

# Day 1

---

## Programming Language Introduction

- e.g : B, BCPL, C, Simula, Python, Java, C++ etc.
- Every language has its grammar/syntax
- Every language has its rules / semantics.
- Language contains token
- Language provide data types
- Language provide some built in features.
- If we want to implement business logic then we should use language.
- Types of application:
  1. CUI[ Console User Interface ]
  2. GUI[ Graphical User Interface ]
  3. Library
- Using language we can develop above application.

## C History

- C is a procedure oriented programming language developed by Dennis Ritchie.
- It is developed at AT&T Bell Lab in 1969-1972.
- It is developed on UNIX operating machine.
- Hardware : DEC PDP-11
- Extension of C source code should be ".c"

```
"File Name : Main.c"
#include<stdio.h>
int main( void )
{
    printf("Hello World\n");
    return 0;
}
```

## Tokens

- Program is collection of statements.
- An instruction given to computer is called statement.
- Every statement is made of tokens.
- token is basic unit of a program.
- Tokens in C:
  1. Identifier
    - Any name given to the variable, function, enum/union/structure is called identifier.
    - e.g
      1. int num1; //num1 : identifier

2. void main(); //main : identifier

## 2. Keyword

- It is reserved word of a language that can be used as a identifier.
- e.g. int, float, double, return break
- In original definition of C, there were 28 keywords( According "The C programming Language, First Edition by Dennis Ritchie").
- In second edition, "entry keyword was removed" hence keywords were 27.
- In 1989, 5 keywords were added.
- In 1999, 5 keywords were added.
- In 2011, 7 keywords were added.

## 3. Constant

- Constant is also called as literal
- e.g
  1. 'A' : Character constant
  2. "Pune" : String const
  3. 3.14 : double constant
  4. 58 : integer constant
- Types
  1. Character constant
  2. Integer constant
    - Octal constant
    - Decimal constant
    - Hexadecimal constant
  3. Floating point constant
    - Float constant
    - Double Constant
  - String constant
  - Enum constant.

```
enum ShapeType //Enum Name
{
    //Enum Constant's / Enumerator
    LINE, RECTANGLE, OVAL
    // LINE = 0, RECTANGLE = 1, OVAL = 2
}
```

## 4. Operator

- If we want write expression then we should user operator.
- e.g : int z = x + y;
- Types:
  1. Unary Operator
    - If operator require only one operand then it is called unary operator.
    - e.g
      - sizeof( number );
      - ++ number;

- & number
- example of unary operator ++, --, sizeof, !, &, unary(+, -, \*), ~ etc
- 2. Binary Operator
  - If operator requires two operands then it is called binary operator.
  - Following are binary operators
    1. Arithmetic operators +, -, \*, /, %
    2. Relational Operators <, >, <=, >=, ==, !=
    3. Logical operators &&, ||
    4. Bitwise operators &, |, ^, <<, >>
    5. Assignment operators =, Short hand operators(+=, -=, \*= )
- 3. Ternary Operator
  - If operator requires three operands then it is called ternary operator.
  - e.g. conditional operator.
  - `int result = x > y ? x : y;`
- 5. Punctuator / Separator e.g {, }, [, ], <, >, :, ;, : space, tab, comma(,)

## ANSI Standards

- American National Standard Institute.
- ANSI is responsible for updating language.
- Standardizing C, C++ SQL etc is a job of ANSI.
- C Standards: before 1989 : The C prog Language( D.R) 1989 : C89 1990 : C90 1995 : C95 1999 : C99 2011 : C11 2018 : C18

## Classification of languages

1. Machine Level Language
  - 1 and 0
2. Low Level Language
  - Assembly
3. High Level Language
  - C, C++ etc.

## Classification of High level programming languages:

1. Procedure Oriented Programming Languages
  - e.g PASCAL, COBOL, FORTRAN C etc.
  - FORTRAN is first high level POP lang
2. Object Oriented Programming Languages
  - e.g C++, Java, C#, Simula, Smalltalk
  - Simula is first OO prog language
3. Object Based Programming Languages
  - e.g Ada, Java Script, Visual Basic
  - Ada is first object based prog lang.
4. Rule Based Programming Languages
  - e.g LISP, PROLOG

- 5. Functional Oriented Programming Languages
  - e.g Python
- 6. Logic Oriented Programming Languages
- 7. Constraint Oriented Programming Languages

## Procedure Oriented Programming

- An entity, whose value we can not change/modify is called constant.
- e.g 0, 10, 3.14 etc
- An entity, whose value we can change/modify is called variable.
- e.g int x, y;
- variable is identifier of memory.
- If we want to declare variable then we should use data types.

## Data type

- Data type of variable describe 3 things:
  1. Size : How much memory is required to store the data.
  2. Nature : Which kind of data memory can store
  3. Operation : Which operations can be performed on the data stored in memory.

## Classification Data types

- Fundamental Data Types ( 5 )
  1. void : Implementation dependant
  2. char : 1 byte (Encoding:ASCII)
  3. int : 2/4 bytes
  4. float : 4 bytes
  5. double : 8 bytes
- Derived Data Types( 5 )
  1. Array : depends of size of type and count of elements
  2. Function : NA
  3. Pointer : 2/4/8 bytes
  4. Union ( User Defined ) : Depends on largest element in union
  5. Structure ( User Defined ) : depends of all the members.
- sizeof() is operator that is used to find out size of variable.
- C/C++/Java are statically type checked languages:

```
x = 10; //Not OK
int x = 10; //OK
```

- java Script/python are dynamically typed checked languages:

```
x = 10; //OK
```

- C/C++/Java are free form languages

```
int x=10,y=20,z;  
z=x+y; //OK  
z = x + y; //OK
```

### Type Modifiers

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

### Type qualifiers

1. const
2. volatile

### Declaration and definition

- Implementation of function is called function definition

```
//Function Definition  
void print( )  
{  
    printf("Hello Students\n");  
}
```

- If we want to use function then we must call the function.

```
//Function Definition  
int main( void ) //Calling Function  
{  
    print( );      //Function Call;  
    return 0;  
}  
//Function Definition  
void print( ) //Called Function  
{  
    printf("Hello Students\n");  
}
```

- Compiler compiles code from top to bottom but execution always start from main function.
- main() is entry point function in C/C++ that we can define any where in the program.
- Calling a main function is a job of operating System.

- Programmer is responsible for defining main function and os is responsible for calling it. In OS context it is called as callback function.
- main is user defined function.
- Signature of main:
  - void main();
  - void main( void );
  - int main();
  - int main( void ); // Recommended
  - void main( int argc, char \*argv[ ] );
  - int main( int argc, char \*argv[ ] );
- If we use any function before its implementation / definition then we must provide its declaration.

```
//Global Function Declaration
//void print(); //Function prototype / Declaration

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

    print( ); //Function Call;
    return 0;
}
```

- In C language, we can not define function inside another function / structure. In other words, c language functions are global.
- We can declare function locally as well as globally.
- If we try to access any element without definition then linker generates error.

```
int main( void ) //Calling Function
{
    void print(); //Function decl
    print( ); //Function Call;
    return 0;
}
//output : Linker Error
```

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

```
//Declaration as well as definition
int x; //Unintialized --> garbage value

//Declaration as well as definition
int y = 20; //Intialized --> 20
```

```
int main( void )
{
    extern int number;//Declaration
    printf("Number :      %d\n", number ); //Linker Error
    return 0;
}
```

```
extern int x; //Only declaration
int y; //Decl as well def
int z = 10; //Decl as well def
```

### Initialization and Assignment

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

- Initialization is the process of storing value inside variable during declaration.
- During life of the variable, we can initialize variable only once.

```
int x; //Initialized with garbage val
x = 10; //Assignment.
```

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

### Constant, pointer

#### Constant

- Keywords introduced in C89
  1. const
  2. volatile
  3. void
  4. enum
  5. signed
- const is a keyword introduced in C89.
- const is type qualifier.
- We can declare variable constant but we can not declare "global function" constant.
- If we dont want to modify value of the variable then we should use const keyword.

- constant variable is also called as read only variable.
- In C, Using pointer, we can modify value of constant variable. Hence initializing constant variable is optional.

```
const int x;           //Ok
const int y = 10;      //OK
```

### Locator Value( L-Value )

- Modifiable variable that is available at left hand side of assignment operator is called L-Value.

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

```
const int x = 10, y = 20;
x = x + 5; //Error : L-Value required
y = y + 5; //Ok : y is L-Value
```

### Reference Value( R-Value )

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

```
int x = 10;
int y = x;
int z = x + y;
```

### Pointer

- Named memory location / name given to memory location is called variable.
- "&" is unary operator which is used to get address of the variable.

```
int main( void )
{
    int number = 10;
    printf("Number : %d\n", number);
    printf("Number : %u\n", &number);
    printf("Number : %p\n", &number);
    return 0;
}
```

- Pointer is derived data type. If we want to store address of variable then we should use pointer.



- Initialization of pointer

```
int number = 10;
int *ptrNumber = &number;    //OK
```

- Uninitialized pointer is also called as "wild pointer".
- Assignment of pointer

```
int number = 10;    //Initialization
int *ptrNumber;    //Wild Pointer
ptrNumber = &number;    //OK
```

```
int num1 = 10;    //Initialization
int *p1;    //Pointer Declaration
int *p2 = &num2;    //Initialization of pointer
p1 = &num2;    //Assignment of pointer.
```

- NULL is a macro having 0 address.

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

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

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

- In above code ptr is NULL pointer.
- Process of accessing value of the variable using pointer is called dereferencing.

## Day 2

---

### Constant pointer combination

1. int \*ptr

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

```
int *ptr = NULL;    //ptr is NULL pointer
```

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

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

## 2 const int \*ptr

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

```
const int *ptr = NULL; //ptr is NULL pointer
const int num1 = 10;
ptr = &num1;
//*ptr = 50;    //Not Ok
printf("Num1    :      %d\n",*ptr); //dereferencing

const int num2 = 20;
ptr = &num2;
//*ptr = 60;    //Not Ok
printf("Num2    :      %d\n",*ptr); //dereferencing
```

## 3. int const \*ptr

- It is same as statement 2 i.e const int \*ptr is same as int const \*ptr.

```
const int *ptr = NULL; //ptr is NULL pointer
int const num1 = 10;

ptr = &num1;
//*ptr = 50;    //Not Ok
printf("Num1    :      %d\n",*ptr); //dereferencing

const int num2 = 20;
ptr = &num2;
//*ptr = 60;    //Not Ok
printf("Num2    :      %d\n",*ptr); //dereferencing
```

## 4. const int const \*ptr

- It is same as statement 2 and 3
- Warning : duplicate 'const' declaration specifier

```

const int const *ptr = NULL;    //ptr is NULL pointer
int const num1 = 10;

ptr = &num1;
// *ptr = 50;    //Not Ok
printf("Num1    :    %d\n", *ptr);    //dereferencing

const int num2 = 20;
ptr = &num2;
// *ptr = 60;    //Not Ok
printf("Num2    :    %d\n", *ptr);    //dereferencing

```

#### 5. int \*const ptr

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

```

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

int num2 = 20;
//ptr = &num2;    //Not OK

```

#### 6. int \*ptr const

- It is invalid syntax

#### 7. const int \*const ptr

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

```

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

const int num2 = 20;    //OK
//ptr = &num2;    //Not OK

```

#### 8. int const \*const ptr

- It is same as statement 7.

## OOPS Introduction

- Object Oriented Programming Structure/System
- OOPS is not a syntax or language. It is programming methodology that is used to solve real world problems.
- OOPS is designed by Dr Alan Kay in 1960.
- Grady Booch is inventor of UML.
- According to Grady Booch there are 4 major and 3 minor pillars/parts/elements of oops.
- 4 major pillars are:
  1. Abstraction : Simplicity
  2. Encapsulation : Data Hiding/security
  3. Modularity : Minimize module dependency
  4. Hierarchy : Reusability
- Here word major means, language without any one of the above feature can not be object oriented.
- 3 major pillars are:
  1. Typing : To reduce maintenance
  2. Concurrency : To utilize CPU efficiently
  3. Persistence : To maintain state of object on secondary storage.
- If language support to above features then it is considered as useful but not essential to classify language OO.

## **C++ History and ANSI Standards**

- C++ is developed by Bjarne Stroustrup at "AT&T" Bell Lab in 1979.
- Initial name of the language was "C With Classes" because C++ is derived from C and simula.
- In 1983, ANSI renamed it to C++.
- It is object oriented programming language.
- Extension of C++ source file must be .cpp
- Using C++, we can develop code in procedure as well as object oriented way hence it is also called as hybrid programming language.
- In old days, "cfront" was used to convert C++ source code into C source code.

## **C++ Standards**

1. Before 1998 : Annotated 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
- Reference
    1. [www.ansi.org](http://www.ansi.org)
    2. [cppreference.com](http://cppreference.com)

## **C++ Software development Kit**

- SDK = Lang tools + Documentation + Libraries + Runtime Env.
- Language Tools
  1. Editor
    - Notepad, gedit, textedit, VSCode
    - It is used to write/edit source code
  2. Preprocessor
    - cpp
    - Job preprocessor is
      1. To expand macros
      2. To remove comments
  3. Compiler
    - Microsoft VS : cl.exe
    - Linux : g++
    - Job of compiler
      1. To check syntax
      2. To convert high level code into low level(assembly)
  4. Assembler
    - Microsoft : masm
    - Linux : as
    - Job of assembler is to convert low level code into machine code
  5. Linker
    - Microsoft : link.exe
    - Linux : ld
    - Job of linker is to link almost executable(.o) to the library file.
  6. Loader
    - OS API.
    - Loader is responsible for loading executable in RAM.
  7. Debugger
    - Microsoft : windbg
    - Linux : gdb
    - It is used to detect bug.
  8. Profiler
    - Linux : valgrind
    - It is used to debug memory.
  - Documentation : Microsoft VS : MSDN Linux : man pages
  - Libraries : glibc.so
  - Runtime Environment : C++ Runtime.

## Data Types in C++

- Fundamental Data Types( 7 )
  1. void : Not Specified
  2. bool : 1 byte
  3. char : 1 byte
  4. wchar\_t : 2 bytes
  5. int : 4 bytes
  6. float : 4 bytes

- 7. double : 8 bytes
- Derived Data Type( 4 )
  - 1. Array
  - 2. Function
  - 3. Pointer
  - 4. Reference
- User Defined Data Type ( 3 )
  - 1. Union
  - 2. Structure
  - 3. Class

## Comments

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

## Structure

- It user defined( actually derived ) data type.
- If we want group related data elements together then we should use structure.
- struct is keyword in C/C++.
- Example 1

```
struct Date
{
    int day;
    int month;
    int year;
};
```

- Example 2

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

- We can declare structure inside function. It is called local structure. We can not declare pointer and object of local structure outside function.

```

struct Employee *ptr;    //Pointer      : Not Ok
struct Employee emp;    //object       : Not Object
int main( void )
{
    struct Employee //Local Structure
    {

    };
    struct Employee *ptr;    //Pointer      : Ok
    struct Employee emp;    //object       :      OK
    return 0;
}

```

- typedef is keyword in C/C++.
- If we want to give another name to the existing datatype then we should use typedef.
- size\_t is a typedef of unsigned int.

```
typedef unsigned int size_t;
```

- Example 1

```

struct Employee
{
    char name[ 30 ];
    int empid;
    float salary;
};
typedef struct Employee Employee_t;
struct Employee emp1;//OK
Employee_t emp2;//OK
struct Employee_t emp3;//Not OK

```

- Example 2

```

typedef struct Employee
{
    char name[ 30 ];
    int empid;
    float salary;
}Employee_t;//OK
struct Employee emp1;//OK
Employee_t emp2;//OK

```

- structure member get space inside object. Hence size of object depends on size of structure members.
- If language allows us to initialize object using initializer list then it is called aggregate object.

- In C language following types are aggregate types:
  1. Array
  2. Structure
  3. Union

```
struct Employee emp = { "Sandeep", 33, 45000 }; //Ok
//{ "Sandeep", 33, 45000 }; -> Initializer List
```

- Using object, if we want to access members of structure then we should use dot/member selection operator.

```
struct Employee emp = { "Sandeep", 33, 45000 };
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 operator.

```
struct Employee *ptr = ( struct Employee* )malloc( sizeof( struct Employee
) );
printf("Name      :      %s\n", ptr->name);
printf("Empid     :      %d\n", ptr->empid);
printf("Salary    :      %f\n", ptr->salary);
```

- Main function is designed to give call to the other functions hence it is called as calling function.
- If we want to use pointer or object of structure in multiple functions then we should declare structure globally.

```
struct Employee //Global Structure
{
    char name[ 30 ];          //30 bytes
    int empid;                //4 bytes
    float salary;             //4 bytes
};
int main( void )
{
    return 0;
}
```

## Access Specifier

- If we want to hide members of structure/class then we should access specifier. In other words access specifier is used to control visibility.



- Access specifiers in C++
  1. private ( - )
  2. protected ( # )
  3. public ( + )

## Class related concepts

### Data Member

- Variable declared inside class is called data member.
- Data member is also called as field/attribute/property.

```
class Employee
{
private:
    //Data Members
    char name[ 30 ];
    int empid;
    float salary;
};
```

### Member Function

- A function, which is implemented inside class is called member function.
- Member function is also called as method/operation/behavior/message.

```
class Employee
{
public:
    //Member Function
    void accept_record( void )
    { }
    void print_record( void )
    { }
};
```

### Class

- It is a collection of data member and member function
- By defining class we are achieving encapsulation.
- In C++, class can contain
  1. Nested Type( structure / class / enum )
  2. Data Member
  3. Member function
  4. Constructor and Destructor

## Object

- It is variable or instance of a class.
- Syntax

```
//class Employee emp;    //OK
Employee emp;    //OK
```

- Process of creating object from class is called instantiation.
- In above code, class Employee is instantiated where emp is an instance.
- During instantiation, use of class keyword is optional.
- With the help of instantiation, we achieve abstraction.
- We can instantiate concrete class but we can not instantiate abstract class.

## Message Passing

```
int main( void )
{
    Employee emp;
    emp.accept_record( );
    emp.print_record( );
    return 0;
}
```

- In above code, accept\_record() and print\_record() is called on object emp.
- This process of calling member function on object is called message passing.

```
int main( void )
{
    Employee emp;
    //emp.accept_record( );
    emp.Employee::accept_record( );

    //emp.print_record( );
    emp.Employee::print_record( );
    return 0;
}
```

## Day 3

---

### Coding Conventions

1. Hungarian Notation
  - Generally used for C/C++ code

2. Camel Case convention
3. Pascal case naming convention
  - camel and pascal case generally used for java and C#.

#### **Pascal Case coding/naming convention**

- Consider Example
  1. System
  2. StringBuffer
  3. NullPointerException
  4. IndexOutOfBoundsException
- In this convention, including first word, first character of each word must be in upper case.
- In C++, we should use it for:
  1. File Name
  2. Type Name[ union, struct, class ]

#### **Camel Case coding/naming convention**

- Consider Example
  1. main()
  2. parseInt()
  3. showInputDialog()
  4. waitForPendingFinalizer( )
- In this convention, except first word, first character of each word must be in upper case.
- In C++, we should use it for:
  1. Data member
  2. Member function
  3. Local and global variable

#### **Convention For Macro, constant and enumerator**

- Consider macro

```
#define NULL ((void*)0)
#define EOF -1
#define SIZE 5
```

- Consider constant

```
const int PI=3.14;
```

- consider enum

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

- Name of macro, constant and enumerator should be in uppercase.

### Convention for global function

- A function which is not a part of any function or structure/class then such function is called global function.
- main is a global function.
- consider example:
  1. main( )
  2. print\_record( )
  3. add\_number\_of\_days( int days )

### this pointer

- Member function do not get space inside object.
- Data member get space once per object and according to their order in class. Hence size of object depends on size of all the data members declared inside class.
- Local and global variable do not get space inside object. "Only data members get space inside object".
- In C++, we can allocate memory for object on stack segment, data segment as well as heap section.

```
Employee e1;//Global : Data Segment
int main( void )
{
    Employee e2;//Local : Stack Seg
    Employee *ptr = new Employee();//Dynamic : Heap Section
    return 0;
}
```

- If we want to perform operations on object then we should call member function on object.
- If we call member function on object then compiler implicitly pass address of that object as a argument to the function.

```
Employee e1,e2,e3;
e1.acceptRecord( );      //e1.acceptRecord(&e1);
e2.acceptRecord( );      //e1.acceptRecord(&e2);
e3.acceptRecord( );      //e1.acceptRecord(&e3);
```

- To store address of current object/calling object, compiler implicitly declare one pointer as a parameter inside member function. Such parameter is called "this" pointer.
- this is a keyword in C++.

- "this" is a local pointer( function parameter ) which available inside only non static member function of the class which always stores address of current object/calling object.
- "this" pointer is a constant pointer. Its general type is:

```
ClassName *const this;
```

- Using this pointer, member function can access data members inside member function hence it is considered as a connection/link between them.
- this pointer is considered as first parameter to the member function.

```
//void setName( Employee *const this, char name[] )
void setName( char name[] )
{
}
int main( void )
{
    Employee emp;
    emp.setName( "Rahul" ); //emp.setName( &emp, "Rahul" );
    return 0;
}
```

- We can use this pointer to access members of the class/structure only.
- Following function do not get this pointer
  1. global function
  2. static member function
  3. friend function
- To access members of the class, use of this keyword is optional.

```
//Employee *const this = &emp
void printRecord( void )
{
    printf("Name      :      %s\n",name);    //OK
    printf("Name      :      %s\n",this->name);    //OK
}
```

- Member function do not get space inside object rather all the objects of same class share single copy of member function.
- This sharing is done with the help of this pointer.

## Characteristics of object

1. State:
  - value stored inside object is called state of the object.
  - value of data member represent state of object.
2. Behavior:

- Set operations that we can perform on object is called behavior of the object.
- Member functions of the class represent behavior of object.

### 3. Identity:

- Value of any data member which is used to identify object uniquely is called identity.
- If state of objects are same then address can be considered as its identity.

## Class and Object

- Class
  1. Class is collection of data member and member function.
  2. Structure and behavior of object is depends on class hence class is considered as a template/model/blueprint for object.
  3. Class represent group/set of such objects which is having common structure and common behavior.
  4. Class is abstract/logical entity.
  5. e.g Car, Mobile Phone, Laptop,Room
  6. Implementation of class represents "Encapsulation".
- Object
  1. Object is variable or instance of a class.
  2. Any entity which is having physical existance / which get space inside memory is called object.
  3. An entity which is having state behavior and identity is called object.
  4. All noun enitities are objects.
  5. It is a physical entity.
  6. e.g Tata Nano, Nokia 1100, MacBook Air, Director Cabin etc.

## Empty class

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

```
class Test
{
    //Empty body
};
```

- Size of object is depends on size of data members declared inside class.
- According to above definition, size of object of empty class should be zero.
- According to oops concept class is logical and object is physical entity. Hence 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 technique it gets one byte space inside memory.

If we want to define member function globally then we should use following syntax:

```
ReturnType ClassName::FunctionName( ... );
```

## Header Guard:

- It is also called as include guard.
- Syntax: `#ifndef HEADER_H_ #define HEADER_H_ //TODO : Declaration #endif`
- If we want to expand/replace contents of header file only once then we should use header guard.

What is the difference between `#include<abc.h>` and `#include"abc.h"`

- If we include header file in `< >` bracket then preprocessor search that header file in standard directory only(`#include<abc.h>`)
- On linux `"/usr/include"` is a standard directory for header files.
- If we include header file in `" "` then preprocessor first search that header file in current working directory. If not found then it searches that header file in standard directory.
- Consider following statment
  1. `#include<stdio.h> //OK`
  2. `#include"stdio.h" //OK`
  3. `#include<Employee.h> //Not OK`
  4. `#include"Employee.h" //OK`
- In C/C++, we can use operators with variable/object of fundamental type only.

```
int x = 10, y = 20, z;  
z = x + y; //OK
```

```
Complex c1,c2,c3;  
c3 = c1 + c2; //Not OK
```

- If we want to use operator with objects then we must overload operator ( ref. operator overloading )
- If we pass/return object to/from function by value then compiler create its copy.

## Storage Classes

1. auto
  2. static
  3. register
  4. extern
- storage class decides scope and lifetime of variable/function.
  - Scope : It decides area/region in which we can access element.
  - Types of scope:
    1. Block scope
    2. Function scope
    3. Prototype scope
    4. Class scope
    5. Namespace scope
    6. File Scope
    7. Program Scope

```

int num1 = 10; //Program Scope
static int num2 = 20; //File Scope
namespace std
{
    int num3 = 30; //Namesapce scope
    class Test
    {
        int num4; //Class Scope
    public:
        void print( )
        {
            int num5 = 40; //Function Scope
            {
                int num6 = 60; //Block scope
            }
        }
    }
}


```

- Lifetime : It indicates duration/time i.e how long variable will be exist inside memory.
- Types of lifetime:
  1. Automatic Lifetime
  2. Static Lifetime
  3. Dynamic Lifetime
- If name of local and global variable is same then preference is given to the local variable.

```

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

```

- Using scope resolution( operator , we can access value of global variable.

```

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

    { //Start of new block
        int num1 = 30;
    }
}

```



```

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

```

```

int num1 = 10;
int num1 = 20; //redefinition of 'num1'
int main( void )
{
    return 0;
}

```

## Namespace

- It is C++ language feature which is used:
  1. To avoid name clashing / ambiguity / collision
  2. To group functionally equivalent / related types together.
- "namespace" is keyword in C++.
- We can not instantiate namespace.
- We can not define namespace inside function or class. Namespaces can only be defined in global or namespace scope.
- We can not define main function inside namespace. Linker will generate error.
- Namespace can contain
  1. variable
  2. function
  3. enum/union/struct/class
  4. namespace
- Naming convention:
  - Generally name of the namespace should be in lowercase.
  - eg. namespace std;
- If we want to access members of namespace then we should use :: operator.

```

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

```

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

```

namespace na
{
    int num1 = 10;
    int num3 = 30; //OK
}
namespace nb
{
    int num2 = 20;
    int num3 = 40; //OK
}
int main( void )
{
    printf("Num1 : %d\n",na::num1);//10
    printf("Num2 : %d\n",nb::num2);//20

    printf("Num3 : %d\n",na::num3);//30
    printf("Num3 : %d\n",nb::num3);//40
    return 0;
}

```

- We can give same name to the namespaces. If name of the namespaces are same then name of members must be different.

```

namespace na
{
    int num1 = 10; //Ok
    int num3 = 30; //OK
}
namespace na
{
    int num2 = 20; //OK
    int num3 = 40; //error: redefinition of 'num3'
}

```

- We can define namespace inside another namespace. It is called nested namespace.
- If we define any member globally without namespace then it is considered as a member of global namespace.

```

int num1 = 10; //Member of global namespace
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 want to access elements of namespace frequently then we should use using directive.

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

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

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

- In above code, preference will be given to local variable.

## Day 4

---

### Namespace

```
namespace na
{
    int number = 10;
}
using namespace na;
void show_record( )
```

```

{
    //printf("Number      :      %d\n", na::number);//OK

    //using namespace na;
    printf("Number :      %d\n", number);
}
void print_record( )
{
    //printf("Number      :      %d\n", na::number);//OK

    //using namespace na;
    printf("Number :      %d\n", number);
}
void display_record( )
{
    //printf("Number      :      %d\n", na::number);//OK

    //using namespace na;
    printf("Number :      %d\n", number);
}
int main( void )
{
    //show_record();          //OK
    ::show_record();          //OK
    ::print_record();
    ::display_record();
    return 0;
}

```

- In C++, we can define namespace without name. It is called unnamed namespace.

```

namespace    //Unnamed namespace
{
    int number = 10;
}
int main( void )
{
    printf("Number :      %d\n", number);
    return 0;
}

```

- How to create alias for namespace?

```

namespace na
{
    int num1 = 10;
}
namespace ntest = na;    //namespace alias

```

- "std" is standard namespace in C++. In other words, all the library defined types and functions are members of std namespace.

## Console Input and Output operation

- Console Input represents Keyboard
- Console Output represent Monitor and Printer
- Console = Keyboard + Monitor
- Stream is an abstraction ( object ) which is used to produce( write ) and consume( read ) information from source to destination.
- Stream is always associated with physical device.

## Standard Stream Objects of C

1. stdin
2. stdout
3. stderr.

## Standard Stream Objects of C++

1. std::cin : Keyboard
2. std::cout : Monitor
3. std::cerr : Error Stream
4. std::clog : Logger Stream

- std is namespace of C++ which is declared in iostream header file.
- Consider declaration of std namespace in header file: File Name : ( No extension )

```
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 declared in std namespace and std is a part of header file.

- Insertion operator( << ) is designed to use with cout object.
- If we want to redirect/display state of object on console( monitor ) then we should use cout object.

## Escape Sequence

- It is a character which is used to format the output.
- e.g : '\n', '\t', '\a', '\b', '\r', '\'
- We can use escape sequences in C++ but it is not recommended. In C++, we should use manipulator.

## Manipulator

- It is a function which is used to format the output.
- e.g : endl, setw, dec, oct, hex, left, right, fixed, scientific, setprecision etc.
- manipulators are declared in std namespace and in context of manipulator std is a part of header file.

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

    using namespace std;
    cout<<"Hello World!!!"<<endl;
    return 0;
}
```C++
int num1 = 10;
int num2 = 20;
using namespace std;
cout<<"Num1      :      "<<num1<<endl;
cout<<"Num2      :      "<<num2<<endl;
```

## Character Input( cin )

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

- cin is an external object of istream class.
- cin is declared inside std namespace and in context of console I/O std is a part of header file.
- Extraction operator( >> ) is designed to use with cin.
- If we want to retrieve/read state of object from console( keyboard ) then we should use cin object.

```
#include<iostream>
int main( void )
{
    int num1;
    using namespace std;
    cout<<"Num1      :      ";
```

```

    cin>>num1;
    cout<<"Num1      :      "<<num1<<endl;
    return 0;
}

```

- string is Standard Template Library(STL) class.

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

- On Mac OS size of string is 24 bytes whereas on Linux its size is 4/8 bytes ( depending on compiler ).

```

#include<iostream>
#include<string>
int main( void )
{
    using namespace std;
    string name;
    cout<<"Name      :      ";
    cin>>name;
    name = name + "Pune";
    cout<<"Name      :      "<<name<<endl;
    return 0;
}

```

## Constructor

- It is a member function of class that is used to initialize the object.
- Due to following reasons, constructor is special function of a class:

1. It's name is same as class name.
2. It doesn't have any return type
3. It is designed to call implicitly
4. In life time of the object it gets called only once.

Note : Constructor gets called once per object.

- In C++, on object, pointer or reference, we can not call constructor explicitly. It is designed to call implicitly.

```

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

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

    Complex &c2 = c1;          //OK
}

```

```

        //c2.Complex( );          //Not OK
        return 0;
    }

```

- We can not declare constructor static, constant, volatile or virtual. We can declare constructor "inline" only.
- We can use any access specifier on constructor.
- If constructor is public then we can create object of the class inside member function as well as non member function.
- If constructor is private then we can create object of the class inside member function and friend function only.
- Constructor calling sequence is depends on order of object declaration.
- Compiler do no call constructor on pointer and reference. Compiler call constructor on only object.

```

class Complex
{
private:
    int real;
    int imag;
public:
    Complex( void );
};
//Global definition of constructor
Complex::Complex( void )
{
    this->real = 0;
    this->imag = 0;
}

```

### Types of constructor( 3 )

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

#### Parameterless constructor

- It is also called as zero argument construtor / User Defined Default constructor.
- A constructor which do not take any parameter is called Parameterless constructor.

```

Complex( )
{
    this->real = 0;
    this->imag = 0;
}

```



- If we create object without passing argument ( i.e zero argument ) then parameterless constructor gets called.

```
Complex c1, c2, c3;
```

- In above statement, compiler will call parameterless constructor on c1, c2 and c3.

#### Parameterized constructor

- A constructor which take parameter is called parameterized constructor.
- If name of data member and local variable is same then use of this keyword to access data member is mandatory.

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

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

```
Complex c1( 10 ), c2, c3(10,20);
```

- In above statement constructor calling sequence is: c1 : single Parameter ctor c2 : parameterless ctor c3 : two Parameter ctor

#### Default constructor

- If we do not define constructor inside class then compiler provides one constructor for the class by default. It is called default constructor.
- Compiler do not generate default parameterized constructor i.e Default constructor is parameterless.
- If we want to create object by passing argument then we must define constructor inside class.
- For the C++ application developer implementation of default constructor is as follows:

```
Complex( void )
{
    //Empty body
}
```

- Compiler provided default constructor do not initialize data member declared by the programmer. It is used to initialize data members declared by the compiler.
- eg. v-ptr, vb-ptr etc.

- In C++98 and C++03, we can not call constructor from another constructor. It constructor chaining is not allowed.
- In C++11, we can call constructor from another constructor. It is called constructor delegation.

### Constructor Member Initializer List:

- If we want to initialize object according to order of data member declaration then we should use constructor member initializer list.
- We can not initialize Array and C Strings( Array of chars ) in ctor's member initializer list.

```
class Test
{
private:
    int x;
    int y;
    int z;
public:
    Test( void )
    : x( 10 ), y( 20 ), z( 30 )
    {
    }
    Test( int x, int y, int z )
    : x( x ), y( y ), z( z )
    {
    }
    void printRecord( void )
    {
    }
};
```

- In case of modular approach, constructors member initializer list must appear in definition part(.cpp) only.

### Constant in C++

- const is qualifier in C/C++
- If we dont want to modify state/value of object then we should use const keyword.

```
const int num1;           //OK : In C
const int num2 = 20;      //OK : In C
```

```
const int num1;           //Not OK : In C++
const int num2 = 20;      //OK : In C++
```

- In C++, using pointer, we can not modify state const variable/object hence initializing constant variable is mandatory.
- 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 data member is constant then we must 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 data member get space inside object.
- If we dont want to modify state of only current object inside member function then we should declare member function constant.
- On non constant object,we can call constant as well as non constant member function.
- Non constant member function get this pointer of following type:

```
ClassName *const this;
```

- Constant member function get this pointer of following type:

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

- Volatile member function get this pointer of following type:

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

- If member function is constant then local variables are not considered as constant.
- We can not declare following functions constant:
  1. Constructor
  2. Destructor
  3. Static Member Function
  4. Global function( e.g main )

Note : We should declare read-only function constant(getter, printRecord etc).

- "mutable" is keyword in C++.
- Inside, constant member function, if we want to modify state of non constant data member then we should declare non constant data member mutable.
- We can use mutable keyword on data member only.
- In case of modular approach, const keyword must appear in delclaration as well as definition.

# Day 5

---

## 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.
- We can declare constant object but we can not declare constant class.

```
Test t1;
const Test t2;
```

## Macro

- Consider example

```
#define SIZE    5
#define EOF     -1
#define NULL    ((void*)0)
```

- Symbolic constant is called as macro.
- C/C++ preprocessor(cpp) is a tool which is used to process macro.
- Job Of Preprocessor
  1. To expand Macros( Find and replace)
  2. To remove the comments.
- File Name : Main.cpp

```
#include<iostream>
using namespace std;
#define message "Good Morning"
//Program : Test macro
//Author   : C++ Batch Students
//Date     : 20 Dec,2019
//main     : entry point function
int main( void )
{
    cout<<message<<endl;
    return 0;    //Exit from main function
} //End of main function.

* Command to create ".i" file
  g++ -E Main.cpp -o Main.i
* Advantages of macro
1. It improves readability of source code.
```

```

2. It reduces maintainance of code
3. It improves performance of the application.
* Disadvantages of macro
1. Macros are not type safe
2. It increases code size
3. We can not debug macros.
* Example of macros
  1. NULL
  2. EOF
  3. __FILE__
  4. __LINE__
  5. __DATE__
  6. __TIME__
  7. __FUNCTION__
```C++
int main( void )
{
    cout<<EOF<<endl;        //-1
    cout<<NULL<<endl;        //0
    cout<<__FILE__<<endl;    //../src/Main.cpp
    cout<<__FUNCTION__<<endl;    //main
    cout<<__LINE__<<endl;    //38
    cout<<__DATE__<<endl;    //Dec 20 2019
    cout<<__TIME__<<endl;    //08:37:57
    return 0;
}

```

## Inline Function

- An ability of operating system to execute single process at a time is called single tasking.
- MS DOS is single tasking os.
- An ability of operating system to execute multiple process at a time is called multitasking.
- We can achive multitasking using:
  1. process
  2. thread
- Process:
  1. Program in execution is called as process
  2. Running instance of a program is called process.
    - Process is also called as task.
- Thread:
  1. Sub process / light weight process is called thread.
  2. Thread is separete path of execution which runs independantly.
    - Every process is by default single threaded.
    - Per thread, operating maintains one stack, it is called runtime stack. This runtime stack is collection of stack frame.
    - Stack frame contains information of called function:
      1. Function Parameters
      2. Local Variable
      3. Function Call

4. Tempory storage for inmediate calculation

5. Return Address.

◦ Above information is also called as Function Activation Record( FAR ).

- If we call any function then its information is managed in runtime stack. Managing runtime stack is heavy process hence giving call to the function is overhead to the compiler.
- If we want to avoid this overhead then we should declare function inline.
- "inline" is keyword in C++.

```
inline int max( int a, int b )
{
    return a > b ? a : b;
}
int main( void )
{
    int result = max( 10, 20 );
    //int result = 10 > 20 ? 10 : 20;
    cout<<"Result    :    "<<result<<endl;
    return 0;
}
```

- If function is inline then compiler do not call that function rather it replaces function call by function body.
- inline is request to the compiler.
- In following condition function can not be considered as inline.
  1. If we implement function using recursion
  2. If we implement function using loop
  3. If we use jump statment inside function
  4. Function is which is taking help of another function.
- We can use inline keyword with member function as well as global function.
- If we define member function inside class then it is by default inline.
- If we define member function globally then we should provide inline keyword explicitly.
- We can not seperate inline function code into multiple files.

## Typedef

- If we want to give short name or meaningful name to the existing data type then we should use typedef keyword.
- Using typedef we can not create/define new data type rather we create alias for existing data type.
- Example typedef unsigned int size\_t; typedef unsigned short wchar\_t; typedef basic\_string string; typedef basic\_istream istream; typedef basic\_ostream ostream;

```
typedef int* IntPtr;
typedef struct Complex
{
    int real;
    int imag;
} Complex_t;
```

```
struct Complex c1;
Complex_t c2;
```

## Reference

```
int num1 = 10; //Referent variable
int &num2 = num1; //Reference Variable
```

- Reference is alias or another name given to the existing memory location.
- Using typedef we can create alias to existing datatype( structure/ class ) whereas using reference we can create alias to existing object( instance / variable ).
- We can not declare reference for class or literals/constant.
- If we initialize reference then we can not change its referent.

```
int num1 = 10;
int num2 = 20;
int &num3 = num1;
num3 = num2;
++ num3;
//num1 : 21
//num2 : 20
//num3 : 21
```

- We can not create reference to constant/literal. We can create reference to object only.

```
int num1 = 10; //Ok
int &num2 = 10; //Not OK
int &num3 = num1; //OK
```

- Reference is automatically dereferenced constant pointer variable.

```
int num1 = 10;
int &num2 = num1;
//int *const num2 = &num1;
cout<<"Num2 : "<<num2<<endl;
//cout<<"Num2 : "<<*num2<<endl;
```

- If we want to minimize complexity of pointer then we should use reference.
- In C++, we can pass argument to the function using three ways:
  1. By Value
  2. By Address
  3. By Reference

- Passing arguments by reference

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

- Array of references

```
int& arr[ 3 ];    //Not OK
```

- Reference to array

```
int arr1[ 3 ] = { 10, 20, 30 };
int (&arr2)[ 3 ] = arr1;    //OK
```

- We should not return local variable from function by reference or address. If we want to return it by address/reference then we should declare it static.

## Exception Handling

- Exception is an object that is used to send notification to the end user of the system if any exceptional situation occurs in the program.
- We should handle exception:
  1. To avoid resource leakage
  2. To handle all the runtime errors at single place so that we can reduce maintenance of application.
- Following are OS resources:
  1. Memory
  2. Thread
  3. File
  4. Socket
  5. I/O Devices
- If we want to handle exception then we should use 3 keywords:



1. try
2. catch
3. throw

## try

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

## throw

- It is a keyword in C++.
- Exception can be generated implicitly / explicitly. To generate exception, we should use throw keyword.
- throw statement is jump statement.

## Catch

- It is a keyword in C++.
- If we want to handle exception then we should use catch block/handler.
- Catch block can handle exception thrown from associated try block.
- Catch block must appear after try block.
- For single try block we, can write multiple catch block.
- For throw exception, if matching catch block is not available then C++ runtime give call to the `std::terminate()` function. Which implicitly call `std::abort` function.
- A catch block, which can handle any type of exception is called generic catch block.

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

- Generic catch block must appear after all specific catch block.
- In C++, we can create object without name. It is called anonymous object;

```
string ex("Welcome");
string("Welcome");//Anonymous object
```

- We can write try-catch block inside another try as well as catch block. It is called nested try catch block.

```
try
{
    try
    {
        A a("message");
    }
}
```

```

        throw a;
    }
    catch( A &a )
    {
    }
}
catch( A &a)
{
}
//Output : Inner catch block

```

```

try
{
    try
    {
        A a("message");
        throw a;
    }
    catch( B &b )
    {
    }
}
catch( A &a)
{
}
//Output : Outer catch block

```

```

try
{
    try
    {
        A a("message");
        throw a;
    }
    catch( B &b )
    {
    }
}
catch( C &c)
{
}
//Output : std::terminate()

```

```

try
{
    try
    {
        A a("message");
        throw a;
    }
    catch( B &b )
    {
    }
}

```

```
catch( ... )
{
}
//Output : Generic catch block
```

- Outer catch block can handle exception thrown from inner as well as outer try block.

```
try
{
    A a("message");
    throw a;
    try
    {
    }
    catch( A &a )
    {
    }
}
catch( A &a)
{
}
//Output : Outer Catch
```

- Inner catch block can handle exception thrown from inner try block only.

```
try
{
    try
    {
        A a("message");
        throw a;
    }
    catch( A &a )
    {
        throw; //rethrow
        //throw a;
    }
}
catch( A &a)
{
}
//Output : Outer catch
```

- If information which is required to handle exception is incomplete then we should rethrow exception.

## Day 6

---

### Exception Specification List.

- We can throw exception from try block as well as catch block. If we want to throw exception from outside try block then we should use function.

```

try
{
    int num1;
    ::accept_record( num1 );
    int num2;
    ::accept_record(num2);
    int result = ::calculate( num1, num2 );
    ::print_record( result );
}
catch( ArithmeticException &ex )
{
    cout<<ex.getMessage()<<endl;
}

```

- If we are throwing exception from outside of try block then for the purpose of documentation we should specify exception specification list.

```

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

```

- In case of modular approach, exception specification list should appear in declaration.
- If type of thrown exception is not available in exception specification list then During failure, C++ runtime do not handle exception in catch block rather it gives call to the std::unexpected function.

## Stack Unwinding

- It is process of destroying F.A.R( Stack Frame ) if exception is raised.

Following functions are frequently used to perform operations on object.

### 1. 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(), setImag() etc.

### 2. Inspector Function

- A member function of a class, which is used to read state of the object is called inspector/selector/getter function.
- e.g getReal(), getImag( )

### 3. Facilitator Function

- A function which provide facility of Console/File/ DB input/output operation is called facilitator function.
- e.g accept\_Record(), print\_record( )

#### 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 object.

### Function Overloading

- It is oops concept.
- C is a procedure oriented programming language. POP languages do not allow us to give same name to the multiple function.

```
int sum( int num1, int num2 )
{
    return num1 + num2;
}
int result = sum( 10,20 ); //Fun Call

//Not allowed in C
double sum( int num1, float num2, double num3 )
{
    return num1 + num2 + num3;
}
int result = sum( 10,20.1f, 30.2 );
```

- C++ is object oriented programming language. OOP languages allow us to give same name to the multiple function.
  - If implementation of a function is logically same/equivalent then we should give same name to the function.
  - If we want to give same name to the function then we must follow some rules.
1. If we want to give same name to the function and if type of all the parameters are same then number of parameters passed to the function must be different.

```
void sum( int num1, int num2 );//Ok
void sum( int num3, int num4 );//Not Ok
void sum( int num1, int num2, int num3 );//OK
```

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

```
void sum( int num1, int num2 );    //OK
void sum( int num1, double num2 ) //Ok
```

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

```
void sum( int num1, float num2 );
void sum( float num1, int num2 );
```

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

```
void sum( int num1, int num2 ); //OK
int sum( int num1, int num2 ); //Not OK
```

- Process of writing multiple functions using above rules is called function overloading.
  - Process of writing function with same name and different signature is called function overloading.
- Functions which are taking part in function overloading are called overloaded function.

```
void sum( int n1, int n2 );
void sum( double n1, double n2 );
void sum( Complex c1, Comple c2 );
```

- In above code function sum is overloaded.
- We can not overload main() function and destructor in C++.
- For function overloading, function must be exist in same scope.
- Return Type is not considered in function overloading.
- Consider function in C

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

- Consider function call

```
int result = sum( 10, 20 ); //OK

sum( 100, 200 ); //OK
```

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

## Name mangling and mangled name

- If we define function in C++, then compiler generates unique name by looking toward function name and type of parameter passed to the function. It is called mangled name

```
void sum( int n1, int n2 );           //__Z3sumii
void sum( int n1, double n2 );       //__Z3sumid
void sum( int n1, float n2, double n3 ); //__Z3sumifd
```

- Since ANSI has not defined any specification on mangled name, it( mangled name ) may vary from compiler to compiler.
- Process / algorithm which decides mangled name for the function is called name mangling.
- If we want to view mangled name then we should use nm tool.
- Command : [ nm a.out ]
- In C++, we can overload constructor but we can not overload destructor.
- If we want to use function implemented in C source file(.c file) into C++ source code(.cpp file) then we should declare C functions extern "C".

```
//extern "C" int sum( int num1, int num2 );
//extern "C" int sub( int num1, int num2 );

extern "C"
{
    int sum( int num1, int num2 );
    int sub( int num1, int num2 );
}
```

- If we declare function extern "C" then compiler do not generate mangled name for it.
- Consider following Code

```
void print( int number )
{
    cout<<"int : "<<number<<endl;
}
void print( float number )
{
    cout<<"float : "<<number<<endl;
}
```

```
print( 10 );    //int : 10
```

```
print( 10.5 ); //error: call to 'print' is ambiguous
```

```
print( 10.5f ); //float : 10.5
```

```
print( ( int )10.5 ); //int : 10
```

## Default argument

- In C++, To reduce programmers effort, we can assign default value to the parameter of function. Such default value is called default argument and parameter is called optional parameter.

```
//Optional Parameters : num3, num4, num5
//Default argument : 0
void sum(int num1, int num2, int num3=0, int num4=0, int num5=0)
{
    int result = num1 + num2 + num3 + num4 + num5;
    cout<<"Result : " << result << endl;
}
```

- Default arguments are always assigned from right to left direction.
- In case of modular approach, default argument must appear in declaration part only.

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

- We can assign default arguments to the parameters of any global function as well as member function.

```
class Complex
{
private:
    int real;
    int imag;
public:
    Complex( int real = 0, int imag = 0 );
};
Complex::Complex( int real, int imag ) : real( real ), imag( imag )
{
}
```



```
int main( void )
{
    Complex c1(10,20);    //10,20
    Complex c2(30); //30,0
    Complex c3;    //0,0
    return 0;
}
```

## Day 7

---

### void pointer

- A pointer, which can store address of any type of variable is called void pointer.

```
void *ptr = NULL;

char ch1 = 'A';
char *p1 = &ch1;
ptr = &ch1; //OK

int num1 = 10;
int *p2 = &num1;
ptr = &num1;    //OK

double num2 = 20.5;
double *p3 = &num2;
ptr = &num2;    //OK

Complex c1;
Complex *p4 = &c1;
ptr = &c1;    //OK
```

- Since void pointer can store address of any type of variable it is also called as generic pointer.
- Using void pointer, we can not do dereferencing.

```
int main( void )
{
    int number = 10;
    void *ptr1 = &number;    //OK
    //cout<<"Number : " << *ptr1 << endl; //Not OK
    int *ptr2 = ( int* )ptr1;
    cout<<"Number : " << *ptr2 << endl; // OK
    return 0;
}
```

### Dynamic Memory Management in C

- In C language, if we want to manage memory dynamically then we should use functions declared in header file.
- Following are the functions declared in header file.
  1. void\* malloc( size\_t size );
  2. void\* calloc( size\_t count, size\_t size );
  3. void\* realloc( void \*ptr, size\_t newSize );
  4. void free( void \*ptr );
- malloc(), calloc() and realloc() functions are used to allocate memory whereas free() function is used to deallocate memory.
- typedef unsigned int size\_t;
- Everything on heap section is anonymous.

## malloc

- It is a function declared in header file.
- Syntax: void\* malloc( size\_t size );
- Using malloc() function, we can allocate space for single variable as well as array but it is designed to allocate memory for single variable only.
- If we reserve space using malloc then memory gets initialized with garbage value.
- Using malloc, we can reserve space only on heap section.
- If malloc fails to allocate memory then it returns NULL.

## calloc

- It is a function declared in header file.
- Syntax: void\* calloc( size\_t count, size\_t size );
- Using calloc() function, we can allocate space for single variable as well as array but it is designed to allocate memory for array.
- If we reserve space using calloc then memory gets initialized with zero.
- Using calloc, we can reserve space only on heap section.
- If calloc fails to allocate memory then it returns NULL.

## Multidimensional Array

- Array of array is called multi dimensional array.
- Consider following code for Multidimensional array

```
int **ptr = ( int** )calloc( 3, sizeof( int* ) );
for( int index = 0; index < 3; ++ index )
    ptr[ index ] = ( int* )calloc( 4, sizeof( int ) );

//TODO : Input and Output operation

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

## realloc

- It is a function declared in header file.
- Syntax: `void* realloc( void *ptr, size_t newSize );`
- It can be used to resize/reallocate memory.
- If realloc function fails to allocate memory then it returns NULL.

```
int *ptr = ( int* )realloc( NULL, sizeof( int ) );
```

- above code is equivalent to

```
int *ptr = ( int* )malloc( sizeof( int ) );
```

## free

- It is a function declared in header file.
- Syntax: `void free( void *ptr );`
- Using free function, we can deallocate memory which is allocated on heap section.

```
int *ptr = NULL;  
free( ptr );
```

- If ptr is a NULL pointer, no operation is performed.

## Dangling pointer

- A pointer, which contains address of deallocated memory then such pointer is called dangling pointer.

```
int *ptr1 = ( int* )malloc( sizeof( int ) );  
*ptr1 = 125;  
int *ptr2 = ptr1;  
cout<<"Value      :   "<<*ptr2<<endl;  
free( ptr2 );  
ptr2 = NULL;  
//ptr1 -> Dangling pointer
```

- To avoid dangling pointer, we should store NULL value inside pointer.

## Memory Leakage

- we allocate space inside memory but loose a way to reach to that memory then it is called memory leakage.

```
int *ptr = ( int* )calloc( 3, sizeof( int) );
ptr[ 0 ] = 10;
ptr[ 1 ] = 20;
ptr[ 2 ] = 30;
ptr = ( int* )calloc( 5, sizeof( int) );
```

- If we want to detect memory leakage then we should use "valgrind" tool.
- To avoid memory leakage, we should use free() function.

## Dynamic Memory Management in C++

- operator new(), operator new, operator delete(), operator delete are functions declared in std namespace and std namespace is declared in header file.
- consider declaration of std namespace

```
File Name : <new>
namespace std
{
    struct nothrow_t {};
    extern const nothrow_t nothrow;
    class bad_alloc : public exception
    {
    public:
        bad_alloc();
        virtual const char* what() const throw( );
    };
    void* operator new( size_t size )throw( bad_alloc );
    void* operator new[]( size_t size )throw( bad_alloc );
    void* operator new( size_t size, nothrow_t )throw( );
    void* operator new[]( size_t size, nothrow_t )throw( );
    void* operator new[]( size_t size, void* ptr )throw( );
    void operator delete( void *ptr )throw( );
    void operator delete[]( void* ptr )throw( );
}
```

- In C++, using new operator, we can allocate memory and using delete operator, we can deallocate memory.
- Using new operator we can allocate space on stack section, data section as well as heap section. It is also called as Free Store.
- If new operator fails to allocate memory then it throw std::bad\_alloc exception.
- Consider code in C:

```
//Memory allocation
int *ptr = (int*)malloc( sizeof( int ) );

//Memory deallocation
free( ptr );
```

- Consider code in C++:

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

//Memory deallocation
delete ptr;
//operator delete( ptr );
```

- What is the difference in following statements:
  - int \*ptr = new int;
    - Here we are allocating memory for single variable but memory will be initialized with garbage value.
  - int \*ptr = new int();
    - Here we are allocating memory for single variable but memory will be initialized with zero.
  - int \*ptr = new int(3);
    - Here we are allocating memory for single variable but memory will be initialized with value 3.
- Consider code in C:

```
//Memory allocation
int *ptr = (int*)malloc( 3 * sizeof( int ) );

//Memory deallocation
free( ptr );
```

- Consider code in C++:

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

//Memory deallocation
delete[] ptr;
//operator delete[]( ptr )
```

- Dynamic memory allocation and deallocation for Multidimensional array.

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

//TODO : Input and Output operation
```

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

- nothrow is object of nothrow\_t structure which is empty structure.
- If we use nothrow object with new operator then during failure it doesn't throw exception rather it returns NULL.

```
int main( void )
{
    try
    {
        int count = 1000000000000;
        int *ptr = new ( nothrow )int[ count ];
        if( ptr != NULL )
            cout<<"Ptr      :      "<<ptr<<endl;
        else
            cout<<"Ptr      :      NULL"<<endl;
    }
    catch( bad_alloc &ex )
    {
        cout<<ex.what()<<endl;
    }
    //Output : NULL
    return 0;
}
```

- If create object using malloc then constructor do not call.

```
Complex *ptr = ( Complex*) malloc( sizeof( Complex ) );
```

- If create object using new then constructor gets call.

```
Complex *ptr = new Complex; //OK    -> Complex::Complex();
//is equivalent to
Complex *ptr = new Complex(); //OK  -> Complex::Complex();
```

## Destructor

- It is a member function of a class which is used to deinitialize the object.
- Constructor calling sequence depends on order of object declaration but destructor calling sequence is exactly opposite.
- We can declare destructor inline and virtual only.

- Due to following reasons destructor is considered as special function of the class.
  1. It doesn't have any return type.
  2. It doesn't take parameter
  3. It is designed to call implicitly.
- If we do not define destructor inside class then compiler generates default destructor for class.

## Shallow Copy

- It is process of copying state of object into another object as it is.
- Consider following code:

```
Complex c1(10,20), c2;
c2 = c1;           //Assignment -> Shallow Copy
c2.printRecord();  //10,20
```

- Consider following code:

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

- Shallow copy is also called as bitwise copy.

## Deep Copy

- By modifying some state, if we create copy of the object then it is called deep copy.
- It is also called as memberwise copy.
- Conditions to create deep copy:
  1. Class must contain at least one pointer type data member
  2. Class must contain user defined destructor
  3. We must Create copy of the object.
- In following cases, copy of the object gets created
  1. Passing object to the function by value.
  2. Return object from function by value
  3. Initialization of object( Complex c2 = c1).
  4. Assignment of object( c2 = c1 ).
  5. By throwing object.
  6. Catching object from function by value.
- Steps to create deep copy
  1. Copy the required size from source object into destination object.
  2. Allocate new resource( memory) for pointer data member of destination object.
  3. Copy the contents from resource of source object into resource of destination object.
- Location to create deep copy
  1. In case of assignment, we should create deep copy inside assignment operator function.

2. In rest of the condition, we should create deep inside copy constructor.

```
//Array *const this = &a2
//const Array &other = a1
Array( const Array &other )
{
    //Step 1 : Copy required size
    this->size = other.size;
    //Step 2 : allocate new resource
    this->arr = new int[ this->size ];
    //Step 3 : Copy the contents
    for( int index = 0; index < this->size; ++ index )
        this->arr[ index ] = other.arr[ index ] ;
}
```

## Copy Constructor

- It is a parameterized constructor of a class which take single parameter of same type but as a reference.
- Job of ctor is to initialize the object. Job of dtor is to deinitialize the object but job copy ctor is to initialize newly created object from existing object.
- Syntax:

```
class ClassName
{
public:
    //ClassName *const this = address of destination object
    //const ClassName &other = reference of source object
    ClassName( const ClassName &other )
    {
        //TODO : Shallow / Deep Copy
    }
};
```

- Implementation 1

```
//Complex *const this = &c2
//const Complex &other = c1
Complex( const Complex &other ) //Copy Constructor
{
    //Shallow Copy
    this->real = other.real;
    this->imag = other.imag;
}
```

- Implementation 2



```
//Complex *const this = &c2
//const Complex &other = c1
Complex( const Complex &other )
    : real( other.real ), imag( other.imag )
{ }
```

- void\* memcpy(void \* dst, const void \* src, size\_t size );
- The memcpy() function copies "size" bytes from memory area src to memory area dst.
- Implementation 3

```
//Complex *const this = &c2
//const Complex &other = c1
Complex( const Complex &other )
{
    memcpy(this, &other, sizeof( Complex ) );
}
```

- If we do not define copy constructor inside class then compiler generates default copy constructor for class. By default it creates shallow copy.
- If we want shallow copy of object then no need to define copy constructor inside class.
- During initialization, if we want deep copy of object then we should define copy constructor inside class.
- Copy constructor gets called in following conditions:
  1. If we pass object 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 on anonymous object copy constructor gets called.
  3. If we initialize object from existing object of same class then on destination object copy constructor gets called.
  4. If we throw object then its copy get created on stack frame. On that instance copy constructor gets called.
  5. If we catch object by value then on catching object copy constructor gets called.
- Since creating copy is expensive, we should avoid copy. To avoid copy we should use reference.
- Consider following code:

```
class Complex
{
private:
    int real;
    int imag;
public:
    Complex( void )
    {
        cout<<"Complex( void )"<<endl;
        this->real = 0;
        this->imag = 0;
    }
    Complex( const Complex &other )
```

```

    {
        cout<<"Complex( const Complex &other )" << endl;
        this->real = other.real;
        this->imag = other.imag;
    }
    Complex( int real, int imag )
    {
        cout<<"Complex( int real, int imag )" << endl;
        this->real = real;
        this->imag = imag;
    }
    void printRecord( void )
    {
        cout<<"Real Number      :      " << this->real << endl;
        cout<<"Imag Number       :      " << this->imag << endl;
    }
};

```

```

Complex c1; //Complex::Complex( )
Complex *ptr = &c1;

```

- In above code on c1 parameterless ctor will call. Compiler do not call any constructor on pointer.

```

Complex c1; //Complex::Complex( )
Complex &c2 = c1;

```

- In above code on c1 parameterless ctor will call. Compiler do not call any constructor on reference.

```

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

```

In above code, on c1 parameterized ctor will and on c2 parameterless constructor will call. In case of assignment assignment operator function( operator=() ) gets called.

```

Complex c1(10,20);      //Complex::Complex( int, int );
Complex c2 = c1;        //Complex::Complex( const Complex &other )
Complex c3( c1 );       //Complex::Complex( const Complex &other )

```

- In above code, on c1 parameterized ctor will and on c2 & c3 copy constructor will call.

## Friend Function

- We can access private and protected members outside class using

1. Member Function( getter and setter )
  2. Friend function/class
  3. Pointer
- Friend function is non member function of a class which is designed to access private and protected members of class.
  - friend is keyword in C++.
  - main function is global function i.e non member function of class. We can declare main function friend inside class.
  - We can write, friend declaration statement inside any section of the class(private/protected/public).
  - If we declare function friend then it is not considered as a member of class.

```

class Test
{
private:
    int num1;
protected:
    int num2;
public:
    Test( void )
    {
        this->num1 = 10;
        this->num2 = 20;
    }
    friend void print( void );
};

void print( void )
{
    Test t;
    cout<<"Num1      :      "<<t.num1<<endl;
    cout<<"Num2      :      "<<t.num2<<endl;
}

int main( void )
{
    //Test t;
    //t.print();    //Not OK
    print( );      //OK
    return 0;
}

```

- From above code, we can conclude that, we can declare global function friend.

```

class A
{
public:
    void sum( void );
};

class B
{
private :

```

```

        int num1;
        int num2;
public:
    B( void );
    friend void A::sum( void );
};
B::B( void )
{
    this->num1 = 10;
    this->num2 = 20;
}
void A::sum( void )
{
    B b;
    int result = b.num1 + b.num2;
    cout<<"Result    :    "<<result<<endl;
}
int main( void )
{
    A a;
    a.sum( );
    return 0;
}

```

- From above code, we can conclude that, we can declare member function as a friend inside another class.

```

class A
{
public:
    void sum( void );
    void sub( void );
    void multiplication( void );
};
class B
{
private :
    int num1;
    int num2;
public:
    B( void );
    /* friend void A::sum( void );
    friend void A::sub( void );
    friend void A::multiplication( void ); */
    friend class A;
};

```

- If we want to access private and protected members inside some of the member function of another class then we should declare member function friend. But if we want to access private and protected members inside all of the member functions of another class then we should declare class friend.

```

class Complex
{
private:
    Complex( void )
    {
    }
};
int main( void )
{
    Complex c1;//error: calling a private constructor of class
    'Complex'
    return 0;
}

```

- In above code, If main function is friend of class Complex then we can create object inside main function.

```

class Complex
{
private:
    Complex( void )
    {
    }
    friend int main( void );
};
int main( void )
{
    Complex c1;//OK
    return 0;
}

```

- We can declare mutual friend classes but we can not declare mutual friend functions.
- If we want to declare any class/function as a friend inside another class then class and function must be exist inside same namespace.
- Friend function/class is C++ language feature which do no violate oops principle.

## Nested class

- In C++, we can define class inside scope of another class. It is called nested class.
- Nested class represent encapsulation.

```

class Outer //Top-Level class
{
public:
    class Inner //Nested class
    {
    };
};

```

- We can use any access specifier on nested class but we can not use it on top level class.

```
public class Outer //Not OK
{
}
```

- Instantiation

```
Outer out;
Outer::Inner in;
```

- Using object, if we want to access private and protected members of nested class inside member function of top level class then we should declare top level class friend inside nested class.

```
class Outer
{
private:
    int num1;
public:
    // Start of nested class
    class Inner
    {
private:
        int num2;
public:
        Inner( void )
        {
            this->num2 = 20;
        }
        friend class Outer;
    }; //end of nested class
public:
    Outer( void )
    {
        this->num1 = 10;
    }
    void printRecord( void )
    {
        cout<<"Num1      :      "<<this->num1<<endl;
        Inner in;
        cout<<"Num2      :      "<<in.num2<<endl;
    }
};
```

- Using object, if we want to access private and protected members of top level class inside member function of nested class then friend statement is not required.

```

class Outer
{
private:
    int num1;
public:
    Outer( void )
    {
        this->num1 = 10;
    }
public:
    class Inner
    {
private:
        int num2;
public:
        Inner( void )
        {
            this->num2 = 20;
        }
        void printRecord( void )
        {
            Outer out;
            cout<<"Num1      :      "<<out.num1<<endl;
            cout<<"Num2      :      "<<this->num2<<endl;
        }
    };
};

```

## Local class

- We can define class inside function. It is called local class.

```

int main( void )
{
    class Complex    //Local class
    {
        //TODO : Declaration
    };
    return 0;
}

```

- We can not use object/pointer/reference of local class outside function.

## Operator Overloading

- operator is keyword in C++ and operator is token in C/C++.
- If we want to form any expression then we should use operator

```
int a = 10, b = 20, c;
c = a + b;
```

- Types of operator
  - Unary Operator
    - If operator require single operand then it is called unary operator.
    - e.g ++, --, &, \*, sizeof, typeid, ~, ! etc
  - Binary Operator
    - If operator require two operands then it is called binary operator.
    - Types of binary operator
      - Arithmetic Operator( +, -, \*, /, %)
      - Relational Operator( <, >, <=, >=, ==, !=)
      - Logical Operator( &&, || )
      - Bitwise Operator( &, |, ^, <<, >>)
      - Assignment Operator( =, +=, -= etc )
  - Ternary operator
    - If operator require three operands then it is called ternary operator.
    - e.g conditional operator( ? : )

```
int num1 = 10;
int num2 = 20;
int result = num1 + num2;    //OK
```

- In C/C++, we can use operator with variable/object of fundamental type directly.

```
struct Point
{
    int xPosition;
    int yPosition;
};
struct Point pt1 = { 10, 20 };
struct Point pt2 = { 10, 20 };
struct Point pt3;
pt3 = pt1 + pt2;    //Not OK
pt3.xPosition = pt1.xPosition + pt2.xPosition;    //OK
pt3.yPosition = pt1.yPosition + pt2.yPosition;    //OK
```

- In C language, we can not use operator with objects of user defined type directly/indirectly.

```
class Complex
{
private:
    int real;
    int imag;
```



```

public:
    Complex( void )
    {
        this->real = 10;
        this->imag = 20;
    }
    Complex( int real, int imag )
    {
        this->real = real;
        this->imag = imag;
    }
};
Complex c1( 10,20 );
Complex c2( 30,40 );
Complex c3 = c1 + c2;    //Not Ok : Directly

```

- In C++, we can not use operator with the 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 it is necessary to define operator function.
- Method 1

```

class Complex
{
private:
    int real;
    int imag;
public:
    //Complex *const this = &c1
    //Complex &other = c2
    Complex operator+( Complex &other )
    {
        Complex temp;
        temp.real = this->real + other.real;
        temp.imag = this->imag + other.imag;
        return temp;
    }
};
int main( void )
{
    Complex c1( 10, 20 );
    Complex c2( 30, 40 );
    Complex c3;
    c3 = c1 + c2;    //c3 = c1.operator+( c2 )
    return 0;
}

```

- Method 2

```

class Complex
{
private:
    int real;
    int imag;
public:
    friend Complex operator+( Complex &c1, Complex &c2 );
};
Complex operator+( Complex &c1, Complex &c2 )
{
    Complex temp;
    temp.real = c1.real + c2.real;
    temp.imag = c1.imag + c2.imag;
    return temp;
}
int main( void )
{
    Complex c1( 10, 20 );
    Complex c2( 30, 40 );
    Complex c3;
    c3 = c1 + c2;    //c3 = operator+( c1, c2 );
    return 0;
}

```

- We can define operator function using
  1. Member function
  2. Non member function
- By defining operator function, it is possible to use operator with objects. This process of giving extension to the meaning of the operator is called operator overloading.
- Using operator overloading we can not create user define operators rather we can increase capability of existing operators.

### Limitations of operator overloading

- We can not overload following operators using member function as well as non member function.
  1. Dot/member selection operator( . )
  2. Pointer to member selection 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 operator's using non member function
  1. Assignment operator( = )
  2. ☐ )

3. Function Call operator[ () ]

4. Arrow / dereferencing operator( -> )

- We can change meaning of operator using operator overloading.
- We can not change number of parameters passed to the operator function.

## Day 9

---

### Operator Overloading

#### Operator Overloading Using Member function

```
Complex c1(10, 20), c2(30, 40), c3;
c3 = c1 + c2;    //c3 = c1.operator+( c2 )
```

```
Complex c1(10, 20), c2(30, 40), c3(50, 60), c4;
c4 = c1 + c2 + c3;    //c4 = c1.operator+( c2 ).operator+( c3 );
```

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

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

- Conclusion : If we want to overload unary operator using member function then operator function should not take parameter.
- Note : We can not overload ternary operator.
- Using operator overloading, we can not change precedence and associativity.

#### Overloading Arithmetic Operators( +, -, \*, /, % )

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

```
Complex c1(10, 20);
Complex c2(30, 40);
Complex c3(50, 60);
Complex c4(70, 80);
```

```
Complex c5;
c5 = c1 + c2 * c3 - c4;
//c5 = c1.operator+( c2.operator*(c3) ).operator-( c4 );
```

### Overloading Relational Operator( <, >, <=, >=, ==, != )

- Relational operators return boolean value( true/false ).

```
bool status = c1 < c2; //bool status = c1.operator<( c2 )
bool status = c1 > c2; //bool status = c1.operator>( c2 )
bool status = c1 <= c2; //bool status = c1.operator<=( c2 )
bool status = c1 >= c2; //bool status = c1.operator>=( c2 )
bool status = c1 == c2; //bool status = c1.operator==( c2 )
bool status = c1 != c2; //bool status = c1.operator!=( c2 )
```

### Overloading Shorthand operator( +=, -=, \*=, /= etc )

```
int x = 10;
x = x + 5;
```

- above statement can be written using shorthand operator

```
int x = 10;
x += 5; //x = x + 5
```

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

- this represents address of current object and "this" represent state of current object.

### Overloading Unary Operator( ++, -- )

- Overloading preincrement operator( ++ )

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

- Overloading postincrement operator( ++ )

```
Complex c1( 10, 20 );
Complex c2;
c2 = c1 ++; //c2 = c1.operator++( 0 )
```

### Operator Overloading Using Non Member function

```
Complex c1(10, 20), c2( 30, 40), c3;
//Using Member Function
c3 = c1 + c2; //c3 = c1.operator+( c2 )

//Using Non Member Function
c3 = c1 + c2; //c3 = operator+( c1, c2 )
```

```
Complex c1(10, 20), c2( 30, 40), c3(50, 60), c4;
//Using Member Function
c4 = c1 + c2 + c3; //c4 = c1.operator+( c2 ).operator+( c3 )

//Using Non Member Function
c4 = c1 + c2 + c3; //c4 = operator+( operator+(c1, c2), c3 );
```

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

```
Complex c1( 10, 20 ), c2;
//Using Member Function
c2 = ++ c1; //c2 = c1.operator++( )
//Using Non Member Function
c2 = ++ c1; //c2 = operator++( c1 );
```

- Conclusion : If we want to overload unary operator using non member function then operator function should take one parameter.

### Overloading Arithmetic Operators( +, -, \*, /, % )

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

### Overloading Realational Operators( <, >, <=, >=, ==, != )

```
bool status = c1 == c2; //bool status = operator==( c1, c2 );
bool status = c1 != c2; //bool status = operator!=( c1, c2 );
```

### Overloading ShortHand Operators( +=, -=, \*= )

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

### Overloading Unary Operator( ++, -- )

```
c2 = ++ c1; //c2 = operator++( c1 );
c2 = c1 ++; //c2 = operator++( c1, 0 );
```

- If copy constructor is private then we can create copy of the object inside member function only.
- If we want to avoid copy then we should use reference.

### Overloading insertion operator( << )

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

- cout is an external object of ostream class.
- Insertion operator( << ) is designed to use with cout.

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

- Copy constructor of ostream class is private( actually protected ) hence we can not create copy of cout object inside our program. If we want to avoid copy operation then we should use reference.

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

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

```
//1. Using Member Function
cout<<c1;    //cout.operator<<( c1 ) //Not OK

//2. Using Non Member Function
cout<<c1;    //operator<<( cout, c1 )    //OK
```

- According to first function call, to print state of c1 on console we should define operator<<() function inside ostream class( because cout is object of ostream class). But we can not modify implementation of ostream class. Hence we can not overload operator << using member function.
- According to second function call, to print state of c1 on console, we should define operator<<() function globally. It is possible for us. Hence we should overload operator<<() using non member function.

```
Complex c1(10,20);
cout<<c1;    //operator<<( cout, c1 );
```

- General Syntax to overload operator insertion( << )

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

### Overloading extraction operator( >> )

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

- cin is external object of istream class.
- Extraction operator( >> ) is designed to use with cin.

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

- Copy constructor of istream class is private hence we can not create copy of cin into in object.

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

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

```
//1. Function Call using member function
cin >> c1;    //cin.operator>>( c1 );
```

```
//2. Function Call using non member function
cin >> c1; //operator>>( cin, c1 );
```

- According to first function call, to accept state of c1 from console, we should define operator>>() function inside istream class, which not possible for us.
- According to second function call, to accept state of c1 from console we should define operator>>() function globally, which practically possible. Hence we should overload operator >> using non member function.

```
Complex c1;
cin>>c1;    //operator>>( cin, c1 );
cout<<c1;   //operator<<(cout, c1 );
```

- General Syntax to overload operator extraction( >> )

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

## Overloading subscript operator( [] )

- If we want to consider object as a array then we should overload index/subscript operator.

```
int& operator[]( int index )
{
    if( index >= 0 && index < this->size )
        return this->arr[ index ];
    throw ArrayIndexOutOfBoundsException("Illegal index");
}
```

- If we use [] operator with object at RHS of assignment operator then expression must return value from memory

```
Array a1(5);
int element = a1[ 2 ]; //int element = a1.operator[]( 2 );
```

- If we use [] operator with object at LHS of assignment operator then expression must return reference from memory



```
Array a1(5);
a1[ 2 ] = 300; //a1.operator[]( 2 ) = 300;
```

- Note : We can not overload subscript operator using non member function.

### Overloading Assignment operator( = )

- If we initialize object from another object of same class then copy constructor gets called.

```
Complex c1( 10, 20 ); //Complex::Complex( int, int )
Complex c2 = c1;
```

- In above code, on c2 object copy constructor gets called.
- If we assign object to the another object of same class then assignment operator function gets called.

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

- In above code, on c2 object assignment operator function gets called.
- 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 we want deep copy then we should overload assignment operator function inside class.
- Compiler provide following functions for a class by default:
  1. Constructor
  2. Destructor
  3. Copy Constructor
  4. Assignment Operator Function.
- General Syntax to overload assignment operator function

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

- Deep Copy inside assignment operator function

```

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

```

## Day 10

---

### Overloading Function Call Operator

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

```

class String
{
private:
    size_t length;
    char *buffer;
public:
    String( void ) : length( 0 ), buffer( NULL )
    {
    }
    String( const char *str )
    {
        this->length = strlen(str);
        this->buffer = new char[ this->length + 1 ];
        strcpy( this->buffer, str );
    }
    void operator()( const char *str )
    {
        this->~String();
        this->length = strlen(str);
        this->buffer = new char[ this->length + 1 ];
        strcpy( this->buffer, str );
    }
    friend ostream& operator<<( ostream &cout, String &other )
    {
        cout<<other.buffer;
        return cout;
    }
    ~String( void )
    {
        if( this->buffer != NULL )
        {
            delete[] this->buffer;
            this->buffer = NULL;
        }
    }
}

```

```
};
int main( void )
{
    String s1("SunBeam");
    s1("Pune");    //s1.operator()("Pune")
    cout<<s1;
    return 0;
}
```

- If we use object as a function then such object is called function object/functor.
- In above code, s1 is functor.
- If we create dynamic object using malloc function then ctor do not call but if create it using new operator then constructor gets called.
- If we deallocate memory using free function then destructor do not call but if we deallocate it using delete operator then destructor gets called.

```
int main( void )
{
    Complex *ptr = new Complex( ); //Complex::Complex( void )
    cout<<endl;
    delete ptr;    //Complex::~~Complex()
    return 0;
}
```

## Overloading Arrow Operator

```
class Complex
{
public:
    void print( void ){ }
};
```

- 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 side operand must be object of a class.

```
Complex c1;
c1.print( );
```

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

```
Complex *ptr = new Complex();
ptr->print( );
delete ptr;
```

- if we use 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( 5 ) );
    obj->acceptRecord( );    //obj.operator ->()->acceptRecord();
    obj->printRecord();      //obj.operator ->()-
>printRecord();
    return 0;
}
```

- We can not overload arrow(->) operator using non member function.

## Constructor Revision

```
class Complex
{
private:
    int real;
    int imag;
public:
    Complex( void )
    {
        this->real = 0;
        this->imag = 0;
    }
}
```

```

Complex( int value)
{
    this->real = value;
    this->imag = value;
}
Complex( 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;                      //Complex( void )
    //Complex c2( 10 );                //Complex( int value)
    //Complex c3(10,20);               //Complex( int real, int imag )
    //Complex c4( );                   //warning: empty parentheses interpreted
as a function declaration
    //Complex c5 = 10;                //Complex c5( 10 );      //Complex( int
value)
    //Complex(10,20);                 //Complex( int real, int imag
//Complex(10,20).printRecord();
    //Complex c6 = 10, 20;             //Complex c6( 10 ), 20; //Error
    //Complex c6 = ( 10, 20);          //Complex c6 = ( 20 );
//Complex c6( 20 );
    /*Complex c1;                     //On c1 parameterless ctor will
call
    Complex c2 = c1;*/                //On c2 default copy ctor will call
    //Complex *ptr; //Compiler do not call constructor on pointer and
reference
    Complex c7 = { 10, 20 };          //error: non-aggregate type
'Complex' cannot be initialized with an initializer list
    return 0;
}

```

## Aggregate Type:

- If language allows us to initialize object using initializer list then its type is called aggregate type and object is called aggregate object.
- In C, Following types are aggregate types:
  1. Array
  2. Structure
  3. Union
- By default class is not an aggregate type.
- If we want to consider class as a aggregate type then it must follow some rules:
  1. All the members of class must be public

2. Class must not contain constructor
  3. Class must not contain virtual function.
  4. Class must not extend any class.
- Aggregate class is also called as Plain Old Data Structure(POD).

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

## Conversion Function

- It is a member function of a class, which is used to convert state of fundamental type into user defined type or vice versa.
- 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 assignment operator function is called conversion function.
- explicit is keyword in C++.
- If we want restrict/prevent automatic instantiation then we should declare Single parameter constructor explicit

```
class Complex
{
private:
    int real;
    int imag;
public:
    Complex( void )
    {
        this->real = 0;
        this->imag = 0;
    }
    explicit Complex( int value )
    {
        this->real = value;
        this->imag = value;
    }
};
```

```
int main( void )
{
    int number = 10;
    Complex c1( number );    //OK
    //Complex c2 = number;    //Not OK
    Complex c2 = ( Complex )number;    //OK
    return 0;
}
```

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

```

int number = 10;
Complex c1;    //OK
//c1 = number;    //Not OK
c1 = ( Complex )number;    //OK
return 0;
}

```

- If we use explicit keyword with constructor then it is called explicit constructor.

## Type Conversion Operator Function

```

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

```

- In above code, type conversion operator function(operator int()) is responsible for converting state of c1 into real hence it is called conversion 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. Unary Operators( ++, -- )
    3. Shorthand operators

## Template

- In C++, if we want to write type-safe generic code then we should use template.
- Types of template
  1. Function template
  2. Class template

### Function template

- Consider function to swap character

```
void swap( char &o1, char &o2 )
{
    char temp = o1;
    o1= o2;
    o2 = temp;
}
```

- Consider function to swap integer

```
void swap( int &o1, int &o2 )
{
    int temp = o1;
    o1= o2;
    o2 = temp;
}
```

- Consider function to swap double

```
void swap( double &o1, double &o2 )
{
    double temp = o1;
    o1= o2;
    o2 = temp;
}
```

- Consider function to swap string object

```
void swap( string &o1, string &o2 )
{
    string temp = o1;
    o1= o2;
    o2 = temp;
}
```

- Generic Swap function.

```
template< typename T>    //T : Type parameter
void swap_object( T &o1, T &o2 )
{
    T temp = o1;
    o1 = o2;
    o2 = temp;
}
int main( void )
{
    int i1 = 10;
    int i2 = 20;
    swap_object<int>(i1, i2);    //int : Type Argument
    cout<<"I1      :      "<<i1<<endl;
    cout<<"I2      :      "<<i2<<endl;
    return 0;
}
```

- Using template, we can not reduce code size or reduce execution time rather we can reduce, programmers effort.
- Template is mainly designed for implementing generic data structure and algorithm.
- An ability of C++ compiler to detect type of function argument automatically and to pass it as a type argument is called type inference.

```
int main( void )
{
    int i1 = 10;
    int i2 = 20;
    swap_object(i1, i2);    //OK
    //is equivalent to
    swap_object<int>(i1, i2);    //Ok
    return 0;
}
```

- How to define function template

```
//template< typename T> //Or
template< class T>
void swap_object( T &o1, T &o2 )
{
    //TODO
}
```

- We can specify multiple type parameters in function definition

```
template< class X, class Y>      //T : Type parameter
void swap_object( X &o1, Y &o2 )
{
    X temp = o1;
    o1 = o2;
    o2 = temp;
}
```

## Class Template

- By passing data type as a argument , we can define generic code in C++. Hence parameterized type is called template.

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

```

void Array<T>::acceptRecord( )
{
    for( int index = 0; index < this->size; ++ index )
    {
        cout<<"Enter element    :    ";
        cin>>this->arr[ index ];
    }
}
template<class T>
void Array<T>::printRecord( )
{
    for( int index = 0; index < this->size; ++ index )
        cout<<this->arr[ index ]<<endl;
}
template<class T>
Array<T>::~~Array( void )
{
    if( this->arr != NULL )
    {
        delete[] this->arr;
        this->arr = NULL;
    }
}
int main( void )
{
    Array<int> a1( 3 );
    a1.acceptRecord();
    a1.printRecord();
    return 0;
}

```

- Limitation : We can divide template code into multiple files.

## Storage Classes

- Storage classes in C/C++
  1. auto
  2. static
  3. register
  4. extern
- Storage class decide scope and lifetime of the element.
- Scope in C++
  1. block scope
  2. function scope
  3. prototype scope
  4. class scope
  5. namespace scope
  6. file scope
  7. program scope
- Scope decide area/region/boundary where we can access element.

- Lifetime in C++
  1. automatic lifetime
  2. static lifetime
  3. dynamic lifetime
- Lifetime decide time/duration i.e how long variable/object should be exist inside memory.

## Local variable

- If we declare variable inside block/function then it is called local variable.

```
void print( )
{
    int count; //Local variable : Function Scope
    for( count = 1; count <=10; ++ count )
    {
        int temp; //Local variable : Block Scope
        //TODO
    }
}
```

- Local variable get space once per function call on stack segment.
- Default value of local variable is garbage.
- If we give call to the function then local variable get space automatically and when function returns control to the calling function then its memory gets deallocated automatically. Hence local variable is also called as automatic variable.
- Default storage class of local variable is auto.

## Static Variable

```
int num1; //BSS Segment : Program Scope
int num2 = 10; //Data Segment : Program Scope

static int num3; //BSS Segment : File Scope
static int num4 = 20; //Data Segment : File Scope
int main( void )
{
    static int num5; //BSS Segment : Function
    Scope
    static int num6 = 30; //Data Segment : Function Scope
    return 0;
}
```

- static is a storage class.
- In C/C++, we can use static keyword with variable as well as function.
- All global and static( local as well as global static variable ) variables get space only once during program loading( or before calling main() function.)

- Uninitialized global and static variable get space on BSS(Block started by symbol) segment.
- Initialized global and static variable get space on data segment.
- Default value of static and global variable is 0.

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

- Consider following code:

```
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 variable between function call then we should declare variable static.
- Static variable is also called as shared variable.
- If we declare local variable static then we can not access it outside function directly. But we can access it using pointer/reference.

```
int* print( void )
{
    static int count = 0;
    count ++;
    return &count;
}
int main( void )
{

```

```

        //cout<<"Count   :           "<<count<<endl; //Not OK

        int *ptr = ::print( );
        cout<<"Count       :           "<<*ptr<<endl; //OK : 1
        *ptr = 10;

        ptr = ::print( );
        cout<<"Count       :           "<<*ptr<<endl; //OK : 11
        return 0;
    }
    ```C++
    * Consider following code
    File Name : Test.cpp
    ```C++
    int num1 = 10; //Global Variable : Program Scope / External Linkage

    static int num2 = 20; //Global Variable : File Scope / Internal Linkage

```

File Name : Main.cpp

```

int main( void )
{
    extern int num1;
    cout<<"Num1       :           "<<num1<<endl; //Ok : 10
    extern int num2;
    cout<<"Num2       :           "<<num2<<endl; //Not OK : Linker Error
    return 0;
}

```

- In Simple words, static variables are same as global variables but having limited scope.
- If we want to access static variable in different file then we should use function. File Name : Test.cpp

```

int num1 = 10;
static int num2 = 20;
void print( void )
{
    cout<<"Num1       :           "<<num1<<endl;
    cout<<"Num2       :           "<<num2<<endl;
}

```

File Name : Main.cpp

```

int main( void )
{
    void print( void ); //Local Function Declration

    print( ); //Function Call : OK
}

```

```
        return 0;
    }
```

- We can declare global function static. If function is static then we can not access it in different file. File Name : Test.cpp

```
int num1 = 10;
static int num2 = 20;
static void print( void )
{
    cout<<"Num1      :      "<<num1<<endl;
    cout<<"Num2      :      "<<num2<<endl;
}
```

File Name : Main.cpp

```
int main( void )
{
    void print( void ); //Local Function Declration

    print( ); //Function Call : Not OK : Linker Error
    return 0;
}
```

- If function is static then local variables are not considered as static.

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

- We can not declare main function static.
- Consider Following code:

```
class Test
{
```



```

private:
    int num1;
    int num2;
    const int num3;
public:
    Test( int num1 = 0, int num2 = 0 ) : num3( 500 )
    {
        this->num1 = num1;
        this->num2 = num2;
    }
    void printRecord( void )
    {
        cout<<"Num1      :      "<<num1<<endl;
        cout<<"Num2      :      "<<num2<<endl;
        cout<<"Num3      :      "<<num3<<endl;

    }
};
int main( void )
{
    Test t1( 10,20 );
    Test t2( 30,40 );
    Test t3( 50,60 );
    t1.printRecord();
    t2.printRecord();
    t3.printRecord();
    return 0;
}

```

- If we want to share value of data member in multiple/all 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 objects depends on size of all non static data members declared inside the class.

### Instance variable:

- \* Data member of the class, which get space inside object is called instance variable. In other words, non static data member is also called as instance variable.
- \* Instance variable get space per instance/object.
- \* How to access it?

```

class Test
{
public:
    //Non static data Member
    int number; //Instance variable;
public:

```

```

//Test *const this = &t
void print( void )
{
    cout<<"Number    :    "<<this->number<<endl;
}
};
int main( void )
{
    Test t;
    t.number = 100;
    t.print( );
    return 0;
}

```

\* Using this pointer, we can access instance variable inside member function and using object we can access it inside non member function.

### Class level Variable:

\* 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.  
 \* How to access it?

```

class Test
{
public:
    //static data member
    static int number; //class level variable;
public:
    //Test *const this = &t
    void print( void )
    {
        cout<<"Number    :    "<<Test::number<<endl;
    }
};
int Test::number; //Global definition
int main( void )
{
    Test::number = 100;
    Test t;
    t.print( );
    return 0;
}

```

- \* Using classname and scope resolution operator, we can access class level variable inside member function as well as non member function.
- \* If data member is static then it is mandatory to provide global definition for it. Otherwise linker will generate error.
- \* In case of modular approach static keyword must appear in declaration(.h) and global definition must appear in definition(.cpp)
- \* It is possible to declare static data member constant.

```

class Test
{
private:
    int num1;
    int num2;
public:
    static const int num3;
public:
    Test( void )
    {
        this->num1 = 10;
        this->num2 = 20;
    }
    void print( void )
    {
        cout<<"Num1      :      "<<this->num1<<endl;
        cout<<"Num2      :      "<<this->num2<<endl;
        cout<<"Num3      :      "<<Test::num3<<endl;
    }
};
const int Test::num3 = 500;
int main( void )
{
    Test t1;
    t1.print();
    return 0;
}

```

## Static member function

- In C++, we can not declare global function constant but we can declare member function constant.
- In C++, except main function we can declare global function as well as member function static.
- If we want to access non static members( data member + member function ) of the class then we should define non static member function. Non static member function is also called instance method.
- Instance methods are designed to call on instance/object.
- If we want to access static members of the class then we should define static member function inside class. Static member function is also called as class level method.
- Class level methods are designed to call on class name.
- Instance Members = { instance variable + instance method }

- Class level Members = { class level variable + class level method }
- Only static member functions are designed to call on class name.

#### this pointer:

- It is implicit pointer which is available in every non static member function of the class which is used to store address of current/calling object.
- Following functions do not get this pointer:
  1. Global function
  2. Static member function
  3. Friend function
- "this" pointer is considered as a link between non static data member and non static member function.

#### Static member function

- Since static member function, do not get this pointer, we can not access non static members inside static member function.
- In other words, static member function can access only static members of the class.
- Using object, we can access non static members inside static member function.
- Inside non static member function, we can access static as well as non static members.

```
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
        cout<<"B::f3()"<<endl;
    }
};
int main( void )
{
    //f3( ); //Not OK
    B::f3();
}
```

```
        return 0;
    }
}
```

- If static members belong to the same class then to access it use of class name and :: operator is optional.
- If static members belong to the different class then to access it use of class name and :: operator is mandatory.
- 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.
- In C++, we can not declare following function static:
  1. constructor
  2. destructor
  3. constant member function
  4. main function
  5. virtual function
- If there is need to use this pointer inside member function then it should be non static other wise we should declare member function static.

```
class Math
{
public:
    static int pow( int base, int index )
    {
        int result = 1;
        for( int count = 1; count <= index; ++ count )
            result = result * base;
        return result;
    }
};

int main( void )
{
    int number = 5;
    int result = Math::pow(number, 2);
    cout<<"Result    :    " << result << endl;
    return 0;
}
```

- Write a program to count objects created by the class

```
namespace ntest
{
    class InstanceCounter
    {
    private:
        static int count;
    public:
        InstanceCounter( void )
        {

```

```

        ++ InstanceCounter::count;
    }
    static int getCount( void )
    {
        return InstanceCounter::count;
    }
};
int InstanceCounter::count = 0;
}
int main( void )
{
    using namespace ntest;
    InstanceCounter c1,c2,c3;
    cout<<"Instance Count :      "<<InstanceCounter::getCount();
    return 0;
}

```

- If we write class without name then it is called anonymous class. We can create object of anonymous class.

```

class
{
public:
    void showRecord( void )
    {
        cout<<"void showRecord( void )"<<endl;
    }
    static void displayRecord( void )
    {
        cout<<"static void displayRecord( void )"<<endl;
    }
}t1;
int main( void )
{
    t1.showRecord();
    t1.displayRecord( );
    return 0;
}

```

- We can call static member function on object but it is designed to call on class name.

```

class Test
{
public:
    void f1( ){ }
    static void f2( ){ }
};
int main( void )
{

```

```

Test t;      //OK
t.f1( );    //OK
t.f2( );    //OK : Not recommended
Test::f1( ); //Not OK
Test::f2( ); //OK : recommended
return 0;
}

```

## Day 12

---

### Inheritance

```

class Person //Parent class / Base Class
{
};
class Employee : public Person //Child Class / 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 declared in base class. In other words, static data members inherit into derived class.
- During inheritance, all the static and non static data members of base class( of any access specifier ) inherit into derived class but only non static data members get space inside object.
- Size of object = sum of size of all non static data members declared in base class and derived class.
- Private data members inherit into derived class.
- We can call non static member function of base class on object of derived class. In other words, non static member function of base class inherit into derived class.
- We can call static member function of base class on Derived class name. In other words, static member function inherit into derived class.
- During inheritance following member functions do not inherit into derived class:
  1. Constructor
  2. Destructor
  3. Copy Constructor
  4. Assignment Operator Function
  5. Friend Function.
- Except above five functions, all the members( including nested class) of base class inherit into derived class.
- If we create object of derived class then first base class ctor and then derived class ctor gets called. Destructor calling sequence is exactly opposite.
- From derived class constructor, by default base class's parameterless constructor gets called. If we want to call any constructor of base class from constructor of derived class then we should use constructors base initializer list.
- Constructors base initializer list represent explicit call to the constructor.

- Inheritance is a process of acquiring/accessing properties( data member) and behavior( member function ) of base class inside derived class.
- Every base class is abstraction for derived class.

```
class Employee : public Person
```

- We can read above statement using two ways.
  1. Class Person is inherited into class employee.
  2. Class Employee is derived from class Person
- To control visibility of members of class we can use private, protected and public keyword. It is called access specifier.
- If we use private, protected and public keyword to extend the class / to create derived class then it is called mode of inheritance.

```
class Employee : public Person
```

- In above statement, mode of inheritance is public.

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

- Default mode of inheritance in C++ is private.

## Day 13

---

### Types of Inheritance

#### Interface Inheritance

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

#### Implementation Inheritance

1. Single Inheritance
2. Multiple Inheritance



### 3. Hierarchical Inheritance

### 4. Multilevel Inheritance

#### Single Inheritance

- Consider example : class B is Derived from class A
- Syntax

```
class A
{
};
class B : public A    //Single Inheritance
{
};
```

- If single base class is having single derived class then it is called single inheritance.
- e.g Car is a vehicle

#### Multiple Inheritance

- Consider example : class D is derived from class A, B and C.
- Syntax:

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

- If multiple base classes are having single derived class then it is called multiple inheritance.
- Example : Allrounder is Baller, Batsman

#### Hierarchical Inheritance

- Consider example : class B, C and D are derived from class A.
- Syntax:

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

- If single base class is having multiple derived classes then it is called hierarchical inheritance.
- HR, Marketing, Finance, Sales department is a department.

#### Multilevel Inheritance

- Consider example : class B is derived from class A, class C is derived from class B and class D is derived from class C.
- Syantx:

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

- If single inheritance is having multiple levels then it is called multilevel inheritance.
- Example : Employee is a Person and Manager is an Employee.

### Hybrid Inheritance.

- Example : istream and ostream are derived from class ios. ifstream and iostream are derived from istream. ofstream and iostream are derived from ostream. fstream is derived from iostream.
- Combination of any two or more than two type of inheritance is called hybrid inheritance.

### When to use inheritance?

- According to client's requirement, if implementation of existing class is partialy complete then we should extend that class ( means we should create its derived class ) i.e we should use inheritance.
- Accoring to clients requirement, if implementation of base class member function is partialy complete then we should redefine member function in derived class.
- If name of members of base class and derived class is same then derived class members hides implementation of inherited members. It is called shadowing. Using derived class object if we try to access these members then preference is always given to the derived class members. In this case, if we want to access members of base class then we should use base class name and :: operator.
- If we want to access, members of base class inside member function of derived class then we should use classname and scope resolution operator.

### Diamond Problem

- Class A is Base class which is having num1 data member and printRecord & showRecord member function.
- Class B is Derived class of class A which is having num2 data member and printRecord & showRecord member function.
- Class C is Derived class of class A which is having num3 data member and printRecord & showRecord member function.
- Class D is Derived class of class B and C which is having num4 data member and printRecord & showRecord member function.
- Problems:
  1. During inheritance, data members of indirect base class inherit into indirect derived class multiple times. Hence if create object of indirect derived class then it affects on size of object.
  2. Member function(s) of indirect base inherit into indirect derived class multiple times. If we try to call such member functions on object of indirect derived class then compiler generates ambiguity error.

3. If create object of indirect derived class then constructor and destructor of indirect base gets called multiple times.

- All above problems generated by hybrid inheritance is called diamond problem.
- virtual is keyword in C++.
- To avoid diamond problem we should declare indirect base class virtual.

```
class A{    };
class B : virtual public A{ };    //Virtual Inheritance
class C : virtual public A{ };    //Virtual Inheritance
class D : public B, public C{    }
```

## Runtime Polymorphism

- Consider following code:

```
class Base
{
private:
    int num1;
    int num2;
public:
    Base( void ) : num1( 10 ), num2( 20 )
    {
    }
    Base( int num1, int num2 ) : num1( num1 ), num2( num2 )
    {
    }
    void showRecord( void )const
    {
        cout<<"Num1      :      "<<this->num1<<endl;
        cout<<"Num2      :      "<<this->num2<<endl;
    }
    void printRecord( void )const
    {
        cout<<"Num1      :      "<<this->num1<<endl;
        cout<<"Num2      :      "<<this->num2<<endl;
    }
};

class Derived : public Base
{
private:
    int num3;
public:
    Derived( void ) : num3( 30 )
    {
    }
    Derived( int num1, int num2, int num3 ) : Base(num1, num2 ), num3(
num3 )
    {
    }
    void printRecord( void )const
    {
        Base::printRecord();
        cout<<"Num3      :      "<<this->num3<<endl;
    }
};
```

```

    }
    void displayRecord( void )const
    {
        Base::showRecord();
        cout<<"Num3      :      "<<this->num3<<endl;
    }
};

```

- During inheritance, members of derived class do not inherit into base class. Hence using base class object we can call member functions of base class only.
- Static object

```

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

```

- Dynamic Object

```

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.
- Static object

```

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

```

```
        return 0;
    }
}
```

- Dynamic Object

```
int main( void )
{
    Derived *ptr = new Derived();
    //ptr->showRecord();    //Base::showRecord();
    //ptr->printRecord();    //Derived::printRecord();
    //ptr->Base::printRecord();    //Base::printRecord()
    //ptr->displayRecord( );    //Derived::displayRecord()
    delete ptr;
    return 0;
}
```

- Members of base class inherit into derived class hence derived class object can be treated/considered as base class object.
- Since derived class object can be considered as base class object, we can use it in place of base class object.

```
Base b1(10,20);
Base b2 = b1;    //OK
```

- Consider following example

```
Derived d1(50,60,70);
Base b1 = d1;    //OK
```

- Consider following example

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

- Consider following example

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

- If we assign derived class object to the base class object then compiler will copy state of base class portion from derived class object into base class object. It is called object slicing.

- In case of object slicing, mode of inheritance must be public.

```
int main( void )
{
    Derived derived(50,60,70);
    Base base;
    base = derived; //OK : Object Slicing
    base.printRecord();    //Base::printRecord(); //50,60
    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.
- Consider following Code:

```
Derived d1( 50,60,70);
Derived d2 = d1;    //OK
```

- Consider following Code:

```
Base b1( 50,60);    //OK
Derived d2 = b1;    //Not OK
```

- Consider following Code:

```
Derived *ptr = new Derived();    //OK
delete ptr;
```

- Consider following Code:

```
Derived *ptr = new Base( ); //Not OK
delete ptr;
```

- Consider following Code:

```
int main( void )
{
    Base b1(50,60);
    Derived d1;
```

```

    d1 = b1;          //Not OK
    d1.printRecord();
    return 0;
}

```

- Process of converting pointer of derived class into pointer of base class is called "Upcasting".
- Upcasting is a form of object slicing.
- Main purpose of upcasting is to minimize object dependency in the code which help us to reduce maintenance of application.

```

int main( void )
{
    Derived *ptrDerived = new Derived( );
    ptrDerived->printRecord();    //Derived::printRecord()
    cout<<endl;
    //Base *ptrBase = ( Base*)ptrDerived;    //OK : Upcasting
    Base *ptrBase = ptrDerived;    //OK : Upcasting
    ptrBase->printRecord(); //Base::printRecord()
    delete ptrDerived;
    return 0;
}

```

- Base class pointer can store address of derived class object. It is also called as upcasting.
- Process of converting pointer of base class into pointer of derived class is called downcasting.
- In case of downcasting explicit type casting is mandatory.

```

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

```

- 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 function in base class is virtual then derived class member function is by default considered as virtual.

- Process of redefining , virtual function of base class inside derived class with same signature is called function overriding.
- Function which is taking part in function overriding is called overridden function.
- Derived class is responsible for overriding function.
- Process of calling member function of Derived class using pointer of base class is called runtime polymorphism.

```
int main( void )
{
    int choice;
    while( ( choice = ::menu_list( ) ) != 0 )
    {
        Product *ptr = NULL;
        switch( choice )
        {
            case 1:
                ptr = new Book();           //Upcasting
                break;
            case 2:
                ptr = new Tape();           //Upcasting
                break;
        }
        if( ptr != NULL )
        {
            ptr->acceptRecord();           //Runtime Polymorphism
            ptr->printRecord();             //Runtime
        }
        delete ptr;
    }
    return 0;
}
```

Polymorphism

## Day 14

---

### Early Binding and Late Binding

- If we give call to the virtual/non virtual function using object then it is considered as early binding.
- If we call non virtual function using pointer or reference then it is considered as early binding.
- If we call virtual function using pointer or reference then it is considered as late binding.

```
class Base
{
public:
    virtual void f1( void )
    {
        cout<<"Base::f1"<<endl;
    }
}
```



```
virtual void f2( void )
{
    cout<<"Base::f2"<<endl;
}
virtual void f3( void )
{
    cout<<"Base::f3"<<endl;
}
void f4( void )
{
    cout<<"Base::f4"<<endl;
}
void f5( void )
{
    cout<<"Base::f5"<<endl;
}
};
class Derived : public Base
{
public:
    virtual void f1( void )
    {
        cout<<"Derived::f1"<<endl;
    }
    void f2( void )
    {
        cout<<"Derived::f2"<<endl;
    }
    void f4( void )
    {
        cout<<"Derived::f4"<<endl;
    }
    virtual void f5( void )
    {
        cout<<"Derived::f5"<<endl;
    }
    virtual void f6( void )
    {
        cout<<"Derived::f6"<<endl;
    }
};
```

```
int main( void )
{
    Base base;
    //base.f1();    //Base::f1 => Early Binding
    //base.f2();    //Base::f2 => Early Binding
    //base.f3();    //Base::f3 => Early Binding
    //base.f4();    //Base::f4 => Early Binding
    //base.f5();    //Base::f5 => Early Binding
    //base.f6();    //error: no member named 'f6' in 'Base'
```

```

        return 0;
    }

```

```

int main( void )
{
    Base *ptr = new Base( );
    //ptr->f1();    //Base::f1 => Late Binding
    //ptr->f2();    //Base::f2 => Late Binding
    //ptr->f3();    //Base::f3 => Late Binding
    //ptr->f4();    //Base::f4 => Early Binding
    //ptr->f5();    //Base::f5 => Early Binding
    //ptr->f6();    //error: no member named 'f6' in 'Base'
    delete ptr;
    return 0;
}

```

```

int main( void )
{
    Base *ptr = new Derived( );    //Upcasting
    //ptr->f1();    //Derived::f1 => Late Binding
    //ptr->f2();    //Derived::f2 => Late Binding
    //ptr->f3();    //Base::f3 => Late Binding
    //ptr->f4();    //Base::f4 => Early Binding
    //ptr->f5();    //Base::f5 => Early Binding
    //ptr->f6();    //error: no member named 'f6' in 'Base'
    delete ptr;
    return 0;
}

```

```

int main( void )
{
    Derived *ptr = new Derived( );
    //ptr->f1();    //Derived::f1 => Late Binding
    //ptr->f2();    //Derived::f2 => Late Binding
    //ptr->f3();    //Base::f3 => Late Binding
    //ptr->f4();    //Derived::f4 => Early Binding
    //ptr->f5();    //Derived::f5 => Late Binding
    //ptr->f6();    //Derived::f6 => Late Binding
    delete ptr;
    return 0;
}

```

- Algorithm to decide early binding and late binding: //Check Whether function is exist in caller datatype if( function is not exist in caller datatype ) { Error : Function is not a member of caller datatype } else { //Check type of caller(Object, pointer/reference ) if( Caller is object ) { Early Binding : Caller type or inherited function

will call. } else { //Check whether function is virtual / non virtual if( function is non virtual ) { Early Binding :  
 Caller type or inherited function will call. } else { Late Binding : Object type / inherited function will call } } }

## Virtual function Internals

### Virtual Function Table

- \* If we declare member function virtual then during compilation C++ compiler generates one table( array/structure)which is used to store address of virtual function declared inside class. It is called virtual function table/vf-table/ v-table.
- \* In other words, a table which stores address of virtual function declared inside class is called v-table.
- \* Compiler generates v-table per class.
- \* Virtual function table is an array of function pointer.

### Virtual Function Pointer

- \* To store address of virtual function table, compiler implicitly declare on pointer as data member inside class. Such hidden pointer is called virtual function pointer/ vf-ptr/v-ptr.
- \* In other words, a pointer which stores address of virtual function table is called virtual function pointer.
- \* V-Table gets generated per class whereas v-ptr gets generated per object.

- Size of object = sum of size of all the non static data members declared in base class and derived class + 2/4/8 bytes ( if Base class / Derived class contains at least on virtual function )
- V-Table and V-Ptr inherit into derived class.

## Need Of virtual Destructor

```
class Base
{
private:
    int *ptr;
public:
    Base( void )
    {
        cout<<"Base( void )"<<endl;
        this->ptr = new int[ 3 ];
    }
    virtual ~Base( void )
    {
        cout<<"~Base( void )"<<endl;
        delete[] this->ptr;
    }
}
```

```

    }
};
class Derived : public Base
{
private:
    int *ptr;
public:
    Derived( void )
    {
        cout<<"Derived( void )" << endl;
        this->ptr = new int[ 3 ];
    }
    ~Derived( void )
    {
        cout<<"~Derived( void )" << endl;
        delete[] this->ptr;
    }
};
int main( void )
{
    Base *ptr = new Derived();
    cout<<endl;
    delete ptr;
    return 0;
}

```

- In case of upcasting, to call destructor of derived class, it is necessary to declare destructor in Base class virtual.

## Pure Virtual Function

- According to client's requirement, if implementation of member function is logically 100% complete then it should be non virtual.
- According to client's requirement, if implementation of member function is logically partially complete then it should be virtual.
- According to client's requirement, if implementation of member function is logically 100% incomplete then it should be pure virtual.
- If we equate virtual function to zero then such virtual function is called pure virtual function.

```

virtual void acceptRecord( void ) = 0;
virtual void calculateArea( void ) = 0;

```

- Pure virtual function do not contain body.
- Member function of a class which is having a body is called concrete method and member function without body is called abstract method.
- Since pure virtual function do not contain body it is also called as abstract method.
- If class contains at least one pure virtual function then such class is called abstract class.
- We can not instantiate abstract class but we can create pointer and reference of abstract class.

- It is mandatory to override pure virtual function in derived class otherwise derived class can be considered as abstract class.
- An ability of different types of object to use same interface to perform different operation is called runtime polymorphism.

```
class A //Pure Abstract class / Interface
{
public:
    virtual void f1( void ) = 0;
    virtual void f2( void ) = 0;
};
```

- If class contains all pure virtual functions then such class is called pure abstract class / interface.

```
class A // Interface
{
public:
    virtual void f1( void ) = 0;
    virtual void f2( void ) = 0;
};
class B : public A //Interface Inheritance
{
public:
    virtual void f3( void ) = 0;
};
```

## Day 15

---

### RTTI

- Runtime Type Identification/Information.
- It is the process of getting information/type of object at runtime.
- is a standard header file of C++ which contains std namespace and std namespace contain declaration of type\_info class.

```
Header File : <typeinfo>
namespace std
{
    class type_info
    {
    public:
        const char* name() const;
        virtual ~type_info();
    private:
        type_info(const type_info &other); //Copy constructor
        type_info& operator=(const type_info& other); //op=
```

```
};

}
```

- name() is member function of class type\_info which returns string that contains name of datatype.
- Copy constructor and assignment operator function of type\_info class is private hence we can not create copy of object of type\_info class in non member function.
- If we want to use RTTI then we must use typeid operator.
- typeid operator returns reference of const object of type\_info class.
- Steps to use RTTI:
  1. Include header file
  2. Use typeid operator with object.
  3. Invoke name() function on object of type\_info class.

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

int main( void )
{
    int number;
    const type_info &type = typeid( number );
    string typeName = type.name();
    cout<<"Type Name      :      "<<typeName<<endl;
    return 0;
}
```

- Or

```
int main( void )
{
    int number;
    const type_info &type = typeid( number );
    cout<<"Type Name      :      "<<type.name()<<endl;
    return 0;
}
```

- or

```
int main( void )
{
    int number;
    cout<<"Type Name      :      "<<typeid( number).name()<<endl;
    return 0;
}
```

- In case of upcasting, if we want to find out type of object then we should use RTTI.

```
class Base
{
private:
    int num1;
    int num2;
public:
    Base( void ) : num1( 0 ), num2( 0 )
    {
    }
    void print( void )
    {
    }
};
class Derived
{
private:
    int num3;
public:
    Derived( void ) : num3( 0 )
    {
    }
    void print( void )
    {
    }
};
```

```
int main( void )
{
    Base base;
    cout<<typeid( base ).name()<<endl;    //4Base
    return 0;
}
```

```
int main( void )
{
    Base *ptr = new Base();
    cout<<typeid( ptr ).name()<<endl;    //P4Base
    cout<<typeid( *ptr ).name()<<endl;    //4Base
    delete ptr;
    return 0;
}
```

```
int main( void )
{
    Derived derived;
    cout<<typeid( derived ).name()<<endl;    //7Derived
    return 0;
}
```

```

int main( void )
{
    Derived *ptr = new Derived( );
    cout<<typeid( ptr ).name()<<endl;      //P7Derived
    cout<<typeid( *ptr ).name()<<endl;      //7Derived
    delete ptr;
    return 0;
}

```

```

int main( void )
{
    Base *ptr = new Derived( );      //Upcasting
    cout<<typeid( ptr ).name()<<endl;      //P4Base
    cout<<typeid( *ptr ).name()<<endl;      //4Base
    delete ptr;
    return 0;
}

```

- In case of upcasting, if we want to find out true type of object then base class must be polymorphic.

```

class Base
{
private:
    int num1;
    int num2;
public:
    Base( void ) : num1( 0 ), num2( 0 )
    {
    }
    //Due to this function class Base is polymorphic
    virtual void print( void )
    {
    }
};
class Derived : public Base
{
private:
    int num3;
public:
    Derived( void ) : num3( 0 )
    {
    }
    void print( void )
    {
    }
};

```



```
int main( void )
{
    Base *ptr = new Derived( );    //Upcasting
    cout<<typeid( ptr ).name()<<endl;    //P4Base
    cout<<typeid( *ptr ).name()<<endl;    //7Derived
    delete ptr;
    return 0;
}
```

- Using NULL pointer, if we try to find out true type of object then typeid operator throws bad\_typeid exception.

```
int main( void )
{
    Base *ptr = NULL;    //ptr is NULL pointer
    cout<<typeid( ptr ).name()<<endl;    //P4Base
    cout<<typeid( *ptr ).name()<<endl;    //Exception :
    std::bad_typeid
    return 0;
}
```

## Advanced Type Casting Operators

1. static\_cast
2. dynamic\_cast
3. const\_cast
4. reinterpret\_cast

### reinterpret\_cast operator

- We can access private data members in non member function using:
  1. Member Function( Getter & Setter )
  2. Friend Function
  3. Pointer
- Accessing private data member using pointer( C - Style )

```
int main( void )
{
    Complex c1;
    cout<<c1<<endl;

    Complex *ptr = &c1;
    int *ptrInt = ( int* )ptr;    //C-Style Type Casting
    *ptrInt = 50;
    ptrInt = ptrInt + 1;
}
```

```

        *ptrInt = 60;

        cout<<c1<<endl;
        return 0;
    }

```

- If we want to do type conversion between incompatible types or if we want to convert pointer of any type into pointer of any other type then we should use reinterpret\_cast operator.
- syntax: reinterpret\_cast( source );
- Accessing private data member using pointer( C++ - Style )

```

int main( void )
{
    Complex c1;
    cout<<c1<<endl;

    Complex *ptr = &c1;
    Casting int *ptrInt = reinterpret_cast<int*>( ptr );    //C++-Style Type
    *ptrInt = 50;
    ptrInt = ptrInt + 1;
    *ptrInt = 60;

    cout<<c1<<endl;
    return 0;
}

```

### const\_cast operator

```

class Test
{
private:
    int number;
public:
    //Test *const this;
    Test( void ) : number( 10 )
    {
    }
    //Test *const this;
    void showRecord( void )
    {
        //TODO
    }
    //const Test *const this;
    void displayRecord( void )const
    {
        //TODO
    }
};

```

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

```
int main( void )
{
    Test t1;
    t1.showRecord();      //OK
    t1.displayRecord();   //OK
    return 0;
}
```

- If we create pointer to non constant object then on pointer we can call constant as well as non constant member function.

```
int main( void )
{
    Test t1;
    Test *ptr = &t1;
    ptr->showRecord();     //OK
    ptr->displayRecord();   //OK
    return 0;
}
```

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

```
int main( void )
{
    const Test t1;
    //t1.showRecord();     //Not OK
    t1.displayRecord();    //OK
    return 0;
}
```

- If we create pointer to constant object then on pointer we can call only constant member function.

```
int main( void )
{
    const Test t1;
    const Test *ptr = &t1;
    //ptr->showRecord();    //Not OK
    ptr->displayRecord();    //OK
    return 0;
}
```

- If we want to convert pointer to constant object into pointer to non constant object and reference to constant object into reference to non constant object then we should use `const_cast` operator.

```
class Test
{
private:
    int number;
public:
    //Test *const this;
    Test( void ) : number( 10 )
    {
    }
    //Test *const this;
    void showRecord( void )
    {
        cout<<"Number    :    "<<this->number<<endl;
    }
    //const Test *const this;
    void displayRecord( void )const
    {
        //Test *const ptr = ( Test *const )this;          //C-Style
        Test *const ptr = const_cast<Test *const>( this );
//C++ - Style
        ptr->showRecord();
    }
};

int main( void )
{
    const Test t1;
    t1.displayRecord();    //t1.displayRecord( &t1 );
    return 0;
}
```

### **static\_cast operator**

- if we want to do type conversion between compatible types then we should use `static_cast` operator.
- Compatible conversion:
  1. Fundamental to fundamental
  2. Base class to derived class or vice versa.

```
int main( void )
{
    int num1 = 10;
    //double num2 = ( double )num1; //OK
    double num2 = static_cast<double>( num1 );    //OK
    cout<<"Num2    :    "<<num2<<endl;
    return 0;
}
```

```

int main( void )
{
    double num1 = 10.5;
    //int num2 = ( int )num1;          //C-Style
    int num2 = static_cast<int>( num1 );    //C++ Style
    cout<<"Num2      :      "<<num2<<endl;
    return 0;
}

```

- In case of non polymorphic type, if we want to do downcasting then we should use static\_cast operator.
- Only in case of upcasting, we can do downcasting otherwise it gets failed.

```

class Base
{
private:
    int num1;
    int num2;
public:
    Base( void )
    {
        this->num1 = 0;
        this->num2 = 0;
    }
    void setNum1( int num1 )
    {
        this->num1 = num1;
    }
    void setNum2( int num2 )
    {
        this->num2 = num2;
    }
    void print( void )
    {
        cout<<"Num1      :      "<<this->num1<<endl;
        cout<<"Num2      :      "<<this->num2<<endl;
    }
};

class Derived : public Base
{
private:
    int num3;
public:
    Derived( void )
    {
        this->num3 = 0;
    }
    void setNum3( int num3 )
    {
        this->num3 = num3;
    }
    void print( void )

```

```

    {
        Base::print();
        cout<<"Num3      :      "<<this->num3<<endl;
    }
};

```

- In case of upcasting, using base class pointer, we can access data members of base class only.
- In case of upcasting, using base class pointer, we can access member functions of base class and overridden functions of derived class.
- In case of upcasting, using base class pointer, we can not access non overdden functions of derived class.
- In case of upcasting, if we want to access data member and non overridden member function of derived class then we should do downcasting.
- Process of converting pointer of base class into pointer of derived class is called downcasting.

```

int main( void )
{
    Base *ptrBase = NULL;
    Derived *ptrDerived = static_cast<Derived*>(ptrBase);
//DownCasting
    return 0;
}

```

```

int main( void )
{
    Base *ptrBase = new Derived( );
    ptrBase->setNum1( 10 );
    ptrBase->setNum2( 20 );
    //Derived *ptrDerived = ( Derived*)ptrBase;      //DownCasting : C-
Style
    Derived *ptrDerived = static_cast<Derived*>(ptrBase);
//DownCasting : C++-Style
    ptrDerived->setNum3(30);
    ptrDerived->print();
    delete ptrDerived;
    return 0;
}

```

- Since members of base class inherit into derived class, we can consider derived class object as a base class object.
- e.g 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.

```
Derived d1(500,600,700);
Base b1 = d1;    //OK
```

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

- Member of derived class do not inherit into base class hence base class object can not be considered as derived class object.
- e.g. Every person is not a Employee.
- Since base class object can not be considered as derived class object, we can not use it in place of derived class Object.

```
Base b1(100,200);
Derived d1 = b1; //Not OK
```

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

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

```
int main( void )
{
    Base *ptrBase = new Base( );
    ptrBase->setNum1(10);
    ptrBase->setNum2(20);
    Derived *ptrDerived = static_cast<Derived*>(ptrBase);
    //DownCasting : OK But Not valid
    ptrDerived->setNum3(30);
    ptrDerived->print();
    delete ptrDerived;
    return 0;
}
```

- static\_cast operator do not check whether type conversion is valid or invalid. It only checks inheritance between source and destination. Hence above code is showing output but conceptually invalid.
- To overcome this limitation, we should use dynamic\_cast operator.

## dynamic\_cast

- In case of polymorphic type, if we want to do downcasting then we should use `dynamic_cast` operator. In other words, to use `dynamic_cast` operator base class must be polymorphic.
- In case of pointer, if `dynamic_cast` operator fails to do downcasting then it returns `NULL`.

```
int main( void )
{
    Base *ptrBase = new Base( );
    ptrBase->setNum1(10);
    ptrBase->setNum2(20);
    Derived *ptrDerived = dynamic_cast<Derived*>(ptrBase);
    //DownCasting :
    if( ptrDerived != NULL )
    {
        ptrDerived->setNum3(30);
        ptrDerived->print();
        delete ptrDerived;
    }
    else
        cout<<"NULL"<<endl;
    //Output : NULL
    return 0;
}
```

- In case of reference, if `dynamic_cast` fails to do downcasting then it throws `bad_cast` exception.

```
int main( void )
{
    Base b1;
    Base &b2 = b1; //Base *const b2 = &b1
    b2.setNum1(10);
    b2.setNum2(20);
    Derived &d1 = dynamic_cast<Derived&>( b2 ); //Downcasting :
    Exception : std::bad_cast
    d1.setNum3(30);
    d1.print();
    return 0;
}
```

```
int main( void )
{
    Base *ptrBase = new Derived( );
    ptrBase->setNum1( 10 );
    ptrBase->setNum2( 20 );
    Derived *ptrDerived = static_cast<Derived*>(ptrBase);
    //DownCasting : OK
    ptrDerived->setNum3(30);
    ptrDerived->print();
    delete ptrDerived;
}
```



```
    return 0;  
}
```

## File Handling

- Variable is a temporary container which is used to store value in RAM.
- File is permanent container which is used to store records on hard disk(HDD).
- Types of file:
  1. Text File
  2. Binary File

### Text File

- e.g .txt, .rtf, .doc, .docx, .c/.cpp/.java, .html/.xml etc
- We can read text file using any text editor.
- It saves data in human readable format.
- It requires more processing hence it is slower in performance.

### Binary File

- e.g .mp3/.mp4, .jpg/.jpeg/.gif/.bmp, .obj/.o, .class etc
- To read binary file, we must use some specific program.
- It doesn't save data in human readable format.
- It requires less processing hence it is faster in performance.

## Stream

- It is an abstraction( object ) which is used to produce( write ) and consume( read ) information from source to destination.
- Stream is associated with physical device.
- Standard streams of C language that is associated with Console
  1. stdin : associated with keyboard
  2. stdout : associated with monitor
  3. stderr : associated with monitor( error stream ).
- Standard streams of C++ language that is associated with Console
  1. cin : associated with keyboard
  2. cout : associated with monitor
  3. cerr : associated with monitor( error stream ).
  4. clog : It is used to maintain log.
- File Streams:
  1. ifstream : To read record from file
  2. ofstream : To write record in file
  3. fstream : To perform read/write operation on file.
- If we want to perform operations on file then we should use header file.

### File Modes:

- File modes in C
  1. "r" / "rb"
  2. "w" / "wb"
  3. "a" / "ab"
  4. "r+" / "rb+"
  5. "w+" / "wb+"
  6. "a+" / "ab+"
- File Modes in C++ case ios\_base::out: case ios\_base::out | ios\_base::trunc: return "w"; case ios\_base::out | ios\_base::app: case ios\_base::app: return "a"; case ios\_base::in: return "r"; case ios\_base::in | ios\_base::out: return "r+"; case ios\_base::in | ios\_base::out | ios\_base::trunc: return "w+"; case ios\_base::in | ios\_base::out | ios\_base::app: case ios\_base::in | ios\_base::app: return "a+"; case ios\_base::out | ios\_base::binary: case ios\_base::out | ios\_base::trunc | ios\_base::binary: return "wb"; case ios\_base::out | ios\_base::app | ios\_base::binary: case ios\_base::app | ios\_base::binary: return "ab"; case ios\_base::in | ios\_base::binary: return "rb"; case ios\_base::in | ios\_base::out | ios\_base::binary: return "r+b"; case ios\_base::in | ios\_base::out | ios\_base::trunc | ios\_base::binary: return "w+b"; case ios\_base::in | ios\_base::out | ios\_base::app | ios\_base::binary: case ios\_base::in | ios\_base::app | ios\_base::binary: return "a+b";

```
#include<cstring>
#include<iostream>
#include<fstream>
using namespace std;
class Employee
{
private:
    char name[ 30 ];
    int empid;
    float salary;
public:
    Employee( )
    {
        strcpy( this->name, "");
        this->empid = 0;
        this->salary = 0;
    }
    Employee( char *name, int empid, float salary)
    {
        strcpy( this->name, name);
        this->empid = empid;
        this->salary = salary;
    }
    friend istream& operator>>( istream &cin, Employee &other )
    {
        cout<<"Name      :      ";
        cin>>other.name;
        cout<<"Empid     :      ";
        cin>>other.empid;
        cout<<"Salary    :      ";
        cin>>other.salary;
        return cin;
    }
}
```

```

    }
    friend ostream& operator<<( ostream &cout, Employee &other )
    {
        cout<<other.name<<"      "<<other.empid<<"      "
<<fixed<<setprecision(2)<<other.salary;
        return cout;
    }
};

void write_record( const char * filename )
{
    ofstream ofs;
    ofs.open(filename, ios_base::out);
    if( ofs.is_open())
    {
        Employee emp;
        cin>>emp;
        ofs.write(reinterpret_cast<char*>(&emp), sizeof(emp));
        ofs.close();
    }
    else
        cout<<"Error while opening file!!!!"<<endl;
}

void read_record( const char * filename )
{
    ifstream ifs;
    ifs.open(filename, ios_base::in);
    if( ifs.is_open())
    {
        if( !ifs.eof())
        {
            Employee emp;
            ifs.read(reinterpret_cast<char*>(&emp), sizeof(
emp ) );
            cout<<emp<<endl;
        }
        ifs.close();
    }
    else
        cout<<"Error while opening file!!!!"<<endl;
}

int menu_list( void )
{
    int choice;
    cout<<"0.Exit"<<endl;
    cout<<"1.Write Record"<<endl;
    cout<<"2.Read Record"<<endl;
    cout<<"Enter choice      :      ";
    cin>>choice;
    return choice;
}

int main( void )
{
    const char *filename = "file.dat";
    int choice;

```

```
while( ( choice = ::menu_list( ) ) != 0 )
{
    switch( choice )
    {
        case 1:
            ::write_record(filename);
            break;
        case 2:
            ::read_record(filename);
            break;
    }
}
return 0;
}
```

## Day 16

---

### Data Structure

- Unprocessed raw things is called data. If we process data then it is called information.
- Data type of variable describe memory, nature and operations that we can perform on data.
- Data Structure Describes two things:
  1. How to organize data inside memory( RAM )
  2. Which operations should be performed/used to organize data inside memory.
- Data structure is also called as collection and value/data stored in collection is called element.
- If we want to process( Searching/sorting) data efficiently then we should use data structure.
- Types of data structure:
  1. Linear Data Structure
  2. Non Linear Structure

### Linear / Sequential Data Structures

1. Array
2. Stack
3. Queue
4. Linkd List

### Non Linear Data Structures

1. Tree
2. Graph

## Linked List

- It is linear/sequential data structure which do not get continuous memory.
- Linked list is collection of elements where each element is called node. Node is object which may contain 2 parts of three parts depending on type of linked list.
- Types of Linked List:
  1. Singly Linked List
  2. Doubly Linked List
- In a linked list, if node contains only 2 parts:
  1. data
  2. a pointer (next) which stores address of next node then such linked list is called singly linked list.
- In a linked list, if node contains 3 parts:
  1. a pointer (prev) which stores address of previous node
  2. data
  3. a pointer (next) which stores address of next node then such linked list is called doubly linked list.
- Head is a pointer which stores address of first node
- Tail is a pointer which stores address of last node.
- LinkedList's( Singly and Doubly) are further classified into two types:
  1. Linear Linked List
  2. Circular Linked List.
- In a singly linked list, if next pointer of last node contains NULL value then such linked list is called linear singly linked list.
- In a singly linked list, if next pointer of last node contains address of first node then such linked list is called circular singly linked list.
- In a doubly linked list, if previous pointer of first node and next pointer of last node contains NULL value then such linked list is called linear doubly linked list.
- In a doubly linked list, if previous pointer of first node contains address of last node and next pointer of last node contains address of first then such linked list is called circular doubly linked list.
- We can perform following operations on Linked List:
  1. bool empty( void )
  2. void addFirst( int data );
  3. void addLast( int data );
  4. void addAtPosition( int position, int data );
  5. void removeFirst( )
  6. void removeLast( )
  7. void removeFromPosition( int position )

8. Node\* find( int data );
9. void reverseList( void );
10. void printList( void );
11. void clear( void );

- If value of head is NULL then linked list is considered as empty.
- Process of visiting node in a linked list is called traversing.
- We can traverse linked list for:
  1. Printing linked list
  2. Adding node at specific position
  3. Deleting node from specific position
  4. Searching node in a linked list.
  5. Reversing a linked list.
- Searching is a process of finding location( index / address ) of element inside collection.
- Limitations of linear singly linked list
  1. We can not revisit previous node
- To overcome above limitation we should use circular Singly linked list.