CME 211: Lecture 15

Topics

- static arrays
- variable scope
- looping

Static arrays

The size (length) of static arrays is known at compile time. See `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;
}
```

Compile and look at the output:

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

Out of bounds access

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
```

Nothing bad happened here. But, the behavior is undefined. This could have cause the universe to collapse. We're lucky it did not.

Address Sanitizer

We can instrument the executable to detect out of bound memory access in static arrays. To do this we enable the "address sanitizer" at compile time.

- https://code.google.com/p/address-sanitizer/
- Incorporated into GNU (and other) compilers
- Adds instrumentation around memory accesses
- Enabled at compile time
- 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
```

Notes:

- The `-g` flag adds debugging symbols to the output executable
- The `-fsanitize=address` enables the address sanitizer
- In `bash` the \ character allows line continuation

Testing Address Sanitizer

```
$ ./array
=================================================================
==23777== ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7fff6e11364c at pc 0x400c64 bp 0x7
WRITE of size 4 at 0x7fff6e11364c thread T0
    #0 0x400c63 (/afs/ir.stanford.edu/users/n/w/nwh/git/cme211-notes/lecture-15/src/array+0x400c63)
    #1 0x7f1dbf75dec4 (/lib/x86_64-linux-gnu/libc-2.19.so+0x21ec4)
    #2 0x400a58 (/afs/ir.stanford.edu/users/n/w/nwh/git/cme211-notes/lecture-15/src/array+0x400a58)
Address 0x7fff6e11364c is located at offset 44 in frame <main> of T0's stack:
  This frame has 1 object(s):
    [32, 44) 'a'
HINT: this may be a false positive if your program uses some custom stack unwind mechanism or swapconte
      (longjmp and C++ exceptions *are* supported)
Shadow bytes around the buggy address:
  0x10006dc1a670: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x10006dc1a680: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x10006dc1a690: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x10006dc1a6a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x10006dc1a6b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x10006dc1a6c0: 00 00 00 00 f1 f1 f1 f1 00[04]f4 f4 f3 f3 f3 f3
  0x10006dc1a6d0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x10006dc1a6e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x10006dc1a6f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x10006dc1a700: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x10006dc1a710: 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
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:     fa
  Heap righ redzone:     fb
  Freed Heap region:     fd
  Stack left redzone:    f1
  Stack mid redzone:     f2
  Stack right redzone:   f3
  Stack partial redzone: f4
  Stack after return:    f5
  Stack use after scope: f8
  Global redzone:        f9
  Global init order:     f6
  Poisoned by user:      f7
  ASan internal:         fe
==23777== ABORTING
```

Address Sanitizer and `gdb`

We can use the GNU debugger `gdb` to get more precise information about the error:

```
$ export ASAN_OPTIONS=abort_on_error=1
$ gdb ./array
...
(gdb) run
Starting program:
/afs/ir.stanford.edu/users/n/w/nwh/git/cme211-notes/lecture-15/src/array
... [lots of output] ...
(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
(gdb) q
```

Multidimensional static arrays

See `src/md_array.cpp`:

```
#include <iostream>

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

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

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

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

  return 0;
}
```

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.

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;

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

  std::cout << "a = " << a << std::endl;
  std::cout << "b = " << b << std::endl;

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

  return 0;
}
```

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;
     ^
```

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

Scope examples

```
#include <iostream>

int main() {
  {
    int n = 5;
  }

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

  return 0;
}
```

Output:

```
$ 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;
                          ^
```

Scope examples

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

int main() {
  std::string n = "Hi";
  std::cout << "n = " << n << std::endl;

  {
    int n = 5;
    {
      std::cout << "n = " << n << std::endl;
    }
  }

  return 0;
}
```

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

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
i = 3
```

```
i = 4
i = 5
i = 6
i = 7
i = 8
i = 9
```

Anatomy of a for loop

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

- `expression1`: evaluated once at the start of the loop
- `expression2`: conditional statement evaluated at the start of each loop iteration, terminate if conditional statement returns false
- `expression3`: evaluated at the end of each iteration

Another `for` loop example

File `src/for_loop2.cpp`:

```
#include <iostream>

int main() {
  int n, sum;

  sum = 0;
  for (n = 0; n < 101; ++n) {
    sum += n;
  }

  std::cout << "sum = " << sum << std::endl;
  return 0;
}
```

Output:

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

Increment and decrement

- Increment (`++`) and decrement (`--`) are just shorthand for incrementing or decrementing by one
- You can put them before or after a variable
- See `src/increment.cpp`

```
#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
```

Prefix (`++n`) vs. postfix (`n++`) increment operators

- The postfix operator creates a temporary and returns the value before incrementing
- The prefix operator returns a reference after incrementing

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
```

Iterating through an array

`src/for_loop3.cpp`:

```
#include <iostream>
```

```
int main() {
  int n = 5;
  double a[16];

  /* Initialize a to zeros. */

  for (int n = 0; n < 16; n++) {
    a[n] = 0.;
  }

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

  return 0;
}
```

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

Variations on for loop

```
#include <iostream>

int main() {
  int n = 0, sum = 0;
  // here n is declared with excessive scope
  // n is not needed outside of the for loop
  for (; n <= 100;) {
    sum += n;
    n++;
  }
  std::cout << "sum = " << sum << std::endl;

  return 0;
}
```

Variations on 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;
}
```

Infinite loops

See `src/inf_loop.cpp`:

```
#include <iostream>

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

  return 0;
}
```

```
$ ./inf_loop
```

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

`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;
}
```

Common mistake

```
#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;
}
```

Common mistake

```
#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;
}
```

Nested loops example

```
#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;
  }

  return 0;
}
```

while loop

```
#include <iostream>

int main() {
  int n = 0, sum = 0;
  while (n <= 100) {
    sum += n;
    n++;
  }
  std::cout << "sum = " << sum << std::endl;

  return 0;
}
```

Common mistake

```
#include <iostream>

int main() {
  int n = 0, sum = 0;

  while (n <= 100); // no loop body
  {
    sum += n;
    n++;
  }
  std::cout << "sum = " << sum << std::endl;

  return 0;
}
```

`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

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

Note the semicolon at the very end!

Reading

- **C++ Primer, Fifth Edition** by Lippman et al.
- http://proquest.safaribooksonline.com/book/programming/cplusplus/9780133053043
- Chapter 1: Getting Started, Sections 1.4.1 and 1.4.2 (while and for)
- Chapter 3: Strings, Vectors, and Arrays: Sections 3.5 and 3.6 (arrays)