Day 1

Classfication of languages:

1. Machine level languages
    - Binary language(1, 0)
2. Low level languages
    - Assembly
3. High level languages
    - C, C++, java

Chracteristics of Language

1. It has own syntax
2. It has its own rule(semantics)
3. It contain tokens:
    1. Identifier
    2. Keyword
    3. Constant/literal
    4. Operator
    5. Seperator / punctuators
4. It contains built in features.
5. We use language to develop application(CUI, GUI, Library)
6. If we want to implement business logic then we should use language.

Classfication of high languages:

1. Procedure Oriented Programming Languages
    - PASCAL, FORTRAN, COBOL, C, ALGOL, BASIC etc
    - FOTRAN is first high level pop language.
2. Object Orineted Programming Languages
    - Simula, Smalltalk, C++, Java, Python, C# etc.
    - Simula is first object oriented programming language. It is developed in 1960 by Alan kay.
    - Smalltalk is first pure object oriented programming language which is developed in 1967.
    - More 2000 languages are object oriented.
3. Object based programming languages
    - Ada, Modula-2, Java Script, Visual Basic etc.
    - Ada is first object based programming language.
4. Rule based programming languages
    - LISP, Prolog etc
5. Logic Orineted programming languages
6. Constraint oriented programming languages
7. Functional programming languages
    - Java, Python etc.

C Language Revision

History

- Inventor of C language is Dennis Ritchie
- It is developed in 1969-1972
- It is develped at AT&T Bell Lab USA
- It is developed on DEC-PDP11(Hardware)
- It is developed on Unix(Operating System)

ANSI Standards

- Set of rules is called standard and standard is also called as specification.
- American National Standard Institute(ANSI) is an organization which is responsible for standardization of C/C++ and SQL.
- ANSI is responsible for updating language ie. adding new features, updating existing features, deleting unused features.
- ANSI C standards:

1. Before 1989 : The C Prog Lang Book
2. C89 : 1989
3. C90 : 1990
4. C95 : 1995
5. C99 : 1999
6. C11 : 2011
7. C18 : 2018

C Language Basics

```c
#include<stdio.h>
int main( void )
{
    printf("Hello World!!!");
    return 0;
}
```

- Set of statement is called program.
- An instruction given to the computer is called statement.
- Every instruction is made up of token.
- Token is basic unit of program.
- Tokens in C:

1. Identifiers
    - Name given to variable, array, function, pointer, union structure, enum etc is called identifier.
    - "main" is name of function hence it is considered as identifier.
2. Keyword
    - It is reservered word that we can not use as a identifier.
    - Kewords in C:
        1. The C Prog Language (1st Edition) : 28 keywords

    2. The C Prog Language (2nd Edition) : 27 keywords(entry keyword was removed)
    3. C89 : 5 keywords
    4. C99 : 5 keywords
    5. C11 : 7 Kewords
3. Constant / Literal
    - An entity whose value we can not change is called constant.
    - Types:
        1. Character constant. e.g 'A'
        2. Integer constant
            1. Decimal Constant
            2. Octal Constant
            3. Hexadecimal Constant
        3. Floating Point Constant
            1. Float constant. e.g 3.14f
            2. Double constant. e.g 3.14
        4. String constant. e.g "CDAC"
        5. Enum Constant

```
enum ShapeType
{
    EXIT, LINE, RECT, OVAL  //Enum constant
};
```

4. Operator

- If we want to create expression then we should use operator
- Types:
    1. Unary Operator e.g ++, --, ~,!, sizeof,& etc
    2. Binary Operator
        1. Arithmetic operator e.g +, -, *, /, %
        2. Relational Operator e.g <, >,>=, <=, ==, !=
        3. Logical Operator e.g &&, ||
        4. Bitwise operator e.g &, |, ^, <<, >>
        5. Assignment operator e.g =, Shorthand operators
    3. Ternary OPerator e.g Conditional operator(? :)

5. Punctuator / Seperator

- ; : , space, tab, { }[ ]< > etc

Software Development Kit

- SDK = Language tools + Documentation + Supporting Library + Runtime Env.
- Language tools
    1. Editor
        - Notepad, Edit Plus, gedit, vim, TextEdit, MSVS Code etc
        - It is used to develop/edit source code.

2. Preprocessor
    - CPP(C/C++ preprocessor)
    - Job of preprocessor:
        1. To remove the comments
        2. To expand macros
3. Compiler
    - For Mrcrosoft Visual Studio : cl.exe
    - For Linux : gcc
    - For Intel : icc
    - For Borland : tcc
    - Job of Compiler:
        1. To check syntax
        2. To convert high level source code into low level code(Assembly)
4. Assembler
    - For Borland : TASM
    - For MSVS : MASM
    - For Linux : as
    - Job of Assembler:
        1. To convert low level code into machine code.
5. Linker
    - For Borland : TLINK.exe
    - For MSVS : Link.exe
    - For Linux : ld
    - Job of linker
        1. .obj/.o file contains machine code. This file is also called as almost executable. Linker is responsible for linking .o file to glibc.so.
6. Loader
    - It is operating system API, which is responsible for loading executable file from HDD into RAM.
7. Debugger:
    - For Linux : gdb
    - For Windows : windbg
    - Job of Debugger:
        1. It is used to find the bug.
8. Profiler:
    - For Linux : valgrind
    - Job of profiler:
        1. To debug the memory and detetcting memory leakage.

- Documentation:
    1. For Windows : MSDN
    2. For Linux : man pages
- Supporting Library:
    1. glibc.so
    2. BOOST, QT
- Runtime Environment
    - It is responsible for managing execution of C application.

    - Runtime Environment for C is "C runtime".

Data Type

- It describes 3 things about variable / object

1. Memory : How much memory is required to store the data.
2. Nature : Which type of data memory can store
3. Operation : Which operations are allowed to perform on data stored inside memory.

- Types of data types:
    1. Fundamental Data Types
    2. Derived Data Types
- Fundamental Data Types(5)
    1. void : Not Specified
    2. char : 1 byte
    3. int : 4 bytes
    4. float : 4 bytes
    5. double : 8 bytes
- Derived Data Types(5)
    1. Array
    2. Function
    3. Pointer
    4. Union
    5. Structure

Type Modifiers(4)

```
1. short
2. long
3. signed
4. unsigned
```

Type Qualifiers(2)

```
1. const
2. volatile
```

Constant and variable

- An entity whose value we can not modify is called constant.
- constant is also called as literal.
- e.g 'A', "Pune", 3.14, 0 etc.
- An entity whose value we can modify is called variable.
- Variable is also called as object/instance.

- e.g. int number; Here number is variable.

Comments

- If we want to maintain documentation of source code then we should use comments.
- Types:
    1. //Single line comment
    2. /* Multiline comment. */

Main function

- According to ANSI, main should be entry point function of C/C++.
- Programmer is responsible for defining main function hence it is considered as user defined function.
- Calling/invoking main function is responsibility of operating system. Hence it is also called as Callback function.
- Since main function is responsible to give call to the other functions, it is also called as calling function.
- Signature of main function;

1. void main( );
2. void main(void);
3. int main(void);
4. ☐);
5. ☐);

- Standard Syntax of main function is:

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

Function Declaration and Definition

```
//Function Definition
int main( void )        //Calling Function
{
        void print( void );     //Local Function Declaration

        print( );       //Function Call
        return 0;
}
//Function Definition
void print( void )      //Called Function
{
        printf("Inside print function\n");
}
```

- Implementation of function is called function definition.
- Local definitions are not allowed in C/C++. In other words, we can not define function inside another function.
- If we use function before its definition then it is mandatory to provide its signature to the compiler. It is called function declaration.
- It is possible to declare function locally as well globally.
- Without definition, if we try to access any element then linker generates error.

```
//Function Definition
int main( void )        //Calling Function
{
        //Local Function Declaration
        void print( void );

        //Function Call
        print( );       //Linker error

        return 0;
}
```

- If we try to build and execute project without main function then linker generates error.

Variable Declaration and Definition

- Declaration refers to the place where nature of the variable is stated but no storage is allocated.
- Definition refers to the place where memory is assigned or memory is allocated.

```
int main( void )
{
    int num1;   //Declaration as well as defintion

    int num2 = 20; //Declaration as well as definition

    extern int num3;    //Declaration
    return 0
}
int num3 = 30;  //Declaration as well as definition
```

Variable Initialization and Assignment

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

- Initialization is the process of storing value inside variable during its declaration.
- We can initialize variable only once.

```
int num1 = 10; //Initialization
num1 = 20;  //Assignment
num1 = 30;  //Assignment
```

- Assignment is process of storing value inside variable after its declaration.
- we can assign value to the variable multiple times.

Day 2

L-Value(Locator Value)

- Non constant(editable/modifiable) memory location which is available at left hand side of assignment operator is called locator value(L-Value).
- Consider Following code:

```
2 + 3 = 5;  //Error - L-Value Required
```

- Consider Following code:

```
5 = 2 + 3; //Error - L-Value Required
```

- Consider Following code:

```
const int number = 10;
number = number + 5; //Error - L-Value Required
```

- Consider Following code:

```
int number = 10;
number = number + 5; //OK - number is L-Value
```

R-Value(Reference Value)

- A constant, variable or expression which is used at right hand side of assignment operator is called R-Value.

```
int num1 = 10;  //10 - R-Value
```

```
int num1 = 10;  //10 - R-Value
int num2 = num1;//num1 - R-Value
```

```
int num1 = 10;  //10 - R-Value
int num2 = num1;//num1 - R-Value
int num3 = num1 + num2; //(num1 + num2) - R Value
```

5 keywords introduced in C89

```
1. const
2. volatile
3. void
4. enum
5. signed
```

Constant in C

```
int num1 = 10;
num1 = num1 + 5;    //15
```

- Once initialized, if we dont want to modify value/state of the variable/object then we should use const keyword.
- const keyword is introduced by ANSI in C89.

```
const int num1 = 10;
num1 = num1 + 5;        //Not OK
```

- Constant variable is also called as read only variable.
- In C, it is optional to initialize constant variable.
- In C, we can declare variable constant but we can not declare function constant.

Pointer in C

- Named memory location is called variable.
- & is a unary operator which is used to get address of variable/object.
- If we want to store address then we need to declare pointer in a program.
- A pointer is a variable which is used to store address of another variable.
- Size of any type of pointer on 16-bit compiler is 2 bytes, on 32-bit compiler 4 bytes and on 64 bit compiler 8 bytes.
- Uninitialized pointer is called wild pointer

```
int main( void )
{
        int *ptrNum1;   //Wild POinter
        return 0;
}
```

- NULL is a macro whose value is 0 address

```
#define NULL ((void*) 0)
```

- If pointer contains NULL value then such pointer is called NULL pointer.

```
int main( void )
{
        int *ptrNum1 = NULL;//ptr1Num1 : NULL Pointer
        return 0;
}
```

- Pointer initialization:

```
int main( void )
{
        int num1 = 10;  //Initialization
        int *ptrNum1 = &num1;   //Initialization
        return 0;
}
```

- Pointer assignment:

```
int main( void )
{
        int *ptrNum1 = NULL; //Initialization
        int num1 = 10;  //Initialization
        ptrNum1 = &num1;        //Assignment
        return 0;
}
```

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

Constant and pointer combination

```
int *ptr
```

- In above statement, ptr is non constant pointer variable which can store address of non constant integer variable.
- Consider following example:

```
int main( void )
{
        int *ptr = NULL;
        int num1 = 10;
        ptr = &num1;
        *ptr = 50;      //Dereferencing
        printf("Num1    :       %d\n", *ptr);//Dereferencing

        int num2 = 20;
        ptr = &num2;
        *ptr = 60;      //Dereferencing
        printf("Num2    :       %d\n", *ptr);//Dereferencing
        return 0;
}
```

```
const int *ptr
```

- In above statement, ptr is non constant pointer variable which can strore address on constant integer variable.

```
int main( void )
{
        const int *ptr = NULL;

        const int num1 = 10;
        ptr = &num1;    //OK
        //*ptr = 50;    //Not OK
        printf("Num1    :       %d\n", *ptr);//Dereferencing : OK

        const int num2 = 20;
        ptr = &num2;    //OK
        //*ptr = 60;    //Not OK
        printf("Num2    :       %d\n", *ptr);//Dereferencing : OK
        return 0;
}
```

```
int const *ptr
```

- Above is 100% same as "const int *ptr"

```
const int const *ptr
```

- Above state is same as "const int *ptr" or "int const *ptr"
- For above compiler will generate warning: "Duplicate const qualifier".

```
int *const ptr
```

- In above statement, ptr is constant pointer variable which can store address of non constant integer variable.

```
int main( void )
{
        int num1 = 10;
        int *const ptr = &num1; //OK
        *ptr = 50;      //Dereferencing : OK
        printf("Num1    :       %d\n", *ptr);//Dereferencing : OK

        int num2 = 20;
        //ptr = &num2;  //Not OK
        return 0;
}
```

```
int *ptr const
```

- Above syntax is invalid.

```
const int *const ptr
```

- In above statement, ptr is constant pointer variable which can store address of constant integer variable.

```
int main( void )
{
        const int num1 = 10;

        const int *const ptr = &num1;   //OK
        //*ptr = 50;    //Not OK
        printf("Num1    :       %d\n", *ptr);//Dereferencing : OK

        const int num2 = 20;
```

```
        //ptr = &num2;  //Not OK
        return 0;
}
```

```
int const *const ptr
```

- This statement is same as "const int *const ptr"

Structure

- If we want to group related data elements together then we should use structure. Related data elements may be of same type or different type.
- Structure is derived data type.
- if we want to define structure then we should use struct keyword.

```
struct Employee
{
    char name[ 30 ];
    int empid;
    float salary;
};
```

- We can declare structure inside function. It is called local structure.
- We can not use object and pointer of local structure outside function.
- We can define/declare function inside structure.
- If we want to store value inside structure then we must create its object.

```
struct Employee emp;
```

- If we create object structure then all the variables declared inside structure get space inside it.
- If type allows us to initialize its element using initializer list then it is called aggregate type and object is called aggregate object.

```
struct Employee emp = {"Abc",33,45000.50f};
```

- Following types are aggregate types:
    1. Array
    2. Structure
    3. Union
- Using object, if we want to access members of the structure then we should use dot/member selection operator.

```
printf("Name    :       %s\n", emp.name );
printf("Empid   :       %d\n", emp.empid);
printf("Salary  :       %f\n",emp.salary);
```

- Using pointer, if we want to access members of structure then we should use arrow/dereferencing operator.

```
printf("Name    :       %s\n", ptr->name );
printf("Empid   :       %d\n", ptr->empid);
printf("Salary  :       %f\n",ptr->salary);
```

- Parameter and argument

```
//a,b -> Function parameter / parameter
void sum( int a, int b )
{
    int c = a + b;
    printf("Result  :   %d\n",c);
}
```

```
sum( 10,20 );   //Function Call
//10,20 -> Function argument / argument
```

```
int x = 10, y = 20;
sum( x, y );    //Function Call
//x,y -> Function argument / argument
```

- In C language, we can pass argument to the function using 2 ways:
    1. By Value
    2. By Address/Reference
- If we declare structure outside function then it is called global structure. We can create object and pointer of global structure anywhere in the program.
- Procedure oriented programming is a kind of programming in which we try to solve real world problems using structure and function.
- Object oriented programming is a kind of programming in which we try to solve real world problems using class and object.
- If we want to control visibility of members of structure/class then we should use access specifier.
- Access specifiers in C++
    1. private(-)
    2. protected(#)
    3. public(+)

- In C++, structure members are by default considered as public.

Day 3

- Inventor of C++ is Bjarne Stroustrup.
- C++ is derived from C and simula.
- Its initial name was "C With Classes".
- At is developed in "AT&T Bell Lab" in 1979.
- It is developed on Unix Operating System.
- Standardizing C++ is a job of ANSI.
- In 1983 ANSI renamed "C With Classes" to C++.
- C++ is objet orieted programming language.
- In C++ we can develop code using Procedure as well as object orieneted fashion. Hence it is also called Hybrid programming language.

C++ Standards:

1. Before 1998 : Annoted C++ Reference Manual
2. 1998 : C++98
3. 2003 : C++03
4. 2011 : C++11
5. 2014 : C++14
6. 2017 : C++17
7. 2020 : C++20

- cfront is a translator developed by Bjarne Stroustrup which was used to convert C++ source code into C source code.

Data Types

- Fundamental Data types(7)
    1. void : Not Specified
    2. bool : 1 byte
    3. char : 1 byte[ASCII]
    4. wchar_t : 2 bytes[Unicode]
    5. int : 4 bytes
    6. float : 4 bytes
    7. double : 8 bytes
- Derived Data types(4)
    1. Array
    2. Function
    3. Pointer
    4. Reference
- User Defined Data Types(3)
    1. Union
    2. Structure
    3. Class

Object Oriented Programming Structure

- OOPS is not a syntax. It is a process / programming methodology which is used to solve real world problems.
- It is invented by Dr. Alan Kay. He is inventor of Simula too.
- Unified Modelling Language(UML) is invented by Grady Booch. If we want to do OOA and OOD then we can use UML.
- According Grady Booch there are 4 main/major and 3 minor elements/parts/pillars of OOPS

4 major pillars of oops

1. Abstraction
2. Encapsulation
3. Modularity
4. Hierarchy Here, word major means, language without any one of the above feature will not be Object orieneted.

3 minor pillars of oops

1. Typing
2. Concurrency
3. Persistence

- Here word minor means, above features are useful but not essential to classify language object oriented.

Data Member

- Variable declared inside class scope is called data member.

```
int num1;   //Golbal Variable
class Test
{
    int num2;   //Data Member
    int num3;   //Data Member
};
int main( void )
{
    int num4;   //Local Variable
    return 0;
}
```

- Data member is also called as field, attribute, property etc.

Member Function

- A function implemented inside class scope is called member function.

```
class Test
{
public:
    void print( )   //Member Function
    {   }
};
//For class Test,"main" is a non member fn.
int main( void )    //Global Function
{
    Test t;
    t.print( );
    return 0;
}
```

- Member function is also called as method, operation, behavior or message.

Class and Object

- Class is collection of data member and member function.
- Class can contain:
    1. Nested Type
        - enum
        - structure
        - union
        - class
    2. Data Member
    3. Member Function
        - constructor
        - destructor
        - copy constructor
        - user defined function
- Variable/ instance of a class is called object.
- Process of creating object in C:

```
struct Employee emp;
```

- Process of creating object in C++:

```
Employee emp;
```

- Process of creating object in Java:

```
Employee emp = new Employee( );
```

- Process of creating object from a class is called instantiation.

```
int main( void )
{
        Employee emp;   //Instantiation
        return 0;
}
```

- In above code, class Employee is instantiated and name of the instance is emp.
- During instantiation use of class keyword is optional

```
int main( void )
{
        class Employee emp1;   //OK
    Employee emp2;   //OK
        return 0;
}
```

Naming / Coding Convention

1. Hungerian Notation(For C/C++)
2. Camel Case Convention(Java and .NET)
3. Pascal Case Convention(Java and .NET)

Camel Case Convention

- Consider following example
    1. main()
    2. parseInt()
    3. showInputDialog
    4. addNumberOfDays(int days)
- In this case, except word, first Character of each word must be in upper case.
- We should use this convention for:
    1. Data member
    2. Member function
    3. Function Parameter
    4. Local and global variable

Pascal Case Convention

- Consider following example
    1. System
    2. StringBuilder
    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 Name:
        1. Union Name
        2. Structure Name
        3. Class Name
        4. Enum Name
    2. File Name

Convention For macro and constant:

- Name of constant, enum constant and macro should be in upper case.

```
#define NULL 0
#define EOF  -1
#define SIZE 5
```

```
const float PI = 3.142
```

```
enum ShapeType
{
    EXIT, LINE, RECT, OVAL
};
```

Naming Convention for namespace

- Name of the namespace should be in lowercase.
- Consider example

```
namespace collection
{
    class Stack
    {   };
}
```

Naming Convention for global function

- Consider example:
    1. void print(void);
    2. void print_record(void);

Message Passing

- Process of calling member function on object is called message passing

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

- In above code, acceptRecord() and printRecord() is called on object emp.

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

- Syntax to define member function globally: ReturnType ClassName::FunctionName( ){ }

Difference between <abc.h> and "abc.h":

- "/usr/include" directory is called standard directory for header files.
- It contains all the standard header files of C/C++
- e.g
    - stdio.h
    - string.h
    - stdlib.h
    - iostream
- If we include header file in angular bracket (e.g #include<abc.h>) then preprocessor try to locate and load header file from standard directory only(/usr/include).
- If we include header file in douvle quotes (e.g #include"abc.h") then preprocessor try to locate and load header file first from current project directory if not found then it try to locate and load from standard directory.

Header Guard

- If we want to expand contents of header file only once then we should use header guard:
- Syntax:

```
#ifndef HEADER_FILE_NAME_H_
#define HEADER_FILE_NAME_H_
    //TODO : Type declaration here
#endif
```

Class and Object

- Member function do not get space inside object.
- If we create object of the class then only data members get space inside object. Hence size of object is depends on size of all the data members declared inside class.
- Data members get space once per object according to the order of data member declaration.
- Structure of the object is depends on data members declared inside class.
- Member function do not get space per object rather it gets space on code segment and all the objects of same class share single copy of it.
- Member function's of the class defines behavior of the object.
- Class:
    1. It is a collection of data member and member function.
    2. Structure and behavior of an object is depends on class hence class is considered as a template/model/blueprint for an object.
    3. Class represents set/group of such objects which is having common structure and common behavior.
    4. Class is a imagnary/logical entity.
    5. Class represents encpasulation.
    6. Examplex:
        - Mobile Phone
        - Laptop
        - Car
- Object
    1. Object is a variable/instance of a class.
    2. An entity, which get space inside memory is called object.
    3. An entity which has state, behavior and identity is also called as object.
    4. It is physical entity.
    5. With the help of instantiation we achive abstraction.
    6. Example:
        - Nokia 1100
        - MacBook Pro
        - Maruti 800

Empty class

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

```
class Test
{   };
```

- Size of object depends on size of all the data memebers declared inside class.
- According to above definition size of object of empty class should be zero.
- To differentiate object from class, object must get space inside memory.
- According to Bjarne Stroustrup, size of object of empty class should be non zero.
- But due to compilers optimization, object of empty class get one byte space inside memory.

Characteristics of object

```
1. State
    * Value stored inside object is called state of the object.
    * Value of data member represent state of the object.
2. Behavior
    * Set of operation that we perform on object is called behavior of an
object.
    * Member function of class represent behavior of the object.
3. Identity
    * Value of any data member, which is used to identify object uniquly
is called its identity.
    * If state of object is same the its address can be considered as its
identity.
```

"this" pointer

1. First we define the class.
2. Then we declare data members inside class.
3. We instantiate class.
4. To process state of the object we should call member function on object. Hence we must define member function inside class.
5. If we call member function on object then compiler implicitly pass address of that object as a argument to the function implicitly.

```
Employee emp;
emp.printRecord( );//emp.printRecord(&emp);
```

6. To store address of object compiler implitly declare one pointer as a parameter inside member function. Such parameter is called this pointer.
7. this is a keyword. "this" pointer is a constant pointer.
8. General type of this pointer is:

```
ClassName *const this;
```

- "this" pointer is implicit pointer, which is available in every non static member function of the class which is used to store address of current object or calling object.
- Following functions do not get this pointer:
    1. Global Function
    2. Static Member function
    3. Friend Function.
- this pointer is considered as first parameter of member function.

- Using this pointer, data member and member function can communicate with each other hence "this" pointer is considered as a link / connection between data member and member function.
- Use of this keyword, to access members is optional.

Day 4

Constructor

- It is a member function of a class which is used to initialize 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. In the life time of the object is gets called only once.

- Constructor gets called once per object and according to order of its declaration.
- We can not call constructor on object, pointer or reference explicitly. It is designed to call implicitly.

```
int main( void )
{
        Point pt1;  //Point::Point()
        pt1.Point( );   //Not OK

        Point *ptr = &pt1;  //Ok
        ptr->Point( );  //Not OK

        Point &pt2 = pt1;   //OK
        pt2.Point( );   //Not OK
        return 0;
}
```

- Compiler do not call constructor on pointer or reference.

```
Complex *ptr;
Complex &c2 = c1;
```

- We can use any access specifier on constructor.
- If ctor is public then we can create object of the class inside member function as well as non member function but if constructor is private then we can create object of the class inside member function only.
- We can not declare constructor static, constant, volatile or virtual. We can declare constructor only inline.
- We can return value from constructor but constructor can contain return statement. It is used to return controll to the calling function.

Types of constructor

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

Parameterless constructor

- A constructor, which do not take any parameter is called Parameterless constructor.
- It is also called zero argument constructor or user defined default constructor.

```
//Point *const this;
Point( void )
{
    this->xPos = 0;
    this->yPos = 0;
}
```

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

```
Point pt1;  //Point::Point( )
Point pt2;  //Point::Point( )
```

Parameterized constructor

- If constructor take parameter then it is called parameterized constructor.

```
//Point *const this;
Point( int xPos, int yPos )
{
    this->xPos = xPos;
    this->yPos = yPos;
}
```

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

```
Point pt1(10,20);   //Point::Point(int,int)
Point pt2;  //Point::Point( )
```

- Copy constructor is a single parameter constructor hence it is considered as parameterized constructor.

Default constructor

- If we do not define constructor inside class then compiler generates default constructor for the class.
- Compiler do not provide default parameterized constructor. Compiler generated default constructor is parameterless.

```
class Point
{
private:
        int xPos;
        int yPos;
};
int main( void )
{
    Point pt1;          //OK
        Point pt2(10,20);   //Not OK
    return 0;
}
```

- If we want to create object by passing argument then its programmers responsibility to write parameterized constructor inside class.
- Default constructor do not initialize data members declared by programmer. It is used to initialize data members declared by compiler(e.g v-ptr, vbptr).
- If compiler do not declare any data member implicitly then it doesnt generate default constructor.
- We can write multiple constructor's inside class. It is called constructor overloading.
- In C++98 and C++ 03, we can not call constructor from another constructor. In other words C++ do not support constructor chaining.

Constructor delegation(C++ 11)

- In C++ 11 we can call constructor from another constructor. It is called constructor delegation. Its main purpose is to reuse body of existing constructor.

```
class Point
{
private:
        int xPos;
        int yPos;
public:
    //Ctor delegation
        Point( void ) : Point( 10, 20 )
        {       }
        Point( int xPos, int yPos)
        {
                this->xPos = xPos;
                this->yPos = yPos;
        }
}
```

> Setting in eclipse : Right click on project -> properties -> C++ build -> Setting -> C++ Compiler -> Miscellaneous -> other flags -> (Give space and paste)-std=c++11

```
class Point
{
private:
        int xPos;
        int yPos;
public:
        Point( void )
        {
                this->xPos = 10;
                this->yPos = 20;
        }
        Point( int value)
        {
                this->xPos = value;
                this->yPos = value;
        }
        Point( int xPos, int yPos)
        {
                this->xPos = xPos;
                this->yPos = yPos;
        }
        void printRecord( void )
        {
                printf("X Position      :       %d\n", this->xPos);
                printf("Y Position      :       %d\n", this->yPos);
        }
};
```

```
int main( void )
{
        //Point pt1;            //Point::Point( void )
        //Point pt2( 10 );      //Point::Point( int )
        //Point pt3(50,60);     //Point::Point( int, int )
        //Point pt4();  //Function Declaration
        //Point pt5 = 30;       //Point pt5( 30 ); //Point::Point( int )
        //Point(30,40); //Point::Point( int, int )
        //Point(30,40).printRecord();
        //Point pt6 = 50, 60;   //Point pt6( 50 ), 60   //Error
        //Point pt6 = ( 50, 60 );       //Point pt6 = ( 60 )
//Point::Point( int )

//      Point pt7;      //Point::Point( void )
//      Point *ptr = &pt7;      //Compiler do not call ctor on ptr

//      Point pt8;      //Point::Point( void )
//      Point &pt9 = pt8;//Compiler do not call ctor on pt9
```

```
        Point pt10;     //Point::Point( void )
        Point pt11 = pt10;      //On pt11, copy constructor will call
        return 0;
}
```

Aggregate Class

- By default class is not considered as aggregate class.
- If we define class using following rules then class can be considered as aggregate class.

1. Members of class must be public.
2. Class must not contain constructor
3. Class must not contain virtual function
4. Class should not extend structure/class

```
//Aggregate class / Plain Old Data Structure
class Point
{
public:
        int xPos;
        int yPos;
public:
        void printRecord( void )
        {
                printf("X Position      :       %d\n", this->xPos);
                printf("Y Position      :       %d\n", this->yPos);
        }
};
int main( void )
{
        Point pt1 = { 10, 20 }; //OK
        pt1.printRecord();
        return 0;
}
```

Namespace

- If we want to access value of global variable then we should use scope resolution operator(:😃

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

        {
                int num1 = 30;
```

```
                printf("Num1 : %d\n", ::num1);//10
                printf("Num1 : %d\n", num1);//30
        }
        return 0;
}
```

- In same scope we can give same name to the multiple variable/function etc.

```
int num1 = 10;
int num1 = 20; //error: redefinition of 'num1'
int main( void )
{
        printf("Num1 : %d\n",num1);
        return 0;
}
```

- Namespace in C++ language feature which is used:

1. To avoid name clashing/collision/ambiguity.
2. To group functionally equivalent/related types together.

- We can not instantiate namespace. It is designed to avoid name ambiguity and grouping related types.
- If we want to define namespace then we should use namespace keyword.
- Syntax:

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

- namespaces can only be defined in global or namespace scope. In other words, we can not define namespace inside function/class.
- If we want to access members of namespace then we should use namespace name and scope resolution operator.

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

- We can not define main function inside namespace.
- Namespace can contain:
    1. Variable
    2. Function
    3. Types[structure/union/class]
    4. Enum
    5. Nested Namespace

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

        printf("Num2 : %d\n",nb::num2);//20
        printf("Num3 : %d\n",nb::num3);//40
        return 0;
}
```

- If name of the namespaces are diffrent then we can give same/diffrent name to the members of namespace.

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

        printf("Num2 : %d\n",na::num2);//20
        //printf("Num3 : %d\n",na::num3);//40
```

```
        return 0;
}
```

- If name of the namespaces are same then name of members must be different.
- std is standard namespace on C++.
- We can define namespace inside another namespace. It is called nested namespace.

```
int num1 = 10;
namespace na
{
        int num2 = 20;
        namespace nb
        {
                int num3 = 30;
        }
}
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;
}
```

- If we define member without namespace then it is considered as member of global namespace.
- If we want to access members of namespace frequently then we should use using directive.

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

```
namespace na
{
        int num1 = 10;
}
int main( void )
{
        int num1 = 20;
        using namespace na;
```

```
        printf("Num1 :  %d\n",num1); //20
        printf("Num1 :  %d\n",na::num1);//10
        return 0;
}
```

Day 5

Console Input and Output operation

- Console Input -> Keyborad
- Console Output -> Monitor
- Console = Keyboard + Monitor
- is standard header file of C++. Consider declaration of iostream:

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

Character Output(cout)

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

- cout is external object of ostream class.
- cout is member of std namespace and std namespace is declared in iostream header file.
- cout represents monitor.
- An insertion operator(<<) is designed to use with cout.

Escape Sequence

- It is a character which is used to format the output.
- Following are the Escape Sequences
    1. '\n'
    2. '\t'
    3. '\b'
    4. '\r'
    5. '\a'
    6. '\v'

Manipulator

- It is a function which is used to format the output.
- Following are the manipulators in C++
    1. endl = '\n' + flush()
    2. setw
    3. fixed
    4. scientific
    5. setprecision
    6. hex
    7. dex
    8. oct
    9. left
    10. right
    11. center
- To use manipulators it is necessary to include header file.
- Method 1:

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

- Method 2:

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

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

Character Input(cin)

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

- cin is an external object of istream class.
- cin is a member of std namespace and std namespace is declared in header file.
- Extraction operator(>>) is designed to use with cin object.
- cin represents keyboard.

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

std::string class

- is standard header file of C++ which contains std namespace and std contains basic_string class.

```
typedef basic_string<char> string;
```

- size of string object is 24 bytes.

Constructor's member initializer list

- If we want to initialize data members according to users requirement then we should use constructor body.

```
class Test
{
private:
        int num1;
        int num2;
        int num3;
public:
        Test( void )
        {
                this->num3 = num2;
        this->num1 = 10;
        this->num2 = num1;
        }
};
```

- If we want to initialize data member according to order of data member declaration then we should use constructors member initializer list.

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

- Except array we can initialize any member inside constructors member initializer list.
- If we provide constructor member initializer list as well Constructor body then compiler first execute constructor member initializer list.
- In case of modular approach, constructors member initializer list must appear in definition part(.cpp).

Constant in C++

- const is type qualifier.
- It is introduced in C89.
- If we dont want modify value of the variable then we should use const keyword.
- constant variable is also called as read only variable.

```
const int num1;   //OK : In C
const int num2;   //Not OK : In C++
const int num3 = 10;   //OK : In C++
```

- In C++, Initializing constant variable is mandatory.

Constant Data Member

- Once initialized, if we dont want to modify state of the data member inside any member function of the class including constructor body then we should declare data member constant.
- If we declare data member constant then it is mandatory to initialize it using constructors member initializer list.

```
class Test
{
private:
        const int num1;
public:
```

```
        Test( void ) : num1( 10 ) //OK
        {
                //this->num1 = 10; //Not OK
        }
};
```

Constant member function

- We can not declare global function constant but we can declare member function constant.
- If we dont want to modify state of current object inside member function then we should declare member function constant.
- Non constant member function get this pointer like:

```
ClassName *const this;
```

- Constant member function get this pointer like:

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

- Volatile member function get this pointer like:

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

- If function do not get this pointer / if we can not call function on object then we can not declare it constant.
- We can not delclare following function constant:
    1. Global Function
    2. Static Member Function
    3. Constructor
    4. Destructor
- Since main function is a global function, we can not delcare it constant.
- We should declare read only function constant. e.g getter function, printRecord function etc.
- In constant member function, if we want to modify state of non constant data member then we should use mutable keyword.

Constant object

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

typedef

- It is C language feature which is used to create alias for exsiting data type.
- Using typedef, we can not define new data type rather we can give short name / meaningful name to the exisiting data type.
- e.g
    1. typedef unsigned short wchar_t;
    2. typdef unsgined int size_t;
    3. typedef basic_istream istream;
    4. typedef basic_ostream ostream;
    5. typedef basic_string string;

Reference

- Reference is derived data type.
- It alias or another name given to the exisiting memory location / object.

```
int num1 = 10;
int &num2 = num1;
```

- In above code num1 is referent variable and num2 is reference variable.
- Using typedef we can create alias for class whereas using reference we can create alias for object.
- Once reference is initialized, we can not change its referent.

```
int main( void )
{
        int num1 = 10;
        int num2 = 20;
        int &num3 = num1;

        num3 = num2;
        ++ num2;

        cout<<"Num1     :       "<<num1<<endl; //20
        cout<<"Num2     :       "<<num2<<endl; //21
        cout<<"Num3     :       "<<num3<<endl; //20
        return 0;
}
```

- It is mandatory to initialize reference.

```
int main( void )
{
        int &num2;      //Not OK
        return 0;
}
```

- We can not create reference to constant value.

```
int main( void )
{
        int &num2 = 10; //Not OK
        return 0;
}
```

- We can create reference to object only.
- Reference is internally considered as constant pointer hence referent of reference must be variable/object.

```
int main( void )
{
        int num1 = 10;
        int &num2 = num1;
        //int *const num2 = &num1;
        cout<<"Num2     :       "<<num2<<endl;
        //cout<<"Num2   :       "<<*num2<<endl;
        return 0;
}
```

Day 6

- Reference is automatically dereferenced constant pointer variable.
- If we want to reduce complexity of pointer then we should use reference.
- Reference to array:

```
int main( void )
{
        int arr1[ 3 ] = { 10, 20, 30 };
        int (&arr2)[ 3 ] = arr1;
        for( int index = 0; index < 3; ++ index )
                cout<<arr2[ index ]<<endl;
        return 0;
}
```

- In C++, we can pass argument to the function using 3 ways:
    1. By Value
    2. By Address
    3. By Reference
- We should not retrun non static local variable from function by address.

```
int* getNumber( void )
{
        int number = 10;
        return &number;
}
int main( void )
{
        int *ptr = ::getNumber();
        cout<<"Number   :       "<<*ptr<<endl;
        return 0;
}
//Output : Garbage Value
```

- If we want to return local variable from function by address then we should use static keyword.

```
int* getNumber( void )
{
        static int number = 10;
        return &number;
}
int main( void )
{
        int *ptr = ::getNumber();
        cout<<"Number   :       "<<*ptr<<endl;
        return 0;
}
//Output : 10
```

- We should not return local variable from function by reference.

```
int& getNumber( void )
{
        int num1 = 10;
        return num1;
}
int main( void )
{
        int &num2 = ::getNumber();
    //num2 will become Dangling reference
        cout<<"Number   :       "<<num2<<endl;
        return 0;
}
//Output : Garbage Value
```

- If we want to return local variable from function by reference then we should declare local variable static

```
int& getNumber( void )
{
        static int num1 = 10;
        return num1;
}
int main( void )
{
        int &num2 = ::getNumber();
        cout<<"Number   :       "<<num2<<endl;
        return 0;
}
//Output : 10
```

Exception Handling

- Following are the operating system resources that we can use in application development:
    1. Memory
    2. File
    3. Thread
    4. Socket
    5. Network connection
    6. IO Devices etc.
- Since OS resources are limited, we should use it carefully.
- If we make syntactical mistake in a program then compiler generates error.
- Without definition, if we try to access any member then linker generates error.
- Logical error / syntacticaly valid but logicaly invalid statements represents bug.

```
int number = 100;
if( number = 0 )    //Bug
    cout<<"True :   "<<number<<endl;
else
    cout<<"False :   "<<number<<endl;
```

```
int number = 100;
if( number == 100 )
    cout<<"True :   "<<number<<endl;
else;   //Bug
    cout<<"False :   "<<number<<endl;
```

```
int count;
for(count=1;count<=5; count ++); //Bug
cout<<count<<endl;
```

```
int arr[2][3] = { {1,2,3},{4,5,6}};
for( int i = 0; i < 2;  i ++ )
{
    for( int j = 0; j < 3; i ++ )//Bug
    {
        cout<<arr[ i ][ j ]<<"  ";
    }
    cout<<endl;
}
```

- If we give wrong input to the application then it generates runtime error/exception.
- Exception is an object, which is used to send notification to the end user of the system if any exceptional situation occurs in the program.
- If we want to manage OS resources carefully then we should use exception handling mechanism.
- If we want to handle exception then we should use 3 keywords:
    1. try
    2. catch
    3. throw

try:

- try is keyword in C++.
- If we want to inspect exception then we should put statements inside try block/handler.
- try block must have at least one catch block/handler

throw:

- throw is keyword in C++.
- If we want to generate exception explicitly then we should use throw keyword.
- "throw statement" is a jump statement.

catch:

- If we want to handle exception then we should use catch block/handler.
- Single try block may have multiple catch block.
- Catch block can handle exception thrown from try block only.
- With the help of function, we can throw exception from outside try block.
- For thrown exception, if we do not provide matching catch block then C++ runtime gives call the std::terminate function which implicitly give call the std::abort function.
- A catch block, which can handle any type of exception is called generic catch block / catch-all handler.

```
try
{   }
catch(...)  //Generic catch block
{   }
```

- Generic catch block must appear after all specific catch block.

Need of exception Handling:

1. To avoid resource leakage.
2. To handle all the runtime errors(exeption) centrally.

- In C++, we can create object without name. It is called annonymous object.
- If we want to use any object only once then we should create annonymous object.

Exception Specification List

```
int calculate( int num1, int num2 )throw( ArithmeticException )
{
        if( num2 == 0 )
                throw ArithmeticException("Divide by zero exception");
        return num1 / num2;
}
```

- If an function fails to perform operation then it can throw exception. To maintain documentation of exception thrown by the function we should use exception specification list. To define exception specification list, we should use throw keyword.
- If exception specification list do not contain type of thrown exception then during failure it doesnt execute catch block rather C++ runtime give call to std::unexpected function which implicitly gives call to the std::terminate function.
- We can write try catch block inside another try block as well as catch block. It is called nested try catch block.
- Outer catch block can handle excpetion's thrown from inner try block.
- Inner catch block, can not handle exception thrown from outer try block.
- If information, that is required to handle exception is incomplete inside inner catch block then we can rethrow that exception to the outer catch block.

```
class ArithmeticException
{
private:
        string message;
public:
        ArithmeticException( string message ) : message( message )
        {       }
        void printStackTrace( void )const
        {
                cout<<this->message<<endl;
        }
};
int main( void )
{
        try
        {
```

```
                try
                {
                        throw ArithmeticException("/ by zero");
                }
                catch( ArithmeticException  &ex)
                {
                        cout<<"Inside inner catch"<<endl;
                        throw;  //throw ex;
                }
        }
        catch( ArithmeticException &ex)
        {
                cout<<"Inside outer catch"<<endl;
        }
        catch(...)
        {
                cout<<"Inside generic catch block"<<endl;
        }
        return 0;
}
```

Stack Unwinding

- During execution of function if any exception occurs then process of destroying FAR and returning control back to the calling function is called stack unwinding.
- During stack unwinding, destructor gets called on local objects(not on dynamic objects).
- We can perform following operations on any object:

1. Inspector function:
    - A member function of class, which is used to read state of the object is called inspector function.
    - It is also called selector function of getter function.
    - e.g getReal() and getImag()
2. Mutator function:
    - A member function of a class, which is used to modify state of the object is called mutator function.
    - It is also called as modifier function or setter function
    - e.g setReal() and setImag()
3. Facilitator function:
    - Member function of a class which allows us to perform operations on Console/file/database is called faciliator function.
    - e.g acceptRecord() and printRecord()
4. Constructor:
    - It is a member function of a class which is used to initialize object.
5. Destructor:
    - It is a member function of a class which is used to deinitialize the object.

Day 8

Default Argument:

- In C++, We can assign default values to the parameters of function. Such default value is called default argument and parameter is called optional parameter.

```
void sum( int num1, int num2, int num3 = 0, int num4 = 0 )
{
        int result = num1 + num2 + num3 + num4;
        cout<<"Result   :       "<<result<<endl;
}
```

- In above code num3 and num4 are optional parameters and 0 is default argument.
- Using default argument, we can reduce, developers efforts.
- Above function will work for following function calls:

```
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.
- Including main function, we can assign default argument to parameters of any global function as well member function.
- In case of modular approach, default argument must appear in declaration part.

Dynamic Memory Management

- If we want to manage memory dynamically in C then we should use functions declared in header file.
- If we want to allocate memory dynamically then we should use following functions:

1. void* malloc(size_t size);
2. void* calloc(size_t size, size_t count)
3. void* realloc(void *ptr, size_t size);

- If we want to deallocate memory dynamically then we should use following function:

4. void free(void *ptr);

- In C++, If we want to allocate memory dynamically then we should use new operator and to deallocate that memory we should use delete operator.
- Everything on heap section is anonymous.
- Memory allocation and deallocation for single integer variable:

```
int main( void )
{
        int *ptr = new int;
```

```
        //int *ptr = ( int* )::operator new( sizeof( int ) * 1 );

        *ptr = 125;     //Dereferencing
        cout<<"Value    :       "<<*ptr<<endl; //Dereferencing

        delete ptr;
        //::operator delete( ptr );

        ptr = NULL;
        return 0;
}
```

- What is the difference in following statement?
    1. int *ptr = new int;
    2. int *ptr = new int();
    3. int *ptr = new int(3);
- If pointer contains, address of deallocated memory then such pointer is called dangling pointer.
- When we allocate space in memory, and if we loose pointer to reach to that memory then such wastage of memory is called memory leakage.
- Memory allocation and deallocation for single dimensional array:

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

        ptr[ 0 ] = 10;
        ptr[ 1 ] = 20;
        ptr[ 2 ] = 30;
        for( int index = 0; index < 3; ++ index )
                cout<<ptr[ index ]<<endl;

        delete[] ptr;
        //::operator delete( ptr );
        ptr = NULL;
        return 0;
}
```

- If malloc/calloc/realloc function fails to allocate memory then it returns NULL.
- If new operator fails to allocate memory then it throws bad_alloc exception.

```
int main( void )
{
        int count = 1000000000000;
        int *ptr = new int[ count ];
    //TODO
        delete[] ptr;
        return 0;
```

```
}
//Output : std::bad_alloc exceptionz
```

- northrow object is used to instruct to new operator to return NULL during failure.

```
struct nothrow_t {};
extern const nothrow_t nothrow;
```

- Consider following code:

```
int main( void )
{
        int count = 1000000000000;
        int *ptr = new ( nothrow ) int[ count ];
        if( ptr == NULL )
                cout<<"NULL"<<endl;
        else
                cout<<ptr<<endl;
        delete[] ptr;
        return 0;
}
//Output : NULL
```

- If we create dynamic object using malloc then constructor do not call. But if we create dynamic object using new operator then constructor gets called.
- Memory allocation and deallocation for multidimensional array.

```
int main( void )
{
        int **ptr = new int*[ 3 ];
        for( int index = 0; index < 3; ++ index )
                ptr[ index ] = new int[ 4 ];

        for( int index = 0; index < 3; ++ index )
                delete[] ptr[ index ];
        delete[] ptr;
        ptr = NULL;
        return 0;
}
```

Destructor:

- It is a member function of a class which is used to release the resources.
- Due to following reasons, it is considered as special function of the class
    1. Its name is same as class name and always preceds with tild operator(~)

    2. It doesnt have return type or doesn't take parameter.
    3. It is designed to call implicitly.
- We can declare destructor as a inline and virtual only.
- Destructor calling sequence is exactly opposite of constructor calling sequence.
- We can not call constructor on object, pointer or reference explicitly. It is designed to call implicitly.
- Destructor is designed to call implicitly but we can call it explicitly.
- If we do not define destructor inside class then compiler generates default destructor for the class.
- Default destructor do not deallocate resources allocated by the programmer. If we want to deallocate it then we should define destructor inside class.