Pointeurs

François Trahay

Contents

Espace mémoire d'un processus

- Espace mémoire dans lequel un processus peut stocker des données/du code
- Séparé en plusieurs parties (*segments*), dont:

– *pile* (*stack*): les variables locales et paramètres de fonctions
– *tas* (*heap*): les variables globales
– *segment de code* : le code (binaire) du programme

Adresse mémoire

* On peut faire référence à n'importe quel octet de l'espace mémoire grace à son adresse * Adresse mémoire virtuelle codée sur k bits[1] * donc 2^k octets accessibles (de `00...00` à `11...11`)

- exemple: à l'adresse `0x1001` est stocké l'octet `0x41`
  - peut être vu comme un `char` (le caractère `A`)
  - peut être vu comme une partie d'un `int` (par exemple l'entier `0x11**41**2233`)

[1] k dépend de l'architecture. Sur les processeurs modernes (64 bits), on a $k = 64$.

Rappel: hexadécimal

Les valeurs préfixées par `0x` sont représentées en hexadécimal (en base 16). Ainsi, `0x200d` correspond au nombre qui s'écrit `200D` en base 16, soit le nombre $2 \times 16^3 + 0 \times 16^2 + 0 \times 16^1 + 13 \times 16^0 = 8205$ écrit en base 10.

La notation hexadécimale est couramment utilisée pour représenter des octets car deux chiffres en hexadécimal permettent de coder 256 (soit 2^8) valeurs différentes. On peut donc représenter les 8 bits d'un octet avec deux chiffres hexadécimaux. Par exemple, `0x41` représente l'octet `0100 0001`.

Architecture des processeurs

Les processeurs équipant les ordinateurs modernes sont généralement de type `x86_64`. Pour ces processeurs, les adresses virtuelles sont codées sur 64 bits. Un processus peut donc adresser 2^{64} octets (16 Exaoctets ou 16 x 1024 Pétaoctets) différents.

ARM est une autre architecture de processeur très répandue puisqu'elle équipe la plupart des smartphones. Jusqu'à très récemment, les processeurs ARM fonctionnaient en 32 bits. Un processus pouvait donc accéder à 2^{32} octets (4 Gigaoctets). Les processeurs ARM récents sont maintenant 64 bits, ce qui permet à un processus d'utiliser une plus grande quantité de mémoire.

Adresse d'une variable

- `&var` désigne l'adresse de `var` en mémoire
- affichable avec `%p` dans `printf`:

```
printf("adresse de var: %p\n", &var);
```

affiche:

```
adresse de var: 0x7ffe8d0cbc7f
```

Il est possible de manipuler l'adresse de n'importe quel objet en C, que ce soit une variable, le champ d'une structure, ou une case d'un tableau.

Le programme suivant:

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

struct point{
  float x;
  float y;
  float z;
  int id;
};

int main() {
  char var='A';
  printf("adresse de var: %p\n", &var);

  struct point p = {.x = 2.5, .y = 7.2, .z=0, .id=27};
  printf("adresse de p: %p\n", &p);
  printf("adresse de p.x: %p\n", &p.x);
  printf("adresse de p.y: %p\n", &p.y);
  printf("adresse de p.z: %p\n", &p.z);
  printf("adresse de p.id: %p\n", &p.id);

  char tab[] = "hello";
  printf("adresse de tab[2] = %p\n", &tab[2]);
  return EXIT_SUCCESS;
}
```

peut donner cet affichage:

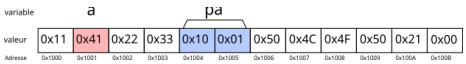
```
adresse de var: 0x7ffe44d7c6df
adresse de p: 0x7ffe44d7c6c0
adresse de p.x: 0x7ffe44d7c6c0
adresse de p.y: 0x7ffe44d7c6c4
adresse de p.z: 0x7ffe44d7c6c8
adresse de p.id: 0x7ffe44d7c6cc
```

```
adresse de tab[2] = 0x7ffe44d7c6b2
```

Pointeur

- Variable dont la valeur est une adresse mémoire
- valeur binaire codée sur k bits (k dépend de l'architecture du processeur)
- déclaration: `type* nom_variable;`
  - `type` désigne le type de la donnée "pointée"
- exemple: `char* pa;` crée un pointeur sur une donnée de type `char`:

```
// pour l'exemple, les adresses sont codees sur 32 bits
char a = 'A'; // a est stocke a l'adresse 0x0000FFFF
              // la valeur de a est 0x41 ('A')
char* pa = &a; // pa est une variable de 32 bits stockee
               // aux adresses 0xFFFB a 0xFFFE
               // la valeur de pa est 0x0000FFFF (l'adresse de a)
```



On peut ensuite manipuler l'adresse de `a` (`0xFFFF`) ou la valeur de `pa` (`0xFFFF`) indifféremment:

```
printf("&a = \%p\n", &a); // affiche 0xFFFF
printf("pa = \%p\n", pa); // affiche 0xFFFF
printf("&pa = \%p\n", &pa); // affiche 0xFFFB, soit l'adresse de pa
```

Un pointeur étant une variable comme les autres, on peut donc stocker son adresse dans un pointeur. Par exemple:

```
char a = 'A'; // a est stockee a l'adresse 0xFFFF et contient 0x41 ('A' ou 65)
char* pa = &a; // pa est stockee a l'adresse 0xFFFB et contient 0xFFFF (l'adresse de a)
char** ppa = &pa; // ppa est stockee a l'adresse 0xFFF7 et contient 0xFFFB (l'adresse de pa)
```

Conseil de vieux baroudeur

Quand vous déclarez un pointeur, initialisez-le immédiatement, soit avec l'adresse d'une variable, soit avec la valeur `NULL` (définie dans `stdlib.h`) qui est la valeur pointant sur "rien". Dit autrement, ne laissez **jamais** une variable pointeur avec un contenu non initialisé.

Arithmétique de pointeur

Les opérateurs `+`, `-`, `++`, et `--` sont utilisables sur des pointeurs, mais avec précaution.

Incrémenter un pointeur sur `type` aura pour effet d'ajouter `sizeof(type)` à la valeur du pointeur. Par exemple:

```
char* pa = &a;  // pa vaut 0xFFFF
pa--;           // enleve sizeof(char) (c'est a dire 1) a pa
                // donc pa vaut 0xFFFE

char**ppa = &pa // ppa vaut 0xFFFB
ppa--;          // enleve sizeof(char*) (c'est a dire 4) a ppa
                // donc ppa vaut 0xFFF7

ppa = ppa - 2;  // soustrait 2*sizeof(char*) (donc 8) a ppa
                // ppa vaut 0xFFEF
```

Exemple complet

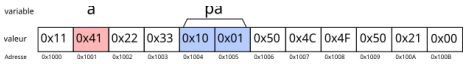
Sur https://codecast.france-ioi.org/, vous pouvez visualiser le contenu de la mémoire d'un programme. Pour cela, saisissez le code source du programme, cliquez sur "compiler", puis exécutez le programme pas à pas en cliquant sur "next expression". {‘A} chaque instant, le contenu de la mémoire est représenté en bas de la page. Essayez avec ce programme:

```c
#include <stdio.h>
int main() {
  //! showMemory(cursors=[a, pa, ppa], start=65528)
  char a = 'A';
  char* pa= &a;
  char** ppa = &pa;

  printf("a = %d, &a=%p\n", a, &a);
  printf("pa = %p, &pa=%p\n", pa, &pa);
  printf("ppa = %p, &ppa=%p\n", ppa, &ppa);

  pa--; // enleve sizeof(char)=1 a pa
  ppa--; // enleve sizeof(char*) a ppa

  printf("a = %d, &a=%p\n", a, &a);
  printf("pa = %p, &pa=%p\n", pa, &pa);
  printf("ppa = %p, &ppa=%p\n", ppa, &ppa);

  ppa = ppa - 2; // enleve 2*sizeof(char*) a ppa
  printf("ppa = %p, &ppa=%p\n", ppa, &ppa);

  return 0;
}
```

Déréférencement

- Permet de consulter la valeur stockée à l'emplacement désigné par un pointeur
  - `* ptr`
  - exemple:

```c
char a = 'A'; // valeur 0x41 (cf. codage ASCII)
char* pa = &a;
printf("pa = %p\n", pa); // affiche "pa = 0xFFFF"
printf("*pa = %c\n", *pa); // affiche "*pa = A"
*pa = 'B';   // modifie l'emplacement memoire 0xFFFF
             // (donc change la valeur de a)
printf("a = %c\n", a); // affiche "a = B"
```



A partir d'un pointeur, on peut donc afficher 3 valeurs différentes:

- `printf("pa = %p\n", pa);` – affiche la valeur du pointeur (ici, l'adresse `0x1001`)
- `printf("*pa = %c\n", *pa);` – affiche la valeur "pointée" par `pa` (ici, la valeur de `a` : `'A'`)
- `printf("&pa = %c\n", &pa);` – affiche l'adresse du pointeur (ici, l'adresse `0x1004`)

Déréférencement dans une structure

Lorsqu'un pointeur `ptr` contient l'adresse d'une structure, l'accès au champ `c` de la structure peut se faire en déréférençant le pointeur, puis en accédant au champ : `(*ptr).c`

Cela devient plus compliqué lorsque la structure contient un pointeur (`p1`) vers une structure qui contient un pointeur (`p2`) sur une structure. La syntaxe devient rapidement indigeste: `(*(*(*ptr).p1).p2).c`

Pour éviter cette notation complexe, on peut utiliser l'opérateur `->` qui combine le déréférencement du pointeur et l'accès à un champ. Ainsi, `ptr->c` est équivalent à `(*ptr).c`. On peut donc remplacer la notation `(*(*(*ptr).p1).p2).c` par `ptr->p1->p2->c`.

Tableaux et pointeurs (1/3)

Si un tableau est **un argument de fonction**

- la déclaration est **remplacée** par celle d'un pointeur
  - `void f(int x[])` <=> `void f(int* x)`
- un accès effectue un décalage + déréférencement

  - `tab[i]` réécrit en `*(tab + i)`
  - exemple: si `tab = 0x1000` et `i=5`
  - `tab[i]` calcule `0x1000 + (5*sizeof(int)) = 0x1000 + 0x14 = 0x1014`
- `sizeof(tab)` donne la taille d'un pointeur
- Remarque : `&tab` donne l'adresse de `int[] tab`, donc `&tab != tab`

Tableaux et pointeurs (2/3)

Si un tableau est **une variable locale ou globale**

- le tableau **n'est pas remplacé par un pointeur**
- le tableau doit avoir une taille connue
  - `int tab[3]; // alloue 3 int`
    - `tab` est le nom de cet espace mémoire
  - `int tab[] = { 1, 2, 3 }; // idem + initialisation`
  - `int tab[]; // interdit !`
- `sizeof(tab)` renvoie la taille du tableau

Tableaux et pointeurs (3/3)

Si un tableau est **une variable locale ou globale** (suite)

- `&tab` donne l'adresse du tableau
  - Remarque : `&tab == &tab[0]` car `tab` et `tab[0]` désignent les mêmes emplacements mémoires
- `tab` est **implicitement transtypé vers son pointeur** au besoin
- Exemple :
  - `int* tab2 = tab;` réécrit en `int* tab2 = &tab`
  - `if(tab == &tab)` récrit en `if(&tab == &tab)`
  - `f(tab)` réécrit en `f(&tab)`
  - `printf("%p %p\n", tab, &tab);` réécrit en `printf("%p %p\n", &tab, &tab);`
  - `tab[i]` réécrit en `(&tab)[i]` puis en `*(&tab + i)`
  - `*(tab + i)` réécrit en `*(&tab + i)`

Passage par référence

Rappel:

- *Passage par référence: une référence vers l'argument de l'appelant est donné à l'appelé* (cf. CI2)
- Cette référence est un pointeur

```c
void f(int* px) {
  *px = 666;          // la variable pointee par px est modifiee
}

int main() {
  int x = 42;
  f(&x);                    // l'adresse de x est donnee à f
            //  => le x de main est modifié par f
  printf("x = \%d\n", x); // la nouvelle valeur de x : 666
  return EXIT_SUCCESS;
}
```

Allocation dynamique de mémoire

- `void* malloc(size_t nb_bytes);`
  - Alloue `nb_bytes` octets et retourne un pointeur sur la zone allouée
- usage:
  - `char* str = malloc(sizeof(char)* 128);`
  - renvoie `NULL` en cas d'erreur (par ex: plus assez de mémoire)

Attention ! Risque de "fuite mémoire" si la mémoire allouée n'est jamais libérée

Signification de `void*`

Le `void*` renvoyé par `malloc` signifie que la fonction retourne un pointeur vers n'importe quel type de donnée. Ce pointeur (qui est donc une adresse) vers `void` peut être converti en pointeur (une adresse) vers `int` ou tout autre type.

Recommandation

Vérifiez systématiquement si `malloc` vous a renvoyé `NULL` et, si c'est le cas, arrêtez votre programme. Une manière simple et lisible de faire cela est d'utiliser la macro `assert` (définie dans `assert.h`) comme dans l'exemple suivant :

```c
char* str = malloc(sizeof(char)* 128);
assert(str);
```

Fuites mémoire

Lorsque l'on déclare une variable (un `int`, un tableau, une structure, ou toute autre variable) depuis une fonction `foo`, l'espace mémoire de cette variable est réservé sur la pile. Lorsque l'on sort de `foo`, la pile est "nettoyée" et l'espace réservé pour les variables locales est libéré.

Lorsque l'on alloue de la mémoire avec `malloc` depuis une fonction `foo`, la mémoire est allouée sur le *tas*. Lorsque l'on sort de la fonction `foo`, l'espace mémoire réservé reste accessible. Si on "perd" l'emplacement de cette zone

mémoire, elle devient donc inaccessible, mais reste réservée: c'est une *fuite mémoire*.

Si la fuite mémoire fait "perdre" quelques octets à chaque appel de la fonction `foo`, la mémoire de la machine risque, à terme, d'être remplie de zones inutilisées. Le système d'exploitation n'ayant plus assez de mémoire pour exécuter des processus devra donc en tuer quelques uns pour libérer de la mémoire.

Libération de mémoire

- `void free(void* ptr);`
- Libère la zone allouée par `malloc` est situé à l'adresse `ptr`
- Attention à ne pas libérer plusieurs fois la même zone!

A chaque fois que vous faites `free` sur un pointeur, pensez à remettre ensuite ce pointeur à `NULL` (pour être sûr que vous n'avez pas un pointeur qui pointe sur une zone de mémoire libérée). Dit autrement, tout "`free(ptr);`" doit être suivi d'un "`ptr = NULL;`".

Notions clés

- L'espace mémoire d'un processus
- Les pointeurs
  - Adresse mémoire d'une variable (`&var`)
  - Pointeur sur `type` : `type* ptr;`
  - Arithmétique de pointeurs (`ptr++`)
  - Adresse nulle: `NULL`
  - Déréférencement d'un pointeur:
    - types simples: `*ptr`
    - structures: `ptr->champ`
    - tableaux: `ptr[i]`
  - Passage de paramètre par référence
- Allocation dynamique de mémoire * allocation: `int* ptr = malloc(sizeof(int)*5);` * désallocation: `free(ptr);`