Strings

A string is a sequence of chars stored in adjacent memory locations **terminated by an ASCII NUL** - written as `'\0'`, the character with ASCII code 0

Considered instead as bytes, a string is a sequence of non-zero bytes terminated by a zero byte

An Example:

chars:	'H'	'e'	'l'	'l'	'o'	' '	't'	'h'	'e'	'r'	'e'	'\0'
bytes:	72	101	108	108	111	32	116	104	101	114	101	0

`char` values of 0–127 are normal ASCII codes that are portable between different platforms

`char` values 128–255 are non portable between platforms

See `man ascii` for the ASCII table

The single biggest mistake people make when dealing with strings is **to fail to ensure that the string is properly terminated**

Any function that walks across a string will **fail to stop if the `'\0'` is missing**

Declaring and initialising string variables

The most obvious way of storing a string in C - one that you can modify, sometimes called a string buffer - is to declare a suitably-sized array of `char`:

```
char stringvar[100];
```

You can of course initialize the string at the same time:

```
char stringvar[100] = "Hello!";
```

This will initialize the array as follows:

0	1	2	3	4	5	6	7	8	9	10	..	99
'H'	'e'	'l'	'l'	'o'	'!'	'\0'	?	?	?	?	..	?

You can initialize the string and omit the size, in which case it's automatically calculated as 1 + the length of the initializer string:

```
char stringvar[] = "Hello!";
```

`string.h` functions

- Once we have our modifiable string buffer `stringvar`, there are many standard library functions (from <`string.h`>) that we can use to alter it's contents:
- You can copy a new string into `stringvar`, overwriting it's current contents:

```
strcpy( stringvar, "Hi!" );
```

This will alter the `stringvar` array as follows:

0	1	2	3	4	5	6	7	8	9	10	..	99
'H'	'i'	'!'	'\0'	'o'	'!'	'\0'	?	?	?	?	..	?

- Note that the array contains traces of the initial, longer, string beyond the terminator. This normally doesn't matter, because the string terminates at the first terminator. Gotcha: unless you deliberately overwrite the terminator: `stringvar[3] = 'H';` giving `Hi!Ho!`
- Gotcha: It's your responsibility to ensure that there's enough space for any string you strcpy() into `stringvar` to fit, otherwise buffer overrun occurs.

Of course we don't have to use strcpy(), we could build up our strings character by character, not forgetting the terminator. But that's painful so let's not:-)

You can append a new string to the end of `stringvar`, using strcat(). If `stringvar` was currently `Hi!` then:

```
strcat( stringvar, " there" );
```

will alter the `stringvar` array as follows:

0	1	2	3	4	5	6	7	8	9	10	..	99
'H'	'i'	'!'	' '	't'	'h'	'e'	'r'	'e'	'\0'	?	..	?

Gotcha: It's your responsibility to ensure that there's enough space for that new string to be appended to `stringvar`, otherwise buffer overrun occurs.

You can determine the length of `stringvar` (or indeed any string including double-quoted string literals) via: `int len = strlen( stringvar );`

If you had to write strlen(stringvar) yourself, you'd write:

```
int len = 0; while( stringvar[len] != '\0' ) len++;
```

Suppose, when `stringvar` contained `Hi!`, and you were about to append `" there"` to it, you decided that you'd like to overwrite the `'!'`, giving `"Hi there"`:

```
stringvar[strlen(stringvar)-1] = '\0';  // chop off last char of non-empty string
strcat( stringvar, " there" );
```

There's one more <`string.h`> function that I'd like to mention: sprintf(). This is a version of printf() which does *formatted output into a string buffer*, overwriting the existing contents:

```
int x = 42; int y = 7;
sprintf( stringvar, "debug: x=%d, y=%d", x, y );
```

then the `stringvar` array will contain the string `"debug: x=42, y=7"`.

Gotcha: As usual, it's your responsibility to ensure that there's enough space for the formatted output string to be copied into `stringvar`.