

Introduction à la programmation impérative

Aymeric Vincent

2011–2012
domaine public

Table des matières

1	Environnement et généralités	1
1.1	Généralités sur la programmation impérative	1
1.2	Généralités sur le langage C	1
1.3	Les étapes dans la compilation d'un programme	2
1.3.1	Prétraitement (<i>preprocessing</i>)	3
1.3.2	Compilation	3
1.3.3	Assemblage	3
1.3.4	Edition de liens (<i>linking</i>)	3
1.4	Environnement d'exécution	3
1.4.1	Vision générale de l'environnement	3
1.4.2	Communication avec le shell	4
1.4.3	Compilation en pratique	5
1.4.4	Options de gcc	6
2	Syntaxe générale du langage C et expressions	7
2.1	Considérations générales	7
2.1.1	Commentaires	7
2.1.2	Code C	7
2.1.3	Directives du préprocesseur	8
2.2	Identificateurs	9
2.3	Syntaxe des expressions	9
2.3.1	Priorité et associativité des opérateurs	10
2.3.2	Opérateurs arithmétiques	10
2.3.3	Opérateurs d'affectation	10
2.3.4	Opérateurs de comparaison	11
2.3.5	Opérateurs booléens	11
2.3.6	Opérateurs sur les bits	12
2.4	Syntaxe des instructions	12
3	Types du langage C	13
3.1	Types scalaires	13
3.1.1	Les entiers	13
3.1.2	Les constantes entières	14
3.1.3	Les flottants	15
3.1.4	Les booléens	15
3.2	Tableaux	15
3.2.1	Définition et initialisation d'un tableau	16
3.2.2	Taille d'un tableau	16
3.2.3	Passage de tableau en paramètre d'une fonction	17

3.2.4	Cas particulier des chaînes de caractères	17
3.2.5	Tableaux à plusieurs dimensions	17
3.3	Structures	18
3.4	Fonctions	19
3.4.1	Déclaration et définition de fonction	19
3.4.2	Retour d'une valeur par une fonction	20
3.4.3	Copie des paramètres et des valeurs de retour	20
3.5	Pointeurs	20
4	Déclarations et définitions	23
4.1	Visibilité des identificateurs	23
4.2	Durée de vie des entités	24
4.3	Programmation modulaire	24
5	Structures de contrôle	27
5.1	Conditions	27
5.2	Boucles	28
5.2.1	La boucle while	28
5.2.2	La boucle do-while	29
5.2.3	La boucle for	29
5.2.4	Sauts	30
6	Bibliothèque C standard	31
6.1	Entrées/sorties	31
6.1.1	Gestion de fichiers	31
6.1.2	Écriture et lecture de données formatées	33
6.1.3	Écriture et lecture de plus bas niveau	34
6.1.4	Position du curseur dans les fichiers	35
6.2	Chaînes de caractères	35
6.3	Fonctions utilitaires	36
6.3.1	Génération de nombres pseudo-aléatoires	36
6.3.2	Conversion d'une chaîne de caractères en entier	36
6.4	Gestion des erreurs	36
6.5	Fonctions mathématiques	37

Ce fascicule vient en support de deux cours dispensés à l'ENSEIRB-MATMECA en première année d'école : PG101 qui est un cours dispensé en filière informatique et PG108 qui est dédié à la filière électronique. Il s'agit d'une introduction à la programmation impérative basée sur le langage C. Nous renvoyons le lecteur curieux à un des nombreux livres qui traitent du langage C, dont certains sont de très bonne qualité. Le document décisif qui définit le langage est la norme ISO qui est elle disponible sur Internet. Nous encourageons vivement chaque étudiant à aller visiter les sites Internet de divers projets *open source* afin de voir vivre du code en langage C.

Ce fascicule est placé dans le domaine public.

Aymeric Vincent

Chapitre 1

Environnement et généralités

1.1 Généralités sur la programmation impérative

Programmer un ordinateur peut se faire de plusieurs façons, qui correspondent à des domaines d'utilisation différents. Une des façons de faire consiste à dire pas par pas à l'ordinateur ce qu'il doit faire. Ainsi à chaque instant, l'ordinateur peut avoir une vision locale du programme à exécuter et cela lui permet une exécution efficace de chaque pas du programme.

Un des effets importants que cela a sur le style d'un programme est qu'il faut très rapidement définir des variables. Ces variables peuvent contenir des valeurs et permettent de stocker l'état du programme en mémoire. Chaque pas d'un programme impératif peut modifier une variable afin plus tard d'utiliser la valeur qui y aura été mise.

On peut citer d'autres styles de programmation comme la programmation par contraintes où on définit un ensemble de relations que l'on peut interroger. Dans ce cas, il n'est pas nécessaire de décrire pas à pas comment résoudre cette requête, c'est l'implémentation du langage qui s'en charge. Certains problèmes se résolvent beaucoup plus facilement par programmation par contraintes.

Le style fonctionnel met lui l'accent sur le fait que chaque élément d'exécution du langage reçoit des paramètres et fournit une valeur. L'utilisation de variables modifiées explicitement devient alors peu fréquente, et là encore certains problèmes se décrivent de manière plus élégante dans ce type de langages.

1.2 Généralités sur le langage C

Le langage C est un langage de programmation impératif, dont les buts sont partagés avec le système UNIX : simplicité et portabilité. Un programme écrit en langage C pur peut être compilé et exécuté de la même manière sur des plate-formes et des systèmes d'exploitation très différents. Il est possible de trouver des compilateurs C pour la plupart des plate-formes existantes.

Ce langage est souvent considéré comme le langage le plus simple au-dessus de l'assembleur qui permette de développer confortablement et de façon portable des applications complètes, en partant du système d'exploitation jusqu'à des traitements de texte ou des jeux vidéo par exemple.

Le langage C a été créé en 1972 par Dennis Ritchie dans les laboratoires Bell, au même endroit et à la même période qui ont vu naître le système d'exploitation UNIX. Il a été créé principalement pour réécrire les premières versions du système UNIX de façon portable ; les versions précédentes étaient écrites en assembleur.

On distingue trois grandes révisions du langage C :

1. Le C “Kernighan & Ritchie” (K&R), tel que décrit en 1978 dans la première édition de l’ouvrage de référence “The C programming language” par Brian Kernighan et Dennis Ritchie.
2. Le C “ANSI” ou “C89” normalisé par l’ANSI (American National Standards Institute), décrit aussi dans la seconde édition de l’ouvrage “The C programming language” des mêmes auteurs.
3. Le C “ISO” ou “C99” normalisé par l’ISO (International Standards Organization) et dont un des derniers brouillons de la norme est librement diffusé sur Internet.

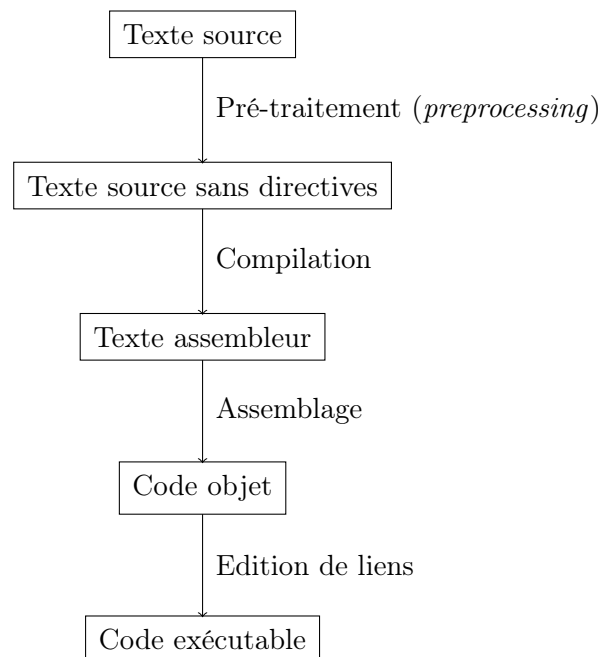
Chaque nouvelle version du langage est quasi totalement compatible ascendante avec la version précédente, et de nouveaux éléments sont introduits dans le langage, souvent empruntés d’extensions jusque là spécifiques à un compilateur ou à d’autres langages comme le C++.

1.3 Les étapes dans la compilation d’un programme

En tant que programmeur en langage C, nous devons savoir *compiler* nos programmes. Il s’agit de l’action pendant laquelle notre code source (du texte respectant la syntaxe du langage C) est transformé en programme exécutable afin de pouvoir ensuite l’exécuter.



Cette action est en fait généralement séparée en plusieurs étapes, et il est utile de connaître cette séparation car elle explique certaines possibilités et limitations du langage C. Les étapes peuvent être représentées comme suit, et nous allons les détailler dans les sous-sections qui suivent :



1.3.1 Prétraitement (*preprocessing*)

Les deux étapes qui constituent la “compilation” proprement dite sont le prétraitement (*preprocessing* en anglais) et la compilation du source C prétraité vers le langage assembleur. L'étape de prétraitement fait partie intégrante de la définition du langage C.

Le prétraitement consiste à transformer le code source d'un programme C en un autre code source C débarrassé des directives de prétraitement, et où les macros du préprocesseur ont été expansées. Les directives de prétraitement permettent d'inclure d'autres fichiers contenant des déclarations (fichiers dits “d'en-tête”), et de définir de nouvelles macros.

Les directives du préprocesseur commencent toutes par un caractère dièse (#) qui doit être le premier caractère autre que d'espace sur une ligne; par convention, les macros du préprocesseur sont écrites en majuscules, et leur utilisation doit être réduite le plus possible.

1.3.2 Compilation

La compilation du code C prétraité en assembleur est la phase la plus compliquée dans toute la chaîne car c'est pendant cette phase que l'on passe d'un langage de haut niveau vers l'assembleur, qui est un langage de très bas niveau.

1.3.3 Assemblage

L'assemblage est une phase anecdotique puisqu'elle effectue une traduction quasiment bijective entre langage assembleur (texte) et langage machine (binaire). Elle est toutefois utile car c'est souvent dans cette phase que les adresses relatives des différents symboles apparaissant dans le code source sont choisies.

1.3.4 Edition de liens (*linking*)

Enfin, l'édition de liens permet de combiner les fichiers “objet” produits par la phase d'assemblage en un exécutable. Elle fait le lien entre des symboles définis dans certains objets (par exemple des bibliothèques) et ces mêmes symboles utilisés dans d'autres objets.

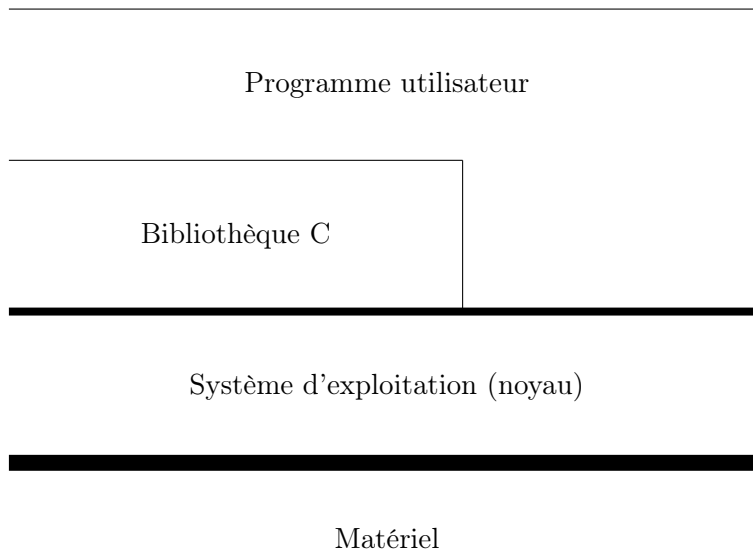
1.4 Environnement d'exécution

1.4.1 Vision générale de l'environnement

Il est important de comprendre dans quel contexte un programme que l'on écrit va s'exécuter. Nous décrivons ici la situation classique d'un système à base de noyau (cas d'UNIX, Windows, MacOS X, ...), où schématiquement le matériel n'est accédé directement que par le noyau du système d'exploitation. Un programme utilisateur qui souhaite écrire des données dans un fichier ou afficher une information à l'écran *doit* effectuer une opération matérielle spécifique appelée “appel système”. Cette séparation doit assurer l'intégrité de l'ensemble du système.

Afin de rendre ces appels système plus faciles à utiliser, le langage C normalise la bibliothèque C standard, qui offre une interface de programmation de plus haut niveau qui propose aussi de nombreuses opérations classiques, par exemple sur les chaînes de caractères.

Cette description peut être schématisée comme suit, les lignes plus épaisses dénotant l'éloignement conceptuel des différentes couches.



Il est fondamental de comprendre que l'accès au matériel par un programme C qui s'exécute sous UNIX *doit* passer par le système d'exploitation. On peut dire qu'il y a deux exceptions à cette règle : le processeur et la mémoire sont manipulés directement par le programme (de manière légèrement limitée pour assurer la stabilité du système complet). Le compilateur C produit donc du code directement interprétable par le processeur, et ce code manipule directement les données qui se trouvent dans la mémoire fournie au programme lors de son exécution.

1.4.2 Communication avec le shell

Un programme est en général lancé par un autre programme qui est fréquemment l'interpréteur de commandes (shell) du système d'exploitation sur lequel il doit être exécuté.

Il existe une communication limitée entre l'interpréteur de commandes et le programme : au lancement du programme, l'interpréteur de commandes peut passer au programme des paramètres qui sont donnés par l'utilisateur dans la ligne de commande invoquant le programme, et lorsqu'il se termine, le programme doit retourner une valeur entière indiquant s'il s'est correctement terminé ou pas.

La fonction `main()` a un statut particulier par rapport aux autres fonctions : c'est elle qui est appelée au démarrage du programme.

Retour d'une valeur par le programme

C'est la valeur de retour de la fonction `main()` qui constituera la valeur retournée au shell. La fonction `main()` doit donc *impérativement* retourner un `int`. Deux valeurs de retour sont possibles : `EXIT_SUCCESS` et `EXIT_FAILURE` qui indiquent respectivement une exécution réussie et une exécution erronée. Ces deux constantes entières sont définies dans le fichier d'en-tête `stdlib.h`.

Passage de paramètres au programme

Lors d'une première lecture, il est préférable de survoler les détails techniques évoqués dans ces quelques paragraphes.

Le prototype de la fonction `main()` admet plusieurs variations dont les deux plus courantes sont :

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

Sous la première forme, le programme ne pourra pas récupérer les éventuels paramètres passés par l'utilisateur sur la ligne de commande. La deuxième forme indique comment les paramètres sont passés lorsque le programme souhaite y accéder.

L'entier `argc` (pour "compteur d'arguments") indique le nombre de paramètres passé au programme, en comptant le nom du programme comme un paramètre. Le tableau de chaînes de caractères `argv[]` contient les pointeurs vers chacun des paramètres du programme.

Ainsi chaque `argv[i]` est de type `char *`, c'est-à-dire une chaîne de caractères, et contient un des arguments du programme. `argv[0]` est le nom sous lequel le programme a été invoqué, `argv[1]` son éventuel premier argument, etc. Le tableau `argv[]` est terminé par un pointeur spécial `NULL`.

Voici un programme simple qui affiche ses arguments.

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

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

    for (i = 0; i < argc; i++)
        printf("argument %d: '%s'\n", i, argv[i]);

    return EXIT_SUCCESS;
}
```

1.4.3 Compilation en pratique

Sous un système UNIX offrant un compilateur C, la commande pour l'invoquer est souvent `cc`. Ce programme sait comment faire effectuer toutes les phases de compilation (prétraitement, compilation, assemblage, et édition de liens).

Soit le programme suivant :

Il est possible de le compiler directement en un programme exécutable grâce à la commande `cc`; l'option `-o` spécifie le nom du fichier exécutable à générer.

```
$ cc -o exemple-001 exemple-001.c
$
```

Une fois compilé, il suffit d'invoquer ce nouvel exécutable depuis le shell.

```
$ ./exemple-001
x = 3, x^2 = 9
$
```

Attention! – Il est important de préciser que l'exécutable se trouve dans le répertoire courant (grâce à `./`). Sans cela, il ne sera pas trouvé, ou il existe un risque qu'un autre programme du même nom soit exécuté à la place du vôtre.

```

/* Premier exemple, affiche le carré d'un nombre */

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

int
main(void) {
    int x = 3;
    int xcarre = x * x;

    printf("x = %d, x^2 = %d\n", x, xcarre);

    return EXIT_SUCCESS;
}

```

FIGURE 1.1 – *exemple-001.c*

1.4.4 Options de gcc

Il est difficile et inutile de reproduire ici la documentation de plusieurs compilateurs donc nous décrirons quelques options utiles du compilateur C du GNU : gcc.

- W -Wall** demande à gcc d’afficher un nombre important de messages d’avertissement
- Werror** fait que les messages d’avertissement se transforment en messages d’erreur. Dans ce cas, le fichier résultant de la compilation ne sera pas écrit, et ce mode force donc à corriger tous les messages d’avertissement
- ansi** précise à gcc que le source C donné en entrée doit être conformant au standard C ANSI (C 89)
- std=c99** précise à gcc que le source C donné en entrée doit être conformant au standard C 99
- MM** génère une liste des dépendances d’un fichier C avec les en-têtes qu’il utilise, sous la forme d’un fragment de *Makefile*
- O** demande à gcc d’optimiser le code produit dans des limites “raisonnables” qui permettent encore de déboguer le programme facilement

Conseil – Nous vous recommandons de toujours utiliser les options `-W -Wall -std=c99`. Ainsi gcc vous aidera à trouver les erreurs les plus courantes, et vérifiera que vous n’utilisez pas d’extensions spécifiques à gcc.

Pour les curieux – Vous pouvez vous amuser à observer le résultat des différentes phases de la chaîne de compilation grâce aux options suivantes.

- E** arrête la chaîne de compilation après le prétraitement, produit un code source C prétraité
- S** arrête la chaîne de compilation après la compilation, produit un code source assembleur
- c** arrête la chaîne de compilation après l’assemblage, produit un code objet binaire

Chapitre 2

Syntaxe générale du langage C et expressions

Ce chapitre décrit une partie de la syntaxe du langage C, certaines constructions ont leur propre description dans le chapitre 5, page 27.

2.1 Considérations générales

Chaque morceau d'un programme C peut être placé dans une des trois catégories suivantes : les commentaires, les directives du préprocesseur, et les code C. Les commentaires et les directives du préprocesseur sont traités par le préprocesseur, et après la phase de prétraitement, il ne subsiste que du code C. Nous décrivons ici les particularités syntaxiques de ces trois catégories.

2.1.1 Commentaires

Les commentaires sont délimités par les deux paires de caractères `/*` et `*/`. Le contenu des commentaires est totalement libre. Cependant, ces commentaires ne peuvent pas être imbriqués : `/* toto /* foo */ */` sera considéré comme le commentaire `/* toto /* foo */` suivi du code C (invalide) `*/`. Ce fait mérite d'être noté car il peut arriver de vouloir commenter tout un fragment de code comportant déjà des commentaires.

Le standard C99 introduit dans la norme du langage C les commentaires “à la C++” qui permettent de demander au compilateur d'ignorer la fin d'une ligne grâce à la paire de caractères `//`. Par exemple :

```
int nombre_d_invites; // Comptés à la porte d'entrée
```

Il devient ainsi possible d'utiliser un commentaire `/* */` pour commenter du code contenant des commentaires `//`, ce qui permet une imbrication limitée de commentaires.

Un peu avancé – Pour commenter des fragments de code C, on préfère utiliser les directives du préprocesseur `#if 0/#endif` car ces directives peuvent être imbriquées sans difficulté entre elles et ignorent les commentaires pré-existants. De plus, en remplaçant le 0 par un 1, on peut réinsérer le code en question sans effort, ce qui est pratique pour effectuer des tests.

2.1.2 Code C

En C, les intructions ainsi que les déclarations sont terminées par un point-virgule (`;`). C'est une source d'erreur fréquente que d'oublier ce point-virgule.

Le formatage du texte n'a principalement pas d'incidence sur la sémantique du code C, la présence d'un ou plusieurs espaces et/ou retours à la ligne est équivalente à la présence d'une unique espace. Le formatage est en revanche crucial pour faciliter la lecture du code source.

Les deux fonctions suivantes ne diffèrent que du point de vue de leur formatage. Le style de la deuxième forme doit être préféré pour faciliter la lecture.

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

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

2.1.3 Directives du préprocesseur

Les directives du préprocesseur tiennent sur une ligne¹ et sont aisément reconnaissables car le premier caractère de la ligne autre qu'une espace est un dièse (#). Cela permet au préprocesseur d'effectuer son travail avec une connaissance limitée du reste du langage C.

#include Grâce à la directive **#include**, il est possible d'inclure le contenu d'un autre fichier contenant du code C. En pratique, on se limite à inclure des fichiers dits "d'en-tête", contenant seulement des déclarations, et dont le nom se termine en **.h** (pour *header*, en-tête).

On peut spécifier le nom du fichier à inclure soit entre guillemets ("), soit entre chevrons (< et >). La seule différence étant que dans le premier cas, le fichier est cherché en priorité dans le répertoire courant, alors que dans le deuxième cas il n'est cherché que dans les répertoires standards du compilateur. Ainsi, vous incluez vos propres fichiers d'en-tête avec des guillemets, et les fichiers d'en-tête de la bibliothèque standard avec des chevrons.

#define Il est possible de définir des macros grâce à la directive **#define**. Ces macros peuvent prendre des paramètres et ressemblent alors à des fonctions. Cependant, lorsqu'une macro est utilisée, le contenu de sa définition est mis en lieu et place de son apparition.

Les macros peuvent être utilisées pour définir des constantes, par exemple :

```
#define AGE_MAJORITE 18
...
    if (age >= AGE_MAJORITE) {
        autoriser_vote();
    }
```

Ou plus généralement pour écrire des macros paramétrées. Dans ce cas il appartient de faire attention car leur utilisation ressemble à un appel de fonction mais il s'agit réellement de remplacement de texte. Par exemple :

```
#define MAX(a, b) (((a) < (b)) ? (a) : (b))
...
    int max = MAX(2, 3);
```

1. Il est toujours possible de finir une ligne du code source par un anti-slash (\) afin que cette fin de ligne soit ignorée par le compilateur

Attention! – Il est facile d’écrire des macros prenant des paramètres qui auront des comportements étranges si on oublie que ce ne sont pas des fonctions. En particulier, dans la définition de la macro il est important de parenthéser les paramètres formels au cas où ils seraient étendus en des constructions syntaxiques complexes. Il faut aussi garder à l’esprit que si un paramètre formel apparaît plusieurs fois, il sera *potentiellement évalué plusieurs fois*, ce qui aura un impact si ce paramètre est remplacé par une expression ayant des effets de bord. Cette dernière raison justifie la convention d’écrire le nom de ces macros en majuscule pour bien les distinguer des fonctions.

#if/#else/#endif Grâce à la directive **#if**, il est possible de faire de la compilation conditionnelle de code. Dans l’exemple suivant, une seule des deux lignes contenant des appels de fonction sera vue par la deuxième phase de compilation.

```
#if defined(HAVE_SOUND)
    sound_make_beep();
#else
    screen_flash();
#endif
```

Le prédicat `defined` permet de tester si une macro du préprocesseur est définie.

2.2 Identificateurs

En langage C, de nombreuses entités portent des noms (fonctions, variables, macros du préprocesseur, étiquettes, noms de types, noms de structures, ...). Ces noms sont appelés identificateurs et doivent satisfaire l’expression régulière suivante, où l est une lettre parmi souligné (`_`, *underscore*) et les lettres de l’alphabet (sans accents), et c est une lettre de l ou un chiffre :

$$l(c)^*$$

Ainsi `i`, `nombre_de_fourchettes`, et `_OS` sont des identificateurs valides. La distinction majuscule/minuscule est significative. Les identificateurs commençant par `_` sont réservés pour le système par convention, vous ne devriez pas les utiliser.

2.3 Syntaxe des expressions

Une expression est une construction syntaxique qui a la particularité d’avoir une valeur. Les expressions du langage C ressemblent aux expressions mathématiques classiques, par exemple :

Expression	Valeur
<code>2</code>	2
<code>x</code>	valeur de la variable <code>x</code>
<code>3 + 2 * x</code>	$3 + 2x$
<code>f(3 * x) + g(y)</code>	$f(3x)+g(y)$

Certaines expressions ont des effets de bord, par exemple parce qu’une fonction appelée a un effet de bord, ou parce qu’elles contiennent une affectation. Par exemple :

Expression	Valeur	Effet de bord
$x = 2$	2	x prend la valeur 2
$x = y = 2$	2	x et y prennent la valeur 2
$(x = 2) * 3$ (<i>style !</i>)	6	x prend la valeur 2
$f(2)$	$f(2)$	dépendant de f

2.3.1 Priorité et associativité des opérateurs

Pour évaluer une expression composée de plusieurs opérateurs, il est nécessaire de savoir dans quel ordre les différents opérateurs la composant doivent être évalués. Pour imposer un ordre partiel sur l'évaluation, il est possible d'utiliser des parenthèses. Ainsi, l'expression $(3 * x) + 2$ précise que la multiplication doit être effectuée avant l'addition. Mais on est habitués à omettre des parenthèses dans les expressions mathématiques, car certains opérateurs sont prioritaires sur d'autres. Notamment pour cet exemple, l'opérateur $*$ est prioritaire sur l'opérateur $+$ et on peut donc écrire l'expression ci-avant comme $3 * x + 2$.

Lorsque des expressions ne contenant pas de parenthèses sont construites avec des opérateurs de même priorité (le même opérateur par exemple), il devient nécessaire d'introduire une autre notion pour placer les parenthèses implicitement. On dit qu'un opérateur est *associatif de gauche à droite* si une séquence de tels opérateurs s'évalue "de gauche à droite". Par exemple l'opérateur de soustraction $-$ est associatif de gauche à droite, et $25 - 5 - 10$ s'évalue comme $(25 - 5) - 10$ et vaut donc 10.

2.3.2 Opérateurs arithmétiques

Les opérateurs arithmétiques binaires disponibles en C sont l'addition $+$, la soustraction $-$, la multiplication $*$, la division $/$, le modulo $\%$. Il existe aussi les opérateurs arithmétiques unaires d'opposé $-$ et (pour la symétrie) de conservation de signe $+$.

Tous ces opérateurs s'appliquent aussi bien aux nombres flottants qu'aux nombres entiers hormis l'opérateur modulo qui ne s'applique qu'aux entiers.

Les priorités et associativités de ces opérateurs sont celles de l'arithmétique usuelle. Ainsi l'expression C $x + 3 * y$ a la même valeur que $x + (3 * y)$.

2.3.3 Opérateurs d'affectation

En C, il existe plusieurs opérateurs binaires permettant de modifier la valeur d'une entité. Ces opérateurs effectuent une action (un effet de bord), mais forment quand même des expressions dont la valeur est la valeur finalement affectée au premier argument ; cela permet notamment de faire des affectations en cascade. Le principal est l'opérateur d'affectation $=$ dont l'opérande gauche doit être une valeur-g (*l-value*), c'est-à-dire une expression représentant un emplacement mémoire. Par exemple, si x est une variable, x est une valeur-g car on peut affecter une valeur à x , mais $2 * x$ n'est pas une valeur-g car cette expression représente uniquement une valeur, et pas un emplacement en mémoire.

Un peu avancé – On définit quand même les valeurs-g comme étant des expressions car elles ne se limitent pas à des variables scalaires. Par exemple, si p est une structure contenant un champ x , $p.x$ est une valeur-g. De même, en utilisant des pointeurs, si la fonction $f()$ retourne un pointeur, alors $*f()$ est une valeur-g, etc.

L'opérateur $=$ affecte simplement la valeur de son deuxième argument (qui doit être une expression) à l'entité décrite par son premier argument (qui doit être une valeur-g).

Chaque opérateur arithmétique binaire a un pendant réalisant en même temps une affectation. Par exemple, l'opérateur `+=`, dérivé de `+` permet d'ajouter à son premier argument la valeur de son deuxième argument. Les opérateurs de cette forme sont `+=`, `-=`, `*=`, `/=`, et `%=`. La sémantique d'une expression `expr1 op= expr2` est `expr1 = expr1 op expr2` à ceci près que `expr1` ne sera évaluée qu'une seule fois.

Pour ajouter ou soustraire 1 à une valeur-g, on peut utiliser les opérateurs unaires `++` et `--`, mais ici une subtilité existe selon que l'opérateur se situe à gauche ou à droite de la valeur-g à modifier. Dans les deux cas, le contenu de la valeur-g sera modifié d'une unité, mais la valeur de l'expression complète sera la valeur précédente de la valeur-g si l'opérateur est après, et la valeur après modification si l'opérateur est avant.

Si par exemple `x` vaut 2, alors `x++` vaut 2 et `++x` vaut 3, même si dans les deux cas `x` vaudra 3 après évaluation de l'opérateur `++`.

Les opérateurs d'affectation ont une priorité très faible, ce qui permet d'écrire `x = 2 * y` pour affecter à `x` la valeur `2 * y` au lieu de `x = (2 * y)`. Aussi, les opérateurs d'affectation sont associatifs de droite à gauche pour pouvoir faire des affectations en cascade : `x = y = 0` est interprété "naturellement" comme `x = (y = 0)`.

Un peu avancé – Les opérateurs sur les bits, vus plus loin dans la sous-section 2.3.6, donnent aussi lieu à des opérateurs d'affectation `&=`, `|=`, `^=`, `<<=`, et `>>=`.

2.3.4 Opérateurs de comparaison

Il est possible de comparer les valeurs de deux expressions avec l'opérateur `==` qui prend la valeur 0 si les deux expressions sont différentes, et la valeur 1 dans le cas contraire. Voir la sous-section 3.1.4, page 15 pour une description générale de la notion de valeur booléenne en C. L'opérateur `!=` est exactement la négation de l'opérateur `==`. Il existe aussi les opérateurs classiques de test d'inégalités entre nombres : `<=`, `<`, `>`, et `>=`.

Les opérateurs de comparaison sont associatifs de gauche à droite, sont moins prioritaires que les opérateurs arithmétiques (ce qui permet de comparer des expressions arithmétiques sans ajouter de parenthèses), et sont plus prioritaires que les opérateurs booléens (ce qui permet de combiner des comparaisons pour des conditions plus complexes sans ajouter de parenthèses).

Attention ! – Il faut bien garder à l'esprit que le résultat d'une comparaison est soit 0, soit 1. Si vous souhaitez comparer une valeur avec deux bornes, ce qui s'écrit en mathématiques usuelles $1 < x < 5$ ne peut pas s'écrire en C `1 < x < 5`. Cette dernière expression est toujours vraie puisque `1 < x` vaut soit 0 soit 1, et donc est une quantité toujours strictement inférieure à 5. On écrira plutôt `1 < x && x < 5`.

2.3.5 Opérateurs booléens

Il est fréquent de vouloir combiner des conditions avec les opérateurs logiques *ou* et *et*. Le C offre pour cela respectivement les opérateurs `||` et `&&`, ainsi que l'opérateur unaire de négation logique noté `!`. Ces opérateurs prennent en paramètre des expressions booléennes et ont une valeur booléenne, telles que décrites dans la sous-section 3.1.4.

L'associativité de ces opérateurs est de gauche à droite et leur priorité est très faible afin de pouvoir combiner logiquement des expressions plus complexes (de comparaison notamment).

Attention! – Les opérateurs `&&` et `||` ont la particularité de spécifier l’ordre dans lequel leurs arguments sont évalués : de gauche à droite. De plus, si l’opérande de gauche vaut vrai (resp. faux), alors l’opérateur `||` (resp. `&&`) *n’évaluera pas* l’opérande de droite. Ceci a des conséquences importantes si l’opérande de droite a des effets de bord.

2.3.6 Opérateurs sur les bits

Le langage C précise bien que les entiers sont codés en binaire, et propose des opérateurs pour manipuler les entiers bit par bit. Ces opérateurs sont comme les opérateurs arithmétiques mais opèrent sur chacun des bits constituant les entiers passés en paramètre.

Les opérateurs `~`, `&`, et `|` calculent respectivement bit après bit le résultat de la négation de son opérande, du *et* ou du *ou* logique de chacun des bits correspondants de leurs opérandes.

Les opérateurs `<<` et `>>` ont pour valeur l’entier donné comme paramètre à gauche, décalé à gauche (`<<`) ou à droite (`>>`) du nombre de bits donné par l’opérande de droite.

Attention! – Dans le cas de l’opérateur `>>`, les bits de plus fort poids doivent être “inventés” puisqu’ils n’étaient pas dans l’entier de départ. Si l’opérande de gauche est une entité non signée, des zéros seront choisis. S’il est signé, c’est le bit de plus fort poids de l’opérande de gauche qui sera répété pour ces nouveaux bits dans le résultat.

2.4 Syntaxe des instructions

En C, les instructions d’un programme sont données à l’intérieur de blocs, délimités par des accolades (`{` et `}`). Notamment, le contenu de toute fonction est défini par un bloc. Chaque bloc contient une succession de déclarations et d’instructions. Depuis la norme C99, déclarations et instructions peuvent être mélangées à loisir, mais dans les normes précédentes, toutes les déclarations d’un bloc devaient précéder la première instruction de ce bloc.

Il y a plusieurs façons de former une “instruction” en langage C. Un bloc est aussi une instruction : il est possible d’imbriquer les blocs. Pour former une instruction à partir d’une expression, il faut ajouter un point-virgule (`;`) à la fin de l’expression. Les autres constructions du langage qui forment des instructions (exécution conditionnelle, boucles, ...) sont décrites dans le chapitre 5, page 27.

Par exemple, `x = 2` est une expression, et `x = 2;` et `{ x = 2; }` sont grammaticalement des instructions du langage C.

Chapitre 3

Types du langage C

Le langage C est un langage typé : chaque variable doit être déclarée avec son type avant toute utilisation. Les types sont soit simples (entier, flottant, ...), soit composés (tableau d'entiers, ...). Les types simples sont appelés types scalaires. On dit que le langage C est *faiblement typé* car il est possible de forcer le compilateur à accepter de changer le type d'une variable, et beaucoup de valeurs peuvent être promues à des types différents. Par exemple on peut utiliser un entier là où un flottant est attendu, et le compilateur fera la conversion de type nécessaire de manière transparente.

3.1 Types scalaires

En C, il est possible de manipuler des booléens¹, des entiers, des flottants, et des caractères.

Dans le cas des types scalaires, la syntaxe pour définir une variable avec son type est simplement de précéder le nom de la nouvelle variable par le nom du type. Par exemple :

```
char c;  
int entier;
```

On distingue principalement deux familles de types scalaires : les entiers et les flottants. Des conversions automatiques ont lieu lorsqu'une valeur d'un type numérique est utilisée là où on attendrait une valeur d'un autre type numérique.

3.1.1 Les entiers

Les types entiers sont `char`, `short`, `int`, `long`, et `long long`². `short`, `long`, et `long long` sont des abréviations respectivement de `short int`, `long int`, et `long long int`. Les types flottants sont `float` et `double`.

Les types entiers acceptent aussi des variantes non-signées en ajoutant le mot-clé `unsigned` devant le spécificateur de type.

Attention! – Le type `char` est par défaut signé ou non-signé, en fonction de la plate-forme sur laquelle on se trouve. Le langage C fournit le mot-clé `signed` pour forcer un des types entiers à être signé. Ce mot-clé n'a d'intérêt pratique que pour le type `char`.

1. depuis la norme C99, les normes précédentes utilisent les entiers pour stocker les booléens
2. depuis la norme C99

Limites des types entiers

Les valeurs entières qui peuvent être stockées dans les objets de type entier ne sont pas totalement spécifiées par la norme. Plus précisément, la taille des différents types entiers n'est pas spécifiée, la norme ne forçant que des valeurs minimales pour les bornes. L'arithmétique étant garantie s'effectuer selon les lois de l'arithmétique modulo 2, la taille d'un type entier peut s'exprimer comme le nombre de bits offerts par le type pour stocker un entier.

type	nombre de bits minimum	valeur min.	valeurs max.
signed char	8	SCHAR_MIN	SCHAR_MAX
unsigned char	8	0	UCHAR_MAX
short	16	SHRT_MIN	SHRT_MAX
unsigned short	16	0	USHRT_MAX
int	16	INT_MIN	INT_MAX
unsigned int	16	0	UINT_MAX
long	32	LONG_MIN	LONG_MAX
unsigned long	32	0	ULONG_MAX
long long	64	LLONG_MIN	LLONG_MAX
unsigned long long	64	0	ULLONG_MAX

Dans ce tableau, pour chaque type entier est donnée la taille minimale de ce type garantie par la norme, et les constantes définies dans le fichier `limits.h` qui donne les valeurs précises dépendant du compilateur et de l'environnement.

Il faut noter qu'à l'heure actuelle énormément de programmes font l'hypothèse que le type `int` et son équivalent non signé ont une taille d'au moins 32 bits, ce qui est une hypothèse raisonnable à notre époque mais n'est pas garanti par la norme.

3.1.2 Les constantes entières

Les constantes entières sont écrites naturellement en décimal. Par exemple, 18 est de type `int` et vaut 18 en base 10, ce que nous notons $(18)_{10}$. Il est aussi possible de spécifier des constantes dans deux autres bases : en octal (base 8) et en hexadécimal (base 16), grâce à des préfixes particuliers.

Ainsi, tout nombre commençant par un zéro (qui ne devrait pas changer la valeur du nombre *a priori*) est en fait interprété en base 8. La constante entière 025 vaut donc $(25)_8 = (21)_{10}$. De même, tout nombre précédé de 0x est interprété en hexadécimal. Par exemple, 0x1b vaut $(1b)_{16} = (27)_{10}$.

Attention! – Notez bien que le zéro au début d'une constante entière change la base dans laquelle le nombre est interprété.

Le langage C précise qu'une constante entière est par défaut de type `int`, sauf si elle est trop grande pour être représentée dans un `int`. Dans ce cas, s'il s'agit d'une constante en octal ou en hexadécimal, le type `unsigned int` est essayé. Dans tous les cas, c'est ensuite le type `long` qui est essayé, puis le type `unsigned long`.

Comme il est parfois nécessaire de choisir le type de la constante entière même si elle reste dans l'intervalle représentable par un type plus petit, il est possible de suffixer la constante par les lettres suivantes :

u la constante devient de type `unsigned int`, exemple : 18u

l la constante devient de type `long`, exemple : 0x3al

11 la constante devient de type `long long`

Il est possible de cumuler le suffixe `u` avec l'un des deux autres pour spécifier des constantes de type `unsigned long` et `unsigned long long`. Par exemple `0777u1`

3.1.3 Les flottants

Le langage C permet de manipuler des nombres à virgule flottante. Il introduit pour cela trois types : `float`, `double`, et `long double`, qui peuvent contenir des nombres à virgule de plus en plus grands et précis.

Les constantes flottantes sont indiquées par la présence d'un point décimal (`.`) dans leur écriture. Par exemple `3.14` est une constante flottante. Le type par défaut pour ces constantes est `double` et il faut ajouter le suffixe `f` pour que la constante soit de type `float`. Ainsi `3.14` est de type `double` et vaut `3,14`; `3.14f` est de type `float` et vaut aussi `3,14`.

3.1.4 Les booléens

Les valeurs de vérité en C sont 0 qui signifie faux, et toute quantité différente de 0 qui signifie vrai. Lorsqu'un opérateur du langage C a une valeur booléenne, "vrai" est toujours représenté exactement comme 1. En revanche, lorsqu'un opérateur prend un paramètre booléen, n'importe quelle valeur différente de 0 sera considérée comme "vrai".

Un peu avancé – Cette définition des valeurs "vrai" et "faux" fait que les valeurs booléennes sont naturellement représentées et stockées dans des entiers. Cela pose deux inconvénients : une valeur booléenne occupe plus de place que nécessaire (un entier au lieu d'un bit), et de plus, elle fait jouer aux entiers un rôle ambigu puisqu'en lisant une déclaration de variable, il est difficile de savoir si elle représente un booléen ou réellement un entier. Pour résoudre cela, la norme C99 introduit explicitement un type `_Bool` pour les booléens. Ce type permet de stocker une des deux valeurs 0 ou 1. Ainsi les constantes booléennes restent 0 et 1, mais il existe un type spécifique pour les stocker.

Si vous souhaitez utiliser le type `_Bool`, nous recommandons d'inclure le fichier d'en-tête `<stdbool.h>` qui définit le type `bool` comme synonyme de `_Bool`. Ce fichier d'en-tête définit aussi les macros `true` et `false`, valant respectivement 1 et 0, mais nous n'en recommandons pas l'usage.

3.2 Tableaux

Il arrive très fréquemment de vouloir manipuler un grand nombre de données d'un même type. Le langage C permet de définir des tableaux, qui sont des séquences de valeurs d'un même type, accessibles en temps constant par un entier appelé *indice*. Les indices d'un tableau de taille n sont les entiers 0 à $n - 1$.

La syntaxe pour accéder à un élément d'indice i d'un tableau `t` est `t[i]`, qui est une valeur-g. Ainsi il est possible de lire la valeur d'un élément du tableau mais aussi d'y affecter une valeur compatible avec le type des éléments du tableau. Syntaxiquement, on peut mettre entre les crochets n'importe quelle expression de type compatible avec `int`.

Il est très facile de passer en revue tous les éléments d'un tableau si on connaît la taille du tableau. Ainsi, pour calculer un élément maximal d'un tableau d'entiers, on pourra procéder ainsi :

```
int
max(int tab[], int n) {
    int i;
```

```

int max = tab[0];

for (i = 1; i < n; i++)
    if (tab[i] > max)
        max = tab[i];

return max;
}

```

3.2.1 Définition et initialisation d'un tableau

Il est possible de définir un nouveau tableau avec la syntaxe suivante :

```
<type> <identificateur> [<expr-constante-entiere>]
```

qui définit un tableau dont le nom est *<identificateur>*, dont chaque élément est de type *<type>*, et dont le nombre d'éléments est la valeur de l'*<expr-constante-entiere>*.

Lors de la définition d'un tableau, il est possible de fournir les valeurs d'initialisation des éléments du tableau. Pour cela, on fait suivre la déclaration du signe égal (=) et on liste les éléments séparés par des virgules entre deux accolades. Par exemple,

```
int tab[10] = { 8, 1, 2, 5 };
```

définit le tableau `tab` pouvant contenir 10 éléments de type `int`, et dont les quatre premiers sont initialisés :

```
tab → 

|   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|
| 8 | 1 | 2 | 5 | ? | ? | ? | ? | ? | ? |
|---|---|---|---|---|---|---|---|---|---|


```

On ne peut pas spécifier plus d'éléments d'initialisation que la taille déclarée du tableau, et si une séquence d'initialisation est donnée, on peut omettre de préciser la taille du tableau entre crochets, auquel cas le nombre d'éléments du tableau sera le nombre d'éléments de la séquence d'initialisation. Par exemple,

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

définit le tableau `tab` pouvant contenir 4 éléments de type `int`, tous initialisés :

```
tab → 

|   |   |   |   |
|---|---|---|---|
| 8 | 1 | 2 | 5 |
|---|---|---|---|


```

3.2.2 Taille d'un tableau

Un tableau occupe une certaine place en mémoire, qui est la taille d'un élément multipliée par le nombre d'éléments du tableau. Lorsque la définition du tableau est visible, `sizeof` appliqué au nom du tableau vaut la taille mémoire occupée par le tableau, en octets. Toujours dans ce cas (définition du tableau visible), on peut donc calculer le nombre d'éléments d'un tableau `tab` avec l'expression `sizeof tab / sizeof tab[0]`, qui divise la taille totale du tableau par la taille d'un élément.

Attention! – Lorsque la définition du tableau n'est plus visible, comme c'est le cas lorsque le tableau est un paramètre de la fonction courante, la taille du tableau est inconnue du compilateur, et n'est stockée nulle part en mémoire, il est donc impossible de retrouver cette information, et l'opérateur `sizeof` aura pour valeur non pas la taille du tableau mais la taille d'un pointeur vers le type d'un élément du tableau!

Pour cette raison, les fonctions prenant en paramètre des tableaux prennent presque toujours le nombre d'éléments du tableau également en paramètre.

3.2.3 Passage de tableau en paramètre d'une fonction

Lorsqu'un tableau est passé en paramètre à une fonction, son contenu n'est pas copié. Il est ainsi possible pour la fonction appelée de modifier le contenu du tableau.

Voir la sous-section 3.4.3, page 20.

3.2.4 Cas particulier des chaînes de caractères

Les chaînes de caractères sont tout simplement des tableaux dont les éléments sont de type `char`. Cependant, le langage C offre pour les chaînes de caractères un moyen plus pratique de les initialiser : pour initialiser un tableau dont les éléments sont de type `char`, il est possible de spécifier entre guillemets une chaîne de caractères. Le premier caractère de la chaîne sera copié dans le premier élément du tableau, le deuxième caractère dans le deuxième, etc. Si aucune taille de tableau n'est spécifiée, ou si la taille spécifiée le permet, le caractère nul `'\0'` sera ajouté à la fin du tableau.

Ainsi les deux définitions suivantes sont équivalentes :

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

et

```
char chaine[] = { 't', 'o', 't', 'o', '\0' };
```

Les chaînes de caractères constantes sont le seul endroit du cœur du langage où la convention de terminer les chaînes de caractères par un `\0` est imposée. Cependant c'est une convention très suivie, et notamment toutes les fonctions de la bibliothèque C standard qui manipulent des chaînes de caractères l'utilisent et l'imposent.

Cette "astuce de codage" a l'inconvénient d'empêcher une chaîne de caractères de contenir un caractère nul (puisque tout caractère nul indique la fin de la chaîne), mais présente l'avantage d'éviter de devoir passer avec toute chaîne de caractère le nombre de caractères la composant.

3.2.5 Tableaux à plusieurs dimensions

Il est possible en C de construire des tableaux de tableaux avec une syntaxe pratique : il suffit lors de la définition d'un tableau d'ajouter une paire de crochets supplémentaires avec le nombre d'éléments qui composeront une ligne du tableau résultant. Par exemple, une matrice 3×3 peut être définie comme suit :

```
int matrice[3][3] = {
    { 1, 0, 0 },
    { 0, 1, 0 },
    { 0, 0, 1 },
};
```

L'initialisation d'un tableau à plusieurs dimensions est facultatif, comme dans le cas des tableaux unidimensionnels. Autant la syntaxe de la définition que la syntaxe de l'initialisation sont cohérentes et extensibles : chaque élément du tableau `"int matrice[3]"` est lui-même un tableau de 3 éléments. On initialise donc chaque ligne comme un tableau situé à l'intérieur du tableau de lignes. L'accès aux éléments se fait en donnant les deux indices dans deux paires de crochets adjacentes. Ces notations se généralisent à un nombre de dimensions quelconques, en ajoutant des paires de crochets supplémentaires.

Par exemple, la fonction suivante multiplie deux matrices 3×3 et met le résultat dans une troisième.

```

void
matrice_multiplier(double resultat[3][3], double a[3][3], double b[3][3]) {
    int i;
    int j;
    int k;

    for (i = 0; i < 3; i++) {
        for (j = 0; j < 3; j++) {
            resultat[i][j] = 0;
            for (k = 0; k < 3; k++)
                resultat[i][j] += a[i][k] * b[k][j];
        }
    }
}

```

3.3 Structures

Les types structure permettent d'agréger plusieurs types hétérogènes en un seul type. La syntaxe du langage C permet en même temps de définir un nouveau type structure et de créer des instances de ce type. On préférera cependant dans la plupart des cas séparer ces deux actions. Ainsi, pour définir un nouveau type structure, on utilisera la syntaxe suivante :

```

struct <identificateur> {
    <declaration>;
    <declaration>;
    <declaration>;
    ...
};

```

Après une telle définition de type, le nouveau type **struct** <identificateur> est utilisable comme n'importe quel autre type de base du langage C. Une variable ayant ce type peut être passée en paramètre d'une fonction, retournée par une fonction et copiée grâce à l'opérateur d'affectation = Dans ce cas, toutes les données contenues dans la structure sont copiées.

Les déclarations qui se trouvent à l'intérieur de la définition du type structure définissent les types des *champs* constituant les structures de ce type. Etant donnée une variable **s** de type structure, on accède aux champs de cette structure grâce à la notation pointée : **s.<champ>**, qui se comporte comme une variable (et est donc une valeur-g) dont le type est donné par la déclaration de champ correspondante dans la définition du type structure.

Enfin, la définition d'une variable de type structure respecte la syntaxe déjà vue de définition d'une variable qui consiste à donner le nom du type (ici **struct** <identificateur>) puis le nom de la variable. Lors de la définition d'une variable de type structure, il est possible de l'initialiser en donnant entre accolades le contenu des champs de la structure, en respectant l'ordre des champs de la définition du type structure.

Voici un exemple :

```

struct point {
    double x;
    double y;
    double z;
};

```

```

struct sphere {
    struct point centre;
    double rayon;
};

...

struct sphere s = { { 1.5, 2.1, 4.8 }, 2.0 };

printf("Sphere de centre (%g, %g, %g) et de rayon %g\n",
    s.centre.x, s.centre.y, s.centre.z, s.rayon);

...
/* Translation en x */
s.centre.x += 3;
...

```

Un peu avancé – Il est bien sûr possible qu'un champ soit de type tableau. Dans ce cas, le tableau fait partie intégrante de la structure et le contenu du tableau sera recopié avec le reste du contenu de la structure en cas de copie par l'opérateur = de la structure entière.

3.4 Fonctions

En C, toute instruction du langage doit se trouver à l'intérieur d'une fonction : il n'est pas possible de décrire un comportement du programme à l'extérieur d'une fonction.

Les fonctions sont utilisées pour scinder un programme en entités cohérentes. La bonne pratique consiste souvent à faire une fonction pour chaque comportement du programme qui peut être autonome. On va par exemple faire une fonction pour ajouter deux matrices, une autre pour les multiplier, etc. Les fonctions peuvent s'appeler entre elles et il est donc ainsi possible d'écrire des fonctions petites qui finissent par avoir des comportements compliqués tout en gardant un programme lisible.

3.4.1 Déclaration et définition de fonction

Le type d'une fonction est constitué de la liste des types de ses paramètres et du type de sa valeur de retour. Syntactiquement, le prototype d'une fonction inclut ces informations et le nom de la fonction de la façon suivante :

$\langle type_0 \rangle \langle identificateur \rangle (\langle type_1 \rangle, \dots, \langle type_n \rangle)$

Pour déclarer une fonction, il suffit d'écrire le prototype de la fonction suivi d'un point-virgule. Pour définir une fonction, il faut ajouter dans le prototype des noms de variable pour chacun des paramètres de la fonction, juste après chacun des types des paramètres, ainsi que le corps de la fonction qui est un bloc de déclarations et d'instructions. Par exemple :

```

double norme(double, double);

double
norme(double x, double y) {
    return sqrt(x * x + y * y);
}

```

3.4.2 Retour d'une valeur par une fonction

Lorsqu'une fonction retourne une valeur, le mot-clé `return` suivi d'une expression compatible avec le type de retour de la fonction permet de quitter la fonction en retournant la valeur de l'expression.

Lorsqu'une fonction ne retourne aucune valeur, on utilise le mot-clé `void` à la place du type de retour. De même, lorsqu'une fonction ne prend aucun paramètre, on écrit `void` à la place de la liste de paramètres.

3.4.3 Copie des paramètres et des valeurs de retour

Lorsqu'un argument est passé à une fonction, sa valeur est *copiée* dans le paramètre formel de la fonction. Ainsi, les variables qui apparaissent dans le prototype d'une fonction se comportent exactement comme des variables locales initialisées à la valeur donnée par l'appelant lors de l'appel de fonction.

Si on souhaite qu'une fonction modifie une entité appartenant à une autre fonction, il faut lui passer un pointeur sur cette entité. Le pointeur sera passé par valeur (i.e. sera copié), mais l'objet pointé pourra être modifié au travers du pointeur.

Attention! – Le passage de tableaux semble déroger à la règle du passage par valeur : si on passe en paramètre à une fonction un tableau, le contenu du tableau n'est pas copié, et la fonction appelée peut donc modifier le contenu du tableau.

Du point de vue du langage, il ne s'agit en fait pas d'une exception, mais simplement le nom d'un tableau en C fait référence au pointeur vers ce tableau et non pas au tableau dans son intégralité. Ainsi lorsqu'on passe un tableau en paramètre, c'est en fait le pointeur vers le tableau qui est passé, même si syntaxiquement cela ne se voit pas.

3.5 Pointeurs

Il est possible en langage C de manipuler explicitement les adresses auxquelles se trouvent les variables et autres objets du langage. Deux opérateurs essentiels sont fournis pour déréférencer un pointeur et pour obtenir l'adresse associée à un objet.

- Si `p` est un pointeur sur un objet de type `t`, `*p` est l'objet de type `t`, se trouvant à l'adresse `p`. Cette action est appelée déréférencement du pointeur.
- Si `x` est une l-valeur de type `t`, `&x` est un pointeur vers l'objet `x`; le type de `&x` est noté `t *`.

L'opérateur `->` est un raccourci d'écriture et `p->a` est équivalent à `(*p).a`, il ne s'applique donc que dans le cas où `p` est un pointeur vers un objet de type structure.

Si on n'utilise que les opérateurs `&` et `*`, on se sert des pointeurs comme des références à des objets. Il est aussi possible en C d'aller plus loin dans la manipulation des adresses et d'ajouter ou retrancher des valeurs entières aux pointeurs pour changer l'adresse vers laquelle ils pointent. Cette manipulation est potentiellement dangereuse. Voici cependant sa sémantique : si on écrit `p + i` et que `p` est un pointeur, l'adresse contenue dans `p` sera incrémentée de `i * sizeof *p`, autrement dit `i` fois la taille d'un élément de type `*p`.

Ceci permet d'introduire un autre raccourci syntaxique qui permet de retrouver la syntaxe des tableaux : lorsqu'on écrit `p[i]` et que `p` est un pointeur, c'est équivalent à `*(p + i)`.

Voilà pour ce qui concerne l'utilisation dans le code de pointeurs. Pour déclarer ou définir des pointeurs, on utilise aussi une étoile, mais après le nom du type de l'objet pointé (d'où un risque de confusion au début puisque l'étoile est aussi utilisée pour le déréférencement).

Enfin, il existe un type très particulier en C qui s'appelle `void *` et qui permet de stocker un pointeur vers n'importe quel type. Il n'est pas possible d'appliquer à ce pointeur d'autre opérateur que `*`, `&`, ou `=`.

Chapitre 4

Déclarations et définitions

La *déclaration* d'une entité du langage (variable, fonction) permet de spécifier son nom et son type. La *définition* d'une entité contient les mêmes informations que la déclaration mais en plus réserve la mémoire nécessaire pour stocker cette entité.

Il peut donc y avoir plusieurs déclarations d'une même entité à condition qu'elles soient consistantes, mais il ne peut y avoir qu'une seule définition, sans quoi un conflit apparaîtrait au moment de l'édition de liens¹ puisque deux emplacements mémoire différents correspondraient à une même entité.

4.1 Visibilité des identificateurs

Lorsqu'on utilise un identificateur, il dénote une entité qui doit au préalable avoir été déclarée. Il faut de plus que l'endroit où se trouve cette utilisation soit dans la zone de visibilité de la définition de cette entité. Ainsi on peut distinguer plusieurs grandes catégories de visibilités :

- locale : lorsqu'on définit une variable dans un bloc, elle est dite locale à ce bloc et elle n'est visible que dans ce bloc et dans les sous-blocs qui ne redéfinissent pas une variable du même nom.
- globale : lorsqu'on définit une variable à l'extérieur de tout bloc, elle est dite globale et est visible depuis tous les fichiers qui seront liés pour former le programme final. Il faut toutefois la *déclarer* dans chaque fichier qui l'utilise. Pour cela, on utilise la syntaxe de la définition globale précédée du mot-clé **extern**. Pour les prototypes de fonctions, le mot-clé **extern** est facultatif.
- fichier : on peut aussi définir une variable "globale" dont la visibilité se limite au fichier source courant. Pour cela il faut préfixer toute définition ou déclaration du mot-clé **static** et les situer en dehors de tout bloc.

On utilise aussi le mot "portée" pour parler de "visibilité".

On essaie autant que possible de ne pas utiliser de variables globales car leur visibilité couvre la totalité du programme compilé, y compris les bibliothèques avec lesquelles il est compilé. Les variables globales peuvent donc introduire des incompatibilités avec certaines bibliothèques à cause de collisions de noms.

1. L'éditeur de liens ne prévient pas forcément de ces conflits pour les variables afin de pouvoir lier aussi des programmes Fortran qui ont une sémantique différente. Il n'en reste pas moins qu'en C la définition multiple d'une variable est une erreur de programmation.

4.2 Durée de vie des entités

La visibilité d'une entité permet de savoir à quel endroit dans le code source quel nom fait référence à quelle entité. Cependant, une variable peut être masquée par une autre pendant la durée d'un bloc, et les variables visibles seulement dans un fichier par définition ne le sont pas dans les autres fichiers. Cependant, ça n'empêche pas ces variables d'exister en mémoire même pendant l'exécution des portions de code dans lesquelles elles ne sont pas visibles.

La durée pendant laquelle une variable existe en mémoire s'appelle sa durée de vie. Il s'agit d'une notion qui fait référence au programme lorsqu'il est en cours d'exécution.

Une variable globale ou une variable dont la visibilité est limitée au fichier courant ont pour durée de vie la totalité de la durée d'exécution du programme.

Une variable locale a la durée de vie de l'instance de la fonction dans laquelle elle se trouve. C'est la raison pour laquelle on ne peut pas retourner d'une fonction un tableau qui y a été défini localement : sa durée de vie est strictement incluse dans la durée de vie de l'instance de la fonction.

4.3 Programmation modulaire

Il est possible d'utiliser les règles de visibilité, de durée de vie, et le préprocesseur du langage C pour écrire des programmes "modulaires", c'est-à-dire séparés en modules.

L'intérêt d'une telle programmation est de simplifier chaque morceau d'un programme en isolant dans un module les opérations associées à un des concepts manipulés par le programme. Par exemple, un programme de rendu d'images pourrait avoir un module de manipulation de matrices, un pour les lampes, un pour les rayons, un par type d'objets représentables dans une scène, etc. Les modules ne se situent pas forcément au même niveau conceptuel (par exemple les matrices joueraient le rôle d'un module utilitaire dans un programme de rendu alors qu'elles seraient un module central dans un programme d'algèbre linéaire); ce qui importe est que les cloisons entre modules soient bien définies.

La notion de module fait intervenir la notion d'interface. Il s'agit d'une liste de définitions de types, de déclarations globales, et de définitions de macros du préprocesseur qui permettent de manipuler le ou les concepts représentés par le module. Par exemple, un module vecteur pourrait avoir une interface comme celle-ci :

```
void vecteur_multiplie_par_scalaire(double *, int);
double vecteur_norme_euclidienne(double *, int);
double vecteur_produit_scalaire(double *, double *, int);
```

Ces déclarations permettent au compilateur de savoir comment elles doivent être appelées. En pratique elles seraient associées à un commentaire ou à une documentation, par exemple pour signaler au programmeur que le paramètre de type `int` donne la taille du vecteur.

Toutes ces déclarations doivent être connues de tous les modules qui vont utiliser le module vecteur. Elles doivent donc apparaître dans un fichier d'en-tête (souvent terminé en `.h` pour "header"), qui sera inclus par tout utilisateur du module. On suggère aussi de l'inclure dans le fichier d'implémentation du module au tout début afin de vérifier qu'il est autonome, et aussi de vérifier que les déclarations correspondent bien aux définitions des entités. Le fichier d'implémentation est le seul endroit où les définitions et les déclarations peuvent être comparées.

Il arrive que des fichiers d'en-tête aient besoin d'autres fichiers d'en-tête. On peut rapidement arriver à des situations où des fichiers sont inclus deux fois, ce qui peut aboutir à des redéfinitions de types qui sont refusées par le compilateur. Pour éviter cela, on prend l'habitude d'entourer tout le contenu des fichiers d'en-tête d'un test sur une macro du préprocesseur qui correspond à

ce fichier, ce qui permet de garantir que son contenu ne sera inclus qu'une seule fois. Par exemple pour le module vecteur :

```
#ifndef VECTEUR_H_
#define VECTEUR_H_

void vecteur_multiplie_par_scalaire(double *, int);
double vecteur_norme_euclidienne(double *, int);
double vecteur_produit_scalaire(double *, double *, int);

#endif /* VECTEUR_H_ */
```

La première fois que le fichier est inclus, la macro `VECTEUR_H_` n'est pas définie, elle est donc définie et les prototypes sont lus par le compilateur. Les fois suivantes où le fichier est inclus, la macro `VECTEUR_H_` est définie, et le contenu du fichier n'est donc lu (et ignoré) que par le préprocesseur.

Notez que tous les identificateurs utilisés par le module sont préfixés par `vecteur_` afin de limiter au maximum les risques de collision avec les noms d'un autre module.

Chaque module est implémenté dans un fichier C qui peut donc être compilé séparément en ajoutant le drapeau `-c` au compilateur, ce qui génère un fichier objet (dont le nom se termine en `.o` sous UNIX).

Enfin, tous les fichiers objet doivent être liés entre eux et avec d'éventuelles bibliothèques supplémentaires, toujours grâce au compilateur C qui sait qu'il doit faire l'édition de liens parce que les fichiers passés en paramètre se finissent en `.o`.

Chapitre 5

Structures de contrôle

Les structures de contrôle du langage C permettent d'altérer l'exécution séquentielle d'un programme en fonction d'un choix. On peut distinguer les conditions, qui permettent de choisir un chemin d'exécution parmi plusieurs, et on peut distinguer les boucles, qui permettent d'itérer un chemin d'exécution plusieurs fois.

5.1 Conditions

Il est fréquent dans un programme de devoir réagir différemment en fonction d'un certain critère. Par exemple, on peut vouloir déclencher l'affichage d'un message seulement si une fonction nous dit qu'une batterie est faible.

```
if (batterie_est_faible(batterie))
    printf("Attention: la batterie est faible.\n");
```

La syntaxe est la suivante :

```
if (<expr>)
    <instruction1>
else
    <instruction2>
```

et la partie `else` est facultative. La sémantique est que si $\langle expr \rangle$ s'évalue à vrai, $\langle instruction_1 \rangle$ sera exécutée, sinon $\langle instruction_2 \rangle$, si elle est donnée par un `else`, sera exécutée. Notez que les parenthèses font partie de la syntaxe et sont obligatoires.

Nous rappelons ici qu'une "instruction" peut être formée d'un bloc d'instructions (cf. section 2.4, page 12).

Il est fréquent de devoir cascader beaucoup de tests pour sélectionner une action dépendant d'une variable pouvant prendre de nombreuses valeurs. On pourrait trouver par exemple dans un programme une cascade de `if/else if` comme ceci :

```
if (couleur == ROUGE)
    ...
else if (couleur == VERT)
    ...
else if (couleur == BLEU)
    ...
```

Le langage C propose un raccourci syntaxique pour ce genre de tests et permet de comparer une valeur *entière* à un ensemble de valeurs possibles. Il s'agit de l'instruction `switch` dont la syntaxe est la suivante :

```
switch (<expr>) {
case <constante1>:
    <instructions1>
case <constante2>:
    <instructions2>
case <constante3>:
    <instructions3>
    ...
}
```

La première constante $\langle \text{constante}_i \rangle$ égale à la valeur de $\langle \text{expr} \rangle$ déclenche l'exécution de toutes les suites d'instructions à partir de $\langle \text{instructions}_i \rangle$, jusqu'à la dernière accolade du `switch` ou jusqu'à la première instruction `break` rencontrée.

Attention! – Il est fréquent d'oublier un `break`; et dans ce cas, le programme exécutera du code non prévu.

La traduction de notre exemple par un `switch` deviendrait donc :

```
switch (couleur) {
case ROUGE:
    ...
    break;
case VERT:
    ...
    break;
case BLEU:
    ...
    break;
    ...
}
```

Enfin, on peut finir un `switch` par un cas `default`: pour exécuter une séquence d'instructions dans le cas où aucun des `case` ne correspondrait à la valeur de l'expression.

5.2 Boucles

Il est fréquent dans un programme de vouloir répéter un comportement un nombre non déterminé à l'avance de fois. Il est alors nécessaire d'utiliser des boucles. Le langage C offre trois constructions syntaxiques de haut niveau permettant de réaliser des boucles, chacune étant adaptée à une utilisation particulière.

5.2.1 La boucle `while`

De l'anglais *while* (tant que), la boucle `while` du C permet de répéter une instruction tant qu'une condition donnée est vraie. La syntaxe de la boucle `while` est la suivante :

```
while (<expr>)
    <instruction>
```

L'⟨*instruction*⟩ est appelée le corps de la boucle. La sémantique d'une boucle `while` est la suivante :

1. évaluer ⟨*expr*⟩
2. si cette valeur est “faux” (0), fin de la boucle
3. sinon exécuter ⟨*instruction*⟩ et revenir en 1.

On constate notamment grâce à la sémantique qu'il est possible que le corps d'une boucle ne s'exécute jamais, si la condition de boucle est fausse avant la boucle. Voici un exemple de boucle `while` :

```
unsigned int nombre;
...
while (nombre != 0) {
    if (nombre % 2 == 1)
        nombre_de_bits++;
    nombre >>= 1;
}
```

5.2.2 La boucle `do-while`

Il est parfois utile d'exécuter le corps de la boucle au moins une fois avant d'effectuer le test. C'est à ça que sert la boucle `do-while`. La syntaxe est la suivante :

```
do
    ⟨instruction⟩
while (⟨expr⟩);
```

La sémantique reflète ce changement d'ordre dans l'évaluation de la condition de continuation de la boucle :

1. exécuter ⟨*instruction*⟩
2. évaluer ⟨*expr*⟩
3. si cette valeur est “vrai” ($\neq 0$), revenir en 1.

Et voici un exemple d'utilisation où il est clair qu'il n'est pas possible de tester la condition avant d'exécuter le corps de la boucle :

```
int c;

/* Lit et ignore tous les caractères jusqu'à la fin de fichier */
do {
    c = lire_un_caractere();
} while (c != FIN_DE_FICHER);
```

5.2.3 La boucle `for`

Beaucoup de boucles `while` réalisent un idiome de la forme initialisation—test condition de boucle—actions—avancer. Par exemple, afficher tous les entiers de 1 à 10 rentre exactement dans ce cadre. On va initialiser une variable à 1, tester si elle est plus petite que 10, l'afficher, et l'incrémenter avant de reboucler.

Le C introduit pour cet idiome une syntaxe à la fois abrégée et générique : la boucle `for`. Voici sa syntaxe et sa traduction immédiate en termes d'une boucle `while` :

<pre>for (<expr₁>; <expr₂>; <expr₃>) <instruction></pre>	→	<pre><expr₁>; while (<expr₂>) { <instruction> <expr₃>; }</pre>
---	---	---

L'intérêt de la boucle `for` est double : d'abord elle est plus concise, mais son véritable intérêt est qu'en respectant sa syntaxe, on informe le lecteur du code qu'on est en train d'utiliser cet idiome classique. On notera au passage que les expressions en jeu ne sont en aucune façon restreintes et qu'on peut utiliser la boucle `for` de manière plus éloignée de l'idiome de départ, pour gagner en concision. Voici un exemple simple :

```
int i;

for (i = 1; i <= 10; i++)
    printf("%d\n", i);
```

5.2.4 Sauts

Il est possible en langage C d'effectuer des sauts, c'est-à-dire de faire en sorte que l'exécution du programme se poursuive à un autre endroit.

Sauts spécifiques aux boucles

Lorsqu'on se trouve à l'intérieur d'une boucle (`for`, `while`, ou `do-while`), deux instructions supplémentaires sont disponibles : `break` et `continue`.

L'instruction `break` met immédiatement fin à la boucle la plus interne dans laquelle elle se trouve. Elle est très pratique par exemple dans les boucles qui recherchent un élément car elle permet d'interrompre la recherche dès que l'élément est trouvé.

L'instruction `continue` saute immédiatement à la fin du corps de la boucle la plus interne dans laquelle elle se trouve. Elle permet donc de "sauter un tour" de boucle en n'exécutant pas la fin du corps de la boucle pour cette itération. Elle est pratique par exemple lorsqu'on parcourt une structure de données et que certains éléments ne doivent pas subir le traitement effectué par la boucle.

Sauts généraux

L'instruction `goto` permet de sauter à une "étiquette", c'est-à-dire à une instruction à laquelle on a donné un nom. Comme le concept structurant en C est la fonction et qu'un `goto` est une instruction d'assez bas niveau, il n'est possible d'effectuer un `goto` que vers une étiquette se situant dans la même fonction.

En règle générale, il faut se passer des `goto` dans presque toutes les situations, mais il arrive qu'ils soient utiles principalement pour gérer les erreurs dans une fonction, ou pour quitter plusieurs boucles imbriquées d'un seul coup.

Du point de vue de la syntaxe, pour créer une étiquette, il faut écrire le nom de l'étiquette suivi de deux-points avant l'instruction qui doit porter ce nom ; et pour `goto`, la syntaxe est :
`goto <étiquette>`

Chapitre 6

Bibliothèque C standard

La bibliothèque C standard est spécifiée dans la même norme que le langage C. Elle est donc disponible avec tout compilateur de programmes utilisateur se réclamant de la norme. Il faut absolument en privilégier l'utilisation pour éviter de réécrire des fonctions déjà disponibles : cela fait gagner un peu de temps, et surtout permet d'avoir une plus grande confiance dans la qualité de l'implémentation de ces fonctions. Certaines fonctionnalités (par exemple la gestion des fichiers) ne sont disponibles de façon portable qu'au travers de la bibliothèque C standard.

Cette bibliothèque est très riche, et nous n'abordons ici que quelques fonctions qui reviennent fréquemment.

6.1 Entrées/sorties

Aucune primitive du langage C ne permet d'interagir avec l'utilisateur ou avec des fichiers. C'est la bibliothèque C qui fournit à l'environnement C ces fonctionnalités. Afin de les utiliser, il est nécessaire d'inclure au début du programme le fichier `stdio.h` avec une directive de la forme

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

6.1.1 Gestion de fichiers

Pour effectuer des opérations de lecture et/ou d'écriture dans un fichier, il faut suivre les étapes suivantes :

1. ouverture du fichier
2. réalisation des opérations d'entrée/sortie
3. fermeture du fichier

Cette sous-section va décrire les opérations d'ouverture et de fermeture des fichiers.

L'étape préalable à toute manipulation de fichier est de récupérer une référence au fichier (plus précisément en anglais un "handle") de type opaque `FILE *`. Cette référence sera ensuite passée à toutes les fonctions devant manipuler ce fichier. Chaque référence à un fichier contient un curseur qui dénote la position courante dans le fichier. Chaque lecture ou écriture se fera à cette position¹ et *fera avancer le curseur* d'autant d'octets qui auront été lus ou écrits. Ainsi l'appel répété aux fonctions de lecture/écriture permet de parcourir tout le fichier.

1. si un fichier est ouvert en mode "ajout", les écritures se feront à la fin du fichier indépendamment de la position du curseur

C'est la fonction `fopen()` qui permet d'obtenir cette référence, et on appelle cette action l'ouverture du fichier. C'est lors de cette étape où on fournit le nom du fichier que l'existence et la légalité de l'action demandée sont vérifiées. Aucune autre fonction ne nécessite de spécifier le nom du fichier.

```
FILE *fopen(const char *filename, const char *mode);
```

La fonction `fopen()` prend en paramètre le nom du fichier à ouvrir et le "mode" dans lequel le fichier sera ouvert. Elle retourne le type opaque "`FILE *`" qu'on ne peut que stocker et passer en paramètre aux différentes fonctions de la bibliothèque traitant des entrées/sorties.

Le mode d'ouverture précise quel sera l'utilisation faite du fichier, et il est spécifié par une chaîne de caractères, dont voici les principales possibilités.

"r"	(read) lecture seule, le fichier doit exister
"w"	(write) écriture seule, vide le fichier ou le crée
"a"	(append) écriture seule, crée le fichier si nécessaire, toutes les écritures se feront à la fin du fichier
"r+"	comme "r" mais les écritures sont aussi autorisées
"w+"	comme "w" mais les lectures sont aussi autorisées
"a+"	comme "a" mais les lectures sont aussi autorisées

La fonction `fopen()` retourne la référence particulière `NULL` en cas d'erreur, voir la section 6.4 p. 36.

Après les opérations souhaitées de lecture et d'écriture, il est nécessaire de fermer le fichier ouvert par `fopen()`. Ceci a plusieurs raisons : il faut libérer la référence au fichier (de type `FILE *`) et surtout, si des écritures ont été effectuées dans le fichier, la fermeture du fichier garantit que toutes les données finiront par être écrites à l'emplacement physique du fichier.

C'est la fonction `fclose()` qui effectue la fermeture d'un fichier. Après appel à `fclose()`, la référence de fichier qui lui est passée en paramètre n'est plus utilisable.

```
int fclose(FILE *flux);
```

Comme l'exécution de `fopen()` peut déclencher des opérations d'écriture sur un support physique, elle peut échouer. La fonction retourne 0 si tout s'est bien passé et la constante `EOF` en cas d'erreur, auquel cas la variable `errno` sera positionnée pour préciser le motif de l'erreur.

Un peu avancé – Vous remarquerez que parfois le mot flux est employé au lieu du mot fichier. En fait, les fichiers constituent un cas particulier et sont souvent des données stockées sur un périphérique de stockage. Mais l'interface de programmation de la bibliothèque C standard traite de la même manière que les fichiers les terminaux (une lecture lit au clavier, une écriture affiche les données) et d'autres types de fichiers spécifiques au système d'exploitation sur lequel le programme s'exécute.

Ainsi certains flux n'ont pas toutes les propriétés des fichiers et peuvent empêcher certaines fonctions de s'exécuter : par exemple il n'est pas possible de changer la position du curseur stocké dans la référence `FILE` d'un flux de type "terminal".

Enfin, les entrées standard, sortie standard, et sortie d'erreur standard sont des flux spéciaux qui sont déjà ouverts au démarrage du programme et dont les `FILE *` respectifs sont des variables globales nommées respectivement `stdin`, `stdout`, et `stderr`.

6.1.2 Ecriture et lecture de données formatées

La bibliothèque C offre des fonctions d'entrée/sortie d'assez haut niveau qui permettent d'écrire et de lire facilement les différents types de base du langage C (entiers, flottants, caractères, chaînes de caractères). Les types composés (structures, tableaux) doivent eux être affichés élément par élément car il serait difficile de communiquer à la bibliothèque la description du type composé pour qu'elle puisse effectuer l'affichage.

Ecriture

Les fonctions de la famille `printf()` permettent d'écrire des données dans un fichier sous forme de texte. La fonction `printf()` écrit sur la sortie standard du programme, la fonction `fprintf()` écrit dans le fichier passé en paramètre, et la fonction `sprintf()` écrit dans la chaîne de caractères passée en paramètre. Leur fonctionnement est par ailleurs identique et leurs prototypes sont donc très similaires :

```
int printf(const char *format, ...);
int fprintf(FILE *flux, const char *format, ...);
int sprintf(char *chaine, const char *format, ...);
```

Toutes ces fonctions retournent le nombre de caractères écrits, ou un nombre négatif pour indiquer qu'une erreur d'écriture a eu lieu. Si l'écriture concerne des données importantes (ce qui n'est pas forcément le cas de messages d'information), il conviendra de tester la valeur de retour de ces fonctions pour déclencher un traitement d'erreur approprié.

La chaîne `format` sera écrite, en effectuant potentiellement des remplacements. Chaque caractère pour-cent (%) suivi de caractères particuliers rencontré dans la chaîne signifie qu'il faut remplacer ce bout de chaîne par le paramètre correspondant dans la liste variable de paramètres donnée à la fonction. Avant de poursuivre, voici un exemple :

```
printf("L'âge de %s est %d\n", prenom, age);
```

exige que `prenom` soit une chaîne de caractères, et `age` soit d'un type compatible avec `int`, et produira un affichage remplaçant `%s` par le contenu de la chaîne `prenom` et `%d` par l'écriture textuelle de l'entier `age` en base dix. Par exemple :

```
L'âge de Toto est 253
```

Les fonctions de cette famille sont riches, et il est possible de formater les sorties de manière assez fine. Ici nous donnons seulement les utilisations basiques pour afficher les types de base du langage et pour plus de détails, nous renvoyons le lecteur à la norme du langage C ou à défaut à la documentation en ligne de son environnement de développement.

<code>%s</code>	chaîne de caractères (<code>const char *</code>)
<code>%c</code>	caractère de code ASCII donné (<code>char</code>)
<code>%d</code>	entier signé, affiché en base 10
<code>%u</code>	entier non signé, affiché en base 10
<code>%x</code>	entier non signé, affiché en base 16
<code>%g</code>	nombre flottant

Pour les affichages d'entiers, il faut donner un paramètre compatible avec le type `int` et pour afficher un entier d'une autre taille, on peut utiliser un modificateur qui est un ensemble de lettres entre le signe pour-cent et la lettre qui décrit l'affichage. Voici une liste de ces modificateurs :

hh	char ou unsigned char
h	short ou unsigned short
l	long ou unsigned long
ll	long long ou unsigned long long

On peut par exemple afficher un long int en base 10 avec %ld.

Lecture

Les fonctions de la famille de `printf()` ont leur équivalent en lecture, et leurs prototypes ne diffèrent de leur homologue d'écriture que par le nom.

```
int scanf(const char *format, ...);
int fscanf(FILE *flux, const char *format, ...);
int sscanf(char *chaine, const char *format, ...);
```

Cependant la valeur de retour est très différente : si une erreur de lecture se produit avant qu'un des paramètres ait été lu, la valeur EOF est retourné. Sinon le nombre de conversions effectuées est retourné. S'il est égal au nombre de % dans la chaîne de format, c'est que l'exécution de la fonction s'est correctement déroulée.

Une autre différence majeure est que les paramètres supplémentaires fournis pour chacune des demandes de conversion doit être un pointeur vers l'objet à remplir avec ce qui sera lu. En effet, comme le langage C fait les appels de fonction par valeur, il est nécessaire de passer l'adresse pour que la fonction ait la possibilité de modifier la valeur de l'objet fourni par l'appelant.

Enfin, dans le cas des nombres flottants, il devient nécessaire de différencier les `float` et les `double`. Ainsi %g lira un float et %lg un double.

Voici un exemple d'utilisation :

```
int age;

scanf("%d", &age);
if (age >= 18)
    printf("Majeur !\n");
```

Lira sur l'entrée standard (qui est par défaut liée au clavier du terminal) un entier et affichera Majeur ! si le nombre entré est supérieur à 18.

6.1.3 Ecriture et lecture de plus bas niveau

L'inconvénient des fonctions de la famille de `scanf()` est qu'elles vont ignorer des espaces, et qu'il est délicat de détecter proprement la fin de fichier. Il est possible et souvent finalement plus confortable d'accéder au fichier caractère par caractère. Ce sont les fonctions `fgetc()` et `fputc()` qui s'en chargent.

```
int fgetc(FILE *flux);
int fputc(int c, FILE *flux);
```

La fonction `fgetc()` lit le caractère suivant dans `flux` et le retourne, ou EOF en cas d'erreur de lecture. La fonction `fputc()` écrit le caractère `c` dans `flux`, et retourne EOF si elle échoue.

6.1.4 Position du curseur dans les fichiers

Plusieurs fonctions sont disponibles pour consulter et déplacer le curseur associé à un `FILE *`. La fonction `feof()` permet de tester si le curseur a atteint la fin du fichier, auquel cas plus aucune lecture n'est possible. Elle retourne donc 1 si la fin de fichier est atteinte et 0 sinon.

```
int feof(FILE *flux);
```

Il est aussi possible avec la fonction `ftell()` de connaître la position courante du curseur dans le flux. Elle retourne le nombre d'octets dans le fichier après lequel se trouve le curseur. Si la notion de curseur n'est pas définie pour ce type de flux, la fonction retourne la constante `-1L`.

```
long int ftell(FILE *flux);
```

La fonction `fseek()` permet d'effectuer l'action duale et positionne donc le curseur en fonction des paramètres passés à `fseek()` si le type de flux le permet (fichiers à accès direct).

Il est possible de spécifier la position comme un décalage en nombre d'octets par rapport à une des positions suivantes :

<code>SEEK_SET</code>	début du fichier
<code>SEEK_CUR</code>	position courante du curseur
<code>SEEK_END</code>	fin du fichier

```
int fseek(FILE *flux, long int decalage, int reference);
```

Le paramètre `reference` donne le point de référence par rapport auquel se fera le décalage et correspond à une des constantes `SEEK_SET`, `SEEK_CUR`, ou `SEEK_END`. Le `decalage` peut être positif ou négatif et il est donc aisé par exemple de reculer le curseur d'un nombre d'octets donné. Cette fonction retourne 0 si aucune erreur n'a été rencontrée.

6.2 Chaînes de caractères

Les chaînes de caractères n'ont dans le langage C aucun statut particulier : ce sont des tableaux de `char`. Seules les constantes de type chaîne ajoutent automatiquement le caractère `'\0'` à la fin de la chaîne. En revanche, les fonctions de la bibliothèque C standard qui manipulent des chaînes de caractères imposent cette convention.

Les fonctions suivantes permettent de faire des manipulations de base sur les chaînes de caractères : calcul de la longueur, copie, concaténation, comparaison. Elles sont définies dans le fichier d'en-tête `string.h`.

```
size_t strlen(const char *chaine);  
    retourne la longueur de la chaîne chaine  
char *strcpy(char *dest, const char *src);  
    copie la chaîne src dans le tableau dest  
char *strcat(char *dest, const char *src);  
    copie la chaîne src à la suite de la chaîne contenue dans dest  
int strcmp(const char *chaine1, const char *chaine2);  
    voir ci-dessous
```

La fonction de comparaison de chaînes de caractères `strcmp()` permet de situer les deux chaînes de caractères l'une par rapport à l'autre dans l'ordre lexicographique (ordre du dictionnaire). Ainsi elle retourne 0 si les chaînes sont égales, un nombre négatif si `chaine1` se trouve avant `chaine2` dans l'ordre lexicographique, et un nombre positif dans le cas contraire.

6.3 Fonctions utilitaires

Voici quelques fonctions qui peuvent s'avérer utiles dans des programmes.

6.3.1 Génération de nombres pseudo-aléatoires

Des appels successifs à la fonction `rand()` renvoient une séquence de nombres pseudo-aléatoire compris entre 0 et `RAND_MAX`. Cela signifie que si on ré-exécute le même programme, la séquence de valeurs sera la même.

Pour pouvoir diversifier les séquences retournées par `rand()`, il est possible d'introduire une graine par la fonction `srand()` qui va déterminer par sa valeur une nouvelle séquence pseudo-aléatoire. Il faut donc bien être conscient que même si on initialise la séquence avec une graine "aléatoire", la séquence n'a pas grand'chose d'aléatoire. Ces fonctions ne doivent pas être utilisées pour des applications cryptographiques sérieuses par exemple.

Elles sont en revanche très pratique pour tester des programmes qui ont besoin d'un peu d'aléatoire tout en facilitant le débogage puisque d'une exécution sur l'autre, le programme suivra exactement le même comportement.

```
#include <stdlib.h>

int rand(void);
void srand(unsigned int graine);
```

6.3.2 Conversion d'une chaîne de caractères en entier

La fonction `atoi()` ("ASCII vers entier") prend en paramètre une chaîne de caractères et retourne l'entier représenté par cette chaîne, interprété en base 10.

```
#include <stdlib.h>

int atoi(const char *chaine);
```

Notez que la fonction `atoi()` n'est pas très robuste : si la chaîne ne représente pas un nombre, elle pourra retourner par exemple 0 et le programmeur est dans l'incapacité de distinguer ce cas d'erreur du cas où la chaîne contenait une représentation textuelle de 0. Cette fonction est pratique pour expérimenter rapidement ou dans des environnements relâchés.

Nous signalons juste ici l'existence de la fonction `strtol()` qui pallie ce problème en utilisant en plus un pointeur qui indiquera à la fin de la fonction où la lecture s'est arrêtée. Si ce pointeur pointe vers le caractère `'\0'`, c'est que toute la chaîne a été lue et qu'elle contenait uniquement des chiffres. Les débordements sont aussi gérés sans ambiguïté par cette fonction.

6.4 Gestion des erreurs

Beaucoup des fonctions qu'on écrit lorsqu'on débute en programmation s'exécutent dans un environnement prédéterminé et entièrement contrôlé par le programmeur. Ce n'est plus le cas dès que le programme souhaite interagir avec son environnement pendant l'exécution, par exemple pour allouer de la mémoire, ou pour accéder à des fichiers.

Les fonctions interagissant avec des fichiers sont un très bon exemple de la multitude d'erreurs pouvant survenir. Par exemple, une tentative d'accès à un fichier lorsqu'on a des droits insuffisants, une tentative d'écriture sur un disque plein, une tentative de lecture sur une clé USB qui est retirée juste avant ou pendant, ...

Il est absolument nécessaire pour que le programme soit correct qu'il traite ces erreurs. La marche à suivre pour chaque erreur potentielle dépend complètement de l'application et dépasse le cadre de ce cours : la gestion des erreurs est une des tâches les plus ardues dans la conception et l'écriture de programmes. Il est néanmoins possible au moins d'afficher un message à l'utilisateur et d'éviter de continuer l'exécution du programme si les étapes qui ont subi une erreur sont cruciales pour la suite de l'exécution.

La bibliothèque C standard ne standardise malheureusement pas complètement la façon dont les fonctions rapportent une erreur. Chaque fonction précise quelle valeur de retour indique une exécution erronée, et il faut absolument vérifier que la fonction s'est correctement exécutée après chaque appel. En revanche, la façon de rapporter *quelle* erreur a eu lieu est standardisée. Il s'agit d'une constante particulière stockée dans la valeur-gauche `errno`. Ainsi même si la détection de l'erreur est spécifique à chaque fonction, l'analyse de sa cause peut être générique. Par exemple, l'erreur `ENOENT` signifie que le fichier n'existe pas, `EACCES` signifie que l'utilisateur n'a pas les droits d'accès au fichier ; beaucoup d'autres valeurs sont définies. Heureusement, la fonction `perror()` permet d'afficher une raison lisible de l'erreur contenue dans `errno`. Notez que le contenu de `errno` n'est pas spécifié si la fonction s'exécute correctement.

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

```
void perror(const char *chaine);
```

La fonction `perror()` affiche sur la sortie d'erreur standard la chaîne de caractères `chaine`, les caractères deux-points (`:`) et espace, puis le texte correspondant à l'erreur spécifiée par `errno`. Elle est donc très pratique pour afficher un message indiquant à l'utilisateur la raison pour laquelle le programme ne fonctionne pas.

Enfin, en fonction de la gravité de l'erreur, il peut être préférable de mettre fin au programme en utilisant la fonction `exit()` qui se trouve dans `stdlib.h`.

```
void exit(int valeur);
```

Cette fonction quitte le programme courant, comme si la valeur `valeur` avait été retournée par la fonction `main()` du programme.

6.5 Fonctions mathématiques

La bibliothèque C standard propose de nombreuses fonctions mathématiques, notamment trigonométriques, pour réaliser des calculs. Ces fonctions sont déclarées dans le fichier `math.h` et en voici quelques unes : `sqrt()` calcule la racine carrée, `pow()` permet d'élever un nombre flottant à une puissance quelconque, `cos()` calcule le cosinus, etc.

En ce qui concerne le type de ces fonctions, il est intéressant de noter que toutes ces fonctions travaillent sur des `double`. Cependant la plupart existent aussi travaillant sur des `float` (donc moins précis mais plus rapides sur certaines architectures) et leur nom est le même suivi d'un `f`, comme par exemple `sqrtf()`, `powf()`, etc.

Enfin, il convient de noter que l'implémentation de ces fonctions réside dans une bibliothèque mathématique qui se trouve physiquement à l'extérieur de la bibliothèque C. Il faut donc lors de l'étape d'édition de liens ajouter l'option `-lm` au compilateur C.

Voici deux exemples simples de lignes de compilation :

```
$ cc -o programme-simple programme-simple.c -lm
```

```
$ cc -o programme programme.o robot.o arbre.o grille.o utilitaires.o -lm
```


Index

- #define, 8
- #else, 9
- #endif, 9
- #if, 9
- #include, 8
- étiquette, 30
- _Bool, 15

- affectation, 10
- associativité, 10
- atoi(), 36

- bloc, 12
- _Bool, 15
- booléen, 15
- break, 28, 30

- cc, 5
- champ de structure, 18
- comparaison, 11
- compilation, 2
- conjonction, 11
- continue, 30
- corps de boucle, 29
- cos(), 37

- déclaration, 23
 - globale, 23
- définition, 23
- déréférencement, 20
- disjonction, 11
- durée de vie, 24

- effet de bord, 10
- en-tête, fichier d', 8
- errno, 37
- exit(), 37

- fclose(), 32
- feof(), 35
- fgetc(), 34
- fichier
 - d'en-tête, 8

- fonction, 19
 - prototype, 19
 - type, 19
- fopen(), 32
- fprintf(), 33
- fputc(), 34
- fscanf(), 34
- fseek(), 35
- ftell(), 35

- gcc, 6
- gestion des erreurs, 36
- goto, 30

- indice, 15
- instruction, 12
- interface, 24

- l-value, 10
- label, 30

- macro, 8
- mathématiques, fonctions, 37

- opérateur
 - arithmétique, 10
 - booléen, 11
 - d'affectation, 10
 - de comparaison, 11
 - sur les bits, 12
- options du compilateur, 6

- paramètre, 20
- passage par valeur, 20
- perror(), 37
- pointeur, 20
- portée, 23
- pow(), 37
- préprocesseur, 8
- prétraitement, 3, 8
- preprocessing, 3, 8
- printf(), 33
- priorité d'un opérateur, 10

- programmation
 - fonctionnelle, 1
 - impérative, 1
 - par contraintes, 1
- prototype, 19
- rand(), 36
- scanf(), 34
- sprintf(), 33
- sqrt(), 37
- srand(), 36
- sscanf(), 34
- stderr, 32
- stdin, 32
- stdout, 32
- strcat(), 35
- strcmp(), 35
- strcpy(), 35
- strlen(), 35
- structure
 - champ, 18
 - définition, 18
- style
 - fonctionnel, 1
 - impératif, 1
- syntaxe, généralités, 7
- tableau, 15
- typage
 - faible, 13
- type
 - fonction, 19
 - scalaire, 13
 - structure, 18
- valeur-g, 10
- visibilité, 23
 - fichier, 23
 - globale, 23
 - locale, 23