Fonctionnement des Ordinateurs

Initiation à la programmation en C

B. Quoitin
Faculté des Sciences
Université de Mons

Résumé

L'objectif de ce document est d'introduire les concepts de la programmation en C qui seront nécessaires durant les travaux pratiques du cours de Fonctionnement des Ordinateurs. Il ne s'agit pas d'un cours complet de programmation en C. Ce document fait l'hypothèse que des notions de programmation avec les langages Python et Java ainsi que le langage d'assemblage MIPS sont acquises.

Les exemples donnés dans ce document reposent sur l'utilisation d'un environnement de type UNIX (Linux, Mac OS X, ...). De plus, la version du langage C utilisée dans ce document est essentiellement C89 (ANSI X3.159-1989) car elle est bien supportée sur les plate-formes embarquées. Certaines fonctionnalités de la version C99 (ISO IEC 9899:1999 [JTC99]) seront également utilisées lorsqu'elles sont disponibles.

Cette introduction au C est volontairement incomplète. Elle ne traite pas de l'allocation dynamique de mémoire (`malloc`), ni des unions, structures, énumérations et accès aux fichiers. Il existe un grand nombre de bonnes références sur la programmation en C. L'étudiant souhaitant aller plus loin pourra consulter notamment l'ouvrage des créateurs du C [KR88] (uniquement C89) et une approche plus moderne [Kin08] (couvre également C99).

Table des matières

1 Exemple minimal

Le premier exemple a pour objectif d'introduire l'organisation, la compilation et l'exécution d'un programme minimal écrit en C. Le programme se limite à l'affichage d'un simple message à la console. Le programme est composé d'un seul fichier écrit en C. Son code source est fourni à la Figure 1. Le code source du programme est placé dans un fichier nommé `ex1.c`.

```
1 #include <stdio.h>
2
3 int main(int argc, char * argv[])
4 {
5   printf("Hello␣World␣!\n");
6   return 0;
7 }
```

FIGURE 1 – Exemple minimal (`ex1.c`).

1.1 Description du code source

Les paragraphes suivants détaillent le code source. Le programme est composé d'une unique fonction appelée `main` qui contient un appel à une autre fonction `printf` afin d'afficher une chaîne de caractères à la console.

Les lignes 3 à 7 définissent la fonction `main`. La déclaration d'une fonction en C est similaire à celle d'une méthode en Java. Elle comprend un type de retour, un nom, des arguments et un bloc d'instructions. Dans le cas de la fonction `main`, le type de retour est `int` et il y a deux arguments : `argc` de type `int` et `argv` de type tableau de chaînes de caractères (`char * []`). Il est important de noter qu'un programme en C ne peut contenir qu'une fonction `main`. Cette fonction sert de point d'entrée dans le programme, i.e. c'est elle qui est exécutée en premier. Les valeurs des arguments de la fonction `main` correspondent aux paramètres fournis au programme.

La ligne 5 écrit une chaîne de caractères sur la sortie standard du programme [1]. La fonction `printf` fait partie de la librairie C (libc) et sa déclaration est fournie dans le fichier `stdio.h`. Les chaînes de caractères sont délimitées par le caractère `"`. La suite `\n` est une séquence d'échappement correspondant au retour à la ligne.

La ligne 6 termine la fonction `main` et, par conséquent, le programme. La valeur retournée par `main` correspond au code d'erreur que le programme transmet au système d'exploitation [2]. Par convention, un code égal à 0 indique que le programme s'est terminé avec succès [3].

La ligne 1 contient une directive `#include` qui permet d'inclure dans le programme un fichier contenant la déclaration de la fonction `printf`. Un tel fichier est appelé fichier d'en-tête ou fichier *header*. Il peut y avoir plusieurs directives `#include` dans un fichier C. Celles-ci sont typiquement placées en début de fichier. Les fichiers *header* à utiliser pour les fonctions standard sont généralement indiqués dans la documentation (*man pages*).

1. Par défaut, la sortie standard du programme est associée à la console
2. Le code d'erreur est transmis au programme parent, typiquement un *shell*.
3. Des codes d'erreur standards `EXIT_SUCCESS` et `EXIT_FAILURE` sont définis dans `stdlib.h`

1.2 Compilation

Avant de pouvoir exécuter le programme de l'exemple, il est nécessaire de le compiler. Il existe plusieurs compilateurs C. Dans ce document, le compilateur `gcc` (GNU Compiler Collection) est utilisé.

La Figure 2 illustre la commande permettant de compiler le programme de l'exemple. Le programme `gcc` tel qu'il est invoqué se charge de plusieurs tâches : (1) la compilation du code source en code binaire (code objet), (2) l'édition des liens (*linker*) et (3) la création d'un programme exécutable. Afin de créer un programme exécutable, il doit contenir une fonction `main`.

```
bqu@:~$ gcc -Wall -Werror -o ex1 ex1.c
bqu@:~$
```

FIGURE 2 – Compilation de l'exemple minimal.

La liste ci-dessous décrit le rôle des différents paramètres et options utilisés.

- L'option `-o ex1` spécifie que le résultat de la compilation doit être placé dans le fichier `ex1`. Si cette option n'est pas utilisée, un nom par défaut est utilisé (typiquement `a.out`).
- L'option `-Wall` spécifie que tout avertissement (*warning*) doit être affiché.
- L'option `-Werror` spécifie que tout avertissement doit être considéré comme une erreur.
- Le reste des arguments passé au compilateur spécifie la liste des fichiers à compiler (typiquement des fichiers source C). Dans l'exemple, un seul fichier (`ex1.c`) est indiqué.

Afin de créer le programme exécutable `ex1`, le compilateur utilise non seulement le fichier source `ex1.c` mentionné comme argument, mais également d'autres fichiers fournis avec le compilateur. Ce processus est illustré à la Figure 3. En particulier, le fichier `stdio.h` mentionné par la directive `#include` dans `ex1.c` ainsi que la librairie C (`libc.so`). L'emplacement de ces fichiers dépend du système d'exploitation. Les fichiers d'en-tête (`.h`) sont généralement [4] situés dans `/usr/include` et les librairies dans `/lib` ou `/usr/lib`.

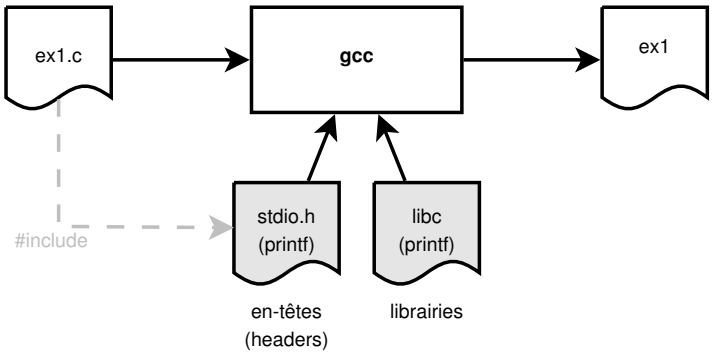

FIGURE 3 – Compilation de l'exemple minimal.

4. Il est possible de forcer `gcc` à fournir de l'information sur les chemins de recherche des *headers* et librairies en lui passant l'option `-v` lors de la compilation. Par exemple, `gcc -v -o- -E ex1.c >/dev/null`

1.3 Exécution

Si la compilation a pu être réalisée avec succès, un programme exécutable est créé. Le nom du programme créé est celui indiqué avec l'option `-o`. Une fois le programme compilé, il est possible de l'exécuter directement, sans utiliser de machine virtuelle. Le code du programme compilé est directement exécutable sur la plateforme (processeur et système d'exploitation). La Figure 4 illustre l'exécution du programme `ex1`. Le nom du programme est précédé de `./` afin d'indiquer le chemin du programme [5].

```
bqu@:~$ ./ex1
Hello World !
bqu@:~$
```

FIGURE 4 – Exécution de l'exemple minimal.

5. Le point (`.`) désigne le chemin courant, i.e. le répertoire dans lequel se trouve l'utilisateur. Afin d'éviter de devoir spécifier le chemin vers les programmes utilisés fréquemment, les systèmes UNIX utilisent une variable d'environnement appelée `PATH` qui contient une liste de répertoires dans lequel le système d'exploitation recherche les programmes à exécuter si un chemin n'est pas spécifié explicitement. Il est possible d'afficher la variable PATH avec la commande `echo $PATH`. Cette variable peut être modifiée.

2 Elements du langage C

2.1 Types primitifs

De la même manière que Java, le langage C est statiquement typé. Les variables et arguments de fonctions sont associés à un type qui ne peut être changé et qui permet au compilateur d'effectuer un certain nombre de vérifications. Les types primitifs disponibles en C permettent de représenter des entiers (signés, en complément à 2), des naturels (non-signés), des flottants (IEEE754) et des caractères. Les types supportés par C89 sont décrits à la Table 1. Pour chaque type, le tableau fournit la taille en bits de la représentation en mémoire ainsi que les plus grandes et plus petites valeurs représentables. La taille des entiers varie selon la plateforme [6]. L'opérateur `sizeof` permet de déterminer la taille d'un type ou d'une variable.

Il est à noter que le type `char` sert à la fois à représenter des entiers ou naturels que des caractères sur 8 bits. De plus, il n'existe pas de type booléen en C [7]. Les booléens sont représentés comme des entiers. La valeur 0 correspond à la valeur faux tandis que les valeurs différentes de 0 correspondent à la valeur vrai.

Depuis C99, plusieurs améliorations ont été introduites. Premièrement, le type `long long` est disponible et occupe au moins 8 octets. Deuxièmement, afin de simplifier la correspondance entre type et taille de représentation, des types de la forme `int`N`_t` et `uint`N`_t` ont également été introduits (via le fichier `stdint.h`). Dans ces types, N désigne le nombre de bits de la représentation. Les valeurs de N supportées sont 8, 16, 32 et 64. Troisièmement, plusieurs constantes disponibles dans le fichier `limits.h` renseignent sur les tailles et limites de représentation des types entiers. Par exemple, la constante `UINT_MAX` contient la valeur maximale qui peut être représentée avec le type `unsigned int`.

2.2 Déclaration de variable

La déclaration d'une variable comporte un type, un nom et une initialisation optionnelle. Une variable doit être déclarée avant de pouvoir être utilisée. La déclaration d'une variable doit précéder son utilisation dans le fichier source.

Il existe plusieurs types de variables en langage C. Le type de la variable a un impact sur sa portée (*scope*), son existence (*lifetime*), sur la façon dont elle est initialisée et sur le segment mémoire dans lequel elle est placée. Les différents types de variables sont décrits ci-dessous.

Variables locales. Une variable locale est déclarée au sein d'une fonction. Une variable locale a une portée limitée à la fonction dans laquelle elle est définie. Elle a également une existence limitée à l'exécution de la fonction. Les variables locales sont placées sur la pile. Les variables locales qui ne sont pas initialisées ont une valeur indéfinie. Il est possible d'étendre l'existence d'une variable locale à la durée du programme en utilisant le modificateur `static`. Dans ce cas, la variable locale sera stockée dans le segment `.data` plutôt que sur la pile.

Variables globales. Une variable globale est déclarée hors d'une fonction. Une variable globale a une portée qui s'étend de l'endroit où elle est définie jusqu'à la fin du fichier. Une variable locale peut également être rendue accessible à partir d'autres fichiers source (voir Section **??**). Les variables globales sont placées dans le segment de données `.bss` si elles ne sont pas initialisées explicitement ou dans le segment `.data` si elles sont intialisées [8]. Les variables globales qui ne sont pas initialisées ont une valeur égale à 0.

La Figure 5 illustre des déclarations de variables. Les variables `f` et `c` sont locales à la fonction `main`. Ces variables seront stockées sur la pile durant l'exécution de la fonction `main`. Les variables `a`, `b`, `c`, `x` et `y` sont globales. Les variables `x` et `y` sont placées dans le segment `.data` et les autres dans le segment `.bss`.

2.3 Commentaires

Deux types de commentaires sont possibles en langage C. La façon la plus standard d'écrire des commentaires (C89) est de les encadrer des séquences `/*` et `*/`. Cette forme de commentaires peut s'étaler sur plusieurs lignes. Un exemple est donné en Figure 6.

Depuis la version C99, des commentaires en une seule ligne sont également possibles en les préfixant de la séquence `//`.

6. Le type `char` occupe typiquement 1 octet. Les type `short` et `int` occupent au moins 2 octets. Habituellement, le type `int` a une taille égale à la taille du plus grand mot manipulable par le processeur, i.e. sur une machine 32 bits, un `int` a une taille de 4 octets. Le type `long` occupe au moins 4 octets.

7. Depuis C99, un type booléen nommé `_Bool` a été introduit.

8. L'initialisation des segments `.bss` et `.data` est effectué par le *C run-time initialization*, un bout de programme (souvent appelé `crt`) automatiquement ajouté par le compilateur lorsqu'un exécutable est généré. Ce bout de programme est responsable de l'initialisation d'un programme C. Le contenu des variables initialisées est copié du fichier exécutable ou du segment `.text` vers la mémoire.

	Nom	Taille	Valeur minimum	Valeur maximum
Entiers	`char`	8 bits	-128	127
	`short`	$\geq$ 16 bits	-32768	32767
	`int`	$\geq$ 16 bits	-32768	32767
	`long`	$\geq$ 32 bits	-2147483648	2147483647
Naturels	`unsigned char`	8 bits	0	255
	`unsigned short`	$\geq$ 16 bits	0	65535
	`unsigned int`	$\geq$ 16 bits	0	65535
	`unsigned long`	$\geq$ 32 bits	0	4294967296
Flottants	`float`	32 bits	-3.4E38	3.4E38
	`double`	64 bits	-1.8E308	1.8E308

TABLE 1 – Types primitifs et tailles.

```
1  int a, b, c;
2  unsigned int x= 3, y= 2;
3
4  int main(int argc, char * argv[])
5  {
6    float f;
7    char c= '\n';
8  }
```

FIGURE 5 – Exemples de déclarations de variables.

```
1  /* Ceci est un exemple de commentaire
2     multi-ligne */
```

FIGURE 6 – Exemple de commentaire multi-lignes.

```
1  // Ceci est un exemple de commentaire
2  // multi-ligne C99
```

FIGURE 7 – Exemple de commentaire C99.

2.4 Constantes

Il est possible de déclarer des constantes avec la directive `#define`. Une déclaration de constante avec `#define` peut être utilisée dans tout le fichier dans lequel elle est définie, à partir de la ligne de la définition. Une constante définie avec `#define` n'a pas de type. La Figure 8 illustre une utilisation de la directive `#define`.

```
1  #include <stdio.h>
2
3  #define PI 3.1415
4
5  int main(int argc, char * argv[])
6  {
7    float r= 25.0;
8    printf("La␣surface␣du␣cercle␣vaut␣%.2f\n",
9           r*r*PI);
10   return 0;
11 }
```

FIGURE 8 – Exemple de déclaration de constante.

Le langage C supporte également le mot clé `const` afin de créer l'équivalent de constantes. Le mot clé `const` permet de marquer une variable comme ne devant pas être changée. Une tentative de modification de cette variable sera refusée par le compilateur [9]. Un exemple est donné à la Figure 9.

2.5 Opérateurs

La plupart des opérateurs supportés par le langage C sont décrits à la Table 2. Une brève description est fournie pour chaque opérateur. Le fonctionnement des opérateurs est similaire à ceux du langage Java, mais il est important de faire attention à un certain nombre de détails repris ci-dessous.

9. Attention, il est possible de contourner `const` en modifiant une variable au travers d'un alias comme, par exemple, un pointeur.

```
1  const int x= 5;
2  x= 7; /* sera refusé par le compilateur */
```

FIGURE 9 – Utilisation du mot-clé `const`.

1. Il ne faut pas confondre l'opérateur d'affectation (=) avec l'opérateur de test d'égalité (==). Au contraire de Java, le langage C ne permet pas au compilateur de détecter cette confusion.
2. Il ne faut pas confondre les opérateurs logiques (`&&` et `||`) avec les opérateurs binaires (`&` et `|`) qui sont similaires mais pas interchangeables !
3. L'opérateur de division `/` utilisé avec des entiers produit un résultat entier (division entière). Pour obtenir la division non-entière de deux entiers, il faut effectuer un transtypage en flottant des opérandes.
4. Les règles de précédence et d'association sont similaires à celles du langage Java et ne sont pas rappelées ici.
5. Les tests d'égalité ne sont généralement pas appropriés avec les flottants et doivent être remplacés par des tests d'appartenance à un intervalle. Les raisons ne sont pas propres au langage C mais à la représentation flottante.

2.6 Boucles et instructions conditionnelles

De manière similaire à Java, le langage C supporte les boucles `for`, `while` et `do...while`. La syntaxe est similaire à celle de Java. Une application de la boucle `while` au calcul de la factorielle est illustrée à la Figure 10.

```
1  int n= 5;
2  int fact= 1;
3  while (n > 0) {
4    fact*= n-1;
5    n--;
6  }
```

FIGURE 10 – Calcul de factorielle avec une boucle `while`.

Les instructions conditionnelles de type `if` et `if...else` sont également supportées. Comme en Java, un problème d'ambiguïté peut se produire lorsque plusieurs `if...else` sont imbriqués (à quel `if` un `else` est-il rattaché ?). Une version récursive du calcul de la factorielle est illustrée à la Figure 11. La structure `if` y est utilisée pour distinguer le cas récursif du cas de base.

La structure `switch` est supportée mais ne fonctionne qu'avec des types entiers.

2.7 Tableaux

Des tableaux à 1 ou plusieurs dimensions sont supportés. Toutes les cellules d'un tel tableau ont le même type et donc la même taille. Une variable tableau contient l'adresse de la première cellule du tableau. La Figure 12 illustre la déclaration et l'utilisation d'un tableau à 1 dimension comportant 10 éléments entiers (type `int`).

	Opérateur	Description
Arithmétiques	$-a$	Opposé de a
	$a * b$	Multiplication de a et b
	a / b	Division de a par b. Division entière si a et b sont entiers.
	a % b	Reste de la division entière de a par b. Uniquement applicable pour a et b entiers.
	a + b	Addition de a et b
	a − b	Soustraction de a et b
Comparaisons	a == b	Egalité : vrai ssi $a = b$
	a != b	Différence : vrai ssi $a \neq b$
	a < b	
	a <= b	
	a > b	
	a >= b	
Logiques	! a	Négation : vrai ssi a est faux.
	a && b	Conjoncton : vrai ssi a ET b sont vrais.
	a || b	Disjonction : vrai ssi a OU b sont vrais.
Binaires	~ a	Complément : tous les bits de a sont inversés.
	a & b	ET bit à bit entre les bits de a et de b.
	a | b	OU bit à bit entre les bits de a et de b.
	a ^ b	XOR bit à bit entre les bits de a et de b.
	a << n	Décalage de n position vers la gauche des bits de a.
	a >> n	Décalage de n position vers la droite des bits de a.
Incrémentation	a++	Post-incrémentation de a.
	a−−	Post-décrémentation de a.
	++a	Pré-incrémentation de a.
	−−a	Pré-décrémentation de a.
Affectation	a= b	Affectation simple
	a+= b	Affectation composée : a devient $a + b$
	a−= b	Affectation composée : a devient $a - b$
	a*= b	Affectation composée : a devient $a * b$
	a/= b	Affectation composée : a devient a/b
	a%= b	Affectation composée : a devient $a \bmod b$
	a<<= n	Affectation composée : a devient a décalé de n bits vers la gauche
	a>>= n	Affectation composée : a devient a décalé de n bits vers la droite
	a&= b	Affectation composée : a devient a & b
	a|= b	Affectation composée : a devient a | b
	a^= b	Affectation composée : a devient a ^ b
Autres	(t) a	Transtypage : interprète a comme étant de type t.
	sizeof(a)	Retourne la taille en octets du type ou de la variable a.

TABLE 2 – Liste des opérateurs supportés en C.

```
1 int fact(int n)
2 {
3   if (n == 1)
4     return 1;
5   return n * fact(n-1);
6 }
```

FIGURE 11 – Calcul récursif de factorielle.

```
1 int tableau[10];
2 tableau[0]= 17;
3 int x= tableau[9];
4 tableau[10]= 20;
```

FIGURE 12 – Exemple de déclaration et utilisation d'un tableau.

Si la taille de `int` vaut 4 octets, ce tableau occupe 40 octets en mémoire.

L'index d'une cellule d'un tableau est compris entre 0 et $N - 1$ où N est le nombre de cellules du tableau. Au contraire du langage Java, il n'y a pas de vérification que l'accès à un tableau se fait bien dans les bornes. Dans l'exemple, l'accès à la position 10 (ligne 4) est incorrect. Le comportement lors de cet accès est indéterminé. Il peut causer un accès mémoire incorrect, l'écrasement d'une autre

donnée du programme stockée à une position qui suit le tableau et causer un comportement incorrect du programme.

Des tableaux à plusieurs dimensions peuvent être facilement créés en utilisant plusieurs paires de crochets (`[]`) lors de la déclaration du tableau. La Figure 13 illustre la création d'un tableau de `short` à 2 dimensions comportant 3 lignes et 5 colonnes. Ce tableau occupe 30 octets en mémoire si la taille d'un `short` est 16 bit. L'instruction de la ligne 1 indique comment écrire dans la cellule de ligne 1 et colonne 2.

```
1 short matrix[3][5];
2 matrix[1][2]= 22;
```

FIGURE 13 – Déclaration d'un tableau à 2 dimensions.

Les cellules d'un tableau à deux dimensions sont placées séquentiellement en mémoire ligne par ligne. Cet ordre est appelé *row-major*. L'adresse de la cellule de ligne i et de colonne j (notée `[i][j]`) est obtenue comme $b+(i*N+j)*$`sizeof(t)` où b est l'adresse de la première cellule du tableau, N est le nombre de colonnes et t est le type d'une cellule. L'agencement en mémoire des cellules du tableau `matrix` est illustré à la Figure 14. L'adresse de base du tableau est contenue dans la variable `matrix` et vaut `0x05000000`. L'adresse de la cellule de ligne 1 et de colonne 2 est donc `0x05000000` $+(1*5+2)*$`sizeof(short)`, ce qui vaut `0x0500000E`.

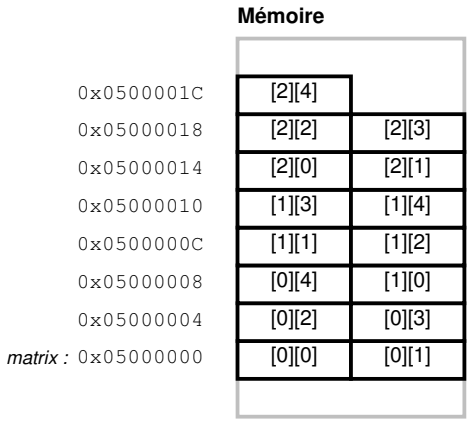

FIGURE 14 – Agencement en mémoire des cellules d'un tableau à 2 dimensions.

2.8 Pointeurs

Un pointeur est une variable qui contient une adresse mémoire. Un pointeur a des similitudes à une référence dans le langage Java. Les différences majeures entre un pointeur en C et une référence en Java sont qu'un pointeur peut désigner n'importe quel endroit dans l'espace mémoire d'un programme et que la valeur d'un pointeur peut être définie arbitrairement. Attention ! Les pointeurs sont un outil extrêmement utile mais ils peuvent facilement amener à des erreurs s'ils ne sont pas utilisés correctement.

Un variable est déclarée comme un pointeur en ajoutant un astérisque `*` entre le type et le nom. Des exemples de déclaration de pointeurs sont donnés à la Figure 15. Un pointeur est généralement typé. Cela signifie qu'il désigne l'adresse en mémoire d'un élément qui sera interprété avec un type particulier. Dans l'exemple de la Figure 15, le pointeur `ptr` est typé. Il désigne l'adresse de base d'un entier en mémoire alors que le pointeur `ptr2` est non-typé (utilisation de `void`). Le pointeur `ptr3` est un double pointeur, i.e. un pointeur destiné à contenir l'adresse d'un autre pointeur.

```
1 int * ptr1;
2 void * ptr2;
3 char ** ptr3;
```

FIGURE 15 – Déclarations de pointeurs.

Les pointeurs sont liés à plusieurs opérations spécifiques. La première de ces opérations est l'obtention de l'adresse d'une variable avec l'opérateur `&`. Par exemple, si `a` est une variable, `&a` donne l'adresse en mémoire de `a`. L'adresse obtenue est associée au type pointeur vers type de la variable `a`. Dans l'exemple de la Figure 16, la variable `a` est de type `float` et par conséquent, `&a` est de type `float *`. La variable `ptr` est initialisée avec une adresse choisie arbitrairement. La variable `ptr_invalide` est initialisée avec la constante `NULL`. Cette constante est utilisée pour marquer qu'un pointeur ne désigne pas une adresse valide.

```
1 float a= 5.37;
2 float * ptr_a= &a;
3 char * ptr= (char *) 0x5432ABCD;
4 char * ptr_invalide= NULL;
```

FIGURE 16 – Obtention de l'adresse d'une variable et initialisation d'un pointeur avec une valeur arbitraire.

La situation de la Figure 16 est illustrée graphiquement à la Figure 17. Les variables pointeurs sont stockées en mémoire tout comme les autres variables. Les flèches illustrent que les adresses contenues dans les variables pointeurs référencent d'autres emplacements en mémoire. Il est important de noter que `ptr` est l'adresse d'un octet (type `char *`) alors que `ptr_a` est l'adresse du premier des 4 octets d'un `float` (type `float *`).

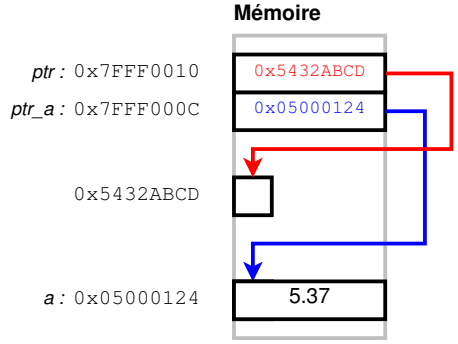

FIGURE 17 – Illustration graphique de la situation de la Figure 16.

La seconde opération importante liée aux pointeurs est l'accès en lecture ou écriture à la zone mémoire dont ils contiennent l'adresse. L'opérateur `*` permet de déréférencer un pointeur. Cet opérateur peut être utilisé pour lire ou écrire la zone mémoire, en fonction de son utilisation comme *right-* ou *left-value*. La Figure 18

illustre deux accès mémoire. Le premier, à la ligne 3, copie dans la variable `b` la valeur de la variable `a` en utilisant l'adresse de `a` stockée dans le pointeur `ptr`. Le second accès a lieu à la ligne 5 où le caractère `z` est écrit à l'adresse `0x5432ABCD`. Il est important de noter que la taille de l'espace mémoire accédé lors du déréférencement d'un pointeur dépend du type de ce pointeur. Par exemple, la lecture de `*ptr_a` concerne 4 octets (la taille d'un `float`) alors que l'écriture de `*ptr` concerne un seul octet (taille d'un `char`).

```
1 float a= 5.37;
2 float * ptr_a= &a;
3 float b= *ptr_a;
4 char * ptr= (char *) 0x5432ABCD;
5 *ptr= 'z';
```

FIGURE 18 – Accès mémoire avec des pointeurs.

Finalement, deux types d'opérations arithmétiques sont possibles avec les pointeurs. Ces opérations permettent d'une part de déterminer une nouvelle adresse à partir d'un pointeur et d'autre part de déterminer la distance entre deux pointeurs.

Déterminer une nouvelle adresse. Cette opération peut être réalisée par l'addition ou la soustraction d'un pointeur et d'un entier. Soit un pointeur d'adresse a et de type t et un entier k. L'addition du pointeur et de l'entier donnera une adresse égale à $a + k * \texttt{sizeof}(t)$. Par exemple, supposons un pointeur `ptr` de valeur `0x1234` de type `float *`. L'expression `ptr + 5` donnera une valeur égale à `ptr + 5 * sizeof(float)`, c'est-à-dire `0x1234 + 20`, soit `0x1248`.

Distance entre pointeurs. Cette opération peut être réalisée par la soustraction de deux pointeurs de même type. Soit deux pointeurs de valeurs a et b et de type t. La distance entre ces pointeurs sera égale à $(a-b)/\texttt{sizeof}(t)$. Par exemple, supposons les pointeurs `ptr1` et `ptr2` de valeurs respectives `0x0124` et `0x0100` et de type `short` (16 bits). La différence `ptr1-ptr2` donnera comme résultat une valeur de type entier égale à `0x0024/sizeof(short)`, c'est-à-dire 18.

2.9 Chaînes de caractères

Il n'y a pas de type spécifique aux chaînes de caractères en C. Une chaîne de caractères est manipulée avec un pointeur vers le premier caractère de la chaîne. La chaîne est terminée par un caractère de code 0 (noté `'\0'`). La Figure 19 donne un exemple de manipulation d'une chaîne de caractères. La fonction `strlen` détermine le nombre de caractères de la chaîne dont l'adresse du premier caractère est passée en argument (argument `s` de type `char *`). La fonction contient une boucle qui détermine l'adresse du caractère terminal (valeur 0). La longueur de la chaîne est obtenue comme la distance entre les pointeurs du début et de la fin de la chaîne (cf. arithmétique des pointeurs en Section 2.8).

Il est important de noter que la chaîne `"Hello World!!!"` exprimée comme litéral dans la fonction `main` à la Figure 19 est constante. Une telle chaîne constante est souvent placée dans un segment spécifique en lecture seule (soit `.text` soit `.rodata`). Une tentative de modification de cette chaîne pourrait causer un

```
1  #include <stdio.h>
2
3  int strlen(char * s)
4  {
5    char * start= s;
6    while (*s != 0)
7      s++;
8    return s-start;
9  }
10
11 int main(int argc, char * argv[])
12 {
13   char * s= "Hello world !!!";
14   printf("La longueur de \"%s\" est %d\n",
15          s, strlen(s));
16   return 0;
17 }
```

FIGURE 19 – Exemple de manipulation d'une chaîne de caractères.

accès mémoire invalide. En revanche, la variable `s` déclarée dans `main` est une variable pointeur qui est placée sur la pile et dont la valeur est initialisée avec l'adresse du premier caractère de la chaîne de caractères [10]. Cette situation est illustrée à la Figure 20.

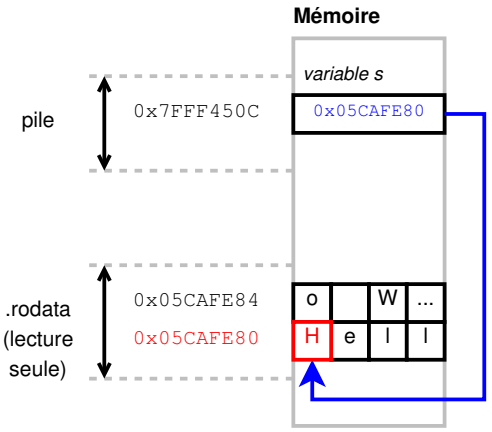

FIGURE 20 – Chaîne constante et pointeur sur la pile référençant cette chaîne.

10. Il serait plus correct d'écrire `const char * s= "Hello World!!!"` car cela définit un pointeur vers des caractères constants. Le compilateur empêchera dans ce cas l'écriture vers le contenu de la chaîne.

Références

[JTC99] JTC 1/SC 22/WG 14. ISO/IEC 9899 :1999 : Programming languages – C. Technical report, International Organization for Standards, 1999.

[Kin08] K. N. King. *C Programming : A Modern Approach*. W. W. Norton and Company, 2nd edition, 2008.

[KR88] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.