CSCI 2270
Data Structures and Algorithms
Lecture 2—Memory part 1

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

Administrivia

- HW1 will post on Friday
  - LAs will come online then too
- HW0 is due this week

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

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

Integers in C++

- Each of those 0 or 1 binary digits takes up 1 bit in memory. One bit is the tiniest amount of memory we can use, because it can only store 0 or 1 (or equivalently, false or true). Since bits are so tiny, we often talk about bytes instead; a byte is equal to 8 bits stuck together, or a series of 8 zeros and ones.
- In my VM, integers take up 32 bits of memory, or 4 bytes.
- If you wanted to find this information out for your machine, you could use the sizeof command:

```
cout << "integer size = " << sizeof(int) << "
      bytes or " << sizeof(int) * 8 << " bits"
      << endl;
```

Integers in C++

- Remember that integers have a maximum and minimum size, too.
- For my VM, the minimum is -2147483648 and the maximum is 2147483647. You can find this (which may differ with your computer and C++ version) out by putting

```
#include <limits>
```

at the top of your code and including a line or 2 like:

```
cout << "minimum int value" <<
      numeric_limits<int>::min() << endl;
cout << "maximum int " <<
      numeric_limits<int>::max() << endl;
```

Integers in C++

- These limits come from having only 32 bits available to hold the data. We use 1 bit to store the sign of the integer. That leaves us 31 bits to use for binary digits.
- If you compute 2^31, you get 2147483648 (which is the size of the largest negative integer we can store).
- If you compute 2^31 – 1, you get 2147483647 (which is the size of the largest positive integer we can store; we get one less digit for the positive range because we also have to store the integer 0).

Integers in C++

- In other words, the memory space for an integer determines what range of numbers that integer is allowed to have.
- When integers ‘roll over’ from a large positive value to a large negative value or vice versa, it’s a direct consequence of this 32-bit limit.
- Full disclosure; computers actually store integers a little differently than this, but we’ll skip the details until CSCI 2400. You should just understand the relationship between the binary bits of memory in an int and the range of values it can take on. This same logic governs how other data types are stored, too.

Integers in C++

What is the maximum value that an integer of 8 bytes (in C++, these integers are called longs) could store?

A) $2^{64} - 1$
B) $2^{63} - 1$
C) 2^{64}
D) 2^{63}
E) 0

Integers in C++

What is the minimum value that an integer of 8 bytes (in C++, these integers are called longs) could store?

A) $-(2^{64} - 1)$
B) $-(2^{63} - 1)$
C) $-(2^{64})$
D) $-(2^{63})$
E) 0

Integers in C++

What is the maximum value that an *unsigned* integer of 8 bytes (in C++, these integers are called unsigned longs) could store? These integers are always >= 0.

A) $2^{64} - 1$
B) $2^{63} - 1$
C) 2^{64}
D) 2^{63}
E) 0

Integers in C++

What is the minimum value that an unsigned integer of 8 bytes (in C++, these integers are called unsigned longs) could store?

A) $-(2^{64} - 1)$
B) $-(2^{63} - 1)$
C) $-(2^{64})$
D) $-(2^{63})$
E) 0

Integers at compile time

Here is a very simple function. It makes one integer variable called a.

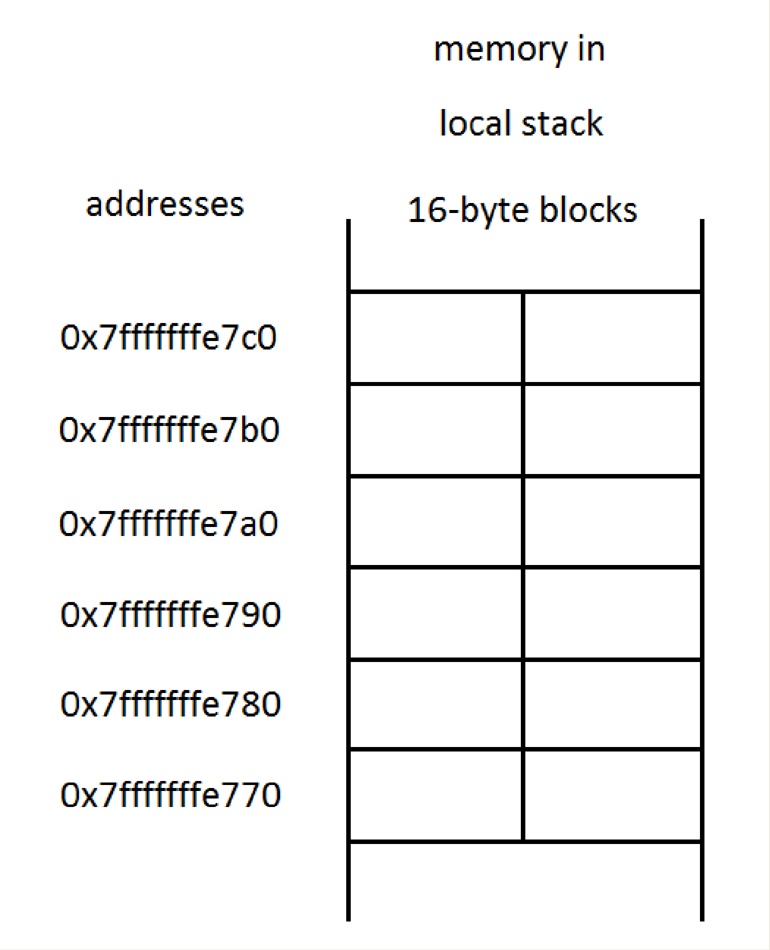
```
void example1()
{
      int a = 5;
}
```

When you compile this, what happens? The compiler sees the `int a` and recognizes this as an integer variable. Like you, the compiler knows how much memory an integer can take up…

Integers at compile time

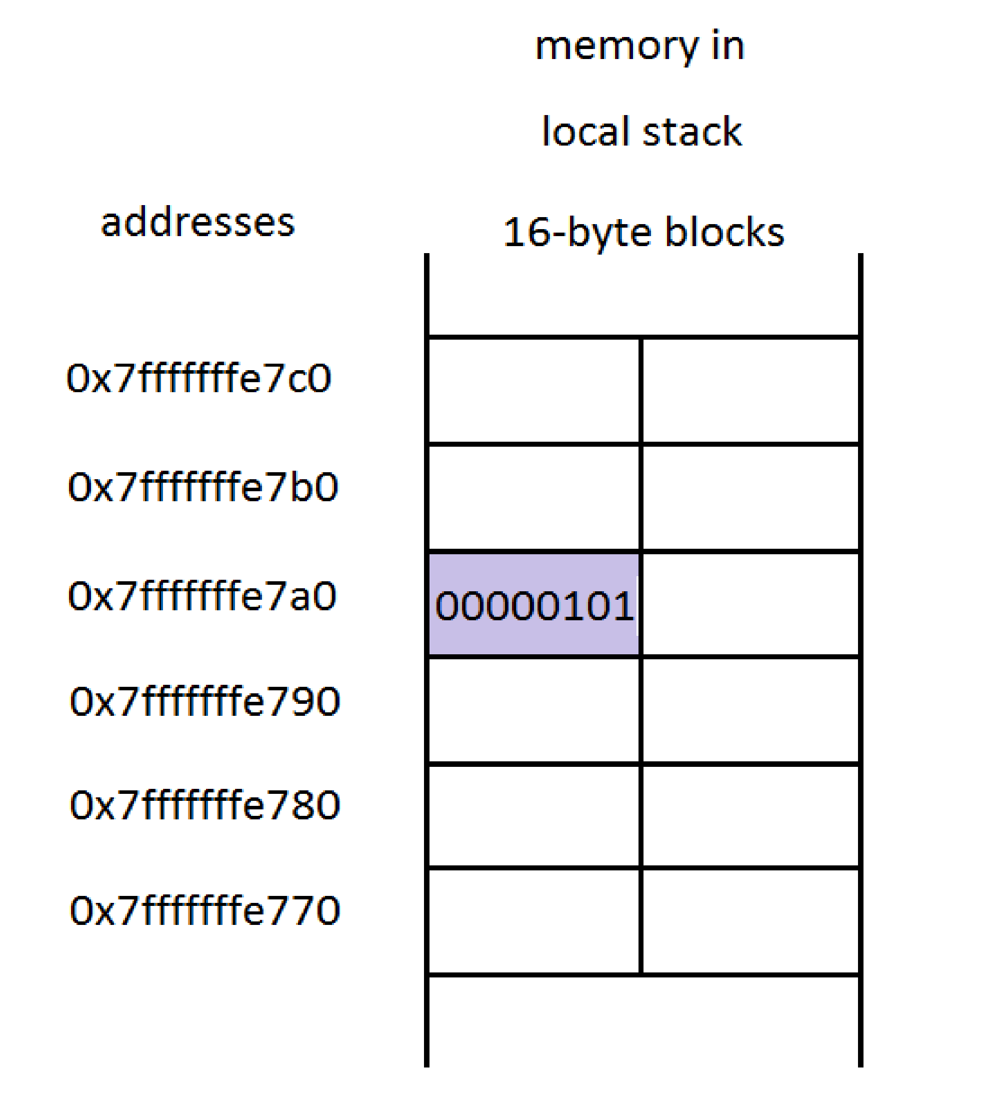
The compiler reserves that memory for a using a local array of memory (called the stack).

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 slots 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. That’s 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 for the address of a, called **`a_ptr`**. This guy's a *pointer* to a. All that means is that **`a_ptr`** is storing the address where a 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
```

Integers at compile time

```
void example5()
{
        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 example5()
{
        int a = 5;
        int* a_ptr = &a;
        *a_ptr = *a_ptr + 1; // a becomes 6
}
```

At the closing bracket, we've now used `a_ptr` to change a (but not `a_ptr`); instead of 5, a is now 6.

We do this by dereferencing `a_ptr`,

```
*a_ptr
```

which 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.

Integers at compile time

```
void example6()
{
      int a = 5;
      int* a_ptr = &a;
      a++;          // a becomes 6
      (*a_ptr)++; // a becomes 7
}
```

This code is just the same; we're just using the ++ operator to increase a by one, rather than a direct addition like + 1. At the closing bracket, instead of 5, a is now 7. But you can see how we can change a's value using either a's name or a's address.

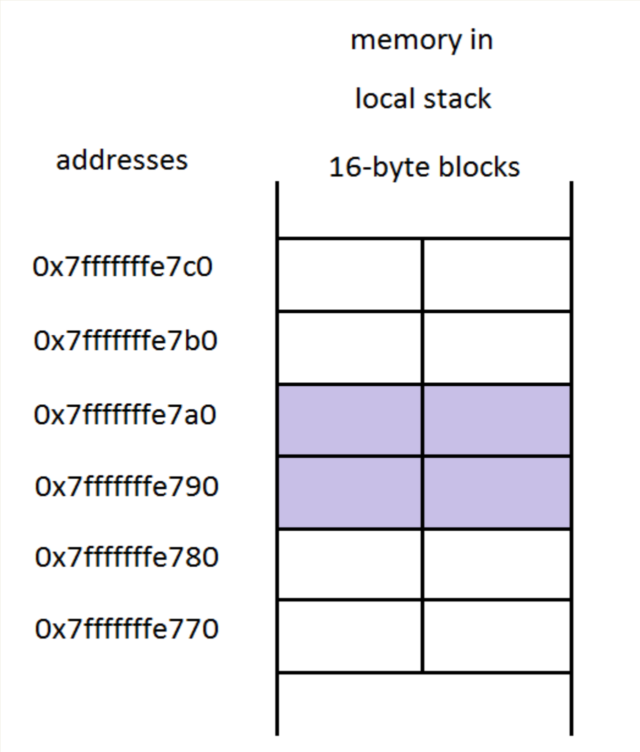
Integers at compile time

```
void example7()
{
        int a[4];
        a[0] = 1;
        a[1] = 2;
        a[2] = 4;
        a[3] = 8;
}
```

The last thing I want to show you today is that we can make an array of integers, instead of a single integer. This code is telling the compiler to find room for 4 integers (that's what the a[4] does). Then we can assign to any of those 4 integers (from a[0] to a[3]).

Integers at compile time

```
void example7()
{
        int a[4];
        a[0] = 1;
        a[1] = 2;
        a[2] = 4;
        a[3] = 8;
}
```



For arrays, the compiler stores each integer in adjacent locations on the stack; here, you can see it's reserved four integer size blocks, one right after the next.