C++

Introductory material

Michael Burrell

January 7, 2020

Readings for this set of slides

Do this. . .

C

- Before we talk about C++, we need to talk about C
- C came about in 1969/1970 as a way to portably write Unix
    - Bell Labs had a new operating system (Unix) that they wanted run on different hardware
    - Creating a program language seemed better than rewriting everything in assembly for a new architecture
- C is fundamentally a systems programming language
    - Its value is writing systems software (operating systems, system utilities, etc.) in a portable way
    - It is sometimes called "the portable assembler"

C

- C is balancing (quite well) between two objectives
  - Be very low-level and expose access to the underlying hardware
  - Be portable and abstract away any differences between hardware

C

- C is balancing (quite well) between two objectives
  - Be very low-level and expose access to the underlying hardware
  - Be portable and abstract away any differences between hardware
- There are some times when we think of something abstractly, but C allows mechanisms to see it concretely, as the machine does
- All of this applies to C++, as well

Object-oriented design

- Then, in the 1970s, a new idea sprouted out of research labs
  - It was called *object-oriented design*
  - It led to two important programming languages called Smalltalk and Simula
- Smalltalk and Simula had *objects* and *classes* which allowed for complex software to be written in a clear way
  - This was difficult to do in C!

Inspirations from Lisp

- Smalltalk, in particular, didn't just invent object-oriented programming
- It took inspiration from earlier *functional* programming languages like Lisp
- It used a style of programming called *metaprogramming*
    - Metaprogramming allowed a lot of power for writing very abstract code

Merging it together

- In 1979, a Bell Labs technician named Bjarne Stroustrup started creating a new programming language called "C With Classes"
- He wanted a low-level portable systems language (like C) with object-oriented design and metaprogramming (like Smalltalk and Simula)

C++ and snowballing

- While developing “C With Classes” (soon renamed C++), a lot of people came to Stroustrup
  - “Can you put in exceptions?”
  - “Can you put in multiple inheritance?”
  - “Can you put in feature X?”
- Infamously (said by other members at Bell Labs), Stroustrup “couldn’t say ‘no’”
- Right from the beginning, C++ was a “kitchen sink” language
  - Today it stands as arguably the most complicated programming language ever made

Timeline since then

C++98 — standardized by ANSI and ISO

C++03 — small bug fix for C++98

C++11 — stronger compatibility with C, type inference, for-each loops, lambdas (functional programming), etc., etc.

C++14 — (used in this course): mostly just a bug fix for C++11

C++17 — mostly syntax cleanups and library additions

C++20 — (not finished yet): big syntax changes, more metaprogramming features

Brief wrapup of C++

- C++ is a portable, low-level systems language
- It incorporates many different programming paradigms (procedural, functional, object-oriented, metaprogramming, etc.)
- It is one of the most complex languages ever made
  - It is not possible to learn (all of) C++ in one course
  - It is not possible to learn (all of) C++ in 10 years
  - We will focus on the major features of C++ which are used most commonly in industry
- The first few months of this course will be learning C++98
- We will stick in a little bit of C++11/C++14 additional features as we become more advanced

Integer widths

- One important example of where C and C++ are both low-level and abstract is with data types
- `char`, `short`, `int`, `long`, `long long`, `float`, `double` and `long double` are all quite loosely defined

Integer widths

- One important example of where C and C++ are both low-level and abstract is with data types
- `char`, `short`, `int`, `long`, `long long`, `float`, `double` and `long double` are all quite loosely defined

Type	**Guarantees in C, C++**
`char`	At least 8 bits in size, holds one character, size is 1

Integer widths

- One important example of where C and C++ are both low-level and abstract is with data types
- `char`, `short`, `int`, `long`, `long long`, `float`, `double` and `long double` are all quite loosely defined

Type	**Guarantees in C, C++**
`char`	At least 8 bits in size, holds one character, size is 1
`short`	At least 16 bits, not smaller than `char`, not bigger than `int`

Integer widths

- One important example of where C and C++ are both low-level and abstract is with data types
- `char`, `short`, `int`, `long`, `long long`, `float`, `double` and `long double` are all quite loosely defined

Type	Guarantees in C, C++
`char`	At least 8 bits in size, holds one character, size is 1
`short`	At least 16 bits, not smaller than `char`, not bigger than `int`
`int`	At least 16 bits, a “natural size” of the machine

Integer widths

- One important example of where C and C++ are both low-level and abstract is with data types
- `char`, `short`, `int`, `long`, `long long`, `float`, `double` and `long double` are all quite loosely defined

Type	Guarantees in C, C++
`char`	At least 8 bits in size, holds one character, size is 1
`short`	At least 16 bits, not smaller than `char`, not bigger than `int`
`int`	At least 16 bits, a "natural size" of the machine
`long`	At least 32 bits, not smaller than `int`

Integer widths

- One important example of where C and C++ are both low-level and abstract is with data types
- `char`, `short`, `int`, `long`, `long long`, `float`, `double` and `long double` are all quite loosely defined

Type	Guarantees in C, C++
`char`	At least 8 bits in size, holds one character, size is 1
`short`	At least 16 bits, not smaller than `char`, not bigger than `int`
`int`	At least 16 bits, a "natural size" of the machine
`long`	At least 32 bits, not smaller than `int`
`long long`	At least 64 bits, not smaller than `long`

Basic integer types

- This means different machines (and different compilers) are free to defined types in different ways
- On a PDP-11 and 16-bit x86, 8-bit byte, 16-bit short, 16-bit int, 32-bit long
- On a CDC 6600, 18-bit byte, 18-bit short, 80-bit int, 80-bit long
- On x86-64 Windows, 8-bit byte, 16-bit short, 32-bit int, 32-bit long
- On x86-64 Linux, 8-bit byte, 16-bit short, 32-bit int, 64-bit long
- On SPARC64 Solaris, 8-bit byte, 16-bit short, 64-bit int, 64-bit long
- On UNICOS, 8-bit byte, 64-bit short, 64-bit int, 64-bit long

How do we write portable code?

- If every compiler makes the data types different, how can our code be portable?
- General rule: don't make unnecessary assumptions
  - Just because `int` is 32-bit on your computer, assume it might be 16-bits (or 80-bits) on someone else's
  - Almost never will you have to rely on a variable being of a specific width
- Also, there are more types that are defined for us. . . .

Other integer types

`size_t` — can represent the size of any object/array in memory. May be defined to be an `int`, or `long` or `long long`. On 64-bit systems, this is (probably) 64 bits, no matter what `int` and `long` are

`ptrdiff_t` — can represent the difference between any two pointers (addresses in memory)

`intptr_t` — can represent any memory address as an integer

In C or C++, *never* use `int` or `long` to represent the length of something: *always* use `size_t`.

sizeof

- C and C++ have a unary operator defined called `sizeof`
- The operand to `sizeof` may be:
  - A type name; or
  - A value (such as a variable)
- `sizeof` returns a value of type `size_t` which is measured in *characters* (*bytes*)
  - Remember that the size of `char` *must* be 1

climits

There is a file called `climits` we can defined which includes a number of constants which are sometimes helpful:

- `CHAR_BIT` — the number of bits in a byte (usually 8)
- `INT_MIN` — -32768 on 16-bit machines, -2147483648 on 32-bit (and many 64-bit) machines
- `INT_MAX` — +32767 on 16-bit machines, +2147483647 on 32-bit/64-bit machines

Similarly, `SHORT_MIN`, `SHORT_MAX`, `LONG_MIN`, `LONG_MAX`, etc.

Signedness

- Every integer type in C and C++ comes in two flavours: signed and unsigned
- Signed integers can represent negative numbers
- Unsigned integers cannot represent negative numbers
- It is not defined how signed integers are represented
  - It's probably 2's complement
  - We shouldn't assume that it will be 2's complement
  - Signed arithmetic overflows in C and C++ cause undefined behaviour
  - Integers are `signed` by default

```cpp
#include <iostream>

using namespace std;

int main()
{
    signed int x = -5; // OK
    unsigned int y = -5;  // NOT OK
    unsigned char z = 200; // OK
    signed char w = 200;  // NOT OK
    int v = w; // int is the same as signed int
    return 0;
}
```