Lecture 10: Intro to C++

October 30th, 2018

Topics Introduced: Introduction to C++: variables, strings, static arrays, and looping.

1 History

"C with Classes" Bjarne Stroustrup first started writing "C with Classes" in 1979. Bjarne was inspired to create a new language after writing his PhD thesis using Simula.[1] He felt that Simula had features that were tremendously helpful for writing complex software, but that it lacked the raw speed of a language like C which made in impracticable. "C with Classes" added features to the C compiler such as *classes*, *strong typing*, and *default arguments*.[2]

C++: A Non-Trivial Incremental Improvement over "C with Classes" In C, there is an increment operator, `++`. In the early 80's "C with Classes" was renamed to C++ alongside the introduction of new features like *operator overloading*, *references*, and *type-safe free-store memory allocation*. These developments took place at AT&T Bell Labs, where languages like B and C were also developed.

Early Updates to the Language The first edition of The C++ Programming language was released in 1985, and this became the definitive reference for the language. In 1989, C++ 2.0 was released, which featured multiple inheritance, abstract classes, const member functions. Later, features like templates, exceptions, additional casts, and a boolean data type were added. Beyond that, the language evolved very slowly until the release of C++11, which added significant new functionality, including expanding the standard library.

Major-Minor Revision System The language is now on a major-minor revision system, wherein C++14 introduced minor updates, and C++17 introduced more notable features. CME211 will primarily introduce C++11; we'll discuss aspects of C++17 in CME212.

[1] Simula (for simulations) was written in the 60's in Oslo, Norway by Dahl and Nygaard. It's considered *the* first object-oriented programming language, and featured classes, inheritance, and garbage collection.

[2] C is statically typed, but weakly enforced: i.e. even though all data has a type implicit conversions may be performed. This programmer is prone to unintended conversions and corresponding unexpected results.

1.1 Philosophy

Since inception, the development of C++ has been guided by the following pragmatic tenants.

- New features shall be immediately applicable to real world programs.
- No implicit violations of the type system (allows explicit conversions by the programmer).
- User created types shall have the same support and performance as built-in types.
- There should be no language beneath C++ except Assembly language.

C++ is one of the most ubiquitous programming languages, behind Java and C.

Other Languages Since C++ There is also an A+ language, developed in the 1980s. Microsoft released implementation of C# language in 2000 in order to provide another alternative to C++. D came out in 2001 and is yet another atempt at "re-engineering" of C++. None of these are tremendously compelling over C++, which has strict language standards with multiple companies and organizations implementing said standards.

Further Comparison Against C C is *almost* a subset of C++. It's certainly a lower level language that has fewer abstractions over the hardware. C is still used for many applications: Linux kernel, CPython interpreter, low power or embedded systems, etc. Although C has fewer abstractions when compared with C++, this doesn't make it "faster"; it's still worth using today for certain applications which don't require the abstractions afforded by C++.

2 Hello World of the C++ Language

In this lecture, we are going to start with a C++ source file and modify it to show various things about C++. If you desire to compile and execute subsequent code block on your own, please modify the provided source (or start a new source file): `src/hello.cpp`.

```
1 #include <iostream>
2
3 int main() {
4   /* Hello world program (this is a comment)...
5      ... and this form of comment can span multiple lines. */
6   // This is also a comment, but only goes to the end of the line.
7   std::cout << "Hello world" << std::endl;
8   return 0; // Return value of the function, where a non-zero indicates error.
9 }
```

`#include` Statements These are analogous to Python `import` statements: it's how functionality from other source files are included into other programs. The <`iostream`> library has standard C++ input output functionality.

`main` Program Each program is required to have exactly one `main` function, and it is the *entry-point* for the program. I.e. code from the `main` function body will be executed upon starting the program in a procedural fashion.

`return` Statements The `return` statement terminates the current function and returns the resulting argument (if specified). The type of the return value must match the specified output type of the function (`int` in this case).

Comments There are two forms of comments in C++. Text between `/*` and `*/` signifies a comment which *can* span multiple lines, whereas a `//` specifies an *inline* comment.

2.1 Compilation

C++ programs have to be compiled from human readable source code into assembly instructions and ultimately machine code, which is ultimately used when the program actually executes. The assembler is responsible for low-level operations such as moving data from one location into another (from `RAM` into a `register`) or performing a simple logical or arithmetic instruction (e.g. bitwise `and`, or addition `+`).

Portability Since the assembler is tied so closely with the *actual* machine architecture, it shouldn't be surprising to learn that athough C++ source code can be made portable (i.e. capable of compilation across multiple machines or operating systems such as Linux, Windows, or Mac) the executables are specific to an operating system *and* underlying processor.

Compiling on Rice We use GNU compilers, available with most any Linux distribution.

```
$ ls
hello.cpp
$ g++ -std=c++11 -Wall -Wconversion -Wextra hello.cpp  -o hello
$ ls
hello  hello.cpp
```

What's going on here? `g++` is the actual GNU C++ compiler program. The option `-std=c++11` informs the compiler to use the C++11 language standard. Note that not all compilers support all language features. The flags `-Wall`, `-Wconversion`, `-Wextra` enable all warnings. The penultimate step specifies `hello.cpp` as the name of the C++ source file to compile. Lastly, we specify the name of the output file to take on a name similar to our source code file; the default is `a.out`.[3]

[3] Why `a.out`? It signifies that the resulting file is *Assembler Output*.

2.2 Execution

When we're in the same working directory as our executable generated from the step above:

```
$ ./hello
Hello world
```

Why `./` Is Required Turns out that the `./` is required. The shell accepts *commands*. When we specify that we want to run a program, the first thing the shell does is look along the `system path` variable in order to find an executable matching the command specified. In general, to execute a program that is not on the system path, we must specify its path completely. The `.` specifies the *current working directory*, and the `/` is used as a path delimiter on linux systems. Therefore, specifying `./` before a command specifies to the shell that it should look in the current working directory in order to run the appropriate executable.

3 Language Basics and Overview

3.1 Streams

Standard C++ uses "streams" for both input and output. A stream in computing is a sequence of data elements made available over time. The pedagogical mental model is a sequence of items arriving on a conveyor belt.

Output Streams (`operator<<`) If we want to print data to console (or write data to a file), we must use an *output* stream. The `operator<<` is a binary infix operator which accepts as argument an *output stream* and an object to insert into the output stream. See stream insertion operator.

```
std::cout << "Hello world" << std::endl;
```

The object `cout` is in the `std` namespace and refers to the standard output (stdout) stream. Note that because `operator<<` is a binary infix operator, this means the above expression is evaluated as `(std::cout << "Hello world") << std::endl;`. This also lets us realize that `(std::cout << "Hello world")` *must* return an output stream object since the result fed into another `operator<<`. In contrast, the `std::endl` is simply a character denoting a newline, and flushes the output buffer to console (or file).

You can also put in a newline yourself, and let the buffer flush automatically as necessary.

```
1 #include <iostream>
2 int main() { std::cout << "Hello world\n"; }
```

```
$ g++ -std=c++11 -Wall -Wconversion -Wextra hello.cpp -o newline
$ ./newline
Hello world
```

Input Streams (`operator>>`) If we want to read in data from console (or from a file), we must use an *input* stream, which is specified by the `operator>>` operator.

3.2 Namespaces, Libraries and Scope (Resolution)

When we do `#include` that is somewhat analogous to an import in Python, giving us access to functionality defined in another file. In C++ the access to even fundamental functionality like outputting to the screen requires specifying the proper include file(s). Include files in C++ work a bit differently when it comes to namespaces. However, namespaces in C++ still generally serve the same purpose as namespaces in Python.

3.2.1 Namespaces

In Python the name of the namespace comes from the file name, and everything in the file is automatically in that one namespace. A C++ include file might contain functions, classes, etc. that are not in a namespace at all. An include file could also contain functions, classes, etc. from multiple namespaces. Namespaces can also span multiple include files.

Example: Namespace for the C++ Standard Library The C++ Standard Library is all the built in functionality that is part of the C++ language. The namespace for this library is `std`. `iostream` contains `cout` in the `std` namespace. By default, when using `cout`, we need to specify the namespace and fully qualify the symbol as `std::cout`.

Scope Resolution Operator Perhaps unsurprisingly, the `::` is called the scope resolution operator. It's used to indicate what namespace something comes from. If a namespace is required that will typically be listed in the documentation, or by inspecting the include file. Will talk about namespaces more when we start writing our own include files.

3.2.2 Common Mistakes

Forgetting to `#include<iostream>`:

```
1 int main() { std::cout << "Hello world" << std::endl; }
```

```
hello.cpp: In function ‘int main()’:
hello.cpp:1:14: error: ‘cout’ is not a member of ‘std’
 int main() { std::cout << "Hello world" << std::endl; }
              ^
hello.cpp:1:44: error: ‘endl’ is not a member of ‘std’
 int main() { std::cout << "Hello world" << std::endl; }
                                            ^
```

Note the helpful error messages we are receiving. The syntax `hello.cpp:1:14` localizes the origin of the first error to `hello.cpp`, on line `1` and starting at character position `14`.

Another mistake is forgetting the `std` namespace:

```
1 #include <iostream>
2 int main() { cout << "Hello world" << endl; }
```

```
$ g++ -std=c++11 -Wall -Wconversion -Wextra hello.cpp -o hello
hello.cpp: In function ‘int main()’:
hello.cpp:2:14: error: ‘cout’ was not declared in this scope
 int main() { cout << "Hello world" << endl; }
              ^
hello.cpp:2:14: note: suggested alternative:
In file included from hello.cpp:1:0:
/usr/include/c++/5/iostream:61:18: note:   ‘std::cout’
   extern ostream cout;  /// Linked to standard output
                  ^      /// ... Terminal output truncated. Similar error for 'endl' is reported.
```

Here, the compiler recognizes that `cout` is an object not found within the current scope. It subsequently suggests we consider the object declared within the `iostream` library at line 61 and starting whose identifier appears at character position `18`.

Other Idioms for Namespaces Analagous to a wildcard import in Python:

```
1 #include <iostream>
2 using namespace std;   // Not considered good practice.
3
4 int main() {
5   cout << "Hello world" << endl;
6   return 0;
7 }
```

```
$ g++ -std=c++11 -Wall -Wconversion -Wextra hello.cpp -o hello
$ ./hello
```

Better practice is to be specific about our imports, akin to `from numpy import dot`.

```
1 #include <iostream>
2 using std::cout;    // Good practice...
3 using std::endl;    // ... except when writing a header file!
4
5 int main() {
6   cout << "Hello world" << endl;
7   return 0;
8 }
```

3.3 Blocks of Code and Scope

Blocks of code (e.g. code comprising a function, conditional, loop, etc.) are indicated by enclosing them in curly brackets.

```
1 #include <iostream>
2 int main(){std::cout<<"Hello world"<<std::endl;return 0;}
```

```
$ g++ -std=c++11 -Wall -Wconversion -Wextra hello.cpp -o hello
$ ./hello5       # Still prints "Hello world" on the next line.
```

There are very few places where whitespace matters to the compiler.

Bracket Style Like using tabs vs. spaces, or Emacs vs. Vim, this one gets personal!

```
1 #include <iostream>
2
3 int main() {
4   std::cout << "Hello world" << std::endl;
5 }
```

Or! Equally valid, but requires an additional "line" of code. Benefit is that the braces *always* line up, whence we can check the limits of scope easily. In contrast, the former method of using braces means that we much match a closing } brace with either a type-specifier (in the case of a function or namespace definition) or a control-flow structure.

```
1 #include <iostream>
2
3 int main()
4 {
5   std::cout << "Hello world" << std::endl;
6 }
```

3.3.1 Scope

A variable declared within a block is only accessible from within that block. Blocks are denoted by curly brackets, typically the same brackets that denote a function, loop or conditional body, etc. Sub-blocks can declare different variables that have the same name as variables at broader scope. Variables should not be declared with excessive scope.

We can't access variables from *narrower* scopes Example:

```
1 #include <iostream>
2
3 int main() {
4   { int n = 5; }     # Narrow most scope. Not accessible from rest of 'main()'.
5   std::cout << "n = " << n << std::endl;
6 }
```

If we compile with warning flags, we're instructed that our variable `n` in fact goes unused. Further, regardless of warning flags used we realize an error on line 8 when we attempt to reference a variable `a` which is not available in scope.

```
$ g++ -Wall -Wconversion -Wextra scope.cpp -o scope
scope.cpp: In function 'int main()':
scope.cpp:5:9: warning: unused variable 'n' [-Wunused-variable]
     int n = 5;
         ^
scope.cpp:8:26: error: 'n' was not declared in this scope
   std::cout << "n = " << n << std::endl;
   ^
```

We *can* access variables from *wider* scopes Note that if we try to re-declare a variable in the same scope, we would realize an error. It's possible to access variables from a parent scope. However, it's also possible to declare new variables in a (narrower) scope with the same identifiers. E.g.

```
#include <iostream>
#include <string>

int main() {
  std::string n = "Hi";
  std::cout << "n = " << n << std::endl;
  {
    int n = 5;                                  // New scope, doesn't yet contain a variable 'n'.
    { std::cout << "n = " << n << std::endl; }  // We *can* access variables in parent scope.
  }
}
```

```
$ g++ -Wall -Wconversion -Wextra scope.cpp -o scope
$ ./scope
n = Hi
n = 5
```

3.4 Return value from `main()`

If missing (as in above examples), the return value for `main` is inferred to be identically zero. Unix systems adhere to the POSIX Standard for exit processes, wherein programs shall return `0` under normal conditions and non-zero upon error.

```
#include <iostream>

int main() {
  std::cout << "Hello world" << std::endl;
  return 7;
}
```

Recall that in a shell, we use `$` to perform parameter expansion, i.e. we can think of it as forcing evaluation on its argument. Separately, the special parameter `?` holds the return value for the last command executed.

```
$ g++ -std=c++11 -Wall -Wconversion -Wextra hello.cpp -o hello
$ ./hello
Hello world
$ echo $?
7
$ ls
a.out  hello  hello.cpp
$ echo $?
0
```

3.5 Variables

3.5.1 Implications of a Typed Language

C++ is *typed*, we must specify the type of each object at the time of creation. See `src/variables.cpp`.

```
#include <iostream>

int main() {
  a = 2;                // Not at all valid...
  b = 3;                // ... we must specify data types!
  c = a + b;            // None of these expressions are valid C++ code.
}
```

Output:

```
$ g++ -std=c++11 -Wall -Wconversion -Wextra variables.cpp -o variables
variables.cpp: In function ‘int main()’:
variables.cpp:4:3: error: ‘a’ was not declared in this scope
   a = 2;
   ^
variables.cpp:5:3: error: ‘b’ was not declared in this scope
   b = 3;
   ^
variables.cpp:6:3: error: ‘c’ was not declared in this scope
   c = a + b;
   ^
```

3.5.2 Variable Declaration

Declaring an object to be of a certain type is much different from initializing the data to a particular value.

```
#include <iostream>

int main() {
  int a;
  int b, c;
  c = a + b;
  std::cout << "c = " << c << std::endl;
}
```

When we declare an object as a certain type in the above example, we're allocating storage for the variable somewhere in memory. But we haven't specified a value for initialization, whence we simply reuse whatever bits were "leftover" at that particular address in memory from whatever program or part of the operating system used it last.

```
$ g++ -std=c++11 variables.cpp -o variables
$ ./variables
c = 32767
```

If the variable we're declaring is atomic (i.e. a real value or a character) then default initialization results in nothing being done, i.e. the value is indeterminate.

3.5.3 Compiler Warnings

Suppose we add an expression to our script which evaluates `a + b`. If we use the right compiler flags, we can trigger an informative warning.

```
$ g++ -std=c++11 -Wall -Wconversion -Wextra variables.cpp -o variables
variables.cpp: In function ‘int main()’:
variables.cpp:9:12: warning: ‘a’ is used uninitialized in this function [-Wuninitialized]
   c = a + b;
           ^
variables.cpp:9:12: warning: ‘b’ is used uninitialized in this function [-Wuninitialized]
```

The mantra is: “enable them, read them, and fix them.” We will not have any sympathy if you have bugs that would have been caught by enabling warnings. Read: you will lose points if compilation of your program generates warnings.

3.5.4 Initializing Variables

This can be done in the same step as declaration.

```
1  #include <iostream>
2
3  int main() {
4    int a = 2;          // Declare and initialize.
5    int b;              // Declare and leave uninitialized.
6
7    b = 3;              // Assign a value.
8    int c = a + b;      // Result is well defined to be 5.
9    std::cout << "c = " << c << std::endl;
10 }
```

```
$ g++ -std=c++11 -Wall -Wconversion -Wextra variables.cpp -o variables
$ ./variables
c = 5
```

4 Revisiting Type System of C++

C++ is *strongly* typed, whence assigning a string to an integral data type yields a type-error.

```
1  #include <iostream>
2
3  int main() {
4    int a;
5    a = "hello";
6  }
```

```
$ g++ -std=c++11 -Wall -Wconversion -Wextra variables.cpp -o variables
variables.cpp: In function ‘int main()’:
variables.cpp:10:5: error: invalid conversion from ‘const char*’ to ‘int’ [-fpermissive]
   a = "hello";
     ^
```

4.1 C++ *Allows* (Certain) Implicit Conversions

Truncation Even though C++ is strongly typed and prohibits eggregious implicit conversions from disparately different data types, it still allows for implicit conversions between "like" types.[4] This could catch the novice programmer by surprise.

```
#include <iostream>

int main() {
  int a, b;   # Declare 'a' and 'b' to both be integers.
  a = 2.7;    # Oops! C++ will implicitly truncate float to convert to int type on LHS of '='.
  b = 3;
  int c = a + b;  # Adds to 5, since 'a' holds the integer value 2.

  std::cout << "c = " << c << std::endl;
}
```

```
$ g++ -std=c++11 -Wall -Wconversion -Wextra variables.cpp -o variables
variables.cpp: In function 'int main()':
variables.cpp:6:5: warning: conversion to 'int' alters 'double' constant value [-Wfloat-con
   a = 2.7;
     ^
```

We could perhaps more explicitly describe what's happening in the assignment to variable `a` above using the following valid sytanx, which relies on an explicit type cast.

```
  a = (int)2.7; // int(2.7) would also work
```

If the above definition of `a` is used, we no longer realize a warning message, since we explicitly told the compiler what our intentions were. The output is still the same.

```
$ g++ -std=c++11 -Wall -Wconversion -Wextra variables.cpp -o variables
$ ./variables
c = 5
```

Numeric Promotion Implicit conversions can also work to our advantage, for example, if an integer is added to a floating point value, the result is (perhaps as one might expect) a floating point value; formally the language specifies rules on numeric promotion.

```
#include<iostream>
int main() {
  int a = 2;
  float b = 3.14;
  float c = a + b;
}
```

Here, the `operator+` applies implicit conversion to get a common real type between the `int` and `float`. The coercion happens within the stack frame belonging to `operator+` and does not affect the state of variable `a` in the scope of `main`.

[4] This can be nuanced. For example, a static array can be cast to a pointer.

An Example with Double Precision Floating Point We take a moment to showcase how we can define a double precision floating point variable, using keyword `double`. Note that when `operator*` acts on `a`, its datatype is implicitly coerced (widened) to a double precision floating point before the multiply is actually executed.

```
#include <iostream>

int main() {
  int    a = 2;
  double b = 3.14;
  double c = a*b;   # In this expression, 'a' implicitly cast to double.
  std::cout << "c = " << c << std::endl;
}
```

```
$ g++ -std=c++11 -Wall -Wconversion -Wextra variables.cpp -o variables
$ ./variables6
c = 6.28
```

Rounding As an aside, we mention that the `round` function is implemented in the `cmath` header, but is *not* part of the standard (`std`) namespace.

```
#include <iostream>
#include <cmath>

int main() {
  double c = 2.7;
  int a = (int)round(c);  // Note: round() is *not* in std namespace.
}
```

The resulting value of variable `a` above is the integer value three, since `round` obeys the expected convention. Note that we still need the explicit cast to prevent a warning, since there is a difference between a floating point value with zero fractional component and the corresponding integer in so far as their internal data representation is concerned.

4.2 Key Data Types

Basic C++ has all of the data types that we talked about when we looked at computer representation of data in conjunction with `numpy`. This includes various (un)signed integers, floating points (single, double, and extended precision), Booleans, and strings. C++ Types.

4.2.1 Boolean

```
#include <iostream>

int main() {
  bool equal = 2 == 3;
  std::cout << equal << std::endl;
  std::cout << bool(true) << std::endl;  // Note: use of all lowercase.
}
```

Here, we'll see that when passed to `operator<<`, our Boolean is implicitly cast to `int`.

```
$ g++ -std=c++11 -Wall -Wconversion -Wextra boolean.cpp -o boolean
$ ./boolean
0
1
```

4.2.2 Strings

There are two options for strings in C++ An array of null-terminated characters (C-style). Or! A string class, which is much safer and easier to use, complete with built-in methods.

```
#include <iostream>
#include <string>

int main() {
  std::string hello = "Hello world";
  std::cout << hello << std::endl;    # Compiles without warning. Prints "Hello world" when executed.
}
```

String Concatenation If we use a `string` class object, the `operator+` is overloaded to signify concatenation of strings. The next example prints `Hello world` to console, with no warnings encountered during compilation (with warning flags enabled).[5]

```
#include <iostream>
#include <string>

int main() {
  std::string hello = "Hello";
  std::string world = " world";
  std::cout << (hello + world) << std::endl;
}
```

String Finding Unsurprisingly, there is a `find` method within the string class.

```
#include <iostream>
#include <string>

int main() {
  std::string hello = "Hello";
  std::string lo = "lo";
  std::cout << hello.find(lo) << std::endl;
}
```

It simply prints the (zero-indexed) location where the match starts (if one is found), else a sentinel value `npos` is used.

```
$ g++ -std=c++11 -Wall -Wconversion -Wextra string.cpp -o string
$ ./string
3
```

[5] If we try to do this with a C-style character-array, we would yield an error since there will be no overloaded version of `operator+` which is designed to accept these two particular lengths of character sequences.

5 Static Arrays

Static means known at compile time. All objects must be typed. So what does it mean to declare a statically allocated array? It means that we specify not only the type but also the size (length) in a way that can be determined at compile time; `src/array.cpp`.

```
#include <iostream>

int main() {
  int a[3]; // Array referenced via a with 3 integer elements

  a[0] = 0;
  a[1] = a[0] + 1;
  a[2] = a[1] + 1;

  std::cout << "a[0] = " << a[0] << std::endl;
  std::cout << "a[1] = " << a[1] << std::endl;
  std::cout << "a[2] = " << a[2] << std::endl;

  return 0;
}
```

The integer literal three appears in our source code, and the compiler can see this symbol and understand that we are requesting an array to store exactly three integers. Compiling:

```
$ g++ -Wall -Wconversion -Wextra array.cpp -o array
$ ./array
a[0] = 0
a[1] = 1
a[2] = 2
```

5.1 Out of Bounds Access $\rightsquigarrow$ Undefined Behavior

Accessing static arrays (or any array for that matter) out of bounds leads to undefined behavior and is a particularly nasty problem. Modify `src/array.cpp` to the following:

```
#include <iostream>

int main() {
  int a[3]; // Array has 3 elements


  a[0] = 0;
  a[1] = a[0] + 1;
  a[2] = a[1] + 1;
  a[3] = a[2] + 1; // Out of bounds access


  std::cout << "a[0] = " << a[0] << std::endl;
  std::cout << "a[1] = " << a[1] << std::endl;
  std::cout << "a[2] = " << a[2] << std::endl;
  std::cout << "a[3] = " << a[3] << std::endl;

  return 0;
}
```

Now, compile and run:

```
$ g++ -Wall -Wconversion -Wextra array.cpp -o array
$ ./array
a[0] = 0
a[1] = 1
a[2] = 2
a[3] = 3
```

Undefined Behavior Nothing "bad" happened here. But, the behavior is undefined to persist if we re-run our program, let alone attempt to compile and run on another machine. From the reference manual:

> Compilers are not required to diagnose undefined behavior... and the compiled program is not required to do anything meaningful.

If the out-of-bounds access is eggregious enough to trip the Operating System to the fact that the executable is accessing memory that is outside the bounds of what the program was alloted, we can encounter a segfault.[6] If we are so unlucky that the out-of-bounds access is *not* eggregious enough to trip a segfault, the memory simply gets silently corrupted; it could be the case that the bits which are referenced by `a[3]` were actually being used by another variable within our program, and in assigning to `a[3]` its possible we corrupted these data.

5.2 Address Sanitizer

We can instruct the executable to detect out of bound memory access in static arrays. To do this we enable the "address sanitizer" at compile time. This functionality is so essential it has been incorporated into GNU (and other) compilers. It adds much needed instrumentation around memory accesses We remark that the compiled program will use more memory and run slower!

Let's enable this with `g++`:

```
$ g++ -Wall -Wconversion -Wextra -g -fsanitize=address array.cpp -o array
```

Here, the `-g` flag adds debugging symbols to the output executable. This is required for `address sanitizer`. The `-fsanitize=address` enables the address sanitizer itself.

[6] It's *possible* that we could have overwritten the bits that were holding the state of e.g. our disk-drive (causing a DVD to be ejected), or OS volume (causing the volume to change), and without hardware protection in place to trigger a segfault and notify the operating system, undesired behaviors could be realized.

5.3 Testing Address Sanitizer

The output can be verbose. We omit some of the output for clarity.

```
=================================================================
==22830==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7ffebe31bd3c at \
    pc 0x000000400dde bp 0x7ffebe31bd00 sp 0x7ffebe31bcf0 \
    WRITE of size 4 at 0x7ffebe31bd3c thread T0
    #0 0x400ddd in main ~/cme211-2018/notes/lecture-10/src/array.cpp:12
    #1 0x7eff93d8f82f in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x2082f)
    #2 0x400c28 in _start \
       (~/cme211-2018/notes/lecture-10/src/array+0x400c28)

Address 0x7ffebe31bd3c is located in stack of thread T0 at offset 44 in frame
    #0 0x400d05 in main ~/cme211-2018/notes/lecture-10/src/array.cpp:3

  This frame has 1 object(s):
    [32, 44) 'a' <== Memory access at offset 44 overflows this variable

SUMMARY: AddressSanitizer: stack-buffer-overflow ~/cme211/notes/lecture-10/src/array.cpp:12 main
Shadow bytes around the buggy address:
  0x100057c5b780: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100057c5b790: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x100057c5b7a0: 00 00 f1 f1 f1 f1 00[04]f4 f4 f3 f3 f3 00 00
  0x100057c5b7b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100057c5b7c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  ... Output ommitted ...
  ASan internal:         fe
==22830==ABORTING
```

Interpretation What's important from the above? We are told that we have a `WRITE` of size 4 bytes (the size of an `int` on the platform/compiler I was using) at hexadecimal memory address given by `0x7ffebe31bd3c`. Specifically, the line starting with `#0` indicates that the invalid write occurs on line 12 of `array.cpp`. We then told that variable `a`, which was declared on line `3` of our program, occupies the contiguous sequence of bits that start at a 32-byte offset relative from the main stack frame and go up through (but not including) the 44th byte offset. Since each integer element of the array requires `4` bytes, we see that we in fact have storage for `(44 - 32) / 4 = 3` elements. We are told that we tried to access the memory just beyond the bounds, i.e. that "Memory access at offset 44 overflows this variable". The memory neighborhood that was corrupted is displayed and marked by a `=>`.

5.4 GDB and Address Sanitizer

We can use the GNU debugger `gdb` to get more precise information about the error. If you took the effort to learn how to use the Python debugger `pdb`, then the basic comands available to you in `gdb` will be familiar. This tool is totally separate from `address sanitizer`, but they also play nicely together. For example, we can use `address sanitizer` to enable the detection of out of bounds memory access, treating them like errors. Then, using `gdb`, we can request a `backtrace` of the call stack to see how we arrived at our error.

```
$ export ASAN_OPTIONS=abort_on_error=1
```

```
$ gdb ./array
...
(Gdb) run
Starting program:
~/cme211-notes/lecture-10/src/array

... [lots of output, same as what we showed first time with address sanitizer above] ...

(gdb) backtrace
#0  0x00007ffff47b8cc9 in __GI_raise (sig=sig@entry=6) at
../nptl/sysdeps/unix/sysv/linux/raise.c:56
#1  0x00007ffff47bc0d8 in __GI_abort () at abort.c:89
#2  0x00007ffff4e66829 in ?? () from /usr/lib/x86_64-linux-gnu/libasan.so.0
#3  0x00007ffff4e5d3ec in ?? () from /usr/lib/x86_64-linux-gnu/libasan.so.0
#4  0x00007ffff4e64012 in ?? () from /usr/lib/x86_64-linux-gnu/libasan.so.0
#5  0x00007ffff4e63121 in __asan_report_error () from
/usr/lib/x86_64-linux-gnu/libasan.so.0
#6  0x00007ffff4e5d7f7 in __asan_report_store4 () from
/usr/lib/x86_64-linux-gnu/libasan.so.0
#7  0x0000000000400c64 in main () at array.cpp:12
```

Specifically, we first `run` our executable until the out-of-bounds memory access is detected by `address sanitizer`, and since we've set the `abort_on_error` flag our executable halts within the debugger accordingly. Then, we simply call `backtrace` such that we can see how our program arrived at the error in question. The last line shows that line 12 of our `array.cpp` program is responsible for triggering the error. We may use `q` to quit out of the debugger and return to console.

5.5 Multidimensional Static Arrays

See `src/md_array.cpp`:

```
1  #include <iostream>
2
3  int main() {
4    // declare a 2D array
5    int a[2][2];
6
7    a[0][0] = 0;
8    a[1][0] = 1;
9    a[0][1] = 2;
10   a[1][1] = 3;
11
12   std::cout << "a = " << a << std::endl;
13
14   std::cout << "a[0][0] = " << a[0][0] << std::endl;
15   std::cout << "a[1][0] = " << a[1][0] << std::endl;
16   std::cout << "a[0][1] = " << a[0][1] << std::endl;
17   std::cout << "a[1][1] = " << a[1][1] << std::endl;
18
19   return 0;
20 }
```

Compile and run:

```
$ g++ -Wall -Wconversion -Wextra md_array.cpp -o md_array
$ ./md_array
a = 0x7fffe2a9e8d0
a[0][0] = 0
a[1][0] = 1
a[0][1] = 2
a[1][1] = 3
```

Note: the first output line prints the memory address.

5.6 Array operations

You can't do assignment with C++ static arrays. Let's modify `src/md_array.cpp`:

```
#include <iostream>

int main() {
  // declare a 2D array
  int a[2][2];

  // declare another 2D array
  int b[2][2];

  b = a;   // Yields an error!
}
```

If we attempt to compile:

```
$ g++ -Wall -Wconversion -Wextra md_array.cpp -o md_array
md_array.cpp: In function 'int main()':
md_array.cpp:10:5: error: invalid array assignment
   b = a;
   ^
```

The `operator=` has not been overloaded to handle array assignment. There's something related that we could do, which is to assign b to *point* to the same sequence of bits that `a` refers to, but this requires a bit more understanding of pointers and addresses than we have time for today.

6 C++ `for`-loop

Start with an example. See `src/for_loop1.cpp`:

```
#include <iostream>

int main() {
  for (int i = 0; i < 10; ++i) {
    std::cout << "i = " << i << std::endl;
  }
  return 0;
}
```

Compile and run:

```
$ g++ -Wall -Wconversion -Wextra for_loop1.cpp -o for_loop1
$ ./for_loop1
i = 0
i = 1
i = 2
... You know how this works by now :-)
i = 9
```

6.1 Anatomy of a for loop

There are a few key ingredients. The most basic C++ syntax affords a lower level of abstraction than does Python for-loops; this is remedied in C++11.

```
1 for (expression1; expression2; expression3) { /* loop body */ }
```

How does this work?

- `expression1` is evaluated *exactly once* at the start of the loop.
- `expression2` is a conditional statement evaluated at the start of *each loop iteration*, where we terminate if the conditional statement returns `false`.
- Finally, `expression3` is one which is evaluated at the *end* of *each iteration*.

Another `for` loop example File `src/for_loop2.cpp`:

```
1  #include <iostream>
2
3  int main() {
4    int n, sum;
5
6    sum = 0;
7    for (n = 0; n < 101; ++n) {
8      sum += n;
9    }
10
11   std::cout << "sum = " << sum << std::endl;
12   return 0;
13 }
```

Output:

```
$ g++ -Wall -Wconversion -Wextra for_loop2.cpp -o for_loop2
$ ./for_loop2
sum = 5050
```

6.2 Increment and Decrement

These are operators that can be applied to (implicitly convertible) numeric types in any context, but are quite useful in control-flow. Specifically, increment (`++`) and decrement (`--`) are just shorthand for incrementing or decrementing by one.

```
#include <iostream>

int main() {
  int n = 2;
  std::cout << "n = " << n << std::endl;
  n++;
  std::cout << "n = " << n << std::endl;
  ++n;
  std::cout << "n = " << n << std::endl;
  n--;
  std::cout << "n = " << n << std::endl;
  --n;
  std::cout << "n = " << n << std::endl;

  return 0;
}
```

Output:

```
$ g++ -Wall -Wconversion -Wextra increment.cpp -o increment
$ ./increment
n = 2
n = 3
n = 4
n = 3
n = 2
```

6.2.1 Pre vs. Post Increment

You can put them before or after a variable, but the exact interpretation/computation performed is *slightly* different: `++x` *first* increments data referenced by variable `x` and *then* returns the value, whereas you can think of `x++` as *first* returning the value originally held by `x` and *then* incrementing the underlying data referenced by `x`. See `src/increment.cpp`

Example (`src/increment2.cpp`):

```
#include <iostream>

int main() {
  int a = 1;
  std::cout << "            a: " << a   << std::endl;
  std::cout << "return of a++: " << a++ << std::endl;
  std::cout << "            a: " << a   << std::endl;
  std::cout << "return of ++a: " << ++a << std::endl;
  std::cout << "            a: " << a   << std::endl;
  return 0;
}
```

Output:

```
            a: 1
return of a++: 1
            a: 2
return of ++a: 3
            a: 3
```

So, we see that using pre or post-increment doesn't really matter unless we directly capture the value returned by the increment expression; if we simply wish to use the variable that we increment in a *separate* expression at a *latter* point in the program, there is no difference.

6.3 Iterating through an array

`src/for_loop3.cpp`:

```
1  #include <iostream>
2
3  int main() {
4    int n = 5;
5    double a[16];
6
7    /* Initialize a to zeros. */
8
9    for (int n = 0; n < 16; n++) {
10     a[n] = 0.;
11   }
12
13   std::cout << "a[0] = " << a[0] << std::endl;
14   std::cout << "n = " << n << std::endl;
15
16   return 0;
17 }
```

```
$ g++ -Wall -Wconversion -Wextra for_loop3.cpp -o for_loop3
$ ./for_loop3
a[0] = 0
n = 5
```

6.4 Syntactic Variations on for loop

We might not declare a variable in the first expression of a `for`-loop.

```
1  #include <iostream>
2
3  int main() {
4    int n = 0, sum = 0;    // n declared with excessive scope...
5    for (; n <= 100;) {
6      sum += n;
7      n++;
8    }                      // n is not needed outside of the for loop...
9    std::cout << "sum = " << sum << std::endl;
10   return 0;
11 }
```

Alternatively, perhaps we want to control how our loop counters are modified with each iteration in a way can't be described by a single expression; the following is a toy example since we could clearly place the simple expression `n++` in third argument of the `for`-loop.

```
#include <iostream>

int main() {
  int sum = 0;
  // n may be declared in the first for loop expression
  for (int n = 0; n <= 100;) {
    sum += n;
    n++;
  }
  std::cout << "sum = " << sum << std::endl;

  return 0;
}
```

6.4.1 Infinite loops

See `src/inf_loop.cpp`:

```
#include <iostream>

int main() {
  for (;;) {
  }

  return 0;
}
```

```
$ ./inf_loop
```

This can generally be terminated with `Ctrl-c`. If that doesn't work use `Ctrl-z` to background and then kill that job number.

6.4.2 `for` loop brackets

```
#include <iostream>

int main() {
  int sum = 0;

  for (int n = 0; n < 101; n++)
    sum += n; // One statement loop body, does not have to be enclosed

  std::cout << "sum = " << sum << std::endl;

  return 0;
}
```

6.4.3 Common mistakes

```cpp
#include <iostream>

int main() {
  int n, sum, product;

  sum = 0;
  product = 1;
  for (n = 1; n < 11; n++)
    sum += n;
    product *= n; // Not part of for loop


  std::cout << "sum = " << sum << std::endl;
  std::cout << "product = " << product << std::endl;

  return 0;
}
```

```cpp
#include <iostream>

int main() {
  int n;

  int sum = 0;
  for (n = 0; n < 101; n++); // no loop body
  {
    sum += n;
  }

  std::cout << "sum = " << sum << std::endl;

  return 0;
}
```

6.5 Nested loops example

```cpp

#include <iostream>

int main() {
  double a[3][3];
  /* Initialize a to zeros. */
  for (int n = 0, i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
      a[i][j] = n;
      n++;
    }
  }

  /* Print a. */
  for (int i = 0; i < 3; i++) {
    std::cout << a[i][0];
    for (int j = 1; j < 3; j++)
      std::cout << " " << a[i][j];
    std::cout << std::endl;
  }
}
```

6.6 `while` loop

```
1
2  #include <iostream>
3
4  int main() {
5    int n = 0, sum = 0;
6    while (n <= 100) {
7      sum += n;
8      n++;
9    }
10   std::cout << "sum = " << sum << std::endl;
11
12   return 0;
13 }
```

6.6.1 Common mistake

```
1  #include <iostream>
2
3  int main() {
4    int n = 0, sum = 0;
5
6    while (n <= 100); // no loop body
7    {
8      sum += n;
9      n++;
10   }
11   std::cout << "sum = " << sum << std::endl;
12
13   return 0;
14 }
```

6.6.2 `do-while` loop

- A while loop may execute zero times if the conditional is not true on initial evaluation
- C/C++ has a do-while loop that is very similar to a while loop, but always executes at least once

```
1  do {
2    // loop body
3  } while (expression);
```

Note the semicolon at the very end!

7 Recommended reading

- **C++ Primer, Fifth Edition** by Lippman et al.
- Available on Safari ProQuest: http://proquest.safaribooksonline.com/book/programming/cplusplus/9780133053043
- Chapter 1: Getting Started, Sections 1.1 - 1.3
- Chapter 2: Variables and Basic Types, Sections 2.1 - 2.2
- Chapter 3: Strings, Vectors, and Arrays: Sections 3.1 - 3.2

8 Resources

- Online C++ compiler for small tests: http://coliru.stacked-crooked.com/
- http://www.cppreference.com
- http://www.cplusplus.com
- http://www.linfo.org/index.html