Strings

String constants

s1.c

```c
/* Demo: string constant */
#include <stdio.h>

int main() {
    printf("Hi\n");
}
```

String constants are in double quotes

A backslash `\` is used to include 'special' characters, with `\\` used to include a backslash, e.g. `"H\\i"`

In `printf` ***only***, `%` is used to print special values, with `%%` to print a percent sign, e.g. `printf("100%%\n")`

Strings as arrays

s2.c

```
...
int main() {
    char hi[] = "Hi\n";
    printf("%s", hi);     (hi already has a newline)
}
```

A string is an array of characters – it can be initialised with a string constant, and printed with `%s`

This version fails, though!

```
char hi[3] = "Hi\n";
```

Marker character

s3.c

```
...
int main() {
    char hi[] = {'H', 'i', '\n', '\0'};
    printf("%s", hi);
}
```

The characters in a string must have a marker character `\0` (= character with code `0` = null character) at the end

This version works!

```
char hi[4] = "Hi\n";
```

Marker not counted

s4.c

```c
/* Demo: length of string */
#include <stdio.h>
#include <string.h>

int main() {
    char hi[] = {'H', 'i', '\n', '\0'};
    int n = strlen(hi);
    printf("%d\n", n);
}
```

This prints 3, not 4

So the \0 marker isn't counted in the length

First marker taken

s5.c

```c
/* Demo: length of string */
#include <stdio.h>
#include <string.h>

int main() {
    char hi[] = {'H','i','\n','\0','L','o','\n','\0'};
    int n = strlen(hi);
    printf("%d\n", n);
}
```

This prints 3, not 7 or 8

So a string ***isn't*** an array of characters, a string is ***contained*** in an array of characters

Random length

s6.c

```c
/* Demo: random length of string */
#include <stdio.h>
#include <string.h>

int main() {
    char hi[] = {'H', 'i', '\n'};
    int n = strlen(hi);
    printf("%d\n", n);
}
```

This prints 3, or 42, or a very big number, or crashes, depending on what is in memory after the array

So a string ***is not valid*** or reliable without a `\0` marker

Characters

A `char` is an ASCII character, i.e. one byte (`0..127`)

It is just a small integer, e.g. `'a'` is a synonym for the number `97`, and `'\0'` is a synonym for the number `0`

Note that `'0'` is a synonym for `48`, not `0`

Use `digit-'0'` to convert a digit into a number, and `letter-'a'+'A'` to convert a letter to upper case

International characters

Any character which is not in the American ASCII set is 'international': in English, that's `£`, `©`, `®`...

UTF-8 is the 'magic' you need (see aside: characters)

Code `printf("Cost: 10£ ...")` works (with `£` being two characters from C's point of view)

This works ***if*** your editor and terminal window are set up properly

Historical notes

The marker idea (instead of a length at the beginning) was probably to save space when memories were tiny: it involves an 'expensive' loop to find the length, and it prevents strings from containing a `\0` character

`strlen` and many other functions have rubbish names because there used to be a 6-letter limit (now its 31)

C has a 'rule' that anything built-in takes one processor instruction (ish) and anything longer needs a function, so `strlen` is a function because it needs a loop

Documentation

To find the documentation for the `strlen` function, type `C strlen` into Google

With luck, you find something like this:

```
#include <string.h>
...
size_t strlen(const char *str);
```

Sometimes the library to use (in this case `string`) is not very explicit

Sizes

The declaration/signature/prototype for `strlen` is

```
size_t strlen(const char *str);
```

`size_t` is a synonym for a suitable integer type on your computer for holding sizes, probably `long`

This does ***not*** work, because `%d` expects an `int`

```
printf("%d\n", strlen(hi));
```

Coercion

This is recommended

```
int n = strlen(hi);
printf("%d\n", n);
```

Technically, the definition of `n` involves a conversion ('coercion') from `size_t` to `int`

You are telling the compiler that you know the length will be small enough to fit in an `int`

Const

The declaration for `strlen` is

```
... strlen(const char *str);
```

`const` means that the `strlen` function promises not to change the string that you give it

Char pointer

The declaration for `strlen`, leaving out `const` is

```
... strlen(char *str);
```

`char *str` means that the argument variable `str` has type `char *` which is 'pointer to character'

For most purposes, C treats 'array of characters' and 'pointer to the first character' the same, so read this as:

```
... strlen(char str[]);
```

Warning

In this unit, before getting to pointers, you can almost always define strings as character arrays (`char s[]`)

However, in documentation, books, tutorials, blogs, on stack overflow, etc., pointer notation is used (`char *s`)

We will clear this up when we get to pointers

There is one place where you have to use pointer notation, not array notation, even now, that is when reading command line arguments in the `main` function

Main

args.c

```
...
int main(int n, char *args[n]) {
    for (int i=0; i<n; i++) {
        printf("Arg %d is %s\n", i, args[i]);
    }
}
```

Run this with `./args`, `./args a`, `./args a b`

The `args` array holds the words you typed on the command line, with `args[0]` being the program name, probably expanded into a full file path

The explanation will come in a later chapter

Returning strings

For the time being, you can't write functions which return strings

That's because we are using array notation, and C doesn't allow raw arrays to be returned from functions

Later, using pointer notation, we will effectively be able to return strings and arrays from functions

Comparing strings badly

Try this:

compare.c

```
...
int main() {
    char s1[] = "x";
    char s2[] = "x";
    if (s1 == s2) printf("Same\n");
    else printf("Different\n");
}
```

This prints out `Different`, because `==` means 'identical' not 'same contents'

Comparing contents involves a loop, so it's a function

Comparing strings well

This works:

compare2.c

```
...
int main() {
    char s1[] = "x";
    char s2[] = "x";
    if (strcmp(s1,s2) == 0) printf("Same\n");
    else printf("Different\n");
}
```

`strcmp` returns `0` if the strings are the same, or a negative or positive number if the first is earlier or later in lexicographic order than the second

Copying a string

copy.c

```c
/* Demo: copy a string */
#include <stdio.h>
#include <string.h>

int main() {
    char s1[] = "Hi\n";
    char s2[4];
    strcpy(s2, s1);
    printf("%s", s2);
}
```

`strcpy(to,from)` copies a string from one array to another

Joining strings

join.c

```
...
int main() {
    char s1[] = "Hi\n";
    char s2[] = "Lo\n";
    char s3[7];
    strcpy(s3, s1);
    strcat(s3, s2);
    printf("%s", s3);
}
```

`strcat(to,from)` joins `from` onto the end of `to`

The length of `s3` could be calculated:

```
char s3[strlen(s1) + strlen(s2) + 1];
```

Printing strings

sprint.c

```
...
int main() {
    char s1[25];
    sprintf(s1, "The square of %d is %d", 37, 37*37);
    printf("%s\n", s1);
}
```

`sprintf(s,...)` prints into `s`, with other arguments the same as `printf`

You can use `snprintf(NULL,0,...)` (C11) to precalculate the length

Challenges

One of the main challenges of C, compared to other languages, is that it has:

- undefined behaviour
- unspecified behaviour
- implementation-defined behaviour
- locale-specific behaviour

See appendix sections J.2, J.1, J.3, J.4 in the C11 standard for details

Undefined behaviour

The program is technically incorrect, but the platform (compiler plus run time system plus libraries etc.) may not detect it, and the result is unpredictable

The program may do the right thing, or some random wrong thing, or crash

Example: accessing a char past the end of a string

Unspecified behaviour

The program is technically incorrect, the standard says the platform has a limited choice of what to do, but the programmer has no official way of finding out what is chosen so must still avoid the situation

Example: accessing a local variable before it has been initialised

The standard says ***some*** value must be provided

Implementation-defined

The program is technically correct, the standard says the platform has a limited choice of what to do, and that choice is supposed to be documented

Examples: handling of international UTF-8 text, whether `stdout` is buffered, whether `char` is signed or not, how many bits `int` and `long` have (beyond 16/32)

Nobody knows where to find this documentation!

So if you can't avoid these things, or test them in the program, you need to document the assumptions your program makes

Locale-specific

The program is technically correct, but the behaviour depends on where in the world you are

Examples: the default human language, money, timezones

The platform is required to provide information about the locale to your program

But the user may need to set up the environment properly for this to work, so you need to provide user documentation