Strings

Learn how to declare strings and how C handles them.

String is one of the most popular data types in programming. To be honest, it's slightly awkward dealing with character strings in C, especially when compared to languages like Python.

A collection of characters grouped together forms a string. We have seen earlier how constant character strings are written inside double-quotation marks as opposed to character constants that are declared using single-quotation marks.

In C, a string is actually, under the hood, an array of characters. String variables can be declared as indicated on line 5 below. The double square brackets signify an array. Just like arrays, we can access string elements through indexing.

```c
#include <stdio.h>

int main(void) {
  char c = 'p';          // A character
  char s[] = "Paul";     // A string
  printf("c = %c and s = %s\n", c, s);
  return 0;
}
```

Declaring a string vs. declaring a character

Null-termination

An important thing to remember about strings is that they are always null-terminated. This means that the last element of the character array is a null character, written as `'\0'`. When we declare a string as on line 4 below, we don't have to put the null termination character in ourselves, the compiler does it for us. We can use the `sizeof` function to inspect our character string above to see how long it actually is:

```c
#include <stdio.h>

int main(void) {
  char s[] = "Paul";
  printf("s is %ld elements long\n", sizeof(s));
  return 0;
}
```

Using `sizeof` to verify the length of a string.

Modifying strings

In C, it is a bit tricky to modify a declared string in the sense that once a string is declared and has a specific length, we cannot just make it longer or shorter by reassigning a new constant to the variable.

We can directly edit the contents of a string because it is simply an array of characters.

```
char s[] = "String";
s[3] = 'o';
```

Now, `s` would be `Strong` instead of `String`.

We can't change the length of the array, but we can sort of make it shorter, by overwriting a character of the string by a null character and terminating the string. Then, essentially, we'll have a short string sitting in a long array, but that's ok since we know where the end is (the null character).

```
#include <stdio.h>

int main(void) {

  char s[] = "This is a string";
  printf("Original String: %s\n", s);

  // Change the contents of the string
  s[2] = 'a';
  s[3] = 't';
  printf("Modified String: %s\n", s);

  // Shorten the string
  s[7] = '\0';
  printf("Shortened String: %s\n", s);

  return 0;
}
```

So, to summarize, in C, strings are simply null-terminated arrays of characters.

Next, we'll look at the various functions we can perform on strings.

String Handling Using the C Standard Library

Learn how the C standard library enables us to manage strings using functions for concatenating strings, finding their lengths, testing for equality, as well as allowing conversions between strings and numbers.

The C standard library (which we can load into our program by including the directive `#include <string.h>` at the top of our program), contains many useful routines for manipulating these null-terminated strings.

It's a good idea to consult a reference source for a long list of all the functions that exist for manipulating null-terminated strings in C. There are functions for copying strings, concatenating strings, getting the length of strings, comparing strings, etc.

Concatenating and finding length

If we place two string constants together side by side, the C compiler will concatenate the constants. This doesn't apply to strings that are stored as arrays of characters and for them we explicitly need to use the `strcat` function.

```
#include <stdio.h>

int main() {
  // Concatenating two string constants and printing the
      resulting string
  printf("Paul" " Gribble");
  return 0;
}
```

Concatenating two string constants.

We can use the `strcat` function to concatenate two strings by appending the second string to the first one. We show this below, by passing the string `s3` as the first argument in each call to `strcat`.

We can use the `strlen` function to find the length of a string in terms of characters that appear before the (first) null character.

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

int main(void) {

  char s1[] = "Paul";
  char s2[] = "Gribble";
  char s3[256] = "";
  printf("s3 = %s , strlen(s3) = %ld\n", s3, strlen(s3));

  strcat(s3, s1);
  printf("s3 = %s , strlen(s3) = %ld\n", s3, strlen(s3));

  strcat(s3, " ");
  printf("s3 = %s , strlen(s3) = %ld\n", s3, strlen(s3));

  strcat(s3, s2);
  printf("s3 = %s , strlen(s3) = %ld\n", s3, strlen(s3));

  return 0;
}
```

Using the `strcat` and `strlen` functions.

Comparing two strings

Importantly, we cannot simply use the `==` operator to test whether two strings are equal. Remember, strings are arrays of characters. We have to use a special string handling function to test the equality of two strings because it has to do a 'deep' comparison, comparing each element against the other. Here's how we would do it:

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

int main(void) {

  char s1[] = "Paul";
  char s2[] = "Paul";
  char s3[] = "Peter";
  char s4[] = "Dave";

  printf("strcmp(s1,s2)? %d\n", strcmp(s1, s2));
  printf("strcmp(s1,s3)? %d\n", strcmp(s1, s3));
  printf("strcmp(s1,s4)? %d\n", strcmp(s1, s4));

  return 0;
}
```

Using the `strcmp` function.

Note that the `strcmp(s1,s2)` function call returns 0 if `s1` and `s2` are equal, a positive value if `s1` is (lexicographically) greater than `s2` and a negative value if `s1` is less than `s2`.

Type conversions

Strings to numbers

There are several functions to convert strings to numeric types like integers and floating-point numbers. We'll need to include `stdlib.h` at the top of our program.

A function call `atof(s)` converts the string pointed to by `s` into a floating-point number (a double), returning the result. A function call `atoi(s)` converts the string `s` into an integer. There are a host of others. Again, we suggest consulting a reference source for a comprehensive list.

Numbers to strings

A common way of converting a numeric type like an integer or a floating-point number into a string is to use the `sprintf` function. It is used much like the `printf` function we have seen earlier, but instead of printing something to the screen, `sprintf` 'prints' something to a character string. Here's how we can use it to convert the given numbers to the strings `s1` and `s2` (passed as the first argument):

```
#include <stdio.h>

int main(void) {
  char s1[256];
  char s2[256];
  int i1 = 12;
  double d1 = 3.141;

  sprintf(s1, "%d", i1);
  sprintf(s2, "%.3f", d1);

  printf("s1 = %s\n", s1);
  printf("s2 = %s\n", s2);

  return 0;
}
```

Using the **`sprintf`** function to convert to a string.

Note how on line 5 and line 6, the arrays `s1` and `s2` are declared as character arrays large enough to hold 256 characters. If we try to `sprintf` to a string that's not big enough to hold what we're trying to put into it, then we'll end up writing values beyond the end of the string and onto who knows what, in memory.

If we are dealing with strings that we know will be relatively short (things like file names, subject names, dates, etc) then probably the easiest way of doing things is to use preallocated strings that are long enough to hold any reasonable value (e.g. 256 characters long). After all, we have enough RAM in our computers these days so we don't need to worry too much about 256 bytes here and there.

Numbers to string II (slightly esoteric)

There is, however, a way to convert to string without having to hard-code the size of the string to be written to, although it's a little bit roundabout. However, it does illustrate several principles of C so let's take a look at it.

1. First, we will use the `snprintf` function in a roundabout way to determine the number of bytes that the numeric to string conversion will result in.

2. Then, we will use `malloc` to allocate a new string (character array) of that length.

3. Finally, we will use `sprintf` to write to that character array.

The first step ensures that we have a character array (a string) that is just the right length to receive the converted numeric: not too small and not too big.

Here is a sample code that demonstrates this, first for an integer conversion and then for a floating-point conversion:

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

```c
int main(void) {

  int size;

  int x = 8765309;
  size = snprintf(NULL, 0, "%d", x);
  char *xc = malloc(size + 1);
  sprintf(xc, "%d", x);

  double y = 876.5309;
  size = snprintf(NULL, 0, "%.4f", y);
  char *yc = malloc(size + 1);
  sprintf(yc, "%.4f", y);

  printf("xc = %s\n", xc);
  printf("yc = %s\n", yc);

  free(xc);
  free(yc);

  return 0;
}
```

Using the functions `snprintf` and `sprintf` to convert to strings.

The trick is to use `snprintf` as shown on line 9 and line 14, with `NULL` passed as the first argument and 0 as the second argument.

The `snprintf` function is like `sprintf`, but it takes as its second argument the maximum number of bytes to write out to the destination string which is passed in the first argument. Therefore, `snprintf` can be thought of as a 'safe' version of `sprintf` in that we know for sure that it'll never write out more than the maximum number of bytes we asked for. Therefore, we can avoid over-writing past the end of our destination string buffer.

As for its return value, the `snprintf` function returns the number of bytes that would have been written had the second argument been sufficiently large (not counting the termination `\0` character).

So here we are passing `NULL` as the first argument and 0 as the second. As a result, `snprintf` won't actually write any characters anywhere, it will simply return the number of characters that would have been written. Then on line 10 and line 15, we can use the `malloc` function to dynamically allocate character arrays of exactly the required length.

We've seen that strings work a lot like arrays. It should then be no surprise that we can create an array of strings. In the next lesson, we'll see how it works.

Arrays of Strings

Learn how to make an array of strings using pointers.

We have seen that strings are just arrays of characters, terminated by a null character. We have also seen that the variables that hold strings (like arrays of other types, e.g. `int` or `double`) are actually pointers to the head of the array.

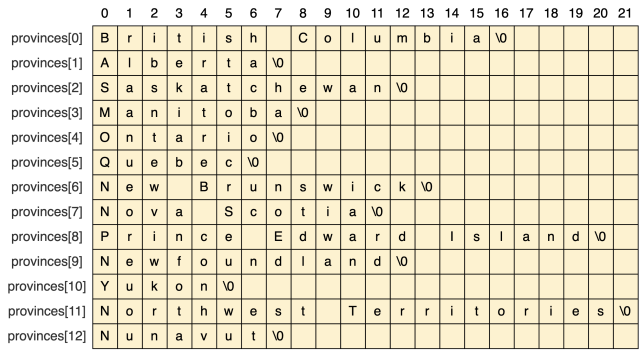
Using an array of pointers

We can use an array of pointers, where each pointer points to the head of a character array (in other words a string), to store an array of strings. Here's a coding example:

```c
#include <stdio.h>

int main(void)
{
  char *provinces[] = { "British Columbia", "Alberta", "
     Saskatchewan",
                        "Manitoba", "Ontario", "Quebec", "
                           New Brunswick",
                        "Nova Scotia", "Prince Edward Island
                           ", "Newfoundland",
                        "Yukon", "Northwest Territories", "
                           Nunavut" };
  int i;
  for (i = 0; i < 13; i++) {
    printf("provinces[%d] = %s\n", i, provinces[i]);
  }
  return 0;
}
```

An array of strings.
Here is the visual representation of an array of the strings:



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
provinces[0]	B	r	i	t	i	s	h		C	o	l	u	m	b	i	a	\0					
provinces[1]	A	l	b	e	r	t	a	\0														
provinces[2]	S	a	s	k	a	t	c	h	e	w	a	n	\0									
provinces[3]	M	a	n	i	t	o	b	a	\0													
provinces[4]	O	n	t	a	r	i	o	\0														
provinces[5]	Q	u	e	b	e	c	\0															
provinces[6]	N	e	w		B	r	u	n	s	w	i	c	k	\0								
provinces[7]	N	o	v	a		S	c	o	t	i	a	\0										
provinces[8]	P	r	i	n	c	e		E	d	w	a	r	d		I	s	l	a	n	d	\0	
provinces[9]	N	e	w	f	o	u	n	d	l	a	n	d	\0									
provinces[10]	Y	u	k	o	n	\0																
provinces[11]	N	o	r	t	h	w	e	s	t		T	e	r	r	i	t	o	r	i	e	s	\0
provinces[12]	N	u	n	a	v	u	t	\0														

Visual representation of an array of pointers.