Questions

What does %zu mean for printf a size_t?
What is the default standard used in gcc?
How does C allow memory for arrays, for example int A[5] = {1,2,3,4,5}? How does C decide the memory address at which it writes a variable's value?, ...
Variable statique: pas compris dans fonction ni hors de fonction...

Introduction

About C

Characteristics as a programming language

C is portable and simple:
- there is no dynamic type casting: if a float is converted to an int on a line, its always the case
- everything must be declared to make sure return types and argument types are correct)
- "compilation is one parsing run, that's why we must declare everything"

Memory layout

https://www.geeksforgeeks.org/memory-layout-of-c-program/amp/

Global: sont visibles et peuvent être modifiée à l'intérieure de nimporte quelle fonction (quelles soient déclarées hors ou dans une fonction, ce qui serait rare)
Static: hors dune fonction => pas accessible en dehors du fichier;dans une fonction => la variable nest pas recréée sur le stack à chaque appel de la fonction mais reste locale à la fonction

Compilers and libraries

gcc/clang/c99

clang is more verbose than gcc => useful for debugging

```
gcc -std=c89 -Wall -lm -o getting-started getting-started.c #gcc will automatically append ".exe" to output on windows
```

-std=c89 for 1989 C standard aka ANSI C
-Wall warns about anything unusual
-lm add standard math functions if necessary

-Werror to set all warnings as errors => to make compilation fail on any warning

GCC does not allow variable declarations in for loop initializers before C99

stdio.h

printf (man printf)

stdlib.h

size_t
EXIT_SUCCESS (=0)

Variables

int, double, size_t, char

```c
//int
int a = -3;
printf("value of a: %d\n", a); // -3

//double
double b = -3.33;
printf("value of a: %f\n", b); // -3.330000

//size_t
// integers in [0, SIZE_MAX] (integers can be negative)
// after SIZE_MAX is 0!
// before 0 is SIZE_MAX!
size_t b = SIZE_MAX; // from stdlib.h
printf("value of b: %zu\n", b); // 18446744073709551615
++b;
printf("value of b: %zu\n", b); // 0
--b;
printf("value of b: %zu\n", b); // 18446744073709551615

//char
// single quotes only!
// a char is not a string (char*)!
char c = "r"; // warning: single quote is for type char, double quotes for type string!!!
char c = 'r';

printf("value of c: %s\n", c);  // warning, c is of type char, not char*!
printf("value of c: %c\n", c);  // r
```

strings, pointers

```c
//string
// "a string is a pointer to a char"
// "abcde" is called a string literal.
// It is a pointer to 'a' and by writing this we also write 'b', 'c', ... at the next memory addresses!

char* d = 'aeogriero'; // error: single quote is for type char, double quotes for type char*!
char* d = 'a';         // warning: right value is a char, not a char*!

char* d = "abcde"; // ok
// this is a weird but valid C assignment.
// "abcde" is called a string literal.
// It is a pointer to 'a' and by writing this we also write 'b', 'c', ... at the next memory addresses!

printf("value of d: %p\n", d); // 00007FF7116332E0
printf("value pointed by d: %c\n", *d); // a
printf("value at the next address: %c\n", *(d + 1)); // b
printf("value at the next address: %c\n", *(d + 2)); // c
printf("value at the next address: %c\n", *(d + 3)); // d
printf("value at the next address: %c\n", *(d + 4)); // e
printf("string at address d: %s\n", d); // abcde (%s follows the chars!)

//pointer
char e = 'e';
char* q = e;   // warning: right side is not a pointer!
char* q = &e;  // ok

printf("address of e: 0x%p\n", &e); // 0x000000E478CFF82B
printf("value of q: 0x%p\n", q);    // 0x000000E478CFF82B (same as &e)
printf("address of q: 0x%p\n", &q); // 0x000000E478CFF820 (not the same value!)
printf("value of q: %c\n", *q);     // e

char* r = 1;   // warning: right side is not a pointer!
char* r = 't'; // warning: right side is not a pointer!
char* r = "t"; // ok: right side is a pointer!

////////////////////////////////
////////////////////////////////

char* a = "abcde";
char* p = &a;    // warning: &a is of type char**, not char*!
char* p = a;     // ok
char* p = *&a;   // ok, same as previous line
char* p = &a[0]; // ok, same as previous line, a[0] is the char 'a'
printf("string at address p: %s\n", *p); // warning: *p is a char, not a char*!
printf("string at address p: %s\n", p);  // abcde
printf("char at address p: %c\n", *p);   // a
```

arrays

```c
//arrays with fixed length
int A[5]; // array of 5 int = a pointer to A[0]
printf("value of A: 0x%p\n", A); // 0x0000002D20EFF980
int* q = &A[0];
printf("value of q: 0x%p\n", q); // 0x0000002D20EFF980

int A[5] = {1,2,3,4,5};

double A[6] = {
    [0] = 9.0,
    [1] = 2.9,
    [3] = 5,      // we can skip [2] (will be 0.0)
    [5] = 3.E+25, // we can define [5] before [4]
    [4] = .00007
};

//arrays without fixed length
int A[];   // error: C does not support arrays without fixed length!
char* A[]; // same error!
           // only accepted as the main function second argument!

// this is what pointers are made for!
int* B; // this is actually an array of unspecified size :-)
```

global & static

Global: sont visibles et peuvent être modifiée à l'intérieure de nimporte quelle fonction (quelles soient déclarées hors ou dans une fonction, ce qui serait rare)

Static:
- hors d'une fonction => pas accessible en dehors du fichier
- dans une fonction => la variable nest pas recréée sur le stack à chaque appel de la fonction mais reste locale à la fonction

incrementation

```c
int a = 5;
int b = ++a; //a == 6; b == 6
int c = b++; //b == 7 but c == 6, the old value of b!!!
int d = --c; //c == 5; d == 5
int e = d--; //d == 4 but e == 5, the old value of d!!!
```

Functions

main

```c
// main function is not mandatory, but will be called if present
// it can take only 0 or two arguments
// in case it takes two arguments they must be int and char**

#include <stdlib.h>
#include <stdio.h>

int main(int argc, char* argv[]) {
    ...
    return EXIT_SUCCESS; //0, part of stdlib.h
}

/* program.exe arg1 arg2 arg3
-> argc = 4
-> argv = ["program.exe", "arg1", "arg2", "arg3"] */

int main(int argc, char** argv) { /* char* argv[]
                                     is EXACTLY the same as
                                     char** argv
                                     (this syntax is only allowed as a parameter of main) */
    ...
   }
```

printf

printf is part of stdio.h

%d: decimal
%f: float (can print a double)
%c: char
%s: string
%p: pointer
%zu: ?

Loops

for

```
////////////////////////////////////
////////////////////////////////////
//FOR LOOPS

// ++i is the same as i++ in a for loop

// GCC does not allow variable declarations in for loop initializers before C99
for (size_t i = 10; i; --i) { // prints numbers from 10 to 1
    ...                       // since "i" is true iff i != 0

}

// before c99 we must write:
size_t i;
for (i = 0; i < 5; ++i) { //n ote that i++ is equivalent to ++i here
    ...
}

////////////////////////////////////
////////////////////////////////////
//DECREMENTING FOR LOOPS

// Beware because if size_t i = 0 then i - 1 = SIZE_MAX
for (size_t k = 3; k >= 0; --k)  { // this is an infinite loop!!!
 ...
}

for (size_t k = 3; k >= 0; k--)  { // this is the exact same (infinite) loop!
 ...
}

// replace by:
for (size_t k = 3; (k >= 0) && (k <= 3); --k)  {
 ...
}
```

Exercises

```
////////////////////////////////////////
////////////////////////////////////////
//Printing a string by using only %c

void print(char* p) {
   if (*p == '\0') // Base case
      return;
   printf("%c", *p);
   print(++p);
}
```