Programmation C et Structures de données (Prog 2)

Cours 2

Divers points techniques
Pointeurs et passage de paramètres

Pierre Fouilhoux & Christophe Tollu
pierre.fouilhoux@lipn.fr et ct@lipn.univ-paris13.fr

30 janvier 2024

1. Fonctions techniques utiles
   - Mesure du temps d'exécution
   - Conversion chaînes de caractères vers nombres
   - Nombres pseudo-aléatoires
   - Paramètres de la fonction principale
2. Pointeurs et adresses mémoire
   - Définitions
   - Arithmétique des pointeurs
3. Passage de paramètres et pointeurs
   - Non-passage par "copie"
   - Passage par "pointeurs"
4. Pointeurs de structures
5. Tableaux et pointeurs
   - Affectation de variables
   - Tableaux versus pointeurs

1. Fonctions techniques utiles
   - Mesure du temps d'exécution
   - Conversion chaînes de caractères vers nombres
   - Nombres pseudo-aléatoires
   - Paramètres de la fonction principale
2. Pointeurs et adresses mémoire
3. Passage de paramètres et pointeurs
4. Pointeurs de structures
5. Tableaux et pointeurs

Mesure du temps d'exécution

Dans la **bibliothèque <time.h>**

Deux fonctions très utiles

```
1 /** Renvoie le nombre de tours d'horloge du processeur depuis une origine */
2 /* (en general, le debut de l'execution du programme) */
3 clock_t clock(void);
4 /** Renvoie le temps ecoule (en secondes) depuis (00:00:00 UTC, January 1, 1970). */
5 /* Si secondes n'est pas NULL, la valeur renvoyee y est aussi stockee */
6 time_t time(time_t *secondes);
```

Un exemple d'utilisation

```
1 #include <time.h>
2
3 time_t t_debut, t_fin;
4 double duree_exec;
5 t_debut = clock();
6 /* appel d'une fonction dont on calcule la duree d'execution */
7 fibo_rec(40);
8 t_fin = clock();
9 duree_exec = (double) (t_fin - t_debut) / CLOCKS_PER_SEC;
```

1. Fonctions techniques utiles
   - Mesure du temps d'exécution
   - Conversion chaînes de caractères vers nombres
   - Nombres pseudo-aléatoires
   - Paramètres de la fonction principale
2. Pointeurs et adresses mémoire
3. Passage de paramètres et pointeurs
4. Pointeurs de structures
5. Tableaux et pointeurs

Fonctions de conversion

Dans la bibliothèque `<stdlib.h>`

Fonctions pour des nombres en base 10

```
1 /** Convertit la chaine s (expansion en base 10) en un int */
2 int atoi(const char * s);
3 /** Convertit la chaine s (expansion en base 10) en un long int */
4 long int atol(const char * s);
5 /** Convertit la chaine s en un double */
6 double atof(const char * s);
```

Fonctions pour des nombres en base b

```
1 /** Convertit la chaine s (expansion en base b) en un (long) int */
2 long int strtol(const char * s, char ** fptr, int b);
3 /** Convertit la chaine s (expansion en base b) en un unsigned (long) int */
4 unsigned long int strtoul(const char *s, char ** fptr, int b);
5 /** Convertit la chaine s en un double */
6 double strtod(const char *s, char ** fptr);
```

Rq : mettre fptr égal à NULL

1. Fonctions techniques utiles
   - Mesure du temps d'exécution
   - Conversion chaînes de caractères vers nombres
   - Nombres pseudo-aléatoires
   - Paramètres de la fonction principale
2. Pointeurs et adresses mémoire
3. Passage de paramètres et pointeurs
4. Pointeurs de structures
5. Tableaux et pointeurs

Générateur pseudo-aléatoire

Fonctions de la bibliothèque <stdlib.h>

```
1 /** Renvoie un nombre entier pseudo-aleatoire */
2 /* compris entre 0 et RAND_MAX */
3 int rand(void);
4 /** Initialise le generateur utilise par rand */
5 void srand(unsigned int germe);
```

Un exemple d'initialisation

```
1 #include <time.h>
2 int i, t[10];
3 srand(time(NULL));
4 /* On initialise chaque case du tableau avec un entier */
5 /* pseudo-aleatoire compris entre 0 et 9 */
6 for (i = 0; i < 10; ++i)
7   t[i] = rand()%10;
```

Générateur pseudo-aléatoire

Remarques d'utilisation

- Un appel à **srand(27)** (ou **srand(42)** etc), donnera la séquence de nombres initialisée par le nombre 27 (ou 42 etc).
- Sans appel préalable à **srand(time(NULL))**, le générateur `rand()` renvoie la même séquence de nombres.
  En initialisation avec la date (en secondes) de l'instant où vous lancez la commande, cela change la séquence observée.

Cas d'utilisation

- Pour écrire un programme, on peut aimer laisser la même séquence de nombre au départ.
  Mais pour tester un programme, on l'exécute avec différentes séquences aléatoires !
- Une astuce classique pour avoir des nombres entre 1 et m :
  rand() modulo m + 1

Nombres pseudo-aléatoires

Aléatoire vs pseudo-aléatoire

- Une suite de nombres produite par un algorithme déterministe **ne peut pas satisfaire tous les critères qualifiant les suites aléatoires**.
- En revanche, elle peut s'en approcher sur certains points : ≪ indépendance ≫ des nombres les uns par rapport aux autres, absence de régularité repérable, etc.
- Cela peut suffire pour de nombreuses applications

1. Fonctions techniques utiles
   - Mesure du temps d'exécution
   - Conversion chaînes de caractères vers nombres
   - Nombres pseudo-aléatoires
   - Paramètres de la fonction principale
2. Pointeurs et adresses mémoire
3. Passage de paramètres et pointeurs
4. Pointeurs de structures
5. Tableaux et pointeurs

Paramètres de la fonction principale

Environnement d'exécution

- En TP, l'exécution d'un programme C sera toujours **contrôlée par le système d'exploitation**, qui constitue l'environnement d'exécution du programme (du processus pour être plus précis).
- L'exécution commence alors par la fonction `main`, qui peut recevoir (ou non) en arguments des informations en provenance de l'environnement.

Prototype(s) de la fonction main

L'entête de la fonction `main` peut prendre une des formes suivantes :

```
1 int main (void) /* ou int  main () */
2 int main (int argc, char *argv[])
```

La seconde forme est indispensable si on doit récupérer des informations de l'environnement

Paramètres de la fonction principale

L'exécutable et ses arguments sur la ligne de commande

- Les informations transmises par l'environnement à la fonction `main` sont toujours des **chaînes de caractères** (séparées par des blancs sur la ligne de commande).

```
1 int main (int argc, char *argv[])
```

- Elles sont transmises sous la forme d'un **tableau de chaînes de caractères** :
  - `argv[0]` est la première chaîne de la ligne de commande (*i.e.* le nom du programme exécuté) ;
  - `argv[1]` est la deuxième chaîne de la ligne de commande (*i.e.* le premier argument de l'exécutable) ;
  - et ainsi de suite.
- `argc` est le nombre de chaînes distinctes de la ligne de commande.

Paramètres de la fonction principale

L'exécutable et ses arguments sur la ligne de commande

Les informations transmises par l'environnement à la fonction `main` sont toujours des **chaînes de caractères** (séparées par des blancs sur la ligne de commande).

- On doit tester si le nombre d'arguments, i.e. chaînes distinctes de la ligne de commande est bien celui attendu : `argc` .
- La fonction `main` doit le plus souvent les **convertir** (en entier, flottant, etc.) pour les manipuler.

Paramètres de la fonction principale

Un premier exemple

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void usage (char[]);
5  int fact (int);
6
7  int main (int argc, char *argv[]) {
8    if (argc != 2) {
9      printf("Erreur ! Nombre d'arguments invalide.\n");
0      usage(argv[0]);
1      return EXIT_FAILURE;
2    }
3    int n = strtol(argv[1], NULL, 10);
4    printf("%d ! = %d\n", n, fact(n));
5    return EXIT_SUCCESS;
6  }
7
8  void usage (char cmd[]) {
9    printf("Usage: %s int \n", cmd);
0  }
1
2  int fact (int n) {  /* corps de la fonction fact */
3  }
```

Paramètres de la fonction principale (2e exemple)

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

double puiss (double, unsigned);
double puiss_rap (double, unsigned);
void usage_exponentielle(char []);

int main (int argc, char ** argv) {
  if (argc != 3) {
    printf ("Erreur sur le nombre d'arguments !\n");
    usage_exponentielle(argv[0]);
    return EXIT_FAILURE;
  }
  else {
    double x = strtod (argv[1], NULL);
    unsigned n = strtoul (argv[2], NULL, 10);

    printf ("%lf puissance %u (rec.) = %lf \n", x, n, puiss (x, n));
    printf ("%lf puissance %u (rec. rapide) = %lf \n", x, n, puiss_rap (x, n));
    return EXIT_SUCCESS;
    }
}

void usage_exponentielle(char cmd[]) {
  printf("Usage: %s double int\n", cmd);
  printf("Argument 1 (double) : base de l'exponentiation \n");
  printf("Argument 2 (int) : l'exposant\n");
}
```

Paramètres de la fonction principale (2e exemple)

Exemple d'utilisation sans et avec erreur.

```
 > ./puiss_main 10 3
10.000000 puissance 3 (rec.) = 1000.0000
10.000000 puissance 3 (rec. rapide) = 1000.0000

> puiss_main 10 3 4
Erreur sur le nombre d'arguments !
Usage: ./puiss_main double int
Argument 1 (double) : base de l'exponentiation
Argument 2 (int) : l'exposant
```

1 Fonctions techniques utiles

2 Pointeurs et adresses mémoire
- Définitions
- Arithmétique des pointeurs

3 Passage de paramètres et pointeurs

4 Pointeurs de structures

5 Tableaux et pointeurs

1 Fonctions techniques utiles

2 Pointeurs et adresses mémoire
- Définitions
- Arithmétique des pointeurs

3 Passage de paramètres et pointeurs

4 Pointeurs de structures

5 Tableaux et pointeurs

Variables et adresse

Rappel

Une variable est caractérisée par :

- son nom
- son type
- sa valeur
- son adresse

Adresse d'une variable

- Soit une variable **int x;**
- Pour récupérer l'adresse de la variable **x**, on utilise &x
- Le signe &, appelé esperluette, désigne l'**opérateur d'adressage**.

Adresse

Adresse mémoire

- Une adresse mémoire est en fait l'accès aux données en mémoire.
- Il s'agit d'un nombre entier dont le format dépend du système : c'est la plupart du temps un nombre hexadécimal.
  pour l'afficher on utilise le format **%p** (pour pointeur) (le format hexadecimal peut aussi s'afficher avec **%x**)

```
1  int a = 3;
2  printf("Valeur de a: %d   |  Adresse de a : %p \n", a, &a);
```

peut afficher :

```
Valeur de a: 3   |  Adresse de a: x7ffcacb08ef4
```

Cette adresse est celle de la variable **a** lors de ce lancement : elle variera à chaque exécution.

En pratique

Il y a peu de situation où l'affichage d'une adresse est utile

Adresse

Adresse mémoire

- Une adresse mémoire est en fait l'accès aux données en mémoire.
- Il s'agit d'un nombre entier dont le format dépend du système : c'est la plupart du temps un nombre hexadécimal.
  pour l'afficher on utilise le format **%p** (pour pointeur) (le format hexadecimal peut aussi s'afficher avec **%x**)

```
1  int a = 3;
2  printf("Valeur de a: %d   |  Adresse de a : %p \n", a, &a);
```

peut afficher :

```
Valeur de a: 3   |  Adresse de a: x7ffcacb08ef4
```

Cette adresse est celle de la variable **a** lors de ce lancement : elle variera à chaque exécution.

En pratique

Il y a peu de situation où l'affichage d'une adresse est utile

Variables de type pointeur

Pointeur

Un **pointeur** est une variable dont **la valeur est une adresse mémoire**.

Un pointeur est donc un **contenant** dont le **contenu** est une adresse.

Pointeur sur variables

Si un pointeur contient l'adresse d'une variable, on dit qu'**il pointe** (sur, vers) cette variable.

```
1 double f;
2 f=3;  /* Affectation de la valeur 3.0 dans la variable f */
3 double* p; /* Declaration d'un pointeur sur une variable de type double */
4 ...
```

Variables de type pointeur

Pointeur

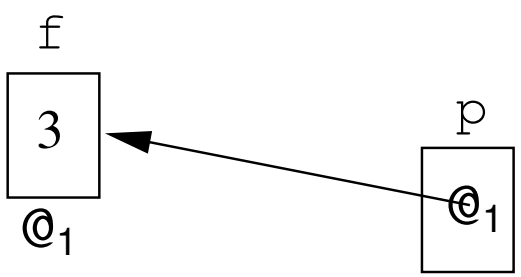
Un **pointeur** est une variable dont **la valeur est une adresse mémoire**.

Un pointeur est donc un **contenant** dont le **contenu** est une adresse.

Pointeur sur variables

Si un pointeur contient l'adresse d'une variable, on dit qu'**il pointe** (sur, vers) cette variable.

```
1 double f;
2 f=3;  /* Affectation de la valeur 3.0 dans la variable f */
3 double* p; /* Declaration d'un pointeur sur une variable de type double */
4 p=&f;   /* Affectation de l'adresse de la variable f dans p */
```



Type pointeur

Type

- Le **type d'un pointeur** est le type des variables sur lesquelles il peut pointer.
- On écrit ce type en utilisant une étoile * après le type pointé

Déclaration

```
int* p;
```

- Nom de variable : `p`
- Type : `int*` (pointeur sur un entier)

```
char* c;
```

- Nom de variable : `c`
- Type : `char*` (pointeur sur un caractère)

Type pointeur

Attention

- On peut indifférement écrire
  ```
  int* p;
  ```
  ou `int *p;`.
  ou `int * p;`.
- Mais si on veut cumuler deux déclarations sur une seule ligne, il faut remettre une étoile à chaque fois !
- Par exemple :
  ```
  int *p, *q; /* Déclare deux pointeurs sur entier p et q */
  int *p, i;  /* Déclare un pointeur p et un entier i */
  ```

Pointeurs

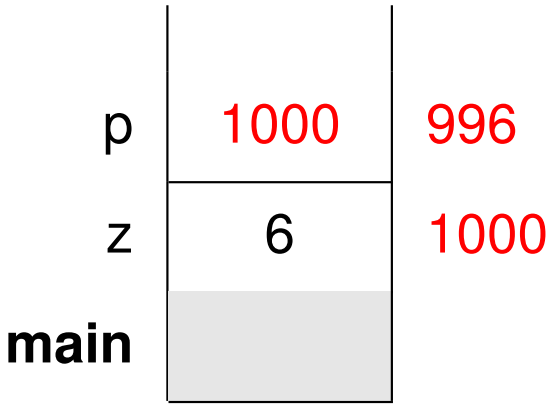
Déréférencement

- En stockant l'adresse d'autres variables,
  on va pouvoir **accéder à n'importe quelle zone de la mémoire**.
- Si `z` est un pointeur contenant une adresse
  L'opérateur étoile * permet d'accéder à l'adresse mémoire pointé par `z`
  On l'appelle l'opérateur de **déréférencement**.

Variables de type pointeur

On peut utiliser `*p` pour accéder au contenu de la variable pointée

```
1    int z = 6;
2    int *p;
3
4    p = &z;
5
6    printf(''%d'',*p);
```

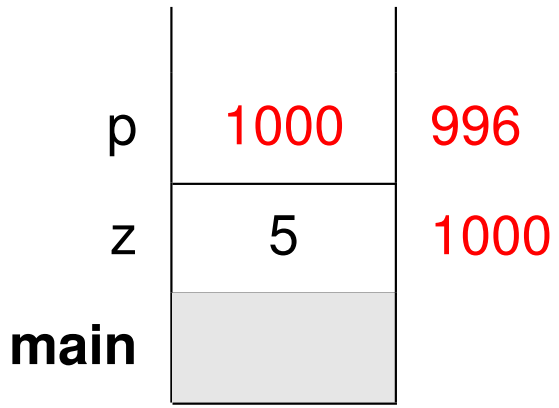


Le programme affiche 6

Variables de type pointeur

On peut utiliser `*p` pour modifier le contenu de la variable pointée

```
1    int z =6;
2    int *p;
3
4    p = &z;
5
6    *p=5;
7
8    printf(''%d'',*p);
```



Le programme affiche 5

Remarque inutile

Going nowhere

Cumuler * puis & revient à utiliser la variable elle-même !

```
1    int z =6;
2
3    printf(``%d'',*&z);
```

Initialisation d'un pointeur

Initialisation

- Comme toute variable, une variable de type pointeur **doit être initialisée**.
- Si un pointeur n'est pas initialisé, il peut contenir une adresse quelconque et modifier par erreur la case mémoire correspondante peut avoir de graves conséquences.

Ne pointer vers rien

- Soit un pointeur `p`. Pour que `p` ne pointe vers aucune case mémoire, on écrit :

```
p=NULL;
```

- Cette instruction est valable **quel que soit le type du pointeur**.

1 Fonctions techniques utiles

2 Pointeurs et adresses mémoire
- Définitions
- Arithmétique des pointeurs

3 Passage de paramètres et pointeurs

4 Pointeurs de structures

5 Tableaux et pointeurs

Arithmétique des pointeurs

Incrémenter un pointeur

Soit un `p` une variable de type pointeur. Lorsque l'on écrit `p+1`, cela signifie « l'adresse contenue dans `p` » + « la taille (en nombre d'octets) d'un objet du type pointé par `p` »

Exemple

On suppose que `p` est un pointeur dont la valeur est (l'adresse) 1200

- Si `p` pointe vers un `int`, alors la valeur de `p+1` est 1204
- Si `p` pointe vers un `char`, alors la valeur de `p+1` est 1201
- Si `p` pointe vers un `double`, alors la valeur de `p+1` est 1208

Arithmétique des pointeurs

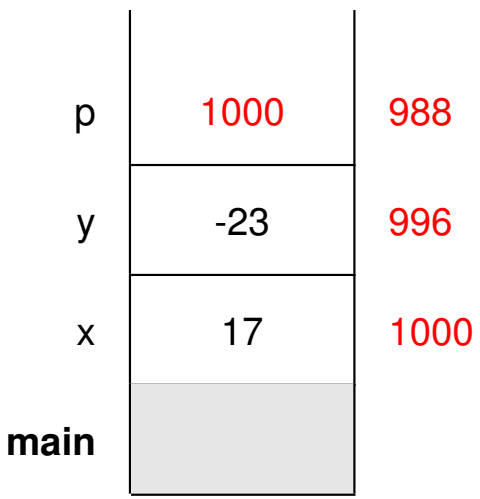
```
1  #include <stdio.h>
2  #include <stdlib.h>
3  /* programme compile et execute sur mon Mac */
4  int main(){
5    int x=17;
6    int y=-23;
7    int *p=&x;
8    printf("%d\n",*p);
9    p=p-1; /* on ecrit p=p-1 car les adresses decroissent quand on empile */
0    printf("%d\n",*p);
1    return EXIT_SUCCESS;
2  }
```


Ligne 8 : Affiche 17

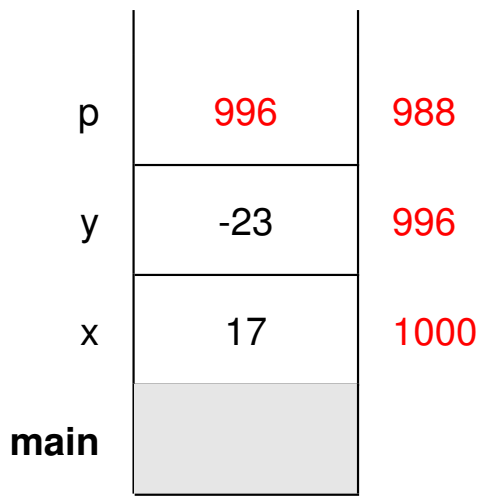
Ligne 10 : Affiche -23

Arithmétique des pointeurs

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  /* programme compile et execute sur mon Mac */
4  int main(){
5     int x=17;
6     int y=-23;
7     int *p=&x;
8     printf("%d\n",*p);
9     p=p-1; /* on ecrit p=p-1 car les adresses decroissent quand on empile */
0     printf("%d\n",*p);
1     return EXIT_SUCCESS;
2  }
```



Ligne 8 : Affiche 17



Ligne 10 : Affiche -23

Arithmétique des pointeurs

Attention au types pointés

Se déplacer par +1 ou -1 déplace du nombre d'octet du type pointé.
Ainsi il ne déplace pas à la variable suivante si elle est de type différent ! ! !

Arithmétique des pointeurs

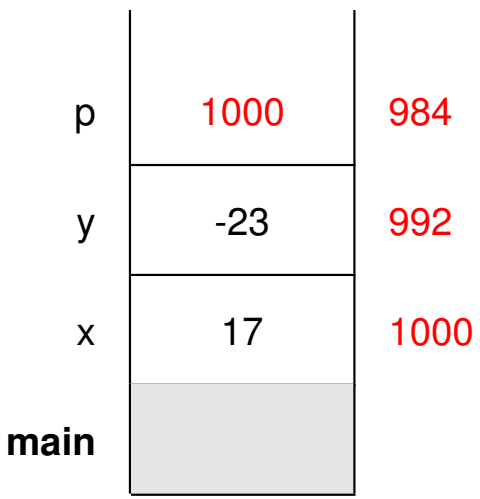
```
1  #include <stdio.h>
2  #include <stdlib.h>
3  /* programme compile et execute sur mon Mac */
4  int main(){
5    int x=17;
6    long y=-23;
7    int *p=&x;
8    printf("%d\n",*p);
9    p=p-1; /* on ecrit p=p-1 car les adresses decroissent quand on empile */
0    printf("%d\n",*p);
1    return EXIT_SUCCESS;
2  }
```


Ligne 8 : Affiche 17

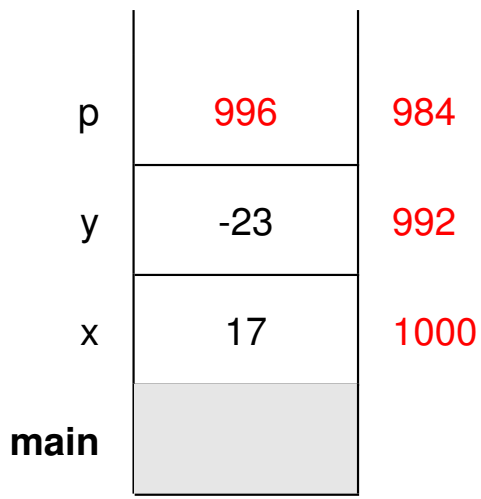
Ligne 10 : Affiche -1

Arithmétique des pointeurs

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  /* programme compile et execute sur mon Mac */
4  int main(){
5    int x=17;
6    long y=-23;
7    int *p=&x;
8    printf("%d\n",*p);
9    p=p-1; /* on ecrit p=p-1 car les adresses decroissent quand on empile */
0    printf("%d\n",*p);
1    return EXIT_SUCCESS;
2  }
```



Ligne 8 : Affiche 17



Ligne 10 : Affiche -1

1 Fonctions techniques utiles

2 Pointeurs et adresses mémoire

3 Passage de paramètres et pointeurs
- Non-passage par “copie”
- Passage par “pointeurs”

4 Pointeurs de structures

5 Tableaux et pointeurs

Les limites de la pile d'appel

Les limites d'accès des variables locales

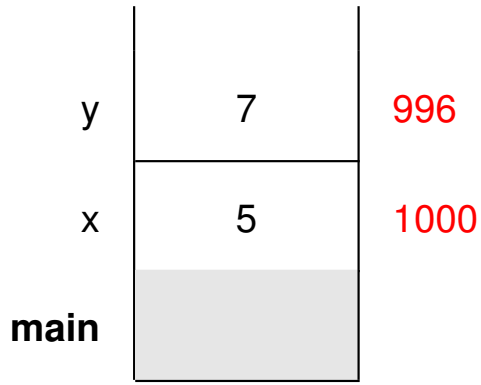
Avec nos connaissances actuelles :

- Il n'est pas possible de modifier une variable locale d'une fonction *A* avec une fonction *B*.
- Il y a une « exception » à cette règle : on peut mettre à jour la valeur d'une variable locale d'une fonction *A* par le **résultat** de la fonction *B*.

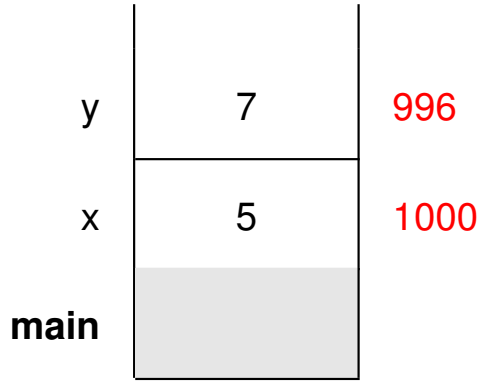
Exemple

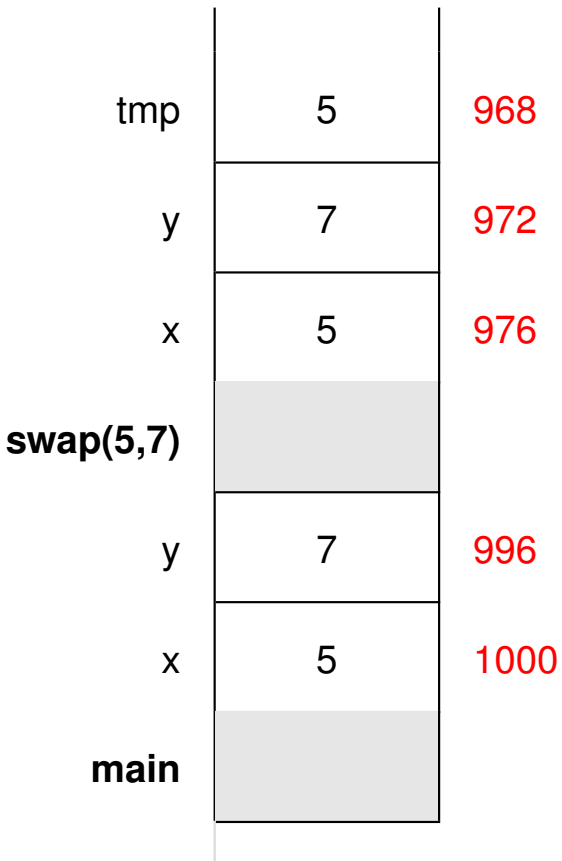
Essayons d'écrire une fonction qui échange la valeur de deux variables.

```
1  void swap(int x,int y){
2    int tmp=x;
3    x=y;
4    y=tmp;
5  }
6  int main(){
7    int x=5;
8    int y=7;
9    swap(x,y);
0    return EXIT_SUCCESS;
1  }
```

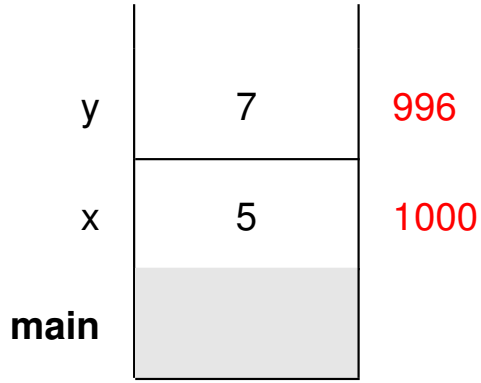


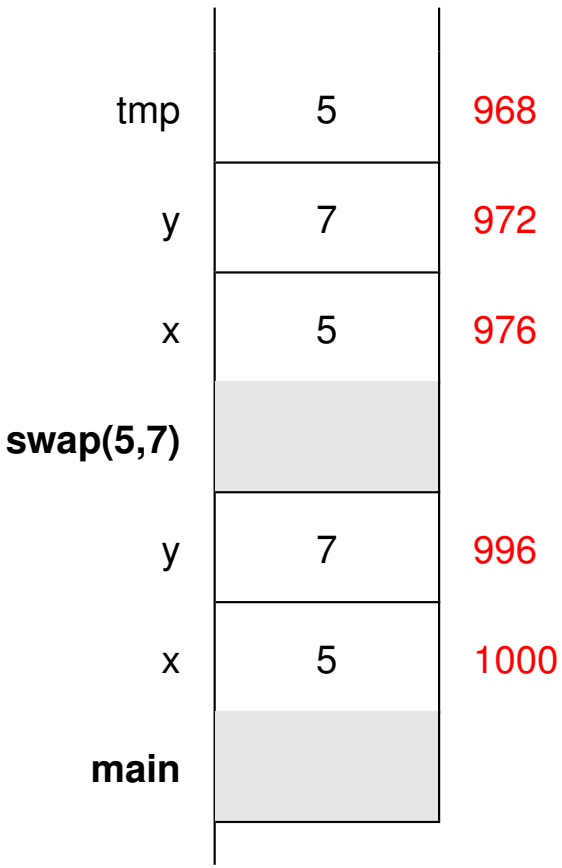
```
1  void swap(int x,int y){
2    int tmp=x;
3    x=y;
4    y=tmp;
5  }
6  int main(){
7    int x=5;
8    int y=7;
9    swap(x,y);
0    return EXIT_SUCCESS;
1  }
```

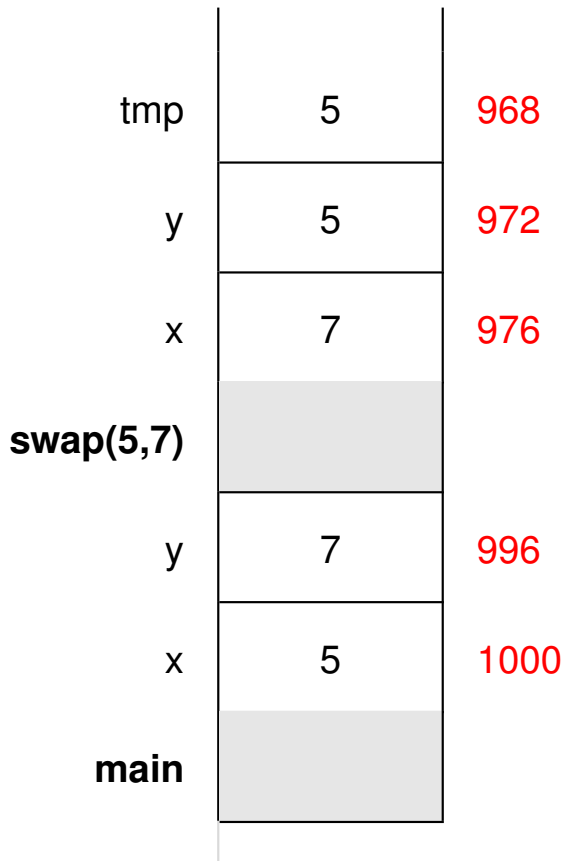




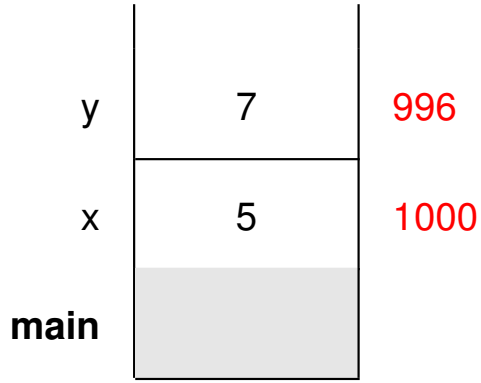
```
1  void swap(int x,int y){
2    int tmp=x;
3    x=y;
4    y=tmp;
5  }
6  int main(){
7    int x=5;
8    int y=7;
9    swap(x,y);
0    return EXIT_SUCCESS;
1  }
```

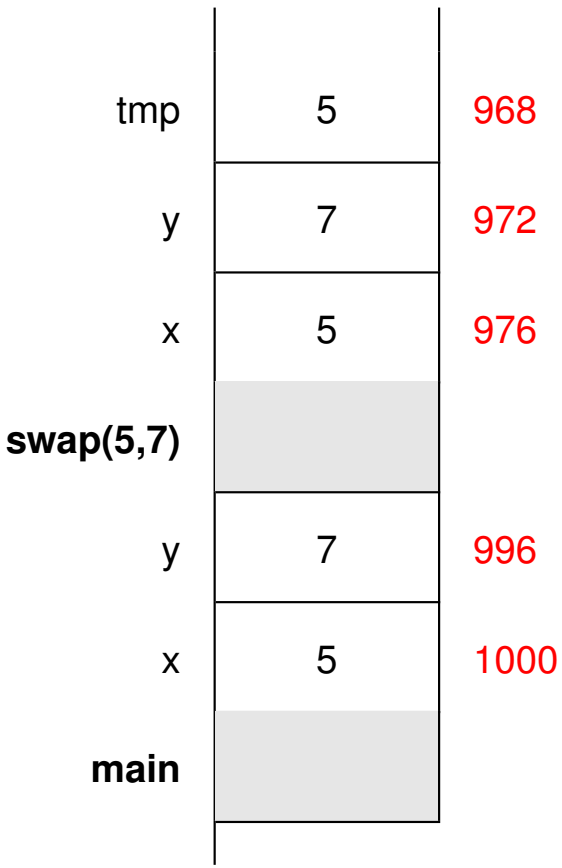


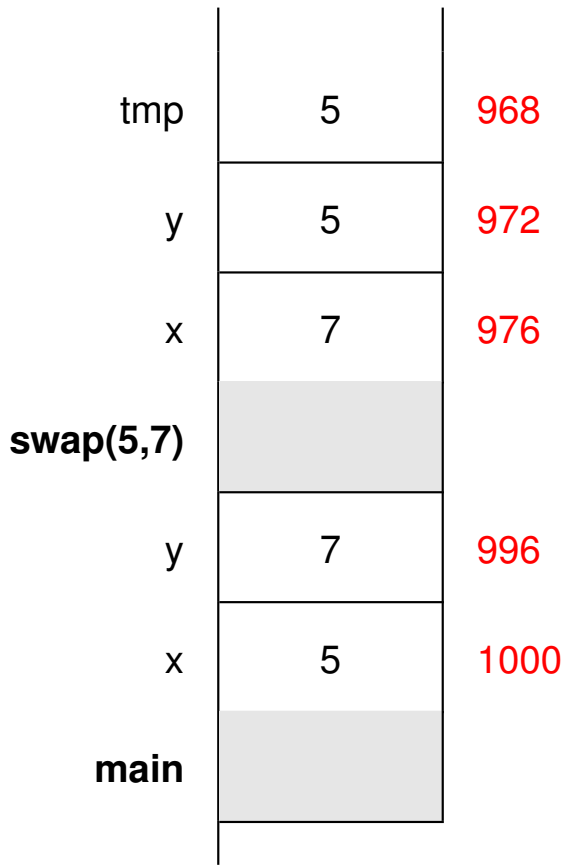


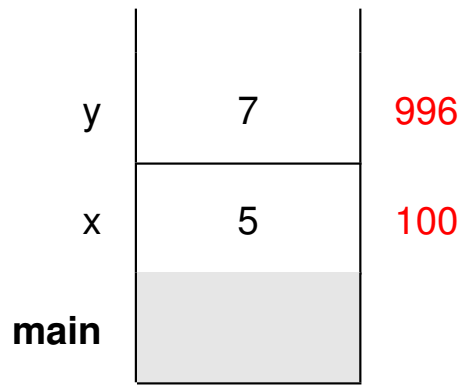


```
1  void swap(int x,int y){
2    int tmp=x;
3    x=y;
4    y=tmp;
5  }
6  int main(){
7    int x=5;
8    int y=7;
9    swap(x,y);
0    return EXIT_SUCCESS;
1  }
```









Constat

- Notre accès à la mémoire est pour l'instant très limité.
- Une fonction **ne peut accéder qu'à ses propres variables locales et paramètres formels ou à des variables de la zone de données**.

1 Fonctions techniques utiles

2 Pointeurs et adresses mémoire

3 Passage de paramètres et pointeurs
- Non-passage par “copie”
- Passage par “pointeurs”

4 Pointeurs de structures

5 Tableaux et pointeurs

Pointeurs et fonction

Échange

Remarquer que la fonction `scanf` prend en paramètre
non pas la valeur contenue dans une variable
mais l'adresse de la variable par l'opérateur &
Ainsi `scanf` permet de modifier le contenu pointée par l'adresse qu'elle reçoit

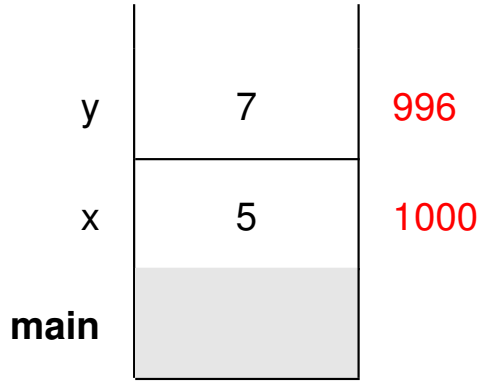
Pointeurs et fonction swap

Échange

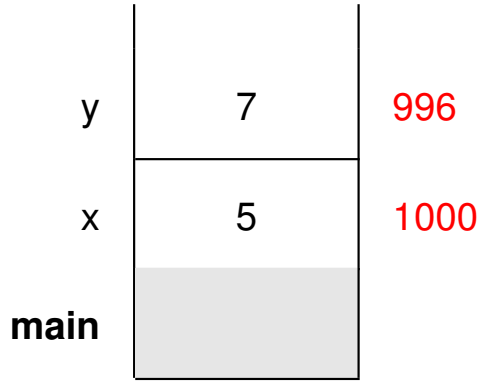
On peut à présent réécrire la fonction `swap`
en utilisant les pointeurs !

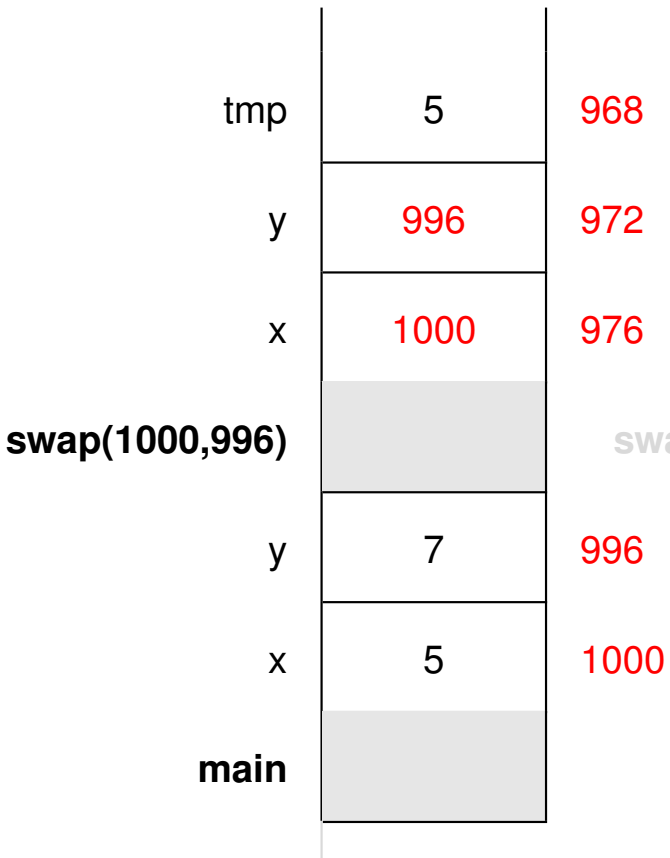
```
1  void swap(int * x,int * y){
2    int tmp=*x;
3    *x=*y;
4    *y=tmp;
5  }
6  int main(){
7    int x=5;
8    int y=7;
9    swap(&x,&y);
0    return EXIT_SUCCESS;
1  }
```

```
1  void swap(int * x,int * y){
2    int tmp=*x;
3    *x=*y;
4    *y=tmp;
5  }
6  int main(){
7    int x=5;
8    int y=7;
9    swap(&x,&y);
0    return EXIT_SUCCESS;
1  }
```

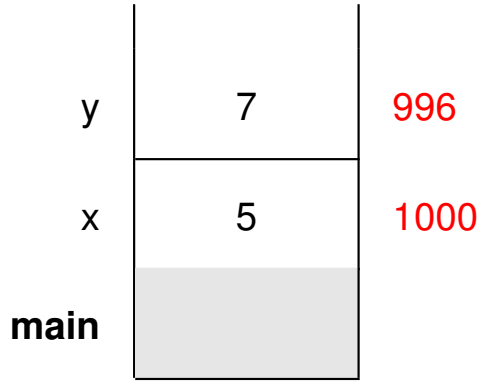


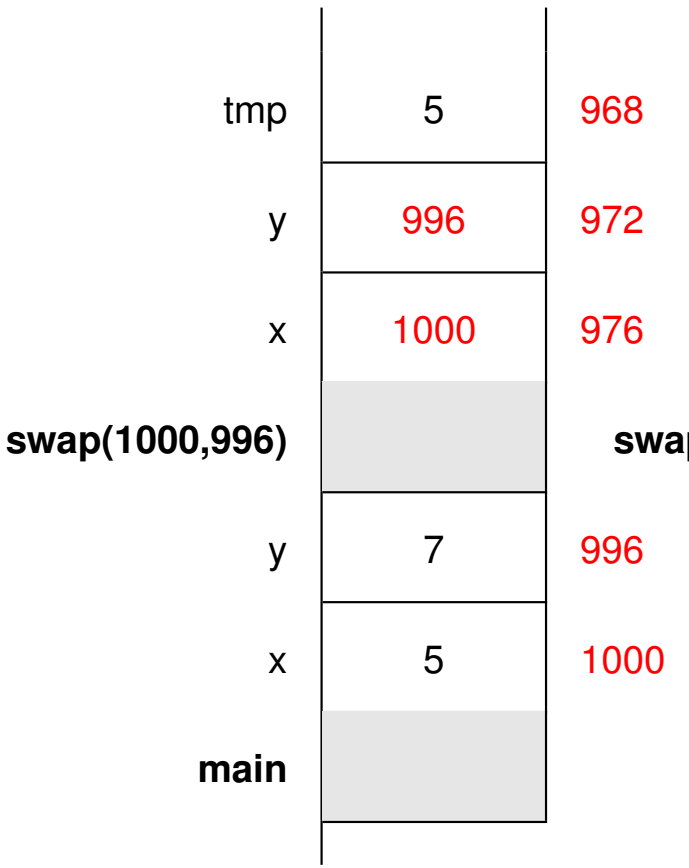
```c
void swap(int * x,int * y){
   int tmp=*x;
   *x=*y;
   *y=tmp;
}
int main(){
   int x=5;
   int y=7;
   swap(&x,&y);
   return EXIT_SUCCESS;
}
```

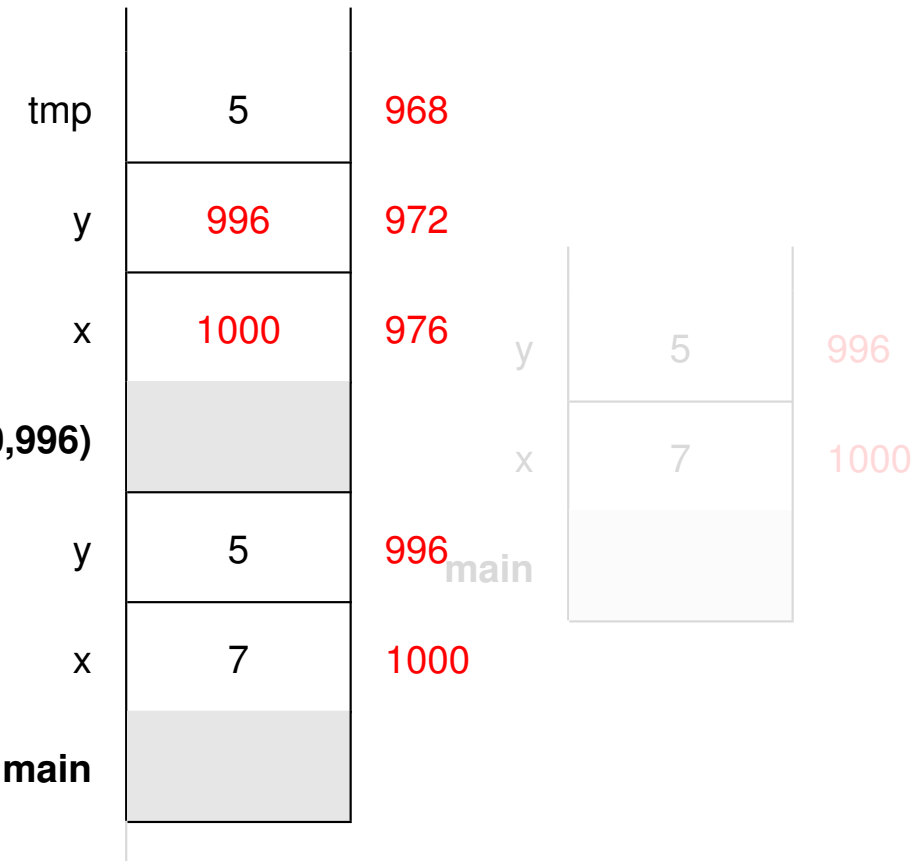




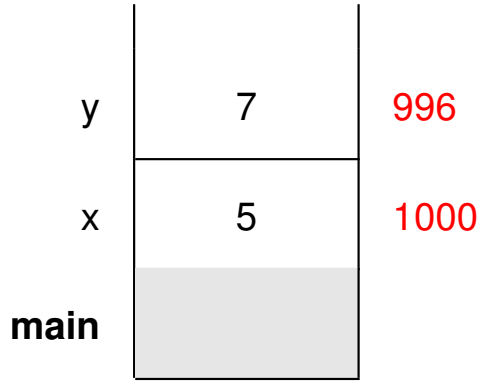
```c
1  void swap(int * x,int * y){
2    int tmp=*x;
3    *x=*y;
4    *y=tmp;
5  }
6  int main(){
7    int x=5;
8    int y=7;
9    swap(&x,&y);
0    return EXIT_SUCCESS;
1  }
```

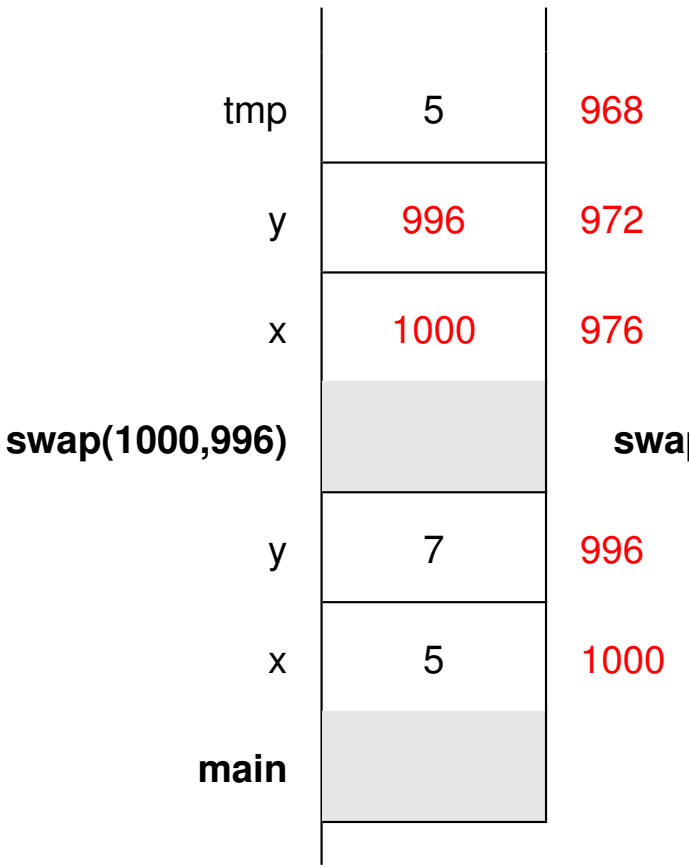


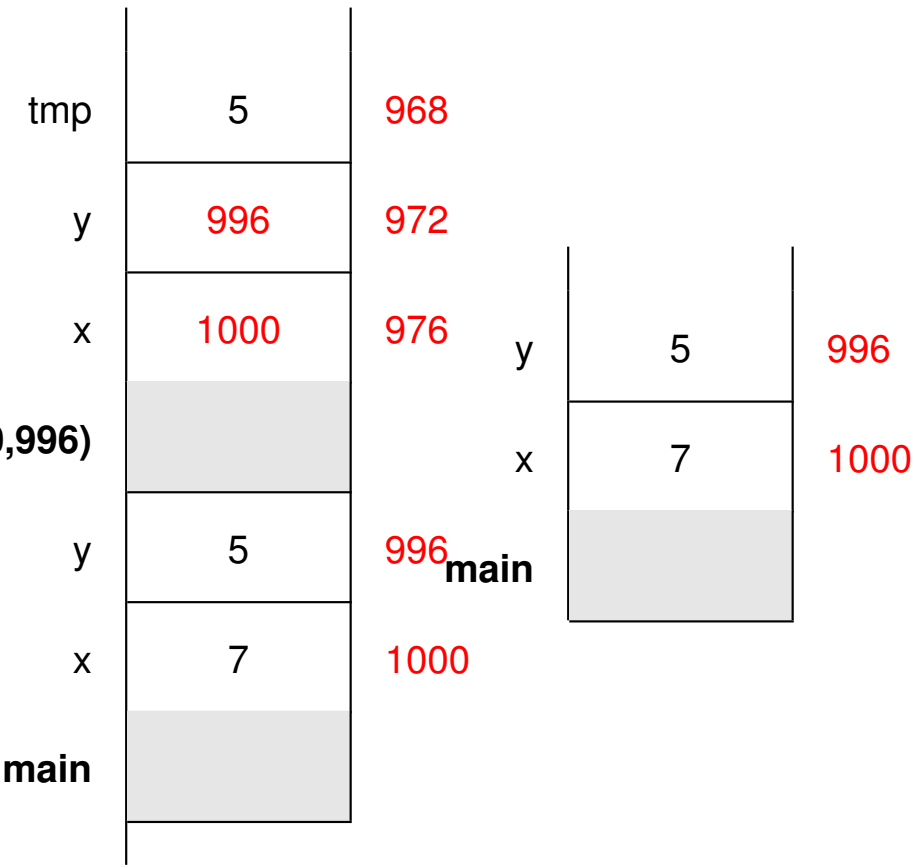




```c
1  void swap(int * x,int * y){
2    int tmp=*x;
3    *x=*y;
4    *y=tmp;
5  }
6  int main(){
7    int x=5;
8    int y=7;
9    swap(&x,&y);
0    return EXIT_SUCCESS;
1  }
```







1 Fonctions techniques utiles

2 Pointeurs et adresses mémoire

3 Passage de paramètres et pointeurs

4 Pointeurs de structures

5 Tableaux et pointeurs

Pointeurs de structures

Pointeurs de structures

Un pointeur peut contenir l'adresse d'un objet de type quelconque, donc en particulier celle d'une structure.

Pointeurs sur `point`

```
1 /* point.h */
2 struct point_s{
3   float x;
4   float y;
5 };
6 typedef struct point_s point;
```

Déclaration du pointeur : `point * p;`

Pointeurs de structures

```
1 /* point.h */
2 struct point_s{
3   float x;
4   float y;
5 };
6 typedef struct point_s point;
```

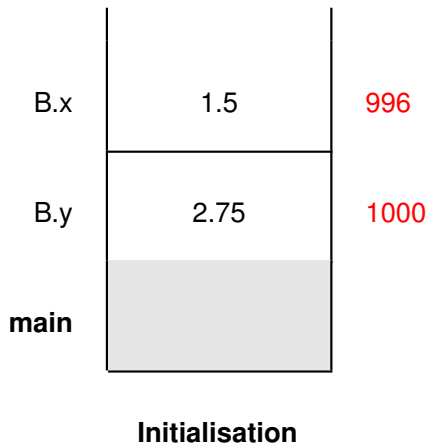
Accès aux champs de la structure

Soit le pointeur `point * p;`
Deux moyens d'accéder au champ `x` de `p` :

- `(*p).x`
- `p->x`

Pointeurs de structures

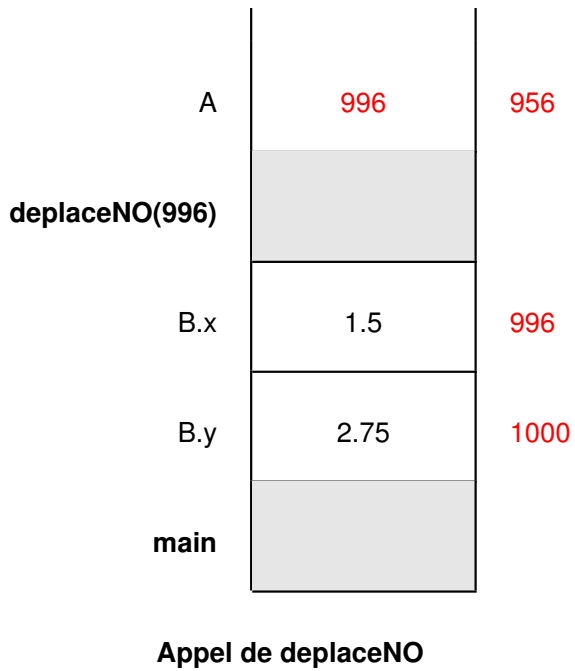
```
1  void deplaceNO(point * A){
2    A->x = A->x+1;
3    A->y = A->y+1;
4  }
5
6  int main(){
7
8    point B = {1.5, 2.75};
9
0    deplace(&B);
1
2    return EXIT_SUCCESS;
3  }
```



Initialisation

Pointeurs de structures

```
1  void deplaceNO(point * A){
2    A->x = A->x+1;
3    A->y = A->y+1;
4  }
5
6  int main(){
7
8    point B = {1.5, 2.75};
9
0    deplace(&B);
1
2    return EXIT_SUCCESS;
3  }
```



Appel de deplaceNO

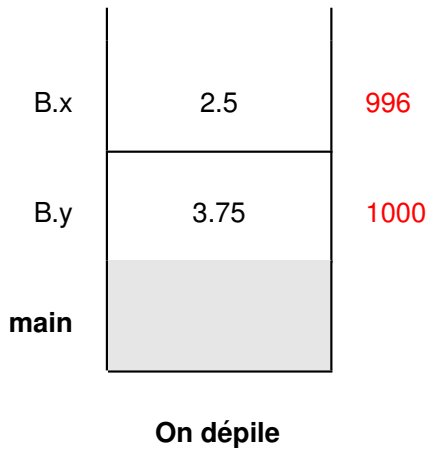
Pointeurs de structures

```
1  void deplaceNO(point * A){
2    A->x = A->x+1;
3    A->y = A->y+1;
4  }
5
6  int main(){
7
8    point B = {1.5, 2.75};
9
0    deplace(&B);
1
2    return EXIT_SUCCESS;
3  }
```

Exécution de deplaceNO

Pointeurs de structures

```
1  void deplaceNO(point * A){
2    A->x = A->x+1;
3    A->y = A->y+1;
4  }
5
6  int main(){
7
8    point B = {1.5, 2.75};
9
0    deplace(&B);
1
2    return EXIT_SUCCESS;
3  }
```



On dépile

1 Fonctions techniques utiles

2 Pointeurs et adresses mémoire

3 Passage de paramètres et pointeurs

4 Pointeurs de structures

5 Tableaux et pointeurs
- Affectation de variables
- Tableaux versus pointeurs

Affectations de variables

Affectations

L'affectation entre deux expressions s'écrit :

var = expr;

donne des rôles tout à fait différents au deux expressions

- A gauche **var** doit être une variable qui soit une **lvalue** (left-hand value) : c'est-à-dire une variable pouvant être utilisée comme contenant d'une valeur.
- A droite **expr** est une expression pouvant être évaluée (appelé parfois rvalue) : la valeur résultant de l'évaluation est alors affectée à la variable de la gauche de l'affectation.

Remarque : en "pseudo-code", on note cela

expr1 $\leftarrow$ **expr2;**

Exemple d'expression qui n'est pas une lvalue : un appel de fonction.

Affectation entre variables

L'affectation entre deux variables :

var1 = var2;

est ainsi le fait de mettre la valeur de **var2** dans la variable **var1**.

Affectation entre variables

- Si **var1** est de type simple : c'est une opération d'affectation
- Si **var1** est une case d'un tableau. e.g. `tab[2]=3` : c'est une opération d'affectation d'une variable simple :
- Si **var1** est d'un type struct : c'est autant d'opérations d'affectation que de champs dans le type struct

Cas des tableaux

Un tableau n'est pas une lvalue

- Si **var1** est un tableau : c'est une opération illicite, car un tableau n'est pas une lvalue

```
1 int tab1[10];
2 tab1++;
```

Affichage : `error: lvalue required as increment operand`

Copie d'un tableau

Pour copier un tableau dans un autre, il faut recopier chaque case une à une : "au prix" d'autant d'affectations qu'il y a de cases

```
1  for (i=0;i<taille;i++)
2     tab2[i]=tab1[i];
```

1 Fonctions techniques utiles

2 Pointeurs et adresses mémoire

3 Passage de paramètres et pointeurs

4 Pointeurs de structures

5 Tableaux et pointeurs
- Affectation de variables
- Tableaux versus pointeurs

Tableaux et pointeurs

Une variable tableau est “presque” un pointeur

- La déclaration `double T[5];` déclare une variable `T` de type `double *` donc `T` peut être vu comme un pointeur.
- La variable `T` contient l’adresse de la première case du tableau.

Tableau en paramètre d’une fonction

- Les deux écritures suivantes sont équivalentes pour les paramètres formels d’une fonction :

```
void init_tab(int tab[], int n)
```

et

```
void init_tab(int * tab, int n)
```

- Ainsi, envoyer la valeur d’un tableau en paramètre effectif d’une fonction revient à envoyer une adresse mémoire : cela explique pourquoi on peut modifier un tableau passé en paramètre effectif.

Tableaux/pointeurs en arguments de fonctions

```
1  void init_tab(int tab[], int n){
2    int i;
3    for(i=0;i<n;i++)
4      tab[i]=0;
5    }
```

```
int main(){
  int tab[3];
  init_tab(tab,3);
  return EXIT_SUCCESS;
}
```


Au début de l'appel

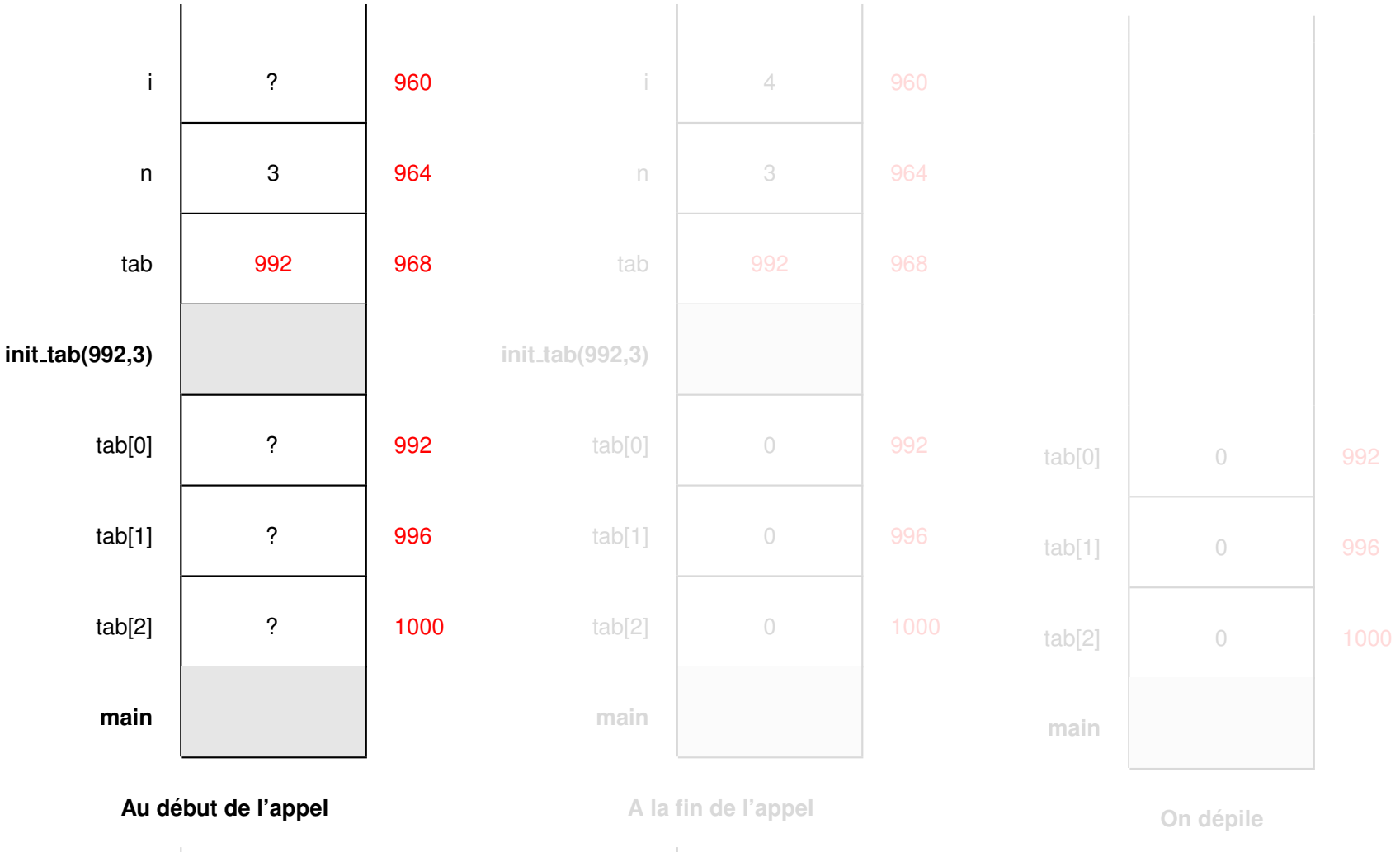
A la fin de l'appel

On dépile

Tableaux/pointeurs en arguments de fonctions

```
1  void init_tab(int tab[], int n){
2    int i;
3    for(i=0;i<n;i++)
4      tab[i]=0;
5    }
```

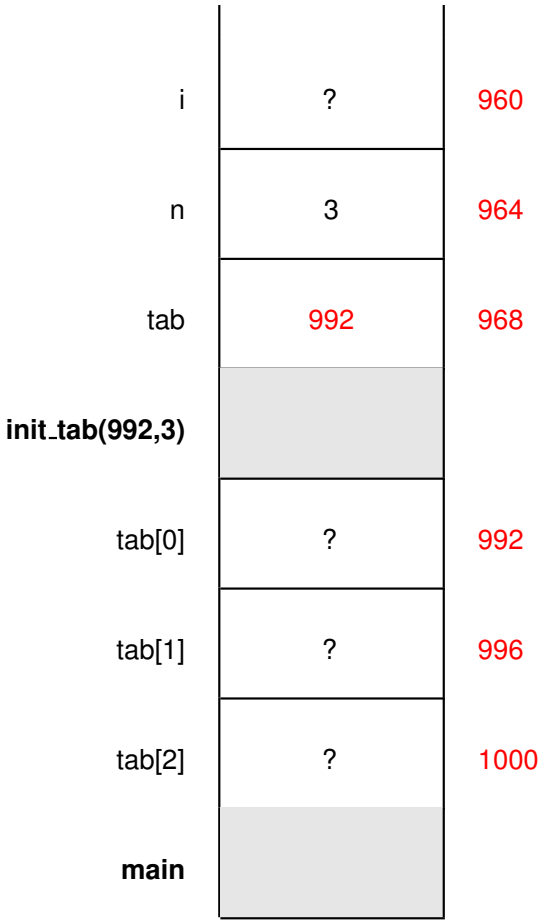
```
int main(){
  int tab[3];
  init_tab(tab,3);
  return EXIT_SUCCESS;
}
```



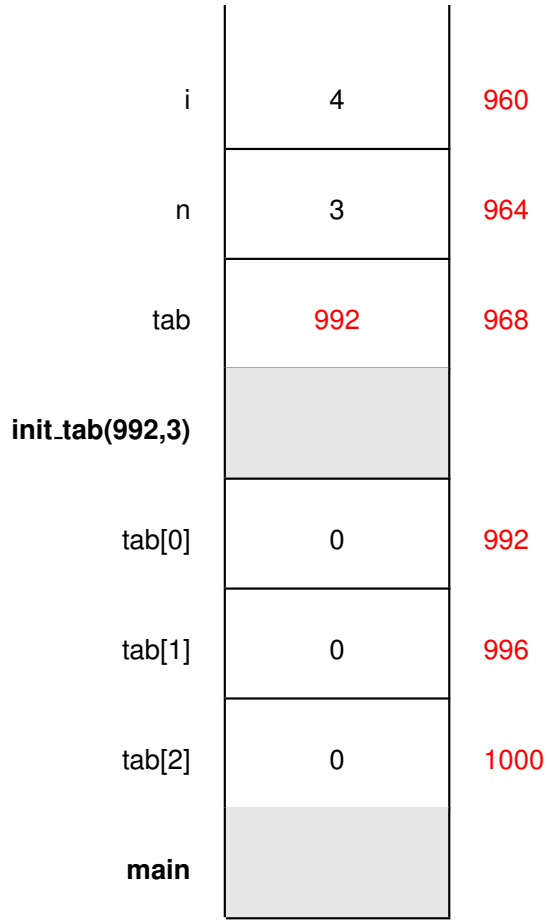
Tableaux/pointeurs en arguments de fonctions

```
1  void init_tab(int tab[], int n){
2    int i;
3    for(i=0;i<n;i++)
4      tab[i]=0;
5    }
```

```
int main(){
  int tab[3];
  init_tab(tab,3);
  return EXIT_SUCCESS;
}
```



Au début de l'appel



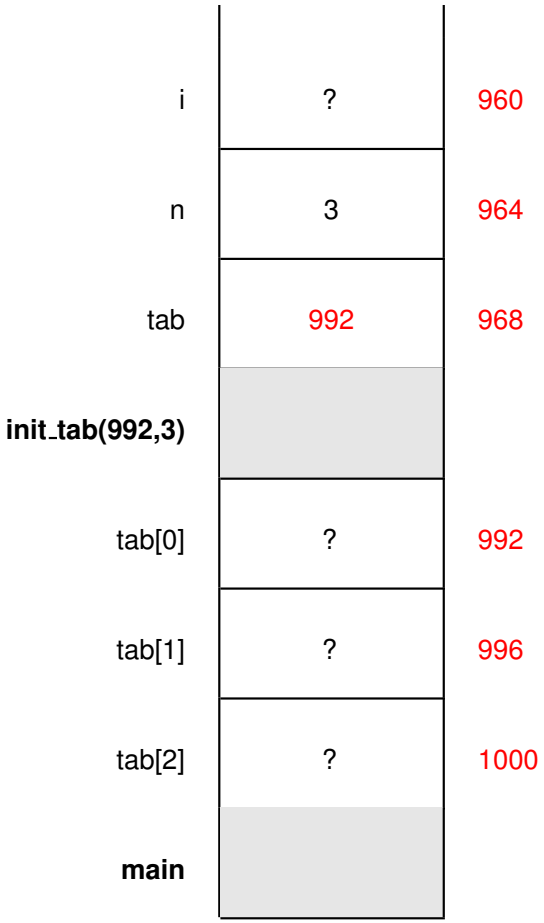
A la fin de l'appel

On dépile

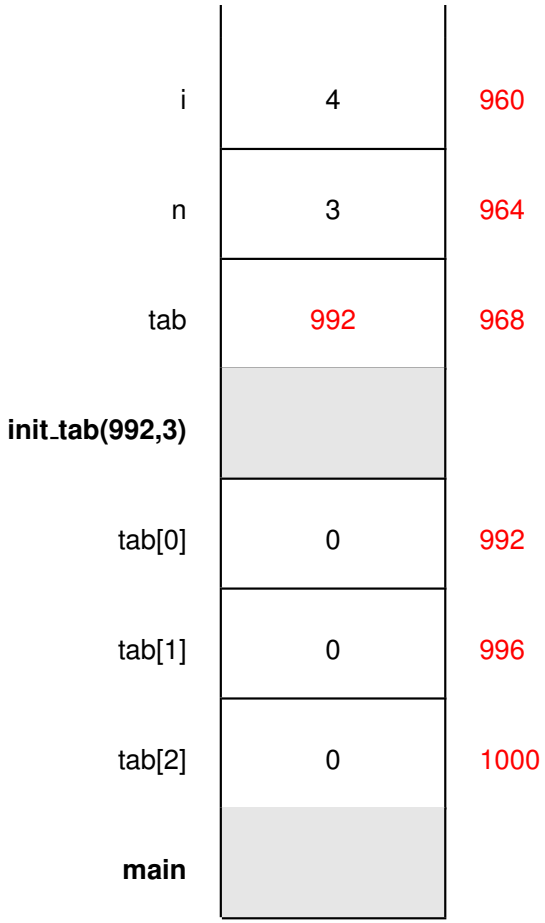
Tableaux/pointeurs en arguments de fonctions

```
1  void init_tab(int tab[], int n){
2    int i;
3    for(i=0;i<n;i++)
4      tab[i]=0;
5    }
```

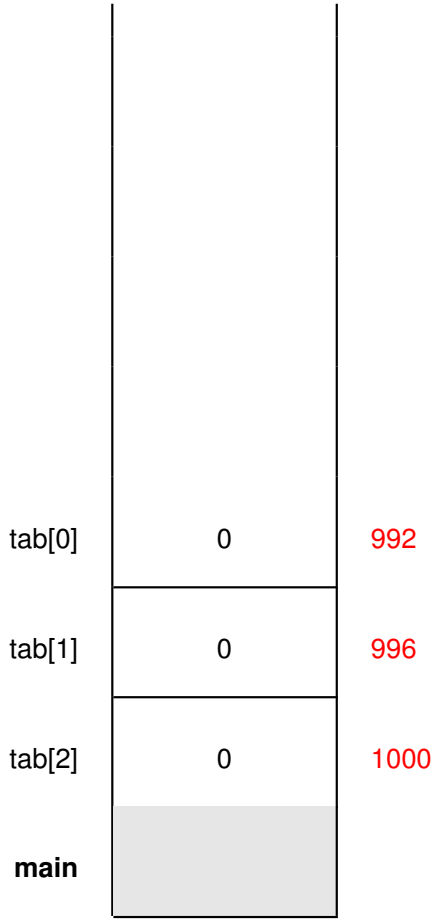
```
int main(){
  int tab[3];
  init_tab(tab,3);
  return EXIT_SUCCESS;
}
```



Au début de l'appel



A la fin de l'appel



On dépile

Tableaux et pointeurs

Une variable tableau “n’est pas la même chose qu’un pointeur

- Un nom de tableau **n’est pas** une *lvalue* : il ne peut pas apparaître comme membre gauche d’une affectation !
- En particulier, déclarer `int tab[10];` rend l’instruction `tab++;` **illégale** ! (Message d’**erreur à la compilation**)

Accès aux clés d'un tableau

Accès à la clé d'indice *i*

- Lorsque l'on accède à la case d'indice *i* d'un tableau `tab`

  ```
  tab[i]
  ```

  On effectue en réalité la commande suivante :

  ```
  *(tab+i)
  ```

- C'est-à-dire qu'accéder à la case *i* d'un tableau, c'est aller à la variable dont l'addresse est `i` cases plus loin que la première case de `tab`

Pointeurs/tableaux

Qu'affiche ce programme ?

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

int main(){
  int table[3] = {4,2,3};
  int i;
  for(i=0; i<3; i++){
    printf("%5d", *(table+i));
  }
  printf("\n");
  for(i=0; i<3; i++){
    printf("%5d", table[i]);
  }
  printf("\n");
  return EXIT_SUCCESS;
}
```

Affichage :

```
    4    2    3
    4    2    3
```

Pointeurs/tableaux

Qu'affiche ce programme ?

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

int main(){
  int table[3] = {4,2,3};
  int i;
  for(i=0; i<3; i++){
    printf("%5d", *(table+i));
  }
  printf("\n");
  for(i=0; i<3; i++){
    printf("%5d", table[i]);
  }
  printf("\n");
  return EXIT_SUCCESS;
}
```

Affichage :

```
    4    2    3
    4    2    3
```

Pointeurs/tableaux

Astuce de programmation

- Si `tab` est un tableau de n cases
  et si i est un entier entre 0 et $n-1$
  Alors `tab + i` est le sous-tableau de `tab` limité au cases de i à $n-1$
- Cette astuce permet de simplifier l'écriture de fonctions

Passage de tableaux en arguments : tri par sélection

L'idée de l'algorithme

On part d'un tableau non trié de taille n.

1. on cherche le plus petit élément du tableau (*min*),
2. on échange le contenu de la première case du tableau et celui de la case contenant *min*,
3. on recommence avec le tableau « raccourci » de sa première clé (qui est bien placée).

Passage de tableaux en arguments : tri par sélection

```c
/** Fonction qui renvoie l'indice de la plus petite clef du tableau tab */
int recherche_pos_min(int * tab, int taille) {
  int i;
  int pos_min = 0;
  for(i = 1; i < taille; ++i) {
    if(tab[i] < tab[pos_min])
      pos_min = i;
  }
  return pos_min;
}

/** Fonction qui trie les clefs du tableau tab selon dans l'ordre croissant */
void tri_selection(int * tab, int taille) {
  int i;
  int pos_min;
  for(i = 0; i < taille-1; ++i) {
    pos_min = recherche_pos_min(tab + i, taille - i);
    /* N.B. L'indice d'une clef dans le tab de tri_selection est
    son indice dans le tab de recherche_pos_min + i */
    swap(&tab[i], &tab[i+pos_min]);
  }
}
```

Passage de tableaux en arguments : tri par sélection

```
1  /** Fonction qui renvoie l'indice de la plus petite clef du tableau tab */
2  int recherche_pos_min(int * tab, int taille) {
3    int i;
4    int pos_min = 0;
5    for(i = 1; i < taille; ++i) {
6      if(*(tab + i) < *(tab + pos_min))
7        pos_min = i;
8    }
9    return pos_min;
0  }
1
2  /** Fonction qui trie les clefs du tableau tab selon dans l'ordre croissant */
3  void tri_selection(int * tab, int taille) {
4    int i;
5    int pos_min;
6    for(i = 0; i < taille-1; ++i) {
7      pos_min = recherche_pos_min(tab + i, taille - i);
8      /* N.B. L'indice d'une clef dans le tab de tri_selection est
9      son indice dans le tab de recherche_pos_min + i */
0      swap(tab + i, tab + i + pos_min);
1    }
2  }
```

Différences sémantiques pointeurs/tableaux

Autres différences entre pointeur et tableau

- La (sémantique de la) fonction `sizeof`
  - Si c'est une variable pointeur :
    Renvoie la taille de stockage d'une variable pointeur
  - Si c'est un tableau :
    Renvoie la taille de stockage de tout le tableau
- L(a sémantique de l)'opérateur & ;
  - Si c'est une variable pointeur :
    Renvoie l'adresse de la variable pointeur
  - Si c'est un tableau :
    Renvoie l'adresse du tableau

Pointeurs/tableaux

Qu'affiche ce programme ?

```c
#include <stdio.h>
#include <stdlib.h>
/* programme compile et execute sur mon Mac */

int main(){
    int tab[] = {1, 2, 3, 4, 5, 6};
    int *ptr = tab;

    printf("Taille de tab : %lu, taille de &tab : %lu\n", sizeof(tab), sizeof(&tab));
    printf("Taille de ptr : %lu\n", sizeof(ptr));
    printf("Adresse de tab (&tab) : %p, (tab) : %p\n", &tab, tab);
    printf("Adresse de ptr : %p\n", &ptr);
    printf("Adresse de tab[0] : %p\n", &tab[0]);
    return EXIT_SUCCESS;
}
```

Taille de tab : 24, taille de &tab : 8
Taille de ptr : 8
Adresse de tab (&tab) : 0x7ffeec022a60, (tab) : 0x7ffeec022a60
Adresse de ptr : 0x7ffeec022a50
Adresse de tab[0] : 0x7ffeec022a60

Pointeurs/tableaux

Qu'affiche ce programme ?

```c
#include <stdio.h>
#include <stdlib.h>
/* programme compile et execute sur mon Mac */

int main(){
    int tab[] = {1, 2, 3, 4, 5, 6};
    int *ptr = tab;

    printf("Taille de tab : %lu, taille de &tab : %lu\n", sizeof(tab), sizeof(&tab));
    printf("Taille de ptr : %lu\n", sizeof(ptr));
    printf("Adresse de tab (&tab) : %p, (tab) : %p\n", &tab, tab);
    printf("Adresse de ptr : %p\n", &ptr);
    printf("Adresse de tab[0] : %p\n", &tab[0]);
    return EXIT_SUCCESS;
}
```

Taille de tab : 24, taille de &tab : 8
Taille de ptr : 8
Adresse de tab (&tab) : 0x7ffeec022a60, (tab) : 0x7ffeec022a60
Adresse de ptr : 0x7ffeec022a50
Adresse de tab[0] : 0x7ffeec022a60