Day 1

Classification of languages

1. Machine level programming languages.
2. Low level programming languages.
3. High level programming languages.

Classification of high level programming languages

1. Procedure Oriened programming languages(POP)
    - Follows : TOP Down Approach
    - Example : C, PASCAL, BASIC etc.
2. Object Oriened programming languages(OOP)
    - Follows : Bottom Up Approach
    - Example : Simula, Smalltalk, C++, Java, C# etc.
    - Simula is first object oriented programming language
    - Smalltalk is first pure OOP language
    - More than 2000 lanuguages in world are OO.
3. Object Based programming languages
    - Example : Ada, Java Script, Modula-2, Visual Basic etc.
    - Ada is first object based programming language
4. Functional Oriented Programming language
    - Example : Python, Java
5. Rule Based Programming languages
    - Example : LISP, PROLOG

Top-Down Approach:

The top-down approach begins with a high-level, holistic view of the problem or system. It involves breaking down the problem into smaller, more manageable components or subproblems. Each component is then further decomposed until the desired level of detail is reached. This approach allows for a systematic and structured problem-solving process.

1. Advantages:

    - Clear Vision: Provides a clear overall vision of the system or project from the outset.
    - Modular Design: Facilitates modular design, making it easier to manage and modify individual components.
    - Risk Management: Allows for early identification and mitigation of potential risks.

2. Disadvantages:

    - Complexity Management: Can be challenging to manage complexity when dealing with large-scale systems.
    - Lack of Detail: Initial high-level design may lack sufficient detail for implementation.

Bottom-Up Approach:

The bottom-up approach starts with the smallest, most basic elements or components of the problem or system. These elements are then gradually integrated and combined to form larger, more complex components until the entire system is constructed. This approach emphasizes building the system from the ground up.

1. Advantages:
    - Detailed Implementation: Provides a solid foundation for implementation by focusing on the details from the beginning.
    - Incremental Development: Allows for incremental development and testing, reducing the risk of major issues.
    - Modularity: Supports modular design, enabling easy integration of new components.
2. Disadvantages:
    - Lack of Overall Vision: May lack a clear overall vision of the system in the early stages.
    - Integration Challenges: Integrating individual components into a cohesive system can be complex.

In summary, the top-down approach provides a structured decomposition of a problem, while the bottom-up approach focuses on building the system from its basic elements. Both approaches have their merits and limitations, and their effective combination can lead to successful problem-solving and system design.

C++ Introduction

- C++ is a object orieneted programming language which is derived from C(POP) and simula(OOP).
- C++ has borrowed efficiency from C and OOPs features form simula.
- Using C++ we can develop procedure orieneted as well as object orieneted code hence it is also called as hybrid programming language.
- It is invented in year 1979 by Bjarne Stroustrup.
- Initial name of the language was "C With Classes". In other words, C++ is better C languge in addition with Classes.
- In 1983, it is named to C++.
- Extension of the C++ source code file must be ".cpp".
- Standardizing C++ is a job of ISO Working group.
- Following are the C++ standards:
    1. C++ 98 : 1998
    2. C++ 03 : 2003
    3. C++ 11 : 2011
    4. C++ 14 : 2014
    5. C++ 17 : 2017
    6. C++ 20 : 2020
    7. C++ 23 : 2023
- Before 1998, book "Annoted C++ reference manual-Bjarne Stroustrup" was used as a reference.

Software Development Kit(SDK)

- SDK = Language tools + Documenation + Libraries + Runtime environment.
- Consider Language tools

    1. Editor : MSVS Code
    2. Preprocessor : CPP
    3. Compiler : g++
    4. Assembler : as
    5. Linker : ld
    6. Loader : OS API
    7. Profiler : valgrind
    8. Debugger : gdb
    9. Build tool : make
- Documentation
    1. Man pages
    2. https://en.cppreference.com/w/cpp/language
- Libraries
    1. glibc.so : C
    2. libstdc++.so : C++
- Runtime environment
    1. C++ runtime.

Editor

- Microsoft Windows : Notepad, Notepad++, Wordpad, EditPlus, MSVS Code
- Linux : vi/vim, gedit/textedit, MSVSCode
- Using editor, we can create file, edit it, delete it etc.

Integrated Development Environment(IDE)

- Example:
    1. Turbo C
    2. Microsoft Visual Studio
    3. eclipse
    4. NetBeans
    5. Intellij Idea
    6. PyCharm
    7. Spyder
    8. Xcode

Flow of execution:

- Link : https://www.tenouk.com/ModuleW.html

Data Type

- Data type of any variable describes 4 things
    1. Memory : How much memory is required to store the data
    2. Nature : Which type of data is allowed to store inside memory
    3. Operation : Which operations are allowed to perform of data stored inside memory
    4. Range : Set of values allowed to store inside memory

Types of data types

1. Fundamental Data types
    - void : NA
    - bool : 1 byte
    - char : 1 byte
    - wchar_t : 2 bytes
    - int : 4 bytes
    - float : 4 bytes
    - double : 8 bytes
2. Derived Data types
    - Array
    - Function
    - Pointer
    - Reference
3. User Defined Data types
    - enum
    - union
    - structure
    - class
4. Type Modifiers
    - short
    - long
    - signed
    - unsigned
5. Type Qualifier
    - const
    - volatile

Software Development Life Cycle(SDLC)

```
1. Requirement
2. Analysis
3. Design
4. Coding/Implementation
5. Testing
6. Installation/Deployment
7. Maintenance
```

Object Oriented Programming Structure/System[OOPS]

- It is a process / methodology that we can implement using any OOP language.
- It is invented by Dr. Alan Kay in 1960. He is inventor of Simula.
- There are 4 major and 3 minor pillars/parts/elements of OOPS
- There are 3 phases in OO software development:
    1. Object Orieneted Analysis(OOA)

        - Analysis in the context on class and Object
    2. Object Orieneted Design(OOD)
        - Design in the context on class and Object
    3. Object Orieneted Programming(OOP) -Programming in the context on class and Object
- Grady Booch:
    - Inventor of Unified Modelling Lamguage(UML)
    - Author of "Object Oriented Analysis and Design with application" book.

4 major pillars of oops

```
1. Abstraction      :   To achieve simplicity
2. Encapsulation    :   To achieve data hiding
3. Modularity       :   To minimize module dependancy
4. Hierarchy        :   To achieve reusability
    - Has-a/Part-of =>  Association => Special forms { Compositon and Aggregation
}
    - Is-a/Kind-of  =>  Inheritance/Generalization
    - Use-a         =>  Dependancy
    - Creates-a     =>  Instantiation
```

- If we want to classify any language OO then it must support all above features.

3 minor pillars of oops

```
1. Typing/Polymorphism  :   To reduce maintenance of the system
    - Compile time polymorphism
        1. Function Overloading
        2. Operator Overloading
        3. Template
    - Runtime polymorphism
        1. Function Overriding
2. Concurrency          :   To utilize CPU efficiently
3. Persistence          :   To maintain state of object on secondry storage.
```

- Word minor means, if language supports to above features then it will be useful but not essential to classify language OO.

Structure in C:

- Consider following examples:
    1. Date : day, month year
    2. Color : red, green blue
    3. Time : hour, minute, second
    4. Employee : name, empid, salary
    5. Student : name, rollNumber, marks

    6. Account : number, type, balance
- If we want to group related data elements together then we should use structure.
- It is user defined data type. Also called as abstract data type.
- Structure can contain:
    1. Nested type definition
    2. Declaration of variables
    3. Declaration of pointers.
- We can not declare/define function inside structure.
- If we declare structure inside function then it is called local structure. We can not create pointer/object of local structure outside function.
- If we declare structure outside function then it is called as global structure.
- Variable declared inside structure get space after creating object of structure.
- Using object, if we want to access memebers of structure then we should use dot operator.
- Using pointer, if we want to access memebers of structure then we should use arrow operator.
- In C, We can not define function inside function/structure. In other words, all the functions in C are global. Global functions are designed to access global data hence giving controlled access to data is difficult.

Object

- An entity, which is having physical existance is called object.
- It is also called as variable/instance.
- If we create object of the class then only data members get space inside it.

this pointer

- this is a keyword in C++.
- It is a function parameter hence it doesnt get space inside object.
- It gets space per function call.
- If we call member function on object then compiler implicitly pass address of calling object/current object as argument to the member function. To store address of the argument, compiler implicitly declare on parameter inside function. It is called this pointer.
- Programmer can not declare this pointer explicitly.
- General type of this pointer is: ClassName *const this;

Access Specifier

- If we want to control visibility of members of structure/class then we should use access specifier.
- There are 3 access specifiers in C++:
    1. private (-)
    2. protetced(#)
    3. public (+)
- Process of declaring data member private is called data hiding/data encapsulation.
- In C++, structure members are by default considered as public.
- In C++, class members are by default considered as private.
- Variable declared inside class/class scope is called data member.
- Data member is also called as field/attribute/property.
- Function implemented inside class is called member function.

- Member function is also called as method/operation/behavior/message
- Class is a collection of data member and member function.
- Object is a variable/instance of a class.
- Instantiation:
    - Process of creating object from class is called instantiation.
    - instantiation in C
        - struct StructureName identifier;
    - instantiation in C++
        - ClassName identifier;
        - Employee emp;
    - instantiation in Java
        - ClassName identifier = new ClassName( );

Coding Convention

1. Hungerian Notation

    - Recommended for C/C++
    - Example
        - int iNum1;
        - double dNum2;
        - char szText[30];

2. Camel Case Convention

    - Recommended for Java/MS.NET
    - Example
        1. main( )
        2. parseInt( )
        3. showMessageDialog( )
        4. addNumberOfDays()
    - In this case, excluding first word, first character of each word must be in upper case.
    - We should use this convention for:
        1. Function parameter
        2. Local and global variable
        3. Data member
        4. Member Function

3. Pascal Case Convention

    - Recommended for Java/MS.NET
    - Example
        1. System
        2. StringBuffer
        3. NullPointerException
        4. IndexOutOfBoundsException
    - In this case, including first word, first character of each word must be in upper case.
    - We should use this convention for:
        1. Type Names(enum, union, structure, class, namespace etc)

2. File Name

- Process if calling member function on object is called message passing.

```
int main( void )
{
    Employee emp;
    emp.acceptRecord( );    //Message Passing
    //here acceptRecord() function is called on emp;
    emp.printRecord( );     //Message Passing
    //here printRecord() function is called on emp;
    return 0;
}
```

```
int main( void )
{
    Employee emp;
    emp.Employee::acceptRecord( );  //Message Passing
    emp.Employee::printRecord( );       //Message Passing
    return 0;
}
```

Namespace

- If name of local and global variable is same then preference is always given to the local variable.
- Using scope resolution operator, we can access value of global variable inside function.

```
int num1 = 10;  //Global Variable
int main( void )
{
    int num1 = 20;  //Local Variable
    printf("Num1    :   %d\n", ::num1); //10
    printf("Num1    :   %d\n", num1);   //20

    {//Start of Block
        int num1 = 30;  //Local Variable
        printf("Num1    :   %d\n", ::num1); //10
        printf("Num1    :   %d\n", num1);   //30
    }
    return 0;
}
```

- In same scope, we can not use same name to multiple elements:

```
int num1 = 10;  //OK
int num1 = 20;  //error: redefinition of 'num1'
```

```
int main( void )
{
    return 0;
}
```

- namespace is C++ language feature that is designed:
    1. to avoid name clashing/collision/ambiguity
    2. to group/organize funtionally equivalent/related types together.
- namespace is a keyword in C++.
- We can define namespace inside another namespace or global. In other words, we can not define namespace inside function/class.
- We can declare/define following things inside namespace:
    1. variable
    2. Function
    3. Types(enum, union, structure, class)
    4. another namespace
- We can not define main function inside ser defined namespace.
- If we want to access members of namespace then we should use namespace name and :: operator.

```
namespace na
{
    int num1 = 10;  //OK
}
int main( void )
{
    printf("Num1    :   %d",na::num1);
    return 0;
}
```

- If name of the namespaces are different then name of members of namespace may/may not be same.

```
namespace na
{
    int num1 = 10;  //OK
    int num3 = 30;  //OK
}
namespace nb
{
    int num2 = 20;  //OK
    int num3 = 40;  //OK
}
int main( void )
{
    printf("Num1    :   %d\n",na::num1);
    printf("Num3    :   %d\n",na::num3);

    printf("Num2    :   %d\n",nb::num2);
```

```
    printf("Num3    :   %d\n",nb::num3);
    return 0;
}
```

- Inside same file as well as different final we can give same name to the namespace.
- If name of the namespaces are same then name of the members must be different.

```
namespace na
{
    int num1 = 10;  //OK
    int num3 = 30;  //OK
}
namespace na
{
    int num2 = 20;  //OK
    int num3 = 30;  //NOT OK : error: redefinition of 'num3'
}
int main( void )
{
    printf("Num1    :   %d\n", na::num1);
    printf("Num3    :   %d\n", na::num3);

    printf("Num2    :   %d\n", na::num2);
    return 0;
}
```

- We can define namespace inside another namespace. It is called as nested namespace.

```
int num1 = 10;
namespace na
{
    int num2 = 20;  //OK
    namespace nb
    {
        int num3 = 30;  //OK
    }
}
int main( void )
{
    printf("Num1    :   %d\n", ::num1); //10
    printf("Num2    :   %d\n", na::num2);   //20
    printf("Num3    :   %d\n", na::nb::num3);   //30
    return 0;
}
```

- Global members are considered as a part of global namespace.
- If we want to access members of namespace then we should use:
    1. F.Q.name

2. Using directive

```
namespace na
{
    int num1 = 10;
}
int main( void )
{
    printf("Num1    :   %d\n", na::num1);
    using namespace na; //using directive
    printf("Num1    :   %d\n", num1);
    return 0;
}
```

```
namespace na
{
    int num1 = 10;
}

int main( void )
{
    int num1 = 20;  //Local variable
    using namespace na; //using directive
    printf("Num1    :   %d\n", num1);   //20
    printf("Num1    :   %d\n", na::num1);   //10
    return 0;
}
```

- iostream is a standrad header file of C++ available in standard directory(/usr/include).
- std is standard namespace of C++ which is defined inside header file.
- cin, cout, cerr and clog are standard stream objects of C++ declared inside std namespace.

```
//Header File : <iostream>
namespace std
{
    extern istream cin;
    extern ostream cout;
    extern ostream cerr;
    extern ostream clog;
}
```

- How will you access above object.
- Method 1

```
1. std::cin
2. std::cout
3. std::cerr
4. std::clog
```

- Method 2

```
using namespace std;
1. cin
2. cout
3. clog
4. cerr
```

- If we want to print value of the variable on console then we should use cout and insertion operator(<<).

```
int main( void )
{
    int number = 10;
    cout<<number<<endl;
    cout<<"Number   :   "<<number<<endl;
    return 0;
}
```

- If we want to accept value of the variable from console then we should use cin and extraction operator(>>).

```
int main( void )
{
    int number;
    cout<<"Number   :   ";
    cin>>number;
    cout<<"Number   :   "<<number<<endl;
    return 0;
}
```

- In C, Escape sequence is a character which is used to format the ouput.
- In C++, manipulator is a function which is used to format the ouput.
- Example: 1.endl : 2. setw, fixed, scientific, hex, dec, oct

Day 2

Function Overloading

- According to oops, If implementation of a function is logically same/equivalent then we should give same name to the function.
- With the help of following rules, we can give same name to the function.

1. If we want to give same name to 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;
}
```

2. If we want to give same name to function and if number of parameters passed to the function 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;
}
```

3. If we want to give same name to the function and if number of parameters passed to the function 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;
}
```

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

```
int sum( int num1, int num2 )   //OK
{
    int result = num1 + num2;
    //cout<<"Result   :   "<<result<<endl;
    return result;
}
void sum( int num1, int num2 )  //Not OK
{
    int result = num1 + num2;
    cout<<"Result   :   "<<result<<endl;
}
```

- Process of defining function with same name using above rules us called function overloading.
- For function overloading minimum 2 functions are required.
- In simple words, function with same name and different signature is called function overloading.
- Functions which are taking part in function overloading are called overloaded functions.
- According to oops, If implementation of a function is logically same/equivalent then we should overload function.
- Note : For function overloading, functions must be exist inside same scope.
- return type is not considered int function overloading.
    - Returning value from function / catching value which is returned from function is optional hence return type is not considered in function overloading.
- Except destructor and main function, we can overload any function in C++.
- If we define function is C++ then by looking towards name of the function and type of parameter passed to the function, compiler generates unique name for that function. It is called mangled name.
- Process/algorithm which decides mangled name for the function is called name mangling.

```
int sum( int num1, int num2 )       //__Z3sumii
{   }
float sum( int num1, float num2 )   //__Z3sumif
{   }
double sum( int num1, float num2, double num3 ) //__Z3sumifd
{   }
```

Characteristics of object

- Data members of the class get space once per object according to 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. Hence size of object depends on size of data members declared inside class.

State

- Data/value stored inside object is called state of the object.

- Value of the data member represent state of the object.

Behavior

- Set of operations, that we can call/perform on object is called behavior of the object.
- Member function, defined inside class represent behavior of the object.

Identity

- A value of data member that is used to identify object uniquely is called its unique identity.
- If state of object is same then its address is considered as its identity.

Class

- Definition:
    1. Class is collection of data member and member function.
    2. structure and behavior of object depends on class hence class is considered as a template/model/blueprint for object.
    3. class represents set/group of all such objects which are having common structure and common behavior.
- Class is a imaginary entity.
- Example : Car, Book, Laptop etc.

Object

- Defintion
    1. Any entity, which has physical existance is called object.
    2. variable/instance of a class is called object.
    3. An entity, which has state,behavior and identity is called object.
- Object is real time entity.
- Example : Tata nano, Let Us C, MacBook Air etc.

Empty class

- A class which do not contain any member is called empty class.
- According to definition, size of object of empty class should zero.
- According Bjarne Stroustrup, since object is a physical/real time entity, it must get some space inside memory. According to hime size of object of empty class should be non zero.
- Due to compiler optimization, object of empty class get one byte space inside memory.

How objects share single copy of member function?

1. Understand problem statement and analyze from percepective of class and object.
2. Define class and declare data memeber/fields inside it.
3. To store state, create object of the class(Instantiation).
4. To process(accept/print/set/get) state of the object we should call member function on object(message passing).
    - If we call member function on object then compiler implicitly pass, address of that object as a argument to the function.

    - To store address of argument, compiler implicitly declare on pointer as a function parameter. It is called this pointer.
    - this pointer is constant pointer which is used to store address of current object/calling object.
    - this is a keyword.
    - this pointer is available inside non static member functions only.
    - using this pointer, data member and member function can communicate with each other hence it is considered as a link/connection between them.
    - Following functions do not get this pointer:
        1. Global Function
        2. Static Member Function
        3. Friend Function

- Definition: This pointer is implicit pointer, that is available in every non static member function of the class which is used to store address of current/calling object.

Initialization and Assignment

```
int num1 = 10;  //Initialization
int num2;
num2 = num1;    //Assignment
num2 = 20;    //Assignment
```

- Initialization is the process of storing value inside variable during its declaration.
- We can initialize variable only once.
- Assignment is the process of storing value inside variable after its declaration.
- We can assign value to the variable multiple times.

Constructor

- It is a member function of a class which is designed to intialize object.
- Due to following reasons, constructor is considered as special function of the class:
    1. Its name is same as class name.
    2. It doesn't have any return type.
    3. It is designed to call implicitly.
    4. It gets called once per object.
- We can not call constructor on object, pointer/reference explicitly. Compiler call it implicitly.

```
int main( void )
{
    Complex c1;
    //c1.Complex( );  //Not OK

    Complex *ptr = &c1;
    //ptr->Complex( );    //Not OK

    Complex &c2 = c1;
    //c2.Complex( );  //Not OK
```

```
    return 0;
}
```

- Non static member functions are designed to call on object and static member functions are designed to call on class name.
- We can use any access specifier on constructor.
- If constructor is public then we can create object of the class inside member function as well as non member function.
- If constructor is private then we can create object of the class inside member function only.
- We can not declare constructor static, constant, volatile / virtual. We can declare constructor inline only. inline is a keyword in C++.
- In C++, we can not call constructor from another constructor.
- Constructor calling sequence is depends on order of object declaration.

Types of constructor

```
1. Parameterless constructor
2. Parameterized constructor
3. Default constructor
```

- Note : Copy constructor is single parameter constructor hence it is considered as parameterized constructor

Parameterless constructor

- A constructor which do not take any parameter is called parameterless constructor.

```
Complex( void ) //parameterless constructor
{
    this->real = 0;
    this->imag = 0;
}
```

- It is also called as user defined default constructor / zero argument constructor.

```
Complex c1; //On c1 parameterless constructor will call.
```

- If we create object without passing argument then parameterless constructor gets called.

Parameterized Constructor

- A constructor which takes parameter is called parametered constructor.

```
Complex( int real, int imag )
{
    this->real = real;
    this->imag = imag;
}
```

- If we create object by passing argument then Parameterized constructor gets called.

Default constructor

- If we do not define constructor inside class then compiler generates one constructor for the class by default. It is called default constructor.
- Default constructor is zero argument constructor. In other words, compiler do not generate default parameterized constructor.
- We can define multiple constructor's inside class. It is called constructor overloading.
- If name of local and global variable is same then preference is given to the local variable.
- If name of local variable and data member is same then preference is given to the local variable.
- If name of local variable and data member is same then we should use this keyword before data member.

```
Complex( int real, int imag )
{
    this->real = real;
    this->imag = imag;
}
```

Constructor member initializer list

- If we want to intialize data members according to their order of declaration inside class then we should use constructor's member initializer list.

```
class Test
{
private:
    int num1;
    int num2;
    int num3;
public:
    Test( void ) : num1( 10 ), num2( 20 ), num3( 30 )
    {   }
    Test( int num1, int num2, int num3 ) : num1( num1 ), num2( num2 ), num3( num3
)
    {   }
};
```

- We can not initialize array using constructor's member intializer list.

Default Argument

- In C++, we can assign default value to the parameter of function. It is called default argument and parameter is called optional parameter.

```
//num3, num4    =>  Optional Parameters
//0 => Default argument
void sum( int num1, int num2, int num3 = 0, int num4 = 0 )
{
    int result = num1 + num2 + num3 + num4;
    cout<<"Result   :   "<<result<<endl;
}
int main( void )
{
    sum(10,20);
    sum(10,20,30);
    sum(10,20,30,40);
    return 0;
}
```

- Default arguments are always assigned from right to left direction.
- We can assign default argument to the parameter of global function as well as member function.

```
class Test
{
private:
    int num1;
    int num2;
    int num3;
public:
    Test( int num1 = 0, int num2 = 0, int num3 = 0 )
                                    : num1( num1 ), num2( num2 ), num3( num3 )
    {   }
};
```

Constant Variable

- const is type qualifier in C/C++
- If we dont want to modify state of the variable then we should declare it constant.
- In C++, it is mandatory to intialize constant variable.

```
const int num1; //Not OK
const int num2 = 10;    //OK
num2 = 20;  //Not OK
```

- In C++, we can declare following entities constant:
    1. Local and global variable
    2. Function parameter
    3. Data member
    4. Member Function
    5. Object
- We can not declare following entities constant
    1. Global Function
    2. Class
    3. Constructor
    4. Destructor
    5. Static Member Function

Constant data member

- Once 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.
- If data member is constant then it is mandatory to intialize it using constructor's member initializer list.

```
class Test
{
private:
    const int number;
public:
    Test( void ) : number( 10 )
    {
        //++ this->number = 10; //Not OK
    }
};
```

Constant member function

- We can not declare global function constant but we can declare member function constant.
- Inside member function, If we don't want to modify state of current object then we should delcare member function constant.
- Declaration of this pointer inside non constant member function:

```
ClassName *const this;
```

- Declaration of this pointer inside constant member function:

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

- If member function is constant then local variable will not be considered as constant.

- On non constant object, we can call constant as well as non constant member function.
- Generally, we should declare read-only function constant.
- We can not declare following function's constant:
    1. Constructor
    2. Destructor
    3. Static Member Function
    4. Global Function
- mutable is a keyword in C++.
- Inside constant member function, if we want to modify value of the non constant data member then we should declare data member mutable.

Constant object

- If we dont want to modify state of the object then instead of declaring data member constant, we should declare object constant.

```
int main( void )
{
   const Test t1;
   Test t2;
   Test t3;
    return 0;
}
```

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

typedef and reference

- typedef is a keyword in C/C++.
- If we want to to give short name / meaningful name to the exisiting data type then we should use typedef.
- Using typedef, we can not create/define new data type rather we create alias for the exisiting data type.

```
typedef unsigned long size_t;
typedef unsigned short wchar_t;
typedef basic_istream<char> istream;
typedef basic_ostream<char> ostream;
typedef basic_string<char> string;
```

- If we want to create alias for the object(not for the literals/constant) then we should use reference in C++.
- Consider example:

```
int num1 = 10;      //Intialization of num1
int num2 = num1;    //Intialization of num2
```

- Consider example:

```
int number = 10;            //Intialization of number
int *ptrNumber = &number;   //Intialization of ptrNumber
```

- Process of accessing value of the variable using pointer is called dereferencing.
- Consider example:

```
int num1 = 10;      //Intialization of num1
int &num2 = num1;
```

- In above code, num2 is reference variable and num1 is referent variable.
- Reference is C++ language feature, which is alias/another name given to the existing memory location.

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

- referent can have multiple references:

```
int main( void )
{
    int num1 = 10;
    int &num2 = num1;
    int &num3 = num1;
    return 0;
}
```

- In C++, we can not create reference to reference:

```
int main( void )
{
    int num1 = 10;
    int &num2 = num1;   //num2 is reference to num1
    int &num3 = num2;   //num3 is reference to num2( actually num1 )
    return 0;
}
```

- Once reference is intialized we can not change its referent:

```
int main( void )
{
    int num1 = 10;
    int num2 = 20;
    int &num3 = num1;
    num3 = num2;
    ++ num1;
    return 0;
}
//Output : 21,20,21
```

- Definition of reference:
    1. (From the perspective of programmer) Reference is alias or another name given to the exisiting variable/object.
    2. (From the perspective of compiler) Reference is automatically dereferenced constant pointer variable.
- If we want to mimize complexity of pointer then we should use reference.
- In C++, we can pass argument to the function using 3 ways:
    1. by value
    2. by address
    3. by reference
- Passing argument by value:

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

- Passing argument by address:

```
void swap( int *x, int *y )
{
    int temp = *x;
    *x = *y;
    *y = temp;
}
```

```
int main( void )
{
    int a = 10;
    int b = 20;
    swap( &a, &b ); //a and b are arguments passing to function by address
    cout<<"a    :   "<<a<<endl; //20
    cout<<"b    :   "<<b<<endl; //10
    return 0;
}
```

- Passing argument to the function by reference

```
//int &x = a;
//int &y = b;
void swap( int &x, int &y )
{
    int temp = x;
    x = y;
    y = temp;
}
int main( void )
{
    int a = 10;
    int b = 20;
    swap( a, b );   //a and b are arguments passing to function by reference
    cout<<"a    :   "<<a<<endl;
    cout<<"b    :   "<<b<<endl;
    return 0;
}
```

Exception handling

- Compiler Error

```
int main( void )
{
    return 0    //Compiler Error. ; missing
}
```

```
- If we make syntactical mistake is a code then compiler generates error
```

- Linker Error

```
int main( void )
{
```

```
    void print( void ); //Local Function Declaration
    print( );   //Function Call =>  Linker Error
    return 0;
}
```

```
- Without definition, if we try to access any member then we get linker error
```

- Bug

```
int main( void )
{
    int number = 10;
    if( number = 0 )    //Here we should use == but = is used.
        cout<<"If :   "<<number<<endl;
    else
        cout<<"Else :   "<<number<<endl;
    return 0;
}
```

```
- Logical error / syntactically valid but logically invalid statement represents
bug.
```

- Exception

```
int main( void )
{
    int num1;
    cout<<"Num1 :   ";
    cin>>num1;  //10

    int num2;
    cout<<"Num2 :   ";
    cin>>num2;  //0

    int result = num1 / num2;
    cout<<"Result   :   "<<result<<endl;
    return 0;
}
```

- Runtime error is called exception.
- if we give wrong input to the application then we get runtime error/exception.
- Operating System Resources:
    1. Memory

    2. File
    3. Thread
    4. Scoket
    5. Network Connection
    6. I/O Devices
    7. System calls
- OS resources are limited hence we should use it carefully. In other words, we should avoid their leakage.
- Why we should handle Exception?
    1. To avoid resource leakage
    2. To handle all the runtime error centrally.
- How to handle exception?
    1. try
        - If we want to keep watch on statements which can generate exception then we should use try block/handler.
        - We can not define try block after catch block
    2. catch
        - If we want to handle exception thrown from try block then we should use catch block/handler
        - try block may have multiple catch blocks.
        - We can not define catch block before try block.
        - A catch block, which can handle all type exception is called generic/default/ catch block / catch all handler.
        - Generic catch block must appear after all specific catch block.
    3. throw
        - It is used to generate new exception.
        - throw is a jump statement.

Dynamic Memory Management

- Dynamic memory management in C:
    1. void* malloc(size_t size);
    2. void* calloc(size_t count, size_t size);
    3. void* realloc(void *ptr, size_t newSize);
    4. void free(void *ptr);
- Using malloc, calloc and realloc, we can allocate memory for the variable on heap section. To deallocate memory of dynamic variable, we should use free() function.
- Consider memory allocation for single variable.

```
int *ptr = ( int* )malloc( sizeof( int ) );
if( ptr != NULL )
{
    *ptr = 125; //Dereferencing
    printf("Value   :   %d\n", *ptr); //Dereferencing
    free( ptr );
}
```

*or

```
int *ptr = ( int* )calloc( 1, sizeof( int ) );
if( ptr != NULL )
{
    *ptr = 125; //Dereferencing
    printf("Value   :   %d\n", *ptr); //Dereferencing
    free( ptr );
}
```

- Consider memory allocation for array.

```
int *ptr = ( int* )malloc( 3 * sizeof( int ) );
if( ptr != NULL )
{
    int index;
    ptr[ 0 ] = 10;
    ptr[ 1 ] = 20;
    ptr[ 2 ] = 30;
    for( index = 0; index < 3; ++ index )
        printf("%d\n", arr[ index ] );
    free( ptr );
}
```

- or

```
int *ptr = ( int* )calloc( 3 , sizeof( int ) );
if( ptr != NULL )
{
    int index;
    ptr[ 0 ] = 10;
    ptr[ 1 ] = 20;
    ptr[ 2 ] = 30;
    for( index = 0; index < 3; ++ index )
        printf("%d\n", arr[ index ] );
    free( ptr );
}
```

- In C++, using new operator, we can allocate memory on heap and using delete operator we can deallocate memory from heap.
- Consider memory allocation for single variable.

```
int *ptr = new int; //Memory Allocation
//int *ptr = (int*)::operator new( sizeof(int));

*ptr = 125; //Dereferencing
```

```
printf("Value   :   %d\n", *ptr); //Dereferencing

delete  ptr;    //Memory Deallocation
//::operator delete( ptr );
```

- Consider memory allocation for array.

```
int *ptr = new int[ 3 ];
//int *ptr = ::operator new[ ]( 3 * sizeof( int ) );

ptr[ 0 ] = 10;
ptr[ 1 ] = 20;
ptr[ 2 ] = 30;
for( int index = 0; index < 3; ++ index )
    printf("%d\n", arr[ index ] );

delete[ ] ptr;
//::operator delete[ ]( ptr);
```

- If new operator fails to allocate memory then it throws bad_alloc exception.
- If we create dynamic object using malloc then constructor do not call but if we create dynamic object using new operator them constructor gets called.

Friend Function and Friend class

- friend is a keyword in C++.
- Friend function is a non member function of the class which is designed to access private and protected members of the class.
- Class can not be friend of function but function can be friend of class.
- We can declare main function friend inside class.
- We can declare friend statement inside any section of the class.
- If function is a friend of class then it is not considered as a member of the class.
- We can declare single function as a friend into multiple classes.
- If we want to access private members of the class inside some of the member function of another class then we should declare member function as a friend.
- If we want to access private members of the class inside all the member function of another class then we should declare class as a friend.