Programmation en C

Alexandre Grison

Ceci est un cours de programmation axé sur le langage **C**.

Chapter 1. Introduction

> C est un langage de programmation impératif et généraliste. Inventé au début des années 1970 pour réécrire UNIX, C est devenu un des langages les plus utilisés. De nombreux langages plus modernes comme C++, C#, Java et PHP reprennent des aspects de C.
>
> — Wikipedia

Le langage C est un langage compilé et de bas niveau, c'est à dire qu'on peut traduire son code en instructions directement comprises par un ordinateur (à l'opposé des **langages interprétés**), et qu'il laisse libre accès à la **gestion de la mémoire** de l'ordinateur, là où d'autres langages de programmation s'en occupent pour le développeur (exemple: Python).

C'est un langage très utilisé dans divers domaines de l'industrie informatique, en réalité on en trouve partout.

Par exemple:

- systèmes d'exploitation
- navigateurs web
- éditeurs de textes
- jeux vidéos
- informatique embarquée: satelittes, automobile, ...
- etc.

Chapter 2. Hello, World!

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    printf("Hello, world!\n");
    return 0;
}
```

Chapter 3. Ecosystème et Outils

3.1. Versions

Il existe plusieurs versions du langage C:

- K&R C: pour Kernighan et Ritchie
- ANSI C / ISO C
- C99: introduit en 1999
- C11: publié en 2011
- Embedded C: qui contient des extensions non standard pour le monde embarqué (fixed point arithmetic, specific I/O operations)

3.2. Compilateur

On trouve plusieurs compilateurs pour le langage C:

- **gcc**
- **Clang**
- **Tiny C Compiler**: Créé par Fabrice Bellard, un ingénieur Français (génie auteur de **LZEXE**, **FFmpeg**, **QEMU**, **BPG image format**, ...)
- **g**: Puisque le C est un sous-ensemble du C, un compilateur C++ est parfaitement capable de compiler du C.
- **Borland C++**
- **Intel C++**
- **Visual C++**

Dans ce cours nous utiliserons **gcc** (ou peut-être **mingw** sur Windows)

En réalité **gcc** est un **driver**, c'est à dire qu'il va lancer un pré-processeur nommé **cc1** pour le langage C et **cc1plus** pour C++), puis il lancera le linker **ld** (voir plus bas).

Clang fait de même, sur mon Mac **gcc** appelle même **Apple LLVM** (**clang**) qui lance à son tour **cc1** et **ld** ;-)

3.3. Linker

Le compilateur produit pour chaque fichier source `.c` un fichier objet `.o`. Le linker a pour role d'assembler ces fichiers objets ainsi que les différentes bibliothèques que vous utilisez (par exemple `readline`) en un fichier exécutable.

Le linker le plus souvent rencontré est `ld`.

3.4. Débogueurs

Un programme C peut être compilé avec support de debug, afin de pouvoir stopper son exécution et analyser l'état du programme pour vérifier son fonctionnement

On trouve **gdb** qui permet d'effectuer ces tâches. De plus, vous trouverez également un programme nommé **Valgrind** qui permet d'effectuer du profilage de code et de vous aider à trouver d'éventuelles fuites mémoires.

3.5. Editeurs de code / IDE

Il existe des centaines de programmes qui vous permettront d'éditer votre code source C.

On les sépares en deux catégories:

- Editeurs
  - Vim
  - Emacs
  - Nano
  - VS Code
  - Atom
  - Notepad++
  - …
- IDE
  - Visual Studio
  - CLion
  - CodeBlocks
  - …

N'importe quel éditeur fera l'affaire, mais il est évident qu'avoir un support de la coloration syntaxique, de l'auto-complétion, surlignage des erreurs est un vrai plus pour éviter les erreurs les plus communes.

3.6. Bonnes pratiques

Voici un ensemble de bonnes pratiques qu'il est bon d'essayer de respecter, il n'est parfois pas possible de le faire, mais essayer c'est déjà l'essentiel.

Vous les trouverez sur cette page Github.

Chapter 4. Types & Variables

4.1. Types

Le langage C possède de nombreux types différents, parmi lesquels nombres entiers, nombres décimaux, charactères, structures, etc.

Table 1. Exemple de tailles des différents types:

Type	Taille
char	0 à 255
signed char	-128 à 127
unsigned char	0 à 255
short / signed short	-32 768 à 32 767
int	-2 147 483 647 à 2 147 483 647
long	-9 223 372 036 854 775 808 à 9 223 372 036 854 775 807

Listing 1. On peut trouver les limites des types dans <limits.h>

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

int main()
{
   printf("signed char = %d à %d\n", SCHAR_MIN, SCHAR_MAX);
   printf("unsigned char = %d\n", UCHAR_MAX);
   printf("short = %d à %d\n", SHRT_MIN, SHRT_MAX);
   printf("int = %d à %d\n", INT_MIN, INT_MAX);
   printf("long = %ld à %ld\n", LONG_MIN, LONG_MAX);

   return 0;
}
```

4.2. Variables

Le langage C permet de déclarer des variables avec le format suivant: `type nom;` ou `type nom = valeur;` pour initialiser la variable avec une valeur initiale.

Listing 2. Par exemple:

```
// pas d'initialisation
int nombre1;
// avec initialisation
int nombre2 = 2;
// plusieurs déclarations
int nombre3 = 3, nombre4 = 4;
```

Chapter 5. Structures de contrôle

5.1. If/Else

```
if (condition)
{
    // code à exécuter si la condition est vraie.
}
else
{
    // code à exécuter sir la condition n'est pas vraie.
}
```

5.2. Boucles

5.2.1. For

```
for (/* initialisations */; /* condition */; /* executé après chaque tour de boucle
*/)
{
    // code à exécuter à chaque tour de boucle
}
```

Par exemple pour imprimer 5 fois `Hello` à l'écran, on initialise une variable nommée `i` à 0, on demande de boucler tant que la variable `i` a une valeur inférieure à `5`, et on indique qu'après chaque tour de boucle, la variable `i` soit incrémentée de `1`.

```
for (int i = 0; i < 5; i = i + 1)
{
    printf("Hello\n");
}
```

5.2.2. While / Do While

```
while (/* condition */)
{
    // code à exécuter tant que la condition est vraie.
}
```

```
do {
    // code à exécuter au moins une fois, puis tant que la condition est vraie.
}
while (/* condition */);
```

5.3. Arrêts et continuation de boucles

5.3.1. Arrêts

Il est possible d'arrêter une boucle avec le mot clé `break`.

Par exemple ce programme permet de demander à un utilisateur indéfiniment son prénom jusqu'à ce que celui-ci fasse au minimum plus de 3 caractères.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <readline/readline.h>

int main(int argc, char **argv)
{
    char *name = NULL;
    for (;;)
    {
        name = readline("Prénom: ");
        if (strlen(name) >= 3)
        {
            break;
        }
        free(name);
    }

    printf("Merci %s!\n", name);
    free(name);
}
```

C'est une alternative au programme suivant:

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

int main(int argc, char **argv)
{
    char *name = NULL;
    do
    {
        if (name != NULL)
        {
            free(name);
        }

        name = readline("Prénom: ");
    } while (strlen(name) < 3);

    printf("Merci %s!\n", name);
    free(name);
}
```

L'avantage du mot clé `break` est de pouvoir interrompre une boucle avant qu'elle se termine.

Considérons le programme suivant:

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

int main(int argc, char **argv)
{
    const char *str = "Bonjour";
    bool found = false;
    int i = 0;
    int size = strlen(str);
    for (; i < size; i++)
    {
        if (str[i] == 'j')
        {
            found = true;
        }
    }
    printf("Le texte contient la lettre 'j' ? %s; i vaut: %d\n",
           (found ? "oui" : "non"), i);

    found = false;
    for (i = 0; i < size; i++)
    {
        if (str[i] == 'j')
        {
            found = true;
            break;
        }
    }
    printf("Le texte contient la lettre 'j' ? %s; i vaut: %d\n",
           (found ? "oui" : "non"), i);
}
```

L'exécution donne le résultat suivant:

```
Le texte contient la lettre 'j' ? oui; i vaut: 7
Le texte contient la lettre 'j' ? oui; i vaut: 3
```

On voit que le fait d'utiliser le mot clé `break` a permis d'intérrompre la boucle et d'éviter 4 itérations à celle-ci.

Imaginez si la chaine de caractère avait une taille bien plus conséquente, par exemple 1 million de caractères?

5.3.2. Continuations

Il est également possible d'ignorer un tour de boucle et de passer directement à la prochaine à l'aide du mot clé `continue`.

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    for (int i = 0; i <= 5; i++)
    {
        if (i == 3)
        {
            continue;
        }

        printf("%d\n", i);
    }

    return 0;
}
```

Le code ci-dessus n'affichera pas 3:

```
0
1
2
4
```

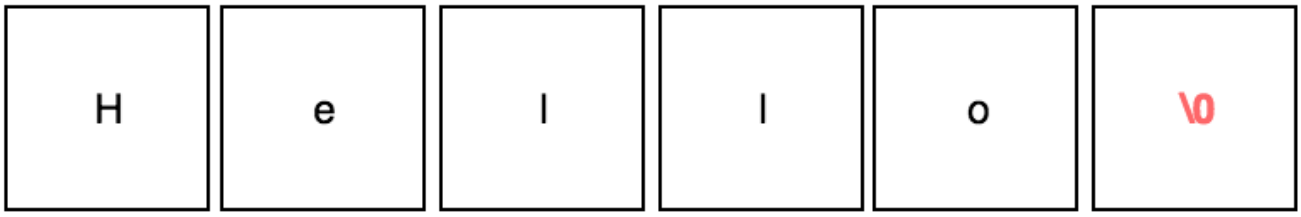
5.4. Switch

Some text

Chapter 6. Chaines de caractères

La gestion des chaines de caractère en C peut très vite devenir compliqué, il y'a de nombreux concepts dont il faut tenir compte, et la gestion de la mémoire est sujete à de nombreuses erreurs.

Les chaines de caractères sont stockées sous forme de tableau, chaque case contenant un caractère, suivi d'un caractère final indiquant la fin de chaine `\0`.



Vous trouverez ci-dessous une liste non exhaustive de fonctions qui vous permettrons de réaliser les différentes manipulations courantes sur les chaines de caractères, ainsi que quelques conseils sur quelle fonction savoir utiliser et savoir ne pas utiliser.

6.1. A l'aide de la libC

6.1.1. Taille

- `strlen(const char* s)`
- `strnlen(const char* s, size_t maxlen)`

La fonction `strlen` calcule la taille de la chaine de caractère passée en paramètre.

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

int main(int argc, char **argv)
{
    const char *s = "Hello";
    printf("%lu\n", strlen(s)); // affiche: 5
    return 0;
}
```

La fonction `strnlen` est sensiblement la même que `strlen` mais n'ira pas tenter de calculer la taille d'une chaine de caractère plus grande que le paramètre `maxlen`.

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

int main(int argc, char **argv)
{
    const char *s = "Hello";
    printf("%lu\n", strnlen(s, 3)); // affiche: 3
    printf("%lu\n", strnlen(s, 10)); // affiche: 5
    return 0;
}
```

6.1.2. Recherche d'un caractère

- `index(const char* s, int c)`
- `rindex(const char*s, int c)`

La fonction `index` trouve la première occurence du caractère `c` dans la chaine de caractère `s`. La fonction `index` permet egalement de trouver le caractère de fin de chaine `\0` si besoin.

La fonction `rindex` fait la même chose mais trouve la dernière occurence.

Si le caractère ne se trouve pas dans la chaine de caractère, alors `index` et `rindex` renvoient `NULL`.

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

int main(int argc, char **argv)
{
    const char *s = "abc def def";
    printf("%s\n", index(s, 'd'));  // affiche: def def
    printf("%s\n", rindex(s, 'd')); // affiche: def
    printf("%s\n", index(s, 'z'));  // affiche: (null)
    printf("%s\n", rindex(s, 'z')); // affiche: (null)
    return 0;
}
```

6.1.3. Comparaison

- `strcmp(const char* s1, const char* s2)`
- `strncmp(const char* s1, const char* s2, size_t n)`

La fonction `strcmp` compare alphabétiquement deux chaines de caractère `s1` et `s2` pour trouver laquelle est censée se trouver avant l'autre.

La fonction `strncmp` fais la même chose mais ne comparera au maximum que les `n` premiers caractères des deux chaines de caractères. L'utilité étant de pouvoir comparer des chaines de caractères de tailles différentes.

Ces fonctions renvoient :

- un entier < à `0` si la chaine `s1` se trouve alphabétiquement avant `s2`
- `0` si les chaines sont identiques
- un entier > à `0` si la chaine `s2` se trouve alphabétiquement après `s2`

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

int main(int argc, char **argv)
{
    const char *s1 = "abc def def";
    const char *s2 = "abc zef zef";
    printf("%d\n", strcmp(s1, s2)); // affiche: < 0
    printf("%d\n", strcmp(s2, s1)); // affiche: > 0
    printf("%d\n", strncmp(s1, s2, 3)); // affiche: 0
    return 0;
}
```

6.1.4. Comparaison (insensible à la casse)

- `strcasecmp(const char* s1, const char* s2)`
- `strncasecmp(const char* s1, const char* s2, size_t n)`

La fonction `strcasecmp` compare alphabétiquement deux chaines de caractère `s1` et `s2` pour trouver laquelle est censée se trouver avant l'autre. Cette fonction ignore la casse, c'est à dire qu'elle ne tiens pas compte des majuscules et des minuscules.

La fonction `strncasecmp` fais la même chose mais ne comparera au maximum que les `n` premiers caractères des deux chaines de caractères. L'utilité étant de pouvoir comparer des chaines de caractères de tailles différentes.

Ces fonctions renvoient :

- un entier < à `0` si la chaine `s1` se trouve alphabétiquement avant `s2`
- `0` si les chaines sont identiques (à la casse près)
- un entier > à `0` si la chaine `s2` se trouve alphabétiquement après `s2`

Il est à noter que ces fonctions utilisent la locale de la machine, mais les fonctions `strcasecmp_l` et `strncasecmp_l` permettent d'effectuer le même test en tenant compte de la locale de la machine.

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

int main(int argc, char **argv)
{
    const char *s1 = "abc def def";
    const char *s2 = "abc zef zef";
    printf("1: %d\n", strcasecmp(s1, s2)); // affiche: < 0
    printf("2: %d\n", strcasecmp(s2, s1)); // affiche: > 0
    return 0;
}
```

6.1.5. Concaténation

- `strcat(char* s1, const char* s2)`
- `strncat(char* s1, const char* s2, size_t n)`
- `strlcat(char* s1, const char* s2, size_t size)`

La fonction `strcat` ajoute une copy de la chaine de caractère `s2` à la fin de la chaine de caractère `s1` et y ajoute un caractère de fin de chaine `\0`. La chaine de caractère `s1` doit avoir assez d'espace disponible pour que l'opération soit possible.

La fonction `strncat` fait la même chose mais au maximum `n` caractères seront copiés de la chaine de caractère `s2`.

Ces fonctions retournent un pointeur vers la chaine de caractère `s1`.

Il est à noter qu'il est préférable de ne pas utiliser `strcat` et `strncat` car on peut facilement s'exposer à des attaques de type buffer overflow.

C'est pourquoi il est préférable d'utiliser la fonction `strlcat`, qui ont été créée pour être plus sure. La fonction `strlcat` à l'inverse de `strncat` prends en 3ème paramètre la longueur du buffer de `s1` et non la longueur maximum de caractères à copier de la chaine de caractère `s2`. Ce qui implique que `strlcat` copiera au maximum `size - strlen(s1) - 1` caractères de la chaine `s2`.

Pour finir, la fonction `strlcat` renvoie la taille totale de la chaine de caractère qu'elle a tenté de créer, il est donc possible de vérifier si la chaine `s1` n'avait pas assez de mémoire nécessaire pour permettre d'y insérer la chaine `s2` en fin.

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

int main(int argc, char **argv)
{
    char s1[20] = "Hello, ";
    const char *s2 = "world!";
    printf("%s\n", strcat(s1, s2)); // affiche: Hello, world!

    char s3[20] = "Hello, ";
    printf("%s\n", strncat(s3, s2, 3)); // affiche: Hello, wor

    char s4[20] = "Hello, ";
    printf("taille: %lu, chaine: %s\n",
           strlcat(s4, s2, sizeof(s4)),
           s4); // affiche: taille: 13, Hello, world!

    // ici nous avons une chaine trop petite pour y accueillir "world!""
    char s5[10] = "Hello, ";
    unsigned long total_size = strlcat(s5, s2, sizeof(s5));
    if (total_size > sizeof(s5))
    {
        printf("La taille nécessaire est de %lu\nmais la taille disponible était de
%lu\n",
               total_size, sizeof(s5));
        printf("Résultat: %s\n", s5);
    }
    else
    {
        printf("Il y'avait assez de placer pour concaténer\nles deux chaines de
caractère\n");
        printf("Résultat: %s\n", s5);
    }
    return 0;
}
```

Résultat:

```
Hello, world!
Hello, wor
taille: 13, chaine: Hello, world!
La taille nécessaire est de 13
mais la taille disponible était de 10
Résultat: Hello, wo
```

6.1.6. Copies de chaines de caractères

- `strcpy(char* dest, const char* source)`

- `strncpy(char* dest, const char* source, size_t len)`
- `stpcpy(char* dest, const char* source)`
- `stpncpy(char* dest, const char* source, size_t len)`

Les fonctions `strcpy` et `stpcpy` copient la chaine de caractères `source` vers la zone mémoire `dest` (incluant le caractère de fin de chaine `0`).

Les fonctions `strncpy` et `stpncpy` copient au plus `len` caractères de la chaine de caractères `source` vers la zone mémoire `dest` (incluant le caractère de fin de chaine `0`). Si la taille de `source` est plus petit que `len` alors le reste de la zone mémoire `dest` est remplie à l'aide de caractère de fin de chaine `\0`. A noter que dans le cas contraire `dest` ne sera pas alors terminée par `\0`.

Les fonctions `strcpy` et `strncpy` retournent un pointeur vers la zone mémoire `dest`, alors que les fonctions `stpcpy` et `stpncpy` retournent un pointeur vers le caractère `\0` de la chaine `dest`. Si comme indiqué auparavant la chaine `dest` n'est pas terminée à l'aide de `\0` par `stpncpy` alors la fonction renverra vers `dest[len]`.

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

int main(int argc, char **argv)
{
    const char *s1 = "Hello";
    char *s2 = malloc(strlen(s1) * sizeof(char));
    char *rs2 = strcpy(s2, s1);
    char *s3 = malloc(strlen(s1) * sizeof(char));
    char *rs3 = strncpy(s3, s1, 3);

    // s1: Hello
    printf("s1: %s\n", s1);
    // s2: Hello ; taille: 5 ; *rs2: H
    printf("s2: %s ; taille: %lu ; *rs2: %c\n", s2, strlen(s2), *rs2);
    // s3: Hel ; taille: 3 ; *rs3: H
    printf("s3: %s ; taille: %lu ; *rs3: %c\n", s3, strlen(s3), *rs3);

    char *p2 = malloc(strlen(s1) * sizeof(char));
    char *ps2 = stpcpy(p2, s1);
    char *p3 = malloc(strlen(s1) * sizeof(char));
    char *ps3 = stpncpy(p3, s1, 3);

    // p2: Hello ; taille: 5 ; *ps2: 0
    printf("p2: %s ; taille: %lu ; *ps2: %d\n", p2, strlen(p2), *ps2);
    // p3: Hel ; taille: 3 ; *ps3: 0
    printf("p3: %s ; taille: %lu ; *ps3: %d\n", p3, strlen(p3), *ps3);

    free(s2);
    free(p2);
    free(s3);
    free(p3);

    return 0;
}
```

6.1.7. Duplication

- `char* strdup(const char *s1)`
- `char* strndup(const char *s1, size_t n)`

Les fonctions `strdup` et `strndup` allouent suffisament de mémoire pour créer une copie de la chaine de caractère `s1`, créent la copie et retourne un pointeur vers cette zone mémoire. Cette zone mémoire étant allouée par `strdup` c'est donc à vous de ne pas oublier de la rendre quand vous n'en avez plus besoin à l'aide de la fonction `free()`.

Si il n'y a plus assez de mémoire pour créer une copie alors les fonctions `strdup` et `strndup` retournent `NULL` et la variable globale `errno` vaut `ENOMEM`.

La fonction `strndup` copie au plus `n` caractères de la chaine `s1`.

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

int main(int argc, char **argv)
{
    char *name1 = strdup("John");
    char *name2 = strdup(name1);

    printf("name1: %s @ %p\n", name1, name1);
    printf("name2: %s @ %p\n", name2, name2);

    free(name1);
    free(name2);

    return 0;
}
```

Donne à l'éxecution:

```
name1: John @ 0x7fa67ec02770
name2: John @ 0x7fa67ec02780
```

On remarque que les chaines sont équivalentes, mais qu'elles n'ont pas la même adresse mémoire, ce sont donc des copies à part entière, et modifier la chaine `name2` ne modifiera donc pas la chaine `name1`.

6.1.8. Scindage

- `char* strtok(char *str, char *separator)`
- `char* strsep(char **stringp, const char *delim)`

La fonction `strtok` permet de scinder la chaine `str` suivant un séparateur `separator`. Il faut l'appeler plusieurs fois pour récupérer les différents tokens. Si le séparateur ne peut plus être trouvé dans le reste de la chaine, alors `strtok` renvoie `NULL`.

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

int main(int argc, char **argv)
{
    char *names = strdup("Alexandre, Julien, Eva, Amélie");
    char separator[] = ", ";

    printf("Recherche de prénoms dans: %s\n", names);

    char *name = strtok(names, separator);
    while (name != NULL)
    {
        printf("  Prénom: %s\n", name);
        name = strtok(NULL, separator);
    }
    printf("Etat de la variable names après recherche: %s\n", names);

    free(names);

    return 0;
}
```

Qui donne après exécution:

```
Recherche de prénoms dans: Alexandre, Julien, Eva, Amélie
  Prénom: Alexandre
  Prénom: Julien
  Prénom: Eva
  Prénom: Amélie
Etat de la variable names après recherche: Alexandre
```

Comme on peut le voir, l'utilisation de `strtok` a modifié la chaine de caractères `names`.

Exemple de la même fonction avec `strsep`:

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

int main(int argc, char **argv)
{
    char *names = strdup("Alexandre, Julien, Eva, Jessica");
    char separator[] = ", ";

    printf("Recherche de prénoms dans: %s\n", names);
    char *name = NULL;
    while ((name = strsep(&names, separator)) != NULL)
    {
        if (!strlen(name))
            continue;

        printf("  Prénom: [%s]\n", name);
    }
    printf("Etat de la variable names après recherche: %s\n", names);
    free(names);

    return 0;
}
```

Qui donne après exécution:

```
Recherche de prénoms dans: Alexandre, Julien, Eva, Jessica
  Prénom: [Alexandre]
  Prénom: [Julien]
  Prénom: [Eva]
  Prénom: [Jessica]
Etat de la variable names après recherche: (null)
```

Comme on peut le voir, l'utilisation de `strsep` a également modifié la chaine de caractères `names`.

6.2. Alternatives

Il existe des librairies que vous pourrez trouver sur internet vous permettant de manipuler des chaines de caractères un plus simplement qu'avec les fonctions de la libC.

Par exemple la librairie **sds** que vous pourrez trouver à l'adresse suivante: https://github.com/antirez/sds.

Cette librairie permet des usages avancées comme la fusion de chaines de caractères, l'élagage (trim en anglais), l'allocation dynamique de mémoire, etc.

Chapter 7. Tableaux

Les tableaux (**array** en anglais) sont une structure de données de taille fixe qui peuvent stocker séquentiellement une collection d'éléments.

Au lieu de déclarer 10 variables de type `int` par exemple `num1`, ... `num10` on peut déclarer une variable de type tableau de 10 éléments.

Pour déclarer un tableau en C on doit déclarer son type, son nom et entre crochets le nombre d'éléments:

```
type variable[taille];
// example
int numbers[10];
```

Pour initialiser un tableau en une seule opération il est possible si on connait la taille à l'avance de faire comme le code suivant :

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

Cela fonctionne également si la taille n'est pas connue mais peut-être déterminée par le compilateur:

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

Il est possible d'accéder à un élément en lecture et en écriture à l'aide de sa position dans le tableau. Les positions commencent à l'indice `0`, c'est à dire qu'un tableau de 5 éléments contient les éléments aux indices `0`, `1`, `2`, `3` et `4` :

```
int numbers[] = {1, 2, 3, 4, 5};
printf("%d\n", numbers[0]); // 1
numbers[3] = 40; // met à jour le 4ème élément (commence à 0)
// numbers vaut {1, 2, 3, 40, 5};
```

7.1. Tableaux multi dimensionnels

Les tableaux peuvent avoir de multiples dimensions, par exemple pour représenter une matrice identité de taille 3 il est possible de la déclarer de la sorte:

```
int matrix[3][3] = {{1, 0, 0}, {0, 1, 0}, {0, 0, 0}};
// pour accéder au deuxième colonne de la deuxième ligne:
printf("%d\n", matrix[1][1]);
```

Dans ce cas particulier il faut déclarer au moins les tailles des `n-1` dimensions, donc pour un

tableau à deux dimensions, il faut au moins déclarer le nombre des colonnes.

```
// invalide:
int matrix[][] = {{1, 0, 0}, {0, 1, 0}, {0, 0, 0}};
// valide
int matrix[][3] = {{1, 0, 0}, {0, 1, 0}, {0, 0, 0}};
// valide
int matrix[3][3] = {{1, 0, 0}, {0, 1, 0}, {0, 0, 0}};
```

Chapter 8. Directives

Chapter 9. Gestion de la mémoire

Le langage C laisse au programmeur la gestion de la mémoire, il est donc important de comprendre les différents concepts.

Vous avez pu voir avec l'utilisation de chaines de caractère, ou bien de tableaux dont on ne connait pas la taille qu'on a donc besoin d'allouer de la mémoire.

Dans ce cas la libC nous fournit à travers la bibliothèque `<stdlib.h>` 4 fonctions dont on ne pourra se passer: `malloc`, `calloc`, `realloc` et `free`.

9.1. `malloc`

C'est la fonction la plus connue, elle sert à allouer de la mémoire. Pour ça il vous suffit de lui dire quelle taille on souhaite allouer (en nombre d'octets). La taille dépends évidemment du type de données à stocker. Cette taille est calculable à l'aide de la fonction `sizeof`.

Par exemple pour allouer de la mémoire suffisante pour plasser 10 nombres entiers on écrira:

```
int *nombres = malloc(10 * sizeof(int));
```

Par exemple on peut créer un programme interactif qui demandera à l'utilisateur de saisir un nombre de prénoms qu'il souhaite entrer, et les stocker dans un tableau:

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

int main(int argc, char **argv)
{
    int number = 0;
    char **names = NULL;

    printf("Combien de noms voulez-vous saisir ?\n");
    scanf("%d", &number);

    int size = number * sizeof(char *);
    printf("-----\n");
    printf("On alloue %d octets\n", size);
    // on alloue autant de mémoire que demandé par l'utilisateur
    names = malloc(size);
    for (int i = 0; i < number; i++)
    {
        names[i] = readline("Prénom: ");
    }

    printf("-----\n");
    for (int i = 0; i < number; ++i)
    {
        printf("Vous avez saisi: %s\n", names[i]);
        // ne pas oublier de libérer chaque chaine alloué par readline()
        free(names[i]);
    }

    // ne pas oublier de libérer la mémoire allouée plus tot
    free(names);

    return 0;
}
```

Qui donne par exemple le résultat suivant :

```
Combien de noms voulez-vous saisir ?
2
-----
On alloue 16 octets
Prénom: Alexandre
Prénom: Eva
-----
Vous avez saisi: Alexandre
Vous avez saisi: Eva
```

9.2. calloc

La fonction `malloc` sert à allouer une zone mémoire. La fonction `calloc` sert à allouer une zone mémoire contigue. La mémoire allouée est alors pré-initialisée avec des bytes de valeur `0`.

Une autre différence avec `malloc` est que `calloc` fais le calcul de la taille seul. Cette fonction demandant le nombre d'éléments et la taille de chaque élement, là où pour `malloc` nous devions faire la multiplication nous même.

Le programme suivant est donc équivalent au programme juste plus haut :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <readline/readline.h>

int main(int argc, char **argv)
{
    int number = 0;
    char **names = NULL;

    printf("Combien de noms voulez-vous saisir ?\n");
    scanf("%d", &number);

    // on alloue autant de mémoire que demandé par l'utilisateur
    names = calloc(number, sizeof(char *));
    for (int i = 0; i < number; i++)
    {
        names[i] = readline("Prénom: ");
    }

    printf("-----\n");
    for (int i = 0; i < number; ++i)
    {
        printf("Vous avez saisi: %s\n", names[i]);
        // ne pas oublier de libérer chaque chaine alloué par readline()
        free(names[i]);
    }

    // ne pas oublier de libérer la mémoire allouée plus tot
    free(names);

    return 0;
}
```

9.3. realloc

La fonction `realloc` permet de réallouer de la mémoire supplémentaire pour une zone mémoire déjà allouée. Imaginons le scenario où l'on prévoit de stocker N prénoms dans une zone mémoire,

mais finalement l'utilisateur veut en tapper plus. Il serait possible d'allouer une autre zone mémoire de la taille initiale + la nouvelle taille et de recopier les anciens éléments vers la nouvelle zone mémoire, mais ce serait couteux pour rien. C'est là que la fonction `realloc` rentre en compte.

Par exemple, voici un programme qui demande à un utilisateur de saisir un prénom jusqu'à ce qu'il entre un prénom vide. A chaque nouveau prénom on réalloue de la mémoire à l'aide de `realloc`.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <readline/readline.h>

int main(int argc, char **argv)
{
    char **names = NULL;

    printf("Saisissez autant de prénoms que vous voulez.\nTerminez par un prénom vide
pour quitter.\n\n");
    names = malloc(1 * sizeof(char *));
    int number = 1;
    for (;; number++)
    {
        // number - 1 car les tableaux commencent à l'indice 0
        names[number - 1] = gets(); //("Prénom: ");
        if (strlen(names[number - 1]) == 0)
        {
            break;
        }
        // ici number + 1 car on ajoute un car on veut allouer
        // de la mémoire pour un prochain prénom
        names = realloc(names, (number + 1) * sizeof(char *));
    }

    for (int i = 0; i < number; ++i)
    {
        if (strlen(names[i]))
        {
            printf("Vous avez saisi: %s\n", names[i]);
        }
        // ne pas oublier de libérer chaque chaine alloué par readline()
        //free(names[i]);
    }

    // ne pas oublier de libérer la mémoire allouée plus tot
    //free(names);

    return 0;
}
```

Chapter 10. Fonctions

On a vu jusqu'à maintenant l'utilisation de nombreuses fonctions, mais pourquoi ne pas créer les notres?

Une fonction se présente ainsi:

```
type_de_retour nom_de_la_fonction(liste des parametres) {
   corps de la fonction

   return valeur;
}
```

Une fonction en C possède une signature (souvent dans un fichier `.h`) et une implémentation (dans un fichier `.c`).

- **Type de retour** – une fonction peut renvoyer une valeur, par exemple un `int`, un `char*`. Parfois les fonctions ne renvoient rien (c'est souvent appelé une **procédure**), et dans ce cas le mot clé à indiquer en type de retour est **void**.
- **Nom de la fonction** – c'est le nom de la fonction, que vous utiliserez pour l'appeler, il faut donc faire attention de ne pas utiliser un nom de fonction déjà pris. Certains langages de programmation possèdent un système de **namespacing** pour éviter ce genre de cas (Java, C++, ...)
- **Liste des paramètres** – une fonction n'est pas obligée de prendre un paramètre, et elle peut en prendre plusieurs aussi
- **Corps de la fonction** - une fonction peut exécuter des choses diverses et variées, c'est ce qui définit son fonctionnement.

Voici quelques exemples:

Calculer le maximum entre deux entiers.

```
int max(int a, int b)
{
    int result;
    if (a > b)
    {
        result = a;
    }
    else
    {
        result = b;
    }

    return result;
}
```

Que l'on peut simplifier par:

```
int max(int a, int b)
{
    if (a > b)
    {
        return a;
    }
    else
    {
        return b;
    }
}
```

Ou encore:

```
int max(int a, int b)
{
    return a > b ? a : b;
}
```

Il existe deux fonctions `toupper` et `tolower` dans la bibliothèque `ctype.h` qui permettent de renvoyer l'équivalent en majuscule et minuscule d'un caractère donné.

Tentons de faire la même chose sur une chaine de caractère entière. Voici deux façons de faire, l'une renvoyant `void` et modifiant directement la chaine de caractère passée en paramètre, et l'autre créant une copie du paramètre et modifiant cette copie avant de la renvoyer.

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

/*
  Modifie la chaine de caractère str pour remplacer chaque
  caractère par leur équivalent en majuscule.
*/
void str_toupper(char *str)
{
    int size = strlen(str);
    for (int i = 0; i < size; i++)
    {
        str[i] = toupper(str[i]);
    }
}

/*
  Crée une copie de la chaine de caractère str
```

```
  pour y remplacer chaque caractère par son
  équivalent en minuscule.
  Enfin cette copie est renvoyée, il est donc à charge
  de la fonction appelante de libérer la mémoire à l'aide
  de la fonction free().
*/
char *str_tolower(char *str)
{
    int size = strlen(str);
    char *lower_str = malloc(size * sizeof(char));
    for (int i = 0; i < size; i++)
    {
        lower_str[i] = tolower(str[i]);
    }
    lower_str[size] = '\0';

    return lower_str;
}

int main(int argc, char **argv)
{
    // attention, ici nous utilisons char[] s1 = et non pas pas char* s1 =
    // autrement s1 pointera sur de la mémoire non modifiable
    char s1[] = "bonjour";
    printf("s1: %s\n", s1);
    str_toupper(s1);
    // ici on voit bien que la variable s1 a été modifiée directement
    // on a donc perdu sa valeur originale
    printf("s1: %s\n", s1);

    // ici on voit que la variable s2 est une copie en minuscule de s1
    // mais que s1 n'a pas été modifiée et reste entièrement en majuscules.
    char *s2 = str_tolower(s1);
    printf("s1: %s\n", s1);
    printf("s2: %s\n", s2);
    // comme indiqué dans le commentaire de la fonction str_tolower
    // il ne faut pas oublier de rendre la mémoire qui a été utilisé
    // lorsqu'on n'en a plus besoin.
    free(s2);

    return 0;
}
```

Quels sont les avantages de l'une et de l'autre ?

Vous remarquez aussi l'utilisation de `malloc`, nous allons y revenir plus en détail dans la prochaine partie.

Chapter 11. Pointeurs

Comme on l'a déjà aperçu, la fonction `malloc` permet d'allouer une portion de mémoire pour y stocker des données. Ces zones mémoires dynamiques sont accessibles en lecture et écriture à l'aide de variables qu'on appelle pointeurs, car elle pointent vers des zones mémoires.

Les pointeurs sont omniprésents dans le langage C, on en a déjà utilisé sans vraiment s'en rendre compte, et ils sont très important à la compréhension du langage. Vous n'irez pas bien loin sans eux, il est donc important de les comprendre et de démystifier tout ça :)

11.1. Adresses

Toute variable a une adresse dans la RAM de l'ordinateur, même une variable simple comme un entier ou même un simple caractère.

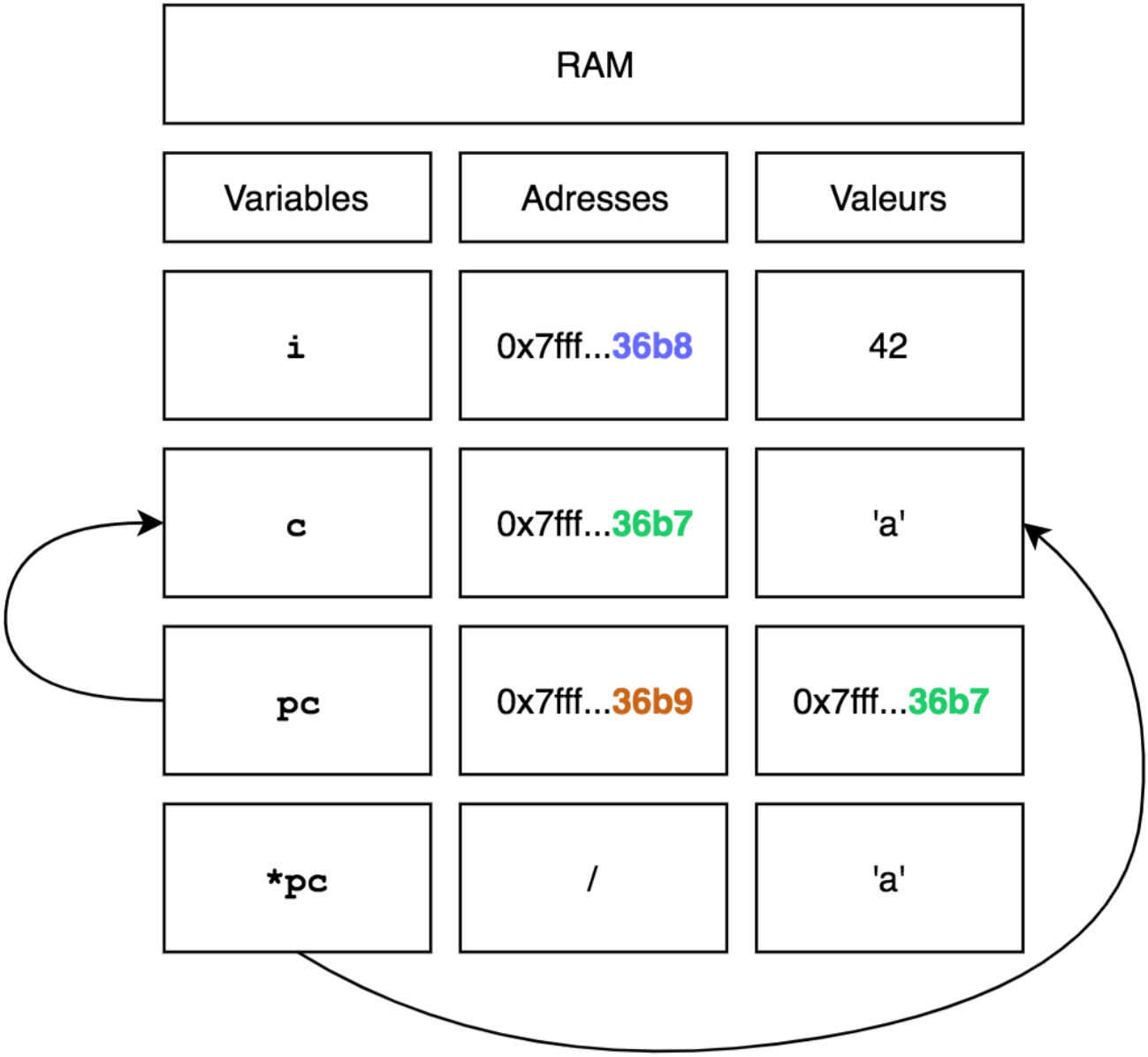
```
int i = 42;
printf("i = %d\n", i);         // i = 42
printf("&i = %p\n", &i);       // &i = 0x7fff525836b8
printf("*(&i) = %d\n", *(&i)); // *(&i) = 42

char c = 'a';
printf("c = %c\n", c);         // c = a
printf("&c = %p\n", &c);       // &c = 0x7fff525836b7
printf("*(&c) = %c\n", *(&c)); // *(&c) = a

char *pc = &c;                 // variable de type pointeur vers caractère
printf("pc = %c\n", *pc);      // on accède a la valeur pointée en utilisant *
```

Voici un petit diagramme de ce qu'il se passe au niveau mémoire, ici on retrouve les variable `i`, `c` et `pc`. Pour des raisons de compréhensions on présente également `*pc` même si ce n'est pas une variable à proprement parler, mais un dé-réferencement.

Vous pouvez aussi remarquer que la variable `pc` a sa propre adresse et est une variable à part entière.



Le langage C permet de récupérer l'adresse d'une variable en mémoire à l'aide du symbole esperluette `&`. Comme on peut le voir au dessus, la variable `c` de type caractère représente le caractère `a`, mais en utilisant `&c` on peut se rendre compte que lorsque j'ai lancé ce programme sur ma machine c'est à l'adresse hexadécimale `0x7fff525836b7` que sa valeur `a` a été stockée dans la RAM. Cette zone mémoire est en quelque sorte déterminée par l'OS de votre ordinateur.

Puisqu'un pointeur contient l'adresse mémoire d'une valeur, il nous faut un méchanisme qui nous permet de récupérer cette valeur. C'est avec le symbole `*` que l'ont peut faire ceci. C'est ainsi que ces expressions sont équivalentes:

```
char c = 'a';
char *pc = &c;

pc == &c;
*pc == *(&c) == c
```

On l'a vu juste un peu plus tôt lors de l'écriture de la fonction `str_toupper` qui modifie directement

la chaine de caractère qu'on lui passe en paramètre. C'est ce même fonctionnement qui est à l'oeuvre ici. Ce n'est pas un méchanisme utilisé uniquement pour les chaines de caractère. Dès lors que vous avez l'adresse d'une variable, ou un pointeur qui pointe vers cette adresse mémoire, vous pouvez la lire, ou la modifier.

Par exemple, créons une fonction qui multiplie par 2 l'entier passé en paramètres :

```
void pow2(int* a)
{
    *a = *a * 2; // ou encore *a *= 2;
}

int x = 5;
multiply2(&x);
printf("x vaut : %d\n", x);  // 10  (5 x 2)

int *px = &x;
multiply2(px);
printf("x vaut : %d\n", x);  // 20  (10 x 2)
```

Avec ce nouveau savoir en poche, pouvez-vous expliquer cette façon d'implémenter la fonction `strlen` ?

```
int strlen(const char *str)
{
    const char *s = str;
    while (*s)
        s++;
    return s - str;
}
```

Chapter 12. Entrées

12.1. à l'aide de <stdio.h>

12.1.1. Ouverture d'un fichier

- `FILE* fopen(char* file, char* mode)`

La fonction `fopen` permet d'ouvrir un fichier selon un mode spécifique:

- **r** ouvre le fichier en lecture, le pointeur de flux est placé au début du fichier ;
- **r+** ouvre le fichier en lecture et écriture. Le pointeur de flux est placé au début du fichier ;
- **w** tronque le fichier à son début ou ouvre le fichier en écriture. Le pointeur de flux est placé au début du fichier ;
- **w+** ouvre le fichier en lecture et écriture. Le fichier est également créé s'il n'existait pas auparavant. Si le fichier n'existait pas alors sa longueur est ramené à 0. Le pointeur de flux est placé au début du fichier ;
- **a** ouvre le fichier en mode ajout (écriture en fin de fichier, mode **append**). Le fichier est également créé s'il n'existait pas auparavant. Le pointeur de flux est placé à la fin du fichier ;
- **a+** ouvre le fichier en lecture et ajout (écriture en fin de fichier). Le fichier est également créé si il n'existait pas. La tête de lecture initiale du fichier est placée au début du fichier, mais l'écriture se fait en fin de fichier;

12.1.2. Lecture d'un caractère

- `int getc(FILE *stream)`

Retourne le prochain caractère (si présent) dans un flux d'entrée (par exemple un fichier ou encore `stdin`).

Le caractère lu est retourné sous forme d'un `unsigned char` converti en `int`. Si le flux est fermé ou a atteint la fin alors cette methode retourne `EOF` (c'est à dire `-1`). La valeur `EOF` étant un entier valide (`int`) il faut alors utiliser les fonctions `feof()` et `ferror()` afin de déterminer entre la fin de lecture du flux ou un erreur lors de la lecture.

En cas d'erreur la variable globale nommée `errno` est alors remplie afin d'indiquer l'erreur qui a eu lieue (cette variable est accessible en incluant la bibliothèque `<errno.h>`.

Afin de déterminer le message d'erreur associé à la valeur de la variable globale `errno` on pourra utiliser la fonction `char* strerror(int code)` de la bibliothèque `<string.h>`.

Exemple de lecture d'un fichier:

```
$ echo Hello > hello.txt
$ gcc hello.c -o hello.exe && ./hello.exe
> H (72)
> e (101)
> l (108)
> l (108)
> o (111)
$
```

hello.c

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

int main(int argc, char **argv)
{
    FILE *hello = fopen("hello.txt", "r");
    if (hello == NULL)
    {
        printf("Erreur: %s\n", strerror(errno));
        return 1;
    }

    int c = 0;
    while ((c = fgetc(hello)) != EOF)
    {
        printf("> %c (%d)\n", c, c);
    }

    fclose(hello);

    return 0;
}
```

12.1.3. Lecture depuis l'entrée standard

- `int getchar(void)`
- `char* gets(char *)`

Exactement la même chose que `getch(stdin)`. La fonction `getc` permet de lire un caractère dans un flux d'entrée comme expliqué plus haut. La fonction `getchar` fait exactement la même chose mais en forçant la lecture depuis `stdin` (donc un terminal et non un fichier).

Exemple de lecture depuis l'entrée standard:

```
$ gcc hello.c -o hello.exe
$ echo Hello | ./hello.exe
> H (72)
> e (101)
> l (108)
> l (108)
> o (111)
$
```

hello.c

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    int c = 0;
    while ((c = getchar()) != EOF)
    {
        printf("> %c (%d)\n", c, c);
    }

    return 0;
}
```

Ici on remarque que le fait de prendre le flux d'entrée sortie évite d'avoir à ouvrir un fichier, s'assurer qu'il s'est ouvert sans problème et également de devoir le fermer avant la fin de programme.

La fonction `gets` essaye de lire autant de caractères possibles jusqu'à un caractère de saut de ligne depuis l'entrée standard (`stdin`). Ce caractère de saut de ligne n'est pas présent dans la chaine de caractère retournée par `gets`.

Il est fortement déconseillé d'utiliser la fonction `gets`, car elle n'est pas utilisable de façon sécurisée et il est souvent très difficile voir impossible de déterminer à l'avance la taille de la chaine de caractère qui sera lue. Ceci provoque des problèmes de sécurité permettant à un utilisateur malicieux d'exploiter une faille de type `buffer overflow`.

Il est donc très conseillé d'utiliser la fonction `fgets` (voir plus bas).

12.1.4. Lecture depuis un flux d'entrée

- `char fgetc(FILE *stream)`
- `char* fgets(char readContent, int maxSize, FILE stream)`

Les fonctions `fgetc` et `getc` sont sensiblement les mêmes, et ont le même paramètre d'entrée et de sortie. Les seules différences entre `getc` et `fgetc` sont que `getc` peut être implémenté sous forme de macro C alors qu'il est assuré que `fgetc` est une réelle fonction, ce qui implique:

- L'argument passé en paramètre à `getc` doit être une expression sans effet de bord ;
- Comme `fgetc` est une fonction il est possible de récupérer son adresse et la passer en paramètre à d'autres fonctions poir une exécution plus tard ;
- L'appel à `fgetc` peut être moins efficiente car appeler une fonction est plus couteux en temps machine (mais négligeable) que du code déjà `inline` par une macro.

La fonction `fgets` est le complément à la fonction `gets` comme `fgetc` l'est à `getc`. Elle prends donc les mêmes paramètres d'entrée et renvoie la même chose que la fonction `gets`.

Exemple de lecture de fichiers.

```
$ echo Hello > hello.txt
$ gcc hello.c -o hello.exe && ./hello.exe
> Hello
> taille: 6
$ echo "Hello\neverybody" > hello.txt
$ gcc hello.c -o hello.exe && ./hello.exe
> Hello
> taille: 6
$ gcc hello-full.c -o hello.exe && ./hello.exe
> Hello
> taille: 6
> everybody
> taille: 10
```

hello.c

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

int main(int argc, char **argv)
{
    FILE *hello = fopen("hello.txt", "r");
    if (hello == NULL)
    {
        printf("Erreur: %s\n", strerror(errno));
        return 1;
    }

    char content[100] = "";
    fgets(content, 100, hello);
    printf("> %s", content);
    int contentSize = strlen(content);
    printf("> taille: %d\n", contentSize);

    fclose(hello);

    return 0;
}
```

L'exemple suivant montre comment récupérer successivement toutes les lignes d'un fichier, puisque `fgets` s'arrête à la lecture d'un caractère de type saut de ligne (`\n`):

hello-full.c

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

int main(int argc, char **argv)
{
    FILE *hello = fopen("hello.txt", "r");
    if (hello == NULL)
    {
        printf("Erreur: %s\n", strerror(errno));
        return 1;
    }

    char content[100] = "";
    while (fgets(content, 100, hello) != NULL)
    {
        printf("> %s", content);
        int contentSize = strlen(content);
        printf("> taille: %d\n", contentSize);
    }

    fclose(hello);

    return 0;
}
```

On notera l'utilisation du test de retour de la fonction `fgets` qui renvoie `NULL` lorsqu'on a atteint la fin de fichier, nous permettant d'arrêter notre boucle `while`.

12.2. à l'aide de GNU Readline

GNU Readline est une bibliothèque qui permet au developpeur de proposer des interfaces en ligne de commande avec des fonctionalités telles que l'auto complétion, l'édition de la ligne en train d'être tappée, ou d'une gestion de l'historique. C'est une logiciel libre maintenu par le projet GNU.

Elle peut donc être très utile et vous faire gagner du temps et éviter tout un tas de soucis ;-).

12.2.1. Lecture d'une ligne

- `char* readline(char* prompt)`

La fonction de base de readline permet de proposer un prompt à l'utilisateur et de récupérer sa saisie.

Example de demande de saisie à l'aide de `readline`:

```
$ gcc hello.c -o hello.exe -lreadline
$ ./hello.exe
Quel est votre prénom ?
> Alexandre
Vous avez saisi: Alexandre
```

Vous noterez l'utilisation du paramètre `-lreadline` au compilateur `gcc` qui lui demande d'utiliser la librairie (`-l`) `readline` lors de la compilation pour y trouver l'implémentation de la fonction `readline` telle que décrit dans le fichier d'en-tête `<readline/readline.h>`.

hello.c

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

int main(int argc, char **argv)
{
    char *name = readline("Quel est votre prénom ?\n> ");
    printf("Vous avez saisi: %s\n", name);
    free(name);

    return 0;
}
```

Lors de l'exécution du programme, il se met en pause après avoir imprimé `> ` et attends que l'utilisateur saisisse quelque chose puis appuye sur la touche entrée pour valider.

Il est à noter que la chaine de charactère est allouée par la fonction `readline` et que sa zone mémoire doit être libérée lorsqu'on n'en a plus besoin à l'aide de la méthode `free()` qui se trouve dans la bibliothèque `<stdlib.h>`.

Par défaut vous remarquez également que l'utilisateur peut auto-compléter le noms des fichiers qui se trouvent dans le répertoire d'où l'on a lancé notre programme.

C'est une des fonctionnalités offertes par la bibliothèque readline. Vous en trouverez de nombreuses en vous rendant sur la documentation en ligne àl 'adresse suivante: http://web.mit.edu/gnu/doc/html/rlman_2.html, à l'aide de `man readline` ou bien encore évidemment de Google ;)

Chapter 13. Sorties

13.1. Ecriture d'un caractère dans un flux

- `int putc(int c, FILE *stream)`
- `int fputs(char* s, FILE* stream)`

La fonction `putc` permet d'écrire un charactère (converti en `unsigned char`) dans un flux de sortie.

```
$ gcc hello.c -o hello.exe && ./hello.exe
$ cat hello.txt
abc
$
```

hello.c

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

int main(int argc, char **argv)
{
    FILE *hello = fopen("hello.txt", "w");
    if (hello == NULL)
    {
        printf("errno:\n\tcode: %d\n\tmessage: %s\n",
               errno, strerror(errno));
        return 1;
    }

    putc('a', hello);
    putc('b', hello);
    putc('c', hello);
    putc('\n', hello);

    fclose(hello);

    return 0;
}
```

La fonction `fputs` permet d'écrire une chaîne de caractère dans un flux de sortie.

Le résultat de la fonction `fputs`est un entier positif si tout s'est bien passé, et `EOF` en cas d'erreur d'écriture.

Exemple d'écriture d'une chaine de caractère sur la sortie standard:

```
$ gcc hello.c -o hello.exe && ./hello.exe
$ cat hello.txt
Hello
$
```

hello.c

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

int main(int argc, char **argv)
{
    FILE *hello = fopen("hello.txt", "w");
    if (hello == NULL)
    {
        printf("errno:\n\tcode: %d\n\tmessage: %s\n",
               errno, strerror(errno));
        return 1;
    }

    fputs("Hello\n", hello);

    fclose(hello);

    return 0;
}
```

13.2. Ecriture dans la sortie standard

- `int putchar(int c)`
- `int puts(char* s)`

La fonction `putc` permet d'écrire un caractère dans un flux de sortie comme expliqué plus haut. La fonction `putchar` fait exactement la même chose mais en forçant l'écriture sur `stdout` (donc un terminal et non un fichier).

Exemple d'écriture sur la sortie standard:

```
$ gcc hello.c -o hello.exe
$ ./hello.exe
abc
$
```

hello.c

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    putchar('a');
    putchar('b');
    putchar('c');
    putchar('\n');

    return 0;
}
```

La fonction `puts` permet d'écrire une chaîne de caractère dans la sortie standard (`stdout`).

Il est à noter que la fonction `puts` ajoutera automatiquement un caractère de saut de ligne à la suite de la chaine de caractère que vous voulez imprimer.

Exemple d'écriture d'une chaine de caractère sur la sortie standard:

```
$ gcc hello.c -o hello.exe
$ ./hello.exe
Hello
$
```

hello.c

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    puts("Hello");

    return 0;
}
```

Chapter 14. Alternatives au C

Comme vous pouvez le constater, le C est un langage de bas niveau qui permet de gérer la mémoire nous même. Cela a ses avantages et aussi ses inconvénients. Pour des tâches simples comme la manipulation de chaines de caractères par exemple, ce n'est pas le langage le plus simple à utiliser.

Plusieurs langages ayant des performances similaires mais étant plus simples à utiliser, avec de nombreuses bibliothèques disponibles existent.

14.1. Google Go

Le langage Go, créé par Google est une bonne alternative au langage C. Go est un langage qui contient ce qu'on appelle un garbage collector, comme en Java, c'est à dire qu'il gère lui même la mémoire.

Il apporte de nombreux avantages comme:

- simplicité d'écriture
- temps de compilation extrêment plus rapide (80% à 90% plus rapide que du C), ce qui le rends même utilisable pour du scripting
- gestion du multi-threading en standard
- nombreuses bibliothèques en standard
    - chaines de caractère
    - HTTP et Web
    - Cryptographie
    - ...

14.2. Rust

Le langage Rust, créé par Mozilla est un langage très récent, extrêmement performant, pouvant être plus compliqué à utiliser que le C car introduisant de nombreux autres concepts, reste pour des programmes simples une bonne alternative. Un langage plein de bonnes pratiques qu'il est très intéressant d'étudier.