Les sous-programmes (C)
Corrigé

Résumé

Ce document décrit les sous-programmes, outil essentiel pour structurer un programme car il permet au programmeur de définir ses propres instructions et opérateurs.

Table des matières

1 Les sous-programmes en C

1.1 Déclaration des variables

1.1.1 Portée des variables

La portée en C fonctionne comme en algorithmique. Notons qu'à l'exception des variables globales, toutes les autres variables sont déclarées en C dans des accolades (elles doivent même être en début d'accolades avant les instructions). Leur portée est donc délimitée par ces accolades.

1.1.2 Variables globales

Les variables globales sont des variables déclarées à l'extérieur des sous-programmes. Nous n'utiliserons pas les variables globales (sauf cas particulier qui devra être justifié) et déclarerons donc toujours les variables dans un sous-programme.

D'autre part, les variables du programme principal ne risque pas d'être considérées comme globales car elles sont déclarées comme variables locales de la fonction `main()`.

1.1.3 Masquage de variable

Enfin, le masquage de variable peut avoir lieu en C au niveau d'un sous-programme seul. En effet, en C on peut déclarer une variable après toute accolade ouvrante. Ainsi, dans un même sous-programme, on peut avoir plusieurs déclaration de variables à des endroits différents. En voici un exemple.

```
1 int main()
2 {
3     double x;
4     {
5         char x;
6             // ce x de type char masque le x de type réel ci-dessus.
7     }
8 }
```

Il faut absolument éviter ce genre de masquage car il est source d'erreurs difficile à identifier.
Remarque : Si ces accolades ne vous semblent pas très naturelles, rappelez-vous qu'on utilise généralement des accolades après les structures de contrôle.

1.1.4 Exemples

Considérons les instructions suivantes :

```
1 {
2     int x;
3     double x;
4 }
```

Ceci est interdit car dans le même bloc on déclare deux variables différentes avec le même nom. Quand on utilisera x, il y aura ambiguité entre le x **`int`** et le x **`double`**. Le compilateur interdit ce cas et le signale comme une erreur de compilation.

Si les deux variables sont déclarées dans deux blocs différents.

```
1  {
2      int x;
3      int n;
4      ... contexte 1 ....
5      {
6          double x;
7          ... contexte 2 ...
8      }
9      ... contexte 1 suite ....
10 }
```

Nous avons toujours deux variables x différentes qui portent le même nom, mais suivant le contexte, ce sera l'une ou l'autre. Dans le contexte 1, il s'agit du x int. Dans le contexte 2, il s'agit du x double car à l'intérieur du contexte 2, on a déclaré une variable qui s'appelle x et masque (cache) la variable du contexte englobant.

On peut imaginer que chaque bloc est une nouvelle feuille, avec ses variables et instructions. Quand on ouvre un nouveau bloc, on ajoute une nouvelle feuille sur les précédentes. Quand dans les instructions on fait référence à une variable on commence par la chercher dans la feuille en cours. Si on ne la trouve pas, on la cherche dans la feuille suivante et ainsi de suite. Ainsi, dans le contexte 2, x est trouvée dans ce bloc (on n'a pas à aller chercher le x du bloc supérieur – de la feuille suivante). Le x du bloc englobant (contexte 1) est donc masqué. Toujours dans le contexte 2, si on fait référence à n, on ne le trouve pas dans le bloc (la feuille) on va donc voir dans le (la) suivant(e). On le trouve on a donc fait référence à n du contexte 1.

Dernier exemple.

```
1  {
2      int x;
3      ... contexte 1 ...
4  }
5
6  {
7      double x;
8      ... contexte 2 ...
9  }
```

x est déclarée dans deux contextes différents mais ils ne sont pas inclus l'un dans l'autre. Il n'y a alors pas de problème. Dans le contexte 1, on a accès à x **`int`** et seulement à lui. Dans le 2, on a accès à x **`double`** et seulement à lui. En dehors de ces deux contextes, il n'y a pas de x accessible.

1.2 Tout est fonction

En C, les notions de « procédures » et « fonctions » existent mais l'usage peut qu'on les appelle toutes fonctions. Nous garderons cependant la terminologie *fonction* et *procédure* avec leur sens algorithmique.

Comme en algorithmique, une fonction est caractérisée par :
– un identifiant (le nom de la fonction) ;
– une liste de paramètres formels (nombre, noms et types) ;
– un type de retour ;
– un corps composé d'instructions.

Cependant, il n'y a pas de variable **`Résultat`**. On utilise l'opérateur **`return`** `expression` qui arrête l'exécution de la fonction et renvoie comme résultat de la fonction la valeur de `expression`. Nous imposerons qu'il n'y ait qu'un seul **`return`** par fonction, nécessairement placé avant l'accolade finale. [1]

Remarque : Une **procédure** est une « fonction » qui ne renvoie pas de résultat. Son type de retour est donc **`void`**. Elle ne comporte pas de **`return`**. [2]

La syntaxe générale d'une fonction est :

```
/* Description de l'objectif du sous-programme avec les éventuelles
 * préconditions et postconditions.
 */
<type_retour> <nom_fonction> (<paramètres_formels_typés>)
{
    /* corps de la fonction */
    return <expression>;
}
```

En C, il n'y a pas de contrainte sur la valeur de retour d'une fonction. On peut donc renvoyer un enregistrement et un tableau (même si le cas des tableaux est un peu particulier, voir section 1.5)

Exemples : Voici un sous-programme appelé `max2`, en fait une fonction, qui calcule le plus grand des deux entiers passés en paramètre.

```
/* Le plus grand des deux entiers a et b. */
int max2(int a, int b) {
    int resultat;
    if (a > b) {
        resultat = a;
    }
    else {
        resultat = b;
    }
    return resultat;
}
```

Voici une fonction `max3` qui calcule le plus grand des trois entiers passés en paramètre en s'appuyant sur la fonction précédente `max2`. Notons qu'en C, on ne peut utiliser une fonction que si elle est déjà connue du compilateur donc si elle a été déclarée avant.

```
/* Le plus grand des trois entiers n1, n2 et n3. */
int max3(int n1, int n2, int n3) {
    int max12;  /* le plus grand de n1 et n2 */
```

1. En C, il est possible de mettre plusieurs **`return`** mais ceci contredit le principe qu'une fonction, un sous-programme en général, doit avoir un seul point d'entrée et un seul point de sortie.
2. En C, il est possible d'avoir un **`return`** sans expression mais nous l'interdisons.

```
4       max12 = max2(n1, n2);
5       return max2(max12, n3);
6
7           /* Remarque : on aurait pu ne pas utiliser la
8            * variable locale max12 en faisant :
9            *
10           *      return max2(max2(n1, n2), n3);
11           */
12  }
```

Enfin, voici un programme de test qui utilise les fonctions `max2` et `max3`.

```
1   /* Programme de test de la fonction max2() */
2   void tester_max()
3   {
4       int a = 2;
5       int b = 5;
6       int i, j;
7
8       printf("a = %d et b = %d\n", a, b);
9       i = max2(a, b);     /* les deux paramètres sont des variables */
10      printf("i = max2(a, b) = %d\n", i);
11      j = max2(8, 2*i);   /* mais peuvent être des expressions quelconques */
12      printf("j = max2(8, 2*i) = %d\n", j);
13      printf("max3(9, i, j) = %d", max3(9, i, j));
14  }
```

Le résultat de l'exécution donne :

```
1   a = 2 et b = 5
2   i = max2(a, b) = 5
3   j = max2(8, 2*i) = 10
4   max3(9, i, j) = 10
```

1.3 Mode de passage des paramètres

En C, il n'existe qu'un seul mode de passage des paramètres, le passage par valeur (section 1.3.1). La question est alors comment faire pour écrire un sous-programme qui doit changer la valeur des paramètres effectifs (comme par exemple permuter) ? La solution est de s'appuyer sur la notion de pointeur (section 1.3.2) pour faire un passage « par adresse » (section 1.3.3) qui est un fait un passage par valeur de l'adresse d'une variable. Ça paraît compliqué... mais c'est développé dans les points suivants.

1.3.1 Le passage par valeur

Le seul mode de passage de paramètres en C est donc le passage par valeur. Ceci signifie que les paramètres formels d'un sous-programme sont assimilables à des variables locales qui sont initialisées avec la valeur du paramètre effectif correspondant (fourni par le programme appelant).

Il y a donc copie dans la pile d'exécution de la valeur des paramètres effectifs. Cette copie est accessible sous le nom du paramètre formel. Les conséquences sont donc qu'en C on peut :

- affecter un paramètre formel et plus généralement changer sa valeur ;
- ses changements étant faits sur la copie dans la pile d'exécution ils ne sont pas visibles du programme appelant.

Considérons l'exemple suivant.

```c
/*****************************************************************************
 *  Auteur   : Xavier Crégut <cregut@enseeiht.fr>
 *  Version  : 1.1
 *  Objectif : Exemple illustrant le passage par valeur en C.
 *****************************************************************************/

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

void proc(int a, double b)
{
    printf("proc début : a = %d et b = %f\n", a, b);
    a++;        /* On incrémente la copie du paramètre effectif */
    b = b * 2;  /* Idem */
    printf("proc fin   : a = %d et b = %f\n", a, b);
}

int main()
{
    int i = 2;
    double x = 3.14;
    printf("main début : i = %d et x = %f\n", i, x);
    proc(i, x);
    printf("main fin   : i = %d et x = %f\n", i, x);

    return EXIT_SUCCESS;
}
```

Les valeurs du programme principal (`main`) sont i et x qui ont respectivement les valeurs 2 et 3.14. Lors de l'appel à la procédure `proc`,[3] de la place est réservée dans le pile d'exécution pour stocker la valeur de deux variables a et b de types respectifs **`int`** et **`double`**. Ces deux variables correspondent aux paramètres formels qui sont initialisés avec la valeur des paramètres effectifs (en l'occurrence 2 et 3.14, les valeurs de i et x). Les valeurs des paramètres formels sont modifiées dans le sous-programmes, c'est donc la zone réservée dans la pile d'exécution qui est modifiée, pas la zone mémoire occupée par les variables i et x du programme principal. Aussi quand la procédure `proc` se termine, les valeurs de i et x n'ont pas changées.

Voici le résultat de l'exécution de ce programme.

```
main début : i = 2 et x = 3.140000
proc début : a = 2 et b = 3.140000
proc fin   : a = 3 et b = 6.280000
```

3. Ce nom n'est pas significatif et le sous-programme n'a pas de commentaire car il a pour seul but d'illustrer le mode de passage de paramètres par valeur.

```
4 main fin   : i = 2 et x = 3.140000
```

Le mode de passage par valeur peut être utilisé pour le mode **in** identifié un algorithmique.

Remarque : Pour avoir réellement un mode **in**, on peut ajouter un **const** devant le type du paramètre. Le compilateur émet alors au moins un message d'avertissement si on essaie de changer la valeur de ce paramètre. Voici une fonction qui calcule la distance entre deux réels en essayant de changer la valeur des paramètres (ce n'est qu'un exemple illustratif, il serait beaucoup plus naturel – et plus lisible – de renvoyer la valeur absolue de la différence !).

```
1  /*****************************************************************************
2   *  Auteur   : Xavier Crégut <cregut@enseeiht.fr>
3   *  Version  : 1.1
4   *  Objectif : Illustrer le const pour réaliser une passage en mode « in ».
5   *****************************************************************************/
6
7  #include <stdio.h>
8  #include <stdlib.h>
9
10 /* distance entre les réels a et b.  */
11 double distance(const double a, const double b)
12 {
13     a = a - b;
14     if (a < 0) {        /* rendre a > 0 */
15         a = - a;
16     }
17     return a;
18 }
19
20 int main()
21 {
22     printf("distance(1,␣3)␣=␣%f\n", distance(1, 3));
23     printf("distance(3,␣1)␣=␣%f\n", distance(3, 1));
24
25     return EXIT_SUCCESS;
26 }
```

Le compilateur `gcc` emet les messages d'avertissement suivants :

```
1 exemple-distance-in.c: In function ‘distance´:
2 exemple-distance-in.c:13:5: error: assignment of read-only parameter ‘a´
3      a = a - b;
4      ^
5 exemple-distance-in.c:15:2: error: assignment of read-only parameter ‘a´
6   a = - a;
7   ^
```

1.3.2 Les pointeurs

Définition : Un pointeur correspond à l'adresse en mémoire d'une donnée typée.

Par extension, on appelle également pointeur une variable dont la valeur est un pointeur et le type de cette variable.

Notation : Une variable de type pointeur est déclarée avec un ∗ entre le type et l'identifiant de la variable.

```
1 int *pi;        /* pi est un pointeur sur un int */
2 double *pr;     /* pr est un pointeur sur un double */
```

Attention : Lorsque l'on parle de pointeur, il faut toujours préciser le type de l'information qui est pointée (qui se trouve à l'adresse considérée en mémoire). Ainsi, un pointeur sur un **int** et un pointeur sur un **double**, même si ce sont deux pointeurs, ne sont pas considérés comme des types compatibles.

Exemple simple : Commençons par un exemple simple qui montre le principe des pointeurs.

```
1  int i;          /* un entier i */
2  int j;          /* un autre entier j */
3  int *pi;        /* un pointeur sur entier pi */
4
5  i = 5;          /* i est initialisé à 5 */
6  pi = &i;        /* pi est initialisé avec l'adresse en mémoire de i */
7  j = *pi;        /* *pi est la valeur contenue dans la zone mémoire
8                   * pointée par pi.  * est l'opérateur de déréférencement
9                   */
10 printf("j = %d\n", j);  /* j = 5 */
11 *pi = 0;        /* changer la valeur de la zone mémoire pointée par pi */
12 printf("i = %d\n", i);  /* i = 0 */
13     /* Conclusion : puisque pi est initialisé avec l'adresse de i, i
14     * et *pi sont deux moyens différents d'accéder à la même zone
15     * mémoire.
16     */
```

Opérateur d'adressage : L'opérateur d'adressage, noté &, permet d'obtenir l'adresse en mémoire d'une variable. Cette adresse est compatible avec le type pointeur correspondant au type de la variable.

Opérateurs sur les pointeurs : Plusieurs opérateurs sont disponibles sur les pointeurs :

- le *déréférencement* : noté ∗, il fait référence à la zone mémoire pointée. Par exemple, si `pi` est un pointeur sur un entier, `*pi` est l'entier pointé.
  On ne peut utiliser l'opérateur de déréférencement que si le pointeur contient l'adresse d'une zone mémoire valide (c'est-à-dire que le programmeur a réservée). Dans le cas contraire, c'est une erreur dans le programme qui n'est pas détectée à la compilation et ne provoque pas toujours (malheureusement !) d'erreurs à l'exécution.
- l'*affectation* : ceci permet d'initialiser un pointeur. Un pointeur peut être initialisé avec :
  - l'adresse en mémoire d'une donnée de même type en utilisant l'opérateur d'adressage.

    ```
    1 int i;
    2 int *pi;
    3 pi = &i;        /* pi est initialisé avec l'adresse en mémoire de i */
    4     /* Remarque : i et *pi sont deux moyens différents pour accéder à
    5      * le même zone en mémoire.
    6      */
    ```

  - la valeur d'un pointeur. Les deux pointeurs pointent alors tous les deux sur la même zone mémoire ;

```c
int i;          /* un entier i */
int *pi;        /* un pointeur sur entier pi */
int *p2;        /* un autre pointeur sur entier */
pi = &i;
p2 = pi;        /* p2 et pi contiennent tous les deux l'adresse de i */
```

– la constante NULL (définie dans `stdlib.h`) qui signifie que le pointeur ne pointe sur aucune zone mémoire. Il ne peut pas être déréférencé.

```c
int *pi;
pi = NULL;
    /* Remarque : *pi n'a alors pas de sens.  Essayer de le faire
     * devrait alors provoquer une erreur à l'exécution (ceci est
     * dépendant du compilateur) !
     */
```

– les opérateurs d'égalité : il est possible d'utiliser `==` et `!=` pour tester l'égalité et la différence.
– les opérateurs relationnels : on peut également utiliser les autres opérateurs de comparaison `>, <, <=, >=` qui consiste à comparer la valeur des adresses (voir section 1.5).
– les opérateurs arithmétiques : on peut incrémenter un pointeur. Ceci revient à avancer dans la mémoire d'un nombre d'octets égal à celui nécessaire pour représenter en mémoire la donnée pointée. On passe donc à la donnée « suivante » en mémoire. Ceci n'a de sens que si les données sont contiguës en mémoire, donc s'il s'agit de tableau (voir section 1.5)

Attention : Les opérateurs de comparaison et les opérateurs arithmétiques font partie de ce que l'on appelle l'arithmétique sur les pointeurs et que nous n'utiliserons pas !
Remarque : Les pointeurs servent également lorsque l'on fait de l'allocation dynamique de mémoire mais ce n'est pas l'objet de ce chapitre !
Important : Pour ce qui concerne le passage de paramètre par adresse, les seuls opérateurs qui nous intéressent sont l'opérateur d'adressage (`&`), et l'opérateur de déférencement (`*`). L'initialisation du pointeur se fera automatiquement pas l'appel du sous-programme.

Notons enfin un cas particulier lorsque l'on a un pointeur sur un enregistrement. Considérons un type Date, une variable de ce type `d1` et un pointeur sur cette variable `p` :

```c
struct Date {
    int jour;
    int mois;
    int annee;
};
typedef struct Date Date;

Date d1;
Data *p = &d1;
```

Le pointeur p est initialisé avec l'adresse de d1. p et d1 permettent donc de manipuler la même donnée Date. En toute rigueur, pour obtenir le jour de cette date à partir de p, il faudrait écrire :

```c
    (*p).jour
```

En effet, `*p` est du type date et on peut donc ensuite sélectionner le champ `jour`. Les parenthèses sont nécessaires à cause de la priorité des opérateurs : l'opérateur d'accès à un champ `.` est

plus prioritaire que l'opérateur de déréférencement ∗. Cette notation est donc lourde et une autre notation est proposée par C :

```
1       p->jour
```

La notation `->` remplace donc `(* ).` qui est plus lourde et moins lisible. Aussi dans le cas d'un pointeur sur un enregistrement, on utiilser toujours la notation `->` pour accéder à un champ.

1.3.3 Le passage par adresse

Pour pouvoir modifier la valeur d'un paramètre en C, l'astuce consiste à ne pas passer en paramètre une variable, mais l'adresse de cette variable. Le sous-programme a alors directement accès à la zone mémoire du programme appelant et peut donc la modifier.

Je qualifie ceci d'une astuce car il s'agit bien d'un passage par valeur de l'adresse de la variable, et non d'un réel passage en mode **`out`** ou **`in out`**. Ainsi, dans le sous-programme, il faudra penser à déréférencer le paramètre formel et lors de l'appel il faudra penser à donner l'adresse de la variable.

Voici comment écrire en C la fonction permuter.

```
1  /* Permuter la valeur des deux caractères c1 et c2 */
2  void permuter(char *c1, char *c2)
3  {
4      char tmp;   /* pour conserver la valeur de *c1 */
5      tmp = *c1;
6      *c1 = *c2;
7      *c2 = tmp;
8  }
```

Voici comment on peut (doit l'utiliser).

```
1  char lettre = 'A';
2  char chiffre = '5';
3
4  printf("lettre = %c et chiffre = %c\n", lettre, chiffre);
5  permuter(&lettre, &chiffre);
6  printf("lettre = %c et chiffre = %c\n", lettre, chiffre);
```

Remarque : C'est ce que nous avons utilisé pour scanf...

1.4 Les préconditions et postconditions de la programmation par contrat

En C, il n'est pas possible d'exploiter automatiquement les préconditions et postconditions qui ont été identifiées lors de la spécification d'un sous-programme.

Cependant, il existe un module appelé `<assert.h>` qui offre la macro [4] `assert(condition)`. Si la condition est vraie alors l'exécution du programme continue normalement sinon un message d'erreur est affiché et l'exécution stoppée. La macro `assert` peut donc être utilisée dans le code pour vérifier qu'une propriété identifiée lors de l'écriture du programme est bien vérifiée

4. Une macro ressemble à une fonction C mais est traitée non pas par le compilateur C mais par le préprocesseur.

lors de son exécution. Si ce n'est pas le cas, l'arrêt du programme et le message d'erreur nous permettrons de savoir qu'il y a un problème.

Voici comment écrire en C la fonction pgcd en tenant compte des préconditions et postconditions.

```c
/*****************************************************************************
 *  Auteur   : Xavier Crégut <cregut@enseeiht.fr>
 *  Version  : 1.1
 *  Objectif : Programme du pgcd et de son test
 *****************************************************************************/

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

#include <assert.h>

/* calculer le pgcd de deux entiers strictement positifs a et b.
 *
 * Nécessite
 *      a > 0;
 *      b > 0;
 */
int pgcd (int a, int b)
{
    assert(a > 0);      /* utilisation de assert pour les préconditions */
    assert(b > 0);

    while (a != b) {
        if (a > b)
            a = a - b;
        else
            b = b - a;
    }
    return b;
}

/* Tester la fonction pgcd */
void tester_pgcd()
{
    printf("pgcd(4, 5) = %d (doit être 1)\n", pgcd(4, 5));
    printf("pgcd(10, 15) = %d (doit être 5)\n", pgcd(10, 15));
    printf("pgcd(0, 15) = %d (doit être 5)\n", pgcd(0, 15));
}

int main()
{
    tester_pgcd();
    return EXIT_SUCCESS;
}
```

On peut alors se poser la question de ce que donne le programme de test. On obtient donc :

```
pgcd(4, 5) = 1 (doit être 1)
```

```
2 pgcd(10, 15) = 5 (doit être 5)
3 exemple-pgcd: exemple-pgcd.c:20: pgcd: Assertion 'a > 0' failed.
```

Si on n'avait pas mis de `assert` pour vérifier les préconditions, on aurait eu un programme qui bouclait sans fin !

Remarque : Dans le version finale du programme, lorsqu'il a été bien testé et qu'on pense qu'il ne contient plus d'erreur, on peut désactiver les vérifications faites par `assert` en ajoutant l'option `-DNDEBUG` au compilateur C. Il n'est donc pas nécessaire de supprimer physiquement les appels à `assert` dans le programme.

1.5 Cas particulier des tableaux

Les tableaux sont un cas particulier en C. Les tableaux n'existent pas réellement et sont représentés comme des pointeurs.

Normalement, lorsque l'on parle d'un tableau d'entiers par exemple, on désigne l'ensemble des cases du tableau et les entiers qu'elles contiennent. On pourrait donc s'attendre à ce que lors d'un passage par valeur d'un tableau, tous les éléments du tableau soient copiés dans la pile. Ainsi, si le sous-programme modifie un élément du tableau, cette modification ne devrait pas être visible du programme appelant. Et bien il n'en est rien !

Prenons l'exemple de deux sous-programmes. Le premier permet de trier un tableau.

```
1  /* Trier le tableau tab qui contient nb éléments.
2   * @param tab   le tableau à trier
3   * @param nb    le nombre d'éléments de tab
4   */
5  void trier_insertion(int tab[], int nb)
6  {
7      int indice;         /* tab[0..indice] est trié */
8      int position;       /* position d'insertion de tab[indice] */
9      int i;              /* variable de boucle */
10     int memoire;        /* conserver la valeur de tab[indice]   */
11
12     for (indice = 1; indice < nb; indice++) {
13         /* Insérer tab[indice] dans tab(0..indice-1) */
14
15         /* Conserver la valeur de tab[indice] */
16         memoire = tab[indice];
17
18         /* déterminer la position théorique de tab[indice] */
19         position = 0;
20         while (position < indice && memoire >= tab[position]) {
21             position++;
22         }
23
24         /* décaler les éléments compris entre position et indice-1 */
25         for (i = indice-1; i >= position; i--) {
26             tab[i+1] = tab[i];
27         }
28
```

```
29          /* ranger l'élément */
30          tab[position] = memoire;
31      }
32  }
```

Le second permet d'afficher les éléments d'un tableau.

```
1  /* Afficher le tableau tab qui contient nb éléments.
2   * @param tab   le tableau à afficher
3   * @param nb    le nombre d'éléments à afficher
4   */
5  void afficher_tableau(int tab[], int nb)
6  {
7      /* Afficher le tableau */
8      printf("[");
9      if (nb > 0) {       /*{ le tableau n'est pas vide }*/
10         int i;  /* pour parcourir les indices du tableau */
11
12         /* afficher le premier élément */
13         printf("%i", tab[0]);
14
15         /* afficher les autres éléments */
16         for (i=1; i < nb; i++) {
17             printf("; %i", tab[i]);
18         }
19     }
20     printf("]");
21 }
```

On peut se demander ce que donne le programme de test suivant. Est-ce que le tableau est trié ou non, sachant que la tableau semble être passé par valeur ?

```
1  void tester1()
2  {
3      int tab1[10] = { 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 };
4      afficher_tableau(tab1, 10); printf("\n");
5      trier_insertion(tab1, 10);
6      afficher_tableau(tab1, 10); printf("\n");
7  }
```

Le résultat de ce programme montre que le tableau est trié. Il ne s'agit donc pas d'un passage par valeur du tableau et de ses éléments !

```
1  [9; 8; 7; 6; 5; 4; 3; 2; 1; 0]
2  [0; 1; 2; 3; 4; 5; 6; 7; 8; 9]
```

En conséquence, le tableau n'est pas passé par valeur mais bien par adresse. C'est en fait l'adresse du premier élément qui est passé.

En fait les tableaux n'existent pas réellement en C et les crochets ne sont qu'une facilité syntaxique. Ainsi, noter `tab[i]` est identique à noter`*(tab+i)`. `tab+i` est donc l'adresse du i^{e} élément du tableau.

```
1  int tab[10] = {9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
2  assert(tab == &tab[0]);                 /* tab == &tab[0] */
```

```
3 assert(tab[0] == *tab && *tab == 9);    /* tab[0] == *tab == 9 */
4 assert(tab[9] == *(tab+9));             /* tab[9] == *(tab+9) */
5 assert(*(tab + 9) == 0);                /*        == 0 */
```

Ceci permet des choses très condensés, peu lisibles mais parfois un peu efficaces. Nous n'utiliserons pas l'arithmétique sur les pointeurs et utiliserons la notation des tableaux avec les crochets.

Revenons sur l'exemple du tri d'un tableau pour voir comment tirer partie du fait que seule l'adresse du premier élément soit passée en paramètre. On peut le voir sur ce deuxième programme dans lequel on commence par trier la première moitié du tableau (on donne 5 pour le nombre d'éléments et non 10), puis la seconde moitié. Pour trier la seconde moitié, on donne comme premier paramètre `tab+5` qui est équivalent à `&tab[5]`, c'est-à-dire l'adresse du 5^{e} élément. On donne 5 pour la taille. On ne doit pas donner plus sinon on sort de la limite du tableau. Après exécution, on constate que les deux moitiés du tableau sont bien triées (le tableau n'étant bien sûr pas trié dans son ensemble).

```
1 void tester2()
2 {
3     int tab1[10] = { 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 };
4     afficher_tableau(tab1, 10); printf("\n");
5     trier_insertion(tab1, 5);
6     afficher_tableau(tab1, 10); printf("\n");
7     trier_insertion(tab1+5, 5);
8     afficher_tableau(tab1, 10); printf("\n");
9 }
```

Le résultat de ce programme montre que le tableau est trié.

```
1 [9; 8; 7; 6; 5; 4; 3; 2; 1; 0]
2 [5; 6; 7; 8; 9; 4; 3; 2; 1; 0]
3 [5; 6; 7; 8; 9; 0; 1; 2; 3; 4]
```

1.6 Remarques diverses

On ne peut pas imbriquer les sous-programmes même si certains compilateurs (en particulier gcc/g++) le permettent !