Day 4 (04-12-2025)

Namespace

- Namespace is a logical grouping of the classes, functions, variables, etc.
- Namespace is used to avoid the name collision, avoid the ambiguity in the code.
- Namespace is used to organize the code in a logical way and to avoid the global scope pollution.

Examples:

```
namespace customer{
//customer
   int calculate(int a, int b){
    return a + b * 0.18;
   }
}

namespace vendor{
   //vendor
   int calculate(int qty){
    return qty;
   }
}
```

```
namespace customer{
//customer
    int calculate(int a, int b){
        return a + b * 0.18;
    }
}

namespace vendor{
    //vendor
    int calculate(int qty){
        return qty;
    }
}

// math.h
using namespace std;  // using directive
int main(){
    using namespace customer;  // customer
//  calculate(10, 20); //scope resolution -> vendor -> scope is present
    cout << calculate(10, 20) << endl;
    cout << vendor::calculate(10) << endl;
```

```
    return 0;
}
```

```
namespace A{
    int val = 10; //namespace variable
}

namespace B{
    int val = 20; //namespace variable
}

int val = 100; //global variable

int main(){

    using namespace B; // use B

//  int val = 50; // ambigious error

//  cout << val << endl;   // 50
    cout << ::val << endl; // 100
    cout << A::val<< endl; // 10
    cout << B::val<< endl; // 20
    return 0;
}
```

```
#include <iostream>
using namespace std; //using directive for standard namespace
namespace CDACMumbai{ // main namespace
    namespace PGDAC{ // nested namespace
        int totalStudent = 240;

        int getCount(){
            return totalStudent;
        }
    }

    namespace PGDBDA{ // nested namespace
        int totalStudent = 80;

        int getCount(){
            return totalStudent;
        }
    }
}

int main(){
    using namespace CDACMumbai::PGDAC;  // with using directive
```

```
//  cout << "PG-DAC Student : " << PGDAC::totalStudent << endl;
//  cout << "PG-DBDA Student : "<< PGDBDA::totalStudent << endl;
    cout << "PG-DAC Student via function : " << getCount() << endl;
    cout << "PG-DBDA Student via function : "<< getCount()<< endl;
    return 0;
}
```

Function Overloading

- We can keep the name of a function same and We can have different parameters, type of parameter, sequence of parameters -> Function Overloading -> Compile-Time Polymorphism
- This is not allowed in the C languages.
- This is a concept of C++ and another OOP languages.
- Function overloading is used to provide multiple definitions of the same function name.
- Function overloading is used to provide multiple definitions of the same function name with different parameters.
- Return type is not considered for function overloading.

```
#include <iostream>

using namespace std;

//return_type method_name(diff parameters [data type], sequence){definition} //
whole function structure

//method_name(diff parameters [data type], sequence){definition} //function
signature

void display(int val, int lav){
    cout << "diplay for int" <<endl;
    cout << val << endl;
}

//overloading
void display(float lav, float val){
    cout << "display for float" <<endl;
    cout << val << endl;
}

int display(float num){ // example for float -> int conversion
    return num;
}

int main(){

    int a = 10;
    int b = 20;

    float c = 10.5;
    float d = 20.5;
```

```
    display(a,b); // int promote -> it is causing the issue
//  display(10.5, 29.5);  // error -> 'display' is ambiguous'
    display(c,d); // valid -> calls float function

    cout << display(10.5) << endl;

    return 0;
}
```

Name Mangling

- Used by C++ to uniquely name the functions in their own way -> to support function overloading

```
C -> display(int num) -> stored name: display     |
  -> display(float num) -> stored name: display   | -> Ambiguity issue ->
Redefinition of display not allowed
-------------------------------------------------------------
C++ -> display(int num) -> stored name: _Z3displayi   |
    -> display(float num) -> stored name: _Z3displayf | -> different & C++
compiler treats them as a different functions
```

> tool -> nm -> used to list the symbols in the object file -> `mangled name` our use case.

```
void display(int val, int lav){ //__Z3displayii
}
//overloading
void display(float lav, float val){ //__Z3displayff
}
//__Z3displayf
int display(float num){
    return num;
}
int main(){
    return 0;
}
```

extern “C”

extern keyword (storage class)

- which is used to declare a variable somewhere else(another file) and define it somewhere else(another file) -> it gets the memory where your definition is done for that perticular variable.

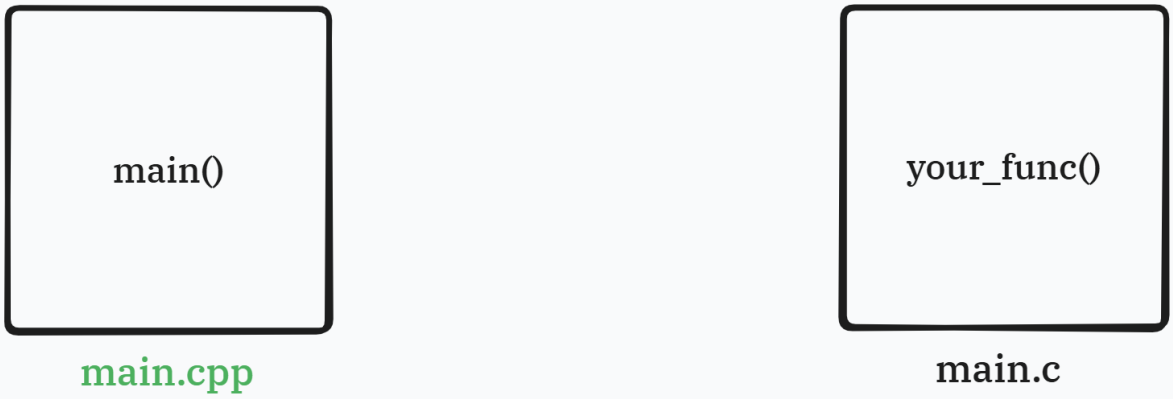
extern "c"

- This is something else, which is used in c++ to tell your compiler that dont mangle the name of the function. -> keep it in the format of your c type.
- Whenever we are working with C API, embedded, a project C & C++ is used together, external libraries which demands this things.
- This is used to avoid the name mangling in C++.

```
#include <iostream>

using namespace std;

extern"C"
    //display
    {
    void display(float val){
        cout << val << endl;
    }
    //error: conflicting declaration of C function 'void display(int)'
    //  void display(int val){
    //      cout << val << endl;
    //  }
}// end of extern "C"

//__Z3displayi
void display(int val){
    cout << val << endl;
}

int main(){
//  print(10);
    return 0;
}
```

> Implementation of extern "C" in C++ compiler with linking of C object file.

```
gcc -c main.c -o main_c.o      # compile C file (your_function)
g++ -c main.cpp -o main_cpp.o  # compile C++ file (main)
g++ main_c.o main_cpp.o -o practice_new.exe  # link them
./practice_new.exe          # run (on Windows: .\practice_new.exe)
```



Enum

- Group of Constant values which can be of single logical group.
- Enum is used to define a set of named integer constants.
- Enum is used to define a set of named constants which are related to each other.

Enum Class

- Enum class is a scoped enumeration.
- Enum class is used to define a set of named constants which are related to each other and it is scoped.
- Enum class is used to avoid the name collision and ambiguity.

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

//define enum Normal Color
//enum Color{
//  ORANGE, BLUE, GREY
//};

// define enum SignalColor
enum class SignalColor{
    RED, YELLOW, GREEN //RED = 0, YELLOW = 1, GREEN = 2
};

enum class Color{
    ORANGE, BLUE, GREY
};

enum class Demo {
    A = 10, B = 20, C = 30, D = 40  // 0, 1, 2, 3
```

```
};

enum class ErrorCode{
    NOT_FOUND = 404,
    INTERNAL_SERVER_ERROR = 500,
    ACCESS_DENIED = 401
};

enum Zemo{
    A, B = 10, C, D = 20, E
};

int main(){
    Color c;
//  if(Color::c == SignalColor::sc){
//      cout << "color are same" << endl;
//  }

//  Demo d;
//
//  cout << Demo::A << endl;
//  cout << Demo::B << endl;
//  cout << Demo::C << endl;
//  cout << Demo::D << endl;
    Zemo x;

    cout << Zemo::A <<endl;
    cout << Zemo::B <<endl;
    cout << Zemo::C <<endl;
    cout << Zemo::D <<endl;
    cout << Zemo::E <<endl;
    return 0;
}
```

Default arguments

- Default arguments are used to provide default values for the function parameters.
- Default arguments are used to provide default values for the function parameters so that we can call the function without passing any arguments.
- Default arguments are used to provide default values for the function parameters so that we can call the function with fewer arguments.
- Do not mix the concepts of function overloading and default arguments.
- Examples:

```
// valid -> works with all the function call syntax
void add(int a = 10, int b = 10){ // default arguments
    cout << "add function" << endl;
```

```
    cout << a << " " << b << endl;
    cout << a+b << endl;
}
```

```
// valid -> add(); will not work
void add(int a, int b = 10){ // default arguments
    cout << "add function" << endl;
    cout << a << " " << b << endl;
    cout << a+b << endl;
}
```

```
//not valid : Compilation Error
void add(int a = 10, int b){ // default arguments
    cout << "add function" << endl;
    cout << a << " " << b << endl;
    cout << a+b << endl;
}
```

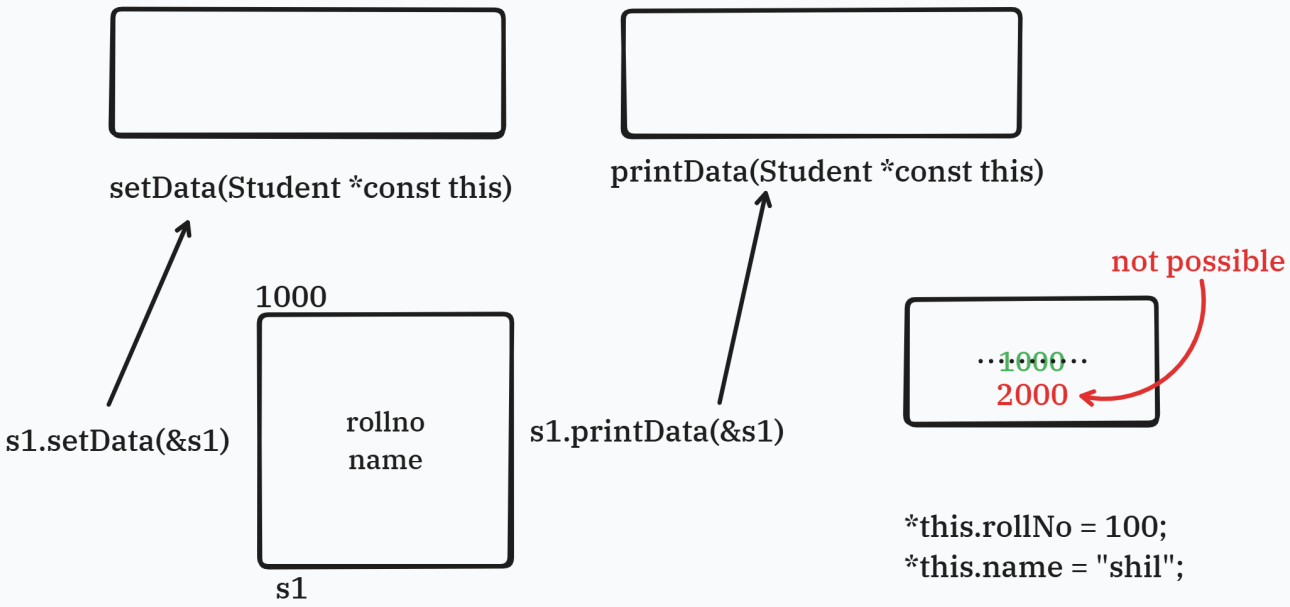
```
//  add();      // compilation error -> arguments are not enough
//  add();      // 20  -> add: error case 2
    add(50);    // 60
    add(80,80); // 160
```

this pointer

- In Java this -> Object reference
- In C++ this -> pointer
- this pointer is associated with all the member functions
- static function this is not associated -> one for class level, static actually out of the context of class
- this pointer is used to refer the current object of the class
- this pointer is used to return the current object from the member function
- this pointer is used to resolve the ambiguity in the member function

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

class Student{
    //data members
    int rollNo;
```

```
    string name;

public:
    //member functions
    // Student *const this
    void setData(int rollNo, string name){
        this->rollNo= rollNo;
        this->name= name;
    }

    void printData(){
        cout << "Roll No :" <<rollNo << endl;
        cout <<"Name : " << name << endl;
    }
};

int main(){
    Student s1;

//for global in c/c++
//  setData(&s1);
//  printData(&s1);

    s1.setData(101,"shil");
    //setData(&s1, remaining_parameters);
    s1.printData();
    return 0;
}
```



Stdudent s1 -> object -> data change -> s1 ka address must not change
if changed -> gbvalue
-> undefined behaviour

Tomorrow's topic

- Initialization vs Assignment
- Constructor types
    - Default, Parameterized, Copy
- Constructor initializer list
- Aggregate initialization
- Arrays of objects