Arrays, Strings, and Files

Arrays

Arrays in C are, for the most part, the same as arrays in Java. The key differences are that arrays in C must be of a constant size (not a variable size from user input, for example). Arrays in C also do not have an associated length field that keep track of the number of slots in the array – you must keep track of this information yourself.

Declaring

Here is the format for declaring an array:

```
type name[size];
```

Here, `type` is the type of elements you want to store in the array (like `int`), `name` is the name of the array, and `size` is how many slots you want to reserve. **Note that `size` MUST be a constant**. Here is an example:

```
int nums[10];
```

However, the following will not compile because the `size` is given by a variable:

```
int size = 10;
int nums[size];
```

Initializing

Unlike Java, arrays in C are not initialized to any value. Instead, each slot in the array holds some random garbage value that is leftover in that spot in memory. Here is how you could initialize all the values in the `nums` array to `0`:

```
int i;
for (i = 0; i < 10; i++) {
     nums[i] = 0;
}
```

Notice that the first index in the array is `0`, and the last index is `size-1`. Also, recall that arrays do not have a length field – we must remember that we reserved 10 spaces for `nums`.

Arrays and Functions

Arrays can be passed to functions just like any other variable. Because arrays don't have a length field, you will almost always want to pass the size of the array and the array itself. Here's an example:

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

```
//function prototype – takes an array of ints and its size
void print(int[], int);

int main() {
     int nums[10];
     int i;

     for (i = 0; i < 10; i++) {
          nums[i] = i;
     }
     print(nums, 10);

     return 0;
}

void print(int arr[], int size) {
     int i;
     for (i = 0; i < size; i++) {
          printf("%d\n", arr[i]);
     }
}
```

Multi-Dimensional Arrays
You can create multi-dimensional arrays in C by specifying extra dimensions at the time of declaration. For example, this declares a 5x10 array of characters:

```
char array[5][10];
```

The first dimension is the row and the second dimension is the column. To access an array element, specify the desired row and column number. For example:

```
array[2][3] = 'A';    //sets element at row 2, column 3 to 'A'
```

Be Careful!
If you access an array element in Java with an index that is either negative or too big, you will get an ArrayIndexOutOfBounds exception. Java will even tell you in what file and on what line the error occurred.

C is not as friendly about this mistake. If you access an element with a bad index, such as:

```
int nums[10];
nums[10] = 0;          //10 is past the bounds of the array
```

Then one of two things will happen:

1) C will allow you to modify the memory that is just past the end of your array (where spot 10 would be if there were that many spots). This memory might belong to one of your other variables!

2) Your program will crash with a **segmentation fault** (seg fault). You will see this error quite a bit when you get started in C – it means that you tried to access memory that isn't yours. Unfortunately, the error message does not give you any information about where the problem occurred – you will have to find it yourself.

Strings

I mentioned before that there is no string type in C. This is true – but you can simulate a string by using an array of characters that is terminated with a special end-of-string character, '`\0`'.

String Variables

A string constant can be declared as follows:

```
char str[] = “Hello”;
```

After this line, `str` references the following characters in memory:

H	e	l	l	o	\0

We could have created the same string like this:

```
char str[6] = {‘H’, ‘e’, ‘l’, ‘l’, ‘o’, ‘\0’};
```

Arrays (and strings) are constant memory addresses. We can change values in strings and arrays, but we can't change the memory address. So, we can do things like this:

```
str[0] = ‘h’;          //OK
```

But we can't change the memory address (the entire string):

```
str = “hi”;            //Compiler error!
```

Later in this document, you will see a function calls `strcpy` that copies the characters from one string to another.

String Variables

If you want to be able to modify a string, you need to fill it like a normal array. For example:

```
char str[6] = {‘H’, ‘e’, ‘l’, ‘l’, ‘o’, ‘\0’};
```

(You can fill arrays like this.) Now we can modify the characters in `str`:

```
str[0] = ‘h’;
```

String Input and Output

Strings can be inputted and outputted just like any other variable. To print a string, use `printf` with the `%s` control string character. To get a string as input, use `scanf` (again with the `%s` control string character). Here’s an example:

```
char name[10];
printf(“Enter your name: “);          //Suppose you enter “Fred”
scanf(“%s”, name);
printf(“Hello, %s!\n”, name);         //Will print “Hello, Fred!”
```

Notice that when you use `scanf` to read in a string, you do not need to put an & in front of the string variable name. This is because a string is an array of characters, and arrays are already addresses of a spot in memory. (We’ll learn more about this in the section on Pointers.)

The trouble with using `scanf` to input strings is that the function doesn’t check the size of the array when it is reading input. So, if you typed the name “`George Washington`” in the above example (which needs 18 characters of space), `scanf` wouldn’t stop writing once it reached the end of the array. Instead, it would try to write past the end of the array. This would cause some of your variables to be overwritten, or a segmentation fault.

A better choice for reading strings is the `fgets` function. Here’s the prototype:

```
char[] fgets(char s[], int size, FILE *stream)
```

You pass `fgets` the string buffer (`s`), the size of the buffer (`size`), and the stream you’re reading from (`stdin` in this case). It returns the string it read, or `NULL` if it was unable to read anything. It will stop reading user input when either:

- It has read `size-1` characters (it needs the last spot for a `\0`)
- It has reached the end of the input
- It has reached a newline

Here the same example using `fgets`:

```
char name[10];
printf(“Enter your name: “);          //Suppose you enter “Fred”
fgets(name, 10, stdin);
printf(“Hello, %s!\n”, name);         //Will print “Hello, Fred!”
```

Conversions

It is sometimes necessary to convert between strings, ints, and doubles. Here is a list of conversion functions:

- `atoi`: converts from a string to an int
- `atof`: converts from a string to a double
- `itoa`: converts from an int to a string
- `ftoa`: converts from a double to a string

To use any of these functions, you need to add:

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

Here is an example of using the conversion functions:

```
char buff[10];
int num;
double d;

printf("Enter an integer: ");
fgets(buff, 10, stdin);           //Suppose you enter "47"
num = atoi(buff);                 //num = 47

printf("Enter a real number: ");
fgets(buff, 10, stdin);           //Suppose you enter "4.75"
d = atof(buff);                   //d = 4.75

itoa(num, buff, 10);              //buff = "47"
itof(d, buff, 10);                //buff = "4.75"
```

<u>String Functions</u>

Below is a list of common string functions. To use any of these, you need to add:

```
#include <string.h>
```

`char[] strcat(char str1[], char str2[])`

This function copies the characters in `str2` onto the end of `str1`. It returns the newly concatenated string (although `str1` also references the concatenated string).

For example:

```
char str1[20];
char str2[20];
printf("Enter two words: ");   //Suppose you entered "hi hello"
scanf("%s %s", str1, str2);

strcat(str1, str2); //str1 = "hihello", str2 = "hello"
```

`int strcmp(char str1[], char str2[])`

This function compares `str1` and `str2` to see which string comes alphabetically before the other. It returns a number less than 0 if `str1` comes before `str2`. It returns 0 if `str1` equals `str2`. It returns a number greater than 0 if `str1` comes after `str2`.

For example:

```
char str1[20];
char str2[20];
printf("Enter two words: ");   //Suppose you entered "hi hello"
scanf("%s %s", str1, str2);

//Would print "hello comes first"
if (strcmp(str1, str2) < 0) {
     printf("%s comes first\n", str1);
}
else if (strcmp(str1, str2) > 0) {
     printf("%s comes first\n", str2);
}
else {
     printf("The strings are equal\n");
}
```

`char[] strcpy(char str1[], char str2[])`

This function copies the characters in `str2` into `str1`, overwriting anything that was already in `str1`. It returns the newly copied string (although `str1` also references the copied string).

For example:

```
char src[20];
char dest[20];
printf("Enter a word: ");
scanf("%s", src);     //Suppose you entered "hello"

strcpy(dest, src);    //Now dest also holds "hello"
src[0] = 'B';         //Now src is "Bello", and dest is "hello"
```

`int strcspn(char str1[], char str2[])`

This function returns the number of characters read in `str1` before reaching ANY character from `str2`.

For example:

```
char str[20];
int index;
printf(“Enter a word: “);//Suppose you entered “hello”
scanf(“%s”, str);

index = strcpsn(str, “la”);    //index is 2
                               //read 2 characters before ‘l’
```

`int strlen(char str[])`
This function returns the number of characters in `str`.

For example:

```
char str[20];
printf(“Enter a word: “);//Suppose you entered “hello”
scanf(“%s”, str);

printf(“%d\n”, strlen(str));  //prints 5
```

`char[] strtok(char str[], char delim[])`
This function returns the first token in `str` before the occurrence of any character in `delim`. (After the first call to `strtok`, pass `NULL` as `str`. It will continue looking for tokens in the original string.)

For example:

```
char buff[200];
char *token;     //We’ll learn about this in “Pointers”
printf(“Enter names, separated by commas: “);

//Suppose you entered “Fred,James,Jane,Lynn”
scanf(“%s”, buff);

token = strtok(buff, “,”);
while (token != NULL) {
     printf(“%s\n”, token);
     token = strtok(NULL, “,”);
}
//Fred, James, Jane, and Lynn will each be printed on separate lines
```

`char[] strncpy(char str1[], char str2[], int n)`

This function copies the first `n` characters from `str2` to `str2`, overwriting anything that was already in `str1`. It returns the newly copied string (although `str1` also references the copied string).

`int strncmp(char str1[], char str2[], int n)`
This function compares the first `n` characters in `str1` and `str2` to see which prefix comes first alphabetically. It returns a number less than 0 if the first `n` characters of `str1` come before the first `n` characters of `str2`. It returns 0 if the first `n` characters of `str1` equal the first `n` characters of `str2`. It returns a number greater than 0 if the first `n` characters of `str1` come after the first `n` characters of `str2`.

`char[] strrchr(char str[], char c)`
This function finds the LAST occurrence of `c` in `str`. It returns the suffix of `str` that begins with the last occurrence of `c`.

`int strspn(char str1[], char str2[])`
This function returns the number of characters read in `str1` before reaching a character that is NOT in `str2`.

`char[] strstr(char str1[], char str2[])`
This function sees whether `str2` is a substring of `str1`. If `str2` is not a substring, it returns `NULL`. If `str2` is a substring, it returns the suffix of `str1` beginning with the `str2` substring.

Be Careful!
It's very easy to make a mistake when using strings. Strings are arrays, so you will get in trouble if you try to access memory beyond the end of the array. For example:

```
char buff[5];
printf("Enter a word: ");

//Suppose you enter  "Hello"
scanf("%s", buff);
```

`scanf` will copy the characters 'H', 'e', 'l', 'l', 'o' into the array. However, it will then try to add the end-of-string character, '\0', into the 6th spot in the array. This is past then end of the array, and so your program will either crash with a segmentation fault, or you will overwrite the value of some other variable. A lot of the string functions involve writing to strings, and none of them will handle an out-of-bounds error gracefully. When you use the following functions, MAKE SURE you have enough memory allocated:

- `scanf`
- `strcpy`
- `strcat`

- `strncpy`

File I/O

This section contains information on opening a file, reading from a file, and writing to a file. I only cover how to interact with text files – it is also possible to read from and write to binary files.

Whenever you are doing file I/O, you need to add:

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

Opening a File

Before we can interact with a file, we need to open it. The `fopen` function lets us open files for different kinds of input and output. Here's the prototype:

```
FILE* fopen(char[] filename, char[] mode)
```

The `FILE*` return type means that the function is returning the address of a `FILE` object. We'll learn more about pointers in the next section. If the file could not be opened, `fopen` returns `NULL`.

Here, `filename` is a string representation of the filename, such as "`data.txt`". `fopen` searches the current directory for the file if no absolute path is given. The string mode specifies what type of operations you want to do on the file. Here are the different options for the mode:

Mode	Description
"r"	Open for reading (file must exist)
"w"	Open for writing (overwrites old data)
"a"	Open for appending (creates file if necessary)
"r+"	Open for reading and writing (file must exist)
"w+"	Open for reading and writing (overwrites old data)
"a+"	Open for reading and appending (opens at end of file)

For example, we can open the file "`data.txt`" for reading, and print an error if we were unsuccessful:

```
FILE *fp = fopen("data.txt", "r");
if (fp == NULL) {
     printf("Error opening file\n");
}
```

After we are done reading from a file or writing to a file, we must close the file with the `fclose` function. Here's the prototype:

```
int fclose(FILE* fp)
```

To close `data.txt`, we'd do:

```
fclose(fp);
```

Reading from a File
There are two major functions for reading from a file – `fscanf` and `fgets`. `fgets` works exactly like we've seen before, except now we specify a `FILE*` instead of `stdin`. `fscanf` works exactly like `scanf`, except we first specify the `FILE*`. We'll start with `fscanf`:

```
int fscanf(FILE *stream, char str[], variable addresses...)
```

`fscanf`, like `scanf`, returns the number of variables that were correctly read in. If it was unable to read any more input, the `EOF` constant is returned. Thus we can compare the return value of `fscanf` to `EOF` to see if we've reached the end of the file.

Suppose the file `data.txt` looks like this (a bunch of names and ages, each on separate lines):

Bob 20
Jill 15
Tony 17
Lisa 22

We want to read this file, and print something like "`Bob is 20 years old`" to the console for each person in the file. Here's how:

```
FILE *fp = fopen(“data.txt”, “r”);
char name[20];
int age;
if (fp != NULL) {
     while (fscanf(fp, “%s %d”, name, &age) != EOF) {
          printf(“%s is %d years old\n”, name, age);
     }
     fclose(fp);
}
```

Now, lets try to do the same thing with the `fgets` function. Here's the prototype:

```
char[] fgets(char s[], int size, FILE *stream)
```

`fgets` reads a string from a specified file into the `s` array. The `size` parameter specifies the size of the string – it will not write past the end of the array. It returns a reference to the string that was read. If no string was read (specifying an error or the end of file), `NULL` is returned. `fgets` will attempt to read `size-1` characters **unless it reaches a newline or the end of the file**.

Here's the same example repeated with `fgets`:

```
FILE *fp = fopen(“data.txt”, “r”);
char name[20];
char buf[30];
int age;
if (fp != NULL) {
     while (fgets(buf, 30, fp) != NULL) {
          //parse the current line
          char *token = strtok(buf, “ “);
          strcpy(name, token);

          //get the age
          token = strtok(buf, “ “);
          age = atoi(token);
          printf(“%s is %d years old\n”, name, age);
     }
     fclose(fp);
}
```

Reading files with `fscanf` is usually simpler (since it doesn't involve parsing lines), but it is more error-prone than `fgets`.

Writing to a File
The primary function for writing to a file is `fprintf`. This function works exactly like `printf`, but the first argument is now a `FILE*`. Here's the prototype:

```
int fprintf(FILE* fp, char str[], variables to print...)
```

Here is an example that will ask the user to input 10 numbers. Each number will be written on a separate line to the file `out.txt`:

```
FILE *fp = fopen(“out.txt”, “w”);
if (fp != NULL) {
int num, i;
for (i = 0; i < 10; i++) {
          printf(“Type a number: “);
          scanf(“%d”, num);
          fprintf(fp, “%d\n”, num);
     }
     fclose(fp);
}
```