Piscine — Tutorial D8

version **#v3.2.0**

Assistant C/UNIX 2021 <assistants@tickets.assistants.epita.fr>

Copyright

Contents

*https://intra.assistants.epita.fr

1 Data structures - Linked lists

1.1 Explanation

Although structures and pointers are useful by themselves, they become even more powerful when tied together. Since a pointer is a value like any other with a fixed size, it can be used as a field in a structure definition.

```
struct my_struct
{
    int a_field;
    float *another_field; // why not?
};
```

However, it is impossible for a structure to refer to itself.

```
struct impossible
{
    int foo;
    struct impossible bar; // Impossible!
};
```

To understand why, ask yourself the question: what would the size of this structure be? To get it, it must determine the size of its own fields, but to get the size of `bar` we will need the size of the structure itself...

However, nothing prevents us from using a pointer to a struct of the type being defined because the size of a pointer is known and is always the same!

```
struct possible
{
    int foo;
    struct possible *bar; // Totally possible: we know the size!
};
```

We just saw a new powerful concept: **recursive** structure definitions, which use a pointer to an instance of the structure in its definition.

A simple example of this concept is the implementation of a singly linked list.

A singly linked list is either:

- an empty list
- an element followed by a list

1.2 Structure declaration

How to represent a linked list in C? Let's consider alternatives:

- If a list is empty, it does not hold any information and thus it's useless to allocate memory for it: just use `NULL` to represent an empty list.
- Otherwise, it holds an element and the next node of the list.

```c
struct list
{
    int data;
    struct list *next;
};
```

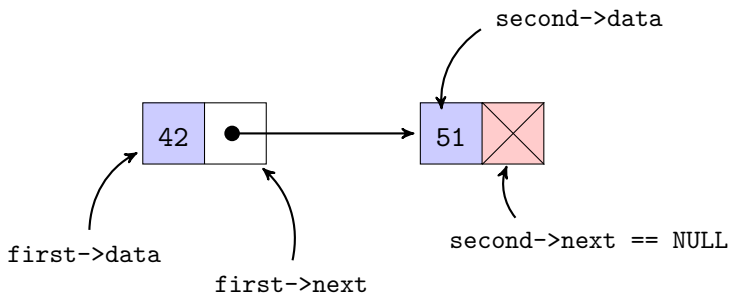
For example, to represent the list `42 -> 51 -> NULL`, we can do:

```c
struct list *first = NULL;
struct list *second = NULL;

second = malloc(sizeof(struct list));
second->data = 51;
second->next = NULL; // The last element is followed by an empty list

first = malloc(sizeof(struct list));
first->data = 42;
first->next = second; // The first element is followed by the second
```

As you can see, we use the notation `->` to access the content of our variables as they are **pointers** to structures. This is syntactic sugar for:

```c
(*second).data = 51;
second->data = 51; // Easier to use this notation
```

The list represented below:



Be careful!

When manipulating a list do not lose your pointer to the head.

This concept is a bit hard to understand, but with experience, you will learn how to correctly handle it. Time to practice!

1.3 Exercise

1.3.1 Integer linked list

A header (`list.h`) containing all the required functions is provided on the intranet.

In this exercise, you will have to implement a linked list, along with its manipulation operations.

- An empty list is represented by a `NULL` pointer.

1.4 list_prepend

```c
struct list *list_prepend(struct list *list, int value);
```

This function must insert a node containing `value` at the beginning of the list. The function must return the new list, or `NULL` if an error occured.

1.5 list_length

```c
size_t list_length(struct list *list);
```

This function returns the length of the list.

1.6 list_print

```c
void list_print(struct list *list);
```

This function displays the elements of the list, separated by a space, ended by a newline.

If the list is empty, nothing is displayed.

1.7 list_destroy

```c
void list_destroy(struct list *list);
```

This function releases all the memory used by `list`.

1.8 Generic linked lists

Normally, you have just implemented a linked list containing integers. But what if we wanted to store not only integers, but other types of data? How can we create a linked list that can be used for any data type?

In C, you can use void pointers as they can be used to point to any data type. We also know the size of a pointer, so we can always know the size of the structure to allocate it. Thus, we can represent a generic linked list like:

```c
struct list
{
    // Any data type can be stored in this node
    void *data;
    struct list *next;
};
```

1.9 Exercise

In this exercise, you will have to implement a generic linked list, along with its manipulation operations.

- An empty list is represented by a `NULL` pointer.

1.10 list_prepend

- **Authorized functions:** `malloc(3)`, `memcpy(3)`

```c
struct list *list_prepend(struct list *list, const void *value,
                          size_t data_size);
```

This function must insert a node containing `value` at the beginning of the list. The function must return the new list, or `NULL` if an error occured. You can use `memcpy(3)` to copy `value` into the `data` field of the list struct.

1.11 list_length

- **Authorized functions:** none.

```c
size_t list_length(struct list *list);
```

This function returns the length of the list.

1.12 list_destroy

- **Authorized functions:** `free(3)`

```
void list_destroy(struct list *list);
```

This function releases all the memory used by `list`.

2 Reading from standard input

You will often find yourselves in cases where you want to read from the standard input. Several functions can handle this operation: you will discover them progressively.

2.1 `getchar(3)`

Prototype:

```
int getchar(void);
```

Example usage:

```
#include <stdio.h>

int main(void)
{
    puts("Press any key :");
    char c = getchar();
    printf("You pressed the button %c\n", c);
    return 0;
}
```

```
42sh$ gcc -Wextra -Wall -Werror -std=c99 -pedantic -o pressed_button pressed_button.c
42sh$ ./pressed_button
Press any key :
azertyuiop
You pressed the button a
```

No matter how much you write, `getchar(3)` will only read one character. To read user input with a specific format, we use the `scanf(3)` function which works like `printf(3)`, but for the input stream.

2.2 `scanf(3)`

Prototype:

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

It uses the same format tags as `printf(3)`, the difference is that we have to put user input in some variable, and to do so, we need to use a pointer.

Example usage :

```
#include <stdio.h>
int main(void)
{
    int value;
    puts("Write some number:");
    if (scanf("%d", &value) != 1)
        return 1;
    printf("You wrote number: %d\n", value);
    return 0;
}
```

Here, `scanf(3)` will read standard input, get the value, and write it at the **address** of `value`. If `scanf(3)` does not manage to read the value, and thus does not return 1, the program stops and returns 1. To get more information about `scanf(3)` return values, you can check **man 3 scanf**.

Output:

```
42sh$ gcc -Wextra -Wall -Werror -std=c99 -pedantic -o scanf scanf.c
42sh$ ./scanf
Write some number:
42
You wrote number: 42
```

If the user didn't write anything, or the input stream closed unexpectedly, `scanf(3)` returns `EOF` (aka *End-Of-File*). `EOF` is a special value, which means this is the end of the stream, in this case, it means the user closed the standard input by pressing *Ctrl^D*.

You can also get user input, split by spaces and newlines using the `%s` format tag.

Example usage :

```
#include <stdio.h>
int main(void)
{
    char buffer[32];
    puts("Write something :");
    if (scanf("%s", buffer) != 1)
        return 1;
    printf("You wrote : %s\n", buffer);
    return 0;
}
```

```
42sh$ gcc -Wextra -Wall -Werror -std=c99 -pedantic -o hello_world hello_world.c
42sh$ ./hello_world
```

(continues on next page)

(continued from previous page)

```
Write something :
Hello World!
You wrote : Hello
```

Here we created a 32 byte buffer, but what if the user's input was more than 31 characters? (Don't forget the **\0** at the end). It would have created an overflow, leading to undefined behavior.

You should do as follow:

```c
#include <stdio.h>
int main(void)
{
    char buffer[32];
    puts("Write something :");
    if (scanf("%31s", buffer) != 1)
        return 1;
    printf("You wrote : %s\n", buffer);
    return 0;
}
```

Here `scanf(3)` will read **at most** 31 characters. The main rule is: **Never trust user input**.

2.3 Exercises

2.3.1 Mirror

Write a program which writes user input on standard output with `scanf` and `printf` as long as the user didn't close the stream.

```
42sh$ gcc -Wextra -Wall -Werror -std=c99 -pedantic -o mirror mirror.c
42sh$ ./mirror
Hello
Hello
World
World
^D
42sh$
```

2.3.2 My average

Write a program that takes a number `n` from its standard input. Afterwards, take `n` numbers from its standard input, compute the average of those numbers, and print it on standard output with a precision of two decimal places.

Example:

```
42sh$ gcc -Wextra -Wall -Werror -std=c99 -pedantic -o my_average my_average.c
42sh$ ./my_average
4
```

(continues on next page)

(continued from previous page)

```
5
10
15
86
29.00
```

3 Standard library functions

The (far from exhaustive) list of syscalls we saw is not the only way to access files. Higher level functions, from the standard C library provide some similar functionalities with some advantages. Most of the functions we will now see are declared in `stdio.h`.

Instead of using file descriptors, these functions work with a pointer to a `FILE`. `FILE` is a structure which has the corresponding file descriptor as one of its fields. However, `FILE` is implementation defined, so depending on which standard library you are using, the fields may not be the same. Only the behavior of the functions is guaranteed.

3.1 Predefined FILE variables

The standard library defines three macros corresponding to the standard stream, already open by the time your `main` function is called. They are `stdin`, `stdout`, and `stderr`, for your program's standard input, standard output, and standard error output respectively.

3.2 `fopen(3)`

The equivalent of `open(2)` is the `fopen(3)` function. It opens a file and returns a `FILE *`. However, the signature is quite different:

```
FILE *fopen(const char *path, const char *mode);
```

`path` is, surprisingly, the path to the file (absolute or relative to `$PWD`). The second one however, is a string rather than an `int`. This string describes the mode for opening the file, the possible values being listed in the `fopen(3)` man page, among which:

- `"r"` is read-only;
- `"r+"` is read-write;
- `"a"` is append (i.e. write only, but with the cursor at the end of the file).

The `FILE *` returned is important to be able to use the other library functions.

3.3 `fclose(3)`

`fclose(3)` closes a file opened with `fopen(3)`:

```
int fclose(FILE *fp);
```

The only parameter is the `FILE *` to close and free. The return value is `0` if all goes well. See the man page for possible errors.

3.4 `fread(3)`

The `fread(3)` function allows to read from a file, like `read(2)`. However, its use can *seem* more complex. Let's have a look at its signature:

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

Here, `ptr` is like `buf` for read, the buffer into which we will read the data. `stream` is the `FILE *` from which we will read. However, where `read(2)` only took one parameter to indicate the number of bytes to read, `fread(3)` takes two: `size` and `nmemb`. The first one is the size, in bytes, of one **element**, and `nmemb` is the **number of elements** to read. For instance, if you have an array of `23` `int` to read, `size` will be `sizeof(int)`, and `nmemb` will be `23`.

Why the difference? Because `fread(3)` returns the number of *elements* read, not the number of bytes.

3.5 `fwrite(3)`

As you might have guessed, `fwrite(3)` is to `fread(3)` what `write(2)` is to `read(2)`. The signature is like so:

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
```

The parameters are the same as `fread(3)`, only the `ptr` buffer contains the data to be written to `stream`.

`fwrite(3)` returns the number of elements written.

3.6 `fseek(3)` and `ftell(3)`

For `fseek(3)`, the signature is the same as `lseek(2)`, except that the file descriptor has been replaced by a `FILE *`:

```
int fseek(FILE *stream, long offset, int whence);
```

To get the read/write head position, instead of calling `fseek(3)` with dummy arguments, you can use `ftell(3)`.

3.7 `getline(3)` and `getdelim(3)`

As you saw `fread(3)`, is well-suited for binary streams. Most commonly however you will want to read a file line by line. These functions are exist for this purpose:

```
ssize_t getline(char **lineptr, size_t *n, FILE *stream);
ssize_t getdelim(char **lineptr, size_t *n, int delim, FILE *stream);
```

Reading the man page carefully is mandatory in order to use the functions correctly.

3.8 Exercises

3.8.1 Hidden message

Read the file `hidden_file` (in the given files), extract the hidden message and print it on stdout. The characters composing the message is each fourth character following a semicolon. Use fseek, getdelim and fwrite. There can not be more than 32 characters between two semicolons.

The only way out is through