

Software

Collection of programs(Collection of instructions)

It's a digitised automated process

s/w

System (os) Heart

Application (application) nose, ears, tongue,

C++ t

OS (dos, unix, linux)

commercial(bank, supermarket etc.,)

DB (oracle, sql)

Translators (interpreter, compiler, assembler)

Device drivers(keyboard, mouse etc.,)

PC & Mobile games (snake, mario, contra)

Protocol (HTTP, FTP, 90%)

C++ is a GP, HL and Compiled PL

Diff between void main | int main

Basic program in CPP

Namespace

Stream

Cin

Cout

For, while in CPP

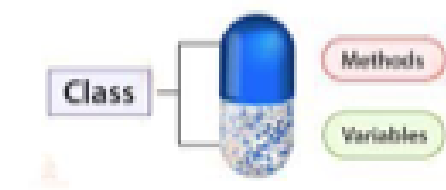
Unit – I

Introduction

C++ began as an expanded version of C. The C++ extensions were first invented by Bjarne Stroustrup in 1979 at Bell Laboratories in Murray Hill, New Jersey. He initially called the new language "C with Classes." However, in 1983 the name was changed to C++. Even though C is an excellent programming language, it has its limits. In C, once a program exceeds from 25,000 to 100,000 lines of code, it becomes so complex that it is difficult to grasp as a totality. The purpose of C++ is to allow this barrier to be broken. The essence of C++ is to allow the programmer to comprehend and manage larger, more complex programs.

Encapsulation

Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse. In an object-oriented language, code and data may be combined in such a way that a self-contained "black box" is created. When code and data are linked together in this fashion, an object is created. In other words, an object is the device that supports encapsulation.



Polymorphism

Object-oriented programming languages support polymorphism, which is characterised by the phrase "one interface, multiple methods." situation. A real-world example of polymorphism is a thermostat. No matter what type of furnace your house has (gas, oil, electric, etc.), the thermostat works the same way. In this case, the thermostat (which is the interface) is the same no matter what type of furnace (method) you have. For example, if you want a 70-degree temperature, you set the thermostat to 70 degrees. It doesn't matter what type of furnace actually provides the heat. Polymorphism helps reduce complexity by allowing the same interface to be used to access a general class of actions. It is the compiler's job to select the specific action (i.e., method) as it applies to each situation. You, the programmer, don't need to do this selection manually. You need only remember and utilise the general interface.

The first object-oriented programming languages were interpreters, so polymorphism was, of course, supported at run time. However, C++ is a compiled language. Therefore, in C++, both run-time and compile-time polymorphism are supported.

Polymorphism example in python (len() method). It can be used for any object in python. The result will be based on the type of object. The same function can be used for the purpose on any object types.

Inheritance

Inheritance is the process by which one object can acquire the properties of another object. This is important because it supports the concept of classification. If you think about it, most knowledge is made manageable by hierarchical classifications. For example, a Red Delicious apple is part of the classification apple, which in turn is part of the fruit class, which is under the larger class food. Without the use of classifications, each object would have to define explicitly all of its characteristics. However, through the use of classifications, an object need only define those qualities that make it unique within its class. It is the inheritance mechanism that makes it possible for one object to be a specific instance of a more general case.

Binding

Consider a base class extended by two derived classes. where both the derived classes has show() function. Which version of draw() does the compiler call? In fact, 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, 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.) Late binding requires some overhead but provides increased power and flexibility.

```
#include <iostream>
using namespace std;
// main() is where program execution begins.
int main()
{
    cout << "Hello World"; // prints Hello World
    return 0;
}
```

Difference between Structure and Class

Features	Structure	Class
Definition	A structure is a grouping of variables of various data types referenced by the same name.	In C++, a class is defined as a collection of related variables and functions contained within a single structure.
Basic	If no access specifier is specified, all members are set to 'public'.	If no access specifier is defined, all members are set to 'private'.

Declaration	<pre>struct structure_name{ type struct_member 1; type struct_member 2; type struct_member 3; . type struct_memberN; };</pre>	<pre>class class_name{ data member; member function; };</pre>
Instance	Structure instance is called the 'structure variable'.	A class instance is called 'object'.
Inheritance	It does not support inheritance.	It supports inheritance.
Memory Allocated	Memory is allocated on the stack.	Memory is allocated on the heap.
Nature	Value Type	Reference Type
Purpose	Grouping of data	Data abstraction and further inheritance.
Usage	It is used for smaller amounts of data.	It is used for a huge amount of data.
Null values	Not possible	It may have null values.
Requires constructor and destructor	It may have only parameterized constructor.	It may have all the types of constructors and destructors.

- By default, all the members of the structure are public. In contrast, all members of the class are private.
- The structure will automatically initialize its members. In contrast, constructors and destructors are used to initialize the class members.
- When a structure is implemented, memory is allocated on a stack. In contrast, memory is allocated on the heap in class.
- Variables in a structure cannot be initialized during the declaration, but they can be done in a class.
- There can be no null values in any structure member. On the other hand, the class variables may have null values.
- A structure is a value type, while a class is a reference type.
- Operators to work on the new data form can be described using a special method.

Program to demonstrate class, object, method definition outside the class using scope resolution.

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

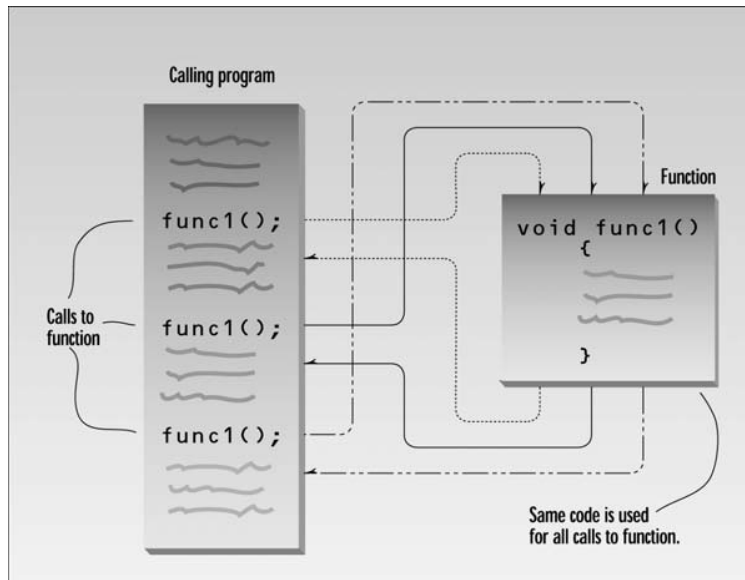
```

{
    int age;
    float sal;
    public:
    void getData();
    void putData();
};
void Person::getData()
{
    cout<<"Enter age";
    cin>>age;
    cout<<"Enter Salary";
    cin>>sal;}
void Person::putData()
{
    cout<<"\nDetails of the candidate are \n";
    cout<<"\nThe age of the person is: "<<this->age;
    cout<<"\nThe Salary of the person is: "<<sal;
}int main()
{
    int num; // Declaring an integer
    cout<<"\nEnter the number of persons";
    cin>>num; // trying to get values from the user.
    Person p[num]; //creating array of objects
    for(int i=0;i<num;i++)
    {
        cout<<"\nEnter the Details of the candidate "<<i+1;
        p[i].getData();
    }
    for(int i=0;i<num;i++)
    {
        cout<<"\nDetails of the candidate "<<i+1<<"are\n";
        p[0].putData();
    }
    return 0;}

```

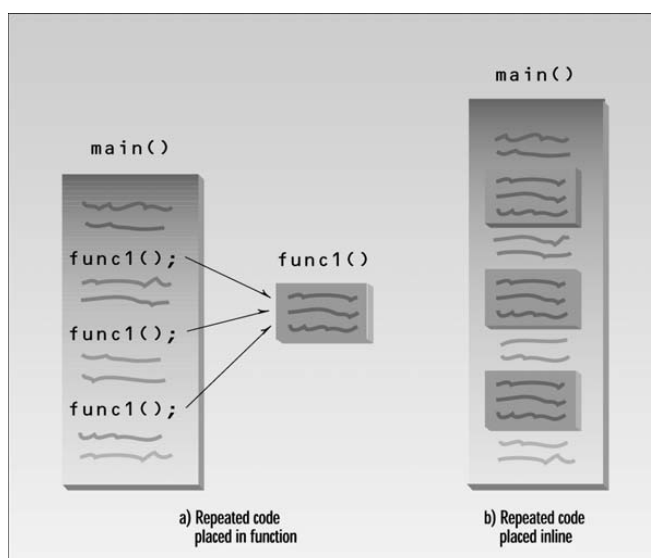
General Behaviour of Function call and need for inline functions

Functions save memory space because all the calls to the function cause the same code to be executed; the function body need not be duplicated in memory. When the compiler sees a function call, it normally generates a jump to the function. At the end of the function it jumps back to the instruction following the call, as shown in Figure below.



Instructions for pushing arguments onto the stack in the calling program and removing them from the stack in the function (if there are arguments), instructions for restoring registers, and an instruction to return to the calling program. The return value (if any) must also be dealt with. All these instructions slow down the program.

To save execution time in short functions, you may elect to put the code in the function body directly **inline** with the code in the calling program. That is, each time there's a function call in the source file, the actual code from the function is inserted, instead of a jump to the function. The difference between a function and inline code is shown in Figure below.



Long sections of repeated code are generally better off as normal functions: The savings in memory space is worth the comparatively small sacrifice in execution speed. But making a short section of code into an ordinary function may result in little savings in memory space, while imposing just as much time penalty as a larger function. In fact, if a function is very short, the instructions necessary to call it may take up as much space as the instructions within the function body, so that there is not only a time penalty but a space penalty as well.

Syntax for inline functions in C++

```
inline return_type function_name(arguments)
{
    //function body
}
```

Example for Inline function

```
include <iostream>
inline int cube(int s)
{
    return s*s*s;
}
int main()
{
    std::cout << "The cube of 3 is: " << cube(3) << "\n";
    return 0;
}
//Output: The cube of 3 is: 27
```

Advantages and Disadvantages of inline functions

Remember, inlining is only a request to the compiler, not a command. Compiler can ignore the request for inlining. Compiler may not perform inlining in such circumstances like:

- 1) If a function contains a loop. (for, while, do-while)
- 2) If a function contains static variables.
- 3) If a function is recursive.
- 4) If a function return type is other than void, and the return statement doesn't exist in the function body.
- 5) If a function contains a switch or goto statement.

Inline functions provide following advantages:

- 1) Function call overhead doesn't occur.
- 2) It also saves the overhead of push/pop variables on the stack when the function is called.
- 3) It also saves overhead of a return call from a function.
- 4) When you inline a function, you may enable the compiler to perform context specific optimization on the body of the function. Such optimizations are not possible for normal

function calls. Other optimizations can be obtained by considering the flows of calling context and the called context.

5) Inline function may be useful (if it is small) for embedded systems because inline can yield less code than the function call preamble and return.

Inline function disadvantages:

- 1) The added variables from the inlined function consume additional registers. After the in-lining function, if the number of variables which are going to use the register increases then they may create overhead on register variable resource utilisation. This means that when the inline function body is substituted at the point of the function call, the total number of variables used by the function also gets inserted. So the number of registers going to be used for the variables will also get increased. So if after function inlining variable numbers increase drastically then it would surely cause an overhead on register utilisation.
- 2) If you use too many inline functions then the size of the binary executable file will be large, because of the duplication of the same code.
- 3) Too much inlining can also reduce your instruction cache hit rate, thus reducing the speed of instruction fetch from that of cache memory to that of primary memory.
- 4) Inline function may increase compile time overhead if someone changes the code inside the inline function then all the calling location has to be recompiled because the compiler would require to replace all the code once again to reflect the changes, otherwise it will continue with old functionality.
- 5) Inline functions may not be useful for many embedded systems. Because in embedded systems code size is more important than speed.
- 6) Inline functions might cause thrashing because inlining might increase the size of the binary executable file. Thrashing in memory causes the performance of computers to degrade.

Constructor:

A constructor is a special type of member function of a class which initializes objects of a class. In C++, Constructor is automatically called when an object(instance of class) is created. It is a special member function of the class because it does not have any return type.

How are constructors different from a normal member function?

- Constructor has same name as the class itself
- Constructors don't have return type
- A constructor is automatically called when an object is created.
- It must be placed in the public section of class.
- If we do not specify a constructor, C++ compiler generates a default constructor for the object (expects no parameters and has an empty body).

Program to illustrate both non-parameterized Constructor and Parameterized Constructor

```
#include<iostream>
using namespace std;
class Person
{
    public:
    int age;
    float salary;
    char* name;
    Person()
    {
        age=29;
        salary=30;
        name="pavan";
    }
    Person(int a,float b,char* c)
    {
        age=a;
        salary=b;
        name=c;
    }
    Person(int a, float b)
    {
        age=a;
        salary=b;
        name="vrsec";
    }
    void getData();
    void putData();
};
void Person::getData()
{
    cout<<"Enter age";
    cin>>age;
    cout<<"Enter Salary";
    cin>>salary;
    cout<<"Enter name of the person";
    cin>>name;
}
void Person::putData()
{
    cout<<"\nDetails of the candidate are \n";
    cout<<"\nName of the person is:"<<name<<"\n";
```

```

    cout<<"\nThe age of the person is: "<<this->age;
    cout<<"\nThe Salary of the person is: "<<salary;
}
int main()
{
    //Person p; // default constructor with no args will be invoked
    //Person p(29,30.3,"pavan"); // constructor with three args will be invoked
    Person p(20,203.3); // default constructor with two args will be invoked
    p.putData();
}

```

Create a class Person with Two member functions getData and putData to take the values from the user and display the entered values to the user respectively. The Person class should have three data members: age, salary and name.

Consider the above scenario and design the solutions

- a. Without Constructors
- b. With default constructors
- c. With Parameterized constructors
- d. With default, parameterized Constructor with overloading

1. When we don't create any constructor by ourselves in the program, the compiler creates a default constructor with no args' and an empty block.

Person()

```

{
}

```

2. When you want to create a default constructor without arguments !

Person() (constructor with no arguments)

```

{
    age=29;
    salary=30;
    name="pavan";
}

```

3. Constructor with arguments

Person(int a,float b,char* c)

```

{
    age=a;
    salary=b;
    name=c;
}

```

4. If we specify both default and parameterized for the same class at once, we are overloading the constructors.

5. Constructor overloading

```
Person(int a,float b)
{
    age=a;
    salary=b;
    name="VRSEC";
}
```

6. We can instantiate the object based on our need !
 - a. Person p; //while creating object def constructor will be invoked.
 - b. Person p(20,30.3,"pavan"); //while creating an object with 3 param constructor will be invoked.
 - c. Person p(20,30.3); //while creating an object with 2 param constructor will be invoked.

Destructor.

Destructor is an instance member function which is invoked automatically whenever an object is going to be destroyed. Meaning, a destructor is the last function that is going to be called before an object is destroyed.

The thing is to be noted here, if the object is created by using new or the constructor uses new to allocate memory which resides in the heap memory or the free store, the destructor should use delete to free the memory.

Syntax:

```
~constructor_name();
```

Properties of destructor:

- Destructor function is automatically invoked when the objects are destroyed.
- It cannot be declared static or const.
- The destructor does not have arguments.
- It has no return type not even void.
- An object of a class with a Destructor cannot become a member of the union.
- A destructor should be declared in the public section of the class.
- The programmer cannot access the address of destructor.

When is destructor called?

A destructor function is called automatically when the object goes out of scope:

- (1) the function ends
- (2) the program ends

(3) a block containing local variables ends

(4) a delete operator is called

How are destructors different from a normal member function?

Destructors have same name as the class preceded by a tilde (~)

Destructors don't take any argument and don't return anything

```
#include<iostream>
using namespace std;
class Demo {
    private:
        int num1, num2;
    public:
        Demo(int n1, int n2) { //creating memory space for the object (instantiation)
            cout<<"Inside Constructor\n";
            num1 = n1;
            num2 = n2;
        }
        void display() {
            cout<<"num1 = "<< num1 <<endl;
            cout<<"num2 = "<< num2 <<endl;
        }
        ~Demo() {
            cout<<"Inside Destructor";
        }
};

int main() {
    Demo obj1(10, 20); //creating & instantiating the object using parameterized constructor
    obj1.display();
    return 0;
}
```

Output:

Inside Constructor

num1 = 10

```
num2 = 20
```

Inside Destructor

1. If any destructor is not specified by the user, a default destructor is invoked by the compiler itself

```
~Person()
```

```
{  
}
```

2. Destructor will not have either arg's or return type
3. If we want to write destructor on our own !

a. **~Person**

```
{  
    cout<<"Inside destructor";  
}
```

4. When destructor will be invoked.
 - a. Function / program terminates
 - b. a block containing local variables ends
 - c. a delete operator is called

The Scope Resolution Operator

As you know, the `::` operator links a class name with a member name in order to tell the compiler what class the member belongs to. However, the scope resolution operator has another related use: it can allow access to a name in an enclosing scope that is "hidden" by a local declaration of the same name. For example, consider this snippet:

```
int i; // global i
void f()
{
    int i; // local i
    i = 10; // uses local i
    .
    .
    .
}
```

As the comment suggests, the assignment `i = 10` refers to the local `i`. But what if function `f()` needs to access the global version of `i`? It may do so by preceding the `i` with the `::` operator, as shown here.

```
int i; // global i
void f()
{
    int i; // local i
    ::i = 10; // now refers to global i
    .
    .
    .
}
```

Friend Functions:

Def: A friend function is a normal function which accesses the private & protected members of the class where it was declared as friend by using the keyword friend.

It is possible to grant a nonmember function access to the private members of a class by using a friend. A friend function has access to all private and protected members of the class for which it is a friend. To declare a friend function, include its prototype within the class, preceding it with the keyword `friend`. Consider the following program

```

#include <iostream>
using namespace std;
class myclass {
int a, b; // private members of this class
public: //method declarations only
//friend is the keyword used to create a friend function. We should use the keyword before
//method signature
friend int sum(myclass x); .
//this sum function can access the private numbers of the class anywhere
void set_ab(int i, int j);
};
void myclass::set_ab(int i, int j)
{
a = i;
b = j;
}
// Note: sum() is not a member function of any class.
int sum(myclass x) //sum is a friend function to myclass
{
/* Because sum() is a friend of myclass, it can
directly access a and b. */
return x.a + x.b;
}
int main()
{
myclass n;
n.set_ab(3, 4);
cout << sum(n); //passing an object to a function/friend functions (passing copy)
return 0;
}

```

In this example, the sum() function is not a member of myclass. However, it still has full access to its private members. Also, notice that sum() is called without the use of the dot operator. Because it is not a member function, it does not need to be (indeed, it may not be) qualified with an object's name.

Things to be noticed with Friend Function:

1. Friend function can be declared anywhere (private, public or protected)
2. Friend function signature should be done within the class with the friend keyword.
3. Method Definition should be there outside the class.

4. As part of C++, private data members cannot be accessed outside the class, but still we are saying that we can access the private members of the class by external functions. This is possible only because we have to **pass the object** to the friend function while calling that friend function .
5. Object means, class copy -> which means the entire information regarding the class will be available with the object, so that we can access the private members of the class.
6. In C++, methods should be called with obj.methodName();
 - a. But friend functions are not methods of any class. They are regular functions, they have to be called like every other function. Ex: functionName();
7. As the friend functions are not regular methods of the class, they cannot access the data members directly. They have to use the object to access the data members of the class with **objName.dataMember** Ex: **Obj.a**
 Inside the friend function we need to use object name along with membership operator (.)
8. The length and size required by the program is reduced a lot because the same function can be used again and again by any class.
9. One regular function can be declared as a friend function to many classes.

Simple program to illustrate friend functions in c++

```
#include<iostream>
using namespace std;
class B; //forward declaration.
class A
{
    int x;
    public:
        void setdata (int i)
        {
            x=i;
        }
    friend void max (A, B); //friend function.
} ;
class B
{
    int y;
    public:
        void setdata (int i)
        {
            y=i;
        }
}
```



```
        friend void max (A, B);
};
void max (A a, B b)
{
    if (a.x >= b.y)
        std::cout<<"The maximum value among both the classes
is:" <<a.x << std::endl;
    else
        std::cout<< b.y << std::endl;
}
int main ()
{
    A a;
    B b;
    a. setdata (10);
    b. setdata (20);
    max (a, b);
    return 0;
}
```

Department of Information Technology:: VRSEC
Object Oriented Programming using C++ Question Bank for Assignment – I

Cognitive Levels(K): K1-Remember; K2-Understand; K3-Apply; K4-Analyze; K5-Evaluate; K6-Create

1	a	Describe the features of Object-Oriented Programming	CO1	K1
	b	Create a C++ program that uses static members to solve movie ticket reservation problem assuming the movie is 'A' rated movie and it shouldn't allow the children below 18 and identify the current status of the seats available and should also display when the house is full.	CO1	K6
2	a	Discuss the role friend functions in C++ with syntax	CO1	K2
	b	Write a C++ program to create a class STUDENT with the following specifications: Data members: name, <u>roll_number</u> Member functions: <u>read()</u> , <u>display()</u> Use the above specifications to read and print the information of 5 students by creating 5 different objects.	CO1	K3
3	a	Explain the structure of C++ program.	CO1	K2
	b	Define Constructor with help of an Example	CO1	K1
4	a	Define Inline function with an example	CO1	K1
	b	Elaborate Static data members and member functions with example	CO1	K2
5	a	Differentiate structure and class	CO1	K1
	b	Create a class book with the fields <u>bookno.</u> , <u>bookname</u> , price, name of the author. Create methods to read the book information, print the details. a. Use the parameterized constructor b. Display method.	CO2	K3
6	a	Write a C++ program to pass object as a parameter to a function	CO1	K3
	b	Write a class called Bank with the data members, <u>acno</u> , <u>actype</u> , <u>name</u> , <u>bal</u> Member functions <u>InsertCustomerDetails()</u> - method to insert the values to the variables, <u>depositAmount()</u> -ask the user to enter the amount to deposit and add the amount to the <u>bal</u> , <u>withdrawAmount()</u> - ask the user to enter the amount to withdraw and update the amount <u>bal</u> finally <u>displayCustomerInformation()</u>	CO2	K3

Static Members in C++

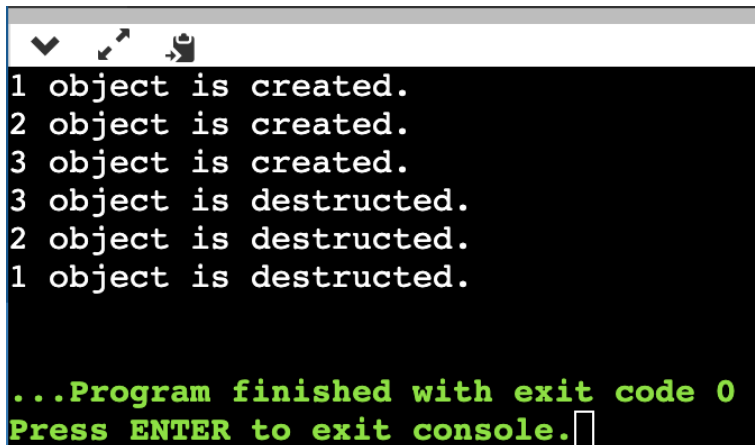
Static data members are class members that are declared using static keywords. A static member has certain special characteristics. These are:

- Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.
- It is initialized to zero when the first object of its class is created. No other initialization is permitted
- It is visible only within the class, but its lifetime is the entire program

Points to be noted

1. Static members have to be declared inside the class
2. Static members has to be defined outside the class
 - a. Syntax: **DataType ClassName::VarName=value**
3. Static member initialization with default of zero
4. Static members are created in data segment ie., public area which means all the data objects can access the same value.
5. **Static members are class members**

```
#include<iostream>
using namespace std;
class Sample
{
    public:
    static int count;
    Sample()
    {
        count++;
        cout<<count<<" object is created."<<"\n";
    }
    ~Sample()
    {
        cout<<count<<" object is destructed."<<"\n";
        count--;
    }
};
int Sample::count;
int main()
{
    Sample s1,s2,s3;
    return 0;
}
```



```

1 object is created.
2 object is created.
3 object is created.
3 object is destructed.
2 object is destructed.
1 object is destructed.

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

```

Code for problem mentioned regarding static members in the question paper **1(b)**

```

#include<iostream>
#include<stdio.h>
using namespace std;
class movie
{
    public:
    int ag;
    static int count,c;
    void age();
};
int movie::count=0; // initially total seats filled is 0
int movie::c=50;    // total 50 seats in theater
void movie::age()
{
    cout<<"total no_of tickets available="<<c<<endl;
    cout<<"enter your age:"<<endl;
    cin>>ag;
    if (ag>18)
    {
        count++;
        cout<<"your ticket is booked"<<endl<<endl;
        c--;
    }
    else if(c==0)
    {
        cout<<"house is full";
    }
    else
    {
        cout<<"your are not allowed to watch the movie"<<endl<<endl;
    }
}

```

```

int main()
{
    int num;
    cout<<"Enter number of tickets to be booked.";
    cin>>num;
    movie m[num];
    for (int i=0;i<num;i++)
    {
        m[i].age();
    }

    cout<<"Age information of all the people who tried to book tickets\n";
    for (int i=0;i<num;i++)
    {

        cout<<"Age of "<<i+1<<"persion is"<<m[i].ag<<"\n";
    }
    return 0;
}

```

Src: pavan (from B section)

Output:

```

Enter number of tickets to be booked.3
total no_of tickets available=50
enter your age:
11
your are not allowed to watch the movie

total no_of tickets available=50
enter your age:
21
your ticket is booked

total no_of tickets available=49
enter your age:
23
your ticket is booked

Age information of all the people who tried to book tickets
Age of 1persion is11
Age of 2persion is21
Age of 3persion is23

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

```

The same problem solved with the help of Constructors:

Src: from soujitha from B section

```
#include <iostream>
using namespace std;
class movie
{
    public:
        int age;
        int count;
        static int no_of_seats;
    movie()
    {
        for(count=0;count<no_of_seats;count++)
        {
            cout << "enter the age:";
            cin >> age;
            if (age<18)
            {
                cout << "you're not allowed!" << endl;
                count = count-1;
            }
            else if (age>18)
            {
                cout << "seat no " << count+1 << " is allotted" << endl;
                cout << "CURRENT STATUS: " << endl;
                cout << "no of seats left:" << no_of_seats-count-1 << endl;
            }
        }
        if (count>=no_of_seats)
        {
            cout << "sorry! house full!!" << endl;
        }
    }
};
int movie::no_of_seats = 5;
int main()
{
    movie m;
    return 0;
}
```

Output:

```
enter the age:19
seat no 1 is allotted
CURRENT STATUS:
no of seats left:4
enter the age:22
seat no 2 is allotted
CURRENT STATUS:
no of seats left:3
enter the age:23
seat no 3 is allotted
CURRENT STATUS:
no of seats left:2
enter the age:43
seat no 4 is allotted
CURRENT STATUS:
no of seats left:1
enter the age:10
you're not allowed!
enter the age:11
you're not allowed!
enter the age:11
you're not allowed!
enter the age:10
you're not allowed!
enter the age:3
you're not allowed!
enter the age:88
seat no 5 is allotted
CURRENT STATUS:
no of seats left:0
sorry! house full!!
```

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

Assignment - I Marks

https://docs.google.com/spreadsheets/d/19FB47c9CGBAjhTjSXz-QSdwyMsRiwLpznu_RdkGqEEI/edit#gid=0

Function overloading:

- Function overloading is a feature of object oriented programming where two or more functions can have the same name but different parameters.
- When a function name is overloaded with different jobs it is called Function Overloading.
- In Function Overloading “Function” names should be the same and the arguments should be different.
- Function overloading can be considered as an example of polymorphism in C++.

Program to demonstrate function overloading.

```
#include <iostream>
using namespace std;
class Demo
{
public:
void displayAvailable(int a)
{
cout<<"\nYou have entered an integer value.. \n which is: "<<a;
}
void displayAvailable(float b)
{
cout<<"\nYou have entered an Floating point value.. \n which is: "<<b;
}
};
int main()
{
int choice,a;
float b;
Demo d;
while(1)
{
cout<<"\n--Menu--\n1. Enter an Integer value \n2.Enter a floating point value\n3. Exit";
cout<<"\nEnter your choice:";
cin>>choice;
switch(choice)
{
case 1:
cout<<"\nEnter an Integer value";
```



```

    cin>>a;
    d.displayAvailable(a);
    break;
    case 2:
    cout<<"\nEnter a Floating point value";
    cin>>b;
    d.displayAvailable(b);
    break;
    case 3:
    cout<<"\nthank you for visiting us ... Please revisit again !";
    exit(0);
}
}
}

```

Output:

```

--Menu--
1. Enter an Integer value
2.Enter a floating point value
3. Exit
Enter your choice:1

Enter an Integer value10

You have entered an integer value..
  which is: 10
--Menu--
1. Enter an Integer value
2.Enter a floating point value
3. Exit
Enter your choice:2

Enter a Floating point value33.3

You have entered an Floating point value..
  which is: 33.3
--Menu--
1. Enter an Integer value
2.Enter a floating point value
3. Exit
Enter your choice:3

thank you for visiting us ... Please revisit again !

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

```

Inheritance

Program:

1. Create a class A with the variables x,y. Create a method to set the data to x and y. Create a subclass B with the variable z. create a method to set the data to z. Write a method to display the information.

```
#include<iostream>
using namespace std;
class ClassA
{
    public:
    int x,y;
    void setData(int a,int b)
    {
        x=a;
        y=b;
    }
};
class ClassB : public ClassA
{
    public:
    int z;
    void assignData(int c) {
        z=c;
    }
    void display()
    {
        cout<<"The values from ClassA are: "<<x<<"and "<<y<<" and from classB is: "<<z;
    }
};

int main()
{
    ClassB objb;
    objb.setData(40,50);
    objb.assignData(90);
    objb.display();
    return 0;
}
```

The values from ClassA are: 40and 50 and from classB is: 90

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

Controlling the parent class with the help of the constructor.

We can call the constructor of the parent class from the base class constructors.

Assigning values for both class and subclass with the help of constructors by creating objects for only one class.

Example program.

```
#include <iostream>
using namespace std;
class Person
{
public:
string fn,ln;
Person()
{
    fn="hello";
    ln="world";
}
};
class employee: public Person
{
public:
int x,z;
string y;
employee(int eno,string edept,int esal)
{
    Person();
    x=eno;
    y=edept;
    z=esal;
}
void display()
{
    cout<<"first name and last name are"<<fn<<","<<ln;
    cout<<"employee no"<<x<<"employee dept"<<y<<"employee
salary"<<z;
}
};
int main()
{
    employee e(12,"It",56789);
    e.display();
    return 0;}
```

Operator overloading:

Operators work on operands

- Unary operators
- Binary operators

For example

Unary Operators (means operator needs only one operand)

+5

-4

Binary operator (means operator needs two operands)

5+4

5-4

Different kinds of data-types

1. Pre-defined data types
2. User defined data types

Usually operators work only on predefined operators

We want to allow these operators to be working on user-defined data types

1. Structures
2. Classes etc.,

In polymorphism we have overloading.

Overloading means same name with different functionalities.

Function overloading

Same function overloading

Different number of arguments

Different data types

Different order also

T1,t2 belong to class Test

T1 obj	T2 obj
a=10	a=20

Program for comparing two objects using functions

```
#include<iostream>
using namespace std;
class Test
{
    public:
        int a; //only one datamember in the class
        void get()
        {
            cin>>a;    }
        void compare(Test t2) //compare implicit obj and explicit obj
        {
            if(a==t2.a) cout<<"both objs are same";
            else cout<<"both objs are not same";
        }
};
int main()
{
    Test t1,t2;
    cout<<"enter the value for t1 obj";
    t1.get();
    cout<<"enter the value for t2 obj";
    t2.get();
    t1.compare(t2);
    return 0;
}
```

Syntax for operator overloading

```
return_type operator symbol(args_list)
{
    //body of function
    [return value]
}
```

few operators **cannot be overloaded**

:: (scope resolution)

. (membership operator)

.*

?: (ternary or conditional operator)

Program for comparing two objects using **Operator Overloading**

```
#include<iostream>
using namespace std;
class Test
{
    public:
        int a; //only one datamember in the class
        void get()
        {
            cin>>a;    }

        //void compare(Test t2) //compare implicit obj and explicit obj
        void operator ==(Test t2)
        {
            if(a==t2.a) cout<<"both objs are same";
            else cout<<"both objs are not same";
        }
};

int main()
{
    Test t1,t2;
    cout<<"enter the value for t1 obj";
    t1.get();
    cout<<"enter the value for t2 obj";
    t2.get();
    //t1.compare(t2);
    t1==t2;
    return 0;
}
```

Assignment for 17-12-2021

Write a MENU driven Program to overload +,-,*,./,% between two objects which have only one data member.

this pointer

how objects look at functions and data members of a class.

- Each object gets its own copy of the data member.
 - All-access the same function definition as present in the code segment.
1. Meaning each object gets its own copy of data members and all objects share a single copy of member functions.
 2. Now the question is that if only one copy of each member function exists and is used by multiple objects, how are the proper data members accessed and updated?
 3. The compiler supplies an implicit pointer along with the names of the functions as **'this'**.
 4. **The 'this' pointer is passed as a hidden argument to all nonstatic member function calls and is available as a local variable within the body of all non-static functions. 'this' pointer is not available in static member functions as static member functions can be called without any object (with class name).**

Operator Overloading (new and delete)

1. If we overload something inside the class -> constructor, method, operator, it is local overloading
2. If we overload outside any class/structure it is global overloading.

The new and delete operators can also be overloaded like other operators in C++. New and Delete operators can be overloaded globally or they can be overloaded for specific classes.

- a) If these operators are overloaded using member functions for a class, it means that these operators are overloaded only for that **specific class**.
- b) If overloading is done outside a class (i.e. it is not a member function of a class), the overloaded 'new' and 'delete' will be called anytime you make use of these operators (within classes or outside classes). **This is global overloading.**

Syntax for New overloading

```
void* operator new(size_t size);
```

Syntax for delete overloading

```
void operator delete(void*);
```

few operators **cannot** be overloaded

:: (scope resolution)

. (membership operator)

.*

?: (ternary or conditional operator)

Overloading new and deleted specific to one particular class (local overloading)

```
#include<iostream>
#include<stdlib.h>

using namespace std;
class student
{
    string name;
    int age;
public:
    student()
    {
        cout<< "Constructor is called\n" ;
    }
    student(string name, int age)
    {
        this->name = name;
        this->age = age;
    }
    void display()
    {
        cout<< "Name:" << name << endl;
        cout<< "Age:" << age << endl;
    }
    void * operator new(size_t size)
    {
        cout<< "Overloading new operator with size: " << size << endl;
        void * p = ::operator new(size);
        //void * p = malloc(size); will also work fine

        return p;
    }

    void operator delete(void * p)
    {
        cout<< "Overloading delete operator " << endl;
        free(p);
    }
};

int main()
{
    student * p = new student("Yash", 24);

    p->display();
    delete p;
}
```

Overloading new and delete **globally**.

```
#include<iostream>
#include<stdlib.h>

using namespace std;
void * operator new(size_t size)
{
    cout << "New operator overloading " << endl;
    void * p = malloc(size);
    return p;
}

void operator delete(void * p)
{
    cout << "Delete operator overloading " << endl;
    free(p);
}

int main()
{
    int n = 5, i;
    int * p = new int[3]; // creating array dynamically by using new operator
which gives more flexibility to the programmer.

    for (i = 0; i<n; i++)
        p[i]= i;

    cout << "Array: ";
    for(i = 0; i<n; i++)
        cout << p[i] << " ";

    cout << endl;

    delete p;
}
```

Exception Handling

Exceptions are run-time anomalies or abnormal conditions that a program encounters during its execution. There are two types of exceptions: a) Synchronous, b) Asynchronous (Ex: which are beyond the program's control, Disc failure etc). C++ provides the following specialised keywords for this purpose.

try: represents a block of code that can throw an exception.

catch: represents a block of code that is executed when a particular exception is thrown.

throw: Used to throw an exception. Also used to list the exceptions that a function throws, but doesn't handle itself.

Need for Exceptional Handling

1. Separation of Error Handling code from Normal Code
2. Functions/Methods can handle any exceptions they choose
3. Grouping of Error Types

Different errors can occur: coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.

When an error occurs, C++ will normally stop and generate an error message. The technical term for this is: C++ will throw an exception (throw an error).

- The **try** statement allows you to define a block of code to be tested for errors while it is being executed.
- The **throw** keyword throws an exception when a problem is detected, which lets us create a custom error.
- The **catch** statement allows you to define a block of code to be executed, if an error occurs in the try block.

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

int main()
{
    int x = -1;

    // Some code
    cout << "Before try \n";
    try {
        cout << "Inside try \n";
        if (x < 0)
        {
            throw x;
            cout << "After throw (Never executed) \n";
        }
    }
```

```

    }
    catch (int x ) {
        cout << "Exception Caught \n";
    }

    cout << "After catch (Will be executed) \n";
    return 0;
}

```

Catchall Exception

```

#include <iostream>
using namespace std;

int main()
{
    try {
        throw 10;
    }
    catch (char *excp) {
        cout << "Caught " << excp;
    }
    catch (...) {
        cout << "Default Exception\n";
    }
    return 0;
}

```

Abnormal Exception because thrown but not handled

```

#include <iostream>
using namespace std;
int main()
{
    try {
        throw 'a';
    }
    catch (int x) {
        cout << "Caught ";
    }
    return 0;
}

```

Multiple Catch Block

```
#include<iostream>

using namespace std;

int main()
{
    int a=10, b=0, c;
    try
    {
        //if a is divided by b(which has a value 0);
        if(b==0)
            throw(c);
        else
            c=a/b;
    }
    catch(char c)    //catch block to handle/catch exception
    {
        cout<<"Caught exception : char type ";
    }
    catch(int i)    //catch block to handle/catch exception
    {
        cout<<"Caught exception : int type ";
    }
    catch(short s)    //catch block to handle/catch exception
    {
        cout<<"Caught exception : short type ";
    }
    cout<<"\n Hello";
}
```

Catching derived class Exceptions

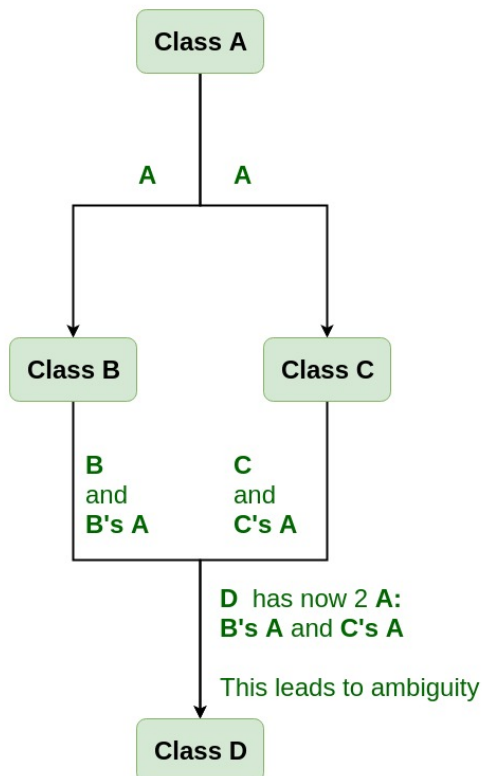
```
#include<iostream>
using namespace std;
class Base {};
class Derived: public Base {};
int main()
{
```

```
Derived d;  
// some other stuff  
try {  
    // Some monitored code  
    throw d;  
}  
catch(Base b) {  
    cout<<"Caught Base Exception";  
}  
catch(Derived d) { //This catch block is NEVER executed  
    cout<<"Caught Derived Exception";  
}  
getchar();  
return 0;  
}
```

Sessional Marks Link

https://docs.google.com/spreadsheets/d/19FB47c9CGBAjhTjSXz-QSdwyMsRiwLpznu_RdkGqEEI/edit#gid=0

Virtual Class



Need	Solution
<pre> #include <iostream> using namespace std; class A { public: void show() { cout << "Hello form A \n"; } }; class B : public A { }; class C : public A { }; class D : public B, public C { </pre>	<pre> #include <iostream> using namespace std; class A { public: int a; A() // constructor { a = 10; } }; class B : public virtual A { }; class C : public virtual A { }; </pre>

<pre>}; int main() { D object; object.show(); }</pre> <p>Output:</p> <p>prog.cpp:29:9: error: request for member 'show' is ambiguous ^ prog.cpp:8:8: note: candidates are: void A::show() ^ prog.cpp:8:8: note: void A::show()</p>	<pre>class D : public B, public C { }; int main() { D object; // object creation of class d cout << "a = " << object.a << endl; return 0; }</pre> <p>Output:</p> <p>a = 10</p>
--	---

1. Whenever we use virtual keywords while inheriting, the properties of base class will be bypassed and a single instance will be invoked.
2. Now only one copy of data/function members will be copied to class C and class B and class A becomes the virtual base class.
3. Virtual base classes offer a way to save space and avoid ambiguities in class hierarchies that use multiple inheritances.
4. When a base class is specified as a virtual base, it can act as an indirect base more than once without duplication of its data members. A single copy of its data members is shared by all the base classes that use virtual base.

Virtual Function

Virtual function is a member function which is declared within a base class and is re-defined (overridden) by a derived class.

1. Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
2. They are mainly used to achieve Runtime polymorphism
3. Functions are declared with a virtual keyword in base class.

4. The resolving of function call is done at runtime.

Rules for Virtual Functions

1. Virtual functions cannot be static.
2. A virtual function can be a friend function of another class.
3. Virtual functions should be accessed using pointer or reference of base class type to achieve runtime polymorphism.
4. The prototype of virtual functions should be the same in the base as well as derived class.
5. They are always defined in the base class and overridden in a derived class. It is not mandatory for the derived class to override (or re-define the virtual function), in that case, the base class version of the function is used.

Abstract Base Class - A class which contains pure virtual functions

- Never allows objects to this class
- Designed only for inheritance concept only
- They are used to create base class pointer at get dynamic/runtime polymorphism

Pure Virtual functions - do nothing functions

- For example, let Shape be a base class. We cannot provide implementation of function draw() in Shape, but we know every derived class must have implementation of draw().
- Animal class doesn't have an implementation of move() (assuming that all animals move), but all animals must know how to move. We cannot create objects of abstract classes.
- But we must override that function in the derived class, otherwise the derived class will also become abstract class

A pure virtual function is declared by assigning 0 in declaration

```
class Test
{
    // Data members of class
public:
    // Pure Virtual Function
    virtual void show() = 0;

    /* Other members */
};
```

Sample program to illustrate pure virtual functions

```
#include<iostream>
using namespace std;
class Shape
{
public:
    float d1,d2;
    void getDim()
    {
        cin>>d1>>d2;
    }
    virtual float area()=0; // pure virtual function, hence it becomes abs class
    // we cannot create obj also for abs class
};
class Triangle:public Shape
{
public:
    virtual float area()
    {
        return 0.5*d1*d2;
    }
};
class Rectangle:public Shape
{
public:
```

```

    virtual float area()
    {
        return d1*d2;
    }
};
int main()
{
    Triangle t;
    cout<<"Enter the dimensions of triangle:";
    t.getDim();
    cout<<"\nArea of the triangle for the given dimensions:"<<t.area()<<endl;
    Rectangle r;
    cout<<"\nEnter the dimensions of Rectangle:";
    r.getDim();
    cout<<"\nArea of the triangle for the given dimensions:"<<r.area();
    return 0;
}

```

```

Enter the dimensions of traigle:10 5

Area of the traingle for the given dimensions:
values are10 , 5
25

Enter the dimensions of Rectangle:10 5

Area of the traingle for the given dimensions:50

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

```

Function overloading

Same function overloading

Different number of arguments

Different data types

Different order also

Different return type also

Ex:

```
#include<iostream>
using namespace std;
int sum(int a,int b)
{
return a+b;
}
float sum(float a,float b)
{
return a+b;
}
float sum(int a,float b)
{
return a+b;
}
int main()
{
clrscr();
cout<<sum(5,6)<<endl;
cout<<sum(10.5,11.5)<<endl;
cout<<sum(5,6.5)<<endl;
return 0;
}
```

Draw backs:

Lines of code are not getting reduced.

Only common thing is the function name. (name stack advantage)

We cannot pass data type as a parameter so that we don't need to write the same code for different data types.

Need for template

For example, a software company may need sort() for different data types. Rather than writing and maintaining the multiple codes, we can write one sort() and pass data type as a parameter.

1. Call by value (this will pass values)
2. Call by reference (this will pass address as argument)

We want to pass the datatype as an argument (hidden)

We will receive both data type as well as data at the function definition

Syntax to create a template

By using keywords: template & **typename/class**

What is difference between typename and class in template in c++

Ans: both words are interchangeable.

```
template <typename T> // template <class T>
T sum(T a,T b)
{
return a+b;
}
```

Templates are expanded at compile time. This is like macros. The difference is, the compiler does type checking before template expansion. The idea is simple: source code contains only function/class, but compiled code may contain multiple copies of the same function/class.

A simple template function

```
//Program to illustrate function template / template function
#include<iostream>
using namespace std;
template <typename T> // template <class T>
T sum(T a,T b)
{
```

```

return a+b;
}
int main()
{
cout<<sum(5,6)<<endl; // sum is a generic function here
cout<<sum(10.5,11.5)<<endl;
return 0;
}

```

Overloading - > same name different functionalities

Function, operator or even a template functions as well.

Now we will see how to overload a template function:

```

// C++ program to illustrate overloading
// of template function using an
// explicit function
#include <iostream>
using namespace std;
// Template declaration
template <typename T>
// Template overloading of function
void display(T t1)
{
    cout << "Displaying Template: "<< t1 << "\n";
}
// Template overloading of function
void display(int t1) // exclusive for integers
{
    cout << "Explicitly display: "<< t1 << "\n";
}
int main()
{
    display(200);
    display(12.40);
    display('G');
    return 0;
}

```

Till now we have discussed function template which can hold one datatype
 Template <typename T>

Now I am supposed to define a function which accepts two two different data types !

```
#include <iostream>
template <typename T, typename U> // We're using two template type
parameters named T and U
U max(T x, U y) // x can resolve to type T, and y can resolve to type U
{
    return (x > y) ? x : y; // uh oh, we have a narrowing conversion problem
    here
}
//driver code -> where the execution starts
int main()
{
    std::cout <<"The maximum of Two number is: " <<max(2, 3.5) << '\n';
    std::cout <<"The maximum of Two number is: " <<max(3.5,2) << '\n';
    std::cout<<"The maximum of Two number is: " <<max(3.5,12.5) << '\n';
    return 0;
}
// why the answer is only 3 not 3.5?
```

Now we will see how to sort values of different data types by using a function template

```
// using template function
#include <iostream>
using namespace std; //swap function
// A template function to implement bubble sort.
// We can use this for any data type that supports
// comparison operator < and swap works for it.
template <typename T>
void bubbleSort(T a[], int n) {
    for (int i = 0; i < n - 1; i++)
        for (int j = n - 1; i < j; j--)
            if (a[j] < a[j - 1])
                swap(a[j], a[j - 1]);
}
```

```

}
int main() {
    int a[5] = {10, 50, 30, 40, 20};
    int n = sizeof(a) / sizeof(a[0]);

    // calls template function
    bubbleSort(a, n);

    cout << " Sorted array : ";
    for (int i = 0; i < n; i++)
        cout << a[i] << " ";
    cout << endl;

    // creating a float array with four values
    float b[4] = {4.5, 6.5, 1, 0};
    int n1 = sizeof(b) / sizeof(b[0]); //to find number of values in the array
    // so that sorting needs it
    // calls template function
    bubbleSort(b, n1);
    cout << " Sorted array : ";
    for (int i = 0; i < n1; i++)
        cout << b[i] << " ";
    cout << endl;
return 0;
}

```

<https://docs.google.com/document/d/1QVgYQqT7604vrL1HJBphBuFmxVn63gPgXJXxy3KAA-o/edit>

1. Create a template function which accepts either integer or float or double values and return the sum value
2. Write a c++ program using function template for swapping of two numbers. (int, float)
3. Write a c++ program using function template for bubble sort. (int, float)
4. Write a c++ program using function template for finding the maximum of two element (int, float)
5. Write a c++ program for overloading a function template.

- a. For addition
- b. For multiplication

operations.

1. Templates are used to create a family of functions effectively
2. Function template / generic function

1. Generic function which can take single datatype
2. With two datatypes
3. Template overloading
4. Template class

We hv created a class with integer data members !

How to use the same class to handle any kind of data members

1. Only one solutions generic class or class template

1. Program to illustrate Function overloading (need for template)
2. Program for designing a template function
3. Program for template overloading
4. Bubble sort using generic function
5. Generic function which uses two different data types
6. Class template / Generic class with Single parameters
7. Class template / Generic class with multiple parameter

1. Function template / generic function
2. Class template / generic class

<pre>class Test { public: int a,b; void display()</pre>	<p>Here we have created a class which as two data members of integer type and has a method display()</p> <p>-> while defining the class we are</p>
---	---

<pre>{ } };</pre>	<p>creating two members and we are specifying data types as well ! which means my class will work only on integers but not floats!</p>
-------------------	--

Need for a generic class

```
#include<iostream>
using namespace std;
class Shape
{
    public:
    int a,b;
    Shape(int x,int y)
    {
        cout<<"in param constructor"<<x<<y;
        a=x;b=y;
    }
};
int main()
{
    Shape s(10,13.5); // .5 in 13.5 is deprecated bcaz in the constructor
    // it is expecting an integer.. this is why need a generic class
}
```

```
in param constructor1013
```

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

Class Templates

A simple class template to get two values and display them

```
// CPP program to illustrate
// Class template with Single parameters

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

```

// Class template with one parameters
template<class T1>
class Test
{
    T1 a;
    T1 b;
public:
    Test(T1 x, T1 y)// param constructor
    {
        a = x;
        b = y;
    }
    void show()
    {
        cout << a << " and " << b << endl;
    }
};
// Main Function
int main()
{
    // instantiation with int and int type
    Test <int>t(5, 6);
    t.show();

    return 0;
}

```

Generic class with Two different data types

Int, float

```

// CPP program to illustrate
// Class template with multiple parameters
#include<iostream>
using namespace std;
// Class template with two parameters
template<class T1, class T2>
class Test

```

```

{
    T1 a;
    T2 b;
public:
    Test(T1 x, T2 y)
    {
        a = x;
        b = y;
    }
    void show()
    {
        cout << a << " and " << b << endl;
    }
};

// Main Function
int main()
{
    // instantiation with float and int type
    Test <float, int> test1 (1.23, 123);

    // instantiation with float and char type
    Test <int, char> test2 (100, 'W');

    test1.show();
    test2.show();

    return 0;
}

```

Generic Templates class Templates

- 1 Write a c++ program using class template for linear search
- 2 Write a c++ program using class template for Binary search
- 3 Write a c++ program using class template for performing calculator operations.

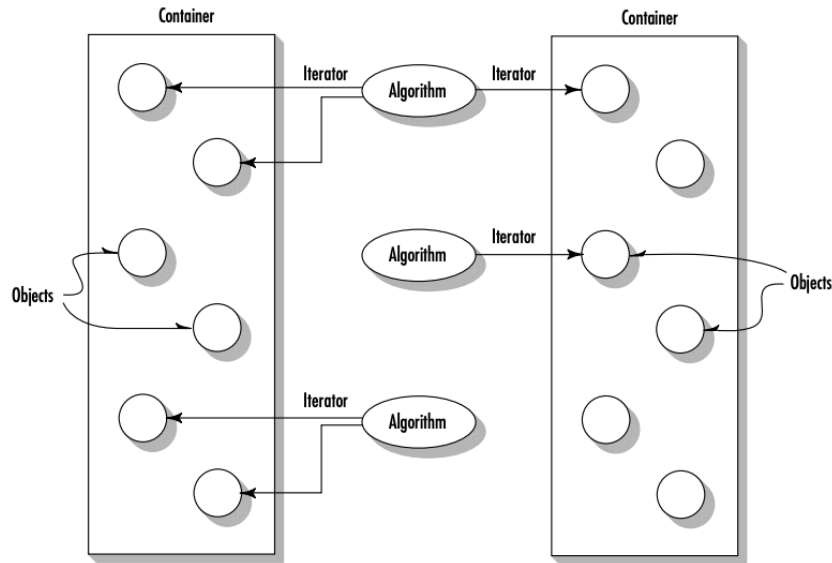
Standard Template Libraries

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 generalised library and so, its components are parameterized.

The STL contains several kinds of entities. **The three most important are containers, algorithms, and iterators.**

A **container** is a way that stored data is organised in memory. However, there are many other kinds of containers, and the STL includes the most useful. The STL containers are implemented by template classes, so they can be easily customised to hold different kinds of data.

Algorithms in the STL are procedures that are applied to containers to process their data in various ways. For example, there are algorithms to **sort, copy, search, swap, and merge data.** Algorithms are represented by template functions. These functions are not member functions of the container classes. Rather, they are standalone functions. Indeed, one of the striking characteristics of the STL is that its algorithms are so general. You can use them not only on STL containers, but on ordinary C++ arrays and on containers you create yourself.



Algorithms use iterators to act on objects in containers

Iterators are a generalisation of the concept of pointers: they point to elements in a container. You can increment an iterator, as you can a pointer, so it points in turn to each element in a container. Iterators are a key part of the STL because they connect algorithms with containers. Think of them as a software version of cables (like the cables that connect stereo components together or a computer to its peripherals).

Containers in the STL fall into two main categories: **sequence and associative**. The sequence containers are vector, list, and deque. The associative containers are set, multiset, map, and multimap. In addition, several specialised containers are derived from the sequence containers. These are stack, queue, and priority queue.

Normally we are supposed to write an algorithm to find the element in the array.

We will create an array to store the some values

We will program/algorithm/procedure which uses loop control variable 'i' to point each position in the array.

We will create a container where the algorithms will be using iterator to point the different values/positions in the container !

In python

```
a =10
```

```
print(type(a))
```

Output:

```
<class 'int'>
```

Container will be like a template class

To assign use the values of a class we have to create some objects.

There were many containers present in the STL

We need to create objects for these containers so that we can apply generic functions over these objects to perform our desired calculations

- Sequence Containers: implement data structures which can be accessed in a sequential manner.
 - [vector](#)
 - [list](#)
 - [deque](#)
 - [arrays](#)
 - [forward_list](#) (Introduced in C++11)
- Container Adaptors : provide a different interface for sequential containers.
 - [queue](#)
 - [priority_queue](#)
 - [stack](#)
- Associative Containers : implement sorted data structures that can be quickly searched ($O(\log n)$ complexity).
 - [set](#)
 - [multiset](#)
 - [map](#)
 - [multimap](#)
- Unordered Associative Containers : implement unordered data structures that can be quickly searched
 - [unordered_set](#) (Introduced in C++11)
 - [unordered_multiset](#) (Introduced in C++11)
 - [unordered_map](#) (Introduced in C++11)
 - [unordered_multimap](#) (Introduced in C++11)

Array : collection of homogeneous ele which are stored together in contiguous memory locations which can be referred to by a common name.

```
int a[10] // static way
```

```
a = (int*)malloc(n * sizeof(int)); // we need to specify 'n'
(dynamic way of allocating memory to the array)
```

But the array is not dynamic, memory allocation is dynamic
The user said he serves 50 customers a day !

Iterators in vectors

begin() – Returns an iterator pointing to the first element in the vector

end() – Returns an iterator pointing to the theoretical element that follows the last element in the vector

rbegin() – Returns a reverse iterator pointing to the last element in the vector (reverse beginning). It moves from last to first element

rend() – Returns a reverse iterator pointing to the theoretical element preceding the first element in the vector (considered as reverse end)

cbegin() – Returns a constant iterator pointing to the first element in the vector.

cend() – Returns a constant iterator pointing to the theoretical element that follows the last element in the vector.

crbegin() – Returns a constant reverse iterator pointing to the last element in the vector (reverse beginning). It moves from last to first element

crend() – Returns a constant reverse iterator pointing to the theoretical element preceding the first element in the vector (considered as reverse end)

```
//using sort algorithm
#include <algorithm>
#include <iostream>
using namespace std;
void show(int a[], int array_size)
{
    for (int i = 0; i < array_size; ++i)
        cout << a[i] << " ";
}
// Driver code
int main()
{
    int a[] = { 1, 5, 8, 9, 6, 7, 3, 4, 2, 0 }; //statically

    // size of the array
    int asize = sizeof(a) / sizeof(a[0]);
    cout << "The array before sorting is : \n";

    // print the array
    show(a, asize);
```

```

    // sort the array
    sort(a, a + asize);

    cout << "\n\nThe array after sorting is :\n";

    // print the array after sorting
    show(a, asize);

    return 0;
}

```

```

//strings

#include <string>
#include<iostream>
using namespace std;
int main()
{
    string str = "something";
    string str1("C S Pavan kumar");
    //we are creating an object for string class in the STL and assigning value
    "something"
    cout << "The size of "<< str<< " is " << str.size()<< " characters."<< endl;
    // we are using str.size | str is obj of string class ans size is method in class
    cout << "The size of "<< str1<< " is " << str1.size()<< " characters."<< endl;

}

```

```

// C++ program to illustrate the
// iterators in vector
#include <iostream>
#include <vector>
using namespace std;
int main()
{

```

```

    vector<int> g1;
//    for (int i = 1; i <= 5; i++)
        g1.push_back(10); // [1,2,3,4,5]
g1.push_back(20);
g1.push_back(30);
g1.push_back(40);
g1.push_back(50);

    cout << "Output of begin and end: ";
    for (auto i = g1.begin(); i != g1.end(); ++i)
        cout << *i << " ";

    cout << "\nOutput of cbegin and cend: ";
    for (auto i = g1.cbegin(); i != g1.cend(); ++i)
        cout << *i << " ";

    cout << "\nOutput of rbegin and rend: ";
    for (auto ir = g1.rbegin(); ir != g1.rend(); ++ir)
        cout << *ir << " ";

    cout << "\nOutput of crbegin and crend : ";
    for (auto ir = g1.crbegin(); ir != g1.crend(); ++ir)
        cout << *ir << " ";

    return 0;
}

```

Capacities of vector

- ☐ size() – Returns the number of elements in the vector.
- ☐ max_size() – Returns the maximum number of elements that the vector can hold.
- ☐ capacity() – Returns the size of the storage space currently allocated to the vector expressed as number of elements.
- ☐ resize(n) – Resizes the container so that it contains ‘n’ elements.
- ☐ empty() – Returns whether the container is empty.
- ☐ shrink_to_fit() – Reduces the capacity of the container to fit its size and destroys all elements beyond the capacity.

- `reserve()` – Requests that the vector capacity be at least enough to contain `n` elements.

in python - `len` fun which accepts any `object_type` (sequence objects) and returns the length or size of the object given

Vector - dynamic array

No need to define the size of the array while creating

Like `int a[10]` (no need)

Here we are specifying the number of elements to be created
=> but, we are creating memory space for 10 ele

Malloc & calloc => dynamic memory alloc !

Vector ?? need

Vector - dynamic array

5 elements at the creation

Lets us say we want to insert the 6th element ?

Then vector allocate double memory for the next insertion !

It shrinks automatically if we are not using the allocated memory

Lab task Feb 12 (1)

DATE AND TIME EXTRACT_REGEX

Meeting Schedule

Ana is the personal security of Ram, So Ram gave all the responsibility of scheduling meetings, appointments from externals and handling the company files. So now Ana, every day has to take the list of meetings, that Ram has to attend and schedule according to that.



So now she has the list of appointments details, She planned to simplify the details so that she can easily understand. Now she wants to fetch the date and time also the name of the meeting from the given detail.

So let us try to write a program to Extract date and time from the given string. Using metacharacter. If AM or PM is not mentioned, consider it as railway timing or else convert it.

For Ex :

Input: Content Meeting on 12th Jan 2016, at 11:02 PM

Output: Content Meeting - 12th Jan 2016 23:02

Procedure:

- 1) Extract date from the string.
- 2) Extract time from the string. Convert it into railway timing, if it's not in railway format.
- 3) Remove the date and time from the string, using the replace function.
- 4) Then, remove 'On', 'on' and 'at' from the string.
- 5) Strip the string, using strip function, to get the context of the meeting.

Problem Constraints:

Use regex methods for validating.

Use re and datetime module.

Input and Output Format:

Input is a String that indicates meeting details.

The output contains a string contains meeting name, date and time. (Refer to sample output format).

Note: All text in bold corresponds to the input and the rest corresponds to output.

Sample Input and Output 1:

Content Meeting on 12th Jan 2016, at 11:02 PM

Content Meeting - 12th Jan 2016 23:02

Sample Input and Output 2:

11th May 2019 at 05:30 PM Meeting with CEO

Meeting with CEO - 11th May 2019 17:30

Additional Sample TestCases

Sample Input and Output 1 :

Content Meeting on 12th Jan 2016, at 11:02

Content Meeting - 12th Jan 2016 11:02

Sample Input and Output 2 :

11th May 2019 at 05:30 PM Meeting with CEO

Meeting with CEO - 11th May 2019 17:30

Lab task Feb 12 (2)

It is IPL Season and the first league match of Dhilip's favorite team, "Chennai Super Kings". The CSK team is playing at the IPL after 2 years and like all Dhoni lovers, Dhilip is also eagerly awaiting to see Dhoni back in action.

After waiting in long queues, Dhilip succeeded in getting the tickets for the big match. On the ticket, there is a letter-code that can be represented as a string of Latin letters.

Dhilip believes that the CSK Team will win the match in case exactly two different letters in the code alternate. Otherwise, he believes that the team might lose. Please see note section for formal definition of alternating code.

You are given a ticket code. Please determine, whether CSK Team will win the match or not based on Dhilip's conviction. Print **"Yes"** or **"No"** corresponding to the situation.

Note:

Two letters x, y where $x \neq y$ are said to be alternating in a code, if code is of form **"xyxyxy..."**(case insensitive).

Input Format:

First and only line of the input contains a string S denoting the letter code on the ticket.

Output Format:

Output a single line containing **"Yes"** (without quotes) based on the conditions given and **"No"** otherwise.

Refer sample input and output for formatting specifications.

Sample Input1:

ABABAB

Sample Output1:

Yes

Sample Input2:

ABC

Sample Output2:

No

Sample Input3:

XYXYX

Sample Output3:

Yes

Assignment Questions for Assignment – 2

1.
 - a. Describe how templates overcome the disadvantage of function overloading and design a solution for adding two integers or floats with help of templates.
 - b. Create an Abstract class called shaped use this class to store two double data type values that could be used to compute the area of figures. Drive two specific class called triangle and rectangle from the base class shape.
 - a. Add a member function get data to the base class to initialize base class data members and another function displaying area.
 - b. Make display area as an abstract function and redefine this function in the derived class.
2.
 - a. Explain the process of Handling Derived Class Exceptions
 - b. Design a solution for handling divide by zero error
3.
 - a. Use Abstract class to perform payroll calculation based on the type of the employee. We can use base class as employee. The derived class for the employee is boss who gets fixed salary per month, weekly based workers, piece workers who get paid by the number of pieces produced and hourly based workers.
 - b. Differentiate Early vs Late binding
4.
 - a. Describe the process of exception handling in C++
 - b. Design a catch block to handle multiple exceptions (divide by zero, array out of bounds exception)
5.
 - a. Explain the need of overloading generic functions with an example.
 - b. Use abstract function for conversion of Fahrenheit to Celsius, meters to kilometers, hours to seconds.
6.
 - a. Design a class template for Linear search application
 - b. Design a solution to check if the given phone number is valid or not, if the number has exactly 5 digits exactly display as valid using exceptional handling?

2a, 2b, 4b, 6b (questions also in week 12)

https://docs.google.com/spreadsheets/d/1xe2leo0YfPoj9NyRCvrDf_VjMgmCBTmipr1SLEFUZvg/edit?usp=sharing&urp=gmail_link

Everyone fill this form now !(certifications only)