CSE 351 Section 2 – Pointers, Bit Operators, Integers

Pointers

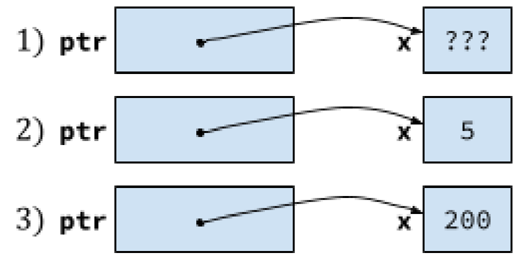
A pointer is a variable that holds an address. C uses pointers explicitly. If we have a variable `x`, then `&x` gives the *address* of `x`, rather than the *value* of `x`. If we have a pointer `p`, then `*p` gives us the value that `p` *points to*, rather than the *value* of `p`.

Consider the following declarations and assignments: **`int`** `x;` **`int*`** `ptr = &x;`

1) We can represent the result of these three lines of code visually as shown. The variable `ptr` stores the address of `x`, and we say "`ptr` points to `x`." `x` currently has an unknown/mystery value since we did not initialize it!

2) After executing `x = 5;`, the memory diagram changes as shown.

3) After executing `*ptr = 200;`, the memory diagram changes as shown. We modified the value of `x` by dereferencing `ptr`.



Pointer Arithmetic

In C, arithmetic on pointers (`++`, `+`, `--`, `-`) is scaled by the size of the data type the pointer points to. That is, if `p` is declared with pointer **`type*`** `p`, then `p + i` will change the value of `p` (an address) by `i*sizeof(`**`type`**`)` (in bytes). If there is a line `*p = *p + 1`, regular arithmetic will apply unless `*p` is also a pointer datatype.

<u>Exercise #1</u>:

Draw out the memory diagram after the sequential execution of each of the lines below:

```
0  int main(int argc, char** argv) {
1    int x = 350;           // assume &x = 0x10, &y = 0x14
2    int y[] = {410, 0};    // p is a pointer to an integer
3    int* p = y;
4    *p = x;
5    p = p + 4;
6    p = &x;
7    x = *p + 1;
8  }
```

Line 1 & 2:	Line 3:	Line 4:
Line 5:	Line 6:	Line 7:

C Bitwise Operators

- **AND** (&) outputs a `1` only when both input bits are `1`s.
- **OR** (|) outputs a `1` when either input bit is a `1`.
- **XOR** (^) outputs a `1` when *exactly one* input bit is a `1`.
- **NOT** (~) outputs the opposite of its input.

```
  1010       1010       1010
& 1100     | 1100     ^ 1100     ~ 1010
------     ------     ------     ------
  1000       1110       0110       0101
```

Masking is very commonly used with bitwise operations. A mask is a binary constant used to manipulate another bit string in a specific manner, such as setting specific bits to `1` or `0`.

Masking Example:

How would you replace the least significant byte of `x` of any length with `0xAA` (*e.g.*, `0x21`**`34`** → `0x21`**`AA`**)?

1) Zero out the least significant byte with an **AND** mask:

```
x = x & ~0xFF;
```

2) Use an **OR** to set the last-significant byte:

```
x = x | 0xAA;
```

> Note that using `~0xFF` (instead of something like `0xFF00`) allows us to handle an x of any length, as required:
>
> If x is 4 bytes, `~0xFF = ~0x000000FF = 0xFFFFFF00`.
>
> If x is 2 bytes, `~0xFF = ~0x00FF = 0xFF00`.

Exercise #2:

[Autumn 2019 Midterm Q1B] If `signed char a = 0x88`, complete the bitwise C statement so that `b = 0xF1`. The first blank should be an *operator*, and the second blank should be a *numeral*.

```
b = a __ 0x______
```

Exercise #3:

Design a series of bitwise operations (*e.g.*, bit masks, bit shifts) to extract the *sign* bit of a `signed int` variable `x`.