From CS61Wiki

Contents

Data Representation 9/10/15

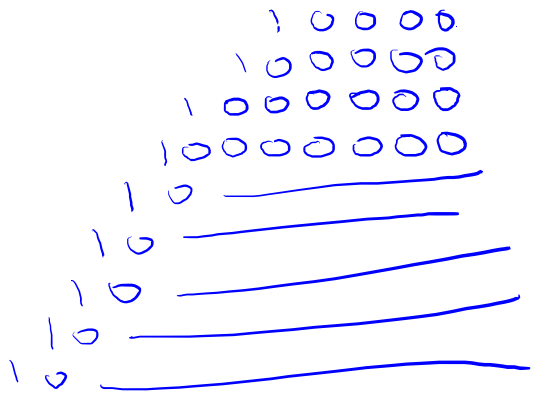
Learning Objectives

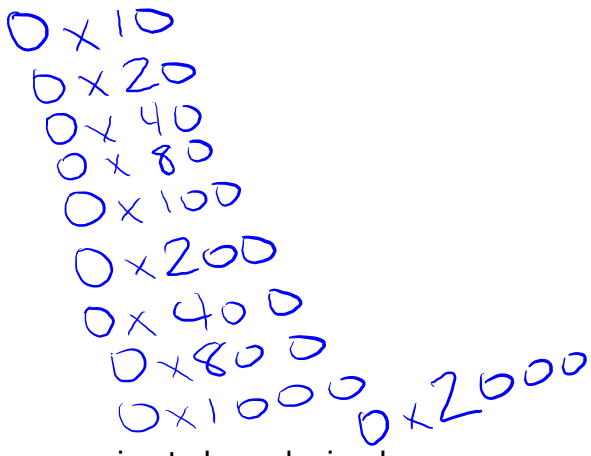
- Convert between binary, decimal, and hexadecimal
- Explain how both fundamental and derived data types are laid out in memory
- Use pointer arithmetic correctly
- Write code that places things in memory how you want them

Warmup: Converting between number bases

1. Write the following decimal numbers in binary.

- 16
- 32
- 64
- 128
- 256
- 512
- 1024
- 2048
- 4096
- 8192





2. Now convert each of the binary numbers in the previous exercise to hexadecimal.

Note: You should learn to recognize these numbers. Some people know larger powers of two, but these should be second nature to you.

3. One megabyte is 2 to what power? 2^{20}

4. One gigabyte is 2 to what power? 2^{30}

5. One terabyte is 2 to what power? 2^{40}

Note: You'll want to have these memorized as well.

6. Write 1 MB, 1 GB, and 1 TB in hexadecimal.

MB = 0x100000
GB = 0x40000000
TB = 0x10000000000

In the following exercises, numbers prefaced with 0x are in hexadecimal and numbers without preface are decimal. Compute the following:

7. 0x80480004 + 12 — 0x80480010

8. 0x804810FC + 8 — 0x80481104

9. 0xbfff801B + 16 — 0xbfff802B

10. 0xbfff8234 - 40 — 0xbfff820C

Examining data in memory

What is the largest value an `unsigned char` can hold? 255

In the `cs61-exercises/l03` directory, you will find the source code and makefile for today's coding. Examine the program `charsize.c` and then build it. Explain why `hexdump` printed out what it did.

Next, predict what will happen when you change the number on line 5 to 256.

Now, try it.

Explain what happened and why. You get 0 - 256 doesn't fit in a char

Now, let's look a little more closely at the output of `hexdump`. Take a look at the program `charval`. Predict what will happen when you run it and then run it.

Discussion question: At the level of abstraction of the machine, can it distinguish between the number 0x47 and the ASCII character "G"? Nope!

If the machine can distinguish between them, how does it do it? If the machine cannot distinguish between them, then how do we ascribe meaning to the values in a program? - Compiler does this or programmer printf format

Next, let's examine an array in memory. Read, build, and run the program `carry.c`. How many bytes were printed out? Why? 16 - that's how large the array is

Can you determine what the addresses are of each of the array elements? Use `gdb` to confirm that you got the right addresses.

Next, compare `carry.c` to `carry2.c`. Before you make and run `carry2`, predict what it will print out. After you run it, explain why it printed out what it did. 4 characters

Now, examine the program `iarray`. How many bytes will it print out? What do you suppose the addresses of the array elements will be? 64 (addresses are 4 bytes apart)

Build and run the program. Did it behave as you expected it to? Run it in gdb and examine the **addresses** of the array elements.

Write me down

We will check these answers at the end of class.

Assume that A is an array of type T. Write an expression that tells me how to compute the address of the i^{th} element of A. A + i * sizeof (T)

Pointer Arithmetic

The compiler knows the types of the variables it manipulates and even more importantly, it knows their size. As a result adding and subtracting numbers to pointers is a bit like magic. Read and run the program `parith.c`.

Take note of the difference between the values of the addresses in each pair.

Now, add a `printf` statement to print the difference between the two pointers. For example, for the character case, add the line: `printf("&carray[5] - carray = %zu\n", &carray[5] - carray);`

Add code to compute and print the same difference for the other two cases.

What do you notice? Can you explain why you get the results you got? Ptr arithmetic.

The type `uintptr_t` represents a value guaranteed to be large enough to hold a pointer, but treats the pointers as integers. Write a function that takes in two pointers to adjacent elements in an array and computes the `sizeof` the elements in the array. In other words, if I call `your function(&iarray[0], &iarray[1])` you should tell me that integers are four bytes. But if I call `your function(&darray[0], &darray[1])`, you should tell me that doubles are 8 bytes (you might not know the name of the types).

Write me down

No difference

If `P` is a pointer to a type `T` and `i` is an index, what is the difference between `&P[i]` and `P + i`?

Next, answer all the problems in ExerciseP and then check your answers here. If you are still having difficulty, review this.

Laying out items in memory

Now, let's dig a bit more deeply into precisely how values larger than a byte are laid out in memory. Examine the source code to `iexperiment.c`, and then build and run it. The value stored in `i`, 0x89ABCDEF, occupies four bytes; what are the addresses of those four bytes? Which address contains the **most significant digits** of the value?

A machine uses a **big endian** representation if the most significant bits of a multi-byte value reside in the first byte of the value (i.e., the one with the lowest address).
A machine uses a **little endian** representation if the least significant bits of a multi-byte value reside in the byte with the first byte of the value (i.e., the one with the lowest address).

Write me down

Is the CS50 appliance a big endian or a little endian machine? Little

---

Next, look at the program `layout.c`.

Explain the printout. The integer takes up the 4 bytes starting at addr &carray[4]

Then edit the program, placing the `s` structure into the midst of the character array and assigning some distinct values to the fields. What does the hexdump reveal this time? Was there anything unusual? Can you come up with an explanation for what you observed? You will find it informative to print out the size of the structure (rather than the size of the character array).

Now experiment with the contents and ordering of the structure s; try placing a variety of different types in the structure and rearrange the fields in different orders. In particular, create fields of different sizes and see what happens when you rearrange them. Experiment enough so that you can derive a set of rules or how structures are allocated in memory. Write your rules down.

Next, do the same thing you did experimenting with `struct s` to experiment with the `u` union.

Write me down

The compiler adds pad bytes in structures.

Based on what you observed, state how you believe structs and unions are laid out in C.

See alignment rider.

Summing Up

- The machine does not ascribe any interpretation to the values it stores; it doesn't know if something is intended to be an integer, a character, part of a structure, etc. It is the programming language constructs that give data its meaning.
- Please take a minute to fill out the post-class survey (https://docs.google.com/forms/d/1XaXQuMty0XxoVTcXvuYoHQlmILsosSP4A5ePOztTgdg/viewform) .

Retrieved from "http://cs61.seas.harvard.edu/cs61wiki/index.php?title=2015/representation&oldid=4213"

- This page was last modified on 10 September 2015, at 12:51.
- This page has been accessed 37 times.