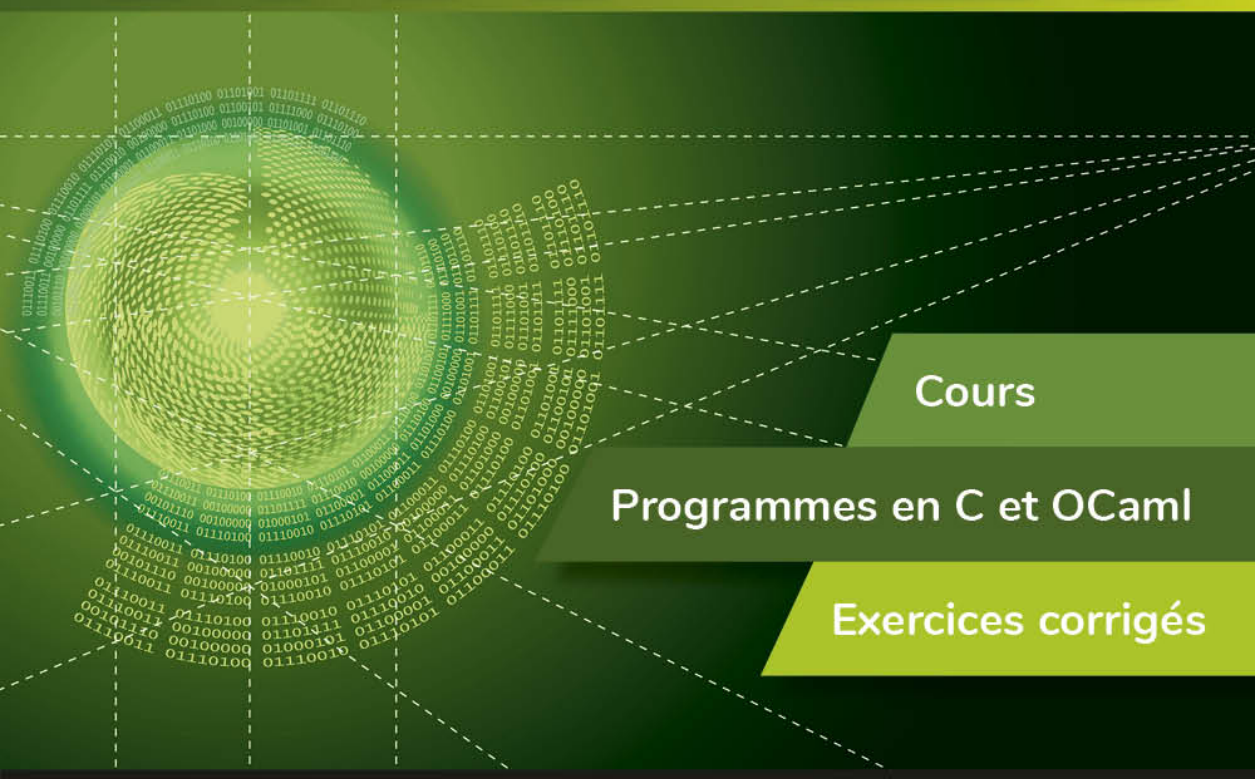
Informatique MP2I



Cours

Programmes en C et OCaml

Exercices corrigés

Olivier Mallet

ellipses

Informatique

MP2I

Cours
Programmes en C et OCaml
Exercices corrigés

Olivier Mallet

ellipses

Conception graphique couverture : Nathalie FOULLOY
Mise en pages : Louise PRÉAULT,

ISBN 9782340-083400
©Ellipses Édition Marketing S.A., 2023
8/10 rue la Quintinie 75015 Paris



Le Code de la propriété intellectuelle n'autorisant, aux termes de l'article L. 122-5.2° et 3°a), d'une part, que les « copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective », et d'autre part, que les analyses et les courtes citations dans un but d'exemple et d'illustration, « toute représentation ou reproduction intégrale ou partielle faite sans le consentement de l'auteur ou de ses ayants droit ou ayants cause est illicite » (art. L. 122-4).
Cette représentation ou reproduction, par quelque procédé que ce soit constituerait une contrefaçon sanctionnée par les articles L. 335-2 et suivants du Code de la propriété intellectuelle.

www.editions-ellipses.fr

Avant-propos

À l'époque où j'étais en classe préparatoire, le programme officiel de l'option informatique en MPSI commençait par la phrase « L'informatique est considérée ici comme une science opérant sur des représentations rigoureuses de concepts bien définis ». Dans la préface d'un livre que j'utilisais à l'époque [2], Daniel Krob déplorait le manque de considération pour la discipline informatique que cette formulation traduisait. Il pointait aussi l'absence de nombreux aspects de l'informatique au sein des programmes de classes préparatoires.

Plus de vingt ans se sont écoulés depuis lors et l'informatique a trouvé une place plus convenable au sein de l'enseignement secondaire et supérieur avec notamment la création de la spécialité NSI au lycée, celle du CAPES et de l'agrégation d'informatique et bien sûr l'ouverture de la filière MP2I/MPI des classes préparatoires scientifiques, pour laquelle a été conçu le présent ouvrage.

Cette évolution était indispensable au vu de la place croissante que l'informatique a prise, au cours de cette période, dans les différents secteurs économiques et dans notre vie en général, notamment avec le développement d'Internet. L'élève qui entre aujourd'hui en classe préparatoire devra, sans aucun doute, utiliser l'informatique dans le cadre de son métier. Mais il devra aussi être conscient, à la fois en tant que professionnel et en tant que citoyen, des nombreux enjeux liés à son usage : protection des données personnelles et de la vie privée, liberté d'expression en ligne, rôle des algorithmes de recommandation dans la diffusion des discours de haine et de la désinformation, impact de l'informatique sur l'environnement et la consommation globale d'énergie, sans parler des problèmes que soulèvent les progrès récents de l'intelligence artificielle. L'importance de ces questions montre qu'une certaine connaissance de l'informatique est nécessaire à la formation de l'« honnête homme » d'aujourd'hui. J'espère que cet ouvrage pourra, à son échelle, contribuer à cette formation… et bien sûr aider ses lecteurs à réussir les concours !

Contenu de ce livre

Les deux premiers chapitres de cet ouvrage sont une introduction aux langages C et OCaml. On y présente l'ensemble des notions au programme de la filière

MPI, à l'exception de celles relevant spécifiquement du programme de deuxième année (fils d'exécution, mutex et sémaphores). Le fait de commencer par présenter les deux langages au programme permet d'utiliser exclusivement ceux-ci dans les exemples des chapitres suivants. Ainsi, seules des notions générales en programmation (variables, tests conditionnels, boucles) sont requises pour aborder le présent ouvrage ; il ne s'appuie pas sur un langage de programmation particulier qui serait supposé connu.

Les chapitres suivants correspondent à ceux du programme officiel de première année : méthodes de programmation (chapitre 3), récursivité et induction (chapitre 4), structures de données (chapitre 5), algorithmique (chapitre 6), gestion des ressources de la machine (chapitre 7), logique (chapitre 8) et bases de données (chapitre 9). Ces chapitres sont classés dans l'ordre du programme et non dans l'ordre de progression du cours. Il peut donc arriver qu'un chapitre utilise une notion que le lecteur n'a pas encore rencontrée ; dans ce cas, il pourra facilement retrouver la définition correspondante à l'aide de l'index ou des renvois dans le corps du texte. Chacun de ces sept chapitres comprend un cours détaillé, où chaque notion est illustrée par un ou plusieurs exemples, et des exercices, dont les corrigés se trouvent en fin d'ouvrage. Au sein d'un même chapitre, l'ordre des exercices correspond à l'ordre dans lequel les notions ont été abordées dans le cours ; les exercices ne sont pas classés selon leur difficulté.

On a indiqué, en particulier au début de certains chapitres ou sections, des références bibliographiques qui permettront aux lecteurs intéressés d'approfondir les notions présentées ici. Par ailleurs, tout au long de ce livre, de nombreuses notes de bas de page donnent des explications sur l'étymologie de certains termes, présentent brièvement les scientifiques mentionnés dans l'ouvrage (notamment ceux ayant donné leur nom à un algorithme, un théorème ou une notion), apportent des compléments scientifiques ou techniques et fournissent la traduction anglaise des principaux termes, afin de faciliter la recherche dans des ressources anglophones.

Notations

Précisons quelques notations utilisées dans ce livre, en dehors de celles définies dans les différents chapitres :

$\lceil x \rceil$ partie entière supérieure du nombre x : c'est le plus petit entier n tel que $x \leqslant n$ (par exemple, $\lceil 3{,}14 \rceil = 4$)

$\lfloor x \rfloor$ partie entière inférieure du nombre x : c'est le plus grand entier n tel que $n \leqslant x$ (par exemple, $\lfloor 3{,}14 \rfloor = 3$)

$\varnothing$ ensemble vide

i unité imaginaire ; on a $\mathrm{i}^2 = -1$

$\leftarrow$ opérateur d'affectation dans les algorithmes écrits en pseudo-langage (voir la section 3.1.1) : ainsi, $\mathsf{x} \leftarrow 2$ signifie que la variable x reçoit la valeur 2

$\log x$ logarithme binaire de x, défini pour $x > 0$ par $\log x = \frac{\ln x}{\ln 2}$ où ln est le logarithme népérien ; on a $y = \log x$ si et seulement si $x = 2^y$

$\mathbb{N}$ ensemble des entiers naturels : $\mathbb{N} = \{0; 1, 2; \ldots\}$

$\mathbb{R}$ ensemble des nombres réels

Remerciements

Tout d'abord, je remercie Corinne Baud, des éditions Ellipses, de m'avoir fait confiance et de m'avoir encouragé au cours de l'écriture de ce livre.

Je tiens ensuite à signaler que cet ouvrage a grandement bénéficié des remarques et suggestions effectuées par mes collègues, qui ont relu la majeure partie des chapitres. Je remercie donc chaleureusement Nicolas Bedon, Giovanna Guaiana, Yannick Guesnet, Éric Laugerotte, Thierry Lecroq, Arnaud Lefebvre, Bruno Patrou, Christopher Rousseau et Carla Selmi pour leur aide précieuse.

Enfin, au cours des quelque deux ans et demi que j'ai passé à écrire ce livre, j'ai été soutenu par mon épouse Ana, mon fils aîné Daniel et mon second fils Victor, qui est né pendant cette période. Je les remercie du fond du cœur pour leur amour et leur patience.

Table des matières

1 Bases de la programmation en C **1**
- 1.1 Généralités . . . 2
- 1.2 Définitions et types de base . . . 9
- 1.3 Structures de contrôle . . . 15
- 1.4 Pointeurs . . . 19
- 1.5 Types structurés . . . 20
- 1.6 Fonctions . . . 30
- 1.7 Compilation séparée . . . 38

2 Bases de la programmation en OCaml **43**
- 2.1 Généralités . . . 44
- 2.2 Définitions et types de base . . . 45
- 2.3 Filtrage . . . 61
- 2.4 Types structurés . . . 67
- 2.5 Exceptions . . . 88
- 2.6 Programmation impérative . . . 90
- 2.7 Programmes compilés . . . 93

3 Méthodes de programmation **97**
- 3.1 Algorithmes, programmes, langages . . . 97
- 3.2 Représentation des nombres . . . 101
- 3.3 Terminaison et correction d'un programme . . . 112
- 3.4 Complexité d'un algorithme . . . 114
- 3.5 Discipline de programmation . . . 124
- 3.6 Validation, test . . . 129
- 3.7 Exercices . . . 136

4 Récursivité et induction **139**
- 4.1 Récursivité . . . 139
- 4.2 Ensembles ordonnés . . . 147
- 4.3 Induction . . . 154
- 4.4 Exercices . . . 160

5 Structures de données **165**
5.1 Types et abstraction . . . 165
5.2 Structures de données séquentielles . . . 171
5.3 Arbres . . . 211
5.4 Graphes . . . 250
5.5 Exercices . . . 273

6 Algorithmique **281**
6.1 Exploration exhaustive . . . 281
6.2 Décomposition en sous-problèmes . . . 285
6.3 Algorithmique des textes . . . 306
6.4 Algorithmique des graphes . . . 325
6.5 Exercices . . . 356

7 Gestion des ressources de la machine **367**
7.1 Gestion de la mémoire d'un programme . . . 367
7.2 Gestion des fichiers et entrées-sorties . . . 374
7.3 Exercices . . . 395

8 Logique **399**
8.1 Syntaxe des formules logiques . . . 399
8.2 Sémantique de vérité du calcul propositionnel . . . 407
8.3 Exercices . . . 425

9 Bases de données **429**
9.1 Le modèle relationnel . . . 429
9.2 Modélisation entité-association . . . 431
9.3 Généralités sur le langage SQL . . . 435
9.4 Requêtes `SELECT` sur une seule table . . . 437
9.5 Opérateurs ensemblistes . . . 443
9.6 Jointures . . . 446
9.7 Agrégation . . . 448
9.8 Exercices . . . 452

Solutions des exercices **455**

Bibliographie **527**

Index **529**

Index des mots-clés et fonctions C **535**

Index des mots-clés et fonctions OCaml **537**

Index des mots-clés et fonctions SQL **541**

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^{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^{e} valeur entrée. Si le type de var_i ne correspond pas au i^{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 :

— les éléments du tableau `argv` sont les paramètres fournis lors de l'appel du programme en ligne de commande (y compris le nom du programme, qui est stocké dans la case d'indice 0) ;
— l'entier `argc` est le nombre de ces paramètres, c'est-à-dire la taille du tableau `argv`.

Par exemple, si on appelle un programme C par la commande

```
mon_prog toto 1 bonjour
```

alors les arguments de `main` prendront les valeurs suivantes :

— `argc` vaudra 4 ;
— `argv[0]` vaudra `"mon_prog"` ;
— `argv[1]` vaudra `"toto"` ;
— `argv[2]` vaudra `"1"` ;
— `argv[3]` vaudra `"bonjour"`.

La fonction `main` renvoie une valeur de type **`int`** (entier). Par convention, on renvoie 0 (ou la constante `EXIT_SUCCESS`) en cas de succès et une valeur non nulle (ou la constante `EXIT_FAILURE`) en cas d'échec (ces deux constantes sont définies dans le fichier d'en-tête `stdlib.h`). La valeur renvoyée par `main` est transmise à l'environnement d'exécution, qui peut l'utiliser ou non.

1.2 Définitions et types de base

1.2.1 Les entiers

Il existe de nombreux types en C pour représenter des nombres entiers. Ces types diffèrent d'une part par leur taille (le nombre d'octets utilisés pour les coder) et d'autre part par leur caractère signé ou non signé : un type non signé ne peut représenter que des entiers positifs ou nuls alors qu'un type signé peut coder des entiers de signe quelconque.

La norme C impose que les entiers non signés, ainsi que les entiers signés positifs, soient codés par leur représentation en base 2. Le codage des entiers signés négatifs peut varier légèrement selon les implémentations : on supposera qu'ils sont représentés en complément à deux (voir la section 3.2.1), ce qui est généralement le cas.

Parmi les types entiers « classiques », on mentionnera :

— pour les entiers signés, le type **`int`**, dont la taille dépend de l'implémentation mais est au moins égale à 2 octets (16 bits) : on est donc sûr de pouvoir coder, au minimum, les entiers compris entre −32 768 et 32 767 ;
— pour les entiers signés, le type **`unsigned int`**, dont la taille dépend de l'implémentation mais est au moins égale à 2 octets : on est donc sûr de pouvoir coder, au minimum, les entiers compris entre 0 et 65 535.

La norme C99 a introduit de nouveaux types entiers de taille fixée, définis dans le fichier d'en-tête `stdint.h` :

— les types signés `int8_t`, `int32_t` et `int64_t` :

Type	Taille (bits)	Valeur minimale	Valeur maximale
`int8_t`	8	−128	127
`int32_t`	32	−2 147 483 648	2 147 483 647
`int64_t`	64	−9 223 372 036 854 775 808	9 223 372 036 854 775 807

D'une façon générale, le type `intN_t`, avec $N \in \{8; 32; 64\}$, permet de représenter les entiers entre -2^{N-1} et $2^{N-1} - 1$.

— les types non signés `uint8_t`, `uint32_t` et `uint64_t` :

Type	Taille (bits)	Valeur minimale	Valeur maximale
`uint8_t`	8	0	255
`uint32_t`	32	0	4 294 967 295
`uint64_t`	64	0	18 446 744 073 709 551 615

D'une façon générale, le type `uintN_t`, avec $N \in \{8; 32; 64\}$, permet de représenter les entiers entre 0 et $2^N - 1$.

Les quatre opérations arithmétiques élémentaires sur les entiers (`+`, `-`, `*` et `/`) sont disponibles avec leurs priorités habituelles :

```
printf("%d\n", 1 + 3 * 4 / 2 - 5); // affiche : 2
```

On notera que si `a` et `b` sont tous deux entiers, `a / b` correspond au quotient (entier) de la division euclidienne de `a` par `b` :

```
printf("%d\n", 7 / 2); // affiche : 3
```

Si `b` vaut 0, le comportement n'est pas défini par la norme C mais une division par zéro provoque généralement une erreur à l'exécution.

En cas de dépassement de capacité sur des entiers signés (le résultat théorique d'une opération est en dehors des bornes du type correspondant), le comportement n'est pas défini par la norme C. En général, le résultat est tronqué (on perd les bits de poids fort). Plus précisément, soit M le plus grand entier représentable dans le type utilisé (les entiers codables dans ce type sont donc compris entre $-M - 1$ et M) et r le résultat théorique. Le résultat effectif est alors l'unique entier x compris entre $-M - 1$ et M et congru à r modulo $2M + 2$.

Dans l'exemple suivant, qui utilise le type `int8_t`, on a $M = 127$. Le résultat théorique de l'opération est $r = 200$. L'unique entier compris entre -128 et 127 qui soit congru modulo à 200 modulo 256 est $x = -56$.

```
int8_t a = 100;
int8_t b = 100;
int8_t c = a + b; // résultat théorique : 200 (trop grand)
printf("%d\n", c); // affiche : - 56
```

En cas de dépassement de capacité sur des entiers non signés (le résultat théorique d'une opération est négatif ou bien trop grand pour le type correspondant), la norme définit le résultat de l'opération comme suit. Soit M le plus grand entier représentable dans le type utilisé et r le résultat théorique. Le résultat effectif est alors l'unique entier x compris entre 0 et M et congru à r modulo $M + 1$.

Dans l'exemple suivant, qui utilise le type `uint8_t`, on a $M = 255$. Pour la première opération, $r = 300$ donc $x = 44$. Pour la deuxième opération, $r = -100$ donc $x = 156$.

```
uint8_t a = 100;
uint8_t b = 200;
uint8_t c = a + b; // résultat théorique : 300 (trop grand)
uint8_t d = a - b; // résultat théorique : -100 (négatif)
printf("%d\n", c); // affiche : 44
printf("%d\n", d); // affiche : 156
```

Si `a` $\geqslant 0$ et `b` > 0, `a % b` calcule le reste de la division euclidienne de `a` par `b` :

```
printf("%d\n", 7 % 2); // affiche : 1
```

1.2.2 Les caractères

Le type **`char`** représente un caractère codé sur un octet. Les valeurs de type **`char`** sont encadrées par des apostrophes :

```
char caractere = 'A';
printf("%c\n", caractere); // affiche : A
```

Pour définir certains caractères (soit parce qu'ils ne sont pas imprimables, soit parce qu'ils jouent un rôle spécial dans le langage), on doit utiliser une *séquence d'échappement* commençant par `\`. Voici quelques-uns de ces caractères :

Séquence	Caractère
`'\0'`	Caractère nul (de code ASCII 0)
`'\n'`	Saut de ligne
`'\t'`	Tabulation
`'\\'`	\ (barre oblique inversée)
`'\''`	' (apostrophe)

1.2.3 Les nombres à virgule

Le type **`double`** représente un nombre à virgule, généralement codé sur 8 octets (64 bits). Une valeur de type **`double`** peut s'écrire avec une partie entière et une partie décimale séparées par un point (par exemple `3.14`) ; si l'une de ces deux parties est nulle, elle peut être omise (par exemple, `1.` est équivalent à `1.0` et `.618` à `0.618`). On peut aussi l'écrire en notation exponentielle : le nombre

$a \times 10^b$, où a est un entier ou un nombre à virgule et b un entier, s'écrira `aeb` ou `aEb` [(2)] :

```
double c = 3e8; // c = 3 × 10^8
double e = 1.602E-19; // e = 1,602 × 10^-19
printf("c = %f\n", c); // affiche : c = 300000000.000000
printf("c = %e\n", c); // affiche : c = 3.000000e+08
printf("e = %f\n", e); // affiche : e = 0.000000
printf("e = %.25f\n", e);
// affiche : e = 0.0000000000000000001602000
/* le code de format %.25f signifie qu'on affiche le nombre
   avec 25 décimales */
printf("e = %e\n", e);
// affiche : e = 1.602000e-19
```

Les quatre opérations arithmétiques élémentaires sur les nombres à virgule (+, −, * et /) sont disponibles avec leurs priorités habituelles :

```
printf("%f\n", 1. + 2.3 * 4.5 / 6.78 - 9.);
// affiche : -6.473451
```

En cas de dépassement de capacité (la valeur absolue du résultat est trop grande pour le type **double** ; voir la section 3.2.2), le comportement n'est pas défini par la norme C. En général, le programme s'arrête ou le calcul renvoie une valeur spéciale correspondant à $+\infty$ (comme dans le premier affichage de l'exemple suivant) ou à $-\infty$ (comme dans le deuxième affichage) :

```
double a = 1e200;
double b = 1e200;
double c = a * b;
double d = -a * b;
printf("%f\n",c); // affiche : inf
printf("%f\n",d); // affiche : -inf
```

En cas de sous-dépassement de capacité (la valeur absolue du résultat est trop petite), le comportement n'est pas défini par la norme C. En général, le programme s'arrête ou le calcul renvoie un résultat nul :

```
double a = 1e-200;
double b = 1e-200;
double c = a * b;
printf("%e\n",c); // affiche : 0.000000e+00
```

En cas de division par zéro, le comportement n'est pas défini par la norme C ; la division d'un nombre x par 0 peut provoquer une erreur à l'exécution ou renvoyer une valeur spéciale correspondant à $+\infty$ (si x > 0), à $-\infty$ (si x < 0) ou à un résultat non défini (si x vaut 0). L'exemple suivant illustre ces trois cas :

(2). Les valeurs des deux variables utilisées dans cet exemple n'ont bien entendu pas été choisies au hasard ; on aura reconnu la vitesse de la lumière dans le vide ($c \approx 3 \times 10^8$ m.s^{-1}) et la charge électrique d'un proton ($e \approx 1{,}602 \times 10^{-19}$ C).

```
double a = 22.0;
double b = 0.0;
double c = a / b;
double d = -a / b;
double e = b / b;
printf("%f\n",c); // affiche : inf
printf("%f\n",d); // affiche : -inf
printf("%f\n",e); // affiche : nan
```

1.2.4 Les booléens

Le type `bool`, utilisable à condition d'inclure le fichier d'en-tête `stdbool.h`, représente un booléen. Une variable de ce type ne peut prendre que les valeurs `true` (« vrai ») ou `false` (« faux »).

Pour construire des expressions booléennes, on dispose des opérateurs de comparaison == (égal), != (différent), < (strictement inférieur), > (strictement supérieur), <= (inférieur ou égal) et >= (supérieur ou égal). Ces opérateurs peuvent porter sur des valeurs numériques (entiers et nombres à virgule mais aussi caractères, ces derniers étant assimilés à des entiers et ordonnés selon leur code ASCII) ou sur des pointeurs (voir la section 1.4) de même type. Par exemple, l'expression `2.718 < 3` est vraie et l'expression `'A'>= 'H'` est fausse.

Remarque. Les opérateurs == et = ne doivent pas être confondus ! C'est une erreur fréquente mais pas toujours facile à détecter car un test de la forme **`if`** `(a = b)` est syntaxiquement correct [(3)]. En effet, l'affectation `a = b` est une instruction qui donne à `a` la valeur de `b` mais c'est aussi une expression dont la valeur est celle de `a` ; cette valeur est assimilée à `false` si `a` vaut 0 et à `true` sinon.

```
int x = 1;
int y = 2;
if (x == 1) { // comparaison correcte
    printf("x vaut 1\n"); // affiche : x vaut 1
}
if (y = 1) {
/* y reçoit la valeur 1
   l'affectation est évaluée à 1, assimilé à true
   le bloc d'instructions est donc exécuté */
    printf("y vaut 1\n"); // affiche : y vaut 1
}
printf("x vaut %d et y vaut %d\n", x, y);
// affiche : x vaut 1 et y vaut 1
```

Remarque. En raison des erreurs d'arrondi, il est fortement déconseillé de tester l'égalité de deux valeurs de type **`double`** car le résultat risque de ne pas être celui attendu, comme dans l'exemple suivant (voir la section 3.2.2 pour plus d'explications) :

(3). Il peut toutefois provoquer un avertissement voire une erreur à la compilation, en fonction des options du compilateur.

```
if (1e16 + 1. - 1e16 == 1.) {
    printf("égal\n"); // n'affiche rien
}
else {
    printf("différent\n"); // affiche : différent
}
printf("%e\n",  (1e16 + 1. - 1e16));
// affiche : 0.000000e+00
```

Il vaut mieux vérifier si la différence entre les deux valeurs à comparer ne dépasse pas un certain seuil.

On dispose également des opérateurs logiques classiques `&&` (« et »), `||` (« ou ») et `!` (négation). Par exemple, si `x` est une variable entière, le code suivant teste si l'année `x` est bissextile (on rappelle qu'une année est bissextile si et seulement si son numéro est multiple de 4 mais pas de 100 ou bien multiple de 400 [4]) :

```
if ((x % 4 == 0 && x % 100 !=0) || x % 400 == 0)
    printf ("L'année %d est bissextile\n", x);
else
    printf ("L'année %d n'est pas bissextile\n", x);
```

Quand un programme C doit évaluer une expression booléenne de la forme `expr1 && expr2`, il commence par évaluer l'expression `expr1` ; si elle est fausse, il n'évalue pas `expr2` puisque l'expression globale sera forcément fausse. On parle d'*évaluation paresseuse*. De même, dans le cas d'une expression de la forme `expr1 || expr2` où `expr1` est vraie, on n'évalue pas `expr2` puisqu'on sait que l'expression sera vraie. Cela permet d'écrire des conditions comme celle-ci, où on suppose que `tab` est un tableau dont les indices de cases sont compris entre 0 et `n-1` (voir la section 1.5.1) :

```
if (i >= 0 && i < n && tab[i] >= 1)
```

Si la variable `i` est en dehors de l'intervalle autorisé, on n'évalue pas la condition `tab[i] >= 1` : on évite ainsi une éventuelle erreur d'exécution due à un accès mémoire interdit.

1.2.5 Constantes

Pour définir une constante, c'est-à-dire une variable qu'on ne pourra plus modifier après l'initialisation, on utilise le mot-clé **`const`** :

```
const double pi = 3.14159;
pi = 22.0 / 7.0; // erreur à la compilation
```

On doit initialiser une constante dès sa déclaration : on ne peut pas la déclarer et l'initialiser ensuite.

```
const double pi;
pi = 3.14159; // erreur à la compilation
```

(4). Pour plus d'information sur le calendrier grégorien et les années bissextiles, voir par exemple `https://promenade.imcce.fr/fr/pages2/277.html#Gregorien`.

1.3 Structures de contrôle

1.3.1 Test conditionnel

Le test conditionnel « si - alors »

si *condition* **alors**
└ instructions

s'écrit en C

```
if (condition) {
    // instructions;
}
```

Par exemple, le programme suivant affichera « i est pair » :

```
int i = 2;
if (i % 2 == 0) {
    printf("i est pair\n");
}
```

Le test conditionnel « si - alors - sinon »

si *condition* **alors**
| instructions_1
sinon
└ instructions_2

s'écrit en C

```
if (condition) {
    // instructions_1;
}
else {
    // instructions_2;
}
```

Par exemple, le programme suivant affichera « i est impair » :

```
int i = 1;
if (i % 2 == 0) {
    printf("i est pair\n");
}
else {
    printf("i est impair\n");
}
```

Enfin, un test conditionnel à plusieurs alternatives de la forme

si *condition_1* **alors**
| instructions_1
sinon si *condition_2* **alors**
| instructions_2
sinon
└ instructions_3

s'écrit

```
if (condition_1) {
    // instructions_1;
}
else if (condition_2) {
    // instructions_2;
}
else {
    // instructions_3;
}
```

Cette syntaxe se généralise à un nombre arbitraire de blocs **`else if`**. Par exemple, le programme suivant affichera « n est strictement négatif » :

```
int n = -40;
if (n > 0) {
    printf("n est strictement positif\n");
}
else if (n < 0) {
    printf("n est strictement négatif\n");
}
else {
    printf("n est nul\n");
}
```

Dans ces tests, on peut omettre les accolades s'il n'y a qu'une seule instruction dans le bloc qu'elles délimitent. Par exemple, le code précédent peut aussi s'écrire

```
int i = 1;
if (i % 2 == 0)
    printf("i est pair\n");
else
    printf("i est impair\n");
```

Toutefois, il est recommandé d'utiliser des accolades même dans ce cas. En effet, si on doit modifier le code par la suite pour que d'autres instructions s'exécutent dans l'un des deux cas, on risque d'oublier d'ajouter les accolades, ce qui conduira à un comportement incorrect du programme. En effet, un code tel que

```
int i = 42;
if (i % 2 == 0)
    printf("i est pair\n");
else
    printf("i est impair\n");
    printf("toutes ses puissances sont impaires\n");
```

est interprété par le compilateur comme

```
int i = 42;
if (i % 2 == 0)
    printf("i est pair\n");
else
```

```
    printf("i est impair\n");
printf("toutes ses puissances sont impaires\n");
```

et affiche

```
i est pair
toutes ses puissances sont impaires
```

Dans un tel cas, le compilateur peut cependant, en fonction des options qui lui sont passées, avertir le programmeur que l'indentation est trompeuse.

1.3.2 Boucle « tant que »

La boucle « tant que »

tant que *condition* **faire**
└ instructions

s'écrit en C

```
while (condition) {
    // instructions;
}
```

La condition est testée au début de chaque étape ; si elle est fausse dès le départ, les instructions contenues dans le corps de la boucle (entre les accolades) ne sont jamais exécutées.

Par exemple, le code suivant affiche les carrés parfaits strictement inférieurs à 100, à raison d'un nombre par ligne :

```
int i = 0;

while (i * i < 100) {
    printf ("%d\n", i * i);
    i = i + 1;
}
```

1.3.3 Boucle « pour »

La syntaxe générale d'une boucle « pour » en C est

```
for (initialisation ; condition ; incrementation) {
    // instructions;
}
```

Au début de l'exécution de la boucle, le programme initialise la ou les variable(s) de boucle en exécutant l'instruction `initialisation`. Ensuite, tant que l'expression `condition` est vraie, il exécute les instructions contenues dans le corps de la boucle puis l'instruction `incrementation`, qui modifie la ou les variable(s)

de boucle. Si l'expression `condition` est fausse dès le départ, le corps de la boucle et l'instruction `incrementation` ne sont jamais exécutés.

Par exemple, le code suivant affiche les valeurs de i^2 pour i allant de 0 à 9 puis il affiche « i vaut maintenant 10 ». En effet, lors de la dernière itération (quand i vaut 9), la variable est incrémentée et vaut donc 10 quand on sort de la boucle.

```
int i;
for (i = 0 ; i < 10 ; i = i + 1) {
    printf("%d\n", i * i);
}
printf("i vaut maintenant %d\n", i);
```

On peut également déclarer les variables de boucle en même temps qu'on les initialise mais dans ce cas, elles ne sont définies que pendant l'exécution de la boucle :

```
for (int i = 0 ; i < 10 ; i = i + 1) {
    printf("%d\n", i * i); // affiche la valeur de i²
}
// la variable i n'est plus définie ici
```

La plupart du temps, il n'y a qu'une variable de boucle mais il peut y en avoir plusieurs. Dans ce cas, les différentes instructions d'initialisation et d'incrémentation sont séparées par des virgules, comme dans l'exemple suivant :

```
for (int i = 1, j = 6 ; i <= 6 ; i = i + 1, j = j - 1) {
    printf("%d et %d\n", i, j);
}
printf("font tous 7\n");
```

qui affiche

```
1 et 6
2 et 5
3 et 4
4 et 3
5 et 2
6 et 1
font tous 7
```

Comme pour les tests conditionnels, il est possible (mais déconseillé) d'omettre les accolades si le corps d'une boucle **`for`** ou **`while`** ne contient qu'une seule instruction.

1.3.4 Interrompre une boucle

L'instruction **`break`** permet d'interrompre une boucle **`for`** ou **`while`**. Par exemple, le code suivant affiche les chiffres de 1 à 11, sort de la boucle (car $11^2 > 100$) et exécute la ligne 7 qui affiche « On a compté jusqu'à 11 ».

```
1 int i;
2 for (i = 1 ; i <= 20 ; i = i + 1) {
3     printf("%d\n", i);
4     if (i * i > 100)
5         break;
6 }
7 printf("On a compté jusqu'à %d\n", i);
```

1.4 Pointeurs

Un *pointeur* est une variable qui contient l'adresse d'un emplacement en mémoire, en général l'adresse d'une autre variable. On dira qu'un pointeur `p` *pointe* sur une variable `v` si la valeur de `p` est l'adresse de `v`.

Quand on déclare un pointeur, il faut indiquer le type de la donnée sur laquelle il pointe, afin de pouvoir récupérer correctement la valeur correspondante. Le type d'un pointeur `p` qui pointe sur une donnée de type `t` est `t *`. Par exemple, l'instruction `int *p;` [(5)] déclare un pointeur sur un entier, sans l'initialiser.

Pour affecter à un pointeur `p` de type `t *` l'adresse d'une variable existante `v` de type `t`, on écrira `p = &v;` :

```
int *p;
int x = 42;
p = &x;
```

La valeur `NULL`, définie dans le fichier d'en-tête `stddef.h`, représente un pointeur qui ne pointe sur rien (il contient l'adresse 0, à laquelle on ne peut rien stocker). On peut affecter cette valeur à un pointeur de n'importe quel type :

```
int *p1 = NULL;
char *p2 = NULL;
```

Pour éviter des erreurs dues à des accès mémoire illégaux ou incorrects, il est vivement recommandé d'initialiser un pointeur dès sa déclaration :

- soit en lui donnant l'adresse d'une variable existante ;
- soit en le faisant pointer vers un emplacement mémoire alloué dynamiquement (voir la section 5.1.4) ;
- soit en lui donnant la valeur `NULL`.

Pour *déréférencer* un pointeur `p`, c'est-à-dire pour accéder (en lecture ou en écriture) à la valeur de la case mémoire vers laquelle pointe `p`, on utilise la notation `*p` :

```
int x = 42;
int *p = &x; // p pointe vers x donc *p vaut 42
printf("%d\n", *p); // affiche 42
```

(5). Pour déclarer un pointeur `p` de type `t *`, on écrit généralement `t *p;` plutôt que `t * p;` ou `t* p;` (ces trois écritures sont cependant correctes). On exprime ainsi que `*p` (la valeur de la case vers laquelle pointe `p` ; voir le paragraphe sur le déréférencement) est de type `t`.

```
*p = *p + 1;
printf("%d\n", *p); // affiche 43
```

Si un pointeur n'est pas initialisé ou vaut `NULL`, on ne pourra pas le déréférencer puisqu'il ne pointe sur aucune case mémoire : il faut donc s'assurer qu'un pointeur est initialisé à une valeur non nulle avant de le déréférencer.

On peut utiliser le mot-clé **`const`** pour définir un pointeur constant :

— si **`const`** est placé au début de la déclaration, on pourra modifier la valeur du pointeur mais pas la valeur pointée :

```
int x = 42;
int y = 13;
const int *p = &x;
p = &y; // autorisé
*p = 26; // erreur à la compilation
```

— si **`const`** est placé entre le type et le nom de la variable, on pourra modifier la valeur pointée mais pas celle du pointeur :

```
int x = 42;
int * const p = &x;
*p = 26; // autorisé
p = NULL; // erreur à la compilation
```

— enfin, si **`const`** est placé à la fois avant et après le type, on ne pourra modifier ni la valeur pointée ni celle du pointeur :

```
int x = 42;
const int * const p = &x;
*p = 26; // erreur à la compilation
p = NULL; // erreur à la compilation
```

1.5 Types structurés

1.5.1 Tableaux statiques

Un *tableau* est une structure de données formée d'un nombre fini d'éléments d'un même type. Ce nombre, appelé *taille* ou *longueur* du tableau, est défini une fois pour toutes lors de la déclaration de celui-ci, qui se fait par la syntaxe

```
t nom[taille];
```

où `t` est le type des éléments, `nom` est le nom du tableau et `taille` est sa taille. Celle-ci est une expression entière constante positive (par exemple 2 ou 3+4). Pour déclarer un tableau de taille non connue à la compilation (par exemple de taille `n` où `n` est une variable entière, dont la valeur peut n'être connue qu'à l'exécution), on utilisera l'allocation dynamique de mémoire (voir la section 5.1.4) [6].

(6). Les normes C99 et C11 permettent d'utiliser une expression entière quelconque pour la taille d'un tableau. On peut donc par exemple écrire **`int`** `tab[n];` où `n` est une variable entière. On ne considèrera pas cette possibilité dans ce livre.

Par exemple, l'instruction

```
int tab[10];
```

déclare un tableau de dix cases contenant des entiers.

Remarque. Un tableau d'éléments de type `t` est de type `t *` car l'identificateur d'un tableau désigne en fait un pointeur sur son premier élément. Le code ci-dessous affiche donc deux fois la même valeur :

```
int tab[4];

printf("%p\n", tab);
printf("%p\n", &tab[0]);
```

Dans un tableau, chaque case est repérée par un indice entier : l'indice de la première case est 0, celui de la deuxième case est 1, etc. Pour accéder (en lecture ou en écriture) à la case d'indice i d'un tableau nommé `tab`, on utilise la syntaxe `tab[i]`, où i doit donc être compris entre 0 et la longueur du tableau moins un :

```
int tab[4];
tab[0] = 1;
tab[1] = 6;
tab[2] = 3;
tab[3] = 5;
printf("%d\n", tab[1]); // affiche : 6
```

Le langage ne vérifie pas (ni à la compilation ni à l'exécution) que l'indice est bien compris entre ces bornes : c'est au programmeur de le vérifier, à l'aide d'un test conditionnel si nécessaire. Si on essaie d'accéder à une case d'indice invalide, on peut aboutir à une erreur d'exécution ou, dans le cas le plus grave, modifier d'autres variables n'ayant rien à voir avec le tableau.

On peut initialiser un tableau à l'aide d'une liste non vide de valeurs séparées par des virgules et placées entre accolades (*initialisateur*), comme dans l'exemple suivant. Le type de ces valeurs doit correspondre à celui indiqué pour les éléments du tableau (**int** dans l'exemple).

```
int fibonacci[] = {1, 1, 2, 3, 5, 8};
```

Ici, la taille du tableau est déterminée automatiquement par le compilateur : c'est le nombre de valeurs contenues dans la liste. On peut cependant indiquer cette taille explicitement, comme ceci :

```
int fibonacci[6] = {1, 1, 2, 3, 5, 8};
```

On peut aussi indiquer une taille plus grande que le nombre de valeurs entre les accolades. Dans ce cas, les cases suivantes du tableaux auront la valeur 0 (pour un tableau de nombres) ou la valeur `NULL` (pour un tableau de pointeurs) :

```
int tab[4] = {1, 2}; // le tableau contient 1, 2, 0, 0
```

Si, en revanche, la taille déclarée pour le tableau est strictement inférieure au nombre de valeurs entre les accolades, cela provoque un avertissement lors de la compilation et les valeurs excédentaires sont ignorées :

```
int tab[4] = {5, 4, 3, 2, 1};
/* le tableau contient 5, 4, 3, 2
   le compilateur indique qu'il y a trop de valeurs dans
   l'initialisateur */
```

Tableaux statiques multidimensionnels

On peut mettre dans un tableau des éléments de tout type. En particulier, on peut définir un tableau dont les éléments sont eux-mêmes des tableaux, ce qu'on peut voir comme une matrice (un tableau à deux dimensions). Par exemple, l'instruction `int tab[2][3];` déclare un tableau de taille 2 dont les éléments sont des tableaux d'entiers de taille 3, c'est-à-dire une matrice d'entiers à deux lignes et trois colonnes [7].

Dans un tableau à deux dimensions `tab`, la ligne i correspond à l'élément `tab[i]`, qui est lui-même un tableau. La valeur qui est dans la ligne i et dans la colonne j est en fait l'élément d'indice i du tableau `tab[i]` : on y accède donc par la notation `tab[i][j]`. La figure 1.1 illustre cette notation sur un tableau `t` de taille n dont les éléments sont des tableaux de taille k, ce qui correspond à une matrice comprenant n lignes et k colonnes.

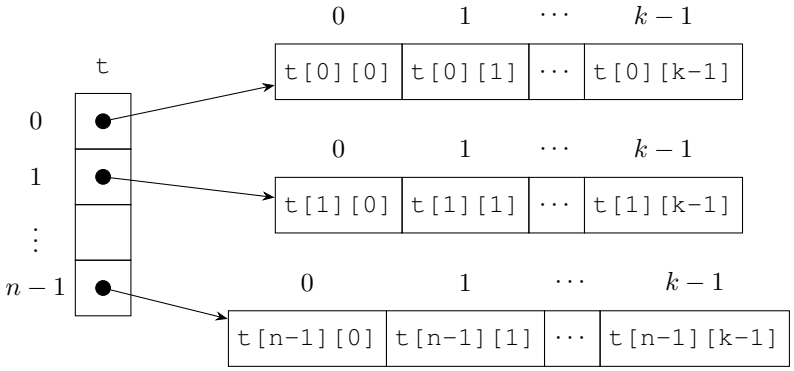


FIGURE 1.1 – Accès aux cases d'un tableau à deux dimensions.

Par exemple, on peut déclarer et initialiser un tableau représentant la matrice

$$\big((5i+j+1)^2\big)_{\substack{0\leqslant i\leqslant 2\\ 0\leqslant j\leqslant 4}} = \begin{pmatrix} 1 & 4 & 9 & 16 & 25 \\ 36 & 49 & 64 & 81 & 100 \\ 121 & 144 & 169 & 196 & 225 \end{pmatrix}$$

à l'aide des instructions suivantes :

```
int m[3][5]; // m est une matrice à 3 lignes et 5 colonnes
```

(7). Par convention, on considère, que ce soit en C ou dans les autres langages de programmation, que la première dimension d'un tableau bidimensionnel (2 dans l'exemple) est le nombre de lignes et que la seconde (3 dans l'exemple) est le nombre de colonnes.

```
for (int i = 0 ; i < 3 ; i = i + 1) {
    for (int j = 0 ; j < 5 ; j = j + 1) {
        m[i][j] = (5 * i + j + 1) * (5 * i + j + 1);
    }
}
```

Comme on l'a déjà fait remarquer, les cases d'un tableau occupent des emplacements consécutifs en mémoire. Cela reste vrai pour les tableaux bidimensionnels. Le tableau `m` de l'exemple précédent sera donc stocké en mémoire comme ceci :

1	4	9	16	25	36	49	64	81	100	121	144	169	196	225

De façon similaire, on peut définir des tableaux de dimension supérieure à deux. Par exemple, l'instruction **double** `tab[3][4][5]` définit un tableau de taille 3 dont les éléments sont des tableaux de taille 4 contenant eux-mêmes des tableaux de nombres à virgule de taille 5. Dans ce cas :

— `tab[i]` (pour $0 \leqslant i \leqslant 2$) est un tableau à deux dimensions (tableau de tableaux de **double**) ;

— `tab[i][j]` (pour $0 \leqslant i \leqslant 2$ et $0 \leqslant j \leqslant 3$) est un tableau de **double** de taille 5 ;

— `tab[i][j][k]` (pour $0 \leqslant i \leqslant 2$, $0 \leqslant j \leqslant 3$ et $0 \leqslant k \leqslant 4$) est une valeur de type **double**.

1.5.2 Chaînes de caractères

En C, il n'existe pas de type spécifique représentant les chaînes de caractères. Une chaîne est simplement un tableau de caractères, dont la dernière case contient le caractère nul (noté `'\0'` et que nous avons vu page 11), qui marque la fin de la chaîne. Par exemple, la chaîne de caractères `"Bonjour"` est équivalente au tableau

`'B'`	`'o'`	`'n'`	`'j'`	`'o'`	`'u'`	`'r'`	`'\0'`

Une constante de type chaîne de caractères est encadrée par des guillemets (par exemple `"Bonjour"`). À l'intérieur, on peut utiliser n'importe quel caractère autre que le caractère nul, en l'écrivant le cas échéant sous forme d'une séquence d'échappement :

```
printf("du texte\navec un saut de ligne"); /* affiche :
du texte
avec un saut de ligne
*/
```

Attention ! Si on veut utiliser le caractère `"` à l'intérieur de la chaîne, il faut l'écrire `\"` ; ainsi, le compilateur comprendra que ce caractère ne délimite pas la chaîne :

```
printf("Je dis "Bonjour""); // incorrect
printf("Je dis \"Bonjour\""); /* correct ;
    affiche : Je dis "Bonjour" */
```

On peut déclarer une chaîne de caractères sous forme d'un tableau de caractères en précisant sa taille et l'initialiser dans la même instruction : par exemple, pour placer la chaîne `"Bonjour"` dans une variable nommée `ma_chaine`, on peut écrire

```
char ma_chaine[8] = "Bonjour";
```

ce qui est en fait équivalent à

```
char ma_chaine[8] = {'B','o','n','j','o','u','r','\0'};
```

La longueur indiquée lors de la déclaration doit être suffisante pour stocker tous les caractères de la chaîne et le caractère nul. Si elle est plus grande, cela ne pose pas de problème :

```
char ma_chaine[20] = "Bonjour";
printf("%s\n", ma_chaine); // affiche : Bonjour
```

En revanche, si elle est plus petite, le comportement dépend de l'implémentation. Au pire, il y aura une erreur de compilation. Au mieux, tous les caractères ne seront pas stockés dans la chaîne :

```
char ma_chaine[3] = "Bonjour";
printf("%s\n", ma_chaine); // affiche : Bon
```

En particulier, il manquera le caractère nul, ce qui peut poser des problèmes, notamment si on souhaite utiliser les fonctions de la bibliothèque standard sur cette chaîne.

On peut aussi déclarer une chaîne sans indiquer la taille du tableau si on l'initialise dans la même instruction :

```
char chaine[] = "Bonjour";
```

Dans l'exemple ci-dessus, la chaîne aura la taille nécessaire pour stocker 7 caractères plus le caractère nul.

On peut également déclarer une chaîne puis l'initialiser plus tard. Dans ce cas, il faut préciser sa taille à la déclaration et initialiser ensuite la chaîne caractère par caractère ou à l'aide d'une fonction. Le code ci-dessous provoque donc une erreur à la compilation car on n'a pas indiqué la taille de la chaîne :

```
char chaine[];
```

De même, le code ci-dessous ne compile pas car `chaine` est considéré comme un pointeur constant :

```
char chaine[10];
chaine = "Bonjour";
```

En revanche, le code suivant fonctionne :

```
char chaine[10];
chaine[0] = 'B';
chaine[1] = 'o';
chaine[2] = 'n';
chaine[3] = 'j';
chaine[4] = 'o';
chaine[5] = 'u';
chaine[6] = 'r';
chaine[7] = '\0';
printf("%s\n", chaine); // affiche : Bonjour
```

On peut aussi déclarer une chaîne comme un pointeur de type **char** * plutôt que comme un tableau. Dans ce cas, on peut l'initialiser dans la même instruction ou plus tard :

```
char *chaine = "Bonjour";
char *chaine2;
chaine2 = "Salut" ;
```

On accède à un caractère d'une chaîne (en lecture ou en écriture) comme à un élément de tableau (rappelons que les indices commencent à 0) :

```
char chaine[] = "Bonjour";
printf("%c\n", chaine[2]); // affiche : n
chaine[3] = 'J';
printf("%s\n", chaine); // affiche : BonJour
```

Attention, si on a déclaré la chaîne comme un pointeur, on ne peut généralement pas la modifier de cette façon :

```
char *ma_chaine = "Bonjour";
ma_chaine[1] = 'a'; // erreur à l'exécution
```

Dans ce cas, il vaut mieux ajouter le mot-clé **const** dans la déclaration, ce qui permettra de détecter le problème dès la compilation :

```
const char *ma_chaine = "Bonjour";
ma_chaine[1] = 'a'; // erreur à la compilation
```

Fonctions sur les chaînes de caractères

Le fichier d'en-tête `string.h` contient différentes fonctions permettant de manipuler les chaînes de caractères. On va à présent étudier trois d'entre elles.

La fonction `strlen(s)` renvoie la longueur de la chaîne de caractères `s`. Celle-ci est calculée en commençant au début de la chaîne et en s'arrêtant avant le caractère nul qui marque la fin de celle-ci. Le type de retour de cette fonction est `size_t` : il s'agit d'un type entier non signé, défini dans `stddef.h`, qui sert à représenter un nombre d'octets. On utilisera le format `%zu` pour afficher une valeur de ce type avec `printf`.

```
char chaine[10] = "Bonjour";
printf("Longueur de la chaîne : %zu\n", strlen(chaine));
// affiche : 7
```

Le programme suivant affiche la longueur d'un mot entré par l'utilisateur à l'aide de la fonction `strlen` :

```
char mot[50];
printf("Entrez un mot\n");
scanf("%s", mot);
printf("C'est un mot de longueur %zu\n", strlen(mot));
```

Remarquons qu'on a écrit `scanf("%s", mot);` et non `scanf("%s", &mot);` car `mot` est déjà une adresse (c'est l'adresse du premier caractère de la chaîne, d'après la remarque de la page 21, vu qu'une chaîne est un tableau de caractères).

La fonction `strcpy(s, t)` copie la chaîne `t` (y compris le caractère nul de fin de chaîne) dans la chaîne `s` et renvoie l'adresse de cette dernière. La chaîne `s` doit être assez grande pour contenir tous les caractères de `t`, comme dans l'exemple suivant :

```
char chaine1[30];
char chaine2[] = "Bonjour";
strcpy(chaine1, chaine2);
printf("%s\n", chaine1); // affiche : Bonjour
```

La fonction `strcat(s, t)` concatène les chaînes `s` et `t` : elle copie la chaîne `t` (y compris le caractère nul de fin de chaîne) à la suite de la chaîne `s` et renvoie l'adresse de cette dernière. La chaîne `s` doit être assez grande pour contenir tous les caractères de `t`, comme dans l'exemple suivant :

```
char chaine1[30] = "Bonjour ";
char chaine2[] = "tout le monde";
strcat(chaine1, chaine2);
printf("%s\n", chaine1);
// affiche : Bonjour tout le monde
```

La bibliothèque standard (fichier `stdlib.h`) contient également des fonctions permettant de convertir une chaîne de caractères en nombre :

- `atoi(s)` (« ASCII to integer ») convertit `s` en un entier de type **`int`** ;
- `atol(s)` (« ASCII to long ») convertit `s` en un entier de type **`long`** (type dont la taille dépend de l'architecture mais est toujours plus grande que celle du type **`int`** ; le code de format correspondant pour `printf` est `%ld`) ;
- `atof(s)` (« ASCII to float ») convertit `s` en un nombre à virgule de type **`double`**.

Dans les trois cas, pour la conversion, on commence à lire au début de la chaîne (en ignorant les éventuels espaces initiaux) et on s'arrête dès qu'on rencontre un caractère qui n'est pas un chiffre. Si la chaîne commence par un caractère non-numérique autre qu'un signe − ou +, les fonctions renvoient 0 (voir l'exemple suivant).

```
int a = atoi("12");
long b = atol("-34");
double c = atof("  567.8");
int d = atoi("90a");
long e = atol("truc");
printf("%d %ld %f %d %ld\n",a, b, c, d, e);
// affiche : 12 -34 567.800000 90 0
```

En cas de sous-dépassement de capacité, la fonction `atof` renvoie 0 :

```
double f = atof("1e-400");
printf("%e\n", f); // affiche : 0.000000e+00
```

En cas de dépassement de capacité, les trois fonctions de conversion renvoient une valeur erronée : il n'est donc pas possible de détecter le dépassement.

1.5.3 Structures

Comme un tableau, une *structure* regroupe plusieurs valeurs mais celles-ci ne sont pas forcément toutes du même type. Par ailleurs, elles ne sont pas repérées par leur indice numérique mais par leur nom.

Avant de créer une structure, on doit d'abord définir un type spécial, appelé *type structure*, permettant de la représenter. Pour définir un type structure contenant n éléments (également appelés *champs* ou *membres*), on utilise la syntaxe

```
struct nom_structure {
    type_1 champ_1;
    type_2 champ_2;
    ...
    type_n champ_n;
};
```

où `nom_structure` est le nom de la structure et pour tout i entre 1 et n, le i^{e} champ est de type `type_i` et a pour nom `champ_i`.

Le type ainsi défini s'appelle **`struct`** `nom_structure`; il peut être utilisé comme type d'une variable ou d'un argument de fonction ainsi que comme type de retour d'une fonction. Pour déclarer une variable de ce type, il faudra écrire

```
struct nom_structure var;
```

ce qui est un peu fastidieux. Après avoir défini un type structure, on définit donc généralement un synonyme pour celui-ci à l'aide de l'instruction **`typedef`** [8] :

```
typedef struct nom_structure ma_structure;
```

Dans cet exemple, on a défini un type `ma_structure` qui est parfaitement équivalent à **`struct`** `nom_structure`. On peut donc maintenant déclarer une variable de ce type en écrivant

(8). On peut définir de cette façon un synonyme pour n'importe quel type, pas seulement pour un type structure. Ainsi, l'instruction **`typedef double`** `nombre_a_virgule;` définit un type `nombre_a_virgule` équivalent au type **`double`**.

```
ma_structure var;
```

Dans l'exemple ci-dessous, on définit un type structure pour représenter une personne puis on déclare une variable de ce type :

```
// définition du type struct p
struct p {
    char nom[30];
    char prenom[30];
    int age;
};

// définition du type personne, synonyme de struct p
typedef struct p personne;

// déclaration d'une variable de type personne
personne untel;
```

On peut également définir un type structure dont l'un des champs est lui-même une structure :

```
struct date {
    int jour;
    int mois;
    int annee;
};

struct p {
    char nom[30];
    char prenom[30];
    struct date date_naissance;
};
```

Toutefois, une structure ne peut pas avoir un champ du même type que la structure elle-même. On ne peut pas, par exemple, inclure les parents d'une personne dans le type correspondant comme ceci (cela provoque une erreur à la compilation) :

```
struct p {
    char nom[30];
    char prenom[30];
    int age;
    struct p pere;
    struct p mere;
};
```

En revanche, on peut inclure des pointeurs vers des structures de type **struct** p qui représentent les parents, comme ceci :

```
struct p {
    char nom[30];
    char prenom[30];
```

```
    int age;
    struct p *pere;
    struct p *mere;
};
```

On obtient alors une structure récursive. C'est de cette façon qu'on pourra définir en C des structures de données comme les listes chaînées ou les arbres (voir le chapitre 5).

Si `s` est une structure comportant un champ `c`, on accède à ce champ (en lecture ou en écriture) par la syntaxe `c.s` :

```
struct point {
    double latitude;
    double longitude;
};
typedef struct point lieu;

lieu x;
x.latitude = 40.0;
x.longitude = 0.0;

printf("x : latitude %f, longitude %f\n",
    x.latitude, x.longitude);
```

Si `p` est un pointeur vers une structure comportant un champ `s`, on accède à ce champ par la syntaxe `p->s`, qui est équivalente à `(*p).s` [9] :

```
// on reprend le type lieu de l'exemple précédent
lieu y;
lieu *p = &y;
p->latitude = -22.6;
p->longitude = 167.5;
printf("y : latitude %f, longitude %f\n",
    p->latitude, p->longitude);
```

On peut initialiser une structure dont les champs sont nommés `ch_1`, `ch_2`, ..., `ch_n` à l'aide d'un initialisateur de la forme

```
{.ch_1 = val_1, .ch_2 = val_2, ..., .ch_n = val_n}
```

Pour tout $i \in \{1; 2; \ldots; n\}$, on donne ainsi au champ `ch_i` la valeur `val_i`. Le type de `val_i` doit donc naturellement correspondre à celui de `ch_i`.

```
struct point {
    int x;
    int y;
};
typedef struct point coords;
```

(9). Il ne faut surtout pas écrire `*p.s` ! Cette écriture serait interprétée comme `*(p.s)`, ce qui n'a pas de sens car `p` n'a pas de champ `s` (c'est un pointeur, pas une structure).

```
coords P = {.x = 2, .y = 1};
printf("P est de coordonnées (%d;%d)\n", P.x, P.y);
// affiche : P est de coordonnées (2;1)
```

Si on n'initialise pas certains champs, ils recevront la valeur 0 (pour un champ numérique) ou la valeur `NULL` (pour un champ de type pointeur), comme dans l'exemple suivant qui reprend le type `coords` défini précédemment :

```
coords P0 = {.y = 1};
printf("P0 est de coordonnées (%d;%d)\n", P.x, P.y);
// affiche : P0 est de coordonnées (0;1)
```

Si on fait référence à des champs n'existant pas dans la structure, cela provoquera une erreur à la compilation.

1.6 Fonctions

On définit une fonction par la syntaxe

```
type_retour nom_fonc(type_1 arg_1, ..., type_n arg_n) {
    // instructions;
}
```

où `type_retour` est le type de la valeur renvoyée par la fonction (on utilise le type **`void`** si elle ne renvoie rien), `nom_fonc` est le nom de la fonction et, pour tout i entre 1 et n, `type_i` est le type du i^{e} argument, nommé ici `arg_i`. Si la fonction ne prend aucun argument, on écrira simplement **`void`** entre les parenthèses, comme ceci :

```
type_retour nom_fonc(void) {
    // instructions;
}
```

Le corps de la fonction doit toujours être entouré d'accolades, même s'il ne contient qu'une instruction.

Le nom d'une fonction doit respecter les mêmes règles que le nom d'une variable (voir la section 1.1). De plus, il ne peut pas être identique au nom d'une autre fonction ou d'une variable globale.

On peut utiliser une fonction à partir de sa définition jusqu'à la fin du programme. Le code suivant est donc correct :

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

void affiche_somme(int x, int y) {
    printf("%d\n", x + y);
}

int main (void) {
```

```
    int a = 42;
    int b = 13;
    affiche_somme(a, b); // affiche : 55
    return EXIT_SUCCESS;
}
```

En revanche, le compilateur produit un avertissement si on inverse les deux fonctions :

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

int main(void) {
    int a = 42;
    int b = 13;
    affiche_somme(a, b);
    return EXIT_SUCCESS;
}

void affiche_somme(int x, int y) {
    printf("%d\n", x + y);
}
```

En effet, on utilise alors la fonction `affiche_somme` avant de l'avoir définie. Pour pouvoir écrire les deux fonctions dans cet ordre, il faut ajouter au début du programme le *prototype* de la fonction `affiche_somme`, c'est-à-dire une déclaration récapitulant le nom de la fonction, son type de retour et les types de ses arguments, comme ceci :

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

// prototype de la fonction affiche_somme
void affiche_somme(int x, int y);

int main(void) {
    int a = 42;
    int b = 13;
    affiche_somme(a,b);
    return EXIT_SUCCESS;
}

void affiche_somme(int x, int y) {
    printf("%d\n", x + y);
}
```

D'une façon générale, il est conseillé (mais pas obligatoire) d'écrire les prototypes des fonctions au début du fichier source (ou dans un fichier d'en-tête ; voir la section 1.7). C'est même indispensable dans le cas de fonctions mutuellement récursives (voir la section 4.1.4).

1.6.1 Valeur de retour

Pour indiquer dans le corps d'une fonction qu'on souhaite renvoyer une valeur et mettre fin à l'exécution de la fonction, on utilise l'instruction **return** suivie de la valeur à renvoyer. Ainsi la fonction suivante renvoie le carré de son argument :

```
int carre(int x) {
    return x * x;
}
```

Une même fonction peut contenir plusieurs instructions **return** :

```
int valeur_absolue(int n) {
    if (n > 0) {
        return n;
    }
    else {
        return -n;
    }
}
```

Cela permet notamment de traiter des cas d'erreur ou des valeurs particulières.

Dans tous les cas, le type de l'expression renvoyée par une instruction **return** doit être compatible avec le type de retour indiqué pour la fonction. Une fonction dont le type de retour est **int** peut ainsi contenir l'instruction **return** `0;` ou même **return** `3.14;` (dans ce cas, la valeur sera tronquée et la fonction renverra 3) mais pas une instruction telle que **return** `&x;` car les types pointeurs ne sont pas compatibles avec le type **int**.

Une fonction ne peut renvoyer qu'une valeur. Toutefois, celle-ci peut être de type structure, ce qui permet de simuler une fonction renvoyant plusieurs valeurs. Ainsi, l'exemple suivant présente une fonction qui prend en arguments deux entiers `a` et `b` et renvoie le quotient et le reste de la division euclidienne de `a` par `b` sous forme d'une structure :

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

typedef struct quotient_reste {
    unsigned int quotient;
    unsigned int reste;
    bool valide; // indique si on a divisé par 0
} q_r;

q_r division(unsigned int a, unsigned int b) {
    q_r resultat;
    if (b > 0) {
        resultat.quotient = a / b;
        resultat.reste = a % b;
        resultat.valide = true;
    }
```

```
    else {
        resultat.quotient = 0;
        resultat.reste = 0;
        resultat.valide = false;
    }
    return resultat;
}

int main(void) {
    unsigned int a = 42;
    unsigned int b = 12;
    q_r div = division(a,b);
    printf("Résultat de la division de %d par %d :\n",
        a, b);
    // affiche : Résultat de la division de 42 par 12 :
    printf("Quotient : %d, reste : %d\n",
        div.quotient, div.reste);
    // affiche : Quotient : 3, reste : 6
    return EXIT_SUCCESS;
}
```

Remarquons qu'on ne peut pas utiliser un tableau à la place d'une structure dans ce cas. Ainsi, le code suivant ne fonctionne pas car la fonction renvoie l'adresse d'une variable locale (en effet, `resultat` est équivalent à `&resultat[0]`) ; or, cette variable n'existe plus après la sortie de la fonction (voir la section 7.1.3) :

```
int *division(unsigned int a, unsigned int b) {
    int resultat[3];
    if (b > 0) {
        resultat[0] = a / b;
        resultat[1] = a % b;
        resultat[2] = true;
    }
    else {
        resultat[0] = 0;
        resultat[1] = 0;
        resultat[2] = false;
    }
    return resultat;
}
```

Si on essaie d'initialiser un pointeur avec le résultat de la fonction, ce pointeur vaudra donc `NULL`.

Si on tient absolument à renvoyer une valeur de type tableau, on peut :

— renvoyer une structure dont l'unique champ est de type tableau ;

— déclarer le tableau en dehors de la fonction, le modifier dans la fonction puis renvoyer son adresse, ce qui donnerait ceci dans l'exemple précédent :

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

int resultat[3];

int *division(unsigned int a, unsigned int b) {
    if (b > 0) {
        resultat[0] = a / b;
        resultat[1] = a % b;
        resultat[2] = true;
    }
    else {
        resultat[0] = 0;
        resultat[1] = 0;
        resultat[2] = false;
    }
    return resultat;
}

int main (void) {
    unsigned int a = 42;
    unsigned int b = 12;
    int *res = division(a,b);
    printf("Résultat de la division de %d par %d :\n",
        a, b);
    printf("Quotient : %d, reste : %d\n",
        res[0], res[1]);
    return EXIT_SUCCESS;
}
```

— utiliser l'allocation dynamique (voir la section 5.1.4) pour déclarer dans la fonction un tableau qui perdure après la sortie de celle-ci, ce qui donnerait ceci dans l'exemple précédent :

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

int *division(unsigned int a, unsigned int b) {
    // allocation dynamique d'un tableau de 3 entiers
    int *resultat = malloc(3 * sizeof(int));
    if (b > 0) {
        resultat[0] = a / b;
        resultat[1] = a % b;
        resultat[2] = true;
    }
    else {
        resultat[0] = 0;
        resultat[1] = 0;
```

```
        resultat[2] = false;
    }
    return resultat;
}

int main(void) {
    unsigned int a = 42;
    unsigned int b = 12;
    int *res = division(a,b);
    printf("Résultat de la division de %d par %d :\n",
        a, b);
    printf("Quotient : %d, reste : %d\n",
        res[0], res[1]);
    return EXIT_SUCCESS;
}
```

1.6.2 Passage des arguments par valeur

Le passage des arguments se fait par valeur, c'est-à-dire que la valeur passée à un argument est copiée dans une variable locale à la fonction. Si un argument est la valeur d'une variable extérieure à la fonction, cette dernière manipulera donc une copie de cette variable et ne modifiera pas la variable d'origine :

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

int doubler(int x) {
    x = x * 2;
    // on a modifié une variable locale à la fonction
    return x;
}

int main(void) {
    int a = 3;
    printf("doubler(a) = %d\n", doubler(a));
    // affiche : doubler(a) = 6
    printf("a = %d\n", a); // affiche : a = 3
    // la valeur de a n'a pas été modifiée
    return EXIT_SUCCESS;
}
```

On peut transmettre de cette façon des arguments de type pointeur, c'est-à-dire des adresses de variables, comme dans l'exemple suivant :

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

void f(int *x) {
```

```
    x = NULL;
    printf("Adresse dans f : %p\n", x);
}

int main (void) {
    int a = 3;
    printf("Adresse de a avant f : %p\n", &a);
    f(&a);
    printf("Adresse de a après f : %p\n", &a);
    return EXIT_SUCCESS;
}
```

Ce code affiche (10)

```
Adresse de a avant f : 0x7ffcc33678f4
Adresse dans f : (nil)
Adresse de a après f : 0x7ffcc33678f4
```

ce qui montre bien qu'il s'agit toujours d'un passage par valeur : la fonction `f` ne modifie pas l'adresse de la variable `a` mais uniquement une copie de cette adresse.

Si une fonction reçoit en argument un pointeur sur une variable, elle pourra alors, à l'aide de l'opérateur de déréférencement, accéder à la valeur de cette variable ou la modifier. Dans le deuxième cas, la modification sera permanente : à la sortie de la fonction, la variable gardera sa nouvelle valeur. Par exemple, la fonction `echanger` du programme suivant échange les valeurs de deux variables entières dont les adresses sont passées en arguments :

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

int echanger(int *p, int *q) {
    int r = *p;
    *p = *q;
    *q = r;
}

int main(void) {
    int a = 2;
    int b = 3;
    printf("a = %d, b = %d\n", a, b);
    // affiche : a = 2, b = 3
    echanger(&a, &b); /* les arguments de la fonction
                         sont les adresses de a et b */
    printf("a = %d, b = %d\n", a, b);
    // affiche : a = 3, b = 2
```

(10). L'adresse de `a` est différente d'une machine à l'autre et même d'une exécution à l'autre sur une même machine. Néanmoins, pour une même exécution, elle est identique avant et après l'appel de `f`.

```
    return EXIT_SUCCESS;
}
```

On n'aurait pas pu arriver au même résultat sans utiliser de pointeurs. En effet, si on avait écrit la fonction `echanger` comme ceci :

```
int echanger(int p, int q) {
    int r = p;
    p = q;
    q = r;
}
```

et qu'on l'avait appelée en écrivant

```
echanger(a,b);
```

elle aurait simplement échangé les valeurs de deux variables locales, ce qui n'aurait pas affecté les variables globales `a` et `b`.

De même le code ci-dessous reprend la fonction `doubler` vue précédemment, mais celle-ci travaille à présent sur un pointeur. La variable `a` est donc modifiée après l'appel de la fonction :

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

int doubler(int *x) {
    *x = *x * 2;
    return *x;
}

int main(void) {
    int a = 3;
    printf("doubler(a) = %d\n", doubler(&a));
    // affiche : doubler(a) = 6
    printf("a = %d\n", a); // affiche : a = 6
    return EXIT_SUCCESS;
}
```

Si on souhaite passer en argument un tableau à une fonction, on peut utiliser trois syntaxes différentes (on les illustre ici sur un tableau `t` d'entiers) :

— `f(int t[5])` (la taille indiquée n'est qu'une information pour les lecteurs du code source ; elle n'est utilisée ni à la compilation ni à l'exécution) ;

— `f(int t[])` ;

— `f(int *t)`.

Dans les trois cas, l'argument que la fonction reçoit est en fait un pointeur, en raison de l'équivalence entre tableau et pointeur mentionnée à la section 1.5.1. Il n'est donc pas possible de récupérer la taille du tableau dans la fonction. Cette limitation n'est pas gênante si la taille du tableau passé en argument à la fonction est fixe et connue lors de l'écriture du programme, comme dans l'exemple ci-dessous, où on calcule le maximum d'un tableau de cinq entiers :

```
int max5(int tab[5]) {
/* on pourrait omettre la taille du tableau dans la
   déclaration et écrire int max5 (int tab[]) */
    int m = tab[0];
    for (int i = 1 ; i < 5 ; i = i + 1) {
        if (tab[i] > m) {
            m = tab[i];
        }
    }
    return m;
}
```

Dans le cas contraire, il faut passer la taille du tableau en argument à la fonction, comme dans l'exemple ci-dessous, où on calcule le maximum d'un tableau de taille quelconque :

```
int maxn(int tab[], int n) { // n est la taille de tab
    int m = tab[0];
    for (int i = 1 ; i < n ; i = i + 1) {
        if (tab[i] > m) {
            m = tab[i];
        }
    }
    return m;
}
```

Voici un exemple d'utilisation de ces deux fonctions :

```
int tab[5] = {4,6,2,9,3};
int tab2[7] = {5,7,4,2,8,7,6};
printf("Maximum de tab avec max5 = %d\n", max5(tab));
// affiche : Maximum de tab avec max5 = 9
printf("Maximum de tab avec maxn = %d\n", maxn(tab, 5));
// affiche : Maximum de tab avec maxn = 9
printf("Maximum de tab2 = %d\n", maxn(tab2, 7));
// affiche : Maximum de tab2 = 8
```

1.7 Compilation séparée

Quand un programme atteint une certaine taille, il devient plus pratique de le découper en plusieurs fichiers. On peut ainsi diviser plus facilement le travail entre les développeurs et éventuellement réutiliser une partie du code dans d'autres projets. De plus, il est possible de compiler séparément ces différents fichiers. On n'a donc pas besoin de recompiler tout le code quand on modifie l'un des fichiers, ce qui simplifie le développement et la maintenance.

Si un programme contient plusieurs fichiers d'implémentation (d'extension `.c`), on peut voir chacun d'entre eux comme un fournisseur de services qui définit un

certain nombre de fonctions, de types, de variables globales... que les autres fichiers peuvent utiliser. On associera à chaque fichier d'implémentation (sauf le fichier principal, qui contient la fonction `main`) un *fichier d'en-tête* qui définira les « services » fournis par ce fichier. Concrètement, un fichier d'en-tête peut notamment contenir des directives, des définitions de constantes ou de types, des déclarations de variables globales ou des prototypes de fonctions. Les fichiers d'en-tête ont l'extension `.h` (11). On a déjà rencontré les fichiers d'en-tête standard, comme `stdio.h`, mais il est possible de créer son propre fichier d'en-tête.

Pour inclure un fichier d'en-tête nommé `fichier.h` dans un fichier d'implémentation, ce qui permet d'utiliser dans ce dernier les types, variables ou fonctions déclarés dans le fichier d'en-tête, on ajoute au fichier `.c` la ligne

```
#include <fichier.h>
```

s'il s'agit d'un fichier d'en-tête standard (que le préprocesseur cherchera dans un répertoire prédéfini) et

```
#include "fichier.h"
```

pour un fichier d'en-tête créé par le programmeur (que le préprocesseur cherchera généralement dans le répertoire où se trouve le fichier `.c` (12)).

Par exemple, définissons une variable globale `r` ainsi qu'une fonction `carre`, qui calcule le carré d'un entier, dans un fichier `carre.c` comme ceci :

```
#include "carre.h"

int r = 42;

int carre(int n) {
    return n * n;
}
```

Une variable globale ne peut être définie que dans un seul fichier. Pour pouvoir l'utiliser dans d'autres fichiers, on doit placer dans le fichier d'en-tête associé au fichier d'implémentation où elle est définie la déclaration de cette variable globale, précédée du mot-clé **extern**. Dans notre exemple, le fichier d'en-tête `carre.h` contiendra donc le code

```
extern int r;

int carre(int n);
```

Remarque. Ici, l'inclusion du fichier d'en-tête `carre.h` dans le fichier `carre.c` permet, lors de la compilation de ce dernier, de vérifier que la définition de la variable et de la fonction est bien compatible avec leur déclaration.

(11). La lettre `h` est l'initiale de *header*, signifiant « en-tête » en anglais.
(12). Si le fichier d'implémentation et le fichier d'en-tête ne sont pas dans le même répertoire, il faut écrire à la suite de **#include** le chemin relatif du fichier d'en-tête par rapport au répertoire où se trouve le fichier d'implémentation (voir page 375), par exemple **#include** `"../includes/fichier.h"`.

Écrivons maintenant un fichier `affiche_carre.c` qui affiche le carré de l'entier `r` à l'aide de la fonction écrite précédemment :

```
#include <stdio.h> // pour pouvoir utiliser printf
#include <stdlib.h> // pour pouvoir utiliser EXIT_SUCCESS
#include "carre.h" /* pour pouvoir utiliser la variable r
                      et la fonction carre */

int main(void) {
    printf("Le carré de %d est %d\n", r, carre(r));
    return EXIT_SUCCESS;
}
```

On remarque les deux syntaxes différentes pour l'inclusion d'un fichier d'en-tête : `stdio.h` est entre chevrons car c'est un en-tête standard tandis que `carre.h` est entre guillemets car il fait partie du code que nous avons écrit.

Pour transformer ces fichiers en programme exécutable, il faut d'abord compiler séparément les fichiers d'implémentation par les commandes `gcc -c carre.c` et `gcc -c affiche_carre.c` puis réaliser l'édition des liens par la commande

```
gcc -o affiche_carre affiche_carre.o carre.o
```

On peut ensuite exécuter le programme par la commande `./affiche_carre`, qui affiche

```
Le carré de 42 est 1764
```

Un fichier d'en-tête peut contenir des directives. En particulier, il peut inclure un autre fichier d'en-tête. Il est ainsi possible qu'un fichier d'implémentation inclue deux fois un même fichier d'en-tête, ce qui peut conduire à des erreurs. Par exemple, soient deux fichiers d'en-tête `a.h` et `b.h` contenant respectivement

```
struct p {
    char nom[30];
    char prenom[30];
    int age;
};

typedef struct p personne;
```

et

```
#include "a.h"

void saluer(personne p);
```

Si on essaie de compiler un fichier `.c` qui inclut les deux fichiers `a.h` et `b.h`, on verra s'afficher un message d'erreur expliquant que le type structure déclaré originellement dans `a.h` a été redéfini dans `b.h`.

La solution consiste à n'inclure le fichier `a.h` que s'il ne l'a pas déjà été. Pour cela, on définit, à l'aide d'une directive **`#define`**, une constante qui sert à repérer

si le fichier a déjà été inclus. Si cette constante n'est pas déjà définie (autrement dit, si le fichier n'a pas déjà été inclus), on la définit et on déclare les types, variables, fonctions... contenus dans le fichier. Pour protéger un fichier d'en-tête contre les inclusions multiples, on le structurera donc de la façon suivante :

```
#ifndef TRUC // si la constante TRUC n'est pas définie
#define TRUC // on définit la constante TRUC
// contenu du fichier proprement dit
#endif // fin du bloc if
```

Une convention usuelle consiste à nommer la constante `FICHIER_H` si le fichier d'en-tête se nomme `fichier.h`. Notre fichier `a.h` deviendra ainsi

```
#ifndef A_H
#define A_H

struct p {
    char nom[30];
    char prenom[30];
    int age;
};

typedef struct p personne;

#endif
```