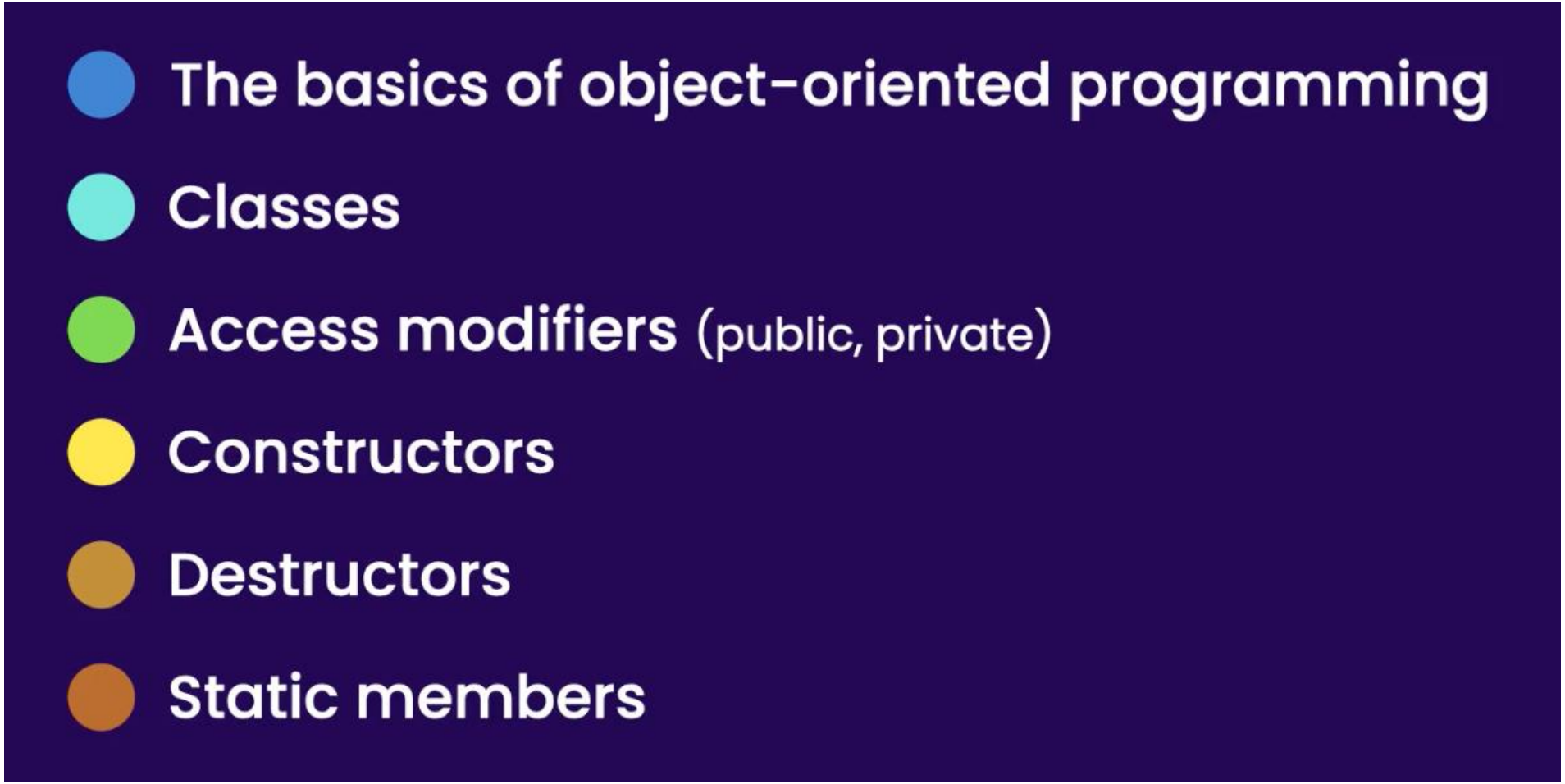
C++ Notes

Part 3-1

Slides were created from CodewithMosh.com

Classes


The basics of object-oriented programming
Classes
Access modifiers (public, private)
Constructors
Destructors
Static members

Introduction to Object-Oriented Programming

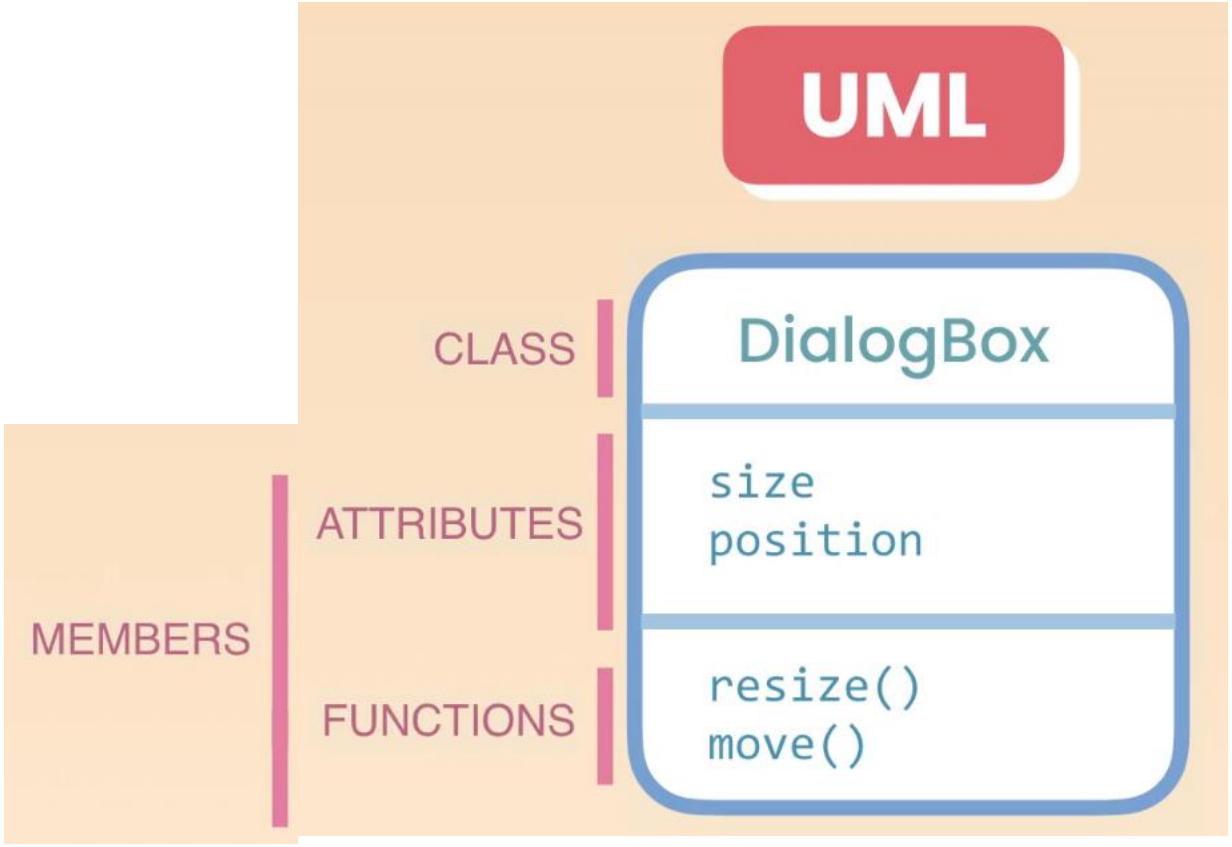
- Programming paradigms: Style of programming.
  - Procedural
  - **Functional (Centered on functions)**
  - **Object-oriented (Centered on objects)**
  - Event-driven
- These paradigms are not mutually exclusive.
- A good software engineer uses right tools for the right job.

Introduction to Object-Oriented Programming

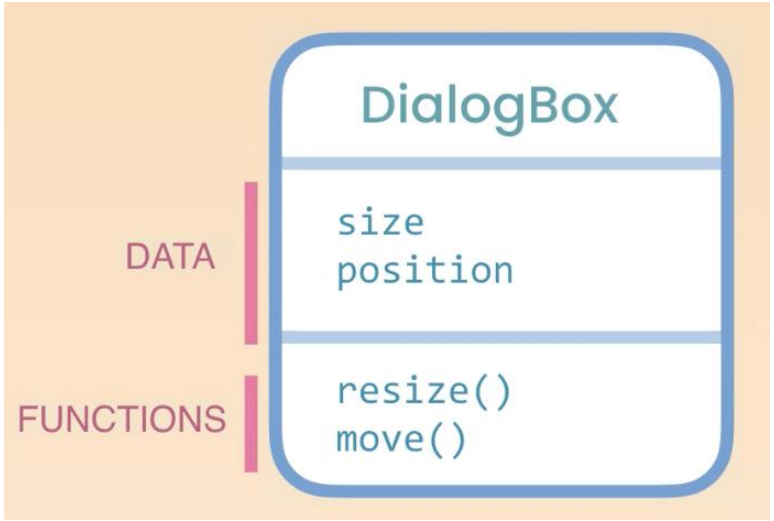
- **Object:** A software entity (a building block) that has **attributes (properties)** and **functions (methods)**.
- **Examples:** A dialog box (GUI) and a video player.

Introduction to Object-Oriented Programming

- **Class:** A blueprint (a recipe) for creating objects.
- **UML (unified Modeling Language):** Used to represent conceptual model.

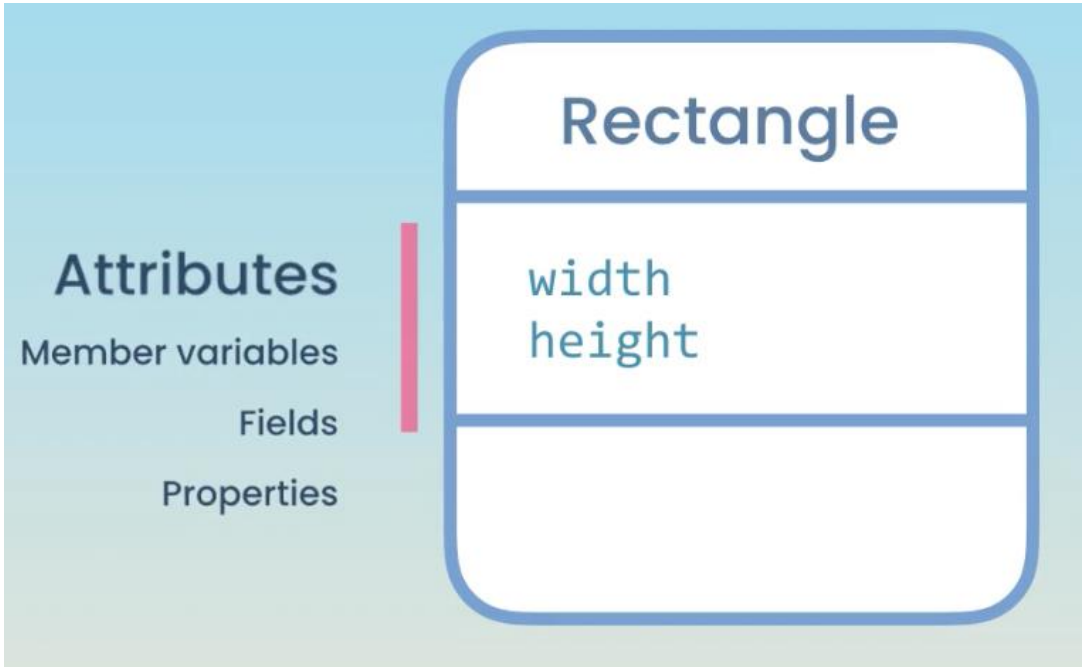


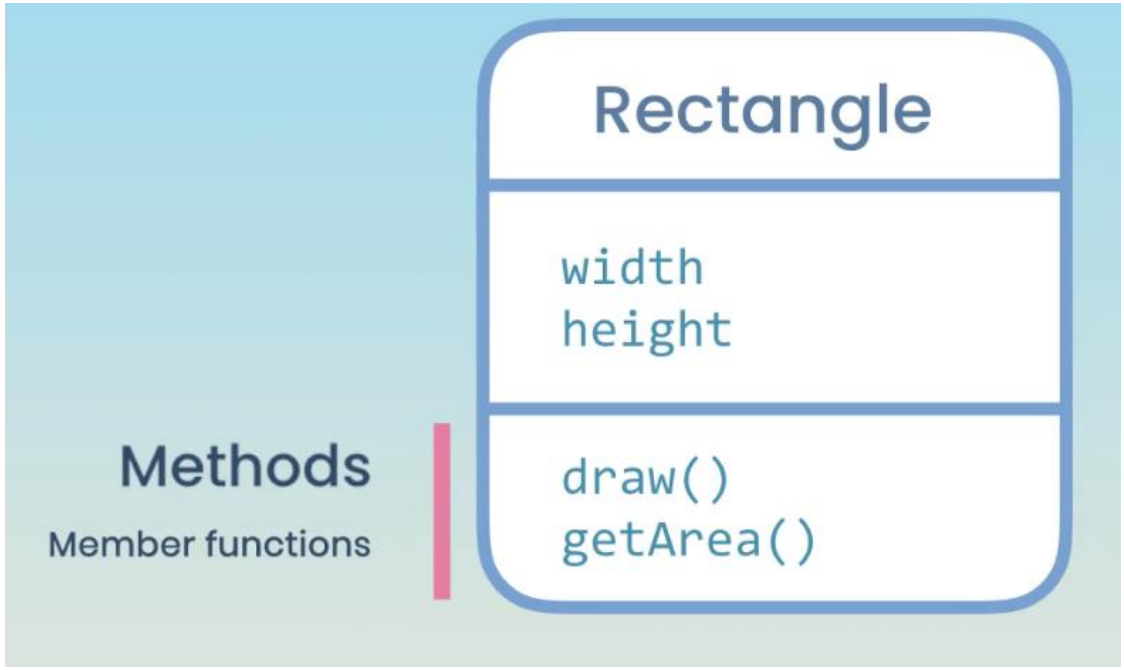
Introduction to Object-Oriented Programming



- **Encapsulation:** Combining the data and functions that operate on the data together into one unit.
- **Object:** An instance of a class.

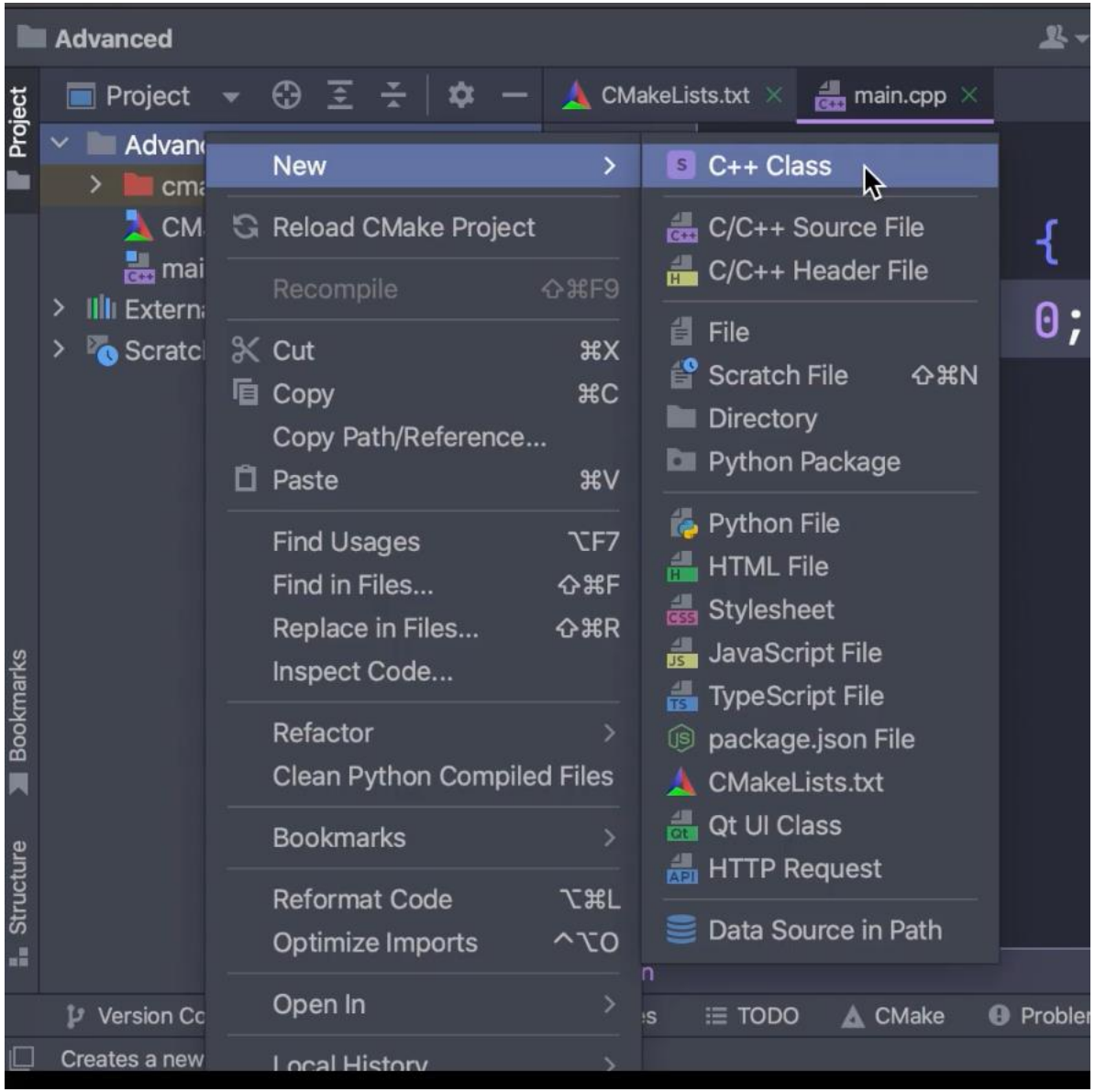
Defining a Class





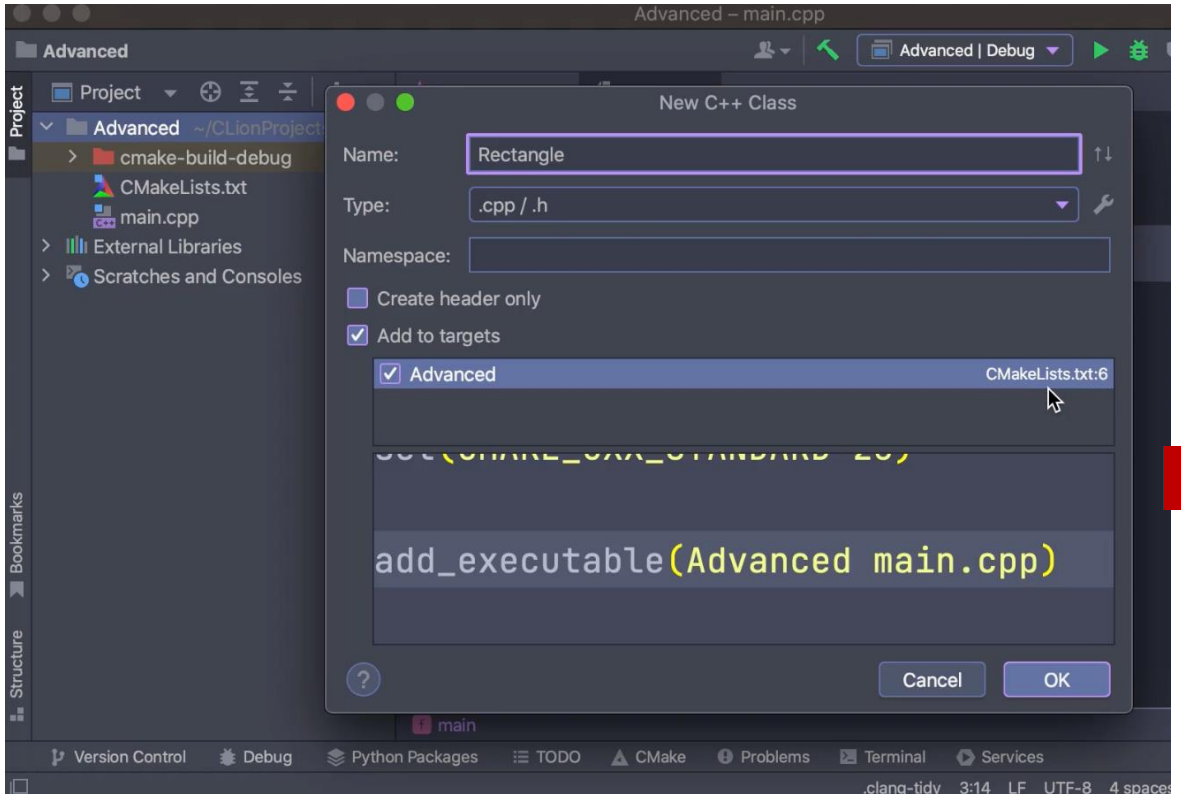
- **Officially in C++:** member variables and member functions.
- **In UML:** Attributes and methods are used.

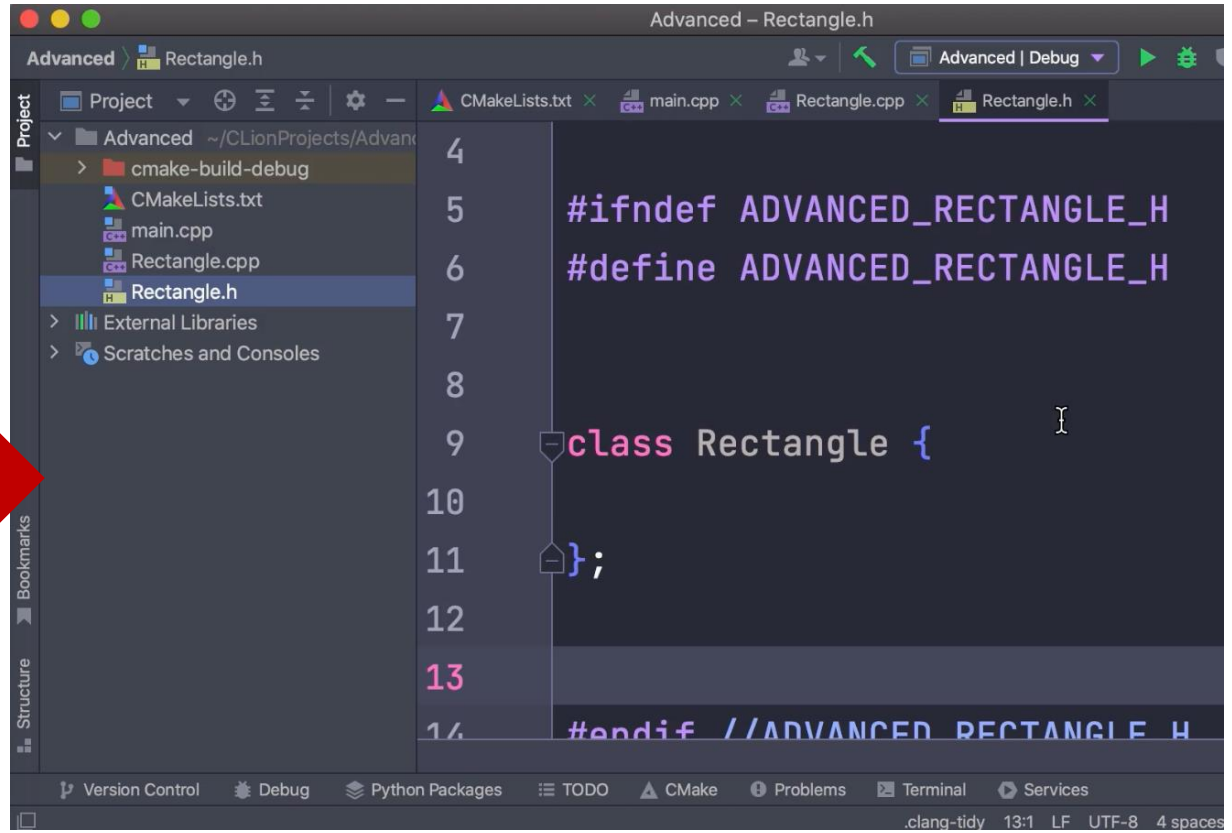
Defining a Class



- Right click on the project name and select **New**, then **C++ Class.**

Defining a Class



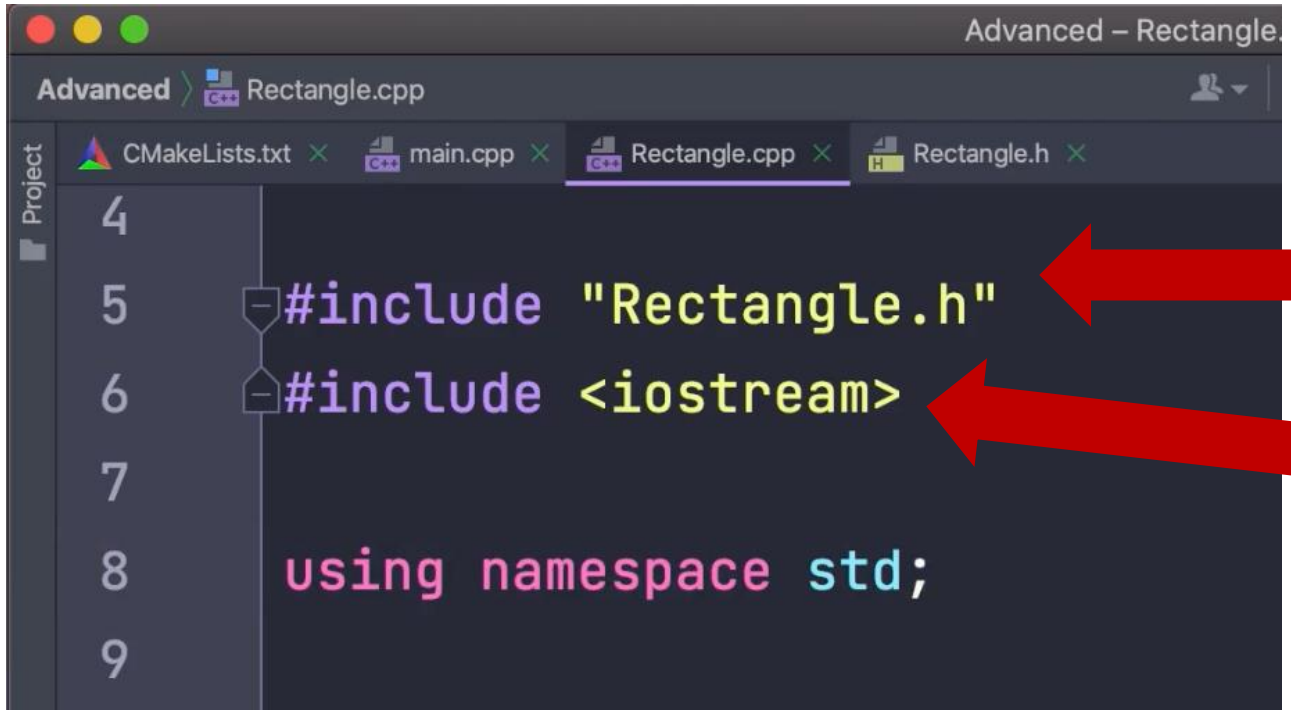


Defining a Class

```
#ifndef ADVANCED_RECTANGLE_H
#define ADVANCED_RECTANGLE_H

class Rectangle {
public:
    int width;
    int height;
    void draw();
    int getArea();
};

#endif //ADVANCED_RECTANGLE_H
```

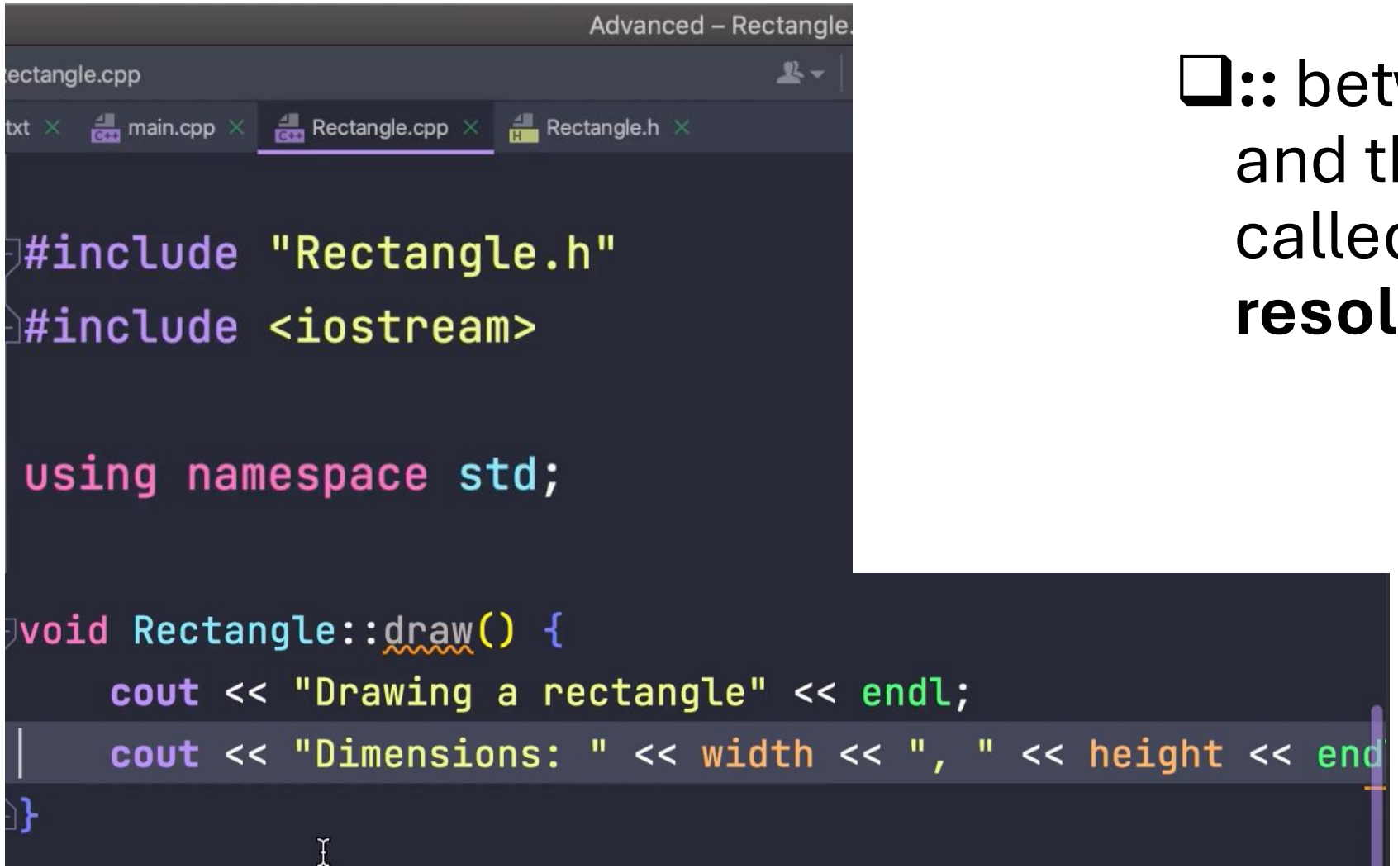
- The header file represents the interface of the futures.
- The header file will be included in the main project file, Rectangle.cpp, file to access the futures.
- By using keyword “public”, all the private members are public and accessible outside of the class.
- Keyword “**public**” is an **access modifier.**
- Others are “**private**” and “**protected**”.

Defining a Class



- Include the header file in .cpp main file.
- Files we create in the project are included using “ “.
- Standard C++ files are included using < >.

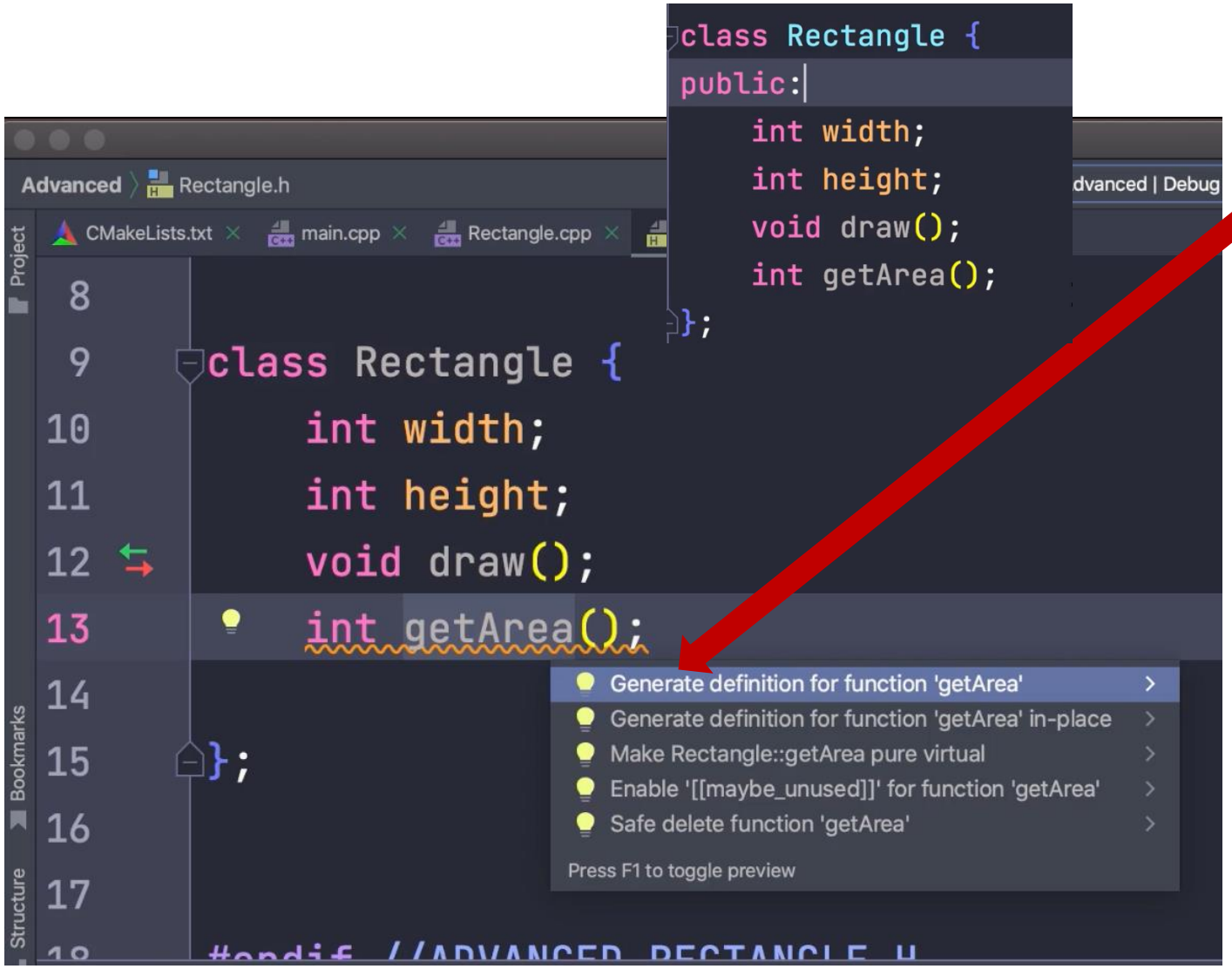
Defining a Class



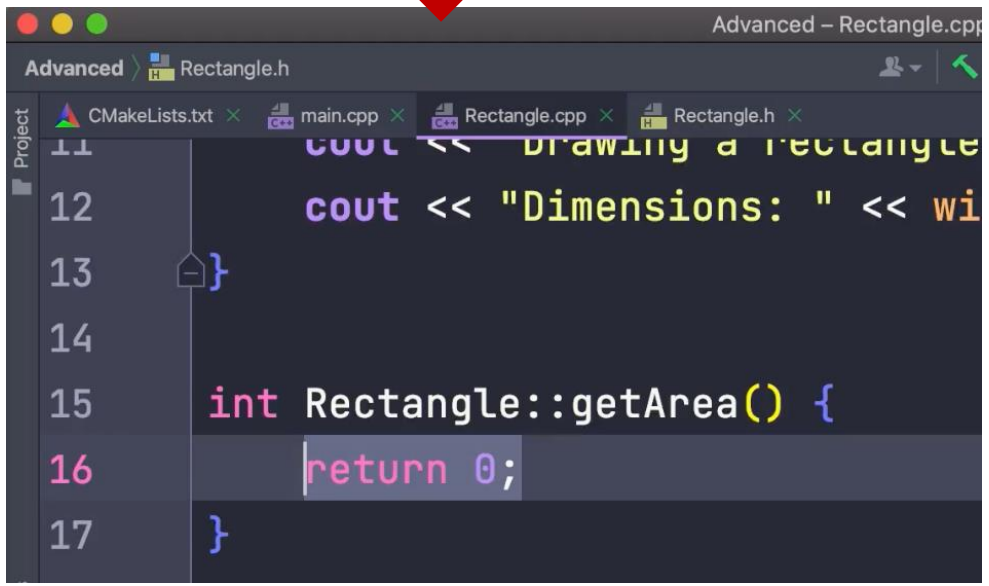
- **::** between the class name and the function name is called the "**scope resolution operator**".

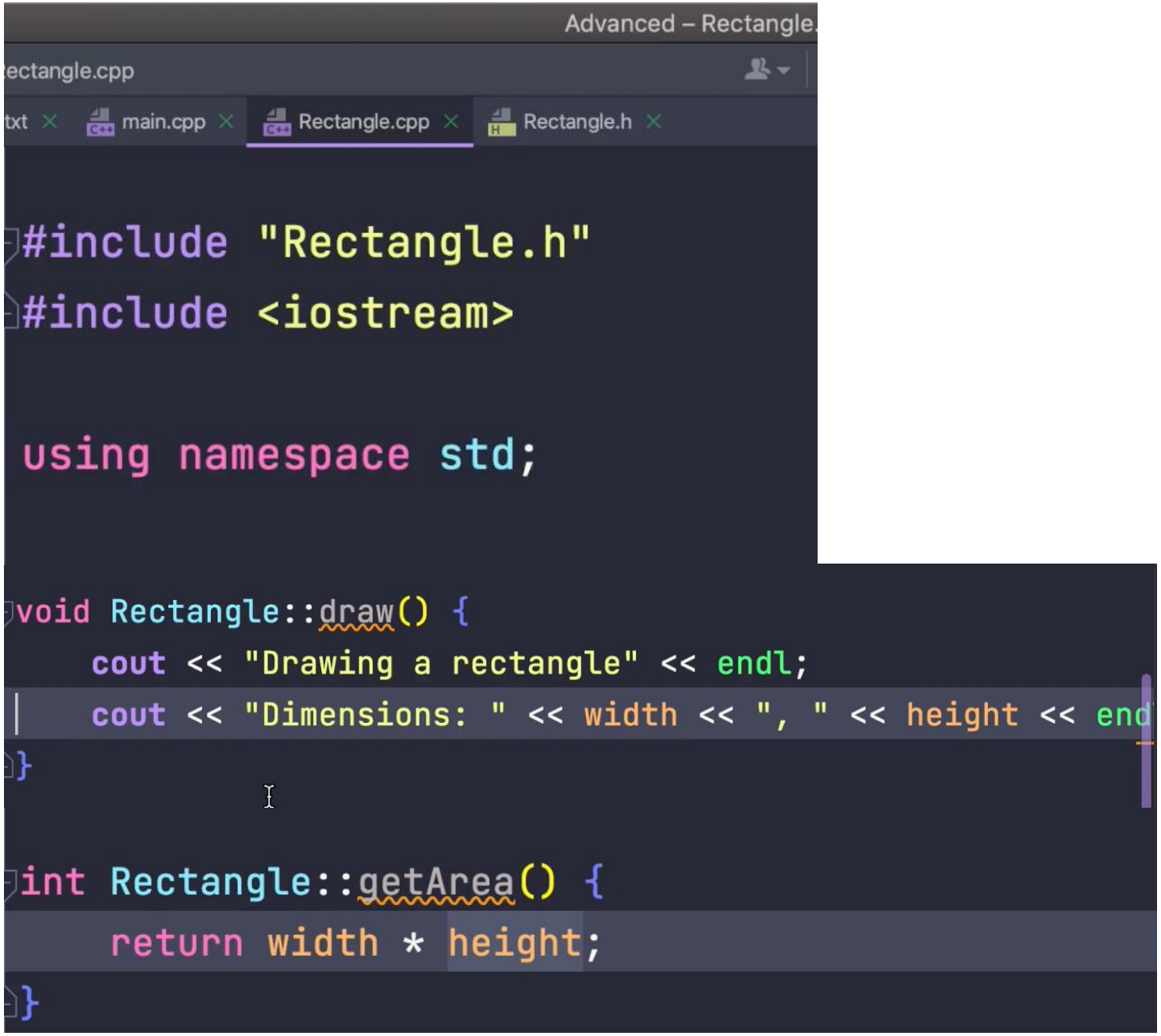
- Implementation of draw function (method).

Defining a Class



- Select “light bulb” icon or press CTRL(or ALT)+ENTER on the name of the function (getArea) for C++ to create the definition of the function in the main project file, Rectangle.cpp.

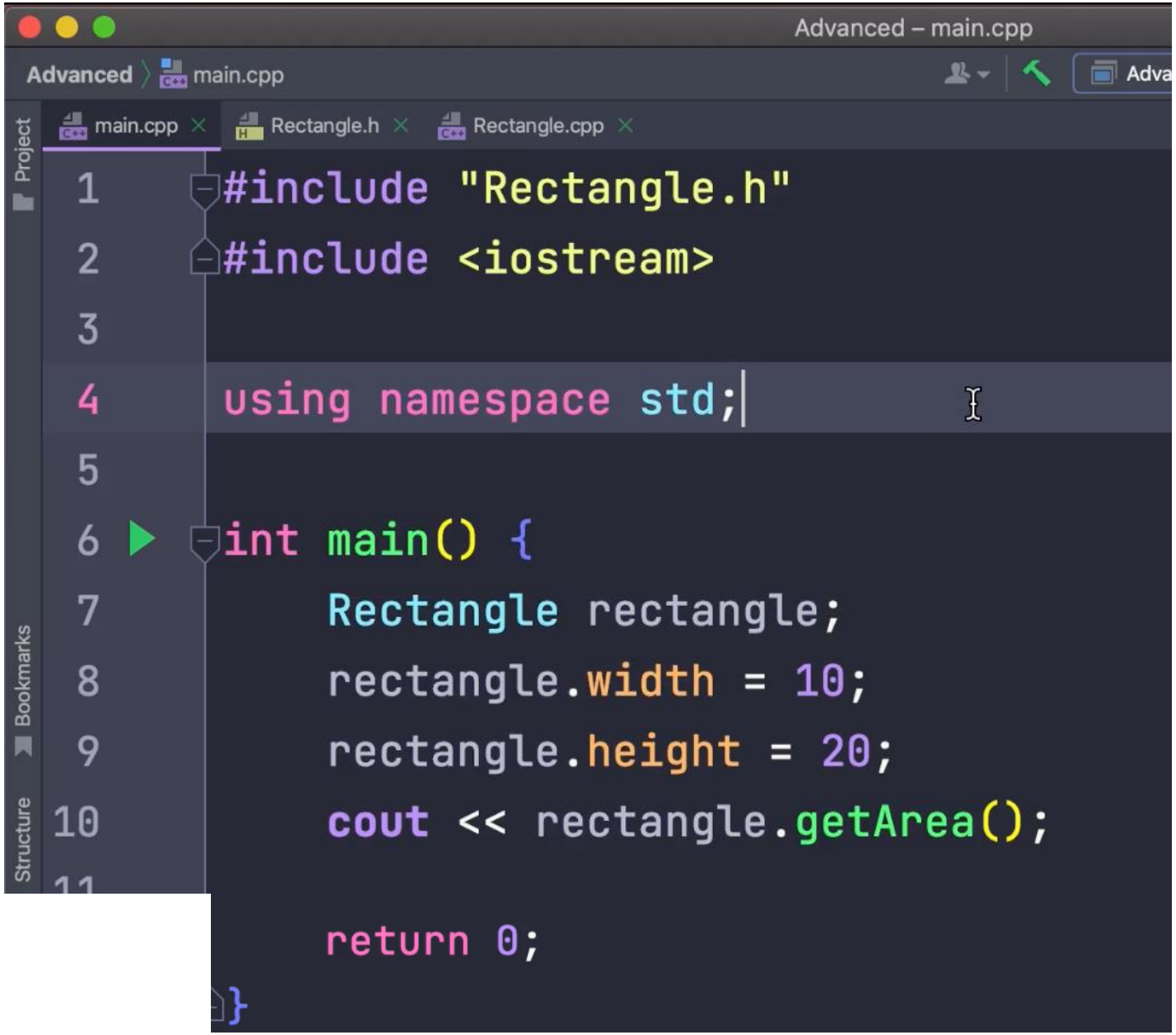




Defining a Class

- **::** between the class name and the function name is called the "**scope resolution operator**".
- Implementation of draw function (method).
- Implementation of getArea function (method).

Creating Objects



- Having two files per class, header (.h) and implementation (.cpp), is to reduce compilation time.
- If there is a change in the header file, then every file that is depended on this file need to be recompiled.
- If there is a change in implementation file, then only the implementation file needs to be recompiled.
- Most of the time, the implementation file is changed.

Creating Objects

Advanced – main.cpp

Advanced > main.cpp

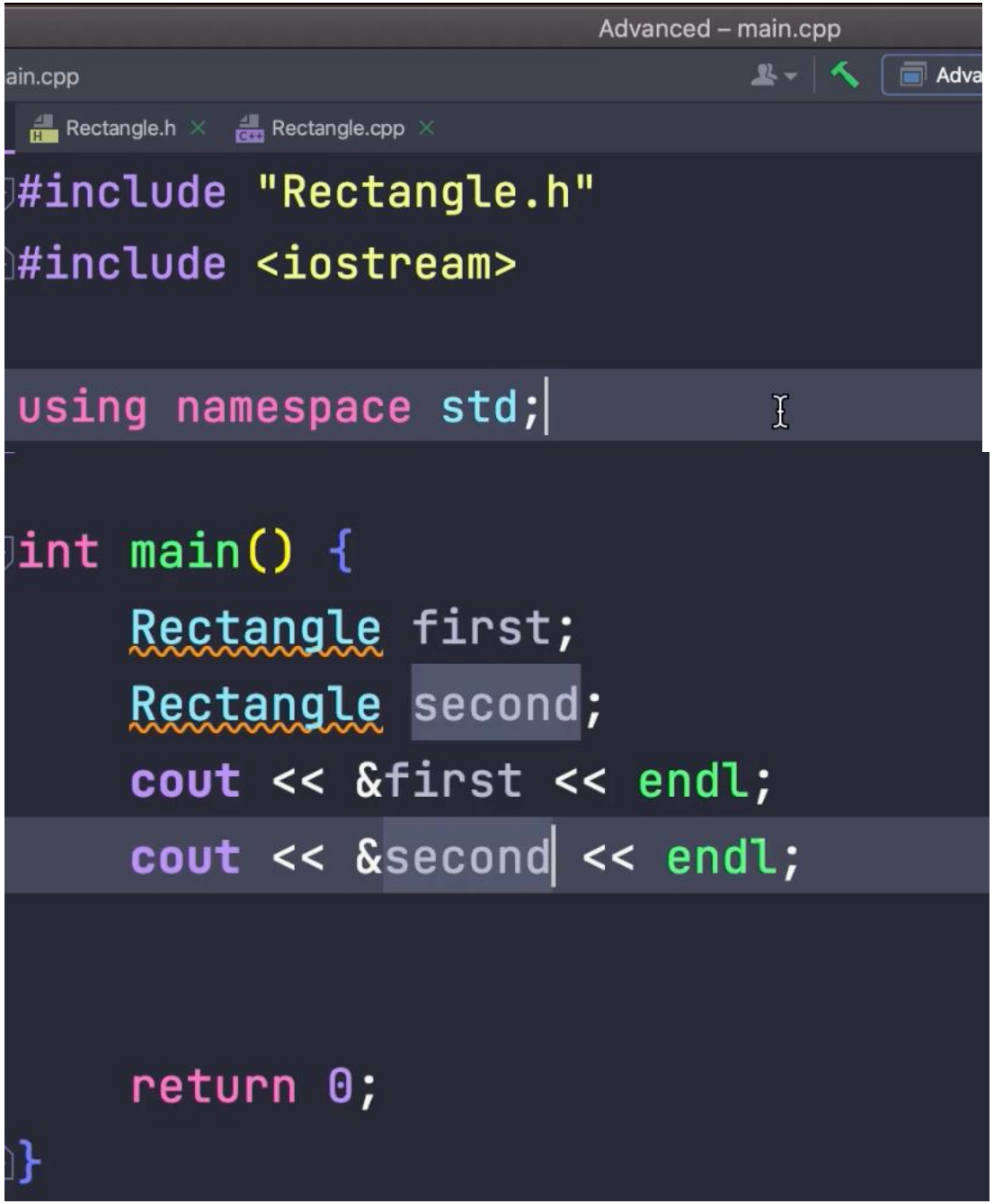
main.cpp | Rectangle.h | Rectangle.cpp

```cpp
#include "Rectangle.h"
#include <iostream>

using namespace std;

int main() {
    Rectangle rectangle;
    rectangle.width = 10;
    rectangle.height = 20;
    cout << rectangle.getArea();

    return 0;
}
```

```cpp
class Rectangle {
public:
    int width;
    int height;
    void draw();
    int getArea();
};
```

- To access the members of a class from outside, they need to be declared as public.
- The default is private.

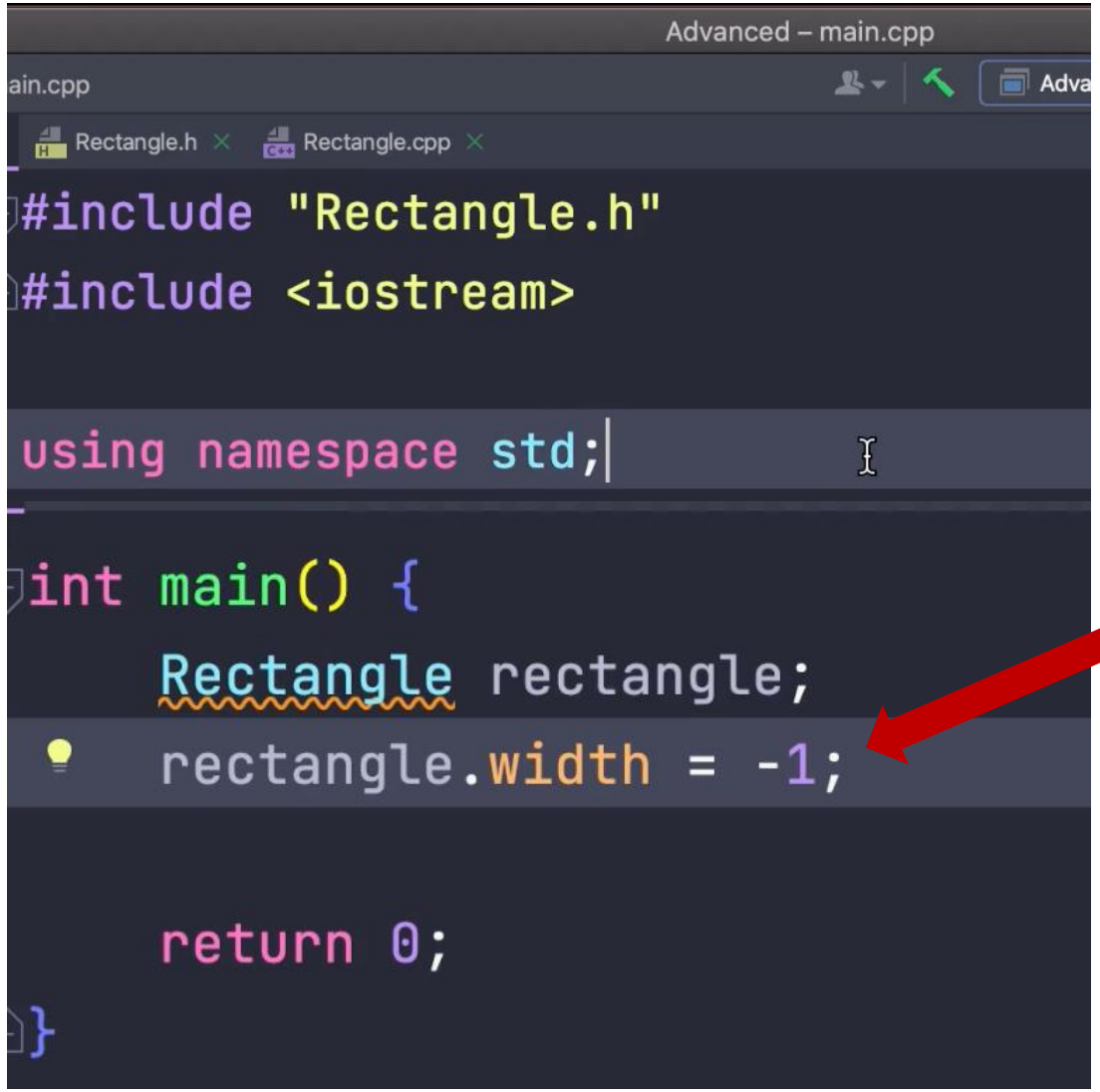
Creating Objects



- The **first** and **second** are two separate **objects** of the Rectangle **class**.

```
0x7ffeeb6389c0
0x7ffeeb6389b8

Process finished with exit code 0
```

- Memory addresses printed.



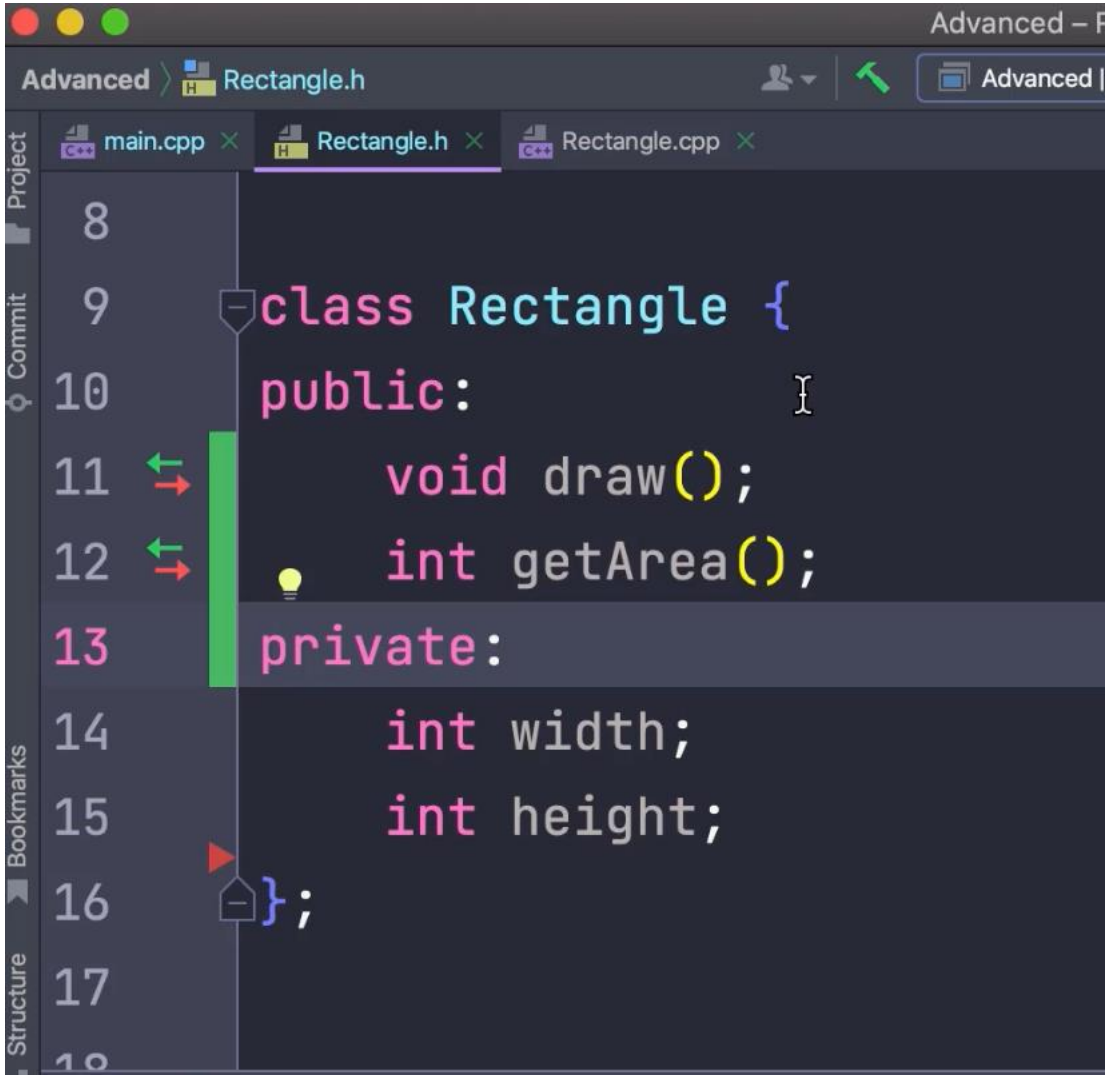
Access Modifiers

- The state of an object is the data stored in it.
- If we assign an invalid data, then the object gets into a bad state and will not be able to behave properly.
- **Solution:** Use **data hiding principle** of OOP in C++.

Data Hiding

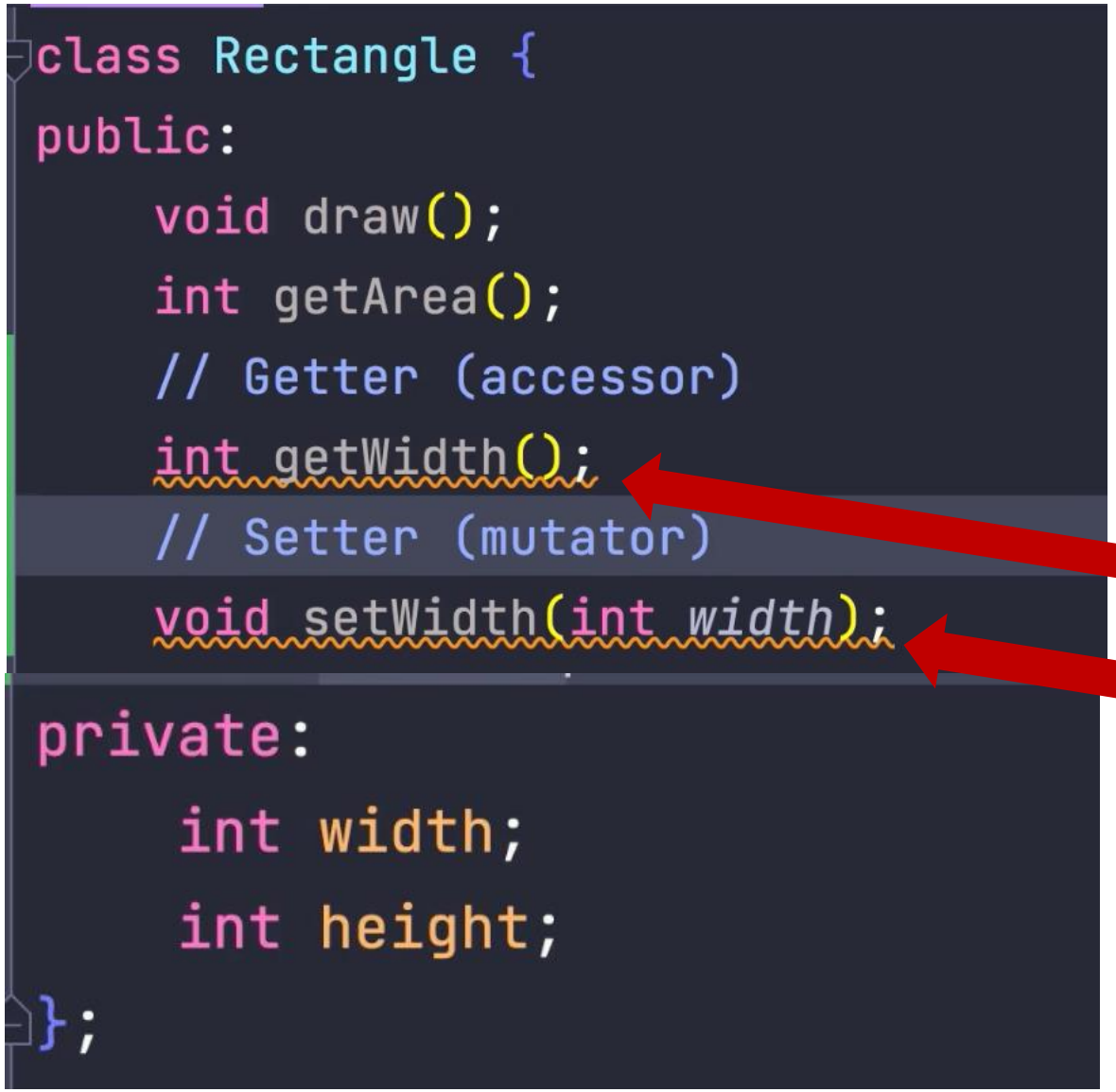
A class should hide its internal data from the outside code and provide functions for accessing the data.

Access Modifiers



- **Solution:** Use **data hiding principle** of OOP in C++.
- Make width and height private.
- Now, we need to provide functions in the class to access private variables.

Getters and Setters



- **Solution:** Use **data hiding principle** of OOP in C++.
- Make width and height private.
- Now, we need to provide functions in the class to access private variables.

Getters and Setters

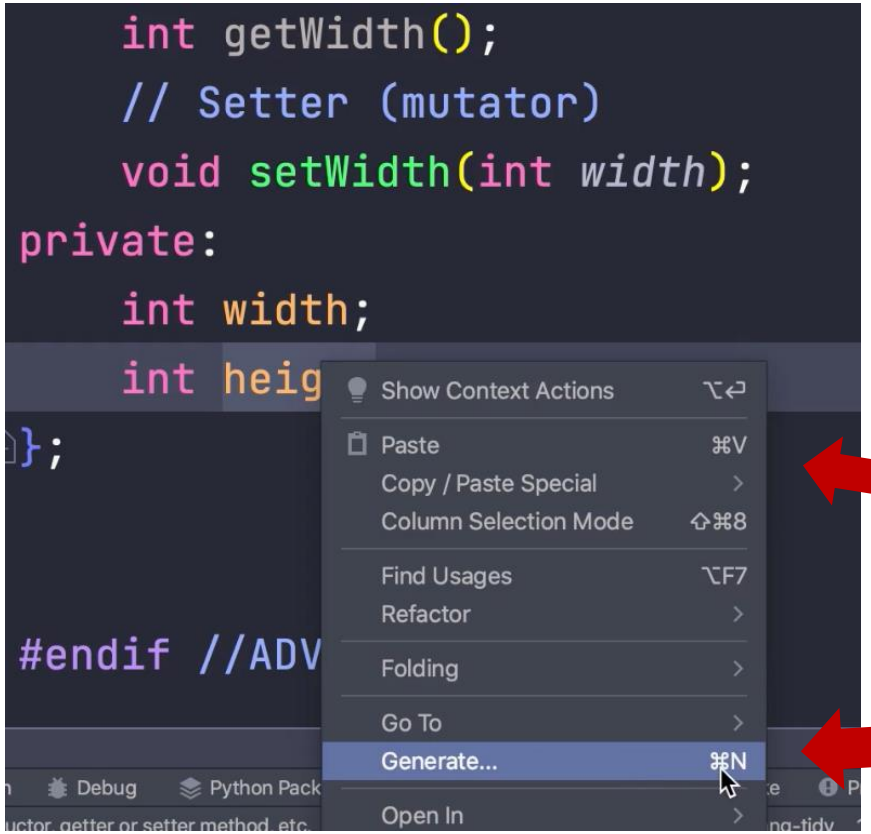
```
void Rectangle::setWidth(int width) {
    if (width < 0)
        throw invalid_argument( s: "width");
    this->width = width;
}
```

- Implementation of **setWidth** function.
- This is an exception.
- **this->** refers to the current object.
- **(*this).width, m_width, or Rectangle::width** can also used.

```
int main() {
    Rectangle rectangle;
    rectangle.setWidth(-1);

    return 0;
}
```

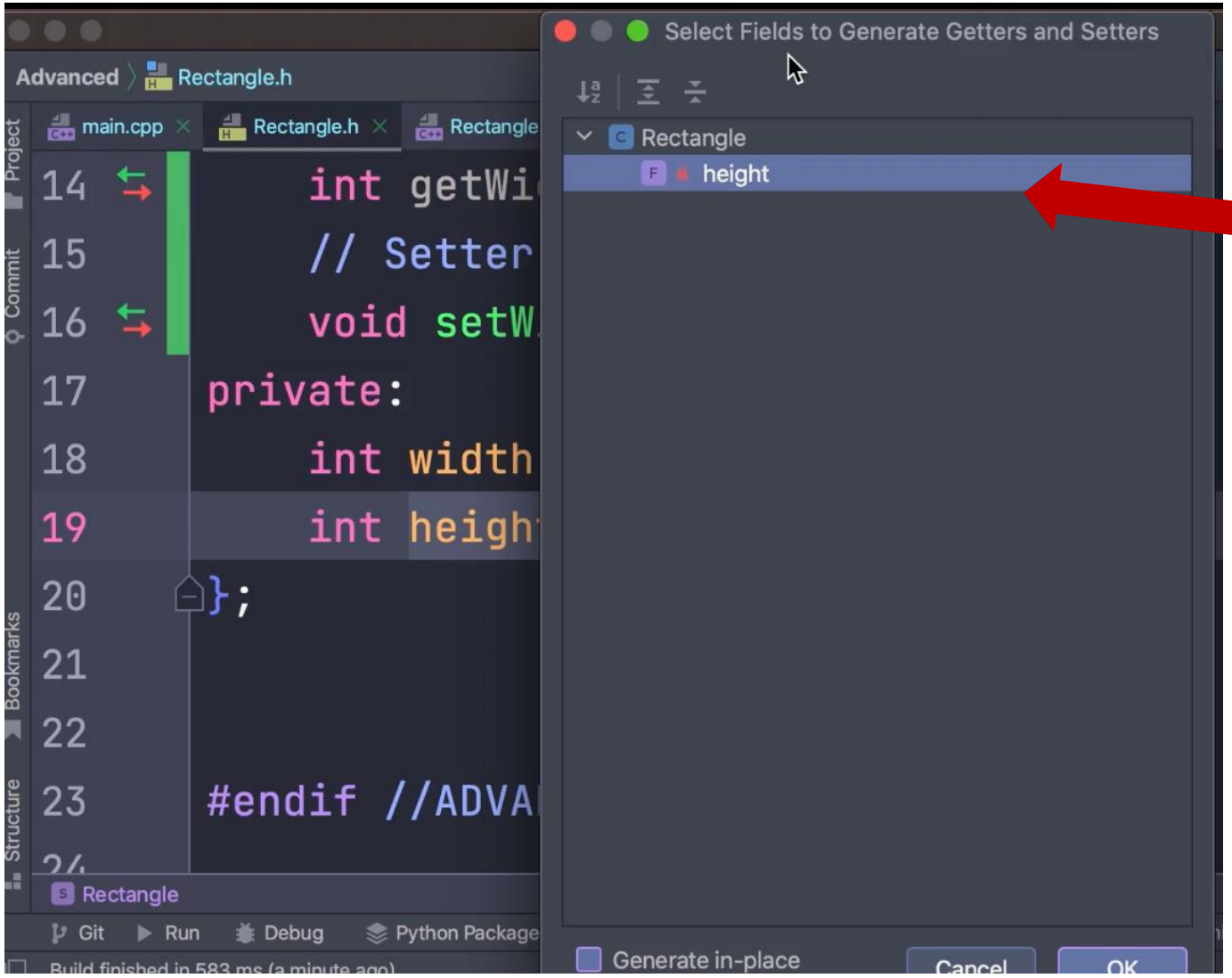
- Calling the **setWidth** function in the **main** function.

Getters and Setters



- ❑Create getter and setter functions for variable **height.**
- ❑Right click on the variable height.
- ❑Select **Generate**
- ❑Select **Getter and Setter**
- ❑If we select **Getter**, then the variable will be read-only.

Getters and Setters



- ❑Select the attribute.
- ❑**Attributes** are also called **fields**.

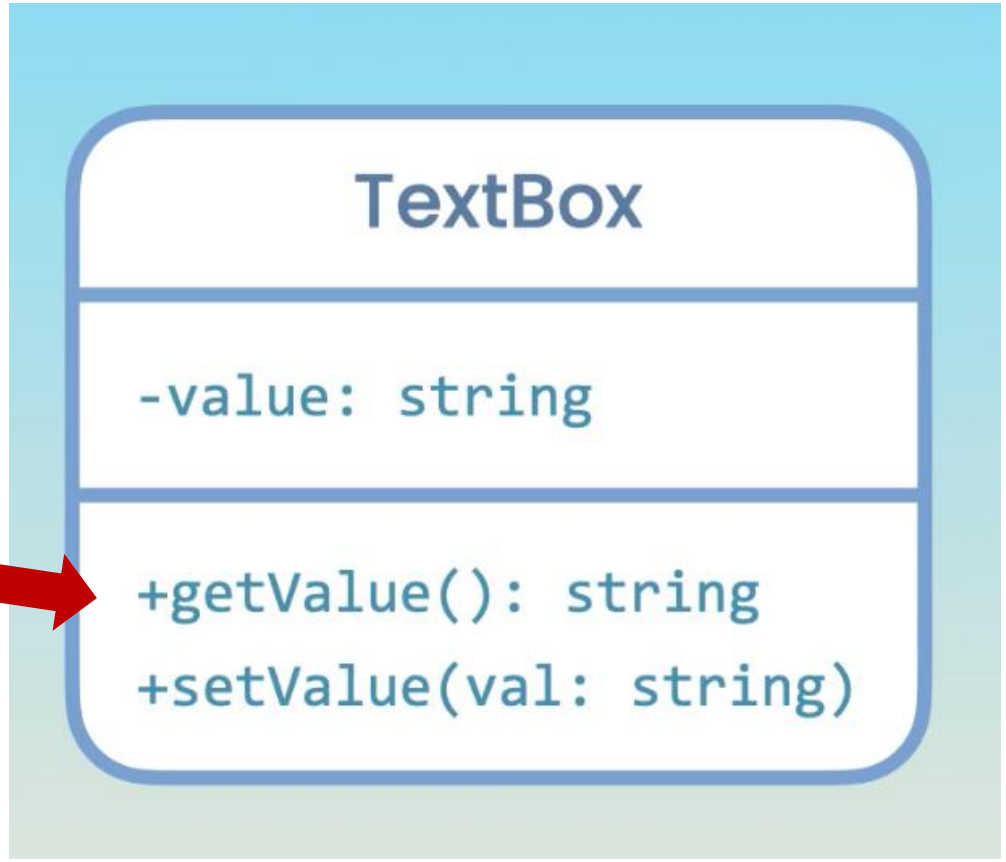
Getters and Setters

```
int Rectangle::getHeight() const {
    return height;
}

void Rectangle::setHeight(int height) {
    if (height < 0)
        throw invalid_argument("heigh")
    this->height = height;
}
```

- Implementation of **setHeight** function.

Example

- **+** means these methods are **public**.



Example

```
#define ADVANCED_TEXTBOX_H

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

Advanced | TextBox.h — Advanced | Debug

main.cpp | TextBox.cpp | TextBox.h

```
10
11  class TextBox {
12  public:
13      string getValue();
14      void setValue(const string& value);
15  private:
16      string value;
17  };
18
19
20  #endif //ADVANCED_TEXTBOX_H
```

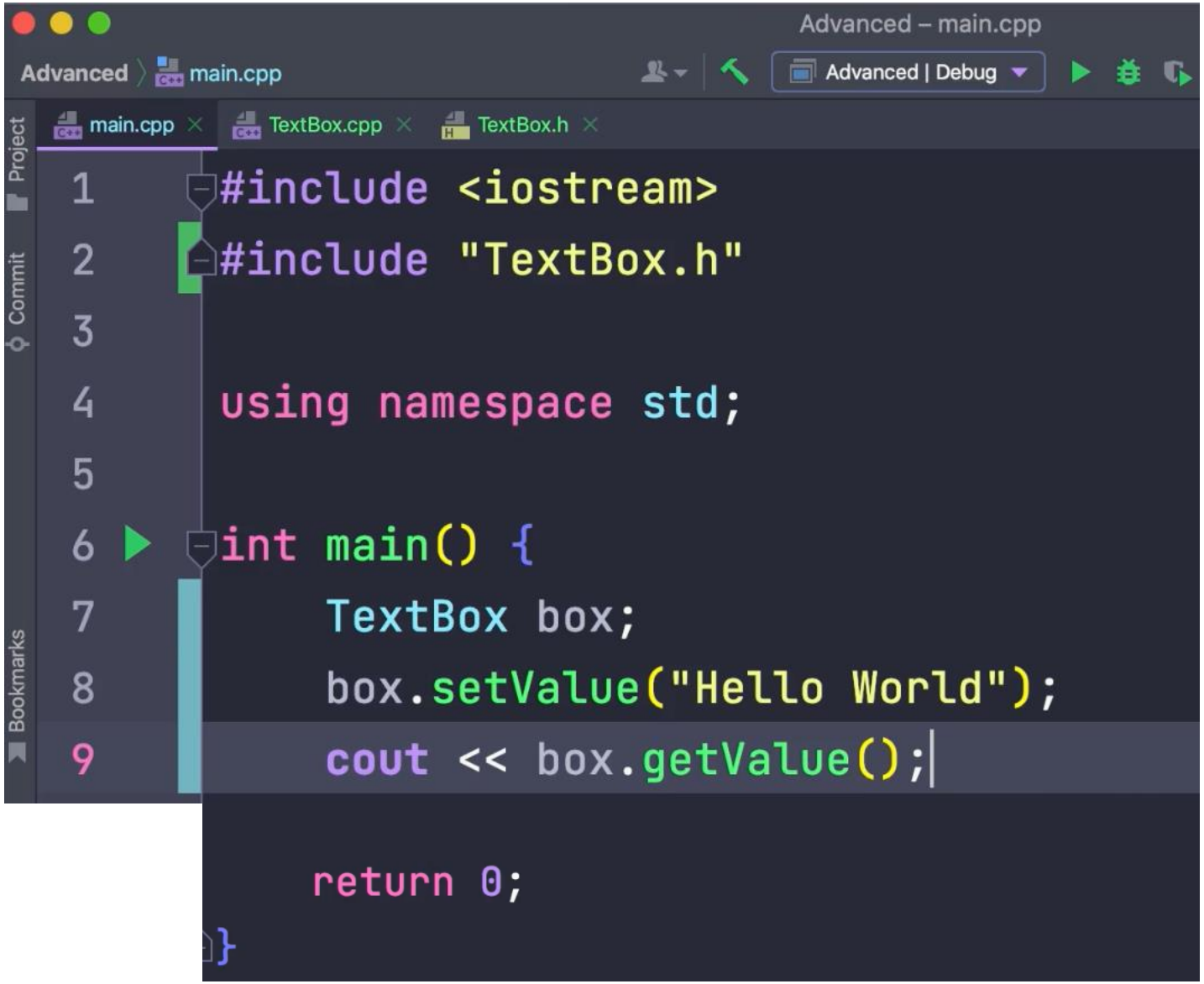
Advanced – TextBox.cpp

Advanced | TextBox.cpp — Advanced | Debug

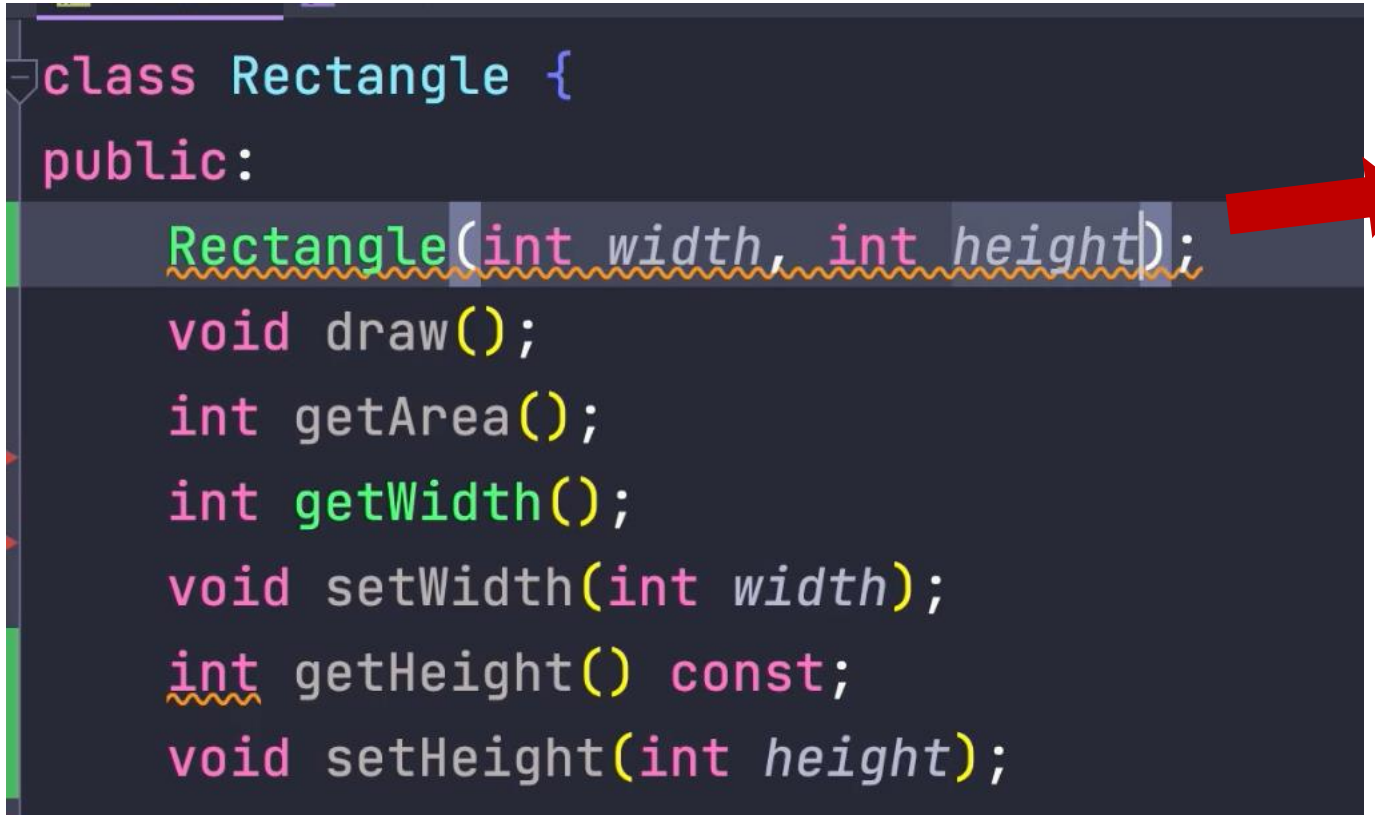
main.cpp | TextBox.cpp | TextBox.h

```
4
5   #include "TextBox.h"
6
7   string TextBox::getValue() {
8       return value;
9   }
10
11  void TextBox::setValue(const string& value) {
12      this->value = value;
13  }
```

Example

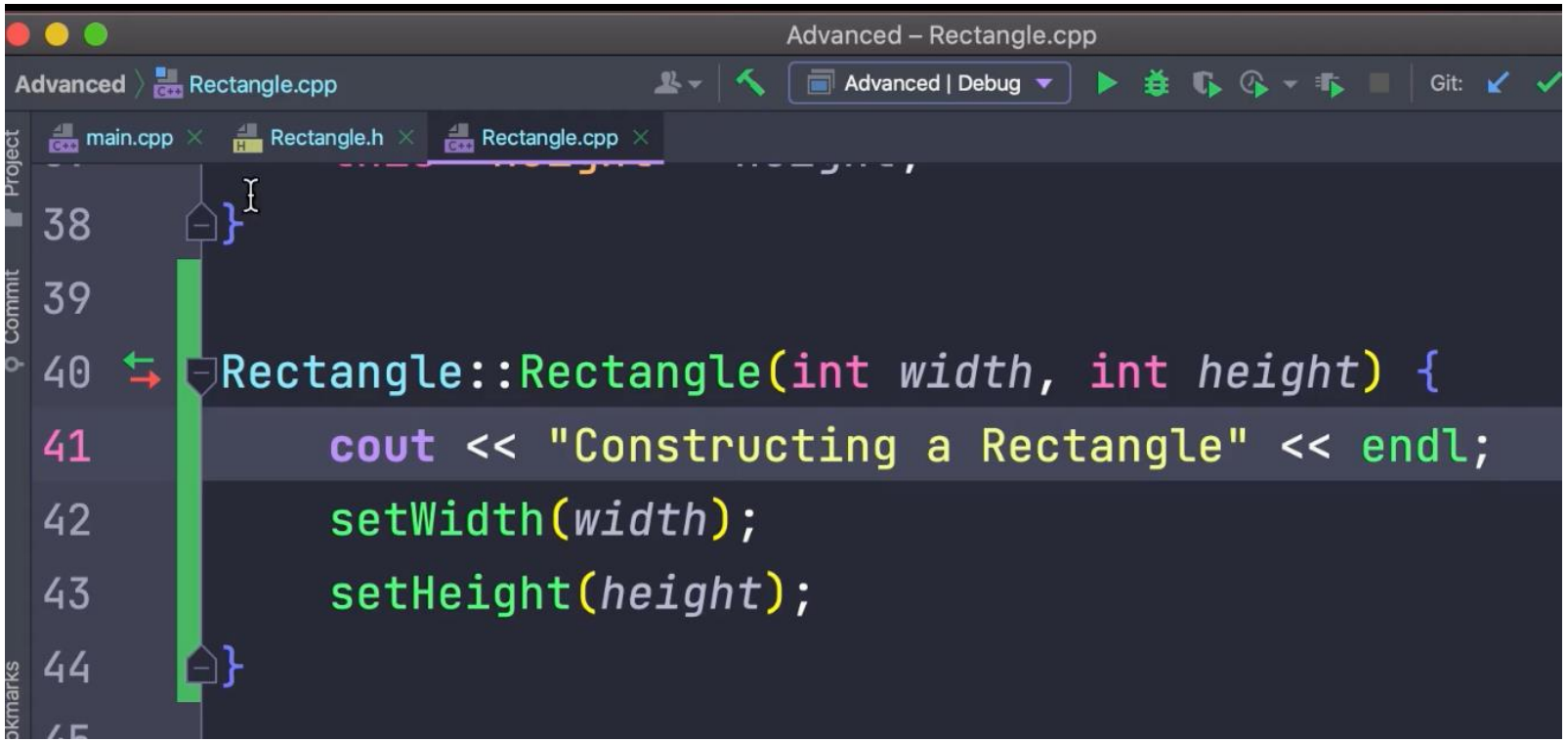

Advanced – main.cpp
Advanced
main.cpp
Advanced | Debug
main.cpp
TextBox.cpp
TextBox.h
Project
Commit
Bookmarks
1 #include <iostream>
2 #include "TextBox.h"
3
4 using namespace std;
5
6 int main() {
7 TextBox box;
8 box.setValue("Hello World");
9 cout << box.getValue();
return 0;
}

Constructors



- A special function that is used to initialize objects.
- Constructors do not have return types.

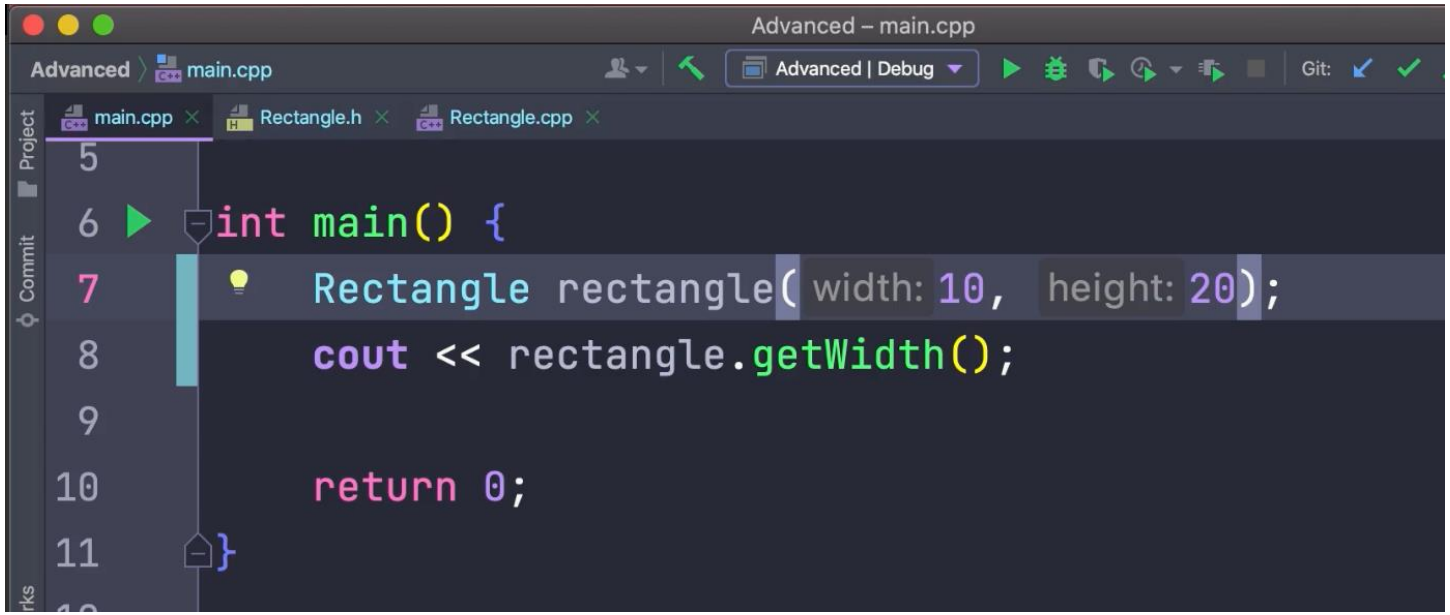
Constructors



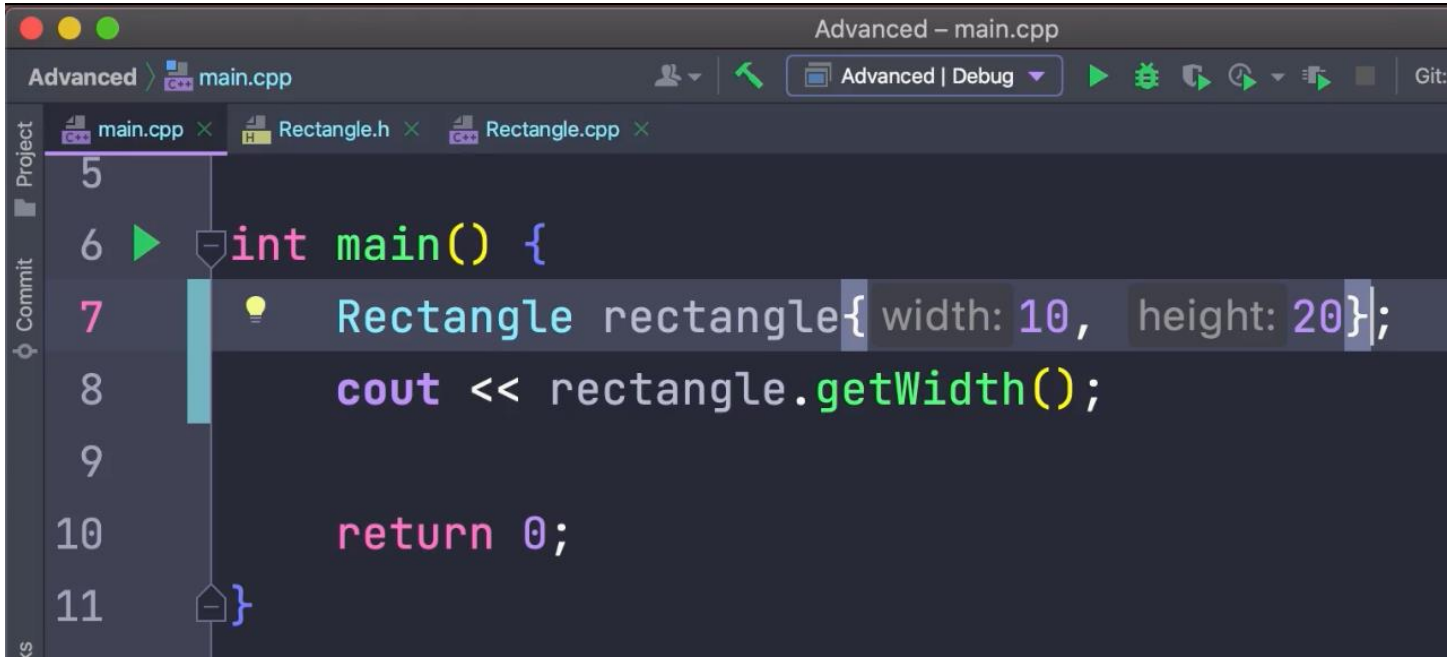
- A special function that is used to initialize objects.

Constructors

- Initializing objects.



- Option 1 - Using ( )



- Option 2 - Using { }

Member Initializer List

```
Rectangle::Rectangle(int width, int height) {
    cout << "Constructing a Rectangle" << endl;
    setWidth(width);
    setHeight(height);
}
```

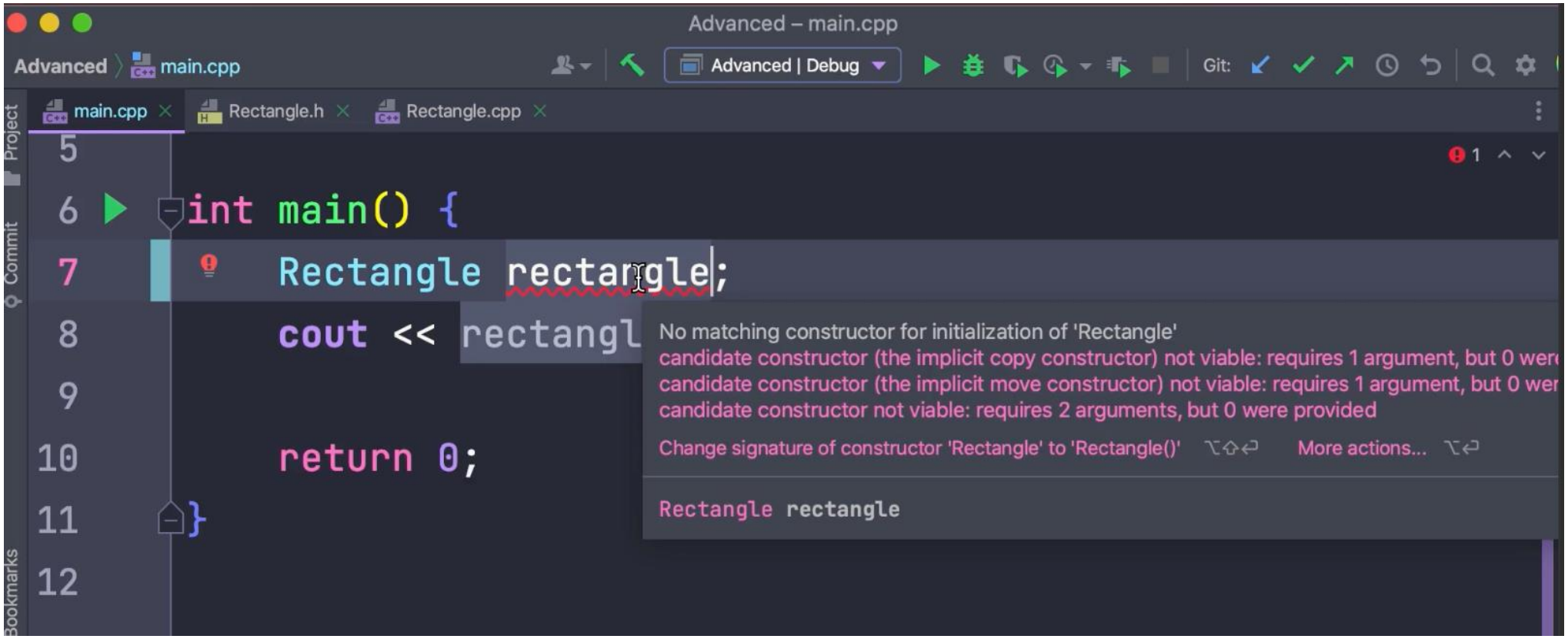
- Initialization is done in the body of the constructor.

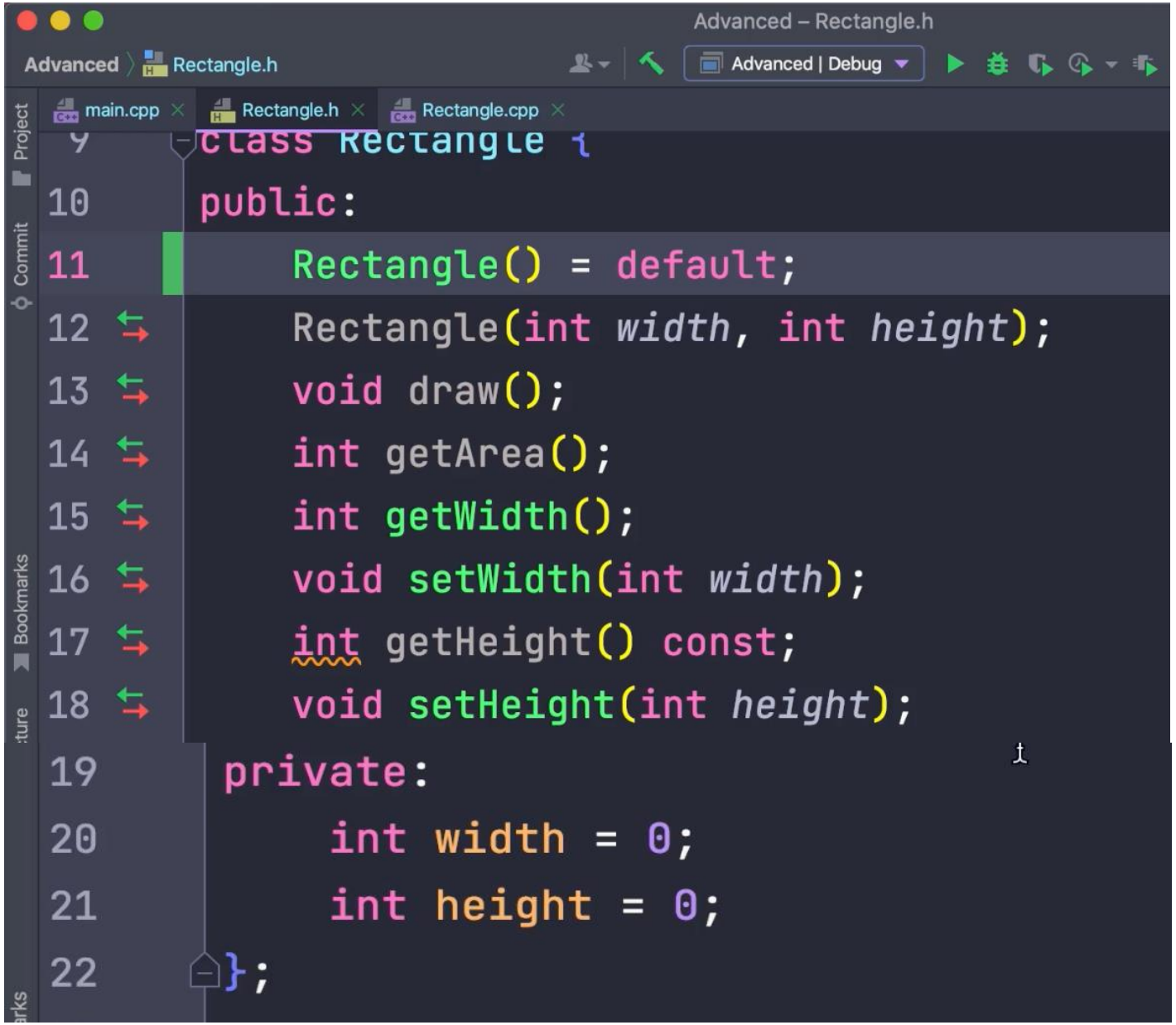
```
Rectangle::Rectangle(int width, int height) : width{width}, height{height} {
```

- Initialization is done in the definition of the class.
- The values of the members are directly assigned; no validation can be done.

The Default Constructor

- Currently, we cannot create a rectangle object without parameters width and height.



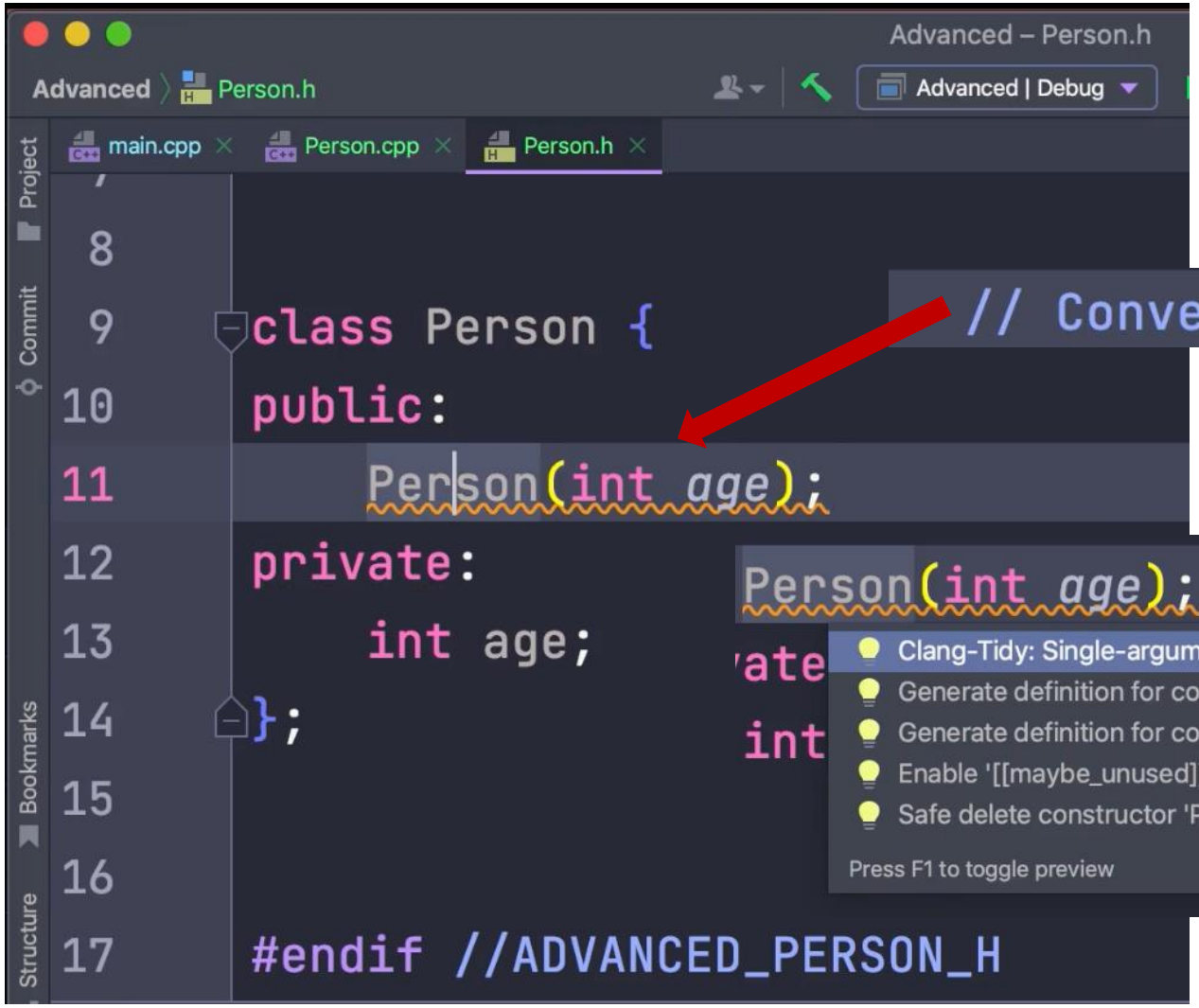


The Default Constructor

- A constructor without any parameters.
- If there is no constructor defined by the programmer, then C++ creates a default constructor automatically.
- If there is a constructor defined, then C++ does not create a default constructor. We need to create a default constructor.

Using the Explicit Keyword

- Sometimes when we declare constructors, we need to use the keyword “explicit”.




```
// Converting constructor
```

- Warning from the Clion.

```
Person(int age);
```

Clang-Tidy: Single-argument constructors must be marked explicit to avoid unintentional implicit conversions
Generate definition for constructor 'Person'
Generate definition for constructor 'Person' in-place
Enable '[[maybe_unused]]' for constructor 'Person'
Safe delete constructor 'Person'
Press F1 to toggle preview

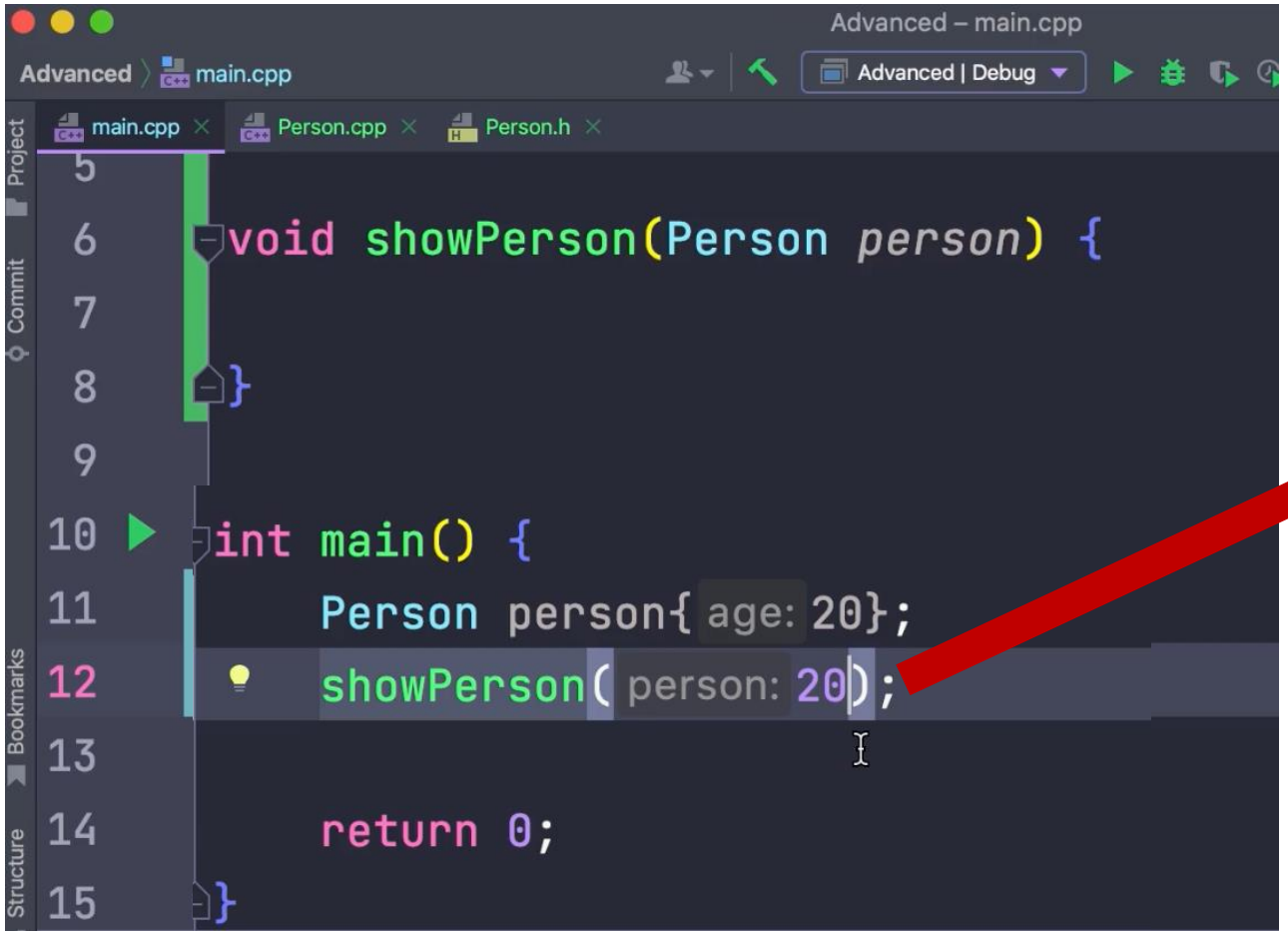
Using the Explicit Keyword

- Sometimes when we declare constructors, we need to use the keyword “explicit”.
- C++ will implicitly create a person object and pass to the function.

```
Advanced – main.cpp
Advanced | Debug
main.cpp  Person.cpp  Person.h

#include <iostream>
#include "Person.h"

using namespace std;

void showPerson(Person person) {

}

int main() {
    Person person{ age: 20};
    showPerson(person);

    return 0;
}
```

```
int main() {
    Person person{ age: 20};
    showPerson( person: 20);

    return 0;
}
```

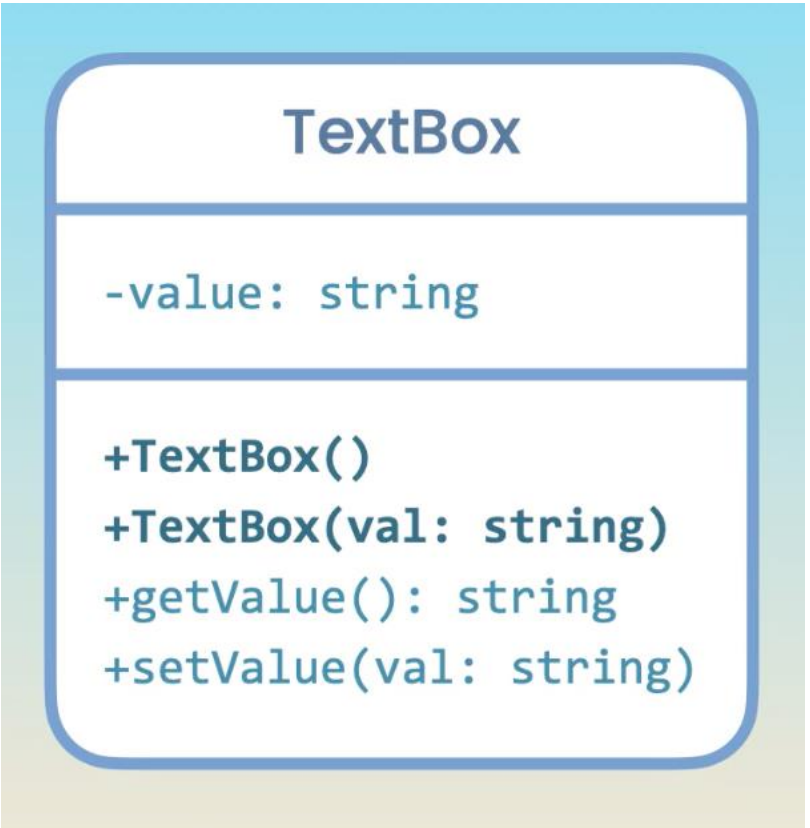
Using the Explicit Keyword



- To prevent this, we use "**explicit**" in the header.

```
class Person {
public:
    explicit Person(int age);
private:
    int age;
};
```

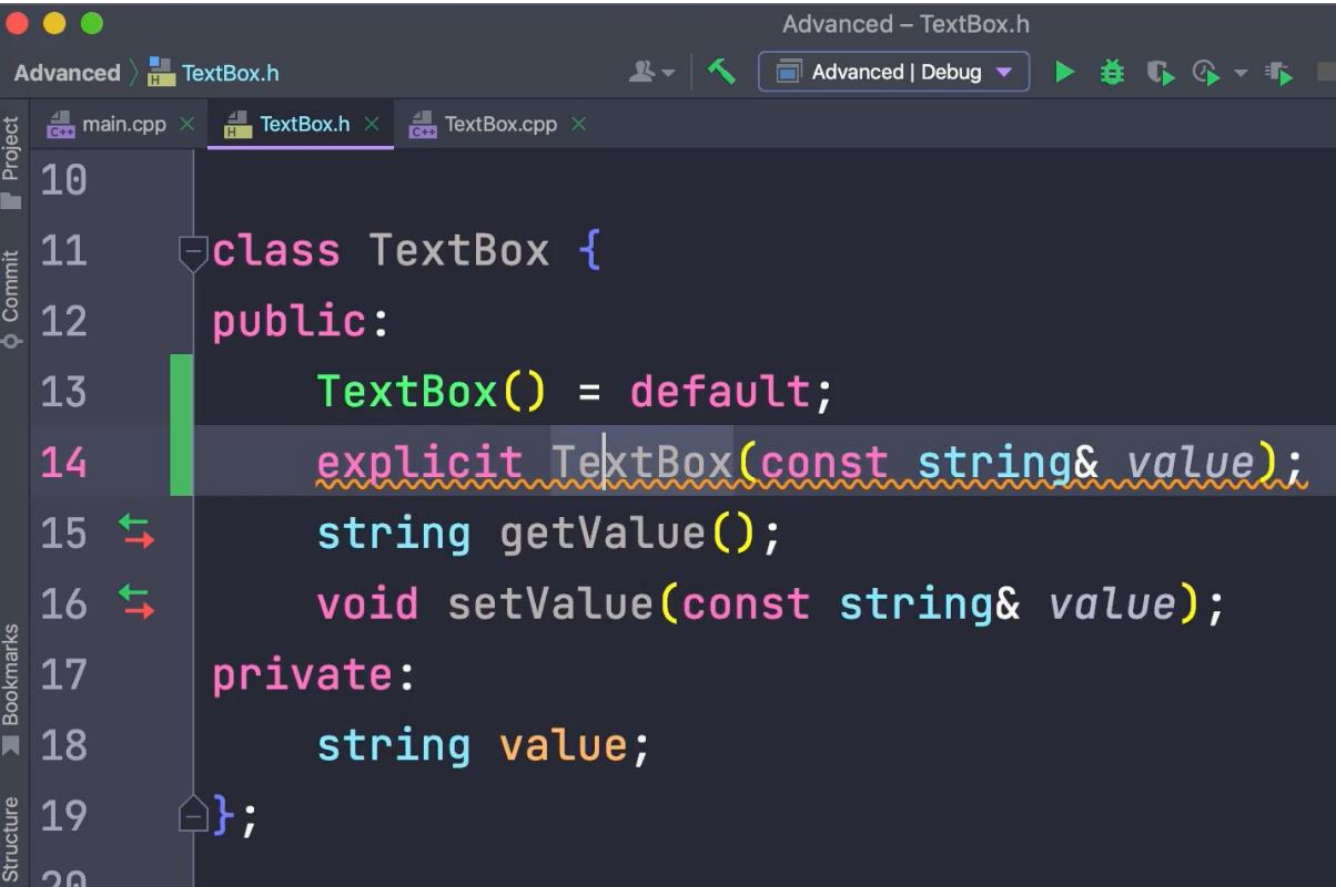
Using the Explicit Keyword

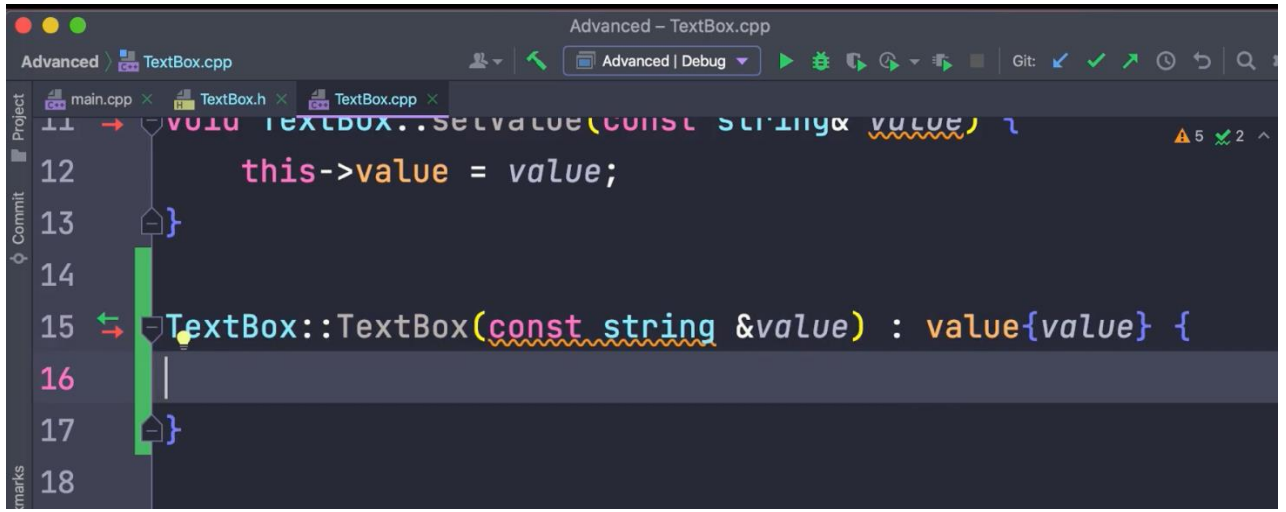


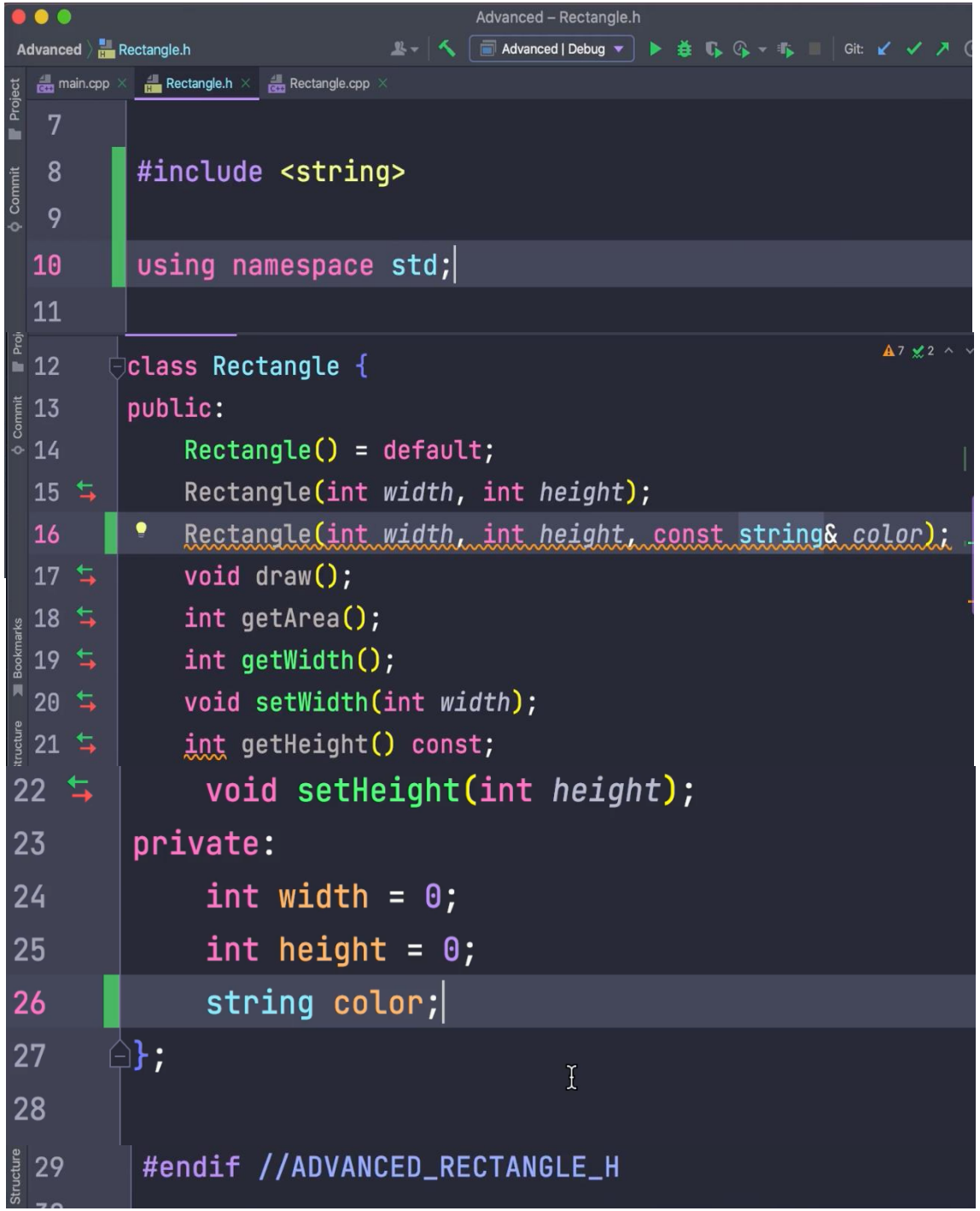
- Example

Using the Explicit Keyword

- Example







Constructor Delegation

- A constructor can delegate the initialization of an object to another constructor.
- Always pass strings as references for efficiency.
- Use const keyword to prevent the parameter from being changed.

Constructor Delegation

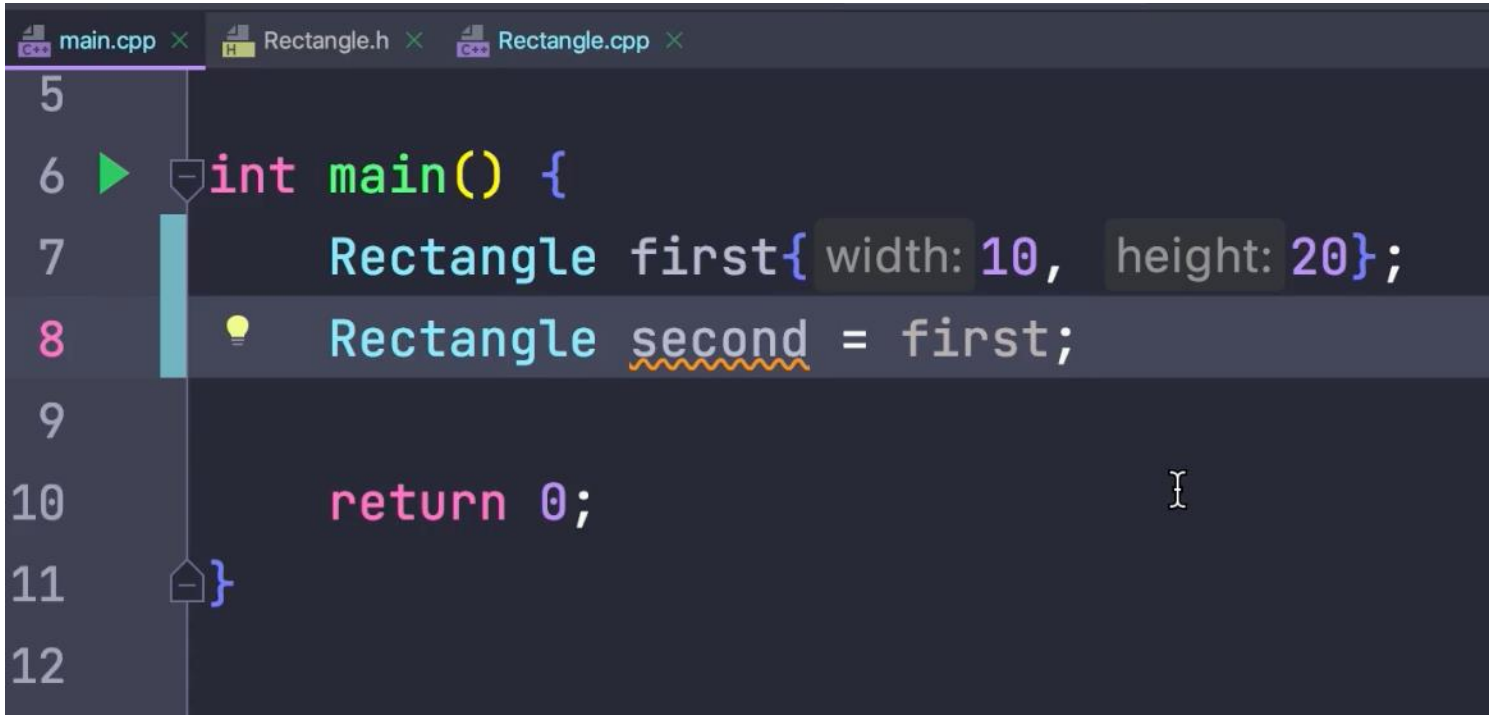
main.cpp | Rectangle.h | Rectangle.cpp

```
39
40  Rectangle::Rectangle(int width, int height) {
41      cout << "Constructing a Rectangle" << endl;
42      setWidth(width);
43      setHeight(height);
44  }
45
46  Rectangle::Rectangle(int width, int height, const string &color) : Rectangle(width, height) {
47      cout << "Constructing a Rectangle with color" << endl;
48      this->color = color;
49  }
```

main.cpp | Rectangle.h | Rectangle.cpp

```
5
6   int main() {
7       Rectangle rectangle{ width: 10, height: 20, color: "blue"};
8
9       return 0;
10  }
```

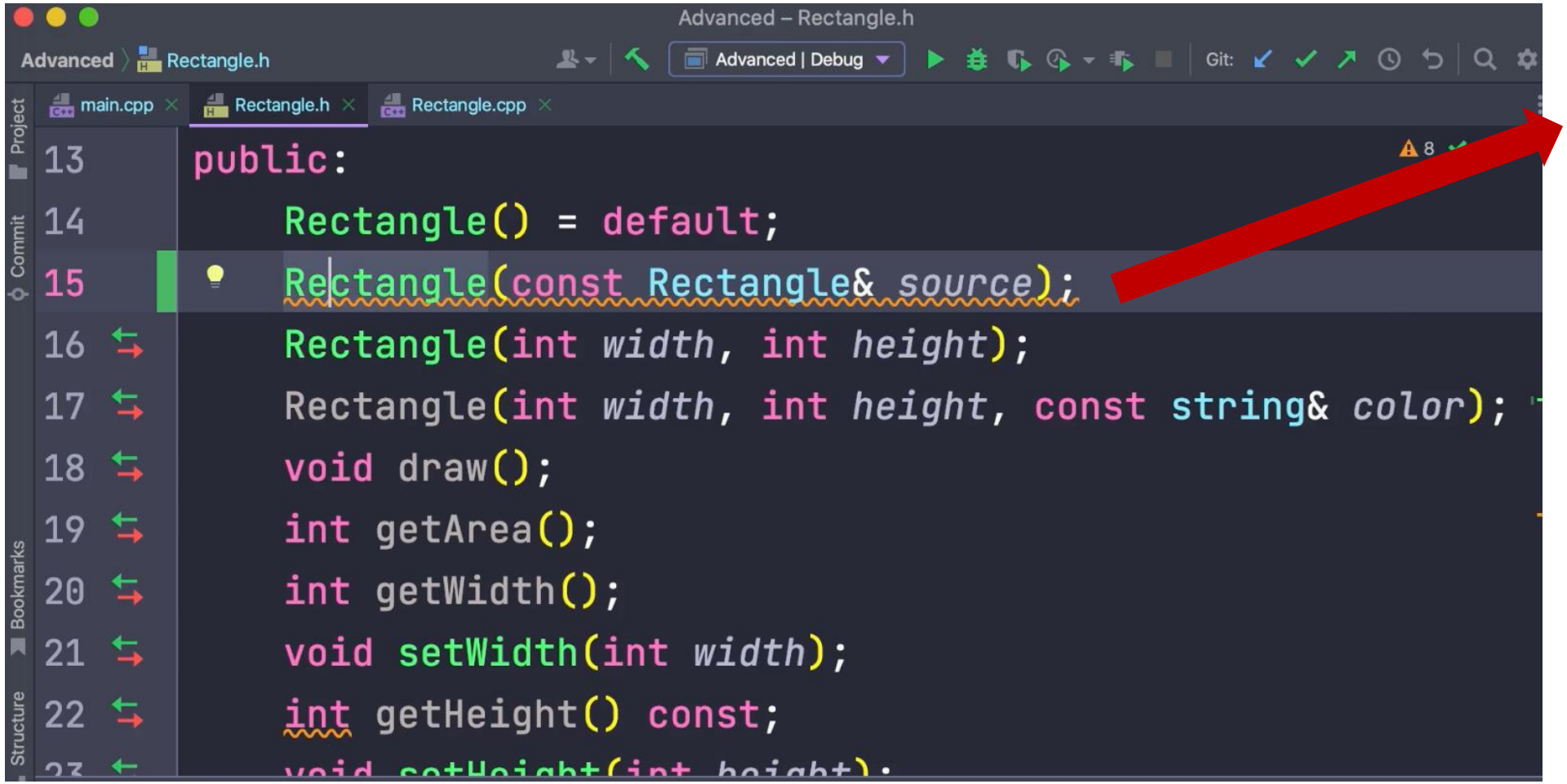
The Copy Constructor



- Used to copy objects.

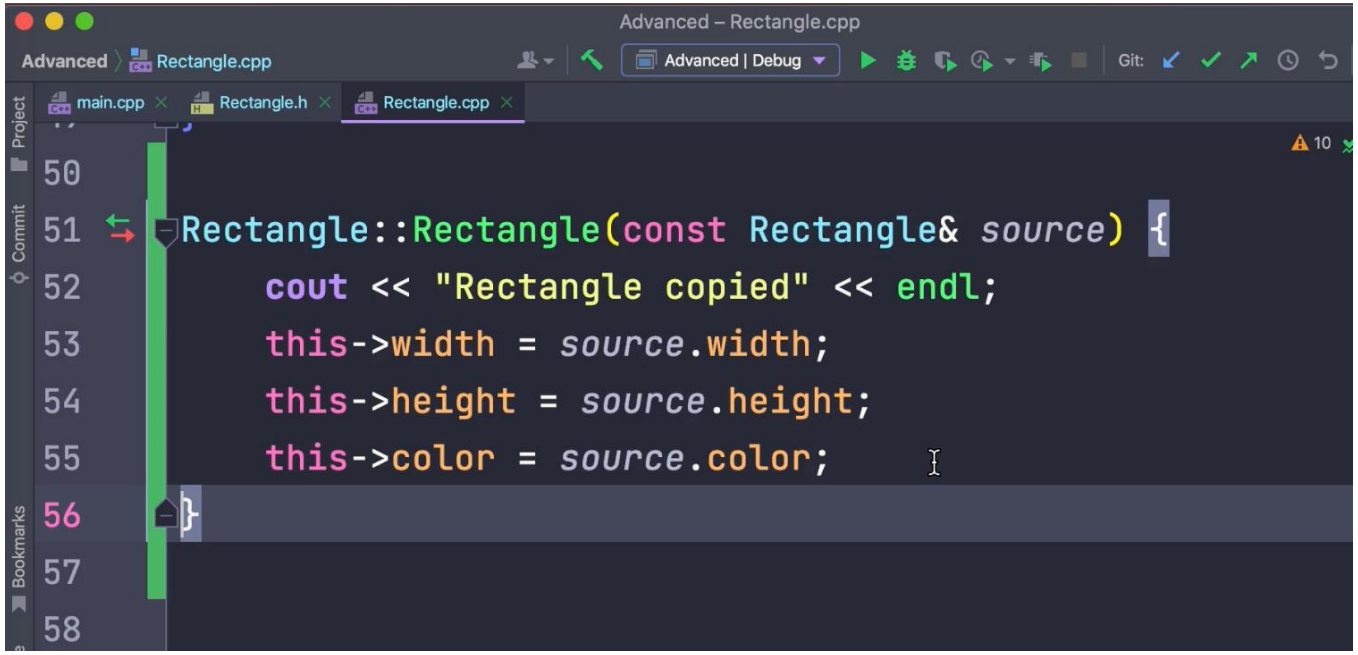
- In general, when we assign an object to another object during the object creation, the compiler generates a copy constructor and takes care of everything.
- Sometimes we need to control how objects are copied and create our own copy constructors.

The Copy Constructor

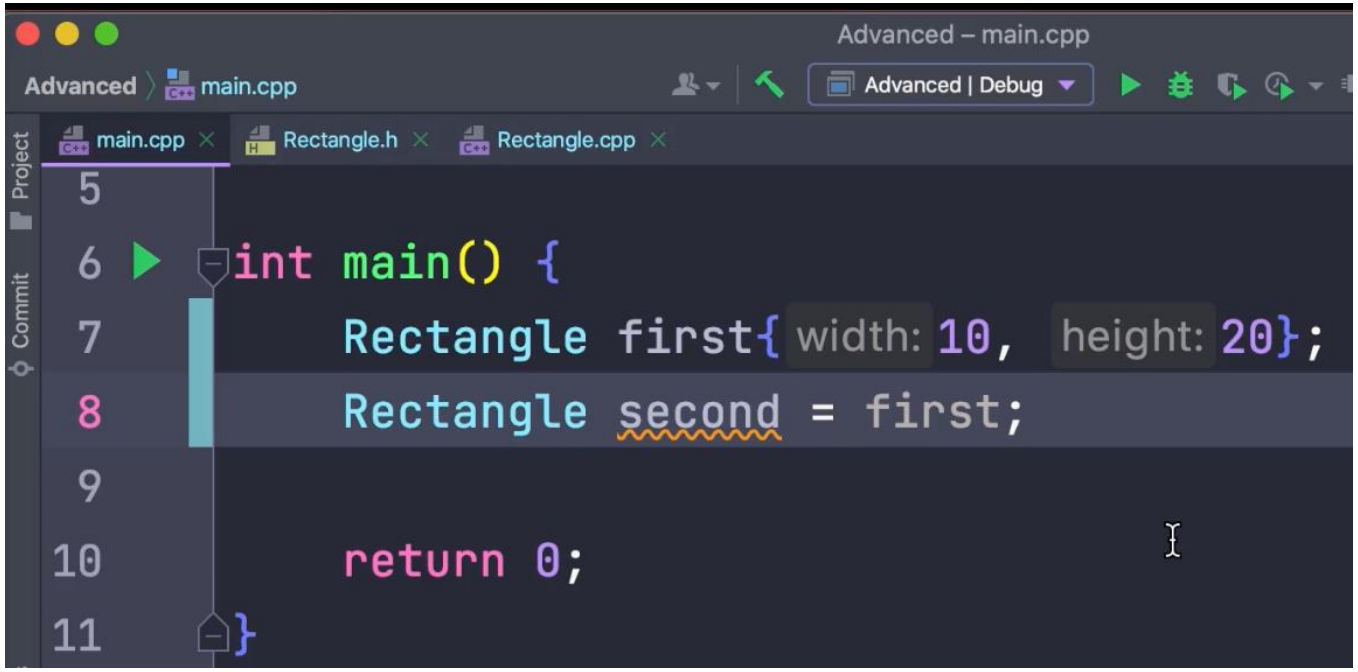


- Definition

The Copy Constructor

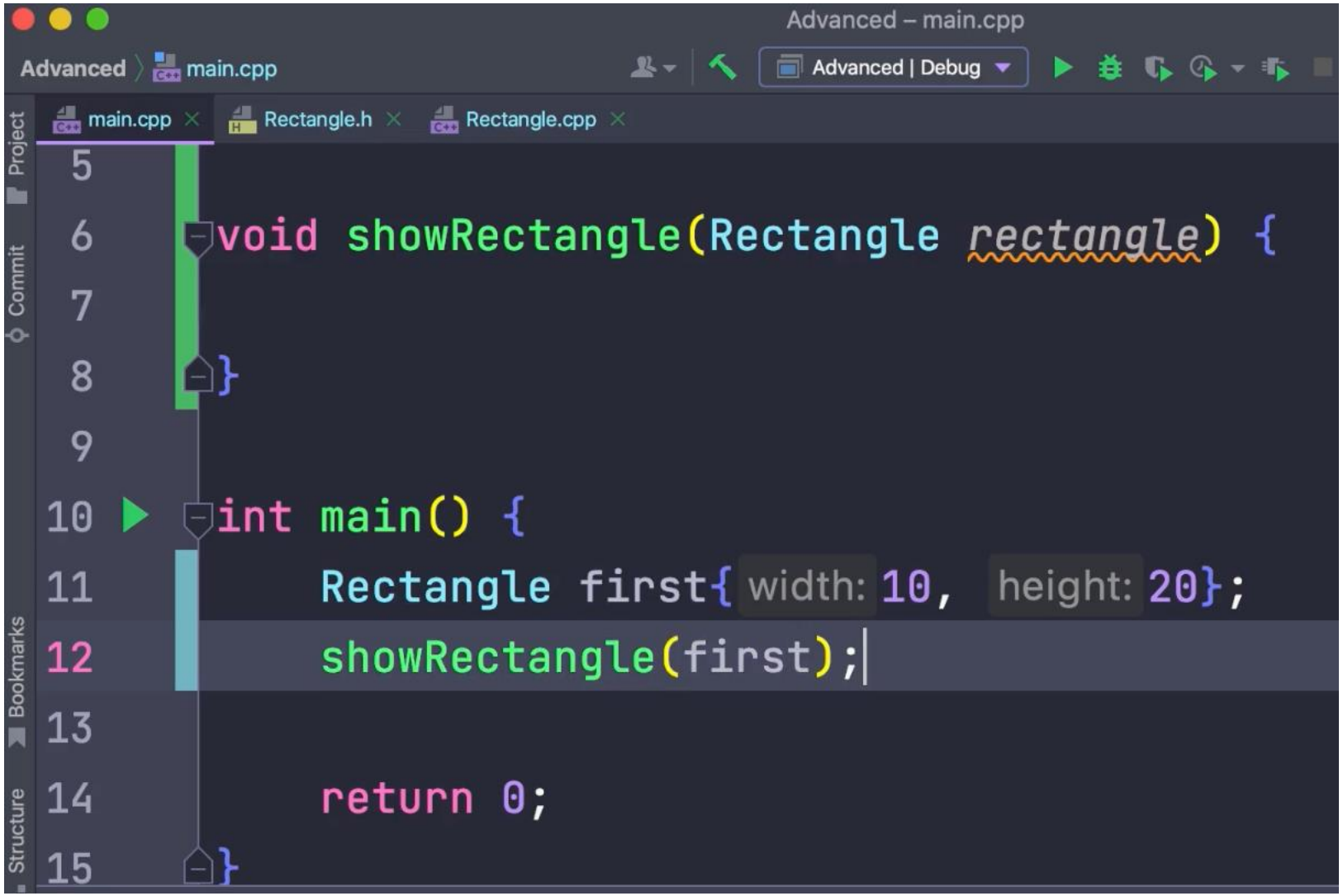


- Implementation



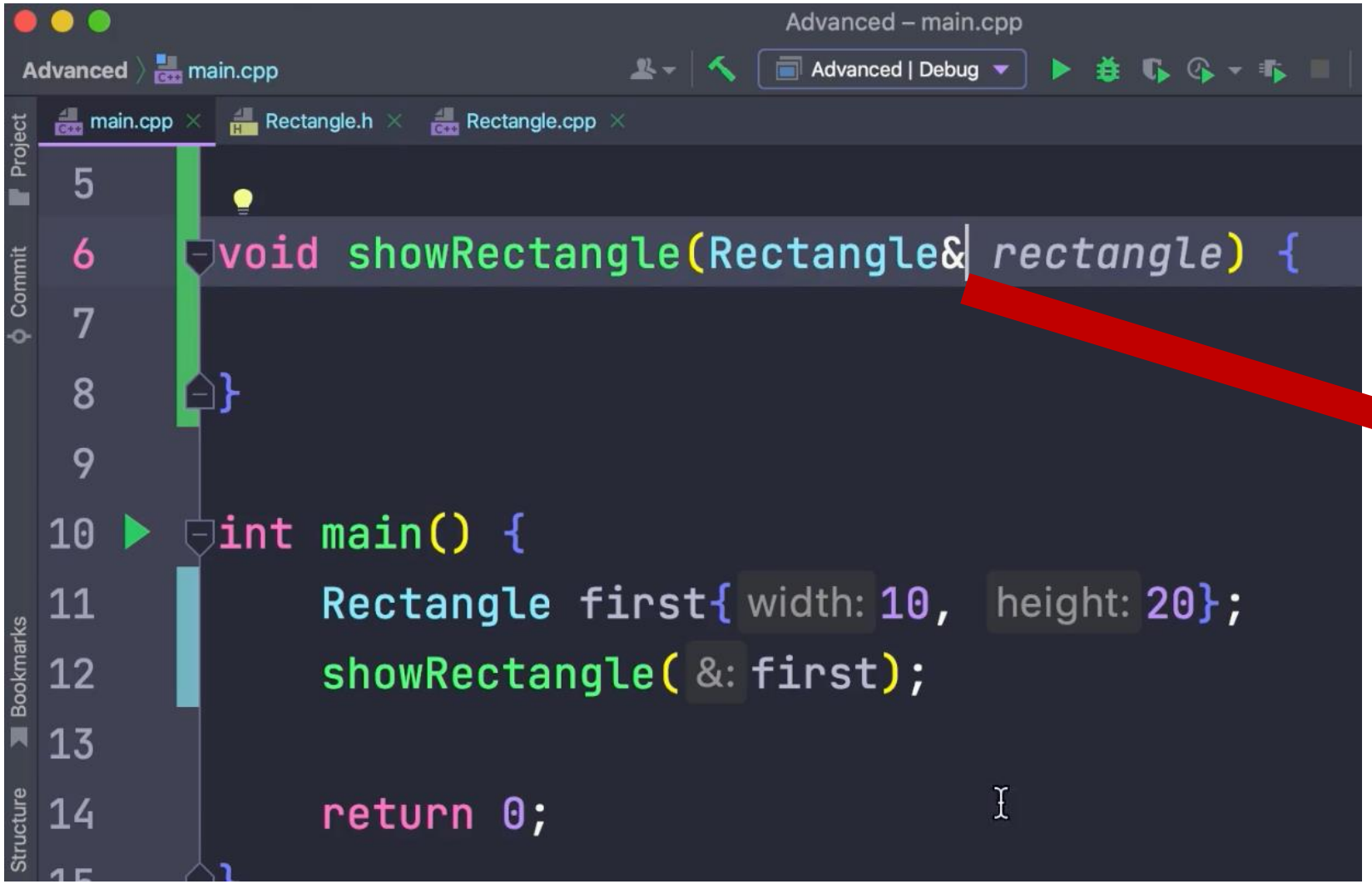
- main function

The Copy Constructor



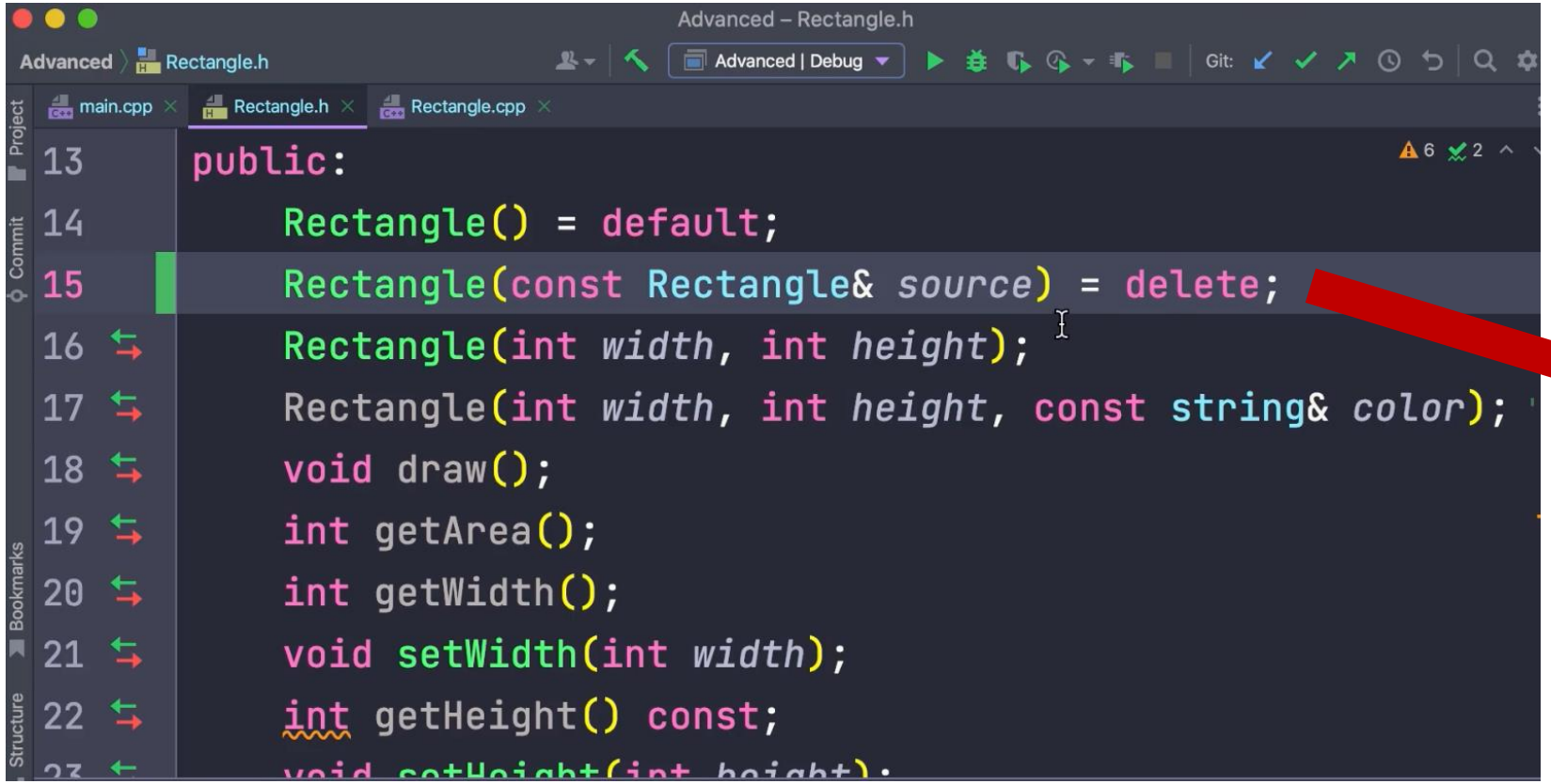
- Objects are also copied when they are passed to functions as parameters.

The Copy Constructor



- The object, first, is not copied in this case since it is passed by reference.

The Copy Constructor



- Used to prevent objects to be copied by value.

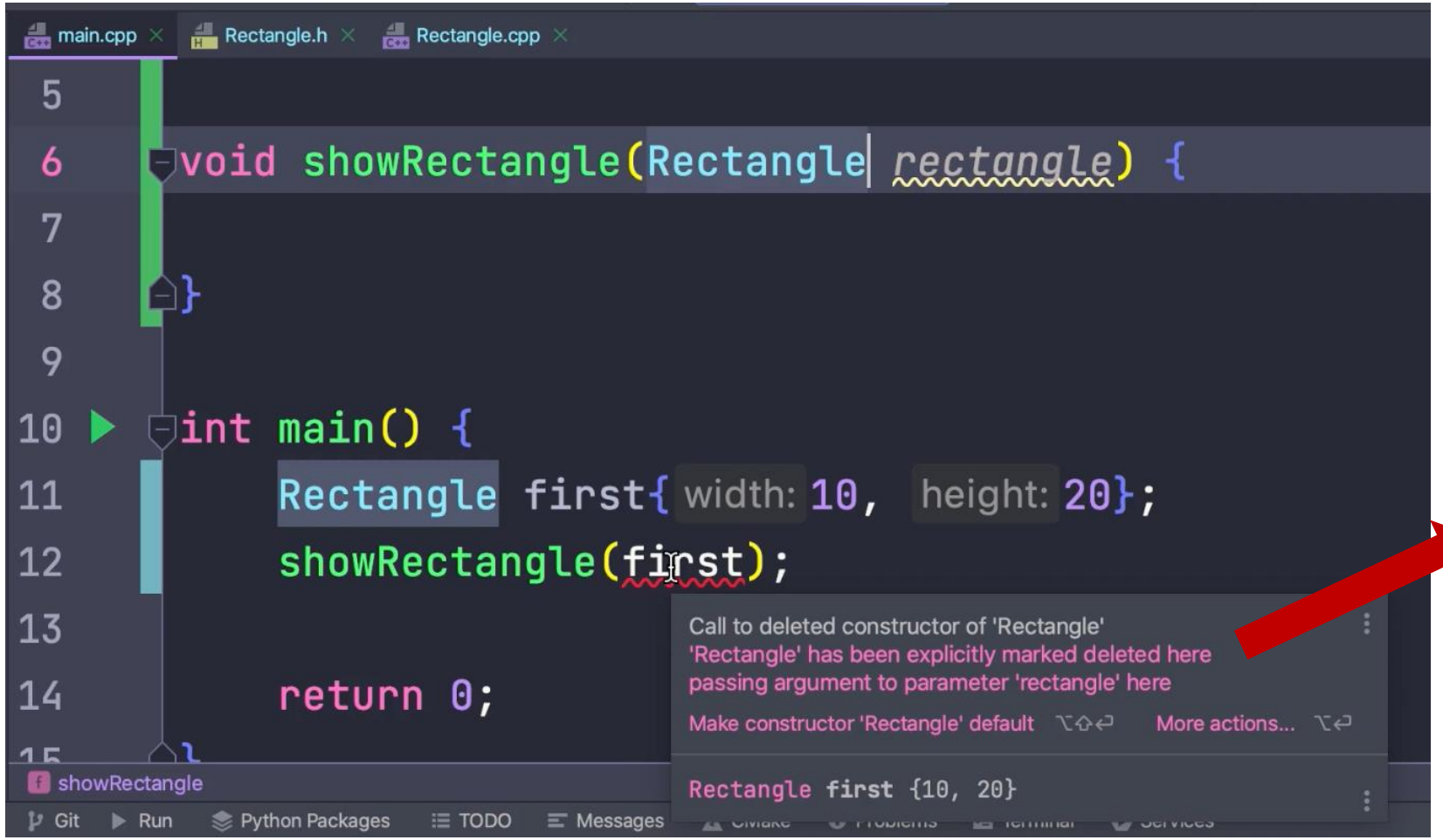
- We need to delete the implementation of the copy constructor.

The Copy Constructor

- Used to prevent objects to be copied by value.

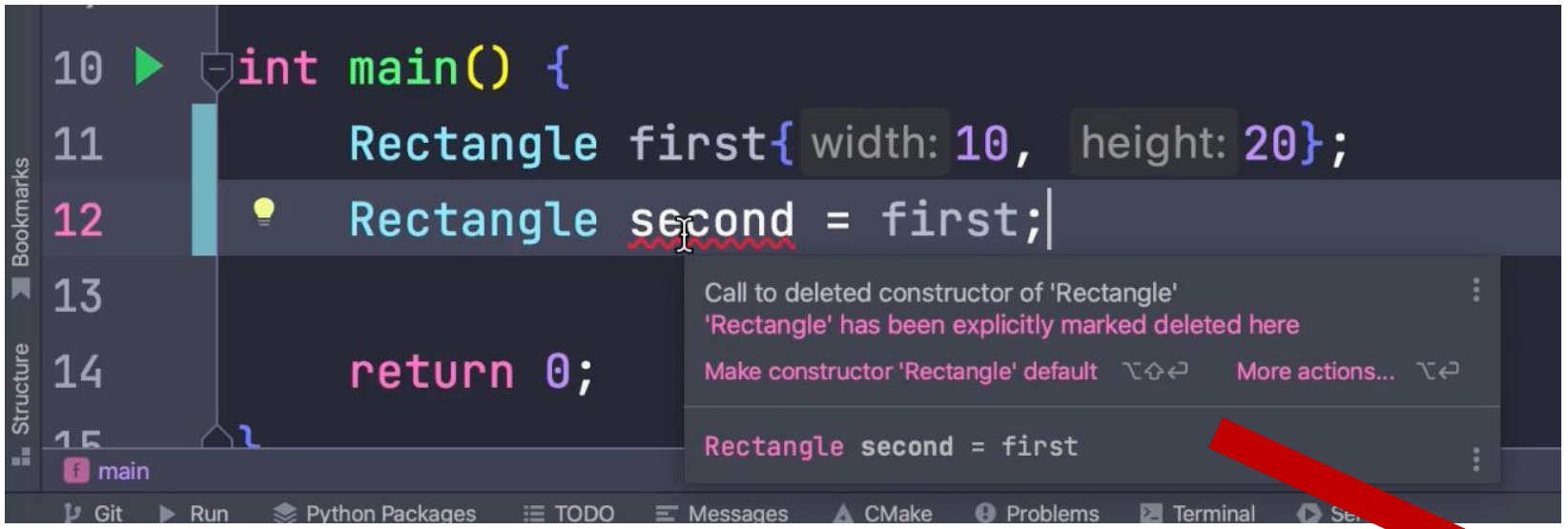
- We need to delete the implementation of the copy constructor.

The Copy Constructor



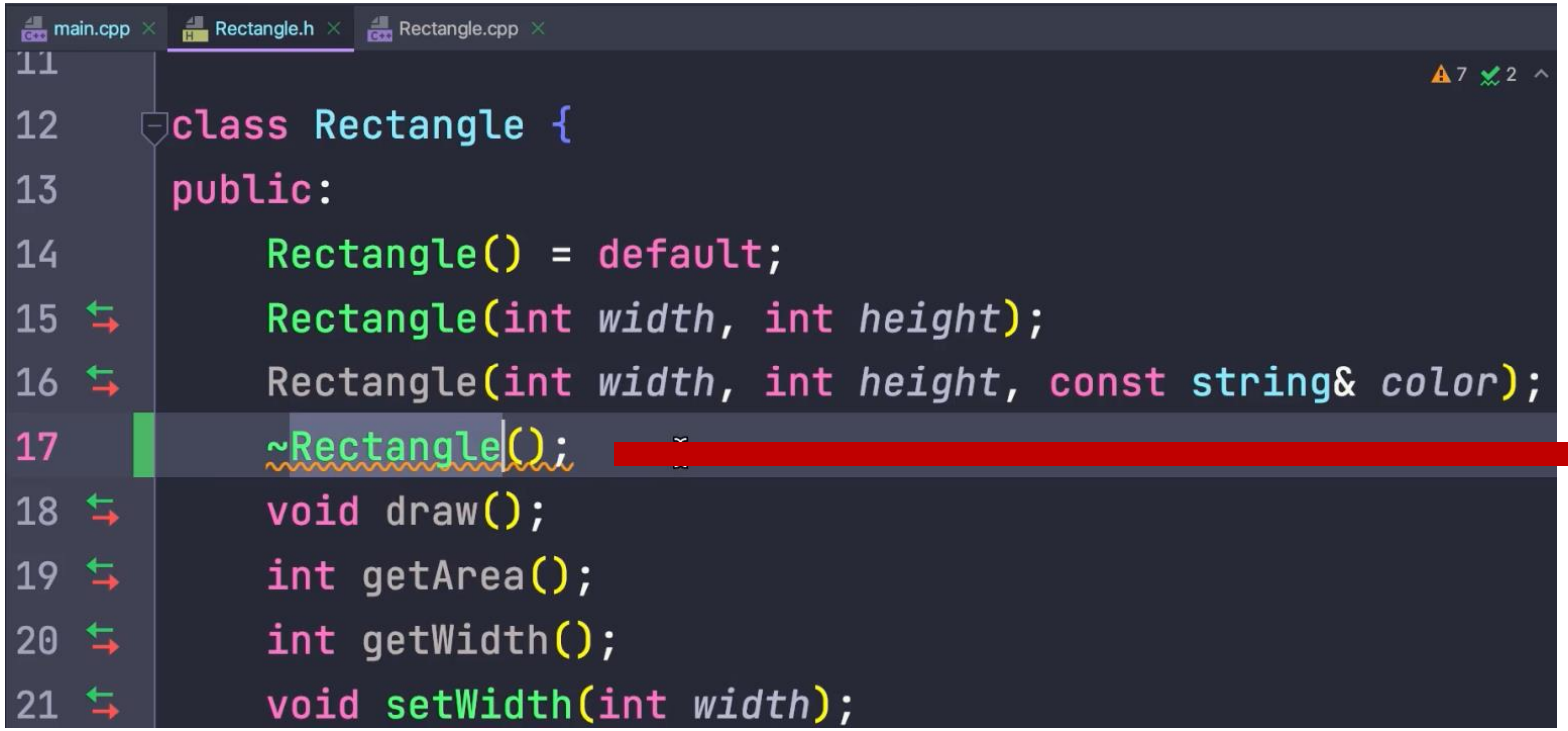
- Error message when we try to pass the object by value to the function because the copy constructor is deleted.

The Copy Constructor



- Error message when we try to copy and object because the copy constructor is deleted.

The Destructor



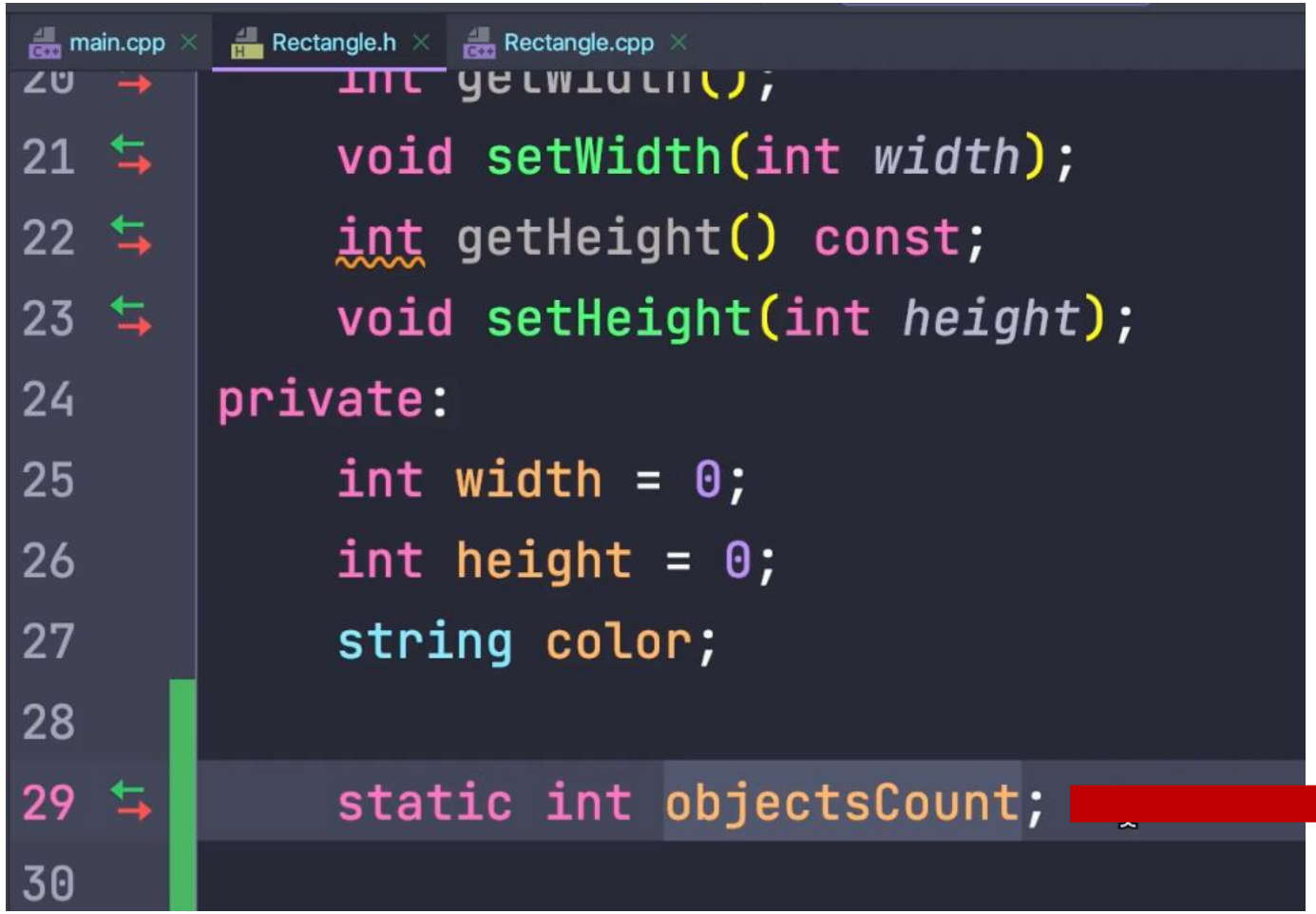
- Automatically called when an object is being destroyed.
- We need to free the resources used by the objects, like memory and files.
- Similar to constructors, destructors do not have a return type.
- There can be only one destructor.

The Destructor

- Implementation

- When the main function ends, the object, first, gets out of scope and is destroyed.

Static Members



- Members that belongs to the class and shared by all instance objects.
- All instances of the class share the static members.

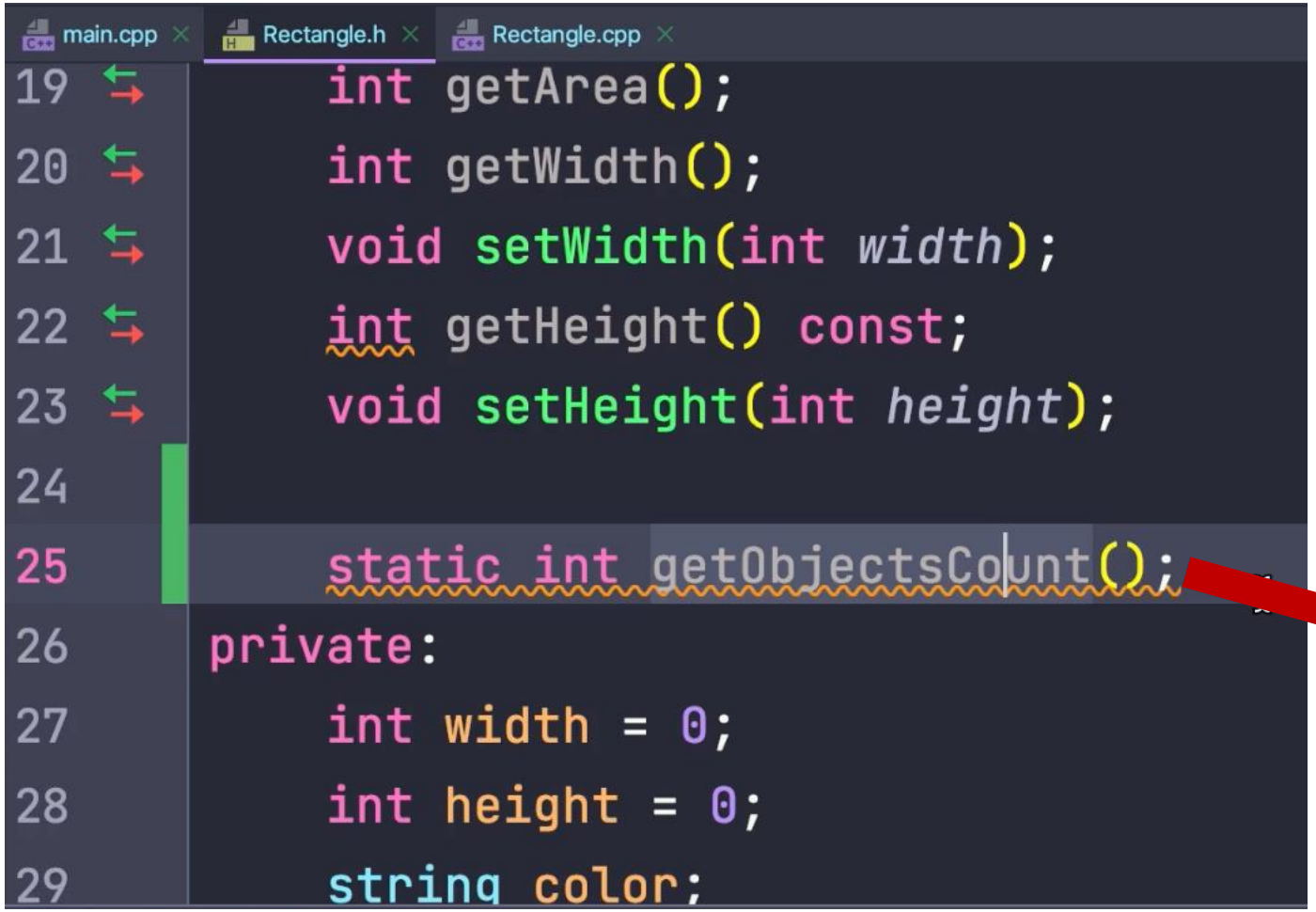
- Static member declaration in the class header file.

Static Members

- Static members should be defined in the main implementation file.

- Static member definition in the class implementation file.

Static Members



- Members that belongs to the class with one copy in memory and shared by all instance objects.

- Static member function definition in the class header file.

Static Members

main.cpp | Rectangle.h | Rectangle.cpp

```
39
40  Rectangle::Rectangle(int width, int height) {
41      objectsCount++;
42      cout << "Constructing a Rectangle" << endl;
43      setWidth(width);
44      setHeight(height);
45  }
46
```

- Update constructor to increase the static member, objectsCount, every time an object is created.

Static Members

```
main.cpp  Rectangle.h  Rectangle.cpp
54  }
55
56  int Rectangle::objectsCount = 0;
57
58  int Rectangle::getObjectsCount() {
59      return objectsCount;
60  }
```

- Static member functions can only access static members.

- Static member function implementation.

```
5
6   int main() {
7       Rectangle first{ width: 10,  height: 20};
8       Rectangle second{ width: 10,  height: 20};
9       cout << Rectangle::getObjectsCount() << endl;
10
11      return 0;
12  }
13
```

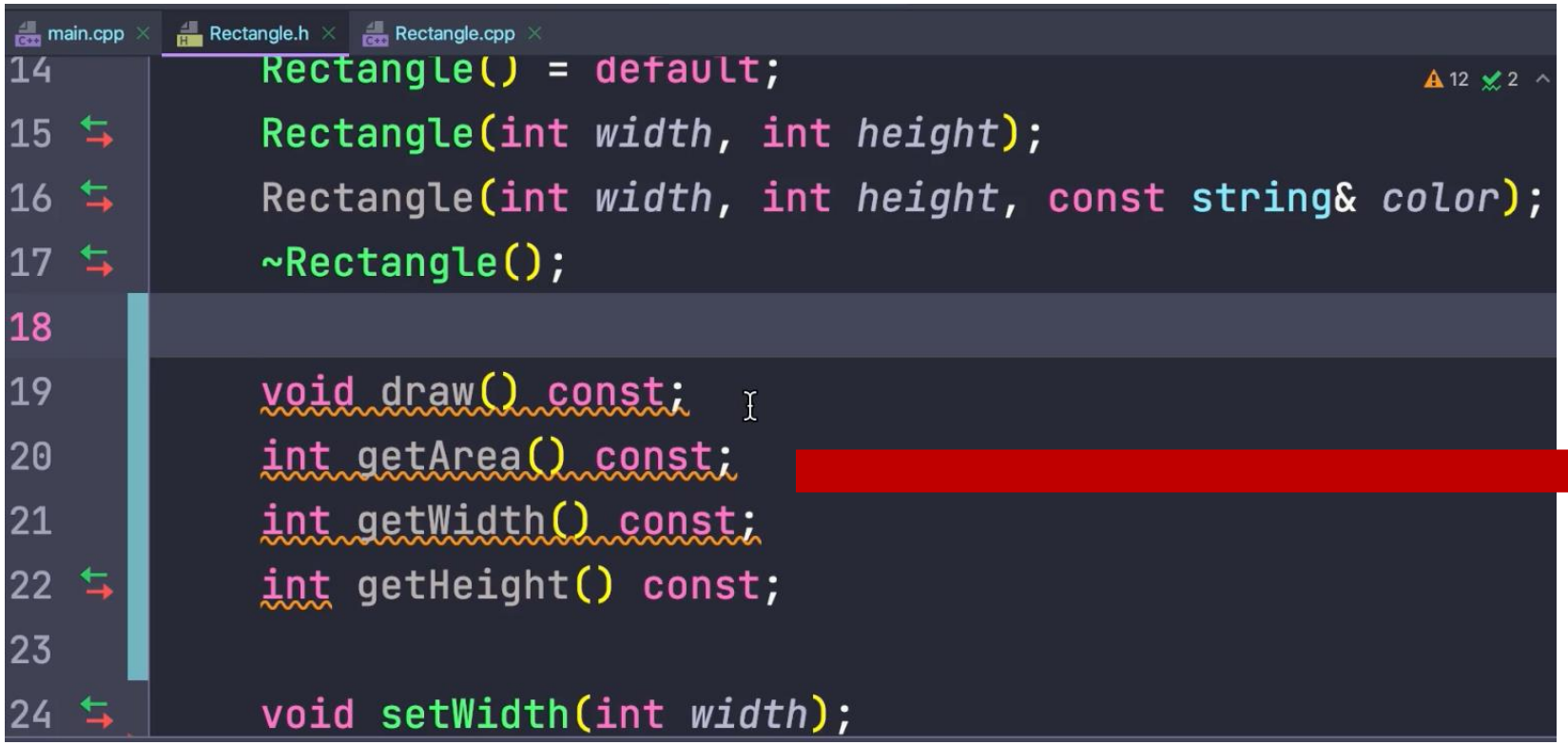
Scope resolution operator

- The main function.

Constant Objects and Functions

- When we make an object constant, all its attributes become constant.
- Only constant functions can be called.
- Constant functions do not change the status of an object.

Constant Objects and Functions



- Functions that do not change the status of an object should be declared as constant.

Constant Objects and Functions

```
9
10  void Rectangle::draw() const {
11      cout << "Drawing a rectangle!" << endl;
12      cout << "Dimensions: " << width << ", " << height << end
13  }
14
15  int Rectangle::getArea() const {
16      return width * height;
17  }
```

```
19  int Rectangle::getWidth() const {
20      return width;
21  }
```

```
29  int Rectangle::getHeight() const {
30      return height;
31  }
```

- Functions that do not change the status of an object should be defined as constant.

Constant Objects and Functions

- All accessible constant functions in the main file.

Constant Objects and Functions

```
main.cpp  Rectangle.h  Rectangle.cpp
5
6  int main() {
7      // Immutable
8      const Rectangle rectangle;
9      rectangle.getObjectsCount()
10
11     return 0;
12 }
```

- Even though we can call the static function from an object, it is not the right way.
- It is not tied to a particular object; it is tied to the class.

```
main.cpp  Rectangle.h  Rectangle.cpp
5
6  int main() {
7      // Immutable
8      const Rectangle rectangle;
9      Rectangle::getObjectsCount()
10
11     return 0;
12 }
```

- Correct way of calling the static function.

Pointer to Objects

```
int main() {
    Rectangle rectangle;

    return 0;
}
```

- The object, rectangle, in this example is created in stack.
- Once the execution of main() functions is completed, it gets out of scope and deleted automatically.

```
int main() {
    auto* rectangle = new Rectangle( width: 10,  height: 20);
    rectangle->draw();
    delete rectangle;
    rectangle = nullptr;

    return 0;
}
```

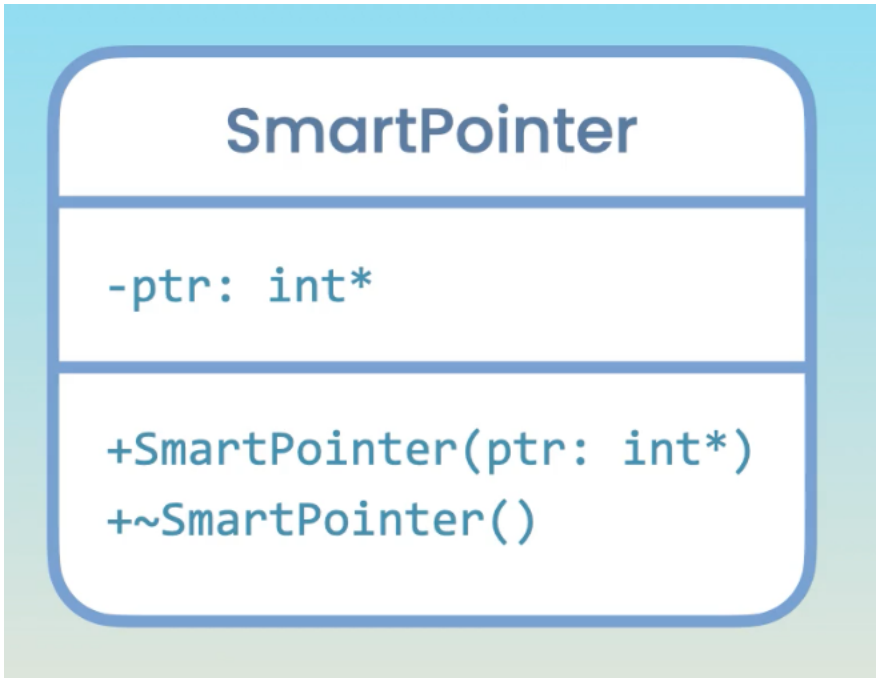
- When we use new to create an object, it is created in the heap memory.
- We are responsible releasing the memory and setting the pointer to null once we are done with the object.
- **Dangling pointer:** A pointer that is not set to null after its use and pointing some random memory address.
- If we reset the pointer but do not free up the memory, we will have a **memory leak**.

Pointer to Objects

```cpp
#include "Rectangle.h"
#include <iostream>
#include <memory>

using namespace std;

int main() {
    auto rectangle : unique_ptr<Rectangle> = make_unique<Rectangle>( width: 10,  height: 20);
    rectangle->draw();

    return 0;
}
```

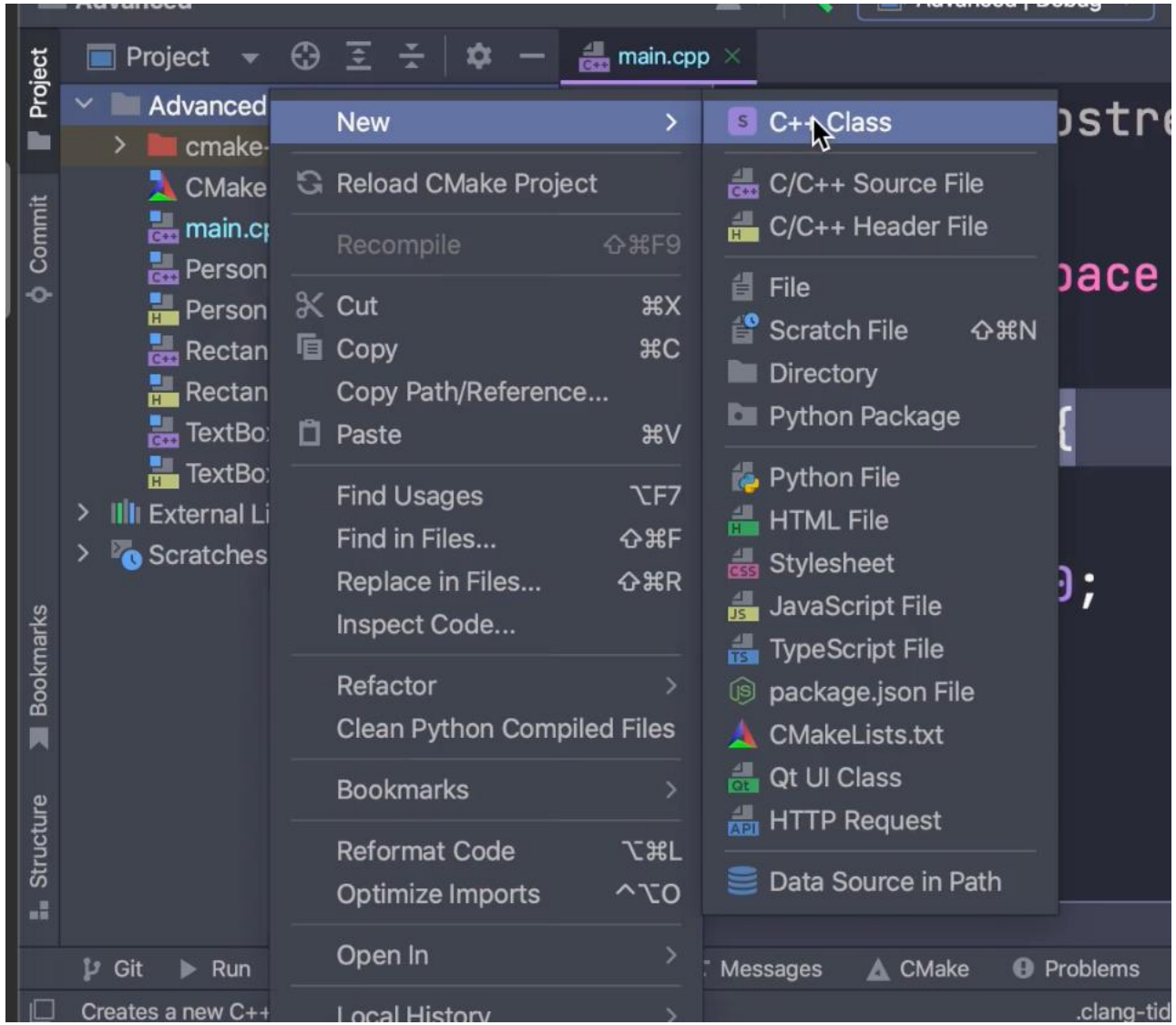
- **Dangling pointer:** A pointer that is not set to null after its use and pointing some random memory address.
- If we reset the pointer but do not free up the memory, we will have a **memory leak**.
- Not to deal with dangling pointers and memory leak, in modern C++, we use **smart pointers**.
- Everything is taken care of automatically.
- **Unique pointer:** Pointer owns the memory it points to; another pointer cannot point to this location.
- **Shared pointer:** Memory address can be shared by multiple pointers.

Pointer to Objects



- **Exercise:** Create the class SmartPointer described in the UML diagram.

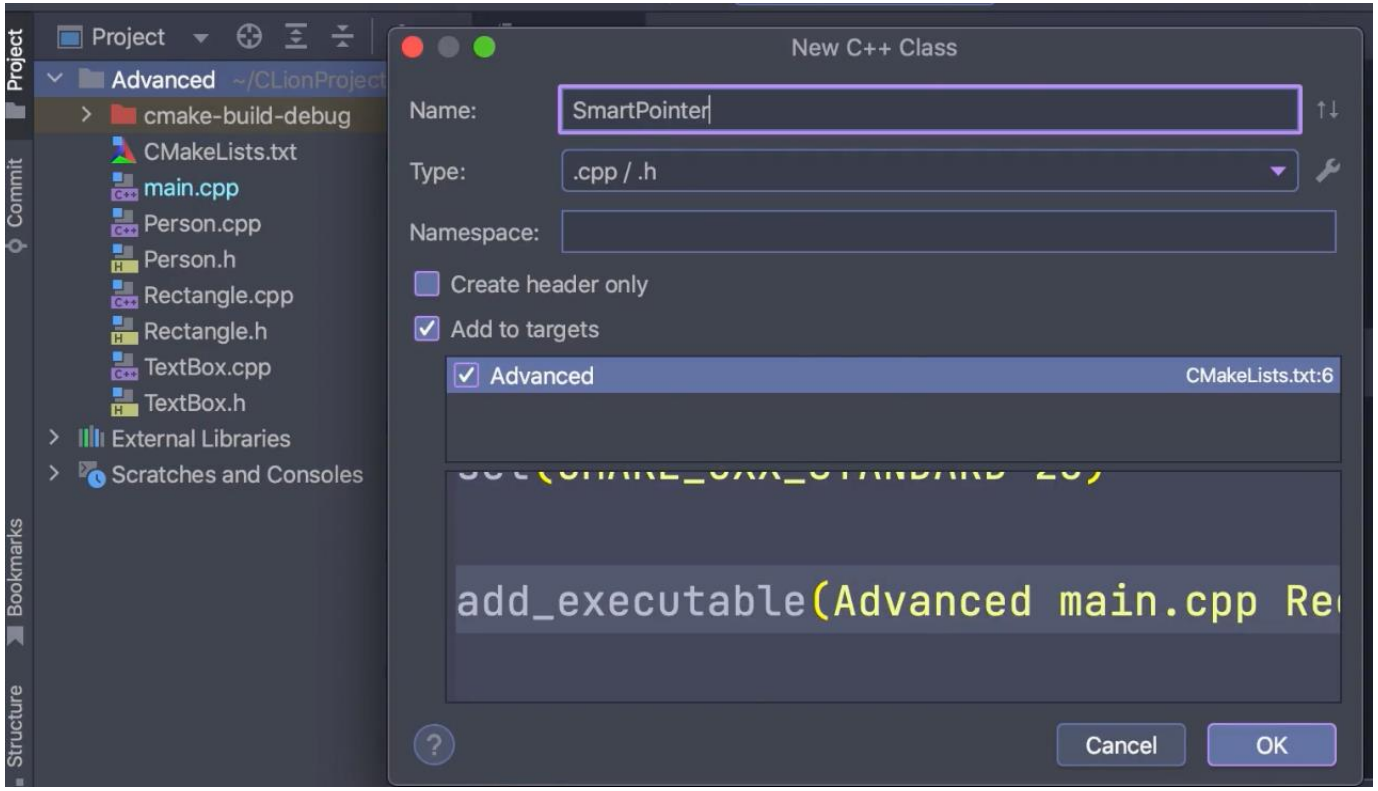
Pointer to Objects



- **Exercise:** Create the class SmartPointer described in the UML diagram.

- Select **New/C+ Class** to create new class.

Pointer to Objects



- **Exercise:** Create the class SmartPointer described in the UML diagram.

- Name the class.

Pointer to Objects

main.cpp | SmartPointer.cpp | SmartPointer.h

```
8
9   class SmartPointer {
10  public:
11      explicit SmartPointer(int* ptr);
12      ~SmartPointer();
13  private:
14      int* ptr;
15  };
```

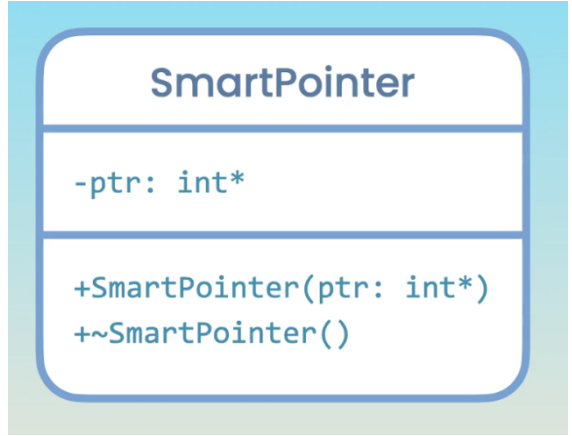
- Declaration

main.cpp | SmartPointer.cpp | SmartPointer.h

```
4
5   #include "SmartPointer.h"
6
7   SmartPointer::SmartPointer(int *ptr) : ptr{ptr} {
8
9   }
11  SmartPointer::~SmartPointer() {
12      delete ptr;
13      ptr = nullptr;
14  }
```

- Definition

- **Exercise:** Create the class SmartPointer described in the UML diagram.



Pointer to Objects

Replace

```
int* ptr = new int;
```

- **Exercise:** Create the class SmartPointer described in the UML diagram.

- Main function.

Array of Objects

- We need to have a default constructor to use this method.

Rectangle.h | Rectangle.cpp

```
class Rectangle {
public:
    Rectangle() = default;
    Rectangle(int width, int height);
    Rectangle(int width, int height, const string& color);
    ~Rectangle();

    void draw() const;
    int getArea() const;
    int getWidth() const;
```

```
int main() {
    Rectangle rectangles[3];

    return 0;
}
```

Array of Objects

- We need to have a default constructor to use this method.

Rectangle.h | Rectangle.cpp

```
class Rectangle {
public:
    Rectangle() = default;
    Rectangle(int width, int height);
    Rectangle(int width, int height, const string& color);
    ~Rectangle();

    void draw() const;
    int getArea() const;
    int getWidth() const;
```

```
int main() {
    Rectangle rectangles[3];

    return 0;
}
```

```
int main() {
    Rectangle rectangles[] = {
        [0]: Rectangle(),
        [1]: Rectangle( width: 10,  height: 20),
        [2]: Rectangle( width: 10,  height: 20,  color: "blue")
    };

    return 0;
}
```

Array of Objects

```
int main() {
    Rectangle rectangles[] = {
        [0]: Rectangle(),
        [1]: Rectangle( width: 10,  height: 20),
        [2]: Rectangle( width: 10,  height: 20,  color: "blue")
    };

    return 0;
}
```

Equivalent

```
int main() {
    Rectangle rectangles[] = {
        [0]: {},
        [1]: {10, 20},
        [2]: {10, 20, "blue"}
    };

    return 0;
}
```

- We need to have a default constructor to use this method.

- The compiler knows that each object in the array is a rectangle object.
- It calls the corresponding constructor and passes the values to create the objects.

Array of Objects

- We need to have a default constructor to use this method.

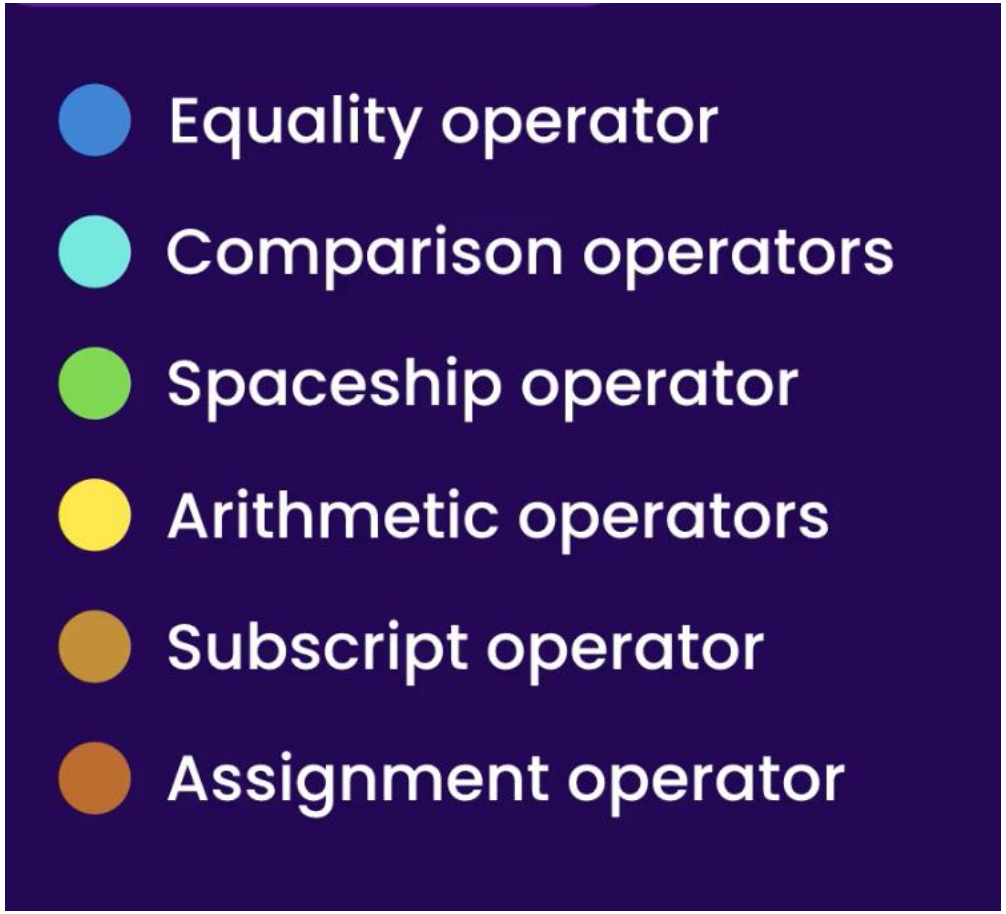
```
int main() {
    Rectangle rectangles[] = {
        [0]: Rectangle(),
        [1]: Rectangle( width: 10,  height: 20),
        [2]: Rectangle( width: 10,  height: 20,  color: "blue")
    };

    return 0;
}
```

Equivalent

```
    Rectangle rectangles[] = {
        [0]: {},
        [1]: {10, 20},
        [2]: {10, 20, "blue"}
    };
    rectangles[0].draw()

    return 0;
}
```

```
    Rectangle rectangles[] = {
        [0]: {},
        [1]: {10, 20},
        [2]: {10, 20, "blue"}
    };

    for (Rectangle& rect: rectangles)
        rect.draw();

    return 0;
}
```

Operator Overloading



- The goal is to implement built in operators for our classes.

Overloading the Equality Operator

Length.cpp | Length.h | main.cpp

```
8
9   class Length {
10  public:
11      explicit Length(int value);
12      bool operator==(const Length& other) const;
13  private:
14      int value;
15  };
```

Length.cpp | Length.h | main.cpp

```
6
7   Length::Length(int value) : value(value) {}
8
9   bool Length::operator==(const Length &other) const {
10      return value == other.value;
11  }
```

Length.cpp | Length.h | main.cpp

```
5
6   int main() {
7       Length first{ value: 10};
8       Length second{ value: 10};
9
10      if (first == second)
11
12      return 0;
13  }
```

Overloading the Equality Operator

Length.cpp | Length.h | main.cpp

```
8
9   class Length {
10  public:
11      explicit Length(int value);
12      bool operator==(const Length& other) const;
13  private:
14      int value;
15  };
```

Length.cpp | Length.h | main.cpp

```
6
7   Length::Length(int value) : value(value) {}
8
9   bool Length::operator==(const Length &other) const {
10      return value == other.value;
11  }
```

Length.cpp | Length.h | main.cpp

```
5
6   int main() {
7       Length first{ value: 10};
8       Length second{ value: 10};
9
10      if (first == second)
11
12      return 0;
13  }
```

Overloading the Equality Operator

```
Length.cpp  Length.h  main.cpp

 8
 9  class Length {
10  public:
11      explicit Length(int value);
12      bool operator==(const Length& other) const;
13  private:
14      int value;
15  };
```

```
Length.cpp  Length.h  main.cpp

 5
 6  int main() {
 7      Length first{ value: 10};
 8      Length second{ value: 10};
 9
10      if (first == 10)
11
12      return 0;
13  }
```

- Since the explicit keyword is used in the constructor, we cannot compare a Length object with an integer.

Overloading the Equality Operator

Length.cpp | Length.h | main.cpp

```
8
9   class Length {
10  public:
11      explicit Length(int value);
12      bool operator==(const Length& other) const;
13      bool operator==(int other) const;
14  private:
15      int value;
16  };
```

- ❑ **Solution:** Add another overload.

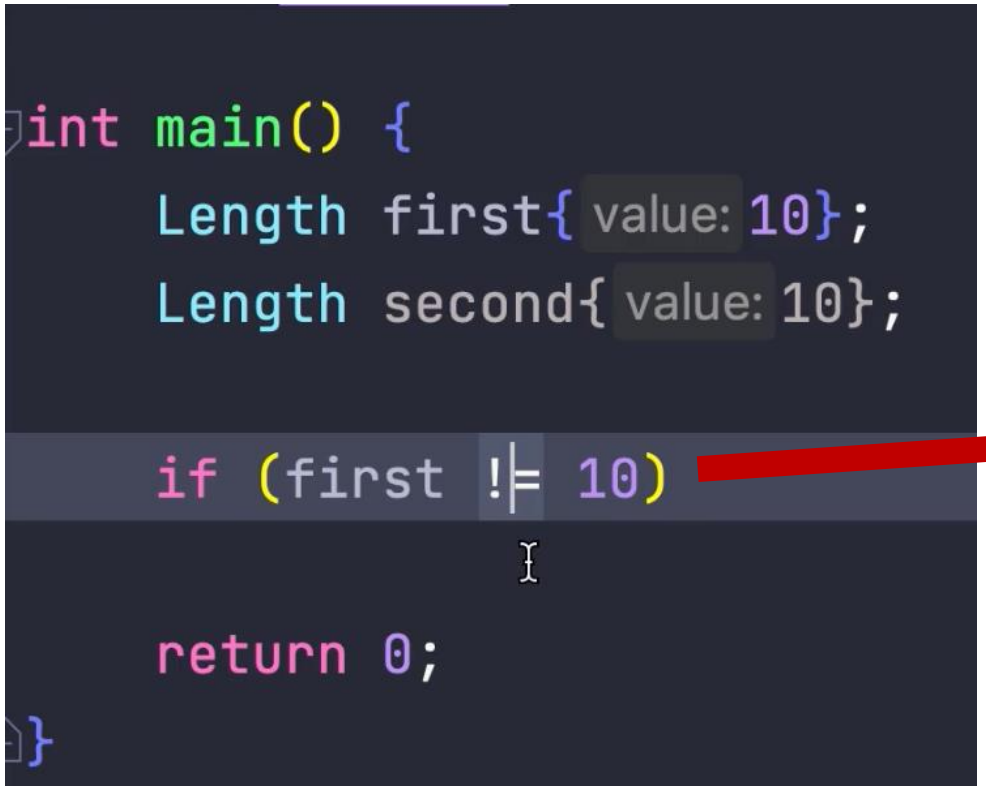
Length.cpp | Length.h | main.cpp

```
6
7   Length::Length(int value) : value(value) {}
8
9   bool Length::operator==(const Length &other) const {
10      return value == other.value;
11  }
12
13  bool Length::operator==(int other) const {
14      return value == other;
15  }
```

Length.cpp | Length.h | main.cpp

```
5
6   int main() {
7       Length first{ value: 10};
8       Length second{ value: 10};
9
10      if (first == 10)
11
12      return 0;
13  }
```

Overloading the Equality Operator



- Modern C++ compilers automatically generate the inequality operator.

Overloading the Equality Operator

```
Length.cpp  Length.h  main.cpp

8
9   class Length {
10  public:
11      explicit Length(int value);
12      bool operator==(const Length& other) const;
13      bool operator==(int other) const;
14      bool operator!=(int other) const;
15  private:
16      int value;
17  };
```

```
Length.cpp  Length.h  main.cpp

6
7   Length::Length(int value) : value(value) {}
8
9   bool Length::operator==(const Length &other) const {
10      return value == other.value;
11  }
12
13  bool Length::operator==(int other) const {
14      return value == other;
15  }
16
17  bool Length::operator!=(int other) const {
18      return !(value == other);
19  }
```

- Implementing our own inequality operator using equality operator.

```
int main() {
    Length first{ value: 10};
    Length second{ value: 10};

    if (first != 10)

    return 0;
}
```

Overloading the Comparison Operator

- Implementing < and > operators.

Length.cpp | Length.h | main.cpp

```
9   class Length {
10  public:
11      explicit Length(int value);
12      bool operator==(const Length& other) const;
13      bool operator==(int other) const;
14      bool operator!=(int other) const;
15      bool operator<(const Length& other) const;
16      bool operator<=(const Length& other) const;
17      bool operator>(const Length& other) const;
18      bool operator>=(const Length& other) const;
19  private:
20      int value;
21  };
```

```
20
21  bool Length::operator<(const Length &other) const {
22      return value < other.value;
23  }
24
25  bool Length::operator>(const Length &other) const {
26      return value > other.value;
27  }
```

Length.cpp | Length.h | main.cpp

```
5
6   int main() {
7       Length first{ value: 10};
8       Length second{ value: 10};
9
10      if (first < second)
11
12      return 0;
13  }
```

Overloading the Comparison Operator

- Implementing <= and >= operators using < and > operators implemented.

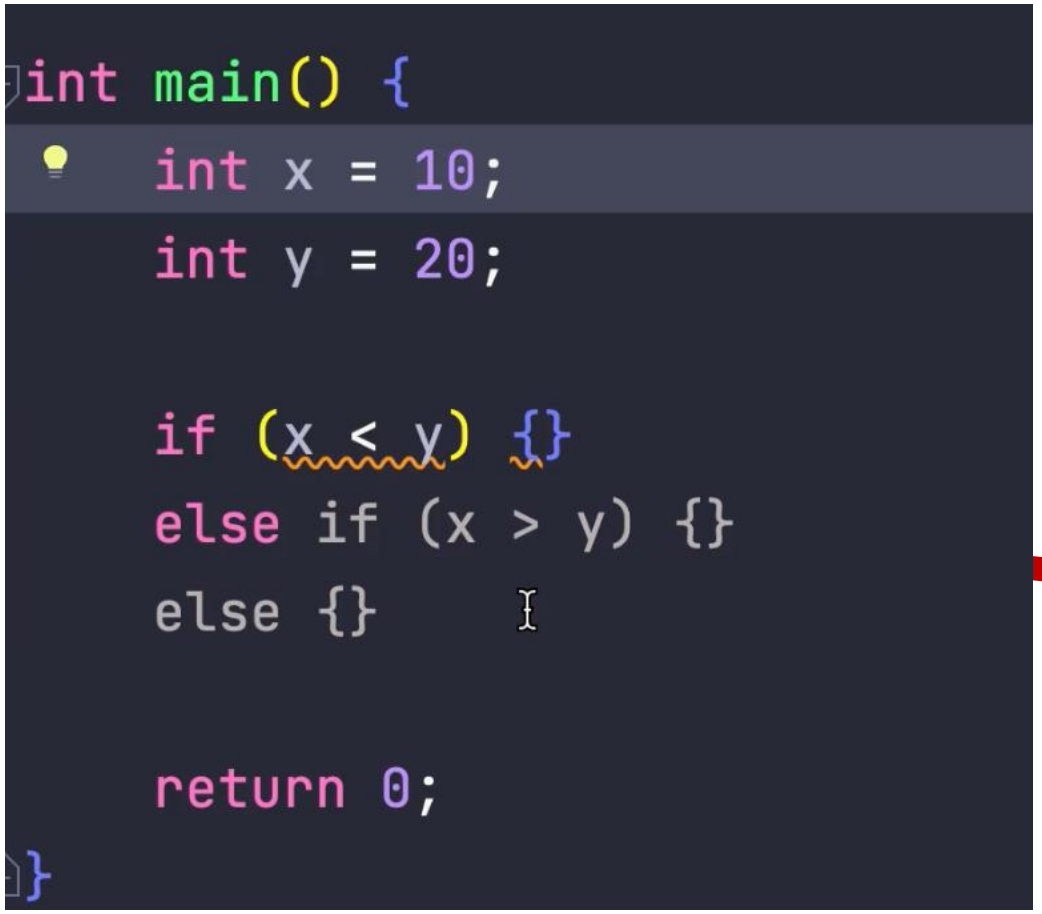
Length.cpp | Length.h | main.cpp

```
9   class Length {
10  public:
11      explicit Length(int value);
12      bool operator==(const Length& other) const;
13      bool operator==(int other) const;
14      bool operator!=(int other) const;
15      bool operator<(const Length& other) const;
16      bool operator<=(const Length& other) const;
17      bool operator>(const Length& other) const;
18      bool operator>=(const Length& other) const;
19  private:
20      int value;
21  };
```

```
28
29  bool Length::operator<=(const Length &other) const {
30      return !(value > other.value);
31  }
32
33  bool Length::operator>=(const Length &other) const {
34      return !(value < other.value);
35  }
```

Length.cpp | Length.h | main.cpp

```
5
6   int main() {
7       Length first{ value: 10};
8       Length second{ value: 10};
9
10      if (first ≤ second)
11
12      return 0;
13  }
```



Overloading the Spaceship Operator

- Also called 3-way comparison operator.
- When the variables are simple, then this comparison is cheap.
- It gets expensive when we have more complex variables that need to be compared.

Overloading the Spaceship Operator

```cpp
int main() {
    int x = 10;
    int y = 20;
    auto result : strong_ordering = x <=> y;

    if (result == strong_ordering::less) {}
    else if (result == strong_ordering::greater) {}
    else {}

    return 0;
```

- Spaceship operator

```cpp
int main() {
    int x = 10;
    int y = 20;

    if (x < y) {}
    else if (x > y) {}
    else {}

    return 0;
}
```

Overloading the Spaceship Operator

- We can remove all of the overloading implementation of <, <, >=, and <= operators.
- C++ compiler generates all the implementation of these operators for us.
- We still need the == overloading.

Length.cpp | Length.h | main.cpp

```
4
5   #ifndef ADVANCED_LENGTH_H
6   #define ADVANCED_LENGTH_H
7
8   #include <compare>
9
10  class Length {
11  public:
12      explicit Length(int value);
13      bool operator==(const Length& other) const;
14      bool operator==(int other) const;
15      bool operator!=(int other) const;
16      std::strong_ordering operator<=>(const Length& other) const;
17  private:
18      int value;
19  };
```

```
20
21  std::strong_ordering Length::operator<=>(const Length &other) const {
22      return value <=> other.value;
23  }
```

```
int main() {
    Length first{ value: 10};
    Length second{ value: 20};

    if (first < second)
        cout << "First is smaller";



    return 0;
}
```

Overloading the Spaceship Operator

```cpp
int main() {
    int x = 10;
    int y = 20;
    auto result : strong_ordering = x <=> y;

    if (result == strong_ordering::less) {}
    else if (result == strong_ordering::greater) {}
    else {}

    return 0;
```

- Spaceship operator

```cpp
int main() {
    int x = 10;
    int y = 20;

    if (x < y) {}
    else if (x > y) {}
    else {}

    return 0;
}
```

main.cpp | Length.h | Length.cpp

```
5   #ifndef ADVANCED_LENGTH_H
6   #define ADVANCED_LENGTH_H
7
8   #include <compare>
9   #include <ostream>
10
11  using namespace std;
12
13  class Length {
14  public:
15      explicit Length(int value);
16      bool operator==(const Length& other) const;
17      bool operator==(int other) const;
18      strong_ordering operator<=>(const Length& other) const;
19  private:
20      int value;
21  };
22
23  ostream& operator<<(ostream& stream, const Length& length);
24
25
26  #endif //ADVANCED_LENGTH_H
27
```

Overloading the Stream Insertion Operator

- Has to be declared outside of the class as a global function.

Overloading the Stream Insertion Operator

main.cpp | Length.h | Length.cpp

```
5   #ifndef ADVANCED_LENGTH_H
6   #define ADVANCED_LENGTH_H
7
8   #include <compare>
9   #include <ostream>
10
11  using namespace std;
12
13  class Length {
14  public:
15      explicit Length(int value);
16      bool operator==(const Length& other) const;
17      bool operator==(int other) const;
18      strong_ordering operator<=>(const Length& other) const;
19      int getValue() const;
20      void setValue(int value);
21  private:
22      int value;
23  };
21  };
22
23  ostream& operator<<(ostream& stream, const Length& length);
24
25
26  #endif //ADVANCED_LENGTH_H
```

- It has to be declared outside of the class as a global function because the first parameter is not an object.

Overloading the Stream Insertion Operator

```
main.cpp  Length.h  Length.cpp
25  void Length::setValue(int value) {
26      Length::value = value;
27  }
28
29  ostream &operator<<(ostream &stream, const Length &length) {
30      stream << length.getValue();
31      return stream;
32  }
```

- Implementation of the insertion (<<) operator.

```
main.cpp  Length.h  Length.cpp
6   int main() {
7       Length length{ value: 10};
8       cout << 1 << 2 << 3;
9
10      return 0;
11  }
```

Friends of Classes

main.cpp | Length.h | Length.cpp

```
28
29  ostream &operator<<(ostream &stream, const Length &length) {
30      stream << length.getValue();
31      return stream;
32  }
33
34  istream &operator>>(istream &stream, Length &length) {
35      int value;
36      stream >> value;
37      length.setValue(value);
38      return stream;
```

- The operator that is defined outside of the class accesses private member of the class (value) using getter and setter functions.

- If an operator defined outside of the class needs to access a private member that has no getter or setter function, then we need to use

```
ostream &operator<<(ostream &stream, const Length &length) {
    stream << length.getValue();
    length.x
    return stream;
}
```

- Not allowed because x is a private member of the class.

Friends of Classes

main.cpp | Length.h | Length.cpp

```
21      void setValue(int value);
22  private:
23      int value;
24      int x;
25      friend ostream& operator<<(ostream& stream, const Length
26
27  };
28
29  ostream& operator<<(ostream& stream, const Length& length);
30  istream& operator>>(istream& stream, Length& length);
```

- Solution: Define the function that wants to access a private member of the class as a friend of the class.

```
ostream &operator<<(ostream &stream, const Length &length) {
    stream << length.getValue();
    length.x
    return stream;
}
```

- Now the function can access the private member, x, of the class.

Inline Functions

- We can define a function in the header file where it is declared as an inline function.
- The call of an inline function is replaced with the code in the inline function.
- It is up to the C++ compiler to replace the code for performance reasons.

main.cpp | Length.h | Length.cpp

```
28
29      // Inline function
30      int getValue() const {
31          return value;
32      }
33      void setValue(int value);
34  private:
35      int value;
36  };
```

main.cpp | Length.h | Length.cpp

```
21
22  int Length::getValue() const {
23      return value;
24  }
25
```

main.cpp | Length.h | Length.cpp

```
6   int main() {
7       Length length{ value: 10};
8       length.getValue()
9
10      return 0;
11  }
```

Inline Functions

```
main.cpp  Length.h  Length.cpp
25      Length operator++(int); // postfix
26      explicit operator int() const;
27
28      int getValue() const;
29      void setValue(int value);
30  private:
31      int value;
32  };
```

- We can also define the function as inline in the implementation file (not in the header file).

```
main.cpp  Length.h  Length.cpp
21
22  inline int Length::getValue() const {
23      return value;
24  }
25
```

- Inline functions can increase the complexity of the code, so it is not recommended in general.