String handling routines in the C standard library

The C standard library allows us to manage strings in a few ways. It also allows conversion between strings and numbers.

We'll cover the following

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

I suggest you consult a reference source (or Wikipedia, e.g. C String Handling) for a list (it's relatively long) 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.

An example: Concatenating two strings

```
#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;
}
```

Comparing two strings

Importantly, you cannot simply use the `==` operator to test whether two strings are equal. Remember, strings are arrays of characters. You have to use a special string handling function to test equality of two strings, since it has to do a “deep” comparison, comparing each element against each other. Here’s how you 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;
}
```

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

Converting strings to and from numeric types

Strings to numbers

There are several functions to convert strings to numeric types like integers and floating-point numbers. You will need to `#include <stdlib.h>` at the top of your program.

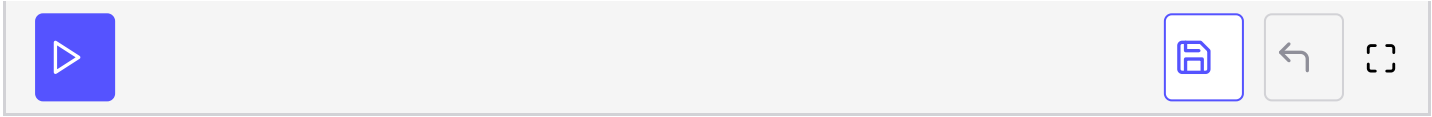
- `double atof(s)` converts the string pointed to by `s` into a floating-point number (a double), returning the result
- `int atoi(s)` converts string `s` into an integer

There are a host of others, again I suggest consulting a reference source for a comprehensive list.

Numbers to strings

The 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 it used much like the `printf()` function we have seen before, but instead of printing something to the screen, `sprintf()` "prints" something to a character string. Here's how to use it:

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



Note how on lines 6 and 7 when `s1` and `s2` are declared, I declare them as character arrays large enough to hold 256 characters. If you try to `sprintf()` to a string that is not big enough to hold what you're trying to put into it, then you will end up writing values beyond the end of the string, and onto who knows what, in memory. If you are dealing with strings that you know will be relatively short (things like filenames, 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 not to have to worry too much about 256 bytes here and there.

Numbers to string II (slightly esoteric)

There is, however, a way to do this 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 have a look at it.

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. Then we will use `malloc()` to allocate a new string (character array) of that length. 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 recieve the converted numeric: not too small, and not too big.

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

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

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;
}
```

Note on lines 10 and 15, where we use `snprintf()`, we are passing `NULL` as the first 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.

Thus `snprintf()` can be thought of as a “safe” version of `sprintf()` in that you **know** for sure that you will never write out more than the maximum number of bytes you ask for. Thus you can avoid over-writing past the end of your destination string buffer. The `snprintf()` function will return as its return value, 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. So 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 lines 11 and 16, we can use `malloc()` 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. Go to the next lesson to learn how.