

CSE 351 Section 2 – Pointers and Bit Operators

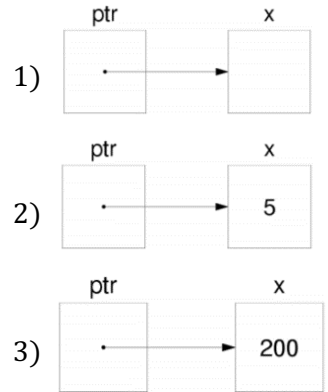
Pointers

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;
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 doesn’t contain a value since we did not assign `x` a value!
- 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). However, `*p` returns the data *pointed at* by `p`, so pointer arithmetic only applies if `p` was a pointer to a pointer.

Exercise:

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

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

Line 1:	<div style="display: flex; justify-content: space-around;"> <div style="text-align: center;">x 410</div> <div style="text-align: center;">y 350</div> </div>
Line 2:	<div style="display: flex; justify-content: space-around;"> <div style="text-align: center;">p 0x10</div> <div style="text-align: center;">x 410</div> </div> <div style="display: flex; justify-content: space-around; margin-top: 10px;"> <div style="text-align: center;">y 350</div> </div>
Line 3:	<div style="display: flex; justify-content: space-around;"> <div style="text-align: center;">p 0x10</div> <div style="text-align: center;">x 350</div> </div> <div style="display: flex; justify-content: space-around; margin-top: 10px;"> <div style="text-align: center;">y 350</div> </div>
Line 4:	<div style="display: flex; justify-content: space-around;"> <div style="text-align: center;">p 0x20</div> <div style="text-align: center;">x 350</div> </div> <div style="display: flex; justify-content: space-around; margin-top: 10px;"> <div style="text-align: center;"></div> <div style="text-align: center;">y 350</div> </div>
Line 5:	<div style="display: flex; justify-content: space-around;"> <div style="text-align: center;">p 0x14</div> <div style="text-align: center;">x 350</div> </div> <div style="display: flex; justify-content: space-around; margin-top: 10px;"> <div style="text-align: center;">y 350</div> </div>
Line 6:	<div style="display: flex; justify-content: space-around;"> <div style="text-align: center;">p 0x14</div> <div style="text-align: center;">x 351</div> </div> <div style="display: flex; justify-content: space-around; margin-top: 10px;"> <div style="text-align: center;">y 350</div> </div>

C Bitwise Operators

&	0	1
0	0	0
1	0	1

 ← **AND (&)** outputs a 1 only when both input bits are 1.

 	0	1
0	0	1
1	1	1

 → **OR (|)** outputs a 1 when either input bit is 1.

^	0	1
0	0	1
1	1	0

 ← **XOR (^)** outputs a 1 when either input is *exclusively* 1.

~	
0	1
1	0

 → **NOT (~)** outputs the opposite of its input.

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.

Exercises:

- 1) What happens when we fix/set one of the inputs to the 2-input gates? Let x be the other input. Fill in the following blanks with either 0, 1, x , or \bar{x} (NOT x):

$$x \ \& \ 0 = \underline{0}$$

$$x \ | \ 0 = \underline{x}$$

$$x \ ^ \ 0 = \underline{x}$$

$$x \ \& \ 1 = \underline{x}$$

$$x \ | \ 1 = \underline{1}$$

$$x \ ^ \ 1 = \underline{\bar{x}}$$

- 2) **Lab 1 Helper Exercises:** Lab 1 is intended to familiarize you with bitwise operations in C through a series of puzzles. These exercises are either sub-problems directly from the lab or expose concepts needed to complete the lab. Start early!

Bit Extraction: Returns the value (0 or 1) of the 19th bit (counting from LSB). Allowed operators: \gg , $\&$, $|$, \sim .

```
int extract19(int x) {
    return (x >> 18) & 0x1;
}
```

Subtraction: Returns the value of $x-y$. Allowed operators: \gg , $\&$, $|$, \sim , $+$.

```
int subtract(int x, int y) {
    return x + ((~y) + 1);
}
```

Equality: Returns the value of $x==y$. Allowed operators: \gg , $\&$, $|$, \sim , $+$, \wedge , $!$.

```
int equals(int x, int y) {
    return !(x ^ y);
}
```

Divisible by Eight? Returns the value of $(x\%8)==0$. Allowed operators: \gg , \ll , $\&$, $|$, \sim , $+$, \wedge , $!$.

```
int divisible_by_8(int x) {
    return !(x << 29);
}
```

Greater than Zero? Returns the value of $x>0$. Allowed operators: \gg , $\&$, $|$, \sim , $+$, \wedge , $!$.

```
int greater_than_0(int x) {
    /* invert and check sign; we need the third operand for the T_min case */
    return ((~x + 1) >> 31) & 0x1 & ~(x >> 31) _OR_ !!x & ~(x >> 31);
}
```