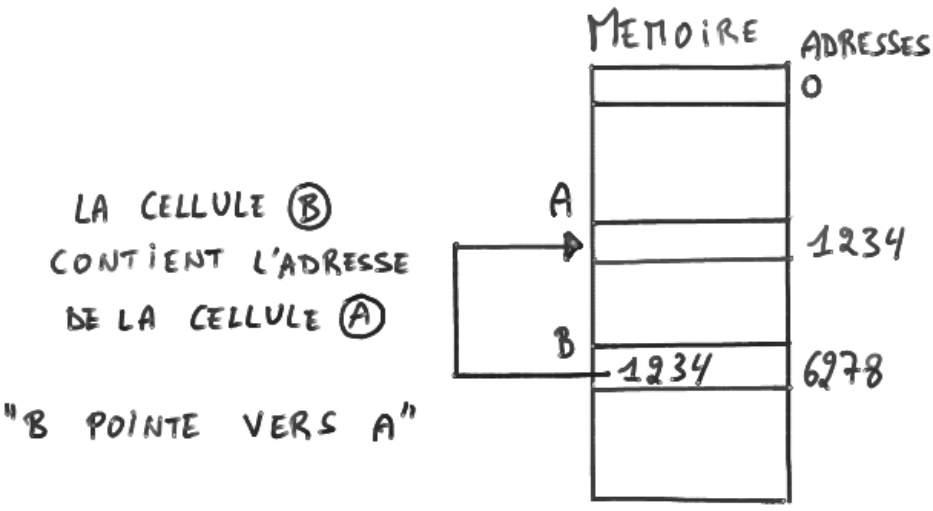
SINF1252
Résumé Programmation C (partie 2)

Faut-il avoir peur des pointeurs ?

Les pointeurs sont un des éléments les plus perturbants lorsque l'on débute la programmation en C. C'est aussi celui avec lequel les étudiants éprouvent habituellement le plus de difficultés. Pourtant, les pointeurs ne sont pas une notion particulièrement compliquée. Cette section tente de démystifier quelque peu la chose...

Introduction

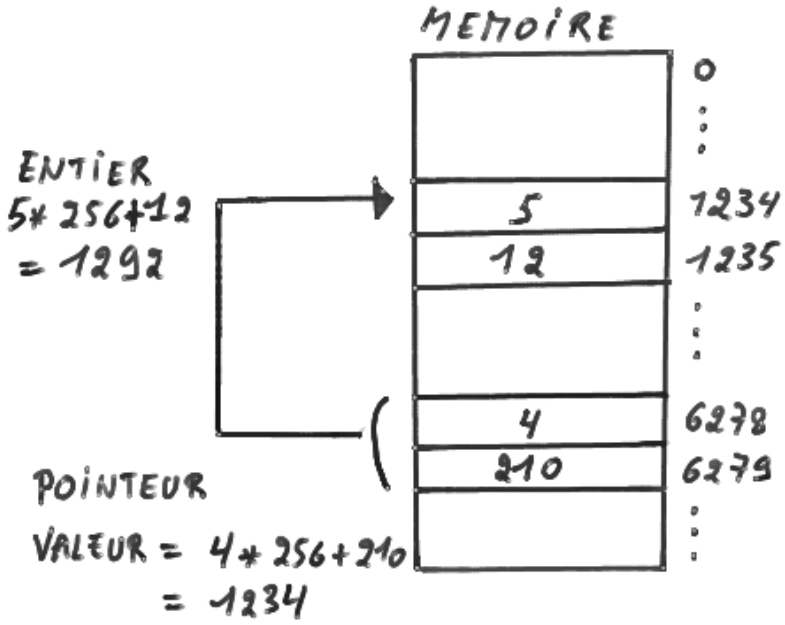
Un pointeur est avant tout une variable qui contient un nombre. La particularité de ce nombre est qu'il représente une adresse en mémoire. Si on voit la mémoire comme un grand tableau d'octets, une adresse est simplement l'index d'une cellule dans le tableau. Un pointeur est donc un moyen pratique d'indexer la mémoire. Pour représenter un pointeur visuellement, on utilise souvent une flèche. On peut donc très facilement imaginer qu'un pointeur est une cellule de mémoire qui pointe vers une autre cellule de mémoire.



Un pointeur, comme toute variable, occupe une certaine place en mémoire. Par exemple, sur une architecture à base de processeur Intel x86, un pointeur a une taille de 32 bits, i.e. 4 octets. Un tel pointeur permet de représenter des adresses

comprises entre 0 et 2^{32}-1. On dit que le pointeur peut addresser 2^{32} cellules d'un octet. Sur une machine à base d'AMD Opteron, un pointeur a une taille de 64 bits. Il est donc possible d'adresser 2^{64} octets.

Jusqu'à présent, nous avons parlé d'un pointeur comme l'adresse d'une cellule ou octets unique dans la mémoire. Mais un pointeur peut être typé. Par exemple, on peut définir un pointeur vers un entier. Un entier peut être représenté par plusieurs cellules (octets). Dans ce cas, le pointeur vers un entier représente l'adresse de la première des cellules qui forment le nombre entier. Supposons une architecture à 16 bits où les pointeurs et les entiers sont encodés avec 2 octets[1]. Soit un entier dont le premier octet est stocké à l'adresse 1234 et le second octet à l'adresse 1235. Un pointeur vers cet entier contiendra la valeur 1234 qui est l'adresse du premier octet de l'entier. Cette situation est illustrée à la figure suivante. De manière générale, on peut définir un pointeur vers n'importe quel type supporté par le langage. En particulier, on peut définir un pointeur vers un pointeur.



La déclaration d'un pointeur en C se fait de la même façon qu'une déclaration de variable classique, à l'exception de l'ajout d'un astérisque "*****" entre le type et le nom de variable. L'exemple suivant montre la déclaration d'une variable entière nommée **var** et la déclaration d'un pointeur vers un entier nommé **ptr_var**.

```
int var;
int * ptr_var;
```

En théorie, on peut affecter au pointeur **ptr_var** une valeur constante, de la même manière qu'on affecte une valeur à la variable **var**. Par exemple, les assignations de l'exemple suivant sont autorisées. Comme on peut affecter n'importe quelle valeur à un pointeur, on constate que l'on peut accéder à n'importe quelle zone de la mémoire, même en dehors des variables déclarées. Il s'agit donc d'un outil très puissant, mais également très dangereux[2].

```
var= 123;
ptr_var= 10;
```

En pratique, on affecte à un pointeur l'adresse d'un élément déjà connu. Par exemple, on peut mettre dans le pointeur **ptr_var** l'adresse de la variable **var**. On obtient l'adresse d'une variable en utilisant le symbole "**&**" (signe "et"). Par exemple:

---

[1] La taille d'un entier (**int**) dépend du compilateur et de l'architecture de l'ordinateur.

[2] Il est rare que l'on doive affecter des valeurs constantes aux pointeurs, sauf lorsque l'on développe le système d'exploitation lui-même ou lorsque l'on travaille sur des plateformes sans système d'exploitation.

```
ptr_var= &var;
```

En partant du pointeur **ptr_var**, il est possible d'accéder au contenu de la variable **var**. Cette opération a un nom technique un peu barbare: il s'agit d'un *déréférencement* du pointeur **ptr_var**. En langage C, le déréférencement se fait en utilisant un astérisque devant le nom du pointeur. L'expression ***ptr_var** est équivalente à la variable **var**. Par exemple, on peut lire et écrire la valeur de la variable **var** en passant par le pointeur **ptr_var**:

```
int var= 5;
int * ptr_var= &var;
*ptr_var= 6;
printf("var contient maintenant %d ou %d\n", var, *ptr_var);
```

Arithmétique des pointeurs

En plus de l'affectation de valeur et du déréférencement, il est possible d'effectuer des opérations arithmétiques sur les pointeurs. Ces opérations permettent de calculer de nouvelles adresses à partir d'une adresse initiale. Ces opérations nécessitent quelques précisions.

Typiquement, il est possible d'additionner un entier à un pointeur. L'entier ajouté est généralement appelé offset. En ajoutant une valeur entière **Offset** à un pointeur **Pointer**, on avance d'un certain nombre de cellules dans la mémoire. Contrairement à ce qu'on pourrait imaginer, on n'avance pas nécessairement d'un nombre **Offset** de cellules. En effet, le nombre de cellules dont on avance dépend du type du pointeur, avec la formule suivante:

Pointer' = **Pointer** + (**Offset** * *sizeof* (**Type**))

La nouvelle adresse se calcule comme l'adresse initiale à laquelle on ajoute le produit de l'offset et de la taille du type. Par exemple, si on additionne 5 à un pointeur P de type caractère, on obtient P + (5 * sizeof(char)), c'est à dire P + 5 car la taille d'un caractère vaut 1. Si, par contre on additionne le même offset à un pointeur P de type entier (4 octets), la valeur finale sera P + (5 * sizeof(int)) = P + (5*4) = P + 20.

Afin d'illustrer ceci en langage C, prenons l'exemple suivant:

```
void * ptr= (void *) 1234;
int * ptr_int= (void *) 1234;
char * ptr_char= (void *) 1234;
ptr= ptr+1;
ptr_int= ptr_int+1;
ptr_char= ptr_char+1;
```

En prenant les trois pointeurs déclarés et initialisés comme ci-dessus et en leur additionnant chacun la même valeur 1, on obtiendra comme valeurs finales: `ptr=1235`, `ptr_int=1238` et `ptr_char=1235`. Il est donc important de bien comprendre les opérations d'addition sur les pointeurs.

Gestion dynamique de la mémoire

Lorsque l'on manipule des données, il faut nécessairement les stocker quelque part en mémoire. Plusieurs possibilités sont offertes au programmeur. En général l'usage d'une possibilité plutôt qu'une autre dépend des objectifs du programme et du caractère dynamique des données à stocker. Le tableau suivant décrit les trois possibilités généralement offertes au programmeur en langage C. La dernière colonne du tableau indique si c'est le programmeur ou le compilateur qui gère cette zone.

Localisation	Description	Gestion
Zone constante (parfois appelée zone de texte)	Cette zone de mémoire contient toutes les constantes définies dans le programme. Par exemple, la chaîne de caractères **“Hello”** ou l’entier **123** sont des constantes.	Compilateur
Pile (*stack* en anglais)	Cette zone de mémoire contient les variables définies dans le programme. Cela concerne aussi bien les variables locales aux fonctions que les variables globales.	Compilateur
Tas (*heap* en anglais)	Cette zone de mémoire est gérée par le programmeur. Le programmeur est responsable de l’allocation et de la libération de blocs dans cette zone.	Programmeur

Gestion de la pile

Le programmeur n’a pas à se soucier à proprement parler de la gestion de la pile. Le compilateur se charge de générer le code nécessaire pour que l’accès aux variables définies dans le programme se fasse correctement et de manière transparente. En général, il ne faut pas se soucier de l’emplacement des variables.

Cependant, une mauvaise compréhension de la gestion de la pile peut amener à une erreur fréquente pour les programmeurs débutants en C. Cette erreur consiste à retourner l’adresse d’une variable locale à une fonction. C’est le cas dans l’exemple suivant:

```
int* func(void)
{
  int a= 0;
  /*...*/
  return &a;
}
```

Le problème de la fonction ci-dessus est que l'on retourne une référence vers une variable locale à la fonction (**a**). L'espace réservé à cette variable (situé à l'adresse retournée par la fonction) pourra être ré-utilisé dès la fin de la fonction. La zone référencée par le pointeur peut donc être écrasée par d'autres variables. Ce type d'écriture mène donc en général à des résultats inattendus ou tout simplement au “crash” du programme. Le programme ci-dessus peut être solutionné de deux façons différentes (voir encarts).

Solution 1

La fonction laisse l'appelant décider de l'emplacement de la variable **a**. Cette manière de faire est courante car elle permet de passer un ou plusieurs pointeurs à la fonction, dont elle pourra utiliser ou modifier la valeur référencée. Un autre avantage de cette manière de faire est qu'elle permet de renvoyer un code d'erreur, puisque le résultat de la fonction est directement écrit dans les zones pointées par les arguments.

```
int func(int* a)
{
  *a= 0;
  /*...*/
  return 0; /*success*/
}
```

Solution 2
La fonction alloue de l'espace sur le tas avec **malloc()**. Dans ce cas il faut spécifier que l'appelant devra lui-même libérer la mémoire associée lorsque le pointeur n'est plus utilisé, en utilisant **free(a)**, ou bien on peut fournir une seconde fonction qui libère les ressources allouées par la première. Le second cas est surtout utile si la libération de ressources suppose plusieurs opérations, comme, par exemple, plusieurs **free().**

```
int* func(void)
{
  int* a= malloc(sizeof(int));
  *a= 0;
  /*...*/
  return a;
}
```

Allocation dynamique
L'allocation et la désallocation de blocs de mémoire sur le tas se fait avec des fonctions standards de la librairie C. La fonction **malloc(*N*)** permet d'allouer un bloc de N octets. La fonction retourne un pointeur vers le premier octet de ce bloc. S'il n'y a plus suffisamment de mémoire, **malloc()** retourne un pointeur invalide avec la valeur constante **NULL**. La fonction **free(*P*)** libère le bloc mémoire dont le premier octet est situé à l'adresse P. On ne devrait passer à la fonction **free()** que des pointeurs retournés par la fonction **malloc()**.

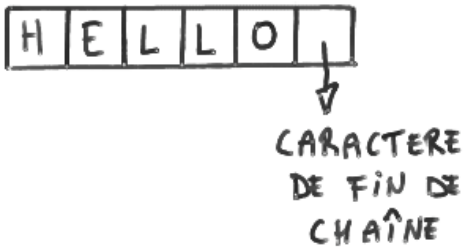
Gérer la mémoire de manière dynamique offre beaucoup de liberté, mais il y a également une contrainte. Il faut être très attentif à **libérer ce qui n'est plus utilisé !** Dans le cas contraire, on risque d'arriver à l'erreur *"out of memory"* (plus de mémoire disponible).

Sur la pile ? Sur le tas ?
Il existe des règles de bonne pratique pour la déclaration des variables. En général, on déclarera dans une fonction uniquement des variables qui ont une utilité locale, comme les "compteurs de boucle". On ne déclarera globalement (i.e. en dehors de toute fonction) que des variables contenant des données qui concernent le programme dans son ensemble. Par exemple, une table de hashage ou la tête d'une liste chaînée qui maintiennent l'état global du programme.

On recourt à une allocation dynamique lorsque l'on ne peut pas déterminer à l'avance l'espace mémoire dont on aura besoin. On réserve alors autant de mémoire que nécessaire au fur et à mesure que le besoin s'en fait sentir. Par exemple, supposons que l'on veuille conserver l'historique de ce qui a été tapé sur la ligne de commande par l'utilisateur. On peut ajouter une entrée dans une liste chaînée pour chaque ligne tapée. Comme on ne sait pas à l'avance la longueur ni le nombre de chaînes de caractères tapées par l'utilisateur, on allouera dynamiquement l'espace à l'historique. Typiquement, chaque nouvelle chaîne sera stockée dans un noeud de liste supplémentaire.

Chaînes de caractères

Une première application pratique des pointeurs est la représentation et la manipulation de chaînes de caractères. La manipulation des chaînes de caractères est une des causes principales des erreurs dans les programmes écrits en langage C par des débutants. Contrairement au langage Java, en C il n'existe pas de type prédéfini pour représenter une chaîne de caractères. Une chaîne de caractères sera simplement représentée par un tableau de caractères ou par un pointeur vers une suite de caractères. La figure suivante montre la chaîne "Hello" dans un tableau de caractères.



Une particularité du C est que la chaîne contient un caractère spécial qui indique sa fin. Ce caractère permet par exemple à une fonction d'impression qui va itérer sur chacun des caractères de s'arrêter lorsque le caractère final est lu. Le caractère final a la valeur numérique **0** et il est représenté par le caractère **'\0'**. On parle de *chaîne à zéro-terminal* ou *nul-terminated string*. Ceci à un impact important pour l'allocation de mémoire aux chaînes de caractères car il faut systématiquement prévoir un caractère supplémentaire pour stocker le zéro terminal. Par exemple, dans le cas de la chaîne "Hello" qui comprend 5 caractères, il faudra prévoir la place pour 6 caractères en prenant en compte le zéro terminal.

Voici un exemple de déclaration d'une chaîne de caractère en C sous la forme d'un tableau. La chaîne constante "Hello" est copiée dans le tableau (ainsi que le zéro terminal, qui n'est pas visible dans le programme):

```
char ma_chaine[6]= "Hello";
```

La déclaration ci-dessous est équivalente. Le pointeur **ma_chaine** contiendra l'adresse de la chaîne constante "Hello".

```
char * ma_chaine= "Hello";
```

Une autre façon de créer une chaîne de caractères consiste à allouer dynamiquement la mémoire, comme dans l'exemple suivant. Ici, la chaîne de caractères est donc représentée par un pointeur. On ne peut pas se contenter d'affecter la chaîne "Hello" au pointeur car le pointeur contiendrait alors l'adresse de la chaîne constante "Hello" et le pointeur retourné par **malloc()** serait perdu. On utilise la fonction strcpy() pour copier le contenu de la chaîne constante "Hello" dans la zone allouée par **malloc()**.

```
char * ma_chaine= (char *) malloc(6*sizeof(char));
strcpy(ma_chaine, "Hello");
```

Si on veut concaténer une chaîne de caractères à une autre, il faut allouer une troisième chaîne de caractères avec suffisamment d'espace pour stocker les deux chaînes initiales plus un zéro terminal. Par exemple[3]:

```
char * ma_chaine1= "Hello";
char * ma_chaine2= " World";
char * ma_chaine3= (char *) malloc((strlen(ma_chaine1)
                                    +strlen(ma_chaine2)+1)
                                   *sizeof(char));
strcpy(ma_chaine3, ma_chaine1);
strcat(ma_chaine3, ma_chaine2);
```

La librairie standard C contient un certain nombre de fonctions permettant de manipuler des chaînes de caractères. Ces fonctions sont définies dans le header **string.h**. Toutes ces fonctions sont documentées en long et en large dans les man

[3] Notez que cette façon de procéder n'est pas très efficace car avant de copier la seconde chaîne de caractères dans la troisième, **strcat()** doit parcourir la chaîne initiale afin de trouver sa fin.

pages. Voici quelques autres exemples utiles: **strlen()**, **strcpy()**, **strdup()**, **strcat()**, **strcmp()**, ... Il existe également des versions plus sûres de certaines fonctions telles que **strncpy()**, **strncat()**, **strncmp()**, ...

Structures de données

Les structures de données permettent d'assembler plusieurs types primitifs (comme des entiers ou des caractères) en un nouveau type de données. L'exemple suivant définit un nouveau type nommé **mon_type** qui est composé de deux champs: **a** et **b**.

```
struct mon_type {
  int a;
  char b;
};
```

On peut alors déclarer des variables de type **mon_type**. La syntaxe est identique à la déclaration de variables de types primitifs. Par exemple

```
struct mon_type var;
```

L'accès aux champs de la variable structurée var se fait en ajoutant un point "." entre le nom de la variable et le nom du champ. Par exemple, si on veut affecter les valeurs **5** et **'a'** aux champs **a** et **b** de la variable **var**, on peut écrire:

```
var.a= 5;
var.b= 'a';
```

On peut aussi créer dynamiquement la structure en mémoire (sur le tas) et y accéder avec des pointeurs. Par exemple, on peut déclarer un pointeur **ptr_var** vers **mon_type**. Il faut alors lui allouer de la mémoire avec **malloc()**. On indique la taille de la structure à **malloc()** en utilisant **sizeof(***type***)**. L'opérateur **sizeof()** retourne le nombre d'octets nécessaire pour stocker en mémoire le type donné. Il est important de remarquer que l'on pourrait calculer manuellement la taille de la structure. Par exemple, **mon_type** a deux champs: un entier encodé avec 4 octets et un caractère encodé avec 1 octet. La taille de la structure est donc 5. Cependant, la taille d'un entier peut varier d'une architecture d'ordinateur à l'autre[4]. On préfèrera donc toujours déterminer la taille d'un type en utilisant **sizeof()**.

```
struct mon_type * ptr_var= malloc(sizeof(struct mon_type));
ptr_var->a= 5;
ptr_var->b= 'a';
```

Précisions concernant opérateur sizeof()

L'opérateur **sizeof()** permet d'obtenir l'espace mémoire occupé par un type de données. Par exemple **sizeof(int)** renvoie la taille d'un entier (en général 4 octets). L'opérateur sizeof() peut également déterminer la taille d'une structure de données plus complexe. Par exemple, si l'on définit la structure **s** ci-dessous, **sizeof(struct s)** fournit la taille de la structure **s**.

```
struct s {
        int a;
        char b;
};
```

[4] La taille d'une structure peut également varier à cause de l'alignement des champs.

Pour ce qui concerne les pointeurs il faut faire plus attention. En effet, **sizeof()** retourne la taille du pointeur et non la taille de la zone de mémoire allouée. Quelque soit le type d'un pointeur, **sizeof()** retournera la même valeur, en général 4 sur une architecture 32 bits et 8 sur une architecture 64 bits. Dans l'exemple ci-dessous, **sizeof(buf)** retournera une valeur indépendante de **BUF_LENGTH**.

```
char* buf=malloc(BUF_LENGTH);
```

Par contre si l'on utilise **sizeof()** avec un tableau comme dans l'exemple suivant, la valeur retournée sera bien **sizeof(char) *BUF_LENGTH**. Ceci est dû au fait que le compilateur est capable dans ce cas de déterminer la taille occupée par le tableau **buf**, alors qu'il ne le peut pas si l'allocation est dynamique. C'est normal puisque cette taille n'est pas définitive dans le cas d'une allocation dynamique.

```
char buf[BUF_LENGTH];
```

Lecture dans un fichier

Cette section introduit une des méthodes d'accès aux fichiers en C: l'accès par flux (stream). Afin d'introduire la manipulation de fichiers, prenons comme exemple un programme **prog.c** qui lit des lignes dans un fichier et les affiche à l'écran.

```
/* Contenu de prog.c */
#include <stdio.h>
#define MAX_BUF_SIZE 1000
int main() {
  FILE * file;
  char buffer[MAX_BUF_SIZE];
  file= fopen("monfichier.txt", "r");
  if (file == NULL)
    return -1;
  while (fgets(buffer, MAX_BUF_SIZE, file) != NULL) {
    /* traitement de la ligne lue */
  }
  fclose(file);
  return 0;
}
```

L'ouverture d'un fichier est effectuée par la fonction **fopen()** de la librairie standard C (libc). Cette fonction prend en argument le nom du fichier ("monfichier.txt") et un mode d'accès. Ici, "**r**" signifie que l'on veut un accès en lecture uniquement. Une fois le fichier ouvert, nous ne pourrons pas écrire dans le fichier. D'autres modes d'accès aux fichiers sont possibles (voir la man page de **fopen()**). La fonction **fopen()** retourne un pointeur vers une valeur de type **FILE**. La structure **FILE** contient diverses informations sur l'état du fichier ouvert utilisées en interne par les fonctions de la librairie C[5].

Il est possible que la fonction **fopen()** ne puisse pas ouvrir le fichier donné en argument. Les raisons d'erreur sont multiples[6]. Par exemple, il est possible que le fichier dont le nom a été passé en argument n'existe pas. Il est aussi possible que l'utilisateur du programme n'ait pas le droit de lire le fichier. Dans le cas d'un échec d'ouverture, la valeur renvoyée par

---

[5] Il n'est pas nécessaire de connaître ou comprendre le contenu de la structure **FILE**. On dit que **FILE** est une structure opaque.

[6] Voir la man page de **fopen()** pour une liste des erreurs possibles.

fopen() est un pointeur **NULL**. On peut donc détecter une erreur en testant la valeur de retour de **fopen()**. Dans le programme d'exemple **prog.c**, on termine l'exécution (**return**) si l'ouverture du fichier échoue. Nous verrons plus loin comment traiter les erreurs plus proprement.

La fermeture du fichier est effectuée par la fonction **fclose()**. Cette fonction prend le pointeur vers la structure **FILE** comme argument. Il est important de fermer un fichier lorsqu'on ne l'utilise plus. En effet, il y a une limite au nombre de fichiers qu'un programme peut ouvrir simultanément. Cette limite dépend du système[7]. Si un programme "oublie" de refermer les fichiers qu'il n'utilise plus, il risque de ne plus pouvoir en ouvrir de nouveaux.

La lecture dans un fichier peut être effectuée par la fonction **fread()**. La fonction fread() nécessite 4 arguments. Premièrement, un pointeur vers une zone de mémoire dans laquelle **fread()** stockera l'information lue. Il faut s'assurer que la mémoire allouée à cette zone soit suffisamment grande. Deuxièmement, un nombre d'éléments à lire. Troisièmement, la taille d'un élément. Finalement, le pointeur de type FILE qui identifie le fichier ouvert. La fonction fread() retourne le nombre d'éléments lus. Si la fonction a pu lire moins d'éléments que demandé ou si une erreur s'est produite, elle retourne une valeur <= 0. On peut alors utiliser les fonctions **feof()** ou **ferror()** pour tester si on est arrivé à la fin du fichier ou si une erreur s'est produite.

Il existe également une fonction **fgets()** qui permet de lire une ligne de texte dans un fichier.

Arguments du programme

Les arguments passés au programme à la ligne de commande peuvent être accédés via deux variables passées à la fonction **main()**: **argc** qui contient le nombre d'arguments et **argv** qui contient les arguments eux-mêmes. La variable **argv** est un tableau de chaîne de caractères.

```
int main(int argc, char * argv[])
{
  unsigned int index;
  for (index= 0; index < argc; index++) {
    printf("arg[%u] : \"%s\"\n", argv[index]);
  }
}
```

La première chaîne contenue dans **argv**, à l'indice 0, est le nom du programme lui-même. Les chaînes suivantes correspondent aux arguments passés en ligne de commande. Voici un exemple d'exécution du programme ci-dessus:

```
toto@intel01$ ./prog arg1 123 plop.txt
arg[0] : ./prog
arg[1] : arg1
arg[2] : 123
arg[3] : plop.txt
toto@intel01$
```

Il faut toujours tester la valeur de **argc** avant d'accéder à **argv**. En effet, si l'utilisateur n'a passé que deux arguments au programme, l'accès à **argv[]** à lindice 3 et au delà retournera un résultat indéfini.

[7] Par exemple, sous Linux, il est commun qu'un maximum de 256 fichiers puissent être ouverts simultanément par un processus.

Résumé des fonctions utiles

Fonction	Déclaration	Description
malloc()	stdlib.h	Alloue un bloc de mémoire (sur le tas).
free()	stdlib.h	Libère un bloc de mémoire.
strcpy() / strncpy()	string.h	Copie le contenu d'une chaîne de caractères.
strlen()	string.h	Retourne la longueur d'une chaîne de caractères.
strcat() / strncat()	string.h	Ajoute une chaîne de caractères à une autre.
strcmp() / strncmp()	string.h	Compare deux chaînes de caractères.
strdup()	string.h	Duplique une chaîne de caractères. Attention, cette fonction alloue de la mémoire qu'il faudra libérer par la suite.
fopen()	stdio.h	Ouvre un fichier.
fclose()	stdio.h	Ferme un fichier.
fread()	stdio.h	Lit des blocs d'octets dans un fichier.
feof()	stdio.h	Indique si la fin d'un fichier est atteinte.
ferror()	stdio.h	Indique si une erreur s'est produite durant la dernière opération sur un fichier.
fgets()	stdio.h	Lit une ligne dans un fichier texte.
sizeof()		Détermine la taille d'un type de données. Notez que sizeof() est un opérateur du langage et non une fonction.