Chapter 10
Pointers

10.1 Chapter Overview

When C was developed, computers were much less powerful and had much less memory to work with. The ability to work directly with particular memory locations was beneficial. Pointers in C allow you to effectively represent complex data structures, to change values passed as arguments to functions, to work with memory that has been dynamically allocated (Chapter 16), and to more concisely and efficiently deal with arrays.

10.2 Defining a Pointer Variable

Suppose you define an integer called `count` as follows:

```
int count = 10;
```

In memory, space was reserved for an integer, the name of this space (the variable) is called `count`, and the value of 10 was placed into that memory space.

You can declare a pointer and have it point to that location in memory (the variable called `count`) as follows:

```
int *ptr;
ptr = &count;
```

The asterisk (`*`) in the declaration defines to the system that `ptr` is to be a `pointer` and the `int` says it will point to something of type `int`. The address operator (`&`) in the second line refers to the address of the item, in this case, the address of `count`, which is of type `int`. So now `ptr` points to the location in memory called `count` which has the value of 10 in it.

To change the value that `ptr` is pointing to, you can write the following:

```
*ptr = 25;
```

which would have had the same effect as if you had done this:

```
count = 25;
```

In either case, the value at that memory location was changed to 25. In the first case, where it says

```
*ptr = 25;
```

`ptr` is **dereferenced** by using the asterisk (`*`) because a direct assignment to a pointer variable will change the address of the pointer, *not* the value at the address that it is already pointing to. So, to access the value of what a pointer is pointing to, you have to **dereference** it using the asterisk (`*`). The same thing is done if you have another integer variable that you want the value of to become the same as what the pointer is pointing to, as in the following:

```
int anotherInteger;
anotherInteger = *ptr;
```

So now, `anotherInteger` has the value of whatever `ptr` is pointing to, which at this point, it is still pointing to `count`, so it **indirectly** has taken on the same value as `count`.

10.3 Pointers and Arrays

One of the most common uses of pointers in C is as pointers to arrays. Pointers to arrays generally result in code that uses less memory and executes faster.

If you have an array of 100 integers, declared as the following:

```
int values[100];
```

`values` – the name of the array, is also the location in memory where the array is located, and to be more specific, it points to the first item in the array. That is why when you send an array as a parameter to a function, whatever changes that are made to the array argument in the function are also made to the original array that was sent to it. Also, the expression `values[i]` refers to the i^{th} element in the array called `values`.

You can declare a pointer to an array, and if the array is an array of integers, you would declare the pointer like the following:

```
int *valuesPtr;
```

To set `valuesPtr` to point to the first element in the array `values`, you would do the following:

```
valuesPtr = values;
```

The address operator (`&`) is not used in this case because the compiler treats the appearance of an array name without a subscript as a pointer to the array (or more specifically, to the first item of the array). The following is equivalent to the above statement – it will set `valuesPtr` to point to the address of the first item in the array called `values`:

```
valuesPtr = &values[0];
```

So, the value of the pointer can be initialized either way.

The real power of using a pointer to an array comes into play when you want to sequence through the elements of the array. If `valuesPtr` is set to point to the first item of the array (as was done above), the expression `*valuesPtr` can be used to access the first integer of the array (same things as using `values[0]`). To access `values[3]` with the pointer variable, you could add 3 to the `valuesPtr` and then apply the indirection operator, as with the following: `*(valuesPtr +3)`. So, to set `values[10]` to 27, you could do the following:

```
values[10] = 27;
```

Or, using the `valuesPtr`, you could write:

```
*(valuesPtr + 10) = 27;
```

If you want to set the value of `valuesPtr` to point to the second element of the array `values`, you can apply the address operator to `values[1]` and assign it to `valuesPtr`:

```
valuesPtr = &values[1];
```

or, if `valuesPtr` points to `values[0]`, you can set it to point to `values[1]` by simply adding 1 to the value of `valuesPtr`:

```
valuesPtr += 1;
```

or

```
++valuesPtr;
```

could have also been used to increment the pointer; pointers can be decremented as well

So, back to the array `values`, having 100 elements – to check for the end of the array, you could write:

```
if (valuesPtr == &values[99])
```

which would return TRUE (nonzero) if `valuesPtr` is pointing to the last element, and FALSE (zero) otherwise. The above expression is the same as:

```
if (valuesPtr == values + 99)
```

10.4 Pointers and Functions

Pointers can be passed to functions the way anything else is passed to a function. Functions can also return a pointer as its result. The function would have in its list of arguments, a pointer, such as:

```
void test (int *intPtr) {
     *intPtr = 100;
}
```

and a pointer can be passed to it, like the following:

```
int main (void) {
     int i = 50, *p = &i;
     …
     test (p);
     …
}
```

The value of the pointer sent to the function is copied into the formal parameter (`intPtr` in the above example) when the function is called. Therefore, any change made to the formal parameter by the function does *NOT* affect the pointer that was passed to the function, but the data elements that the pointer references *CAN* be changed. So, in the above example, the pointer `p` is still pointing to `i` – that doesn't change; but the value of what `p` points to was changed when `intPtr` was assigned the value of 100 (and therefore, the value of `i` also changed to 100).

10.5 Command-Line Arguments (from Chapter 16, pages 380-384)

Arguments can be passed to a program upon execution of the program. For example, in class I showed the Unix utility program called `grep` which will search for a word (or phrase) specified at the command-line in the file(s) also specified at the command-line, such as:

```
grep gameChoice *.c
```

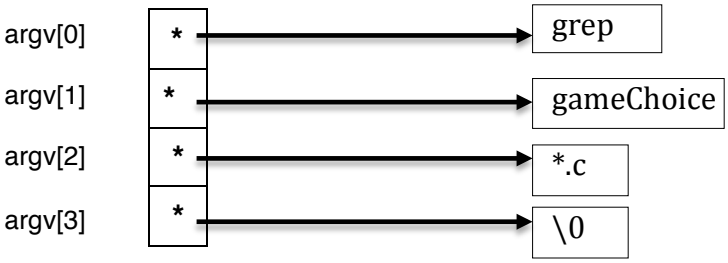
where `grep` is the name of the executable file (the program). The `*.c` specifies that all the `.c` files in the current directory are to be searched for the word `gameChoice`.

When command-line arguments are passed to a program, the main function definition looks like the following:

```
int main (int argc, char *argv[]) {
```

where `argc` (***arg***ument ***c***ount) is an integer that takes on the value of however many arguments were passed in from the command-line; and `argv` (***arg***ument ***v***ector) is the second argument, which is an array of character pointers. The first pointer in `argv` points to the name of the program that was typed in on the command-line. The remaining pointers point to each of the other arguments that were passed in. Then the last pointer in the `argv` array is defined to be `null`. So, there are always `argc + 1` pointers in the `argv` array.

With the above `grep` example, `argc = 3` and `argv` would look like the following:



10.6 `sscanf()`

`sscanf()` is a string `scanf()`. We've used `scanf()` for scanning input from the terminal; `sscanf()` scans an array (or string) already in memory and reads each item into a pointer that points to the address of a variable (just like `scanf()`) according to the format. The function prototype is as follows:

```
int sscanf  ( buffer, format(s), arg(s) )
```

where:

1. buffer is the given array (or string),
2. the format(s) are your `%i`, `%d`, `%f`, etc – an appropriate one for each argument,
3. and the arguments are pointers to the address of each argument, (uses the address operator, just like with `scanf()` unless it is an array name, which is already pointing to an item in memory)

An example follows:

```
int numberOfArguments = 0;
numberOfArguments = sscanf ( my_array, "%d %d %d", &int1, &int2, &int3 );
```

`sscanf()` is used frequently with command-line programs because the arguments from the command-line are put into an array, and then each one can be scanned from the array and accessed (pointed to by a variable that is declared in the program).

```
/*
 * This program illustrates the use of command line
 * parameters and sscanf.
 *
 * Sample run is   ./a.out John 7 12
 */

#include <stdio.h>


// argc is the number of command line arguments
// argv[] contains the arguments themselves
int main(int argc, char * argv[]) {
    int num1 = 5, num2;

    printf("I have %d arguments\n", argc);
    printf("program executable is %s\n", argv[0]);

    printf("Hi there %s\n", argv[1]);

    // read the arguments from memory and store them in variables
    sscanf(argv[2], "%d", &num1);  // 3rd command-line argument
                                   // num1 should change from 5 to whatever
                                   //    the user entered
    sscanf(argv[3], "%d", &num2);  // 4th command-line argument

    printf("%d + %d = %d\n", num 1, num 2, num 1 + num 2);

    return 0;
}
```

10.7 Pointers to Character Strings

Instead of sending a string (character array) to a function, you can send a pointer to the character string.
See program 10.13 page 265 for an example of a pointer version of copyString function…

10.8 Constant Character Strings and Pointers

Note that when a string literal, or constant character string, is passed to a function, it is a pointer to that string that is produced. So, if `textPtr` is declared to be a character pointer, as in:

```
char *textPtr;
```

then the statement

```
textPtr = "A character string.";
```

assigns to `textPtr` a ***pointer*** to the character string "A character string."

So, with the above, we declared a character pointer and assigned a constant character string to it. If, however, you declare a character array, as with the following:

```
char text[80];
```

you cannot assign to it a constant character string, like the following:

```
text = "This is not valid";
```

The only time you can get away with that type of assignment is if you initialize it when it is declared, such as:

```
char text[80] = "This is okay.";
```

Initializing the text array in this manner does not have the effect of storing a pointer to the character string `"This is okay."` inside `text`, but rather the actual characters themselves inside corresponding elements of the `text` array.

If `text` is a character pointer, initializing text with the statement:

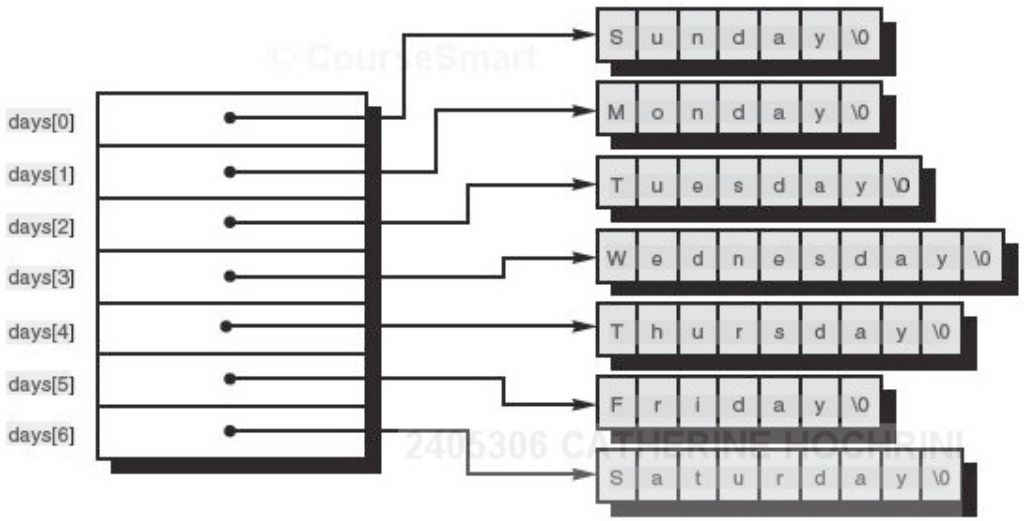
```
char *text = "This is okay.";
```

assigns to it a pointer to the character string "This is okay."

Another example to show this distinction between character strings and character string pointers:

```
char *days[] =
     { "Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday" };
```

creates an array of character pointers, each one pointing to the corresponding character array (string).



10.9 The Increment and Decrement Operators Re-visited

Recall the following:

```
i = 5;
j = i++;
```

`j` would be equal to 5 because the value of `i` is assigned to `j` first, then incremented.

If it were the other way around:

```
i = 5;
j = ++i;
```

then `j` would be equal to 6 because the pre-increment operator says to increment `i` first before its value is used, then assign it to `j`.

With respect to indexes of arrays, if you have the following (with `i` equal to 1):

```
x = a[--i];
```

then `x` takes on the value of `a[0]`, because the value of `i` is decremented before it is used as the index to `a`.

With the following example, the value of `a[1]` is assigned to `x` because `i` is decremented after its value has been used to index `a`.

```
x = a[i--];
```

More examples:

```
printf ("%i \n", ++i);
```

increments `i` then sends it to the `printf()` function.

```
printf ("%i \n", i++);
```

sends `i` it to the `printf()` function first, then increments `i`.

With pointers:

```
*(++textPtr)
```

first increments `textPtr`, and then fetches the character that it points to

```
*(textPtr++)
```

fetches the character pointed to by `textPtr` first, then increments the pointer

10.10 Operations on Pointers

We have seen that integer values can be added or subtracted from pointers; also pointers can be compared to see if they are equal, of if one pointer is less than or more than another pointer.

The only other operation permitted on pointers is the subtraction of two pointers of the same type. The result of such an operation is the number of elements contained between the two pointers. So if `a` points to an array, and `b` points to an element somewhere further down in that array, then `b-a` represents the number of elements between the two pointers (if it's a string, then it results in the number of characters between the two pointers; if it's not a string, but say, an array of integers, then it results in the number of integers between the two pointers).

10.11 The Keyword `const` and Pointers

Assume the following declarations:

```
char c = 'X';
char *charPtr = &c;
```

The pointer `charPtr` is set pointing to the variable called `c`. If the pointer `charPtr` is always going to be pointing to that variable, it can be declared as a const pointer, such as:

```
char * const charPtr = &c;
```

(i.e. "`charPtr` is a constant pointer to `c`").

Therefore, the following subsequent statement would not be allowed:

```
charPtr = &d;
```

because it has been defined to always point to `c`.

If, instead, you declared the following:

```
const char *charPtr = &c;
```

this is interpreted as "`charPtr` is pointing to a constant character, meaning the value of `c` cannot be changed by `charPtr`".

So the following would not be allowed:

```
*charPtr = 'Y';
```

but this would:

```
c = 'Y';
```