Prog C — td4

version **#0.0.1-dirty**

Assistants C/Unix 2024 `<assistants@tickets.forge.epita.fr>`

Contents

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

1 Constness

The *C* language has a keywork `const`. This keyword is what we call a `type qualifier`. This is an additionnal information that we attached to a type and that is understand by the compiler. This keyword add the notion of **constness** in the *C* language.

When declaring a variable with the keyword `const`, the variable's value cannot be modified. The only value it can hold is its first value. We can compare `const` with a contract. The compiler ensures us that the variable's value cannot change during the execution of the program.

It is also a very useful information for readers to know which variable will change or not.

Let us take a look a some examples:

```c
const int bar; /* valid but will cause an error with the required flags */
const int foo = 5; /* valid */
char const c = 'c'; /* valid, keywords can be placed anywhere before the name */
```

The first example is valid. However, when using the flag `-Wall` it will add one warning. Either that `bar` is defined but not used or `bar` is used uninitialized.

While not initializing you variables, the compiler has the freedom to give the value it wants to your variables. Therefore, `bar` has a value that you do not know and would not be able to modify; which is perfectly useless.

Be careful!

Remember, always initialize your variables!

The second example is correct and serve its purpose, it is declared and initialized with a value that you specified.

The last example is correct syntaxically but not allowed by the coding style[1] at EPITA. You will need to respect it during the piscine and all along the year. For now, just remember it exists.

As said before, the constness is ensured by the compiler. Below is an example of the error message you will get if you try to modify a const variable.

```c
/**
** \file const.c
*/

int main(void)
{
    const int nb = 3;
    nb += 2;
}
```

```
42sh$ gcc const.c
const.c: In function 'main':
const.c:4:8: error: assignment of read-only variable 'nb'
  nb += 2;
```

[1] A coding style is a set of rules or guidelines used when writing the source code for a computer program. It enforces a standard for writing code, making it cleaner and more readable for any reader. At EPITA, the respect of the school's coding-style is enforced.

1.1 Constness and pointers

The `const` keyword can be easily used with pointers. The syntax is the following:

```c
const char * j = NULL;
int * const i = NULL;
```

You can now open `man 3 strlen`. The prototype of strlen is the following:

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

This function takes a `const` parameter. It means that this parameter cannot be modified in the function `strlen`. The purpose of `strlen(3)` is to return the length of a char array, it does not modify its content. Therefore, it is logical to specify to the compiler that it would not be modified.

To be more precise, the pointer `s` is not constant but the value it holds is. Indeed, `const char *` means that the `char` pointed by the pointer is constant. Hence, we can not modifiy the pointed value in any way. However, we can change `s`, the pointer itself. This means that we can assign a new pointer to `s`.

On the contrary, if we use the syntax `char * const`, the pointer is then constant but not the pointed value. We cannot change the pointer itself, but we can modify the value its holds.

A little reminder, in *C* all arguments are passed by copy. Therefore, any modification made to a parameter would not be persistant after the function call. That means that qualifying a parameter as `const` is kind of useless. However, that is not the case with pointer, the modification of the pointed value of a pointer is persistant after the function call. So, qualifying a pointer parameter as `const` has a real utility.

Be careful!

You need to be extra carefull with pointers. It can get quite confusing mixing `const`, pointers and function parameters. As a rule of thumb, you can remember the constness is associated to the nearest keyword leftwise.

1.1.1 Recap

Let us look at the following function:

```c
void weird_constness(const int *a, int * const b)
{
    *b = 1; /* works well because the pointed value of b is not const */
    b = NULL; /* generate a compilation error because b is const */

    *a = 1; /* generate a compiler error because the pointed value of a is const */
    a = NULL; /* works well because a is not const */
}
```

Try to play a little with this function to fully understand the difference.

2 Strings

2.1 ASCII

In **C** a variable of type `char` can take values from `-128` to `127`. Each value in the range of [0, 127] corresponds to a characted from the ASCII table.

ASCII is a standard to encode letters as numbers. Typing `man 7 ascii` in your terminal displays the ASCII table, in which you can see the number associated to each letter.

Going further...

The extended ASCII table associates number in the range of [0, 255]. Hence, following this table, each value of a `char` or `unsigned char` can be matched by a human readable character. Remember, `unsigned` specify just the range of the value possible that is any positive value for one type [0, nb_values - 1] but it does not define the total number of value available. Indeed, -125 will corresponds to the value 131 for a `unsigned char` and *ƒ* in the extended ASCII table.

We strongly recommend you take a look at the ASCII table and notice a few things:

- The character '0' does not have the value 0.
- Characters are sorted logically, 'a' to 'z' are contiguous, as well as 'A' to 'Z' and '0' to '9'.

The value of a `char` variable being a number, all arithmetic operators can be used on `char`. Variables of type `char` take any value possible for one byte. Therefore, they can be used as if there were smaller int. of 4 bytes (on the PIE).

```c
#include <stdio.h>

int main(void)
{
    char a = 'A';
    a += 32;

    if (a >= 97 && a <= 122)
    {
        puts("'a' has become a lowercase character!");
    }

    return 0;
}
```

It is very impractical to use ASCII codes instead of chars. Here is what we will prefer:

```c
#include <stdio.h>

int main(void)
{
    char a = 'A';
    a += 'a' - 'A';

    if (a >= 'a' && a <= 'z')
        puts("'a' has become a lowercase character!");
```

(continues on next page)

(continued from previous page)

```c
    return 0;
}
```

Examples:

```c
char a = 'a';
char b = 'b';
char z = 'z';
int result = (b - a) * z;
```

At the end of these instructions, the resulting value is `(98 - 97) * 122 = 122`, representing the character ‘z’ in the *ASCII table*.

2.1.1 Practice

Now can you tell if the following assertions are true or false?

```c
'a' > 'A'
'b' > 'a'
'0' == 0
```

What conditions should you use to check if a character is uppercase?

2.2 Strings

A string is a contiguous sequence of characters terminated by a null character. If their are multiple null characters in a sequence of characters, the first null character is considered as the end of the string.

In **C**, in order to represent strings, we use a **character array ended by a ‘\0’**, a special character symbolizing the end of the string.

Going further...

if you check the value of *\0* in the ascii table you can see it is 0.

There are many special characters, including:

`\n`	line break
`\t`	tabulation
`\0`	end of string character
`\\`	backslash (\)
`\"`	double quote
`\'`	single quote

Let’s look at a basic example:

```c
#include <stdio.h>

int main(void)
{
    char s[] =
    {
        't', 'e', 's', 't', '\0'
    };
    puts(s);
    return 0;
}
```

We can easily see that writing strings in this form is not practical at all. Fortunately for us, the **C** language provides a simple way to write strings: the **string literals** (or *constant strings*).

The following example is semantically identical to the one above. Moreover the termination character (*'\0'*) is automatically added at the end of the string:

```c
#include <stdio.h>

int main(void)
{
    char s[] = "test";

    puts(s);
    return 0;
}
```

Another advantage of string literals is that they can be passed directly to functions taking `char[]` as arguments!

Be careful!

A `char` is single quoted wheras a string is double quoted. Therefore, 'c' is a `char` of value 'c', while "c" is a string of length 1 with 'c' as it first character. It is also null terminated '0'.

```c
#include <stdio.h>

int main(void)
{
    puts("test");
    return 0;
}
```

Finally, the length of a string is the number of characters before `\0`.

```c
char str1[] = "Portable";
char str2[] = "Por\0table";
```

Try to print these two strings with `puts`. You will see that the first string has 8 characters while the second one has only 3 characters.

Be careful!

String literals are null-terminated (`'\0'` added automatically at the end), but arrays declared with braces are not.

Be careful when passing a string literal to a function like this:

```
foo("bar");
```

In this case, the string `bar` cannot be modified. If the `foo` function tries to change it, the program will crash. If you need to modify a string literal, you must put it in a variable of type char[] before calling the function:

```
char str[] = "bar";
foo(str);
```

The following function declarations are equivalent:

```
void foo(char arr[]);
void foo(char *arr);
```

2.3 Practice

2.3.1 Print Reverse

Write a function that prints the string given as argument in reverse order. You must follow this example:

```
#include <stdio.h>

void reverse_print(char s[], int size)
{
    // FIXME
}

int main(void)
{
    reverse_print("Hello World!", 12);
    puts(""); // To print a blank line at the end
}
```

You must obtain this

```
42sh$ gcc -Wall -Wextra -Werror -pedantic -std=c99 reverse.c -o reverse
42sh$ ./reverse
!dlroW olleH
```

Tips

Don't forget to check `man 3 putchar` for some help.

2.3.2 My Strlen

Goal

You must implement the following `strlen(3)` function :

```
size_t my_strlen(const char *s);
```

This function must have the same behavior as the `strlen(3)` function described in the third section of the man. Your code will not be tested with `NULL` pointers.

2.3.3 My Strupcase

Goal

You must implement the following `strupcase` function :

```
void my_strupcase(char *str);
```

This function changes all lower case ASCII letters into upper case. `NULL` pointer will not be tested.

Example

```
#include <stdio.h>
#include "my_strupcase.h"

int main(void)
{
  char str[] = "azerty1234XYZ &(";
  my_strupcase(str);
  printf("%s\n", str);
  return 0;
}
```

```
42sh$ ./my_strupcase | cat -e
AZERTY1234XYZ &($
```

I must not fear. Fear is the mind-killer.