CSCI 2270
Data Structures and Algorithms
Lecture 3—Memory part 2

Elizabeth White
elizabeth.white@colorado.edu
Office hours: ECCS 112
Wed 9:30am-11:00am
Thurs 10:00am-11:30am

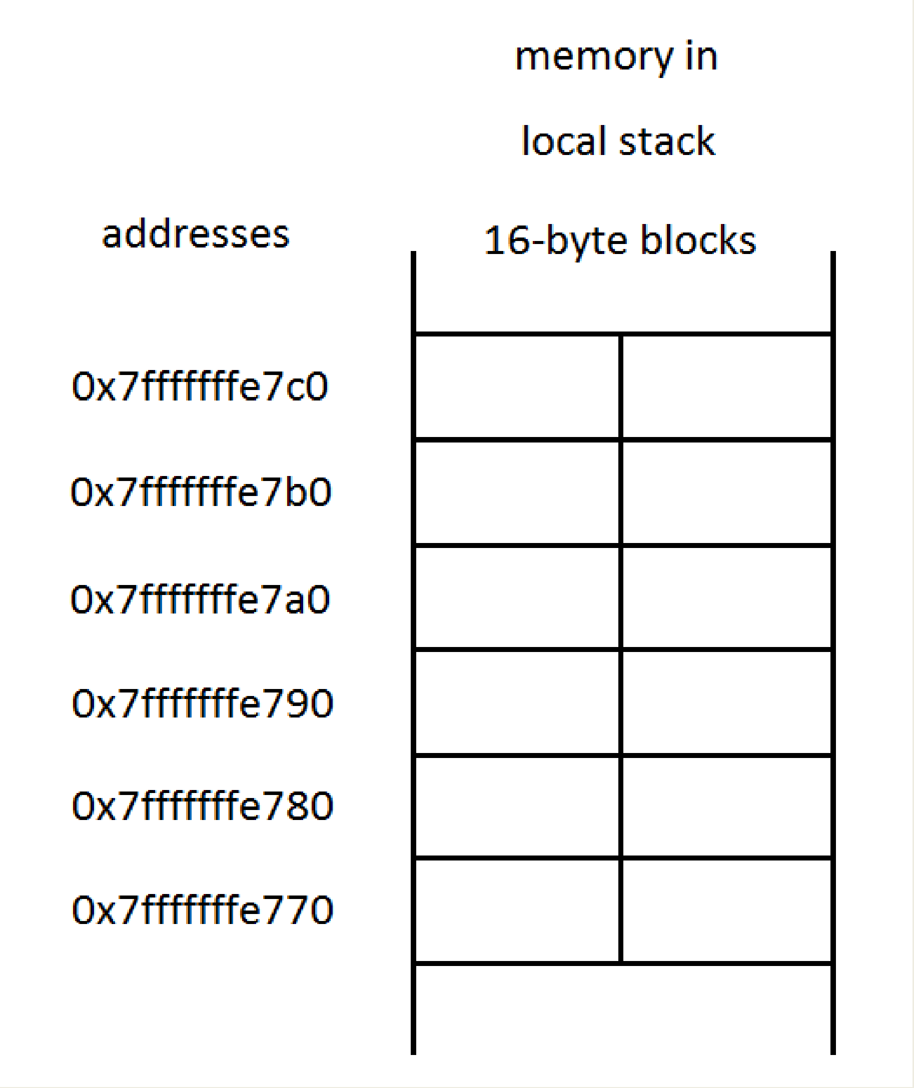
Administrivia

- HW1 will post today
  - It’s posting early and will be due Sunday, Feb. 2nd
  - You have not seen everything you’ll need for this, so don’t jump on it and drive yourself crazy
  - We’ll cover function parameters and classes next week
  - LAs will begin to come online today, too
- HW0 is due this week
- No lab next week
- No lecture on Monday
- Lab resumes week after next

Integers at compile time

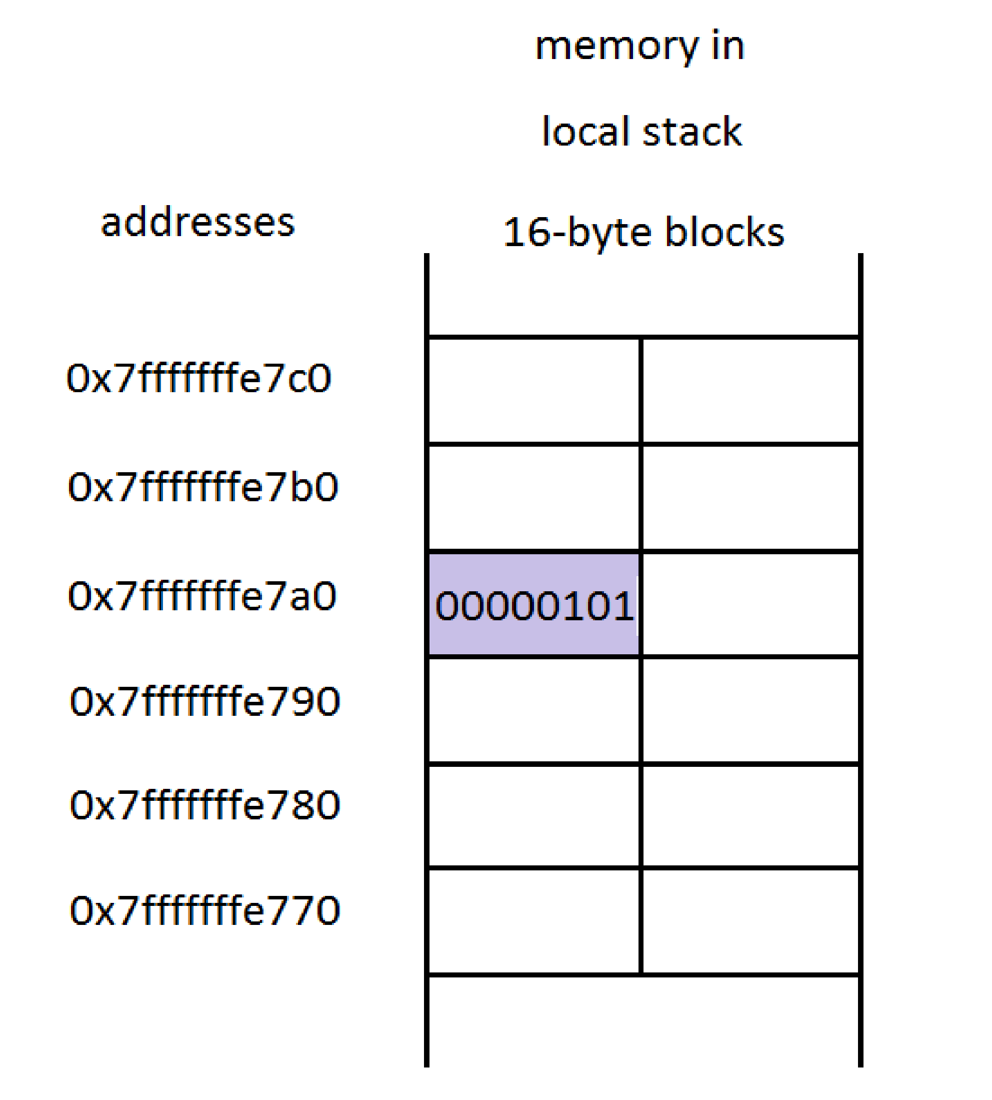
The compiler reserves that memory for a using a local array of memory (called the stack).

I'm drawing it in 8-bit blocks, and I'm showing you the hexadecimal addresses of some of the blocks.



Integers at compile time

After the compiler sees the line `int a = 5;` it reserves a spot for that integer `a` on this stack of memory, and writes the 5 into this spot (in binary). If each slot is a byte, this `int` will take up 4 slots. The operating system makes sure that no other stuff gets written into here, for as long as `a` exists.



Integers at compile time

```
void example1()
{
    int a = 5;
}
```

At the end of our function, just at the closing bracket **}**, the **int a** gets destroyed. Destroying it means that *we no longer keep that memory reserved* on the stack for **a**.

Now other variables can be written into **a**'s former slot in the memory stack. We can't predict exactly when this will happen, but it's only a matter of time before something else writes data over this slot.

Integers at compile time

```
void example2()
{
      int a = 5;
      cout << a << endl;
}
```

When it runs, this code prints out the current value of **`a`** in memory. That's 5.

Integers at compile time

```
void example3()
{
        int a = 5;
        cout << &a << endl;
}
```

This code prints out the current address where a is stored in the stack memory. We get the address of **`a`**, instead of its value, by putting an ampersand in front (**`&a`**). Notice that this address is one of those hexadecimal numbers.

From this example, you can deduce that any variable in C++ also knows where it lives in memory.

Integers at compile time

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

This code makes a separate variable called **`a_ptr`** to store the address of **`a`**. Again, we get the address by using the **`&`** sign in front of the variable name.

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 = &c;
```

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

Integers at compile time

```
void example6()
{
      int a = 5;
      int* a_ptr = &a;
      a = a + 1;
}
```

We can still change **`a`** in the normal way, as in the last line above, where it becomes 6.

Integers at compile time

```
void example7()
{
    int a = 5;
    int* a_ptr = &a;
    *a_ptr = *a_ptr + 1; // a is now 6
}
```

Here, at the closing bracket, we've used **`a_ptr`** to change the value of **`a`** (but not **`a_ptr`**). Instead of 5, **`a`** is now 6.

Integers at compile time

```
void example7()
{
        int a = 5;
        int* a_ptr = &a;
        *a_ptr = *a_ptr + 1; // a is now 6
}
```

We change **`a`** by dereferencing **`a_ptr`**, and we get that by saying **`*a_ptr`**. Dereferencing means that we find the address that **`a_ptr`** stores, and then we go to that memory slot and grab **`a`**, which is stored there. (It's a little extra work.)

Integers at compile time

```
void example8()
{
      int a = 5;
      int* a_ptr = &a;
      a++;              // a becomes 6
      (*a_ptr)++;       // a becomes 7
}
```

This code is similar to examples 6 and 7. First we use the **`++`** operator to increase a by 1, rather than doing a direct addition like **`+ 1`**. And then we dereference **`a_ptr`** to get **`a`** and increase that by 1 again, using the **`++`** operator. Now you can see how we can change **`a`**'s value using either its name or its address.

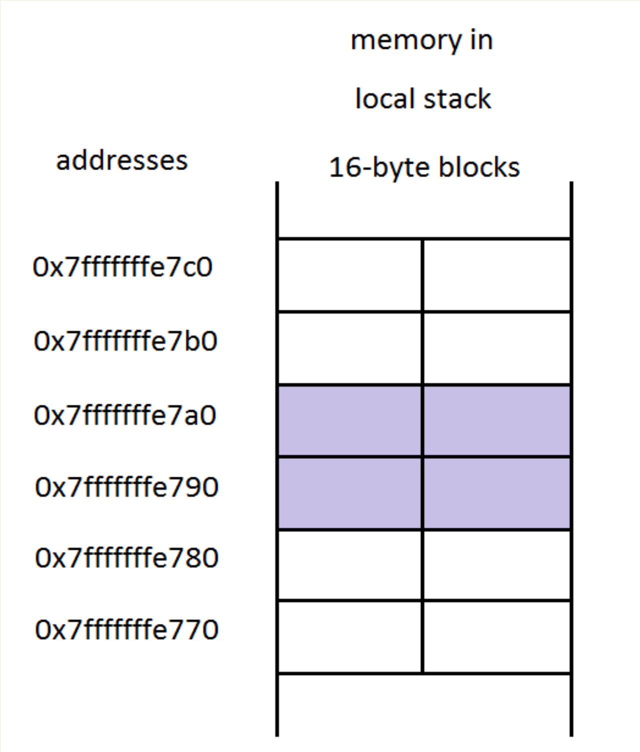
Constant size integer array

```
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;
}
```

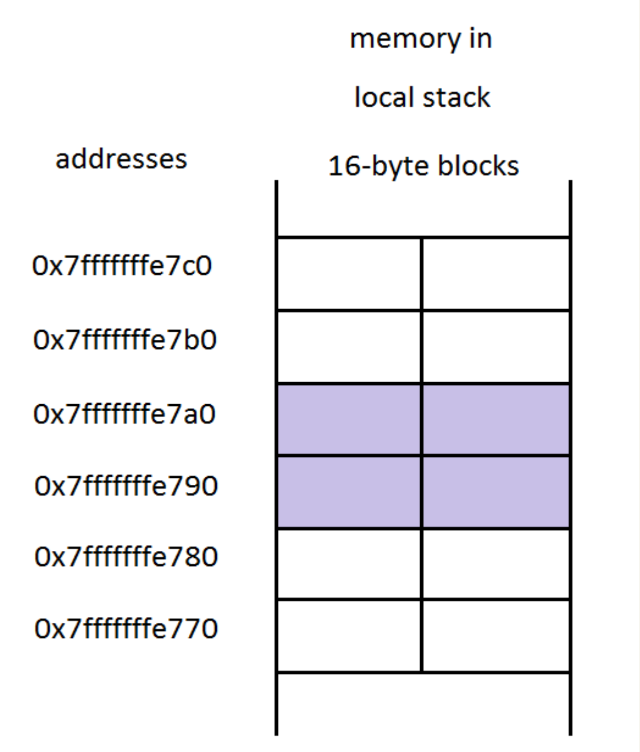
Then we can assign to any of those 4 integers, from `b[0]` to `b[3]`.

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

Naughty constant size integer array

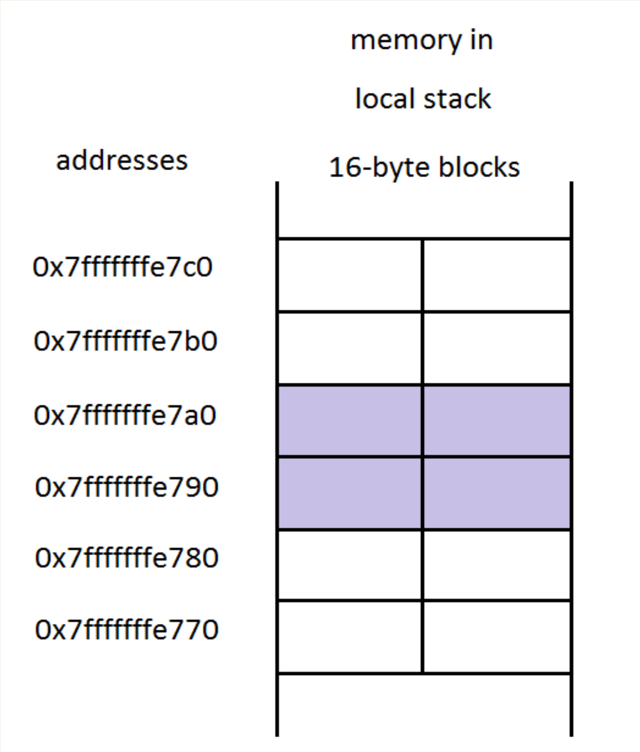
```
void example5()
{
    int b[4];
    b[4] = 5;
}
```

What happens here?



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.

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

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

and so forth. All still powers of 2.

Floating point numbers

Some familiar numbers don’t work out nicely in binary.

Decimal	Binary
0	0
0.1	0.000110011…
0.333…. (= 1/3)	0.010101…
0.2	0.00110011…

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!