Labo C
Circuits logiques et numériques [ELEC-H-305]

v1.2.0

Question 1. Hello world

Ce premier extrait de code présente le squelette de base d'un programme en C. Pour le compiler, vous pouvez utiliser Codeblocks ou un compilateur comme `gcc` en ligne de commande : `gcc -Wall hello.c`.

```c
#include <stdio.h> /* Allows the use of printf.*/
#include <stdlib.h>

int main()
{
    printf("Hello world!\n");

    /* If the program did not encounter any problem during
    its execution, it's supposed to return a signal saying so.
    On most systems, it's '0'. However, some OS are different
    and a more generic way to end a C program is to use the
    macro 'EXIT_SUCCESS' from stdlib.h. */
    return EXIT_SUCCESS;
}
```

Question 2. Type

Complétez le tableau suivant avec les valeurs maximales et minimales de chacun de ces types.

Type	Minimum	Maximum
int		
unsigned int		
short		
unsigned short		
long		
unsigned long		
float		
double		
char		
unsigned char		

Question 3. Overflow

Dans le code suivant, quelle sera la sortie ?

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

int main()
{
    int a = 2147483647;
    unsigned int b = 0;
    printf("a = %i, a+1 = %i\n", a, a+1);
    /* '%u' prints an unsigned int. */
    printf("b = %u, b-1 = %u\n", b, b-1);
    return EXIT_SUCCESS;
}
```

Question 4. C't'un bon type.

Choisissez le bon ordre des opérations afin de maximiser la précision du résultat final. Vous ne pouvez utiliser que des `short`.

```
short a = 50;
short b = 100;
short c = 600;
short d = 350;

short res1 = a * b / c;
short res2 = a / d * b;
short res3 = b * c / d;
```

Question 5. Casting

Une variable peut passer d'un type à un autre en utilisant le « casting ». Par exemple, pour transformer un `int` en `long`, on écrira

```
int a;
long b = (long)a;
```

Castez correctement le code suivant pour obtenir les bons résultats aux différentes opérations.

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

int main()
{
    char a = 50;
    char b = 70;
    char c = 100;

    // prod = a*b;
    // div = a/c;
    return EXIT_SUCCESS;
}
```

Question 6. Condition

Le code suivant présente la syntaxe de la structure conditionnelle `if`.

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

int main()
{
    int foo = 0;
    if (foo == 1) {
        /* Do something */
    }
    else if (foo == 2) {
        /* Do something else */
    }
    else {
        /* Do something? */
    }
    return EXIT_SUCCESS;
}
```

La structure conditionnelle ternaire permet de condenser certaines parties du code. Cette structure fonctionne de la façon suivante :

`(foo == 1) ? /* résultat si condition vraie */ : /* résultat si condition fausse */;`

Réécrivez le code du début de la question en utilisant des expressions ternaires.

Question 7. Loop

Il existe plusieurs façon d'exécuter une boucle en C : `while`, `for` et `do while`. Le code suivant présente leur syntaxe.

Quelle sera la sortie dans les différentes boucles ?

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

int main()
{
    int foo = 0;

    while (foo < 10) {
        printf("While loop iteration: %i\n", foo);
        foo++;
    }

    /* In C, the variable used in the 'for' loop control has to be
    declared outside of the loop. */
    for (foo = 0; foo < 15; foo++) {
        /* Do something. */
        printf("For loop iteration: %i\n", foo);
    }

    do {
        /* First execution of the loop. */
        foo++;
        printf("Do while loop iteration: %i\n", foo);
    } while (foo < 10);

    return EXIT_SUCCESS;
}
```

Question 8. Pointeurs Le code ci-dessous présente le fonctionnement des pointeurs. Exécutez le code pour observer les changements de valeur et d'adresse.

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

int main()
{
    /* Create a variable containing 42.*/
    int foo = 42;

    /* &foo gives the address of the variable foo.*/
    printf("foo = %i, address: %p\n", foo, &foo);

    /* Create a pointer 'bar' pointing to the address of 'foo'.*/
```

```c
    int * bar = &foo;

    /* Any subsequent invocation to '*bar' gives the value at the address
    pointed by 'bar'.*/
    printf("bar = %i, address: %p\n", *bar, bar);

    /* To create an "empty" pointer, we can allocate it some memory.*/
    int * baz = (int *)malloc(sizeof(int));
    /* 'baz' is now a pointer to a memory space of one 'int'.*/

    *baz = foo;
    /* *baz = 42 */

    *baz = *bar;
    /* *baz = *bar = 42 */

    baz = bar;
    /* baz and bar now point to the same address. */

    /* If we change the memory space pointed by 'bar': */
    *bar = 23;
    /* *baz will also change since it points to the same place.*/

    return EXIT_SUCCESS;
}
```

Question 9. Structures

Losque l'on veut déclarer un groupe de variables qui forment un ensemble cohérent, il est judicieux d'utiliser une structure adéquate. Dans le paradigme orienté objet, on créerait par exemple une classe dont on peut instancier des objets.

Le C n'étant pas orenté objet, il faut se tourner vers une autre solution : les « structures ».

```c
struct MyStruct {
    int a;
    int b;
};
```

On peut ensuite instancier cette structure :

```c
struct MyStruct foo;
```

On a maintenant une variable `foo` contenant deux variables `a` et `b` de type `int`. Afin d'éviter de devoir écrire `struct` à chaque fois que l'on veut créer cette structure, on peut aussi créer une variable portant le même nom que notre structure.

```c
typedef struct MyStruct Mystruct;
```

`typedef` fonctionne ainsi : je déclare une type (`typedef`) portant le nom « `MyStruct` » et étant en réalité un « `struct MyStruct` ». On peut à présent redéclarer `foo` :

```c
MyStruct foo;
```

L'accès au contenu de la structure dépend de la façon dont elle a été instanciée. Le code suivant présente les différentes façons de procéder.

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

typedef struct MyStruct MyStruct;

struct MyStruct{
    int a;
    int b;
};

int main()
{
    /* Direct instance */
    MyStruct foo;

    /* Content accessed using a '.' */
    foo.a = 1;
    foo.b = foo.a;

    /* Pointer instance */
    MyStruct * bar;

    /* Don't forget to allocate it some memory. */
    bar = malloc(sizeof(MyStruct));

    /* If using its address, content accessed using a '->' */
    bar->a = 2;

    /* If using its value, content accessed using a '.' */
    (*bar).b = 5;

    return EXIT_SUCCESS;
}
```

Créez à présent votre propre structure permettant de contenir la note finale de tous vos cours de l'année, en supposant que toutes les notes sont arrondies à l'unité. Veillez à choisir le type optimal pour minimiser l'espace occupé par votre structure. Quelle sera sa taille totale ?

Question 10. Arrays

Travaillez avec le code suivant pour vour familiariser avec le fonctionnement des tableaux.

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

int main()
{
    /* First create a static array of five char.*/
    char tab[5];

    /* We can then check the size of the table.*/
    printf("%lu\n",sizeof(tab));
```

```c
    /* What if we want to create an array dynamically, without knowing
    its size at compiltation?*/
    int a;
    scanf("%i",&a);

    /* This can work on some laxist compilers, but is not standard in all
    vesrions of C.*/
    char tabi[a];

    /* To create a dynamic array, we need to allocate it memory by hand.*/
    char * tabii = (char*)malloc(a*sizeof(char));

    /* However, when we check the table size:*/
    printf("%lu\n",sizeof(tabii));
    /* we do not get the actual size of the array, but only the size of the
    pointer. The variable 'a' needs to ne kept in order to remember the size
    of the allocated array.*/

    /* The size of the array can be changed during the execution using realloc.*/
    tabii = (char*)realloc(tabii, a*2*sizeof(char));

    /* Later, if you don't use the array anymore, you need to free the allocated
    memory.*/
    free(tabii);

    return EXIT_SUCCESS;
}
```

Comment créer un tableau à double entrée ? Comment lui allouer de la mémoire dynamiquement ?

Question 11. String

Une chaîne de caractères n'est rien d'autre qu'un tableau de `char` se terminant par le caractère `\0`. Elle peut être initialisée de plusieurs façons différentes :

```c
char foo[4] = {'f', 'o', 'o', '\0'};
char *bar = "bar";
char baz[] = "baz";
```

Pour afficher un caractère seul ou une chaîne de caractères entière, on utilise les arguments suivants :

```c
printf("%c, %s", char, string);
```

Question 12. ASCII

Un caractère peut être contenu dans une variable de type `char`. Cependant, cette variable n'a que le sens qu'on lui donne.

Quelle est la différence entre ces trois variables ?

```c
char a = 0x42;
char b = 66;
char c = 'B';
```

Il existe une correspondance entre ces trois valeurs qui est reprise dans la table `ASCII`. Vous pouvez y accéder facilement sous Linux en tapant `man ascii` dans un terminal.

Traduisez la chaîne de caractère suivante : `char foo[64] = {73, 32, 115, 117, 114, 101, 32, 104, 111, 112, 101, 32, 121, 111, 117, 32, 100, 105, 100, 32, 110, 111, 116, 32, 99, 111, 110, 118, 101, 114, 116, 32, 116, 104, 105, 115, 32, 115, 116, 114, 105, 110, 103, 32, 98, 121, 32, 104, 97, 110, 100, 46, 10, 85, 115, 101, 32, 112, 114, 105, 110, 116, 102, 46};`

Question 13. Opérations logiques

Une variable peut être déclarée de plusieurs façons différentes :

```c
char a = 8; // décimal
char b = 0x8; // Hexadécimal
char c = 'f'; // Caractère
char d = 08; // Octal
char e = 0b1000; // Binaire
```

En plus de ces déclarations, il existe des opérateurs permettant d'effectuer des opérations logiques « bit-à-bit » :

&	AND	$0 \,\&\, 1 = 0$
|	OR	$0 \mid 1 = 1$
^	XOR	$0b0101 \,\hat{}\, 0b1111 = 0b1010$
~	NOT	$\sim 0b1010 = 0b0101$
<<	Décalage à gauche	$0b001 << 2 = 0b100$
>>	Décalage à droite	$0b110 >> 2 = 0b001$

Complétez le code suivant pour afficher un nombre décimal sous sa forme binaire.

```c
#include <stdio.h>

int main(){
    int n;
    printf("Entrez un nombre entier: ");
    scanf("%i", &n);

    /* ... */

    return EXIT_SUCCESS;
}
```

Question 14. Errors

Corrigez le code suivant pour qu'il compile correctement.

```c
#include <stdio.h>
#inclued <stdlib.h>

int main()
{
    int a = 0

    for (int i = 0; i>=0; i++) {
```

```
        printf("That's a good loop. Not infinite at all.\n");
    }

    return EXIT_SUCCESS
}
```

En quoi l'exécution est-elle problématique ?

Question 15. SM

Un ouvre-porte Z_1 est commandé par deux boutons a et b qu'il faut actionner dans le bon ordre sous peine de déclencher une alarme Z_2. Celui-ci est décrit par la table de Huffman ci-dessous. Implémentez l'automate correspondant en C.

	ab				Z_1	Z_2
	00	01	11	10		
1	**1**	5	5	2	0	0
2	3	5	5	**2**	0	0
3	**3**	4	5	5	0	0
4	1	**4**	5	5	1	0
5	1	**5**	**5**	**5**	0	1

Question 16. RTFM

Toutes les fonctions présentes dans les bibliothèques standards ont un manuel. Sous Linux, ces informations sont directement accessibles en tapant « man 3 nom_de_la_fonction » dans un terminal, le « 3 » indiquant qu'on souhaite accéder à la section du manuel afférente au C.

Ces *man pages* sont aussi accessibles en ligne, comme par exemple pour la fonction `printf` : `https://linux.die.net/man/3/printf`.

Question 17. En utilisant le manuel de la bibliothèque `string` et celui de la fonction `printf`, concaténez deux chaînes de caractères `foo` et `bar` dans une troisième `baz` et imprimez-la dans la console.

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

int main()
{
    char* foo = "foo";
    char* bar = "bar";
    char* baz;
    return EXIT_SUCCESS;
}
```

Question 18. En incluant la bibliothèque `stdbool.h`, il est possible d'utiliser le type `bool` pouvant prendre les valeurs `true` ou `false`. Cependant, la fonction `printf` n'a pas de modificateur pour ce type, il n'est donc pas possible par défaut d'afficher « true » ou « false » directement dans la console comme on afficherait la valeur d'un `int` avec le modificateur `%i`.

Utilisez vos connaissances des expressions ternaires et de la fonction `printf` pour combler ce manque.

Question 19. Function

Comme dans la plupart des langages, il est possible de créer des fonctions en C. Leur structure générale est la suivante :

```
<return type> function (<type> arg1, <type> arg2) {
    return <return variable>
}
```

Ce qui est proche du Python :

```
def function(arg1, arg2) :
    return <return variable>
```

Ou encore des méthodes en Java :

```
<access modifier> <return type> function(<type> arg1, <type> arg2) {
    return <return variable>
}
```

Il existe trois niveaux différents à une fonction :

```
/* Prototype */
int func(int a, int b);

void main() {
    int a, int b;

    /* Call */
    int c = func(a, b);
}

/* Declaration */
int func(int a, int b) {
    return a+b;
}
```

Il faut veiller à plusieurs choses :

— N'appeler la fonction qu'après sa déclaration ou après son prototype. Si on appelle la fonction à la ligne 50 du code, il faut qu'elle ait été déclérée avant cette ligne ou que son prototype se trouve avant cette ligne.

— Le type de la fonction doit être le type de la variable de retour.

— Si la fonction en renvoie aucune variable, elle est de type `void`.

Question 20. Arguments

Il existe deux façons de passer des arguments à une fonction en C : par valeur ou par variable.

Dans le premier cas, on donne la valeur de la variable en argument de la fonction. Lorsque cette dernière est exécutée, elle commence par créer une nouvelle variable pour chaque arguments et les initialise à la valeur renseignée. Ainsi, les variables données en arguments ne sont pas modifiées, on utilise simplement leur valeur.

```c
void func(int a) {
    a++;
}

void main() {
    int a = 2;
    func(a); // 'a' est toujours égal à 2.
}
```

De le second cas, on donne l'adresse de la variable en argument et on travaille avec son pointeur dans la fonction. Cette fois-ci, la variable n'est plus copiée et on travaille directement sur l'original.

```c
void func(int * a) {
    *a++;
}

void main() {
    int a = 2;
    func(&a); // a = 3.
}
```

Question 21. Écrivez une fonction récursive calculant les n premiers termes de la suite de Fibonacci.

Question 22. Écrivez une fonction `void doubleStr(char* str)` recevant une chaîne de caractère et doublant son contenu. Par exemple :

```c
char * str="coucou";
printf("%s\n", str); // coucou
doubleStr(str);
printf("%s\n", str); // coucoucoucou
```

Question 23. Header

Lorque le nombre de fonctions augmente dans votre programme, il faut commencer à organiser son code. Pour ce faire, on ne garde pas tout dans un seul fichier `main.c`, mais on regroupe certaines fonctions dans des fichiers séparés. Deux types de fichiers sont utilisés en C : les `.h`, contenant les prototypes des fonctions, et les `.c` contenant leurs déclarations. Cette séparation fonctionnelle des deux types de fichiers est purement conventionnelle, il est tout à fait possible de déclarer une fonction dans un `.h`.

Par exemple, votre code de la question précédente pourrait ressembler à ceci :

```c
/* doublestr.h */
void doubleStr(char* str);

/* doublestr.c */
#include "doublestr.h"

void doubleStr(char* str) {
    /* Contenu de la fonction */
}

/* main.c */
#include <stdio.h>
#include "doublestr.h"
```

```
int main() {
    /* ... */
]
```

Si vous utilisez GCC en ligne de commande, la commande de compilation devient alors : `gcc -Wall main.c doublestr.c` Dans Codeblocks, il suffit d'ajouter les fichiers au projet et l'IDE s'occupe du reste dans les coulisses.

Vous aurez remarqué que `doublestr.h` est inclus dans l'en-tête du fichier `main.c`. C'est ainsi qu'il sait que la fonction `doubleStr` existe et où aller la chercher. De plus, ce header étant un fichier faisant partie du projet, on indique au compilateur d'aller chercher le `.h` dans le dossier courant en mettant son nom entre guillemets. Lorsque l'on souhaite utiliser une bibliothèque standard comme `stdio.h`, on met son nom entre chevrons et le compilateur ira chercher le fichier dans le dossier d'installation correspondant quelque part dans le système.

Reprenez vos fonctions `doubleStr` et `fibonacci`, placez-les dans des headers séparés et vérifier que votre programme fonctionne toujours.

Question 24. Implémentez la méthode de Quine-McCluskey.

Question 25. Écrivez une fonction permettant de détecter les problèmes de course dans une table de Huffman donnée en paramètre sous forme d'un tableau à double entrée.

Question 26. Écrivez une fonction permettant d'extraire les K-maps sous-jacentes d'une table de Huffman donnée en paramètre.