Introduction à la Programmation des Algorithmes

3.3. Langage C – Pointeurs et mémoire

François Fleuret
https://fleuret.org/11x001/

Comme nous l'avons vu, les informations que manipule un ordinateur sont stockée dans une **mémoire vive**, qui est une suite d'octets, chacun avec une adresse, et pouvant stocker une valeur entre 0 et 255 inclus.

En particulier, une variable est une suite consécutive d'octets qui représentent une valeur.

Contrairement à la plupart des autres langages de programmation, le langage C permet de manipuler directement la mémoire.

Le concept central pour cela est celui de **pointeur**, qui est une variable contenant une adresse en mémoire.

Deux nouveaux opérateurs nous permettent de manipuler des pointeurs:

- l'opérateur `*` permet de déclarer une variable de type pointeur, et également de manipuler la variable qui est pointée,
- l'opérateur `&` permet d'obtenir l'adresse en mémoire d'une variable.


```
1   int a = 1;
2   printf("a=%d\n", a);
3
4   int *q;
5   q = &a;
6
7   printf("*q=%d\n", *q);
8
9   *q = 2;
10  printf("a=%d\n", a);
```

La ligne 4 déclare une variable q de type pointeur vers `int` (ou plus simplement de type `int *`). La ligne 5 copie dans la variable q l'adresse de la variable a.

La ligne 7 accède à la valeur de la variable pointée par q, donc a, et la ligne 9 modifie la variable pointée par q.

La déclaration d'un pointeur se fait donc avec

```
type *identifiant;
```

l'accès à une variable pointée avec

```
*identifiant
```

et l'accès à l'adresse d'une variable avec

```
&identifiant
```

L'adresse d'une variable est en réalité celle du premier octet qui la compose.

Un pointeur est concrètement un entier de taille suffisante pour représenter une adresse arbitraire. Un entier sur 4 octets permet de représenter au maximum $2^{32} = 4'294'967'296$ adresses, donc un maximum de 4Gb.

Les ordinateurs actuels représentent les adresses sur 8 octets ce qui lève toute contrainte sur l'espace mémoire adressable.

```
1  int *q;
2  printf("%d\n", sizeof(q));
```

affiche

```
8
```

La spécification de format %p de `printf` permet d'afficher des pointeurs (en base 16).

```
1  int a, b;
2  printf("%p %p\n", &a, &b);
```

affiche

```
0x7fffbf56d658 0x7fffbf56d65c
```


Une variable de type tableau est directement convertible en pointeur, sans l'opérateur &.

```
1  int u[10];
2  int *p = u;
3  printf("%p\n", u);
4  printf("%p\n", p);
5  printf("%p\n", &u[0]);
```

affiche

```
0x7ffee59e7a20
0x7ffee59e7a20
0x7ffee59e7a20
```

Il est possible de faire des opérations arithmétiques qui combinent pointeurs et entiers, ce qui permet par exemple de manipuler le contenu d'un tableau.

Ajouter ou soustraire un entier `n` à un pointeur `un_type *p` ajoute ou soustrait `n` fois la taille de `un_type` pour que cela corresponde à un déplacement de `n` éléments et non pas de `n` octets.

L'opérateur `[]` que nous avons appliqué à des tableaux peut être appliqué à des pointeurs. Dans ce cas, l'expression

```
p[n]
```

est exactement équivalente à

```
*(p + n)
```



```
1  int u[10];
2  for(int k = 0; k < 10; k++) u[k] = k;
3  *(u + 3) = 13;
4  int n = 3;
5  *(u + n * 2 + 1) = 17;
6  for(int k = 0; k < 10; k++) printf("%d %d\n", k, u[k]);
```

affiche

```
0 0
1 1
2 2
3 13
4 4
5 5
6 6
7 17
8 8
9 9
```

La différence de deux pointeurs de même type retourne de manière similaire un entier égal au nombre d'éléments qui les séparent.

```
1  float tab[1024];
2  float *p, *q;
3  p = tab;
4  q = &(tab[277]);
5  printf("%d\n", q-p);
```

affiche

277

La manipulation de pointeurs doit être faite de manière très rigoureuse, car comme pour les accès hors tableau, l'accès à une zone mémoire incorrecte peut entraîner des erreurs très imprédictibles et graves. Cela peut être la corruption de valeurs ou le crash au niveau du système.

Passage d'arguments par référence

Comme nous l'avons vu, lors de l'appel à une fonction, les valeurs passées en arguments sont **copiées** dans des variables locales créées au moment de l'appel.

Dans le cas d'un type composé comme un tableau ou une structure, cette copie peut prendre du temps.

Si la valeur à passer en argument est dans une variable, il est plus efficace de passer à la fonction un pointeur vers cette variable, qui est un type de petite taille fixe, plutôt que de faire une copie.

On parle alors de passer les arguments par **référence** plutôt que par **valeur**.

Les structures sont très souvent manipulées de cette façon, et le C offre l'opérateur `->` pour accéder aux champs d'une structure à partir de son adresse.

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

typedef struct {
  float x, y, z;
} vecteur3d;

float norme_l2(vecteur3d v) {
  return sqrt(v.x * v.x + v.y * v.y + v.z * v.z);
}

int main(void) {
  vecteur3d u = { -1, -1, 0 };
  float l = norme_l2(u);
  u.x /= l;
  u.y /= l;
  u.z /= l;
  printf("%f\n", norme_l2(u));
  return 0;
}
```

affiche

```
1.000000
```



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

typedef struct {
  float x, y, z;
} vecteur3d;

float norme_l2(vecteur3d *v) {
  return sqrt(v->x * v->x + v->y * v->y + v->z * v->z);
}

int main(void) {
  vecteur3d u = { -1, -1, 0 };
  float l = norme_l2(&u);
  u.x /= l;
  u.y /= l;
  u.z /= l;
  printf("%f\n", norme_l2(&u));
  return 0;
}
```

affiche

```
1.000000
```

Passer un pointeur sur une variable permet également de la modifier si besoin est.

```
1  void trie(float *a, float *b) {
2    if(*a > *b) {
3      float c = *a;
4      *a = *b;
5      *b = c;
6    }
7  }
8
9  int main(void) {
10   float x = 1.2, y = 1.1;
11   printf("%f %f\n", x, y);
12   trie(&x, &y);
13   printf("%f %f\n", x, y);
14   return 0;
15 }
```

affiche

```
1.200000 1.100000
1.100000 1.200000
```



```
1  float norme_l2(vecteur3d *v) {
2    return sqrt(v->x * v->x + v->y * v->y + v->z * v->z);
3  }
4
5  void normalise(vecteur3d *v) {
6    float l = norme_l2(v);
7    v->x /= l;
8    v->y /= l;
9    v->z /= l;
10 }
```

Utiliser un pointeur sur une variable locale en dehors de la portée de cette dernière n'a aucun sens puisqu'elle n'existe plus.

```
1  int *ne_faites_jamais_ca(int n) {
2    int k[n];
3    for(int i = 0; i < n; i++) k[i] = i;
4    return k;
5  }
```


Allocation dynamique de la mémoire

Il arrive très fréquemment que l'on veuille créer un tableau dont le nombre d'éléments n'est pas connu à l'avance et dépend du contexte dans lequel le programme est exécuté.

Un texte, une image, un échantillon sonore peuvent être de tailles variables et il est nécessaire pour les manipuler de créer des tableaux dynamiquement.

Dans certaines situations, on peut déclarer une variable locale de type tableau et de taille qui dépend de l'exécution mais:

- la taille mémoire disponible sur la pile est limitée, et
- on peut avoir besoin que ce tableau continue à exister en dehors de la portée où il a été créé (par exemple si l'on veut qu'une fonction le crée).

La solution est de créer explicitement une variable dynamiquement en réservant un espace mémoire pour cela.

L'essentiel de la mémoire disponible constitue le **tas** (*heap*), et c'est dans cette zone que les programmes placent la quasi-totalité des variables créées dynamiquement.

On a donc les **variables locales** qui sont sur la pile et les **variables dynamiques** sur le tas.

Les mécanismes de réservations de sous-parties du tas en C sont très primitifs et reposent sur deux fonctions de la librairies `stdlib.h`:

- `void *malloc(size_t size);` pour allouer de la mémoire (*"memory allocate"*), et
- `void free(void *ptr);` pour libérer de la mémoire.

Le type `void *` peut être copié dans n'importe quel type pointeur sans erreur du compilateur.

La fonction `malloc` renvoie `0` si l'espace demandé n'est pas disponible, et `free` ne fait rien si l'adresse qui lui est passée en argument est nulle.


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

int main(void) {
  int *t = malloc(sizeof(int));
  *t = 5;
  printf("%d\n", *t);
  free(t);
  return 0;
}
```

affiche

```
5
```

On peut de même allouer dynamiquement un tableau

```
1  int n = 25;
2  int *tab = malloc(sizeof(int) * n);
3  for(int k = 0; k < n; k++) tab[k] = 0;
4  free(tab);
```


Les fonctions `malloc` et `free` se limitent à réserver ou libérer de la mémoire.

En particulier:

- `malloc` ne met pas à zéro la zone mémoire réservée,
- `free` ne met pas à zéro la zone mémoire libérée,
- `free` ne change évidemment pas le pointeur qui lui est passé en argument.

Les erreurs de programmation dues aux pointeurs sont graves, imprédictibles, et difficiles à trouver.

Bien qu'un tableau puisse être converti en pointeur (c'est alors l'adresse du premier élément) et que l'opérateur [] puisse être appliqué à un tableau ou à un pointeur, ces deux types sont différents. En particulier **un pointeur ne fournit aucune information sur la taille de la zone allouée.**

```
1  int blah[100];
2  int *blih = malloc(sizeof(int) * 100);
3  printf("%d %d\n", sizeof(blah), sizeof(blih));
4  free(blih);
```

affiche

```
400 8
```



```
1  int nb_nb_premiers(int max) {
2    int *est_premier = malloc(sizeof(int) * max);
3    int nb = 0;
4
5    for(int n = 0; n < max; n++)
6      est_premier[n] = (n >= 2);
7
8    for(int n = 0; n < max; n++)
9      if(est_premier[n]) {
10       nb++;
11       for(int k = 2 * n; k < max; k += n)
12         est_premier[k] = 0;
13     }
14
15   free(est_premier);
16
17   return nb;
18 }
```

Si on avait utilisé une variable locale pour `est_premier` au lieu d'une allocation dynamique ligne 2, le programme crasherait pour `max` > `2M`.

On appelle **fuite de mémoire** une erreur de programmation qui fait que le programme utilise de plus en plus de mémoire.

C'est généralement parce qu'un `free` manque et qu'un tableau qui n'est plus utile n'est pas libéré.