Lecture 12: The C Standard Library Part 1

COMP26020 Part 1 (C) Lecture Notes

Pierre Olivier

These notes summarise the important points mentioned in the lectures. They are supposed to be a help for revising and not a way to avoid attending the live lectures and watching the videos. In other words, live lectures and videos may include examinable content that is not present in these notes.

The slides for this lecture are available here:
https://olivierpierre.github.io/comp26020-lectures/12-standard-library-1.

Videos and recordings of live sessions can be found on the video portal: https://video.manchester.ac.uk/lectures.

In this lecture we start discussing the C standard library, a set of functions you can use in your C programs to realise various low-level tasks. We cover functions regarding string and memory manipulation, console input, math and time.

Manual Pages

We already came across a few functions of the standard library, such as `printf` or `malloc`. For every function of the standard library, `man <function name>` in a Linux terminal will give you everything you need to know to use that function: function description, prototype, required headers, return value. Manual pages are also available online, e.g. for `printf`: https://man7.org/linux/man-pages/man3/printf.3.html.

String Copy

The `=` operator should not be used to copy a string in C. Remember that strings are arrays, and arrays are pointer. So effectively with `=` one just creates a copy of the pointer that points to the same array's content.

A proper string copy is realised with `strcpy`:

```
char *strcat(char *dest, const char *src);
```

It takes the destination string as first parameter, and the source string as parameter. The `const` keyword is simply there to indicate that `strcpy` will not modify the `src` parameter. Care should be taken with respect to the sizes of the strings: if the destination buffer is smaller than the source string, it will overflow and a memory error will occur.

`strncpy` is somehow safer than `strcpy` as it accepts a 3rd argument, `n`, and copies only up to `n` bytes:

```
char *strncat(char *dest, const char *src, size_t n);
```

`n` can for example be set to the size of the destination space to avoid overflows.

Note that both `strcpy` and `strncpy` do copy the termination character. Below is an example of usage of both functions:

```
#include <string.h> // needed for strcpy/strncpy
/* ... */

char *string1 = "hello";
char *string2 = string1;   // this is not a string copy!
char string3[10];          // allocated space of 10 bytes, it's called a buffer
```

```c
/* not super safe, what if the length of string1 is larger than the 10 bytes of string3? */
strcpy(string3, string1);
/* better */
strncpy(string3, string1, 10);
```

In this example, `string3` corresponds to a region of contiguous memory, writable, used to hold data: it is called a **buffer**.

String Concatenation

To concatenate two strings, use `strcat` or `strncat`:

```c
char *strcat(char *dest, const char *src);
char *strncat(char *dest, const char *src, size_t n);
```

`strcat` takes two parameters, a destination and source strings. In effect, it concatenates the source at the end of the destination. Once again the programmer needs to consider the buffer sizes. Same as `strncpy`, `strncat` can be used to concatenate up to a given number of characters. Here is an example of usage of both functions:

```c
#include <string.h>
/* ... */
char world[6] = "world";
char s1[32] = "hello ";
char s2[32] = "hello ";
strcat(s1, world);       // possibly unsafe
strncat(s2, world, 32);  // better
```

Format-based String Creation

`sprintf` is a useful function to create a string that is filled with values from variables of various type. This is done with `printf`-like specifiers:

```c
int sprintf(char *str, const char *format, ...);
```

It takes as parameter the destination string. Then, a constant string filled with text as well as specifiers, same as with `printf`. Then, a list of variables or expression corresponding to the specifiers, like with `printf`. The destination string should be large enough to avoid memory errors. Here is an example of usage:

```c
#include <string.h>

int main(int argc, char **argv) {
    int a = 12;
    float b = 4.5;
    char *s = "hello";
    char string[64];

    sprintf(string, "a is %d, b is %f, s is %s\n", a, b, s);
    printf("%s", string); // a is 12, b is 4.500000, s is hello
    return 0;
}
```

String Length and Comparison

To get the size of a string, in characters, use `strlen`:

```c
size_t strlen(const char *s);
```

Note that this function does not count the termination character. To compare two strings, use `strcmp`:

```c
int strcmp(const char *s1, const char *s2);
```

It returns `0` if the string matches, a negative number if `s1` is before `s2` in the lexicographical order, and a positive one if `s1` is after `s2`. Here is an example of usage of both functions:

```
#include <string.h>
/* ... */
char *s1 = "hello"; char *s2 = "hello";
char *s3 = "not hello";

printf("strcmp(s1, s2) returns: %d\n", strcmp(s1, s2));   // 0
printf("strcmp(s1, s3) returns: %d\n", strcmp(s1, s3));   // -6

printf("length of s3: %d\n", strlen(s3)); // 9
```

For more information about string manipulation functions type `man string` in a terminal. Further resources are also available online[1]

Console Input

`fgets` receives a string from the console:

```
char *fgets(char *s, int size, FILE *stream);
```

It takes the destination buffer as first parameter, the size of that buffer, and the special keyword `stdin` that indicate the standard input. The program will then read what is typed on the keyboard until the user presses enter.

To input numbers one should rather use `scanf.=`:

```
int scanf(const char *format, ...);
```

It takes a `printf`-like format string, with specifiers for the variable we wish to update the value based on the user input. Then we have the addresses of the variables in question. Below is an example of usage of both functions:

```
int main(int argc, char **argv) {
    int int1, int2;
    double double1;
    float float1;
    char s[128];

    printf("Please input a string:\n");
    fgets(s, 128, stdin);

    printf("Please input an integer:\n");
    scanf("%d", &int1);

    printf("Please input a float:\n");
    scanf("%lf", &double1); /* make sure to us %lf for double and %f for float */

    printf("Please enter an integer and a float separated by a space\n");
    scanf("%d %f", &int2, &float1);

    printf("You have entered: %d, %d, %lf, %f, and %s\n", int1, int2, double1,
            float1, s);
    return 0;
}
```

Writing Bytes to Memory, Copying Memory

`memset` repeatedly writes the byte `c`, `n` times starting from address `s`:

```
void *memset(void *s, int c, size_t n);
```

[1] https://en.wikibooks.org/wiki/C_Programming/String_manipulation.

It can be useful for example when you want to zero out a buffer.

`memcpy` copies a buffer into another one:

```
void *memcpy(void *dest, void *src, size_t n);
```

An example of usage of both functions is presented below:

```c
#include <stdio.h>
#include <string.h>   // needed for memcpy and memset
#include <stdlib.h>

int main(int argc, char **argv) {
    int buffer_size = 10;

    char *ptr1  = malloc(buffer_size);
    char *ptr2 = malloc(buffer_size);

    if(ptr1 && ptr2) {
        memset(ptr1, 0x40, buffer_size);  // 0x40 is ascii code for @
        memcpy(ptr2, ptr1, buffer_size);
        for(int i=0; i<buffer_size; i++) {
            printf("ptr1[%d] = %c\n", i, ptr1[i]);
            printf("ptr2[%d] = %c\n", i, ptr2[i]);
        }
        free(ptr1); free(ptr2);
    }
    return 0;
}
```

Math Functions

There are a few: square root, power, cosinus, amongst many others. Most have a float and double version, for example `ceil` and `ceilf`. Here is an example with a few calls to some math functions:

```c
// When compiling a program using math.h, // use -lm on the command line:
// gcc program.c -o program -lm
#include <stdio.h>
#include <math.h> // needed for math functions

int main(int argc, char **argv) {
    printf("ceil 2.5:  %f\n", ceil(2.5));
    printf("floor 2.5: %f\n", floor(2.5));
    printf("2^5:       %f\n", pow(2, 5));
    printf("sqrt(4):   %f\n", sqrt(4));
    return 0;
}
```

The full list of available functions can be found online[2]. Note that when building a program using math functions, you need to use a specific `-lm`flag when you call the compiler:

```
gcc src.c -o program -lm
```

Sleeping

To have the program wait for a given time one can use `sleep` for sleeping in seconds, and `usleep` for sleeping in microseconds:

[2] https://cplusplus.com/reference/cmath/

```
unsigned int sleep(unsigned int seconds);
int usleep(useconds_t usec);
```

It is useful in certain applications that generally have an infinite main loop, but don't want to hang 100% of the CPU, such as servers. Here is a code example:

```
#include <stdio.h>
#include <unistd.h> // needed for sleep and usleep

int main(int argc, char **argv) {
    printf("hello!\n");

    printf("Sleeping for 2 seconds ...\n");
    sleep(2);

    printf("Now sleeping for .5 seconds ...\n");
    usleep(500000);
    return 0;
}
```

Current Time

To get the current time the`time` function can be used:

```
time_t time(time_t *tloc);   // time_t is generally a long long unsigned int
```

A simple way to call it is with `NULL` as a parameter. It returns the number of seconds elapsed since the 1st of January 1970, which is a standard timestamp for UNIX computers.

Measuring Execution Time

To get a more precise measurement of the current time we can use gettimeofday:

```
int gettimeofday(struct timeval *tv, struct timezone *tz);
// struct timeval {
//     time_t      tv_sec;     /* seconds (type: generally long unsigned) */
//     suseconds_t tv_usec;    /* microseconds (type: generally long unsigned) */
//  };
```

It takes a pointer to a `struct timeval` as parameter. This struct has two fields, one for seconds, `tv_sec` and the other for microseconds, `tv_usec`. The second parameter should always be NULL. Like `time` it also returns the time elapsed since the 1st of January 1970. `gettimeofay` is very useful to measure execution time, as shown in this example:

```
#include <stdio.h>
#include <sys/time.h> // needed for gettimeofday

int main(int argc, char **argv) {
    struct timeval tv, start, stop, elapsed;

    gettimeofday(&tv, NULL);
    printf("Seconds since the epoch: %lu.%06lu\n", tv.tv_sec, tv.tv_usec);

    gettimeofday(&start, NULL);
    for(int i=0; i<1000000000; i++);
    gettimeofday(&stop, NULL);

    timersub(&stop, &start, &elapsed);
    printf("Busy loop took %lu.%06lu seconds\n", elapsed.tv_sec,
            elapsed.tv_usec);
```

```
    return 0;
}
```

3 `struct timeval` variables are declared. A first call to `gettimeofay` is done before the code for which we want to measure the execution time. Here it is just a busy loop. A second call to `gettimeofday` is done right after that. Then `timersub` is called to compute the difference between the two timestamps. Notice how a `struct timeval` is printed, with `"%lu.%06lu"`: this forces the inclusion of up to six leading zeroes if the microsecond value is inferior to one million.