

# COP2334

## Introduction to Object Oriented Programming with C++

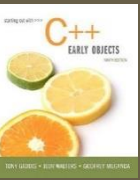
D. Singletary

### Module 13

Special Topics:

C++ Lambda Expressions (Text Ch. 11)

Debugging with Code::Blocks



# Special Topic:

# C++ Lambda Expressions

- Start by declaring a class which contains a function named sum that adds two numbers:

```
class SumFunction
{
public:
    int sum(int x, int y)
    {
        return (x + y);
    }
};
```

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

class SumFunction
{
public:
    int sum(int x, int y)
    {
        return (x + y);
    }
};

int main()
{
    SumFunction sf;

    cout << sf.sum(5, 3) << endl;
    return 0;
}
```

- Replace the sum function with a function operator overload
  - The C++ function operator is `()` (two parentheses)
- This is known as a function object

```
class SumFunctionOv
{
public:
    int operator()(int x, int y)
    {
        return (x + y);
    }
};
```

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

```
class SumFunctionOv
{
public:
    int operator()(int x, int y)
    {
        return (x + y);
    }
};
```

```
int main()
{
    SumFunctionOv sf1;
    SumFunctionOv sf2;

    cout << sf1(5, 3) << endl;
    cout << sf2(10, 20) << endl;
    return 0;
}
```

```

C:\temp\SumFunctionOv.exe
8
30

Process returned 0 (0x0)
Press any key to continue.
```

- We don't have to create variables for our SumFunction objects
  - we can create the objects anonymously

```
int main()
{
    SumFunctionOv sf1;
    SumFunctionOv sf2;

    cout << sf1(5, 3) << endl;
    cout << sf2(10, 20) << endl;

    cout << SumFunctionOv()(5, 3) << endl;
    cout << SumFunctionOv()(10, 20) << endl;
    return 0;
}
```

- A lambda expression also known as a *closure*) is an efficient way of creating an anonymous function object using a class *whose only member is the function call operator*
  - (this type of class is known as a closure type)
- The lambda operator is `[]` (two square brackets)
- The operator replaces the name and return type of the function (the compiler deduces the return type)

```
[](int a, int b) { return a + b; }
```

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

```
int main()
{
    int x = 5;
    int y = 3;
    cout << [](int a, int b) { return a + b; }(x, y) << endl;
    return 0;
}
```



- We can assign the lambda expression to a variable
  - NOTE: compiler insists on declare type as auto

```
auto sum = [](int a, int b) { return a + b; };
```

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

```
int main()
{
```

```
    // lambda expression
```

```
    auto sum = [](int a, int b) { return a + b; };
```

```
    int x = 5;
```

```
    int y = 3;
```

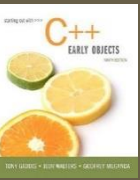
```
    cout << sum(x, y) << endl;
```

```
    return 0;
```

```
}
```

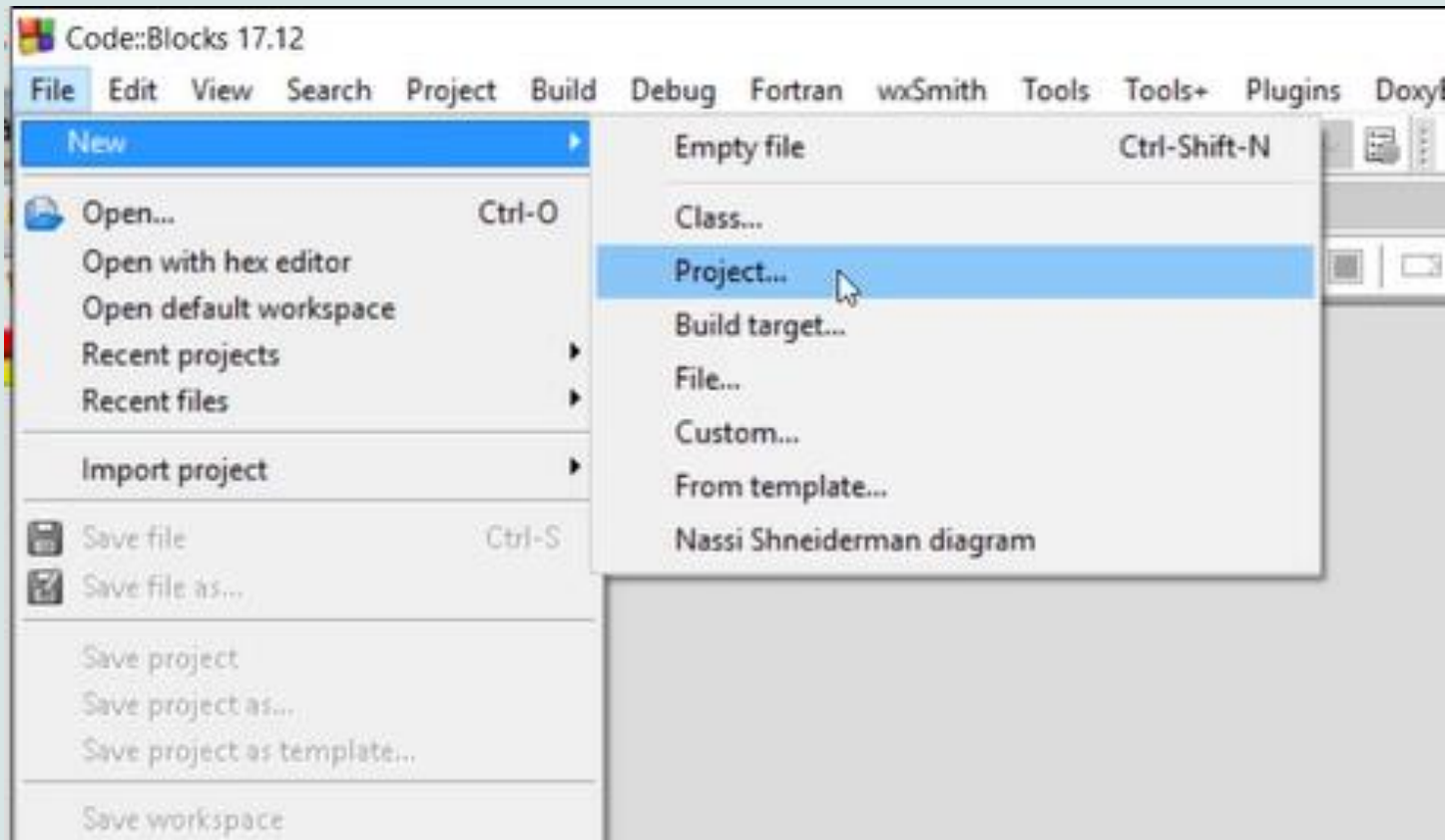
# Key Benefits to Lambdas

- Conciseness
- Reduction in code bloat
- Readability
- Elimination of shadow variables
- Encouragement of functional programming
- Code reuse
- Enhanced iterative syntax
- Simplified variable scope
- Less boilerplate code
- Parallel processing opportunities

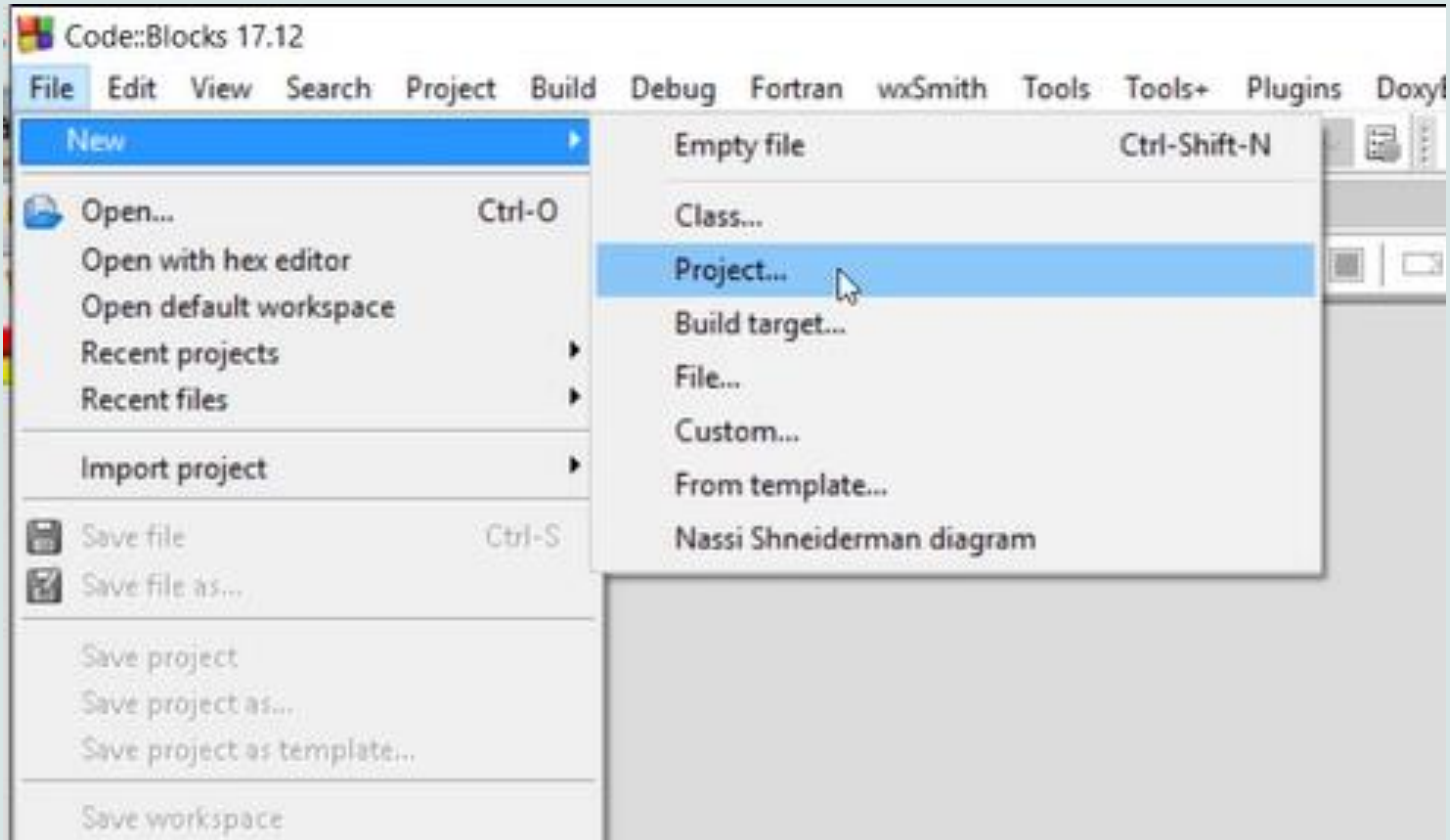


# Special Topic: Debugging with Code::Blocks

- Create a Project



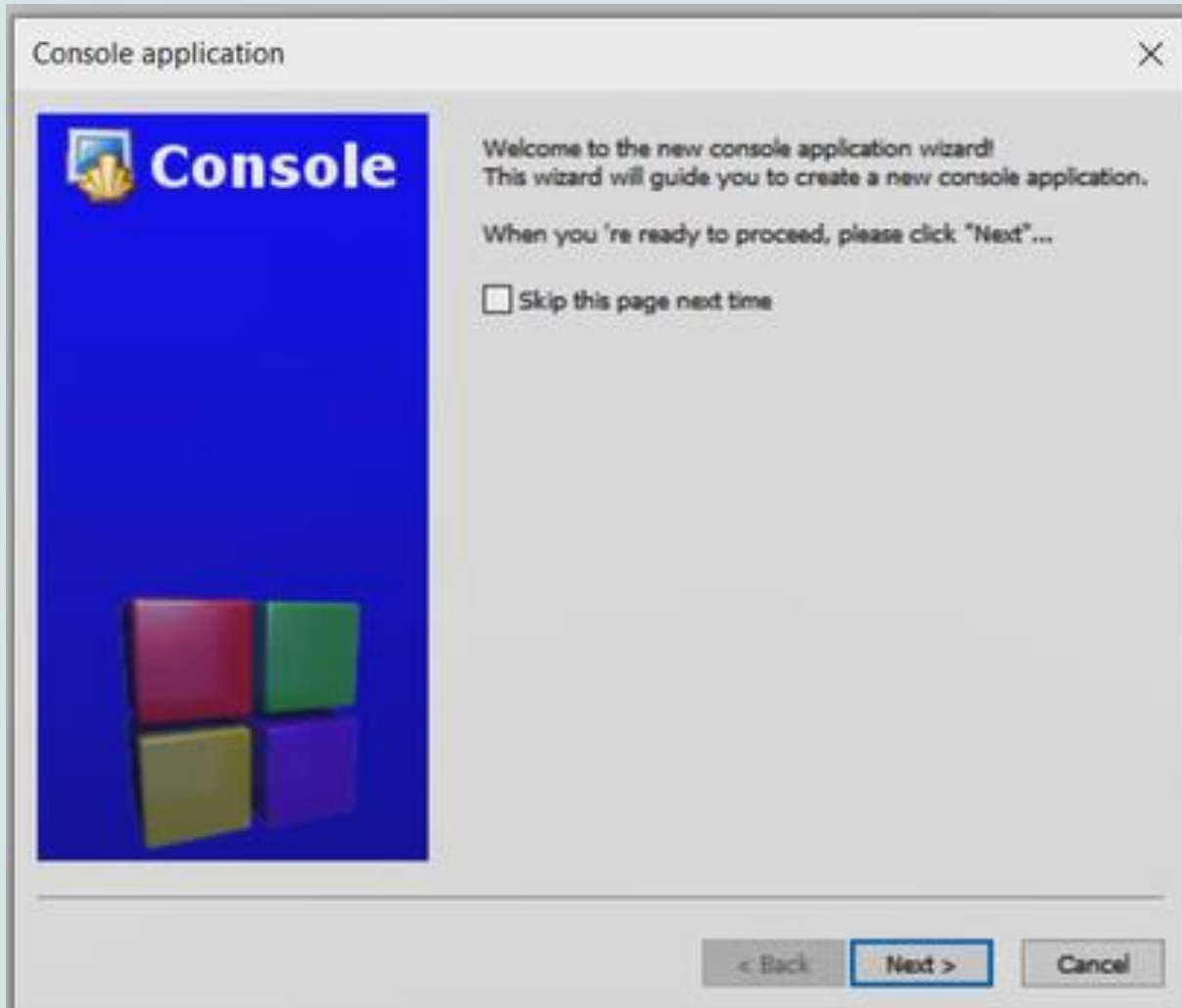
- Create a Project



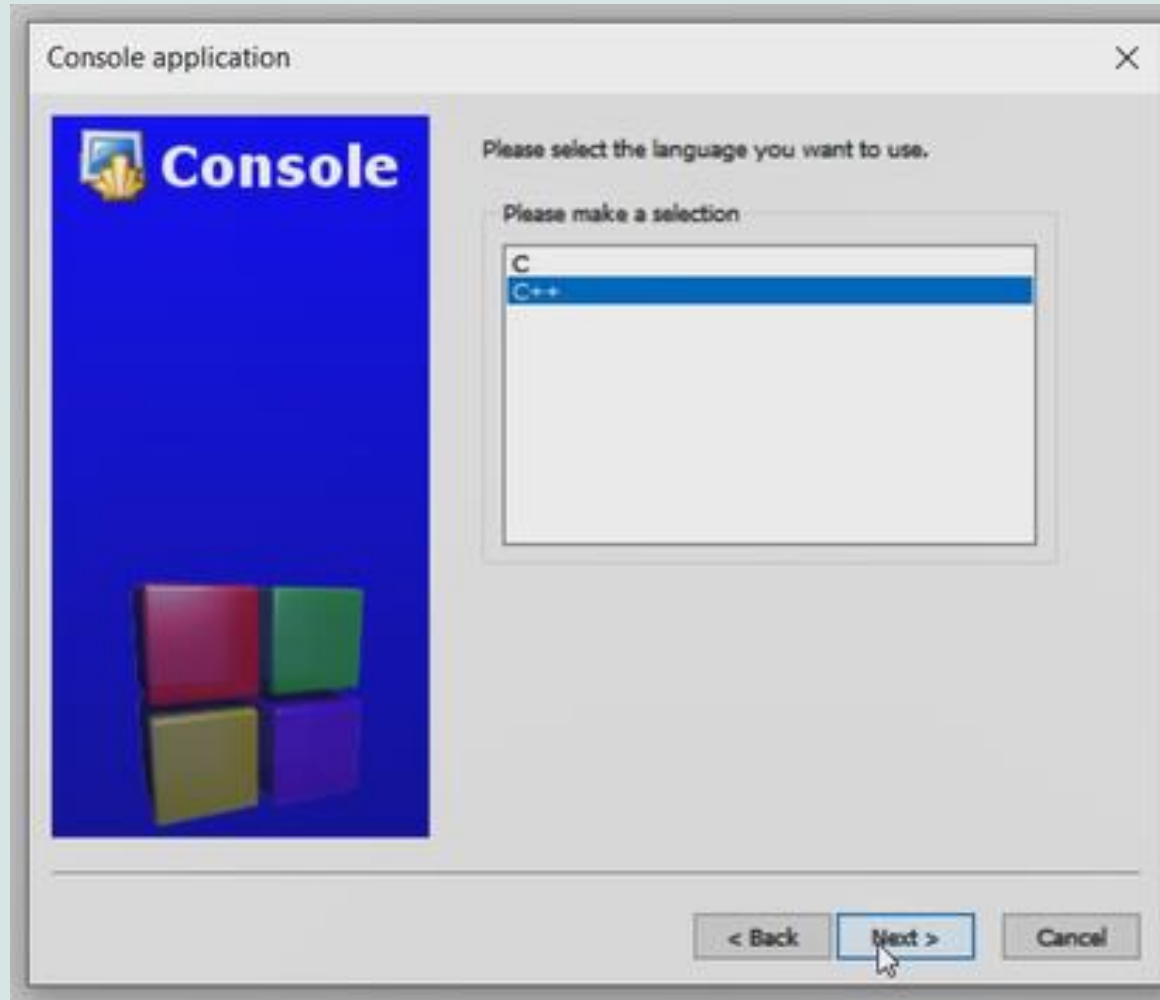
- Create a Project (Console application)



- Click Next

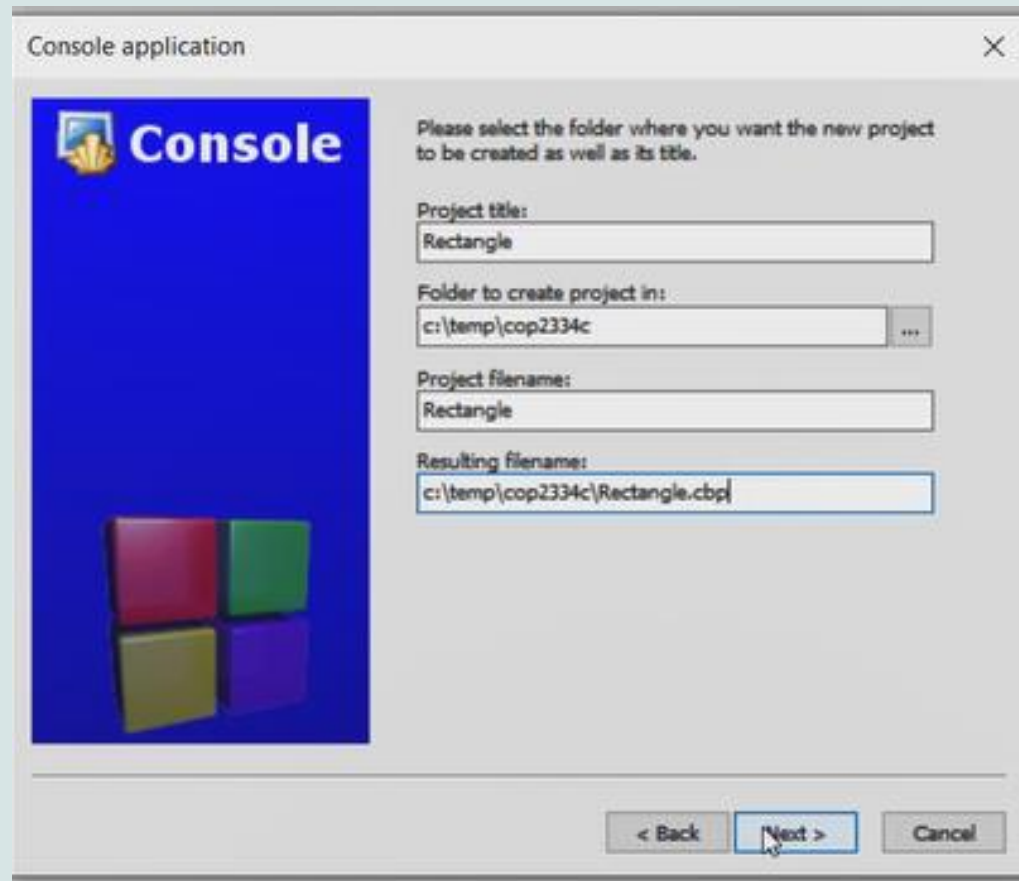


- Create a Project (C++, not C)

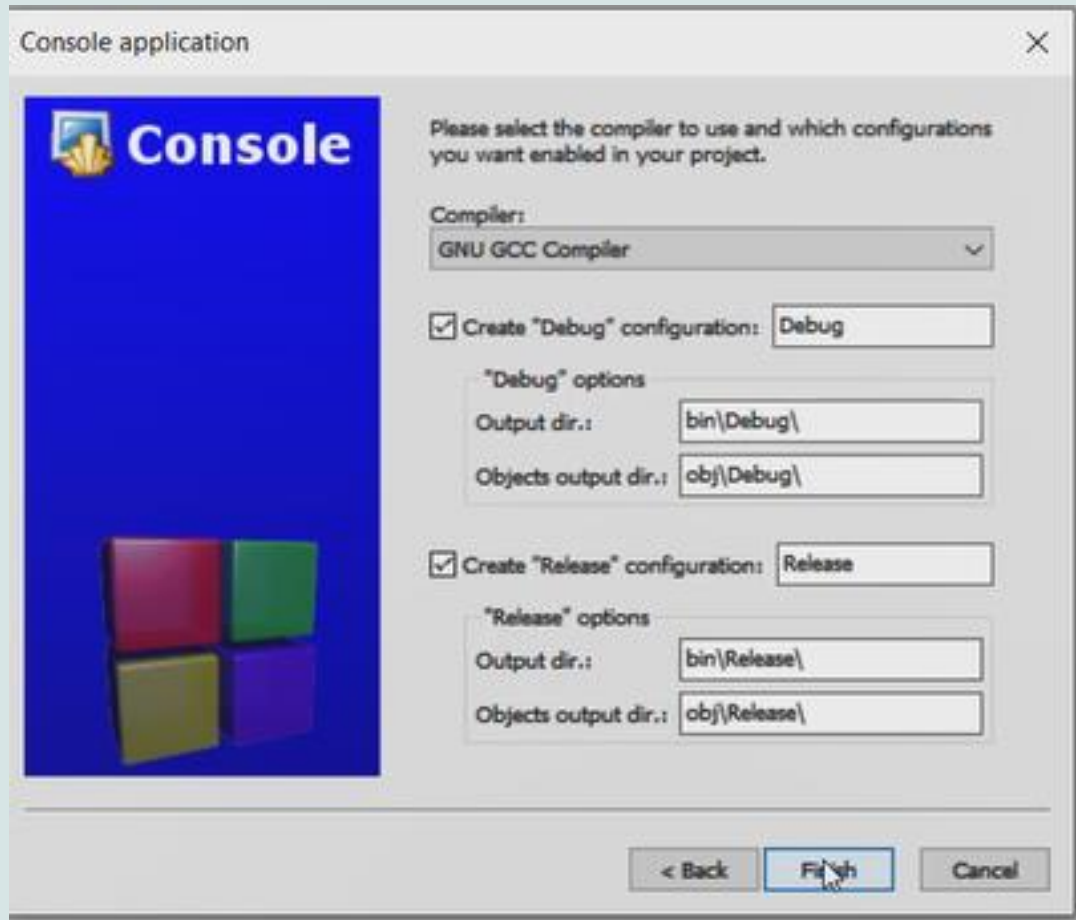




- Enter title and folder
- Code::Blocks will want to create a subfolder for the project, I usually remove this from the path

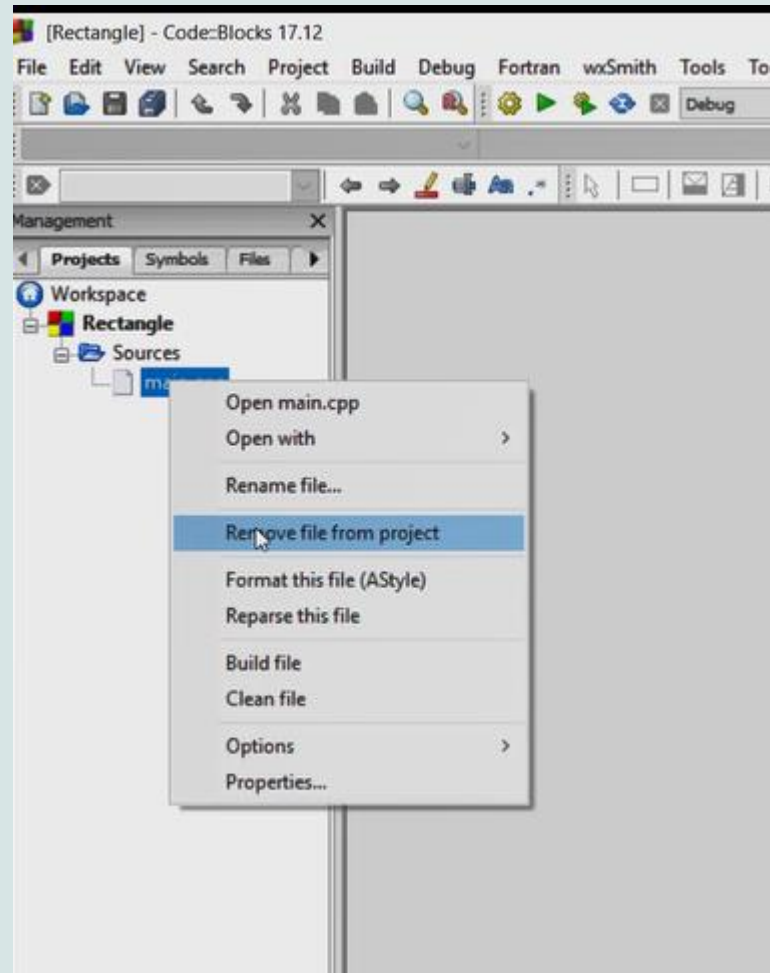


- Build Debug and Release configurations
  - Default is to build both debug and release configurations

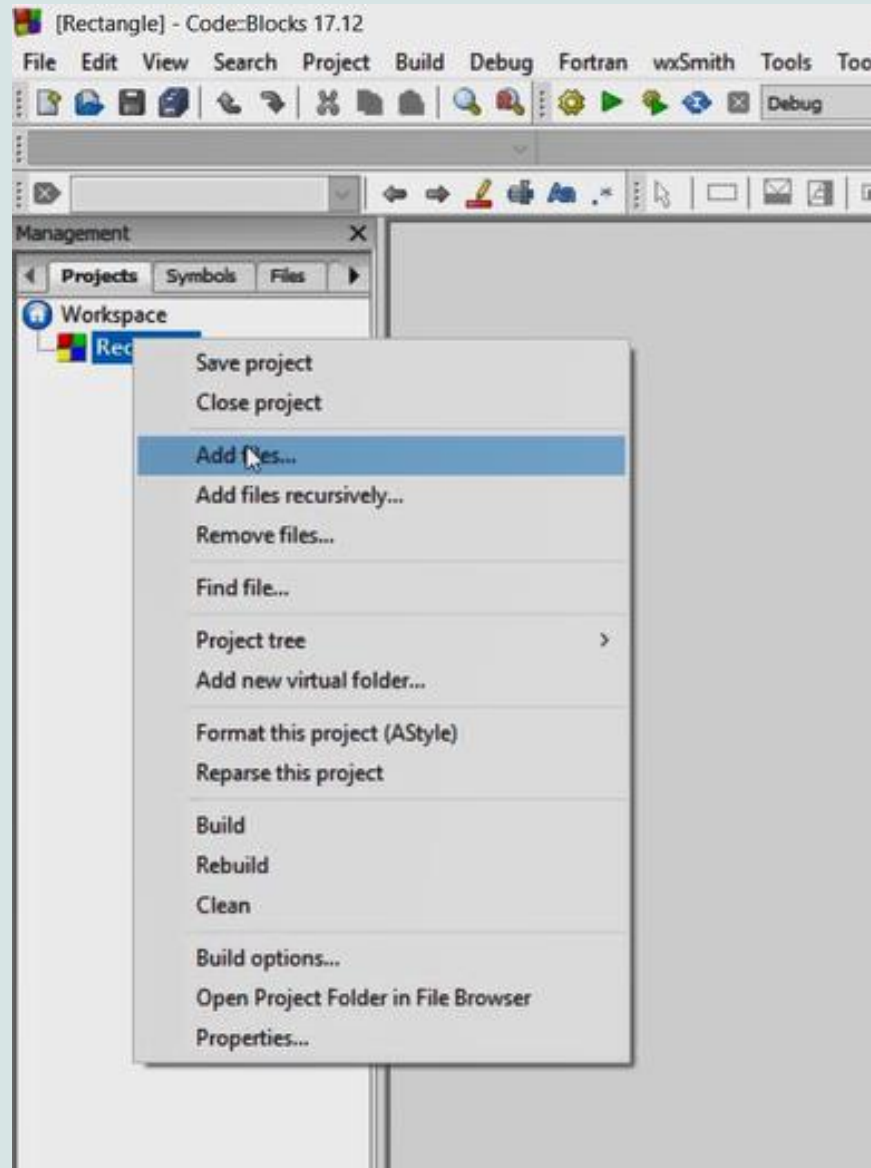


- A debug configuration contains extra information in the object files and subsequent executable image
  - The executable is bigger but allows visualization/manipulation of symbols (e.g. variables) in the debugger
  - A release configuration does not provide useful debugging information
- Never provide a debugging image to a customer unless you are working on a specific issue and have the customer's permission
  - Debugging images should only be executed under strictly controlled conditions for security and performance reasons

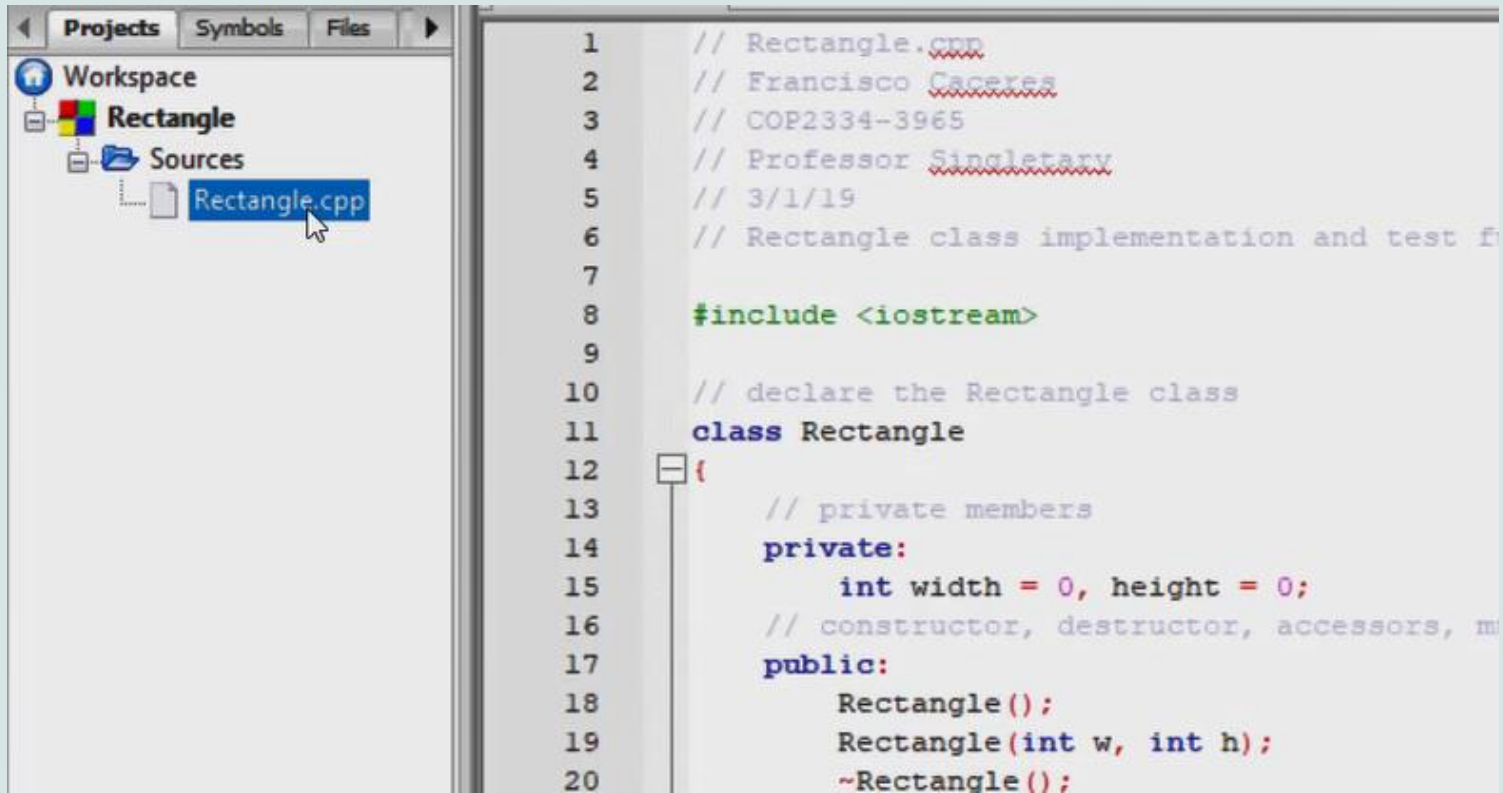
- Code::Blocks will create a new main.cpp file containing a basic main function.
- Use it if creating a project from scratch
- If you are using existing sources you can remove it from the project



- Add existing files to the project as necessary

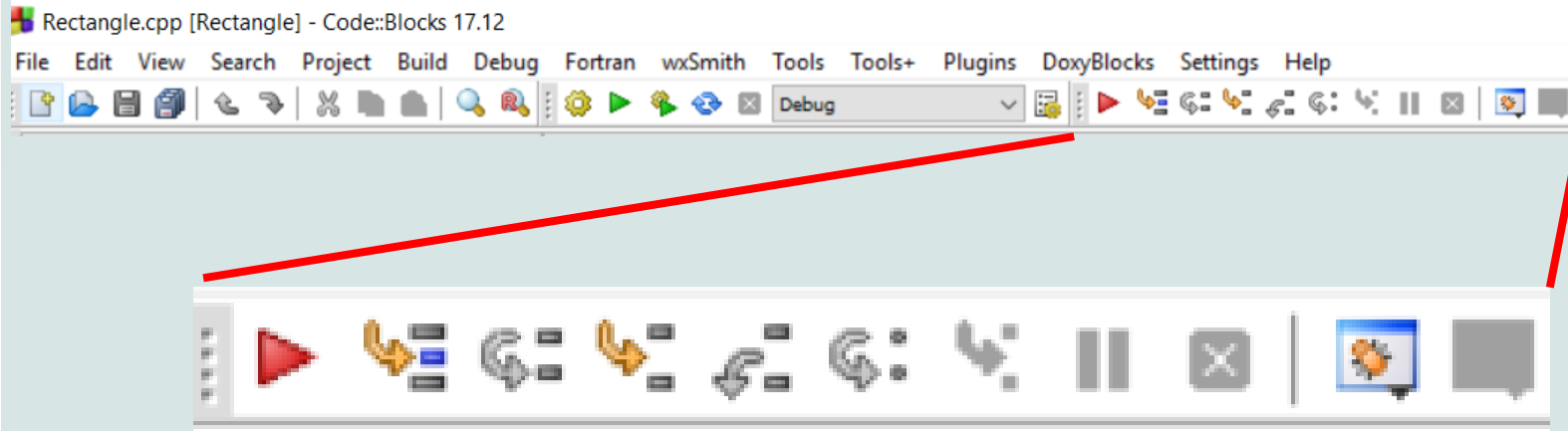


- Using existing Rectangle.cpp file



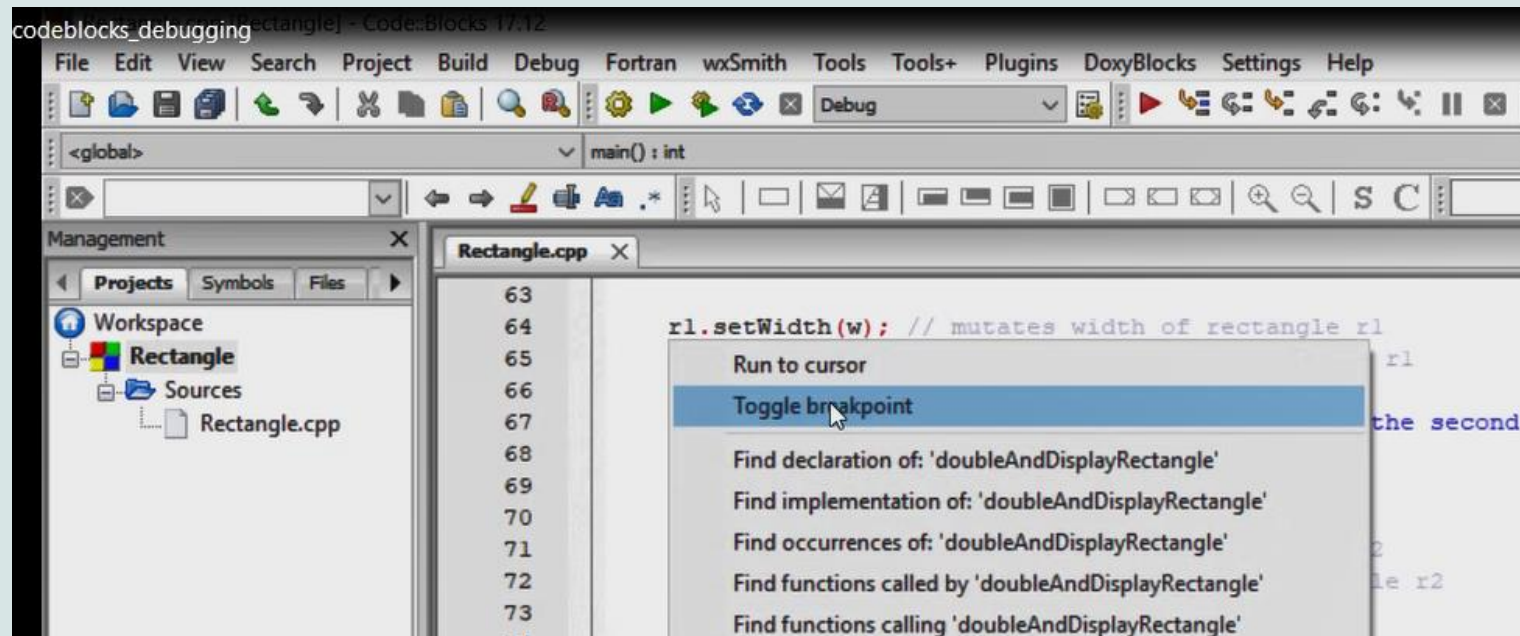
```
1 // Rectangle.cpp
2 // Francisco Caceres
3 // COP2334-3965
4 // Professor Singletary
5 // 3/1/19
6 // Rectangle class implementation and test f
7
8 #include <iostream>
9
10 // declare the Rectangle class
11 class Rectangle
12 {
13     // private members
14     private:
15         int width = 0, height = 0;
16     // constructor, destructor, accessors, m
17     public:
18         Rectangle();
19         Rectangle(int w, int h);
20         ~Rectangle();
```

# Debugging Tools



# • Setting Breakpoints

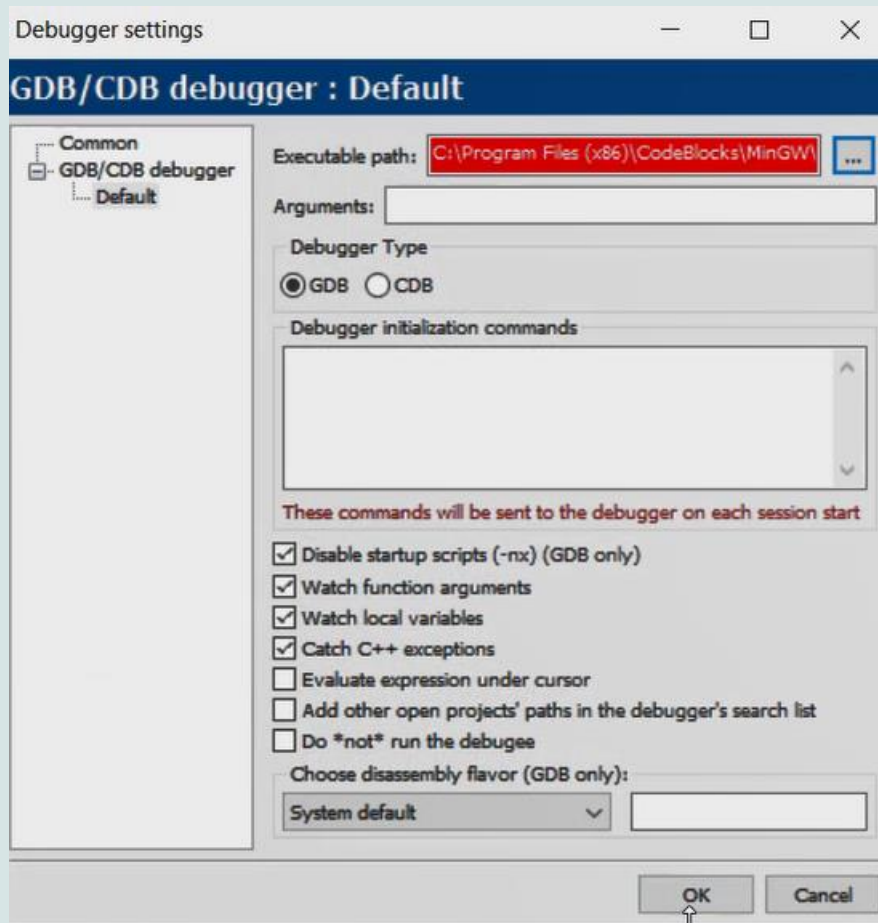
- Breakpoints will "break" execution of a program at a specific point so the call stack and variables can be examined
- Position your cursor on the beginning of the line where the breakpoint is desired
- Right-click and select "Toggle breakpoint"
- Select "Debug/Continue" (red arrow)



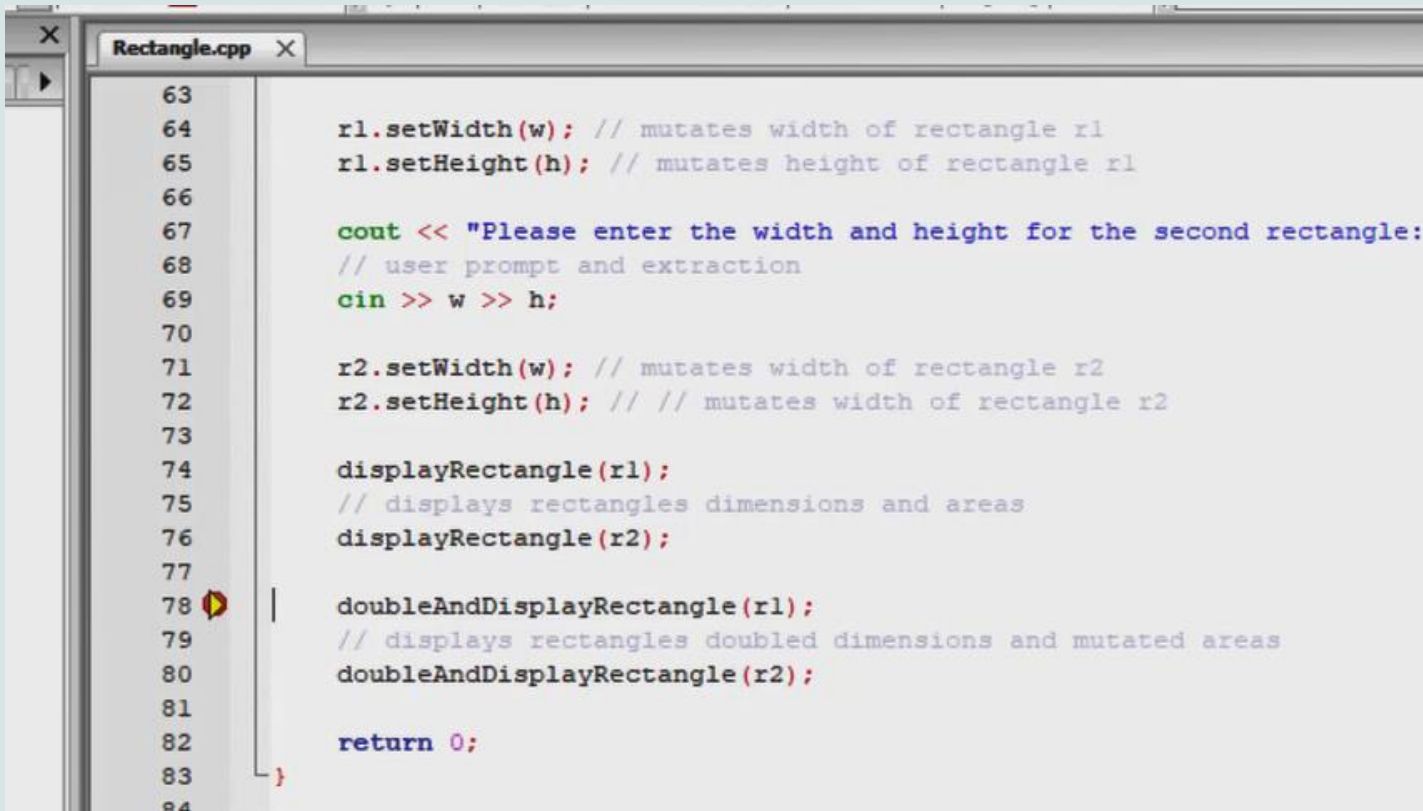


# • Configuring the debugger

- You may need to browse for your installed debugger
- Settings/Debugger
- Example: under the CodeBlocks install folder, MinGW\bin\gdb32.exe



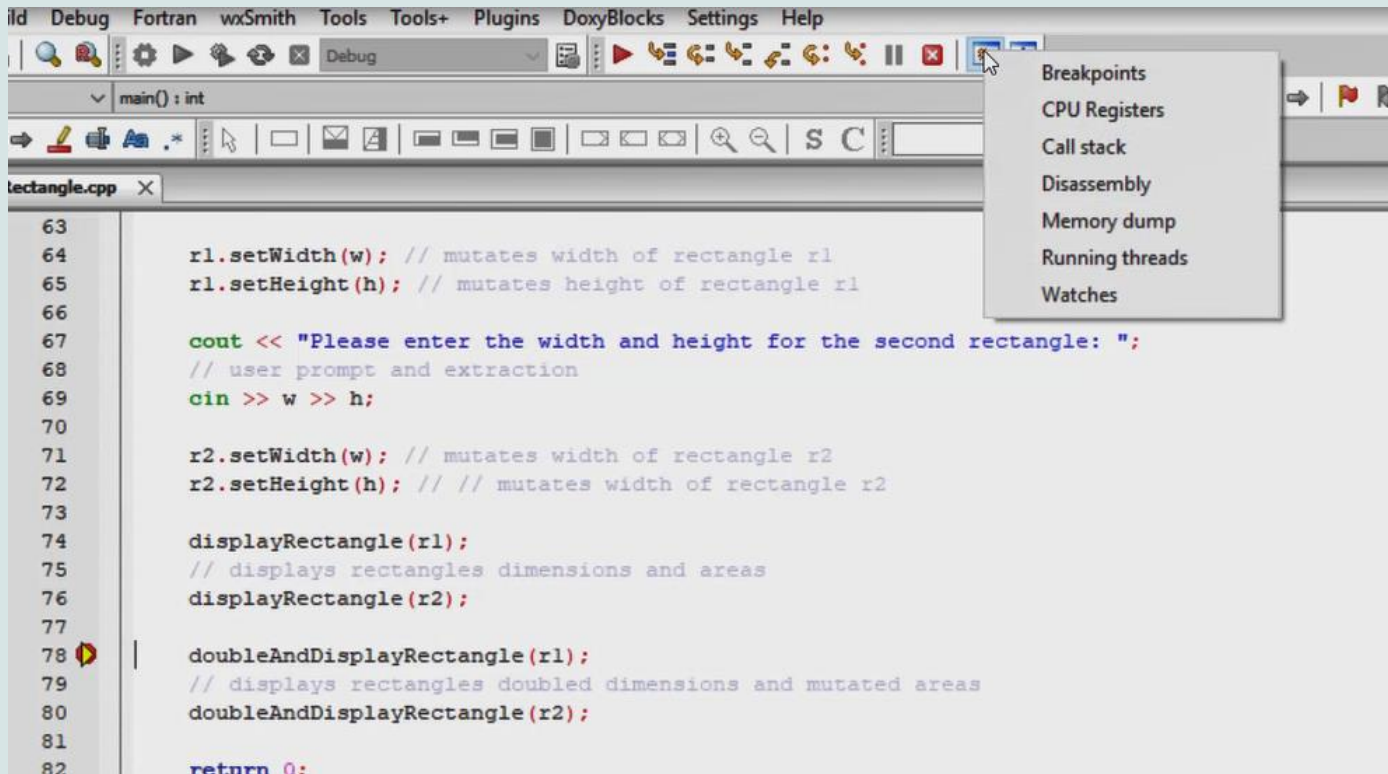
- When a breakpoint is hit
  - The yellow arrow inside the red breakpoint ("stop sign") indicates the program has broken at that statement



```

63
64     r1.setWidth(w); // mutates width of rectangle r1
65     r1.setHeight(h); // mutates height of rectangle r1
66
67     cout << "Please enter the width and height for the second rectangle:
68     // user prompt and extraction
69     cin >> w >> h;
70
71     r2.setWidth(w); // mutates width of rectangle r2
72     r2.setHeight(h); // // mutates width of rectangle r2
73
74     displayRectangle(r1);
75     // displays rectangles dimensions and areas
76     displayRectangle(r2);
77
78     doubleAndDisplayRectangle(r1);
79     // displays rectangles doubled dimensions and mutated areas
80     doubleAndDisplayRectangle(r2);
81
82     return 0;
83 }
84
  
```

- Viewing the debug windows
  - Call stack: stack frames at the current execution point
  - Watches: variable values



The screenshot shows a C++ IDE with a code editor and a debug menu. The code editor displays the following code:

```

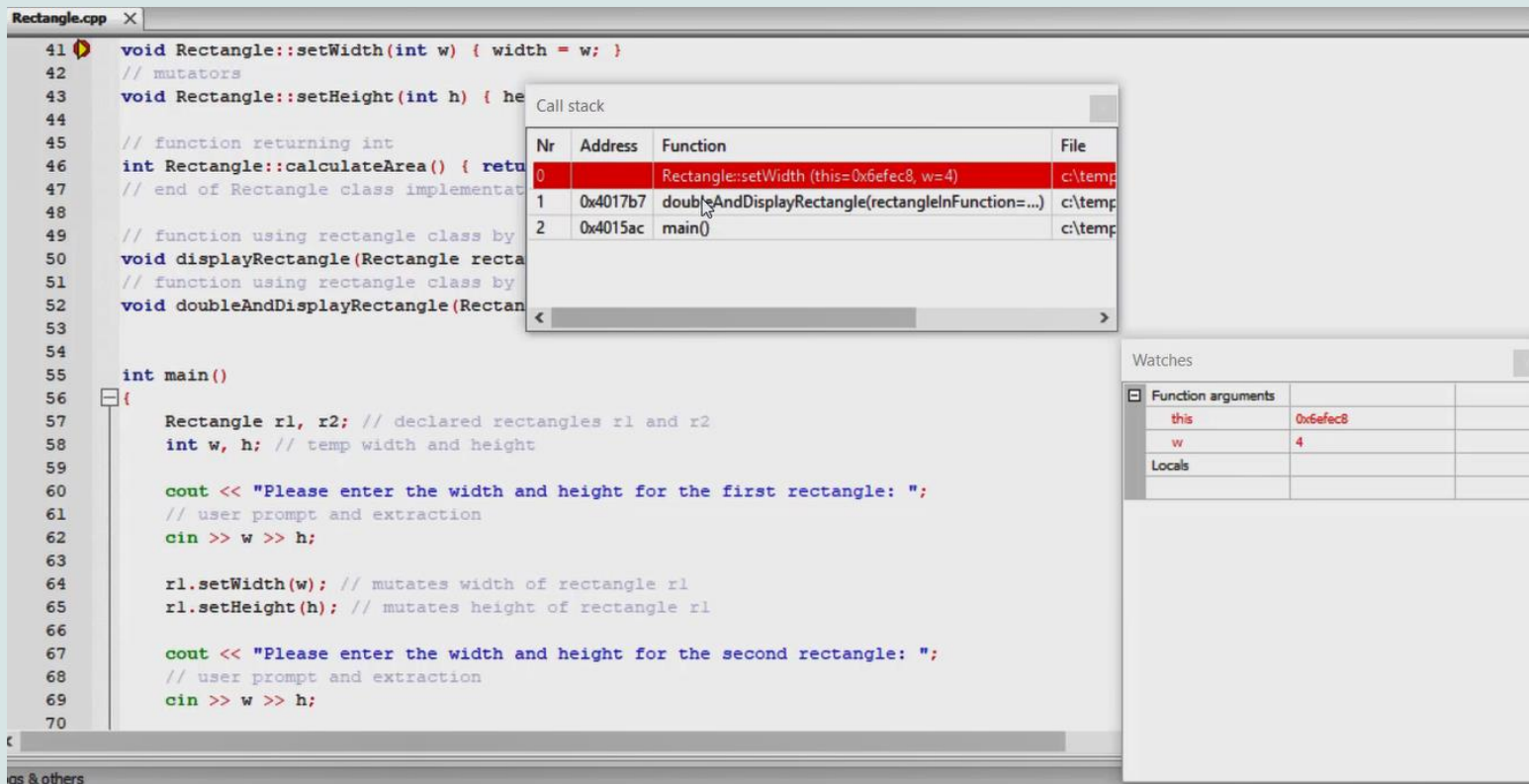
63
64     r1.setWidth(w); // mutates width of rectangle r1
65     r1.setHeight(h); // mutates height of rectangle r1
66
67     cout << "Please enter the width and height for the second rectangle: ";
68     // user prompt and extraction
69     cin >> w >> h;
70
71     r2.setWidth(w); // mutates width of rectangle r2
72     r2.setHeight(h); // // mutates width of rectangle r2
73
74     displayRectangle(r1);
75     // displays rectangles dimensions and areas
76     displayRectangle(r2);
77
78     doubleAndDisplayRectangle(r1);
79     // displays rectangles doubled dimensions and mutated areas
80     doubleAndDisplayRectangle(r2);
81
82     return 0;
  
```

The debug menu is open, showing the following options:

- Breakpoints
- CPU Registers
- Call stack
- Disassembly
- Memory dump
- Running threads
- Watches

# • Breakpoint set at Rectangle class mutator

- When breakpoint is hit from the call inside the doubleAndDisplayRectangle function, we see a 3-deep stack trace in the Call stack window and values for the w parameter and "this" in the Watches window
- Double click on a stack frame to activate in Watches



The screenshot shows a C++ IDE with a file named `Rectangle.cpp`. A breakpoint is set at line 41, `void Rectangle::setWidth(int w) { width = w; }`. The call stack window shows three frames:

Nr	Address	Function	File
0		Rectangle::setWidth (this=0x6efec8, w=4)	c:\temp
1	0x4017b7	doubleAndDisplayRectangle(rectangleInFunction=...)	c:\temp
2	0x4015ac	main()	c:\temp

The Watches window shows the following values:

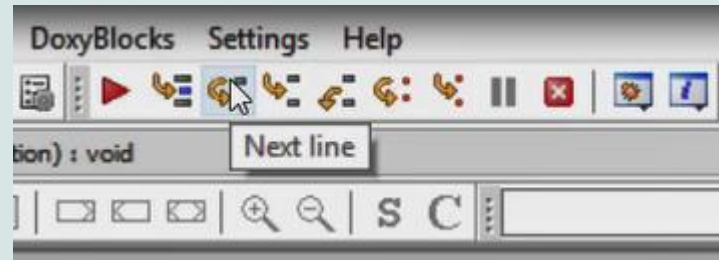
Function arguments	
this	0x6efec8
w	4
Locals	

# • Adding variables to Watches

- Add a global variable and set it at various points
- Toggle a breakpoint and step through the code using "Next line" tool to watch its values
- "Step into" will enter a function, "Next line" will just execute it and step over it

```

8      #include <iostream>
9
10     int globalVar = 100;
11
  
```



```

void doubleAndDisplayRectangle(Rectangle& rectangleInFunction)
{
    const string DOUBLE_MESSAGE = "Doubling the rectangle dimensions!";

    globalVar = 200;
    cout << DOUBLE_MESSAGE << endl; // outputs message
    rectangleInFunction.setWidth(rectangleInFunction.getWidth() * 2);
    //mutates dimensions
    rectangleInFunction.setHeight(rectangleInFunction.getHeight() * 2);

    globalVar = 300;
}
  
```

# • Program crashes when running a debug configuration will activate debugger

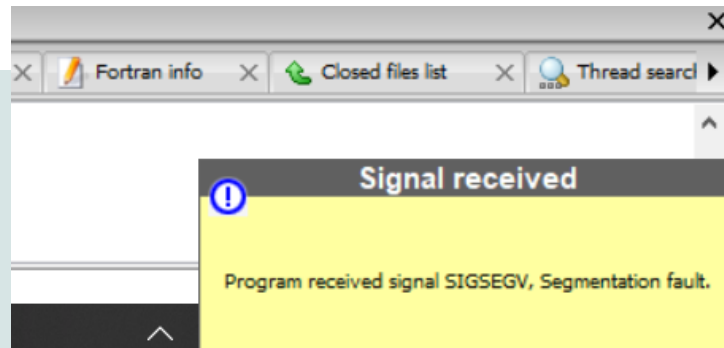
- Add a pointer variable, initialize to null, then attempt to dereference
- Execute program from debugger with no breakpoints

```
int * ptr = nullptr;

globalVar = 200;
cout << DOUBLE_MESSAGE << endl; // outputs message
rectangleInFunction.setWidth(rectangleInFunction.getWidth() * 2);
//mutates dimensions
rectangleInFunction.setHeight(rectangleInFunction.getHeight() * 2);

globalVar = 300;

cout << *ptr << endl;
```





- Notice null pointer value in Watches window
- Program is broken at line 110 (no breakpoint)

```

82 doubleAndDisplayRectangle(r2);
83
84 ret
85 }
86
87 // outputs dimensions and area
88 void doubleAndDisplayRectangle(Rectangle& rectangleInFunction)
89 {
90     cout << "Doubling the rectangle dimensions and area\n";
91
92     // passes rectangle class by reference and outputs dimensions and area
93     // mutates dimensions
94     // mutates area
95     // passes rectangle class by reference and outputs dimensions and area
96     void doubleAndDisplayRectangle(Rectangle& rectangleInFunction)
97     {
98         const string DOUBLE_MESSAGE = "Doubling the rectangle dimensions and area\n";
99
100         int * ptr = nullptr;
101
102         globalVar = 200;
103         cout << DOUBLE_MESSAGE << endl; // outputs message
104         rectangleInFunction.setWidth(rectangleInFunction.getWidth());
105         //mutates dimensions
106         rectangleInFunction.setHeight(rectangleInFunction.getHeight());
107
108         globalVar = 300;
109
110         cout << *ptr << endl;
111     }

```

Call stack

Nr	Address	Function	File
0	0x4017f0	doubleAndDisplayRectangle(rectangleInFunction=...)	c:\temp\...
1	0x4015ac	main()	c:\temp\...

Watches

Variable	Value	Type
rectangleInFunction	@0x5efec8:	
DOUBLE_MESSAGE		
ptr	0x0	
globalVar	300	int