Class 01

Introduction & Getting Started

Outline

- Introduction & Syllabus
- What is Scientific Computing
- What is C++
- Hello World!
    - Example & Program Outline
    - Compiling Code
    - Running a Program
    - Data Types

What is this Course

- This course is mostly a C++ course. This means learning how to *write code* using the C++ programming language.

- The code that we will be writing will be relatively simple in nature but will allow us to solve various problems in mathematics and the sciences.

- A key component of this course is to also introduce the student to industry standard tools, processes, and techniques.

- Classes will typically consist of a lecture covering C++ topics followed by introducing a topic in scientific computing and hands-on coding.

What is Scientific Computing

- Scientific computing is a broad subject. It encompasses all problem spaces that utilize computers to solve problems in mathematics and the sciences.

- At a high level this includes modeling & simulation, data analysis, machine learning, and many other subjects.
    - e.g. We can use computers to simulate the flight path of a ballistic missile and assign a probability to its risk to national assets. This allows us to predict outcomes and prepare accordingly in the real world.

- We will be writing software to instruct the computer to perform numerical tasks to solve various problems across a few areas within STEM. This typically involves writing formulas and algorithms in code.

What is C++?

- C++ is a compiled, mid-level language.
  - Compiled – code needs to be processed by a compiler before being executed on the computer.
    - C++ code is compiled **directly** to native code. i.e. code is compiled directly to a form that the computer can readily read.
    - Note that this *is not* the case for languages like Java, JavaScript, Python, C#, and many others!
    - Others like C++ are C, Rust, Go, Carbon, Ada, ...
  - Mid-level – the language provides constructs and mechanisms to give developers access to low level functions and memory facilities, while also being abstracted enough to be considered high-level.

Why Do/Don't We Use C++?

- Projects use C++ for many reasons:
  - Compiled C++ code runs *fast*.
  - Compiled C++ code runs ***very fast***.
  - C++ gives us access to low level functions and facilities:
    - Memory control
    - Pipes, sockets, and other file descriptors
    - Threads
  - Legacy. C++ has been around for a while, and so some projects use C++ so that they may leverage older legacy code.
- Some projects avoid C++ for several reasons:
  - **Writing C++ is not easy. Writing C++ is not easy. Writing C++ is not easy.**
  - It is not portable; a C++ program on one system may not run on another system!

Writing C++ is not Easy

- C++ is a *footgun* language.
- It is extremely easy to write broken, bad, and otherwise poor C++ code, and the language does very little to mitigate this.
  - Your code may compile, run, and produce results... and it is still probably bad.
- This has given rise to many other languages to potentially replace C++ in many ecosystems.
  - Rust
  - Go
  - Carbon

Common Use Cases for C++

- Operating Systems
- Simulations
- Audio Editing
- Computer Graphics
  - graphic design
  - computer animation & special effects
  - video editing
- Videogames
- *Other Programming Languages*

MySQL
UNREAL ENGINE
PhysX by NVIDIA
MAYA
Ps
Pr

My C++ Experiences

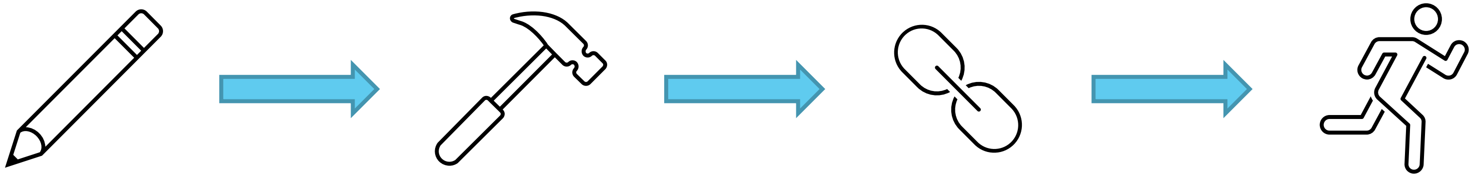
- Lockheed Martin
  - Medium fidelity missile simulation
  - Simulation model integration
- Susquehanna International Group
  - Middleware, systems monitoring/diagnostics
- Two Six Technologies
  - Network/packet analysis, binary payload analysis
- Lockheed Martin/Actalent
  - High performance analysis tools
- Improbable
  - Highly parallel patterns-of-life simulations backend
- EpiSci
  - Network-collaborative autonomous systems, SoC + simulation

Unlocking Trusted Autonomy for a Secure Future

Basic Software Development Workflow

- At its most basic form the development workflow takes us from writing code to executing a program.

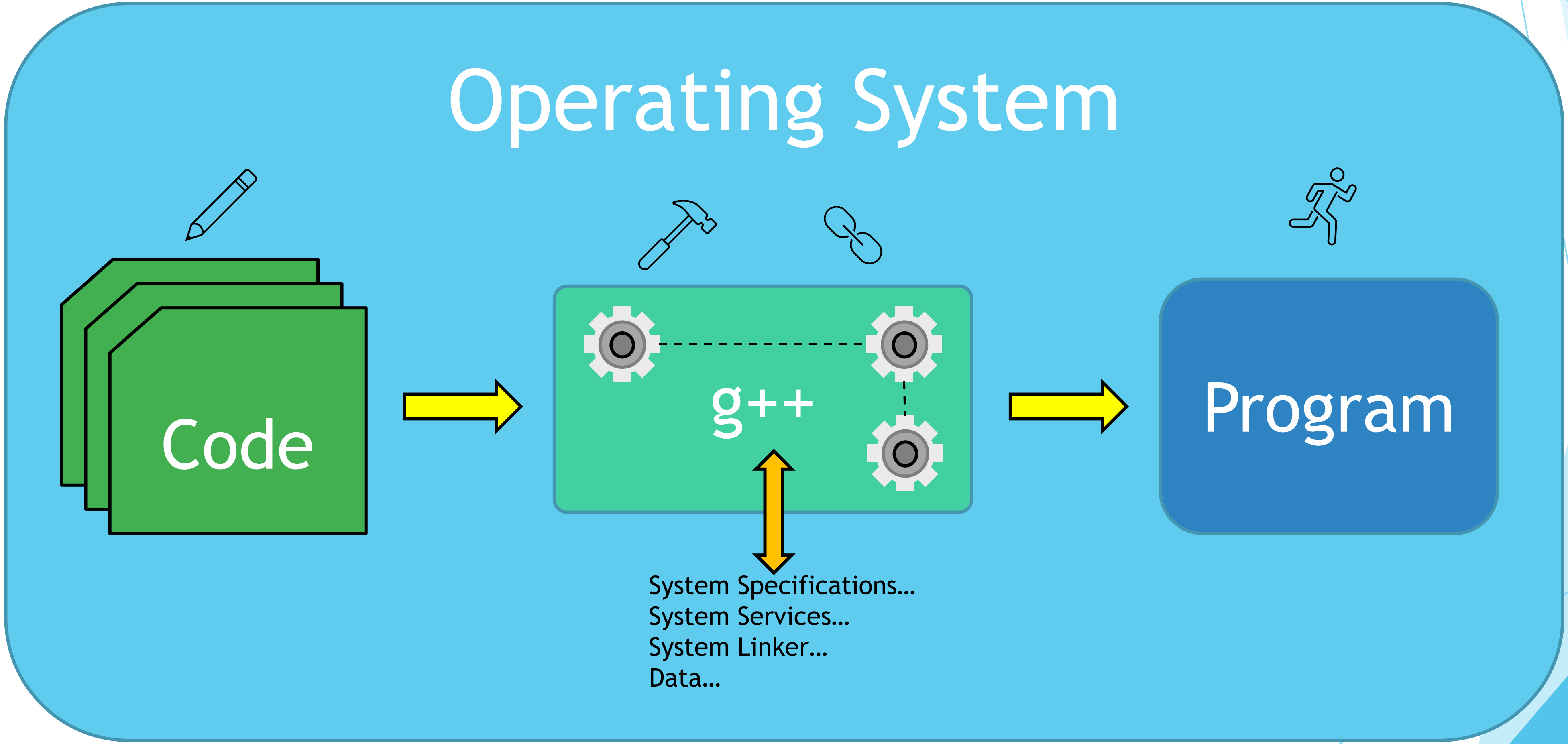
- A language like C++ has more than those 2 steps:
  - Write – writing human readable C++ instructions for the computer
  - Compile – converting the human readable C++ instructions into binary instructions
  - Link – combining all binary instructions into a cohesive program
  - Execute – running the program



Compilers & Linkers

- Compilers read code and convert it into machine code.
  - Machine code is "language" that the operating system/CPU understands, and this machine code is different for operating systems, CPU architectures, and more.
  - Your operating system dictates the specific machine language used.
- Linkers read machine code along with whatever else is needed by the target computer’s operating system (Windows, MacOS, Linux, etc.) and combine it all into a binary (an executable or a library).
- The compiler we will be using is GNU's **g++**, and the linker we will be using is the GNU system linker **ld**.

Compilers & Linkers



System Specifications…
System Services…
System Linker…
Data…

Writing C++

- Writing C++ simply means writing a text file.

- C++ text files will not have the usual ".txt" file extension. They instead have the extensions ".cpp", ".h", and ".hpp".
  - Other extensions used by some include ".cxx", ".hh", ".hxx"
  - Each extension indicates the purpose of the file.

- It is as simple as creating a file with the appropriate extensions, writing in it, and finally saving it.

Compiling & Linking C++

- Once we have code written we need to compile and link it into a program.
- We pass the file containing the code to the compiler (g++).
- g++ will compile the code, and then forward the compiled code to the linker to create your program.
- We will typically access and run g++ through the *terminal*.
  - A terminal is a tool for running commands on and interacting with a system.

Compiling & Linking C++ Example

- Let's say we have a file named "main.cpp" that contains our code, and we want to build a program named "command".

- To compile and link (or simply – to build) our code we do the following in the terminal:

```
g++ main.cpp –o command
```

- This tells g++ to build the file "main.cpp" into the program "command".

Running C++ Programs

- While this is not specific to C++, once we have our program built, we can run it via the *terminal*.

```
./command
```

- This tells the terminal to run the program named "command".

- The "./" in the beginning tells the terminal to look in the current folder for the program.

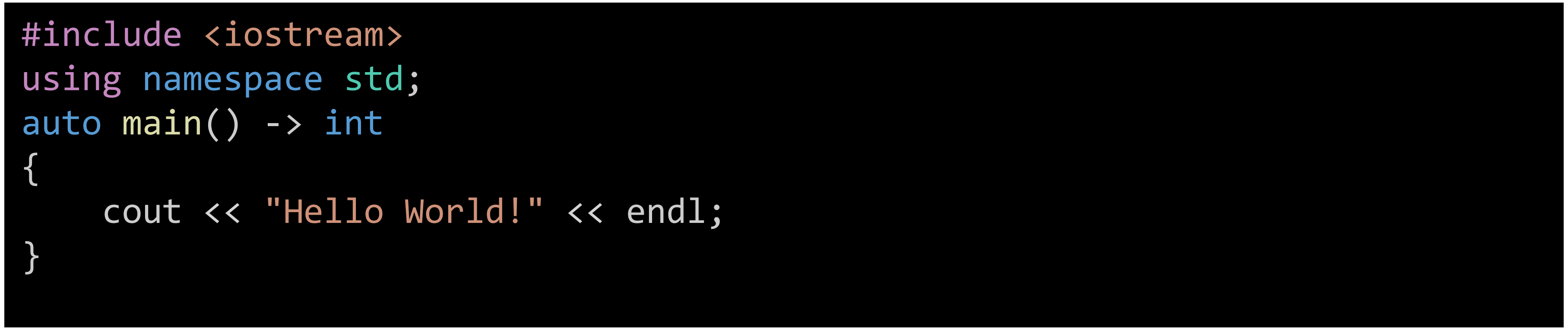
Hello World

- We are going to consider one of the most widely written programs in the history of programming and see how it is written in C++.

- We will glaze over some details at first, as we want to get the fundamental basics out of the way.

- Everything starts from here!

Hello World



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

Hello World

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

- This line instructs the computer to print out the phrase “Hello World!”.
- *cout* is the object that does the printing
- *endl* is the object that adds a new line after printing out the phrase
- << is an **operator** that combines the objects together to print them.
  - We end all statements with a semicolon. This is how C++ knows when a statement ends

Hello World

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

- The first line gives us access to *cout* and *endl*.
  - C++ only gives us basic functionality out of the box and so we need to instruct it to give us more!
- The *main* and *curly braces ({ and })* define where our program starts. The braces denote a **block**.
  - Every C++ program has a **main block**.
- Note that none of these lines are considered statements, and thus do not end in a semicolon!

Let's Try Things Out

- Now we will check out the technology and tools used for this course and run through the Hello World example ourselves.

Primitive Types

- Within a program we will be dealing with many different pieces of data, all of which will need to be expressed and stored within memory in different ways.

- Some data types will behave one way while other data types behave other ways.

- The primitive types in C++ are the basic building blocks of everything else in the language and can be used to express anything.

- We will list all primitive types but will cover only a few in detail.

Primitive Types (and their typical sizes)

bool	logical true/false	1 byte
char	standard characters	1 byte
wchar_t	wide characters	4 bytes
char8_t	UTF-8 character	1 byte
char16_t	UTF-16 character	2 bytes
char32_t	UTF-32 character	4 bytes
short	small integer	2 bytes
int	integer	4 bytes
long	large integer	8 bytes
long long	large integer	8 bytes
float	single precision floating point	4 bytes
double	double precision floating point	8 bytes
long double	extended precision floating point	16 bytes

Primitive Types (and their typical sizes)

bool	- logical true/false	1 byte
char	- standard characters	1 byte
wchar_t	- wide characters	4 bytes
char8_t	- UTF-8 character	1 byte
char16_t	- UTF-16 character	2 bytes
char32_t	- UTF-32 character	4 bytes
short	- small integer	2 bytes
int	- integer	4 bytes
long	- large integer	8 bytes
long long	- large integer	8 bytes
float	- single precision floating point	4 bytes
double	- double precision floating point	8 bytes
long double	- extended precision floating point	16 bytes

Data & Variables

- Naturally, as our programs become more complex, we will need ways to manage our data. We can use *variables* to hold onto data.

```
int x = 8;        // basic, bad
auto x = 8;       // semi-modern, good/ok
int x {8};        // semi-modern, good/ok
auto x = int{8};  // modern, best
```

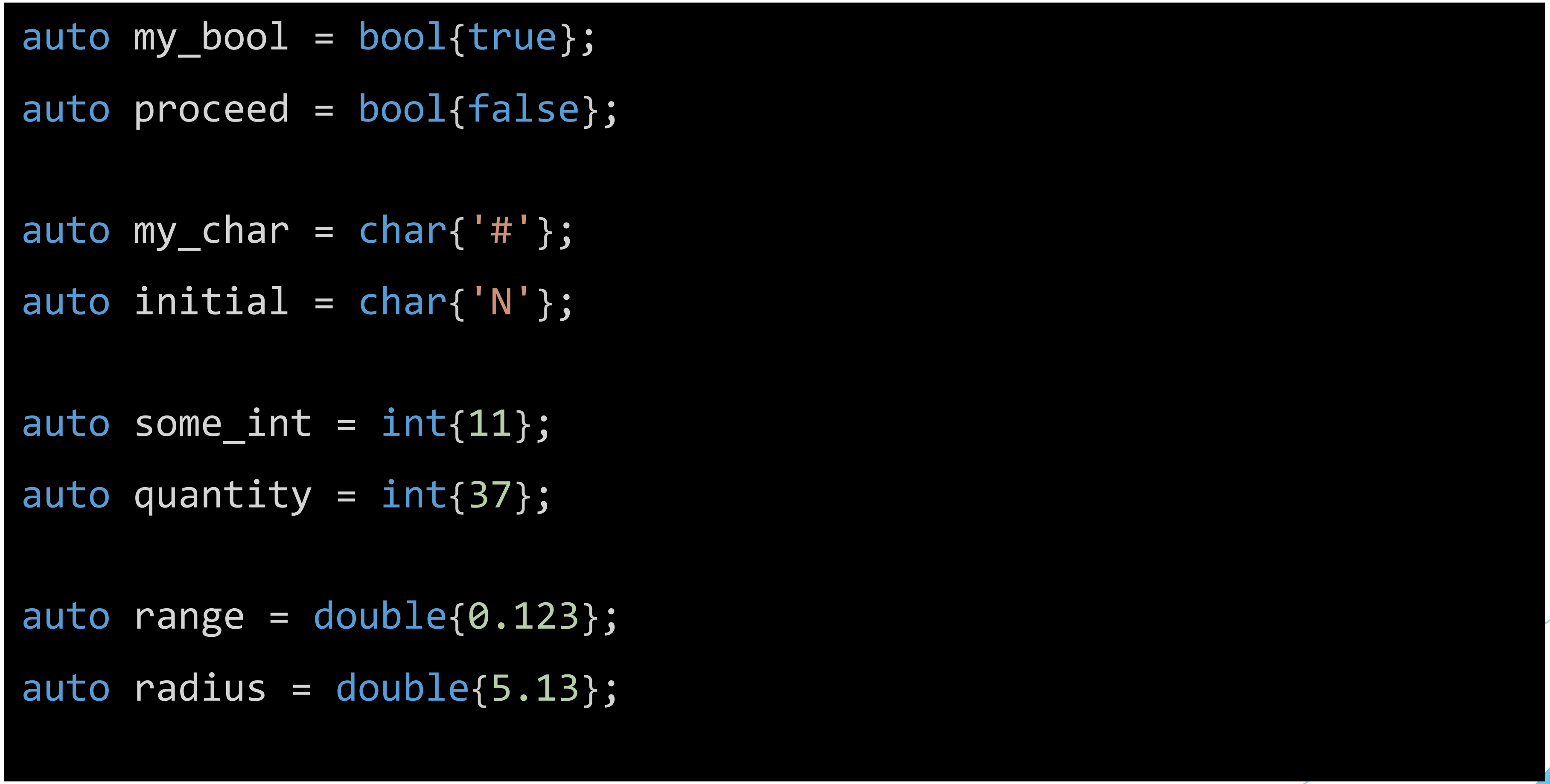
- Here, our data is named *x* and it is of the type *int*. It has a value of 8. We can now use *x* like any other piece of data in our statements.

```
cout << "x is equal to " << x << endl;

auto y = int{x + 1};

cout << "y is equal to " << y << endl;
```

Data & Variables Examples



```
auto my_bool = bool{true};
auto proceed = bool{false};

auto my_char = char{'#'};
auto initial = char{'N'};

auto some_int = int{11};
auto quantity = int{37};

auto range = double{0.123};
auto radius = double{5.13};
```