C++ Programming Introduction

Peter Chapin

Vermont State University

CIS-3012, C++ Programming

The Language People Love to Hate

- C++ is…
  - … **old**. The first standard was finished 1998, but the language dates from circa 1980 when it was originally called “C with Classes.”
  - … **very large**. C++ has many features. It has the dubious distinction of being possibly the largest and most complex programming language ever created.
  - … **quirky**. C++ has multiple ways of doing the same thing (for legacy compatibility with itself) and strives for C compatibility.
  - … **hard to master**. Because C++ is so large and quirky, it takes a lot of study to learn it well. As a result, there is a lot of bad C++ out there.
  - … **unsafe**. Because the language supports the same low-level features as C, it is possible to accidentally write programs that break themselves.

OTOH

- C++ is…
  - … **flexible**. You can use multiple programming paradigms and many programming techniques with C++. It gives wide design flexibility.
  - … **powerful**. C++ provides features for building easy-to-use abstractions that, when written well, can make solving hard problems simple.
  - … **fast**. C++ programs are normally as fast as C programs, and sometimes even faster. They are often *far* faster than programs written in other languages.
  - … **modern**. Yes, C++ has a bunch of cruft from its early days, but if you can navigate around that, it is a very modern language with advanced abilities.

"Within C++ there is a much smaller and cleaner language struggling to get out."
-- Bjarne Stroustrup (creator of C++)

C++ is Compatible with C

- In the following ways:
  - You can (mostly) compile C programs using a C++ compiler. This is *source code compatibility*.
    - Not 100%. There are a few C features that produce errors with C++, but the errors are usually easy to fix.
  - You can call C functions from C++ if they were compiled with a compatible C compiler (e.g., g++ and gcc).
    - This allows you to use pre-compiled C libraries in C++ programs.
  - You can use C++ as a "better C."
    - That is, you can use the same methods and designs as you would for C, and use C++ features here and there to improve the quality of what would otherwise be plain C code.

Yet C++ is not C

- This is not C*:

```
map<string, int> composers{ { “Bach”, 1685 }, { “Mozart”, 1756 }, { ”Chopin”, 1810 } };

// Loop over the names of the first three composers born after 1700.
for( const auto &elem : composers
                        | v::filter( [](const auto& y) { return y.second > 1700; } )
                        | v::take(3)
                        | v::keys ) {

    // Print the names.
    cout << elem << endl;
}
```

The technique of pipelining data through multiple transformative stages is like what is done in advanced functional languages such as Haskell, Scala, OCaml, etc.

* Code adapted from *C++20 The Complete Guid*e (version 2022-10-30) by Nicolai M. Josuttis, page 123

Modern C++

- Using C++ as a better C is fine…
  - … especially if you are migrating an existing C program to C++
- … but it leaves a lot of what C++ has to offer on the table!
  - Modern C++ allows you to write programs *almost* as concisely as can be done in scripting languages.
- Also, modern C++ is much, *much* safer than many people assume.
  - Can automatically reclaim memory and other resources.
  - Can use "smart" pointers.
  - Has tools that help with thread safety and exception safety.
- Some of the haters think C++ is just C warmed over. *It isn't!*

Standard C++

- C++ is standardized by the International Organization for Standardization (ISO)
  - It is *not* defined by a single reference implementation.
  - It is *not* defined by a proprietary standard.
  - Anyone can use the name "C++" without paying fees or getting sued.
- The standard dictates what is and is not correct C++
  - If your compiler disagrees with the standard, your compiler is wrong *by definition.*
  - If you write programs that follow the standard they should work on all *conforming implementations of the language* (i.e., compilers).

Standard C++: History

- C++ 1998 (aka “C++ 98”)
  - The original version, now very old, but still a rich and powerful language.
- C++ 2011
  - A major update with many new features
- C++ 2014
  - A minor update
- C++ 2017
  - A minor update
- C++ 2020
  - A very significant update with several important new features
- C++ 2023
  - A minor update that smooths off some of the C++ 2020 features

Which Version?

- When working with C++ it is probably best to choose a version
  - Then don’t use features from newer versions!
  - This allows your code to work with older compilers (if desired). For example, to support a legacy compiler, you might have to program against C++ 98 (worst case), or C++ 2011, and not have access to the newer features.
- <u>We will use C++ 2020 in this class</u>
  - It is modern and contains some important new features
  - It is old enough to have reasonable compiler support
  - C++ 2023 is too new; compiler support is spotty
  - `g++ -std=c++20 myprog.cpp`

Compilers? The Big Three

- There are three major, independent C++ compilers
  - Microsoft Visual C++ (Windows)
    - Closed source
  - G++ (Unix-like)
    - Open source
  - Clang++ (Unix-like, especially macOS)
    - Open source, sponsored by Apple (among others) as part of the LLVM effort, but available widely
- All three are…
  - Modern: follow the latest standards.
  - Advanced: many sophisticated compiler features.

Other Compilers?

- There are many other C++ compilers but…
  - Some are re-packagings of one of the Big Three and not independent.
  - Some are very old are not maintained or only “slightly” maintained (e.g., Open Watcom C++)
  - Intel C++
    - Uses LLVM technology, but I don’t think it uses the Clang front-end (unsure).
    - Does not support Apple’s ARM processors (no surprise).
  - IBM C++
    - Also uses LLVM technology, but again without the Clang front-end (I think).

Our Compiler

- Officially for this course we will use g++ on Lemuria (Ubuntu 22.04).
  - Standardizing allows you to be sure I will see the same effects you see.
    - If your program compiles and runs, it will for me also.
    - If you have a problem with your program, I will see the same problem when I help you.
  - We all have access to this compiler, and its environment is the same for all.
- But…
  - I'm fine with you using something different most of the time. In fact, I encourage it.
  - Just be sure to check your programs using g++ on Lemuria before you submit!
  - That's the compiler I will use to test your programs.

Recommended Development Environments

- I will support (to various degrees) the following:
  - SSH access to Lemuria with a text editor (Nano, Emacs, Vim) and g++.
  - Visual Studio Code with g++ on Linux, macOS, and Windows (via Cygwin).
  - Eclipse with the CDT with g++ on Linux, macOS, and Windows (via Cygwin).
  - CLion from JetBrains with g++ on Linux, macOS, and Windows (via Cygwin).
  - Visual Studio. Windows only. Very powerful. No Makefiles.
  - Code::Blocks with g++ on Linux or Windows (macOS support is very old).
    - The current official release of Code::Blocks (20.03) is somewhat old and has limited C++ 2020 support. That may affect some assignments.
    - I'm not sure if Code::Blocks has Makefile support; but I think it does.
  - XCode? macOS only. No Makefiles?

Why Does It matter?

- If C++ is standardized, shouldn’t all compilers be the same?
  - Yes, but only if you write programs that: ***make no use of platform-specific libraries or compiler extensions, and don’t engage in any implementation-defined, unspecified, or undefined behavior.***
  - In general, it is difficult to do this, so it is common for a program to work in one environment and not in another
  - Also, compilers have bugs and limitations, so even programs that should work across all compilers might not
- Let’s examine each of the conditions mentioned above…

Platform-Specific Libraries

- The *Least Common Denominator* effect:
  - The standard defines facilities that make sense on **all** platforms
  - Most programs want to use specialized facilities available for their platform
- For example…
  - The header <windows.h> declares Windows API functions. These functions are not available on Unix-like systems.
  - OTOH the header <unistd.h> declares Unix API functions that are not available on Windows.
- *Quick!* Is the header <fcntl.h> a platform-specific header?
  - Now you are seeing the problem.

Compiler Extensions 1

- gcc accepts the following program:

```c
int main( )
{
    // Nested function definition is not standard!
    int increment( int number ) {
        return number + 1;
    }
    int x = 0;

    x = increment( x );
    printf( "The answer = %d\n", x );

    return EXIT_SUCCESS;
}
```

Compiler Extensions 2

- The standard allows compilers to implement extensions…
  - … if those extensions do not interfere with standard programs!
- The C standard does not allow nested functions…
  - … but the compiler vendor can implement it without changing the meaning of a program that doesn't try to use the feature.
- However, the previous program isn't *portable*
- The --pedantic option on gcc warns about these things:

```
check.c:6:5: warning: ISO C forbids nested functions [-Wpedantic]
    6 |     int increment( int number ) {
      |     ^~~
```

Implementation Defined Behavior 1

- The standard allows compilers to do things differently.
- If something is *implementation defined* it means:
  - Each implementation (… of the language, i.e., each compiler) is allowed to make its own choices, generally within limits.
  - AND… each implementation <u>must document</u> those choices
- If your program uses something that is implementation defined…
  - … it may behave differently with a different implementation (of the language), and so it is not portable
  - BUT your program is perfectly well behaved on the implementation that has the documented behavior you need.

Implementation Defined Behavior 2

- The C++ 2020 standard states the *range of type int is at least -32768 to +32767*. The precise range is implementation defined.
- The following program is not portable:

```
int main( )
{
    // Code assumes 1'000'000 can fit into int. That is not guaranteed.
    int x = 1'000'000;
    …
    return 0;
}
```

Implementation Defined Behavior 3

- Why???
- The C and C++ standards assume target systems might exist over a huge range…
  - … from tiny, 16-bit microcontrollers with 4 KB or RAM (or less!)
  - … to huge supercomputers with 128-bit registers.
- … and over highly exotic architectures…
  - … with memory segments and segmented addresses (32-bit x86 has this)
  - … with NUMA (non-uniform memory access)
  - … with 9-bit bytes
  - … etc!

Implementation Defined Behavior 4

- By allowing implementations to define certain aspects of the language it is possible to implement the language effectively across a broad range of machines.
- Contrast: Java
  - Fixes the sizes of the primitive types
  - … this is convenient for the programmer
  - … but means that Java can't reasonably be implemented on a tiny system (16-bit microcontroller with 4 KB RAM. Ha!)
  - … and can't take full advantage of a huge system. For example, in Java you can't create an array with more than 2,147,483,647 elements because int is forced to be 32-bits.

An Aside…

- The Big Three compilers when targeting 64-bit systems are all *LP64* implementations.
  - L…64 long integers are 64-bits
  - …P64 pointers (addresses) are 64-bits
  - BUT plain integers are still 32 bits
  - short integers (‘short int’) are 16 bits
  - long long integers (‘long long int’) are also 64 bits
- Some people think C and C++ are always like this. Not so!
  - Note that the standard requires ‘long long int’ to always be (at least) 64 bits on all implementations, even on a tiny microcontroller.

Demonstration

- See the program types.cpp
  - When compiled and run, it displays information about the range of all the primitive types used by your particular compiler.
  - If you compile and run it on a different compiler, you might get different results…
  - … but for any particular implementation, the results will be consistent and can be relied upon.

Unspecified Behavior 1

- Certain aspects of the language are left unspecified by the standard…
  - … but are things correct programs still do.
- If your program's output depends on unspecified behavior…
  - … your program is not portable
  - … it may even not do the same thing each time you run it!
  - Computer science jargon: your program is *nondeterministic*.
  - Compiler vendors are *not* required to document what their compiler does.

Unspecified Behavior 2

- Classic example: *the order of evaluation of function arguments is unspecified*

```
do_something( x + y, z - w );
```

- Which is computed first: `x + y` or `z - w`?
  - It could go either way
  - The compiler doesn't have to document what it does.
  - It could be different in each place where `do_something` is called.
  - It could be different each time the program runs (although that is very rare).
- In this example it doesn't matter. <u>That's what you want</u>!

Unspecified Behavior 3

- Now consider this example:

```
do_something( f( ), g( ) );
```

- Suppose `f( )` outputs “Hello” and `g( )` outputs “World”
  - Does this output “HelloWorld” or “WorldHello”?
  - You don’t know what you’ll get!
- If it happens to do what you want, you might think your program is fine.
  - Actually, your program is non-portable and unreliable.

Unspecified Behavior 4

- Why??
  - By allowing compiler vendors the freedom to do certain things however they see fit, and not documenting it…
  - ... the optimizer can be more aggressive.
- There might be advantages with using different evaluation orders:
  - Better register usage
  - Fewer register/memory moves
  - Reuse of previously computed values (which are effectively computed first)
- Sometimes a program will work in debug mode but fail in release.
  - Often that's because the program is engaging in unspecified behavior.

Undefined Behavior 1

- *"Behavior for which this document imposes no requirements."*
  - Section 3.57 of ISO/IEC 14882:2020(E), "Programming Languages – C++"

Anything Can Happen!

- Usually, the program crashes.
- But…
  - ... the program might, say, reformat your hard drive.
  - … terminate with an easy-to-understand error message.
  - … fail to compile.
  - … work normally and do something sensible (i.e., some kind of extension).

Undefined Behavior 2

- Classic Example: Accessing an array out of bounds.

```
int array[128];

array[128] = 0;    // Undefined behavior!
```

- The effect is usually a program crash, but it might…
  - … change the value of an unrelated variable
  - … cause the program to execute unrelated code when the function returns
  - … or do pretty much anything else!
- Undefined behavior is often called ***UB*** on internet forums.

Undefined Behavior 3

- Other examples of undefined behavior:
  - *Reading* or writing out of bounds of an array.
  - Integer overflow.
    - Most compilers just let integers wrap-around, but technically it is UB.
    - A compiler could check for integer overflow and throw an exception as an extension.
  - Dereferencing a null pointer for reading or writing.
  - Computing the difference between two pointers that point into different arrays.
  - Using a bit shift distance that is greater than the number of bits in the value being shifted.
  - Many, many other things.

Undefined Behavior 4

- Why??
  - Having undefined behavior gives the compiler a lot of room to maneuver
    - More aggressive optimization is possible.
    - Space for experimental features while still conforming to the standard.
      - For example, there was an experimental version of `gcc` that *did* check array bounds!
    - Space for future standards to define currently undefined things without breaking backwards compatibility.
    - Avoid forcing compilers to implement hard things.
      - That experimental version of `gcc`? It's complicated and causes programs to have high space/time overheads.
  - *Programmers want everything well-defined and well-specified.*
  - *Compiler vendors want everything undefined and unspecified!*

I Want Portability

- If you want to write a program that will work everywhere:
  - Do not use any platform-specific features.
  - Do not use any compiler extensions.
  - Do not rely on any implementation defined aspects of the language.
  - Do not depend on any unspecified behavior.
  - And for God’s sake, <u>don’t do anything undefined!</u>
- Oh, and…
  - Only use compilers that fully implement the standard and are bug-free.

Good luck with that!

Okay, It’s Not That Bad

- To enhance portability:
  - Well-written programs use a style that tends to avoid undefined and unspecified behaviors.
  - Many implementations on the same or similar platforms make the same implementation-defined choices.
    - For example, LP64 compilers for 64-bit targets.
  - Programs with simple I/O requirements can often use the standard I/O library.
  - Compiler bugs are relatively rare, especially on mundane code.
  - Using an older standard increases the likelihood of a full implementation.
  - Non-portable code can be partitioned into its own module.
    - Thus, only a well-defined section of the program will need adjusting.

About Me

- I first learned C++ in the late 1980s.
- I participated on X3J16 in the early to mid 1990s.
  - X3J16 was the ANSI committee charged with standardizing C++.
  - IOW, I worked on the C++ 98 standard.
- I wrote some C++ programs as a consultant (1990s)
- Taught C++ at Vermont Technical College to the CPE and CSE students.
  - Our Algorithms and Data Structures course was once in in C++.
- Contributed to the Open Watcom project…

Open Watcom

- Open Watcom has a special place in my heart
  - We used the commercial Watcom compiler at VTC in the 1990s.
  - Compiler was taken off the market (couldn't compete against Microsoft).
  - Released as open source around 2000.
- I contributed to Open Watcom starting in the early 2000s
  - I worked on updating the compiler to C++ 98 (the latest standard at the time).
  - I wrote about half of the C++ standard template library for Open Watcom.
  - I served as the project maintainer for a year or so.

And Now

- I took a hiatus from C++ for a while.
  - I went back to school (2004) to get my PhD in computer science where I used Scala extensively in my dissertation research (2004-2013).
  - I got involved the VTC's CubeSat project which uses Ada and SPARK to write ultra-reliable flight software for spacecraft (2008-2022).
- More recently…
  - … I started teaching this elective (CIS-3012) using C++ 2011.
  - This year I'm upgrading to C++ 2020. Spent time last summer studying the new features and updating my notes, examples, and slides.

Enjoy!