CS 3460

Introduction to the C++ language

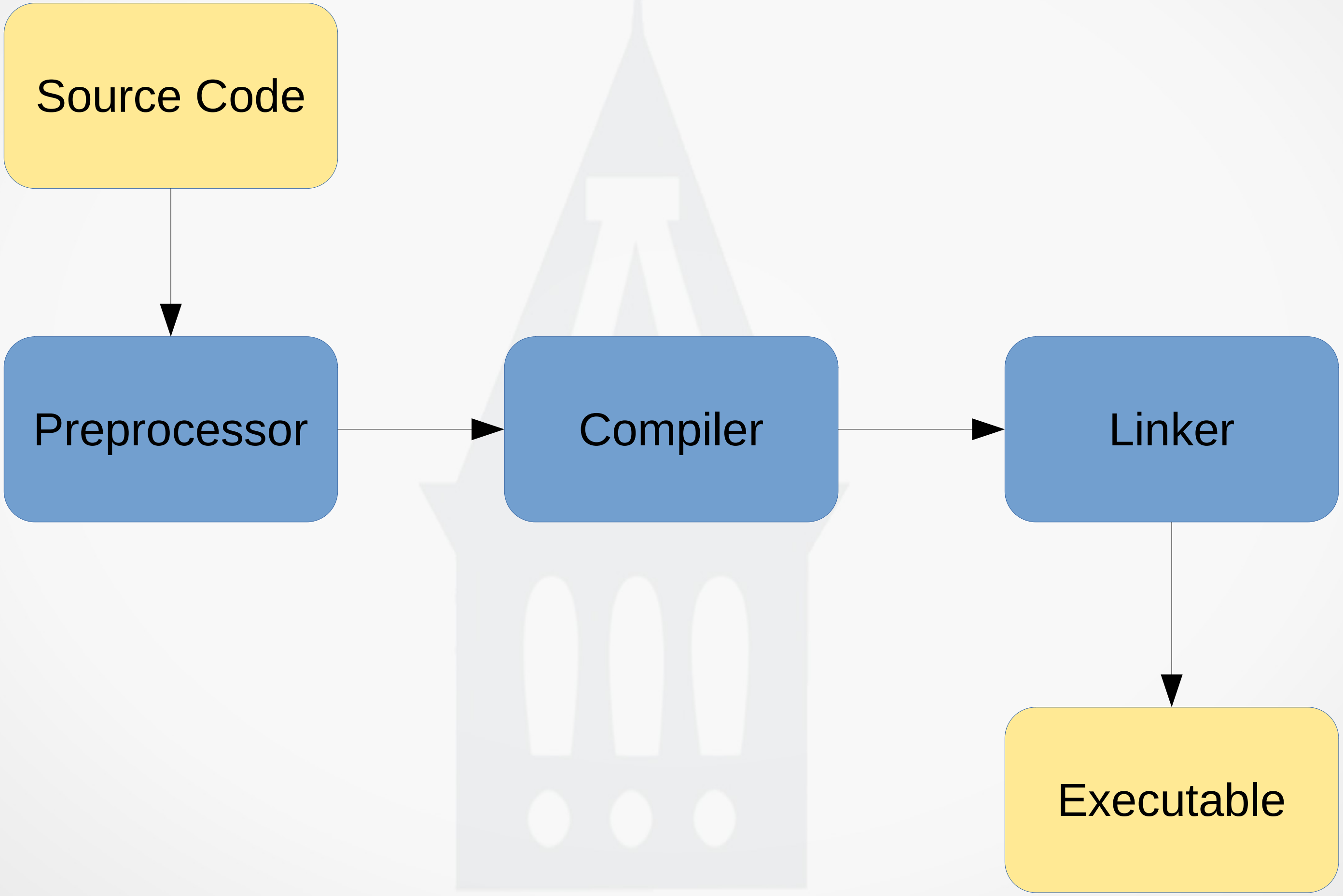
Language Overview

- A massive, multi-paradigm language
  - Imperative, Procedural
  - Structured, Object-Oriented
  - Functional
  - Generic
- Intended to enable high-performance code
  - “Leave no room for a lower-level language below C++ (except assembler)”

Language Features

- Cross-platform, compiled to native OS binary
- Conditional flow-control
- Looping structures
- Dynamic memory
- Classes/Objects
- Polymorphism
- Operator overloading
- Exception handling
- Functional programming
- Generic programming
- Compile time code evaluation
- Large standard library

A Compiled Language



Processing Steps

- **Preprocessor**
  - Any statement that begins with #, is a compiler directive
    - Provides an instruction to the compiler/preprocessor
  - Modifies the C++ code according to the directive
  - Called the “first pass” over the C++ code
- **Compiler**
  - Translates the C++ code into mostly executable code
  - Placeholders for items it is unable to resolve (e.g., functions in libraries)
- **Linker**
  - Resolve the placeholders, based on info from other compiled code
  - Create the final executable, ready for execution

First C++ Program

```
#include <iostream>

//
// This is the main entry point of a C++ program.
//
int main(int argc, char* argv[])
{

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

    return 0;
}
```

First C++ Program

```
#include <iostream>

//
// This is the main entry point of a C++ program.
//
int main(int argc, char* argv[])
{

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

    return 0;
}
```

- The `#include` directive
- Comments
- Entry point – plus command line params
- Data types & arrays
- Function declaration, scope, and return type
- Statements
- Streaming output to the console

First C++ Program

```cpp
#include <iostream>

//
// This is an alternative main function
//
int main()
{

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

    return 0;
}
```

Language Basics

Primitive Data Types

- `char` – Used for storing characters (ASCII); 8 bits
- `int` – Integer type; size depends, typically 32 bits
- `float` – Single precision float type; size depends, typically 32 bits
- `double` – Double precision float type; size depends, typically 64 bits
- `bool` – Boolean type (`true`/`false`); size depends, typically 8 bits
- `void` – Indicates no value; no size.
- (modifier) `signed` – Type to have negative and positive range
- (modifier) `unsigned` – Type to have only positive range
- (modifier) `short` – Reduces type storage by half; reduces range
- (modifier) `long` – Increases type storage by two; increases range

Raw Arrays

- `type name[size];`
  - Declares and creates storage for the array
  - Storage is on the stack, not the heap
  - Similar statement in Java only creates an array reference, not the array elements

Raw Arrays

- `type name[size];`
  - Declares and creates storage for the array
  - Storage is on the stack, not the heap
  - Similar statement in Java only creates an array reference, not the array elements

```
int primes[4];

primes[0] = 2;
primes[1] = 3;
primes[2] = 5;
primes[3] = 7;
```

Raw Arrays

```
int primes[] = { 2, 3, 5, 7 };
```

- Declares, creates storage for 4 elements, and initializes the elements to the values
- Storage is on the stack, not the heap

```
Person employees[10];
```

- Declares, creates storage for 10 elements
- Storage is on the stack, not the heap
- Similar statement in Java only creates an array reference, not the array of references, or even they array of objects

Raw Arrays

- Multi-dimensional arrays are declared as…

```
int table[3][3];

table[0][0] = 0;
table[0][1] = 1;
table[0][2] = 2;
table[1][0] = 3;

int table[3][3] = { {0, 1, 2 }, { 3, 4, 5 }, { 6, 7, 8} };
```

Raw Arrays

- Multi-dimensional arrays are declared as…

```
int table[3][3];

table[0][0] = 0;
table[0][1] = 1;
table[0][2] = 2;
table[1][0] = 3;

int table[3][3] = { {0, 1, 2 }, { 3, 4, 5 }, { 6, 7, 8} };
```

- With all this said, I don't recommend the use of raw arrays in C++! We'll cover more array types…
  - `std::array`
  - `std::vector`

Function Declarations

- Functions have the general form of…

```
<return type> name(<parameters>)
{
   … statements …

   return <value>;
}
```

Function Declarations

- Functions have the general form of…

```
<return type> name(<parameters>)
{
   … statements …

   return <value>;
}
```

- `<return type>` any valid data type, including `void`
- `name` is any valid C++ identifier
- `<parameters>` zero or more parameters; can also define default values (Java does not have default param values)
- function body between `{ }` defines scope (same as Java)

Statements

- ; (semicolon) is used to terminate a statement (same as Java)
  - A semicolon itself ***is*** a statement
  - Some rules for semicolons are different from Java, we'll talk about them as they come up. e.g. class declaration/definitions
  - Preprocessor directives are not statements, they don't end with semicolons
- Any number of statements in a function body or `{ }` block

Console Output

- Header file `<iostream>`
- `std::cout` is the streaming target for console output
  - `std` is a namespace; much more later
  - `cout` is a static object
- `<<` insertion operator
- `>>` extraction operator

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

Numeric Limits

- `std::numeric_limits<>`
  - In the `<limits>` header/library
- Allows a program to determine (a lot of) details about primitive numeric types
  - min/max value of a type
  - is it integral, signed
  - rounding error (float types), rounding modes
  - much, more!
- Also `sizeof()` to get the number of bits for a type

`std::numeric_limits` – Code Demo

Default Values

- C++ does not define the default value for *uninitialized* variables
  - Java does, C++ doesn’t
- For example
  - In Java, integral types are initialized to 0
  - In C++ there is no defined initialization. It might be 0, but that is up to the compiler, you can’t count on it
- Uniform Initialization (more on this at a later time)

```
int i{0}; // initializes i to 0
int i{}; // initializes i to the int “default” value
MyType t{}; // initializes t to the MyType “default” value
```

Additional Integer Types `<cstdint>`

- `std::intmax_t, std::uintmax_t`
- `std::int8_t, std::uint8_t`

  …
- `std::int64_t, std::uint64_t`
- `std::int_least8_t, std::uint_least8_t`

  …
- `std::int_least64_t, std::uint_least64_t`

Literals & Constants

- Literal
  - A value, numeric, boolean, or string located anywhere in the code
- Constant
  - Defined using `const` or `constexpr` (more later)

Literals & Constants – Code Demo

Type Conversions

- Implicit : Recognized by the compiler and automatically performed

```
short a{1000};
long b = a; // also long b{a};

float pi1{3.14159f};
double pi2 = pi1; // also double pi2{pi1}
```

Type Conversions

- Explicit – Type Casting
  - Consider this code…

```
long c{1000};
short d{c};
```

  - To eliminate the warning, need to type cast

```
short d{static_cast<short>(c)};
```

Type Conversions

- General form: `static_cast<type>(var)`
- Other casting operators
  - `const_cast<type>`
  - `dynamic_cast<type>`
  - `reinterpret_cast<type>`
- From C and earlier C++ you will also see

```
short d = short(c);
short e = (short)c;
```

- Don’t do these!! Why…
  - `const_cast`
  - `static_cast`
  - `reinterpret_cast`

Strings – c-string

- One dimensional array of `char`s, null terminated

```
char name[5] = {'D', 'e', 'a', 'n', '\0'};
```

  - `\0` is the null termination character

don’t use them, just don’t!

Strings – `std::string`

- Part of the standard library: `<string>`
- High-level manipulation
  - construction
  - copying
  - manipulation
  - etc.
- Knows its size: `.size()`
- Access individual characters using the `[]` operator
- Unlike Java, they are mutable

`std::string` – Code Demo

String Views

- `std::string_view`
  - A lightweight, read-only window into an `std::string`
  - Use when you don’t want/need copies of strings, or parts of strings

`std::string_view` – Code Demo

Loops

- Quite similar to Java, but some important differences
- Types
  - `while`
  - `do { } while`
  - `for`
    - counted

      `for (init-statement; loop-condition; iteration-expr) {}`

    - range-based

      `for (init-statement; range-declaration : range-expr) {}`

- *loop-condition* may resolve to boolean or numeric
  - interpreted same as conditional statements

Loops – Range Based

- Same syntactical pattern as Java
- The *range-expression* has some requirements
  - Can be a raw array
  - Any object with compatible `.begin` and `.end` methods or free functions (We'll talk about `.begin` and `.end` eventually)
    - `std::array` and `std::vector` provide them

```
std::array primes{ 2, 3, 5, 7 };
int sumOfPrimes{0};
for (int prime : primes)
{
    sumOfPrimes += prime;
}
```

Loops – Range Based

- The *init-statement* is optional
  - Often used to define a type
  - Also used to declare and initialize a counter
  - Scope is the loop

```
std::array primes{ 2, 3, 5, 7 };
for (auto which{1}; int prime : primes)
{
    std::cout << std::format("The {} prime is {}\n", which++, prime);
}
```

Range Based Loops – Code Demo

Functions

- Before a function can be called, the C++ compiler must already know of its existence
  - This is different from Java, where you can write a static method after the place where it is called

```
std::array<int, 4> byTwo(std::array<int, 4> values)
{
    for (std::uint8_t i = 0; i < values.size(); i++)
    {
        values[i] *= 2;
    }

    return values;
}

int main(int argc, char* argv[])
{
    std::array primes{ 2, 3, 5, 7 };

    primes = byTwo(primes);
     ... more code here ...

    return 0;
}
```

Functions – Prototype

- Alternative is to declare a function prototype (declaration), then implement (definition) the function later

```
std::array<int, 4> byTwo(std::array<int, 4> values);

int main(int argc, char* argv[])
{
    std::array primes{ 2, 3, 5, 7 };

    primes = byTwo(primes);
    for (auto prime : primes)
    {
        std::cout << prime << std::endl;
    }

    return 0;
}

std::array<int, 4> byTwo(std::array<int, 4> values)
{
    for (std::uint8_t i = 0; i < values.size(); i++)
    {
        values[i] *= 2;
    }

    return values;
}
```

Functions - Pass-by-Value

- By *default*, function parameters are pass-by-value
  - Same meaning as Java; the value is copied
  - As with Java, you have to pay attention to what the ‘value’ is; but it is ***more complex*** in C++ than Java
- Return values are also by-value; return by-value
- In the code example…
  - `values` parameter is by-value; copy made
  - `return values` is by-value; copy made

```
std::array<int, 4> byTwo(std::array<int, 4> values)
{
    for (std::uint8_t i = 0; i < values.size(); i++)
    {
        values[i] *= 2;
    }

    return values;
}
```

Pass/Return by Value – Code Demo

Functions – Default Parameter Values

- Function parameters may specify a default value

```
std::array<int, 4> byN(std::array<int, 4> values, int n = 2);
```

- This function can be called as…

```
auto result = byN(primes);
auto result = byN(primes, 4);
```

- More than one parameter may have a default

```
std::vector<int> createArray(int size = 10, int initialValue = 0);
```

```
auto array1 = createArray();
auto array2 = createArray(20);
auto array3 = createArray(20, 1);
```

Multiple Files

- Java will resolve a method called from one file that is found in another; you don’t have to do anything (or much)
- C++ requires telling the compiler where to find functions, classes, etc. (translation units, more on this later)
- First step, separate header and implementation files
  - Prototypes, class declarations, etc. in a header file; .hpp
  - Definitions in an implementation file; .cpp

Multiple Files

- Header file (.hpp)
  - First line: `#pragma once`
  - prototypes
  - class declarations
  - constants
- Implementation file (.cpp)
  - `#include "utilities.hpp"`
    - `""` search local folders first
    - `<>` search compiler folders

Header/Implementation Files – Code Demo