

# Programmation Langage C

## Les fonctions

Youssef ALJ

26 mars 2020

## 1 Les fonctions

- Pourquoi utiliser les fonctions
- Déclaration, définition, appel
- Paramètres
- L'expression return
- Différents types de fonction

Un programme C est un ensemble d'instructions qu'on veut exécuter.

Pour écrire un programme () C, on peut soit :

- Tout écrire dans un seul endroit, c'est à dire dans le `main()`. C'est ce qu'on faisait jusqu'à maintenant.
- Diviser le programme en plusieurs petites tâches (ou fonctions). Et utiliser la fonction `main()` pour exécuter ces tâches.

# On veut savoir si un nombre est pair ou non

## Version sans fonction

```
#include <stdio.h>
void main(){
    int a;
    printf("saisir entier\n");
    scanf("%d", &a);
    if (a % 2 == 0){
        printf("pair\n");
    }
    else{
        printf("impair\n");
    }
}
```

## Version avec fonction

```
#include <stdio.h>
void affiche_parite(int a){
    if (a % 2 == 0){
        printf("pair\n");
    }
    else{
        printf("impair\n");
    }
}
void main(){
    int a;
    printf("saisir entier\n");
    scanf("%d", &a);
    affiche_parite(a);
}
```

- Une fonction = suite d'instructions groupées pour faire une tâche spécifique.
- Exemple de fonctions C déjà utilisées : `printf`, `scanf`, `sqrt`, etc.
- La plus importante reste la fonction `main` qui est indispensable pour chaque programme C.

## Avantage 1

- **Ré-utilisabilité** : une fonction peut être définie une fois et ré-utilisée plusieurs fois.

## Exemple : ré-utilisabilité

```
#include <stdio.h>
void affiche_parite(int x){
    if (x % 2 == 0){
        printf("pair\n");
    }
    else{
        printf("impair\n");
    }
}

void main(){
    int a,b,c;
    scanf("%d", &a);
    scanf("%d", &b);
    scanf("%d", &c);
    // utilisation 1
    affiche_parite(a);
    // utilisation 2
    affiche_parite(b);
    // utilisation 3
    affiche_parite(c);
}
```

## Avantage 2 : Abstraction

- Abstraction : cacher les détails d'implémentation.
- Exemple : On utilise tous la fonction `printf()`.
- Mais on ne sait pas comment fait cette fonction réellement pour faire l'affichage.
- Ceci est un avantage car l'utilisateur de la fonction n'a pas besoin de savoir les détails de l'implémentation de `printf()` pour pouvoir l'utiliser.

## Exemple : Abstraction

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

void main(){
    printf("la racine carree de 2 est %f",
        sqrt(2));
}
```

## Avantage 3 : Débogage

- **Débogage** (en anglais debugging) :
- Dans l'exemple ci-contre si on n'avait pas utilisé la fonction `affiche_parite` on devrait faire un test de parité pour `a`, `b` et `c`.
- Si on s'aperçoit plus tard qu'il y a une erreur dans ce test il faudra corriger cette erreur 3 fois.
- avec l'utilisation de la fonction `affiche_parite`, on ne corrigerait cette erreur **qu'une seule fois**.

## Exemple : débogage

```
#include <stdio.h>
void affiche_parite(int x){
    if (x % 2 == 0){
        printf("pair\n");
    }
    else{
        printf("impair\n");
    }
}

void main(){
    int a,b,c;
    scanf("%d", &a);
    scanf("%d", &b);
    scanf("%d", &c);
    // utilisation 1
    affiche_parite(a);
    // utilisation 2
    affiche_parite(b);
    // utilisation 3
    affiche_parite(c);
}
```



Il faut faire la distinction entre trois notions importantes concernant les fonctions : déclaration, définition et appel de fonction.

## Définition de fonction

On définit le traitement de la fonction.

```
type_retour nom(params){  
    // code  
}
```

## Exemple de définition

```
void affiche_parity(int x){  
    if (){  
        ...  
    }  
    else{  
        ...  
    }  
}
```

## Déclaration de fonction

On dit au compilateur que la fonction existe.  
Syntaxe :

```
type_retour nom(params);
```

## Exemple de déclaration

```
void affiche_parity(int x);
```

## Appel de fonction

On utilise la fonction.  
Syntaxe : `nom(params);`

## Exemple d'appel

```
affiche_parity(9);
```

## Exemple 2 :

```
// exemple : calcul du produit de deux nombres
#include <stdio.h>

// declaration de la fonction calcul_produit
int calcul_produit(int x, int y);

// definition de la fonction main
void main(){
    // declaration des variables
    int a, b, produit;

    printf("saisissez deux entiers\n");
    scanf("%d%d", &a, &b);

    // appel de la fonction produit
    produit = calcul_produit(a,b);

    // affichage du resultat
    printf("Le produit est=%d", produit);
}

// definition de la fonction calcul_produit
int calcul_produit(int x, int y){
    int prod = x * y;
    return prod;
}
```

On distingue deux types de paramètres : formels et effectifs.

- Les paramètres formels : sont les paramètres avec lesquels la fonction est définie. **Ils sont utilisés avec leur type.**
- Les paramètres effectifs : sont les paramètres avec lesquels la fonction est effectivement appelée. **Ils sont utilisés sans leur type.**

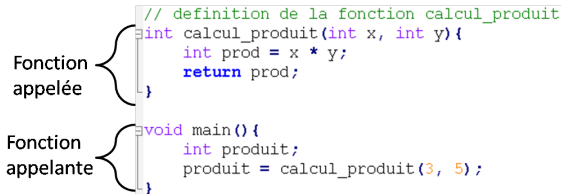
```
// definition de la fonction calcul_produit
int calcul_produit(int x, int y){
    int prod = x * y;
    return prod;
}

void main(){
    int produit;
    produit = calcul_produit(3, 5);
}
```

**Paramètres formels**

**Paramètres effectifs**

- Pour récupérer le résultat de calcul d'une fonction, on utilise l'expression `return valeur;` **dans la définition de la fonction.**
- On récupère ensuite le résultat de la fonction et on le stocke dans une nouvelle variable.



```
// definition de la fonction calcul_produit
int calcul_produit(int x, int y){
    int prod = x * y;
    return prod;
}

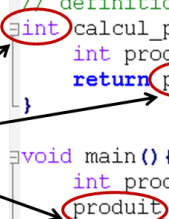
void main(){
    int produit;
    produit = calcul_produit(3, 5);
}
```

The diagram illustrates the relationship between a function definition and its caller. A curly brace on the left groups the first function definition (`calcul_produit`) and is labeled "Fonction appelée" (Function called). Another curly brace on the left groups the `main` function, which calls `calcul_produit`, and is labeled "Fonction appelante" (Calling function).

Même type!!

```
// definition de la fonction calcul_produit
int calcul_produit(int x, int y){
    int prod = x * y;
    return prod;
}

void main(){
    int produit;
    produit = calcul_produit(3, 5);
}
```



The diagram consists of three arrows originating from the text 'Même type!!' on the left. The first arrow points to the 'int' keyword in the function signature 'int calcul\_produit'. The second arrow points to the 'return prod;' statement in the function body. The third arrow points to the 'produit' variable in the 'main' function, which is assigned the result of the function call. The 'int' keyword in the function signature, the 'return prod;' statement, and the 'produit' variable are all circled in red in the original image, highlighting that they all represent the same integer type.

- 1 Écrire une fonction appelée `max` qui prend en entrée deux entiers `a` et `b` et qui renvoie le maximum des deux.
- 2 Tester votre fonction dans une fonction `main` en demandant à l'utilisateur de saisir les valeurs de son choix.
- 3 Ajouter une nouvelle fonction `somme` qui calcule la somme des entiers de départ. La tester.
- 4 Ajouter une nouvelle fonction `moyenne` pour calculer la moyenne des deux entiers. La tester.

```
type_retour nom_de_la_fonction (parametres)
```

Cette syntaxe de la déclaration d'une fonction autorise 4 cas de figures possibles :

- Pas de type de retour et pas de paramètres.
- Pas de type de retour mais avec des paramètres.
- Avec un type de retour mais sans paramètres.
- Avec un type de retour et avec des paramètres.

Pas de type de retour et pas de paramètres.

### Syntaxe

```
void nom_fonction()  
{  
    // corps de la fonction  
}
```

Avec un type de retour mais sans paramètres.

### Syntaxe

```
type_retour nom_fonction()  
{  
    // corps de la fonction  
    return valeur;  
}
```

Pas de type de retour mais avec des paramètres.

### Syntaxe

```
void nom_fonction(type1 param1, type2  
    param2, ...)  
{  
    // corps de la fonction  
}
```

Avec un type de retour et avec des paramètres

### Syntaxe

```
type_retour nom_fonction(type1 param1,  
    type2 param2, ...)  
{  
    // corps de la fonction  
    return valeur;  
}
```



# Exemple 1 : Pas de type de retour pas de paramètres

Pas de type de retour pas de paramètres

## Syntaxe

```
void nom_fonction()  
{  
    // corps de la fonction  
}
```

## Exemple

```
#include <stdio.h>  
// declaration  
void generer_pairs();  
  
// definition de main  
void main(){  
    // appel  
    generer_pairs();  
}  
  
// definition de generer-pairs  
void generer_pairs(){  
    int i;  
    for (i=0; i<100; i = i +2){  
        printf("%d\n", i);  
    }  
}
```

## Exemple 2 : Pas de type de retour avec paramètres

Pas de type de retour avec paramètres.

### Syntaxe

```
void nom-fonction(type1 param1, type2  
    param2, ...)  
{  
    // corps de la fonction  
}
```

### Exemple

```
#include <stdio.h>  
// declaration de affiche-parite  
void affiche-parite(int x);  
  
// definition  
void main(){  
    // appel  
    int a;  
    scanf("%d", &a);  
    affiche-parite(a);  
}  
  
// definition de affiche-parite  
void affiche-parite(int x){  
    if (x % 2 == 0){  
        printf("pair\n");  
    }  
    else{  
        printf("imppair\n");  
    }  
}
```

## Exemple 3 : Avec type de retour pas de paramètres

Avec type de retour pas de paramètres

### Syntaxe

```
type_retour nom_fonction()  
{  
    // corps de la fonction  
    return valeur;  
}
```

### Exemple

```
#include <stdio.h>  
#include <stdlib.h> // utilise pour rand()  
  
// declaration de alea  
int alea();  
  
// definition main  
void main(){  
    int resultat;  
    // appel  
    resultat = alea();  
    printf("nb aleatoire=%d", resultat);  
}  
  
// definition de alea  
int alea(){  
    int a;  
    a = rand();  
    return a;  
}
```

# Exemple 4 : Avec type de retour avec de paramètres

Avec type de retour avec de paramètres.

## Syntaxe

```
type-retour nom-fonction(type1 param1,  
    type2 param2, ...)  
{  
    // corps de la fonction  
    return valeur;  
}
```

## Exemple

```
#include <stdio.h>  
// declaration de pair_impair  
int pair_impair(int num);  
  
// definition main  
void main(){  
    int a, resultat;  
    scanf("%d", &a);  
    // appel  
    resultat = pair_impair(a);  
    if (resultat == 0){  
        printf("pair");  
    }  
    else{  
        printf("impair");  
    }  
}  
  
// definition de pair_impair  
int pair_impair(int x){  
    if (x % 2 == 0){  
        return 0;  
    }  
    else{  
        return 1;  
    }  
}
```