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.

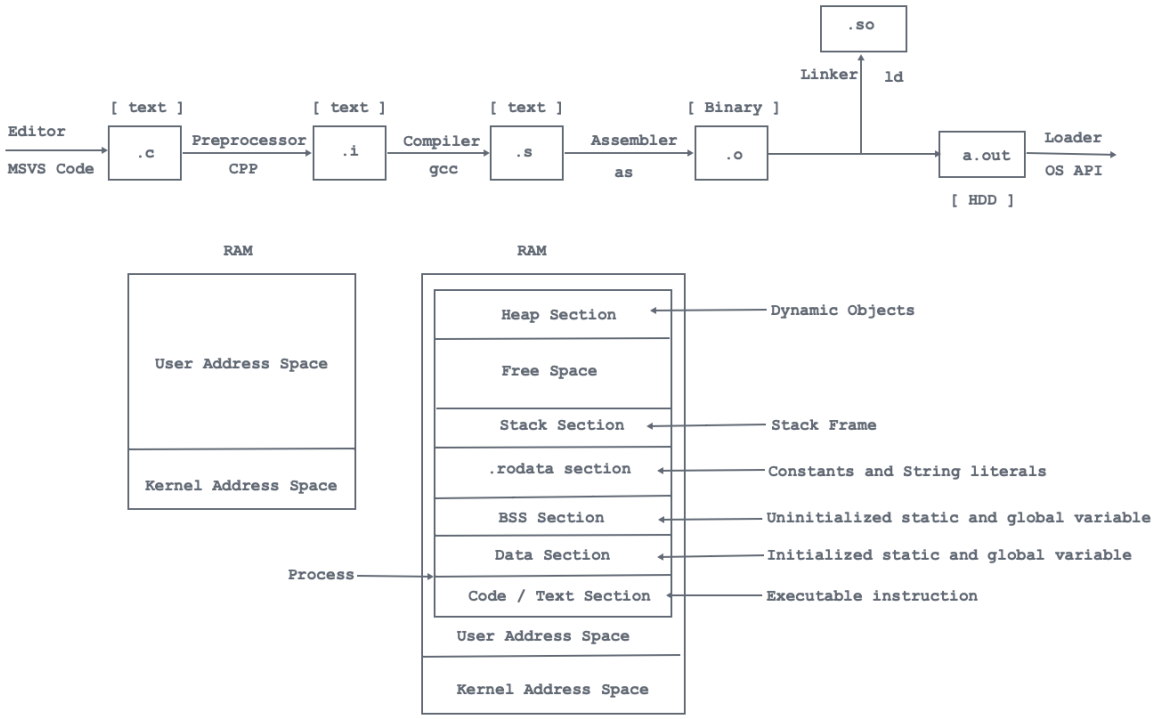
Software Development Kit

- 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:

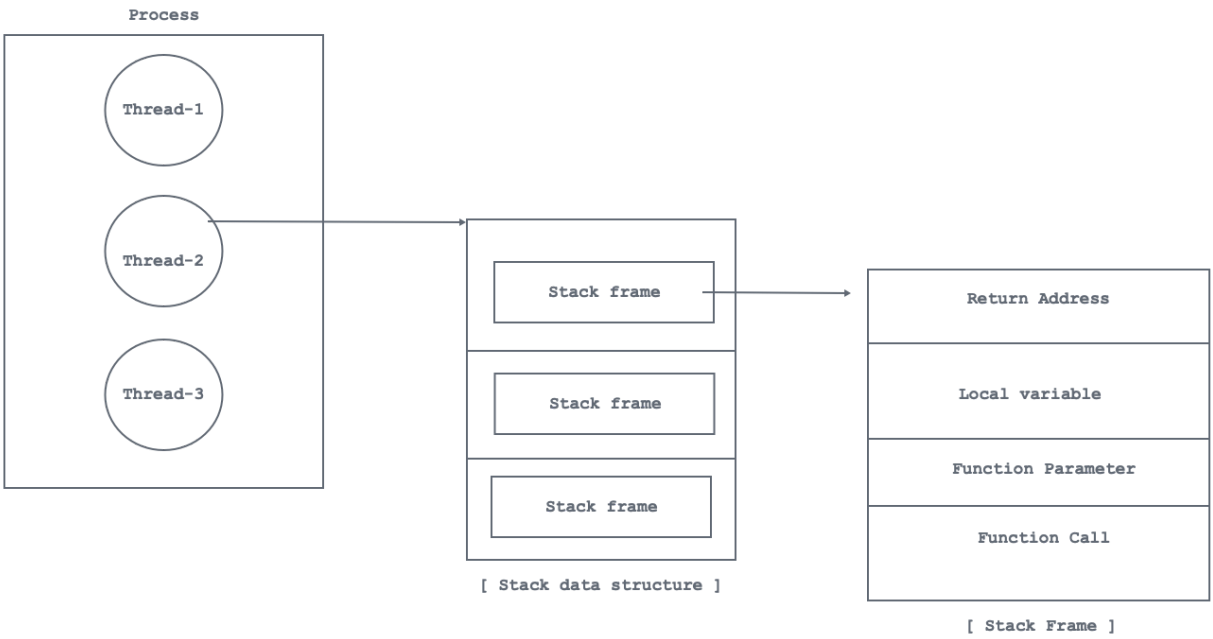
```
int number = 10;  //Initialization
number = 20;  //Assignment
```

- 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
```

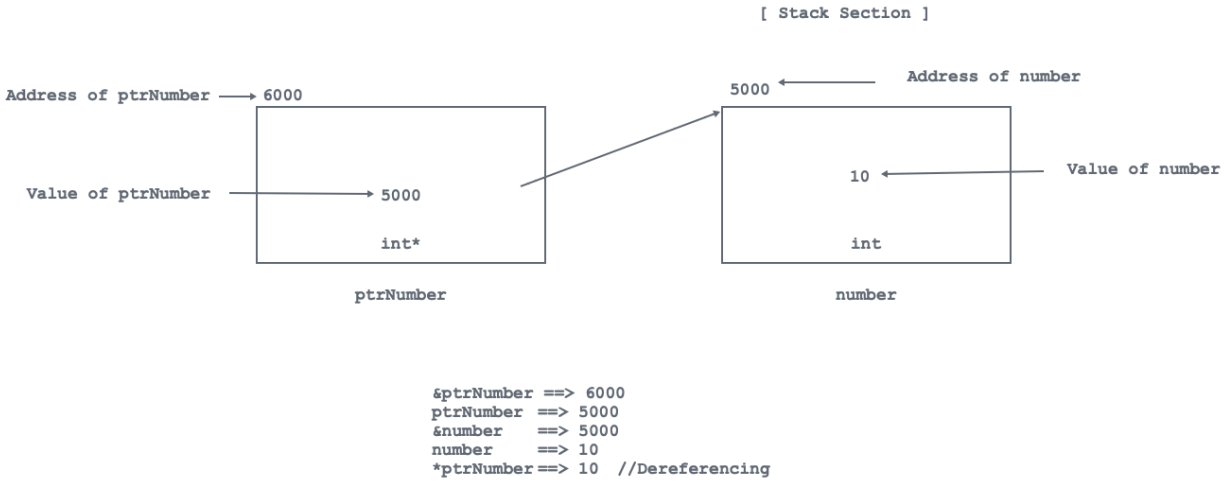
- To initializer pointer or to avoid dangling pointer we should use NULL;
  - Example 4

```
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
```

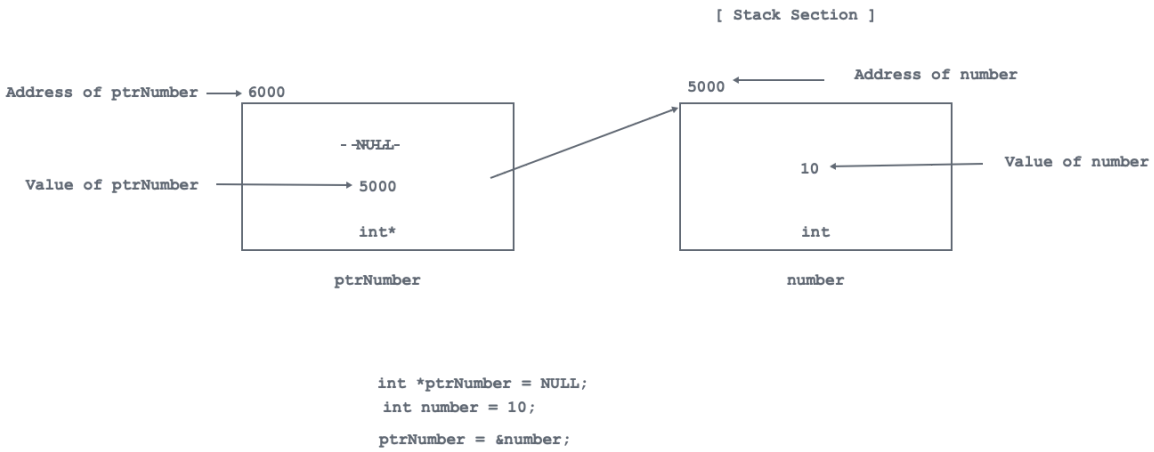



```
&ptrNumber ==> 6000
ptrNumber  ==> 5000
&number    ==> 5000
number     ==> 10
*ptrNumber ==> 10  //Dereferencing
```

- 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;
}
```

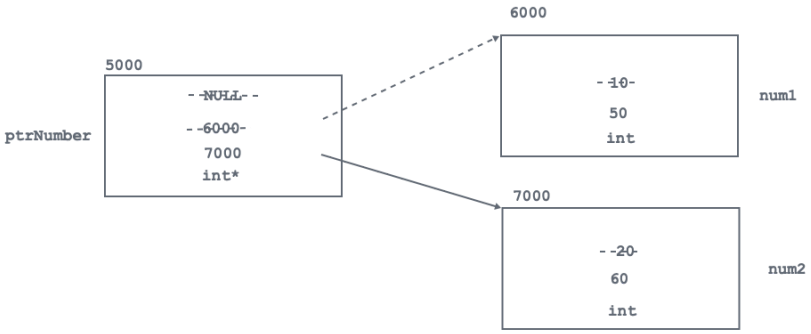
- We can not modiy value of constant variable but we can read its value. Hence it is called as read-only variable.

Constant and Pointer combinations

int *ptrNumber

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

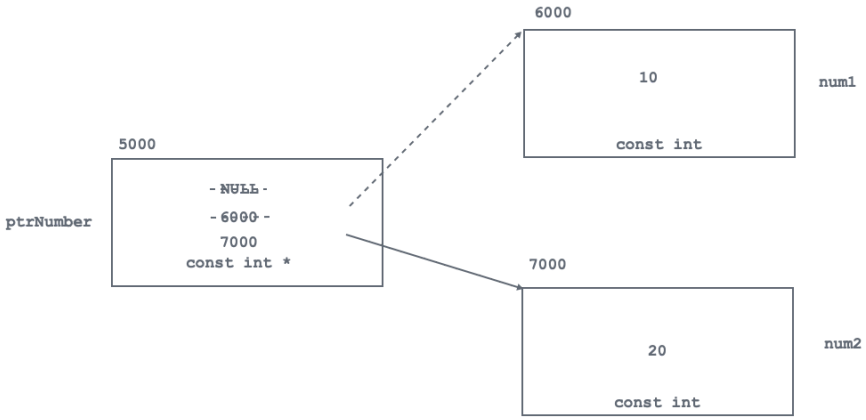
```c
int main( void ){
  int *ptrNumber = NULL;

  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:

```
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.

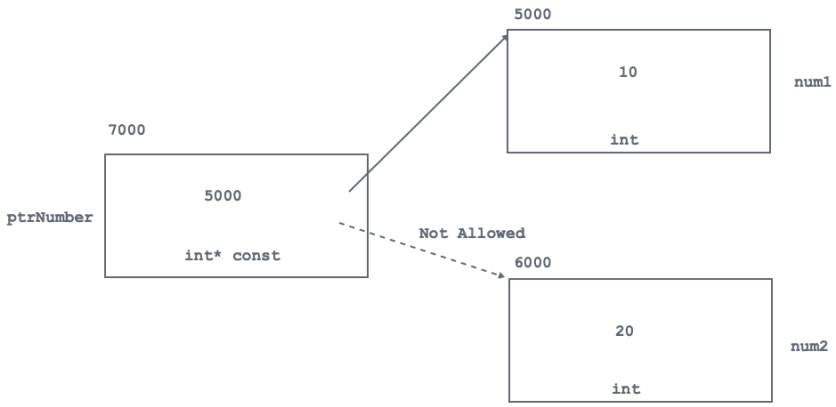
const int const *ptrNumber

- 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:

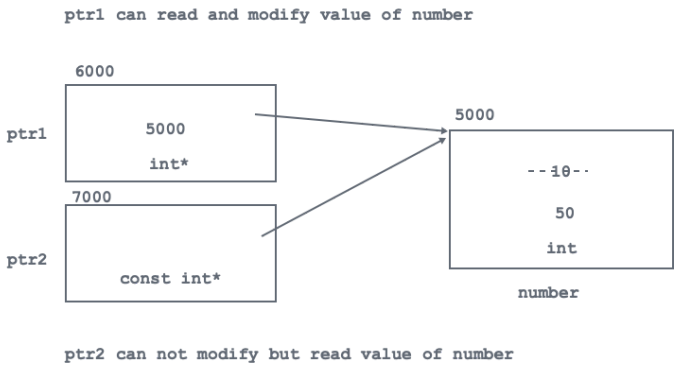
```
int main( void ){
  const int num1 = 10;  //OK
  const int *const ptrNumber = &num1;

  //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

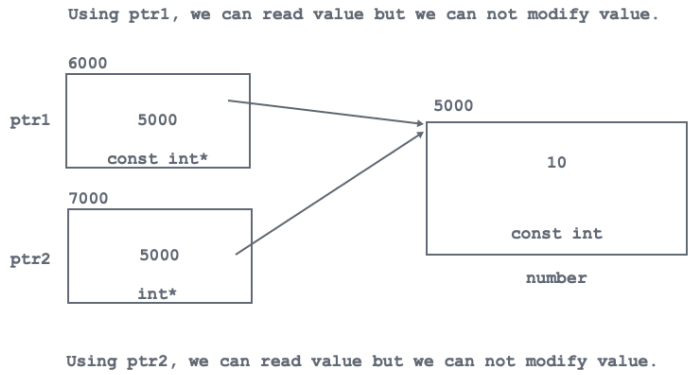
```
int main( void ){
    int number = 10;
    int *ptr1 = &number;
    *ptr1 = 50; //OK: Dereferencing
    printf("Number  :   %d\n", number); //50
    printf("Number  :   %d\n", *ptr1);  //50: Dereferencing

    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
```

```c
    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;
}
```

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:

```c
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:

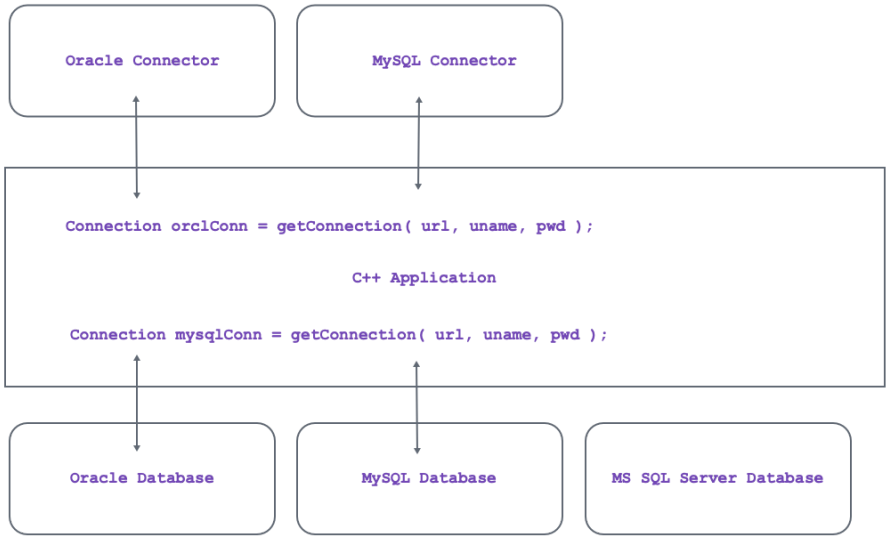
```
int num1 = 10;  //Global Variable
int main( void ){
  int num1 = 20;    //Local variable
  printf("Num1  :   %d\n", ::num1); //10
  printf("Num1  :   %d\n", num1);   //20

  {//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.

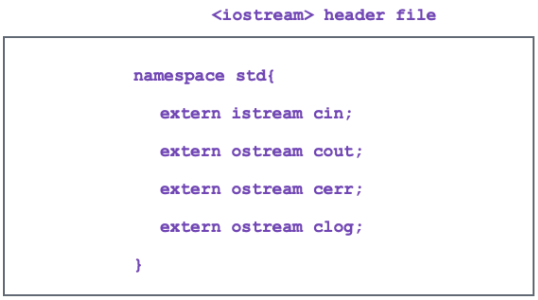
```
printf("%d", number);
//same as
fprintf(stdout, "%d", number);
```

  - 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( );

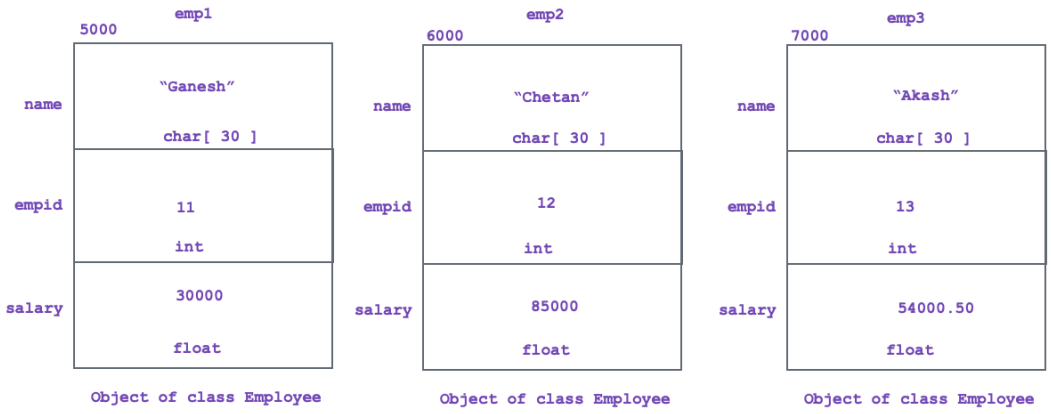
Coding Convention

- 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];

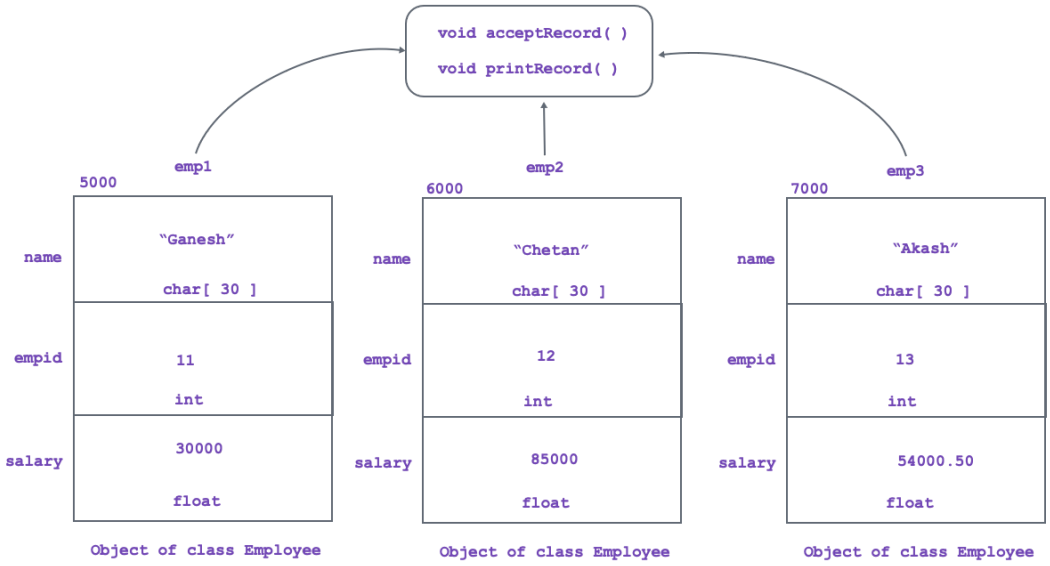
Object oriented concepts

- 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.

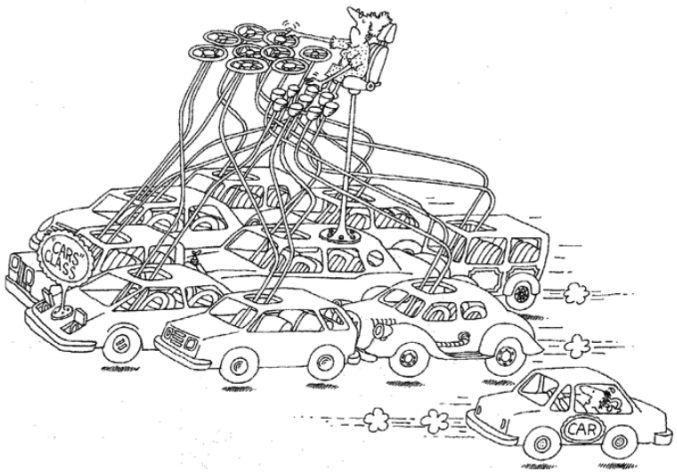
Characteristics of Object

- 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.

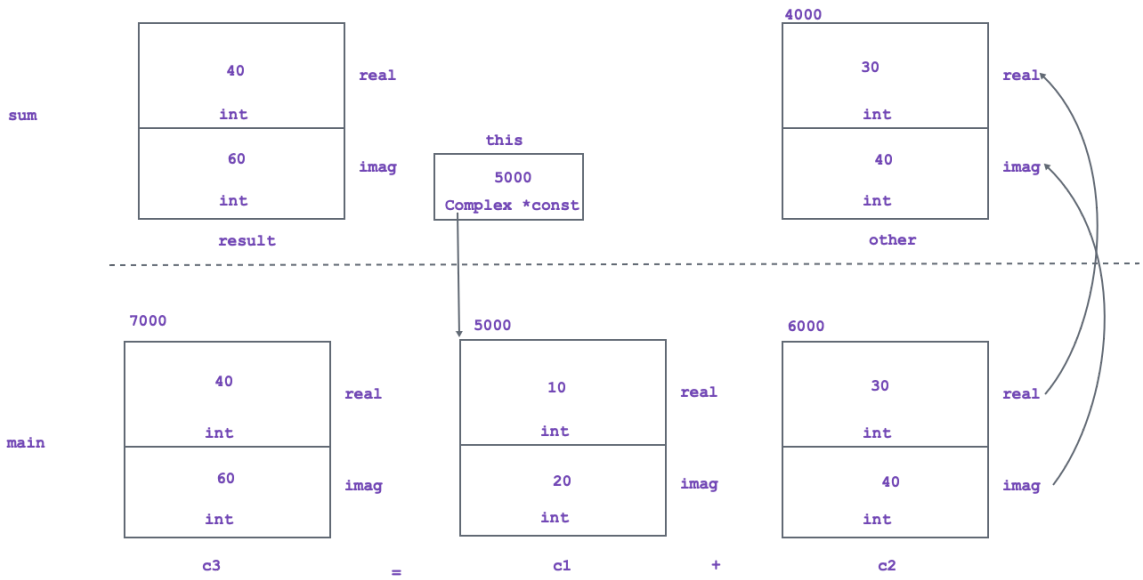
```
class Complex{
private:
  int real;
  int imag;
public:
  //Complex *const this = &c1
  void setReal( int real ){
    this->real = real;
  }
};
int main( void ){
  Complex c1;
  c1.setReal( 10 ); //c1.setReal( &c1, 10 );
  return 0;
}
```

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:

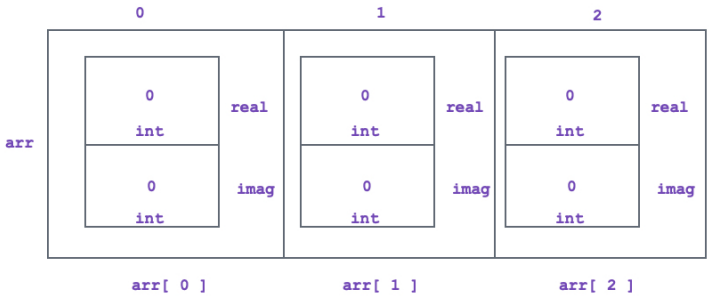
```
class Complex{
private:
  int real;
  int imag;
public:
  Complex( void ){
    this->real = 0;
    this->imag = 0;
  }
  Complex( int value ){
    this->real = value;
    this->imag = value;
  }
  Complex( int real, int imag ){
    this->real = real;
    this->imag = imag;
  }
  void printRecord(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

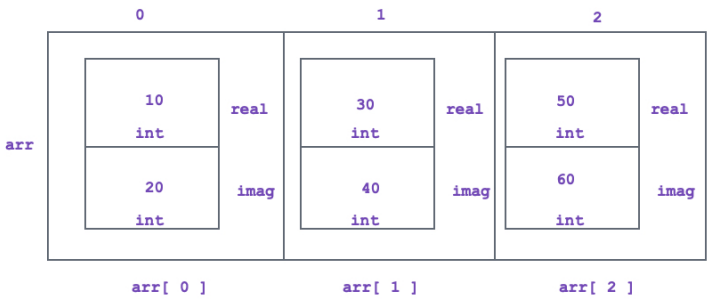
- Example 1:

```
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;
}
```

Reference

Exception Handling