CSCI 2270
Data Structures and Algorithms

Elizabeth White
elizabeth.white@colorado.edu
Office hours: ECCS 128
Wednesday 1:00-2:00pm
Thursday 2:00-3:00pm

What about a less dumb example?

Lab this week: letter shifter (lowercase, no letters except a-z)

- “abc” shifted by 3: “def”
- “abc” shifted by 26: “abc” again
- “abc” shifted by 25: “zab”

You’ll do this by functions calls to:

- print a string to the screen
- read a string from the keyboard
- compute a shifted string from an existing string

And you’ll write one bigger function that uses these little functions to get a string from the user, get a shift distance from the user, and print the shifted string.

Make sure you upload your files to the Moodle and show your TA when you get this working.

Admin

HW0 is posted and is due Friday night. It's about you. Let me know if you have trouble with moodle while filling it out.

I have posted some Java-to-C++ and C++ resources on the moodle.

If you struggled with Lab 1, please let me know; I'm trying to get a head count for that.

For Friday, read up on http://www.cplusplus.com/doc/tutorial/pointers/; stop at the section Pointers and Arrays

Why C++?

Exposes most of the important processes ‘under the hood’.

Allows you to do more direct memory management.

Allows faster operations in memory (via pointers).

These operations are also sometimes more dangerous!

C/C++ are still the best choice when speed is the main concern.

Much of the basic C++ language looks like Java, though there are philosophical differences. You’ll understand a lot more about what Java hides from its users after this class.

I have posted some Java-to-C++ and C++ resources on the moodle.

C++ for Java programmers

From pages 13-16 of http://cse.unl.edu/~choueiry/S13-235/files/Cpp4java.pdf.

The main() function still defines the place where your code begins running, inside curly brackets {}. In C++, main() doesn't need to be inside a class.

Instead of importing packages or classes to get extra functionality, we #include libraries

Instead of System.out.println("blah"), we say cout << "blah" << endl;

Instead of String s = System.console().readLine(), we say string s; cin >> s;

Primitive data types: the integer

The range of allowed integers is determined by how much space they take up in memory

What happens to an integer that gets too big or too small?

Integer data type 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 these numbers are 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 range, too.
- For my VM, the minimum value an integer can have 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 memory space 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++

- 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 memory space 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.

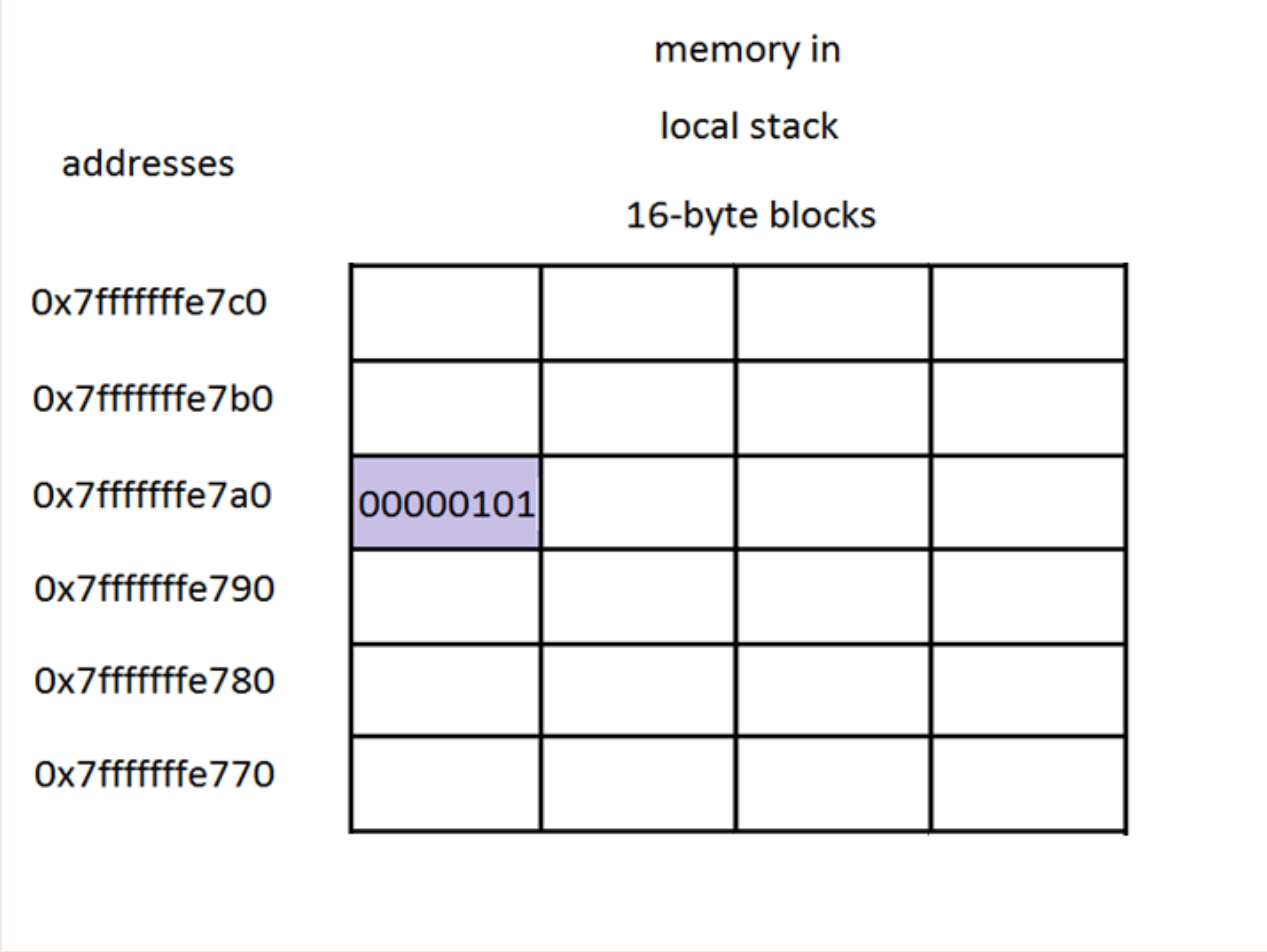
Integers at compile time

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. So it reserves that memory for this variable a using a local array of memory (called the stack).

Integers at compile time



I'm drawing the stack in 8-bit blocks, and I'm showing you the hexadecimal addresses of some of the blocks. a's in purple.

Integers at compile time

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

At the end of our function, just at the closing bracket }, that 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 0x… 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]).