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:

\[
\begin{align*}
\text{int } &x; \\
\text{int } &\*\text{ptr; }
\end{align*}
\]

\[
\text{ptr} = \&x;
\]

1) We can represent the result of these three lines of code visually as shown. The variable \( \text{ptr} \) stores the address of \( x \), and we say "\( \text{ptr} \) 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 \( \*\text{ptr} = 200; \), the memory diagram changes as shown. We modified the value of \( x \) by dereferencing \( \text{ptr} \).

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 \( \text{type* } p \), then \( p + i \) will change the value of \( p \) (an address) by \( i \times \text{sizeof(\text{type})} \) (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:

\[
\begin{align*}
\text{int } &\text{main(int argc, char **argv) } \\
&\{ \\
&\text{int } x = 410, y = 351; \quad \text{// assume } \&x = 0x10, \&y = 0x18 \\
&\text{int } *p = &x; \quad \text{// } p \text{ is a pointer to an integer} \\
&*p = y; \\
p = p + 4; \\
p = &y; \\
x = *p + 1; \\
&\}
\end{align*}
\]

<table>
<thead>
<tr>
<th>Line 1:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line 2:</td>
</tr>
<tr>
<td>Line 3:</td>
</tr>
<tr>
<td>Line 4:</td>
</tr>
<tr>
<td>Line 5:</td>
</tr>
<tr>
<td>Line 6:</td>
</tr>
</tbody>
</table>
C Bitwise Operators

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

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 binary bitwise operators? Let x be the other input. Fill in the following blanks with either 0, 1, x, or x̅ (NOT x):

\[
\begin{align*}
x \& 0 &= \underline{\quad} \\
x \mid 0 &= \underline{\quad} \\
x \wedge 0 &= \underline{\quad} \\
x \& 1 &= \underline{\quad} \\
x \mid 1 &= \underline{\quad} \\
x \wedge 1 &= \underline{\quad}
\end{align*}
\]

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: >>, &, |, ~.

```c
int extract19(int x) {
    return
}
```

**Subtraction:** Returns the value of x–y. Allowed operators: >>, &, |, ~, +.

```c
int subtract(int x, int y) {
    return
}
```

**Equality:** Returns the value of x==y. Allowed operators: >>, &, |, ~, +, ^, !.

```c
int equals(int x, int y) {
    return
}
```

**Divisible by Eight?** Returns the value of (x%8)==0. Allowed operators: >>, <<, &, |, ~, +, ^, !.

```c
int divisible_by_8(int x) {
    return
}
```

**Greater than Zero?** Returns the value of x>0. Allowed operators: >>, &, |, ~, +, ^, !.

```c
int greater_than_0(int x) {
    return
}
```