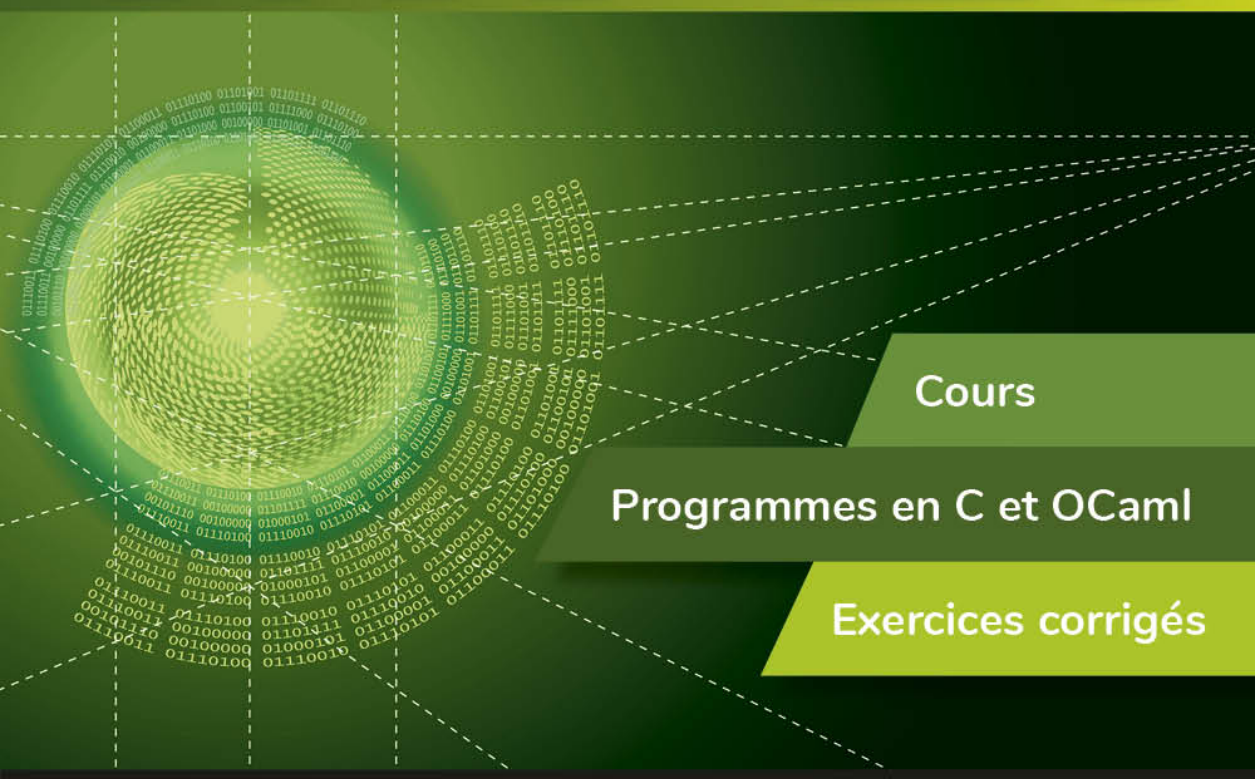
# Informatique MP2I



Cours

Programmes en C et OCaml

Exercices corrigés

Olivier Mallet

ellipses

Chapitre 1

# Bases de la programmation en C

Le langage C a été créé au début des années 1970 par l'informaticien américain Dennis Ritchie (1941–2011). Son développement est intimement lié à celui du système d'exploitation Unix. Le nom du langage vient du fait qu'il s'agit d'un successeur du langage B, mis au point par Ritchie et son collègue Ken Thompson.

La première spécification informelle du langage C a été l'ouvrage de Brian Kernighan et Dennis Ritchie *The C Programming Language* [12], paru en 1978 et familièrement appelé le « K&R » d'après les initiales de ses auteurs. Par la suite, plusieurs compilateurs C sont apparus sur différentes architectures, supportant chacun des extensions du langage qui pouvaient varier d'une implémentation à l'autre. Il est donc devenu nécessaire de standardiser le langage C. La première norme C a été publiée en 1989 par l'ANSI (American National Standards Institute) puis adoptée en 1990 par l'organisme international de normalisation ISO. Cette version est connue sous le nom de C89, C90, C ANSI, C ISO ou C standard. L'ISO a publié par la suite deux révisions de la norme C, en 1999 (norme C99) et en 2011 (norme C11). Les compilateurs actuels supportent la norme C89 mais pas forcément toutes les fonctionnalités introduites par les versions ultérieures. Les fragments de code C présentés dans cet ouvrage sont compatibles avec les normes C99 et C11.

Les principaux avantages du langage C sont sa syntaxe relativement simple et la proximité de ses instructions avec celles du langage machine, ce qui permet d'écrire des programmes s'exécutant rapidement. Parmi les inconvénients, on peut citer le peu de vérifications lors de la compilation et de l'exécution, ce qui favorise les bugs et les failles de sécurité (certaines options de compilation permettent quand même de vérifier la fiabilité du code). De plus, les structures de données classiques (listes, piles, files…) ne sont pas présentes nativement ; il existe toutefois des bibliothèques logicielles externes qui les implémentent.

En raison de la popularité du C et de son efficacité, de nombreux logiciels sont écrits dans ce langage : noyau Linux, compilateurs ou interpréteurs pour d'autres langages (Python, PHP…), serveurs web (Apache, nginx), systèmes de gestion de bases de données (PostgreSQL, SQLite), outils de calcul scientifique (Maple,

MATLAB), pour n'en citer que quelques-uns. De plus, de nombreux langages de programmation, tels que C++, C#, Java, JavaScript ou PHP, ont une syntaxe inspirée de celle du C.

Pour approfondir l'étude du langage C, on pourra se reporter aux ouvrages utilisés pour écrire ce chapitre [3, 9, 12, 14, 16].

## 1.1 Généralités

### 1.1.1 Un premier exemple

Commençons par un exemple inspiré du fameux *Hello, World !* de Kernighan et Ritchie :

```
1 #include <stdio.h>
2
3 int main(void) {
4     printf("Coucou les MP2I !\n");
5     return 0;
6 }
```

La première ligne est constituée d'une directive **`#include`**. Celle-ci sert à inclure un fichier d'en-tête (voir la section 1.7 pour plus de détails). Dans cet exemple, il est nécessaire d'inclure le fichier `stdio.h` pour pouvoir utiliser la fonction `printf` à la ligne 4. Le nom du fichier d'en-tête est ici entre chevrons, ce qui indique qu'il fait partie de la bibliothèque standard du langage C. Les fichiers de cette bibliothèque que nous aurons l'occasion d'utiliser sont les suivants :

- `assert.h` (vérification d'assertions ; voir la section 3.5.2) ;
- `stdbool.h` (définition du type booléen ; voir la section 1.2.4) ;
- `stddef.h` (définition de types et de macros) ;
- `stdint.h` (définition de différents types entiers) ;
- `stdio.h` (fonctions d'entrée-sortie) ;
- `stdlib.h` (fonctions standards) ;
- `string.h` (fonctions sur les chaînes de caractères ; voir la section 1.5.2).

Le reste du programme (lignes 3 à 6) contient le code d'une fonction nommée `main`. C'est la fonction principale du programme [(1)], qui sera appelée au lancement de celui-ci (voir la section 1.1.5 pour plus de détails sur la fonction `main`). Les instructions qui composent le corps de cette fonction sont encadrées par des accolades (`{` et `}`).

Les lignes 4 et 5 sont constituées chacune d'une instruction. En C, une instruction se termine toujours par un point-virgule. Il est possible d'écrire plusieurs instructions sur une même ligne : ainsi, on aurait pu remplacer les lignes 4 et 5 par une unique ligne contenant

```
printf("Coucou les MP2I !\n");return 0;
```

(1). D'où son nom : l'adjectif *main* signifie « principal » en anglais.

Toutefois, pour des raisons de lisibilité, il est conseillé de mettre chaque instruction sur une ligne différente.

À la ligne 4, on appelle une fonction nommée `printf`. Celle-ci affiche son argument, en l'occurrence la chaîne de caractères `"Coucou les MP2I !\n"`. Dans cet exemple, la chaîne se termine par la suite de caractères `\n` qui représente un saut de ligne : le programme écrit donc

```
Coucou les MP2I !
```

à l'écran puis passe à la ligne suivante.

La ligne 5 indique que la fonction `main` renvoie 0. Cette ligne est présente car la fonction doit renvoyer une valeur entière mais elle n'a pas d'incidence sur l'exécution du programme. La valeur 0 signifie que le programme s'est déroulé normalement ; on peut renvoyer une valeur différente pour indiquer une situation anormale.

Pour pouvoir exécuter ce programme, il est nécessaire de *compiler* le fichier contenant le code source précédent, c'est-à-dire de transformer ce code source en un code directement exécutable par un ordinateur. Il existe différents programmes, appelés *compilateurs*, permettant de faire cela. Dans la suite de cet ouvrage, on supposera qu'on travaille sous Linux et on utilisera GCC, qui est le compilateur standard pour ce système d'exploitation. Si le code source précédent se trouve dans un fichier nommé `exemple.c` et qu'on souhaite obtenir un fichier exécutable nommé `exemple`, on tapera dans un terminal Linux la commande

```
gcc -o exemple exemple.c
```

Dans cet exemple, il n'y a qu'un seul fichier à compiler mais un programme complexe peut être constitué de plusieurs fichiers ; il est alors nécessaire de tous les compiler pou obtenir un programme exécutable (voir la section 1.7).

Il est possible de paramétrer le comportement du compilateur à l'aide d'options. Par exemple, `-o` est une option permettant de choisir le nom du fichier de sortie ; sans cette option, celui-ci aura un nom par défaut qui dépend du système (par exemple `a.out` sous Linux).

La compilation d'un fichier source C comprend plusieurs étapes.

1. Tout d'abord, un programme appelé *préprocesseur*, intégré au compilateur, effectue un premier traitement du code source. Ce traitement est notamment défini par les *directives* (commandes spéciales à destination du préprocesseur commençant par un **#**, comme **`#include`**). Dans notre exemple, c'est lors de cette phase que le fichier d'en-tête `stdio.h` va être inclus.
2. Ensuite, le compilateur analyse le code source et vérifie qu'il respecte les règles syntaxiques et sémantiques du langage. Si ce n'est pas le cas, la compilation échoue et un message d'erreur s'affiche. Dans certaines situations, en fonction des options qui lui sont passées, le compilateur peut aussi afficher un avertissement pour attirer l'attention du programmeur sur une construction autorisée par la norme mais qui risque de produire un comportement inattendu. Il est recommandé d'utiliser l'option `-Wall` de GCC, qui permet d'afficher la plupart de ces avertissements.

3. Si le code source est correct, le compilateur le transforme en un fichier objet, qui est une suite d'instructions écrites en langage machine.
4. La dernière étape est appelée *édition de liens*. Les différents fichiers objets (ceux obtenus à partir des fichiers sources du programme et ceux correspondant aux bibliothèques externes) sont rassemblés pour produire le programme exécutable final.

Une fois la compilation terminée, on peut exécuter le programme. Dans notre exemple, le fichier exécutable s'appelle `exemple` ; il suffit donc, pour l'exécuter, de taper dans un terminal

```
./exemple
```

ce qui affichera à l'écran

```
Coucou les MP2I !
```

suivi d'un saut de ligne.

On peut connaître la valeur de retour du programme, c'est-à-dire la valeur renvoyée par la fonction `main`, en tapant dans un terminal la commande

```
echo $?
```

ce qui affichera 0 dans notre exemple.

Par la suite, on présentera souvent des fragments de code ne correspondant pas forcément à un programme complet. Il faut bien retenir que toute instruction (à part les déclarations de variables globales) doit être contenue dans une fonction et que tout programme doit contenir une fonction `main` (il peut contenir éventuellement d'autres fonctions ; voir la section 1.6).

### 1.1.2 Variables et types

Le langage C est *typé statiquement* : le type des variables est connu dès la compilation, de même que le type de retour des fonctions. Quand on déclare une variable, il faut indiquer son type : par exemple le code

```
int x;
```

déclare une variable nommée `x` qui est de type **`int`** (l'un des types représentant des entiers ; voir la section 1.2). On peut ensuite donner une valeur à cette variable. Ainsi, on pourra écrire

```
int x;
x = 42;
```

On peut aussi déclarer la variable (toujours en indiquant son type) et l'initialiser en une seule instruction :

```
int x = 42;
```

De même, quand on déclare une fonction, il faut indiquer le type de ses arguments et celui de la valeur qu'elle renvoie (voir la section 1.6).

Le nom d'une variable ne peut comporter que des lettres non accentuées (majuscules ou minuscules), des chiffres ou des blancs soulignés (_). Il ne peut commencer par un chiffre et il est déconseillé de le faire commencer par un blanc souligné pour éviter des conflits avec d'éventuelles bibliothèques logicielles extérieures. Par exemple, `toto`, `Truc42`, `ma_Variable`, `vol_714` sont des noms de variables valides, contrairement à `prénom` ou à `1variable`. De plus, le nom d'une variable ne doit pas être un mot-clé réservé du langage, comme par exemple **`const`**, **`double`**, **`if`** ou **`struct`**. Enfin, il faut remarquer que le langage C différencie les majuscules et les minuscules dans les noms de variables (on dit que ces derniers sont *sensibles à la casse*) : ainsi, les identificateurs `var`, `Var` et `vaR` désignent trois variables distinctes.

Une variable peut être globale (déclarée en dehors de toute fonction) ou locale (déclarée à l'intérieur d'un bloc délimité par des accolades). On peut faire référence à une variable locale à partir de la ligne où elle est déclarée jusqu'à la fin du programme, tandis qu'une variable locale ne peut être référencée qu'entre sa déclaration et la fin du bloc où elle est déclarée. Ainsi, dans le fragment de code suivant, une instruction qui affiche la valeur de `x` pourrait être ajoutée n'importe où après la déclaration de cette variable. En revanche, une instruction qui affiche la valeur de `y` ne pourrait être ajoutée qu'entre les accolades :

```
int x = 42;

{
    int y = 0;
}
```

Une variable locale peut porter le même nom qu'une variable globale ou qu'une variable globale définie dans un bloc englobant. Considérons par exemple le code suivant :

```
1  int x = 0 ;
2
3  {
4      int x = 1;
5
6      {
7          int x = 2;
8
9      }
10
11 }
12
```

Si on écrit à la ligne 2 une instruction qui affiche la valeur de `x`, le programme affichera 0, tandis que si cette instruction se place à la ligne 5, elle affichera 1. Enfin, si c'est à la ligne 8 qu'on affiche la valeur de `x`, on obtiendra 2. On dit qu'une variable locale à un bloc *masque* une variable de même nom et définie

précédemment à l'extérieur du bloc. Ce phénomène de masquage cesse à la sortie du bloc interne : ainsi, x vaut à nouveau 1 à la ligne 10 et 0 à la ligne 12.

On peut afficher la valeur d'une ou plusieurs variables (ou expressions) à l'aide de printf. La syntaxe générale de cette fonction est

```
printf(format, expr_1, expr_2, ...)
```

où expr_1, expr_2, ... sont des expressions (il peut y en avoir un nombre arbitraire) et format est une chaîne de caractères, dite *chaîne de format*, qui peut contenir des suites de caractères spéciales (codes de format) correspondant aux différents types du langage. Voici les plus courants de ces codes :

**%d** entier signé

**%u** entier non signé

**%f** réel en notation classique (par exemple, `printf("%f\n", 12.3);` affiche 12.300000)

**%e** réel en notation scientifique (par exemple, `printf("%e\n", 12.3);` affiche 1.230000e+01)

**%c** caractère

**%s** chaîne de caractères

**%p** pointeur (adresse mémoire sous forme d'un nombre en base 16 ; voir la section 1.4)

**%%** caractère %

À l'affichage, le $i^{\text{e}}$ code de format apparaissant dans format est remplacé par la valeur de l'expression expr_i. Par exemple, le code

```
int x = 42;
int y = 13;
printf("x vaut %d et y vaut %d\n", x, y);
```

affiche « x vaut 42 et y vaut 13 ».

Pour initialiser des variables à des valeurs entrées au clavier par l'utilisateur, on utilise la fonction scanf, qui est en quelque sorte l'inverse de printf. Cette fonction est aussi définie dans stdio.h. La syntaxe générale de scanf est

```
scanf(format, &var_1, &var_2, ...)
```

où format est une chaîne de format, comme dans printf, et &var_1, &var_2, ... sont des adresses mémoire de variables (il ne faut pas oublier le caractère &, qui indique qu'il s'agit d'une adresse). Les variables var_1, var2, ... doivent être déclarées avant l'appel à scanf. S'il y a $n$ codes de format dans la chaîne format, la fonction scanf lit $n$ valeurs entrées par l'utilisateur et pour tout $i$ entre 1 et $n$, initialise la variable var_i avec la $i^{\text{e}}$ valeur entrée. Si le type de var_i ne correspond pas au $i^{\text{e}}$ code de format, cette variable n'est pas initialisée et la fonction s'arrête : les variables var_(i+1), var_(i+2)... ne seront donc pas initialisées non plus.

*Remarque.* Les codes de format ne sont pas exactement les mêmes pour printf et pour scanf. En particulier, pour les nombres à virgule étudiés dans cet ouvrage (type **double**), on utilisera le code %f pour printf et %lf pour scanf.

Par exemple, le programme suivant utilise `scanf` pour demander à l'utilisateur d'entrer deux entiers et affiche ensuite leur somme :

```
#include <stdio.h>

int main(void) {
    int x;
    int y;
    printf("Entrez deux nombres (par exemple 7 12)\n");
    scanf("%d %d", &x, &y);
    printf("La somme de %d et %d est %d\n", x, y, x + y);
    return 0;
}
```

Voici un exemple d'exécution de ce programme :

```
Entrez deux nombres (par exemple 7 12)
42 13
La somme de 42 et 13 est 55
```

Quand on utilise `scanf`, la lecture d'une information correspondant à un code de format commence au premier caractère autre qu'un espace blanc (espace, saut de ligne, tabulation, fin de page), sauf pour le code `%c` qui lit tous les caractères. Dans le programme précédent, on peut donc écrire

```
scanf("%d%d", &x, &y);
```

au lieu de

```
scanf("%d %d", &x, &y);
```

Dans les deux cas, si l'utilisateur tape deux entiers séparés par un nombre quelconque d'espaces, les variables `x` et `y` seront bien initialisées.

La fonction `scanf` renvoie le nombre de variables initialisées ou la valeur `EOF` (*End Of File*) en cas d'erreur. Considérons par exemple le code suivant :

```
int a;
int b;
printf("Entrez deux nombres a et b\n");
int retour = scanf("%d%d", &a , &b);
printf("Nombre de valeurs lues : %d\n", retour);
```

Voici quelques exemples d'exécution :

— si l'utilisateur tape `42 13`, les variables `a` et `b` sont bien initialisées et le programme affiche « Nombre de valeurs lues : 2 » ;

— s'il tape `42 bonjour`, la variable `a` est bien initialisée mais pas `b` puisque `bonjour` ne correspond pas à un entier ; le programme affiche donc « Nombre de valeurs lues : 1 » ;

— s'il tape `bonjour 42`, la variable `a` n'est pas initialisée car `bonjour` ne correspond pas à un entier ; la fonction s'arrête donc et `b` n'est pas initialisée non plus. Le programme donc affiche « Nombre de valeurs lues : 0 ».

### 1.1.3 Retours à la ligne et indentation

Contrairement à d'autres langages, comme Python, les retours à la ligne et les espaces ne sont pas porteurs de sens. Par exemple, le code suivant

```
int f(int n) {
    if (n == 0)
        return 1;
    else
        return n * f (n-1);
}
```

est équivalent au code suivant :

```
int f(int n){if(n==0)return 1;else return n*f(n-1);}
```

Bien entendu, pour mettre en évidence la structure de vos programmes et pouvoir les relire facilement, il est recommandé de passer à la ligne et d'indenter le code autant que nécessaire.

### 1.1.4 Commentaires

Les commentaires servent à expliquer le fonctionnement d'un programme afin d'aider les lecteurs du code source à le comprendre. Ils n'ont pas d'incidence sur le fonctionnement du programme (le préprocesseur supprime les commentaires avant la compilation proprement dite). En C, il existe deux types de commentaires :

— les commentaires commençant par `//` s'arrêtent à la fin de la ligne ;

— les commentaires encadrés par `/*` et `*/` peuvent s'étendre sur une ou plusieurs lignes.

Il n'est pas possible d'imbriquer un commentaire dans un autre.

L'exemple suivant utilise ces deux syntaxes :

```
/* Fonction qui calcule le double
   de son argument */
int doubler(int x) { // x est un entier
    return 2 * x;
}
```

### 1.1.5 La fonction `main`

Tout programme possède une et une seule fonction principale, nommée `main`, qui est appelée au lancement du programme. Cette fonction peut s'écrire :

— soit sans arguments : `int main(void)` ;

— soit avec deux arguments : le premier est un entier et le deuxième un tableau de chaînes de caractères `int main(int argc, char* argv[])`.

Dans le deuxième cas, les arguments de la fonction sont initialisés à partir de la ligne de commande :