Revised: 7/24/2019

C++ Introduction

Michael C. Hackett

Computer Science Department

Community College of Philadelphia

A little background on C++

- Released in 1985 by Bjarne Stroustrup.
  - Started in 1979 as “C with Classes”
  - Renamed “C++” in 1983

- Added classes/object-oriented programming (among other things) to the C programming language.

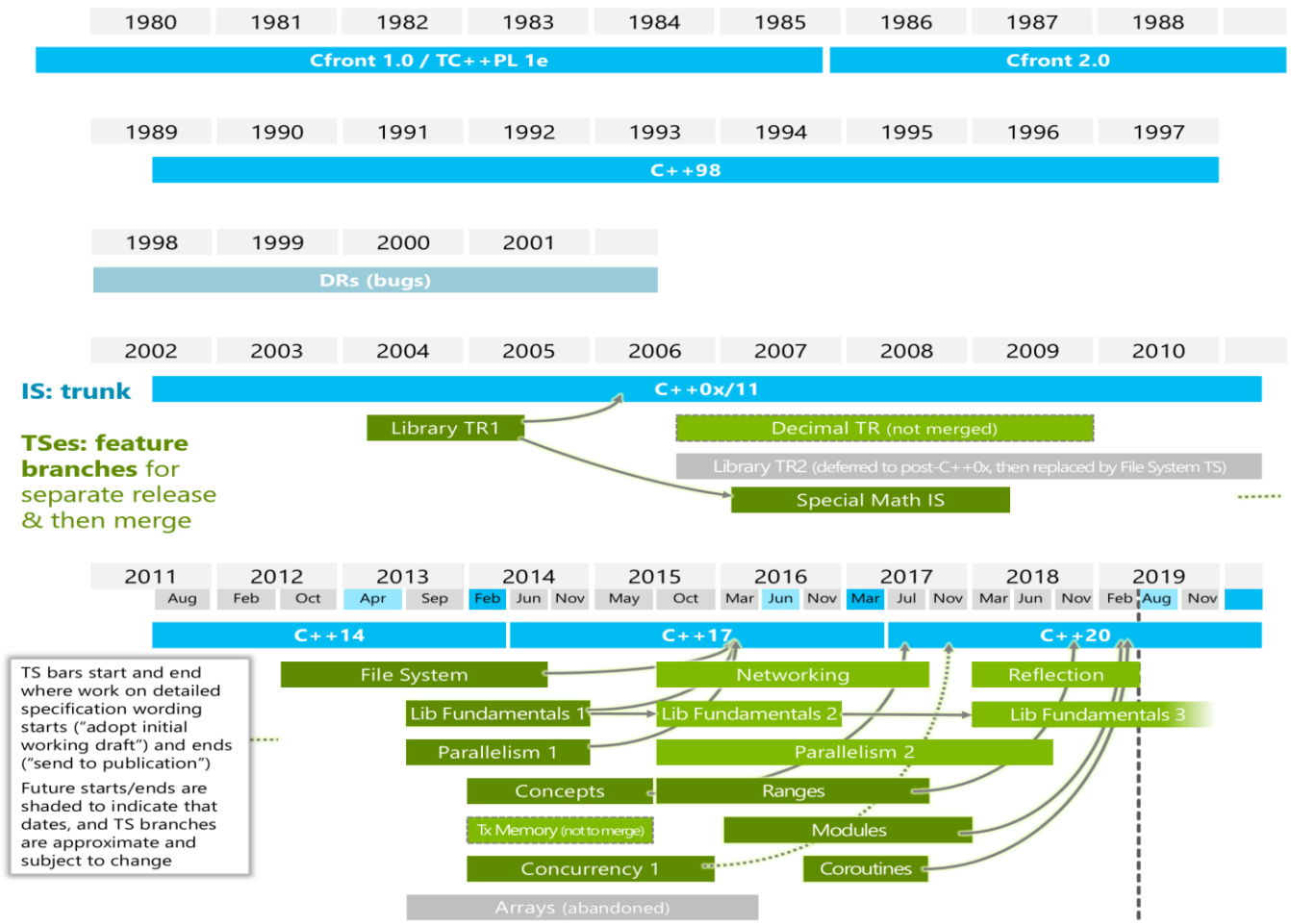
- General-purpose, high-level, compiled language.

A little background on C++

- C++ is standardized by an ISO (International Standards Organization) group.
  - https://isocpp.org/

- Standards (kind of like “versions”):
  - C++98: 1998-2013
  - C++03: 2003-2011
  - C++11: 2011-2014
  - C++14: 2014-2017
  - C++17: 2017-2020 (Current standard)
  - C++20: 2020-? (Expected sometime in early 2020)

A little background on C++



https://isocpp.org/std/status

Creating a C++ Program

- C++ source code is written in a plaintext file.
  - Compiled to an executable program (.exe file on Windows, for example)
- The main function is the starting point for a C++ application.
  - The main function must be written as shown below.
  - It is similar to the main method in a Java program.

```
int main() {

}
```

Creating a C++ Program

- Unlike the main method in a Java program (return type is void), the return type of the C++ main function is int.
- The main function typically returns 0 when finished.

```
int main() {
  //Statements to execute

  return 0;
}
```

Creating a C++ Program

- Java shares a similar syntax to C++.
- Most of the basics require no “re-education”.
- The point of these slides is to highlight the key differences.

Preprocessor Directives

- A **preprocessor directive** is a statement at the beginning of the source code that includes the functions/objects of other source code.
  - Similar to an import statement in Python or Java.

- The include statements allow the program to use the code of header files and libraries.
  - Explained later.

Preprocessor Directives

- Syntax:

```
#include <header>
```

- *header* is replaced with the actual name of the header file.

Preprocessor Directives

- The first header we will see is the **iostream** header.
  - Part of the standard input/output library.

- Will allow the program to print output and get keyboard input.

```
#include <iostream>
```

Preprocessor Directives

```
#include <iostream>

int main() {
   //Statements to execute

   return 0;
}
```

Console Output

- The statement below prints the string literal “Hello World!” as the program’s output.
  - iostream header must be included.

```
std::cout << "Hello World!";
```

Console Output

```
std::cout << "Hello World!";
```

- **std::cout** indicates the standard output stream, which is usually printing to the console/screen.

- **<<** in this context is the *insertion operator*.
  - In other contexts, it is the left-shift bitwise operator.
  - Indicates what is being inserted into the standard output stream.

Console Output

```
#include <iostream>

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

   return 0;
}
```

Output:

```
Hello World!
```

Console Output

```
std::cout << "Hello World!";
```

Is equivalent to the following Java statement:

```
System.out.print("Hello World!");
```

Console Output

```
#include <iostream>

int main() {
   std::cout << "Hello World 1";
   std::cout << "Hello World 2";

   return 0;
}
```

Output:

```
Hello World 1Hello World 2
```

Console Output

- To end a line (insert a line break), use the following:

```
std::cout << "Hello World!" << std::endl;
```

Console Output

```
std::cout << "Hello World!" << std::endl;
```

Is equivalent to the following Java statement:

```
System.out.println("Hello World!");
```

Console Output

```
#include <iostream>

int main() {
   std::cout << "Hello World 1" << std::endl;
   std::cout << "Hello World 2";

   return 0;
}
```

Output:

```
Hello World 1
Hello World 2
```

Console Output

- You can use the insertion operator multiple times in a single output statement:

```
std::cout << "Hello" << " " << "World!" << std::endl;
```

Output:

```
Hello World!
```

Namespaces

- A **using declaration** allows us to specify a namespace (which are used to prevent name conflicts) that we want to include in our program.

- For example:

```
using std::cout;
```

would allow us to simply use **cout** instead of **std::cout** in our program.

Namespaces

- Using declarations typically appear after any preprocessor directives.

```
#include <iostream>

using std::cout;

int main() {
   cout << "Hello World 1" << std::endl;
   cout << "Hello World 2";

   return 0;
}
```

Namespaces

- We can use a using declaration to use an entire namespace.

- For example:

```
using namespace std;
```

would allow us to no longer need to use the **std::** prefix in our program.

Namespaces

See Example1_ConsoleOutput.cpp
See Example2_ConsoleOutput.cpp
See Example3_ConsoleOutput.cpp

- No need to use **std::** before cout or endl

```
#include <iostream>

using namespace std;

int main() {
   cout << "Hello World 1" << endl;
   cout << "Hello World 2";

   return 0;
}
```

Fundamental Data Types

- Similar concept to the eight Java primitives, which are simplified versions of C++'s fundamental types.

- Boolean type
- Integer type
- Floating Point type
- Character type

Boolean Type

- **`bool`** (1 byte/8 bits; Only one bit is used though)
  - Can be **`true`** or **`false`**.

Integer Types

- **`int`** (4 bytes/32 bits)
  - Can represent any integer between -2,147,483,648 and 2,147,483,647
  - C++ standard says a minimum of 16 bits for an int, but it will be 32 bits on modern computers.

- Modifiers
  - **`signed`** and **`unsigned`**
  - **`short`** and **`long`**

Integer Types

Specifier	Equivalent to	Range	Size
`short`	`short int`	-32768 through 32767	16 bits (2 bytes)
`short int`			
`signed short`			
`signed short int`			
`unsigned short`	`unsigned short int`	0 through 65535	
`unsigned short int`			

`short int` is equivalent to Java's **`short`** type

Integer Types

Specifier	Equivalent to	Range	Size
`int`	**`int`**	-2,147,483,648 through 2,147,483,647	32 bits* (4 bytes)
`signed`			
`signed int`			
`unsigned`	**`unsigned int`**	0 through 4,294,967,295	
`unsigned int`			

*The C++ standard specifies *at leas*t 16 bits for the **`int`** type. It's generally going to be 32 bits on modern systems.

Technically, **`int`** is also equivalent to Java's **`short`** type; on modern systems it is like Java's **`int`** type.

Integer Types

Specifier	Equivalent to	Range	Size
`long`	**`long int`**	+/- $9.22 * 10^{18}$ (Approx)	64 bits* (8 bytes)
`long int`			
`signed long`			
`signed long int`			
`unsigned long`	**`unsigned long int`**	0 through $1.84 * 10^{19}$	
`unsigned long int`			

*The C++ standard specifies *at leas*t 32 bits for the **`long int`** type. It's generally going to be 64 bits on modern systems.

Technically, **`long int`** is the equivalent to Java's **`int`** type; on modern systems it is like Java's **`long`** type.

Integer Types

Specifier	Equivalent to	Range	Size
`long long`	`long long int`	+/- $9.22 * 10^{18}$ (Approx)	64 bits (8 bytes)
`long long int`			
`signed long long`			
`signed long long int`			
`unsigned long long`	`unsigned long long int`	0 through $1.84 * 10^{19}$	
`unsigned long long int`			

*The C++ standard specifies *at leas*t 64 bits for the **`long long int`** type.
`long long int` is equivalent to the **`long`** type in Java

Integer Primitives

- It's a lot to remember, but I only expect you to use:
  - **`short int`** (will act like Java's short)
  - **`int`** (will act like Java's int)
  - **`long int`** (will act like Java's long)

Floating Point Types

- **`float`** (usually 4 bytes/32 bits) (Same as Java)
- **`double`** (usually 8 bytes/64 bits) (Same as Java)

- **`long double`** (usually 12 bytes/96 bits on modern systems)
  - Extended precision
  - Doesn't always map to the IEEE-754 floating point standard

Character Type

- **`char`** (1 byte/8 bits)
  - Can represent a single, 8-bit Unicode character.
  - Can represent an 8-bit number

- This is sort of a mix between Java's byte and char primitive types.

Character Types

Specifier	Range (Numeric)	Size
`char`	-128 through 127	8 bits (1 byte)
`signed char`		
`unsigned char`	0 through 255	

`char` is equivalent to Java's **`byte`** type with regard to integer data
`char` is like Java's **`char`** type with regard to character data, but is only 8 bits (Java's char is 16 bits)

Variables

See Example4_VariablesAndTypes.cpp

- Variables are declared and used in an identical fashion to Java:

```
int age;
age = 22;

double temp = 98.6;
```

- Like Java, variables in C++ are ***statically typed***.
  - It cannot reference values of a different data type after it has been declared.

Literals

- Similar rules as found in Java for long literals (needs an l or L) and float literals (needs an f or F)

```
long int exampleLongLiteral = 255L;
float myExampleFloat = 15.5F;
```

Literals (Char)

- char literals can be expressed as a character literal or a number.
- **Use single quotes for characters.**

```
char exampleCharLiteral = 'A';
char example8BitNumber = 65;
```

Naming Rules for Variables

- Same rules as Java.

- A variable's data type cannot be changed after declaration.
- Keywords cannot be the name of a variable.

- "Camel-case" is the convention used for variable names in C++, as it was in Java.

Naming Rules for Variables

- Names must start with a letter, dollar sign, or underscore.
- Names may contain numbers, but **cannot** start with numbers.
- Aside from letters, dollar signs, underscores, and numbers, no other characters may be used.
- Names cannot contain spaces. Use underscores, if necessary.

```
int someName;        Valid.

int _someName;       Valid. Can start with underscore.
int some_Name;       Valid. Can contain any underscores.

int $someName;       Valid. Can start with dollar sign.
int some$Name;       Valid. Can contain any dollar signs.

int 0someName;       INVALID. Can't start with a number.
int some0Name;       Valid. Can contain any numbers.

int some Name;       INVALID.
int some_Name;       Valid.
```

Constants

See Example6_Constants.cpp

- Constants (variables that can't be changed) are declared by using the **const** keyword.
  - By convention, the name of a constant is in uppercase with underscores between words.

```
const int FREEZING_POINT = 32;
```

Comments

- Same as Java
- // - Inline Comments
- /* */ - Multiline Comments

Strings

See Example7_Strings.cpp

- The string object is provided through the iostream header.
- Without std namespace, must specify std::string

Declaring/Initializing Strings

- Lowercase s in string.

```
string hello = "Hello There!";
```

- Without using the std namespace:

```
std::string hello = "Hello There!";
```

String Concatenation/Appending

- Same as Java

- Concatentation: +
- Appending: +=

See Example8_ConcatenationAndApending.cpp

Keyboard Input

```
std::cin >> someVariable;
```

- **std::cin** indicates the standard input stream, which is usually keyboard entries.

- **>>** in this context is the *extraction operator*.
  - In other contexts, it is the right-shift bitwise operator.
  - Indicates data is extracted from the standard input stream and stored into the provided variable.

Keyboard Input

```
#include <iostream>

using namespace std;

int main() {
    string name;
    cout << "Enter your name: ";
    cin >> name;
    cout << "Hello, " + name + "!" << endl;

    return 0;
}
```

Output:

```
Enter your name: John
Hello, John!
```

Arithmetic Operators

- Same as Java

- Addition: **+**
- Subtraction: **-**
- Multiplication: *****
- Division: **/**
- Remainder: **%**

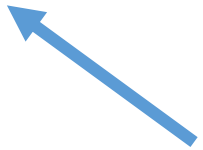
- Operator precedence is the same as Java.
- Augmented Assignment (+=, -=, etc) is the same as Java.

See Example11_TypeConversion.cpp

Typecasting

- The process is identical to Java.

```
double myDouble = 453.87;
int myInt;
myInt = (int)myDouble;
```



Put the desired data type, in parenthesis, before the variable name to ***typecast*** the value as that type.

In the example above, the value stored at the memory location referenced by myInt is 453 not 453.87

Typecasting Numbers to Strings

- Use the std::to_string() function.
  - Or simply call to_string() when std namespace is used.

```
double myDouble = 453.87;
string doubleString = to_string(myDouble);
```

Typecasting Numbers to Strings

- Numeric data must be converted to strings before you can concatenate/append.

```
int myInt = 123;
string output = "Your number is: " + to_string(myInt);
```

Typecasting Integers from Strings

- Use the std::stoi() function.
  - Or simply call stoi() when std namespace is used.
  - stoi = “string to int”

```
string numString = "123";
int myInt = stoi(numString);
```

Typecasting Doubles from Strings

- Use the std::stod() function.
  - Or simply call stod() when std namespace is used.
  - stod = “string to double”

```
string numString = "456.78";
double myDouble = stod(numString);
```

Math Header

See Example12_MathHeader.cpp

- Provides access to mathematical functions like rounding, exponents, and square roots.
  - Similar to Java's Math object

```
#include <cmath>
```

- Provides a number of familiar functions like:
  - sqrt, pow, round, ceil, and floor

Relational Operators

- Same Operators as Java

- ==, !=, <, >, <=, >=

Logical Operators

- Same Operators as Java

- &&, ||, !

If Statements

- Same syntax as Java

- See Example1_IfStatements.cpp

Loops

- For, while, and do-while loops have same syntax as Java.

- See Example2_ForLoops.cpp
- See Example3_WhileLoops.cpp
- See Example4_DoWhileLoops.cpp

Subroutines and Functions

See Example1_BasicSubroutine.cpp

- Generally, the syntax for a subroutine/function/method is the same as Java.

- However, the subroutine must be defined before it can be called.

Prototypes

See Example2_SubroutinePrototypes.cpp
See Example3_Arguments.cpp

- Placing a prototype at the beginning of the program declares the subroutine, its return type and parameter list types.

- This allows the subroutine to be defined anywhere in the source code.

Arrays

See Example1_ArrayBasics.cpp

- The basic usage of an array is the same as it was in Java.
- The declaration is a little different.
  - The [] comes after the variable name.

- With data: `int numArray[] = {4, 7, 9, 10};`

- Without data: `int numArray[5];`
  - The number specifies the length.

Arrays

See Example3_ArrayLength.cpp

- To find the length of an array, you need to use the sizeof function to determine the total byte size.
- Then, divide by the byte size of each element

```
int numArray[] = {4, 7, 9, 10};
int byteSize = sizeof(numArray);
int length = byteSize / 4;
```

ints are 4 bytes in size

Arrays

- Alternatively…

```
int numArray[] = {4, 7, 9, 10};
int length = sizeof(numArray) / 4;
```