LECTURE III

C++ Programming

SECTION I

Programs and Programming Languages

What is a Program?

- A computer **program** is a **sequence of instructions** for a computer to execute
- **Software** is a **set of programs** and **data** that are used to perform a specific task on the computer
  - Ex) MS Word, Discord, Minecraft
  - Contrasts from **hardware**, which is the collection of **physical components** that make up the system

Programming Languages

- A **programming language** is a **notation** for the instructions we give to the computer
- Just like a human language, a programming language is **defined by grammar**
  - There is **syntax** - a **structure** or way of organizing symbols in the language
    - Ex) In English, sentences are *structured* as follows:
      - Subject + Verb + Predicate
  - There are also **semantics** - the **meaning** of a set of symbols or their arrangement

Programming Languages (Cont'd)

- Popular programming languages include **C++**, **Java**, **Python**, **Javascript**
  - Each of these languages have a **unique grammar** (with some overlap)
  - Each have **strengths and weaknesses**
    - Ex) Javascript is widely used in websites but isn't used as much for desktop applications
    - Ex) Python is great for quick scripting but more challenging when writing software that interfaces with hardware or manages memory






Machine Language

- Computers execute instructions in a **machine language** or **machine code**
  - Machine code is written in **binary** 1s and 0s, which is converted into HIGH and LOW voltages on the hardware level
- Human-readable programming languages must be **translated** into machine language that the computer can execute

Source Code

```
a = "hello";
b = a + "!";
```

Translation

Machine Code

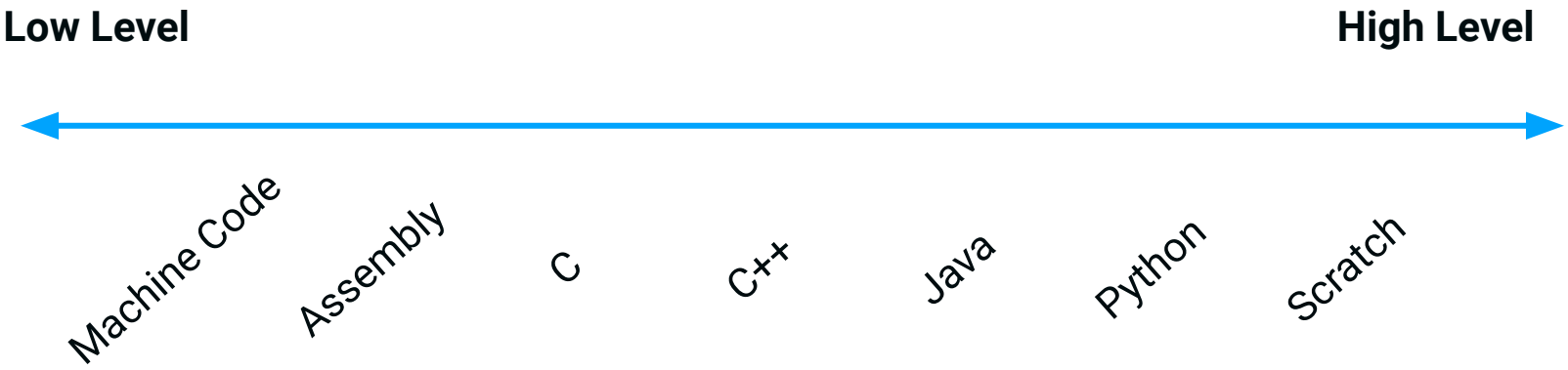
```
1011 1010
1010 1011
0110 1011
```

High vs Low Level Languages

- **High-level** programming languages provide strong abstractions from the computer hardware
  - The language may automate memory management
  - Ex) **C**, **C++, Python**, **Java**
- **Low-level** programming languages provide little to no abstractions and tend to be structurally similar to machine language instructions
  - Ex) **Assembly** languages, **Machine languages**

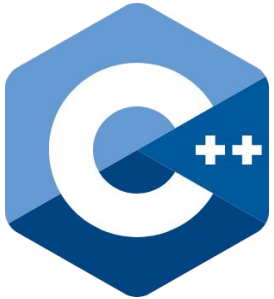
High vs Low Level Languages (Cont'd)

There is a **spectrum** of low to high level programming languages:



C++ Programming Language

- C++ is a **general-purpose** programming language
  - Used to write operating systems, video games, **embedded software**, etc.
- It was designed as a **superset of the C programming language**
  - Much of what you write in C can run in a C++ program
- Why would we focus on C++?
  - We will eventually use **Arduino**, which is based on C++

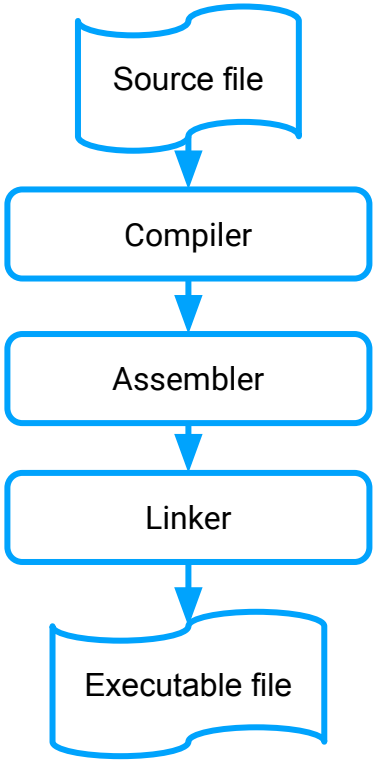


SECTION II

Compilation Process

Compilers, Assemblers, and Linkers

- We must **translate** C++ source code into machine code using a program called a **compiler**
- Technically, translation is handled by multiple programs:
  - A **compiler** converts C++ source code into assembly language
  - An **assembler** converts the assembly code into machine code
  - A **linker** takes the machine code files and “links” them together into one **executable file**



Target Architectures

- Machine code is **architecture-dependent**
  - Ex) Intel or AMD-powered computers use **x86-64 machine code**, many phones are ARM-based which use **ARM64 machine code**
    - Intel, AMD, and ARM are all processors which execute program instructions within the computer (to be discussed in future lectures…)
- The compilation process must have a **target architecture**
  - Ex) For C++ programs to run on Intel machines, they must be compiled to an x86-64 executable

SECTION III

Variables, Statements, and Operators

Statements

- Programs are composed of **statements**
- A **statement** is an instruction that causes the program to perform some action
  - **Syntax**: statements in C++ often **end with a semicolon (;)**

```
int n = 1;                    // declaration statement
n = n + 1;                    // expression statement
return 0;                     // return statement
```

Variables

- **Variables** are containers for data values
  - C++ variables have **types** - integers, characters, booleans, etc.
- Variables must be **declared**

```
int a;              // integer declaration
int b, c, d;        // three variable integer
declaration
```

- Variables are **assigned** values

```
a = 3;              // copy assignment to a
b = 102324325;
```

Variables (Cont'd)

- **Initialize** your variables when declaring them

```
int a = 50;              // integer variable
initialization
char c = 'w';
```

- **Syntax**: `type variableName = value;`
- **Naming Convention:** Variable names are written in **camelCase**
  - The first word is not capitalized and the first letter of all the following words is capitalized

Basic Data Types

Type	Definition	Example
`int`	Integer	`-2, 0, 1, 300`
`unsigned int`	Positive Integer	`0, 1, 5, 6`
`char`	Character	`'c', 'g', 'w'`
`float`	Floating Decimal	`1.2367`
`bool`	Boolean	`True, False`

Operators and Expressions

- **Operators** are used to perform operations on variables and values
  - Some are for arithmetic, assignment, comparison, etc.
- An **expression** is a combination of values, variables, and operators that **evaluate** to one single value

```
b + 5;                      // example expression
a + 4 / 5 - b * c;
```

Arithmetic Operators

Operator	Name	Definition	Example
+	Add	Adds two values together	`x + y`
–	Subtract	Subtracts one value from another	`x – y`
*	Multiply	Multiplies two values together	`x * y`
/	Divide	Divides one value from another	`x / y`
%	Modulus	Remainder of Division	`x % y`
++	Increment	Increases value of a variable by one	`x++`
––	Decrement	Decreases value of a variable by one	`y––`

Assignment Operators

Operator	Example	Equivalent Statement
=	`x = y`	
+=	`x += 5`	`x = x + 5`
-=	`x -= 5`	`x = x - 5`
*=	`x *= 5`	`x = x * 5`
/=	`x /= 5`	`x = x / 5`

Comparison Operators

Operator	Name	Example
==	Equal to	`x == y`
!=	Not equal to	`x != y`
>	Greater than	`x > y`
<	Less than	`x < y`
>=	Greater than or equal to	`x >= y`
<=	Less than or equal to	`x <= y`

Logical Operators

Operator	Name	Example
`&&`	Logical AND (Both True)	`x && y`
`||`	Logical OR (Either is True)	`x || y`

- Expressions with **logical** or **comparison** operators evaluate to **True** or **False**

SECTION IV

Control Flow

If Statement

```
if (condition) {
  // block of code to be executed if the condition is true
}
```

- The **if statement** is a type of conditional statement
- If the **condition** evaluates to true, the program executes the block below
- The **block** is the set of statements enclosed by { }

Else Statement

```
if (condition) {
  // block of code to be executed if the condition is true
}
else {
  // code that executes if the condition is false
}
```

- An **else statement** is an optional statement that executes only when the condition is **false**

If-Else Statement

```
int x = 2;
if (x > 1) {
  x = 1;
}
else {
  x = 6;
}
```

Ex) Suppose the program executes this code. What is the value of x?

- The condition (`x > 1`) evaluates to **`True`**, so the if-block executes
- The else statement is skipped
- **Solution**: `x = 1`

Else If Statement

```
if (condition) {
  // block of code to be executed if the condition is true
}
else if (condition2) {
  // code that executes if the previous conditions are
  // false and condition2 is true
}
```

- An **else if statement** is an optional statement that executes *only* if the **previous conditions are false** *and* the **new condition is true**

If-Else Statement

I/A

```
int x = 5;
if (x > 2) {
  x = 1;
}
else {
  x = 6;
}
```

What is the value of **x** after the code executes?

A. `x = 1`

B. `x = 2`

C. `x = 5`

D. `x = 6`

While Loop

```
while (condition) {
  // code to be executed while the condition is true
}
```

- The **while loop** executes the block of code repeatedly *while* the condition is **true**; the loop ends when the condition is **false**
- The **condition** is evaluated at the beginning of each loop

While Loop (Cont'd)

```
int x = 1;
while (x != 5) {
  x++;
}
```

Ex) Suppose the program executes this code. What is the value of x?

- The while loop executes as long as `x != 5`
- **Solution**: `x = 5`

For Loop

```
for (init-statement; condition; end-expression) {
  // code to be executed while the condition is true
}
```

- The **for loop** executes the block of code repeatedly until the condition is **false**
- The **init-statement** is executed once when the for loop starts
- The **condition** is evaluated at the beginning of each loop
- The **end expression** is executed at the end of each loop

For Loop (Cont'd)

```
int sum = 0;
for (int i = 0; i < 10; i++) {
  sum += i;
}
```

Ex) Suppose the program executes this code. What is the value of sum?

- The for loop executes 10 times; `i` increments once each loop
- **Solution**: `sum = 0 + 1 + 2 … + 8 + 9 = 45`

For Loop (Cont'd)

I/A

```
int x = 2;
for (int i = 1; i <= 3; i++)
{
  x = x-i;
}
```

What is the value of **x** after the code executes?

A. `x = 2`

B. `x = -7`

C. `x = -1`

D. `x = -4`

SECTION V

Arrays

Arrays

- An **array** is a **series of elements** of the **same type** that is referenced with a single identifier
- **Syntax**:

```
type arrayName[size];
```

- The array **size** or **number of elements** is fixed at declaration (for static allocation)
- By default, arrays are **uninitialized** (none of its elements are set) at declaration
    - Best practice is for us to initialize the elements at this time

Array Initialization

```
int foo[6];
```

Array declaration without initialization

```
int foo[6] = {9, 2, 5, 4, 8, 11};
```

Array declaration with initializer list

```
int foo[]{9, 2, 5, 4, 8, 11};
```

Universal initialization does not require an equal sign or an explicit array size; they are implicit

Array Access

- Elements of the array can be accessed using an **index** starting from `0`
  - Ex) An array with a size of `5` has indices ranging from `0` to `4`
- **Syntax**:

```
variableName[index]
```

```
foo[2] = 76;
```

Assigns the *third* element of **`foo`** to `76`

```
x = foo[2];
```

Assigns `x` to the *third* element of **`foo`**

Array Access

I/A

```
int x[3] = {5, 2, 4};
if (x[1] > 3) {
  x[0] = 3;
}
```

What is the value of **`x[0]`** after the code executes?

A. `x[0] = 5`

B. `x[0] = 2`

C. `x[0] = 3`

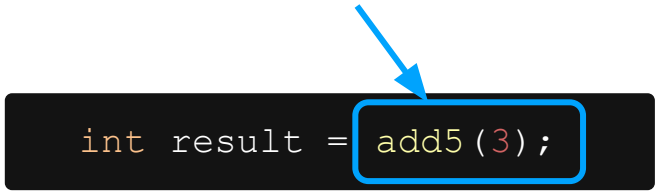
D. `x[0] = 4`

SECTION VI

Functions and Scope

Functions

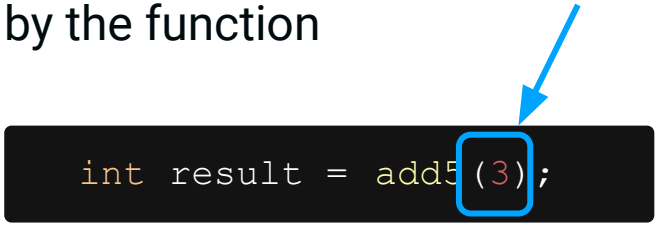
- A **function** is a reusable sequence of statements designed to do a particular job
- We use a **function call** to tell the program to execute the function



- The example above includes a call to some function **`add5()`**

Functions (Cont'd)

- Function calls include **arguments** which are used and/or manipulated by the function



- After the function executes, a **return value** replaces the original function call
  - The example function **`add5()`** *returns* the argument + 5

Function Declaration

- A **function declaration** is needed to designate a new function

```
returnType identifier(paramType paramName);
```

- The **return type** is the **data type** of the return value
  - Some functions have **no return value**, so the return type is **void**
- The **identifier** is the **name** of the function
  - **Naming Convention:** Function names are written in **camelCase**

Function Declaration (Cont'd)

```
returnType identifier(paramType paramName);
```

- A function has **parameters** which are assigned/bound to the arguments of the function call
  - Functions may have **multiple parameters** or none
    - **Syntax**: Each parameter is separated by a **comma** (,) in the parenthesis
  - These parameters are used as **variables** in the body of the function definition…

Function Definition

- The **function definition** is where the function's code is implemented

```
returnType identifier(paramType paramName) // function header
{
    // function body - where the function's code goes
}
```

- The **header** must match the function declaration
- The body must contain a **return statement** if the return type is not void

Function Definition (Cont'd)

```
int add5(int x)
{
    return x + 5;
}
```

- The example function **`add5()`** has the `int` parameter **`x`**
- The function **returns** an `int`, which is the sum of **`x`** `+ 5`

```
int result = add5(3);
```

This call **returns 8**

Function Definition (Cont'd)

I/A

```
int myFunc(bool a, int b, float c)
{
    if (a)
    {
        return b;
    }
    return 0;
}
```

Which is the correct function call for myFunc?

A. `myFunc(false);`

B. `myFunc(true, 3, 0.23);`

C. `myFunc(3, 0.58);`

D. `myFunc(false, 3.10, 2);`

Main Function

```
int main()
{
    // This is the starting point for program execution.
    // Write the code you want to run here.
}
```

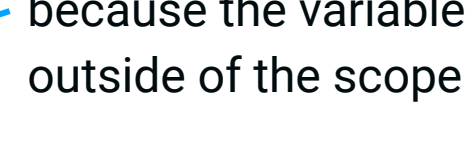
- Every C++ program starts at **`main()`**
  - You *must* define the main function. There is no declaration; it's built in.
- Write the code you want to run in the body of **`main()`**

Local Scope

- Variables created within functions and loops are called **local variables**
- **Local variables** are visible *only* within the **scope** of that function or loop after being declared
    - meaning… local variables cannot be accessed outside of the scope

```
int add5(int x)
{
    return x + 5;
}

x = 2;
```



The code here results in an **error** because the variable **`x`** is referenced outside of the scope of **`add5()`**

Global Scope

- Variables created outside functions and loops are called **global variables**
- **Global variables** have **file scope**, which means they are visible *anywhere* in the file *after* being declared
    - You **cannot reference an identifier before its declaration**
- It is best practice to only declare global variables with the `const` keyword (for constant variables)

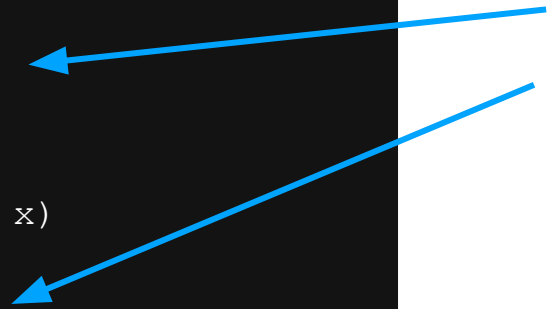
```
const float PI = 3.14159;
```

Global Scope (Cont'd)

```
const float PI = 3.14159;

int addPi(int x)
{
    return x + PI;
}

int subtractPi(int x)
{
    return x - PI;
}
```



The variable `PI` is **accessible at function scope** because it was **defined globally** before it was referenced

SECTION VII

Classes

Classes

- A **class** is a user-defined data type
- Classes may contain **member variables** and **functions**
- **Instances** of a class, or **objects**, are created as follows:

```
className objectName(arg1, arg2, ...);
```

- Objects are created using a special member function called a **constructor**, whose arguments are given at declaration
- To initialize the object as a default version ditch the parentheses

Classes (Cont'd)

```
Oven easyBake;   // instance of the Oven class, default initialized
Oven myOven("Red"); // another oven initialized with a constructor

easyBake.contents = "cookie dough"; // accessing a member variable
easyBake.setTemp(450); // calling a member function
easyBake.bake();
```

- We have created an object of the class **`Oven`**, oven called **`easyBake`**
- **`easyBake`** has member variables and functions which can be accessed using the **dot (.)** operator

SECTION VIII

Includes, iostream, and string

#include

- **#include** is a directive which inserts to contents of a file into the current file
- A **header file** contains declarations of classes, functions, and variables that can be accessed using #include

```
#include <iostream>
```

This example is an include of a header file which contains functionality for inputting/outputting text to the terminal

iostream

```
#include <iostream>
```

- The **iostream** header gives us access to a library of objects and functions that support input/output
- Input/output is managed through **streams** - sequences of bytes that represent data
- We use **standard output stream** object `std::cout` for output to the terminal
- The **standard input stream** object `std::cin` is for input from the terminal

std::cout

- We can insert data into **`std::cout`** using the **stream insertion operator** **<<**

```
int x = 4;
std::cout << x;
```

`x` is *inserted* into the standard output stream

```
int x = 4;
int y = 9;
std::cout << x << 7 << 'w' << y;
```

We can insert more data in the same statement

std::cout (Cont'd)

- We can insert data into **`std::cout`** using the **stream insertion operator** **<<**

```
int x = 4;
std::cout << x;
```

Output: `4`

```
int x = 4;
int y = 9;
std::cout << x << 7 << 'w' << y;
```

Output: `47w9`

std::cout (Cont'd)

- We use several **escape sequences** - special character combinations which carry additional meaning beyond their literal values - to enhance the output
- Start a **new line** using `'\n'` or `std::endl`

```
std::cout << 4 << '\n' << 6;
```

```
std::cout << 4 << std::endl << 6;
```

Output:

```
4
6
```

- Use `'\t'` to **insert a tab**

std::cin

- We can read data from **`std::cin`** using the **stream extraction operator** **>>**

```
int x;
std::cin >> x;
```


Input is *extracted* from the standard input stream and assigned to **`x`**

- **`std::cin`** reads until it hits **any whitespace** (space, tab, or newline)
  - If you type `321 5` into the terminal while executing the code above...

    `x` will only be assigned to `321`

std::string

```
#include <string>
```

- The **string** header gives us support for managing “strings” of characters, which make text
- We can create `std::string` objects and assign them to **string literal values**

```
std::string str = "Test String";
```

std::string (Cont'd)

- Strings can be **concatenated** with the + or += operators

```
std::string a = "Apple";
std::string b = "Banana";
std::string c = a + b + "Orange";
```

c == "AppleBananaOrange"

- Strings can be **indexed** just like arrays!
  - They are technically character arrays

```
std::string a = "Apple";
a[1] = 'm';
```

a == "Ample"

SECTION IX

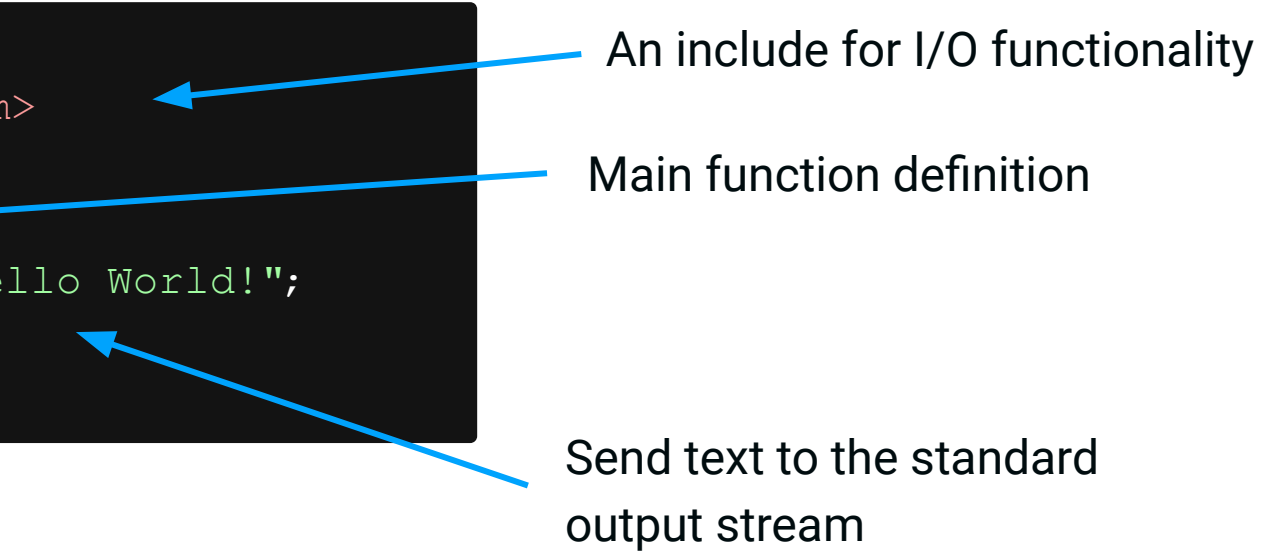
Your First Program

Hello World!

Let’s use everything we’ve learned to write a program that prints some text to the terminal…

```
#include <iostream>

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



SECTION X

Integrated Development Environments

Integrated Development Environments

- An **integrated development environment** (**IDE**) is a suite of applications used for software development
  - It is *more* than a text editor for you to write source code
  - It includes a **source-code editor**, **compiler**, **debugger**, and **automation tools**
    - When you create a program, you will write and edit it in the **source-code editor**
    - The IDE will come with a built-in **compiler** for code translation
    - The **debugger** will help troubleshoot errors and unexpected behavior while the program is running

Popular IDEs

- Common **IDEs for C++** developers include:
  - Visual Studio
  - CLion
  - XCode

- There are multi-language **cloud-based IDEs** as well:
  - Replit
- Learn more about how to use select IDEs in our workshop video!

SECTION XI

C++ Extras

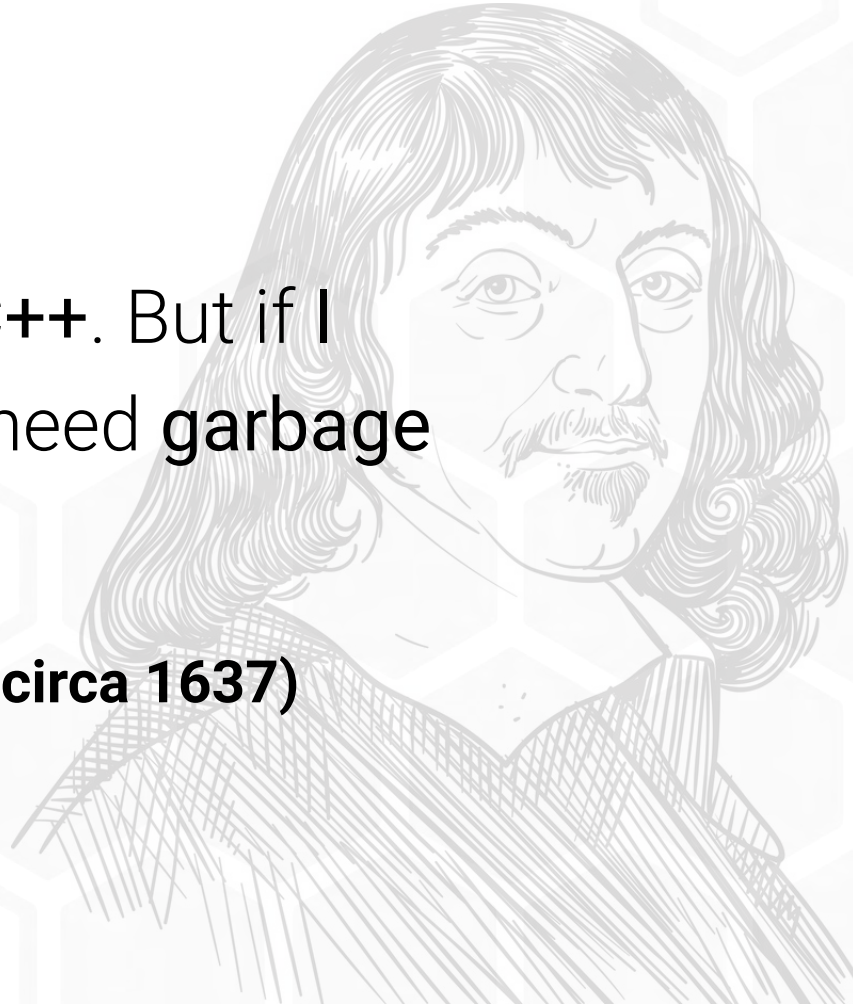
Bonus Topics

- Structs
- Function overloading
- Recursion
- Pointers
- Dynamic memory allocation
- Type casting
- Go to www.learncpp.com and learn more!

“**I think**, therefore **I use C++**. But if **I think too much**, I might need **garbage collection**.”

René Descartes (circa 1637)

Famous Misquotes



FAIR USE DISCLAIMER

Copyright Disclaimer under section 107 of the Copyright Act 1976, allowance is made for “fair use” for purposes such as criticism, comment, news reporting, teaching, scholarship, education and research.

Fair use is a use permitted by copyright statute that might otherwise be infringing.

Non-profit, educational or personal use tips the balance in favor of fair use.