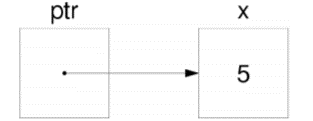
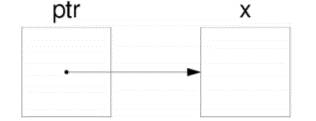
**CSE 351 Section 2 – Pointers, Bit Operators, Integers**

**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.

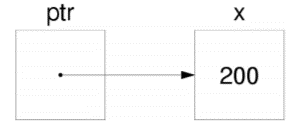
Consider the following declarations and assignments:



1)

2)

3)



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). If there is a line \*p = \*p + 1, regular arithmetic will apply unless \*p is also a pointer datatype.

**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: | Line 2: | Line 3: |
| Line 4: | Line 5: | Line 6: |

**C Bitwise Operators**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **&** | 0 | 1 |  | **AND** (&) outputs a 1 only when both input bits are 1. |  | **|** | 0 | 1 |
| 0 | **0** | **0** |  |  |  | 0 | **0** | **1** |
| 1 | **0** | **1** |  | **OR** (|) outputs a 1 when either input bit is 1. |  | 1 | **1** | **1** |
|  |  |  |  |  |  |  |  |  |
| **^** | 0 | 1 |  | **XOR** (^) outputs a 1 when either input is *exclusively* 1. |  | **~** |  |  |
| 0 | **0** | **1** |  |  |  | 0 | **1** |  |
| 1 | **1** | **0** |  | **NOT** (~) outputs the opposite of its input. |  | 1 | **0** |  |

*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. [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\_\_

2) Implement the following C function using control structures and bitwise operators.

// returns the number of pairs of bits that are the

// opposite of each other (i.e. 0 and 1 or 1 and 0)

//

// bits are "paired" by taking adjacent bits

// starting at the lsb (0) and pairs do not overlap.

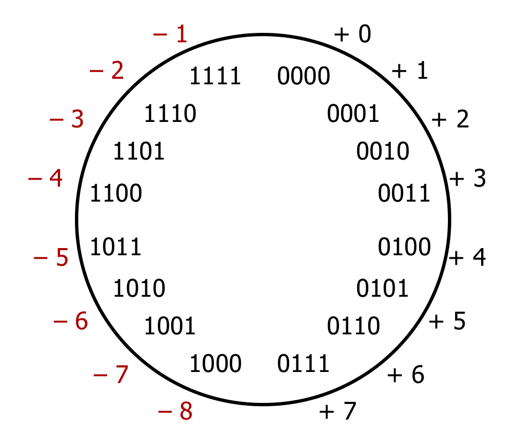
// For example, there are 16 distinct pairs in a 32-bit integer

**int num\_pairs\_opposite(int x) {**

**}**

**Signed Integers with Two’s Complement**

Two’s complement is the standard for representing signed integers:

* The most significant bit (MSB) has a negative value; all others have positive values (same as unsigned)
* Binary addition is performed the same way for signed and unsigned
* The bit representation for the negative value (additive inverse) of a Two’s Complement number can be found by:

flipping all the bits and adding 1 (i.e. ).

The “number wheel” showing the relationship between 4-bit numerals and their Two’s Complement interpretations is shown on the right:

* The largest number is 7 whereas the smallest number is -8
* There is a nice symmetry between numbers and their negative counterparts except for -8

**Exercises:**

1. If we have 8 bits to represent integers, answer the following questions:
   1. What is the **largest integer**? The **largest integer + 1**? The most **negative integer**? If it doesn’t apply, write n/a.

|  |  |
| --- | --- |
| Unsigned:  **Largest:**  **Largest + 1:**  **Most Negative:** | Two’s Complement:  **Largest:**  **Largest + 1:**  **Most Negative:** |

* 1. How do you represent (if possible) the following numbers: **39**, **-39**, **127**?

|  |  |
| --- | --- |
| Unsigned:  39:  -39:  127: | Two’s Complement:  39:  -39:  127: |

1. [Autumn 2017 Final M1A] Take the 32-bit numeral 0xC0800000. Circle the number representation below that has the most negative value for this numeral.

Sign & Magnitude Two’s Complement Unsigned

1. [Winter 2018 Midterm 1C] Given the 4-bit bit vector 0b1101, what is its value in decimal (base 10)? Circle your answer.

13 -3 -5 Undefined