

## Week 03

- Inline Functions
- Separation of Interface & Implementations
- Constructors
- Constructor Overloading
- Member Initialization in Constructors

## Lecture 07

- Interface & Implementation
- Inline Functions
- Separation of Interface & Implementation

## Object has an Interface

- An object encapsulates data and behavior
- So how objects interact with each other?
- Each object provides an interface (operations)
- Other objects communicate through this interface

## Example – Interface of a Phone

- Screen
- Keypad
- Mic
- Ear piece
- Functions:
  - Input Number
  - Place Call
  - Disconnect Call
  - Add number to address book
  - Remove number
  - Update number

## Implementation

- Provides services offered by the object interface
- This includes
  - Data structures to hold object state
  - Functionality that provides required services

## Example

```
class Complex{ //old
    float x;
    float y;
public:
    void setNumber(float i, float j){
        x = i;
        y = j;
    }
    ...
};
```

## Review

- Class
  - Concept
  - Definition
- Data members
- Member Functions
- Access specifier

## Member Functions

- Member functions are the functions that operate on the data encapsulated in the class
- Public member functions are the interface to the class

## Member Functions (contd.)

- Define member function inside the class definition
- OR
- Define member function outside the class definition
    - But they must be declared inside class definition

## Function Inside Class Body

```
class ClassName {  
    ...  
    public:  
    ReturnType FunctionName() {  
        ...  
    }  
};
```

# Example

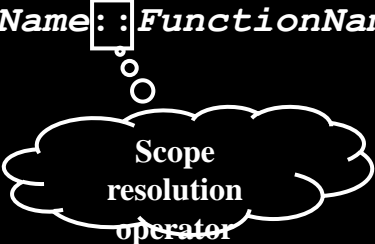
- Define a class of student that has a roll number. This class should have a function that can be used to set the roll number

## Example

```
class Student{
    int rollNo;
public:
    void setRollNo(int aRollNo){
        rollNo = aRollNo;
    }
};
```

## Function Outside Class Body

```
class ClassName{
    ...
    public:
        ReturnType FunctionName();
};
ReturnType ClassName::FunctionName()
{
    ...
}
```



## Example

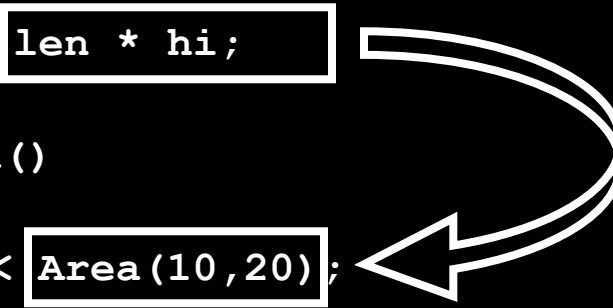
```
class Student{
    ...
    int rollNo;
public:
    void setRollNo(int aRollNo);
};
void Student::setRollNo(int aRollNo){
    ...
    rollNo = aRollNo;
}
```

## Inline Functions

- Instead of calling an inline function compiler replaces the code at the function call point
- Keyword 'inline' is used to request compiler to make a function inline
- It is a request and not a command

## Example

```
inline int Area(int len, int hi)
{
    return len * hi;
}
int main()
{
    cout << Area(10,20);
}
```



The diagram illustrates the inlining process. A box highlights the expression `len * hi;` within the `Area` function's return statement. A curved arrow originates from this box and points to another box that highlights the function call `Area(10,20);` in the `main` function. This visualizes the compiler replacing the function call with the actual code of the function.



## Inline Functions

- If we define the function inside the class body then the function is by default an inline function
- In case function is defined outside the class body then we must use the keyword 'inline' to make a function inline

## Example

```
class Student{  
    int rollNo;  
public:  
    void setRollNo(int aRollNo){  
        ...  
        rollNo = aRollNo;  
    }  
};
```

## Example

```
class Student{
    ...
    public:
        inline void setRollNo(int aRollNo);
};
void Student::setRollNo(int aRollNo){
    ...
    rollNo = aRollNo;
}
```

## Example

```
class Student{
    ...
    public:
        void setRollNo(int aRollNo);
};
inline void Student::setRollNo(int
                                aRollNo){
    ...
    rollNo = aRollNo;
}
```

## Example

```
class Student{
    ...
    public:
        inline void setRollNo(int aRollNo);
};
inline void Student::setRollNo(int
                                aRollNo) {
    ...
    rollNo = aRollNo;
}
```

## Tradeoffs of Inline vs. Regular Member Functions

- When a regular function is called, control passes to the called function
  - the compiler stores return address of call, allocates memory for local variables, etc.

- Code for an inline function is copied into the program in place of the call when the program is compiled
  - larger executable program, but
  - less function call overhead, possibly faster execution

## Separation of Interface & Implementation

- Means change in implementation does not effect object interface
- This is achieved via principles of information hiding and encapsulation

## Example – Separation of Interface & Implementation

- A driver can drive a car independent of engine type (petrol, diesel)
- Because interface does not change with the implementation

## Example – Separation of Interface & Implementation

- A driver can apply brakes independent of brakes type (simple, disk)
- Again, reason is the same interface
- A user can call independent of vendor from a phone.

## Advantages of Separation

- Users need not to worry about a change until the interface is same
- Low Complexity
- Direct access to information structure of an object can produce errors

## Messages

- Objects communicate through messages
- They send messages (stimuli) by invoking appropriate operations on the target object
- The number and kind of messages that can be sent to an object depends upon its interface

## Examples – Messages

- A Person sends message (stimulus) “stop” to a Car by applying brakes
- A Person sends message “place call” to a Phone by pressing appropriate button

## Separation of interface and implementation

- Public member function exposed by a class is called interface
- Separation of implementation from the interface is good software engineering

## Complex Number

- There are two representations of complex number
  - Euler form
    - $z = x + i y$
  - Phasor form
    - $z = |z| (\cos \theta + i \sin \theta)$
    - $|z|$  is known as the complex modulus and  $\theta$  is known as the complex argument or phase

## Example

### Old implementation

Complex
☞ float x
☞ float y
float getX()
float getY()
void setNumber (float i, float j)
...

### New implementation

Complex
☞ float z
☞ float theta
float getX()
float getY()
void setNumber (float i, float j)
...



## Example

```
class Complex{ //new
    float z;
    float theta;
public:
    void setNumber(float i, float j){
        theta = arctan(j/i);
        ...
    }
    ...
};
```

## Advantages

- User is only concerned about ways of accessing data (interface)
- User has no concern about the internal representation and implementation of the class

## Separation of interface and implementation

- Usually functions are defined in implementation files (.cpp) while the class definition is given in header file (.h)
- Some authors also consider this as separation of interface and implementation

### Student.h

```
class Student{  
    int rollNo;  
public:  
    void setRollNo(int aRollNo);  
    int getRollNo();  
    ...  
};
```

## Student.cpp

```
#include "student.h"

void Student::setRollNo(int aNo){
    ...
}
int Student::getRollNo(){
    ...
}
```

## Driver.cpp

```
#include "student.h"

int main(){
    Student aStudent;
}
```

# Review

## Lecture 08

## Constructor

## Constructor

- Constructor is used to initialize the objects of a class
- Constructor is used to ensure that object is in well defined state at the time of creation
- Constructor is automatically called when the object is created
- Constructor are not usually called explicitly

## Constructor (contd.)

- Constructor is a special function having same name as the class name
- Constructor does not have return type
- Constructors are commonly public members

## Example

```
class Student{
    ...
public:
    Student(){
        rollNo = 0;
        ...
    }
};
```

## Example

```
int main()
{
    Student aStudent;
    /*constructor is implicitly
    called at this point*/
}
```

## Two Types of Constructors

- Default Constructor
- Parametrized Constructor

## Default Constructor

- Constructor without any argument is called default constructor
- If we do not define a default constructor the compiler will generate a default constructor
- This compiler generated default constructor initialize the data members to their default values

## Example

```
class Student
{
    int rollNo;
    char *name;
    float GPA;
public:
    ...    //no constructors
};
```

## Example

Compiler generated default constructor

```
{
    rollNo = 0;
    GPA = 0.0;
    name = NULL;
}
```




## Default Constructor

```
class addition
{
    int a, b;
    public:
        addition() {
            a = 100;
            b = 175;
        }
}
```

## Default Constructor Example

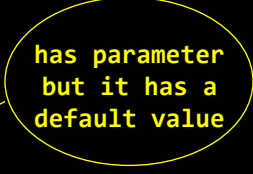
```
class Rectangle
{
    private:
        double length;
        double width;
    public:
        Rectangle() // default constructor
        {
            length = 1;
            width = 1;
        }
    // Other member functions goes here
};
```



## Another Default Constructor Example

```
class Rectangle
{
    private:
        double length;
        double width;
    public:
        Rectangle(double l=1, double w=1)
        // default constructor
        {
            length = l;
            width = w;
        }
        // Other member functions goes here
};
```

has parameter  
but it has a  
default value



## Parameterized Constructor

- A constructor with at least one parameter is called a parametrized constructor.
- The advantage of a parametrized constructor is that you can initialize each instance of the class to different values.

## Constructor Overloading

- A class can have more than 1 constructor
- Overloaded constructors in a class must have different parameter lists

```
Rectangle();  
Rectangle(double, double);
```

## Parameterized Constructor

```
class paraconstructor  
{  
    private: int a, b;  
  
    public:  
        paraconstructor (int x, int y) {  
            a = x;  
            b = y;  
        }  
}
```

## Example

```
class Student{
...
public:
    Student();
    Student(char * aName);
    Student(char * aName, int aRollNo);
    Student(int aRollNo, int aRollNo,
            float aGPA);
};
```

## Example

```
Student::Student(int aRollNo,
                 char * aName){
    if(aRollNo < 0){
        rollNo = 0;
    }
    else {
        rollNo = aRollNo;
    }
    ...
}
```

## Example

```
int main()  
{  
    Student student1;  
    Student student2 ("Name");  
    Student student3 ("Name", 1);  
    Student student4 ("Name", 1, 4.0);  
}
```

## Constructor Overloading

- Use default parameter value to reduce the writing effort

## Example

```
Student::Student(    char * aName = NULL,
                    int  aRollNo= 0,
                    float aGPA = 0.0){
```

...

```
}
```

Is equivalent to

```
Student();
```

```
Student(char * aName);
```

```
Student(char * aName, int aRollNo);
```

```
Student(char * Name, int aRollNo, float
        aGPA);
```

## Member Initialization in Constructor

- When a constructor is used to initialize other members, these other members can be initialized directly, without resorting to statements in its body.
- This is done by inserting, before the constructor's body, a colon (:) and a list of initializations for class members.

## Example - Member Initialization in Constructor

```
class Rectangle{  
private:  
    double length, width;  
  
public:  
    Rectangle();  
    Rectangle (double,double);  
    double getArea ();  
};
```

## Member Initialization in Constructor

- The constructor for this class could be defined, as usual, as:

```
Rectangle::Rectangle(double l,double w) {  
    length=l;  
    width=w;  
}
```

## Member Initialization in Constructor

- It could also be defined using *member initialization* as:

```
Rectangle::Rectangle (double l,  
double w): length(l){  
width = w;  
}
```

## Member Initialization in Constructor

- Or even:

```
Rectangle::Rectangle (double l,  
double w): length(l),width(w)  
{}
```

[Note how in this last case, the constructor does nothing else than initialize its members, hence it has an empty function body.]



```
class MyClass {  
    public:  
    MyClass();  
    private:  
    int x;  
};  
MyClass::MyClass() :  
    x(100) { }
```

```
int main() {  
    MyClass m;  
}
```

## Order of Initialization

- ▶ Data member are initialized in order they are declared
- ▶ Order in member initializer list is not significant at all

## Example

```
class ABC{  
    int x;  
    int y;  
    int z;  
public:  
    ABC();  
};
```

## Example

```
ABC::ABC() : y(10), x(y), z(y)  
{  
    ...  
}  
/* x = garbage value  
   y = 10  
   z = 10 */
```

## Class Activity

- Define a class called Pizza that has member variables to track the type of pizza (either deep dish, hand tossed, or pan) along with the size (either small, medium, or large) and the number of pepperoni or cheese toppings. Include mutator and accessor functions for your class. Create a void function, `outputDescription()`, that outputs a textual description of the pizza object. Also include a function, `computePrice()`, that computes the cost of the pizza and returns it as a double according to the following rules:
- Small pizza = \$10 + \$2 per topping
- Medium pizza = \$14 + \$2 per topping
- Large pizza = \$17 + \$2 per topping
- Write a suitable test program that creates and outputs a description and price of various pizza objects.
- 

## Review