C99

Malek.Bengougam@gmail.com

Rappel adresse mémoire

Tout élément (exécutables et dll-s y compris) se trouvant en mémoire est référençable par son emplacement. Le début de cet emplacement est appelé adresse mémoire.
La taille de mémoire maximale accessible sur un OS 32-bit est de 4 GO (2^{32} octets) et de plusieurs milliers de giga-octets sur un OS 64-Bit (2^{64} octets).
Un OS 64-bit nécessite une architecture CPU 64-bit afin de pouvoir stocker une adresse mémoire complète (64 bits, soit 8 octets) dans un registre de manière atomique (càd non interruptible).
Très souvent, comme il n'y a pas suffisamment de registres, les adresses peuvent être stockées en mémoire, sur la pile (*stack*) ou l'espace libre (*free store*, anciennement *heap*).

Adresse d'une variable

Ainsi, selon l'architecture et l'OS la taille d'une adresse mémoire est de 4 ou 8 octets.
En C, toute adresse mémoire peut être stockée dans un type de variable particulier que l'on appelle un **pointeur**.

Toute variable a une adresse en mémoire, et toute variable est accessible via cette adresse.
Un pointeur est une variable standard si ce n'est que son type a toujours une taille (**sizeof**) dépendant de l'OS (32 ou 64 bit).

Une variable est appelée pointeur car son contenu « pointe » vers une autre donnée. Un pointeur étant une variable il est donc possible, par définition, d'obtenir l'adresse du pointeur (et l'adresse d'un pointeur de pointeur etc…).

```
int valeur = 42;
int* pointeur = &valeur;
int** adresseDePointeur = &pointeur;
```

L'opérateur '**&**' permet de récupérer l'adresse de n'importe quelle variable.

L'adresse elle-même est indépendant du type du pointeur, il est donc possible de réinterprété la valeur pointée en modifiant le type du pointeur

```
int* pointeurEntier = &valeur;
float* pointeurReel = (float *)pointeurEntier;
void* pointeurOpaque = (void *)pointeurEntier;
```

Le type de la valeur pointée est important car il détermine le **pas** (ou incrément) du pointeur. En effet, il est possible d'effectuer des opérations arithmétiques sur les pointeurs - essentiellement incrémentation et décrémentation, addition et soustraction.

Incrémenter un pointeur, c'est incrémenter l'adresse –qui est donc stockée dans le pointeur- de la taille du type déclaré. Ceci est transparent et géré automatiquement par le compilateur. Il est important de connaitre ce détail si l'on souhaite accéder manuellement à la mémoire.

```
// prochaine adresse vaut pointeur + sizeof int
int* prochaineAdresse = pointeur + 1;
```

Attention : les règles usuelles relatives à la pré et post incrémentation s'appliquent également aux pointeurs. Dans l'exemple ci-dessous, dans les deux cas la variable nommée « pointeur » est incrémentée, mais les variables « precedenteAdresse » et « nouvelleAdresse » ne pointent pas sur les mêmes adresses.

```
int* precedenteAdresse = pointeur++;
int* nouvelleAdresse = ++pointeur;
```

Tableau

Etant donné qu'il est possible d'accéder à la mémoire de manière séquentielle à partir d'un pointeur, les pointeurs sont donc à la base de l'implémentation des tableaux en C++.

Il existe deux types de tableaux : tableaux alloués statiquement et tableaux dynamiques.

Un tableau alloué de manière statique est une zone mémoire se trouvant sur la pile, donc comme n'importe quelle autre variable locale. Lorsque le tableau alloué statiquement est global ou déclaré avec le mot clé 'static' la zone mémoire est réservée à l'avance et se trouve dans l'exécutable.

```
// 42 entiers réservés sur la pile
int tableauEntier[42];
int *pointeurEntier = tableauEntier;
int cinquiemeElement = tableauEntier[5];
assert(cinquiemeElement == *(pointeur + 5));
```

Ici, tableauEntier est techniquement un pointeur sur entiers, mais du point de vue syntaxique (notamment dans le passage de paramètre aux fonctions) les deux types –pointeur et tableau statique- sont considérés comme différents par le compilateur. Dans cet exemple, le compilateur peut générer des warnings ou erreurs si on essaye d'accéder à tableauEntier[-3] ou tableauEntier[50], ce qui ne sera pas le cas avec pointeurEntier[50]…

Pour lire le contenu du tableau il faut utiliser l'opérateur d'indexation [] ou alternativement, de procéder à un déréférencement c'est-à-dire à lire la valeur pointée par le pointeur, à l'aide de l'opérateur de déréférencement '*'. On peut donc lire le cinquième élément ainsi :

```
int cinquiemeElement = *(pointeur + 5);
```

Structure en mémoire

Comme dans tout langage il est possible de regrouper des données hétérogènes en un bloc contigu. En C il existe plusieurs types de regroupement : les structures ainsi que les unions.

Rapidement, une union est un bloc particulier qui sert à définir une même zone mémoire avec des types différents. Techniquement cela reste une variable ou structure standard sauf que l'on a alors plusieurs alias possible sur le type et l'agencement de la variable.

Pour les structures et les classes il va être question de la façon dont on référence ou pointe l'ensemble ou un élément en particulier.

Le compilateur ne réordonnance pas les données d'une structure, elles sont agencées en mémoire en suivant l'ordre de la déclaration.
En C99, une déclaration **struct** suffit à définir un type, on peut donc connaitre la taille d'une structure ou d'une donnée de la structure, à l'aide du mot clé **sizeof**.

L'adresse mémoire d'une structure est tout simplement l'adresse mémoire de la première donnée déclarée. On peut donc manipuler les structures et classes à l'aide de pointeurs et références comme tout autre type et avec les mêmes règles que vue précédemment.

```
struct Data
{
	unsigned int entierPositif;
	float reel;
	union {
		int unionEntier;
		float unionReel;
	};
	short court;
	bool booleen;
	char caractere;
};

Data tableau[42];
tableau[0].caractere = '0';
Data* ptrData5 = &tableau[5];
ptrData5->caractere = 'A';
Data* ptrData40 = *(tableau + 40);
ptrData40->caractere = '\0';
```

On accède à un élément d'une structure via l'opérateur . (point) lorsqu'on dispose d'un accès direct (variable déclarée localement par exemple) ou lorsque l'on passe par une référence.

Dans le cas d'un accès par pointeur c'est un petit peu plus complexe, il faut comme on l'a vu déréférencer le pointeur et ceci est possible de deux manières différentes. D'une part en utilisant l'opérateur * comme on l'a vu précédemment. Cependant vu que l'opérateur d'accès fait précédence il faut penser à ajouter des parenthèses.
D'autre part en utilisant l'opérateur -> (flèche) qui est un raccourci. Les deux lignes suivantes sont donc équivalentes :

```
ptrData5->caractere = 'A';
(*ptrData5).caractere = 'A';
```

Allocations dynamiques

Allouer les données de manière statique est limitatif car cela oblige à fixer le nombre maximum d'objets que l'on peut utiliser pendant la compilation.
Qui plus est on est limité par la taille de la pile de l'exécutable.

En C on peut utiliser les fonctions malloc() et free() mais attention la valeur de retour de malloc() n'est pas typée.

```
int* pointeurEntier = (int *)malloc(sizeof int);
free(pointeurEntier);
```

Le cas des allocations de tableaux est très peu différent. Ici on alloue 42 entiers:

```
int* pointeurTableauEntier = (int *)malloc(sizeof(int) * 42);
free(pointeurTableauEntier)
```

Notez que l'on peut initialiser les éléments d'un tableau dynamique par défaut mais il n'est pas possible d'appeler d'autres formes de constructeurs, il faut le faire manuellement dans une boucle. Nous verrons par la suite d'autres méthodes issues du C++ 11.

Attention, en C il n'y a pas de *garbage collector* (ramasse-miette) : une allocation avec **malloc** doit forcément être libéré par un free lorsque l'on n'a plus besoin de cette zone mémoire ; autrement c'est le risque d'une fuite mémoire (*memory leak*).

Exercices

1. Créez un pointeur p1 et un pointeur p2 sur la variable k de sorte que l'on puisse modifier la valeur avec le pointeur p1 mais pas avec le pointeur p2.

   ```
   float k = 3.14159f;
   ```

2. Créez un autre pointeur, cette fois-ci pointeur sur **int**, auquel vous affecterez l'adresse de k.

3. Quelle est la valeur de nouvelleAdresse ? On suppose un exécutable 64-bit, donc une taille de pointeur de 8 octets. Quelle serait sa valeur si l'exécutable était compilé en 32-bit ?

   ```
   int* pointeur = 0x12340;
   int* precedenteAdresse = pointeur++;
   int* nouvelleAdresse = ++pointeur;
   ```

4. Soit la chaine de caractère suivante,

   ```
   const char* str = "ceci est un texte";
   ```

   Créez un pointeur str2 de sorte que le code suivant affiche « texte ».

   ```
   printf("%d\n", str2);
   ```

5. La fonction size_t **strlen**(const char*) déclarée dans <string.h> retourne la taille d'une chaine de caractère. Ecrire une fonction similaire à strlen().

   Note : size_t est un type particulier qui représente la taille maximale d'une zone mémoire. Pour rappel, une chaine de caractère en C se termine toujours par la valeur 0 (zéro).