On to C++

Let’s look at “Hello, World!”

(HelloWorld.java)

```
class HelloWorld
{
    public static void main(String args[]) {
        System.out.println("Hello, World");
    }
}
```

Let’s look at “Hello, World!”

(main.cpp)

```
// uses no includes like the java example


int main(int argc, char* argv[]) {
    printf("Hello, World!\n");
    return 0;
}
```

Let’s look at “Hello, World!”

(main.cpp)

```
// using old-fashioned “c-style” output

int main(int argc, char* argv[]) {
   printf("Hello, World!\n");
   return 0;
}
```

Let’s look at “Hello, World!”

(main.cpp)

```
#include <iostream>
// C++ style output
int main() {
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

Let’s look at “Hello, World!”

But, wait! What was this?

HellWorld.java:

```
    public static void main(String args[]) {}
```

main.cpp:

```
    int main(int argc, char* argv[]) {}
```

char* argv[] ???

We’ll unpack this another day.

Some Familiar Things

Syntax - very similar

Comments - identical

“main()” - note that main() is NOT part of a class!

Some New Things

“#include” -

analogous to Java’s “import”

“::” -

“Scope resolution operator”

“<<” -

“Insertion operator”

C++ vs Java

(Executable vs Byte-Code)

C++ is compiled

- The compiler converts your source to **machine code** which runs directly on your processor.
- It is platform-dependent.

Java is compiled and interpreted

- source code is converted into **bytecode** and then run/interpreted by the Java Virtual Machine (JVM).
- It is platform-independent.
- The JVM is platform-dependent.

C++ Source Code to Executable Process

Three Stages:

1) Pre-processor
2) Compilation
3) Linking

Pre-processing

This step takes all of the #includes, #defines and other compiler directives and combines them into “pure” C++.

This is important to understand because it will both help you to write code and to exploit language features such as:

- #include - insert the text from this file
- #define - replace this symbol with this text
- template - write code using this form

These are expanded in the preprocessor stage.

Compilation

Takes the output from step one (pre-processor) and produces an “object” file*.

The object* file is an interim form of executable. It contains both executable binary and information that allows the linker to resolve symbols.

These files typically have the extension “.o”.

*object: this is not to be confused with the instantiation of a class.

Linking

This is where previously unresolved symbols are resolved.

What do we mean by “symbols are resolved”?

For every named thing (variable, class, function) that was NOT defined in the current file, the linker attempts to find where it is defined so that an address can be assigned in its place. This step wipes out the names altogether.

This is the step that generates the “unresolved symbol” errors.

IDE: CLion

We’ll be using CLion this semester.

This is what you will also be using.

Looks and feels like IntelliJ.

Want to use an alternative?

It may make it difficult to get help related to the use of a different compiler/IDE

Let’s look at “Hello, World!”

(main.cpp)

```
#include <iostream>


int main() {
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

Let’s look at “Hello, World!”

(main.cpp)

```
#include <iostream>
using namespace std;
int main() {
    cout << "Hello, World!" << endl;
    return 0;
}
```

Some practical differences

- Source code
  - File names vs class names
  - Header files
- Functions can exist outside of classes
- Error Messages
  - Compile
  - Link

Some practical differences: File names vs class names

There is no required relationship between the name of a file and the class it contains.

There is a convention which creates PRACTICAL relationship, however.

Some practical differences: Header Files

Source Code

- Source code files typically work in “matched pairs”. That is, for each source (.cpp) file there is typically a matching header (.h) file.
- The header is the source code that is the “common” part of a module.
- We use “#include” to use a header file.
- Again, there is no require relationship between the name of a source file and the matching header file but...
- the convention exists.
- The PURPOSE of a header is to share just the necessary parts of a module

Some practical differences: Header Files

Source Code

Source code files typically work in “matched pairs”. That is, for each source (.cpp) file there is typically a matching header (.h) file:

myModule.cpp

myModule.h

One of the first lines in myModule.cpp will typically include the matching header:

```
#include “myModule.h”
```

Header Files Contain:

- Directives:
  - #define
  - #include
- Declarations
  - Class
  - Namespace
  - Template
  - struct
- Macro definitions
- Template definitions

Header Files Should NOT Contain:

The following are not allowed, or are considered bad practice:

- built-in type definitions at namespace or global scope
- non-inline function definitions
- non-const variable definitions
- aggregate definitions
- unnamed namespaces
- “using” directives

These things have ramifications that become hidden when placed in the header file. They will frustrate you and anyone using your code.

Let’s Extend Hello, World!

Parameter passing is Pass-by-Value.

Functions can exist outside of classes!

Void addOne(int valInt);

Namespace

Let’s look at a use case:

```
namespace woodley {
    int myInt;
}
int myInt;
```

*this is to demonstrate the effect, not to recommend a usage

Namespace

Let’s look at a use case:

```
namespace woodley {
    int myInt;
}
int myInt;
```

I can distinguish the first myInt from the second by: `woodley::myInt`

`woodley::myInt` is the “full name” of the first variable above

Namespace

Let’s look at a use case:

```
namespace woodley {
    int myInt;
}
int myInt;
```

I can do this but it has other implications:

```
using namespace woodley;
```

Namespace

We WILL expect “proper” use of namespaces. There will be at least one assignment where this is part of the rubric.

Let’s Extend Hello, World!

Let’s move addOne() into a namespace

Let’s Extend Hello, World!

Let’s move addOne() out of main.cpp

Error Messages

Compile Errors - won’t compile

```
std::cout
```

Link Errors - compiles but won’t link*

```
extern int thisInt;
thisInt = 0;
```

*you won’t see this if it doesn’t compile

Next Time:

Classes in C++
Namespace