Base V

Programmation séquentielle en C, 2020-2021

Orestis Malaspinas (A401), ISC, HEPIA

2020-10-14

Inspirés des slides de F. Glück

Rappel: représentation des variables en mémoire

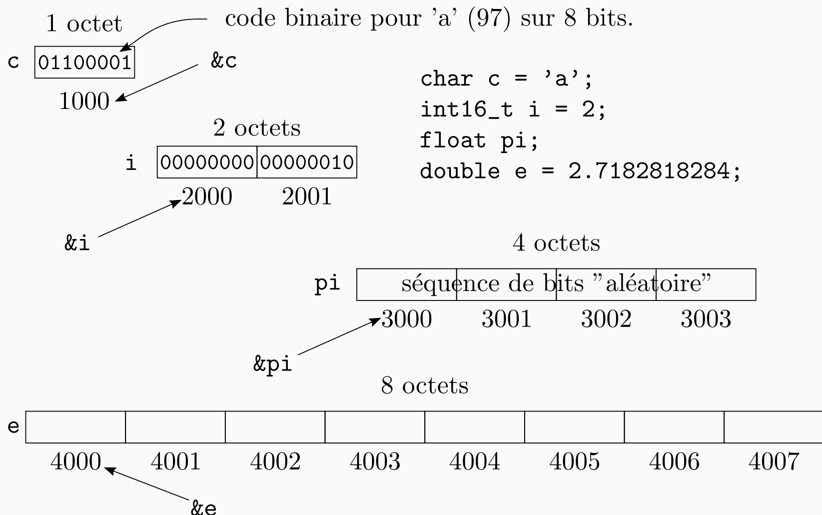


Figure 1 – Les variables en mémoire.

Rappel: Les pointeurs (1/3)

- Un pointeur est une adresse mémoire.

```
type *id;
```

- Pour interpréter le contenu de ce qu'il pointe, il doit être typé.
- Un pointeur n'est rien d'autre qu'un entier (64bit sur x86-64, soit 8 octets).
- Un pointeur peut être **déréférencé**: on accède à la valeur située à l'adresse mémoire sur laquelle il pointe.

```
char *c; // déclaration pointeur de char
*c = 'a'; // assign. 'a' à valeur pointée par c
c = 1000; // on modifie l'adresse pointée par c
char d = *c; // on lit la valeur pointée par c. UB!
```

- `NULL` (ou `0`) est la seule adresse **toujours** invalide.

Rappel: Les pointeurs (2/3)

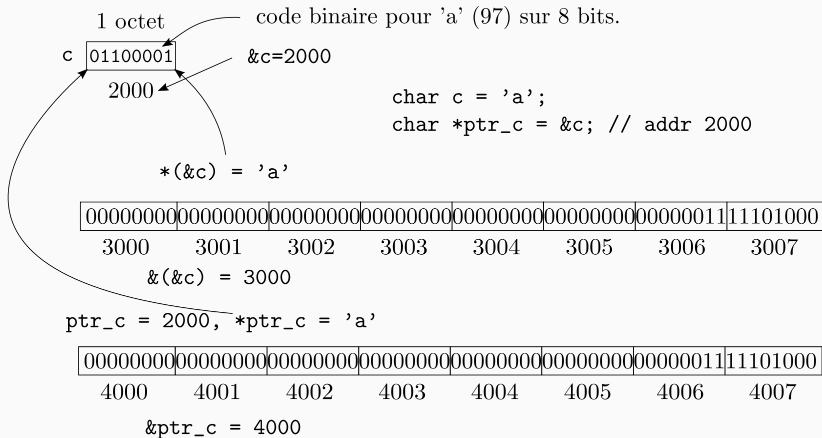


Figure 2 – Les pointeurs, le déréférencement, et la mémoire.

Rappel: Les pointeurs (3/3)

- Permettent d'accéder à une valeur avec une indirection.

```
int a = 2;
int *b = &a;
*b = 7; // on met 7 dans la case pointée par b
        // ici  a == 7 aussi
a = -2; // ici *b == -2 aussi
```

- Permettent d'avoir plusieurs chemins d'accès à une valeur.
- Lire **et** écrire en même temps dans un bout de mémoire devient possible: **danger**.

Pointeurs et `const`

Deux niveaux de constance

- Le mot clé `const` permet de déclarer des valeurs “constantes” qui ne changeront plus en cours d’exécution du programme.
- Mais qu’est-ce que cela veut dire pour les pointeurs?

```
int n = 12;

const int *p = &n; // la valeur *p est const, p non
int const *p = &n; // la valeur *p est const, p non
int *const p = &n; // la valeur p est const, *p non
const int *const p = &n; // la valeur p et *p sont const
```

Exemples

```
int n = 12; int m = 13;

const int *p = &n; // la valeur *p est const, p non
*p = m; // erreur de compilation.
p = &m; // OK
```

La fonction `sizeof()` (1/2)

- La fonction `sizeof()` permet de connaître la taille en octets:
  - d'une valeur,
  - d'un type,
  - d'une variable.
- Soit `int a = 2`, sur l'architecture x86_64 que vaut:
  - `sizeof(a)`?
  - `sizeof(&a)`?
- Soit `char b = 2`,
  - `sizeof(b)`?
  - `sizeof(&b)`?

La fonction `sizeof()` (2/2)

- Réponses:
  - `sizeof(a) == 4`, `int` entier 32 bits.
  - `sizeof(&a) == 8`, une adresse est de 64 bits.
  - `sizeof(b) == 1`, `char` entier 8 bits.
  - `sizeof(&b) == 8`, une adresse est de 64 bits.

Allocation dynamique de mémoire (1/8)

- La fonction `malloc` permet d'allouer dynamiquement (pendant l'exécution du programme) une zone de mémoire contiguë.

```c
#include <stdlib.h>
void *malloc(size_t size);
```

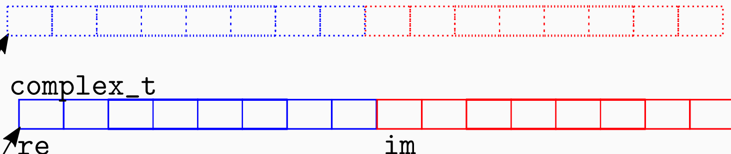
- `size` est la taille de la zone mémoire **en octets**.
- Retourne un pointeur sur la zone mémoire ou `NULL` en cas d'échec: **toujours vérifier** que la valeur retournée est `!= NULL`.

Allocation dynamique de mémoire (2/8)

- On peut allouer un `complex_t`:

```
complex_t *num = malloc(sizeof(complex_t));
num->re = 1.0;
num->im = -1.0;
```

- La zone mémoire **n'est pas** initialisée.
- La mémoire doit être désallouée explicitement $\Rightarrow$ **fuites mémoires**.



Allocation dynamique de mémoire (3/8)

- La fonction `free()` permet de libérer une zone préalablement allouée avec `malloc()`.

```c
#include <stdlib.h>
void free(void *ptr);
```

- Pour chaque `malloc()` doit correspondre exactement un `free()`.
- Si la mémoire n'est pas libérée: **fuite mémoire** (l'ordinateur plante quand il y a plus de mémoire).
- Si la mémoire est **libérée deux** fois: *seg. fault*.
- Pour éviter les mauvaises surprises mettre `ptr` à `NULL`.

Allocation dynamique de mémoire (4/8)

Tableaux dynamiques

- Pour allouer un espace mémoire de 50 entiers:

```
int *p = malloc(50 * sizeof(int));
```

- Cette espace peut alors être utilisé comme un tableau de 50 entiers:

```
for (int i = 0; i < 50; ++i) {
    p[i] = 0;
}
```

Arithmétique de pointeurs

- On peut parcourir la mémoire différemment qu'avec l'indexation

```
int *p = malloc(50 * sizeof(int));
// initialize somehow
double a = p[7];
double b = *(p + 7); // on avance de 7 "double"
p[0] == *p; // le pointeur est le premier élément
```

Allocation dynamique de mémoire (5/8)

Arithmétique de pointeurs

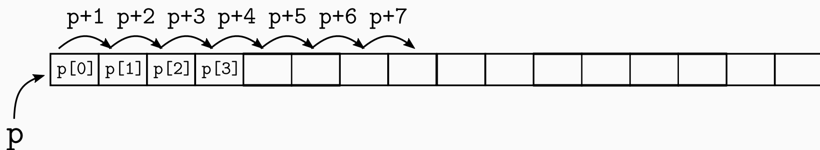


Figure 4 – L'arithmétique des pointeurs.

Allocation dynamique de mémoire (6/8)

Pointeur de pointeur

- Tout comme une valeur a une adresse, un pointeur a lui-même une adresse:

```
int a = 2;
int *b = &a;
int **c = &b;
```

- Chaque $*$ ou & rajoute une indirection.

Allocation dynamique de mémoire (7/8)

Pointeur de pointeur

```
int a = 2;
int *b = &a;
int **c = &b;
...
```

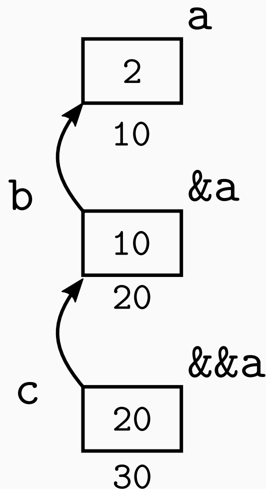


Figure 5 – L'arithmétique des pointeurs.

Allocation dynamique de mémoire (8/8)

- Avec `malloc()`, on peut allouer dynamiquement des tableaux de pointeurs:

```c
int **p = malloc(50 * sizeof(int*));
for (int i = 0; i < 50; ++i) {
    p[i] = malloc(70 * sizeof(int));
}
int a = p[5][8]; // on indexe dans chaque dimension
```

- Ceci est une matrice (un tableau de tableau).

Les *sanitizers*

Il existe différents outils pour détecter les problèmes mémoire:

- Dépassement de capacité de tableaux.
- Utilisation de mémoire non allouée.
- Fuites mémoire.
- ...

Notamment:

- Valgrind.
- Sanitizers.

Ici on utilise les sanitizers (modification de la ligne de compilation):

```
gcc -o main main.c -g -fsanitize=address -fsanitize=leak
```

Attention: Il faut également faire l'édition des liens avec les sanitizers.