

Introduction au C

September 26, 2021

1 Cadre de ce TP

Dans ce TP on va faire du C à l'intérieur d'un noyau interactif permettant d'écrire du C++. 98% de la syntaxe du C est directement utilisable en C++, donc ça ne devrait pas changer beaucoup de choses pour vous.

Normalement un programme en C est composé d'un fichier `.c` (le plus souvent plusieurs qui sont réunis) contenant une fonction `main` qui est le point d'entrée du programme.

Voici un exemple de contenu d'un tel fichier :

```
// déclaration de fonctions d'impression pour pouvoir les utiliser
#include <stdio.h>

int x = 3; // une variable globale entière x pour tout le programme

// déclaration d'une fonction qui prend deux entiers en entrée et renvoie un entier
int ajoute(int a, int b)
{
    return a+b;
}

// la fonction principale

int main()
{
    int a = 2; // déclaration de variables locales
    int b; // non initialisée ici

    b = 3; // nécessite que b soit déclarée.

    printf("La somme de %d + %d = %d\n",
           b, x, ajoute(b, x)); // la fonction principale d'affichage, véritable couteau-suisse
                               // ici %d signifie un entier qui figure dans la liste juste après

    // On retrouve des boucles for
    a = 0;
    for (int i = 0; i < 10; i++) // mais on doit gérer notre compteur nous même
    {
```

```

        a = a + i;
    }
    printf("La somme des 9 premiers entiers vaut %d\n", a);

    return 0; // pour main signifie que tout va bien
}

```

Ici, avec l'interprète, on va pouvoir simuler l'exécution de ce programme mais en séparant les différentes déclarations sur plusieurs cellules et en remplaçant la fonction main par une cellule ne contenant que les instructions.

Évaluez la suite d'instructions suivante pour vous en convaincre.

```

[7]: // déclaration de fonctions d'impression pour pouvoir les utiliser
#include <stdio.h>

```

```

[2]: int x = 3;

```

```

[1]: int ajoute(int a, int b)
{
    return a+b;
}

```

```

[6]: int a = 2; // déclaration de variables locales
int b; // non initialisée ici

b = 3; // nécessite que b soit déclarée.

printf("La somme de %d + %d = %d\n",
      b, x, ajoute(b, x)); // la fonction principale d'affichage, véritable_
↪couteau-suisse           // ici %d signifie un entier qui figure dans la liste juste_
↪après

// On retrouve des boucles for
a = 0;
for (int i = 0; i < 10; i++) // mais on doit gérer notre compteur nous même
{
    a = a + i;
}
printf("La somme des 9 premiers entiers vaut %d\n", a);

```

La somme de 3 + 3 = 6

La somme des 9 premiers entiers vaut 45

```

[44]: // si on veut récupérer une valeur, on ne met pas le point-virgule à la fin :
int x = 3;
x + 2

```

[44]: 5

On procédera donc comme en OCaml pour ce TP : on vous demandera de remplir le corps de fonction et ensuite une série de tests sera effectuée pour vous assurer que vous ayez réussi.

```
[1]: int multiplie(int a, int b)
{
    // renvoie le produit de a et de b
    // pour le moment allez voir le corps de ajoute au dessus et adaptez

    // BEGIN SOLUTION
    return a * b;
    // END SOLUTION
}
```

```
[5]: // LES TESTS MAINTENANT RESSEMBLENT À ÇA
if(multiplie(2, 3) != 6) throw 0;
```

2 Les bases du C

2.1 Types et variables

Mise à part l'ajout des points-virgules à la fin des instructions, la différence principale entre OCaml, Python et C dans ce qu'on vient de voir se situe au niveau des instructions explicites de type et donc de la notion de déclaration de variable.

On a en C des types de bases sur lesquels on reviendra dans les exercices : `int`, `float`, `bool` mais une variable doit être **déclarée** comme ayant un type donné et, bien qu'on puisse en changer la valeur, on ne pourra pas changer le type des données qu'elle contient.

Pour déclarer une variable `x` ayant le type `t` on écrit `t x;` par exemple `int b;` ici. Une fois une variable déclarée, on peut l'utiliser en lecture et en écriture. Mais il faut écrire dedans AVANT d'y lire quoi que ce soit. Pour cela, on pourra écrire `x = val;` pour que la variable `x` prenne la valeur `val`. Il est possible de déclarer et d'initialiser une variable, comme dans `int x = val;`.

ATTENTION : si on écrit juste `int x;` sans valeur, on ne peut pas supposer que `x` contienne une valeur spécifique comme 0. On peut juste dire que `x` contient un entier quelconque qu'on se gardera bien d'utiliser dans un calcul.

Ainsi, comme OCaml, C est un langage fortement type, mais dont les types sont explicites.

```
int x; // x n'est pas initialisé mais c'est un entier
float y = 3.2; // y est un flottant initialisé à 3.2
bool b = true; // b est un booléen qui vaut true
char c = 'a'; // le type des caractères
```

Un type propre à C est le type de rien, qui se nomme `void`. Cela correspond au type `unit` de OCaml.

2.2 Blocs et portée

En C on a une notion de bloc qui sont des instructions ou des déclarations de variables entourées d'accolades, comme dans :

```
{  
    int x = 3;  
    printf("x vaut %d\n", x);  
}
```

Une variable déclarée dans un bloc n'existe que dans celui-ci. Dès qu'on sort du bloc on ne peut plus y accéder. Les fonctions, les instructions conditionnelles ou les boucles utilisent des blocs.

[7]: *// exemple avec deux blocs imbriqués où on retrouve la valeur précédente de x
// qui a juste été masquée dans le sous-bloc*

```
{  
    int x = 3;  
    printf("x vaut %d\n", x);  
  
    {  
        int x = 4;  
        printf("x vaut %d\n", x);  
    }  
  
    printf("x vaut %d\n", x);  
}
```

```
x vaut 3  
x vaut 4  
x vaut 3
```

[8]: *// ATTENTION par contre, le code suivant utilise x dans le bloc pour le changer
↳ de*

```
// valeur et comme c'est le x du bloc principal, ça le modifie  
{  
    int x = 3;  
    printf("x vaut %d\n", x);  
  
    {  
        x = 4;  
        printf("x vaut %d\n", x);  
    }  
  
    printf("x vaut %d\n", x);  
}
```

```
x vaut 3  
x vaut 4  
x vaut 4
```

2.3 Fonctions

Une fonction en C à un type de retour et chacun des arguments est donné avec son type. S'en suit un bloc d'instructions qui peut comporter une ou plusieurs instructions **return v**; qui vont renvoyer la valeur *v*. S'il n'y a pas d'instructions cela correspond à renvoyer la valeur de type *void*, ce qu'on peut faire explicitement avec **return**; sans valeur.

On définit une fonction ainsi :

```
type_retour nom_fonction(type_arg1 nom_arg1, ..., type_argn nom_argn)
{
    // bloc d'instructions qui constitue le corps
    // de la fonction
}
```

On est donc très proches des fonctions **Python** mais avec cette contrainte de type.

```
[2]: int f(int a, char c)
    {
        printf("a=%d et c=%c\n", a, c);
        return a*a;
    }
```

on appelle alors la fonction comme en **Python** avec **nom_fonction(val1, ..., valn)** :

```
[5]: f(3, 'z')
```

a=3 et c=z

```
[5]: 9
```

On voit que la valeur de sortie de la fonction est bien le résultat du **return**, le reste c'est juste de l'affichage. On rappelle qu'il a été nécessaire d'omettre le ; pour avoir cette valeur de retour. C'est une spécificité de l'interprète qu'on utilise.

Les arguments sont des variables qu'on peut utiliser directement, cela ne changera pas la valeur passée en paramètre si on les redéfinit. Une fonction peut s'appeler elle-même avec toutes les réserves vues quant à l'utilisation de la pile.

2.4 Instructions conditionnelles

Pour effectuer une **instruction** conditionnelle, on écrit

```
if (condition)
{
    // bloc d'instructions à exécuter quand la condition est vraie
}
else
{
    // bloc d'instructions à exécuter quand la condition est fausse
}
```

Par exemple, on pourra écrire :

```
if (n % 2 == 0)
{
    printf("n est pair\n");
}
else
{
    printf("n est impair\n");
}
```

Ici, comme il n'y a qu'une instruction, on pourrait ne pas faire de bloc et écrire juste l'instruction. Mais on se gardera de faire ça pour le moment, car c'est une source classique d'erreurs.

On note qu'il n'est pas nécessaire d'avoir un **else**. Ici, comme en Python et contrairement à OCaml, on n'a pas de notion de type associé aux branches, ce sont juste des suites d'instructions.

On peut ainsi écrire

```
if (condition)
{
    // instructions si condition est vérifiée
}
```

Notamment : en C, ce **if** est une **instruction** et pas une expression. Si on peut écrire

```
let x = if n % 2 then 1 else 2
```

en OCaml, en C, on devra écrire :

```
int x;

if (n % 2 == 0)
{
    x = 1;
}
else
{
    x = 2;
}
```

Au passage, remarquons qu'on utilise **==** comme en Python. Par contre, contrairement à Python, comme les déclarations ne sortent pas des blocs, on ne peut pas écrire

```
if n % 2 == 0:
    x = 1
else:
    x = 2
print("x vaut ", x)
```

car en C le programme suivant est erroné

```
if (n % 2 == 0)
{
```

```

    int x = 1;
}
else
{
    int x = 2;
}
printf("x vaut %d\n", x);

```

car `x` n'est pas définie en dehors des blocs.

2.4.1 Exercice

Pour la question suivante, on rappelle la définition de la suite de Syracuse :

Étant donné $a \in \mathbb{N}^*$, on appelle suite de Syracuse de premier terme a , la suite $(u_n)_{n \in \mathbb{N}}$ définie par $u_0 = a$ et pour $n \in \mathbb{N}$:

$$u_{n+1} = \begin{cases} \frac{u_n}{2} & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{sinon.} \end{cases}$$

Indications :

- pour déterminer si un nombre a est pair on peut utiliser le modulo et tester `a % 2 == 0` comme en Python.
- une fonction peut avoir plusieurs `return`, donc quand vous avez calculé une valeur, il suffit de la renvoyer.

```

[8]: int suivant(int u)
{
    // renvoyez la valeur du terme qui suit u
    // dans la suite

    // BEGIN SOLUTION
    if (u % 2 == 0)
    {
        return u/2;
    }
    else
    {
        return 3*u+1;
    }
    // END SOLUTION
}

```

```

[8]: if (suivant(14) != 7) throw 0;
    if (suivant(7) != 22) throw 0;

```

2.5 Boucles conditionnelles

Pour effectuer une boucle conditionnelle, on écrit

```
while(condition)
{
    // instructions à réaliser
}
```

Ce sont, comme en Python, les boucles les plus simples mais elles demandent de gérer plus de choses qu'une boucle inconditionnelle. Par exemple, pour calculer la somme $\sum_{k=0}^n k$, on doit nécessairement gérer les indices à la main :

```
int somme_entiers(int n)
{
    int k = 0; // l'indice
    int s = 0; // la somme courante

    while (k <= n)
    {
        s = s + k;
        k = k+1;
    }

    return s;
}
```

Le cas le plus approprié d'une telle boucle est quand on ne peut maîtriser le nombre d'itérations comme dans la fonction suivante :

```
[1]: int sup_log2(int n)
{
    // renvoie le plus petit entier naturel d >= 0 tel que
    // n < 2^d (attention cette puissance n'existe pas en C)

    // BEGIN SOLUTION
    int d = 0;
    int p = 1;
    while (p <= n)
    {
        p = p * 2;
        d = d + 1;
    }

    return d;
    // END SOLUTION
}
```

```
[2]: if (sup_log2(0) != 0) throw 0;
if (sup_log2(1) != 1) throw 0;
if (sup_log2(15) != 4) throw 0;
```

2.6 Boucles for

Contrairement à Python, il n'y a aucun mécanisme permettant de parcourir des valeurs. Mais contrairement aux boucles `for` de OCaml, il ne s'agit pas simplement de boucles inconditionnelles. Le mécanisme est plus complexe.

Commençons par un exemple en reprenant la fonction `somme_entiers` :

```
int somme_entiers(int n)
{
    int s = 0;

    for(int k = 0; k <= n; k++)
    {
        s = s + k;
    }

    return s;
}
```

Déjà, précisons que `k++` est une manière d'écrire `k = k+1`. On parle d'incrementation de `k`.

Ensuite, on peut voir qu'il y a trois parties dans la boucle `for` :

```
for (initialisation; condition; miseajour)
{
    // instructions
}
```

L'initialisation est la plupart du temps la déclaration d'une variable de boucle. Ici, on peut utiliser le raccourci permettant de déclarer et définir la variable directement. Par opposition au fait d'écrire :

```
int k;
for (k = 0; k <= n; k++)
{
    s = s+k;
}
```

Ensuite, vient la condition d'arrêt qui est très proche de celle d'un `while`.

Enfin, la mise à jour est une instruction qui sera effectuée à la fin de chaque itérations. Finalement, une boucle `for` en C n'est qu'un `while` déguisé :

```
initialisation;

while(condition)
{
    // instructions

    miseajour;
}
```

Pour autant, la lisibilité d'une boucle `for` est bien meilleure que celle d'un `while` car elle structure plus en mettant en valeur les **vraies** instructions du corps de boucle. On ne sera pas surpris de trouver ainsi des `for` dès que c'est possible.

```
[3]: int somme_carres(int n)
{
    // renvoie la somme des carrés des entiers de 0 à n :
    //  $0^2 + 1^2 + \dots + n^2$ 

    // BEGIN SOLUTION
    int s = 0;
    for (int k = 0; k <= n; k++)
    {
        s = s + k*k;
    }
    return s;
    // END SOLUTION
}
```

```
[6]: if (somme_carres(13) != 819) throw(0);
     if (somme_carres(0) != 0) throw(0);
     if (somme_carres(1) != 1) throw(0);
```

3 J'aimerais tant (re)voir Syracuse

```
[15]: int syracuse(int a, int n)
{
    // renvoie le n-ieme terme de la suite de Syracuse de premier
    // terme a
    // ATTENTION : On utilisera suivant et on ne fera pas du copier-coller de
    ↪ code.

    // BEGIN SOLUTION
    int u = a;
    for (int k = 1; k <= n; k++)
    {
        u = suivant(u); // on calcule u(k) ici
    }
    return u;
    // END SOLUTION
}
```

```
[17]: if (syracuse(3, 5) != 4) throw(0);
     if (syracuse(1, 0) != 1) throw(0);
     if (syracuse(1, 1) != 4) throw(0);
     if (syracuse(1, 2) != 2) throw(0);
```

```
[18]: int temps_de_vol(int a)
{
    // renvoie le temps de vol pour a c'est-à-dire le plus petit entier n
    // tel que  $u(n) = 1$  quand  $u(0) = a$ 

    // BEGIN SOLUTION
    int n = 0;
    int u = a;
    while (u != 1)
    {
        u = suivant(u);
        n = n+1;
    }
    return n;
    // END SOLUTION
}
```

```
[19]: if (temps_de_vol(1) != 0) throw(0);
      if (temps_de_vol(14) != 17) throw(0);
```

```
[21]: int altitude_maximale(int a)
{
    // renvoie l'altitude maximale de a, c'est-à-dire
    // la plus grande valeur prise par la suite quand  $u(0) = a$ 

    // BEGIN SOLUTION
    int m = 4;
    int u = a;
    while (u != 1)
    {
        u = suivant(u);
        if (u > m)
            m = u;
    }
    return m;
    // END SOLUTION
}
```

```
[22]: if (altitude_maximale(1) != 4) throw(0);
      if (altitude_maximale(2) != 4) throw(0);
      if (altitude_maximale(14) != 52) throw(0);
```

4 Les tableaux

4.1 Déclaration et initialisation

La structure de données la plus primitive en **C** est celle des tableaux. Attention, ce sont des tableaux de taille fixe comme les 'a array de OCaml. Pour déclarer un tableau **t** de **n** valeurs (**où n est une CONSTANCE et pas une variable**) d'un type **T** on écrit

```
T t[n];
```

par exemple `int a[10];` déclare **a** comme étant un tableau de 10 entiers.

On peut alors accéder en lecture et en écriture à la *i*ème case du tableau en écrivant `t[i]`. Ici tout se passe comme si **t** était une collection de variables accessibles par des indices. De plus, on a l'assurance que les cases se suivent linéairement en mémoire, ce qui sera utile dans la suite.

REMARQUE : **C** est un langage sans garde-fou, on peut tout à fait écrire `t[100]` pour un tableau de taille 3 et ainsi aller écrire n'importe où en mémoire et

explosion d'Ariane V au premier lancement suite à une mauvaise écriture en mémoire ayant lancé le programme de destruction

ATTENTION : dans presque tous les cas, les valeurs ne sont pas initialisées, et comme pour les variables, il faut attribuer des valeurs avant de travailler sur le tableau. Notamment, les cases ne sont pas initialisées à zéro.

Il est possible d'initialiser un tableau au moment de sa déclaration. Pour cela, on met les valeurs entre accolades comme ici :

```
int t[3] = { 1, 3, 5 };
```

Ce qui équivaut à l'écriture de :

```
int t[3];
t[0] = 1;
t[1] = 3;
t[2] = 5;
```

Si jamais on donne moins de valeurs de **n** valeurs pour `t[n]`, on va compléter par des valeurs nulles. Cela permet d'utiliser le cas spécial `{}` pour tout initialiser à zéro. Les trois définitions suivantes sont alors équivalentes :

```
int t[3] = { 0, 0, 0 };
int t[3] = { 0, 0 };
int t[3] = {};
```

Quand on donne une succession complète de valeurs, on connaît déjà le nombre d'éléments du tableau. Il est alors assez redondant de le donner une seconde fois dans la déclaration. On dispose ainsi d'un mécanisme automatique en écrivant juste `T t[] = ...;`. On peut reprendre la déclaration précédente ainsi :

```
int t[] = { 1, 3, 5 };
```

4.2 Tableaux en arguments

Pour passer un tableau en paramètre à une fonction on pourra utiliser, entre autres, `int t[]` ou `int t[10]` avec une taille précise, mais **il n’y a aucune vérification de taille en C**. On prendra donc l’habitude de donner en arguments le nombre d’éléments du tableau.

Ainsi la fonction suivante calcule la somme des éléments d’un tableau.

```
[7]: int somme_tableau(int t[], int nb_elements)
{
    int s = 0;

    for (int i = 0; i < nb_elements; i++)
    {
        s = s + t[i];
    }

    return s;
}
```

```
[9]: int t[] = { 0, 1, 2, 3 };
somme_tableau(t, 4)
```

[9]: 6

```
[45]: // mais on peut très bien appeler la fonction sur le même tableau avec une
      ↪taille plus petite
somme_tableau(t, 3)
```

[45]: 3

```
[47]: // ou une taille plus grande mais alors rappelez-vous de la fusée au dessus...
somme_tableau(t, 1000)
```

[47]: 1310320687

```
[54]: int recherche(int t[], int nb_elements, int x)
{
    // renvoie le plus petit indice de x dans le tableau t
    // si x n'est pas présent, on renvoie -1 (qui est toujours un indice
    ↪invalide)
    // Rappel : on peut faire des return quand on veut

    // BEGIN SOLUTION
    for (int i = 0; i < nb_elements; i++)
    {
        if (t[i] == x)
        {
```

```

        return i;
    }
}

return -1;
// END SOLUTION
}

```

```

[64]: int t[] = { 0, 2, 2, 6 };
      if (recherche(t, 4, 2) != 1) throw(0);
      if (recherche(t, 1, 2) != -1) throw(0);
      if (recherche(t, 0, 2) != -1) throw(0);
      if (recherche(t, 1, 0) != 0) throw(0);
      if (recherche(t, 4, 7) != -1) throw(0);

```

```

[57]: int maximum(int t[], int nb_elements)
{
    // renvoie la valeur du maximum du tableau t
    // on considère que nb_elements > 0 donc il existe toujours

    // BEGIN SOLUTION
    int m = t[0];
    for (int i = 1; i < nb_elements; i++)
    {
        if (m < t[i])
        {
            m = t[i];
        }
    }

    return m;
    // END SOLUTION
}

```

```

[58]: int t[] = { 5, 0, 2, 1, 6 };
      if (maximum(t, 4) != 5) throw(0);
      if (maximum(t, 5) != 6) throw(0);

```

```

[60]: int arg_max(int t[], int nb_elements)
{
    // en déduire une fonction arg_max qui renvoie le plus petit indice
    // du maximum de t où nb_elements > 0
    // ON NE FERA PAS DE BOUCLES ICI

    // BEGIN SOLUTION
    return recherche(t, nb_elements, maximum(t, nb_elements));
    // END SOLUTION
}

```

```
}
```

```
[63]: int t[] = { 5, 0, 5, 1, 6 };  
      if (arg_max(t, 4) != 0) throw(0);  
      if (arg_max(t, 5) != 4) throw(0);
```

4.3 Des tableaux particuliers : les chaînes de caractères

Une chaîne de caractères en C est juste un tableau de `char` dont on impose que la dernière case soit nulle. Dans le contexte des caractères on note `'\0'` le caractère nul mais cela correspond à la valeur 0.

Ainsi :

```
char chaine[] = { 'B', 'o', 'n', 'j', 'o', 'u', 'r', '\0' };
```

est la chaîne contenant `Bonjour`. Comme écrire ainsi les chaînes serait très fastidieux, on utilise la notion usuelle :

```
char chaine[] = "Bonjour";
```

et tout se passe comme dans la définition ci-dessus.

```
[10]: char chaine[] = "Bonjour";  
      chaine[7] // montre bien qu'on a placé un caractère nul en dernière position
```

```
[10]: '0x00'
```

Attention, les caractères en C sont limités, hors de question d'écrire " ". Par contre, on a quand même la possibilité de faire des caractères accentués. Il y a un certain nombre de caractères spéciaux comme `\n` pour effectuer un saut de ligne.

On va maintenant revenir sur la fonction `printf` vue précédemment. C'est une fonction spéciale qui peut prendre un nombre variable d'argument mais qui en aura toujours au moins un `printf(format, ...)` où `format` est une chaîne de caractères. Cette chaîne peut comporter des symboles spéciaux commençant par `%` et affichant, dans l'ordre de gauche à droite, les valeurs passées en argument après `format`.

On a par exemple `%d` pour afficher un entier signé, `%s` pour une chaîne de caractères ou `%c` pour un caractère. Exemple :

```
[11]: char s[] = "Test";  
      int i = 2;  
      printf("Le %deme caractère de la chaine %s est %c\n",  
            i, s, s[i]) // oui on commence à compter à 0
```

Le 2eme caractère de la chaine Test est s

```
[11]: 43
```

Le nombre renvoyé par `printf` est le nombre de caractères affichés après ces substitutions.

On pourra utiliser la fonction suivante pour afficher un tableau :

```
[14]: void print_tableau(int t[], int nb_elements)
{
    for(int i = 0; i < nb_elements; i++)
    {
        char c = ',';
        if (i == nb_elements - 1)
            c = '\n'; // fin de ligne pour le dernier élément.
        printf("%d%c", t[i], c);
    }
}

int t[] = {1, 2, 3};
print_tableau(t, 3);
```

1,2,3

4.4 Tableaux en valeur de retour ?

On ne peut pas renvoyer un tableau défini ainsi en C. En effet, il correspond à une zone mémoire locale à la fonction qui ne peut en sortir. Comme souvent en C, on peut l'écrire, mais ça va avoir un comportement erroné pouvant aboutir à une catastrophe.

La bonne manière d'avoir un tableau en valeur de retour, c'est paradoxalement de le passer en paramètre et d'écrire dedans.

La fonction suivante, par exemple, va renvoyer un tableau **à l'envers**. Pour bien identifier le tableau en entrée et le tableau en sortie, on les a nommé **in** et **out**.

```
[20]: void renverse_tableau(int in[], int out[], int nb_elts)
{
    for(int i = 0; i < nb_elts; i++)
    {
        out[i] = in[nb_elts-1-i];
    }
}

int t[] = {1,2,3};
int q[3];
renverse_tableau(t, q, 3);
print_tableau(t, 3);
print_tableau(q, 3);
```

1,2,3

3,2,1

5 Tri par sélection

On va maintenant revenir sur un problème classique d'algorithmique : la notion de **tri**. On considère des tableaux et on va étudier différentes manières de les trier.

On a des méthodes de tri dites **en place** où on trie un tableau directement en changeant ses valeurs, et des méthodes **non en place** qui vont placer dans un autre tableau le résultat. Pour effectuer des tris en place, on va considérer une fonction permettant d'échanger deux valeurs dans un tableau :

```
[9]: void swap(int t[], int i, int j)
{
    // echange dans t les valeurs des cases d'indice i et j

    // BEGIN SOLUTION
    int temp = t[i];
    t[i] = t[j];
    t[j] = temp;
    // END SOLUTION
}
```

```
[18]: int t[] = { 1, 2, 3 };
swap(t, 0, 2);
if (t[0] != 3 || t[2] != 1) throw (0);
```

Le tri par sélection est un tri qui va constituer un sous-tableau trié de plus en plus grand en tête du tableau. C'est une méthode en place. A l'étape i , on considère que les cases d'indices à 0 à $i-1$ sont triés et contiennent les i plus petites valeur du tableau.

- si $i = n$, où n est la taille du tableau, alors le sous-tableau est en fait le tableau complet : il est trié.
- sinon, on va chercher le minimum du tableau entre les indices i et $n - 1$ et on le place à l'indice i . On passe alors à l'étape $i + 1$.

```
[2]: int indice_minimum_sous_tableau(int t[], int i, int nb_elts)
{
    // renvoie l'indice du plus petit élément du tableau t à partir de l'indice_
    ↪ i
    // on appellera toujours cette fonction avec i < nb_elts

    // BEGIN SOLUTION
    int val_min = t[i];
    int indice_min = i;

    for (int j = i+1; j < nb_elts; j++)
    {
        if (val_min > t[j])
        {
            val_min = t[j];
            indice_min = j;
        }
    }
}
```

```

    }
}

return indice_min;
// END SOLUTION
}

```

```

[4]: int t[] = { 4, 5, 2, 1, 6 };
if (indice_minimum_sous_tableau(t, 0, 5) != 3) throw(0);
if (indice_minimum_sous_tableau(t, 2, 5) != 3) throw(0);
if (indice_minimum_sous_tableau(t, 4, 5) != 4) throw(0);

```

```

[15]: void tri_par_selection(int t[], int nb_elts)
{
    // tri par selection t a l'aide de la fonction précédente

    // BEGIN SOLUTION
    for (int i = 0; i < nb_elts; i++)
    {
        int j = indice_minimum_sous_tableau(t, i, nb_elts);
        swap(t, i, j);
    }
    // END SOLUTION
}

```

```

[16]: int t[] = { 4, 5, 2, 1, 6 };
int q[] = { 1, 2, 4, 5, 6 };
tri_par_selection(t, 5);
for (int i = 0; i < 5; i++)
{
    if (t[i] != q[i])
    {
        printf("Erreur à l'indice %d\n", i);
        throw(0);
    }
}

```