C++ Notes Day-5 Date: 09 June 2025

Lets revise

- Structure in C++
- Class and Object
- static and extern storage class
- Scope and Lifetime of the variables
- Namespace

Stream concept

- Variable is a container which is used to store data in RAM.
- File is a container which is used to store data in HDD.
- Stream is an abstraction(object), which either produce(write) or consume(read) inform from source to destination.
- Console is also called as terminal = Keyboard + Monitor / Printer.
- Standered stream objects in C
    - stdin
        - standerd input stream associated with keyboard to read the data from keyboard.

```
scanf("%d",&Num1);
//fscanf(stdin,"%d",&Num1);
```

    - stdout
        - standerd output stream associated with monitor to write the data.

```
printf("%d",Num1);
//fprintf(stdout,"%d",Num1);
```

    - stderr
        - standerd output stream associated with monitor to write the error.
- Standard stream objects in C++ associated with console (Keyboard + Monitor).
    - cin,
    - cout,
    - cerr
    - clog objects
    - Above listed stream objects are decalred in std namesapces in iostream header file.
        - cin (Character Input): Keyboard
        - cout (Character Output): Monitor
        - cerr + clog (Character Error + Chracter Log): Monitor
    - cin, cout, cerr and clog are external objects declared in std namespace. Hence to use it we should use std::cin, std::cout, std::cerr, std::clog.

Character Output(cout)

```
    typedef basic_ostream<char> ostream;
```

- As shown above, ostream is alias / another name given to the basic_ostream class.
- cout is object of ostream class. It is external object declared in std namespace.
- It represents monitor which is used to write data on monitor.
- Example 1:

```
#include<cstdio>
#include<iostream>
int main( void ){
 printf("Hello World\n");
 std::cout << "Hello World\n";
 return 0;
}
```

- "<<" operator is called as insertion operator.
- In C language, escape sequence is a character which is used to format the output.
    - Example: '\n', '\t', '\r' etc.
- In C++ language, manipulator is a function which is used to format the output.
    - Example: endl, setw, fixed,scientific, dec, oct, hex etc.
- Example 2:

```
#include<iostream>
int main( void ){
 std::cout << "Hello World" << std::endl;
 //or
 using namespace std;
 cout << "Hello World" <<endl;
 return 0;
}
```

- Example 3:

```
#include<iostream>
int main( void ){
int num1 = 10;
int num2 = 20;
using namespace std;
cout << num1 << num2 << endl;
return 0;
}
```

- Example 4:

```
#include<iostream>
int main( void ){
 int num1 = 10;
 int num2 = 20;
 using namespace std;
 cout << num1 << endl;
 cout << num2 << endl;
 return 0;
}
```

- Example 5:

```
#include<iostream>
int main( void ){
 int num1 = 10;
 int num2 = 20;
 using namespace std;
 cout << "Num1 : " << num1 << endl;
 cout << "Num2 : " << num2 << endl;
 return 0;
}
```

Character Input(cin)

```
    typedef basic_istream<char> istream;
```

- As shown above, istream is another name given to the basic_istream class.
- cin is object of istream class. It is external object declared in std namespace.
- It represents keyboard which is used to read data from keyboard.
- Example 1:

```
#include<cstdio>
#include<iostream>
int main( void ){
 int num1;
 //In C programming language
 printf("Num1 : ");
 scanf("%d", &num1 );
 //In C++ programming language
 std::cout << "Num1 : ";
 std::cin >> num1;
 return 0;
}
```

- ">>" operator is called as extraction operator.
- Example 2:

```
#include<iostream>
int main( void ){
 int num1;
 std::cout << "Num1 : ";
 std::cin >> num1;
 //or
 using namespace std;
 cout << "Num1 : ";
 cin >> num1;
 return 0;
}
```

- Example 3:

```
#include<iostream>
int main( void ){
 int num1, num2;
 using namespace std;
 cin >> num1 >> num2;
 cout << num1 << num2 << endl;
 return 0;
}
```

- Example 4:

```
#include<iostream>
int main( void ){
 using namespace std;
 int num1;
 cout << "Num1 : ";
 cin >> num1;
 int num2;
 cout << "Num2 : ";
 cin >> num2;
 cout << "Num1 : " << num1 << endl;
 cout << "Num2 : " << num2 << endl;
 return 0;
}
```

Character Error(cerr) and Character Log(clog)

- Consider below program:

```
#include<iostream>
#include<iomanip>
int main( void ){
using namespace std;
int num1;
cout << "Num1 : ";
cin >> num1;
clog << "Numerator is accepted" <<endl;
int num2;
cout << "Num1 : ";
cin >> num2;
clog << "Denominator is accepted" <<endl;
if( num2 == 0 ){
cerr << "Value of denominator is 0" <<endl;
clog << "Can not calculate Result because value of denominator is
0." <<endl;
}else{
int result = num1 / num2;
clog << "Result is calculated" <<endl;
cout<< "Result : "<< result << endl;
clog << "Result is printed" <<endl;
}
return 0;
}
```

Polymorphism in C++

Function Overloading

- In C programming language, we can not give same name to the multiple functions in same project.
- In C++, we can give same name to the multiple functions.
- If implementation of functions are logically same / equivalent then we should give same name to the function.
- If we want to give same name to the function then we must follow some rules:
- Rule 1:
    - If we want to give same name to the function and if type of all the parameters are same then number of parameters passed to the function must be different.

```
void sum( int num1, int num2 ){
int result = num1 + num2;
cout<<"Result : "<<result<<endl;
}
void sum( int num1, int num2, int num3 ){
int result = num1 + num2 + num3;
cout<<"Result : "<<result<<endl;
}
int main( void ){
sum( 10, 20 );
sum( 10, 20, 30 );
```

```
return 0;
}
```

- Rule 2:
  - If we want to give same name to the function and if number of parameters are same then type of at least one parameter must be different.

```
void sum( int num1, int num2 ){
int result = num1 + num2;
cout<<"Result : "<<result<<endl;
}
void sum( int num1, double num2 ){
double result = num1 + num2 ;
cout<<"Result : "<<result<<endl;
}
int main( void ){
sum( 10, 20 );
sum( 10, 20.5 );
return 0;
}
```

- Rule 3:
  - If we want to give same name to the function and if number of parameters are same then order of type of parameters must be different.

```
void sum( int num1, float num2 ){
float result = num1 + num2;
cout<<"Result : "<<result<<endl;
}
void sum( float num1, int num2 ){
float result = num1 + num2 ;
cout<<"Result : "<<result<<endl;
}
int main( void ){
sum( 10, 20.2f );
sum( 10.1f, 20 );
return 0;
}
```

- Rule 4
  - Only on the basis of different return type, we can not give same name to the function.

```
int sum( int num1, int num2 ){
int result = num1 + num2;
return result;
}
```

```
void sum( int num1, int num2 ){ //Error: Function definition is
not allowed
int result = num1 + num2;
}
int main( void ){
return 0;
}
```

- Definition of Function Overloading
  - When we define multiple functions with the help of above 4 rules then process is called as function overloading.
  - Process of defining functions with same name and different signature is called as function overloading.
  - Functions which take part into overloading are called as overloaded functions.
  - If implementation of functions are logically same / equivalent then we should overload function.
  - In C++ we can overload:
    - global function
    - member function
    - constructor
    - static member function
    - constant member function
    - virtual member function
  - In C++ we can not overload:
    - main function
    - destructor
  - Per project, we can define only one main function. Hence we can not overload main function in C++.
  - Since destructor do not take any parameter, we can not overload destructor.

Why retrun type is not considered in function overloading?

- Since catching value from function is optional, return type is not considered in function overloading.

Name mangling and Mangled name

- nm is a tool which is used to print symbol table. We can used it to see mangled name.
- if we define function in C++, then compiler generate unique name for each function by looking toward name of the function and type of parameter passed to the function. Such name is called as mangled name.

```
void Add(int num1)      //_Z3Addi, Mangled Name
{
    cout<<"Am Sum";
}
void Add(int num1, int num2)    //_Z3Addii
{
    cout<<"Am Sum";
```

```
}
void Add(float num1, int num2)  //_Z3Addfi
{
    cout<<"Am Sum";
}
void Add(int num1, float num2)  //_Z3Addif
{
    cout<<"Am Sum";
}
int main()
{
    return 0;
}
```

- Process or algorithm which generates mangled name is called as name mangling.
- ISO has not defined any specification on mangled name hence it may vary from compiler to compiler

Default Argument in C++

- Example:

```
#include<iostream>
using namespace std;
void sum( int num1, int num2 ){
int result = num1 + num2;
cout << "Result : " << result << endl;
}
void sum( int num1, int num2, int num3 ){
int result = num1 + num2 + num3;
cout << "Result : " << result << endl;
}
void sum( int num1, int num2, int num3, int num4 ){
int result = num1 + num2 + num3 + num4;
cout << "Result : " << result << endl;
}
void sum( int num1, int num2, int num3, int num4, int num5 ){
int result = num1 + num2 + num3 + num4 + num5;
cout << "Result : " << result << endl;
}
int main( void ){
sum( 10, 20 );
sum( 10, 20, 30 );
sum( 10, 20, 30, 40 );
sum( 10, 20, 30, 40, 50);
return 0;
}
```

- In C++, we can assign default value to the parameter of function. It is called as default argument.
- Using default argument, we can reduce developers effort.
- Default value can be:

    - constant
    - variable
    - macro
- Example-1:

```
void sum( int num1, int num2, int num3 = 0, int num4 = 0, int num5 = 0){
 int result = num1 + num2 + num3 + num4 + num5;
 cout << "Result : " << result << endl;
}
int main( void ){
 sum( 10, 20 );
 sum( 10, 20, 30 );
 sum( 10, 20, 30, 40 );
 sum( 10, 20, 30, 40, 50);
 return 0;
}
```

- Example-2:

```
int defaultArgument = 0;
void sum( int num1, int num2, int num3 = defaultArgument, int num4 =
defaultArgument, int num5 = defaultArgument ){
 int result = num1 + num2 + num3 + num4 + num5;
 cout << "Result : " << result << endl;
}
int main( void ){
 sum( 10, 20 );
 sum( 10, 20, 30 );
 sum( 10, 20, 30, 40 );
 sum( 10, 20, 30, 40, 50);
 return 0;
}
```

- Example-3:

```
 #define DEFAULT_VALUE 0
 void sum( int num1, int num2, int num3 = DEFAULT_VALUE, int num4 =
DEFAULT_VALUE, int num5 = DEFAULT_VALUE ){
 int result = num1 + num2 + num3 + num4 + num5;
 cout << "Result : " << result << endl;
 }
 int main( void ){
 sum( 10, 20 );
 sum( 10, 20, 30 );
 sum( 10, 20, 30, 40 );
 sum( 10, 20, 30, 40, 50);
 return 0;
```

- Default arguments are always given from right to left direction.
- We can assign, default argument to the parameters of member function as well as global function.
- When we separate , function declaration and definition then default argument must appear in declaration part:

```
 #include<iostream>
 using namespace std;
 #define DEFAULT_VALUE 0
 void sum( int num1, int num2, int num3 = DEFAULT_VALUE, int num4 = DEFAULT_VALUE,
int num5 = DEFAULT_VALUE );
 int main(){
 sum( 10, 20 );
 sum( 10, 20, 30 );
 sum( 10, 20, 30, 40 );
 sum( 10, 20, 30, 40, 50);
 return 0;
 }
 void sum( int num1, int num2, int num3, int num4, int num5 ){
 int result = num1 + num2 + num3 + num4 + num5;
 cout << "Result : " << result << endl;
 }
```

'extern' keyword in C++

- Using extern "C", we can invoke, C language function into C++ source code.
- If we declared any function using exten "C" then compiler do not generate mangled name for it.
- Consider the following MyFunctions.h Header file:

```
#ifndef MYFUNCTIONS_H_
#define MYFUNCTIONS_H_
extern "C"
{
int Method1();
int Method2();
int Method3();
int Method4();
}
#endif /* MYFUNCTIONS_H_ */
```

- Consider the MyFunctions.c file:

```
int Method1()
{
    return 100;
}
int Method2()
```

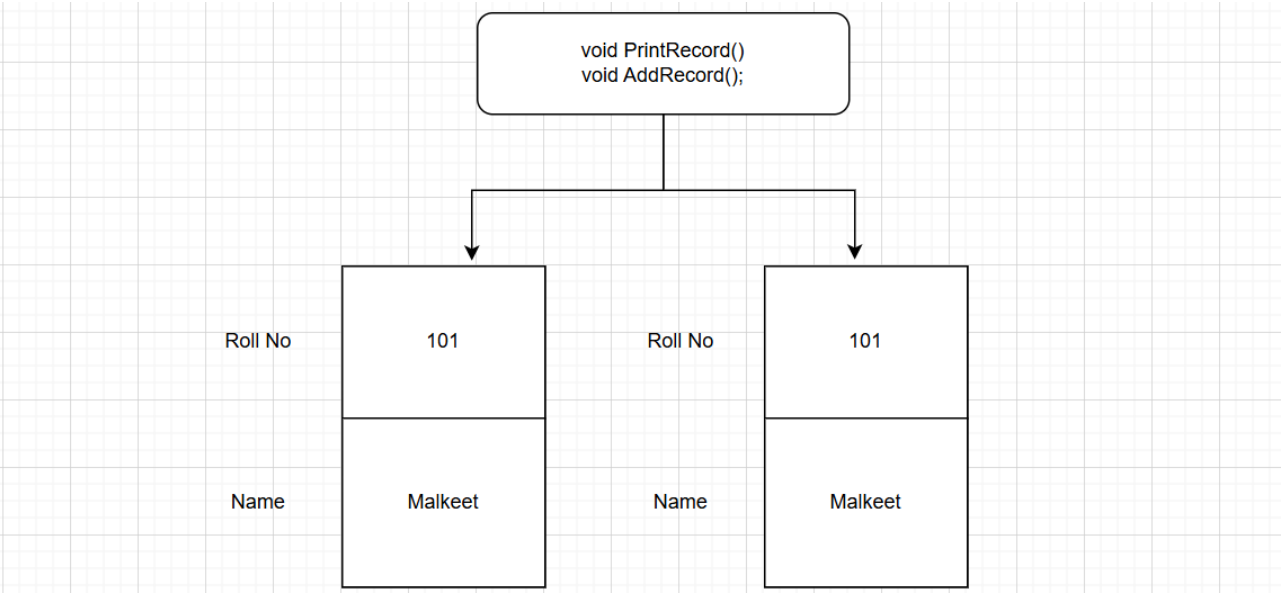
```
{
    return 200;
}
int Method3()
{
    return 300;
}
int Method4()
{
    return 400;
}
```

- Consider the Demo.cpp file:

```
#include <iostream>
#include "../MyHeaderFiles/MyFunctions.h"
using namespace std;
int main()
{
    cout<<"Value return by Method1: "<<Method1()<<endl;
    cout<<"Value return by Method2: "<<Method2()<<endl;
    cout<<"Value return by Method3: "<<Method3()<<endl;
    cout<<"Value return by Method4: "<<Method4()<<endl;
    return 0;
}
```

Object oriented concepts

- Only data members get space inside object. Member function do not get space inside object.
- Data members of the class get space once per object according their order of declaration inside class.
- Member function do not get space inside object, rather all the objects of same class share single copy of it.



- Size of object depdends on size of all the data members declared inside class.

- Characteristics of Object
    - State:
        - Value stored inside object is called as state of the object.
        - Value of the data member represents state of the object.
    - Behavior
        - Set of operations which are allowed to perform on object is called behavior of the object.
        - Member function defined inside class represents behavior of the object.
    - Idenitity
        - Value of any data member, which is used to identify object uniquely, is called as identity of the object.
        - When state of objects are same then its address can be considered as its identity.
- Lets revise Class & Object
    - Class Definition:
        - Class is collection of data members and member function.
        - Structure and behaviour of the object depends on class. Hence class is considered as a template / model / blueprint for object.
        - Class represents, group of objects which is having common structure and common behavior.
        - Class is an imaginary / logical entity.
        - Example: Book, Laptop, Mobile Phone, Car.
    - Object Definition:
        - Object is instance/variable of a class.
        - An entity which is having physical existance is called as object.
        - An entity, which is having state, behavior and identity is called as object.
        - Object is real time / physical entity.
        - Example: "More Effective C++", "MacBook Air", "iPhone 15", "Skoda Kushaq".

Empty class

- A class which do not contain any member is called as empty class.

```
Consider example:
class Student{
};
```

- Size of the object depends on data members declared inside class.
- According to above definition, size of object of empty class should be zero.
- According to oops concept, class is imaginary/logica term/entity and object is real time / physical term/entity. It means that object must get some space inside memory.
- According to Bjarne Stroustrup, size of object of empty class should be non zero.
- Due to compiler optimization, object of empty class get one byte space.

'this' Pointer

- As we know to process/manupulate state of the object we should call and define member function.

- If we call member function on object then compiler implicitly pass, address of current/ calling object as a argument to the member function. To catch/accpet address, compiler implicitly declare/create one paramater inside member function. Such parameter is called as this pointer.
- this is a keyword in C++.
- Parameter do not get space inside object. Since this pointer is a function parameter, it doesnt get space inside object.
- this pointer get space once per function call on stack section / segment.
- this pointer is a constant pointer. General type of this pointer is:

```
    ClassName *const this;
```

- To access members of the class, use of this keyword is optional. If we do not use this then compiler implicitly use this keyword.
- Using this pointer, data member and member function can communicate with each other.
- Hence this pointer is considered as a link / connection between them.
- Following functions do not get this pointer:
    - Global function
    - Static member function
    - Friend function
- this pointer is considered as first parameter of the member function.
- Example:

```
class Test{
private:
 int Num1;
 int Num2;
public:
 void SetData( /* Test *const this, */ int n1, int n1 ){
 cout << "Enter Num1 : ";
 cin >> this->Num1;
 cout << "Enter Num2 : ";
 cin >> this->Num2;
 }
};
int main( void ){
 Test t1;
 t1.SetData( 10, 20 ); //t1.SetData( &t1, 10, 20 );
 return 0;
}
```

- Definition:
    - this pointer is implicit pointer, which is available in every non static member function of the class and which is used to store address of current / calling object.
    - If name of data member and local variable / function parameter is same then preference will be given to local variable. In this case we should use this pointer before data members.
- Example:

```
class Test{
private:
 int Num1;
 int Num2;
public:
 void SetData( /* Test *const this, */ int Num1, int Num2 ){
 this->Num1=Num1;
 this->Num2=Num2;
 }
};
int main( void ){
 Test t1;
 t1.SetData( 10, 20 ); //t1.SetData( &t1, 10, 20 );
 return 0;
}
```

Getter and Setter methods in C++

- A member function of class, which is used to read state of the object is called as inspector / selector / getter function.
- A member function of class, which is used to modify state of the object is called as mutator / modifier / setter function.
- Example:

```
#include <iostream>
using namespace std;
class Test
{
private:
    int Num1;
    int Num2;
public:
    int getNum1() const {
        return Num1;
    }

    void setNum1(int num1) {
        Num1 = num1;
    }

    int getNum2() const {
        return Num2;
    }

    void setNum2(int num2) {
        Num2 = num2;
    }
};
int main()
```

```
{
    Test t1;
    //t1.Num1=100;  //NOT OK, Num1 is not visible
    //t1.Num2=100;  //NOT OK, Num2 is not visible

    t1.setNum1(10); //OK
    t1.setNum2(20); //OK

    cout<<"value of Num1:   "<<t1.getNum1();
    cout<<"value of Num2:   "<<t1.getNum2();
}
```

Constructor and its type in C++

- Member function of a class which is used to initialize the object is called as constructor.
- Note: Constructor do not create object rather it initializes object.
- Due to below reasons constructor is considered as special function of the class:
    - Its name is always same as class name.
    - It does not have any return type
    - It is designed to call implicitly
    - It gets called once per instance.
- We can not call constructor on object, pointer or reference explicitly.
- Example 1:

```
Test t1;
t1.Test( ); //Not OK
```

- Example 2:

```
Test t1;
Test *ptr = &t1; //ptr is pointer
ptr->Test( ); //Not OK
```

- Example 3:

```
Test t1;
Test &t2 = t1; //t2 is reference
t2.Test( ); //Not OK
```

- We can use any access specifier on constrctor:
- If constructor is public then we can create object inside member function of the class as non member function of the class.
- If constructor is private then we can create object inside member function of the class only.
- We can not declare constructor static, constant, volatile or virtual but we can declare constructor inline.

- Types of constructor:
  - Parameterless constructor
  - Parameterized constructor
  - Default constructor.
- Parameterless constructor:
  - It is also called as zero argument constructor or user defined default constructor.
  - Constructor of the class which do not take any parameter is called as parameterless constructor.
- Example:

```
Test( void ){
this->Num1 = 0;
this->Num2 = 0;
}
```

  - If we create object without passing argument, then compile invoke parameterless constrctor.
- Example:

```
Test t1; //Here on t1 parameterless constructor will call.
```

- Parameterized constructor
  - Constructor of the class which is having parameter(s) is called as parameterized constructor.
- Example:

```
Test( int value ){ //Single parameter constructor
this->Num1 = value;
this->Num2 = value;
}
Test( int Num1, int Num2 ){ // 2 parameter constructor
this->Num1 = Num1;
this->Num2 = Num2;
}
```

  - If we create object by passing arguments then parameterized constructor gets called.
  - Example:

```
Test t1( 10, 20 );
Test t2( 30 );
```

- We can overload constructor. Consider below code:

```
class Test{
private:
```

```
 int Num1;
 int Num2;
public:
 Test( ){ //Parameterless constructor
 this->Num1 = 0;
 this->Num2 = 0;
 }
 Test( int Num1, int Num2 ){ //Parameterized constructor
 this->Num2 = Num1;
 this->Num2 = Num2;
 }
};
```

- Constructor calling sequence depends on order of object declaration:
- Example:

```
Test t1(10,20), t2;
//First, parameterized constructor on t1 will call
//Then parameterless constructor on t2 will call
```

- Default constructor
  - If we do not define constructor inside class then compiler generate constructor for the class. Such constructor is called as default constructor.
  - Compiler never generate parameterized constructor. In other words, compiler generated constructor is zero argument / parameterless constructor.
  - Example:

```
class Test{
};
int main( void ){
Test t1; //On t1 Default constructor will call
Test t2( 10, 20 ); //Compiler error
return 0;
}
```

Aggregate Type and Aggregate initialization

- In C, below types are aggregate types whose object can be initialize using initializer list.
  - Array
  - Structure
  - Union
- Example:

```
 int arr[ 2 ] = { 100,200};
 struct Student s1 = { 101, "Malkeet", 446.89f };
```

- Aggregate class class following properties:
    - It does not contain private or protected non static data member.
    - It does not contain any user defined constructor.
    - It does not have base class
    - It does not contain virtual function
- Aggregate initialization:
- Example-1:

```
class Test{
public:
 int Num1;
 int Num2;
public:
 void printRecord( void ){
 cout << "Num1 Number : " << this->Num1 << endl;
 cout << "Num2 Number : " << this->Num2 << endl;
 }
};
int main( void ){
 Test t1{ 10, 20 }; //Aggregate initialization
 return 0;
}
```

- Example-2:

```
struct Employee
{
    int EmpId;
    string Name;
    void PrintRecord()
    {
        cout<<"EmpId:   "<<this->EmpId<<" Name: "<<this->Name<<endl;
    }
};
int main()
{
    //Aggregate Type (Array, Structure, Union), Aggregate Initialization,
Initializer List
    int Num1=90;
    int Num2=100;
    int Arr[10]={10,20,30,40};      //Valid, OK :Aggregate Initialization
    Employee emp={101,"Malkeet"};
    emp.PrintRecord();      //emp.PrintRecord(&emp);
    return 0;
}
```

- More example of Constructors of the class:

```
class Test{
private:
 int Num1;
 int Num2;
public:
 Test( void ){
 this->Num1 = 0;
 this->Num2 = 0;
 }
 Test( int value ){
 this->Num1 = value;
 this->Num2 = value;
 }
 Test( int Num1, int Num2 ){
 this->Num1 = Num1;
 this->Num2 = Num2;
 }
 void printRecord( void ){
 cout << "Num1 Number : " << this->Num1 << endl;
 cout << "Num2 Number : " << this->Num2 << endl;
 }
};
```

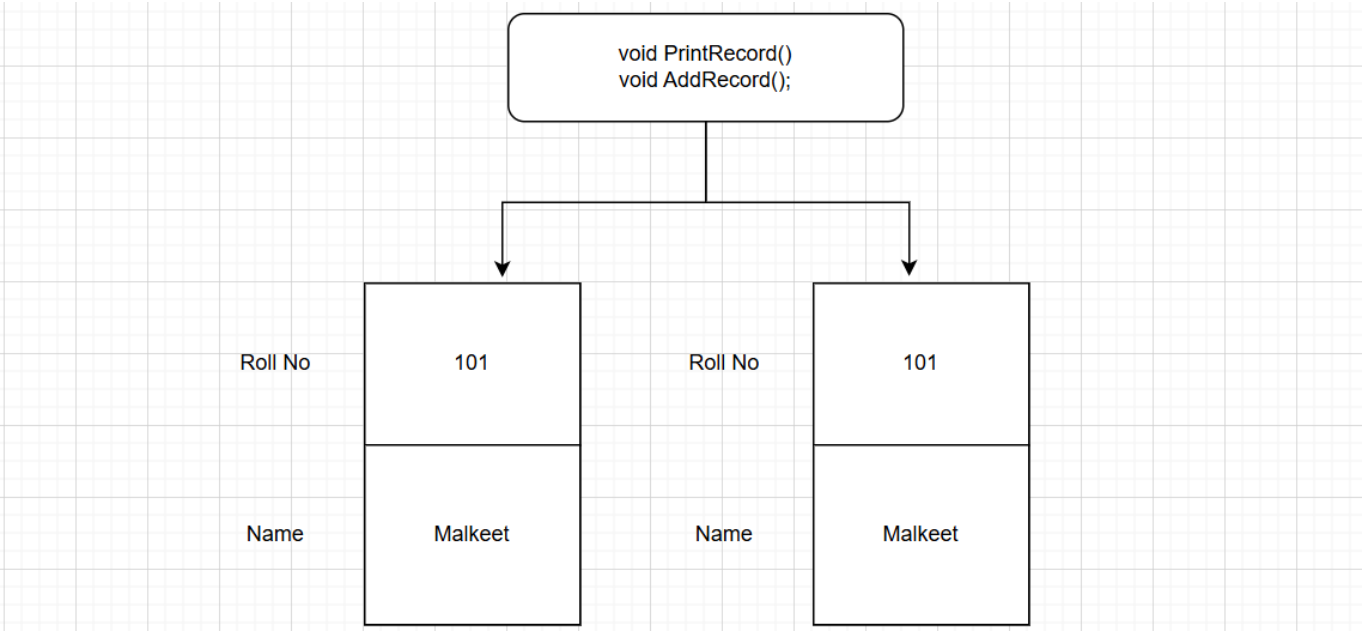
- Test t1;
  - Here on t1 object, parameterless constructor will call.
- Test t2(10);
  - Here on t2 object, single parameter constructor will call.
- Test t3(10, 20);
  - Here on t3 object, 2 parameter constructor will call.
- Test t4( );
  - It is declaration of t4 function which do not take any parameter and return object or Test type.
  - Constructor will not call here.
- Test t5 = 30;
  - It is same as Test t5(30).
  - Hence on t5, single parameter constructor will call.
- Test(40, 50);
  - It is anonymous object.
  - On object, 2 parameter constructor will call.
- Test t6 = 60, 70;
  - Compiler error.

Array of objects

- In C++ we can create array of objects aparts from creating the objects of the class one by one.
- We can process the objects as we process the elements of an array.
- Example:

```
```

```
Student sarr[3];    //Here, sarr is an array of the obects of class Student
```



- Example:

```
#include <iostream>
#include <string.h>
using namespace std;
class Student
{
private:            //Data Hiding
    int RollNo;     //Data Member
    string Name;

public:
    Student()
    {
        this->Name="No Name";
        this->RollNo=1234;
    }
    Student(int RollNo, string Name)
    {
        this->Name=Name;
        this->RollNo=RollNo;
    }
    void AddRecord(/*Student *const this*/)
    {
        cout<<"Enter Roll No:   "<<endl;
        cin>>this->RollNo;
        cout<<"Enter Name:  "<<endl;
        cin>>this->Name;
    }
    void PrintRecord(/*Student *const this*/)
    {
```

```
        cout<<"Roll No: "<<this->RollNo<<" Name:    "<<this->Name<<endl;
    }

    string getName(/*Student *const this*/){
        return this->Name;
    }

    void setName(/*Student *const this*/string Name) {
        this->Name = Name;
    }

    int getRollNo(/*Student *const this*/) {
        return this->RollNo;
    }

    void setRollNo(/*Student *const this*/int RollNo) {
        this->RollNo = RollNo;
    }
};
int main()
{
Student sarr[3];    //sarr is array of the objects of Student class

//Adding data in three objects
for(Student s:sarr)
{
    s.AddRecord();
}
//Print data of three objects
for(Student s:sarr)
{
    s.PrintRecord();
}
}
```

Constant (const) keyword in C++

Constant variable

- const and volatile are type qualifiers in C/C++.
- Once a variable initialized, if we dont want to modify state/value of the variable then we should use const qualifier.
- In C++, we can not modify value of the variable using pointer. Hence initialization of constant variable is mandatory.
- Example:

```
const int num1; //Not OK
const int num2 = 10; //OK
```

Constant data member

- Once a class data member initialized, if we dont want to modify value of the data member inside any member function of the class including constructor body then we should declare data member constant.
- We can initialize non constant data member using constructor member initializer list or constructor body but we must initialize constant data member using constructor member initializer list.
- Example: Non-Constant data member

```
#include<iostream>
using namespace std;
class Test{
private:
int Num1;
public:
Test( void ){
this->Num1 = 0;
this->Num1 = this->Num1 + 10; //OK
}
void showRecord( void ){
this->Num1 = this->Num1 + 2; //OK
cout << "Num1 : " << this->Num1 << endl;
}
void printRecord( void ){
this->Num1 = this->Num1 + 3; //OK
cout << "Num1 : " << this->Num1 << endl;
}
};
int main( void ){
Test t;
t.showRecord( ); //12
t.printRecord( ); //15
t.printRecord( ); //18
t.showRecord( ); //20
return 0;
}
```

- Example: constant data member

```
#include<iostream>
using namespace std;
class Test{
private:
const int Num1;
public:
Test( void ) : Num1( 10 ){
//this->Num1 = this->Num1 + 10; //Not OK
}
void showRecord( void ){
```

```
//this->Num1 = this->Num1 + 2; //Not OK
cout << "Num1 : " << this->Num1 << endl;
}
void printRecord( void ){
//this->Num1 = this->Num1 + 3; //Not OK
cout << "Num1 : " << this->Num1 << endl;
}
};
int main( void ){
Test t;
t.showRecord( ); //10
t.printRecord( ); //10
t.printRecord( ); //10
t.showRecord( ); //10
return 0;
}
```

Constant Member Function

- Example-1:

```
ClassName *const this;
```

- In above statement, this pointer is constant pointer which can store address of any non constant object.
- It means that this pointer can contain address of only one object but using this pointer we can modify state of the object.
- Example-2:

```
const ClassName *const this;
```

- In above statement, this pointer is constant pointer which can store address of any onstantobject.
- It means that this pointer can contain address of only one object and using this pointer we can not modify state of the object.
- If we want to modify state of the non constant object inside member function then type of this pointer should be "ClassName *const this" but If we dont want to modify state of the non constant object inside member function then type of this pointer should be "const ClassName *const this".
- If we dont want to modify state of the only current/calling object inside member function then we should declare member function constant.
- Example:

```
#include<iostream>
using namespace std;
class Test{
private:
int Num1;
```

```
public:
//Test *const this
Test( void ) : Num1( 10 ){
this->Num1 = this->Num1 + 10; //OK
}
//Test *const this
void showRecord( void ){
this->Num1 = this->Num1 + 2; //OK
cout << "Num1 : " << this->Num1 << endl;
}
//const Test *const this
void printRecord( void )const{
//this->Num1 = this->Num1 + 3; //Not OK
cout << "Num1 : " << this->Num1 << endl;
}
};
int main( void ){
Test t;
t.showRecord( ); //12
t.printRecord( ); //12
t.printRecord( ); //12
t.showRecord( ); //14
return 0;
}
```

- Note: Only state of current object will not be changed inside constant member function. Other object can be modifed inside constant member function.
- Example:

```
#include<iostream>
using namespace std;
class Test{
private:
int Num1;
public:
//Test *const this
Test( void ) : Num1( 10 ){
this->Num1 = this->Num1 + 10; //OK
}
//Test *const this
void showRecord( void ){
this->Num1 = this->Num1 + 2; //OK
cout << "Num1 : " << this->Num1 << endl;
}
//const Test *const this
void printRecord( void )const{
Test t;
t.Num1 = 20; //OK
t.showRecord( ); //It will print 22
}
};
```

```
int main( void ){
Test t;
t.printRecord( );
return 0;
}
```

- On non-constant object, we can call constant member function as well as non constant member function.
- Below functions are not allowed to declare as constant:
    - Global function
    - Static Member Function
    - Constructor
    - Destructor
- Since main function is global function, we can't make it constant.
- Why we can not declare global function constant?
    - According to concept, if we dont want to modify state of the current object inside member function then we should declare member function constant.
    - In other words, constant member function is designed to call on object.
    - Since global function is not designed to call on object, we can not make it constant.
- Use of 'mutable' keyword in C++
    - Exceptionlly, if we want to modify state of non constant data member inside constant member function then we should declare that data member mutable.
- Example:

```
#include<iostream>
using namespace std;
class Test{
private:
int num1;
int num2;
mutable int num3;
public:
Test( void ) : num1( 10 ), num2( 20 ), num3( 0 ){
}
void printRecord( void )const{
//this->num1 ++; //Not OK
cout<< "Num1 : " << this->num1 << endl;
//this->num2 ++; //Not OK
cout<< "Num2 : " << this->num2 << endl;
this->num3 ++; //OK
cout<< "Num3 : " << this->num3 << endl;
}
};
int main( void ){
Test t1;
t1.printRecord( );
return 0;
}
```

Constant Object

- If we want some objects to be constant and some objects to be non constant then we should use constant keyword.
- Example:

```
Test t1, t2; //non constant objects
const Test t2; //constant object
```

- On non-constant object we can call constant as well as non-constant member function.
- On constant object, we can call only constant member function.

```
#include<iostream>
using namespace std;
class Test{
private:
int Num1;
public:
//Test *const this
Test( ) : Num1( 0 ){
}
//Test *const this
void printRecord( void ){
cout << "printRecord" <<endl;
}
//const Test *const this
void printRecord( void )const{
cout << "const printRecord" <<endl;
}
};
int main( void ){
Test t1;
t1.printRecord( ); //printRecord
const Test t2;
t2.printRecord( ); //const printRecord
return 0;
}
```

Typedef

- typedef if a keyword in C/C++.
- Using typedef we can not define new Type / new user defined data type.
- If we want to give short and meaningful name then we should use typedef.
- Using typedef we can create alias for Type / class not for object.
- Example 1:

```
typedef unsigned short wchar_t;
```

- Example 2:

```
typedef struct Employee{
//TODO
}Employee_t;
```

- Example 3:

```
typedef basic_istream<char> istream;
typedef basic_ostream<char> ostream;
```

Reference

- Example 1:

```
int num1 = 10; //Initialization
int num2 = num1; //Initialization
```

- Example 2:

```
int num1 = 10; //Initialization
int *num2 = &num1; //Initialization
```

- Example 3:

```
int num1 = 10; //Initialization
int &num2 = num1; //Initialization, Here num2 is reference variable and num1 is
referent variable.
```

- Reference is an alias or another name given to the existing object.
- Using typedef we can create alias for class and using reference we can create alias for object.
- Example 1:

```
int main( void ){
 int num1 = 10;
 int &num2 = num1;
 ++ num1; //11
 ++ num2; //12
 cout<<"Num1 : "<< num1<<endl; //12
```

```
 cout<<"Num2 : "<< num2<<endl; //12
 return 0;
}
```

- We can create multiple references to the same memory location. Consider below code:
- Example 2:

```
 int main( void ){
 int num1 = 10;
 int &num2 = num1;
 int &num3 = num1;
 ++ num1; //11
 ++ num2; //12
 ++ num3; //13
 cout<<"Num1 : "<< num1<<endl; //13
 cout<<"Num2 : "<< num2<<endl; //13
 cout<<"Num3 : "<< num3<<endl; //13
 return 0;
 }
```

- Example 3:

```
 int main( void ){
 int num1 = 10;
 int &num2 = num1; //using num2, we can read/modify value of num1
 const int &num3 = num1; //using num3, we can read value but can not modify value
of num1
 ++ num2; //OK: 11
 //++ num3; //Not OK
 cout<<"Num1 : "<< num1<<endl;
 cout<<"Num2 : "<< num2<<endl;
 cout<<"Num3 : "<< num3<<endl;
 return 0;
 }
```

- We can not chnage referent of reference variable.
- Example 4:

```
 int main( void ){
 int num1 = 10;
 int num2 = 20;
 int &num3 = num1;
 num3 = num2;
 ++ num3;
 cout<<"Num1 : "<< num1<<endl; //21
 cout<<"Num2 : "<< num2<<endl; //20
 cout<<"Num3 : "<< num3<<endl; //21
```

```
return 0;
}
```

- We can create pointer to pointer but we can not create reference to reference.
- Example 5:

```
int main( void ){
int num1 = 10;
int &num2 = num1;
int &num3 = num2;
++ num1; //11
++ num2; //12
++ num3; //13
cout<<"Num1 : "<< num1<<endl;
cout<<"Num2 : "<< num2<<endl;
cout<<"Num3 : "<< num3<<endl;
return 0;
}
```

- Proecss of accessing value of the variable using pointer is called as dereferencing.
- NULL is macro whose value is 0 address.

```
int *ptr = NULL;
```

- Reference is automatically dereferenced constant pointer variable.

```
int main( void ){
 int num1 = 10;
 int &num2 = num1;
 //int *const num2 = &num1;
 cout << num2 <<endl;
 //cout << *num2 <<endl;
 return 0;
}
```

- How to check size of reference:

```
class Test{
private:
char &ch;
public:
Test( char &ch2 ) : ch( ch2 ){
}
};
int main( void ){
```

```
char ch1 = 'A';
Test t( ch1 );
size_t size = sizeof( t );
cout << "Size : " << size << endl;
return 0;
}
```

- What is the difference between pointer and reference
  - Initialization:
    - Pointer initialization is not mandatory but reference initialization is mandatory.
  - NULL:
    - We can initialize pointer to NULL but we can not initialize reference to NULL.
  - Pointer to pointer & reference to reference:
    - We can create pointer to pointer but we can not create reference to reference.
  - Array:
    - We can create array of pointers but we can not create array of reference.
  - Dereferencing:
    - To access the value of variable pointer need dereferencing but reference need not to do dereferencing.
- In C++, we can pass argument to the function by value, by address as well as by reference.
  - Passing argument by value

```
void swap_number( int x, int y ){
int temp = x;
x = y;
y = temp;
}
int main( void ){
int a = 10;
int b = 20;
swap_number( a, b ); //a , b are arguments; we are passing
passing it by value to the function
cout << "a : " << a << endl;
cout << "b : " << b << endl;
return 0;
}
```

  - Passing argument by address

```
//int *const x = &a;
//int *const y = &b;
void swap_number( int *const x, int *const y ){
int temp = *x;
*x = *y;
*y = temp;
}
int main( void ){
```

```
int a = 10;
int b = 20;
swap_number( &a, &b ); //adress of a , b are arguments; we are
passing passing it by address to the function
cout << "a : " << a << endl; //20
cout << "b : " << b << endl; //10
return 0;
}
```

- Passing argument by reference

```
//int &x = a; //int *const x = &a;
//int &y = b; //int *const y = &b;
void swap_number( int &x, int &y ){
int temp = x; //int temp = *x;
x = y; //*x = *y;
y = temp; //*y = temp;
}
int main( void ){
int a = 10;
int b = 20;
swap_number( a, b ); //Function call by reference
cout << "a : " << a << endl; //20
cout << "b : " << b << endl; //10
return 0;
}
```

- See below example:

```
#include<iostream>
using namespace std;
void print( int number ){
 cout<<"int : "<<number<<endl;
}
void print( int &number ){
 cout<<"int& : "<<number<<endl;
}
int main( void ){
 print( 10 ); //int : 10
 int value = 10;
 //print( value ); //error: call to 'print' is ambiguous
 return 0;
}
```

- We can not create array of references but we can create reference to array.
- Example:

```
 #include<iostream>
 using namespace std;
 int main( void ){
 //int& arr[ 3 ]; //Not OK: Array of references
 int arr1[ 3 ] = { 10, 20, 30 };
 int (&arr2)[ 3 ] = arr1; //arr2 is reference and arr1 is
referent
 for( int index = 0; index < 3; ++ index )
 cout<<arr2[ index ]<<endl;
 return 0;
 }
```

To be disscussed tommorow (10-06-2025)

- Exception handling in C++
- Object Oriented Design
- Object Oriented Programming
- OOPs Pillers