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.

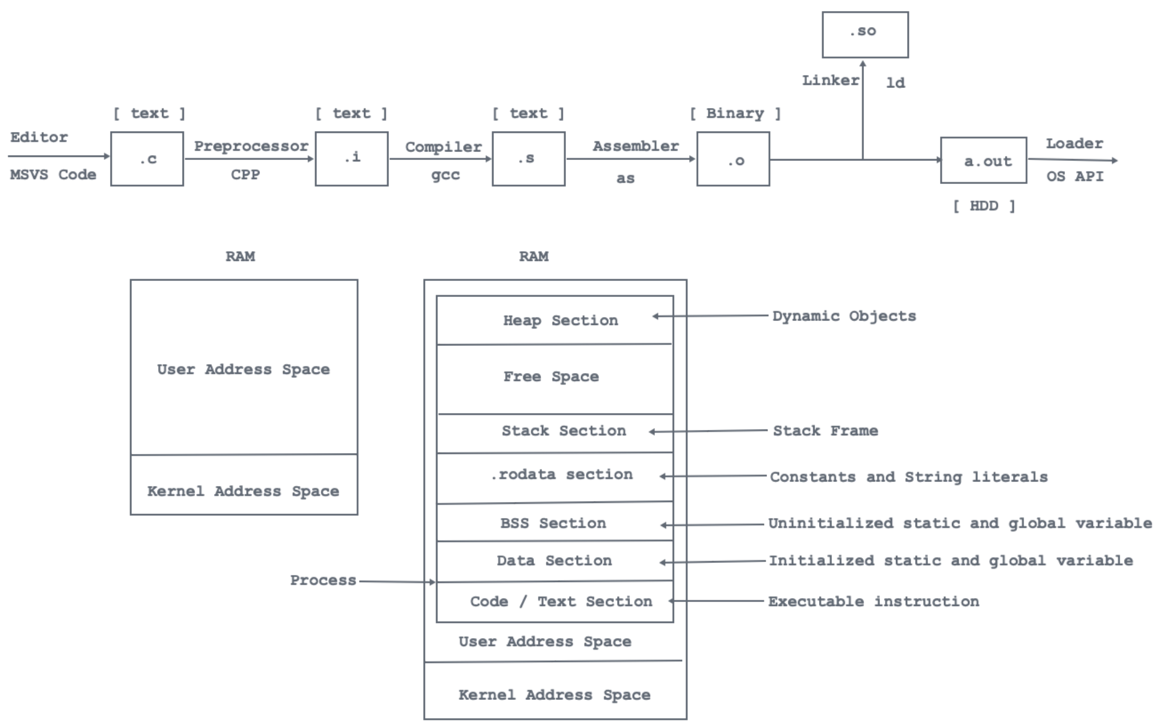
Software Development Kit

- 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:

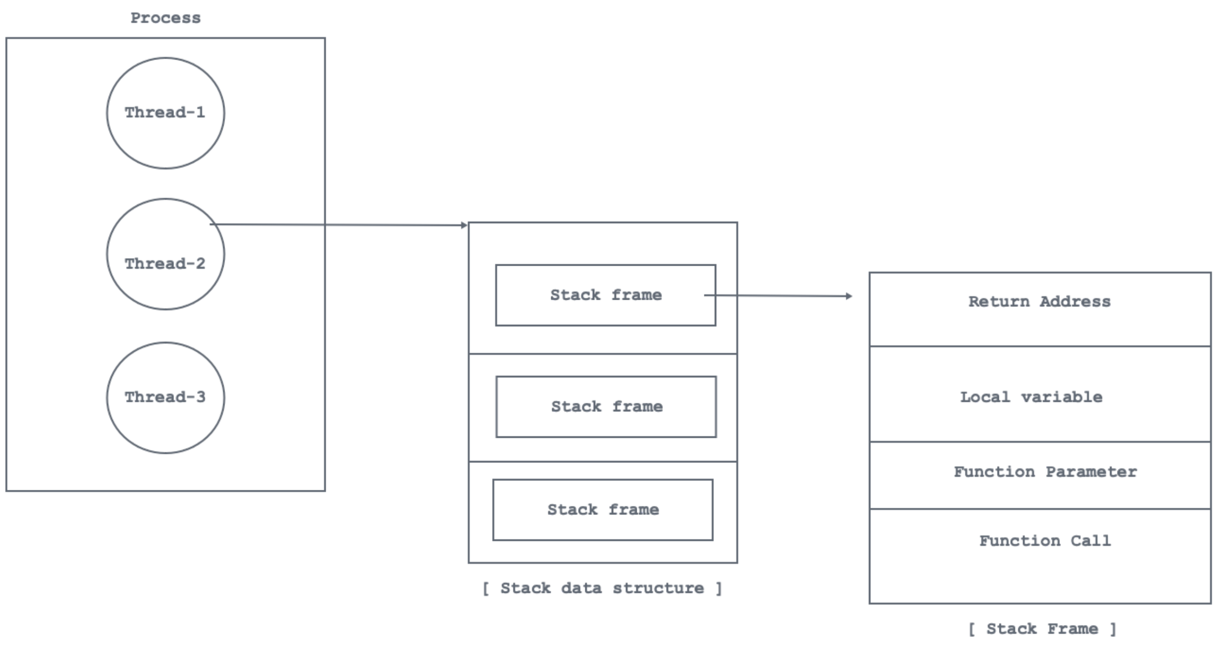
```
int number = 10;  //Initialization
number = 20;  //Assignment
```

- 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
```

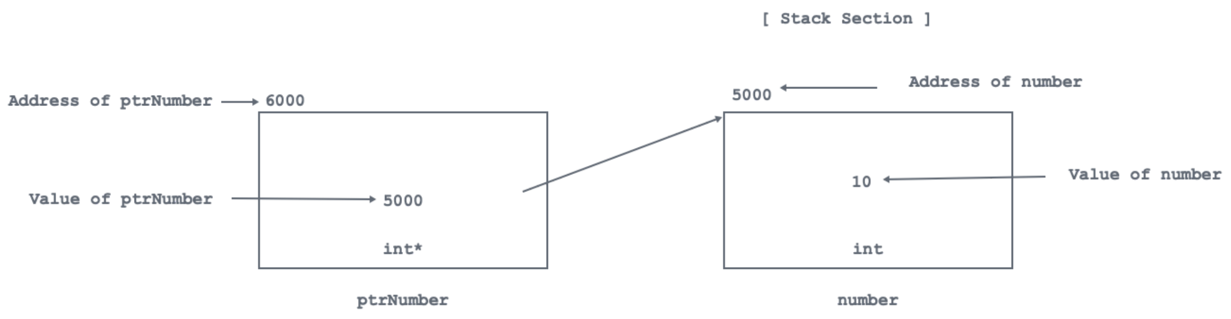
- To initializer pointer or to avoid dangling pointer we should use NULL;
  - Example 4

```
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
```

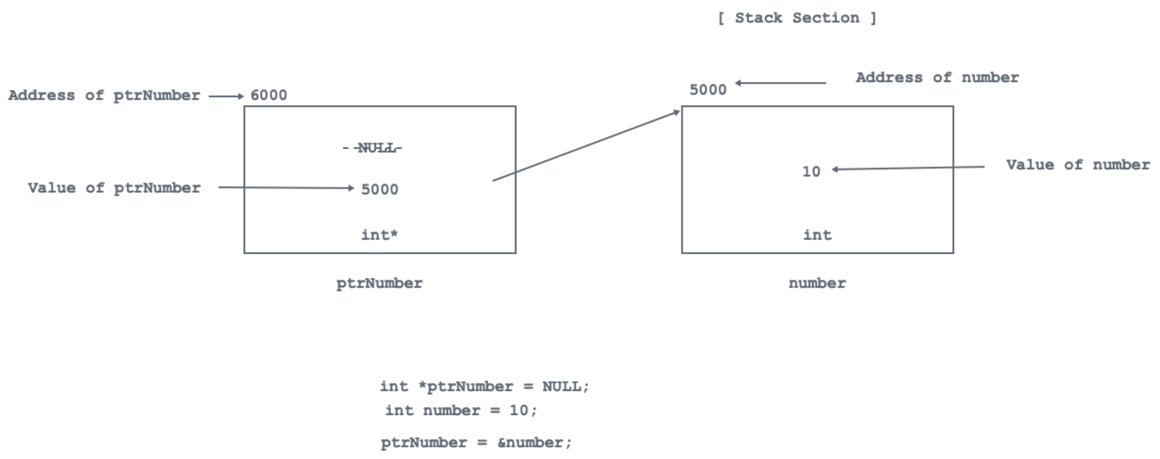


```
&ptrNumber ==> 6000
ptrNumber  ==> 5000
&number    ==> 5000
number     ==> 10
*ptrNumber ==> 10  //Dereferencing
```

- Pointer Assignment

```
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

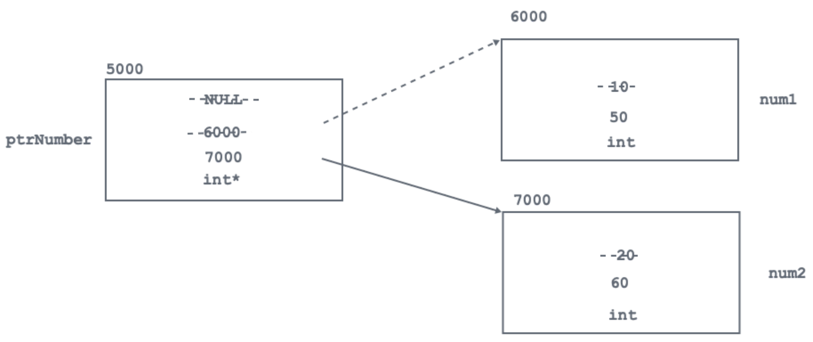
- Here ptrNumber is non constant pointer variable which can store address of non constant integer variable.
- Example:

```
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;
}
```



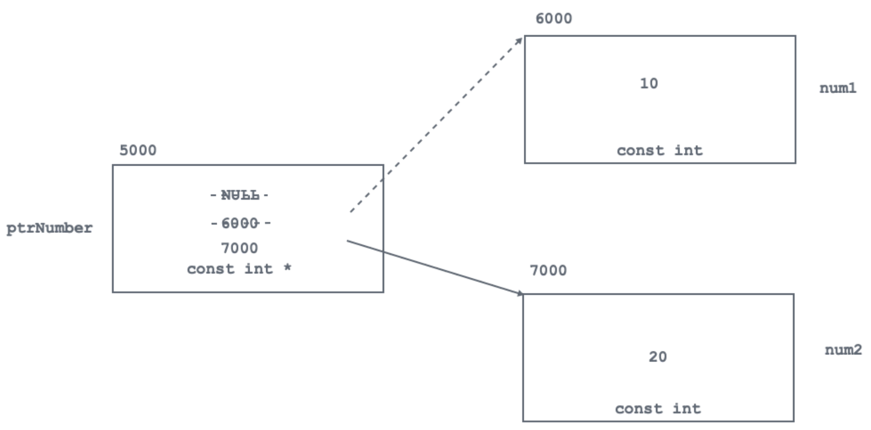
const int *ptrNumber

- 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

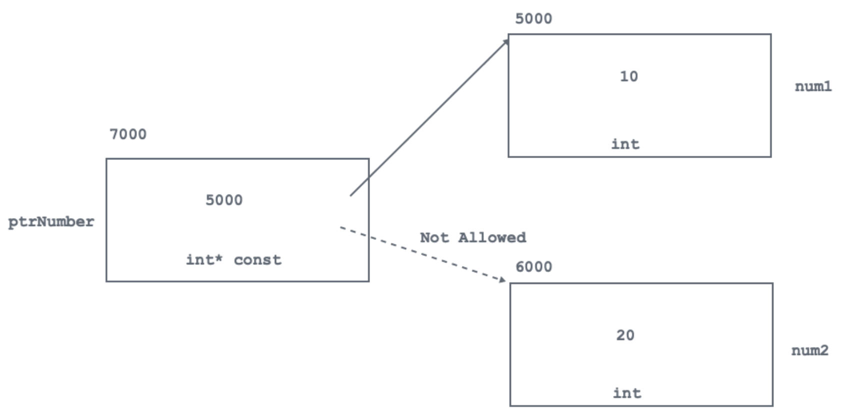
- const int *ptrNumber, int const *ptrNumber and const int const *ptrNumber are same.
- warning: duplicate 'const' declaration specifier

int *const ptrNumber

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

```
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

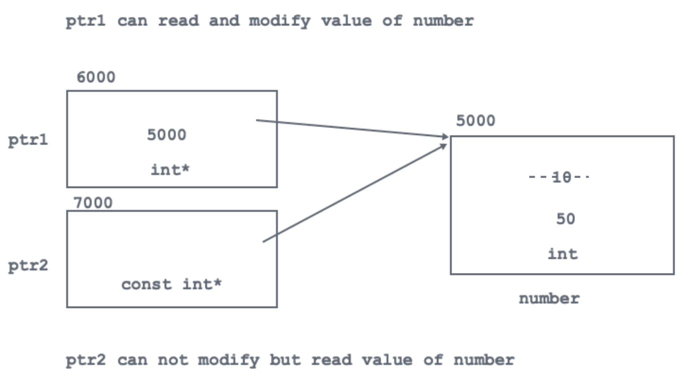
- const int *const ptrNumber and int const *const ptrNumber are same.

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;
}
```



ptr1 can read and modify value of number
6000
ptr1
5000
int*
7000
ptr2
const int*
5000
~~10~~
50
int
number
ptr2 can not modify but read value of number

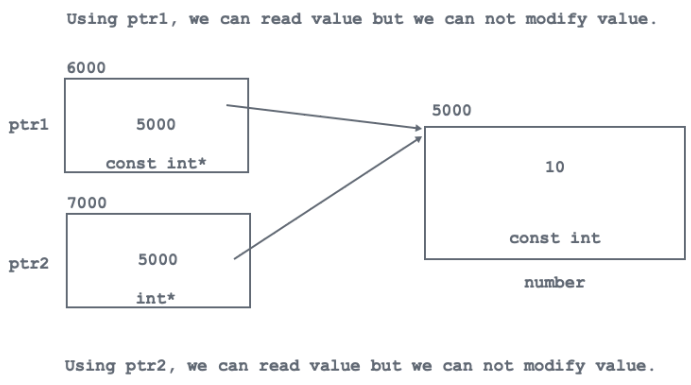
Consider following Pointer Example

```
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;
}
```


Using ptr1, we can read value but we can not modify value.
6000
ptr1
5000
const int*
7000
ptr2
5000
int*
5000
10
const int
number
Using ptr2, we can read value but we can not modify value.

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

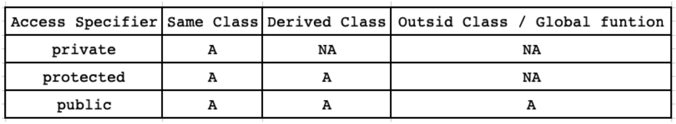
- const
- volatile

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:

```
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;
}
```

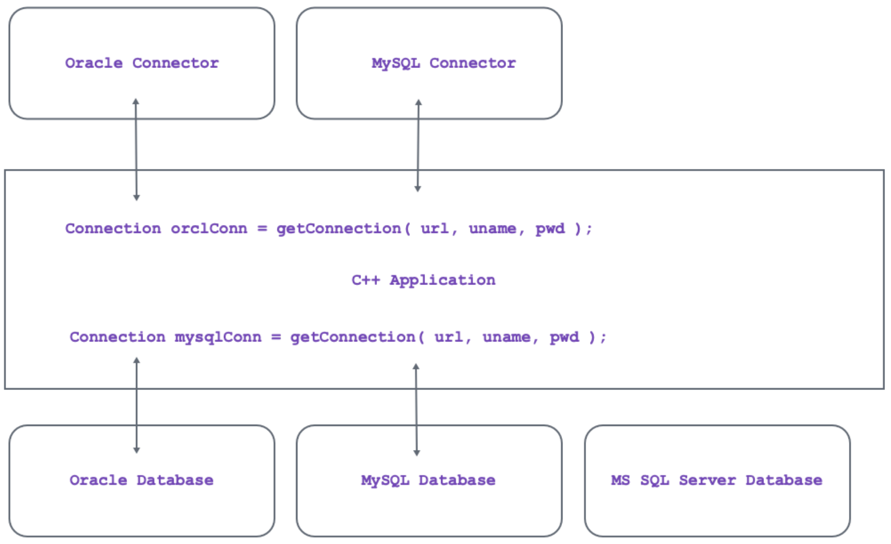
- We can use scope resolution operator with function too.

```
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;
}
```


Oracle Connector
MySQL Connector
Connection orclConn = getConnection(url, uname, pwd);
C++ Application
Connection mysqlConn = getConnection(url, uname, pwd);
Oracle Database
MySQL Database
MS SQL Server Database

- 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;
}
```