TELECOM Paris

IP PARIS

Partie 2.3 - Le langage C

ECE_3TC31_TP/INF107

Guillaume Duc

2025

Institut Mines-Télécom

Erreurs et bibliothèque standard

Gestion des erreurs : `errno`

La plupart des fonctions de la bibliothèque standard indique la survenue d'une erreur en retournant une valeur spéciale (voir la documentation). Par exemple :

- `printf` retourne une valeur négative
- `malloc` retourne `NULL`
- `fscanf` retourne `EOF`
- `fork` retourne `-1`

Néanmoins, cette valeur n'indique pas pourquoi il y a eu un problème (pénurie de mémoire, fichier fermé, disque plein…).

Pour cela, en cas d'erreur, les fonctions de la bibliothèque standard utilisent la variable globale `errno` (déclarée dans `errno.h`) de type `int` en y mettant une valeur indiquant la cause de l'erreur.

https://en.cppreference.com/w/c/error/errno

Gestion des erreurs : `perror`

Sauf à connaître par cœur la correspondance entre la valeur de `errno` et la cause correspondante, ce n'est pas plus clair.

Heureusement, la fonction `perror` peut nous aider à produire un message d'erreur compréhensible par l'utilisateur.

```
void perror(const char *s);
```

- `perror` est déclarée dans `stdio.h`
- `perror` affiche sur la sortie d'erreur standard
  - tout d'abord la chaîne `s`
  - puis `": "`
  - et enfin un message correspondant à la valeur courante de `errno`

https://en.cppreference.com/w/c/io/perror

Gestion des erreurs : `exit`

Dans certains cas, continuer l'exécution suite à une erreur n'a pas de sens.

```
void exit(int status);
```

- `exit` est déclarée dans `stdlib.h`
- `exit` arrête l'exécution du programme
- L'argument `status` est le code de retour du programme (il a la même signification que la valeur de retour de `main`)
  - Il est donc possible d'utiliser `EXIT_FAILURE`

Un exemple d'utilisation sera donné dans quelques slides.

https://en.cppreference.com/w/c/program/exit

Allocation mémoire dynamique : le tas

Le tas

Le tas est une zone de la mémoire gérée directement par le programmeur/la programmeuse.

Le compilateur ne place pas de code ni de données dans le tas par lui-même.

Le programmeur doit :

- **Allouer** la mémoire (demander à réserver une plage d'adresse dans le tas)
- **Libérer** la mémoire (indiquer qu'une plage précédemment allouée n'est plus utilisée)

Pour effectuer ces deux opérations, le programmeur utilise des fonctions fournies par la bibliothèque standard (qui elles-mêmes utilisent le système d'exploitation).

Allocation mémoire dans le tas : `malloc`

```
void *malloc(size_t size);
```

- Déclarée dans le fichier d'en-tête `stdlib.h`
- Alloue `size` octets dans le tas
- Retourne la première adresse de la zone allouée
  - Ou la valeur `NULL` en cas d'échec (pénurie de mémoire)
  - Le type de cette valeur est donc un pointeur (la valeur est une adresse)
  - Par contre, cette fonction ne sait pas ce que vous voulez faire de cette zone, donc c'est un pointeur non typé : `void *`
- Le contenu de la zone allouée n'est pas initialisé

https://en.cppreference.com/w/c/memory/malloc

Allocation mémoire dans le tas : `malloc` (exemple)

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

int main(int argc, char *argv[]) {
  int *heapData = malloc(sizeof(int)*4); // Alloue une zone pouvant contenir 4 int
  if (!heapData) {                       // Équivalent à (heapData != NULL)
    perror("malloc failed");             // Affiche un message d'erreur
    return EXIT_FAILURE;                 // Quitte le programme
  }
  heapData[0] = 42;                      // Utilisation de la mémoire allouée
  return EXIT_SUCCESS;
}
```

Remarquez le traitement en cas d'erreur de `malloc` avec `perror` et `exit`.

Libération de la mémoire : `free`

```
void free(void *ptr);
```

- Déclarée dans le fichier d'en-tête `stdlib.h`
- Son argument doit être un pointeur précédemment renvoyé par `malloc` (ou une fonction similaire) et qui n'a pas déjà été libéré (sinon le comportement est non défini)
- Libère la mémoire correspondante

⚠ Une fois la mémoire libérée, il n'est plus question d'essayer d'y accéder.

https://en.cppreference.com/w/c/memory/free

Libération de la mémoire : `free` (exemple)

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

int main(int argc, char *argv[]) {
  int *heapData = malloc(sizeof(int)*4); // Alloue une zone pouvant contenir 4 int
  if (!heapData) {                       // Équivalent à (heapData != NULL)
    perror("malloc failed");             // Affiche un message d'erreur
    return EXIT_FAILURE;                 // Quitte le programme
  }
  heapData[0] = 42;                      // Utilisation de la mémoire allouée
  free(heapData);                        // Libère explicitement la mémoire allouée
  return EXIT_SUCCESS;
}
```

Toute la mémoire allouée est automatiquement libérée à la fin de l'exécution si cela n'a pas été fait plus tôt.

⚠ Néanmoins, c'est une bonne pratique de libérer la mémoire par un appel explicite à `free` dès qu'une zone n'est plus utilisée. C'est particulièrement important pour les programmes destinés à s'exécuter pendant une longue période (risque de *fuites mémoires*).

Réallocation : realloc

```
void *realloc(void *ptr, size_t size);
```

- Déclarée dans le fichier d'en-tête `stdlib.h`
- `ptr` doit être un pointeur précédemment renvoyé par `malloc` (ou une fonction similaire) et qui n'a pas déjà été libéré (sinon le comportement est non défini)
- `realloc` change la taille de la zone allouée, `size` est la nouvelle taille
  - Cette nouvelle taille peut être plus petite ou plus grande
- Les données de la zone sont conservées (éventuellement tronquées si la nouvelle taille est plus petite que l'ancienne)
- Retourne un pointeur vers la première adresse de la nouvelle zone (cela peut être la même adresse que précédemment ou non) ou `NULL` en cas d'erreur

https://en.cppreference.com/w/c/memory/realloc

Réallocation : `realloc` (exemple)

```c
int main(int argc, char *argv[]) {
  int *heapData = malloc(sizeof(int)*4);
  if (!heapData) {
    perror("malloc failed");
    return EXIT_FAILURE;
  }

  heapData[0] = 42;

  if (heapData = realloc(heapData, sizeof(int))) {  // Libère une partie de la zone
    perror("realloc failed");
    return EXIT_FAILURE;
  }

  // headData[0] vaut toujours 42
  return EXIT_SUCCESS;
}
```

Manipulation des fichiers

Manipulation des fichiers

Les différentes opérations (ouverture, fermeture, lecture, écriture…) sur les fichiers sont réalisées par le système d'exploitation.

La bibliothèque standard fournit un ensemble de fonctions qui utilisent les services fournis par le système d'exploitation (via des *appels systèmes*).

Plus précisément, deux API (*Application Programming Interface*) sont fournies par la bibliothèque standard :

- Une API « bas-niveau » (`open`, `close`, `read`, `write`…) offrant un accès direct aux appels systèmes
- Une API « haut-niveau » (`fopen`, `fclose`, `fprintf`, `fscanf`…) offrant des fonctionnalités plus riches

Dans la suite de ce cours, nous utiliseront l'API « haut-niveau ». L'API « bas-niveau » sera vue lors de la dernière partie de 3TC31 consacrée aux systèmes d'exploitation.

Ouverture d'un fichier : fopen (1)

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

- Déclarée dans le fichier d'en-tête `stdio.h`
- Ouvre un fichier
  - `filename` : chaîne de caractères indiquant le nom du fichier
  - `mode` : chaîne de caractères indiquant comment le fichier sera utilisé (lecture seule…, cf. slide suivant)
- Retourne un pointeur vers un type `FILE`
  - Nommé **file stream**
  - Le type `FILE` n'est pas spécifié
  - Ce pointeur identifie le fichier ouvert pour les opérations ultérieures sur ce fichier
  - En cas d'erreur (fichier inexistant, problème de permission, etc.), la valeur `NULL` est renvoyée (`errno` est positionnée)

https://en.cppreference.com/w/c/io/fopen
https://en.cppreference.com/w/c/io/FILE

Ouverture d'un fichier : `fopen` (1)

Valeurs possibles pour `mode`

`mode`	Description	Fichier existant	Fichier inexistant
`"r"`	Lecture seule	Lecture depuis le début	Erreur
`"w"`	Écriture seule	Contenu détruit	Le fichier est créé
`"a"`	Écriture seule	Ajout à la fin	Le fichier est créé
`"r+"`	Lecture/Écriture	Lecture/Écriture depuis le début	Erreur
`"w+"`	Lecture/Écriture	Contenu détruit	Le fichier est créé
`"a+"`	Lecture/Écriture	Lecture/Écriture depuis le début	Le fichier est créé

https://en.cppreference.com/w/c/io/fopen

Fermeture d'un fichier : `fclose`

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

- Déclarée dans le fichier d'en-tête `stdio.h`
- Ferme le fichier identifié par `stream`
  - `stream` ne doit plus être utilisé par la suite (comportement non défini)
- Retourne 0 en cas de succès
  - ou `EOF` (une constante) en cas d'erreur (`errno` est alors positionnée pour indiquer la cause)

https://en.cppreference.com/w/c/io/fclose

Exemple (fopen et fclose)

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

int main(int argc, char *argv[]) {
  FILE *f = fopen("output.txt", "w"); // Ouvre output.txt en écriture seule
  if (!f) {                           // Équivalent à f != NULL
    perror("fopen failed");
    return EXIT_FAILURE;
  }

  // ... Utilisation du fichier ...

  if (fclose(f)) {                    // Ferme le fichier, équivalent à fclose(f) != 0
    perror("fclose failed");          // En cas d'erreur
    return EXIT_FAILURE;
  }

  return EXIT_SUCCESS;
}
```

File Streams spéciaux

La bibliothèque standard défini (dans le fichier d'en-tête `stdio.h`) trois *file streams* (des pointeurs `FILE *`) spéciaux ouverts automatiquement au début du programme :

- `stdout` : Un flux qui représente la **sortie standard** (*standard output*)
- `stderr` : Un flux qui représente la sortie d'**erreur standard** (*standard error*)
- `stdin` : Un flux qui représente l'**entrée standard** (*standard input*)

https://en.cppreference.com/w/c/io/std_streams

Sorties formattées

```c
int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int sprintf(char *buffer, const char *format, ...);
```

- Ces trois fonctions ont un fonctionnement similaire
  - `printf` : affiche la chaîne formattée sur la sortie standard (équivalent à `fprintf` vers le flux `stdout`)
  - `fprintf` : la chaîne formattée est écrite dans le fichier représenté par le flux `stream`
  - `sprintf` : la chaîne formattée est écrite à l'adresse indiquée par `buffer` (⚠ l'adresse doit correspondre à une zone allouée de taille suffisante pour stocker le résultat)
- Ces fonction retourne
  - Le nombre de caractères écrits (sur la sortie standard, dans le fichier ou en mémoire)
  - Une valeur négative en cas d'erreur

https://en.cppreference.com/w/c/io/fprintf

Compléments sur les indications de formattage (1)

`'%' <flags>? <width>? ('.' <precision>)? <size modifier>? <format>`

- `width` : entier spécifiant la taille minimale du champ (si moins de caractères doivent être affichés, la valeur sera complétée par des espaces à gauche par défaut)
- `precision` : entier, précédé par un `.` spécifiant :
  - Le nombre minimum de chiffres pour les entiers
  - Le nombre de chiffres après la virgule pour les flottants
  - Le nombre maximum de caractères à afficher pour les chaînes de caractères

Compléments sur les indications de formattage (2)

- `flags` : un ou plusieurs caractères parmi

Flag	Description
`'-'`	Aligne le texte à gauche (défaut : droite), cf. `width`
`'+'`	Affiche toujours le signe d'un nombre, même s'il est positif
`' '`	Affiche une espace devant les nombres positifs (à la place du signe)
`'0'`	Complète avec des `0` au lieu d'espaces pour les nombres
`'#'`	Utilise l'affichage alternatif

Compléments sur les indications de formattage (3)

- `modifier` : Indique le type de l'entier ou du flottant passé en argument

Modifier	Description
(rien)	Type `int`
`"hh"`	Type `signed char`/`unsigned char`
`"h"`	Type `short`/`unsigned short`
`"l"`	Type `long`/`unsigned long`
`"ll"`	Type `long long`/`unsigned long long`
`"z"`	Type `size_t`/`ssize_t`
`"L"`	Type `long double`

Compléments sur les indications de formattage (4)

- `format` :

Format	Description
`'c'`	Affiche un caractère
`'d'`	Affiche un entier signé en base 10
`'u'`	Affiche un entier non signé en base 10
`'x'`	Affiche un entier en base 16
`'f'`	Affiche un flottant en décimal
`'e'`	Affiche un flottant en notation exponentielle
`'s'`	Affiche une chaîne de caractères
`'p'`	Affiche la valeur d'un pointeur (adresse en base 16)

Exemple : Écriture dans un fichier (1)

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

int main(int argc, char *argv[]) {
  FILE *f = fopen("output.txt", "w");     // Ouvre le fichier en écriture
  if (!f) {
    perror("fopen failed");
    return EXIT_FAILURE;
  }

  fprintf(f, "Hello World %d\n", 57);   // Écrit dans le fichier

  if (fclose(f)) {                      // Ferme le fichier
    perror("fclose failed");
    return EXIT_FAILURE;
  }
  return EXIT_SUCCESS;
}
```

`file-output.c`.

Exemple : Écriture dans un fichier (2)

```
ouessant:~/tmp$ ls
file-output.c

ouessant:~/tmp$ gcc -Wall -pedantic -std=c11 -g -o file-output file-output.c

ouessant:~/tmp$ ls
file-output  file-output.c

ouessant:~/tmp$ ./file-output

ouessant:~/tmp$ ls
file-output  file-output.c  output.txt

ouessant:~/tmp$ cat output.txt
Hello World 57
```

Exemple : Instructions de formattage (1)

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

int main(int argc, char *argv[])
{
  short s = 540;
  int i = 0xfbfb;
  float f = i * 1.133e5;
  static const char string[] = "|Left Aligned|";

  printf("size_t:%zd\n", sizeof(int));
  printf("Integer numbers (decimal): % hd and %+d\n", s, i);     // Printing of sign.
  printf("Integer numbers (hex): 0x%8x and 0x%08X\n", s, i);     // Padding with ' ' or '0'.
  printf("Floating-point numbers: %020.3f and %12.3e\n", f, f);  // Width before/after '.'.
  printf("String: %-20s is aligned\n", string);                  // Left aligned.

  return EXIT_SUCCESS;
}
```

print2.c

Exemple : Instructions de formattage (2)

```
ouessant:~/tmp$ ls
print2.c

ouessant:~/tmp$ gcc -Wall -pedantic -std=c11 -g -o print2 print2.c

ouessant:~/tmp$ ls
print2 print2.c

ouessant:~/tmp$ ./print2
size_t:4
Integer numbers (decimal):  540 and +64507
Integer numbers (hex): 0x     21c and 0x0000FBFB
Floating-point numbers: 0000007308643328.000 and    7.309e+09
String: |Left Aligned|       is aligned
```

Entrées formattées

```c
int scanf(const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
int sscanf(const char *buffer, const char *format, ...);
```

- Ces fonctions lisent depuis une entrée et l'analysent
  - `scanf` lit depuis l'entrée standard (identique à `fscanf` avec `stdin`)
  - `fscanf` lit depuis un fichier (`stream`)
  - `sscanf` lit depuis une chaîne de caractères (`buffer`)
- `format` est une chaîne de caractères contrôlant le comportement de ces fonctions : elle indique ce qu'elles doivent chercher à lire
- Valeur de retour
  - Nombre de valeurs lues avec succès
  - `EOF` en cas d'erreur

https://en.cppreference.com/w/c/io/fscanf

Chaîne de format (`format`) pour `scanf` (1)

La chaîne `format` se comporte de manière semblable à celle utilisée pour `printf`:

- Caractère normal (sauf espaces et `'%'`) : ce caractère doit être trouvé immédiatement en entrée
  - Si c'est le cas, il est consommé, sinon une erreur est levée
- Caractère d'espacement (`' '`, `'\t'`, `'\n'`…) : tous les caractères d'espacement successifs en entrée sont consommés
- Le caractère `%` a une signification particulière
  - Il indique qu'une valeur doit être lue depuis l'entrée
  - Ce qui suit `%` indique ce qui doit être lu (un entier, un caractère…)
  - La valeur lue est stockée dans les arguments successifs (ceux passés après la chaîne `format` lors de l'appel à `scanf`). La fonction modifiant ses arguments, ces derniers doivent être passés par adresse (être des pointeurs)

https://en.cppreference.com/w/c/io/fscanf

Chaîne de format (`format`) pour `scanf` (2)

```
'%' '*'? <width>? <length modifier>? <format>
```

- Une `*` au début indique de lire la valeur, sans l'affecter à un argument
- `width` : un entier optionnel indiquant le nombre maximum de caractères à lire (utile en particulier pour la lecture des chaînes de caractères)
- `length modifier` : indique le type de l'argument, comme pour `printf` (`hh`, `h`, `l`, `ll`, `z`, et `L`)
- `format` : indique ce qui doit être lu en entrée
  - `d`, `u`, `x` : un entier (signé en base 10, non signé en base 10, non signé en base 16)
  - `f` et `e` : un flottant
  - `c` : un ou plusieurs caractères (cf. `width`)
  - `s` : une série de caractères, au plus `width` ou jusqu'au premier caractère d'espacement (la première condition rencontrée), et ajoute un caractère de fin de chaîne (`'\0'`). ⚠ le pointeur de destination doit donc pouvoir stocker jusqu'à `width + 1` caractères

Exemple (1)

```c
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
  FILE *f = fopen(argv[1], "r");                // Ouvre le fichier en lecture
  if (!f) {
    perror("fopen failed"); return EXIT_FAILURE;
  }
  char c, s[50];
  int i;
  int read = fscanf(f, "%s%d%c\n", s, &i, &c);  // Lit une chaîne, un entier et un caractère
  if (read != 3) {
    perror("fscanf failed"); return EXIT_FAILURE;
  }
  printf("%s\n%d\n0x%x\n", s, i, c);
  if (fclose(f)) {                              // Ferme le fichier
    perror("fclose failed"); return EXIT_FAILURE;
  }
  return EXIT_SUCCESS;
}
```

`file-input.c`.

Exemple (2)

```
ouessant:~/tmp$ ls
file-input.c  input.txt

ouessant:~/tmp$ cat input.txt
String_without_whitespace_plus_null_character
3
a

ouessant:~/tmp$ gcc -Wall -pedantic -std=c11 -g -o file-input file-input.c

ouessant:~/tmp$ ls
file-input file-input.c input.txt

ouessant:~/tmp$ ./file-input no-file.txt
fopen failed: No such file or directory

ouessant:~/tmp$ ./file-input input.txt
String_without_whitespace_plus_null_character
3
0xa
```

La sortie vous paraît-elle normale ?

Trouvons le problème

Regardons le contenu du fichier `input.txt` avec `hexdump` (à gauche les adresses, au milieu les différents caractères représentés en base 16 et à droite les caractères ASCII correspondant)

```
ouessant:~/tmp$ hexdump -C input.txt
00000000  53 74 72 69 6e 67 5f 77  69 74 68 6f 75 74 5f 77  |String_without_w|
00000010  68 69 74 65 73 70 61 63  65 5f 70 6c 75 73 5f 6e  |hitespace_plus_n|
00000020  75 6c 6c 5f 63 68 61 72  61 63 74 65 72 0a 33 0a  |ull_character.3.|
00000030  61 0a                                             |a.|
00000032
```

Voyez-vous le problème et comment le corriger (en changeant la chaîne de formattage passée à `fscanf`) ?

Réponse

- Après l'entier `3` lu par `'%d'` se trouve un caractère de saut de ligne (*line feed* de valeur `0x0a`)
- C'est donc ce caractère qui est lu par le `%c` et affecté à la variable `c`

```
ouessant:~/tmp$ hexdump -C input.txt
00000000  53 74 72 69 6e 67 5f 77  69 74 68 6f 75 74 5f 77  |String_without_w|
00000010  68 69 74 65 73 70 61 63  65 5f 70 6c 75 73 5f 6e  |hitespace_plus_n|
00000020  75 6c 6c 5f 63 68 61 72  61 63 74 65 72 0a 33 0a  |ull_character.3.|
00000030  61 0a                                             |a.|
```

- Pour corriger, il suffit d'ajouter un caractère d'espacement entre `%d` et `%c` (par exemple une espace) dans la chaîne de formattage pour `fscanf`

Exemple corrigé (1)

```c
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
  FILE *f = fopen(argv[1], "r");                // Ouvre le fichier en lecture
  if (!f) {
    perror("fopen failed"); return EXIT_FAILURE;
  }
  char c, s[50];
  int i;
  int read = fscanf(f, "%s%d %c\n", s, &i, &c);  // Lit une chaîne, un entier et un caractère
  if (read != 3) {
    perror("fscanf failed"); return EXIT_FAILURE;
  }
  printf("%s\n%d\n0x%x\n", s, i, c);
  if (fclose(f)) {                              // Ferme le fichier
    perror("fclose failed"); return EXIT_FAILURE;
  }
  return EXIT_SUCCESS;
}
```

file-input2.c.

Exemple corrigé (2)

```
ouessant:~/tmp$ ls
file-input2.c  input.txt

ouessant:~/tmp$ cat input.txt
String_without_whitespace_plus_null_character
3
a

ouessant:~/tmp$ gcc -Wall -pedantic -std=c11 -g -o file-input2 file-input2.c

ouessant:~/tmp$ ls
file-input2 file-input2.c input.txt

ouessant:~/tmp$ ./file-input2 input.txt
String_without_whitespace_plus_null_character
3
0x61
```

Chaînes de caractères

Comme vu précédemment, une chaîne de caractères en C est une suite de caractères en mémoire terminée par un caractère spécial (de valeur `0`, souvent écrit `'\0'`).

Elles sont manipulées et transmises sous la forme d'un pointeur vers le premier caractère (`char *`) ou éventuellement d'un tableau.

Les opérateurs de comparaison (==) et d'affectation (=) ne fonctionnent donc pas comme on le souhaiterait sur les chaînes (ils opèrent sur les pointeurs et non les chaînes elles-mêmes).

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
  char sa[] = "aaa";
  char sb[] = "aaa";
  // Affiche toujours "false"
  if (sa == sb)
    printf("true\n");
  else
    printf("false\n");
  return EXIT_SUCCESS;
}
```

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
  char *sa = "aaa";
  char *sb = "aaa";
  // Peut afficher "true" ou "false"
  if (sa == sb)
    printf("true\n");
  else
    printf("false\n");
  return EXIT_SUCCESS;
}
```

Opérations sur les chaînes de caractères (1)

```
int    strncmp(const char* lhs, const char* rhs, size_t count);
char  *strncpy(char *dest, const char *src, size_t count);
char  *strncat(char *dest, const char *src, size_t count);
char  *strndup(const char *src, size_t size);
size_t strlen(const char* str);
```

- Déclarées dans le fichier d'en-tête `string.h`
- `strncmp` : compare lexicographiquement au plus `count` caractères des deux chaînes (retourne 0 si les deux chaînes sont égales)
- `strncpy` : Copie au plus `count` caractères de la chaîne `src` vers `dest`
- `strncat` : Concatène au plus `count` caractères de la chaîne `src` vers `dest`
- `strndup` : Copie la chaîne `src` vers une nouvelle chaîne allouée automatiquement sur le tas
- `strlen` : Retourne la taille d'une chaîne (sans le caractère de fin)

https://en.cppreference.com/w/c/string/byte/strncmp
https://en.cppreference.com/w/c/string/byte/strncpy
https://en.cppreference.com/w/c/string/byte/strncat
https://en.cppreference.com/w/c/string/byte/strndup
https://en.cppreference.com/w/c/string/byte/strlen.html

Opérations sur les chaînes de caractères (2)

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

int main(int argc, char *argv[]) {
  char *dest;
  char *src = "Hello world!";
  strncpy(dest, src, strlen(src) + 1);
  printf("%s\n", dest);
  return EXIT_SUCCESS;
}
```

Opérations sur les chaînes de caractères (3)

```
ouessant:~/tmp$ gcc -Wall -pedantic -std=c11 -g -o test test.c
test.c: Dans la fonction « main »:
test.c:8:3: attention: « dest » est utilisé sans avoir été initialisé [-Wuninitialized]
    8 |   strncpy(dest, src, strlen(src) + 1);
      |   ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
test.c:6:9: note: « dest » a été déclaré ici
    6 |   char *dest;
      |         ^~~~

ouessant:~/tmp$ ./test
zsh: segmentation fault (core dumped)  ./test
```

Que s'est-il passé (le message du compilateur, bien qu'un simple avertissement est une piste) ?

Opérations sur les chaînes de caractères (4)

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

int main(int argc, char *argv[]) {
  char *dest;
  char *src = "Hello world!";
  strncpy(dest, src, strlen(src) + 1);
  printf("%s\n", dest);
  return EXIT_SUCCESS;
}
```

En définissant `dest`, le compilateur a alloué la mémoire pour le pointeur lui-même (et bien évidemment pas pour ce vers quoi il pourrait pointer), et sa valeur (l'adresse vers laquelle il pointe) est non définie.

`strncpy` a essayé d'écrire les caractères vers cette adresse, et donc n'importe où en mémoire, très probablement vers une zone non allouée, ce qui a entraîné une l'arrêt brutal de notre programme par le système d'exploitation.

Opérations sur les chaînes de caractères (5)

Correction

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

int main(int argc, char *argv[]) {
  char *dest;
  char *src = "Hello world!";
  dest = malloc(strlen(src) + 1);
  strncpy(dest, src, strlen(src) + 1);
  printf("%s\n", dest);
  return EXIT_SUCCESS;
}
```

On aurait aussi pu :

- Définir `dest` comme un tableau de bonne taille (`char dest[13];`)
- Utiliser `strndup`

Opérations sur des zones mémoires

```c
int   memcmp(const void* lhs, const void* rhs, size_t count);
void *memcpy(void *dest, const void *src, size_t count);
void *memset(void *dest, int c, size_t count);
```

- Déclarées dans le fichier d'en-tête `string.h`
- `memcmp` : compare lexicographiquement le contenu de deux zones mémoires de taille `count` octets (retourne 0 si elles sont égales)
- `memcpy` : copie `count` octets depuis `src` vers `dest`
- `memset` : remplit les `count` premiers octets de la zone pointée par `dest` avec la valeur de l'octet de poids faible de `c`

https://en.cppreference.com/w/c/string/byte/memcmp
https://en.cppreference.com/w/c/string/byte/memcpy
https://en.cppreference.com/w/c/string/byte/memset

Manipulation des adresses et des pointeurs

Manipulation des adresses et des pointeurs

Le C permet de manipuler explicitement les adresses et les pointeurs

- Conversion
  - Il est possible de convertir un entier en un pointeur (c'est dépendant de l'implémentation)
  - Il est possible de transformer un pointeur vers un type en un pointeur vers un autre type
    - Depuis/vers `void *` (ex. pour `malloc`, `memcpy`…)
    - Depuis/vers `char *` pour accéder octet par octet à une zone
- Arithmétique des pointeurs
  - Il est possible d'incrémenter/décrémenter un pointeur (`+/-/++/--/+=/-=`)
  - Il est possible d'utiliser `[]` sur un pointeur
- ⚠ Dans tous les cas
  - La valeur d'un pointeur doit être valide (correspondre à un objet global, local ou sur le tas)
  - Les contraintes d'alignement doivent être respectées
  - Dans le cas contraire, le résultat est **non défini** (et le plus souvent votre programme recevra une erreur de segmentation)

Signification de [] sur un pointeur

Accès au n-ème élément d'un tableau ou d'un pointeur

```c
#include <stdio.h>

// Tableau, valeurs stockées successivement
int array[] = {1, 2, 3, 4, 5, 6};

int main(int argc, char *argv[]) {
  // Pointeur vers le premier élément du tableau
  int *ptr = array;

  printf("ptr    : %p\n", ptr);
  printf("&ptr[2]: %p\n", &ptr[2]);
  printf("ptr[2] : %d\n", ptr[2]);

  return EXIT_SUCCESS;
}
```

```
ouessant:~/tmp$ ./pointer-address
ptr    : 0x402020
&ptr[2]: 0x402028
ptr[2] : 3
```

- Mémoire

Adresse	Valeur (32 bits)	Valeur (32 bits)	Valeur de
	Valeur (64 bits)		
0x402020	0x00000001	0x00000002	array
0x402028	0x00000003	0x00000004	array
0x402030	0x00000005	0x00000006	array
...	...	...	...
0x7f....	0x0000000000402020		ptr

- Adresse calculée par `ptr[2]`

  `0x402020` $+ 2 *$ `sizeof(int)`

Arithmétique sur les pointeurs

Même fonctionnement que précédemment

```c
#include <stdio.h>

// Tableau, valeurs stockées successivement
int array[] = {1, 2, 3, 4, 5, 6};

int main(int argc, char *argv[]) {
  // Pointeur vers le premier élément du tableau
  int *ptr = array;

  printf("ptr       : %p\n", ptr);
  printf("ptr + 2   : %p\n", ptr + 2);
  printf("*(ptr + 2): %d\n", *(ptr + 2));

  return EXIT_SUCCESS;
}
```

```
ouessant:~/tmp$ ./pointer-address2
ptr       : 0x402020
ptr + 2   : 0x402028
*(ptr + 2): 3
```

- Mémoire

Adresse	Valeur (32 bits)	Valeur (32 bits)	Valeur de
	Valeur (64 bits)		
0x402020	0x00000001	0x00000002	array
0x402028	0x00000003	0x00000004	array
0x402030	0x00000005	0x00000006	array
...	...	...	...
0x7f....	0x0000000000402020		ptr

- Adresse calculée par `ptr + 2`

  $\texttt{0x402020} + 2 * \texttt{sizeof(int)}$

Arithmétique sur les pointeurs

Écrire `ptr[index]` ou `ptr + index` est similaire

- Calcule une adresse relative à la valeur du pointeur
  - `Adresse` + `index` ∗ `sizeof( <Type> )` ou
    `Adresse` − `index` ∗ `sizeof( <Type> )`
- Les pointeurs peuvent également être incrémentés ou décrémentés (`++` or `--`)
  - Le fonctionnement est le même

Exemple : Arithmétique des pointeurs (1)

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

// Les éléments du tableaux sont stockés successivement en mémoire
int array[] = {1, 2, 3, 4, 5, 6};

int main(int argc, char *argv[]) {
  int *ptr = array;                          // Pointe vers le début du tableau
  for(int i = 0; i < 3; i++) {               // Incrémente le pointeur, élément par élément
    printf("ptr: %p\t*ptr: ", ptr);
    printf("0x%08x ", *ptr++); printf("0x%08x\n", *ptr++);
  }

  printf("\nbyteswise:\n");                  // Incrémente le pointeur, octet par octet
  for(char *charptr = (char*)array; charptr < (char*)array + sizeof(array); charptr++)
    printf("%02x", *charptr);
  printf("\n");

  return EXIT_SUCCESS;
}
```

pointer-increment.c.

Exemple : Arithmétique des pointeurs (2)

```
ouessant:~/tmp$ ls
pointer-increment.c

ouessant:~/tmp$ gcc -Wall -pedantic -std=c11 -g -o pointer-increment pointer-increment.c

ouessant:~/tmp$ ls
pointer-increment pointer-increment.c

ouessant:~/tmp$ ./pointer-increment
ptr: 0x402030   *ptr: 0x00000001 0x00000002
ptr: 0x402038   *ptr: 0x00000003 0x00000004
ptr: 0x402040   *ptr: 0x00000005 0x00000006

byteswise:
01000000020000000300000004000000050000000600000 0
```

Annexes

Crédits

- Le cours se base sur celui écrit par Florian Brandner pour 3TC31