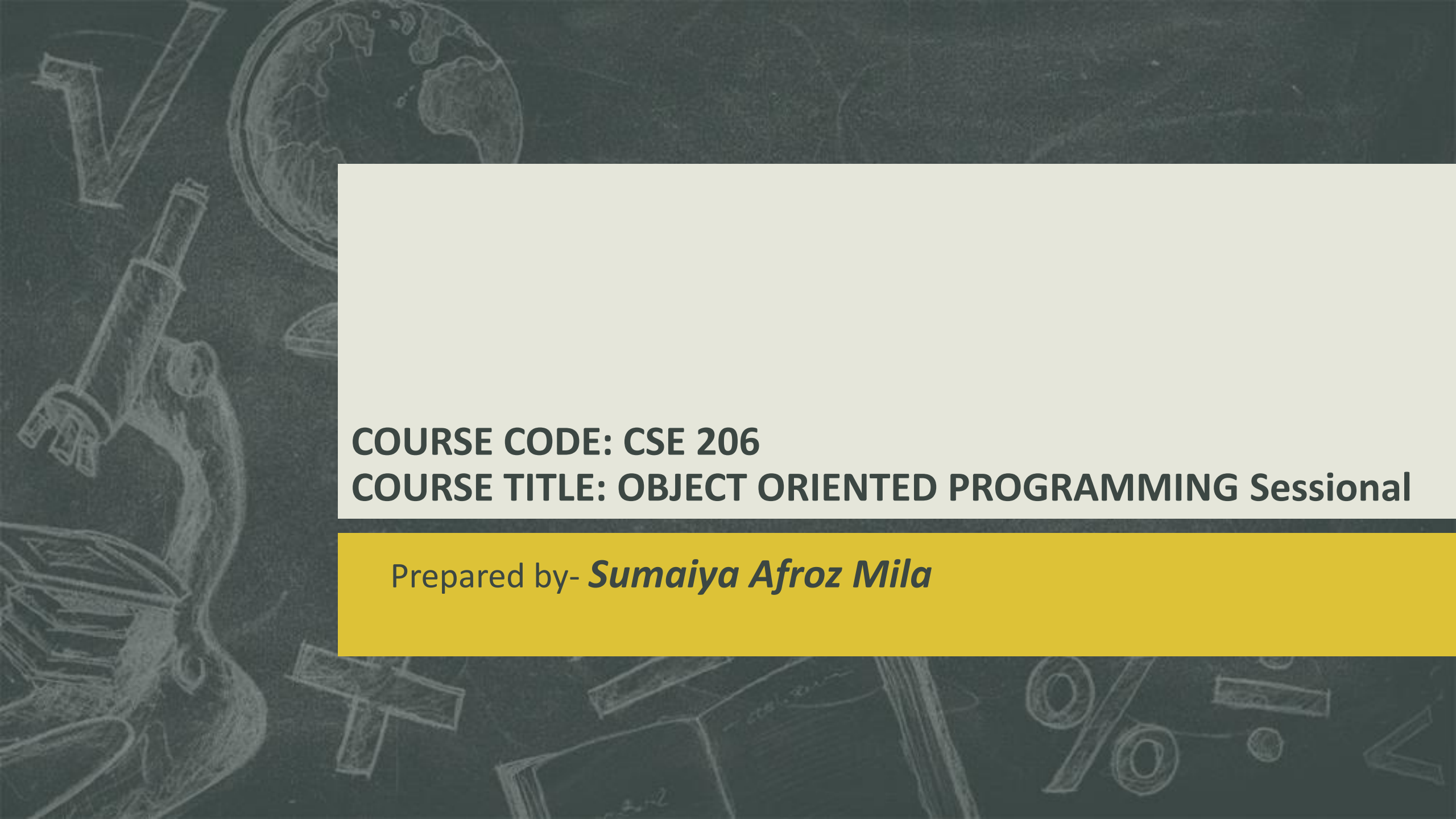


COURSE CODE: CSE 206
COURSE TITLE: OBJECT ORIENTED PROGRAMMING Sessional

Prepared by- ***Sumaiya Afroz Mila***

C++ Program Structure

- Line 1 & 2 : Preprocessor directives
  - Lines beginning with # are preprocessor commands
  - Include the header file for functions like cin, cout, etc
  - Set the default namespace as standard
- Line 3: Entry point of the program. Begin execution here.
- Line 4-6 : Body of the program

Untitled1.cpp | Untitled1.c

```
1  #include<iostream>
2  using namespace std;
3  int main()
4  {
5      cout<<"Hello CSE-19";
6  }
7
```

C vs C++ Output

```
#include <iostream>
using namespace std;
int main(){
cout<<“hello world”;
}
```

```
#include <stdio.h>
int main(){
printf(“hello world”);
}
```

```
#include <iostream>
using namespace std;
int main(){
int a;
cin>>a;
}
```

```
#include <stdio.h>
int main(){
int a;
scanf(“%d”,&a);
}
```

C vs C++ Output (cntd.)

C++

```
#include<iostream>
using namespace std;
main(){
int i;
float j;
char c;
cin>> i >> j >> c;
cout<< i <<“ “<<j<<“ “<<c<<endl;
}
```

C

```
#include<stdio.h>
main(){
int i;
 float j;
char c;
scanf(“%d%f%c”,&i,&j,&c);
printf(“%d %f %c\n”,i,j,c);
}
```

Why Using ‘namespace std’?

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

namespace first_space{
    void printf(char *p){
cout << "Inside first_space" << endl; }
 }

int main(){
   printf("hello world");
   first_space::printf("hello world");
}
```

What will be the output?

```
1   #include<iostream>
2   using namespace std;
3
4   void display()
5   {
6       cout<<"Display Function 01"<<endl;
7
8   }
9   void display()
10  {
11      cout<<"Display Function 02"<<endl;
12
13  }
14  int main()
15  {
16      display();
17  }
```

What will happen???

Compilation error

- One way to solve this is function overloading.(We will learn this technique in later sessional class)

- Another way is to use user defined namespaces

What is a namespace?

- Additional information to differentiate similar functions, classes, variables etc. with the same name, available in different libraries.
- Namespace, defines a scope. To call namespace-enabled version of either function or variable, prepend the namespace name to the function / variable :

```
std::cout << “hello”<<std::endl;
std::cin.get();
```

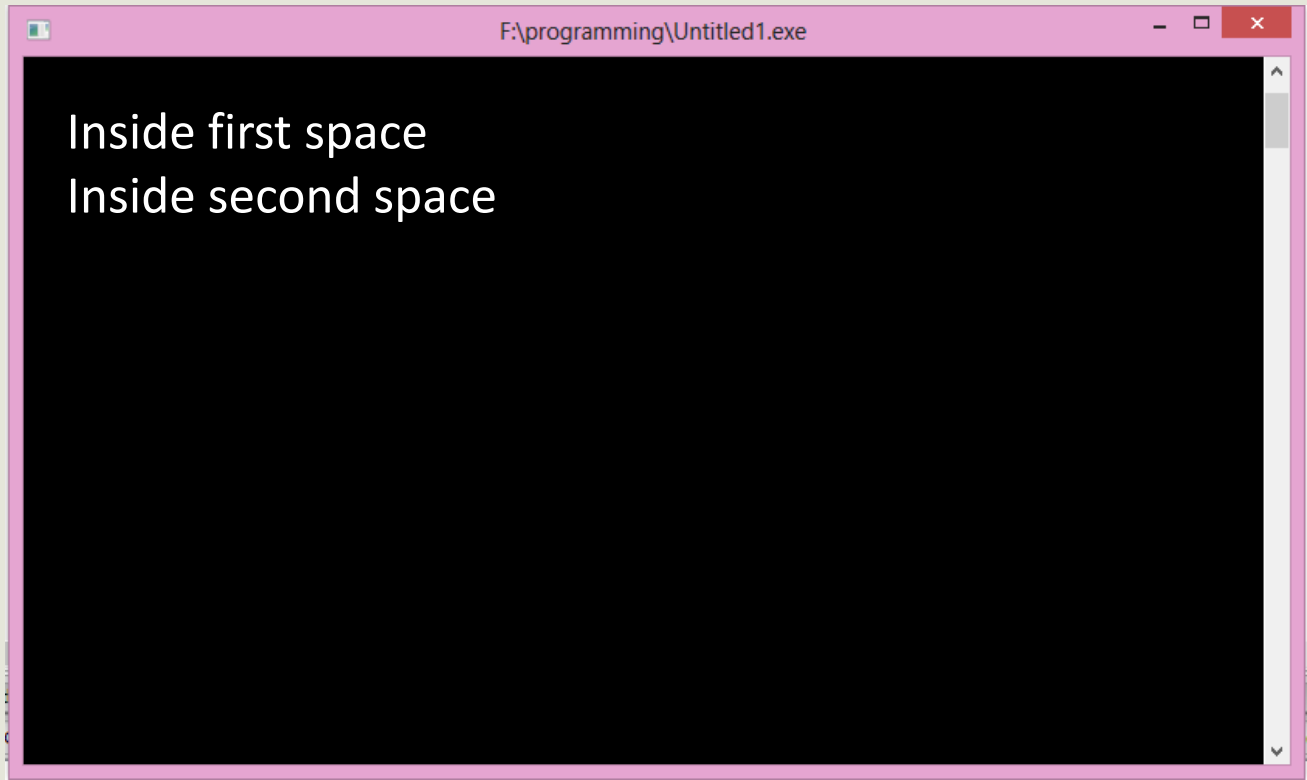
- This can be avoided with preprocessor directives :

```
using namespace std; // std is abbreviation of namespace standard
cout <<“hello”<<endl;
```

Using user defined namespace

```
#include <iostream>
using namespace std;
namespace first_space{
    void display(){
cout << "Inside first_space" << endl; }
 }
namespace second_space{
    void display(){
cout << "Inside second_space" << endl; }
 }
int main () {
    // Call function from first name space.
first_space::display();
   // Call function from second name space.
second_space::display();
}
```

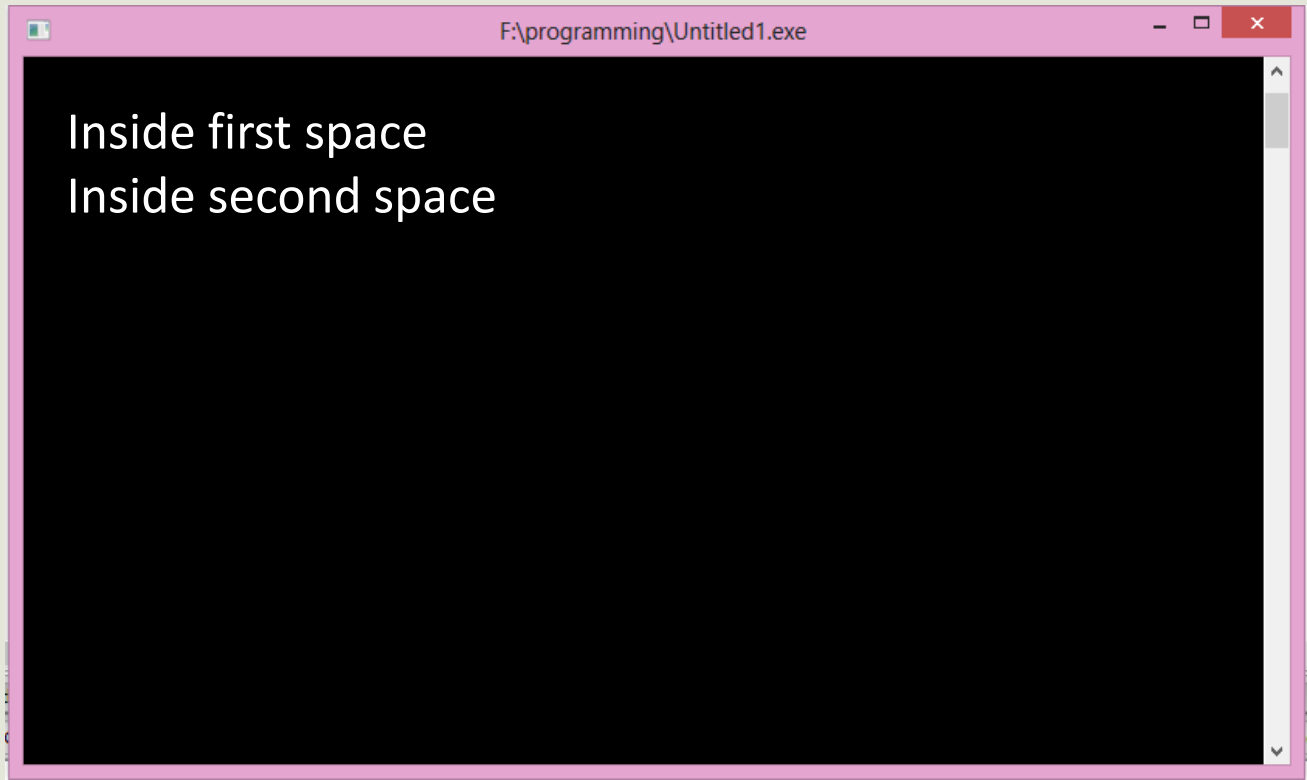
What will be the output?



Using user defined namespace

```
#include <iostream>
using namespace std;
namespace first_space{
    void display(){
cout << “Inside first space”<< endl;
} }
namespace second_space{
    void display(){
cout << “Inside second space”<< endl;
} }
Using namespace first_space ;
int main () {
    // Call function from first name space.
display();
   // Call function from second name space.
second_space::display();   }
```

What will be the output?



Using user defined namespace

1. **Define two namespaces “SquareSpace” and “CircleSpace” both containing a function with same name “Area”. Use necessary variables.**
2. **Take input of a variable a. Call both functions from main program and show the output. Provide the input as parameter.**
3. **The “Area()” function in the SquareSpace should return the area of a square considering the parameter.**
4. **The “Area()” function in the CircleSpace should return the area of a circle considering the parameter.**

String in C++

C++ provides following two types of string representations −

1.The C-style character string.

2.The string class type is introduced with Standard C++.

Output??

Input: Mist

Input: Programming is fun

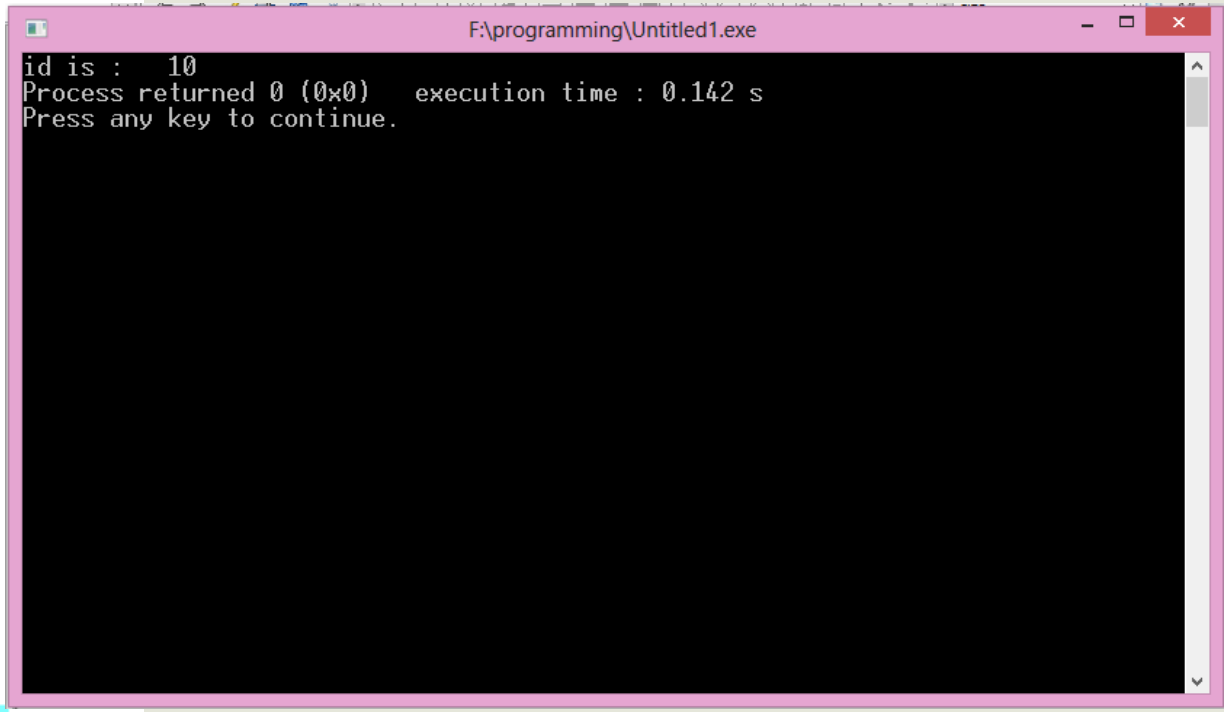
```
#include<iostream>
using namespace std;

int main(){
    string str;
    cin>>str;
    cout<<str;
}
```

Structure in C

Untitled1.c | Untitled1.cpp

```
1  #include<stdio.h>
2  struct employee{
3      int id;
4      char designation[100];
5      char branch[100];
6  };
7
8  int main(){
9      struct employee emp1;
10     emp1.id = 10;
11     printf("id is :    %d",emp1.id);
12 }
```

What will be the output?



Structure in C

```c
#include<stdio.h>
struct employee{
    int id;
    char designation[100];
    char branch[100];
    int getID(){
        return id;
    }
};
int main(){
    struct employee emp1;
    emp1.id = 10;
    printf("id is :%d",emp1.getID());
}
```

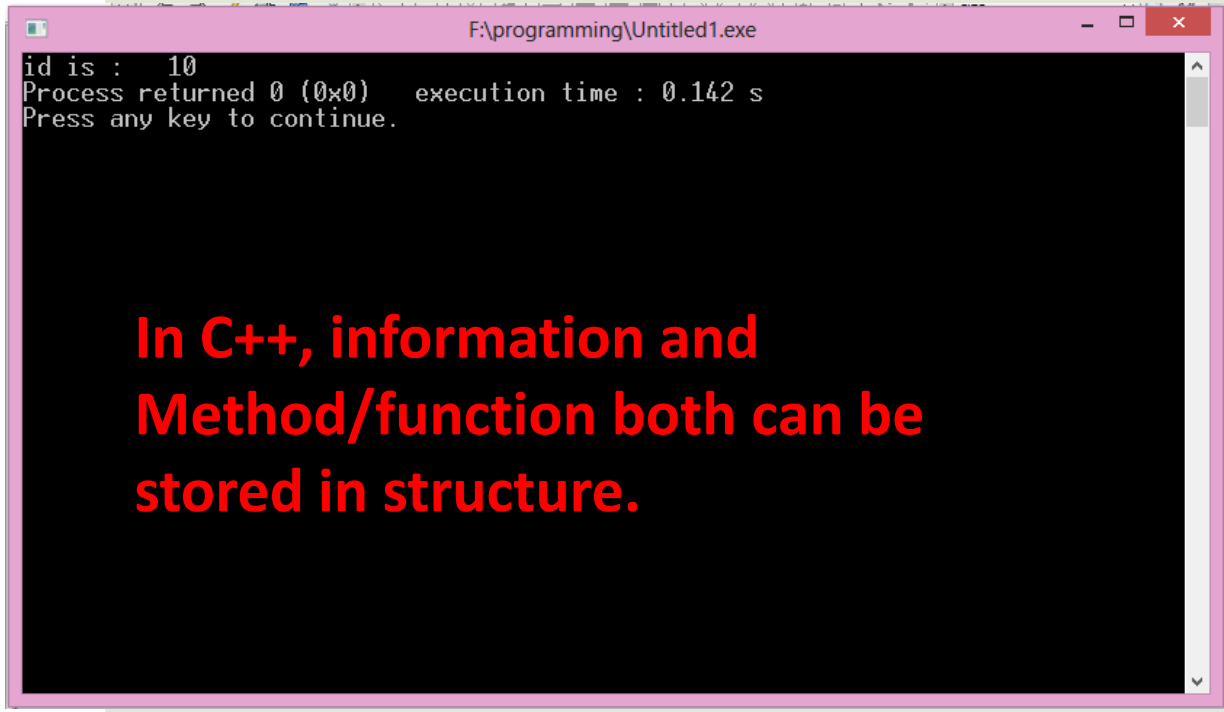
What will happen now?
What will be the output?

There will be compilation error. In C, only information can be stored in structure. Method/function can not be stored.

Structure in C++

```
#include<iostream>
using namespace std;
struct employee{
    int id;
    char designation[100];
    char branch[100];
    int getID(){
        return id;
    }
};
int main(){
    struct employee emp1;
    emp1.id = 10;
    printf("id is :%d",emp1.getID());
}
```

What will happen now?
What will be the output?



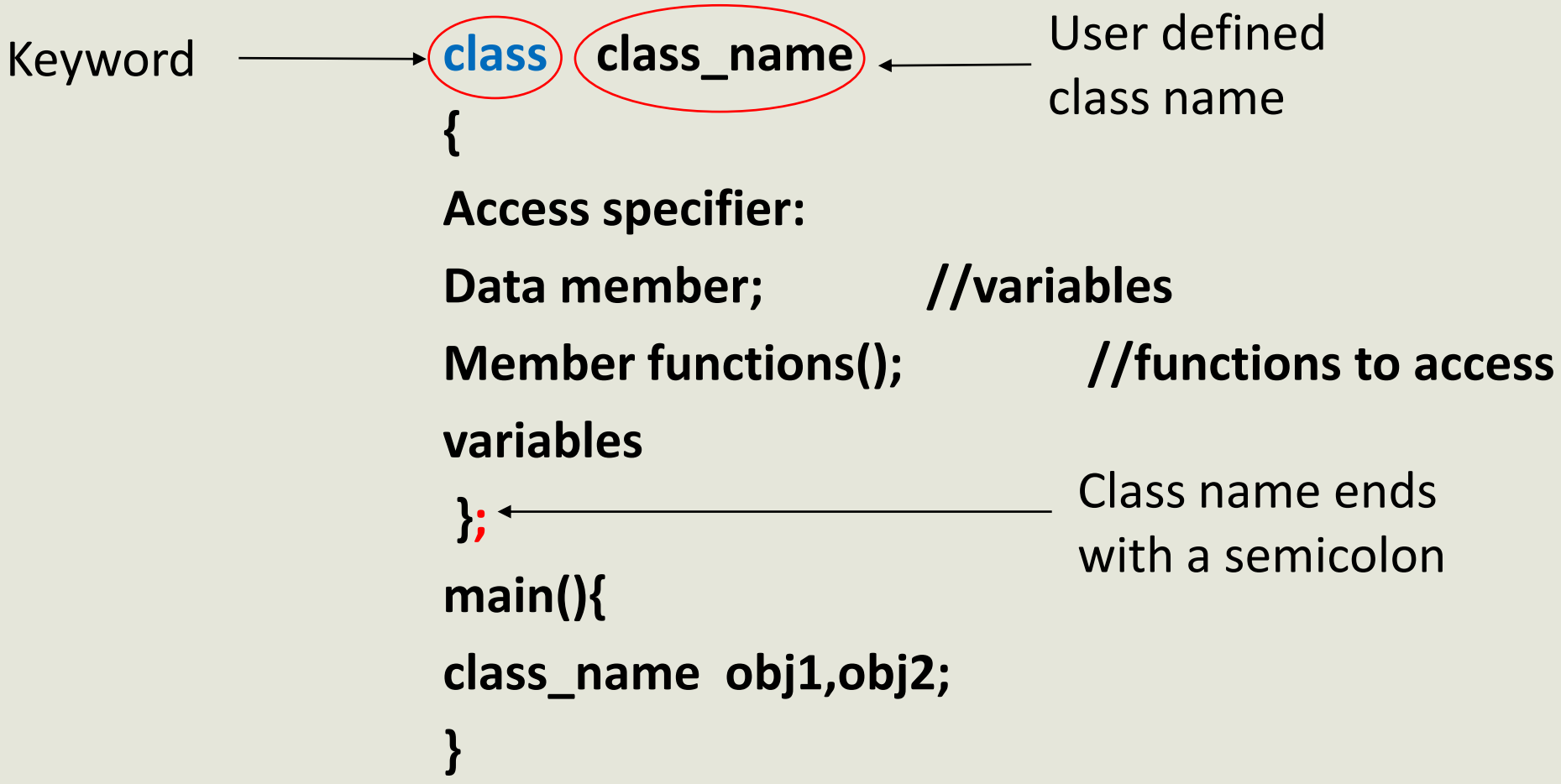
In C++, information and Method/function both can be stored in structure.

Class vs structure

```
struct struct_name{
        //all members
};
main(){
struct struct_name st1,st2;
 }
```

```
class class_name{
//private members
public:
 //public members
 };

main(){
class_name  obj1,obj2;
}
```

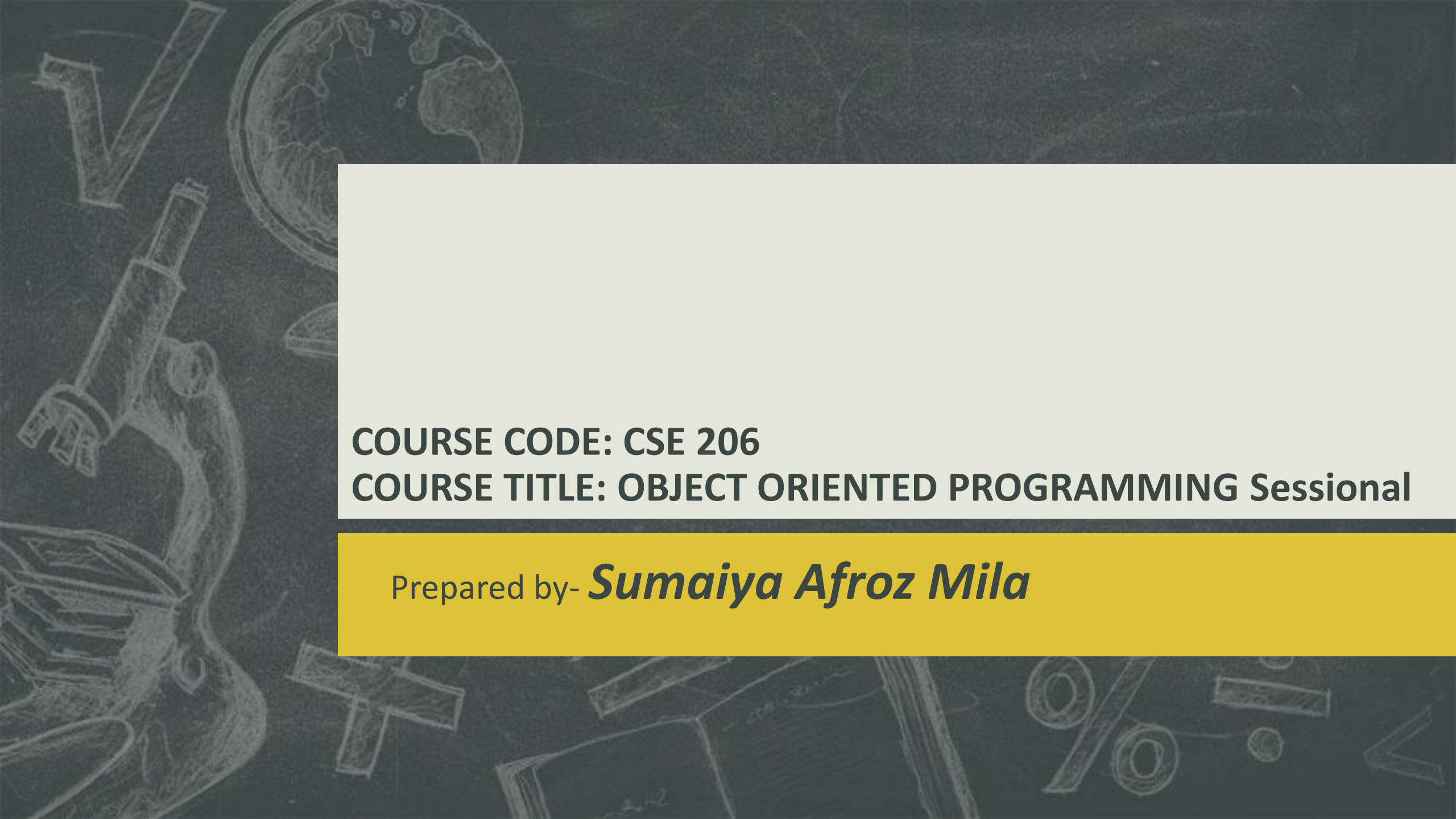
Class syntax



Class Task

- Now your task is to Create a Student class containing variables - ID, Name
- Make the variables private
- Create public functions to set the variables.
- Create public functions to show the variables.
- Create object of the student class in the main function.
- Set the ID and name of the object and show the values


Thank
you!



COURSE CODE: CSE 206
COURSE TITLE: OBJECT ORIENTED PROGRAMMING Sessional

Prepared by- ***Sumaiya Afroz Mila***

Design a student class

- Write a C++ program to create a student class.
- Your class will contain the variables - ID, Name, cgpa(assume all the members of the class are public)
- Create a member function named “Display()” to print the values of the variables.
- Create 2 objects of the student class in the main function.
- Set the ID, name and cgpa of both the objects from main function
- Call the display function and print the values of the variable

Design a student class

- Write a C++ program to create a student class.
- Your class will contain the variables - ID, Name, cgpa(assume all the members of the class are public)
- Create a member function named “Display()” to print the values of the variables.
- Create 2 objects of the student class in the main function.
- Take input of the ID, name and cgpa for both the objects from main function
- Call the display function and print the values of the variable

Design a Point Class

- Design a Point class which will contain the variables X, Y(keep the variables public)
- Create a function “SetX()” to set the value of X.
- Create a function “SetY()” to set the value of Y.
- Create a function “Distance (int X2, int Y2)” which will calculate and show the distance between two points (X,Y) and(X2,Y2).
- Formula to calculate distance: d = $\sqrt{(X_2 - X)^2 + (Y_2 - Y)^2}$
- Include the “math.h” to use sqrt() function

```
class Point{
public:
    int X;
    int Y;

    void setX();
    void setY();
    void Distance(int X2, int Y2);
};
```

Design a student class

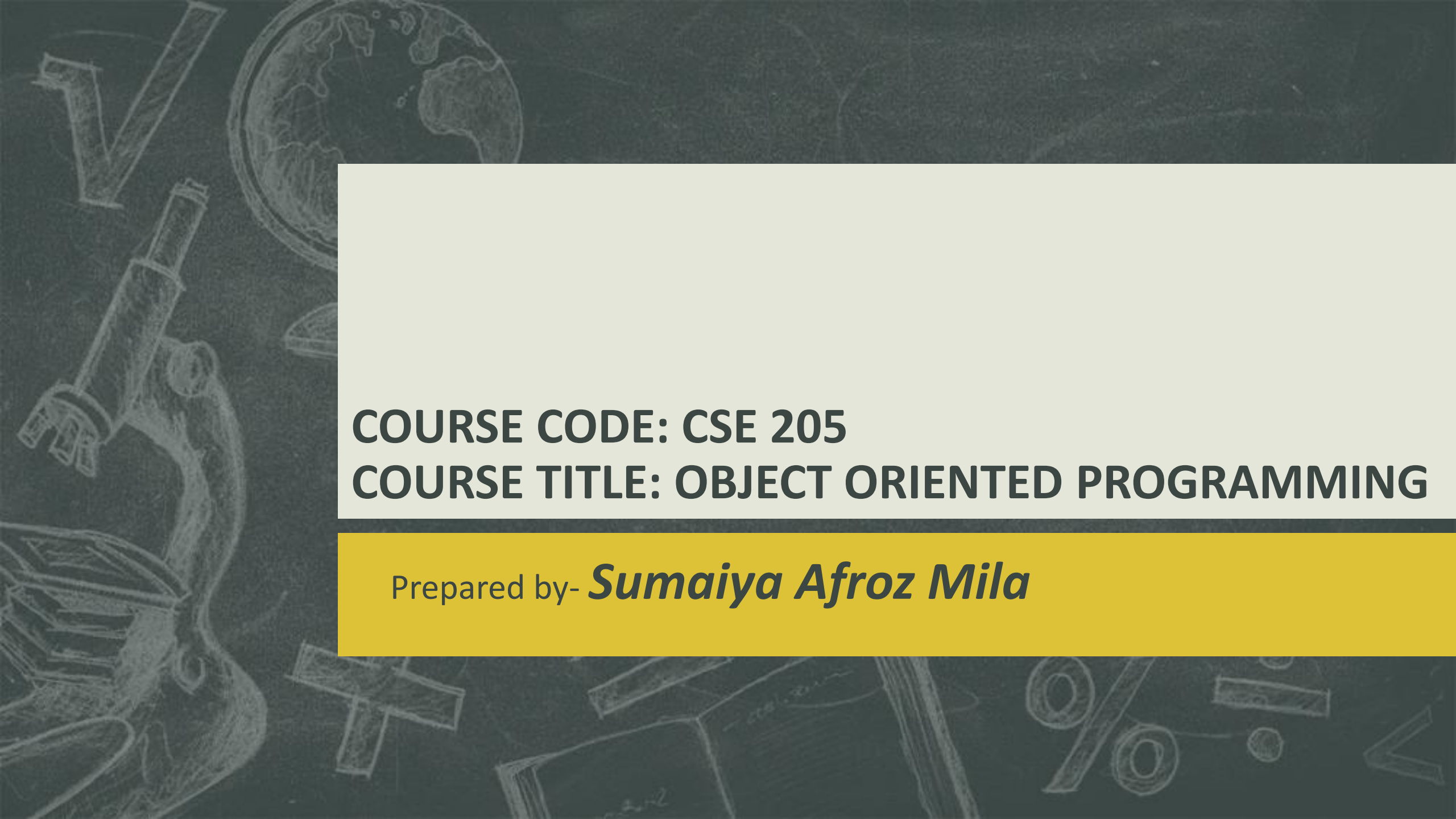
- Write a C++ program to create a student class.
- Your class will contain the variables - ID, Name, cgpa (**Now make the variables private**)
- Create a member function named “Setter()” to set the values of the variables. (**This function will be public**)
- Create a member function named “Display()” to print the values of the variables. (**This function will be public**)
- Create an object of the student class in the main function.
- Call the setter() function to set the values of the variables.
- Call the display function and print the values of the variable

Design a Point Class

- Design a Point class which will contain the variables X, Y(private)
- Create a function “SetX()” to set the value of X.
- Create a function “SetY()” to set the value of Y.
- Create a function “Distance (int X2, int Y2)” which will calculate and show the distance between two points (X,Y) and(X2,Y2).
- Formula to calculate distance: d = $\sqrt{(X_2 - X)^2 + (Y_2 - Y)^2}$
- Include the “math.h” to use sqrt() function

```
class Point{
private:
    int X;
    int Y;
public:
    void setX();
    void setY();
    void Distance(int X2, int Y2);
};
int main(){
    Point p1;
}
```


Thank
you!



COURSE CODE: CSE 205
COURSE TITLE: OBJECT ORIENTED PROGRAMMING

Prepared by- ***Sumaiya Afroz Mila***

Procedural Programming

How it works

- Split a program into tasks and subtasks
- Write functions to carry out the tasks
- Instruct computer to execute the functions in a sequence

Disadvantages

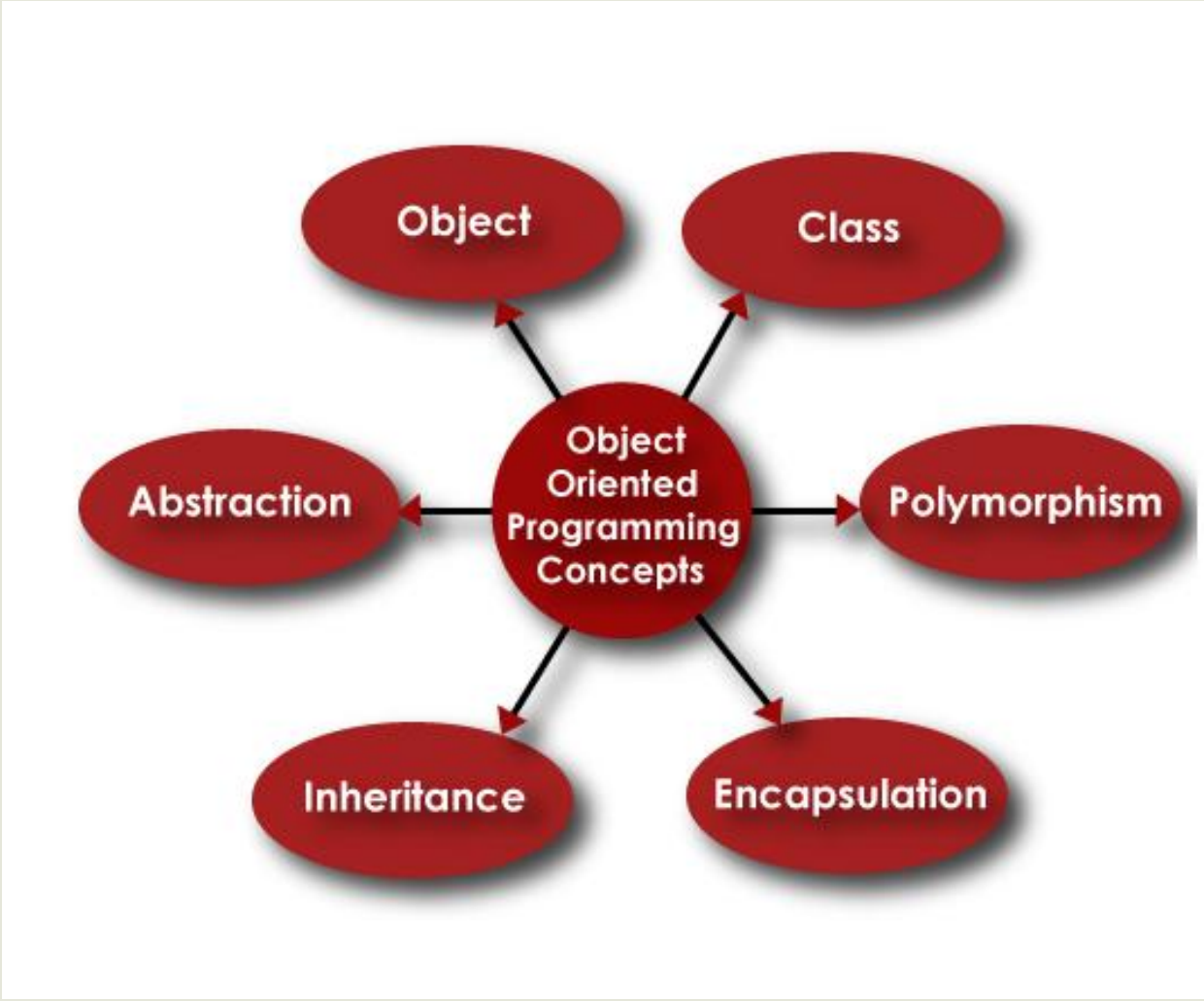
- Large amount of data intricate flow of tasks : complex
- Gives more importance to algorithms & functions than data being used by the functions
- Multi-function programs - shared data items are placed as global
- Changes in data structure affects functions and involves re-writing code

Why OOP?

- OOP treats data as a critical element in the program development.
- Does not allow data to flow freely around the system.
- Supports class/structure creation
- Makes maintenance of large program easy
- Contains following features
    - Encapsulation
    - Polymorphism
    - Inheritance

What is OOP?

Object Oriented programming is a programming style which is associated with the concepts like **class**, object, **Inheritance**, **Encapsulation**, **Abst raction**, **Polymorphism**.



Features of OOP

- Encapsulation
  - Binds together the code and data it manipulates
  - Keeps both safe from accidental interference from outside and misuse
- Polymorphism
  - Allows one name to be used for more than one related but technically different purpose
  - Type of data used to call the function determines which specific version of the function to be executed
  - Function overloading and operator overloading are types of polymorphism
- Inheritance
  - By this process one object can take properties of another

C++ over C

- C++ allows function overloading and operator overloading which reduces redundancy
- Through inheritance, redundant code can be eliminated and extend the use of existing classes
- Principle of data hiding(encapsulation) helps to build a secure program that can not be invaded by code in other parts of the program
- Object oriented systems can be easily upgraded from small to large systems
- Software complexity easily managed

Differences between C & C++

- In C if a function accepts no parameters, then void is mentioned in the parameter list. This is optional in C++

```
void display (void);      // c style
void display () ;         // c++
```

- All funcs must be proto-typed / declared before usage, in C++.
- If a func return type is mentioned it must return a value in C++. In C it is optional.
- In C if return-type of a func is not specified, it is assumed to be int. In C++ it has to be explicitly mentioned. No defaults are allowed
- C++ has an additional bool datatype. C doesnot

C++ Program Structure

- Line 1 & 2 : Preprocessor directives
  - Lines beginning with # are preprocessor commands
  - Include the header file for functions like cin, cout, etc
  - Set the default namespace as standard
- Line 3: Entry point of the program. Begin execution here.
- Line 4-6 : Body of the program

Untitled1.cpp | Untitled1.c

```
1  #include<iostream>
2  using namespace std;
3  int main()
4  {
5      cout<<"Hello CSE-19";
6  }
7
```

C vs C++ Output

```
#include <iostream>
using namespace std;
int main(){
cout<<“hello world”;
}
#include <iostream>
using namespace std;
int main(){
int a;
cin>>a;
}
```

```
#include <stdio.h>
int main(){
Printf(“hello world”);
}

#include <stdio.h>
int main(){
int a;
Printf(“%d”,&a);
}
```

C vs C++ Output Cntd.

C++

```
include<iostream>
main(){
int i;
float j;
char c;
cin>> i >> j >> c;
cout<< i <<“ “<<j<<“ “<<c<<endl;
}
```

C

```
include<stdio.h>
main(){
int i;  float j;  char c;
scanf(“%d%f%c”,&i,&j,&c)
;
printf(“%d %f %c\n”,i,j,c);
}
```

Why Using ‘namespace std’?

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

namespace first_space{
    void printf(char *p){
cout << "Inside first_space" << endl; }
 }

int main(){
   printf("hello world");
   first_space::printf("hello world");
}
```

What will be the output?

```
1   #include<iostream>
2   using namespace std;
3
4   void display()
5   {
6       cout<<"Display Function 01"<<endl;
7
8   }
9   void display()
10  {
11      cout<<"Display Function 02"<<endl;
12
13  }
14  int main()
15  {
16      display();
17  }
```

What will be the output???

What is a namespace?

- Additional information to differentiate similar functions, classes, variables etc. with the same name, available in different libraries.
- Namespace, defines a scope. To call namespace-enabled version of either function or variable, prepend the namespace name to the function / variable :

```
std::cout << "hello"<<std::endl;
std::cin.get();
```

- This can be avoided with preprocessor directives :

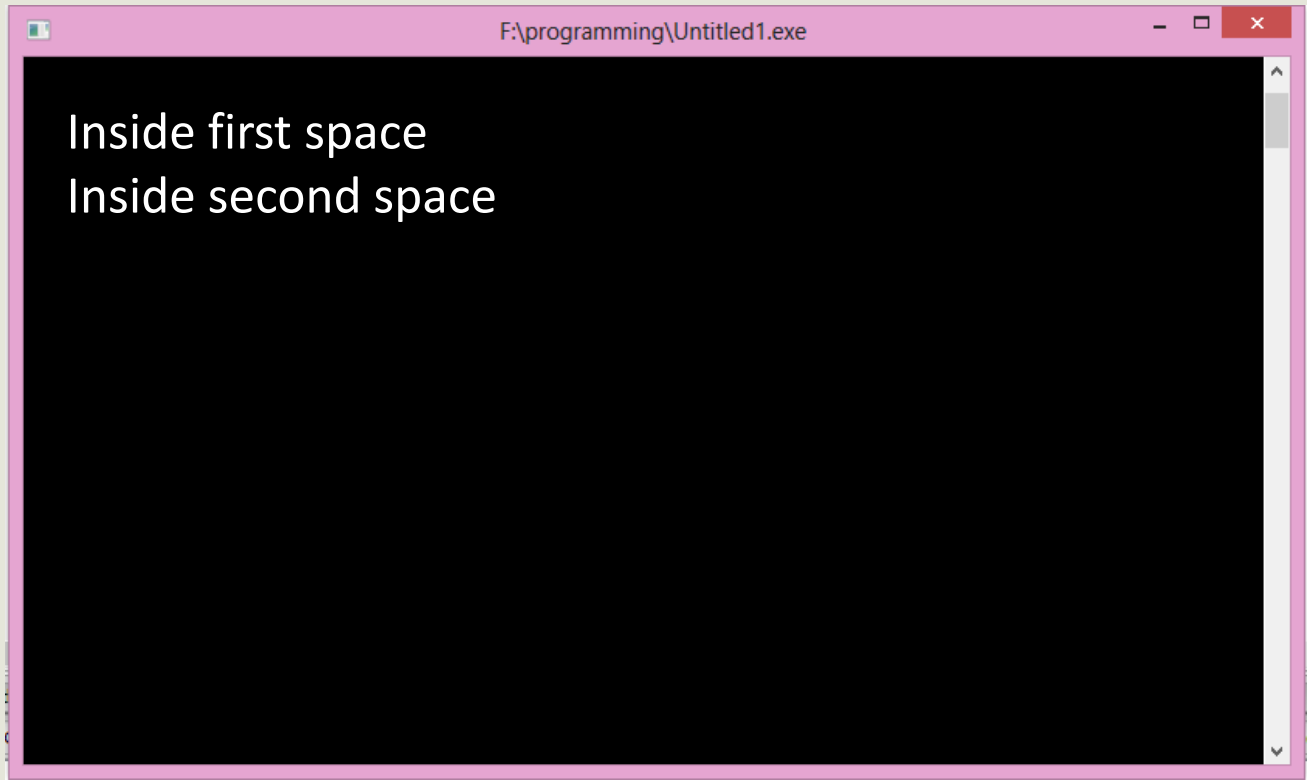
```
using namespace std; // std is abbreviation of namespace standard
cout <<"hello"<<endl;
```

- 3 types of namespace-
  - Standard namespace
  - User defined namespace
  - Anonymous namespace

Using user defined namespace

```
#include <iostream>
using namespace std;
namespace first_space{
    void display(){
cout << "Inside first_space" << endl; }
 }
namespace second_space{
    void display(){
cout << "Inside second_space" << endl; }
 }
int main () {
    // Call function from first name space.
first_space::display();
   // Call function from second name space.
second_space::display();
}
```

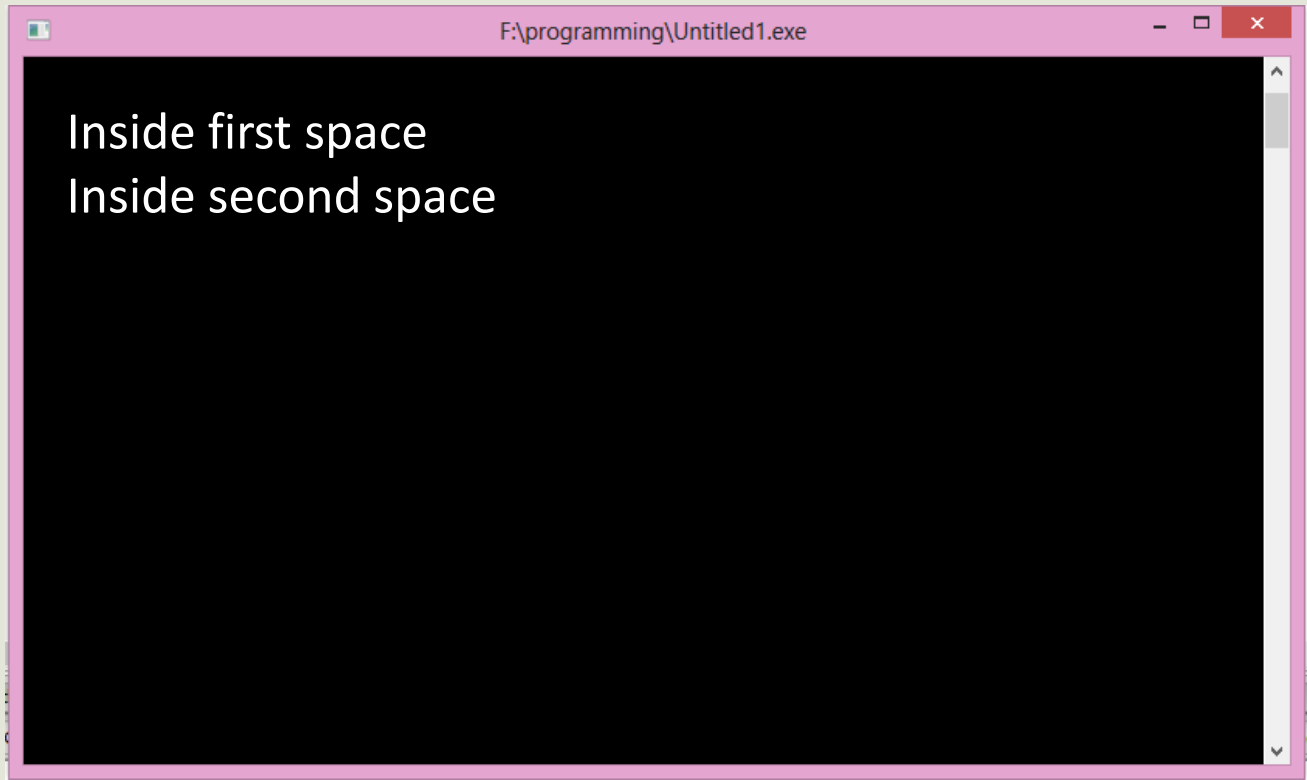
What will be the output?



Using user defined namespace

```
#include <iostream>
using namespace std;
namespace first_space{
    void display(){
cout << “Inside first space”<< endl;
} }
namespace second_space{
    void display(){
cout << “Inside second space”<< endl;
} }
Using namespace first_space ;
int main () {
    // Call function from first name space.
display();
   // Call function from second name space.
second_space::display();
```

What will be the output?



Using user defined namespace

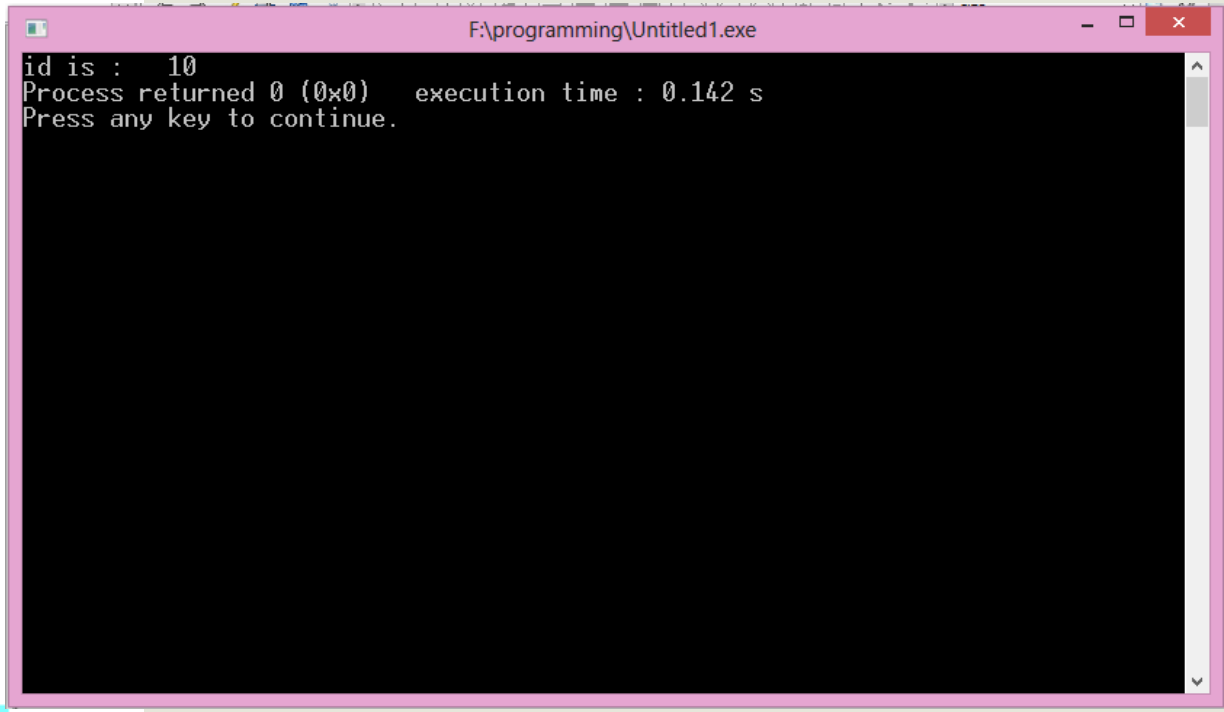
1. **Define two namespaces “SquareSpace” and “CircleSpace” both containing a function with same name “Area”. Use necessary variables.**
2. **Take input of a variable a. Call both functions from main program and show the output. Provide the input as parameter.**
3. **The “Area()” function in the SquareSpace should return the area of a square considering the parameter.**
4. **The “Area()” function in the CircleSpace should return the area of a circle considering the parameter.**

Structure in C

Untitled1.c | Untitled1.cpp

```
1  #include<stdio.h>
2  struct employee{
3      int id;
4      char designation[100];
5      char branch[100];
6  };
7
8  int main(){
9      struct employee emp1;
10     emp1.id = 10;
11     printf("id is :    %d",emp1.id);
12 }
```

What will be the output?



Structure in C

```c
#include<stdio.h>
struct employee{
    int id;
    char designation[100];
    char branch[100];
    int getID(){
        return id;
    }
};
int main(){
    struct employee emp1;
    emp1.id = 10;
    printf("id is :%d",emp1.getID());
}
```

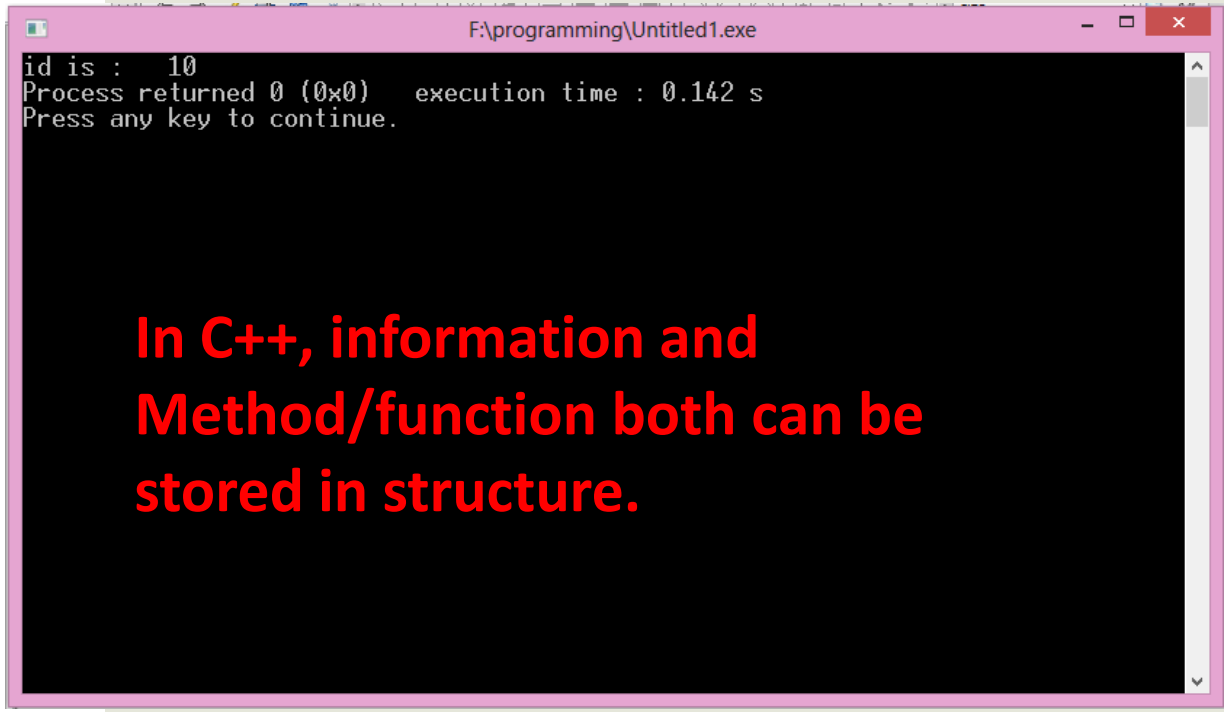
What will happen now?
What will be the output?

There will be compilation error. In C, only information can be stored in structure. Method/function can not be stored.

Structure in C++

```
#include<iostream>
using namespace std;
struct employee{
    int id;
    char designation[100];
    char branch[100];
    int getID(){
        return id;
    }
};
int main(){
    struct employee emp1;
    emp1.id = 10;
    printf("id is :%d",emp1.getID());
}
```

What will happen now?
What will be the output?



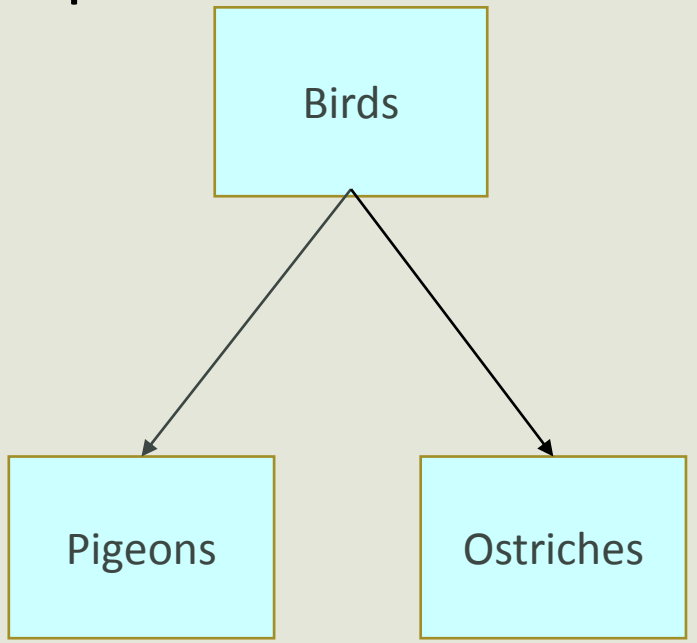
In C++, information and Method/function both can be stored in structure.

Class vs Structure

- Class and structure have identical capabilities
- Both can contain information and methods
- By default members of class are private and members of structure are public.
- Class provides the feature of inheritance and structure does not.

User defined data type: Class

- A class is a collection of method and variables. It is a blueprint that defines the data and behavior of a type.
- Almost all birds have the functionality
  - Have feather
  - Sharp claws
  - Lays eggs
- But birds (pigeons) have some distinct functionality which other birds (ostriches) does not have



- Here we can take Birds as a class. A class is a blueprint for any functional entity which defines its properties and its functions. Like birds having feather, laying eggs.

Objects

- When we say, Birds, Ostriches or Pigeons, we just mean a type/kind
- all the ostriches and pigeons are the forms of these classes. They have a physical existence while a class is just a logical definition. So all the ostriches and pigeons are the objects/instances.

Class and Object more examples

- Assume a **Human** class. Human can talk, walk, eat, see etc.
- Both **Male** and **Female**, performs some common functions, but there are some specifics to both, which is not valid for the other.
- Females can give birth which males can not.
- So **Human**, **Male** and **Female** are classes.
- I am Mila, and I am an **instance** or **object** of **Female** class as well as **Human** class.

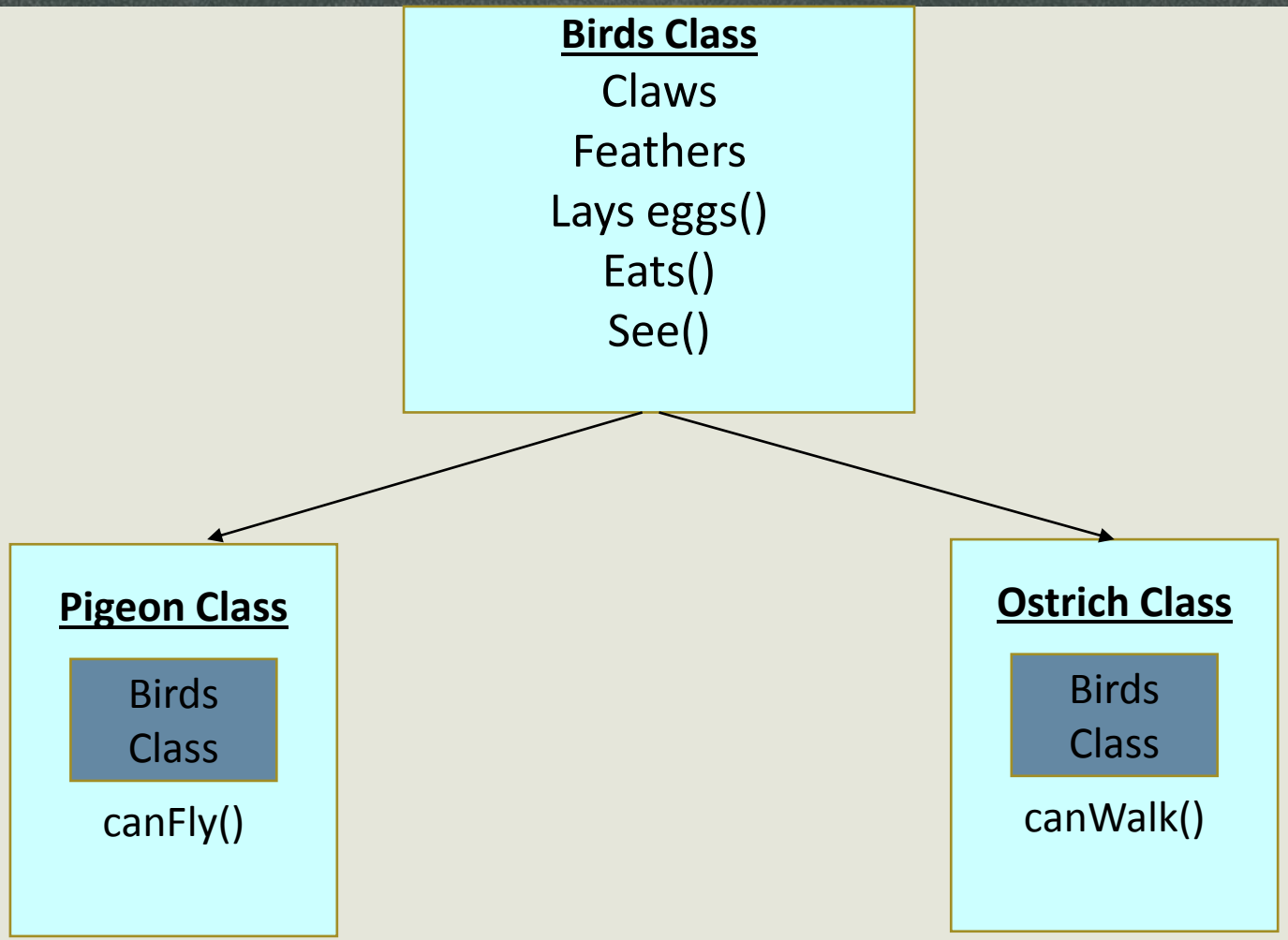
Abstraction

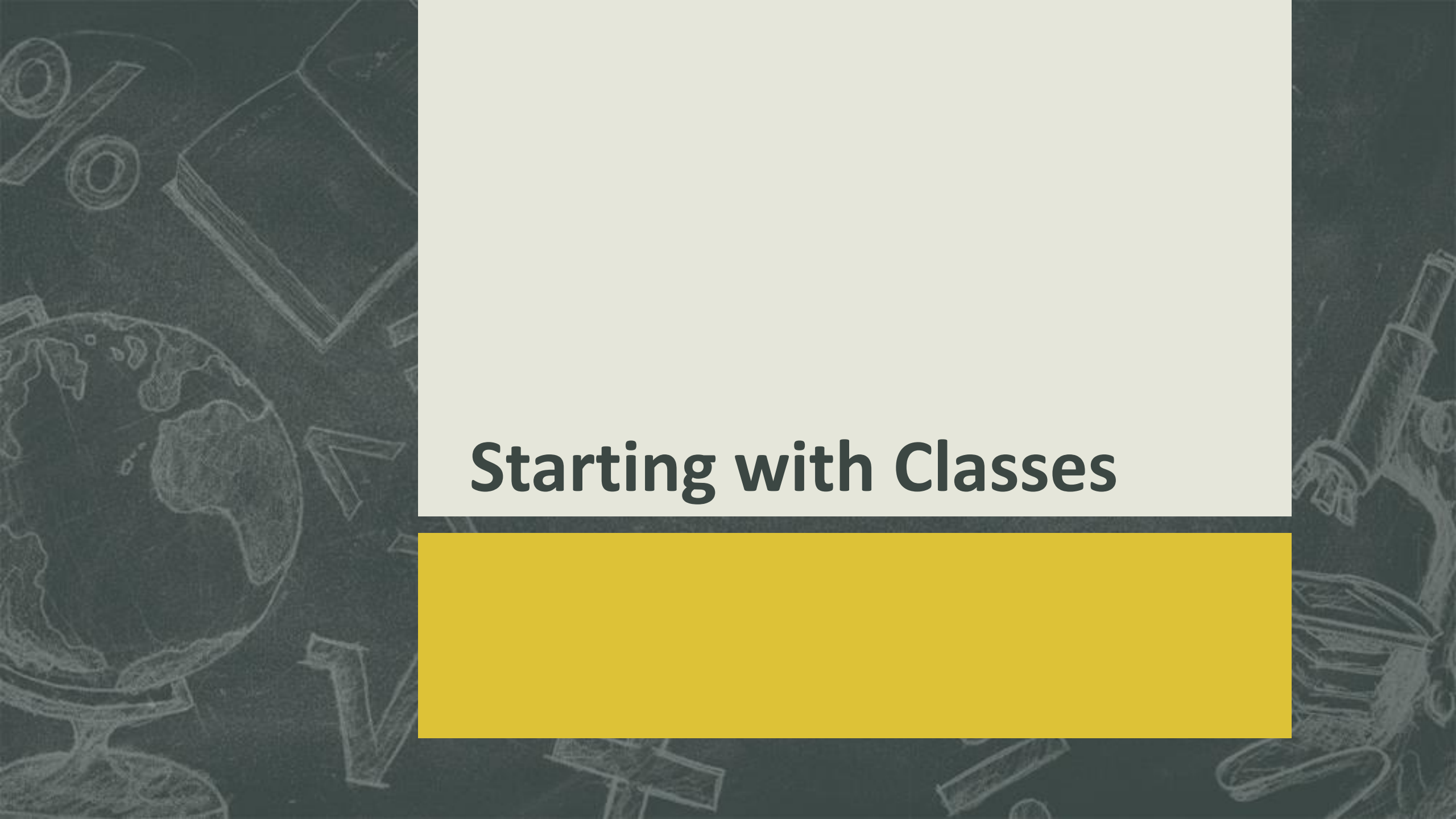
- Abstraction means, showcasing only the required things to the outside world while hiding the details
- Continuing our example, Human can talk, walk, hear, eat, but the details are hidden from the outside world.
- **Difference between abstraction and encapsulation**
  - The most important difference between **Abstraction** and **Encapsulation** is that Abstraction solves the problem at design level while Encapsulation solves it at implementation level.
  - Abstraction is used for hiding the unwanted data and showing the related data. Encapsulation hides the data and code into a single unit to protect the data from outside

Inheritance

- Considering Birds a class, which has properties like claws, feathers, eyes etc, and functions like laying eggs, eat, see etc.
- Chicken and Ostriches are also classes, but most of the properties and functions are included in Birds, hence they can inherit everything from class Birds using the concept of Inheritance.

Inheritance





Starting with Classes

Class

- Class: The building block of C++ that leads to Object Oriented programming is a Class.
- It is a user defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class.
- A class is like a blueprint for an object/instance.

Class

```
struct struct_name{
        //all members
};
main(){
struct struct_name st1,st2;
 }
```

```
class class_name{
//private members
public:
 //public members
 };

main(){
class_name  obj1,obj2;
}
```

Class

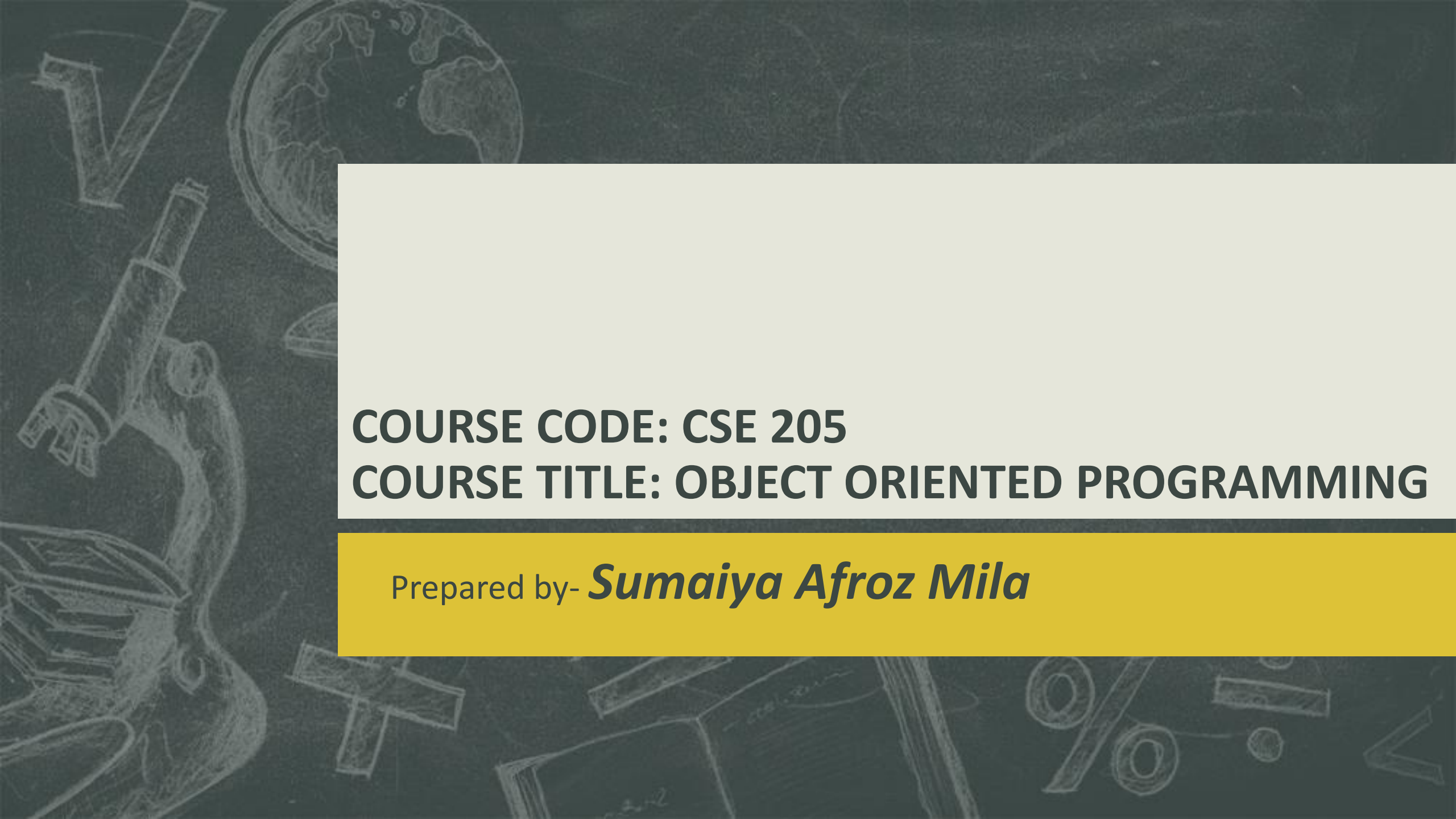
Keyword → **class** **class_name** ← User defined class name

```
{
Access specifier:
Data member;            //variables
Member functions();      //functions to access variables
 };   ← Class name ends with a semicolon

main(){
class_name  obj1,obj2;
}
```

You should also study these topics…

- Differences between C and C++ (teach yourself 1.6)
- What is abstraction and what is encapsulation? Explain with example.(https://www.guru99.com/difference-between-abstraction-and-encapsulation.html)


Thank
you!



COURSE CODE: CSE 205
COURSE TITLE: OBJECT ORIENTED PROGRAMMING

Prepared by- ***Sumaiya Afroz Mila***

Class

- Class: The building block of C++ that leads to Object Oriented programming is a Class.
- It is a user defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class.
- A class is like a blueprint for an object/instance.
- Specifying a class-
  - Class Declaration
  - Class Function Definition – (inside class declaration, outside class declaration)

Class Declaration

- Class declaration starts with the **keyword "class", ** followed by user-defined class name
- Body of the class enclosed within braces and terminated by a semicolon
- Class body contains the **declaration** of **variables** and **functions**. These variables and functions are collectively called **class members**.(Function declaration is same as the function prototype)
- **Variables** declared inside the class are known as **data members** and **functions** are known as **member functions**.

Class Declaration Syntax

Keyword

```
class class_name
{
Access specifier: [we will learn more later]
Data member;            //variables
Function declaration();          //functions to access variable
 };
```

User defined class name

Class name ends with a semicolon

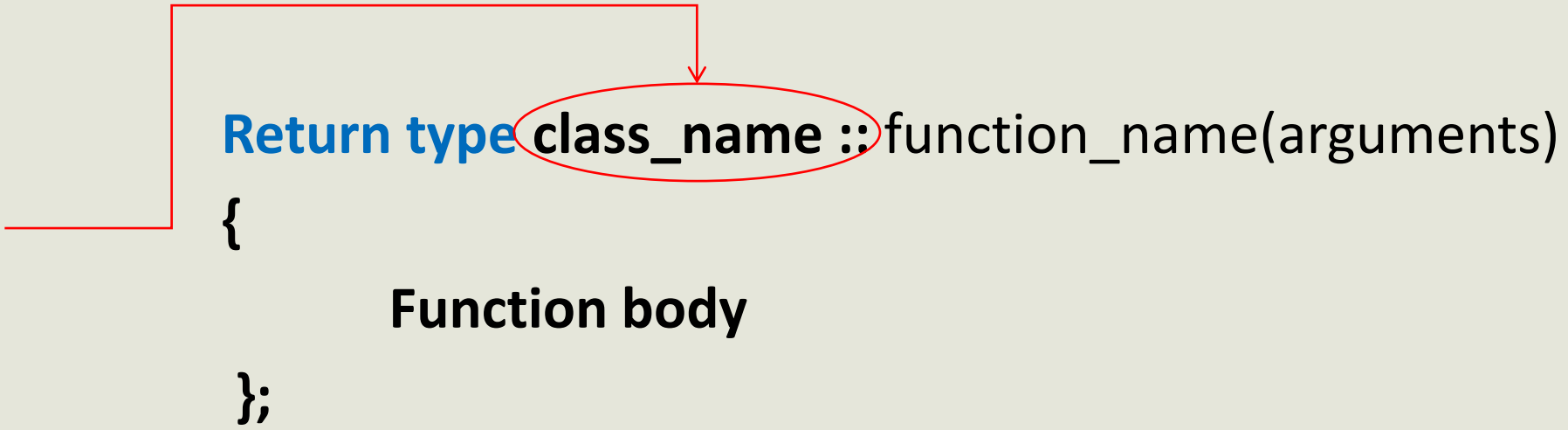
Class definition does not occupy memory

```
main(){
class_name  obj1,obj2;
}
```

Declaration of objects of a class occupies memory

Function Definition – outside class declaration

Membership identity label.
This label tells the compiler, which class this function belongs to.



Example: Function Definition – outside class declaration

```
3   class student{
4   public:
5       int ID;
6       float cgpa;
7
8       void setID();
9       void showID();
10  };
11  void student:: setID(){
12          cin>>ID;
13      }
14  void student:: showID(){
15          cout<<"ID is : "<<ID;
16      }
```

Function should be declared inside the class to be a member function of the class

Member function definition outside the class

Function Definition – inside class declaration

Another method of defining member function is to replace the function declaration by the actual function definition inside the class.

```
3  class student{
4  public:
5      int ID;
6      float cgpa;
7
8      void setID(){
9          cin>>ID;
10     }
11     void showID(){
12         cout<<"ID is : "<<ID;
13     }
14     void showcgpa(){
15     }
16 };
```

Let’s write our own ‘student class’

```
3  class student{
4  public:
5      int ID;
6      float cgpa;
7
8      void setID(){
9          cin>>ID;
10     }
11     void showID(){
12         cout<<"ID is : "<<ID;
13     }
14     void showcgpa(){
15     }
16 };
```

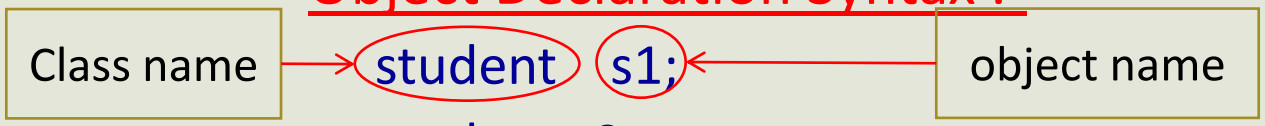
```
17 int main(){
18     student s1;
19     s1.ID = 21;
20     s1.showID();
21 }
```

This is called ‘class declaration’. Class declaration is not a physical entity. This is just a logical representation of a student. So does not occupy memory

Creating objects of a class means creating a physical entity for that class. So creating object occupies memory

Accessing members of the class

- Objects : Instances / occurrence of the class
- Object Declaration Syntax :-




```
student s2;
```

- Each instance has its own set of class variables
- Member functions are only maintained as one copy
- Accessing member variables of objects :-
  - Using object name and dot operator :

```
s1.ID= 21;
```

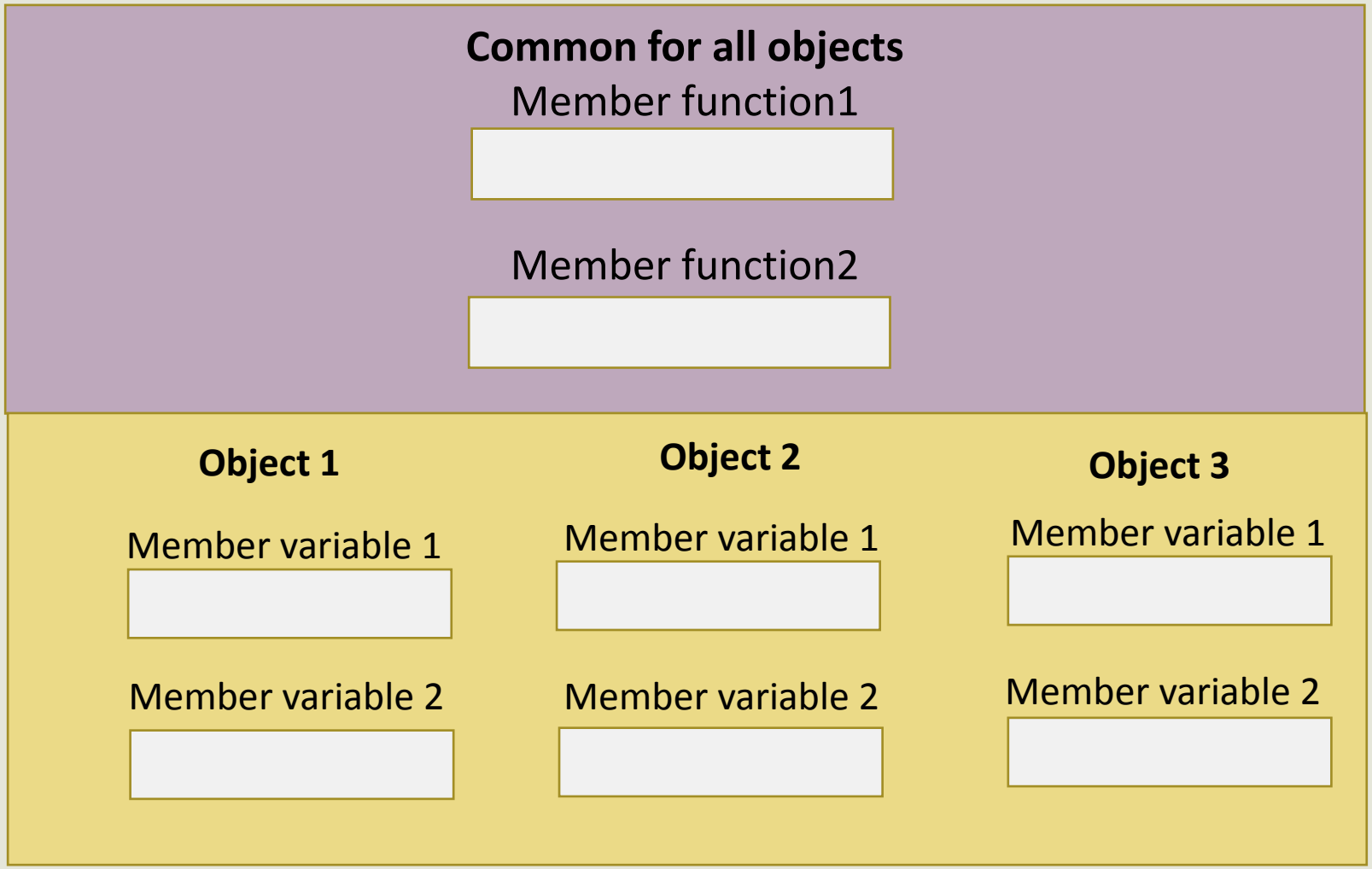
  - Using member functions.

```
s1.showID();
```

Memory allocation for class variables

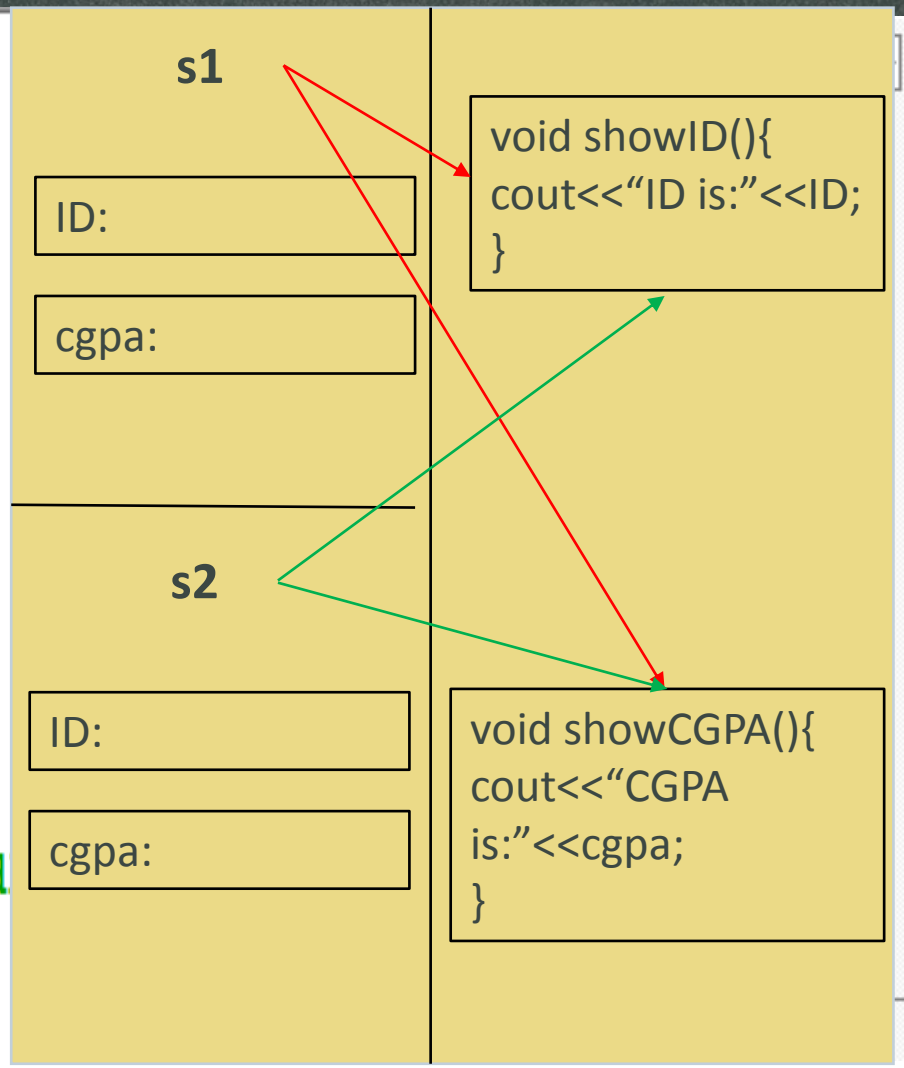
- **The member functions are created and placed in the memory space only once when they are defined as a part of class declaration**
- **As all the objects of that class use the same member function, no separate space is allocated for func when objects are created.**

- **For each object, space for member variables is allocated separately.**
- **Separate memory locations for the objects are essential, as the member variables will hold different values for different objects.**



Let’s write our own ‘student class’

```
#include<iostream>
using namespace std;
class student{
public:
    int ID;
    float cgpa;

    void showID(){
        cout<<"ID is : "<<ID<<endl;
    }
    void showCGPA(){
        cout<<"CGPA is : "<<cgpa<<end
    }
};
```




```
int main(){
    student s1;
    student s2;

    s1.ID=21;
    s2.ID=25;
    s1.cgpa=2.5;
    s2.cgpa=3.5;

    s1.showID();
    s2.showID();
    s1.showCGPA();
    s2.showCGPA();
}
```

Overview: Access Specifiers

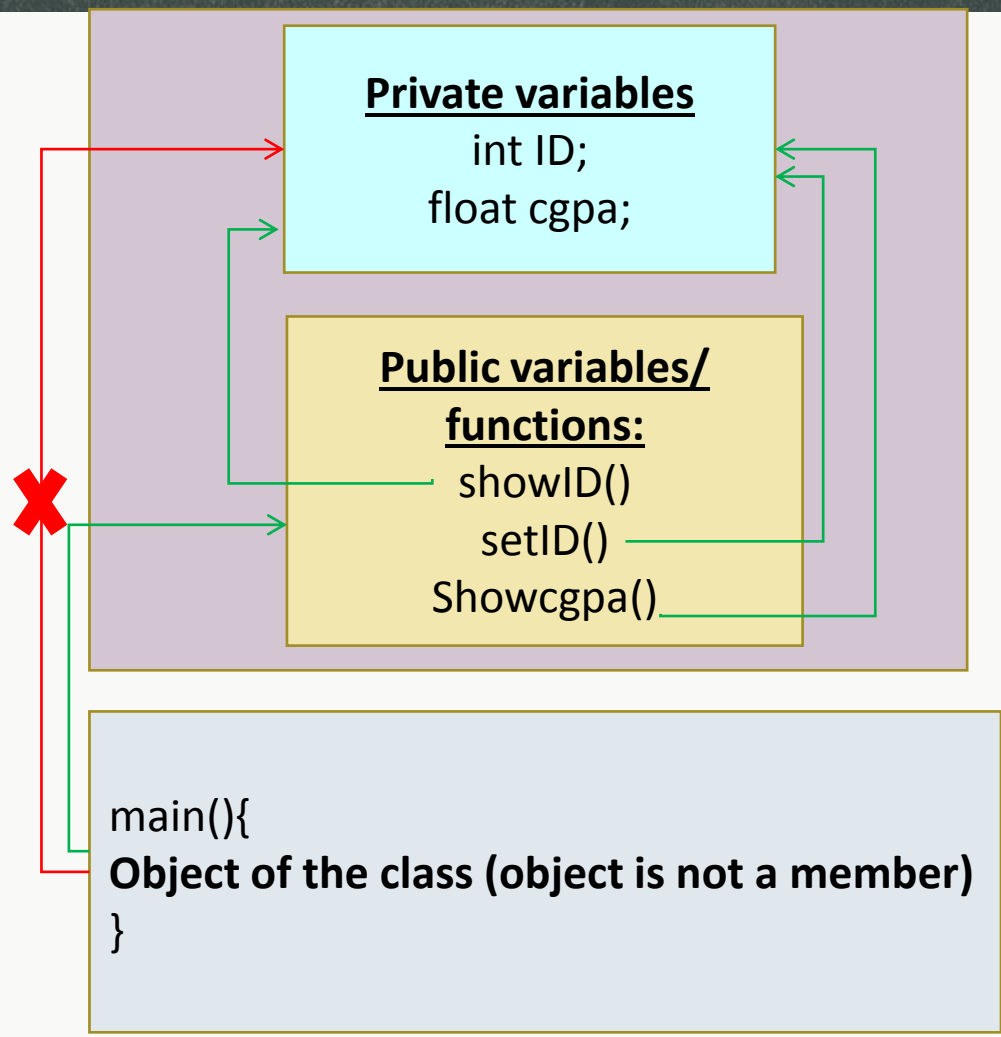
- There are three levels of access specifiers-
  - **Public** - can be accessed from everywhere of the code
  - **Private** - only accessible by the class members
  - **Protected** - similar to private. These members can not be accessed from everywhere of the code. But during inheritance derived class can access protected members. (You will learn about this more in inheritance part)

Visualizing Access Specifiers

```
3  class student{
4  private:
5      int ID;
6      float cgpa;
7  public:
8      void setID(){
9          cin>>ID;
10     }
11     void showID(){
12         cout<<ID;
13     }
14     void showcgpa(){
15     }
16 };
17 int main(){
18     student s1;
19     s1.setID();
20     s1.showID();
21 }
```

- Private members are only accessible by the members of the class.
- Public members are accessible from anywhere

- **Green** arrows mean access **allowed**
- **Red** arrows mean access is **not allowed**



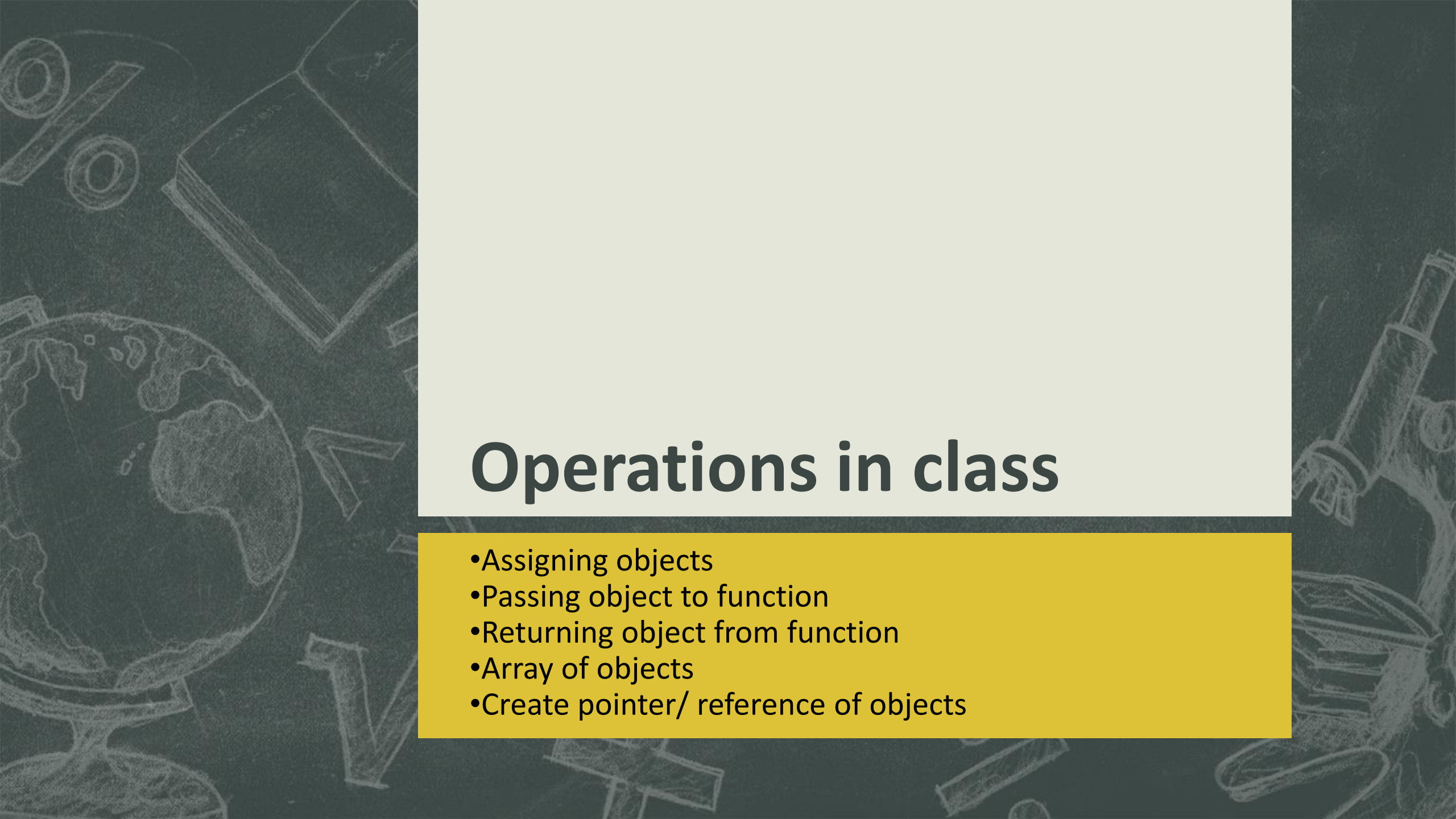
Detailed about Access Specifiers

- There are three levels of access specifiers-
  - Private
    - Usage - Default / ***private:***
    - Private vars accessible only by other member functions of the class & *friend functions*
    - *Private functions can't be called using objects !*
    - *Private member vars are not accessible from child class objects!*
  - Public
    - Usage – ***public :***
    - Accessible by other members and by any other part of the program that contain the class
    - Can set and get the value of public variables without any member function
  - Protected
    - Usage – ***protected :***
    - Protected vars can only be accessed using member functions of the class.
    - *Protected functions can't be called using objects !*
    - Inherited protected vars can be accessed using public functions in child / derived classes.


Thank
you!

COURSE CODE: CSE 205
COURSE TITLE: OBJECT ORIENTED PROGRAMMING

Prepared by- ***Sumaiya Afroz Mila***



Operations in class

- Assigning objects
- Passing object to function
- Returning object from function
- Array of objects
- Create pointer/ reference of objects

Assigning Objects

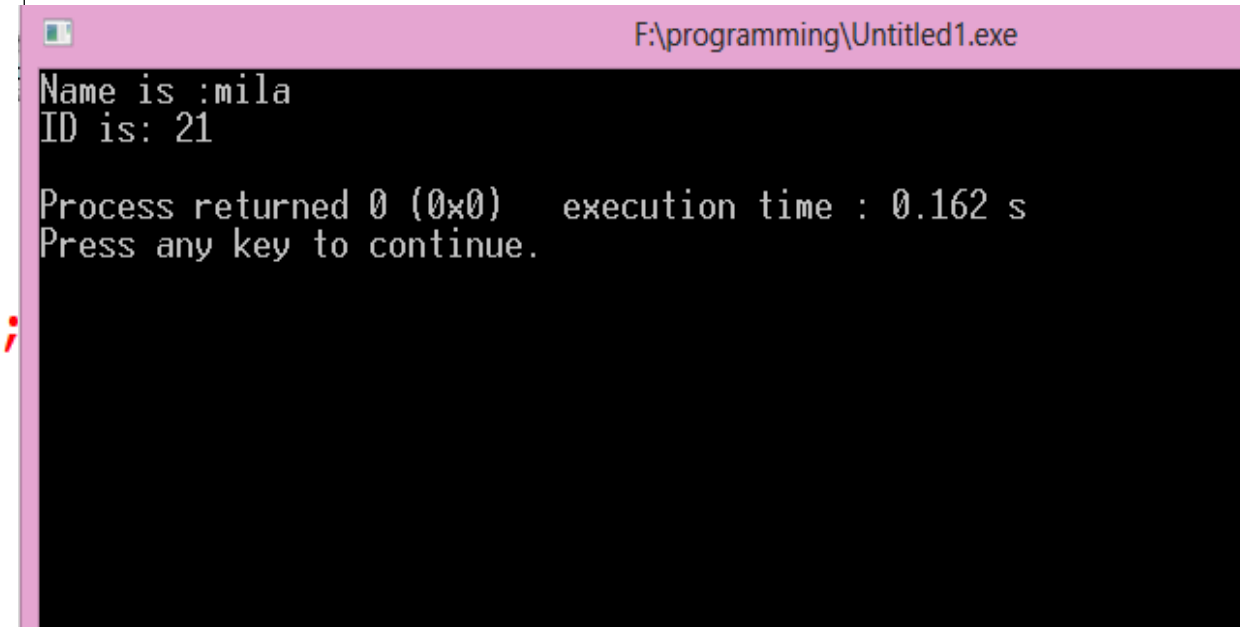
- One object can be assigned to another if both objects are of the same type (both are objects of the same class)
- When one object assigned to another, bitwise copy of all the members is made.

Assigning Objects Example

```
class student{
    char *name;
    int id;
public:
    void setValues(char *n, int i){
        name = n;
        id = i;
    }
    void showValues(){
        cout<<"Name is :"<<name<<endl;
        cout<<"ID is: "<<id<<endl;
    }
};
```

```
int main(){
    student ob1,ob2;
    ob1.setValues("mila", 21);
    ob1.showValues();
```

What will be the output??

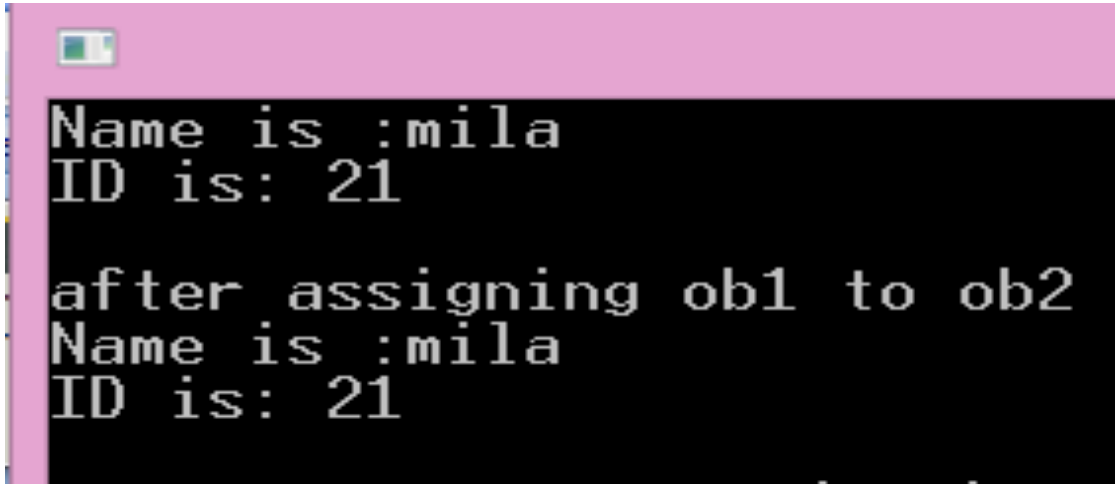


Assigning Objects Example

```
class student{
    char *name;
    int id;
public:
    void setValues(char *n, int i){
        name = n;
        id = i;
    }
    void showValues(){
        cout<<"Name is :"<<name<<endl;
        cout<<"ID is: "<<id<<endl;
    }
};
```

OUTPUT???

```
int main(){
    student ob1,ob2;
    ob1.setValues("mila", 21);
    ob1.showValues();

    ob2 = ob1;
cout<<"after assigning ob1 to ob2";
    ob2.showValues();
}
```



Assigning Objects Example

```
class student{
    char *name;
public:
    void setValues(char *n){
        name = n;
    }
    void showValues(){
        cout<<"Name is :"<<name;
        cout<<endl;
    }
};
```

```
class teacher{
    char *name;
public:
    void setValues(char *n){
        name = n;
    }
    void showValues(){
        cout<<"Name is :"<<name;
        cout<<endl;
    }
};
```

```
int main(){
    student ob1;
    teacher ob2;
    ob1.setValues("mila");
    ob1.showValues();
}
```

Output??

Name is: mila

Assigning Objects Example

```
class student{
    char *name;
public:
    void setValues(char *n){
        name = n;
    }
    void showValues(){
        cout<<"Name is :"<<name;
        cout<<endl;
    }
};
```

```
class teacher{
    char *name;
public:
    void setValues(char *n){
        name = n;
    }
    void showValues(){
        cout<<"Name is :"<<nam
        cout<<endl;
    }
};
```

ERROR

Assignment not allowed as objects are not of the same class

```
int main(){
    student ob1;
    teacher ob2;
    ob1.setValues("mila");
    ob1.showValues();

    ob2 = ob1;
cout<<endl;
cout<<"after assigning ob1
to ob2";
cout<<endl;
    ob2.showValues();
}
```

Passing Objects to a function

- Objects can be passed to functions as argument in the same way other types of data are passed.
- Simply declare the functions parameter as a class type
- Use an object of that class as an argument when calling the function
- As with other types of data, by default all objects are passed by value to a function.
  - Means a bitwise copy of the argument is made
  - This copy is used by the function.
  - Changes to the object inside the function do not affect the calling object.

Syntax

```
void func(student ob){
 // function body
}

int main(){
Student ob;
func(ob);
}
```

Passing Objects to a function Example

- Design a Point class which will contain the variables X, Y(private)
- Create a function "SetXY(int a, int b)" to set the value of X and Y.
- Create an object p1 of the class. Set the values.
- Create a function "Distance (Point p2)" which will calculate and show the distance between two points p1 and p2.
- Formula to calculate distance: d = $\sqrt{(X_2 - X)^2 + (Y_2 - Y)^2}$
- Include the "math.h" to use sqrt() function

```
class Point{
private:
    int X;
    int Y;
public:
    void setXY(int a, int b);
    void Distance(Point p2);
};
int main(){
    Point p1,p2;
    p1.setXY(0,0);
    p2.setXY(4,0);
    p1.Distance(p2);
}
```

Returning Objects from a function

- Objects can be returned from functions in the same way other types of data are returned.
- Simply declare the function as returning a class type
- Return an object from that type using return statement
- When an object is returned by a func, a temporary object is automatically created which holds the return value.
- After returning the value, this object is destroyed

Syntax

```
student func(){
        student ob1;
        return ob1;
}
int main(){
Student ob2;
ob2 = func();
}
```

Returning Objects from a function Example

- Design a Point class which will contain the variables X, Y(private)
- Create a function “SetXY(int a, int b)” to set the value of X and Y.
- Create a “showXY()” function to show the values of the coordinates
- Create an object p1 of the class. Set the values.
- Create a function “CalcMidPoint(Point p2)” which will calculate and return the midpoint between two points p1 and p2.
- Formula to calculate midpoint:
  - x = (x1+x2)/2 , y = (y1+y2)/2

```
class Point{
private:
    int X;
    int Y;
public:
    void setXY(float a,float b);
    Point CalcMidPoint(Point p2);
    void showXY();
};
int main(){
    Point p1,p2,p3;
    p1.setXY(0,0);
    p2.setXY(6,4);
    p3 = p1.CalcMidPoint(p2);
    p3.showXY();
}
```

Array of Objects

- Objects can be arrayed.
- Syntax for declaring an array of objects is exactly same as declaring an array of other type of variables
- Arrays of objects are accessed in the same way as accessing arrays of other types of variables

Syntax

Declaration:

```
Class_name  object_name[size];
```

Accessing:

```
object_name[index].member_variable;
object_name[index].member_function();
```

Rewrite this Code declaring array of objects

```
class student{
    char *name;
    int id;
public:
    void setValues();
    void showValues();
};
void student::setValues(){
    name = (char *)malloc(10*sizeof(char));
    cin>>name;
    cin>>id;
}
void student::showValues(){
    cout<<"Name is: "<<name<<endl;
    cout<<"ID is: "<<id<<endl;
}
```

```
int main(){
        student ob1,ob2;

        ob1.setValues();
        ob2.setValues();

        ob1.showValues();
        ob2.showValues();
}
```

Rewrite this Code declaring array of objects

```
class student{
    char *name;
    int id;
public:
    void setValues();
    void showValues();
};
void student::setValues(){
    name = (char *)malloc(10*sizeof(char));
    cin>>name;
    cin>>id;
}
void student::showValues(){
    cout<<"Name is: "<<name<<endl;
    cout<<"ID is: "<<id<<endl;
}
```

```
int main(){
        student ob[2];

        for(int i=0; i<2;i++){
            ob[i].setValues();
        }
        for(int i=0; i<2;i++){
            ob[i].showValues();
        }
}
```

Using pointer to objects

- Objects can be accessed via pointers.
- When a pointer to an object is used, the **object's members** are referenced using the **arrow(->) operator** instead of the **dot(.) operator.**
- When an object pointer is incremented , it points to the next object. When an object pointer is decremented, it points to the previous object.

Using pointer to objects Example 1

```
class student{
    char *name;
    int id;
public:
    void setValues();
    void showValues();
};
void student::setValues(){
    name = (char *)malloc(10*sizeof(char));
    cin>>name;
    cin>>id;
}
void student::showValues(){
    cout<<"Name is: "<<name<<endl;
    cout<<"ID is: "<<id<<endl;
}
```

```
int main(){
        student ob1    ;

        ob1.setValues();


        ob1.showValues();

}
```

Using pointer to objects Example 1

```
class student{
    char *name;
    int id;
public:
    void setValues();
    void showValues();
};
void student::setValues(){
    name = (char *)malloc(10*sizeof(char));
    cin>>name;
    cin>>id;
}
void student::showValues(){
    cout<<"Name is: "<<name<<endl;
    cout<<"ID is: "<<id<<endl;
}
```

```
int main(){
        student ob1;
        student *p;

        p=&ob1;
        p->setValues();
        p->showValues();

}
```

Using pointer to objects Example 2

```
class student{
    char *name;
    int id;
public:
    void setValues();
    void showValues();
};
void student::setValues(){
    name = (char *)malloc(10*sizeof(char));
    cin>>name;
    cin>>id;
}
void student::showValues(){
    cout<<"Name is: "<<name<<endl;
    cout<<"ID is: "<<id<<endl;
}
```

```
int main(){
        student ob[2];

        for(int i=0; i<2;i++){
            ob[i].setValues();
        }
        for(int i=0; i<2;i++){
            ob[i].showValues();
        }
}
```

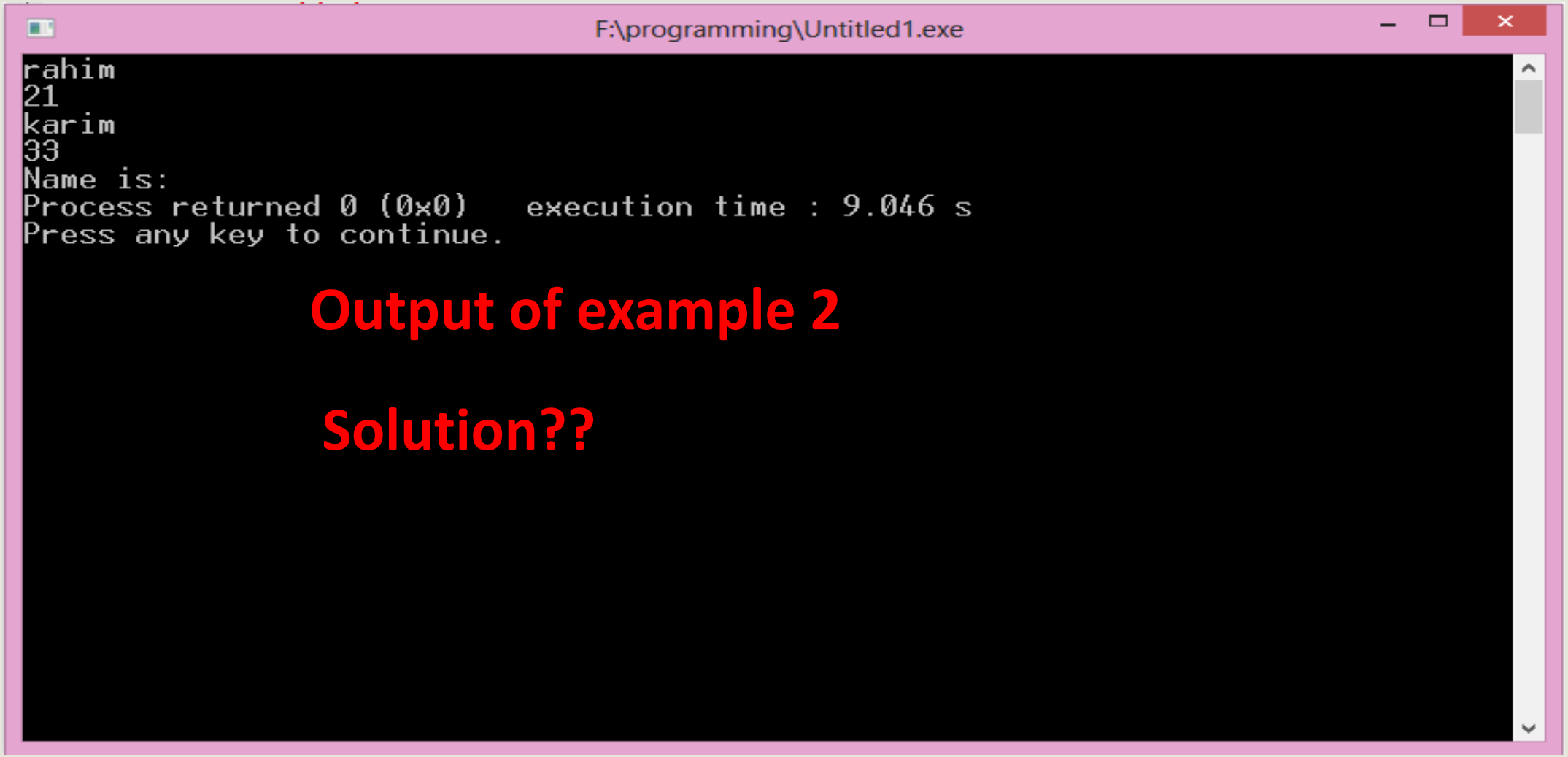
Using pointer to objects Example 2

```
class student{
    char *name;
    int id;
public:
    void setValues();
    void showValues();
};
void student::setValues(){
    name = (char *)malloc(10*sizeof(char));
    cin>>name;
    cin>>id;
}
void student::showValues(){
    cout<<"Name is: "<<name<<endl;
    cout<<"ID is: "<<id<<endl;
}
```

OUTPUT???

```
int main(){
        student ob[2];
        student *p;

        p=&ob[0];

        for(int i=0; i<2 ; i++){
            p->setValues();
            p++;
        }
        for(int i=0; i<2 ; i++){
            p->showValues();
            p++;
        }

}
```

Using pointer to objects Example 2



Using pointer to objects Example 2

```
class student{
    char *name;
    int id;
public:
    void setValues();
    void showValues();
};
void student::setValues(){
    name = (char *)malloc(10*sizeof(char));
    cin>>name;
    cin>>id;
}
void student::showValues(){
    cout<<"Name is: "<<name<<endl;
    cout<<"ID is: "<<id<<endl;
}
```

```
int main(){
        student ob[2];
        student *p;

        p=&ob[0];

        for(int i=0; i<2 ; i++){
            p->setValues();
            p++;
        }
        p=&ob[0];
        for(int i=0; i<2 ; i++){
            p->showValues();
            p++;
        }

}
```

The "**this**" Pointer

- "**this**" pointer is a special type of pointer.
- Automatically passed to any member function when it is called. It is a pointer to the object that generates the call
- The **this pointer** is an implicit parameter to all member functions.
- Only member functions are passed "this" pointer.
- Friend functions are not passed "this" pointer

Example

Ob.f1();

•Ob is the object that calls the function

•The function f1() is passed a pointer to "ob"

Using “this” pointer to objects Example 1

```
class student{
    char *name;
    int id;
public:
    void setValues();
    void showValues();
};
void student::setValues(){
    this->name = (char *)malloc(10*sizeof(char));
    cin>>this->name;
    cin>>this->id;
}
void student::showValues(){
    cout<<"Name is: "<<this->name<<endl;
    cout<<"ID is: "<<this->id<<endl;
}
```

```
int main(){
        student ob1    ;

        ob1.setValues();


        ob1.showValues();

}
```

Passing Objects to a function Example

- Design a Point class which will contain the variables X, Y(private)
- Create a function "SetXY(int a, int b)" to set the value of X and Y.
- Create an object p1 of the class. Set the values.
- Create a function "Distance (Point p2)" which will calculate and show the distance between two points p1 and p2.
- Formula to calculate distance: d = $\sqrt{(X_2 - X)^2 + (Y_2 - Y)^2}$
- Include the "math.h" to use sqrt() function

```
class Point{
private:
    int X;
    int Y;
public:
    void setXY(int a, int b);
    void Distance(Point p2);
};
int main(){
    Point p1,p2;
    p1.setXY(0,0);
    p2.setXY(4,0);
    p1.Distance(p2);
}
```

Solution of Passing Objects to a function Example

```
class Point{
private:
    int X;
    int Y;
public:
    void setXY(int a, int b);
    void Distance(Point p2);
};
void Point::setXY(int a, int b){
    X=a;
    Y=b;
}
void Point::Distance(Point p2){
float dist = sqrt(((p2.X-X)*(p2.X-X)+(p2.Y-Y)*(p2.Y-Y)));
cout<<"Distance between the two points is: "<<dist;
}
```

Does the "p2.X " and "X" make any confusion??

"this" pointer can be a solution

```
class Point{
private:
    int X;
    int Y;
public:
    void setXY(int a, int b);
    void Distance(Point p2);
};
int main(){
    Point p1,p2;
    p1.setXY(0,0);
    p2.setXY(4,0);
    p1.Distance(p2);
}
```

Solution of Passing Objects to a function Example

```
class Point{
private:
    int X;
    int Y;
public:
    void setXY(int a, int b);
    void Distance(Point p2);
};
void Point::setXY(int a, int b){
    X=a;
    Y=b;
}
void Point::Distance(Point p2){
float dist;
dist=sqrt(((p2.X-this->X)*(p2.X-this->X)+(p2.Y-this->Y)*(p2.Y-this->Y)));
cout<<"Distance between the two points is: "<<dist;
}
```

References

- Can be passed to a function
- Can be returned by a function
- Independent reference can be created

What is a Reference?

- A **reference** variable is an another name for an already existing variable. Once a **reference** is initialized with a variable, either the variable name or the **reference** name may be used to refer to the variable.
- To understand, what a reference parameter is and how it works, let's first start with a program that uses a pointer as parameter.

Pointer vs Reference

```
void f(int *n){
*n=100;
cout<<"value of n:"<<*n<<endl;
}
int main(){
int i=0;
cout<<"before sending the addr to func:"<<i;
cout<<endl;
f(&i);
cout<<"after returning from func:"<<i;
}
```

```
void f(int &n){
n=100;
cout<<"value of n:"<<n<<endl;
}

int main(){
int i=0;
cout<<"before calling the func:"<<i;
cout<<endl;
f(i);
cout<<"after returning from func:"<<i;
cout<<endl;
}
```

Advantages of using a Reference?

- References don’t need dereferencing operator to access the value. They can be used like normal variables. ‘&’ operator is needed only at the time of declaration.
- Members of an object reference can be accessed with dot operator (‘.’), unlike pointers where arrow operator (->) is needed to access members.
- When an object is passed to a func as a reference, no copy is made. This is one way to eliminate the troubles associated with the copy of an argument damaging something needed elsewhere in the program

Self Study(Section 4.4, Teach yourself c++)

- Show general forms for “**new**” and “**delete**”.
- What are some advantages of using them instead of **malloc**() and **free**()?

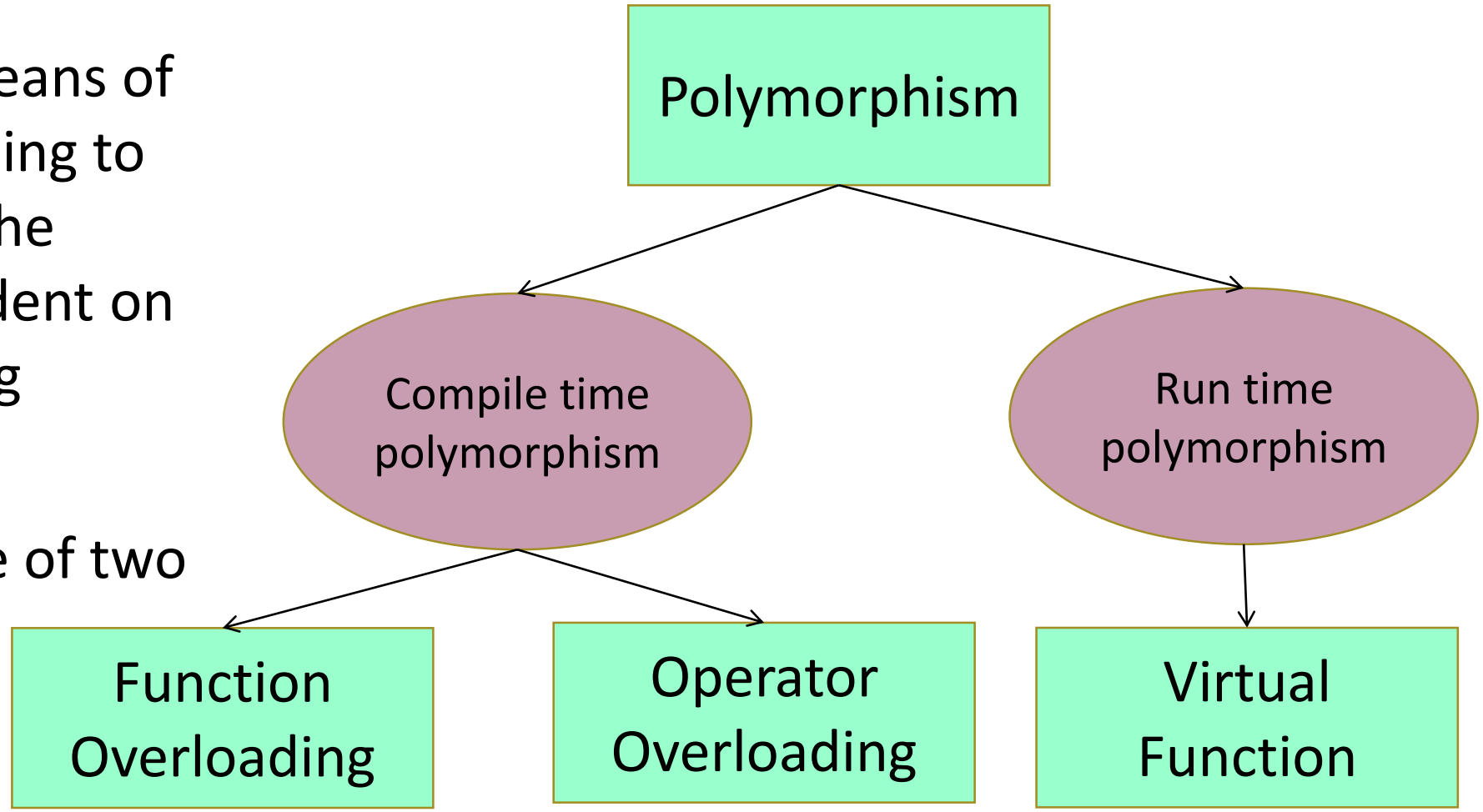

Thank
you!

COURSE CODE: CSE 205
COURSE TITLE: OBJECT ORIENTED PROGRAMMING

Prepared by- ***Sumaiya Afroz Mila***

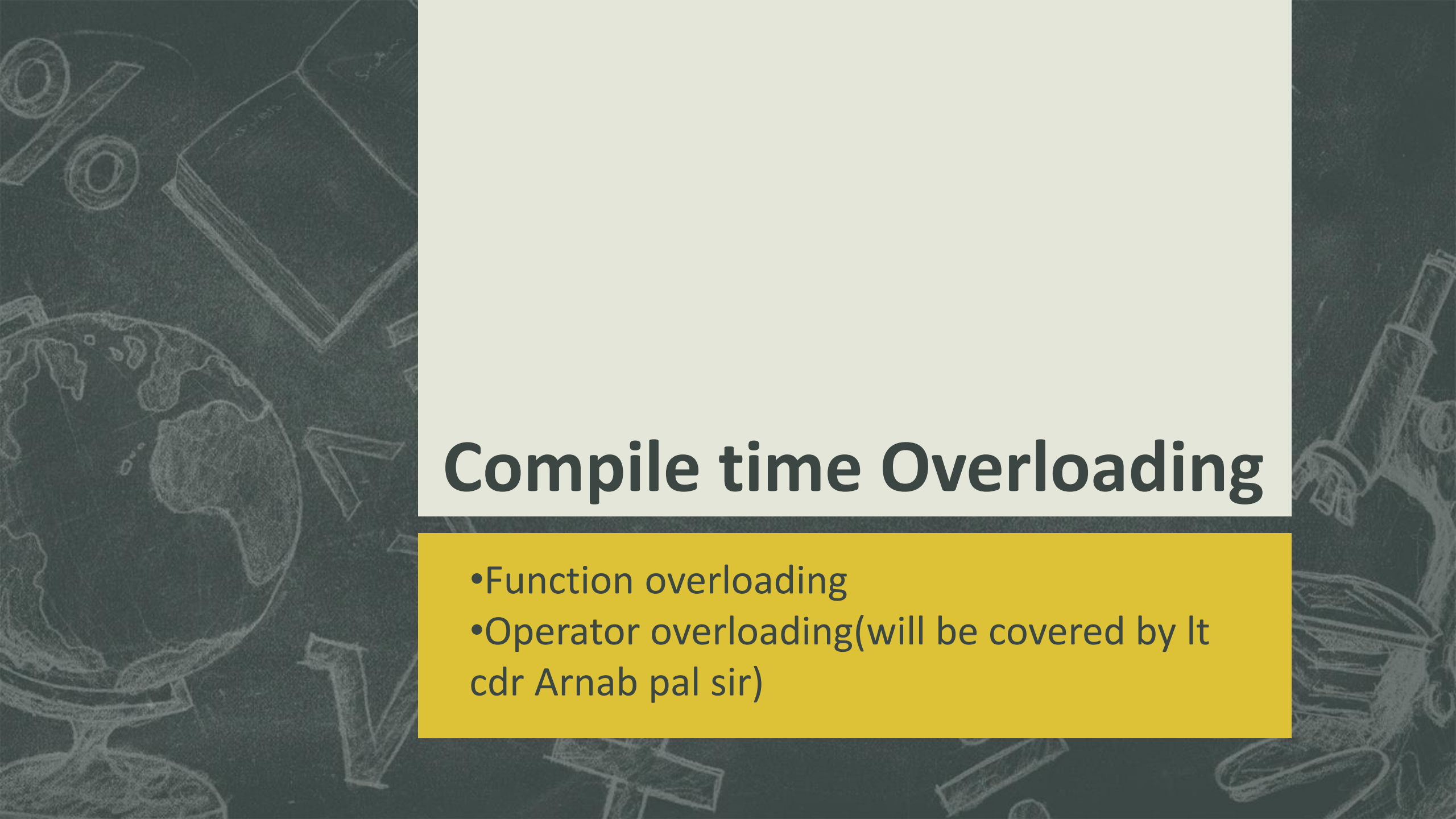
Polymorphism

- Polymorphism is a means of giving different meaning to the same message. The meanings are dependent on the type of data being processed.
- Polymorphism can be of two types-



Polymorphism

- Polymorphism is a means of giving different meaning to the same message. The meanings are dependent on the type of data being processed.
- Polymorphism can be of two types-
- Compile time polymorphism:
  - **Function Overloading**: gives the same name different meaning. The name has several interpretations that depend on function selection.
  - **Operator Overloading**: provides different interpretation of C++ operator depending on type of its argument
- Pure/Run time polymorphism: runtime function selection. Related to inheritance & will be covered in later sessions. Function overriding.



Compile time Overloading

- Function overloading
- Operator overloading(will be covered by lt cdr Arnab pal sir)

Function Overloading

- Functions are either -
  - *member functions—defined inside a class*
  - *free functions—defined in global scope*
- In C, every function has a unique name
- C++ allows several functions with the same name

Early Binding vs Late Binding

- "Early binding refers to events that occur at **compile time**."
- Early binding occurs when all information needed to call a function is known at compile time.
- **Examples :**

  function calls ,overloaded function calls, and overloaded operators.

Early Binding vs Late Binding

- "Dynamic binding occurs at run-time and is also called late-binding."
- "Late binding refers to function calls that are **not resolved until run time**."
- Late binding can make for somewhat **slower** execution times.
- **Example:**

  virtual functions

Function Overloading

C

```
int abs(int i);
float fabs(float f);
long labs(long l);

void main(){
int i=-2;
float f = - 3.4;
long l = -3400l;
Printf("%d",abs(i));
Printf("%f",fabs(f));
Printf("%ld",labs(l));
}
```

C++

```
int abs(int i);
float abs(float f);
long abs(long l);

void main(){
Int i=-2;
float f = - 3.4;
long l = -3400l;
cout<<abs(i);
cout<<abs(f);
cout<<abs(l)
}
```

Overloading mechanism

- Function are differentiated by their names and argument list
- For overloading, Argument list must vary in –
  - Number of arguments
  - Types of arguments
- Only return types can't be used to differentiate functions
  - int func(int i, float f);
  - float func(int i, float f);

Cannot be overloaded

Function Overloading

```
int sum(int a, int b){
        int result = a+b
        return result;
}
float sum(float a, float b){
        float result = a+b
        return result;
}
```

```
void main(){
        int  sumInt = sum(5,6);
        float  sumFloat = sum(4.5,6.0);

        cout<<"Summation of the two
integer numbers : "<<sumInt<<endl;
        cout<<"Summation of the two
float numbers : "<<sumFloat<<endl;
}
```

Function Overloading Example

```
int fun(int a, int b){
        return a + b;
}

float fun(int x, int y) {
        return x – y;
}
```

```
int fun(int a, int b){
        return a + b;
}

float fun(float x, float y) {
        return x – y;
}
```

Overloading Example

```
int totalMark (int phy, int chem){
        int total = phy+chem;
        return total;
}
int totalMark(int phy, int chem, int opt){
        int total = phy+chem+opt;
        return total;
}
void main(){
cout << totalMark (80,90);
cout << totalMark (80,90,80);
}
```

Default Argument

- Must be to the right of any parameter that don’t have default value
- Default can not be specified both in prototype and definition
- Default arguments must be constant or global variables

Default Argument Example

```
int totalMark(int phy, int chem, int opt = 0){
        int total = phy+chem+opt;
        return total;
}

void main(){
 cout << totalMark (80,90);
cout << totalMark (80,90,80);
}
```

Default Argument Restriction

```
int totalMark(int opt = 0, int phy, int chem){
        int total = phy+chem+opt;
        return total;
}
```

ERROR
Must be to the right of any parameter that don't have defaults

---

```
int totalMark( int phy, int chem, int opt = 0);

int totalMark( int phy, int chem, int opt = 0){
        int total = phy+chem+opt;
        return total;
}
```

ERROR
Default can't be specified both in prototype and definition

```
int totalMark( int phy, int chem, int opt = phy){
        int total = phy+chem+opt;
        return total;
}
```

ERROR
Default arguments must be constant or global variables


Thank
you!

COURSE CODE: CSE 205
COURSE TITLE: OBJECT ORIENTED PROGRAMMING

Prepared by- ***Sumaiya Afroz Mila***

Inline Function

How function call works

- Argument of the function is stored in stack
- Control transferred to the func being called.
- CPU then executes the functions code
- Stores the return value in a predefined memory register
- Control is returned to the calling function
- overhead of the function call increases if the function is called a lot of time

```
int odd(int x)
{
        return x%2;
}
int main()
{       int result;
        for(int  i=0; i<=10000; i++){
                result = odd(i) ;
                if(result==1)
                cout <<i<< “  is odd”;
        }
}
```

Solution?

Inline Function - Inline function is a function that is expanded in line when it is called.

- When the inline function is called, whole code of the inline function gets inserted or substituted at the point of inline function call.
- This substitution is performed by the C++ compiler at compile time.

Inline function syntax

```
inline return_type function_name(parameters)
{
        // function code
}
```

Inline Function

```
inline int odd(int x)
{
        return x%2;
}
int main()
{       int result;
        for(int  i=0; i<=10000; i++){
                result = odd(i) ;
                if(result==1)
                cout <<i<< “  is odd”;
        }
}
```

Function statement is copied by the compiler

Advantages

- Function call overhead doesn’t occur. Much faster than normal functions when it is small
- It also saves the overhead of push/pop variables on the stack when function is called.
- It also saves overhead of a return call from a function.
- Increases locality of reference by utilizing instruction cache
- Once inlining is done, compiler can perform intra-procedural optimization if specified.
- Inline function may be useful (if it is small) for embedded systems because inline can yield less code than the function call preamble and return.

Disadvantages

- If Inline functions are too large and called too often, program grows larger and Overhead increases.
- If too many inline functions are used, codes become larger.

Therefore only short functions are declared as inline functions.

Restrictions

- An inline function must be defined before it is first called.
- Inline specifier is a request, not a command.
- If, for various reasons, the compiler is unable to fulfill the request, the function is compiled as a normal function and the inline request is ignored.
- Some more restrictions: functions should not consists of
  - Static variable
  - A loop statement
  - A switch or a goto statement
  - Recursive function

Example 1-Inlining Member Function

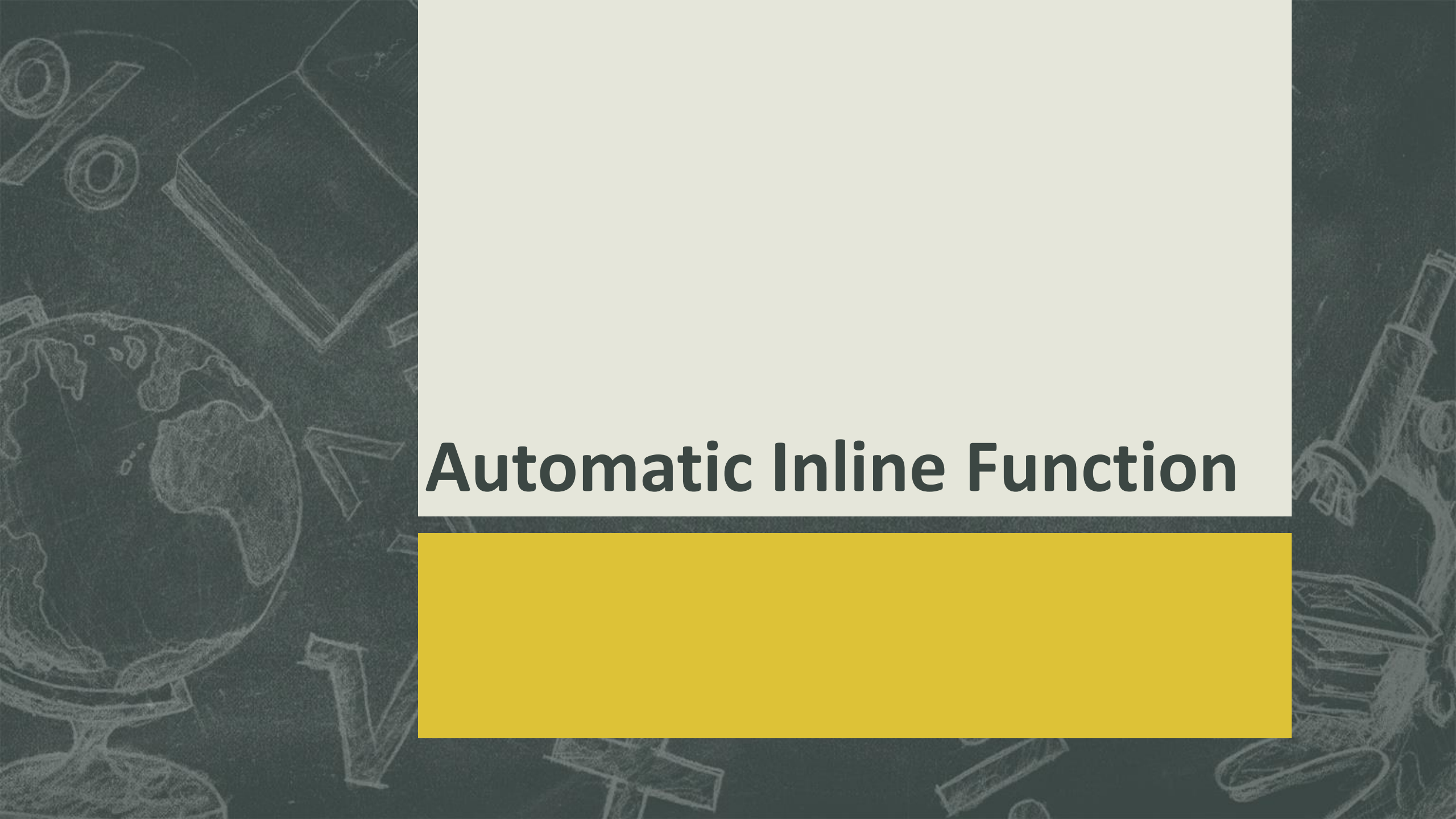
```
class samp
{
        int i, j;
public:
        initialValue (int a, int b);
        int divisible();
};
samp:: initialValue (int a, int b)
{
        i = a;
        j = b;

}
```

```
inline int samp::divisible(){
        return !(i%j);
}
int main()
{
        samp ob1(10, 2), ob2(10, 3);
        ob1. initialValue (10, 2);
        ob2. initialValue (10, 3);
        if(ob1.divisible())
                cout<<“10 is divisible by 2”;
}
```

Example 2

```
inline int min (int a, int b)
{
        return a<b ? a : b;
}
inline long min (long a, long b)
{
        return a<b ? a : b;
}
 inline double min (double a, double b)
{
        return a<b ? a : b;
}
```

```
int main()
{
    cout << min (-10, 10);
    cout << min (-10.01, 100.002);
    cout << min (-10L, 12L);

    return 0;
}
```



Automatic Inline Function

Automatic In-Lining

- If a member function's definition is short enough, the definition can be included inside the class declaration. It causes the function to automatically become an in-line function.

- When a function is defined within a class declaration, the inline keyword is no longer necessary. However, it is not an error to use it in this situation.

Example 1

```
class samp
{
        int i, j;
public:
         initialValue (int a, int b);
        int divisible()
                {
                        return !(i%j);
                }
};
```

```
samp:: initialValue (int a, int b)
{
        i = a;
        j = b;

}
int main(){
        samp ob1;
        ob1. initialValue (10, 2);
        if(ob1.divisible())
                cout<<“10 is divisible by 2”;
}
```

Friend Function

Introduction to Friend Function

- Can private members of a class be accessed from outside the class by a non member function of that class?
- Consider two classes, "**manager**" and "**scientists**". You need a function **income_tax()** to perform some operations on objects of both the classes. What should you do?
  - Write **income_tax()** function in both the classes?
  - Is it redundant?
  - Is there any way to optimize the situation?
  - **"Friend Function"** is the solution

Friend Function

- Friend function is not a member of the class
- It is defined outside of the class
- A friend function can access the private members of the class, it is made friend to
- A friend function is defined exactly the same way as other functions
- Just the function is declared as friend inside the class
- A func can be declared as friends in any number of classes

```
void income_tax(){
//function body
}
```

```
Manager{
        //private members
        //public members
friend void income_tax();
}
```

```
Scientist{
        //private members
        //public members
friend void income_tax();

}
```

Special Characteristics of Friend Functions

- Is not a member of the class to which it is made friend with. Hence friend func is not in the scope of the class.
- As it is not a member, friend func can not be called using the object of that class.
- Unlike member functions, it can not access the members directly. Has to use an object name and the dot operator to access.

```
void income_tax(Manager ob){
        salary = 1000;
        ob.salary = 1000;
}
```

```
Manager{
private:
 double salary;
 friend void income_tax(); //Just declared as friend
                           // not member of the class
}
```

```
main{
        Manager ob;
        ob.income_tax();
        income_tax(ob);
}
```

Special Characteristics of Friend Functions

- It can be declared either in the public or in the private part of the class without affecting its meaning
- Can be friend of more than one class
- Does not have “this” pointer
- Friend function is not inherited.

Friend Function Example

```
#include<iostream>
using namespace std;
class student {
    int dept;
    public:
        void setDept(int d) {dept=d; }

        bool sameDept (student st){
        if (dept==st.dept)
            return true;
        else
            return false;
        }
};

int main(){
  student st1, st2;
  st1.setDept(5);
  st2.setDept(6);
  if (st1.sameDept(st2)==true)
    cout<<"Same";
  else cout<<"Different";
}
```

```
class student {
    int dept;
    public:
        void setDept(int d) {dept=d; }

        bool sameDept (student st){
        if (dept==st.dept)
            return true;
        else
            return false;
        }

        bool TsameDept (teacher t){
        if (dept==t.dept)
            return true;
        else
            return false;
        }
};
class teacher {
    int dept;
    public:
        void setDept(int d) {dept=d; }
};
```

```
int main(){
  student st1; teacher t1;
  st1.setDept(5);
  t1.setDept(6);
  if (st1.TsameDept(t1)==true)
    cout<<"Same";
  else cout<<"Different";
}
```

ERROR

Dept is private in class teacher. Can not be accessed from nonmember of the class

Solution???

Solution 1

```
class student {
    int dept;
    public:
        void setDept(int d) {dept=d; }

        bool sameDept (student st){
        if (dept==st.dept)
            return true;
        else
            return false;
        }

        bool TsameDept (teacher t){
        if (dept==t.getDept())
            return true;
        else
            return false;
        }
};
```

```
class teacher {
    int dept;
    public:
        void setDept(int d) {dept=d; }
        int getDept(){
            return dept;
        }
};
int main(){
  student st1; teacher t1;
  st1.setDept(5);
  t1.setDept(6);
  if (st1.TsameDept(t1)==true)
    cout<<"Same";
  else cout<<"Different";
}
```

Solution 2 using Friend Function

```
class teacher;
class student {
    int dept;
    public:
        void setDept(int d) {dept=d; }

        bool sameDept (student st){
        if (dept==st.dept)
            return true;
        else
            return false;
        }
        friend bool TsameDept (student s,teacher t);
};
class teacher {
    int dept;
    public:
        void setDept(int d) {dept=d; }
         friend bool TsameDept (student s,teacher t);
};
```

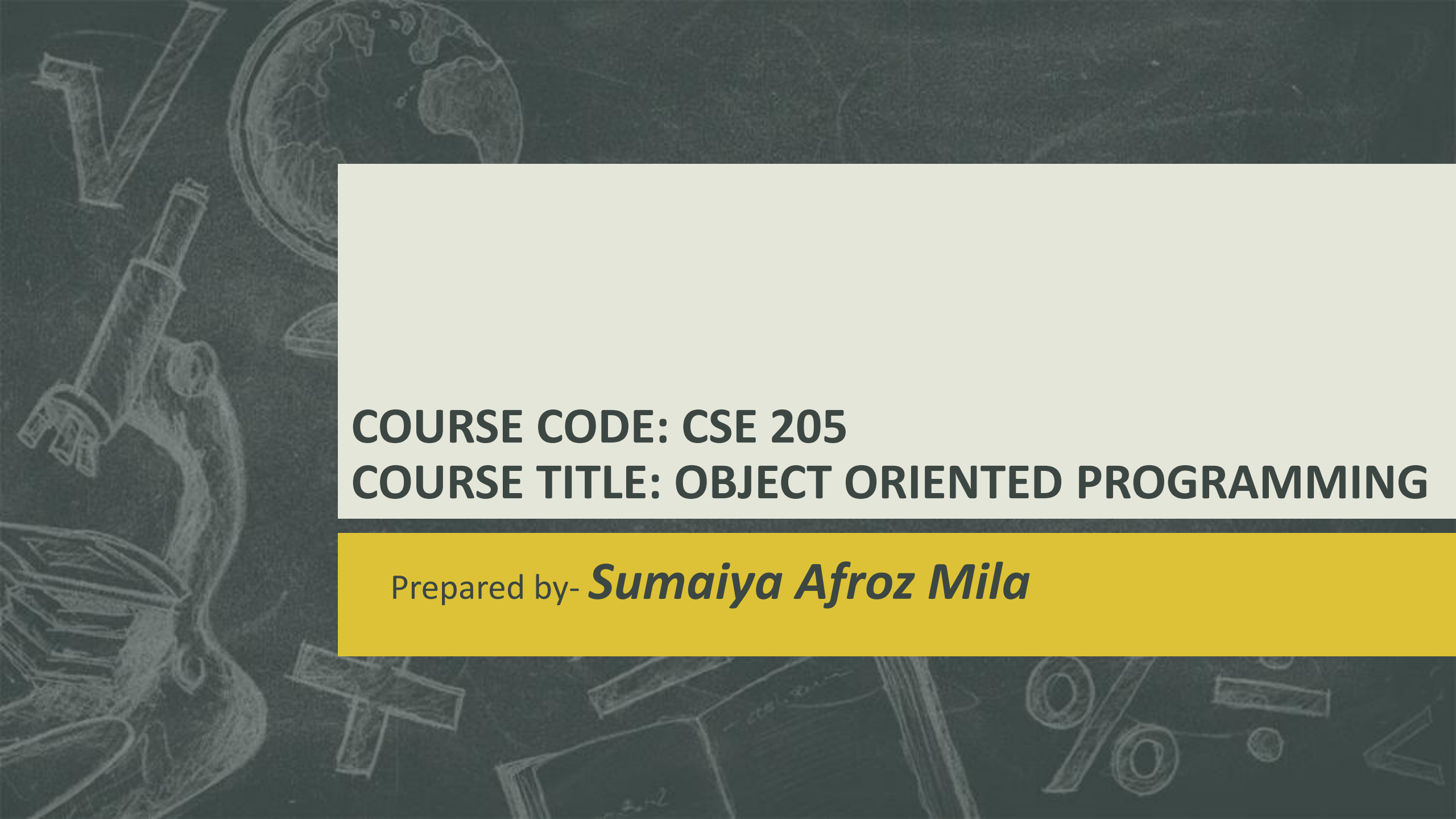
```
bool TsameDept (student s,teacher t){
        if (s.dept==t.dept)
            return true;
        else
            return false;
        }

int main(){
  student st1; teacher t1;
  st1.setDept(5);
  t1.setDept(6);
  if (TsameDept(st1,t1)==true)
    cout<<"Same";
  else cout<<"Different";
}
```

Solution 3 using Friend Function

```
class teacher;
class student {
    int dept;
    public:
        void setDept(int d) {dept=d; }

        bool sameDept (student st){
        if (dept==st.dept)
            return true;
        else
            return false;
        }
        bool TsameDept (teacher t);
};
class teacher {
    int dept;
    public:
        void setDept(int d) {dept=d; }
         friend bool student::TsameDept (teacher t);
};
```

```
bool student::TsameDept (teacher t){
        if (dept==t.dept)
            return true;
        else
            return false;
        }
int main(){
  student st1; teacher t1;
  st1.setDept(5);
  t1.setDept(6);
  if (st1.TsameDept(t1)==true)
    cout<<"Same";
  else cout<<"Different";
}
```


Thank
you!



COURSE CODE: CSE 205
COURSE TITLE: OBJECT ORIENTED PROGRAMMING

Prepared by- ***Sumaiya Afroz Mila***

Constructor

- A ***constructor*** is a special member function which initializes the objects
  - Has the exact same name as its class
  - No return type
  - It has to be public
- Invoked implicitly/explicitly every time an instance/object of that class is created
- Can be overloaded
- Are called in the order of creation
- Involve memory allocation
- Once a constructor is defined, the default one stops working

Type of Constructors

- 3 types of constructors
  - Default / Zero argument – A constructor that accepts no parameter is called default constructor. Initializes the members of all the objects to a predefined value.
  - Parameterized – A constructor that accepts parameters is called parameterized constructor. Used when it is necessary to initialize different values to the members of different objects.
  - Copy Constructor

Default Constructor

```
class student {
        int id;
public:
        void setValues(){
        id = 0;
        }
        int getId(){
        return id;
        }
};
int main(){
student st;
st. setValues();
cout<<st.getId();
}
```

```
class student {
        int id;
public:
        student() {
        id=0;
        }
        int getId() {
        return id;
        }
};
int main(){
student st;
cout<<st.getId();
}
```

Constructor

Destructor

- A ***destructor*** is a special member function which destroys the objects
  - Has the exact same name as its class, preceded by the sign '~'
  - No return type
  - It has to be public
- Can not be overloaded
- Called when an object is destroyed
  - A function ends
  - Global objects destroyed when the program ends
  - a delete operator is called
  - A block containing local variable ends
- Are called in the reverse order of creation
- Cleans memory or deallocates memory

Why Defining Destructor is Necessary?

- If user defined ***destructor*** is not written in class, compiler calls the ***default destructor.*** Default destructor deallocates memory unless we have dynamically allocated memory. When a class contains a pointer to allocate memory dynamically, we should write a destructor to release memory before the class instance/object is destroyed.
- This has to be done to avoid memory leak.

Constructor and Destructor

```
class student {
        int id;
public:
        void setValues(){
        id = 0;
        }
        int getId(){
        return id;
        }
};
int main(){
student st;
st. setValues();
cout<<st.getId();
}
```

```
class student {
        int id;
public:
        student() {
        id=0;
        }
        ~student() {
        }
        int getId() {
        return id; }
};
int main(){
student st;
cout<<st.getId();
}
```

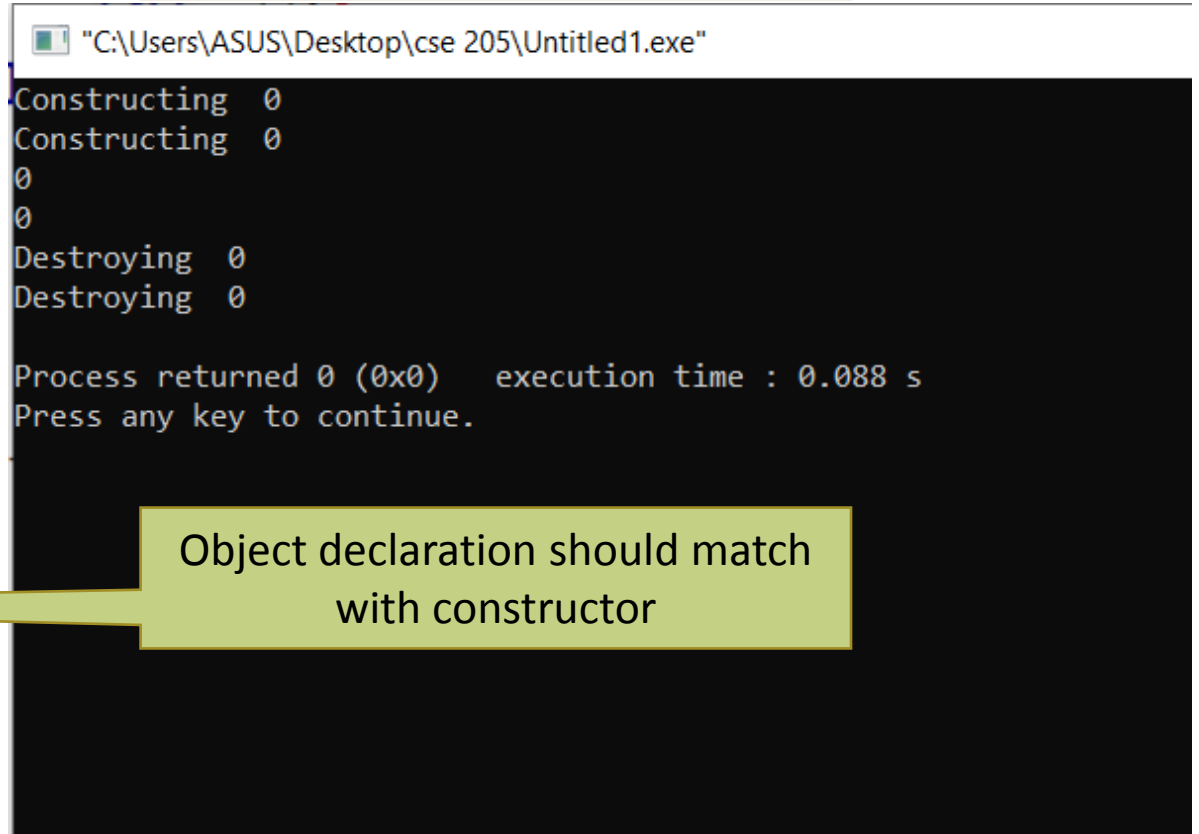
Constructor

Destructor

Default/ Zero Argument Constructor

```
class student {
    int id;
public:
    student() {
    id = 0;
    cout << "Constructing "<<id<<endl;
    }
    ~student(){
    cout << "Destructing "<<id<<endl;
    }
    int getId() {return id;}
};

int main(){
student st1,st2;
cout<<st1.getId()<<endl;
cout<<st2.getId()<<endl;
}
```

Constructor with zero parameter



Object declaration should match with constructor

Parameterized Constructors

- Parameterized Constructor – When a constructor has been parameterized, passing the initial values as arguments to the constructor can be done in two ways while declaring the object:
  - By calling the constructor explicitly – student object = student(2);
  - By calling the constructor implicitly – student object(2);
- When a constructor is parameterized, appropriate arguments should be sent for the constructor.

Parameterized Constructors

```
class student {
    int id;
public:
    student(int i) {
    id = i;
    cout << "Constructing "<<id<<endl;
    }
    ~student(){
    cout << "Destructing "<<id<<endl;
    }
    int getId() {return id;}
};

int main(){
student st1(1),st2(2);
cout<<st1.getId()<<endl;
cout<<st2.getId()<<endl;
}
```

```
"C:\Users\MP & MC LAB\Desktop\Untitled1.exe"
Constructing 1
Constructing 2
1
2
Destructing 2
Destructing 1

                returned 0 (0x0)   execution time : 0.173 s
```

Parameterized constructor

Object declaration should match the constructor

Parameterized Constructors

```
class student {
    int id;
public:
    student(int i) {
    id = i;
    cout << "Constructing "<<id<<endl;
    }
    ~student(){
    cout << "Destructing "<<id<<endl;
    }
    int getId() {return id;}
};

int main(){
    student st(1);
    student st2 = student(2);
    cout<<st.getId()<<endl;
    cout<<st2.getId()<<endl;
    }
```

Parameterized constructor

Both the declaration is allowed

Parameterized Constructors

```
class student {
    int id;
public:
    student(int i) {
    id = i;
    cout << "Constructing "<<id<<endl;
    }
    ~student(){
    cout << "Destructing "<<id<<endl;
    }
    int getId() {return id;}
};

int main(){
    student st(1),st2;
    cout<<st.getId()<<endl;
    cout<<st2.getId()<<endl;
    }
```

What will happen now?

```
16  int main(){
17      student st(1),st2;
```

Blocks | Search results | Cccc | Build log | Build messages | CppCheck

	Line	Message
\SUS\...	6	note: candidate expects 1 argument, 0 provided
\SUS\...	3	note: candidate: constexpr student::student(const student&)
\SUS\...	3	note: candidate expects 1 argument, 0 provided

Solution : Multiple Constructor in a Class/ Constructor Overloading

- Once a constructor is defined, the default one stops working.
- While defining constructor, we should take care of the fact that, there must be a constructor function for each way that an object of a class will be created.
- If a program attempts to create an object for which no matching constructor is found, compile-time error occurs.
- In the previous example, there was compilation error, because the declaration "student st2" did not match any constructor call. The only constructor function in the class accepts one parameter. But the declaration provides no argument.
- To avoid the error in the previous example, we could define 2 constructors.
  - **Default constructor** – this would match the declaration of student st2
  - **Parameterized constructor** – this would match the declaration of student st1(1)

Multiple Constructors

```
class student {
    int id;
public:
    student() {
    id = 0;
    cout << "Constructing "<<id<<endl;
    }
    student(int i) {
    id = i;
    cout << "Constructing "<<id<<endl;
    }
    ~student(){
    cout << "Destructing "<<id<<endl;
    }
    int getId() {return id;}
};
```

```
int main(){
    student st(1),st2;
    cout<<st.getId()<<endl;
    cout<<st2.getId()<<endl;
    }
```

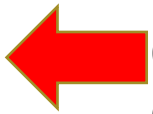
Necessity of Constructor Overloading

- To gain flexibility
- To support arrays
- To create copy constructors

Constructor-Destructor Call

```
class student {
    int id;
    char* name;
public:
    student(char* p , int q) {
    id=q;
    name = new char[strlen(p)];
    strcpy(name,p);
    cout << "Constructing: "<<name<<endl;
    }
    ~student(){
    cout << "Destructing "<<name<<endl;
    delete [] name;
    }
    int getId() {return id;}
};
```

```
void func(student st) {
cout << st.getId()<<endl;
}

int main(){
student st1("St1",1);
func(st1);
cout<<"Finish"<<endl;
}
```



Object declaration

OUTPUT??

Constructing st1

Constructor-Destructor Call

```
class student {
    int id;
    char* name;
public:
    student(char* p , int q) {
    id=q;
    name = new char[strlen(p)];
    strcpy(name,p);
    cout << "Constructing: "<<name<<endl;
    }
    ~student(){
    cout << "Destructing "<<name<<endl;
    delete [] name;
    }
    int getId() {return id;}
};
```

```
void func(student st) {
cout << st.getId()<<endl;
}
int main(){
student st1("St1",1);
func(st1);
cout<<"Finish"<<endl;
}
```

- This statement is not object declaration.
- This is an assignment stmt student st = st1;

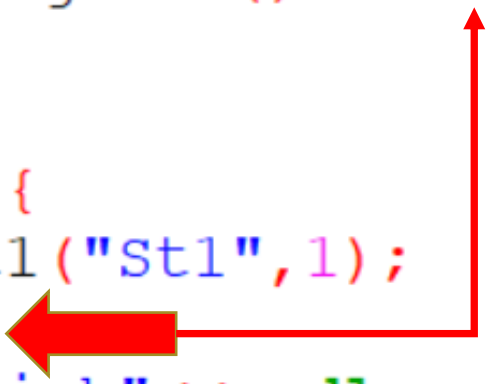
OUTPUT??

Constructing st1

Constructor-Destructor Call

```
class student {
    int id;
    char* name;
public:
    student(char* p , int q) {
    id=q;
    name = new char[strlen(p)];
    strcpy(name,p);
    cout << "Constructing: "<<name<<endl;
    }
    ~student(){
    cout << "Destructing "<<name<<endl;
    delete [] name;
    }
    int getId() {return id;}
};
```

```
void func(student st) {
cout << st.getId()<<endl;
}

int main(){
student st1("St1",1);
func(st1);
cout<<"Finish"<<endl;
}
```



OUTPUT??

Constructing st1
1

Constructor-Destructor Call

```
class student {
    int id;
    char* name;
public:
    student(char* p , int q) {
    id=q;
    name = new char[strlen(p)];
    strcpy(name,p);
    cout << "Constructing: "<<name<<endl;
    }
    ~student(){
    cout << "Destructing "<<name<<endl;
    delete [] name;
    }
    int getId() {return id;}
};
```

```
void func(student st) {
cout << st.getId()<<endl;
}

int main(){
student st1(
func(st1);
cout<<"Finish"<<endl;
}
```

- func() ends here
- The object "st" is a local variable of the function.
- So, destructor for object "st" will be called

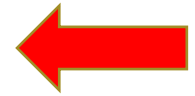
OUTPUT??

Constructing st1
1
Destructing st1

Constructor-Destructor Call

```
class student {
    int id;
    char* name;
public:
    student(char* p , int q) {
    id=q;
    name = new char[strlen(p)];
    strcpy(name,p);
    cout << "Constructing: "<<name<<endl;
    }
    ~student(){
    cout << "Destructing "<<name<<endl;
    delete [] name;
    }
    int getId() {return id;}
};
```

```
void func(student st) {
cout << st.getId()<<endl;
}

int main(){
student st1("St1",1);
func(st1);
cout<<"Finish"<<endl;
}
```



OUTPUT??

Constructing st1
1
Destructing st1
Finish

Constructor-Destructor Call

```
class student {
    int id;
    char* name;
public:
    student(char* p , int q) {
    id=q;
    name = new char[strlen(p)];
    strcpy(name,p);
    cout << "Constructing: "<<name<<endl;
    }
    ~student(){
    cout << "Destructing "<<name<<endl;
    delete [] name;
    }
    int getId() {return id;}
};
```

```
void func(student st) {
cout << st.getId()<<endl;
}

int main(){
student st1("St1",1);
func(st1);
cout<<"Finish"<<endl;
}
```

- Int main() ends here
- The object "st1" is a local variable of the function.
- So, destructor for object "st1" will be called

OUTPUT??

Constructing st1
1
Destructing st1
Finish
Destructing st1

Constructor-Destructor Call

```
class student {
    int id;
    char* name;
public:
    student(char* p , int q) {
    id=q;
    name = new char[strlen(p)];
    strcpy(name,p);
    cout << "Constructing: "<<name<<endl;
    }
    ~student(){
    cout << "Destructing "<<name<<endl;
    delete [] name;
    }
    int getId() {return id;}
};
```

```
void func(student st) {
cout << st.getId()<<endl;
}

int main(){
student st1("St1",1);
func(st1);
cout<<"Finish"<<endl;
```

- Names of the objects character pointer
- While calling the function ***func(st1)***, value of st1 is stored in st(a bitwise copy made)
- ***st.name = st1.name & st.id = st1.id***
- Both the objects names are pointing to the same memory. ***st.name & st1.name*** are pointing to the same memory location
- When the destructor is called 1st time, it destroys the memory containing the name

OUTPUT??

Constructing st1
1
Destructing st1
Finish
Destructing st1

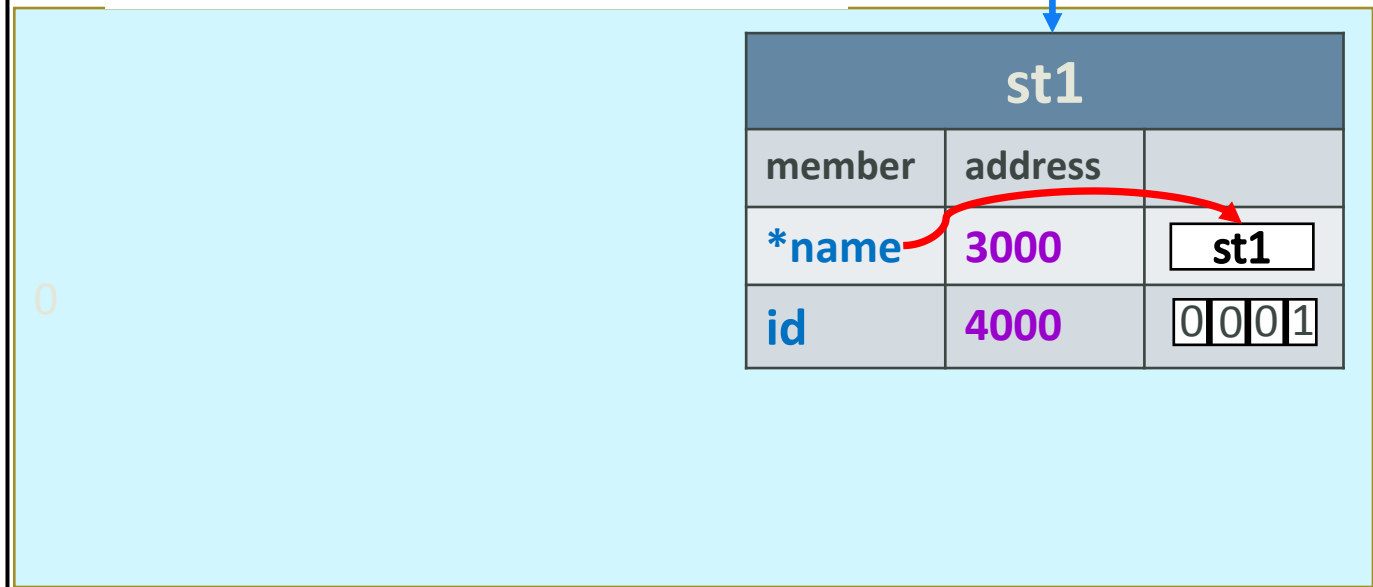
Constructor-Destructor Call

```
class student {
    int id;
    char* name;
public:
    student(char* p , int q) {
    id=q;
    name = new char[strlen(p)];
    strcpy(name,p);
    cout << "Constructing: "<<name<<endl;
    }
    ~student(){
    cout << "Destructing "<<name<<endl;
    delete [] name;
    }
    int getId() {return id;}
};
```

```
void func(student st) {
cout << st.getId()<<endl;
}

int main(){
student st1("St1",1);
func(st1);
cout<<"Finish"<<endl;
}
```

OUTPUT??

Constructing st1



Constructor-Destructor Call

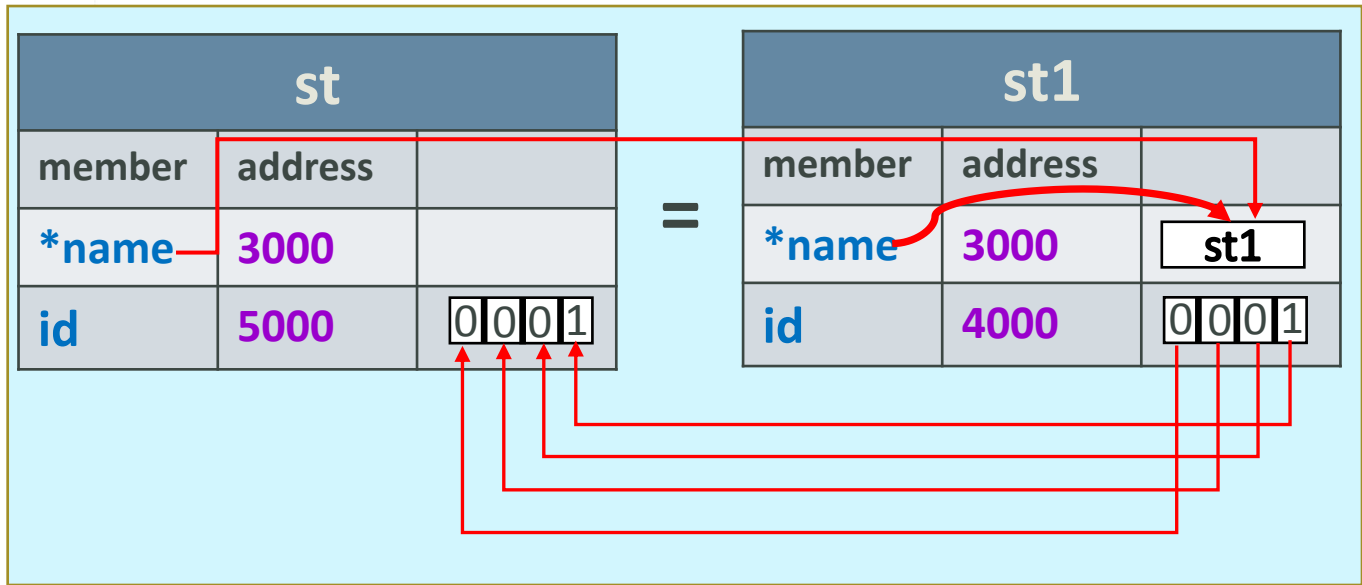
```
class student {
    int id;
    char* name;
public:
    student(char* p , int q) {
    id=q;
    name = new char[strlen(p)];
    strcpy(name,p);
    cout << "Constructing: "<<name<<endl;
    }
    ~student(){
    cout << "Destructing "<<name<<endl;
    delete [] name;
    }
    int getId() {return id;}
};
```

```
void func(student st) {
cout << st.getId()<<endl;
}

int main(){
student st1("St1",1);
func(st1);
cout<<"Finish"<<endl;
}
```

OUTPUT??

Constructing st1

st = st1(bitwise copy)



Constructor-Destructor Call

```
class student {
    int id;
    char* name;
public:
    student(char* p , int q) {
    id=q;
    name = new char[strlen(p)];
    strcpy(name,p);
    cout << "Constructing: "<<name<<endl;
    }
    ~student(){
    cout << "Destructing "<<name<<endl;
    delete [] name;
    }
    int getId() {return id;}
};
```

```
void func(student st) {
cout << st.getId()<<endl;
}

int main(){
student st1("St1",1);
func(st1);
cout<<"Finish"<<endl;
}
```

OUTPUT??

Constructing st1
1

st		
member	address	
*name	3000	
id	5000	0 0 0 1

=

st1		
member	address	
*name	3000	st1
id	4000	0 0 0 1

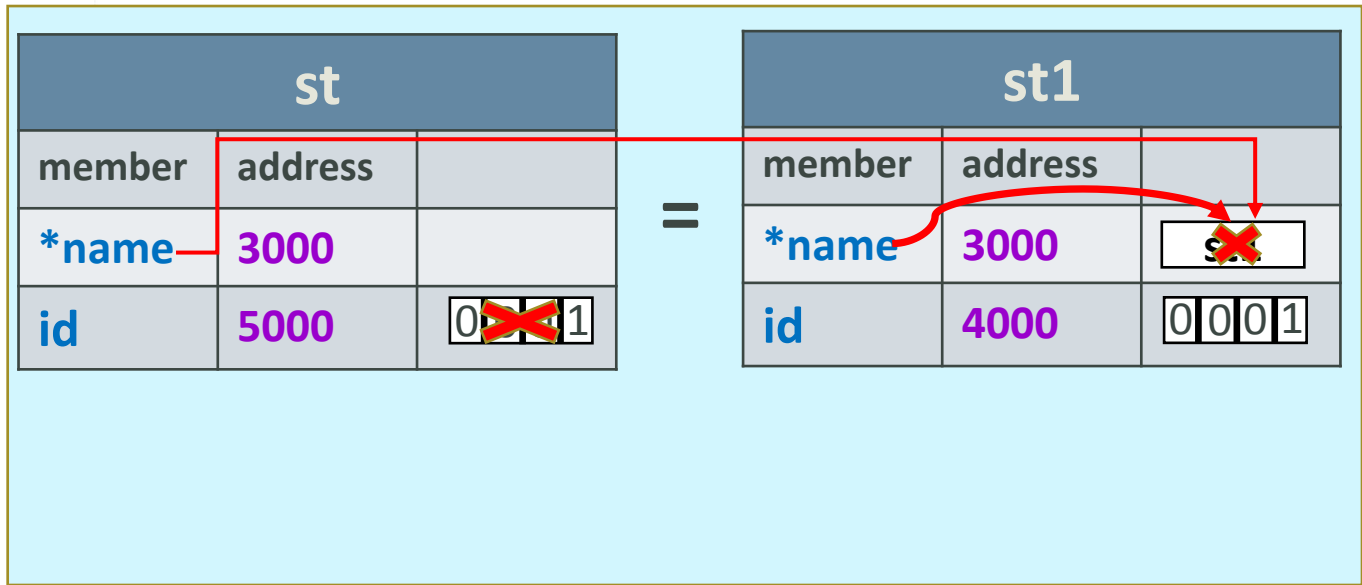
Constructor-Destructor Call

```
class student {
    int id;
    char* name;
public:
    student(char* p , int q) {
    id=q;
    name = new char[strlen(p)];
    strcpy(name,p);
    cout << "Constructing: "<<name<<endl;
    }
    ~student(){
    cout << "Destructing "<<name<<endl;
    delete [] name;
    }
    int getId() {return id;}
};
```

```
void func(student st) {
cout << st.getId()<<endl;
}

int main(){
student st1("St1",1);
func(st1);
cout<<"Finish"<<endl;
}
```

OUTPUT??

Constructing st1
1
Destructing st1



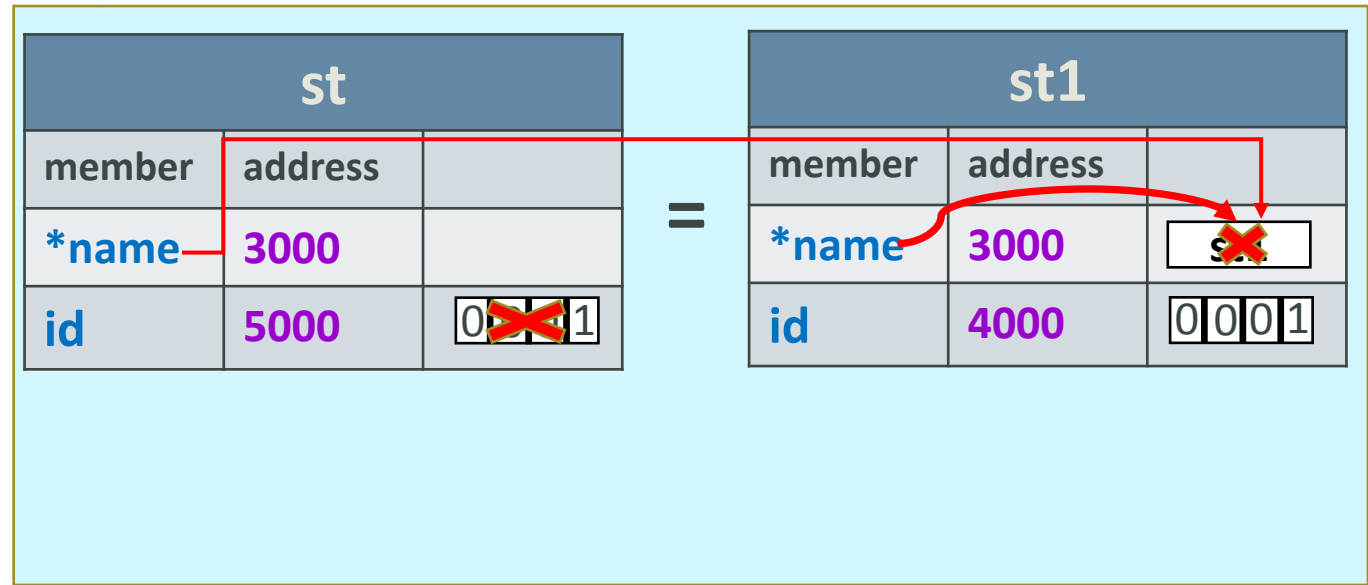
Constructor-Destructor Call

```
class student {
    int id;
    char* name;
public:
    student(char* p , int q) {
    id=q;
    name = new char[strlen(p)];
    strcpy(name,p);
    cout << "Constructing: "<<name<<endl;
    }
    ~student(){
    cout << "Destructing "<<name<<endl;
    delete [] name;
    }
    int getId() {return id;}
};
```

```
void func(student st) {
cout << st.getId()<<endl;
}

int main(){
student st1("St1",1);
func(st1);
cout<<"Finish"<<endl;
}
```

OUTPUT??

Constructing st1
1
Destructing st1
Finish



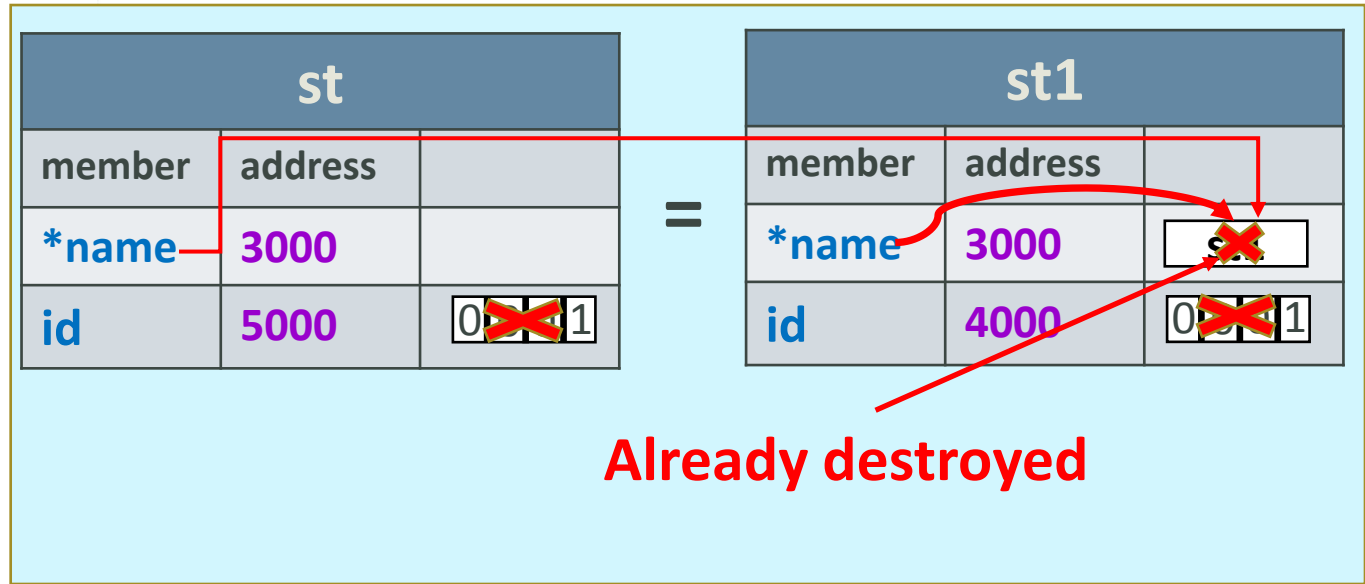
Constructor-Destructor Call

```
class student {
    int id;
    char* name;
public:
    student(char* p , int q) {
    id=q;
    name = new char[strlen(p)];
    strcpy(name,p);
    cout << "Constructing: "<<name<<endl;
    }
    ~student(){
    cout << "Destructing "<<name<<endl;
    delete [] name;
    }
    int getId() {return id;}
};
```

```
void func(student st) {
cout << st.getId()<<endl;
}

int main(){
student st1("St1",1);
func(st1);
cout<<"Finish"<<endl;
}
```

OUTPUT??

Constructing st1
1
Destructing st1
Finish
Destructing st1



Solution??

- Assignment
- Find out the solution

Constructor-Destructor Call

```
class student {
    int id;
    char* name;
public:
    student(char* p , int q) {
    id=q;
    name = new char[strlen(p)];
    strcpy(name,p);
    cout << "Constructing: "<<name<<endl;
    }
    ~student(){
    cout << "Destructing "<<name<<endl;
    delete [] name;
    }
    int getId() {return id;}
};
```

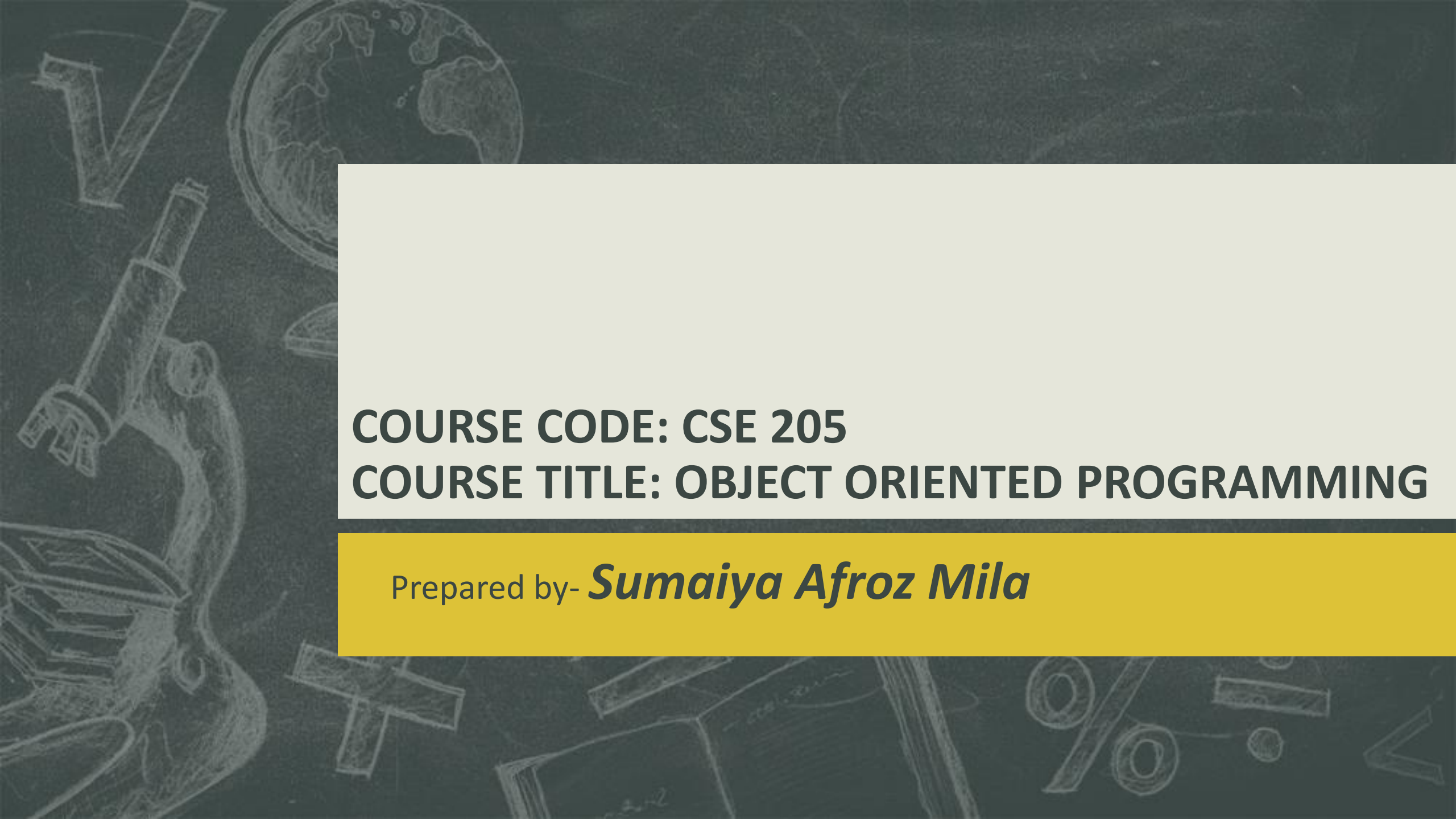
```
void func(student &st) {
cout << st.getId()<<endl;
}

int main(){
student st1("St1",1);
func(st1);
cout<<"Finish"<<endl;
}
```

Constructor-Destructor Call

```
class student {
    int id;
    char* name;
public:
    student(char* p , int q) {
    id=q;
    name = new char[strlen(p)];
    strcpy(name,p);
    cout << "Constructing: "<<name<<endl;
    }
    ~student(){
    cout << "Destructing "<<name<<endl;
    delete [] name;
    }
    int getId() {return id;}
};
```

```
void func(student *st) {
cout << st->getId()<<endl;
}

int main(){
student st1("St1",1);
func(&st1);
cout<<"Finish"<<endl;
}
```


Thank
you!



COURSE CODE: CSE 205
COURSE TITLE: OBJECT ORIENTED PROGRAMMING

Prepared by- ***Sumaiya Afroz Mila***

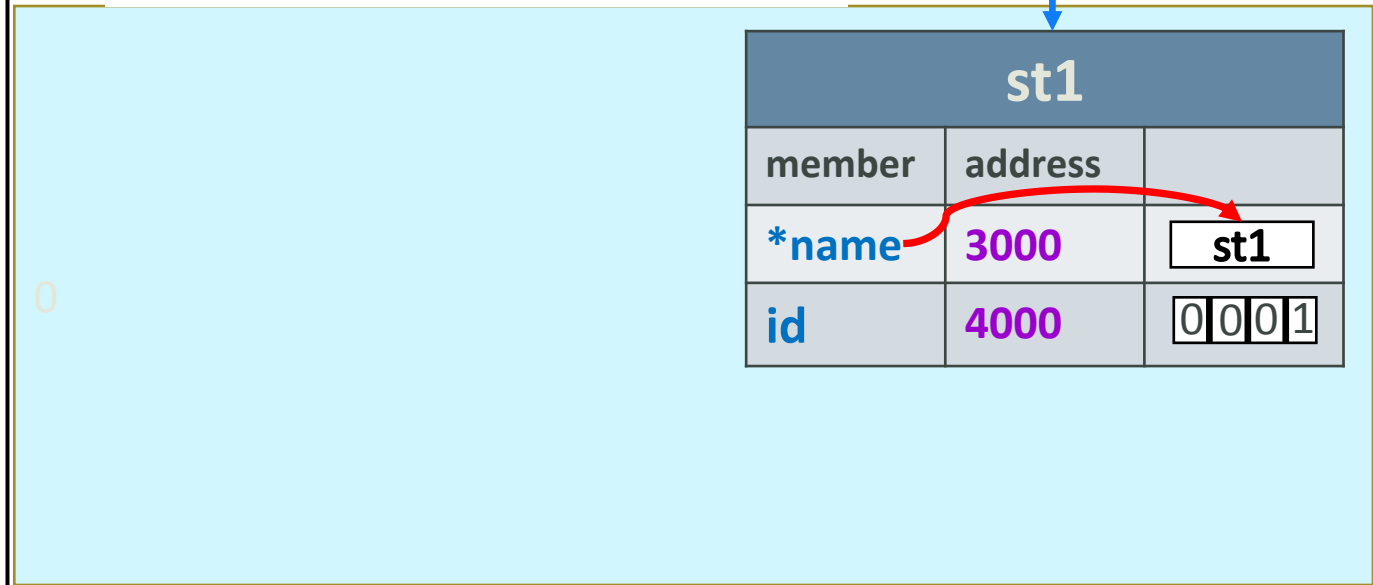
The Old Problem

```
class student {
    int id;
    char* name;
public:
    student(char* p , int q) {
    id=q;
    name = new char[strlen(p)];
    strcpy(name,p);
    cout << "Constructing: "<<name<<endl;
    }
    ~student(){
    cout << "Destructing "<<name<<endl;
    delete [] name;
    }
    int getId() {return id;}
};
```

```
void func(student st) {
cout << st.getId()<<endl;
}

int main(){
student st1("St1",1);
func(st1);
cout<<"Finish"<<endl;
}
```

OUTPUT??

Constructing st1



The Old Problem

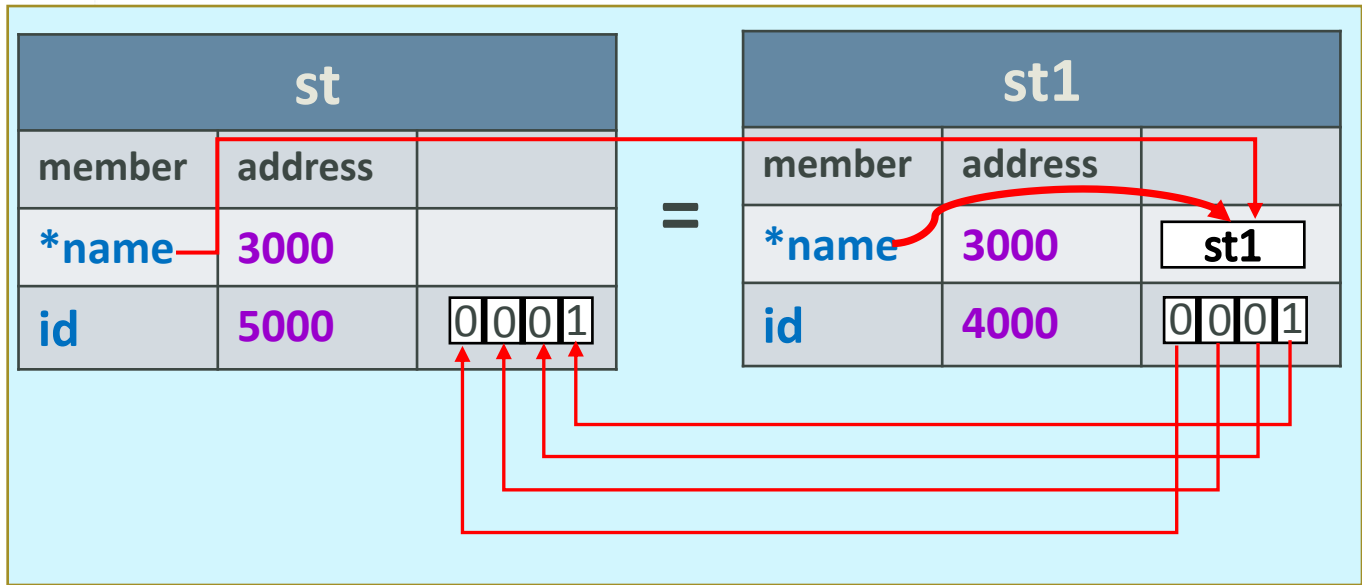
```
class student {
    int id;
    char* name;
public:
    student(char* p , int q) {
    id=q;
    name = new char[strlen(p)];
    strcpy(name,p);
    cout << "Constructing: "<<name<<endl;
    }
    ~student(){
    cout << "Destructing "<<name<<endl;
    delete [] name;
    }
    int getId() {return id;}
};
```

```
void func(student st) {
cout << st.getId()<<endl;
}

int main(){
student st1("St1",1);
func(st1);
cout<<"Finish"<<endl;
}
```

OUTPUT??

Constructing st1

st = st1(bitwise copy)



The Old Problem

```
class student {
    int id;
    char* name;
public:
    student(char* p , int q) {
    id=q;
    name = new char[strlen(p)];
    strcpy(name,p);
    cout << "Constructing: "<<name<<endl;
    }
    ~student(){
    cout << "Destructing "<<name<<endl;
    delete [] name;
    }
    int getId() {return id;}
};
```

```
void func(student st) {
cout << st.getId()<<endl;
}

int main(){
student st1("St1",1);
func(st1);
cout<<"Finish"<<endl;
}
```

OUTPUT??

Constructing st1
1

st		
member	address	
*name	3000	
id	5000	0 0 0 1

=

st1		
member	address	
*name	3000	st1
id	4000	0 0 0 1

The Old Problem

```
class student {
    int id;
    char* name;
public:
    student(char* p , int q) {
    id=q;
    name = new char[strlen(p)];
    strcpy(name,p);
    cout << "Constructing: "<<name<<endl;
    }
    ~student(){
    cout << "Destructing "<<name<<endl;
    delete [] name;
    }
    int getId() {return id;}
};
```

```
void func(student st) {
cout << st.getId()<<endl;
}

int main(){
student st1("St1",1);
func(st1);
cout<<"Finish"<<endl;
}
```

OUTPUT??

Constructing st1
1
Destructing st1

st		
member	address	
*name	3000	
id	5000	0 0 0 1

=

st1		
member	address	
*name	3000	St1
id	4000	0 0 0 1

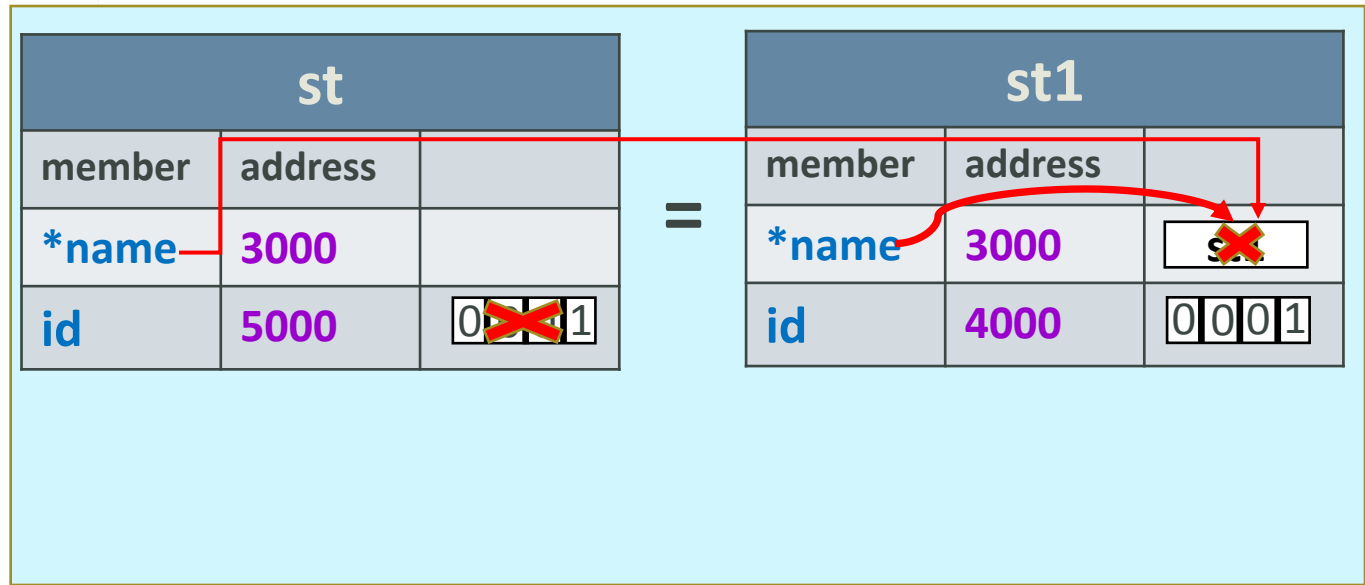
The Old Problem

```
class student {
    int id;
    char* name;
public:
    student(char* p , int q) {
    id=q;
    name = new char[strlen(p)];
    strcpy(name,p);
    cout << "Constructing: "<<name<<endl;
    }
    ~student(){
    cout << "Destructing "<<name<<endl;
    delete [] name;
    }
    int getId() {return id;}
};
```

```
void func(student st) {
cout << st.getId()<<endl;
}

int main(){
student st1("St1",1);
func(st1);
cout<<"Finish"<<endl;
}
```

OUTPUT??

Constructing st1
1
Destructing st1
Finish



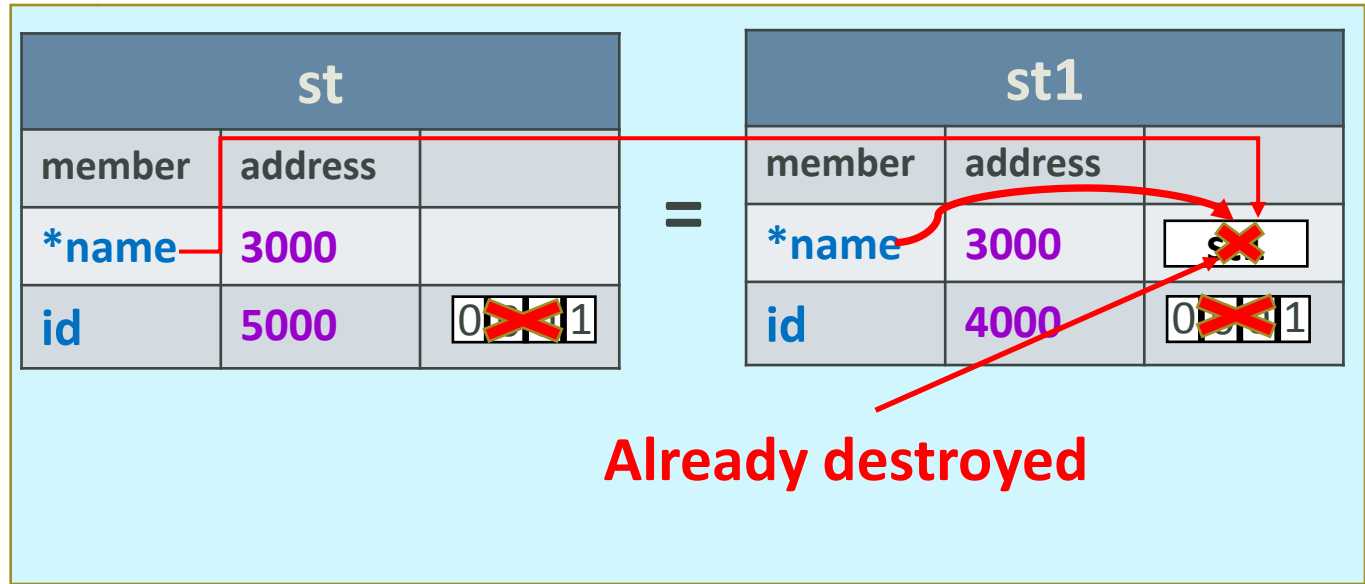
The Old Problem

```
class student {
    int id;
    char* name;
public:
    student(char* p , int q) {
    id=q;
    name = new char[strlen(p)];
    strcpy(name,p);
    cout << "Constructing: "<<name<<endl;
    }
    ~student(){
    cout << "Destructing "<<name<<endl;
    delete [] name;
    }
    int getId() {return id;}
};
```

```
void func(student st) {
cout << st.getId()<<endl;
}

int main(){
student st1("St1",1);
func(st1);
cout<<"Finish"<<endl;
}
```

OUTPUT??

Constructing st1
1
Destructing st1
Finish
Destructing st1



A New Solution – User Defined Copy Constructor

- The ***default copy constructor*** makes a bitwise copy.
- Copy Constructor invoked in 3 ways –
  - When an object is used to **initialize** another in a **declaration** statement(***student st2=st1***;)
  - **When an object is passed as a parameter to a function**
  - **When a temporary object is created for use as a return value by a function**
- ***User defined copy constructor*** specifies exactly what occurs when a copy of an object is made

Copy Constructor Syntax

- Most Common Form –

```
classname (const classname &obj){
// body
}
```

Here ***obj*** is a **reference** to an **object** that is being **used to initialize another object.**

- Copy Constructor of the student class –

```
student (const student &ob){
        id = obj.id;
        name = new char[strlen(obj.name)+1];
        strcpy(name,obj.name);
}
```

Copy Constructor Invocation

```
class student {
    int id;
    char* name;
public:
    student(char* p , int q) {
    id=q;
    name = new char[strlen(p)];
    strcpy(name,p);
    cout << "Constructing: "<<name<<endl;
    }
    student(const student &obj){
        id = obj.id;
        name = new char[strlen(obj.name)+1];
        strcpy(name,obj.name);
    }
    ~student(){
    cout << "Destructing "<<name<<endl;
    delete [] name;
    }
    int getId() {return id;}
};
```

```
void func(student st) {
cout << st.getId()<<endl;
}

int main(){
student st1("St1",1);
func(st1);
cout<<"Finish"<<endl;
}
```

Copy Constructor Invocation

Copy Constructor Invocation

- Student s1 = s2; // y explicitly initializing x
- func(s2); // y passed as a parameter
- s1 = func(s1); // y receiving a returned object

Copy Constructor Invocation Example

```
int main(){
student st1("St1",1), st2("st2",2);
student st3 = st1;
st3 = func(st1);
cout<<"Finish"<<endl;
}
```

Copy Constructor Invocation(initialization)

Copy Constructor Invocation (passing as parameter)

Copy Constructor Invocation(return from function)

Copy Constructor Invocation Example

```
class student {
    int id;
    char* name;
public:
    student(char* p , int q) {
    id=q;
    name = new char[strlen(p)];
    strcpy(name,p);
    cout << "Constructing: "<<name<<endl;
    }
    student(const student &obj){
        id = obj.id;
        name = new char[strlen(obj.name)+1];
        strcpy(name,obj.name);
        cout<<"Copy Constructor calling  "<<name<<endl;
    }
    ~student(){
    cout << "Destructing "<<name<<endl;delete [] name;
    }
    int getId() {return id;}
};
```

```
student func(student st) {
student temp("temp",3);
return temp;
}

int main(){ 1
student st1("St1",1), st2("st2",2);
student st3 = st1;
st3 = func(st1);
cout<<"Finish"<<endl;
}
```

1.Constructing: St1 (In main)	

Copy Constructor Invocation Example

```
class student {
    int id;
    char* name;
public:
    student(char* p , int q) {
    id=q;
    name = new char[strlen(p)];
    strcpy(name,p);
    cout << "Constructing: "<<name<<endl;
    }
    student(const student &obj){
        id = obj.id;
        name = new char[strlen(obj.name)+1];
        strcpy(name,obj.name);
        cout<<"Copy Constructor calling  "<<name<<endl;
    }
    ~student(){
    cout << "Destructing "<<name<<endl;delete [] name;
    }
    int getId() {return id;}
};
```

```
student func(student st) {
student temp("temp",3);
return temp;
}

int main(){
student st1("St1",1), st2("st2",2);
student st3 = st1;
st3 = func(st1);
cout<<"Finish"<<endl;
}
```

1.Constructing: St1 (In main)
2.Constructing: St2 (In main)

Copy Constructor Invocation Example

```
class student {
    int id;
    char* name;
public:
    student(char* p , int q) {
    id=q;
    name = new char[strlen(p)];
    strcpy(name,p);
    cout << "Constructing: "<<name<<endl;
    }
    student(const student &obj){
        id = obj.id;
        name = new char[strlen(obj.name)+1];
        strcpy(name,obj.name);
        cout<<"Copy Constructor calling  "<<name<<endl;
    }
    ~student(){
    cout << "Destructing "<<name<<endl;delete [] name;
    }
    int getId() {return id;}
};
```

```
student func(student st) {
student temp("temp",3);
return temp;
}

int main(){
student st1("St1",1), st2("st2",2);
student st3 = st1;
st3 = func(st1);
cout<<"Finish"<<endl;
}
```

1.Constructing: St1(In main)
2.Constructing: St2(In main)
3.Copy Constructor Calling St1 (In main)

Copy Constructor Invocation Example

```
class student {
    int id;
    char* name;
public:
    student(char* p , int q) {
    id=q;
    name = new char[strlen(p)];
    strcpy(name,p);
    cout << "Constructing: "<<name<<endl;
    }
    student(const student &obj){
        id = obj.id;
        name = new char[strlen(obj.name)+1];
        strcpy(name,obj.name);
        cout<<"Copy Constructor calling  "<<name<<endl;
    }
    ~student(){
    cout << "Destructing "<<name<<endl;delete [] name;
    }
    int getId() {return id;}
};
```

```
student func(student st) {
student temp("temp",3);
return temp;
}

int main(){
student st1("St1",1), st2("st2",2);
student st3 = st1;
st3 = func(st1);
cout<<"Finish"<<endl;
}
```

1. Constructing: St1(In main)
2. Constructing: St2(In main)
3. Copy Constructor Calling St1 (In main)
4. Copy Constructor Calling St1 (In function parameter)

Copy Constructor Invocation Example

```
class student {
    int id;
    char* name;
public:
    student(char* p , int q) {
    id=q;
    name = new char[strlen(p)];
    strcpy(name,p);
    cout << "Constructing: "<<name<<endl;
    }
    student(const student &obj){
        id = obj.id;
        name = new char[strlen(obj.name)+1];
        strcpy(name,obj.name);
        cout<<"Copy Constructor calling  "<<name<<endl;
    }
    ~student(){
    cout << "Destructing "<<name<<endl;delete [] name;
    }
    int getId() {return id;}
};
```

```
student func(student st) {
student temp("temp",3);
return temp;
}

int main(){
student st1("St1",1), st2("st2",2);
student st3 = st1;
st3 = func(st1);
cout<<"Finish"<<endl;
}
```

1.Constructing: St1(In main)
2.Constructing: St2(In main)
3.Copy Constructor Calling St1 (In main)
4.Copy Constructor Calling St1 (In function parameter)
6.Constructing: temp(In local)

Copy Constructor Invocation Example

```
class student {
    int id;
    char* name;
public:
    student(char* p , int q) {
    id=q;
    name = new char[strlen(p)];
    strcpy(name,p);
    cout << "Constructing: "<<name<<endl;
    }
    student(const student &obj){
        id = obj.id;
        name = new char[strlen(obj.name)+1];
        strcpy(name,obj.name);
        cout<<"Copy Constructor calling  "<<name<<endl;
    }
    ~student(){
    cout << "Destructing "<<name<<endl;delete [] name;
    }
    int getId() {return id;}
};
```

```
student func(student st) {
student temp("temp",3);
return temp;
}

int main(){
student st1("St1",1), st2("st2",2);
student st3 = st1;
st3 = func(st1);
cout<<"Finish"<<endl;
}
```

1.Constructing: St1(In main)
2.Constructing: St2(In main)
3.Copy Constructor Calling St1
(In main)
4.Copy Constructor Calling St1
(In function parameter)
6.Constructing: temp(In local)
7.Copy Constructor Calling temp(in
local)

Copy Constructor Invocation Example

```
class student {
    int id;
    char* name;
public:
    student(char* p , int q
    id=q;
    name = new char[strlen(
    strcpy(name,p);
    cout << "Constructing:
    }
    student(const student &
        id = obj.id;
        name = new char[str
        strcpy(name,obj.name);
        cout<<"Copy Constructor calling  "<<name<<endl;
    }
    ~student(){
    cout << "Destructing "<<name<<endl;delete [] name;
    }
    int getId() {return id;}
};
```

The copy is **done before** the called function **exits**, and copies the then-existing local variable into the return value.
The called function has access to the memory the return value will occupy, even though that memory is not "in scope" when the copy is being made, it's still available.

```
student func(student st) {
student temp("temp",3);
return temp;
}

int main(){
student st1("St1",1), st2("st2",2);
student st3 = st1;
st3 = func(st1);
cout<<"Finish"<<endl;
}
```

1.Constructing: St1(In main)
2.Constructing: St2(In main)
3.Copy Constructor Calling St1 (In main)
4.Copy Constructor Calling St1 (In function parameter)
6.Constructing: temp(In local)
7.Copy Constructor Calling temp(in local)

Copy Constructor Invocation Example

```
class student {
    int id;
    char* name;
public:
    student(char* p , int q) {
    id=q;
    name = new char[strlen(p)];
    strcpy(name,p);
    cout << "Constructing: "<<name<<endl;
    }
    student(const student &obj){
        id = obj.id;
        name = new char[strlen(obj.name)+1];
        strcpy(name,obj.name);
        cout<<"Copy Constructor calling  "<<name<<endl;
    }
    ~student(){
    cout << "Destructing "<<name<<endl;delete [] name;
    }
    int getId() {return id;}
};
```

```
student func(student st) {
student temp("temp",3);
return temp;
}

int main(){
student st1("St1",1), st2("st2",2);
student st3 = st1;
st3 = func(st1);
cout<<"Finish"<<endl;
}
```

1.Constructing: St1(In main)
2.Constructing: St2(In main)
3.Copy Constructor Calling St1 (In main)
4.Copy Constructor Calling St1 (In function parameter)
6.Constructing: temp(In local)
7.Copy Constructor Calling temp(in local)

8.Destructing temp(local)

Copy Constructor Invocation Example

```
class student {
    int id;
    char* name;
public:
    student(char* p , int q) {
    id=q;
    name = new char[strlen(p)];
    strcpy(name,p);
    cout << "Constructing: "<<name<<endl;
    }
    student(const student &obj){
        id = obj.id;
        name = new char[strlen(obj.name)+1];
        strcpy(name,obj.name);
        cout<<"Copy Constructor calling  "<<name<<endl;
    }
    ~student(){
    cout << "Destructing "<<name<<endl;delete [] name;
    }
    int getId() {return id;}
};
```

```
student func(student st) {
student temp("temp",3);
return temp;
}

int main(){
student st1("St1",1), st2("st2",2);
student st3 = st1;
st3 = func(st1);
cout<<"Finish"<<endl;
}
```

1.Constructing: St1(In main)
2.Constructing: St2(In main)
3.Copy Constructor Calling St1 (In main)
4.Copy Constructor Calling St1 (In function parameter)
6.Constructing: temp(In local)
7.Copy Constructor Calling temp(in local)

8.Destructing temp(local)
9.Destructing St1(local)

Copy Constructor Invocation Example

```
class student {
    int id;
    char* name;
public:
    student(char* p , int q) {
    id=q;
    name = new char[strlen(p)];
    strcpy(name,p);
    cout << "Constructing: "<<name<<endl;
    }
    student(const student &obj){
        id = obj.id;
        name = new char[strlen(obj.name)+1];
        strcpy(name,obj.name);
        cout<<"Copy Constructor calling  "<<name<<endl;
    }
    ~student(){
    cout << "Destructing "<<name<<endl;delete [] name;
    }
    int getId() {return id;}
};
```

```
student func(student st) {
student temp("temp",3);
return temp;
}

int main(){
student st1("St1",1), st2("st2",2);
student st3 = st1;
st3 = func(st1);
cout<<"Finish"<<endl;
}
```

1.Constructing: St1(In main)
2.Constructing: St2(In main)
3.Copy Constructor Calling St1 (In main)
4.Copy Constructor Calling St1 (In function parameter)
6.Constructing: temp(In local)
7.Copy Constructor Calling temp(in local)

8.Destructing temp(local)
9.Destructing St1(local)
10. Destructing temp(return value)

Copy Constructor Invocation Example

```
class student {
    int id;
    char* name;
public:
    student(char* p , int q) {
    id=q;
    name = new char[strlen(p)];
    strcpy(name,p);
    cout << "Constructing: "<<name<<endl;
    }
    student(const student &obj){
        id = obj.id;
        name = new char[strlen(obj.name)+1];
        strcpy(name,obj.name);
        cout<<"Copy Constructor calling  "<<name<<endl;
    }
    ~student(){
    cout << "Destructing "<<name<<endl;delete [] name;
    }
    int getId() {return id;}
};
```

```
student func(student st) {
student temp("temp",3);
return temp;
}

int main(){
student st1("St1",1), st2("st2",2);
student st3 = st1;
st3 = func(st1);
cout<<"Finish"<<endl;
}
```

1.Constructing: St1(In main)
2.Constructing: St2(In main)
3.Copy Constructor Calling St1 (In main)
4.Copy Constructor Calling St1 (In function parameter)
6.Constructing: temp(In local)
7.Copy Constructor Calling temp(in local)

8.Destructing temp(local)
9.Destructing St1(local)
10. Destructing temp(return value)
Finish
11. Destructing temp(main)

Copy Constructor Invocation Example

```
class student {
    int id;
    char* name;
public:
    student(char* p , int q) {
    id=q;
    name = new char[strlen(p)];
    strcpy(name,p);
    cout << "Constructing: "<<name<<endl;
    }
    student(const student &obj){
        id = obj.id;
        name = new char[strlen(obj.name)+1];
        strcpy(name,obj.name);
        cout<<"Copy Constructor calling  "<<name<<endl;
    }
    ~student(){
    cout << "Destructing "<<name<<endl;delete [] name;
    }
    int getId() {return id;}
};
```

```
student func(student st) {
student temp("temp",3);
return temp;
}

int main(){
student st1("St1",1), st2("st2",2);
student st3 = st1;
st3 = func(st1);
cout<<"Finish"<<endl;
}
```

1.Constructing: St1(In main)
2.Constructing: St2(In main)
3.Copy Constructor Calling St1 (In main)
4.Copy Constructor Calling St1 (In function parameter)
6.Constructing: temp(In local)
7.Copy Constructor Calling temp(in local)

8.Destructing temp(local)
9.Destructing St1(local)
10. Destructing temp(return value)
Finish
11. Destructing temp(main)
12. Destructing St2(main)

Copy Constructor Invocation Example

```
class student {
    int id;
    char* name;
public:
    student(char* p , int q) {
    id=q;
    name = new char[strlen(p)];
    strcpy(name,p);
    cout << "Constructing: "<<name<<endl;
    }
    student(const student &obj){
        id = obj.id;
        name = new char[strlen(obj.name)+1];
        strcpy(name,obj.name);
        cout<<"Copy Constructor calling  "<<name<<endl;
    }
    ~student(){
    cout << "Destructing "<<name<<endl;delete [] name;
    }
    int getId() {return id;}
};
```

```
student func(student st) {
student temp("temp",3);
return temp;
}

int main(){
student st1("St1",1), st2("st2",2);
student st3 = st1;
st3 = func(st1);
cout<<"Finish"<<endl;
}
```

1.Constructing: St1(In main)
2.Constructing: St2(In main)
3.Copy Constructor Calling St1 (In main)
4.Copy Constructor Calling St1 (In function parameter)
6.Constructing: temp(In local)
7.Copy Constructor Calling temp(in local)

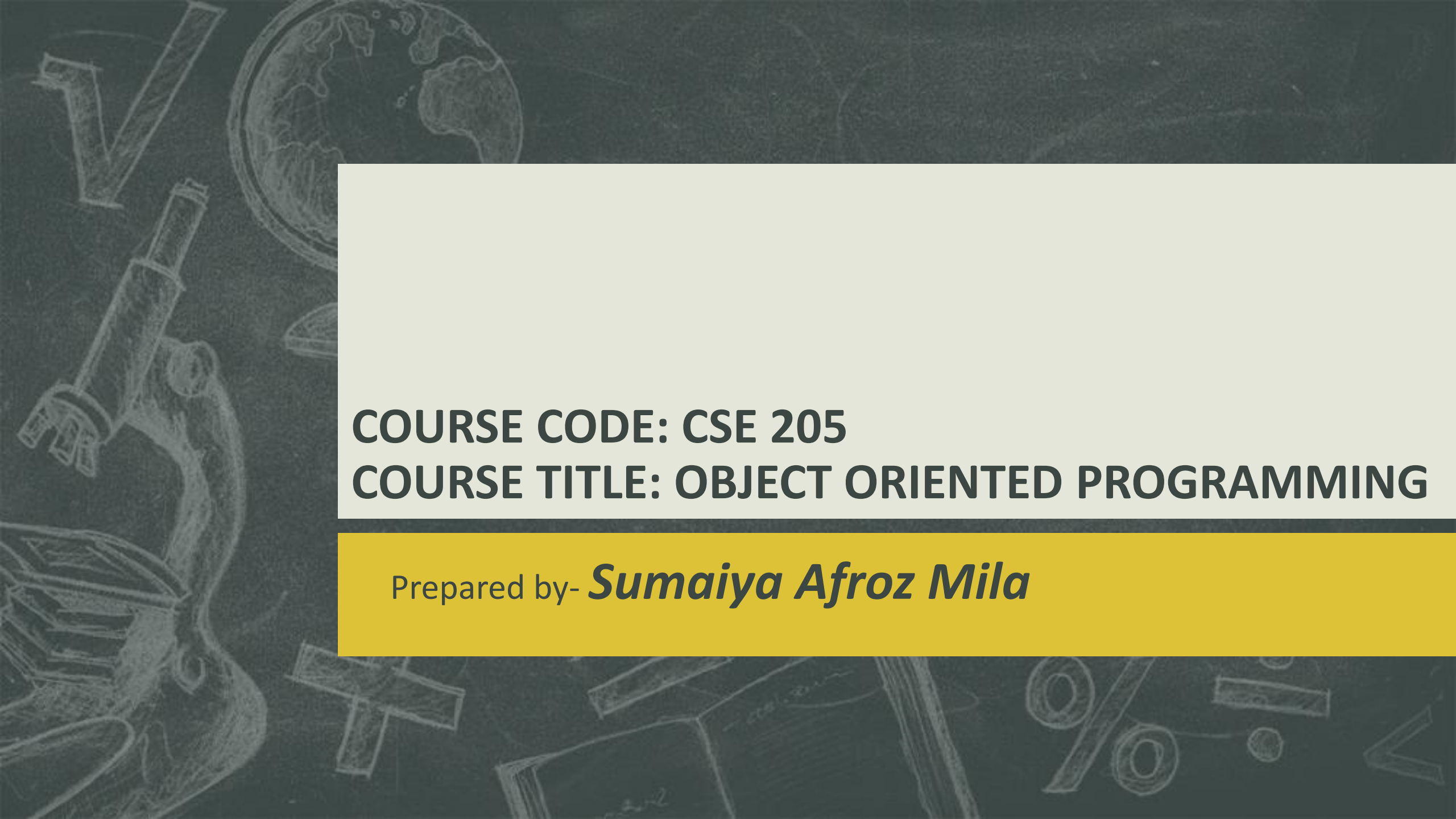
8.Destructing temp(local)
9.Destructing St1(local)
10. Destructing temp(return value)
Finish
11 Destructing temp(main)
12. Destructing St2(main)
13. Destructing St1(main)

Class Member of Another Class

```
class date {
    int day;
    int mon;
    int year;
public:
    date(){day=0; mon=0;year=0;}
    date(int d, int m, int y) {
    day = d; mon = m; year = y;
    }
    void setDay(int d) {day=d;}
    void setMon(int m) {mon=m;}
    void setYear(int y) {year=y;}
    int getDay() {return day;}
    int getMon() {return mon;}
    int getYear() {return year;}
};
```

```
class student {
    int id;
    date dob; //date of birth
public:
    student(int i, int d,int m,int y) {
    id = i;
    dob. setDay(d);
    dob. setMon(m);
    dob. setYear(y);
}
    int getId() {return id;}
    void printDob(){
    cout<<"day : "<<dob.getDay()<<endl;
    cout<<"Mon : "<<dob.getMon()<<endl;
    cout<<"Year : "<<dob.getYear()<<endl;
    }
};

int main(){
 student st(21,8,8,97);
 cout<<" ID is: "<<st.getId()<<endl;
 st.printDob();
}
```


Thank
you!



COURSE CODE: CSE 205
COURSE TITLE: OBJECT ORIENTED PROGRAMMING

Prepared by- ***Sumaiya Afroz Mila***

Static Class Member

- When a member variable is declared as static, It causes only **one copy of that variable to exist – no matter how many objects of that class are declared.**
- All the objects of that class simply **shares** that variable.
- Same static variable will be used by any classes derived from the class that contains a static variable – related to inheritance
- A static member variable exists ***before*** any object of that class is created.
- A static class member is a global variable that simply has its scope restricted to the class in which it is declared.

Static Class Member

- A static variable can only be declared inside a class. Definition must be outside the class.
- All static member variables are initialized to 0 **by default**.
- However, it is possible to choose an initial value other than 0 for the static member variable.
- A member function of a class can be declared as static

Assignment

- Why is it necessary to use static data members?
- Why static member function does not have “this” pointer?
- Why non static member variable can not be accessed by a static function?

Example 1

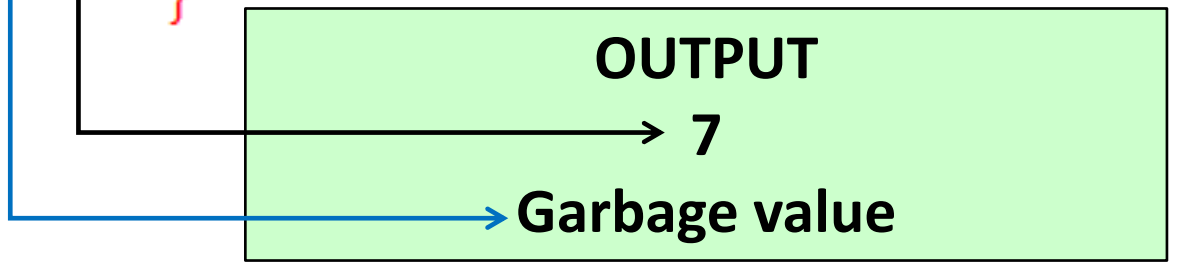
```
class myclass
{
    int x;
public:
    void setx(int n)
    {x = n; }
    int getx()
    {return x; }
};
```

```
int main()
{
myclass o1, o2;
o1.setx(7);
cout<< o1.getx() <<endl;
cout<< o2.getx() <<endl;
}
```

Example 1

```
class myclass
{
    int x;
public:
    void setx(int n)
    {x = n; }
    int getx()
    {return x; }
};
```

```
int main()
{
myclass o1, o2;
o1.setx(7);
cout<< o1.getx() <<endl;
cout<< o2.getx() <<endl;
}
```



Example 2

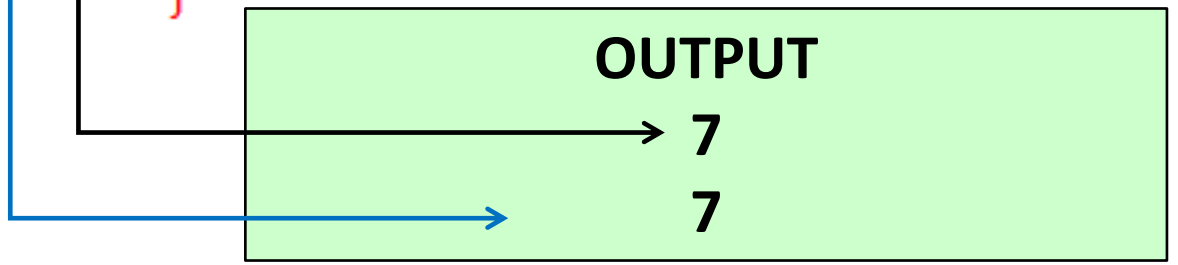
```
class myclass
{
      static int x;
public:
      void setx(int n)
      {x = n; }
      int getx()
      {return x; }
};

int myclass::x;
```

```
int main()
{
myclass o1, o2;
o1.setx(7);
cout<< o1.getx() <<endl;
cout<< o2.getx() <<endl;
}
```

Example 2

```
class myclass
{
      static int x;
public:
      void setx(int n)
      {x = n; }
      int getx()
      {return x; }
};

int myclass::x;
```

```
int main()
{
myclass o1, o2;
o1.setx(7);
cout<< o1.getx() <<endl;
cout<< o2.getx() <<endl;
}
```



Example 3

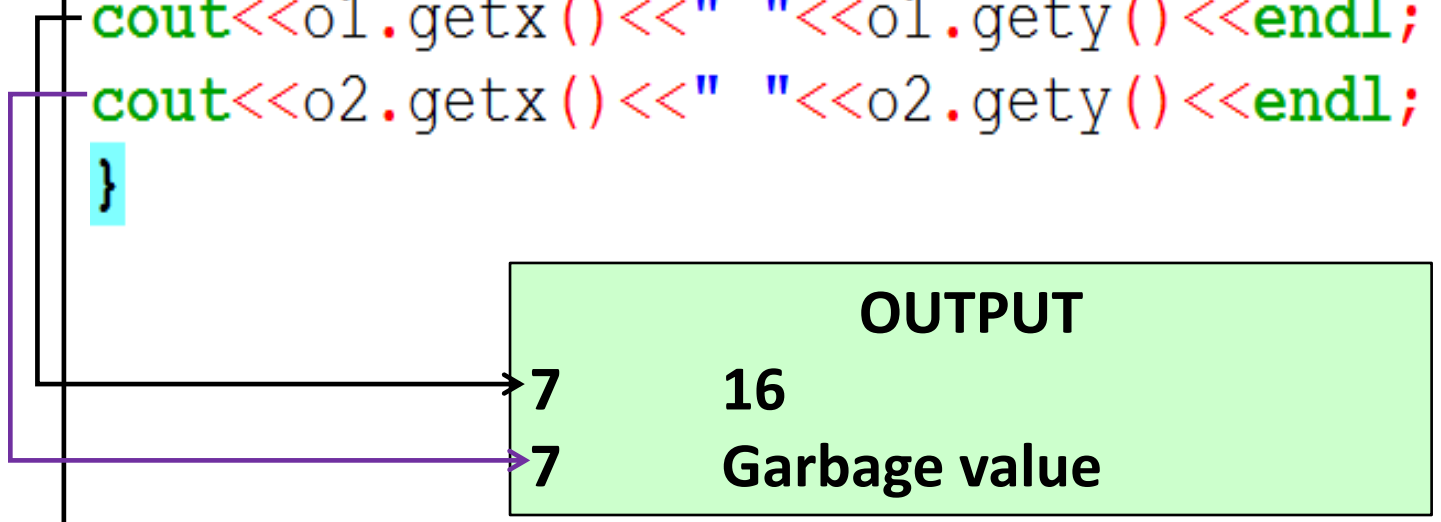
```
class myclass{
    static int x;
    int y;
public:
    void setx(int n){
        x = n; }
    void sety(int n){
        y = n; }
    int getx(){
        return x; }
    int gety(){
        return y; }
};
int myclass::x;
```

```
int main()
{
myclass o1, o2;
o1.setx(7);
o1.sety(16);
cout<<o1.getx()<<" "<<o1.gety()<<endl;
cout<<o2.getx()<<" "<<o2.gety()<<endl;
}
```

Example 3

```
class myclass{
    static int x;
    int y;
public:
    void setx(int n){
        x = n; }
    void sety(int n){
        y = n; }
    int getx(){
        return x; }
    int gety(){
        return y; }
};
int myclass::x;
```

```
int main()
{
myclass o1, o2;
o1.setx(7);
o1.sety(16);
cout<<o1.getx()<<" "<<o1.gety()<<endl;
cout<<o2.getx()<<" "<<o2.gety()<<endl;
}
```



Example 4

```
class myclass{
public:
    static int x;
    int y;
    void setx(int n){
        x = n; }
    void sety(int n){
        y = n; }
    int getx(){
        return x; }
    int gety(){
        return y; }
};
int myclass::x;
```

```
int main()
{
myclass o1, o2;
myclass :: x = 7;
o1.sety(10);
o2.sety(16);
cout<<o1.getx()<<" "<<o1.gety()<<endl;
cout<<o2.getx()<<" "<<o2.gety()<<endl;
}
```

Example 4

```
class myclass{
public:
    static int x;
    int y;
    void setx(int n){
        x = n; }
    void sety(int n){
        y = n; }
    int getx(){
        return x; }
    int gety(){
        return y; }
};
int myclass::x;
```

```
int main()
{
myclass o1, o2;
myclass :: x = 7;
o1.sety(10);
o2.sety(16);
cout<<o1.getx()<<" "<<o1.gety()<<endl;
cout<<o2.getx()<<" "<<o2.gety()<<endl;
}
```

OUTPUT

7 10

7 16

Example 6

```
class myclass{
public:
    static int x;
    int y;
    void setx(int n){
        x = n; }
    void sety(int n){
        y = n; }
    int getx(){
        return x; }
    int gety(){
        return y; }
};
int myclass::x;
```

```
int main()
{
myclass::x=7;
myclass o1, o2;
o1.sety(10);
o2.sety(16);
cout<<o1.getx()<<" "<<o1.gety()<<endl;
cout<<o2.getx()<<" "<<o2.gety()<<endl;
}
```

It is possible to use a static variable before creating any object of the class

Example 5

```
class myclass{
private:
    static int x;
    int y;
public:
    void setx(int n){
        x = n; }
    void sety(int n){
        y = n; }
    int getx(){
        return x; }
    int gety(){
        return y; }
};
int myclass::x  ;
```

```
int main()
{
myclass o1, o2;
myclass :: x = 7;
o1.sety(10);
o2.sety(16);
cout<<o1.getx()<<" "<<o1.gety()<<endl;
cout<<o2.getx()<<" "<<o2.gety()<<endl;
}
```

Compilation error

int myclass::x is private

Example 5

```
15          return x; }
16      int gety(){
17          return y; }
18  };
19  int myclass::x;
20
21  int main()
22  {
23  myclass o1, o2;
24  myclass :: x = 7;
25  o1.sety(10);
26  o2.sety(16);
27  cout<<o1.getx()<<"
```

Blocks | Search results | Cccc | Build log

	Line	Message
mming...	19	error: 'int myclass::x' is private
mming...	24	error: within this context

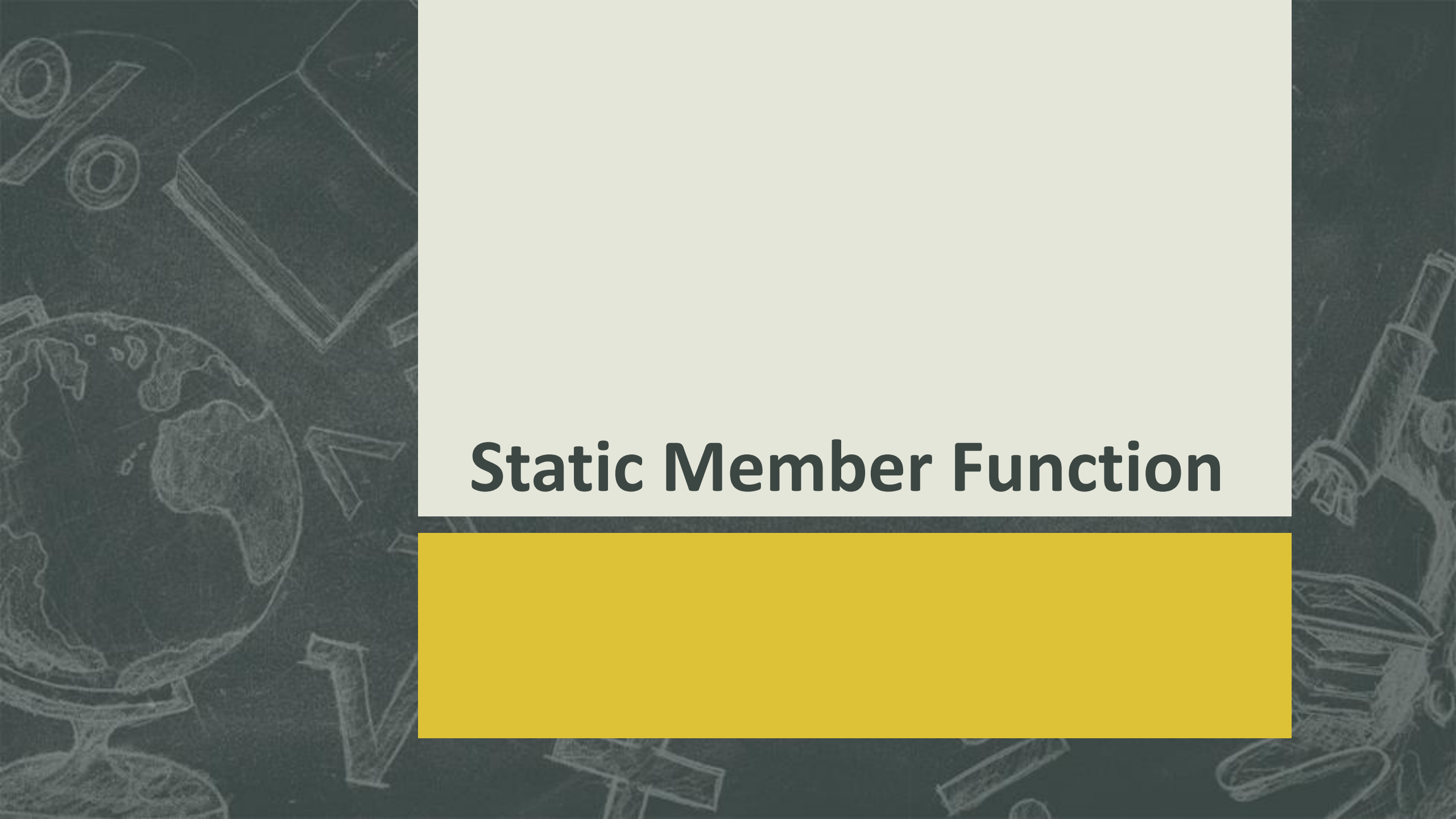
Example 5

```
class myclass{
private:
     static int x;
     int y;
public:
     void setx(int n){
          x = n; }
     void sety(int n){
          y = n; }
     int getx(){
          return x; }
     int gety(){
          return y; }
};
int myclass::x=8;
```

```
int main()
{
myclass o1, o2;

o1.sety(10);
o2.sety(16);
cout<<o1.getx()<<" "<<o1.gety()<<endl;
cout<<o2.getx()<<" "<<o2.gety()<<endl;
}
```

Initializing static variable within class is not allowed

Still, you can initialize here.



Static Member Function

Example 1

```
class myclass{
public:
    static void show(int n){
    cout<<"Value of n: "<<n<<endl;
    }
};
```

As static member functions are not attached to a particular object, they can be called directly by using the class name and the scope resolution operator.

```
int main()
{
myclass ob1;
ob1.show(10);
myclass::show(15);
}
```

OUTPUT

Value of n: 10

Value of n: 15

Example 2

```
class myclass{
public:
    static void show(int n){
    cout<<"Value of n: "<<n<<endl;
    }
};
```

It is possible to use a static function *before* any object instantiation

```
int main()
{
myclass::show(15);
myclass ob1;
ob1.show(10);
}
```

OUTPUT

Value of n: 15

Value of n: 10

Example 3

```
class myclass{
public:
    static int x;
    static void setx(int n){
        x=n;
    }
    int getx(){
        return x;
    }
};
int myclass::x;
```

```
int main()
{
myclass::setx(15);
myclass ob1;
cout<<ob1.getx()<<endl;
ob1.setx(10);
cout<<ob1.getx()<<endl;
}
```

OUTPUT

Value of n: 15

Value of n: 10

It is possible to use a static variable/function or global variable/ function inside a static function

Example 3

```
class myclass{
public:
        int x;
    static void setx(int n){
        x=n;
    }
    int getx(){
        return x;
    }
};
int myclass::x;
```

What will happen if non static member variable is accessed inside static function??

```
int main()
{
myclass::setx(15);
myclass ob1;
cout<<ob1.getx()<<endl;
ob1.setx(10);
cout<<ob1.getx()<<endl;
}
```

OUTPUT

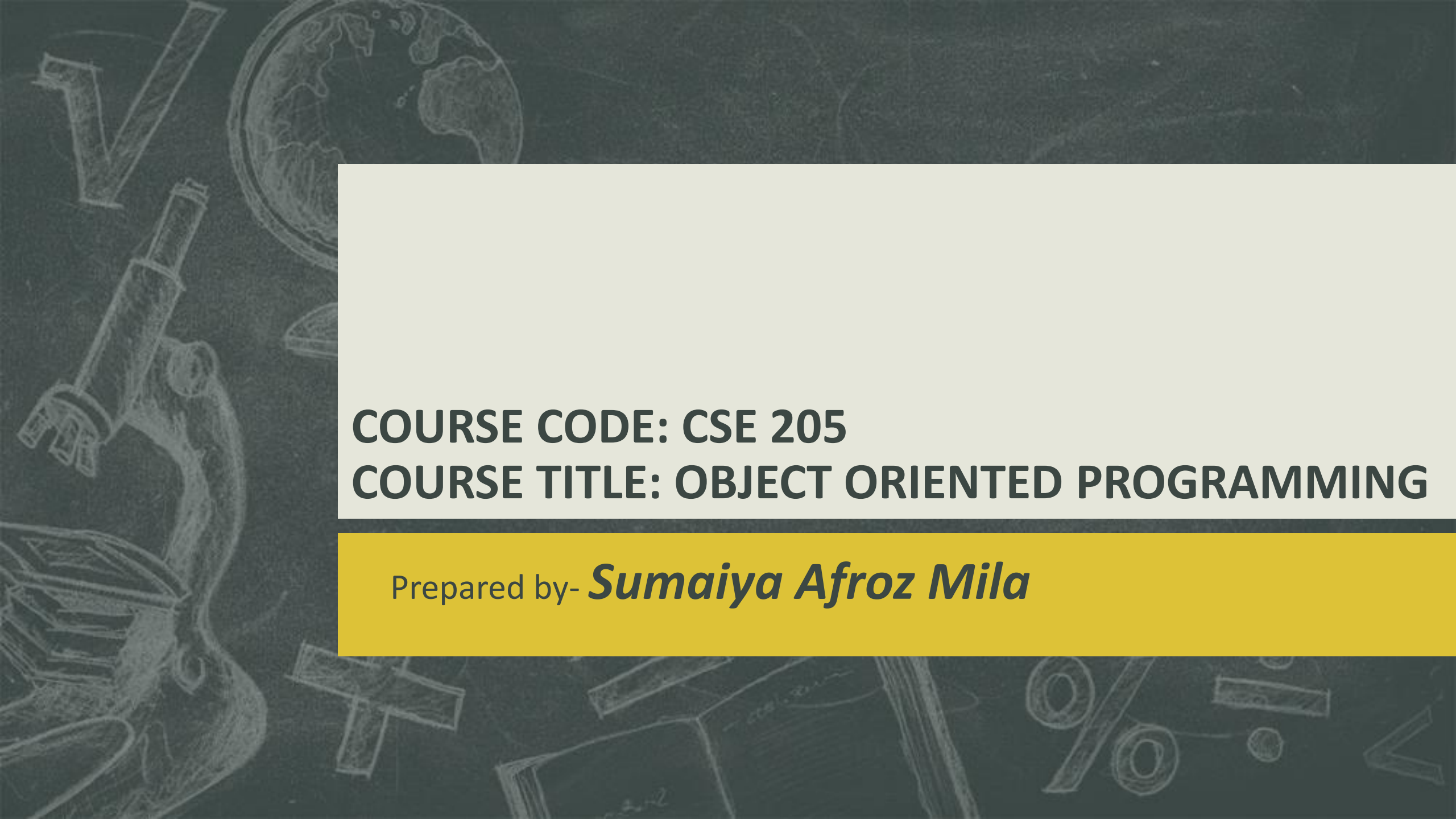
Value of n: 15

Value of n: 10

Static Member Function

- A static member function can access only other static members of its class.
- It can access non-static global data and functions
- Static member function does not have a “this” pointer
- Virtual static member functions are not allowed - inheritance
- A static member function can be invoked by an object of its class, or it can be called independent of any object. Via the class name and the scope resolution operator.


Thank
you!



COURSE CODE: CSE 205
COURSE TITLE: OBJECT ORIENTED PROGRAMMING

Prepared by- ***Sumaiya Afroz Mila***

Exception Example 1

```
double divide(double a, double b){
        double result=a/b;
         return result;
}
```

What if b=0??

```
main(){
   double a,b,c;
   cin>>a>>b;
   c=divide(a,b);
   cout<<c;
}
```

This is divide by zero exception

Exception Example 2

```
class Numlist{
  int* a;
  int size;
  public:
  Numlist(int s) {a=new int[s]; size=s; }
  void putAt(int index, int v){a[index]=v;}
  int getAt(int index){ return a[index];}
};
```

```
main(){
   Numlist list(100);          int index, val;
   cin>> index>>val;
   list.putAt(index,val);
   cout<< list.getAt(index);
}
```

What if

index<0 ?

index>100 ?

Array Index out of bound Exception

Exception Example 3

```
class Numlist{
  int* a;
  int size;
  public:
  Numlist(int s) {a=new int[s]; size=s; }
  void putAt(int index, int v){a[index]=v;}
  int getAt(int index){ return a[index];}
};
```

```
main(){
   Numlist list(100);          int index, val;
   cin>> index>>val;
   list.putAt(index,val);
   cout<< list.getAt(index);
}
```

What if

size> memory available?

Memory Allocation Exception

Exception Example 4

```
void manipList (Numlist* list, int op){
   int index;
   cin>>index;
   if (op==1){ int val; cin>>val;
                  list->putAt(index,val); }
   else if (op==2){cout<<list->getAt(index);}
   return;
}
```

```
main(){
   Numlist* list;
   int option;
   cout<<"Put/Get: ";
   cin>>option;
   manipList (list,option);
}
```

What if

list is not initialized?

Null Pointer Exception

Exception

- A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as, an attempt to **divide by zero**.
- Exceptions provide a way to transfer control from one part of a program to another.
- C++ exception handling is built upon three keywords: **try**, **catch**, and **throw**.

Exception

- **try:** program statements to be monitored for exceptions are contained in try block. A function called from within a try block can throw exception too
- **Throw:** If an exception occurs within the **try block**, it is thrown
- **Catch:** when an exception is thrown, it is caught using **catch,** and processed. There can be more than one catch statement associated with a try

```
try{
        //try block
        throw exception;
}
catch(type1 arg){
        //catch block
}
catch(type2 arg){
        //catch block
}

catch(typeN arg){
        //catch block
}
```

throw statement

- throw is used to signal the fact that an exception has occurred; also called *raise*
- Can throw any valid expression/ object.
- **Syntax:**

  throw expression;

Default catch statement

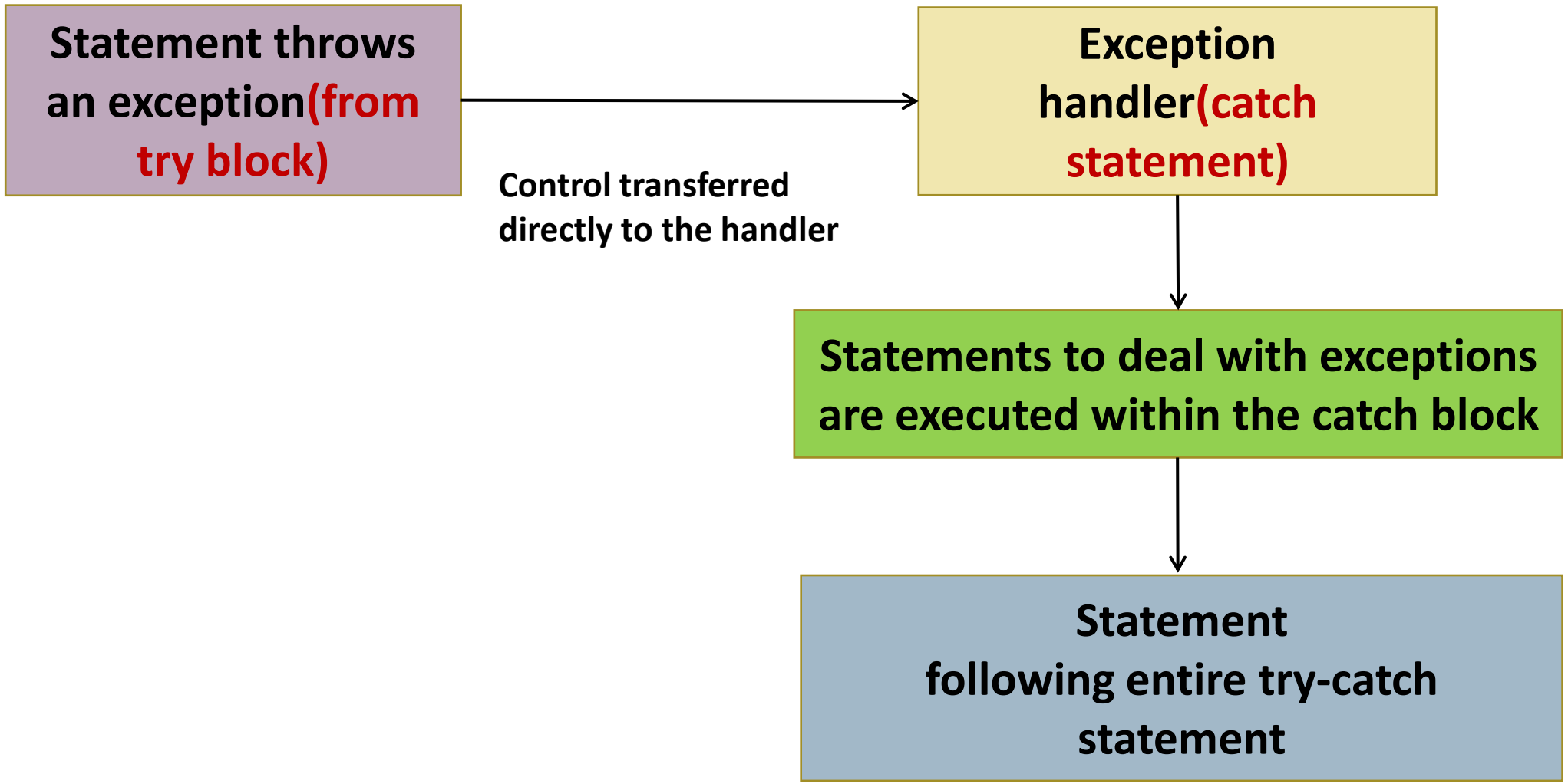
- If an exception is thrown for which there is no applicable catch statement, an abnormal program termination might occur
- In some circumstances you will want an exception handler to catch all exceptions instead of just a certain type
- To solve above situations, we can use default **catch statement**
- **Syntax:**

```
catch(...){
// process all exceptions
}
```

Execution of try-catch

- Once exception is thrown, **control passed** to the **catch** expression and **try block is terminated**
- **Catch** is not called
- Program execution is transferred to it
- After the execution of the catch statement, program continues with the **statement following** the **catch**

Execution of try-catch



Exception

An exception is an unusual, often unpredictable event, detectable by software or hardware, that requires special processing occurring at runtime.

- If without handling,
  - Program crashes
  - Falls into unknown state

Using try-catch

```
main(){
        cout<<“start\n”;
        try{
                cout<<“Inside try block”<<endl;
                throw 10;
                 cout<<“This will not be executed”<<endl;
        }
        catch(int i ){
                cout<<“exception occurred”<<endl;
                cout<<“number is: ”<<i<<endl;
        }
}
```

OUTPUT

start
Inside try block
exception occurred
number is: 10

Using try-catch

```
main(){
        cout<<“start\n”;
        try{
                cout<<“Inside try block”<<endl;
                throw 10;
                 cout<<“This will not be executed”<<endl;
        }
        catch(double i ){
                cout<<“exception occurred”<<endl;
                cout<<“number is: ”<<i<<endl;
        }
}
```

OUTPUT

start
Inside try block
Abnormal Program Termination

This program produces such output because the **integer exception** will not be caught by a **double catch statement**

SOLUTION??

Write another catch statement with integer argument

Exception thrown from a stmt outside try block

```
void Xtest(int test){
    cout<<"inside Xtest: "<<test<<endl;
    if(test>0)
        throw test;
}
```

```
int main (){
    cout<<"start\n";
    try{
        cout<<"inside a try block\n";
        Xtest(0);
        Xtest(1);
        Xtest(2);
    }
    catch(int i){
        cout<<"caught number: “<<i<<endl;
     }
            cout<<“end”;
}
```

Exception Handling

- An exception handler is a section of program code that is designed to execute when a particular exception occurs
    - Resolve the exception
    - Lead to known state, such as exiting the program

Exception Handling(divide by zero exception)

```
double divide(double a,
                    double b){
   if(b!=0){
        double result=a/b;
        return result;
   }
   else{
      throw  -1;
  }
}
```

```
main(){
   double a,b,c;
   cin>>a>>b;
   try{
            c=divide(a,b);
            cout<<c;
   }
   catch (int i){
       cout<<"Division Exception: "<<i;
   }
}
```

Exception Handling(divide by zero exception)

```
double divide(double a,
                    double b){
    if(b!=0){
          double result=a/b;
          return result;
    }
    else{
       throw  "Divide by
zero";
   }
}
```

```
main(){
   double a,b,c;
   cin>>a>>b;
   try{
            c=divide(a,b);
            cout<<c;
   }
   catch (const char* message){
       cout<<"Exception: ";
       puts(message);
   }
}
```

Exception Handling

```
class MyException{
    char message[30];
    public:
    MyException ( char* m){
                strcpy (message, m);
     }
  char* getMessage(){ return message; }
};
```

```
double divide(double a,  double b){
   if(b!=0){ double result=a/b;
                return result;  }
  else{
     MyException Ex("Divide By Zero");
     throw  Ex ;    }
}
```

```
main(){
   double a,b,c;
   cin>>a>>b;
   try{
                c=divide(a,b);
                cout<<c;
   }
   catch (----------? e){
       cout << "Code: ";
       cout << e.getMessage();
   }
}
```

Exception Handling

```
class MyException{
    char message[30];
    public:
    MyException ( char* m){
                strcpy (message, m);
     }
  char* getMessage(){ return message; }
};
```

```
double divide(double a,  double b)
{
   if(b!=0){ double result=a/b;
                return result;  }
  else{
     MyException Ex("Divide By Zero");
     throw  Ex ;    }
}
```

```
main(){
   double a,b,c;
   cin>>a>>b;
   try{
                c=divide(a,b);
                cout<<c;
   }
   catch (MyException e){
       cout << "Code: ";
       cout << e.getMessage();
   }
}
```

Anonymous object

- In certain cases, we need a variable only temporarily.

- This is done by creating objects like normal, but omitting the variable name.

Example:

```
ComplexNumber A(10, 2)          //normal object
ComplexNumber(10, 2)    // Anonymous Object
```

Multiple Exception

```
#define MAX 3
class StackFull { };
class StackEmpty{ };
class Stack
{
      int S[MAX], top;
      public:
    Stack () { top= -1;}
    void push(int v){
        if (top>= MAX-1)
            throw StackFull();
        S[++top] = v;    }
    int pop(){
        if (top<0)
            throw StackEmpty();
        return S[top--];     }
};
```

```
int main(){
    Stack st;
try{
    st.push(10);
    st.push(20);
    st.push(30);
    st.push(40);
cout<<st.pop()<<endl;    cout<<st.pop()<<endl;
cout<<st.pop()<<endl;    cout<<st.pop()<<endl;
}
catch(StackFull  F){
    cout<<"Exception: Stack is Full" <<endl;
catch(StackEmpty  E){
    cout<<"Exception: Stack is Empty" <<endl;
}
```

Nested try catch

```
try{
        try{
                try{
                }
                catch (dataType varname){
                        //Exception Handler
                }
        }
        catch (dataType varname){
                        //Exception Handler
        }

}
catch (...){
                //Default Exception Handler
}
```

Rethrow an exception

- An exception is rethrown to allow multiple handlers access to the exception
- Perhaps, one exception handler manages one aspect of an exception, and a second handler copes with another
- An exception can **only** be **rethrown** from within a **catch block(or from any function called from within that block)**
- When an exception is rethrown, it is not recaught by the same **catch** statement
- It will propagate to an outer catch exception

Rethrow an exception example

```
void Xhandler(){
    try{
        throw "hello";
    }
    catch(const char *c){
    cout<<"caught char* inside
xhandler\n";
        throw;
    }
}
```

```
int main (){
    cout<<"start\n";
    try{
        Xhandler();
    }
    catch(const char *c){
    cout<<"caught char* inside main\n";
    }
    cout<<"end";
}
```

Restricting Exception in a Function

- You can restrict the type of exceptions that a function can throw back to its caller
- You can in fact prevent a function from throwing any exception
- To do so, a **throw** clause is to be added to the function definition

```
Ret-type func_name(arg list)throw(type list)
{
        //body
}
```

Restricting Exception in a Function

```
// can throw anything
void func();
```

```
// promises not to throw
void func() throw();
```

```
// promises to only throw int
void func() throw(int);
```

```
// throws only char or int
void func() throw(char,int);
```

By default, a function can throw anything it wants.

- A throw clause in a function's signature
  - •Limits what can be thrown
  - •A promise to calling function
  - •If other type thrown, standard library function **unexpected()** is called
- A throw clause with no types
  Says nothing will be thrown
- Can list multiple types
  Comma separated

Source of Exceptions

- Exceptions Thrown by user code, using *throw* statement.

- Exceptions Thrown by the Language

  for example while using: new

- Exceptions Thrown by Standard Library Routines

Exception Thrown by New

```
#include<iostream>
#include<new>
using namespace std;
int main(){
        int *p;
        try{
                p = new int [100000];
        }
        catch (bad_alloc  xa){
                cout<<“allocation Failure”;
        }
        delete [ ] p;
}
```

Good Programming Style with C++ Exceptions

-Don’t use exceptions for normal program flow

- Only use where normal flow isn’t possible

- Don’t let exceptions leave main or constructors

- Violates “normal” initialization and termination

- Always throw some type

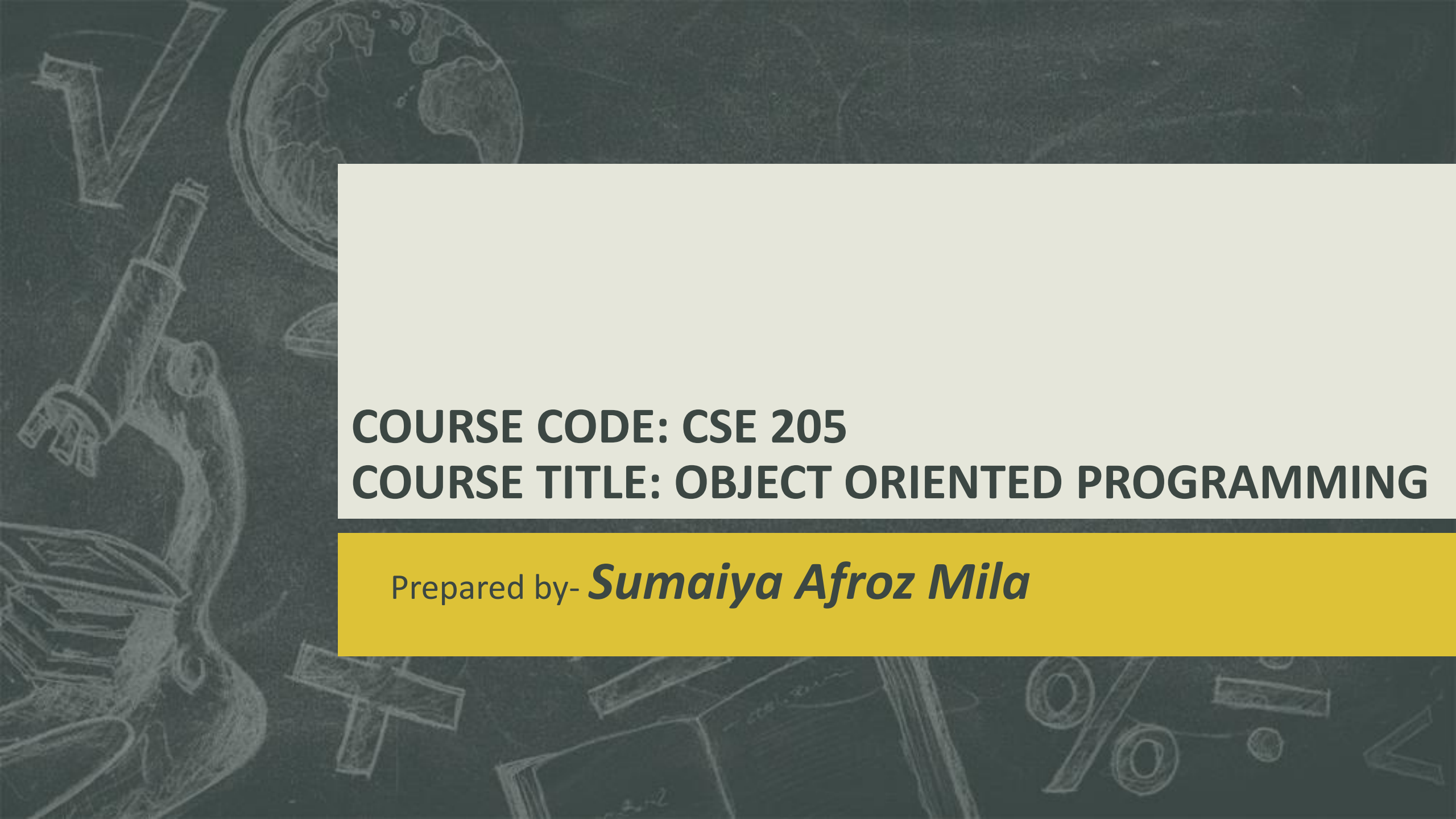
- So the exception can be caught

- Resources may have been allocated when exception thrown. Catch handler should delete space allocated by new and close any opened files

- Use exception specifications widely

- Helps caller know possible exceptions to catch


Thank
you!



COURSE CODE: CSE 205
COURSE TITLE: OBJECT ORIENTED PROGRAMMING

Prepared by- ***Sumaiya Afroz Mila***

An Old Example in C

```
int abs (int n){
  return (n < 0) ? -n : n;
}
long labs (long n){
  return (n < 0) ? -n : n;
}
float fabs (float n){
  return (n < 0) ? -n : n;
}
```

```
main(){
  int i=-3;
  cout<<abs (i);

  long l=-7;
  cout<< labs (l);

  float f=-2.0;
  cout<<fabs (f);
}
```

An Old Example in C++

```
int abs (int n){
  return (n < 0) ? -n : n;
}
long abs (long n){
  return (n < 0) ? -n : n;
}
float abs (float n){
  return (n < 0) ? -n : n;
}
```

```
main(){
  int i=-3;
  cout<<abs (i);

  long l=-7;
  cout<< abs (l);

  float f=-2.0;
  cout<<abs (f);
}
```

Using C++ Template

Generic Function/ Template Function

```
template <class T>  T abs(T n){
    return (n < 0) ? -n : n;
}
```

```
main(){
   int i=-3;
   cout<<abs (i);

   long l=-7;
   cout<< abs (l);

   float f=-2.0;
   cout<<abs (f);
 }
```

Generic Function

- Programming that works regardless of type is called generic programming.
- A generic function defines a general set of operations that will be applied to various types of data.
- Another type of polymorphism, known as parametric polymorphism.
- Give the user the ability to reuse code in a simple, type-safe manner that allows the compiler to automate the process of type *instantiation*.

Function Template Syntax

- A generic function is created using the keyword **template**.
- Also called Template Function

```
template <class identifier>return_type function_name (arguments of type identifier)
{ … }
```

```
template <class identifier>

return_type function_name (arguments of type identifier) { … }
```

- If Second form is used, no other statement can occur between the template statement and the start of the generic function definition
- Function return type does not take into account to select the template function (just like function overloaded).

Function Template Syntax

- Instead of using the keyword ***class,*** keyword ***typename*** can also be used to specify a generic type in a template definition

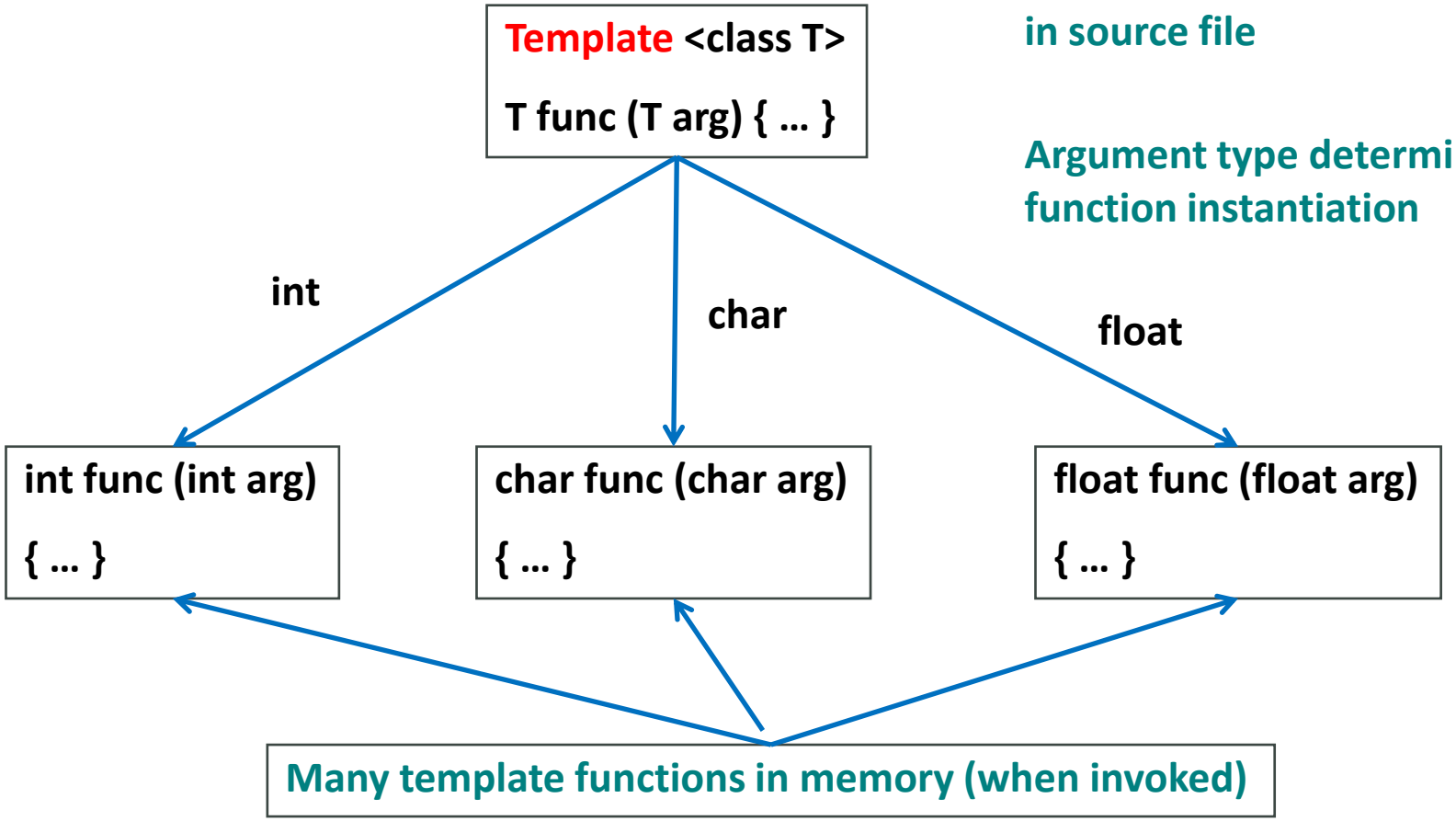
```
template <class identifier>
return_type function_name (arguments of type identifier) { … }
```

```
template <typename identifier>
return_type function_name (arguments of type identifier) { … }
```

What Compiler does?

- Function template does not cause the compiler to generate any code (similar to the way a class does not create anything in memory).
- Compiler simply remembers the template for possible future use.
- Code generation does not take place until the function is actually called (invoked).
- Compiler generate a template function (specific version of a function template) depending on the function signature (that's why we called parametric polymorphism).

Function Template Example



One function template in source file

Argument type determines function instantiation

Function Template with Multiple Arguments

```
template<class T1, class T2> void copy(T1 a[], T2 b[], int n){ //two arguments template
        for(int i=0;  i<n; i++)
                a[i] = b[i];
}
```

```
main(){
  char c[50] = {'a', 'b', 'd', 'e', 'f'};
  int i[50] = {10, 20 30, 40, 50};
  float f[50] = {1.1, 2.2, 3.3, 4.4, 5.5};
  copy (i,f,5);     // ok, but loss of data
  copy (c,f,5);   // ok, but loss of data
  copy (f,i,5);   // ok, int to float
  copy (f,c,5);  //ok, char to float
}
```

Function Template with Multiple Arguments

```
template <class T> void swap( T& x, T& y){
    T tmp;
    tmp = x;
    x = y;
    y = tmp;
}
```

```
main(){
     int x, y;
    char c;
    double m, n;
    swap( x, y );
    swap( m, n );
    swap( x, m ); // error
}
```

Generic Function vs Function Overloading

- When functions are overloaded, one can have different actions performed within the body of each function but **generic function must perform the same general version for all actions.**

- A generic function can be overloaded. In that case the overloaded function overrides the generic function relative to that specific version.

Generic Function vs Function Overloading

```
template <class X>
 void func(X n){
      cout << n << endl;
}
```

```
template <class X, class Y>
 void func(X a, Y b){
      cout <<a<<"  " << b << endl;
}
```

```
main(){
      func(10);
      func(20, 30);
}
```

Generic Function vs Function Overloading

```
template <class X>
 void func(X n){
     cout << “Inside Generic Function” << endl;
}
```

```
int func(int n){
     cout<<“Inside Specific Function: ”<<n;
}
```

```
main(){
          int i=10;
          func(i);
          double d=5.5;
          func(d);
}
```

Generic Class

- A generic class defines the algorithm used by the class but the actual type of data is specified when object of that class are created.

- The *identifier* is a template argument that essentially stands for an arbitrary type.

- The template argument can be used as a type name throughout the class definition.

```
template < class identifier >

class classname { ... };
```

```
classname<type> ob; //Object Instantiation
```

Generic Class

```
// template stack implementation
template <class TYPE>
class stack {
  TYPE*  s;
  int  top;
  public:
   stack (int size =100){
         s = new TYPE[size];
         top=0;
    }
   ~stack() { delete []s; }
    void  push (TYPE c){
          s[top++] = c;   }
   TYPE  pop(){
          return s[--top];  }
};
```

```
void main(){

 //100  char stack
 stack<char>    stk_ch;

 //200   int stack
 stack<int>   stk_int(200);

 //20  float stack
 stack<float>   stk_float(20);
}
```


Thank
you!