Object Oriented Programming Using C++

Day 1

Quick Review of C programming language

History

- Inventor: Dennis Ritchie
- Location: At&T Bell Lab
- Development Year: 1969-1972
- Operating System: Unix
- Hardware: PDP-11
- C is statically type checked as well as strongly type checked language.
- C is a general purpose programming language.
- Extension: .c
- Standardization: ANSI
    - C89
    - C95
    - C99
    - C11
    - C17
    - C23

Data Type

- Data Type Describe following things:
    - Size: How much memory is required to store the data.
    - Nature: Which type of data is allowed to stored inside memory
    - Operation: Which operations are allowed to perform on the data stored inside memory
    - Range: How much data is allowed to store inside memory
- Types:
    - Fundamental Data Types(5)
        - void
        - char
        - int
        - float
        - double
    - Derived Data Types
        - Array
        - Function
        - Pointer
    - User Defined Data Types
        - Structure

        - Union
- Type Modifiers
    - short
    - long
    - signed
    - unsigned
- Type Qualifiers
    - const
    - volatile

Entry Point Function

- According to ANSI specification, entry point function should be "main".
- Syntax: 1

```
int main( int argc, char *argv[ ], char *envp[ ] ){
  return 0;
}
```

- Syntax: 2

```
void main( int argc, char *argv[ ], char *envp[ ] ){

}
```

- Syntax: 3

```
int main( int argc, char *argv[ ] ){
  return 0;
}
```

- Syntax: 4

```
void main( int argc, char *argv[ ] ){

}
```

- Syntax: 5

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

- Syntax: 6

```
void main( void ){

}
```

- Syntax: 7

```
void main(  ){

}
```

- main is user defined function.
- Calling main function is a responsibility of operating system. Hence it is called as callback function.
- main function must be global function.
- We can define only one main function per project. If we do not define main function then linker generates error.

Software Development Kit

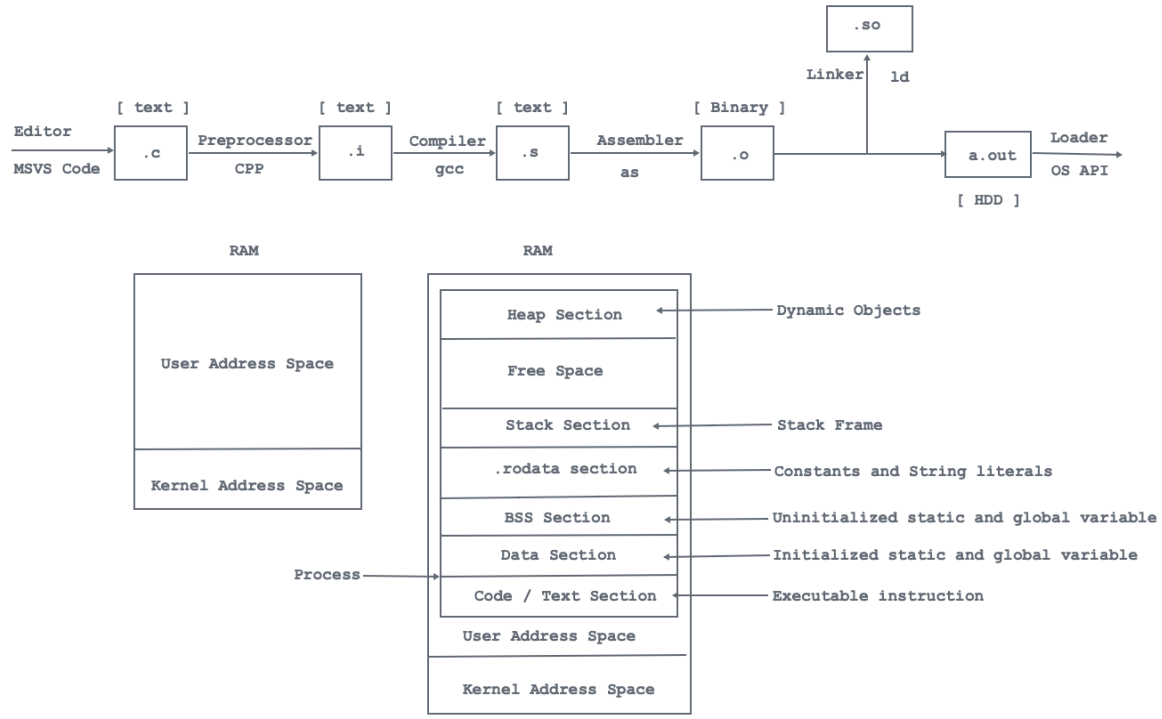
- SDK = Development tools + Documentation + Runtime Environment + Supporting Libraries
- Development tools
  - Editor
    - It is used to create/edit source file(.c/.cpp)
    - Example:
      - MS Windows: Notepad, Notepad++, Edit Plus, MS Visual Studio Code, Wordpad etc.
      - Linux: vi, vim, TextEdit, MS Visual Studio Code etc.
      - Mac OS: vi, vim, TextEdit, MS Visual Studio Code etc.
  - Preprocessor
    - It is a system program whose job is:
      - To remove the comments
      - To exapand macros
    - Example: CPP(C/C++ Pre Processor)
    - Preprocessor generates intermediate file(.i /.ii)
  - Compiler
    - It is a system program whose job is:
      - To check syntax

        - To convert high level code into low level(Assembly code)
    - Example:
        - Turbo C: tcc.exe
        - MS Visual Studio: cl.exe
        - Linux: gcc
    - Compiler generates .asm / .s file.
- Assembler:
    - It is a system program which is used to convert low level code into machine level code.
    - Example:
        - Turbo C: Tasm
        - MS Visual Studio: Masm
        - Linux: as
    - It generates .obj / .o file.
- Linker
    - It is a program whose job is to link machine code to library files.
    - It is responsible for generating executable file.
    - Example:
        - Turbo C: Tlink.exe
        - MS Visual Studio: link.exe
        - Linux: ld
- Loader:
    - It is an OS API.
    - It is used to load executable file from HDD into primary memory(RAM).
- Debugger:
    - Logical error is also called as bug.
    - To find the bug we should use debugger
    - Example
        - Linux: gdb, ddd

- Documentation
    - It can be in the form of html / pdf / text format.
    - Example: https://en.cppreference.com/w/c/language
- Runtime Environment
    - It is responsible for managing execution of application
    - Example: C Runtime

Flow Of Execution

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



Comments

- If we want to maintain documentation of the source code then we should use comments.
- Comments in C/C++
  - Single Line Comment

```
//This is single line comment
```

  - Multiline / Block Comment

```
/*
  This is multiline comment
*/
```

- "-save-temps" Save intermediate compilation results

Local Function Declaration

```
int main( void ){//Calling Function
  int sum( int num1, int num2 );  //Local Function Declaration: OK
  int result = sum( 10, 20 ); //Function Call
  return 0;
}
int sum( int num1, int num2 ){  //Called Function
```

```
  int result = num1 + num2;
  return result;
}
```

Global Function Declaration

```
int sum( int num1, int num2 );  //Local Function Declaration: OK
int main( void ){//Calling Function
  int result = sum( 10, 20 ); //Function Call
  return 0;
}
int sum( int num1, int num2 ){  //Called Function
  int result = num1 + num2;
  return result;
}
```

Function Definition as a Declaration

```
//Treated as declaration as well as definition
int sum( int num1, int num2 ){  //Called Function
  int result = num1 + num2;
  return result;
}
int main( void ){//Calling Function
  int result = sum( 10, 20 ); //Function Call
  return 0;
}
```

Linker Error

- Without definition, If we try to use function then linker generates error.

```
int sum( int num1, int num2 );  //Function Declaration
int main( void ){//Calling Function
  int result = sum( 10, 20 ); //Function Call
  return 0;
}
//Output: Linking Error
```

Argument versus Parameter

- During function call, if we use variable or constant value then it is called as argument.
- Example 1

```
int main( void ){
  int result = sum( 10, 20 );   //Here 10 and 20 are arguments
  return 0;
}
```

- Example 2

```
int main( void ){
  int num1 = 50;
  int num2 = 60;
  int result = sum( num1, num2 );   //Here num1 and num2 are arguments
  return 0;
}
```

- Example 3

```
int main( void ){
  int num1 = 110;
  int result = sum( num1, 120 );   //Here num1 and 120 are arguments
  return 0;
}
```

- During function definition, if we use variables then it is called as function parameter or simply parameter.
- Example 1:

```
//Here num1 and num2 are parameters
int sum( int num1, int num2 ){
  int result = num1 + num2;
  return result;
}
```

Declaration and Definition

- Declaration refers to the term where only nature of the variable is stated but no storage is allocted.
- Definition refers to the place where memory is assigned / allocated.
- Example 1

```
int main( void ){
  //Uninitialized non static local variable
  int num1; //Declaration as well as definition
```

```
  return 0;
}
```

- Example 2

```
int main( void ){
  //Initialized non static local variable
  int num1 = 10; //Declaration as well as definition
  return 0;
}
```

- Example 3

```
  //Initialized non static global variable
int num1 = 10; //Declaration as well as definition
int main( void ){
  printf("Num1  : %d\n", num1);
  return 0;
}
```

- Example 4

```
int main( void ){
  extern int num1;  //Declaration
  printf("Num1  : %d\n", num1);
  return 0;
}
//Initialized non static global variable
int num1 = 10; //Declaration as well as definition
```

- Example 5

```
int main( void ){
  extern int num1;  //Declaration
  printf("Num1  : %d\n", num1); //Linker Error
  return 0;
}
```

Initialization and Assignment

- During declaration, process of storing value inside variable is called as initialization.
- Consider example:

```
int number = 10;  //Initialization
```

- We can do initialization of variable only once.

```
int number = 10;  //Initialization: OK
int number = 20;  //Not OK
```

- After declaration, process of storing value inside variable is called as assignment.
- Example 1:

```
int number;
number = 10;  //Assignment
```

- Example 2:

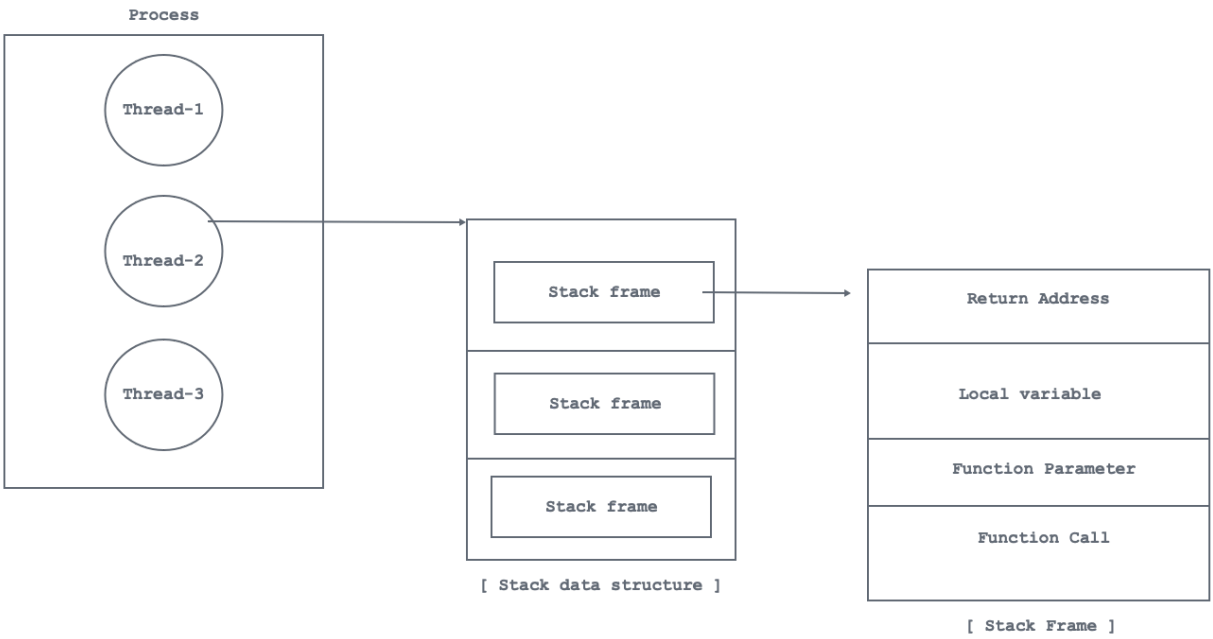
```
int number = 10;  //Initialization
number = 20;  //Assignment
```

- We can do assignment multiple times.
- Example 3:

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

Day 2

Function Activation Record



Pointer

- Variable Definition:
  - An entity whose value can be change is called as variable.
  - Named memory location / name given to memory location is called as variable.
  - Variable is also called as identifier.
- Assignement:
  - Identify the rules for variable/identifier name.
- Pointer is a variable which is designed to store address of another variable.
- Size of pointer:
  - 16-bit : 2 bytes
  - 32-bit : 4 bytes
  - 64-bit : 8 bytes
- Pointer Declaration:
  - Example 1

```
int* ptrNumber; //OK
```

  - Example 2

```
int * ptrNumber; //OK
```

  - Example 3

```
int *ptrNumber; //OK: Recommended
```

- Example 4

```
int main( void ){
  //Uninitialied non static local pointer variable
  int *ptrNumber; //Wild Pointer
  return 0;
}
```

- Uninitialied pointer is called as wild pointer.
- NULL is a macro whose value is 0.

```
#define NULL 0
```

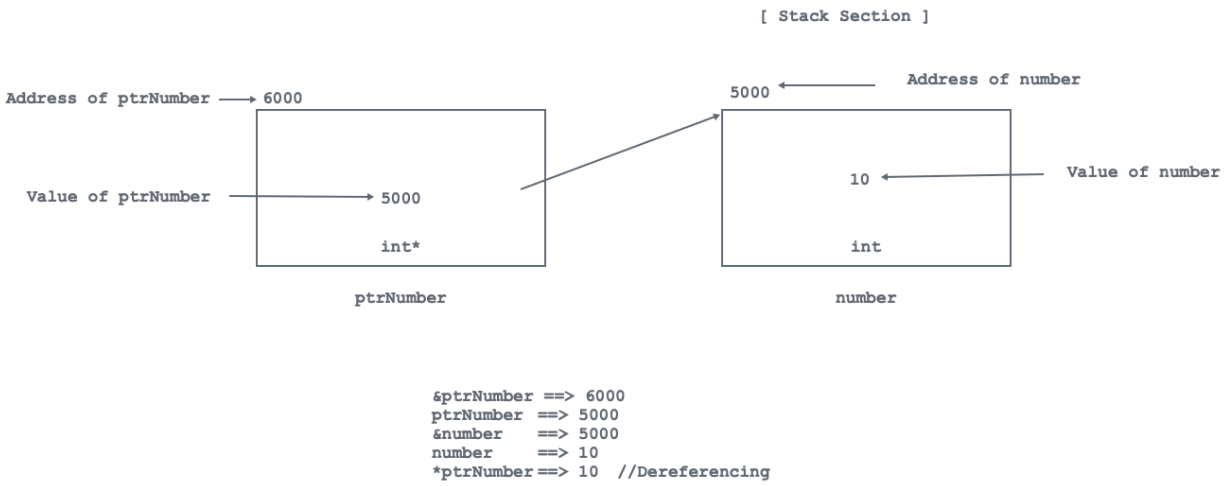
- To initializer pointer or to avoid dangling pointer we should use NULL;
  - Example 4

```
int main( void ){
  //NULL is a macro
  int *ptrNumber = NULL;
  //ptrNumber is a NULL pointer
  return 0;
}
```

- If pointer contains NULL value then it is called as Null pointer

- Pointer Initialization

```
int number = 10;  //Initialzation
int *ptrNumber = &number; //Initialization
//How will you print value 10
printf("Value : %d\n", number);
printf("Value : %d\n", *ptrNumber); //10
```

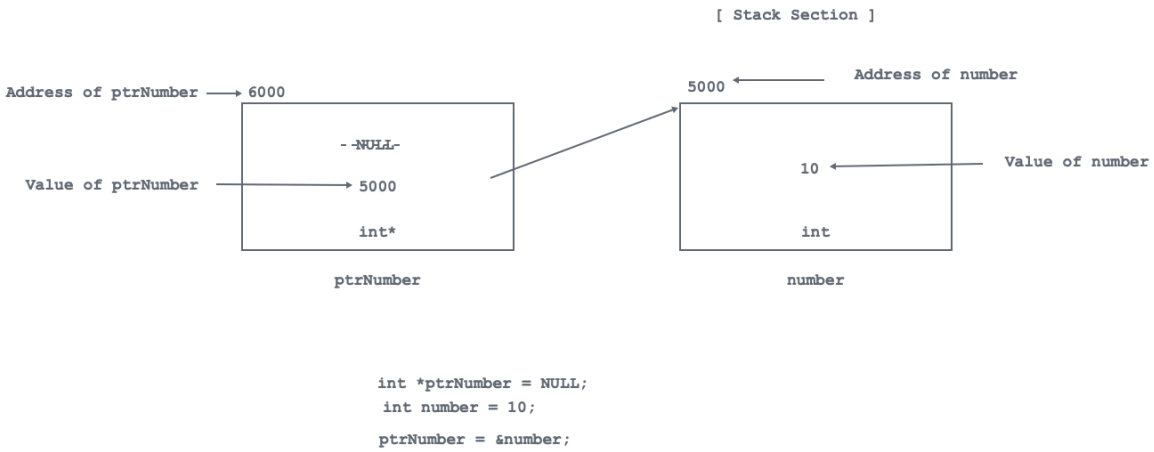



```
&ptrNumber ==> 6000
ptrNumber  ==> 5000
&number    ==> 5000
number     ==> 10
*ptrNumber ==> 10  //Dereferencing
```

- Pointer Assignment

```c
int *ptrNumber = NULL; //Initialzation
int number = 10;  //Initialzation
ptrNumber = &number;  //Assignment
//How will you print value 10
printf("Value : %d\n", number);
printf("Value : %d\n", *ptrNumber); //10
```

- We should not derefer Null pointer. Behaviour will be unpredictable.



Constant Qualifier

- const is a keyword in C/C++ and it is consider as type qualifier.
- Example 1

```
#include<cstdio>
int main( void ){
  int number = 10;  //Initialization
  printf("Number    :   %d\n", number); //10
  number = number + 5;
  printf("Number    :   %d\n", number); //15
  return 0;
}
```

- If we dont want to modify value of the variable then we should use const qualifier.
- Example

```
#include<cstdio>
int main( void ){
  const int number = 10;    //Initialization
  printf("Number    :   %d\n", number); //10
  //number = number + 5;    //Not OK
  return 0;
}
```

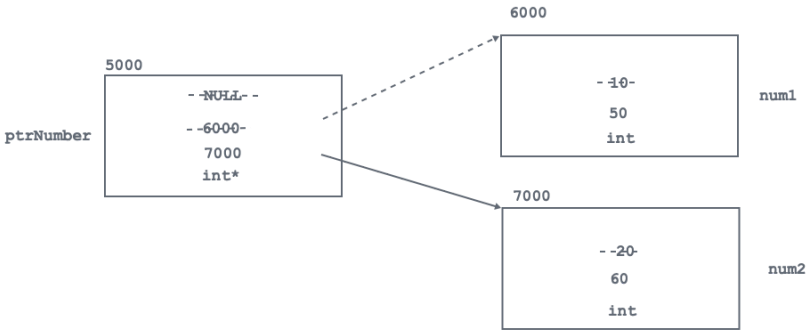
- We can not modiy value of constant variable but we can read its value. Hence it is called as read-only variable.

Constant and Pointer combinations

int *ptrNumber

- Here ptrNumber is non constant pointer variable which can store address of non constant integer variable.
- Example:

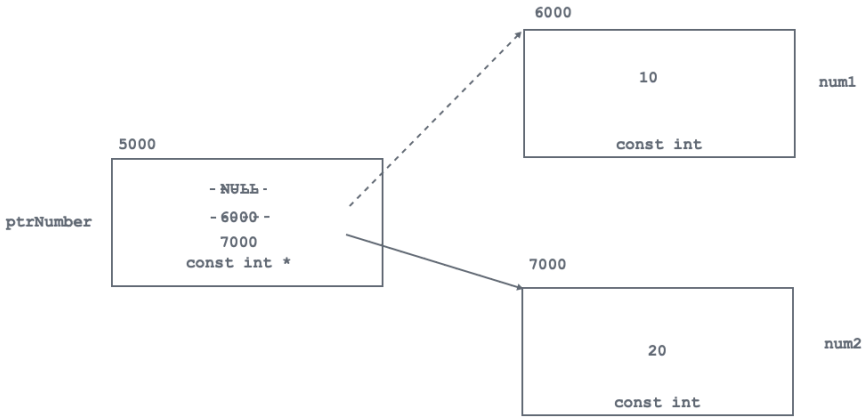
```c
int main( void ){
  int *ptrNumber = NULL;

  int num1 = 10;
  ptrNumber = &num1;
  //num1 = 50;  //OK
  *ptrNumber = 50;  //Dereferencing

  printf("Num1  :   %d\n", num1);   //50
  printf("Num1  :   %d\n", *ptrNumber); //50: Dereferencing


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



const int *ptrNumber

- Here ptrNumber is non constant pointer variable which can store address of constant integer variable.
- Example:

```c
int main( void ){
  const int *ptrNumber = NULL;  //OK

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

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



int const *ptrNumber

- const int *ptrNumber and int const *ptrNumber are same.

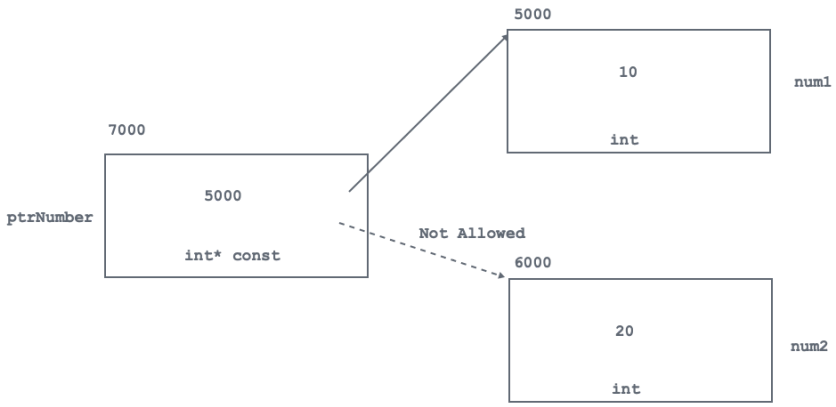
const int const *ptrNumber

- const int *ptrNumber, int const *ptrNumber and const int const *ptrNumber are same.
- warning: duplicate 'const' declaration specifier

int *const ptrNumber

- Here, ptrNumber is constant pointer variable, which can store address of non constant integer variable.

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

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



int *ptrNumber const

- It is invalid syntax.

const int *const ptrNumber

- Here ptrNumber is constant pointer variable which can store address of constant integer variable.
- Example:

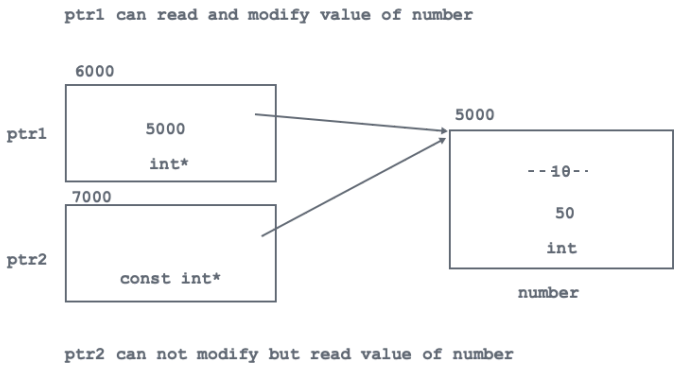
```
int main( void ){
  const int num1 = 10;  //OK
  const int *const ptrNumber = &num1;

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

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

int const *const ptrNumber

- const int *const ptrNumber and int const *const ptrNumber are same.

Consider following Pointer Example

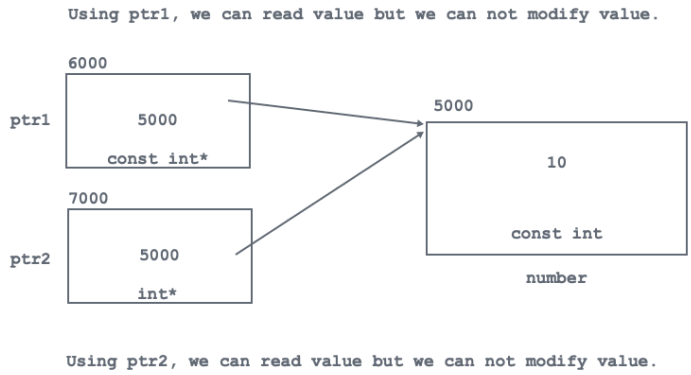
```
int main( void ){
    int number = 10;
    int *ptr1 = &number;
    *ptr1 = 50; //OK: Dereferencing
    printf("Number  :   %d\n", number); //50
    printf("Number  :   %d\n", *ptr1);  //50: Dereferencing

    printf("---------\n");

    const int *ptr2 = &number;
    //*ptr2 = 60;   //Not OK
    printf("Number  :   %d\n", number); //50
    printf("Number  :   %d\n", *ptr2);  //50: Dereferencing
    return 0;
}
```



Consider following Pointer Example

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

    const int *ptr1 = &number;
    //*ptr1 = 50;   //Not OK
    printf("Number  :   %d\n", number);//10
```

```
    printf("Number  :   %d\n", *ptr1);//10: Dereferencing

    printf("---------\n");

    int *ptr2 = (int *)&number;
    *ptr2 = 50;
    printf("Number  :   %d\n", number);//10
    printf("Number  :   %d\n", *ptr2);//50: Unexpected behavior
    return 0;
}
```



Lab Assignment

- Write a menu driven program to test accept/print employee record.
- Define structure:
  - Employee:
    - name: char[30]
    - empid: int
    - salary: float
- Create object and test the functionality
  - int main(void)
  - void accept_record(struct Employee *ptr);
  - void print_record(struct Employee *ptr);

Structure

- Structure is derived data type in C/C++. But generally it is called as user defined data type.

- If we want to group related data elements together then we should use structure.
- Consider below examples
    - name:char[30], empid:int, salary:float: Employee
    - number:int, balance:float, type:char[30]: BankAccount
    - day:int, month:int, year:int: Date
    - hour:int, minute:int, second:int : Time
    - red:int , green:int, blue:int : Color
- struct is keyword in C/C++.
- To declare structure and to create object of the structure we must use struct keyword.
- Example 1:

```
struct Employee{
  char name[ 30 ];  //structure member
  int empid;  //structure member
  float salary;  //structure member
};
struct Employee emp;
//struct Employee : Type Name
//emp: object
```

- If we want to give another name to the existing data type then we should use typedef.
- typedef is a keyword.
- Example 2:

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

- Example 3:

```
typedef struct Employee{
  char name[ 30 ];  //structure member
  int empid;  //structure member
  float salary;  //structure member
}Employee_t;
```

```
  struct Employee emp1; //OK
  Employee_t emp2;  //OK
```

- Consider following example:

```
int main( void ){
  char name[ 30 ];
  int empid;
  float salary;

  printf("Name    :   ");
  scanf("%s",name);
  printf("Empid   :   ");
  scanf("%d",&empid);
  printf("Salary  :   ");
  scanf("%f", &salary);


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

  //printf("%-30s%-5d%-10.2f\n", name, empid, salary);
  return 0;
}
```

- Consider following example:

```
int main( void ){
  //Local structure
  struct Employee{
    char name[ 30 ];
    int empid;
    float salary;
  };

  struct Employee emp;
  //struct Employee: Data type
  //emp: object

  printf("Name  :   ");
  scanf("%s",emp.name);
  printf("Empid :   ");
  scanf("%d",&emp.empid);
  printf("Salary    :   ");
  scanf("%f", &emp.salary);


  printf("Name  :   %s\n", emp.name);
```

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

  //printf("%-30s%-5d%-10.2f\n", name, empid, salary);
  return 0;
}
```

- We can declare structure inside function. It is called as local structure.
- We can not create object/pointer of local structure outside function.
- If we create, object of the structure then all the members declared inside structure get space inside object.
- Using object, If we want to access members of structure then we should use dot / member selection operator.
- Using pointer, If we want to access members of structure then we should use arrow operator.
- Consider following example:

```
int main( void ){
  //Local structure
  struct Employee{
    char name[ 30 ];
    int empid;
    float salary;
  };

  struct Employee emp;
  struct Employee *ptr = &emp;

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


  printf("Name  :   %s\n", ptr->name);
  printf("Empid :   %d\n", ptr->empid);
  printf("Salary    :   %f\n", ptr->salary);
  return 0;
}
```

- Reference: https://grandidierite.github.io/structure-alignment-and-packing-in-C-programming/

Day 3

Limitations with C programming languages

- In C languages, all the functions are global. Any global function can access any global data. Hence achieving data security is difficult.
- There is no string data type in C hence string memory managment is difficult
- If number of lines gets increased then code management becomes difficult.

C++ History

- Inventor: Bjarne Stroustrup
- Development Year: 1979
- Initial name : C with Classes
- Renamed in 1983 by ANSI: C++
- Standardization: ISO Working Group
- C++ Standards:
  - C++98
  - C++03
  - C++11
  - C++14
  - C++17
  - C++20
  - C++23
  - C++26
- C++ is object oriented programming language.
- C++ is derived from C and Simula(First object oriented programming language).
- C++ is also called as hybrid programming language.
- C++ is statically as well as strongly type checked language.

Data Types

- Fundamental Data Types(7)
  - void
  - bool
  - char
  - wchar_t (typedef unsigned short wchar_t)
  - int
  - float
  - double
- Derived Data Types(4)
  - Array
  - Function
  - Pointer
  - Reference
- User Defined Data Types(3)
  - Structure
  - Union
  - Class

Type Modifiers

- short
- long
- signed
- unsigned

Type Qualifiers

- const
- volatile

Execution Flow

- cfront is translator developed by Bjarne strostrup. It was used to convert C++ source code into C source code.
- Name of the C++ compiler for linus is g++.

Access Specifier

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

Access Specifier	Same Class	Derived Class	Outsid Class / Global funtion
private	A	NA	NA
protected	A	A	NA
public	A	A	A

Structure in C++

- We can define function inside structure.
- To create object of structure keyword struct is optional.
- Structure members are by default considered as public.

- Structure is not an object oriented concept.

What is the difference between structure and class?

- structure members are by default public whereas class members are by default private.

Data Member

- Variable declared inside class / structure is called as data member.

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

- Data member is also called as property / field / attribute.

Member Function

- A function implemented / defined inside class / structure is called as member function.

```
class Employee{
public:
  void accept_record( void ){   //Member function
    printf("Name    :   ");
    scanf("%s", name );
    printf("Empid   :   ");
    scanf("%d", &empid );
    printf("Salary  :   ");
    scanf("%f", &salary );
  }

  void print_record( void ){    //Member function
    printf("Name    :   %s\n", name);
    printf("Empid   :   %d\n", empid);
    printf("Salary  :   %f\n", salary);
  }
};
```

- Member function is also called as method / operation / behaviour / message
- Member function of the class which is having body is called as concrete method.
- Member function of the class which do not have body is called as abstract method.

Class

- A class is collection of data member and member function.
- Inside class, we can define:
    - Nested type
        - enum
        - union
        - structure
        - class
    - Data member
        - non static
        - static
    - Member function
        - static
        - non static
            - const
            - virtual
    - Constructor
    - Destructor
- A class from which we can create object/instance is called as concrte class.
- A class from which we can not create object/instance is called as abstract class.

Object

- Variable of a class is called as object.
- Object is also called as instance.

```
class Employee emp1; //OK

Employee emp; //OK
```

- Process of creating object from class is called as instantiation;
    - C:
        - struct Structure_Name object_name;
    - C++
        - Structure_Name object_name;
        - Class_Name object_name;
    - Java:
        - Class_name reference_name = new Class_name( );

```
Employee emp;   //Here class Employee is instantiated and name of the
instance is emp.
```

Message Passing

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

```
int main( void ){
  Employee emp; //Here class Employee is instantiated and name of the
instance is emp.

  emp.acceptRecord( );  //acceptRecord() function is called  on object
emp;

  emp.printRecord( );       //printRecord() function is called  on
object emp;

  return 0;
}
```

- Consider following code:

```
int main( void ){
  Employee emp;

  //:: is called as scope resolution operator

  emp.Employee::acceptRecord( );    //OK

  emp.Employee::printRecord( ); //OK

  return 0;
}
```

Syntax to define member function global

```
ReturnType ClassName::functionName( ){
  //TODO
}
```

Header guard / Include guard

- If we want to expand contents of header file only once then we should use Header guard inside header file.

```
#ifndef EMPLOYEE_H_
#define EMPLOYEE_H_
  //TODO: Declaration
#endif /* EMPLOYEE_H_ */
```

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

- Standard directory for standard header file : C:\MicGW\include
- If we include header file in angular bracket(< >) then preprocessor try to locate that file inside standard directory only.
- Example: #include<stdio.h>
- If we include header file in double quotes(" ") then preprocessor first try to locate that file inside current project directory. If not found then it will try to locate it from standard directory.
- Example:
  - #include<stdio.h>
  - #include"stdio.h"

Storage Classes

- In C/C++ there are 4 storage classes:
  - auto
  - register
  - static
  - extern
- Storage class decide scope and lifetime of the elements

Scope

- Scope of the variable / function describes area / region / boundry where we can access it.
- Scope in C
  - Block Scope
  - Function Scope
  - Function Prototype Scope
  - File Scope
- Consider below example:

```
int num4 = 10; //File Scope
static int num3 = 20; //File Scope
int sum( int num1, int num2 ){  //Function Prototype Scope
  return num1 + num2;
}
int main( void ){
  int count;  //Function Scope
  for(  count = 1; count <= 10; count ++ ){
    int temp = 0; //Block Scope
    //TODO
  }
  return 0;
}
```

- Scope in C++
  - Block Scope
  - Function Scope

- Function Prototype Scope
- Enumeration Scope
- Class Scope
- Namespace Scope
- File Scope
- Program Scope

What is the difference between non static global variable and static global variable?

- We can access non static global variable inside same file where it is declared as well as inside diffrent file using extern keyword.
- We can access static global variable inside same file where it is declared. But we can not access it inside diffrent file. We will get linker error.

Lifetime

- Lifetime describes time i.e how long object will be exist inside memory.
- Lifetime in C/C++
  - Automatic Lifetime
    - All the local variables are having automatic lifetime.
  - Static Lifetime
    - All the static and global variables are having static lifetime
  - Dynamic Lifetime
    - All the dynamic objects are having dynamic lifetime.

Namespace

- We can not give same name to the multiple variables inside same scope.
- We can give same name to the local variable as well as global variable.
- If name of the local variable and global variable are same then preference will be given to the local variable. Consider below code:

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

- Using scope resolution operator, we can use value of global variable inside program.

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

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

- Consider below code:

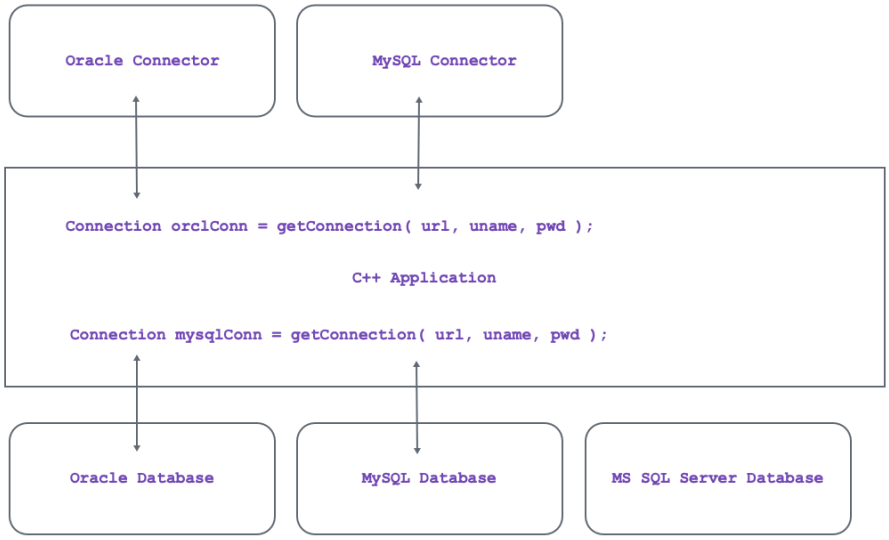
```
int num1 = 10;  //Global Variable
int main( void ){
  int num1 = 20;    //Local variable
  printf("Num1  :   %d\n", ::num1); //10
  printf("Num1  :   %d\n", num1);   //20

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

- We can use scope resolution operator with function too.

```
void print_message( ){
  printf("Good Evening!!\n");
}
int main( void ){
  print_message( ); //OK

  ::print_message( );   //OK
  return 0;
}
```

- Consider below code:

```
int num1 = 10;  //OK
int num1 = 20;  //error: redefinition of 'num1'
int main( void ){
  int num2 = 30;  //OK
  //int num2 = 40;  //error: redefinition of 'num2'
  return 0;
}
```



- Namespace is a C++ feature which is designed:
  - to avoid name clashing / conflict / collision / ambiguity.
  - to group/organize functionally equivalent / related types together.
- namespace is a keyword in C++.
- Example 1:

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

- Example 2:

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

```
  return 0;
}
```

- Example 3:

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

- Example 4:

```
namespace na{
  int num1 = 10;
  int num2 = 20;
}
namespace nb{
  int num1 = 30;
  int num3 = 40;
}

int main( void ){
  printf("Num1  :   %d\n", na::num1);   //OK: 10
  printf("Num2  :   %d\n", na::num2);   //OK: 20

  printf("Num1  :   %d\n", nb::num1);   //OK: 30
  printf("Num3  :   %d\n", nb::num3);   //OK: 40
  return 0;
}
```

- Example 5:

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

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

- We can not define namespace inside block scope / function scope or class scope. Namespace definition must appear in either namespace scope or file/program scope.

```
int main( void ){
  namespace na{ //error: namespaces can only be defined in global or
namespace scope
    int num1 = 10;
  }
  return 0;
}
```

- Example 6:

```
int num1 = 10;

//File Scope
namespace na{
  int num2 = 20;

  //Namespace scope
  namespace nb{ //Nested namespace
    int num3 = 30;
  }
}

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

- If we define variable/function/class without namespace globally then it is considered as a member of global namespace.
- If we dont want to use namespace name and :: operator every time then we should use using directive.
- Example 7:

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

- Example 8:

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

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

- Example 9:

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

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

  using namespace nb;
  //printf("Num1    :   %d\n", num1 );  //error: reference to 'num1'
is ambiguous
  printf("Num1  :   %d\n", nb::num1 );  //10
  return 0;
}
```

- Example 10:

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

  ::print_record( );

  ::display_record( );
  return 0;
}
```

- Example 11:

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

  ::print_record( );

  ::display_record( );
  return 0;
}
```

- Example 12:

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

  printf("Num1  :   %d\n", num1);
}
void display_record( ){

  printf("Num1  :   %d\n", num1);
}
int main( void ){
  ::show_record( );

  ::print_record( );

  ::display_record( );
  return 0;
}
```

- Except main function, we can declare any member inside namespace.
- Example 13:

```
namespace na{
int num1 = 10;
}
using namespace na;
namespace nb{
  void show_record( ){
    printf("Num1    :   %d\n", num1);
  }
  void print_record( ){

    printf("Num1    :   %d\n", num1);
  }
  void display_record( ){

    printf("Num1    :   %d\n", num1);
  }
}
int main( void ){
  nb::show_record( );

  nb::print_record( );

  nb::display_record( );
```

```
  return 0;
}
```

- Example 14:

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

Day 4

- Variable is a container which is used to store data in RAM.
- File is a container which is used to store data in HDD.
- Stream is an abstraction(object), which either produce(write) or consume(read) inform from source to destination.
- Console is also called as terminal = Keyboard + Monitor / Printer.
- In C, Standard stream objects associated with Console:
  - stdin
    - Standard input stream associated with keyboard which is used to read data.

```
scanf("%d", &number);
//same as
fscanf( stdin, "%d", &number );
```

  - stdout
    - Standard output stream associated with monitor which is used write data.

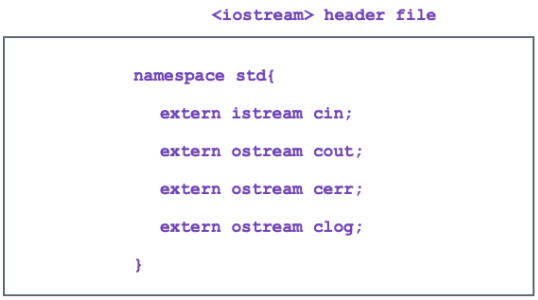
```
printf("%d", number);
//same as
fprintf(stdout, "%d", number);
```

  - stderr

- Standard output stream associated with monitor which is used write error.

```
fprintf(stderr, "Array index out of bounds.");
```

- In C++, Standard stream objects associated with Console:
  - cin
  - cout
  - cerr
  - clog



- std is a standard namespace of C++ which is declared in header file.
- cin, cout, cerr and clog are external objects declared in std namespace. Hence to use it we should use std::cin, std::cout, std::cerr, std::clog.

Character Output(cout)

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

- As shown above, ostream is alias / another name given to the basic_ostream class.
- cout is object of ostream class. It is external object declared in std namespace.
- It represents monitor which is used to write data on monitor.

- Example 1:

```
#include<cstdio>
#include<iostream>
int main( void ){
  printf("Hello World\n");

  std::cout << "Hello World\n";
  return 0;
}
```

- "<<" operator is called as insertion operator.
- In C language, escape sequence is a character which is used to format the output.
- Example: '\n', '\t', '\r' etc.
- In C++ language, manipulator is a function which is used to format the output.
- Example: endl, setw, fixed,scientific, dec, oct, hex etc.
- Example 2:

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

  //or

  using namespace std;
  cout << "Hello World" <<endl;
  return 0;
}
```

- Example 3:

```
#include<iostream>
int main( void ){
    int num1 = 10;
    int num2 = 20;

    using namespace std;
    cout << num1 << num2 << endl;
    return 0;
}
```

- Example 4:

```
#include<iostream>
int main( void ){
  int num1 = 10;
  int num2 = 20;

  using namespace std;
  cout << num1 << endl;
  cout << num2 << endl;
  return 0;
}
```

- Example 5:

```
#include<iostream>
int main( void ){
  int num1 = 10;
  int num2 = 20;

  using namespace std;
  cout << "Num1 :   " << num1 << endl;
  cout << "Num2 :   " << num2 << endl;
  return 0;
}
```

Character Input(cin)

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

- As shown above, istream is another name given to the basic_istream class.
- cin is object of istream class. It is external object declared in std namespace.
- It represents keyboard which is used to read data from keyboard.
- Example 1

```
#include<cstdio>
#include<iostream>
int main( void ){
  int num1;
  //In C programming language
  printf("Num1  :   ");
  scanf("%d", &num1 );

  //In C++ programming language
  std::cout << "Num1    :   ";
```

```
  std::cin >> num1;
  return 0;
}
```

- ">>" operator is called as extraction operator.
- Example 2

```
#include<iostream>
int main( void ){
  int num1;

  std::cout << "Num1    :   ";
  std::cin >> num1;

  //or
  using namespace std;
  cout << "Num1 :   ";
  cin >> num1;
  return 0;
}
```

- Example 3

```
#include<iostream>
int main( void ){
  int num1, num2;

  using namespace std;
  cin >> num1 >> num2;
  cout << num1 << num2 << endl;
  return 0;
}
```

- Example 4

```
#include<iostream>
int main( void ){
  using namespace std;

  int num1;
  cout << "Num1 :   ";
  cin >> num1;

  int num2;
  cout << "Num2 :   ";
  cin >> num2;
```

```
    cout << "Num1 :    " << num1 << endl;
    cout << "Num2 :    " << num2 << endl;
    return 0;
}
```

Character Error(cerr)

Character Log(clog)

```
#include<iostream>
#include<iomanip>
int main( void ){
    using namespace std;
    int num1;
    cout << "Num1    :    ";
    cin >> num1;
    clog << "Numerator is accepted" <<endl;

    int num2;
    cout << "Num1    :    ";
    cin >> num2;
    clog << "Denominator is accepted" <<endl;

    if( num2 == 0 ){
        cerr << "Value of denominator is 0" <<endl;
        clog << "Can not calculate Result because value of denominator is
0." <<endl;
    }else{
        int result = num1 / num2;
        clog << "Result is calculated" <<endl;
        cout<< "Result  :    "<< result << endl;
        clog << "Result is printed" <<endl;
    }
    return 0;
}
```

Lab Assignment:

- Class : Date
  - Data Member:
    - day: int
    - month: int
    - year: int
  - Member Function
    - void acceptRecord
    - void printRecord
    - void addDays(int count);
    - bool validateDate( );

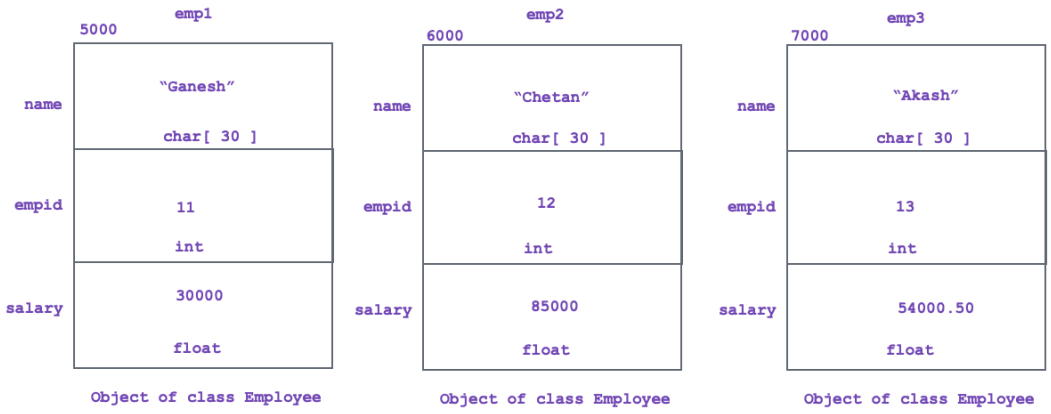
Coding Convention

- Pascal Case Convention
    - Consider examples:
        - Date
        - StringBuffer
        - NullPointerException
        - ArrayIndexOutOfBoundsException
    - In this case, including first word, first character of each word should be in upper case.
    - In C++, we will use this convention for giving name to:
        - Type Names(enum, union, structure, class)
        - File Name
- Camel Case Convention
    - Consider examples:
        - main
        - parseInt;
        - showInputDialog
        - addNumberOfDays
    - In this case, excluding first word, first character of each word should be in upper case.
    - In C++, we will use this convention for giving name to:
        - Data member
        - Member function
        - local variable and function parameter
- Snake Case Convention
    - Consider examples:
        - accept_record
        - print_record
    - In this case, mulitple word names are joined using underscore.
    - In C++, we will use this convention for giving name to:
        - global function
        - constant
        - macro
- Hungerian Notation
    - It is convention recommended for C/C++.
    - Consider examples:
        - int iNum1;
        - double dNum2;
        - char szText[100];

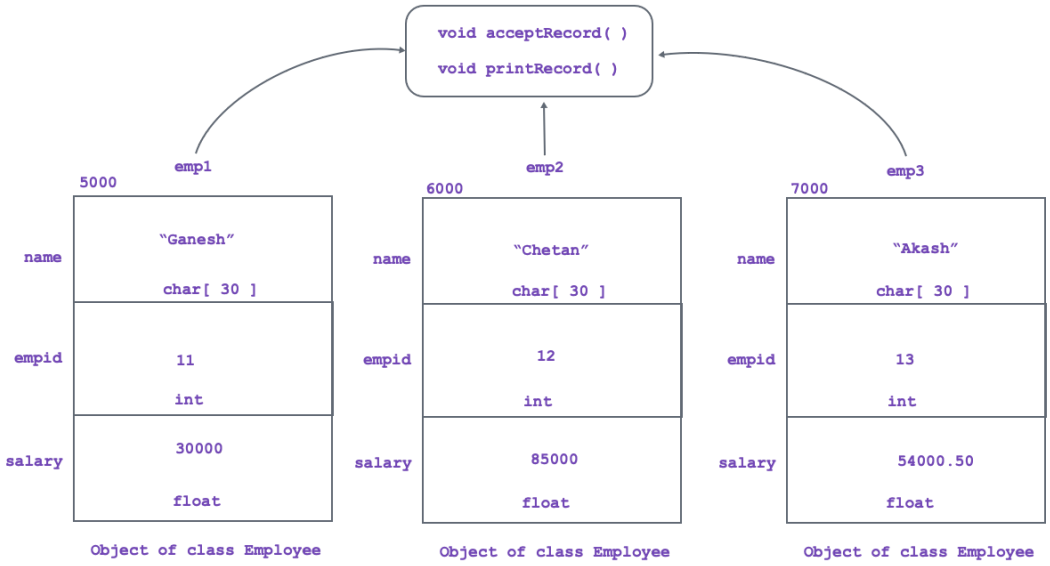
Object oriented concepts

- Only data members get space inside object. Member function do not get space inside object.

- Data members of the class get space once per object according their order of declaration inside class.



- Member function do not get space inside object, rather all the objects of same class share single copy of it.



- Size of object depdends on size of all the data members declared inside class.

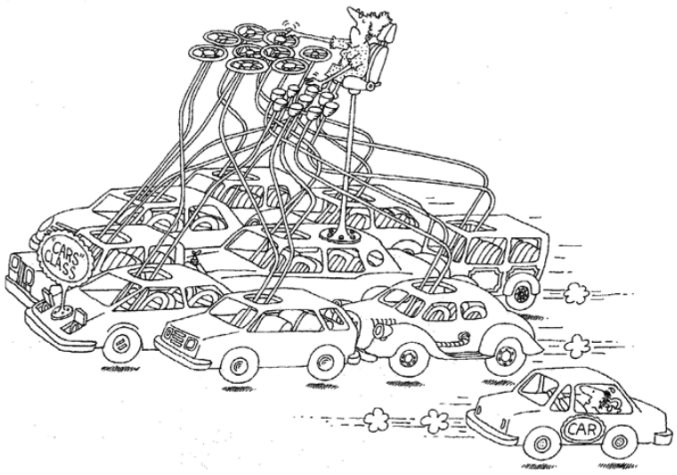
Characteristics of Object

- State:
  - Value stored inside object is called as state of the object.

    - Value of the data member represents state of the object.
- Behavior
    - Set of operations which are allowed to perform on object is called behavior of the object.
    - Member function defined inside class represents behavior of the object.
- Idenitity
    - Value of any data member, which is used to identify object uniquely, is called as identity of the object.
    - When state of objects are same then its address can be considered as its identity.

Class

- Definition:
    - Class is collection of data members and member function.
    - Structure and behaviour of the object depends on class. Hence class is considered as a template / model / blueprint for object.
    - Class represents, group of objects which is having common structure and common behavior.
- Class is an imaginary / logical entity.
- Example: Book, Laptop, Mobile Phone, Car.
- Class implementation represents encapsulation.



Object

- Definition:
    - Object is instance/variable of a class.
    - An entity which is having physical existance is called as object.
    - An entity, which is having state, behavior and identity is called as object.
- Object is real time / physical entity.
- Example: "More Effective C++", "MacBook Air", "iPhone 15", "Skoda Kushaq".

- Instantiation represents abstraction.

Day 5

Empty class

- A class which do not contain any member is called as empty class.
- Consider example:

```
class Test{

};
```

- Size of the object depends on data members declared inside class.
- According to above definition, size of object of empty class should be zero.
- According to oops concept, class is imaginary/logica term/entity and object is real time / physical term/entity. It means that object must get some space inside memory.
- According to Bjarne Stroustrup, size of object of empty class should be non zero.
- Due to compiler optimization, object of empty class get one byte space.

Function Overloading

- In C programming language, we can not give same name to the multiple functions in same project.
- In C++, we can give same name to the multiple functions.

- If implementation of functions are logically same / equivalent then we should give same name to the function.
- If we want to give same name to the function then we must follow some rules:
- Rule 1:
    - If we want to give same name to the function and if type of all the parameters are same then number of parameters passed to the function must be different.

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

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

```
void sum( int num1, int num2 ){
  int result = num1 + num2;
  cout<<"Result :   "<<result<<endl;
}
void sum( int num1, double num2 ){
  double result = num1 + num2 ;
  cout<<"Result :   "<<result<<endl;
}
int main( void ){
  sum( 10, 20 );
  sum( 10, 20.5 );
  return 0;
}
```

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

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

```
void sum( float num1, int num2 ){
  float result = num1 + num2 ;
  cout<<"Result :   "<<result<<endl;
}
int main( void ){
  sum( 10, 20.2f );
  sum( 10.1f, 20 );
  return 0;
}
```

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

```
int sum( int num1, int num2 ){
  int result = num1 + num2;
  return result;
}
void sum( int num1, int num2 ){ //Error: Function definition is
not allowed
  int result = num1 + num2;
}
int main( void ){
  return 0;
}
```

- Definition:
  - When we define multiple functions with the help of above 4 rules then process is called as function overloading.
  - Process of defining functions with same name and different signature is called as function overloading.
  - Functions which take part into overloading are called as overloaded functions.
    - If implementation of functions are logically same / equivalent then we should overload function.
  - In C++ we can overload:
    - global function
    - member function
    - constructor
    - static member function
    - constant member function
    - virtual member function
  - In C++ we can not overload:
    - main function
    - destructor
- Per project, we can define only one main function. Hence we can not overload main function in C++.
- Since destructor do not take any parameter, we can not overload destructor.

Why retrun type is not considered in function overloading:

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

Function overloading twister

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

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

int main( void ){
  //print( 10 );    //int   :   10

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

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

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

  return 0;
}
```

Name mangling and Mangled name

- nm is a tool which is used to print symbol table. We can used it to see mangled name.
- if we define function in C++, then compiler generate unique name for each function by looking toward name of the function and type of parameter passed to the function. Such name is called as mangled name.
- Consider below code:

```
    void sum( int num1, int num2 ){   //__Z3sumii
      int result = num1 + num2;
    }
    void sum( int num1, int num2, int num3 ){ //__Z3sumiii
      int result = num1 + num2 + num3;
    }
    void sum( int num1, float num2){  //__Z3sumif
      float result = num1 + num2;
    }
    void sum( int num1, float num2, double num3 ){    //__Z3sumifd
      double result = num1 + num2 + num3;
    }
    int main( void ){

      return 0;
    }
```

- Process or algorithm which generates mangled name is called as name mangling.
- ISO has not defined any specification on mangled name hence it may vary from compiler to compiler.
- Using extern "C", we can invoke, C language function into C++ source code.
- If we declared any function using exten "C" then compiler do not generate mangled name for it.
- Consider ArithmeticOperation Header file()

```
#ifndef ARITHMETIC_OPERATION_H_
#define ARITHMETIC_OPERATION_H_

typedef enum ArithmeticOperation{
  EXIT, SUM, SUB, MULTIPLICATION, DIVISION
}ArithmeticOperation_t;

#endif
```

- Consider Calculator Header file

```
#ifndef CALCULATOR_H_
#define CALCULATOR_H_

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

  int sub( int num1, int num2 );

  int multiplication( int num1, int num2 );

  int division( int num1, int num2 );
}

#endif
```

- Consider Calculator.c file

```
int sum( int num1, int num2 ){
  return num1 + num2;
}
int sub( int num1, int num2 ){
  return num1 - num2;
}
int multiplication( int num1, int num2 ){
  return num1 * num2;
}
```

```
int division( int num1, int num2 ){
  return num1 / num2;
}
```

- Consider Main.cpp file

```
#include<iostream>
using namespace std;
#include"../include/ArithmeticOperation"
#include"../include/Calculator"


ArithmeticOperation_t menu_list( void ){
  int choice;
  cout << "0.Exit." <<endl;
  cout << "1.Sum." <<endl;
  cout << "2.Sub." <<endl;
  cout << "3.Multiplication." <<endl;
  cout << "4.Division." <<endl;
  cout<<"Enter choice   :   ";
  cin >> choice;
  return ArithmeticOperation_t( choice );
}
int main( void ){
  ArithmeticOperation_t choice;
  while ( ( choice = ::menu_list( ) ) != 0 ){
    int result = 0;
    switch( choice ){
    case SUM:
      result = sum( 100, 20 );
      break;
    case SUB:
      result = sub( 100, 20 );
      break;
    case MULTIPLICATION:
      result = multiplication( 100, 20 );
      break;
    case DIVISION:
      result = division( 100, 20 );
      break;
    }
    cout << "Result :   "<< result <<endl;
  }
  return 0;
}
```

Default Argument

- Consider following code:

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

void sum( int num1, int num2 ){
  int result = num1 + num2;
  cout << "Result :   " << result << endl;
}
void sum( int num1, int num2, int num3 ){
  int result = num1 + num2 + num3;
  cout << "Result :   " << result << endl;
}
void sum( int num1, int num2, int num3, int num4 ){
  int result = num1 + num2 + num3 + num4;
  cout << "Result :   " << result << endl;
}
void sum( int num1, int num2, int num3, int num4, int num5 ){
  int result = num1 + num2 + num3 + num4 + num5;
  cout << "Result :   " << result << endl;
}

int main( void ){
  sum( 10, 20 );

  sum( 10, 20, 30 );

  sum( 10, 20, 30, 40 );

  sum( 10, 20, 30, 40, 50);
  return 0;
}
```

- In C++, we can assign default value to the parameter of function. It is called as default argument.
- Using default argument, we can reduce developers effort.
- Default value can be:
  - constant
  - variable
  - macro
- Example 1:

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

int main( void ){
  sum( 10, 20 );
```

```
  sum( 10, 20, 30 );

  sum( 10, 20, 30, 40 );

  sum( 10, 20, 30, 40, 50);
  return 0;
}
```

- Example 2

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

int main( void ){
  sum( 10, 20 );

  sum( 10, 20, 30 );

  sum( 10, 20, 30, 40 );

  sum( 10, 20, 30, 40, 50);
  return 0;
}
```

- Example 3:

```
  #define DEFAULT_VALUE 0
  void sum( int num1, int num2, int num3 = DEFAULT_VALUE, int num4 =
DEFAULT_VALUE, int num5 = DEFAULT_VALUE ){
    int result = num1 + num2 + num3 + num4 + num5;
    cout << "Result :   " << result << endl;
  }

  int main( void ){
    sum( 10, 20 );

    sum( 10, 20, 30 );

    sum( 10, 20, 30, 40 );

    sum( 10, 20, 30, 40, 50);
    return 0;
  }
```

- Default arguments are always given from right to left direction.
- We can assign, default argument to the parameters of member function as well as global function.
- When we separate , function declaration and definition then default argument must appear in declaration part:

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

  #define DEFAULT_VALUE 0

  void sum( int num1, int num2, int num3 = DEFAULT_VALUE, int num4 =
DEFAULT_VALUE, int num5 = DEFAULT_VALUE );

  int main( int argc, char *argv[ ] ){
    sum( 10, 20 );

    sum( 10, 20, 30 );

    sum( 10, 20, 30, 40 );

    sum( 10, 20, 30, 40, 50);
    return 0;
  }

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

this pointer

- Software development life cycle:
  - Requirement
  - Analysis
  - Design
  - Implementation / Coding
  - Testing
  - Deployment / Installation
  - Maintenance
- Problem Statment: Write a program to test functionality(accept and print record) of complex number.
  - Analysis
    - class Complex:
      - real number : int
      - imag number : int

- Understand problem statement and do analysis from object oriented point of view. In other words, decide class and data members for it.
- Create object of the class
    - Inside object only data member will get space.
- To process state of the object we should call and define member function.
- If we call member function on object then compiler implicitly pass, address of current/calling object as a argument to the member function. To catch/accpet address, compiler implicitly declare/create one paramater inside member function. Such parameter is called as this pointer.
    - this is a keyword in C++.
    - Parameter do not get space inside object. Since this pointer is a function parameter, it doesnt get space inside object.
    - this pointer get space once per function call on stack section / segment.
    - this pointer is a constant pointer. General type of this pointer is:

```
ClassName *const this;
```

    - To access members of the class, use of this keyword is optional. If we do not use this then compiler implicitly use this keyword.
    - Using this pointer, data member and member function can communicate with each other. Hence this pointer is considered as a link / connection between them.
    - Following functions do not get this pointer:
        - Global function
        - Static member function
        - Friend function
    - this pointer is considered as first parameter of the member function.

```
class Complex{
private:
  int real;
  int imag;
public:
  void acceptRecord( /* Complex *const this, */ int n1, int n1 ){
    cout << "Enter real number  :   ";
    cin >> this->real;
    cout << "Enter imag number  :   ";
    cin >> this->imag;
  }
};
int main( void ){
  Complex c1;
  c1.acceptRecord( 10, 20 );    //c1.acceptRecord( &c1, 10, 20 );
  return 0;
}
```

- Definition:
  - this pointer is implicit pointer, which is available in every non static member function of the class and which is used to store address of current / calling object.
- If name of data member and local variable / function parameter is same then preference will be given to local variable. In this case we should use this pointer before data members.

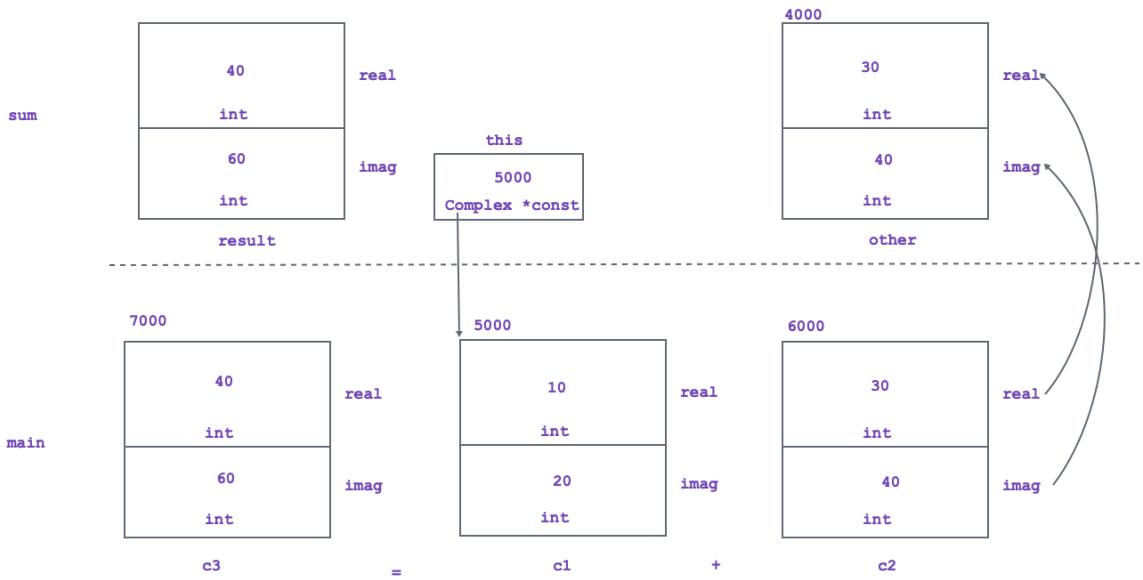
```
class Complex{
private:
  int real;
  int imag;
public:
  //Complex *const this = &c1
  void setReal( int real ){
    this->real = real;
  }
};
int main( void ){
  Complex c1;
  c1.setReal( 10 ); //c1.setReal( &c1, 10 );
  return 0;
}
```

Getter and Setter methods:

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

class Complex{
private:
  int real;
  int imag;
public:
  //Complex *const this = &c1
  int getReal( void ){
    return this->real;
  }
  //Complex *const this = &c1
  void setReal( int real ){
    this->real = real;
  }
  //Complex *const this = &c1
  int getImag( void ){
    return this->imag;
  }
  //Complex *const this = &c1
  void setImag( int imag ){
    this->imag = imag;
  }
```

```
};
int main( void ){
  Complex c1;

  c1.setReal( 10 );
  c1.setImag( 20 );

  cout <<"Real Number   :   " << c1.getReal( ) << endl;
  cout <<"Imag Number   :   " << c1.getImag( ) << endl;

  return 0;
}
int main1( void ){
  Complex c1;
  c1.setReal( 10 );
  c1.setImag( 20 );

  int real = c1.getReal( ); //c1.getReal( &c1 );
  cout <<"Real Number   :   " << real << endl;

  int imag = c1.getImag( );
  cout <<"Imag Number   :   " << imag << endl;
  return 0;
}
```

- A member function of class, which is used to read state of the object is called as inspector / selector / getter function.
- A member function of class, which is used to modify state of the object is called as mutator / modifier / setter function.



Day 6

Constructor

- Member function of a class which is used to initialize the object is called as constructor.
- Note: Constructor do not create object rather it initializes object.
- Due to below reasons constructor is considered as special function of the class:
  - Its name is always same as class name.
  - It does not have any return type
  - It is designed to call implicitly
  - It gets called once per instance.
- We can not call constructor on object, pointer or reference explicitly.
- Example 1:

```
Complex c1;
c1.Complex( );  //Not OK
```

- Example 2:

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

- Example 3:

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

- We can use any access specifier on constrctor:
  - If constructor is public then we can create object inside member function of the class as non member function of the class.
  - If constructor is private then we can create object inside member function of the class only.
- We can not declare constructor static, constant, volatile or virtual but we can declare constructor inline.
- Types of constructor:
  - Parameterless constructor
  - Parameterized constructor
  - Default constructor.

Parameterless constructor:

- It is also called as zero argument constructor or user defined default constructor.
- Constructor of the class which do not take any parameter is called as parameterless constructor.
- Example:

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

- If we create object without passing argument, then compile invoke parameterless constrctor.
- Example:

```
Complex c1; //Here on c1 parameterless constructor will call.
```

Parameterized constructor

- Constructor of the class which is having parameter(s) is called as parameterized constructor.
- Example:

```
Complex( int value ){ //Single parameter constructor
  this->real = value;
  this->imag = value;
}
Complex( int real, int imag ){  // 2 parameter constructor
  this->real = real;
  this->imag = imag;
}
```

- If we create object by passing arguments then parameterized constructor gets called.
- Example:

```
Comple c1( 10, 20 );
Complex c2( 30 );
```

- We can overload constructor. Consider below code:

```
class Complex{
private:
  int real;
  int imag;
public:
  Complex( ){ //Parameterless constructor
    this->real = 0;
    this->imag = 0;
  }
  Complex( int real, int imag ){  //Parameterized constructor
    this->imag = real;
```

```
    this->imag = imag;
  }
};
```

- Constructor calling sequence depends on order of object declaration:
- Example:

```
Complex c1(10,20), c2;
//First, parameterized constructor on c1 will call
//Then parameterless constructor on c2 will call
```

Default constructor

- If we do not define constructor inside class then compiler generate constructor for the class. Such constructor is called as default constructor.
- Compiler never generate parameterized constructor. In other words, compiler generated constructor is zero argument / parameterless constructor.
- Example:

```
class Complex{

};
int main( void ){
  Comple c1;  //On c1 Default constructor will call

  Complex c2( 10, 20 ); //Compiler error
  return 0;
}
```

Aggregate Type and Aggregate initialization

- In C, below types are aggregate types whose object can be initialize using initializer list.
  - Array
  - Structure
  - Union
- Example:

```
  int arr[ 3 ] = { 10, 20, 30 };
  struct Account account = { 3052707, "Saving", 85000.50f };
```

- Plain Old Data (POD) structure is also called as aggregate class in C++.
- Aggregate class class following properties:

    - It does not contain private or protected non static data member.
    - It does not contain any user defined constructor.
    - It does not have base class
    - It does not contain virtual function
- Aggregate initialization:

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

Miscellaneous

- Consider below code:

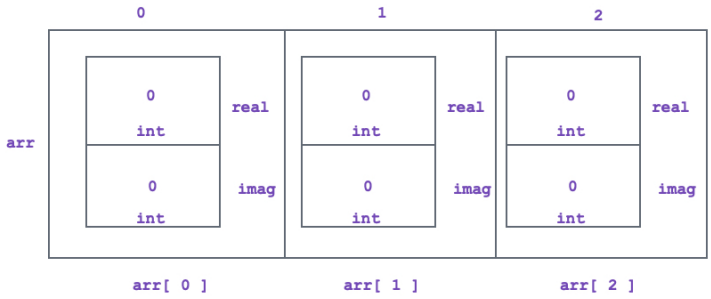
```
class Complex{
private:
  int real;
  int imag;
public:
  Complex( void ){
    this->real = 0;
    this->imag = 0;
  }
  Complex( int value ){
    this->real = value;
    this->imag = value;
  }
  Complex( int real, int imag ){
    this->real = real;
    this->imag = imag;
  }
  void printRecord(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

- Complex c1;
  - Here on c1 object, parameterless constructor will call.
- Complex c2(10);
  - Here on c2 object, single parameter constructor will call.
- Complex c3(10, 20);
  - Here on c2 object, 2 parameter constructor will call.
- Complex c4( );
  - It is declaration of c4 function which do not take any parameter and return object or Complex type.
  - Constructor will not call here.
- Complex c5 = 30;
  - It is same as Complex c5(30).
  - Hence on c5, single parameter constructor will call.
- Complex(40, 50);
  - It is anonymous object.
  - On object, 2 parameter constructor will call.
- Complex c6 = 60, 70;
  - Compiler error.
- Complex c7{ 80, 90 };
  - class Complex is not agrregate type. Hence it is compiler error.

Array Of Objects

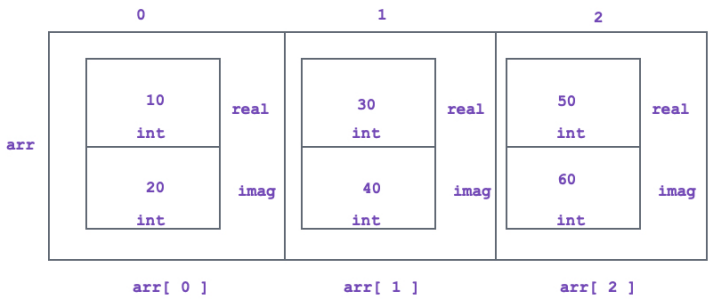
- Example 1:

```
Complex arr[ 3 ];
for( int index = 0; index < 3; ++ index )
  arr[ index ].printRecord( );
```



- Example 2:

```
Complex arr[ 3 ] = { Complex(10,20), Complex(30,40), Complex(50,60)};
for( int index = 0; index < 3; ++ index )
  arr[ index ].printRecord( );
```



Constructor member initializer list

- If we want to initialize data members according to order of data member declaration then we should use constructor member initializer list.
- Example 1:

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

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

  }
  Test( int num1, int num2, int num3 ) : num1( num1 ), num2( num2 ),
num3( num3 ) {

  }
  void printRecord( void ){
    cout << "Num1 :   " << this->num1 << endl;
    cout << "Num2 :   " << this->num2 << endl;
    cout << "Num3 :   " << this->num3 << endl;
  }
};
int main( void ){
  Test t1;
  t1.printRecord( );

  Test t2( 50, 60, 70 );
  t2.printRecord( );
  return 0;
}
```

- If we want to separate declaration and definition then constructor member initializer list must appear in definition part.
- Example 2:

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

class Test{
private:
  int num1;
  int num2;
  int num3;
public:
```

```
    Test( void );

    Test( int num1, int num2, int num3 ) ;

    void printRecord( void );

  };
  Test::Test( void ) : num1( 10 ), num2( 20 ), num3( 30 ) {

  }
  Test::Test( int num1, int num2, int num3 ) : num1( num1 ), num2(
num2 ), num3( num3 ) {

  }
  void Test::printRecord( void ){
    cout << "Num1   :   " << this->num1 << endl;
    cout << "Num2   :   " << this->num2 << endl;
    cout << "Num3   :   " << this->num3 << endl;
  }
  int main( void ){
    Test t1;
    t1.printRecord( );

    Test t2( 50, 60, 70 );
    t2.printRecord( );
    return 0;
  }
```

- Example 3:

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

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

    void printRecord( void );

  };
  Test::Test( int num1, int num2, int num3 ) : num1( num1 ), num2(
num2 ), num3( num3 ) {
  }
  void Test::printRecord( void ){
    cout << "Num1   :   " << this->num1 << endl;
    cout << "Num2   :   " << this->num2 << endl;
    cout << "Num3   :   " << this->num3 << endl;
  }
```

```
int main( void ){
  Test t1;
  t1.printRecord( );

  Test t2( 50, 60, 70 );
  t2.printRecord( );
  return 0;
}
```

- Example 4:

```
#include<cstring>
#include<iostream>
#include<iomanip>
using namespace std;

class Employee{
private:
  char name[ 30 ];
  int empid;
  float salary;
public:
  Employee( const char *name = "", int empid = 0, float salary  =
0.0f );

  void printRecord( void );
};
Employee::Employee( const char *name, int empid, float salary ) :
empid( empid ), salary( salary ){
  strcpy( this->name, name);
}
void Employee::printRecord( void ){
  cout << "Name   :   " << this->name << endl;
  cout << "Empid  :   " << this->empid << endl;
  cout << "Salary :   " << fixed << setprecision( 2 ) << this-
>salary<< endl;
}
int main( void ){
  Employee emp1;
  emp1.printRecord( );

  Employee emp2("Sandeep", 3778, 45000.50f);
  emp2.printRecord( );

  return 0;
}
```

Constant Variable in C++

- const and volatile are type qualifiers in C/C++.

- Once initialized, if we dont want to modify state/value of the variable then we should use const qualifier.
- In C++, we can not modify value of the variable using pointer. Hence initialization of constant variable is mandatory.
- Consider Example:

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

Constant Data Member

- Consider below code:

```
#include<iostream>
using namespace std;
class Test{
private:
  int number;
public:
  Test( void ){
    this->number = 0;
    this->number = this->number + 10; //OK
  }
  void showRecord( void ){
    this->number = this->number + 2;  //OK
    cout << "Number   :   " << this->number << endl;
  }
  void printRecord( void ){
    this->number = this->number + 3;  //OK
    cout << "Number   :   " << this->number << endl;
  }
};
int main( void ){
  Test t;
  t.showRecord( );    //12
  t.printRecord( );   //15
  t.printRecord( );   //18
  t.showRecord( );    //20
  return 0;
}
```

- Once initialized, if we dont want to modify value of the data member inside any member function of the class including constructor body then we should declare data member constant.
- We can initialize non constant data member using constructor member initializer list or constructor body but we must initialize constant data member using constructor member initializer list.

```
#include<iostream>
using namespace std;
class Test{
private:
  const int number;
public:
  Test( void ) : number( 10 ){
    //this->number = this->number + 10;   //Not OK
  }
  void showRecord( void ){
    //this->number = this->number + 2;    //Not OK
    cout << "Number   :   " << this->number << endl;
  }
  void printRecord( void ){
    //this->number = this->number + 3;    //Not OK
    cout << "Number   :   " << this->number << endl;
  }
};
int main( void ){
  Test t;
  t.showRecord( );    //10
  t.printRecord( );   //10
  t.printRecord( );   //10
  t.showRecord( );    //10
  return 0;
}
```

Constant Member Function

- Consider below statement:

```
ClassName *const this;
```

  - In above statement, this pointer is constant pointer which can store address of any non constant object.
  - It means that this pointer can contain address of only one object but using this pointer we can modify state of the object.
- Consider below statement:

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

  - In above statement, this pointer is constant pointer which can store address of any onstant object.
  - It means that this pointer can contain address of only one object and using this pointer we can not modify state of the object.

- If we want to modify state of the non constant object inside member function then type of this pointer should be "ClassName *const this" but If we dont want to modify state of the non constant object inside member function then type of this pointer should be "const ClassName *const this".
- If we dont want to modify state of the only current/calling object inside member function then we should declare member function constant.

```
#include<iostream>
using namespace std;
class Test{
private:
  int number;
public:
  //Test *const this
  Test( void ) : number( 10 ){
    this->number = this->number + 10; //OK
  }
  //Test *const this
  void showRecord( void ){
    this->number = this->number + 2;  //OK
    cout << "Number   :   " << this->number << endl;
  }
  //const Test *const this
  void printRecord( void )const{
    //this->number = this->number + 3;    //Not OK
    cout << "Number   :   " << this->number << endl;
  }
};
int main( void ){
  Test t;
  t.showRecord( );    //12
  t.printRecord( );   //12
  t.printRecord( );   //12
  t.showRecord( );    //14
  return 0;
}
```

- Note: Only state of current object will not be changed inside constant member function. Other object can be modifed inside constant member function.

```
#include<iostream>
using namespace std;
class Test{
private:
  int number;
public:
  //Test *const this
  Test( void ) : number( 10 ){
    this->number = this->number + 10; //OK
  }
  //Test *const this
```

```
  void showRecord( void ){
    this->number = this->number + 2;  //OK
    cout << "Number   :   " << this->number << endl;
  }
  //const Test *const this
  void printRecord( void )const{
    Test t;
    t.number = 20;    //OK
    t.showRecord( );  //It will print 22
  }
};
int main( void ){
  Test t;
  t.printRecord( );
  return 0;
}
```

- On non constant object, we can call constant member function as well as non constant member function.
- Below functions are not allowed to declare as constant:
  - Global function
  - Static Member Function
  - Constructor
  - Destructor
- Since main function is global function, we can make it constant.

Why we can not declare global function constant?

- According to concept, if we dont want to modify state of the current object inside member function then we should declare member function constant.
- In other words, constant member function is designed to call on object.
- Since global function is not designed to call on object, we can not make it constant.

mutable keyword

- Exceptionlly, if we want to modify state of non constant data member inside constant member function then we should declare that data member mutable.

- Consider below code:

```
#include<iostream>
using namespace std;
class Test{
private:
  int num1;
  int num2;
  mutable int num3;
public:
  Test( void ) : num1( 10 ), num2( 20 ), num3( 0 ){
  }
```

```
    void printRecord( void )const{
      //this->num1 ++;  //Not OK
      cout<< "Num1  :   " << this->num1 << endl;
      //this->num2 ++;  //Not OK
      cout<< "Num2  :   " << this->num2 << endl;
      this->num3 ++;    //OK
      cout<< "Num3  :   " << this->num3 << endl;
    }
};
int main( void ){
    Test t1;
    t1.printRecord( );
    return 0;
}
```

Constant Object

- If we want some objects to be constant and some objects to be non constant then we should use constant keyword.
- Example:

```
Test t1, t2;    //non constant objects
const Test t2;  //constant object
```

- 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.

```
#include<iostream>
using namespace std;
class Test{
private:
    int number;
public:
    //Test *const this
    Test( ) : number( 0 ){
    }

    //Test *const this
    void printRecord( void ){
      cout << "printRecord" <<endl;
    }
    //const Test *const this
    void printRecord( void )const{
      cout << "const printRecord" <<endl;
    }
};
int main( void ){
```

```
  Test t1;
  t1.printRecord( );  //printRecord

  const Test t2;
  t2.printRecord( );  //const printRecord
  return 0;
}
```

Day 7

Typedef

- typedef if a keyword in C/C++.
- Using typedef we can not define new Type / new user defined data type.
- If we want to give short and meaningful name then we should use typedef.
- Using typedef we can create alias for Type / class not for object.
- Example 1:

```
typedef unsigned short  wchar_t;
```

- Example 2:

```
typedef enum ArithmeticOperation{
  //TODO
}ArithmeticOperation_t;
```

- Example 3:

```
typedef struct Employee{
  //TODO
}Employee_t;
```

- Example 4:

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

Reference

- Consider below example:

```
```

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

- Consider below example:

```
int num1 = 10;    //Initialization
int *num2 = &num1;  //Initialization
```

- Consider below example:

```
int num1 = 10;    //Initialization
int &num2 = num1;  //Initialization
//Here num2 is reference variable and num1 is referent variable.
```

- Reference is an alias or another name given to the existing object.
- Using typedef we can create alias for class and using reference we can create alias for object.
- Example 1:

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

  ++ num1;  //11
  ++ num2;  //12
  cout<<"Num1   :   "<< num1<<endl; //12
  cout<<"Num2   :   "<< num2<<endl; //12
  return 0;
}
```

- We can create multiple references to the same memory location. Consider below code:
- Example 2

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

    ++ num1;    //11
    ++ num2;    //12
    ++ num3;    //13

    cout<<"Num1 :   "<< num1<<endl; //13
```

```
    cout<<"Num2 :   "<< num2<<endl; //13
    cout<<"Num3 :   "<< num3<<endl; //13
    return 0;
  }
```

- Example 3

```
  int main( void ){
    int num1 = 10;
    int &num2 = num1;       //using num2, we can read/modify value of
num1
    const int &num3 = num1; //using num3, we can read value but can
not modify value of num1

    ++ num2;    //OK: 11
    //++ num3;  //Not OK

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

- We can not chnage referent of reference variable.

- Example 4:

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


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

- We can create pointer to pointer but we can not create reference to reference.

- Example 5:

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

```
    int &num2 = num1;
    int &num3 = num2;

    ++ num1;    //11
    ++ num2;    //12
    ++ num3;    //13

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

- Proecss of accessing value of the variable using pointer is called as dereferencing.
- NULL is macro whose value is 0 address.

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

- Reference is automatically dereferenced constant pointer variable.

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

  int &num2 = num1;
  //int *const num2 = &num1;

  cout << num2 <<endl;
  //cout << *num2 <<endl;
  return 0;
}
```

- How to check size of reference:

```
  class Test{
  private:
    char &ch;
  public:
    Test( char &ch2 ) : ch( ch2 ){

    }
  };
  int main( void ){
    char ch1 = 'A';
    Test t( ch1 );
    size_t size = sizeof( t );
    cout << "Size   :   " << size << endl;
```

```
    return 0;
}
```

What is the difference between pointer and reference

- Initialization:
  - Pointer initialization is not mandatory but reference initialization is mandatory.
- NULL:
  - We can initialize pointer to NULL but we can not initialize reference to NULL.
- Pointer to pointer & reference to reference:
  - We can create pointer to pointer but we can not create reference to reference.
- Array
  - We can create array of pointers but we can not create array of reference.
- Dereferencing:
  - To access the value of variable pointer need dereferencing but reference need not to do dereferencing.

In C++, we can pass argument to the function by value, by address as well as by reference.

Passing argument by value

- Example 1:

```
  void swap_number( int x, int y ){
    int temp = x;
    x = y;
    y = temp;
  }
  int main( void ){
    int a = 10;
    int b = 20;

    swap_number( a, b );    //a , b are arguments; we are passing
passing it by value to the function

    cout << "a  :   " << a << endl;
    cout << "b  :   " << b << endl;
    return 0;
  }
```

Passing argument by address

- Example 2:

```
  //int *const x = &a;
//int *const y = &b;
void swap_number( int *const x, int *const y ){
```

```
  int temp = *x;
  *x = *y;
  *y = temp;
}
int main( void ){
  int a = 10;
  int b = 20;

  swap_number( &a, &b );    //adress of a , b are arguments; we are
passing passing it by address  to the function

  cout << "a    :   " << a << endl; //20
  cout << "b    :   " << b << endl; //10
  return 0;
}
```

Passing argument by reference

- Example 3:

```
//int &x = a;   //int *const x = &a;
//int &y = b;   //int *const y = &b;
void swap_number( int &x, int &y ){
  int temp = x; //int temp = *x;
  x = y;            //*x = *y;
  y = temp;     //*y = temp;
}
int main( void ){
  int a = 10;
  int b = 20;

  swap_number( a, b );  //Function call by reference

  cout << "a    :   " << a << endl; //20
  cout << "b    :   " << b << endl; //10
  return 0;
}
```

- Consider below example:

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

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

```
int main( void ){
  print( 10 );  //int   :   10

  int value = 10;
  //print( value ); //error: call to 'print' is ambiguous
  return 0;
}
```

- We can not create array of references but we can create reference to array.
- Example:

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

  int main( void ){
    //int& arr[ 3 ];  //Not OK: Array of references
    int arr1[ 3 ] = { 10, 20, 30 };
    int (&arr2)[ 3 ] = arr1;    //arr2 is reference and arr1 is
referent
    for( int index = 0; index < 3; ++ index )
      cout<<arr2[ index ]<<endl;
    return 0;
  }
```

Exception Handling

- If we make some syntatical mistake in the code then compiler generates error.
- Example:

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

- Without definition, if we try to access any member then linker generates error.
- Example 1:

```
int main( void ){
  extern int number;  //OK: Declaration
  cout << "Number : " << number << endl;  //Linker Error
  return 0;
}
```

- Example 2:

```
void print( ); //Declration
int main( void ){
  print( ); //Linker error
  return 0;
}
```

- Logical error is called bug. In other words, syntacticaly valid but logically invalid statment represents bug.
- Example 1:

```
int main( void ){
  int status = 0;
  if( status == 0 )
    cout<<"If"<<endl;
  else;
    cout<<"Else"<<endl;
}
//Output:
//If
//Else
```

- Example 2:

```
int main( void ){
  int status = 10;
  if( status = 0 )  //Bug
    cout<<"If"<<endl;
  else
    cout<<"Else"<<endl;

  cout<<"Status : "<<status<<endl;  //0
}
//Output: Else
//Status : 0
```

- Example 3:

```
int main( void ){
  int count;
  for( count = 1; count <= 10; ++ count );
  cout << "Count  : "<< count <<endl  //11
}
```

- Runtime error is called as 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 system.
- Below are the operating system resources that we use for the application development
    - Memory
    - File
    - Thread
    - API
    - Socket
    - I/O devices
    - Processor
- Since operating system resources are limited, we should handle it carefully. In other words, we should avoid their leakage.
- If we want handle/manage OS resources carefully then we should use exception handling mechanism in the code.
- In C++, we can handle exception using 3 keywords:
    - try
    - catch
    - throw
- try block
    - try is keyword in C++.
    - try block is also called as try handler.
    - If we want to keep watch of group of statements then we should use try block / try handler.
    - Example:

```
int num1;
accept_record( num1 );

int num2;
accept_record( num2 );

try{
  int result = num1 / num2;
  print_record( result );
}
```

    - We can not define try block after catch block.
- throw
    - throw is a keyword in C++.

- If we want to generate new exception then we should use throw keyword.

- catch
  - catch is a keyword in C++.
  - catch block is also called as catch handler.
  - To handle execption, we should use catch block/catch handler.
  - Single try block may have multiple catch blocks but it must have at least one catch block.
  - A catch block, which can handle all types of exception is called as generic catch block.
  - We must define generic catch block after all specific catch block.
  - For thrown exception, if we do not define matching catch block then C++ runtime, implicitly give call to the std::terminate() function which implicitly give call to the std::about function.

- Example:

```
  #include<iostream>
  #include<string>
  using namespace std;
  class ArithmeticException{
  private:
    string message;
    int lineNumber;
    string functionName;
    string fileName;
  public:
    ArithmeticException( string message, int lineNumber, string
functionName, string fileName )
      : message( message ), lineNumber( lineNumber ), functionName(
functionName ), fileName( fileName ){
    }
    void printStackTrace( ){
      cout << this->message <<" in " << this->fileName <<":"<< this-
>functionName <<" at line no. "<< this->lineNumber<<endl;
    }
  };

  void accept_record( int &number ){
    cout << "Number :   ";
    cin >> number;
  }
  int calculate( int num1, int num2 ){
    if( num2 == 0 )
      throw ArithmeticException("Divide by zero exception", __LINE__,
__FUNCTION__, __FILE__ );
    return num1 / num2;
  }
  void print_record( int &result ){
    cout << "Result :   " << result << endl;
  }
  int main( void ){
    try{
      int num1;
```

```
        accept_record( num1 );

        int num2;
        accept_record( num2 );

        int result = calculate( num1, num2 );
        print_record( result );

      }catch( ArithmeticException &ex ){
        ex.printStackTrace();
      }
      return 0;
    }
```

- Exception Specification List:

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

- List of type(s) of exception that we specify after function name using throw keyword is called as exception specification list.
- Its responsibility of C++ developer to specify exception specification list.
- If type of thrown exception is not available in exception specification list then C++ runtime implicitly give call to the std::unexpected function which internally give call to the std::terminate() function.
- If we separate function declaration and definition then we should specify exception specification list in declaration as well as definition.
- Using throw keyword, we can rethrow exception from netsed catch block into outer catch block:

```
  #include<iostream>
  #include<string>
  using namespace std;
  int main( void ){
    try{
      try{
        string ex("exception" );
        throw ex;
      }catch( string &ex ){
        cout << "Inside nested catch block" <<endl;
        throw;  //Rethrow exception ex
      }
```

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

- Outer catch block can handle exception thrown from inner try block. But inner catch block can not handle exception thrown from outer try block.

Day 8

Dynamic Memory Management in C

- Below functions are declared in stdlib header file.
  - void* malloc(size_t size);
  - void* calloc(size_t count, size_t size);
  - void* realloc(void *ptr, size_t size);
  - void free(void *ptr);
- malloc, calloc and realloc are used to allocate memort whereas free function is used to deallocate memory.

malloc function

- malloc is a function declared in header file.
- prototype:

```
void* malloc(size_t size);
```

- It is designed to allocate memory for single variable / single object. But we can use it to allocate memory for array too.
- Using malloc function, we can allocate memory on only heap section.
  - Everything on heap section is anonymous. In other words, dynamic object created using malloc is anonymous.
- If we allocate memory using malloc function then memory gets initialized with garbage value.
- If malloc function fails to allocate memory then it returns NULL.
- malloc(0):
  - Some implementations of malloc will return a null pointer when we request to allocate zero bytes. This is a way to handle the situation gracefully and indicate that no memory has been allocated.
  - In other implementations, malloc(0) may return a valid, non-null pointer that you can use to manipulate memory. However, this can lead to unexpected behavior and should generally be avoided because it doesn't allocate any usable memory.
- memory allocated using malloc function should be deallocate using free() function.
- Example 1:

```
void *ptr = malloc( 4 );
//or
void *ptr = malloc( sizeof( int ) );
```

- If dereferencing is required then we must take specific pointer.
- Example 2:

```
void *ptr1 = malloc( sizeof( int ) );
//*ptr1 = 10; //Not OK
int *ptr2 = ( int* )ptr1; //Type casting is required.
*ptr2 = 10; //OK
```

- Example 3:

```
int *ptr = ( int* )malloc( sizeof( int ) );
*ptr = 10;  //OK
free( ptr );
```

calloc function

- calloc is a function declared in header file.
- prototype:

```
void* calloc(size_t count, size_t size);
```

- It is designed to allocate memory for array. But we can use it to allocate memory for variable / single object too.
- Using calloc function, we can allocate memory on only heap section.
- If we allocate memory using calloc function then memory gets initialized with zero(0) value.
- If calloc function fails to allocate memory then it returns NULL.
- memory allocated using calloc function should be deallocate using free() function.

realloc function

- realloc is a function declared in header file.
- prototype:

```
void* realloc(void *ptr, size_t size);
```

- The realloc() function tries to change the size of the allocation pointed to by ptr to size, and returns ptr.

- If there is not enough room to enlarge the memory allocation pointed to by ptr, realloc() creates a new allocation, copies as much of the old data pointed to by ptr as will fit to the new allocation, frees the old allocation, and returns a pointer to the allocated memory.
- If ptr is NULL, realloc() is identical to a call to malloc() for size bytes.
- If size is zero and ptr is not NULL, a new, minimum sized object is allocated and the original object is freed.
- memory allocated using realloc function should be deallocate using free() function.

free function

- free is a function declared in header file.
- prototype:

```
void free(void *ptr);
```

- The free() function deallocates the memory allocation pointed to by ptr.
- If ptr is a NULL pointer, no operation is performed.

Memory allocation and deallocation for single variable using malloc/free function.

```
#include<iostream>
#include<cstdlib>
using namespace std;

int main( void ){
  //Memory allocation for single integer variable
  int *ptr = (int*)malloc( sizeof( int ) );

  //Dereferencing
  *ptr = 123;
  cout<<"Value  :   "<< *ptr << endl;   //Dereferencing

  //Memory deallocation for single integer variable
  free( ptr );
  return 0;
}
```

Memory allocation and deallocation for single variable using calloc/free function.

```
#include<iostream>
#include<cstdlib>
using namespace std;

int main( void ){
  //Memory allocation for single integer variable
  int *ptr = (int*)calloc( 1, sizeof( int ) );
```

```
  //Dereferencing
  *ptr = 123;
  cout<<"Value  :   "<< *ptr << endl;   //Dereferencing

  //Memory deallocation for single integer variable
  free( ptr );
  return 0;
}
```

Memory allocation and deallocation for array using malloc/free function.

```
#include<iostream>
#include<cstdlib>
using namespace std;

int main( void ){
  //Memory allocation for integer array
  int *ptr = (int*)malloc( 3 * sizeof( int ) );

  //Dereferencing
  ptr[ 0 ] = 10;    //*( ptr + 0 )  = 10
  ptr[ 1 ] = 20;    //*( ptr + 1 )  = 20
  ptr[ 2 ] = 30;    //*( ptr + 2 )  = 30

  //Dereferencing
  for( int index = 0; index < 3; ++ index )
    cout << ptr [ index ] <<endl;   //cout << *( ptr + index ) <<endl;

  //Memory deallocation array
  free( ptr );
  return 0;
}
```

Memory allocation and deallocation for array using calloc/free function.

```
#include<iostream>
#include<cstdlib>
using namespace std;

int main( void ){
  //Memory allocation for integer array
  int *ptr = (int*)calloc( 3 , sizeof( int ) );

  //Dereferencing
  ptr[ 0 ] = 10;    //*( ptr + 0 )  = 10
  ptr[ 1 ] = 20;    //*( ptr + 1 )  = 20
  ptr[ 2 ] = 30;    //*( ptr + 2 )  = 30
```

```
  //Dereferencing
  for( int index = 0; index < 3; ++ index )
    cout << ptr [ index ] <<endl;   //cout << *( ptr + index ) <<endl;

  //Memory deallocation array
  free( ptr );
  return 0;
}
```

Memory allocation and deallocation for multidimensional array using malloc/free function.

```
#include<iostream>
#include<cstdlib>
using namespace std;

int main( void ){

  int **ptr = (int**)malloc( 2 * sizeof( int ) );
  for( int index = 0; index < 2; ++ index )
    ptr[ index ] = ( int* ) malloc( 3 * sizeof( int* )  );

  //TODO: accept and print record

  for( int index = 0; index < 2; ++ index )
    free( ptr[ index ] );
  free( ptr );
  return 0;
}
```

Memory allocation and deallocation for multidimensional array using calloc/free function.

```
#include<iostream>
#include<cstdlib>
using namespace std;

int main( void ){

  int **ptr = (int**)calloc( 2 * sizeof( int ) );
  for( int index = 0; index < 2; ++ index )
    ptr[ index ] = ( int* ) calloc( 3 * sizeof( int* )  );

  //TODO: accept and print record

  for( int index = 0; index < 2; ++ index )
    free( ptr[ index ] );
  free( ptr );
```

```
    return 0;
}
```

Dynamic Memory Management in C++

- In C++, new is operator which is used to allocate memory and delete is a operator which is used to deallocate memory.
- Consider memory allocation and deallocation for single integer variable.
- Example 1:

```
int *ptr = new int; //Here memory will be initialized to garbage value
//int *ptr = ( int* )::operator new ( sizeof( int ) );

cout << "Value  : " << *ptr << endl;

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

- Example 2:

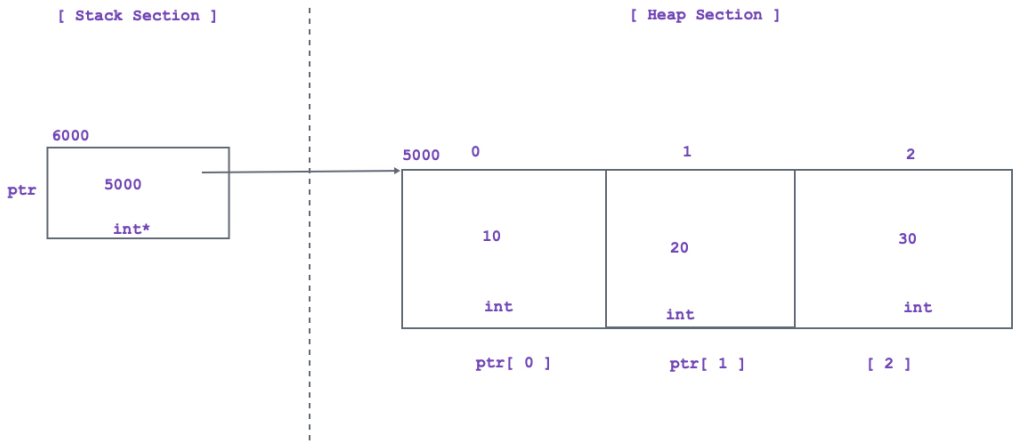
```
int *ptr = new int( 123 ); //Here memory will be initialized to 123
value
//int *ptr = ( int* )::operator new ( sizeof( int ) );

cout << "Value  : " << *ptr << endl;

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

- Example 3:

```
  int *ptr = new int; //Here memory will be initialized to garbage
value
    //int *ptr = ( int* )::operator new ( sizeof( int ) );

    *ptr = 123; //dereferencing
    cout << "Value  : " << *ptr << endl;

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

- Example 4:
```

```
int main( void ){
//Memory allocation for single integer variable
//int *ptr = new int; //Dyanamic memory allocation: Garbage Value
//int *ptr = new int( ); //Dyanamic memory allocation: 0
int \*ptr = new int( 123 ); //Dyanamic memory allocation: 123

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

    //Memory deallocation for single integer variable
    delete ptr; //Dynamic memory deallocation
    ptr  = nullptr;
    return 0;

}
```

- Consider memory allocation and deallocation for array.

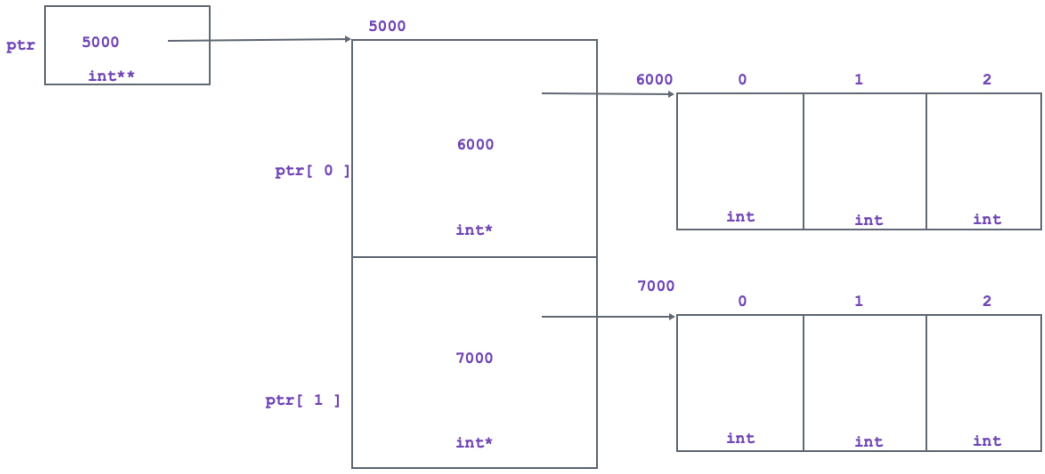


- Example 5

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

  for( int index = 0; index < 3; ++ index ){
    cout << "Enter number : ";
    cin >> ptr[ index ];
  }

  for( int index = 0; index < 3; ++ index )
```

```
    cout << ptr[ index ] << endl;

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

- Consider memory allocation and deallocation for multi dimensional array.



- Example 6:

```
  //Memory Allocation
  int *ptr = new int*[ 2 ];
  for( int index = 0; index < 2; ++ index )
    ptr[ index ] = new int[ 3 ];

  //Accept record for multi dimensional array

  //Print record for multi dimensional array

  //Memory Deallocation
  for( int index = 0; index < 2; ++ index )
    delete[] ptr[ index ];
  delete[ ] ptr;
```

- Example 7:

```
  int *ptr1 = new int;  //Single variable. Memory will be initialized
to garbage value
```

```
  int *ptr2 = new int(3);  //Single variable. Memory will be
initialized to 3

  int *ptr3 = new int[3];  //Array. Memory will be initialized to
garbage value
```

Difference between malloc and new

- malloc is function and new is operator.
- In case of failure malloc returns NULL but new operator throws bad_alloc exception.
- If we create dyanmic object using malloc then contructor do not call but if we create dynamic object using new operator then constructor gets called.
- Using malloc function, we can allocate space only on heap section but using new operator, we can allocate space on free store(stack section /data segment/heap section);

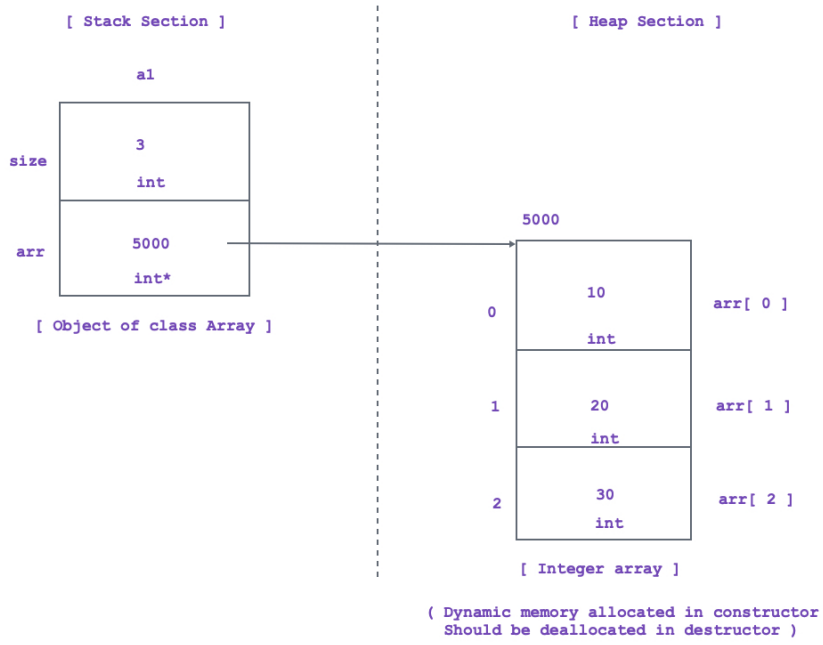
Array:

- Collection: Array/Stack/Queue/LinkedList/Tree/Graph/Hashtable etc.
- value stored inside collection is called as element.
- Array is linear data structure in which we can store multiple elements of same type in continous memory location.
- Types of array:
  - Single dimensional array
  - Multidimensional array
- To access elements of the array we should use integer index. Array index always begins with zero.
- Advantage of array
  - We can access elements of array randomly.
- limitations of array
  - It requires continous memory location.
  - We can not resize array dynamically.
  - Insertion and deletion of element in array is a time consuming task.
  - Using assignment operator, we can not copy contents of array into another array.
- Solution:
  - Use Linked List.
  - Encapsulate array inside class and perform operations on its object by considering it as a array.

```
class Array{
private:
  int arr[ 3 ];
};
int main( void ){
  Array a1; //a1 is object
  return 0;
}
```

Destructor

- Destructor is a member function of the class which is used to release the resources hold by the object.
- Destructor do not deallocate memory of object.
- Destructor is considered as special function of the class due to following reasons:
    - Its name is same as class name which precedes with tild(~) operator.
    - It doesnt take any parameter or return any value.
    - It is designed to call implicitly.
- Example:




```
#include<iostream>
using namespace std;
class Array{
private:
  int size;
  int *arr;
public:
  //Array *const this = &a1
  Array( void ){
    this->size = 0;
    this->arr = nullptr;
  }
  //Array *const this = &a1
  Array( int size ){
    cout << "Array( int size )" << endl;
    this->size = size;
    this->arr = new int[ size ];
  }
  //Array *const this = &a1
  void acceptRecord( void ){
    for( int index = 0; index < this->size; ++ index ){
```

```
        cout << "Enter element  :   ";
        cin >> this->arr[ index ];
      }
    }
    //Array *const this = &a1
    void printRecord( void ){
      for( int index = 0; index < this->size; ++ index )
        cout << this->arr[ index ] << endl;
    }
    //Array *const this = &a1
    ~Array( void ){ //Destructor
      if( this->arr != nullptr ){
        delete[] this->arr;
        this->arr = nullptr;
      }
    }
};
int main( void ){
    Array a1(3);    //Static memory allocation for object
    a1.acceptRecord( );
    a1.printRecord( );
    return 0;
}
```

- If we do not provide destructor for the class then compiler provide one destructor for the class by deafult. It is called default destructor.
- Destructor calling sequence is exactly opposite of constructor calling sequence.
- We can not declare destructor static/const/volatile. We can declare constructor as inline and virtual.
- We can overload constructor but we can not overload destructor.
- Note: Even though destructor is designed to call implicitly, we can call it explicitly too.

```
    //Array *const this = &a1
~Array( void ){   //Destructor
    if( this->arr != nullptr ){
      delete[] this->arr;
      this->arr = nullptr;
    }
}
```

Why we can not overload destructor?

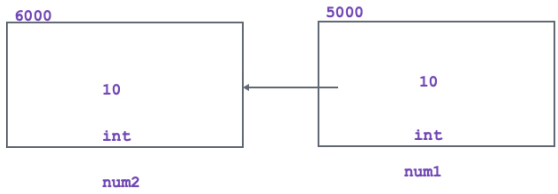
- To overload function, either number parameters / type parameters or order of type of parameters must be different.
- But destructor do not take any parameter. Hence we can not overload destructor.

Day 9

Shallow Copy

- Proecss of copying contents from source object into destination object as it is, is called as shallow copy.
- Shallow copy is also called as bitwise copy / bit-by-bit copy.
- Example 1:

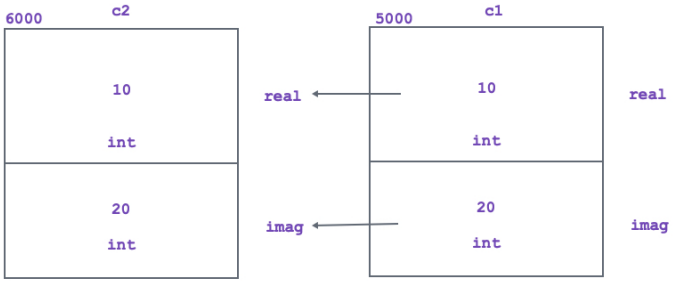
```
int num1 = 10;
int num2 = num1;
```



[Shallow Copy]

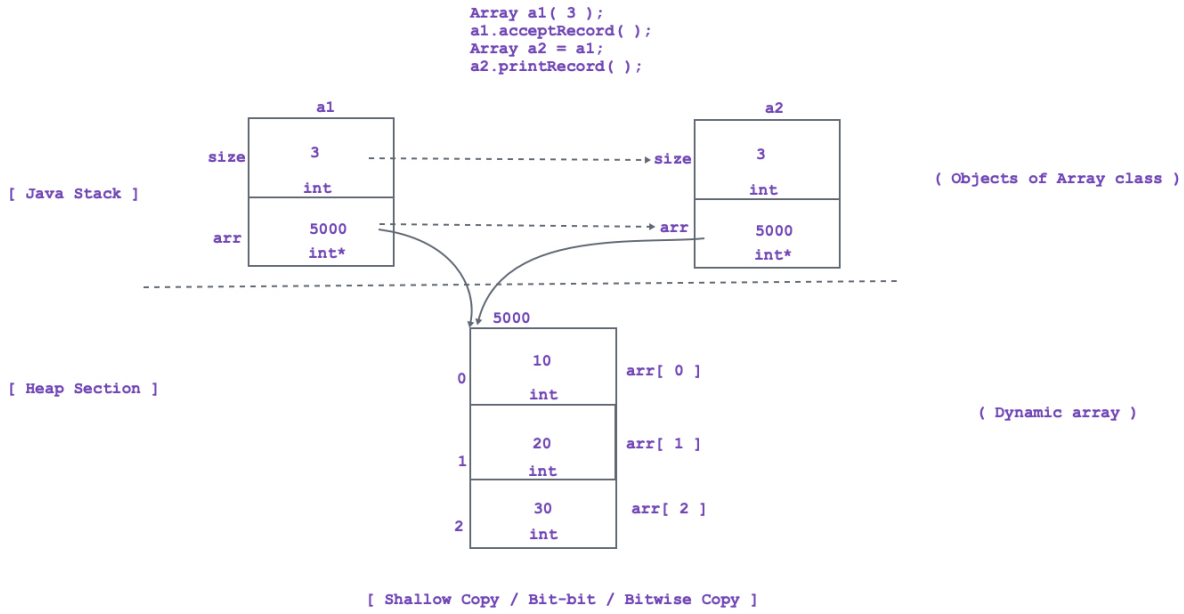
- Example 2:

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



[Shallow Copy]

- Example 3:



Copy constructor

- If we try to intialize newly created object from exisiting object of same class then on newly created object copy constructor gets called.
- Example:

```
int main( void ){
  Complex c1( 10, 20 ); //On C1, Parameterized ctor will call
  Complex c2;           //On c2, parameterless constructor will call
  Complex c3 = c1;  //On c3, copy constructor will call
  return 0;
}
```

- If we do not define copy constructor inside class, then compiler generate one copy constructor for the class by default. It is called as default copy constructor.
- Default copy constructor, by default creates shallow copy.
- Copy constructor is a parameterized constructor of the class which take single parameter of same type as a reference.
- Copy constructor is not a new type of constructor. It is parameterized constructor.
- Syntax:

```

```

```
class ClassName{
public:
  //const ClassName &other = reference of existing object
  //ClassName *const this = Address of newly created object
  ClassName( const ClassName &other ){
    //TODO: Shallow Copy or Deep Copy
  }
};
```

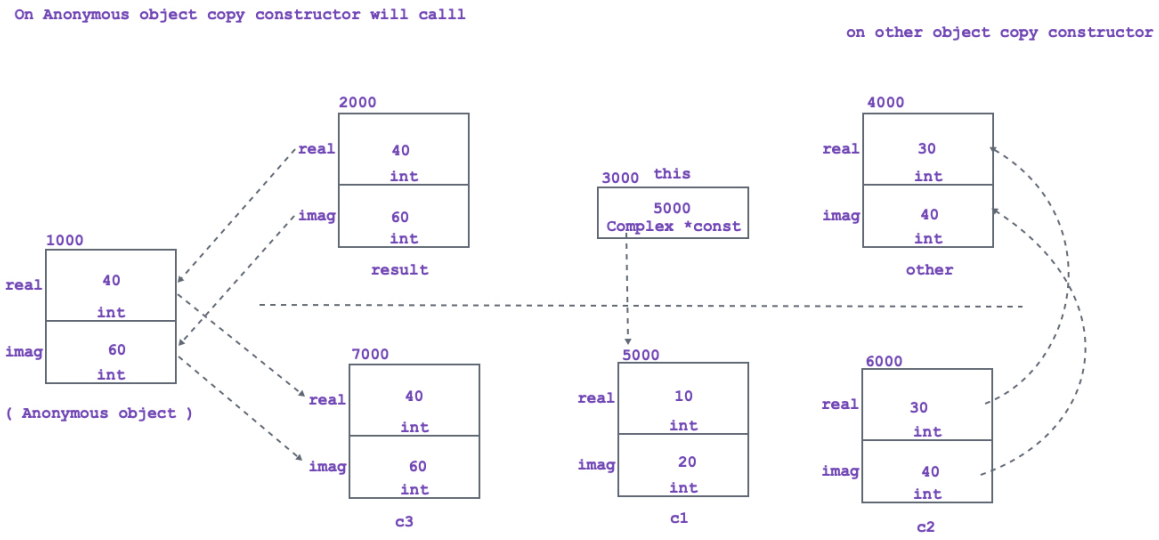
- Example:

```
class Complex{
private:
  int real;
  int imag;
public:
  Complex( void ){
    this->real = 0;
    this->imag = 0;
  }
  //const Complex &other = c1;
  //Complex *const this = &c3
  Complex( const Complex &other ){  //Copy Constructor
    this->real = other.real;    //Shallow Copy
    this->imag = other.imag;    //Shallow Copy
  }
  Complex( int real, int imag ){
    this->real = real;
    this->imag = imag;
  }
  void printRecord( void ){
    cout << "Real Number    :   " << this->real << endl;
    cout << "Imag Number    :   " << this->imag << endl;
  }
};
int main( void ){
  Complex c1( 10, 20 ); //On C1, Parameterized ctor will call
  Complex c2;           //On c2, parameterless constructor will call
  Complex c3 = c1;  //On c3, copy constructor will call
  return 0;
}
```

Copy constructor gets called in following conditions:

- If we pass object of a class as a argument to the function then on function parameter copy constructor gets called.
- If we return object from function by value then compiler implicitly generate anonymous object. On anonymous object compiler implicitly call copy constructor.

- If we initialize newly created object from exisiting object of same class then on newly created object copy constrcutor gets called.
- If we throw object then copy of the object gets created on stack frame. On that object copy constructor gets called.
- If we catch object object by value then on catching object copy constructor gets called.



use "-fno-elide-constructors" compiler option in project settings

In below conditions, copy of the object gets created

- If we pass object to the function by value.
- If we return object from function by value.
- If we initialize object from another object.
- If we assign object from another object.
- If we throw object
- If we catch object by value

Deep Copy

- If we make copy of the object with some modifications then such type of copy is called as deep copy.

Conditions to create deep copy.

- Class must contain at least one pointer
- Class must contain user defined destructor with deallocation.
- We must create copy of the object.

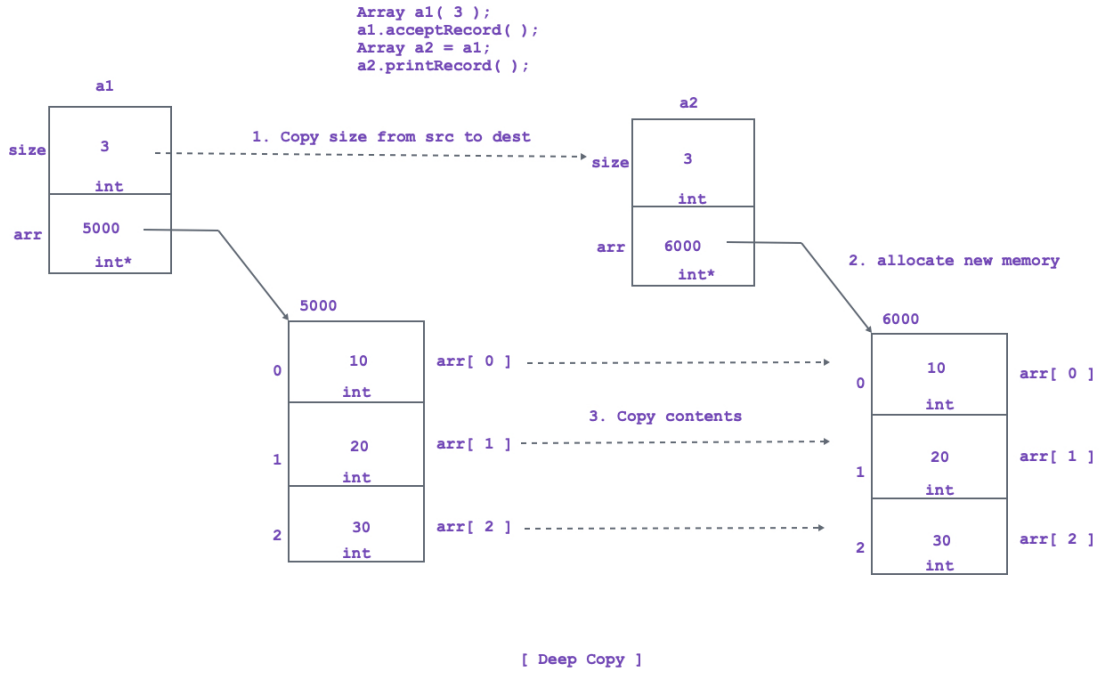
Steps to create deep copy

- Copy the required size from source object into destination object.

- Allocate new resource for the pointer of destincation object.
- Copy the contents from resource of source object into resource of destination object.

Where to create deep copy

- If we pass object to the function by value (Copy Constructor).
- If we return object from function by value (Copy Constructor).
- If we initialize object from another object (Copy Constructor).
- If we assign object from another object (operator=( ))
- If we throw object (Copy Constructor).
- If we catch object by value (Copy Constructor).



- Example 4:

```
//const Array &other = a1
//Array *const this = &a2
Array( const Array &other ){
    cout << "Array( const Array &other )" << endl;
    //1. Copy size
    this->size = other.size;
    //2. Allocate new resource
    this->arr = new int[ this->size ];
    //3. Copy the contents from source object into destination object.
    for( int index = 0; index < this->size; ++ index )
        this->arr[ index ] = other.arr[ index ];
}
```

Friend function:

- friend is keyword in C++.

- If we want to access private and protected members of the class inside non member function then we should declare non member function friend.
- We can use friend keyword only inside class.
- Example:

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

class Test{
private:
  int num1;
protected:
  int num2;
public:
  Test( void ){
    this->num1 = 10;
    this->num2 = 20;
  }
  friend int main( void );  //We can declare it in either
private/protected/public section
};
int main( void ){
  Test t;
  cout << "Num1   :   " << t.num1 << endl;
  cout << "Num2   :   " << t.num2 << endl;
  return 0;
}
```

- If we declare function as a friend inside class then it is not considered as a member of a class.
- Example:

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

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

```
  cout << "Num1   :   " << t.num1 << endl;
  cout << "Num2   :   " << t.num2 << endl;
}
int main( void ){
  //Test t;
  //t.print( );   //Not OK

  print( );
  return 0;
}
```

- We can declare same function friend into multiple classes:

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

class A{
private:
  int num1;
public:
  A( void ){
    this->num1 = 10;
  }
  friend void print( );
};
class B{
private:
  int num2;
public:
  B( void ){
    this->num2 = 20;
  }
  friend void print( );
};
void print( ){
  A a;
  cout << "Num1   :   " << a.num1 << endl;
  B b;
  cout << "Num2   :   " << b.num2 << endl;
}
int main( void ){
    print( );
  return 0;
}
```

- We can declare member function of a class as a friend inside another class.
- Example:

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

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

- If we want to access private members of the class inside all the member functions of another class then we should declare class as a friend.

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

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

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

- We can declare mutual friend classes but we can not declare mutual friend functions.
- Example:

```
class A{
private:
  int num2;
public:
  A( void );
  void showRecord( );
  friend class B;
};
class B{
private:
  int num1;
public:
  B( void );
  void displayRecord( );
  friend class A;
};
A::A( void ){
  this->num2 = 200;
}
```

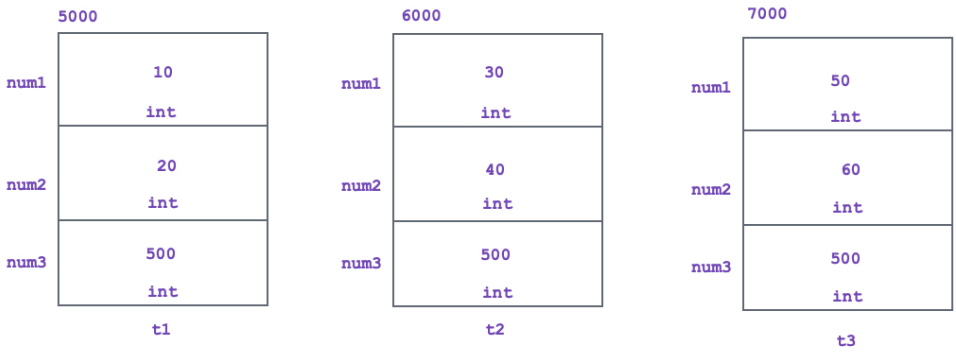
```
B::B( void ){
  this->num1 = 100;
}
void A::showRecord( void ){
  B obj;
  cout << "Num1   :   " << obj.num1 <<endl;
}
void B::displayRecord( ){
  A obj;
  cout << "Num2   :   " << obj.num2 <<endl;
}
int main( void ){
  A a;
  a.showRecord( );

  B b;
  b.displayRecord( );
  return 0;
}
```

- Real time example:

```
//Here class Remote is used before definition
class Remote; //Forward declaration

class Television{
  friend class Remote;
}
class Remote{

};
```

Static Data member

```
Test t1( 10, 20 );
Test t2( 30, 40 );
Test t2( 50, 60 );
```

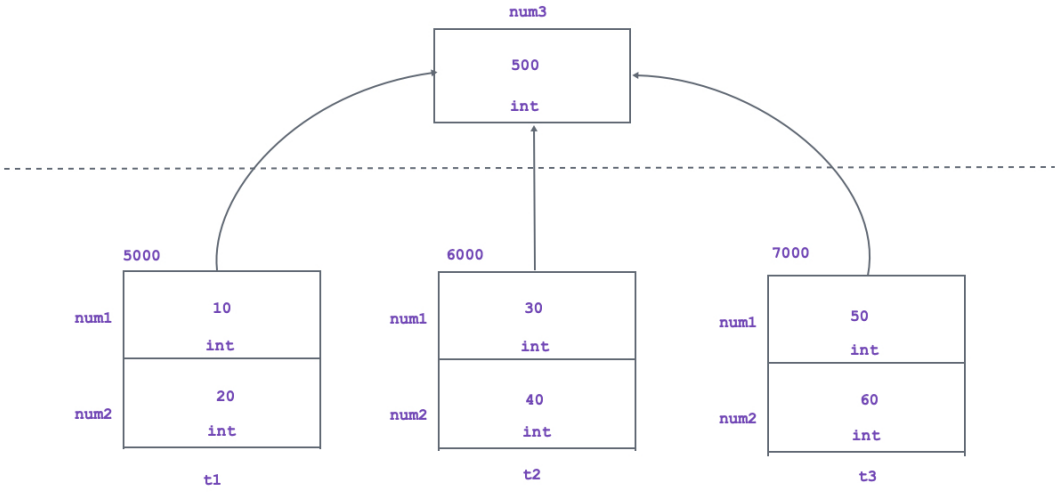


- Example:

```cpp
#include<iostream>
using namespace std;

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

```
    Test t2( 30, 40 );

    Test t3( 50, 60 );
    return 0;
}
```

- If we want to share value of any data member in all the objects of same class then we should declare data member static.



- Static data member get space during program loading once per class on data segment.
- if we create object of the class then only non static data member get space inside it. Hence size of object depends on size of all the data members declared inside class.
- Data member of the class which get space inside object is called as instance variable. In other words non static member is also called as instance variable.
- Instance variable get space once per object.
- To access instance variable either we should use object or pointer/reference to that object.
- Data member of the class which do not get space inside object is called as class level variable. In other words static member is also called as class level variable.
- Class level variable get space once per class.
- To access class level variable we should class name and :: operator.
- Example 1:

```
```

```
class A{
  int n1;
  int n2;
  static int count;
};
int main( void ){
  A a1,a2,a3;
  return 0
}
```

- Example 2:

```
class B{
  int n3;
  int n4;
  static int count;
};
int main( void ){
  B b1,b2,b3;
  return 0
}
```

- Example 3:

```
class C{
  int n5;
  int n6;
  static int count;
};
int main( void ){
  C c1,c2,c3;
  return 0
}
```

- if we want to declare data member static then we must provide global definition for it. Otherwisw linker will generate error.

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

  class Test{
  public:
    int num1;           //Instance variable
    int num2;           //Instance variable
    static int num3;    //Class level variable
  public:
```

```
  Test( void ){
    this->num1 = 0;
    this->num2 = 0;
  }
  Test( int num1, int num2 ){
    this->num1 = num1;
    this->num2 = num2;
  }
  void printRecord( void ){
    cout << "Num1 :   " << this->num1 << endl;
    cout << "Num2 :   " << this->num2 << endl;
    cout << "Num3 :   " << Test::num3 << endl;
  }
};
int Test::num3 = 500; //Global definition
int main( void ){
  Test t1;
  t1.printRecord( );
  return 0;
}
```

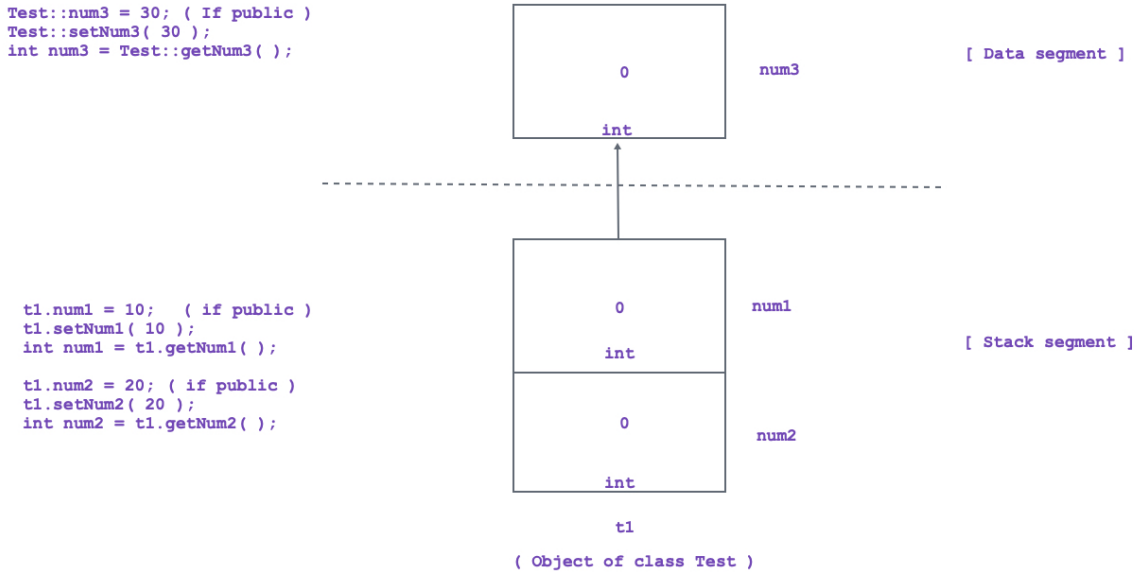
- We can declare static data member constant. Consider following code:

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

class Test{
public:
  int num1;           //Instance variable
  int num2;           //Instance variable
  const static int num3;  //Class level variable
public:
  Test( void ){
    this->num1 = 0;
    this->num2 = 0;
  }
  Test( int num1, int num2 ){
    this->num1 = num1;
    this->num2 = num2;
  }
  void printRecord( void ){
    cout << "Num1 :   " << this->num1 << endl;
    cout << "Num2 :   " << this->num2 << endl;
    cout << "Num3 :   " << Test::num3 << endl;
  }
};

const int Test::num3 = 500;   //Global definition

int main( void ){
  Test t1;
  t1.printRecord( );
```

```
    return 0;
}
```

Day 10

Static member function



- Example:

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

class Test{
private:
  int num1;           //Non static data member / Instance variable
  int num2;           //Non static data member / Instance variable
  static int num3;    //Static data member / Class level variable
public:
  Test( void ){
    this->num1 = 0;
    this->num2 = 0;
  }
  void setNum1( int num1 ){
    this->num1 = num1;
  }
  void setNum2( int num2 ){
    this->num2 = num2;
  }
```

```
    static void setNum3( int num3 ){
      Test::num3 = num3;
    }
    void printRecord( void ){
      cout << "Num1 :   " << this->num1 << endl;
      cout << "Num2 :   " << this->num2 << endl;
      cout << "Num3 :   " << Test::num3 << endl;

    }
};
int Test::num3 = 0;   //Global definition
int main( void ){
  Test t1;
  t1.setNum1( 10 );
  t1.setNum2( 20 );
  Test::setNum3( 30 );

  t1.printRecord( );
  return 0;
}
int main1( void ){
  Test t1;
  //t1.num1 = 10; //error: 'num1' is a private member of 'Test'
  //t1.num2 = 20; //error: 'num2' is a private member of 'Test'
    //Test::num3 = 30;//error: 'num3' is a private member of 'Test'
  t1.printRecord( );
  return 0;
}
```

- To access non static members of the class, we should define non static member function inside class.
- Non static member functions are designed to call on object. Hence it is also called as instace method.
- Since non static member functions / instance methods are designed to call on object/instance, it gets this pointer. Since non static member function get this pointer, we can access static as well as non static members inside non static member function.
- To access static member of the class, we should define static member function inside class.
- Static member functions are designed to call on class name. Hence it is also called as class level method.
- Since static member functions / class level methods are designed to call on class name, it doesn't get this pointer. Since static member function doesn't get this pointer, we can access only static members inside static member function.
- Static member function do not get this pointer but we can create object inside static member function.
- Using object, we can access non static members inside static member function.

- Example:

```
class Test{
  private:
    int num1;
    static int num2;
  public:
    Test( void ) : num1( 10 ){
    }
    static void print( void ){
      Test t;
      cout << "Num1 : " << t.num1 << endl;
      cout << "Num2 : " << Test::num2 << endl;
    }
};
int Test::num2 = 20;  //Global definition
int main( void ){
  Test::print( );
  return 0;
}
```

Why static member function do not get this pointer?

- If we call non static member function on object then non static member function get this pointer.
- Static member function is designed to call on class name.
- Since static member function is not designed to call on object, it does not get this pointer.

Can we declare static member function constant?

- If we dont want to modify state on only current object inside member function then we should declare that member function constant. In other words, constant member functions are designed to call on object.
- Static member function is designed to call on class name.
- Since static member function is not designed to call on object, we can not declare static member function constant.

Conclusion

- In C++, we can declare static data member constant but we can not declare static member function constant.
- If there is need to use this pointer inside member function then it should be non static otherwise it should be static.

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

  class Math{
```

```
    public:
      static const double PI;
    public:
      static double pow( double base, int index ){
        double result = 1;
        for( int count = 1; count <= index ; ++ count )
          result = base * result;
        return result;
      }
    };
    const double Math::PI = 3.14;
    int main( void ){
      double result = Math::pow( 2.0, 3 );
      cout << "Result :   "<<result <<endl;
      return 0;
    }
```

- We can not declare below functions static:
    - constructor
    - destructor
    - constant member function
    - volatile member function
    - virtual member function
    - main function (other global functions can be static)

How will you write code to count number of instances created from class?

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

  class InstanceCounter{
  private:
    static int count;
  public:
    InstanceCounter( ){
      InstanceCounter::count = InstanceCounter::count + 1;
    }
    static int getCount( void ){
      return InstanceCounter::count;
    }
    ~InstanceCounter( ){
      InstanceCounter::count = InstanceCounter::count - 1;
    }
  };
  int InstanceCounter::count = 0;
  int main( void ){
    InstanceCounter c1, c2, c3;
    cout << "Instance Counter   :   " << InstanceCounter::getCount( ) <<
endl;
```

```
    return 0;
  }
```

Anonymous class

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

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

Operator Overloading

Token

- Token is a basic unit of a program.
- Classification of tokens:
    - Identifier
    - Keywords
    - Constant
    - Operator
    - Separator / punctuators
- Classfication of Operators
    - Unary Operators
        - An operator which requires only one operand(e.g sizeof(a)) is called as unary operator.
        - Example: sizeof, typeid, ++, --, !(Logical NOT), ~, +, -, * etc
    - Binary Operators
        - An operator which requires two operands(e.g a + b) is called as binary operator.
        - Arithmetic operators
            - +, -, *, /, %
        - Relational operators
            - <, <= >, >=, ==, !=

        - Logical operators
            - &&(Logical AND), || (Logical OR),
        - Bitwise operators
            - &(Bitwise AND), | (Bitwies OR), ^(Bitwise XOR), <<, >>
        - Assignment operators
            - =, short hand operators(+=, -=, *= etc)
    - Ternary Operators
        - An operator which requires three operands is called as ternary operator.
        - Conditional operator(? :)

- Consider code in C programming language:
- Example 1:

```
int main( void ){
  int num1 = 10;
  int num2 = 20;
  int result = num1 + num2; //OK
  return 0;
}
```

    - In C, we can use operator with the variables of fundamental types.

- Example 2:

```
struct Point{
  int xPos;
  int yPos;
};
int main( void ){
  struct Point pt1 = { 10, 20 };  //OK
  struct Point pt2 = { 30, 40 };  //OK
  struct Point pt3; //OK
  pt3.xPos = pt1.xPos + pt2.xPos;  //OK
  pt3.yPos = pt1.yPos + pt2.yPos;  //OK
  return 0;
}
int main1( void ){
  struct Point pt1 = { 10, 20 };  //OK
  struct Point pt2 = { 30, 40 };  //OK
  struct Point pt3; //OK
  pt3 = pt1 + pt2;  //Not OK
  return 0;
}
```

    - In C, we cannot use operator with objects of user defined type directly.

- Consider code in C++ programming language:

- Example 1:

```
int main( void ){
  int num1 = 10;
  int num2 = 20;
  int result = num1 + num2; //OK
  return 0;
}
```

  - In C++, we can use operator with the variables of fundamental types.

- Example 2:

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

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

  int main( void ){
    Complex c1( 10, 20 );
    Complex c2( 30, 40 );
    Complex c3;
    c3 = c1 + c2;   //error: invalid operands to binary expression
('Complex' and 'Complex')
    c2.printRecord( );
    return 0;
  }
```

- If we want to use operator with the objects of user defined type(structure, class etc.) then we should overload operator.

- To overload operator, we should define operator function.

- operator is keyword in C++ which is used to define operator function.
- We can define operator function using 2 ways:
    - Member function
    - Non member function
- By defining operator function, we are increasing capability of exsiting operators. This process of givining extension to the meaning of the operator is called as operator overloading.
- Consider Example using member function:

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

class Complex{
private:
  int real;
  int imag;
public:
  Complex( void ){
    this->real = 0;
    this->imag = 0;
  }
  Complex( int real, int imag ){
    this->real = real;
    this->imag = imag;
  }
  //Complex other = c2
  //Complex *const this = &c1
  Complex operator+( Complex other ){
    Complex result;
    result.real = this->real + other.real;
    result.imag = this->imag + other.imag;
    return result;
  }
  void printRecord( void ){
    cout << "Real Number  :   " << this->real <<endl;
    cout << "Imag Number  :   " << this->imag <<endl;
  }
};

int main( void ){
  Complex c1( 10, 20 );
  Complex c2( 30, 40 );
  Complex c3;
  c3 = c1 + c2;   // c3 = c1.operator+( c2 )
  c3.printRecord( );
  return 0;
}
```

- Consider Example using non member function:

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

class Complex{
private:
  int real;
  int imag;
public:
  Complex( void ){
    this->real = 0;
    this->imag = 0;
  }
  Complex( int real, int imag ){
    this->real = real;
    this->imag = imag;
  }
  void printRecord( void ){
    cout << "Real Number  :   " << this->real <<endl;
    cout << "Imag Number  :   " << this->imag <<endl;
  }
  friend Complex operator+( Complex c1, Complex c2 );
};
Complex operator+( Complex c1, Complex c2 ){
  Complex result;
  result.real = c1.real + c2.real;
  result.imag = c1.imag + c2.imag;
  return result;
}
int main( void ){
  Complex c1( 10, 20 );
  Complex c2( 30, 40 );
  Complex c3;
  c3 = c1 + c2;   // c3 = operator+( c1, c2 )
  c3.printRecord( );
  return 0;
}
```

- Using operator overloading we can not create user defined operators rather we can increase capability of existing operators.

Limitations of operator overloading

- We can not overload below operators using member function as well as non member function
    - dot(.) / meber selection operator
    - .* (pointer to member selection operator)
    - sizeof operator
    - ::(scope resolution operator)

    - Conditional (? :) / Ternary operator
    - typeid operator
    - static_cast operator
    - dyanmic_cast operator
    - const_cast operator
    - reinterpret_cast operator

- We can not overload below operators using non member function but we can overload it using member function

    - Assignment operator(=)
    - Index / subscript operator
    - Call / Function call operator[()]
    - Arrow(->) operator

- using operator overloading, we can change meaning of the operator.

Arithmetic operator overloading

- Example 1:

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

- Example 2:

```
Complex c1( 10, 20 );
Complex c2( 40, 30 );
Complex c3;
c3 = c1 + c2; //c3 = operator+( c1, c2 );  //Using non member function
```

- Example 3:

```
Complex c1( 10, 20 );
Complex c2( 40, 30 );
Complex c3;
c3 = c1 - c2; //c3 = c1.operator-( c2 );  //Using member function
```

- Example 4:

```
Complex c1( 10, 20 );
Complex c2( 40, 30 );
```

```
Complex c3;
c3 = c1 - c2; //c3 = operator-( c1, c2 );  //Using non member function
```

- Example 5:

```
Complex c1( 10, 20 );
Complex c2( 40, 30 );
Complex c3;
c3 = c1 * c2; //c3 = c1.operator*( c2 );  //Using member function
```

- Example 6:

```
Complex c1( 10, 20 );
Complex c2( 40, 30 );
Complex c3;
c3 = c1 * c2; //c3 = operator*( c1, c2 );  //Using non member function
```

- Example 7:

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

- Example 8:

```
Complex c1( 10, 20 );
Complex c2( 40, 30 );
Complex c3;
c3 = c1 / c2; //c3 = operator/( c1, c2 );  //Using non member function
```

Relational operator overloading

- Example 1:

```
Complex c1( 10, 20 );
Complex c2( 10, 20 );
bool status = c1 == c2; //status = c1.operator==( c2 );  //Using
member function
```

- Example 2:

```
Complex c1( 10, 20 );
Complex c2( 10, 20 );
bool status = c1 == c2; //status = operator==( c1, c2 );  //Using non
member function
```

- Example 3:

```
Complex c1( 10, 20 );
Complex c2( 10, 20 );
bool status = c1 != c2; //status = c1.operator!=( c2 );  //Using
member function
```

- Example 4:

```
Complex c1( 10, 20 );
Complex c2( 10, 20 );
bool status = c1 != c2; //status = operator!=( c1, c2 );  //Using non
member function
```

- Example 5:

```
Complex c1( 10, 20 );
Complex c2( 10, 20 );
bool status = c1 < c2; //status = c1.operator<( c2 );  //Using member
function
```

- Example 6:

```
Complex c1( 10, 20 );
Complex c2( 10, 20 );
bool status = c1 < c2; //status = operator<( c1, c2 );  //Using non
member function
```

- Example 7:

```
Complex c1( 10, 20 );
Complex c2( 10, 20 );
bool status = c1 > c2; //status = c1.operator>( c2 );  //Using member
function
```

- Example 8:

```
Complex c1( 10, 20 );
Complex c2( 10, 20 );
bool status = c1 > c2; //status = operator>( c1, c2 );  //Using non
member function
```

Unary operator overloading

- Example 1:

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

- Example 2:

```
Complex c1( 10, 20 );
Complex c2 = ++ c1; //c2 = operator++( c1 );  //Using non member
function
```

- Example 3:

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

- Example 4:

```
Complex c1( 10, 20 );
Complex c2 = c1 ++; //c2 = operator++( c1, 0 );  //Using non member
function
```

Extraction operator overloading

- cin: character input, which represents keyboard.
- extraction operator(>>) is designed to use with cin.
- If we want to accept state the object from keybaord using cin then we should overload extraction operator.
- Consider call using member function:

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

- Here to accept record for c1, we should define operator>>() function inside istream class, which is not recommended. So we will not overload it using member function.

- Consider call using non member function:

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

- Here to accept record for c1, we should define operator>>() function global, which is possible for us. Hence we will overload operator >> using non member function.

- Consider another example:

```
Complex c1;
Complex c2;
cin >> c1 >> c2; //operator>>( operator>>( cin, c1 ), c2 );
```

- General Syntax:

```
class ClassName{
  friend istream& operator>>( istream &cin, ClassName &other );
};
istream& operator>>( istream &cin, ClassName &other ){
  //TODO: accept record using other reference variable
  return cin;
}
```

Insertion operator overloading

- cout: character output, which represents monitor.
- insertion operator(<<) is designed to use with cout.
- If we want to print state the object on monitor using cout then we should overload insertion operator.
- Consider call using member function:

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

- Here to print record of c1, we should define operator<<() function inside ostream class, which is not recommended. So we will not overload it using member function.

- Consider call using non member function:

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

- Here to print record of c1, we should define operator<<() function global, which is possible for us. Hence we will overload operator << using non member function.

- Consider another example:

```
Complex c1( 10, 20 );
Complex c2( 30, 40 );
cout << c1 << c2; //operator<<( operator<<( cout, c1 ), c2 );
```

- General Syntax:

```
class ClassName{
  friend ostream& operator<<( ostream &cout, ClassName &other );
};
ostream& operator<<( ostream &cout, ClassName &other ){
  //TODO: print record using other reference variable
  return cout;
}
```

Assignment operator overloading

- Example 1:

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

- Process of storing value during declaration of variable is called as initialization.
- Example 2:

```
Complex c1( 10, 20 ); //on c2 parameterized constructor will call
Complex c2 = c1;  //on c2 copy constructor will call.
Complex c3( c1 );  //on c3 copy constructor will call.
```

- If we "initialize" object from another object of same class the copy constructor gets called.
- Syntax:

```
class ClassName{
public:
  ClassName( const ClassName &other ){
      //TODO: Shallow / Deep Copy
  }
}
```

- Example 3:

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

- Process of storing value after declaration of variable is called as assignment.
- Example 4:

```
Complex c1( 10, 20 ); //on c2 parameterized constructor will call
Complex c2;  //on c2 parameterless constructor will call.
c2 = c1;  //c2.operator=( c1 );
```

- If we assign object to another object of same class the assignment operator function gets called.
- If we do not define assignment operator function for the class then compiler generates one assignment operator function for the class by default, it is called as default assignment operator function.
- Default copy constructor and default assignment operator function by default creates shallow copy.
- Example 5:

```
Complex c1( 10, 20 ); //on c2 parameterized constructor will call
Complex c2;  //on c2 parameterless constructor will call.
Complex c3;  //on c3 parameterless constructor will call.
c3 = c2 = c1;  //c2.operator=( c2.operator=( c1 ), c2 );
```

- Syntax:

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

- We get following functions for any class by default:
    - Constructor
    - Destructor
    - Copy constructor
    - Assignment operator function

Index / Subscript operator overloading

- Array
    - Definition: It is linear / sequential data structure / collection in which we can store multiple elements of same type in continuous memory location.
    - Types:
        - Single dimensional array
        - Multi dimensional array
    - To access elements of array, we should use integer index. Array index always begins with 0.
    - We can create array statically as well as dynamically.

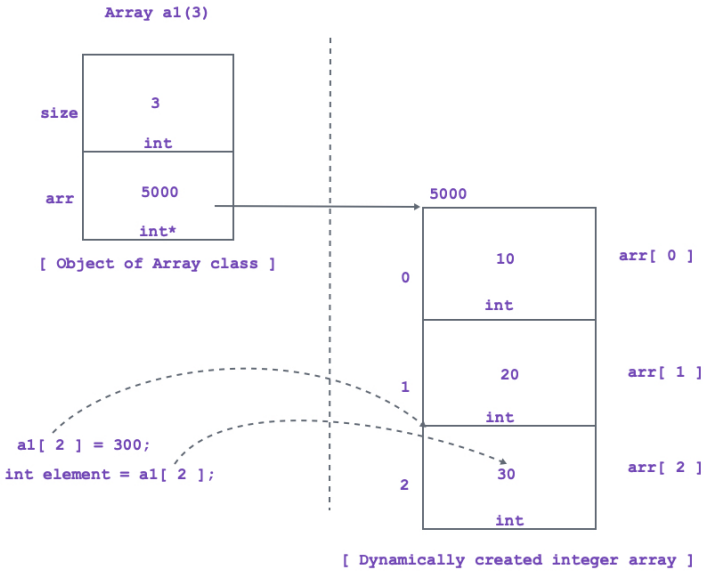
```
int arr[ 3 ]; //Static memory allocation
int *ptr = new int[ 3 ];   //Dynamic memory allocation
```

    - Advnatage of Array over linkked
        - We can access elements of array randomly.
    - Limitations of Array
        - It requires continous memory
        - We can not resize array
        - Element insertion and deletion is time consuming task
        - Using assignment operator we can not copy elements of array into another array.
- We can overcome limitations of array using 2 ways:
    - Use LinkedList instead of Array.
    - Encapsulate(declare variable as a data meber/member function) array inside class. Create object of the class and consider that object as a array.
- If we want to consider object as a array then we should overload subscript / index operator.
- If we want to use subscript operator with object at R.H.S of assignment operator then expression should return value.
- Example:

```
int main( void ){
  Array a1( 3 );

  cin >> a1;  //operator>>( cin, a1 )

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

  cout << "Element    :   " << element << endl;

  cout << a1; //operator<<( cour, a1 )
```

```
    return 0;
}
```

- If we want to use subscript operator with object at L.H.S of assignment operator then expression should not return value. Rather it should return pointer/reference of the memory location.
- Example:




```
int main( void ){
  Array a1( 3 );

  cin >> a1;    //operator>>( cin, a1 )

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

  cout << a1;   //operator<<( cour, a1 )
  return 0;
}
```

- Consider below code:

```
#include<iostream>
using namespace std;
class Array{
private:
    int size;
    int *arr;
```

```
public:
    Array( void  ){
        this->size = 0;
        this->arr = nullptr;
    }
    Array( int size  ){
        this->size = size;
        this->arr = new int[ this->size ];
    }
    Array( const Array &other){
        this->size = other.size;
        this->arr = new int[ this->size ];
        for( int index = 0; index < this->size; ++ index )
            this->arr[ index ] = other.arr[ index ];
    }
    //const Array &other = a1
    //Array* const this = &a2
    Array& operator=( const Array &other ){
        this->~Array( );
        this->size = other.size;
        this->arr = new int[ this->size ];
        for( int index = 0; index < this->size; ++ index )
            this->arr[ index ] = other.arr[ index ];
        return(*this);
    }
    //Array *const this
    int& operator[ ]( int index ){
        return this->arr[ index ];
    }
    friend istream& operator>>( istream &cin, Array &other ){
        for( int index = 0; index < other.size; ++ index ){
            cout<<"Enter element    :   ";
            cin >> other.arr[ index ];
        }
        return cin;
    }
    friend ostream& operator<<( ostream &cout, Array &other ){
        for( int index = 0; index < other.size; ++ index ){
            cout << other.arr[ index ] <<endl;
        }
        return cout;
    }
    ~Array( void ){
        if( this->arr != nullptr ){
            delete[] this->arr;
        }
        this->arr = nullptr;
    }
};
int main( void ){
    Array a1( 3 );

    cin >> a1;  //operator>>( cin, a1 )
```

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

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

      cout << a1; //operator<<( cour, a1 )
      return 0;
}
```

Call operator / function call operator overloading

- If we want to consider object as a function then we should overload call/function call operator.
- If we consider object as a function then such object is called as function object / functor.
- Example:

```
class Complex{
private:
  int real;
  int imag;
public:
  Complex( void ) : real( 0 ), imag( 0 ){
  }
  void operator()( int real, int imag ) {
    this->real = real;
    this->imag = imag;
  }
  friend ostream& operator<<( ostream &cout, const Complex &other ){
    cout << "Real Number    :   "<< other.real <<endl;
    cout << "Imag Number    :   "<< other.imag <<endl;
    return cout;
  }
};
int main( void ){
  Complex c1;

  c1( 10, 20 ); //c1. operator()( 10, 20 );

  cout << c1;   //operator<<( cout, c1 );
  return 0;
}
```

How will you swap 2 numbers without third variable(ref: use +/- or bitwise operator)

Template

- In C++, if we want to write typesafe generic code then we should use template.

- template is keyword in C++.
- In C++, by passing data type as a argument, we can write generic code. Hence paramerized type is called as template.
- Example:

```
template<typename T>    //T: Type Parameter
void swap_object( T &object1, T &object2 ){
  T temp = object1;
  object1 = object2;
  object2 = temp;
}

int main( void ){
  char ch1 = 'A';
  char ch2 = 'B';

  swap_object<char>( ch1, ch2 );
  //char: Type argument
  //ch1,ch2: function argument

  cout << "ch1  :   " << ch1 << endl;
  cout << "ch2  :   " << ch2 << endl;
  return 0;
}
```

- We can use typename and class keyword interchangably.

```
template<class T>   //T: Type Parameter
void swap_object( T &object1, T &object2 ){
  T temp = object1;
  object1 = object2;
  object2 = temp;
}
```

- Process of identifing type and passing it as a argument implicilty to the function is called as type inference.

```
swap_object<char>( ch1, ch2 );  //OK

swap_object( ch1, ch2 );  //OK
```

- Template feature is designed for the data structure.
- Using template, we can not reduce code size or execution time rather we can reduce developers effort/ development time.

- Types of template:
    - Function Template
    - Class Template

Function template

```
template<class T>   //T: Type Parameter
void swap_object( T &object1, T &object2 ){
  T temp = object1;
  object1 = object2;
  object2 = temp;
}
int main( void ){
  int a = 10;
  int b = 20;

  swap_object<int>( a, b );

  cout << "a    :   " << a << endl;
  cout << "b    :   " << b << endl;
  return 0;
}
```

Class Template

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

template<class T>
class Array{
private:
    int size;
    T *arr;
public:
    Array( void  ){
        this->size = 0;
        this->arr = nullptr;
    }
    Array( int size  ){
        this->size = size;
        this->arr = new T[ this->size ];
    }
    Array( const Array &other){
        this->size = other.size;
        this->arr = new T[ this->size ];
        for( int index = 0; index < this->size; ++ index )
            this->arr[ index ] = other.arr[ index ];
```

```
    }
    //const Array &other = a1
    //Array* const this = &a2
    Array& operator=( const Array &other ){
        this->~Array( );
        this->size = other.size;
        this->arr = new T[ this->size ];
        for( int index = 0; index < this->size; ++ index )
            this->arr[ index ] = other.arr[ index ];
        return(*this);
    }
    //Array *const this
    T& operator[ ]( int index ){
        return this->arr[ index ];
    }

    ~Array( void ){
        if( this->arr != nullptr ){
            delete[] this->arr;
        }
        this->arr = nullptr;
    }
    friend istream& operator>>( istream &cin, Array<T> &other ){
        for( int index = 0; index < other.size; ++ index ){
            cout<<"Enter element    :   ";
            cin >> other.arr[ index ];
        }
        return cin;
    }
    friend ostream& operator<<( ostream &cout, Array<T> &other ){
        for( int index = 0; index < other.size; ++ index ){
            cout << other.arr[ index ] <<endl;
        }
        return cout;
    }
};
int main( void ){
    Array<string> a1( 3 );

    cin >> a1;

    cout << a1;
    return 0;
}
int main2( void ){
    Array<double> a1( 3 );

    cin >> a1;

    cout << a1;
    return 0;
}
int main1( void ){
    Array<int> a1( 3 );
```

```
    cin >> a1;

    cout << a1;
    return 0;
}
```

Standard Template Library(STL)

- The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions.
- STL has 4 components:
    - Algorithms
        - https://en.cppreference.com/w/cpp/algorithm
    - Containers
        - https://en.cppreference.com/w/cpp/container
    - Functors
        - https://en.cppreference.com/w/cpp/utility/functional
    - Iterators
        - https://en.cppreference.com/w/cpp/iterator

C++ STL Containers

- A container is an object that stores a collection of objects of a specific type
- Types of STL Container in C++
    - Sequential Containers
    - Associative Containers
    - Unordered Associative Containers

OOPS

- Object oriented programming structure / system
- It is not a syntax rather it is a object oriented thought process / concept.
- Alan Kay coined the term “Object Oriented Programming” in the 1960s.
    - https://medium.com/javascript-scene/the-forgotten-history-of-oop-88d71b9b2d9f
- In object-oriented programming (OOP), there are three fundamental concepts:
    - Object-Oriented Analysis (OOA)
    - Object-Oriented Design (OOD)
    - Object-Oriented Programming (OOP).

Object-Oriented Analysis (OOA):

- OOA is the first phase in the OOP process. It focuses on understanding and defining the problem domain, the real-world entities involved, and their relationships.
- The goal of OOA is to create a conceptual model of the problem domain, often represented using UML (Unified Modeling Language) diagrams, such as class diagrams and use case diagrams.
- Example:

- Suppose you are developing software for a library management system. During OOA, you identify key entities like "Book," "Library," and "Member." You also define their attributes and relationships.
- For example:
  - Book class: Attributes - Title, Author, ISBN, Genre; Relationships - Can be borrowed by a Member.
  - Library class: Attributes - Name, Address; Relationships - Contains Books.
  - Member class: Attributes - Name, Member ID; Relationships - Borrows Books.

Object-Oriented Design (OOD):

- OOD is the second phase in OOP, where you take the conceptual model created in OOA and refine it into a detailed, implementable design. In this phase, you define the classes, their methods, and the interactions between objects.
- The goal of OOD is to create a design that is modular, maintainable, and efficient.
- Example:
  - Based on the OOA, during OOD, you design classes like "Book," "LibraryMember," and "LibraryStaff." You define the methods and attributes of each class. For the "Book" class, you might have methods like "checkOut" and "return," and attributes like "title" and "author." You also consider how these classes will interact, such as how a "LibraryStaff" object will call the "checkOut" method on a "Book" object when a book is borrowed.

```
// Book class
class Book {
private:
    string title;
    string author;
    string ISBN;
    string genre;
public:
    // Constructor, getters, and setters
    // Other methods like displaying book details
};

// Library class
class Library {
private:
    string name;
    string address;
    vector<Book> books; // A library can contain multiple books
public:
    // Constructor, methods to add and remove books, etc.
};

// Member class
class Member {
private:
```

```
        string name;
        int memberID;
        vector<Book> borrowedBooks; // A member can borrow multiple
  books
    public:
        // Constructor, methods to borrow and return books, etc.
    };
```

Object-Oriented Programming (OOP):

- OOP is the final phase where you implement the design created during OOD using a programming language like Java, Python, or C++. In this phase, you write the actual code for the classes, methods, and their interactions based on the design.
- Example:
    - In OOP, you would write code to create instances of the classes like "Book," "LibraryMember," and "LibraryStaff." You would implement methods like "checkOut" and "return" with actual code that performs the desired actions.

```
  int main() {
      // Create a library
      Library myLibrary("Central Library", "123 Main St");

      // Create books and add them to the library
      Book book1("Introduction to C++", "John Smith", "123456789",
"Programming");
      Book book2("The Art of Fiction", "Jane Doe", "987654321",
"Fiction");
      myLibrary.addBook(book1);
      myLibrary.addBook(book2);

      // Create members
      Member member1("Alice", 101);
      Member member2("Bob", 102);

      // Members can borrow books
      member1.borrowBook(book1);
      member2.borrowBook(book2);

      // Perform other operations like returning books, displaying
book details, etc.

      return 0;
  }
```

Major and minor pillars of oops

- Grady Booch is best known for developing the Unified Modeling Language with Ivar Jacobson and James Rumbaugh.
- According to Grady Booch, there are 4 major and 3 minor pillars of oops:

4 major parts / elements / features/ pillars of oops

- There are four major pillars of the oops:
    - Abstraction
    - Encapsulation
    - Modularity
    - Hierarchy
- By major, we mean that a language without any one of these element is not object oriented.

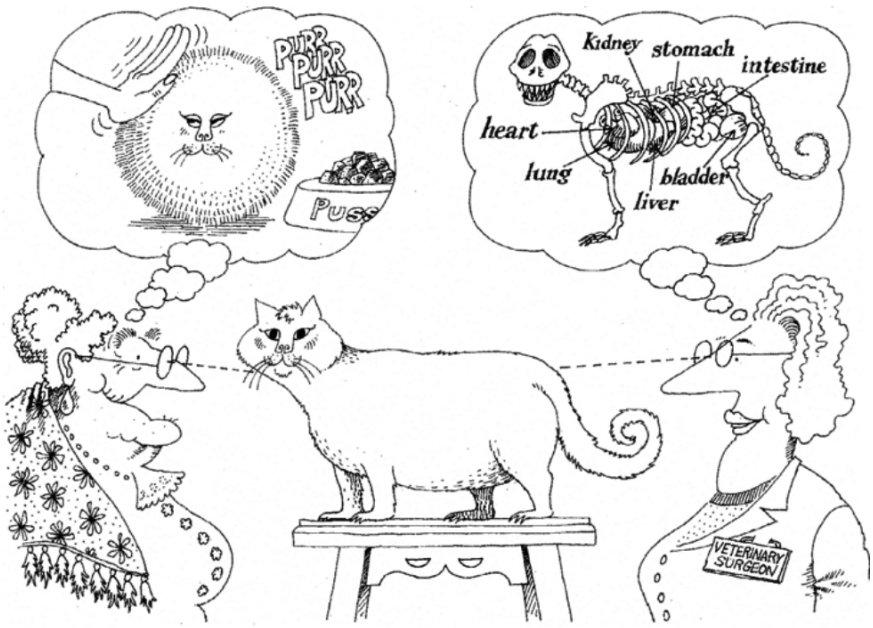
3 minor parts / elements / features/ pillars of oops

- There are three minor pillars of the oops:
    - Typing
    - Concurrency
    - Persistence
- By minor, we mean that each of these elements is a useful, but not essential.

Abstraction

- It is a major pillar of oops.
- Process of getting essential things from object is called as abstraction.
- Main goal of abstraction is to achieve simplicity.

- According to Grady Booch, Abstraction focuses on the essential characteristics of some object, relative to the perspective of the viewer.



- Abstraction in C++:

```
int main( void ){
  Complex c1;
  c1.acceptRecord( );
  c1.printRecord( );
  return 0;
}
```

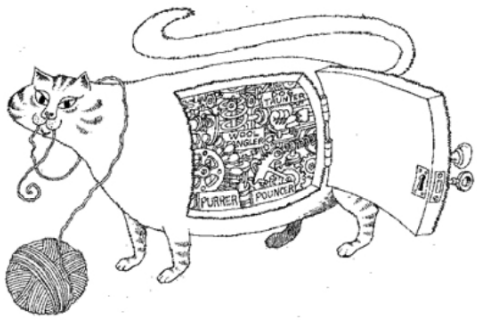
Encapsulation

- It is a major pillar of oops.
- Definition:
  - Binding of data(data member) and code(member function) together is called as encapsulation.
  - Implementation of abstraction is called as encapsulation.
- Encapsulation in C++:

```
class Complex{
private:
  int real;
  int imag;
public:
  void acceptRecord( void ){
    //TODO
```

```
    }
    void printRecord( void ){
      //TODO
    }
};
```

- Abstraction focuses on the observable behavior of an object, whereas encapsulation focuses on the implementation that gives rise to this behavior.



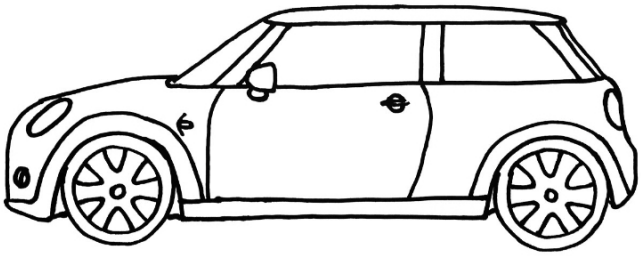
Encapsulation hides the details of the implementation of an object.

- Main goal of encapsulation is to hide the data.
    - Process of declaraing data member private is called as data hiding.
    - Data hiding is also called as data encapsulation.

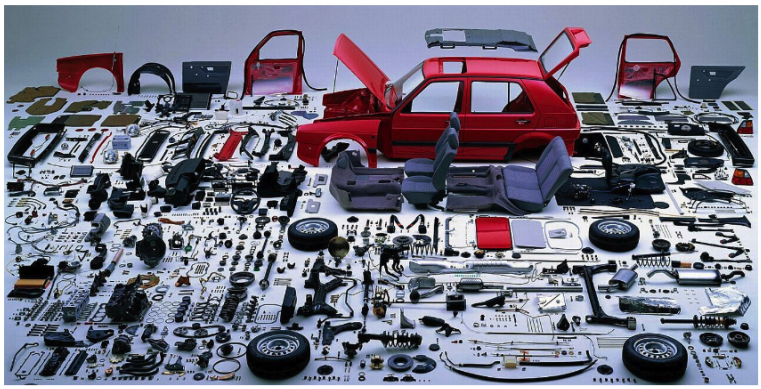
Modularity

- It is a major pillar of oops.

- Consider car:



- Consider car disassembly:



- Process of developing application with the help of small units/parts/module is called as modularity.
- Main goal of modularity is to minimize module dependency.
- In C++, you can achieve modularity through various mechanisms, including:
    - Functions
    - Classes and Objects
    - Namespaces
    - Header Files

Hierarchy

- It is a major pillar of oops.

Abstractions form a hierarchy.

- Level / order / ranking of abstraction is called as hierarchy.
- Main goal of hierarchy is to achieve reusability.
- Benefits of core reusability:
    - To reduce development time
    - To reduce development cost
    - To reduce developers effort
- Types of hierarchy in oops:
    - Has-a
        - also called as parts-of hierarchy
        - It represents Association
    - Is-a
        - Also called as kind of hierarchy
        - It represents generalization / inheritance
    - Use-a
        - It represents dependency
    - Creates-a Hierarchy
        - It represents instantiation

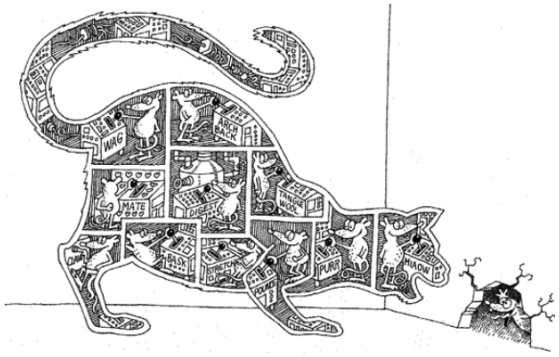
Typing

- It is a minor pillar of oops.
- Typing is also called as polymorphism.
- Polymorphism definition:
    - An ability of an object to take multiple forms is called as polymorphism.
- Main goal of typing / polymorphism is to reduce maintenance of the system.

- Types of polymorphism:
    - Compile time polymorphism
        - Also called as static typing
        - We can achieve it using 3 ways:
            - Function overloading
            - Operator overloding
            - Template
    - Run time polymorphism
        - Also called as dynamic typing
        - We can achieve it using:
            - Function overriding
- The opposite of polymorphism is monomorphism.

Concurrency

- It is a minor pillar of oops.
- Process of executing multiple task simultaneously is called as concurrency.
- Main goal of concurrency is utilize H/W resources efficiently.



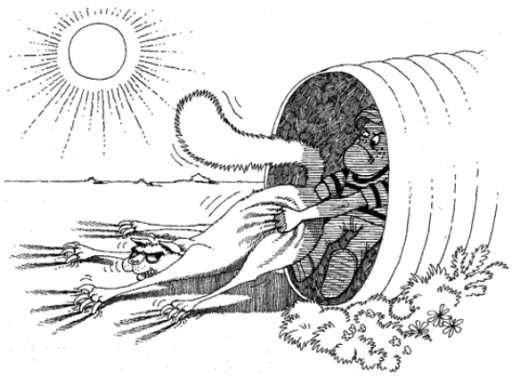
Concurrency allows different objects to act at the same time.

- With the help of thread, we can develop concurrent application in C++.

Persistance.

- It is a minor pillar of oops.

- Process of maintaing state of the object on secondary storage(File / HDD) is called as persitance.



Persistence saves the state and class of an object across time or space.

- For the persitance we can use Database/file.

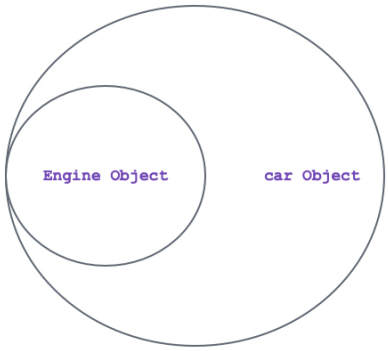
Association:

- Consider below examples:
    - Car has a engine
        - Dependent object: Car object
        - Dependency object: Engine object
    - Car has a music system
        - Dependent object: Car object
        - Dependency object: Music system object
    - Room has a wall
        - Dependent object: Room object
        - Dependency object: Wall object
    - Room has a chair
        - Dependent object: Room object
        - Dependency object: chair object

- If has-a relationship is exist between the objects then we should use association.

```
- Car has a engine
or
- Engine is a part of Car.
```

```
class Engine{
   //TODO
};
class Car{
   //car has a engine
   Engine e; //Association
};
int main( void ){
   Car c;
   return 0;
}
```



- If object is part of / component of another object then it is called as association.
- To implement association, we should declare object of a class as a data member inside another class.
- Example:
    - Employee has a join Date

```
class Date{
  private:
    int day;
    int month;
    int year;
  public:
    //TODO: constructor(s)
    //TODO: getters and setters
};
```

```
class Employee{
  private:
    string name;
    int empid;
    float salary;
    Date joinDate;  //Association
  public:
    //TODO: constructor(s)
```

```
    //TODO: getters and setters
};
```

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

Composition

- Consider below example:
  - Human has a heart
    - Dependent object: Human object
    - Dependency object: Heart object
- In case of association, if dependency object do not exist without dependent object then it is called as composition.

```
class Heart{
  //TODO
};
class Human{
  Heart hrt;
};
```

- It represents tight coupling.

Aggregation

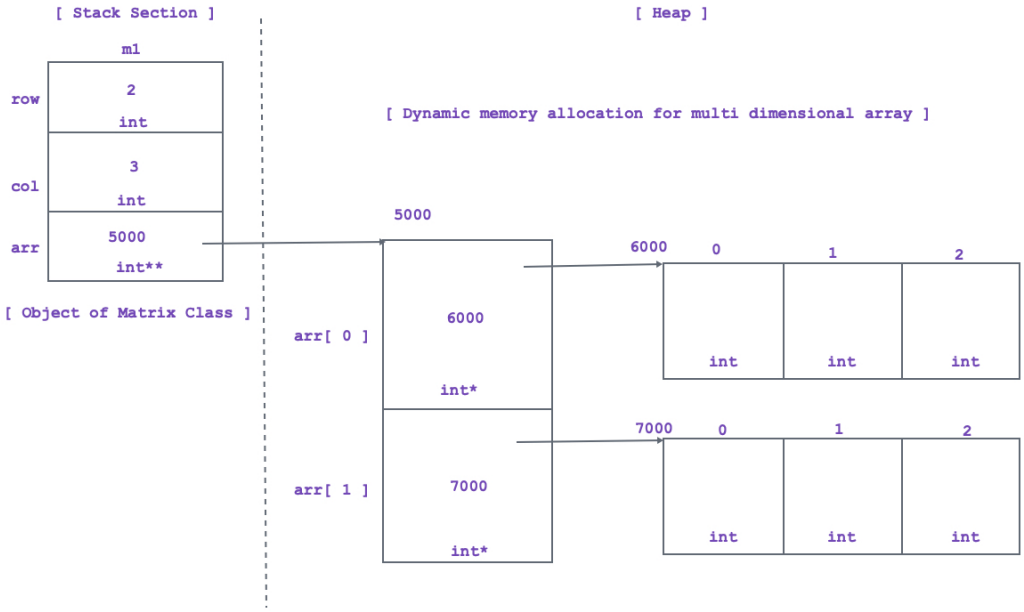
- Consider below example:
  - Department has a faculty
    - Dependent object: Department object
    - Dependency object: Faculty object
- In case of association, if dependency object exist without dependent object then it is called as aggregation.

```
class Faculty{
  //TODO
};
class Department{
  Faculty faculty;
};
```

- It represents loose coupling.

Day 12

- Reference: https://www.eskimo.com/~scs/cclass/notes/top.html
- size of std::string is 24 bytes.
- Consider Matrix class object



Inheritance

- Consider below examples:
    - Manager is a Employee
    - Book is a Product
    - Rectangle is a Shape
    - SavingAccount is a Account
    - Car is a Vehicle
- If is a relationship is exist between the types then we should use inheritance.
- Inheritance is also called as generalization.
- Consider below code:

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

```
};
// here public is mode of inheritance
```

- In C++, parent class is called as Base class and child class is called as Derived class.
- If we create object of derived class then all the non static data member declared in base class and derived class get space inside it. In other words, non static data member of the base class inherit into derived class.
- Using derived class, we can access static data member declared in base class. In other words, static data member of base class inherit into derived class.
- All the data members(static & non static of any access specifier) of base class inherit into derived class. But only non static data members get space inside object.
- Data members of drived class, do not inherit into base class hence size of object of base class depends on non static data members declared inside base class only.
- Size of object of derived class = size of all the non static data members declared in base class and derived class.
- Note: private/protected/public data members(static & non static) inherit into derived class.
- We can call, non static member function of base class on object of derived class. In other words, non static member function of base class inherit into derived class.
- We can call, static member function of base class on derived class name. In other words, static member function of base class inherit into derived class.
- Below functions, do not inherit into derived class:
    - constructor
    - destructor
    - copy constructor
    - assignment operator function
    - friend function
- Except above five functions, all the static and non static member functions of base class inherit into derived class.
- During inheritance, member functions of base class inherit into derived class. Hence using derived class object, we can call member function of base class as well as derived class.
- During inheritance, member functions of derived class do not inherit into base class. Hence using base class object, we can call member function of base class only.
- Nested class of base class inherit into derived class.
- Final Conclusion: Except constructor, destructor, copy constructor, assignment operator function and friend function all the members of base class inherit into derived class.
- If we create object of Base class then only base class constructor gets called.

- If we create object of Derived class, then first base class constructor gets called and the derived class constructor gets called. Destructor calling sequence is exactly opposite of constructor calling sequence.
- From any constructor of deived class, by default, base class's parameterless constructor gets called.
- Using constructor's base initializer list, we can call, any constructor of base class from constructor of derived class.

```
Employee( string name, int age, int empid, float salary ) : Person(
name, age ){
    cout << "Employee( string name, int age, int empid, float salary
)" << endl;
    this->empid = empid;
    this->salary = salary;
}
```

- How to read below statement:

```
class Employee : public Person
```

  - Class Person is inherited into class Employee.
  - Class Employee is derived from class Person.

- Example 1:

```
int num1 = 10;
int main( void ){
  int num1 = 20;
  cout << "Num1 : " << num1 << endl;  //20
  return 0;
}
```

  - In above code, local variable is hiding global variable. It is also called as shadowing

- Example 2:

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

```
    cout<<"Num1 : "<< num1 << endl; //20
  }
};
int main( void ){
  Test t;
  t.print( ); //20
  return 0;
}
```

- In above code, local variable is hiding data member. It is also called as shadowing

- Example 3:

```
class A{
public:
  int num1;
public:
  A( void ){
    this->num1 = 10;
  }
};
class B : public A{
public:
  int num1;
public:
  B( void ){
    this->num1 = 20;
  }
};
int main( void ){
  B b;
  cout << "Num1 : "<< num1 << endl; //20
  return 0;
}
```

- In above code, derived class data member is hiding base class data member. It is also called as shadowing

- According client's requirement, if implementation of base class member function is logically incomplete then we should override/redefine member function inside derived class.

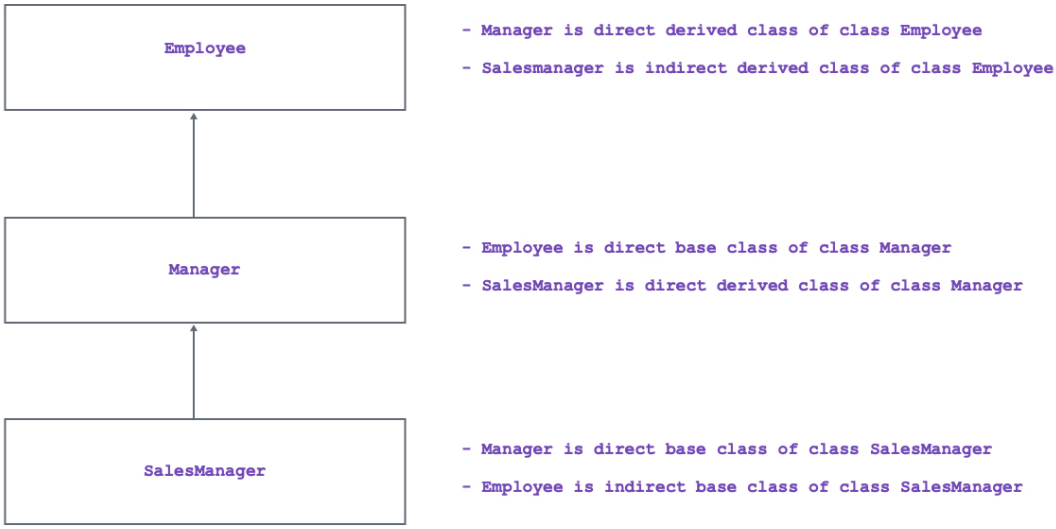
- If name of member function defined in base class & derived class is same and if we call such member function on object of derived class then preference will be given to derived class member function. In this case, derived class member function hides implmentation of base class member function. It is called as shadowing.

- In General, to access any member of base class inside member function of derived class, we should use class name and :: operator.

Applications of scope resolution operator:

- To define member function global
- To access members of namespace
- To access static members
- To access members of base class inside member function of derived class.
- According client's requirement, if implementation of exisiting class is logically incomplete / partially complete then to make it complete we should extend that class in other words we should create its derived class. It means we should use inheritance.
- Process of reusing members of parent class inside child class is called as inheritance.

Mode of Inheritance



- When we use private/protected/public keyword to control visibility of members of the class inside class, it is called access specifier. Default access specifier of class is private.
- When we use private/protected/public keyword to create derived class then it is called mode of inheritance.
- Example 1:

```
class Employee : public Person
```

  - In above statement, mode of inheritance is public.
- Example 2:

```
class Employee : Person
```

  - In above statement, mode of inheritance is private.
- In C++, default mode of inheritance is private.
- If has-a relationship is exisit between the type then either we should use association or private mode of inheritance.
- Car has a engine
- Example 1:

```
class Engine{
  //TODO
};
class Car{
  Engine e; //Association
};
```

- Example 2:

```
class Engine{
  //TODO
};
class Car : private Engine{
  //TODO
};
```

- If is-a relationship is exisit between the type then we should use public mode of inheritance.
- Example: Tape(CD / DVD / Cassete) is a Product.

```
class Product{
  //TODO
};
class Tape : public Product{
  //TODO
};
```

public mode of Inheritance					
Access Specifier	Same Class	Derived class	Indirect Derived class	Friend Function	Non Member Function
private	A	NA	NA	A	NA
protected	A	A	A	A	NA
public	A	A	A	A	A

private mode of Inheritance					
Access Specifier	Same Class	Derived class	Indirect Derived class	Friend Function	Non Member Function
private	A	NA	NA	A	NA
protected	A	A	NA	A	NA
public	A	A	NA	A	Accesible using Base class object Not Accessible using derived class object

protected mode of Inheritance					
Access Specifier	Same Class	Derived class	Indirect Derived class	Friend Function	Non Member Function
private	A	NA	NA	A	NA
protected	A	A	A	A	NA
public	A	A	A	A	Accesible using Base class object Not Accessible using derived class object

Types of inheritance

- Interface inheritance

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

    - During inheritance, if parent type and child type is interface then such type of inheritance is called as interface inheritance.
        - Single inheritance
        - Multiple inheritance
        - Hierarchical inheritace
        - Multilevel inheritance
- Implementation inheritance

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

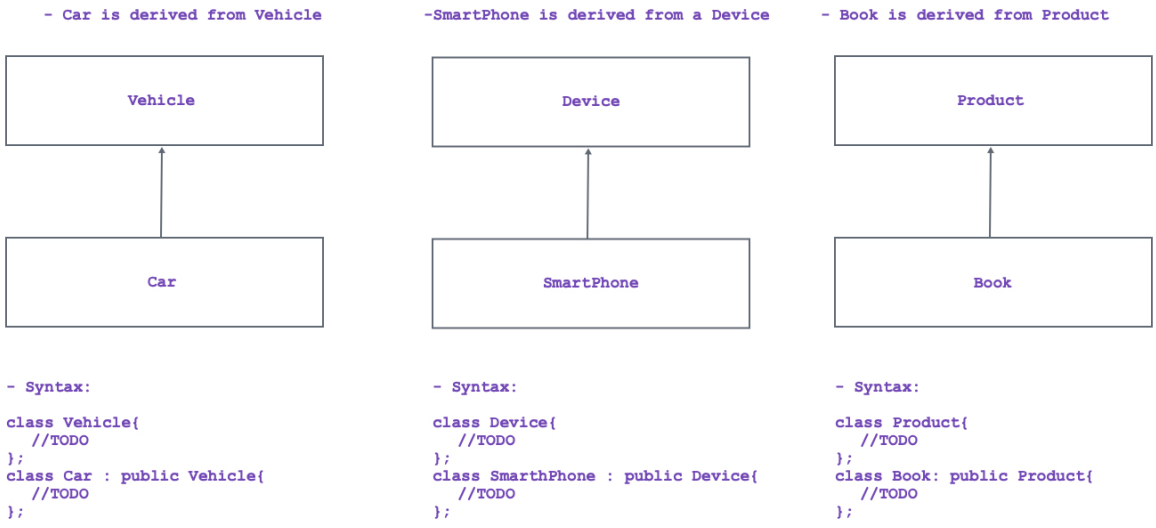
- During inheritance, if parent type and child type is class then such type of inheritance is called as implementation inheritance.
  - Single inheritance
  - Multiple inheritance
  - Hierarchical inheritace
  - Multilevel inheritance

Single inheritance

- Consider below example:
  - class B is derived from class A

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

- If single base class is having single derived class then such type of inheritance is called as single inheritance.

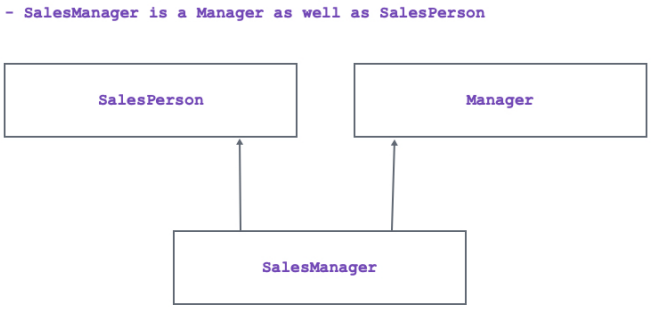


Multiple inheritance

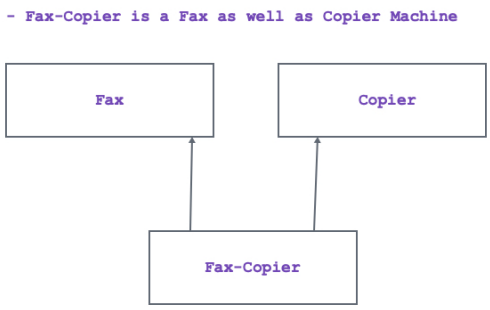
- Consider below example:
  - class C is derived from class A and class B

```
class A{
  //TODO
};
class B{

};
class C : public A, public B{
    //TODO
};
```

- If multiple base classes are having single derived class then such type of inheritance is called as multiple inheritance.




```
- Syntax:

class SalesPerson{
   //TODO
};
class Manager{
   //TODO
};
class SalesManager : public SalesPerson, public Manager{
   //TODO
};
```




```
- Syntax:

class Fax{
   //TODO
};
class Copier{
   //TODO
};
class FaxCopier : public Fax,public Copier{
   //TODO
};
```

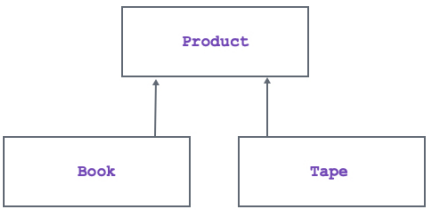
Hierarchical inheritace

- Consider below example:
  - class B and C are derived from class A

```
class A{
  //TODO
};
class B : public A{
  //TODO
};
class C : public A{
  //TODO
};
```

- If single base class is having multiple derived classes then such type of inheritance is called has hierarchical inheritance.

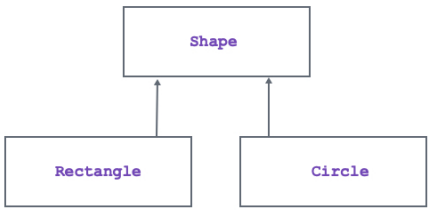
- Book is a product, Tape is a Product



- Syntax:

```
class Product{
   //TODO
};
class Book : public Product{
   //TODO
};
class Tape : public Product{
   //TODO
};
```

- Rectangle is a Shape, Circle is a Shape



- Syntax:

```
class Shape{
   //TODO
};
class Rectangle : public Shape{
   //TODO
};
class Circle : public Shape{
   //TODO
};
```

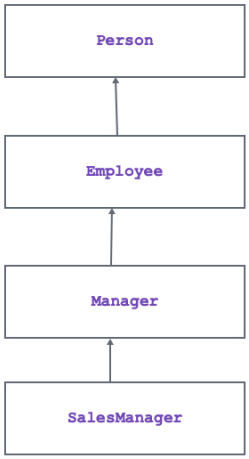
Multilevel inheritance

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

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

- If single inheritance is having multiple levels then such type of inheritance is called as multilevel inheritance.

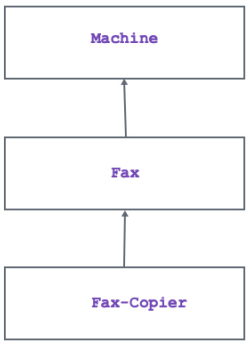
- Employee is Person, Manager is a Employee, SalesManager is a Manager



- Syntax:

```
class Person{
   //TODO
};
class Employee : public Person{
   //TODO
};
class Manager : public Employee{
   //TODO
};
class SalesManager : public Manager{
   //TODO
};
```

- Fax machine is machine, Fax-Copier machine is a Fax machine

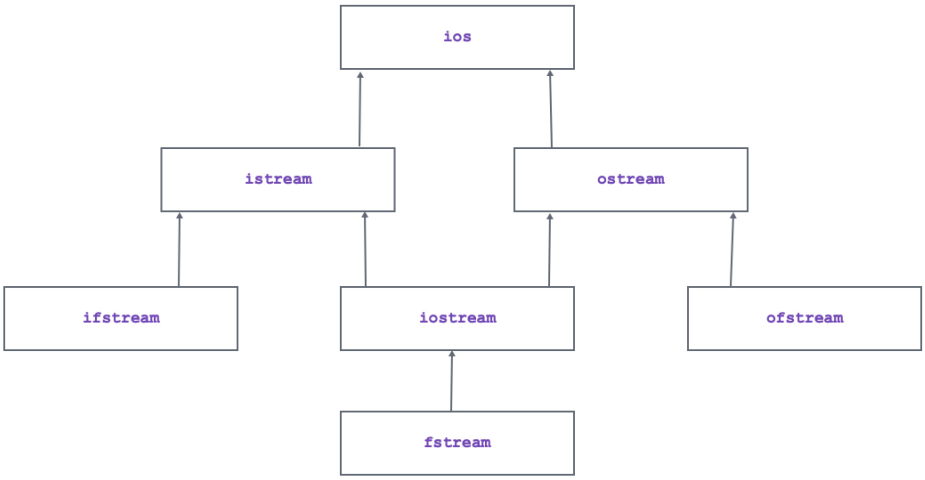


- Syntax:

```
class Machine{
   //TODO
};
class Fax : public Machine{
   //TODO
};
class FaxCopier : public Fax{
   //TODO
};
```

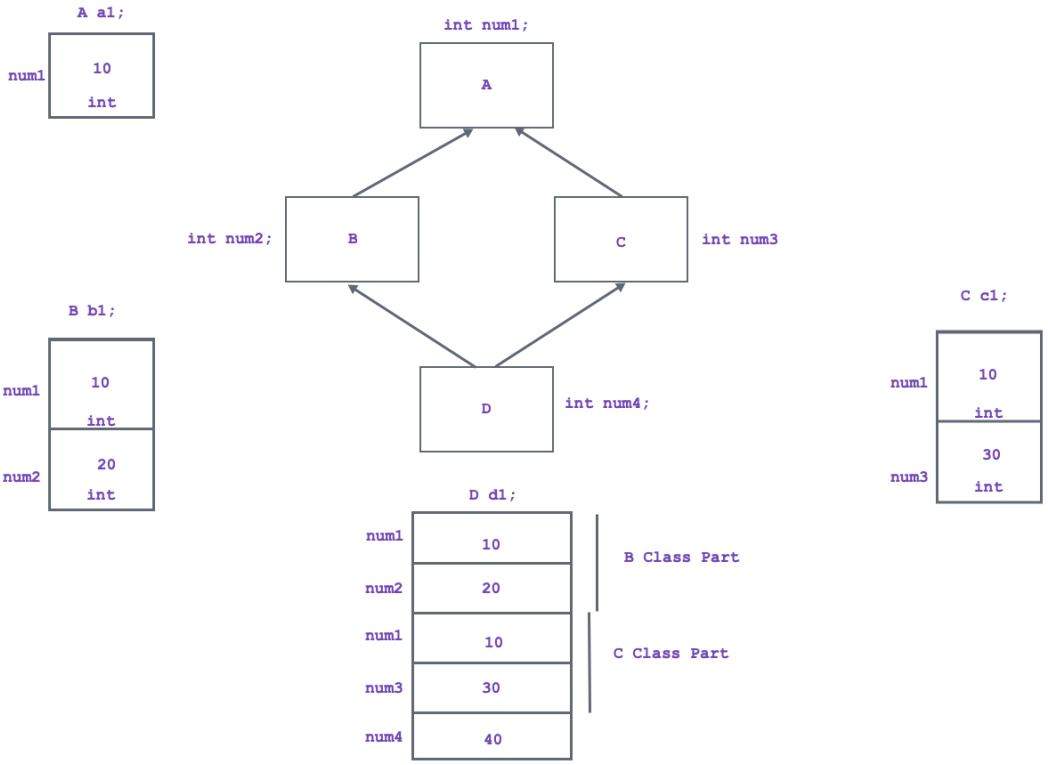
Hybrid inheritance

- If we combine any two / more than two types of inheritance then it is called as hybrid inheritance.

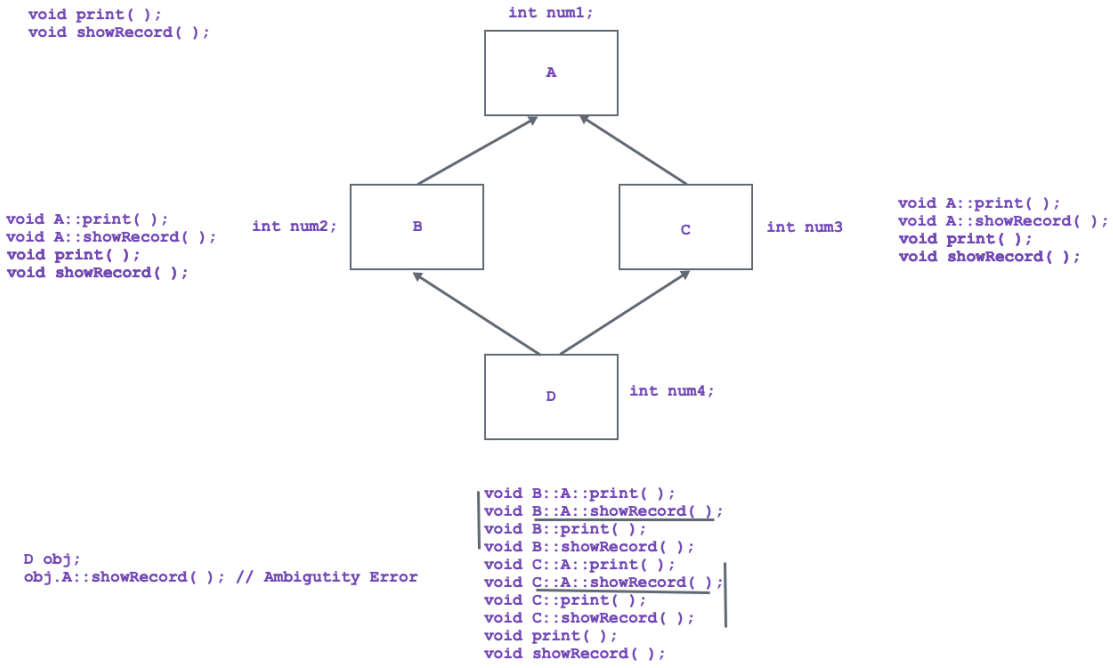


Diamond Problem

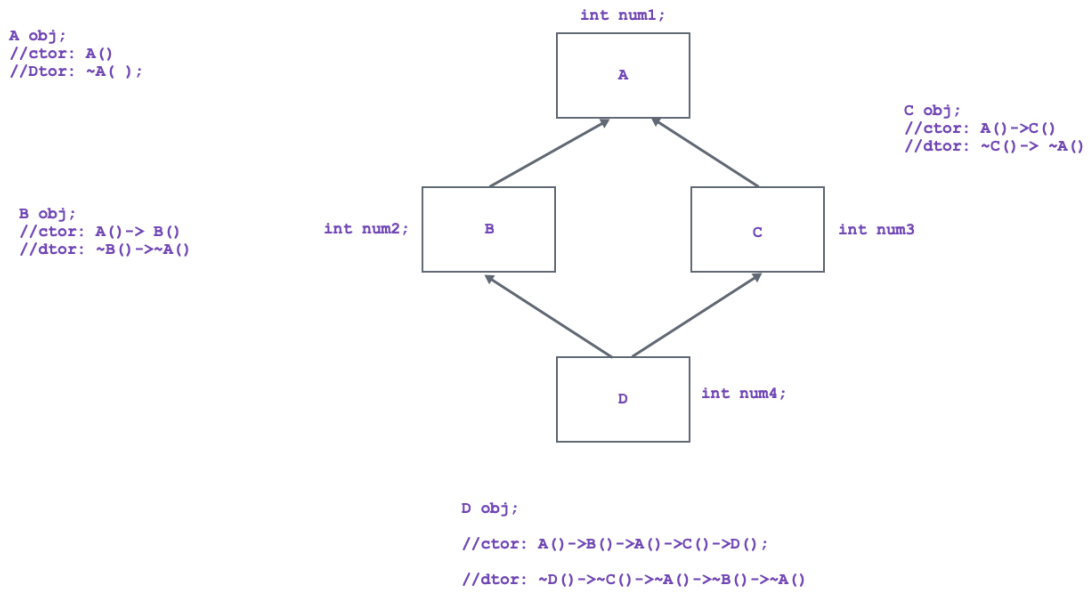
- In case of hybrid inheritance, if we create object of indirect derived class then data members of indirect base class gets inherited multiple times. Hence it increases size of object.



- In case of hybrid inheritance, member functions of indirect base class gets inherited multiple times. In this case, if we try to call member function of indirect base class on object of indirect derived class then compiler generate ambiguity error.



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



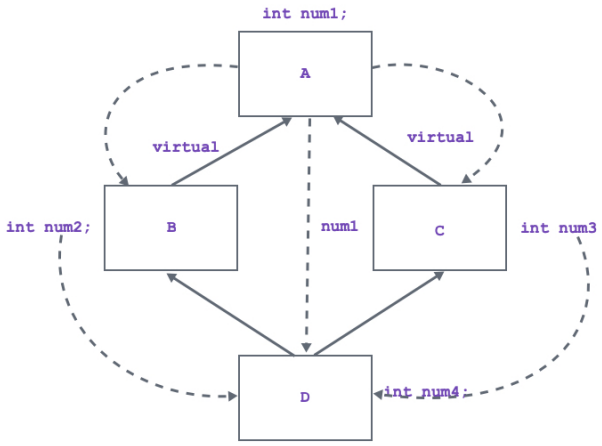
Diamond Problem solution

- virtual is a keyword in C++.
- To avoid diamond problem, we should declare base class virtual.

```
virtual class A{  //Not OK
  //TODO
};
```

- To declare base class virtual, we should use following syntax:

```
class B : virtual public A{ //Virtual inheritance
  //TODO
}
class C : virtual public A{ //Virtual inheritance
  //TODO
}
```



Runtime Polymorphism

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

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

```
public:
  B( void ){
    this->num3 = 30;
  }
  B( int num1, int num2, int num3 ) : A( num1, num2 ){
    this->num3 = num3;
  }
  void printRecord( void ){
    A::printRecord( );
    cout << "Num3 :   " << this->num3 << endl;
  }
  void displayRecord( void ){
    A::showRecord( );
    cout << "Num3 :   " << this->num3 << endl;
  }
};
```

- During inheritance, members(data members + member functions + nested class) of derived class do not inherit into base class. Hence using base class object, we can call/access members of base class only.

```
int main( void ){
  A a1;
  //a1.showRecord( );       //OK: A::showRecord( );
  //a1.printRecord( );  //OK: A::printRecord( );
  //a1.displayRecord( );    //Not OK:  error: no member named
'displayRecord' in 'A'
  //a1.B::printRecord( );   //Not OK:  error: 'B::printRecord' is not
a member of class 'A'
  return 0;
}
```

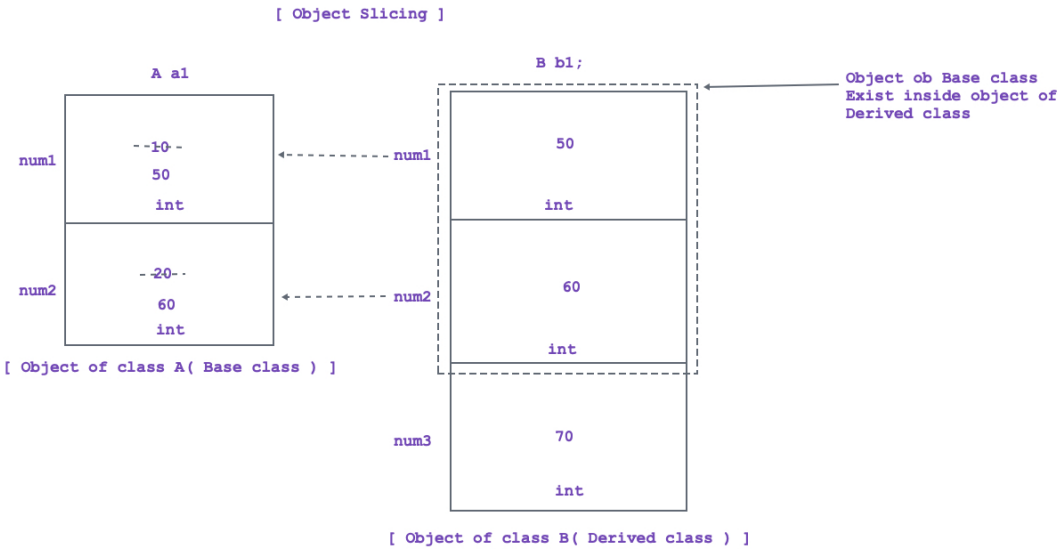
- During inheritance, members of base class inherit into derived class. Hence using derived class object, we can access members of base class as well as derived class.

```
int main( void ){
  B b;
  //b.showRecord( );      //OK: A::showRecord( );
  //b.printRecord( );     //OK: B::printRecord( );
  //b.A::printRecord( );  //OK: A::printRecord( );
  b.displayRecord( );     //OK: B::displayRecord( );
  return 0;
}
```

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

- Since derived class object can be considered as base class object, we can assign derived class object to the Base class Object.

```
int main( void ){
  B b1( 50, 60, 70 );
  A a1;
  a1 = b1;  //OK: Object Slicing
  a1.printRecord( );    //OK: A::printRecord( );    //50, 60
  return 0;
}
```



- If we assign, Derived class object to the Base class Object then compiler copy state of Base class portion from derived class object into base class object. This process is called as Object slicing.
- During inheritance, members of derived class do not inherit into base class. Hence we can not consider object of base class as a object of derived class.
- Since base class object can not be considered as derived class object, we can not assign base class object to the derived class object.

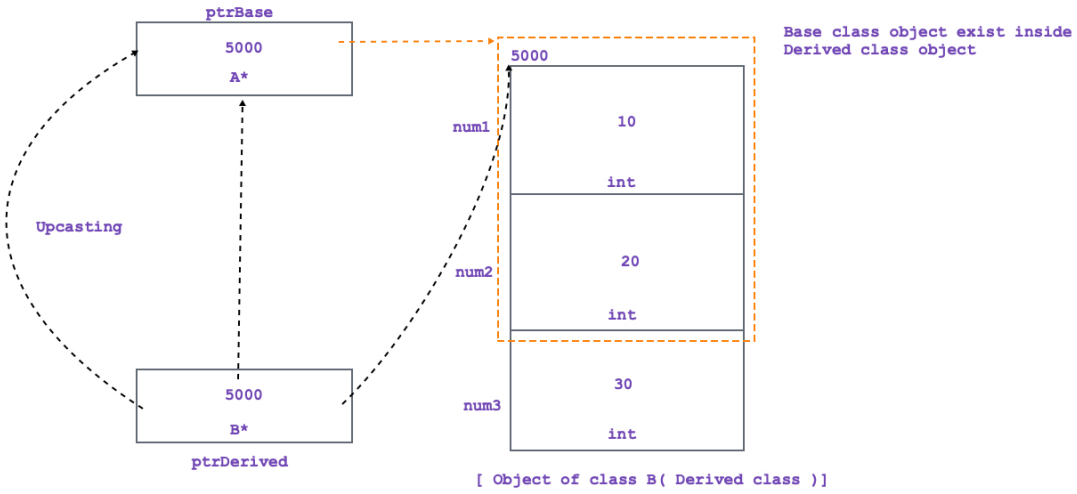
```
int main( void ){
  A a1( 50, 60 );
  B b1;
  //b1 = a1;    //Not OK:
  b1.printRecord( );
  return 0;
}
```

- Process of converting pointer of Derived class into pointer of Base class is called as upcasting.

```
int main( void ){
  B *ptrDerived = new B( );
  ptrDerived->printRecord( );   //OK: B::printRecord( );
  //A *ptrBase = ( A* )ptrDerived;  //Upcasting: OK
  A *ptrBase = ptrDerived;  //Upcasting: OK
  ptrBase->printRecord( );  //OK: A::printRecord( );
  delete ptrDerived;
  return 0;
}
```

- In case of upcasting, explicit typecasting is optional.
- We can store address of derived class object into pointer of base class. It is also called as upcasting.

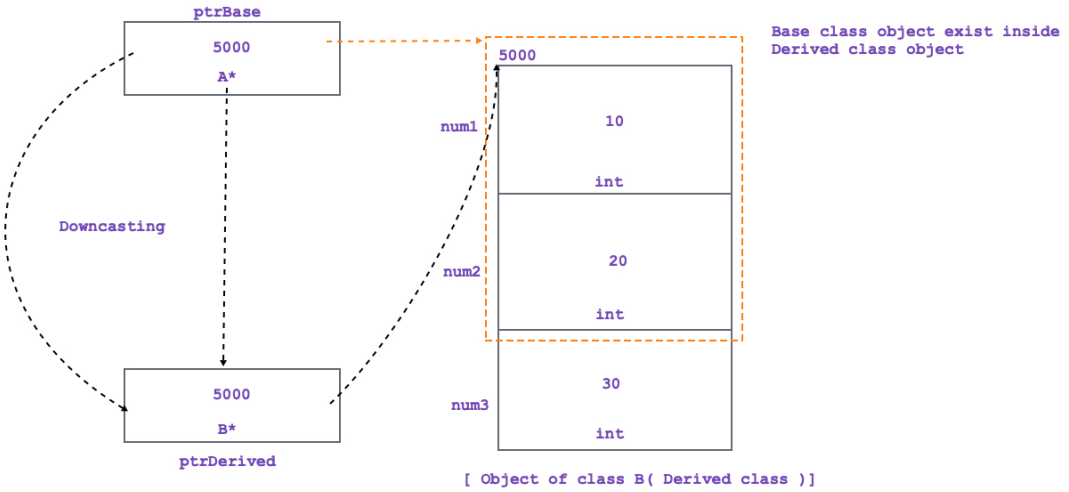
```
int main( void ){
  A *ptrBase = new B( );    //Upcasting: OK
  ptrBase->printRecord( );
  delete ptrBase;
  return 0;
}
```



- process of converting pointer of base class into pointer of derived class is called as downcasting.

```
int main( void ){
  A *ptrBase = new B( );    //Upcasting: OK
  ptrBase->printRecord( );  //A::printRecord( );
  B* ptrDerived = ( B* )ptrBase;    //OK: Downcasting
  ptrDerived->printRecord( );
```

```
    delete ptrBase;
    return 0;
}
```



Virtual function

- In case of upcasting, if we want to call member funtion, depending on type of object rather than type of pointer then we should declare member function in base class virtual.
- If class contains at least one virtual function then such class is called as polymorphic class.
- If signature of base class member function and derived class member function is same and if function in base class is virtual then derived class member function will be considered as virtual.
- Process of redefining virtual member function of base, class inside derived class, with same signature is called as function overriding and virtual function redefined in derived class is called as overrided function.
- For function overriding:
    - Function must be exist in base class and derived class
    - Signature of functions (including return type) must be same.
    - Function in base class must be virtual
- Definition:
    - In case of upcasting, A member function, which gets called depending on type of object rather that type of pointer is called as virtual funtion.
    - In case of upcasting, A member function of derived class which is deisigned to call using pointer/reference of base class is called as virtual function.
- It means that virtual functions are not designed to call on object / class rather it is designed to call on base class pointer or base class reference.

Can we declare static member function virtual?

- Virtual member function is designed to call on base class pointer / reference.
- Static member function is designed to call on class name.
- Since static member function is not designed to call on base class pointer / referece, we can not declare static member function virtual.
- Since we can not declare static member function virtual, we can not override it inside derived class.

What is runtime polymorphism:

- Process of calling member function derived class on pointer / reference of base class is called as runtime polymorphism.

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

Day 13

Early Binding and Late Binding

- If call to the function gets resolved at compile time then it is called as early binding. In other words, if binding between function and object gets resolved at compile time then it is called as early binding.
- If call to the function gets resolved at runtime then it is called as late binding. In other words, if binding between function and object gets resolved at run time then it is called as late binding.
- If we call virtual or non virtual function on object then it is considered as early binding. This call always gets resolved at compile time.
- If we call non virtual function on pointer/reference then it is considered as early binding. This call always gets resolved at compile time.

- If we call virtual function on pointer/reference then it is considered as late binding. This call always gets resolved at run time. Consider below code:

```
class A{
private:
  int num1;
  int num2;
public:
  A( void ){
    this->num1 = 10;
    this->num2 = 20;
  }
  virtual void f1( void ){
    cout << "A::f1" << endl;
  }
  virtual void f2( void ){
    cout << "A::f2" << endl;
  }
  virtual void f3( void ){
    cout << "A::f3" << endl;
  }
  void f4( void ){
    cout << "A::f4" << endl;
  }
  void f5( void ){
    cout << "A::f5" << endl;
  }
};
```

```
class B : public A{
private:
  int num3;
public:
  B( void ){
    this->num3 = 30;
  }
  virtual void f1( void ){
    cout << "B::f1" << endl;
  }
  void f2( void ){
    cout << "B::f2" << endl;
  }
  void f4( void ){
    cout << "B::f4" << endl;
  }
  virtual void f5( void ){
    cout << "B::f5" << endl;
  }
  virtual void f6( void ){
    cout << "B::f6" << endl;
```

```
  }
};
```

```
int main( void ){
  A a;  //OK
  a.f1( );  //OK: A::f1 -> Early Binding
  a.f2( );  //OK: A::f2 -> Early Binding
  a.f3( );  //OK: A::f3 -> Early Binding
  a.f4( );  //OK: A::f4 -> Early Binding
  a.f5( );  //OK: A::f5 -> Early Binding
  a.f6( );  //Not OK: f6 is not a member of class A
  return 0;
}
```

- If we call any member function on object then it is considered as early binding

```
int main( void ){
  A *ptr = new A( );  //OK
  ptr->f1( );  //OK: A::f1 -> Late Binding
  ptr->f2( );  //OK: A::f2 -> Late Binding
  ptr->f3( );  //OK: A::f3 -> Late Binding
  ptr->f4( );  //OK: A::f4 -> Early Binding
  ptr->f5( );  //OK: A::f5 -> Early Binding
  ptr->f6( );  //Not OK: f6 is not a member of class A
  return 0;
}
```

- If we call virtual function on pointer then it is considered as late binding.
- If we call non virtual function on pointer then it is considered as early binding.

```
int main( void ){
  A *ptr = new B( );  //OK: Upcasting
  ptr->f1( );  //OK: B::f1 -> Late Binding
  ptr->f2( );  //OK: B::f2 -> Late Binding
  ptr->f3( );  //OK: A::f3 -> Late Binding
  ptr->f4( );  //OK: A::f4 -> Early Binding
  ptr->f5( );  //OK: A::f5 -> Early Binding
  ptr->f6( );  //Not OK: f6 is not a member of class A
  return 0;
}
```

```
int main( void ){
  B *ptr = new B( );  //OK: Upcasting
  ptr->f1( );  //OK: B::f1 -> Late Binding
  ptr->f2( );  //OK: B::f2 -> Late Binding
```

```
  ptr->f3( );  //OK: A::f3 -> Late Binding
  ptr->f4( );  //OK: B::f4 -> Early Binding
  ptr->f5( );  //OK: B::f5 -> Late Binding
  ptr->f6( );  //OK: B::f6 -> Late Binding
  return 0;
}
```

```
int main( void ){
  B *ptr = new A( );  //NOT OK
  ptr->f1( );
  ptr->f2( );
  ptr->f3( );
  ptr->f4( );
  ptr->f5( );
  ptr->f6( );
  return 0;
}
```

- Members of Base class inherit into Derived class. Hence we can consider Derived class object as Base class object.
- Also Base class pointer can contain address of derived class object.
- Members of Derived class do not inherit into Base class. Hence we can not consider Base class object as Derived class object.
- Also Derived class pointer can not contain address of Base class object.

```
int main( void ){
  B b;  //OK
  b.f1( );  //OK: B::f1 -> Early Binding
  b.f2( );  //OK: B::f2 -> Early Binding
  b.f3( );  //OK: A::f3 -> Early Binding
  b.f4( );  //OK: B::f4 -> Early Binding
  b.f5( );  //OK: B::f5 -> Early Binding
  b.f6( );  //OK: B::f6 -> Early Binding
  return 0;
}
```

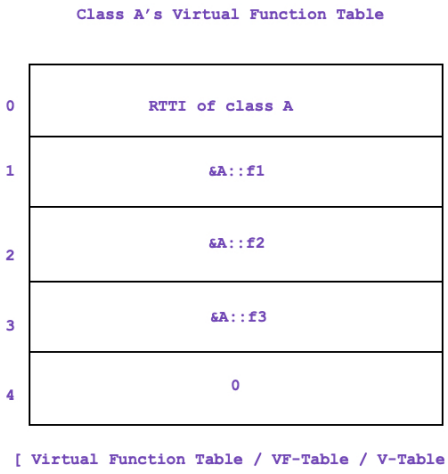
- If we want to decide early binding / late binding then we can use below algorithm.

```
//Check the Data type of caller i.e. Base class or derived class
if( function is not exist in caller data type ){
    Compiler error: Function is not a member of caller type.
}else{ //Function is exist in caller data type
  //Check whether caller is object, pointer/reference
  if( caller is object ){
    Early Binding: Member function exist / inherited function of
object type will call.
```

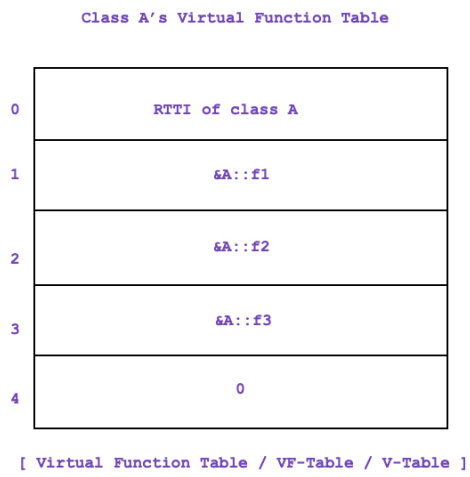
```
  }else{//Caller is pointer
    //Check whether function is virtual / non virtual
    if( function is non virtual ){//Call non virtual function on
pointer/reference
      Early Binding: Member function exist / inherited function of
pointer type will call.
    }else{//Call virtual function on pointer/reference
      Late Binding: Member function exist / inherited function of
object type will call.
    }
  }
}
```

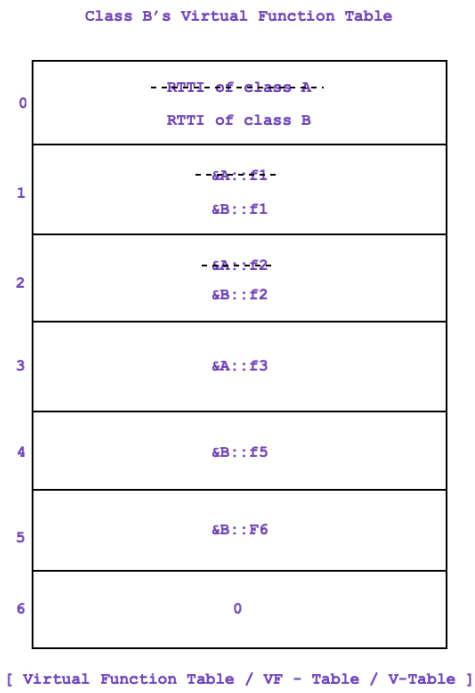
Virtual Function Table and Virtual Function Pointer.

- In case of upcasting, using base class pointer, if we want to call member function of derived class then we should declare function in base class virtual.
- If class contains at least one virtual function then compiler implicitly generate one table to store address of that virtual function(s). Such table(which can be array/strucrure depending on compiler vendor) is called as virtual function table / vf-table/v-table.
- In short, a table which contains address of virtual function is called as v-table.

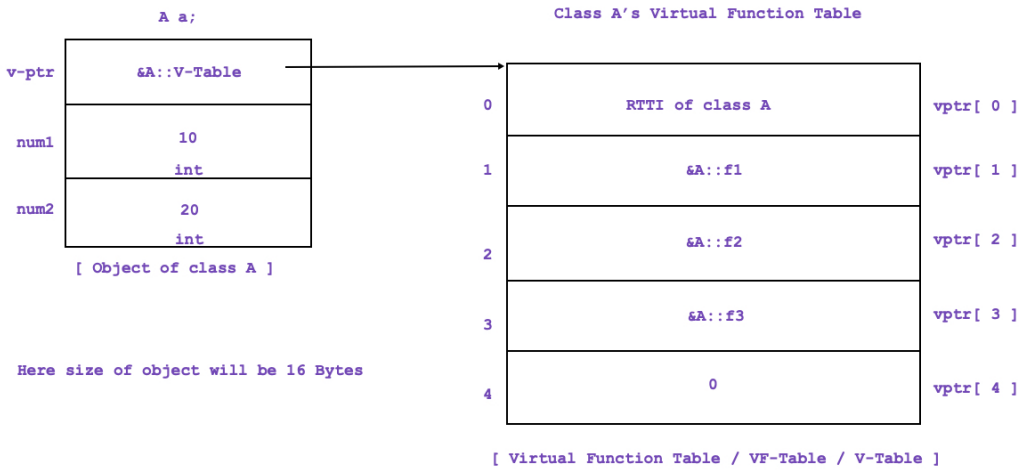


- At the time of creation of V-Table for derived class, compiler simply copy Base class V-Table and make necessary changes.

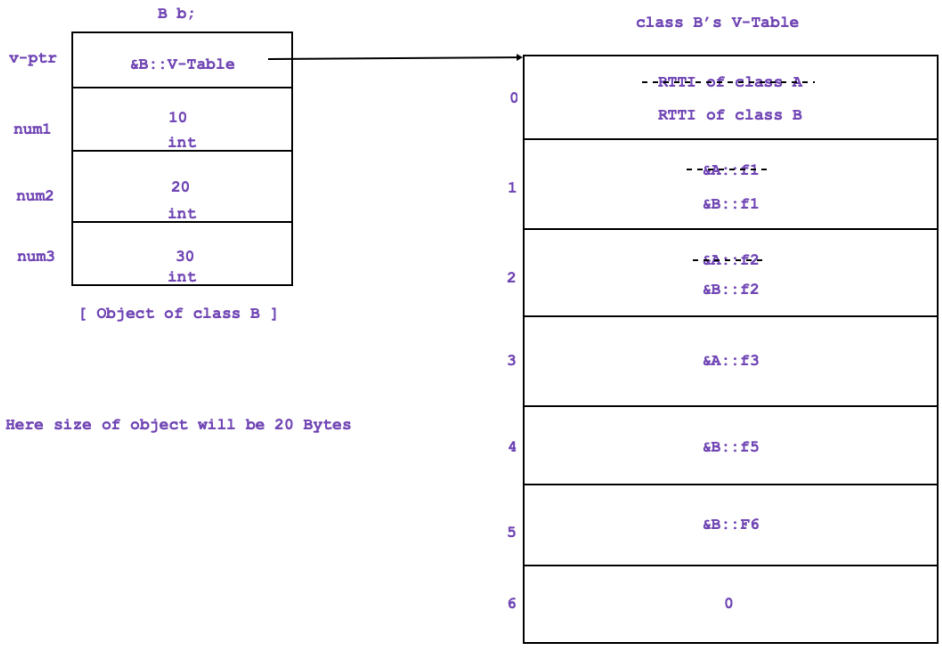




- Compiler generate v-table per class. It gets generated at compile time.
- To store address of virtual function table, compiler implicitly declare pointer as a data meber inside class. Such pointer is called as virtual function pointer/vf-pointer/v-ptr.
- In short, a pointer which contain address of virtual function table is called as v-ptr.
- Consider V-Table and V-Ptr for the class A.



- Consider V-Table and V-Ptr for the class B.



- Compiler generates V-Table and V-Ptr at compile time.
- New definition of size of object:
    - sum of all the non static data members declared in base and derived class + (2/4/8) bytes depending on the compiler.
- Address of V-Table come into V-Ptr inside constructor. It means that V-ptr gets initialized after calling constructor.

Can we declare constructor virtual? Why?

- In C++, we can not declare constructor virtual
- Reason 1:
    - Virtual functions are designed to call on base class pointer/reference only.
    - In C++, We can not call constructor on object, pointer or reference explicitly.
    - Since constructor is not designed to call on pointer or reference explicitly, we can not declare constructor virtual.
- Reason 2:
    - To call any virtual function, compiler need to access value of vptr then it can do indexing into V-Table.
    - But V-Ptr gets initialized after calling constructor. Hence we can not declare constructor virtual.

Can we declare destructor virtual? Why?

- We can not declare constructor virtual but we can declare destructor virtual.
- In Case of upcasting, constructor of Base class and Derived class gets call properly. But when we use delete operator on Base class pointer then only Base class destructor gets called.
- To get call to destructor of Derived class first, we need to declare destructor in Base class virtual.

- Consider Below Code:

```
class Base{
private:
  int *ptr;
public:
  Base( void ){
    this->ptr = new int[ 3 ];
  }
  virtual ~Base( void ){
    delete[] this->ptr;
  }
};
class Derived : public Base{
private:
  int *ptr;
public:
  Derived( void ){
    this->ptr = new int[ 5 ];
  }
  ~Derived( void ){
    delete[] this->ptr;
  }
};
int main( void ){
  Base *ptr = new Derived( );
  //TODO
  delete ptr;
  return 0;
}
```

What is the difference between Function Overloading and Function Overriding

- We achieve compile time polymorphism using function overloading and run time polymorphism using function overriding.
- In case of function overloading functions must be exist in same scope but in case function overriding functions must be exist in base class and derived class.
- In case of function overloading signature of functions must be different but In case of function overriding signature of function must be same.
- Return type is not considered in function overloading but return type is considered in function overriding.
- Function overloading is based on mangled name whereas function overriding is based on V-Table and V-ptr
- Function overloading do not require any keyword but for function overriding, function in base class must be virtual.

Runtime Type Polymorphism

- How to get size of object in C++

```
int main( void ){
  Complex c1;
  size_t size = sizeof( c1 );
  //TODO
  return 0;
}
```

- size_t is a alias for unsigned long.
- How to get type of object in C++.

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

- typeid is a operator which return reference of constant object of type_info class.
- type_info class is declared in std namespace and it is available int typeinfo header file.
- Consider declaration of type_info class

```
namespace std {
  class type_info{
  public:
      const char* name() const noexcept;
      bool operator==(const type_info& rhs) const noexcept;
      bool operator!=(const type_info& rhs) const noexcept;
      virtual ~type_info();
  private:
      type_info(const type_info& rhs);
      type_info& operator=(const type_info& rhs);
  };
}
```

- Process of getting type of object at runtime is called as Runtime Type Identification / Information.
- In case of upcasting, if we want to find out true type of object then we should use RTTI.
- Consider below code:

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

class Base{
  int num1;
public:
  Base( void ){
    this->num1 = 10;
  }
  void print( void ){
    cout << "Num1 :   " << this->num1 << endl;
  }
};
class Derived : public Base{
  int num2;
public:
  Derived( void ){
    this->num2 = 20;
  }
  void print( void ){
    Base::print( );
    cout << "Num2 :   " << this->num2 << endl;
  }
};
int main( void ){
  Base *ptrBase = new Derived( ); //Upcasting
  cout << typeid( ptrBase ).name( ) << endl;  //P4Base
  cout << typeid( *ptrBase ).name( ) << endl; //4Base
  return 0;
}
int main4( void ){
  Derived *ptrDerived = new Derived( );
  cout << typeid( ptrDerived ).name( ) << endl;   //P7Derived
  cout << typeid( *ptrDerived ).name( ) << endl;  //7Derived
  return 0;
}
int main3( void ){
  Derived derived;
  cout << typeid( derived ).name( ) << endl;  //7Derived
  return 0;
}

int main2( void ){
  Base *ptrBase = new Base( );
  cout << typeid( ptrBase ).name( ) << endl;  //P4Base
  cout << typeid( *ptrBase ).name( ) << endl; //4Base
  return 0;
}
int main1( void ){
  Base base;
  cout << typeid( base ).name( ) << endl; //4Base
  return 0;
```

```
  }
```

- In case of upcasting, using RTTI, to get true type of object, Base class must be polymorphic.

```
class Base{
  int num1;
public:
  Base( void ){
    this->num1 = 10;
  }
  virtual void print( void ){
    cout << "Num1   :   " << this->num1 << endl;
  }
};
class Derived : public Base{
  int num2;
public:
  Derived( void ){
    this->num2 = 20;
  }
  void print( void ){
    Base::print( );
    cout << "Num2   :   " << this->num2 << endl;
  }
};
int main( void ){
  Base *ptrBase = new Derived( );   //Upcasting
  cout << typeid( ptrBase ).name( ) << endl;    //P4Base
  cout << typeid( *ptrBase ).name( ) << endl;   //7Derived
  return 0;
}
```

- Using Null pointer, if we try to find out true type of object then typeid operator throws bad_typeid exception.

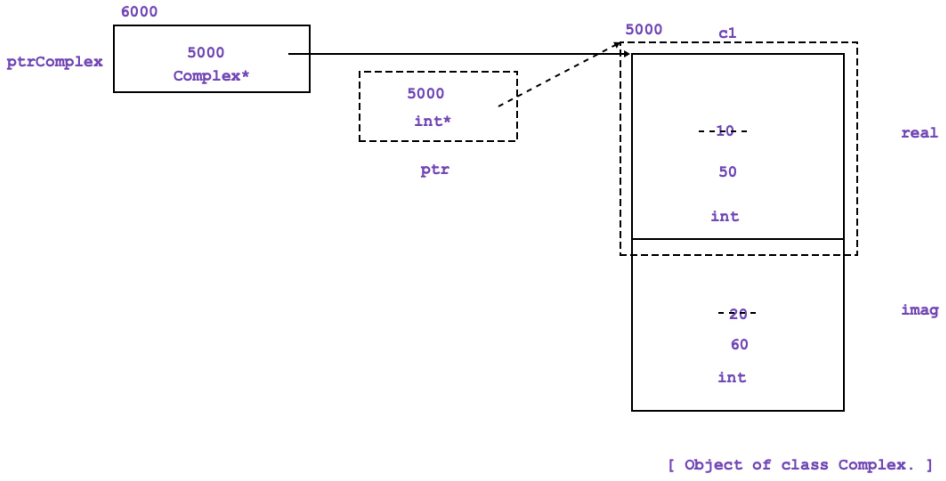
```
int main( void ){
  try{
    Base *ptrBase = NULL;   //Upcasting
    cout << typeid( ptrBase ).name( ) << endl;  //P4Base
    cout << typeid( *ptrBase ).name( ) << endl; //7Derived
  }catch( bad_typeid &ex ){
    cout << ex.what() << endl;
  }
  return 0;
}
```

Advanced Type Casting Operators

- static_cast
- dynamic_cast
- const_cast
- reinterpret_cast

reinterpret_cast operator

- We can access private data members of the class inside non member function using:
    - Member functions e.g. getter and setter
    - Friend function
    - Pointer
- If we want to convert pointer of any type into pointer of any other type then we should use reinterpret_cast operator




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

class Complex{
private:
  int real;
  int imag;
public:
  Complex( void ){
    this->real = 10;
    this->imag = 20;
  }
  friend ostream& operator<<( ostream &cout, Complex &other ){
    cout << "Real Number    :   " << other.real << endl;
```

```
    cout << "Imag Number    :   " << other.imag << endl;
    return cout;
  }
};
int main( void ){
  Complex c1;
  cout << c1 << endl;
  //int *ptr = (int*)(&c1); //C-Style
  int *ptr = reinterpret_cast<int*>( &c1 ); //C++ Style
  *ptr = 50;
  ptr = ptr + 1;
  *ptr = 60;
  cout << c1 << endl;
  return 0;
}
```

const_cast operator

- If we want to 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.

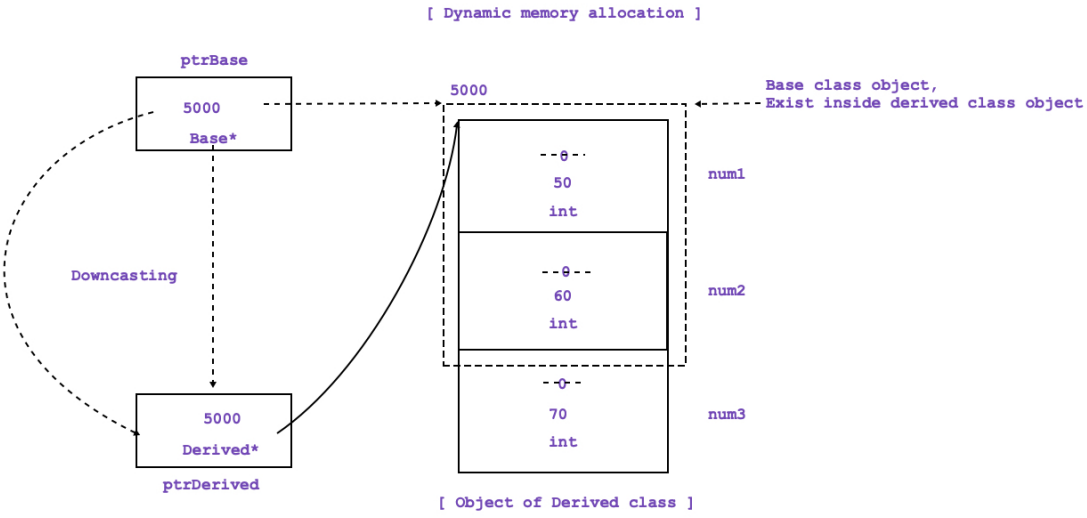
```
#include<iostream>
using namespace std;

class Test{
  int number;
public:
  //Test *const this
  Test( void ){
    this->number = 10;
  }
  //Test *const this
  void showRecord( void ){
    cout << "Number   :   "<<this->number << endl;
  }

  //const Test *const this
  void displayRecord( void ) const{
    //Test *const ptr = ( Test *const)this;   //C-Style
    Test *const ptr = const_cast<Test *const>( this );    //C++
Style
    ptr->showRecord( );
  }
};

int main( void ){
  const Test t;
  t.displayRecord( );
  return 0;
}
```

static_cast operator

- If we want to do type conversion between compatible types then we should use static_cast operator.

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

- In case of non polymorphic type, if we want to do downcasting the we should use static_cast operator.
- static_cast operator do not check whether type consversion is valid or invalid. it only checks inheritance relationship at compile time.



dynamic_cast operator

- In case of polymorphic type, if we want to do downcasting the we should use dynamic_cast operator.
- If we want to check whether, type conversion is valid or invalid then we should use dynamic_cast operator.
- dynamic_cast operator checks inheritance relationship at runtime.
- In case of pointer, if dynamic_cast operator fail to do conversion then it returns NULL.
- In case of reference, if dynamic_cast operator fail to do conversion then it throws bad_cast excetion.

```
int main( void ){
  Base *ptrBase = new Derived( );   //OK: Upcasting
  ptrBase->setNum1( 10 );   //OK
  ptrBase->setNum2( 20 );   //OK
  Derived *ptrDerived = dynamic_cast< Derived*>( ptrBase );
//Downcasting: C++ -Style
  if( ptrDerived != NULL ){
    ptrDerived->setNum3( 30 );
    ptrDerived->Base::print( );
    ptrDerived->Derived::print( );
    delete ptrBase;
  }
  return 0;
}
```

Pure Virtual Function and Abstract Class

- According to client's requirement, If implementation of base class member function is logically 100% complete then we should not declare base class member function 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.
- If we equate virtual function to 0 then such virtual function is called as pure virtual function.
- Consider below code:

```
class Shape{
protected:
  float area;
public:
  Shape( void ){
    this->area = 0;
  }
  virtual void acceptRecord( void ) = 0;  //Pure Virtual Function
  virtual void calculateArea( void ) = 0; //Pure Virtual Function
  void printRecord( void ){
    cout << "Area   :   " << this->area << endl;
  }
  virtual ~Shape( void ){ }
};
```

- We can not provide body to the pure virtual function. Hence it is also called as abstract method.
- If class contains at least one pure virtual function then such class is called as abstract class.
- If class contains all pure virtual function then such class is called as pure abstract class / interface.
- We can not create object of abstract class and interface but we can create pointer / reference of it.
- It is mandatory to override pure virtual function in derived class otherwise derived class can be considered as abstract.

- If we create object of derived class then constructor and destructor of base class gets called. Hence abstract class can contain constructor as well as destructor.

Virtual Base Class Pointer