Le C — Pointeurs et allocation dynamique https://tinyurl.com/35wfete3

Nathanael Demacon

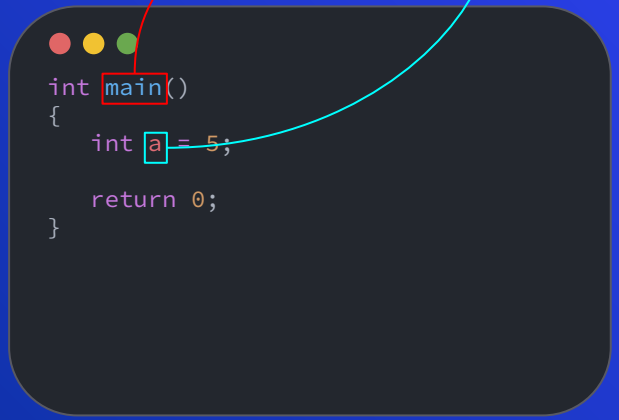
Pointeurs

Un pointeur est une référence d'une autre variable, ou plus précisément un emplacement de la mémoire.

On peut voir la mémoire comme un gigantesque tableau et un pointeur comme un index.

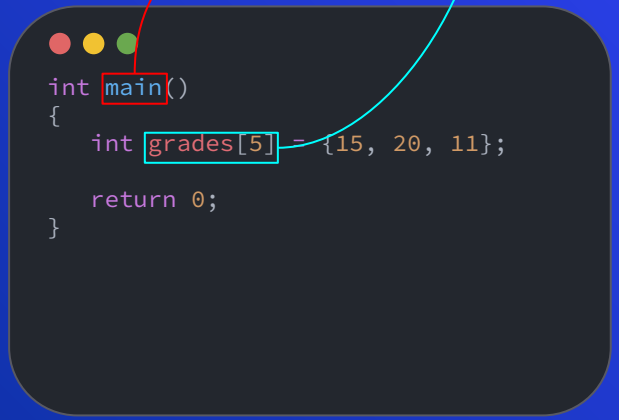
Représentation

0x0	[...]	0x401126	[...]	0x7ffccf55b13c	[...]	END
NULL		main		5		



Représentation

0x0	[...]	0x401126	[...]	0x7ffccf55b13c	0x7ffccf55b140	0x7ffccf55b144	0x7ffccf55b148	0x7ffccf55b14c	[...]	END
NULL		main		15	20	11	?	?		



Opérateurs

- &a (référencement - récupère l'adresse de la variable)
- *a (déréférencement - pointe vers la valeur à l'emplacement du pointeur)

Attention : l'opérateur "&" ne fonctionne que sur les variables et les fonctions. Il ne fonctionnera pas sur les expressions.

```c
int main()
{
   int a = 0;

   int *b = &a;       // ok
   int *c = &5;       // error
   int *d = &(a + a); // error

   return 0;
}
```

Opérateur

Avec les opérateurs “&” et “*” on peut donc passer de pointeur à valeur très facilement.

```c
int main()
{
    int a = 0;

    int *pointer_to_a = &a;
    int value_of_a = *pointer_to_a;

    return 0;
}
```

On peut modifier la valeur d'un pointeur lors de sa déréférence.

```c
int main()
{
    int a = 0;

    int *pointer_to_a = &a;
    *pointer_to_a = 5;

    printf("%d\n", a); // 5

    return 0;
}
```

Type

Tout comme les tableaux, les pointeurs ont un type particulier. Ajouter une étoile à un type permet de signifier que c'est un pointeur.

```c
int main()
{
    type *my_pointer;

    return 0;
}
```

Adresses
Affichage

Un pointeur contient l'adresse d'un emplacement dans la mémoire. On peut l'afficher avec `printf` en utilisant le format "%p".

```c
#include <stdio.h>

int main()
{
    int a;
    printf("%p", &a); // 0x7ff7b0a977f8

    return 0;
}
```

Une adresse sera toujours un entier de la même taille que le processeur (par exemple 64 bits sur un processeur 64 bits).

Adresses
Arithmétique

Étant donné qu'une adresse est un entier, on peut lui appliquer des opérations arithmétiques comme des additions et des soustractions.

```c
#include <stdio.h>

int main()
{
   int a = 0;

   int *pointer_to_a = &a;
   int *next = pointer_to_a + 1;

   printf("%p\n", pointer_to_a); // 0x7ff7b6df17f4
   printf("%p\n", next);         // 0x7ff7b6df17f8

   return 0;
}
```

Attention : lorsque l'on fait de l'arithmétique sur les pointeurs, il faut être sûr de pouvoir accéder à l'emplacement mémoire ciblé lors d'un déréférencement sous risque de faire crasher le programme

Adresses
Arithmétique

On remarque que le pointeur s'est décalé de 4 alors qu'on a juste fait + 1.

```c
#include <stdio.h>

int main()
{
   int a = 0;

   int *pointer_to_a = &a;
   int *next = pointer_to_a + 1;

   printf("%p\n", pointer_to_a); // 0x7ff7b6df17f4
   printf("%p\n", next);         // 0x7ff7b6df17f8

   return 0;
}
```

Faire "`+ 1`" sur un pointeur correspond à faire "`+ (1 * sizeof(type))`" sur l'adresse.

Dans l'exemple c'est donc "`+ (1 * 4)`" qui s'est exécuté car "a" est un entier de 4 octets, on passe donc à l'entier suivant.

Tableaux et pointeurs

Les tableaux sont très similaires aux pointeurs, on peut les utiliser comme tel.

```c
#include <stdio.h>

int main()
{
   int grades[5] = {15, 20, 11};

   printf("%p\n", grades);     // 0x7ff7b7c5e7e0
   printf("%p\n", grades + 1); // 0x7ff7b7c5e7e4

   return 0;
}
```

Tableaux et pointeurs

On peut transformer un tableau en pointeur mais pas l'inverse.

```c
#include <stdio.h>

int main()
{
   int grades[5] = {15, 20, 11};
   int *p = grades;

   printf("%p\n", p);      // 0x7ff7b7c5e7e0
   printf("%p\n", p + 1); // 0x7ff7b7c5e7e4

   return 0;
}
```

Tableaux et pointeurs

Pour accéder aux éléments d'un pointeur autre que le premier, on utilise la syntaxe "`*(ptr + n)`" où n est l'index de l'élément désiré du tableau.

Astuce : en C, la syntaxe "`*(ptr + n)`" a le même fonctionnement que "`ptr[n]`".

```c
#include <stdio.h>

int main()
{
   int grades[5] = {15, 20, 11};
   int *p = grades;

   printf("%d\n", *p);       // 15
   printf("%d\n", *(p + 0)); // 15
   printf("%d\n", *(p + 1)); // 20
   printf("%d\n", *(p + 2)); // 11

   return 0;
}
```

```c
#include <stdio.h>

int main()
{
   int grades[5] = {15, 20, 11};
   int *p = grades;

   printf("%d\n", grades[1] == p[1]); // 1
   printf("%d\n",  *(p + 1) == p[1]);  // 1

   return 0;
}
```

En entrée de fonction

Dans un cours précédent, nous avions dit que les paramètres des fonctions ne pouvaient pas être modifiés en dehors des tableaux.

```c
#include <stdio.h>

void increment(int x)
{
   x += 1;
}

int main()
{
   int a = 0;
   increment(a);

   printf("%d", a); // 0

   return 0;
}
```

Comme les tableaux, on peut aussi modifier le contenu d'un pointeur dans une autre fonction. Pour que notre fonction d'incrémentation fonctionne nous pouvons donc utiliser les pointeurs :

```c
#include <stdio.h>

void increment(int *x)
{
   *x += 1;
}

int main()
{
   int a = 0;
   increment(&a);

   printf("%d", a); // 1

   return 0;
}
```

En sortie de fonction

Attention à ne JAMAIS renvoyer en retour de fonction un pointeur venant d'une variable locale. Seul la mémoire allouée manuellement ou les arguments doivent être retournés.

```c
#include <stdlib.h>

int *get_pointer_bad()
{
   int a = 15;
   return &a; // NOOOOO
}

int *get_pointer_good(int *ptr)
{
   return ptr; // ok
}

int *get_pointer_good2()
{
   int *a = malloc(sizeof(int));
   return a; // ok
}
```

Allocation dynamique

L’allocation dynamique est le principe d’allouer manuellement de la mémoire en fonction des besoins du programme.

Bases

Pour utiliser les fonctionnalités de gestion de la mémoire il faut inclure la bibliothèque “stdlib.h”.

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

Bases

Une allocation simple se fait avec la fonction “malloc” qui prend en paramètre le nombre d’octets à allouer. Si nous voulons allouer 5 entiers, nous pouvons utiliser sizeof pour obtenir la taille d’un entier et le multiplier par 5.

```c
#include <stdlib.h>

int main()
{
   int *grades = malloc(sizeof(int) * 5);

   return 0;
}
```

Bases

Une fois que l'allocation n'est plus nécessaire, il faut libérer la mémoire allouée grâce à la fonction "`free`". Si cette étape est ignorée, le programme pourrait remplir toute la mémoire de l'ordinateur.

```c
#include <stdlib.h>

int main()
{
   int *grades = malloc(sizeof(int) * 5);

   *grades = 15;
   // grades[1] = 20;
   *(grades + 1) = 20;
   *(grades + 2) = 11;

   printf("%d\n", *grades); // 15

   free(grades);

   return 0;
}
```

Attention : seuls les pointeurs que vous avez alloué manuellement peuvent être passés à la fonction "`free`".

Bases

Par défaut toutes les valeurs d'un pointeur ont des valeurs inconnues (comme pour les tableaux). On peut utiliser la fonction "calloc" pour initialiser à 0 tous les emplacements alloués.

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

int main()
{
   int *grades = calloc(5, sizeof(int));

   for (int i = 0; i < 5; i++)
   {
       printf("Grade %d: %d\n", i, grades[i]);
   }

   free(grades);

   return 0;
}
```

Sortie

```
Grade 0: 0
Grade 1: 0
Grade 2: 0
Grade 3: 0
Grade 4: 0
```

Pointeur “NULL”

Un pointeur qui n’a aucune valeur doit être assigné à “NULL”, une constante présente dans stdlib.h. “NULL” est égal au pointeur 0, converti au type `void *`.

```c
#include <stdlib.h>

int main()
{
    int *p = NULL;

    return 0;
}
```

On peut comparer un pointeur à “NULL” dans une condition.

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

int main()
{
    int *p = NULL;

    if (p == NULL)
    {
        printf("p is NULL\n");
    }

    return 0;
}
```

Tableaux
Le problème

Pour créer un tableau jusque là, nous devions fournir la taille du tableau à sa déclaration. C'est problématique lors de tableaux à taille inconnue.

Si i > 4 alors le programme va planter

```c
#include <stdlib.h>

int main()
{
    int grades[5] = {0};

    int i = 0;
    while (1)
    {
        int grade;
        printf("Grade %d: ", i + 1);
        scanf("%d", &grade);

        if (grade == -1)
        {
            break;
        }

        grades[i] = grade;
        i++;
    }

    return 0;
}
```

Tableaux
La solution

Une solution consiste à agrandir le tableau dès qu'il n'y a plus d'espace.

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

int *new_array(int size)
{
    return malloc(size * sizeof(int));
}

void copy_array(int *dest, int *src, int size)
{
    for (int i = 0; i < size; i++)
    {
        *(dest + i) = *(src + i);
    }
}
```

```c
int main()
{
    int capacity = 16;
    int *grades = new_array(capacity);

    int i = 0;
    while (1)
    {
        int grade;
        printf("Grade %d: ", i + 1);
        scanf("%d", &grade);

        if (grade == -1)
        {
            break;
        }

        if (i == capacity)
        {
            capacity *= 2;

            int *tmp = new_array(capacity);
            copy_array(tmp, grades, capacity);
            free(grades);
            grades = tmp;
        }

        grades[i] = grade;
        i++;
    }

    free(grades);

    return 0;
}
```

Tableaux
La solution++

Encore mieux : nous pouvons utiliser la fonction “realloc” qui s’occupe du redimensionnement et de la copie des données du pointeur en plus de quelques optimisations.

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

int main()
{
   int capacity = 16;
   int *grades = malloc(sizeof(int) * capacity);

   int i = 0;
   while (1)
   {
       int grade;
       printf("Grade %d: ", i + 1);
       scanf("%d", &grade);

       if (grade == -1)
       {
           break;
       }

       if (i == capacity)
       {
           capacity *= 2;
           grades = realloc(grades, sizeof(int) * capacity);
       }

       grades[i] = grade;
       i++;
   }

   free(grades);

   return 0;
}
```