int *ptr=&a`).

```
int a=5;
int &r =a;
Here,
```


'r' is an alias of 'a' i.e. 'r' points on the same memory location as that of 'a' and holds the same value as that of 'a'.

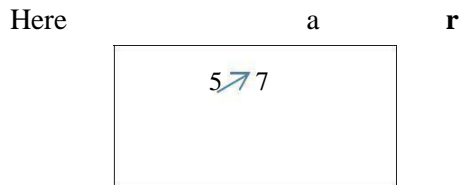
Here, `cout<<a<<endl; cout<<r<<endl;`

Both have same output 5. If value of r changed to 7 as `r=7`

`cout<<a<<endl;`

`cout<<r<<endl;`

Then again, both prints the same output `r=7`.



Q. Check to see whether it is correct or not?

```
#include<iostream>
using namespace std;
int main(void)
{
    int a;
    int &r=a;
    cout<<&a<<endl;
    cout<<&r<<endl;
    return 0;
}
```

Output:

```
0x68fee8
0x68fee8
```

Note: 'a' and 'r' have same memory address (you may have different output but both must be same value). It is like two men say 'A' and 'B' standing outside the door of same room. Suppose, A sees 5 men inside the room then B also sees 5 men. If in addition 2 men enter in room and B sees 7 then A also sees 7.

Q. what is its benefit?

Ans: Due to the creation of reference variable there is no need to create another variable and copy the value to newly created one. Its benefit can be observed when passing an argument to the function.

<p><u>Pass by Value</u></p> <pre> #include<iostream> using namespace std; void swap(int a,int b) { int temp; temp=a; a=b; b=temp; } int main(void) { int x=2,y=3; cout<<"Beforeswap: "<<endl; cout<<x<<" "<<y<<endl; swap(x,y); cout<<"After swap: "<<endl; cout<<x<<" "<<y; return 0; } </pre>	<p><u>Pass by Reference</u></p> <pre> #include<iostream> using namespace std; void swap(int& a,int& b) { int temp; temp=a; a=b; b=temp; } int main(void) { int x=2,y=3; cout<<"Beforeswap: "<<endl; cout<<x<<" "<<y<<endl; swap(x,y); cout<<"After swap: "<<endl; cout<<x<<" "<<y; return 0; } </pre>
---	---

Output:

Before swap: 2 3
After swap: 2 3

Before swap: 2 3
After swap: 3 2

In Pass by Value,

Analogy example: Suppose 'A' copy the text document from Bs laptop and change the text document in As own laptop. The changes occurs in the text document of A's laptop cannot be visible to B's document.

Similarly, Here value of x is copied in a and value of y is copied in b so change in one variable value does not affect the other.

In Pass by Reference,

Here, int&a=x and a=2alsoint &b=yand b=3. Afterswapa=3andb=2whichimplies x=3 and y=2 because both point to the same memory location.

4. Return by reference

<u>In Normal function</u>	<u>In return by reference</u>
<pre>#include<iostream> using namespace std; int x; int showx() { return x; } int main() { x=100; int b; b=showx(); cout<<b<<endl; showx()=20; //here it shows error cout<<b; return 0; }</pre>	<pre>#include<iostream> using namespace std; int x; int & showx() { return x; } int main() { x=100; int b; b=showx(); // normal function cout<<b<<endl; showx()=20;// return by ref. cout<x; return 0; }</pre>

Output: Error- lvalue case(look it)

Output:100
20

In return by reference, function which return reference variable is an alias for the referred variable as in case of (int & variable_name), in above case it is something like reference variable is created for global variable in statement “showx()=20”; so the value of the global variable change from 100 to 20.

Problem2: Write a function that passes two temperatures by reference and sets the larger of the two numbers to 100 by using return by reference.

Solution:

```
#include<iostream>
using namespace std;
int &temp(int &a,int &b)
{
    if(a>b)
    {
        return a;
    }
    else {
        return b;
    }
}
```

```

}
int main()
{
    int temp1,temp2;
    cout<<"enter the two value of the temperature"<<endl;
    cin>>temp1>>temp2;
    temp(temp1,temp2)=100;
    cout<<"New Temperature are : "<< temp1<<" and
"<<temp2<<endl;
    return 0;
}

```

Output:

```

enter the two value of the temperature
30
20
New Temperature are : 100 and 20

```

5. User Defined constant const:

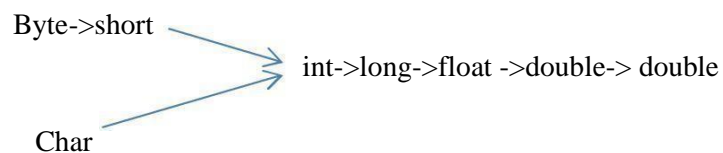
```

#include<iostream>
using namespace std;
int main(void)
{
    cout<<"Enter the radius of the circle:"<<endl;
    float radius;
    cin>>radius;
    const float PI=3.1416f;    // constant
    float area = PI*radius*radius;
    cout<<"The area of circle is: "<<area<<endl;
    return 0;
}

```

In C you used the syntax: #define PI 3.1416 but in C++ you can declare the constant value using
const+ data_type+ variable_name=value.

6. Promotion Rules



```

#include<iostream>
using namespace std;
void m1(char c)
{
    cout<<"char-args:"<<endl;
}
void m1(int i)
{
    cout<<"int- args:"<<endl;
}
void m1(int i,float f)
{
    cout<<"int- float-args:"<<endl;
}
void m1(float f,int i)
{
    cout<<"float-int-args:"<<endl;
}

int main()
{
    m1('c');
    m1(2);
    m1(3,2.0);
    m1(2.0,3);
    return 0;
}

```

Output:
char-args:
int-args:
int- float-args: float-int-args

But when you run the below program you may find an unexpected result due to the promotion rules.

```

#include<iostream>
using namespace std;
void m1(int i)

```

```

{
    cout<<"int- args:"<<endl;
}

void m1(int i,float f)
{
    cout<<"int- float-args:"<<endl;
}

int main()
{
    m1('c');
    m1(3,2.0);
    m1(2.0,3);
    return 0;
}

```

Output:

```

int- args:
int- float-args:
int- float-args:

```

Here, when function is called with the character argument (as `m1('c')`) it does not find the match so according to the rule “char” is promoted to “int”. Here it finds the match (`m1(int)`) and statement of it gets executed through `m1('c')`.

7. Conversion rules

```

#include<iostream>
using namespace std;
int main()
{
    int a=2,b=3; float c; c=a/b;
    cout<<"The value of c:"<<c<<endl;
    return 0;
}
Output
The value of c: 0

```

Here, the output must be 0.666667 but it shows 0. Why?

Ans: $(2/3) = (0.66667)$ but as default it takes only integer value and so 0 becomes output.

```

#include<iostream>
using namespace std;
int main()
{
    int a=2,b=3; float c;
    c=static_cast<float>(a)/b;
    cout<<"The value of c:"<<c<<endl;
    return 0;
}

```

Output

The value of c: 0.666667

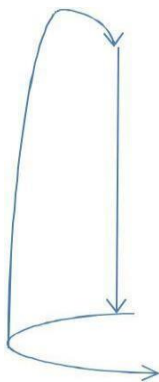
Here, static_cast is used for safe conversion which convert the calculation into the float value instead of int value.

8. Inline function

Inline keyword used in function is just a request to compiler to copy the function definition in a place, from where it is called rather than to transfer the control from one memory location to another. It is feasible only when the function definition has small segment of code.

Advantages

- It increases execution time of function.



```

#include<iostream>
using namespace std;
inline void add()
{
    int a,b,c;
    cout<<"Enter a" <<endl; cin>>a;
    cout<<"Enter b"<<endl;
    cin>>b; c=a+b;
    cout<<"c value "<<c <<endl;
}
int main(){
    add();
    cout <<"again call"<<endl;
    add(); return 0;
}

```

Let $T(mf)$ be the time taken to transfer control from main () function to void add (). $T(fe)$ be the total time taken to execute the statement inside function. $T(fm)$ be the time taken to return control from void add () to main () function. Let, total time taken = $T(mf) + T(fe) + T(fm)$. Suppose, function void add () used 64 KB memory

space. If function is written at the place from where it's called then total time taken
= T (fe).

If again same function is necessary somewhere in the program then again 64 KB memory is consumed by it if function is written without calling it. But function call saves next 64 KB of memory space. Here in such scenario, time taken to execute function increases but it increases memory efficiency i.e. memory space consumed by it is reduced.

But this scenario is feasible only if the function statement in program is very long. But if program is short then it consumes small fraction of memory space which is not considered big problem, If program is written every time when necessary.

Here time is in big important so programmer used inline keyword to reduce time taken to execute function by concept as: when function is necessary in program just copy function definition in place from where it's called rather than to transfer control. Inline is just a request to the compiler, and compiler determines whether to accept the request or to reject the request.

9.Function Overloading

Function overloading occurs in the program when number of function has the same name but different type of arguments.

Advantages:

- Having same name for all function reduces ambiguity because it is easy to remember the name of function.

```
#include<iostream>
using namespace std;
void m1(int i)
{
    cout<<"Int-args:"<<endl;
}
void m1(float f)
{
    cout<<"Float-args:"<<endl;
}
int main()
{
    m1(2);
    m1(2.3f);
    return 0;
}
```


Output:**Int-args:****Float-args:**

Here, both functions have same name but different arguments. When integer is passed then “Int- args:” is executed and when float is passed then “Float-args” is executed.

Problem 3.

Write a program using the function overloading that converts feet to inches. Use function with no argument, one argument and two arguments. Decide yourself the types of arguments. Use pass by reference in any one of the Function above.

Solution:

```
#include<iostream>
using namespace std;
void feetToinch()
{
    cout<<"Enter the feet_value: ";
    float feet_value;
    cin>>feet_value;
    float inch_value= feet_value*12;
    cout<<"corresponding inch_value: "<<inch_value<<endl;
}

void feetToinch(float &inch_value)
{
    inch_value= inch_value*12;
}

void feetToinch(float feet_value, float conversion_value)
{
    float inch_value= feet_value*conversion_value;
    cout<<"corresponding inch_value: "<<inch_value<<endl;
}
int main()
{
    cout<<"Enter 0 for no argument, 1 for one argument and 2 for
two argument: "<<endl;
    int choice_num;
    cin>>choice_num;
    if(choice_num==0)
    {
        feetToinch();
    }
    else if(choice_num==1)
    {
        cout<<"Enter the feet_value: ";
        float feet_value;
        cin>>feet_value;
        feetToinch(feet_value);
    }
}
```

```

        cout<<"corresponding inch_value: "<<feet_value<<endl;

    }else if(choice_num==2)
    {
        cout<<"Enter the feet_value: ";
        float feet_value;
        cin>>feet_value;
        feetToinch(feet_value,12.0f);
    }else
    {
        cout<<"Enter valid number: "<<endl;
    }
}
return 0;
}

```

10.Default arguments

Consider a function for printing an integer. Giving the user an option of what base to print it in seems reasonable, but in most programs integers will be printed as decimal integer values. For example:

```

#include<iostream>
using namespace std;
void print(int value, int base=10)
{
    if(base==10)
    {
        cout<<"Equivalent decimal value of decimal "<<value<<" is
: "<<value<<endl;
    }
    else if(base==8)
    {
        int old_value=value;
        int oct_num[10],i=1;
        while(value!=0)
        {
            oct_num[i++]=value%8;
            value=value/8;
        }
        cout<<"Equivalent octal value of decimal "<<old_value<<" is :";
        for(int j=i-1; j>0; j--)
        {
            cout<<oct_num[j];
        }
        cout<<endl;
    }
}
int main(void)
{
    print(10);
    print(10,10);
    print(50,8);
    return 0;
}

```

Output:

```
Equivalent decimal value of decimal 10 is :10
Equivalent decimal value of decimal 10 is :10
Equivalent octal value of decimal 50 is :62
```

In above program, if user do not pass any information about the base value then by default it assumes that the base is decimal as (int base=10) else it is replaced by the argument. When print(50,8) is passed in the function then base is taken as 8 in place of 10. So it gives the octal value of the 50 as 62.

Note: default value must be assigned from the right side.

```
Ex: void sum(int a=2,int b) // error
    void sum(int a=2,int b,int c=12) // error
    void add(int a, int b=1, int d=11) //valid
```

Example:

```
#include<iostream>
using namespace std;
void add(int a, int b=10)
{
    int c=a+b;
    cout<<"value of c is:"<<c<<endl;
}
int main()
{
    int a=2,b=3;
    add(a);
    add(a,b);
    return 0;
}
```

Output:

```
value of c is:12
value of c is:5
```

Here, add (a) has only one value passed in function. Here argument int b =10 tells the compiler that if no any value is passed in me than I considered default value 10 as my argument otherwise I take the value which is passed for me. It is the concept of default argument.

11. Features of C++

- a. **Namespace:** It provides an encapsulation i.e member of namespace can be used without knowing any necessary details about the member.
- b. **Function Overloading:** It improves the readability of code.
- c. **Inline function:** It speeds up execution time for small code.
- d. **Reference variable:** It works for the memory efficiency, no need to create new variable and copy value to it.
- e. **Default argument:** It provides flexibility.
- f. **Generics:** It provides reusability and more powerful features through template.
- g. **Object oriented:** It prefers creation of objects for large program to handle data with high security.

12. C++ vs C

- a. More Libraries in C++ than C
- b. In C++, less reliance on preprocessor i.e #define is replaced by const
- c. More type safe features than C
- d. In C++, declaration can be done anywhere but the freedom is restricted in C.
- e. C uses pointer to swap and focused in pointer especially where as C++ prefer concept of reference variable.
- f. C++ focus on Object oriented where as C is imperative.

13.Function Template:

For example, assume you need a simple function named reverse() that reverses the sign of a number. Figure below shows three overloaded versions of the reverse() function. Each has a different parameter list so the function can work with integers, doubles, or floats.

```
#include<iostream>
using namespace std;
int reverse(int x)
{
    return -x;
}
float reverse(float x)
{
    return -x;
}
double reverse(double x)
{
    return -x;
}
int main(){
    int i=reverse(3);
    cout<<i<<endl;
    float f= reverse(3.2f);
    cout<<f<<endl;
    double d= reverse(3.0);
    cout<<d<<endl;
    return 0;
}
```

Output:

```
-3
-3.2
3.0
```

The three function bodies in are identical. Because these functions differ only in the parameter and return types involved, it would be convenient to write just one function with a variable name standing in for the type which can hold all types of data types and return all of it as required as,

```
#include<iostream>
using namespace std;
template <class T>
T reverse(T x)
{
    return -x;
}
int main()
{
```

```

    int i=reverse(3);
    cout<<i<<endl;
    float f= reverse(3.2f);
    cout<<f<<endl;
    double d= reverse(3.0);
    cout<<d<<endl;
    return 0;
}

```

Output:

```

3
-3.2
3.0

```

Before you code a function template, you must include a template definition with the following information:

- » the keyword **template**
- » a left angle bracket (<)
- » a list of generic types, separated with commas if more than one type is needed
- » a right angle bracket (>)

Each generic type in the list of generic types has two parts:

- » the keyword **class**
- » an identifier that represents the generic type

When you call the reverse() function template, as in the following code, the compiler determines the type of the actual argument passed to the function (in this case, a double).
double amount = -9.86; amount = reverse(amount)

The compiler substitutes the argument type (a double in this example) for the generic type in the function template, creating a generated function that uses the appropriate type. The designation of the parameterized type is implicit; that is, it is determined not by naming the type, but by the compiler's ability to determine the argument type. The compiler generates code for as many different functions as it needs, depending on the function calls that are made. In this case, the generated function is:

```

double reverse(double x)
{
    return -x;
}

```

You can place function templates at the start of a program file, in the global area above the main() function. Alternatively, you can place them in a separate file, and include that file in your program with an #include statement. In either case, the definition of the class template (the statement that contains the word template and the angle brackets holding the class name) or the function template itself must reside in the same source file. When

you create a function template, you can't place the template definition and the function code in separate files because the function can't be compiled into object format without knowing the types, and the function won't "know" that it needs to recognize the types without the template definition.

Note: In below program, compiler generates an error because T does not hold different data types. One class of template hold only one datatype i.e template<class T> holds either all int or all float(double) but not the both under T

```
#include<iostream>
using namespace std;
template <class T,class U>
void invert(T &x, T &y)      // T must be same type
{
    cout<<x<<" "<<y;
}
int main()
{
    int a=2;
    double b=3.3;
    invert(a,b);             // invert(int, double)
    return 0;
}
```

Correction is:

```
#include<iostream>
using namespace std;
template <class T,class U>
void invert(T &x, U &y) // T and U are different templates
{
    cout<<x<<"+"<<y;
}
int main()
{
    int a=2;
    double b=3.3;
    invert(a,b);           // invert(int, double)
    return 0;
}
```

Output:
2 + 3.3

T holds one data types and U holds another data types so produces output.

Problem 4: Write a program that will find the sum and average of elements in an array using Function templates.

```
#include<iostream>
using namespace std;
template <class T>
void arraySum(T a[], int size_of_array)
{ T sum=0;
  for(int i=0;i<size_of_array;i++)
  {
    sum=sum+a[i];
  }
  T avg=static_cast<float>(sum)/size_of_array;
  cout<<"sum of elements: "<<sum<<endl;
  cout<<"avg of sum of elements: "<<avg;
}
int main()
{
  cout<<"How many elements do you want to add in an array ? ";
  int size_of_array;
  cin>>size_of_array;

  int a[size_of_array];
  for(int i=0;i<size_of_array;i++)
  {
    cin>>a[i];
  }

  arraySum(a,size_of_array);
  cout<<endl;
  cout<<"How many elements do you want to add in an array ? ";
  int size_of_array1;
  cin>>size_of_array1;
  float b[size_of_array1];
  for(int i=0;i<size_of_array1;i++)
  {
    cin>>b[i];
  }
  arraySum(b,size_of_array1);

  return 0;
}
```

Output:

```
How many elements do you want to add in an array ? 3
2
32
3
sum of elements: 37
avg of sum of elements: 12

How many elements do you want to add in an array ? 2
2.22
4.33
sum of elements: 6.55
avg of sum of elements: 3.275
```


Also,

```
#include<iostream>
using namespace std;
template <class T>
void arraySum(T a[], int size_of_array)
{ T sum=0;
  for(int i=0;i<size_of_array;i++)
  {
    sum=sum+a[i];
  }
  T avg=static_cast<double>(sum)/size_of_array;
  cout<<"sum of elements: "<<sum<<endl;
  cout<<"avg of sum of elements: "<<avg<<endl;
}
int main()
{
  int a[]={1,2,3};
  int n1 = sizeof(a)/sizeof(a[0]);

  double b[]={1.2,2.3,3.4};
  int n2 = sizeof(b)/sizeof(b[0]);

  arraySum<int>(a,n1);
  cout<<endl;
  arraySum<double>(b,n2);
  return 0;
}
```

Output:

```
sum of elements: 6
avg of sum of elements: 2

sum of elements: 6.9
avg of sum of elements: 2.3
```

14. USING MULTIPLE PARAMETERS IN FUNCTION TEMPLATES:

Function templates can support multiple parameters. You might, for example, write a function that compares three parameters and returns the largest of the three. Below example shows the function template named findLargest(), which accomplishes this task.

```
#include<iostream>
using namespace std;
template <class T>
T findLargest(T x,T y,T z) //multiple parameters in function with T
{
    T big;
    if(x>y)
    {
        big=x;
    }
    else
    {
        big=y;
    }
    if(z>big)
    {
        big=z;
    }
    return big;
}
int main()
{
    cout<<findLargest(2,4,5)<<endl;
    cout<<findLargest(2.3,4.6,5.9)<<endl;
    cout<<findLargest(3.2f,1.2f,2.2f);
    return 0;
}
```

Output:

```
5
5.9
3.2
```

The findLargest() function takes three parameters of type T (whatever type T is). Because all three parameters are the same type, the compiler can create object code for findLargest(int, int, int) and findLargest(double, double, double), but it will not compile findLargest(int, double, double) or findLargest(double, int, int), or any other combination in which the parameter types are not the same. For example, findLargest(3, 5, 9); returns 9, and findLargest(12.3, 5.7, 2.1); returns 12.3. However, findLargest(3, 5.7, 9); produces a compiler error because the parameter types do not match. Wherever T occurs in the function template, T must stand for the same type.

But it's possible as:

```

#include<iostream>
using namespace std;
template <class T>
T check(T x,T y,float z)
{
    return (x+y+static_cast<int>(z));    // float along with template T
}
int main()
{
    cout<<check(2,4,5.8)<<endl;
    return 0;
}

```

Output:

11

15. OVERLOADING FUNCTION TEMPLATES:

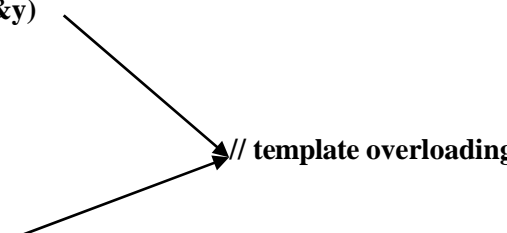
You overload functions when you create functions with the same name but with different parameter lists. Likewise, you can overload function templates only when each version of the function takes a different parameter list, allowing the compiler to distinguish between the functions. For example, you can create an invert() function template that swaps the values of its parameters if it receives two parameters, but reverses the sign if it receives only one parameter.

```

#include<iostream>
using namespace std;
template <class T>
void invert(T &x, T &y)
{
    T temp;
    temp=x;
    x=y;
    y=temp;
}
template <class T>
void invert(T &x)
{
    x=-x;
}

int main()
{
    int a=2,b=3;
    cout<<"Before "<<"a: "<<a<<" b: "<<b<<endl;
    invert(a,b);
}

```



```
        cout<<"After "<<"a: "<<a<<" b: "<<b<<endl;

        int x=5;
        cout<<"Before "<<"x: "<<x<<endl;
        invert(x);
        cout<<"After "<<"x: "<<x;
        return 0;
    }
```

Output:

```
Before a: 2 b: 3
After a: 3 b: 2
Before x: 5
After x: -5
```

16. Class and Objects:

Imagine, In every manufacture company like car company, part company etc before manufacture it, designing is necessary and it done by drawing or any other modeling software.

Suppose, you are engineer and according to given specifications you design the car in chart paper. The drawing of car is just the blueprint, you cannot ride in that car and not able to perform any real operation directly on that paper like: opening the door, braking and speeding up etc.

To perform all the operation you have to make the car from that blueprint. And using that real object you can perform all the function. Real car then occupied space in area. With the help of the same blueprint you can create number of car having different color, weight, behavior etc.

Like as:

Class is same as the blueprint from which programmer may produce any number of an objects. Whereas an **Object** is the instance of the class which works on the behavior defined inside the class.

Q. Why there is the need of class?

Consider the task of dealing with the geometric points. In mathematics, a single point is represented with an ordered pair of real numbers, usually expressed as (x1, y1). A single variable cannot stores two values so it requires two variables to represent a single point, like as

```
double x=x1;
```

```
double y=y1;
```

Ideally we should be able to use one variable to represent point and it can be done with two dimensional vectors.

```
vector<double> pt={x1,x2}
```

But it has many disadvantages:

1. We cannot restrict the vector's size to two. A programmer may accidentally push extra items onto the back of a vector representing a point object.
2. We must use numeric indices instead of names to distinguish between the two components of a point object. We may agree that pt[0] means the x coordinate of point pt and pt[1] means the y coordinate of point pt, but the compiler is powerless to detect the error if a programmer uses an expression like pt[19] or pt[-3].
3. We cannot use a vector to represent objects in general. Consider a bank account object. A bank account object could include, among many other diverse things, an account number (an integer), a customer name (a string), and an interest rate (a double-precision floating-point number). A vector implementation of such an object is impossible because the elements in a vector must all be of the same type.

Syntax

```
class class_name
{
    private:
    //statement

    public:

    //statement

};

Class_name object_name;
Object_name.statement;
```

- a. Class is user defined data types. You are free to choose the class_name.
- b. Most important data with data type (int , float, double,string etc) are placed under private for security reason.
- c. Almost all functions are placed under public which are able to access the private members.
- d. Data are called data members while function are called member function in class.

Ex: **class Addaccount**

```
{
    private:
        double balance;
        string name;
    public:
        void getdata()
        { //statement
        }
};
```

- e. Semicolon must be there to end the class.
- f. Class is defined above the main() function where as object is defined inside the main function in most of the cases.

Ex:

```
int main()
{
    Addaccount account;
    Account.getdata();
    return 0;
}
```

- g. account is object name which is written after class name and to access the member function always have to write objectname.name_of_member function.

Ex:

Public case: <pre>#include<iostream> using namespace std; class Point { public: int x; int y; }; int main(void) { Point p1, p2; p1.x=4; p1.y=3; p2.x=1; p2.y=2; cout<<p1.x<<" "<<p1.y<<endl; cout<<p2.x<<" "<<p2.y<<endl; return 0; }</pre> Output: 4, 3 1, 2	Private case: <pre>#include<iostream> using namespace std; class Point { private: int x; int y; }; int main(void) { Point p1, p2; p1.x=4; p1.y=3; p2.x=1; p2.y=2; cout<<p1.x<<" "<<p1.y<<endl; cout<<p2.x<<" "<<p2.y<<endl; return 0; }</pre> Output: Error Int Point:: x is private.
---	---

In public case,

Data and function members can be access from outside the class (from main function). It's like anybody can have authority to use the public places.

In private case,

Only member functions (function inside the class besides friend function) can accessed the private data members. It's like only family member have authority to access the private property of house.

If anybody from outside the member tries to access the private property of house an objection is made because it is like the robbery. So to access the private member first the public member should be access from outside and with the help of public member private member can be accessed.

```

#include<iostream>
using namespace std;
class Point
{
private:
    int x;
    int y;

public:
    void getAccess()
    {
        x=4;
        y=3;
        cout<<x<<" "<<y<<endl;
    }
};

int main(void)
{
    Point p1, p2;
    p1.getAccess();
    p2.getAccess();
    return 0;
}

```

Output:

```

4, 3
4, 3

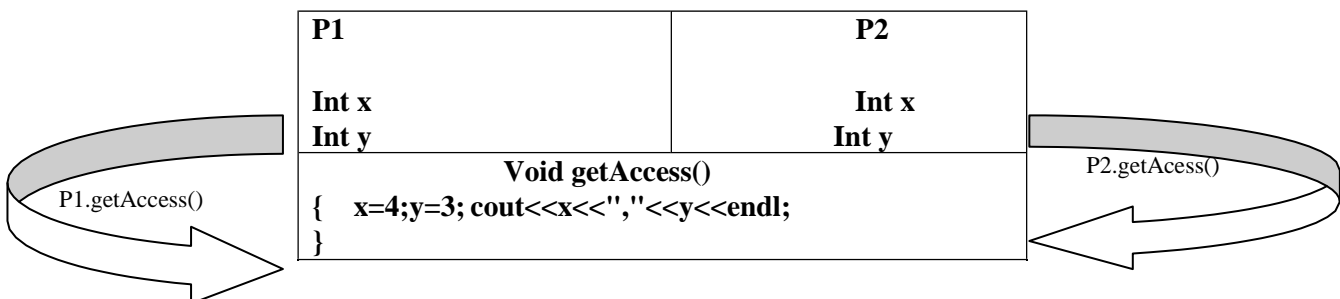
```

Explanation:

1. At the time of creating an object, only the private member is carried by itself ie both p1 and p2 have private data member x and y at the time of their creation. Each object has the separate set of data.

P1	P2
Int x Int y	Int x Int y

2. But objects share the public member function



3. **p1.getAccess();**

P1	P2
Int x=4 Int y=3	Int x Int y

Till now only p1 object call the getAccess() where it's private member is initialized as 4 and 3. And output can be seen 4,3.

4. **P2.getAccess();**

P1	P2
Int x=4 Int y=3	Int x =4 Int y=3

Till now only p1 object call the getAccess() where it's private member is initialized as 4 and 3. Both values are set so complete output is

4, 3
4, 3

Problem5:

Write a program with class to represent circle. Class should have data member radius. Find perimeter and area of the object. Use the class to create objects.

Method I: Direct input the value of private member i.e radius

```
#include<iostream>
using namespace std;
double const PI=3.1416;

class Circle
{
private:
double radius;
public:
void areaCircle()
{
cout<<"Enter the value of radius: "<<endl;
cin>>radius;
double area= PI*radius*radius;
double perimeter=2*PI*radius;
cout<<"Area of circle is: "<<area<<endl;
cout<<"Perimeter of circle is: "<<perimeter<<endl;
}
};
```

```

int main(void)
{
    Circle crc;
    crc.areaCircle();
    return 0;
}

```

Output:

```

Enter the value of radius:
3
Area of circle is: 28.2744
Perimeter of circle is: 18.8496

```

Method II: Setting the radius value through function and using radius.

```

#include<iostream>
using namespace std;
double const PI=3.1416;

class Circle
{
    private:
        double radius;
    public:
        void setRadius(double r)
        {
            radius=r;
        }
        void areaCircle()
        {
            double area= PI*radius*radius;
            double perimeter=2*PI*radius;
            cout<<"Area of circle is: "<<area<<endl;
            cout<<"Perimeter of circle is: "<<perimeter<<endl;
        }
};

int main(void)
{
    Circle crc;
    cout<<"Enter the value of radius: "<<endl;
    double rad;
    cin>>rad;
    crc.setRadius(rad);
    crc.areaCircle();
}

```

```
        return 0;
    }
}
```

Output:

```
Enter the value of radius:
3
Area of circle is: 28.2744
Perimeter of circle is: 18.8496
```

In method II, first from main() function value is input, this value is passed to the setRadius(int r) and radius =r puts the value of r in radius. Now area and perimeter is calculate using the formula Where areaCircle() is called using objectname.member function as crc.areaCricle(rad).

I prefer **method II**, find reason by yourself if interested.

Problem 6: Write a program with classes to represent circle, rectangle and triangle. Class should have their required data member. Find perimeter and area of the object. Use the class to create objects.

```
#include<iostream>
#include<math.h>
using namespace std;
double const PI=3.1416;

class Circle
{
    private:
        double radius;
    public:
        void setRadius(double r)
        {
            radius=r;
        }
        void areaCircle()
        {
            double area= PI*radius*radius;
            double perimeter=2*PI*radius;
            cout<<"Area of circle is: "<<area<<endl;
            cout<<"Perimeter of circle is: "<<perimeter<<endl;
        }
};

class Rectangle
{
    private:
        double lenght;
```

```

        double breadth;
public:
    void setParameter(double l,double b)
    {
        lenght=l;
        breadth=b;
    }
    void areaRectangle()
    {

        double area= lenght*breadth;
        double perimeter=2*(lenght+breadth);
        cout<<"Area of rectangle is: "<<area<<endl;
        cout<<"Perimeter of rectangle is: "<<perimeter<<endl<<endl;
    }
};

class Triangle
{
private:
    double a,b,c;
public:
    void setParameter(double l,double m,double n)
    {
        a=l;
        b=m;
        c=n;
    }
    void areaRectangle()
    {
        double s=(a+b+c)/2;
        double area=pow((s*(s-a)*(s-b)*(s-c)),0.5);
        double perimeter= ((a+b+c));
        cout<<"Area of rectangle is: "<<area<<endl;
        cout<<"Perimeter of rectangle is: "<<perimeter<<endl<<endl;
    }
};

int main(void)
{
    Circle crc;
    cout<<"Enter the value of radius: "<<endl;
    double rad;
    cin>>rad;
    crc.setRadius(rad);
    crc.areaCircle();

    Rectangle rec;
    cout<<endl<<"Enter the value of length and breadth: "<<endl;

```

```

double len,breth;
cin>>len>>breth;
rec.setParameter(len,breth);
rec.areaRectangle();

Triangle tri;
cout<<endl<<"Enter the value of three sides of triangle: "<<endl;
double l,m,n;
cin>>l>>m>>n;
tri.setParameter(l,m,n);
tri.areaRectangle();

return 0;
}

```

Output:

```

Enter the value of radius:
2
Area of circle is: 12.5664
Perimeter of circle is: 12.5664

Enter the value of length and breadth:
2
3
Area of rectangle is: 6
Perimeter of rectangle is: 10

Enter the value of three sides of triangle:
3
4
5
Area of rectangle is: 6
Perimeter of rectangle is: 12

```

17. Class Templates

Function templates allow you to create generic functions that have the same bodies but can take different data types as parameters. Likewise, in some situations classes are similar and you want to perform very similar operations with them. If you need to create several similar classes, you might consider developing a class template to generate a class in which at least one type is generic or parameterized. The class template provides the outline for a family of similar classes.

Note::To create a class template, you begin with the template definition, just as you do with a function template. Then you write the class definition using the generic type or

types in each instance for which a substitution should take place.

```
#include<iostream>
using namespace std;
template<class T>
class Number
{
    private:
        T number;
    public:
        Number(T a)
        {
            number=a;
        }
    void display()
    {
        cout<<"The number is: "<<number<<endl;
    }
};
int main(void)
{
    Number<int>n(2);
    Number<double>n1(2.3);
    n.display();
    n1.display();
    return 0;
}
```

Output:

```
The Number is: 2
The Number is: 2.3
```

Within a program, you can instantiate objects that possess the class template type. You add the desired type to the current class instantiation by placing the type's name between angle brackets following the generic class name. For example, if you want an object named **n** to be of type Number, and you want **n** to hold an integer with value 2 (that is, you want to pass 2 to the constructor), your declaration is:

```
Number<int> n(2);
```

To use the Number class member display()function with the n object, you add the dot operator, just as you would with an instantiation of any other class:

```
n.display();
```

If you instantiate another Number object with a double, the display()function is called in exactly the same way:

```
Number<double> n1(2.3);
n1.display();
```

An object of type Number might “really” be an int, double, or any other type behind the scenes. The advantage of using a class template to work with these values lies in your ability to use the Number class functions with data of any type. When you need to write similar functions to display or use data of several different types, it makes sense to create

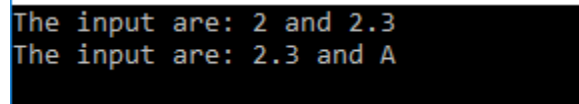
a class template.

Q. Can there be more than one argument to templates?

Yes, like normal parameters, we can pass more than one data types as arguments to templates. The following example demonstrates the same.

```
#include<iostream>
using namespace std;
template<class T,class U>
class Number
{
    private:
        T FirstInput;
        U SecondInput;
    public:
        Number(T a, U b)
        {
            FirstInput=a;
            SecondInput=b;
        }
    void display()
    {
        cout<<"The input are: "<<FirstInput<<" and "<<SecondInput<<endl;
    }
};
int main(void)
{
    Number<int,double>n1(2,2.3);
    Number<double,char>n2(2.3,'A');
    n1.display();
    n2.display();
    return 0;
}
```

Output:



```
The input are: 2 and 2.3
The input are: 2.3 and A
```

Q. Can we specify default value for template arguments?

Yes, like normal parameters, we can specify default arguments to templates. The following example demonstrates the same.

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

```

template<class T,class U>
class Number
{
    private:
        T FirstInput;
        U SecondInput;
    public:
        Number(T a, U b)
        {
            FirstInput=a;
            SecondInput=b;
        }
    void display()
    {
        cout<<"The input are: "<<FirstInput<<" and "<<SecondInput<<endl;
    }
};
int main(void)
{
    Number<int,double>n1(2,2.3);
    Number<double>n2(2.3,'A');    //only one argument "double" is passed
    n1.display();
    n2.display();
    return 0;
}

```

Output: The following program shows the error as “Wrong number of template arguments”

Correction:

```

#include<iostream>
using namespace std;
template<class T,class U=char>    //use default argument
class Number
{
    private:
        T FirstInput;
        U SecondInput;
    public:
        Number(T a, U b)
        {
            FirstInput=a;
            SecondInput=b;
        }
    void display()

```



```

{
    cout<<"The input are: "<<FirstInput<<" and "<<SecondInput<<endl;
}
};
int main(void)
{
    Number<int,double>n1(2,2.3);
    Number<double>n2(2.3,'A'); //one argument but other by default char
    n1.display();
    n2.display();
    return 0;
}

```

Output:

```

The input are: 2 and 2.3
The input are: 2.3 and A

```

18.Constructor

Sometimes a programmer can forget to initialize the class object which yields to the unexpected result as:

```

#include<iostream>
using namespace std;
class Construct
{
    private:
        float real;
        float img;
    public:
        void setdata()
        {
            float add=real+img;
            cout<<add;
        }
};
int main(void)
{
    Construct c ;
    c.setdata();
    return 0;
}

```

Output:

```

Sum is: 1.56484e-038

```

A better approach is to allow the programmer to declare a function with the explicit purpose of initializing objects. Because such a function constructs values of a given type, it is called a constructor. A constructor is recognized by having the same name as the class itself.

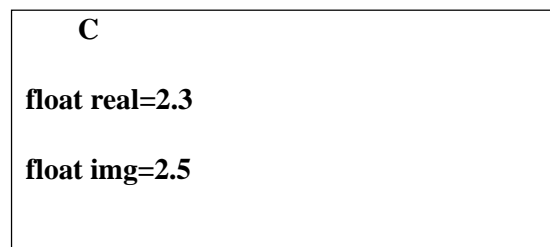
```
#include<iostream>
using namespace std;
class Construct
{
    private:
        float real;
        float img;
    public:
        Construct(float x, float y) //constructor, class name
        {                          //initialization of object
            real=x;
            img=y;
        }
        void setdata()
        {
            float add=real+img;
            cout<<"Sum is: " <<add;
        }
};
int main(void)
{
    Construct C(2.3,2.5); //value passed at the time of object
    C.setdata();          //creation
    return 0;
}
```

Output:

```
Sum is: 4.8
```

Explanation:

At the time of creating an object, only the private member is carried by itself i.e an object **C** has the private data member **real** and **img** at the time of its creation and through the constructor they are initialized at the same time of its creation i.e the data member does not wait for programmer to set values.



19.Types Of Constructor:

a. Default constructor:

Default constructor is one in which no argument is passed at the time of object creation. It has no parameter/s.

Syntax:

```
class class_name
{
public:
class_name()    // default constructor

{
// content
}

};
```

```
#include<iostream>
using namespace std;
class Construct
{
private:
float real;
float img;
public:
Construct() // Default Constructor
{
real=10;
img=20;
}
void setdata()
{
float add=real+img;
cout<<"Sum is: " <<add;
}
};
int main(void)
{
Construct C;
C.setdata();
return 0;
}
```

Output: Sum is: 30

b. Parameterized constructor

Parameterized constructor is one in which an argument is passed at the time of object creation. It has parameter/s.

Syntax:

```
class class_name

{
public:
class_name(data_type variable_name)    // parameterized constructor
{
// content
}

};
```

```
#include<iostream>
using namespace std;
class Construct
{
private:
float real;
float img;
public:
Construct(float x, float y) //parameterized constructor
{
real=x;
img=y;
}
void setdata()
{
float add=real+img;
cout<<"Sum is: " <<add;
}
};
int main(void)
{
Construct C(2.3,2.5);
C.setdata();
return 0;
}
```

Output: Sum is: 4.8

c. Copy constructor:

Copy constructor is called when one object is initialized using another object of same class.

Syntax:

```
ClassName (const ClassName &old_obj);
```

```
#include<iostream>
using namespace std;
class Construct
{
    private:
        float real;
        float img;
    public:
        Construct(float x, float y)    //parameterized constructor
        {
            real=x;
            img=y;
        }
        Construct (const Construct &C)  //copy constructor
        {
            real=C.real;
            img=C.img;
        }
        void displayData()
        {
            cout<<"real: " <<real<<" img "<<img<<endl;
        }
};
int main(void)
{
    Construct C(2.3,2.5);
    Construct C1=C;
    C.displayData();
    C1.displayData();
    return 0;
}
```

Output:

```
real: 2.3 img: 2.5
real: 2.3 img: 2.5
```

20. Destructor:

Destructor is a special class function which destroys the object as soon as the scope of object ends. The destructor is called automatically by the compiler when the object goes out of scope. The syntax for destructor is same as that for the constructor, the class name is used for the name of destructor, with a tilde ~ sign as prefix to it.

```
class A
{
public:
~A();
};
```

Destructors will never have any arguments.

```
#include<iostream>
using namespace std;
class A
{
private:
    static int i;
public:
    A()
    {
        i++;
        cout << "Constructor called for "<<i<<endl;
    }
    public:
    ~A()
    {
        cout << "Destructor called for "<<i<<endl;
        i--;
    }
};
int A::i=0;
int main()
{

    A obj1;
    // Constructor Called int x=1;
    int x=1;

    if(x)
    {
        A obj2; // Constructor Called
    } // Destructor Called for obj2
} // Destructor called for obj1
```

Output:

Constructor called for 1
Constructor called for 2
Destructor called for 2
Destructor called for 1

21. Object as Function argument:

Problem 7: WAP to add two point of complex number by passing object as arguments.
(Note: by default you have to use constructor concept as well).

```
#include<iostream>
using namespace std;
class Complex
{
private:
    float x,y;
public:
    Complex()
    {
        x=0;
        y=0;
    }
    Complex(float a, float b)
    {
        x=a;
        y=b;
    }

    void Add(Complex c1, Complex c2) //object as parameters
    {
        Complex c3;
        c3.x=c1.x+c2.x;
        c3.y=c1.y+c2.y;
        cout<< c3.x <<" +j " <<c3.y;
    }
};
int main(void)
{
    Complex c1(2.2,4.5),c2(3.4,5.9);
    c1.Add(c1,c2); // passing object as argument
    return 0;
}
```

Output:

5.6+j10.4

Explanation:

- a. **Passing object as arguments:** c1 and c2 are two objects which are initialized at the time of their creation and the object is passed as argument through the function c1 and c2.
- b. **Object as parameters:** The objects values are copied in the parameter objects and two values are added, and store in object c3 and display it.

22. Return object Type:

Problem 8: WAP to add two point of complex number by passing object as arguments and returning object as an argument. (Note: by default you have to use constructor concept as well).

```
#include<iostream>
using namespace std;
class Complex
{
private:
    float x,y;
public:
    Complex()
    {
        x=0;
        y=0;
    };
    Complex(int a, int b):x(a),y(b){}

    Complex Add(Complex c1,Complex c2) //return object
    {
        Complex c3;
        c3.x=x+c2.x;
        c3.y=y+c2.y;
        return c3;
    }
    void show(Complex c4)
    {
        cout<<c4.x <<" +j " <<c4.y;
    }
};
int main(void)
{
    Complex c1(2,4),c2(3,5);
    Complex c3=c1.Add(c1, c2); // object is return, result
    c3.show(c3);               //are assign in c3 and display
    return 0;
}
```


Output:

5 + j 9

Explanation:

- Object has data_type as class_name. When returning an object we have to replace void by data_type of object and is class_name. So in above program Complex is data type of object so **Complex Add(complex, Complex)** is used as return data_type.
- To accept the return object **Complex c3=c1.Add(c1, c2);** syntax is written ,here the concept of copy constructor is used.
- Finally, the values are displayed by passing object as an argument.

Problem 9: WAP that computes the sum of complex number using templates (verify the program for int and double, real and img values) by passing object as an argument and return object as an argument.

```
#include<iostream>
using namespace std;
template<class T>
class Complex
{
    private:
        T real;
        T img;
    public:
        Complex()
        {

        }
        Complex(T a, T b)
        {
            real=a;
            img=b;
        }
        Complex addComplex(Complex const &c1, Complex const &c2)
        {
            Complex c3;
            c3.real=c1.real+c2.real;
            c3.img=c1.img+c2.img;
            return c3;
        }
        void displayComplex(Complex const &c3)
        {
            cout<<"("<<c3.real<<" +j"<<c3.img<<" )"<<endl<<endl;
        }
};
int main(void)
```

```

{
//Double number case
Complex<double>n1(2.3,4.3);
Complex<double>n2(3.2,4.6);
Complex<double>n3;
n3=n1.addComplex(n1,n2);
n3.displayComplex(n3);

//Interger number case
Complex<int>n4(2,4);
Complex<int>n5(3,4);
Complex<int>n6;
n6=n4.addComplex(n4,n5);
n6.displayComplex(n6);

return 0;
}

```

Output:

```

(5.5+j8.9)
(5+j8)

```

23.Array of objects:

An array of a class type is also known as an array of objects. An array of objects is declared in the same way as an array of any built-in data type. The syntax for declaring an array of objects is:

class_name object_name [size] ;

```

#include <iostream>
using namespace std;
class MyClass
{
    private:
    int x;
    public:
    void setX(int i)
    {
        x = i;
    }
    int getX()
    {
        return x;
    }
};
int main()
{

```

```

MyClass obs[4]; // array of an object
int i;
for(i=0; i < 4; i++)
{
    obs[i].setX(i);
}

for(i=0; i < 4; i++)
{
    cout << "obs[" << i << "].getX(): " << obs[i].getX() << "\n";
}

return 0;
}

```

Output:

```

obs[0].getX(): 0
obs[1].getX(): 1
obs[2].getX(): 2
obs[3].getX(): 3

```

24. Pointer to objects and member access

Define a pointer of class type. We can define pointer of class type, which can be used to point to class objects.

```

#include<iostream>
using namespace std;
class Simple
{
public:
    int a;
};
int main()
{
    Simple obj;
    Simple* ptr; //Pointer of class type ptr = &obj;
    ptr=&obj;

    ptr->a=20; //assigning the value of a =20
    cout << "value of a access through the object is: "<<obj.a<<endl;
    cout << "value of a access through the pointer is: "<<ptr->a<<endl; // Accessing
//member with pointer
return 0;
}

```

Output:

value of a access through the object is: 20
value of a access through the pointer is:

Here you can see that 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.

25.this pointer

The 'this' pointer is passed as a hidden argument to all non static member function calls and available as a local variable within the body of all non static functions. **'this' pointer is a constant pointer that holds the memory address of the current object.** 'this' pointer is not available in static member functions as static member functions can be called without any object (with class name).

<u>Without this pointer</u>	<u>With this pointer</u>
<pre>#include<iostream> using namespace std; class Test { private: int x; public: void setX (int x) { /* local variable is same as a member's name */ x = x; cout<< "inside SetX x= "<<x<<endl; } void print() { cout << "outside Setx = " << x << endl; } }; int main() { Test obj; int x = 20; obj.setX(x); obj.print(); return 0; }</pre> <p>Output: inside SetX x=20 outside SetX x =-2 (some unexpected values)</p>	<pre>#include<iostream> using namespace std; class Test { private: int x; public: void setX (int x) { /* local variable is same as a member's name */ this-> x = x; cout<< "inside SetX x= "<<x<<endl; } void print() { cout << "outside Setx = " << x << endl; } }; int main() { Test obj; int x = 20; obj.setX(x); obj.print(); return 0; }</pre> <p>Output: inside SetX x=20 outside SetX x =20</p>

In without pointer case, $x=x$ is the case where value of local variable is assign to the same local variable and outside that function the x is destroyed. If x is used in another function then it holds some garbage value. Here the x of object does not play any role.

In with pointer case, this pointer holds the address of the current object so $this \rightarrow x$ access the member of the current object and $this \rightarrow x=x$ is the case of assigning the value of local variable to the x of an object and outside the function if x is used, it takes the value of object member x .

26.Static data member and static function

Static data member

We can define class member static using static keyword. When we declare a member of a class as static it means no matter how many objects of the class are created, there is only one copy of the static member.

A static member is shared by all objects of the class. All static data is initialized to zero when the first object is created, if no other initialization is present. We can't put it in the class definition but it can be initialized outside the class as done in the following example by re declaring the static variable, using the scope resolution operator $::$ to identify which class it belongs to.

Static function

These functions work for the class as whole rather than for a particular object of a class. It can be called using an object and the direct member access $.$ operator. But, its more typical to call a static member function by itself, using class name and scope resolution $::$ operator.

Problem 10 Create a class with a data member to hold "serial number" for each object created from the class. That is, the first object created will be numbered 1, the second 2 and so on by using the basic concept of static data members. Use static member function if it is useful in any of the member functions declared in the program. Otherwise make separate program that demonstrate the use of static member function.

```
#include<iostream>
using namespace std;
class Static
{
private:
    static int serial_number;
public:
    Static()
    {
        ++serial_number;
        Static::objectNumber();
    }
    static void objectNumber()
    {
        cout<<" the object is numbered "<< serial_number<<endl;
    }
};
int Static::serial_number=0;
int main()
```

```

{
    Static s[5];
    return 0;
}

```

Output:

```

the object is numbered 1
the object is numbered 2
the object is numbered 3
the object is numbered 4
the object is numbered 5

```

27 Constant member function and constant objects

a. non constant function:

A non-const member function guarantees that it will modify any of its class's member data

```

#include<iostream>
using namespace std;
class Constfun
{
private:
    int num;
public:
    void setdata(int x)
    {
        num=x;
    }
    void change()      // non constant member function
    {
        num=100;      //data member changed from 20 to 100
        cout<<"num is: "<<num<<endl;
    }
};
int main()
{
    Constfun confun;
    confun.setdata(20);
    confun.change();
    return 0;
}

```

Output:

num is: 100

b. constant function:

A const member function guarantees that it will never modify any of its class's member data.

```
#include<iostream>
using namespace std;
class Constfun
{
private:
    int num;
public:
    void setdata(int x)
    {
        num=x;
    }
    void change()
    {
        num=100;
        cout<<"num is: "<<num<<endl;
    }
    void nochange() const
    {
        num=200;                //const function tries to modify data member
        cout<<"num is: "<<num<<endl;
    }
};
int main()
{
    Constfun confun;
    confun.setdata(20);
    confun.change();
    confun.nochange();
    return 0;
}
```

Result: Error because const function tries to modify data member

c. Const object with const function

<u>Non constant function and non constant object</u>	<u>Const obj and non const function</u>
<pre>#include<iostream> using namespace std; class FootballPlayer { private: int player_num; public:</pre>	<pre>#include<iostream> using namespace std; class FootballPlayer { private: int player_num; public:</pre>

<pre> FootballPlayer(int n):player_num(n){ void changePlayernumber() { cout<<"Player_number: "<<player_num<<endl; } }; int main() { FootballPlayer fp(11); fp.changePlayernumber(); return 0; } </pre> <p>Output: Player_number: 11</p>	<pre> FootballPlayer(int n):player_num(n){ void changePlayernumber() { cout<<"Player_number: "<<player_num<<endl; } }; int main() { const FootballPlayer fp(11); fp.changePlayernumber(); return 0; } </pre> <p>Output: Error- non const function does not guarantee about object's constant</p>
---	--

```

#include<iostream>
using namespace std;
class FootballPlayer
{
private:
    int player_num;
public:
    FootballPlayer(int n):player_num(n){

        void changePlayernumber() const
        {
            cout<<"Player_number: "<<player_num<<endl;

        }
};
int main()
{
    const FootballPlayer fp(11);
    fp.changePlayernumber();
    return 0;
}

```

Output: Player_number: 11

When an object is declared as const, you can't modify it. It follows that you can use only const member functions with it, because they're the only ones that guarantee not to modify it.

28. Friend Function:

If a function is defined as a friend function then, the private and protected data of a class can be accessed using the function. The compiler knows a given function is a friend function by the use of the keyword friend. For accessing the data, the declaration of a friend function should be made inside the body of the class (can be anywhere inside class either in private or public section) starting with keyword friend.

Syntax:

```
class className
{
    ... ..
    friend return_type functionName(argument/s);
    ... ..
};

return_type functionName(argument/s)
{
    ... ..
    // Private and protected data of className can be accessed from
    // this function because it is a friend function of className.
    ... ..
}
```

Problem:11 WAP to add two complex number by using the **friend function** by passing object as an argument and returning object as an argument

```
#include<iostream>
using namespace std;
class Complex
{
private:
    float real;
    float img;
public:
    Complex():real(0.0),img(0.0){}

    Complex(float x, float y):real(x),img(y){}

    friend Complex addComplex(Complex c1, Complex c2);

    void showData(Complex c3)
    {
        cout<<c3.real<< " +j " <<c3.img<<endl;
    }
};
```

```

Complex addComplex(Complex c1, Complex c2)
{
    Complex c3;
    c3.real=c1.real+c2.real;
    c3.img=c1.img+c2.img;
    return c3;
}

int main()
{
    Complex c1(12.3,23.97),c2(11.2,144.37);
    Complex c3=addComplex(c1,c2);
    c1.showData(c3);

    return 0;
}

```

Output:

23.5+ j 168.34

29. Friend class

Similarly, like a friend function, a class can also be made a friend of another class using keyword friend. For example:

```

... ..
class B;

class A
{
    // class B is a friend class of class A

    friend class B;
    ... ..
}

class B
{
}

```

When a class is made a friend class, all the member functions of that class becomes friend functions. In this program, all member functions of class B will be friend functions of class A. Thus, any member function of class B can access the private and protected data of class A. But, member functions of class A cannot access the data of class B.

Problem:12 WAP to add two complex number by using the **friend class** by passing object as an argument.

```
#include<iostream>
using namespace std;
class ComplexA
{
private:
    float real;
    float img;
public:
    friend class ComplexB;

    ComplexA():real(0.0),img(0.0){}

    ComplexA(float x, float y):real(x),img(y){}

};

class ComplexB
{
public:
    void addComplex(ComplexA c1, ComplexA c2)
    {
        ComplexA c3;
        c3.real=c1.real+c2.real;
        c3.img=c1.img+c2.img;
        cout<<c3.real <<" +j "<<c3.img<<endl;
    }
};

int main()
{
    ComplexA c1(12.3,23.97),c2(11.2,144.37);
    ComplexB c3;
    c3.addComplex(c1,c2);
    return 0;
}
```

Output:

23.5+ j 168.34

30. Operator Overloading

```
#include<iostream>
using namespace std;
int main()
{
    int a=2,b=3;
    int c=a+b;
    cout<<"sum is: "<<c<<endl;
    return 0;
}
```

Output: sum is: 5

Int, Float, double, char are the pre-defined data types. When you code an expression such as **a + b**, C++ understands that you intend to carry out binary integer addition because of the context of the + symbol; that is, you surrounded the operator with integers, defining the operation.

Q. What about the User-defined data types?

```
#include<iostream>
using namespace std;
class Complex
{
private:
    float x,y;
public:
    Complex()
    {
        x=0;
        y=0;
    };
    Complex(int a, int b):x(a),y(b){}

    Complex Add(Complex c1,Complex c2)
    {
        Complex c3;
        c3.x=x+c2.x;
        c3.y=y+c2.y;
        return c3;
    }
    void show(Complex c4)
    {
        cout<<c4.x <<" +j " <<c4.y;
    }
}
```

```

};
int main(void)
{
    Complex c1(2,4),c2(3,5);
    Complex c3=c1+c2          // what about this ??
    c3.show(c3);
    return 0;
}

```

Output: Error: no match for operator + in c1+c2

Binary Operator Overloading

When you code “**c1+c2**”, if C++ can recognize **c1** and **c2** as two instances of a class, then C++ tries to find an overloaded operator function you have written for the + symbol for that class. The name of the function that overloads the + symbol is **operator+()** function. + **takes two operand** so called **binary operator overloading**.

```

#include<iostream>
using namespace std;
class Complex
{
private:
    float x,y;
public:
    Complex()
    {
        x=0;
        y=0;
    }
    Complex(int a, int b):x(a),y(b){}

    Complex operator+(Complex c2)
    {
        Complex c3;
        c3.x=x+c2.x;
        c3.y=y+c2.y;
        return c3;
    }
    void show(Complex c4)
    {
        cout<<c4.x <<" +j " <<c4.y;
    }
}

```

```

    }
};
int main(void)
{
    Complex c1(2,4),c2(3,5);
    Complex c3=c1+c2;    //same as Complex c3=c1.operator+(c2);
    c3.show(c3);
    return 0;
}

```

Output:

5+j9

Explanation: **Complex c3=c1+c2;** is same as **Complex c3=c1.operator+(c2)** , that is in **c1+c2** , **c1** acts as the object through which it calls the function with arguments **c2**.

Check??

```

#include<iostream>
using namespace std;
class Complex
{
private:
    float x,y;
public:
    Complex()
    {
        x=0;
        y=0;
    };
    Complex(int a, int b):x(a),y(b){}

    Complex operator+(Complex c2)
    {
        Complex c3;
        c3.x=x+c2.x;
        c3.y=y+c2.y;
        return c3;
    }
    void show(Complex c4)
    {
        cout<<c4.x <<" +j " <<c4.y;
    }
};
int main(void)
{
    Complex c1(2,4),c2(3,5);
    Complex c3=c1.operator+(c2); //same as Complex c3=c1+c2
}

```

```

        c3.show(c3);
        return 0;
    }

```

Output: 5+j9

Unary Operator Overloading

Unary operator: are operators that act upon a single operand to produce a new value.

Types of unary operators:

```

        unary minus(-)
        increment(++ )
        decrement(- -)
        NOT(!)
        Address of operator(&)
        sizeof()

        ++a called prefix
        a++ called postfix

```

I. Prefix unary operator overloading:

In prefix (++a) the syntax **void operator++()** { } should be used to perform operation.

```

#include<iostream>
using namespace std;
class Complex
{
private:
    float real,img;
public:
    Complex()
    {
        real=0;
        img=0;
    }
    Complex(float a, float b)
    {
        real=a;
        img=b;
    }

    void operator++()    // prefix case
    {
        ++real;
        ++img;
        cout<<real <<" +j "<<img;
    }
}

```

```

    }

};
int main(void)
{
    Complex c1(2.2,4.5);
    ++c1;
    return 0;
}

```

Output: 3.2 + j 5.5

ii. Postfix unary operator overloading.

In postfix (a++) the syntax **void operator++(int) {}** should be used to perform operation.

```

#include<iostream>
using namespace std;
class Complex
{
private:
    float real,img;
public:
    Complex()
    {
        real=0;
        img=0;
    }
    Complex(float a, float b)
    {
        real=a;
        img=b;
    }

    void operator++(int)
    {
        ++real;
        ++img;
        cout<<real <<" +j "<<img;
    }

};
int main(void)
{
    Complex c1(2.2,4.5);
    c1++;
    return 0;
}

```

Output: 3.2 + j 5.5

iii. Unary minus

Note: (-a) is unary but careful about (a-)

Unary minus use the function **operator-()** in operator overloading

```
#include<iostream>
using namespace std;
class Complex
{
private:
    float real,img;
public:
    Complex()
    {
        real=0;
        img=0;
    }
    Complex(float a, float b)
    {
        real=a;
        img=b;
    }

    void operator-()
    {
        real--;
        img--;
        cout<<real <<" +j "<<img;
    }

};
int main(void)
{
    Complex c1(2.2,4.5);
    -c1;
    return 0;
}
```

Output: 1.2 + j 3.5

Check about c- by yourself????

Problem No 13: Write a program that add two complex number using friend functions and show the operator overloading concept as well.

```
#include<iostream>
using namespace std;
class Complex
{
private:
    float real;
    float img;
public:
    Complex():real(0.0),img(0.0){}

    Complex(float x, float y):real(x),img(y){}

    friend Complex operator+(Complex c1, Complex c2);

    void showData(Complex c3)
    {
        cout<<c3.real<< " +j "<<c3.img<<endl;
    }
};

Complex operator+(Complex c1, Complex c2)
{
    Complex c3;
    c3.real=c1.real+c2.real;
    c3.img=c1.img+c2.img;
    return c3;
}

int main()
{
    Complex c1(12.3,23.97),c2(11.2,144.37);
    Complex c3=c1+c2;
    c3.showData(c3);

    return 0;
}
```

Output:

23.5 + j 168.34

Rules of operator overloading:

- 1) Only built-in operators can be overloaded. New operators cannot be created.
- 2) Precedence and associativity of the operators cannot be changed.
- 3) Operators cannot be overloaded for built-in types only. At least one operand must be used of a defined type.
- 4) Assignment (=), subscript ([]), function call ("()"), and member selection (->) operators must be defined as member functions.
- 5) Except the operators specified in point 4, all other operators can be either member functions or non-member functions.
- 6) Some operators like (assignment)=, (address)& and comma (,) are by default overloaded.

List of operators that cannot be overloaded

- 1> Scope Resolution Operator (::)
- 2> Pointer-to-member Operator (.*)
- 3> Member Access or Dot operator (.)
- 4> Ternary or Conditional Operator (?:)
- 5> Object size Operator (sizeof)
- 6> Object type Operator (typeid)

Problem No:14 Compare the two objects of complex number that contain real and imaginary values that demonstrate the overloading of equality (==), less than (<), greater than (>), not equal (!=) operators.

```
#include<iostream>
using namespace std;
class Complex
{
private:
    float real,img;
public:
    Complex()
    {
        real=0.0;
        img=0.0;
    }
    Complex(int a, int b):real(a),img(b){}

    Complex operator+(Complex c2)
    {
```

```

        Complex c3;
        c3.real=real+c2.real;
        c3.img=img+c2.img;
        return c3;
    }

Complex operator-(Complex c2)
{
    Complex c3;
    c3.real=real-c2.real;
    c3.img=img-c2.img;
    return c3;
}

bool operator==(Complex c2)
{
    return ((real == c2.real) && (img==c2.img));
}

bool operator!=(Complex c2)
{
    return !((real== c2.real) && (img==c2.img));
}

bool operator<(Complex c2)
{
    float x=real*real+img*img;
    float y=c2.real*c2.real+c2.img*c2.img;
    return (x<y ? true : false);
}

bool operator>(Complex c2)
{
    float x=real*real+img*img;
    float y=c2.real*c2.real+c2.img*c2.img;
    return (x>y ? true : false);
}

void show(Complex c4)
{
    cout<<c4.real <<" +j " <<c4.img<<endl;
}
};

int main(void)
{
    Complex c1(1,1),c2(1,0);
    Complex c3=c1+c2;    //same as Complex c3=c1.operator+(c2);
    Complex c4=c1-c2;
    c3.show(c3);
    c4.show(c4);
}

```

```

    if(c3==c4)
    {
        cout<<"Objects are equal: "<<endl;
    }

    if(c3!=c4)
    {
        cout<<"Objects are not equal: "<<endl;
    }

    if(c3<c4)
    {
        cout<<"magnitude of c3 is less than magnitude of c4: "<<endl;
    }

    if(c3>c4)
    {
        cout<<"magnitude of c3 is greater than magnitude of c4: "<<endl;
    }

    return 0;
}

```

Output:

```

2 +j 1
0 +j 1
Objects are not equal:
magnitude of c3 is greater than magnitude of c4:

```

Problem 15 WAP to overload the insertion (<<) and the extraction (>>) operators.

```

#include<iostream>
using namespace std;
class Complex
{
private:
    float real,img;
public:
    Complex()
    {
        real=0;
        img=0;
    };
    Complex(float a, float b):real(a),img(b){}
    friend istream &operator>>(istream &in,Complex &c1);
    friend ostream &operator<<(ostream &out,const Complex &c1);
}

```

```

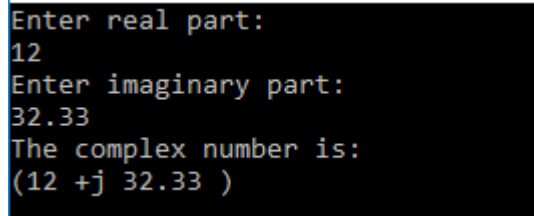
};
istream &operator>>(istream &in, Complex &c1)
{
    cout<<"Enter real part: "<<endl;
    in>>c1.real;
    cout<<"Enter imaginary part: "<<endl;
    in>>c1.img;
    return in;
}

ostream &operator<<(ostream &out,const Complex &c1)
{
    out<<"("<<c1.real<<" +j "<<c1.img<<" ) "<<endl;
}

int main(void)
{
    Complex c1;
    cin>>c1;                                //same as operator>>(cin,c1);
    cout<<"The complex number is: "<<endl;
    cout<<c1;                                // same as operator<<(cout,c1);
    return 0;
}

```

Output:



```

Enter real part:
12
Enter imaginary part:
32.33
The complex number is:
(12 +j 32.33 )

```

Explanation: In previous overloading we added an object of same class but while overloading >> and << the cases are different. cin and cout are objects of class istream and ostream while c1 is an object of user defined data type Complex. So here we have to add an object of two different class. In cin>>c1, we have to overload the member function of class istream but to access the private data from class Complex, the function must be made global and using friend it can access the private data member of class Complex.

30.5 Data Conversion: Basic - User Defined and User Defined – User Defined

a. Basic to Basic Defined

```
#include<iostream>
using namespace std;
int main(void)
{
    float a=12.33;
    cout<<"The value of a: "<<a<<endl;
    int b=a;
    cout<<"The value of b: "<<b<<endl;
    return 0;
}
```

Output:

The value of a: 12.33

The value of b: 12

The data type of a is float and value is 12.33 but when the same value is assign to the b it shows 12. Here the implicit conversion takes place.

b. Basic-User Defined:

When we want to convert between user-defined data types and basic types, we can't rely on Built-in conversion, since the compiler doesn't know anything about user-defined Types besides what we tell it. We must write these functions ourselves.

```
#include<iostream>
using namespace std;
class Distance
{
private:
    float meter;
    const float CHANGE_FACTOR;
public:
    Distance():meter(0.0f),CHANGE_FACTOR(0.01f){}
    Distance(float cm):CHANGE_FACTOR(0.01f)
    {
        meter =CHANGE_FACTOR*cm;
    }
    void showMeter()
    {
        cout<<"The length in meter is: "<<meter<<endl;
    }
};
int main(void)
{
    Distance d=22.3f;
    d.showMeter();
}
```

```

        return 0;
    }

```

Output: The length of meter is: 0.223

Explanation: To go from a basic type—float in this case—to a user-defined type such as Distance, we use a constructor with one argument. These are sometimes called conversion constructors.

```

Distance(float cm):CHANGE_FACTOR(0.01f)
{
    meter =CHANGE_FACTOR*cm;
}

```

It converts the argument cm, and assigns the resulting values to the object as meter. Thus the conversion from cm to Distance is carried out along with the creation of an object in the statement

```

Distance d=22.3f;

```

c. User Defined-Basic Defined:

What about going the other way, from a user-defined type to a basic type? The trick here is to create something called a conversion operator.

```

#include<iostream>
using namespace std;
class Distance
{
private:
    float meter;
    const float CHANGE_FACTOR;
public:
    Distance():meter(0.0f),CHANGE_FACTOR(0.01f){}
    Distance(float cm):CHANGE_FACTOR(0.01f)
    {
        meter =CHANGE_FACTOR*cm;
    }

    operator float()
    {
        float cm=(meter)*100;
        return cm;
    }

    void showMeter()
    {
        cout<<"The length in meter is: "<<meter<<endl;
    }
};
int main(void)
{

```



```

Distance d=22.3f;
d.showMeter();
float centimeter;
centimeter=static_cast<float>(d);
cout<<"The length in centimeter is: "<<centimeter<<endl;
return 0;
}

```

Output:

```

The length in meter is: 0.223
The length in centimeter is: 22.3

```

Explanation: Going in the other direction, it converts a Distance to cm in the statements.

```

operator float()
{
    float cm=(meter)*100;
    return cm;
}

```

This operator takes the value of the Distance object of which it is a member, converts it to a float value representing cm, and returns this value.

```

float centimeter;
centimeter=static_cast<float>(d);

```

30.6 Explicit constructor

```

#include<iostream>
using namespace std;
class Distance
{
private:
    float meter;
    const float CHANGE_FACTOR;
public:
    Distance():meter(0.0f),CHANGE_FACTOR(0.01f){}
    Distance(float cm):CHANGE_FACTOR(0.01f)
    {
        meter =CHANGE_FACTOR*cm;
    }
    void showMeter()
    {
        cout<<"The length in meter is: "<<meter<<endl;
    }
};
int main(void)
{
    Distance d1(23.22);
    Distance d=22.3f;
    d.showMeter();
    d1.showMeter();
}

```

```
        return 0;
    }
```

Output: The length in meter is: 0.2322
The length in meter is: 0.223

Here, Distance d=22.3f; is the implicit conversion so to avoid the implicit conversion we have to use explicit keyword. Reason is here one argument constructor is used for initialization but it also works for the **Distance d=22.3f** which may be the unwanted outcomes.

31 Inheritance

Suppose, Apple company launched i3, after some years i4, i5 and so on. You have better notice, the i4 features=i3 features + some extra features, i5 features=i4 features + some extra features and so on. Here you can see, for i4 you only have to add extra features and for other remaining features you can use original features of i3. Here you can see i4 extends the features of i3.

By using previous features of i3, time can be saved because there is no need to make again the same features for i4 which is already in i3 and such concept in Object oriented programming is called inheritance. It helps to reuse code.

Suppose, i3 has some features which is not suitable or efficient then in i4 you may find same features with better efficiency because same features is re-written in i4.

And such concept of rewriting the code in OOP either to improve the implementation or to change implementation is called method overriding.

Suppose we have class B defined as shown here:

```
class B
{
    // Details omitted
};
```

To derive a new class D from B we use the following syntax:

```
class D: public B
{
    // Details omitted
};
```

Here B is the base class and D is the derived class. B is the pre-existing class, and D is the new class based on B.

To see how inheritance works, consider classes B and D with additional detail:

```

#include<iostream>
using namespace std;
class B {
// Other details omitted
public:
void f()
{
    cout << "In function 'f'" << endl;
}
};

class D: public B
{
// Other details omitted
public:
    void g()
    {
        cout << "In function 'g'" << endl;
    }
};
int main()
{
    B myb;
    D myd;
    myb.f();
    myd.g();
    myd.f();
}

```

Output:

```

In function 'f'
In function 'g'
In function 'f'

```

Even though the source code for class D does not explicitly show the definition of a method named f, it has such a method that it inherits from class B. Note that inheritance works in one direction only. Class D inherits method f from class B, but class B cannot inherit D's g method. Given the definitions of classes B and D above, the following code is illegal:

```

B myb;
myb.g(); // Illegal, a B object is NOT a D object

```

Problem 16: Program which clearly explains the inheritance properties:

```
#include<iostream>
using namespace std;
class Iphoneversionfirst
{
    private:
        string model;
        string color;
        int weight;

    protected:
        string access= "I am access to derived class ";

    public:
        void setinformation(string imodel, string icolor, int iweight)
        {
            model=imodel;
            color=icolor;
            weight=iweight;
        }
        void getinformation()
        {
            cout<<"model: "<<model<<endl;
            cout<<"color: "<<color<<endl;
            cout<<"weight: "<<weight<< " gm "<<endl;
            cout<<endl;
        }
        void cameraDisplay()
        {
            cout<<"version 1 has 12 Mpx camera "<<endl<<endl;
        }
};

class Iphoneversionsecond: public Iphoneversionfirst
{
    public:
        void cameraDisplay()
        {
            cout<<"version 2 has 24Mpx camera "<<endl;    // Overriding
            //Iphoneversionfirst::cameraDisplay();    //calling display of base
        }
        void accessProtected()
        {
            cout<<"Protected member of base is access in derived class: "<<endl;
            cout<<access<<endl;
        }
        void accessOriginalCameraDisplayfromderived()
        {
            cout<<endl;
            Iphoneversionfirst::cameraDisplay(); //access from derived using this syntax
        }
};
```

```

    }
};
int main()
{
    Iphoneversionfirst i3;
    cout<<"Version 1 iphone camera properties: "<<endl;
    i3.cameraDisplay();

    cout<<"set the properties using derived class object: "<<endl;
    Iphoneversionsecond i31, i32; // object creation
    i31.setinformation ("i3","black",32);          //access member func.
    i31.getinformation();

    i32.setinformation ("i3","white",30);
    i32.getinformation();

    cout<<"Version 2 iphone override the camera properties: "<<endl;
    i32.cameraDisplay();          //calling overriding func.

    cout<<endl;
    i32.accessProtected(); //verified of access of protected from derived class

    i32.accessOriginalCameraDisplayfromderived();
    return 0;
}

```

Output:

```

Version 1 iphone camera properties:
version 1 has 12 Mpx camera

set the properties using derived class object:
model: i3
color: black
weight: 32 gm

model: i3
color: white
weight: 30 gm

Version 2 iphone override the camera properties:
version 2 has 24Mpx camera

Protected member of base is access in derived class:
I am access to derived class

version 1 has 12 Mpx camera

```

Explanation: Here, Iphoneversionsecond is the derived class which possesses all the features of Iphoneversionfirst except one which is redefined so called overriding function.

Here protected member has visibility in derived class, without calling public function of base class to access protected member, you can easily access the protected member of base class from derived class. If you want to call overridden function of base class from derived class then use syntax as

Syntax:

Baseclass_name:: function_name () i.e. Iphoneversionfirst::display();

Access specifiers:

Q. How do you access private, protected and public member in Derived class?

a. private member

From base class:

private member can be accessed only from public member of base class. But cannot accessed from the derived class and outside the main function.

```
#include<iostream>
using namespace std;
class AccessCheck
{
    private:
        int a;
    public:
        void check()
        {
            cout<<"Enter the value of a "<<endl;
            cin>>a;
            cout<<"The value of a: "<<a<<endl;
        }
};
class DerivedAccessCheck:public AccessCheck
{
};

int main()
{
    AccessCheck ac;
    ac.check();
    return 0;
}
```

Output:

```
Enter the value of a
5
The value of a: 5
```

From derived class:

Private member of the base class cannot be accessed from the derived class and outside main function.

```
#include<iostream>
using namespace std;
class AccessCheck
{
    private:
        int a;
};
class DerivedAccessCheck: public AccessCheck
{
    public:
        void check()
        {
            cout<<"Enter the value of a"<<endl;
            cin>>a;
            cout<<"The value of a: "<<a<<endl;
        }
};

int main()
{
    DerivedAccessCheck dc;
    dc.check();
    return 0;
}
```

Output:

Error: Private member of base class cannot be accessed to the derived class

b. protected member

protected member is accessible to the derived class but not to the outside of the main() function.

From derived class:

```
#include<iostream>
using namespace std;
class AccessCheck
{
    protected:
        int a;
};
class DerivedAccessCheck:public AccessCheck
{
    public:
        void check()
```

```

        {
            cout<<"Enter the value of a"<<endl;
            cin>>a;
            cout<<"The value of a: "<<a<<endl;
        }
    };

    int main()
    {
        DerivedAccessCheck dc;
        dc.check();
        return 0;
    }

```

Output:

```

Enter the value of a
5
The value of a: 5

```

From main function:

```

#include<iostream>
using namespace std;
class AccessCheck
{
    protected:
        int a;
};
class DerivedAccessCheck:public AccessCheck
{
    public:

};

int main()
{
    DerivedAccessCheck dc;
    dc.check();
    dc.a=5;
    return 0;
}

```

Output:

Error: protected member cannot be accessed from outside the main function.

c. Public member:

Public member can be accessed from derived class as well as from main function.

From derived class:

```

#include<iostream>
using namespace std;

```



```

class AccessCheck
{
    public:

        int a;
};
class DerivedAccessCheck:public AccessCheck
{
    public:
        void check()
        {
            cout<<"Enter the value of a"<<endl;
            cin>>a;
            cout<<"The value of a: "<<a<<endl;
        }
};
int main()
{
    DerivedAccessCheck dc;
    dc.check();
    return 0;
}

```

Output:

```

Enter the value of a
5
The value of a: 5

```

From outside the main function:

The public member of base and derived both can be accessed from anywhere, here it's access from the main function is shown.

```

#include<iostream>
using namespace std;
class AccessCheck
{
    public:
        int a;

};
class DerivedAccessCheck:public AccessCheck
{

};
int main()
{
    DerivedAccessCheck dc;
    dc.a=5;
    cout<<"value of a is: "<<dc.a<<endl;
    return 0;
}

```

```
}
```

Output: value of a is: 5

Q. what are the role of publicly and privately Derived class?

If we omit the word public from class D's definition, as in

```
class D: B
{
    // Details omitted
};
```

all the public members of B inherited by D objects will be private by default. **What's new the difference between publicly derived and privately derived classes.** Objects of the publicly derived class D can access public members of the base class B, while objects of the privately derived class D cannot; they can only access the public members of their own derived class.

```
#include<iostream>
using namespace std;
class B {
// Other details omitted
public:
void f()
{
    cout << "In function 'f'" << endl;
}
};

class D: public B
{
// Other details omitted
public:
void g()
{
    cout << "In function 'g'" << endl;
}
};
int main()
{
    B myb;
    D myd;
    myd.f();
    return 0;
}
```

Output: In function 'f'

Explanation: Here the publicly derived class Ds object can access the public member of the base class B from the main function.

```

#include<iostream>
using namespace std;
class B {
// Other details omitted
public:
void f()
{
    cout << "In function 'f'" << endl;
}
};

class D: private B
{
// Other details omitted
public:
    void g()
    {
        cout << "In function 'g'" << endl;
    }
};
int main()
{
    B myb;
    D myd;
    myd.f();
    return 0;
}

```

Output: Error:- Privately derived class Ds object cannot access the public member of the base class B even from the main function.

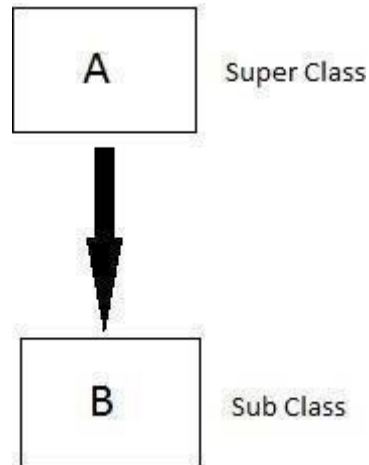
Level of inheritance

In C++, we have 5 different types of Inheritance. Namely,

1. Single Inheritance
2. Multiple Inheritance
3. Hierarchical Inheritance
4. Multilevel Inheritance
5. Hybrid Inheritance (also known as Virtual Inheritance)

1. Single Inheritance:

In this type of inheritance one derived class inherits from only one base class. It is the simplest form of Inheritance.



Syntax:

```
class subclass_name : access_mode base_class
{
    //body of subclass
};
```

Example:

```
#include <iostream>
using namespace std;
// base class
class Vehicle
{
public:
    Vehicle()
    {
        cout << "This is a Vehicle" << endl;
    }
};
// sub class derived from two base classes
class Car: public Vehicle{

};
// main function
int main()
{
    // creating object of sub class will
    // invoke the constructor of base classes
```

```

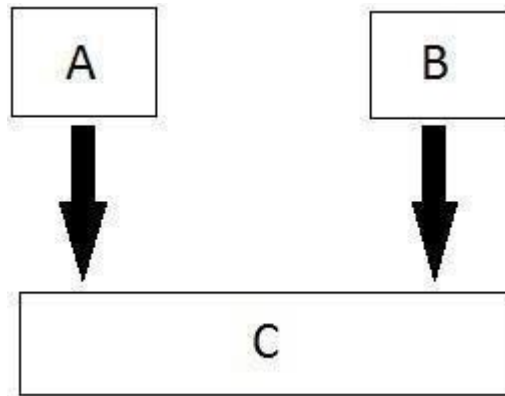
    Car obj;
    return 0;
}

```

Output: This is a vehicle

2. Multiple Inheritances:

In this type of inheritance a single derived class may inherit from two or more than two base classes.



Syntax:

```

class subclass_name : access_mode base_class1, access_mode base_class2, ....
{
    //body of subclass
};

```

```

#include <iostream>
using namespace std;

```

```

// first base class
class Vehicle {
public:
    Vehicle()
    {
        cout << "This is a Vehicle" << endl;
    }
};

```

```

// second base class
class FourWheeler {
public:
    FourWheeler()
    {
        cout << "This is a 4 wheeler Vehicle" << endl;
    }
};

```

```

    }
};

// sub class derived from two base classes
class Car: public Vehicle, public FourWheeler {

};

// main function
int main()
{
    // creating object of sub class will
    // invoke the constructor of base classes
    Car obj;
    return 0;
}

```

Output:

```

This is a vehicle
This is a 4 wheeler vehicle

```

Problem17. What kind of problem is encountered in multiple inheritances?

Ans:

Two base classes have functions with the same name, while a class derived from both base classes has no function with this name. How do objects of the derived class access the correct base class function?

```

#include <iostream>
using namespace std;
class A
{
public:
    void show()
    {
        cout << "Class A\n"; }
};
class B
{
public:
    void show()
    {
        cout << "Class B\n"; }
};
class C : public A, public B
{

```

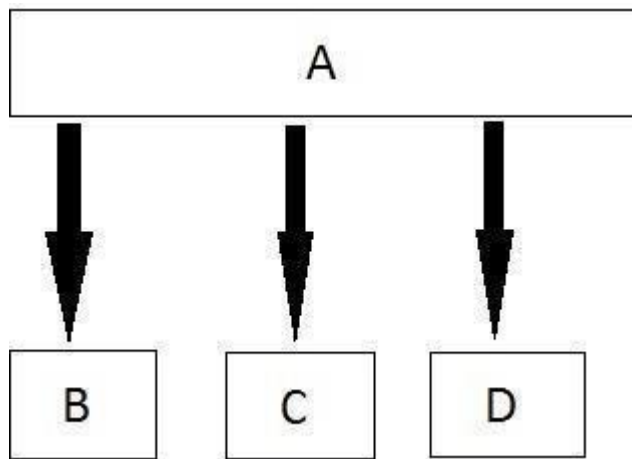
```
};
int main()
{
    C objC;           //object of class C
    objC.show();       //ambiguous--will not compile
    return 0;
}
```

Output: Ambiguity arises here because object of C does not explicitly mention, show() belongs to which base class either A or B, which is also called diamond problem.

Correction: `objC.A::show();` is used to call the show() function of class A. Similarly, `objC.B::show();` is used to call the show() function of class B.

3. Hierarchical Inheritance:

In this type of inheritance, multiple derived classes inherit from a single base class.



```
#include <iostream>
using namespace std;
// base class
class Vehicle
{
    public:
    Vehicle()
    {
        cout << "This is a Vehicle" << endl;
    }
};

// first sub class
```

```

class Car: public Vehicle
{

};

// second sub class
class Bus: public Vehicle
{

};

// main function
int main()
{
    // creating object of sub class will
    // invoke the constructor of base class
    Car obj1;
    Bus obj2;
    return 0;
}

```

Output:

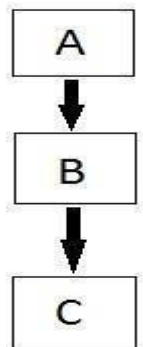
```

This is a Vehicle
This is a Vehicle

```

4. Multilevel Inheritance

In this type of inheritance the derived class inherits from a class, which in turn inherits from some other class. The Super class for one, is sub class for the other.



```

#include <iostream>
using namespace std;

```

```

// base class

```



```

class Vehicle
{
    public:
    Vehicle()
    {
        cout << "This is a Vehicle" << endl;
    }
};
class fourWheeler: public Vehicle
{
    public:
    fourWheeler()
    {
        cout<<"Objects with 4 wheels are vehicles"<<endl;
    }
};
// sub class derived from two base classes
class Car: public fourWheeler
{
    public:
    Car()
    {
        cout<<"Car has 4 Wheels"<<endl;
    }
};

// main function
int main()
{
    //creating object of sub class will
    //invoke the constructor of base classes
    Car obj;
    return 0;
}

```

Output:

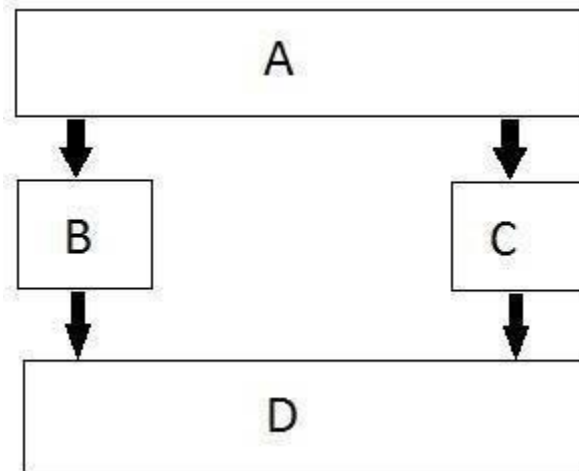
```

This is a Vehicle
Objects with 4 wheels are vehicles
Car has 4 Wheels

```

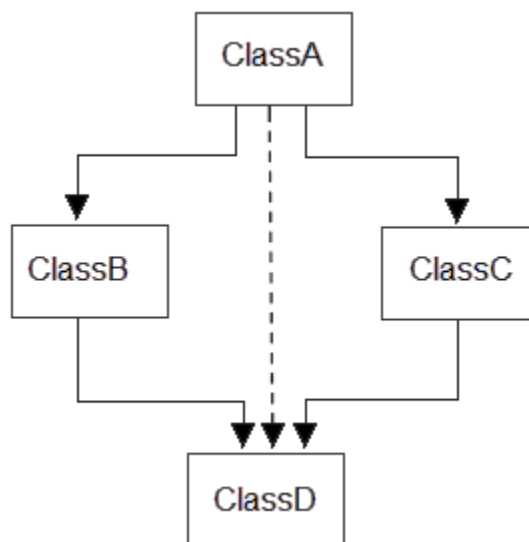
5. Hybrid (Virtual) Inheritance

Hybrid Inheritance is the combination of Hierarchical and Multilevel Inheritance.



6. Multipath inheritance

A derived class with two base classes and these two base classes have one common base class is called multipath inheritance.



Problem18: What is diamond problem or ambiguity problem in multipath inheritances?

In the above example, both ClassB & ClassC inherit ClassA, they both have single copy of ClassA. However ClassD inherits both ClassB & ClassC, therefore ClassD has two copies of ClassA, one from ClassB and another from ClassC.

If we need to access the data member `a` of ClassA through the object of ClassD, we must specify the path from which `a` will be accessed, whether it is from ClassB or ClassC, because the compiler can't differentiate between two copies of ClassA in ClassD.

```

#include <iostream>
using namespace std;
class A
{
public:
    A()
    {
        cout << "Class A\n";
    }
};

class B:public A
{
public:
    B()
    {
        cout << "Class B\n";
    }
};

class C:public A
{
public:
    C()
    {
        cout << "Class C\n";
    }
};

class D : public B, public C
{
public:
    D()
    {
        cout << "Class D\n";
    }
};

int main()
{
    D objd; //object of class D
    return 0;
}

```

Output:

```

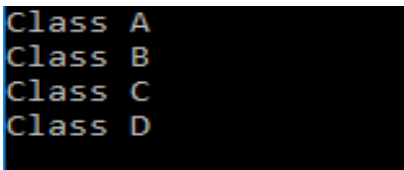
Class A
Class B
Class A
Class C
Class D

```

In this program, constructor of A called two times because D has two copies of all members of A, this causes ambiguities. **Solution:** The solution to this problem is **virtual** keyword.

```
#include <iostream>
using namespace std;
class A
{
public:
    A()
    {
        cout << "Class A\n";
    }
};
class B:virtual public A
{
public:
    B()
    {
        cout << "Class B\n";
    }
};
class C:virtual public A
{
public:
    C()
    {
        cout << "Class C\n";
    }
};
class D : public B, public C
{
public:
    D()
    {
        cout << "Class D\n";
    }
};
int main()
{
    D objd;           //object of class D
    return 0;
}
```

Output:

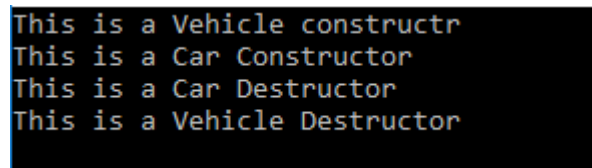


```
Class A
Class B
Class C
Class D
```

Constructor and destructor invocation in single inheritance

```
#include <iostream>
using namespace std;
// base class
class Vehicle
{
public:
    Vehicle()
    {
        cout << "This is a Vehicle constructr" << endl;
    }
    ~Vehicle()
    {
        cout << "This is a Vehicle Destructor" << endl;
    }
};
// sub class derived from two base classes
class Car: public Vehicle
{
public:
    Car()
    {
        cout<<"This is a Car Constructor" <<endl;
    }
    ~Car()
    {
        cout<<"This is a Car Destructor" <<endl;
    }
};
// main function
int main()
{
    // creating object of sub class will
    // invoke the constructor of base classes
    Car obj;
    return 0;
}
```

Output:



```
This is a Vehicle constructr
This is a Car Constructor
This is a Car Destructor
This is a Vehicle Destructor
```

The destructors are called in reverse order of constructors.

Constructor and destructor invocation in multiple inheritance

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

class A
{
public:
    A()
    {
        cout << "A's constructor called" << endl;
    }
    ~A()
    {
        cout << "A's Destructor called" << endl;
    }
};

class B
{
public:
    B()
    {
        cout << "B's constructor called" << endl;
    }
    ~B()
    {
        cout << "B's Destructor called" << endl;
    }
};

class C: public B, public A // Note the order
{
public:
    C()
    {
        cout << "C's constructor called" << endl;
    }
    ~C()
    {
        cout << "C's Destructor called" << endl;
    }
};

int main()
{
    C c;
    return 0;
}
```

Output:

```
B's constructor called
A's constructor called
C's constructor called
C's Destructor called
A's Destructor called
B's Destructor called
```

The destructors are called in reverse order of constructors.

32 polymorphism

Virtual means existing in appearance but not in reality. When virtual functions are used, a program that appears to be calling a function of one class may in reality be calling a function of a different class. Why are virtual functions needed? Suppose you have a number of objects of different classes but you want to put them all in an array and perform a particular operation on them using the same function call.

For example, suppose a graphics program includes several different shapes: a triangle, a ball, a square, and so on. Now suppose you plan to make a picture by grouping a number of these elements together, and you want to draw the picture in a convenient way. One approach is to create an array that holds pointers to all the different objects in the picture. The array might be defined like this:

```
shape* ptrarr[100];    // array of 100 pointers to shapes
```

If you insert pointers to all the shapes into this array, you can then draw an entire picture using a simple loop:

```
for(int j=0; j<N; j++)
    ptrarr[j]->draw();
```

This is an amazing capability: Completely different functions are executed by the same function call. If the pointer in `ptrarr` points to a ball, the function that draws a ball is called; if it points to a triangle, the triangle-drawing function is called. This is called polymorphism, which means different forms. The functions have the same appearance, the `draw()` expression, but different actual functions are called, depending on the contents of `ptrarr[j]`.

For the polymorphic approach to work, several conditions must be met.

- a. **First, all the different classes of shapes, such as balls and triangles, must be descended from a single base class.**
- b. **Second, the `draw()` function must be declared to be virtual in the base class. (Ref: Robert Lafore)**

Explanation:

```
#include <iostream>
using namespace std;
class Base //base class
{
public:
void draw() //normal function
{ cout << "Base\n"; }
};
class Circle: public Base //derived class 1
{
public:
void draw()
{ cout << "Circle\n"; }
};
class Triangle : public Base //derived class 2
{
public:
void draw()
{ cout << "Triangle\n"; }
};
int main()
{
    Circle cir;           //object of derived class 1
    Triangle tri;         //object of derived class 2
    Base* ptr;            //pointer to base class
    ptr = &cir;            //put address of cir in pointer
    ptr->draw();           //execute show()
    ptr = &tri;            //put address of tri in pointer
    ptr->draw();           //execute show()
    return 0;
}
```

Q. Pointer is of base but base class pointer hold the address of child class object so which display is going to execute???

Output:

Base
Base

Explanation: As you can see, the function in the base class is always executed. The compiler ignores the contents of the pointer ptr and chooses the member function that matches the type of the Pointer.

Expected outcomes:

Circle
Triangle

So how can base class pointer be able to point the function of derived class so that using only one base pointer we can easily point to function of different derived class??

Problem19. What is the need of virtual keyword in polymorphism explain?

Virtual: concept of virtual keyword is arises. If base class pointer is meant to call the function of derived class then use the virtual keyword just before datatypes of function in base class which is overridden in derived class. This suggests that the function is just as an interface and it is implemented in derived class. So when base class pointer call base funtion, virtual keyword tells compiler that the function is just interface so look for it's implementation in derived class. The address of object of derived class which pointer of base class holds tells where this function is implemented.

```
#include <iostream>
using namespace std;
class Base //base class
{
public:
    virtual void draw() //normal function
    { cout << "Base\n"; }
};
class Circle: public Base //derived class 1
{
public:
    void draw()
    { cout << "Circle\n"; }
};
class Triangle : public Base //derived class 2
{
public:
    void draw()
    { cout << "Triangle\n"; }
};
int main()
{
    Circle cir;           //object of derived class 1
    Triangle tri;         //object of derived class 2
    Base* ptr;            //pointer to base class
    ptr = &cir;            //put address of cir in pointer
    ptr->draw();           //execute show()
    ptr = &tri;            //put address of tri in pointer
    ptr->draw();           //execute show()
    return 0;
}
```

Output:

Circle
Triangle

Explanation:

Which version of draw() does the compiler call? Infact the compiler doesn't know what to do, so it arranges for the decision to be deferred until the program is running. At runtime, when it is known what class is pointed to by ptr, the appropriate version of draw will be called. **This is called late binding or dynamic binding.** (Choosing functions in the normal way, during compilation, is called **early binding or static binding.**) (Ref. Lafore)

Array of pointer to base class

```
#include <iostream>
using namespace std;
class Base //base class
{
public:
    virtual void draw() //normal function
    { cout << "Base\n"; }
};
class Circle: public Base //derived class 1
{
public:
    void draw()
    { cout << "Circle\n"; }
};
class Triangle : public Base //derived class 2
{
public:
    void draw()
    { cout << "Triangle\n"; }
};
class Rectangle : public Base //derived class 2
{
public:
    void draw()
    { cout << "Rectangle\n"; }
};

int main()
{
    Base *ptr[3];
    ptr[0]=new Circle();
    ptr[1]=new Triangle();
    ptr[2]=new Rectangle();

    for(int i=0;i<3;i++)
    {
        ptr[i]->draw();
    }
    for(int i=0;i<3;i++)
    {
```

```

        delete ptr[i];
    }
    return 0;
}

```

Output:

```

Circle
Triangle
Rectanlge

```

pure virtual function and abstract class

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,

Virtual void f() = 0;

Pure Virtual functions can be given a small definition in the Abstract class, which you want all the derived classes to have. Still you cannot create object of Abstract class. Also, the Pure Virtual function must be defined outside the class definition. If you will define it inside the class definition, compiler will give an error. Inline pure virtual definition is Illegal.

Abstract Class is a class which contains at least 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. With pure virtual function, we cannot create object of base class.

Characteristics of Abstract Class

1. Abstract class **cannot be instantiated**, but pointers and references 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.

```

#include <iostream>
using namespace std;
class Base //base class

```

```

{
public:
virtual void draw()=0; //pure virtual function
};
class Circle: public Base //derived class 1
{
public:
void draw()
{ cout << "Circle\n"; }
};
class Triangle : public Base //derived class 2
{
public:
void draw()
{ cout << "Triangle\n"; }
};
int main()
{
    Circle cir;           //object of derived class 1
    Triangle tri;         //object of derived class 2
    Base* ptr;            //pointer to base class
    ptr = &cir;           //put address of cir in pointer
    ptr->draw();           //execute show()
    ptr = &tri;           //put address of tri in pointer
    ptr->draw();           //execute show()
    return 0;
}

```

Output:

Circle
Triangle

Virtual Destructor

Suppose you use delete with a base class pointer to a derived class object to destroy the derived-class object. If the base-class destructor is not virtual then delete, like a normal member function, calls the destructor for the base class, not the destructor for the derived class. (Ref. Lafore)

```

#include <iostream>
using namespace std;
class Base
{
public:
~Base()
{
    cout<<"Base Destroyed\n";
}           //non-virtual destructor;
};
class Derv : public Base
{

```

```

public:
~Derv()
{
    cout << "Derv destroyed\n"; }
};

int main()
{
    Base* pBase = new Derv;
    delete pBase;
    return 0;
}

```

Output:

Base Destroyed

This shows that the destructor for the Derv part of the object isn't called. In the listing the base class destructor is not virtual, but you can make it so by adding the keyword virtual for the destructor as:

```

#include <iostream>
using namespace std;
class Base
{
public:
virtual ~Base()
{
    cout<<"Base Destroyed\n";
}          //non-virtual destructor;
};
class Derv : public Base
{
public:
~Derv()
{
    cout << "Derv destroyed\n"; }
};

int main()
{
    Base* pBase = new Derv;
    delete pBase;
    return 0;
}

```

Output:

Derv destroyed
Base Destroyed

Example 1:

```
#include<iostream>
using namespace std;
class PolyBase
{
private:
    string name;
public:
    void setname(string name1)
    {
        name=name1;
    }
    void getname()
    {
        cout<<name<<endl;
    }
    void display()
    {
        cout<<"Base class display"<<endl;
    }
    void baseclass()
    {
        cout<<"baseclass function"<<endl;
    }
};

class PolyChild:public PolyBase
{
public:
    void display()
    {
        cout<<"child class display"<<endl;
    }
    void childclass()
    {
        cout<<"child class function"<<endl;
    }
};

int main()
{
    PolyBase pbase;
    PolyBase *pb;
    pb=&pbase;
    pb->setname("Harke");//Base class pointer call base class function
    pb->getname();
    pb->display();
    pb->baseclass();
    return 0;
}
```

Output:

```
Harke
Base class display
baseclass function
```

Explanation:

In above example, base class pointer holds the address of base class objects. So when calling the base class function through the base class pointer there is no error in compilation. It works fine.

Example 2:

```
#include<iostream>
using namespace std;
class PolyBase
{
private:
    string name;
public:
    void setname(string name1)
    {
        name=name1;
    }
    void getname()
    {
        cout<<name<<endl;
    }
    void display()
    {
        cout<<"Base class display"<<endl;
    }
    void baseclass()
    {
        cout<<"baseclass function"<<endl;
    }
};

class PolyChild:public PolyBase
{
public:
    void display()
    {
        cout<<"child class display"<<endl;
    }
    void childclass()
    {
        cout<<"child class function"<<endl;
    }
};
int main()
```

```

{
    PolyChild pchild;
    PolyChild *pc;           //Child class pointer
    pc=&pchild;
    pc->setname("Birkhe");    //Child class pointer call child class function
    pc->getname();
    pc->display();
    pc->childclass();
    return 0;
}

```

Output:

```

Birkhe
child class display
child class function

```

Example3:

```

#include<iostream>
using namespace std;
class PolyBase
{
private:
    string name;
public:
    void setname(string name1)
    {
        name=name1;
    }
    void getname()
    {
        cout<<name<<endl;
    }
    void display()
    {
        cout<<"Base class display"<<endl;
    }
    void baseclass()
    {
        cout<<"baseclass function"<<endl;
    }
};

class PolyChild:public PolyBase
{
public:
    void display()
    {
        cout<<"child class display"<<endl;
    }
}

```



```

void childclass()
{
    cout<<"child class function"<<endl;
}
};
int main()
{
    PolyBase pbase; //Child class pointer PolyChild pchild;
    PolyChild *pc;
    pc=&pbase;      //invalid conversion from 'PolyBase*' to 'PolyChild*'
    return 0;
}

```

Output: invalid conversion from 'PolyBase*' to 'PolyChild*'

Explanation: In above example, child class pointer holds the address of base class objects. It shows invalid conversion from 'PolyBase*' to 'PolyChild*'

Example 4:

```

#include<iostream>
using namespace std;
class PolyBase
{
private:
    string name;
public:
    void setname(string name1)
    {
        name=name1;
    }
    void getname()
    {
        cout<<name<<endl;
    }
    void display()
    {
        cout<<"Base class display"<<endl;
    }
    void baseclass()
    {
        cout<<"baseclass function"<<endl;
    }
};

class PolyChild:public PolyBase
{

```

```

    public:
    void display()
    {
        cout<<"child class display"<<endl;
    }
    void childclass()
    {
        cout<<"child class function"<<endl;
    }
};
int main()
{
    PolyBase pbase;
    PolyBase *pb;          //Base class pointer
    PolyChild pchild;
    pb=&pchild;             //Base class pointer holds the address of child class
    pb->display();
    return 0;
}

```

Q. Pointer is of base but base class pointer hold the address of child class object so which display is going to execute???

Output: Base class Display

Explanation: As you can see, the function in the base class is always executed. The compiler ignores the contents of the pointer pb and chooses the member function that matches the type of the Pointer.

dynamic cast

```

#include <iostream>
#include <typeinfo>          //for dynamic_cast
using namespace std;
class Base
{
protected:
    int ba;

public:
    Base() : ba(0)
    { }
    Base(int b) : ba(b)
    { }
    virtual void vertFunc()      //needed for dynamic_cast
    { }
    void show()
    {

```

```

        cout << "Base: ba=" << ba << endl;
    }
};
class Derv : public Base
{
private:
    int da;
public:
    Derv(int b, int d) : da(d)
    {
        ba = b;
    }
void show()
{
    cout << "Derv: ba=" << ba << ", da=" << da << endl;
}
};
int main()
{
    Base* pBase = new Base(10);                //pointer to Base
    Derv* pDerv = new Derv(21, 22);            //pointer to Derv

    pBase->show();
    pDerv->show();

    //derived-to-base: upcast -- points to Base subobject of Derv
    pBase = dynamic_cast<Base*>(pDerv);
    pBase->show();                            //"Base: ba=21"

    pBase = new Derv(31, 32);                  //normal
    pBase->show();
    //base-to-derived: downcast -- (pBase must point to a Derv)
    pDerv = dynamic_cast<Derv*>(pBase);
    pDerv->show();                            //"Derv: ba=31, da=32"
    return 0;
}

```

Output:

```

Base: ba=10
Derv: ba=21, da=22
Base: ba=21
Base: ba=31
Derv: ba=31, da=32

```

In an upcast you attempt to change a derived-class object into a base-class object. What you get is the base part of the derived class object. In the example we make an object of class Derv. The base class part of this object holds member data ba, which has a value of 21, and the derived part holds data member da, which has the value 22. After the cast, pBase points to the base-class part of this Derv class object, so when called upon to display itself, it prints Base: ba=21. Upcasts are fine if all you want is the base part of the object. In a downcast we put a derived class object, which is pointed to by a base-class pointer, into a derived-class pointer.

Typeid operator

```
#include <iostream>
#include <typeinfo>           //for typeid()
using namespace std;
class Base
{
    virtual void virtFunc()    //needed for typeid
{ }
};
class Derv1 : public Base
{ };
class Derv2 : public Base
{ };
void displayName(Base* pB)
{
    cout << "pointer to an object of ";    //display name of class
    cout << typeid(*pB).name() << endl;    //pointed to by pB
}
//-----
int main()
{
    Base* pBase = new Derv1;
    displayName(pBase);    //"pointer to an object of class Derv1"
    pBase = new Derv2;
    displayName(pBase);    //"pointer to an object of class Derv2"
    return 0;
}
```

Output:

```
pointer to an object of Derv1
pointer to an object of Derv2
```

reinterpret_cast

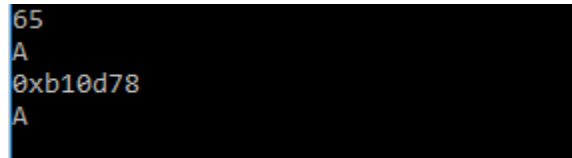
reinterpret_cast is a type of casting operator used in C++. It is used to convert one pointer of another pointer of any type, no matter either the class is related to each other or not. It does not check if the pointer type and data pointed by the pointer is same or not.

Syntax :

```
data_type *var_name =  
reinterpret_cast <data_type *>(pointer_variable);
```

```
#include <iostream>  
using namespace std;  
int main()  
{  
    int* p = new int(65);  
    char* ch = reinterpret_cast<char*>(p);  
    cout << *p << endl;  
    cout << *ch << endl;  
    cout << p << endl;  
    cout << ch << endl;  
    return 0;  
}
```

Output:



```
65  
A  
0xb10d78  
A
```

33 ERROR HANDLING

```
#include<iostream>  
using namespace std;  
int main()  
{  
    int x,y,z;  
    cout<<"Enter a dividend:"<<endl;  
    cin>>x;  
    cout<<"Enter a divisor:"<<endl; cin>>y;  
    z=x/y;  
    cout<<"the ans is"<<z<<endl;  
    return 0;  
}
```

Input $x = 1$ and $y = 0$ then $z = \text{infinity}$ which is not simply handled in general. Errors that occur at program runtime can seriously interrupt the normal flow of a program.

Some common causes of errors are

- a. division by 0, or values that are too large or small for a type no memory available for dynamic allocation
- b. errors on file access, for example, file not found attempt to access an invalid address in main memory invalid user input

Anomalies like these lead to incorrect results and may cause a computer to crash. Both of these cases can have fatal effects on your application. One of the programmer's most important tasks is to predict and handle errors. You can judge a program's quality by the way it uses error-handling techniques to counteract any potential error, although this is by no means easy to achieve. Later a concept is developed which is called exception handling.

Exception handling is based on keeping the normal functionality of the program separate from error handling. The basic idea is that errors occurring in one particular part of the program are reported to another part of the program, known as the calling environment. The calling environment performs central error handling. It proposes three concepts:

1. Try{ }
2. Throw{ }
3. Catch{ }

```
#include<iostream>
using namespace std;
class Infinity
{
};

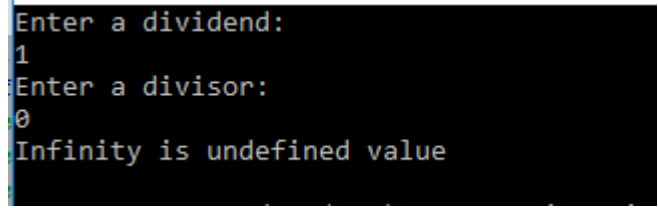
float division(int a, int b)
{
    if(b!=0)
    {
        return (float)a/b;
    }
    else
    {
        Infinity infinityerror;
        throw infinityerror;
    }
}
int main()
{
```

```

int x,y;
float z;
cout<<"Enter a dividend:"<<endl;
cin>>x;
cout<<"Enter a divisor:"<<endl;
cin>>y;
try
{
    z=division(x,y);
    cout<<"the ans is "<<z<<endl;
}
catch(Infinity& errorcheck)
{
    cout<<"Infinity is undefined value"<<endl;
}
return 0;
}

```

Output:



```

Enter a dividend:
1
Enter a divisor:
0
Infinity is undefined value

```

Explanation:

If in input $x=1$ and $y=0$ then first it enter into try block. In try block it call call division function where condition is checked. If divisor is not zero then no error takes place and gives output as: the ans is “Somevalue” else it throws an object of exception class(Infinity) to the catch block where argument has type as that of defined class(Infinity & errorcheck). Here errorcheck is an alias of an object infinityerror.

Following are main advantages of exception handling over traditional error handling.

- 1) **Separation of Error Handling code from Normal Code:** In traditional error handling codes, there are always if else conditions to handle errors. These conditions and the code to handle errors get mixed up with the normal flow. This makes the code less readable and maintainable. With try catch blocks, the code for error handling becomes separate from the normal flow.
- 2) **Functions/Methods can handle any exceptions they choose:** A function can throw many exceptions, but may choose to handle some of them. The other exceptions which are thrown, but not caught can be handled by caller. If the caller chooses not to catch them, then the exceptions are handled by caller of the caller. In C++, a function can specify the exceptions that it throws using the throw keyword. The caller of this function must handle the exception in some way (either by specifying it again or catching it).

- 3) **Grouping of Error Types:** In C++, both basic types and objects can be thrown as exception. We can create a hierarchy of exception objects, group exceptions in namespaces or classes, categorize them according to types.

Multiple Exception

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

class Infinity
{};

class charconst
{};

float division(int a, int b)
{
    if(b!=0)
    {
        return (float)a/b;
    }
    else
    {
        Infinity infinityerror;
        throw infinityerror;
    }
}

int main()
{
    int x,y;
    float z;
    cout<<"Enter a dividend:"<<endl;
    cin>>x;
    cout<<"Enter a divisor:"<<endl;
    cin>>y;

    try
    {
        z=division(x,y);
        cout<<"the ans is "<<z<<endl;
        if(z<1)
        {
            charconst char1;
            throw char1;
        }
    }
    catch(Infinity& errorcheck)
    {

```



```

        cout<<"Infinity is undefined value"<<endl;
    }
    catch(charconst& char1)
    {
        cout<<"value is less than 1"<<endl;
    }

    return 0;
}

```

Output:

```

Enter a dividend:
1
Enter a divisor:
0
Infinity is undefined value

```

```

Enter a dividend:
1
Enter a divisor:
233
the ans is 0.00429185
value is less than 1

```

Rethrowing Exception

In C++, try-catch blocks can be nested. Also, an exception can be re-thrown using “throw”.

```

#include <iostream>
using namespace std;
int main()
{
    try
    {
        try
        {
            throw 20;
        }

        catch (int n)
        {
            cout << "Handle Partially "<<endl;
            throw;                                     //Re-throwing an exception
        }
    }
}

```

```

    }
    catch (int n)
    {
        cout << "Handle remaining" << endl;
    }
    return 0;
}

```

Output:

```

    Handle Partially
    Handle remaining

```

Catch all exceptions

There is a special catch block called ‘catch all’ `catch(...)` that can be used to catch all types of exceptions. For example, in the following program, put `x=0` and `y=22` then an `int` is thrown as an exception, but there is no catch block for `int`, so `catch(...)` block will be executed.

```

#include<iostream>
using namespace std;
class Infinity
{};

float division(int a, int b)
{
    if(b!=0)
    {
        return (float)a/b;
    }
    else
    {
        Infinity infinityerror;
        throw infinityerror;
    }
}

int main()
{
    int x,y;
    float z;
    cout<<"Enter a dividend:"<<endl;
    cin>>x;
    cout<<"Enter a divisor:"<<endl;
    cin>>y;
    try
    {
        z=division(x,y);
        cout<<"the ans is "<<z<<endl;
    }
}

```

```

        if(z==0)
        {
            throw z;
        }
    }
    catch(Infinity& errorcheck)
    {
        cout<<"Infinity is undefined value"<<endl;
    }

    catch(...)
    {
        cout<<"catches all undefined exceptions"<<endl;
    }
    return 0;
}

```

Output:

```

Enter a dividend:
0
Enter a divisor:
22
the ans is 0
catches all undefined exceptions

```

Exceptions with Arguments/Exceptions Specification for Function

```

#include<iostream>
using namespace std;
void add(int a,int b)throw(int )
{
    if(b==0)
        throw b;
}
int main()
{
    try
    {
        add(1,0);
    }
    catch(int)
    {
        cout<<"argument no not matches"<<endl;
    }
    return 0;
}

```

Output: argument no not matches

Explanation: Here inside function `b==0` is matched so it throws an exception with an integer value where this exception is take by `throw(int)` of function and hence catch it.

34 Introduction to standard template library

The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as lists, stacks, arrays, etc. It is a library of container classes, algorithms and iterators. It is a generalized library and so, its components are parameterized. A working knowledge of template classes is a prerequisite for working with STL. STL has four components

1. Algorithms
2. Containers
3. Functions
4. Iterators

1. Algorithm

The header `algorithm` defines a collection of functions especially designed to be used on ranges of elements. They act on containers and provide means for various operations for the contents of the containers.

a.Sorting.

Sorting is one of the most basic functions applied on data. The prototype for sort is:
`sort(startaddress, endaddress)`

```
#include<iostream>
#include<algorithm>
using namespace std;
void show(int a[])
{
    for(int i = 0; i < 10; ++i)
        cout << '\t' << a[i];
}
int main()
{
    int a[10]= {1, 5, 8, 9, 6, 7, 3, 4, 2, 0};
    cout << "\n The array before sorting is : ";
    show(a);
    sort(a, a+10);
    cout << "\n\n The array after sorting is : ";
    show(a);
    cout<<endl;
    return 0;
}
```

Output:

```
The array before sorting is : 1      5      8      9      6      7      3      4      2      0
The array after sorting is : 0      1      2      3      4      5      6      7      8      9
```

b.Searching:

Binary search is a widely used searching algorithm that requires the array to be sorted before search is applied. The prototype for binary search is : **binary_search(startaddress, endaddress, valuetofind)**

```
#include<iostream>
#include<algorithm>
using namespace std;
void show(int a[])
{
    for(int i = 0; i < 10; i++)
    {
        cout << '\t'<< a[i];
    }
}
int main()
{
    int a[10]= {1, 5, 8, 9, 6, 7, 3, 4, 2, 0};
    cout << "\n The array before sorting is : ";
    show(a);
    sort(a, a+10);
    cout << "\n\n The array after sorting is : ";
    show(a);

    if(binary_search(a, a+10,10))
    {
        cout<<endl<<"\n\n 10 is found"<<endl;
    }
    else
    {
        cout<<"\n 10 is Not found"<<endl;
    }

    if(binary_search(a, a+10,3))
    {
        cout<<"\n 3 is found"<<endl;
    }
    else
    {
        cout<<"\n 3 is Not found"<<endl;
    }
    return 0;}

```

c.max():

```
#include<iostream>
#include<algorithm>
using namespace std;
int main()
{
    cout<<"The maximum value is"<<max(7,4);
    return 0;
}
```

Output:

The maximum value is 7

d.min():

```
#include<iostream>
#include<algorithm>
using namespace std;
int main()
{
    cout<<"The minimum value is"<<min(7,4);
    return 0;
}
```

Output:

The minimum value is 4

e.reverse():

```
#include<iostream>
#include<algorithm>
using namespace std;
void show(int a[])
{
    for(int i = 0; i < 10; ++i) cout << '\t' << a[i];
}
int main(){
    int a[10]= {1, 5, 8, 9, 6, 7, 3, 4, 2, 0};
    cout << "\n The array before sorting is : ";
```

```

show(a); reverse(a, a+10);
cout << "\n\n The array after sorting is : ";
show(a);
return 0;
}

```

4. Iterators:

Iterators are used to point at the memory addresses of STL containers. They are primarily used in sequence of numbers, characters etc. They reduce the complexity and execution time of program. Operations of iterators :-

- 1.begin() :- This function is used to return the beginning position of the container.
- 2.end() :- This function is used to return the end position of the container.
- 3.advance() :- This function is used to increment the iterator position till the specified number mentioned in its arguments.
- 4.next() :- This function returns the new iterator that the iterator would point after advancing the positions mentioned in its arguments.
- 5.prev() :- This function returns the new iterator that the iterator would point after decrementing the positions mentioned in its arguments.

**// C++ code to demonstrate the working of
// iterator, begin() and end()**

```

#include<iostream>
#include<iterator> // for iterators
#include<vector> // for vectors
using namespace std;
int main()
{
vector<int> ar = { 1, 2, 3, 4, 5 };
// Declaring iterator to a vector
vector<int>::iterator ptr;
// Displaying vector elements using begin() and end()
cout << "The vector elements are : ";
for (ptr = ar.begin(); ptr < ar.end(); ptr++)
cout << *ptr << " ";
return 0;
}

```

Output:

The vector elements are : 1 2 3 4 5

2. Containers

Containers or container classes store objects and data. There are in total seven standard “first-class” container classes and three container adaptor classes and only seven header files that provide access to these containers or container adaptors.

Sequence Containers: implement data structures which can be accessed in a sequential manner.

Vector, list, deque, arrays

a. Vector:

Vectors are same as dynamic arrays with the ability to resize itself automatically when an element is inserted or deleted, with their storage being handled automatically by the container. Vector elements are placed in contiguous storage so that they can be accessed and traversed using iterators. In vectors, data is inserted at the end. Inserting at the end takes differential time, as sometimes there may be a need of extending the array. Removing the last element takes only constant time, because no resizing happens. Inserting and erasing at the beginning or in the middle is linear in time.

Certain functions are associated with vector :

- 1.size() – Returns the number of elements in the vector.
- 2.max_size() – Returns the maximum number of elements that the vector can hold.

```
#include <iostream>
#include <vector>
using namespace std;
void show(vector<int>g1){
    cout<<"Size of vector is:"<<g1.size()<<"\n";
    for (int i = 0; i <g1.size(); i++){
        cout<<"\t"<<g1[i];
    }
}

int main()
{
    vector <int> g1;
    for (int i = 1; i <= 5; i++){ g1.push_back(i);
    }
    show(g1); return 0;
}
```

Output:

Size of vector is:5 1 2 3 4 5

b. List

Lists are sequence containers that allow non-contiguous memory allocation. As compared to vector, list has slow traversal, but once a position has been found, insertion and deletion are quick. Normally, when we say a List, we talk about doubly linked list. For implementing a singly linked list, we use forward list.

Functions used with List :

- front() – Returns reference to the first element in the list
- back() – Returns reference to the last element in the list
- push_front(g) – Adds a new element ‘g’ at the beginning of the list
- push_back(g) – Adds a new element ‘g’ at the end of the list
- pop_front() – Removes the first element of the list, and reduces size of the list by 1
- pop_back() – Removes the last element of the list, and reduces size of the list by 1

begin() – Returns an iterator pointing to the first element of the list
 end() – Returns an iterator pointing to the theoretical last element which follows the last element
 empty() – Returns whether the list is empty(1) or not(0)
 insert() – Inserts new elements in the list before the element at a specified position
 erase() – Removes a single element or a range of elements from the list
 assign() – Assigns new elements to list by replacing current elements and resizes the list
 remove() – Removes all the elements from the list, which are equal to given element
 reverse() – Reverses the list

```

#include <iostream>
#include <list>
#include <iterator>
using namespace std;

//function for printing the elements in a list
void showlist(list<int> g)
{
  list<int> :: iterator it;
  for(it = g.begin(); it != g.end(); ++it)
    cout << 't' << *it;
  cout << '\n';
}

int main()
{
  list<int> gqlist1, gqlist2;
  for (int i = 0; i < 10; ++i)
  {
    gqlist1.push_back(i * 2);
    gqlist2.push_front(i * 3);
  }
  cout << "\nList 1 (gqlist1) is : ";
  showlist(gqlist1);
  cout << "\nList 2 (gqlist2) is : ";
  showlist(gqlist2);
  cout << "\ngqlist1.front() : " << gqlist1.front();
  cout << "\ngqlist1.back() : " << gqlist1.back();
  cout << "\ngqlist1.pop_front() : ";
  gqlist1.pop_front();
  showlist(gqlist1);
  cout << "\ngqlist2.pop_back() : ";
  gqlist2.pop_back();
  showlist(gqlist2);
  cout << "\ngqlist1.reverse() : ";
  gqlist1.reverse();
  showlist(gqlist1);
  cout << "\ngqlist2.sort() : ";
  gqlist2.sort();
  showlist(gqlist2);
}

```

```

        return 0;
    }

```

c. Queue:

Queues are a type of container adaptors which operate in a first in first out (FIFO) type of arrangement. Elements are inserted at the back (end) and are deleted from the front.

The functions supported by queue are :

`empty()` – Returns whether the queue is empty.

`size()` – Returns the size of the queue.

`queue::swap()` in C++ STL: Exchange the contents of two queues but the queues must be of same type, although sizes may differ.

`queue::emplace()` in C++ STL: Insert a new element into the queue container, the new element is added to the end of the queue.

`queue::front()` and `queue::back()` in C++ STL– `front()` function returns a reference to the first element of the queue. `back()` function returns a reference to the last element of the queue.

`push(g)` and `pop()` – `push()` function adds the element 'g' at the end of the queue. `pop()` function deletes the first element of the queue.

```

#include <iostream>
#include <queue>

using namespace std;

void showq(queue <int> gq)
{
    queue <int> g = gq;
    while (!g.empty())
    {
        cout << '\t' << g.front();
        g.pop();
    }
    cout << '\n';
}

int main()
{
    queue <int> gquiz;
    gquiz.push(10);
    gquiz.push(20);
}

```

```

gquiz.push(30);

cout << "The queue gquiz is : ";
showq(gquiz);

cout << "\ngquiz.size() : " << gquiz.size();
cout << "\ngquiz.front() : " << gquiz.front();
cout << "\ngquiz.back() : " << gquiz.back();

cout << "\ngquiz.pop() : ";
gquiz.pop();
showq(gquiz);

return 0;
}

```

Output:

The queue gquiz is : 10 20 30

```

gquiz.size() : 3
gquiz.front() : 10
gquiz.back() : 30
gquiz.pop() :   20   30

```

d.stack

Stacks are a type of container adaptors with LIFO (Last In First Out) type of working, where a new element is added at one end and (top) an element is removed from that end only.

The functions associated with stack are:

empty() – Returns whether the stack is empty – Time Complexity : O(1)

size() – Returns the size of the stack – Time Complexity : O(1)

top() – Returns a reference to the top most element of the stack – Time Complexity : O(1)

push(g) – Adds the element 'g' at the top of the stack – Time Complexity : O(1)

pop() – Deletes the top most element of the stack – Time Complexity : O(1)

```

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

void showstack(stack <int> s)
{
    while (!s.empty())
    {
        cout << '\t' << s.top();
        s.pop();
    }
    cout << '\n';
}

```

```

int main ()
{
    stack <int> s;
    s.push(10);
    s.push(30);
    s.push(20);
    s.push(5);
    s.push(1);

    cout << "The stack is : ";
    showstack(s);

    cout << "\ns.size() : " << s.size();
    cout << "\ns.top() : " << s.top();

    cout << "\ns.pop() : ";
    s.pop();
    showstack(s);

    return 0;
}

```

Output:

The stack is : 1 5 20 30 10

s.size() : 5

s.top() : 1

s.pop() : 5 20 30 10

35 Basic features of Object Oriented Programming (Exam Point of view)

1. Class:

Class is an interface to create an object. Class has functions and attributes bind together. Function defines the behavior of an object/s and attribute defines the property of an object/s which it holds. From single class any number of an object can be created.

```
class Car
{
    int weight;
    int height;
    string color;

    void start()
    {
        cout<<"Start"<<endl;
    }
    void run()
    {
        cout<<"Run"<<endl;
    }
}
```

Here, Class Car has attributes as weight, height, color and function as start () and run ().

2. Object:

Object is a run time entity. Function and attributes comes into play only after creating an object. This refers that features are defined on class where access of these features are possible only after creating an object.

Ex

Car lamborghini, ferari;

Here Lamborghini and ferari are objects of a class Car. You can use start and run function only after real manufacturing of lamborghini, ferari. Note: For easy visualization, compare class with drawing of a car. By using single drawing many car can be manufactured which represents an object in C++.

3. Data Hiding:

Data Hiding is the process of hiding an information (data) from an unauthentic users. Information is provided only after the proper validation. If validation fails then information is not shown.

Ex

- a. In ATM machine, people are able to access account information only after proper validation of pin number used for respective ATM card. If pin number goes wrong at the time of validation then information is secure.
- b. In Facebook, Gmail and others social sites, Password and ID entered must have to match with the previously stored Password and ID, resides in database to view Information.

Advantages:

It provides security.

4. Abstraction:

Abstraction says the detail mechanism is not necessary, just basic knowledge of using system's features is sufficient for proper utilization of system.

Ex

To use ATM machine people does not have to learn how ATM card is validated after entering into machine, which language is used to validate data, which databases either SQL or Oracle is implemented in bank to store data. Just basic knowledge of how to use ATM machine is sufficient to access account information. Here detail mechanism is hidden and only basic feature is provided and it's called an abstraction.

Advantages:

It provides simplicity.

5. Encapsulation:

Encapsulation is just like capsule. It combines the features of both data hiding and abstraction in single unit.

Ex

In ATM system, GUI is integrated with the database through certain programming language. If GUI is needed to be update then without customizing all the features of database and language, it can be updated.

Advantages:

It makes customization easy.

6. Inheritance:

```
class P
{
    250 functions()
}
class Q
{
    250 functions()
}
class R
{
    250 functions()
}
```

Suppose, 3 class are there each having 250 number of functions. Let 200 number of functions are common among them .we do not know about the inheritance concept yet then the total number of functions to be written = 750. Say, 750 functions take 90 hour of time to code.

Inheritance concept say, if common functions are there among classes then place them in one class. After that use such policy, so other classes can easily access those common functions.

```

class common{
200 functions()
};
class P:common{
50 functions()
};
Class Q:common{
50 functions()
};
Class R:common{
50 functions()
}

```

The total number of functions to be written = 350. Total time to write 350 function takes 42 hour of time.

Advantages:

It helps to reuse code and thus reduce time to code.

7. Polymorphism:

Polymorphism refers to many forms. This means under different situation same thing can behave differently. We human being is best example of polymorphism.

In university a person is a student, In home same person can be son/daughter of parent or husband/wife or father/mother of their child. According to location, relation changes for same person.

In OOP, same concept applies in polymorphism. Function Overloading and Dynamic Binding is an example of polymorphism.

Ex

```

#include<iostream>
using namespace std;
void m1(int i)
{
cout<<"int-args:"<<endl;
}
void m1(float f)
{
cout<<"float-args:"<<endl;
}
int main()
{
m1(2);
m1(2.3f);
return 0;
}

```

Output: int-args: float-args:

Here, both functions have same name but depending upon type of argument it gives different output. If int argument is passes then it shows “int-args” as output. If float argument is passes then it shows “float-args” as output.

Advantages:
It provides flexibility.

35.1 Procedure Oriented Programming vs Object Oriented Programming(ref: wiki)

Divided Into	In POP, program is divided into small parts called functions.	In OOP, program is divided into parts called objects.
Importance	In POP, Importance is not given to data but to functions as well as sequence of actions to be done.	In OOP, Importance is given to the data rather than procedures or functions because it works as a real world.
Approach	POP follows Top Down approach.	OOP follows Bottom Up approach.
Access Specifiers	POP does not have any access specifier.	OOP has access specifiers named Public, Private, Protected, etc.
Data Moving	In POP, Data can move freely from function to function in the system.	In OOP, objects can move and communicate with each other through member functions.
Expansion	To add new data and function in POP is not so easy.	OOP provides an easy way to add new data and function.
Data Access	In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the system.	In OOP, data can not move easily from function to function, it can be kept public or private so we can control the access of data.
Data Hiding	POP does not have any proper way for hiding data so it is less secure.	OOP provides Data Hiding so provides more security.
Overloading	In POP, Overloading is not possible.	In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.
Examples	Example of POP are: C, VB, FORTRAN, Pascal.	Example of OOP are : C++, JAVA, VB.NET, C#.NET.

36. File handling

Introduction:

All programs we looked earlier:

-> Input data from the keyboard.

-> Output data to the screen.

-> Output would be lost as soon as we exit from the program.

Q. How do we store data permanently?

-> We can use secondary storage device.

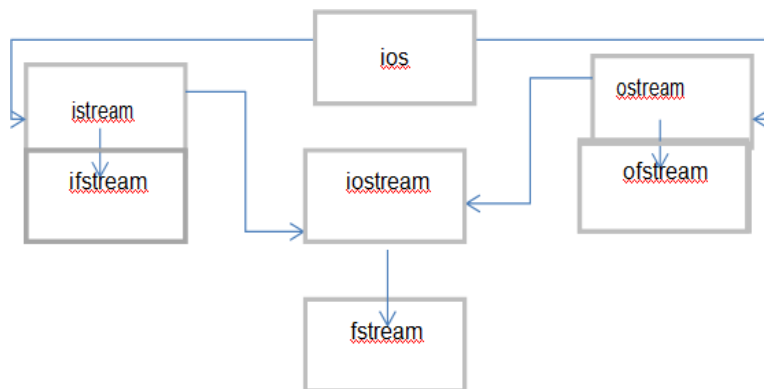
-> Data is packaged up on the storage device as data structures called files

Q. Why use File Handling

-> For permanent storage.

-> The transfer of input – data or output – data from one computer to another can be easily done by using files.

I/O in C++ uses streams and the operations on files are performed by using streams too.



A stream is a general name given to the flow of data. The **ios class** is the granddaddy of all the stream classes, and contains the majority of the features you need to operate C++ streams. The three most important features are the formatting flags, the error status flags, and the file operation mode.

The istream and ostream classes are derived from ios and are dedicated to input and output respectively. The istream class contains such functions as get(), getline(), read() and the overloaded extraction(>>) operators, while ostream contains put(), write() and the insertion(<<) operators.

The iostream class is derived from both istream and ostream by multiple inheritance. Classes derived from it can be used with devices, such as disk files, that may be opened for both input and output at the same time.

Ofstream - stream used for output to files. Contains open() with default output mode. Inherits

put(), seekp(), teelp() and write() function from ostream.

Ifstream - stream used for input from files. Contains open() with default input mode. Inherits the functions get(), getline(), read(), seekg() and tellg() function from istream.

Fstream - stream for both input and output operations. Contains open() with default input mode. Inherits all function from istream and ostream classes through iostream.

- a. Ios Functions:** The ios class contains a number of functions that you can use to set the formatting flags and perform other tasks. Table shows most of these functions, except those that deal with errors, which we'll examine separately. **These functions are called for specific stream objects using the normal dot operator. (Ref:wiki)**

Functions	Purpose
ch = fill();	Return the fill character (fills unused part of field; default is space)
fill(ch);	Set the fill character
p = precision();	Get the precision (number of digits displayed for floating-point)
precision(p);	Set the precision
w = width();	Get the current field width (in characters)
width(w);	Set the current field width
setf(flags);	Set specified formatting flags (for example, ios::left)
unsetf(flags);	First clear field, then set flags

1. width()

```
#include<iostream>
using namespace std;
int main()
{
    int a=2;
    cout<<a<<endl;
    cout.width(2);
    cout<<a<<endl;
    return 0;
}
```

Output:

2
2

```
#include<iostream>
using namespace std;
int main()
{
int a=2,b=534;
cout.width(2);
cout<<a;
cout.width(4);
cout<<b;
return 0;
}
```

Output:

2 534

2.fill()

```
#include<iostream>
using namespace std;
int main()
{
int a=2,b=534;
cout.fill('*');
cout.width(2);
cout<<a;
cout.fill('*');
cout.width(4);
cout<<b;
return 0;
}
```

Output:

*2*534

What happens if cout.width(3); is used in place of cout.width(4); ?

```
#include<iostream>
using namespace std;
int main(){
int a=2,b=534;
cout.fill('*');
cout.width(2);
cout<<a;
```

```
cout.fill('*');
cout.width(3);
cout<<b;
return 0;
}
```

Output:
*2534

3.setf()

```
#include<iostream>
using namespace std;
int main(){
cout.setf(ios::hex,ios::basefield);
cout.setf(ios::showbase);
cout<< 100;
return 0;
}
```

Output:
0x64

- b. Formatting flags:** Formatting flags are set of enum definitions in ios. They act as on/off switches that specify choices for various aspects of input and output format an operation.
- ios Formatting Flags

Flag	Meaning
skipws	Skip (ignore) whitespace on input
left	Left-adjust output
right	Right-adjust output
internal	Use padding between sign or base indicator and
dec	Convert to decimal
oct	Convert to octal
hex	Convert to hexadecimal
boolalpha	Convert bool to “true” or “false” strings
showbase	Use base indicator on output (0 for octal, 0x for hex) showpoint point on output
uppercase	Use uppercase X, E, and hex output letters (ABCDEF)—the default is lowercase
showpos	Display + before positive integers
scientific	Use exponential format on floating-point output
fixed	Use fixed format on floating-point output [912.34]
unitbuf	Flush all streams after insertion
stdio	Flush stdout, stderr after insertion

```

#include <iostream>
using namespace std;
int main ()
{
    cout << showbase << hex;
    cout << uppercase << 77 << '\n';
    cout << nouppercase << 77 << '\n';
    return 0;
}

```

Output:
0x4D
0x4d

c. Manipulators: Manipulators are formatting instructions inserted directly into a stream.

1. Ios Manipulators with No-argument:

Manipulator	Purpose
ws	Turn on whitespace skipping on input
dec	Convert to decimal
oct	Convert to octal
hex	Convert to hexadecimal
endl	Insert newline and flush the output stream
ends	Insert null character to terminate an output string
flush	Flush the output stream
lock	Lock file handle
unlock	Unlock file handle

1.

```

#include <iostream>
using namespace std;
int main ()
{
    int n = 70;
    cout << dec << n << '\n';
    cout << hex << n << '\n';
    cout << oct << n << '\n';
    return 0;
}

```

Output:
70
46
106

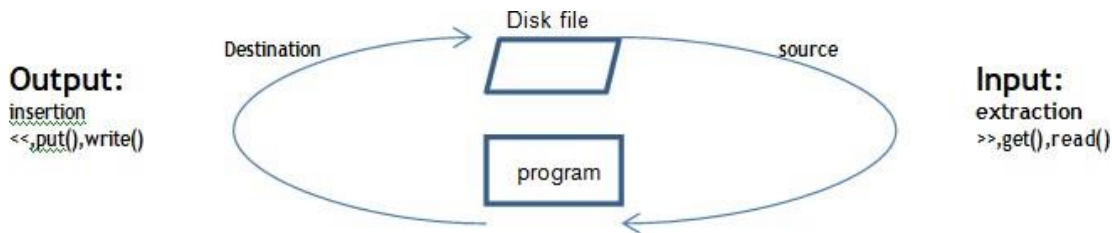
2. Ios Manipulators with arguments:

Manipulator	Argument	Purpose
setw()	field width (int)	Set field width for output
setfill()	fill character (int)	Set fill character for output(default is a space)
setprecision()	precision (int)	Set precision (number of digits displayed)
setiosflags()	formatting flags (long)	Set specified flags
resetiosflags()	formatting flags (long)	Clear specified flags

```
#include <iostream>
#include <iomanip>
using namespace std;
int main () {
    double f =3.14159;
    cout << setprecision (5) << f << endl;
    cout << setprecision (9) << f << endl;
    return 0;
}
```

Output:
3.1416
3.14159000

Figure of file input and output:



1. The istream class: The istream class, which is derived from ios, performs input specific activities, or extraction.

istream Functions

Function	Purpose
>>	Formatted extraction for all basic (and overloaded) types.
get(ch);	Extract one character into ch.
get(str)	Extract characters into array str, until '\n'.
get(str, MAX)	Extract up to MAX characters into array.
get(str, DELIM)	Extract characters into array str until specified delimiter (typically '\n'). Leave delimiting char in stream.
get(str, MAX, DELIM)hello	Extract characters into array str until MAX characters or the DELIM character. Leave delimiting char in stream.
getline(str, MAX,DELIM)	Extract characters into array str, until MAX characters or the DELIM

peek(ch)	character. Extract delimiting character.
putback(ch)	Read one character, leave it in stream.
ignore(MAX, DELIM)	Insert last character read back into input stream. Extract and discard up to MAX characters until (and including) the specified delimiter (typically ‘\n’).
count = gcount()	Return number of characters read by a (immediately preceding) call to get(), getline(), or read().
read(str, MAX)	For files—extract up to MAX characters into str, until EOF.
seekg()	Set distance (in bytes) of file pointer from start of file.
seekg(pos, seek_dir)	Set distance (in bytes) of file pointer from specified place in file. seek_dir can be ios::beg, ios::cur, ios::end.
pos = tellg(pos)	Return position (in bytes) of file pointer from start of file.

```
#include<iostream>
using namespace std;
int main()
{
    char c;
    while ( c != '\n' )
    {
        cin.get(c);
        cout << "-";
        cout<<c;          //-----display the entered character..
    }
    cout << endl; return 0;
}
```

Output:
-h-e-l-l-o- , if you enter hello

2.The ostream class: The ostream class handles output or insertion activities.

ostream Functions

Function	Purpose
<<	Formatted insertion for all basic (and overloaded) types.
put(ch)	Insert character ch into stream.
flush()	Flush buffer contents and insert newline.
write(str, SIZE)	Insert SIZE characters from array str into file.
seekp(position)	Set distance in bytes of file pointer from start of file.
seekp(position,seek_dir)	Set distance in bytes of file pointer, from specified place in file. seek_dir can be ios::beg, ios::cur,ios::end
pos = tellp()	Return position of file pointer, in by

```
#include<iostream>
using namespace std;
int main()
{
    char c;
```

```

while ( c != '\n' )
{
    cin.get(c);
    cout << "-";
    cout.put(c);    //-----display the entered character..
}
cout << endl; return 0;
}
Output:
-H-e-l-l-o-      , if you enter hello

```

Q. How to test Stream Errors in C++?

The objects cin and cout works well in normal conditions for input and output respectively. However, there can be several abnormal situation in I/O process such as entering string instead of digit, pressing enter key without entering value or some hardware failure. When errors occurred during input or output operations, the errors are reported by stream state. Every stream (istream or ostream) has a state associated with it. Error conditions are detected and handled by testing the state of the stream.

Following are the member functions of ios class that are used to test the state of the stream.

bool ios::good() : This function returns true when everything is okay, that is , when there is no error conditions.

bool ios::eof(): This function returns true if the input operation reached end of input sequence.

bool ios::bad(): This function returns true if the stream is corrupted and no read/write operation can be performed. For example in an irrecoverable read error from a file.

bool ios::fail(): This functions returns true if the input operation failed to read the expected characters, or that an output operation failed to generate the desired characters. When in fail condition the stream may not be corrupted.

void ios::clear(ios::iostate f = ios::goodbit): This function clears all the flag if no argument is supplied. It can also be used to clear a particular state flag if supplied as argument.

ios::iostate ios::rdstate(): This function returns the state of a stream object of type iostate.

void ios::setstate(ios::iostate f): This function adds the flag in argument to the iostate flags.

ios::goodbit : This state flag indicates that there is no error with streams. In this case the status variables has value 0.

ios::eofbit: This state flag indicates that the input operation reached end of input sequence.

ios::badbit: This state flag indicates that the stream is corrupted and no read/write operation can be performed.

ios::failbit: indicates that input/output operation failed. The fail condition may not be because of corrupted stream.

File Handling

Sometime, it is important to store the information entered by the user into the file for further use. After storing the information into the file, later you can retrieve that information from that file. File helps in storing the information permanently.

A file itself is a bunch of bytes stored on some storage devices like taps, or magnet disk etc. In C++, the file input/output operations are performed through a component header file `fstream` of C++ standard library.

In C++, a file at its lowest level - is interpreted simply as a sequence, or stream, of bytes. At the user level - consists of a sequence of intermixed data types such as characters, arithmetic values, class objects.

The `fstream` library predefines a set of operations for handling file related input and output operation. It defines certain class that helps in performing the file input and output operations. C++ provides the following classes to perform output and input of characters to/from files:

ofstream: Stream class to write on files

ifstream: Stream class to read from files

fstream: Stream class to both read and write from/to files.

These classes are derived directly or indirectly from the classes `istream`, and `ostream`. We have already used objects whose types were these classes: `cin` is an object of class `istream` and `cout` is an object of class `ostream`. Therefore, we have already been using classes that are related to our file streams. And in fact, we can use our file streams the same way we are already used to use `cin` and `cout`, with the only difference that we have to associate these streams with physical files.

Character Input/Output in Files: (program of file handling by: Er.Bikal Adhkari, Lecturer)

WAP in CPP to write the characters entered by the user to a file until s/he presses enter key

```
#include<iostream>
#include<fstream>

using namespace std;

int main()
{
    char ch;
    ofstream out("student.txt",ios::out);
    cout<<"Start writing the characters..."<<endl;
    while((ch=cin.get())!='\n')
    {
        out.put(ch);
    }
    cout<<"File written!"<<endl;
    return 0;
}
```

WAP in CPP to read the file using the concept of character input from file. Also check for error while opening file.

```
#include<iostream>
#include<fstream>
#include<cstdlib>

using namespace std;

int main()
{
    char ch;
    ifstream in("student.txt",ios::in);
    if(in.fail()) //Error handling while opening a file
    {
        cout<<"Error opening file"<<endl;
        cout<<"Exiting..."<<endl;
        exit(1);
    }
    cout<<"Contents of the file:"<<endl;
    while(in.get(ch))
    {
        cout<<ch;
    }
    in.close();
    return 0;
}
```

WAP to copy contents of one file into another. Your program should input the source filename from the user and duplicate the contents for destination filename entered from the user using the concept of character IO for files.

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
    char ch,sfname[20],dfname[20];
    cout<<"Enter source file name:";
    cin>>sfname;
    cout<<"Enter destination file name:";
    cin>>dfname;
    ifstream in(sfname,ios::in);
```

```

ofstream out(dfname,ios::out);
while(in.get(ch))
{
out.put(ch);
}
cout<<"File copied"<<endl;
in.close();
out.close();
return 0;
}

```

Formatted Input/Output in Files:

WAP in CPP to illustrate the concept of formatted output to file. Your program should input details of n student such as name, address, roll number and telephone from the user and write the entered details into a file.

```

#include<iostream>
#include<fstream>
using namespace std;
int main(void)
{
char name[20],addr[20];
int roll;
long int tel;
ofstream out("student.txt",ios::out);
cout<<"Enter number of records to be stored";
int a;
cin>>a;
for(int i=1;i<=a;i++){
cout<<"Enter name:";cin>>name;
cout<<"Enter roll:";cin>>roll;
cout<<"Enter address:";cin>>addr;
cout<<"Enter tel:";cin>>tel;
out<<name<<"\t"<<roll<<"\t"<<addr<<"\t"<<tel<<endl;
}
cout<<"The file is completely written!";
return 0;
}

```

WAP in CPP to read the details of students such as name, roll, address and telephone number stored in a file using the concept of formatted input from the file.

```
#include<iostream>
#include<fstream>

using namespace std;

int main(void)
{
    char line[100];
    char name[20],addr[20];
    int roll;
    long int tel;
    ifstream in("student.txt",ios::in);
    in>>name>>roll>>addr>>tel;
    while(in){
        cout<<name<<'\t'<<roll<<'\t'<<addr<<'\t'<<tel<<endl;
        in>>name>>roll>>addr>>tel;
    }
    in.close();
    return 0;
}
```

WAP in CPP to write details of n students into a file and read the stored details from that file in a same program

```
#include<iostream>
#include<fstream>

using namespace std;

int main(void)
{
    char name[20],addr[20];
    int roll;
    long int tel;

    fstream inout;
    inout.open("student.txt",ios::out);

    /*Code to write to file*/
```

```

cout<<"Enter number of records to be stored";
int a;
cin>>a;
for(int i=1;i<=a;i++){
cout<<"Enter name:";cin>>name;
cout<<"Enter roll:";cin>>roll;
cout<<"Enter address:";cin>>addr;
cout<<"Enter tel:";cin>>tel;
inout<<name<<"\t"<<roll<<"\t"<<addr<<"\t"<<tel<<endl;
}
cout<<"The file is completely written!\n";
inout.close();

inout.open("student.txt",ios::in);
/*Code to Read from file*/
cout<<"\nNow Reading File\n";
inout>>name>>roll>>addr>>tel;
while(inout){
cout<<name<<"\t"<<roll<<"\t"<<addr<<"\t"<<tel<<endl;
inout>>name>>roll>>addr>>tel;
}
inout.close(); //Close File
return 0;
}

```

FILE IO using class and object concept

WAP in CPP to write name, roll, marks and address of n students entered by the user into a file using class and object concepts

```

#include<iostream>
#include<fstream>
#include<iomanip>

```

```

using namespace std;

```

```

class student
{
private:
char name[20];
char address[50];
int roll;
float marks;

```

```

public:

```

```

void getdata()
{
    cout<<"Enter the name:";cin>>name;
    cout<<"Enter the address:";cin>>address;
    cout<<"Enter the roll no:";cin>>roll;
    cout<<"Enter the marks:";cin>>marks;
}

void putdata()
{

    cout<<setw(10)<<name<<setw(15)
    <<address<<setw(10)<<roll
    <<setw(2)<<marks<<endl;
}
};

int main(void)
{
    student s;
    fstream inout("student.txt",ios::out);
    int n;
    cout<<"Enter number of students:";
    cin>>n;

    for(int i=0;i<n;i++)
    {
        s.getdata();
        inout.write((char *)&s,sizeof(s));
    }
    cout<<"File Written!!!";
    inout.close();
    return 0;
}

```

WAP in CPP to read the details of students such as name, roll, marks and address from a file and display it in console using the concept of class and object.

```

#include<iostream>
#include<fstream>
#include<iomanip>

using namespace std;

```

```

class student
{
private:
char name[20];
char address[50];
int roll;
float marks;

public:
    void getdata()
    {
        cout<<"Enter the name:";cin>>name;
        cout<<"Enter the address:";cin>>address;
        cout<<"Enter the roll no:";cin>>roll;
        cout<<"Enter the marks:";cin>>marks;
    }

    void putdata()
    {

        cout<<setw(10)<<name<<setw(15)
        <<address<<setw(10)<<roll
        <<setprecision(2)<<setw(10)
        <<marks<<endl;
    }
};

int main(void)
{
    student s;
    fstream inout("student.txt",ios::in);
    while(inout.read((char *)&s,sizeof(s)))
    {
        s.putdata();
    }
    inout.close();
}

```

Complete Input/Output Operations on File

WAP in CPP to perform complete file IO operations such as writing to file, reading from file, adding a record to file, searching a record from file, modifying the record from file, deleting a record from file and counting the file size and number of objects or records within the file. Use class and object concept.

```
#include<iostream>
#include<cstdlib>
#include<fstream>
#include<iomanip>
using namespace std;
class student
{
private:
int roll;
float marks;
char name[50],address[50];

public:
void getdata()
{
cout<<"Enter name";cin>>name;
cout<<"Enter roll";cin>>roll;
cout<<"Enter marks";cin>>marks;
cout<<"Enter address";cin>>address;
}

void putdata()
{
cout<<setw(10)<<name<<setw(15)<<roll<<setw(20)<<marks<<setw(30)<<address<<"\n";
}

int getroll()
{
return roll;
}
};

int main()
{
student std;
fstream inout;
```



```

int m,roll,object,isfound;
cout<<"What do you like to do?"<<endl;
cout<<"1.Write records into file:"<<endl;
cout<<"2.Read records from the file:"<<endl;
cout<<"3.Update record in the file:"<<endl;
cout<<"4.Search record of a student:"<<endl;
cout<<"5.Modify record of a student:"<<endl;
cout<<"6.Delete record of a student:"<<endl;
cout<<"7.count the number of objects in file and total file size"<<endl;
cin>>m;
switch(m)
{
case 1://Write into the file
inout.open("REC.txt",ios::out);
int z;
cout<<"Enter the number of students"<<"\n";
cin>>z;
for(int i=1;i<=z;i++)
{
std.getdata();
inout.write((char*)&std,sizeof(std));
}
cout<<"File Written!"<<endl;
inout.close();
break;

case 2://Read the file
inout.open("REC.txt",ios::in);
cout<<"Current contents of file"<<endl;
cout<<setw(10)<<"name"<<setw(15)<<"roll"<<setw(20)<<"marks"<<setw(
30)<<"address"<
<"\n";
while(inout.read((char *)&std,sizeof(std)))
{
std.putdata();
}
inout.close();
break;

case 3://Update the record of the student
inout.open("REC.txt",ios::app);
cout<<"Add Student Record:"<<endl;
std.getdata();
inout.write((char *)&std,sizeof(std));

```

```

cout<<"Record Added!!!"<<endl;
inout.close();
break;

case 4://Search the record of the student
inout.open("REC.txt",ios::in);
isfound=0;
cout<<"Enter Roll number:"<<endl;
cin>>roll;
inout.seekg(0,ios::beg);
while(inout.read((char *)&std,sizeof(std)))
{
if(std.getroll()==roll){
isfound=1;
cout<<"Searched record!:"<<endl;
cout<<setw(10)<<"name"<<setw(15)<<"roll"<<setw(20)<<"marks"<<setw(
30)<<"address"<
<<"\n";
std.putdata();
break;
}
}
if(isfound==0)
cout<<"The record with the roll number "<<roll<<" is not found!"<<endl;
inout.close();
break;

```

```

case 5://Modify Record
inout.open("REC.txt",ios::out|ios::in|ios::ate);
int location;
object=isfound=0;
cout<<"Enter Roll number:"<<endl;
cin>>roll;
inout.seekg(0,ios::beg);

while(inout.read((char *)&std,sizeof(std)))
{
++object;
if(std.getroll()==roll)
{
isfound=1;
location=(object-1)*sizeof(std);
inout.seekp(location,ios::beg);
cout<<"Enter new data"<<endl;

```

```

std.getdata();
inout.write((char *)&std,sizeof(std))<<flush;
cout<<"Record Modified!!!"<<endl;
break;
}
}
if(isfound==0)
cout<<"The record with the roll number "<<roll<<"is not found!"<<endl;
inout.close();
break;

```

```

case 6://Delete Record
inout.open("REC.txt",ios::in);
student st[48];
int i,j;
i=0;
cout<<"Enter Roll number:"<<endl;
cin>>roll;
inout.seekg(0,ios::beg);
while(inout.read((char *)&st[i],sizeof(st[i])))
{
++i;
}
j=i;
inout.close();
inout.open("REC.txt",ios::out);
for(i=0;i<j;i++)
{
if(st[i].getroll()!=roll)
{
inout.write((char *)&st[i],sizeof(st[i]));
}
}
cout<<"Record Deleted!!!"<<endl;
inout.close();

break;

```

```

case 7://Calculates file size and the number of Objects
int filesize;
inout.open("REC.txt",ios::in);
inout.seekg(0,ios::end);
filesize =inout.tellg();
cout<<"The total file size is:"<<filesize<<" Bytes"<<endl;

```

```

        cout<<"And number of objects:"<<(filesize/sizeof(std))<<endl;
        inout.close();
        break;

    default:
        cout<<"wrong choice:";
        break;
    }
    return 0;
}

```

Overloading insertion and extraction operator for file input/output

WAP to write data of n students into file and read the corresponding file by overloading insertion and extraction operator

```

#include<iostream>
#include<fstream>

using namespace std;

class student
{
public:
    char name[20];
    int roll;
    float marks;
public:
    friend istream& operator >>(istream&,student&);
    friend ostream& operator <<(ostream&,const student&);
    friend ifstream& operator >>(ifstream&,student&);
    friend ofstream& operator <<(ofstream&,const student&);
};

istream& operator >>(istream &in,student &s)
{

    cout<<"Enter name, roll and marks of student:"<<endl;
    in>>s.name;
    in>>s.roll;
    in>>s.marks;
    return in;
}

ostream& operator <<(ostream &out,const student &s)

```

```

{
out<<s.name<<s.roll<<s.marks<<endl;
return out;
}

ifstream& operator >>(ifstream &fin,student &s)
{
fin>>s.name>>s.roll>>s.marks;
return fin;
}

ofstream& operator <<(ofstream &fout,student &s)
{
fout<<s.name<<s.roll<<s.marks<<endl;
return fout;
}

int main()
{
int n,i,j=0;
student s[48];
ofstream out("student.txt",ios::out);
cout<<"Enter the number of students:";
cin>>n;
for(i=0;i<n;i++)
{
cin>>s[i];
out<<s[i];
}
cout<<"File written!"<<endl;
out.close();
cout<<"Now reading the contents of the
file"<<endl; ifstream
in("student.txt",ios::in); while(in)
{
in>>s[j];
cout<<s[j];
cout<<"\n";
fflush(stdin);
j++;
}
in.close();
return 0;
}

```

Q. WAP for the addition of two 2*2 matrices using operator overloading.

```
#include<iostream>
using namespace std;
class Matrix
{
    private:
        int mata[2][2];

    public:
        Matrix()
        {
            for(int i=0;i<2;i++)
            {
                for(int j=0;j<2;j++)
                {
                    mata[i][j]=0;
                }
            }
        }

        void setMatrix()
        {
            for(int i=0;i<2;i++)
            {
                for(int j=0;j<2;j++)
                {
                    cout<<"Enter Mata "<<i <<j <<" element:\t"<<endl;
                    cin>>mata[i][j];
                }
            }
        }

        Matrix operator+(Matrix M2)
        {
            Matrix matresult;
            for(int i=0;i<2;i++)
            {
                for(int j=0;j<2;j++)
                {
                    matresult.mata[i][j]=mata[i][j]+M2.mata[i][j];
                }
            }
            return matresult;
        }

        void showMatrix(Matrix M4)
        {
            cout<<endl;

            for(int i=0;i<2;i++)
            {
                for(int j=0;j<2;j++)
                {
```

```

        cout<<M4.mata[i][j]<<"\t";
    }
    cout<<endl;
}

}

};

int main(void)
{
    Matrix M1,M2;
    M1.setMatrix();
    M2.setMatrix();
    Matrix M3 = M1+M2;
    M1.showMatrix(M3);
    return 0;
}

```

Q.WAP to add two time by passing object as an argument and returning object as an argument.

```

#include<iostream>
using namespace std;
class Time
{
    int hr,min,sec;
public:
    void get()
    {
        cin>>hr>>min>>sec;
    }
    void disp()
    {
        cout<<hr<<":"<<min<<":"<<sec;
    }
}

Time sum(Time t1,Time t2)
{
    Time t3;
    t3.sec=t1.sec+t2.sec;
    t3.min=t3.sec/60;
    t3.sec=t3.sec%60;
    t3.min=t3.min+t1.min+t2.min;
    t3.hr=t3.min/60;
    t3.min=t3.min%60;
    t3.hr=t3.hr+t1.hr+t2.hr;
    return t3;
}

};

int main()
{
    Time t1, t2 , t3;

```

```

        cout<<"Enter 1st time:";
        t1.get();
        cout<<"Enter 2nd time:";
        t2.get();
        cout<<"The 1st time is";
        t1.disp();
        cout<<"\nThe 2nd time is";
        t2.disp();
        t3 = t3.sum(t1,t2);
        cout<<"\nThe resultant time is";
        t3.disp();

    }

```

Dynamic memory allocation (new and delete): Dynamic memory allocation in C/C++ refers to performing memory allocation manually by programmer. Dynamically allocated memory is allocated on Heap and non-static and local variables get memory allocated on Stack

C++ supports these functions and also has two operators new and delete that perform the task of allocating and freeing the memory in a better and easier way.

new operator

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

Syntax to use new operator: To allocate memory of any data type, the syntax is:

pointer-variable = new data-type;

Here, pointer-variable is the pointer of type data-type. Data-type could be any built-in data type including array or any user defined data types including structure and class.

Example:

```

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

```

OR

```

// Combine declaration of pointer
// and their assignment

```

```

int *p = new int;

```

Initialize memory: We can also initialize the memory using new operator:

pointer-variable = new data-type(value);

Example:

```

int *p = new int(25);
float *q = new float(75.25);

```

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

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

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

Example:

```
int *p = new int[10]
```

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

delete operator:

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

Syntax:

```
// Release memory pointed by pointer-variable
```

```
delete pointer-variable;
```

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

Examples:

```
delete p;
```

```
delete q;
```

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

```
// Release block of memory
```

```
// pointed by pointer-variable
```

```
delete[] pointer-variable;
```

Example:

```
// It will free the entire array
```

```
// pointed by p.
```

```
delete[] p;
```

Q. Write a C++ program to join two strings using dynamic constructor concept.

```
#include<iostream>
#include<string.h>
using namespace std;
class String
{
    char *name;
    int length;
public:
    String()
    {
        length=0;
        name=new char[length+1];    //Dynamic constructor
    }

    String(char *s)
    {
        length=strlen(s);
        name=new char[length+1];
        strcpy(name,s);
    }
}
```

```

void display()
{
    cout<<name;
}

void join(String &a,String &b)
{
    length=a.length+b.length;
    delete name;
    name=new char[length+1];
    strcpy(name,a.name);
    strcat(name,b.name);
}

};

int main()
{
    String name1((char*)"Engineers are");
    String name2((char*)" Creatures of logic");
    String s1;
    s1.join(name1,name2);
    s1.display();
}

```

Important Questions:

Q1. Mention the characteristics of OOP? Write down the features of C++. Explain the difference between OOP and POP. Have you ever realized the importance of C++ over C in your practice? If so how? If not why? Explain your opinion. (features of C++ and characteristics of OOP are different).

Q2. What is Token? Explain default argument and function overloading. Also explain relation between them. Mention importance of inline function. What is reference variable and mention its advantage? Mention the importance of namespace.

Q3. How static member is used? Why constructor is needed and explain types of constructor. What is this pointer? What is dynamic memory allocation explain with an example? Write a C++ program to join two strings using dynamic constructor concept.

Q4. What do you mean by friend function and friend class? Explain with example. Explain the relation between constant member function and constant objects.

Q5. Define operator overloading. Write operator functions as member function of a class to overload arithmetic operator +, logical operator '<=' and stream operator '<<' to operate on the objects of user defined type time (hr, min, sec). Create a class mdistance to store the values in meter and centimeter and class edistance to store values in feet and inches. Perform addition of object of mdistance and object of edistance by using friend function.

Q6. List the non-overloaded operators. Is the overloading of binary operator with member function does require only one argument? Explain. How do you convert user-defined data type to a basic data type, user-defined data type to a user defined data type? Show with example.

Q7. Define access specifiers. Explain function overriding. WAP to show the order of constructor and destructor invocation in single, multiple, multipath inheritances.

Q8. What is diamond problem? How do you resolve it? What is ambiguity problem? How you resolve it.

Q9. Explain in brief about RTTI mechanism (type id, reinterpret and dynamic casting). Explain what is explicit or conversion constructor.

Q10. Explain try, catch and throw. What are the advantages of Exception handling over traditional error handling
WAP of multiple and catch all exception

Q11. Define templates. Mention advantages of function template. Find average of array having five elements using class template concept. Does class template have default argument? Explain

Q12. What is virtual function, pure virtual function and abstract class? Explain with an example. Define stack and queue.

Q13. What is manipulator? What is file stream? Explain different manipulators available in C++.

Important Programs

Q1. WAP to add two complex number or time as object:

By passing object as an argument, by returning object as an argument, by friend function and friend class.

Q2. WAP to overload matrix or "<<" and ">>" operator or [] operator or <, >, !=, == operator.

Q3. Write a C++ program to join two strings using dynamic constructor concept.