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.

Day 9

Object Copy Semantics:

```
1. Shallow Copy
2. Deep Copy
3. Lazy Copy
4. Defensive Copy
```

Shallow Copy

- Consider Example:

```
int x = 10;     //Initialization
int y = x;      //Initialization : Shallow Copy
```

- Consider Example:

```
Complex c1( 10,20 ),c2;
c2 = c1;        //Assignment : Shallow Copy
```

- Consider Example:

```
Array a1( 3 );
a1.acceptRecord( );
Array a2 = a1;  //Initialization: Shallow Copy
a2.printRecord( );
```

- Process of copying state of object into another object as it is, is called shallow copy.
- It is also called as bit-wise copy / bit by bit copy.

Compiler can create copy of the object in following scenario:

1. If we pass variable / object as a argument to the function by value.
2. If we return object from function by value.
3. If we initialize object:

```
        Complex c2 = c1;
```

4. If we assign the object

```
        c2 = c1
```

5. By throwing object
6. If we catch object by value.

Deep Copy.

- It is also called as member-wise copy.
- By modifing some state, if we create copy of the object then it is called deep copy.

Conditions to create deep copy

1. Class must contain at least one pointer type data member.

```
class Array
{
private:
        int size;
        int *arr;       //Case - I
public:
        Array( int size )
        {
                this->size = size;
                this->arr = new int[ this->size ];
        }
};
```

2. Class must contain user defined destructor.

```
class Array
{
public:
        ~Array( void )  //Case-II
        {
                if( this->arr != NULL )
                {
                        delete[] this->arr;
```

```
                        this->arr = NULL;
                }
        }
};
```

3. We must create copy of the object;

```
Array a1( 3 );
Array a2 = a1;  //Case III
//or
Array a2(2);
a2 = a1;        //Case III
```

Steps to create deep copy

1. Copy the required size from source object into destination object.
2. Allocate new resource for the destination object.
3. Copy the contents from resource of source object into destination object.

Location to create deep copy

1. In Case of assignment, we should create deep inside assignment operator function (operator=( )).
2. In remaining case, we should create deep copy inside copy constructor.

Copy Constructor

- "memcpy" is a function declared in string header file(#include)
- Syntax:

```
void* memcpy(void *dest, void *src, size_t size );
```

- The memcpy() function copies size bytes from memory area src to memory area dest.

```
int arr1[ 3 ] = { 10,20,30};
int arr2[ 3 ];
memcpy( arr2, arr1, sizeof( int ) * 3 );
```

- Copy constructor is a parametered constructor of the class which take single parameter of same type but using reference.

General Syntax of Copy Constructor:

```

```

```
class ClassName
{
public:
        //this : Address of dest object
        //other : Reference of src object
        ClassName( const ClassName &other )
        {
                //TODO : Shallow/Deep Copy
        }
};
```

Copy constructor gets called in following conditions:

1. If we pass object(of structure/class) as a argument to the function by value then on function parameter, copy constructor gets called.
2. If we return object from function by value then to store the result compiler implicitly create annonymoys object inside memoty. On that annonymous object, copy constructor gets called.
3. If we try to initialize object from another object then on destination object, copy constructor gets called.
4. If we throw object then its copy gets created into stack frame. To create copy on stack frame, copy constructor gets called.
5. If we catch object by value then on catching object, copy constructor gets called.

- If we do not define copy constructor inside class then compiler generate copy constructor for the class. It is called, default copy constructor. By default it creates shallow copy.
- Job of constructor is to initialize object. Job of destructor is to release the resources. Job of copy constructor is to initialize newly created object from existing object.
- Note : Creating copy of object is expesive task hence we should avoid object copy operation. To avoid the copy, we should use reference.
- During initialization of object, if there is need to create deep copy then we should define user defined copy constructor inside class.

Operator Overloading

- operator is token in C/C++.
- It is used to generate expression.
- operator is keyword in C++.
- Types of operator:
    1. Unary operator
    2. Binary Operator
    3. Ternary operator
- Unary Operator:
    - If operator require only one operand then it is called unary operator.
    - example : Unary(+,-,*) , &, !, ~, ++, --, sizeof, typeid etc.
- Binary Operator:
    - If operator require two operands then it is called binary operator.
    - Example:
        1. Arithmetic operator

        2. Relational operator
        3. Logical operator
        4. Bitwise operator
        5. Assignment operator
- Ternary operator:
    - If operator require three operands then it is called ternary operator.
    - Example:
        1. Conditional operator(? :)
- Consider following code:

```
int num1 = 10;  //Initialization
int num2 = 20;  //Initialization
int num3 = num1 + num2; //OK
```

- In C/C++, we can use operator with objects of fundamental type directly.(No need to write extra code). Hence above statement is valid.
- Consider following C source Code:

```
struct Point
{
        int x;
        int y;
};
int main( void )
{
        struct Point pt1 = { 10,20};
        struct Point pt2 = { 30,40};
        struct Point pt3;
        pt3 = pt1 + pt2;        //Not OK
        //pt3.x = pt1.x + pt2.x;
        //pt3.y = pt1.y + pt2.y;
        return 0;
}
```

- In C language, we can not use operator with objects of user defined type directly/indirectly. Hence above code is invalid.
- Consider following code in C++:

```
class Point
{
private:
        int x;
        int y;
public:
        Point( void )
        {
```

```
                this->x = 0;
                this->y = 0;
        }
        Point( int x, int y )
        {
                this->x = x;
                this->y = y;
        }
};
int main( void )
{
        Point pt1(10,20);
        Point pt2(30,40);
        Point pt3;
        pt3 = pt1 + pt2;        //Not OK( implicitly)
        return 0;
}
```

- In C++, also we can not use operator with objects of user defined type directly.
- If we want to use operator with objects of user defined type then we should overload operator.
- To overload operator, we should define operator function.
- We can define operator function using 2 ways:
    1. Using member function
    2. Using non member function.
- By defining operator function, it is possible to use operator with objects of user defined type. This process of giving extension to the meaning of operator is called operator overloading.
- Using operator overloading we can not define user defined operators rather we can increase capability of existing operators.

Limitations of operator overloading

- We can not overloading following operator using member as well as non member function:

1. dot/member selection operator(.)
2. Pointer to member selectiion operator(.*)
3. Scope resolution operator(::)
4. Ternary/conditional operator(? :)
5. sizeof() operator
6. typeid() operator
7. static_cast operator
8. dynamic_cast operator
9. const_cast operator
10. reinterpret_cast operator

- We can not overload following operators using non member function:

1. Assignment operator(=)
2. Subscript / Index operator([])
3. Function Call operator[()]

4. Arrow / Dereferencing operator(->)

- Using operator overloading, we can change meaning of operator.
- Using operator overloading, we can not change number of parameters passed to the operator function.

Operator overloading using member function(operator function must be member function):

- If we want to overload, binary operator using member function then operator function should take only one parameter.

```
c3 = c1 + c2;   //c3 = c1.operator+( c2 )
```

- Using operator overloading, we can not change, precedance and associativity of the operator.

```
c4 = c1 + c2 + c3;
//c4 = c1.operator+( c2 ).operator+( c3 );
```

- If we want to overload unary operator using member function then operator function should not take any parameter.

Operator overloading using non member function(operator function must be global function):

- If we want to overload binary operator using non member function then operator function should take two parameters.

```
c3 = c1 + c2;   //c3 = operator+(c1,c2);
```

```
c4 = c1 + c2 + c3;
//c4 = operator+(operator+(c1,c2),c3);
```

- If we want to overload unary operator using non member function then operator function should take only one parameters.

```
c2 = ++ c1;     //c2=operator++( c1 );
```

Day 10

- If constructor and destructor is public then we can create object of the class inside member function as well as non member function.
- If constructor or destructor is private then we can create object of the class inside member function only.

- If constructor/destructor is private and if non member function is friend of class then we can create object of the class inside non member function.
- If Copy constructor is public then we can initialize object from existing object inside member function as well as non member function.
- If Copy constructor is private then we can initialize object from existing object inside member function only.
- Note: If copy constructor is private then we should avoid copy operation in non member function. We should is reference or pointer

```
class Complex
{
private:
        int real;
        int imag;
public:
        Complex( int real = 0, int imag = 0)
        {
                this->real = real;
                this->imag = imag;
        }
private:
        Complex( const Complex &other )
        {
                this->real = other.real;
                this->imag = other.imag;
        }
};
int main( void )
{
        Complex c1(10,20);      //OK
        Complex c2 = c1;        //Not OK
        Complex &c3 = c1;       //OK
        Complex *c4 = &c1;      //OK
        return 0;
}
```

Overloading Insertion Operator(<<)

```
Header File : <iostream>
namespace std
{
        extern ostream cout;
}
```

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

- cout is an external object of ostream class which is declared in std namespace.
- ostream class is typdef of basic_ostream class.
- cout represents monitor.

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

- If we want print state of object on console(monitor) then we should use cout object and insertion operator(<<).
- Copy constructor of ostream class is private hence we can not copy of cout object inside our program

```
//ostream out = cout;   //Not OK
```

- If we want to avoid copy then we should use reference.

```
ostream &out = cout;    //OK
```

- If we want to print state of object(of structure/class) on console then we should overload insertion operator.

```
1. cout<<c1;    //cout.operator<<( c1 );
2. cout<<c1;    //operator<<(cout, c1 );
```

- According to first statement, to print state of c1 on console, we should define operator<<( ) function inside ostream class. But ostream class is library defined class hence we should not modify its implementation.
- According to second statement, to print state of c1 on console, we should define operator<<( ) function globally. Which possible for us. Hence we should overload operator<<( ) using non member function.
- General Syntax

```
class ClassName
{
        friend ostream& operator<<( ostream &cout, ClassName &other );
};
ostream& operator<<( ostream &cout, ClassName &other )
{
        //TODO : print state of object using other
        return cout;
}
```

Overloading Extraction Operator(>>)

```
Header File : <iostream>
namespace std
{
        extern istream cin;
}
```

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

- cin stands for character input. It represents keyboard.
- cin is external object of istream class which is declared in std namespace.
- istream class is typedef of basic_istream class.

```
int main( void )
{
        int number;
        cin>>number;
        return 0;
}
```

- If we want to accept data/state of the variable/object from console/keyboard then we should use cin object and extraction operator.
- Copy constructor of istream class is private hence, we can not create copy of cin object in out program.

```
istream in = cin;       //Not OK
```

- To avoid copy, we should use reference.

```
istream &in = cin;      //OK
```

- If we want to accept state of object (of structure/class) from console(keyboard) then we should overload extraction operator.

```
1. cin>>c1;     //cin.operator>>( c1 )
2. cin>>c1;     //operator>>( cin, c1 );
```

- According to first statement, to accept state of c1 from console, we should define operator>>( ) function inside istream class. But istream class is library defined class hence we should not modify its implementation.
- According to second statement, to accept state of c1 from console, we should define operator>>( ) function globally. Which possible for us. Hence we should overload operator>>( ) using non member function.
- General Syntax:

```
class ClassName
{
        friend istream& operator>>( istream &cin, ClassName &other );
};
istream& operator>>( istream &cin, ClassName &other )
{
        //TODO : accept state of object using other
        return cin;
}
```

Overloading Call / Function Call operator:

- If we want to consider any object as a function then we should overload function call operator.

```
class Complex
{
private:
        int real;
        int imag;
public:
        Complex( int real = 0, int imag = 0 )
        {
                this->real = real;
                this->imag = imag;
        }
        void operator()( int real, int imag )
        {
                this->real = real;
                this->imag = imag;
        }
        void printRecord( void )
        {
                cout<<"Real Number      :       "<<this->real<<endl;
                cout<<"Imag Number      :       "<<this->imag<<endl;
        }
};
int main( void )
{
        Complex c1;
        c1( 10,20 );//c1.operator()( 10, 20 );
        c1.printRecord( );
```

```
        return 0;
}
```

- If we use any object as a function then such object is called function object or functor.
- In above code, c1 is function object.

Index/Subscript Operator Overloading

- If we want to overcome limitations of array then we should encapsulate array inside class and we should perform operations on object by considering it array.
- If we want to consider object as a array then we should overload sub script/index operator.

```
//Array *const this = &a1
int& operator[]( int index )throw( ArrayIndexOutOfBoundsException )
{
        if( index >= 0 && index < SIZE )
                return this->arr[ index ];
        throw ArrayIndexOutOfBoundsException("Array Index Out Of Bounds
Exception");
}
```

- If we use subscript operator with object at RHS of assignment operator then expression must return value from array.

```
Array a1;
cin>>a1;        //operator>>( cin, a1 );
cout<<a1;       //opeator<<( cout, a1 );
int element = a1[ 2 ];
//int element = a1.operator[]( 1 );
```

- If we want to use sub script operator with object at LHS of assignment operator then expression should not return a value rather it should return either address / reference of memory location.

```
Array a1;
cin>>a1;        //operator>>( cin, a1 );
a1[ 1 ] = 200;
//a1.operator[]( 1 ) = 200;
cout<<a1;       //opeator<<( cout, a1 );
```

Overloading assignment operator.

- If we initialize newly created object from existing object of same class then copy constructor gets called.

```
Complex c1(10,20);
Complex c2 = c1;        //On c2 copy ctor will call
```

- If we assign, object to the another object then assignment operator function gets called.

```
Complex c1(10,20);
Complex c2;
c2 = c1;        //c2.operator=( c1 )
```

- General Syntax:

```
class ClassName
{
public:
        ClassName& operator=( const ClassName &other )
        {
                //TODO : Shallow/Deep Copy
                return *this;
        }
};
```

- If we do not define assignment operator function inside class then compiler generates default assignment operator function for the class. By default it creates shallow Copy.
- During assignment, if there is need to create deep copy then we should overload assignment operator function.

```
//Array *const this = &a2
//const Array &other = a1;
Array& operator=( const Array &other )
{
        this->~Array();
        this->size = other.size;
        this->arr = new int[ this->size ];
        memcpy(this->arr, other.arr, this->size * sizeof( int ) );
        return *this;
}
```

Overloading dereferencing/arrow operator.

- Consider following code:

```
class Array
{       };
```

- Static memory allocation for the object

```
class Array a1;
//or
Array a1;
```

- Dynamic memory allocation for the object

```
Array *ptr = new Array( );

delete ptr;
ptr = NULL;
```

- If we create object of a class dynamically then implicitly ctor gets called and if try to deallocate that memory then dtor gets called.

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

        delete ptr;     //Array::~Array( );
        return 0;
}
```

- Using object, if we want to access members of the class then we should use dot/member selection operator.
- In other words, using dot operator, if we want to access members of the class then left hand operand must be object of a class.

```
int main( void )
{
        Array a1;
        a1.acceptRecord();
        a1.printRecord();
        return 0;
}
```

- Using pointer, if we want to access members of the class then we should use arrow or dereferencing operator.
- In other words, using arrow operator, if we want to access members of the class then left side operand must be pointer of a class.

```
int main( void )
{
        Array *ptr = new Array( );
        ptr->acceptRecord();
        ptr->printRecord();
        delete ptr;
        return 0;
}
```

- if we use any object as a pointer then such object is called smart pointer.

```
class AutoPtr
{
private:
        Array *ptr;
public:
        AutoPtr( Array *ptr )
        {
                this->ptr = ptr;
        }
        Array* operator->( )
        {
                return this->ptr;
        }
        ~AutoPtr( )
        {
                delete this->ptr;
        }
};
int main( void )
{
        AutoPtr obj( new Array( 3 ) );
        obj->acceptRecord();
        //obj.operator ->()->acceptRecord();
        obj->printRecord();
        //obj.operator ->()->printRecord();
        return 0;
}
```

Day 11

Conversion Function

- It is a member function of a class which is used to convert state of object of fundamental type into user defined type or vice versa.
- Following are conversion functions in C++
    1. Single Parameter Constructor

2. Assignment operator function
3. Type conversion operator function.

Single Parameter constructor:

```
int main( void )
{
        int number = 10;
        Complex c1 = number;
        //Complex c1( number );
        c1.printRecord();
        return 0;
}
```

- In above code, single parameter constructor is responsible for converting state of number into c1 object. Hence single parameter constructor is called conversion function.

Assignment operator function

```
int main( void )
{
        int number = 10;
        Complex c1;
        c1 = number;//c1 = Complex( number );
        //c1.operator=( Complex( number ) );
        c1.printRecord();
        return 0;
}
```

- In above code, assignment operator function is responsible for converting state of number into c1 object hence it is considered as converion function.
- If we want to put restriction on automatic instantiation then we should declare single parameter constructor explicit.
- "explicit" is a keyword in C++.
- We can use it with any constructor but it is designed to use with single parameter constructor.

Type conversion operator function

```
int main( void )
{
        Complex c1(10,20);
        int real = c1;  //real = c1.operator int( )
        cout<<"Real Number      :       "<<real<<endl;
        return 0;
}
```

- In above code, type conversion operator function is responsible for converting state of c1 into integer variable(real). Hence it is considered as conversion function.

Operator overloading using member function vs non member function:

- During operator overloading, if left side operand need not to be l-value then we should overload operator using non member function.
- We should overload following operators using non member function:
    1. Arithmetic Operators
    2. Relational Operators
    3. Logical Operators
- During operator overloading, if left side operand need to be l-value then we should overload operator using member function.
- We should overload following operators using member function:
    1. =, [], (), ->
    2. Short hand operators
    3. Unary Operators(++, --)

Template

- If we want to write generic program in C++ then we should use template.
- Using template we can not reduce code size or execution time but we can reduce developers effort.
- It is designed for implementing generic data structure and algorithms
- Types of template:
    1. Function Template
    2. Class Template

Function Template

```
//template<typename T>//T : Type Parameter
template<class T> //T : Type Parameter
void swap_number( T &o1, T &o2 )
{
        T temp = o1;
        o1 = o2;
        o2 = temp;
}
int main( void )
{
        int num1 = 10;
        int num2 = 20;
        swap_number<int>( num1, num2 );
        //Here int is type argument
        cout<<"Num1     :       "<<num1<<endl;
        cout<<"Num2     :       "<<num2<<endl;
        return 0;
}
```

- Type inference : It is ability of compiler to detect type of argument at compile time ans passing it as a argument to the function.

```
template<class X, class Y>
void swap_number( X &o1, Y &o2 )
{
        X temp = o1;
        o1 = o2;
        o2 = temp;
}
int main( void )
{
        float num1 = 10.5f;
        double num2 = 20.5;
        swap_number<float, double>(num1,num2 );
        cout<<"Num1     :       "<<num1<<endl;
        cout<<"Num2     :       "<<num2<<endl;
        return 0;
}
```

- We can pass multiple type arguments to the function.
- Using template argument list, we can pass data type as a argument to the function.
- Using template we can write type safe generic code.

Class Template

- In C++, by passing data type as a argument, we can write generic code hence parameterized type is called template.

```
template<class T>
class Array // Parameterized type
{
private:
        int size;
        T *arr;
public:
        Array( void ) : size( 0 ), arr( NULL )
        {       }
        Array( int size )
        {
                this->size = size;
                this->arr = new T[ this->size ];
        }
        void acceptRecord( void ){      }
        void printRecord( void ){       }
        ~Array( void ){ }
};
int main( void )
{
```

```
        Array<char> a1( 3 );
        a1.acceptRecord();
        a1.printRecord();
        return 0;
}
```

- We can not divide template code into multiple files.
- Standard Template Library(STL) is a collection of readymade template data structure classes and algorithms.

Storage Classes

```
1. auto
2. static
3. extern
4. register
```

- Storage classes decides scope and lifetime of the element.

Scope

- Types of scope in C++:
    1. Block scope
    2. Function scope
    3. Prototype scope
    4. Class scope
    5. Namespace scope
    6. File scope
    7. Program scope
- It decides area/region/boundry in which we can access the element.

```
int num6;       //Program Scope
static int num5;        //File Scope
namespace ntest
{
        int num4;       //Namespace scope
        class Test
        {
                int num3;       //Class Scope
        };
}
void sum( int num1, int num2 ); //Prototype scope
int main( void )
{
        int num1 = 10;  //Function Scope
        while( true )
        {
```

```
                int temp = 0;   //Block Scope
        }
        return 0;
}
```

Lifetime

- Types of lifetime:
    1. Automatic Lifetime
    2. Static Lifetime
    3. Dynamic Lifetime
- It decides time/duration. In other words how long variable will be exist inside memory is nothing but lifetime of the variable.

Local Variable

- A variable, which is declared inside block/function is called local variable.

```
int main( void )
{
        //Local Variable
        int num1 = 10;  //Function Scope
        {
                //Local Variable
                int num2 = 20;  //Block Scope
        }
        return 0;
}
```

- Local variable get space onces per function call on stack segment.
- Default value of local variable is garbage.
- Local variable get space automatically and its space gets deallocated automatically hence it is also called as automatic variable.
- Default storage class of local variable is auto.

```
void print( void )
{
        int count = 0;
        ++ count;
        cout<<"Count    :       "<<count<<endl;
}
int main( void )
{
        ::print();      //1
        ::print();      //1
        ::print();      //1
```

```
        return 0;
}
```

Static and global variable:

- All the static and global variables get space only once during program loading / before starting execution of main function.

```
int num1;               //BSS Segment
int num2 = 10;  // Data Segment
static int num3;        ////BSS Segment
static int num4 = 20;   // Data Segment
int main( void )
{
        int num5;       //Stack Segment
        static int num6;        //BSS Segment
        static int num7 = 30; //Data Segment
        return 0;
}
```

- Unintialized static and global variable get space on BSS segment.
- Intialized static and global variable get space on Data segment.
- Default value of static and global variable is zero.
- Static variables are same as global variables but it is having limited scope.

```
void print( void )
{
        static int count = 0;
        ++ count;
        cout<<"Count    :       "<<count<<endl;
}
int main( void )
{
        ::print();      //1
        ::print();      //2
        ::print();      //3
        return 0;
}
```

- If we want to share, value of local variable between function call then we should declare local variable static.
- Static variable is also called as shared variable.
- If we dont want to access any global function inside different file then we should declare global function static.
- If we declare function static then local variables are not considered as static.
- In C/C++, we can not declare main function static.

Static Data Member

- If we want to share value of the data member in all the objects of same class then we should declare data member static.
- Static data member do not get space inside object rather all the objects of same class share single copy of it. Hence size of object is depends on size of all the non static data members declared inside class.
- If class contains all static data members then size of object will be 1 byte.
- Data member of a class, which get space inside object is called instance variable. In short, non static data member is also called as instance variable.
- Instance variable gets space once per object. Hence to access it we must use object, pointer or reference.
- Data member of the class, which do not get space inside object is called class level variable. In other words, static data member is also called as class level variable.
- Class level variable get space once per class. Hence to access it we should use class name and scope resolution operator.
- If we want to declare data member static then we must provide global definition for it otherwise linker generates error.
- Instance variable get space inside instance hence we should initialize it using constructor.
- Class level variable do not get space inside instance hence we should not initialize it inside constructor. We must initialize it in global definition.
- We can declare constant data member static.

Static Member Function

- We can not declare global function constant but we can declare member function constant.
- Except main function, we can declare global function as well as member function static.
- To access non static members of the class, we should declare member function non static and to access static members of the class we should declare member function static.
- Member function of a class which is designed to call on object is called instance method. In short non static member function is also called as instance method.
- To access instance method either we should use object, pointer or reference to object.
- Member function of a class which is designed to call on class name is called class level method. In short static member function is also called as class level method.
- To access class level method we should use classname and :: operator.
- Since static member functions are not designed to call on object it doesnt get this pointer.
- this pointer is considered as link between non static data member and non static member function.
- Since static member function do not get this pointer, we can not access non static members inside static member function directly.
- Inside non static member function, we can access static as well as non static members.
- Using object, we can access non static members inside static member function.

```
class A
{
public:
        static void f1( void )
        {
                cout<<"A::f1"<<endl;
```

```
        }
};
class B
{
public:
        static void f2( void )
        {
                cout<<"B::f2"<<endl;
        }
        static void f3( void )
        {
                //f1( );        //Not OK
                A::f1();        //OK

                f2( );  //OK
                B::f2( );       //OK
        }
};
int main( void )
{
        //f3( );        //Not OK
        B::f3( );
        return 0;
}
```

- We can declare static data member constant but we can not declare static member function constant.
- We can not declare static member function constant, volatile and virtual.

Day 12

Macro

- Example:
    1. NULL
    2. EOF
    3. **FILE**
    4. **LINE**
    5. **DATE**
    6. **TIME**
    7. **FUNCTION**
- Symbolic constant is called as macro
- How define macro:

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

- Expanding macro is a job of preprocessor.
- Advantages:
    1. It helps to improve readability of source code.
    2. It helps to reduce maintenance of program
    3. If we use macro in a program then it helps to execute application very fast.
- Disadvantages:

1. Use of macro may increase code size.
2. We can not write typesafe code using macro.

```
#define PI      3.142   //Not type safe
const double PI=3.142;  //Type Safe
```

- Note : Job of preprocessor is to find and replace. In other words to expand the macros.

```
#define FUN( X, Y)      X * Y
int main(void )
{
        cout<< FUN( 2 + 3, 2 + 3 )<<endl; //11
        cout<< FUN( (2+3), (2+3) )<<endl;//25
        return 0;
}
```

Inline Function

- Process :
    1. Program in execution is called process.
    2. Running instance of application is called process.
- Process is also called as task.
- Thread :
    1. Light weight process / sub process is called thread.
    2. It is a seperate path of execution which runs independantly.
- Thread always resides within process.
- Single tasking:
    - An ability of operating system to execute single process at a time is called single tasking
    - MS DOS is single user single tasking operating system.
- Multitasking:
    - An ability of operating system to execute multiple processes at a time is called multitasking
    - All modern operating systems are multitasking.
    - Multitasking cab be achieved using process/thread.
- If any application use single thread then it is called single threaded application.
- By default every application/process is single thread.
- If any application uses multiple threads then it is called multithreaded application.
- Operating system maintains one stack per thread. It is called runtime stack. This stack is used to store stack frames. And stack frame contains Function Activation record.

- Function Activaton Record(FAR) contains:
    1. Function Parameter
    2. Local Variable
    3. Function Call
    4. Tempory storage for calculation
    5. Return Address
- Managing function activation record is a job of compiler.
- If we give call to the function then compiler need to create Stack Frame and push it into stack. Upon returning control back to the calling function it needs to destroy stackframe from stack. In other words giving call the function is overhead to the compiler.
- If we want to reduce compilers overhead then we should use inline keyword.
- If we declare function inline then compiler do not call function rather it replaces function call by function body.
- Inline is request to the compiler.
- In following cases, function is not considered as inline:
    1. If we use loop(for/while) inside function
    2. If we implement function using recursion
    3. If we use jump statement inside function
- In case of modular approach, we can use inline keyword in either declaration, definition or both places.
- We can not declare main function static, constant, virtual or inline.
- If we define member function inside class then function are by default considered as inline.
- If we want make member function inline whose definition is global then we must explicitly use inline keyword.

```
class Complex
{
private:
        int real;
        int imag;
public:
        Complex( int real = 0, int imag = 0 );
        void print( void );
};
inline Complex::Complex( int real, int imag)
{
        this->real = real;
        this->imag = imag;
}
inline void Complex::print( void )
{
        cout<<"Real Number      :       "<<this->real<<endl;
        cout<<"Imag Number      :       "<<this->imag<<endl;
}
```

- We can not divide inline function code in multiple files.

OOPS Concepts

- Major Pillars/Parts/Elements:
    1. Abstraction
    2. Encapsulation
    3. Modularity
    4. Hierarchy
- Minor Pillars/Parts/Elements
    1. Typing
    2. Concurrency
    3. Persistence

Abstraction

- It is a major pillar of oops.
- Its main purpose is to achive simplicity.
- Process of getting essential things from object is called abstraction.
- Abstraction describes outer behavior of object.
- Abstraction focuses on the essential characteristics of some object, relative to the perspective of the viewer.
- Abstraction in program:

```
int main( void )
{
        Complex c1;
        c1.accept( );
        c1.print( );
        return 0
}
```

- If we create object and call member functions on it then it represents abstraction programatically.

Encapsulation

- It is a major pillar of oops
- Its main pupose is to achive abstraction and data hiding.
- Definition:
    1. Binding of data and code together is called encpasulation
    2. To achive abstraction we need to provide some implementation. It is called encapsulation.
- By defining class we achive encapsulation

```
class Complex
{
private:
        int real;
        int imag;
public:
        void accept( void )
        {       }
```

```
        void print( void )
        {       }
};
```

- Encapsulation describes internal behavior of object.
- Hiding represents encapsulation.
- Abstraction and encapsulation are complementary concepts: Abstraction focuses on the observable behavior of an object, whereas encapsulation focuses on the implementation that gives rise to this behavior.

Data Hiding:

- Process of declaring members of the class private is called hiding.
- Process of declaring data member private is called data hiding.
- Data hiding is also called data encapsulation.
- Data hiding is required to achive data security.

Data Security

- Process of giving controlled access to the data is called data security.
- Consider example:

```
class Report
{
private:
        int rollNumber;
        float marks;
public:
        float getMarks( )
        {
                return this->marks;
        }
        void setMarks( float marks )
        {
                if( marks >=0 && marks <= 100 )
                        this->marks = marks;
                else
                        throw IllegalArgumentException("Invalid marks");
        }
};
```

- If we want to achive data security then:
    1. We should encapsulate data member inside class and we should should make it private
    2. We should access private data members using member function only(getter/setter)
    3. If data is valid then and only then it should be assign to data member. Hence we should write controlled access logic inside setter method.
- Main purpose of data security is to achive data reliability.

Modularity

- It is a major pillar of oops.
- Process of developing complex system using small parts is called modularity.
- Main purpose of modularity is to minimize module dependancy.
- With the help of library files we can achive modularity.
- Examples:
    1. MS.NET -> .dll
    2. Java -> .jar / .war / .ear
    3. C/C++ -> .a / .so, .lib / .dll

Hierarchy

- It is a major pillar of oops.
- Its main purpose is to achive reusability.
- Advantages of reusability:
    1. To reduce develoers efforts.
    2. To reduce development time and development cost.
- Level / order / ranking of abstraction is called hierarcy.
- Types of Hierarcy:

1. Has-a/Part-of Association/Containment
2. Is-a/Kind-of Inheritance/Generalization
3. Use-a Dependancy
4. Creates-a Instantiation

- Unified Modelling Language(UML).
- Inventor of UML is Grady Booch.

Typing

- It is minor pillar of oops.
- It is also called as polymorphism
- polymorphism = poly(many) + morphism(forms)
- An ability of object to take multiple forms is called polymorphism.
- Main purpose of polymorphism is to reduce maintenance of system.
- Types of polymoprhism:

1. Compile time polymoprhism
    - It is also called as static polymoprhism / Early Binding / False polymoprhism / Weak Typing
    - We can achive it using:
        1. Function Overloading
        2. Operator Overloading
        3. Template
2. Runtime polymorphism
    - It is also called as dynamic polymoprhism / Late Binding / True polymoprhism / Strong Typing
    - We can achive it using:
        1. Function Overriding

Concurrency

- It is minor pillar of oops
- In context of OS it is called multitasking
- Process of executing multiple task simultaneously is called concurrency.
- If we want to utilize hardware resources efficiently then we should use concurrency.
- Using multithreading, we can achive concurrency.

Persistance

- It is minor pillar of oops
- It is the process of maintaing state of object on secondry storage(HDD).
- We can achive it using file handling and database programming.

Association

- Example:
    1. Room has-a wall
    2. Room has-a chair
    3. Car has-a engine
    4. Car has-a music player
    5. Department has-a faculty
    6. Human has-a heart
- If has-a relationship exist between two types then we should use association.
- Consider following example:
    - Car has-a engine
- In other words:
    - engine is part-of car.
- Let us discuss programatically:

```
class Engine
{       };
class Car
{
private:
        Engine e;       //Association
};
int main( void )
{
        Car car;
        return 0;
}
//Dependant Object : Car Object
//Dependancy Object : Engine Object
```

- Consider another example:
    - Employee has-a join date

```
class Date
{       };
class Employee
{
        Date joinDate;  //Association
};
```

- If object is part-of / component of another object then it is called association.
- Composition and aggregation are specialized form of association.
- If we declare object of a class as a data member inside another class then it represents association.

Composition

- Consider following example:
    - Human has-a heart.

```
class Heart
{       };
class Human
{
        Heart hrt; //Association->Composition
};
int main( void )
{
        Human h;
        return 0;
}
//Dependant Object : Human Object
//Dependancy Object : Heart Object
```

- If dependancy object do not exist without Dependant object then it represents composition.
- Composition represents tight coupling.

Aggregation

- Consider following example:
    - Department has-a faculty.

```
class Faculty
{       };
class Department
{
        Faculty f; //Association->Aggregation
};
int main( void )
{
        Department d;
```

```
        return 0;
}
//Dependant Object : Department Object
//Dependancy Object : Faculty Object
```

- If dependancy object exist without Dependant object then it represents Aggregation.
- Aggregation represents loose coupling.

Inheritance

- Consider example:
    1. Employee is-a Person
    2. Book is-a product
    3. Car is-a Vehicle
    4. Rectangle is-a Shape
    5. Loan Account is-a Account
- If "is-a" relationship exist between two types then we should use inheritance.
- Inheritance is also called as "Generalization".

```
//Parent class
class Person//Base class
{       };
//Child class
class Employee:public Person//Derived class
{       };
```

- During inheritance, members of base class inherit into derived class.
- If we create object of derived class then non static data members declared in base class get space inside it. In other words non static data members of base class inherit into derived class.
- Using derived class name, we can access static data member(if public) of base class. In other words, static data member inherit into derived class.
- All the data members(private/protected/public, static/non static) of base class inherit into derived class but only non static data members get space inside object.
- Size of object = sum of size of non static data members declared in base class and derived class.
- We can call non static member function of base class on object of derived class. In other words, using derived class object we can call non static member function of base class. It means that, non static member function inherit into derived class.
- We can call static member function of base class on derived class. In other words, using derived class name, we ca access static member function of base class. It means that, static member function inherit into derived class.
- Following function's do not inherit into derived class:
    1. Constructor
    2. Destructor
    3. Copy constructor
    4. Assignment operator function
    5. Friend Function

- Except above five function's, all the member's of base cass(data member, member function and nested type) inherit into derived class.
- If we create object of derived class then first base class and then derived class constructor gets called. Destructor calling sequence is exactly opposite.
- From derived class constructor, by default, base class's parameterless constructor gets called.
- Using constructors base initializer list, we can call any constructor of base class from constructor of derived class.
- In C++, we can not call constructor on object, pointer or reference explicitly. But constructor's base initializer list represent explict call to the constructor.
- We can read following statement using 2 ways:

```
class Employee : public Person
```

1. Class Person is inherited into class Employee.
2. Class Employee is derived from class Person(Recommended).

- If we use private/protected/public keyword to control visibility of members of class then it is called access specifier.
- If we use private/protected/public keyword to extend the class then it is called mode of inheritance.
- In following statement, mode of inheritance is public.

```
class Employee : public Person
```

- Default mode of inheritance is private.

```
class Employee : Person
//is equivalent to
class Employee : private Person
```

- Process of acquiring/getting/accessing properties(data members) and behavior (member function) of base class inside derived class is called is called inheritance.
- Every base class is abstraction for the derived class.

Day 13

- In all types of mode, private members inherit into derived class but we can not access it inside member function of derived class.
- If we want to access private members inside derived class then:

1. Either we should use member function(getter/setter).
2. or we should declare derived class as a friend inside base class.

- If we want to create object of derived class then constrctor of base class and derived must be public.

- According to client's requirement, if implementation of existing class is logically incomplete / partially complete then we should extend the class i.e we should use inheritance.
- In other words, without changing implementation of exsiting class, if we want to extend meaning of that class then we should use inheritance.
- According client's requirement, if implementation of base class member function is logically incomplete then we should redefine function in derived class.
- If name of members of base class and derived class are same then derived class members hides implementation of base class members. Hence preference is given to the derived class members. This process is called shadowing.
- If we want to access members of base class inside member function of derived class then we should use classname and scope resolution operator.

Types of inheritance:

- During inheritance, if parent and child is interface then it is called interface inheritance.
- During inheritance, if parent and child is class then it is called implementation inheritance.
- During inheritance, if parent is interface and child is class then it is called interface implementation inheritance.

1. Interface Inheritance
    1. Single Inheritance
    2. Multiple Inheritance
    3. Hierarchical Inheritance
    4. Multilevel Inheritance
2. Implementation Inheritance
    1. Single Inheritance
    2. Multiple Inheritance
    3. Hierarchical Inheritance
    4. Multilevel Inheritance

Single Inheritance

- class B is derived from class A
- If single base class is having single derived class then it is called single inheritance.
- Syntax:

```
class A{        };
class B : public A{     };
```

- Example : Car is a Vehicle

Multiple Inheritance

- class D is derived from class A, B and C
- If multiple base classes are having single derived class then it is called multiple inheritance,
- Syntax:

```
class A{        };
class B{        };
class C{        };
class D : public A, public B, public C{ };
```

- Example : AllRounder

Hierarchical Inheritance

- class B, C and D are derived from class A
- If single base class is having multiple derived classes then such inheritance is called hierarchical inheritance.
- Syntax:

```
class A{        };
class B : public A{     };
class C : public A{     };
class D : public A{     };
```

- Example : Book and Tape is a Product.

Multilevel Inheritance

- class B is derived from class A, class C is derived from class B and class D is derived from class C.
- If single inheritance is having multiple levels then it is called multilevel inheritance.
- Syntax:

```
class A{        };
class B : public A{     };
class C : public B{     };
class D : public C{     };
```

- Example: Employee is a Person and Manager is a Employee, SalesManager is a Manager.

Hybrid Inheritance

- Combination of any two or more than two types of inheritance is called hybrid inheritance.

Diamond Problem

```
#include<iostream>
using namespace std;
class A
{
private:
```

```
        int num1;
public:
        A( void )
        {
                this->num1 = 10;
        }
        A( int num1 )
        {
                this->num1 = num1;
        }
        void printRecord( void )
        {
                this->showRecord( );
        }
protected:
        void showRecord( void )
        {
                cout<<"Num1     :       "<<this->num1<<endl;
        }
};
class B : virtual public A
{
private:
        int num2;
public:
        B( void )
        {
                this->num2 = 20;
        }
        B( int num2 )
        {
                this->num2 = num2;
        }
        B( int num1, int num2 ) : A( num1 )
        {
                this->num2 = num2;
        }
        void printRecord( void )
        {
                A::showRecord();        //Num1
                this->showRecord();     //Num2
        }
protected:
        void showRecord( void )
        {
                cout<<"Num2     :       "<<this->num2<<endl;
        }
};
class C : virtual public  A
{
private:
        int num3;
public:
        C( void )
```

```
        {
                this->num3 = 30;
        }
        C( int num3 )
        {
                this->num3 = num3;
        }
        C( int num1, int num3 ) : A( num1 )
        {
                this->num3 = num3;
        }
        void printRecord( void )
        {
                A::showRecord();        //Num1
                this->showRecord();     //Num3
        }
protected:
        void showRecord( void )
        {
                cout<<"Num3     :       "<<this->num3<<endl;
        }
};
class D : public B, public C
{
private:
        int num4;
public:
        D( void )
        {
                this->num4 = 40;
        }
        D( int num1, int num2, int num3, int num4 ) : A( num1 ),B( num2 ),
C(  num3 )
        {
                this->num4 = num4;
        }
        void printRecord( void )
        {
                A::showRecord();        //Num1
                B::showRecord();        //Num2
                C::showRecord();        //Num3
                this->showRecord();     //Num4
        }
protected:
        void showRecord( void )
        {
                cout<<"Num4     :       "<<this->num4<<endl;
        }
};
int main( void )
{
        D obj( 100,200,300,400);
        obj.printRecord();
```

```
        return 0;
}
```

Runtime Polymorphism:

- Members of derived class do not inherit into the base class hence using base class object we can access members of base class only.

```
int main( void )
{
        Base base;
        //base.showRecord();//Base::showRecord
        //base.printRecord();//Base::printRecord
        //base.Derived::printRecord();//Not OK
        //base.displayRecord( );//Not OK
        return 0;
}
```

```
int main( void )
{
        Base *ptrBase = new Base( );
        //ptrBase->showRecord();        //Base::showRecord
        //ptrBase->printRecord();       //Base::printRecord
        //ptrBase->Derived::printRecord();      //Not OK
        //ptrBase->displayRecord( );    //Not OK
        delete ptrBase;
        return 0;
}
```

- During inheritance, members of base class inherit into derived class hence using derived class object, we can access members of base class as well as derived class.

```
int main( void )
{
        Derived derived;
        //derived.showRecord(); //Base::showRecord
        //derived.printRecord();        //Derived::printRecord
        //derived.Base::printRecord();  //Base::printRecord
        //derived.displayRecord();      //Derived::displayRecord
        return 0;
}
```

- Members of base class inherit into derived class hence derived class object can be considered as base class object.
- Example : Employee object is-a Person object.

- Since Derived class object can be considered as Base class object, we can use it in place of Base class object.

```
Base b1;
Base b2 = b1;   //OK
```

```
Base b1;
Derived d1;
b1 = d1;        //OK
```

```
Base *ptr = NULL;
ptr = new Base();       //OK
```

```
Base *ptr = NULL;
ptr = new Derived();    //OK
```

- If we assign derived class object to the base class object then compiler copies state of base class portion from derived class object into base class object. It is called Object slicing. `
- During Object slicing, mode of inheritance must be public.

```
int main( void )
{
        Base base;
        Derived derived( 500,600,700);
        base = derived; //OK : Object Slicing
        base.printRecord();     //Base::printRecord() : 500,600
        return 0;
}
```

- Members of derived class do not inherit into base class. Hence base class object, can not be considered as derived class object.
- Since base class object, can not be considered as derived class object, we can not use it in place of derived class object.

```
Derived d1;
Derived d2 = d1;        //OK
```

```
Base b1;
Derived d2 = b1;        //Not OK
```

```
Derived *ptr = NULL;
ptr = new Derived();    //Ok
```

```
Derived *ptr = NULL;
ptr = new Base();       //Not Ok
```

- Process of converting, pointer of derived class into pointer of base class is called upcasting.
- Upcasting represents object slicing.
- In case of upcasting, explicit type casting is optional.

```
int main( void )
{
        Derived *ptrDerived = new Derived( );
        ptrDerived->printRecord();      //Derived::printRecord()

        //Base *ptrBase = ( Base* )ptrDerived;  //Upcasting
        Base *ptrBase = ptrDerived;     //Upcasting
        ptrBase->printRecord(); //Base::printRecord()
        return 0;
}
```

- Main purpose of upcasting is to reduce object dependancy in the code.
- Process of converting pointer of base class into pointer of derived class is called downcasting.
- In Case of downcasting, explicit typecasting is mandatory.

```
int main( void )
{
        Base *ptrBase = new Derived( ); //Upcasting
        ptrBase->printRecord(); //Base::printRecord()
        Derived *ptrDerived = ( Derived*)ptrBase;//Downcasting
        ptrDerived->printRecord();
        return 0;
}
```

- Note: Only in case of upcasting, we can do downcasting. Otherwise downcasting will fail.

Virtual Function:

- In case of upcasting, if we want to call function, depending on type of object rather than type of pointer then we should declare function in base class virtual.
- If class contains, at least one virtual function then such class is called polymorphic class.
- If signature of base class and derived class member function is same and if function in base class is virtual then derived class member function is by default considered as virtual.
- If base class is polymorphic then derived class is also considered as polymorphic.
- Process of redefining, virtual function of base class, inside derived class, with same signature, is called function overriding.
- Rules for function overriding
    1. Function must be exist inside base class and derived class(different scope)
    2. Signature of base class and derived class member function must be same(including return type).
    3. At least, Function in base class must be virtual.
- Virtual function, redefined in derived class is called overriden function.
- Definition 1: In case of upcasting, a member function, which gets called depending on type of object rather than type of pointer, is called virtual function.
- Definition 2: In case of upcasting, a member function of derived class which is designed to call using pointer of base class is called virtual function.
- We can call virtual function on object but it is designed to call on Base class pointer or reference.