Pointers

Generally, when we have been using variables in our programs, we use a declared variable name, like `theThing`. The variable holds a particular kind of value (integer, double, char, etc.) and we assign a value in the program, and change the value as needed. We haven't worried about where the quantity is stored in memory — we let the compiler and the operating system take care of that.

However, sometimes it is useful to know where in the memory the variable is stored. This is particularly true when working with blocks of data. When we refer to the *location* of a stored value, we are using a *pointer* to the memory location.

memory	`name`	value
87		
88		
89		
90	`theThing`	42
91		
92		
93		
94		
95		
96		
97		

Pointers: declaring

A pointer is a variable that contains an address to a memory location. We can access that memory location using the pointer. There are not generic pointers — each defined pointer must be associated with a specific data type.

To declare a pointer variable, put an asterisk after the data type. For example

```
int* xPtr;
```

defines a pointer to a chunk of memory that is the size of an integer. And

```
double* zPtr;
```

defines a pointer to a chunk of memory that is the size of an double.

(You can also put the put the asterisk before the variable name, like

```
int *xPtr;

double *zPtr;
```

I sometimes use this, but the usual convention is put the asterisk after the data type.)

Pointers: `NULL`

In the program, you will have variables names xPtr and zPtr. No matter what data type the pointer is associated, the pointer itself is must an integer. (It might be very big, but it is still an integer.)

It is a good idea (though not essential) to initialize pointers when they are declared. The keyword NULL is used to indicate pointers that don't point to anything.

```
int* xPtr = NULL;
double* zPtr = NULL;
```

Pointers: de-referencing

To access the data that is stored in memory location pointed to by a pointer, you add an asterisk in front of (pre-pend) the pointer. For instance if `xPtr` points to a memory location that has the integer 17 stored in and `zPtr` points to a memory location that has the double –72.963 stored in it, the commands

```
x = *xPtr;
z = *zPtr;
```

will put 17 into `x` and –72.963 into `z`, where `x` is a "regular" integer variable and `z` is a "regular" double variable.

Again: `xPtr` is a memory location `*xPtr` is what is stored at the memory location. (You will probably need to repeat this little mantra over and over as you are trying learn how to use pointers.)

Pointers: &

If you have a regular variable, and you need to use the address for it, then add (prepend) an ampersand in front of the variable. For example if **`x`** is a regular integer variable, then **`&x`** is the address where the integer is stored. **`&x`** is a pointer. And so if we have declared variables **`x`** and **`y`** (of any type) and a pointer **`xPtr`** that points to the corresponding type, then we could write the following:

```
x = 42;

xPtr = &x;

y = *xPtr
```

The first line assigns the integer 42 to the "regular" variable **`x`**.

The second line assigns the address of **`x`** to **`xPtr`**.

The third line assigns the value that is at the address pointed to by **`xPtr`** to the "regular" variable **`y`**. So **`y`** will be 42.

Printing out the value of pointers

Generally, we do not want to print (or even know) the value of pointer is. But while learning how to work with pointers or in debugging, it is sometimes helpful to print the pointer value. This can clarify make clear the difference between the pointer (an address) and what it is pointing to (some sort of data stored at the address).

To print the pointer in `printf`, use `"%p"` in the formatting string, and then typecast the pointer variable to `(void*)`. (Yes, it's weird. But just do it. If you will neglect the typecasting, the address will probably still print OK, but using the typecasting should assure that everything is kosher.)

See the short program on the next slide.

```c
//EE 285 - fun with pointers

#include <stdio.h>

int main(void){

   int x, y;
   int* xPtr = NULL;

   x = 42;
   xPtr = &x;
   y = *xPtr;

   printf( "x = %d, y = %d, xPtr = %p.", x, y, (void*)xPtr );

   printf( "\n\n" );
   return 0;
}
```

```
x = 42, y = 42, xPtr = 0x7ffeefbff478.

Program ended with exit code: 0
```

The address is an integer printed in hexadecimal form. (The 0x in front indicates hex format.)

```c
//EE 285 - more fun with pointers

#include <stdio.h>

int main(void){

   double burp = 0;    //a standard double
   double* burpPtr;    //a pointer to a double

   burpPtr = &burp;    //burpPtr points to burp

   printf( "Enter the value: " );
   scanf( "%lf", burpPtr );       //no & !!

   printf( "\nburp %4.3lf and its address is %p.", *burpPtr, (void*)burpPtr );

   printf( "\n\n" );
   return 0;
}
```

```
Enter the value: -19.41

burp = -19.410 and its address is 0x7ffeefbff470.

Program ended with exit code: 0
```

```c
//EE 285 - still more fun with pointers

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(void){

   int anArray[8];   //an array of integers
   int* intPtr;     //a pointer to a integer
   int i;

   srand( (int)time(0) );

   for( i = 0; i < 8; i++ ){
      anArray[i] = rand()%10 + 1;
      intPtr = &anArray[i];
      printf( "%d: %d at %p.\n", i, *intPtr, (void*)intPtr );
    }

   printf( "\n\n" );
   return 0;
}
```

```
0: 7 at 0x7ffeefbff450.
1: 5 at 0x7ffeefbff454.
2: 9 at 0x7ffeefbff458.
3: 1 at 0x7ffeefbff45c.
4: 5 at 0x7ffeefbff460.
5: 7 at 0x7ffeefbff464.
6: 3 at 0x7ffeefbff468.
7: 9 at 0x7ffeefbff46c.

Program ended with exit code: 0
```

Math with pointers

At the most basic level, pointers are just integers, so we could, in principle, do any sort of math with them. However, because the pointer integer represents a chunk of memory, there is very little math that makes sense. Why would you want the product of memory locations? Or even the sum? Taking the cosine or the square-root of memory address is meaningless.

The only useful math that we might do with pointers is to add or subtract a specific value to a pointer. For example:

`xPtr++;` Increment the pointer value — go to the next memory location. (Also `++xPtr;`)

`xPtr--;` Decrement the pointer value — go to the previous memory location. (Also `--xPtr;`)

```
xPtr = xPtr + 42;
```
Move ahead 42 memory locations. (Also `xPtr += 42;`)

```
xPtr = xPtr - 17;
```
Move back 17 memory locations. (Also `xPtr -= 17;`)

It is a good idea to use parentheses when working with pointers, so that everything is dereferenced properly.

`aPtr + 1` → points to the next memory location just after `aPtr`.

`*(aPtr + 1)` → refers to value stored in the next memory location just after `aPtr`

`aPtr + n` → points to n memory locations beyond `aPtr`.

`*(aPtr + n)` → refers to value stored in nth memory location beyond `aPtr.`

`aPtr++` → increments `aPtr` to point to the next memory location.

`*(aPtr++)` → refers to the value pointed by to `aPtr`, and then increments `aPtr` to the next memory location. (Note the subtle difference with `*(++aPtr)`

Note that when we "increment" a pointer, we are not simply adding 1 to it's previous value. The increment is by one memory location. Since different types of variables require different chunks of memory, the true size of the increment is relative to the type of variable being referenced. The compiler takes care of all of these details. Values can be subtracted and the decrement operator works similarly.

If you have been following along, you probably have an important question: Why bother with pointers? If the regular variable names gives you access to the value stored at the particular memory location, then what value is there in using a pointer to access the value stored at a particular memory location?

There are two reasons: arrays and functions. (And even better is the combination of arrays and functions.)

To get a hint of the value of pointers with arrays, look back at the memory addresses printed in the previous program. Each integer uses four bytes (32 bits) in memory. The elements of the array are stored in consecutive 4 bytes chunks of memory. (Note the memory locations are listed in hex format.) If we know where the first element of the array is stored, we can access all of the remaining elements based off of the memory location of the first element. In fact, this is exactly what is happening when we work with arrays.