

Fonctions

Programmation séquentielle en C, 2026-2027

Orestis Malaspinas (A401)

2026-09-28

Informatique et Systèmes de Communication, HEPIA

Rappel : les fonctions (1/2)

Anatomie d'une fonction

```
int max(int a, int b) { // type de retour, nom, paramètres
    if (a > b) {          // corps de la fonction
        return a;        // return: on sort de la fonction
    }                    // avec une valeur de type int
    return b;
}

int main() {
    int c = max(4, 5); // appel de la fonction, c vaut 5
    return 0;
}
```

Rappel : les fonctions (2/2)

Rappels

- **void** : pas de valeur de retour, ou pas de paramètre.
- Le **prototype** déclare la signature avant l'implémentation :

```
int max(int a, int b); // prototype
```

Anatomie d'un appel de fonction

Que se passe-t-il lors de l'appel `max(4, 5)` ?

1. Les arguments sont **évalués**.
2. Un espace mémoire est réservé **sur la pile** pour la fonction appelée.
3. Les paramètres sont de **nouvelles variables** allouées dans cet espace. Elles sont initialisées avec une **copie** des arguments.
4. Le corps de la fonction est exécuté.
5. La valeur de retour est **copiée** vers l'appelant.
6. La mémoire associée à la fonction est **détruite** : paramètres et variables locales disparaissent.

- Comment sont passés les arguments d'une fonction en C ?

Passage par copie (1/3)

- Comment sont passés les arguments d'une fonction en C ?
- **Toujours par copie.**

```
void foo(int a) {  
    a = 2;  
}  
  
int main() {  
    int x = 1;  
    foo(x);  
    // Que vaut x ici?  
    return 0;  
}
```

Passage par copie (2/3)

La pile pendant l'appel à `foo(x)`, après `a = 2`

mémoire	variable	adresse	valeur
main	x	1000	1
foo	a	1004	2

- `x` et `a` sont **deux variables distinctes**, à deux adresses différentes.
- `a` a été initialisée avec une **copie** de la valeur de `x`.
- Modifier `a` ne touche pas à `x`.

Passage par copie (3/3)

Après le retour de `foo(x)`

mémoire	variable	adresse	valeur
<code>main</code>	<code>x</code>	1000	1

- La mémoire associée à `foo` est détruite : `a` n'existe plus.
- `x` vaut toujours 1.

Le nom ne compte pas

```
void foo(int x) { // un autre x, dans la mémoire de foo
    x = 2;
}
```

- Même résultat : le `x` de `foo` et celui de `main` sont deux variables distinctes.

Passer une adresse (1/2)

- Comment modifier une variable de l'appelant ?

Passer une adresse (1/2)

- Comment modifier une variable de l'appelant?
- On passe son **adresse**.

```
void foo(int *a) { // le pointeur a
    *a = 2;        // déréférencement du pointeur a
}

int main() {
    int x = 1;
    foo(&x); // adresse de x
    // Ici x vaut 2
    return 0;
}
```

- Une nouvelle variable `int *a` est créée lors de l'appel à `foo`, et on lui assigne une copie de `&x`.
- `&x` n'est jamais modifiée **mais** `x` l'est.

Passer une adresse (2/2)

La pile pendant l'appel à `foo(&x)`, après `*a = 2`

mémoire	variable	adresse	valeur
main	x	1000	2
foo	a	1004	1000

- `a` contient l'**adresse** de `x` : `*a` **est** `x`.
- On copie **toujours** quelque chose : ici, c'est une adresse.

Le pointeur aussi est copié

```
void foo(int *a) {  
    *a = 2;    // modifie la variable pointée (x)  
    a = NULL; // modifie seulement la copie locale a  
}  
  
int main() {  
    int x = 1;  
    int *p = &x;  
    foo(p);  
    // Que valent x et p ici?  
    return 0;  
}
```

Le pointeur aussi est copié

```
void foo(int *a) {  
    *a = 2;    // modifie la variable pointée (x)  
    a = NULL; // modifie seulement la copie locale a  
}  
  
int main() {  
    int x = 1;  
    int *p = &x;  
    foo(p);  
    // Que valent x et p ici?  
    return 0;  
}
```

- `x` vaut 2.
- `p` vaut toujours `&x` : `a` n'était qu'une **copie** de `p`.

Retourner une valeur

- On peut *retourner* une valeur depuis une fonction.

```
int foo(int a) {  
    return a + 2;  
}  
  
int main() {  
    int x = 1;  
    int y = foo(x); // x n'est pas modifiée  
                   // et y vaut x + 2  
  
    return 0;  
}
```

- La valeur de retour est **copiée** vers l'appelant.
- On ne peut retourner qu'**une seule** valeur.
- Pour en "retourner" plusieurs : passer des **pointeurs** en argument (cf. `swap()` plus loin).

Ne jamais retourner l'adresse d'une variable locale

```
int *foo() {  
    int x = 2;  
    return &x;  
}  
  
int main() {  
    int *p = foo();  
    // Que vaut *p?  
    return 0;  
}
```

Ne jamais retourner l'adresse d'une variable locale

```
int *foo() {  
    int x = 2;  
    return &x;  
}  
  
int main() {  
    int *p = foo();  
    // Que vaut *p?  
    return 0;  
}
```

- `x` est **détruite** à la fin de `foo` : `p` pointe vers une zone de la pile qui n'existe plus.
- Lire `*p` est un **comportement indéfini** !

Exercice : échanger deux variables

```
void swap(int a, int b) {  
    int tmp = a;  
    a = b;  
    b = tmp;  
}  
  
int main() {  
    int x = 1, y = 2;  
    swap(x, y);  
    // Que valent x et y ici?  
    return 0;  
}
```

- Pourquoi cette fonction ne fonctionne-t-elle pas ?
- Corrigez-la.

Exercice : solution

```
void swap(int *a, int *b) {  
    int tmp = *a;  
    *a = *b;  
    *b = tmp;  
}  
  
int main() {  
    int x = 1, y = 2;  
    swap(&x, &y);  
    // x vaut 2 et y vaut 1  
    return 0;  
}
```

- `a` et `b` sont des copies des **adresses** de `x` et `y`.
- `*a` et `*b` permettent de modifier `x` et `y`.