CSCI 2270
Data Structures and Algorithms
Lecture 3

Elizabeth White
elizabeth.white@colorado.edu
Office hours: ECCS 128
Wed 1pm-2pm
Thurs 2pm-3pm

Administrivia

- HW0 is due today
- Lab 1 is due today
- Lecture questions from today are due Sunday night
- No lecture on Monday
- Lab will still take place next week

Integers at compile time

```
void example4()
{
      int a = 5;
      int* a_ptr = &a;
}
```

This **`a_ptr`** is a *pointer* to **`a`**. A pointer stores an address in memory where a variable is living. To make a pointer to an **`int`**, we need to add the **`*`** to the **`int`** type when we declare it, as below.

```
int* a_ptr = &a;
```

It's easy to misplace **`&`** and **`*`** at first.

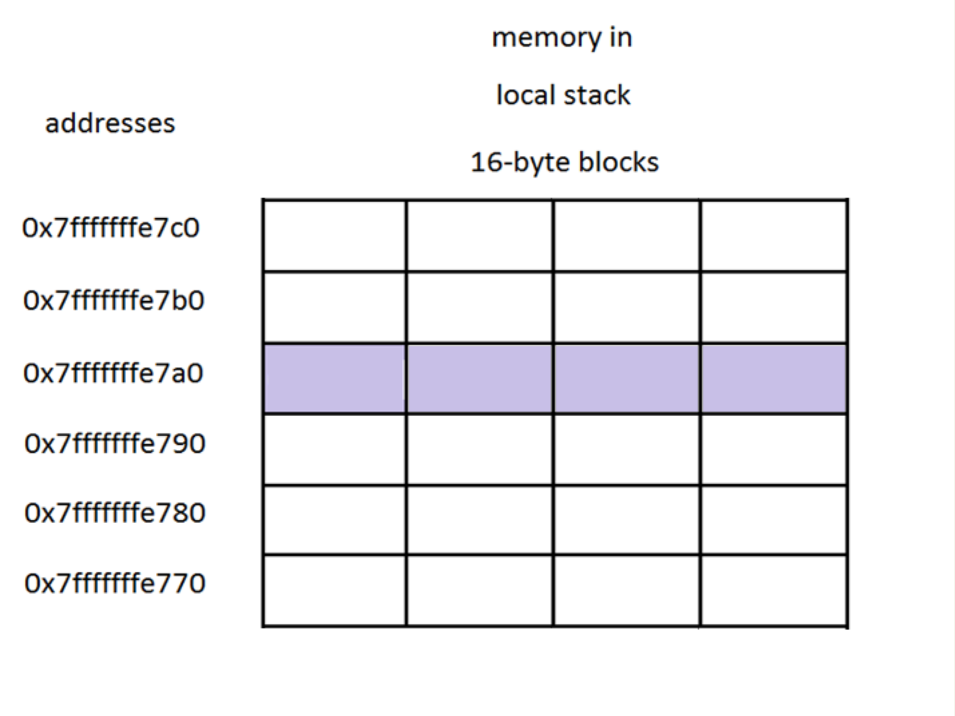
Constant size integer array

```
void example9()
{
      int b[4];
      b[0] = 1;
      b[1] = 2;
      b[2] = 4;
      b[3] = 8;
}
```

We can make a variable that's an array of integers, instead of one single integer. This code is telling the compiler to find room for 4 integers (that's what the `b[4]` does).

Constant size integer arrays

```
void example9()
{
        int b[4];
        b[0] = 1;
        b[1] = 2;
        b[2] = 4;
        b[3] = 8;
}
```



For arrays, the compiler stores each integer in adjacent locations in memory. After the line `int b[4];`, it reserves four integer size blocks, one right after the next, for this array.

Constant size integer arrays

```
void example9()
{
        int b[4];
        b[0] = 1;
        b[1] = 2;
        b[2] = 4;
        b[3] = 8;
}
```

Then we can assign to any of those 4 integers, from **`b[0]`** to **`b[3]`**.

Note that arrays in C++ count from 0, not from 1.

Why is array size constant?

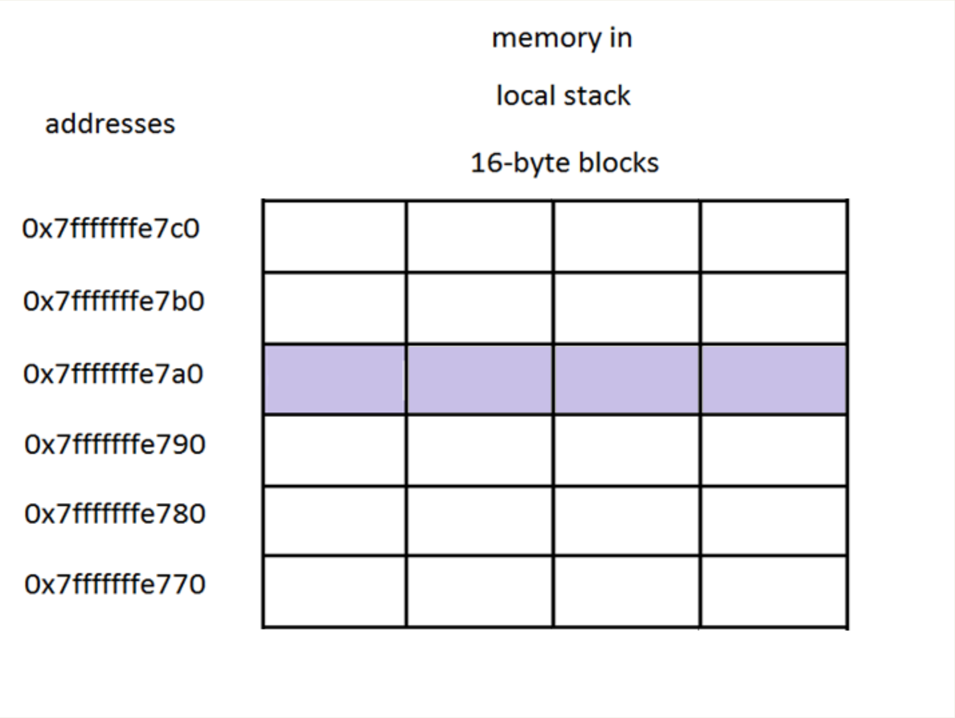
```
void example9()
{
        int b[4];
        b[0] = 1;
        b[1] = 2;
        b[2] = 4;
        b[3] = 8;
}
```

An array like `b`, which gets declared with a size in the square brackets (`[]`), is stuck at its starting size forever. We can change the integers in `b[0]` through `b[3]`, but *we can't change b's size* from 4 to something like 10 once we build it with 4 slots.

Naughty constant size integer array

```
void example10()
{
      int b[4];
      b[4] = 5;
}
```

What happens here?



Printing out an integer array with a counter

```
void example11()
{
     int b[4];
     b[0] = 1; b[1] = 2; b[2] = 4; b[3] = 8;
     for (int i = 0; i < 4; i++)
          cout << b[i] << endl;
}
```

We can make a variable that's an array of integers, instead of one single integer. This code is telling the compiler to find room for 4 integers (that's what the **`b[4]`** does).

Printing out an integer array with a counter and a pointer

```
void example12()
{
      int b[4];
      int* b_ptr = &(b[0]);
      b[0] = 1; b[1] = 2; b[2] = 4; b[3] = 8;
      for (int i = 0; i < 4; i++)
      {
            cout << b_ptr << " " << *b_ptr <<
endl;
            b_ptr++;
      }
}
```

Printing out an integer array with a pointer

```
void example13()
{
	int b[4];
	int* b_ptr;
	b[0] = 1; b[1] = 2; b[2] = 4; b[3] = 8;
	for (b_ptr = &(b[0]); b_ptr != &(b[4]);
b_ptr++)
	{
		cout << b_ptr << " " << *b_ptr <<
endl;
	}
}
```

Floating point numbers

A 4-byte (single precision) floating point number uses:

1 bit for the sign,

8 bits for the exponent, and

23 bits for the binary digits.*

An 8-byte (double precision) floating point number uses:

1 bit for the sign,

11 bits for the exponent, and

52 bits for the binary digits.*

* To a first approximation, anyway. Take 2400 to learn the details. This is all you need for 2270.

Floating point numbers in C++

C++ offers you the choice between floats and doubles.

Floats in the VM

- Take up 32 bits of memory, or 4 bytes.
- Smallest magnitude: 1.17549e-38
- Largest magnitude: 3.40282e+38

Doubles in the VM

- Take up 64 bits of memory, or 8 bytes.
- Smallest magnitude: 2.22507e-308
- Largest magnitude: 1.79769e+308

Recall binary integers in C++

Computers store integers in binary form.

Decimal	Binary
0	0
1	1
7	111
87	1010111

For binary numbers, a 1 in the last place is 2^0,

and a 1 in the next to last place is 2^1,

and so forth. All powers of 2.

Binary 1010111 = $2^6 + 2^4 + 2^2 + 2^1 + 2^0$

= 64 + 16 + 4 + 2 + 1 = decimal 87.

Recall binary integers in C++

To convert a decimal number to a binary one, divide the decimal number by 2 and keep track of the remainder.

23/2 = 11	23 % 2 = 1
11/2 = 5	11 % 2 = 1
5/2 = 2	5 % 2 = 1
2/2 = 1	2 % 2 = 0
1/2 = 0	1 % 2 = 1

Reading the remainders out in reverse order gives us the binary: 10111

Floating point numbers

Like integers, computers store floating point numbers in binary form. Consider these little numbers:

Decimal	Binary
0	0
0.5	0.1
0.25	0.01
0.125	0.001

For binary numbers,

a 1 in the first place after the decimal (.) is 2^{-1}, or ½,

and a 1 in the next place is 2^{-2}, or ¼,

and so forth. All still powers of 2.

Why should we care?

http://sydney.edu.au/engineering/it/~alum/patriot_bug.html

Time step: 0.1 sec

Accumulated error after 100 hours: 0.34 sec

Error in missile position: 690 m

Boom!