CS61C Summer 2025

Precheck: C and Num Rep

Discussion 1

1 Conceptual Introduction

Since computers typically use binary internally whereas we as humans tend to use base 10, we have come up with some representation schemes to easily convert between our human notion of numbers and the computer's. Here are some of the most common ones:

Unsigned Numbers

If we have an n-digit unsigned numeral $d_{n-1}d_{n-2}\cdots d_0$ in *radix* (or *base*) r, then the value of that numeral is

$$\sum_{i=0}^{n-1} r^i d_i,$$

which is just fancy notation to say that instead of a 10's or 100's place we have an r's or r^2's place. For the three radices binary, decimal, and hex, we just let r be 2, 10, and 16, respectively.

Signed Numbers

Unsigned binary numbers work for natural numbers, but many calculations use negative numbers as well. To deal with this, a number of different schemes have been used to represent signed numbers. Here are two common schemes:

Two's Complement:

(a) We can write the value of an n-digit two's complement number as

$$\sum_{i=0}^{n-2} 2^i d_i - 2^{n-1} d_{n-1}.$$

(b) Negative numbers will have a 1 as their most significant bit (MSB). Plugging in $d_{n-1} = 1$ to the formula above gets us

$$\sum_{i=0}^{n-2} 2^i d_i - 2^{n-1}.$$

(c) Meanwhile, positive numbers will have a 0 as their MSB. Plugging in $d_{n-1} = 0$ gets us

$$\sum_{i=0}^{n-2} 2^i d_i,$$

which is very similar to unsigned numbers.

(d) To negate a two's complement number: flip all the bits and add 1.

(e) Addition is exactly the same as with an unsigned number.

(f) Only one 0, and it's located at `0b0`.

Biased Representation:

(a) The number line is shifted so that the smallest number we want to be representable would be `0b0...0`.

(b) To find out what the represented number is, read the representation as if it was an unsigned number, then add the bias.

(c) We can shift to any arbitrary bias we want to suit our needs. To represent (nearly) as much negative numbers as positive, a commonly-used bias for N bits is $-(2^{N-1}-1)$.

2 Precheck: Number Representation

2.1 Depending on the context, the same sequence of bits may represent different things.

True. The same bits can be interpreted in many different ways with the exact same bits! The bits can represent anything from an unsigned number to a signed number or even, as we will cover later, a program. It is all dependent on its agreed upon interpretation.

2.2 It is possible to get an overflow error in Two's Complement when adding numbers of opposite signs.

False. Overflow errors only occur when the correct result of the addition falls outside the range of $\left[-\left(2^{\{n-1\}}\right), 2^{\{n-1\}}-1\right]$. Adding numbers of opposite signs will not result in numbers outside of this range.

2.3 If you interpret a N bit Two's complement number as an unsigned number, negative numbers would be smaller than positive numbers.

False. In Two's Complement, the MSB is always 1 for a negative number. This means EVERY negative number in Two's Complement, when converted to unsigned, will be larger than the positive numbers.

2.4 If you interpret an N bit Bias notation number as an unsigned number (assume there are negative numbers for the given bias), negative numbers would be smaller than positive numbers.

True. In bias notation, we add a bias to the unsigned interpretation to create the value. Regardless of where we 'shift' the range of representable values, the negative numbers, when converted to unsigned, will always stay smaller than the positive numbers. This is unlike Two's Complement (see description above).

2.5 We can represent fractions and decimals in our given number representation formats (unsigned, biased, and Two's Complement).

False. Our current representation formats has a major limitation; we can only represent and do arithmetic with integers. To successfully represent fractional values as well as numbers with extremely high magnitude beyond our current boundaries, we need another representation format.

3 Precheck: Introduction to C

3.1 The correct way of declaring a character array is `char[] array`.

False. The correct way is `char array[]`.

3.2 True or False: C is a pass-by-value language.

True. If you want to pass a reference to anything, you should use a pointer.

3.3 In compiled languages, the compile time is generally pretty fast, however the run-time is significantly slower than interpreted languages.

False. Reasonable compilation time, excellent run-time performance. It optimizes for a given processor type and operating system.

3.4 What is a pointer? What does it have in common with an array variable?

As we like to say, "everything is just bits." A pointer is just a sequence of bits, interpreted as a memory address. An array acts like a pointer to the first element in the allocated memory for that array. However, an array name is not a variable, that is, &arr = arr whereas &ptr != ptr unless some magic happens (what does that mean?).

3.5 If you try to dereference a variable that is not a pointer, what will happen? What about when you free one?

It will treat that variable's underlying bits as if they were a pointer and attempt to access the data there. C will allow you to do almost anything you want, though if you attempt to access an "illegal" memory address, it will segfault for reasons we will learn later in the course. It's why C is not considered "memory safe": you can shoot yourself in the foot if you're not careful. If you free a variable that either has been freed before or was not malloced/calloced/realloced, bad things happen. The behavior is undefined and terminates execution, resulting in an "invalid free" error.

3.6 Memory sectors are defined by the hardware, and cannot be altered.

False. The four major memory sectors, stack, heap, static/data, and text/code for any given process (application) are defined by the operating system and may differ depending on what kind of memory is needed for it to run.

What's an example of a process that might need significant stack space, but very little text, static, and heap space? (Almost any basic deep recursive scheme, since you're making many new function calls on top of each other without closing the previous ones, and thus, stack frames.)

What's an example of a text and static heavy process? (Perhaps a process that is incredibly complicated but has efficient stack usage and does not dynamically allocate memory.)

What's an example of a heap-heavy process? (Maybe if you're using a lot of dynamic memory that the user attempts to access.)