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:

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

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

<table>
<thead>
<tr>
<th>Line 1:</th>
<th>Line 2:</th>
<th>Line 3:</th>
</tr>
</thead>
<tbody>
<tr>
<td>unknown</td>
<td>What is the value at ( p ) before line 2</td>
<td>350</td>
</tr>
<tr>
<td></td>
<td>0x10</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Line 4:</th>
<th>Line 5:</th>
<th>Line 6:</th>
</tr>
</thead>
<tbody>
<tr>
<td>What is the value at ( p ) after line 4</td>
<td>( y = 350 )</td>
<td>What is the value of ( x ) after line 6</td>
</tr>
<tr>
<td>0x20</td>
<td></td>
<td>351</td>
</tr>
</tbody>
</table>
C Bitwise Operators

<table>
<thead>
<tr>
<th>&amp; 0 1</th>
<th>← AND(&amp;) outputs a 1 only when both input bits are 1.</th>
<th></th>
<th>0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>1 0 1</td>
<td>OR (|) outputs a 1 when either input bit is 1.</td>
<td>→ 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>^ 0 1</th>
<th>← XOR (^) outputs a 1 when either input is exclusively 1.</th>
<th>~</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1</td>
<td>1 1 0</td>
<td>NOT (~) outputs the opposite of its input.</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 2-input gates? Let \( x \) be the other input.
   Fill in the following blanks with either 0, 1, \( x \), or \( \bar{x} \) (NOT \( x \)):

   \[
   \begin{align*}
   x \& 0 &= \ \_\_\_\_ \quad & x \| 0 &= \ \_\_\_\_ \quad & x \^\ 0 &= \ \_\_\_\_ \\
   x \& 1 &= \ \_\_\_\_ \quad & x \| 1 &= \ \_\_\_\_ \quad & x \^\ 1 &= \ \_\_\_\_ \\
   \end{align*}
   \]

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 Answer : (x >> 18) & 0x1;
}
```

**Subtraction:** Returns the value of \( x - y \). Allowed operators: >>, &, |, ~, +.

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

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

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

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

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

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

```c
int greater_than_0(int x) {
    return (~x + 1) >> 31) & 0x1, !x & ~(x >> 31);, ~(x >> 31)
}
```