

CSC 3GIN1 TA

Programmation impérative (AKA « C »)

François Pessaux & Rodéric Moitié

L'école des grandes avancées

ENSTA | INSTITUT POLYTECHNIQUE DE PARIS

Table des matières

1 Un peu de contexte **9**
- 1.1 Un peu d'histoire . . . 9
- 1.2 Pourquoi apprendre C ? . . . 10
- 1.3 Exemple : Hello World . . . 10
- 1.4 Compiler, exécuter . . . 11
- 1.5 Comprendre le "Hello World" . . . 12
  - 1.5.1 Aspects généraux de la syntaxe . . . 12
  - 1.5.2 Les commentaires . . . 13
  - 1.5.3 La fonction `main` . . . 13
  - 1.5.4 `#include` . . . 14
- 1.6 En résumé . . . 16

2 Les types de base **17**
- 2.1 Types scalaires . . . 18
  - 2.1.1 Les entiers . . . 18
    - 2.1.1.1 Signe . . . 18
    - 2.1.1.2 Taille . . . 18
    - 2.1.1.3 Plage de représentation . . . 18
  - 2.1.2 Les réels . . . 19
  - 2.1.3 Les caractères . . . 19
  - 2.1.4 Les absents : les booléens . . . 20
- 2.2 En résumé . . . 21

3 Les définitions **23**
- 3.1 Définition de fonction . . . 23
- 3.2 Définition de variable . . . 24
  - 3.2.1 Variable globale, variable locale . . . 26
- 3.3 Définition de type scalaire énuméré (`enum`) . . . 28
  - 3.3.1 Des booléens . . . 29
- 3.4 En résumé . . . 30

4 Expressions et instructions **31**
- 4.1 Expressions . . . 31
  - 4.1.1 Constantes . . . 31
  - 4.1.2 Expressions arithmétiques . . . 32
  - 4.1.3 Expressions relationnelles . . . 33
  - 4.1.4 Expressions logiques . . . 34
  - 4.1.5 Appel de fonction . . . 34
- 4.2 Instructions (hors contrôle) . . . 34
  - 4.2.1 Affectation . . . 35
  - 4.2.2 `return` . . . 36
  - 4.2.3 Affectation-arithmétique abrégée . . . 36
  - 4.2.4 Bloc . . . 37
    - 4.2.4.1 Regrouper des instructions . . . 37
    - 4.2.4.2 Définir des variables locales . . . 38

4.2.5 Et les appels de fonction ? . . . 39
4.3 En résumé . . . 40

5 Conditionnelles et boucles **41**
5.1 Conditionnelles . . . 41
5.1.1 `if else` . . . 41
5.1.2 Le `elseif/elif` n'existe pas . . . 42
5.1.3 `switch case` . . . 43
5.2 Boucles . . . 45
5.2.1 Boucle `while` . . . 45
5.2.2 Boucle `for` . . . 45
5.2.3 Boucle `do while` . . . 47
5.2.4 Récapitulatif des formes de boucles . . . 48
5.3 Ruptures de séquence . . . 48
5.3.1 `break` . . . 48
5.3.2 `continue` . . . 49
5.4 En résumé . . . 51

6 Affichage avec `printf` **53**
6.1 Messages constants . . . 53
6.2 Messages formatés . . . 54
6.3 En résumé . . . 56

7 Pointeurs **57**
7.1 Adresse . . . 57
7.2 Pointeur . . . 58
7.2.1 Déclaration de pointeur . . . 58
7.2.2 Initialisation de pointeur . . . 59
7.2.3 Déréférencement de pointeur . . . 60
7.2.3.1 Lecture à une adresse . . . 60
7.2.3.2 Écriture à une adresse . . . 62
7.3 En résumé . . . 64

8 Appel de fonction et passage d'arguments **65**
8.1 Pile d'exécution . . . 65
8.2 Passage d'argument par adresse . . . 68
8.2.1 Le problème . . . 68
8.2.2 Paramètre en sortie . . . 69
8.2.3 Paramètre en entrée/sortie . . . 69
8.2.4 Copie versus adresse ? . . . 70
8.2.5 Ménager la pile (1) . . . 70
8.2.6 Et pour les tableaux ? . . . 71
8.3 En résumé . . . 72

9 Tableaux statiques **73**
9.1 Définition de tableau . . . 73
9.2 Accès, indexation . . . 74
9.2.1 Arithmétique sur pointeur . . . 75
9.3 Tableaux statiques à plusieurs dimensions . . . 76
9.4 Retourner un tableau . . . 77
9.5 En résumé . . . 78

10 Chaînes de caractères **79**
10.1 Définition . . . 79
10.1.1 Chaînes non modifiables . . . 80
10.2 Opérations sur les chaînes . . . 80
10.2.1 Affichage . . . 80

10.2.2 Modification des caractères . . . 81
10.2.3 ~~Affectation~~ . . . 82
10.2.4 Longueur d'une chaîne . . . 83
10.2.5 Copie de chaîne . . . 83
10.2.6 Comparaison de chaînes . . . 85
10.2.7 Concaténation de chaînes . . . 86
10.2.8 Recherche d'une sous-chaîne . . . 87
10.3 En résumé . . . 88

11 Arguments de la ligne de commande **89**
11.1 Récupération des arguments . . . 89
11.2 Utilisation des arguments . . . 91
11.3 Meilleure utilisation des arguments . . . 92
11.3.1 Conversion vers entiers . . . 92
11.3.2 Conversion vers réels . . . 95
11.4 En résumé . . . 96

12 Allocation dynamique **97**
12.1 Motivations . . . 97
12.1.1 Taille trop figée . . . 97
12.1.2 Retourner un tableau . . . 97
12.2 Allocation dans le tas . . . 98
12.2.1 Allocation : `malloc` . . . 98
12.2.2 Libération : `free` . . . 100
12.3 Quelques remarques . . . 101
12.3.1 Ménager la pile (2) . . . 101
12.3.2 Allocations dynamiques partout ? . . . 102
12.3.3 Ne pas faire `free` . . . 102
12.4 En résumé . . . 103

13 Entrée au clavier : `scanf` **105**
13.1 Interlude : redirection de l'entrée standard . . . 105
13.2 Les bases de `scanf` . . . 106
13.3 Lecture d'un caractère . . . 108
13.4 Lecture d'une chaîne de caractères . . . 108
13.5 Détecter les erreurs de lecture . . . 110
13.6 Formats complexes . . . 111
13.7 En résumé . . . 112

14 Structures **113**
14.1 Définition de type structure . . . 113
14.2 Définition de variable de type structure . . . 114
14.3 Accès dans une `struct` . . . 115
14.3.1 Accès via un pointeur . . . 116
14.4 Interaction avec les fonctions . . . 117
14.4.1 Des `struct` en arguments . . . 117
14.4.2 Des `struct` en résultat . . . 117
14.4.3 Structures pour des champs de même type . . . 119
14.5 En résumé . . . 120

15 Le préprocesseur `cpp` **121**
15.1 Fichiers d'en-tête et `#include` . . . 121
15.1.1 `#include < >` . . . 122
15.1.2 `#include " "` . . . 122
15.1.3 Fichiers `.h` . . . 122
15.2 Macros, constantes symboliques et `#define` . . . 124
15.2.1 Macros constantes . . . 125

15.2.2 Macros paramétrées . . . 126
15.2.3 Fonctions ou macros paramétrées ? . . . 127
15.3 En résumé . . . 129

16 Considérations plus avancées 131
16.1 Alias de type . . . 131
16.1.1 Abréger les types . . . 132
16.1.2 Documenter les types . . . 132
16.2 Type `union` . . . 133
16.2.1 Initialisation . . . 135
16.2.2 Utilisation incohérente . . . 135
16.3 Qualificateur `const` . . . 136
16.3.1 Pointeur `const`ant . . . 137
16.3.2 Pointeur sur `const`ant . . . 137
16.3.3 Pointeur `const`ant sur `const`ant . . . 138
16.3.4 Arguments pointeurs et `const` . . . 139
16.3.5 Extraction d'adresse sur variable `const` . . . 139
16.4 Qualificateur `volatile` . . . 140
16.5 Tableaux à longueur variable (VLA) . . . 141
16.5.1 Ne pas utiliser de VLA : raison 1 . . . 141
16.5.2 Ne pas utiliser de VLA : raison 2 . . . 142
16.5.3 Conclusion à propos des VLA . . . 142
16.6 Débordements, arrondis . . . 142
16.6.1 Problèmes d'entiers . . . 143
16.6.1.1 Débordement . . . 143
16.6.1.2 Mélange signé / non signé . . . 144
16.6.1.3 Division entière . . . 145
16.6.2 Problèmes de réels . . . 146
16.6.2.1 Conversion vers entier . . . 146
16.6.2.2 Problème de précision . . . 146
16.6.2.2.1 Flottants différents égaux . . . 147
16.6.2.2.2 Flottants égaux différents . . . 147
16.6.2.2.3 Mauvais conditionnement . . . 149
16.7 Expression conditionnelle (`?`) . . . 149
16.8 Séquence d'expressions (`,`) . . . 150
16.9 L'instruction `goto` . . . 152
16.9.1 Simuler des exceptions . . . 153
16.10 Opérateurs bit-à-bit . . . 155
16.10.1 Opérateurs logiques . . . 155
16.10.2 Décalages de bits . . . 156
16.11 Transtypage . . . 156
16.11.1 Utilité du transtypage . . . 158
16.12 Variable locale `static` . . . 159
16.13 Variable globale `static` . . . 160
16.14 Déclaration de variable (`extern`) . . . 160
16.15 Inclusion conditionnelle . . . 160

17 Compilation séparée 163
17.1 Motivations . . . 163
17.2 `make` . . . 164
17.3 Structure d'un `Makefile` . . . 164
17.4 Structure d'une règle simple . . . 165
17.5 Interlude : compiler manuellement plusieurs fichiers C . . . 165
17.6 Exemple de `Makefile` simple pour du C . . . 167
17.7 Importance des dépendances . . . 167
17.8 Incohérences de dépendances . . . 168

17.9 Variables . . . 169
17.10 Variables automatiques . . . 170
17.11 Métacaractères (*wilcards*) dans les règles . . . 171
17.12 Cibles `PHONY` . . . 171
17.13 Substitution dans les variables . . . 173
17.14 Règles implicites (*pattern rules*) . . . 173
17.15 Appel à `make` et (quelques) options . . . 175

18 Annexes **177**
18.1 Mots-clés (C ANSI) . . . 177
18.2 Priorité, associativité . . . 178

Préambule

Ce document est le support accompagnant le cours CSC_3GIN1_TA « Programmation impérative » dispensé à l'ENSTA dans le cadre de la première année du cursus ingénieur. Il s'adresse à un public d'élèves ingénieurs généralistes ayant déjà rencontré l'informatique dans leur cursus antérieur, sans pour autant avoir approfondi le domaine. Ce document se concentre sur l'apprentissage du langage C, support des cours d'informatique en première année. Il est volontairement technique, sans considérations d'ingénierie logicielle ni de conception d'algorithmes.

Chaque chapitre est terminé par une section « En résumé », volontairement (très) synthétique pouvant servir de référence rapide en cas d'urgence.

Ce document se veut plus approfondi que les capsules vidéo qui l'accompagnent. Certaines notions explicitement marquées « pour aller plus loin », ainsi que le chapitre d'approfondissement 16 sont présentés avec le souci de donner une présentation assez complète du langage, même si cette présentation ne prétend pas à l'exhaustivité.

Si vous détectez des erreurs, des typos, je vous serai très reconnaissant de m'en faire part afin de pouvoir rendre ce document de meilleure facture pour les générations suivantes. Bonne lecture...



Chapitre 1

Un peu de contexte

1.1 Un peu d'histoire

Le langage C a été créé aux Bell Laboratories en 1972 par Ken Thompson et Dennis Richie auxquels s'est joint Brian Kernigan un peu plus tard. Il est issu de plusieurs langages antérieurs, en particulier de BCPL et de B. La principale motivation pour la conception de ce langage était d'obtenir un outil de développement plus haut niveau pour écrire des programmes et en particulier pour adapter le code d'Unix (un ancêtre de Linux en quelque sorte) d'une machine PDP-7 vers une machine PDP-11.



FIGURE 1.1 – Ken Thompson, Dennis Richie, Brian Kernigan

En effet, sur PDP-7, Unix était écrit en assembleur, donc en utilisant directement les instructions du micro-processeur et en tenant manuellement compte des contraintes matérielles de celui-ci : par exemple, des mots machine de 18 bits (oui, pas 16 et encore moins 32). Le PDP-11, outre le fait qu'il bénéficiait de beaucoup plus de mémoire (24 Ko contre 8 Ko pour le PDP-7 !), avait des mots machine de 16 bits et des instructions différentes. De ce fait, pour obtenir une version d'Unix sur cette nouvelle machine, il fallait réécrire tout le programme dans ce nouvel assembleur (tout aussi abscons).

Il était urgent de disposer d'un langage qui puisse automatiquement et mécaniquement être traduit vers l'assembleur d'une machine quelconque. Cette traduction est le processus de compilation et le compilateur est le programme qui permet de traduire ... des programmes vers l'assembleur d'une machine cible. Ainsi, on écrit un programme dans le langage commun à toutes les machines qui disposent d'un compilateur pour ce langage et c'est le compilateur qui fait le travail de génération du code spécialisé pour chaque machine.

Bien entendu, les langages de programmation, quels qu'ils furent et sont, présentent en outre des constructions, des instructions infiniment plus concises et plus lisibles que leurs équivalents en assembleur.

Le langage C a ensuite beaucoup évolué, à été normalisé à plusieurs reprises pour le standardiser et garantir que tous les compilateurs C donnent la même signification aux constructions du langage, donc

aux programmes. Ces normes portent les doux surnoms de C89, C99, C11, C18, C23 et encadrent les nouveautés et changements apportés au langage au cours des années.

De nos jours, tous les systèmes d'exploitation disposent d'un compilateur C, Linux, macOS, Android pour ne citer que les plus connus. Depuis 1973, le C est le langage d'implantation des Unix, le langage Rust commençant depuis peu ($\sim$ 2023) à faire des apparitions dans le code source de Linux à côté de l'immense base de code C (plus de 30 millions de lignes de code estimées en 2021).

1.2 Pourquoi apprendre C ?

Le choix du langage pour les cours de première année à l'ENSTA s'est porté sur le langage C, pour les raisons que nous allons voir.

C est un langage relativement pauvre, sans constructions que l'on qualifiera de « magiques », dans lequel, en particulier, la gestion de la mémoire est à la charge du programmeur. On pourrait en quelque sorte qualifier C « d'assembleur de haut niveau ». Mais pourquoi donc commencer par un langage aussi rustique qui ne nous facilite pas vraiment la tâche ?

La raison principale a été énoncée il y a quelques instants : il ne comporte pas de constructions « magiques » dont le coût, la complexité algorithmique sont invisibles. Savoir écrire, en langage C, un algorithme que l'on a imaginé assure de savoir l'écrire dans à peu près n'importe quel autre langage de programmation. De plus, se restreindre aux constructions proposées par C, qui sont en fait disponibles dans n'importe quel autre langage de programmation, ça permet de se prémunir contre l'abominable tentation d'imaginer un algorithme à partir des constructions et des fonctions déjà existantes d'un langage de programmation très riche.

Une fois les bases de l'algorithmique et de la programmation acquises, le coût des opérations compris, on pourra bien entendu s'autoriser d'autres langages de plus haut niveau lorsqu'ils seront plus pertinents selon le cas d'utilisation. Mais ce sera en toute connaissance de cause et de coût et non par dépit, par défaut ou par ignorance. On pourra alors juger de la pertinence d'utiliser tel ou tel langage selon une évalation du coût de développement versus le coût d'utilisation et des contraintes du projet (doit-il être temps réel, en mémoire contrainte, etc).

En attendant, malgré son grand âge, C reste un langage largement utilisé dans l'industrie et disponible sur toutes machines et tous les systèmes d'exploitation, des puces embarquées à 5 centimes aux gros calculateurs à plusieurs millions d'euros. Il est polyvalent et permet soit une programmation qualifiée de « bas-niveau » où il est important de contrôler finement le matériel, soit une programmation plus haut-niveau où les détails matériels de la machine restent plus abstraits.

Soyons toutefois honnête, il laisse quand même parfois apparaître des détails techniques inhérents à la structure d'un ordinateur (taille des mots machine, notion de pointeurs et d'adresses, protocole d'appel des fonctions avec passage par copie, etc). Néanmoins, c'est le prix à payer pour cette polyvalence et surtout l'efficacité dont il fait preuve à l'exécution en terme de consommation de ressources. Des travaux de recherche affirment qu'un programme écrit en C consomme 75 fois moins d'énergie que le même écrit en Python.

Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, João Saraiva
Ranking programming languages by energy efficiency,
Science of Computer Programming, Volume 205,2021, ISSN 0167-6423
https://doi.org/10.1016/j.scico.2021.102609
https://www.sciencedirect.com/science/article/pii/S0167642321000022

1.3 Exemple : Hello World

Comme il est de coutume, le premier programme exhibé dans n'importe quel langage de programmation est le "Hello World", petit programme introduit par Dennis Ritchie et Brian Kernigan dans leur livre " The C Programming Language" en 1978. Voici donc (en exclusivité bien entendu) le fameux programme "Hello World" en C :

```c
#include <stdio.h>
int main () {
  printf ("Hello World !!\n");
  return 0 ;
}
```

1.4 Compiler, exécuter

Certains langages comme OCaml ou Python permettent une utilisation interactive : on entre les instructions au fur et à mesure dans ce que l'on appelle *l'interpréteur* et les résultats ou les effets s'affichent immédiatement. Une autre possibilité est d'écrire l'intégralité du programme dans un éditeur de textes, de le sauver dans un fichier pour l'exécuter après.

Un programme écrit en langage C ne peut pas être exécuté au fil de l'eau. Il nécessite d'abord une étape, dite de *compilation*, avant de pouvoir être exécuté. Cette étape de compilation permet de passer du texte source d'un programme à un fichier binaire contenant des instructions connues du micro-processeur. C'est ce fichier *binaire* qui pourra, ensuite, être exécuté.

La compilation est réalisée par un programme nommé *compilateur*. Le compilateur C que nous utilisons dans ce cours est le "GNU C Compiler", qui est invoqué dans le terminal par la commande `gcc`. Pour transformer un fichier source `hello.c` en un exécutable `hello.x`, il faut invoquer dans le terminal :

```
gcc hello.c -Wall -o hello.x
```

Si la compilation s'effectue sans problème, `gcc` reste totalement silencieux et produit le fichier exécutable. Cet exécutable peut ensuite être lancé en invoquant dans le terminal son nom précédé de `./` :

```
$ ./hello.x
Hello World !!
$
```

A contrario, si une ou plusieurs erreurs sont détectées, `gcc` les signalera par des messages d'erreur et l'exécutable ne sera pas créé. Ces messages d'erreur indiquent la ligne du programme où le problème a été détecté et la nature de ce dernier. Donc, lire au moins la ligne est une bonne idée...

```c
#include <stdio.h>
int main () { @  /* @ n'existe pas en C. */
  printf ("Hello World !!\n")   /* Et le ; alors ? */
  return 0 ;
}
```

```
$ gcc error.c
error.c: In function 'main':
error.c:2:15: error: stray '@' in program
 int main () { @  /* @ n'existe pas en C. */
               ^
error.c:4:3: error: expected ';' before 'return'
   return 0 ;
   ^~~~~~
```

L'option `-o` suivie d'un nom de (futur) fichier permet de spécifier le nom à donner à l'exécutable. Sans cette option, ce dernier est automatiquement nommé `a.out` et pourra donc être lancé par `./a.out` dans le terminal.

Important

L'option `-o` doit être suivie du nom que l'on veut donner à l'exécutable et surtout pas du nom du fichier source. Exécuter

```
gcc hello.c -o hello.c
```

aura pour irrémédiable conséquence de générer l'exécutable en écrasant le fichier source. Vous aurez bien un exécutable, oui, mais plus de fichier source.

L'option `-Wall` permet d'activer la recherche approfondie de code suspicieux susceptible de mener à des erreurs d'exécution. Des *avertissements* peuvent alors être émis mais ils n'empêchent pas la création de l'exécutable. Néanmoins, ces avertissements indiquent toujours des problèmes latents qui peuvent entraîner des erreurs d'exécution. La plupart du temps le programme fonctionnera, parfois, il plantera.

Important

C'est pourquoi il est indispensable d'utiliser l'option `-Wall` afin de forcer la recherche de ces erreurs en puissance. Les avertissements doivent donc être corrigés au même titre que les erreurs avant de lancer le programme.

L'option `-Werror` a été conçue afin de ne pas succomber à la tentation (souvent forte et toujours funeste) d'ignorer ces avertissements. Elle permet de transformer les avertissements en erreurs, donc d'empêcher la création de l'exécutable. Il va sans dire qu'elle est totalement recommandée.

Très souvent la première erreur en déclenche d'autres en cascade, résultant en des pages d'erreurs dans lesquelles il faut remonter pour trouver la première erreur. L'option `-Wfatal-errors` permet d'arrêter la compilation dès la première erreur. De toute façon, puisque c'est la première, il faudra bien commencer par corriger celle-là.

En résumé, la ligne de commande préconisée pour compiler notre fichiers source, donc tout ficher source est :

```
gcc -Wall -Werror -Wfatal-errors hello.c -o hello.x
```

1.5 Comprendre le "Hello World"

Ce programme est l'occasion de faire une présentation superficielle et rapide de la syntaxe du langage C. Les chapitres suivants reviendront sur ces points plus en détail et aborderont de manière plus exhaustive les constructions du langage.

1.5.1 Aspects généraux de la syntaxe

C est un langage *impératif* : ses programmes sont constitués de définitions de fonctions et de variables. Les fonctions sont composées d'instructions suivies d'un point-virgule (i.e. `;`).

Contrairement à Python, l'indentation, donc le décalage horizontal du code en fonction de l'imbrication des constructions, n'a aucun impact sur la signification d'un programme. Espaces, tabulations, retours à la ligne sont totalement ignorés. Dit autrement, "Hello World" se comportera parfaitement s'il est écrit sous la forme :

```
#include <stdio.h>
      int    main(  ) {printf("Hello World !!\n"    );  return 0;}
```

Néanmoins, l'indentation est un pré-requis crucial pour la lisibilité, la compréhension des programmes par les êtres humains. Il est donc primordial que vous indentiez vos programmes correctement. Ne pas

le faire est d'autant plus impardonnable que tous les éditeurs de textes modernes disposent d'une indentation automatique du code C, généralement déclenchée par l'appui sur la touche Tab lors de la rédaction du code.

1.5.2 Les commentaires

Les commentaires sont également totalement ignorés lors de la compilation (et *a fortiori* l'exécution) des programmes. Néanmoins ils sont eux aussi indispensables à la compréhension, la maintenabilité des programmes. Deux formes de commentaires existent.

En premier lieu, ceux qui s'étendent sur plusieurs lignes, débutés par la séquence `/*` et terminés par `*/`. Outre leur intérêt évident en terme de documentation du code source, ils peuvent servir à inhiber temporairement un morceau de code. Pourquoi temporairement ? Simplement car il n'est pas une bonne pratique de laisser du code inactif : lors d'une relecture, une tierce personne peut se demander si c'est un oubli, si ce morceau de programme devrait en fait être activé.

En second lieu, lorsque le texte du commentaire tient sur une seule ligne, il peut être débuté par la séquence `//` et il se terminera implicitement à la fin de la ligne.

```
/* Ceci est un commentaire...
     ... sur plusieurs lignes ...
          ... et il est vraiment très beau.          */

// Ce commentaire, lui, il est sur une seule ligne, mais très beau aussi.
```

Les commentaires sont indispensables pour plusieurs personnes. Tout d'abord, pour les personnes avec qui vous développerez, maintiendrez, partagerez le code. Vous savez peut-être exactement ce que vous écrivez, mais les autres personnes ne sont pas dans votre esprit. Elles auront besoin de comprendre le code de manière efficace, sans devoir refaire tout le cheminement de pensée qui vous a permis de l'écrire.

Les commentaires sont également indispensables **pour vous**. Vous savez peut-être exactement ce que vous écrivez ... aujourd'hui, maintenant, là, car vous êtes en plein dans l'action et vous y avez réfléchi pendant des heures. Mais dans une semaine, un mois, un an, vous ne serez plus dans le contexte. Vous vous retrouverez alors à devoir refaire le chemin de pensée, en vous disant « mais comment ça marchait, pourquoi j'avais fait ça, qu'est-ce que ça fait déjà ? ». Que ce soit pour modifier votre programme, le corriger ou tenter d'en réutiliser des parties.

Pour terminer, il existe des outils qui permettent de générer automatiquement la documentation d'un code source, mais ils le font à partir des commentaires qui annotent ce code. Ces documentations sont indispensables lorsque l'on développe des composants logiciels réutilisables (« bibliothèques ») qui seront mis à la disposition d'autres développeurs. Le monde du logiciel libre regorge de tels développements qui contribuent massivement à tous les logiciels dont la planète bénéficie gratuitement. Mais il existe aussi des bibliothèques logicielles commerciales, dont le code source n'est pas accessible : seule la documentation l'est, donc elle est absolument incontournable.

1.5.3 La fonction `main`

Une fonction est définie en énonçant le type de la valeur qu'elle retourne, son nom, ses arguments et enfin son corps constitué d'une suite d'instructions entre accolades (i.e. `{ }`). Ici, la fonction nommée `main` appelle majoritairement la fonction `printf` dont le but est d'afficher un message (cf. chapitre 6 pour plus de détails).

La fonction `main` a un statut particulier en C. Tout programme doit comporter une unique fonction nommé `main`. C'est par cette fonction que l'exécution du programme commencera. Si un programme ne comporte pas cette fonction, alors la compilation échouera.

```c
#include <stdio.h>
int mafonction () {
  printf ("Hello World !!\n");
  return 0 ;
}
```

```
$ gcc missing.c
/usr/lib/gcc/x86_64-linux-gnu/7/../../../x86_64-linux-gnu/Scrt1.o : Dans la fonction \_start:
(.text+0x20) : référence indéfinie vers «˜main˜»
collect2: error: ld returned 1 exit status
```

Dans le cas où un programme est constitué de plusieurs fichiers sources, ce qui arrive dans l'immense majorité des cas, un et un seul de ces fichiers devra avoir défini la fonction `main`.

La fonction `main` retourne une valeur entière. Cette valeur est appelée *code de retour* et sert à signaler à celui qui a lancé le programme, si ce dernier considère qu'il s'est terminé correctement ou bien s'il a rencontré une erreur. Par convention :

- 0 signifie « terminé sans erreur à signaler »,
- $+0$ signifie « terminé mais avec une erreur à signaler ».

Très souvent (et ce sera le cas durant ce cours) le programme est lancé depuis le *terminal* au moyen de *l'interprète de commandes* (aussi appelé un *shell* en anglais), `bash`, `sh`, `zsh` ou bien d'autres encore. Dans le shell, il est possible de récupérer le code de retour de la dernière commande utilisée (donc programme exécuté). Cela permet d'enchaîner plusieurs commandes sous forme d'un *script*, en s'assurant que les précédentes ont été exécutées avec succès.

On imagine facilement un scénario où notre programme permet de chiffrer un fichier, il est utilisé dans un script qui supprime ensuite le fichier initial pour ne laisser subsister que la version désormais « protégée par le secret ». Avant de supprimer irrémédiablement le fichier initial, mieux vaut s'assurer que le chiffrement a fonctionné, sinon on a tout perdu.

1.5.4 `#include`

Fréquemment, un fichier source fait référence à des fonctions « externes », c'est-à-dire dont la définition n'est pas dans le fichier. C'est le cas de la fonction `printf` qui appartient à la *bibliothèque standard* de C, une collection de nombreuses fonctions déjà écrites pour des traitements que l'on qualifiera approximativement « d'habituels » ou « de base ».

Pour que ces fonctions soient connues lorsque le fichier source est compilé, il faut importer leurs déclarations qui sont contenues dans des fichiers particuliers, d'extension `.h`, appelés *fichiers d'en-tête*. Ceci se fait avec la directive `#include` suivie du nom de fichier d'en-tête. C'est en quelque sorte le `import` de Python. Nous reviendrons plus en détail sur le fonctionnement de la directive `#include` dans le chapitre 15. Disons simplement pour le moment que l'on sait que pour trouver `printf`, il faut importer `<stdio.h>`.

Important

Pour savoir quel fichier d'en-tête inclure pour avoir accès à une fonction de la bibliothèque standard de C, il suffit de consulter sa page de manuel en section 3. Cela se fait en invoquant dans le terminal la commande `man` suivi de `3` puis du nom de la fonction recherchée.

```
man 3 cos
```

affiche la documentation de la fonction `cos`, avec le type de son argument et son type de retour. Nous y voyons également, dans la partie « SYNOPSIS », qu'elle requiert l'inclusion du fichier d'entête `<math.h>`.

→

```
cos(3)                Manuel des fonctions de la bibliothèque                cos(3)

NOM
       cos, cosf, cosl - Fonction cosinus

BIBLIOTHÈQUE
       Bibliothèque de math (libm, -lm)

SYNOPSIS
       #include <math.h>

       double cos(double x);
...
```

Il est possible de regarder le contenu du fichier `stdio.h` qui se trouve dans `/usr/include/stdio.h` sur un système Linux. Pour voir que l'utilisation de cette directive est pas inutile, supprimons ce `#include` de notre programme et compilons.

```
$ gcc missing.c
missing.c: In function 'main':
missing.c:2:3: warning: implicit declaration of function 'printf'
 [-Wimplicit-function-declaration]
   printf ("Hello World !!\n") ;
   ^~~~~~
missing.c:2:3: warning: incompatible implicit declaration of built-in function printf'
missing.c:2:3: note: include '<stdio.h>' or provide a declaration of 'printf'
```

Un message d'avertissement indique que la fonction `printf` n'est pas déclarée explicitement et que l'importation du fichier d'en-tête `<stdio.h>` pourrait y remédier.

Remarque : à partir de la version 14 de `gcc` l'avertissement est remplacé par une erreur.

```
$ gcc-14 -Wall missing.c
missing.c: In function 'main':
missing.c:2:3: error: implicit declaration of function 'printf'
 [-Wimplicit-function-declaration]
    2 |   printf("Hello\n");
      |   ^~~~~~
missing.c:1:1: note: include '<stdio.h>' or provide a declaration of 'printf'
  +++ |+#include <stdio.h>
    1 | int main() {
missing.c:2:3: warning: incompatible implicit declaration of built-in function 'printf'
 [-Wbuiltin-declaration-mismatch]
    2 |   printf("Hello\n");
      |   ^~~~~~
missing.c:2:3: note: include '<stdio.h>' or provide a declaration of 'printf'
```

1.6 En résumé

- Un programme C s'écrit sous forme d'un texte source. On utilise donc un éditeur de textes.
- Une fois écrit, il faut transformer ce texte en exécutable. On utilise un compilateur C.
- Le compilateur C s'appelle `gcc` et s'invoque dans un terminal.
- L'option `-o` permet de spécifier le nom de l'exécutable à créer.

```
gcc hello.c -o hello.x
```

- Sans option `-o`, le fichier exécutable se nomme `a.out`.
- On lance l'exécutable dans le terminal en invoquant son nom précédé de « `./` ». Par exemple `./hello.x`
- L'option `-Wall` permet d'activer les avertissements, qui représentent toujours des erreurs en puissance. Elle est indispensable.
- L'option `-Werror` permet de transformer les avertissements en erreurs, donc d'empêcher la création d'un exécutable manifestement incorrect.
- L'option `-Wfatal-errors` permet d'interrompre la compilation dès la première erreur.
- Un programme en C a toujours une unique fonction principale `main`. C'est par ce point que commence l'exécution d'un programme.
- Cette fonction commence par `int main ()`. Nous verrons plus tard qu'il existe une variante.
- Son corps est constitué de tout ce qu'il y a entre les accolades.
- Chaque instruction est terminée par un point-virgule (sauf pour les blocs dont il sera question dans la section 4.2.4).
- Elle doit toujours se terminer par le retour d'une valeur entière : `return ...`.
- Par convention, `main` retourne 0 pour signifier « pas d'erreur », $\neq 0$ pour signifier « erreur survenue ».

Chapitre 2

Les types de base

Contrairement à d'autres langages, C est un langage *explicitement typé*. Toute variable doit être explicitement déclarée en indiquant son *type*. Cela signifie que l'information de type est connue et surtout vérifiée au moment de la compilation. Dans d'autres langages comme Python ou JavaScript, les types sont vérifiés lors de l'exécution. Le choix de la vérification statique des types (donc à la compilation) permet de garantir qu'un certain nombre d'erreurs ne se produiront pas lors de l'exécution, ce qui est plutôt une bonne nouvelle.

```
float pi = 3.14 ;
int eight = 8 ;
long int big = 2409476653055 ;
struct point_t p = { 0, 0, 0 } ;
int t[4] = { 3, 2, 1, 0 } ;
```

Le type d'une variable indique quelle est la nature des valeurs pouvant résider dans cette variable et sert à plusieurs choses.

Il permet en premier lieu de définir quelles sont les opérations légales sur les variables. Par exemple, lorsque nous écrivons `3 + 4`, le compilateur en déduit que cette opération est légale et qu'il faut utiliser l'opération du micro-processeur qui réalise l'addition de deux entiers.

Si nous écrivons plutôt `3.14 + 4.0`, il en déduira qu'il s'agit de l'addition entre deux valeurs réelles, ce qui ne convoquera pas la même instruction du micro-processeur.

Si maintenant nous écrivons `3.14 + 4`, les choses sont moins évidentes. Il y a trois choix possibles puisque l'on mélange un réel et un entier dans cette opération, *a priori* d'addition. Tout d'abord, le compilateur peut choisir de convertir le réel en entier puis d'appliquer l'addition entre entiers. Ce n'est pas ce qu'impose la norme du langage C. Cette dernière oblige le compilateur à convertir l'entier en réel pour utiliser l'addition entre réels. Une troisième solution est possible, bien plus brutale : rejeter cette opération. L'idée n'est pas si saugrenue et est appliquée dans d'autres langages : plutôt que laisser le compilateur faire un choix arbitraire (que le néophyte ne connaît pas forcément), on demande alors au programmeur (ou à la programmeuse) de rendre explicite la conversion qu'il veut appliquer. C'est d'ailleurs toujours une excellente solution que de privilégier l'explicite à l'implicite : il ne faut pas chercher à faire le bonheur des gens malgré eux.

Pour terminer, quelle sémantique (signification) donner à `"3" + 4` ou `"3" + "4"` ? En C ces expressions sont rejetées (enfin, dans certaines circonstances la première peut être acceptée mais avec un comportement très contre-intuitif que nous tairons ici). Certains langages ont fait d'autres choix, en faisant des conversions qui leur sont propres.

En second lieu, les types permettent au compilateur de déterminer statiquement, c'est-à-dire au moment de la compilation, la taille des zones mémoire à allouer et à manipuler au cours des calculs.

2.1 Types scalaires

Les types scalaires font référence aux types de données atomiques, non composées. Ces types permettent en fait de représenter des nombres. Ils sont au nombre de trois : les entiers, les réels et les caractères.

En particulier, les chaînes de caractères n'en font pas partie et seront détaillées dans le chapitre 10. Disons simplement pour le moment que les constantes de chaînes de caractères sont, comme nous l'avons vu dans le premier programme, des suites de caractères placées entre guillemets : `"Hello World"`.

2.1.1 Les entiers

Le type des entiers est nommé `int`. Les constantes de ce type ont la forme habituelle dans les langages de programmation : `10  0  -5  +3748`. Ce type peut être qualifié par deux types d'annotations qui permettent de moduler leur capacité à être signés ou leur taille.

2.1.1.1 Signe

Le modificateur `unsigned` indique des entiers sans signe, donc toujours positifs. Le modificateur `signed` fait l'inverse, indiquant des entiers pouvant être positifs ou négatifs. Sans spécification particulière, les entiers sont signés, donc implicitement `signed`. Nous reviendrons sur l'impact de cette caractéristique juste après avoir parlé de taille d'entiers ainsi que dans la section d'approfondissement 16.6.1.

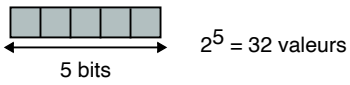
2.1.1.2 Taille

La taille en nombre de bits d'un entier peut être modulée par les trois modificateurs `short`, `long` et `long long` selon le tableau suivant. Sans aucune spécification, le type `int` a une taille de 32 bits. Il est possible d'utiliser indifféremment `short` ou `short int`, `long` ou `long int` : le « terme » `int` est implicite.

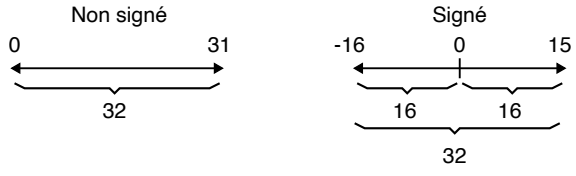
Type	Bits
`short int`	16
`int`	32
`long int`	32 ou 64
`long long int`	64

2.1.1.3 Plage de représentation

Avec un nombre n de bits, le codage binaire d'un entier permet de représenter 2^n valeurs. C'est une représentation en base 2. Ainsi, la taille d'un entier définit le nombre de valeurs qu'il peut représenter.



Quand un entier de taille n bits est non signé, les valeurs qu'il peut prendre sont comprises entre 0 et $2^n - 1$ inclus. Quand un entier de taille n bits est signé, un de ses bits, celui de poids fort (i.e. le plus à gauche lorsque l'on l'écrit) est utilisé pour représenter le signe. Fort logiquement, sa plage de valeurs représentable change : elle contient toujours 2^n valeurs possibles, mais elles sont équitablement réparties autour de 0, sachant que 0 est considéré comme positif. Donc il peut représenter des valeurs de -2^{n-1} à $2^{n-1} - 1$.



Les plages de représentation d'un entier en fonction de s'il est signé et de sa taille sont résumées dans le tableau suivant. Dans la section 16.6.1 sur les conversions, nous reviendrons sur les conséquences de ces notions de tailles et de signes, qui peuvent être source de bugs excessivement difficiles à trouver. En effet, le dépassement de capacité reste un problème totalement silencieux.

	Bits	Min	Max
`short`	16	-32768	32767
`unsigned short`	16	0	65535
`int`	32	-2^{31}	$2^{31}-1$
`unsigned int`	32	0	$2^{32}-1$
`long`	32 ou 64	dépend de l'architecture	
`unsigned long`	32 ou 64	dépend de l'architecture	
`long long`	64	-2^{63}	$2^{63}-1$
`unsigned long long`	64	0	$2^{64}-1$

2.1.2 Les réels

Le type des réels est nommé `float` ou `double`. On appelle parfois les réels des « flottants » à cause du fait que leur représentation utilise la méthode de la virgule flottante (norme IEEE 754). Différentes formes de constantes, sont possibles, comme dans la majorité des langages de programmation, allant d'un nombre sans partie décimale à la notation exponentielle en passant par l'habituelle écriture à virgule : `3.14159` `-5e-3` `+4.5` `+6.7e+3`. Il est également possible d'utiliser des constantes entières comme flottants, mais le compilateur fera silencieusement une conversion en un réel.

Les `float` et les `double` sont toujours signés : cela fait intrinsèquement partie de leur représentation. Leurs tailles diffèrent : le type type `float` a une taille de 32 bits alors que le type `double` a une taille de 64 bits.

Important

C'est `float` ou `double`. Le « terme » `double` n'est pas un modificateur de `float`. Dit autrement, `double float` n'existe pas.

Même si le codage des réels est très différent de celui des entiers (une mantisse – avec un bit de signe – et un exposant), comme pour les entiers, la plage de valeurs réelles représentables dépend de la taille. De plus, les réels étant denses, entre deux réels il existe une infinité de réels. Il existe donc en plus une notion de *précision* (c'est-à-dire la distance entre deux valeurs représentables consécutives) qui dépend de cette taille. Les plages de valeurs représentables selon le type de flottant sont résumées dans le tableau suivant.

	Taille (bits)	Valeur		
		Min	Max	Précision max
float	24 + 8	$1.1 \cdot 10^{-38}$	$3.4 \cdot 10^{38}$	10^{-6}
double	53 + 11	$2.2 \cdot 10^{-308}$	$1.8 \cdot 10^{308}$	10^{-15}

La précision indiquée dans ce tableau est la meilleure qui puisse être obtenue : ce n'est pas la précision disponible dans l'absolu. En effet, la précision dépend de la plage de valeur où se trouve le nombre à représenter. Schématiquement, pour des réels de très grande valeur absolue, la précision sera plus faible que pour des réels proches de 0. Dans le chapitre 16.6.2 sur les conversions, nous reviendrons sur les conséquences de cette représentation.

2.1.3 Les caractères

Le type des caractères, donc des éléments constitutifs des chaînes de caractères est `char`. Ce type est en réalité un type entier codé sur 8 bits, donc qui permet de représenter 256 caractères. Les constantes de caractères sont écrites entre (simples) apostrophes, avec la lettre au milieu qui représente le caractère voulu : `'A'` `'b'` `','`.

Important

Une chaîne et un caractère ne sont pas du même type. Il n'y a aucune « équivalence » entre les deux. La constante `','` est le caractère s'affichant comme une virgule. La constante `","` est une chaîne qui ne contient qu'un seul caractère, la virgule.

Certains caractères ne correspondent pas à « des lettres » au sens de la typographie : par exemple le retour à la ligne, la tabulation. D'autre part, puisque l'apostrophe sert à délimiter une constante de caractère, comment peut-on écrire la constante caractère qui représente l'apostrophe lui-même ?

Les caractères spéciaux sont notés avec l'échappement « \ ». Les principaux sont récapitulés dans le tableau ci-dessous. Nous reviendrons sur le `'\0'` lorsque nous aborderons les chaînes de caractères dans le chapitre 10.

`'\n'`	:	retour à la ligne
`'\\'`	:	le caractère \ (“*backslash*”) lui-même
`'\''`	:	le caractère ' (apostrophe) lui-même
`'\t'`	:	la tabulation
`'\0'`	:	la fin de chaîne de caractères

Il est possible de qualifier `char` par `signed` ou `unsigned`. Nous pouvons donc nous poser la question que nous nous étions posée pour les `int` : par défaut, un `char` est-il signé ou non signé ?

Ce point est une sordide subtilité du langage C que nous passons volontairement sous silence. Nous allons considérer que ce type ne doit être utilisé que dans le contexte d'éléments constitutifs de chaînes de caractères. Dit autrement, les `char` n'auront pour but que de représenter les « lettres » de messages. Et on se moquera bien de savoir quelle est la valeur entière sur 8 bits qui permet de représenter `'A'` et *a fortiori* si elle est positive ou négative.

2.1.4 Les absents : les booléens

Il n'existe pas de type booléen en langage C. Tout entier peut être utilisé comme valeur de vérité, avec la convention que 0 est équivalent à « faux » et toute autre valeur est équivalente à « vrai ».

Dans le chapitre 3 traitant des définitions, nous verrons un moyen technique de masquer cette absence. Néanmoins, le principe restera le même.

2.2 En résumé

- Il existe 3 types scalaires en C : les entiers, les réels et les caractères.
- Les entiers sont de type `int`.
- Les entiers peuvent être signés (`signed`) ou non signés (`unsigned`). Par défaut ils sont signés et peuvent donc représenter des valeurs négatives.
- Il est possible de moduler la taille des entiers avec `short`, `long` ou `long long`.
- Taille et signe changent la plage des valeurs représentables.
- Les réels sont de type `float` ou `double`.
- Les réels sont toujours signés.
- La taille change la plage des valeurs représentables.
- Les caractères sont de type `char`.
- Ce type est un entier sur 8 bits : il permet de représenter 256 valeurs possibles.
- Les constantes de caractère sont écrite entre (simples) apostrophes : `'A'`.
- Les caractères spéciaux sont représentés en utilisant l'échappement `\` : `'\n'`.
- Il n'y a pas de booléens en C, les entiers jouent ce rôle.
- La valeur 0 représente « faux », toutes les autres représentent « vrai ».

Chapitre 3

Les définitions

Le langage C est un langage impératif. Un programme est une suite de définitions de types, de variables et de fonctions. L'une de ces fonction, `main`, a un statut spécial : c'est par elle que débute l'exécution du programme.

3.1 Définition de fonction

Les fonctions sont les éléments exécutables de base. Une fonction enchaîne des instructions dont certaines parties impliquent des expressions. La définition d'une fonction comporte 4 parties :

```
type-retour nom (type arg, ..., type arg) { corps }
```

```
int min_int (int x, int y) {
  if (x <= y) return x ;
  else return y ;
}
```

1. Le type de la valeur qu'elle retourne à l'issue de son exécution (`int` dans notre exemple).
2. Le nom de la fonction (`min_int` dans notre exemple).
3. Les types et noms de ses arguments, séparés par des virgules (`x` et `y` tous deux de type `int` dans notre exemple).
4. Son corps, une suite d'instructions entre des accolades.

Important

La définition d'une fonction ne se termine pas par un point-virgule après l'accolade fermante de son corps. Les fonctions ne peuvent pas être imbriquées en C (même si `gcc` l'autorise comme extension du langage, ce qui n'est pas portable).

Pour une fonction, la spécification du type de valeur retournée, de son nom et de ses paramètres avec leur type est en fait le « type de la fonction », appelé son *prototype*.

Lorsqu'une fonction est déclarée comme retournant une valeur d'un certain type, tous ses chemins d'exécution doivent se terminer par une instruction `return` dont l'expression argument doit être ce type. C'est bien le cas de notre fonction `min_int` qui effectue un `return` dans chaque branche de la conditionnelle `if-else`.

Dans le cas contraire, si l'option `-Wall` est activée, le compilateur émet un avertissement indiquant que certains chemins d'exécution terminent sans retourner de valeur. Sans l'option `-Wall`, aucun avertissement n'est émis et le bug passera inaperçu. D'où le caractère indispensable de cette option tel que cela a été prêché dès le chapitre 1.4 !

```c
int min_int_wrong (int x, int y) {
  if (x <= y) x++ ;  /* Plus de return ici... */
  else return y ;
}
```

```
$ gcc -c wrong.c
$ gcc -c -Wall wrong.c
wrong.c: In function 'min_int':
wrong.c:4:1: warning: control reaches end of non-void function [-Wreturn-type]
 }
 ^
```

Sans rentrer dans les détails mais pour ne pas laisser de mystère, l'option `-c` utilisée ici sert à compiler le fichier sans chercher à créer l'exécutable. C'est utile lorsqu'un programme est composé de plusieurs fichiers sources à compiler séparément, donc que le `main` n'est pas dans le fichier.

La première compilation n'a levé aucun avertissement alors que la seconde, avec l'option `-Wall`, indique qu'à la ligne 4, correspondant à l'accolade fermante, la fonction peut terminer sans retourner de valeur. En effet, dans le cas où l'exécution ne passe pas dans le `else`, il n'y a pas de `return` à ce point.

Certaines fonctions peuvent ne pas avoir de valeur à retourner. C'est le cas lorsque leur exécution se contente de produire de effets comme l'affichage. Dans ce cas, leur type de retour est `void`. C'est le cas de cette nouvelle fonction qui se contente d'afficher qui est le minimum de ses arguments.

```c
void print_min_int (int x, int y) {
  if (x <= y) printf ("Premier\n") ;
  else printf ("Second\n") ;
}
```

Quand une fonction a le type `void`, l'instruction `return` finale est optionnelle : soit on la met sans argument, soit on l'omet. L'instruction `return` peut toutefois rester nécessaire dans le cas où une fonction doit terminer à d'autres endroits que la fin syntaxique de son corps.

Les points de sortie multiples sont à pratiquer avec parcimonie et raison car ils peuvent rendent l'algorithme d'une fonction difficile à comprendre. En effet, au lieu d'avoir un point de sortie bien identifié, avec une condition de sortie unique, cette condition de sortie se retrouve dispersée à différents endroits dans la fonction.

3.2 Définition de variable

Les variables sont des réceptacles d'information permettant de mémoriser cette information pour une utilisation ultérieure. Le besoin de variable est intrinsèquement lié à la notion d'état qui varie au cours du temps.

Nous pouvons tout à fait définir une fonction qui calcule $n!$ sans définir de variable. C'est une forme de programmation dite fonctionnelle par opposition à la programmation dite impérative (dans les faits, on aime bien avoir les deux paradigmes dans un langage).

```
int fact (int n) :=
  si n = 0 retourner 1
  sinon retourner (n * fact (n - 1)
```

Néanmoins, si nous voulons écrire la même fonction en utilisant une boucle, il va y avoir besoin de plusieurs variables : une qui joue le rôle d'accumulateur et l'autre de compteur. Et dans les deux cas, leurs valeurs varient au cours du temps : ce sont les mêmes variables mais leurs contenus changent.

```
int fact (int n) :=
  var res = 1
  pour var i = 1 à n
    res := res * i
  retourner res
```

En C, la définition d'une variable est composée de 3 ou 4 parties :

```
type nom ;
type nom = init ;
```

1. Son type, indiquant la forme des valeurs qu'elle peut contenir.
2. Son nom.
3. De manière optionnelle, une initialisation.
4. Un point-virgule final.

```c
int main () {
  int a = 5 ;
  int b ;
  float c = 3.14159 ;
  char d = 'D' ;
  ...
}
```

Important

L'initialisation est optionnelle lors de la définition d'une variable. Pour autant, lire dans une variable non initialisée avant qu'une écriture n'y ait été faite est une erreur. En effet, cette variable contiendra bien une valeur : celle qui se trouve en mémoire à l'endroit où le compilateur aura logé cette variable. Mais en aucun cas cette valeur ne sera prédictible. D'une exécution à une autre, elle pourra changer rendant le comportement du programme non déterministe.

L'initialisation lors de la définition est une bonne pratique. Lorsque `gcc` est invoqué avec l'option `-Wall`, il détecte **parfois** de telles erreurs. Malheureusement, cette détection est un problème indécidable dans le cas général et certains de ces bugs passent au travers des mailles du filet de certains compilateurs.

Les deux fonctions `f` et `g` suivantes présentent une telle erreur, toutefois `gcc` (version 12.2.0) n'en détecte qu'une occurrence. *A contrario*, le compilateur `clang` (version 6.0.0) détecte les deux.

```c
int f (int x) {
  int i ;
  i = i + 4 ;
  return i ;
}

int g (int x) {
  int i ;
  if (x < 0) i = 10 ;
  return i ;
}
```

```
$ gcc -c -Wall foo.c
foo.c: In function ’f’:
foo.c:3:5: warning: ’i’ is used uninitialized [-Wuninitialized]
    3 |   i = i + 4 ;
      |   ~~^~~~~~~
foo.c:2:7: note: ’i’ was declared here
    2 |   int i ;
      |       ^
$ clang -c -Wall foo.c
foo.c:3:7: warning: variable ’i’ is uninitialized when used here
      [-Wuninitialized]
  i = i + 4 ;
      ^
foo.c:9:7: warning: variable ’i’ is used uninitialized whenever ’if’ condition
      is false [-Wsometimes-uninitialized]
  if (x < 0) i = 10 ;
      ^~~~~
```

3.2.1 Variable globale, variable locale

Une variable peut être globale au programme. Dans ce cas, elle est définie en dehors de toute fonction et elle sera accessible durant toute l'exécution du programme, depuis n'importe quelle fonction.

```c
int vglob = 25 ;   /* Variable globale : accessible partout. */

void f () {
  vglob = vglob + 1 ;
}

int main () {
  vglob = vglob * 10 ;
  return 0 ;
}
```

A contrario, une variable peut être locale à une fonction, plus généralement à un bloc (d'instructions groupées au sein d'accolades). Une telle variable n'existe et n'est accessible qu'au sein du bloc dans lequel elle est définie. Elle disparaît à la fin du bloc. De ce fait, une variable locale à une fonction est créée à l'entrée dans la fonction et est détruite à sa sortie.

```c
int f () {
  int x = 0 ;
  x = x + 1 ;  /* Variable locale x. */
  return x ;
}

int g () {
  x = x + 10 ;  /* Aucune variable x visible. */
  return x ;
}

int h (int arg) {
  if (arg < 0) {
    int y = 0 ;
    y = y + 1 ;
  }
```

→

```
  return y + 20 ;  /* Variable y n'existe plus. */
}
```

```
$ gcc -Wall foo.c
foo.c: In function 'g':
foo.c:8:3: error: 'x' undeclared (first use in this function)
   x = x + 10 ;
   ^
foo.c: In function 'h':
foo.c:17:10: error: 'y' undeclared (first use in this function)
   return y + 20 ;
          ^
```

Si deux variables de même nom existent, une globale et une locale, alors la variable locale a la priorité. Dans le programme ci-dessous, la variable globale `x` n'est pas modifiée par la fonction `f`. Dans l'utilisation de `printf`, la séquence `%d` permet d'afficher un entier et sera détaillée dans le chapitre 6.

```
#include <stdio.h>

int x = 50 ;

int f () {
  int x = 0 ;
  x = x + 1 ;
  return x ;
}

int main () {
  printf ("Avant: %d\n", x) ;
  printf ("Retour f: %d\n", f ()) ;
  printf ("Ensuite: %d\n", x) ;
  return 0 ;
}
```

```
$ gcc -Wall foo.c -o foo.x
$ ./foo.x
Avant: 50
Retour f: 1
Ensuite: 50
```

Jusqu'à la norme C90, il n'était possible de définir des variables locales qu'en début de bloc, interdisant de mélanger instructions et définitions de variables. Désormais, les variables locales peuvent être définies « au vol » n'importe où dans un bloc (pas uniquement au début). Leur durée d'existence est alors celle du bloc immédiatement englobant. Le programme suivant est donc tout à fait correct.

```
#include <stdio.h>

int main () {
  int x = 0 ;
  printf ("x = %d\n", x) ;
  int y = x + 10 ;
  y = y + 1 ;
  printf ("y = %d\n", y) ;
```

```
  return 0 ;
}
```

3.3 Définition de type scalaire énuméré (`enum`)

Il est parfois intéressant de définir de nouveaux types atomiques dont nous souhaitons nommer explicitement les valeurs. Par exemple, le type absent des booléens gagnerait à exister, avec deux valeurs que nous pourrions nommer `faux` et `vrai` ou `false` et `true` ou `non` et `oui`. Autre exemple, les points cardinaux permettant de donner des directions pourraient avoir les valeurs `nord`, `sud`, `est` et `ouest`.

Une première solution est de représenter ces valeurs par des entiers. Nous décidons arbitrairement que `nord` = 0, `sud` = 1, `est = 2` et `ouest = 3`. Un tel encodage amène au moins quatre constats non satisfaisants.

1. Pourquoi devrions-nous choisir explicitement ces valeurs plutôt que d'autres toutes aussi légitimes ? C'est du bruit dans notre réflexion.
2. À chaque fois que nous devrons faire référence au sud dans le programme, il faudra utiliser la valeur 1, ce qui est tout sauf lisible.
3. Si nous décidons de changer l'attribution des valeurs (pour quelle que raison que ce soit, bonne ou mauvaise), il faut repasser sur tout le programme et changer les « bonnes » valeurs entières. C'est tout sauf maintenable.
4. Il est très facile de se tromper entre les valeurs de ces simples entiers, encore plus lorsque plusieurs personnes travaillent sur le même programme.

```
int cherche (...) {
  int nb_essai = 0 ;
  int d = 2 ;          /* On commence par l'est. */
  ...
  if (d == 0) printf ("Pas ici.\n") ;        /* Nord ? */
  else if (d == 3) printf ("Non plus.\n") ; /* Ouest ? */
  nb_essai = nb_essai + 1 ;
  ...
```

La définition d'un type, dit *énuméré*, permet de s'affranchir de tous ces problèmes en laissant au compilateur le soin d'attribuer des valeurs entières, de manière cohérente et sûre, à des nouveaux symboles qui deviendront les valeurs de ce type. La syntaxe générale d'une définition de type énuméré est donc :

`enum` *nom–type* `{` *nom–valeur*$_1$`, ...,` *nom–valeur*$_n$ `} ;`

La définition

```
enum direction_t { nord, sud, est, ouest } ;
```

définit un nouveau type nommé `enum direction_t` dont les valeurs sont exclusivement les symboles `nord`, `sud`, `est` et `ouest`. On définit des variables dont le type est un énuméré de la même façon que des variables d'un autre type, donc son type suivi de son nom :

```
enum direction_t dir = sud ;
```

Important

Lorsque nous définissons un type `enum t`, le nom de ce type n'est pas juste `t`. C'est `enum t` : le qualificatif `enum` fait partie intégrante du nom de ce type.

Après leur définition, les valeurs d'un type énuméré s'utilisent comme n'importe quelle autre constante, puisque ce sont en fait de nouvelles constantes. Nous pouvons les passer en argument de fonction, en faire des tableaux (abordés dans le chapitre 9), les utiliser dans des conditionnelles, etc.

```
enum direction_t { nord, sud, est, ouest } ;

int cherche (...) {
  int nb_essai = 0 ;
  enum direction_t d = est ;      /* On commence par l'est. */
  ...
  if (d == nord) printf ("Pas ici.\n") ;        /* Nord ? */
  else if (d == ouest) printf ("Non plus.\n") ; /* Ouest ? */
  nb_essai = nb_essai + 1 ;
  ...
```

3.3.1 Des booléens

Nous avons dit dans la section 2.1.4 que les booléens n'existent pas en C. Il est possible de les simuler en définissant un type énuméré contenant deux valeurs `false` et `true` :

```
enum bool { false, true } ;
```

ce qui nous permet alors de définir des variables booléennes comme :

```
enum bool is_succes = false ;
```

Nous laissons donc le compilateur définir quelles sont les valeurs entières qui représentent `false` et `true`. Cela fonctionne car, par défaut, la première valeur d'un `enum` est représentée par 0, la seconde par 1 et ainsi de suite. Donc `false` est bien égal à 0.

La bibliothèque standard de C fournit justement un « pseudo-type » booléen, nommé `bool` avec justement les valeurs `true` et `false`. Pour y accéder, il suffit d'inclure le fichier d'en-tête `<stdbool.h>`. Nous pouvons alors définir une variable de ce type par :

```
bool b = false ;
```

où le nom du type est bien `bool` et non `enum bool`. L'`enum` sous-jacent a été caché par mécanisme que nous ignorons pour le moment mais qui sera décrit dans la section d'approfondissement 16.1.

3.4 En résumé

- Un programme en C est une suite de définitions de fonctions, de variables et de types.
- Une définition de fonction a la forme :
  $type\text{–}retour\ nom$ `(`$type_1\ arg_1$`, ...,` $type_n\ arg_n$`) {` $corps$ `}`
- Une fonction ne retournant aucune valeur a pour type de retour `void`.
- Une fonction déclarée comme retournant un certain type doit retourner une valeur de ce type sur tous ses chemins d'exécution.
- Une définition de variable a la forme :
  $type\ nom$ `[=` $init$`] ;`
  où l'initialisation est optionnelle (mais plus que conseillée).
- Lire dans une variable avant qu'elle n'ait reçu une valeur est une erreur aux effets imprédictibles.
- Une variable globale est accessible dans tout le programme.
- Une variable locale est uniquement accessible dans le bloc où elle est définie.
- Un type énuméré permet de définir un nouveau type scalaire avec des valeurs symboliques.
- Le nom d'un type énuméré `foo_t` est `enum foo_t`.
- On peut récupérer un type `bool` avec les valeurs `true` et `false` en faisant `#include <stdbool.h>`.

Chapitre 4

Expressions et instructions

En introduction de la section 3.1, nous avons écrit : « *une fonction enchaîne des instructions dont certaines parties impliquent des expressions* ». Ces deux catégories syntaxiques se différencient schématiquement par le fait que :

- quand elle est évaluée, une expression a une valeur : en quelque sorte, elle « vaut quelque chose »,
- quand elle est exécutée, une instruction a un effet : en quelque sorte, elle « fait quelque chose ».

Nous verrons plus tard que cette différence est un peu plus subtile pour certaines instructions qui ont quand même une valeur, même si l'on n'est pas obligé de l'exploiter. Garder en tête cette différence schématique est amplement suffisant dans l'immense majorité des cas.

4.1 Expressions

En C, nous retrouvons les mêmes genres d'expressions que dans les langages habituels, avec parfois une syntaxe légèrement différente, à savoir :

- les constantes : `3`, `'A'`, `5e-4`, `sud`,
- les variables (locales et globales),
- les opérateurs : `! (x * 3.1416 + 2.5 > f (x + 2))`,
- les appels de fonction `cos (f (-x)) + sin (y * 0.5)`.

Les opérateurs permettent de combiner des expressions pour faire de nouvelles expressions. Ils sont soit unaires, donc n'ont qu'une seule opérande, soit binaires, donc en ont 2. Il en existe un opérateur ternaire (i.e. avec 3 opérandes) que nous verrons dans la 16.7 du chapitre d'approfondissement. Du point de vue de leur sémantique, les opérateurs sont classés en quatre catégories : arithmétiques, relationnels, logiques ou bit-à-bit (cette dernière catégorie étant abordée dans la section d'approfondissement 16.10).

4.1.1 Constantes

Nous retrouvons donc toutes les constantes des types de base et énumérés que nous avons décrits dans les sections 2.1 et 3.3.

Important

Sciemment, nous n'avons pas inclus les constantes chaînes de caractères dans cette liste. Même si nous avons déjà vu des constantes chaînes de caractères (comme le message `"Hello world !!"` de notre premier programme), ces constantes ne se manipulent pas aussi simplement que des entiers ou des réels. Dit autrement, ce ne sont pas des « constantes normales » à cause de la manière dont les chaînes de caractères sont représentées. Le chapitre 10 est dédié à leur étude.

4.1.2 Expressions arithmétiques

Les opérateurs arithmétiques permettent de former des expressions arithmétiques, donc « numériques ». Syntaxiquement, rien ne différencie leur version « entière » de leur version « réelle ».

+	addition
-	soustraction
*	multiplication
/	division
%	reste de la division entière

À l'exception du modulo % qui n'opère que sur des entiers,

- lorsque les opérandes d'un opérateur sont entières, l'opération associée est celle sur les entiers,
- lorsque les opérandes sont réelles, c'est l'opération sur les flottants qui est invoquée,
- lorsque l'on mélange entiers et réels, l'opérande entière est convertie en réel et l'opération flottante est utilisée.

Important

En présence de deux opérandes entières, l'opérateur `/` est la division entière. Donc, `7 / 2 = 3` et `1 / 2 = 0`. La section d'approfondissement 16.6.1.3 revient en détail sur ce point.

Les priorités des opérateurs sont celles habituelles en mathématiques. En cas de priorités égales, les opérateurs sont associés de gauche à droite (cf. annexe 18.2). Cet ordre peut avoir une importance capitale dans les calculs.

```c
#include <stdio.h>

int main () {
  printf ("%d\n", 100 / 5 / 2) ;
  printf ("%d\n", 100 / (5 / 2)) ;
  return 0 ;
}
}
```

```
$ gcc -Wall arith.c -o arith.x
$ ./arith.x
10
50
```

Important

En cas de doute sur l'ordre d'évaluation des expressions, il est indispensable d'ajouter des parenthèses. Quand cet ordre n'est pas évident mais primordial, les parenthèses sont aussi recommandées. En effet, vous êtes peut-être un expert de C qui connaît les priorités subtiles sur le bout des doigts, mais vos collègues ne le sont peut-être pas.

L'opérateur % est souvent appelé affectueusement *modulo*. Néanmoins, son comportement est celui du « reste de la division entière ». La différence subtile survient lorsque l'on l'utilise avec des entiers négatifs.

Pour aller plus loin

La division entière de a par b donne un quotient q et un reste r reliés par la relation $a = b \times q + r$. Le reste de la division entière est donc $r = a - b \times q$.

→

Avant la norme C99, le comportement du % avec des négatifs était à la discrétion du compilateur. Désormais, l'opérateur % doit tronquer le quotient q vers 0. Avec $a = -7$ et $b = 3$, la valeur réelle du quotient est $q = -2.\widehat{3}$ qui est arrondie, vers 0 en -2. De ce fait `-7 % 3` $= -7 - 3 \times (-2) = -1$.

Python a fait le choix inverse : arrondir vers $-\infty$. Donc $-2.\widehat{3}$ se retrouve arrondi en -3. De ce fait `-7 % 3` $= -7 - 3 \times (-3) = 2$.

```c
#include <stdio.h>

int main () {
  printf ("%d\n", 7 % 3) ;
  printf ("%d\n", -7 % 3) ;
  printf ("%d\n", 7 % -3) ;
  printf ("%d\n", -7 % -3) ;
  return 0 ;
}
```

```
$ gcc -Wall mod.c -o mod.x
$ ./mod.x
1
-1
1
-1
```

4.1.3 Expressions relationnelles

Les opérateurs relationnels permettent de former des expressions de comparaison entre scalaires. Leur priorité est inférieure à celle des opérateurs arithmétiques (cf. annexe 18.2).

==	égalité
!=	différence
<	inférieur
>	supérieur
<=	inférieur ou égal
>=	supérieur ou égal

Important

Le test d'égalité est == et non le simple = qui est l'affectation. Les problèmes engendrés par cette confusion sont détaillés dans la section 4.2.1 qui traite de l'affectation.

Important

Les opérateurs relationnels sont dédiés aux scalaires, donc aux entiers (au sens large, ce qui inclut les `char` et les `enum`) et aux réels. En aucun cas ils ne doivent être utilisés avec les chaînes de caractères car leur comportement n'est pas celui que l'on pourrait naïvement espérer. Le chapitre 10 présentant la manipulation des chaînes de caractères décrit comment les comparer (plus précisément en section 10.2.6).

Le moyen mnémotechnique pour se souvenir de l'opérateur « différent » est que « différent c'est non égal ». Or comme nous allons le voir dans la section suivante, « non » c'est `!`.

Important

Tester si une valeur est comprise entre deux autres sous la forme `x <= y <= z` est incorrect. De par l'associativité gauche-droite des opérateurs logiques, cela est lu comme `(x <= y) <= z`. Or `(x <= y)` est une valeur de vérité. Donc l'expression complète revient à tester si `z` est supérieur ou égal à une valeur de vérité, ce qui n'a sémantiquement aucun sens.

4.1.4 Expressions logiques

Les opérateurs logiques permettent de combiner des valeurs de vérité. Comme en logique, la conjonction est prioritaire sur la disjonction et la négation a la plus forte priorité. Comme pour les opérateurs arithmétiques, l'associativité se fait de gauche à droite. Leur priorité est inférieure à celle des opérateurs relationnels (cf. annexe 18.2).

`!`	négation
`&&`	et (conjonction)
`||`	ou (disjonction)

Les opérateurs logiques binaires `&&` et `||` sont dits *paresseux* : ils n'évaluent leurs expressions arguments que si cela est nécessaire pour déterminer la valeur globale de la condition.

- `e1 && e2` :
  - Si `e1` s'évalue en `false`, alors `e2` n'est pas calculée (le résultat est forcément `false`).
  - Si `e1` s'évalue en `true`, alors `e2` est calculée.
- `e1 || e2` :
  - Si `e1` s'évalue en `true`, alors `e2` n'est pas calculée (le résultat est forcément `true`).
  - Si `e1` s'évalue en `false`, alors `e2` est calculée.

4.1.5 Appel de fonction

Comme déjà aperçu dans la section 1.3 avec le programme "Hello world", appeler une fonction se fait en faisant suivre son nom de la liste, entre parenthèses, de ses arguments, séparés par des virgules. Un appel de fonction étant une expression, il peut servir d'argument à l'appel d'une autre fonction. Dans

```
printf ("Cos (x) = %f.\n", cos (x)) ;
```

la fonction d'affichage `printf`, est appelée en lui fournissant deux arguments. Le premier est une chaîne de caractères qui représente la forme du message que nous voulons afficher. Le second est le résultat d'un autre appel de fonction, celui à `cos` dont le seul argument est une variable `x`.

Important

Utiliser le résultat d'un appel de fonction n'a de sens que si cette fonction n'est pas de type `void`.

4.2 Instructions (hors contrôle)

Comme exposé en introduction de ce chapitre, une instruction a un effet, une action. Cet effet peut agir sur la mémoire, l'affichage, des fichiers, etc. Il peut aussi agir le séquencement de l'éxécution du programme. C dispose bien entendu des instructions de contrôle habituelles des langages de programmation, à savoir les conditionnelles et les boucles qui feront l'objet de la section 5.2. La présente section présente donc les instructions que l'on pourrait qualifier de « simples ».

Important

À l'exception des blocs, toutes les instructions sont terminées par un point-virgule.

4.2.1 Affectation

L'affectation est une instruction. Dans

```
x = x + 1 ;
```

où l'on note le point-virgule final, l'expression `x + 1` est évaluée et la valeur résultat est stockée dans la variable `x`, ce qui écrase son ancienne valeur. Il y a donc un effet sur la mémoire.

Bien entendu, x est une variable qui aura été préalablement déclarée. Notons que la déclaration d'une variable combinée à son initialisation peut être vue comme une déclaration suivie d'une affectation :

```
int x = 10 ;
```

est équivalent à :

```
int x ;
x = 10 ;
```

Important

L'affectation est =, le test d'égalité est ==.

La confusion entre les deux ne provoque pas d'erreur à la compilation car, même si nous avons présenté l'affection comme une instruction . . . elle a aussi une valeur ! Donc oui, une fois encore, notre classification est un peu simpliste et l'affectation peut être utilisée comme expression. La valeur d'une affectation est la valeur affectée. Dit autrement, `x = 3` « vaut » 3.

Considérons le programme suivant :

```c
#include <stdio.h>

int main () {
  int x = 0 ;
  if (x = 10) printf ("C'est pas nul.\n") ;
  printf ("x: %d.\n", x) ;
  return 0 ;
}
```

```
$ gcc badcond.c -o badcond.x
$ ./badcond.x
C'est pas nul.
x: 10.
$ gcc -Wall badcond.c -o badcond.x
badcond.c: In function 'main':
badcond.c:5:7: warning: suggest parentheses around assignment used as truth value [-Wparentheses]
   if (x = 1) printf ("C'est pas nul.\n") ;
       ^
```

Nous commençons par compiler sans l'option `-Wall`. Aucun problème n'est signalé. Lors de l'exécution, la variable `x` est affectée avec la valeur `10` et le résultat de cette affectation est donc également `10`. Or, nous avons vu qu'en matière de « booléens » en C, $\neq 0$ signifie « vrai ». Donc la condition du test est toujours vraie et l'exécution passe toujours dans l'instruction du `if`.

Il est très intéressant de noter que lorsque nous avons compilé avec l'option `-Wall`, un avertissement a été généré. Ce n'est certes par une erreur, l'exécutable est quand même créé, mais cela signale quand même quelque chose de suspect. En conséquence, il semble judicieux de rappeler les trois informations suivantes :

Important

- Toujours compiler avec l'option `-Wall`.
- Ne jamais ignorer un avertissement.
- Toujours corriger la source d'un avertissement pour le faire disparaître.

Pour aller plus loin

L'utilisation intentionnelle d'une affectation dans une condition se rencontre parfois dans une forme très idiomatique de C lorsque l'on souhaite condenser une affectation suivie d'un test. Plutôt qu'écrire :

```
combien = f (...) ;
if (combien != 0) faireqqchose (combien) ;
```

où la fonction `faireqqchose` requiert que son argument soit dfférent de 0, on condense ces deux lignes de code en :

```
if ((combien = f (...))) faireqqchose (combien) ;
```

où il faut noter les doubles parenthèses autour de la condition comportant l'affectation. Ces doubles parenthèses indiquent explicitement que cette affectation est volontaire et ne relève pas d'une confusion avec ==. C'est le sens du message d'avertissement émis ci-dessus par le compilateur : « *Vous voulez réellement faire une affectation ? Mettez des parenthèses autour de cette affectation.* ».

L'utilisation typique est la mémorisation d'une adresse dans un pointeur (cf. chapitre 7) tout en faisant la vérification que cette adresse est valide avant de l'utiliser dans la suite.

4.2.2 `return`

Cette instruction permet à une fonction d'interrompre son exécution en retournant à celle qui l'a appelée. Si la fonction a été définie comme retournant un valeur d'un certain type, alors `return` devra être suivi d'une expression de ce type sur tous ses chemins d'exécution.

Si la fonction a été déclarée `void`, alors le `return` final est optionnel. Les autres éventuels `return` ne sont pas suivis d'une expression. Pour plus de détails, se référer à la section 3.1 qui décrit les définitions de fonctions.

4.2.3 Affectation-arithmétique abrégée

Dans un soucis de concision, C propose les instructions « d'affectation-arithmétique abrégées » présentées dans le tableau suivant. Les formes de droite et de gauche sont équivalentes (à l'exception subtile de ++ et - dans certains cas, dont il est question après le tableau).

`i++`	`++i`	≡	`i = i + 1`
`i-`	`-i`	≡	`i = i - 1`
`i += 3`		≡	`i = i + 3`
`i -= 3`		≡	`i = i - 3`
`i *= 3`		≡	`i = i * 3`
`i /= 3`		≡	`i = i / 3`
`i %= 3`		≡	`i = i % 3`
`i |= 3`		≡	`i = i | 3`
`i &= 3`		≡	`i = i & 3`
`i ^= 3`		≡	`i = i ^ 3`

Tout comme l'affectation (cf. section 4.2.1), ces instructions ont également une valeur. Donc, elles peuvent être utilisées dans un contexte d'expression.

Pour aller plus loin

Les opérations d'incrémentation `++` et de décrémentation `-` existent sous deux formes, « pré » et « post ». Comme pour l'affectation, ces instructions sont également des expressions : elles ont une valeur.

- La pré-incrémentation `++i` incrémente `i` puis ensuite récupère sa valeur « qui est celle de l'instruction ».
- La post-incrémentation `i++` commence par récupérer la valeur de `i` puis incrément la variable.

La différence est donc significative dans le cas où la « valeur de cette instruction » est utilisée comme expression, par exemple dans une affectation. Considérons le programme suivant.

```
#include <stdio.h>

int main () {
  int x = 0, y, z ;
  y = x++ ;
  z = ++x ;
  printf ("x: %d, y: %d, z: %d\n", x, y, z) ;
  return 0 ;
}
```

Son exécution aura pour effet d'afficher `x: 2, y: 0, z: 2`, en accord avec les opérations élémentaires effectuées, comme le récapitule le tableau suivant.

`x = 0`	$x \leftarrow 0$	
`y = x++`	$y \leftarrow x$	(i.e. $y \leftarrow 0$)
	$x \leftarrow x + 1$	(i.e. $x \leftarrow 0 + 1$)
`z = ++x`	$x \leftarrow x + 1$	(i.e. $x \leftarrow 1 + 1$)
	$z \leftarrow x$	(i.e. $z \leftarrow 2$)

4.2.4 Bloc

Un bloc permet de grouper plusieurs instructions et déclarations. Il est délimité par des accolades. Le corps de toute fonction est ainsi un bloc. Contrairement aux autres instructions, un bloc n'est pas terminé par un point-virgule. En mettre un n'est pas faux mais totalement inutile : ça revient à le faire suivre par une instruction vide, qui ne fait rien. Le compilateur se chargera bien entendu de la faire disparaître, mais autant ne pas écrire de code qui ne fait rien, on a souvent déjà bien assez à écrire avec le code qui fait quelque chose . . .

4.2.4.1 Regrouper des instructions

La première utilité des blocs est de permettre l'exécution d'un ensemble d'instructions dans le corps d'une boucle ou d'une conditionnelle (cf. chapitre 5).

Rappelons que l'indentation, même si indispensable pour la compréhension humaine d'un code source, est totalement ignorée par le compilateur. Ainsi, contrairement à Python, le regroupement d'instructions dans un bloc ne s'appuie par sur l'indentation. Le programme suivant exécute donc une boucle infinie car seule l'instruction `res = res * x ;` fait partie du corps de la boucle.

```
int fact (int n) {
  int x = 1 ;
  int res = 1 ;
```

```
  while (x <= n)
    res = res * x ;  /* Seule instruction du corps. */
    x++ ;            /* Belle indentation mais pas dans la boucle. */
  return res ;
}
```

Il faut donc encadrer les deux instructions par des accolades pour qu'elles soient considérées ensemble. En réalité, un bloc est lui-même une instruction qui en comporte plusieurs.

```
  while (x <= n) {
    res = res * x ;
    x++ ;
  }
```

Dit autrement, le corps de note boucle `while` est suivi d'une seule instruction ... qui est un bloc contenant, deux instructions.

4.2.4.2 Définir des variables locales

Comme indiqué dans la section 3.2.1, un bloc est aussi l'occasion de définir localement des variables qui n'existent que dans ce bloc. De manière évidente, cela s'illustre avec les variables locales définies dans une fonction, au début du bloc qui constitue son corps. Lorsque l'on entre dans la fonction, ses variables sont créées et initialisées : lorsque l'on quitte la fonction, ses variables cessent d'exister.

Ce mécanisme n'est pas propre au bloc corps d'une fonction et se produit pour n'importe quel bloc. C'est donc un moyen de limiter la portée (i.e. visibilité, durée de vie) d'une variable.

```
#include <stdio.h>

int main () {
  int x = 0 ;
  int sum = 0 ;
  while (x <= 10) {
    int xmore = 5 + x ;
    sum += xmore ;
    x++ ;
  }
  printf ("Sum [5, 15] = %d\n", sum) ;
  return 0 ;
}
```

Non seulement en dehors du bloc corps du `while` la variable `xmore` n'existe pas, mais à chaque itération, nous rentrons dans le bloc, donc `xmore` est redéfinie. Ainsi ce programme calcule la somme des entiers entre 5 et 15 inclus, c'est-à-dire 110.

Comme c'est déjà le cas dans l'exemple ci-dessus avec le bloc « corps de la fonction » et celui « corps du `while` », les blocs peuvent être arbitrairement imbriqués. Il est possible de définir des blocs en dehors de tout corps de boucle ou de conditionnelle. Cela permet alors seulement de restreindre la visibilité des variables.

```
#include <stdio.h>

int main () {
```

```
    int x = 10 ;
    { int y = x + 1 ;
      { int z = y * 10 ;
        printf ("%d\n", z) ;
      }     /* z n'existe plus. */
      printf ("%d\n", y) ;
    }       /* y n'existe plus. */
    printf ("%d\n", x) ;
    return 0 ;
  }
```

4.2.5 Et les appels de fonction ?

Nous avons présenté les appels de fonctions comme des expressions sous prétexte qu'ils retournaient généralement une valeur. Néanmoins, appeler une fonction qui a un effet (sur la mémoire, l'affichage, etc.) provoque bien entendu un (cet) effet. À ce titre, un appel de fonction peut être classé aussi parmi les instructions. C'est encore plus pertinent pour les fonctions de type `void`. Puisqu'elles ne retournent pas de valeurs, pourquoi les appeler sinon pour leur effet ? Parfois, certaines fonctions font les deux : elles retournent une valeur et ont un effet. Et dans certains cas, la valeur retournée n'est pas forcément utilisée (à tort ou à raison). C'est le cas de la fonction d'affichage `printf` détaillée dans le chapitre 6 : il est rare que l'on se soucie de ce qu'elle retourne, on veut surtout qu'elle affiche.

Il faut être clair : comment on classe les appels de fonctions n'a pas une grande importance. Nous voulons ici simplement dissiper d'éventuelles questions philosophiques sur leur statut.

4.3 En résumé

- Une expression est une entité qui s'évalue en une valeur.
- Une instructions est une entité dont l'exécution a un effet.
- En C, la limite n'est pas si tranchée, certaines instructions ont également une valeur et les appels de fonctions peuvent avoir des effets.
- Expressions
  - De base : variables, constantes (entiers, réels, caractères, valeurs d'`enum`).
    Les chaînes ont un statut à part.
  - Arithmétiques : `+`, `-`, `*`, `/`, `%` (reste division entière).
  - Relationnelles : `==` (égalité), `!=` (inégalité), `<`, `<=`, `>`, `>=`.
  - Logiques : `!` (négation), `&&` (et), `||` (ou).
  - Bit-à-bit (voir en section d'approfondissement 16.10).
  - Appels de fonctions : `f (x + 1, g (true))`.
- Les opérateurs arithmétiques, relationnels, logiques ne s'appliquent qu'à des scalaires.
- Les opérateurs logiques sont paresseux.
- Instructions
  - Affectation : (un seul) `=`
  - `return` pour interrompre une fonction et éventuellement retourner une valeur à la fonction appelante.
  - Opérateurs d'affectation abrégée : `+=`, `-=`, `*=`, etc.
    `x ♪= y` ≡ `x = x ♪ y`.
  - Bloc : entre `{ }` pour grouper des instructions et déclarer des variables locales.
  - Boucles et conditionnelles (cf. chapitre 5).
- Ne pas confondre l'affectation `=` avec le teste d'égalité `==`.

Chapitre 5

Conditionnelles et boucles

5.1 Conditionnelles

5.1.1 `if else`

En C, l'instruction conditionnelle habituelle des langage de programmation se présente sous la forme :

```
if (expression) instruction_t ;
else instruction_f ;
```

L'expression de condition est obligatoirement entre parenthèses. Si elle s'évalue en « vrai » (donc une valeur entière $\neq 0$), alors l'instruction $instruction_t$ est exécutée. *A contrario*, si elle s'évalue en « faux » (donc en la valeur 0), alors l'instruction $instruction_f$ est exécutée.

Dans le cas où nous souhaitons exécuter plusieurs instructions dans les branches de la conditionnelle, il faut les grouper dans un bloc, donc les encadrer par des accolades. Nous pourrions ainsi dire que la forme générale de cette construction est :

```
if (expression) {
  instructions_t ;
}
else {
  instructions_f ;
}
```

Pour aller plus loin

Mettre des accolades même si une seule instruction est à exécuter est une pratique recommandée dans le monde du logiciel critique. D'une part, cela permet part d'expliciter le caractère volontaire de l'unicité de l'instruction. D'autre part, cela évite d'oublier de rajouter des accolades si l'on modifie le corps de la boucle ou de la conditionnelle *a posteriori* pour rajouter d'autres instructions à exécuter.

Si aucun traitement n'est à exécuter dans le cas où la condition est fausse, la clause `else` peut être supprimée (plutôt que de faire un `else` avec un bloc vide). La construction conditionnelle prend alors la forme :

```
if (expression) instruction_t ;
```

ou

```
if (expression) {
  instructions_t ;
}
```

Le problème de savoir, lorsque nous avons des `if` imbriqués, auquel se rattache un `else` porte le nom de *sinon en suspens* (“*dangling else*”). En l'absence d'un « parenthésage » explicite avec des accolades, C fait le choix de rattacher un `else` au `if` le plus proche en terme d'imbrication, ce qui donne une impression d'une instruction `else if` atomique (cf. section 5.1.2 ci-dessous).

Le programme suivant est rédigé avec une indentation particulièrement trompeuse laissant penser que le `else` est le pendant du `if (x == 1)`.

```c
#include <stdio.h>

int main () {
  int x = 2 ;
  int y = 2 ;
  int z = 0 ;
  int t = 0 ;
  if (x == 1)
    if (y == 1)
      z = 10 ;
  else            /* Oh l'indentation horriblement sournoise ! */
    t = 20 ;
  printf ("z: %d t: %d\n", z, t) ;
  return 0 ;
}
```

Le `else` étant le plus proche du `if (y == 1)`, il y est associé. Il n'est pas du tout associé, comme l'indentation le laisse penser, au `if (x == 1)`. De ce fait, `x` valant 2, le premier test est faux, il n'a pas de `else`, donc aucun traitement n'est effectué. L'affichage est donc `z: 0 t: 0`.

Fort heureusement, mais uniquement en utilisant l'option `-Wall`, le compilateur repère l'ambiguïté et émet un avertissement. Une raison de plus de ne pas oublier `-Wall`...

```
$ gcc -Wall dangling.c
dangling.c: In function 'main':
dangling.c:8:6: warning: suggest explicit braces to avoid ambiguous 'else' [-Wdangling-else]
   if (x == 1)
      ^
```

5.1.2 Le `elseif/elif` n'existe pas

Certains langages propose un mot-clé `elseif/elif` pour imbriquer des conditions mutuellement exclusives. Un tel mot-clé n'existe pas en C. Néanmoins, écrire `else` immédiatement suivi de `if`, sans ouvrir d'accolade après le `else` fournit l'effet attendu. Dans l'exemple suivant (où nous remarquons l'utilisation d'un type énuméré pour l'occasion) les deux fonctions de comparaison ont exactement le même comportement.

```c
enum compare_t { Equal, Lower, Greater } ;

enum compare_t compare (int x, int y) {
  if (x == y) return Equal ;
  else {
    if (x < y) return Lower ;
    else return Greater ;
  }
}

enum compare_t compare2 (int x, int y) {
```

```
  if (x == y) return Equal ;
  else if (x < y) return Lower ;
  else return Greater ;
}
```

Pour aller plus loin

Les plus attentifs des lecteurs auront remarqué que, chaque cas faisant directement un `return`, donc faisant quitter la fonction, il n'était pas nécessaire d'utiliser des `else`. En effet, soit une condition est vraie et l'on ne regarde pas les autres puisque l'on sort de la fonction. Soit elle est fausse et il faut de toute façon examiner la suivante. Donc oui, nous pourrions plus simplement écrire cette fonction sous la forme :

```
enum compare_t compare2 (int x, int y) {
  if (x == y) return Equal ;
  if (x < y) return Lower ;
  return Greater ;
}
```

5.1.3 `switch case`

Lorsque l'on souhaite comparer une expression à une suite de valeurs constantes mutuellement exclusives, une première solution consiste à enchaîner plusieurs `if` et `else` imbriqués. Grâce au comportement du `else if`, présenté dans la précédente section, nous obtenons alors une cascade de conditions de la forme :

```
if (x == 0) { ... }
else if (x == 1) { ... }
else if (x == 10) { ... }
else if (x == 3) { ... }
else { ... }
```

De manière plus concise, la construction conditionnelle généralisée permet d'énoncer des suites de cas constants, mutuellement exclusifs, chacun associé à un traitement. La forme générale de cette construction est :

switch ($expression$) {
  case $constante_1$:
    $instructions_{constante_1}$
  break ;
  case $constante_2$:
    $instructions_{constante_2}$
  break ;
  ...
  default :
    $instructions_{default}$
  break ;
}

Comme pour la construction `if`, l'expression de condition est mise entre parenthèses. Chaque clause `case` doit être suivie d'une constante se réduisant à un entier : un `int` (quelles que soient sa taille et sa capacité à être signé) ou un `enum`. Un deux-point (i.e. `:`) termine l'énoncé de chaque cas.

L'expression sur laquelle le `switch` discrimine est évalue et comparée « à chacun » des cas et un saut est effectué au cas dont la constante correspond (égalité). Les instructions suivant le `case` sélectionné sont alors exécutées. L'instruction `break` permet de terminer le cas et de sortir du `switch`. Si aucun `case` ne correspond à la valeur de l'expression, alors le cas `default` est sélectionné s'il existe. S'il n'existe pas, aucun traitement n'est effectué et le `switch` n'a aucun effet.

L'instruction `break` termine explicitement la séquence des instructions d'un `case` et provoque la sortie du `switch`. Si aucun `break` n'est présent entre deux `case`, leurs traitements s'enchaînent. Dans le programme suivant, le cas 3 a sa propre instruction mais il continue avec celles du cas 4. Les cas 9 et 7 ont exactement le même traitement.

Le `break` terminant le dernier cas (il se trouve que c'est le `default` mais ce n'est pas une obligation) n'est pas utile. En effet, la fin du dernier cas débouche automatiquement sur la fin du `switch`. Nous l'avons mis par soucis d'homogénéité, voire d'esthétisme : il sera ignoré lors de la compilation.

```
void f (int x) {
  switch (1 + x - 1) {
  case 2:
    printf ("C'est 2.\n") ;
    break ;
  case 3:
    printf ("C'est 3.\n") ;
  case 4:       /* Yes fallthrough. */
    printf ("C'est 4 mais possiblement 3 aussi.\n") ;
    break ;
  case 9:       /* Yes fallthrough. */
  case 7:
    printf ("C'est 9 ou 7.\n") ;
    break ;
  default:
    printf ("C'est autrechose.\n") ;
    break ;   /* Inutile, on sort du switch, c'est sa fin. */
  }
}
```

`f (1)`	→	"C'est autrechose."
`f (2)`	→	"C'est 2."
`f (3)`	→	"C'est 3.↵C'est 4 mais possiblement 3 aussi."
`f (4)`	→	"C'est 4 mais possiblement 3 aussi."
`f (5)`	→	"C'est autrechose."
`f (7)`	→	"C'est 9 ou 7."
`f (8)`	→	"C'est autrechose."
`f (9)`	→	"C'est 9 ou 7."

Pour aller plus loin

L'enchaînement de `case` sans `break` (comme 3-4 et 9-7) est appelé en anglais "*fallthrough*". Les bonnes pratiques (et de manière indispensable dans le monde du logiciel critique) veulent qu'un commentaire accompagne un fallthrough de manière à rendre explicite la volonté de l'absence du `break`, donc de montrer qu'elle ne relève pas d'un oubli.

L'ordre des cas est quelconque, il n'a pas besoin de suivre l'ordre croissant des valeurs testées. En particulier, le cas `default` est usuellement positionné en dernier, mais il peut en réalité être à n'importe quelle position.

L'instruction `switch` se marrie fort bien avec les types énumérés. Puisque ces deniers sont définis par cas, la plupart du temps les traitements réalisés dessus le sont également. Si nous reprenons la définition du type `direction_t` donnée dans le chapitre 3.3 :

```
enum direction_t { nord, sud, est, ouest } ;
```

il est pratique de définir une fonction, par exemple d'impression, en énumérant chacun des cas :

```
void print_dir (enum direction_t dir) {
  switch (dir) {
  case nord: printf ("nord") ; break ;
  case sud: printf ("sud") ; break ;
  case est: printf ("est") ; break ;
  case ouest: printf ("ouest") ; break ;
  }
}
```

Pour aller plus loin

Notre fonction `print_dir` couvre *a priori* toutes les valeurs possibles du type `enum direction_t`, il n'y a donc pas de raison de lui ajouter un cas `default`.
Puisque les `enum` sont en fait des entiers, rien n'empêche d'appeler `print_dir` avec un entier quelconque, par exemple `27` (aucun message du compilateur). Dans ce cas `print_dir` ne fera rien, donc laissera passer ce cas d'erreur silencieusement.

Pour plus de robustesse et de manière indispensable dans le monde du logiciel critique, on peut toujours mettre un cas `default` qui servira à détecter et signaler une erreur lors de l'utilisation de la fonction, que ce soit dû à une erreur de programmation ou une corruption d'informations en mémoire.

5.2 Boucles

Certains processus algorithmiques nécessite de répéter (itérer) un traitement. Le langage C dispose de trois types de boucles : `while`, `for` et `do while`, les deux premières étant équivalentes.

5.2.1 Boucle `while`

Identiques à celle des autres langages, la boucle `while` a la forme générale :

`while` (*expression*) *instruction* ;

où l'on ajoute obligatoirement des accolades dans le cas où le corps de la boucle est constitué de plusieurs instructions :

while (*expression*) {
  instructions ;
}

Tant que l'expression condition s'évalue en une valeur différente de 0 (donc équivalente à « vrai »), le corps de la boucle est exécuté.

5.2.2 Boucle `for`

La boucle `for` est la boucle iconique du langage C. Elle est composée de 4 parties :

`for` (*expression*$_1$; *expression*$_2$; *expression*$_3$) *instruction* ;

où l'on ajoute obligatoirement des accolades dans le cas où le corps de la boucle est constitué de plusieurs instructions :

for (*expression*$_1$; *expression*$_2$; *expression*$_3$) {
  instructions ;
}

et sa sémantique est en fait celle d'une boucle `while` équivalente qui aurait la forme :

```
expression1 ;
while (expression2) {
  instructions ;
  expression3 ;
}
```

Nous pouvons donc préciser le rôle de chaque élément de cette construction de la manière suivante :

1. $expression_1$: l'initialisation,
2. $expression_2$: condition de boucle,
3. $instructions$: corps de la boucle,
4. $expression_3$: post-traitement du corps.

```c
#include <stdio.h>

int main () {
  int i ;
  for (i = 0; i < 10; i++) {
    printf ("%d ", i) ;
  }
  printf ("\n") ;
  return 0 ;
}
```

Le programme ci-dessus boucle donc tant que `i < 10`, avec `i` initialisé à `0` et incrémenté à chaque fin de tour de boucle. L'affichage est donc `0 1 2 3 4 5 6 7 8 9` suivi d'un retour à la ligne final. C'est donc la forme typique d'une boucle qui itère « un certain nombre de fois ».

Cependant, la condition est totalement quelconque, pas forcément le test d'une borne d'arrêt. Nous pouvons donc implanter n'importe quelle forme de boucle avec un `for` qui peut l'être avec un `while`. C'est normal, nous avons montré l'équivalence entre les deux.

Important

Le séparateur entre les éléments entre parenthèses est un point-virgule. En particulier, ce n'est pas une virgule (qui a un autre rôle abordé dans la section d'approfondissement 16.8).

Pour aller plus loin

Vous avez peut-être remarqué que l'initialisation et le post-traitement ont été présentés comme des expressions. Pourtant, nous avons utilisé `i = 0` et `i++` qui sont des instructions.

Il convient de se rappeler (comme expliqué dans les sections 4.2.1 et 4.2.3) que les affectations sont présentées comme des instructions, mais sont en fait aussi des expressions car elles ont une valeur. C'est cette « double personnalité » qui leur permet de servir d'initialisation et de post-traitement de boucle.

Dans cette forme idiomatique de la boucle `for`, un « compteur » est utilisé (ici `i`) pour déterminer le « nombre de tours » de la boucle. Il est possible de le déclarer directement dans l'élément d'initialisation. Ainsi, il n'existera que dans la boucle.

```c
#include <stdio.h>

int main () {
  for (int i = 0; i < 10; i++) {
    printf ("%d ", i) ;
  }
```

→

```
    printf ("\n") ;
    for (int i = 0; i < 10; i += 2) {
      printf ("%d ", i) ;
    }
    printf ("\n") ;
    return 0 ;
  }
```

Dans les deux boucle, le même nom de variable `i` est utilisé, mais il s'agit de variables différentes. Nous pouvons ainsi réutiliser le même nom de variable à plusieurs endroits (`i` est le nom typique d'un compteur). Cela permet également d'éviter de définir tout un tas de variables locales en début de fonction, qui seront utilisées bien plus loin dans son corps. La localité et la proximité de définition / utilisation rendent le code source plus lisible.

Il est possible d'omettre tout ou partie des 3 expressions situées entre les parenthèses. Les éléments correspondants sont alors inexistants. Poussées à l'extrême, ces omissions aboutissent à des boucles `for` dégénérées qui pourraient être avantageusement remplacées par le `while` qu'elles représentent. Les 4 boucles ci-dessous sont toutes équivalentes, mais il est légitime de se demander pourquoi convoquer un `for` dans les 3 dernières.

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

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

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

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

Pour aller plus loin

On voit parfois la forme `for (;;)` pour définir une boucle infinie. Cette forme idiomatique était préconisée il y a fort longtemps, lorsque les compilateurs étaient trop rudimentaires pour supprimer le test toujours vrai que l'écriture `while (1)` aurait introduit. Cette époque est révolue depuis longtemps, mais la forme du « `for` infini » est restée dans les mœurs.

5.2.3 Boucle `do while`

Contrairement aux 2 précédentes boucles, la boucle `do while` exécute toujours au moins une fois son corps puisque le test est relégué après ce corps. Il s'agit donc d'un motif d'itération réellement différent. Cette boucle a la forme générale :

`do` *instruction* `;` `while (`*expression*`) ;`

où l'on ajoute obligatoirement des accolades dans le cas où le corps de la boucle est constitué de plusieurs instructions :

```
do {
  instructions ;
} while (expression) ;
```

Important

Il faut noter la présence d'un point-virgule après la condition, ce qui en fait la seule construction de boucle suivie par un point-virgule.

Un des usages typiques est la récupération d'une information depuis une saisie de l'utilisateur au terminal (cf. fonction `scanf` présentée dans le chapitre 13) en redemandant tant que la saisie ne satisfait pas un certain critère.

```
int c ;
...
do {
  printf ("Entrez un chiffe: ") ; /* Lecture ... */
  scanf (" %c", &c) ;
} while ((c < '0') || (c > '9')) ; /* ... tant que pas valide. */
...
```

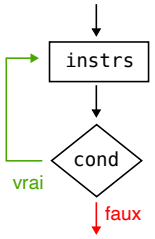
La condition de sortie de la boucle s'assure que le `char` saisi se trouve bien entre les deux caractères `'0'` et `'9'` inclus. Dans le cas contraire, la boucle recommence.

Notons que les caractères sont ordonnés (comme tous entiers puisque ce ne sont que des entiers sur 8 bits – cf. section 2.1.3) de manière « pratique ». Les valeurs entières représentant les « lettres » de '0' à '9' sont consécutives, en ordre croissant.

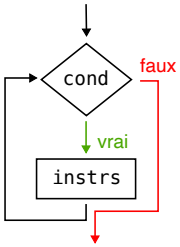
5.2.4 Récapitulatif des formes de boucles

Selon la sagesse populaire, petit schéma valant mieux qu'un long discours...

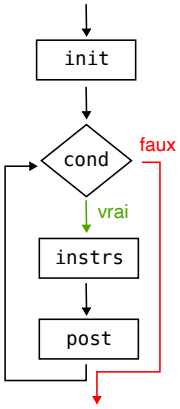
`do { instrs ; } while (cond) ;`



`while (cond) { instrs ; }`



`for (init; cond; post) { instrs ; }`



5.3 Ruptures de séquence

Important

Les deux instructions de ce chapitre sont à utiliser avec une extrême parcimonie tant elles peuvent mener à des boucles difficiles à comprendre et à maintenir.
Aussi, elles sont présentées dans ce document à titre d'exhaustivité, mais leur abus est particulièrement déconseillé (à l'exception du `break` dans un `switch` comme présenté en section 5.1.3).

Les deux instructions décrites dans cette section ne sont pas des boucles. Cependant leur utilisation est intimement liée à celle des boucles, c'est pourquoi nous les présentons dans ce chapitre.

5.3.1 `break`

L'instruction `break` a déjà été présentée en section 5.1.3 comme permettant de terminer un cas de `switch`. Son effet était plus précisément de permettre de sortir du `switch` immédiatement englobant.

Elle permet également de sortir de la boucle immédiatement englobante. Autrement dit, elle permet de sortir d'une boucle de manière anticipée, sans attendre que la condition soit devenue fausse.

Pourquoi avons-nous dit en introduction de cette section que cette instruction devait être utilisée avec sagesse ?

```
for (int i = 0; i < 10; i++) {
  if (i == 7) break ;
  printf ("%d\n", i) ;
}
```

La boucle ci-dessus est-elle réellement raisonnable ? La réponse est non, la condition de sortie ne sert à rien, sauf à rendre le code confus. En effet, la boucle va tourner tant que `i < 10` ... sauf que, arrivée à $i = 7$, elle va terminer.

De surcroît, comme le `if break` est placé avant l'affichage, la valeur 7 en sera pas affichée. C'est donc en fait une boucle `for (int i = 0; i < 7; i++)`. Si le `if break` avait été placé après le `printf`, nous aurions eu une boucle `for (int i = 0; i < 8; i++)`. Mais attendons l'instruction `continue` ci-dessous pour enfoncer le clou...

5.3.2 `continue`

L'instruction `continue` permet d'interrompre l'exécution de l'itération courante du corps de la boucle, pour sauter directement à l'évaluation de la condition. Autrement dit, elle permet de sauter le reste de la boucle pour recommencer (éventuellement) une itération.

Améliorons notre précédent exemple pour illustrer ce qu'une utilisation déraisonnable permet d'écrire comme code.

```
for (int i = 0; i < 10; i++) {
  if (i == 3) continue ;
  if (i == 7) break ;
  printf ("%d\n", i) ;
}
```

L'exemple restant simple, nous voyons qu'en plus de s'arrêter quand `i` atteint 7, la boucle « saute » l'affichage de la valeur 3. Nous pouvons convenir qu'une manière plus lisible, plus maintenable, plus raisonnable d'écrire cette fonction eût été :

```
for (int i = 0; i < 7; i++) {
  if (i != 3) printf ("%d\n", i) ;
}
```

Bien souvent, il est tentant de corriger une « petite » erreur sur la condition de boucle, que l'on a identifiée lors d'un test, avec une petite « exception » à la condition de sortie. Puis une autre, puis encore une autre... Si nous y mettons un peu de bonne volonté, nous pouvons, par petites entorses successives, arriver à du code « intéressant » comme le suivant.

```
int sum_something (int a, int b) {
  int sum = 0, i = a, last = 0, skip_next = 0 ;

  while (1) {
    if (i > b) break ;
    if (i == a) { i++ ; continue ; }
```
→

```
      if (skip_next) { skip_next = 0 ; i++ ; continue ; }
      if (i % 2 == 0 && last % 2 == 0) { i++ ; continue ; }
      for (int k = 0; k < 2; k++) {
        if (k == 1 && i % 3 == 0) break ;
        sum += i ;
      }
      if (i % 5 == 0) { skip_next = 1 ; }
      if (sum > 500) break ;
      last = i ;
      i++ ;
    }
    return sum ;
  }
```

5.4 En résumé

- La conditionnelle simple a la forme :

```
if (expression) {
  instructions_t ;
}
else {
  instructions_f ;
}
```

- La partie `else` est optionnelle.
- L'expression de condition est toujours entre parenthèses.
- Les accolades forment des blocs d'instructions à exécuter.
- Pour l'expression condition, $0 \equiv$ « faux », $+\,0 \equiv$ « vrai ».
- La conditionnelle généralisée a la forme :

```
switch (expression) {
  case constante_1 :
      instructions_constante1
  break ;
  case constante_2 :
      instructions_constante2
  break ;
  ...
  default :
    instructions_default
  break ;
}
```

- Chaque cas est une constante se réduisant à un entier.
- L'instruction `break` permet de séparer les cas.
- Sans `break` les cas s'enchaînent.
- Le cas `default` est optionnel.
- La boucle `while` a la forme :

```
while (expression) {
  instructions ;
}
```

- La boucle `for` a la forme :

```
for (expression_1 ; expression_2 ; expression_3) {
  instructions ;
}
```

- Le séparateur dans les parenthèses est un point-virgule, pas une virgule.
- La boucle `for` est en fait une boucle `while`.
- La boucle `do while` a la forme :

```
do {
  instructions ;
} while (expression) ;
```

- Elle exécute toujours au moins une fois son corps au début.
- Les ruptures de séquence `break` (hors `switch`) et `continue` doivent être utilisées de manière très parcimonieuse et raisonnable.
- `break` permet de sortir de la boucle ou du `switch` englobant.
- `continue` permet d'interrompre l'itération courante de la boucle englobante pour sauter à son expression de condition.

Chapitre 6

Affichage avec `printf`

L'affichage de messages est une fonctionnalité incontournable de n'importe quel langage de programmation. Au travers de sa bibliothèque standard, C met à disposition la fonction `printf` à cet effet. Cette fonction a déjà été aperçue dans les précédents chapitres, dès le tout premier programme "Hello world" du chapitre introductif. Le présent chapitre a pour objectif de détailler les possibilités offertes par cette fonction.

Important

L'utilisation de `printf` est conditionnée à l'inclusion du fichier d'en-tête `<stdio.h>` afin que celle-ci soit déclarée.

6.1 Messages constants

Dans sa forme la plus simple, `printf` prend en argument une chaîne de caractères qui représente un texte fixe. C'était le cas dans le programme "*Hello World*" de la section 1.3.

```
printf ("Ceci est un message fixe.\n") ;
```

Important

Quelle que soit la forme du message que l'on veut afficher, le `\n`, en plus d'afficher un retour à la ligne, force l'affichage effectif.
En effet, pour des questions d'efficacité, `printf` se permet de mémoriser le texte à afficher dans une mémoire tampon interne et attend d'avoir « suffisamment de texte » en suspens pour déclencher l'affichage effectif. Néanmoins, la présence d'un `\n` dans un message interrompt cette mémorisation et émet, dans le canal de sortie, tout ce qui avait été mis en suspens.

Ce trait de fonctionnement est important dans le cas ou l'affichage est utilisé pour débugger un programme qui plante en étant interrompu de manière abrupte (généralement par un *Segmentation Fault*, *Bus error*, *Division by 0*, etc.). Si les messages de trace ne comportent pas de `\n` et le programme est tué, alors le texte de trace ne sera pas visible bien que les instructions `printf` associées aient été exécutées.
Ainsi, nous croirons que le programme n'est pas passé par les points du code que nous surveillons et notre compréhension de ce qui se passe vraiment sera faussée.

Le programme suivant est une caricature de programme faux et va planter sans aucun doute. C'est l'instruction `*p = 496` qui va engendrer une erreur fatale (cf. chapitre 7 pour comprendre pourquoi). Le message qui la précède ne comporte pas de `\n`.

→

```c
#include <stdio.h>

int main () {
  int *p = NULL ;
  printf ("Ca va planter !") ;
  *p = 496 ;
  return 0 ;
}
```

```
$ ./trace.x
Segmentation fault: 11
```

Aucun message n'apparaît à l'exécution, qui elle, se termine effectivement plutôt mal. Si nous remplaçons simplement la ligne d'affichage par :

```
printf ("Ca va planter !\n") ;
```

donc en rajoutant juste un `\n` final, alors l'affichage a bien lieu avant l'erreur de segmentation, ce qui nous indique visiblement que nous somme bien passés par ce `printf`.

```
$ gcc -Wall trace.c -o trace.x
$ ./trace.x
Ca va planter !
Segmentation fault: 11
```

6.2 Messages formatés

Il est possible d'insérer des parties variables dans les messages afin qu'ils contiennent l'écriture textuelle des valeurs d'expressions. La manière dont la valeur d'une expression doit être interprétée pour l'afficher est dictée par une séquence commençant par % insérée dans la chaîne du message qu'est le premier argument de `printf`. Cette chaîne est appelée un *format*.

Par exemple, pour afficher un message de la forme « *Il est* hh *heure(s) et* mm *minute(s).* » où hh et mm sont les écritures décimales d'expressions entières, le format devra avoir la forme :

```
"Il est %d heure(s) et %d minute(s)."
```

Le format est donc plus qu'une simple chaîne, c'est un schéma, une famille de messages dont la variabilité réside dans ce qui sera affiché en lieu et place de chaque séquence %.

En plus d'un format comportant n séquences %, la fonction `printf` prend n expressions qui seront interprétées selon le format qui leur correspond positionnellement. Pour compléter l'exemple des heures et minutes, si `h` et `m` sont des variables de type `int`, nous pourrions écrire l'instruction d'affichage :

```
printf ("Il est %d heure(s) et %d minute(s).", h + 1, m % 60) ;
```

La tableau suivant décrit les formats les plus courants avec le type des valeurs qu'ils permettent d'afficher.

%d	`int` en décimal
%ld	`long int` en décimal
%u	`unsigned int` en décimal
%lu	`long unsigned int` en décimal
%x	`int` en hexadécimal
%lx	`long int` en hexadécimal
%f	`float` ou `double`
(%lf	comme %f, le `l` n'a pas d'effet)
%e	`float` ou `double` en notation scientifique
%s	chaîne de caractères
%c	`char` comme une « lettre », pas comme entier
%p	pointeur comme un entier en hexadécimal
%%	le caractère %

Important

Les formats de `printf` sont similaires à ceux de la fonction duale `scanf` présentée dans le chapitre 13, qui permet de faire de la lecture de données. La différence importante est sur le format `%lf` qui est indispensable pour lire un `double` avec `scanf` alors qu'il se comporte comme `%f` avec `printf`.

```c
#include <stdio.h>

int main () {
  printf ("%d\n", 42) ;
  printf ("%u\n", 42) ;
  printf ("%x\n", 42) ;
  printf ("%f\n", 42.0) ;
  printf ("%e\n", 42.0) ;
  printf ("%c\n", 42) ;
  return (0) ;
}
```

```
42
42
2a
42.000000
4.200000e+01
*
```

La structure des formats et les séquences % sont plus élaborées que le tableau récapitulatif précédent. Il est par exemple possible de contrôler le nombre de décimales à afficher pour les réels, de faire de l'alignement en remplissant avec des espaces ou des 0...

```c
#include <stdio.h>

int main () {
  printf ("%.2f\n", 22.0 / 7.0) ;
  printf ("%.4f\n", 22.0 / 7.0) ;
  printf ("%07d\n", 42) ;
  printf ("% 7d\n", 42) ;
  return (0) ;
}
```

```
3.14
3.1429
0000042
     42
```

La description complète des formats de `printf` est disponible sur les systèmes Unix/Linux en consultant sa page de manuel, section 3 (i.e. section dédiée aux fonctions de la bibliothèque standard de C), par la commande :

```
man 3 printf
```

6.3 En résumé

- L'affichage dans le terminal est réalisé par la fonction `printf`.
- Elle requiert l'inclusion du fichier d'en-tête `<stdio.h>`.
- Le format est la chaîne qui représente la forme du message et est toujours le premier argument.
- Les formats les plus courants sont :
  - `%d` pour les entiers,
  - `%f` pour les réels,
  - `%s` pour les chaînes,
  - `%c` pour les caractères.
- Il y a autant d'expressions en arguments après le format qu'il y a de séquences % dans le format.
- Le `\n` final permet de forcer l'affichage immédiatement, (important pour les messages de trace / debug).

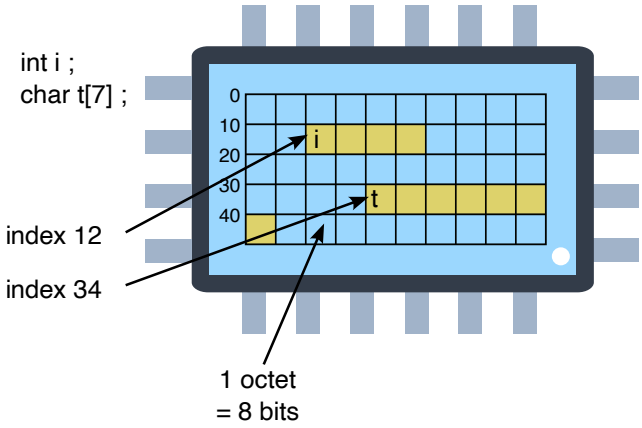
Chapitre 7

Pointeurs

La mémoire sert à stocker de l'information. Du point de vue d'un programme applicatif (donc, qui n'est pas le système d'exploitation), cette mémoire peut être vue comme un grand tableau d'octets, donc de paquets de 8 bits.

7.1 Adresse

Toutes les variables d'un programme, quel que soit leur type (entier, réel, tableau...) sont stockées dans la mémoire, donc à un certain « endroit ». Si la mémoire est vue comme un grand tableau, alors un « endroit » est une sorte d'index dans ce grand tableau, donc en première approximation un entier. Un tel index, donc un tel « endroit » est appelé une *adresse*. Une adresse désigne donc un endroit ponctuel dans la mémoire où se trouve une valeur ou à partir duquel se situent plusieurs valeurs consécutives (comme dans le cas des tableaux).



Chaque programme a le droit d'accéder à une partie de ce grand tableau, donc à une partie de la mémoire du système. Ses variables sont rangées dans cette zone. S'il tente d'accéder en dehors de la zone qui lui appartient, le système d'exploitation détectera la violation et l'interrompra immédiatement.

Important

Une adresse particulière nommée symboliquement `NULL` permet de représenter « l'adresse invalide ». Sur la majorité des systèmes, il s'agit de celle de valeur 0 ce qui permet de l'utiliser dans un `if` avec la même sémantique que « faux ». Cette valeur `NULL` est particulièrement importante dans le cadre de l'allocation dynamique (cf. chapitre 12 et plus particulièrement la section 12.2.1) et nécessite l'inclusion de `<stdlib.h>`.

7.2 Pointeur

Une adresse est également une information et à ce titre nous pouvons désirer mémoriser des adresses pour les manipuler ultérieurement. Osons quelques métaphores avec la vie courante à propos de manipulations d'adresses :

- noter l'adresse d'une réunion pour la transférer plus tard à des personnes non invitées initialement,
- noter l'adresse où remettre un document une fois qu'il sera rédigé,
- noter l'adresse où aller réceptionner un colis,
- noter une adresse pour parcourir tous les bâtiments de la rue consécutivement et y déposer un courrier municipal,
- à partir d'une adresse, calculer celle de la nième maison (bon, dans un monde simplifié où toutes les maisons feraient la même largeur).

Une variable destinée à contenir des adresses est appelée un *pointeur*. Comme pour toute variable, les deux opérations évidentes sont la lecture et l'écriture : nous pouvons écrire dans un pointeur (pour l'initialiser par exemple), nous pouvons lire dedans pour récupérer l'adresse qu'il contient.

De plus, puisque la valeur contenue dans un pointeur est une adresse, on peut également lire la valeur se trouvant à cette adresse et écrire une valeur à cette adresse.

Dans les deux cas, il est nécessaire de connaître la nature, donc le type de l'information à récupérer ou bien le type de ce que l'on peut déposer. Dans le premier cas c'est pour savoir comment interpréter la suite de bits que nous allons trouver à l'adresse contenue dans le pointeur, ainsi que le nombre d'octets qui devront y être récupérés. Dans le second cas, il faut être certain que l'adresse contenue dans le pointeur désigne un « endroit » où il y a exactement la place (i.e. le nombre d'octets) pour stocker l'information.

Par conséquent, une adresse doit être associée à un type, donc un pointeur doit refléter le type des informations vers lesquelles il « pointe ». Nous ne parlerons donc pas simplement de « type pointeur », mais de type « pointeur vers un type » : celui des informations dont il contient l'adresse. Ainsi, par exemple, nous pourrons parler de :

- pointeur vers des `int` (qui contient l'adresse où se trouve un `int`),
- pointeur vers des `double` (qui contient l'adresse où se trouve un `double`),
- pointeur vers des pointeurs vers des `int` (qui contient l'adresse où se trouve une adresse d'`int`).

Important

Malheureusement, la langue naturelle est facétieuse et regorge d'ambiguïtés. On dit souvent qu'un pointeur (donc une variable contenant une adresse) a le type « pointeur ». On assimile donc le nom d'une catégorie de variables avec son type. C'est le contexte de la discussion qui permet de savoir si l'on parle de la variable ou de son type.

7.2.1 Déclaration de pointeur

La déclaration d'un pointeur, donc d'une variable destinée à mémoriser des adresses, doit spécifier deux informations (en plus du nom de la variable bien entendu) :

- que c'est un pointeur,
- vers quel type d'information (i.e. de valeur).

L'information « être un pointeur » est indiquée par une étoile devant le nom de la variable déclarée. Vers quoi elle pointe est indiqué en préfixant par ce type. Ainsi, pour définir un pointeur `p` devant stocker des adresses d'entiers, nous définirons :

```
int *p ;
```

Cette manière d'expliquer la définition d'un pointeur revient à dire « `p` *est une variable pointeur, vers un* `int` ».

Nous pouvons exactement signifier la même chose en disant que pour définir un pointeur `p`, on préfixe le nom de cette variable par le type des adresses qu'elle peut contenir suivi par une étoile. L'écriture de la déclaration serait donc plutôt :

```
int* p ;
```

Cette manière d'expliquer la définition d'un pointeur revient à dire `p` est une variable, qui contient un `int*`.

Important

Ces deux manières d'écrire sont totalement identiques et signifient la même chose. C'est uniquement une interprétation personnelle qui se retrouve dans la manière dont les programmeuses et programmeurs conçoivent la notion de pointeur. Il n'y a pas de meilleure interprétation et c'est heureux puisque les deux représentent exactement le même concept. La place des espaces n'a aucune importance.

- `int *p ;` pointeur vers des entiers.
- `long int *p ;` pointeur vers des entiers longs.
- `short unsigned int *p ;` pointeur vers des entiers courts non signés.
- `float **p ;` pointeur vers des pointeurs qui eux-mêmes pointent vers des réels.
- `enum direction_t *p ;` pointeur vers des valeurs du type énuméré `direction_t` (cf. section 3.3).
- `double **********p ;` pointeur vers pointeur vers . . . oui, c'est possible.

7.2.2 Initialisation de pointeur

Comme toute variable, un pointeur doit être initialisé avant que l'on puisse manipuler son contenu, en particulier aller lire ou écrire à l'adresse qu'il contient. Sans initialisation, comme toute variable, il contiendra une valeurs indéfinie, donc désignera un « endroit » quelconque en mémoire. Et comme précisé en section 7.1, un programme n'a le droit d'accéder qu'à une partie de la mémoire du système, le reste lui étant interdit sous peine d'interruption immédiate par le système d'exploitation.

Important

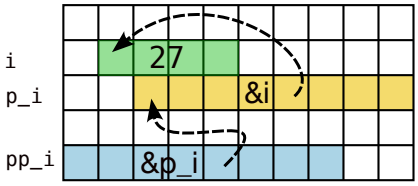
Dit autrement, si nous ne faisons que déclarer un pointeur, cela ne signifie pas « qu'au bout du pointeur », nous avons une zone mémoire où nous avons le droit de lire ou d'écrire. Lecture et écriture via des pointeurs non initialisés sont les raisons les plus courantes de *Segmentation Fault* et *Bus error*.

La première façon de faire pour initialiser un pointeur est de récupérer l'adresse d'une variable de notre programme. L'autre façon est grâce à l'allocation dynamique de mémoire qui est décrite dans le chapitre 12. Extraire l'adresse d'une variable `i` existante est réalisé avec l'opérateur `&`.

```
int i = 27 ;
int *p_i = &i ;
int **pp_i = &p_i ;
```

L'expression `&i` nous permet de récupérer l'adresse de la variable `i`, que nous pouvons stocker dans la variable `p_i` qui est de type pointeur sur `int`. Les types sont bien respectés puisque `p_i` peut mémoriser des adresses d'`int` et la variable `i` est bien de type `int`.

Comme esquissé en section 7.2.1, il est possible de créer un pointeur sur pointeur, donc une variable qui contient l'adresse d'un autre pointeur. Puisque `p_i` est un pointeur sur `int`, donc un `int*`, nous pouvons récupérer sa propre adresse avec l'opérateur `&`. C'est donc une adresse où il y a un `int*`, donc son type est `int**`. Nous pouvons donc la mémoriser en déclarant le pointeur `pp_i` qui est bien de type `int**`.



Nous avons uniquement donné des exemples qui obtiennent l'adresse de variables. Dans la réalité, il est possible d'obtenir l'adresse de toute entité syntaxique qui fait référence à une zone où des informations sont stockées. Il est ainsi possible d'obtenir l'adresse d'une case particulière d'un tableau (cf. chapitre 9) par `&t[2]` ou celle d'un membre particulier d'une structure (cf. 14) par `&point.abscisse`.

7.2.3 Déréférencement de pointeur

Accéder à la valeur qui se trouve à une adresse (que ce soit pour la lire ou pour la remplacer) est réalisé grâce à l'opérateur `*`. Cette opération est nommée *déréférencer* un pointeur.

Important

Nous parlons ici de l'opérateur `*` qui est utilisé dans le contexte d'une expression. Il est différent du `*` utilisé dans le contexte d'un type pour signifier « type pointeur sur ». La proximité syntaxique cache (malheureusement) une différence fondamentale (sans parler du `*` utilisé comme opérateur de multiplication, qui se distingue par les types de ses opérandes qui doivent être numériques).

7.2.3.1 Lecture à une adresse

Utiliser une expression `*p` en dehors de la partie gauche d'une affectation effectue une lecture de la valeur située à l'adresse contenue dans le pointeur (la variable) `p`.

```c
#include <stdio.h>

int main () {
  int i = 27 ;
  int *p_i = &i ;
  int **pp_i = &p_i ;
  printf ("Valeur à l'adresse de p: %d\n", *p_i) ;
  printf ("Valeur de p: %p\n", p_i) ;
  return 0 ;
}
```

```
$ gcc -Wall ptr.c -o ptr.x
$ ./ptr.x
Valeur à l'adresse de p: 27
Valeur de p: 0x16f812f58
```

Dans le programme ci-dessus, `*p_i` vaut la valeur 27 car `p_i` désigne l'adresse de `i` et dans `i`, donc à cette adresse, il y a la valeur 27. Il faut noter que cette valeur est affichée comme il se doit avec un `%d` puisque c'est un entier.

C'est totalement différent du second `printf` qui affiche la valeur de `p`, donc l'adresse qu'il contient (et non la valeur à l'adresse qu'il contient). Le format utilisé est alors `%p` qui permet d'afficher des adresses en notation hexadécimale, donc en base 16 (cf. section 6.2).

Quant à la variable `pp_i`, si nous la déréférençons avec une `*`, nous obtenons une adresse d'entier, celle justement de la variable `i`. Nous pouvons donc encore déréférencer cette adresse, avec une

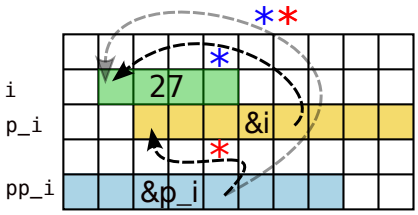
seconde `*`, pour aller lire la valeur qui s'y trouve. C'est donc la valeur qui se trouve dans `i`, comme l'illustre le programme suivant qui étend le précédent.

```c
#include <stdio.h>

int main () {
  int i = 27 ;
  int *p_i = &i ;
  int **pp_i = &p_i ;
  printf ("Valeur à l'adresse de p: %d\n", *p_i) ;
  printf ("Valeur de p: %p\n", p_i) ;
  printf ("Valeur de pp_i: %p\n", *pp_i) ;
  printf ("Valeur à l'adresse de pp_i: %p\n", *pp_i) ;
  printf ("Valeur en suivant 2 fois les adresses: %d\n", **pp_i) ;
  return 0 ;
}
```

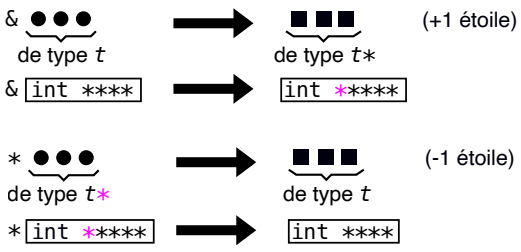
```
$ gcc -Wall ptr.c -o ptr.x
$ ./ptr.x
Valeur à l'adresse de p: 27
Valeur de p: 0x16faeaf58
Valeur de pp_i: 0x16faeaf58
Valeur à l'adresse de pp_i: 0x16faeaf58
Valeur en suivant 2 fois les adresses: 27
```

Si nous regardons globalement, `pp_i` était un double pointeur sur `int` : le déréférencer une fois nous donne un pointeur sur `int`, le déréférencer deux fois nous donne un `int`.



Nous pouvons remarquer au passage qu'entre les deux exécutions, la valeur de `p` (donc l'adresse qu'il contient, qui est celle de la variable `i`) a changé. En règle générale, les adresses ne sont pas des valeurs fixes d'une exécution à une autre.

La relation entre les opérateurs `*`, `&` et le type des valeurs obtenues est parfaitement mécanique. Le `&` rajoute un niveau de pointeur. Faire `&` sur une variable de type `t` nous donne un pointeur de type `t*`. Inversement, l'opérateur `*` enlève un niveau de pointeur. Faire `*` sur une expression de type `t*` nous donne une valeur de type `t`.



Même si techniquement le nombre de niveaux de pointeurs est illimité, dans la pratique il est rare de dépasser 2 ou 3 niveaux d'étoiles.

7.2.3.2 Écriture à une adresse

Pour modifier la valeur se trouvant à l'adresse désignée par un pointeur, il faut le déréférencer en partie gauche d'une affectation :

```
*p = ... ;
```

Le programme suivant déclare les mêmes variables que celui de la section 7.2.3.1 précédente : un entier `i`, un pointeur `p_i` qui mémorise l'adresse de `i` et enfin un pointeur `pp_i` qui stocke l'adresse de `p_i`. Le but est de rajouter des affectations au travers des différents pointeurs.

```
#include <stdio.h>

int main () {
  int i = 27 ;
  int *p_i = &i ;
  int **pp_i = &p_i ;
  printf ("(Init) i: %d, *p_i: %d\n", i, *p_i) ;
  *p_i = 72 ;
  printf ("(Par *p_i) i: %d, *p_i: %d\n", i, *p_i) ;
  **pp_i = 123 ;
  printf ("(Par **pp_i) i: %d, *p_i: %d, **pp_i: %d\n", i, *p_i, **pp_i) ;
  return 0 ;
}
```

L'affectation `*p_i = 72`, permet d'écrire la valeur 72 à l'adresse contenue dans `p_i`, donc à l'endroit où réside `i`. De ce fait, à l'issue de cette affectation, la variable `i` vaut 72. Dit autrement, puisque `p_i` contient l'adresse où se trouve `i`, alors aller écrire à l'adresse que contient `p_i` c'est aller écrire où se trouve `i`, donc dans `i`.

```
$ gcc -Wall ptr.c -o ptr.x
$ ./ptr.x
(Init) i: 27, *p_i: 27
(Par *p_i) i: 72, *p_i: 72
(Par **pp_i) i: 123, *p_i: 123, **pp_i: 123
```

Pour ce qui est du pointeur sur pointeur `pp_i`, l'écriture se comporte de la même manière au travers de deux niveaux d'indirection. Le pointeur `pp_i` contient l'adresse de `p_i`, donc `*pp_i` désigne `p_i` qui lui-même contient l'adresse de `i`. Lorsque nous faisons `**pp_i = 123`, nous demandons d'écrire 123 à l'adresse qui se trouve à l'adresse que contient `pp_i`. Dit autrement, avec la première étoile nous allons à l'endroit que désigne `pp_i`. Avec la seconde étoile, nous allons à l'adresse qui s'y trouve, c'est-à-dire à l'adresse de `i`.

En changeant l'adresse que contient un pointeur, il est possible de contrôler dans quelle variable une écriture au travers de ce pointeur agit. La fonction `sum_odd_even` ci-dessous exploite ce mécanisme pour calculer la somme des entiers pairs et celle des entiers impairs compris entre 0 et `n`.

```
void sum_odd_even (int n) {
  int sum_odd = 0 ;
  int sum_even = 0 ;
  int *where_to_add ;

  for (int i = 0; i <= n; i++) {
    if (i % 2 == 0) where_to_add = &sum_even ;
    else where_to_add = &sum_odd ;
    *where_to_add = *where_to_add + i ;
  }
  printf ("Somme pairs: %d, somme impairs: %d\n", sum_odd, sum_even) ;
```

```
}
```

Plutôt que de faire deux affectations distinctes dans les branches de la conditionnelle, la fonction fait pointer `where_to_add` vers la variable adéquate en fonction de la parité du compteur `i`. Ensuite, une unique affectation aveugle est effectuée à l'adresse contenue dans `where_to_add`.

7.3 En résumé

- Une variable sert à stocker des informations.
- Une variable réside en mémoire à un « certain endroit » : à une adresse.
- Si l'on voit la mémoire comme un grand tableau d'octets, une adresse est l'équivalent d'un indice dans la mémoire.
- Une adresse désigne un point dans la mémoire : localisation d'une donnée ou d'une zone contiguë de données.
- Une variable permettant de mémoriser une adresse est appelée un pointeur.
- Le type d'un pointeur indique le type de l'information se trouvant à l'adresse qu'il contient.
- On déclare un pointeur en donnant le type de donnée se trouvant à l'adresse qu'il désigne suivi de `*`, suivi son nom.
  ```
  unsigned long int *ptr ;
  ```
- Un pointeur doit désigner une zone appartenant à l'espace mémoire du programme, donc contenir une adresse dans cet espace.
- Comme toute variable, un pointeur doit être initialisé avant utilisation.
- On obtient l'adresse d'une variable avec l'opérateur `&`.
  ```
  int *ptr = &myvar ;
  ```
- Accéder à la valeur située à l'adresse contenue dans un pointeur est nommé déréférencer.
- Pour accéder à la valeur située à l'adresse contenue dans un pointeur on utilise l'opérateur `*`.
  ```
  printf ("%d\n", *ptr) ;
  ```
- Pour modifier la valeur contenue à l'adresse désignée par un pointeur il faut une affectation dont la partie gauche déréférence le pointeur.
  ```
  *ptr = 91 ;
  ```
- L'opérateur `&` ajoute un niveau d'étoile.
  `untype **i ;` → `&i` a pour type `untype***`.
- L'opérateur `*` retire un niveau d'étoile.
  `untype **i ;` → `*i` a pour type `untype*`.
- On peut d'obtenir l'adresse de toute entité syntaxique qui fait référence à une zone où des informations sont stockées (variable, case de tableau, membre d'une structure).

Chapitre 8

Appel de fonction et passage d'arguments

Appeler une fonction permet d'établir un lien entre deux mondes : celui de la fonction *appelante* et celui de la fonction *appelée*. Il y a donc un double mécanisme de communication à établir :

- d'une part la fonction appelante doit transmettre ses arguments à la fonction appelée,
- d'autre part, la fonction appelée doit retourner son résultat à la fonction appelante.

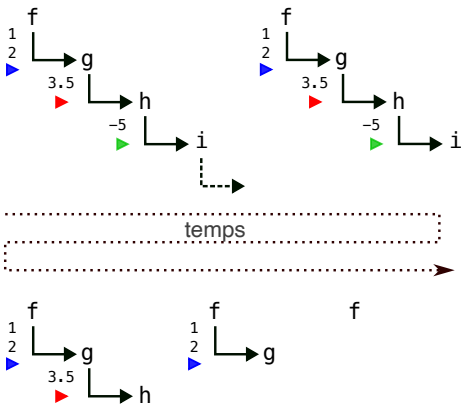
Lorsque l'exécution quitte temporairement le code d'une fonction pour en appeler une autre, il est également nécessaire de mémoriser à quel endroit cette exécution devra reprendre une fois la fonction appelée terminée. Cette information est appelée *adresse de retour*.

8.1 Pile d'exécution

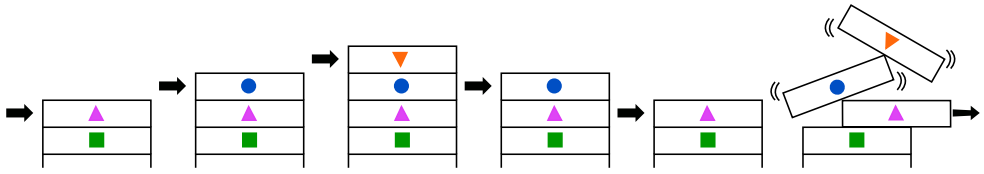
La durée de vie de ces informations est totalement dépendante de la structure des appels de fonctions. À chaque appel, les arguments et l'adresse de retour doivent être mémorisés. Ces informations sont pertinentes tant que la fonction appelée est en cours d'exécution. Dès lors qu'elle se termine, ces informations n'ont plus de raison d'être. Puisque les appels de fonctions peuvent s'imbriquer, nous en déduisons que ces informations s'empilent au cours des appels et qu'elles sont détruites en sens inverse lorsque nous sortons des appels imbriqués.

```
i (int t) {
  ... ;
}

h (float z) {
  i (-5)
▶... ;
}
g (int x, int y) {
  h (3.5) ;
▶...
}
f () {
  g (1, 2) ;
▶...
}
```



Toutes ces informations sont gérées par l'intermédiaire de ce qu'on appelle la *pile d'exécution*, un endroit connu des deux fonctions où elles peuvent se transmettre des informations. Il s'agit d'une zone de la mémoire du programme dont le fonctionnement logique est celui d'une pile : on ajoute au-dessus et on retire du dessus. Il est possible d'aller regarder au milieu de la pile, mais sans retirer. Un marqueur indique le haut actuel de la pile.



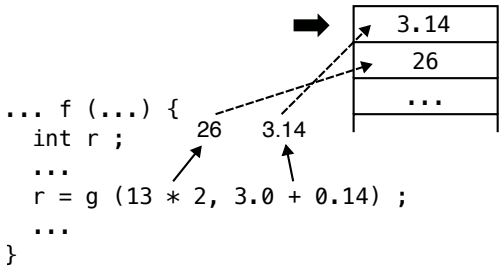
Nous allons examiner en détail (de manière toutefois quelque peu simplifiée) ce qui se passe lors d'un appel de fonction. Le programme ci-dessous est composé d'une fonction `f` qui appelle une fonction `g` en lui fournissant deux arguments, un entier et un réel.

Du côté de la fonction appelée `g`, ce qu'elle reçoit est matérialisé par les deux paramètres `i` et `f`. Nous pouvons les voir comme deux « variables prédéfinies » lorsque la fonction `g` débute. Du côté de la fonction appelante `f`, l'appel à `g` est effectué en fournissant autant d'expressions, nommées *arguments*, que `g` possède de *paramètres*.

```
int g (int i, float f) {
  int a, b ;
  ...
  return ... ;
}

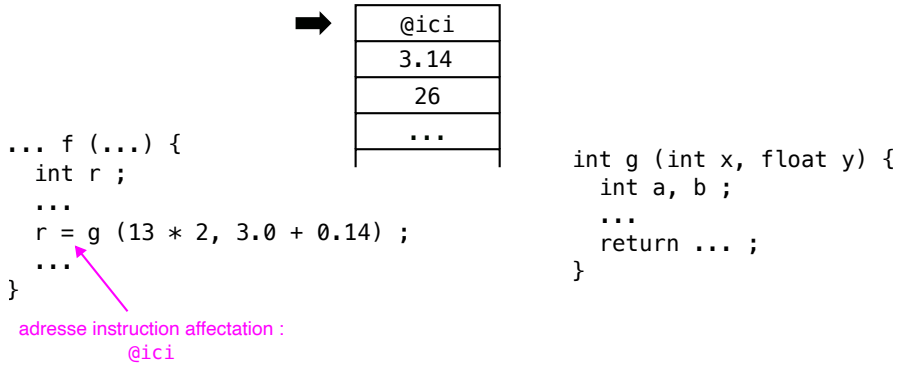

... f (...) {
  int r ;
  ...
  r = g (13 * 2, 3.0 + 0.14) ;
  ...
}
```

Lorsque `f` appelle `g`, elle calcule la valeur des deux expressions arguments, donc `13 * 2` et `3 + 0.14`. L'appel fait une **copie** de ces deux valeurs sur la pile. Dit autrement, il va écrire 26 et 3.14 dans la pile aux prochaines cases libres.



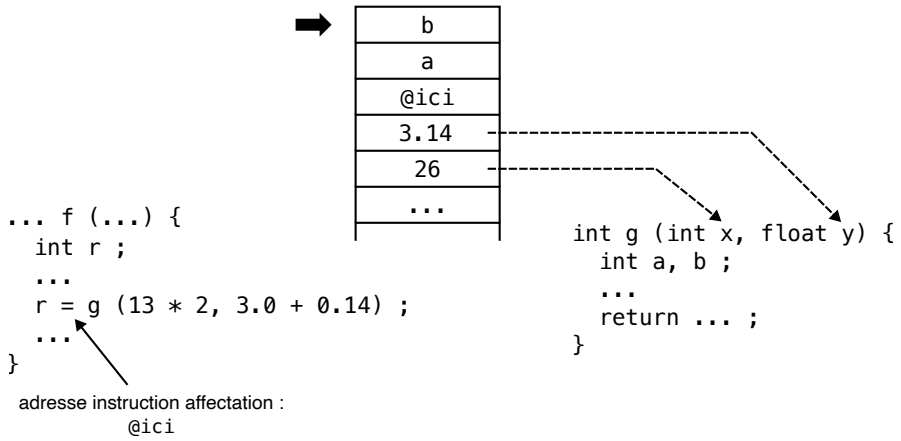

```
int g (int x, float y) {
  int a, b ;
  ...
  return ... ;
}
```

Ensuite, l'adresse de l'instruction située juste après l'appel, donc l'affectation du résultat dans `r`, est copiée sur la pile. Cette copie est effectuée automatiquement par l'instruction du micro-processeur qui provoque un appel de fonction.

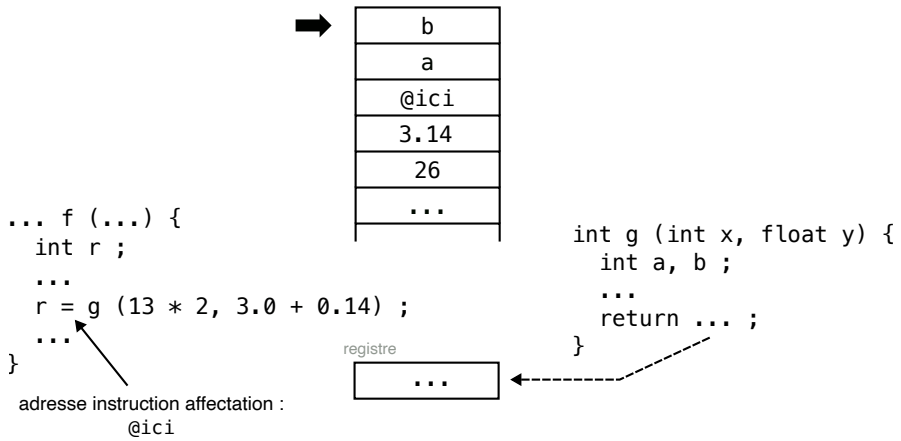


L'exécution continue désormais dans le code de la fonction `g`. Elle va pouvoir aller récupérer les valeurs de ses paramètres `x` et `y` dans la pile : elle sait qu'il y en a deux, elle connaît leurs types donc leurs tailles et donc elle sait où ils se trouvent.

Notons au passage que les variables locales `a` et `b` de la fonction `g` sont également allouées sur la pile. Cette mécanique est sensée puisque ces variables n'ont d'existence que durant l'exécution de la fonction : elles peuvent et doivent être « détruites » lorsque la fonction se termine.



Une fois que la fonction `g` a terminé son calcul, grâce à l'instruction `return`, elle va stocker sa valeur résultat à un endroit particulier connu de `f` (qui est généralement un *registre* spécifique du micro-processeur).

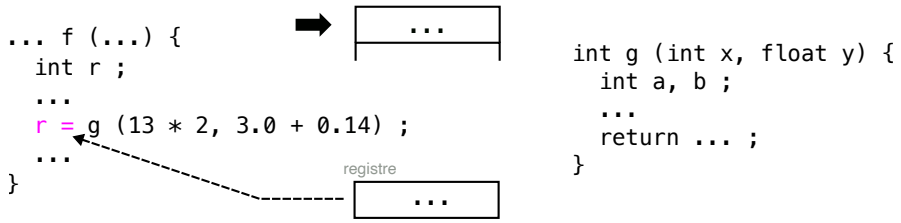


Une instruction de retour spécifique du micro-processeur va aller chercher dans la pile l'adresse où retourner pour continuer l'exécution de `f`. Au moment de ce retour, tout ce qui avait été mis sur la pile

pour l'appel à `g` devient inutile, donc le marqueur de haut de la pile redescend et l'espace au-dessus dans la pile redevient libre pour de futurs appels de fonctions.

Important

Cet espace n'est pas « physiquement » détruit, la nouvelle position du marqueur indique seulement que cet espace n'est plus occupé. En particulier, les valeurs qui avaient été déposée dedans restent en place jusqu'à temps qu'un prochain appel de fonction vienne les écraser. Cette remarque prendra toute son importance lorsque nous présenterons la nécessité de l'allocation dynamique en chapitre 12 (et plus particulièrement dans la section 12.1.2).



Important

Tout ce qui est transmis par une fonction appelante est une copie de l'information qu'elle veut transmettre à la fonction appelée, donc une copie de la valeur qu'elle veut transmettre.

8.2 Passage d'argument par adresse

8.2.1 Le problème

La limitation de l'instruction `return` réside dans le fait qu'elle ne permet de retourner qu'une seule valeur. Pourtant, il est aisé d'imaginer des fonctions qui ont besoin de retourner plusieurs valeurs, par exemple pour calculer à la fois le minimum et le maximum des éléments d'un tableau d'entiers.

Une première solution est de choisir de retourner un tableau (avec les précautions indispensables dont il est question dans le chapitre 12 et surtout la section 12.1.2) de deux cases puisque les deux valeurs à retourner sont des entiers. Cette solution n'est pas satisfaisante pour au moins trois raisons.

- Elle nécessite l'utilisation d'allocation dynamique qui peut-être techniquement subtile (cf. chapitre 12).
- Elle n'empêche pas structurellement à la fonction qui récupère le tableau d'aller accéder à des cases autres de 0 et 1. Une erreur de programmation est donc facile et sera silencieuse.
- Cette méthode n'est pas généralisable si les différentes valeurs à retourner sont de types différents.

Une seconde solution consiste à retourner une structure (cf. chapitre 14) comportant autant de champs qu'il y a de valeurs à retourner. Cette solution élimine les trois inconvénients ci-dessus, mais en présente un nouveau. Il faut définir autant de types de structures qu'il y a de combinaisons de valeurs à retourner (2 entiers, 1 entier et un booléen, 1 entier et une chaîne de caractères, 1 entier et 3 réels, etc.).

8.2.2 Paramètre en sortie

Plutôt que d'essayer de transmettre un résultat avec un `return`, pourquoi ne pas dire à la fonction appelée à quel endroit elle devra déposer, donc écrire, son résultat ?

Osons de nouveau une métaphore : plutôt que d'attendre que le livreur nous remette un paquet en main propre, nous disons au livreur à quel endroit il devra le déposer. Et cet endroit, nous le connaissons et il est dans une zone à laquelle nous avons le droit d'accéder. En terme mémoire, cet endroit est donc une adresse.

Considérons une fonction `f` qui appelle une fonction `g`. Cette dernière a besoin de « retourner » un entier et un réel. Nous pouvons déjà choisir de retourner l'entier ou le réel par le `return` « normal ». Disons que la fonction retourne l'entier.

```
... f (...) {
  int r ;
  ...
  r = g (13 * 2, 3.0 + 0.14) ;
  ...
}
```

```
int g (int x, float y) {
  ...
  return 100 + x - ... ;
}
```

Il reste donc la valeur réelle à communiquer à la fonction `f` qui appelle `g`. Nous ajoutons alors un paramètre à `g`, nommé `fres`, qui est une adresse de réel, donc un `float*`. Lorsque `g` devra « retourner » son résultat réel, elle ira l'écrire aveuglément à endroit indiqué par `fres`, donc là où nous lui aurons dit en l'appelant.

```
int g (int x, float y, float *fres)
    {
  ...
  *fres = y + 2.5 - ... ;
  return 100 + x - ... ;
}
```

```
f (...) {
  int r1 ;
  float r2 ;
  ...
  r1 =
    g (13 * 2, 3.0 + 0.14, &r2) ;
  ...
}
```

Du côté de `f` qui appelle `g`, il faut bien sûr fournir les arguments pour les paramètres « normaux » `x` et `y`, mais également l'adresse d'un endroit où `g` pourra écrire et donc où `f` pourra récupérer la valeur. Il suffit à `f` d'avoir une variable locale (donc à laquelle elle peut accéder) et de donner à `g` l'adresse de cette variable. Nous ajoutons donc une variable `r2` de type `float` et donnons son adresse à `g` lors de l'appel avec `&r2`. De cette manière, `g` écrira à l'endroit où se trouve `r2`, donc écrira dans `r2` (sans le savoir).

Ce mécanisme est appelé *passage d'argument par adresse*. Il se généralise à un nombre et types arbitraires de valeurs à « retourner » d'une fonction.

8.2.3 Paramètre en entrée/sortie

Le mécanisme de passage par adresse est utile pour indiquer à une fonction appelée là où elle doit stocker un de ses résultats, en utilisant un pointeur. Cependant, nous pouvons également nous servir de ce pointeur pour recevoir des informations, comme avec un paramètre « habituel ». La fonction appelante peut avoir initialisé sa variable passée par adresse avec une valeur que l'appelée pourra utiliser avant de l'écraser.

Soit une fonction qui reçoit un point dans le plan et multiplie ses coordonnées par un réel. Cette fonction peut effectuer ce calcul en modifiant directement les coordonnées du point qu'elle reçoit sous forme de deux pointeurs.

```
void scale (float *x, float *y, float s) {
  *x = *x * s ;
  *y = *y * s ;
}
```

Elle récupère d'abord la valeur de chaque coordonnée en lisant ce qui se trouve à l'adresse correspondante, puis elle écrase cette valeur en y écrivant le résultat du calcul. Ainsi, la fonction `scale` n'a même plus besoin d'utiliser d'instruction `return` : tout est effectué au travers des paramètres passés par adresse. Son type de retour est donc `void`.

Il faut toutefois être conscient que cette fonction modifie directement les variables dont la fonction appelante aura donné les adresses. Si cette dernière devait réutiliser les valeurs présentes avant l'appel, il serait à sa charge d'en faire une sauvegarde.

8.2.4 Copie versus adresse ?

Au premier abord, le passage d'arguments par adresse peut sembler contradictoire avec le fait que nous avons dit que C faisait toujours des copies des arguments dans la pile. En fait, pas du tout.

Important

C fait bien du passage par copie au niveau de la mécanique. Lorsque nous passons une adresse en argument, c'est l'adresse l'argument. Et c'est bien cette adresse qui est copiée dans la pile. C'est donc bien un passage par copie d'une adresse. Le point important est de comprendre ce que nous copions : la valeur d'une variable ou bien son adresse.
La mécanique interne de C est un passage par copie. Et nous, en tant que programmeuse et programmeur, nous nous en servons à haut niveau pour faire un passage « par valeur » ou « par adresse ».

8.2.5 Ménager la pile (1)

Pour aller plus loin

Cette section est destinée à étudier une autre utilisation du passage par adresse. Elle utilise la notion de structure abordée dans le chapitre 14. Il convient donc de connaître cette notion ou de lire ledit chapitre au préalable.

Nous avons dit que les arguments d'un appel de fonction étaient toujours copiés dans la pile. C'est le cas pour des entiers, des réels, mais aussi des structures. Or une structure, si elle comporte de nombreux champs (ou des champs de grande taille comme des tableaux statiques), peut occuper un espace mémoire conséquent.

De ce fait, la copie effectuée lors d'un passage de structure en argument peut consommer à la fois beaucoup de la place sur la pile et de temps d'exécution. Or il se trouve que la pile est un espace de taille relativement modeste comparée au reste de la mémoire (de l'ordre de quelques Ko à Mo, loin derrière les Go de mémoire vive de certaines machines modernes).

Afin de ménager la pile et le temps de copie, il est courant de passer l'adresse d'une structure à la place de la structure elle-même. Cette adresse est de taille fixe, celle d'un pointeur. Et tous les pointeurs ont la même taille, indépendamment de vers quoi ils pointent (64 bits sur les machines standard de nos jours). Il faut simplement que la fonction appelée utilise son paramètre au travers du pointeur, même si ce n'est que pour lire (et non modifier) les informations situées à l'adresse que contient ce pointeur.

```
struct verybig {
  int x ; int y ; int z ;
  int data[10000] ;
  ...
} ;

... g (struct verybig s) {
  ...
  s.x ... s.y ... s.z ...
}

... f (...) {
  struct verybig s ;
  ...
  g (s) ;
}
```



```
struct verybig_t {
  int x ; int y ; int z ;
  int data[10000] ;
  ...
} ;

... g (struct verybig_t *s) {
  ...
  s->x ... s->y ... s->z ...
}

... f (...) {
  struct verybig_t s ;
  ...
  g (&s) ;
}
```

8.2.6 Et pour les tableaux ?

Pour aller plus loin

Cette section nécessite de comprendre comment sont représentés les tableaux en C, tel que décrit dans le chapitre 9. Il convient donc de connaître cette notion ou de lire ledit chapitre au préalable.

Puisqu'un tableau est désigné par un pointeur qui indique l'adresse où il débute, lors d'un appel de fonction où nous passons un tableau, nous passons en fait déjà un pointeur. Il n'y a donc jamais de copie du contenu du tableau. Nous ne copions que l'adresse. C'est donc automatiquement un passage par adresse. C'est exactement ce qui a été décrit en section 8.2.4 pour expliquer que le mécanisme par copie n'est pas incompatible avec le passage par adresse.

8.3 En résumé

- Lors d'un appel de fonction, les arguments sont transmis dans la pile d'exécution par un mécanisme de copie.
- L'adresse de l'instruction où reprendre l'exécution à la sortie de la fonction appelée est également enregistrée dans la pile.
- Les variables locales d'une fonction sont également enregistrées dans la pile.
- Le passage d'arguments par adresse permet de « retourner » plus d'une seule valeur en contournant ou complétant un `return`.
- La fonction appelée retourne ses résultats par l'intermédiaire d'arguments supplémentaires qui sont des pointeurs.

  ```
  ... f (..., int *result) { ... ; *result = ... ; }
  ```
- La fonction appelante doit fournir autant d'adresses de variables (en utilisant l'opérateur `&`).
- Lors d'un passage par adresse, c'est l'adresse qui est copiée dans la pile.

Chapitre 9

Tableaux statiques

Un tableau est une structure de donnée permettant de regrouper plusieurs valeurs d'un même type de manière consécutive. Les éléments d'un tableau (donc les valeurs) sont donc stockés sous forme d'une séquence. Puisque tous les éléments sont du même type, ils ont la même taille. Ainsi, il est possible d'accéder à chaque élément du tableau en spécifiant sont indice, i.e. la « case » qu'il occupe.

Important

Contrairement à Python un tableau possède une taille définitivement fixée lors de sa définition. Il n'est pas possible de rajouter des « cases » au début ni à la fin d'un tableau.

Important

Très souvent en Python « tableaux » et «listes » sont des termes utilisés sans distinction. En terme algorithmique, ces deux notions sont fondamentalement différentes et les « tableaux » de Python ne sont ni de simples tableaux ni des listes.

Les tableaux « dynamiques », dont la taille n'est pas connue au moment de la compilation (et non dont la taille varie après création), feront l'objet du chapitre 12 traitant de l'allocation dynamique.

9.1 Définition de tableau

La définition d'un tableau statique, i.e. dont la taille est connue au moment de compilation, est composée de 4 ou 5 parties :

type–elems *nom*`[`*taille*`]` `;`
type–elems *nom*`[`*taille*`] = { ..., ..., ... } ;`

- Le type des éléments qu'il contient.
- Son nom.
- Sa taille entre crochets, une constante entière strictement supérieure à 0.
- De manière optionnelle, une initialisation qui est une énumération de valeurs entre accolades et séparées par des virgules.
- Un point-virgule final.

```
int t1[4] ;
unsigned short int t2[10] ;
float t3[2 + 1] = { 3.0, 1.0, 4.159 } ;
char t4[] = { 'a', 'b', 'c', 'd' } ;
```

Dans toutes ces définitions, la taille est une expression constante que le compilateur calcule au moment de la compilation. En particulier, `2 + 1` ne sera pas calculé à l'exécution du programme : le compilateur la remplace par 3. Il convient de bien remarquer qu'aucune des tailles n'est le résultat d'un calcul qui sera effectué à l'exécution, la valeur d'une variable, d'un appel de fonction, etc.

Important

Il arrive de voir des déclarations de tableaux de la forme `int t[n] ;` avec `n` une variable dont la valeur est calculée lors de l'exécution. La section 16.5 vous expliquera pourquoi vous ne devez pas utiliser ce mécanisme introduit par la norme C99 (et débattu dans les normes suivantes, c'est dire).

Pour aller plus loin

Lorsqu'un tableau est défini avec une taille n et initialisé avec $m < n$ valeurs, il contient les n premières valeurs énumérées, les $n - m$ cases suivantes sont initialisées avec la « valeur par défaut » du type de ses éléments. De manière un peu simplifiée, il s'agit du « 0 » de ce type, donc `0` pour les entiers et `0.0` pour les réels.

```c
#include <stdio.h>

int main () {
  float t1[5] = { 0.0, 1.5, 2.4 }
      ;
  for (int i = 0; i < 5; i++)
      printf ("%f ", t1[i]) ;
  printf ("\n") ;
  return 0 ;
}
```

```
$ gcc -Wall array.c -o array.x
$ ./array.x
1.500000 2.400000 0.000000 0.000000
```

9.2 Accès, indexation

Comme dans de nombreux langages, l'accès dans un tableau se fait en indiquant entre crochets l'expression qui représente l'indice (i.e. décalage) de la case lue ou écrite.

```
t[0]   t[i + 1]   t[guess_index (...)]
```

Les tableaux sont par défaut des structures de données mutables comme les variables scalaires en C. Il est possible de lire ou d'écrire les valeurs des cases du tableau.

Important

Les indices d'un tableau commencent à 0 et s'arrêtent donc à sa taille moins 1. Pour un tableau `int t[20] ;` les indices valides sont dans l'intervalle $[0, 19]$.

Important

La taille d'un tableau ne fait pas partie de sa structure de données. Il est à la charge du programme (et donc de son développeur) de savoir quelle est la taille du tableau qu'il manipule. Cette information devra donc être mémorisée et transmise à toute fonction qui doit manipuler le tableau.

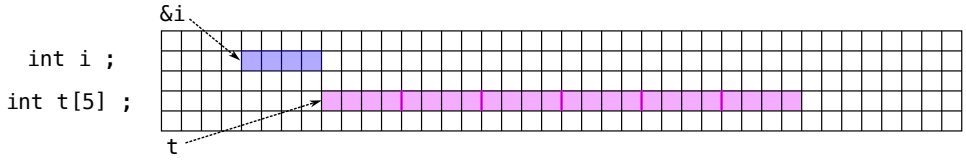
```c
#include <stdio.h>

int main () {
  int t[10] = { 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 } ;
  for (int i = 0; i < 10 / 2; i++) t[i] = t[10 - i - 1] ;
  for (int i = 0; i < 10; i++) printf ("%d ", t[i]) ;
  printf ("\n") ;
  return 0 ;
}
```

Lorsque nous définissons un tableau, le nom de celui-ci correspond en fait à l'adresse où ce tableau débute. Ainsi, avec la définition :

```c
int t[3] ;
```

le symbole `t` a pour type `int*`. Il a donc un type de pointeur. En effet, une adresse représente un « endroit » en mémoire. Un tel endroit peut désigner l'emplacement d'une variable scalaire du programme, comme nous l'avons majoritairement fait en présentant les pointeurs et le passage d'argument par adresse dans les chapitres 7 et 8.2. Mais il peut également représenter l'emplacement où a été réservée la place pour stocker plusieurs valeurs consécutivement, ce qui est exactement un tableau.



Important

En omettant volontairement des subtilités techniques sordides principalement rencontrées avec les chaînes de caractères, nous pouvons dire qu'un tableau d'éléments de type `ty` est représenté par un pointeur de type `ty*`.

L'accès en dehors des limites d'un tableau est une erreur de programmation puisqu'il consiste à lire ou écrire en dehors de la zone qui représente ce tableau.

Important

C ne vérifie pas la légitimité d'un indice i lors d'un accès à la case i d'un tableau. Dit autrement, l'accès est effectué et s'il référence une zone mémoire en dehors du tableau, le programme aura un comportement non prévisible.

Ce trait du langage est parfaitement compréhensible : un tableau est uniquement désigné par son adresse de début, un pointeur. Cela explique que la taille du tableau ne fasse pas partie de la structure du tableau. Or, sans taille connue, il est impossible d'effectuer une vérification sur la validité de l'indice utilisé dans un accès au tableau.

Parfois, l'indice incorrect va désigner un espace mémoire qui n'appartient pas au programme. C'est plutôt le cas favorable car le système d'exploitation va interrompre le programme abruptement, donc le bug sera visible. Malheureusement, dans d'autres cas l'endroit désigné en mémoire appartiendra au programme et désignera donc d'autres variables (ou tableaux ou structures de données) de ce dernier. Donc le système d'exploitation n'aura aucune raison d'interrompre le programme et la corruption sera totalement silencieuse, avec des effets visibles ou non, immédiatement ou non.

9.2.1 Arithmétique sur pointeur

Évaluer l'expression `t[i]` revient à « sauter » `i` « cases de tableau » depuis son début pour accéder à la $\texttt{i}^{\text{ème}}$. Connaissant la taille, donc le nombre d'octets d'un élément de tableau, cela revient à aller chercher la valeur se trouvant en mémoire . . . à l'adresse `t` + `i`× « nb octets par élément ».

L'opération d'addition d'une adresse `a` pointant vers une valeur de type `ty` et d'un entier `i` est intrinsèquement définie en C comme `a + i× sizeof (ty)`. De ce fait, puisqu'un tableau `t` représente une adresse, puisque `t[i]` revient à aller chercher une valeur en mémoire à « `t` plus `i` fois la taille d'un élément », puisque l'addition entre pointeur et entier fait naturellement le « fois la taille d'un élément », nous en déduisons que :

```
t[i]   ≡   *(t + i)
```

Cette équivalence implique que `*t` est identique à `t[0]`. Donc l'adresse de début du tableau est l'adresse de sa première case. Si nous souhaitons récupérer l'adresse d'une case particulière d'un tableau, il suffit d'utiliser l'opérateur d'extraction d'adresse `&` sur cette case : `&t[i]` est l'adresse de la case d'indice `i` du tableau `t`.

Pour aller plus loin

Et comme l'addition est commutative, $t + i = i + t$, donc `*(t + i) = *(i + t)`, donc nous pouvons légalement écrire l'accès au tableau comme `i[t]`. Mais soyons honnête, même si ça fait très guru de savoir cela, il n'est pas forcément utile d'introduire de la confusion en violant l'écriture conventionnelle `t[i]...`

9.3 Tableaux statiques à plusieurs dimensions

Les tableaux statiques peuvent être définis avec plusieurs dimensions : deux pour une « matrice », trois pour un « parallélépipède », en fait autant que désiré. Dans la déclaration d'un tel tableau, la taille de chaque dimension doit être définie entre crochets. La définition

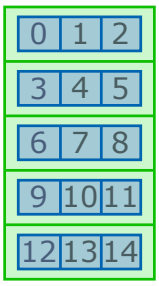
```
int t[5][3] ;
```

crée une « matrice » avec 5 lignes, chacune de 3 éléments. Donc 5 lignes et 3 colonnes. C'est en quelque sorte un tableau taille 5 qui contient dans chaque « case » un tableau de 3 cases. Il est possible d'initialiser un tel tableau en imbriquant les accolades pour respecter le nombre d'éléments dans chaque dimension du tableau. L'accès dans le tableau est effectué en spécifiant l'indice voulu dans chaque dimension : `t[i + 1][j]`.

```c
#include <stdio.h>

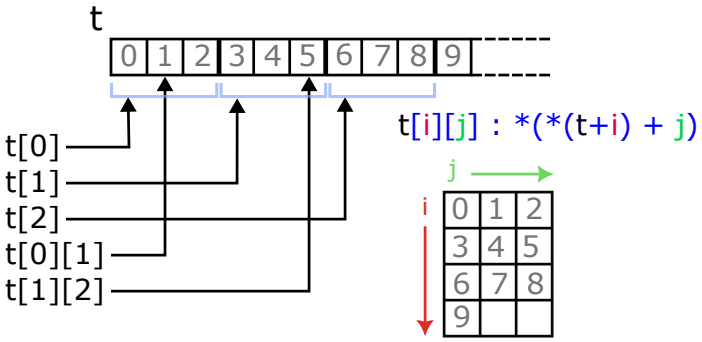
int main () {
  int t[5][3] = {
    { 0, 1, 2 },
    { 3, 4, 5 },
    { 6, 7, 8 },
    { 9, 10, 11 },
    { 12, 13, 14 }
  } ;
  for (int line = 0; line < 5; line++)
      {
    for (int col = 0; col < 3; col++)
        {
      printf ("%d ", t[line][col]) ;
    }
    printf ("\n") ;
  }
  return 0 ;
}
```

int t[5][3] ;




```
$ gcc -Wall array.c -o array.x
$ ./array.x
0 1 2
3 4 5
6 7 8
9 10 11
12 13 14
```

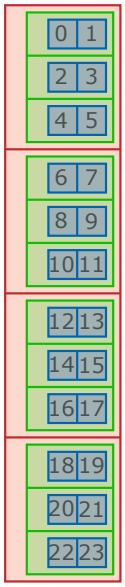
Dans la réalité en mémoire, il n'y a pas de notion de « plusieurs » dimensions. Un tableau à 2 dimensions est en fait un tableau à une dimension dans lequel les lignes sont disposées les unes à la suite des autres. L'interprétation lignes / colonnes n'est qu'une représentation mentale que nous faisons pour raisonner sur cette structure de données.



À la place de créer un tableau à deux dimensions 5 et 3, nous aurions pu créer un tableau à une seule dimension et gérer nous-mêmes la linéarisation des accès. Dans un tel tableau, l'accès logique à une case `[l][c]` aurait simplement été `t[l * 3 + c]`, mais nous aurions dû faire ce calcul nous-mêmes dans le programme au lieu de laisser le compilateur s'en charger.

Le mécanisme de dimensions multiples se généralise à un nombre de dimensions arbitraire, avec toujours la même logique de placer récursivement les éléments de chaque « sous-tableau » consécutivement en mémoire.

int t[4][3][2]




```c
#include <stdio.h>

int main () {
  int t[4][3][2] = {
    { { 0, 1 }, { 2, 3 }, { 4, 5} },
    { { 6, 7 }, { 8, 9 }, { 10, 11 } },
    { { 12, 13 }, { 14, 15 }, { 16, 17 } },
    { { 18, 19 }, { 20, 21 }, { 22, 23 } }
  } ;

  for (int i = 0; i < 4 * 3 * 2; i++)
    printf ("%d ", ((int*) t)[i]) ;
  printf ("\n") ;
  return 0 ;
}
```

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
```

9.4 ~~Retourner un tableau~~

Important

Retourner un tableau statique est une erreur. Le compilateur émet toujours un avertissement qu'il est indispensable de ne pas ignorer.

Les raisons de cet état de fait et la manière d'y remédier sont décrites dans le chapitre 12 et surtout la section 12.1.2.

9.5 En résumé

- Un tableau représente une zone contiguë de valeurs d'un même type.
- Un tableau possède une taille fixée à sa création : il ne peut pas être étendu.
- Un tableau statique a une taille connue à la compilation et définie par une constante entière strictement supérieure à 0.
- Pas de `int t[n] ;` avec `n` une variable dont la valeur est calculée à l'exécution.
- La taille d'un tableau ne fait pas partie du tableau, elle doit être mémorisée à part par le programme.
- Un tableau est défini par le type de ses éléments, son nom, sa taille entre crochets et une éventuelle initialisation.

  $type\text{–}elems\ nom$`[`$taille$`] ;`

  $type\text{–}elems\ nom$`[`$taille$`] = { ..., ..., ... } ;`
- On accède à une « case » d'un tableau par un indice entier.

  `t[0]  t[i + 1]  t[guess_index (...)]`
- Les indices d'un tableau de taille n appartiennent à $[0, n-1]$.
- C ne vérifie pas *a priori* qu'un accès dans un tableau est fait avec un indice légal. Un indice incorrect peut faire planter le programme mais est toujours une erreur.
- Le nom d'un tableau correspond à un pointeur sur le type des éléments du tableau (dans l'immense majorité des cas).

  `int t[27] ;` $\longrightarrow$ `t` $\equiv$ `int*`
- `t[i]` $\equiv$ `*(t + i)`
- Les tableaux peuvent avoir un nombre quelconque de dimensions.

  `int t[5][3] ;`
- On accède à un tableau à n dimensions en spécifiant un indice dans chaque dimension.

  `t[i][j] = t[i][j] + 1 ;`

Chapitre 10

Chaînes de caractères

Il n'existe pas de type explicite pour les chaînes de caractères en C. Elles sont encodées comme de simples tableaux de caractères, donc de `char`. Ainsi, le type des chaînes est `char*`.

Puisque les tableaux ne mémorisent pas leur taille, il faut un moyen pour déterminer la longueur d'une chaîne. La fin d'une chaîne est matérialisée par une *sentinelle*, c'est-à-dire un marqueur qui n'est autre que le caractère noté `'\0'`. Ce caractère est celui dont la valeur numérique est 0. Ainsi, la chaîne `"Glop"` est un tableau constitué des 5 caractères `'G'`, `'l'`, `'o'`, `'p'`, `'\0'` dont les valeurs numériques sont 71, 108, 111, 112, 0.

La syntaxe du langage offre l'écriture pratique de constantes chaînes de caractères entre `"` comme nous en avons déjà vu de nombreuses occurrences dans ce document. Néanmoins tout cela se réduit *in fine* en tableaux de `char` terminés par un `'\0'`.

10.1 Définition

Fort logiquement, définir une « variable chaîne » revient à définir un tableau de `char`. Comme toute définition de tableau, une initialisation est possible lors de la définition.

```
char une_chaine[10] ;
char chaine_glop[5]   = { 'G', 'l', 'o', 'p', '\0' } ;
char chaine_glop2[10] = { 'G', 'l', 'o', 'p', '\0' } ;
char chaine_glop3[]   = { 'G', 'l', 'o', 'p', '\0' } ;
char chaine_glop4[]   = "Glop" ;
```

La chaîne `une_chaine` n'est pas initialisée. C'est un simple tableau de 10 `char` qu'il conviendra de remplir ultérieurement. Puisque sa longueur est 10, elle pourra stocker des « mots » d'au plus 9 « lettres utiles », sachant qu'il faudra un `'\0'` pour marquer sa fin.

La chaîne `chaine_glop` est explicitement définie de longueur 5 et initialisée en énumérant les caractères à stocker dans chaque case. Il faut noter la présence explicite du `'\0'` dans l'initialisation qui vient se rajouter aux 4 « lettres utiles », pour exactement une longueur de 5. La chaîne `chaine_glop2` est initialisée de la même manière, mais possède une plus grande taille. Elle peut donc largement mémoriser la séquence de caractères avec laquelle elle est initialisée.

Lorsque l'on souhaite définir une chaîne constante, donc dont la longueur n'a pas de raison de changer, plutôt que de la calculer soi-même pour la faire figurer en taille du tableau déclaré, il est possible de laisser le compilateur inférer la taille du tableau à partir de la longueur de l'initialisation. On ne met alors aucune taille entre les crochets, comme dans la définition de `chaine_glop3`. Il faut néanmoins être conscient que la taille inférée par le compilateur est au plus juste. Donc si l'on modifie la chaîne ultérieurement, il n'y aura toujours la place que pour 5 `char`, dont le `'\0'` final.

Énoncer un par un les caractères d'un message est fastidieux. À la place, il est possible d'utiliser une constante chaîne « habituelle » dans l'initialisation, en utilisant la forme avec des `"`. Le compilateur

se chargera de rajouter le `'\0'` final.

10.1.1 Chaînes non modifiables

Dans la section 9.2, nous avons vu qu'un tableau est représenté par un pointeur. Donc `char t[]` est similaire à `char *t`. Il semble alors logique de pouvoir définir des chaînes initialisées sous la forme :

```
char *chaine_glop = "Glop" ;
```

Important

Une telle définition induit des différences importantes. Certaines sont subtiles, mais une est fondamentale et très visible. Définir une chaîne en utilisant un type pointeur plutôt qu'un type tableau par

```
char *chaine_glop = "Glop" ;
```

rend la chaîne non modifiable. Dit autrement, il ne sera pas possible de modifier ultérieurement son contenu, ses caractères. Consultez la section 10.2.2 pour une illustration de cette différence.

L'autre risque en confondant aveuglement la définition d'une chaîne comme un tableau et comme un pointeur est lors de la définition (prétendue, car nous allons voir que c'est incorrect) d'une chaîne « pour plus tard». Imaginons que nous soyons dans l'optique de définir une chaîne que nous remplirons et utiliserons plus tard dans le programme. Ce scénario est le pendant avec les chaînes de ce que nous ferions avec une variable entière dans :

```
int i ;
...
i = 5 ;
...
i = i + 2 ;
...
```

Il ne viendrait pas à l'idée de définir notre chaîne comme un tableau vide (`char str[] ;` est rejeté par le compilateur) ou pire, de taille nulle (`char str[0] ;` est accepté comme extension de `gcc` pour un cas d'usage très particulier).

Par contre, la définition de notre chaîne par juste `char *str ;` est beaucoup moins ostensiblement incorrecte. La petite musique « ♪ *chaînes et pointeurs c'est la même chose* ♪ » a la vie dure et ce type de définition est très courante chez les débutants.

Important

Comme expliqué dans la section 7.2.2, la définition `char *str ;` ne définit pas une chaîne de caractères. Une telle déclaration ne fait que définir un pointeur non initialisé. Ce pointeur désigne un endroit quelconque en mémoire, certainement pas là où il y a de la place pour entreposer des caractères pour former une chaîne. Tenter de se servir de ce pointeur comme d'une chaîne aura un résultat imprévisible, non déterministe.

10.2 Opérations sur les chaînes

10.2.1 Affichage

La fonction `printf` a été présentée dans le chapitre 6. Nous avons vu qu'elle prenait en premier argument une chaîne nommé *format* qui représente la forme du message à afficher, dans laquelle les séquences % sont substituées. Pour afficher une chaîne, la séquence est `%s`.

```c
#include <stdio.h>

int main () {
  char str[] = "Glop" ;
  printf ("%s", str) ;
  return 0 ;
}
```

Le programme ci-dessus affiche donc le contenu de la chaîne `str`. Il n'affiche pas de retour à la ligne final (ce qui peut poser le problème décrit en section 6.1), mais c'est intentionnel. Même pour afficher uniquement le contenu d'une chaîne, sans message autour, la présence du format en premier argument de `printf`, avec un `%s` est indispensable.

Dit autrement, tenter d'afficher `str` en la passant directement comme seul argument de `printf` (oui, le format c'est une chaîne après tout) avec `printf (str) ;` est une pratique dangereuse en terme de sécurité et le compilateur le signale par un avertissement (même sans `-Wall`).

```
$ gcc pr.c
pr.c:5:11: warning: format string is not a string literal
      (potentially insecure) [-Wformat-security]
    5 |   printf (str) ;
      |           ^~~
pr.c:5:11: note: treat the string as an argument to avoid
      this
    5 |   printf (str) ;
      |           ^
      |           "%s",
1 warning generated.
```

10.2.2 Modification des caractères

Hormis dans le cas des chaînes non modifiables (cf. section 10.1.1), les caractères d'une chaîne peuvent être individuellement modifiés comme dans n'importe quel tableau.

```c
#include <stdio.h>

int main () {
  char str[] = "Glop" ;
  str[0] = 'P' ; str[3] = '\0' ;
  printf ("%s\n", str) ;
  return 0 ;
}
```

```
$ gcc -Wall modstr.c -o modstr.x
$ ./modstr.x
Plo
```

Le programme ci-dessus remplace la première lettre par un `'P'` et la troisième par un `'\0'`. De ce fait, la fin de la chaîne va être rencontrée plus tôt et le message affiché sera simplement « Plo ». La chaîne est entre guillemets plus courte, même si la taille du tableau `str` n'a pas changé : il y a toujours la place pour stocker 5 caractères.

Modifions très légèrement notre programme pour déclarer la chaîne via un pointeur, donc avec un `char*` plutôt qu'un `char[]`. Comme expliqué en section 10.1.1, la chaîne devient non modifiable. Aucun avertissement n'est émis lors de la compilation mais l'exécution est sans appel : ça plante.

```c
#include <stdio.h>


int main () {
  char *str = "Glop" ;
  str[0] = 'P' ; str[3] = '\0' ;

  printf ("%s\n", str) ;
  return 0 ;
}
```

```
$ gcc -Wall horror.c -o horror.x
$ ./horror.x
Bus error: 10
```

10.2.3 ~~Affectation~~

Important

Contrairement à d'autres langages comme OCaml ou Python, il n'y a pas d'opérateur d'affectation `s1 = s2` entre les chaînes.

Si la chaîne `s1` est définie comme un tableau, alors quelle que soit la définition de `s2` la compilation provoquera une erreur. C'est compréhensible car `s1` est un tableau et il n'y a pas d'affectation de tableau.

```c
int main () {
  char s1[] = "Oui" ;
  char s2[] = "Non" ;
  s1 = s2 ;
  return 0 ;
}
```

```c
int main () {
  char s1[] = "Oui" ;
  char *s2 = "Non" ;
  s1 = s2 ;
  return 0 ;
}
```

```
$ gcc horror.c
horror.c: In function 'main':
horror.c:4:6: error: assignment to expression with array type
    4 |   s1 = s2;
      |      ^
```

Par contre, si `s1` est définie comme un pointeur, la compilation ne génère aucun avertissement. Toutefois, cette instruction n'effectue aucune copie entre les chaînes. L'opérateur = se comporte comme pour les autres scalaires avec les pointeurs : il copie les valeurs d'adresse.

```c
#include <stdio.h>

int main () {
  char *s1 = "Oui" ;
  char s2[] = "Non" ;
  s1 = s2 ;
  s2[0] = 'B' ;
  printf ("%s %s\n", s1, s2) ;
  printf ("s1: %p s2: %p\n", s1, s2)
      ;
  return 0 ;
}
```

```
$ gcc -Wall horror.c -o horror.x
$ ./horror.x
Bon Bon
s1: 0x16b32af3c s2: 0x16b32af3c
```

Ainsi, dans le programme ci-dessus, à la fin `s1` et `s2` désignent la même adresse : celle de `s2`. Et lorsque nous modifions `s2` après « l'affectation » (incorrecte, bien sûr), les « deux » chaînes sont bien identiques... puisque c'est la même qui est désignée par `s1` et par `s2`.

10.2.4 Longueur d'une chaîne

Calculer la longueur d'une chaîne revient à compter le nombre de `char` depuis son début tant que l'on ne rencontre pas le caractère `'\0'`. Il est donc facile d'écrire une telle fonction de calcul de longueur :

```
int length_string (char *str) {
  int i = 0 ;
  while (str[i] != '\0') i++ ;
  return i ;
}
```

La bibliothèque standard fournit directement une fonction qui retourne la longueur d'une chaîne de caractères nommée `strlen` qui nécessite l'inclusion du fichier `<string.h>`.

Important

La fonction `strlen` ne compte pas le `'\0'` final. Ainsi, lorsque l'on s'en sert pour calculer la longueur d'un tableau à créer pour copier une chaine, il faut toujours rajouter `+1` à la valeur retournée par `strlen`.

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

int main () {
  char str[10] = "Foobar" ;
  printf ("Long sans \\0: %lu\n", strlen (str))
      ;
  return 0 ;
}
```

```
$ gcc -Wall len.c -o len.x
$ ./len.x
Long sans \0: 6
```

Notez que, en accord avec sa page de manuel (`man 3 strlen`), la fonction `strlen` retourne un entier long non signé (plus exactement, un `size_t` qui est un alias de `unsigned long int`). Il convient donc d'utiliser le format `%lu` dans la fonction `printf`.

10.2.5 Copie de chaîne

Copier une chaîne dans une autre revient simplement à copier le contenu d'un tableau dans un autre. C'est moral puisqu'une chaîne est un tableau. Il est donc facile d'écrire une telle fonction de copie :

```
void copy_string (char *dest, char *src) {
  int i = 0 ;
  while (src[i] != '\0') {
    dest[i] = src[i] ;
    i++ ;
  }
  dest[i] = '\0' ;
```

→

```
}
```

Puisque la fonction devra toujours au moins copier le `'\0'` final, nous avons ici un cas où la boucle `do while` (présentée en section 5.2.3) est particulièrement adaptée (et ce n'est pas tous les jours). La fonction `copy_string` peut donc s'écrire de manière totalement équivalente de la façon suivante :

```
void copy_string (char *dest, char *src) {
  int i = 0 ;
  do {
    dest[i] = src[i] ;
  } while (src[i++] != '\0') ;
}
```

Pour aller plus loin

... ou en encore plus concis (mais aussi plus cryptique) en utilisant l'arithmétique de pointeurs (cf. section 9.2.1), les opérateurs de post-incrémentation (et non pré-décrémentation) et le fait que `'\0'` vaut 0, donc « faux » :

```
void copy_string (char *dest, char *src) {
  while (*src) *(dest++) = *(src++) ;
  *dest = '\0' ;
}
```

Exercice : Écrivez (en C bien entendu) cette fonction (en conservant son nom) en utilisant le moins de caractères possible. La fonction devra compiler sans avertissement avec l'option `-Wall`. Record actuel : 52 caractères.

La bibliothèque standard fournit directement une fonction de copie de chaîne nommée `strcpy` qui nécessite l'inclusion du fichier `<string.h>`. Attention à l'ordre des paramètres :

```
strcpy (dest, src) ;
```

copie le contenu de la chaîne `src` dans la chaîne `dest`. L'ordre des paramètres suit la logique d'une affectation `dest` ← `src`.

Important

Lorsque nous copions une chaîne `s1` dans une chaîne `s2`, il faut que les deux chaînes existent (i.e. que les tableaux aient été créés) et que la chaîne `s2` (i.e. le tableau) ait une longueur supérieure ou égale à celle de `s1` (en comptant le `'\0'` final). Dans le cas contraire, la copie écrasera les données qui suivent `s2` en provoquant un comportement imprévisible et non déterministe du programme.

Important

Lorsque nous copions une chaîne `s1` dans une chaîne `s2`, il faut que la chaîne `s1` soit terminée par un `'\0'` final). Dans le cas contraire, la boucle de parcours des caractères continuera à parcourir la mémoire jusqu'à tomber sur un octet nul en provoquant un comportement imprévisible et non déterministe du programme.

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

int main () {
  char str[10] = "Foobar" ;
  char copied[100] ;
  strcpy (copied, str) ;
  str[0] = 'G' ;
  printf ("str: %s copied: %s\n",
          str, copied) ;
  return 0 ;
}
```

```
$ gcc -Wall cpystr.c -o cpystr.x
$ ./cpystr.x
str: Goobar copied: Foobar
```

La modification effectuée dans `str` après la copie par `strcpy` n'a modifié que `str` puisque `copied` est une copie de la chaîne initiale et est logée à un endroit différent de `str` en mémoire.

10.2.6 Comparaison de chaînes

Puisque les chaînes sont des tableaux, représentés par des pointeurs, utiliser les opérateurs ==, <, >, etc. ne fonctionne pas. En effet, ils ne feraient que comparer les adresses des chaînes comparées. Pour preuve, les deux chaînes `s1` et `s2` ont le même contenu mais sont considérées différentes par ==. À la compilation un avertissement est d'ailleurs émis.

```c
int main () {
  char s1[10] = "Foobar" ;
  char s2[10] = "Foobar" ;
  if (s1 == s2) printf ("Pareilles.\n") ;
  else printf ("Différentes.\n") ;
  return 0 ;
}
```

```
$ gcc -Wall cmpstr.c -o cmpstr.x
cmpstr.c: In function 'main':
cmpstr.c:6:10: warning: comparison between two arrays [-Warray-compare]
    6 |   if (s1 == s2) printf ("Pareilles.\n") ;
      |          ^~
cmpstr.c:6:10: note: use '&s1[0] == &s2[0]' to compare the addresses
$ ./cmpstr.x
Différentes.
```

Bien entendu, la construction `switch` présentée en section 5.1.3 ne fonctionne pas non plus puisque ses `case` doivent se réduire à des constantes entières.

La bibliothèque standard fournit directement une fonction de comparaison de chaînes nommée `strcmp` qui nécessite l'inclusion du fichier `<string.h>`.

```
strcmp (s1, s2) ;
```

retourne un entier qui vaut :

- 0 si `s1` et `s2` ont le même contenu, donc sont « égales » en terme de contenu,
- < 0 si `s1` $<$ `s2` selon l'ordre lexicographique,
- > 0 si `s1` $>$ `s2` selon l'ordre lexicographique.

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

int main () {
  char s1[10] = "Aaa" ;
  char s2[10] = "Aaa" ;
  char s3[10] = "Aa" ;
  char s4[10] = "Aab" ;

  printf ("%d\n", strcmp (s1, s2)) ;
  printf ("%d\n", strcmp (s1, s3)) ;
  printf ("%d\n", strcmp (s1, s4)) ;
  printf ("%d\n", strcmp (s4, s1)) ;
  return 0 ;
}
```

```
$ gcc -Wall cmpstr.c -o cmpstr.x
$ ./cmpstr.x
0
97
-1
1
```

Les contenus `s1` et `s2` sont identiques, donc `strcmp` retourne 0. La chaîne `s1` a le même préfixe que `s3` mais est plus longue, donc `strcmp retourne` une valeur strictement positive. Les chaînes `s1` et `s4` sont de même longueur, avec un préfixe commun et diffèrent sur leur 3$^{\text{ème}}$ caractère. Comme le `'b'` est situé après le `'a'` en terme de caractère (donc `'a' < 'b'`), la chaîne `s1` est inférieure à `s4` et `strcmp` retourne une valeur strictement négative. Fort logiquement, la comparaison inverse retourne une valeur strictement positive.

Pour aller plus loin

L'ordre lexicographique $\leqslant_S$ sur les chaînes de caractères est plus formellement défini de la manière suivante. Soit `s1` et `s2` deux chaînes de caractères. On note `s`$[i]$ le caractère d'indice i dans une chaîne `s` et $|$`s`$|$ la longueur de `s`. On note $<_C$ l'ordre sur les caractères.

`s1` $\leqslant_S$ `s2` ssi ou bien :

- $|$`s1`$| < |$`s2`$| \wedge \forall i,\ 0 \leqslant i < |$`s1`$| \Rightarrow$ `s1`$[i] =$ `s2`$[i]$
- $\exists i,\ 0 \leqslant i < min(|$`s1`$|, |$`s2`$|) \wedge \forall j,\ 0 \leqslant j < i \Rightarrow$ `s1`$[j] =$ `s2`$[j] \wedge$ `s1`$[i] <_C$ `s2`$[i]$

10.2.7 Concaténation de chaînes

La bibliothèque standard fournit directement une fonction permettant de rajouter une chaîne à la suite d'une autre nommée `strcat` qui nécessite l'inclusion du fichier `<string.h>`.

```c
strcat (dest, src) ;
```

recopie le contenu de la chaîne `src` à la fin de `dest`, c'est-à-dire à partir de l'endroit où était situé son `'\0'` final.

Important

Il est indispensable que la chaîne `dest` ait une taille suffisante pour accueillir les caractères de `src`, y compris son `'\0'` final. Aucune vérification n'est effectuée.

Après concaténation, la longueur effective de `dest` sera son ancienne longueur plus celle de `src` plus le sempiternel `'\0'` final.

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

int main () {
  char add_here[100] = "Fluctuat" ;
  char add_this[] = " nec mergitur" ;
  printf ("%s\n", add_here) ;
  strcat (add_here, add_this) ;
  printf ("%s\n", add_here) ;
  return 0 ;
}
```

```
$ gcc -Wall catstr.c -o catstr.x
$ ./catstr.x
Fluctuat
Fluctuat nec mergitur
```

10.2.8 Recherche d'une sous-chaîne

La bibliothèque standard fournit directement une fonction de recherche de la première occurrence d'une sous-chaîne dans une chaîne nommée `strstr` qui nécessite l'inclusion du fichier `<string.h>`.

```
strstr (in, sub) ;
```

retourne l'adresse du début de la sous-chaîne `sub` trouvée dans `in` si elle existe et sinon retourne le pointeur invalide `NULL`.

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

int main () {
  char big_string[] = "Tic tac tic tac tic..." ;
  char sub_string1[] = "tic" ;
  char sub_string2[] = "Tac" ;
  printf ("Trouvé 1: %p\n", strstr (big_string, sub_string1)) ;
  printf ("Adresse char #8: %p\n", &big_string[8]) ;
  printf ("Trouvé 2: %p\n", strstr (big_string, sub_string2)) ;
  return 0 ;
}
```

```
$ gcc -Wall substr.c -o substr.x
$ ./substr.x
Trouvé 1: 0x16d9eaf38
Adresse char #8: 0x16d9eaf38
Trouvé 2: 0x0
```

La première recherche retourne un pointeur qui indique bien l'adresse du caractère d'indice 8 où commence la première occurrence de « tic ». La seconde recherche échoue et le pointeur retourné est `NULL` (dont la valeur numérique est bien 0).

10.3 En résumé

- Une chaîne est un tableau de `char` terminé par `'\0'`.
- Le `'\0'` final est indispensable.
- On peut initialiser une chaîne soit caractère par caractère (avec `'\0'` explicite), soit avec une constante entre `"`.

```
char chaine_glop[5] = { 'G', 'l', 'o', 'p', '\0' } ;
char chaine_glop2[] = "Glop" ;
```

- Une chaîne définie par un tableau est modifiable.

```
char chaine_glop3[10] = { 'G', 'l', 'o', 'p', '\0' } ;
```

- Une chaîne définie par un pointeur est non modifiable.

```
char *chaine_glop4 = "Je ne suis pas modifiable" ;
```

- Pour afficher une chaîne avec `printf` il faut utiliser `%s`.
- On modifie les caractères d'une chaîne (modifiable) comme pour n'importe quel tableau.

```
str[4] = 'a' ; str[7] = '\n' ;
```

- Les fonctions de manipulation de chaînes de la bibliothèque standard nécessitent `#include <string.h>`
- Il n'y a pas d'affectation entre chaînes avec `=`.
- Longueur d'une chaîne : `strlen`. Ne compte pas le `'\0'` final.
- Copie d'une chaîne : `strcpy`. Premier argument = destination, second argument = source.
- Les opérateurs `==`, `!=`, `<`, etc. ne fonctionnent pas sur les chaînes.
- Comparaison entre chaînes : `strcmp`.
- Concaténation de chaîne : `strcat`.
- Recherche de sous-chaîne : `strstr`.

Chapitre 11

Arguments de la ligne de commande

Exécuter un programme pour différentes valeurs pourrait se faire en changeant ces valeurs dans le code source du programme et en le recompilant. Outre le fait que ce serait terriblement inefficace, il est courant de ne pas disposer du code source d'un logiciel.

Lorsque l'on souhaite injecter des informations en entrée d'un programme pour piloter son exécution, il est très avantageux de le faire au moment du lancement du programme. Il est aussi possible de le faire en cours d'exécution du programme, par exemple en utilisant la fonction `scanf` (et ses variantes) présentée dans le chapitre 13. Néanmoins, cela implique que l'utilisateur soit présent au bout du clavier à chaque moment où le programme se rend compte qu'il a besoin d'une information.

C'est pourquoi l'on invoque les programmes qui nécessitent une entrée textuelle de la part de l'utilisateur en ajoutant sur la ligne de commande, après le nom du programme lancé, différentes chaînes de caractères. Par exemple, lorsque vous invoquez :

```
gcc -Wall foo.c -o foo.x
```

au tout début de son exécution le programme `gcc` reçoit entre autres choses les 4 arguments `-Wall`, `foo.c`, `-o` et `foo.x` sous forme de chaînes de caractères.

Important

Il faut remarquer que ce mécanisme est très adapté lorsqu'il y a relativement peu d'informations à communiquer au programme. Il n'est pas concevable qu'un utilisateur doive entrer 50 informations sur la ligne de commande pour lancer un programme. Dans le cas contraire, l'utilisation de fonctions comme `scanf` (en combinaison avec la redirection de fichier sur l'entrée standard) sera bien plus adaptée.

11.1 Récupération des arguments

Jusqu'à présent, la fonction `main` avait été présentée avec la structure `int main () { ... }` ce qui lui interdisait toute possibilité de recevoir des arguments. Plus généralement, cette fonction a la forme :

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

où figurent deux paramètres (comme dans une fonction « normale ») dont les types sont immuables. Les noms donnés à ces paramètres peuvent être quelconques, mais par convention, ils sont appelés `argc` et `argv`.

Lorsque la fonction `main` débute l'exécution du programme, elle reçoit ces arguments de la part du mécanisme qui a permis d'invoquer le programme. Typiquement, lorsque le programme est lancé depuis un interprète de ligne de commande dans un terminal (`bash` par exemple), c'est ce dernier qui récupère la ligne entrée par l'utilisateur et la transmet au noyau du système d'exploitation. Ce dernier se charge de transmettre toutes ces information au processus créé pour l'exécution du programme. Lorsqu'un programme est lancé par un « double-clic » dans une interface graphique, le mécanisme est similaire, c'est l'interface graphique qui transmet les informations au noyau.

Le premier argument, `argc` (pour "*arguments count*"), est un entier qui représente le nombre d'arguments sur la ligne de commande.

Important

Lorsque nous parlons du nombre d'arguments sur la ligne de commande, il ne s'agit pas du nombre d'arguments passés au programme. Cela inclut le « nom » du programme.

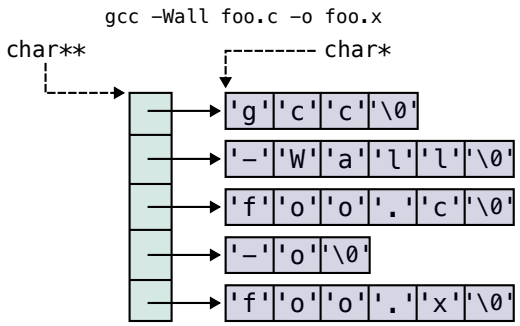
Si nous reprenons l'exemple `gcc -Wall foo.c -o foo.x`, la ligne de commande effectivement entrée dans le terminal comporte 5 arguments :

1. `gcc`
2. `-Wall`
3. `foo.c`
4. `-o`
5. `foo.x`

Le second argument, `argv` (pour "*arguments values*" ou "*arguments vector*"), est un tableau de chaînes de caractères qui contient tous les arguments présents sur la ligne de commande qui a invoqué le programme, dans leur ordre d'apparition.

Puisqu'une chaîne de caractères est un tableau de `char` (terminé par un `'\0'`), elle est de type `char*`. Puisque `argv` est un tableau de telles chaînes, il est de type tableau de `char*`, donc `char**`. Les écritures `char **argv` et `char *argv[]` sont totalement équivalentes.

En reprenant toujours le même exemple `gcc -Wall foo.c -o foo.x`, le tableau `argv` aura la forme suivante où la première entrée est le « nom » du programme.



Pour être exact, la première entrée de `argv` n'est pas juste le « nom » du programme : c'est le chemin complet qui a été invoqué sur la ligne de commande pour lancer le programme. Si le programme avait été invoqué par :

```
/usr/bin/gcc -Wall foo.c -o foo.x
```

alors la première chaîne aurait été `"/usr/bin/gcc"`.

Important

Avant de tenter d'utiliser les arguments qu'elle reçoit, la fonction `main` doit s'assurer qu'elle en a bien reçu, de surcroît le nombre attendu si celui-ci est fixé.

Ainsi tout programme qui utilise les arguments de la ligne de commande doit commencer par une vérification de la présence de ces derniers. Et dans le cas contraire, il doit émettre un message d'erreur et quitter en signalant une erreur (`return` $+\ 0$). Ne pas faire cette vérification et accéder aveuglement à `argv` revient à accéder en dehors des bornes d'un tableau (cf. section 9.2).

```
#include <stdio.h>

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

```c
  if (argc != 3) {
    printf ("Erreur: 2 arguments sont attendus.\n") ;
    return 1 ;
  }
  printf ("Mes 2 args: '%s' et '%s'.\n", argv[1], argv[2]) ;
  return 0 ;
}
```

```
$ gcc -Wall varg.c -o varg.x
$ ./varg.x
Erreur: 2 arguments sont attendus.
$ ./varg.x Foo 123
Mes 2 args: 'Foo' et '123'.
$ ./varg.x ,,$ "avec espace"
Mes 2 args: ',,$' et 'avec espace'.
```

11.2 Utilisation des arguments

Puisque `argv` est un tableau de chaînes de caractères, il est très simple de le parcourir pour afficher son contenu avec le %s de `printf`. Une chose est certaine, il a toujours au moins une entrée : le chemin du programme. L'exemple suivant se déroule dans le répertoire `/tmp/` comme l'indique l'affichage de la commande `pwd` (“*print working directory*”) du shell.

```c
#include <stdio.h>

int main (int argc, char *argv[])
    {
  for (int i = 0; i < argc; i++)
    printf ("%s\n", argv[i]) ;
  return 0 ;
}
```

```
$ pwd
/tmp
$ gcc ./printargs.c -o printargs.x
$ ./printargs.x
./printargs.x
$ /tmp/printargs.x
/tmp/printargs.x
$ ../tmp/printargs.x Alice 123 3.14 Bob
../tmp/printargs.x
Alice
123
3.14
Bob
```

Dans l'exemple précédent, « 123 » et « 3.14 » sont des chaînes de caractères, même si manifestement ces arguments sont censés dénoter un entier et un réel. D'ailleurs, si ce n'étaient pas des chaînes, le %s de `printf` aurait provoqué un affichage incorrect (voire aurait planté le programme).

Lorsqu'un programme attend des arguments devant représenter des valeurs numériques, il est nécessaire de transformer ces chaînes en véritables valeurs numériques, donc en entiers ou en réels.

La bibliothèque standard de C fournit différentes fonctions à cet effet, qui permettent de convertir une chaîne de caractères en valeur numérique, entier, signé, non signé, flottant, double, etc.

Dans un souci de simplicité, nous commençons ici par présenter le jeu de fonctions les plus simples. Leur désavantage est de ne pas être très robustes. En effet, elles ne permettent pas de détecter les erreurs si la chaîne passée en argument ne correspondait pas à un nombre (les fonctions plus robustes sont présentées dans la section 11.3).

Ces fonctions sont celles de la famille `ato` dont le suffixe indique vers quel type numérique est faite la transformation. La chaîne passée en argument est supposée être écrite en base 10 (et sera invariablement interprétée comme tel). Toutes ces fonctions nécessitent l'inclusion du fichier d'en-tête `<stdlib.h>`.

atoi	:	char* ⇝ int
atol	:	char* ⇝ long int
atoll	:	char* ⇝ long long int
atof	:	char* ⇝ float / double

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

int main (int argc, char *argv[]) {
  if (argc != 5) {
    printf ("Erreur: 4 arguments attendus.\n") ;
    return 1 ;
  }
  int i = atoi (argv[1]) ;
  long l = atol (argv[2]) ;
  long long ll = atoll (argv[3]) ;
  float f = atof (argv[4]) ;
  printf ("i: %d, l: %ld, ll: %lld, f: %f\n", i, l, ll, f) ;
  return 0 ;
}
```

```
$ gcc -Wall ato.c -o ato.x
$ ./ato.x
Erreur: 4 arguments attendus.
$ ./ato.x 10 5000000000 98765432109876543 3.14
i: 10, l: 5000000000, ll: 98765432109876543, f: 3.140000
$ ./ato.x Alice 5000000000 98765432109876543210 Bob
i: 0, l: 5000000000, ll: 9223372036854775807, f: 0.000000
```

La dernière invocation du programme illustre la faiblesse des fonctions de la famille `ato`. Les premier et dernier arguments fournis sont des chaînes qui ne représentent pas des valeurs numériques. Le troisième est une chaîne numérique mais dont la valeur excède la capacité de représentation d'un `long long int` (ici 8 octets, comme pour `long int` sur l'architecture utilisée pour les exemples de ce document). Dans tous ces cas, les fonctions ont retourné des valeurs manifestement incorrectes, mais impossible à différencier de valeurs qui auraient pu être correctes.

11.3 Meilleure utilisation des arguments

Pour aller plus loin

Légèrement plus complexes, il existe une autre famille de fonctions permettant de convertir des chaînes de caractères en valeurs numériques, tout en ayant le moyen de vérifier que la conversion a réussi. Dit autrement, ces fonctions permettent de vérifier si la chaîne passée en argument était bien l'écriture textuelle d'un nombre.

Ces fonctions sont celles de la famille `strto`. Elles sont séparées en deux catégories : celles pour convertir des vers entiers et celles vers des réels. Les deux nécessitent l'inclusion du fichier d'en-tête `<stdlib.h>`.

11.3.1 Conversion vers entiers

La fonction `strtol` permet de convertir vers un `long int`, `strtoll` vers un `long long int`. Leurs prototypes sont les suivants et nous allons détailler leurs arguments.

```c
long int strtol (char *nptr, char **endptr, int base) ;
```

```
long long int strtoll (char *nptr, char **endptr, int base) ;
```

Ces fonctions prennent en argument `nprt` la chaîne à convertir, `endptr` un pointeur sur chaîne et `base` la base dans laquelle la chaîne doit être interprétée. Nous voyons que ces fonctions ne sont pas limitées à la base 10 comme le sont celles de la famille `ato`. Ces deux fonctions retournent la valeur entière issue de l'interprétation de la chaîne désignée par `nprt`.

Si le paramètre `endptr` n'est pas `NULL`, ces fonctions « retournent » (grâce au passage d'argument par adresse), l'adresse du premier caractère invalide, c'est-à-dire qui n'est pas un « chiffre » dans la base d'interprétation. Il en existe forcément un :

- soit la chaîne possède un préfixe représentant un nombre et d'autres caractères après, par exemple comme dans `"123soleil"`,
- soit la chaîne représente entièrement un nombre et le `'\0'` final qui la clôt est bien un caractère invalide.

C'est donc grâce à ce paramètre qu'il est possible de vérifier la validité de la chaîne que nous avons fournie en argument. Si cette chaîne n'était pas vide (donc pas uniquement un `'\0'`) et si le caractère situé à l'adresse « retournée » dans `endptr` est un `'\0'`, alors la chaîne entière représentait bien un nombre.

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

int main () {
  long int i ;
  char *endscan ;
  char s1[] = "  123" ;
  char s2[] = "123soleil" ;
  char s3[] = "eh non" ;
  char s4[] = "12AEF00" ;

  i = strtol (s1, &endscan, 10) ;
  printf ("%ld, *endscan: '%c'\n", i, *endscan) ;
  i = strtol (s2, &endscan, 10) ;
  printf ("%ld, *endscan: '%c'\n", i, *endscan) ;
  i = strtol (s3, &endscan, 10) ;
  printf ("%ld, *endscan: '%c'\n", i, *endscan) ;
  i = strtol (s4, &endscan, 10) ;
  printf ("%ld, *endscan: '%c'\n", i, *endscan) ;
  i = strtol (s4, &endscan, 16) ;
  printf ("%ld (hexa %lx), *endscan: '%c'\n", i, i, *endscan) ;
  return 0 ;
}
```

```
$ gcc -Wall strtotest.c -o strtotest.x
$ ./strotest.x
123, *endscan: ''
123, *endscan: 's'
0, *endscan: 'e'
12, *endscan: 'A'
19590912 (hexa 12aef00), *endscan: ''
```

Le programme précédent déclare `endscan` un pointeur sur `char` dont l'adresse sera passée (en passage d'argument par adresse) aux appels à `strtol` pour vérifier les conversions en entier. Viennent ensuite les 4 chaînes qui seront soumises à conversion. À chaque fois, le programme affiche la valeur retournée par `strtol` ainsi que le caractère où s'est arrêtée la conversion.

1. Chaîne `s1` en base 10. Cette chaîne représente bien un nombre (les espaces en début sont ignorés par `strtol`). L'entier obtenu est bien 123 et le caractère sur lequel s'est terminée la

conversion est le `'\0'` final (qui ne s'affiche pas comme une « lettre »). Comme expliqué dans le paragraphe précédent, cela signifie que la chaîne complète représentait bien un entier.
2. Chaîne `s2` en base 10. Le préfixe de cette chaîne représente bien un nombre dans cette base. La conversion est effectuée sur ce préfixe et le caractère où elle s'est arrêtée est bien le `'s'` de `"soleil"`.
3. Chaîne `s3` en base 10. Clairement, cette chaîne ne représente pas une valeur numérique et la conversion s'arrête dès le premier caractère. La valeur retournée n'est donc pas significative.
4. Chaîne `s4` en base 10. Les deux premiers caractères représentent bien une valeur décimale. La conversion retourne donc la valeur 12 et s'arrête sur le caractère `'A'`.
5. Chaîne `s4` en base 16. Dans sa totalité, cette chaîne représente un nombre écrit en hexadécimal (base 16 avec les lettres de `'a'` à `'f'` pour les chiffres au-dessus de 9). Cette fois-ci la conversion consomme toute la chaîne, contrairement au cas précédent. La base a donc une importance capitale. La valeur résultat est affiché en base 10 et 16 pour comparaison.

Bien entendu, `strtol` peut être utilisée pour convertir vers un `int` tant que la chaîne analysée représente une valeur numérique tenant sur 32 bits. Pour s'en assurer, la pratique consiste à stocker temporairement la valeur retournée par `strtol` dans un `long int`, puis à vérifier manuellement que cette valeur peut être stockée de manière sûre dans un `int` avant de faire cette dernière conversion implicite. Illustrons ce principe avec le programme suivant.

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

int main (int argc, char *argv[]) {
  long int tmp_long ;
  int the_i ;
  char *endscan ;

  if (argc != 2) {
    printf ("Erreur: 1 argument numérique attendu.\n") ;
    return 1 ;
  }
  tmp_long = strtol (argv[1], &endscan, 10) ;
  if (*endscan == '\0' && argv[1][0] != '\0') {
    /* Conversion réussie. */
    if (tmp_long <= INT_MAX && tmp_long >= INT_MIN) {
      the_i = tmp_long ;
      printf ("C'est bon, the_i vaut %d\n", the_i) ;
    }
    else {
      printf ("Erreur: valeur hors limites d'un int.\n") ;
      return 1 ;
    }
  }
  else {
    printf ("Erreur: argument chaîne vide.\n") ;
    return 1 ;
  }

  return 0 ;
}
```

```
$ gcc -Wall strotest.c -o strotest.x
$ ./strotest.x
Erreur: 1 argument numérique attendu.
```

```
$ ./strotest.x ""
Erreur: argument chaîne vide.
$ ./strotest.x 2147483647
C'est bon, the_i vaut 2147483647
$ ./strotest.x 2147483648
Erreur: valeur hors limites d'un int.
$ ./strotest.x -2147483648
C'est bon, the_i vaut -2147483648
$ ./strotest.x -2147483649
Erreur: valeur hors limites d'un int.
```

La toute première chose à vérifier est la présence de l'argument sur la ligne de commande. Ensuite, le résultat de `strtol` est stocké dans `tmp_long`, un `long int`. Avant de considérer la valeur de cette variable, nous vérifions que la chaîne fournie à `strtol` représentait effectivement un entier en s'assurant que le caractère pointé par `endscan` est `'\0'`. Si tel est bien le cas, pour terminer il faut vérifier que la valeur récupérée dans `tmp_long` est comprise entre les deux bornes `INT_MAX` et `INT_MIN` (définies dans le fichier d'en-tête `<limits.h>`) qui représentent respectivement la valeur la plus grande et la plus petite représentable sur 32 bits (la taille d'un `int`).

Les différentes exécutions testent l'absence d'argument, un argument présent mais qui est la chaîne vide (obtenue en mettant explicitement des `"` avec rien entre), la valeur `INT_MAX` (i.e. 2147483647), la valeur `INT_MAX + 1`, la valeur `INT_MIN` et pour finir `INT_MIN - 1`.

11.3.2 Conversion vers réels

La fonction `strtof` permet de convertir vers un `float`, `strtod` vers un `double`. Leurs prototypes sont les suivants :

```
float strtof (char *nptr, char **endptr) ;
double strtod (char *nptr, char **endptr) ;
```

Puisque la notion de base d'écriture n'a pas de signification pour des réels, ceux-ci étant naturellement écrits en base 10, ces fonctions n'ont que deux arguments `nptr` et `endptr` dont les rôles sont exactement identiques à ceux de leurs homonymes pour la conversion vers entiers décrite en section 11.3.1.

11.4 En résumé

- La fonction `main` peut recevoir des arguments en provenance de la ligne de commande qui a invoqué le programme.
- Dans ce cas, elle a la forme `int main (int argc, char *argv[]) { ... }`
- `argc` est le nombre d'arguments présents sur la ligne de commande invoquée (y compris le chemin du programme).
  `gcc -Wall foo.c -o foo.x` ⟶ 5 arguments.
- `argv` est un tableau de chaînes de caractères qui contient tous les arguments présents sur la ligne de commande invoquée (y compris le chemin du programme).
  `gcc -Wall foo.c -o foo.x` ⟶ `"gcc"`, `"-Wall"`, `"foo.c"`, `"-o"` et `"foo.x"`.
- Avant d'accéder aux arguments, il faut vérifier qu'ils sont présents (tester `argc` avant d'accéder à `argv[`i`]`).
- `argv[0]` existe toujours : c'est le chemin complet du programme tel qu'invoqué sur la ligne de commande.
- Quand les arguments sont des chaînes représentant des valeurs numériques, il faut les convertir en « nombres » (entiers ou réels).
- La bibliothèque standard de C fournit des fonctions de conversion qui nécessitent le fichier d'en-tête `<stdlib.h>`.
- Les fonctions simples mais ne permettant pas la détection d'erreur de conversion sont `atoi`, `atol`, `atoll`, `atof`.
- Les fonctions avancées permettant la détection d'erreur de conversion sont `strtol`, `strtoll`, `strtof`, `strtod`.
- Ces fonctions avancées utilisent le passage d'argument par adresse.
- Les constantes `INT_MIN` et `INT_MAX` définissent les valeurs minimale et maximale représentables sur 32 bits. Elles nécessitent le fichier d'en-tête `<limits.h>`.

Chapitre 12

Allocation dynamique

12.1 Motivations

12.1.1 Taille trop figée

La création de variables, globales ou locales, de types scalaires ou tableaux statiques répond à des besoins de stockage d'information dont la taille est connue au moment de la compilation. Lorsque nous déclarons `int t[50] ;` la taille de l'espace alloué est fixé avant même l'exécution (50×4 octets).

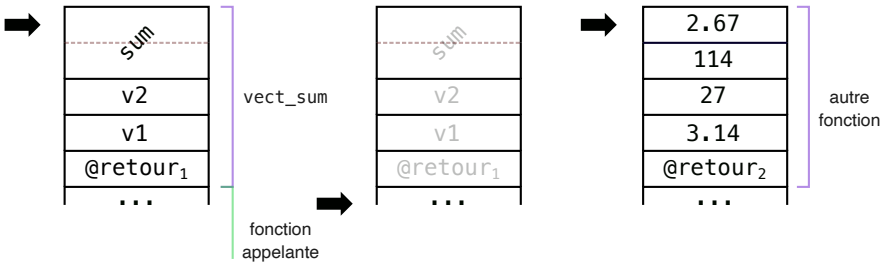
Si la taille du tableau `t` dépend désormais d'une information calculée lors de l'exécution, par exemple en fonction d'une entrée de l'utilisateur, il est nécessaire de disposer d'un autre mécanisme d'allocation de mémoire qui sera effectué lors de l'exécution du programme.

12.1.2 Retourner un tableau

Comme présenté en section 8.1, les variables locales sont créées dans la pile d'exécution. Les tableaux statiques définis dans une fonction ne font pas exception à la règle. Lorsque nous écrivons le programme (incorrect) suivant où une fonction `vect_sum` retourne la somme de deux vecteurs :

```
float* vect_sum (float v1[2], float v2[2]) {
  float sum[2] ;
  sum[0] = v1[0] + v2[0] ;
  sum[1] = v1[1] + v2[1] ;
  return sum ;
}
```

le tableau résultat `sum` est une variable locale. Il est donc créé sur la pile d'exécution. Lorsque la fonction termine, elle retourne donc l'adresse où débute ce tableau, donc une adresse située dans la pile.



L'exécution reprend dans la fonction appelante qui fera peut-être d'autres appels de fonctions qui viendront écrire leurs propres informations dans la pile. De ce fait, l'adresse retournée par `vect_sum` ne contiendra plus les valeurs qu'elle y avait mis.

Pour aller plus loin

La réalité est plus complexe à cause du protocole d'appel normalisé qui impose l'utilisation de registres pour passer les arguments aux fonctions et à cause des possibles optimisations effectuées par les compilateurs pour maximiser l'utilisation des registres. Ainsi, dans des cas simples (comme l'exemple jouet présenté), la pile ne sera pas forcément utilisée aussi naïvement et le programme pourra « tomber en marche ». Néanmoins, dans de véritables programmes, retourner une adresse située dans la pile est toujours une erreur.

En résumé, pour retourner un tableau qui devra survivre à la fin d'une fonction, il est nécessaire de disposer d'un mécanisme d'allocation mémoire qui retourne une adresse qui reste valide en dehors de cette fonction.

12.2 Allocation dans le tas

Le tas ("heap" en anglais) est une zone mémoire attribuée au programme par le système d'exploitation juste avant le début de son exécution. Cette zone mémoire appartient au programme et constitue une réserve mise à sa disposition. Cette zone peut s'agrandir au besoin de manière « arbitrairement » grande. Néanmoins, la mémoire globale de la machine étant une ressource finie et partagée avec tous les programmes en cours d'exécution, le « arbitrairement » a ses limites.

Chaque programme doit donc puiser dans sa réserve de mémoire lorsqu'il en a besoin et restituer la mémoire dont il n'a plus besoin pour éviter la saturation. Ce mécanisme d'allocation / restitution permet donc de recycler la mémoire entre différents instants d'exécution.

La bibliothèque standard de C fournit deux fonctions fondamentales pour gérer cet espace mémoire : une pour demander (allouer) de la mémoire, une pour en libérer.

12.2.1 Allocation : `malloc`

La fonction permettant de demander une zone de mémoire est `malloc` et nécessite le fichier d'entête `<stdlib.h>`. Son prototype est le suivant (que vous pouvez obtenir avec la commande `man 3 malloc` dans un terminal) :

```
void* malloc (size_t size) ;
```

où le paramètre `size` est un entier (`size_t` est un alias d'un type entier non signé) représentant le nombre d'octets demandés. Le type de retour est `void*` qui représente un pointeur « générique », donc vers n'importe quel type. Ce pointeur est l'adresse du début de la zone que `malloc` a attribué, si elle a effectivement trouvé une zone libre de la taille demandée.

Si `malloc` ne trouve pas de zone libre suffisante pour la taille demandée, elle retourne l'adresse invalide dénotée par la constante symbolique `NULL`.

Important

Appeler `malloc` demande de la mémoire. Comme toute demande, elle peut-être refusée, ce qui est signalé par le retour du pointeur `NULL`. Ainsi, après une allocation dynamique, il faut vérifier que l'adresse retournée est différente de `NULL` avant d'aller y écrire.

Une fois la mémoire allouée pour n objets de type `ty_t`, nous disposons d'une zone consécutive de n éléments de type `ty_t`. Nous pouvons donc accéder aux n éléments comme dans un tableau standard. Bien entendu, si $n = 1$ il ne s'agit pas d'un tableau (ou d'un tableau trivial de 1 case) et nous pouvons y accéder avec l'opérateur de déréférencement `*` (cf. section 7.2.3).

Puisque l'allocation d'objets de type `ty_y` nous renvoie une adresse où il y a la place de stocker des objets de type `ty_y`, le pointeur dans lequel mémoriser cette adresse doit être un `ty_y*`. De manière totalement mécanique, le `sizeof` utilisé pour définir la taille de la zone allouée devra être un `sizeof (ty_t)`. Nous avons donc le motif suivant à chaque allocation :

```
ty_t *tab = malloc (n * sizeof (ty_t)) ;
```

```
...
int *tab = malloc (256 * sizeof (int)) ;
if (tab == NULL) { ... /* Gérer l'absence de mémoire. */ }
for (int i = 0; i < 256; i++) tab[i] = i * 10 ;
...
```

Spécifier la taille demandée en octets nécessite de connaître *a priori* la taille en octets de chaque type individuel. Si nous voulons un tableau de 256 `int`, il faut en réalité 256×4 octets. Devoir apprendre ces tailles est peu commode, sachant qu'en fonction des architectures, certains types ont des tailles différentes. De plus, lorsque nous voudrons travailler avec des types composites comme les structures (cf. chapitre 14), il sera hors de question de devoir additionner manuellement la taille de chaque élément de la structure. Ce serait très pénible et très fragile : changer la définition d'une structure reviendrait à changer sa taille, donc à modifier le programme à de multiples endroits. Pire encore, la manière dont le compilateur agence les éléments d'une structure est à sa discrétion : impossible d'être certain à l'avance de ses choix (sauf à connaître parfaitement son compilateur, à faire des tests, à maîtriser parfaitement les options de compilation).

L'opérateur `sizeof` permet de s'affranchir de tous ces problèmes et retourne la taille en octets d'un type. Nous pouvons remarquer dans l'exemple qui suit, que la taille de la `struct foo_t` n'est pas la somme des tailles individuelles de ses champs.

```
#include <stdio.h>

enum dir_t { nord, sud, est, ouest } ;
struct foo_t { char c ; int i ; int j ; } ;

int main () {
  printf ("char: %ld\n", sizeof (char)) ;
  printf ("int: %ld\n", sizeof (int)) ;
  printf ("float: %ld\n", sizeof (float)) ;
  printf ("long: %ld\n", sizeof (long int)) ;
  printf ("char*: %ld\n", sizeof (char*)) ;
  printf ("double*: %ld\n", sizeof (double*)) ;
  printf ("dir_t: %ld\n", sizeof (enum dir_t)) ;
  printf ("foo_t: %ld\n", sizeof (struct foo_t))
      ;
  return 0 ;
}
```

```
$ gcc -Wall size.c -o size.x
$ ./size.x
char: 1
int: 4
float: 4
long: 8
char*: 8
double*: 8
dir_t: 4
foo_t: 12
```

Pour aller plus loin

L'opérateur `sizeof` peut également être utilisé pour obtenir la taille d'une variable. Elle retourne alors la taille de son type. Dans le cas d'un tableau statique, la taille est bien celle du tableau complet. C'est totalement différent de celle d'un pointeur qui est immuable, indépendante de la zone mémoire que l'on pourrait avoir allouée et dont on aurait stocké l'adresse dans ce pointeur. →

```c
#include <stdio.h>

int main () {
  int i = 5 ;
  float t[10] ;
  char *s ;
  printf ("i: %ld\n", sizeof (i)) ;
  printf ("t: %ld\n", sizeof (t)) ;
  printf ("s: %ld\n", sizeof (s)) ;
  return 0 ;
}
```

```
$ gcc -Wall size2.c -o size2.x
$ ./size2.x
i: 4
t: 40
s: 8
```

Important

On entend parfois l'argument suivant pour s'autoriser à ne pas vérifier le résultat de `malloc` :
« *C'est inutile, sur les machines modernes il y a toujours assez de mémoire* ».

Nous prétendons que cet argument est fallacieux pour au moins deux raisons.

- D'une part, nous ne programmons pas toujours pour des ordinateurs « confortables » avec 32 ou 64 Go de mémoire. Bon nombre de programmes tournent sur des architectures (éventuellement plus ou moins embarquées) avec des contraintes mémoire fortes : à peine quelques Ko ou Mo de mémoire. Et là, l'épuisement de la mémoire disponible arrive nettement plus vite !
- D'autre part, la mémoire est une ressource au même titre que de nombreuses autres soumises à demande avec possibilité de refus : fichiers, sockets, etc. Pourquoi vérifierait-on le succès d'ouverture d'un fichier et pas celui d'une allocation mémoire ? Les bonnes habitudes doivent s'exercer pour ne pas s'oublier.

Important

Un pointeur (i.e. variable servant à mémoriser des adresses – cf. section 7.2) est utilisé pour récupérer l'adresse retournée par `malloc`. C'est cette fonction qui initialise le pointeur avec une zone mémoire à laquelle il est légal d'accéder. Juste déclarer :

```
int *t ;
```

ne signifie pas que `t` indique une zone où il est possible d'accéder. C'est juste un pointeur non initialisé : « il n'y a pas de mémoire au bout ». Écrire ou lire à l'adresse que contient `t` avec `*t` aura toutes les chances de faire planter le programme avec un *Segmentation fault* ou un *Bus error* (cf. section 7.2.2).

12.2.2 Libération : `free`

La fonction permettant de restituer une zone de mémoire devenue inutile est `free` et nécessite le fichier d'en-tête `<stdlib.h>`. Son prototype est le suivant (que vous pouvez obtenir avec la commande `man 3 free` dans un terminal) :

```
void free(void *ptr) ;
```

où le paramètre `ptr` est un pointeur « générique », donc vers n'importe quel type, qui représente le début de la zone mémoire à restituer.

Important

L'adresse fournie à `free` pour libération doit obligatoirement avoir été obtenue par une allocation dynamique avec `malloc` (soit directement soit via une fonction qui a appelé `malloc` et retourné →

cette adresse d'une manière ou d'une autre). Dit autrement, on n'appelle pas `free` avec l'adresse d'un tableau statique ou avec une adresse en plein milieu d'un tableau dynamique.

```
bool do_something (int nb) {
  double *tab = malloc (nb * sizeof (double)) ;
  if (tab == NULL) return false ;
  ...
  ... tab[i] ...
  free (tab) ;
  ... /* Plus aucune utilisation de tab. */
  ...
  return true ;
}
```

Dès lors que nous avons appelé `free`, la zone mémoire est retournée dans le tas et redevient disponible pour une future allocation. De ce fait, il ne faut surtout pas tenter d'accéder à cette zone après l'avoir libérée, en continuant à utiliser le pointeur qui la désignait. Cela revient à l'avoir libérée trop tôt.

Il est courant que la mémoire allouée dans une fonction ne soit pas libérée dans cette fonction. C'est même l'effet recherché pour être capable de retourner un tableau sans tomber dans l'erreur décrite au début de ce chapitre en section 12.1.2. Nous voulons que la mémoire de ce tableau survive à la fonction pour que celle qui l'a appelée puisse utiliser ce tableau.

Néanmoins, il faudra à un certain moment que « quelqu'un» (i.e. une instruction du programme) décide de libérer cette zone mémoire quand elle ne sera plus utilisée. Déterminer la durée de vie de la zone est entièrement à la charge du développeur qui doit avoir une vue complète de l'architecture logicielle de son programme.

La fonction `vect_sum` qui était incorrectement écrite en début de ce chapitre peut désormais l'être de manière sûre avec une allocation dynamique.

```
#include <stdlib.h>

float* vect_sum (float v1[2], float v2[2]) {
  float *sum = malloc (2 * sizeof (float)) ;
  if (sum == NULL) return NULL ;
  sum[0] = v1[0] + v2[0] ;
  sum[1] = v1[1] + v2[1] ;
  return sum ;
}
```

12.3 Quelques remarques

12.3.1 Ménager la pile (2)

Dans la section 8.2.5, nous avons dit que la pile était une zone mémoire de taille relativement modeste. Dans le cas où, localement à une fonction, nous avons besoin d'une « grande » zone de mémoire, il peut être nécessaire d'avoir recours à l'allocation dynamique pour ne pas saturer la pile. Et ce, malgré le fait que nous savons statiquement que cette zone mémoire est temporaire, de taille constante et n'aura pas besoin de survivre à la fonction.

Les deux programmes ci-dessous allouent un tableau de 5000000 `int`. Le premier le fait localement, donc sur la pile et plante. Le second passe par une allocation dynamique de la même taille et se porte comme un charme.

```c
#include <stdio.h>

int main () {
  int t[5000000] ;
  printf ("Still alive!\n") ;
  return 0 ;
}
```

```
$ gcc toobig.c
$ ./a.out
Segmentation fault: 11
```

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

int main () {
  int *t = malloc (5000000 * sizeof (int))
      ;
  if (t != NULL) free (t) ;
  printf ("Still alive!\n") ;
  return 0 ;
}
```

```
$ gcc nottoobig.c
$ ./a.out
Still alive!
```

12.3.2 Allocations dynamiques partout ?

Puisque l'allocation dynamique permet de gérer la mémoire en temps réel, en adaptant la consommation selon le besoin du moment, en épargnant la pile, pourquoi ne pas utiliser uniquement ce mécanisme ?

La première réponse est certainement celle qui vient immédiatement à l'idée de toute personne qui programme en C : « *Parce que c'est compliqué (voire pénible) à mettre en œuvre !* ». En effet, il faut appeler `malloc`, vérifier son succès, puis appeler `free` un jour.

La seconde réponse est que l'allocation dynamique est un mécanisme nettement plus coûteux en temps d'exécution. La gestion de la mémoire disponible / occupée est un algorithme non trivial qui nécessite beaucoup plus de travail que déplacer le marqueur de pile d'exécution pour réserver la place pour des variables locales. Et il en est de même pour `free` qui ne se contente pas de déplacer le même marqueur en sens inverse.

Donc, effectivement, on réserve cette mécanique aux cas (toutefois très courants dans programmes) où il n'y a pas le choix.

12.3.3 Ne pas faire `free`

Que se passe-t-il si un programme ne libère pas la mémoire dont il n'a plus besoin ?

Dans le principe, si ce programme fait des allocations répétées suffisamment longtemps, sans jamais rien libérer, il va finir par saturer la mémoire et ralentir le système d'exploitation sur lequel il tourne. Sur des machines modernes où il y a un système d'exploitation, en cas de manque aigu de mémoire, ce dernier finira par tuer des processus afin de se sauver lui-même (le système est le garant de la « survie » de la machine).

Dans les faits, si le programme n'est pas un ogre vorace en mémoire, quand il se termine, le système d'exploitation s'occupera de se restituer l'espace mémoire non libéré. Lorsqu'un programme termine, le système d'exploitation reprend la main et se charge de faire un ménage complet dans les ressources attribuée à ce programme.

Néanmoins, ce n'est pas une raison pour prendre l'habitude de compter sur autrui pour faire ce ménage. Si tous les programmes restituaient leurs ressources inutiles à leur toute fin, l'empreinte mémoire globale serait inutilement exagérée, sachant que certains programmes peuvent s'exécuter pendant fort longtemps (heures, jours).

12.4 En résumé

- Lorsque la taille d'une zone mémoire n'est pas connue à la compilation, il faut faire une allocation dynamique.
- Lorsqu'un tableau (une zone mémoire) doit être retournée par une fonction (donc y survivre), il faut faire une allocation dynamique.
- Lorsque l'on a besoin d'allouer une zone mémoire, même de taille fixe, même ne devant pas survivre à sa fonction, mais de très grande taille, il faut faire une allocation dynamique.
- Pour demander dynamiquement de la mémoire il faut appeler
  ```
  void* malloc (size_t size) ;
  ```
- `size` est le nombre d'octets demandés.
- L'opérateur `sizeof` permet de connaître le nombre d'octets d'une valeur d'un certain type.
  ```
  sizeof (int)
  ```
- Le pointeur utilisé pour récupérer le retour de `malloc` doit être sur le même type que celui utilisé dans `sizeof` :
  ```
  bidule_t *tab = malloc (... * sizeof (bidule_t)) ;
  ```
- `malloc` retourne l'adresse de début de la zone allouée en cas de succès, sinon `NULL`.
- Avant d'accéder à l'adresse retournée par `malloc` il faut vérifier qu'elle n'est pas `NULL`.
- Juste déclarer `int *tab ;` crée seulement un pointeur non initialisé, sans allouer de mémoire.
- Pour restituer de la mémoire devenue inutile, il faut appeler
  ```
  void free(void *ptr) ;
  ```
- L'adresse passée à `free` doit avoir été obtenue par `malloc` (début de la zone).
- Une fois une zone libérée, il ne faut plus y accéder via le pointeur qui la désignait.
- Une zone allouée doit être libérée une et une seule fois.

Chapitre 13

Entrée au clavier : `scanf`

Dans le chapitre 11, nous avons vu comment injecter des informations en entrée d'un programme pour piloter son exécution par l'intermédiaire des arguments de ligne de commande. Nous y avons clamé que cela présentait le grand avantage de spécifier ces informations au lancement du programme plutôt que de devoir attendre qu'il les demande au moment où il en a besoin, ce qui obligerait l'utilisateur à se trouver devant son clavier à ce moment précis.

Cependant, nous avons aussi dit que les arguments de ligne de commande étaient raisonnables tant qu'il n'y en avait pas un nombre exagéré.

Dans le cas où beaucoup de données doivent être injectées dans un programme, il est possible de faire appel à un mécanisme de lecture sur l'entrée standard (i.e. le « clavier au terminal ») qui présente de surcroît l'intérêt de permettre des saisies selon différents formats (entier, réels, chaînes, caractères, etc.) et respectant une certaine structure. Cette lecture sur l'entrée standard revêt encore plus d'intérêt lorsqu'elle est exploitée, non pas avec un utilisateur qui tape au clavier, mais en utilisant le mécanisme de redirection d'un fichier dans l'entrée standard.

13.1 Interlude : redirection de l'entrée standard

Faisons un petit détour en dehors du langage C pour expliquer ce processus qui s'avérera fort utile pour injecter des données dans un programme grâce à l'interprète de lignes de commande.

Sous Unix, le programme `bc` est une calculatrice interactive. Une fois lancé, l'utilisateur entre les calculs à effectuer par le clavier dans le terminal.

```
$ bc
>>> 3 * 4 + 7
19
>>> 22 / 7
3
>>> scale = 4
>>> 22 / 7
3.1428
>>> scale = 10
>>> 2 * sqrt (2)
2.8284271246
```

Il est possible d'écrire dans un fichier texte (disons `calculs.txt`) les calculs à la place de les entrer au clavier. Cela permet de les conserver pour les rejouer ou de les modifier si nous nous sommes trompés sans devoir tout reprendre depuis le début. Pour injecter le contenu de ce fichier, il suffit d'invoquer la commande sous la forme :

```
bc < calculs.txt
```

où le chevron (signe « inférieur ») est suivi du nom du fichier à injecter.

```
$ more calculs.txt
3 * 4 + 7
22 / 7
scale = 4
22 / 7
scale = 10
2 * sqrt (2)
$ bc < calculs.txt
19
3
3.1428
2.8284271246
```

Ce mécanisme de redirection fonctionne avec n'importe quel programme qui attend des informations sur son entrée standard. Ainsi, ceux que nous écrirons ne dérogeront pas à la règle.

13.2 Les bases de `scanf`

Important

La fonction `scanf` n'est pas considérée comme robuste et peut être source de vulnérabilités en terme de sécurité. Pour autant, il est bon d'en connaître les usages basiques et ses limitations.

La fonction `scanf` est duale de la fonction d'affichage `printf`. En particulier, elle utilise la même notion de format, donc de séquences avec des %. Avec `printf`, pour afficher un entier il faut utiliser `%d`, pour un flottant `%f`, pour une chaîne `%s`, etc. Avec `scanf`, pour lire un entier, il faut aussi utiliser `%d`, pour un flottant aussi `%f`, pour une chaîne aussi `%s`, etc. Fort logiquement, elle nécessite également l'inclusion du fichier d'en-tête `<stdio.h>`.

De la même manière que `printf` permet d'afficher plusieurs valeurs d'un coup, `scanf` permet de lire plusieurs valeurs d'un coup. Puisqu'une fonction ne peut retourner qu'une seule valeur, `scanf` utilise le mécanisme de passage d'arguments par adresse présenté en section 8.2. Ainsi, pour chaque valeur que nous souhaitons lire avec `scanf`, après la chaîne de format qui définit le nombre et le type des valeurs que nous souhaitons lire, figurera l'adresse d'une variable de ce type.

Le programme ci-dessous attend l'entrée d'un entier puis un réel. `scanf` découpe le flot d'entrée selon les caractères d'espacement, donc les espaces, les retours à la ligne et les tabulations. Et s'il y en a plusieurs entre deux éléments à lire, il les ignore comme le montre la seconde exécution du programme.

Lorsque l'entrée est effectuée depuis le terminal (i.e. sans utiliser le mécanisme de redirection) c'est le retour à la ligne final qui déclenche le retour de `scanf` une fois que le nombre de valeurs attendu a été lu. C'est pourquoi, lors de la seconde exécution, après avoir entré « 10 » le programme est resté en suspens puisque `scanf` attendait encore le réel.

```c
#include <stdio.h>

int main () {
  int amount ;
  float price ;

  scanf ("%d %f", &amount, &price)
      ;
  printf ("Prix : %.2f\n", amount
      * price) ;
  return 0 ;
}
```

```
$ gcc -Wall scan.c -o scan.x
$ ./scan.x
100
5.55
Prix : 555.00
$ ./scan.x
10

          6
Prix : 60.00
```

Les principales séquences % utilisables dans un `scanf` sont récapitulées dans le tableau suivant. Pour de plus amples information, la page de manuel est disponible 7j/7 en invoquant `man 3 scanf`.

`%d`	`int` en décimal
`%ld`	`long int` en décimal
`%u`	`unsigned int` en décimal
`%lu`	`long unsigned int` en décimal
`%s`	chaîne de caractères
`%c`	`char`
`%f`	`float`
`%lf`	`double`

Quelques différences de comportement sont à noter par rapport aux séquences utilisées avec `printf` présentées en section 6.2.

Important

Lors de la lecture de réels, la présence ou l'absence du `l` dans `%lf` ou `%f` est primordiale. Rappelons-nous que `scanf` utilise le passage d'arguments par adresse, donc elle va aveuglement écrire aux adresses reçues en argument ce qu'elle a récupéré.

```c
#include <stdio.h>

int main () {
  float a_float ;
  double a_double ;
  scanf ("%lf", &a_float) ;      /* Erreur %lf versus float. */
  printf ("%f\n", a_float) ;
  scanf ("%f", &a_double) ;      /* Erreur %f versus double. */
  printf ("%f\n", a_double) ;
  return 0 ;
}
```

```
$ gcc -Wall badscan.c -o badscan.x
badscan.c: In function 'main':
badscan.c:6:13: warning: format '%lf' expects argument of type 'double *', but argument 2 has
 type 'float *' [-Wformat=]
   scanf ("%lf", &a_float) ;      /* Erreur %lf versus float. */
          ~~^   ~~~~~~~~
          %f
badscan.c:8:12: warning: format '%f' expects argument of type 'float *', but argument 2 has
 type 'double *' [-Wformat=]
   scanf ("%f", &a_double) ;      /* Erreur %f versus double. */
          ~~^   ~~~~~~~~~
          %lf
$ ./badscan.x
3.14
126443839488.000000
4.5
0.000000
```

Si nous lui donnons l'adresse d'un `float` (4 octets) avec un `%lf`, alors elle ira écrire 8 octets à cette adresse. Il en résulte deux effets. D'une les 4 octets du `float` constitueront une information partielle, donc un flottant corrompu lorsque nous le lirons plus tard. D'autre part, sans doute pire, les 4 octets « en trop » iront silencieusement écraser les variables se trouvant à côté du `float`.

Si nous lui donnons l'adresse d'un `double` (8 octets) avec un `%f`, alors elle ira seulement écrire 4 octets, laissant les 4 autres de la variable réceptrice inchangés. De ce fait, lorsque nous lirons le `double` plus tard, il sera incomplet donc représentera une valeur corrompue.

Notons que dans les deux cas, un avertissement est généré, qu'il convient donc (comme pour tous les autres), de corriger impérativement de toute urgence.

De manière moins grave (car signalé par une erreur à la compilation), il n'y a pas de moyen de contrôler le nombre de décimales lors de la lecture d'un réel. Ainsi, `scanf ("%.2f", &a_float) ;` est incorrect.

13.3 Lecture d'un caractère

Parmi les séquences % présentées dans la section précédente figurait le `%c` pour lire un seul caractère (i.e. un `char`). Cette séquence présente une subtilité de comportement. Considérons le programme suivant.

```
#include <stdio.h>

int main () {
  char c ;
  scanf ("%c", &c) ;
  printf ("Lu : <%c>\n", c) ;
  return 0 ;
}
```

Par défaut, `scanf` va lire le prochain caractère quel qu'il soit. En particulier, si c'est un espace, ou un retour à la ligne, elle le lira et le retournera. Ce sera un caractère comme un autre, il ne sera pas ignoré contrairement à ce qui se produit lorsque nous utilisons d'autres séquences %. Nous avons demandé de lire un `char` : ça lit le prochain `char` disponible.

```
$ gcc scanchar.c -o scanchar.x
$ ./scanchar.x
    Aloha
Lu : < >
```

Si nous souhaitons lire un « vrai » caractère, c'est-à-dire ce que l'on pourrait qualifier de « lettre », en ignorant tous les « blancs » le précédant (i.e. espaces, retours à la lignes, tabulations), il faut précéder le `%c` par un espace. Cela indique à `scanf` d'ignorer les caractères d'espacement.

```
#include <stdio.h>

int main () {
  char c ;
  scanf (" %c", &c) ;
  printf ("Lu : <%c>\n", c) ;
  return 0 ;
}
```

```
$ gcc scanchar2.c -o scanchar2.x
$ ./scanchar2.x
    Aloha
Lu : <A>
```

13.4 Lecture d'une chaîne de caractères

Comme exposé dans le chapitre 10, une chaîne de caractères est un tableau de `char` terminé par le caractère `'\0'`. Comme expliqué dans le chapitre 9, un tableau est représenté par un pointeur.

Pour donner à `scanf` un endroit où stocker la chaîne qu'elle va lire, il est nécessaire d'avoir défini une chaîne temporaire, donc un tableau temporaire de `char`. Puisque ce tableau est directement désigné par un pointeur, il n'y a pas besoin d'utiliser (et il ne faut surtout pas) l'opérateur d'extraction d'adresse `&` pour fournir à `scanf` l'endroit où stocker ce qu'elle aura lu : le nom du tableau temporaire est exactement ce qui est requis.

```
#include <stdio.h>

int main () {
  char nom[256] ;
  scanf ("%s", nom) ;
  printf ("Aloha %s\n", nom) ;
  return 0 ;
}
```

Important

En règle générale, il est impossible de connaître à l'avance quelle sera la longueur maximale de l'entrée utilisateur, donc de décider de la taille idéale pour la chaîne temporaire. C'est la faiblesse fondamentale de la fonction `scanf`.

Il n'y a pas d'autre solution que de choisir une taille arbitraire que nous espérerons suffisamment grande. Si l'estimation est incorrecte, qu'une chaîne plus longue est saisie, malheureusement `scanf` écrasera silencieusement la zone mémoire dans laquelle le débordement aura lieu.

C'est à peu près le seul endroit où il est possible de tolérer une taille de chaîne figée dans un programme. Et c'est bien parce que nous n'avons pas le choix si nous utilisons `scanf` pour lire une chaîne de caractères.

Pour illustrer ce qui peut survenir en cas de débordement, considérons le programme suivant dans lequel nous définissons une chaîne temporaire `nom` de 256 caractères, ainsi qu'une variable `i` dont le seul but est d'être affichée. Nous effectuons ensuite une lecture de chaîne avec `scanf` dont le résultat sera stocké dans le tableau temporaire `nom`.

```
#include <stdio.h>

int main () {
  int i = 25 ;
  char nom[256] ;
  scanf ("%s", nom) ;
  printf ("Aloha %i %s\n", i, nom) ;
  return 0 ;
}
```

À l'exécution, nous entrons une chaîne de longueur (300) supérieure à la taille du tableau `nom`. L'effet le plus visible est l'erreur de segmentation qui interrompt le programme. C'est une erreur typique quand un programme essaye de lire ou d'écrire à une adresse où il n'a pas le droit.

```
$ gcc -Wall overf.c -o overf.x
$ ./overf.x
troplongggggg
```

```
ggggggg
```

Au terminal, il est possible de signaler une fin de saisie en pressant Ctrl + d : c'est l'équivalent d'une fin de fichier, mais au clavier. Si `scanf` rencontre une telle fin de saisie, elle retourne la valeur `-1` (symboliquement désignée par la constante `EOF`). Cela permet de détecter que la saisie a été interrompue, non pas à cause d'une entrée incorrecte, mais à cause d'une interruption du flot d'entrée.

13.6 Formats complexes

Le format fourni à `scanf` peut être plus compliqué qu'une simple suite de `%` afin d'effectuer des saisies structurées autour d'une partie fixe du flot d'entrée. Dit autrement, dans la chaîne de format, il est possible d'insérer du texte qui formera le motif dans lequel extraire les sous-parties correspondant aux différents `%`. Dans ce texte, les « blancs » (espaces, retours à la ligne, tabulations) serviront de délimiteurs, qu'il y en ait un ou plusieurs dans le format ou dans le flot d'entrée.

```c
#include <stdio.h>

int main () {
  int i ;
  float f ;
  scanf ("%d et          ensuite %f
      ", &i, &f) ;
  printf ("Lus : %d %f\n", i, f) ;
  return 0 ;
}
```

```
$ gcc -Wall smarter.c -o smarter.x
$ ./smarter.x
4 et  ensuite
 3.14
```

Dans cet exemple, le format attend un entier, un morceau de texte qui comporte les mots « et » puis « ensuite » et enfin un flottant. Le flot d'entrée a bien cette forme, avec quelques espaces et même un retour à la ligne qui ont servi à délimiter les « mots ». Cela illustre le fait que le nombre de caractères d'espacement et leur type n'ont pas d'importance, dans le format comme dans le texte entré. Le tout c'est qu'il y en ait au moins un.

13.7 En résumé

- L'entrée d'information au clavier est réalisé avec la fonction `scanf`.
- Elle nécessite l'inclusion du fichier d'en-tête `<stdio.h>`.
- `scanf` est la duale de `printf`.
- Elle utilise aussi la notion de format, donc de séquences %.
- Les formats les plus courants sont :
  - `%d` pour les entiers,
  - `%f` pour les réels (`float`),
  - `%lf` pour les réels (`double`),
  - `%s` pour les chaînes,
  - `%c` pour les caractères.
- La différence entre `%f` et `%lf` est importante (contrairement à `printf`).
- Elle utilise le passage d'arguments par adresse pour « retourner » plusieurs valeurs lues.
- À chaque % doit correspondre positionnellement l'adresse d'une variable où stocker le résultat de la lecture.
- Pour lire une chaîne de caractères, il faut avoir défini une chaîne temporaire de taille arbitraire.
- Une chaîne étant un tableau, pas besoin de `&` pour la passer à `scanf`.
- Si la chaîne temporaire est trop petite, il y aura débordement en mémoire, donc corruption.
- Pour lire un `char` en ignorant les « blancs » devant, il faut rajouter un espace à `%c` : `" %c"`.
- `scanf` retourne le nombre de séquences % lues avec succès.
- En cas de fin du flot d'entrée, elle retourne `EOF` (qui vaut -1).
- `scanf` permet des formats plus complexes que des séquences de % pour des lectures selon un motif.

Chapitre 14

Structures

Les tableaux présentés dans le chapitre 9 permettent uniquement de grouper des valeurs d'un même type. Lorsque nous avons besoin d'agréger des données de types différents, il faut faire appel au type structure. En section 14.4.3, nous verrons également que ce type peut présenter un intérêt certain pour grouper des valeurs de même type.

14.1 Définition de type structure

Une structure désigne un nouveau type qui groupe des données en les désignant par des *champs* nommés. Elle est introduite par le mot-clé `struct` suivi d'un nouveau nom de type. Vient ensuite une énumération de noms avec leur type qui définissent l'ensemble des informations regroupées dans une valeur de ce type structure. Cette énumération est placée entre accolades et termine par un point-virgule.

```
struct nom-struct {
  type-champ nom-champ ;
  type-champ nom-champ ;
  ...
} ;
```

Considérons la représentation d'une position GPS, par exemple 48°42'39.1"N 2°13'09.4"E. Elle est constituée de différentes informations de différentes natures :

- une orientation de latitude (Nord ou Sud),
- des degrés et minutes de latitude (entiers),
- des secondes et fractions de latitude (flottants),
- une orientation de longitude (Est ou Ouest),
- etc. comme pour la latitude mais pour la longitude...

La latitude et la longitude ne pouvant prendre chacune que 2 valeurs, il est élégant de définir un `enum` pour chacune. Nous regroupons ensuite latitude, longitude et degrés, minutes, secondes pour chacune dans une `struct` car les types de ces données sont différents : nous ne pouvons pas utiliser un tableau.

```
enum lat_orient_t { La_North, La_South } ;
enum long_orient_t { Lo_West, Lo_East } ;

struct gps_loc_t {
  enum lat_orient_t lao ;
  unsigned int la_deg, la_min ;
  float la_sec ;
  enum long_orient_t loo ;
  unsigned int lo_deg, lo_min ;
```

```
  float lo_sec ;
} ;
```

Important

Le nouveau type défini a pour nom `struct gps_loc_t` et non pas `gps_loc_t`. Le mot-clé `struct` fait intrinsèquement partie du nom du type.

Une définition de `struct` crée un nouveau type. Il est donc possible de créer des tableaux de `struct`, des pointeurs sur `struct`, etc. Et bien entendu, il possible d'avoir un champ de structure qui soit d'un autre type `struct` déjà défini : nous avons alors des structures *imbriquées*. Un type permettant de représenter des points d'intérêt pourrait être défini de la façon suivante, où le champ `position` a bien pour type `struct gps_loc_t` et non `gps_loc_t`.

```
struct poi_t {
  char *name ;
  char *descrip ;
  struct gps_loc_t position ;
  bool public ;
} ;
```

Important

Il est possible de définir des structures imbriquées, mais pas des structures récursives. Dit autrement, une définition de `struct t` ne peut pas avoir un champ de ce même type `struct t`.

Une telle définition n'aurait aucun sens puisqu'elle serait de taille infinie. En effet, un de ses champs aurait pour champs... ceux-là mêmes que nous sommes en train de définir.

Néanmoins il est tout à fait possible d'avoir un champ qui soit un pointeur sur `struct t`, donc un `struct t*`. Une telle définition n'est plus de taille infinie puisqu'un pointeur a une taille fixe, indépendante de vers quoi il pointe. Cette pratique est la base de la représentation très nombreuses structures de données algorithmiques comme les listes, les arbres, les graphes, etc.

14.2 Définition de variable de type structure

Fidèle à la syntaxe habituelle de C, la définition d'une variable d'un type structure prend la forme de son type suivi de son nom et d'une éventuelle initialisation :

```
struct nom-struct nom-var ;
struct nom-struct nom-var = { ..., ..., ... } ;
```

L'initialisation prend la forme d'une énumération, entre accolades, de valeurs constantes qui correspondent aux différents champs dans l'ordre qui est le leur dans la définition de type. Chaque valeur d'initialisation d'un champ doit être du type de ce champ. Dans le cas de structures imbriquées, l'initialisation (donc la « valeur ») d'une sous-structure est elle-même donnée entre accolades.

```
struct gps_loc_t msm_gps = { La_North, 48, 38, 8.9, Lo_West, 1, 30, 40.47 } ;

struct poi_t mt_st_michel = {
  "Mont Saint-Michel",
  "Commune de la Manche",
  { La_North, 48, 38, 8.9, Lo_West, 1, 30, 40.47 },
  true
```

```
} ;
```

Important

Les valeurs énumérées dans une initialisation de structure doivent être des constantes.

Dans l'exemple ci-dessus, ce n'est pas un hasard que nous ayons textuellement recopié les coordonnées GPS du Mont Saint-Michel, les mêmes que celles initialisant la variable `msm_gps`, à la place d'utiliser directement `msm_gps`. En effet, la variable `msm_gps` ne représente pas une constante (puisque c'est une variable).

Il est possible de définir des tableaux de `struct` de la même façon que pour n'importe quel autre type, de les initialiser en donnant entre accolades les valeurs de `struct` (elles-mêmes entre accolades).

```
struct gps_loc_t locations[20] ;

struct poi_t sites[2] = {
  { "...", "...", { ... }, ... },
  { "...", "...", { ... }, ... }
} ;
```

14.3 Accès dans une `struct`

Important

L'affectation globale d'une structure n'est pas possible. Dit autrement, il est syntaxiquement incorrect d'écrire :

```
a_struct = { 10, true, 3.14 } ;
another_struct = a_struct ;
```

Seule l'initialisation permet une forme « d'affectation » au moment de la définition d'une variable de type structure, avec une énumération entre accolades.

Pour accéder à un champ, en lecture comme en écriture, il faut utiliser la *notation pointée* qui permet de désigner le nom du champ auquel accéder dans la valeur de type structure :

$$expression.nom\text{–}champ$$

```
void print_lat (enum lat_orient_t lat) {
  switch (lat) {
  case La_North: printf ("nord") ; break ;
  case La_South: printf ("sud") ; break ;
  }
}

void print_lon (enum long_orient_t lon) {
  switch (lon) {
  case Lo_West: printf ("ouest") ; break ;
  case Lo_East: printf ("est") ; break ;
  }
}

void print_gps (struct gps_loc_t loc) {
  printf ("%u° %u' %f'' ", loc.la_deg, loc.la_min, loc.la_sec) ;
```

```
  print_lat (loc.lao) ;
  printf (" %u° %u' %f'' ", loc.lo_deg, loc.lo_min, loc.lo_sec) ;
  print_lon (loc.loo) ;
}
```

Puisqu'il n'y a pas d'affectation globale dans une structure, il est seulement possible d'affecter séparément chacun de ses champs.

```
...
struct gps_loc_t loc ;
...
loc.la_deg = 5 ;
...
loc.loc.loo = Lo_West ;
...
```

14.3.1 Accès via un pointeur

La notation pointée sert à accéder à un champ d'une expression de type `struct xyz`. Lorsque l'expression est un pointeur sur un type structure, donc un `struct xyz*`, il faut d'abord déréférencer le pointeur pour obtenir une valeur de type `struct xyz` avant d'utiliser la notation pointée pour accéder au champ.

```
#include <stdio.h>

struct dummy_t { int foo ; int bar ; } ;

int main () {
  struct dummy_t s ;
  struct dummy_t *ps = &s ;
  (*ps).foo = 27 ;
  (*ps).bar = 123 ;
  printf ("%d %d\n", s.foo, s.bar) ;
  printf ("%d %d\n", (*ps).foo, (*ps).bar) ;
  return 0 ;
}
```

```
$ gcc -Wall sptr.c -o sptr.x
$ ./sptr.x
27 123
27 123
```

Important

La présence des parenthèses est importante dans `(*ps).foo` car la priorité de la notation pointée est supérieure à celle du déréférencement. Sans parenthèses, `*ps.foo` aurait été interprété comme `*(ps.foo)`, donc comme aller récupérer la valeur se trouvant à l'adresse contenue dans `ps.foo`, ce qui est manifestement incorrect puisque le champ `foo` a le type `int` et non pointeur.

Cette écriture pour accéder aux champs de `struct` au travers de pointeurs est quelque peu encombrante. Comme c'est une opération courante en C, une syntaxe plus courte et totalement équivalente a été introduite : la « flèche » `->`.

Important

Ainsi, si `e` est une expression de type `struct t*`, alors `e->champ` ≡ `(*e).champ` (si bien sûr le type `struct t` possède un champ `champ`).

Le programme précédent peut donc être réécrit de manière totalement équivalente sous la forme suivante :

```
int main () {
  struct dummy_t s ;
  struct dummy_t *ps = &s ;
  ps->foo = 27 ;
  ps->bar = 123 ;
  printf ("%d %d\n", s.foo, s.bar) ;
  printf ("%d %d\n", ps->foo, ps->bar) ;
  return 0 ;
}
```

14.4 Interaction avec les fonctions

14.4.1 Des `struct` en arguments

Comme l'exemple précédent l'a déjà montré, une fonction peut prendre un argument de type structure, comme pour n'importe quel autre type. Cette possibilité est très intéressante pour limiter le nombre d'arguments que nous devrions passer à une fonction.

Plutôt que de passer séparément les 6 informations qui composent une coordonnée GPS, les grouper dans une structure et passer cette structure en argument d'une fonction permet de ne plus passer qu'un seul argument.

Comme expliqué en section 8.2.5, cette solution a toutefois une limite dans le cas de structures avec de nombreux champs, donc volumineuses. Afin de ménager la pile et le temps d'appel, au lieu de passer directement une valeur de type structure, il est plus efficace d'utiliser un passage par adresse, donc de passer un pointeur sur la structure. Au lieu d'utiliser la notation pointée (i.e. `.`) pour accéder aux champs, il faudra utiliser la notation fléchée (i.e. `->`).

```
struct verybig_t { int foo ; ... ... ... ... ... } ;

void f_heavy (struct verybig_t s) {
  ... s.foo ...
}


void f_light (struct verybig_t *s) {
  ... s->foo ...
}
```

14.4.2 Des `struct` en résultat

Les structures peuvent être un moyen facile de retourner plusieurs valeurs puisqu'il est possible de retourner une valeur d'un type `struct`.

Nous pouvons désormais réécrire avec des structures la fonction `vect_sum` que nous avions réécrite (correctement) dans la section 12.2.2 pour qu'elle utilise des tableaux. Cette fois-ci, nous représentons les vecteurs par des structures à la place de tableaux de 2 `float`.

```
struct vect2d_t {
  float x ;
  float y ;
                                                                    →
```

```
} ;

struct vect2d_t vect_sum (struct vect2d_t v1, struct vect2d_t v2) {
  struct vect2d_t sum ;
  sum.x = v1.x + v2.x ;
  sum.y = v1.y + v2.y ;
  return sum ;
}
```

Les structures peuvent aussi permettre de s'affranchir du passage d'argument par adresse présenté en section 8.2. La fonction `g` qui y était décrite, devant retourner un entier et un réel, avait nécessité le passage par adresse de son résultat réel (celui entier étant retourné directement avec `return`) :

```
int g (int x, float y, float *fres) ...
```

Pour s'épargner cette pirouette technique, il nous suffit de définir un nouveau type `struct` composé d'un entier et d'un réel et de faire retourner à `g` une valeur de ce type.

```
struct g_return_t {
  int res_int ;
  float res_float ;
} ;

struct g_return_t g (int x, float y) {
  struct g_return_t res ;
  ...
  res.res_float = y + 2.5 - ... ;
  res.res_int = 100 + x - ... ;
  return res ;
}
```

Même si cette solution est simple, elle possède un désavantage en terme de passage à l'échelle. Si nous avons de nombreuses fonctions devant retourner des valeurs de la sorte, avec pour chacune une combinaison de types différents, il va nous falloir déclarer autant de nouveaux types `struct` purement utilitaires. Nous pouvons ainsi nous retrouver à « polluer » notre programme avec une myriade de définitions de type qui n'ont pas de véritable rapport avec les types servant à modéliser réellement le problème algorithmique à résoudre.

```
struct ret_int_real_t { int rint ; float rfloat; } ;
struct ret_int_double_t { int rint ; double rdouble; } ;
struct ret_int_double_string_t { int rint ; double rdouble; char *rstr ; } ;
struct ret_int_int_real_t { int rint1 ; int rint2 ; float rfloat; } ;
...
```

N'attendez pas à trouver dans les lignes qui suivent une mesure, un critère absolu permettant de trouver un juste équilibre entre utilisation des structures et utilisation du passage par adresse. C'est une question de génie logiciel à trancher au cas par cas.

Dernier point pour alimenter la réflexion : (sauf exceptions, car il y en a toujours) une structure ne tient pas dans un registre au moment du `return`. Donc retourner un structure passe par la pile, avec les mêmes effets que ceux induits par le passage d'arguments qui sont des structures (cf. section 8.2.5) : espace dans la pile et temps de copie.

14.4.3 Structures pour des champs de même type

En début de ce chapitre, nous avons justifié l'intérêt des structures par leur capacité à grouper des éléments de types différents. Bien entendu, il est tout à fait possible de créer une structure dont tous les champs ont le même type. Pourquoi ne pas alors utiliser un tableau puisqu'un tableau permet de grouper des valeurs d'un même type ?

La première réponse est la facilité de retourner d'un coup l'ensemble des valeurs. C'est ce que nous avons fait pour le vecteur dans la section 14.4.2 précédente, nous évitant d'avoir recours à l'allocation dynamique.

La seconde est un argument de robustesse. Représenter un vecteur par une structure est intrinsèquement plus intéressant que le représenter par un tableau. En effet, un vecteur en 2 dimensions n'a que deux coordonnées. En le représentant par une `struct` avec deux champs `x` et `y`, nous ne pouvons structurellement accéder qu'à ces deux champs. À l'inverse, en représentant un vecteur par un tableau de 2 `float`, nous n'avons aucun mécanisme qui nous empêche d'accéder par inadvertance à la case d'indice 2, 3 ou plus de ce tableau. Rappelons-nous qu'un tableau ne comporte pas sa taille et que C ne fait aucune vérification quant aux indices lors de l'accès dans un tableau.

Important

Lorsqu'un langage de programmation fournit une construction qui apporte une garantie structurelle contre des erreurs, on s'en sert !

14.5 En résumé

- Une structure répond au besoin d'agréger des données de types différents.
- Une structure est un groupement de données par champs nommés.
- Déclaration d'un type structure :

```
struct nom-struct {
  type-champ nom-champ ;
  type-champ nom-champ ;
  ...
} ;
```

- Le nom du type est `struct nom-struct`, pas juste `nom-struct`.
- Le type de chaque champ peut-être quelconque, sauf du type `struct` en train d'être défini (pas de récursion).
- Définition d'une variable de type `struct` :

```
struct nom–struct nom–var ;
struct nom–struct nom–var = { ..., ..., ... } ;
```

- Quand une variable de type `struct` est initialisée, les valeurs sont données dans l'ordre des champs de la définition du type.
- L'affectation globale d'une structure n'existe pas.

```
ma_struct = { 1, false, 3.14 } ;
```

- Pour accéder au champ `champ` d'une expression `e` de type `struct` on utilise la notation pointée.

```
x = e.champ + 1 ;
e.champ = x + 1 ;
```

- Pour accéder au champ `champ` d'une expression `pe` de type `struct *` (donc pointeur) on utilise la notation fléchée.

```
x = pe->champ + 1 ;
*e->champ = x + 1 ;
```

- `pe->champ` $\equiv$ `(*pe).champ` (parenthèses importantes)
- Une fonction peut prendre en argument des structures et retourner une structure.

Chapitre 15

Le préprocesseur `cpp`

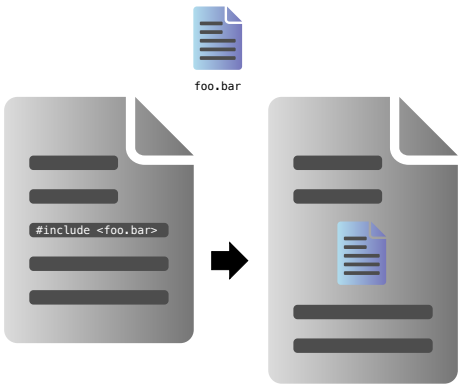
Le préprocesseur est un programme nommé `cpp`, séparé du compilateur `gcc`, qui est invoqué par ce dernier au tout début de la compilation. Ils sont donc conçus pour opérer main dans la main. Le préprocesseur prend en entrée un fichier texte source et retourne en sortie un autre fichier texte source (même si ce fichier intermédiaire est invisible lorsque l'on appelle `gcc`).

Le but de cette toute première passe du processus de compilation est de traiter des directives qui ne font pas partie du langage C, mais rendent la programmation plus aisée dans ce langage. Les deux directives les plus emblématiques sont `#include` et `#define`. Dit autrement, un code source qui contient un `#include` n'est pas encore du C que le compilateur peut transformer en exécutable : c'est le préprocesseur qui va se charger de faire disparaître cette directive pour obtenir en résultat un texte source respectant la syntaxe pure du langage C.

Les traitements réalisés par `cpp` relèvent principalement de la manipulation de texte, donc de chaînes de caractères, sans aucune analyse de la signification du programme. Le préprocesseur peut d'ailleurs être appelé sur n'importe quel type de fichier texte : aucune nécessité que le fichier contienne du C, même approximatif. Son effet est de remplacer les `#include` et les `#define` par du texte, de manière subtile, souple et pratique toutefois.

15.1 Fichiers d'en-tête et `#include`

Dans la majorité des exemples des chapitres précédents, à commencer par le "Hello World" de la section 1.3, la première ligne de code était un `#include` dont le rôle était de rendre « connues » un certain nombre de fonctions de la bibliothèque standard du langage C. Suivant cette directive se trouvait un nom de fichier souvent entre « piquants » (i.e. <>). L'effet du préprocesseur sur une directive `#include <foo.bar>` est de la remplacer par le contenu textuel du fichier `foo.bar`.



15.1.1 `#include < >`

Lorsque le nom de fichier est placé entre piquants, le préprocesseur recherche ce fichier parmi un ensemble de répertoires par défaut, en particulier ceux de la bibliothèque standard de C. L'option `-v` de `cpp` permet d'obtenir beaucoup d'informations à propos de sa version, entre autres la liste des répertoires explorés par défaut.

```
$ cpp -v /dev/null -o /dev/null
Using built-in specs.
COLLECT_GCC=cpp
...
#include <...> search starts here:
 /usr/lib/gcc/x86_64-linux-gnu/12/include
 /usr/local/include
 /usr/include/x86_64-linux-gnu
 /usr/include
End of search list.
...
```

Il est possible de rajouter un répertoire à cette liste par défaut en appelant `gcc` avec l'option `-I` suivie du nom de ce répertoire. `gcc` se chargera de transférer l'information au préprocesseur `cpp`. L'extrait d'exécution ci-dessous rajoute manuellement le répertoire `/usr/games` qui se retrouve bien listé dans les répertoires par défaut.

```
$ cpp -I /usr/games -v /dev/null -o /dev/null
Using built-in specs.
COLLECT_GCC=cpp
...
#include <...> search starts here:
 /usr/games
 /usr/lib/gcc/x86_64-linux-gnu/12/include
 /usr/local/include
 /usr/include/x86_64-linux-gnu
 /usr/include
End of search list.
...
```

15.1.2 `#include " "`

Lorsque le nom de fichier est placé entre guillemets, le préprocesseur recherche ce fichier directement depuis le répertoire où se trouve le fichier source. Pour être exact, ce qui est mis entre piquants ou guillemets n'est pas forcément juste un nom de fichier. C'est un chemin de fichier qui peut donc mentionner des répertoires. Par exemple, avec

`#include "myincludes/foo.h"`.

le préprocesseur ira rechercher le fichier `foo.h` se trouvant dans un répertoire `myincludes` lui-même situé au même endroit que le fichier source contenant cette directive.

15.1.3 Fichiers `.h`

Dans un fichier source, lorsqu'une fonction est appelée mais n'a pas déjà été définie dans ce fichier, le compilateur a besoin de savoir ce qu'elle attend comme arguments et surtout ce qu'elle retourne comme type. C'est son prototype (cf. section 3.1). Il en est de même dans le cas d'une référence à une variable (forcément globale) définie dans un autre fichier : il faut connaître son type.

Le contenu d'un fichier `.h` est constitué de définitions de types, de déclarations de fonctions et de variables. Ces informations sont écrites une fois pour toutes, centralisées au même endroit. Tout fichier source nécessitant ces informations de type n'a qu'à inclure le `.h`.

Le pré-traitement de `cpp` effectué en rencontrant `#include <foo.h>` ou `#include "foo.h"` va donc incorporer au fichier source qui est en train d'être compilé, le contenu du fichier `foo.h`, comme si son contenu avait été directement écrit dans le fichier compilé.

L'intérêt de ce mécanisme et de la centralisation de ces définitions est manifeste : si nous avions dû manuellement réécrire toutes les définitions dans de multiples fichiers sources, outre la perte de temps occasionnée, il y aurait eu le risque d'introduire des incohérences entre les différentes occurrences d'une même définition. Une petite faute de frappe, d'inattention est si vite arrivée... De plus, en cas de modification sur une définition de type ou sur un prototype de fonction (parce que nous aurions changé cette fonction), il aurait fallu propager cette modification dans tous les fichiers sources concernés. Une fois de plus, un oubli est si vite arrivé....

Pour illustrer le mécanisme d'inclusion de fichier, nous allons utiliser des fichiers textes qui ne contiennent même pas du code C correct. Considérons le fichier source `bidouille.c` suivant :

```
#include "header1.h"

Mon code source
int pas du C ce truc () {

#include "header2.h"
Toujours pas du C ;
}
```

Ce fichier procède à l'inclusion de deux fichiers, `header1.h` et `header2.h` (qui ne contienne pas non plus du code C forcément décent).

```
int mult (int x, int y) ;
```

```
Du blabla ici.
```

L'appel au préprocesseur sur `bidouille.c` est réalisée en invoquant `cpp bidouille.c` dans le terminal. Rappelons que c'est ce que le compilateur `gcc` fait de manière invisible au tout début du processus de compilation. Le résultat du pré-traitement est directement affiché dans le terminal.

```
$ cpp bidouille.c
# 0 "bidouille.c"
# 0 "<built-in>"
# 0 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 0 "<command-line>" 2
# 1 "bidouille.c"
# 1 "header1.h" 1
int mult (int x, int y) ;
# 2 "bidouille.c" 2

Mon code source
int pas du C ce truc () {

# 1 "header2.h" 1
Du blabla ici.
# 7 "bidouille.c" 2
Toujours pas du C ;
}
```

Nous pouvons constater que le résultat est bien à nouveau un texte, dans lequel nous remarquons que les 2 directives `#include` ont été remplacées par l'inclusion textuelle du contenu des 2 fichiers `.h`.

Pour aller plus loin

Le code source réellement envoyé dans la suite du processus de compilation est celui issu du pré-traitement. Il a donc la forme de ce qui a été montré dans l'exemple ci-dessus. Les lignes supplémentaires commençant par # sont des repères permettant de mettre en correspondance les numéros de ligne de code des fichiers sources initiaux (ceux réellement écrits) et ceux du fichier issu du pré-traitement. Cela est indispensable pour indiquer des numéros de ligne cohérents dans les messages d'erreur et d'avertissement. En effet, l'utilisateur ne voit que les fichiers qu'il a écrits, pas celui issu de `cpp`. Donc les numéros de ligne qui doivent lui être communiqués doivent être relatifs à ses propres fichiers sources, `bidouille.c`, `header1.h` ou `header2.h` (car oui, il peut y avoir des erreurs dans un `.h` inclus).

L'utilisation de fichiers d'en-tête revêt une utilité incontournable dès lors que nous utilisons le mécanisme de *compilation séparée*. Un programme réaliste est composé de plusieurs fichiers sources qui définissent des fonctions utilisées dans les autres fichiers. Afin de ne pas recompiler tous les fichiers sources à chaque fois (ce qui peut être réellement long), il est possible de compiler chaque fichier séparément (s'il a changé) puis d'effectuer une phase finale permettant de regrouper ces résultats de compilation partielle pour en faire l'exécutable final. Généralement, chaque fichier source `.c` est accompagné de son fichier d'en-tête `.h` qui déclare les fonctions et autres éléments « publics » présents dans le `.c`.

15.2 Macros, constantes symboliques et `#define`

Dans le chapitre 9 sur les tableaux statiques, nous avons insisté sur le fait que leur taille devait être une constante. Ainsi, toutes les boucles que nous pouvons écrire sur un tel tableau sont bornées par cette constante.

Considérons le schéma de programme suivant où un tableau de taille 27 est défini, avec une boucle dessus et divers autres traitements. La valeur 27 se retrouve à différents endroits. Néanmoins, nous pouvons légitimement imaginer que celle utilisée dans `int jour = 27` n'a rien à voir avec la taille du tableau.

```
...
int t[27] ;
for (int i = 0; i < 27; i++) ... ;
int len_times_10 = 27 * 10 ;
for (int j = 0; j < len_times_10; j++) ... ;
int taille = ... ;
...
if (indice < 27) ... t[indice] ... ;
int jour = 27 ;
...
```

Dans ce programme, si nous souhaitons modifier la taille du tableau en la remplaçant par 34, nous devons manuellement changer toutes les occurrences pertinentes de « 27 » par « 34 » pour obtenir :

```
...
int t[34] ;
for (int i = 0; i < 34; i++) ... ;
int len_times_10 = 34 * 10 ;
for (int j = 0; j < len_times_10; j++) ... ;
int taille = ... ;
...
if (indice < 34) ... t[indice] ... ;
```

```
int jour = 27 ;
...
```

dans quoi la dernière occurrence n'a pas été modifiée puisqu'elle ne concernait pas la taille du tableau. Ces modifications manuelles sont donc longues et périlleuses : il faut s'assurer que nous ne changeons que les « bonnes » occurrences de « 27 » et que nous n'en oublions pas. De surcroît, si cette taille est utilisée dans du code de plusieurs fichiers sources, il faudra effectuer ces modifications dans autant de fichiers.

15.2.1 Macros constantes

Il serait préférable de disposer d'un moyen de définir une constante symbolique, nommée, par exemple `SIZE`, une fois pour toutes puis d'utiliser ce nouveau symbole partout où pertinent à la place de « 27 ». C'est exactement ce que permet la directive

```
#define SIZE 27
```

qui définit un symbole nommé `SIZE` qui sera textuellement remplacé partout par `27`. Un tel symbole est nommé *macro* (pour abréger *macro-définition*).

Notre programme ci-dessus peut donc être réécrit sous la forme ci-dessous, en tirant parti de ce mécanisme. Si nous souhaitons maintenant une taille différente, il suffit de la redéfinir à un seul endroit, dans le `#define`.

```
#define SIZE 42 + 1
...
int t[SIZE] ;
for (int i = 0; i < SIZE; i++) ... ;
int len_times_10 = SIZE * 10 ;
for (int j = 0; j < len_times_10; j++) ... ;
int taille = ... ;
...
if (indice < SIZE) ... t[indice] ... ;
int jour = 27 ;
...
```

Durant le pré-traitement, `cpp` remplace chaque occurrence du symbole `SIZE` par le texte qui suit sa définition dans la directive `#define`. C'est donc un « rechercher-remplacer » (sensible à la casse) similaire à celui disponible dans tout bon éditeur de textes. Une fois le préprocesseur appelé avec ce fichier, nous obtenons le code source suivant dans lequel toutes les occurrences de `SIZE` ont été remplacées par `42 + 1`.

```
$ cpp foo.c
# 0 "foo.c"
# 0 "<built-in>"
# 0 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 0 "<command-line>" 2
# 1 "foo.c"

...
int t[42 + 1] ;
for (int i = 0; i < 42 + 1; i++) ... ;
int len_times_10 = 42 + 1 * 10 ;
for (int j = 0; j < len_times_10; j++) ... ;
int taille = ... ;
```

```
...
if (indice < 42 + 1) ... t[indice] ... ;
int jour = 27 ;
...
```

Le code source obtenu ci-dessus nécessite un examen attentif. Dans notre code source initial, nous avions écrit

```
int len_times_10 = SIZE * 10 ;
```

vraisemblablement pour spécifier que la variable `len_times_10` représente la taille du tableau multipliée par 10. Le remplacement textuel aveugle a substitué `TAILLE` par `42 + 1`, ce qui transforme cette instruction en

```
int len_times_10 = 42 + 1 * 10 ;
```

La priorité de la multiplication étant supérieure à celle de l'addition, l'expression est devenue $42+(1\times10)$ et certainement pas $(42+1)\times10$. Il faut garder à l'esprit que le préprocesseur remplace sans prise en compte du contexte où sont les symboles qu'il remplace : il n'a aucune compréhension de la sémantique du fichier qu'il traite.

Important

Pour éviter ce genre d'erreur dépendante du contexte de substitution, il faut prendre l'habitude de mettre des parenthèses autour du texte des macros. Une définition correcte de la macro précédente est donc :

```
#define TAILLE (42 + 1)
```

Avec ces parenthèses, qui font bien partie du texte suivant le nom du symbole de la macro, le résultat du pré-traitement correspond à ce que nous attentions :

```
...
int t[(42 + 1)] ;
for (int i = 0; i < (42 + 1); i++) ... ;
int len_times_10 = (42 + 1) * 10 ;
for (int j = 0; j < len_times_10; j++) ... ;
int taille = ... ;
...
if (indice < (42 + 1)) ... t[indice] ... ;
int jour = 27 ;
...
```

15.2.2 Macros paramétrées

Il est possible de définir des macros plus souples, comportant des paramètres qui apparaissent dans le texte de remplacement. Dit autrement, le texte de remplacement peut varier selon les arguments avec lesquels nous utilisons le symbole de macro. Imaginons que nous ayons besoin, à différents endroits d'un programme, de calculer le volume d'un parallélépipède. Une première solution serait de définir une fonction

```
float vol (float larg, float haut, float prof) { return larg * haut * prof ; }
```

ce qui peut paraître excessif pour effectuer deux multiplications. Plutôt que de répéter partout le code de ces 2 multiplications, nous pouvons définir la macro paramétrée

```
#define VOL(larg, haut, prof) ((larg) * (haut) * (prof))
```

Important

Il ne faut pas d'espace entre le nom de la macro et la parenthèse ouvrante. Dans le cas contraire, ce sera une définition de macro constante (non paramétrée) qui sera simplement remplacée par le reste de la ligne.

L'utilisation d'une telle macro se fait exactement comme pour les fonctions, avec des arguments placés entre parenthèse et séparés par des virgules. Par exemple, de lors de l'utilisation de la macro VOL

```
nrt = p * VOL (3, x, z * cos (a)) ;
```

celle-ci sera remplacée par son texte dans lequel `larg`, `haut` et `prof` seront remplacés par le texte fourni pour chacun lors de l'utilisation de la macro `VOL`.

```
#define VOL(larg, haut, prof) ((larg) * (haut) * (prof))

... ma fonction () {
  ...
  rt = p * VOL (3, x, z * cos (a))
  ...
  if (VOL (a, 5, 9.5 * "... à celui qui le lira")) {
  ...
```

```
$ cpp pmacro.c
# 0 "pmacro.c"
# 0 "<built-in>"
# 0 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 0 "<command-line>" 2
# 1 "pmacro.c"

... ma fonction () {
  ...
  rt = p * ((3) * (x) * (z * cos (a)))
  ...
  if (((a) * (5) * (9.5 * "... à celui qui le lira"))) {
  ...
```

Comme précédemment expliqué, le préprocesseur remplace sans se poser de questions d'ordre sémantique. Dans la seconde invocation de la macro `VOL`, le dernier argument est une chaîne, ce qui n'a aucun sens (car c'est mal typé), mais le remplacement est quand même réalisé. Ce sera le compilateur qui, en analysant le code issu du préprocesseur, tombera sur cette erreur et la signalera.

Important

Remarquons que nous avons pris soin de mettre des parenthèses autour de chaque occurrences des paramètres à l'intérieur de la macro, afin d'éviter le problème présenté pour les macros constantes et les erreurs qui en découlent.

15.2.3 Fonctions ou macros paramétrées ?

Le débat de savoir s'il vaut mieux utiliser des fonctions ou des macros paramétrées est compliqué car de nombreux éléments techniques interviennent, avec des avantages et des inconvénients pour chaque solution.

En approximation grossière, nous pouvons considérer que lorsque le traitement est court (ici nous avions 2 multiplications), il est raisonnable de définir une macro paramétrée. Dès lors que la complexité du traitement augmente, une fonction s'imposera assez naturellement.

Pour aller plus loin

Nous pouvons donner ici quelques éléments de jugement plus précis pour différencier le comportement de chaque solution. Bien entendu, les avantages de l'une sont les désavantages de l'autre. Sauf contexte particulier, même avec ces éléments de jugement, il est difficile de tirer des conclusions définitives pour le cas général. Une part de subjectivité reste présente dans le choix individuel de chaque personne qui programme.

- **Fonction** :
  - \+ La fonction existe réellement dans le binaire. Ainsi, pour débugger un programme avec un outil de trace (nommé "*debugger*"), il sera possible de poser précisément des points d'arrêt sur le point d'entrée de la fonction et sur ses instructions.
  - \+ Le code de la fonction existe une seule fois dans l'exécutable généré.
  - \+ Dans le monde du logiciel critique, des évaluations de traçabilité entre le code source et le binaire sont effectuées. Il est généralement nettement plus simple de vérifier qu'une fonction est bien présente dans l'exécutable généré que de devoir retrouver, au milieu d'autres instructions, celles qui constituaient le corps d'une macro (de surcroît, pour chaque occurrence d'utilisation de la macro).
  - \+ Les arguments d'une fonction sont évalués une seule fois (cf. chapitre 8) donc il n'y a pas de risque de dupliquer des effets de bords (modifications mémoire, affichages, etc.) qui pourraient être présents dans les arguments (par exemple `f (i++)`).
  - \- Un appel de fonction a un coût : il faut respecter le protocole d'appel qui effectue des copies, sauvegarde l'adresse de retour, etc. Néanmoins, les compilateurs modernes sont très sophistiqués et peuvent remplacer des appels de fonctions par leur corps où les paramètres ont été habilement substitués. Donc, ce point négatif est plus subtil à évaluer.
  - \- Un appel de fonction complique fortement les analyses permettant certaines optimisations. En effet, le flot d'exécution du programme n'est plus linéaire et un certain nombre d'informations nécessaires aux optimisations, calculées par le compilateur, passent difficilement la barrière d'un appel de fonction.
- **Macro paramétrée** :
  - \+ Une macro permet d'éviter le coût d'un appel de fonction.
  - \+ Il peut être plus facile d'optimiser localement le code qui n'est pas interrompu par un appel de fonction.
  - \- Dans un contexte de recherche de bug avec un "*debugger*", il ne sera pas possible de mettre un point d'arrêt sur le début de la macro, ni même sur ses instructions.
  - \- Le code de la macro étant expansé à chaque utilisation, il en résulte une duplication de ce dernier, contribuant à l'augmentation de la taille de l'exécutable.
  - \- La traçabilité dans le cadre d'analyses de sûreté est généralement nettement plus compliquée.
  - \- Le texte de l'argument fourni pour le paramètre à l'utilisation d'une macro étant dupliqué autant de fois que ce paramètre apparaît dans la macro, un effet de bord pourra se retrouver dupliqué.

15.3 En résumé

- Le préprocesseur s'appelle `cpp`.
- Il est appelé automatiquement par le compilateur `gcc` au début du processus de compilation.
- Le préprocesseur prend le texte source et rend un nouveau texte source.
- Il effectue de la manipulation textuelle sans aucune compréhension du texte qu'il traite.
- Il remplace toutes les directives `#include` par le contenu textuel du fichier spécifié dans la directive.
- Les fichiers d'en-tête servent à regrouper à un endroit unique des déclarations de fonctions, de variables (globales), des définitions de types, des constantes symboliques (`#define`).
- Le préprocesseur remplace les définitions de macros, introduites par `#define` : chaque occurrence du symbole défini par la macro est remplacée par le texte qui suit dans la macro.
  ```
  #define TAILLE (42 + 1)
  ```
- Pour éviter des erreurs dues à l'expansion aveugle d'une macro, il faut entourer son corps par des parenthèses.
- Les macros peuvent être paramétrées.
  ```
  #define VOL(larg, haut, prof) ((larg) * (haut) * (prof))
  ```
- Il n'y a pas d'espace avant la parenthèse ouvrante qui introduit les paramètres.
- Pour éviter des erreurs dues à l'expansion aveugle d'une macro paramétrée, il faut en plus entourer les occurrences de ses paramètres dans son corps par des parenthèses.

Chapitre 16

Considérations plus avancées

Ce chapitre aborde un certain nombre de traits, de constructions, de concepts du langage C allant au-delà de ce qui est attendu comme connaissances de base. L'intention est de ne pas totalement omettre certains aspects du langage sous couvert de simplicité et que ce document puisse servir de référence raisonnable y compris aux lecteurs désireux d'approfondir le sujet. Les sujets abordés dans les sections suivantes sont relatifs à diverses catégories de constructions du langage et pourraient être rattachés chacun à un des chapitres précédents.

16.1 Alias de type

Il est possible de donner un nom à une expression de type grâce à la construction `typedef` :

`typedef` $type\text{–}expr$ $type\text{–}nom$ `;`

qui permet de définir un nouveau nom de type $type\text{–}nom$ qui sera synonyme de l'expression de type $type\text{–}expr$. Cela signifie que $type\text{–}nom$ et $type\text{–}expr$ seront totalement interchangeables. Il s'agit bien d'une définition d'alias : cela ne crée pas une nouvelle structure de données comme `enum` ou `struct` le permettent. Une fois qu'un nom de type est introduit, il peut entrer dans la composition d'autres définitions de type.

```
typedef short int myty_t ;

struct pair_t {
  myty_t left ;
  myty_t right ;
} ;

myty_t identity (myty_t x) { return x ; }

void foobar () {
  myty_t i = 123 ;
  myty_t t[5] = { 4, 5, 3, 2, 1 } ;
  myty_t *p = &t[2] ;
  short int zero = 0 ;
  short int r = identity (zero) + identity (i) ;
  ...
}
```

Dans le programme ci-dessus, `myty_t` est défini comme alias de `short int`. Il est ensuite utilisé pour créer une structure, une fonction qui prend et retourne une valeur de ce type, une variable `i`, un tableau et un pointeur. L'appel à la fonction `identity` montre que `short int` est utilisable là où `myty_t` est attendu, puisque `zero` (qui est un `short int`) est fourni sans problème à la fonction qui attend un `myty_t`. De même, le résultat de `identity` est stocké dans une variable `r` de type `short int`.

16.1.1 Abréger les types

La construction `typedef` permet de raccourcir de longues expressions de type. Par exemple, plutôt qu'écrire `unsigned long int*` nous pouvons définir

```
typedef unsigned long int* ulptr_t ;
```

et utiliser `ulptr_t` qui est plus court en lieu et place de l'expression qu'il représente. C'est en fait l'artifice utilisé dans le fichier d'en-tête `<stdbool.h>` pour masquer le qualificatif `enum` du type `bool` (cf. section 3.3.1).

```
enum bool { false, true } ;
typedef enum bool bool ;
```

Il est possible de combiner directement la définition d'un `enum`, d'une `struct` ou d'une `union` (cf. section 16.2) et le `typedef` qui la nomme. La définition précédente du type `bool` peut alors s'écrire de manière plus concise comme :

```
typedef enum bool { false, true } bool ;
```

16.1.2 Documenter les types

L'utilisation de `typedef` peut permettre de rendre plus explicite la sémantique d'une expression de type. Ce point est particulièrement important pour écrire des programmes maintenables puisqu'il contribue, d'une certaine manière, à documenter naturellement le code source.

Par exemple, le type `float` peut représenter des valeurs de natures totalement différentes. Considérons la fonction suivante qui calcule l'énergie cinétique et prend naturellement en argument une masse et une vitesse.

```
float kinetic_energy (float s, float m) {
  return 0.5 * m * s * s ;
}
```

À part les noms des arguments, pas totalement limpides mais pas totalement obscurs non plus (oui, `m` peut faire penser à "mass" et `s` à "speed"), rien ne les différencie en terme de type. Pourtant, ces deux grandeurs ne sont pas interchangeables. De surcroît, elles ne sont pas utilisables dans n'importe quel contexte : additionner un `float` qui représente une masse avec un `float` qui représente une vitesse n'a aucun sens physique. Pour terminer, la valeur de retour de cette fonction est une énergie, aussi représentée par un `float`.

Définir des alias du `type float` avec des noms représentant la sémantique des valeurs physiques qu'ils représentent permet de rendre le code source infiniment plus lisible. En particulier, rien qu'en lisant le prototype de la fonction il est possible de voir clairement à quoi correspondent ses arguments.

```
typedef float mass_t ;
typedef float speed_t ;
typedef float energy_t ;

energy_t kinetic_energy (speed_t s, mass_t m) {
  return 0.5 * m * s * s ;
}
```

Soyons clairs : tous ces types sont des alias de `float`, donc totalement compatibles entre eux. Les mélanger de manière sémantiquement (i.e. physiquement dans cet exemple) incorrecte ne provoquera aucune erreur ni avertissement du compilateur. Cependant, faciliter la lecture du code est absolument indispensable.

Important

Bien entendu, cela n'enlève en rien le caractère également indispensable de la présence de commentaires dans le code source.

16.2 Type `union`

Dans le cas où une fonction peut retourner des valeurs de différentes formes, donc de différents types, nous n'avons actuellement pas d'autre solution que de créer une structure avec autant de champs qu'il y a de types possibles à retourner plus un champ spécial permettant de savoir quel champ est effectivement pertinent dans la valeur retournée.

Prenons l'exemple d'un modèle de données permettant de représenter des formes géométriques : cercle ou rectangle. Chaque forme peut individuellement être représentée par une structure :

```
struct circle_t {
  float x_center ; float y_center ;
  float radius ;
} ;
struct rect_t {
  float x_top_left ; float y_top_left ;
  float x_bot_right ; float y_bot_right ;
} ;
```

Si nous souhaitons manipuler des collections de formes de manière générique, il nous faut par exemple pouvoir les stocker dans un tableau et écrire des fonctions dont le comportement s'adapte à l'individualité de la forme que nous lui donnons (par exemple, une fonction d'affichage). Puisque les éléments d'un tableau doivent être de même type il est hors de question d'imaginer faire un tableau mélangeant `struct circle_t` et `struct rect_t`. De manière similaire, une fonction d'affichage qui prendrait en argument un `struct circle_t` ne pourrait pas accepter un `struct rect_t` et serait de fait dédiée à une seule catégorie de forme.

Nous devons donc créer un type unique regroupant ces deux types. Une structure semble adaptée. Néanmoins, il nous faut aussi une information permettant de savoir à chaque instant quelle forme représente effectivement une valeur de la structure (c'est un rectangle ou un cercle, mais pas les deux à la fois). Il suffit de rajouter à notre future structure un champ d'un type énuméré défini pour l'occasion indiquant « cercle » ou « rectangle ». Nous complétons donc notre modèle de données de la manière suivante.

```
enum shape_kind_t { Circle , Rectangle } ;
struct shape_t {
  enum shape_kind_t kind ;
  struct circle_t as_circle ;
  struct rect_t as_rectangle ;
} ;
```

Nous pouvons, avec ce modèle de données, désormais écrire notre fonction d'affichage. Elle va simplement regarder dans son argument quelle est la valeur du champ `kind` et, en fonction de si c'est

`Rectangle` ou `Circle`, elle affichera les valeurs contenues dans les champs pertinents : `as_rectangle` ou `as_circle`.

```
void print_shape (struct shape_t s) {
  switch (s.kind) {
  case Rectangle:
    printf ("Rect (%f, %f) - (%f, %f)\n",
            s.as_rectangle.x_top_left, s.as_rectangle.y_top_left,
            s.as_rectangle.x_bot_right, s.as_rectangle.y_bot_right) ;
    break ;
  case Circle :
    printf ("Circ (%f, %f) - %f\n",
            s.as_circle.x_center, s.as_circle.y_center, s.as_circle.radius) ;
    break ;
  }
}
```

Même si cette solution fonctionne parfaitement, nous pouvons constater que la taille d'une `struct shape_t` est la somme des tailles de ses 3 champs, en particulier celles d'un rectangle et d'un cercle. Pourtant, à chaque instant une valeur de type `struct shape_t` représente une seule de ces deux formes. Il y a donc un gaspillage de mémoire.

Le type `union` permet de définir une structure de données comme étant la réunion de plusieurs autres, occupant un même emplacement mémoire, dont la taille est le plus grande de toutes. Dit autrement, le compilateur va réserver la place pour stocker la plus grande donnée de l'union.

```
union nom-union {
  type-champ nom-champ ;
  type-champ nom-champ ;
  ...
} ;
```

Chaque champ de l'union permet d'accéder à la valeur (unique, forcément) qu'elle contient en l'interprétant selon le type de ce champ. L'union permet donc autant de vues d'une même zone mémoire qu'elle comporte de champs. Nous définissons donc une `union core_shape_t` avec deux champs pour fusionner cercles et rectangles afin de représenter le « cœur » de la forme.

```
union core_shape_t {
  struct circle_t as_circle ;
  struct rect_t as_rectangle ;
} ;
```

Il ne nous reste plus qu'à modifier notre `struct shape_t` pour lui faire contenir le champ `kind` (comme précédemment) et uniquement un champ de type `union core_shape_t` à la place des deux anciennes structures.

```
struct shape_t {
  enum shape_kind_t kind ;
  union core_shape_t core ;
} ;
```

Le fonctionnement de la fonction d'affichage `print_shape` ne va pas fondamentalement changer : la seule modification est qu'il faudra accéder au champ `core` qui contient l'union, puis au champ pertinent dans cette union.

```
void print_shape (struct shape_t s) {
  switch (s.kind) {
  case Rectangle:
    printf ("Rect (%f, %f)-(%f, %f)\n",
            s.core.as_rectangle.x_top_left, s.core.as_rectangle.y_top_left,
            s.core.as_rectangle.x_bot_right, s.core.as_rectangle.y_bot_right)
                ;
    break ;
  case Circle :
    printf ("Circ (%f, %f)-%f\n",
            s.core.as_circle.x_center, s.core.as_circle.y_center,
            s.core.as_circle.radius) ;
    break ;
  }
}
```

L'indirection supplémentaire visible dans la syntaxe (i.e `.as_circle` ou `.as_rectangle`) ne correspond pas à une structure de donnée avec plus d'imbrication en mémoire : elle correspond juste à la sélection de comment interpréter la zone mémoire.

16.2.1 Initialisation

Pour initialiser une variable de type `union` au moment de sa définition, il faut spécifier selon quelle vue la valeur doit être stockée. Cela permet en particulier au compilateur de vérifier que la constante servant à l'initialisation est bien en accord avec cette vue. Une constante de type `union` est donnée entre accolades en spécifiant le champ à initialiser précédé par un point.

`union ty_t v = { .`*nom–champ* `= ... } ;`

Considérant l'exemple des formes géométriques, nous pouvons ainsi définir et initialiser une variable représentant un cercle par :

```
struct shape_t a_circle = {
  Circle,
  { .as_circle = { 5.0, 6.5, 12.0 } }
} ;
```

16.2.2 Utilisation incohérente

À part dans de rares cas qui nécessitent des précautions techniques hors sujet, écrire puis lire dans une union en utilisant deux vues différentes mène à un comportement incohérent. En effet, après l'écriture, lors de la lecture, l'interprétation des bits présents en mémoire sera différente de celle avec laquelle ils avaient été écrits.

```c
#include <stdio.h>

union int_or_float_t {
  int as_int ;
  float as_float ;
} ;

int main () {
  union int_or_float_t u =
    { .as_float = 3.14 } ;
  printf ("%d\n", u.as_int) ;
  return 0 ;
}
```

```
$ gcc -Wall badunion.c -o badunion.x
$ ./badunion.x
1078523331
```

16.3 Qualificateur `const`

Lorsqu'une variable est destinée à ne plus changer de valeur après son initialisation son type peut être annoté par le mot-clé `const` lors de la déclaration.

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

int main (int argc, char *argv[]) {
  if (argc != 2) return 1 ;
  const int nochange = atoi (argv[1]) * 10 ;
  nochange++ ;
  printf ("%d\n", nochange) ;
  return 0 ;
}
```

```
$ gcc -Wall const.c
const.c: In function 'main':
const.c:7:11: error: increment of read-only variable 'nochange'
   nochange++ ;
           ^~
```

Toute tentative d'affectation après la déclaration de la variable est rejetée lors de la compilation. Déclarer une variable `const` est différent de déclarer une constante avec `#define` : la valeur de la variable n'est pas une constante symbolique, elle peut être le résultat d'un calcul à l'exécution. C'est le cas dans l'exemple ci-dessus où sa valeur d'initialisation est le résultat d'un calcul utilisant un des arguments de la ligne de commande. Par contre, une fois définie, sa valeur ne pourra plus changer.

Il est également possible de déclarer un paramètre de fonction `const`. Nous savons que par défaut une fonction peut utiliser ses paramètres comme des variables locales. Les modifications effectuées dessus n'auront pas d'effet du côté de la fonction appelante (sauf cas du passage d'argument par adresse) mais seront visibles dans la fonction elle-même. Déclarer un paramètre `const` aura le même effet que sur une variable locale.

```c
int f (const int x) {
  x++ ;
  return x ;
}
```

```
$ gcc const.c
const.c: In function 'f':
const.c:2:4: error: increment of read-only parameter 'x'
   x++ ;
    ^~
```

16.3.1 Pointeur `const`ant

Un pointeur étant une variable, il peut être défini `const` pour signifier qu'après son initialisation avec une adresse, il ne sera plus modifié :

```
int* const p = &... ;
```

définit et initialise un pointeur constant sur `int`.

```
void f () {
  int x = 5 ;
  int y = 20 ;
  int* const p = &x ;
  *p = 100 ;  /* Ok, c'est p qui est constant, pas *p. */
  p = &y ;    /* Non. */
}
```

```
$ gcc const.c
const.c: In function 'f':
const.c:6:5: error: assignment of read-only variable 'p'
   p = &y ;
     ^
```

Modifier la valeur située à l'adresse contenue dans `p`, donc affecter `*p` ne pose pas de problème : c'est `p` qui est constant, pas les données vers lesquelles il pointe. Par contre, tenter de donner une nouvelle valeur à `p` avec l'adresse de `y` est rejeté.

16.3.2 Pointeur sur `const`ant

Un pointeur peut désigner un endroit où se trouvent des valeurs considérées comme constantes. Dans ce cas, ce n'est pas le pointeur qui est constant, mais la (ou les) zone(s) mémoire qu'il désigne :

```
const int* p = &... ;
```

définit et initialise un pointeur vers des `int` considérés comme constants.

```
void f () {
  int x = 5 ;
  int y = 20 ;
  const int *p = &x ;
  p = &y ;    /* Ok, ce n'est pas p qui est constant. */
  *p = 100 ;  /* Non. */
}
```

```
$ gcc const.c
const.c: In function 'f':
const.c:6:6: error: assignment of read-only location '*p'
   *p = 100 ;  /* Non. */
    ^
```

Important

Il est important de comprendre que l'annotation `const` ici revient à une contrainte que nous nous imposons à propos des zones mémoires que `p` désignera. En tant que telle, aucune de ces zones mémoire ne seront placées par le compilateur dans une partie physiquement non modifiables. Une même adresse mémorisée dans un pointeur sur `const` et dans un pointeur « normal » sera non modifiable ou modifiable selon le pointeur par lequel on y accède.
Ainsi, l'annotation `const` a pour but de nous protéger de nous-mêmes ou des utilisateurs de nos fonctions, lorsque nous programmons, pour demander au compilateur de vérifier (autant que possible) que ce que nous avons imaginé (et qui est souvent nécessaire à la correction du programme) à propos de la mutabilité de telle ou telle zone mémoire est bien respecté.

```
void f () {
  int x = 5 ;
  int *p = &x ;
  const int *q = &x ;
  *p = 100 ;
  *q = 100 ;  /* Non. */
}
```

```
$ gcc const.c
const.c: In function 'f':
const.c:6:6: error: assignment of read-only location '*q'
   *q = 100 ;  /* Non. */
      ^
```

Les deux pointeurs `p` et `q` désignent le même endroit mémoire : la variable `x`. Par contre, `p` n'est pas qualifié comme désignant une zone constante alors que `q` l'est. Ainsi, accéder à `x` en écriture via l'adresse contenue dans `p` ne pose pas de problème alors que le faire via `q` est rejeté.

16.3.3 Pointeur `const`ant sur `const`ant

Il est possible de combiner à la fois la constance d'un pointeur et celle des zones mémoire qu'il désigne :

```
const int* const ptr = &... ;
```

définit un pointeur constant vers une zone contenante des valeurs constantes.

```
void f () {
  int x = 5 ;
  int y = 20 ;
  const int* const p = &x ;
  *p = 100 ;    /* Non. */
  p = &y ;      /* Non. */
}
```

```
$ gcc const.c
const.c: In function 'f':
const.c:5:6: error: assignment of read-only location '*p'
   *p = 100 ;
      ^
const.c:6:5: error: assignment of read-only variable 'p'
   p = &y ;
```

→

```
    ^                                                                    ←
```

Il devient à la fois impossible de modifier la valeur de `p` ainsi que la ou les valeurs se trouvant à l'adresse qu'il contient.

16.3.4 Arguments pointeurs et `const`

Le fait de spécifier qu'un paramètre de fonction de type pointeur désigne une zone contenant des données constantes permet en premier lieu d'interdire à la fonction de modifier la zone dont elle a reçu l'adresse en argument.

```
void f (const char *s)
    {
  s[0] = 'F' ;
}
```

```
$ gcc -Wall const.c -o const.x
const.c: In function 'f':
const.c:3:10: error: assignment of read-only location '*s'
   s[0] = 'F' ;
        ^
```

Le second intérêt est pour les utilisateurs de la fonction : ce qualificateur `const` apparaît dans le type de la fonction. Il indique donc clairement que cette fonction ne modifiera pas la zone (tableau, chaîne, variable) correspondant à l'adresse que la fonction appelante transmettra à `f`. C'est donc une information qui apporte une garantie importante (bien entendu, sous réserve que `f` soit bien écrite et ne commette pas l'imprudence décrite dans la section 16.3.5 suivante).

16.3.5 Extraction d'adresse sur variable `const`

Le fait de qualifier une définition de variable comme `const` n'empêche pas d'obtenir son adresse avec l'opérateur `&` pour la mémoriser dans un pointeur sans aucune qualification `const`. De ce fait, il devient alors possible de modifier cette variable, prétendument constante, en déréférençant le pointeur. Toutefois, lors de la compilation un avertissement est émis.

```
#include <stdio.h>

int main () {
   const int x = 5 ;
   int *p = &x ;
   (*p) = (*p) + 10 ;
   printf ("x: %d\n", x) ;
   return 0 ;
}
```

```
$ gcc -Wall const.c -o const.x
const.c: In function 'main':
const.c:8:13: warning: initialization discards 'const' qualifier from pointer target type
 [-Wdiscarded-qualifiers]
   int *p = &x ;
            ^
$ ./const.x
x: 15
```

Le problème est double. D'une part, la mémorisation de l'adresse de la variable `x`, initialement déclarée constante, provoque une possible violation du contrat de constance que nous avions annoncé. Et dans notre exemple, la violation est réelle puisque nous écrivons dans `x` au travers de `*p`.

Important

D'autre part, le compilateur ayant été averti que `x` est immuable, il peut en profiter pour faire des optimisations qui dépendent du contexte du programme, du modèle du compilateur, de sa version, du niveau des options d'optimisation. Ainsi, le comportement de ce programme est imprédictible.

Le résultat d'exécution ci-dessus a été obtenu avec `GNU gcc` 7.5.0 (les options d'optimisation ne changent rien). Le résultat d'exécution ci-dessous, totalement différent, est obtenu avec le compilateur `clang` 6.0.0.

```
$ clang -Wall const.c -o const.x
const.c:8:9: warning: initializing 'int *' with an expression of type
      'const int *' discards qualifiers
      [-Wincompatible-pointer-types-discards-qualifiers]
   int *p = &x ;
        ^   ~~
1 warning generated.
$ ./const.x
x: 5
```

Enfonçons le clou avec `GNU gcc` 12.2.0 qui provoque un affichage différent selon niveau d'optimisation demandé lors de la compilation. L'option `-O` suivi d'un chiffre (entre 0 et 3 inclus) définit le niveau d'agressivité des optimisations.

```
$ gcc -Wall const.c -o const.x
const.c: In function 'main':
const.c:8:13: warning: initialization discards 'const' qualifier from pointer target type
 [-Wdiscarded-qualifiers]
    8 |    int *p = &x ;
      |             ^
$ ./const.x
x: 15
$ gcc -Wall -O2 const.c -o const.x
const.c: In function 'main':
const.c:8:13: warning: initialization discards 'const' qualifier from pointer target type
 [-Wdiscarded-qualifiers]
    8 |    int *p = &x ;
      |             ^
$ ./const.x
x: 5
```

16.4 Qualificateur `volatile`

Qualifier une définition ou une déclaration de variable avec le mot-clé `volatile` indique au compilateur de ne faire aucune supposition quant à sa valeur entre deux lectures, qu'il y ait eu écriture explicite ou non dedans. Ainsi, toute lecture de cette variable provoquera forcément l'accès à la zone mémoire qu'elle représente. Cela empêche le compilateur d'effectuer des optimisations telles que stocker la variable dans un registre (qui est plus rapide d'accès que la mémoire vive).

```
volatile long int x ;
```

Ce qualificateur est indispensable dans le contexte de la programmation parallèle, où plusieurs fils d'exécution (i.e. fonctions d'un programme, en grossière approximation) tournent en même temps tout en partageant la mémoire. En effet, dans ce contexte le séquencement des lectures et écritures n'est plus calculable à la compilation. C'est également ce qui peut se produire lors de la programmation bas niveau puisque certaines zones mémoire peuvent être modifiées par le matériel sans intervention du logiciel.

16.5 Tableaux à longueur variable (VLA)

Le chapitre 9 a présenté les tableaux statiques. Leur taille est une constante calculable lors de la compilation. Le chapitre 12 a présenté l'allocation dynamique comme moyen d'allouer de la mémoire à l'exécution. La taille d'une telle allocation peut donc être calculée durant l'exécution.

Certaines versions des normes du langage C rendent possible l'allocation de tableaux dont la taille est déterminée à l'exécution, mais sans utiliser la fonction d'allocation dynamique `malloc`. Ces tableaux sont dits *à longueur variable*, en anglais "*variable length array*", abrégé en VLA.

Important

Il est absolument déconseillé par les recommandations de différentes normes (ANSSI [a] pour ne citer qu'elle) d'utiliser ce type de tableaux (i.e. les VLA). Dans le cadre de vos programmes (au moins en 1A), il vous est demandé de ne jamais y faire appel.

a. https://cyber.gouv.fr/publications/regles-de-programmation-pour-le-developpement-securise-de-logiciels-en-langage-c

Important

Cette notion est abordée dans ce document car elle existe et si vous la rencontrez, vous saurez de quoi il en retourne. Cette section est donc présente pour vous convaincre de pourquoi vous devrez l'éviter, sauf cas très particuliers pour lesquels vous pourrez justifier l'absence de risque.

Une déclaration de VLA a la même forme qu'une allocation statique, à l'exception de sa taille qui est une expression arbitraire qui n'est pas une constante. De ce fait, un VLA est forcément un tableau local à un bloc (*a fortiori* une fonction). Un VLA étant une variable locale, il est alloué sur la pile. Tous les exemples qui suivent sont des VLA.

```
#include <string.h>

... f (int x, char *s) {
  int t1[x] ;
  int i = 2 * x + 3 ;
  float t2[i + 10] ;
  float matrix[x][2 * x] ;
  char str[1 + strlen (s)] ;
  ...
}
```

Voyons maintenant pourquoi cette forme d'allocation en apparence si pratique (qui relègue `malloc` au rang de tracasserie technique particulièrement pénible) est à proscrire. Les deux raisons qui suivent ne sont en réalité pas des nouveautés : elles convoquent des concepts déjà présentés dans les chapitres précédents.

16.5.1 Ne pas utiliser de VLA : raison 1

La première raison relève des risques de débordement de pile. Comme déjà expliqué en section 12.3.1, la pile est un espace mémoire de taille modeste par rapport au tas (où sont faites les allocations dynamiques par la fonction `malloc`). Or, comme expliqué ci-dessus, un VLA est alloué dans la pile.

Allouer un tableau d'une grande taille calculée à l'exécution pourra avoir le même effet qu'allouer un tableau statique de grande taille : pas assez de place dans la pile. Dans ce cas, le programme s'interrompra abruptement (encore une fois, se référer à la section 12.3.1). Quand nous allouons un tableau statique, sa taille est explicite, clairement visible. Si nous écrivons la déclaration :

```
int t[100000000] ;
```

il est manifeste que nous sommes en train de faire quelque chose de fort discutable. Par contre, si la taille est masquée derrière un calcul compliqué, il devient nettement plus difficile de s'en rendre compte.

De plus, avec un VLA, il n'y a aucun moyen de détecter le manque de mémoire : ça plantera sans prévenir. À l'inverse, avec un `malloc`, il suffit de vérifier que le pointeur retourné n'est pas `NULL` pour s'assurer que l'allocation a réussi.

Important

En résumé, un programme utilisant un VLA pourra planter alors qu'il aurait pu très bien fonctionner avec un `malloc`.

16.5.2 Ne pas utiliser de VLA : raison 2

La seconde raison, plus sournoise, est celle décrite en section 12.1.2. Un VLA, comme toute variable locale, est détruit à la sortie de la fonction qui le crée (plus généralement, du bloc qui le définit). Il est courant de devoir allouer dynamiquement de la mémoire (des tableaux) pour les faire retourner par la fonction qui les crée ou pour mémoriser ces tableaux (i.e. leur adresse de début) dans des structures de données devant survivre à la fonction (variables globales par exemple ou membre d'une structure de données elle-même retournée par la fonction).

Si nous utilisons un VLA, puisqu'il est détruit à la fin de l'appel de la fonction, tout traitement tentant d'y accéder ultérieurement accédera en fait à une zone mémoire qui n'est plus valide. Le pire est que cette zone appartiendra toujours à l'espace mémoire du programme donc aucun accès illégal ne sera détecté par le système d'exploitation.

Par contre, les appels de fonctions ultérieurs pourront avoir réutilisé cette zone pour leur propres variables locales (paramètres et autres informations). Donc le programme détruira de manière non déterministe (au gré des futurs appels de fonctions) les informations stockées dans cet ancien tableau… sur lequel « on croit » encore pouvoir compter. C'est donc une source de bugs difficilement analysables et localisables.

Avec un `malloc`, la zone mémoire serait allouée ailleurs que sur la pile (dans le tas) et survivrait donc à la fin de la fonction (tant que personne n'aura fait de `free`). Il en résulte que le traitement serait parfaitement sûr.

16.5.3 Conclusion à propos des VLA

Important

Afin d'éviter de devoir se demander (au risque d'oublier de temps en temps voire tout le temps) si nous sommes dans un cas favorable aux VLA ou l'inverse, autant adopter un mode d'allocation qui fonctionne dans tous les cas : `malloc`. Et bannir les VLA…

Comme indiqué précédemment, cela permet de surcroît de vérifier que l'espace mémoire demandé est bien disponible. Certes, il faut vérifier le pointeur retourné `malloc`. Mais au moins, cela évite (si nous le faisons effectivement) que notre programme ne plante sèchement. Et un programme bien fait vérifie toujours que les ressources qu'il demande lui ont bien été attribuées.

16.6 Débordements, arrondis

Comme présenté dans la section 2.1, les entiers et les réels sont codés sur un nombre fini de bits. De ce fait, la plage des valeurs qu'ils peuvent représenter est finie et peut silencieusement être dépassée.

16.6.1 Problèmes d'entiers

Les entiers sont codés en base 2, avec une interprétation de leur configuration de bits différente en fonction de s'ils sont signés ou non. Néanmoins, ces deux interprétations ont la bonne propriété d'être compatibles et d'autoriser, sous certaines conditions à effectuer des opérations arithmétiques mélangeant signés et non signés.

Un entier non signé est codé comme sa décomposition en somme de puissances de deux. C'est la notation positionnelle habituelle en mathématique, mais en base 2 à la place de notre base 10 quotidienne. Dans une écriture de gauche à droite, on appelle bit de *poids fort* celui le plus à gauche et bit de *poids faible* celui le plus à droite. Par convention les bits d'un mot binaire de taille n sont numérotés de gauche à droite de $n-1$ à 0. La valeur entière dénotée par ce mot où b_i représente le bit de poids i est donc :

$$\sum_{i=0}^{n-1} b_i \times 2^i$$

Ainsi, 421 s'écrit 0000000110100101 en binaire, grâce à la décomposition $2^8 + 2^7 + 2^5 + 2^2 + 2^0$ (sur 16 bits).

Un entier signé est codé en utilisant le complément à deux qui garantit l'unicité du zéro et une forme de compatibilité entre les opérations entre signés et non signés. Dans ce codage, le bit de poids fort est utilisé pour dénoter le signe de l'entier : 0 l'entier est positif, 1 il est négatif.

Pour un entier positif le complément à deux coïncide avec le codage positionnel introduit ci-dessus. Pour obtenir le codage de l'opposé d'un entier de longueur l, le complément à deux consiste à :

1. inverser les bits de son écriture binaire,
2. ajouter 1 (en conservant seulement les l bits de poids fort).

En considérant un entier sur 4 bits, nous pouvons représenter 16 valeurs, 8 négatives, 8 positives, sachant que 0 est positif. Ainsi, -8 est la plus petite valeur représentable et 7 la plus grande.

Décimal	0	1	4	5	-5	7
Binaire	0000	0001	0100	0101	1011	0111
Négation	1111	1110	1011	1010	1100	1000
+1	0000	1111	1100	1011	1101	1001
Décimal	0	-1	-4	-5	5	-7

16.6.1.1 Débordement

Considérons les deux entiers décimaux 9 et 7, non signés, codés sur 4 bits. Chacun est représentable sur 4 bits : $9 \equiv 1001$ et $7 \equiv 0111$. Leur somme est égale à $16 \equiv 10000$. Cette valeur nécessite un bit de plus que les 4 disponibles pour nos entiers : le bit de poids fort est perdu et le résultat du calcul est 0000. Il y a silencieusement eu un débordement de capacité.

Important

L'arithmétique entière est réalisée modulo la taille des entiers mis en jeu. En cas de débordement de capacité au cours d'une opération entre entiers, aucune erreur n'est signalée.

Pour aller plus loin

Il existe un bit au sein du micro-processeur qui permet de déterminer s'il y a eu débordement (bit usuellement nommé "*Carry*"). Le langage C, comme la majorité des langages de programmation, a fait le choix de ne pas consulter ce bit à chaque opération arithmétique pour ne pas pénaliser dramatiquement le temps de calcul. C'est à la personne qui rédige le programme de faire l'effort de cette vérification dans le cas où elle estime que c'est indispensable (ou que des normes drastiques l'y obligent).

Bien entendu, le même problème se passe entre des entiers négatifs ou entre des négatifs et des positifs. Considérons la soustraction des deux entiers décimaux -8 et 1. La soustraction n'étant jamais que l'addition de l'opposé, cela revient à additionner -8 et -1. Chacun est représentable sur 4 bits : $-8 \equiv 1000$ et $-1 \equiv 1111$. Leur somme est égale à 10111 qui, tronquée sur 4 bits, est égale $0111 \equiv 7$ et non -9.

Nous avons vu que le codage en complément à deux sépare la plage de représentation en deux : positifs et négatifs. Pour un nombre fixé de bits, la valeur absolue du plus petit négatif est toujours supérieure de 1 à la valeur du plus grand entier représentable. Sur 4 bits, la plage va de -8 à 7, sur 8 bits elle va de -128 à 127, sur 16 bits de -32768 à 32767, etc.

Si nous cherchons à prendre l'opposé, sur 4 bits, de $-8 \equiv 1000$, nous obtenons $0111 + 1 = 1000$. Cette configuration de bits est bien entendu à interpréter comme un entier signé puisque nous avons calculé un opposé. Et il se trouve, que c'est la valeur signée -8 elle-même.

Pour terminer, des problèmes d'interprétation surviennent également lorsque l'on souhaite interpréter un entier signé comme un non signé et inversement. C'est exactement ce qui se passe lors d'une conversion silencieuse (ou explicite avec l'opération de transtypage décrite en section 16.11) entre signé et non signé.

Si dans la configuration binaire de l'entier le bit de poids fort est à 0, tout se passe bien. Par contre, s'il est à 1, interpréter aveuglement cette configuration comme signée donnera un nombre négatif et comme non signée un grand entier positif.

16.6.1.2 Mélange signé / non signé

Dans la section précédente (16.6.1), nous avons dit que les opérations arithmétiques étaient compatibles avec à la fois la représentation des entiers signés et non signés. Pour autant, cela ne signifie pas que le mélange des deux soit sans danger.

Bien entendu les problèmes de débordement expliqués pour chaque catégorie d'entiers (signés ou non) restent présents lorsque ces types sont mélangés. Mais cela ne se limite pas aux opérations arithmétiques. Le programme suivant effectue une comparaison entre deux `int`, l'un signé et l'autre non.

```c
#include <stdio.h>

int main () {
  int s = -1 ;
  unsigned u = 1 ;
  if (s < u) printf ("-1 < 1\n") ;
  else printf ("1 < -1\n") ;
  return 0 ;
}
```

```
$ gcc -Wall mixed.c -o mixed.x
$ ./mixed.x
1 < -1
```

Pas même un petit avertissement, mais un résultat peut-être inattendu. La norme du langage C impose qu'en cas de mélange, l'entier signé soit converti silencieusement entier non signé. En accord avec le codage en complément à 2, la valeur -1 a son bit de poids fort mis à 1. Avec une interprétation non signée, ce bit fait partie de la magnitude de l'entier, ce qui correspond à un entier plutôt grand (i.e. $\geqslant 2147483647$ pour un `int`).

Une autre erreur courante qui s'apparente à un tel mélange est celui du test d'un non signé par rapport à 0. Ce programme effectue une boucle tant que la variable `i` est supérieure ou égale à 0.

```c
#include <stdio.h>

int main () {
  for (unsigned int i = 3; i >= 0; i
      --)
    printf ("i = %u\n", i) ;
  return 0 ;
}
```

```
$ gcc -Wall unstest.c -o unstest.x
$ ./unstest.x
i = 3
i = 2
i = 1
i = 0
i = 4294967295
i = 4294967294
...
```

Malheureusement, cette variable est de type `unsigned int` : elle ne peut donc pas être négative. Nous sommes donc implicitement en train d'attendre qu'un entier non signé devienne signé (*de facto*, en particulier négatif). Fidèle au mécanisme d'arithmétique modulo, soustraire 1 à 0 sur un non signé fait retomber sur le plus grand entier. Ainsi la condition de fin ne sera jamais atteinte et la boucle est infinie.

Dans le programme ci-dessus, la raison de la boucle infinie est clairement visible au travers de l'affichage. Cela est dû au fait que nous avons bien utilisé le format `%u` dédié à l'affichage des entiers non signés. Si en plus de l'erreur déjà expliquée, nous n'avions pas respecté le bon format et utilisé de manière aveugle et systématique `%d` en se disant « *Pff, de toute façon un entier c'est un entier* », la source de l'erreur aurait été infiniment moins limpide...

```c
#include <stdio.h>

int main () {
  for (unsigned int i = 3; i >= 0; i
      --)
    printf ("i = %d\n", i) ;
  return 0 ;
}
```

```
$ gcc -Wall unstest2.c -o unstest2.x
$ $ ./unstest2.x
i = 3
i = 2
i = 1
i = 0
i = -1
i = -2
...
```

16.6.1.3 Division entière

La division entière est également une source de bugs silencieux. De par sa nature, elle fournit un résultat sans partie fractionnaire puisqu'elle la tronque purement et simplement.

Ainsi, `(1 / 4) * 4` n'est pas égal à `(1 * 4) / 4`. Dans le premier cas le résultat est 0.25 tronqué, donc 0, multiplié par 4 ce qui donne sans grande surprise 0. Dans le second, c'est 4 divisé par 4 ce qui donne 1.

La nature des scalaires mis en jeu dans une opération arithmétique détermine quelle version de cette opération est utilisée. Ce n'est pas le type de l'éventuelle variable dans laquelle on stockerait le résultat du calcul comme le montre l'exemple suivant.

```c
#include <stdio.h>

int main () {
  float t = 1 / 4 * 4 ;
  printf ("%f\n", t) ;
  return 0 ;
}
```

```
$ gcc -Wall div.c -o div.x
$ ./div.x
0.000000
```

L'ordre dans lequel le compilateur choisit d'évaluer les expressions a également un impact. Nous pourrions nous dire qu'en écrivant le 4 le plus à droite sous forme explicite d'un flottant le compilateur en déduirait qu'il doit utiliser les opérations arithmétiques flottantes.

```c
#include <stdio.h>

int main () {
  float t = 1 / 4 * 4.0 ;
  printf ("%f\n", t) ;
  return 0 ;
}
```

```
$ gcc -Wall div.c -o div.x
$ ./div.x
0.000000
```

En l'occurrence le résultat est le même. Si par contre nous écrivons l'une des deux opérandes de la division comme des flottants explicites, le résultat obtenu est enfin celui escompté.

```c
#include <stdio.h>

int main () {
  float t1 = 1.0 / 4 * 4 ;
  float t2 = 1 / 4.0 * 4 ;
  printf ("%f %f\n", t1, t2) ;
  return 0 ;
}
```

```
$ gcc -Wall div.c -o div.x
$ ./div.x
1.000000 1.000000
```

16.6.2 Problèmes de réels

16.6.2.1 Conversion vers entier

Puisqu'un réel peut avoir une partie fractionnaire mais pas un entier, de manière évidente une conversion d'un `float` ou `double` vers n'importe quel type d'`int` provoquera une approximation par troncature. Si de plus la conversion est faite vers un `unsigned`, le signe sera de plus perdu. Pour terminer, dans le cas où la valeur flottante serait plus grande que la capacité de l'entier, il y aura en plus débordement.

```c
#include <stdio.h>

int main () {
  float mpi = -3.14 ;
  int int_mpi = mpi ;
  unsigned uint_mpi = mpi ;
  float very_big = 10e30 ;
  int too_small = very_big ;
  printf ("%f %d %u\n",
          mpi, int_mpi,
          uint_mpi) ;
  printf ("%f %d\n",
          very_big, too_small)
              ;
  return 0 ;
}
```

```
$ gcc -Wall f2i.c -o f2i.x
$ ./f2i.x
-3.140000 -3 0
99999998482432072951095948738 56.000000 2147483647
```

16.6.2.2 Problème de précision

Comme annoncé dans la section 2.1.2, il existe une notion de *précision* qui définit la distance entre deux valeurs réelles représentables consécutives. Dit autrement, il existe des « trous » entre les réels consécutifs représentables. Cette précision dépend d'une part de si le réel est de type `float` ou de type `double`, mais également de la valeur à représenter. Schématiquement, plus on se rapproche de

0 en valeur absolue, plus la précision augmente (i.e. la taille des « trous » diminue) et plus on s'en éloigne plus la précision diminue (i.e. la taille des « trous » augmente). Cette précision est à l'origine de nombreuses approximations induisant d'étranges comportements. Ces comportements ne sont pas propres au langage C : tout langage de programmation qui utilise les flottants respectant la norme IEEE 754 provoque les mêmes comportements. Et les micro-processeurs modernes incluent en leur sein toute l'électronique permettant de réaliser des opérations flottantes selon cette norme.

16.6.2.2.1 Flottants différents égaux Ainsi, des constantes littérales textuellement écrites dans un programme, visuellement différentes pourront s'avérer égales. Dit autrement, elles tombent toutes dans un « trous » et se trouvent représentées par le même flottant le plus proche.

```c
#include <stdio.h>

int main () {
  float fl1 = 48431.1231 ;
  float fl2 = 48431.1239 ;
  float fl3 = 48431.1250 ;
  printf ("fl1: %f  fl2: %f  fl3: %f\n", fl1, fl2, fl3) ;
  printf ("fl1 == fl2 ? %d\n", (fl1 == fl2)) ;
  printf ("fl2 == fl3 ? %d\n", (fl2 == fl3)) ;
  printf ("fl1 == fl3 ? %d\n", (fl1 == fl3)) ;
  fl1 = fl1 + 0.0001 ;
  printf ("fl1 + 0.0001: %f\n", fl1) ;
  return 0 ;
}
```

```
$ gcc -Wall thiskillsme.c -o thiskillsme.x
$ ./thiskillsme.x
fl1: 48431.125000  fl2: 48431.125000  fl3: 48431.125000
fl1 == fl2 ? 1
fl2 == fl3 ? 1
fl1 == fl3 ? 1
fl1 + 0.0001: 48431.125000
```

Les trois constantes littérales ont toutes été représentées par le flottant le plus proche, à savoir 48431.125000. Nous pouvons remarquer au passage que rajouter 0.0001 à une de ces variables ne suffit pas à « franchir le trou » puisque le résultat reste égal à la valeur initiale.

16.6.2.2.2 Flottants égaux différents Les calculs sur les flottants étant contraints par la précision, des erreurs d'arrondi peuvent faire plus ou moins légèrement dévier les résultats par rapport à ce que nous attendrions d'une arithmétique exacte. Dans ce programme, trois variables sont définies et initialisées avec des valeurs réelles somme toute assez banales.

```c
#include <stdio.h>
#include <math.h>
#define EPSILON (1e-6)

int main () {
  double fl1 = 0.1 ;
  double fl2 = 0.2 ;
  double fl3 = 0.3 ;
  double fl4 = fl1 + fl2 ;
  printf ("fl1: %f  fl2: %f  fl3: %f  fl4: %f\n", fl1, fl2, fl3, fl4) ;
  printf ("fl3 = fl4 ? %d\n", (fl3 == fl4)) ;
```

→

```c
  printf ("fl3 ~ fl4 ? %d\n", (fabs (fl3 - fl4) < EPSILON)) ;
  return 0 ;
}
```

```
$ gcc -Wall thiskillsmeagain.c -o thiskillsmeagain.x
$ ./thiskillsmeagain.x
fl1: 0.100000  fl2: 0.200000  fl3: 0.300000  fl4: 0.300000
fl3 = fl4 ? 0
fl3 ~ fl4 ? 1
```

Lors de l'exécution, il semble que les trois premières variables (`fl1`, `fl2` et `fl3`) aient été initialisées correctement, l'affichage l'atteste. La variable `fl4` doit être initialisée avec 0.3, ce que l'affichage atteste à nouveau. De ce fait, `fl4` doit être égale à `fl3`, ce que le test avec == dément.

Dans la réalité `fl1`, `fl2` et `fl3` n'ont pas été initialisées exactement avec 0.1, 0.2 et 0.3 : elles l'ont été avec les flottants les plus proches car ces trois valeurs ne sont pas exactement représentables sur un `double` (l'ordre de grandeur de l'approximation est 10^{-17}). Lors du calcul de la valeur de `fl4`, l'addition a été effectuée avec ces flottants approximés, donnant un résultat également approximé. L'opérateur == vérifiant l'égalité strict des bits, n'a donc pas conclu à l'égalité.

Important

Il ne faut jamais comparer l'égalité entre deux expressions e_1 et e_2 de types réels (`float` ou `double` avec l'opérateur ==.
Il faut les comparer à une tolérance près, c'est-à-dire en vérifiant si $|e_1 - e_2| \leqslant \epsilon$ pour un « certain » ϵ.

Comme l'indique le dernier affichage qui effectue un test modulo `EPSILON`, ces deux valeurs sont « suffisamment » proches pour être considérées égales. La fonction `fabs` calcule la valeurs absolue d'un flottant et nécessite l'inclusion du fichier d'en-tête `<math.h>`.

Comment savoir quelle valeur prendre pour ϵ afin de considérer que deux valeurs sont « suffisamment » proches pour être considérées égales ? Il n'y a pas de valeur universelle : cela dépend de la nature du calcul. Si le calcul vérifie les positions de deux molécules, ou de deux galaxies, il est évident que les « distances » indiquant que deux de ces entités sont au même endroit seront fort différentes.

Pourquoi les affichages nous ont-ils induits en erreur ? La raison est que la fonction `printf` réalise un affichage lui-même approximatif. Elle n'affiche pas la bonne vingtaine de décimales significatives pour ces valeurs.

Pour terminer, il est intéressant de noter qu'avec ces valeurs, si nous utilisons des `float` à la place de `double`, les approximations se comportent différemment et se compensent : le test avec == fonctionne (ou plutôt « tombe en marche »). Bien entendu, cela ne signifie absolument pas que ce programme soit correct. Il suffit de changer les valeurs manipulées pour retomber sur le même problème.

```c
#include <stdio.h>

int main () {
  float fl1 = 0.1 ;
  float fl2 = 0.2 ;
  float fl3 = 0.3 ;
  float fl4 = fl1 + fl2 ;
  printf ("fl1: %f  fl2: %f  fl3: %f  fl4: %f\n", fl1, fl2, fl3, fl4) ;
  printf ("fl3 = fl4 ? %d\n", (fl3 == fl4)) ;
  return 0 ;
}
```

```
$ gcc -Wall thiskillsmeagain2.c -o thiskillsmeagain2.x
$ ./thiskillsmeagain2.x
fl1: 0.100000  fl2: 0.200000  fl3: 0.300000  fl4: 0.300000
fl3 = fl4 ? 1
```

16.6.2.2.3 Mauvais conditionnement La précision dépendant de la proximité de 0, combiner de grandes valeurs réelles avec des petites peut provoquer de mauvais conditionnements des calculs menant à des résultats incohérents. La précision de l'une peut devenir insignifiante par rapport à celle de l'autre.

```c
int main () {
  float small = 0.001 ;
  float large = 1e10 ;
  float old_large = large ;
  printf ("%f %f\n", small, large) ;
  for (int i = 0; i < 10000; i++) {
    large = large + small ;
    printf ("%f (eq? %d)\n",
            large, (large == old_large))
                ;
  }
  return 0 ;
}
```

```
$ gcc -Wall mixed.c -o mixed.x
$ ./mixed.x
10000000000.000000 (eq? 1)
10000000000.000000 (eq? 1)
10000000000.000000 (eq? 1)
10000000000.000000 (eq? 1)
...
```

Dans ce programme, la quantité `small` à rajouter à chaque tour de boucle à `large` est bien inférieure à la précision autour de 10^{10}. Ainsi, ajouter `small` à `large` ne change pas la valeur de `large`. Dit autrement, `small` est trop petite pour franchir le « trou » entre la valeur de `large` et sa suivante. Le test d'égalité (bit-à-bit) montre bien que la variable `large` ne change pas de sa valeur initiale d'une itération à l'autre.

16.7 Expression conditionnelle (?)

Les conditionnelles présentées en section 5.1 sont des instructions : elles modifient le flot de contrôle du programme mais n'ont pas de valeur. L'opérateur ternaire `? :` (point d'interrogation / deux points) permet d'exprimer une expression conditionnelle dont la valeur dépend d'une condition :

$$expr\text{–}cond \texttt{ ? } expr\text{–}then \texttt{ : } expr\text{–}else$$

Si $expr\text{–}cond$ s'évalue en « vrai » (i.e. $\neq 0$) alors la valeur de l'expression sera celle de $expr\text{–}then$. Dans le cas contraire, ce sera celle de $expr\text{–}else$. Il s'agit donc du « if » standard des langages fonctionnels comme OCaml (Python dispose aussi d'une expression conditionnelle).

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

void print_bool (bool b) {
  printf ("%s", b ? "true" : "false") ;
}

int sign (int i) {
  return (i < 0) ? -1 : ((i == 0) ? 0 : 1) ;
}

int main () {
  print_bool (true) ; printf ("\n") ;
  print_bool (false) ; printf ("\n") ;
  printf ("%d %d %d\n", sign (-5), sign (0),
      sign (7)) ;
  return 0 ;
}
```

```
$ gcc -Wall eif.c -o eif.x
$ ./eif.x
true
false
-1 0 1
```

Les parenthèses ne sont pas indispensables (contrairement à l'expression de condition de la construction `if`). Néanmoins elles peuvent aider à la lisibilité comme dans le cas de la fonction `sign` qui imbrique deux expressions conditionnelles.

16.8 Séquence d'expressions (,)

Il est naturel d'enchaîner des instructions en séquence : chaque instruction est terminée par un point-virgule. Les effets des instructions se suivent dans le même ordre que se suivent les instructions. L'opérateur binaire `,` (virgule) permet de former une expression composée de l'enchaînement de deux expressions.

`expr`$_1$`, expr`$_2$

L'expression `expr`$_1$ est évaluée, son résultat est ignoré, puis `expr`$_2$ est évalué et sa valeur est celle de l'expression globale. L'associativité de l'opérateur `,` permet de combiner un nombre arbitraire d'expressions : elles sont évaluées de gauche à droite et la valeur globale est celle de l'expression la plus à droite.

L'utilité de cette construction est assez subtile : puisque, comme expliqué tout au début du chapitre 4, une expression est « quelque chose qui vaut », quel serait l'intérêt d'enchaîner des calculs de valeurs pour les ignorer ?

Dans cette même introduction du chapitre 4, nous avons presque immédiatement tempéré cette affirmation, en disant que certaines instructions, comme l'affectation (cf. section 4.2.1) pouvaient en fait être utilisées comme expressions. C'est le cas de toutes les formes d'affectations présentées en section 4.2.3. Et puis, un appel de fonction est une expression comme expliqué en section 4.1.5. Pourtant, il peut avoir des effets comme l'affichage, la modification de variables globales, la modification de zones mémoire par passage par adresse, etc.

Donc, enchaîner des expressions peut avoir de l'intérêt dès lors qu'elles ont un effet mais que nous sommes « coincés dans un endroit » où, syntaxiquement, seule une expression est autorisée. C'est le cas dans des conditions de boucles et de manière absolument idiomatique dans le cas du `for` qui est à peu près la seule utilisation répandue (et facilement justifiable).

Lorsqu'une boucle `for` possède un unique compteur, elle est souvent écrite sous la forme :

```c
for (int i = 0; i < len; i++) { ... }
```

où les 3 parties du `for` sont des expressions (cf. section 5.2.2). Si maintenant la boucle possède deux compteurs `i` et `j`, nous allons vouloir les déclarer d'un coup dans la partie gauche et les modifier dans la partie droite du `for`. Il n'y a alors pas d'autre solution que d'utiliser l'opérateur virgule pour conserver une boucle `for` (cela dit, un `while` irait fort bien).

```
for (int i = 0, j = 50; (i < j); i++, j-) { ... }
```

Important

L'utilisation de l'opérateur virgule en dehors du cas illustré avec la boucle `for` est très technique, source de confusions et peut nuire à la lisibilité des programmes. Cet opérateur ne doit pas être utilisé en remplacement de la séquence d'instructions quand elle est utilisable.

Considérons le programme suivant qui affecte une variable avec une expression séquence. *A priori*, puisqu'une expression séquence est évaluée de gauche à droite, `i` doit recevoir la valeur 10. Pour terminer le `return` semble retourner un nuplet, comme c'est faisable dans d'autres langages de programmation (Ocaml ou Python par exemple).

```c
#include <stdio.h>

int main () {
  int i ;
  i = 5, 10 ;
  printf ("%d\n", i) ;
  return (0, 20, 30, 40) ;
}
```

```
$ gcc -Wall oddsec.c -o oddsec.x
oddsec.c: In function 'main':
oddsec.c:5:8: warning: right-hand operand of comma expression has no effect [-Wunused-value]
   i = 5, 10 ;
        ^
oddsec.c:7:12: warning: left-hand operand of comma expression has no effect [-Wunused-value]
   return (0, 20, 30, 40) ;
            ^
oddsec.c:7:16: warning: left-hand operand of comma expression has no effect [-Wunused-value]
   return (0, 20, 30, 40) ;
                ^
oddsec.c:7:20: warning: left-hand operand of comma expression has no effect [-Wunused-value]
   return (0, 20, 30, 40) ;
                    ^
$ ./oddsec.x
5
$ echo $?
40
```

Lors de la compilation plusieurs avertissements sont émis, ce qui est normal car dans notre exemple caricatural, les expressions mises en séquence n'ont aucun effet, donc sont réellement totalement inutiles.

Lors de l'exécution, nous constatons que `i` ne vaut pas 10 : elle vaut 5. D'ailleurs, le premier avertissement nous a dit que la partie droite de l'expression virgule était inutile. La priorité des opérateurs a en fait interprété cette affectation comme :

```
(i = 5), 10 ;
```

où l'expression de la partie gauche est l'affectation et `10` est celle de la partie droite. L'opérateur virgule a une priorité plus faible que l'affectation (cf. annexe 18.2). Donc, l'affectation a d'abord été effectuée et le résultat global de cette « instruction » est la valeur 10, qui est totalement ignorée.

Pour ce qui est du `return` qui pourrait induire en erreur les débutants ayant utilisé des nuplets dans leur langages précédents, c'est bien la dernière valeur qui est retournée par le programme. Cela est visible en demandant au shell d'afficher le code de retour de la dernière commande exécutée avec `echo $?`.

16.9 L'instruction `goto`

Les différentes structures de contrôle présentées, boucles, conditionnelles, séquence permettent une programmation qualifiée de *structurée*. Le flot de contrôle, c'est-à-dire l'enchaînement des instructions, est clairement identifié au travers des itérations et des conditions. Ce n'est pas une garantie de lisibilité et de maintenabilité, mais c'est au moins une condition indispensable.

Nous avons déjà vu deux instructions dont l'utilisation sans retenue pouvait sérieusement altérer cette manière de décrire un algorithme : `break` en section 5.3.1 et `continue` en section 5.3.2. Nous avions d'ailleurs insisté sur la mesure dont il faut faire preuve quant à leur utilisation (en dehors du `switch` pour le `break` bien entendu – cf. section 5.1.3).

Mais il y a pire : l'instruction `goto` qui permet de sauter à une instruction arbitraire au sein d'une fonction. La destination d'un `goto` est un identificateur correspondant à un label qui identifie une instruction.

```c
#include <stdio.h>

int main () {
  printf ("Début.\n") ;
  goto label_ici ;
  printf ("Suite.\n") ;
 label_ici:
  printf ("Fin.\n") ;
  return 0 ;
}
```

```
$ gcc -Wall goto.c -o goto.x
$ ./goto.x
Début.
Fin.
```

Important

L'utilisation de l'instruction `goto` est absolument déconseillée si ce n'est bannie par toutes les bonnes pratiques et normes de développement tant elle nuit à la compréhension et la maintenabilité des programmes. La seule exception (car comme toujours il y en a une) est expliquée un peu plus loin dans cette section.

Avant de voir comment `goto` peut quand même être raisonnablement utilisé, voyons un exemple (très simple) de ce qu'il ne faut jamais faire en terme d'architecture d'algorithme. La fonction `horrible_exists` vérifie si une valeur `v` se trouve dans un tableau `t` d'entiers de longueur `len`.

```c
int horrible_exists (int *t, int tlen, int v) {
  bool found = false ;
  int i = 0 ;
 test:
  if (i >= tlen) goto end ;
  if (t[i] == v) {
    found = true ;
    goto end ;
  }
  i++ ;
  goto test ;
 end:
  return found ;
}
```

Bien évidemment, une version structurée avec une simple boucle `for` s'écrit de manière nettement plus concise et surtout plus compréhensible :

```
int exists (int *t, int tlen, int v) {
  bool found = false ;
  for (int i = 0; i < tlen && !found; i++) {
    if (t[i] == v) found = true ;
  }
  return found ;
}
```

L'histoire de bien marquer les esprits, le soin vous est laissé de reconnaître dans le programme suivant une fonction de multiplication matricielle (la fonction `make_null_matrix` est considérée existante qui retourne une matrice initialisée à 0 partout). Bon courage (et encore, les noms de labels sont choisis pour être compréhensibles)...

```
struct mat_t {
  int m_nlines ;
  int m_ncols ;
  float **m_coefs ;
} ;

struct mat_t* horrible_matrix_mult (struct mat_t left, struct mat_t right) {
  if (left.m_ncols != right.m_nlines) return NULL ;
  int r_nlines = left.m_nlines ;
  int r_ncols = right.m_ncols ;
  struct mat_t *r_matrix = make_null_matrix (r_nlines, r_ncols) ;
  if (r_matrix == NULL) goto end_l ;
  float **r_coefs = r_matrix->m_coefs ;
  int l = 0 ;
 test_l: if (! (l < r_nlines)) goto end_l ;
  int c = 0;
 test_c: if (! (c < r_ncols)) goto end_c ;
  int i = 0;
 test_i : if (! (i < right.m_nlines)) goto end_i ;
  r_coefs[l][c] = r_coefs[l][c] + left.m_coefs[l][i] * right.m_coefs[i][c] ;
  i++ ;
  goto test_i ;
 end_i: c++ ;
  goto test_c ;
 end_c: l++ ;
  goto test_l ;
 end_l: return r_matrix ;
}
```

16.9.1 Simuler des exceptions

Les exceptions sont un mécanisme très pratique pour centraliser la gestion des erreurs. On les retrouve dans plusieurs langages comme OCaml, C++, Python, Java, etc. Malheureusement elles n'existent pas en C. Rappelons brièvement leur fonctionnement sur un exemple écrit en OCaml.

```
try
  let file1 = open_in_bin "fichier.dat" in
  let res1 = compute_machin ... in
  let res2 = compute_bidule .. in
  ...
                                                                    →
```

```
with
| Sys_error _ -> ...
| Not_found -> ...
```

Si un des calculs protégés entre `try` et `with` lève une exception, l'exécution normale est interrompue. Le programme saute alors vers le gestionnaire d'exceptions se trouvant après le `with`. Si l'exception levée correspond à l'une de ce gestionnaire, son traitement associé est exécuté. Dans le cas contraire, l'exception levée se propage à la fonction appelante qui peut avoir un gestionnaire adapté. Tous les traitements d'erreurs sont donc regroupés au même endroit et ne « polluent » le code nominal (i.e. celui qui s'exécute si aucune erreur ne survient).

Du fait de l'absence de mécanisme d'exceptions en C, la gestion des erreurs occasionne souvent une dispersion et une répétition du code de traitement de ces erreurs. Il est alors courant de voir des fonctions ayant la structure suivante. Plus on progresse dans le programme, plus il y a de ressources à libérer en cas d'erreur.

```
FILE *in = fopen (...) ;
if (in == NULL) return -1 ;
int *t = malloc (...) ;
  if (t == NULL) {
  fclose (in) ;
  return -1 ;
}
int *u = malloc (...) ;
if (u == NULL) {
  free (t) ;
  fclose (in) ;
  return -1 ;
}
...
free (u) ;
free (t) ;
fclose (in) ;
return ... ;
```

Utiliser des `goto` permettant de sauter à un point de sortie unique permet de simuler des exceptions locales à une fonction. En effet, il n'est possible de sauter qu'à un label situé dans la fonction qui effectue le `goto`. Il devient alors possible de délocaliser tous les traitements d'erreur et de nettoyage de fin de fonction au même endroit.

```
FILE *in = NULL ;          /* Valeur d'erreur par défaut. */
int *t = NULL, *u = NULL ; /* Valeur d'erreur par défaut. */
int ret_val = -1 ;         /* Valeur d'erreur par défaut. */

in = fopen (...) ;
if (in == NULL) goto cleanup ;
t = malloc (...) ;
if (t == NULL) goto cleanup ;
u = malloc (...) ;
if (u == NULL) goto cleanup ;
...
   ret_vval = ... ;
...
cleanup:
if (u) free (u) ;
```

```
    if (t) free (t) ;
    if (in) fclose (in) ;
    return ret_val ;
```

Cela implique que les variables soient initialisées avec les valeurs qui correspondent à des erreurs. C'est uniquement lorsque le traitement qui doit leur donner une autre valeur a réussi qu'elles changent. Ce mécanisme fait partie des pratiques de *programmation défensive*.

Important

Un programme ne doit jamais effectuer un retour arrière avec un `goto`, c'est-à-dire revenir à un label dénotant une instruction précédente dans le flot de contrôle. Ce point n'est pas négociable. L'utilisation du `goto` doit être réservée à la gestion des erreurs.

16.10 Opérateurs bit-à-bit

Les opérateurs bit-à-bit permettent d'appliquer une opération au niveau des bits individuels d'une ou deux valeurs entières (ou se réduisant à un entier). Ces opérateurs sont principalement utilisés dans le cadre de la programmation bas-niveau, à l'interface avec le matériel où le positionnement de bits dans certains registres du micro-processeur a une signification matérielle plutôt que mathématique (par exemple, configuration du micro-processeur, activation/désactivation de sorties physiques, lecture d'entrée physiques, etc.). On retrouve aussi des utilisations lorsqu'il s'agit d'encoder des messages ou d'optimiser précisément la taille de données.

16.10.1 Opérateurs logiques

Un opérateur unaire applique l'opérateur logique sur chacun des bits de l'opérande séparément. Un opérateur binaire applique l'opérateur logique sur chaque couple de bits de même poids séparément.

non	et	ou	ou-exclusif
~	&	|	^

Le programme suivant affiche en hexadécimal les résultats de chaque opérateur bit-à-bit appliqué aux valeurs 12 et 10. La lecture « humaine » des entiers comme séquences de bits est facilitée par l'écriture hexadécimale. En effet, chaque paquet de 4 bits correspondant à un chiffre hexadécimal, il nous suffit de seulement mémoriser les 16 configurations binaires de 0 à 15 pour pouvoir décortiquer une valeur hexadécimale sous forme de suite de bits.

Décimal	0	1	2	3	4	5	6	7
Hexadécimal	0	1	2	3	4	5	6	7
Binaire	0000	0001	0010	0011	0100	0101	0110	0111
Décimal	8	9	10	11	12	13	14	15
Hexadécimal	8	9	A	B	C	D	E	F
Binaire	1000	1001	1010	1011	1100	1101	1110	1111

```c
#include <stdio.h>

int main () {
  int x = 12 ;     /* 1100 */
  int y = 10 ;     /* 1010 */
  printf ("%x\n", ~x) ;
  printf ("%x\n", x & y) ;
  printf ("%x\n", x | y) ;
  printf ("%x\n", x ^ y) ;
  return 0 ;
}
```

```
$ gcc -Wall bitwise.c -o bitwise.x
$ ./bitwise.x
fffffff3
8
e
6
```

Les résultats de ces opérations écrits en binaire sont les suivants. Il est aisé de les transformer en hexadécimal (comme affiché par le programme) grâce à la table ci-dessus. L'écriture « 0… » représente le fait que nos entiers sont sur 32 bits et que nous n'avons écrit que les 4 derniers bits « intéressants ».

x	y	~x	x & y	x | y	x ^ y
0…1100	0…1010	1…0011	1000	1110	0110

Important

L'opérateur `^` n'est pas l'élévation à la puissance. C'est le ou exclusif bit-à-bit (b_1 ^ $b_2 = 1$ si et seulement si un seul des deux bits est égal à 1).

16.10.2 Décalages de bits

À côté des opérateurs logiques, il existe deux opérations de décalage des bits d'une valeur entière, à droite ou à gauche, un nombre de fois donné.

`e << n` décale les bits de l'expression `e` vers la gauche `n` fois en insérant des 0 à droite. De ce fait, cette opération réalise exactement $e \times 2^n$ mais de manière extrêmement rapide car aucune opération arithmétique n'est requise (en particulier, pas d'élévation à la puissance).

`e >> n` : décale les bits de `e` vers la droite `n` fois. Si l'expression `e` est non signée, alors des 0 sont insérés à gauche. Par contre, si `e` est signée, alors le bit inséré à gauche est dépendant du compilateur. Cependant, c'est généralement le même bit que celui de poids fort dans `e` (on parle alors de décalage *arithmétique* plutôt que *logique*. Ainsi, cette opération réalise $\lfloor e/2^n \rfloor$.

L'expression représentant le nombre de décalages `n` doit être supérieure ou égale à 0 sans quoi le comportement de ces opérateurs est indéfini.

Ces opérateurs permettent de fabriquer facilement des constantes puissances de 2 : `1 << 7` est la constante entière ayant son 8ème bit (en partant de la droite) à 1. Ils peuvent également permettre de récupérer des bits particuliers d'une valeur : `(e >> 5) & 1` permet de récupérer le 6ème bit (en partant de la droite) de la valeur de l'expression `e`.

16.11 Transtypage

Comme exposé dans le chapitre 2, C est un langage typé statiquement, c'est-à-dire que le typage est effectué au moment de la compilation. Une erreur de typage est ainsi détectée avant l'exécution. Pour autant, c'est un langage dit *faiblement typé* car il est possible d'outrepasser le type d'une expression en imposant au compilateur le type de notre choix, sous certaines conditions toutefois. Dit autrement, cela revient à faire taire le compilateur sur ce qui serait parfois un avertissement.

Forcer le type d'une expression est appelé un *transtypage* (*cast* en anglais). Cela est réalisé en préfixant l'expression par le type désiré entre parenthèses :

(*type–expr*) *expr*

Ainsi, `(int*) (4 + 5)` revient à forcer le compilateur à considérer que l'entier 9 doit maintenant être considéré comme une adresse d'entier.

Important

Le transtypage ne doit jamais être utilisé pour faire disparaître magiquement un avertissement ou une erreur dont la raison n'a pas été comprise.

Le transtypage ne permet toutefois pas n'importe quelles conversions. Par exemple, il n'est pas possible de forcer une expression d'un type `struct` à être considérée comme un entier. Les transtypages légitimes sont :

- entre types arithmétiques (entiers, réels, `enum`),
- entre pointeurs et entiers,
- entre pointeurs.

Pour illustrer l'effet du transtypage considérons le programme suivant qui mémorise l'adresse de la variable `x` dans un pointeur `px` (ce qui est tout à fait légitime) mais également dans un entier long `fake_px`. Le choix d'un `long int` est volontaire car il doit avoir la même taille qu'un pointeur pour éviter un débordement.

```c
#include <stdio.h>

struct dummy_t { float foo ; int bar ; } ;

int main () {
  int x ;
  int *px = &x ;
  long int fake_px = &x ;
  printf ("    %p %lx\n", px, fake_px) ;
  printf ("+1: %p %lx\n", px + 1, fake_px + 1) ;
  return 0 ;
}
```

Lors de la compilation un avertissement est généré indiquant une conversion implicite du type pointeur vers le type entier. L'exécutable est toutefois créé puisqu'il ne s'agit que d'un avertissement.

```
$ gcc -Wall cast.c -o cast.x
cast.c: In function 'main':
cast.c:8:22: warning: initialization of 'long int' from 'int *' makes integer from pointer
 without a cast [-Wint-conversion]
    8 |   long int fake_px = &x ;
      |                      ^
```

Afin de faire taire cet avertissement, nous effectuons maintenant un transtypage de `&x` vers le type `long int`.

```c
#include <stdio.h>

int main () {
  int x ;
  int *px = &x ;
  long int fake_px = (long int) &x ;
  printf ("    %p %lx\n", px, fake_px)
      ;
  printf ("+1: %p %lx\n",
          px + 1, fake_px + 1) ;
  return 0 ;
}
```

```
$ gcc -Wall cast.c -o cast.x
$ ./cast.x
    0x7fff617cde5c 7fff617cde5c
+1: 0x7fff617cde60 7fff617cde5d
```

Lors de l'exécution, le programme affiche les valeurs initiales de `px` et de `fake_px`, puis celles après leur avoir rajouté 1. Notons que nous avons pris soin d'utiliser le format `%p` pour afficher `px` puisque c'est un pointeur et `%lx` pour `fake_px` puisque c'est un entier long que nous voulons afficher en hexadécimal pour le comparer facilement avec `px`.

Nous remarquons que `px` et `fake_px` contiennent bien la même valeur. Néanmoins, leur ajouter 1 ne provoque pas le même comportement. Pour `px` c'est une arithmétique de pointeur (cf. section 9.2.1), donc l'adresse est incrémentée de la taille d'un `int`, c'est-à-dire 4. Pour `fake_px`, puisque c'est un entier, lui rajouter 1 c'est juste faire $+1$.

16.11.1 Utilité du transtypage

Le transtypage est utile afin de rendre explicite et forcer la conversion entre types numériques. L'exemple de la division présenté en section 16.6.1.3 peut être généralisé avec des variables et non des constantes. Dans ce cas, le seul moyen de rendre explicite que les opérandes de la division doivent être considérées comme des réels est de leur appliquer un transtypage.

```c
#include <stdio.h>

void f (int x, int y) {
  float res = (float) x / y ;
  printf ("%f\n", res) ;
}
```

Important

La conversion entre type numériques peut effectuer un changement physique de représentation. Un des effets les plus visibles est que passer d'un `float` à un `int` provoque la troncature de la partie fractionnaire.

Le transtypage entre pointeurs peut également être utile pour implanter un mécanisme de généricité ou de polymorphisme au niveau des fonctions. C'est un moyen technique d'écrire des fonctions qui manipulent des arguments de types différents sans passer par des unions (cf. section 16.2).

Il peut également servir à signaler dans le code source une volonté de conversion pour laquelle nous sommes conscient des dangers. Par exemple, affecter un `int` avec une valeur provenant d'un `long int`. Il faut bien entendu être certain de ce que nous faisons et disposer des arguments démontrant que les conditions d'utilisation du transtypage ne provoqueront pas d'erreur de débordement.

Pour terminer, dans des contextes très particuliers, il permet de totalement (et sans filet) réinterpréter la représentation binaire d'une zone mémoire. Le programme suivant fait en sorte de récupérer le « premier » octet d'un champ de structure pour l'afficher.

```c
#include <stdio.h>

struct dummy_t { float foo ; int bar ; } ;

int main () {
  struct dummy_t dum = { 3.14, 42 } ;
  char *p = (char*) &dum.foo ;
  printf ("%x\n", *p & 0xFF) ;
  return 0 ;
}
```

Pour considérer notre `struct` comme un `char`, nous avons dû utiliser un subterfuge. Nous avons utilisé l'opérateur `&` pour obtenir une adresse de `struct dummy_t` que nous avons transtypée en `char*` : entre pointeurs, c'est légal. Finalement, pour récupérer la valeur comme un `char`, il a fallu déréférencer ce pointeur avec l'opérateur `*`. Bref, le passage par les adresses nous a permis de contourner sauvagement un transtypage interdit.

Nous avons choisi d'afficher le résultat en hexadécimal, donc en utilisant le format `%x`. Cependant, ce format est dédié à une valeur de type `int`, donc de 32 bits. Appeler `printf` avec `*p` provoque l'extension de la valeur de type `char` sur 32 bits (pour avoir effectivement une valeur de type `int`). Si le bit de poids fort de notre expression `*p` est à 1, alors il serait propagé dans les 24 bits de poids fort de la valeur `int` transmise à `printf`. C'est pourquoi nous utilisons le et bit-à-bit (cf. section 16.10.1) afin de ne conserver que les 8 bits (donc octet) de poids faible : les autres seront mis à 0.

De telles utilisations du transtypage sont très techniques et nécessitent une réelle connaissance de l'architecture sur laquelle le programme s'exécute ainsi que du compilateur utilisé. Les programmes obtenus sont possiblement hautement non portables d'une architecture à une autre.

Pour aller plus loin

Contrairement à ce que l'on peut encore voir dans des codes sources, on ne transtype jamais le pointeur retourné par `malloc` en C (pour C++, c'est une autre histoire). Autrement dit, c'est une mauvaise pratique d'écrire :

```c
int *p = (int*) malloc (n * sizeof (int)) ;
```

D'une part c'est inutile car `malloc` retourne un `void*` qui a la particularité de se convertir implicitement en tout autre pointeur (c'est même son unique rôle).

D'autre part, puisque ce n'est pas utile, autant ne pas alourdir le code source... inutilement.

Pour terminer, cela peut masquer une véritable erreur. Si vous oubliez d'inclure `<stdlib.h>`, alors le compilateur supposera que cette fonction retourne un `int`. Le transtypage fera alors taire un avertissement disant que `int` et pointeur ne sont pas *a priori* compatibles. Si le type `int` n'a pas la même taille que le type pointeur, alors la conversion provoquera une corruption de l'adresse retournée par `malloc`.

Mais... ce dernier argument devient de moins en moins applicable : les compilateurs modernes ont maintenant des cas particuliers pour certaines fonctions emblématiques de la bibliothèque standard. De surcroît, ils émettent toujours un avertissement en cas de fonction non déclaré, donc au pire, il reste quand même cet avertissement à lire.

16.12 Variable locale `static`

Une variable locale à une fonction peut être qualifiée `static` pour la rendre rémanente : elle conserve sa valeur entre les différents appels de la fonction qui l'héberge. Son initialisation, à une expression qui doit être une constante calculable au moment de la compilation, est uniquement réalisée lors du premier appel à sa fonction hôte.

En réalité, il s'agit une variable globale cachée, mais dont l'accès n'est possible que dans la fonction qui l'a définie. C'est donc une solution plus robuste qu'une variable globale explicite qui serait accessible par toutes les fonctions du programme. Le fait que ce soit une variable globale explique l'obligation qu'elle soit initialisée avec une valeur constante calculable au moment de la compilation.

```c
#include <stdio.h>

void f () {
  static int x = 0 ;
  printf ("%d\n", x) ;
  x++ ;
}

int main () {
  for (int i = 0; i < 3; i++) f ()
      ;
  return 0 ;
}
```

```
$ gcc -Wall static.c -o static.x
$ ./static.x
0
1
2
```

16.13 Variable globale `static`

Qualifier une variable globale par le mot-clé `static` a une toute autre signification. Cela signifie qu'elle ne sera accessible que par les fonctions du fichier source où elle est définie.

L'utilité de ce qualificatif s'inscrit donc forcément dans le cadre de programmes composés de plusieurs fichiers sources. Cela permet une meilleure modularité en évitant que des fonctions d'autres fichiers ne puissent accéder à cette variable.

16.14 Déclaration de variable (`extern`)

Dans les fichiers d'en-tête figurent principalement des déclarations (cf. section 15.1.3). Ces fichiers sont indispensables dans le cas où un programme est composé de plusieurs fichiers compilés séparément (et *a fortiori* lors de l'utilisation de bibliothèques logicielles).

La déclaration d'une fonction, appelée son *prototype*, se distingue syntaxiquement de sa définition : elle n'a pas de corps. Le prototype de la fonction :

```
int add (int x, int y) { return x + y ; }
```

est : `int add (int x, int y) ;`

Pour déclarer une variable, il n'est pas possible de donner juste son nom et son type comme dans :

```
int x ;
```

puisque c'est syntaxiquement une définition de variable sans initialisation. La déclaration d'une variable débute ainsi par le mot-clé `extern`, suivi de ce qui ressemble à une définition, bien entendu sans initialisation :

```
extern int x ;
```

indique qu'il existera, parmi les fichiers sources du programme, une variable nommée `x` et de type `int`. Cette déclaration n'alloue pas la variable : ce sera sa définition qui le fera.

16.15 Inclusion conditionnelle

Cette section revient sur des fonctionnalités du préprocesseur (cf. chapitre 15) qui seront particulièrement intéressantes dans le cadre de la compilation séparée de plusieurs fichiers sources (cf. chapitre 17).

Comme exposé dans la section 15.1.3, les fichiers d'en-tête servent à regrouper des déclarations, en particulier celles de fonctions et de types. Ce mécanisme permet de centraliser ces déclarations qui seront ensuite incluses dans tout fichier source en ayant besoin.

Les déclarations d'un `.h` peuvent nécessiter celles d'autres `.h`. Par exemple, nous pouvons imaginer un fichier `date.h` qui définit une structure permettant de représenter des dates.

```
struct date_t {
  int day ;
  int month ;
  int year ;
};
```

Nous faisons maintenant le choix de répartir les fonctions permettant de manipuler des dates en plusieurs fichiers sources, qui auront donc chacun leur `.h` (`print_d.h` et `valid_d.h`). Chacun doit donc inclure `"date.h"` pour connaître la définition du type `struct date_t`.

```
#include "date.h"

void print_date (struct date_t d)
     ;
```

```
#include <stdbool.h>
#include "date.h"

bool valid_date (struct date_t d) ;
```

Tout fichier source nécessitant à la fois d'appeler `print_date` et `valid_date` doit inclure à la fois `print_d.h` et `valid_d.h` pour connaître le prototype de ces deux fonctions. Par effet de bord, le fichier `date.h"` se retrouve inclus deux fois, la seconde inclusion masquant la première définition du type `struct date_t`, ce qui provoque plusieurs erreurs au moment de la compilation.

```
#include "print_d.h"
#include "valid_d.h"

int do_something () {
  struct date_t d ;
  ...
  print_date (d) ;
  ... valid_date (d) ...
  ...
}
```

```
$ gcc -c superprog.c
In file included from valid_d.h:2:0,
                 from superprog.c:2:
date.h:1:8: error: redefinition of 'struct date_t'
 struct date_t {
        ^~~~~~
In file included from print_d.h:1:0,
                 from superprog.c:1:
date.h:1:8: note: originally defined here
 struct date_t {
        ^~~~~~
superprog.c: In function 'do_something':
superprog.c:7:15: error: incompatible type for argument 1 of 'print_date'
   print_date (d) ;
               ^
In file included from superprog.c:1:0:
                                                                        →
```

```
print_d.h:3:6: note: expected 'struct date_t' but argument is of type 'struct date_t'
 void print_date (struct date_t d) ;
      ^~~~~~~~~~
```

Nous pouvons remarquer que certains messages peuvent sembler curieux, en particulier le dernier qui indique qu'un type `struct date_t` est attendu alors ... qu'un type `struct date_t` a été fourni. En fait, derrière ces deux noms se cachent des définitions différentes pour le compilateur (même si leur code est le même). Nous pouvons également remarquer qu'aucune erreur n'est levée pour l'appel à la fonction `valid_date` : la définition « de son » type `struct date_t` est bien visible car c'est celle qui a été apportée par le dernier `#include "date.h"`, celui provenant de `#include "valid_d.h"`.

Pour remédier à cette situation, il est nécessaire de rendre l'inclusion de fichiers d'en-tête idempotente : inclure plusieurs fois le même `.h` ne doit pas poser de problème. Techniquement, cela est réalisé en faisant en sorte que le contenu d'un `.h` ne puisse pas s'expanser plusieurs fois lors d'inclusions multiples de ce `.h`. On encadre le contenu d'un fichier d'en-tête par les directives idiomatiques suivantes :

```
#ifnfef MARQUEUR_UNIQUE
#define MARQUEUR_UNIQUE
  ...
  ...
#endif
```

où il est usuel de faire en sorte que le marqueur unique rappelle le nom du fichier `.h` (ce qui lui confère justement une grande chance d'être unique).

La directive conditionnelle `#ifndef` signifie « si non défini » et doit être terminée par une directive `#endif`. Elle agit comme une conditionnelle habituelle, mais elle est interprétée par le préprocesseur. Si le symbole suivant `#ifndef` n'a pas été défini par un `#define`, alors le préprocesseur inclut ce qui la suit jusqu'au `#endif` final. Dans le cas contraire, il l'ignore.

Dans le corps de cette directive conditionnelle, nous commençons par définir le symbole, qui devient alors connu, (notons qu'il n'y a pas besoin de lui donner une valeur particulière) et les définitions et déclarations du `.h` suivent, jusqu'au `#endif` final. Au prochain #include de ce fichier, le marqueur unique aura déjà été défini et rien ne sera textuellement inclus.

Nous pouvons donc réécrire notre fichier d'en-tête sous la forme suivante, où nous nommons le marqueur unique `__DATE_H__` en référence au fichier `date.h`.

```c
#ifndef __DATE_H__
#define __DATE_H__

struct date_t {
  int day ;
  int month ;
  int year ;
};

#endif
```

Pour aller plus loin

Le préprocesseur dispose d'autres directives conditionnelles, comme `#ifdef` (qui est l'opposé de `#ifndef`), `#else` et `#if` qui prend en argument une expression logique entre constantes évaluée par le préprocesseur.

Chapitre 17

Compilation séparée

Ce chapitre est en dehors du périmètre d'un cours de langage C. Il est applicable quel que soit le langage de programmation et décrit un outil utile au-delà des activités de programmation. Bien qu'il soit intégré dans ce document, ce chapitre concerne plus particulièrement le cours CSC_3GIN2_TA.

17.1 Motivations

Lors de l'écriture de programmes modestes, c'est-à-dire de quelques centaines de lignes, il est possible de rédiger le code source dans un unique fichier. Dans ce cas, la construction de l'exécutable est réalisée en appelant le compilateur `gcc` sur ce fichier source, avec les options recommandées déjà mentionnées dans les précédents chapitres.

Lorsque les programmes prennent de l'ampleur, il devient plus pratique de répartir leur code source dans différents fichiers, chacun regroupant les définitions traitant d'un même concept. Par exemple un fichier pour les traitements relatifs aux entrées utilisateur, un pour ceux relevant de la résolution proprement dite du problème et un pour la gestion de la mise en forme et la présentation des résultats.

Ce découpage permet également la réutilisation de code entre différents projets. Si nous avons mis ensemble toutes les définitions (types, fonctions) permettant la manipulation d'une structure de données complexe (liste, file, pile, graphe, etc.), nous pouvons directement récupérer ce code au travers de son fichier hôte pour l'intégrer dans un autre programme.

Pour compiler un programme composé de plusieurs fichiers, la solution la plus simple consiste à appeler le compilateur `gcc` avec tous les fichiers sources `.c` sur sa ligne de commande :

```
gcc -Wall list.c utils.c graph.c main.c -o awesome.x
```

Cette solution naïve présente de nombreux inconvénients. Les projets réalistes sont composés de nombreux fichiers sources (dizaines, centaines, voire milliers). Lors d'une modification d'un seul fichier, recompiler l'intégralité des fichiers est non seulement très long (des heures) mais en plus inutile : seul le fichier modifié devrait être recompilé puis l'exécutable final généré de nouveau.

La création d'un exécutable ou de tout autre objet construit à l'aide d'outils informatiques (comme ce document) peut nécessiter plusieurs étapes intermédiaires. Entrer à la suite au terminal les différentes lignes de commandes (dans le bon ordre de surcroît) est au bas mot fastidieux et assurément long.

Pour terminer, les lignes de commandes à invoquer peuvent être longues. Rien que celle nous avons recommandée pour compiler un programme C, avec toutes les options d'intérêt prioritaire est un peu longue à taper (43 caractères au minimum) :

```
gcc -Wall -Werror -Wfatal-errors blabla.c -o blabla.x
```

Alors certes, l'historique du terminal accessible avec les flèches ↑ et ↓ peut aider, mais gageons que les plus flegmatiques d'entre nous ne supporteront pas ce calvaire bien longtemps.

17.2 `make`

Il est donc nécessaire de disposer d'un outil qui permette de n'écrire les lignes de commandes à invoquer qu'une seule fois. Un *script shell* ferait l'affaire pour répondre à ce besoin (nous reviendrons d'ici peu sur l'option `-c` et les fichiers `.o`).

```
#! /bin/bash
gcc -Wall -Werror -Wfatal-error -c list.c
gcc -Wall -Werror -Wfatal-error -c utils.c
gcc -Wall -Werror -Wfatal-error -c graph.c
gcc -Wall -Werror -Wfatal-error -c main.c
gcc -Wall -Werror -Wfatal-error list.o utils.o graph.o main.o -o awesome.x
```

Pour autant, cette solution recompile toujours tous les fichiers sources. Il nous faut donc en plus ne reconstruire que ce qui est nécessaire suite à des modifications de fichiers. Pour y parvenir il est indispensable de pouvoir décrire les différentes étapes à invoquer, ce dont elles ont besoin en entrée, ce qu'elles produisent en sortie, mais surtout les *dépendances* entre les fichiers. Cette dernière notion permet de savoir ce qui doit être reconstruit en fonction de ce qui a changé.

L'outil `make` apporte la solution à tous ces besoins. Lors de son invocation au terminal, il recherche dans le répertoire courant et analyse un fichier nommé (par défaut) `Makefile` qui contient toutes les informations susdites.

17.3 Structure d'un `Makefile`

Un fichier `Makefile` définit des *règles* (*receipes* en anglais) qui contiennent des *cibles* à fabriquer à partir de *prérequis* en appliquant des *commandes*.

Au moyen de la date de dernière modification des fichiers, `make` vérifie quels fichiers *prérequis* ont été modifiés par rapport à chaque *cible* et ne reconstruit que ce qui dépend de ces changements. Si une cible a une date antérieure à l'un de ses prérequis, alors elle doit être reconstruite. Ce mécanisme est appliqué récursivement sur les prérequis de chaque cible, causant si besoin des reconstructions en cascade.

Avant d'entrer dans les détails, nous pouvons examiner l'exemple ci-dessous et décrire intuitivement ce qu'il signifie. Là règle nommée `all` indique qu'elle dépend des règles `test_stack.x` et `enonce.pdf`. Donc, « pour faire `all`, il faut d'abord faire `test_stack.x` et `enonce.pdf` ». La règle `test_stack.x` dépend des trois prérequis, `stack.h`, `stack.c` et `test_stack.c` qui sont vraisemblablement des fichiers sources. La commande à invoquer pour appliquer cette règle est l'invocation de `gcc`. Ensuite, `enonce.pdf` dépend de `pictures` (qui est elle-même la cible d'une autre règle) et de `enonce.tex` qui est vraisemblablement un fichier source LaTeX. Trois commandes doivent successivement être appelées pour réaliser la cible de cette règle.

```
.PHONY: all pictures clean
all: test_stack.x enonce.pdf

test_stack.x: stack.h stack.c test_stack.c
  gcc -Wall stack.c test_stack.c -o test_stack.x

enonce.pdf: pictures enonce.tex
  echo "" > flags.tex
  latex enonce.tex
  dvipdf enonce.dvi

pictures: auto1.fig
  fig2dev -L eps auto1.fig > auto1.eps

clean:
  rm -f a.out *.o *.x *~ .*~ flags.tex *.dvi *.pdf *.aux *.log *.eps
```

Important

Par défaut, l'invocation de la commande `make` cherche à reconstruire la cible de la première règle du fichier `Makefile`. Il est possible de demander explicitement de traiter une cible particulière en mettant son nom en argument lors de l'invocation de `make`.

```
make test_stack.x
```

17.4 Structure d'une règle simple

La forme générale d'une règle est la suivante :

```
cible : prérequis
  commande
  commande
  ...
```

La cible représente souvent le nom d'un fichier qui doit être généré par la règle. Ce n'est pas une obligation comme nous l'avons vu dans l'exemple précédent pour la règle `all` et nous reviendrons plus en détail sur le comportement de ces cibles dans la section 17.12. Qu'il y ait des prérequis ou non, le nom d'une cible doit être suivi de deux points (`:`).

Les prérequis, s'ils existent, suivent le nom de la cible après les deux points. Ces prérequis sont souvent des noms de fichiers préexistants mais ils peuvent également être des noms d'autres cibles de règles.

Dans le cas où un prérequis est un nom de fichier, `make` va regarder sa date de dernière modification pour savoir si la règle doit être invoquée pour reconstruire la cible.

Dans le cas où un prérequis est le nom d'une cible d'une autre règle, `make` va d'abord invoquer cette autre règle, qui à son tour considérera si elle doit reconstruire sa cible en fonction de ses propres prérequis. Et récursivement. . .

La ou les commandes à invoquer dans la règle sont n'importe quels exécutables présents sur la machine. En règle générale, l'exécution de ces commandes doit résulter en la création d'un fichier de même nom que celui de la cible. Et en règle générale, ces commandes prennent en arguments tout ou partie des fichiers de prérequis.

Important

Une ligne décrivant une commande doit impérativement débuter par une tabulation. En particulier, des espaces ne sont pas corrects. Les éditeurs de textes modernes reconnaissent qu'ils sont en train d'éditer un `Makefile` et font en sorte que la touche ⇤⇥ insère une véritable tabulation et non des espaces.

Par défaut, `make` affiche les commandes qu'il exécute. Si cet affichage est indésirable, il est possible de préfixer les commandes par le symbole `@` pour rendre l'exécution silencieuse. Dans ce cas, seuls les affichages provoqués par les commandes seront visibles.

17.5 Interlude : compiler manuellement plusieurs fichiers C

Avant de pouvoir illustrer l'intérêt de `make` et des `Makefile`, il faut s'attarder sur la manière dont `gcc` construit un exécutable à partir de plusieurs fichiers sources. Cette mécanique, totalement cachée lorsque nous compilons d'un coup plusieurs fichiers `.c` sur la ligne de commande, doit être explicitée pour en tirer parti dans les `Makefile`.

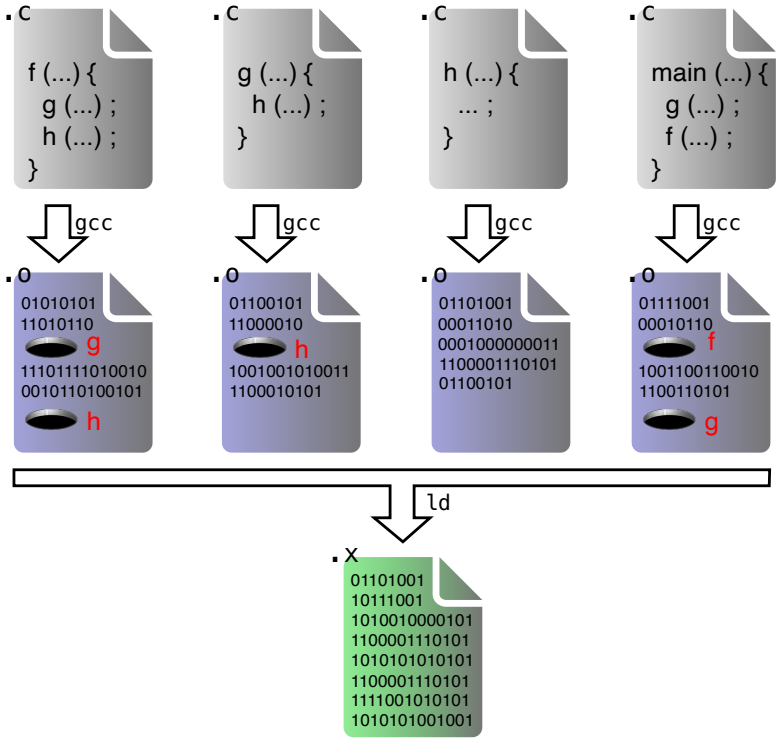
Pour chaque fichier source (i.e. `.c`), après l'étape de *preprocessing* réalisée par le préprocesseur (cf. chapitre 15), le compilateur génère un fichier dit *objet* d'extension `.o`. Ce fichier contient les instructions relatives aux définitions du fichier source `.c` compilé, mais avec des « trous ». Ces trous correspondent

aux variables globales et aux fonctions utilisées dans ce fichier, mais qui ne sont pas définies dans ce fichier.

Que ce soit la fonction `printf` que n'avons pas écrite (et qui appartient à la bibliothèque standard de C), des fonctions d'autres bibliothèques tierces ou bien des fonctions que nous avons écrites dans d'autres fichiers, au moment où `gcc` compile, il ne peut pas savoir où est leur code (ni même s'il existe déjà).

À l'issue de la compilation séparée de chaque fichier source, nous obtenons autant de fichiers objets. Une dernière étape nommée *édition de liens* est réalisée en regroupant tous les fichiers objets `.o` pour en faire l'exécutable final. Durant cette phase, les « trous » doivent être résolus et remplacés par les adresses des fonctions et variables précédemment inconnues.

L'éditeur de liens, nommé `ld`, est silencieusement appelé par `gcc` lorsque nous lui donnons des fichiers objets en arguments. Il est d'ailleurs très courant dans les `Makefile`, d'appeler `gcc` et non `ld` pour générer l'exécutable final.



Important

Pour compiler un fichier `.c` en fichier objet `.o`, il est nécessaire d'utiliser l'option `-c` de `gcc`. Par défaut, le nom du fichier créé a la même base que le fichier source, avec le suffixe `.o` à la place de `.c`. L'option `-o` n'est pas indispensable.

Compiler vers un fichier objet ne cherche donc pas à créer un exécutable. C'est la raison pour laquelle le compilateur ne se plaindra pas de ne pas trouver de fonction `main`. Avec la commande :

```
gcc -Wall -c foo.c
```

le fichier objet sera automatiquement nommé `foo.o`. L'option `-o` peut toutefois être utile dans le cas où nous souhaitons nommer le fichier objet résultat différemment.

Important

Une fois tous les fichiers objets obtenus, la phase finale consiste à invoquer `gcc` avec tous ces fichiers en argument (et d'éventuelles autres options pour gérer les bibliothèques tierces) afin de générer l'exécutable :

```
gcc foo.o bar.o gee.o buzz.o -o awesome.x
```

17.6 Exemple de `Makefile` simple pour du C

Maintenant que nous savons quelles sont les étapes nécessaires pour compiler des fichiers sources C, nous pouvons donner un premier exemple de `Makefile` fort simple.

Notre programme est constitué de deux fichiers sources `foo.c` et `bar.c`, chacun accompagné de son propre fichier d'en-tête `foo.h` et `bar.h` contenant diverses déclarations. Par hypothèse, `foo.c` fait un `#include "foo.h"`. Quant à `bar.c`, il fait en fait deux : `#include "foo.h"` et `#include "bar.h"`.

```
prog.x: foo.o bar.o
  gcc -o prog.x foo.o bar.o

foo.o: foo.c foo.h
  gcc -c foo.c

bar.o: bar.c bar.h foo.h
  gcc -c bar.c
```

Le fichier exécutable `prog.x` nécessite l'existence de `foo.o` et `bar.o`. Si l'un des deux est plus récent que `prog.x`, alors ce dernier est reconstruit avec la commande `gcc -o prog.x foo.o bar.o`.

Pour être généré, le fichier `foo.o` a besoin de `foo.c` et de `foo.h`. Pour être plus exact, il faudrait dire que « `foo.o` *dépend* de `foo.c` et de `foo.h` ». Même si `gcc` n'est pas appelé sur le fichier d'en-tête `foo.h`, il y a une dépendance car `foo.c` fait un `#include` de `foo.h`.

Pour finir, `bar.o` dépend de `bar.c` ainsi que des deux fichiers d'en-tête qu'il inclut.

```
$ ls
bar.c   bar.h   foo.c   foo.h   Makefile
$ make
gcc -c foo.c
gcc -c bar.c
gcc -o prog.x foo.o bar.o
```

17.7 Importance des dépendances

Nous dirons qu'un fichier a *dépend* d'un fichier b si toute modification de b implique une reconstruction de a. La notion de modification inclut bien entendu la création du fichier b. Si le fichier b n'existe pas, soit il existe une règle pour le créer et elle sera appelée d'abord, soit il n'en existe pas et une erreur sera levée comme dans l'exécution qui suit où le fichier `foo.c` n'existe pas.

```
$ ls
bar.c   bar.h   foo.h   Makefile
$ make
make: *** No rule to make target ‘foo.c’, needed by ‘foo.o’.  Stop.
```

Nous pouvons examiner l'effet des dépendances sur la reconstruction des cibles. Nous reprenons l'exemple de la section 17.6 précédente, avec les fichiers `.c` et `.h` et leur dépendances :

```
prog.x: foo.o bar.o
foo.o: foo.c foo.h
bar.o: bar.c bar.h foo.h
```

La commande `touch` utilisée dans le scénario suivant permet de changer la date de dernière modification d'un fichier en la positionnant à la date courante. Cela permet donc de simuler une édition du fichier.

```
$ touch bar.h
$ make
gcc -c bar.c
gcc -o prog.x foo.o bar.o
$ touch foo.h
$ make
gcc -c foo.c
gcc -c bar.c
gcc -o prog.x foo.o bar.o
```

Nous commençons par modifier `bar.h`. Puisque `bar.o` en dépend, ce dernier est reconstruit. Mais `prog.x` dépendant de `bar.o`, il est également reconstruit à la suite. Par contre, `foo.o` n'est pas touché puisqu'il ne dépend pas de `bar.h`.

Si maintenant nous modifions `foo.h`, alors `foo.o` et `bar.o` sont reconstruits car ils dépendent tous les deux de ce fichier. Bien entendu, `prog.x` est ensuite reconstruit à cause des changements de `foo.o` et de `bar.o`.

Nous avons donc bien une reconstruction minimale des fichiers en fonction des modifications et non une re-compilation aveugle de tout le projet. Le gain de temps est ainsi appréciable. Mais plus important encore, `make` n'oublie pas de reconstruire de fichiers ... tant que nous avons bien décrit les dépendances entre les fichiers, comme nous allons l'illustrer dans quelques instants.

Pour aller plus loin

Au travers des dépendances, un fichier `Makefile` décrit un graphe, justement appelé *graphe de dépendances*. L'outil `make` met en œuvre un algorithme de *tri topologique* qui permet d'obtenir un ordre total sur les sommets du graphe que sont les cibles et prérequis.
Puisqu'il ne peut pas y avoir de cycles (un fichier ne peut pas dépendre de lui-même par transitivité), ce type de graphe est plus particulièrement un graphe acyclique orienté (DAG).

17.8 Incohérences de dépendances

Un fichier objet `.o` peut dépendre principalement de son .c associé, des fichiers `.h` que le `.c` « **#include** » et des éventuels fichiers `.h` que ces mêmes `.h` « **#include** ». Un fichier exécutable dépend principalement de ses `.o`.

Considérons les deux fichiers `f.c`, `main.c` ainsi que le fichier d'en-tête `f.h` suivants. À la lecture de ces fichiers sources, il apparaît que `f.c` et `main.c` dépendent tout deux du fichier `f.h` (ils le « `#include`-nt »).

```c
#include <stdio.h>
#include "f.h"

void f () {
  printf
    ("f: %d\n", NUM)
      ;
}
```

```c
#include <stdio.h>
#include "f.h"

int main () {
  f () ;
  printf
    ("main: %d\n", NUM) ;
  return 0 ;
}
```

```c
#define NUM
    (1234)
void f (void) ;
```

Délibérément, nous écrivons le `Makefile` suivant, dans lequel nous omettons la dépendance de `main.c` sur `f.h` et voyons les conséquences de cet oubli.

```
prog.x: f.o main.o
  gcc -o prog.x f.o main.o

f.o: f.c f.h
  gcc -c f.c

main.o: main.c      # Dépendance sur f.h oubliée ici !
  gcc -c main.c
```

Nous compilons le projet une première fois puis lançons l'exécutable obtenu. Le résultat n'a rien d'inattendu.

```
$ make
gcc -c f.c
gcc -c main.c
gcc -o prog.x f.o main.o
$ ./prog.x
f: 1234
main: 1234
```

Maintenant, modifions le fichier `f.h` pour remplacer la valeur de `NUM` par `4321`. Ce fichier contient désormais la première ligne `#define NUM (4321)`. Nous pouvons rappeler `make` afin qu'il reconstruise l'exécutable que nous lançons juste après.

```
$ make
gcc -c f.c
gcc -o prog.x f.o main.o
$ ./prog.x
f: 4321
main: 1234
```

Nous constatons tout d'abord que seuls `f.o` et `prog.x` ont été reconstruits. C'est normal puisque `main.o` est prétendument (et à tort) indépendant de `f.h`. Ensuite, à l'exécution, nous remarquons que les deux affichages divergent. C'est encore normal puisque `main.o` est resté inchangé, avec l'ancienne valeur de la macro `NUM`, alors que `f.o` a bien intégré sa modification.

Important

L'oubli de dépendances dans un `Makefile` peut conduire à des re-compilations partielles, donc incorrectes. L'exécutable généré peut alors contenir des portions de code ayant fait des hypothèses différentes sur des mêmes symboles.

17.9 Variables

Dans un `Makefile`, certaines commandes, certaines options (et plus généralement certaines chaînes de caractères) peuvent apparaître à de multiples endroits. Dans l'exemple ci-dessous, une version spécifique de `gcc` est utilisée en donnant son chemin d'accès, l'option `-I` permet d'étendre le chemin de recherche par défaut des fichier d'en-tête, `-O3` demande un niveau d'optimisations agressif et la macro `DEBUG` est définie au vol. Lors de la phase finale d'édition de liens, l'option `-L` permet d'étendre le chemin de recherche par défaut des bibliothèques et `-larch` demande d'intégrer la bibliothèque nommée `libarch.a`.

```
all: exec.x

stack.o: stack.c stack.h
```

```
  /opt/local/bin/gcc-14 -Wall -I/opt/local/share/extlib -O3 -DDEBUG stack.c -c

inet.o: inet.c stack.h
  /opt/local/bin/gcc-14 -Wall -I/opt/local/share/extlib -O3 -DDEBUG inet.c -c

exec.o: exec.c arch.h
  /opt/local/bin/gcc-14 -Wall -I/opt/local/share/extlib -O3 -DDEBUG exec.c -c

exec.x: stack.o inet.o exec.o
  /opt/local/bin/gcc-14 -L/opt/local/lib -larch stack.o inet.o exec.o -o exec.x
```

D'une part, la répétition de toutes les options à chaque ligne de compilation est fastidieuse. D'autre part, si nous souhaitons changer la valeur d'une option ou en rajouter une (un chemin de `-I`, le niveau d'optimisation de `-O`, etc.), nous devrons le faire à de multiples endroits.

Il est possible de définir des variables qui se comportent de manière similaire a un **#define** du pré-processeur. La définition est alors centralisée à un unique endroit et toutes les occurrences d'utilisation de la variable font référence à cette définition. Une variable est définie par :

$$nom_variable = chaine$$

et utiliser la valeur d'une variable se fait en mettant son nom entre parenthèse, le tout préfixé par un signe $:

$$\$(nom_variable)$$

Il est ainsi possible de réécrire notre précédent `Makefile` de manière plus concise et surtout de rendre la modification de son contenu après-coup nettement plus aisée.

```
BINDIR=/opt/local/bin/
GCC=$(BINDIR)gcc-14
OPTLEVEL=3
COMPFLAGS=-Wall -I/opt/local/share/extlib -O$(OPTLEVEL) -DDEBUG
LDFLAGS=-L/opt/local/lib -larch

all: exec.x

stack.o: stack.c stack.h
  $(GCC) $(COMPFLAGS) stack.c -c

inet.o: inet.c stack.h
  $(GCC) $(COMPFLAGS) inet.c -c

exec.o: exec.c arch.h
  $(GCC) $(COMPFLAGS) exec.c -c

exec.x: stack.o inet.o exec.o
  $(GCC) $(LDFLAGS) stack.o inet.o exec.o -o exec.x
```

17.10 Variables automatiques

Un certain nombre de variables prédéfinies existent pour faire référence à certaines parties d'une règle. Les plus emblématiques de ces variables dites *automatiques* sont :

- `$@` : nom de la cible de la règle courante.
- `$<` : nom du premier prérequis de la règle courante.
- `$^` : noms de tous les prérequis de la règle courante.

La règle d'édition de liens finale de notre précédent `Makefile` :

```
exec.x: stack.o inet.o exec.o
  $(GCC) $(LDFLAGS) stack.o inet.o exec.o -o exec.x
```

peut ainsi être réécrite de manière plus concise, sans répétition des prérequis ni de la cible dans la commande de compilation, sous la forme :

```
exec.x: stack.o inet.o exec.o
  $(GCC) $(LDFLAGS) $^ -o $@
```

Pour plus d'informations à propos des variables automatiques, il est conseillé de se référer à la documentation en ligne de GNU `make` : https://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html.

17.11 Métacaractères (*wilcards*) dans les règles

Il est possible d'utiliser des *métacaractères* (*wilcards* en anglais) dans les règles afin de dénoter un ensemble de noms de fichiers. Les métacaractères permettent une forme réduite d'expressions rationnelles et permettent de représenter des noms de fichiers ayant une certaine forme en commun. Ces métacaractères sont les mêmes que ceux utilisés par le Bourne Shell :

- `*` : n'importe quel caractère, n'importe quel nombre de fois.
- `?` : n'importe quel caractère, une seule fois.
- `[c1,c2,c3]` : les caractères c_1, c_2 ou c_3, une seule fois.
- `[c1-c2]` : un caractère entre c_1 et c_2, une seule fois.

L'utilisation de ces séquences peut se faire aussi bien dans les cibles, que dans les prérequis ou les commandes. Néanmoins, elles ne fonctionnent pas dans les définitions de variables où il faut utiliser la forme `$(wildcard` *pattern*`)` à la place (pour plus de détails, se référer à la documentation en ligne sur https://www.gnu.org/software/make/manual/html_node/Wildcard-Function.html).

```
$ ls
bar.c   bar.o   foo.h   hoo.h   prog.x
bar.h   foo.c   foo.o   Makefile
$ more Makefile
target: foo.o bar.o *.c
        @echo Prequisites are $^     # Wildcards expansées par make
        @ls *.h                      # Wildcards expandées par le shell.
$ make
Prequisites are foo.o bar.o bar.c foo.c
bar.h foo.h hoo.h
```

L'expansion des métacaractères présents dans les prérequis est réalisée par `make`. Par contre, ceux présents dans les commandes sont expansés par le shell qui exécute la commande. Notons au passage l'utilisation du @ préfixant les commandes pour éviter que `make` ne les affiche avant leur exécution (cf. section 17.4).

Ainsi, dans l'exemple ci-dessus, le `*.c` présent dans les prérequis est bien transformé en `bar.c foo.c` avant que la commande `echo` ne soit transmise au shell. Dit autrement, la commande envoyée au shell est `echo bar.c foo.c`.

De manière totalement différente, le `*.h` présent en argument de la commande `ls` n'est pas expansé par `make`, il le sera par le shell juste avant d'exécuter la commande `ls`. Dit autrement, la commande envoyée au shell est `ls *.h`.

17.12 Cibles PHONY

À l'exception de rares cas qui ont pu passer inaperçus dans les précédents exemples, une cible correspond généralement à un nom de fichier. C'est même grâce à cela que `make` peut savoir (grâce aux dates des fichiers) ce qui doit être reconstruit.

Cependant, certaines règles peuvent ne pas générer de fichier lors de leur invocation. L'exemple le plus emblématique (qui a d'ailleurs été silencieusement montré dans le premier exemple de `Makefile`

en section 17.3) est la cible usuellement nommée `clean`, qui a pour but de faire le ménage en supprimant tous les fichiers construits lors de l'invocation de `make`.

```
clean:
  rm -f a.out *.o *.x *~ .*~ flags.tex *.dvi *.pdf *.aux *.log *.eps
```

Cette cible et sa règle permettent de supprimer les fichiers objets, les fichiers temporaires, les exécutables, etc. Fort logiquement, une telle règle n'a pas besoin de prérequis : elle doit faire le ménage à chaque fois que nous l'appelons, indépendamment de s'il y a effectivement des fichiers à supprimer.

```
dosomething:
  touch foo.o bar.o a.out junk.aux res.log log.log

clean:
  rm -f a.out *.o *.x *~ .*~ flags.tex *.dvi *.pdf *.aux *.log *.eps
```

```
$ make dosomething
touch foo.o bar.o a.out junk.aux res.log log.log
$ ls
a.out   bar.o   foo.o     Makefile
bar.c   foo.c   junk.aux  res.log
bar.h   foo.h   log.log
$ make clean
rm -f a.out *.o *.x *~ .*~ flags.tex *.dvi *.pdf *.aux *.log *.eps
$ ls
bar.c   bar.h   foo.c   foo.h   Makefile
```

Puisque le répertoire ne contient pas de fichier nommé `clean`, fort logiquement `make` considère qu'il faut exécuter la règle. Si maintenant notre répertoire contient un fichier nommé `clean`, lors de l'invocation `make clean` la règle va être analysée pour voir si ses prérequis sont plus récents que notre fichier `clean`. Puisqu'il n'y a pas de prérequis à la règle, ils ne sont pas plus récents et `make` considère que la règle n'a pas besoin d'être exécutée.

```
$ make dosomething
touch foo.o bar.o a.out junk.aux res.log log.log
$ ls
a.out   bar.o   foo.o     Makefile
bar.c   foo.c   junk.aux  res.log
bar.h   foo.h   log.log
$ touch clean
$ ls
a.out   bar.o   foo.h     log.log
bar.c   clean   foo.o     Makefile
bar.h   foo.c   junk.aux  res.log
$ make clean
make: 'clean' is up to date.
$ ls
a.out   bar.o   foo.h     log.log
bar.c   clean   foo.o     Makefile
bar.h   foo.c   junk.aux  res.log
```

Afin de forcer une règle à être exécutée de manière inconditionnelle, il faut avoir l'avoir préalablement déclarée comme `.PHONY`. Dans l'exemple ci-dessous, deux telles règles existent : la fameuse `clean` ainsi que `distrib` dont on imagine que l'effet est d'aller copier un fichier PDF (préalablement construit) quelque part sur un site Web distant (par la commande `scp`).

```
.PHONY: clean distrib

book.pdf: main.tex content.tex
  pdflatex main.tex -job-name=$@

clean:
  rm -f *.log *.aux *.out *.nav *.snm *.toc book.pdf

distrib: book.pdf
  scp book.pdf 147.xxx.yyy.zzz:/home/site/PDFs/
```

17.13 Substitution dans les variables

Dans la section 17.9 nous avons vu comment déclarer des variables. Nous pouvons donc définir une variable qui représente la liste des fichiers sources de notre programme :

```
SRCS = a.c b.c c.c d.c e.c
```

puis une variable qui représente la liste des fichiers objets correspondants à ces fichiers sources :

```
OBJS = a.o b.o c.o d.o e.o
```

où nous remarquons que `OBJS` est la répétition de `SRCS` en changeant uniquement les suffixes `.c` en `.o`.

Il est possible d'automatiser la substitution de suffixes dans la valeur d'une variable :

```
$(VAR:.aa=.bb)
```

évalue la valeur de la variable `VAR` et remplace le `aa` final de chaque mot par un `bb` final.

Nous pouvons donc définir `OBJS` à partir de `SRCS` de manière très concise par :

```
OBJS = $(SRCS:.c=.o)
```

où toutes les occurrences de `.c` finaux dans la chaîne `a.c b.c c.c d.c e.c` sont remplacées par des `.o` finaux, donnant la chaîne résultat `a.o b.o c.o d.o e.o`.

```
SRCS = a.c b.c c.c d.c e.c f.c h.c dir/i.c dir/j.c dir/i.c dir/l.c m.c n.c
OBJS = $(SRCS:.c=.o)

dummy: $(SRC)
        @echo "Please compile $(SRCS) into $(OBJS)"
```

```
$ make
Please compile a.c b.c c.c d.c e.c f.c h.c dir/i.c dir/j.c dir/i.c dir/l.c m.c n.c into a.o
b.o c.o d.o e.o f.o h.o dir/i.o dir/j.o dir/i.o dir/l.o m.o n.o
```

L'exemple de `Makefile` ci-dessus est certes limité car il ne provoque pas réellement la compilation de tous les `.c` en `.o`. Si nous voulions parvenir à ce but, il nous faudrait écrire les 13 règles permettant de passer de chaque `.c` au `.o` en invoquant `gcc -c`. La section suivante (17.14) va apporter une solution élégante pour s'épargner cette peine. . .

17.14 Règles implicites (*pattern rules*)

L'exemple de la section précédente (17.13) ne nous a pas permis de tirer pleinement parti de la substitution dans les variables pour automatiser la compilation de plusieurs fichiers sources. Dans son `Makefile` manquaient encore les véritables règles permettant de construire chaque `.o` à partir de son `.c`.

Les *règles automatiques* permettent de décrire comment construire une cible à partir de prérequis selon la forme de leurs noms (et non selon leurs noms exacts). Ce sont donc des règles génériques dans lesquelles le métacaractère % sert à signifier « n'importe quelle chaîne non vide ». Ainsi,

```
%.o : %.c
  gcc -c $(CFLAGS) $< -o $@
```

définit une règle disant qu'une cible dont le nom se termine par `.o` s'obtient à partir d'un prérequis de même racine mais se terminant par `.c` en appliquant la commande `gcc` avec les options spécifiées. Puisque dans cette règle les noms de la cible et des prérequis ne sont pas explicitement donnés, l'utilisation des variables automatiques `$<` et `$@` prend un intérêt capital. Ainsi, le texte de la commande reste indépendant des noms effectifs des cibles et prérequis.

Nous pouvons donc écrire des `Makefile` assez concis et génériques où seules quelques définitions de variables sont à changer lorsque nous rajoutons des fichiers sources ou modifions des options de compilation. Dans une certaine mesure, un tel `Makefile` peut être réutilisé d'un projet (simple) à un autre avec relativement peu de modifications.

```
BINDIR=/usr/bin/
GCC=$(BINDIR)gcc
OPTLEVEL=2
CFLAGS=-Wall -I/opt/local/share/extlib -O$(OPTLEVEL) -DDEBUG
LDFLAGS=-L/opt/local/lib -lpcre

SRCS = stack.c inet.c exec.c
OBJS = $(SRCS:.c=.o)
EXE = exec.x

.PHONY: all clean

all: $(EXE)

%.o: %.c
 gcc -c $(CFLAGS) $< -o $@

$(EXE): $(OBJS)
 $(GCC) $(LDFLAGS) $^ -o $@

clean:
 rm -f a.out $(EXE) $(OBJS) *~ .*~
```

Ce `Makefile` est très élégant, mais peut-être un point vous interpelle-t-il : il n'y a aucun fichier d'en-tête `.h` qui soit mentionné. Or, en section 17.7, nous avons vu l'importance capitale de ne pas oublier de dépendances entre `.c` et `.h`. Dans un projet réaliste, il y a toujours des fichiers d'en-tête.

Important

L'utilisation de règles implicites ne doit pas faire oublier les dépendances entre `.c` et `.h`. Ces dépendances doivent être rajoutées aux règles implicites.

Les règles implicites ne sont pas suffisantes pour prendre en compte les dépendances particulières de chaque fichier `.c` sur ses `.h` d'intérêt. Néanmoins, elles permettent de les rajouter de manière très concise sous la simple forme de cible et prérequis. La commande de compilation sera reprise de la règle implicite.

Supposons que nous ayons les fichiers d'entête `header1.h`, `header2.h` et `header3.h`. Considérons que `stack.c` inclut `header1.h`, que `inet.c` inclut `header1.h` et `header2.h`, que `exec.c` inclut `header2.h` et `header3.h`. Il nous suffit d'ajouter explicitement ces dépendances dans le `Makefile`.

```
BINDIR=/usr/bin/
GCC=$(BINDIR)gcc
                                                                    →
```

```
OPTLEVEL=2
CFLAGS=-Wall -I/opt/local/share/extlib -O$(OPTLEVEL) -DDEBUG
LDFLAGS=-L/opt/local/lib -lpcre

SRCS = stack.c inet.c exec.c
OBJS = $(SRCS:.c=.o)
EXE = exec.x

.PHONY: all clean

all: $(EXE)

%.o: %.c
 gcc -c $(CFLAGS) $< -o $@

stack.o: stack.c header1.h
inet.o: inet.c header1.h header2.h
exec.o: exec.c header2.h header3.h

$(EXE): $(OBJS)
 $(GCC) $(LDFLAGS) $^ -o $@

clean:
 rm -f a.out $(EXE) $(OBJS) *~ .*~
```

La bonne nouvelle est que `gcc` dispose de l'option `-MM` qui permet de générer les dépendances directement au format attendu dans un `Makefile`. Il suffit alors de les copier-coller (ou de les inclure avec le directive `include` de `make`) dans le `Makefile`.

```
$ more f1.c
#include "f1.h"
#include "f2.h"
$ more f2.c
#include "f3.h"
$ more f3.c
#include "f2.h"
#include "f1.h"
$ more f2.h
#include "f4.h"
$ gcc -MM *.c
f1.o: f1.c f1.h f2.h f4.h
f2.o: f2.c f3.h
f3.o: f3.c f2.h f4.h f1.h
```

17.15 Appel à `make` et (quelques) options

Le fichier analysé par `make` s'appelle par défaut `Makefile`. L'option `-f` permet de spécifier un autre nom de fichier.

```
make -f otherMakefile
```

Il est possible de simuler les commandes sans les exécuter avec l'option `-n`. Cela revient à vérifier ce que ferait `make` avec l'état actuel des cibles et des prérequis.

Un projet peut contenir différents répertoires. Dans une règle, pour invoquer `make` récursivement dans les répertoires, deux solutions sont possibles. Souvent l'appel récursif dans une règle est fait avec le même nom de cible (`clean` dans l'exemple suivant), mais ce n'est en rien une obligation.

- L'option `-C` suivie du nom de répertoire :

```
clean:
```

```
    rm -f *.o
    make -C docs/ clean
```

- La descente explicite dans le répertoire avec `cd` puis l'appel à `make` :

```
clean:
    rm -f *.o
    cd docs ; make clean
```

Il est possible de définir ou redéfinir au vol les variables d'un `Makefile` à la place de changer leurs définitions en éditant le fichier. Pour redéfinir (ou définir) une variable lors de l'invocation de `make`, il faut passer en argument le nom de la variable, suivi du signe =, suivie de la chaîne représentant la valeur de cette variable. Aucun espace ne doit être présent autour du signe =. Généralement, la chaîne représentant la valeur de la variable est placée entre apostrophes (ou guillemets en fonction de l'effet voulu) afin d'autoriser les espaces et autres caractères qui seraient interprétés par le shell.

```
GCC = gcc
SRCS = foo.c
EXE = foo.x

$(EXE): $(SRCS)
 $(GCC) $(CFLAGS) $^ -o $@
```

```
$ make
gcc  foo.c -o foo.x
$ make CFLAGS='-Wall'
gcc -Wall foo.c -o foo.x
$ make SRCS='bla.c'
make: *** No rule to make target `bla.c', needed by `foo.x'.  Stop.
$ make GCC='echo'
echo  foo.c -o foo.x
foo.c -o foo.x
```

Chapitre 18

Annexes

18.1 Mots-clés (C ANSI)

Sont donnés ici les mots-clés de la norme ANSI du langage C. D'autres mots-clés sont apparus dans les normes suivantes du langage mais ils ne concernent pas les constructions de base du langage et sont, à ce titre, ignorés dans ce document.

Quelques-uns de ces mots-clés font référence à des traits ou des subtilités non abordées : ils sont donnés à titre d'exhaustivité.

auto	break	case	char
const	continue	default	do
double	else	enum	extern
float	for	goto	if
int	long	register	return
short	signed	sizeof	static
struct	switch	typedef	union
unsigned	void	volatile	while

18.2 Priorité, associativité

Les opérateurs sont donnés par ordre de précédence croissante. Lorsque plusieurs opérateurs figurent sur une même ligne, leur précédence relative augmente de gauche à droite (sens de lecture). La précédence représente la priorité entre les opérateurs. Plus sa précédence est faible, plus un opérateur est prioritaire (il sera évalué en premier).

L'associativité représente le parenthésage entre plusieurs occurrences d'un même opérateur binaire. Un opérateur $\odot$ est associatif gauche-droite si l'expression $e_1 \odot e_2 \odot e_3$ représente $(e_1 \odot e_2) \odot e_3$.

Précédence	Opérateur	Description	Associativité

1	`++ --`
	`()`
	`[]`
	`.`
	`->`
		post in/décrémentation
		appel de fonction
		indexation de tableau
		accès membre `struct` et `union`
		accès membre `struct` et `union` par pointeur
			gauche-droite

2	`++ --`
	`+ -`
	`! ~`
	`(type)`
	`*`
	`&`
	`sizeof`
		pré in/décrémentation
		modificateurs unaires de signe
		négation logique, bit-à-bit
		transtypage
		déréférencement
		extraction d'adresse
		taille de type en octets
			droite-gauche

3	`* / %`	multiplication, division, reste division entière	gauche-droite
4	`+ -`	addition, soustraction	gauche-droite
5	`« »`	décalage à gauche, à droite	gauche-droite
6	`< <=` `> >=`	test infériorité test supériorité	gauche-droite
7	`== !=`	égalité, différence	gauche-droite
8	`&`	et bit-à-bit	gauche-droite
9	`^`	ou exclusif bit-à-bit	gauche-droite
10	`|`	ou bit-à-bit	gauche-droite
11	`&&`	et logique	gauche-droite
12	`||`	ou logique	gauche-droite

13	`?:`
	`=`
	`+= -=`
	`*= /= %=`
	`«= »=`
	`&= ^= |=`
		conditionnelle ternaire
		affectation
		affectation par addition, soustraction
		affectation par multiplication, division, reste
		affectation par décalage à gauche, à droite
		affectation par et, ou exclusif, ou bit-à-bit
			droite-gauche

14	`,`	séquence d'expressions	gauche-droite

Index

-Wall, 42
#define, 160
#endif, 160
#ifndef, 160
!=, 33
!, 34
(), 156
*=, 36
*
    arithmétique, 32
    déréférencement, 60
    type pointeur, 58
++, 36
+=, 36
+, 32
,, 150
-, 36
-=, 36
->, 116
-I, 122
-MM, 175
-Wall, 12, 23–25, 35
-Werror, 12
-Wfatal-errors, 12
-c, 24, 166
-o, 11
-, 32
/=, 36
/, 32
«, 156
<=, 33
<, 33
==, 33
=, 35
>=, 33
», 156
>, 33
?, 149
@ (commande), 165
#define, 124
#include, 14, 121
$<, 170
$@, 170
$^, 170
%, 32
&&, 34
&
    bit-à-bit, 155
    pointeur, 59
^D, 110
^, 155
~, 155
\, 20
    0, 20, 79, 83
    n, 20, 53
{ }, 37
stdlib.h, 57
||, 34
|, 155

accolades, 37
addition, 32
adresse, 57
    extraction, 59
    passage d'argument, 69, 70, 106
affectation, 35
    abrégée, 36
    forcée, 36
affichage, 53
allocation, 98
arithmétique pointeur, 75
associativité, 178

backslash, 20
bibliothèque standard, 14
bit-à-bit, 155
bloc, 37
bool, 29
booléen, 20, 29
boucle
    do while, 47
    for, 45
    while, 45
break, 44, 48
bus error, 59, 100

caractère, 19
    chaîne, 79
    échappement, 20
case, 43
cast, 156
champ, 113
    ->, 116
    ., 115
char, 19
    *, 79–81
    [], 79
chaîne, 79
    comparaison, 85
    concaténation, 86
    copie, 83
    longueur, 83
    non modifiable, 80
    sous-chaîne, 87
cible, 164
code retour (main), 14
commentaire, 13
compilateur, 9, 11
compilation, 9, 11
    séparée, 163
conditionnelle
    expression, 149
    instruction, 41
    ternaire, 149
const, 136
constante, 31
    symbolique, 124
continue, 49
copie
    passage d'argument, 70
cpp, 121, 160

division, 32
    entière, 145
do while, 47
double, 19
décalage, 156
définition
    fonction, 23
    type, 28, 113
    variable, 24
dépendance, 164, 167

éditeur de liens, 165
égalité
    flottant, 148
en-tête, 121, 160
entier, 18, 143
    division, 145
    débordement, 143
    signe, 18, 144
    taille, 18, 143
entrée standard, 105
enum, 28
EOF, 110
et
    bit-à-bit, 155
    logique, 34
expression, 31
    arithmétique, 32
    logique, 34
    relationnelle, 33
    séquence, 150
extern, 160

false, 29
faux, 20
fichier
    en-tête, 14, 160
    objet, 165
float, 19
flottant, 19
    égalité, 148

fonction, 23
  appel, 34
  prototype, 23
for, 45
format, 54
free, 100

gcc, 11
goto, 152

heap, 98
hexadécimal, 155

inclusion conditionnelle, 160
incrémentation (pré/post), 36
indentation, 12
indexation, 74
inférieur à, 33
int, 18, 143
INT_MAX, 95
INT_MIN, 95

ld, 166
libération, 100
ligne de commande, 89
limits.h, 95
linker, 165
long, 18
long long, 18

macro, 124
  constante, 125
  paramétrée, 126
make, 164
Makefile, 164
malloc, 98
man, 14
manuel, 14
matrice, 76
modulo, 32
mot-clé, 177
multiplication, 32
mémoire
  allocation, 98
  débordement, 108
  libération, 100
métacaractère, 171

non
  bit-à-bit, 155
  logique, 34
notation
  fléchée, 116, 117
  hexadécimale, 94
  pointée, 115
NULL, 57, 98

opérateur
  arithmétique, 32
  logique, 34
  relationnel, 33
ou
  bit-à-bit, 155
  exclusif bit-à-bit, 155
  logique, 34

paresse, 34
pile, 65, 97, 141
poids (bit), 143
pointeur, 58
  arithmétique, 75
  déclaration, 58
  déréférencement, 60
  initialisation, 59
  nul, 57
printf, 53
priorité, 178
prototype, 23, 122, 160
précision, 19, 146
précédence, 178
préprocesseur, 121, 160
prérequis, 164

redirection, 105
reste division entière, 32
return, 23, 36
règle, 164
  PHONY, 171
  implicite, 173
réel, 19

segmentation fault, 59, 100, 109
short, 18
signed, 18
sizeof, 99
soustraction, 32
stack, 65
static
  globale, 160
  locale, 159
stdbool.h, 29
stdio.h, 14, 53, 106
stdlib.h, 91, 92, 98, 100
strcat, 86
strcmp, 85
strcpy, 83
strlen, 83
strstr, 87
struct, 113
structure, 113
supérieur à, 33

tableau
  accès, 74
  statique, 73
    définition, 73
    plus. dimensions, 76
tas, 98
transtypage, 156
true, 29
type
  de base, 17
  entier, 18
  énuméré, 28
typedef, 131

union, 133
  initialisation, 135
unsigned, 18

variable, 24
  Makefile, 169
    automatique, 170
  initialisation, 25
  locale, globale, 26
  locale,globale, 38
  substitution, 173
VLA, 141
void, 24, 34, 36
  *, 98, 159
volatile, 140
vrai, 20

while, 45

Depuis 1741, nous formons des femmes et des hommes, audacieux et créatifs, prêts à explorer les possibles et anticiper l'inattendu.

Par la recherche et l'innovation, nous faisons émerger les solutions nouvelles, performantes et responsables, pour le monde de demain.

À la croisée de multiples disciplines de l'ingénierie, augmentées par le numérique, notre excellence scientifique et technique éclaire et façonne les transformations des grands domaines de souveraineté.

Nous sommes l'école des grandes avancées.

ENSTA | INSTITUT POLYTECHNIQUE DE PARIS

Campus de Paris-Saclay (siège)

828, boulevard des Maréchaux
91762 Palaiseau Cedex - France

Campus de Brest
2 rue François Verny
29806 Brest Cedex 9

www.ensta.fr

1re édition

2026-2027 v1.1.1