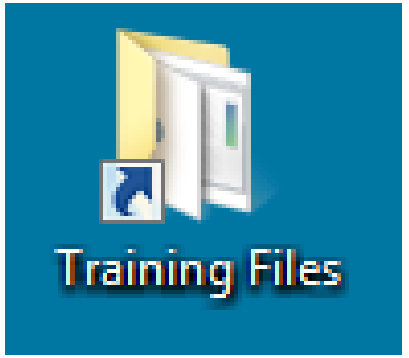
Introduction to C++: Part 1

tutorial version 0.1

Brian Gregor

Research Computing Services

Getting started with the room B27 terminals



- Log on with your BU username
- On the desktop is a *Training Files* folder. Open it and go to the subfolder:

RCS_Tutorials\Tutorial Files\Introduction to C++

- Copy the *CodeBlocks Projects* folder to your desktop.

Getting started on the SCC

- If you prefer to work on the SCC *and have your own account*, login using your account to the host **scc2.bu.edu**
  - On the room terminals there is a MobaXterm link on the desktop

- Load the codeblocks module:

```
module load gcc/5.3.0
module load hunspell/1.4.1
module load wxwidgets/2.8.12
module load gdb/7.11.1
module load codeblocks
```

- Make a folder in your home directory and copy in the files:

```
mkdir cpp_tutorial && cd !$
unzip /scratch/intro_to_cpp_tutorial_0.1.zip
```

Getting started with your own laptop

- Go to:

  http://www.bu.edu/tech/support/research/training-consulting/live-tutorials/

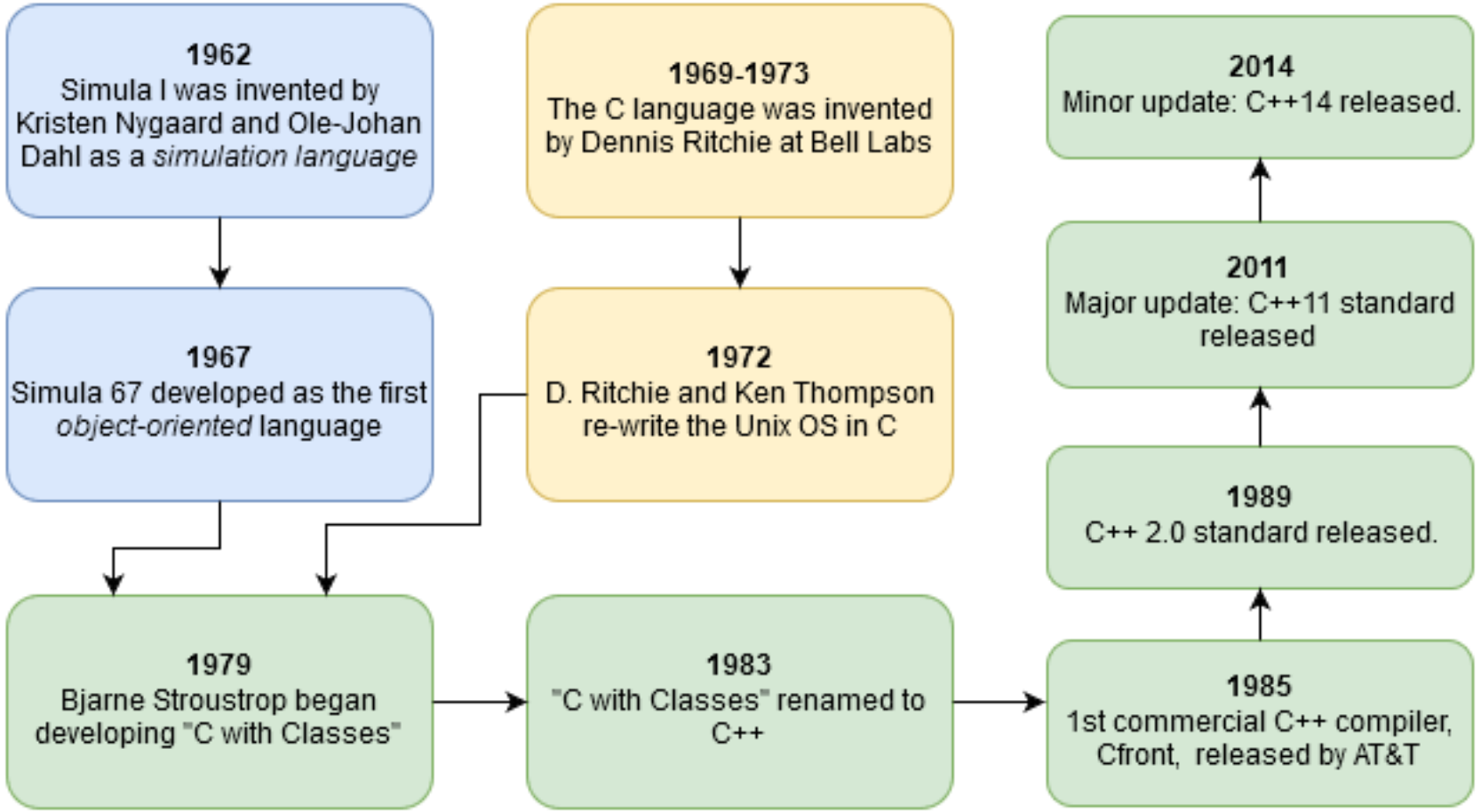
  and download the Powerpoint or PDF copy of the unified presentation.
- Easy way to get there: Google “bu rcs tutorials” and it’s the 1st or 2nd link.
- Also download the “Additional Materials” file and unzip it to a convenient folder on your laptop.
- Download the Code::Blocks development environment:

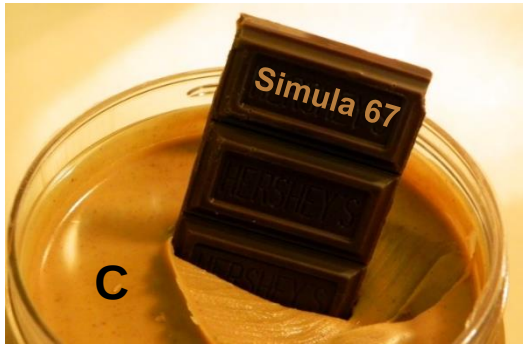
  http://www.codeblocks.org/downloads/26
- Windows: get the **codeblocks-16.01mingw-nosetup.zip** file and unzip it to a convenient folder.
- Linux: likely available from your Linux distro’s package management system
- Mac OSX: get the **CodeBlocks-13.12-mac.zip** file and unzip it to a convenient folder.
  - Also you will need Apple’s Xcode software with the **command line tools** installed.

Tutorial Outline: Part 1

- Very brief history of C++
- Definition object-oriented programming
- When C++ is a good choice
- The Code::Blocks IDE
- Object-oriented concepts
- First program!
- Some C++ syntax
- Polymorphism

Very brief history of C++





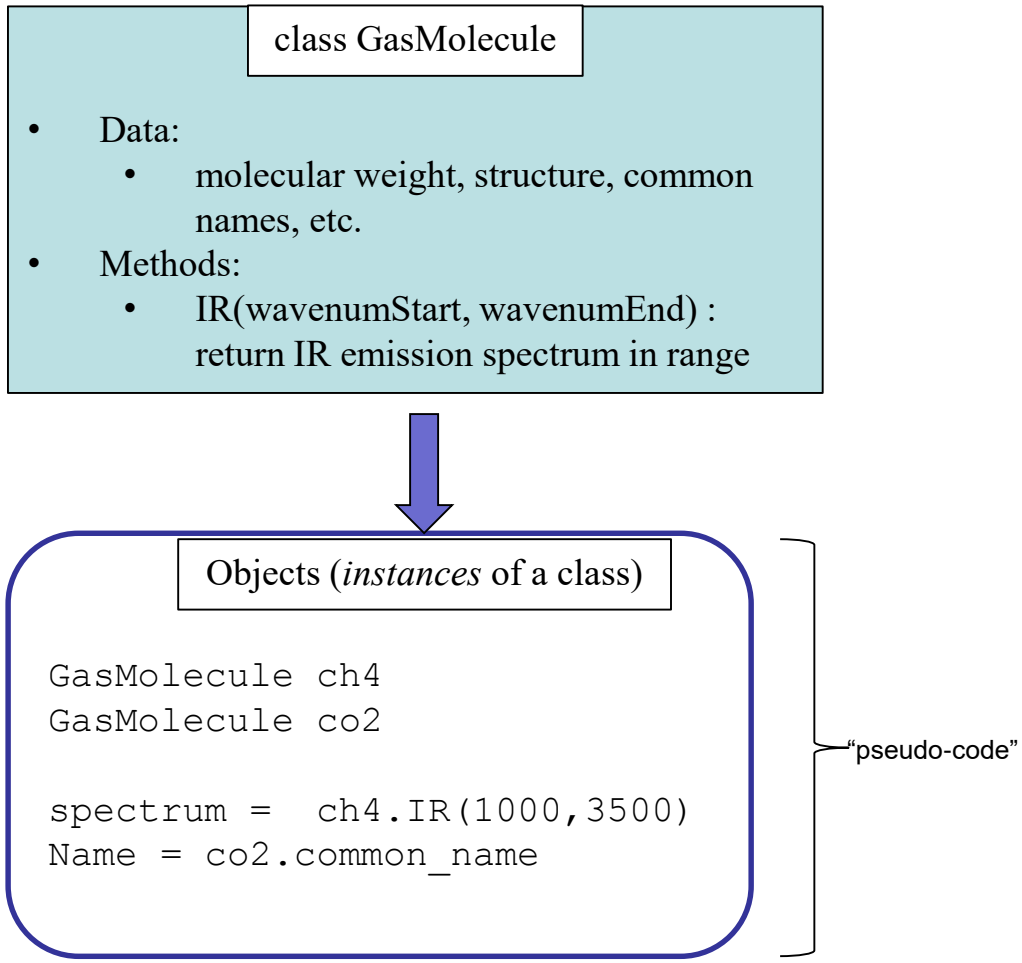
C++

For details more check out [A History of C++: 1979−1991]

BOSTON UNIVERSITY

Object-oriented programming

- Programming has many *paradigms*, or styles, which are used when writing programs.
  - Wikipedia lists >40!: https://en.wikipedia.org/wiki/Programming_paradigm
  - Procedural (C, Fortran, Matlab)
  - Dataflow (Simulink, VHDL, Labview)
  - Functional (Excel, Lisp, F#)
- Object-oriented programming (OOP):
  - Seeks to define a program in terms of the things in the problem (files, molecules, buildings, cars, people, etc.) and what they need and can do.



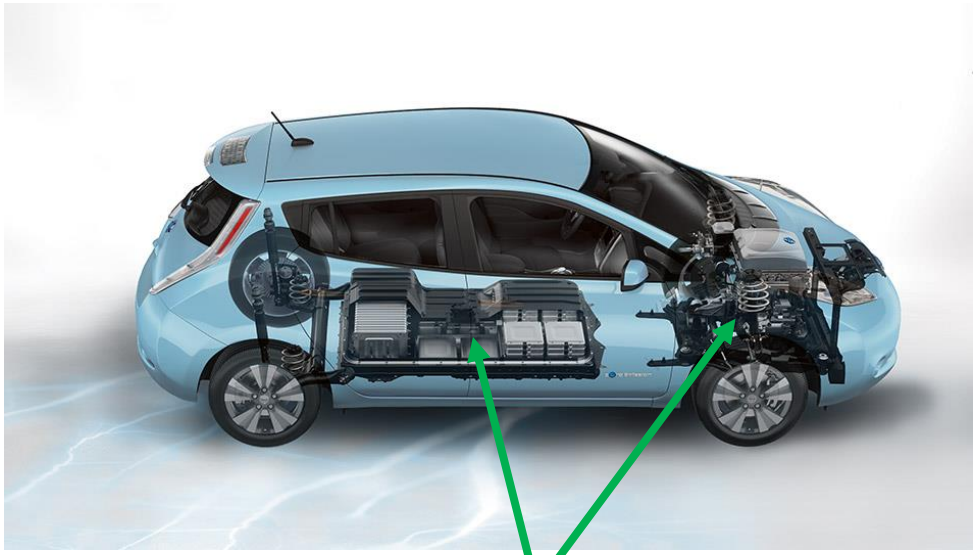
BOSTON UNIVERSITY

Object-oriented programming

- OOP defines *classes* to represent these things.
- Classes can contain data and methods (internal functions).
- Classes control access to internal data and methods. A public interface is used by external code when using the class.
- This is a highly effective way of modeling real world problems inside of a computer program.

"Class Car"

public interface



private data and methods

BOSTON UNIVERSITY

C++ Compared to Some Other Languages

	C++	Python	Fortran
Language Type	compiled	interpreted	compiled
Variable type style	Strong	Strong	Strong
Variable Safety	unsafe	safe	mostly safe
Type checking	At compilation and at run-time	Run-time	Compilation
Paradigms	OO, procedural, functional, generic, dataflow, and others	OO, procedural, functional	Procedural
C compatibility	Nearly 100%	Not directly	Call C libraries, with many pitfalls
Relative speed	Fast	Slow	Fast

> "Actually I made up the term 'object-oriented', and I can tell you I did not have C++ in mind."
>
> – Alan Kay (helped invent OO programming, the Smalltalk language, and the GUI)

When to choose C++

> "If you're not at all interested in performance, shouldn't you be in the Python room down the hall?"
> — Scott Meyers (author of Effective Modern C++)

- Despite its many competitors C++ has remained popular for ~30 years and will continue to be so in the foreseeable future.
- Why?
  - Complex problems and programs can be effectively implemented
  - OOP works in the real world!
  - No other language quite matches C++'s combination of performance, expressiveness, and ability to handle complex programs.

- Choose C++ when:
  - Program performance matters
    - Dealing with large amounts of data, multiple CPUs, complex algorithms, etc.
  - Programmer productivity is less important
    - It is faster to produce working code in Python, R, Matlab or other scripting languages!
  - The programming language itself can help organize your code
    - Not everything is a vector or matrix, right Matlab?
  - Access to libraries that will help with your problem
    - Ex. Nvidia's CUDA Thrust library for GPUs
  - Your group uses it already!

Pros/Cons of C++

Pros

- Enormous number of available libraries
- Flexibility for programmers
  - High (objects) and low (fiddling with memory) level styles are supported
- No automatic memory management
  - You are in control of memory usage
- Compiled
- Strong type system
- High performance

Cons

- A *very* large language - this tutorial won't even **attempt** to describe all of it.
    - And your instructor makes no claim to know the entire language!
- No automatic memory management
  - You are in control of memory usage
- Includes all the subtleties of C and adds its own
- Generally requires careful attention to detail!

> "C++: an octopus made by nailing extra legs onto a dog."
> – Steve Taylor

Code::Blocks

- In this tutorial we will use the Code::Blocks integrated development environment (IDE) for writing and compiling C++
  - Run it right on the terminal or on the SCC (`module load codeblocks`)
- About C::B
  - cross-platform: supported on Mac OSX, Linux, and Windows
  - Oriented towards C, C++, and Fortran, supports others such as Python
  - Short learning curve compared with other IDEs such as Eclipse or Visual Studio
- Has its own automated code building system, so we can concentrate on C++
  - It can convert its build system files to *make* and Makefiles so you are not tied to C::B
- Project homepage: http://www.codeblocks.org

IDE Advantages

- Handles build process for you
- Syntax highlighting and live error detection
- Code completion (fills in as you type)
- Creation of files via templates
- Built-in debugging
- Code refactoring (ex. Change a variable name everywhere in your code)
- Higher productivity

IDEs available on the SCC

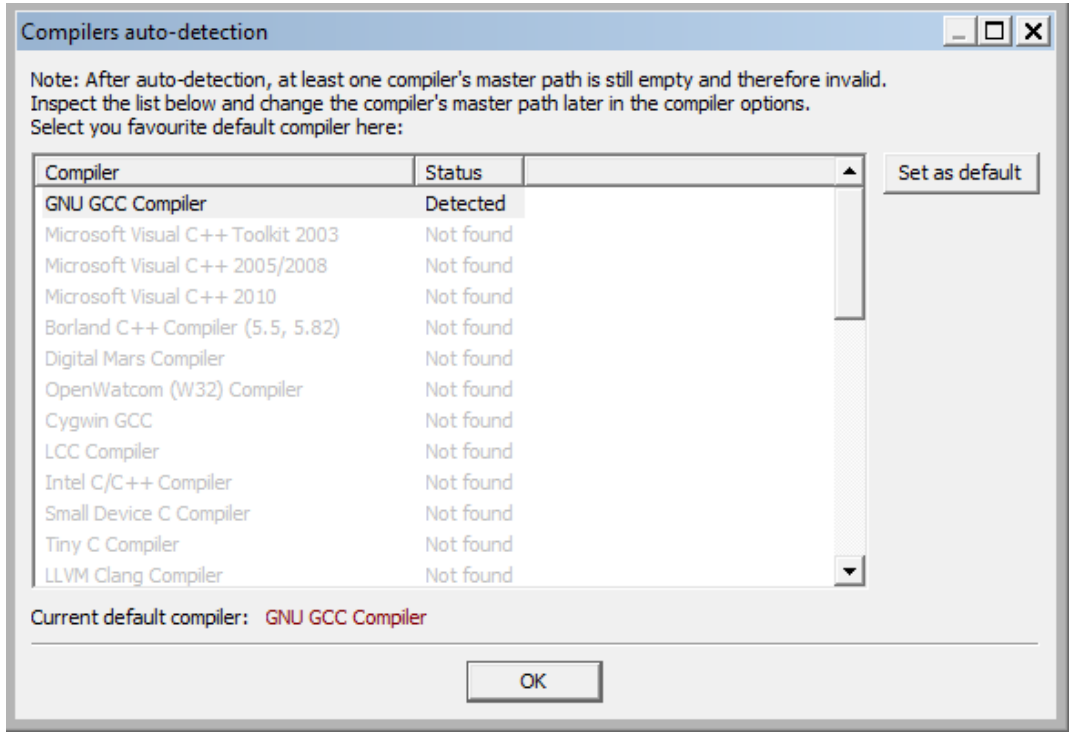
- Code::Blocks (used here)
- geany – a minimalist IDE, simple to use
- Eclipse – a highly configurable, adaptable IDE. Very powerful but with a long learning curve
- Spyder – Python only, part of Anaconda

Some Others

- Xcode for Mac OSX
- Visual Studio for Windows
- NetBeans (cross platform)

Opening C::B

- The 1st time it is opened C::B will search for compilers it can use.
- A dialog that looks like this will open. Select GCC if there are multiple options:

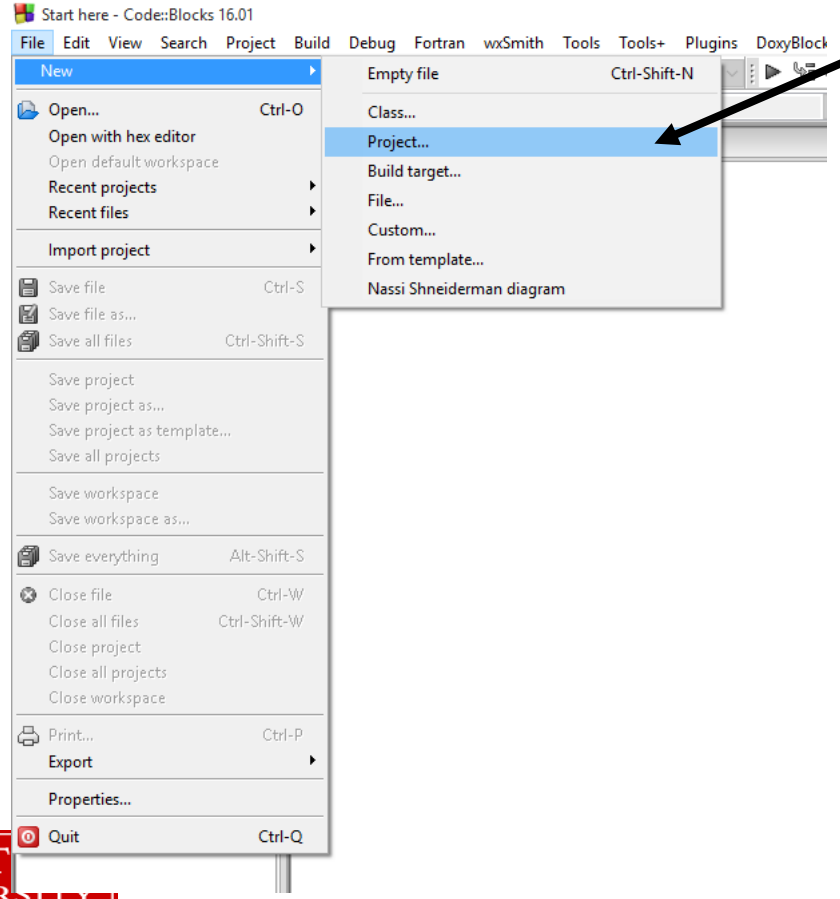


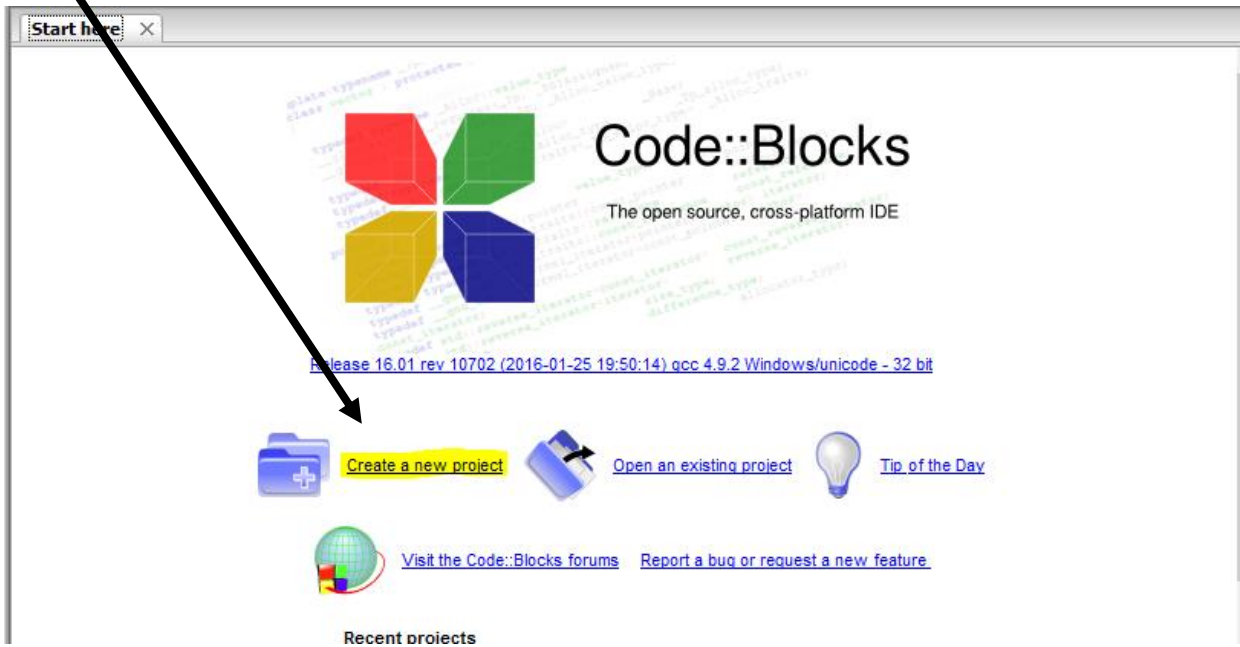
- And click OK.

BOSTON UNIVERSITY

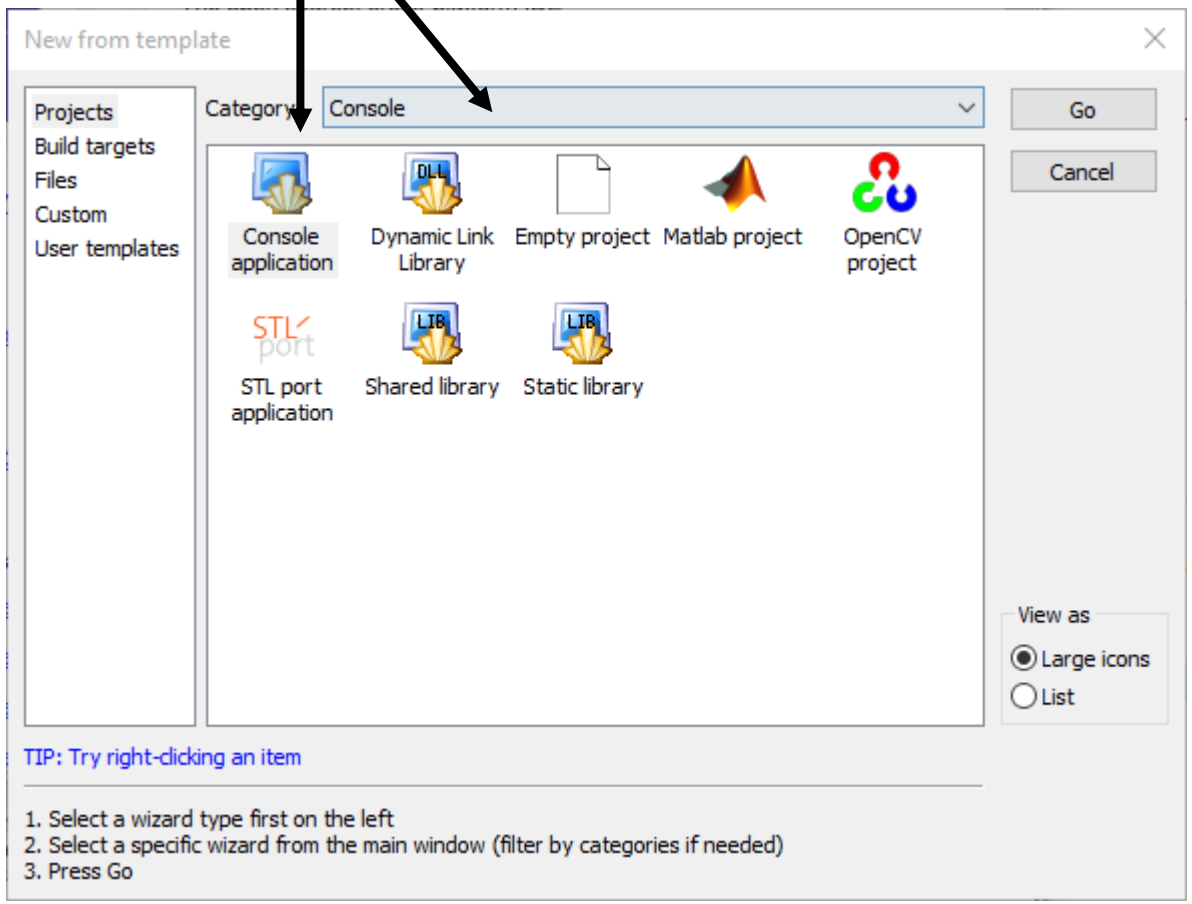
Opening C::B and creating a 1st C++ project…

- Step 1. Create a project from the File menu or the Start Here tab:



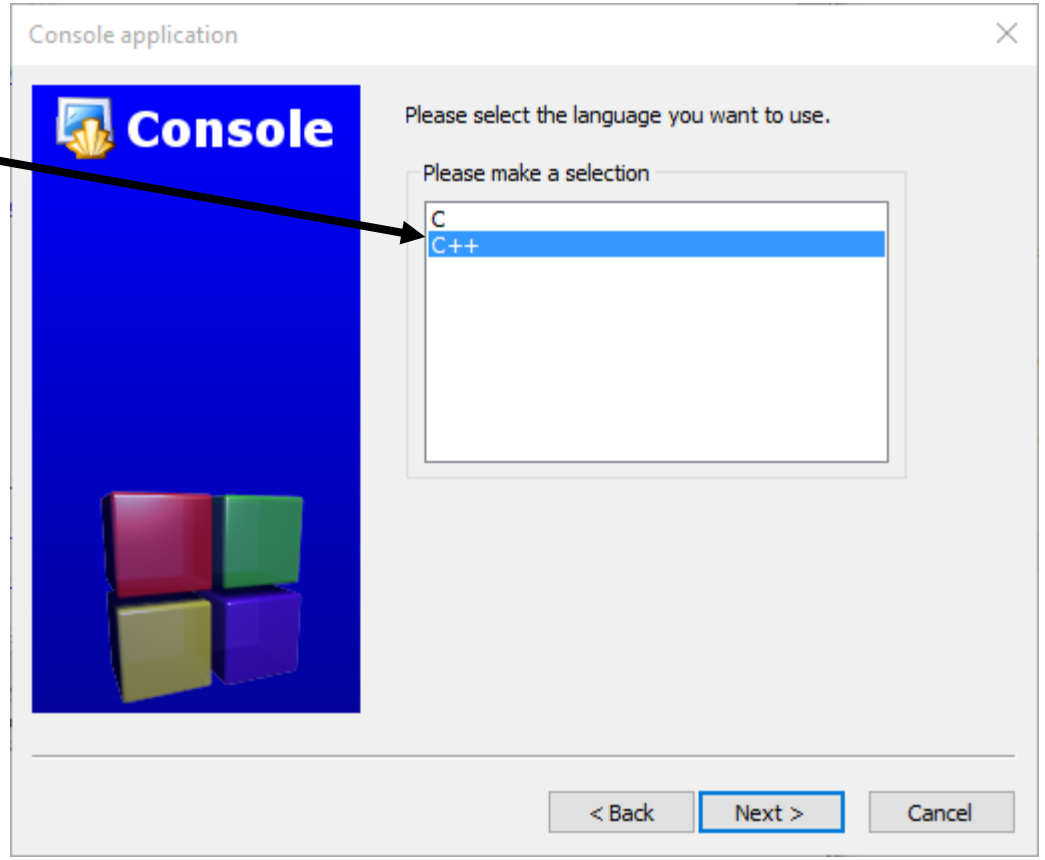


- Step 2. Choose the Console category and then the Console application and click Go.



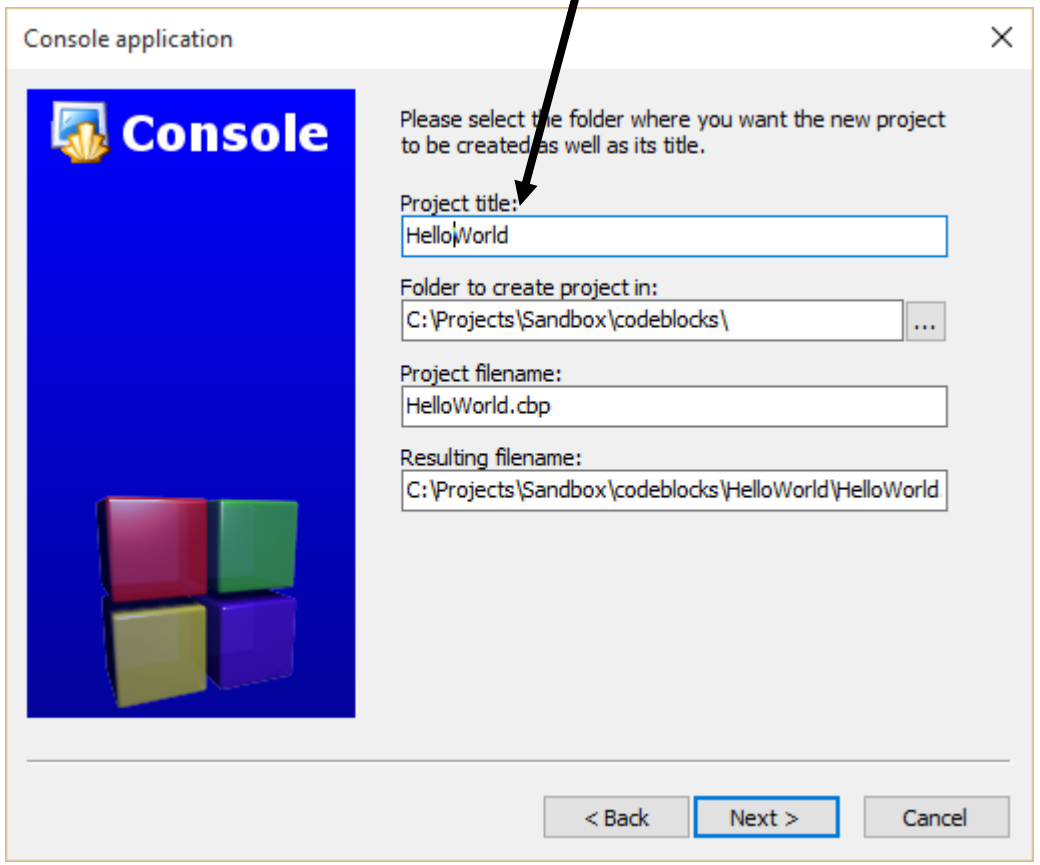
BOSTON UNIVERSITY

- Step 3: Click Next on the “Welcome to the new console application wizard!” screen.
- Step 4: Choose C++!
- …then click Next.

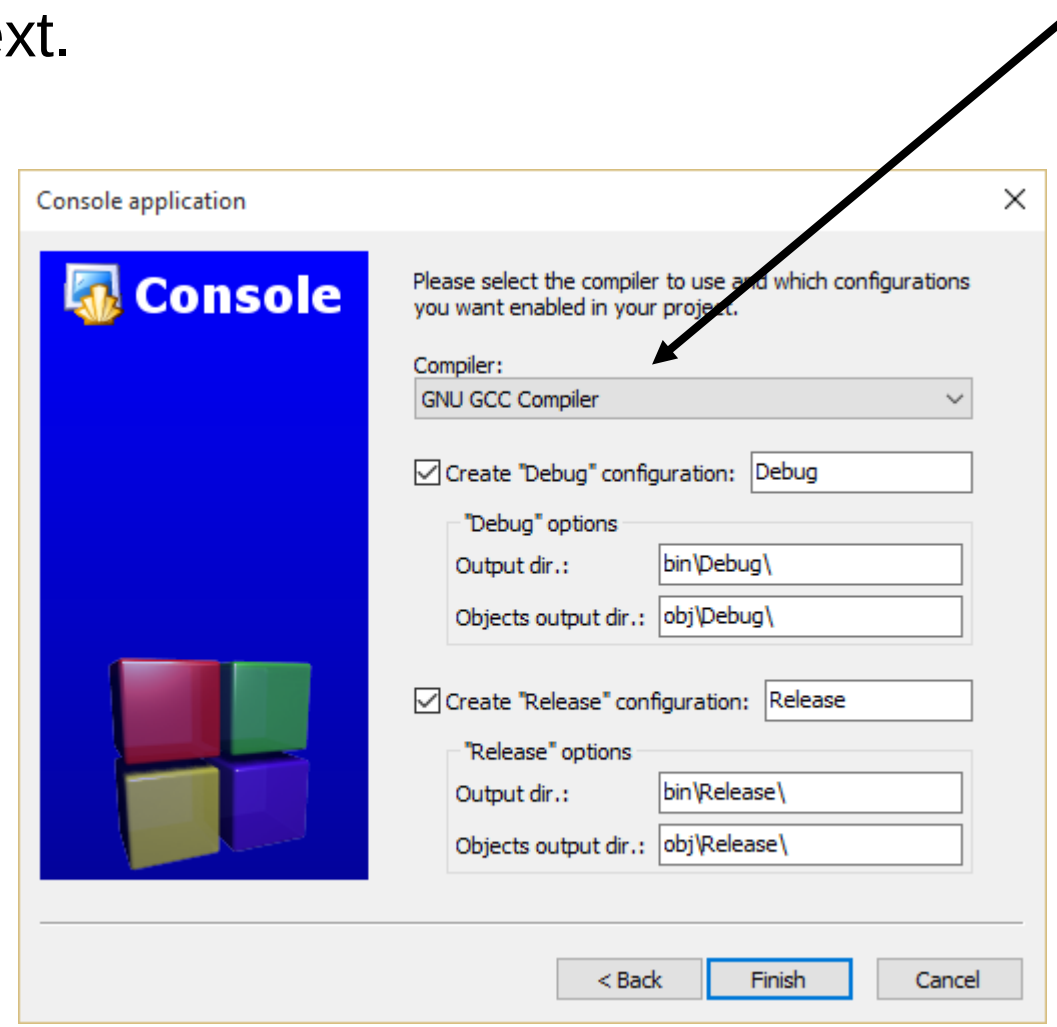


BOSTON UNIVERSITY

- Step 5. Enter a project title. Let C::B fill in the other fields for you. If you like you can change the default folder to hold the project. Click Next.

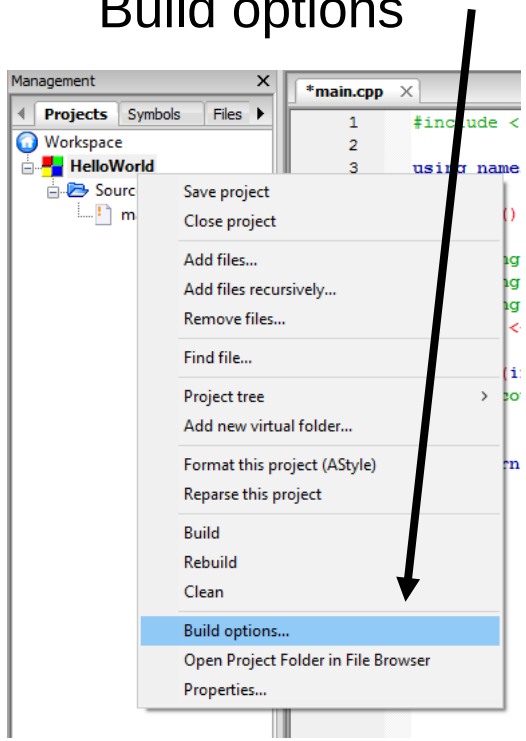


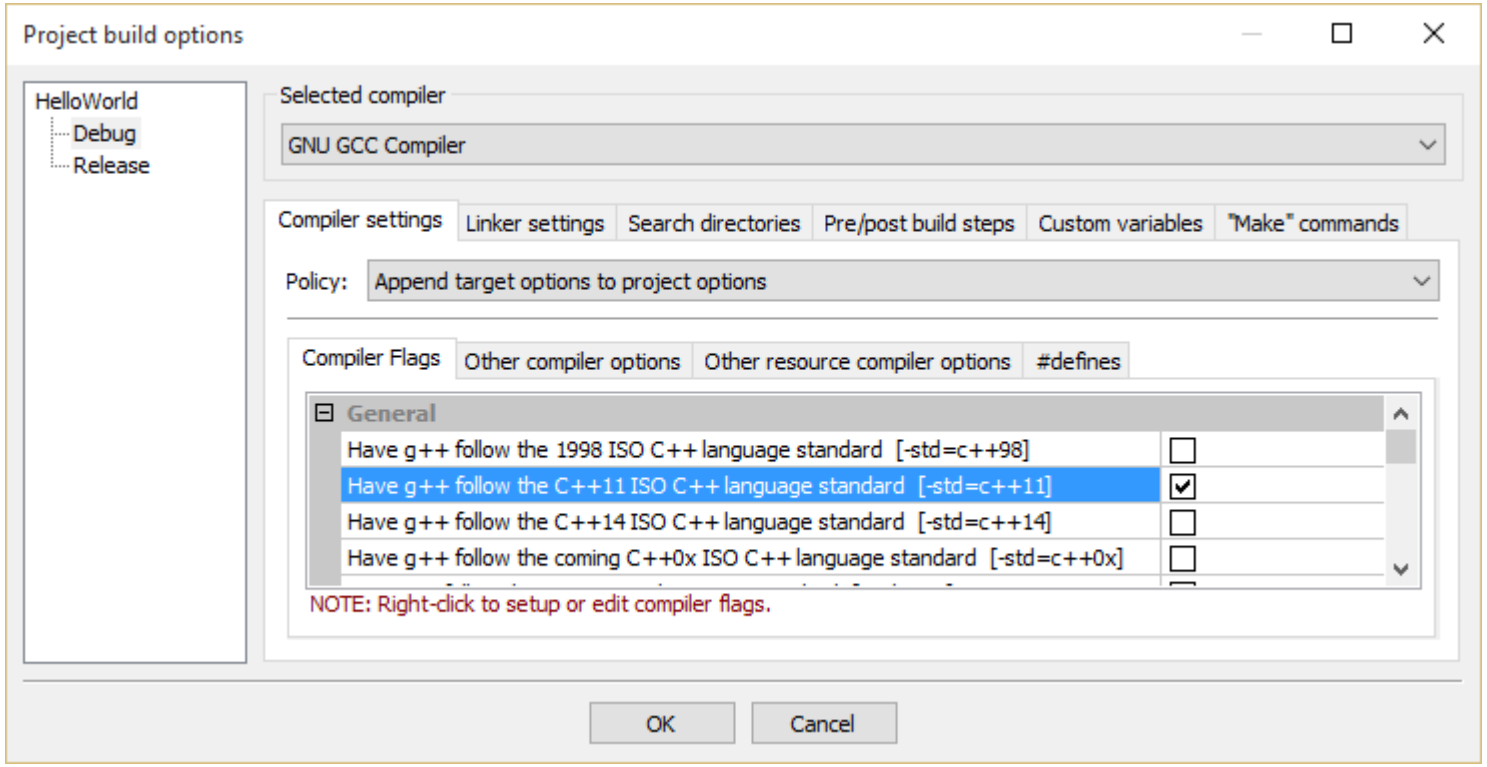
- Step 6: Choose the compiler. For this tutorial, choose GNU GCC as the compiler. Click Next.



Enable C++11 standard

- Step 7.l Right-click on your project name and choose Build options

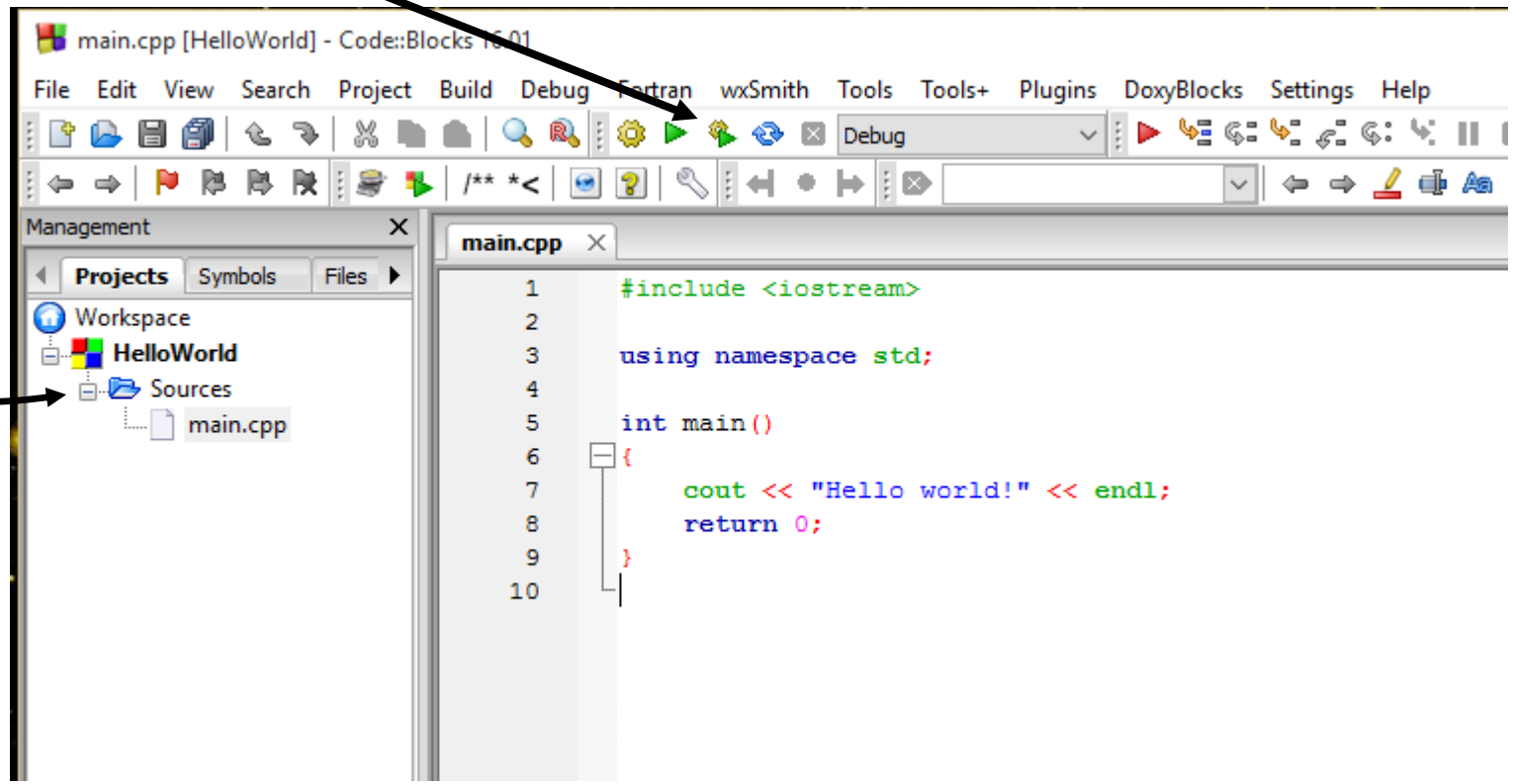




- Check off the C++11 option. Click *Release* on the left and do the same there as well.
- Do this anytime we create a project in C::B

BOSTON UNIVERSITY

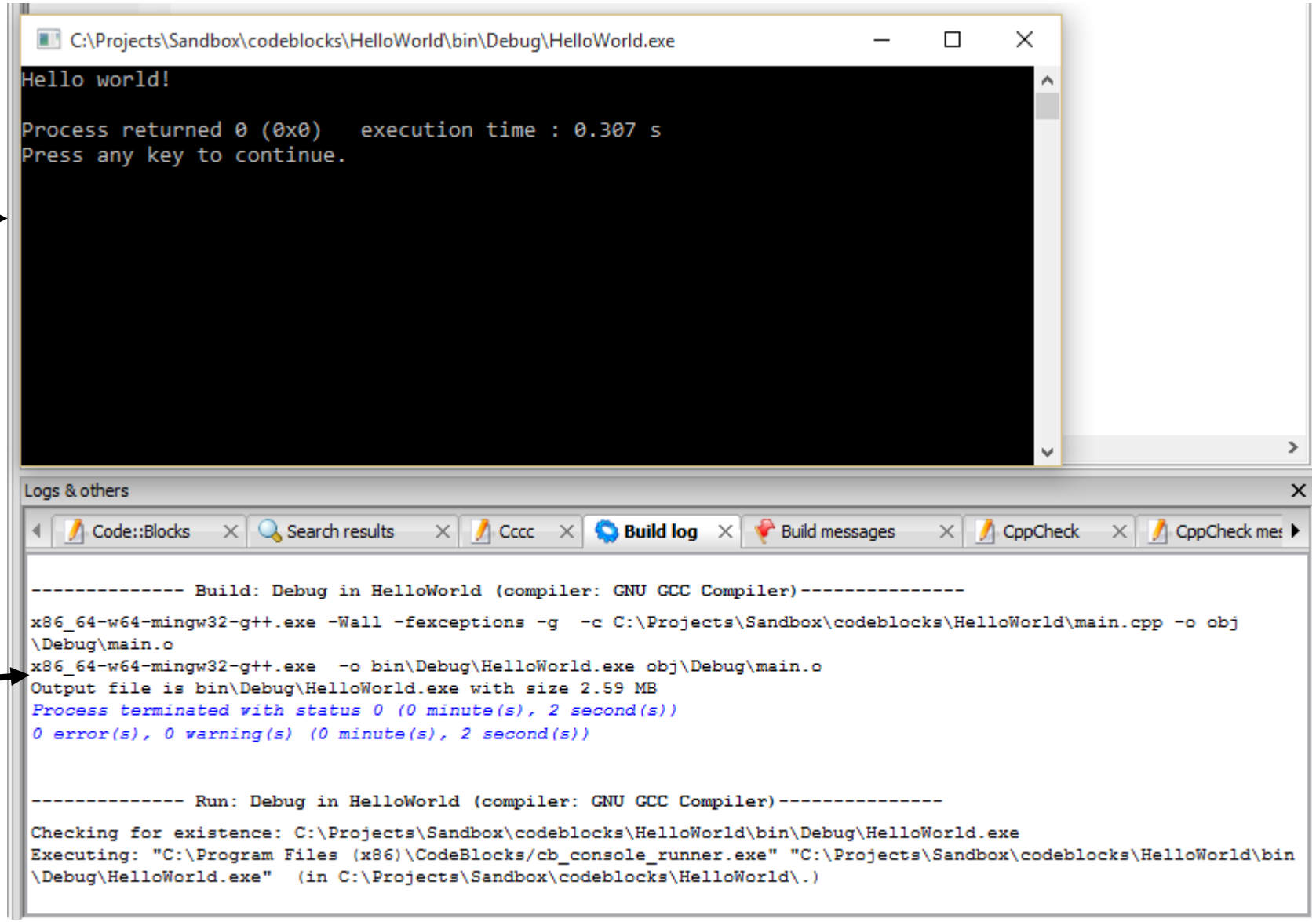
- Step 8: Your project is now created! Click on Sources in the left column, then double-click *main.cpp*.
- Click the  icon in the toolbar or press F9 to compile and run the program.



BOSTON UNIVERSITY

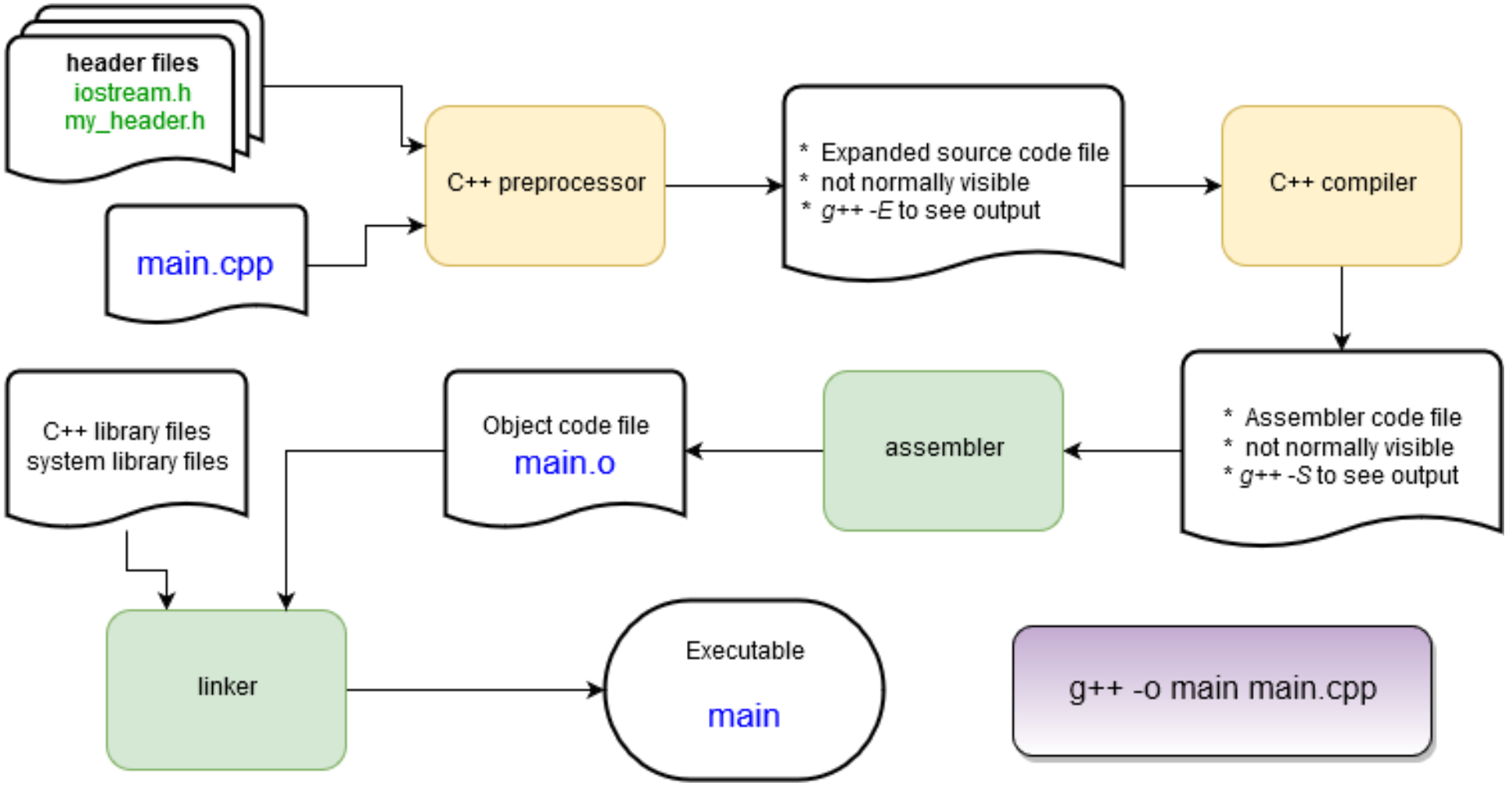
Hello, World!

- Console window:
- Build and compile messages



BOSTON UNIVERSITY

Behind the Scenes: The Compilation Process



Hello, World! explained

main.cpp

```
1   #include <iostream>
2
3   using namespace std;
4
5   int main()
6   {
7       cout << "Hello world!" << endl;
8       return 0;
9   }
10
```

The *main* routine – the start of every C++ program! It returns an integer value to the operating system and takes no arguments ().

Statement that returns an integer value to the operating system after completion. 0 means “no error”

Hello, World! explained

main.cpp

```
1   #include <iostream>
2
3   using namespace std;
4
5   int main()
6   {
7       cout << "Hello world!" << endl;
8       return 0;
9   }
10
```

loads a *header* file containing function and class definitions

Loads a *namespace* called *std*. Namespaces are used to separate sections of code for programmer convenience. To save typing we'll always use this line in this tutorial.

cout is the *object* that writes to the stdout device, i.e. the console window. It is part of the C++ standard library. Without the "using namespace std;" line this would have been called as *std::cout*. It is defined in the *iostream* header file.

<< is the C++ *insertion operator*. It is used to pass characters from the right to the object on the left. *endl* is the C++ newline character.

Slight change

- Let's put the message into some variables of type *string* and print some numbers.
- Things to note:
  - Strings can be concatenated with a + operator.
  - No messing with null terminators as in C
- Some string notes:
  - Access a string character by brackets or function:
    - msg[0] → "H" or msg.at(0) → "H"
    - C++ strings are *mutable* – they can be changed in place.
- Press F9 to recompile & run.

```
#include <iostream>

using namespace std;

int main()
{
    string hello = "Hello";
    string world = "world!";
    string msg = hello + " " + world ;
    cout <<  msg << endl;
    msg[0] = 'h';
    cout <<  msg << endl;
    return 0;
}
```


Basic Syntax

- C++ syntax is very similar to C, Java, or C#. Here's a few things up front and we'll cover more as we go along.
- Curly braces are used to denote a code block:

```
{ … some code … }
```

- Statements end with a semicolon:

```
int a ;
a = 1 + 3 ;
```

- Comments are marked for a single line with a *//* or for multilines with a pair of */** and ***/* :

```
// this is a comment.
/* everything in here
        is a comment */
```

- Variables can be declared at any time in a code block.

```
void my_function() {
   int a ;
   a=1 ;
   int b;
}
```

- Functions are sections of code that are called from other code. Functions always have a return argument type, a function name, and then a list of arguments:

```
int my_function(int x) {
    return x ;
}
```

```
// No arguments? Still need ()
void my_function() {
        /* do something...
           but a void value means the
           return statement can be skipped.*/
}
```

- Variables are declared with a type and a name:

```
// Usually enter the type
int x = 100;
float y;
vector<string> vec ;
// Sometimes it can be inferred
auto z = x;
```

- A sampling of Operators:
  - Arithmetic: + - * / % ++ --
  - Logical: && (AND) ||(OR) !(NOT)
  - Comparison: == > < >= <= !=

Built-in (aka primitive or intrinsic) Types

- “primitive” or “intrinsic” means these types are not objects
- Here are the most commonly used types.
- Note: The exact bit ranges here are **platform and compiler dependent**!
  - Typical usage with PCs, Macs, Linux, etc. use these values
  - Variations from this table are found in specialized applications like embedded system processors.

Name	Name	Value
char	unsigned char	8-bit integer
short	unsigned short	16-bit integer
int	unsigned int	32-bit integer
long	unsigned long	64-bit integer
bool		true or false

Name	Value
float	32-bit floating point
double	64-bit floating point
long long	128-bit integer
long double	128-bit floating point

http://www.cplusplus.com/doc/tutorial/variables/

Need to be sure of integer sizes?

- In the same spirit as using *integer(kind=8)* type notation in Fortran, there are type definitions that exactly specify exactly the bits used. These were added in C++11.
- These can be useful if you are planning to port code across CPU architectures (ex. Intel 64-bit CPUs to a 32-bit ARM on an embedded board) or when doing particular types of integer math.
- For a full list and description see: http://www.cplusplus.com/reference/cstdint/

#include <cstdint>

Name	**Name**	**Value**
int8_t	uint8_t	8-bit integer
int16_t	uint16_t	16-bit integer
int32_t	uint32_t	32-bit integer
int64_t	uint64_t	64-bit integer

Type Casting

- C++ is strongly typed. It will auto-convert a variable of one type to another in a limited fashion: if it will not change the value.

```
short x = 1 ;
int y = x ;    // OK
short z = y ; // NO!
```

- Conversions that don't change value: increasing precision (float → double) or integer → floating point of at least the same precision.
- C++ allows for C-style type casting with the syntax: (new type) expression

```
double x = 1.0 ;
int y = (int) x ;
float z = (float) (x / y) ;
```

- In addition to this C++ offers 4 different variations in a C++ style.

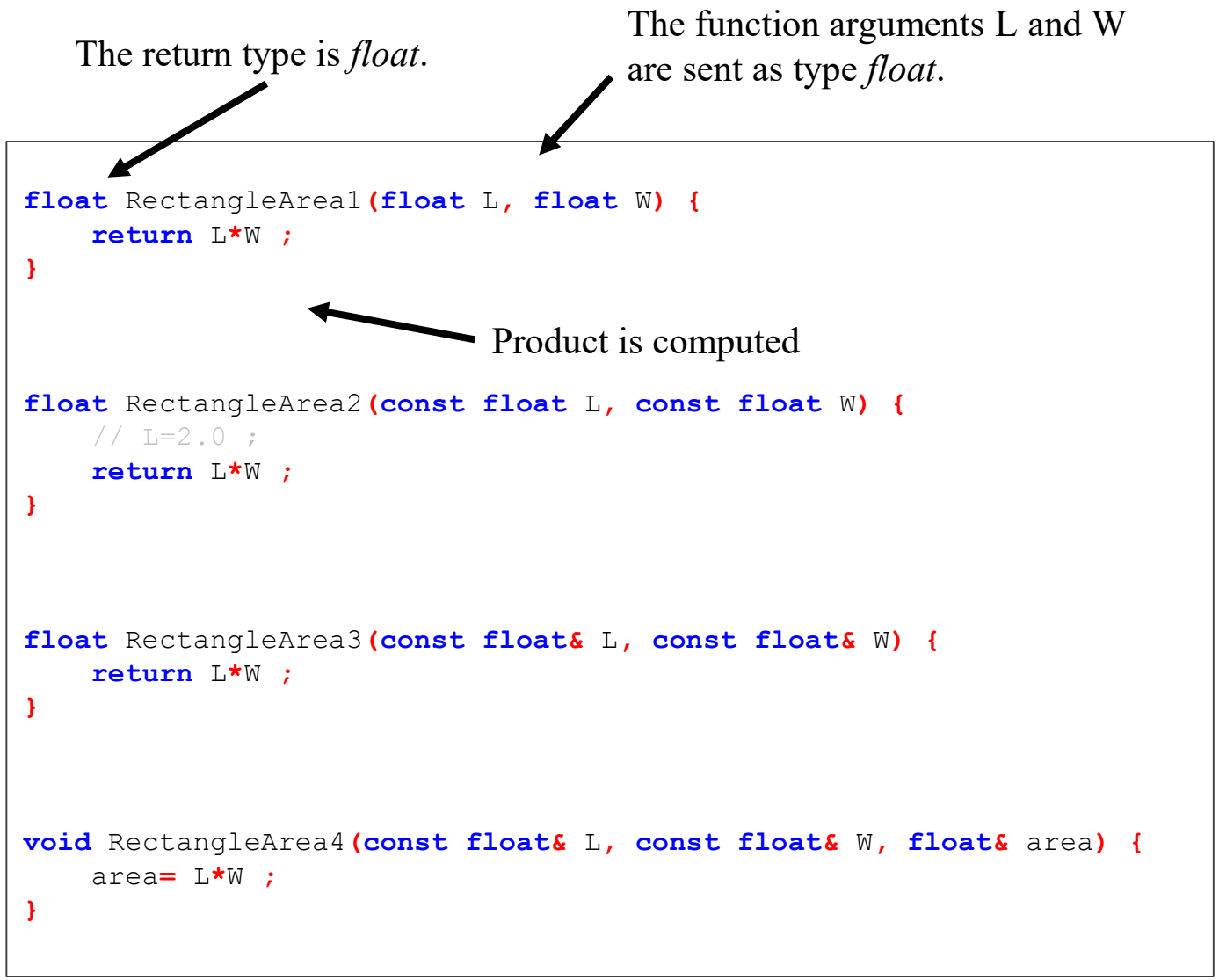
Type Casting

- `static_cast<new type>( expression )`
  - This is exactly equivalent to the C style cast.
  - This identifies a cast at compile time and the compiler inserts the CPU type conversion instructions for primitive types.
  - Can do casting that reduces precision (ex. double → float)
- `dynamic_cast<new type>( expression)`
  - Special version where type casting is performed at runtime, only works on reference or pointer type variables.
- `const_cast<new type>( expression )`
  - Variables labeled as *const* can't have their value changed.
  - const_cast lets the programmer remove or add *const* to reference or pointer type variables.
- `reinterpret_cast<new type>( expression )`
  - Takes the bits in the expression and re-uses them **unconverted** as a new type. Also only works on reference or pointer type variables.

"unsafe": the compiler will not protect you here.

Functions

- Open the project “FunctionExample” in C::B files
  - Compile and run it!
- Open main.cpp
- 4 function calls are listed.
- The 1st and 2nd functions are identical in their behavior.
  - The values of L and W are sent to the function, multiplied, and the product is returned.
- RectangleArea2 uses *const* arguments
  - The compiler **will not** let you modify their values in the function.
  - Try it! Uncomment the line and see what happens when you recompile.
- The 3rd and 4th versions pass the arguments by *reference* with an added *&*

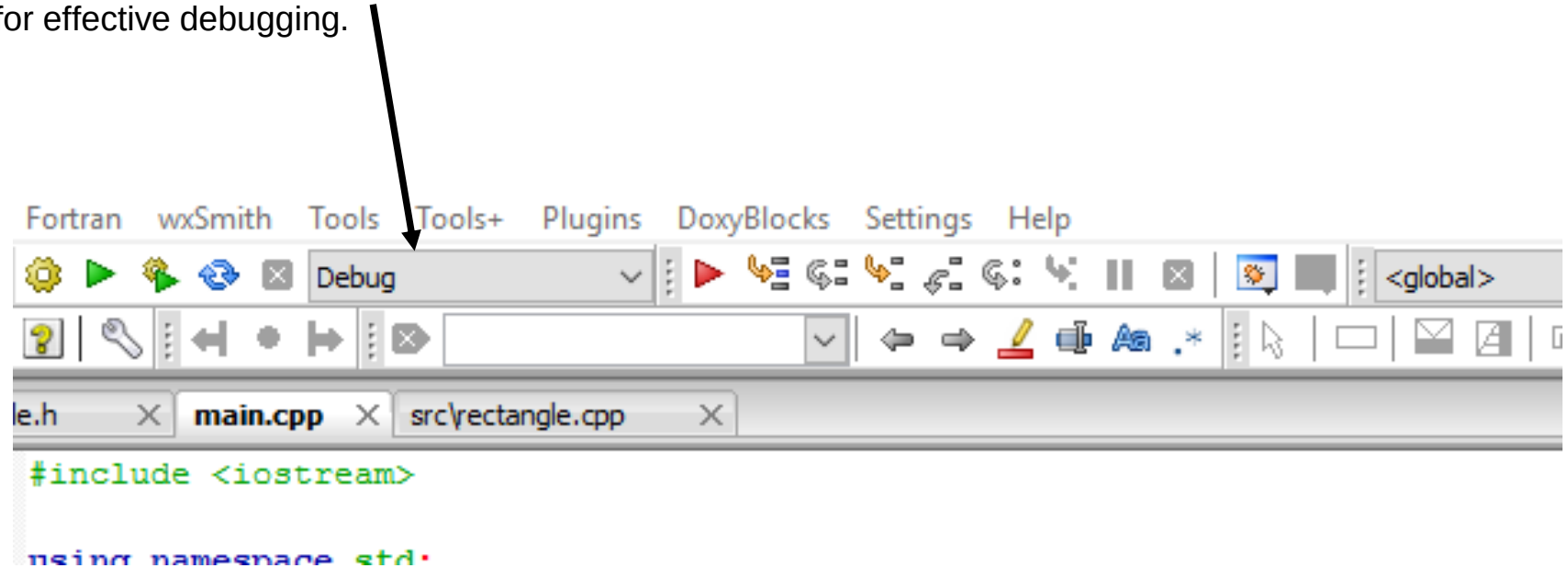



```
float RectangleArea1(float L, float W) {
    return L*W ;
}

float RectangleArea2(const float L, const float W) {
    // L=2.0 ;
    return L*W ;
}

float RectangleArea3(const float& L, const float& W) {
    return L*W ;
}

void RectangleArea4(const float& L, const float& W, float& area) {
    area= L*W ;
}
```

BOSTON UNIVERSITY

Using the C::B Debugger

- To show how this works we will use the C::B interactive debugger to step through the program line-by-line to follow the function calls.
- Make sure you are running in *Debug* mode. This turns off compiler optimizations and has the compiler include information in the compiled code for effective debugging.



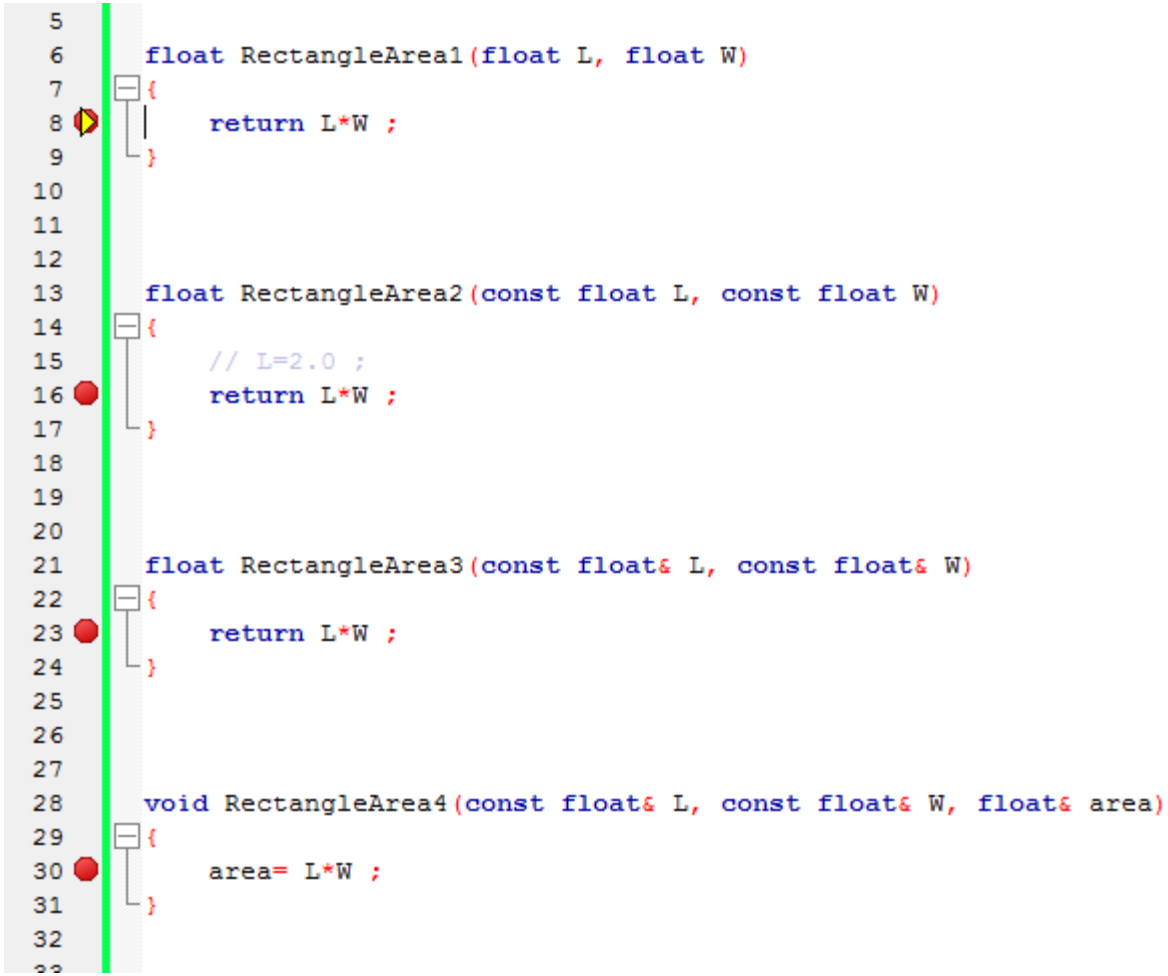
Add a Breakpoint

- Breakpoints tell the debugger to halt at a particular line so that the state of the program can be inspected.
- In main.cpp, double click to the left of the lines in the functions to set a pair of breakpoints. A red dot will appear.
- Click the red arrow to start the code in the debugger.


```
1  #include <iostream>
2
3  using namespace std;
4
5
6  float RectangleArea1(float L, float W)
7  {
8      return L*W ;
9  }
10
11
12
13 float RectangleArea2(const float L, const float W)
14 {
15     // L=2.0 ;
16     return L*W ;
17 }
18
19
20
21 float RectangleArea3(const float& L, const float& W)
22 {
23     return L*W ;
24 }
25
26
27
28 void RectangleArea4(const float& L, const float& W, float& a
29 {
30     area= L*W ;
31 }
32
33
```

BOSTON UNIVERSITY

- The debugger will pause in the first function at the breakpoint.



```
5
6   float RectangleArea1(float L, float W)
7   {
8       return L*W ;
9   }
10
11
12
13  float RectangleArea2(const float L, const float W)
14  {
15      // L=2.0 ;
16      return L*W ;
17  }
18
19
20
21  float RectangleArea3(const float& L, const float& W)
22  {
23      return L*W ;
24  }
25
26
27
28  void RectangleArea4(const float& L, const float& W, float& area)
29  {
30      area= L*W ;
31  }
32
```

- Click the Debug menu, go to Debugging Windows, and choose *Call Stack*. Drag it to the right, then go back and choose *Watches*. Drag it to the right. Do the same for the *Breakpoints* option. Your screen will look something like this now…
- Controls (hover mouse over for help):

Place the cursor in the function, click to run to the cursor

Run the next line

Step into a function call

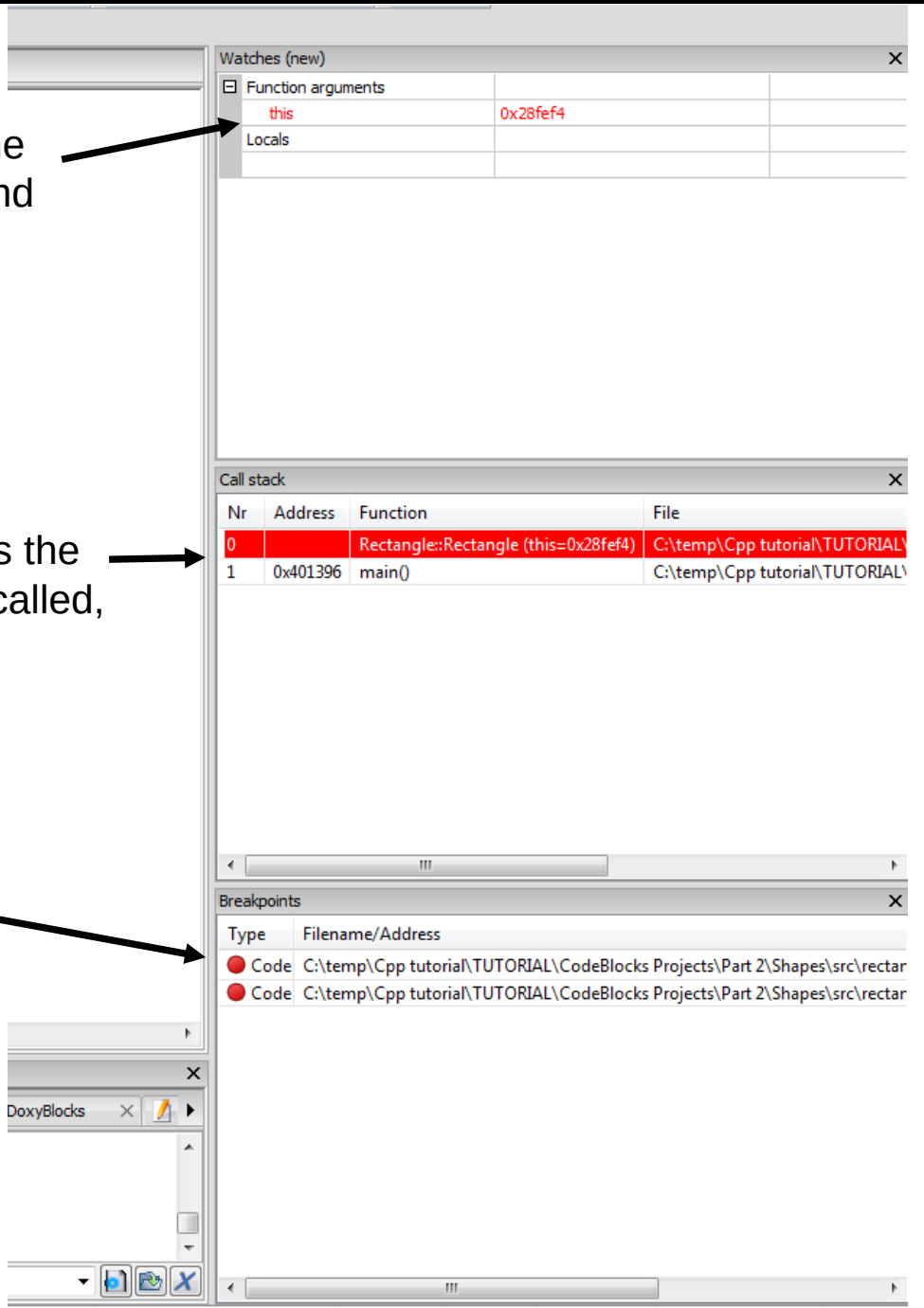
Step out of a function to the calling function.

Step by CPU instruction. Less useful, generally.

Watches shows the variables in use and their values

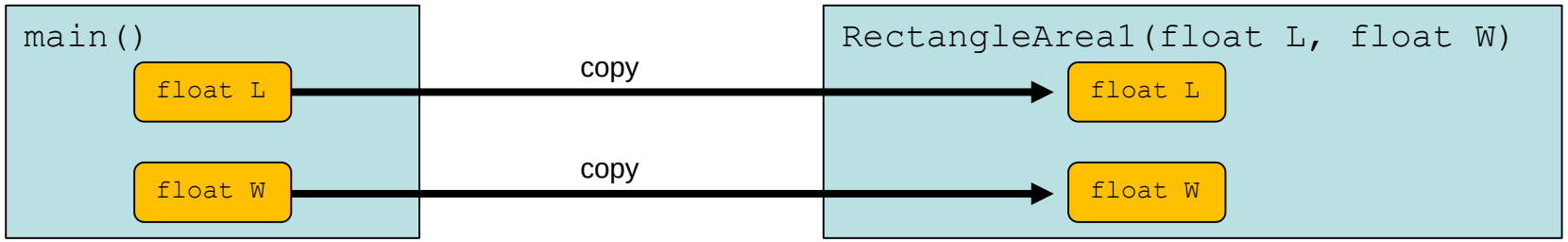
Call Stack shows the functions being called, newest on top.

Breakpoints lists the breakpoints you've created.



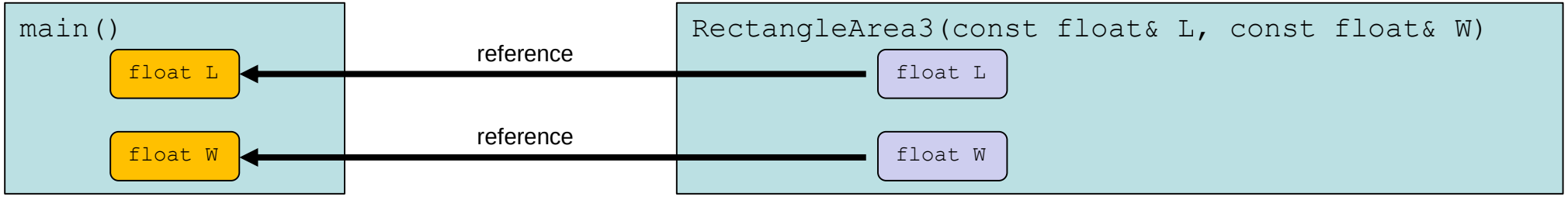
BOSTON UNIVERSITY

Pass by Value



- C++ defaults to *pass by value* behavior when calling a function.
- The function arguments are **copied** when used in the function.
- Changing the value of L or W in the RectangleArea1 function does **not** effect their original values in the main() function
- When passing objects as function arguments it is important to be aware that potentially large data structures are automatically copied!

Pass by Reference



- *Pass by reference* behavior is triggered when the & character is used to modify the type of the argument.
- Pass by reference function arguments are **NOT** copied. Instead the compiler sends a *pointer* to the function that references the memory location of the original variable. The syntax of using the argument in the function does not change.
- Pass by reference arguments almost always act just like a pass by value argument when writing code **EXCEPT** that changing their value changes the value of the original variable!!
- The *const* modifier can be used to prevent changes to the original variable in main().

void does not return a value.

```
void RectangleArea4(const float& L, const float& W, float& area) {
    area= L*W ;
}
```

- In RectangleArea4 the pass by reference behavior is used as a way to return the result without the function returning a value.
- The value of the *area* argument is modified in the main() routine by the function.
- This can be a useful way for a function to return multiple values in the calling routine.

- In C++ arguments to functions can be objects…which can contain any quantity of data you've defined!
  - Example: Consider a string variable containing 1 million characters (approx. 1 MB of RAM).
    - Pass by value requires a copy – 1 MB.
    - Pass by reference requires 8 bytes!
- Pass by value could potentially mean the accidental copying of large amounts of memory which can greatly impact program memory usage and performance.
- When passing by reference, use the *const* modifier whenever appropriate to protect yourself from coding errors.
  - Generally speaking – use *const* anytime you don't want to modify function arguments in a function.

"C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off." – Bjarne Stroustrop

A first C++ class

- You can start a new project in C::B or just modify the Hello World! code.
- In the main.cpp, we'll define a class called BasicRectangle
- First, just the basics: length and width
- Enter the code on the right before the main() function in the main.cpp file (copy & paste is fine) and create a BasicRectangle object in main.cpp:

```
#include <iostream>

using namespace std;

class BasicRectangle
{
public:
    // width ;
    float W ;
    // length
    float L ;
};

int main()
{
    cout << "Hello world!" << endl;

    BasicRectangle rectangle ;
    rectangle.W = 1.0 ;
    rectangle.L = 2.0 ;

    return 0;
}
```

BOSTON UNIVERSITY

Basic C++ Class Syntax

class keyword

Name of class

Curly braces at the beginning and end followed by a semi-colon

public keyword indicates everything following the keyword is accessible by any other code outside of this class.

Internal variables are called *members*

```
class BasicRectangle
{
public:
    // width ;
    float W ;
    // length
    float L ;
};
```

The class can now be used to declare an object named *rectangle*. The width and length of the rectangle can be set.

```
BasicRectangle rectangle ;
rectangle.W = 1.0 ;
rectangle.L = 2.0 ;
```

BOSTON UNIVERSITY

Accessing data in the class

- Public members in an object can be accessed (for reading or writing) with the syntax:

  object.member

- Next let's add a function inside the object (called a *method*) to calculate the area.

```
int main()
{
    cout << "Hello world!" << endl;

    BasicRectangle rectangle ;
    rectangle.W = 1.0 ;
    rectangle.L = 2.0 ;

    return 0;
}
```

method Area does not take any arguments, it just returns the calculation based on the object members.

Methods are accessed just like members:

object.method(arguments)

```
class BasicRectangle
{
public:
    // width ;
    float W ;
    // length
    float L ;
    float Area() {
        return W * L ;
    }

};

int main()
{
    cout << "Hello world!" << endl;

    BasicRectangle rectangle ;
    rectangle.W = 21.0 ;
    rectangle.L = 2.0 ;

    cout << rectangle.Area() << endl ;

    return 0;
}
```

BOSTON UNIVERSITY

Basic C++ Class Summary

- C++ classes are defined with the keyword *class* and must be enclosed in a pair of curly braces **plus a semi-colon**:

  class ClassName { …. } **;**

- The *public* keyword is used to mark members (variables) and methods (functions) as accessible to code outside the class.

- The combination of data and the functions that operate on it is the OOP concept of *encapsulation.*

Encapsulation in Action

- In C – calculate the area of a few shapes…

```
/* assume radius and width_square are assigned
   already ; */
float a1 = AreaOfCircle(radius) ;  // ok
float a2 = AreaOfSquare(width_square) ; // ok
float a3 = AreaOfCircle(width_square) ; // !! OOPS
```

- In C++ with Circle and Rectangle classes…not possible to miscalculate.
  - Well, provided the respective Area() methods are implemented correctly!

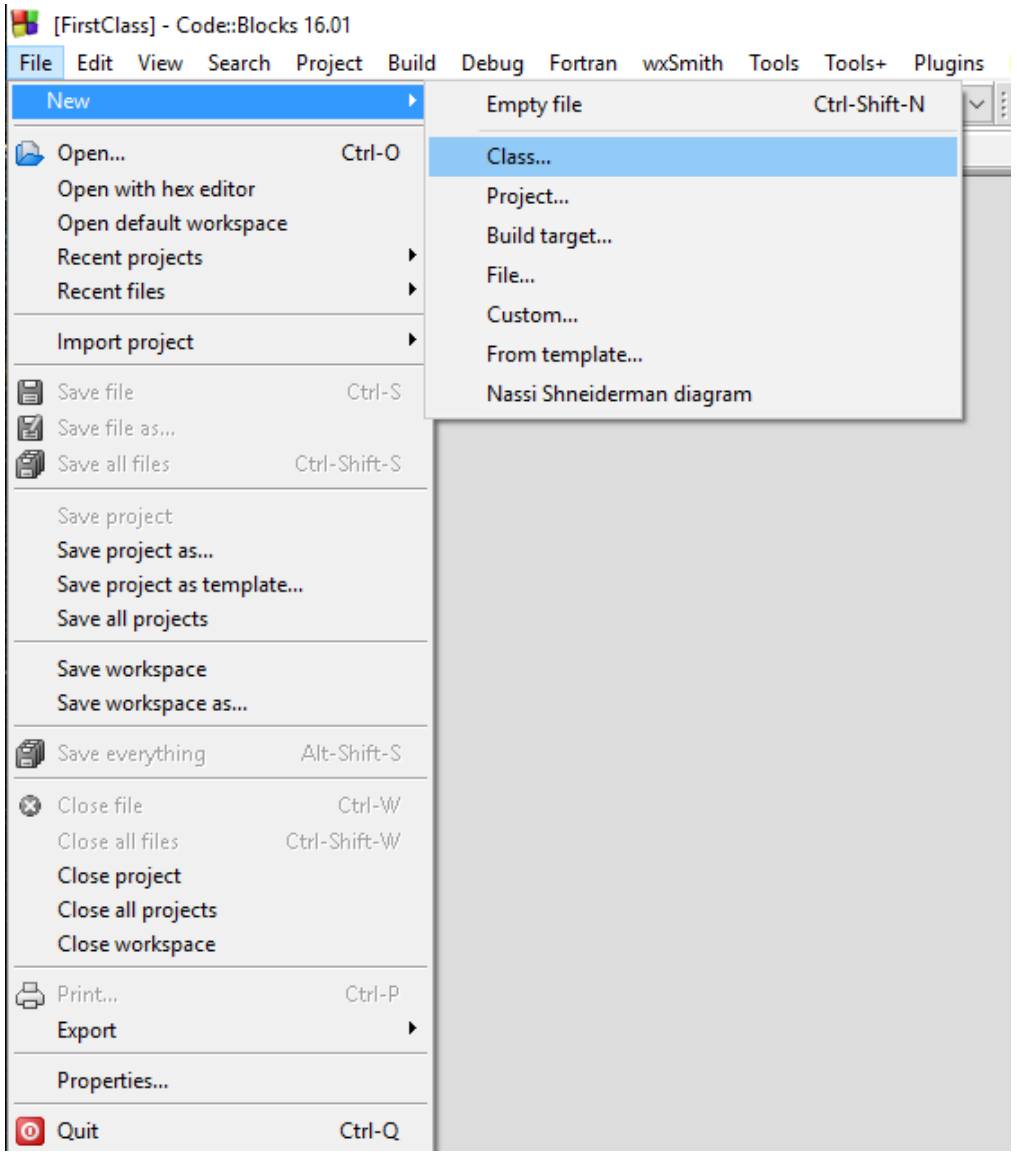
```
Circle c1 ;
Rectangle r1 ;
// ... assign radius and width ...
float a1 = c1.Area() ;
float a2 = r1.Area() ;
```

Now for a “real” class

- Defining a class in the main.cpp file is not typical.
- Two parts to a C++ class:
  - Header file (my_class.h)
    - Contains the interface (definition) of the class – members, methods, etc.
    - The interface is used by the compiler for type checking, enforcing access to private or protected data, and so on.
    - Also useful for programmers when *using* a class – no need to read the source code, just rely on the interface.
  - Source file (my_class.cc)
    - Compiled by the compiler.
    - Contains implementation of methods, initialization of members.
  - In some circumstances there is no source file to go with a header file.

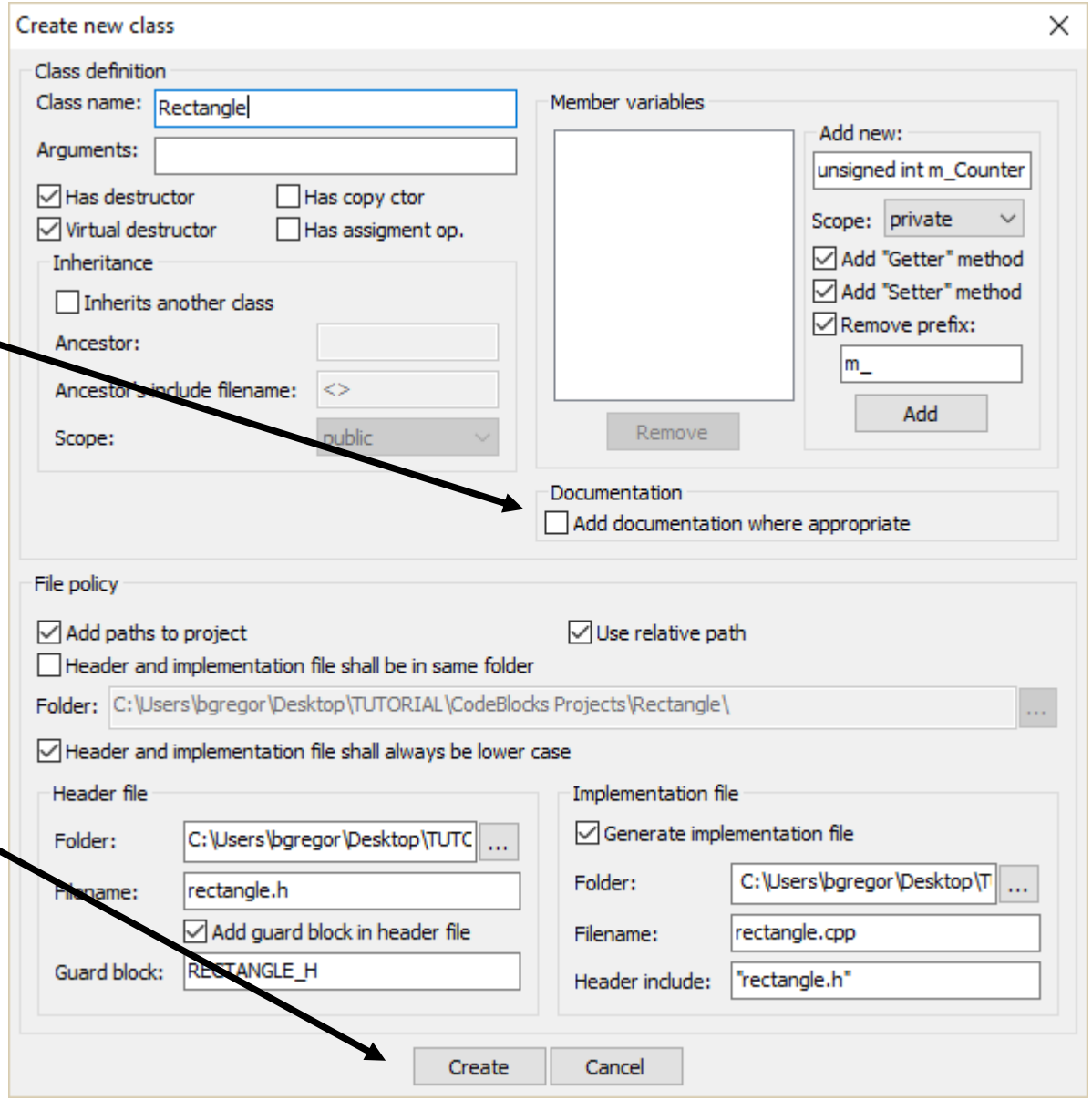
Create a new class in C::B

- Using an IDE is especially convenient in C++ to keep the details straight when defining a class
  - Typically header and source files are needed
  - Some default methods are created to save you some typing
- Create another project in C::B, call it **Shapes**.
- Once open, go the File menu, click New, then click Class.
- This opens a wizard GUI that helps get things started.
- This will create the header and source files for you.



BOSTON UNIVERSITY

- Name the class *Rectangle*
- Uncheck the Documentation option
  - This will just confuse things for now
- Click Create!



BOSTON UNIVERSITY

rectangle.h

```
#ifndef RECTANGLE_H
#define RECTANGLE_H


class Rectangle
{
    public:
        Rectangle();
        virtual ~Rectangle();

    protected:

    private:
};

#endif // RECTANGLE_H
```

rectangle.cpp

```
#include "rectangle.h"

Rectangle::Rectangle()
{
    //ctor
}

Rectangle::~Rectangle()
{
    //dtor
}
```

- 2 files are automatically generated: rectangle.h and rectangle.h.cpp

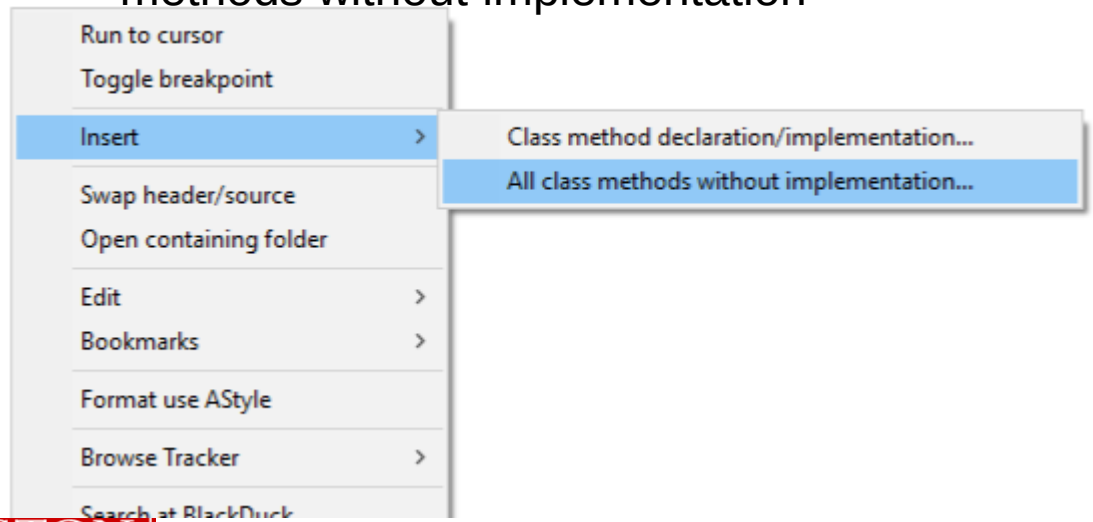
Modify rectangle.h

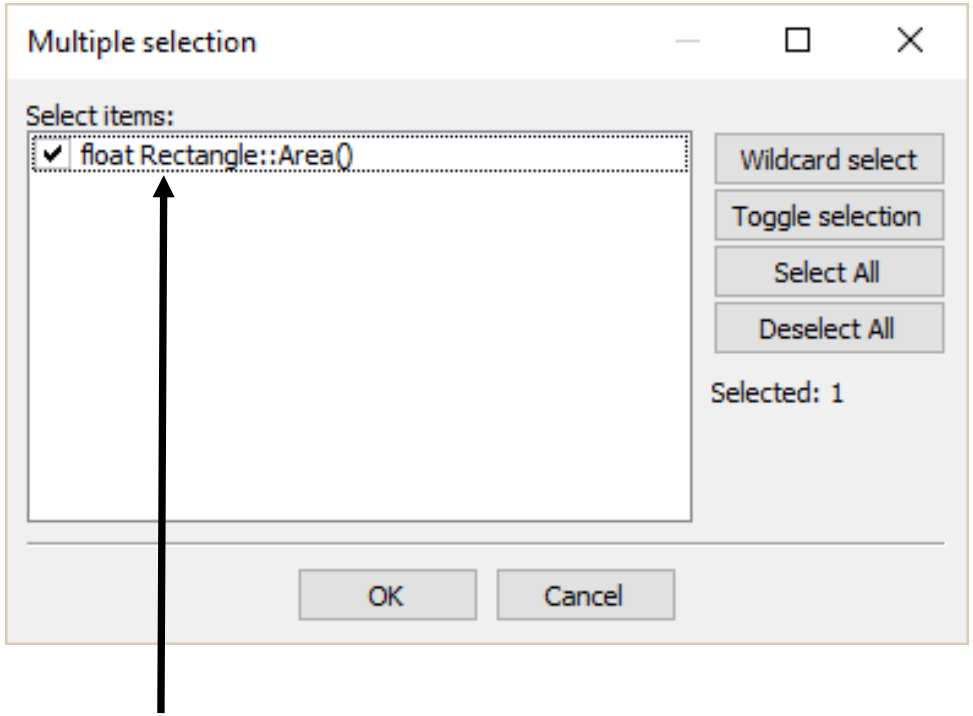
- As in the sample *BasicRectangle*, add storage for the length and width to the header file. Add a *declaration* for the Area method.
- The *protected* keyword will be discussed later.
- The *private* keyword declares anything following it (members, methods) to be visible only to code **in this class**.

```
#ifndef RECTANGLE_H
#define RECTANGLE_H


class Rectangle
{
    public:
        Rectangle();
        virtual ~Rectangle();

        float m_length ;
        float m_width ;

        float Area() ;
    protected:

    private:
};
#endif // RECTANGLE_H
```

Modify rectangle.cpp

- This will now contain the code for the Area() method.
- Use the C::B environment to help out here!
- Open rectangle.cpp (under Sources/src)
- Right-click and choose Insert/All class methods without implementation





- The Area() method is automatically found from the header file.
- Click OK.

rectangle.cpp

- The Area() method now has a basic definition added.
- The syntax:

  class::method

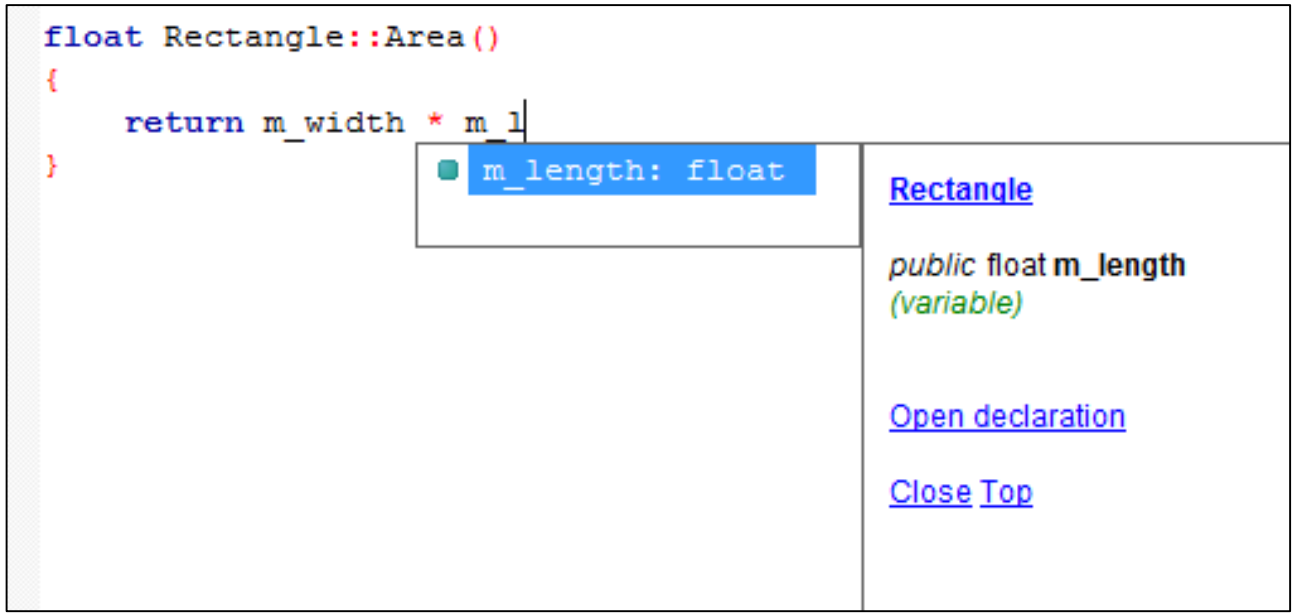
tells the compiler that this is the code for the Area() method declared in rectangle.h

- Now take a few minutes to fill in the code for Area().
  - Hint – look at the code used in BasicRectangle...

```
#include "rectangle.h"

Rectangle::Rectangle()
{
    //ctor
}

Rectangle::~Rectangle()
{
    //dtor
}

float Rectangle::Area()
{

}
```

More C::B assistance

- You may have noticed C::B trying to help when entering the code for Area()
- Press the Tab key to accept the suggestion
- It will offer up variable names, member names, class names, etc. that match what you're typing when appropriate to save you effort.
  - This can be a *huge* convenience when dealing with large code bases.



BOSTON UNIVERSITY

Last Step

- Go to the main.cpp file
- Add an include statement for “rectangle.h”
- Create a Rectangle object in main()
- Add a length and width
- Print out the area using *cout*.

- Hint: just like the BasicRectangle example…

Solution

- You should have come up with something like this:

```
#include <iostream>

using namespace std;

#include "rectangle.h"

int main()
{
    Rectangle rT ;
    rT.m_width = 1.0 ;
    rT.m_length = 2.0 ;

    cout << rT.Area() << endl ;

    return 0;
}
```