4. Strings

Strings are `char` arrays with a special null terminator to mark the end of the text. It is `\0`.

String literals are strings that cannot be changed. They are initialised as - `char *text = "hello";`

String variables are initialised as - `char text[] = "hello";` The size of the `char` array when a string is initialised like this is equal to one more than the number of characters. This additional index is for the null terminator. If the size of the string is explicitly stated and the initialisation list is smaller than this stated size, then the remaining indices are initialised to the null terminator. If the initialiser list is the same length as the stated size, then the null terminator isn't added.

When a string literal is used to initialise a char array, the string literal is first stored in the data section of memory, and then copied to the variable to initialise it. The size of a `char` is guaranteed by the standard to be one byte.

`string.h` contains all the C string functions.

Command Line Arguments

Command line arguments are passed to the `main` function as an array of strings. `argv[0]` holds the program's name. `argv[argc]` is guaranteed to equal `NULL`.

Converting Strings to Integers

The library function `strtol` from `stdlib.h` can be used to parse strings as long.

The prototype of `strtol` is `long strtol(const char *restrict str, char **restrict endptr, int base);`

`strtol` converts the value in the string `str` to an integer of the base `base`. If `endptr` is not null, `strtol` stores the address of the first non-digit character in it.

The string may begin with an arbitrary amount of white space followed by a single optional `+` or `-` sign. If base is `0` or `16`, the string may then include a `0x` prefix, and the number will be read in base `16`; otherwise, a `0` base is taken as `10` unless the next character is `0`, in which case it is taken as `8`.

If no conversion could be performed, `0` is returned and the global variable `errno` is set to `EINVAL`. If an overflow or underflow occurs, `errno` is set to `ERANGE` and the function return value is clamped.

Useful Standard Library Functions

`strlen`

The prototype for `strlen` is `size_t strlen(const char *s);`

`strlen` computes the length of the string `s`. It returns the number of characters that precede the null terminator.

`strcpy` and `strncpy`

The prototype for `strcpy` is `char *strcpy(char * dst, const char * src);`

`strcpy` copies the string `src` to `dst` (including the terminating '\0' character.)

The prototype for `strncpy` is `char *strncpy(char * dst, const char * src, size_t len);`

`strncpy` copies at most len characters from `src` into `dst`. If `src` is less than len characters long, the remainder of `dst` is filled with `\0` characters. Otherwise, `dst` is not null terminated.

The source and destination strings should not overlap, as the behaviour is undefined. The input `dst` need not be null terminated as it will be overwritten anyways. For safety, the null terminator should be manually added to the end of `dst`.

`strcpy` and `strncpy` return `dst`.

Safe Usage

```
strncpy(str1, str2, sizeof(str1));
str1[sizeof(str1) - 1] = '\0';
```

`strcat` and `strncat`

The prototype for `strcat` is `char *strcat(char *restrict s1, const char *restrict s2);`

The prototype for `strncat` is `char *strncat(char *restrict s1, const char *restrict s2, size_t n);`

`strcat` and `strncat` append a copy of the null-terminated string `s2` to the end of the null-terminated string `s1`, then add a terminating `\0`. The string `s1` must have sufficient space to hold the result. `strncat` appends not more than n characters from `s2`, and then adds a terminating `\0`.

The source and destination strings should not overlap, as the behaviour is undefined.

`strcat` and `strncat` return the pointer `s1`.

`strdup` and `strndup`

The prototype for `strdup` is `char *strdup(const char *s1);`

`strdup` allocates sufficient memory for a copy of the string `s1`, does the copy, and returns a pointer to it. The pointer may subsequently be used as an argument to `free`.

The prototype for `strndup` is `char *strndup(const char *s1, size_t n);`

`strndup` copies at most n characters from the string `s1` and always null terminates the copied string.

If insufficient memory is available, `NULL` is returned and `errno` is set to `ENOMEM`.

`strchr` and `strstr`

The prototype for `strchr` is `char *strchr(const char *s, int c);`

`strchr` locates the first occurrence of `c` (converted to a char) in the string pointed to by `s`. The terminating null terminator is considered to be part of the string; therefore if `c` is `\0`, the functions locate the terminating `\0`. It returns a pointer to the located character, or NULL if the character does not appear in the string.

The prototype for `strstr` is `char *strstr(const char *haystack, const char *needle);`

`strstr` locates the first occurrence of the null-terminated string `needle` in the null-terminated string `haystack`. If `needle` is an empty string, `haystack` is returned; if `needle` occurs nowhere in haystack, `NULL` is returned; otherwise a pointer to the first character of the first occurrence of `needle` is returned.

`strcmp` and `strncmp`

The prototype of `strcmp` is `int strcmp(const char *s1, const char *s2);`

The prototype of `strncmp` is `int strncmp(const char *s1, const char *s2, size_t n);`

The `strcmp` and `strncmp` lexicographically compare the null terminated strings `s1` and `s2`.

`strncmp` compares not more than n characters. Because `strncmp` is designed for comparing strings rather than binary data, characters that appear after a '\0' character are not compared. They return an integer greater than, equal to, or less than `0`, according as the string `s1` is greater than, equal to, or less than the string `s2`. The comparison is done using unsigned characters, so that `\200` is greater than `\0`.
