Agenda

- Language Fundamentals
- Introduction to CPP
- OOP Concepts
- Hello World
- Data types
- ~~Structure in C++~~
- ~~Inline Functions~~

Classfication of languages:

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

Classfication of high languages:

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

Chracteristics of Language

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

ANSI

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

Histroy of C++

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

C++ Standards

- In 1985, the first edition of The C++ Programming Language was released.
- In 1989, C++ 2.0 was released. New features in 2.0 included multiple inheritance, abstract classes, static member functions, const member functions, and protected members.Later feature additions included templates, exceptions, namespaces, new casts, and a Boolean type.
- In 1998, C++98 was released, standardizing the language, and a minor update (C++03) was released in 2003.
- After C++98, C++ evolved relatively slowly until, in 2011, the C++11 standard was released, adding numerous new features,enlarging the standard library further, and providing more facilities to C++ programmers.
- A minor C++14 update was released in December 2014.
- A major revision where various new additions were introduced in C++17.
- Year and version of cpp
    1. 1998 : C++98
    2. 2003 : C++03
    3. 2011 : C++11
    4. 2014 : C++14
    5. 2017 : C++17
    6. 2020 : C++20

Object-oriented software development (OOSD)

- In the past, the problems faced by software development were relatively simple, from task analysis to programming, and then to the debugging of the program, if its not too big it can be done by one person or a group.
- With the rapid increase of software scale, software personnel faces the problem that is very complicated, and there are many factors that need to be considered.

- The errors generated and hidden errors may reach an astonishing degree, this is not something that can be solved in the programming stage.
- Need to standardize the entire software development process and clarify the software
- The tasks of each stage in the development process, while ensuring the correctness of the work of the previous stage, proceed to the next stage work.
- This is the problem that software engineering needs to study and solve.
- Object-oriented software development and engineering include the following parts:

1. Object oriented analysis (OOA)

- The first step of Object-oriented software development is Object-Oriented Analysis (OOA)
- In the system analysis stage of software engineering, system analysts must integrate with users to make precise Accurate analysis and clear description, summarize what the system should do (not how) from a macro perspective.
- Face right the analysis of the image should be based on object-oriented concepts and methods.
- In the analysis of the task, from the objective existence of things and the relationship between the related objects (including the attributes and behaviors of the objects) and the relationship between the objects are summarized.
- The Objects with the same attributes and behaviors are represented by a class.
- Establish a need to reflect the real work situation model. The model formed at this stage is relatively rough (rather than fine).

2. Object oriented design (OOD)

- The second step of Object-oriented software development is Object-Oriented Design (OOD).
- According to the demand model formed in the object-oriented analysis stage, each part is specifically designed.
- The design of the line class may contain multiple levels (using inheritance).
- Then these classes put forward the ideas and methods of program design, including the design of algorithms.
- In the design stage no specific plan is involved, but a more general description tool (such as pseudo code or flowchart)is used to describe.

3. Object-oriented programming (OOP)

- The third step of Object-oriented software development is Object-oriented Programming (OOP).
- According to the results of object-oriented design, to write it into a program in a computer language, it is obvious that object-oriented Computer language (e.g. C++) needs to be used.
- Otherwise the requirements of object-oriented design cannot be achieved.

Object-oriented programming

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

- 4 major pillars of oops

1. Abstraction
2. Encapsulation
3. Modularity
4. Hierarchy

- 3 minor pillars of oops

1. Typing
2. Concurrency
3. Persistence

- Here, word major means, language without any one of the above feature will not be Object orieneted.
- Here word minor means, above features are useful but not essential to classify language object oriented.

Abstraction

- Getting only essential things and hiding unnecessary details is called as abstraction.
- Abstraction always describe outer behavior of object.
- In console application when we give call to function in to the main function , it represents the abstraction

Encapsulation

- Binding of data and code together is called as encapsulation.
- Implementation of abstraction is called encapsulation.
- Encapsulation always describe inner behavior of object
- Function call is abstraction
- Function definition is encapsulation.

Modularity

- Dividing programs into small modules for the purpose of simplicity is called modularity.

Hirerachy

- Level / order / ranking of abstraction is called hireracy.
- Its main purpose is to achive reusability.
- Advantages of reusability:

1. To reduce develoers efforts.
2. To reduce development time and development cost.

- Types of Hierarcy:

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

4. Creates-a Instantiation

Typing/Polyorphism

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

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

Concurrency

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

Persistance

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

Installation

- To install the gcc/g++ compiler use the below link

```
https://sourceforge.net/projects/tdm-gcc/
```

- Download the vsCode from the below link

```
https://code.visualstudio.com/download
```

Hello World

```

```

```c
// header file
#include <iostream>

// Entry point function
int main()
{
    printf("Hello World");
    return 0;
}
```

- Steps for the compilation

```
g++ demo01.cpp
a.exe
```

Execution of a C++ program

- It involves four stages using different compiling/execution tool, these tools are set of programs which help to complete the C/C++ program's execution process.

1. Preprocessor
    - This is the first stage of any C/C++ program execution process; in this stage Preprocessor processes the program before compilation. Preprocessor include header files, expand the Macros.
2. Compiler
    - This is the second stage of any C/C++ program execution process, in this stage generated output file after preprocessing (with source code) will be passed to the compiler for compilation. Complier will compile the program, checks the errors and generates the object file (this object file contains assembly code).
3. Linker
    - It will link the multiple files
4. Loader
    - It will load the executable for the execution

- We can view the intermediate files like preprocesssed (.i),assembly (.asm) file for out .cpp using below commands
- we can view the .i and .asm files but we cannot view the object .o and executable .exe files

```
# creates an preprocessed File
g++ -E demo01.cpp -o demo.i

# creates an Assembly File
g++ -S demo01.cpp -o demo.asm

# cretaes an object file
g++ -c demo01.cpp
```

```
# cretaes an executable file
g++ demo01.cpp
```

Data Types in cpp

- It describes 3 things about variable / object
    1. Memory : How much memory is required to store the data.
    2. Nature : Which type of data memory can store
    3. Operation : Which operations are allowed to perform on data stored inside memory.
- Their are 3 types of Data types in cpp

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

Type Modifiers

- C++ allows the char, int, and double data types to have modifiers preceding them.
- A modifier is used to alter the meaning of the base type so that it more precisely fits the needs of various situations.
- The data type modifiers are
    1. short
    2. long
    3. signed
    4. unsigned
- The modifiers signed, unsigned, long, and short can be applied to integer base types. In addition, signed and unsigned can be applied to char, and long can be applied to double.
- The modifiers signed and unsigned can also be used as prefix to long or short modifiers. For example, unsigned long int.

Type Qualifiers

- The type qualifiers provide additional information about the variables they precede.
- their are two qualifiers

1. const
2. volatile

Bool Datatype

- it can take true or false value.
- It takes one byte in memory.
- by default the value is false

wchar_t Datatype

- wchar_t stands for Wide Character.
- This should be avoided because its size is implementation defined and not reliable.
- It is similar to char data type, except that it take up twice the space and can take on much larger values.
- As char can take 256 values which corresponds to entries in the ASCII table. On the other hand, wide char can take on 65536 values which corresponds to UNICODE values which is a recent international standard which allows for the encoding of characters for virtually all languages and commonly used symbols.
- The type for character constants is char, the type for wide character is wchar_t. This data type occupies 2 or 4 bytes depending on the compiler being used. Mostly the wchar_t datatype is used when international languages like Japanese are used. This data type occupies 2 or 4 bytes depending on the compiler being used. L is the prefix for wide character literals and wide-character string literals which tells the compiler that that the char or string is of type wide-char.

Agenda

- Structure in C++
- Inline Functions
- Class
- Object
- Console I/O
- this pointer
- Namespace
- Menu Driven Code
- Function Overloading
- Default Argument Function
- Types of Member Functions

Structure in CPP

- In Cpp we can define the functions within the structure
- to access the members of structure we have to create the variable of the structure and access the members using .operator.

```
struct Time
{

    int hrs;
    int min;

    void acceptTime()
    {
        printf("Enter hrs and mins - ");
        scanf("%d%d", &hrs, &min);
    }

    void printTime()
    {
        printf("Time - %d : %d \n", hrs, min);
    }
};

int main()
{
    struct Time t1;
    t1.acceptTime();
    t1.printTime();
    return 0;
}
```

Access Specifier in Structure

- By default all members in structure are accessible everywhere in the program by dot(.) or arrow(□) operators.
- But such access can be restricted by applying access specifiers
    - private: Accessible only within the struct
    - public: Accessible within & outside struct

Inline Function

- Managing function activation record is a job of compiler.
- If we give call to the function then compiler need to create Stack Frame and push it into stack. Upon returning control back to the calling function it needs to destroy stackframe from stack. In other words giving call the function is overhead to the compiler.
- If we want to reduce compilers overhead then we should use inline function.
- C++ provides a keyword inline that makes the function as inline function.
- If we declare function inline then compiler do not call function rather it replaces function call by function body.
- As Inline functions get replaced by compiler at its call statement. It ensures faster execution of function just like macros.
- Inline is request to the compiler.
- In following cases, function is not considered as inline:
    1. If we use loop(for/while) inside function
    2. If we implement function using recursion
    3. If we use jump statement inside function
- In case of modular approach, we can use inline keyword in either declaration, definition or both places.
- We can not declare main function static, constant, virtual or inline.
- Advantage of inline functions over macros: inline functions are type-safe.
- If we define member function inside class then function are by default considered as inline.
- If we want to make member function inline whose definition is global then we must explicitly use inline keyword.
- We can not divide inline function code in multiple files.

Naming Convention

- The naming convention used for software development are

1. Camel Case Convention

- In this case, except word, first Character of each word must be in upper case.

- Consider following example.

    - main()
    - parseInt()
    - showInputDialog
    - addNumberOfDays(int days)

- We should use this convention for

    - Data member

- Member function
- Function Parameter
- Local and global variable

2. Pascal Case Convention

- In this case, including first word, first character of each word must be in upper case.
- Consider following example
  - System
  - StringBuilder
  - NullPointerException
  - IndexOutOfBoundsException
- We should use this convention for
  - Union Name
  - Structure Name
  - Class Name
  - Enum Name

3. Convention For macro and constant

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

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

const float PI = 3.142;

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

4. Naming Convention for namespace

- Name of the namespace should be in lowercase.

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

Class

- Class is a logical entity.
- It is a collection of data member and member function.
- Structure and behavior of an object depends on class hence class is considered as a template/model/blueprint for an object.
- Class represents encpasulation.
- Example: Mobile Phone,Laptop,Car

Object

- It is physical entity.
- Object is a variable/instance of a class.
- An entity, which get space inside memory is called object.
- With the help of instantiation we achive abstraction.
- Example: Nokia 1100, MacBook Pro, Maruti 800

Characteristics of object

- Object defines 3 things

1. State

- Value stored inside object is called state of the object.
- Value of data member represent state of the object.

2. Behavior

- Set of operation that we perform on object is called behavior of an object.
- Member function of class represent behavior of the object.

3. Identity

- Value of any data member, which is used to identify object uniquly is called its identity.
- If state of object is same the its address can be considered as its identity.

Object Size

- If we create object of the class then only non static data members get space inside object.
- Hence size of object is depends on size of all the non static data members declared inside class.
- Member function do not get space inside object.
- Data members get space once per object according to the order of data member declaration.
- Member function do not get space per object rather it gets space on code segment and all the objects of same class share single copy of it.
- Object of an empty class is 1 byte.

Access Specifiers

- If we want to control visibility of members of structure/class then we should use access specifier.
- Access specifiers in C++

1. private(-)

    - visiblity only within the class
2. protected(#)
    - visibility in the derived classes
3. public(+)
    - visibility every where on structure/class object

- In C++, structure members are by default considered as public and class members are by default considered as private

Namespace

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

```
int num1 = 10;
int main()
{
    int num1 = 10;
    printf("value of local num1 = %d\n", num1);
    printf("value of global num1 = %d\n", ::num1);
    return 0;
}
```

- Namespace in C++ language is used:

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

- Namespace can contain:

1. Variable
2. Function
3. Types[structure/union/class]
4. Enum
5. Nested Namespace

- Namespaces are used to organize code into logical groups and to prevent name collisions that can occur especially when your code base includes multiple libraries.s
- We can not instantiate namespace.
- If we want to define namespace then we should use namespace keyword.
- namespaces can only be defined in global or namespace scope. In other words, we can not define namespace inside function/class.
- If we want to access members of namespace then we should use namespace name and scope resolution operator.
- If name of the namespaces are diffrent then we can give same/diffrent name to the members of namespace.
- If name of the namespaces are same then name of members must be different.
- We can define namespace inside another namespace. It is called nested namespace.
- If we define member without namespace then it is considered as member of global namespace.

- If we want to access members of namespace frequently then we should use using directive.

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

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

Console Input and OutPut Operation

- C++ provides an easier way for input and output.
- Console Input -> Keyborad
- Console Output -> Monitor
- Console = Keyboard + Monitor
- iostream is standard header file of C++.

1. cout

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

2. cin

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

Menu driven code

- When we want to execute the code in continious manner and want to execute the code based on users choice then we can write a menu driven code.

Function Overloading

- Functions with same name and different signature are called as overloaded functions.
- Return type is not considered for function overloading.
- Function call is resolved according to types of arguments passed.
- Function overloading is possible due to name mangling done by the C++ compiler (Name mangling process , mangled name)
- Differ in number of input arguments
- Differ in data type of input arguments

- Differ at least in the sequence of the input arguments

Default Argument Function

- In C++, functions may have arguments with the default values. Passing these arguments while calling a function is optional.
- A default argument is a default value provided for a function parameter/argument.
- If the user does not supply an explicit argument for a parameter with a default argument, the default value will be used.
- If such argument is not passed, then its default value is considered. Otherwise arguments are treated as normal arguments.
- Default arguments should be given in right to left order.

this pointer

- If we call member function on object then compiler implicitly pass address of that object as a argument to the function implicitly.
- To store address of object compiler implitly declare one pointer as a parameter inside member function.
- Such parameter is called this pointer.
- this is a keyword. "this" pointer is a constant pointer.
- General type of this pointer is:
    - Classname * const this;

Member Functions

- The functions declared inside the class are called as Member Functions.
- The member functions according to their behaviour are classified into following types

1. constructor
2. destructor
3. Mutator
4. Inspector
5. Facilitator

Mutator

- 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 setHrs() and setMins()

Inspector

- A member function of class, which is used to read state of the object is called inspector function.
- It is also called selector function of getter function.
- e.g getHrs() and getMins()

Facilitator

- Member function of a class which allows us to perform operations on Console/file/database is called faciliator function.
- e.g acceptData() and printData()

Agenda

- Types of Member Functions
- constructor and their types
- Constant
- Dynamic Memory Allocation
- ~~Reference~~

Constructor

- It is a member function of a class which is used to initialize object.
- Due to following reasons, constructor is considered as special function of the class:
    1. Its name is same as class name
    2. It doesn't have any return type.
    3. It is designed to call implicitly.
    4. In the life time of the object is gets called only once.
- We can not call constructor on object, pointer or reference explicitly. It is designed to call implicitly.
- constructor does not get called if we create pointer or reference.
- We can use any access specifier on constructor.
- If ctor is public then we can create object of the class inside member function as well as non member function but if constructor is private then we can create object of the class inside member function only.
- We can not declare constructor static, constant, volatile or virtual. We can declare constructor only inline.
- constructor can contain return statement,it cannot return any value from constructor as return statement is used only to return controll to the calling function.

Types of Constructor

1. Parameterless Constructor
    - A constructor, which do not take any parameter is called Parameterless constructor.
    - It is also called zero argument constructor or user defined default constructor.
    - If we create object without passing argument then parameterless constructor gets called.

```
class Point
{
    int x;
    int y;
public:
    Point()
    {
        x = 1;
        y = 1;
    }
}

int main(){
 Point pt1;
```

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

2. Parameterized Constructor

- If constructor take parameter then it is called parameterized constructor.
- If we create object, by passing argument then paramterized constructor gets called.
- Copy constructor is a single parameter constructor hence it is considered as parameterized constructor.

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

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

3. Default constructor

- If we do not define constructor inside class then compiler generates default constructor for the class.
- Compiler do not provide default parameterized constructor. Compiler generated default constructor is parameterless.
- If we want to create object by passing argument then its programmers responsibility to write parameterized constructor inside class.
- Default constructor do not initialize data members declared by programmer. It is used to initialize data members declared by compiler(e.g v-ptr).
- If compiler do not declare any data member implicitly then it doesnt generate default constructor.
- We can write multiple constructor's inside class. It is called constructor overloading.

```
 Point()
{
    cout << "Inside Parameterless Ctor" << endl;
    x = 1;
    y = 1;
}
// constructor overloading
Point(int value)
{
    x = value;
    y = value;
```

```
}
// constructor overloading
Point(int x, int y)
{
    cout << "Inside Parameterized Ctor" << endl;
    this->x = x;
    this->y = y;
}
```

Constructor delegation(C++ 11)

- In C++98 and C++ 03, we can not call constructor from another constructor. In other words C++ do not support constructor chaining.
- In C++ 11 we can call constructor from another constructor. It is called constructor delegation. Its main purpose is to reuse body of existing constructor.

```
Point() : Point(1, 1)
{
    cout << "Inside Parameterless Ctor" << endl;
}

Point(int value) : Point(value, value)
{
    cout << "Inside single Parameterized Ctor" << endl;
}

Point(int x, int y)
{
    cout << "Inside Parameterized Ctor" << endl;
    this->x = x;
    this->y = y;
}
```

Constructor's member initializer list

- If we want to initialize data members according to users requirement then we should use constructor body.
- If we want to initialize data member according to order of data member declaration then we should use constructors member initializer list.
- Except array we can initialize any member inside constructors member initializer list.
- If we provide constructor member initializer list as well Constructor body then compiler first execute constructor member initializer list.
- In case of modular approach, constructors member initializer list must appear in definition part(.cpp).
- If we declare data member constant then it is mandatory to initialize it using constructors member initializer list.

```
class Point
{
    int x;
    int y;
    const int num;

public:
    // ctor members initializer list initialize data member according to order of
data member declaration in class
    // here x will get initialized first then y and then num
    Point(int value) : y(value), x(++value), num(value) // x= 3, y = 3, num = 3
    {
    }

    // Point(int value)
    // {
    //     this->y = value;   // y = 2
    //     this->x = ++value; // x = 3
    //     this->num = value; // NOT OK
    // }
}
```

Destructor

- It is a member function of a class which is used to release the resources.
- Due to following reasons, it is considered as special function of the class
    1. Its name is same as class name and always preceds with tild operator(~)
    2. It doesnt have return type or doesn't take parameter.
    3. It is designed to call implicitly.
- We can declare destructor as a inline and virtual only.
- Destructor calling sequence is exactly opposite of constructor calling sequence.
- We can not call constructor on object, pointer or reference explicitly. It is designed to call implicitly.
- Destructor is designed to call implicitly but we can call it explicitly.
- If we do not define destructor inside class then compiler generates default destructor for the class.
- Default destructor do not deallocate resources allocated by the programmer. If we want to deallocate it then we should define destructor inside class.

Constant

- const is type qualifier.
- If we dont want modify value of the variable then we should use const keyword.
- constant variable is also called as read only variable.
- In C++, Initializing constant variable is mandatory.

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

- We can even make

1. Data Member as constant

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

```CPP
    class Test
    {
    private:
        const int num1;
    public:
        Test( void ) : num1( 10 ) //OK
        {
            //this->num1 = 10; //Not OK
        }
    };
```
```

2. Member Function as constant

```
- We can not declare global function constant but we can declare member function constant.
- If we dont want to modify state of current object inside member function then we should declare member function constant.
- Non constant member function get this pointer like: `ClassName *const this`
- Constant member function get this pointer like: `const ClassName *const this;`
- We can not delclare following function constant:

  1. Global Function
  2. Static Member Function
  3. Constructor
  4. Destructor

- Since main function is a global function, we can not delcare it constant.
- We should declare read only function constant. e.g getter function, printRecord function etc.
- In constant member function, if we want to modify state of non constant data member then we should use mutable keyword.
```

3. Object as Constant

```
- If we dont want to modify state of the object then instead of declaring data
member constant, we should declare object constant.
- On non constant object, we can call constant as well as non constant member
function.
- On Constant object, we can call only constant member function of the class.
```

Dynamic Memory Allocation

- If we want to allocate memory dynamically then we should use new operator and to deallocate that memory we should use delete operator.
- If pointer contains, address of deallocated memory then such pointer is called dangling pointer.
- When we allocate space in memory, and if we loose pointer to reach to that memory then such wastage of memory is called memory leakage.
- If new operator fails to allocate memory then it throws bad_alloc exception.
- If malloc/calloc/realloc function fails to allocate memory then it returns NULL.
- If new operator fails to allocate memory then it throws bad_alloc exception.
- If we create dynamic object using malloc then constructor do not call. But if we create dynamic object using new operator then constructor gets called.

```cpp
int main()
{
    int *ptr = new int;
    *ptr = 20;
    cout << "Address of dynamic Memory - " << ptr << endl;
    cout << "Value on dynamic memory - " << *ptr << endl;
    delete ptr;
    ptr = NULL;
    return 0;
}
```

Difference between malloc() vs new and free() vs delete

malloc() vs. new:

- Type:
  - malloc(): malloc() is a function. Declared in <stdlib.h>.
  - new: new is an operator. No separate header file needed.
- Initialization:
  - malloc(): Memory allocation only. Doesn't call constructors for objects.
  - new: Memory allocation and initialization. Calls constructors for objects.
- Usage with Arrays:
  - malloc(): No special handling for arrays.
  - new: Supports array allocation. new[] is used for arrays, which can later be deallocated with delete[].
- Return Type:

    - malloc(): Returns a void* pointer.
    - new: Returns a pointer to the type of object being allocated.
- Type Safety:
    - malloc(): Not type-safe. Requires explicit casting.
    - new: Type-safe. No need for explicit casting.
- Overloading:
    - new: Supports overloading to customize memory allocation behavior.
    - malloc(): malloc() is not meant to be overloaded.
- Usage in C++:
    - malloc(): Can be used in C++ but not recommended due to lack of support for constructors.
    - new: Preferred in C++ for dynamic memory allocation because it supports constructors.

free() vs. delete:

- Type:
    - free(): free() is a function. Declared in <stdlib.h>.
    - delete: delete is an operator. No separate header file needed.
- Type of Memory:
    - free(): Used to deallocate memory allocated with malloc() or calloc().
    - delete: Used to deallocate memory allocated with new.
- De-Initialization:
    - free(): Only deallocates memory. Doesn't call destructors for objects.
    - delete: Deallocates memory and calls destructors for objects.
- Usage with Arrays:
    - free(): No special handling for arrays.
    - delete: Used with delete[] to deallocate memory allocated for arrays.
- Overloading:
    - delete: Supports overloading to customize memory deallocation behavior.
    - free(): free() is not meant to be overloaded.
- Usage in C++:
    - free(): Not used in C++. Deallocating memory allocated with malloc() or calloc() using free() in C++ can lead to undefined behavior if
    - the object has non-trivial constructors or destructors.
    - delete: Preferred in C++ for deallocating memory allocated with new because it properly calls destructors for objects.

Link for DrMemory

```
https://drmemory.org/page_download.html
```

Agenda

- Reference
- Static (Data Member & Member Functions),
- simple and dynamic Array(1D)
- ~~simple and dynamic Array(2D)~~
- ~~enum~~
- ~~multiple files~~

typedef

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

Reference

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

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

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

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

- We can not create reference to constant value.

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

```
int &num2 = 10; //Not OK
return 0;
}
```

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

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

Static

- All the static and global variables get space only once during program loading
- Static variable is also called as shared variable.
- If we declare function static then local variables are not considered as static.
- If we dont want to access any global function inside different file then we should declare global function static.
- In C/C++, we can not declare main function static.
- In C++ we can declare
    1. Data member as static
    2. Member function as static

Static Data Member

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

- If we want to declare data member static then we must provide global definition for it otherwise linker generates error.
- Instance variable get space inside instance hence we should initialize it using constructor.
- Class level variable do not get space inside instance hence we should not initialize it inside constructor. We must initialize it in global definition.
- We can declare constant data member static.

Static Member Function

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

Array

- Array is a data structure that is used to store the elements of same type in contagious memory locations.
- the elements stored in the array can be accessed using their index number;
- Types of array
    1. Single Dimension Array
    2. Multi Dimension Array
- we can create array for fundamental data types as well as derived data types

Single Dimension Array

```
// single dimension array
int main()
{
    // int arr[5] = {10, 20, 30, 40, 50};
    int arr[] = {10, 20, 30, 40, 50};
    // int arr[5];
    //  arr[0] = 10;
```

```
    //  ...

    for (int i = 0; i < 5; i++)
        cout << arr[i] << ",";
    cout << endl;
    return 0;
}

// single dimension array of ptrs (Dynamic memory allocation)
int main()
{
    int *arr[5];
    for (int i = 0; i < 5; i++)
        arr[i] = new int(10 * (i + 1));

    for (int i = 0; i < 5; i++)
        cout << *arr[i] << ",";
    cout << endl;

    for (int i = 0; i < 5; i++)
        delete arr[i];
    return 0;
}

// single dimension array with Dynamic memory allocation
int main()
{
    int *arr = new int[5]{10, 20, 30, 40, 50};

    for (int i = 0; i < 5; i++)
        cout << arr[i] << ",";
    cout << endl;
    delete[] arr;
    return 0;
}
```

Agenda

- 2D Array
- enum
- multiple files
- Hierarchy and its type.
- Association
- Inheritance
- ~~Type of Inheritance~~
- ~~Diamond problem~~
- ~~Virtual base class~~
- ~~Mode of Inheritance~~

Multi Dimension Array

```
// multi dimension array
int main()
{
    int arr[][3] = {10, 20, 30, 40, 50, 60};
    // int arr[2][3] = {10, 20, 30, 40, 50, 60};
    for (int i = 0; i < 2; i++)
        for (int j = 0; j < 3; j++)
            cout << arr[i][j] << ",";
    cout << endl;
    return 0;
}

// multi dimension array of ptrs (Dynamic memory allocation)
int main()
{
    int *arr[2][3];
    for (int i = 0; i < 2; i++)
        for (int j = 0; j < 3; j++)
            arr[i][j] = new int(i + j);

    for (int i = 0; i < 2; i++)
        for (int j = 0; j < 3; j++)
            cout << arr[i][j] << endl;

    for (int i = 0; i < 2; i++)
        for (int j = 0; j < 3; j++)
            delete arr[i][j];
    return 0;
}


// multi dimension array with Dynamic memory allocation
int main2()
{
```

```
    int **arr = new int *[2];
    arr[0] = new int[3]{10, 20, 30};
    arr[1] = new int[3]{40, 50, 60};

    for (int i = 0; i < 2; i++)
        for (int j = 0; j < 3; j++)
            cout << arr[i][j] << ",";
    cout << endl;
    delete[] arr[0];
    delete[] arr[1];
    delete[] arr;
    return 0;
}
```

enum

- Enumeration (Enumerated type) is a user-defined data type that can be assigned some limited values. These values are defined by the programmer at the time of declaring the enumerated type.
- Enums provide a way to define symbolic names for sets of integers, making the code more readable and maintainable.

```
#include <iostream>

// Define an enum named Color
enum Color {
    RED,    // 0
    GREEN,  // 1
    BLUE    // 2
};

int main() {
    // Declare a variable of type Color
    Color myColor = GREEN;

    // Check the value of myColor
    if (myColor == GREEN) {
        cout << "The color is green." << endl;
    } else {
        cout << "The color is not green." << std::endl;
    }
    return 0;
}
```

Modularity (Multiple Files)

- "/usr/include" directory is called standard directory for header files.
- It contains all the standard header files of C/C++

- If we include header file in angular bracket (e.g #include<filename.h>) then preprocessor try to locate and load header file from standard directory only(/usr/include).
- If we include header file in double quotes (e.g #include“filename.h") then preprocessor try to locateand load header file first from current project directory if not found then it try to locate and load from standard directory.

```
// Header Guard
  #ifndef HEADER_FILE_NAME_H
  #define HEADER_FILE_NAME_H
    //TODO : Type declaration here
  #endif
```

Hierarchy

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

Types of Hierarcy:

1. Has-a/Part-of Association/Containment
2. Is-a/Kind-of Inheritance/Generalization
3. Use-a Dependancy
    - This hierarchy represents how classes depend on each other.
    - Dependencies occur when one class relies on another class but does not own or control its lifetime.
    - For example, if Class A uses Class B as a method parameter or local variable, there's a dependency between A and B.
4. Creates-a Instantiation
    - This can often be seen in factory design patterns or in scenarios where one class encapsulates the creation logic of another class.

Association

- If has-a relationship exist between two types then we should use association.
- Example:
    1. Room has-a wall
    2. Room has-a chair
    3. Car has-a engine
    4. Car has-a music player
    5. Department has-a faculty
    6. Human has-a heart
- If object is part-of / component of another object then it is called association.

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

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

//Dependant Object : Car Object
//Dependancy Object : Engine Object
```

1. Composition

- If dependancy object do not exist without Dependant object then it represents composition.
- Composition represents tight coupling.
- If we create object of dependency class as data member inside the dependent class it represents composition.

2. Aggegration

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

Agenda

- Inheritance
- Type of Inheritance
- Diamond problem
- Virtual base class
- Mode of Inheritance
- Runtime Polymorphism
- Virtual Functions
- vptr and vtable
- ~~RTTI~~

Inheritance

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

```
//Parent class
class Person//Base class
{
};

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

- During inheritance, members of base class inherit into derived class.
- If we create object of derived class then non static data members declared in base class get space inside it. In other words non static data members of base class inherit into derived class.
- Using derived class name, we can access static data member(if public) of base class. In other words, static data member inherit into derived class.
- All the data members(private/protected/public, static/non static) of base class inherit into derived class but only non static data members get space inside object.
- Size of object = sum of size of non static data members declared in base class and derived class.
- We can call non static member function of base class on object of derived class. In other words, using derived class object we can call non static member function of base class. It means that, non static member function inherit into derived class.
- We can call static member function of base class on derived class. In other words, using derived class name, we can access static member function of base class. It means that, static member function inherit

into derived class.
- Following function's do not inherit into derived class:
    1. Constructor
    2. Destructor
    3. Copy constructor
    4. Assignment operator function
    5. Friend Function
- Except above five function's, all the member's of base cass(data member, member function and nested type) inherit into derived class.
- If we create object of derived class then first base class and then derived class constructor gets called.
- Destructor calling sequence is exactly opposite.
- From derived class constructor, by default, base class's parameterless constructor gets called.
- Using constructors base initializer list, we can call any constructor of base class from constructor of derived class.
- In C++, we can not call constructor on object, pointer or reference explicitly. But constructor's base initializer list represent explict call to the constructor.
- We can read following statement using 2 ways:
    - `class Employee : public Person`
    1. Class Person is inherited into class Employee.
    2. Class Employee is derived from class Person(Recommended).
- Process of acquiring/getting/accessing properties(data members) and behavior (member function) of base class inside derived class is called is called inheritance.
- Every base class is abstraction for the derived class.

Types of inheritance

1. Single Inheritance

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

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

2. Multiple Inheritance

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

```
class A{
};
class B{
};
class C{
```

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

3. Hierarchical Inheritance

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

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

4. Multilevel Inheritance

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

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

Hybrid Inheritance

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

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

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

Diamond Problem

- It is hybrid inheritance. Its shape is like diamond hence it is also called as diamond inheritance.

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

- Data members of indirect base class inherit into the indirect derived class multiple times.
- Hence it effects on size of object of indirect derived class.
- Member functions of indirect base class inherit into indirect derived class multiple times.
- If we try to call member function of indirect base class on object of indirect derived class, then compiler generates ambiguity error.
- If we create object of indirect derived class, then constructor and destructor of indirect base class gets called multiple times.
- All above problems generated by hybrid inheritance is called diamond problem.
- If we want to overcome diamond problem, then we should declare base class virtual i.e. we should derive class B & C from class A virtually. It is called virtual inheritance. In this case, members of class A will be inherited into B & C but it will not be inherited from B & C into class D.

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

Mode of inheritance

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

```
class Employee : public Person

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

- In private mode of inheritance, the visibility of base class members that inherit inside the derived class is made as private inside the derived class
- In protected mode of inheritance except private members, the visibility of base class members that inherit inside the derived class is made as protected inside the derived class
- In public mode of inheritance, the visibility of base class members that inherit inside the derived class does not change inside the derived class
- In all types of mode, private members inherit into derived class but we can not access it inside member function of derived class.
- If we want to access private members inside derived class then
    1. Either we should use member function(getter/setter).
    2. or we should declare derived class as a friend inside base class.
- If we want to create object of derived class then constructor of base class and derived must be public

Mode Of inheritance – Private, Protected & Public

Irrespective of Mode of Inheritance			
Access Specifiers	Same Class	Friend Function	Non Member Function
private	A	A	NA
protected	A	A	NA
public	A	A	A

Public Mode of Inheritance		
Access Specifiers from Base class	Derived Class	Indirect Derived Class
private	NA	NA
protected	A	A
public	A	A

Private Mode of Inheritance		
Access Specifiers from Base class	Derived Class	Indirect Derived Class
private	NA	NA
protected	A	NA
public	A	NA

Protected Mode of Inheritance		
Access Specifiers from Base class	Derived Class	Indirect Derived Class
private	NA	NA
protected	A	A
public	A	A

RunTime Polymorphism

- 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.
- Members of derived class do not inherit into the base class hence using base class object we can access members of base class only.
- Members of base class inherit into derived class hence derived class object can be considered as base class object.
- Example : Employee object is-a Person object.
- Since Derived class object can be considered as Base class object, we can use it in place of Base class object.

```
Base b1;
Base b2 = b1;//OK
Derived d1;
b1 = d1;//OK
```

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

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

```
class Base{
    public:
    int n1,n2;
    void printBase(){
        cout<<num1<<","<<num2<<endl;
    }
}
class Derived:public Base(){
    public:
    int n3;
    Derived(int n1,int n2,int n3){
        this->n1=n1;
        this->n2=n2;
        this->n3=n3;
    }
}
int main( void )
{
Base base;
Derived derived( 500,600,700);
base = derived; //OK : Object Slicing
base.printRecord(); //Base::printRecord() : 500,600
return 0;
}
```

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

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

ptr = new Base();//Not Ok
```

- Process of converting, pointer of derived class into pointer of base class is called upcasting.
- Upcasting represents object slicing.
- In case of upcasting, explicit type casting is optional.
- Main purpose of upcasting is to reduce object dependancy in the code.
- Process of converting pointer of base class into pointer of derived class is called downcasting.
- In Case of downcasting, explicit typecasting is mandatory.
- Note: Only in case of upcasting, we can do downcasting. Otherwise downcasting will fail.

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

    Derived *ptrDerived = ( Derived*)ptrBase;//Downcasting
    ptrDerived->printRecord();

    return 0;
}
```

Virtual Function

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

- Definition 1: In case of upcasting, a member function, which gets called depending on type of object rather than type of pointer, is called virtual function.
- Definition 2: In case of upcasting, a member function of derived class which is designed to call using pointer of base class is called virtual function.
- We can call virtual function on object but it is designed to call on Base class pointer or reference.

Early Binding And Late Binding

- If Call to the function gets resolved at compile time then it is called early binding.
- If Call to the function gets resolved at run time then it is called late binding.
- If we call any virtual/non virtual function on object then it is considered as early binding.
- If we call any non virtual function on pointer/reference then it is considered as early binding.
- If we call any virtual function on pointer/reference then it is considered as late binding.

v-ptr and v-table

- Size of object = size of all the non static data members declare in base class and derived class + 2/4/8 bytes(if Base/Derived class contains at least one virtual function).
- If we declare member function virtual then to store its address compiler implicitly create one table(array/structure). It is called virtual function table/vf-table/v-table.
- In other words, virtual function table is array of virtual function pointers.
- Compiler generates V-Table per class.
- To store address of virtual function table, compiler implicitly declare one pointer as a data member inside class. It is called virtual function pointer / vf-ptr / v-ptr.
- v-ptr get space once per object.
- ANSI has not defined any specification/rule on position of v-ptr hence compiler vendors are free to decide its position in object. But generally it gets space at the start of the object.
- The vptr is managed by the compiler and is automatically set up during object construction. It is not something that you need to initialize or manage explicitly in your code. It's a mechanism provided by the compiler to enable polymorphic behavior and dynamic dispatch of virtual function calls.
- V-Table and V-Ptr inherit into derived class.
- Process of calling member function of derived class using pointer/reference of base class is called Runtime Polymorphism.
- According to client's requirement, if implementation of Base class member function is logically 100% complete then we should declare Base class member function non virtual.
- According to client's requirement, if implementation of Base class member function is logically incomplete / partially complete then we should declare Base class member function virtual.
- According to client's requirement, if implementation of Base class member function is logically 100% incomplete then we should declare Base class member function pure virtual.

Pure Virtual Function:

- If we equate, virtual function to zero then such virtual function is called pure virtual function.
- We can not provide body to the pure virtual function.
- If class contains at least one pure virtual function then such class is called abstract class.
- If class contains all pure virtual functions then such class is called pure abstract class/interface.

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

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

- We can instantiate concrete class but we can not instantiate abstract class and interface.
- We can not instantiate abstract class but we can create pointer/reference of it.
- If we extend abstract class then it is mandatory to override pure virtual function in derived class otherwise derived class can be considered as abstract.
- Abstract class can contain, constructor as well as destructor.
- An ability of different types of object to use same interface to perfrom different operation is called Runtime Polymorphism.

Agenda

- RTTI
- Virtual Destructor
- Advanced Casting Operators
- Exception Handling
- Friend Function and class
- Manipulators
- ~~Template~~
- ~~Shallow Copy and Deep Copy~~
- ~~Copy Constructor~~

Runtime Type Information/Identification[RTTI]

- It is the process of finding type(data type/ class name) of object/variable at runtime.
- It is in standard C++ Header file(/usr/include). It contains declaration of std namespace. std namespace contains declaration of type_info class.
- Since copy constructor and assignment operator function of type_info class is private we can not create copy of it in our program.
- If we want to use RTTI then we must use typeid operator.
- typeid operator return reference of constant object of type_info class.
- To get type name we should call name() member function on type_info class object.

```
#include<iostream>
#include<string>
#include<typeinfo>
using namespace std;
int main( void )
{
float number = 10;
const type_info &type = typeid( number );
string typeName = type.name();
cout<<"Type Name : "<<typeName<<endl;
return 0;
}
```

- In case of upcasting, if we want to find out type of object then we should use RTTI.
- In case of upcasting, if we want to find out true type of object then base class must be polymorphic.
- Using NULL pointer, if we try to find out true type of object then typeid throws std::bad_typeid exception.

Virtual Destructor

- A destructor is implicitly invoked when an object of a class goes out of scope or the object's scope ends to free up the memory occupied by that object.
- Due to early binding, when the object pointer of the Base class is deleted, which was pointing to the object of the Derived class then, only the destructor of the base class is invoked

- It does not invoke the destructor of the derived class, which leads to the problem of memory leak in our program and hence can result in undefined behavior.
- To correct this situation, the base class should be defined with a virtual destructor.
- Making base class destructor virtual guarantees that the object of derived class is destructed properly, i.e., both base class and derived class destructors are called.
- to make a virtual destrictor use virtual keyword preceded by a tilde(~) sign and destructor name inside the parent class.
- It ensures that first the child class's destructor should be invoked and then the destructor of the parent class is called.
- Note: There is no concept of virtual constructors in C++.

Advanced Typecasting Operators:

1. dynamic_cast
2. static_cast
3. const_cast
4. reinterpret_cast

1. dynamic_cast operator

- In case of polymorphic type, if we want to do downcasting then we should use dynamic_cast operator.
- dynamic_cast operator check type conversion as well as inheritance relationship between type of source and destination at runtime.
- In case of pointer if, dynamic_cast operator fail to do downcasting then it returns NULL.
- In case of reference, if dynamic_cast operator fail to do downcasting then it throws std::bad_cast exception.

2. static_cast operator

- If we want to do type conversion between compatible types then we should use static_cast operator.
- In case of non polymorphic type, if we want to do downcasting then we should use static_cast operator.
- In case of upcasting, if we want to access non overriden members of Derived class then we should do downcasting.
- static_cast operator do not check whether type conversion is valid or invalid. It only checks inheritance between type of source and destination at compile time.
- Risky conversion not be used, should only be used in performance-critical code when you are certain it will work correctly.
- The static_cast operator can be used for operations such as converting a pointer to a base class to a pointer to a derived class. Such conversions are not always safe.

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

3. const_cast operator

- Using constant object, we can call only constant member function.
- Using non constant object, we can call constant as well as non constant member function.
- If we want convert pointer to constant object into pointer to non constant object or reference to constant object into reference to non constant object then we should use const_cast operator.
- Used to remove the const, volatile, and __unaligned attributes.
- const_cast<class *> (this)->membername = value;

4. reinterpret_cast operator.

- If we want to convert pointer of any type into pointer of any other type then we should use reinterpret_cast operator.
- The reinterpret_cast operator can be used for conversions such as char* to int*, or One_class* to Unrelated_class*, which are inherently unsafe.

Exception Handling

- Following are the operating system resources that we can use in application development
    1. Memory
    2. File
    3. Thread
    4. Socket
    5. Network connection
    6. IO Devices etc.
- Since OS resources are limited, we should use it carefully.
- If we make syntactical mistake in a program then compiler generates error.
- Without definition, if we try to access any member then linker generates error.
- Logical error / syntacticaly valid but logicaly invalid statements represents bug.
- If we give wrong input to the application then it generates runtime error/exception.
- Exception is an object, which is used to send notification to the end user of the system if any exceptional situation occurs in the program.
- If we want to manage OS resources carefully then we should use exception handling mechanism.
- Need of exception Handling:
    1. To avoid resource leakage.
    2. To handle all the runtime errors(exeption) centrally.
- If we want to handle exception then we should use 3 keywords:
    1. try
    2. catch
    3. throw

1. try:

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

2. throw:

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

3. catch:

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

```
try
{
}
catch(...)
{
}
```

Nested Exception Handling

- We can write try catch block inside another try block as well as catch block. It is called nested try catch block.
- Outer catch block can handle excpetion's thrown from inner try block.
- Inner catch block, cannot handle exception thrown from outer try block.
- If information, that is required to handle exception is incomplete inside inner catch block then we can rethrow that exception to the outer catch block.

```
class ArithmeticException{
private:
    string message;
public:

    ArithmeticException( string message ) : message( message ){}
    void printStackTrace( void )const{
        cout<<this->message<<endl;
    }
};
int main( void ){
    try{
        try{
            throw ArithmeticException("/ by zero");
        }
```

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

Stack Unwinding

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

Friend function & class

- If we want to access private members inside derived class
- Either we should use member function(getter/setter).
- Or we should declare a facilitator function as a friend function.
- Or we should declare derived class as a friend inside base class.
- Friend function is non-member function of the class, that can access/modify the private members of the class.
- It can be a global function.
- Or member function of another class.
- Friend functions are mostly used in operator overloading.
- If class C1 is declared as friend of class C2, all members of class C1 can access private members of C2.
- Friend classes are mostly used to implement data struct like linked lists.

Manipulator

- It is a function which is used to format the output.
- Manipulators are of two types

1. Without arguments
2. With arguments

- Following are the manipulators in C++

1. endl
2. setw
3. fixed
4. scientific

5. setprecision
6. hex
7. dex
8. oct
9. left
10. right

- To use manipulators it is necessary to include header file.

Agenda

- Templates
- Shallow Copy and Deep Copy
- Copy Constructor
- STL

Document Link

```
https://en.cppreference.com/w/
```

Template

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

1. Function Template

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

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

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

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

2. Class Template

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

```
template<class T>
class Array // Parameterized type
{
    private:
    int size;
    T *arr;
    public:
    Array( void ) : size( 0 ), arr( NULL )
    {
    }
    Array( int size )
    {
    this->size = size;
    this->arr = new T[ this->size ];
    }
    void acceptRecord( void ){
    }
    void printRecord( void ){
    }
    ~Array( void ){ }
};
int main( void )
{
    Array<char> a1( 3 );
```

```
    a1.acceptRecord();
    a1.printRecord();
    return 0;
}
```

- Operator overloading
- Conversion Function

Copy Constructor

- Copy constructor is a parametered constructor of the class which take single parameter of same type but using reference.
- Copy constructor gets called in following conditions:

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

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

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

STL

- We can not divide template code into multiple files.

- Standard Template Library(STL) is a collection of readymade template data structure classes and algorithms.
- The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as lists, stacks, arrays, etc.
- It is a library of container classes, algorithms, and iterators.
- It is a generalized library and so, its components are parameterized.
- Working knowledge of template classes is a prerequisite for working with STL.
- STL has 4 components:

    1. Containers
    2. Algorithms
    3. Function Objects
    4. Iterators

1. Container

- Containers or container classes to store objects and data.
- The C++ container library categorizes containers into four types:

1. Sequence containers
2. Associative containers
3. Unordered associative containers
4. Sequence container adapters

```
STL
│
├── Containers
   ├── Sequence Containers
   │   ├── vector
   │   ├── deque
   │   ├── list
   │   └── forward_list
   ├── Associative Containers
   │   ├── set
   │   ├── multiset
   │   ├── map
   │   └── multimap
   ├── Unordered Associative Containers
   │   ├── unordered_set
   │   ├── unordered_multiset
   │   ├── unordered_map
   │   └── unordered_multimap
   └── Container Adapters
       ├── stack
       ├── queue
       └── priority_queue
```

2. Algorithm

- They act on containers and provide means for various operations for the contents of the containers.
    - Sorting
    - Searching

3. Functions

- The STL includes classes that overload the function call operator.
- Instances of such classes are called function objects or functors.
- Functors allow the working of the associated function to be customized with the help of parameters to be passed.

4. Iterators

- As the name suggests, iterators are used for working upon a sequence of values.
- They are the major feature that allows generality in STL.
- Iterators are used to point at the memory addresses of STL containers.
- They are primarily used in sequences of numbers, characters etc.
- They reduce the complexity and execution time of the program.

Sequence Containers

- Sequence containers are used for data structures that store objects of the same type in a linear manner.
- The STL Sequence Container types are:
- vector: A dynamic array that can grow and shrink in size. It provides fast random access to elements and efficient insertion and deletion at the end.
- deque: A double-ended queue that supports efficient insertion and deletion at both ends. It provides similar functionality to vector but may have better performance for inserting and deleting elements at the beginning.
- list: A doubly-linked list that allows for efficient insertion and deletion of elements at any position. It does not provide random access to elements and has slower traversal compared to vector and deque.
- forward_list: A singly-linked list that provides similar functionality to list but with reduced memory overhead. It allows for efficient insertion and deletion at the beginning and after an element.

Associative Containers

- Associative containers implement sorted data structures that can be quickly searched.
- set : collection of unique keys, sorted by keys
- map : collection of key-value pairs, sorted by keys, keys are unique
- multiset : collection of keys, sorted by keys
- multimap : collection of key-value pairs, sorted by keys

Container adaptors

- Container adaptors provide a different interface for sequential containers.

- stack : adapts a container to provide stack (LIFO data structure)
- queue : adapts a container to provide queue (FIFO data structure)

Vector

- Vectors are the same as dynamic arrays with the ability to resize itself automatically when an element is inserted or deleted, with their storage being handled automatically by the container.
- Vector elements are placed in contiguous storage so that they can be accessed and traversed using iterators.
- In vectors, data is inserted at the end. Inserting at the end takes differential time, as sometimes the array may need to be extended. Removing the last element takes only constant time because no resizing happens.
- When the vector needs to grow beyond its current capacity, it typically doubles its capacity.
- Inserting and erasing at the beginning or in the middle is linear in time.
- the iterator in the vector is a Random Access Iterator that Supports all iterator operations including arithmetic (e.g., +, -), comparison (<, >, etc.), and dereferencing (*, []).

Agenda

- STL
- File IO

map

- map is a sorted associative container that contains key-value pairs with unique keys.
- Keys are sorted by using the comparison function Compare.
- Search, removal, and insertion operations have logarithmic complexity.
- Maps are usually implemented as Red–black trees
- Iterators of map iterate in ascending order of keys, where ascending is defined by the comparison that was used for construction.
- Iterator of map are bidirectional iterator that supports dereferencing (*, ->) and bidirectional movement (++, --).

iterator

- An iterator in C++ is an object that enables traversal over the elements of a container (such as std::vector, std::list, std::map, etc.) and provides access to these elements.

- Iterators are a fundamental part of the Standard Template Library (STL) and are designed to abstract the concept of element traversal, making it possible to work with different containers in a consistent manner.

- Key Characteristics of Iterators:

1. Traversal: Iterators allow you to move through the elements of a container, one element at a time.
2. Access: Iterators provide access to the element they point to, typically through the dereference operator (*).
3. Type-Specific: Iterators are strongly typed, meaning that an iterator for an std::vector will be different from an iterator for an std::list.

Types of Iterators:

1. Input Iterators:

- Can read elements from a container. Only allow single-pass access (i.e., you can only move forward through the container).

2. Output Iterators:

- Can write elements to a container.
- Also allow only single-pass access.

3. Forward Iterators:

- Can read and write elements.
- Support multi-pass traversal, meaning you can go through the container multiple times.

4. Bidirectional Iterators:

- Can move both forward and backward in a container.
- Support all operations of forward iterators, with the additional ability to decrement the iterator.

5. Random Access Iterators:

- Provide all the capabilities of bidirectional iterators.
- Allow direct access to any element in the container using arithmetic operations like addition and subtraction.

Common Operations on Iterators:

- Dereferencing (*): Access the element the iterator points to.
- Incrementing (++): Move the iterator to the next element.
- Decrementing (--): Move the iterator to the previous element (not supported by input or output iterators).
- Equality/Inequality (==, !=): Compare iterators to check if they point to the same position.
- Addition/Subtraction (+, -): For random access iterators, allows moving the iterator by a specific number of elements.

Stream

- We give input to the executing program and the execution program gives back the output.
- The sequence of bytes given as input to the executing program and the sequence of bytes that comes as output from the executing program are called stream.
- In other words, streams are nothing but the flow of data in a sequence.
- The input and output operation between the executing program and the devices like keyboard and monitor are known as “console I/O operation”.
- The input and output operation between the executing program and files are known as “disk I/O operation”.
- The I/O system of C++ contains a set of classes which define the file handling methods
- These include ifstream, ofstream and fstream classes. These classes are derived from fstream and from the corresponding iostream class.
- These classes are designed to manage the disk files, are declared in fstream and therefore we must include this file in any program that uses files.
- Standard Stream Objects of C++ associated with console:
    1. cin -> Associated with Keyboard
    2. cout -> Associated with Monitor
    3. cerr -> Error Stream
    4. clog -> Logger Stream
- ifstearm is a derived class of istream class which is declared in std namespace. It is used to read record from file.
- ofstearm is a derived class of ostream class which is declared in std namespace. It is used to write record inside file.
- fstream is derived class of iostream class which is declared in std namespace. It is used to read/write record to/from file.


stream hirerachy

Classes for File stream operations

- ios:
    - ios stands for input output stream.
    - This class is the base class for other classes in this class hierarchy.
    - This class contains the necessary facilities that are used by all the other derived classes for input and output operations.
- istream :
    - istream stands for input stream.
    - This class is derived from the class ‘ios’.
    - This class handle input stream.
    - The extraction operator(>>) is overloaded in this class to handle input streams from files to the program execution.
    - This class declares input functions such as get(), getline() and read().
- ostream :
    - ostream stands for output stream.
    - This class is derived from the class ‘ios’.
    - This class handle output stream.
    - The insertion operator(<<) is overloaded in this class to handle output streams to files from the program execution.
    - This class declares output functions such as put() and write().
- ifstream :
    - This class provides input operations.
    - It contains open() function with default input mode.
    - Inherits the functions get(), getline(), read(), seekg() and tellg() functions from the istream.
- ofstream :
    - This class provides output operations.
    - It contains open() function with default output mode.
    - Inherits the functions put(), write(), seekp() and tellp() functions from the ostream.
- fstream :
    - This class provides support for simultaneous input and output operations.
    - Inherits all the functions from istream and ostream classes through iostream.

File Handling

- A variable is a temporary container, which is used to store record in RAM.
- A file is permanent container which is used to store record on secondry storage.
- File is operating system resource.
- Types of file:
    1. Text File

2. Binary File

1. Text File

1. Example : .txt,.doc, .docx, .rtf, .c, .cpp etc
2. We can read text file using any text editor.
3. Since it requires more processing, it is slower in performance.
4. If we want to save data in human readable format then we should create text file.

2. Binary File

1. Example : .mp3, .jpg, .obj, .class
2. We can read binary file using specific program/application.
3. Since it requires less processing, it is faster in performance.
4. It doesnt save data in human readable format.

File Modes in C++

- "w" mode
    - ios_base::out:
    - ios_base::out | ios_base::trunc
- "r" mode
    - ios_base::in
- "a" mode
    - ios_base::out | ios_base::app
    - ios_base::app
- "r+" mode
    - ios_base::in | ios_base::out
- "w+" mode
    - ios_base::in | ios_base::out | ios_base::trunc
- "a+" mode
    - ios_base::in | ios_base::out | ios_base::app
    - ios_base::in | ios_base::app:
- In case of binary use "ios_base::binary"
- In C++, files are mainly dealt by using three classes fstream, ifstream, ofstream available in fstream headerfile.
- ofstream: Stream class to write on files
- ifstream: Stream class to read from files

- fstream: Stream class to both read and write from/to files.

Serilization and DeSerilization in binary Files

- When working with string data types or other derived data types (like objects or pointers) in a class and writing or reading the data to/from a binary file, you need to handle serialization and deserialization properly.
- Directly reading or writing the object's memory representation as binary data may not work correctly for derived/user defined data types due to issues like memory layout, internal pointers, and dynamic memory allocation.
- To handle string data types (and other derived data types) correctly when reading or writing binary files, you should implement custom serialization and deserialization functions in your class.
- These functions should convert your object's data into a binary representation (serialization) and reconstruct the object from binary data (deserialization).

```
// Seralizing employee class with datamembers int id,string name,double salary.
void seralize(ofstream &fout)
{
    fout.write(reinterpret_cast<const char *>(&empid), sizeof(int));
    size_t length = name.size();
    fout.write(reinterpret_cast<const char *>(&length), sizeof(size_t));
    fout.write(name.c_str(), length);
    fout.write(reinterpret_cast<const char *>(&salary), sizeof(double));
}

//Deseralizing employee class
void deseralize(ifstream &fin)
{
  fin.read(reinterpret_cast<char *>(&empid), sizeof(int));
  size_t length;
  fin.read(reinterpret_cast<char *>(&length), sizeof(size_t));
  char *buffer = new char[length + 1];
  fin.read(buffer, length);
  buffer[length] = '\0';
  name = buffer;
  delete[] buffer;
  fin.read(reinterpret_cast<char *>(&salary), sizeof(double));
}
```

Agenda

- Operator Overloading
- Function Object
- Conversion Function

Operator Overloading

- operator is token in C/C++.
- It is used to generate expression.
- operator is keyword in C++.
- Types of operator:
    1. Unary operator
    2. Binary Operator
    3. Ternary operator
- Unary Operator:
    - If operator require only one operand then it is called unary operator.
    - example : Unary(+,-,*) , &, !, ~, ++, --, sizeof, typeid etc.
- Binary Operator:
    - If operator require two operands then it is called binary operator.
    - Example:
        1. Arithmetic operator
        2. Relational operator
        3. Logical operator
        4. Bitwise operator
        5. Assignment operator
- Ternary operator:
    - If operator require three operands then it is called ternary operator.
    - Example:
        - Conditional operator(? :)
- In C/C++, we can use operator with objects of fundamental type directly.(No need to write extra code).

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

- In C++, also we can not use operator with objects of user defined type directly.
- If we want to use operator with objects of user defined type then we should overload operator.

```
class Point
{
    int x;
    int y;
};
```

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

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

Limitations of operator overloading

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

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

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

    1. Assignment operator(=)
    2. Subscript / Index operator([])
    3. Function Call operator[()]
    4. Arrow / Dereferencing operator(->)

- Using operator overloading, we can change meaning of operator.

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

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

- If we want to overload, binary operator using member function then operator function should take only one parameter.
- Using operator overloading, we can not change, precedance and associativity of the operator.
- If we want to overload unary operator using member function then operator function should not take any parameter.

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

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

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

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

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

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

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

Overloading Insertion Operator(<<)

-cout is an external object of ostream class which is declared in std namespace.

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

```
//ostream out = cout; // NOT OK
ostream &out = cout; //OK
```

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

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

```
class ClassName
{
  friend ostream& operator<<( ostream &cout, ClassName &other );
};

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

Overloading Extraction Operator(>>)

- cin stands for character input. It represents keyboard.
- cin is external object of istream class which is declared in std namespace.
- istream class is typedef of basic_istream class.
- If we want to accept data/state of the variable/object from console/keyboard then we should use cin object and extraction operator.
- Copy constructor of istream class is private hence, we can not create copy of cin object in out program.
- To avoid copy, we should use reference.

```
istream in = cin; // NOT OK
istream &in = cin; // OK
```

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

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

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

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

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

Index/Subscript Operator Overloading

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

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

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

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

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

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

Overloading assignment operator.

- If we initialize newly created object from existing object of same class then copy constructor gets called.
- If we assign, object to the another object then assignment operator function gets called.

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

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

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

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

Overloading Call / Function Call operator:

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

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

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

```
    Complex c1;
    c1(10, 20); // c1.operator()( 10, 20 );
    c1.printRecord();
    return 0;
}
```

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

Conversion Function

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

1. Single Parameter Constructor

```
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.

2. Assignment operator function

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

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

3. Type conversion operator function.

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

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

Agenda

- Singleton class
- Factory Design Pattern
- nested class and local class
- constexpr
- noexcept
- nullptr
- nested namespace
- Smart Pointer

Singleton class

- It is a design pattern
- Design patterns are a standard solution to well-known problem
- It enables to use a single object of the class through out the application

Factory Design Pattern

- It is a creational design that offers a way to produce objects in a basesclass while letting derived classes change the kind of objects that are created.
- It is useful when there is a need to create multiple objects of the same type, but the type of the objects is not known until runtime.
- It is implemented using a factory function, which is a method that returns an object of the specified type.

Local Class

- If inside a function you declare a class then such classes are called as local classes.
- Inside local class you can access static and global members but you cannot access the local members declared inside the function where the class is declared.
- Inside local classes we cannot declare static data member however defining static member functions are allowed.
- As every local variable declared in function goes on its individual stack frame, such variables cannot be accessed in the local class functions.
- static and global variales gets space on data section which are designed to be shared.
- In a class static data members are designed to be accessed using classname and :: , the static member are designed to be shareed however in local classes we cannot access them ouside the function in ehich they are declared.
- Hence their is no purpose of keeping static data members inside local classes

Nested class

- A class declared inside another class is called as nested class
- Inside Nested class we can access private static members of outer class directly.
- A nested class can access all the private and public members of outer class directly on the outer class object

- Inside Nested class you can access static and global variables.
- As static and global variables are designed to shared they are accessiable inside the nested class.
- Data members gets the memory only when object is created and hence the nested class cannot access outer class data non static data members directly as they do not get memory inside them.

noexcept operator

- The noexcept operator performs a compile-time check that returns true if an expression is declared to not throw any exceptions.
- It can be used within a function template's noexcept specifier to declare that the function will throw exceptions for some types but not others.

```
void may_throw(); // it will throw exception
void no_throw() noexcept; // it will not throw exception

int main()
{
cout<<"may_throw() is noexcept(" << noexcept(may_throw())<<")"<<endl;
cout<<"no_throw() is noexcept(" << noexcept(no_throw())<<")"<<endl;
}

// output
// may_throw() is noexcept(false)
// no_throw() is noexcept(true)
```

constexpr

- constexpr is a feature added in C++ 11.
- The constexpr specifier declares that it is possible to evaluate the value of the function or variable at compile time.
- Such variables and functions can then be used where only compile time constant expressions are allowed (provided that appropriate function arguments are given).
- A constexpr specifier used in an object declaration or non-static member function(until C++14) implies const.
- A constexpr specifier used in a function or static data member(since C++17) declaration implies inline.

constexpr vs inline

- The constexpr keyword is used to declare that a function or variable can be evaluated at compile-time.
- It guarantees that the value or result of the function can be computed at compile-time if all arguments are known at compile-time.
- constexpr functions must have a return type that is literal and must consist of a single return statement.
- Variables declared as constexpr must be initialized by constant expressions.
- constexpr functions can be used in contexts where constant expressions are required, such as array sizes, template arguments,etc.

- The inline keyword is used to suggest to the compiler that a function should be expanded in place at each call site rather than being called like a regular function.
- It is primarily used to improve performance by reducing the overhead of function calls, especially for small, frequently called functions.
- The compiler may choose to ignore the inline keyword and not inline the function if it determines that inlining would not be beneficial.
- inline functions may still have a separate definition in a translation unit, and they can contain any code that a regular function can have.

Rules for constexpr

- In C++ 11, a constexpr function should contain only one return statement. C++ 14 allows more than one statement.
- constexpr function should refer only to constant global variables.
- constexpr function can call only other constexpr functions not simple functions.
- The function should not be of a void type.

nullptr

- A nullptr is a keyword introduced in C++11 to represent a null pointer.
- It provides a type-safe and clearer alternative to using NULL or 0 for null pointer constants.
- It's recommended to use nullptr in modern C++ code.
- nullptr helps avoid ambiguities in function overloading and template specialization scenarios.

```
void f1(int *n)
{
    cout << "Function with int* " << endl;
}
void f1(int n)
{
    cout << "Function with int " << endl;
}

int main()
{
    int *ptr1 = 0;
    int *ptr2 = NULL;
    int *ptr3 = nullptr;
    cout << "ptr1 = " << ptr1 << endl;
    cout << "ptr2 = " << ptr2 << endl;
    cout << "ptr3 = " << ptr3 << endl;

    f1(0); // fun with int
    // f1(NULL); // ambugity
    f1(nullptr); // fun with int*
```

```
    return 0;
}
```

- Smart pointers are objects that behave like pointers but provide automatic memory management.
- Smart pointers enable automatic, exception-safe, object lifetime management.
- They help prevent memory leaks and manage the lifetime of dynamically allocated objects.
- Smart pointers are part of the C++ Standard Library and are implemented as template classes.
- The main smart pointers in C++ are:

1. std::unique_ptr

- `std::unique_ptr` is a smart pointer that owns a dynamically allocated object and ensures that the object is deleted when the `unique_ptr` goes out of scope or is reset.
- It is unique in that it cannot be copied or shared. It can only be moved.
- It is lightweight and efficient because it does not incur the overhead of reference counting.
- The object is disposed of, using the associated deleter when either of the following happens:
    1. the managing unique_ptr object is destroyed.
    2. the managing unique_ptr object is assigned another pointer via operator= or reset().

2. std::shared_ptr

- `std::shared_ptr` is a smart pointer that retains shared ownership of an object through a pointer.
- Several shared_ptr objects may own the same object.
- The object is destroyed and its memory deallocated when either of the following happens:
    1. the last remaining shared_ptr owning the object is destroyed;
    2. the last remaining shared_ptr owning the object is assigned another pointer via operator= or reset().
- It manages the ownership of a dynamically allocated object using reference counting.

3. std::weak_ptr

- `std::weak_ptr` is a smart pointer that provides a non-owning reference to an object managed by `std::shared_ptr`.
- It does not participate in reference counting and does not keep the object alive.
- It is used to break cyclic dependencies between `std::shared_ptr` objects.
- It must be converted to `std::shared_ptr` in order to access the referenced object.
- It models temporary ownership when an object needs to be accessed only if it exists, and it may be deleted at any time by someone else