Systèmes d'exploitation — Exercices
Bases du langage C

Seweryn Dynerowicz (Seweryn.DYNEROWICZ@umons.ac.be)

1 Bases

Au cours de ces travaux pratiques, nous utiliserons le langage C pour lequel le compilateur sous Linux est GCC [1]. Le listing ci-dessous reprend les éléments principaux dans un exemple de programme introductif.

```
1  #include <stdio.h>  // printf(..)
2  #include <stdlib.h> // EXIT_SUCCESS
3
4  int total = 0;
5
6  int main(int argc, char* argv[]) {
7    int a = 42;
8    printf("Hello, world! %i\n", a);
9    return EXIT_SUCCESS;
10 }
```

FIGURE 1 – Programme `Hello, world!`

Les *directives d'inclusion* (*e.g.* `#include <stdio.h>` ; ligne 1) permettent d'inclure dans un fichier source les déclarations de fonctions d'une librairie particulière. Étant donné que notre programme utilise la fonction d'affichage `printf(..)` qui est implémentée dans la librairie `stdio`, nous devons inclure le fichier de *headers* correspondant pour pouvoir l'utiliser [2].

Les *constantes symboliques* (*e.g.* `EXIT_SUCCESS` ; ligne 9) permettent d'améliorer la lisibilité du code en dissimulant des valeurs utilisées derrière des noms [3].

La fonction principale `main(..)` (ligne 6) représente le point d'entrée de l'exécution du programme produit par le compilateur [4]. Cette fonction reçoit toujours deux paramètres ; le nombre d'argument(s) reçu(s) à l'invocation du programme à la ligne de commande (*i.e.* `argc` ; ligne 6) ainsi qu'un tableau de chaînes de caractères encodant lesdits arguments (*i.e.* `argv` ; ligne 6). Notez que le premier argument contenu dans le tableau est toujours le nom du fichier contenant le programme. Cette fonction renvoye une valeur de retour qui indique si l'exécution s'est déroulée correctement ou si une erreur a été rencontrée [5].

Il est possible de déclarer des variables à deux niveaux différents ; *locales* à une fonction (*e.g.* `a`, ligne 7) qui ne sont accessibles que pendant son exécution et dont la valeur est perdue après le retour de cette fonction, ou *globales* au programme (*e.g.* `total`, ligne 4) qui sont accessibles pendant l'exécution de n'importe quelle fonction et dont la valeur est préservée entre les appels.

L'affichage textuel se fait par le biais de la fonction `printf(..)` qui prend en argument une chaîne de caractères. Dans l'éventualité où l'on souhaite afficher les valeurs de certaines variables, des *tokens* de formatage (*e.g.* `%d`, `%f`) peuvent être introduits dans la chaîne de caractères et ces derniers seront remplacés par les valeurs des variables passées en paramètres additionnels (dans l'ordre donné).

Afin de compiler un programme, le compilateur peut être invoqué à la ligne de commande pour compiler un programme comme ci-dessous. Le fichier source à utiliser est spécifié et le compilateur produit un fichier exécutable (format `ELF`) qui peut être ensuite invoqué [6]. L'option `-o` permet de spécifier le nom de l'exécutable (`a.out` par défaut)

1. *GNU Compiler Collection* : `https://gcc.gnu.org/`
2. Chemins absolus respectifs : `/usr/include/stdio.h` (*header*) et `/usr/lib/libc.a` (librairie)
3. `/usr/include/stdlib.h` contient la définition `#define EXIT_SUCCESS 0`
4. Techniquement, le point d'entrée est la fonction `_start` qui appelle la fonction `main`
5. Cette valeur de retour peut être consultée après l'exécution d'un programme en utilisant la commande `echo $?`
6. Pour autant que la compilation se soit passée sans problème

```
sdy@mentat $ gcc -o program sourcecode.c
sdy@mentat $ ./program
Hello, world! 42
```

FIGURE 2 – Compilation et exécution d'un programme

2 Variables & types

Les différents types élémentaires disponibles en C sont repris dans la table ci-dessous. Les tailles exactes pour une machine donnée peuvent être trouvées dans le fichier de *headers* nommé `limits.h`. Les tables ci-dessous reprennent les tailles pour un système GNU/Linux typique. Il est possible d'utiliser l'opérateur `sizeof` sur un type ou une variable pour obtenir sa taille en octets considérée par le compilateur.

Type	Mot-clef	*Token*	Octets	Minimum	Maximum
Caractère	`char`	`%c`	1	-128	127
Entier	`int`	`%i`	4	$-2.147.483.648$	$2.147.483.647$
Virgule flottante simple	`float`	`%f`	4	$\approx -3.40282 \times 10^{38}$	$\approx 3.40282 \times 10^{38}$
Virgule flottante double	`double`	`%lf`	8	$\approx -1.79769 \times 10^{308}$	$\approx 1.79769 \times 10^{308}$

FIGURE 3 – Table des types basiques

Il est possible de qualifier plus spécifiquement la nature du type d'entier utilisé pour une variable. La table suivante reprend les variantes possibles ainsi que les tailles typiques et valeurs minimale/maximale correspondantes.

Type	Mot-clef	*Token*	Octets	Minimum	Maximum
Non-signé	`unsigned int`	`%u`	4	0	4.294.967.295
Court	`short`	`%hi`	2	-32.768	32.767
Court non-signé	`unsigned short`	`%hu`	2	0	65.535
Long	`long`	`%li`	8	$-9.223.372.036.854.775.808$	$9.223.372.036.854.775.807$
Long non-signé	`unsigned long`	`%lu`	8	0	$18.446.744.073.709.551.615$

FIGURE 4 – Table des variantes d'entiers

n.b. : il n'existe pas en C de type *booléen*. Les tests de valeurs de vérité reposent directement sur la représentation binaire d'un mot qui repose sur la correspondance suivante ; le mot binaire dont **tous** les *bits* sont positionnés à 0 représente *false*, tandis que tous les mots binaires contenant **au moins** un *bit* positionné à 1 représentent *true*.

2.1 Structures

La notion de structure en C permet de regrouper plusieurs variables qui sont logiquement liées dans un programme afin de rendre plus évident le raisonnement sous-jacent. Par exemple, pour manipuler des vecteurs, au lieu de passer séparément les coordonnées x et y, il est possible de définir une structure qui regroupe ces deux variables.

```
struct vector {
  double x;
  double y;
};

double norm(struct vector v) {
  return sqrt(v.x*v.x + v.y*v.y);
}
```

```
int main(int argc, char* argv[]) {
  struct vector v1;
  v1.x = 1.0L;
  v1.y = 1.0L;
  printf("Norm =%lf\n", norm(v1));

  return EXIT_SUCCESS;
}
```

FIGURE 5 – Déclaration et utilisation de structure

2.2 Opérateurs & instructions

Les opérateurs *arithmétiques* classiques (*i.e.* `+`, `-`, `*`, `/`) sont disponibles et sont automatiquement adaptés selon le type des opérandes utilisées. Les *opérateurs de comparaison* (*i.e.* `<=`, `>=`, `<`, `>`, `==`, `!=`) permettent de tester la relation d'ordre entre deux valeurs/variables. Les opérateurs *logiques* classiques (*i.e.* `&&`, `||`, `!`) peuvent être utilisés pour combiner les résultats de comparaisons afin d'exprimer des conditions plus complexes.

Les opérateurs d'*assignation augmentée* (*i.e.* `+=`, `-=`, `*=`, `/=`) viennent compléter l'assignation classique (*i.e.* `=`) pour permettre d'exprimer rapidement la mise à jour d'une variable sur base d'une autre valeur. Par exemple, `x = x+1;` peut

être écrit `x += 1;`. Les opérateurs d'*incrémentation* et de *décrémentation* (*i.e.* `++`, `--`) permettent d'exprimer encore plus rapidement ces opérations. Par exemple, `x = x+1;` peut être écrit `x++;`.

A titre informatif[7], il existe également des opérateurs *bitwise* (*i.e.* `&`, `|`, `^`, `~`, `>>`, `<<`) qui permettent de manipuler les opérandes au niveau des *bits* individuels. Par exemple, `1 | 2` va combiner les *bits* de poids identiques par un OU binaire pour produire `3`.

Les instructions de contrôle habituelles (*i.e.* alternative et boucle) sont reprises dans les listings ci-dessous.

```
if (a == 0) {
  // Branche then
} else {
  // Branche else (facultative)
}
```

```
unsigned int counter = 0;
while(counter < maximum) {
  // Do something using counter
  counter++;
}
```

```
for(unsigned int counter = 0; counter < maximum; counter++) {
  // Do something using counter
}
```

FIGURE 6 – Structures de contrôle. La branche `else` peut être omise si son corps est vide. Une boucle `while` traversant un ensemble de valeurs de `counter` peut être remplacée par une boucle `for`.

2.3 Tableaux & chaînes de caractères

La définition de tableaux en C requiert un type et une taille. Pour un tableau contenant n éléments, les indices valides[8] vont de `0` à `n-1`.

```
1  #include <stdio.h>  // printf(...)
2  #include <stdlib.h> // EXIT_SUCCESS
3
4  unsigned int array[24] = {
5    10, 23, 27, 31, 32, 37, 41, 47,
6    51, 55, 57, 58, 60, 66, 68, 69,
7    70, 73, 74, 78, 79, 83, 93, 94
8  };
9
10 unsigned int sum(unsigned int numbers[], unsigned int length) {
11   unsigned int total = 0;
12   for(unsigned int index = 0 ; index < length; index++) {
13     total += numbers[index];
14   }
15   return total;
16 }
17
18 int main(int argc, char* argv[]) {
19   printf("Total : %u\n" , sum(array) );
20   return EXIT_SUCCESS;
21 }
```

FIGURE 7 – Manipulation de tableau

Une chaîne de caractères est simplement stockée dans un tableau de caractères. Les caractères sont encodés suivant la norme ASCII[9] qui établit une correspondance entre les valeurs décimales qu'un octet peut prendre et les symboles correspondant. Par exemple, la valeur 65 correspond au caractère A (majuscule) tandis que la valeur 111 correspond au caractère o (minuscule).

La fin d'une chaîne de caractères est dénotée par un caractère nul[10] (*i.e.* `'\0'`). Ceci à pour conséquence que tout code qui modifie la longueur d'une chaîne de caractère doit correctement placer un caractère nul au bon endroit pour refléter la nouvelle longueur.

7. *cfr.* `https://graphics.stanford.edu/~seander/bithacks.html`
8. **Attention :** un accès dans un tableau en utilisant un indice hors de ces bornes donne lieu à un comportement indéfini.
9. *American Standard Code for Information Interchange.* Les valeurs décimales de 32 à 126 correspondent à des caractères *affichables* tandis que les valeurs décimales de 0 à 31 ainsi que 127 représentent des caractères de *contrôle.*
10. C'est de cette représentation que vient le terme *null-terminated string*

3 Énoncés

Exercice 1

Écrivez une fonction qui renvoye le nombre de caractères (excepté le `'\0'`) dans une chaîne de caractères.

```
unsigned int string_length(char string[]);
```

Exercice 2

Écrivez une fonction qui affiche en majuscules une chaîne de caractères passée en paramètre.

```
void capitalize(char string[]);
```

Exercice 3

Écrivez une fonction qui **remplace** toutes les occurrences d'un certain caractère dans une chaîne de caractères par un autre caractère.

```
void replace(char string[], char target, char replacement);
```

Exercice 4

Écrivez une fonction qui **supprime** toutes les occurrences d'un certain caractère dans une chaîne de caractères.

```
void delete(char string[], char target);
```

Exercice 5

Écrivez une fonction, en utilisant l'opérateur `sizeof`, qui affiche la taille (en octets) ainsi que le nombre théorique de valeurs possibles pour une variable pour chacun des types suivants : `int`, `unsigned int`, `short`, `long`, `float`, `double`

```
void print_type_stats();
```

Astuce : pour calculer la puissance d'un nombre, utilisez la fonction `pow(x,y)` déclarée dans `math.h`. Au moment de compiler, vous devez spécifier l'utilisation de la librairie `math` (*i.e.* `gcc -lm codesource.c`)

Exercice 6

Écrivez une fonction qui renvoye le nombre d'occurrences d'un certain caractère dans une chaîne de caractères.

```
unsigned int count_char_occurrences(char string[], char target);
```

Exercice 7

Écrivez une fonction qui renvoye le nombre d'occurrences d'un certain mot dans une chaîne de caractères.

```
unsigned int count_word_occurrences(char string[], char target);
```

Exercice 8

Écrivez une fonction qui calcule le nombre d'occurrences de chaque lettre de l'alphabet dans une chaîne de caractères passée en paramètre et affiche ces nombres.
Modifiez la fonction pour que l'affichage se fasse dans l'ordre décroissant et que les comptes nuls ne soient pas affichés.

```
void print_histogram(char string[]);
```

Astuce : pour effectuer le triage, vous pouvez utiliser un tableau de structures regroupant chaque caractère avec sa fréquence et adapter l'algorithme de tri *bubble-sort* pour fonctionner sur la fréquence de chaque élément.