An Overview of C++

This is the Course Reader for CS 150, C++ Programming I, at Orange Coast College. CS 150 is a **second-semester** course for majors who have already taken a course covering variables, I/O, calculations, loops, decisions, functions and arrays.

The reader is short and concise, in line with Leonard Elmore's advice: *leave out the parts readers skip*. It focuses on the parts you need to get your work done.

To get more details, consult one of these highly recommended texts.

- **The C++ Primer, 5th Edition. Lippman**, Lajoie and Moo
- **The C++ Programming Language, 4th Edition**, Bjarne Stroustrup

These are **professional** books to keep on your shelf after the course is over. Throughout the text you'll also find links pointing to additional reading, which is **optional**.

The C++ Programming Language

C++ is a **compiled**, high-level programming language. Compilers produce **native machine code** programs, which run directly on your hardware, without an interpreter (like Python) or intermediate software, such as the Java Virtual Machine. Even though C++ programs are often more efficient than their Java or Python counterparts, certain kinds of errors cannot easily be detected, so you will find that you must be more careful and detail-oriented when writing programs in C++.

C++ is also a **standardized language**, specified by the ISO, or International Standards Organization. Unlike **proprietary languages**, controlled by a single company, **anyone** may implement the C++ language without fear of lawsuits. There are several versions of C++ you should know about.

- **Pre-standard C++**: 1980-98. Often incompatible versions, vendor specific.
- **C++ 98**: the first standard version; it was updated in 2003 (**C++03**).
- **C++ 11**: the second **major** standard; it was updated in **C++14** and **C++ 17**.
- C++20, the latest major standard, is now **complete**. Most recent compilers offer a sampling of some of its new features.

In this class we will be using **C++ 17**. Compilers that implement C++17 include **Visual Studio 2019**, GNU **g++ 7** (or later), and **clang 6**+.

Hello World

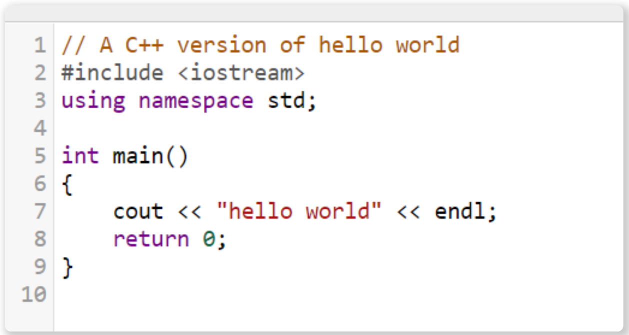
On the first page of Dennis Ritchie's seminal programming book, **The C Programming Language,** he writes:

> *The only way to learn a new programming language is by writing programs in it. The first program to write is the same for all languages:*
>
> *Print the words: **hello, world***
>
> *This is the big hurdle; to leap over it you have to be able to create the program text somewhere, compile it successfully, load it, run it, and find out where the output went. With these mechanical details mastered, everything else is comparatively easy.*

I'm sure his last sentence was **somewhat** tongue-in-cheek, but let's follow his example and look at a C++ version of "Hello World" by clicking the little "running-man" to your right to open the program in an online IDE. Leave it open for the next few pages.

Source Code

When you click the link you'll find the human-readable **source code** for this program, in the file which we would name hello.cpp.



```cpp
 1 // A C++ version of hello world
 2 #include <iostream>
 3 using namespace std;
 4 
 5 int main()
 6 {
 7     cout << "hello world" << endl;
 8     return 0;
 9 }
10 
```

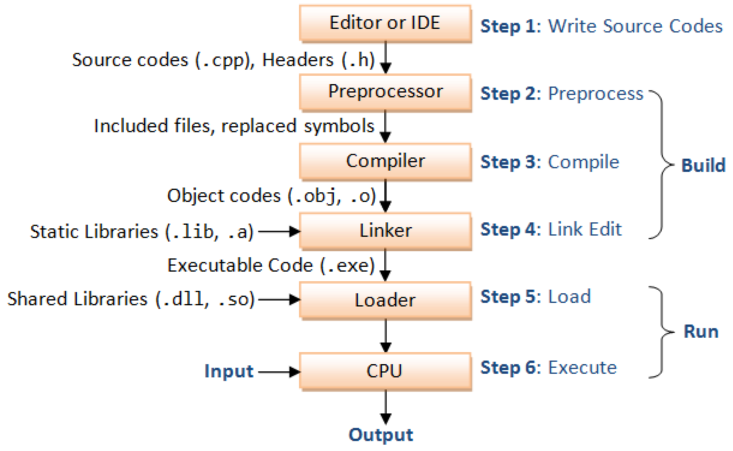
Let's review each line:

1. A **comment**, designed for the reader, but ignored by the compiler.
2. A **preprocessor directive** which makes the C++ library facilities available.
3. A **namespace directive** makes standard names available without qualification.
4. A **blank line**, used to make your code more readable.
5. The `main` function or **entry point** to your program. Each program has one `main`.
6. The **opening** brace. How you say **begin** the actions in C++.
7. Prints to the console, using the **standard output** object named `cout`.
8. Ends the main function, **returning** the value `0` back to the operating system.
9. The **closing** brace marks the **end** of `main`, matching the opening brace.

Program Mechanics

To create your own C++ programs, you follow a **mechanical process**, a well-defined set of steps called the **edit-build-run** cycle. In short you:

1. **Create** your source code using a **text editor**.
2. **Build** your executable using a **compiler toolchain**
3. **Run** your program using your operating system facilities



The Build Process

Once you have written your source code, the **build process** turns that source code into an executable program. The build process involves several tools:

- **Preprocessor**—performs text substitution on your source code.
- **Compiler**—generates **assembly code** from the preprocessed source code.
- **Assembler**—converts assembly code into **object** or **native machine code**.
- **Linker**—combines machine-code modules into an **executable** that runs.
- **Make**—provides instructions for **building** each program.
- **Loader**—reads the executable from disk into memory and starts it running.
- **Debugger**—runs your program inside a controlled environment.

A **combination** a C++ compiler, linker, assembler and libraries, is known as a toolchain. The toolchain we'll use for this course is the Linux GCC 9 toolchain. In an online IDE, like the one in these lessons, you just click the **Run** button to perform all the steps and run the executable. In lecture, you'll look at each of these steps in more detail.

Early Computing

In the 1930s, **Alan Turing** (dramatized by Benedict Cumberbach in *The Imitation Game*), imagined a **universal computing machine** which could store both its data, **and** its **programs** in memory. 1940s-era computers, such as the ENIAC, **had no programs** as such. Instead, they were **hard-wired** to perform specific calculations. Programming them entailed rewiring their circuit cabinets, a process that could take several days.

Turing's ideas were realized in 1946, when mathematician and physicist **John Von Neumann** described the first real **stored program** computer system, the EDVAC, whose **machine language** instructions were stored in memory as a **binary numeric code**.

```
1000 1011 0100 1110 0000 0110
```

To run a program on the **EVAC**, each instruction was **fetched** from memory by the CPU Control Unit (CU) and then stored in **registers** on the CPU. The instruction was then **decoded** and **executed** by the CPU's Arithmetic/Logic unit (ALU). Finally, the results were written back into memory where they could be examined. This sequence-read, decode, execute and store-is known as the instruction cycle; it's how every computer works.

Machine Code and Assembly Language

Here's an example. The **Intel** CPU instruction which **copies a value from memory** into the `CX` register is `8B4E06`. We humans see this as a **hexadecimal (base 16) number**. The computer, on the other hand, is a **digital electronic device**, which **doesn't know about numbers at all**; it is built using integrated circuits (or **transistorized switches**), each of which is either on or off. We humans **interpret** the "on" state as a `1` and the off state as a `0`.

So, how does the computer "know" what to do with the instruction `8B4E06`? It doesn't! In **binary** this instruction is `100010110100111000000110`, but inside the hardware, it is simply a block of switches. **Electricity flows** through each `1` to another part of the device. The flow of electricity is blocked by a `0`.

As a physical analogy, imagine the player-piano roll, where a hole in the paper causes a note to be played, allowing a hammer to strike a particular string.

Machine code is also called **native code**, since the computer can use it without any translation. Machine language programs are difficult to understand and, inherently **non-portable**, since they are designed for a single type of CPU.

Yet, high-performance programs are still written in machine language (or its symbolic form, assembly language). You can examine the native code for the APPLE II Disk Operating System, written by Steve Wozniak, at the Computer History Museum.

High-Level Languages

Machine and assembly language are **low-level languages**, tied to a specific CPU's instruction set. In the mid-1950s, **John Backus** lead a team at IBM which developed the first high-level programming **language.** FORTRAN (or the **FOR**mula **TRAN**slator), allowed scientists and engineers to write their own programs.

COBOL (the **CO**mmon **B**usiness **O**riented **L**anguage), allowed accountants and bankers to write programs using a vocabulary with which they were comfortable. In 1958, **John McCarthy** at MIT built a third high-level language named LISP (the **LIS**t **P**rocessing Language) to help him with his research into artificial intelligence.

These three high-level languages allowed non-computer specialists to write their own programs, but they were not **general-purpose** languages; scientists could not use COBOL to write code for NASA, and accountants could not use FORTRAN.

In 1960, the designers of FORTRAN, COBOL and LISP gathered together in Paris to remedy that, producing the **Algo**rithmic**L**anguage (ALGOL). Modern languages derive much of their syntax and many core concepts from ALGOL.

This photo, taken at the 1974 ACM Conference on the History of Programming Languages, shows six of the original participants who attended the 1960 Algol Conference in Paris. Top row: John McCarthy (LISP), Fritz Bauer, Joe Wegstein (COBOL). Bottom row: John Backus (FORTRAN), Peter Naur, Alan Perlis.

Many new languages followed rapidly in the next seven decades. Here are two:

- BASIC, the Beginner's All-purpose Symbolic Instruction Code, was developed at Dartmouth University in 1964. The professors **Kemeny** and **Kurtz** wanted to teach programming to university students using the new, interactive, time-share computers. BASIC was later the first language available on micro-computers, implemented by Bill Gates for the Altair. Even later, in the 1990s, Microsoft introduced **Visual Basic**, a graphical version, popular in the business world.
- In 1972, the Swiss computer scientist Nicholas Wirth felt that BASIC was teaching students bad programming habits, so he created the popular teaching language Pascal, (named after the philosopher, Blaise Pascal). Strongly influenced by Algol, Pascal enforced structured programming techniques. It was the first programming language taught in most university computing programs until about 2000. Wirth later designed Modula, Oberon, and Ada, a language still in wide use in avionics and in defense.

The C Programming Language

In 1970, two researchers at Bell Labs in New Jersey (**Ken Thompson** and **Dennis Ritchie**) developed a **portable operating system** for the new wave of **mini-computers** just coming on the market. They named their operating system Unix (or `UNIX` if you like).

Unix was originally written in assembly language. Later was converted, (or ported), to a high-level language named **B**. Then, to simplify the development of Unix, Dennis Ritchie modified the **B** language and created the programming language named C.

Both Unix and C have had enormous impacts on the field of computing. Many of today's most important language are based on C, including C++, Java and C#, which still use much of the original syntax designed by Ritchie.

Unlike the other technologies we have discussed, UNIX is not acronymic (like FORTRAN). The original 1974 CACM paper— The UNIX Time-Sharing System —used all caps because, in Ritchie's words, " we had a new typesetter and troff had just been invented and we were intoxicated by being able to produce small caps. " Later Ritchie tried to change the spelling to 'Unix' in a few of his papers. He failed and eventually gave up. See the Jargon file.

Object-Oriented Programming

`C` and `Pascal` are **imperative** languages, where the emphasis is on the **actions** that the computer should do. In the **procedural paradigm**, (or programming style), programs consist of a hierarchy of subprograms (**procedures** or **functions**) which process external data.

Object-oriented programming, or `OOP`, is a different style of programming. In the `OOP` paradigm, programs are communities of somewhat self-contained components (called **objects**) in which **data** and **actions** are combined.

The first object-oriented language was `SIMULA`, a **simulation** language designed by the Scandinavian computer scientists **Ole-Johan Dahl** and **Kristen Nygaard** in 1967. Much of the terminology we use today to describe object-oriented languages comes from their original 1967 report.

The C++ Language

A Danish Ph.D. student at Cambridge University, **Bjarne Stroustrup**, was heavily influenced by `SIMULA`. In 1980, working as a researcher at AT&T Bell Labs, he began adding object-oriented features to `C`. This first version of what would later become C++, was named C with Classes.

Unlike **idealistic** languages that enforce "one true path" to programming Utopia, C++ is **pragmatic**. C++ is designed as a multi-paradigm **language**.

In C++ the object-oriented and the procedural programming styles are **complementary**, since you may find a use each approach. So, consider each paradigm or style a **tool** that you can pull out of your toolbox when needed, to apply the conceptual model that is most appropriate for the task at hand.