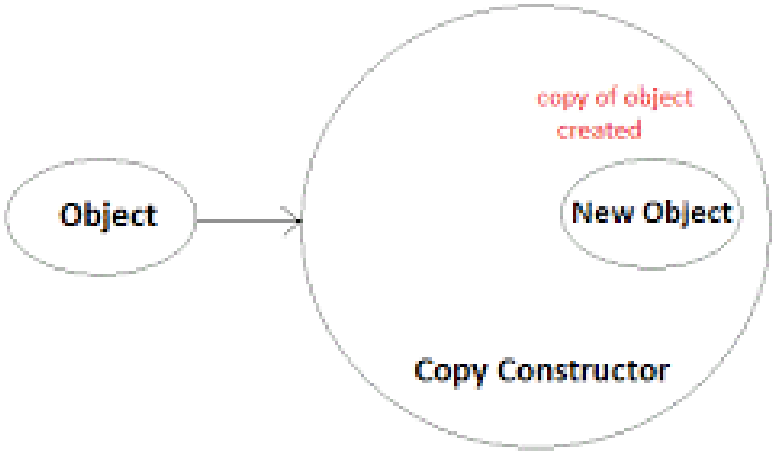
C/C++ Programming: C++ interm. (2/3)



Course **EE3093** – Lecturer: Dr F. Verdicchio

Any question?

Relevant topics to ask questions:

- C++ Objects:
  - Constructors; destructor; static members.
  - Any nagging doubt.

Operators applied to objects

For “simple” variables the behaviour of basic operators, such as “ = ” , “ + ” is straightforward (mostly – remember pointers?!) :

```
int x, y = 5;
x = y +2;
```

But what happens when those are applied to objects?
Let’s start with “ = ”

Operators applied to objects: example

```
void test_default_operators()
{
    rectangle testobj0;
    testobj0.inputSides(6,6);
    cout << " testobj0:"<< endl;
    testobj0.printRectangleInfo();
    rectangle::printRectangleCount();
    cout<<endl;
    {
        cout << "Entering local scope;"<< endl << endl;
        rectangle testobj1;
        testobj1=testobj0;
        cout << " testobj1:"<< endl;
        testobj1.printRectangleInfo();
        rectangle::printRectangleCount();
        cout<<endl;

        rectangle testobj2=testobj0;
        cout << " testobj2:"<< endl;
        testobj2.printRectangleInfo();
        rectangle::printRectangleCount();
        cout << "Exiting local scope;"<< endl << endl;
    }
    rectangle::printRectangleCount();

    char final[10];
    cout << " Press any key then Enter to finish ";
    cin >> final;
}
```

One Instance of class Rectangle instantiated (created) & initialized

Operators applied to objects: example

```
void test_default_operators()
{
    rectangle testobj0;
    testobj0.inputSides(6,6);
    cout << " testobj0:"<< endl;
    testobj0.printRectangleInfo();
    rectangle::printRectangleCount();
    cout<<endl;
    {
        cout << "Entering local scope;"<< endl << endl;
```

One Instance of class Rectangle instantiated (created) & initialized

```
H:\fverdiccABDN\UniABDN\MyCourses\EE3093\LectureSlidesRepository\Code\EE309
 testobj0:
Rectangle ID is: 0
Rectangle side A is: 6
Rectangle side B is: 6
Rectangle area is: 36
Rectangle perimeter is: 24
Total numbers for Rectangle instantiations:
 TOT instatntiated Rectangles (currently active or not): 1
 TOT currently Active Rectangles: 1
 TOT currently Initialized Rectangles: 1
```

```
    char final[10];
    cout << " Press any key then Enter to finish ";
    cin >> final;
}
```

Operators applied to objects: example

```
void test_default_operators()
{
    rectangle testobj0;
    testobj0.inputSides(6,6);
    cout << " testobj0:"<< endl;
    testobj0.printRectangleInfo();
    rectangle::printRectangleCount();
    cout<<endl;
    {
        cout << "Entering local scope;"<< endl << endl;
        rectangle testobj1;
        testobj1=testobj0;
        cout << " testobj1:"<< endl;
        testobj1.printRectangleInfo();
        rectangle::printRectangleCount();
        cout<<endl;

        rectangle testobj2=testobj0;
        cout << " testobj2:"<< endl;
        testobj2.printRectangleInfo();
        rectangle::printRectangleCount();
        cout << "Exiting local scope;"<< endl << endl;
    }
    rectangle::printRectangleCount();

    char final[10];
    cout << " Press any key then Enter to finish ";
    cin >> final;
}
```

A second Instance of class Rectangle instantiated (created) in this local scope;

Operators applied to objects: example

```
void test_default_operators()
{
    rectangle testobj0;
    testobj0.inputSides(6,6);
    cout << " testobj0:"<< endl;
    testobj0.printRectangleInfo();
    rectangle::printRectangleCount();
    cout<<endl;
    {
        cout << "Entering local scope;"<< endl << endl;
        rectangle testobj1;
        testobj1=testobj0;
        cout << " testobj1:"<< endl;
        testobj1.printRectangleInfo();
        rectangle::printRectangleCount();
        cout<<endl;

        rectangle testobj2=testobj0;
        cout << " testobj2:"<< endl;
        testobj2.printRectangleInfo();
        rectangle::printRectangleCount();
        cout << "Exiting local scope;"<< endl << endl;
    }
    rectangle::printRectangleCount();

    char final[10];
    cout << " Press any key then Enter to finish ";
    cin >> final;
}
```

A second Instance of class Rectangle instantiated (created) in this local scope;
and set “to be equal to” another object.

Operators applied to objects: example

```
void test_default_operators()
{
    rectangle testobj0;
    testobj0.inputSides(6,6);
    cout << " testobj0:"<< endl;
    testobj0.printRectangleInfo();
    rectangle::printRectangleCount();
    cout<<endl;
    {
        cout << "Entering local scope;"<< endl << endl;
        rectangle testobj1;
        testobj1=testobj0;
        cout << " testobj1:"<< endl;
        testobj1.printRectangleInfo();
        rectangle::printRectangleCount();
        cout<<endl;

        rectangle testobj2=testobj0;
        cout << " testobj2:"<< endl;
        testobj2.printRectangleInfo();
        rectangle::printRectangleCount();
        cout << "Exiting local scope;"<< endl << endl;
    }
    rectangle::printRectangleCount();

    char final[10];
    cout << " Press any key then Enter to finish ";
    cin >> final;
}
```

A second Instance of class Rectangle instantiated (created) in this local scope;
and set “to be equal to” another object. What happens?

Operators applied to objects: example

```
Entering local scope;

 testobj1:
Rectangle ID is: 0
Rectangle side A is: 6
Rectangle side B is: 6
Rectangle area is: 36
Rectangle perimeter is: 24
Total numbers for Rectangle instantiations:
 TOT instatntiated Rectangles (currently active or not): 2
 TOT currently Active Rectangles: 2
 TOT currently Initialized Rectangles: 1
```

```
    rectangle testobj1;
    testobj1=testobj0;
    cout << " testobj1:"<< endl;
    testobj1.printRectangleInfo();
    rectangle::printRectangleCount();
    cout<<endl;

    rectangle testobj2=testobj0;
    cout << " testobj2:"<< endl;
    testobj2.printRectangleInfo();
    rectangle::printRectangleCount();
    cout << "Exiting local scope;"<< endl << endl;
}
rectangle::printRectangleCount();

char final[10];
cout << " Press any key then Enter to finish ";
cin >> final;
}
```

A second Instance of class Rectangle instantiated (created) in this local scope;
and set "to be equal to" another object. What happens?

The second rectangle is a *field-by-field copy* of the first (same ID); its instantiation is accounted for; initialization is not.

Operators applied to objects: example

```
void test_default_operators()
{
    rectangle testobj0;
    testobj0.inputSides(6,6);
    cout << " testobj0:"<< endl;
    testobj0.printRectangleInfo();
    rectangle::printRectangleCount();
    cout<<endl;
    {
        cout << "Entering local scope;"<< endl << endl;
        rectangle testobj1;
        testobj1=testobj0;
        cout << " testobj1:"<< endl;
        testobj1.printRectangleInfo();
        rectangle::printRectangleCount();
        cout<<endl;

        rectangle testobj2=testobj0;
        cout << " testobj2:"<< endl;
        testobj2.printRectangleInfo();
        rectangle::printRectangleCount();
        cout << "Exiting local scope;"<< endl << endl;
    }
    rectangle::printRectangleCount();

    char final[10];
    cout << " Press any key then Enter to finish ";
    cin >> final;
}
```

A third instance of class Rectangle instantiated (created) and immediately set “to be equal to” another object. What happens?

Operators applied to objects: example

```
 testobj2:
Rectangle ID is: 0
Rectangle side A is: 6
Rectangle side B is: 6
Rectangle area is: 36
Rectangle perimeter is: 24
Total numbers for Rectangle instantiations:
 TOT instatntiated Rectangles (currently active or not): 2
 TOT currently Active Rectangles: 2
 TOT currently Initialized Rectangles: 1
Exiting local scope;
```

```
void test_default_operators()
{
        rectangle testobj1;
        testobj1=testobj0;
        cout << " testobj1:"<< endl;
        testobj1.printRectangleInfo();
        rectangle::printRectangleCount();
        cout<<endl;

        rectangle testobj2=testobj0;
        cout << " testobj2:"<< endl;
        testobj2.printRectangleInfo();
        rectangle::printRectangleCount();
        cout << "Exiting local scope;"<< endl << endl;
    }
    rectangle::printRectangleCount();

    char final[10];
    cout << " Press any key then Enter to finish ";
    cin >> final;
}
```

A third instance of class Rectangle instantiated (created) and immediately set "to be equal to" another object. What happens?

The third rectangle is directly obtained as a *field-by-field copy* of the object used to instantiate this copy; the constructor was not called (instantiation should be done at the same time as the copy).

Operators applied to objects: example

```
void test_default_operators()
{
    rectangle testobj0;
```

```
Total numbers for Rectangle instantiations:
 TOT instatntiated Rectangles (currently active or not): 2
 TOT currently Active Rectangles: 0
 TOT currently Initialized Rectangles: -1
 Press any key then Enter to finish
```

```
    {
        cout << "Entering local scope;"<< endl << endl;
        rectangle testobj1;
        testobj1=testobj0;
        cout << " testobj1:"<< endl;
        testobj1.printRectangleInfo();
        rectangle::printRectangleCount();
        cout<<endl;

        rectangle testobj2=testobj0;
        cout << " testobj2:"<< endl;
        testobj2.printRectangleInfo();
        rectangle::printRectangleCount();
        cout << "Exiting local scope;"<< endl << endl;
    }
    rectangle::printRectangleCount();

    char final[10];
    cout << " Press any key then Enter to finish ";
    cin >> final;
}
```

Exiting the local scope the destructor is called for the **two** (local) objects: the *counter of active* and the *counter of initialized rectangles* are decremented….
… but were not incremented in the first place, so the total is now wrong!

Operators applied to objects

For “simple” variables the behaviour of basic operators, such as “ = ” , “ + ” is straightforward (mostly – remember pointers?!) :

```
int x, y = 5;
x = y +2;
```

But what happens when those are applied to objects?

Let’s start with “ = ”

(1) Initialize one (previously instantiated) object with content of another

```
testobj1=testobj0;
```

(2) Instantiate and initialize one object with content of another

```
rectangle testobj1=testobj0;
```

Operators applied to objects

For “simple” variables the behaviour of basic operators, such as “ = ” , “ + ” is straightforward (mostly – remember pointers?!) :

```
int x, y = 5;
x = y +2;
```

But what happens when those are applied to objects?

Let’s start with “ = ”

(1) Initialize one (previously instantiated) object with content of another

```
testobj1=testobj0;
```

start with this: write a member function that initializes testobj1 with the values of the sides of testobj0

Operators applied to objects

Public member function of class Rectangle:

```
void copyFrom(      rectangle& other)
{
    // if the input rectangle is initialized, copy its sides
    if(other.isInitialized())
    {
        resetRectangle();
        inputSides(other.getSide(1), other.getSide(2));
    }
}
```

Operators applied to objects

```
void test_default_operators()
{
    rectangle testobj0;
    testobj0.inputSides(6,6);
    cout << " testobj0:"<< endl;
    testobj0.printRectangleInfo();
    rectangle::printRectangleCount();
    cout<<endl;
    {
        cout << "Entering local scope;"<< endl << endl;
        rectangle testobj1;
        //testobj1=testobj0;
        testobj1.copyFrom(testobj0);
        cout << " testobj1:"<< endl;
        testobj1.printRectangleInfo();
        rectangle::printRectangleCount();
        cout<<endl;

        rectangle testobj2=testobj0;
        cout << " testobj2:"<< endl;
        testobj2.printRectangleInfo();
        rectangle::printRectangleCount();
        cout << "Exiting local scope;"<< endl << endl;
    }
    rectangle::printRectangleCount();

    char final[10];
    cout << " Press any key then Enter to finish ";
    cin >> final;
}
```

Operators applied to objects

```
void test_default_operators()
{
    rectangle testobj0;
    testobj0.inputSides(6,6);
    cout << " testobj0:"<< endl;
    testobj0.printRectangleInfo();
    rectangle::printRectangleCount();
    cout<<endl;
    {
        cout << "Entering local scope;"<< endl << endl;
        rectangle testobj1;
        //testobj1=testobj0;
        testobj1.copyFrom(testobj0);
        cout << " testobj1:"<< endl;
        testobj1.printRectangleInfo();
        rectangle::printRectangleCount();
        cout<<endl;
```

```
Entering local scope;

 testobj1:
Rectangle ID is: 1
Rectangle side A is: 6
Rectangle side B is: 6
Rectangle area is: 36
Rectangle perimeter is: 24
Total numbers for Rectangle instantiations:
 TOT instatntiated Rectangles (currently active or not): 2
 TOT currently Active Rectangles: 2
 TOT currently Initialized Rectangles: 2
```

```
    cout << " Press any key then Enter to finish ";
    cin >> final;
}
```

Operators applied to objects

Public member function of class Rectangle:

```
void copyFrom(       rectangle& other)
{
    // if the input rectangle is initialized, copy its sides
    if(other.isInitialized())
    {
        resetRectangle();
        inputSides(other.getSide(1), other.getSide(2));
    }
}
```

Interlude: the *other* object is passed by reference: this is more efficient than passing an object by value (which needs to create a copy – more on this later), but, potentially, could allow *copyFrom* to **modify its input** (*other* object).

Operators applied to objects

Public member function of class Rectangle:

```
void copyFrom(const rectangle& other)
{
    // if the input rectangle is initialized, copy its sides
    if(other.isInitialized())
    {
        resetRectangle();
        inputSides(other.getSide(1), other.getSide(2));
    }
}
```

Interlude: the *other* object is passed by reference: this is more efficient than passing an object by value (which needs to create a copy – more on this later), but, potentially, could allow *copyFrom* to modify its input (*other* object).
If this is not the intended behaviour for the function (as in the case of *copyfrom*), then the **input reference** can be declared as **const**.

Operators applied to objects

Public member function of class Rectangle:

```
void copyFrom(const rectangle& other)
{
    // if the input rectangle is initialized, copy its sides
    if(other.isInitialized())
    {
        resetRectangle();
        inputSides(other.getSide(1), other.getSide(2));
    }
}
```

A member function can have a **const input reference**, in this example
copyFrom(const rectangle& other);
This imposes that the function *copyFrom* does not modify *other* in any way;

Operators applied to objects

Public member function of class Rectangle:

```
void copyFrom(const rectangle& other)
{
    // if the input rectangle is initialized, copy its sides
    if(other.isInitialized())
    {
        resetRectangle();
        inputSides(other.getSide(1), other.getSide(2));
    }
}
```

A member function can have a **const input reference**, in this example copyFrom(const rectangle& other);
This imposes that the function *copyFrom* does not modify *other* in any way; in turn this means that *copyFrom* can call one of *other's* member functions only if that **function** is **defined** as **const**, i.e. a function that does not modify the object.

Operators applied to objects

Public member function of class Rectangle:

```
// gets
double getArea() const { ... }
double getPerimeter() const { ... }
bool isInitialized() const {return init_flag;}
double getSide(int sidenum) const { ... }
void inputSides(double in_sideA, double in_sideB) { ... }
void inputSidesFromKeyboard() { ... }
void printRectangleInfo() const { ... }
void inputRandomSides(double max_val=100) { ... }
void resetRectangle(){set_init_flag(false);}
int getRectangleID() const {return rectangle_ID;}
int getAciveRectanglesCount() const {return rectangle_instances_alive_count;}
int getInitializedRectanglesCount() const {return initialized_rectangle_instances_count;}
// static function to get the
static void printRectangleCount() { ... }
```

function is **defined** as **const**, i.e. a function that does not modify the object (the compiler issues an error if the member function modifies the object).

Operators applied to objects

Public member function of class Rectangle:

```
void copyFrom(const rectangle& other)
{
    // if the input rectangle is initialized, copy its sides
    if(other.isInitialized())
    {
        resetRectangle();
        inputSides(other.getSide(1), other.getSide(2));
    }
}
```

A member function can have a **const input reference**, in this example copyFrom(const rectangle& other);
This imposes that the function *copyFrom* does not modify *other* in any way; in turn this means that *copyFrom* can call one of *other's* member functions only if that **function** is **defined** as **const**, i.e. a function that does not modify the object.

Operators applied to objects

For “simple” variables the behaviour of basic operators, such as “ = ” , “ + ” is straightforward (mostly – remember pointers?!) :

```
int x, y = 5;
x = y +2;
```

But what happens when those are applied to objects?

Let’s start with “ = ”

(1) Initialize one (previously instantiated) object with content of another

```
// testobj1=testobj0;
testobj1.copyFrom(testobj0);
```

(2) Instantiate and initialize one object with content of another

```
rectangle testobj1=testobj0;
```

Operators applied to objects

For “simple” variables the behaviour of basic operators, such as “ = ” , “ + ” is straightforward (mostly – remember pointers?!) :

```
int x, y = 5;
x = y +2;
```

But what happens when those are applied to objects?

Let’s start with “ = ”

(1) Initialize one (previously instantiated) object with content of another

```
// testobj1=testobj0;
testobj1.copyFrom(testobj0);
```

(2) Instantiate and initialize one object with content of another

```
rectangle testobj1=testobj0;
```

Solution: implement a ***copy constructor***

Operators applied to objects

```
    void basicInitialization()
    {
        // basic initialization (init_flag is still undetermined)
        init_flag=false;
        //set the ID and update the count.
        rectangle_ID=rectangle_instances_created_count++;
        rectangle_instances_alive_count++;
    }
public:
    // constructor
    rectangle() {basicInitialization();}
    // destructor
    ~rectangle() { ... }
    void copyFrom(const rectangle& other)
    {
        // if the input rectangle is initialized, copy its sides
        if(other.isInitialized())
        {
            resetRectangle();
            inputSides(other.getSide(1), other.getSide(2));
        }
    }
    //*
    // copy constructor
    rectangle(const rectangle& other)
    {
        basicInitialization();
        copyFrom(other);
        // temporarily: notify this has been done
        cout << " ====> Element copied via constructor" << endl;
    }
```

Operators applied to objects

```
void test_default_operators()
{
    rectangle testobj0;
    testobj0.inputSides(6,6);
    cout << " testobj0:"<< endl;
    testobj0.printRectangleInfo();
    rectangle::printRectangleCount();
    cout<<endl;
    {
        cout << "Entering local scope;"<< endl << endl;
        rectangle testobj1;
        //testobj1=testobj0;
        testobj1.copyFrom(testobj0);
        cout << " testobj1:"<< endl;
        testobj1.printRectangleInfo();
        rectangle::printRectangleCount();
        cout<<endl;

        //rectangle testobj2=testobj0;
        rectangle testobj2(testobj0);
        cout << " testobj2:"<< endl;
        testobj2.printRectangleInfo();
        rectangle::printRectangleCount();
        cout << "Exiting local scope;"<< endl << endl;
    }
    rectangle::printRectangleCount();
```

Operators applied to objects

```
void test_default_operators()
{
 ====> Element copied via constructor
 testobj2:
Rectangle ID is: 2
Rectangle side A is: 6
Rectangle side B is: 6
Rectangle area is: 36
Rectangle perimeter is: 24
Total numbers for Rectangle instantiations:
 TOT instatntiated Rectangles (currently active or not): 3
 TOT currently Active Rectangles: 3
 TOT currently Initialized Rectangles: 3
Exiting local scope;

Total numbers for Rectangle instantiations:
 TOT instatntiated Rectangles (currently active or not): 3
 TOT currently Active Rectangles: 1
 TOT currently Initialized Rectangles: 1
Press any key then Enter to finish
    cout << " testobj1:" << endl;
    testobj1.printRectangleInfo();
    rectangle::printRectangleCount();
    cout<<endl;

    //rectangle testobj2=testobj0;
    rectangle testobj2(testobj0);
    cout << " testobj2:"<< endl;
    testobj2.printRectangleInfo();
    rectangle::printRectangleCount();
    cout << "Exiting local scope;"<< endl << endl;
}
rectangle::printRectangleCount();
```

Operators applied to objects

```
void test_default_operators()
{
 ====> Element copied via constructor
 testobj2:
Rectangle ID is: 2
Rectangle side A is: 6
Rectangle side B is: 6
Rectangle area is: 36
Rectangle perimeter is: 24
Total numbers for Rectangle instantiations:
 TOT instatntiated Rectangles (currently active or not): 3
 TOT currently Active Rectangles: 3
 TOT currently Initialized Rectangles: 3
Exiting local scope;

Total numbers for Rectangle instantiations:
 TOT instatntiated Rectangles (currently active or not): 3
 TOT currently Active Rectangles: 1
 TOT currently Initialized Rectangles: 1
 Press any key then Enter to finish
    cout << " testobj1:"<< endl;
    testobj1.printRectangleInfo();
    rectangle::printRectangleCount();
    cout<<endl;

    rectangle testobj2=testobj0;
    cout << " testobj2:"<< endl;
    testobj2.printRectangleInfo();
    rectangle::printRectangleCount();
    cout << "Exiting local scope;"<< endl << endl;
}
rectangle::printRectangleCount();
```

Now, this also works: the copy constructor is called

Operators applied to objects

For “simple” variables the behaviour of basic operators, such as “ = ” , “ + ” is straightforward (mostly – remember pointers?!) :

```
int x, y = 5;
x = y +2;
```

But what happens when those are applied to objects?

Let’s start with “ = ”

(1) Initialize one (previously instantiated) object with content of another

```
// testobj1=testobj0;
testobj1.copyFrom(testobj0);
```

(2) Instantiate and initialize one object with content of another

```
rectangle testobj1=testobj0;
```

Solved by implementing a ***copy constructor***

Operators applied to objects

For “simple” variables the behaviour of basic operators, such as “ = ” , “ + ” is straightforward (mostly – remember pointers?!) :

```
int x, y = 5;
x = y +2;
```

But what happens when those are applied to objects?

Let’s start with “ = ”

(1) Initialize one (previously instantiated) object with content of another

```
// testobj1=testobj0;  ← Can we fix this now?
testobj1.copyFrom(testobj0);
```

(2) Instantiate and initialize one object with content of another

```
rectangle testobj1=testobj0;
```

Solved by implementing a ***copy constructor***

Operators applied to objects

Tools at our disposal:

- Cpp allows to overload **operators** such as “ = ” when applied to objects, i.e. to define a member function that is called on the object when the operator is found.

Operators applied to objects

Tools at our disposal:

- Cpp allows to overload **operators** such as " = " when applied to objects, i.e. to define a member function that is called on the object when the operator is found.

- Given a Cpp object my_class testobject; We call *memberfunction*() on the object via: testobject.memberfunction();
  Given a **pointer** to Cpp object
  my_class* testobject_ptr = &testobject;
  We call *memberfunction*() on the object via the pointer and "->" :
  testobject_ptr->memberfunction();
  (The same applies to accessing a member variable.)

Operators applied to objects

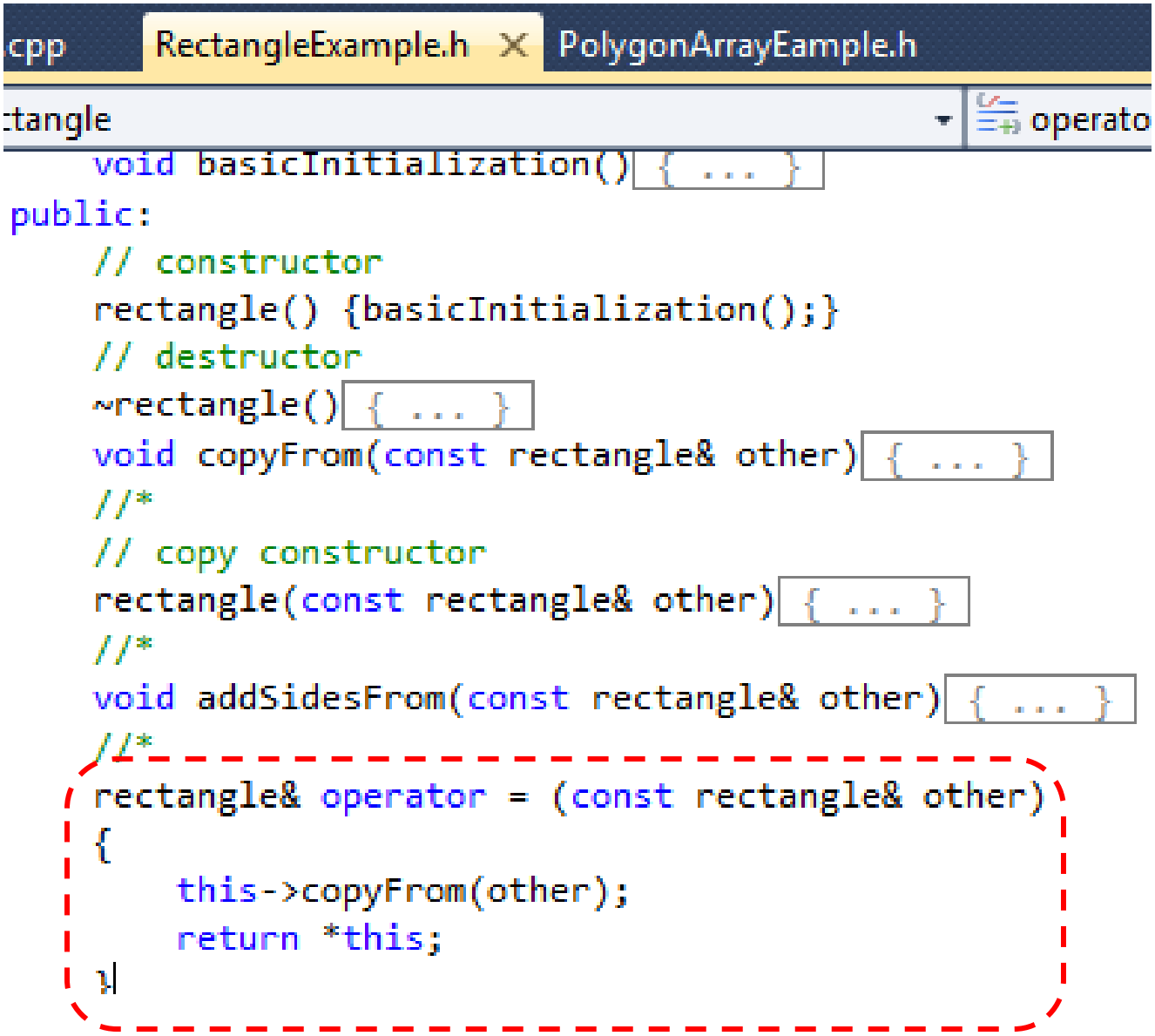
Tools at our disposal:

- Cpp allows to overload **operators** such as " = " when applied to objects, i.e. to define a member function that is called on the object when the operator is found.

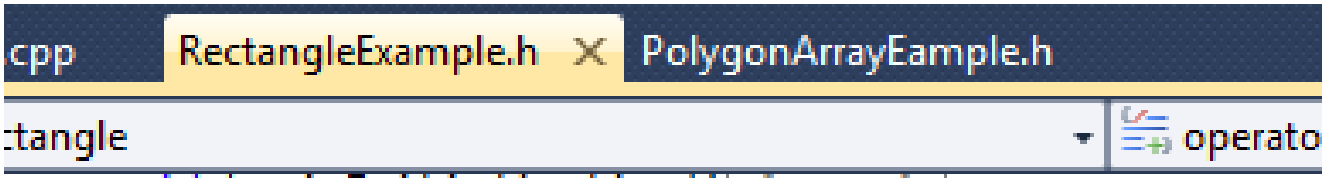
- Given a Cpp object my_class testobject; We call *memberfunction*() on the object via: testobject.memberfunction();
  Given a **pointer** to Cpp object
  my_class* testobject_ptr = &testobject;
  We call *memberfunction*() on the object via the pointer and "->" :
  testobject_ptr->memberfunction();
  (The same applies to accessing a member variable.)

- Any Cpp object possesses a member variable called "this", which is a pointer to the object itself.

Operators applied to objects



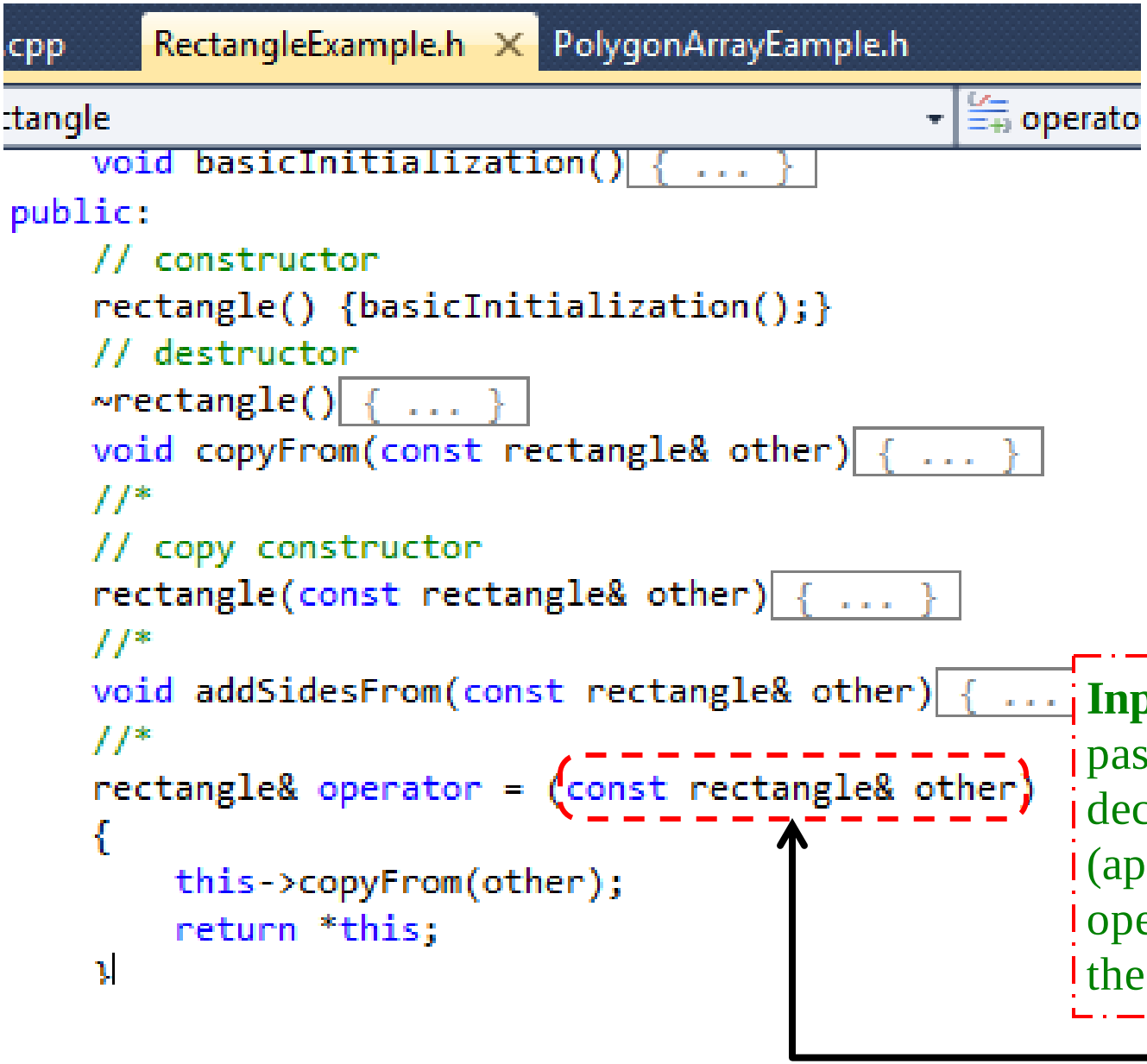
Operators applied to objects



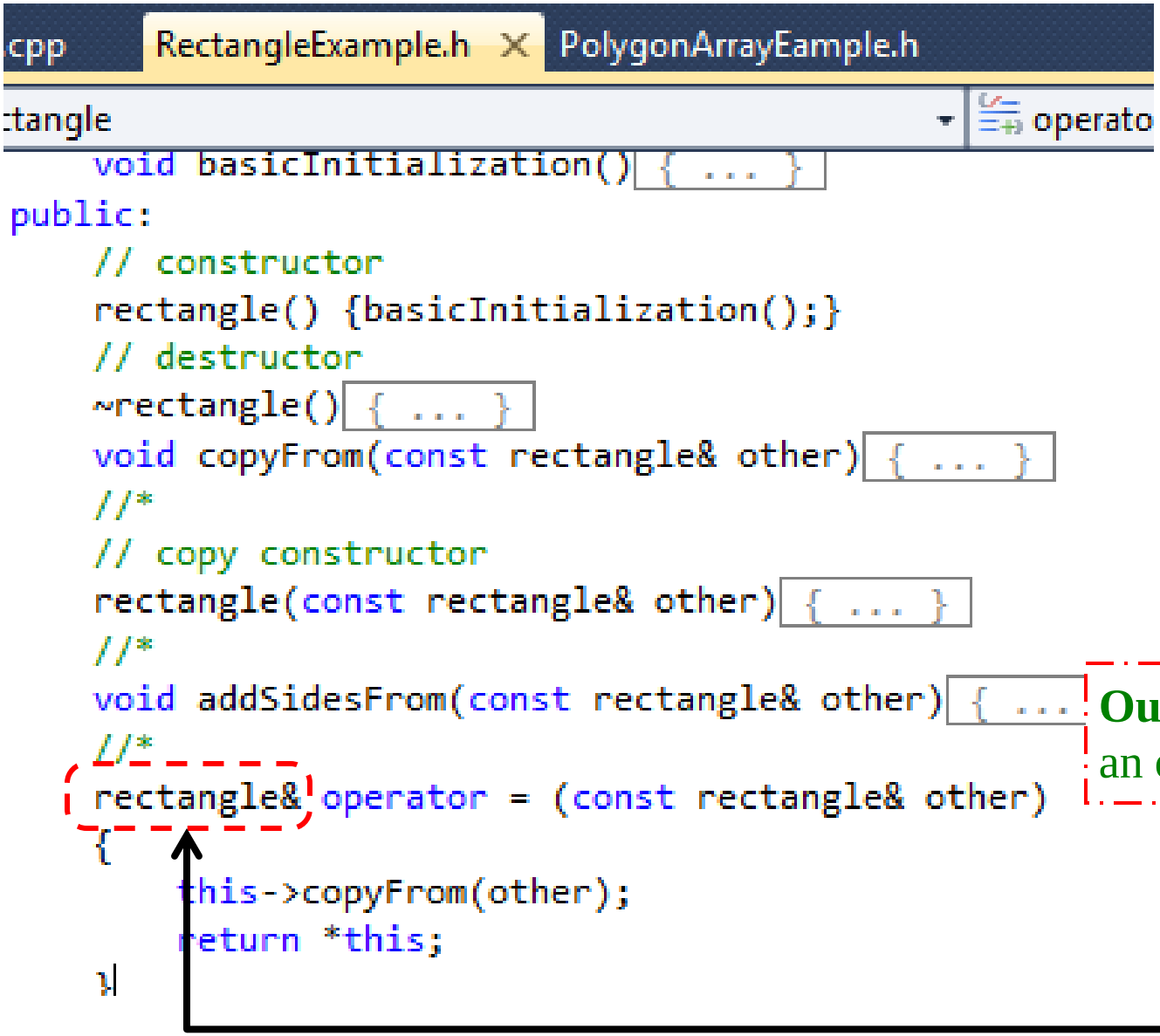

```
    void basicInitialization() { ... }
public:
    // constructor
    rectangle() {basicInitialization();}
    // destructor
    ~rectangle() { ... }
    void copyFrom(const rectangle& other) { ... }
    //*
    // copy constructor
    rectangle(const rectangle& other) { ... }
    //*
    void addSidesFrom(const rectangle& other) { ... }
    //*
    rectangle& operator = (const rectangle& other)
    {
        this->copyFrom(other);
        return *this;
    }
```

Keyword **operator** followed by the symbol (in this case " = ")

Operators applied to objects

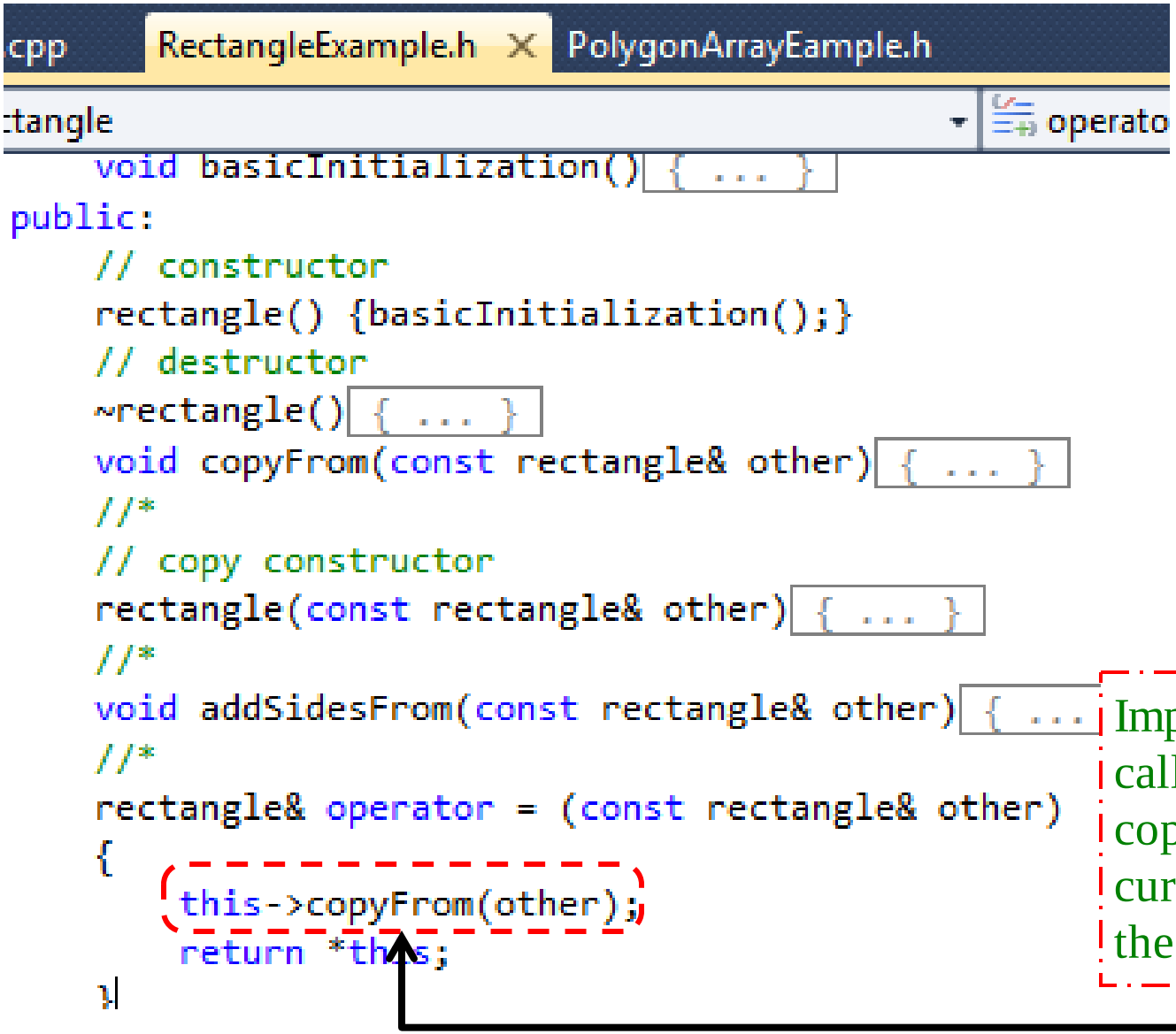


Operators applied to objects



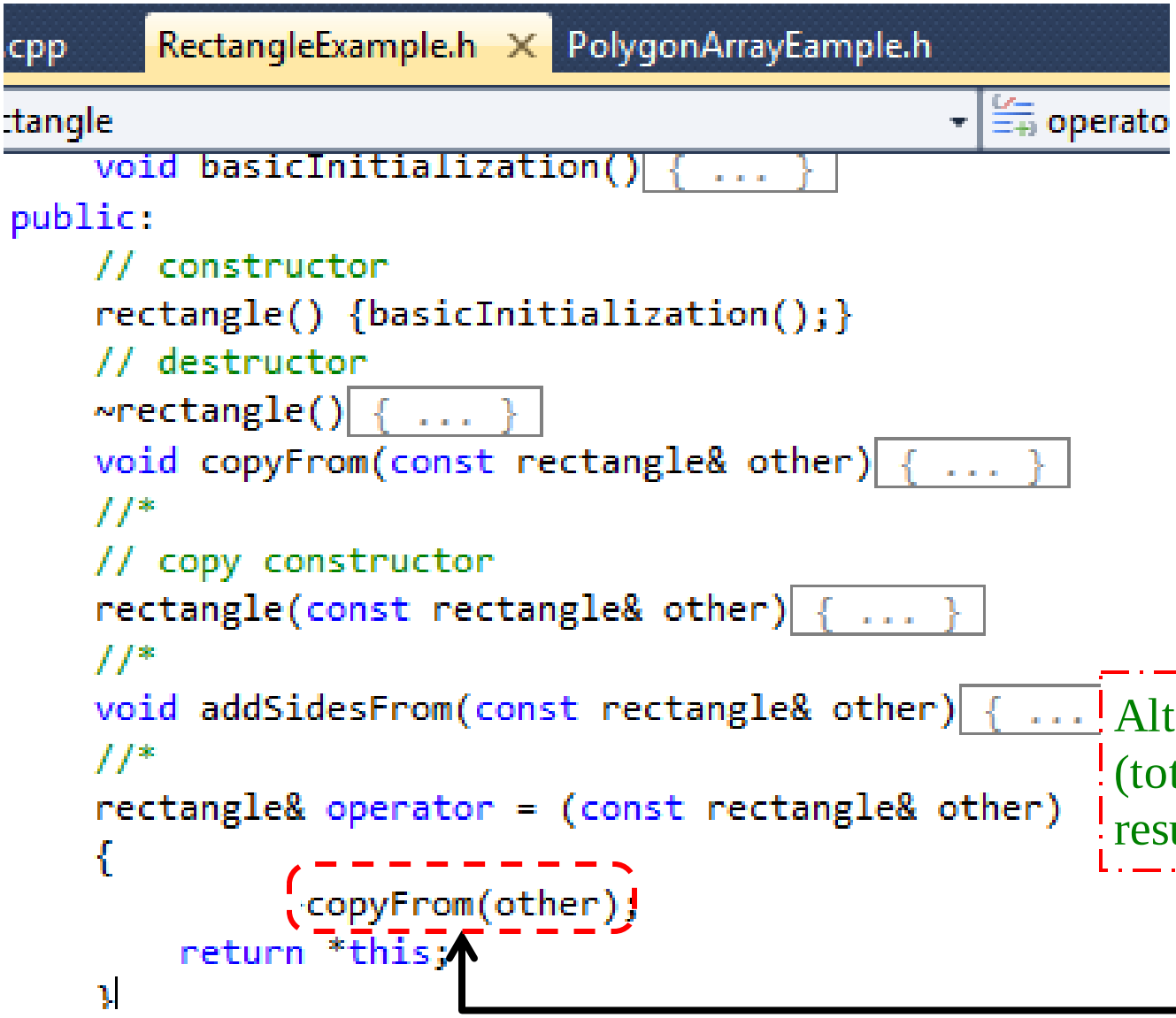
Output is the **reference** to an object

Operators applied to objects

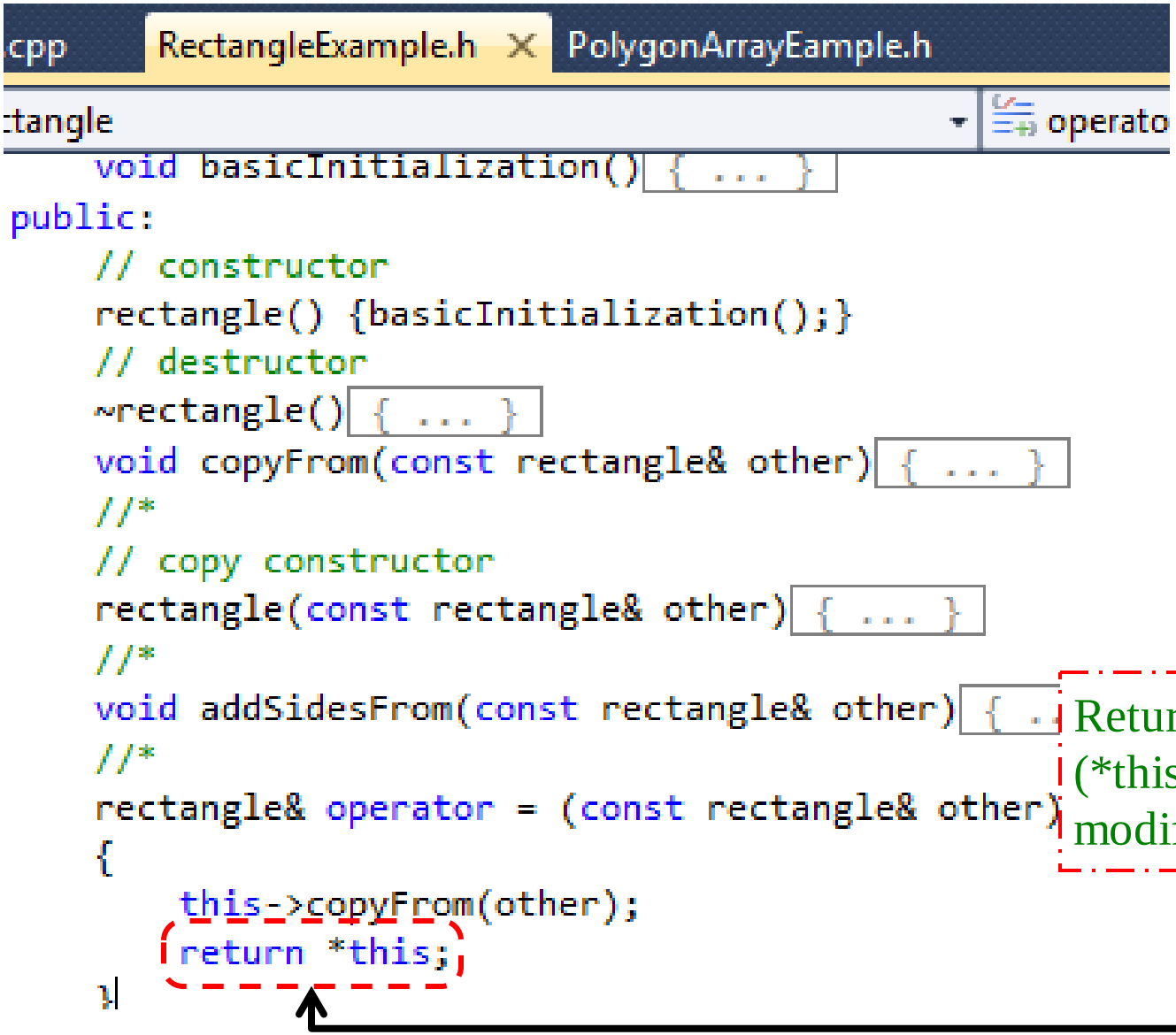


Implementation:
call member function copyFromOther(.) on the current (this) object using the “other” object as input

Operators applied to objects



Operators applied to objects



Return the current object (*this), after it has been modified by copyFromOther()

Operators applied to objects

Operator " = " is used and the relative **member function** is **called on the object on the left** side of the operator (testobj1);

```
void test_default_operators()
{
    rectangle testobj0;
    testobj0.inputSides(6,6);
    cout << " testobj0:"<< endl;
    testobj0.printRectangleInfo();
    rectangle::printRectangleCount();
    cout<<endl;
    {
        cout << "Entering local scope;"<< endl << endl;
        rectangle testobj1;
        // the following line works when the opertor = is overloaded
        testobj1=testobj0;
        // the following line works when copyFrom is defined
        //testobj1.copyFrom(testobj0);
        cout << " testobj1:"<< endl;
        testobj1.printRectangleInfo();
        rectangle::printRectangleCount();
        cout<<endl;
        // the following line works when the copy constructor is defined
        rectangle testobj2=testobj0;
        cout << " testobj2:"<< endl;
        testobj2.printRectangleInfo();
        rectangle::printRectangleCount();
        cout << "Exiting local scope;"<< endl << endl;
    }
    rectangle::printRectangleCount();
```

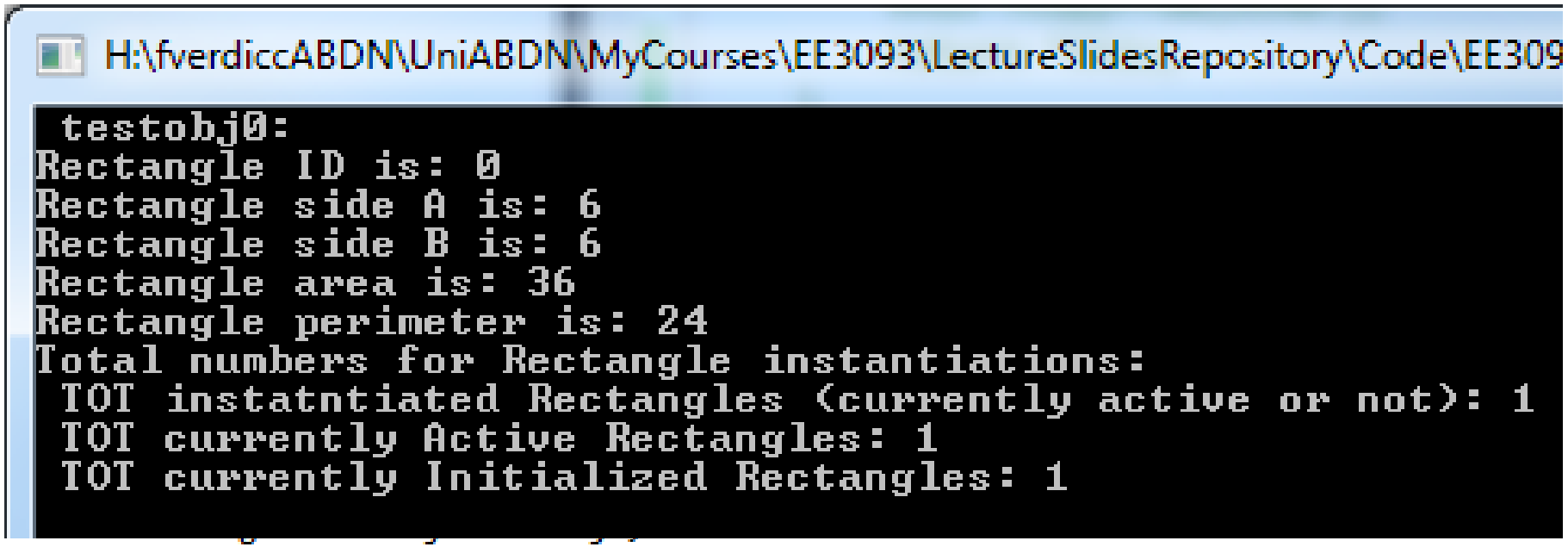
Operators applied to objects

```
void test_default_operators()
{
    rectangle testobj0;
    testobj0.inputSides(6,6);
    cout << " testobj0:"<< endl;
    testobj0.printRectangleInfo();
    rectangle::printRectangleCount();
    cout<<endl;
    {
        cout << "Entering local scope;"<< endl << endl;
        rectangle testobj1;
        // the following line works when the opertor = is overloaded
        testobj1=testobj0;
        // the following line works when copyFrom is defined
        //testobj1.copyFrom(testobj0);
        cout << " testobj1:"<< endl;
        testobj1.printRectangleInfo();
        rectangle::printRectangleCount();
        cout<<endl;
        // the following line works when the copy constructor is defined
        rectangle testobj2=testobj0;
        cout << " testobj2:"<< endl;
        testobj2.printRectangleInfo();
        rectangle::printRectangleCount();
        cout << "Exiting local scope;"<< endl << endl;
    }
    rectangle::printRectangleCount();
```

Operator " = " is used and the relative **member function** is **called on the object on the left** side of the operator (testobj1):
testobj1 performs copyFrom(testobj0)

Operators applied to objects

```
void test_default_operators()
{
    rectangle testobj0;
    testobj0.inputSides(6,6);
    cout << " testobj0:"<< endl;
    testobj0.printRectangleInfo();
    rectangle::printRectangleCount();
    cout<<endl;
```




```
        cout << " testobj2:"<< endl;
        testobj2.printRectangleInfo();
        rectangle::printRectangleCount();
        cout << "Exiting local scope;"<< endl << endl;
    }
    rectangle::printRectangleCount();
```

```
Entering local scope;
{
 testobj1:
Rectangle ID is: 1
Rectangle side A is: 6
Rectangle side B is: 6
Rectangle area is: 36
Rectangle perimeter is: 24
Total numbers for Rectangle instantiations:
 TOT instatntiated Rectangles (currently active or not): 2
 TOT currently Active Rectangles: 2
 TOT currently Initialized Rectangles: 2
```

```
    testobj1=testobj0;
    // the following line works when copyFrom is defined
    //testobj1.copyFrom(testobj0);
    cout << "  testobj1:"<< endl;
    testobj1.printRectangleInfo();
    rectangle::printRectangleCount();
    cout<<endl;
    // the following line works when
    rectangle testobj2=testobj0;
    cout << "  testobj2:"<< endl;
    testobj2.printRectangleInfo();
    rectangle::printRectangleCount();
    cout << "Exiting local scope;"<< endl << endl;
}
rectangle::printRectangleCount();
```

Operator " = " is used and the relative **member function** is **called on the object on the left** side of the operator (testobj1):
testobj1 performs copyFrom(testobj0);
The result (updated testobj1) goes to testobj1

Operators applied to objects

For “simple” variables the behaviour of basic operators, such as “ = ” , “ + ” is straightforward (mostly – remember pointers?!) :

```
int x, y = 5;
x = y +2;
```

But what happens when those are applied to objects?

Let’s start with “ = ”

(1) Initialize one (previously instantiated) object with content of another

```
testobj1=testobj0;
```

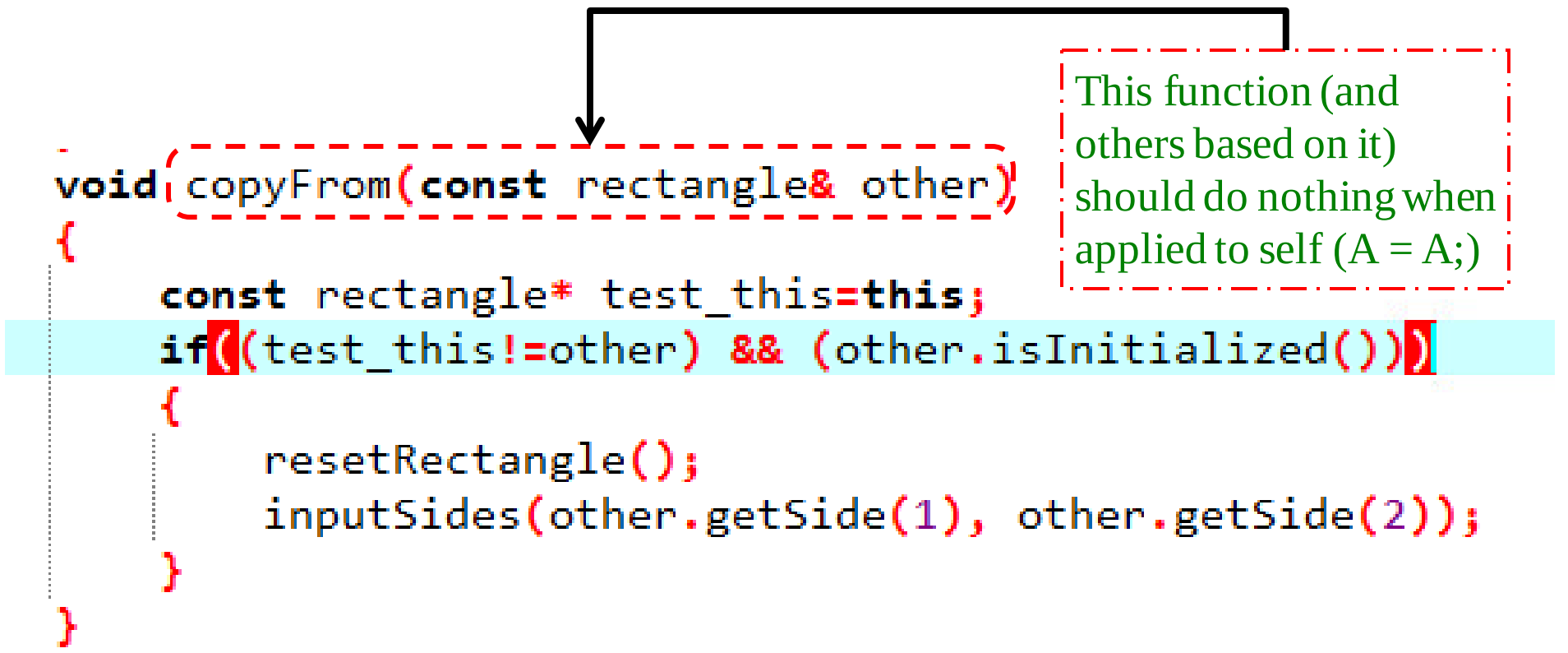
Solved by overloading ***operator =***

(2) Instantiate and initialize one object with content of another

```
rectangle testobj1=testobj0;
```

Solved by implementing a ***copy constructor***

Observation:




```
void copyFrom(const rectangle& other)
{
    const rectangle* test_this=this;
    if((test_this!=other) && (other.isInitialized()))
    {
        resetRectangle();
        inputSides(other.getSide(1), other.getSide(2));
    }
}
```

Observation:

```
void copyFrom(const rectangle& other)
{
    const rectangle* test_this=this;
    if((test_this!=other) && (other.isInitialized()))
    {
        resetRectangle();
        inputSides(other.getSide(1), other.getSide(2));
    }
}
```

This function (and others based on it) should do nothing when applied to self (A = A;)

Observation:

```
void copyFrom(const rectangle& other)
{
    const rectangle* test_this=this;
    if((test_this!=other) && (other.isInitialized()))
    {
        resetRectangle();
        inputSides(other.getSide(1), other.getSide(2));
    }
}
```

Typecast (from **rectangle*** to **const rectangle***) to perform comparison with **other** (required in later versions of C++ syntax; your compiler may need it)

Any question?