Apprenez à programmer en C !

40 heures Moyenne

Mis à jour le 29/07/2019

Les fonctions

Nous terminerons la partie I du cours (« Les bases ») par cette notion fondamentale que sont les fonctions en langage C. Tous les programmes en C se basent sur le principe que je vais vous expliquer dans ce chapitre.

Nous allons apprendre à structurer nos programmes en petits bouts... un peu comme si on jouait aux Legos.
Tous les gros programmes en C sont en fait des assemblages de petits bouts de code, et ces petits bouts de code sont justement ce qu'on appelle... des fonctions !

Créer et appeler une fonction

Nous avons vu dans les tout premiers chapitres qu'un programme en C commençait par une fonction appelée `main`.
Je vous avais même fait un schéma récapitulatif, pour vous rappeler quelques mots de vocabulaire (fig. suivante).

En haut, on y trouve les directives de préprocesseur (un nom barbare sur lequel on reviendra d'ailleurs). Ces directives sont faciles à identifier : elles commencent par un `#` et sont généralement mises tout en haut des fichiers sources.

Puis en dessous, il y avait ce que j'avais déjà appelé « une fonction ». Ici, sur mon schéma, vous voyez une fonction `main` (pas trop remplie il faut le reconnaître).

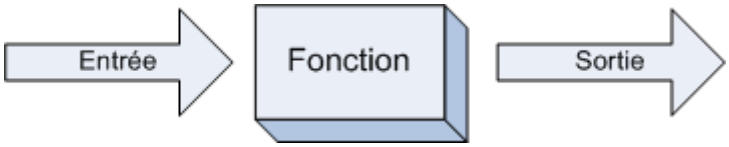
Je vous avais dit qu'un programme en langage C commençait par la fonction `main`. Je vous rassure, c'est toujours vrai ! Seulement, jusqu'ici nous sommes restés à l'intérieur de la fonction `main`. Nous n'en sommes jamais sortis. Revoyez vos codes sources et vous verrez : nous sommes toujours restés à l'intérieur des accolades de la fonction `main`.

> Eh bien, c'est mal d'avoir fait comme ça ?

Non ce n'est pas « mal », mais ce n'est pas ce que les programmeurs en C font dans la réalité. Quasiment aucun programme n'est écrit uniquement à l'intérieur des accolades de la fonction `main`. Jusqu'ici nos programmes étaient courts, donc ça ne posait pas de gros problèmes, mais imaginez des plus gros programmes qui font des milliers de lignes de code ! Si tout était concentré dans la fonction `main`, bonjour le bazar...

Nous allons donc maintenant apprendre à nous organiser. Nous allons en fait découper nos programmes en petits bouts (souvenez-vous de l'image des Legos que je vous ai donnée tout à l'heure). Chaque « petit bout de programme » sera ce qu'on appelle une fonction.

Une fonction exécute des actions et renvoie un résultat. C'est un **morceau de code** qui sert à faire quelque chose de précis.
On dit qu'une fonction possède une entrée et une sortie. La fig. suivante représente une fonction schématiquement.



Lorsqu'on appelle une fonction, il y a trois étapes.

1. **L'entrée**: on fait « rentrer » des informations dans la fonction (en lui donnant des informations avec lesquelles travailler).
2. **Les calculs** : grâce aux informations qu'elle a reçues en entrée, la fonction travaille.
3. **La sortie** : une fois qu'elle a fini ses calculs, la fonction renvoie un résultat. C'est ce qu'on appelle la sortie, ou encore le retour.

Concrètement, on peut imaginer par exemple une fonction appelée `triple` qui calcule le triple du nombre qu'on lui donne, en le multipliant par 3 (fig. suivante). Bien entendu, les fonctions seront en général plus compliquées.

La fonction « triple » multiplie le nombre en entrée par 3

Le but des fonctions est donc de simplifier le code source, pour ne pas avoir à retaper le même code plusieurs fois d'affilée.

Rêvez un peu : plus tard, nous créerons par exemple une fonction `afficherFenetre` qui ouvrira une fenêtre à l'écran. Une fois la fonction écrite (c'est l'étape la plus difficile), on n'aura plus qu'à dire « Hep ! toi la fonction `afficherFenetre`, ouvre-moi une fenêtre ! ». On pourra aussi écrire une fonction `deplacerPersonnage` dont le but sera de déplacer le personnage d'un jeu à l'écran, etc.

Schéma d'une fonction

Vous avez déjà eu un aperçu de la façon dont est faite une fonction avec la fonction `main`.
Cependant pour bien que vous compreniez il va falloir que je vous montre quand même comment on construit une fonction.

Le code suivant représente une fonction schématiquement. C'est un modèle à connaître :

c

```c
type nomFonction(parametres)
{
    // Insérez vos instructions ici
}
```

Vous reconnaissez la forme de la fonction `main`.
Voici ce qu'il faut savoir sur ce schéma.

- `type` (correspond à la sortie) : c'est le type de la fonction. Comme les variables, les fonctions ont un type. Ce type dépend du résultat que la fonction renvoie : si la fonction renvoie un nombre décimal, vous mettrez sûrement `double`, si elle renvoie un entier vous mettrez `int` ou `long` par exemple. Mais il est aussi possible de créer des fonctions qui ne renvoient rien !

Il y a donc deux sortes de fonctions :

- les fonctions qui renvoient une valeur : on leur met un des types que l'on connaît (`char`, `int`, `double`, etc.) ;
- les fonctions qui ne renvoient pas de valeur : on leur met un type spécial `void` (qui signifie « vide »).

- `nomFonction` : c'est le nom de votre fonction. Vous pouvez appeler votre fonction comme vous voulez, du temps que vous respectez les mêmes règles que pour les variables (pas d'accents, pas d'espaces, etc.).

- `parametres` (correspond à l'entrée) : entre parenthèses, vous pouvez envoyer des paramètres à la fonction. Ce sont des valeurs avec lesquelles la fonction va travailler.

  Vous pouvez envoyer autant de paramètres que vous le voulez. Vous pouvez aussi n'envoyer aucun paramètre à la fonction, mais ça se fait plus rarement.

Par exemple, pour une fonction `triple`, vous envoyez un nombre en paramètre. La fonction « récupère » ce nombre et en calcule le triple, en le multipliant par 3. Elle renvoie ensuite le résultat de ses calculs.

- Ensuite vous avez les **accolades** qui indiquent le début et la fin de la fonction. À l'intérieur de ces accolades vous mettrez les instructions que vous voulez. Pour la fonction `triple`, il faudra taper des instructions qui multiplient par 3 le nombre reçu en entrée.

Une fonction, c'est donc un mécanisme qui reçoit des valeurs en entrée (les paramètres) et qui renvoie un résultat en sortie.

Créer une fonction

Voyons un exemple pratique sans plus tarder : la fameuse fonction `triple` dont je vous parle depuis tout à l'heure. On va dire que cette fonction reçoit un nombre entier de type `int` et qu'elle renvoie un nombre entier aussi de type `int`. Cette fonction calcule le triple du nombre qu'on lui donne :

c

```c
int triple(int nombre)
{
    int resultat = 0;

    resultat = 3 * nombre;  // On multiplie le nombre fourni par 3
    return resultat;       // On retourne la variable resultat qui vaut le triple de nombre
}
```

Voilà notre première fonction ! Une première chose importante : comme vous le voyez, la fonction est de type `int`. Elle doit donc renvoyer une valeur de type `int`.

Entre les parenthèses, vous avez les variables que la fonction reçoit. Ici, notre fonction `triple` reçoit une variable de type `int` appelée `nombre`.

La ligne qui donne pour consigne de « renvoyer une valeur » est celle qui contient le `return`. Cette ligne se trouve généralement à la fin de la fonction, après les calculs.

c

```c
return resultat;
```

Ce code signifie pour la fonction : « Arrête-toi là et renvoie le nombre `resultat` ». Cette variable `resultat` DOIT être de type `int`, car la fonction renvoie un `int` comme on l'a dit plus haut.

La variable `resultat` est déclarée (= créée) dans la fonction `triple`. Cela signifie qu'elle n'est utilisable que dans cette fonction, et pas dans une autre comme la fonction `main` par exemple. C'est donc une variable propre à la fonction `triple`.

Mais est-ce la façon la plus courte d'écrire notre fonction `triple` ?
Non, on peut faire tout cela en une ligne en fait :

c

```c
int triple(int nombre)
{
    return 3 * nombre;
}
```

Cette fonction fait exactement la même chose que la fonction de tout à l'heure, elle est juste plus rapide à écrire. Généralement, vos fonctions contiendront plusieurs variables pour effectuer leurs calculs et leurs opérations, rares seront les fonctions aussi courtes que `triple`.

Plusieurs paramètres, aucun paramètre

Plusieurs paramètres

Notre fonction `triple` contient un paramètre, mais il est possible de créer des fonctions acceptant plusieurs paramètres.
Par exemple, une fonction `addition` qui additionne deux nombres `a` et `b` :

c

```c
int addition(int a, int b)
{
    return a + b;
}
```

Il suffit de séparer les différents paramètres par une virgule comme vous le voyez.

Aucun paramètre

Certaines fonctions, plus rares, ne prennent aucun paramètre en entrée. Ces fonctions feront généralement toujours la même chose. En effet, si elles n'ont pas de nombres sur lesquels travailler, vos fonctions serviront juste à effectuer certaines actions, comme afficher du texte à l'écran. Et encore, ce sera forcément toujours le même texte puisque la fonction ne reçoit aucun paramètre susceptible de modifier son comportement !

Imaginons une fonction `bonjour` qui affiche juste « Bonjour » à l'écran :

c

```c
void bonjour()
{
    printf("Bonjour");
}
```

Je n'ai rien mis entre parenthèses car la fonction ne prend aucun paramètre.
De plus, j'ai utilisé le type `void` dont je vous ai parlé plus haut.

En effet, comme vous le voyez ma fonction n'a pas non plus de `return`. Elle ne retourne rien. Une fonction qui ne retourne rien est de type `void`.

Appeler une fonction

On va maintenant tester un code source pour s'entraîner un peu avec ce qu'on vient d'apprendre.
Nous allons utiliser notre fonction `triple` (décidément je l'aime bien) pour calculer le triple d'un nombre.

Pour le moment, je vous demande d'écrire la fonction `triple` AVANT la fonction `main`. Si vous la placez après, ça ne marchera pas. Je vous expliquerai pourquoi par la suite.

Voici un code à tester et à comprendre :

c

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

int triple(int nombre)
{
    return 3 * nombre;
}

int main(int argc, char *argv[])
{
    int nombreEntre = 0, nombreTriple = 0;

    printf("Entrez un nombre... ");
    scanf("%d", &nombreEntre);

    nombreTriple = triple(nombreEntre);
    printf("Le triple de ce nombre est %d\n", nombreTriple);

    return 0;
}
```

Notre programme commence par la fonction `main` comme vous le savez.
On demande à l'utilisateur d'entrer un nombre. On envoie ce nombre qu'il a entré à la fonction `triple`, et on récupère le résultat dans la variable `nombreTriple`. Regardez en particulier cette ligne, c'est la plus intéressante car c'est l'appel de la fonction :

c

```c
nombreTriple = triple(nombreEntre);
```

Entre parenthèses, on envoie une variable en **entrée** à la fonction `triple`, c'est le nombre sur lequel elle va travailler.
Cette fonction renvoie une valeur, valeur qu'on récupère dans la variable `nombreTriple`. On ordonne

donc à l'ordinateur dans cette ligne : « Demande à la fonction `triple` de me calculer le triple de `nombreEntre`, et stocke le résultat dans la variable `nombreTriple` ».

Les mêmes explications sous forme de schéma

Vous avez encore du mal à comprendre comment ça fonctionne concrètement ?
Pas de panique ! Je suis sûr que vous allez comprendre avec mes schémas.

Ce code particulièrement commenté vous indique dans quel ordre le code est lu. Commencez donc par lire la ligne numérotée 1, puis 2, puis 3 (bon vous avez compris je crois !) :

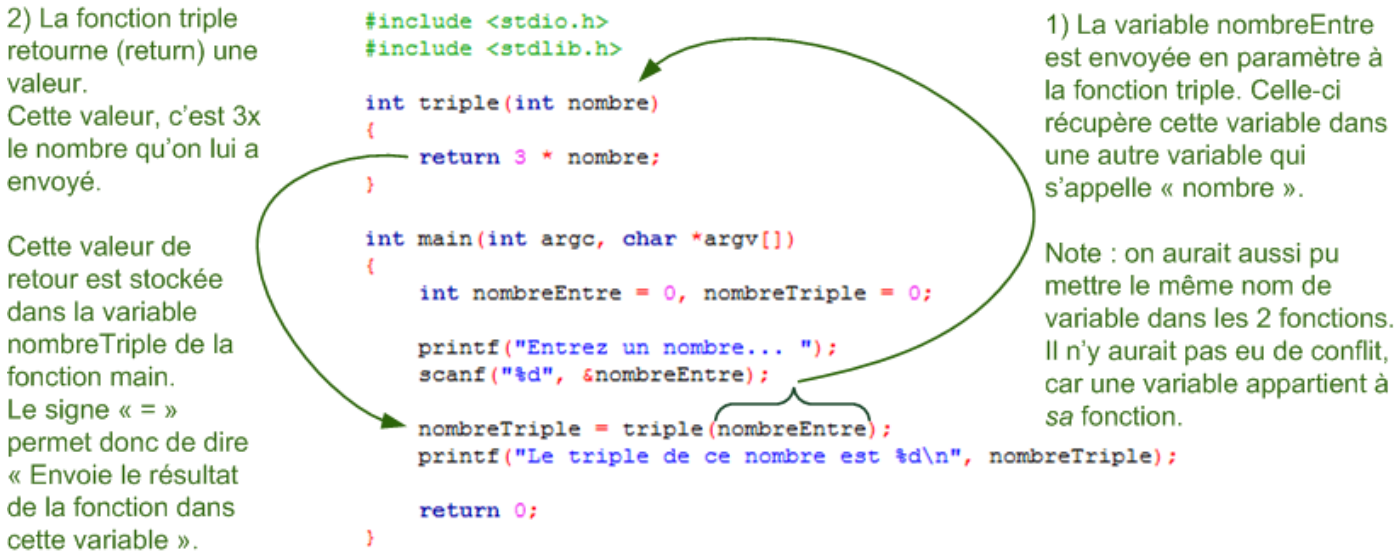
c

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

int triple(int nombre) // 6
{
    return 3 * nombre; // 7
}

int main(int argc, char *argv[]) // 1
{
    int nombreEntre = 0, nombreTriple = 0; // 2

    printf("Entrez un nombre... "); // 3
    scanf("%d", &nombreEntre); // 4

    nombreTriple = triple(nombreEntre); // 5
    printf("Le triple de ce nombre est %d\n", nombreTriple); // 8

    return 0; // 9
}
```

Voici ce qui se passe, ligne par ligne.

1. Le programme commence par la fonction `main`.
2. Il lit les instructions dans la fonction une par une dans l'ordre.
3. Il lit l'instruction suivante et fait ce qui est demandé (`printf`).
4. De même, il lit l'instruction et fait ce qui est demandé (`scanf`).
5. Il lit l'instruction... Ah ! On appelle la fonction `triple`, on doit donc sauter à la ligne de la fonction `triple` plus haut.
6. On saute à la fonction `triple` et on récupère un paramètre (`nombre`).
7. On fait des calculs sur le nombre et on termine la fonction. `return` signifie la fin de la fonction et permet d'indiquer le résultat à renvoyer.
8. On retourne dans le `main` à l'instruction suivante.
9. Un `return` ! La fonction main se termine et donc le programme est terminé.

Si vous avez compris dans quel ordre l'ordinateur lit les instructions, vous avez déjà compris le principal. Maintenant, il faut bien comprendre qu'une fonction reçoit des paramètres en entrée et renvoie une valeur en sortie (fig. suivante).



Note : ce n'est pas le cas de toutes les fonctions. Parfois, une fonction ne prend aucun paramètre en entrée, ou au contraire elle en prend plusieurs (je vous ai expliqué ça un peu plus haut).
De même, parfois une fonction renvoie une valeur, parfois elle ne renvoie rien (dans ce cas il n'y a pas de `return`).

Testons ce programme

Voici un exemple d'utilisation du programme :

```
Entrez un nombre... 10
Le triple de ce nombre est 30
```

> Vous n'êtes pas obligés de stocker le résultat d'une fonction dans une variable ! Vous pouvez directement envoyer le résultat de la fonction triple à une autre fonction, comme si `triple(nombreEntre)` était une variable.

Regardez bien ceci, c'est le même code mais il y a un changement au niveau du dernier `printf`. De plus, on n'a pas déclaré de variable `nombreTriple` car on ne s'en sert plus :

c

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

int triple(int nombre)
{
    return 3 * nombre;
}

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

```
10 {
11     int nombreEntre = 0;
12 
13     printf("Entrez un nombre... ");
14     scanf("%d", &nombreEntre);
15 
16     // Le résultat de la fonction est directement envoyé au printf et n'est pas stocké dans une
   variable
17     printf("Le triple de ce nombre est %d\n", triple(nombreEntre));
18 
19     return 0;
20 }
```

Comme vous le voyez, `triple(nombreEntre)` est directement envoyé au `printf`.
Que fait l'ordinateur quand il tombe sur cette ligne ?

C'est très simple. Il voit que la ligne commence par `printf`, il va donc appeler la fonction `printf`.
Il envoie à la fonction `printf` tous les paramètres qu'on lui donne. Le premier paramètre est le texte à afficher et le second est un nombre.
Votre ordinateur voit que pour envoyer ce nombre à la fonction `printf` il doit d'abord appeler la fonction `triple`. C'est ce qu'il fait : il appelle `triple`, il effectue les calculs de `triple` et une fois qu'il a le résultat il l'envoie directement dans la fonction `printf` !

C'est un peu une imbrication de fonctions. Et le plus fin dans tout ça, c'est qu'une fonction peut en appeler une autre à son tour !
Notre fonction `triple` pourrait appeler une autre fonction, qui elle-même appellerait une autre fonction, etc. C'est ça le principe de la programmation en C ! Tout est combiné, comme dans un jeu de Lego.

Au final, le plus dur sera d'écrire vos fonctions. Une fois que vous les aurez écrites, vous n'aurez plus qu'à appeler les fonctions sans vous soucier des calculs qu'elles peuvent bien faire à l'intérieur. Ça va permettre de simplifier considérablement l'écriture de nos programmes et ça croyez-moi on en aura bien besoin !

Des exemples pour bien comprendre

Vous avez dû vous en rendre compte : je suis un maniaque des exemples.
La théorie c'est bien, mais si on ne fait que ça on risque de ne pas retenir grand-chose et surtout ne pas comprendre comment s'en servir, ce qui serait un peu dommage...

Je vais donc maintenant vous montrer plusieurs exemples d'utilisation de fonctions, pour que vous ayez une idée de leur intérêt. Je vais m'efforcer de faire des cas différents à chaque fois, pour que vous puissiez avoir des exemples de tous les types de fonctions qui peuvent exister.

Je ne vous apprendrai rien de nouveau, mais ce sera l'occasion de voir des exemples pratiques. Si vous avez déjà compris tout ce que j'ai expliqué avant, c'est très bien et normalement aucun des exemples qui vont suivre ne devrait vous surprendre.

Conversion euros / francs

On commence par une fonction très similaire à `triple`, qui a quand même un minimum d'intérêt cette fois : une fonction qui convertit les euros en francs. Pour ceux d'entre vous qui ne connaîtraient pas ces monnaies sachez que 1 euro = 6,55957 francs.

On va créer une fonction appelée `conversion`.
Cette fonction prend une variable en entrée de type `double` et retourne une sortie de type `double` car on va forcément manipuler des nombres décimaux. Lisez-la attentivement :

c

```c
double conversion(double euros)
{
    double francs = 0;

    francs = 6.55957 * euros;
    return francs;
}

int main(int argc, char *argv[])
{
    printf("10 euros = %fF\n", conversion(10));
    printf("50 euros = %fF\n", conversion(50));
    printf("100 euros = %fF\n", conversion(100));
    printf("200 euros = %fF\n", conversion(200));

    return 0;
}
```

```
10 euros = 65.595700F
50 euros = 327.978500F
100 euros = 655.957000F
200 euros = 1311.914000F
```

Il n'y a pas grand-chose de différent par rapport à la fonction `triple`, je vous avais prévenus. D'ailleurs, ma fonction `conversion` est un peu longue et pourrait être raccourcie en une ligne, je vous laisse le faire je vous ai déjà expliqué comment faire plus haut.

Dans la fonction `main`, j'ai fait exprès de faire plusieurs `printf` pour vous montrer l'intérêt d'avoir une fonction. Pour obtenir la valeur de 50 euros, je n'ai qu'à écrire `conversion(50)`. Et si je veux avoir la conversion en francs de 100 euros, j'ai juste besoin de changer le paramètre que j'envoie à la fonction (100 au lieu de 50).

À vous de jouer ! Écrivez une seconde fonction (toujours avant la fonction `main`) qui fera elle la conversion inverse : Francs => Euros. Ce ne sera pas bien difficile, il y a juste un signe d'opération à changer.

La punition

On va maintenant s'intéresser à une fonction qui ne renvoie rien (pas de sortie).
C'est une fonction qui affiche le même message à l'écran autant de fois qu'on lui demande. Cette fonction prend un paramètre en entrée : le nombre de fois où il faut afficher la punition.

c

```c
void punition(int nombreDeLignes)
{
    int i;

    for (i = 0 ; i < nombreDeLignes ; i++)
    {
        printf("Je ne dois pas recopier mon voisin\n");
    }
}

int main(int argc, char *argv[])
{
    punition(10);

    return 0;
}
```

```
Je ne dois pas recopier mon voisin
Je ne dois pas recopier mon voisin
Je ne dois pas recopier mon voisin
Je ne dois pas recopier mon voisin
Je ne dois pas recopier mon voisin
Je ne dois pas recopier mon voisin
Je ne dois pas recopier mon voisin
Je ne dois pas recopier mon voisin
Je ne dois pas recopier mon voisin
Je ne dois pas recopier mon voisin
```

On a ici affaire à une fonction qui ne renvoie aucune valeur. Cette fonction se contente juste d'effectuer des actions (ici, elle affiche des messages à l'écran).
Une fonction qui ne renvoie aucune valeur est de type `void`, c'est pour cela qu'on a écrit `void`. À part ça, il n'y a rien de bien différent.

Il aurait été bien plus intéressant de créer une fonction `punition` qui s'adapte à n'importe quelle sanction. On lui aurait envoyé deux paramètres : le texte à répéter et le nombre de fois qu'il doit être répété. Le problème, c'est qu'on ne sait pas encore gérer le texte en C (au cas où vous n'auriez pas vu, je vous rappelle qu'on n'a fait que manipuler des variables contenant des nombres depuis le début du cours !). D'ailleurs à ce sujet, je vous annonce que nous ne tarderons pas à apprendre à utiliser des variables qui retiennent du texte. C'est plus compliqué qu'il n'y paraît et on ne pouvait pas l'apprendre dès le début du cours !

Aire d'un rectangle

L'aire d'un rectangle est facile à calculer : `largeur * hauteur`.

Notre fonction nommée `aireRectangle` va prendre deux paramètres : la largeur et la hauteur. Elle renverra l'aire.

c

```c
double aireRectangle(double largeur, double hauteur)
{
    return largeur * hauteur;
}

int main(int argc, char *argv[])
{
    printf("Rectangle de largeur 5 et hauteur 10. Aire = %f\n", aireRectangle(5, 10));
    printf("Rectangle de largeur 2.5 et hauteur 3.5. Aire = %f\n", aireRectangle(2.5, 3.5));
    printf("Rectangle de largeur 4.2 et hauteur 9.7. Aire = %f\n", aireRectangle(4.2, 9.7));

    return 0;
}
```

```
Rectangle de largeur 5 et hauteur 10. Aire = 50.000000
Rectangle de largeur 2.5 et hauteur 3.5. Aire = 8.750000
Rectangle de largeur 4.2 et hauteur 9.7. Aire = 40.740000
```

> Pourrait-on afficher directement la largeur, la hauteur et l'aire dans la fonction ?

Bien sûr !

Dans ce cas, la fonction ne renverrait plus rien, elle se contenterait de calculer l'aire et de l'afficher immédiatement.

c

```c
void aireRectangle(double largeur, double hauteur)
{
    double aire = 0;

    aire = largeur * hauteur;
    printf("Rectangle de largeur %f et hauteur %f. Aire = %f\n", largeur, hauteur, aire);
}

int main(int argc, char *argv[])
{
    aireRectangle(5, 10);
    aireRectangle(2.5, 3.5);
    aireRectangle(4.2, 9.7);

    return 0;
}
```

Comme vous le voyez, le `printf` est *à l'intérieur* de la fonction `aireRectangle` et produit le même affichage que tout à l'heure. C'est juste une façon différente de procéder.

Un menu

Ce code est plus intéressant et concret. On crée une fonction `menu()` qui ne prend aucun paramètre en entrée. Cette fonction se contente d'afficher le menu et demande à l'utilisateur de faire un choix. La fonction renvoie le choix de l'utilisateur.

c

```c
int menu()
{
    int choix = 0;

    while (choix < 1 || choix > 4)
    {
        printf("Menu :\n");
        printf("1 : Poulet de dinde aux escargots rotis a la sauce bearnaise\n");
        printf("2 : Concombres sucres a la sauce de myrtilles enrobee de chocolat\n");
        printf("3 : Escalope de kangourou saignante et sa gelee aux fraises poivree\n");
        printf("4 : La surprise du Chef (j'en salive d'avance...)\n");
        printf("Votre choix ? ");
        scanf("%d", &choix);
    }

    return choix;
}

int main(int argc, char *argv[])
{
    switch (menu())
    {
        case 1:
            printf("Vous avez pris le poulet\n");
            break;
        case 2:
            printf("Vous avez pris les concombres\n");
            break;
        case 3:
            printf("Vous avez pris l'escalope\n");
            break;
        case 4:
            printf("Vous avez pris la surprise du Chef. Vous etes un sacre aventurier dites donc !\n");
            break;
    }

    return 0;
}
```

J'en ai profité pour améliorer le menu (par rapport à ce qu'on faisait habituellement) : la fonction `menu` affiche à nouveau le menu tant que l'utilisateur n'a pas entré un nombre compris entre 1 et 4. Comme ça, aucun risque que la fonction renvoie un nombre qui ne figure pas au menu !

Dans le `main`, vous avez vu qu'on fait un `switch(menu())`. Une fois que la fonction `menu()` est terminée, elle renvoie le choix de l'utilisateur directement dans le `switch`. C'est une méthode rapide et pratique.

À vous de jouer ! Le code est encore améliorable : on pourrait afficher un message d'erreur si l'utilisateur entre un mauvais nombre plutôt que de simplement afficher une nouvelle fois le menu.

En résumé

- Les fonctions s'appellent entre elles. Ainsi, le `main` peut appeler des fonctions toutes prêtes telles que `printf` ou `scanf`, mais aussi des fonctions que nous avons créées.
- Une fonction récupère en entrée des variables qu'on appelle **paramètres**.
- Elle effectue certaines opérations avec ces paramètres puis retourne en général une valeur à l'aide de l'instruction `return`.

Que pensez-vous de ce cours ?

TP : PLUS OU MOINS, VOTRE PREMIER JEU

QUIZ : QUIZ 1

Le professeur

Mathieu Nebra

Entrepreneur à plein temps, auteur à plein temps et co-fondateur d'OpenClassrooms :o)

Découvrez aussi ce cours en...

Livre PDF

OpenClassrooms

L'entreprise