Object Oriented Programming

Lecture 4

Dr. Naveed Anwar Bhatti

Webpage: naveedanwarbhatti.github.io

Copy Constructor

Copy Constructor

- Copy constructor can used when:
  - Initializing an object at the time of creation using another object
  - When an **object is passed by value** to a function

“Pass by Value” and “Pass by Reference”

Pass by Value:

- Makes a copy in memory of the actual parameters

- Use pass by value when you are only **using** the parameter for some computation, not changing it

Pass by Reference:

- Forwards the actual parameters

- Use pass by reference when you are **changing** the parameter passed in the program

“Pass by Value”

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

int add(int a)
{
  int b = 0;
  a = a + 1;
  b=a;

  return b;
}


int main() {
  int x = 0;
  int result = add(x);
  cout << result << endl;
  cout << x << endl;
  return 0;
}
```

“Pass by Reference”

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

int add(int* a)
{
  int b = 0;
  *a = *a + 1;
  b=*a;

  return b;
}


int main() {
  int x = 0;
  int result = add(&x);
  cout << result << endl;
  cout << x << endl;
  return 0;
}
```

“Pass by Value”

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

int add(int a)
{
  int b = 0;
  a = a + 1;
  b=a;

  return b;
}


int main() {
  int x = 0;
  int result = add(x);
  cout << result << endl;
  cout << x << endl;
  return 0;
}
```

Function Declaration

“Pass by Reference”

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

int add(int* a)
{
  int b = 0;
  *a = *a + 1;
  b=*a;

  return b;
}


int main() {
  int x = 0;
  int result = add(&x);
  cout << result << endl;
  cout << x << endl;
  return 0;
}
```

Function Declaration

“Pass by Value”

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

int add(int a)
{
  int b = 0;
  a = a + 1;
  b=a;

  return b;
}

int main() {
  int x = 0;
  int result = add(x);
  cout << result << endl;
  cout << x << endl;
return 0;
}
```

Function Definition

“Pass by Reference”

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

int add(int* a)
{
  int b = 0;
  *a = *a + 1;
  b=*a;

  return b;
}

int main() {
  int x = 0;
  int result = add(&x);
  cout << result << endl;
  cout << x << endl;
  return 0;
}
```

Function Definition

“Pass by Value”

Pass by Pointer ~~“Pass by Reference”~~

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

int add(int a)
{
  int b = 0;
  a = a + 1;
  b=a;

  return b;
}


int main() {
  int x = 0;
  int result = add(x);
  cout << result << endl;
  cout << x << endl;
  return 0;
}
```

Function Calling

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

int add(int* a)
{
  int b = 0;
  *a = *a + 1;
  b=*a;

  return b;
}


int main() {
  int x = 0;
  int result = add(&x);
  cout << result << endl;
  cout << x << endl;
  return 0;
}
```

Function Calling

Another way for “Pass by Reference”

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

int add(int &a)
{
  int b = 0;
  a = a + 1;
  b=a;

  return b;
}


int main() {
  int x = 0;
  int result = add(x);
  cout << result << endl;
  cout << x << endl;
  return 0;
}
```

Reference Variable:
Reference variable is an alias for a variable which is assigned to it.

Different from pointer:

- The reference variable can only be initialized at the time of its creation
- The reference variable returns the address of the variable preceded by the reference sign ‘&’
- The reference variable can never be reinitialized again in the program
- The reference variable can never refer to NULL

Back to Copy Constructor

- Copy constructor are used when:
  - Initializing an object at the time of creation using another object
  - When an object is passed by value to a function

- Example:

```
void func1(Rectangle rect)
{
    …
}
void main()
{
    Rectangle a;
    Rectangle b = a;

    func1(a);

}
```

Copy Constructor

- Copy constructor are used when:
  - Initializing an object at the time of creation using another
  - When an object is passed by value to a function

Because if it's not by reference, it's by value. To do that you make a copy, and to do that you call the copy constructor.

You would have infinite recursion because "to make a copy, you need to make a copy".

- Example:

```
void func1(Rectangle rect)
{
    …
}
void main()
{
    Rectangle a;
    Rectangle b = a;

    func1(a);

}
```

- Syntax:

```
Rectangle::Rectangle(Rectangle const &rect)
{
    width = rect.width;
    height = rect.height;
}
```

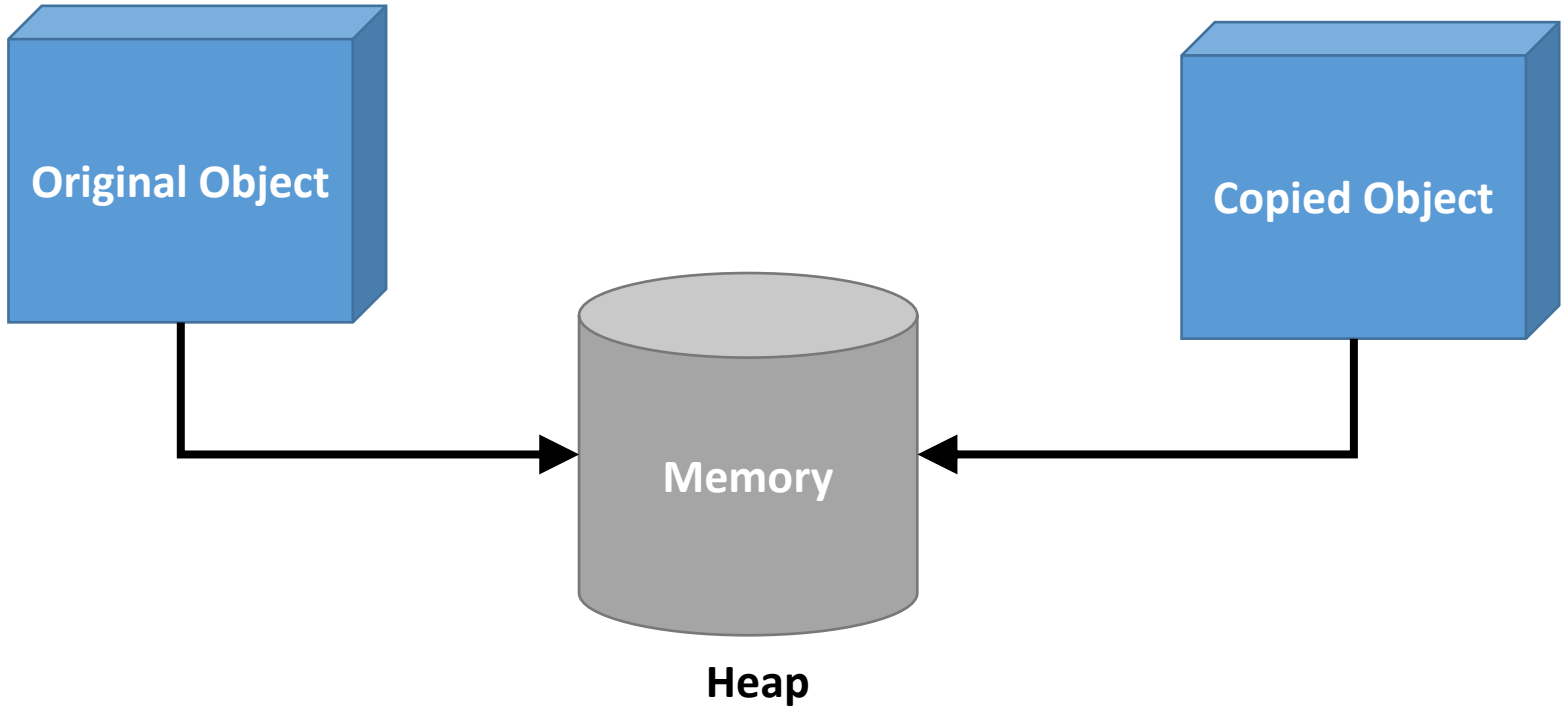
Copy Constructor

Shallow Copy **vs** Deep Copy

Shallow Copy

Shallow copying is creating a new object and then copying the ***data members*** of the **Original Object** to the **Copied Object**.

If the ***data members*** is a ***reference type***, the reference is copied but the referred object is not, therefore the original object and its clone refer to the same object.



Example

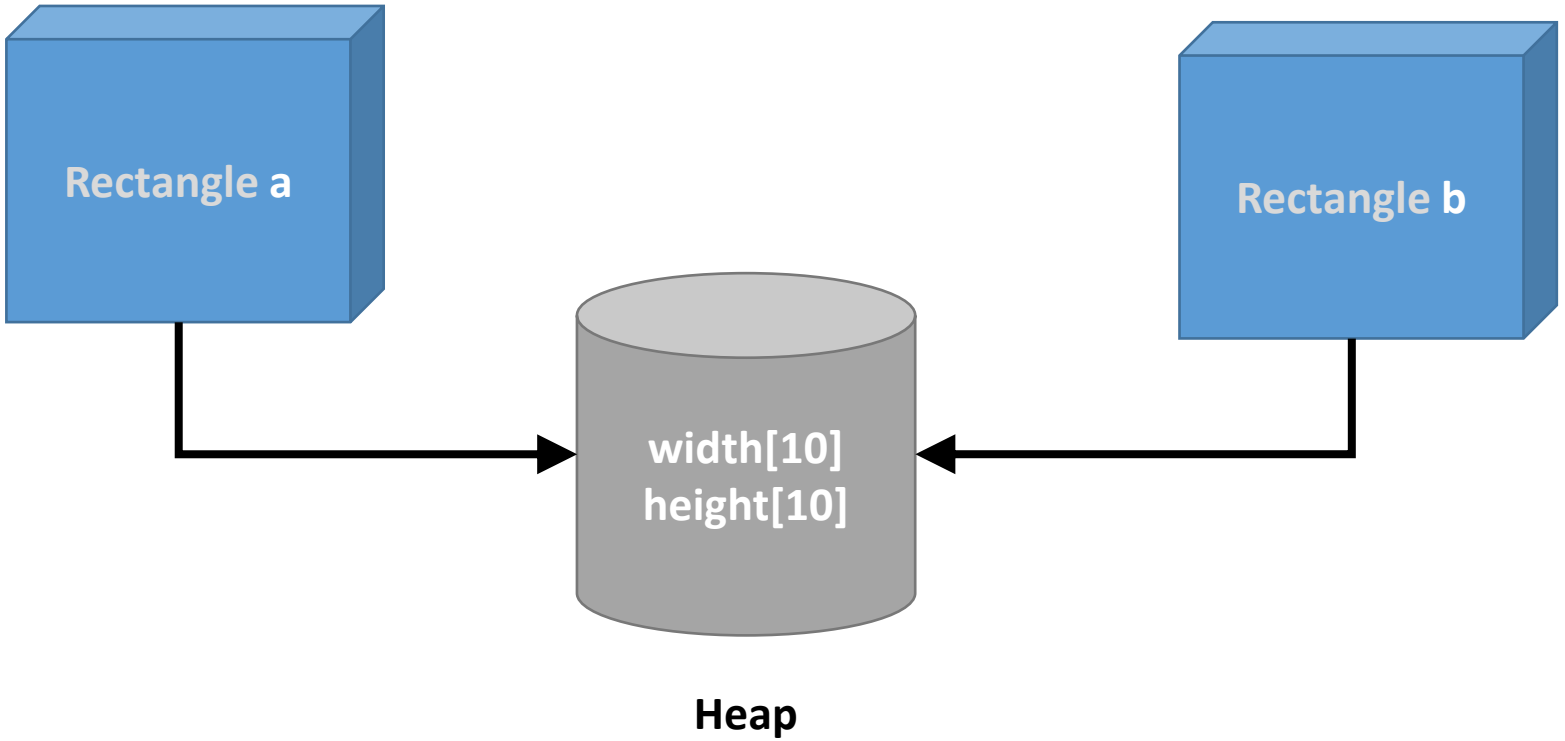
```
class Rectangle{

public:
    int *width, *height;
    Rectangle();
    Rectangle(Rectangle const& rect);

};

Rectangle::Rectangle()
{
    width = new int[10];
    height = new int[10];
}

Rectangle::Rectangle(Rectangle const& rect)
{
    width = rect.width;
    height = rect.height;
}
```

reference type

```
int main()
{
    Rectangle a;
    Rectangle b = a;

    a.width[0] = 10;
    cout << b.width[0];

}
```

```
10
```

Shallow Copy



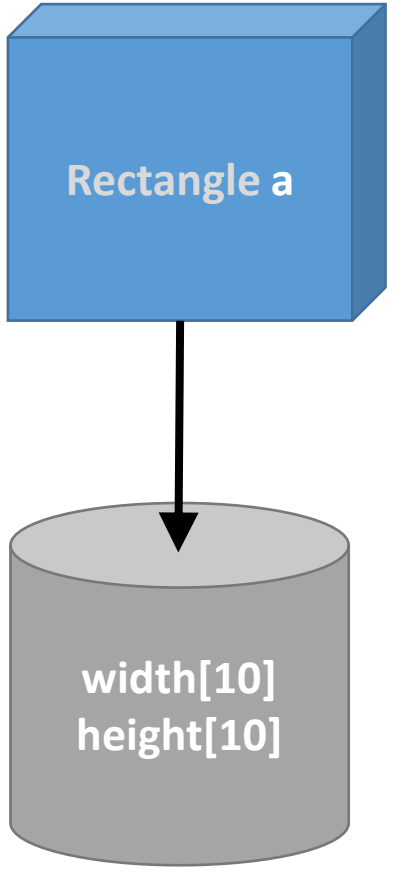
Deep Copy

```
Rectangle::Rectangle(Rectangle const& rect)
{
    width = new int[10];
    height = new int[10];

    for (int i = 0; i < 10; i++)
    {
        width[i] = rect.width[i];
        height[i] = rect.height[i];
    }
}
```

```
int main()
{
    Rectangle a;
    Rectangle b = a;

    a.width[0] = 10;
    b.width[0] = 20;
    cout << a.width[0];

}
```

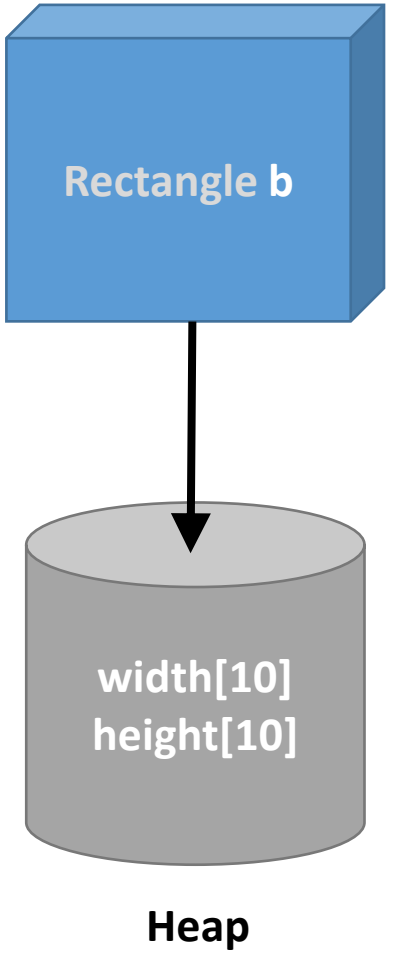
```
10
```

Deep Copy



Deep copying is creating a new object and then copying the ***data members*** of the **Original Object** to the **Copied Object**.

If the ***data members*** is a ***reference type***, a **new copy** of the referred object is performed. A deep copy of an object is a new object with entirely new instance variables, it does not share objects with the old.



Copy Constructor (contd.)

- **Copy constructor** is normally used to perform **deep copy**

- If we do not make a **copy constructor** then the compiler performs **shallow copy**

Copy Constructor (contd.)

- **Question**

```
MyClass t1, t2;
MyClass t3 = t1;  // ----> (1)
t2 = t1;          // -----> (2)
```

Which of the following two statements call copy constructor and which one calls assignment operator?

this Pointer

this Pointer

```
class Rectangle
{
    int width, height;
public:
    void set_width(int a);
    void set_height(int b);
    int area();
};
```

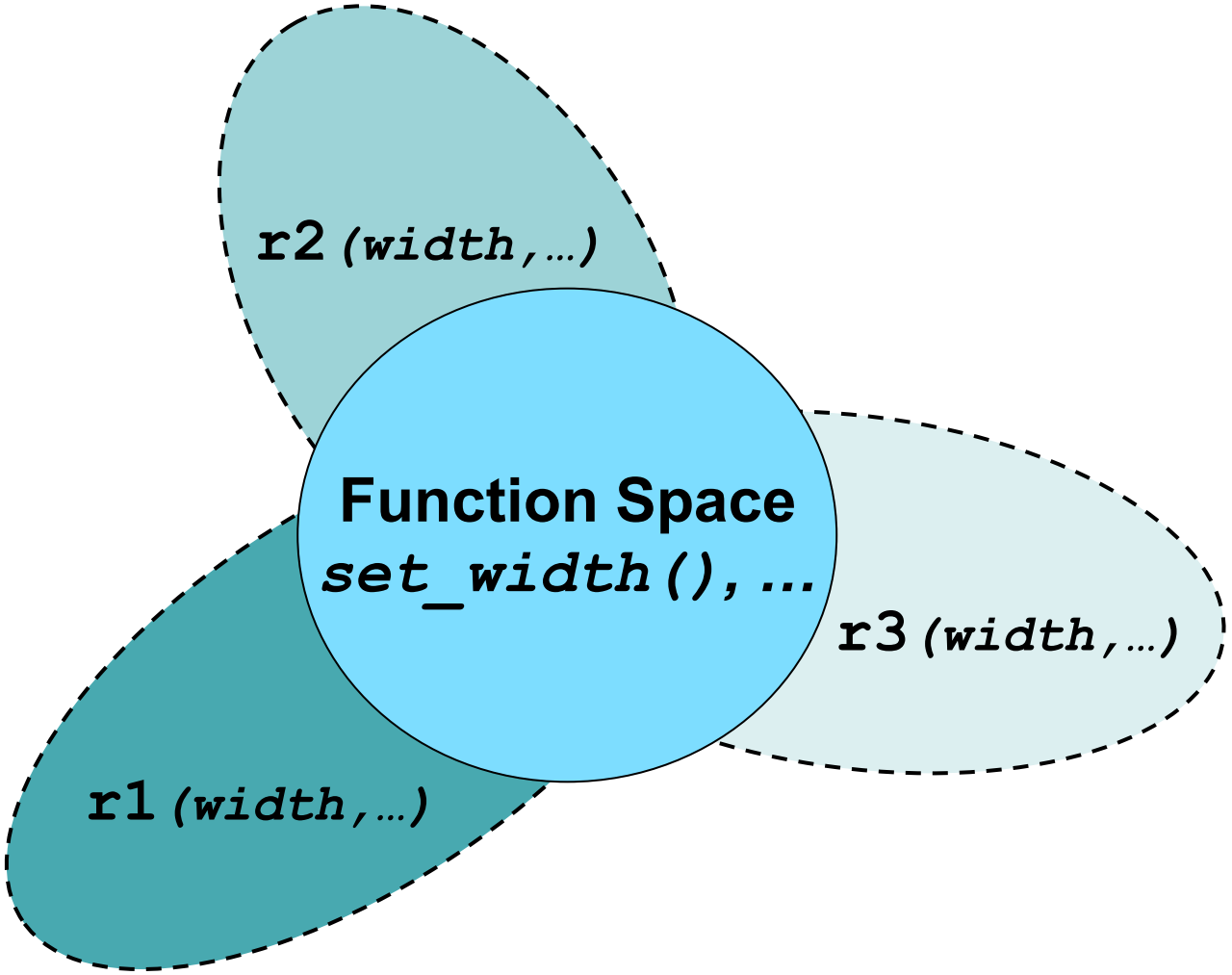
```
void Rectangle::set_width(int a)
{
    width = a;

}

void Rectangle::set_height(int b)
{
    height = b;

}

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

`this` Pointer

- The compiler reserves space for the functions defined in the class
- Space for data is not allocated (*since no object is yet created*)

this Pointer

- Rectangle **r1, r2, r3;**

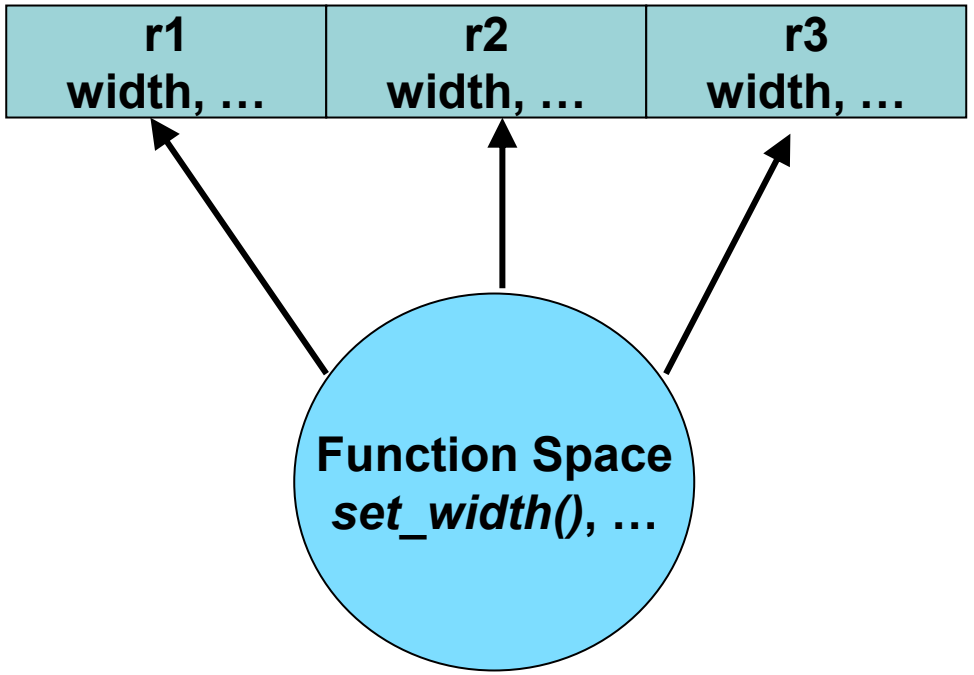


this Pointer

- Function space is **common** for every object
- Whenever a new object is created:
  - Memory is reserved for variables only
  - Previously defined functions are used over and over again

this Pointer

- **Memory layout for objects created:**



How does the functions know on which object to act?

this Pointer

- Address of each object is passed to the calling function
- This address is deferenced by the functions and hence they act on correct objects

r1 width, …	r2 width, …	r3 width, …	r4 width, …
address	*address*	*address*	*address*

The variable containing the “self-address” is called ***this*** pointer

Passing *this* Pointer

- Whenever a function is called the ***this*** pointer is passed as a parameter to that function
- Function with *n* parameters is actually called with *n+1* parameters

Example

```
void Rectangle::set_width(int a)
```

is internally represented as

```
void Rectangle::set_width(int a, Rectangle* const this)
```

Compiler Generated Code

```
Rectangle::set_width(int a)
{
 width = a;
}
```

is internally represented as

```
Rectangle::set_width(int a, Rectangle* const this)
{
 this->width = a;
}
```

this Pointer

There are situations where designer wants to use ***this*** pointer **explicitly**

Case 1: When local variable's name is same as member's name

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

```
void Rectangle::set_width(int width)
{
    width = width;

}

void Rectangle::set_height(int height)
{
    height = height;

}

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

Case 1: When local variable's name is same as member's name

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

```
void Rectangle::set_width(int width)
{
    this->width = width;

}

void Rectangle::set_height(int height)
{
    this->height = height;

}

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

Case 2: To return reference to the calling object

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

```
Rectangle& Rectangle::set_width(int width)
{
    this->width = width;
    return *this;

}

Rectangle& Rectangle::set_height(int height)
{
    this->height = height;
    return *this

}

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

Case 2: To return reference to the calling object

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

```
int main()
{

Rectangle r1;
r1.set_width(10).set_height(10);
cout << r1.area();
return 0;
}
```

```
Rectangle& Rectangle::set_width(int width)
{
    this->width = width;
    return *this;

}

Rectangle& Rectangle::set_height(int height)
{
    this->height = height;
    return *this

}

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

Case 2: To return reference to the calling object

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

```
Rectangle& Rectangle::set_width(int width)
{
    this->width = width;
    return *this;
```

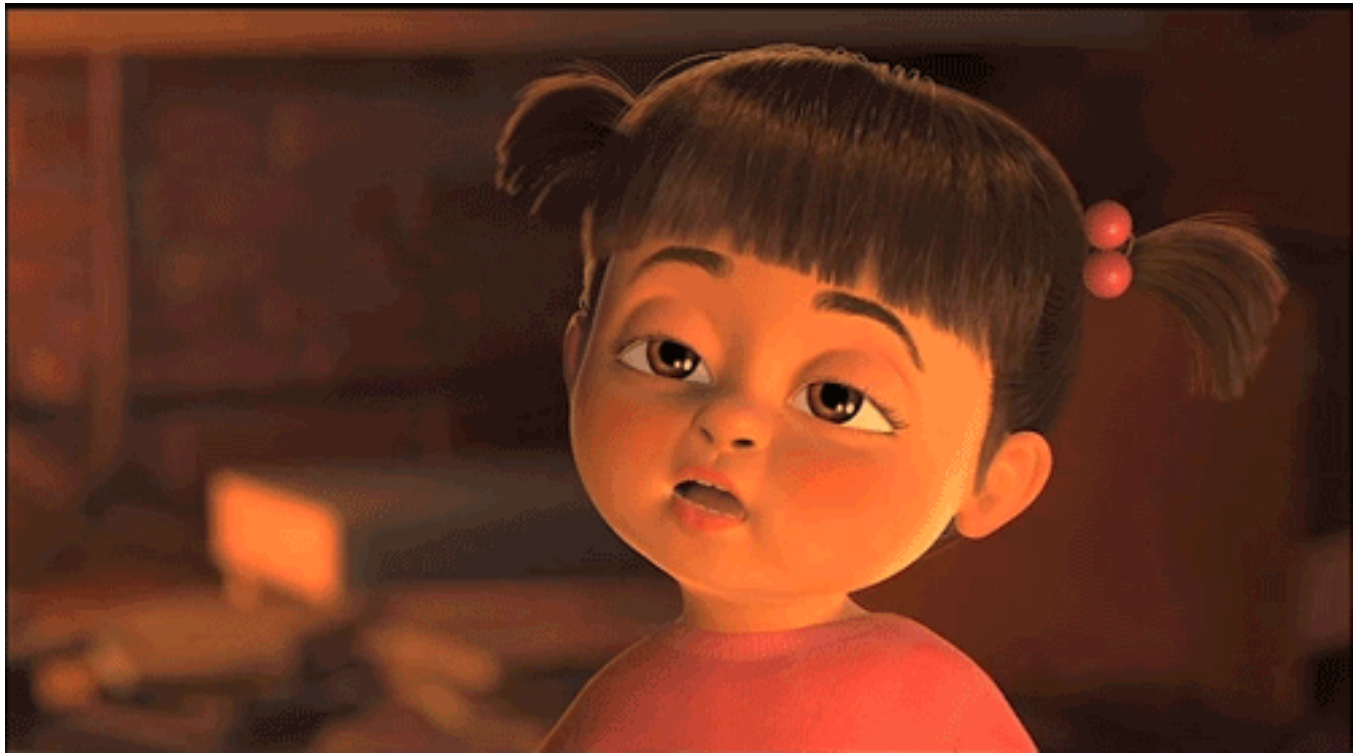
```
int height)
```

When a reference to a local object is returned, the returned reference can be used to ***chain function*** **calls** on a single object.

```
int main()
{

Rectangle r1;
r1.set_width(10).set_height(10);
cout << r1.area();
return 0;
}
```

```
}

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

Thanks a lot



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