Pointers

In C, it is possible for one variable to hold the address of another. This type of variable is called a pointer.

Recall that in LC-3 if we had:
```
DATA .FILL x1234
```
then there were three ways to access that data:
```
LD R0, DATA ; R0 holds the value x1234
LEA R1, DATA ; R1 holds the address labeled “DATA”
LDI R2, DATA ; the value at “DATA” is an address;
              ; R2 holds the value at address x1234
```
We say that a pointer “points” to the variables whose address it contains.

```
int x = 5;
int *ptr = &x;
```
We refer to “&x” as “the address of x”.

If those were local variables in a function, the stack would look like this:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>x3040</td>
<td>x3041</td>
<td>ptr stored here</td>
</tr>
<tr>
<td>x3041</td>
<td>5</td>
<td>x stored here</td>
</tr>
</tbody>
</table>

If x were the first variable declared in the function, then in the symbol table, the offset of x would be 0, and the offset of ptr would be -1.

To access the value pointed to by the pointer, we dereference the pointer.

```
*ptr = 11; // this sets x to 11
```

In LC-3, assuming R0 holds the value 11, this would be:
```
LDR R1, FP, #-1 ; R1 holds the value of ptr,    ; i.e. the address of x
STR R0, R1, #0 ; now write 11 to that address
```
In this case, it is safe to say that *ptr and x refer to the same spot in memory:
```
x = 9;          // now *ptr is 9, too
*ptr = *ptr + 1; // what is x?
```
Be careful about expressions like `ptr++` and `*ptr++`. The first one increments the value of the pointer, and not that of the thing it points to. The second one is the same as `*(ptr++)`, and not `(*ptr)++`, which perhaps is what you had in mind.

**Passing variables by reference**

Previously, we saw that variables of primitive types (ints, chars, etc.) are passed by value when they are arguments to functions. Recall the example:

```c
#include <stdlib.h>
#include <stdio.h>

int swap(int, int);

main() {
    int a = 5, b = 9;
    swap(a, b);
    printf("now a is %d\n", a);
}

void swap(int x, int y) {
    int temp = x;
    x = y;
    y = temp;
}
```

In this case, `a` is still 5 when it's printed on line 11, because its value is passed to the function `swap`. This is clear if we look at what the stack looks like when `swap` is called:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>x3040</td>
<td>??</td>
<td>temp</td>
</tr>
<tr>
<td>x3041</td>
<td>x3047</td>
<td>FP_main</td>
</tr>
<tr>
<td>x3042</td>
<td>...</td>
<td>RA_main</td>
</tr>
<tr>
<td>x3043</td>
<td>??</td>
<td>RV_main</td>
</tr>
<tr>
<td>x3044</td>
<td>5</td>
<td>x</td>
</tr>
<tr>
<td>x3045</td>
<td>9</td>
<td>y</td>
</tr>
<tr>
<td>x3046</td>
<td>9</td>
<td>b</td>
</tr>
<tr>
<td>x3047</td>
<td>5</td>
<td>a</td>
</tr>
</tbody>
</table>
Recall that, right before \texttt{main} calls "JSR SWAP" it pushes the \textit{values} of the arguments onto the stack. Thus, when \texttt{swap} executes lines 16 and 17, the values at addresses \texttt{x3044} and \texttt{x3045} (\texttt{x} and \texttt{y}, respectively) are the ones that get changed, not the values at \texttt{x3047} and \texttt{x3046} (\texttt{a} and \texttt{b}, respectively).

So what if we wanted to change the values of \texttt{a} and \texttt{b}? Then we would pass them by reference, i.e. we would pass pointers to \texttt{a} and \texttt{b} as the arguments.

\begin{verbatim}
void swap(int *x, int *y) {
    int temp = *x;
    *x = *y;
    *y = temp;
}
\end{verbatim}

Now \texttt{a} and \texttt{b} are no longer ints, but rather pointers to ints, i.e. variables that hold the address of an int. And we can access the values pointed to by those variables by dereferencing them.

To call this function, we would do this:

\begin{verbatim}
int a = 5, b = 9;
int *pa = &a;
int *pb = &b;
swap(pa, pb);
// now a is 9 and b is 5
\end{verbatim}

This is what the stack would like like when \texttt{swap(pa, pb)} is called:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>x3040</td>
<td>??</td>
<td>temp</td>
</tr>
<tr>
<td>x3041</td>
<td>x3047</td>
<td>FP_main</td>
</tr>
<tr>
<td>x3042</td>
<td>...</td>
<td>RA_main</td>
</tr>
<tr>
<td>x3043</td>
<td>??</td>
<td>RV_main</td>
</tr>
<tr>
<td>x3044</td>
<td>x3049</td>
<td>x</td>
</tr>
<tr>
<td>x3045</td>
<td>x3048</td>
<td>y</td>
</tr>
<tr>
<td>x3046</td>
<td>x3048</td>
<td>pb</td>
</tr>
<tr>
<td>x3047</td>
<td>x3049</td>
<td>pa</td>
</tr>
<tr>
<td>x3048</td>
<td>9</td>
<td>b</td>
</tr>
<tr>
<td>x3049</td>
<td>5</td>
<td>a</td>
</tr>
</tbody>
</table>
One way to think of it is that the value of \( p \) is passed to \( \text{swap} \) and is stored as parameter \( x \), but that value is actually an address. That's why when we change \( *x \), we're not changing what's held at address \( x3044 \), we're changing what's held at \( x3049 \), which is \( a \).

Note that we do not call \( \text{swap}(\*p, \*pb) \); that would be exactly the same as calling \( \text{swap}(a, b) \), which is not what we want.

Note also that we could simply do this:

```c
int a = 5, b = 9;
swap(&a, &b);
// now a is 9 and b is 5
```

This is a more common way of calling the function, and one that you've seen before: \( \text{scanf}! \)

**Pointers and Arrays**

We just saw that variables can be passed by reference if you use a pointer, and last time we also mentioned that arrays are passed by reference. Hmmm... does that mean arrays are pointers? Sort of!

The variable used to name an array is a pointer to its first element. For instance:

```c
int a[] = { 10, 20, 30, 40 };
```

When we pass \( a \) as an argument to a function, we're actually passing the address of the first element, i.e. \( \&a[0] \).

In a lot of cases, there's really no difference between an array name and a pointer:

```c
int a[] = { 10, 20, 30, 40 };
int *p;
p = a; // same as p = &a[0]; *NOT* p = &a
```

In this case, here's what the stack would look like:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>x3040</td>
<td>x3041</td>
<td>( p ) stored here</td>
</tr>
<tr>
<td>x3041</td>
<td>10</td>
<td>( a[0] ) stored here</td>
</tr>
<tr>
<td>x3042</td>
<td>20</td>
<td>( a[1] ) stored here</td>
</tr>
<tr>
<td>x3043</td>
<td>30</td>
<td>( a[2] ) stored here</td>
</tr>
<tr>
<td>x3044</td>
<td>40</td>
<td>( a[3] ) stored here</td>
</tr>
</tbody>
</table>
Now you can do this:

```c
int x = *p; // same as x = a[0]; x is 10
int y = *(p + 1); // same as y = a[1]; y is 20
int z = *p + 1; // same as z = a[0] + 1; z is 11
```

Note that `a` does not actually exist on the stack, and that we can't assign a value to it. For instance, we can't do `a = 14`. Rather, `a` only exists as an offset in the symbol table.

Note also that, in the above example, all of these refer to the same value (20):

```
a[1]
*(p + 1)
p[1]
*(a + 1)
1[a]
```

The first two are considered “normal”. The third and fourth, while technically correct, are a bit misleading and are not considered good style. The last one is just plain weird.

One last thing: be careful when someone (like, on a job interview) asks you “are pointers and arrays the same thing?” They're not! A pointer takes up exactly one memory address: it is a single value representing the address of another variable. An array takes up multiple memory addresses: one for each element in the array.

---

**Pointers and Strings**

Recall from last time that a string in C is just a null-terminated char array. So if we have this:

```c
char word[] = “hello”;
```

then we could also do this:

```c
char *p;
p = word;
char c = *(p + 1); // c is the char 'e'
```

Another way to declare a string is like this:

```c
char *werd = “howdy”;
```

In this case, `werd` is a pointer to a char, specifically to the first char of the string (which would be 'h'). That is, it holds the address of the char 'h' in that string.

Where is that string? It's *not* on the stack. It's actually part of the program text, and on most UNIX platforms, that string would be immutable (meaning: you can't change it).

So this would be okay:
char c = *(werd + 1); // c is the char 'o'

but this would not:
*(werd + 1) = 'a';

**Arrays of Pointers and Pointers to Pointers**

Can you have an array of pointers? Sure.
Can you have a pointer to a pointer? Definitely.

Let's take a look at this example:

```c
int odd[] = {1, 3, 5, 7, 9};
int even[] = {2, 4, 6, 8};
int *a[] = {odd, even}; // array of pointers!
ton *pa; // pointer to pointer!
pa = a; // or pa = &a[0] but NOT pa = &a
```

Let's see what this looks like on the stack:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>x3040</td>
<td>x3041</td>
<td>pa stored here</td>
</tr>
<tr>
<td>x3041</td>
<td>x3047</td>
<td>a[0] stored here</td>
</tr>
<tr>
<td>x3042</td>
<td>x3043</td>
<td>a[1] stored here</td>
</tr>
<tr>
<td>x3043</td>
<td>2</td>
<td>even[0] stored here</td>
</tr>
<tr>
<td>x3044</td>
<td>4</td>
<td>even[1] stored here</td>
</tr>
<tr>
<td>x3045</td>
<td>6</td>
<td>even[2] stored here</td>
</tr>
<tr>
<td>x3046</td>
<td>8</td>
<td>even[3] stored here</td>
</tr>
<tr>
<td>x3047</td>
<td>1</td>
<td>odd[0] stored here</td>
</tr>
<tr>
<td>x3048</td>
<td>3</td>
<td>odd[1] stored here</td>
</tr>
<tr>
<td>x3049</td>
<td>5</td>
<td>odd[2] stored here</td>
</tr>
<tr>
<td>x304A</td>
<td>7</td>
<td>odd[3] stored here</td>
</tr>
<tr>
<td>x304B</td>
<td>9</td>
<td>odd[4] stored here</td>
</tr>
</tbody>
</table>

It's important to note that a[0] and a[1] are pointers, specifically to the first element of the odd and even arrays, respectively.

Likewise, pa is a pointer, specifically to the array a's first element (which, itself, is a pointer).
To put this all together, let's consider that if we use 2d-array notation, then we see that \[a[1][3]\] is the value 8. How does that work?

Well, since \(p_a = a\), we can call this \(p_a[1][3]\). And it we use pointer notation, then we'd have \(*(*(p_a + 1) + 3)\).

And if we do a bit of math:
\[
*(*(p_a + 1) + 3) = \\
*(*(x3041 + 1) + 3) = \\
*(*(x3042) + 3) = \\
*(x3043 + 3) = \\
*(x3046) = \\
8
\]
w00t!!