

Class 1 Handout

Setting up the Environment

To start working with C++, we are going to install an IDE. An IDE is “Integrated Development Environment”; It includes the text editor, the compiler that turns the C++ code to binary, and tools for debugging and more.

In the class, I will be using Code::Blocks, which is available for Windows, Linux, and OSX. You can also use Microsoft Visual C++ Express 2010 if you would like; it is somewhat more polished and has better autocompletion, but choosing one over the other isn't really going to make-or-break your experience with C++ this early in. Both of these are **free**!

Code::Blocks
<http://www.codeblocks.org/>

Microsoft Visual C++ Express
<http://www.microsoft.com/visualstudio/eng/downloads#d-2010-express>

If you're running a Linux operating system, you may also need to install G++. If you're running Debian, Ubuntu, or Linux Mint, you can do this through Synaptic.

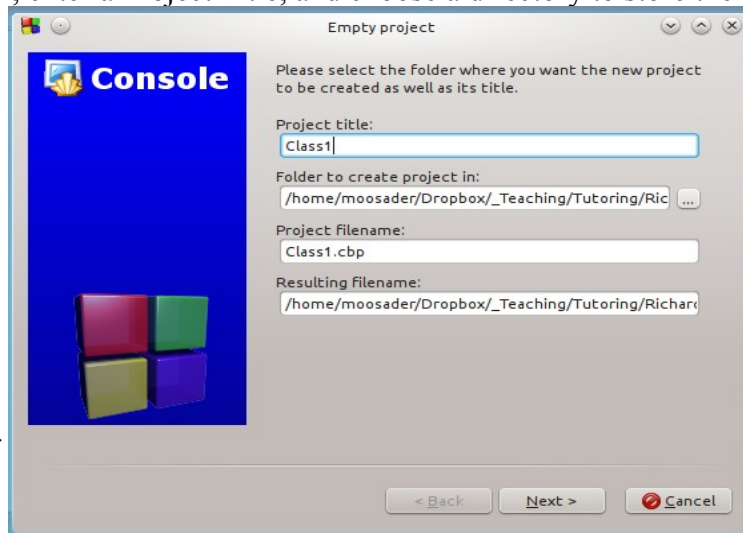
Testing out the IDE with a first program

New Project

Let's write a quick program to make sure the IDE is working properly.

Open **Code::Blocks**, go to **File > New > Project...** Select **Empty Project**.

In the setup dialog box, enter a Project Title, and choose a directory to store the project in.

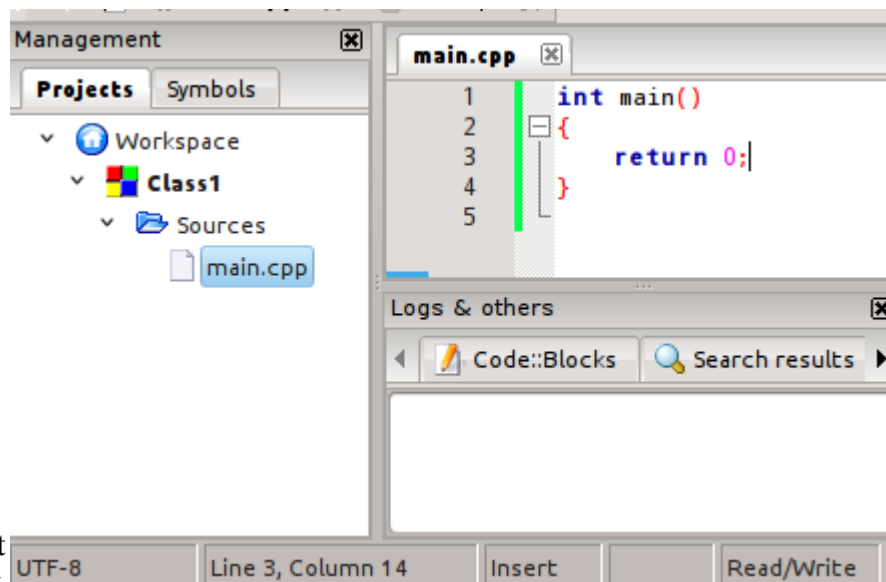


Hit **Next**, then **Finish**.

Creating main

We will now have an empty project. To write any code, we need to create a new Source file.

Go to **File > New > Empty File** (or **CTRL+SHIFT+END**). Name the file whatever you would like, and end it with **“.cpp”**. Make sure to add it to the active project.



In C++,
whenever you
run a program, it
will always look

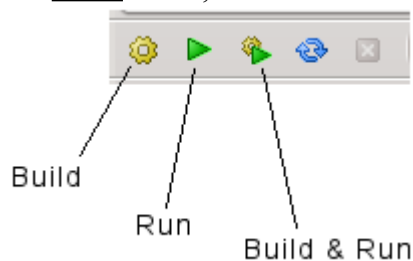
for **main**. This is the starting point, and if you're missing **main**, it will complain, saying “Undefined reference to main”.

We are **returning** a value of zero inside of our main block, and this just signifies that the program ended fine. We can use different numbers as **error codes**, so if something fails to load, we could return 2, 10, 50, or any arbitrary number. That way, if the program crashes and we get an error code, it helps us figure out where the program crashed.

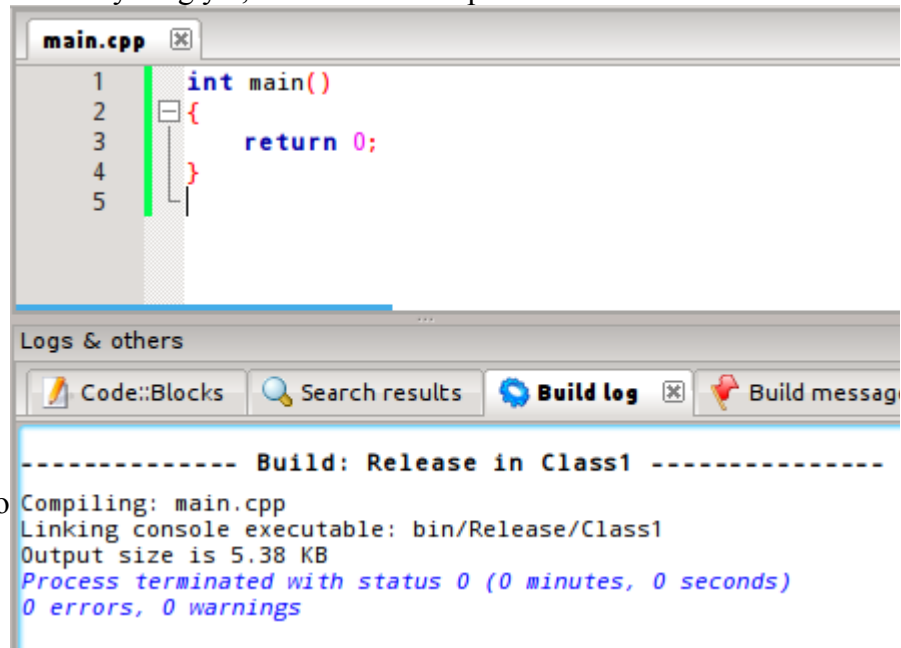
On the same line as **main**, we have **int**. This means integer, and it means that **main** must return an integer. That's what our error-code is.

If we build the program, we should have no problems.

You can use the toolbar icons, go to the Build menu, or use F9 to Build & Run.



The program won't do anything yet, but the build output should show:



Zero errors, Zero warnings.
Let's add some input, output, and math before we move on to concepts.

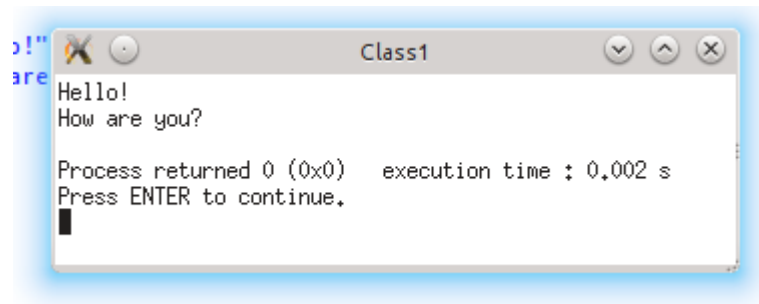
In C++, we use “**cout**” (console out) to output text to the screen, and “**cin**” (console in) to input text from the user's keyboard. However, this won't work right off the bat!

C++ has a **standard library** with a lot of prewritten functionality, but you have to include parts of it at a time (otherwise, your .exe files would end up really big!)

We will use an **#include** statement to add these additional libraries. A **library** is essentially pre-written code, either by you, something from the standard library, or from another third-party.

```
#include <iostream>  
  
int main()  
{  
    std::cout << "Hello!" << std::endl;  
    std::cout << "How are you?" << std::endl;  
  
    return 0;  
}
```

1. On the top line, we are including the standard library for input/output, "IO Stream". Streaming here is similar to how you would *stream music* or *stream video* online. Except in this context, it's streaming information between the keyboard and the program (input), and the program and the monitor (output). IO stands for Input/Output.
2. Then, inside main, we've added a line to output the text, "Hello!"
Because output is part of C++'s **standard library**, **cout** needs to be prefixed with **std::**.
At the end of the line, we have **std::endl**, which will create a new line. If the endl weren't there, the next line of text ("How are you?") would show up on the same line as "Hello".
3. When you run the program, you should see something like:



We can shorten our program a bit by telling it that we're going to use the **std** (standard) library.

```
using namespace std;
```

This will go after any **#include** lines. Now, our program is a little simpler:

```
main.cpp
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      cout << "Hello!" << endl;
7      cout << "How are you?" << endl;
8
9      return 0;
10 }
11
```

Now we can add some input to make things more interesting.

Similar to **cout** for console-output, we use **cin** for console-input. But, to get information from the user, we also need to set up **variables** to store information the program is given.

Variable Declaration looks like:

Datatype Variablename;

A **Data Type** is what a program uses to figure out what kind of data it holds. Some common types are:

- **int** – Integers. Whole numbers, positive, negative, and 0.
- **float** – Floats. 0, Positive & Negative numbers with decimal points.
- **char** – A single character; it could be any symbol such as numbers, letters, !@#\$, etc.
- **string** – A string of *characters*. If you were storing names, it would be in a string.
- **bool** – Booleans. These will only ever have two values: True or False. Handy for asking the program questions to choose behavior, like “Is Car in Drive?”

Let's make the program ask the user for two numbers, and then output the result when added together.

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

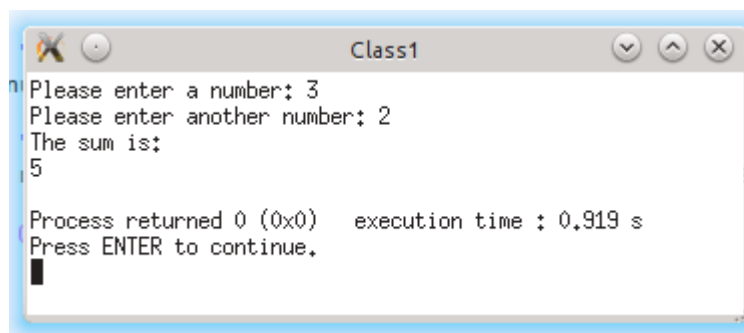
int main()
{
    int number1, number2;

    cout << "Please enter a number: ";
    cin >> number1;

    cout << "Please enter another number: ";
    cin >> number2;

    cout << "The sum is: " << endl;
    cout << number1 + number2 << endl;

    return 0;
}
```



Syntax Notes!

We will get more in-depth into writing code a bit later, but for now I would like review some things with our program:

- In C++, **commands** end with a semi-colon. If you didn't have the semi-colon terminator, it would look at the next line to get additional instructions.
 - **If Statements** and **While Loops** contain commands, so these do not end with semi-colons.
- Statements that contain additional instructions (including **main**), will have curly-braces to signify the beginning and ending of its contents. It is important to keep the curly-braces straight. *It is also standard to indent commands within curly-braces for readability.*
- The **cout** and **cin** commands are followed by << and >>, respectively. These indicate that we're working with *streams*, but it is not important to fully grasp this yet.
- Don't forget to **return 0;** at the end of the program!
- Names given to variables and objects cannot contain spaces or certain keywords. You can separate words *using CamelCaseLikeThis*, or *using underscores_for_spaces*.

Comments!

In software code, it is sometimes good to leave comments in your code to help anybody reading (yourself in the future, included!) figure out what the program is doing. You can type comments in C++ in two ways:

While you're trying to figure out the steps of your programs, you might start by adding comments, in English, of what the steps are, and then filling in the code in each “region”.

Types of comments:

```
// Single-line Comment
```

```
/* Multi-line Comment
```

```
My program takes two numbers  
from the user & adds them together.  
*/
```

```
int main()  
{  
    // Get input from user  
    int number1, number2;  
    cout << "Number: ";  
    cin >> number1;  
  
    cout << "Number: ";  
    cin >> number2;  
  
    // Output sum  
    cout << number1 + number2 << endl;  
  
    return 0;  
}
```

Object Oriented Programming

C++, as well as Java, C#, and some other languages, are Object Oriented languages. To solve every-day problems with computers, these languages model real-world Objects and Functionality in code. This is called *abstraction*.

For example, we might want to write a program to keep track of or model the following:

Student and Class database for a college

For a school, we might want to keep track of Classes offered. Classes would have a title, short description, list of prerequisites, an assigned teacher, and a list of students registered for the class. Students would have information like their first and last name, student ID number, address, GPA, a list of completed classes and grades, etc. etc.

We would be able to use functionality like “Add student to class x”, “Drop student from class x”, “Update GPA”, “Add completed class”.

An E-book

If we were abstracting a book into an e-book, the book would have data like a title, author, price, and a list of pages. Each page would have data like the actual text on the page.

The E-Book would have functionality like “Turn page forward”, “Turn page backward”, “Go to page x”, “Save page user is currently on”.

Think of abstraction in terms of Object, Functionality, and Attributes.

Car Object, **Drive forward** Functionality, **Red color** Attribute.

Oven Object, **Set temperature to x** Functionality, **Current temperature** Attribute.

Cat Object, **Sleep** Functionality, **Breed** Attribute.

In C++, we represent **Objects** with the class concept. Classes take the form:

```
class ObjectName  
{  
  
};
```

You can give your class' name anything you would like. In C++, it is usually standard for each word in a class to start with an uppercase letter, like above. In Java, you usually start with the first word in all-lowercase. There is no concrete rule, and for the most part it is a stylistic choice.

With a class object, it will have public and private attributes and functionality. Public attributes and functions will be visible to the user (for example, the ignition in a car, where you plug in a key and turn).

Private attributes would be functionality or attributes that the user doesn't need to see (such as, the engine and inner-workings that work to get the car started).

So perhaps, we would model an Engine first:

```
class Engine  
{  
    public:  
    void StartEngine();  
    private:  
    // Engine inner-workings go here!  
};
```

In the Engine, we can see the “Start Engine” function. This isn't exposed to the driver, but the Car itself knows how to run the “Start Engine” function.

In C++, the line that says

```
void StartEngine();
```

Is a function. The first word, “void”, specifies what kind of information this function will return. It is a *datatype* (as mentioned above), and “void” means that this function won't return any data.

If we were writing a “Sum two numbers” function, it might return an *int*, which would be the sum.

For the Car itself, we might model it as:

```
12  class Car
13  {
14      public:
15          void TurnKey();
16          void PressGasPedal();
17          void PressBrake();
18          void ChangeGear( int gear );
19
20      private:
21          Engine carEngine;
22          float gallonsOfGas;
23          int currentGear;
24  };
```

Notice that the car **contains** an Engine object on line 21. We can write class objects, and include them inside of other class objects.

Engine, the **Current Gear**, and the **Gallons of Gas** are all private; the user isn't going to directly interact with these. They will turn the Engine on through the Ignition, they will change gears through the Shifter, and they will add more gas through the Fill Spout.

Currently, we do not have any functions under the **private** area, and we do not have any variable attributes under the **public** area, but you can have functions and attributes under either section.

What we have publicly exposed to the driver, are functions like “Turn the key in the ignition”, “Press the gas pedal”, “Press the brake pedal”, and “Change gear to (gear variable)”.

The only one different here is the **ChangeGear** function. Inside of the parenthesis, it contains **int gear**. This means that we are going to give this function data- an *integer* that symbolizes which gear we're shifting to...:

- 0 = Park
- 1 = Reverse
- 2 = Neutral
- 3 = Drive

etc. etc.

Now, the Car object also has logic to it – Pressing the gas won't do anything if the car isn't on, or if it's in park.

The Engine might not turn on if the car is in a gear other than Park, or if the Brake pedal isn't pressed down.

If the Car is in Reverse and the Gas pedal is pushed, the car will go backwards. If the Car is in Drive and the Gas pedal is pushed, the car will go forwards.

This different behavior based on attributes of the car (Current Gear, Whether the engine is on...) are *different states* that the car is sitting in. The car behaves differently based on these attributes, but how it behaves will be the same every time, given the same attribute values.

If you can't predict what your program is going to do, there is something wrong with your program! This is called “Undefined Behavior”. We'll bring up a few simple things that can cause unpredictable behavior as we write additional programs.

Let's back up from the car example, and write some code with a simpler object.

Let's create a Cat object. The Cat has an “age” attribute, and a “speak” function. If it helps, think of Class Objects as Nouns, and think of Functionality as Verbs.

```
class Cat
{
    public:
    void Speak()
    {
        cout << "Meow!" << endl;
    }

    int age;
};
```

Right now, we're going to make the Cat's “age” attribute public, even though you don't directly change a cat's age. This would normally be private, with a function like “Increment Age” once a year.

When the “Speak” function is called, it will output “Meow!” to the console.

Remember: When we create a Class, the closing curly brace } needs to have a semi-colon afterwards!

Not quite standard...

- Normally, we would store classes in their own Source files. For now, we can throw this Cat object just above **main** and below **using namespace std;**
- Also, in C++ it is not standard to define the function *inside of the class* like it is in Java and C#. Usually, you will have a **.h** (header) file to prototype the class and objects, and a **.cpp** (C++ source file) to write the function body (what is between the { and } curly-braces). Don't worry, we'll go over this later!
- We're just keeping it simple for now! This will work fine, it would just get a bit messy if this program were a lot bigger!

```
15  int main()
16  {
17      // Create our cat, give it an age.
18      Cat tesla;
19      tesla.age = 2;
20
21      // Output to the screen what the cat's age is, and its meow.
22      cout << "Tesla's age is: " << tesla.age << endl;
23      cout << "Tesla says: ";
24      tesla.Speak();
25
26      // Program ran successful; end.
27      return 0;
28  }
```

Here is the actual program.

Now, we can't just use `Cat` straight out of the box. The `Cat` class defines what kind of Attributes and Functionality a cat has.

On line 18 in **main**, we have to create a *new cat*. It is using the `Cat` class as a template, and creating our own version is called *creating an instance of Cat*.

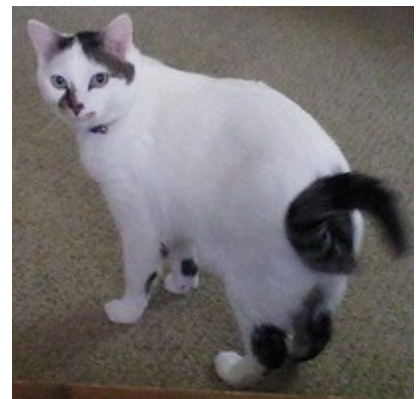
With one `Cat` class, we can create as many *cat instances* as we want!

On line 19, we setup our cat's age. If we don't set this at the beginning, it won't have a value. C++ will seriously just assign the “age” attribute to some random data in memory. This is usually totally nonsensical, and the actual term for this is *garbage*.

Not initializing your variables and getting this garbage is an easy way to have UNDEFINED BEHAVIOR in your program!

Once we get the cat setup (I named my Tesla), we can then output the cat's age through the **cout** stream.

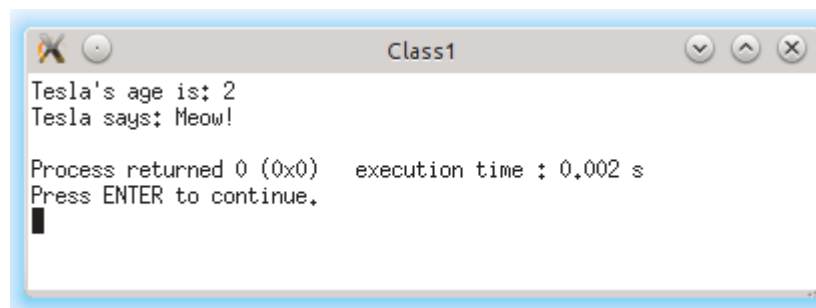
If we call `tesla.Speak()`, we will call the `Speak` function for the `Cat`, which will output “Meow!”



Tesla!

Here is the full program:

```
1  #include <iostream>
2  using namespace std;
3
4  class Cat
5  {
6  public:
7      void Speak()
8      {
9          cout << "Meow!" << endl;
10     }
11
12     int age;
13 };
14
15 int main()
16 {
17     // Create our cat, give it an age.
18     Cat tesla;
19     tesla.age = 2;
20
21     // Output to the screen what the cat's age is, and its meow.
22     cout << "Tesla's age is: " << tesla.age << endl;
23     cout << "Tesla says: ";
24     tesla.Speak();
25
26     // Program ran successful; end.
27     return 0;
28 }
```



This is just a simple usage of C++ Classes. You usually don't abstract cats into software, unless it is for a game or something.

That's all we're going to cover this time. Try to copy the programs and see if you can get them to run. After that, try modifying different things and experimenting!