# ENGR 101 – Chapter 11

Introduction to C++

# Motivation

- Different languages have different strengths!

- MATLAB works well for:
  - Analyzing and presenting data
  - Engineering problems and scientific computing
  - Rapid prototyping and interactive coding

- C++ works well for:
  - Writing larger, more complex programs
  - Problems that require complex design and control flow
  - Learning more about the way computers and programs really work!

# VIDEO

DIFFERENCES BETWEEN MATLAB AND C++

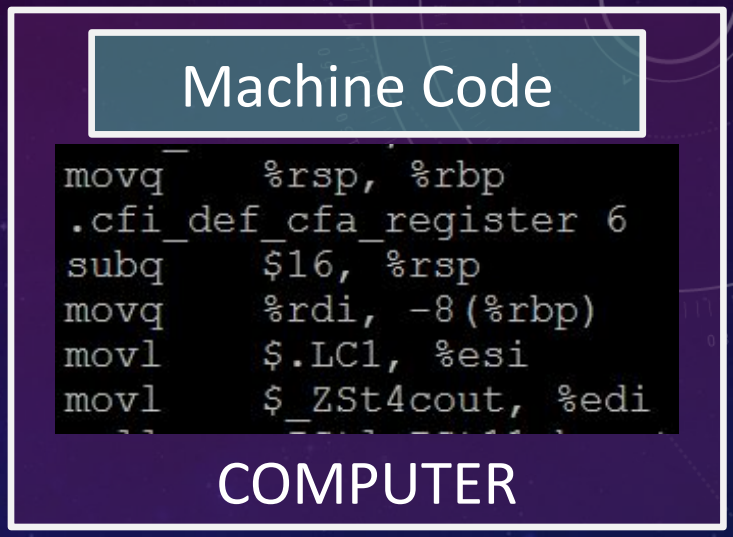
# Interpreted Languages

- MATLAB is an **interpreted** language.
  - Your code doesn't actually run on your computer!
  - Instead, MATLAB (the program) runs on your computer and processes your code **line-by-line**, telling the computer what to do.

- Think of this like having a conversation through an interpreter (i.e. the MATLAB program).

# Machine Code

- Computers can only directly run programs written in "machine code".
- Different kinds of computers also have different machine languages.
- Machine code is not human-friendly.



# Compiling Source Code

- Nobody wants to write machine code!
- Instead, write **source code** in a language like C++ and use a program called a **compiler** to translate it to machine code.

Source Code (C++)

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

int main() {
  cout << "Hello World!" << endl;
}
```

Compile

Machine Code

```
movq    %rsp, %rbp
.cfi_def_cfa_register 6
subq    $16, %rsp
movq    %rdi, -8(%rbp)
movl    $.LC1, %esi
movl    $_ZSt4cout, %edi
```

- There are many C++ compilers. For ENGR 101, we use g++.

# Compiling vs. Running (NOT THE SAME!)

- Step 1: Compile
  - Translate a source code file into an executable file (i.e. machine code).

**hello.cpp (C++)**

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

int main() {
  cout << "Hello World!" << endl;
}
```

Compile

**hello.exe**

```
movq    %rsp, %rbp
.cfi_def_cfa_register 6
subq    $16, %rsp
movq    %rdi, -8(%rbp)
movl    $.LC1, %esi
movl    $_ZSt4cout, %edi
```

- Step 2: Run the program
  - Take an executable program file and actually run it on the computer.

Note: We call the executable program hello.exe, but a common convention is to leave off the .exe on executable files.

# END VIDEO

DIFFERENCES BETWEEN MATLAB AND C++

# VIDEO

DEMO: COMPILING WITH G++

# Hello C++

- Let's get started with a classic C++ program…

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

int main() {
  cout << "Hello World!" << endl;
}
```

This code prints a message "Hello World!".

We'll look at the details later.

# Demo: Compiling with g++

Let's write up our "Hello World!" program, compile it with g++, and run it!

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

int main() {
  cout << "Hello World!" << endl;
}
```

# END VIDEO

DEMO: COMPILING WITH G++

# Recap: Compiling with g++

- Step 1: Compile

```
g++ program.cpp helper.cpp -o program
```

The names of files containing your source code.

In ENGR 101, this may often be just one file.

-o followed by the name of the executable program you want to create. This is the output of the compiler, and what you use to run your program.

- Step 2: Run the program

```
./program
```

Must match executable name from step 1!

# Review: WARNING

Never do this:

```
g++ program.cpp -o program.cpp
```

You accidentally specify the source file as the destination for the machine code. The compiler is happy to overwrite it.[1]

Please back-up your code regularly :)

1 Some modern versions of g++ will actually catch this and prevent the overwriting. But don't count on it!!

# VIDEO

MORE RANDOM C++ SHIT

# More C++

- Let's add some more code and look at some details…

```
int main() {
  cout << "Hello World!" << endl; // print a greeting

  // Declare some variables
  int x = 10;
  int y = x * 7;
  int z = x + y;

  // Print out the result
  cout << "The result is" << z << "!" << endl;

}
```

# Statements

- Just as in MATLAB, we use sequences of **statements** to give instructions for what we want our program to do.
- In C++, all statements MUST end with a semicolon **;**.

```
int main() {
  cout << "Hello World!" << endl; // print a greeting

  // Declare some variables
  int x = 10 + 5;
  int y = x * 7;
  int z = x + y;

  // Print out the result
  cout << "The result is" << z << "!" << endl;
}
```

# Variable Declarations

- In C++, you are required to **declare** variables before you can use them.

```
int x = 10;
int y = x * 7;
int z = x + y;
```

- A declaration lets the compiler know about the variable:
  - Its **name**.
  - Its **type** (i.e. what kind of data it holds).
  - An optional **initializer** expression.

```
type name = initializer;
```

# Basic Types

- C++ supports many different types. Here are a few of the basics:

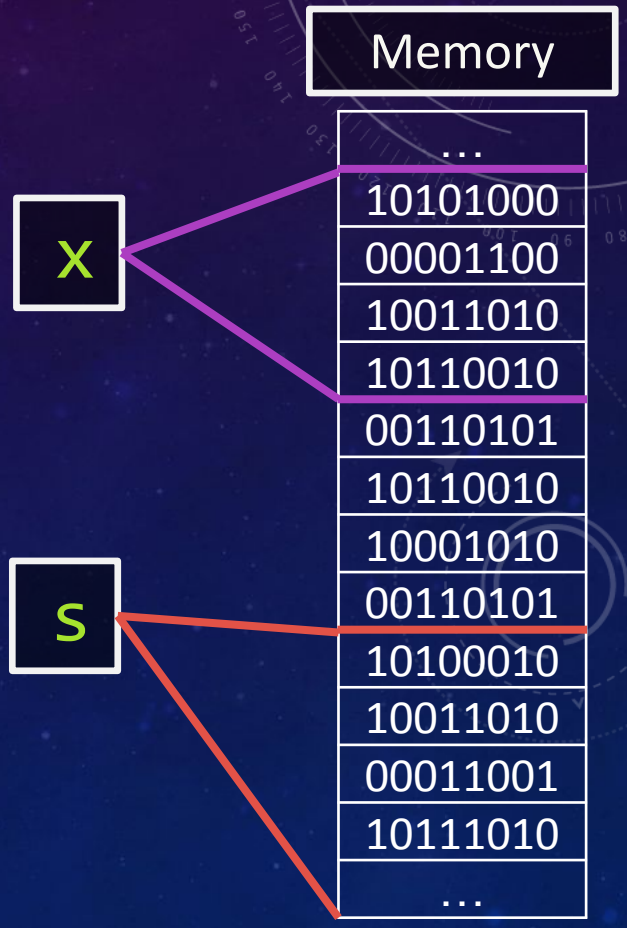
| Type | Description | Example |
|---|---|---|
| `int` | A signed integer. (Can be negative) | `int x = 3;` |
| `double` | A floating point number. (i.e. has a fractional part) | `double y = 2.5;` |
| `bool` | A Boolean (i.e. logical) value. 1 – true, 0 – false. | `bool z = true;` |
| `char` | A single character. | `char c = 'w';` |
| `string` | A sequence of characters. | `string word = "hello";` |

- A big difference from MATLAB… No more built-in matrices!

# Variables and Memory

```
int x = 10;
string s = "hello";
```

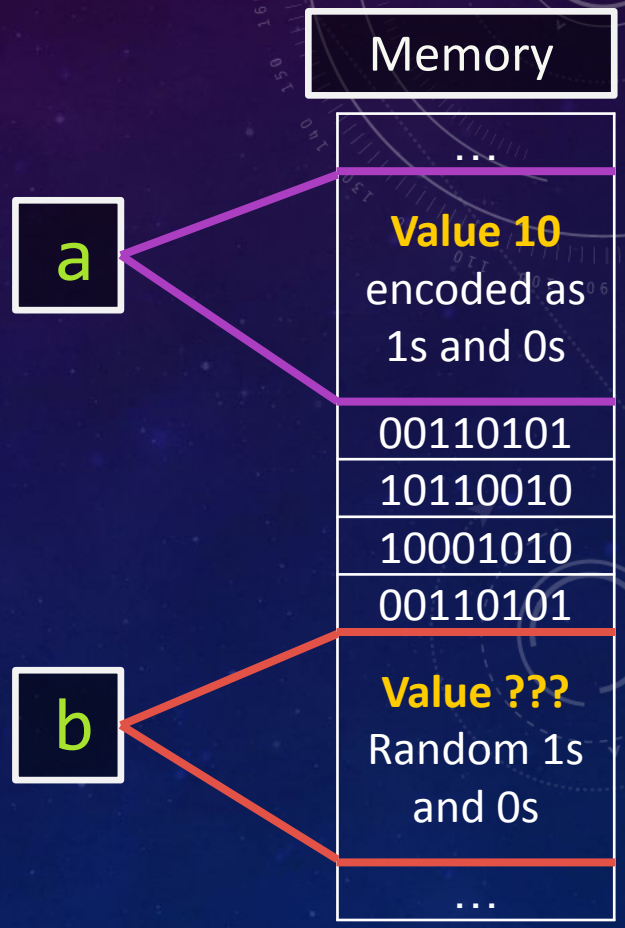
- Declaring a variable basically requests memory space to store some data.
  - Data is stored in bytes, which consist of eight binary digits (i.e. ones and zeros).
- The **type** determines how much space.
  - e.g. a `string` takes more space than an `int`.
- The **name** allows you to refer to that memory throughout the rest of your code.



# Uninitialized Variables

```
int a = 10;
int b;
```

- If you don't provide an initializer, the value of the variable is **undefined** until you assign something into it.
- The value is just based on whatever **memory junk** was there previously.



# Expressions in C++

- Just as in MATLAB, we use **expressions** in C++ to perform computations on variables and other data.

- Expressions may consist of:
  - **Literals** (e.g. 3, 7.5)
  - **Variables** (e.g. x, y, z)
  - **Function Calls** (e.g. `sin(3)`, `sqrt(x)`).

# Expression Types

- Question: What is the type of this expression?

```
x + y
```

- We need to see the declarations of x and y!

```
int x = 10;
int y = 7;
int z = x + y;
```

Type = int

```
string x = "app";
string y = "le";
string z = x + y;
```

Type = string

# Comments in C++

- Two different styles:
  - Single-line – anything after a // on a line becomes a comment.
  - Block – anything between a /* and a */ becomes a comment.

```
int main() {
  cout << "Hello World!" << endl; // single line

  // another single line comment
  int x = 10 + 5;

  /* a comment on multiple lines
     about all the program things */

  int x = 10 /* don't */ + /* do */ 5 /* this :( */;
}
```

# END VIDEO

MORE RANDOM C++ SHIT

# VIDEO

DEMO COMPILE ERRORS

# Printing Output

- To print output in C++, we need to send it to the "standard output stream".
- `cout` is a variable that represents this stream.
  - That's pronounced "C out" (two words).
- The `<<` operator sends output, and can be "chained" to send many different pieces on one line.

```
cout << "Hello World!" << endl; // print a greeting

// Print out the result
cout << "The result is" << z << "!" << endl;
```

`endl` represents a new line.

# User Input

- When the user types input at the terminal, it comes in via the "standard input stream", represented by `cin`.
- The `>>` operator reads input from a stream.
  - It can be chained, just like the << operator.
  - Input is interpreted according to the type of the target variable.

```
int x;
string s1;
string s2;

cin >> x; // read an int (e.g. 3, 72, -4)

cin >> s1 >> s2; // read two strings (e.g. "hi", "cat")
```

# END VIDEO

MORE RANDOM C++ SHIT

# VIDEO

DEMO COMPILE ERRORS

# Demo: Compile Errors in g++

- Let's upgrade the "hello" program from earlier…
- We'll also take a look at how g++ reports errors in our code.

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

int main() {
  cout << "What is your name?" << endl;

  string name;
  cin >> name;

  cout << "Hello " << name << "!" << endl;
}
```

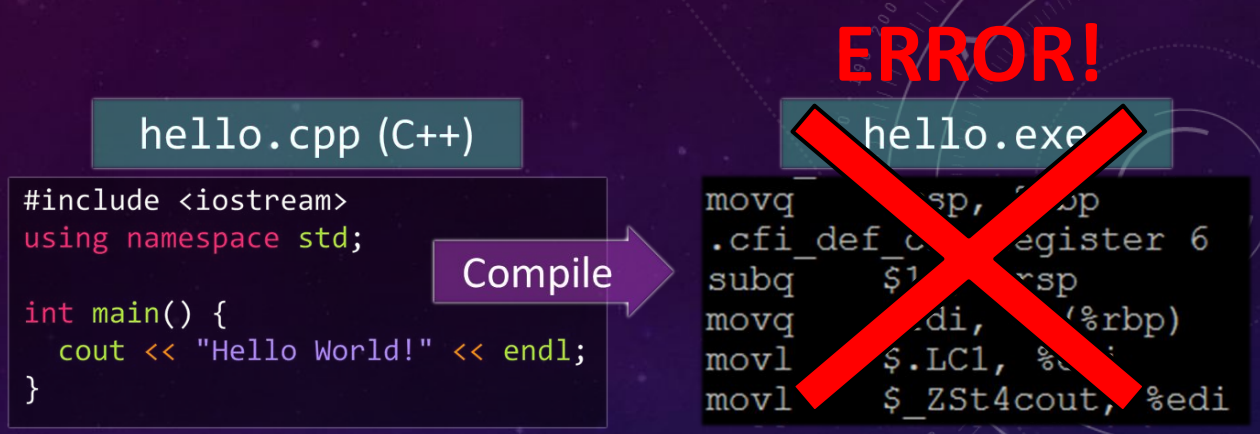
# END VIDEO

DEMO COMPILE ERRORS

# VIDEO

COMPILE ERRORS

# Compile Errors



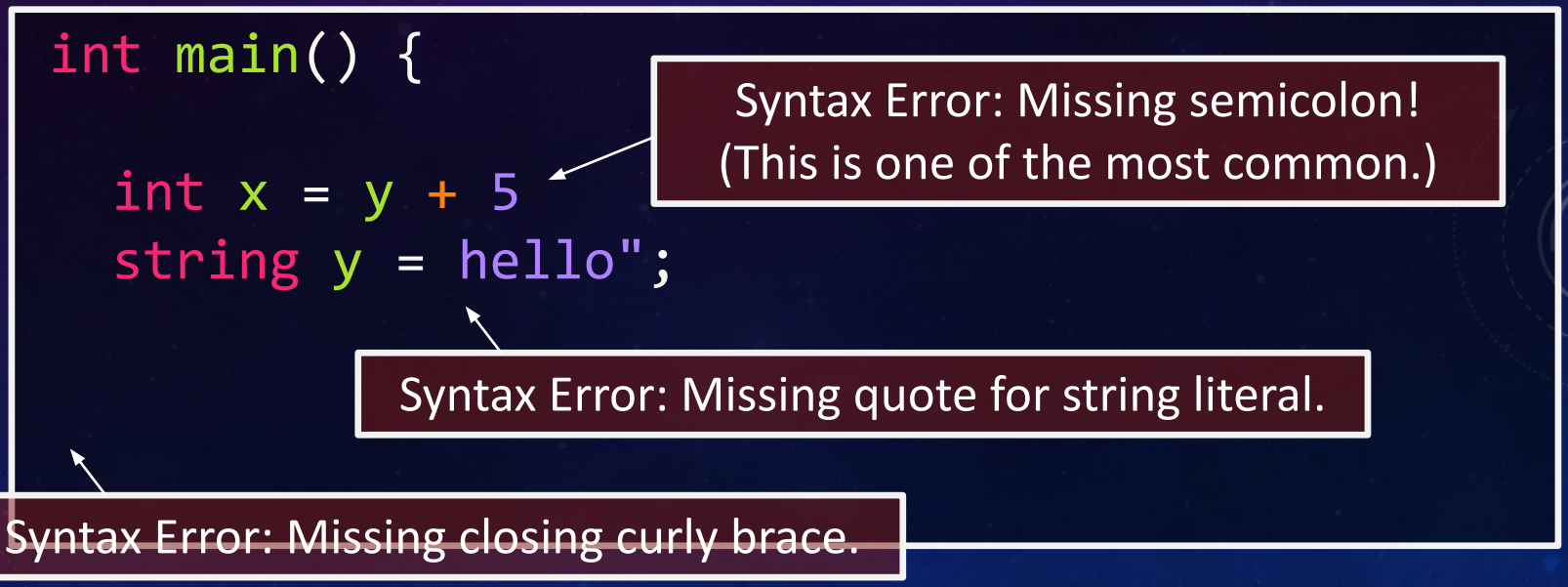

- Step 1: Compile
  - Translate a source code file into an executable file (i.e. machine code).
  - **Also, analyze the code for errors.**
- If the compiler finds an error, no executable file is produced.
- Good news! The compiler can catch errors for you!
- Bad news! You can't run your program until you fix them.[1]

1 This is actually good news, even though it can be frustrating!

# Compile Errors: Syntax

- All programming languages have **syntax** rules.
  - These are the proper grammar and "punctuation" you must use.

- Here are some examples:



# Compile Errors: Semantics

- A statement might not make sense, even if it's syntax is fine.
  - An example from linguistics: "Colorless green ideas sleep furiously."

- Here are some examples in C++:

```
int main() {

  int x = y + 5;

  int y = "hello" / 5;

}
```

Error: Undeclared variable y!

Error: Can't divide a `string` by an `int`. This is a **semantic error**, because the operation we requested doesn't make sense.

# Compile Errors: Semantics

- A statement might not make sense, even if it's syntax is fine.
  - An example from linguistics: "Colorless green ideas sleep furiously."

- Here are some MORE examples in C++:

```
using namespace std;

int main() {
  int x = 5;
  x = 6;
  int x = 7;
  cout << "Hello!";
}
```

Error: Duplicate definition of x!

Error: cout doesn't exist. (We forgot to use #include <iostream> above!)

# Compile Errors: Type Errors

- There are two main kinds of type errors:
  - Invalid operations
  - Invalid conversions

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

int main() {
  int i = 5;
  double d = 3.5;
  string s = 7;
  i = s; // Invalid conversion (string to int)
  i + s; // Invalid operation (string + int)
  d = i; // Conversion allowed (int to double)
}
```

When are conversions allowed? We'll get into rules for that later…

# Static vs. Dynamic Typing

- C++ is **statically typed**
  - A variable's type is known at compile-time, based on its declaration.
  - A variable's type is fixed, and never changes!

- MATLAB is **dynamically typed**
  - A variable's type can change at runtime.
  - It's type just depends on they type of value it is currently holding.

# The Type System

- Because C++ is statically typed, the compiler is able to check for ALL type errors before your program even runs!
  - All variables and expressions have fixed types.
  - The compiler's **type system** can check to make sure we never mix types in an inappropriate way.

```
string x = "app";
int y = 5;
```

C++

```
x + y
```

`string` + `int` can never work!

- This isn't the case in MATLAB.
  - It is dynamically typed and can't check up front.
  - A variable can end up holding any type of value.
  - We only discover type errors at runtime.

```
x = 'app';
y = 5;
```

MATLAB

```
x + y
```

```
x = 2;
```

This could work…

# END VIDEO

COMPILE ERRORS

# Compile-time vs. Runtime

- **Compile-time**
  Things known when compiling your code (i.e. BEFORE running it!)
  - Are there syntax errors in the code?
  - Are all variables declared before they are used?
  - What is the type of this variable? Are all operations on it valid?

- **Runtime**
  Things that don't happen until your code actually runs!
  - What value did the user enter?
  - Did your program accidentally divide by zero and crash?

# Runtime Errors

- The compiler can't always predict things that could go wrong...

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

int main() {
  cout << "Please enter two numbers to divide...";

  int x;
  int y;
  cin >> x >> y;

  cout << "The result is: " << ( x / y ) << endl;
}
```

Hypothetical: If the user enters 0 for y, the divide by zero causes a **runtime error** and the program crashes!

# VIDEO

TEMPERATURE CONVERTER (WITH BUGS)

5 min

# Exercise: Temperature Converter

- Write a program that accepts a temperature in degrees Celsius and converts to degrees Fahrenheit. (F = 9/5 * C + 32)
- Your program should:
  - Prompt the user for a temperature in Celsius
  - Read the temperature from the standard input stream (i.e. `cin`)
  - Compute the temperature in Fahrenheit, and print it.
- If you've got C++ set up on your computer, try to compile/run the code!

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

int main() {
  // Your code here
}
```

# Demo: Temperature Converter Attempt

- Let's attempt a solution for the temperature converter.
- Full disclosure: the code on this slide has **bugs**!

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

int main() {
  cout << "Enter a temperature in Celsius: ";

  int c;
  cin >> c;

  int f = 9 / 5 * c + 32;
  cout << f << " degrees Fahrenheit.";
}
```

# END VIDEO

TEMPERATURE CONVERTER (WITH BUGS)

# Logic Errors

- Logic errors occur when the program compiles and runs without crashing… but it's still doesn't work right!
  - It's harder to catch these, because they're less obvious.
  - This is why **testing** your programs thorougly is important!

- Example -- Check your temperature converter with a couple of known test cases:
  - Temperature that water freezes (0 deg C, 32 deg F)
  - Temperature that water boils (100 deg C, 212 deg F)

- Did it work correctly?

# Corrected Solution: Temperature Converter

- Write a program that accepts a temperature in degrees Celsius and converts to degrees Fahrenheit. (F = 9/5 * C + 32)

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

int main() {
  cout << "Enter a temperature in Celsius: ";

  double c;
  cin >> c;

  double f = 9.0 / 5. * c + 32;
  cout << f << " degrees Fahrenheit.";
}
```

Add either `.` or `.0` to the literals to ensure we get floating point division instead of integer division.

The variable `f` needs to be a `double`, or else the value gets truncated.