CS102: Strings

In computer terminology, a **string** is a sequence of characters. In C, strings are generally stored in arrays or else pointed to by pointers. Often, both happen at once (the string is stored in an array, and you use a pointer to walk through it one character at a time). In general, strings can have varying lengths, and there are at least three different ways to deal with this hypothetically (in C, you almost always use the third method):

1) Make sure all strings have a fixed length. Sometimes, this can be done. For instance, let's say you want to store a string representing a zipcode, and you know it will be five digits. You could use the following array:

```
char zipcode[5];
```

Each element of the array "zipcode" will store an ASCII character that represents a digit. For example, if you want to store my home-town zipcode of "07410", the following values could be assigned to the array:

```
zipcode[0] = '0';
zipcode[1] = '7';
zipcode[2] = '4';
zipcode[3] = '1';
zipcode[4] = '0';
```

2) Use the first character of an array to store its length. For instance, let's say you want to store a person's name, and you assume it will be at most 40 characters (including the one that stores the length). You could use the following array:

```
char name[40];
```

If you want to fill it in with the name "Bob", one way to do so would be as follows:

```
name[0] = 3;
name[1] = 'B';
name[2] = 'o';
name[3] = 'b';
```

Of course, since "name" is an array of characters, C will think that name[0] is the ASCII character whose value is 3, and your code will just have to know to interpret it as a length instead (the real name starts at "name[1]"). Partly for this reason, and partly because the third method discussed below is assumed by many of the provided library functions, this method is rarely used to represent strings of varying lengths in C.

3) Have a special character, called a **delimiter**, to end the string. This is how C almost always handles strings of varying lengths, and the delimiter used to end the string is the ASCII **null character**. Its value is 0, and it can also be represented as '\0'. If we

want to use the same "name" array as above and fill it in with "Bob", one way to do so would be as follows:

```
name[0] = 'B';
name[1] = 'o';
name[2] = 'b';
name[3] = '\0';
```

The last assignment could also be:

```
name[3] = 0;
```

A string ending in a null character is called a **null-terminated string**. Most strings you deal with in C are null-terminated strings. In fact, most library functions that C provides assume that strings passed to them are null-terminated. Also, whenever you specify a string in quotes, the C compiler automatically adds a null character to the end of it. So when we use the statement:

```
printf("Hello World!\n");
```

The "printf" function assumes that the string being passed to it ends in a null character. It displays one character at a time to standard output, handling conversion codes when necessary, and stops when it encounters a null character. The fact that we are expressing the above string in quotes causes the C compiler to automatically add a null character to the end of it. Remember the '\n' represents one character (the newline character), so all together, not including the null character, the string contains 13 characters, and with the null character it contains 14 characters.

You don't have to pass "printf" a string constant. You can pass it a variable representing a string. For instance, let's say we have the null-terminated version of "Bob" in the "name" array as above. We could say:

```
printf(name);
```

This will print the name "Bob" to standard output. However, notice there is no newline at the end of the string, so the next Unix command prompt will appear directly after the string. If we wanted a newline, we would have had to set "name[3]" to '\n' and then "name[4]" to '\0'.

It is also possible to use a special conversion code, "%s", to print out a string variable along with a constant string. For example:

```
printf("My name is %s.\n", name);
```

This will print "My name is Bob." to the screen on its own line.

Note in these examples that name is an array of 40 characters, but only 4 are being used. The rest have not been initialized, and contain whatever random values happen to be in the memory locations where the rest of the array is being stored. Since we're not using those locations yet, it doesn't matter.

What happens if you try to print out a string that is not null-terminated? For instance, let's say we're using the "zipcode" array above, and we have a statement:

```
printf("My zipcode is %s.\n", zipcode);
```

You might see something like this:

```
My zipcode is 07410ÿùô.
```

Why? Because the "printf" routine prints the characters of a string one character at a time until a null character is seen. Even though "zipcode" has been defined as an array of 5 characters, an array name is also a pointer constant to the first element of the array, and "printf" printed out the characters in "zipcode" followed by whatever characters happened to be in memory afterwards until one of them (four characters later) had the value of 0. If you try out the experiment, you'll see other garbage characters printed. Often, there are many more than just 3. It is a very common error to try to print out a string that is not null-terminated.

You can initialize a string, whether it is stored in an array or with a pointer, with the declaration. For example:

```
char message[20] = "Hello World!\n";
```

This creates an array of 20 characters and fills in the first 14 with the 13 characters inside the double quotes followed by a null character.

The statement:

```
printf(message);
```

would then do what we have seen so many times.

Also, if you want the array to be just long enough for the message, you can do:

```
char message[] = "Hello World!\n";
```

The compiler will count the length of the string, add one, and interpret "message" to be an array of 14 characters, just long enough to hold the string with the null character.

You can also use a pointer:

```
char *pc = "Hello World!\n";
```

which might be followed by:

```
printf(pc);
```

Of course, there is a difference in the two things we've done above. The variable "message" is a pointer constant, it can't be changed. You can change what's in the array, but you can't make "message" point to other memory. The variable "pc" is a pointer, and you can change what's in the memory it points to or you can make it point somewhere else.

Often, you will want to read a string from the user. One way to do this is with "scanf". For instance, let's look at our first full program with strings:

```
#include <stdio.h>

int main(void)
{
  char name[20];

  printf("Please enter your name: ");
  scanf("%s", name);

  printf("Hello %s! Glad to meet you.\n", name);
  return 0;
}
```

You can probably guess what this does, but there are a few things to point out.

One is that when "scanf" reads a string, it skips leading whitespace. Once it finds a character, it keeps reading until it finds more whitespace, putting each character into the array or memory pointed to by the pointer variable one at a time in order. When it finds trailing whitespace (which might be the newline character but doesn't have to be), it ends the string with a null character, and the string is null-terminated. So if you run this program and type in your first name followed by your last name and hit enter, only your first name will be printed out in the message.

Also, notice above we are using only a 20 character array. This is enough to hold 19 characters in the name plus a null character. What if the user types more? The "scanf" function will write the entire name into the memory pointed to by the array name followed by a null character. Therefore, you will be writing passed the end of the array into memory that you don't own. In such a simple program, you might get lucky and have things work anyway, but you might crash, and in a more complex program, if you do something like this you'll almost certainly crash. One way to protect against this is to use a width field specification with the string conversion code. This causes "scanf" to only read up to a certain number of characters. So we could use:

```
scanf("%19s", name);
```

Notice the width is one less than the size of the array since we need room for the null character! Now, if the user enters more than 19 characters, only the first 19 will be read, and they will be written into the array followed by a null character. The rest of the characters typed by the user will still be in the standard input buffer! This can cause problems. Often, after reading anything from the user, you want to flush what is left in the input buffer, so that the next time you use "scanf" or "getchar" or any input function, you'll be starting with a clean slate and force the user to type more. One way to flush the remainder of the input buffer is with the following statement:

```
while(getchar() != '\n');
```

This will read all characters in the input buffer up to and including the newline character at the end of it. It's a very useful statement!

You can also a width specification when printing a string to help format your output. For instance:

```
printf("Name: %20s\n", name);
```

This will ensure that the name takes up 20 characters when printed to standard output. As many spaces as necessary will be added to the left of the string.

The standard input and output library provides two additional functions for inputting strings from standard input and outputting strings to standard output.

The first is "gets". The prototype statement for the function is:

```
char *gets (char *str);
```

It gets passed a pointer to a string (which could be the name of an array) and it reads one whole line from standard input and writes it to the memory pointed to by the string (which might be the memory allocated for an array). If you are passing a pointer, it is important to make sure that the pointer points to allocated memory, and that there is enough memory to hold the maximum possible length of a line. When reading the line of input to the string, the final newline character is converted to a null character so the string is null-terminated.

If the input is successful, the function returns a pointer to the string (i.e., the value returned by the function is the same as the value of the pointer passed to the function). If the input is not successful, it returns NULL. (Note, this is a NULL pointer, not a null character!) Usually, if you use the return value of "gets" at all, it is just to check if it is NULL. The main reason that input might not be successful is if an EOF (end-of-file character) is encountered before any data is read.

The other function is "puts". The prototype statement for this function is:

```
int puts(const char *str);
```

It gets passed a string and writes the string to standard output (not including the null character) followed by a newline character. Don't worry about the "const". Basically, it just means that the function is not allowed to change the value of the pointer.

Now we'll look at another simple program:

```
#include <stdio.h>

int main(void)
{
  char line[40];

  while (gets(line) != NULL)
  {
    puts(line);
  }

  return 0;
}
```

This program reads one line at a time from standard input and writes the line back to standard output. (Since each iteration of the loop executes just one statement, we didn't have to make it a compound statement.) If we redirect standard input to come from a file, the file will be displayed to standard output, which we could also redirect to a file if we want. When we covered standard input and standard output, we saw another way to do this using "getchar" and "putchar".

C comes with two standard libraries that make working with strings simpler. One is the **string library** with header file "string.h". A few of the functions in it are:

"strlen" - This function gets passed a string and returns the length of the string (i.e., the number of characters in the string up to but not including the null character). For example, you might see:

```
length = strlen(name);
```

Note: If you try to use "strlen" to compute the length of a string that is not null-terminated, the results are unpredictable! It will search in memory starting at the beginning of the string, passed the end of the string, until it sees a null character, and it will think this is the end of the string!

"strcpy" - This function takes two strings and copies the second to the first. It also returns the value of the first pointer, but often this value does not get used in the program. For example, you might see:

```
strcpy(name2, name1);
```

So this would copy the contents of "name1" including the null character to "name2". If the first argument is a pointer, you need to ensure it points to allocated memory (an array or memory allocated using "malloc") before the call to "strcpy"!

"strcmp" - This function takes two strings and compares them. If they are equal, it returns a 0. If the first string is less than the second string (sort of alphabetically, but determined by the ASCII character set), it returns some number less than 0. If the first string is greater than the second string, it returns some number greater than 0. This function is often used in programs that need to sort strings, and it is also often used to check if two strings are equal. For example:

```
if (strcmp(name1, name2) == 0)
     printf("The two names are identical.\n");
```

or equivalently:

```
if (!strcmp(name1, name2))
     printf("The two names are identical.\n");
```

Let's say that "pc1" is pointing to a string, and you want to make a copy of it in "pc2". You can't just say:

```
pc2 = pc1;
```

That is a valid C statement, but it will mean that "pc2" points to the same memory as "pc1", and if you change the characters of the string through either pointer, you would be changing the contents of both strings. Instead, you have to do something like this:

```
length = strlen(pc1);
pc2 = (char *) malloc(sizeof(char) * (length + 1));
strcpy(pc2, pc1);
```

The first line gets the length of the string pointed to by "pc1" (not including the null character) and stores the length in the variable "length" which we're assuming is an integer. The next line allocates enough memory to store the string and has "pc2" point to the first byte of this memory. Note that we need enough room for "length+1" characters, since the length does not include the null character, which must get copied also! (Otherwise, the copied string would not be null-terminated.) The third line copies the contents of "pc1" to the memory pointed to by "pc2" (including the null character).

Now we're going to examine a program that reads one line at a time from standard input until the user types the word "quit" on a line by itself. For each line up until the final one, the program will display the line backwards to the screen. We'll assume that each line will contain 40 or less characters (including the null character).

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

int main(void)
{
  char line[50];
  char *pc;

  while(1)
  {
    if(gets(line) == NULL)
    {
      printf("End of file encountered.\n");
      break;
    }
    if (strcmp(line, "quit") == 0)
    {
      printf("Good bye!\n");
      break;
    }

    pc = line + strlen(line) - 1;
    while (pc >= line)
    {
      putchar(*pc);
      pc--;
    }
    putchar('\n');
  }

  return 0;
}
```

There are obviously many ways to do this, and this is just one. We loop potentially forever inside the "while(1)" loop, until we encounter an EOF (in which case "gets" returns a NULL) or until the user enters "quit"; after either of these occurrences, we break out of the loop. For each line, we set a pointer to point to the end of the string and then walk down to the beginning of the string, printing characters one at a time. When we are done with the current line, we print out the newline character. In this program I used "putchar" to displaly each character, but I could have used "printf" with a "%c" conversion code. I used a "while" loop, but could have just as easily used a "for" loop. Of course, I could have treated the strings using arrays and indexes, but I chose to use pointers instead.

If you compile this program and run the executable redirecting the standard output to come from its source code, the output looks like this:

```
>h.oidts< edulcni#
>h.gnirts< edulcni#

)diov(niam tni
{
;]05[enil rahc
;cp* rahc

)1(elihw
{
)LLUN == )enil(steg(fi
{
;)"n\.deretnuocne elif fo dnE"(ftnirp
;kaerb
}
)0 == )"tiuq" ,enil(pmcrts( fi
{
;)"n\!eyb dooG"(ftnirp
;kaerb
}

;1 - )enil(nelrts + enil = cp
)enil => cp( elihw
{
;)cp*(rahctup
;--cp
}
;)'n\'(rahctup
}

;0 nruter
}
End of file encountered.
```

The other library often helpful when dealing with strings is the **character handling library** with head file "ctype.h". Some of the functions are:

"isalpha" - takes a character as a parameter and returns a non-zero value (true) if the character is a letter (upper-case or lower-case) and zero (false) otherwise.

"islower" - takes a character as a parameter and returns a non-zero value (true) if the character is a lower-case letter and zero otherwise.

"isupper" - you can guess!

"tolower" - takes a character as a parameter. If this character is a letter, it returns the lower-case version of the letter (not changing it at all if it was already lower-case). If the character is not a letter, it just returns the character unchanged.

"toupper" - you can guess!

Let's say that "pc1" and "pc2" are two pointers to characters. You want to check if they both point to letters that are the same letter (case-insensitive). One way to do this is:

```
if (isalpha(*pc1) && isalpha(*pc2) && (tolower(*pc1) ==
tolower(*pc2)))
```

Remember, the "isalpha", "islower", and "isupper" functions return integers that you will generally use as Boolean values (true or false). The functions "tolower" and "toupper" return a character.

Here is another program similar to the previous program, but this program prints the characters of each line in forwards order converting all letters to lower-case.

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

int main(void)
{
  char line[40];
  char *pc;
  int length;

  while(1)
  {
    if(gets(line) == NULL)
    {
      printf("End of file encountered.\n");
      break;
    }
    if (strcmp(line, "quit") == 0)
    {
      printf("Good bye!\n");
      break;
    }

    length = strlen(line);
    for (pc = line; pc - line < length; pc++)
    {
      putchar(tolower(*pc));
    }
```

```
    putchar('\n');
  }

  return 0;
}
```

Note that we are including three header files at the top of this program, and using functions from all three of the correspodning libraries.