Plan

Plan

Variables et fonctions : *une entente peu cordiale*

Exemple

```
1 void echange(int x, int y) {
2   int r;
3   r = x;
4   x = y;
5   y = r;
6 }
```

l'appel à la fonction `echange` sur des variables `x` et `y` n'effectue pas l'effet attendu (échanger les valeurs).

↪ les paramètres sont toujours passés par copie !

La solution

Transmettre les adresses des variables plutôt que leur valeur

↪ *manipuler des paramètres (et variables) de type* ***pointeur***

Les pointeurs

Définition

Un **pointeur** est une variable (ou un paramètre) qui contient l'adresse mémoire d'une donnée typée (une autre variable, un paramètre, ...).

Syntaxe

```
type *nom;
```

- `nom` est l'identificateur de la variable pointeur
- `type*` est le type de cette variable (pointeur de données de type `type`)
- `type` est donc le type de la donnée pointée

Un pointeur occupe 8 octets et contient comme valeur une adresse mémoire qui repère une zone contenant une valeur du type du pointeur.

Exemple

`int *pi;` définit la variable `pi` comme un pointeur sur un entier.
⇒ `pi` devra donc contenir l'adresse d'une donnée de type `int`

Exemples de déclarations et affectations de pointeurs

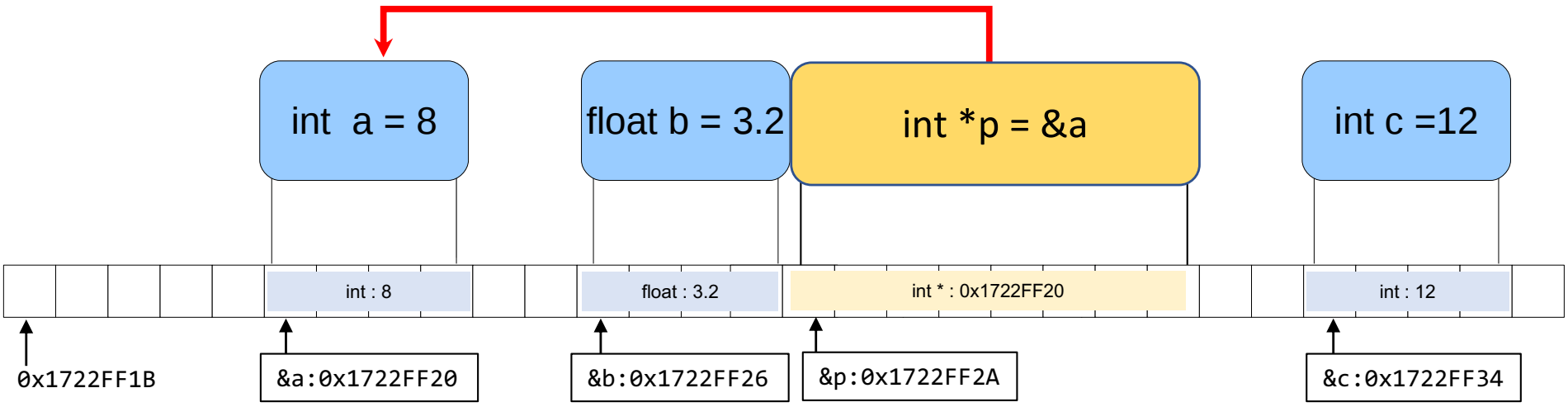
Déclaration de pointeurs :

```
1 int *pa;
2 double *pb;
3 int *pc;
```

Affectation de pointeurs :

```
1 int a; double b;
2
3 pa = &a;   // OK
4 pb = &b;   // OK
5 pa = &b;   // Pb : type pointeur
6 pb = &a;   // Pb : type pointeur
7 pa = pb;   // Pb : type pointeur
8 pa = pc;   // OK
```

les pointeurs en mémoire



L'affectation des pointeurs

On peut affecter une variable pointeur avec :

- l'adresse d'une variable simple *du type de donnée pointée par le pointeur*,
- la valeur d'un autre pointeur *du même type*,
- le pointeur vide : 0 ou constante `NULL` définie dans `<stddef.h>`.

Exemple

```
int a, *p, *q;
p = &a;
q = NULL;
q = p;
```

Le déréférencement

Pour accéder à la zone mémoire pointée par un pointeur, il faut utiliser l'opérateur préfixe de déréférencement `*`.

Définition

Soit `var` une variable pointeur d'un type type (le type de var est donc type *), son déréférencement `*var` désigne la donnée stockée à l'adresse mémoire contenue dans `var`
↪ on parle de donnée pointée.

`*var` est vu comme une variable de type type.

```
1 int a;
2 int *ptr;
3 ptr = &a; // affecte ptr avec l'adresse de a
4 *ptr = 3; // affecte la zone memoire d'adresse ptr
5           // avec la valeur 3 : des lors a vaut 3 !
```

Manipulation des données pointées

L'opérateur de déréférencement permet de manipuler la donnée pointée comme une variable du type pointé. On peut donc :

- utiliser la valeur stockée dans la donnée pointée dans une expression
- modifier la valeur stockée dans la donnée pointée par affectation : *seul cas où la partie gauche d'une affectation n'est pas une variable !*

Exemple

```
int a, *p;
a = 10; p = &a;
a = *p+1;
*p = 13;
```

Attention, ne pas confondre le symbole * :

- comme déclarateur de pointeur : `int *p1, v1, *p2;` ou `int v2, *p3;` (`p1`, `p2`, `p3` *sont des pointeurs (d'entiers)*, `v1`, `v2` *des entiers)*
- comme opérateur de déréférencement dans une expression ou une affectation : `*p1 = *p1 + 2`

Pointeurs et affectation

Syntaxe

Si `p` est une variable pointeur de type `t` : `t *p;`

- `p = exp;` affecte à `p` la valeur de `exp` qui doit donc être une adresse repérant une donnée de ce type `t`
- `*p = exp;` affecte à `*p` (la donnée pointée par `p`) la valeur de `exp` qui doit être une valeur du type `t`

Attention

La déclaration avec initialisation d'un pointeur initialise le pointeur (l'adresse) pas la donnée pointée !

- `type *p = exp ;` déclare `p` comme un pointeur et initialise `p` (son adresse) à la valeur de `exp`.
  ↪ rien n'est fait sur `*p` !
- `type *p = exp ;` est donc équivalent à `type *p ; p = exp ;`

Les pointeurs : Exemple

```c
#include <stdio.h>

int main() {
    int a, *pa;
    a = 150; pa = &a;
    printf("a=%d, &(a)=%p, pa=%p, &(pa)=%p, *(pa)=%d\n",
                a,      &a,    pa,       &pa,       *pa);
    a = 99;
    printf("a=%d, &(a)=%p, pa=%p, &(pa)=%p, *(pa)=%d\n",
                a,      &a,    pa,       &pa,       *pa);
    *pa = 45;
    printf("a=%d, &(a)=%p, pa=%p, &(pa)=%p, *(pa)=%d\n",
                a,      &a,    pa,       &pa,       *pa);
    return 0;
}
```

Exécution :

```
a=150, &(a)=0x16b7c3958, pa=0x16b7c3958, &(pa)=0x16b7c3950, *(pa)=150
```

Les pointeurs : Exemple

```c
#include <stdio.h>

int main() {
    int a, *pa;
    a = 150; pa = &a;
    printf("a=%d, &(a)=%p, pa=%p, &(pa)=%p, *(pa)=%d\n",
                a,      &a,    pa,       &pa,       *pa);
    a = 99;
    printf("a=%d, &(a)=%p, pa=%p, &(pa)=%p, *(pa)=%d\n",
                a,      &a,    pa,       &pa,       *pa);
    *pa = 45;
    printf("a=%d, &(a)=%p, pa=%p, &(pa)=%p, *(pa)=%d\n",
                a,      &a,    pa,       &pa,       *pa);
    return 0;
}
```

Exécution :

```
a=150, &(a)=0x16b7c3958, pa=0x16b7c3958, &(pa)=0x16b7c3950, *(pa)=150
a=99,  &(a)=0x16b7c3958, pa=0x16b7c3958, &(pa)=0x16b7c3950, *(pa)=99
```

Les pointeurs : Exemple

```c
#include <stdio.h>

int main() {
    int a, *pa;
    a = 150; pa = &a;
    printf("a=%d, &(a)=%p, pa=%p, &(pa)=%p, *(pa)=%d\n",
                a,      &a,     pa,         &pa,        *pa);
    a = 99;
    printf("a=%d, &(a)=%p, pa=%p, &(pa)=%p, *(pa)=%d\n",
                a,      &a,     pa,         &pa,        *pa);
    *pa = 45;
    printf("a=%d, &(a)=%p, pa=%p, &(pa)=%p, *(pa)=%d\n",
                a,      &a,     pa,         &pa,        *pa);
    return 0;
}
```

Exécution :

```
a=150, &(a)=0x16b7c3958, pa=0x16b7c3958, &(pa)=0x16b7c3950, *(pa)=150
a=99,  &(a)=0x16b7c3958, pa=0x16b7c3958, &(pa)=0x16b7c3950, *(pa)=99
a=45,  &(a)=0x16b7c3958, pa=0x16b7c3958, &(pa)=0x16b7c3950, *(pa)=45
```

Les pointeurs : Exemples de mauvaise utilisation

```c
#include <stdio.h>

int main() {
    int a, *pa;
    a = 150; pa = &a;
    *a = 100;
    &a = pa;
    pa = 50;
    printf("a=%d, &(a)=%p, pa=%p, &(pa)=%p, *(pa)=%d\n",
                a,      &a,     pa,       &pa,       *pa);
    return 0;
}
```

Compilation :

Les pointeurs : Exemples de mauvaise utilisation

```
1  #include <stdio.h>
2
3  int main() {
4      int a, *pa;
5      a = 150; pa = &a;
6      *a = 100;
7      &a = pa;
8      pa = 50;
9      printf("a=%d, &(a)=%p, pa=%p, &(pa)=%p, *(pa)=%d\n",
10                 a,       &a,     pa,         &pa,        *pa);
11     return 0;
12 }
```

Compilation :

- affectePointeur.c:6:5: error: indirection requires pointer operand ('int' invalid)

Les pointeurs : Exemples de mauvaise utilisation

```
1  #include <stdio.h>
2
3  int main() {
4      int a, *pa;
5      a = 150; pa = &a;
6      *a = 100;
7      &a = pa;
8      pa = 50;
9      printf("a=%d, &(a)=%p, pa=%p, &(pa)=%p, *(pa)=%d\n",
10                 a,       &a,     pa,         &pa,        *pa);
11     return 0;
12 }
```

Compilation :

- affectePointeur.c:6:5: error: indirection requires pointer operand ('int' invalid)
- affectePointeur.c:7:8: error: expression is not assignable

Les pointeurs : Exemples de mauvaise utilisation

```c
#include <stdio.h>

int main() {
    int a, *pa;
    a = 150; pa = &a;
    *a = 100;
    &a = pa;
    pa = 50;
    printf("a=%d, &(a)=%p, pa=%p, &(pa)=%p, *(pa)=%d\n",
                a,      &a,    pa,       &pa,       *pa);
    return 0;
}
```

Compilation :

- affectePointeur.c:6:5: error: indirection requires pointer operand ('int' invalid)
- affectePointeur.c:7:8: error: expression is not assignable
- affectePointeur.c:8:8: warning: incompatible integer to pointer conversion assigning to 'int *' from 'int'

Les pointeurs : Exemples de mauvaise utilisation

```
1  #include <stdio.h>
2
3  int main() {
4      int a, *pa;
5      a = 150; pa = &a;
6      // *a = 100;
7      // &a = pa;
8      pa = 50;
9      printf("a=%d, &(a)=%p, pa=%p, &(pa)=%p, *(pa)=%d\n",
10                 a,       &a,     pa,         &pa,        *pa);
11     return 0;
12 }
```

Compilation :

- affectePointeur.c:8:8: warning: incompatible integer to pointer conversion assigning to 'int *' from 'int'

Exécution :

Segmentation fault: 11

Les pointeurs : Exemples de mauvaise utilisation

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

int main() {
    int a, *pa;
    a = 150; pa = &a;
    pa = NULL;
    printf("a=%d, &(a)=%p, pa=%p, &(pa)=%p, *(pa)=%d\n",
                a,      &a,    pa,       &pa,       *pa);
    return 0;
}
```

Compilation : ok

Les pointeurs : Exemples de mauvaise utilisation

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

int main() {
    int a, *pa;
    a = 150; pa = &a;
    pa = NULL;
    printf("a=%d, &(a)=%p, pa=%p, &(pa)=%p, *(pa)=%d\n",
                a,      &a,    pa,       &pa,       *pa);
    return 0;
}
```

Compilation : ok

Exécution :
Segmentation fault: 11

Propriétés des opérateurs & et *

Adresse et déréférencement :

- L'opérateur **&** s'applique à n'importe quel objet (variable, constante, fonction...) et se lit *"l'adresse de"* : elle retourne un pointeur (du type de l'objet)
- L'opérateur * ne s'applique qu'aux pointeurs et se lit *"l'élément pointé par"* : elle retourne une valeur (du type du pointeur)

Propriété

Si `var` est une variable on a : `*(&var)` est équivalent à `var`

Propriété

Si `pt` est une variable pointeur on a : `&(*pt)` est équivalent à `pt`

Plan

Fonctions avec pointeurs

Syntaxe

```
typeRes nomFonction(type1 *p, ...) {...}
```

Comme pour le passage de paramètre classique, c'est un pointeur copie du paramètre effectif qui est manipulé dans la fonction et non pas le paramètre effectif.
↪ toute modification du pointeur `p` reste locale à la fonction.

Cependant, grâce au déréférencement :

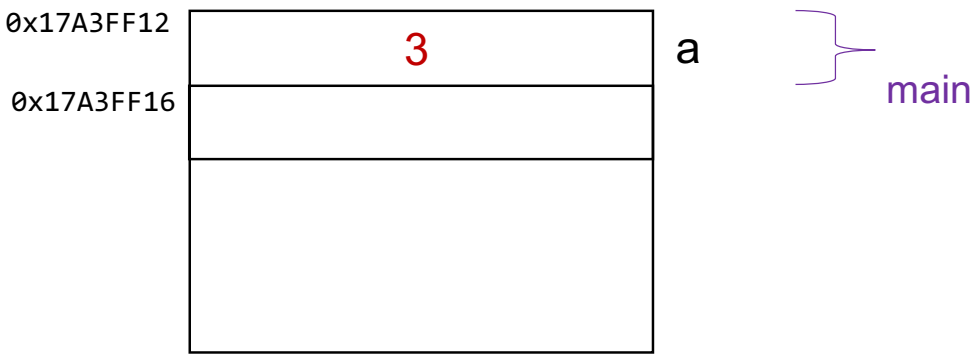
- on peut récupérer la donnée pointée : `*p`
- on peut modifier la valeur de la donnée pointée : `*ptr = ...`

Ainsi cela permet de simuler des paramètres d'Entrée/Sortie

Fonctions avec pointeurs : Exemple 1

```
1  #include <stdio.h>
2
3  void plusUn (int *p) {
4     *p = *p+1;
5  }
6
7  int main(){
8     int a;
9     a = 3;
10    plusUn(&a);
11    printf("a=%d\n", a);
12    return 0;
13 }
```

Évolution de la pile

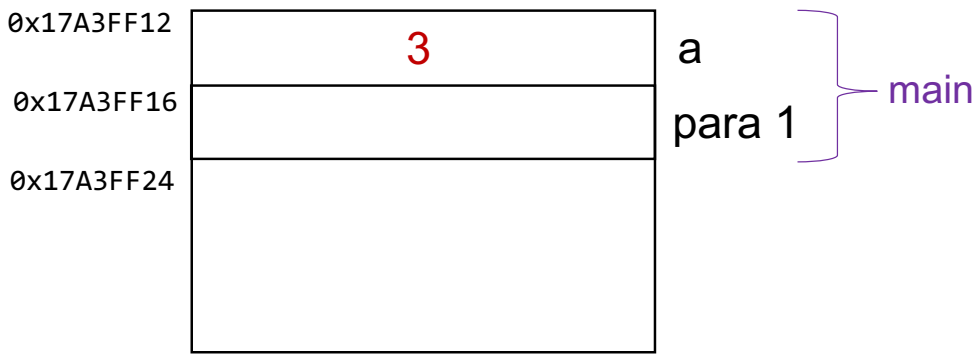


après la ligne 9 du main.

Fonctions avec pointeurs : Exemple

```
1  #include <stdio.h>
2
3  void plusUn (int *p) {
4     *p = *p+1;
5  }
6
7  int main(){
8     int a;
9     a = 3;
10    plusUn(&a);
11    printf("a=%d\n", a);
12    return 0;
13 }
```

Évolution de la pile

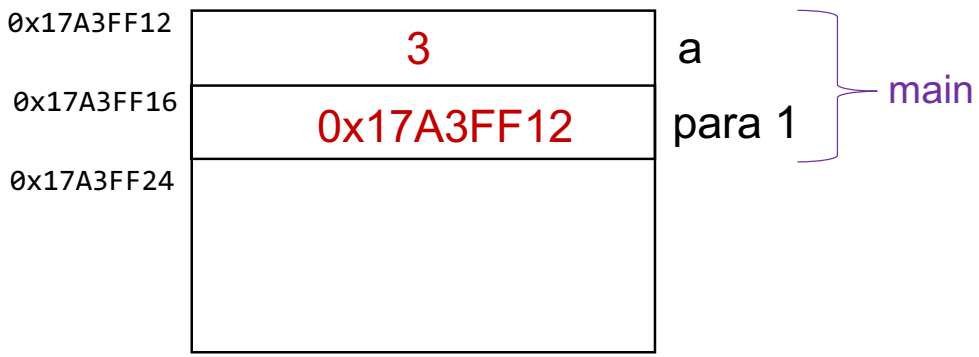


ligne 10 : préparation de l'appel à plusUn
allocation d'une zone pour &a

Fonctions avec pointeurs : Exemple

```
1  #include <stdio.h>
2
3  void plusUn (int *p) {
4     *p = *p+1;
5  }
6
7  int main(){
8     int a;
9     a = 3;
10    plusUn(&a);
11    printf("a=%d\n", a);
12    return 0;
13 }
```

Évolution de la pile

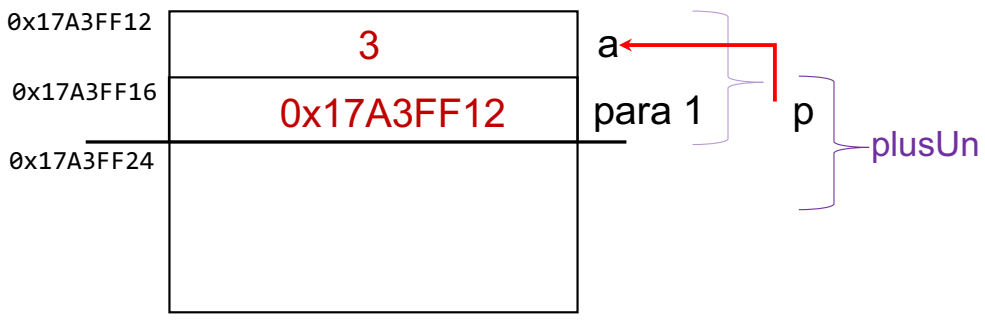


ligne 10 : préparation de l'appel à plusUn
copie de la valeur de &a

Fonctions avec pointeurs : Exemple

```
1  #include <stdio.h>
2
3  void plusUn (int *p) {
4     *p = *p+1;
5  }
6
7  int main(){
8     int a;
9     a = 3;
10    plusUn(&a);
11    printf("a=%d\n", a);
12    return 0;
13 }
```

Évolution de la pile



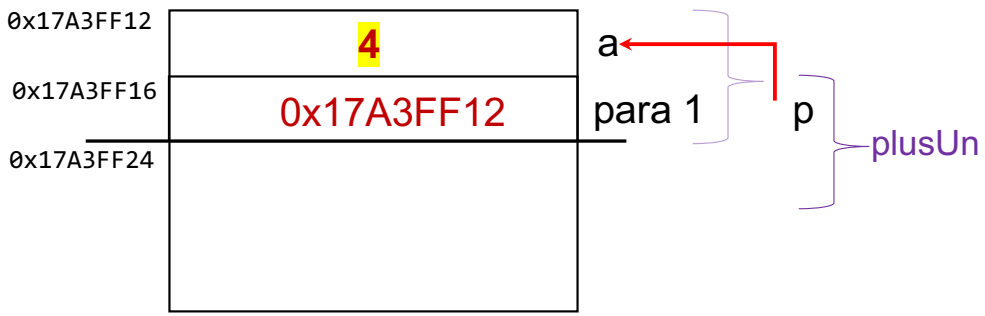
ligne 3 : appel à plusUn

le paramètre formel p pointe sur a

Fonctions avec pointeurs : Exemple

```
1  #include <stdio.h>
2
3  void plusUn (int *p) {
4     *p = *p+1;
5  }
6
7  int main(){
8     int a;
9     a = 3;
10    plusUn(&a);
11    printf("a=%d\n", a);
12    return 0;
13 }
```

Évolution de la pile

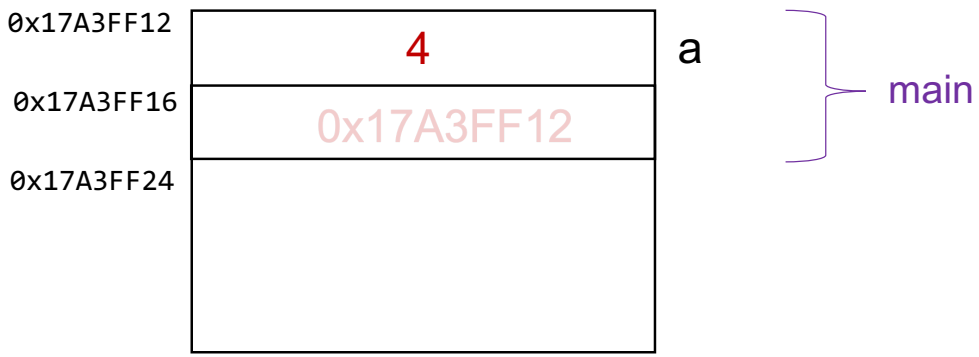


ligne 4 : affectation avec déréférencement
affectation de la donnée que p pointe
(et donc modification de a)

Fonctions avec pointeurs : Exemple

```
1  #include <stdio.h>
2
3  void plusUn (int *p) {
4     *p = *p+1;
5  }
6
7  int main(){
8     int a;
9     a = 3;
10    plusUn(&a);
11    printf("a=%d\n", a);
12    return 0;
13 }
```

Évolution de la pile



ligne 11 : après l'appel à plusUn
a est restée modifiée !

Fonctions avec paramètres d'Entrée/Sortie : Exemple 2

Implémentation C d'un algorithme d'échange de valeurs

```
Procédure ech(ES a : Nombre, ES b : Nombre)
// permute les valeurs de a et b
```

Fonctions avec paramètres d'Entrée/Sortie : Exemple 2

Implémentation C d'un algorithme d'échange de valeurs

```
Procédure ech(ES a : Nombre, ES b : Nombre)
// permute les valeurs de a et b
```

```c
#include <stdio.h>

void ech(int *a, int *b) {
   int r;
   r = *a;
   *a = *b;
   *b = r;
}

int main() {
   int x = 2, y = 3;
   ech(&x,&y);
   printf("x=%d y=%d\n",x,y);      ==> affiche  x=3 y=2
   return 0;
}
```

Fonction avec paramètres d'entrée et de sortie : Exemple 3

Implémentation C d'un algorithme de tri de 3 valeurs

```
Fonction mediane(E a : Nombre, E b : Nombre, E c : Nombre,
                 S min : Nombre, S max : Nombre) : Nombre
// retourne la médiane et affecte min et max selon les valeurs de a,b,c
```

Fonction avec paramètres d'entrée et de sortie : Exemple 3

Implémentation C d'un algorithme de tri de 3 valeurs

```
Fonction mediane(E a : Nombre, E b : Nombre, E c : Nombre,
                 S min : Nombre, S max : Nombre) : Nombre
// retourne la médiane et affecte min et max selon les valeurs de a,b,c
```

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

int mediane(int a, int b, int c, int *min, int *max) {
    *min = fmin(a,fmin(b,c));
    *max = fmax(a,fmax(b,c));
    return a + b + c - *min - *max;
}

int main() {
    int mini, maxi, medi;
    medi = mediane(65, 21, 42, &mini, &maxi);
    printf("mi=%d me=%d ma=%d\n",mini, medi, maxi);
    return 0;                    => affiche mi=21 me=42 ma=65
}
```