

C++ Notes

Part 3-1

**Slides were created from
CodewithMosh.com**

Classes

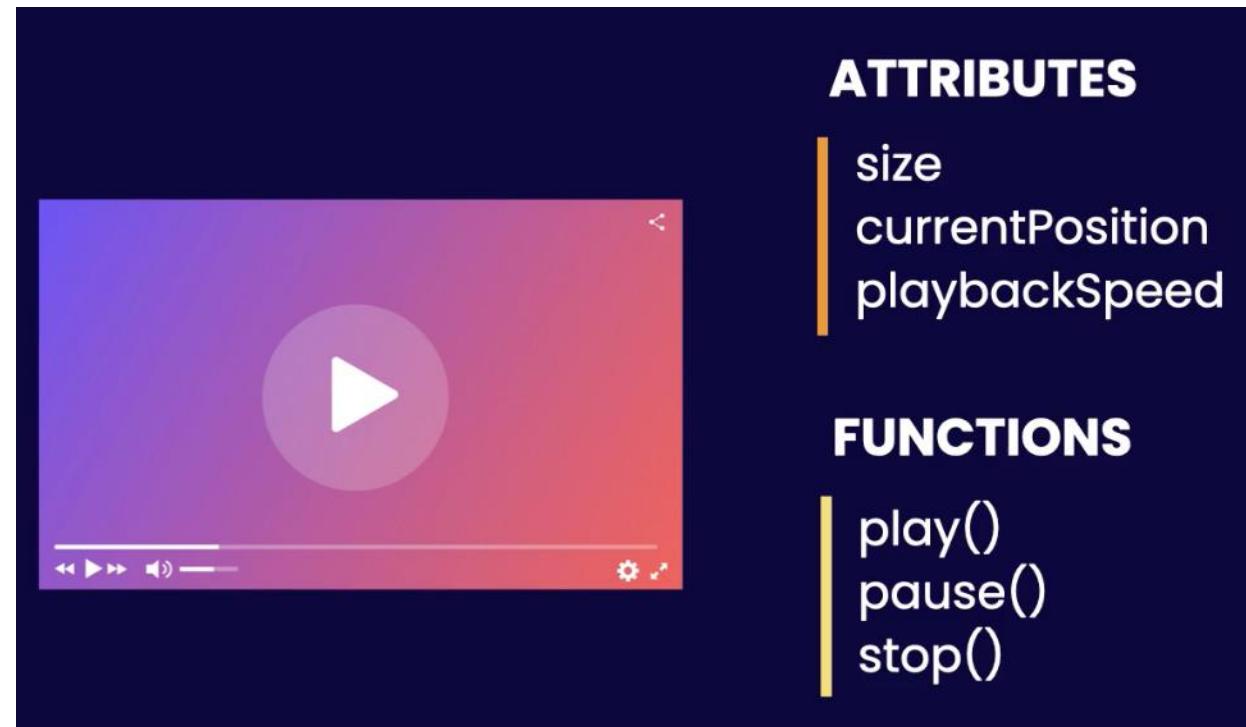
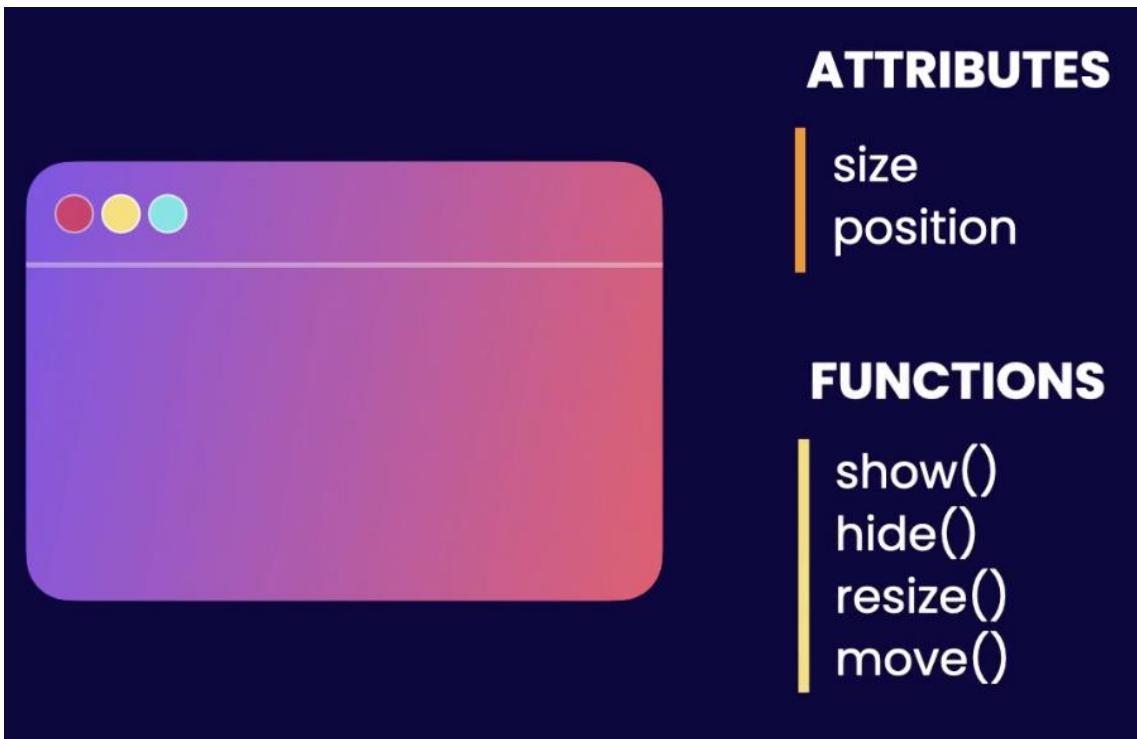
- The basics of object-oriented programming
- Classes
- Access modifiers (public, private)
- Constructors
- Destructors
- Static members

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.

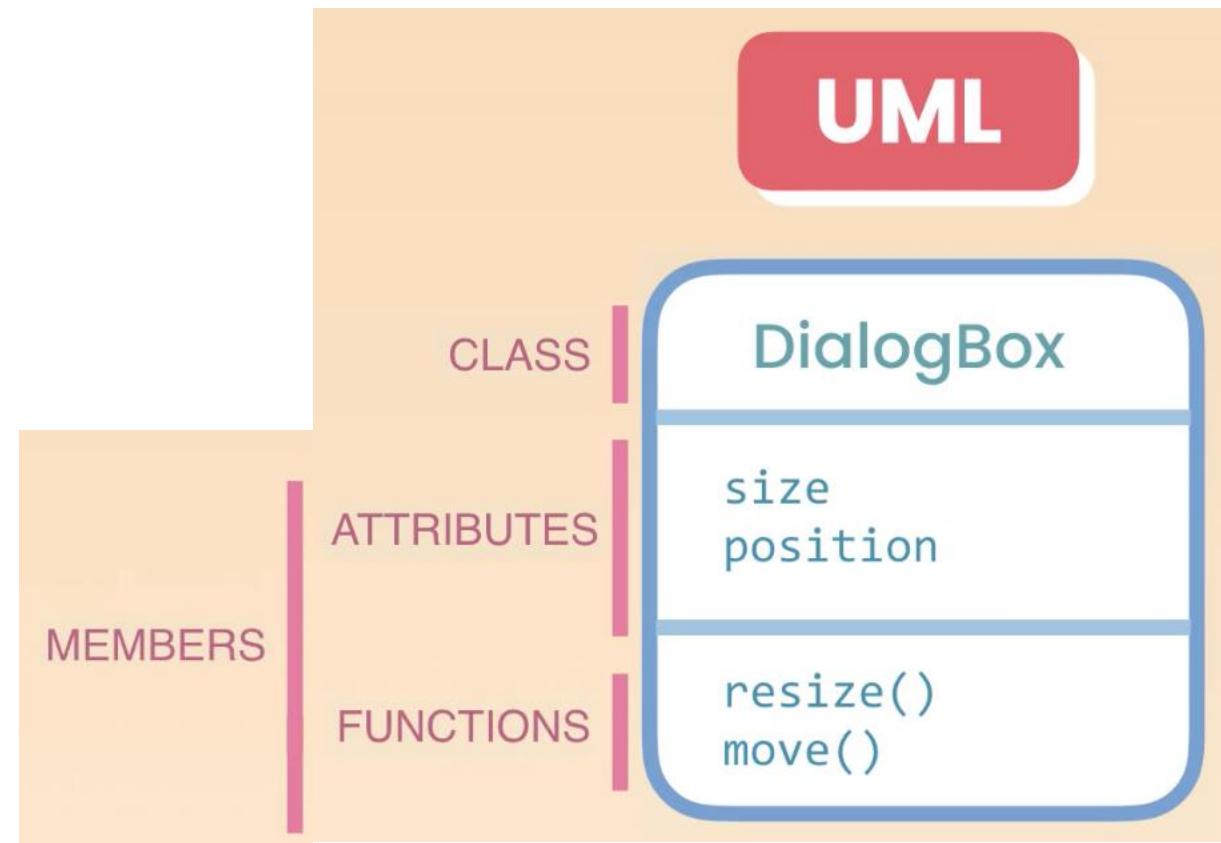
Introduction to Object-Oriented Programming

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

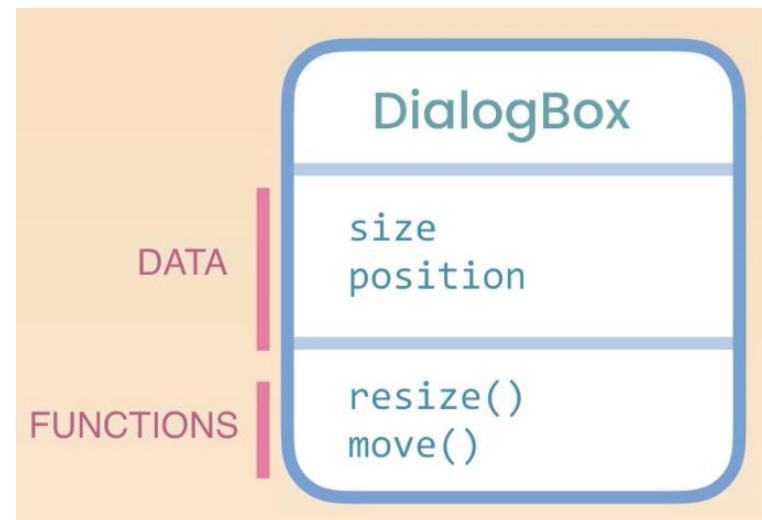


Introduction to Object-Oriented Programming

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

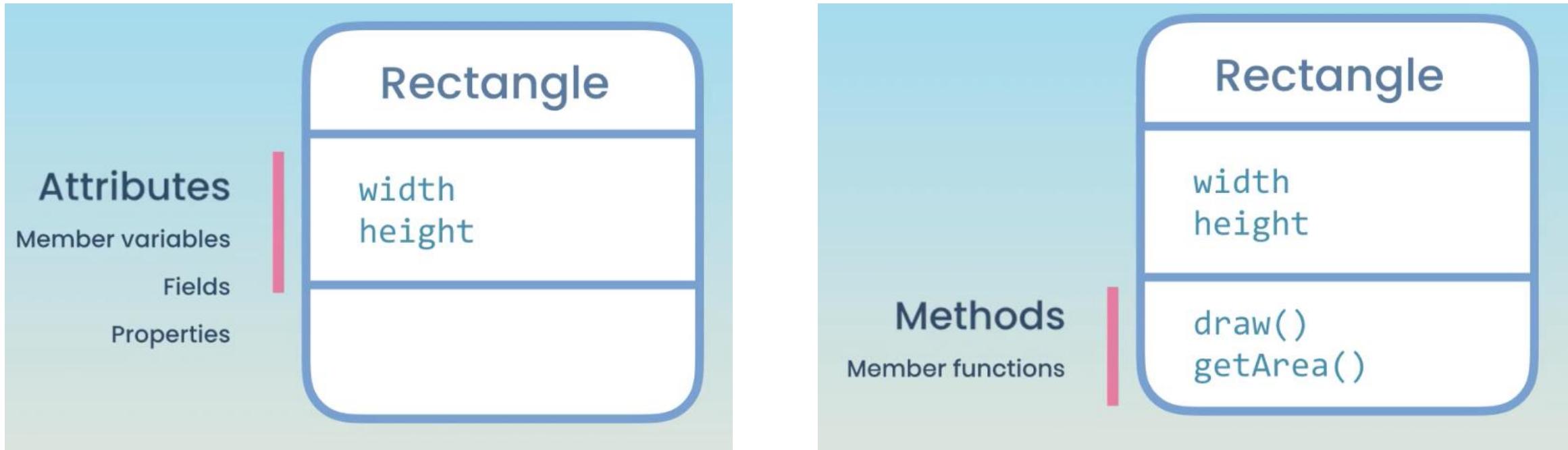


Introduction to Object-Oriented Programming



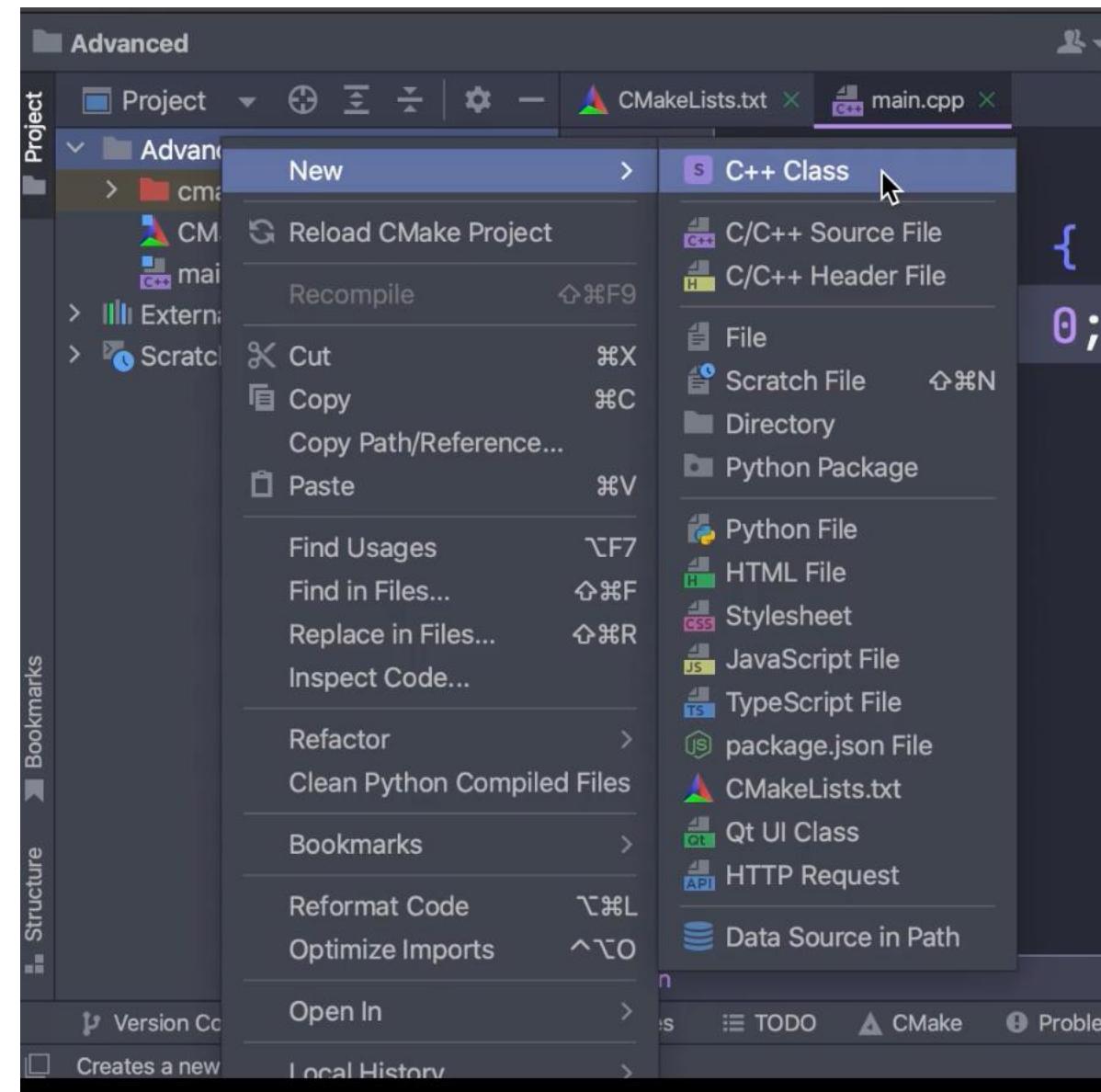
- ❑ **Encapsulation:** Combining the data and functions that operate on the data together into one unit.
- ❑ **Object:** An instance of a class.

Defining a Class



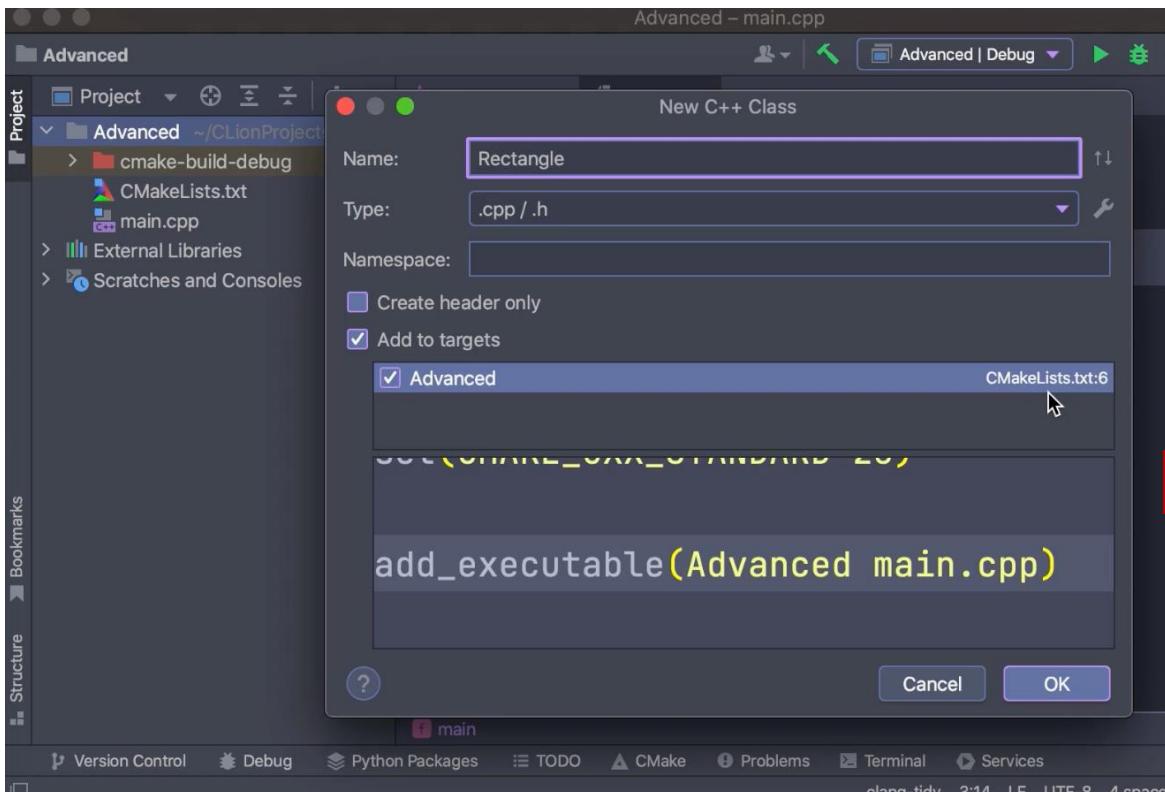
- ❑ **Officially in C++:** member variables and member functions.
- ❑ **In UML:** Attributes and methods are used.

Defining a Class



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

Defining a Class



The screenshot shows the CLion IDE interface with the file 'Rectangle.h' open. The code defines a class 'Rectangle' with a private destructor. The code is as follows:

```
4
5 #ifndef ADVANCED_RECTANGLE_H
6 #define ADVANCED_RECTANGLE_H
7
8 class Rectangle {
9     ~Rectangle();
10};
11
12
13#endif // ADVANCED_RECTANGLE_H
```

Defining a Class

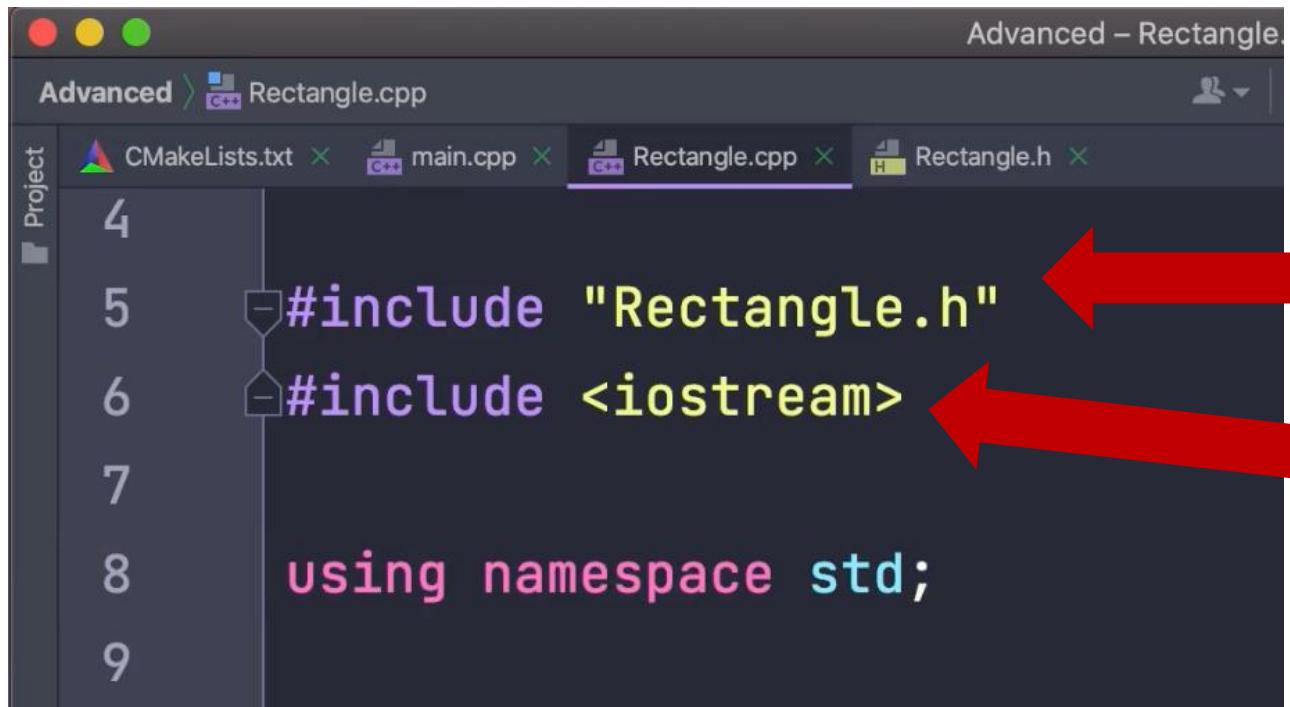
```
#ifndef ADVANCED_RECTANGLE_H
#define ADVANCED_RECTANGLE_H

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



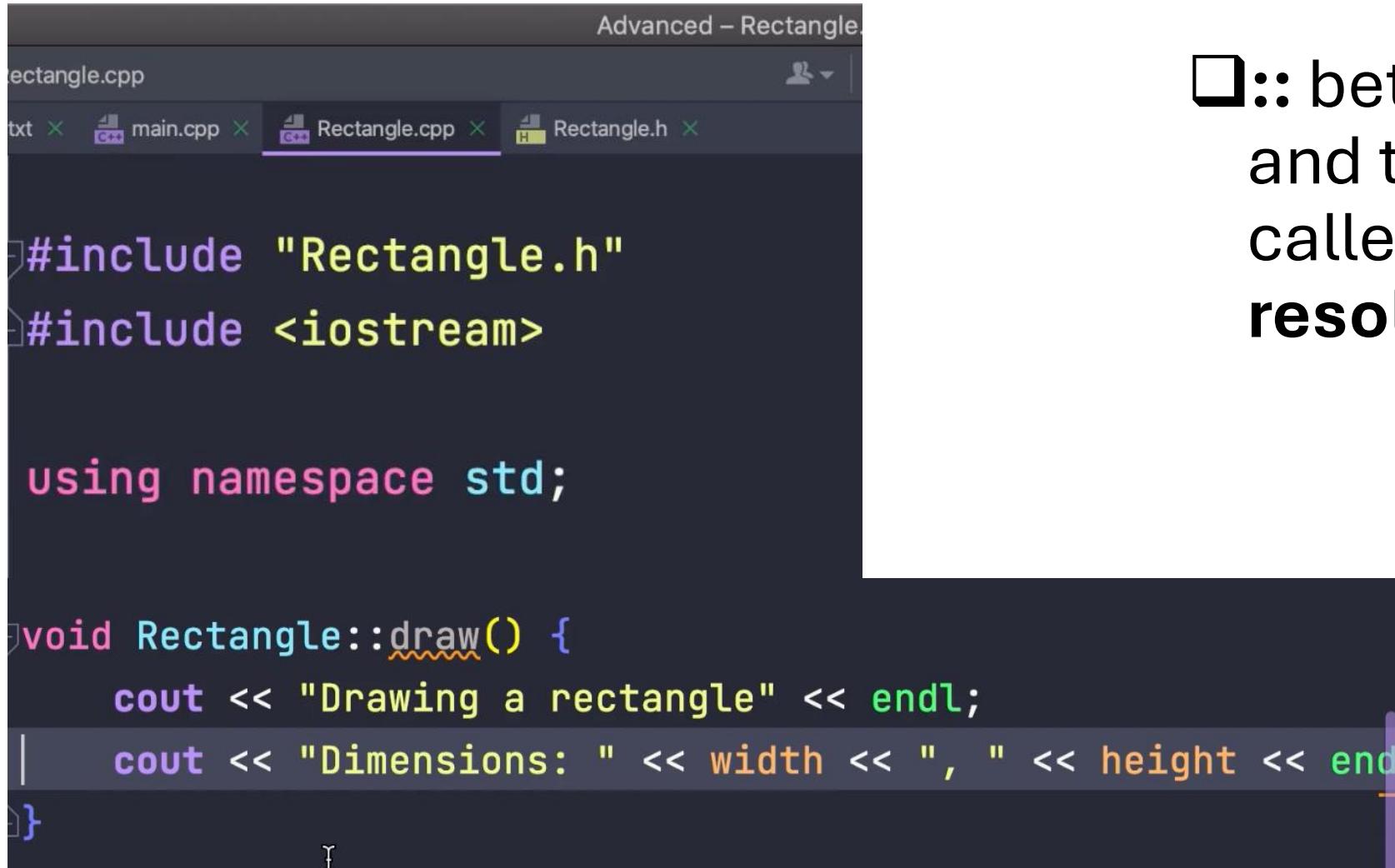
A screenshot of a C++ IDE interface titled "Advanced – Rectangle". The project navigation bar shows files: CMakeLists.txt, main.cpp, Rectangle.cpp (selected), and Rectangle.h. The code editor displays the following C++ code:

```
4  
5 #include "Rectangle.h"  
6 #include <iostream>  
7  
8 using namespace std;  
9
```

Two red arrows point from the text "Include the header file in .cpp main file." and "Standard C++ files are included using <>." to the "#include" lines in the code editor.

- ❑ 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



The screenshot shows a code editor window titled "Advanced – Rectangle". The tab bar at the top has "rectangle.cpp", "main.cpp", "Rectangle.cpp" (which is selected), and "Rectangle.h". The code editor displays the following C++ code:

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

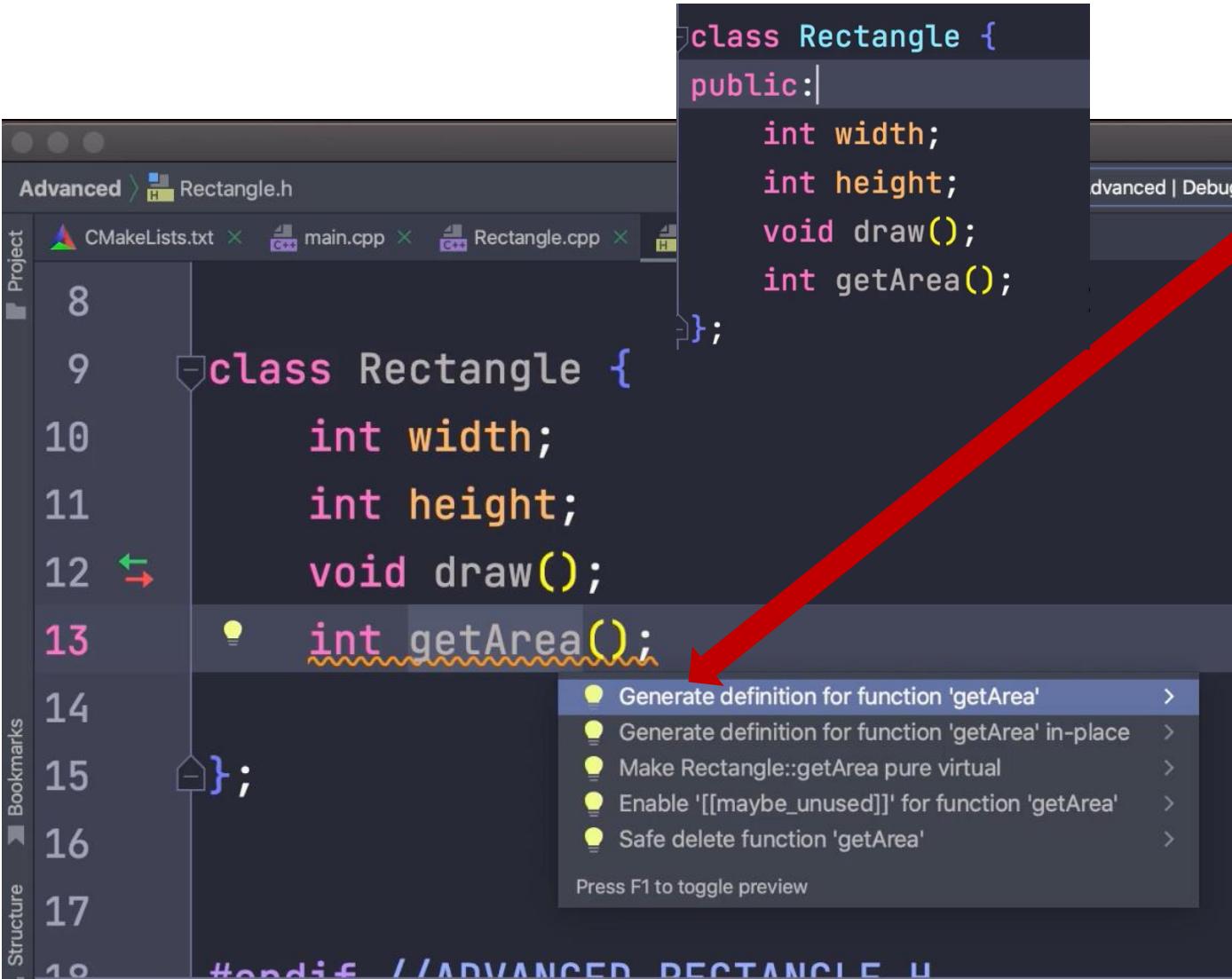
using namespace std;

void Rectangle::draw() {
    cout << "Drawing a rectangle" << endl;
    cout << "Dimensions: " << width << ", " << height << endl
}
```

❑ `::` between the class name and the function name is called the “**scope resolution operator**”.

❑ Implementation of draw function (method).

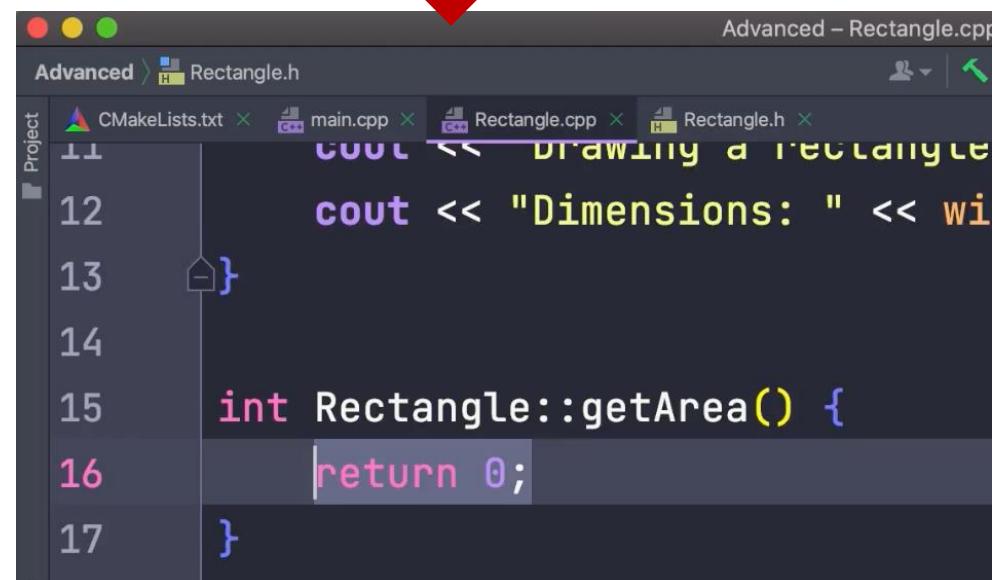
Defining a Class



```
class Rectangle {  
public:  
    int width;  
    int height;  
    void draw();  
    int getArea();  
};  
  
class Rectangle {  
public:  
    int width;  
    int height;  
    void draw();  
    int getArea();  
};  
};  
  
#endif // ADVANCED_RECTANGLE_H
```

A screenshot of a C++ IDE showing the `Rectangle.h` file. The code defines a `Rectangle` class with private members `width` and `height`, a `draw()` method, and a `getArea()` method. A red arrow points from the `getArea()` declaration in the header to a context menu that appears when the mouse is right-clicked on it. The menu contains five options: "Generate definition for function 'getArea'", "Generate definition for function 'getArea' in-place", "Make Rectangle::getArea pure virtual", "Enable '[[maybe_unused]]' for function 'getArea'", and "Safe delete function 'getArea'".

□ 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`.



```
cout << "Drawing a rectangle"  
cout << "Dimensions: " << wi  
}  
  
int Rectangle::getArea() {  
    return 0;  
}
```

A screenshot of the C++ IDE showing the `Rectangle.cpp` file. It contains the implementation for the `getArea()` method, which returns 0. A red arrow points from the `getArea()` declaration in the `Rectangle.h` header to the corresponding implementation in the `Rectangle.cpp` file.

rectangle.cpp

txt main.cpp Rectangle.cpp Rectangle.h

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

using namespace std;

void Rectangle::draw() {
    cout << "Drawing a rectangle" << endl;
    cout << "Dimensions: " << width << ", " << height << endl;
}

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

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.

The screenshot shows a C++ IDE interface with the following details:

- Title Bar:** Advanced – main.cpp
- Project Bar:** Advanced > C++ main.cpp
- File List:** main.cpp (selected), Rectangle.h, Rectangle.cpp
- Code Editor:** The main.cpp file contains the following code:

```
1 #include "Rectangle.h"
2 #include <iostream>
3
4 using namespace std;
5
6 int main() {
7     Rectangle rectangle;
8     rectangle.width = 10;
9     rectangle.height = 20;
10    cout << rectangle.getArea();
11
12    return 0;
13 }
```
- Sidebar:** Project, Bookmarks, Structure

A screenshot of a C++ IDE interface. The title bar says "Advanced – main.cpp". The project navigation bar shows "Advanced > C++ main.cpp". The code editor displays the following C++ code:

```
1 #include "Rectangle.h"
2
3 #include <iostream>
4
5 using namespace std;
6
7 int main() {
8     Rectangle rectangle;
9     rectangle.width = 10;
10    rectangle.height = 20;
11    cout << rectangle.getArea();
12
13    return 0;
14}
```

Creating Objects

```
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 screenshot shows a code editor window titled "Advanced – main.cpp". The file contains the following C++ code:

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

using namespace std;

int main() {
    Rectangle first;
    Rectangle second;
    cout << &first << endl;
    cout << &second << endl;

    return 0;
}
```

The word "Rectangle" is underlined with a red squiggly line, indicating a potential error or warning in the IDE.

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

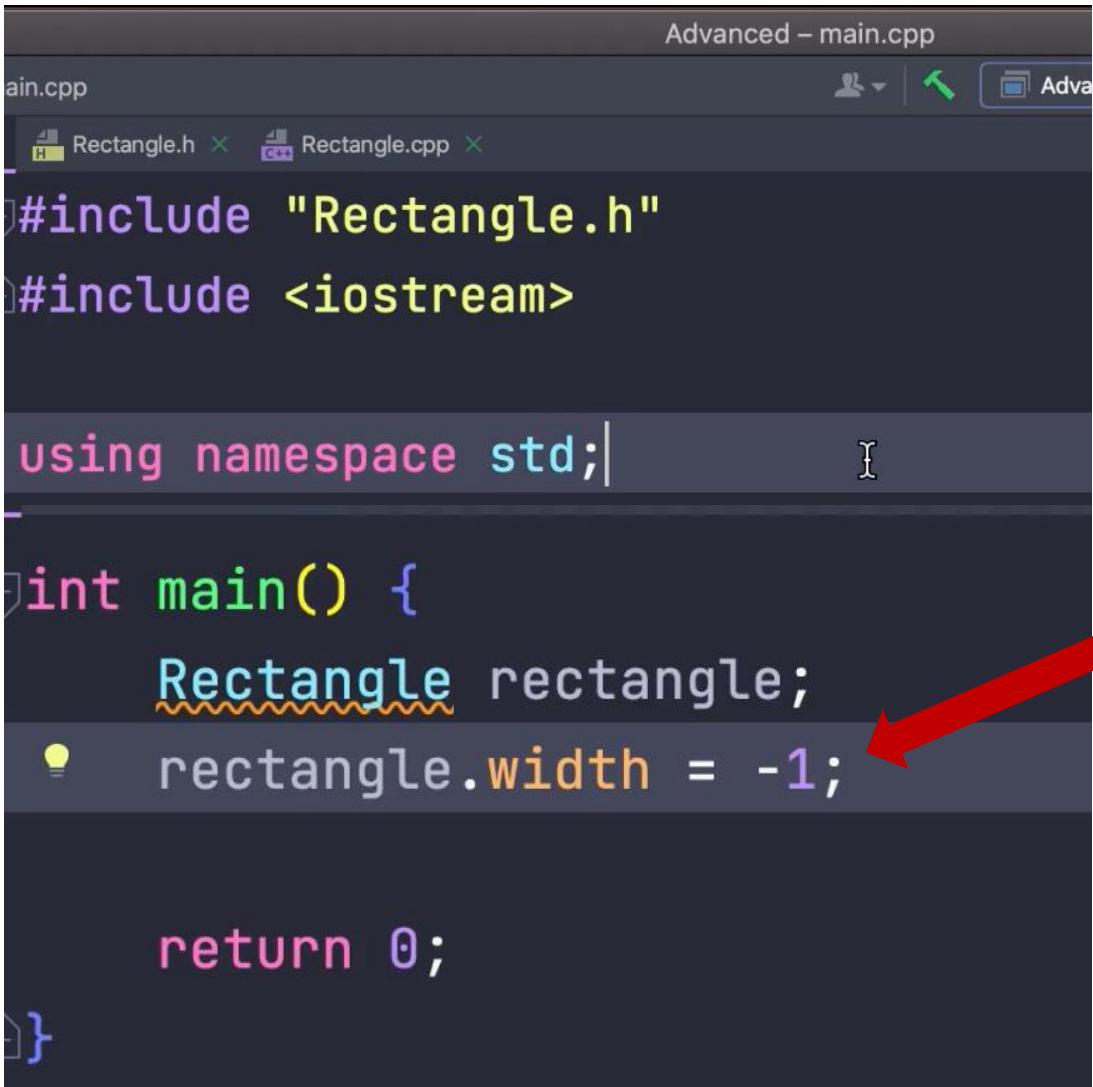
```
0x7ffeeb6389c0
0x7ffeeb6389b8
```

```
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++.



A screenshot of a C++ IDE showing a file named "Advanced - main.cpp". The code contains the following:

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

using namespace std;

int main() {
    Rectangle rectangle;
    rectangle.width = -1;

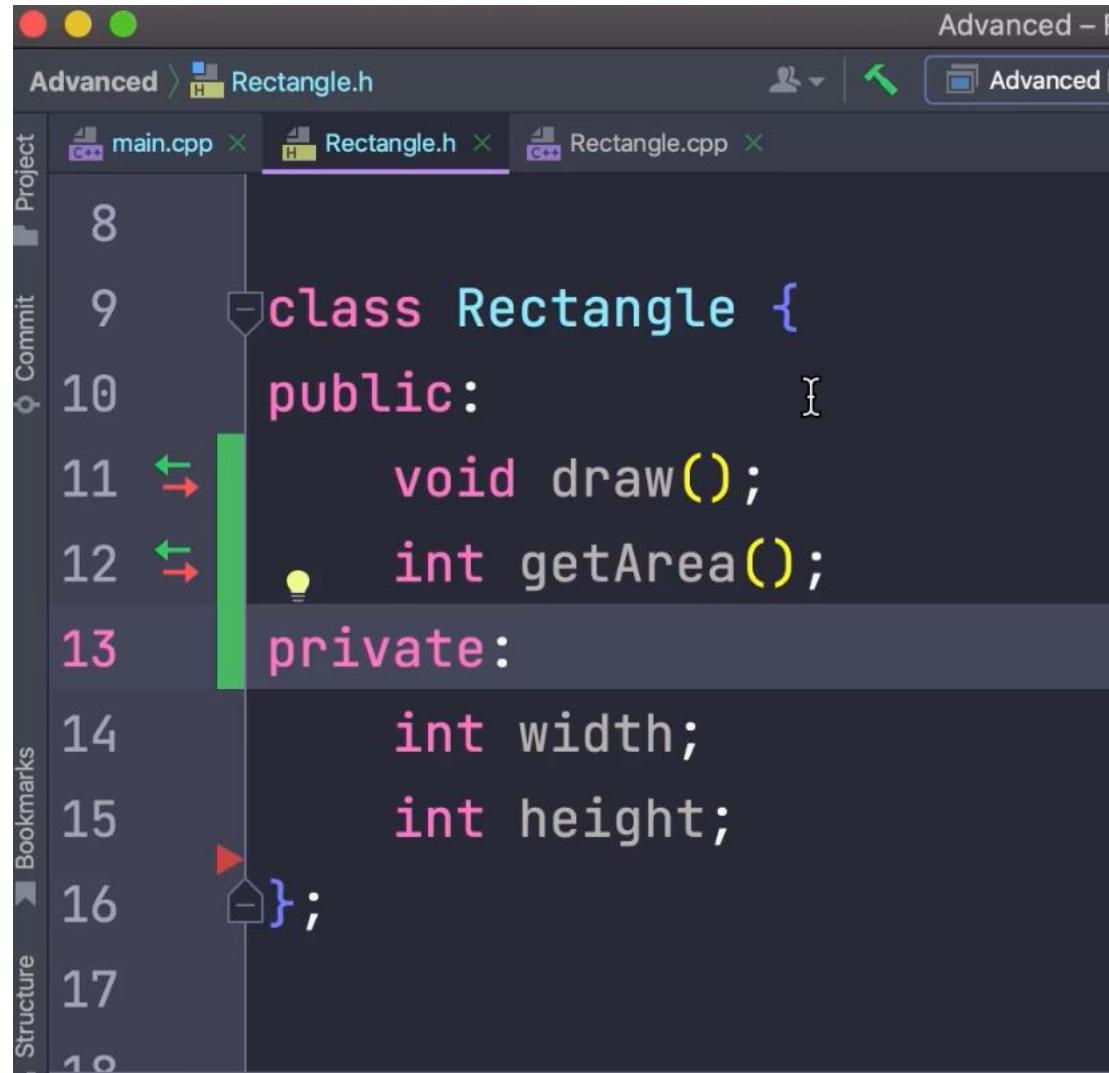
    return 0;
}
```

A red arrow points from the text "rectangle.width = -1;" to the "width" member variable in the code.

Data Hiding

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

Access Modifiers



A screenshot of a code editor showing the `Rectangle.h` file. The code defines a `Rectangle` class with `public` methods `draw()` and `getArea()`, and `private` data members `width` and `height`. The code editor interface includes tabs for `main.cpp`, `Rectangle.h`, and `Rectangle.cpp`, and various toolbars and status bars.

```
8
9 class Rectangle {
10 public:
11     void draw();
12     int getArea();
13 private:
14     int width;
15     int height;
16 };
```

- 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

```
class Rectangle {  
public:  
    void draw();  
    int getArea();  
    // Getter (accessor)  
    int getWidth();  
    // Setter (mutator)  
    void setWidth(int width);  
private:  
    int width;  
    int height;  
};
```

- 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;  
}
```

```
int main() {  
    Rectangle rectangle;  
    rectangle.setWidth(-1);  
  
    return 0;  
}
```

❑ Implementation of **setWidth** function.

❑ This is an exception.

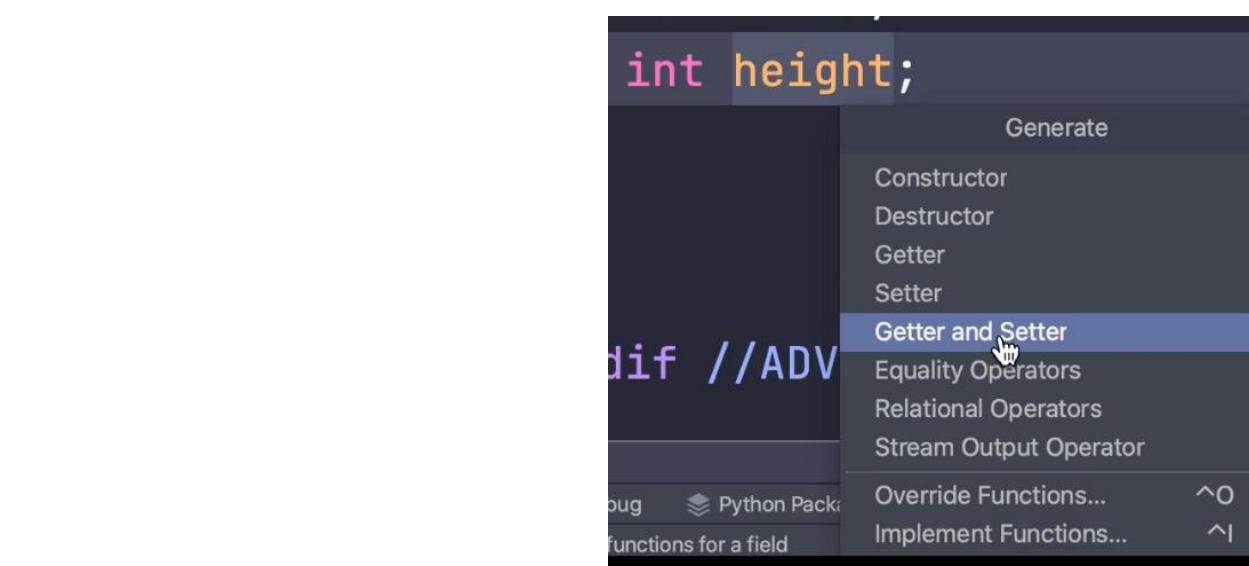
❑ **this->** refers to the current object.

❑ **(*this).width**, **m_width**, or **Rectangle::width** can also be used.

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

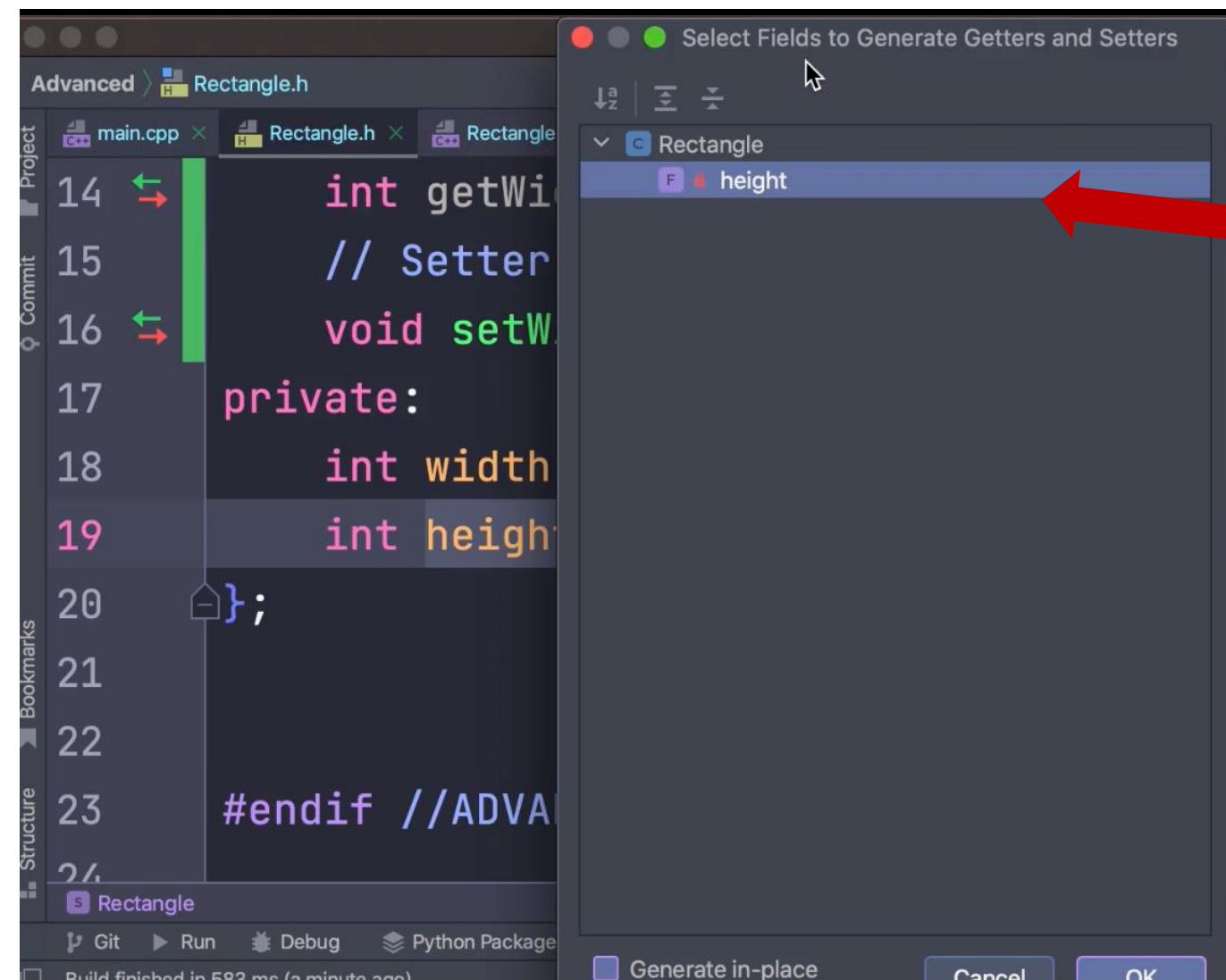
Getters and Setters

```
int getWidth();  
// Setter (mutator)  
void setWidth(int width);  
  
private:  
    int width;  
    int heig  
};  
  
#endif //ADV
```



- ❑ 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



The screenshot shows a code editor interface with a file named `Rectangle.h` open. The code defines a `Rectangle` class with private attributes `width` and `height`, along with their corresponding getters and setters. A red arrow points to the `height` attribute in the `Rectangle` class definition, which is highlighted in the list of fields in the dialog.

```
Advanced > Rectangle.h
main.cpp Rectangle.h Rectangle
14 int getWidth();
15 // Setter
16 void setWidth(int width);
17
private:
18     int width;
19     int height;
20 };
21
22
23 #endif // ADVANCED_RECTANGLE_H_
24
```

Select Fields to Generate Getters and Setters

Rectangle

height

Generate in-place Cancel OK

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

Getters and Setters

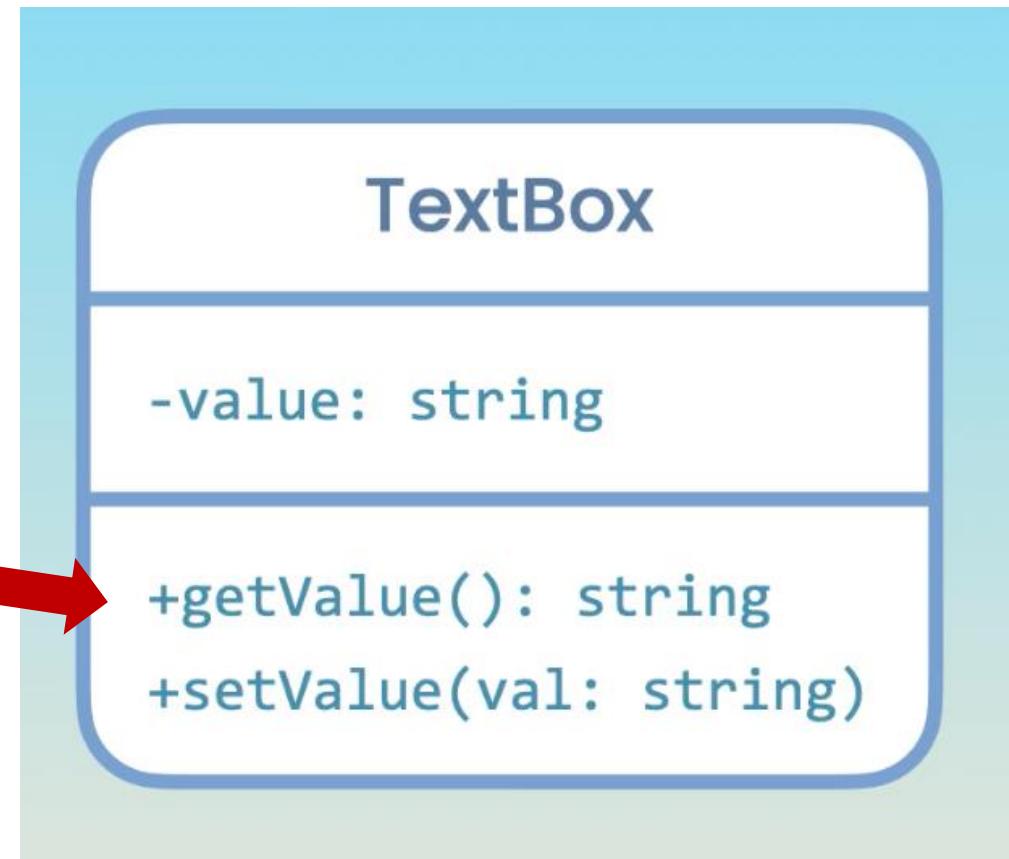
```
int Rectangle::getHeight() const {
    return height;
}

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

❑ Implementation of **setHeight** function.

Example

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



Example

```
#define ADVANCED_TEXTBOX_H
```

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

A screenshot of a code editor showing the declaration of a `TextBox` class in a header file (`TextBox.h`). The code defines a public interface with `getValue()` and `setValue(const string& value)` methods, and a private member variable `value`. The code is annotated with red squiggly underlines under `string`, `get`, `set`, and `value`, indicating potential errors or warnings.

```
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
```

A screenshot of a code editor showing the implementation of the `TextBox` class in a source file (`TextBox.cpp`). It includes the header file inclusion, the `getValue()` method which returns the private `value`, and the `setValue()` method which assigns the new value to `this->value`. The code is annotated with red squiggly underlines under `string`, `get`, `set`, and `value`, indicating potential errors or warnings.

```
3
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

The screenshot shows a C++ development environment with the following details:

- Title Bar:** Advanced – main.cpp
- Toolbar:** Includes standard icons for file operations, search, and navigation.
- Project Explorer:** Shows three files: main.cpp (selected), TextBox.cpp, and TextBox.h.
- Code Editor:** Displays the following C++ code:

```
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();
10
11     return 0;
12 }
```

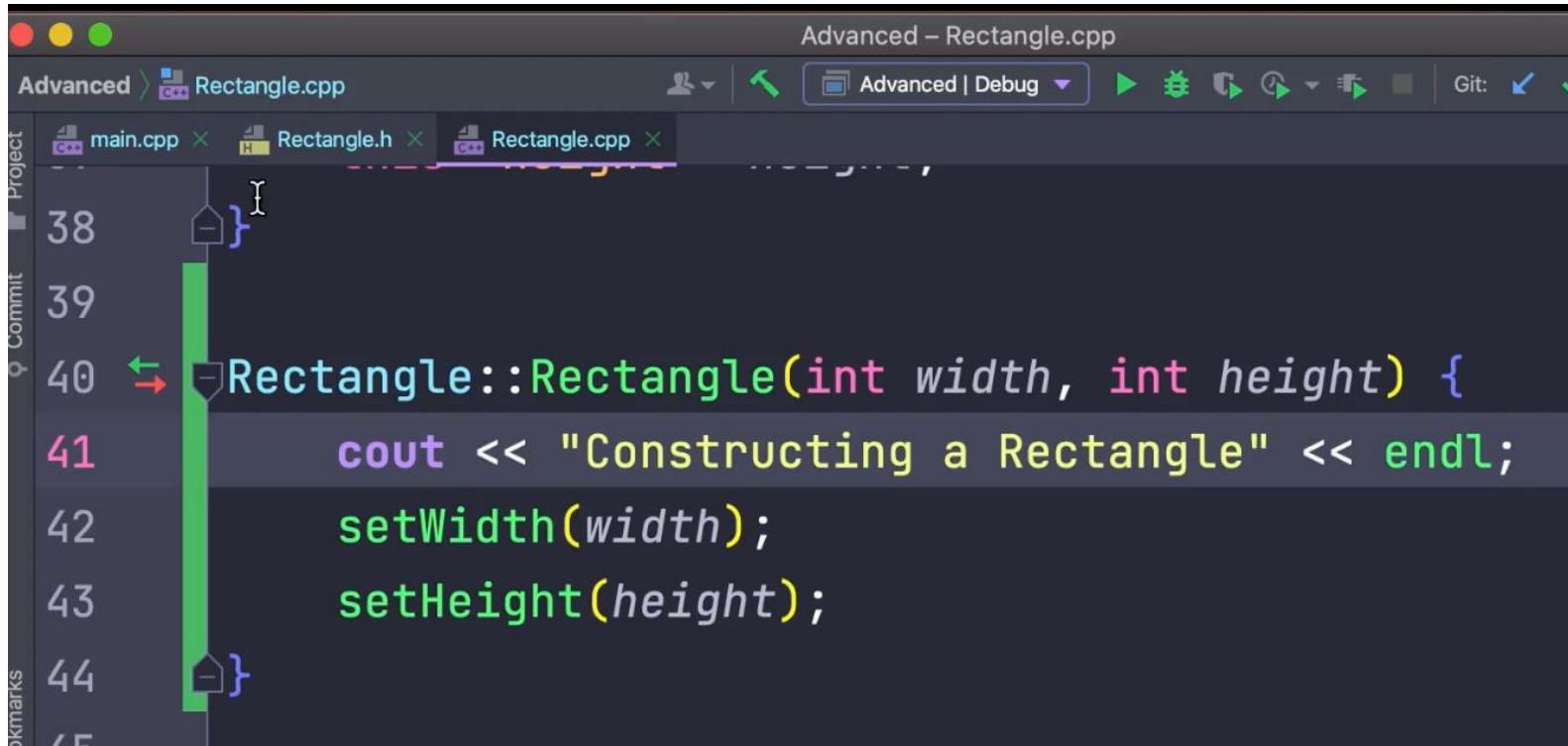
The code demonstrates the use of a `TextBox` class, which has methods `setValue` and `getValue`. The `main` function creates an instance of `TextBox`, sets its value to "Hello World", and then prints it to the console.

Constructors

```
class Rectangle {  
public:  
    Rectangle(int width, int height);  
    void draw();  
    int getArea();  
    int getWidth();  
    void setWidth(int width);  
    int getHeight() const;  
    void setHeight(int height);
```

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

Constructors



A screenshot of a C++ IDE interface. The title bar says "Advanced - Rectangle.cpp". The project navigation bar shows "Advanced" and "Rectangle.cpp" as the active file. The code editor displays the following C++ code:

```
Advanced - Rectangle.cpp
Advanced > C++ Rectangle.cpp
Project main.cpp Rectangle.h Rectangle.cpp
Commit
38 }
39
40 Rectangle::Rectangle(int width, int height) {
41     cout << "Constructing a Rectangle" << endl;
42     setWidth(width);
43     setHeight(height);
44 }
```

The code defines a constructor for a class named Rectangle, which takes two integer parameters: width and height. Inside the constructor, it prints a message to the console and calls two member functions, setWidth and setHeight, with the respective parameters.

- ❑ A special function that is used to initialize objects.

Constructors

A screenshot of a C++ IDE showing the main.cpp file. The code defines a Rectangle class and creates an object named rectangle with width 10 and height 20. It then prints the width of the rectangle.

```
Advanced – main.cpp
Advanced > C++ main.cpp Advanced | Debug Git:
Project Commit
main.cpp Rectangle.h Rectangle.cpp
5
6 ► int main() {
7     Rectangle rectangle( width: 10, height: 20 );
8     cout << rectangle.getWidth();
9
10    return 0;
11 }
```

- Initializing objects.

A screenshot of a C++ IDE showing the main.cpp file. The code defines a Rectangle class and creates an object named rectangle with width 10 and height 20. It then prints the width of the rectangle.

```
Advanced – main.cpp
Advanced > C++ main.cpp Advanced | Debug Git:
Project Commit
main.cpp Rectangle.h Rectangle.cpp
5
6 ► int main() {
7     Rectangle rectangle{ width: 10, height: 20 };
8     cout << rectangle.getWidth();
9
10    return 0;
11 }
```

- 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 screenshot shows a dark-themed IDE interface with the title bar "Advanced - main.cpp". The project navigation bar shows "Advanced" and "main.cpp". The code editor displays the following C++ code:

```
Advanced – main.cpp
Advanced > main.cpp
Project main.cpp Rectangle.h Rectangle.cpp
1
2
3
4
5
6 int main() {
7     Rectangle rectangle;
8     cout << rectangle;
9
10    return 0;
11 }
12
```

A tooltip is displayed over the line "Rectangle rectangle;" with the following message:

No matching constructor for initialization of 'Rectangle'
candidate constructor (the implicit copy constructor) not viable: requires 1 argument, but 0 were provided
candidate constructor (the implicit move constructor) not viable: requires 1 argument, but 0 were provided
candidate constructor not viable: requires 2 arguments, but 0 were provided

Below the tooltip, there are two options:

- Change signature of constructor 'Rectangle' to 'Rectangle()'
- More actions...

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.

A screenshot of a code editor showing the `Rectangle.h` header file. The code defines a `Rectangle` class with a public section containing a default constructor, a constructor taking width and height, a `draw` method, `getArea`, `getWidth`, `setWidth`, `getHeight`, and `setHeight` methods. The private section contains `width` and `height` variables both initialized to 0. The code editor interface shows tabs for `main.cpp`, `Rectangle.h`, and `Rectangle.cpp`. The status bar at the top right shows "Advanced - Rectangle.h".

```
Advanced – Rectangle.h
Advanced | Debug
Project
Commit
Bookmarks
ture
arks
Advanced > Rectangle.h
main.cpp Rectangle.h Rectangle.cpp
1 class Rectangle {
2     public:
3         Rectangle() = default;
4         Rectangle(int width, int height);
5         void draw();
6         int getArea();
7         int getWidth();
8         void setWidth(int width);
9         int getHeight() const;
10        void setHeight(int height);
11
12    private:
13        int width = 0;
14        int height = 0;
15
16    };
17
18
19
20
21
22 }
```

Using the Explicit Keyword

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

The screenshot shows a code editor in Clion for a file named Person.h. The code defines a class Person with a single-argument constructor. A red arrow points from the text "Warning from the Clion." to the constructor definition. A tooltip window titled "Clang-Tidy" provides five suggestions:

- 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

```
Advanced - Person.h
Advanced > Person.h
main.cpp Person.cpp Person.h

class Person {
public:
    Person(int age); // Converting constructor
private:
    int age;
};

#endif //ADVANCED_PERSON_H
```

Using the Explicit Keyword

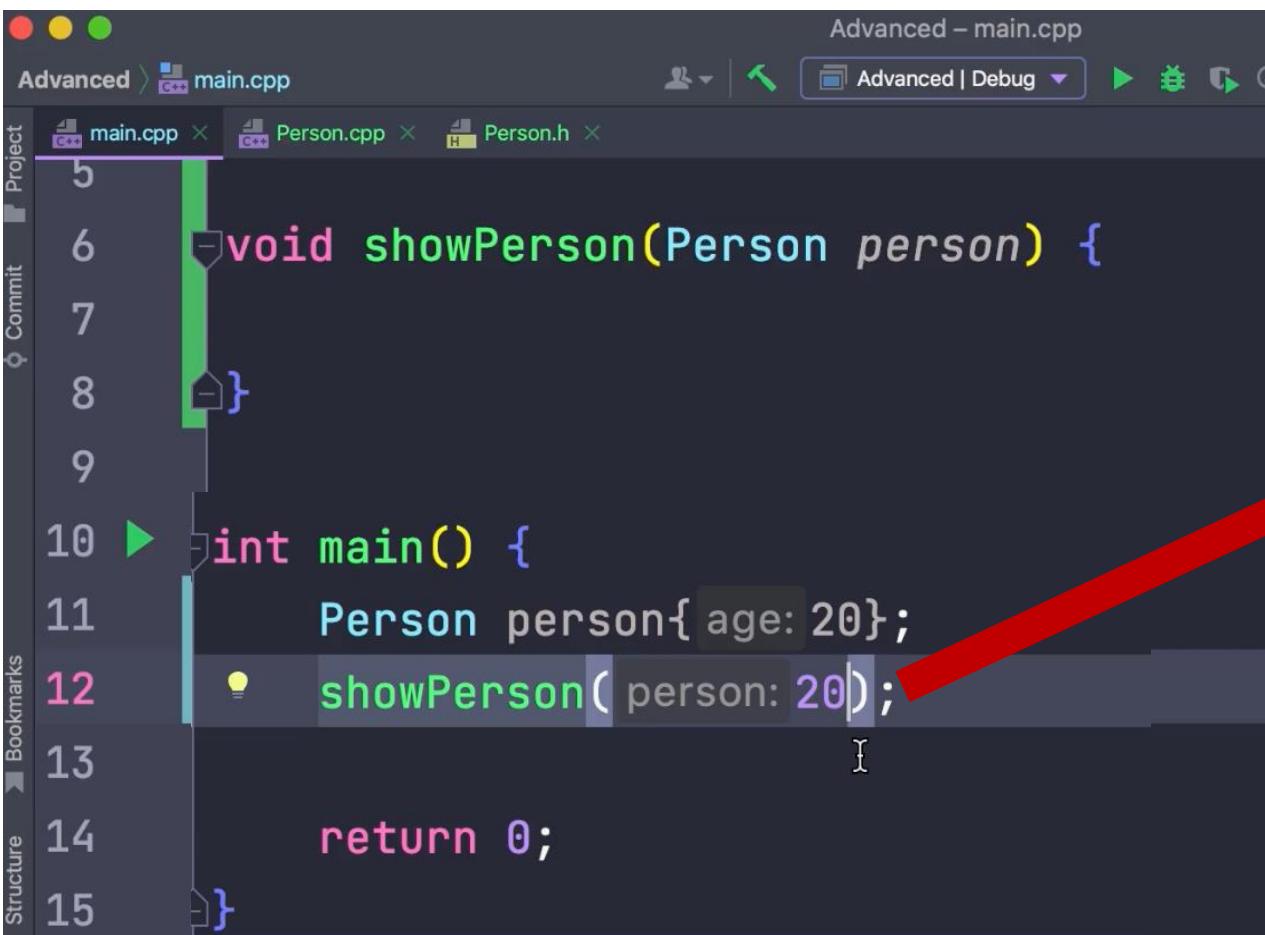
- Sometimes when we declare constructors, we need to use the keyword “explicit”.
- C++ will implicitly create a person object and pass to the function.

The screenshot shows a C++ IDE interface with two tabs: "Advanced" and "main.cpp". The "main.cpp" tab contains the following code:1 #include <iostream>
2 #include "Person.h"
3
4 using namespace std;
5
6 void showPerson(Person person) {
7 }
8
9
10 int main() {
11 Person person{ age: 20 };
12 showPerson(person);
13
14 return 0;
15 }

A red arrow points from the line "showPerson(person);" in the main function to a second screenshot of the code, where the line has been modified.

The screenshot shows the same C++ IDE interface with the "main.cpp" tab selected. The code has been modified to include the "explicit" keyword:int main() {
 Person person{ age: 20 };
 showPerson(person: 20);
 return 0;
}A red arrow points from the original code in the first screenshot to this modified version in the second screenshot.

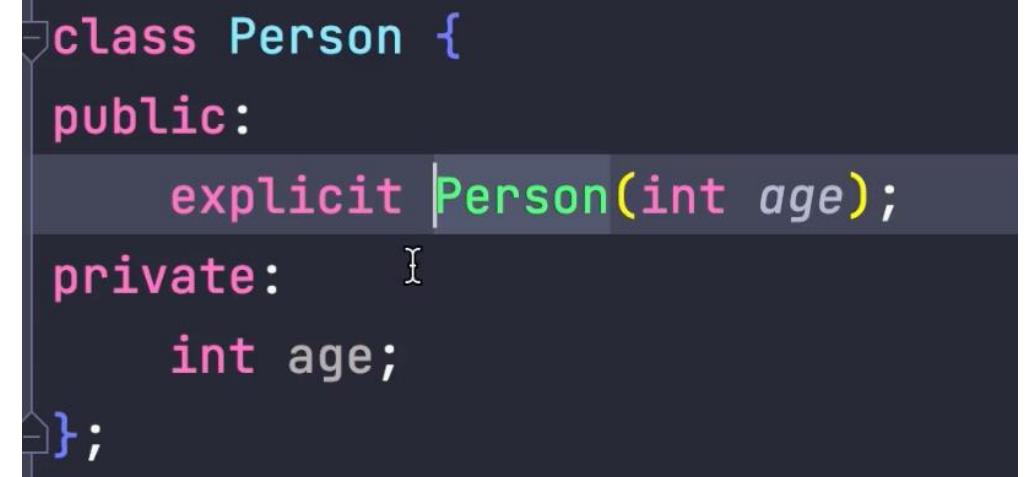
Using the Explicit Keyword



A screenshot of a C++ IDE showing a code editor with the file "main.cpp". The code contains a function `showPerson` and the `main` function. A red arrow points from the warning in the code editor to the corresponding constructor definition in the "Person.h" header file.

```
Advanced – main.cpp
Advanced | Debug
Project main.cpp Person.cpp Person.h
1 void showPerson(Person person) {
2
3 }
4
5
6 int main() {
7     Person person{age: 20};
8     showPerson(person: 20);
9
10    return 0;
11 }
```

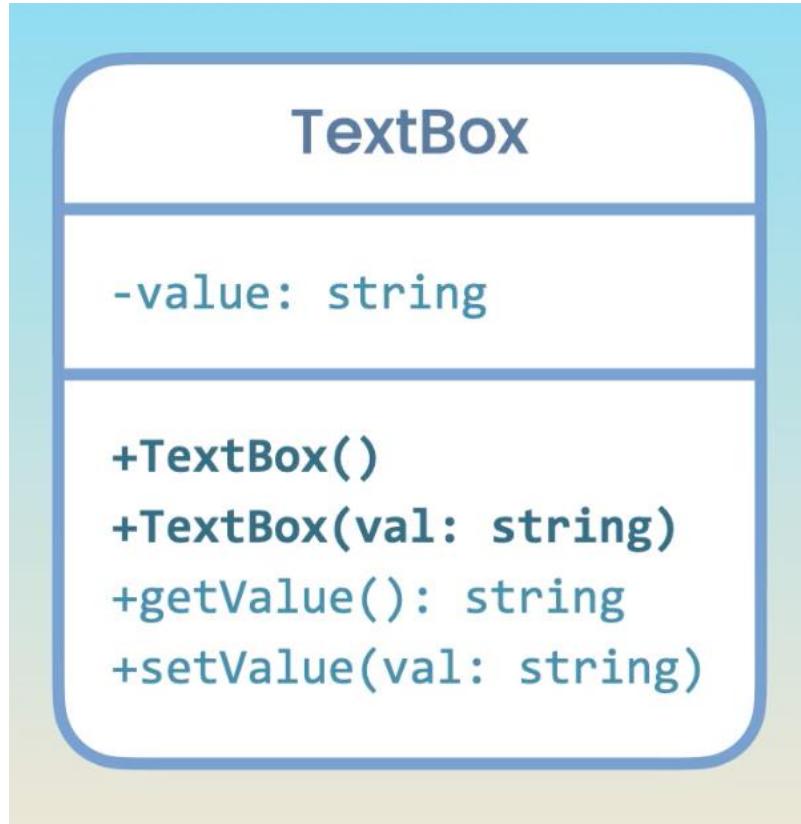
- To prevent this, we use “**explicit**” in the header.



A screenshot of a C++ IDE showing the "Person.h" header file. It defines a class `Person` with a constructor that uses the `explicit` keyword.

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

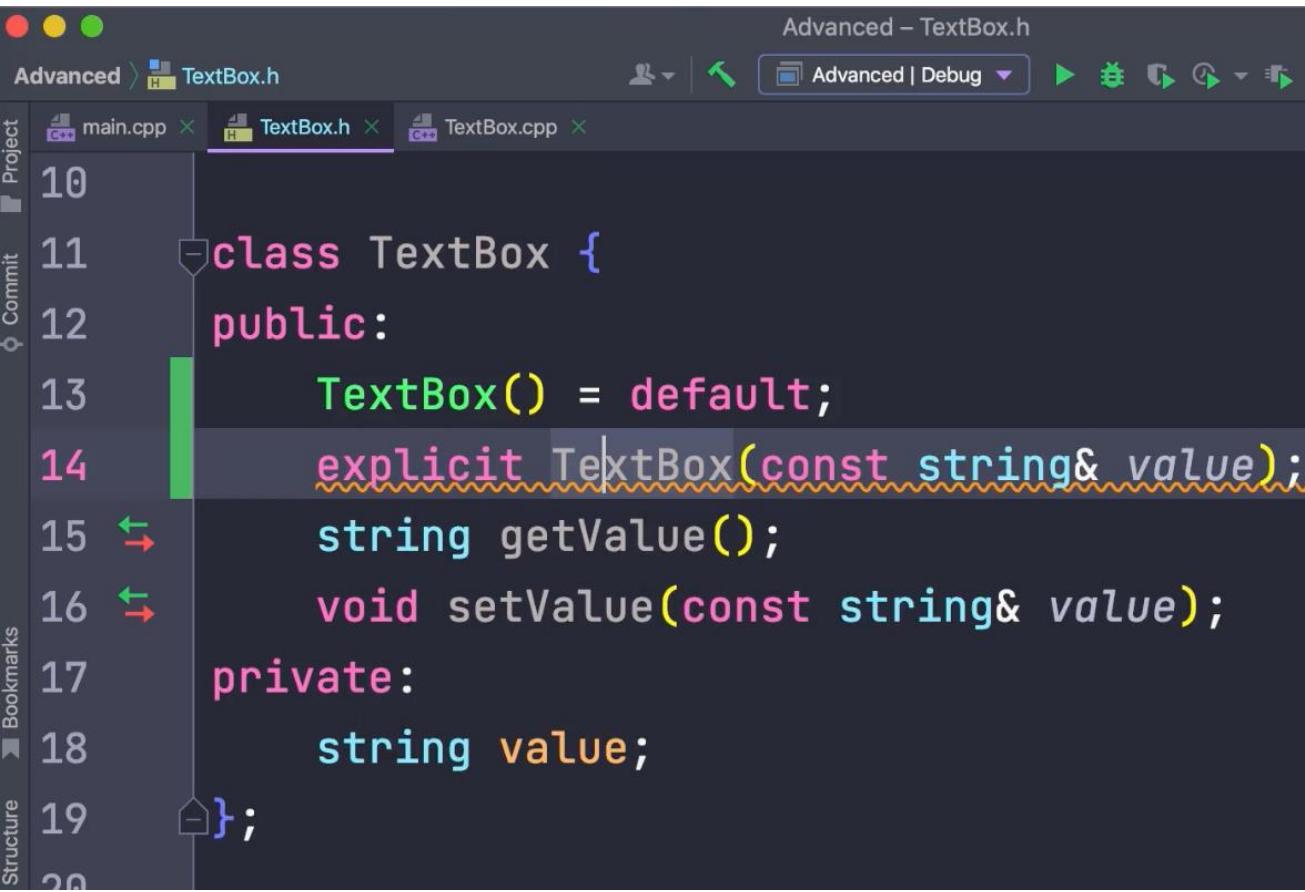
Using the Explicit Keyword



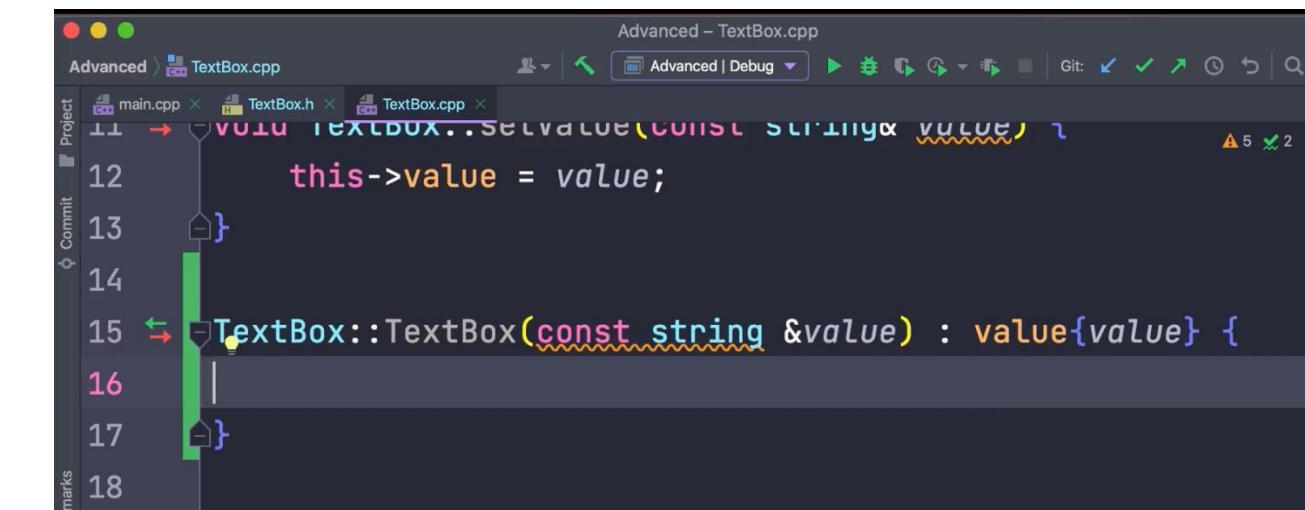
□ Example

Using the Explicit Keyword

Example



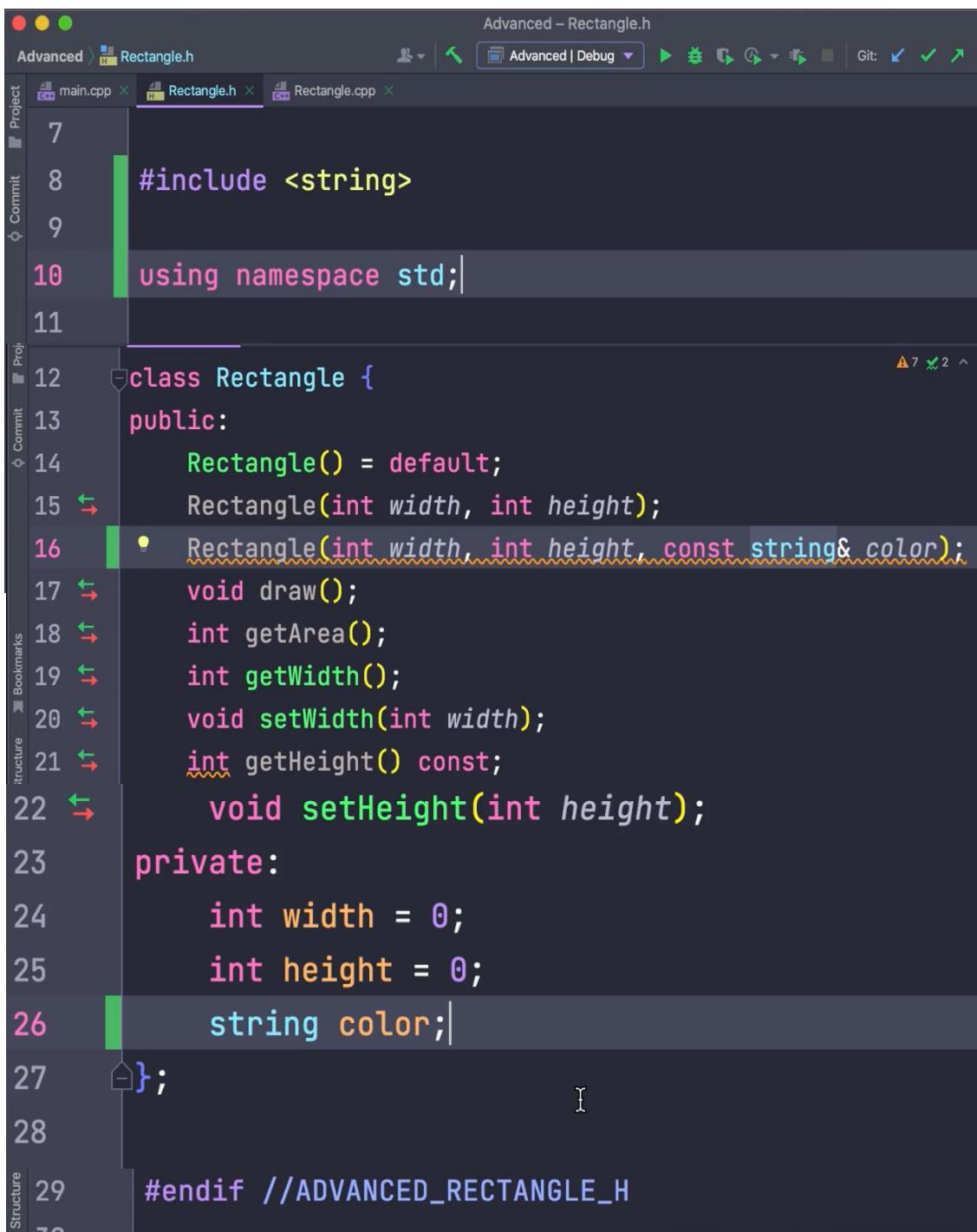
```
Advanced – TextBox.h
Advanced > TextBox.h
Project  main.cpp  TextBox.h  TextBox.cpp
10
11 class TextBox {
12 public:
13     TextBox() = default;
14     explicit TextBox(const string& value);
15     string getValue();
16     void setValue(const string& value);
17 private:
18     string value;
19 }
```



```
Advanced – TextBox.cpp
Advanced > TextBox.cpp
Project  main.cpp  TextBox.h  TextBox.cpp
12     this->value = value;
13 }
14
15 TextBox::TextBox(const string &value) : value{value} {
16
17 }
```

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.



A screenshot of a code editor showing the `Rectangle.h` header file. The file contains the following code:

```
Advanced - Rectangle.h
Advanced Rectangle.h Rectangle.cpp
main.cpp Rectangle.h Rectangle.cpp

7
8 #include <string>
9
10 using namespace std;

11
12 class Rectangle {
13 public:
14     Rectangle() = default;
15     Rectangle(int width, int height);
16     Rectangle(int width, int height, const string& color);
17     void draw();
18     int getArea();
19     int getWidth();
20     void setWidth(int width);
21     int getHeight() const;
22     void setHeight(int height);
23 private:
24     int width = 0;
25     int height = 0;
26     string color;
27 };
28
29 #endif //ADVANCED_RECTANGLE_H
```

The constructor at line 16 is highlighted with a yellow underline. The code editor interface includes a toolbar with icons for file operations, a status bar with Git status, and a sidebar with project navigation and commit history.

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

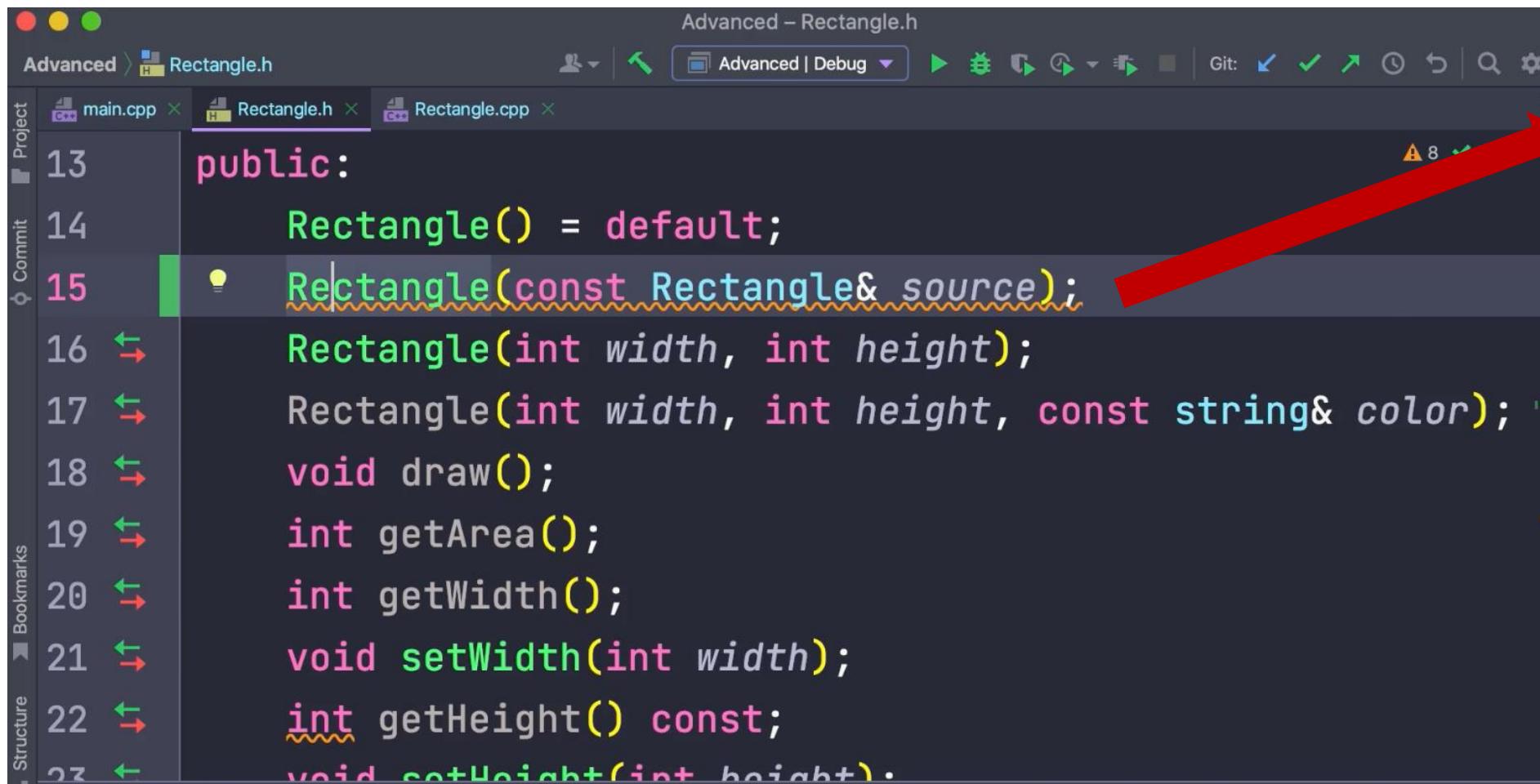


A screenshot of a C++ code editor showing a main() function. The code creates a Rectangle object 'first' with width 10 and height 20, and then creates another Rectangle object 'second' by assignment from 'first'. The code editor has tabs for main.cpp, Rectangle.h, and Rectangle.cpp.

```
5
6 ► int main() {
7     Rectangle first{ width: 10, height: 20};
8     Rectangle second = first;
9
10    return 0;
11 }
12
```

- 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



Advanced – Rectangle.h

Advanced Rectangle.h Rectangle.cpp

```
13 public:  
14     Rectangle() = default;  
15     Rectangle(const Rectangle& source); // Red arrow points here  
16     Rectangle(int width, int height);  
17     Rectangle(int width, int height, const string& color);  
18     void draw();  
19     int getArea();  
20     int getWidth();  
21     void setWidth(int width);  
22     int getHeight() const;  
23     void setHeight(int height);
```

□ Definition

The Copy Constructor

□ Implementation

A screenshot of a code editor showing the implementation of the copy constructor in `Rectangle.cpp`. The code is as follows:

```
50
51 Rectangle::Rectangle(const Rectangle& source) {
52     cout << "Rectangle copied" << endl;
53     this->width = source.width;
54     this->height = source.height;
55     this->color = source.color;
56 }
```

The file is part of a project named "Advanced" with files `main.cpp`, `Rectangle.h`, and `Rectangle.cpp`.

A screenshot of a code editor showing the main function in `main.cpp`. The code is as follows:

```
5
6 int main() {
7     Rectangle first{ width: 10, height: 20 };
8     Rectangle second = first;
9
10    return 0;
11 }
```

The file is part of a project named "Advanced" with files `main.cpp`, `Rectangle.h`, and `Rectangle.cpp`.

□ main function

The Copy Constructor

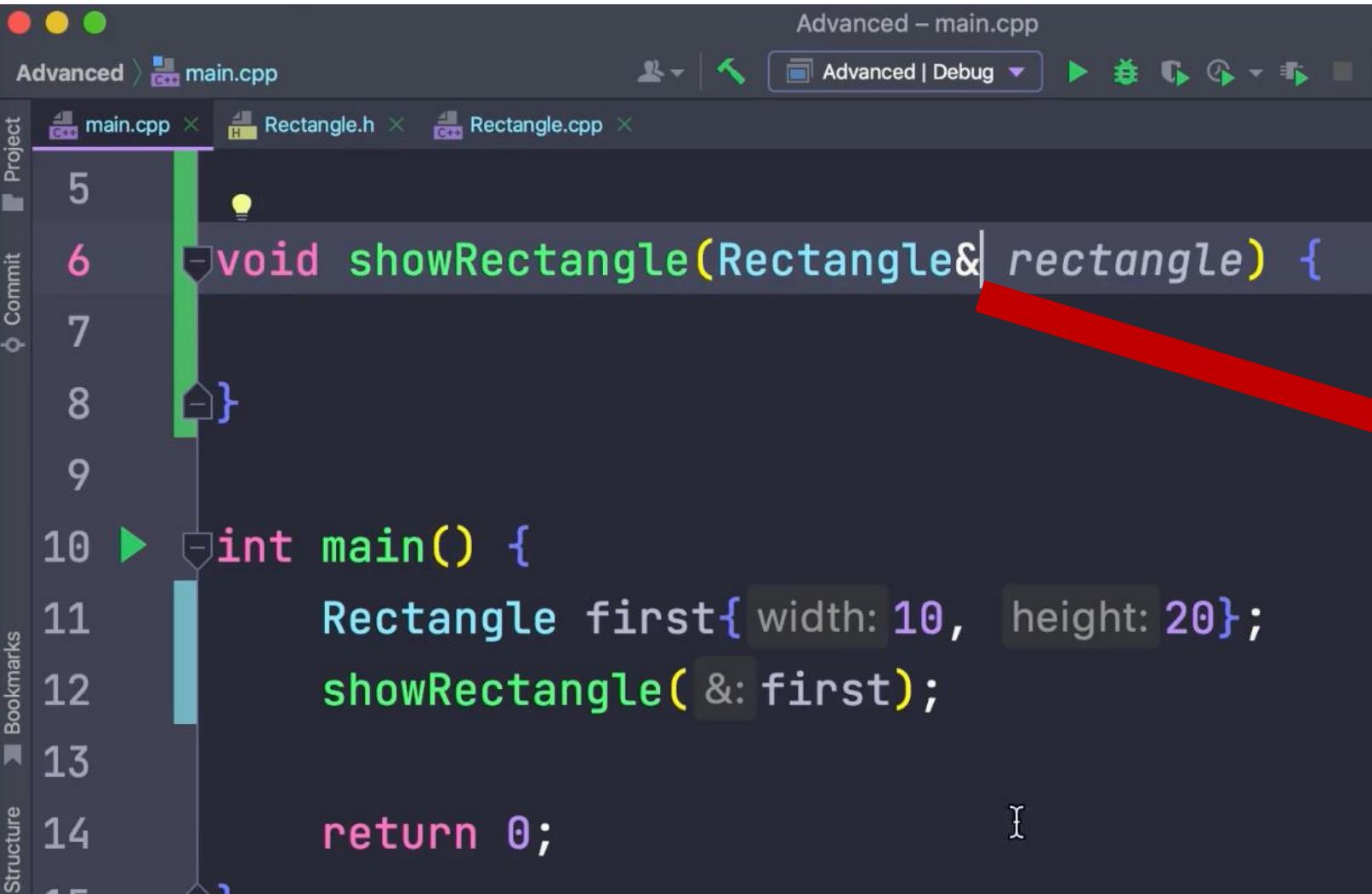
- ❑ Objects are also copied when they are passed to functions as parameters.

A screenshot of a C++ IDE interface. The title bar says "Advanced – main.cpp". The project navigation bar shows "Advanced > main.cpp". Below the title bar, there are tabs for "main.cpp", "Rectangle.h", and "Rectangle.cpp". The main code editor area contains the following C++ code:

```
5
6 void showRectangle(Rectangle rectangle) {
7
8 }
9
10 int main() {
11     Rectangle first{ width: 10, height: 20 };
12     showRectangle(first);
13
14     return 0;
15 }
```

The code defines a function `showRectangle` that takes a `Rectangle` object as a parameter. In the `main` function, a `Rectangle` object named `first` is created with width 10 and height 20, and then passed to the `showRectangle` function. The code uses brace initialization for the `first` object.

The Copy Constructor



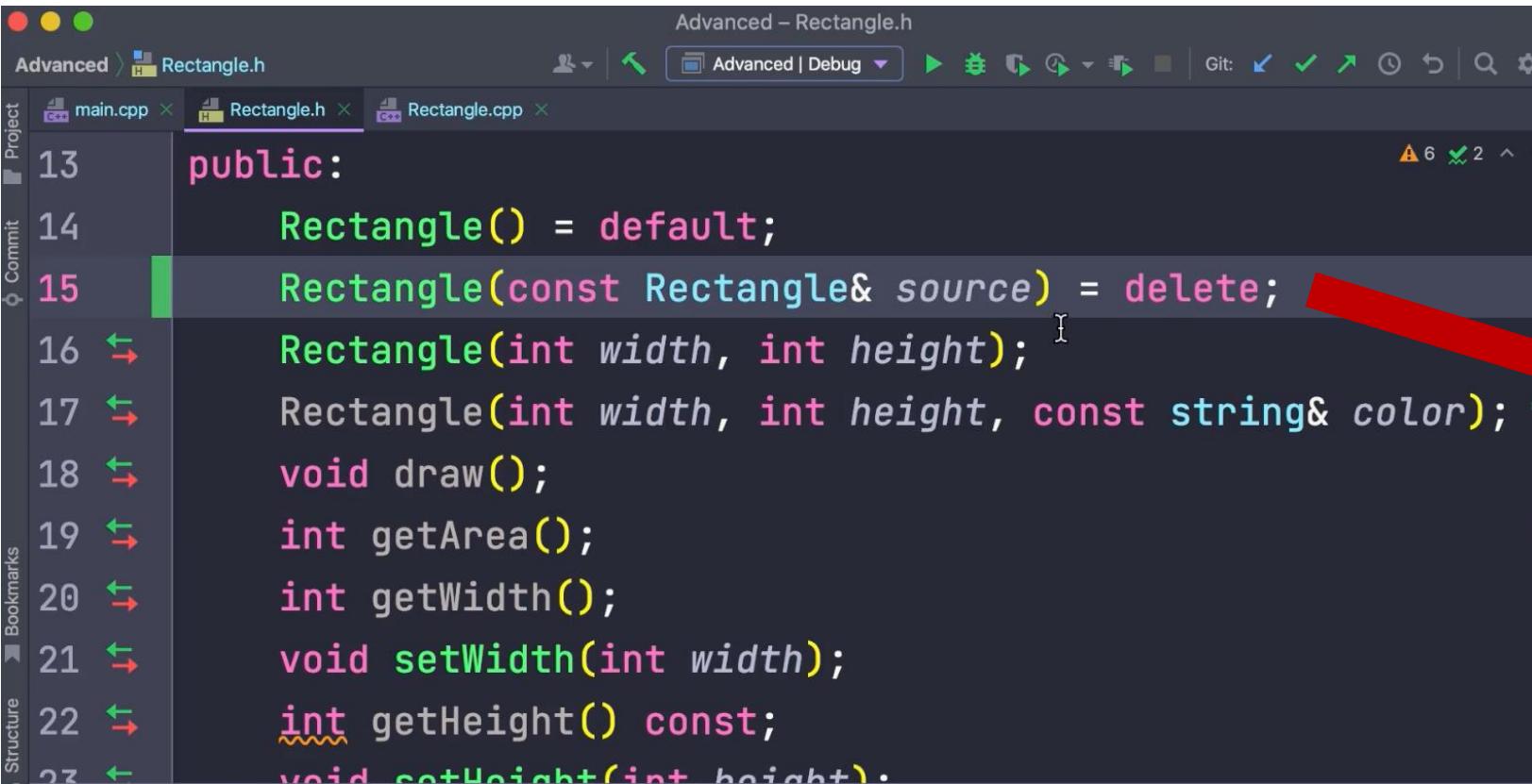
A screenshot of a C++ IDE interface. The title bar says "Advanced - main.cpp". The project navigation bar shows "main.cpp", "Rectangle.h", and "Rectangle.cpp". The code editor displays the following C++ code:

```
Advanced - main.cpp
Advanced > main.cpp
Project Rectangle.h Rectangle.cpp
1
2
3
4
5
6 void showRectangle(Rectangle& rectangle) {
7
8 }
9
10 int main() {
11     Rectangle first{ width: 10, height: 20 };
12     showRectangle( &first );
13
14     return 0;
15 }
```

A red arrow points from the explanatory text below to the line "showRectangle(&first);".

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

The Copy Constructor



A screenshot of a code editor showing the file `Advanced - Rectangle.h`. The code defines a class `Rectangle` with the following members:

```
public:
    Rectangle() = default;
    Rectangle(const Rectangle& source) = delete;
    Rectangle(int width, int height);
    Rectangle(int width, int height, const string& color);
    void draw();
    int getArea();
    int getWidth();
    void setWidth(int width);
    int getHeight() const;
    void setHeight(int height);
```

A red arrow points from the text "Used to prevent objects to be copied by value." to the deleted copy constructor line.

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

- Used to prevent objects to be copied by value.

The Copy Constructor

- ❑ Used to prevent objects to be copied by value.



```
main.cpp Rectangle.h Rectangle.cpp
50
51     -> Rectangle::Rectangle(const Rectangle& source) {
52         cout << "Rectangle copied" << endl;
53         this->width = source.width;
54         this->height = source.height;
55         this->color = source.color;
56     }
```

- ❑ 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.

A screenshot of a code editor showing a C++ file named `main.cpp`. The code contains a `showRectangle` function and a `main` function. The `showRectangle` function takes a `Rectangle` object by value. In the `main` function, a `Rectangle` object named `first` is created with width 10 and height 20, and then passed to `showRectangle`. A tooltip appears over the call to `showRectangle(first)`, stating: "Call to deleted constructor of 'Rectangle' 'Rectangle' has been explicitly marked deleted here passing argument to parameter 'rectangle' here". Below the tooltip, there are options to "Make constructor 'Rectangle' default" or "More actions...". The code editor interface shows tabs for `main.cpp`, `Rectangle.h`, and `Rectangle.cpp`. The bottom of the screen shows standard IDE navigation and search bars.

```
5
6 void showRectangle(Rectangle rectangle) {
7
8 }
9
10 int main() {
11     Rectangle first{ width: 10, height: 20 };
12     showRectangle(first);
13
14     return 0;
15 }
```

Call to deleted constructor of 'Rectangle'
'Rectangle' has been explicitly marked deleted here
passing argument to parameter 'rectangle' here

Make constructor 'Rectangle' default More actions...

Rectangle first {10, 20}

The Copy Constructor

A screenshot of a code editor showing a C++ file named `main.cpp`. The code contains the following:

```
10 > int main() {
11     Rectangle first{ width: 10, height: 20};
12     Rectangle second = first;
13
14     return 0;
15 }
```

The line `Rectangle second = first;` has a yellow warning icon next to it. A tooltip appears over the line, displaying the message: "Call to deleted constructor of 'Rectangle' 'Rectangle' has been explicitly marked deleted here". Below the tooltip, there are options: "Make constructor 'Rectangle' default" and "More actions...". A red arrow points from the text block below to this tooltip.

- ❑ Error message when we try to copy an 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.

A screenshot of a code editor showing a C++ class definition. The class has three constructors: a default constructor, a constructor taking width and height, and a constructor taking width, height, and color. It also includes a destructor named `~Rectangle()`. The code editor interface shows tabs for main.cpp, Rectangle.h, and Rectangle.cpp, with Rectangle.h currently selected. A red arrow points from the text "destructors do not have a return type." to the destructor declaration in the code.

```
11
12 class Rectangle {
13     public:
14         Rectangle() = default;
15         Rectangle(int width, int height);
16         Rectangle(int width, int height, const string& color);
17         ~Rectangle(); ->
18         void draw();
19         int getArea();
20         int getWidth();
21         void setWidth(int width);
```

The Destructor

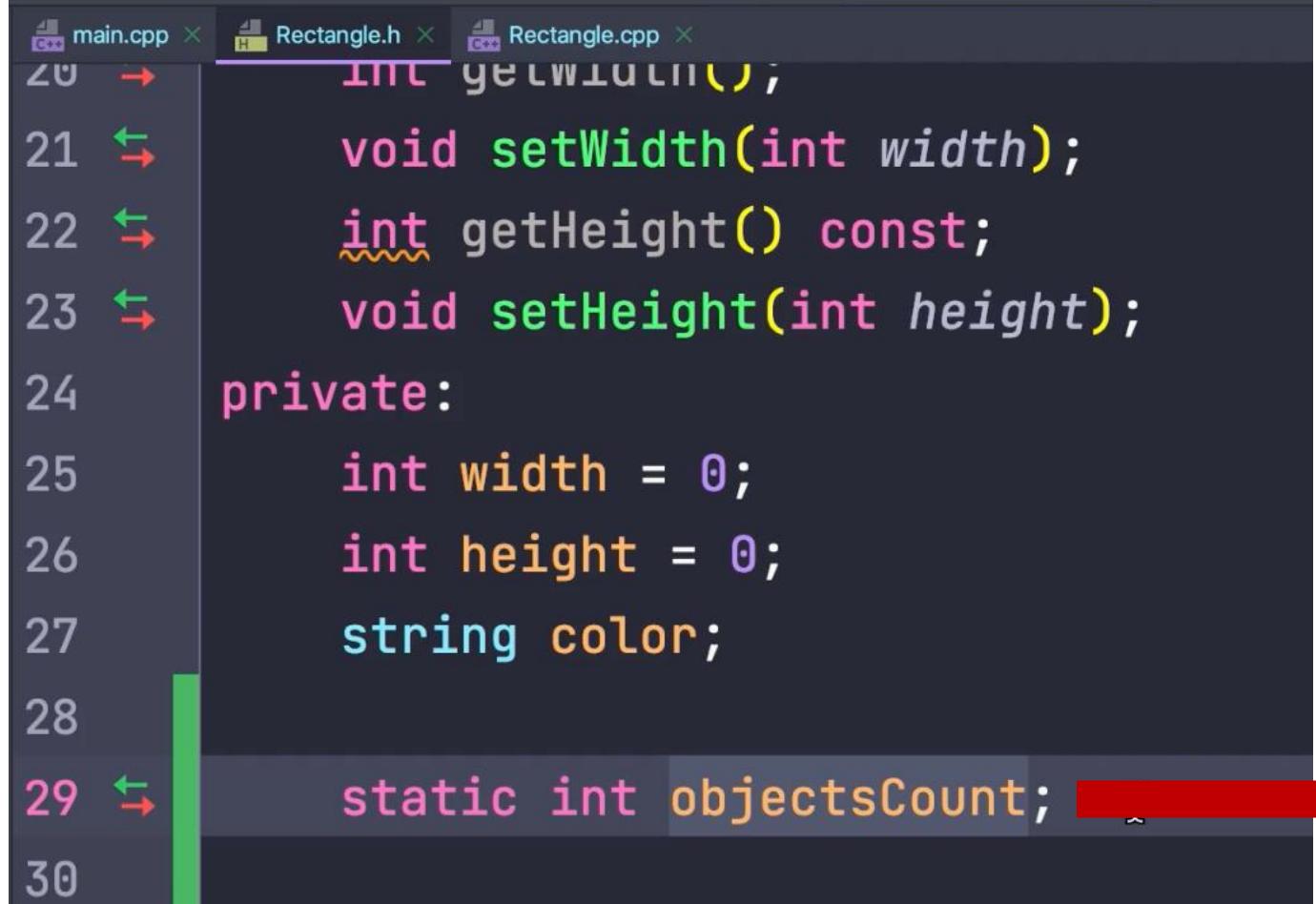
- Implementation

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

```
47     cout << "CONSTRUCTING A RECTANGLE WITH"
48     this->color = color;
49 }
50
51 ~Rectangle() {
52     cout << "Destructor called" << endl;
53 }
54
```

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

Static Members



```
20     int getWidth();  
21     void setWidth(int width);  
22     int getHeight() const;  
23     void setHeight(int height);  
24 private:  
25     int width = 0;  
26     int height = 0;  
27     string color;  
28  
29     static int objectsCount; ~~~~~  
30
```

- 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

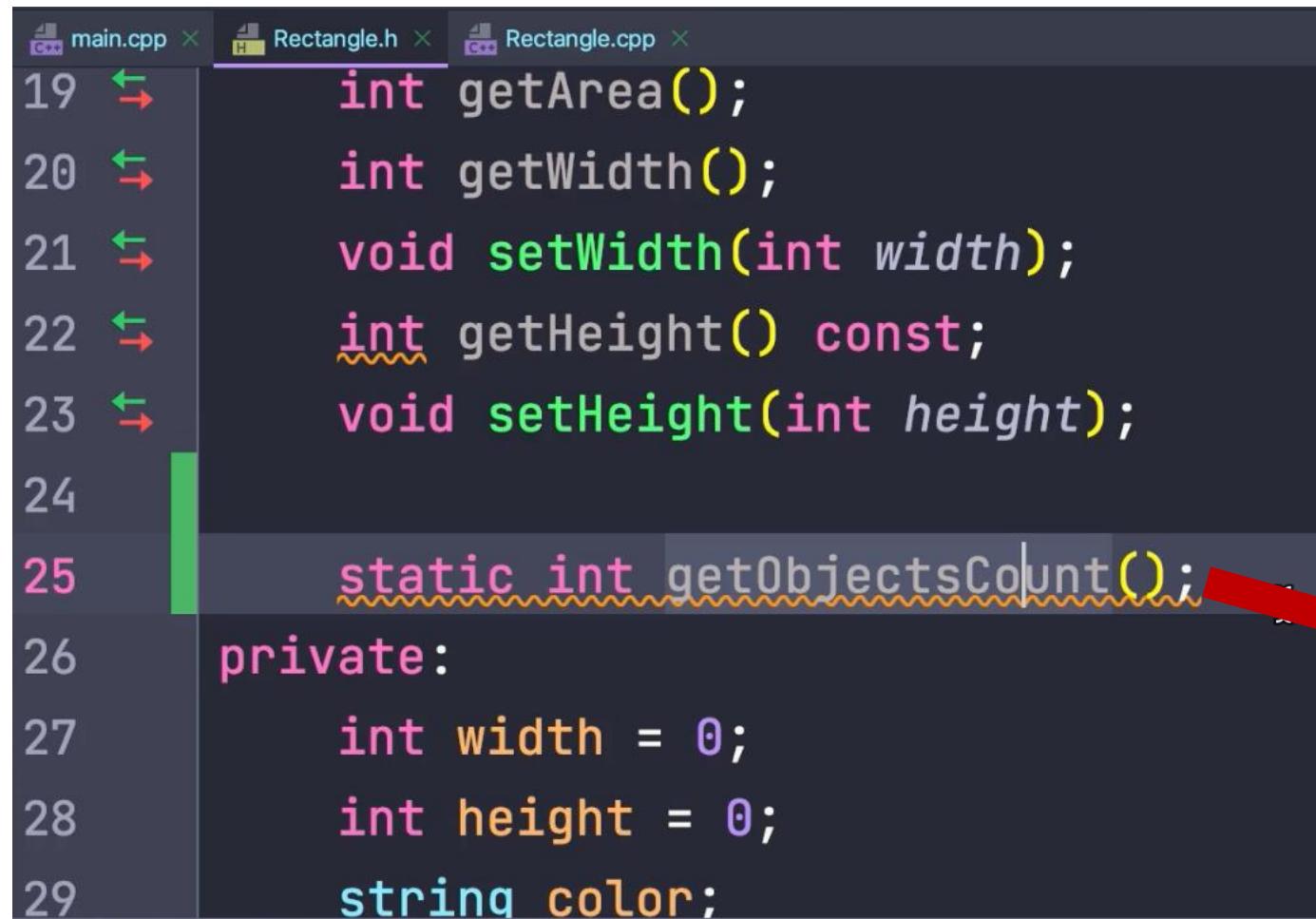


A screenshot of a code editor showing the `Rectangle.cpp` file. The code contains a destructor and a static member variable. A red arrow points from the static member declaration to the second bullet point.

```
50  
51     ~Rectangle() {  
52         cout << "Destructor called" << endl;  
53     }  
54  
55     int Rectangle::objectsCount = 0;
```

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

Static Members

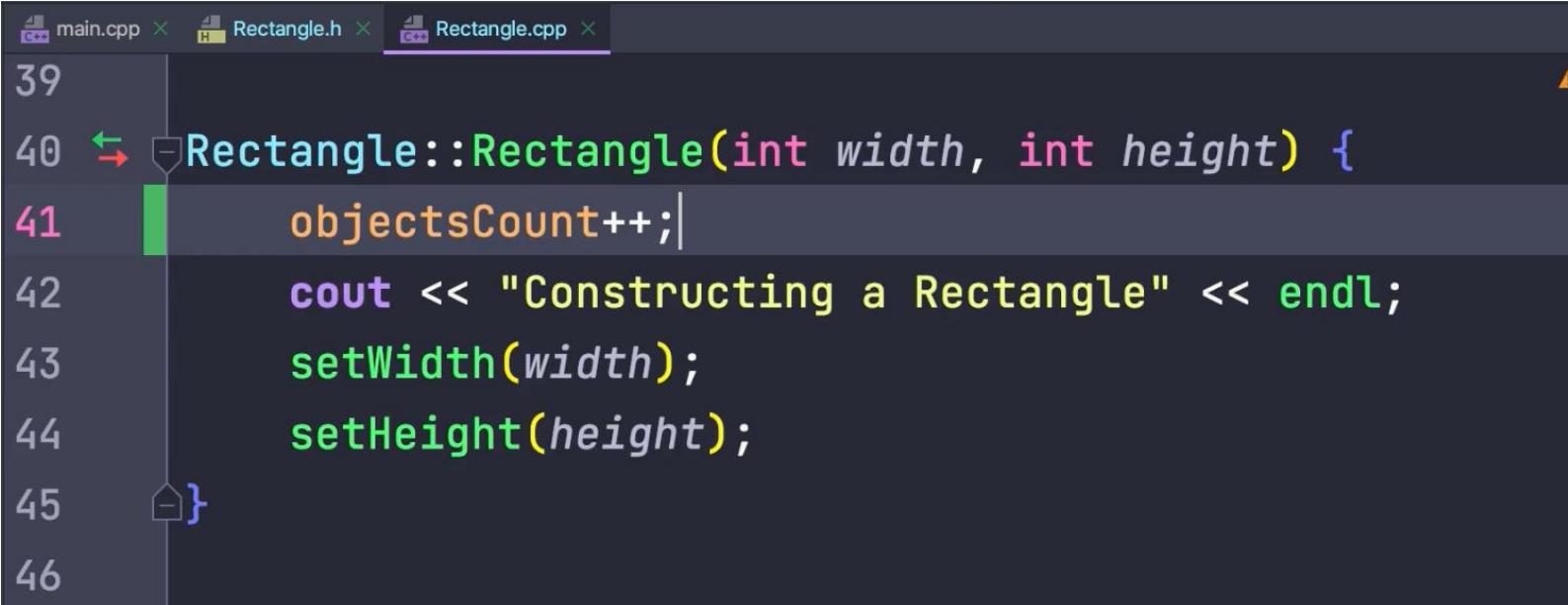


A screenshot of a code editor showing the `Rectangle.h` header file. The code defines a class `Rectangle` with several methods: `getArea()`, `getWidth()`, `setWidth(int width)`, `getHeight() const`, `setHeight(int height)`, and a static member function `getObjectsCount()`. A red arrow points from the `getObjectsCount()` line to the first bullet point below.

```
19     int getArea();
20     int getWidth();
21     void setWidth(int width);
22     int getHeight() const;
23     void setHeight(int height);
24
25     static int getObjectsCount();
26
27     private:
28         int width = 0;
29         int height = 0;
30         string color;
```

- 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



The screenshot shows a code editor with three tabs at the top: main.cpp, Rectangle.h, and Rectangle.cpp. The Rectangle.cpp tab is active. The code in the editor is:

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

Line 40 shows a constructor for the Rectangle class. Line 41 increments a static member variable called objectsCount. Lines 42 through 44 call two member functions, setWidth and setHeight, with their respective parameters.

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

Static Members

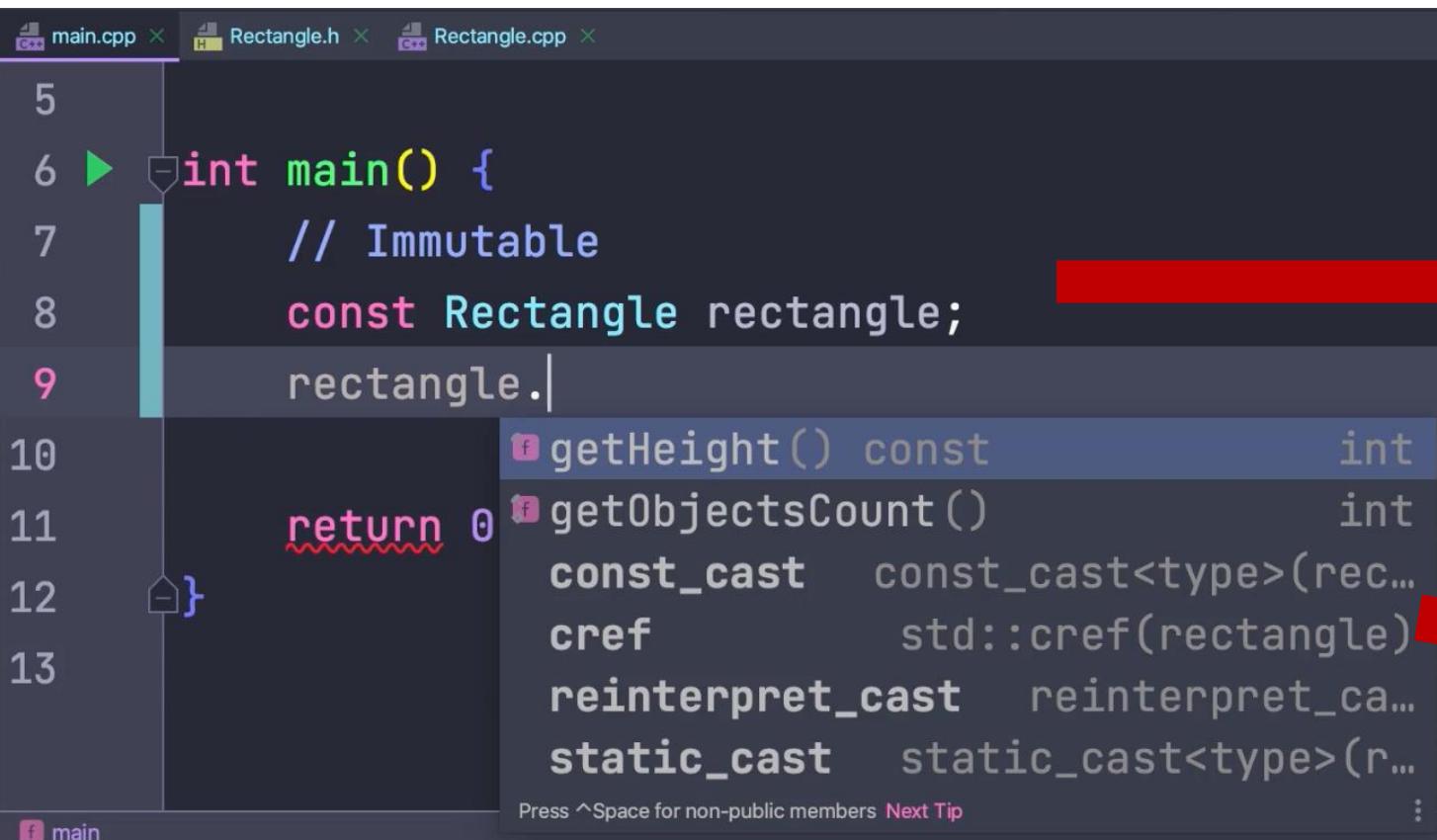
- Static member functions can only access static members.
- Static member function implementation.
- The main function.

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

```
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
```

A red arrow points from the line "cout << Rectangle::getObjectsCount()" to the colon in "Rectangle::getObjectsCount()", with the text "Scope resolution operator" written along the arrow.

Constant Objects and Functions



A screenshot of a C++ IDE showing a code editor with the following code:

```
5
6 int main() {
7     // Immutable
8     const Rectangle rectangle;
9     rectangle.|
```

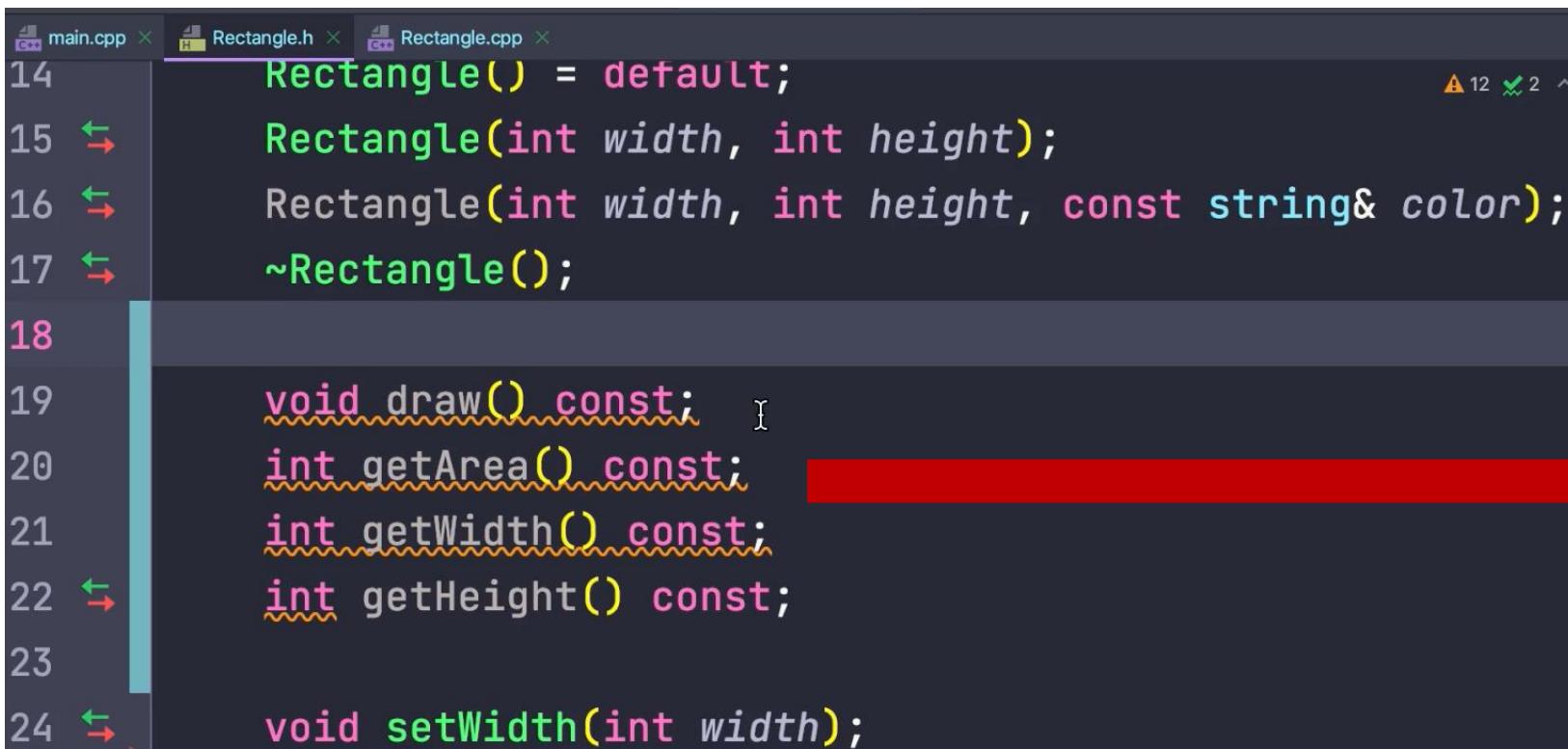
The cursor is at the end of the line 'rectangle.|'. A code completion dropdown menu is open, listing the following members:

- f getHeight() const int
- f getObjectsCount() int
- const_cast const_cast<type>(rec...
- cref std::cref(rectangle)
- reinterpret_cast reinterpret_ca...
- static_cast static_cast<type>(r...

Red arrows point from the text to the right of the list to the corresponding bullet points in the list below.

- 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



```
main.cpp x Rectangle.h x Rectangle.cpp x
14     Rectangle() = default;
15     → Rectangle(int width, int height);
16     → Rectangle(int width, int height, const string& color);
17     ~Rectangle();
18
19     void draw() const;
20     int getArea() const;
21     int getWidth() const;
22     int getHeight() const;
23
24     void setWidth(int width);
```

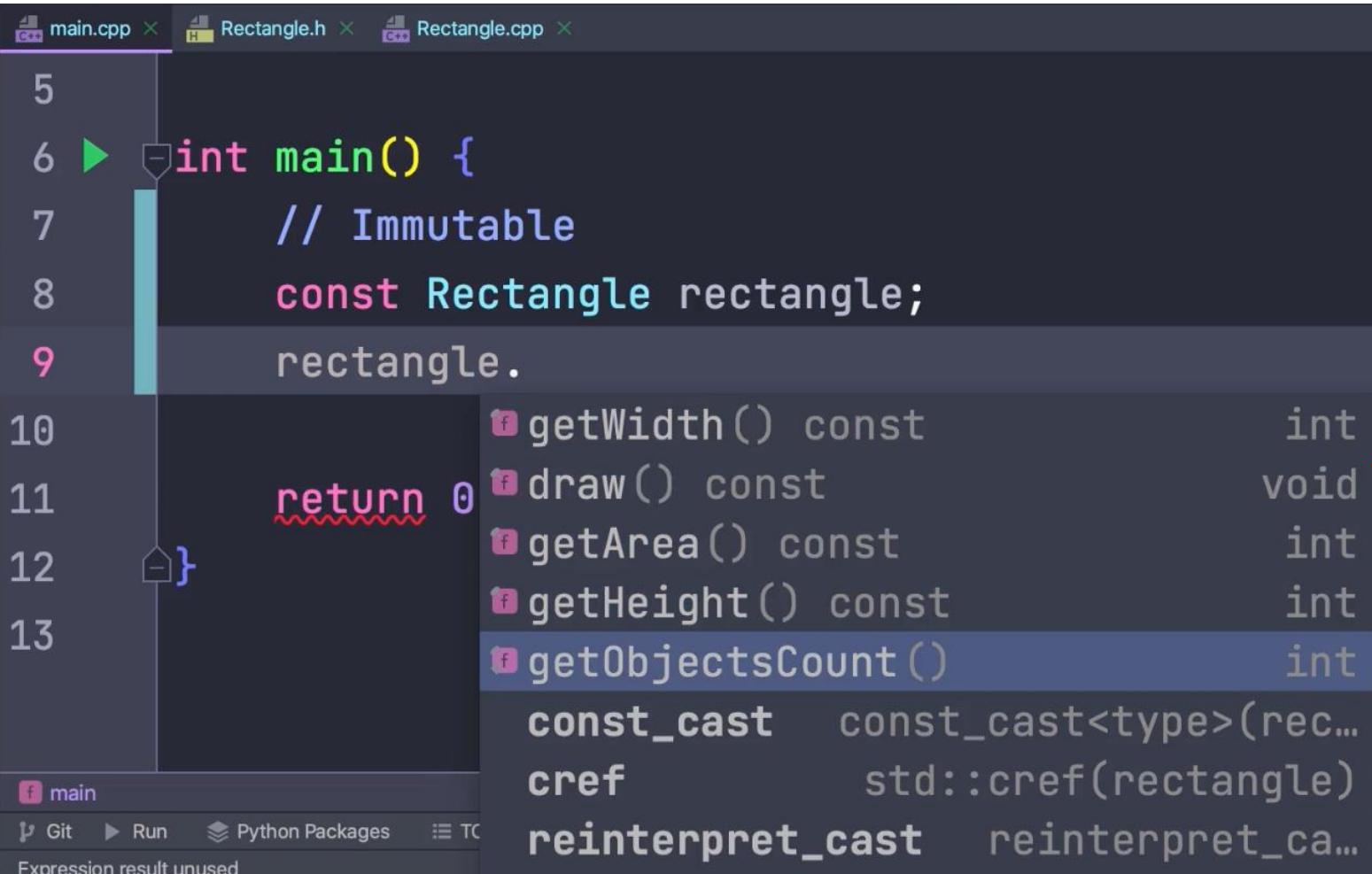
- Functions that do not change the status of an object should be declared as constant.

Constant Objects and Functions

```
9
10 void Rectangle::draw() const {
11     cout << "Drawing a rectangle!" << endl;
12     cout << "Dimensions: " << width << ", " << height << endl;
13 }
14
15 int Rectangle::getArea() const {
16     return width * height;
17 }
18
19 int Rectangle::getWidth() const {
20     return width;
21 }
22
23 int Rectangle::getHeight() const {
24     return height;
25 }
```

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

Constant Objects and Functions



A screenshot of a C++ IDE interface. The main window shows a code editor with the following C++ code:

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

The cursor is at the end of the line 'rectangle.', and a code completion dropdown menu is open. The menu lists several member functions of the `const Rectangle` type, each with its name, return type, and parameter types:

Function	Return Type	Parameters
<code>getWidth() const</code>	<code>int</code>	
<code>draw() const</code>	<code>void</code>	
<code>getArea() const</code>	<code>int</code>	
<code>getHeight() const</code>	<code>int</code>	
<code>getObjectsCount()</code>	<code>int</code>	
<code>const_cast<type>(rec...)</code>		
<code>cref</code>		<code>std::cref(rectangle)</code>
<code>reinterpret_cast<type>(ca...</code>		

The IDE's status bar at the bottom shows: Expression result unused.

- All accessible constant functions in the main file.

Constant Objects and Functions

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

```
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.
- 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



A screenshot of a code editor showing a C++ file named `main.cpp`. The code includes headers for `Rectangle.h`, `<iostream>`, and `<memory>`. It uses `unique_ptr` from the `<memory>` header to manage a pointer to a `Rectangle` object. The `make_unique` function is used to create the object with width 10 and height 20. The `draw` method is called on the rectangle.

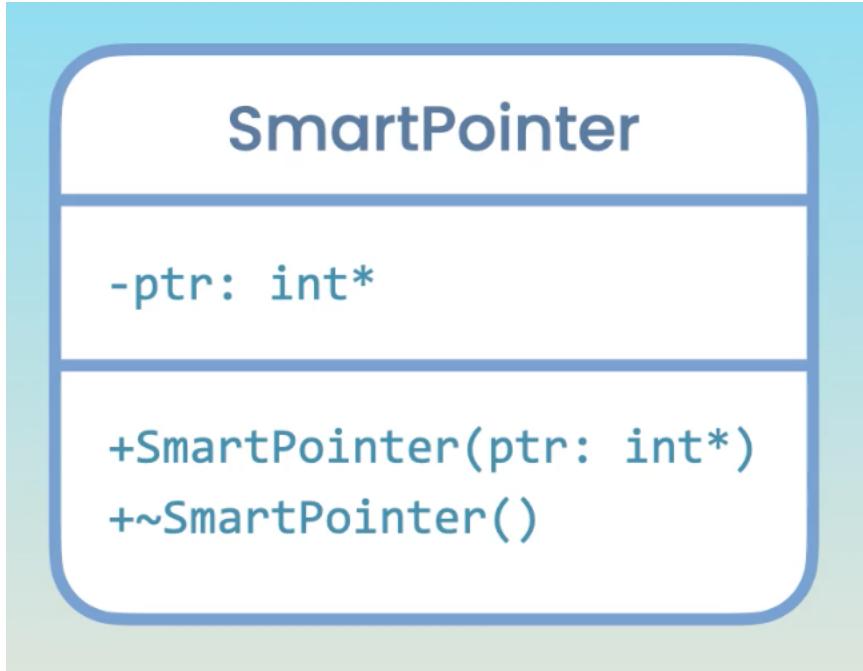
```
#include "Rectangle.h"
#include <iostream>
#include <memory>

using namespace std;

int main() {
    auto rectangle : unique_ptr<Rectangle> = make_unique<Rectangle>( width: 10, height: 20 );
    rectangle->draw();
    return 0;
}
```

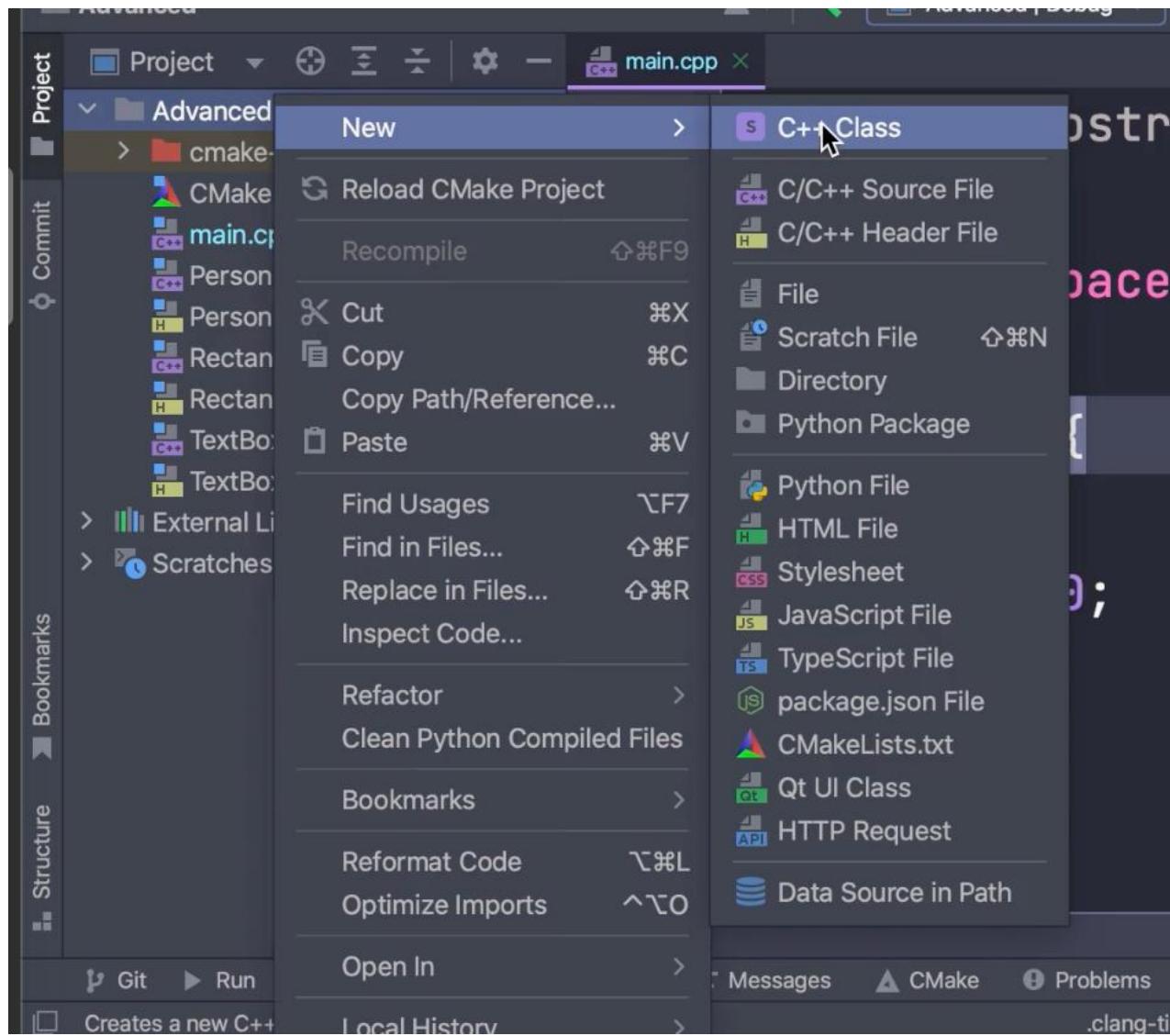
- **Dangling pointer:** A pointer that is not set to null after its use and pointing some random memory address.
- If we reset the pointer but do not free up the memory, we will have a **memory leak**.
- Not to deal with dangling pointers and memory leak, in modern C++, we use **smart pointers**.
- Everything 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.

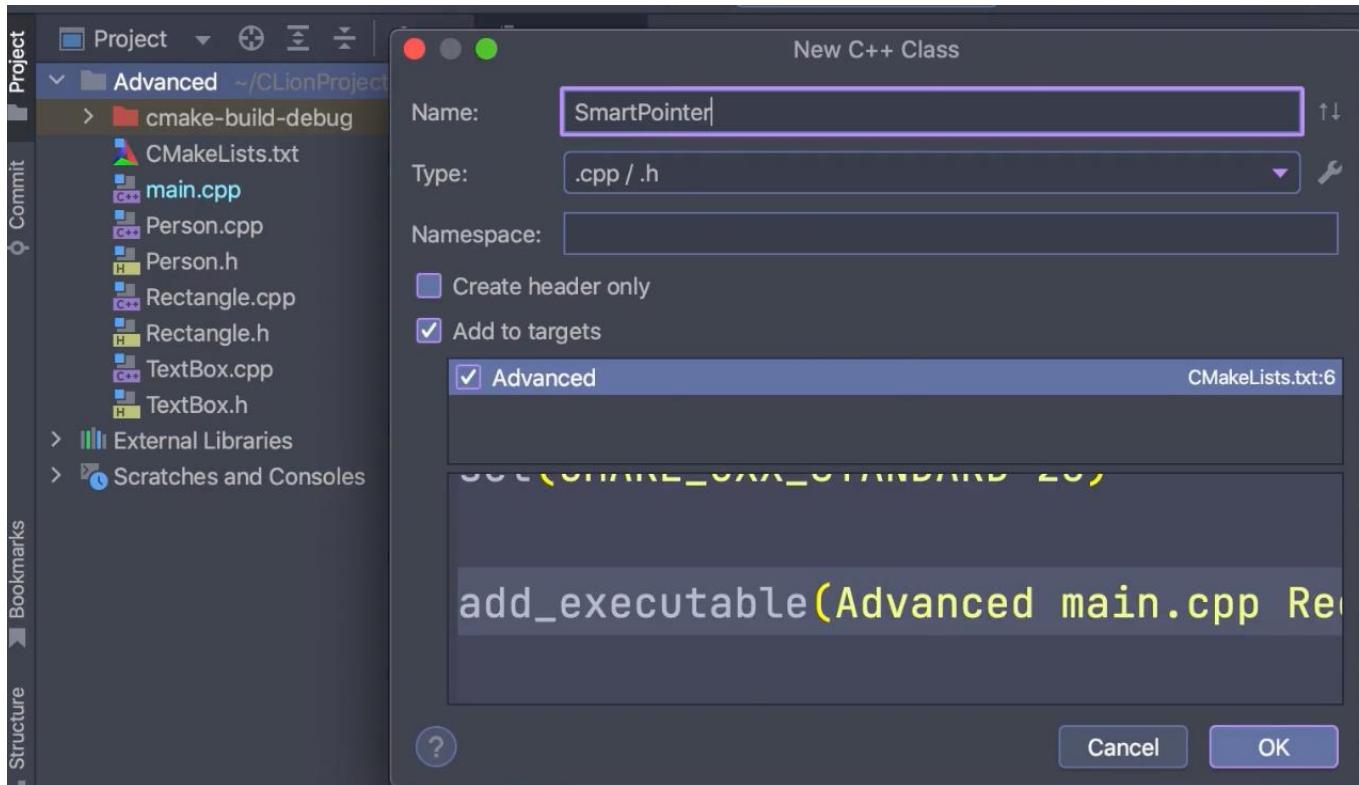
Pointer to Objects



❑ **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

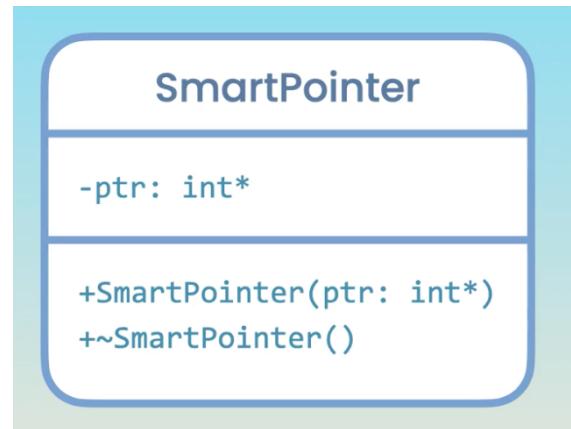
```
8  
9     class SmartPointer {  
10    public:  
11        explicit SmartPointer(int* ptr);  
12        ~SmartPointer();  
13    private:  
14        int* ptr;  
15    };
```

```
3 //  
4  
5 #include "SmartPointer.h"  
6  
7 SmartPointer::SmartPointer(int *ptr) : ptr{ptr} {}  
8  
9 }  
10  
11 SmartPointer::~SmartPointer() {  
12     delete ptr;  
13     ptr = nullptr;  
14 }
```

□ Declaration

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

□ Definition



Pointer to Objects

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

```
1 #include <iostream>
2 #include "SmartPointer.h"
3
4 using namespace std;
5
6 int main() {
7     SmartPointer ptr{ptr: new int};| Replace
8
9     return 0;
10 }
```

Replace

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

☐ Main function.

Array of Objects

```
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;  
}
```

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

Array of Objects

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

```
Rectangle.h
```

```
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;
```

```
Rectangle.cpp
```

```
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.

```
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;
}
```

Array of Objects

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

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

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

return 0;
}
```

Operator Overloading

- Equality operator
- Comparison operators
- Spaceship operator
- Arithmetic operators
- Subscript operator
- Assignment operator

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

Overloading the Equality Operator

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

```
C++ Length.cpp x Length.h x C++ main.cpp x
5
6
7 Length::Length(int value) : value(value) {}
8
9 bool Length::operator==(const Length &other) const {
10     return value == other.value;
11 }
```

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

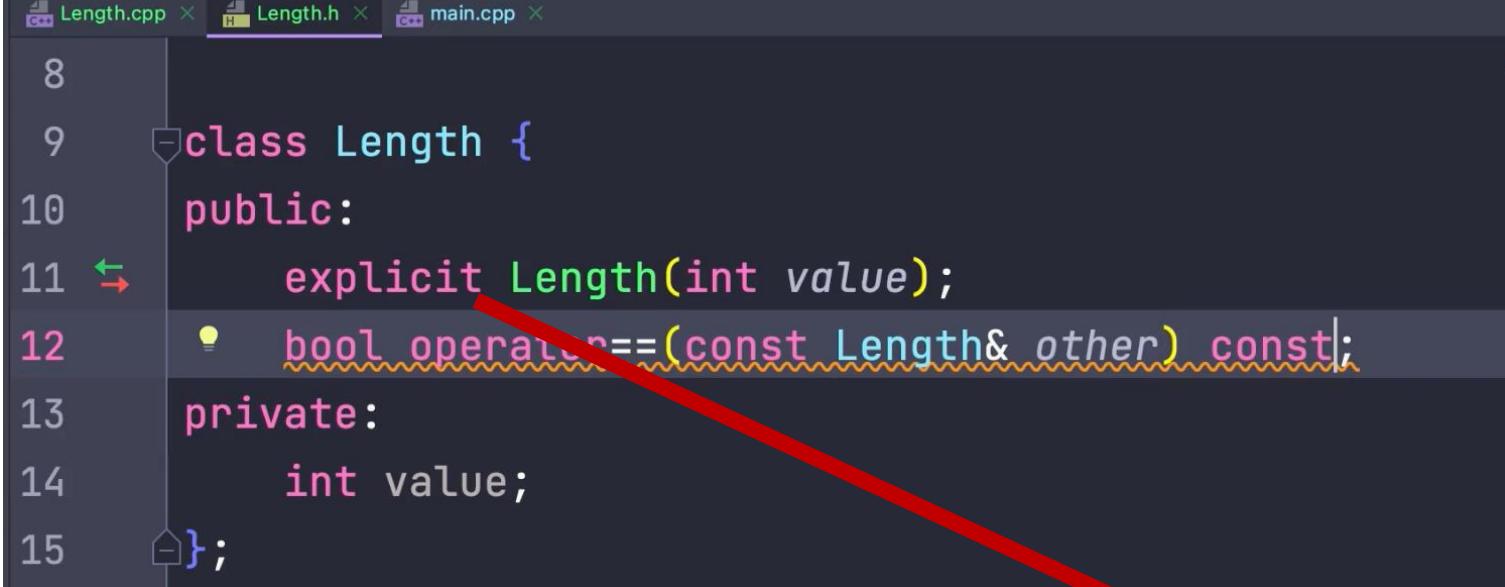
Overloading the Equality Operator

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

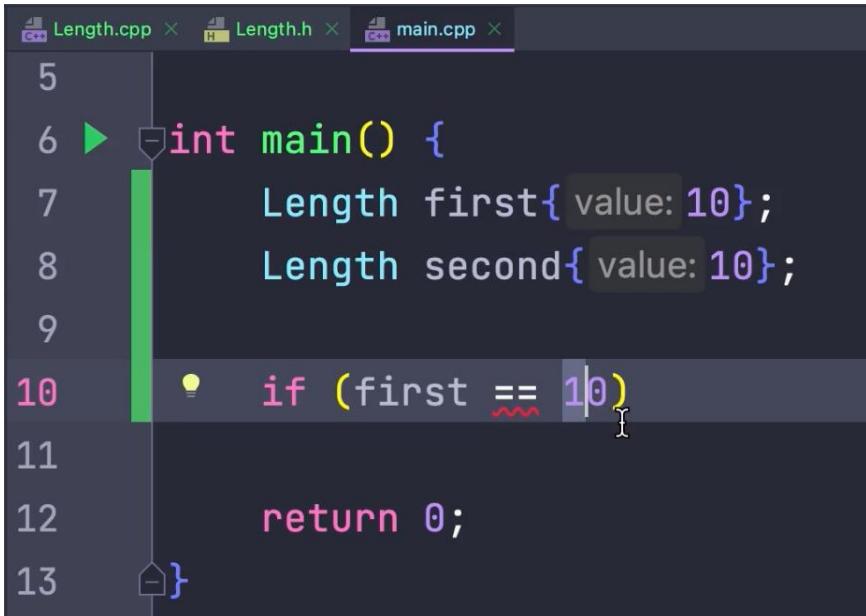
```
C++ Length.cpp x Length.h x C++ main.cpp x
5
6
7 Length::Length(int value) : value(value) {}
8
9 bool Length::operator==(const Length &other) const {
10     return value == other.value;
11 }
```

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

Overloading the Equality Operator



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



```
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

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

```
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     }
```

□ Solution: Add another overload.

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

Overloading the Equality Operator

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

    if (first != 10)
        return 0;
}
```

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

Overloading the Equality Operator

```
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 };
```

```
5   
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.h
Length.cpp
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 };
22
23 bool Length::operator<(const Length &other) const {
24     return value < other.value;
25 }
26
27 bool Length::operator>(const Length &other) const {
28     return value > other.value;
29 }
```

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

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

```
Length.cpp x Length.h x main.cpp x
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     }

```

Overloading the Comparison Operator

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

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

```
int main() {  
    int x = 10;  
    int y = 20;  
  
    if (x < y) {}  
    else if (x > y) {}  
    else {}  
  
    return 0;  
}
```

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

```
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

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

```
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

```
int main() {  
    int x = 10;  
    int y = 20;  
  
    if (x < y) {}  
    else if (x > y) {}  
    else {}  
  
    return 0;  
}
```

Overloading the Stream Insertion Operator

```
main.cpp x Length.h x Length.cpp x
5 #ifndef ADVANCED_LENGTH_H
6 #define ADVANCED_LENGTH_H
7
8 #include <compare>
9 #include <iostream>
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
20 private:
21     int value;
22 };
23
24 ostream& operator<<(ostream& stream, const Length& length);
25
26#endif //ADVANCED_LENGTH_H
27
```

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

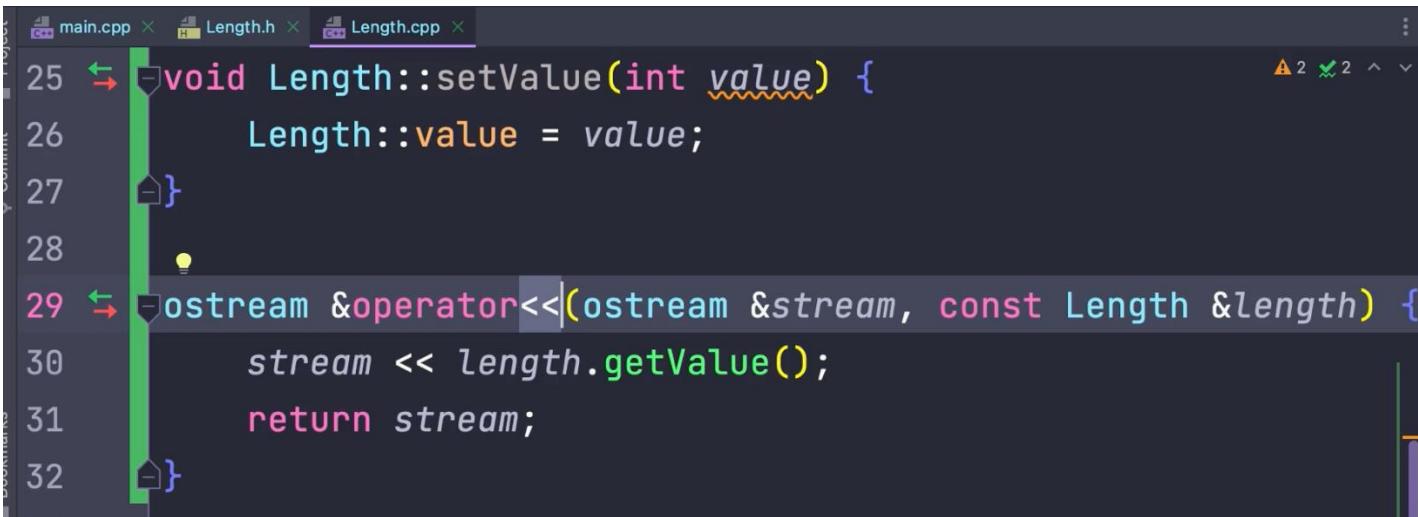
Overloading the Stream Insertion Operator

```
main.cpp x Length.h x Length.cpp x
C++ ADVANCED_LENGTH_H

5 #ifndef ADVANCED_LENGTH_H
6 #define ADVANCED_LENGTH_H
7
8 #include <compare>
9 #include <iostream>
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 };
24
25
26 ostream& operator<<(ostream& stream, const Length& length);
27
28 #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



A screenshot of a code editor showing the implementation of the stream insertion operator. The code is as follows:

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

- Implementation of the insertion (<<) operator.



A screenshot of a code editor showing a main function demonstrating the use of the overloaded insertion operator. The code is as follows:

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

Friends of Classes



```
28
29 ostringstream &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



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

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

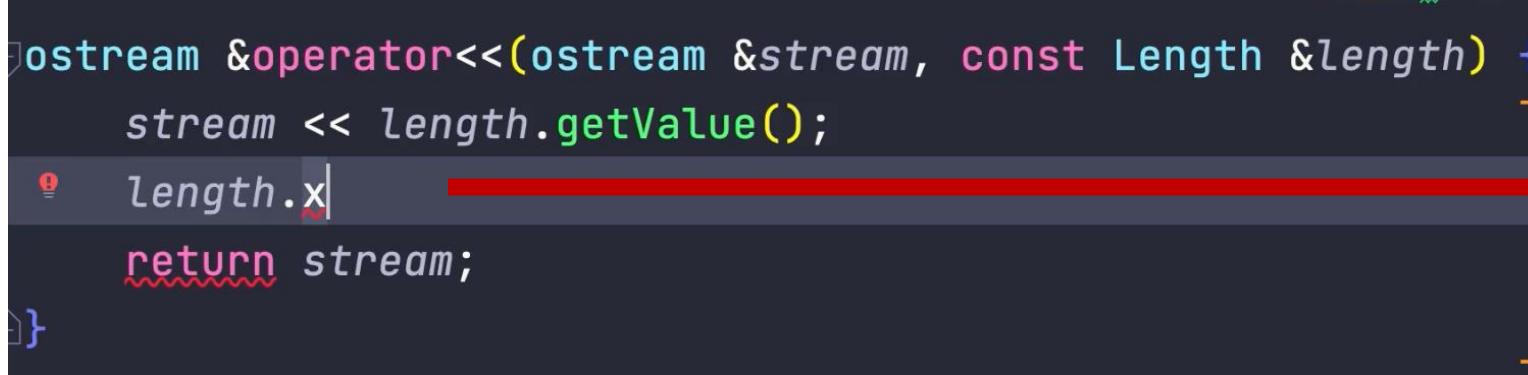
Friends of Classes



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

A screenshot of a C++ code editor showing a class definition named Length. The class has a private member 'value' and a private member 'x'. It contains three friend function declarations: an output operator '<<' that takes an ostream reference and a const Length reference, and two input operators '>>' that take an istream reference and a Length reference respectively. A red arrow points from the text 'friend' in the second friend declaration to the explanatory text below.

- 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;
}
```

A screenshot of a C++ code editor showing the implementation of the friend output operator. The function takes an ostream reference and a const Length reference. It first outputs the value of the 'length' object using its 'getValue()' method. Then, it attempts to access the private member 'x' of the 'length' object. A red arrow points from the line 'length.x' to the explanatory text below.

- 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 x Length.h x Length.cpp x
28
29     // Inline function
30     int getValue() const {
31         return value;
32     }
33     void setValue(int value);
34
35 private:
36     int value;
37 };
```

A screenshot of a code editor showing the header file Length.h. It contains an inline function declaration for getValue() and a definition for setValue(). A red arrow points from the inline function declaration in the header to its definition in the corresponding .cpp file.

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

A screenshot of a code editor showing the implementation of the inline function in Length.cpp. The declaration from the header is shown again, followed by the actual implementation code.

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

A screenshot of a code editor showing the main() function in main.cpp. It creates an object of type Length and calls its getValue() method. The code is annotated with various IDE features like breakpoints and code navigation icons.

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
31 private:
32     int value;
33 }
```

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

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

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