Object Oriented Programming

Lecture 3

Dr. Naveed Anwar Bhatti

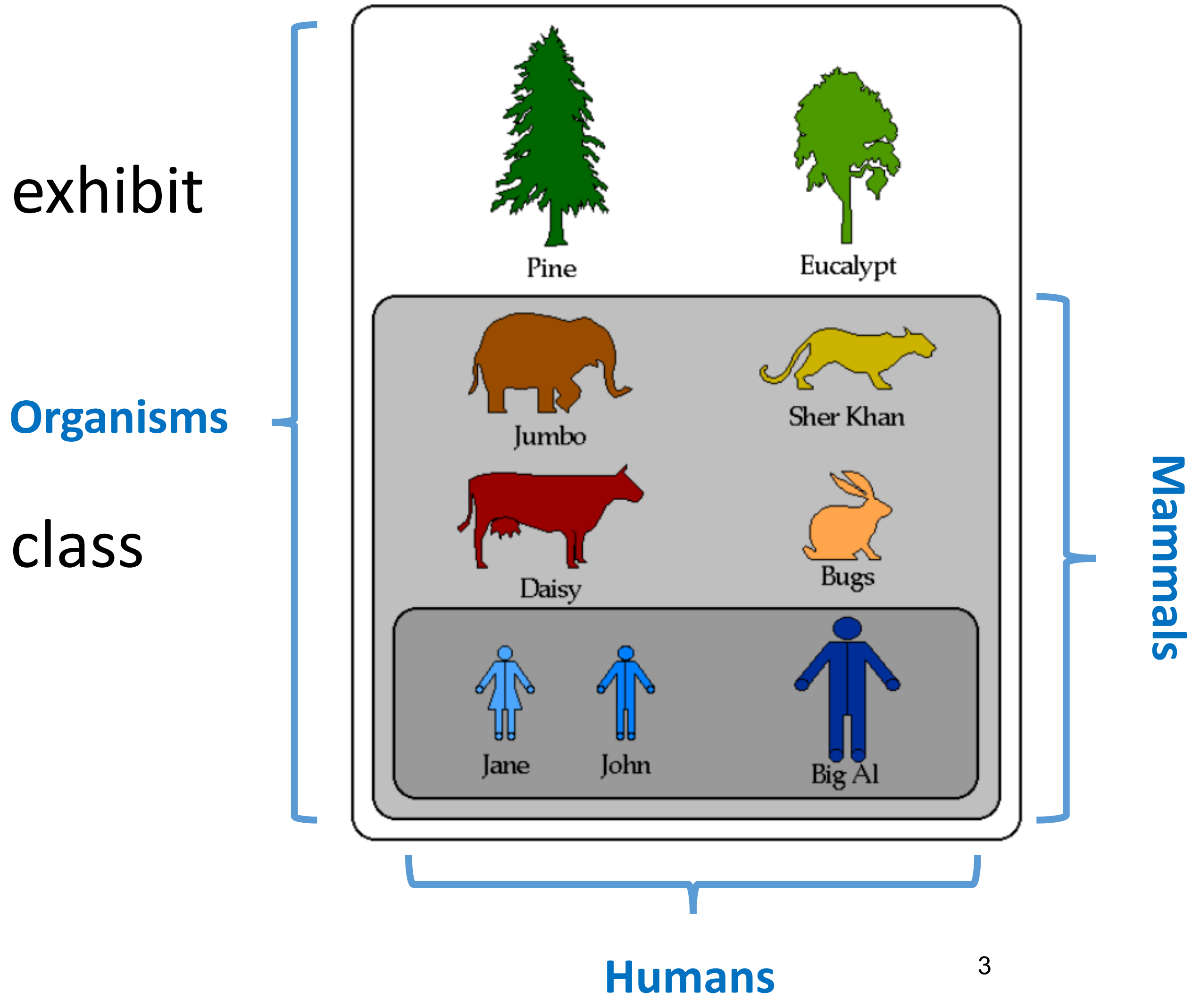
Webpage: naveedanwarbhatti.github.io

Object-Oriented Programming in C++

Classes

- In an OO model, some of the objects exhibit identical characteristics (information structure and behavior)

- We say that they belong to the same class



Few More Examples – Class

- **Ali** studies mathematics
- **Anam** studies physics
- **Sohail** studies chemistry

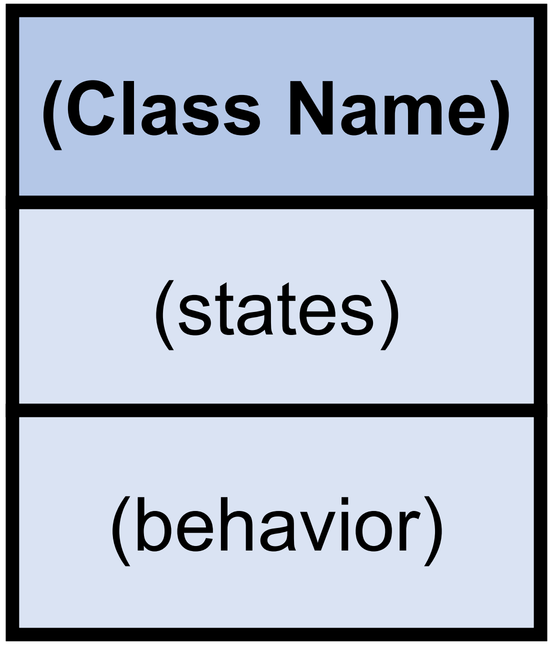
- Each one is a **Student**
- We say these **objects** are ***instances*** of the **Student** class

Few More Examples – Class

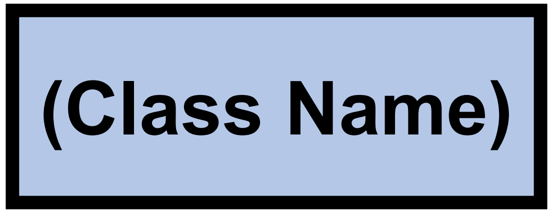
- **Ahsan** teaches mathematics
- **Aamir** teaches computer science
- **Atif** teaches physics

- Each one is a **Teacher**
- We say these **objects** are ***instances*** of the **Teacher** class

Graphical Representation of Classes

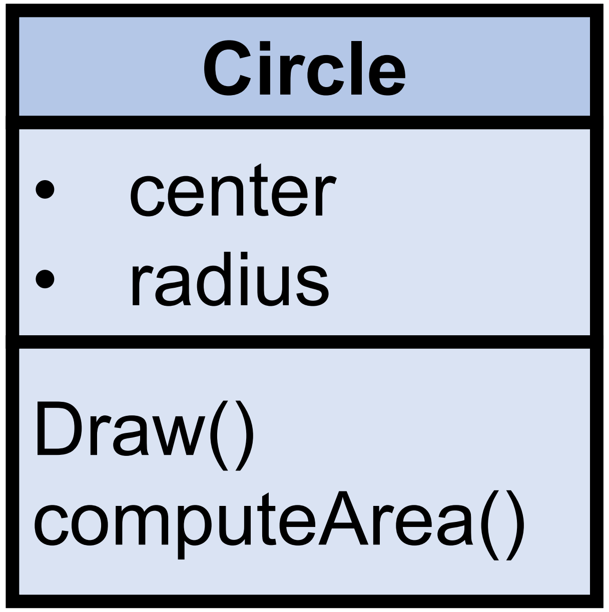


Normal Form

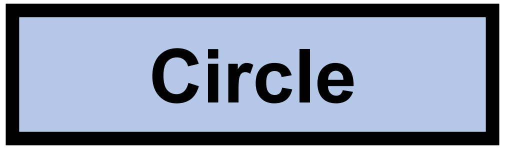


Suppressed Form

Example - Graphical Representation of Classes

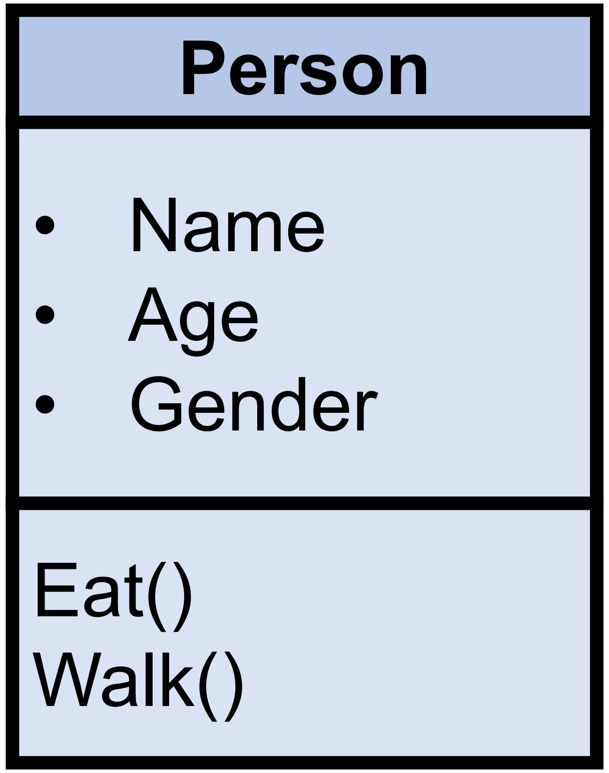


Normal Form

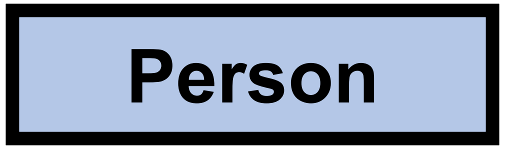


Suppressed Form

Example - Graphical Representation of Classes



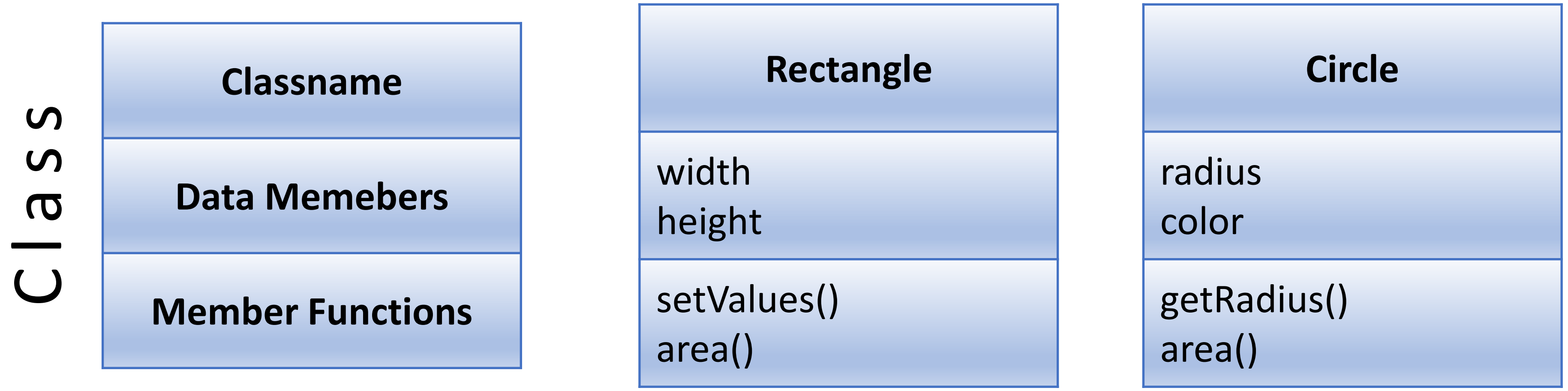
Normal Form



Suppressed Form

Class

- What is *Class*?
  - Class is a blueprint from which individual objects are created
  - An expanded concept of data structures: like data structures, they can contain data members, but they can **also contain functions** as members.



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

- 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

- Member functions are the functions that operate on the data **encapsulated** in the class

- Public member functions are the **interface** to the class

????

- Class members (both ***data*** and ***functions***)can restrict their access through ***access specifiers***

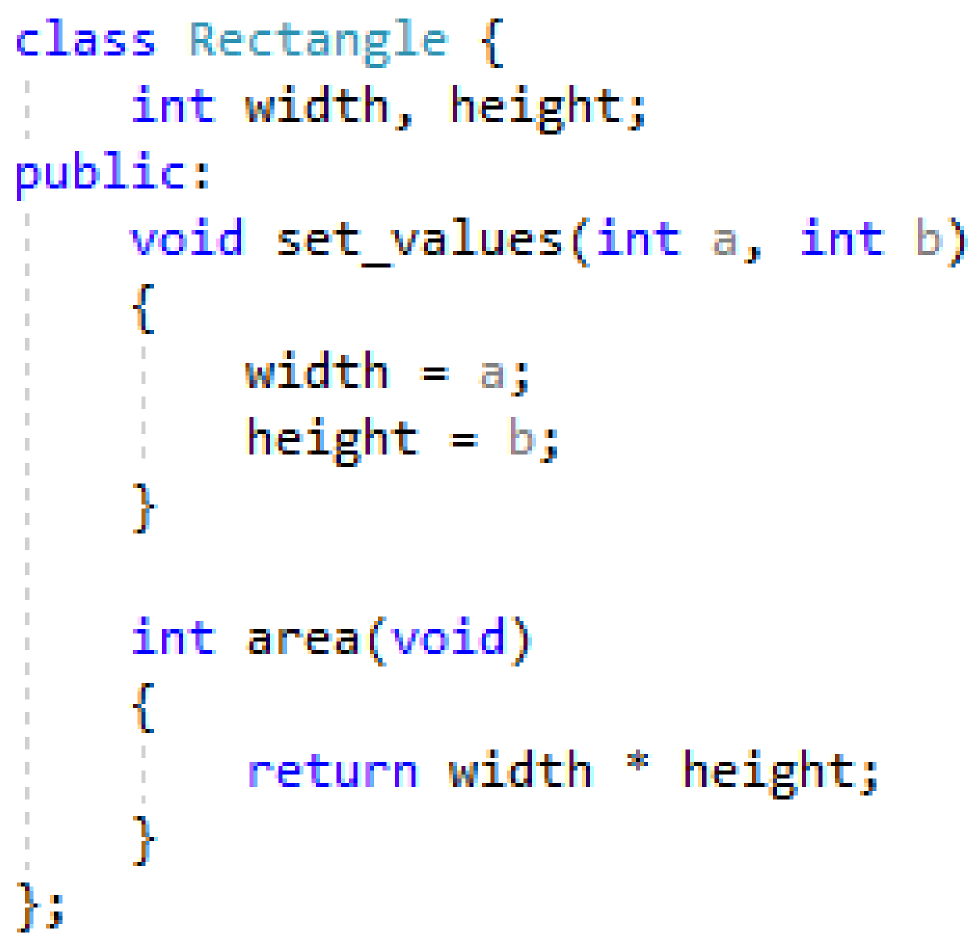
Classes Intro: Access Specifier

- An ***access specifier*** determines what kind of access do you want to give to class members

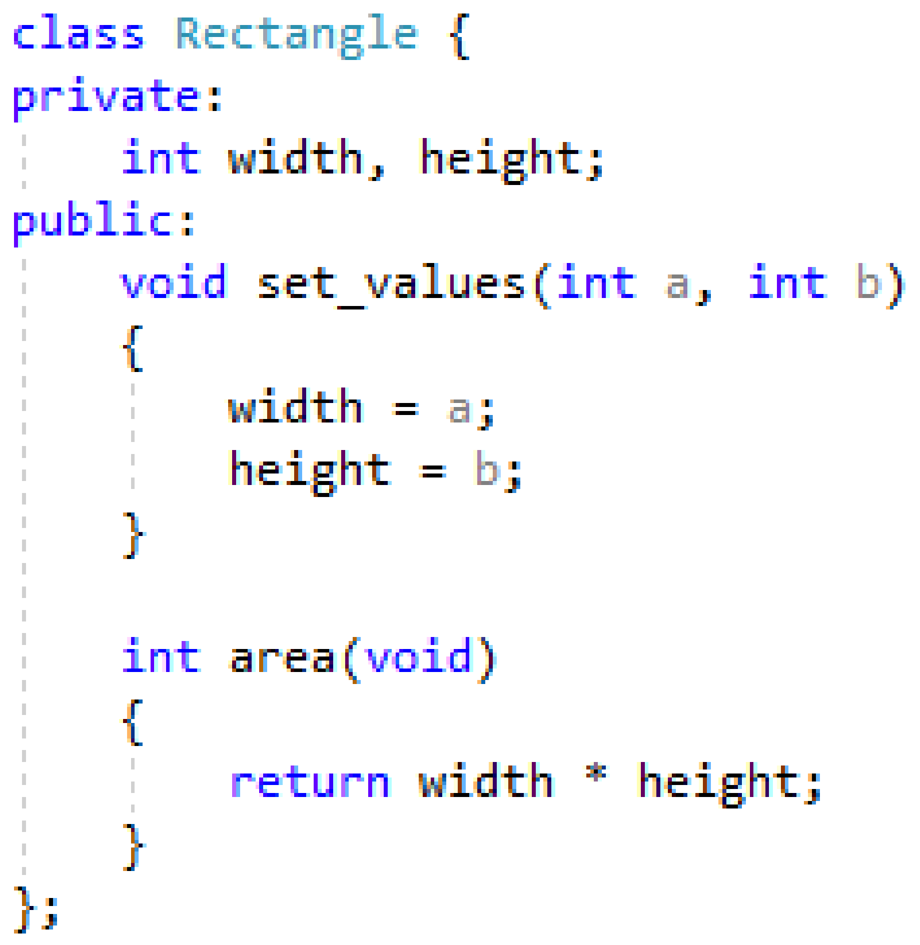
- Access can be of three types:
  - **Private**: members of a class are accessible only from within the same class
  - **Protected**: members of a class are not accessible outside of its members, but is accessible from the members of any class derived from same class
  - **Public:** members are accessible from anywhere where the object is visible

Class definition

- A class definition starts with the keyword ***class*** followed by the class name



```
class Rectangle {
    int width, height;
public:
    void set_values(int a, int b)
    {
        width = a;
        height = b;
    }

    int area(void)
    {
        return width * height;
    }
};
```



```
class Rectangle {
private:
    int width, height;
public:
    void set_values(int a, int b)
    {
        width = a;
        height = b;
    }

    int area(void)
    {
        return width * height;
    }
};
```

Class definition

- Complete example:

```
class Rectangle {
    int width, height;
public:
    void set_values(int a, int b)
    {
        width = a;
        height = b;
    }

    int area(void)
    {
        return width * height;
    }
};
```

Accessor Functions

```
int main()
{
    Rectangle rect;
    rect.set_values(3, 4);
    cout << "area: " << rect.area();
    return 0;
}
```

Member Functions (contd.)

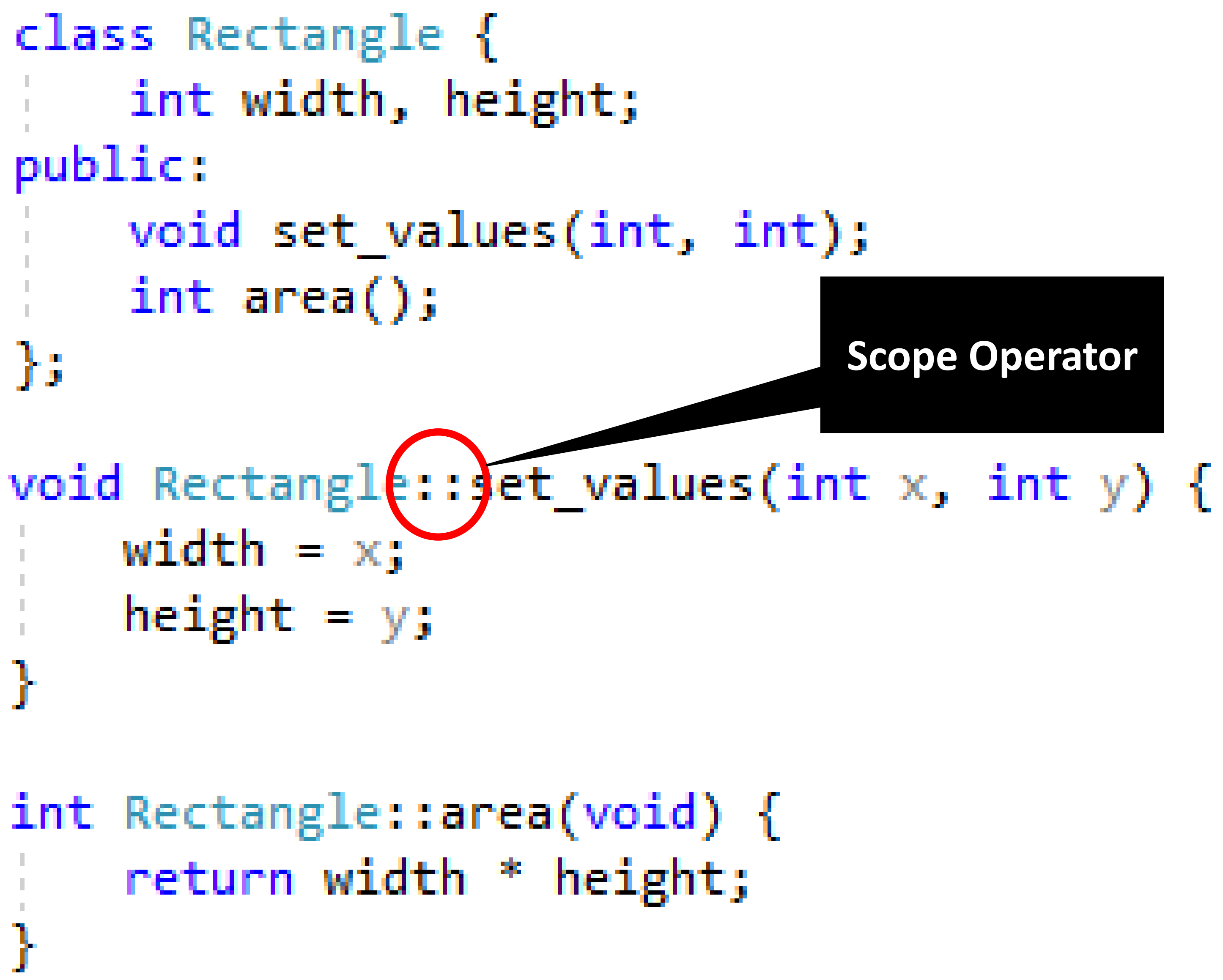
- Define member function inside the class definition

OR

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

Class: Scope Operator

- Outside Class:



```
class Rectangle {
    int width, height;
public:
    void set_values(int, int);
    int area();
};

void Rectangle::set_values(int x, int y) {
    width = x;
    height = y;
}

int Rectangle::area(void) {
    return width * height;
}
```

Class: Scope Operator

- Another Example:

```
#include <iostream>
using namespace std;
class Student
{
    int rollNo;
    public:
    void setRollNo(int aRollNo);
};

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

Scope Operator

Inline Functions

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

Example

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

inline void hello()
{
    cout << "Hello World";
}

int main()
{
    hello();
}
```

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


int main()
{
    cout << "Hello World";
}
```

Inline Functions

- It is a request and not a command. Compiler may not perform inlining in such circumstances like:

  1. If a function contains a **loop**. (for, while, do-while)
  2. If a function contains static variables.
  3. If a function is **recursive**.
  4. If a function contains **switch** or **goto** statement.

Inline Functions – Advantages and Disadvantages

Advantages:

1. Function call overhead doesn’t occur.
2. It also saves the overhead of push/pop variables on the stack when function is called.
3. It also saves overhead of a return call from a function.

Disadvantages:

1. Too many inline functions will increase the size of the binary executable because of the duplication of same code.
2. Inline function may increase compile time overhead. If someone changes the code inside the inline function then all the calling location has to be recompiled.
3. Inline functions may not be useful for many embedded systems. Because in embedded systems code size is more important than speed.

Inline Functions and Classes

- 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

```
#include <iostream>
using namespace std;
class Student
{
   int rollNo;
   public:
   inline void setRollNo(int aRollNo);
};

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

Constructor & Destructor

Back to this example

What would happen if we called member function ***area()*** before having called ***set_values(int, int)***?

```cpp
class Rectangle {
    int width, height;
public:
    void set_values(int a, int b)
    {
        width = a;
        height = b;
    }

    int area(void)
    {
        return width * height;
    }
};
```

```cpp
int main()
{
    Rectangle rect;

    cout << "area: " << rect.area();
    return 0;
}
```

Class: Constructor

- Class can include a special function called its ***constructor***
- Constructor is used to ensure that object is in **well defined state** at the time of creation
- Automatically called when new object is created, allowing class to initialize member variables or (allocate storage). **Cannot be call explicitly**
- Declared just like regular member function, but with a name that **matches the class name** and without any return type; **not even void**

Class: Constructor

- Example:

```cpp
class Rectangle {
    int width, height;
public:
    Rectangle(int, int);
    void set_values(int, int);
    int area();
};

Rectangle::Rectangle(int a, int b) {
    width = a;
    height = b;
}

void Rectangle::set_values(int x, int y) {
    width = x;
    height = y;
}

int Rectangle::area(void) {
    return width * height;
}
```

```cpp
int main() {
    Rectangle rect(3,4);
    cout << "area: " << rect.area();
    return 0;
}
```

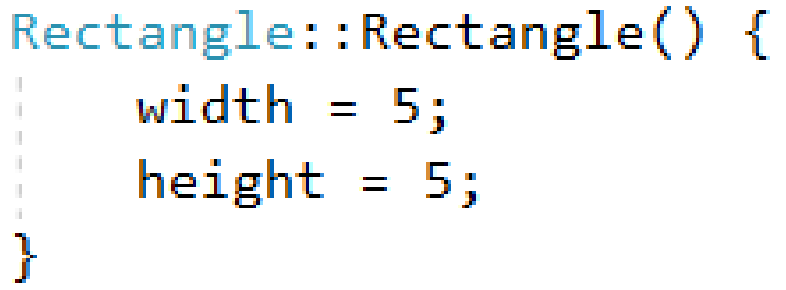
Default Constructor

- Constructor without any argument is called **default constructor**

- If we do not define a default constructor the compiler will generate a default constructor

- Compiler created default constructor has empty body, i.e., **it doesn’t assign default values to data members**

- **Example**



```
Rectangle::Rectangle() {
    width = 5;
    height = 5;
}
```

Class: Constructors Overloading

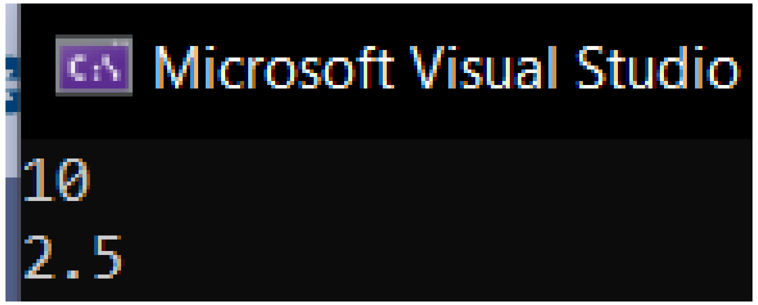
- Constructors Overloading is derived from ***Function Overloading***
- What is ***Function Overloading?***
  - Two functions can have the same name if their parameters are different;
    - ❖ either because they have a **different number of parameters**
    - ❖ or because any of their **parameters are of a different type**

Function Overloading

- Example

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

int operate(int a, int b)
{
    return (a * b);
}

double operate(double a, double b)
{
    return (a / b);
}
```

```cpp
int main()
{
    int x = 5, y = 2;
    double n = 5.0, m = 2.0;
    cout << operate(x, y) << '\n';
    cout << operate(n, m) << '\n';
    return 0;
}
```


Microsoft Visual Studio

```
10
2.5
```

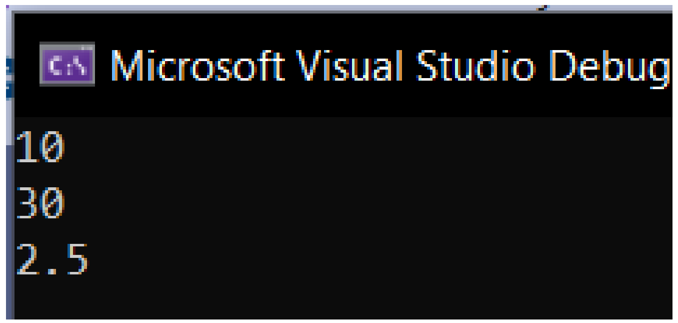
Function Overloading

- Another example

Function cannot be overloaded only by its return type. At least one of its parameters must have a different type.

```cpp
#include <iostream>
using namespace st

int operate(int a, int b)
{
    return (a * b);
}

int operate(int a, int b, int c)
{
    return (a * b * c);
}

double operate(double a, double b)
{
    return (a / b);
}
```

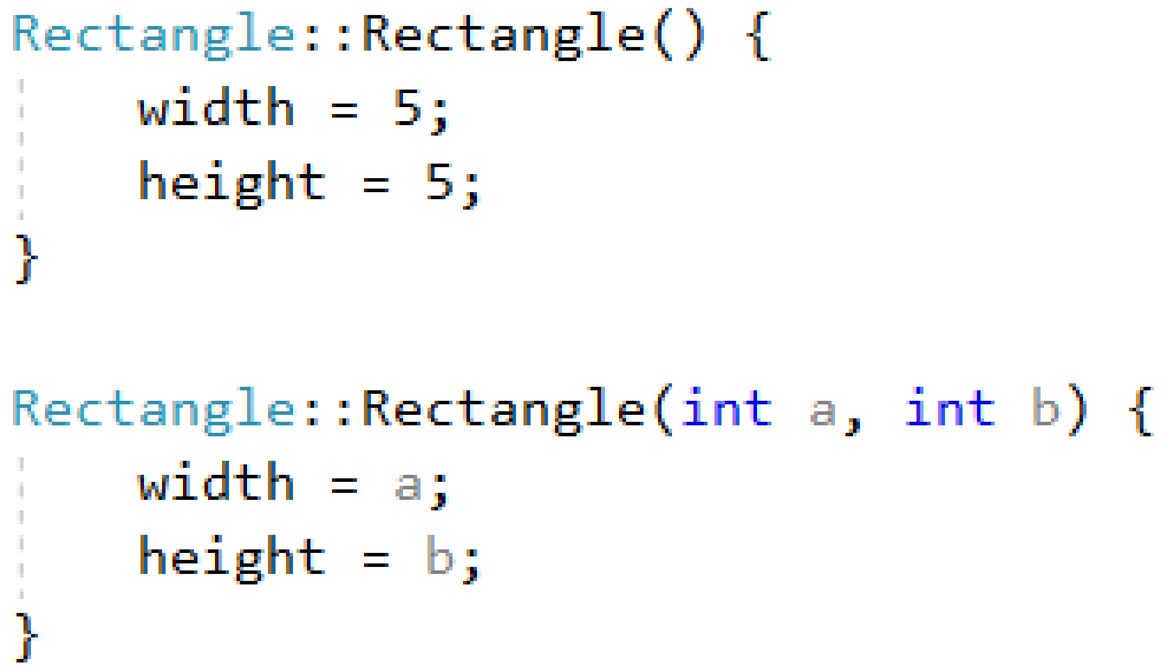
```cpp
int main()
{
    int x = 5, y = 2, z = 3;
    double n = 5.0, m = 2.0;
    cout << operate(x, y) << '\n';
    cout << operate(x, y, z) << '\n';
    cout << operate(n, m) << '\n';
    return 0;
}
```



```
10
30
2.5
```

Class: Constructors Overloading

- Back to Constructor Overloading;
    - Like function, constructor can also be overloaded with different versions taking different parameters



```
Rectangle::Rectangle() {
    width = 5;
    height = 5;
}

Rectangle::Rectangle(int a, int b) {
    width = a;
    height = b;
}
```

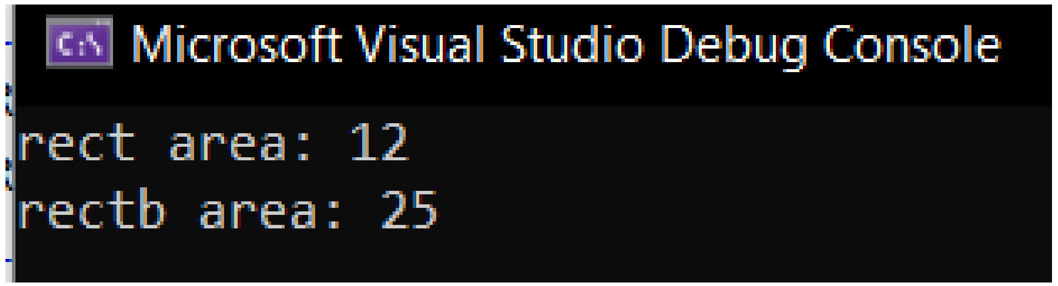
Class: Constructors Overloading

- Complete Example

```
class Rectangle {
    int width, height;
public:
    Rectangle();
    Rectangle(int, int);
    int area();
};

Rectangle::Rectangle() {
    width = 5;
    height = 5;
}

Rectangle::Rectangle(int a, int b) {
    width = a;
    height = b;
}
```

Is called “*default constructor*”.

```
int Rectangle::area() {
    return width * height;
}

int main() {
    Rectangle rect(3, 4);
    Rectangle rectb;
    cout << "rect area: " << rect.area() << endl;
    cout << "rectb area: " << rectb.area() << endl;
    return 0;
}
```


Microsoft Visual Studio Debug Console

```
rect area: 12
rectb area: 25
```

Class: Constructors Overloading

- Use **default parameter value** to reduce the writing effort

```
Rectangle::Rectangle(int a=0, int b=0)
{
   width = a;
   height = b;
}
```

- Is equivalent to

```
Rectangle::Rectangle()
Rectangle::Rectangle(int a)
Rectangle::Rectangle(int a, int b)
```

Class: Destructor

- Automatically called when class object passes **out of scope** or is **explicitly deleted**
- Mainly used to de-allocate the memory that has been allocated for the object by the constructor (or any other member function).
- Syntax is same as constructor except preceded by the tilde sign

```
~class_name() { };    //syntax of destructor
```

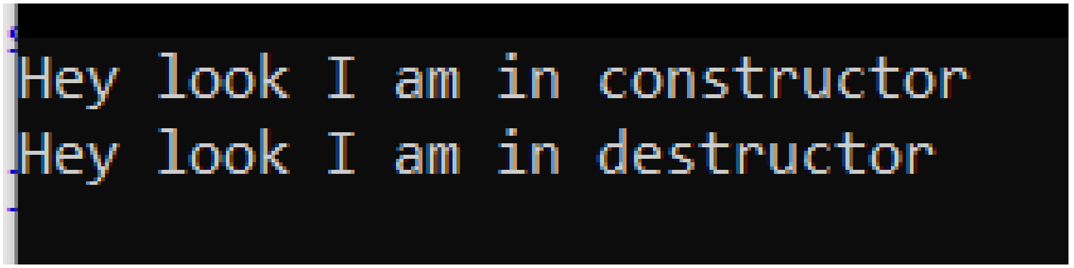
- **Neither takes any arguments nor does it returns value**
- **Can’t be overloaded**

Class: Destructor

- Example (out-of-scope)

```cpp
class Rectangle {
    int width, height;
public:
    Rectangle();
    ~Rectangle();
};

Rectangle::Rectangle() {
    cout << "Hey look I am in constructor" << endl;
}

Rectangle::~Rectangle() {
    cout << "Hey look I am in destructor" << endl;
}
```

```cpp
int main() {
    Rectangle rect;
    return 0;
}
```



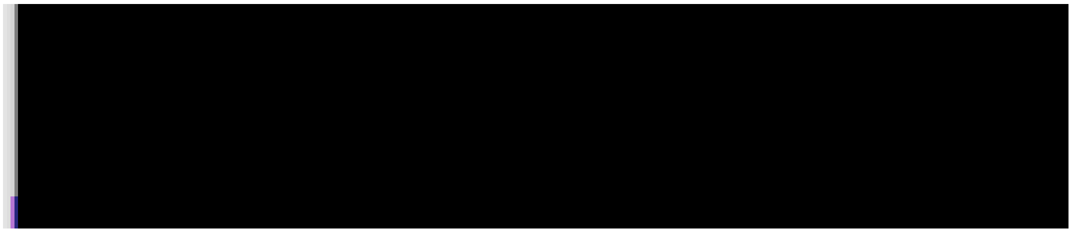
```
Hey look I am in constructor
Hey look I am in destructor
```

Class: Destructor

- Example (out-of-scope)

```cpp
class Rectangle {
    int width, height;
public:
    Rectangle();
    ~Rectangle();
};

Rectangle::Rectangle() {
    cout << "Hey look I am in constructor" << endl;
}

Rectangle::~Rectangle() {
    cout << "Hey look I am in destructor" << endl;
}
```

```cpp
int main() {
    Rectangle *rect;
    return 0;
}
```

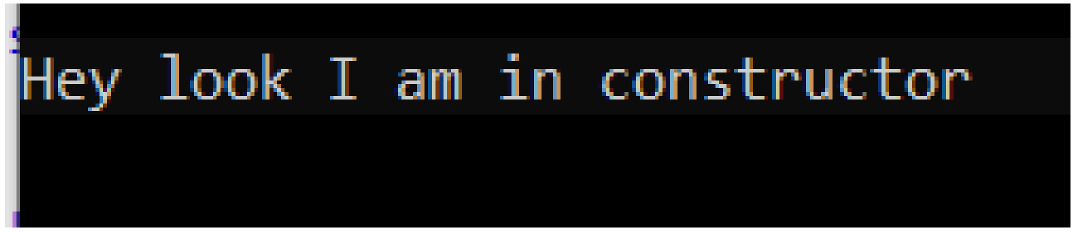


Class: Destructor

- Example (out-of-scope)

```
class Rectangle {
    int width, height;
public:
    Rectangle();
    ~Rectangle();
};

Rectangle::Rectangle() {
    cout << "Hey look I am in constructor" << endl;
}

Rectangle::~Rectangle() {
    cout << "Hey look I am in destructor" << endl;
}
```

```
int main() {
    Rectangle *rect= new Rectangle;
    return 0;
}
```



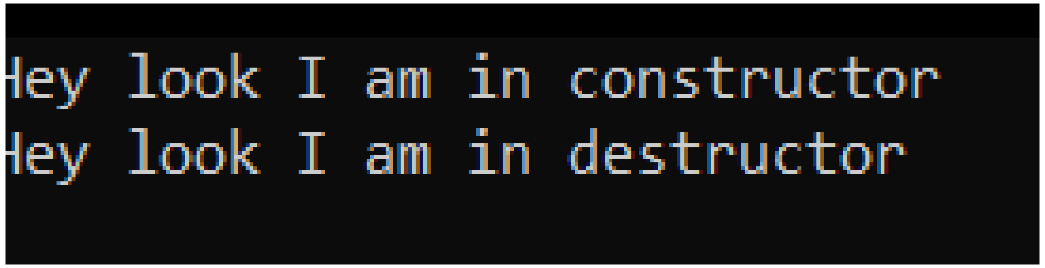
```
Hey look I am in constructor
```

Class: Destructor

- Example (delete)

```cpp
class Rectangle {
    int width, height;
public:
    Rectangle();
    ~Rectangle();
};

Rectangle::Rectangle() {
    cout << "Hey look I am in constructor" << endl;
}

Rectangle::~Rectangle() {
    cout << "Hey look I am in destructor" << endl;
}
```

```cpp
int main() {
  Rectangle *rect= new Rectangle;
  delete rect;
  return 0;
}
```



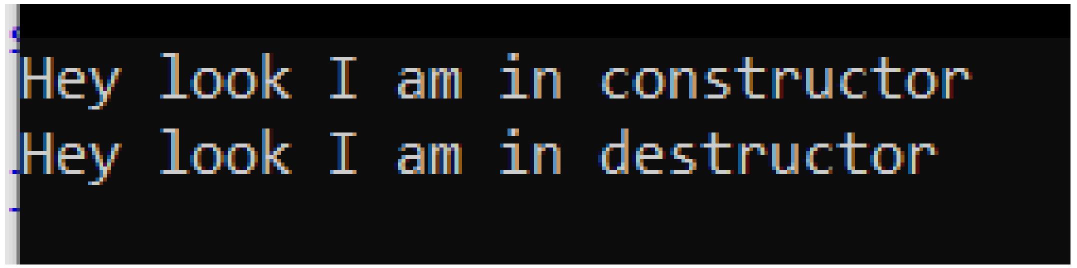
```
Hey look I am in constructor
Hey look I am in destructor
```

Class: Destructor

- Example (when its useful)

```cpp
class Rectangle {
    int *width, *height;
public:
    Rectangle();
    ~Rectangle();
};

Rectangle::Rectangle() {
    cout << "Hey look I am in constructor" << endl;
    width = new int[10];
    height = new int[10];
}

Rectangle::~Rectangle() {
    cout << "Hey look I am in destructor" << endl;
    delete [] width;
    delete [] height;
}
```

```cpp
int main() {
    Rectangle rect;
    return 0;
}
```



```
Hey look I am in constructor
Hey look I am in destructor
```

Sequence of Calls

Constructor & Destructor

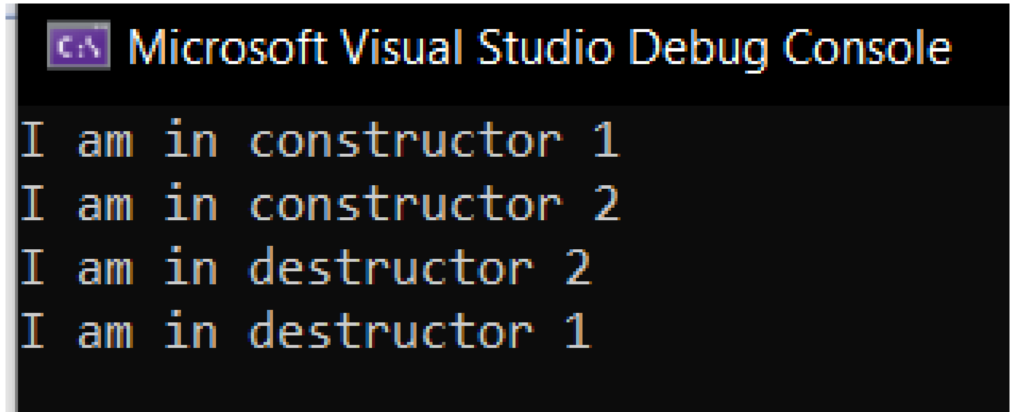
Sequence of Calls

- [*Again Remember*] Constructors and destructors are called automatically
- Constructors are called in the sequence in which object is declared
- Destructors are called in reverse order

Sequence of Calls

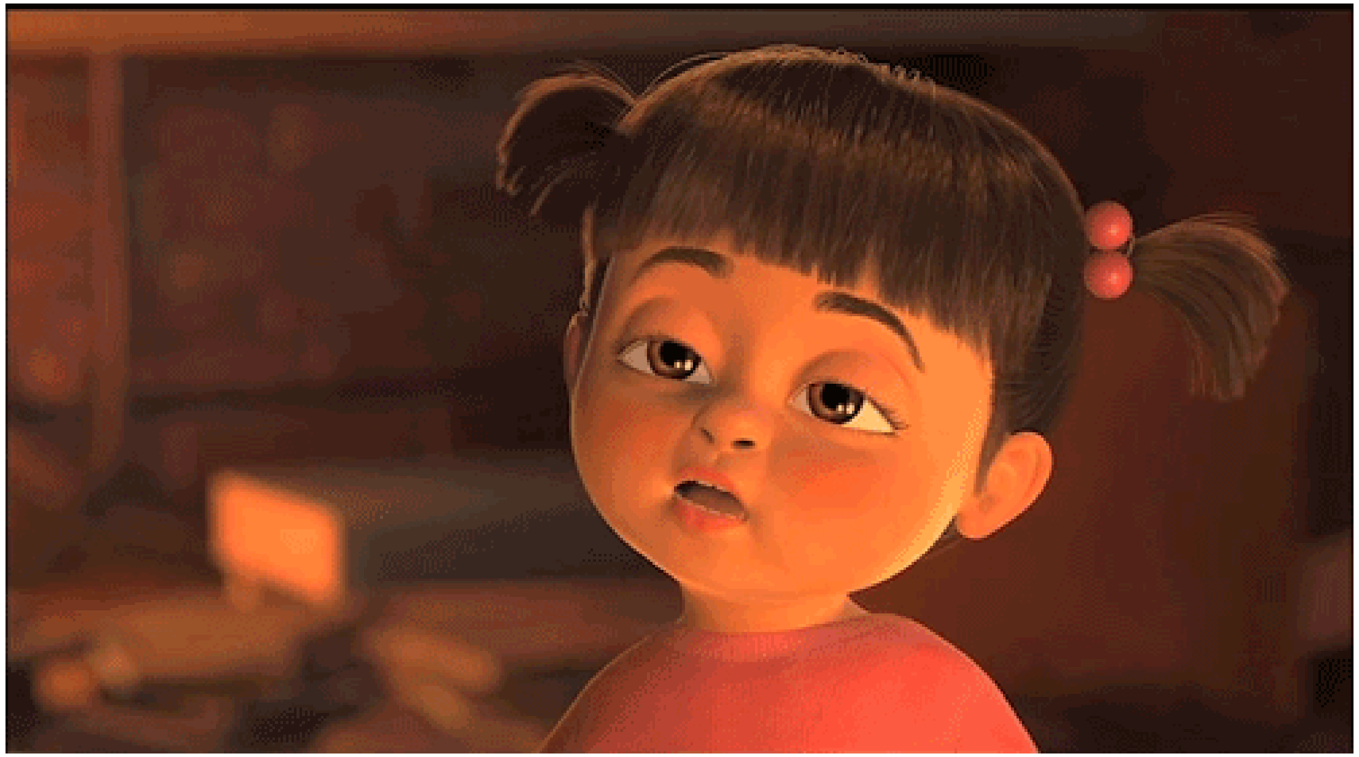
```cpp
#include <iostream>
using namespace std;

class Sequence {
    int check;
public:
    Sequence(int a);
    ~Sequence();
};

Sequence::Sequence(int a)
{
    check = a;
    cout << "I am in constructor " << check << endl;
}

Sequence::~Sequence()
{
    cout << "I am in destructor " << check << endl;
}
```

```cpp
int main()
{
Sequence rect1(1);
Sequence rect2(2);

return 0;
}
```


Microsoft Visual Studio Debug Console

```
I am in constructor 1
I am in constructor 2
I am in destructor 2
I am in destructor 1
```

Thanks a lot



If you are taking a Nap, **wake up**........Lecture Over