

Fonctions

Temps d'étude conseillé : 6h.

Les fonctions sont un des outils fondamentaux de la programmation impérative. Elles permettent de clarifier le code et d'éviter de nombreuses répétitions. Ce chapitre permet de détailler la déclaration et l'appel des fonctions, ainsi que des notions connexes qui y sont liées comme la portée des variables ou le passage par valeur.

1 Introduction

Vous avez entendu parler des fonctions depuis plusieurs années déjà, notamment au travers des cours de mathématiques. Vous avez ainsi pu en déduire qu'il s'agit d'objets prenant des données en entrées et les transformant pour donner une nouvelle valeur. La bonne nouvelle, c'est que vous avez bien déduit ! La mauvaise nouvelle, il n'y en a pas, c'est vraiment ce que vous connaissez déjà. Il en reste donc qu'à formaliser cette notion et voir comment l'implémenter en C.

Définition 1 (Fonction). Une fonction est constituée d'un ensemble d'instructions, est définie en dehors du programme principal et peut être appelée par toute instruction l'utilisant correctement.

Le développement et l'usage d'une fonction sont deux choses différentes. La première consiste à rédiger les spécifications de la fonction : son nom, ses arguments, son fonctionnement... Une fois cette étape réalisée, la fonction est prête à être utilisée dans toute portion de code connaissant son existence.

La première étape consiste à *définir* une fonction. Au travers de cette étape, on indique au compilateur l'existence de la fonction et on spécifie ses paramètres. Cette étape est souvent combinée à la partie *déclaration* qui consiste à décrire ce que fait réellement la fonction. Le formatage à utiliser est alors le suivant.

```
type nom (type1 arg1, type2 arg2, ...)
{
    Déclarations;
    Instructions;
}
```

```
int multiplication(int a, int b)
{
    int multi = a * b;
    return multi
}
```

La première ligne s'appelle le *prototype* de la fonction. Elle permet de définir (dans l'ordre d'apparition de la première ligne) :

- le type de la valeur de retour ;
- le nom de la fonction ;
- les arguments (*i.e.* les données d'entrée).

dans le main, on appellera :

`printf("%d\n", multiplication(4,3));` 1

```

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

int kekchose(int b) //Là on fait un truc
{
    b = b +1;
    return b;
}

int UnTruc(int a) //Ici aussi on fait un truc
{
    int objet = kekchose(a);
    return objet;
}

int main()
{
    printf("%d\n", kekchose(2) * UnTruc(3));
    return 0;
}

```

les fonctions
sont lu dans
l'ordre. Il
faut donc les
déclarer dans
l'ordre

```

v #include<stdlib.h>
#include<stdio.h>

v int UnTruc(int a) //Ici aussi on fait un truc
{
    int objet = kekchose(a);
    return objet;
}

v int kekchose(int b) //Là on fait un truc
{
    b = b +1;
    return b;
}

v int main()
{
    printf("%d\n", kekchose(2) * UnTruc(3));
    return 0;
}

```

donc
appeler
une fonction
qui vient
plus tard
dans le code

ne compile pas

Une fois définie et déclarée, il faut rédiger les instructions dans le *corps* de la fonction (la partie entre les accolades) pour définir son fonctionnement. Cela consiste à rédiger des instructions manipulant les arguments et d'autres variables locales à la fonction pour produire une valeur qui sera au final retournée.

Règles. Les règles suivantes sont à suivre pour déclarer correctement une fonction :

- le type de retour peut être choisi parmi tous les types connus, y compris `void` ;
- le type de la valeur retournée doit être identique au type de retour ;
- seul le mot-clé `return` est défini pour renvoyer un résultat ;
- le nom de la fonction doit être unique et suit les règles des noms de variable ;
- le nombre d'arguments est potentiellement illimité ;
- chaque argument doit être défini par son type et son nom ;
- on ne définit jamais une fonction dans une fonction

La déclaration d'une fonction peut se réaliser dans le même fichier que celui contenant le `main()`. L'appel d'une fonction peut être réalisée n'importe où, tant que celle-ci est préalablement définie. Aussi, la déclaration de la fonction doit être placée avant le `main()`.

Exemple 1. On cherche à créer la fonction `trunc()` qui prend en entrée un réel et renvoie la partie entière de ce réel. Puisque la fonction retourne une partie entière, le type de retour est `int`. La fonction s'appelle `trunc` et prend en paramètre un réel : son seul argument est donc une variable de type `float`. Une fois la partie entière récupérée, on renvoie le résultat à l'aide du mot-clé `return`.

```
int trunc (float x)    // Prototype
{
    int res = (int)x;   // Stockage du résultat à l'aide d'un cast
    return res;        // Renvoi du résultat
}
```

Une fois la fonction `trunc()` déclarée, il est possible de l'appeler et de stocker le résultat dans une nouvelle variable. Les lignes suivantes peuvent alors être rédigées dans un `main()`.

```
float pi = 3.14;       // Déclaration d'une variable
int e;                // Pour stocker le résultat
e = trunc(pi);         // Appel à la fonction trunc()
```

TD Exercices 1 à 3.

2 Variables et fonctions

L'usage des fonctions implique directement la maîtrise de la notion de *portée des variables*. Cette notion explique les règles de vie et de mort des variables au sein du code. En plus de cette notion, il est nécessaire de comprendre comment la valeur d'une variable à une fonction. Ces deux notions fondamentales sont détaillées dans cette section.

2.1 Passage par valeur

Lors de l'appel d'une fonction, l'information qui est donnée à la fonction est la *valeur* de la variable, et non la variable en entier. En effet, lors de la déclaration d'une variable, cette dernière se voit dotée d'une adresse. Or, il est impossible de *déplacer* une variable : on peut modifier ou déplacer son contenu mais pas la variable en elle-même.

Il est donc logique que lors de l'appel d'une fonction, les paramètres de la fonction se comportent comme des nouvelles variables. Ainsi, lors de l'appel, la valeur de la variable passée en argument de la fonction est donnée aux arguments de la fonction.¹ Voyons cela au travers d'un exemple.

Exemple 2. Reprenons l'exemple précédent avec la fonction `trunc()` et observons l'état de la mémoire au fur et à mesure du déroulement du code.

Adresse	Nom de var.	Valeur de var.
0x0160	pi	3.14
0x0161		
0x0162		
0x0163		
0x0164	e	?
0x0165		
0x0166		
0x0167		
0x0168	x	3.14
0x0169		
0x016A		
0x016B		
0x016C	res	3
0x016D		
0x016E		
0x016F		
0x0170		
0x0171		
0x0172		

```

1  int trunc (float x)
2  {
3      int res = (int)x;
4      return res;
5  }
6  ...
7  float pi = 3.14;
8  int e;
9  e = trunc(pi);

```

La définition de la fonction (lignes 1 à 5) n'est pas évaluée en premier lors de l'exécution. La première ligne exécutée est donc la ligne 7 avec la déclaration d'une variable de type `float` suivi par la déclaration d'une variable entière `e`. Aucune valeur particulière n'est affectée à `e` lors de sa déclaration, sa valeur est donc inconnue.

L'appel de la fonction est réalisé sur la ligne 9. Lors de cet appel, le code « saute » à l'endroit de la déclaration de la fonction, ligne 1. La variable `x` est alors déclarée et prend la valeur de la variable `pi`. Le reste de la fonction se déroule avec la déclaration de `res` et son affectation. En arrivant sur la ligne 4, le code « saute » à nouveau sur l'appel de la fonction, ligne 9, et affecte finalement le résultat retourné (valeur de `res`) à la variable `e`.

Résumons l'ordre d'exécution de ce court extrait de code et son effet sur la mémoire :

- ligne 7 puis 8 : déclarations de deux variables ;
- ligne 9 : appel de fonction ;
- ligne 1 : déclaration du paramètre `x` ;
- ligne 3 : déclaration d'une variable ;
- ligne 4 : retour du résultat ;
- ligne 9 : affectation du résultat à la variable `res`.

1. Oui, cette phrase semble ne pas avoir de sens, mais en fait si un peu quand même.

A retenir. Lors d'un appel de fonction, seule la valeur de la variable est donnée à la fonction, pas la variable elle-même.

2.2 Portée des variables

Votre programme ayant une durée de vie limitée (son temps d'exécution sur l'ordinateur), les variables ont naturellement une espérance de vie qui ne dépasse pas la durée de vie du programme. En réalité, la plupart ont une durée de vie bien plus courte. En plus de cette vie relativement courte, une variable n'a qu'une portée locale. C'est à dire qu'elle n'est accessible que dans une portion de code qui l'entoure.

Définition 2 (Portée des variables). La *portée* d'une variable indique la zone de code dans laquelle elle est accessible.


Règles. En règle générale, une variable n'est accessible qu'entre les accolades entre lesquelles elle est définie.

La durée de vie de la variable est ainsi très corrélée à sa portée : elle « vit » dans une zone délimitée par des accolades, et est accessible uniquement dans cette zone. En dehors de cette zone, la variable n'existe pas. Quelques exceptions existent comme les variables globales ou celles définies dans un autre fichier mais nous ignorons pour le moment ces cas plus complexes.

Exemple 3. Reprenons une nouvelle fois le code précédent en le complétant d'un `main()`.

```
1  int trunc (float x)
2  {
3      int res = (int)x;
4      return res;
5  }
```

```
7  int main()
8  {
9      float pi = 3.14;
10     int e;
11     e = trunc(pi);
12     return 0;
13 }
```

La variable `x` définie ligne 1 est accessible dans toute la fonction (lignes 1 à 5). La variable `res` définie ligne 3 est aussi accessible ligne 4. Les variables `pi` et `e` sont quant à elles uniquement accessibles dans le `main()`. 

A retenir. Une variable est accessible seulement entre les accolades entre lesquelles elle est définie.

2.3 Nom des variables

La propriété de portée des variables implique indirectement une nouvelle possibilité : deux variables peuvent avoir le même nom si elles n'ont pas la même portée. Autrement dit, il est possible de déclarer une variable `n` dans une fonction et une autre dans le `main()`. N'étant pas définies au même endroit, il n'y a aucune confusion possible. En règle générale, il est bon d'éviter les noms identiques, sauf pour des cas particuliers :


- une variable de boucle se nomme souvent `i` ou `j` ;

- une variable permettant de retourner une valeur dans une fonction peut se nommer `res` ;
- des variables au nom générique comme `size`, `count` ou `length` peuvent être utilisées régulièrement.

Exemple 4. Reprenons une nouvelle fois le code précédent en le modifiant pour y introduire des noms de variable identiques.

```
1 int trunc (float x)
2 {
3     int res = (int)x;
4     return res;
5 }
```

```
7 int main()
8 {
9     float x = 3.14;
10    int res;
11    res = trunc(x);
12    return 0;
13 }
```

Nous retrouvons dans ce code deux variables nommées `x` et deux variables nommées `res`. Chacune vit dans sa propre zone délimitée par des accolades, il n'y a pas de confusion possible. 

TD Exercices 4 à 6.

3 Compilation et fichier d'entête

Pour que le compilateur réalise correctement son travail, il doit connaître la définition d'une fonction avant de rencontrer son appel. Aussi jusqu'à présent, il était obligatoire de déclarer la fonction en amont son appel dans le même fichier. Pour autant, vous parvenez tout de même à utiliser correctement la fonction `printf()` alors que vous n'avez pas rédigé la déclaration de cette dernière...

Le secret réside dans l'utilisation de bibliothèques.² Ces dernières contiennent un ensemble de fonctions déjà compilées qu'il est possible d'appeler depuis votre programme, sous condition de fournir au compilateur la version compilée de la fonction ainsi que son prototype. La version compilée et l'entête de la fonction sont habituellement déjà présents dans votre système (souvent dans les dossiers `/usr/lib` et `/usr/include`). Afin d'utiliser une fonction d'une bibliothèque, il faut donc respecter les étapes suivantes.

3.1 Inclusion de fichier d'entête

La première étape consiste à prévenir le compilateur qu'une fonction non définie dans le code source va être utilisée. On copie alors le prototype de la fonction voulue à l'aide de la directive `#include <lib.h>` en remplaçant `lib` par le nom de la bibliothèque contenant la fonction. Cette instruction a pour effet de copier l'intégralité du fichier `.h` dans notre fichier source lors de l'étape de *pré-processeur* (absente du schéma 1). Il est maintenant possible d'appeler les fonction de la bibliothèque n'importe où dans le fichier.

2. Version française du terme anglais *library*.

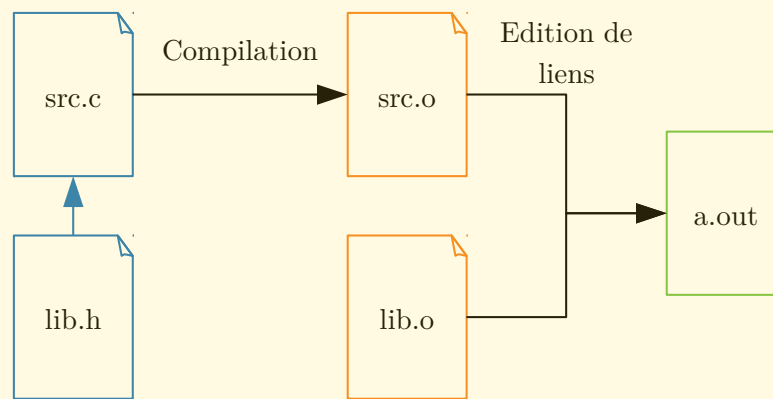


FIGURE 1 – Compilation basique utilisant une bibliothèque interne.

3.2 Compilation et édition de liens

La deuxième étape est plus simple pour l'utilisateur puisque seul le compilateur travaille. Une fois le code terminé, l'étape de *compilation* commence. Le compilateur traduit le code source en code machine³ et génère un fichier `.o` contenant le binaire correspondant au fichier source. Ce binaire doit ensuite être relié au binaire de la bibliothèque contenant le code machine pour l'exécution de la fonction. Cette étape est réalisée lors de l'*édition de liens* : le compilateur fait la correspondance entre l'appel de fonction présent dans le fichier `src.o` et les instructions de la fonction éponyme présentes dans `lib.o`. Le fichier ainsi produit est votre binaire, prêt à être exécuté. Ces étapes sont résumées dans la figure 1.

3.3 Créer son propre fichier d'entête

Vous avez alors compris l'intérêt des bibliothèques : il s'agit d'un ensemble de fonctions qu'il est possible d'appeler depuis n'importe quel code source y faisant appel. Dès lors, rien ne vous empêche de développer votre propre bibliothèque. Il vous faut alors réaliser les éléments suivants :

- un fichier `.c` contenant la déclaration des fonctions de votre bibliothèque ;
- un fichier `.h` contenant les prototypes des fonctions du fichier `.c` ;
- un fichier `main.c` permettant de tester les fonctions de votre bibliothèque.

Comme on peut l'observer sur la figure 2, il est même possible de réaliser plusieurs couples de fichiers `.c/.h` si vous souhaitez séparer votre bibliothèque en plusieurs parties. Lors de la compilation, chaque couple de fichiers `.c/.h` va générer un fichier `.o` puis l'ensemble des fichiers `.o` sont reliés pour former le binaire final. Dans ce cas, il convient de prendre certaines précautions.

On peut observer que le fichier `main.c` peut inclure plusieurs `.h`, eux-mêmes inclus par les fichiers de leur bibliothèque respective. Si aucune précaution n'est prise, les définitions des fonctions rédigées dans les fichiers entêtes se trouvent intégrées dans plusieurs fichiers `.o`. Lors de l'édition de lien, un conflit peut alors survenir car le compilateur risque de voir plusieurs définitions de fonctions identiques. Pour éviter ce problème, on peut rédiger les lignes de code suivantes dans chaque fichier d'entête.

3. Avec une transition vers l'assembleur entre les deux.

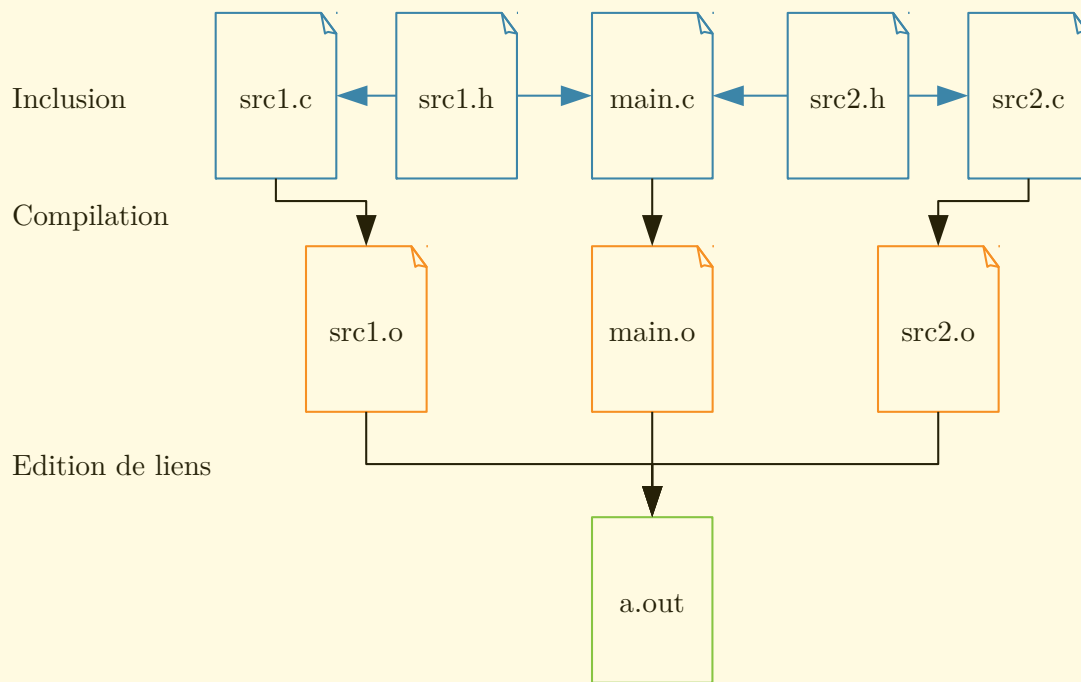


FIGURE 2 – Compilation séparée.

Lignes à inclure dans tout fichier d'entête

```
#ifndef _UN_NOM_DE_CONSTANTE_UNIQUE
#define _UN_NOM_DE_CONSTANTE_UNIQUE

// Prototypes des fonctions

#endif
```

TD Exercices 7 et 8.

4 Point d'entrée du programme

Le `main()` est une fonction (presque) comme les autres. Elle est également nommée *point d'entrée* car la première instruction réalisée lorsque votre programme est lancé se situe à la première ligne du `main()`. Comme toute fonction, elle possède donc un type de retour (`int`) et un nom (`main`). A la différence des autres fonctions, ses paramètres sont optionnels. Nous ne détaillerons pas cette notion d'*optionnel* mais nous allons étudier l'utilité de ces paramètres.

Prototype complet du `main()`

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

Le prototype complet du `main()` est donc donné ci dessus. La valeur des arguments du `main()` est automatiquement affecté lors du lancement de l'exécutable. Vous n'avez donc pas à modifier leur valeur. Détaillons maintenant ce que contient chaque argument :

- `int argc` contient le nombre d'arguments de la ligne de commande ;

- `char **argv` contient la liste des mots donnés sur la ligne de commande.

En pratique, lorsqu'on lance un exécutable, il est possible d'ajouter au nom de l'exécutable une série d'arguments. Vous l'avez par exemple fait lorsque vous avez utilisé la commande `chmod`. Lorsque vous lancez l'exécutable que vous avez réalisé, l'intégralité des mots présents sur la ligne de commande est passée en paramètres du `main()`. Il est donc possible de réutiliser ces arguments dans votre programme.

Exemple 5. Étudions le code suivant.

Compte le nombre de "mot" ici 3

```
int main (int argc, char **argv)
{
    if (argc<3) // Vérification du nombre d'args de la ligne de
        commande
    {
        printf("Pas assez d'arguments sur la ligne de commande!\n");
        return EXIT_FAILURE;
    }
    printf("Bonjour %s, vous avez %d ans.\n", argv[1], atoi(argv[2]));
    return EXIT_SUCCESS;
}
```

tableau de tableau

chaîne de caractères

remplace un nombre

Supposons que ce programme soit correctement compilé et que nous l'exécutons avec les paramètres suivants.

3

```
nibijau@TeamIbijau:~$ ./prog.exe potoobird 113
Bonjour potoobird, vous avez 113 ans.
nibijau@TeamIbijau:~$
```

Nous observons que le contenu de la ligne de commande a bien été récupéré. Puisque la ligne contient trois mots différents, l'instruction `(argc<3)` est fausse, donc on ne rentre pas dans le `if`. Le `printf` suivant vous permet d'afficher les arguments 1 et 2 de l'exécutable. Notez que le mot `potoobird` est affiché en l'état à l'aide d'un `%s`. La valeur 113 doit elle être interprétée en entier à l'aide de la fonction `atoi()` disponible dans la bibliothèque `stdlib.h`. En effet, chaque mot présent sur la ligne est interprété de base comme une chaîne de caractères, même les nombres! Nous verrons plus tard dans le semestre qu'il n'y a pas d'alternatives à `atoi()` pour convertir une chaîne de caractères en nombre. ▲

TD Exercices 9 à 12.