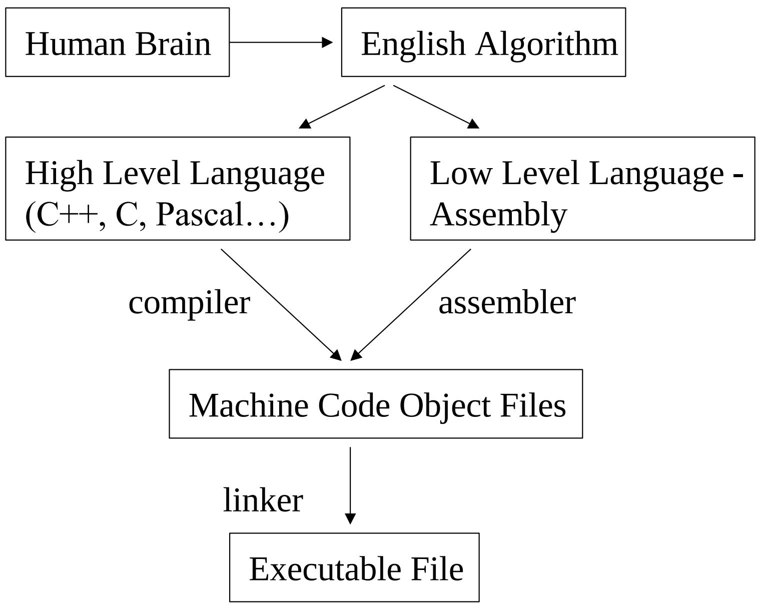
CSCE A211
Programming Intro

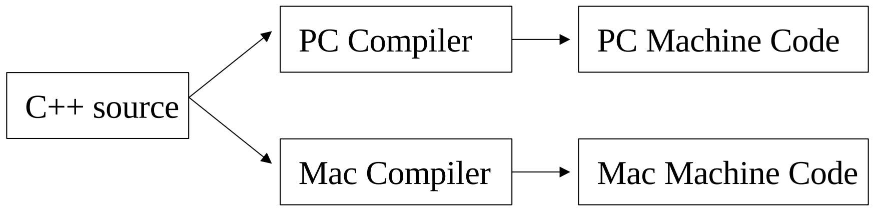
What is a Programming Language – Assemblers, Compilers, Interpreters

A **compiler** translates programs in high level languages into machine language that can be executed by the computer. C++, C, Pascal, Ada, etc. are all examples of high level languages. Here is a big picture of what is happening when you write and compile a C or C++ program.


Human Brain
English Algorithm
High Level Language (C++, C, Pascal…)
Low Level Language - Assembly
compiler
assembler
Machine Code Object Files
linker
Executable File

Note that the compiler takes the source program and typically produces an **object** program – the compiled or machine code version of the source program. If there are multiple source files that make up a final program, these source programs must then be **linked** to produce a final executable. The executable is generally unique to a particular CPU and operating system.

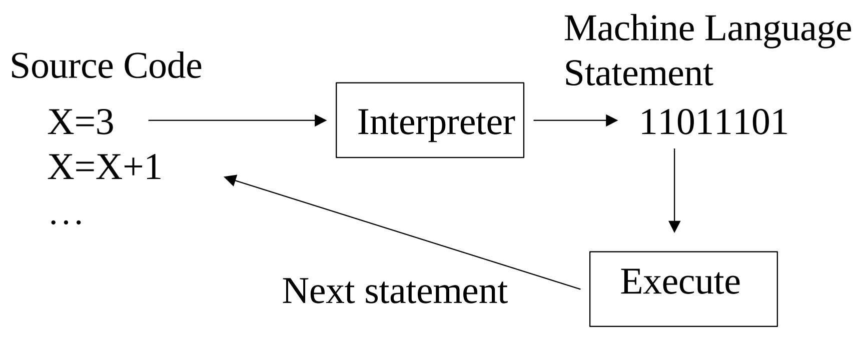
For example, a Macintosh cannot understand machine code intended for an Intel processor. However, if there is a standard version of C++ (which there is, the latest major revision was C++11), then one could write a program and have it compile on the two different architectures. This is the case for "standard" programs, but any programs that take part of a machine's unique architecture or OS features will typically not compile on another system.



Under this model, **compilation** and **execution** are two different processes. During compilation, the compiler program runs and translates source code into machine code and finally into an executable program. The compiler then exits. During execution, the compiled program is loaded from disk into primary memory and then executed.

C++ falls under the compilation/execution model.

However, note that some programming languages fall under the model of **interpretation**. In this mode, compilation and execution are combined into the same step, interpretation. The interpreter reads a single chunk of the source code (usually one statement), compiles the one statement, executes it, then goes back to the source code and fetches the next statement.



Examples of some interpreted programming languages include JavaScript, VB Script, some forms of BASIC (not Visual Basic), Lisp, and Prolog.

What are the pro's and con's of interpreted vs. compiled?

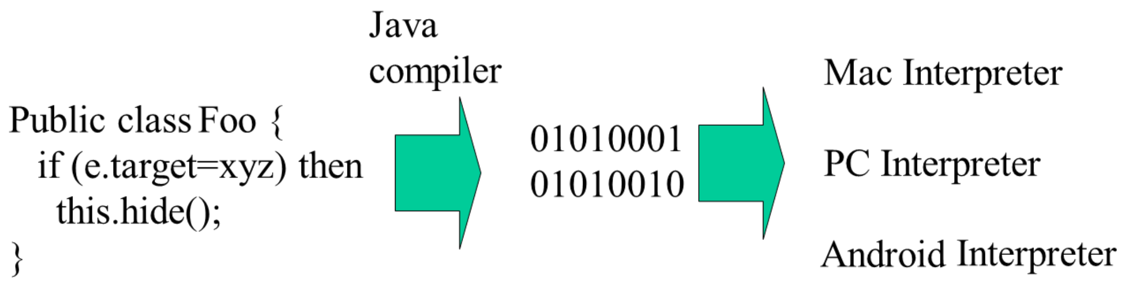
Compiled:

- Runs faster
- Typically has more capabilities
  - Optimize
  - More instructions available
- Best choice for complex, large programs that need to be fast

Interpreted:

- Slower, often easier to develop
- Allows runtime flexibility (e.g. detect fatal errors, portability)
- Some are designed for the web

In the midst of compiled vs. interpreted programming languages is Java. Java is unique in that it is both a compiled and an interpreted language (this is a bit of a simplification with "Just In Time" compilers, but we will ignore that distinction for now). A Java compiler translates source code into machine independent **byte code** that can be executed by the Java **virtual machine**. This machine doesn't actually exist – it is simply a specification of how a machine would operate if it did exist in terms of what machine code it understands. However, the byte code is fairly generic to most computers, making it fairly easy to translate this byte code to actual native machine code. This translation is done by an interpreter that must be written on different architectures that can understand the virtual machine. This interpreter is called the Java Virtual Machine (**JVM**) or Java Runtime Environment.



The great benefit of Java is that if someone (e.g. Oracle, the makers of Java) can write interpreters of java byte code for different platforms, then code can be compiled once and then run on any other type of machine. Unfortunately there is still a bit of variability among Java interpreters, so some programs will operate a bit differently on different platforms. However, the goal is to have a single uniform byte code that can run on any arbitrary type of machine architecture.

Another benefit is we can also control "runaway" code that does things like execute illegal instructions, and better manage memory.   We will discuss these later in the course.

Hello, World in C++

The first program that many people write is one that outputs a line of text. In keeping with this vein, we will start with a program that prints, "Hello, world".   First, here is the program in its entirety:

```
// A first program in C++.  The double slashes are comments for one line.
/*
    We could add more comments if we like using slash and an asterisk.
     The commends end with an asterisk and then a slash.
*/

#include <iostream>              // Not iostream.h like in C
using namespace std;             // include all members of the standard namespace
int main()
{
        cout << "Hello, world" << endl;
        return 0;                // return code
}
```

Before we try to compile and run this program, let's go through a description of what is in here. First, the // and /* … */ statements are for comments. Descriptive text about what the program does should go into the comment fields. We'll use these to describe the input, output, and dependency behavior of the program.

The line "#include <iostream>" is a **preprocessor directive**. This is a message to the compiler to perform some directive before compiling the program. In this case, we are "including" the contents of the iostream.h file (the .h is implicit and can be left off on most compilers, but not all). The iostream.h file includes input and output library routines – essentially you will be using software written by someone else. We will talk more about these later.

The next line, "using namespace std;" will identify the scope of names we will be using for the program. The default is the standard namespace. Most compilers will assume the standard namespace if this line is left out. Some programmers consider it bad form to use this exact line, since it includes everything from the standard namespace, and we might only want to use a little bit from the standard namespace.

The line 'cout << "Hello, world" << endl;' will output the line of text "Hello, world". If we did not use the line "using namespace std;" above, then we would have had to write this line as "std::cout << "Hello, world" << endl;". It becomes cumbersome to add the "std::" in front of all the statements, which is why in this class we will use the "using" statement. No matter how the function is invoked, "cout" is a function that belongs to the namespace "std". cout is the standard output stream, which is where we will see our output text.

The "<<" operator is the **stream insertion operator**. When the program runs, items to the right of the operator, i.e. the **operands**, are fed into the output stream to the left.

Finally, the last line "return 0;" returns the number 0 to the caller. We could return any number we want here, but perhaps we have defined the program in such a way that 0 indicates that the program has terminated successfully.

Compilation

The process of entering and compiling your program is different depending upon what development environment you are using. We'll have a few assignments in both Unix and Microsoft Visual C++, and then you will be able to choose which environment you prefer for the rest of the class. Visual Studio is more graphical than the pure Unix environment and in general offers more features. A convenient benefit of the Unix environment is that your code and programs are all in one place, so if you work from home and also from school you won't have to shuttle your files back and forth on floppies to work on your program from different locations. Also, if you don't have a Windows machine at home, you would only be able to use Visual Studio on the campus labs. **Finally, be warned that there are some subtle differences between Unix's g++ and Visual C++, so a program that works in one environment might not work in the other!**

Compiling under Unix

See the lab assignment for details about logging in and editing files.

To compile the program, invoke the C++ compiler using "g++":

➢ g++ hello.cpp

If there were any errors, the compiler will complain and tell you it encountered problems. If all goes well, you'll be returned to the prompt and you should now have a file named "a.out" in your current directory. To run it, simply enter a.out and it should run. You can rename the file to something else if you wish (the mv command) or if you wanted to name the output file something other than a.out, you could compile with the flag: -o outname. For example:

➢ g++ -o hello hello.cpp

Would produce a file named "hello" instead of "a.out".

Note, to compile a Java program under Unix we would use:

➢ javac Hello.java

and this produces a file called Hello.class. This contains the Java Byte Code for the compiled code that would run on the hypothetical Java virtual machine.

To run this code use:

➢ java Hello

Make sure your case matches or there will be error messages.