

Le langage C

Auteurs :
Lucas-84
paraze
Taurre
informaticienzero

Catégorie :
Programmation et algorithmique
Temps de lecture estimé : 1 jour et 6 heures

Licence CC 0

Le langage C

**Auteurs :
Lucas-84
paraze
Taurre
informaticienzero**

**Catégorie :
Programmation et algorithmique
Temps de lecture estimé : 1 jour et 6 heures**

Licence CC 0

Avant-propos

Vous souhaitez apprendre à programmer, mais vous ne savez pas comment vous y prendre? Vous connaissez déjà le C, mais vous avez besoin de revoir un certain nombre de points? Ou encore, vous êtes curieux de découvrir un nouveau langage de programmation? Si oui, alors permettez-nous de vous souhaiter la bienvenue dans ce cours de programmation consacré au langage C.

Pour pouvoir suivre ce cours, aucun prérequis n'est nécessaire: tout sera détaillé de la manière la plus complète possible, accompagné d'exemples, d'exercices et de travaux pratiques.

Remerciements

Avant de commencer, nous souhaitons remercier plusieurs personnes:

- Mewtow pour sa participation à la rédaction et à l'évolution de ce cours ainsi que pour ses nombreux conseils;
- Arius et Saroupille pour la validation de ce cours;
- Pouet_forever, SofEvans, paraze et Mathuin pour leur soutien lors des débuts de la rédaction;
- Dominus Carnufex pour son suivi minutieux et sa bienveillance face à nos (nombreuses) fautes de français;
- Karnaj pour ses suggestions et corrections;
- Maëlan pour sa relecture attentive du chapitre sur les encodages;
- toute l'équipe de Progdupeupl et de Zeste de Savoir;
- tous ceux qui, au fil du temps et de la rédaction, nous ont apporté leurs avis, leurs conseils, leurs points de vue et qui nous ont aidés à faire de ce cours ce qu'il est aujourd'hui;
- et surtout vous, lecteurs, pour avoir choisi ce cours.

Sommaire

Avant-propos	1
Remerciements	1
Sommaire	4
I Les bases du langage C	5
1 Introduction à la programmation	7
2 Rencontre avec le C	13
3 Les variables	21
4 Manipulations basiques des entrées/sorties	33
5 Les opérations mathématiques	41
6 Tests et conditions	49
7 Les sélections	55
8 TP : déterminer le jour de la semaine	67
9 Les boucles	75
10 Les sauts	87
11 Les fonctions	91
12 TP : une calculatrice basique	103
13 Découper son projet	109
14 La gestion d'erreurs (1)	115

II Agrégats, mémoire et fichiers	119
15 Les pointeurs	121
16 Les structures	129
17 Les tableaux	141
18 Les chaînes de caractères	157
19 TP : l'en-tête <string.h>	169
20 L'allocation dynamique	177
21 Les fichiers (1)	185
22 Les fichiers (2)	197
23 Le préprocesseur	205
24 TP : un Puissance 4	217
25 La gestion d'erreur (2)	245
III Notions avancées	253
26 La représentation des types	255
27 Les limites des types	267
28 Manipulation des bits	279
29 Jeux de caractères et encodages	293
30 Les énumérations	305
31 Les unions	309
32 Les définitions de type	313
33 Les pointeurs de fonction	315
34 Les fonctions à nombre variable d'arguments	321
35 T.P. : un allocateur statique de mémoire	327
Index	339
Table des figures	339
Liste des tableaux	341
Table des matières	353

Première partie

Les bases du langage C

Introduction à la programmation

La programmation est un sujet qui fascine énormément. Si vous lisez ce cours, c'est que vous avez décidé de franchir le pas et de découvrir de quoi il s'agit. Cependant, avant de commencer à apprendre quoi que ce soit sur le C et la programmation, il est d'abord nécessaire de découvrir en quoi la programmation consiste. En effet, pour le moment, vous ne savez pas réellement ce qu'est la programmation, ce que signifie « programmer » ou encore ce qui caractérise le langage C. Ce chapitre va donc consister en une introduction au monde de la programmation, et plus particulièrement au langage C.

1.1 Avant-propos

1.1.1 Esprit et but du tutoriel

Ce cours a été écrit dans un seul but : vous enseigner le langage C de la manière la plus complète, la plus rigoureuse et la plus instructive possible. Pour ce faire, celui-ci combinera théorie, détails techniques et exercices pratiques. Dès lors, nous ne vous le cachons pas : cette approche va réclamer de votre part des **efforts**, certains passages étant assez complexes.

Nous avons choisi cette méthode d'apprentissage, car c'est celle que nous jugeons la plus profitable. Elle s'oppose à une autre, plus fréquente et plus superficielle, qui permet certes d'acquérir des connaissances rapidement, mais qui s'avère bien souvent peu payante sur le long terme.

En effet, beaucoup de programmeurs débutants se retrouvent ainsi perdus lorsqu'ils sont jetés dans la jungle de la programmation à la fin d'un cours, ceux-ci manquant souvent de connaissances techniques, de (bonnes) pratique(s) et de rigueur.

Ne soyez toutefois pas apeuré, notre objectif n'est pas de vous noyer d'informations ou de vous perdre avec des termes techniques. Nous vous précisons simplement que ce cours nécessite d'avoir les doigts sur le clavier et non dans le nez et que le début risque d'être un peu moins « *cool* » que ce que vous pourrez trouver ailleurs. ;)

1.1.2 À qui est destiné ce cours ?

À n'importe quelle personne intéressée : que vous soyez un(e) programmeur(euse) expérimenté(e), un(e) total(e) débutant(e) ou que vous vouliez réviser certaines notions du C, vous êtes tous et toutes les bienvenus(es). Les explications seront les plus claires possibles afin de rendre la lecture accessible à tous.

Toutefois, quelques qualités sont opportunes pour arriver au bout de ce cours :

- De la **motivation** : ce cours va présenter de nombreuses notions, souvent théoriques, et qui sembleront parfois complexes. Il vous faut donc être bien motivés pour profiter pleinement de cet apprentissage.

- De la **logique** : apprendre la programmation, c'est aussi être logique. Bien sûr, ce cours vous apprendra à mieux l'être, mais il faut néanmoins savoir réfléchir par soi-même et ne pas compter sur les autres pour faire le travail à sa place.
- De la **patience** : vous vous apprêtez à apprendre un langage de programmation. Pour arriver à un sentiment de maîtrise, il va vous falloir de la patience pour apprendre, comprendre, vous entraîner, faire des erreurs et les corriger.
- De la **rigueur** : cette qualité, nous allons tenter de vous l'inculquer à travers ce cours. Elle est très importante, car c'est elle qui fera la différence entre un bon et un mauvais programmeur.
- De la **curiosité** : n'hésitez pas à apporter des modifications aux codes proposés et à sortir un peu des balises du cours, cela ne vous sera que profitable.
- De la **passion** : le plus important pour suivre ce tutoriel, c'est de prendre plaisir à programmer. Amusez-vous en codant, c'est le meilleur moyen de progresser !

À noter qu'un niveau acceptable en anglais est un plus indéniable, beaucoup de cours, de forums et de documentations étant rédigés en anglais. Si ce n'est pas le cas, gardez ceci à l'esprit : en programmation, vous y serez confrontés tôt ou tard.

Enfin, un dernier point au sujet des mathématiques : contrairement à la croyance populaire, un bon niveau en maths n'est absolument pas nécessaire pour faire de la programmation. Certes, cela peut vous aider en développant votre logique, mais si les mathématiques ne sont pas votre fort, vous pourrez suivre ce cours sans problèmes.

1.2 Aller plus loin

Un des concepts fondamentaux de l'apprentissage de notions informatiques sur Internet est le *croisement des sources*. Il permet de voir la programmation sous un angle différent. Par exemple, quelques cours de [Développez](#) recourant à des approches différentes sont à votre entière disposition. N'hésitez pas non plus à lire des livres sur le C, notamment le [K&R](#), écrit par les auteurs du langage (une version traduite en français est disponible [aux éditions Dunod](#)). C'est un livre qui pourra vous être utile.

1.3 La programmation, qu'est-ce que c'est ?



Dans cette section, nous nous contenterons d'une présentation succincte qui est suffisante pour vous permettre de poursuivre la lecture de ce cours. Toutefois, si vous souhaitez un propos plus étayé, nous vous conseillons la lecture du [cours d'introduction à la programmation](#) présent sur ce site.

La programmation est une branche de l'informatique qui sert à créer des **programmes**. Tout ce que vous possédez sur votre ordinateur est un programme : votre navigateur Internet (Internet Explorer, Firefox, Opera, etc.), votre système d'exploitation (Windows, GNU/Linux, Mac OS X, etc.) qui est un regroupement de plusieurs programmes appelés **logiciels**, votre lecteur MP3, votre logiciel de discussion instantanée, vos jeux vidéos, etc.

1.3.1 Les programmes expliqués en long, en large et en travers

Un programme est une séquence d'**instructions**, d'ordres, donnés à l'ordinateur afin qu'il exécute des actions. Ces instructions sont généralement assez basiques. On trouve ainsi des instructions d'addition, de multiplication, ou d'autres opérations mathématiques de base, qui font que notre ordinateur est une vraie machine à calculer. D'autres instructions plus complexes peuvent exister, comme des opérations permettant de comparer des valeurs, traiter des caractères, etc.

Créer un programme, c'est tout simplement utiliser une suite d'instructions de base qui permettra de faire ce que l'on veut. Tous les programmes sont créés ainsi : votre lecteur MP3 donne des instructions à l'ordinateur pour écouter de la musique, le *chat* donne des instructions pour discuter avec d'autres gens sur le réseau, le système d'exploitation donne des instructions pour dire à l'ordinateur comment utiliser le matériel, etc.



Notez qu'il n'est pas possible de créer des instructions. Ces dernières sont imprimées dans les circuits de l'ordinateur ce qui fait qu'il ne peut en gérer qu'un nombre précis et qu'il ne vous est donc pas loisible d'en construire de nouvelles (sauf cas particuliers vraiment tordus).

Notre ordinateur contient un composant électronique particulier, spécialement conçu pour exécuter ces instructions : le **processeur**. Ce qu'il faut retenir, c'est que notre ordinateur contient un circuit, le processeur, qui permet d'effectuer de petits traitements de base qu'on appelle des instructions et qui sont la base de tout ce qu'on trouve sur un ordinateur.

Les instructions sont stockées dans notre ordinateur sous la forme de chiffres binaires (appelés *bits* en anglais), autrement dit sous forme de zéros ou de uns. Ainsi, nos instructions ne sont rien d'autre que des suites de zéros et de uns conservées dans notre ordinateur et que notre processeur va interpréter comme étant des ordres à exécuter. Ces suites de zéros et de uns sont difficilement compréhensibles pour nous, humains, et parler à l'ordinateur avec des zéros et des uns est très fastidieux et très long. Autant vous dire que créer des programmes de cette façon revient à se tirer une balle dans le pied.

Pour vous donner un exemple, imaginez que vous deviez communiquer avec un étranger alors que vous ne connaissez pas sa langue. Communiquer avec un ordinateur reviendrait à devoir lui donner une suite de zéros et de uns, ce dernier étant incapable de comprendre autre chose. Ce langage s'appelle le **langage machine**.

Une question doit certainement vous venir à l'esprit : comment communiquer avec notre processeur sans avoir à apprendre sa langue ?

L'idéal serait de parler à notre processeur en français, en anglais, etc, mais disons-le clairement : notre technologie n'est pas suffisamment évoluée et nous avons dû trouver autre chose. La solution retenue a été de créer des langages de programmation plus évolués que le langage machine, plus faciles à apprendre et de fournir le traducteur qui va avec. Il s'agit de langages assez simplifiés, souvent proches des langages naturels et dans lesquels on peut écrire nos programmes beaucoup plus simplement qu'en utilisant le langage machine. Grâce à eux, il est possible d'écrire nos programmes sous forme de texte, sans avoir à se débrouiller avec des suites de zéros et de uns totalement incompréhensibles. Il existe de nombreux langages de programmation et l'un d'entre-eux est le **C**.

Reste que notre processeur ne comprend pas ces langages évolués et n'en connaît qu'un seul : le sien. Aussi, pour utiliser un langage de programmation, il faut disposer d'un traducteur qui fera le lien entre celui-ci et le langage machine du processeur. Ainsi, il ne vous est plus nécessaire de connaître la langue de votre processeur. En informatique, ce traducteur est appelé un **compilateur**.

Pour illustrer notre propos, voici un code écrit en C (que nous apprendrons à connaître).

```
1 #include <stdio.h>
2
3 int main(void) { printf("Salut !\n"); return 0; }
```

Et le même en langage machine (plus précisément pour un processeur de la famille x86-64).

```

1 01010101 01001000 10001001 11100101 10111111 00100100 00101100
2 01001000 00000000 10111000 00000000 00000000 00000000 00000000
3 11101000 10011101 00001011 00000000 00000000 10111000 00000000
4 00000000 00000000 00000000 01011101 11000011 01010011 01100001
5 01101100 01110101 01110100 00100000 00100001 00001010 00000000

```

Nous y gagnons tout de même au change, non ? :pMalgré tous ces langages de programmation disponibles nous allons, dans ce tutoriel, nous concentrer sur un seul d’entre-eux : le C. Avant de parler des caractéristiques de ce langage et des choix qui nous amènent à l’étudier dans ce cours, faisons un peu d’histoire.

1.4 Le langage C

1.4.1 L’histoire du C

Le langage C est né au début des années 1970 dans les laboratoires de la société AT&T aux États-Unis. Son concepteur, [Dennis MacAlistair Ritchie](#), souhaitait améliorer un langage existant, le B, afin de lui adjoindre des nouveautés. En 1973, le C était pratiquement au point et il commença à être distribué l’année suivante. Son succès fut tel auprès des informaticiens qu’en 1989, l’ANSI, puis en 1990, l’ISO, décidèrent de le normaliser, c’est-à-dire d’établir des règles internationales et officielles pour ce langage. À l’heure actuelle, il existe trois normes : la norme ANSI C89 ou ISO C90, la norme ISO C99 et la norme ISO C11.

[AT&T] : *American Telephone and Telegraph Company* [ANSI] : American National Standards Institute *[ISO] : International Organization for Standardization



Si vous voulez en savoir plus sur l’histoire du C, lisez donc [ce tutoriel](#).

1.4.2 Pourquoi apprendre le C ?

C’est une très bonne question. :D Après tout, étant donné qu’il existe énormément de langages différents, il est légitime de se demander pourquoi choisir le C en particulier ? Il y a plusieurs raisons à cela.

- Sa **popularité** : le C fait partie des langages de programmation les plus utilisés. Il possède une communauté très importante, de nombreux cours et beaucoup de documentations. Vous aurez donc toujours du monde pour vous aider. De plus, il existe un grand nombre de programmes et de bibliothèques développés en C.
- Sa **rapidité** : le C est connu pour être un langage très rapide, ce qui en fait un langage de choix pour tout programme où la vitesse d’exécution est cruciale.
- Sa **simplicité** : le C est un langage minimaliste pourvu de peu de concepts ce qui permet d’en faire le tour *relativement* rapidement et d’éviter un niveau d’abstraction trop important.
- Sa **légèreté** : le C est léger, ce qui le rend utile pour les programmes embarqués où la mémoire disponible est faible.
- Sa **portabilité** : cela signifie qu’un programme développé en C peut être compilé pour fonctionner sur différentes machines sans devoir changer ledit code.

Ce ne sont que quelques raisons, mais elles sont à notre goût suffisantes pour justifier l’apprentissage de ce langage. Bien entendu, le C comporte aussi sa part de défauts. On peut citer la tolérance aux comportements dangereux qui fait que le C demande de la rigueur pour ne pas tomber dans certains « pièges », un nombre plus restreint de concepts (c’est parfois un désavantage, car on est alors obligé de recoder certains mécanismes qui existent nativement dans

d'autres langages), etc. D'ailleurs, si votre but est de développer rapidement des programmes amusants, sachez que le C n'est pas adapté pour cela et que nous vous conseillons, dans ce cas, de vous tourner vers d'autres langages, comme par exemple le [Python](#) ou le [Ruby](#).

Le C possède aussi une caractéristique qui est à la fois un avantage et un défaut : il s'agit d'un langage dit de « **bas niveau** ». Cela signifie qu'il permet de programmer en étant « proche de sa machine », c'est-à-dire sans trop vous cacher son fonctionnement interne. Cette propriété est à double tranchant : d'un côté elle rend l'apprentissage plus difficile et augmente le risque d'erreurs ou de comportements dangereux, mais de l'autre elle vous laisse une grande liberté d'action et vous permet d'en apprendre plus sur le fonctionnement de votre machine. Cette notion de « bas niveau » est d'ailleurs à opposer aux langages dit de « **haut niveau** » qui permettent de programmer en faisant abstraction d'un certain nombre de choses. Le développement est rendu plus facile et plus rapide, mais en contrepartie, beaucoup de mécanisme interne sont cachés et ne sont pas accessibles au programmeur. Ces notions de haut et de bas niveau sont néanmoins à nuancer, car elles dépendent du langage utilisé et du point de vue du programmeur (par exemple, par rapport au langage machine, le C est un langage de haut niveau).

Une petite note pour terminer : peut-être avez-vous entendu parler du **C++** ? Il s'agit d'un langage de programmation qui a été inventé dans les années 1980 par [Bjarne Stroustrup](#), un collègue de Dennis Ritchie, qui souhaitait rajouter des éléments au C. Bien qu'il fût très proche du C lors de sa création, le C++ est aujourd'hui un langage très différent du C et n'a pour ainsi dire plus de rapport avec lui (si ce n'est une certaine proximité au niveau d'une partie de sa syntaxe). Ceci est encore plus vrai en ce qui concerne la manière de programmer et de raisonner qui sont *radicalement* différentes.

Ne croyez toutefois pas, comme peut le laisser penser leur nom ou leur date de création, qu'il y a un langage meilleur que l'autre, ils sont simplement *différents*. Si d'ailleurs votre but est d'apprendre le C++, nous vous encourageons à le faire. En effet, contrairement à ce qui est souvent dit ou lu, *il n'y a pas besoin de connaître le C pour apprendre le C++*.

1.5 La norme

Comme précisé plus haut, le C est un langage qui a été normalisé à trois reprises. Ces normes servent de référence à tous les programmeurs et les aident chaque fois qu'ils ont un doute ou une question en rapport avec le langage. Bien entendu, elle ne sont pas parfaites et ne répondent pas à toutes les questions, mais elles restent *la* référence pour tout programmeur.

Ces normes sont également indispensables pour les compilateurs. En effet, le respect de ces normes par les différents compilateurs permet qu'il n'y ait pas de différences d'interprétation d'un même code. Finalement, ces normes sont l'équivalent de nos règles d'orthographe, de grammaire et de conjugaison. Imaginez si chacun écrivait ou conjugait à sa guise, ce serait un sacré bazar...

Dans ce cours, nous avons décidé de nous reposer sur la norme ANSI C89 (ou ISO C90, c'est pareil). En effet, même s'il s'agit de la plus ancienne, elle nous permettra néanmoins de développer avec n'importe quel compilateur et sous n'importe quel système sans problèmes et sans nous poser de questions sur la présence ou non de telle ou telle fonctionnalité.

Rassurez-vous néanmoins : le fait de nous baser sur la norme C89 ne signifie pas que vous allez découvrir une version obsolète du langage C. En effet, d'une part, ce que vous allez voir tout au long de ce cours est toujours valable au regard des normes plus récentes et, d'autre part, les changements induits par les autres normes sont le plus souvent mineurs et consistent pour ainsi dire tous en des *ajouts* et non en des modifications. De ce fait, il vous sera aisé, une fois ce cours parcouru, de passer à une norme plus récente.



Pour les curieux, voici [un lien](#) vers le brouillon de cette norme. Cela signifie qu'il ne s'agit pas de la version définitive et officielle, cependant il est largement suffisant pour



notre niveau et, surtout, il est gratuit (la norme officielle coûtant *très* cher :-°). Notez que celui-ci est rédigé en anglais

1.6 L’algorithmique

L’algorithmique est liée à la programmation et constitue même une branche à part des mathématiques. Elle consiste à définir et établir des **algorithmes**.

Un algorithme peut se définir comme étant une suite finie et non-ambiguë d’opérations permettant de résoudre un problème. En clair, il s’agit de calculs qui prennent plusieurs paramètres et fournissent un résultat. Les algorithmes ne sont pas limités à l’informatique, ils existaient même avant son apparition ; prenez les recettes de cuisine par exemple, ou des instructions de montage d’un meuble ou d’un Lego, ce sont des algorithmes.

L’intérêt principal des algorithmes est qu’ils sont très utiles lorsqu’ils sont en relation avec des ordinateurs. En effet, ces derniers peuvent exécuter des milliards d’instructions à la seconde, ce qui les rend bien plus rapides qu’un humain. Illustrons : imaginez que vous deviez trier une liste de dix nombres dans l’ordre croissant. C’est assez facile et faisable en quelques secondes. Et pour plusieurs milliards de nombres ? C’est impossible pour un humain, alors qu’un ordinateur le fera rapidement.

Ce qu’il faut retenir, c’est qu’un algorithme est une suite d’opérations destinée à résoudre un problème donné. Nous aurons l’occasion d’utiliser quelques algorithmes dans ce cours, mais nous ne nous concentrerons pas dessus.

Si vous voulez en savoir plus, lisez le tutoriel sur [l’algorithmique pour l’apprenti programmeur](#) en même temps que vous apprenez à programmer avec celui-ci.

1.6.1 Le pseudo-code

Pour représenter un algorithme indépendamment de tout langage, on utilise ce qu’on appelle un **pseudo-code**. Il s’agit de la description des étapes de l’algorithme en langage naturel (dans notre cas le français). Voici un exemple de pseudo-code.

```

1  Fonction max (x, y)
2
3  Si x est supérieur à y Retourner x Sinon Retourner y
4
5  Fin fonction
```

Dans ce cours, il y aura plusieurs exercices dans lesquels un algorithme fourni devra être mis en œuvre, traduit en C. Si vous voulez vous entraîner davantage tout en suivant ce cours, nous vous conseillons [France-IOI](#) qui permet de mettre en application divers algorithmes dans plusieurs langages, dont le C. Cela pourra être un excellent complément.

Comme vous avez pu le constater, la programmation est un monde vaste, très vaste, et assez complexe. Comme il existe une multitude de langages de programmation, il faut se concentrer sur un seul d’entre eux à la fois. Dans notre cas, il s’agit du C. Ce langage, et retenez-le bien, est à la fois puissant et complexe. Rappelez-vous bien qu’il vous faudra faire des efforts pour l’apprendre correctement.

Si vous vous sentez prêts, alors rendez-vous dans le chapitre suivant, qui vous montrera les outils utilisés par un programmeur C.

Maintenant que les présentations sont faites, il est temps de découvrir les outils nécessaires pour programmer en C. Le strict minimum pour programmer se résume en trois points :

- un **éditeur de texte** (à ne pas confondre avec un **traitement de texte** comme *Microsoft Word* ou *LibreOffice Writer*) : ce logiciel va servir à l'écriture du code source. Techniquement, n'importe quel éditeur de texte suffit, mais il est souvent plus agréable d'en choisir un qui n'est pas trop minimaliste ;
- un **compilateur** : c'est le logiciel le plus important puisqu'il va nous permettre de transformer le code écrit en langage C en un fichier exécutable ;
- un **débogueur** (ou *debugger* en anglais) : ce logiciel vous sera très utile en cas de problèmes pour rechercher d'éventuelles erreurs dans votre programme.

À partir de là, il existe deux solutions : utiliser ces trois logiciels séparément ou bien les utiliser au sein d'un **environnement intégré de développement** (abrégé EDI). Dans le cadre de ce cours, nous avons choisi la première option, majoritairement dans un souci de transparence et de simplicité. En effet, si les EDI peuvent être des compagnons de choix, ceux-ci sont avant tout destinés à des programmeurs expérimentés et non à de parfaits débutants

2.1 Windows

2.1.1 Le compilateur

Nous vous proposons de télécharger MinGW, qui est une adaptation pour Windows du compilateur GCC.

Rendez-vous sur le [site de MinGW](#) dans la section « *download* » et cliquez sur le lien en haut de la page « *looking for the latest version ? Download mingw-get-install-xxxxxxx.exe (xxx.x kB)* ».

Exécutez le programme, cliquez sur « *install* », décochez la case « *also install support for graphical user interface* » et enfin cliquez sur « *continue* ».

Ceci étant fait, il nous faut désormais créer une variable d'environnement afin de spécifier à notre invite de commande le chemin vers les différents composants de MinGW.

- sous Windows XP et antérieur, faites un clic-droit sur « poste de travail » puis choisissez « propriétés ». Dans la fenêtre qui s'ouvre, cliquez sur « avancés » puis sur « variables d'environnement » ;
- sous Windows Vista, Seven, faites un clic-droit sur l'icône « ordinateur » dans le menu « démarrer » ou bien sur « poste de travail ». Ensuite, cliquez sur « paramètres systèmes avancés ». Dans la nouvelle fenêtre qui s'ouvre, cliquez sur « variables d'environnement » ;

- sous Windows 8, rendez-vous dans le panneau de configuration à la rubrique « système ». Cliquez sur « avancé » puis sur « variables d'environnement ».

Dans la partie « utilisateur courant », créez une nouvelle variable nommée `PATH` et donnez lui pour valeur : `%PATH%;C:\MinGW\bin` (le chemin après le point-virgule peut varier en fonction de où vous avez décidés d'installer MinGW, l'important est de bien avoir le répertoire `bin` à la fin).

À présent, exécutez l'invite de commandes (il est situé dans les accessoires sous le même nom) et entrez la ligne suivante.

```
1 mingw-get install gcc gdb
```

Le compilateur et le débogueur sont à présent installés. À présent, lancez le bloc-note et placez y le texte suivant.

```
1 @echo off gcc -D__USE_MINGW_ANSI_STDIO=1 -Wall -Wextra -pedantic -std=c89 -fno-common
2 -fno-builtin %*
```

Ensuite, enregistrez ce fichier dans le dossier « bin » de MinGW (par défaut `C:\MinGW\bin`) sous le nom « `zcc.bat` » en choisissant « autres types de fichiers ».

Maintenant, rendez-vous dans le menu des accessoires, réalisez un clic droit sur l'invite de commande et sélectionnez « propriétés ». Dans l'onglet « raccourci », remplacer le champ « cible » par « `%windir%\system32\cmd.exe /k "chcp 65001"` ». Enfin, dans l'onglet « police », choisissez « Consolas » ou « Lucida Console » et adaptez la taille suivant vos envies.

2.2 L'éditeur de texte

L'éditeur de texte va nous permettre d'écrire notre code source et de l'enregistrer. L'idéal est d'avoir un éditeur de texte facile à utiliser et pas trop minimaliste. Si jamais vous avez déjà un éditeur de texte et que vous l'appréciez, n'en changez pas, il fera sûrement l'affaire.

Si vous n'avez pas d'idée, nous vous conseillons [Notepad++](#) qui est simple, pratique et efficace. Pour le télécharger, rendez-vous simplement dans la rubrique « Téléchargements » du menu principal.



Veillez-bien à ce que l'encodage de votre fichier soit « UTF-8 (sans BOM) » (voyez le menu éponyme à cet effet).

2.3 Introduction à la ligne de commande

La ligne de commande, derrière son aspect rustre et archaïque, n'est en fait qu'une autre manière de réaliser des tâches sur un ordinateur. La différence majeure avec une interface graphique étant que les instructions sont données non pas à l'aide de boutons et de cliques de souris, mais exclusivement à l'aide de texte. Ainsi, pour réaliser une tâche donnée, il sera nécessaire d'invoquer un programme (on parle souvent de **commandes**) en tapant son nom.

La première chose que vous devez garder à l'esprit, c'est le dossier dans lequel vous vous situez. Celui-ci est indiqué au tout début de chaque ligne et se termine par le symbole `>`. Ce dossier est celui dans lequel les actions (par exemple la création d'un répertoire) que vous demanderez seront réalisées. Normalement, par défaut, vous devriez vous situez dans le répertoire `C:\Users\Utilisateur` (où `Utilisateur` correspond à votre nom d'utilisateur). Ceci étant posé, voyons quelques commandes basiques.

La commande `mkdir` (pour *make directory*) vous permet de créer un nouveau dossier. Pour ce faire, tapez `mkdir` suivi d'un espace et du nom du nouveau répertoire. Par exemple, vous pouvez créer un dossier « Programmation » comme suit.

```
1 C:\Users\Utilisateur> mkdir Programmation
```

La commande `dir` (pour *directory*) vous permet de lister le contenu d'un dossier. Vous pouvez ainsi vérifier qu'un nouveau répertoire a bien été créé.

```
1 C:\Users\Utilisateur> dir Répertoire de C:\Users\Utilisateur
2
3 30/03/2015 17:00 <REP> .   30/03/2015 17:00 <REP> ..  30/03/2015
4 17:00 <REP> Programmation
```



Le résultat ne sera pas forcément le même que ci-dessus, cela dépend du contenu de votre dossier. L'essentiel est que vous retrouviez bien le dossier que vous venez de créer.

Enfin, la commande `cd` (pour *change directory*) vous permet de vous déplacer d'un dossier à l'autre. Pour ce faire, spécifiez simplement le nom du dossier de destination.

```
1 C:\Users\Utilisateur> cd Programmation
2 C:\Users\Utilisateur\Programmation>
```



Le dossier spécial « .. » représente le répertoire parent. Il vous permet donc de revenir en arrière dans la hiérarchie des dossiers. Le dossier spécial « . » représente quant à lui le dossier courant.

Voilà, avec ceci, vous êtes fin prêt pour compiler votre premier programme. Vous pouvez vous rendre à la deuxième partie de ce chapitre.

[MinGW] : *Minimalist GNU for Windows* [GCC] : GNU Compiler Collection.

2.4 GNU/Linux, *BSD et autres Unixôides

2.4.1 Le compilateur

Suivant le système que vous choisissiez, vous aurez ou non le choix entre différents compilateurs. Si vous n'avez pas d'idée, nous vous conseillons d'opter pour GCC en installant le paquet éponyme à l'aide de votre gestionnaire de paquet. Également, vous pouvez installer le paquet « gdb » qui est un débogueur.

2.4.2 Configuration

Ceci dépend de votre interpréteur de commande. Pour savoir quel est celui dont vous disposez, ouvrez un terminal (le plus souvent, vous pouvez y accéder via la catégorie « accessoires » de votre menu principal) et entrez la commande suivante.

```
1 echo $SHELL
```

bash

Exécutez la commande suivante depuis votre dossier personnel (vous y êtes par défaut au lancement de l'invite de commande).

```
1 echo "alias zcc='gcc -Wall -Wextra -pedantic -std=c89 -fno-common -fno-builtin'"
2 >> .bashrc
```

csh ou tcsh

Exécutez la commande suivante depuis votre dossier personnel (vous y êtes par défaut au lancement de l'invite de commande).

```
1 echo "alias zcc 'gcc -Wall -Wextra -pedantic -std=c89 -fno-common -fno-builtin'"
2 >> .cshrc # (ou .tcshrc)
```

ksh, zsh ou sh

Exécutez les commandes suivante depuis votre dossier personnel (vous y êtes par défaut au lancement de l'invite de commande).

```
1 echo "alias zcc='gcc -Wall -Wextra -pedantic -std=c89 -fno-common -fno-builtin'"
2 >> .kshrc # (ou .zshrc ou .shrc) echo "export ENV=\$HOME.kshrc"
3 >> .profile # (ou .zshrc ou .shrc)
```

2.4.3 L'éditeur de texte

Ce serait un euphémisme de dire que vous avez l'embarras du choix. Il existe une pléthore d'éditeurs de texte fonctionnant en ligne de commande ou avec une interface graphique, voire les deux.

Pour n'en citer que quelques-uns, en ligne de commande vous trouverez par exemple : Vim et Emacs (les deux monstres de l'édition), Nano ou Joe. Côté graphique, vous avez entre autres : Gedit, Mousepad et Kate.

2.4.4 Introduction à la ligne de commande

La ligne de commande, derrière son aspect rustre et archaïque, n'est en fait qu'une autre manière de réaliser des tâches sur un ordinateur. La différence majeure avec une interface graphique étant que les instructions sont données non pas à l'aide de boutons et de cliques de souris, mais exclusivement à l'aide de texte. Ainsi, pour réaliser une tâche donnée, il sera nécessaire d'invoquer un programme (on parle souvent de **commandes**) en tapant son nom.

La première chose que vous devez garder à l'esprit, c'est le dossier dans lequel vous vous situez. Suivant le terminal que vous employez, celui-ci est parfois indiqué en début de ligne et terminé par le symbole `$` ou `%`. Ce dossier est celui dans lequel les actions (par exemple la création d'un répertoire) que vous demanderez seront exécutées. Normalement, par défaut, vous devriez vous situez dans le répertoire : `/home/utilisateur` (où `utilisateur` correspond à votre nom d'utilisateur). Ceci étant posé, voyons quelques commandes basiques.

La commande `pwd` (pour *print working directory*) vous permet de connaître le répertoire dans lequel vous êtes.

```
1 $ pwd /home/utilisateur
```

La commande `mkdir` (pour *make directory*) vous permet de créer un nouveau dossier. Pour ce faire, tapez `mkdir` suivi d'un espace et du nom du nouveau répertoire. Par exemple, vous pouvez créer un dossier « programmation » comme suit :

```
1 $ mkdir programmation
```

La commande `ls` (pour *list*) vous permet de lister le contenu d'un dossier. Vous pouvez ainsi vérifier qu'un nouveau répertoire a bien été créé.

```
1 $ ls programmation
```



Le résultat ne sera pas forcément le même que ci-dessus, cela dépend du contenu de votre dossier. L'essentiel est que vous retrouviez bien le dossier que vous venez de créer.

Enfin, la commande `cd` (pour *change directory*) vous permet de vous déplacer d'un dossier à l'autre. Pour ce faire, spécifiez simplement le nom du dossier de destination.

```
1 $ cd programmation
2 $ pwd /home/utilisateur/programmation
```



Le dossier spécial « `..` » représente le répertoire parent. Il vous permet donc de revenir en arrière dans la hiérarchie des dossiers. Le dossier spécial « `.` » représente quant à lui le dossier courant.

Voilà, avec ceci, vous êtes fin prêt pour compiler votre premier programme. Vous pouvez vous rendre à la deuxième partie de ce chapitre.

*[GCC] : GNU Compiler Collection

2.5 Mac OS X

2.5.1 Le compilateur

Allez dans le dossier `/Applications/Utilitaires` et lancez l'application « terminal.app ». Une fois ceci fait, entrez la commande suivante :

```
1 xcode-select --install
```

et cliquez sur « installer » dans la fenêtre qui apparaît. Si vous rencontrez le message d'erreur ci-dessous, cela signifie que vous disposez déjà des logiciels requis.

```
1 Impossible d'installer ce logiciel car il n'est pas disponible
2 actuellement depuis le serveur de mise à jour de logiciels.
```

Si vous ne disposez pas de la commande indiquée, alors rendez-vous sur le site de développeur d'Apple : [Apple developer connection](https://developer.apple.com). Il faudra ensuite vous rendre sur le « *mac dev center* » puis, dans « *additional download* », cliquez sur « *view all downloads* ». Quand vous aurez la liste, il vous suffit de chercher la version 3 de Xcode (pour Leopard et Snow Leopard) ou 2 pour les versions antérieures (Tiger). Vous pouvez aussi utiliser votre CD d'installation pour installer Xcode (sauf pour Lion).

2.5.2 Configuration

Voyez ce qui est dit pour GNU/Linux, *BSD et les autres Unixoides.

2.5.3 L'éditeur de texte

Comme pour les autres Unixoides, vous trouverez un bon nombres d'éditeurs de texte. Si toutefois vous êtes perdu, nous vous conseillons [TextWrangler](#) ou [Smultron](#).

2.5.4 Introduction à la ligne de commande

Référez-vous à l'introduction dédiée à GNU/Linux, *BSD et les autres Unixoides.

2.6 Notre cible

Avant de commencer à programmer, il nous faut aussi définir ce que nous allons programmer, autrement dit le type de programme que nous allons réaliser. Il existe en effet deux grands types de programmes : les programmes **graphiques** et les programmes **en console**.

Les programmes graphiques sont les plus courants et les plus connus puisqu'il n'y a pratiquement qu'eux sous Windows ou Mac OS X par exemple. Vous en connaissez sans doute énormément tel que les lecteurs de musique, les navigateurs Internet, les logiciels de discussions instantanées, les suites bureautiques, les jeux vidéos, etc. Ce sont tous des programmes graphiques, ou programmes GUI. En voici un exemple sous GNU/Linux.

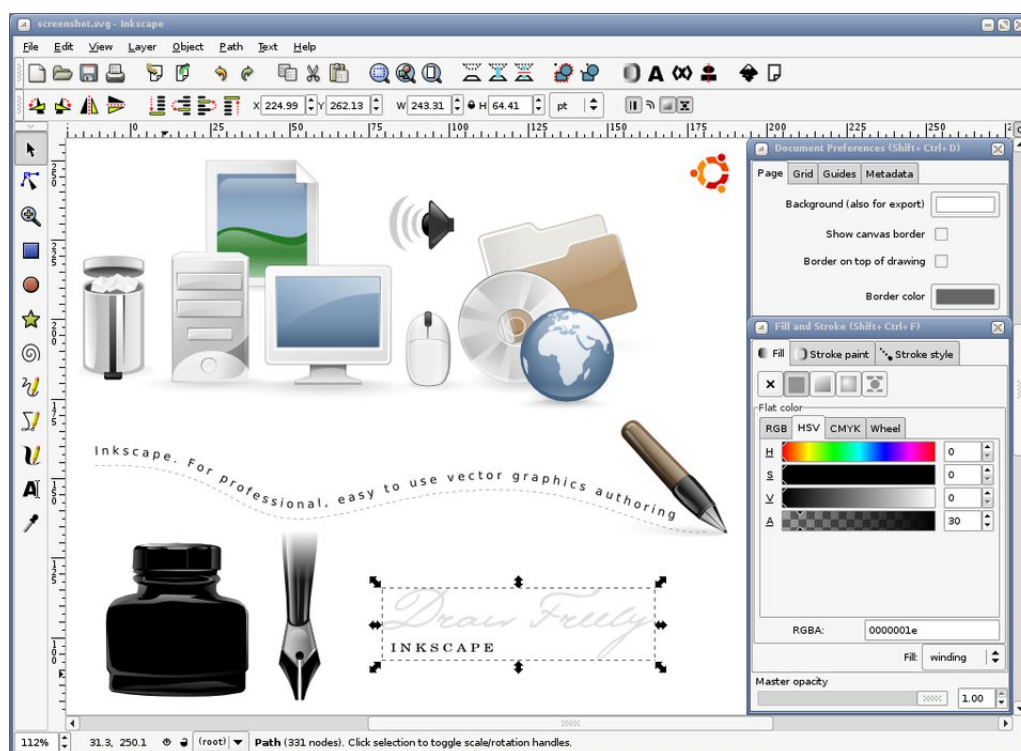


FIGURE 2.1 – L'éditeur d'images vectorielles Inkscape est un programme graphique

Cependant, écrire ce genre de programmes demande beaucoup de connaissances, ce qui nous manque justement pour l'instant. Aussi, nous allons devoir nous rabattre sur le deuxième type de programme : les programmes en console.

Les programmes console sont les premiers programmes et sont apparus en même temps que l'écran. Ils étaient très utilisés dans les années 1970/1980 (certains d'entre vous se souviennent

peut-être de MS-DOS), mais ont fini par être remplacés par une interface graphique avec la sortie de Mac OS et de Windows. Cependant, ils existent toujours et redeviennent quelque peu populaires chez les personnes utilisant GNU/Linux ou un *BSD.

Voici un exemple de programme en console (il s'agit de [GNU Chess](#), un jeu d'échecs performant entièrement en ligne de commande).

```

1  White (1) : d4
2  1. d4
3
4  black KQkq d3
5  r n b q k b n r
6  p p p p p p p p
7  . . . . .
8  . . . . .
9  . . . P . . .
10 . . . . .
11 P P P . P P P P
12 R N B Q K B N R
13
14 Thinking...
15
16 white KQkq
17 r n b q k b . r
18 p p p p p p p p
19 . . . . . n . .
20 . . . . .
21 . . . P . . .
22 . . . . .
23 P P P . P P P P
24 R N B Q K B N R
25
26
27 My move is : Nf6
28 White (2) :
```

Ce sera le type de programme que nous allons apprendre à créer. Rassurez-vous, quand vous aurez fini le cours, vous aurez les bases pour apprendre à créer des programmes graphiques. Tout est possible.

2.7 Première rencontre

Bien, il est à présent temps d'écrire et de compiler notre premier programme ! Pour ce faire, ouvrez votre éditeur de texte et entrez les lignes suivantes.

```

1  int main(void)
2  {
3      return 0
4  }
```

Ensuite, enregistrez ce fichier dans un dossier de votre choix et nommez-le « main.c ». Une fois ceci fait, rendez-vous dans le dossier contenant le fichier à l'aide d'un terminal et exécutez la commande ci-dessous.

```

1  zcc main.c
```


Si vous n'êtes pas sous Windows et que vous utilisez l'interpréteur de commande `sh`, `ksh` ou `zsh`, la commande `zcc` ne sera connue de votre invite de commande qu'une fois que vous aurez ouvert une nouvelle session. En attendant, vous pouvez entrer la commande `alias zcc='gcc -Wall -Wextra -pedantic -std=c89 -fno-common -fno-builtin'` dans votre terminal pour que cela fonctionne.

Si tout se passe bien, vous devriez obtenir un fichier « a.exe » sous Windows et un fichier « a.out » sinon. Vous pouvez exécuter ce programme en tapant `a.exe` ou `./a.out`.

❓ Je viens de le faire, mais il ne se passe rien

Cela tombe bien, c'est exactement ce que fait ce programme : rien. :p Voyons cela plus en détails.

Ce bout de code est appelé une **fonction**. Un programme écrit en C n'est composé pratiquement que de fonctions qui sont des morceaux de programme donnant des instructions à l'ordinateur. Elles ont toujours un objectif, une *fonction* particulière, par exemple calculer la racine carrée d'un nombre.

Notre fonction s'appelle `main` (prononcez « mène »). C'est la fonction de base commune à tous les programmes en C, le programme commence et finit toujours par elle.

Une fonction est délimitée par des accolades (`{` et `}`). Après les accolades, il n'y a rien, car pour l'instant nous n'avons que la fonction `main`.

Le nom de la fonction est précédé du mot-clé `int` qui est un nom de type (nous verrons cette notion au chapitre suivant) qui indique que la fonction retourne une valeur entière. À l'intérieur des parenthèses, il y a le mot `void` qui signifie que la fonction ne reçoit pas de paramètres, nous y reviendrons en temps voulu.

Enfin, la fonction se termine par l'instruction `return 0` qui signifie en l'occurrence que la fonction a terminé son travail (bon, pour l'instant elle n'en a pas, mais nous y arriverons rapidement) et que tout s'est bien passé.

2.8 Les commentaires

Il est souvent nécessaire de **commenter son code source** pour décrire des passages un peu moins lisibles ou tout simplement pour offrir quelques compléments d'information au lecteur du code. Nous en utiliserons souvent dans la suite de ce cours pour rendre certains exemples plus parlant.

Un commentaire est ignoré par le compilateur, il disparaît et n'est pas présent dans l'exécutable. Il ne sert qu'au programmeur et aux lecteurs du code.

Un commentaire en C est écrit entre les signes `/*` et `*/` et peut très bien prendre plusieurs lignes.

```
1  /* Ceci est un commentaire. */
```

```
1  /* Ceci est un commentaire qui
2
3  prend plusieurs lignes. */
```

Voilà, vous avez enfin fait la connaissance du C à travers du code. Certes, nous n'avons vu qu'un petit code et avons seulement survolé les différents éléments, mais il n'empêche que cela représente certainement beaucoup de nouveautés pour vous. Relisez donc ce chapitre à tête reposée si nécessaire.

Programmer, c'est avant tout donner des ordres à notre ordinateur afin qu'il réalise ce que l'on souhaite. Ces ordres vont permettre à notre ordinateur de manipuler de l'information sous différentes formes (nombres, textes, vidéos, etc). À ce stade, nous savons que ces ordres, ces instructions sont exécutées par notre processeur. Cependant, nous ne savons toujours pas comment donner des ordres, ni comment manipuler de l'information.

Ce chapitre vous expliquera comment manipuler les types de données les plus simples du langage C, les nombres et les lettres (ou caractères), grâce à ce qu'on appelle des **variables**. Après celui-ci, vous pourrez ainsi profiter de votre ordinateur comme s'il s'agissait d'une grosse calculatrice. Néanmoins, rassurez-vous, le niveau en maths de ce chapitre sera assez faible : si vous savez compter, vous pourrez le comprendre facilement !

Cela peut paraître un peu bête et pas très intéressant, mais il faut bien commencer par les bases. Manipuler du texte ou de la vidéo est complexe et nécessite en plus de savoir comment manipuler des nombres. *Eh oui !* Comme vous allez le voir, tout est nombre pour notre ordinateur, même le texte et la vidéo.

3.1 Qu'est-ce qu'une variable ?

Pour comprendre ce qu'est une variable et comment manipuler celles-ci, il faut commencer par comprendre comment notre ordinateur fait pour stocker des données. En théorie, un ordinateur est capable de stocker tout type d'information. Mais comment est-il possible de réaliser un tel miracle alors qu'il ne s'agit finalement que d'un amas de circuits électriques ?

3.1.1 Codage des informations

Peut-être avez-vous déjà entendu le proverbe suivant : « si le seul outil que vous avez est un marteau, vous verrez tout problème comme un clou » (Abraham Maslow). *Hé* bien, l'idée est un peu la même pour un ordinateur : ce dernier ne sachant utiliser que des nombres, il voit toute information comme une suite de nombres.

L'astuce consiste à transformer une information en nombre pour que l'ordinateur puisse la traiter, autrement dit la **numériser**. Différentes techniques sont possibles pour atteindre cet objectif, une des plus simples étant une table de correspondance, par exemple entre un nombre et un caractère.

Caractère	Nombre
A	1
B	2
C	3

3.1.2 Binaire

Cependant, comme si cela ne suffisait pas, un ordinateur ne compte pas comme nous : il compte en base deux (l'andouille!).

? En base deux ?

La base correspond au nombre de chiffres disponibles pour représenter un nombre. En base 10, nous disposons de dix chiffres : zéro, un, deux, trois, quatre, cinq, six, sept, huit et neuf. En base deux, nous en avons donc... deux : zéro et un. Pour ce qui est de compter, c'est du pareil au même : nous commençons par épuiser les unités : 0, 1 ; puis nous passons aux dizaines : 10, 11 ; puis aux centaines : 100, 101, 110, 111 ; et ainsi de suite. Ci-dessous un petit tableau de correspondance entre la base deux et la base dix.

Base deux	Base dix
0	0
1	1
10	2
11	3
100	4
101	5
110	6
111	7
1000	8
1001	9
1010	10

TABLE 3.1 – Correspondance base deux-base dix

Un chiffre binaire (un zéro ou un un) est appelé un *bit* en anglais. Il s'agit de la contraction de l'expression « *binary digit* ». Nous l'emploierons assez souvent dans la suite de ce cours par souci d'économie.

? Mais pourquoi utiliser la base deux et non la base dix ?

Parce que les données circulent sous forme de courants électriques. Or, la tension de ceux-ci n'étant pas toujours stable, il est difficile de réaliser un système fiable sachant détecter dix valeurs différentes. Par contre, c'est parfaitement possible avec deux valeurs : il y a du courant ou il n'y en a pas.

3.2 La mémoire

Nous savons à présent que notre ordinateur ne sait employer que des nombres représentés en base deux.

? Mais comment stocker tout ce fatras de nombres ?

Hé bien, les *bits* sont stockés dans un composant électronique particulier de l'ordinateur : la **mémoire**. Enfin, nous disons « la mémoire », mais il y en a en fait plusieurs.

? Mais pourquoi plusieurs mémoires et pas une seule ?

Le fait est qu'il est actuellement impossible de créer des mémoires qui soient à la fois rapides et capables de contenir beaucoup de données. Nous ne pouvons donc utiliser une seule grosse

mémoire capable de stocker toutes les données dont nous avons besoin. Ce problème s'est posé dès les débuts de l'informatique, comme en témoigne cette citation des années 1940, provenant des concepteurs d'un des tout premiers ordinateurs.

Idéalement, nous désirerions une mémoire d'une capacité indéfiniment large telle que n'importe quelle donnée soit immédiatement accessible. Nous sommes forcés de reconnaître la possibilité de la construction d'une hiérarchie de mémoire, chacune ayant une capacité plus importante que la précédente, mais accessible moins rapidement.

Source : Burks, Goldstine, et Von Neumann

Mais les chercheurs et ingénieurs du début de l'informatique ont trouvé une solution : segmenter la mémoire de l'ordinateur en plusieurs sous-mémoires, de taille et de vitesse différentes, utilisées chacune suivant les besoins. Nous aurons donc des mémoires pouvant contenir peu de données et rapides, à côté de mémoires plus importantes et plus lentes.

Nous vous avons dit que l'ordinateur utilisait plusieurs mémoires. Trois d'entre elles méritent à notre sens votre attention :

- les registres ;
- la mémoire vive (ou RAM en anglais) ;
- le disque dur.

*[RAM] : Random Access Memory

Les **registres** sont des mémoires intégrées dans le processeur, utilisées pour stocker des données temporaires. Elles sont très rapides, mais ne peuvent contenir que des données très simples, comme des nombres.

La **mémoire vive** est une mémoire un peu plus grosse, mais plus lente que les registres. Elle peut contenir pas mal de données et est généralement utilisée pour stocker les programmes en cours d'exécution ainsi que les données qu'ils manipulent.

Ces deux mémoires (les registres et la mémoire vive) ont tout de même un léger défaut : elles perdent leur contenu quand elles ne sont plus alimentées... Autant dire que ce n'est pas le meilleur endroit pour stocker un système d'exploitation ou des fichiers personnels. Ceci est le rôle du **disque dur**, une mémoire avec une capacité très importante, mais très lente qui a toutefois l'avantage d'assurer la persistance des données.

En C, la mémoire la plus manipulée par le programmeur est la mémoire vive. Aussi, nous allons nous y intéresser d'un peu plus près dans ce qui suit.

Bits, multipléts et octets

Dans la mémoire vive, les *bits* sont regroupés en « paquets » de quantité fixe : des « **cases mémoires** », aussi appelées **multipléts** (ou *bytes* en anglais). À quelques exceptions près, les mémoires utilisent des multipléts de huit *bits*, aussi appelés **octet**. Un octet peut stocker 256 informations différentes (vous pouvez faire le calcul vous-même : combien vaut 11111111 en base deux ? :p). Pour stocker plus d'informations, il sera nécessaire d'utiliser plusieurs octets.

3.2.1 Adresse mémoire

Néanmoins, il est bien beau de stocker des données en mémoire, encore faut-il pouvoir remettre la main dessus.

Dans cette optique, chaque octet de la mémoire vive se voit attribuer un nombre unique, **une adresse**, qui va permettre de le sélectionner et de l'identifier parmi tous les autres. Imaginez la mémoire vive de l'ordinateur comme une immense armoire, qui contiendrait beaucoup de tiroirs (les cases mémoires) pouvant chacun contenir un octet. Chaque tiroir se voit attribuer un numéro pour le reconnaître parmi tous les autres. Nous pourrions ainsi demander quel est le contenu du tiroir numéro 27. Pour la mémoire, c'est pareil. Chaque case mémoire a un numéro : son adresse.

Adresse	Contenu mémoire
0	11101010
1	01111111
2	00000000
3	01010101
4	10101010
5	00000000

TABLE 3.2 – Adresse mémoire

En fait, vous pouvez comparer une adresse à un numéro de téléphone : chacun de vos correspondants a un numéro de téléphone et vous savez que pour appeler telle personne, vous devez composer tel numéro. Les adresses mémoires fonctionnent exactement de la même façon !

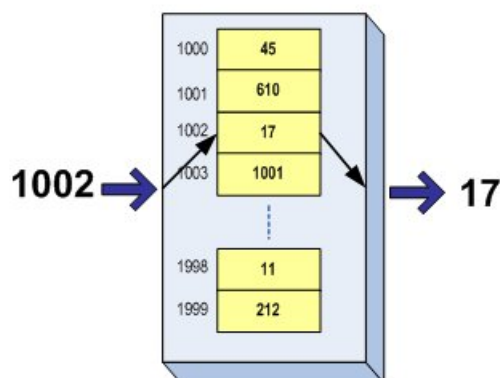


FIGURE 3.1 – Exemple : on demande à notre mémoire de sélectionner la case mémoire d’adresse 1002 et on récupère son contenu (ici, 17).



Plus généralement, toutes les mémoires disposent d’un mécanisme similaire pour retrouver les données. Aussi, vous entendrez souvent le terme de **référence** qui désigne un moyen (comme une adresse) permettant de localiser une donnée. Il s’agit simplement d’une notion plus générale.

3.2.2 Les variables

Tout cela est bien sympathique, mais manipuler explicitement des références (des adresses si vous préférez) est un vrai calvaire, de même que de s’évertuer à calculer en base deux. Heureusement pour nous, les langages de programmation (et notamment le C), se chargent d’effectuer les conversions pour nous et remplacent les références par des variables.

Une variable correspondra à une portion de mémoire, appelée **objet**, à laquelle nous donnerons un nom. Ce nom permettra d’identifier notre variable, tout comme une référence permet d’identifier une portion de mémoire parmi toutes les autres. Nous allons ainsi pouvoir nommer les données que nous manipulons, chacun de ces noms étant remplacés lors de la compilation par une référence (le plus souvent une adresse).

3.3 Déclarer une variable

Entrons maintenant dans le vif du sujet en apprenant à déclarer nos variables. Tout d'abord, sachez qu'une variable est constituée de deux éléments obligatoires :

- un **type** ;
- un **identificateur** qui est en gros le « nom » de la variable.

Le type d'une variable permet d'indiquer ce qui y sera stocké, par exemple : un caractère, un nombre entier, un nombre à virgule (ou nombre **flottant**), etc. Pour préciser le type d'une variable, il est nécessaire d'utiliser un mot-clé spécifique (il y en a donc un pour chaque type).

Une fois que nous avons décidé du nom et du type de notre variable, nous pouvons la créer (on dit aussi la déclarer) comme suit.

```
1 type identificateur;
```

En clair, il suffit de placer un mot-clé indiquant le type de la variable et de placer le nom qu'on lui a choisi immédiatement après.

! Faites bien attention au point-virgule à la fin !

3.3.1 Les types

Comme dit précédemment, un type permet d'indiquer au compilateur quel genre de données nous souhaitons stocker. Ce type va permettre de préciser :

- toutes les valeurs que peut prendre la variable ;
- les opérations qu'il est possible d'effectuer avec (il n'est par exemple pas possible de réaliser une division entière avec un nombre flottant, nous y reviendrons).

Définir le type d'une variable permet donc de préciser son contenu potentiel et ce que nous pouvons faire avec. Le langage C fournit sept¹ types de base.

Type	Sert à stocker
char	un caractère ou un entier
short int	un entier
int	un entier
long int	un entier
float	un flottant
double	un flottant
long double	un flottant

TABLE 3.3 – Les 7 types de base

Les types `short int`, `int` et `long int` servent tous à stocker des nombres entiers qui peuvent prendre des valeurs positives, négatives, ou nulles. On dit qu'il s'agit de types signés (car il peut comporter un signe). Pour ces trois types, il existe un type équivalent dit **non signé**. Un type entier non signé est un type entier qui n'accepte que des valeurs positives ou nulles : il ne peut pas stocker de valeurs négatives. Pour déclarer des variables d'un type non signé, il vous suffit de faire précéder le nom du type entier du mot-clé `unsigned`.

Le type `char` peut lui aussi servir à stocker des nombres. Il sert surtout au stockage de caractères, mais ces derniers étant stockés dans l'ordinateur sous forme de nombres, il est possible de stocker des nombres dans un `char`. Le seul problème, c'est que ce type peut être signé ou non

1. Depuis la norme C99, un type entier supplémentaire a été ajouté : le type `long long int`. Ses bornes garanties par la norme sont comprises entre -9 223 372 036 854 775 807 et 9 223 372 036 854 775 807 s'il est signé et entre 0 et 18 446 744 073 709 551 615 s'il est non signé.

signé de base suivant les compilateurs. Pour éviter les ennuis, spécifiez ce que vous souhaitez lors de la déclaration : non signé (`unsigned char`) ou signé (`signed char`).



En cas de manque d'information concernant le type lors d'une déclaration, c'est le type `int` qui sera utilisé. Ainsi, `long` et `short` sont respectivement des raccourcis pour `long int` et `short int`. De même, le mot-clé `unsigned` seul signifie `unsigned int`.

Capacité d'un type

Tous les types stockant des nombres ont des bornes, c'est-à-dire une limite aux valeurs qu'ils peuvent stocker. En effet, le nombre de multipliants occupés par une variable est limité suivant son type. En conséquence, il n'est pas possible de mettre tous les nombres possibles dans une variable de type `int`, `float`, ou `double`. Il y aura *toujours* une valeur minimale et une valeur maximale. Ces limites sont les suivantes.

Type	Minimum	Maximum
<code>signed char</code>	-127	127
<code>unsigned char</code>	0	255
<code>short</code>	-32 767	32 767
<code>unsigned short</code>	0	65 535
<code>int</code>	-32 767	32 767
<code>unsigned int</code>	0	65 535
<code>long</code>	-2 147 483 647	2 147 483 647
<code>unsigned long</code>	0	4 294 967 295
<code>float</code>	-1×10^{37}	1×10^{37}
<code>double</code>	-1×10^{37}	1×10^{37}
<code>long double</code>	-1×10^{37}	1×10^{37}

TABLE 3.4 – Les limites des types

Si vous regardez bien ce tableau, vous remarquez que certains types ont des bornes identiques. En vérité, les valeurs présentées ci-dessus sont les minimums garantis par la norme²

Taille d'un type

Peut-être vous êtes vous demandés pourquoi il existe autant de types différents. La réponse est toute simple : la taille des mémoires était très limitée à l'époque où le langage C a été créé. En effet, le [PDP-11](#) sur lequel le C a été conçu ne possédait que 24Ko de mémoire (pour comparaison, une calculatrice TI-Nspire possède 100Mo de mémoire, soit environ 4000 fois plus). Il fallait donc l'économiser au maximum en choisissant le type le plus petit possible. Cette taille dépend des machines, mais de manière générale, vous pouvez retenir les deux suites d'inégalités suivantes : `char` `short` `int` `long` et `float` `double` `long double`.

Aujourd'hui ce n'est plus un problème, il n'est pas nécessaire de se casser la tête sur quel type choisir (excepté si vous voulez programmer pour de petits appareils où la mémoire est plus petite). En pratique, nous utiliserons surtout `char` pour les caractères, `int` ou `long` pour les entiers et `double` pour les flottants.

3.3.2 Les identificateurs

Maintenant que nous avons vu les types, parlons des identificateurs. Comme dit précédemment, un identificateur est un nom donné à une variable pour la différencier de toutes les autres.

2. Programming Language C, X3J11/88-090, § 2.2.4.2, Numerical limits

Et ce nom, c'est au programmeur de le choisir. Cependant, il y a quelques limitations à ce choix.

- seuls les 26 lettres de l'alphabet latin (majuscules ou minuscules), le trait de soulignement « `_` » (*underscore* en anglais) et les chiffres sont acceptés. Pas d'accents, pas de ponctuation ni d'espaces ;
- un identificateur ne peut pas commencer par un chiffre ;
- les mots-clés ne peuvent pas servir à identifier une variable ; il s'agit de :

```

1 auto    double int    struct
2 break   else    long   switch
3 case    enum    register typedef
4 char    extern  return  union
5 const   float   short   unsigned
6 continue for    signed  void
7 default goto    sizeof  volatile
8 do      if      static  while

```

- deux variables ne peuvent avoir le même identificateur (le même nom). Il y a parfois quelques exceptions, mais cela n'est pas pour tout de suite ;
- les identificateurs peuvent être aussi longs que l'on désire, toutefois le compilateur ne tiendra compte que des 31 premiers caractères.

Voici quelques exemples pour bien comprendre.

Identificateur correct	Identificateur incorrect	Raison
variable	Nom de variable	Espaces interdits
nombre_de_vie	1nombre_de_vie	Commence par un chiffre
test	test !	Caractère « ! » interdit
un_dernier_pour_la_route1	<code>continue</code>	Mot-clé réservé par le langage

TABLE 3.5 – Les identificateurs

À noter que le C fait la différence entre les majuscules et les minuscules (on dit qu'il **respecte la casse**). Ainsi les trois identificateurs suivants sont différents.

```

1 variable
2 Variable
3 VaRiAbLe

```

3.3.3 Déclaration et initialisation

Maintenant que nous connaissons toutes les bases, entraînons-nous à déclarer quelques variables.

```

1 int main(void)
2 {
3     double taille;
4     unsigned int age;
5     char caractere;
6     short petite_valeur;
7
8     return 0;
9 }

```

Il est possible de déclarer plusieurs variables **de même type** sur une même ligne, en séparant leurs noms par une virgule.

```

1 int age, taille, nombre;

```


Ceci permet de regrouper les déclarations suivant les rapports que les variables ont entre elles.

```
1 int annee, mois, jour;
2 int age, taille;
3 int x, y, z;
```

Initialisation

En plus de déclarer une variable, il est possible de **l'initialiser**, c'est-à-dire de lui attribuer une valeur. La syntaxe est la suivante.

```
1 type identificateur = valeur;
```

Ou comme ceci s'il s'agit d'un caractère.

```
1 char identificateur = "lettre";
```

Quelques exemples d'initialisations de variables.

```
1 unsigned int age = 25;
2 short petite_valeur = 1;
3 const long abc = 314159265;
4 char caractere = 'h';
```



Notez qu'une déclaration comporte le mot-clé `const`. Celui-ci permet de préciser qu'une variable ne pourra pas être modifiée par la suite. Ceci peut être utile pour stocker une valeur qui ne changera jamais (comme la constante π qui vaut toujours 3,14159265).



Les déclarations doivent toujours se situer en début de bloc, c'est-à-dire juste après une accolade ouvrante.

Initialisation des nombres flottants

En notation simple

Petite précision concernant la manière d'initialiser une variable de type flottant : celles-ci étant faites pour contenir des nombres à virgule, à l'initialisation, il est nécessaire de placer cette « virgule ». Toutefois, cette dernière est représentée par un point.

```
1 const double pi = 3.14;
```

Ceci vient du fait que le C est une invention américaine et que les anglophones utilisent le point à la place de la virgule.

Notez qu'il est important de bien placer ce point, *même si vous voulez stocker un nombre entier*. Par exemple, vous ne devez pas écrire `double a = 5` mais `double a = 5.` (certains préfèrent `double a = 5.0`, cela revient au même). Si vous ne le faites pas, vous risquez d'avoir quelques problèmes.

En notation scientifique

Par ailleurs, sachez qu'il est également possible de représenter un nombre flottant à l'aide de la notation scientifique, c'est-à-dire sous la forme d'un nombre décimal et d'une puissance de dix. Cela se traduit par un nombre flottant en notation simple (comme ci-dessus) suivi de la lettre « e » ou « E » et d'un exposant *entier*.

```
1 double f = 1E-1; /* 1x10^-1 = 0.1 */
```

3.3.4 Affectation

Nous savons donc déclarer (créer) nos variables et les initialiser (leur donner une valeur à la création). Il ne nous reste plus qu'à voir la dernière manipulation possible : **l'affectation**. Celle-ci permet de modifier la valeur contenue dans une variable pour la remplacer par une autre valeur.

Il va de soi que cette affectation n'est possible que pour les variables qui ne sont pas déclarées avec le mot-clé `const` puisque, par définition, de telles variables sont constantes et ne peuvent voir leur contenu modifié.

Pour faire une affectation, il suffit d'opérer ainsi.

```
1 identificateur = nouvelle_valeur;
```

Nous voyons que la syntaxe est similaire à celle d'une déclaration avec initialisation. La seule différence, c'est que nous n'avons pas à préciser le type. Celui-ci est en effet fixé une fois pour toute lors de la déclaration de notre variable. Aussi, il n'est pas nécessaire de le préciser à nouveau lors d'une affectation.

```
1 age = 30;  
2 taille = 177.5;  
3 petite_valeur = 2
```

Notez qu'il n'y a aucune limite au nombre d'affectations, comme le démontre l'exemple ci-dessous.

```
1 petite_valeur = 2;  
2 petite_valeur = 4;  
3 petite_valeur = 8;  
4 petite_valeur = 16;  
5 petite_valeur = 8;  
6 petite_valeur = 4;  
7 petite_valeur = 2;
```

À chaque affectation, la variable va prendre une nouvelle valeur. Par contre, ne mettez pas le type quand vous voulez changer la valeur, sinon vous aurez le droit à une belle erreur du type « *redefinition of 'nom_de_votre_variable'* » car vous aurez créé deux variables avec le même identificateur !

Le code suivant est donc incorrect.

```
1 int age =15;  
2 int age =20;
```

Si vous exécutez tous ces codes, vous verrez qu'ils n'affichent toujours rien et c'est normal puisque nous n'avons pas demandé à notre ordinateur d'afficher quoique ce soit. Nous apprendrons comment faire au chapitre suivant.



Il n'y a pas de valeur par défaut en C. Aussi, sans initialisation ou affectation, la valeur d'une variable est indéterminée ! Veillez donc à ce que vos variables aient une valeur connue avant de les utiliser !

3.4 Les représentations octales et hexadécimales

Pour terminer ce chapitre, nous vous proposons un petit aparté sur deux représentations particulières : la représentation octale et la représentation hexadécimale.

Nous avons déjà vu la représentation binaire au début de ce chapitre, les représentations octale et hexadécimale obéissent au même schéma : au lieu d'utiliser dix chiffres pour représenter un nombre, nous en utilisons respectivement huit ou seize.



Seize chiffres ?! Mais... Je n'en connais que dix moi !

La représentation hexadécimale est un peu déroutante de prime abord, celle-ci ajoute six chiffres (en fait, six lettres) : a, b, c, d, e et f. Pour vous aider, voici un tableau présentant les nombres de 1 à 16 en binaire, octal, décimal et hexadécimal.

Binaire	Octal	Décimal	Hexadécimal
00001	1	1	1
00010	2	2	2
00011	3	3	3
00100	4	4	4
00101	5	5	5
00110	6	6	6
00111	7	7	7
01000	10	8	8
01100	14	12	c
01101	15	13	d
01110	16	14	e
01111	17	15	f
10000	20	16	10

TABLE 3.6 – Les nombres de 1 à 16 en binaire, octal, décimal et hexadécimal



Notez que dix dans n'importe quelle base équivaut à cette base.



Quel est l'intérêt de ces deux bases exactement ?

L'avantage des représentations octale et hexadécimale est qu'il est facilement possible de les convertir en binaire contrairement à la représentation décimale. En effet, un chiffre en base huit ou en base seize peut-être facilement traduit, respectivement, en trois ou quatre *bits*.

Prenons l'exemple du nombre 35 qui donne 43 en octal et 23 en hexadécimal. Nous pouvons nous focaliser sur chaque chiffre un à un pour obtenir la représentation binaire. Ainsi, du côté de la représentation octale, 4 donne `100` et 3 `011`, ce qui nous donne finalement `00100011`. De même, pour la représentation hexadécimale, 2 nous donne `0010` et 3 `0011` et nous obtenons `00100011`. Il n'est pas possible de faire pareil en décimal.

En résumé, les représentations octale et hexadécimale permettent de représenter un nombre binaire de manière plus concise et plus lisible.

3.5 Constantes octales et hexadécimales

Il vous est possible de préciser la base d'une constante entière en utilisant des préfixes. Ces préfixes sont `0` pour les constantes octales et `0x` ou `0X` pour les constantes hexadécimales.

```
1 long a = 65535; /* En décimal */  
2 int b = 0777; /* En octal */  
3 short c = 0xFF; /* En hexadécimal */
```



Les lettres utilisées pour la représentation hexadécimale peuvent être aussi bien écrites en minuscule qu'en majuscule.



Il n'est pas possible d'utiliser une constante en base deux ?

Malheureusement, non, le langage C ne permet pas d'utiliser de telles constantes.

Voilà, c'est la fin de ce chapitre. Nous avons vu beaucoup de choses, n'hésitez pas à potasser pour bien assimiler tout ça. Les variables sont vraiment la base de la programmation, aussi il nécessaire de bien les comprendre. Rendez-vous au prochain chapitre qui sera très intéressant : vous pourrez par exemple demander l'âge de l'utilisateur pour ensuite l'afficher !

Manipulations basiques des entrées/sorties

Durant l'exécution d'un programme, le processeur a besoin de communiquer avec le reste du matériel. Ces échanges d'informations sont les **entrées** et les **sorties** (ou *input* et *output* pour les anglophones), souvent abrégées E/S (ou I/O par les anglophones).

Les entrées permettent de recevoir une donnée en provenance de certains périphériques. Les données fournies par ces entrées peuvent être une information envoyée par le disque dur, la carte réseau, le clavier, la souris, un CD, un écran tactile, bref par n'importe quel périphérique. Par exemple, notre clavier va transmettre des informations sur les touches enfoncées au processeur : notre clavier est donc une entrée.

À l'inverse, les sorties vont transmettre des données vers ces périphériques. On pourrait citer l'exemple de l'écran : notre ordinateur lui envoie des informations pour qu'elles soient affichées.

Dans ce chapitre, nous allons apprendre différentes fonctions fournies par le langage C qui vont nous permettre d'envoyer des informations vers nos sorties et d'en recevoir depuis nos entrées. Vous saurez ainsi comment demander à un utilisateur de fournir une information au clavier et comment afficher quelque chose sur la console.

4.1 Les sorties

Intéressons-nous dans un premier temps aux sorties. Afin d'afficher du texte, nous avons besoin d'une **fonction**.

Une fonction est un morceau de code qui a un but, une *fonction* particulière et qui peut être **appelée** à l'aide d'une référence, le plus souvent le nom de cette fonction (comme pour une variable, finalement). En l'occurrence, nous allons utiliser une fonction qui a pour objectif d'afficher du texte dans la console : la fonction `printf()`.

4.1.1 Première approche

Un exemple valant mieux qu'un long discours, voici un premier exemple.

```

1  #include<stdio.h>
2
3  int main(void)
4  {
5      printf("Bonjour tout le monde !\n");
6      return 0;
7  }
```

```

1  Bonjour tout le monde !
```

Deux remarques au sujet de ce code.

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

Il s'agit d'une **directive du préprocesseur**, facilement reconnaissable car elles commencent toutes par le symbole `#`. Celle-ci sert à inclure un fichier (« `stdio.h` ») qui contient les références de différentes fonctions d'entrée et sortie (« *stdio* » est une abbréviation pour « *Standard input output* », soit « Entrée-sortie standard »).

Un fichier se terminant par l'extension « `.h` » est appelé un **fichier d'en-tête** (*header* en anglais) et fait partie avec d'autre d'un ensemble plus large appelée la **bibliothèque standard** (« *standard* » car elle est prévue par la norme¹).

```
1 printf("Bonjour tout le monde !\n");
```

Ici, nous **appelons** la fonction `printf()` (un appel de fonction est toujours suivi d'un groupe de parenthèses) avec comme **argument** (ce qu'il y a entre les parenthèses de l'appel) un texte (il s'agit plus précisément d'une **chaîne de caractères**, qui est toujours comprise entre deux guillemets double). Le `\n` est un caractère spécial qui représente un retour à la ligne, cela est plus commode pour l'affichage.

Le reste devrait vous être familier.

4.1.2 Les formats

Bien, nous savons maintenant afficher une phrase, mais ce serait quand même mieux de pouvoir voir les valeurs de nos variables. Comment faire ? *Hé* bien, pour y parvenir, la fonction `printf()` met à notre disposition des **formats**. Ceux-ci sont en fait des sortes de repères au sein d'un texte qui indique à `printf()` que la valeur d'une variable est attendue à cet endroit. Voici un exemple pour une variable de type `int`.

```
1 #include<stdio.h>}
2
3 int main(void)
4 {
5     int variable = 20 ;
6
7     printf("%d\n", variable);
8     return 0 ;
9 }
```

```
1 20
```

Nous pouvons voir que le texte de l'exemple précédent a été remplacé par `%d`, seul le `\n` a été conservé. Un format commence toujours par le symbole `%` et est suivi par une ou plusieurs lettres qui indiquent le type de données que nous souhaitons voir afficher. Cette suite de lettre est appelée un **indicateur de conversion**. En voici une liste non exhaustive²

1. Programming Language C, X3J11/88-090, § 4, Library.

2. Pour le type `long long` introduit en C99, les indicateurs de conversions sont `lld` et `llu`. Il en va de même pour `scanf()`.

Type	Indicateur(s) de conversion
char	c, d (ou i)
short	d (ou i)
int	d (ou i)
long	ld (ou li)
unsigned char	u, x (ou X) ou o
unsigned short	u, x (ou X) ou o
unsigned int	u, x (ou X) ou o
unsigned long	lu, lx (ou LX) ou lo
float	f, e (ou E) ou g (ou G)
double	f, e (ou E) ou g (ou G)
long double	Lf, Le (ou LE) ou Lg (ou LG)

TABLE 4.1 – Liste des indicateurs de conversion



Notez que pour le type `char`, l'indicateur est soit `c`, soit `d` (ou `i`). Cela dépend si vous l'utilisez pour contenir un caractère ou un entier. Également, notez que les indicateurs de conversions sont identiques pour les types `char` (s'il stocke un entier), `short` et `int` (pareil pour leurs équivalents non signés) ainsi que pour les types `float` et `double`.

Les indicateurs `x`, `X` et `o` permettent d'afficher un nombre en représentation hexadécimale ou octale (l'indicateur `x` affiche les lettres en minuscules alors que l'indicateur `X` les affiche en majuscules).

Les indicateurs `f`, `e` et `g` permettent quant à eux d'afficher un nombre flottant. L'indicateur `f` affiche un nombre en notation simple avec, par défaut, six décimales ; l'indicateur `e` affiche un nombre flottant en notation scientifique (l'indicateur `e` utilise la lettre « e » avant l'exposant alors que l'indicateur « E » emploie la lettre « E ») et l'indicateur `g` choisi quant à lui entre les deux notations précédentes suivant le nombre fourni et supprime la partie fractionnaire si elle est nulle de sorte que l'écriture soit concise (la différence entre les indicateurs `g` et `G` est identique à celle entre les indicateurs `e` et `E`).

Allez, un petit exemple pour reprendre tout cela et retravailler le chapitre précédent par la même occasion.

```

1  #include<stdio.h>
2
3  int main(void )
4  {
5      char z = 'z';
6      char a = 10;
7      unsigned short b = 20;
8      int c = 30;
9      long d = 40;
10     float e = 50 .;
11     double f = 60.0;
12     long double g = 70.0;
13
14     printf("%c\n", z);
15     printf("%d\n", a);
16     printf("%u\n", b);
17     printf("%o\n", b);
18     printf("%x\n", b);
19     printf("%d\n", c);
20     printf("%li\n", d);
21     printf("%f\n", e);
22     printf(" %e\n", f);
23     g = 80.0;
24     printf(" %Lg\n", g);
25     return 0;
26 }
```



```

1  z
2  10
3  20
4  24
5  14
6  30
7  40
8  50.000000
9  6.000000e+01
10 80

```



Si vous souhaitez afficher le caractère `%` vous devez le doubler : `%%`.

4.1.3 Précision des nombres flottants

Vous avez peut-être remarqué que lorsqu'un flottant est affiché avec le format `f`, il y a un certain nombre de zéros qui suivent (par défaut six) et ce, peu importe qu'ils soient utiles ou non. Afin d'en supprimer certains, vous pouvez spécifier une **précision**. Celle-ci correspond au nombre de chiffre suivant la virgule qui seront affichés. Elle prend la forme d'un point suivi par un nombre : la quantité de chiffres qu'il y aura derrière la virgule.

```

1 double x = 42.42734;
2
3 printf("%.2f", x);

```

```

1 42.43

```

4.1.4 Les caractères spéciaux

Dans certains cas, nous souhaitons obtenir un résultat à l'affichage (saut de ligne, une tabulation, un retour chariot, etc.). Cependant, ils ne sont pas particulièrement pratiques à insérer dans une chaîne de caractères. Aussi, le C nous permet de le faire en utilisant une **séquence d'échappement**. Il s'agit en fait d'une suite de caractères commençant par le symbole `\` et suivie d'une lettre. En voici une liste non exhaustive.

Séquence d'échappement	Signification
<code>\</code>	Caractère d'appel
<code>\b</code>	Espacement arrière
<code>\f</code>	Saut de page
<code>\n</code>	Saut de ligne
<code>\r</code>	Retour chariot
<code>\t</code>	Tabulation horizontale
<code>\v</code>	Tabulation verticale
<code>\"</code>	Le symbole « <code>"</code> »
<code>\\</code>	Le symbole « <code>\</code> » lui-même

TABLE 4.2 – Les nombres de 1 à 16 en binaire, octal, décimal et hexadécimal.

En général, vous n'utiliserez que le saut de ligne, la tabulation horizontale et de temps à autre le retour chariot, les autres n'ont quasiment plus d'intérêt. Un petit exemple pour illustrer leurs effets.

```

1 #include<stdio.h>
2

```

```

3  int main(void)
4  {
5      printf( "Quelques sauts de ligne\n\n\n");
6      printf( " \tIl y a une tabulation avant moi !\n");
7      printf( "Je voulais dire que... \r ");
8      printf( "Hey ! Vous pourriez me laisser parler !\n");
9      return 0 ;
10 }

```

```

1  Quelques sauts de ligne
2
3
4      Il y a une tabulation avant moi !
5  Hey ! Vous pourriez me laisser parler !

```



Le retour chariot provoque un retour au début de la ligne courante. Ainsi, il est possible d'écrire par-dessus un texte affiché.

4.1.5 Sur plusieurs lignes

Notez qu'il est possible d'écrire un long texte sans appeler plusieurs fois la fonction `printf()`. Pour ce faire, il suffit de le diviser en plusieurs chaînes de caractères.

```

1  #include<stdio.h>
2
3  int main(void )
4  {
5      printf("Texte écrit sur plusieurs "
6             "lignes dans le code source "
7             "mais sur une seule dans la console.\n");
8      return 0;
9  }

```

```

1  Texte écrit sur plusieurs lignes dans le code source mais sur une seule dans la console.

```

4.2 Interagir avec l'utilisateur

Maintenant que nous savons déclarer, utiliser et même afficher des variables, nous sommes fin prêts pour interagir avec l'utilisateur. En effet, jusqu'à maintenant, nous nous sommes contentés d'afficher des informations. Nous allons à présent voir comment en récupérer grâce à la fonction `scanf()`, dont l'utilisation est assez semblable à `printf()`.

```

1  #include<stdio.h>
2
3  int main(void)
4  {
5      int age;
6
7      printf("Quel âge avez-vous ? ");
8      scanf("%d", &age);
9      printf("Vous avez %d an(s)\n ", age);
10     return 0 ;
11 }

```

```

1  Quel age avez-vous ? 15
2  Vous avez 15 an(s).

```

Comme vous le voyez, l'appel à `scanf()` ressemble très fort à celui de `printf()` mise à part l'absence du caractère spécial `\n` (qui n'a pas d'intérêt puisque nous récupérons des informations) et le symbole `&`.

À son sujet, souvenez-vous de la brève explication sur la mémoire au début du chapitre précédent. Celle-ci fonctionne comme une armoire avec des tiroirs (les adresses mémoires) et des objets dans ces tiroirs (nos variables). La fonction `scanf()` a besoin de connaître l'emplacement en mémoire de nos variables afin de les modifier. Afin d'effectuer cette opération, nous utilisons le symbole `&` (qui est en fait un opérateur que nous verrons en détail plus tard). Ce concept de transmission d'adresses mémoires est un petit peu difficile à comprendre au début, mais ne vous inquiétez pas, vous aurez l'occasion de bien revoir tout cela en profondeur dans le chapitre traitant des pointeurs.

Ici, `scanf()` attend patiemment que l'utilisateur saisisse un nombre au clavier afin de modifier la valeur de la variable `age`. On dit que c'est une **fonction bloquante**, car elle suspend l'exécution du programme tant que l'utilisateur n'a rien entré.

Pour ce qui est des indicateurs de conversions, ils sont un peu différents de ceux de `printf()`.

Type	Indicateur(s) de conversion
char	c
short	hd ou hi
int	d ou i
long	ld ou li
unsigned short	hu, hx ou ho
unsigned int	u, x ou o
unsigned long	lu, lx ou lo
float	f
double	lf
long double	Lf



Faites bien attention aux différences ! Si vous utilisez le mauvais format, le résultat ne sera pas celui que vous attendez. Les changements concernent les types `char`, `short` et `double`.



Notez que l'indicateur `c` ne peut être employé que pour récupérer un caractère et non un nombre.



Remarquez également qu'il n'y a plus qu'un seul indicateur pour récupérer un nombre hexadécimal : `x` (l'utilisation de lettres majuscules ou minuscules n'a pas d'importance).

En passant, sachez qu'il est possible de lire plusieurs entrées en même temps, par exemple comme ceci.

```
1 int x, y;
2
3 scanf("%d %d", &x, &y);
4 printf("x = %d | y = %d \n", x, y);
```

L'utilisateur a deux possibilités, soit insérer un (ou plusieurs) espace(s) entre les valeurs, soit insérer un (ou plusieurs) retour(s) à la ligne entre les valeurs.

```
1 14
2 6
3 x = 14 | y = 6
```

```
1 14 6
2 x = 14 | y = 6
```

La fonction `scanf()` est en apparence simple (oui, *en apparence*), mais son utilisation peut devenir très complexe lorsqu'il est nécessaire de vérifier les entrées de l'utilisateur (entre autres). Cependant, à votre niveau, vous ne pouvez pas encore effectuer de telles vérifications. Ce n'est toutefois pas très grave, nous verrons cela en temps voulu. ;)

Maintenant, vous êtes capable de communiquer avec l'utilisateur. Cependant, nos actions sont encore un peu limitées. Nous verrons dans les prochains chapitres comment mieux interagir avec l'utilisateur.

Les opérations mathématiques

Nous savons désormais déclarer, affecter et initialiser une variable, mais que diriez-vous d'apprendre à réaliser des opérations dessus ? Il est en effet possible de réaliser des calculs sur nos variables, comme les additionner, les diviser voire même des opérations plus complexes. C'est le but de cette sous-partie. Nous allons donc enfin transformer notre ordinateur en grosse calculatrice programmable !

5.1 Les opérations mathématiques de base

Jusqu'à présent, nous nous sommes contenté d'afficher du texte et de manipuler très légèrement les variables. Voyons à présent comment nous pouvons réaliser quelques opérations de base. Le langage C nous permet de réaliser cinq :

- l'addition (opérateur `+`) ;
- la soustraction (opérateur `-`) ;
- la multiplication (opérateur `*`) ;
- la division (opérateur `/`) ;
- le modulo (opérateur `%`).



Le langage C fournit bien entendu d'autres fonctions mathématiques et d'autres opérateurs, mais il est encore trop tôt pour vous les présenter.

5.2 Division et modulo

Les quatre premières opérations vous sont connues depuis l'école primaire. Cependant, une chose importante doit être précisée concernant la division : quand les deux nombres manipulés sont des entiers, il s'agit d'une division entière. Autrement dit, le quotient sera un entier et il peut y avoir un reste. Par exemple, $15 \div 6$, ne donnera pas 2,5 (division réelle), mais un quotient de 2, avec un reste de 3.

```
1 printf("15 / 6 = %d\n", 15 / 6);
```

```
1 15 / 6 = 2
```

Le modulo est un peu le complément de la division entière : au lieu de donner le quotient, il renvoie le reste d'une division euclidienne. Par exemple, le modulo de 15 par 6 est 3, car $15 = 2 \times 6 + 3$.

```
1 printf("15 % 6 = %d\n", 15 % 6);
```

```
1 15 % 6 = 3
```

Avec des flottants, la division se comporte autrement et n'est pas une division avec reste. La division de deux flottants donnera un résultat « exact », avec potentiellement plusieurs chiffres après la virgule.

```
1 printf("15 / 6 = %f\n", 15. / 6.); /* En C, ce n'est pas la même chose que 15 / 6 */
```

```
1 15 / 6 = 2.500000
```

5.3 Utilisation

Il est possible d'affecter le résultat d'une expression contenant des calculs à une variable, comme lorsque nous les utilisons comme argument de `printf()`.

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     int somme = 5 + 3;
6
7     printf("5 + 3 = %d\n", somme);
8     return 0;
9 }
```

```
1 5 + 3 = 8
```

Toute opération peut manipuler :

- des constantes ;
- des variables ;
- les deux à la fois.

Voici un exemple avec des constantes.

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("2 + 3 = %d\n", 2 + 3);
6     printf("8 - 12 = %d\n", 8 - 12);
7     printf("6 x 7 = %d\n", 6 * 7);
8     printf("11 % 4 = %d\n", 11 % 4);
9     return 0;
10 }
```

```
1 2 + 3 = 5
2 8 - 12 = -4
3 6 x 7 = 42
4 11 % 4 = 3
```

Un autre avec des variables.

```
1 int a = 5;
2 int b = 3;
3 int somme = a + b;
4
5 printf("%d + %d = %d\n", a, b, somme);
```

```
1 5 + 3 = 8
```

Et enfin, un exemple qui mélange variables et constantes.

```
1 int a = 5;
2 int b = 65;
3
4 printf("%d\n", b / a * 2 + 7 % 2);
```

```
1 27
```

5.4 La priorité des opérateurs

Dans l'exemple précédent, nous avons utilisé plusieurs opérations dans une même ligne de code, une même expression. Dans ces cas là, faites attention à la **priorité des opérateurs** ! Comme en mathématiques, certains opérateurs passent avant d'autres : les opérateurs `* \%` ont une priorité supérieure par rapport aux opérateurs `+ -`.

Dans le code ci-dessous, c'est `c * 4` qui sera exécuté d'abord, puis `b` sera ajouté au résultat. Faites donc attention sous peine d'avoir de mauvaises surprises. Dans le doute, ajoutez des parenthèses.

```
1 c a = b + c * 4;}
```

5.5 Les expressions

Il est possible de combiner opérateur, variables et constantes pour former des **expressions**, des lignes de code qui sont évaluées et produisent un résultat. Les lignes de code suivantes sont toutes des expressions.

```
1 "Bonjour !"
2 2 + 3
3 10 > 2
```

Généralement, une expression ne peut être écrite seule et doit faire partie d'une **instruction**. La frontière entre instruction et expression est assez floue puisqu'une instruction peut être composée de nombreuses expressions. Le code ci-dessous est un exemple d'instruction qui est *quasi* une expression (on parle d'**expression-instruction**).

```
1 x = 2 + 3;
```

Nous donnons en effet un ordre à l'ordinateur (« affecte la valeur `2 + 3` à `x` »), mais c'est aussi une expression qui produit la valeur 5 comme résultat. Vous verrez qu'en C, la majorité des lignes de code sont des instructions-expressions. C'est ce qui est appelé la **programmation impérative**. C'est le choix des concepteurs du langage, mais ce n'est pas la seule possibilité (il en existe d'autres, mais ça ne nous concerne pas en tant qu'utilisateurs du C).

5.6 Type d'une expression

Vous avez sans doute remarqué que nous avons utilisé directement des expressions (`2 + 3` par exemple) comme argument de la fonction `printf()`. Rien de bien surprenant me direz-vous... À un détail près : quel indicateur de format doit-on utiliser ? Autrement dit, quel est le type d'une expression ? D'ailleurs, ont-elles un type ?

Hé bien, oui. Tout comme les variables, les expressions ont un type. Ce dernier dépend toutefois du type des éléments qui la compose. En l'occurrence, une constante entière comme `2` ou `3` est par défaut de type `int`. Le résultat d'une somme, par exemple, sera donc également de type `int`. Les constantes flottantes comme `5.` ou `78.0` sont, elles, de type `double` et le résultat d'une opération sera alors de type `double`.

? D'accord, mais si j'additionne un `int` avec un `double`, cela me donne quoi ?

Heureusement pour nous, la norme¹ a prévu ces différents cas et a fixé des règles :

- si un opérande est de type `long double`, le résultat est de type `long double` ; si non
- si un opérande est de type `double`, le résultat est de type `double` ; si non
- si un opérande est de type `float`, le résultat est de type `float` ; si non
- si un opérande est de type `unsigned long`, le résultat est de type `unsigned long` ; si non
- si un opérande est de type `long`, le résultat est de type `long` ; si non
- si un opérande est de type `unsigned int`, le résultat est de type `unsigned int` ; si non
- le résultat est de type `int`.

5.6.1 Suffixes

? Heu... D'accord, mais vous venez de dire que les constantes entières étaient de type `int` et que les constantes flottantes étaient de type `double`. Du coup, je fais comment pour obtenir une constante d'un autre type ?

À l'aide d'un suffixe. Celui-ci se place à la fin de la constante et permet de modifier son type. En voici la liste complète².

Type	Suffixe
u ou U	unsigned
l ou L	long
f ou F	float
l ou L	long double

i Notez que les suffixes `L` (ou `l`) et `U` (ou `u`) peuvent être combinés.

5.6.2 Exemple

Allez, un petit récapitulatif.

```

1 #include <stdio.h>
2
3
4 int main(void)
5 {
```

1. Programming Language C, X3J11/88-090, § 3.2.1.5, Usual arithmetic conversions.

2. Pour le type `long long` introduit en C99, le suffixe est `LL` ou `ll`

```

6  /* long double + int = long double */
7  printf("78.56 + 5 = %Lf\n", 78.56L + 5);
8
9  /* long + double = double */
10 printf("5678 + 2.2 = %f\n", 5678L + 2.2);
11
12 /* long + unsigned long = unsigned long */
13 printf("2 + 5 = %lu\n", 2L + 5UL);
14
15 /* long + int = long */
16 printf("1 + 1 = %ld\n", 1L + 1);
17 return 0;
18 }

```

```

1  78.56 + 5 = 83.560000
2  5678 + 2.2 = 5680.200000
3  2 + 5 = 7
4  1 + 1 = 2

```



Nous vous conseillons d'opter pour les lettres majuscules qui ont l'avantage d'être plus lisibles.

5.7 Conversions de types

La **conversion de type** est une opération qui consiste à changer le type de la valeur d'une expression. Ainsi, il vous est par exemple possible de convertir une valeur de type `float` en type `int`.

5.7.1 Perte d'informations

Une perte d'informations survient quand le type d'une variable est converti vers un autre type ayant une capacité plus faible *et* que celui-ci ne peut pas contenir la valeur d'origine. Si, par exemple, nous convertissons un `double` de cent chiffres en un `int`, il y a perte d'informations, car le type `int` ne peut pas contenir un nombre de cent chiffres. Retenez donc bien cette assertion : une conversion d'un type *T* vers un type *S* de plus faible capacité entraîne une perte d'informations (une perte de précision pour les nombres).

Les conversions peuvent être vicieuses et doivent être manipulées avec précaution, au risque de tomber sur des valeurs fausses en cas de perte d'informations. Nous découvrirons d'ici quelques chapitres comment connaître la taille d'un type pour éviter ces pertes d'informations.

5.7.2 Deux types de conversions

Il existe deux types de conversions : les conversions explicites et les conversions implicites.

Les conversions explicites

Les **conversions explicites** sont des conversions demandées par le programmeur. Elles s'utilisent suivant ce modèle.

```

1  (<Type><Expression>

```

Voici par exemple un code où nous demandons explicitement la conversion d'un `double` en `int`.

```

1  int a;
2  const double pi = 3.14;
3
4  a = (int)pi;

```

La valeur de `pi` reste inchangée, elle vaudra toujours 3.14 dans la suite du programme. Par contre, `a` vaut maintenant 3, puisque la valeur de `pi` a été convertie en `int`.

Conversions implicites

Les **conversions implicites** sont des conversions spécifiées par la norme et réalisées automatiquement par le compilateur. En général, cela ne pose pas de problèmes et cela est même désirable. Par exemple, il y a toujours une conversion implicite dans le cadre d'une affectation.

Ainsi, la conversion explicite du code précédent n'est en fait pas nécessaire.

```

1  int a;
2  const double pi = 3.14;
3
4  /* Il y a conversion implicite du type double vers le type int. */
5  a = pi;

```

5.8 Sucre syntaxique

Dans les expressions vues au-dessus, nous avons utilisé des affectations pour sauvegarder le résultat de l'opération dans une variable. Les expressions obtenues ainsi sont assez longues et on peut se demander s'il existe des moyens pour écrire moins de code. Hé bien, le langage C fournit des écritures pour se simplifier la vie. Certains cas particuliers peuvent s'écrire avec des raccourcis, du « **sucre syntaxique** ».

5.9 Les opérateurs combinés

Comment vous y prendriez-vous pour multiplier une variable par trois ? La solution qui devrait vous venir à l'esprit serait d'affecter à la variable son ancienne valeur multipliée par trois.

```

1  int variable = 3;
2
3  variable = variable * 3;
4  printf("variable * 3 = %d\n", variable);

```

```

1  variable * 3 = 9

```

Ce qui est parfaitement correct. Cependant, cela implique de devoir écrire deux fois le nom de la variable, ce qui est quelques peu pénible et source d'erreurs. Aussi, il existe des opérateurs combinés qui réalisent une affectation et une opération en même temps.

Opérateur combiné	Équivalent à
variable += nombre	variable = variable + nombre
variable -= nombre	variable = variable - nombre
variable *= nombre	variable = variable * nombre
variable /= nombre	variable = variable / nombre
variable %= nombre	variable = variable % nombre

Avec le code précédent, nous obtenons ceci.

```

1  int variable = 3;
2
3  variable *= 3;
4  printf("variable * 3 = %d\n", variable);

```

```

1  variable * 3 = 9

```

5.9.1 L'incrément et la décrémentation

L'**incrément** et la **décrémentation** sont deux opérations qui, respectivement, ajoute ou enlève une unité à une variable. Avec les opérateurs vu précédemment, cela se traduit par le code ci-dessous.

```

1  variable += 1; /* Incrément */
2  variable -= 1; /* Décrément */

```

Cependant, ces deux opérations étant très souvent utilisées, aussi elles ont droit chacune à un opérateur spécifique disponible sous deux formes :

- une forme **préfixée** ;
- une forme **suffixée**.

La forme préfixée s'écrit comme ceci.

```

1  ++variable; /* Incrément */
2  --variable; /* Décrément */

```

La forme suffixée s'écrit comme cela.

```

1  variable++; /* Incrément */
2  variable--; /* Décrément */

```

Le résultat des deux paires d'opérateurs est le même : la variable `variable` est incrémentée ou décrémentée, à *une différence près* : le résultat de l'opération.

1. Dans le cas de l'opérateur préfixé (`--variable` ou `++variable`), le résultat sera la valeur de la variable augmentée ou diminuée d'une unité.
2. Dans le cas de l'opérateur suffixé (`variable--` ou `variable++`), le résultat sera la valeur de la variable.

Illustration !

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      int x = 1;
6      int y = 1;
7      int a = x++;
8      int b = ++y;
9
10     printf("a = %d\n", a);
11     printf("b = %d\n", b);
12     printf("x = %d\n", x);
13     printf("y = %d\n", y);
14     return 0;
15 }

```

```

1  a = 1
2  b = 2
3  x = 2
4  y = 2

```

Comme vous pouvez le constater, la valeur de l'expression `x++` est 1 alors que la valeur de l'expression `++y` est 2. Cela étant, dans les deux cas, les variables `x` et `y` ont bien été incrémentées.

5.10 Exercices

Vous êtes prêts pour un exercice ?

Essayez de réaliser une minicalculatrice qui :

- dit « bonjour » ;
- demande deux nombres entiers à l'utilisateur ;
- les additionne, les soustrait, les multiplie et les divise (au millième près) ;
- dit « au revoir ».

Un exemple d'utilisation pourrait être celui-ci.

```

1 Bonjour !
2 Veuillez saisir le premier nombre : 4
3 Veuillez saisir le deuxième nombre : 7
4 Calculs :
5     4 + 7 = 11
6     4 - 7 = -3
7     4 * 7 = 28
8     4 / 7 = 0.571
9 Au revoir !

```

Bien, vous avez maintenant toutes les cartes en main, donc : au boulot ! :)

```

1 #include <stdio.h>
2
3
4 int main(void)
5 {
6     int a;
7     int b;
8
9     printf("Bonjour !\n");
10
11     /* Nous demandons deux nombres à l'utilisateur */
12     printf("Veuillez saisir le premier nombre : ");
13     scanf("%d", &a);
14     printf("Veuillez saisir le deuxième nombre : ");
15     scanf("%d", &b);
16
17     /* Puis nous effectuons les calculs */
18     printf("Calculs :\n");
19     printf("\t%d + %d = %d\n", a, b, a + b);
20     printf("\t%d - %d = %d\n", a, b, a - b);
21     printf("\t%d * %d = %d\n", a, b, a * b);
22     printf("\t%d / %d = %.3f\n", a, b, a / (double)b);
23     printf("Au revoir !\n");
24     return 0;
25 }

```

Vous y êtes arrivé sans problèmes ? Bravo ! Dans le cas contraire, ne vous inquiétez pas, ce n'est pas grave. Relisez bien tous les points qui ne vous semblent pas clairs et ça devrait aller mieux.

Dans le chapitre suivant, nous nous pencherons sur les **conditions**.

Jusqu'à présent, vous avez appris à écrire du texte, manipuler des nombres et interagir un tout petit peu avec l'utilisateur.

En gros, pour le moment, un programme est quelque chose de sacrément simple et linéaire, ce dernier ne nous permettant que d'exécuter des instructions dans un ordre donné. Techniquement, une simple calculatrice peut en faire autant (voire plus). Cependant et heureusement, les langages de programmation actuels fournissent des moyens permettant de réaliser des tâches plus évoluées.

Pour ce faire, diverses **structures de contrôle** ont été inventées. Celles-ci permettent de modifier le comportement d'un programme suivant la réalisation de différentes conditions. Ainsi, si une condition est vraie, le programme se comportera d'une telle façon et à l'inverse, si elle est fausse, le programme fera telle ou telle chose.

Dans ce chapitre, nous allons voir comment rédiger des conditions à l'aide de deux catégories d'opérateurs :

- les **opérateurs de comparaison**, qui comparent deux nombres ;
- les **opérateurs logiques**, qui permettent de combiner plusieurs conditions.

6.1 Les booléens

Comme les opérateurs que nous avons vu précédemment (`+`, `-`, `*`, etc), les opérateurs de comparaison et les opérateurs logiques donnent un résultat : « vrai » si la condition est vérifiée, et « faux » si la condition est fausse. Toutefois, comme vous le savez, notre ordinateur ne voit que des nombres. Aussi, il est nécessaire de représenter ces valeurs à l'aide de ceux-ci.

Certains langages fournissent pour cela un type distinct pour stocker le résultat des opérations de comparaison et deux valeurs spécifiques : `true` (vrai) et `false` (faux). Néanmoins, dans les premières versions du langage C, ce type spécial n'existait pas¹. Il a donc fallu ruser et trouver une solution pour représenter les valeurs « vrai » et « faux ». Pour cela, la méthode la plus simple a été privilégiée : utiliser directement des nombres pour représenter ces deux valeurs. Ainsi, le langage C impose que :

- la valeur « faux » soit représentée par zéro ;
- et que la valeur « vrai » soit représentée par tout sauf zéro.

Les opérateurs de comparaison et les opérateurs logiques suivent cette convention pour représenter leur résultat. Dès lors, une condition vaudra zéro si elle est fausse et un si elle est vraie.

1. Depuis la norme C99, le type `_Bool` a été introduit ainsi que l'en-tête `<stdbool.h>` qui fournit un synonyme pour ce nouveau type : `bool`, et deux constantes entières `true` (qui vaut 1) et `false` (qui vaut zéro).

6.2 Les opérateurs de comparaison

Le langage C fournit différents opérateurs qui permettent d'effectuer des comparaisons entre des nombres. Ces opérateurs peuvent s'appliquer aussi bien à des constantes qu'à des variables (ou un mélange des deux). Ces derniers permettent donc par exemple de vérifier si une variable est supérieure à une autre, si elles sont égales, etc.

6.2.1 Comparaisons

L'écriture de conditions est similaire aux écritures mathématiques que vous voyez en cours : l'opérateur est entre les deux expressions à comparer. Par exemple, dans le cas de l'opérateur `>` (« strictement supérieur à »), il est possible d'écrire des expressions du type `a > b`, qui vérifie si la variable `a` est strictement supérieure à la variable `b`.

Le tableau ci-dessous reprend les différents opérateurs de comparaisons.

Opérateur	Signification
<code>==</code>	Égalité
<code>!=</code>	Inégalité
<code><</code>	Strictement inférieur à
<code><=</code>	Inférieur ou égal à
<code>></code>	Strictement supérieur à
<code>>=</code>	Supérieur ou égal à

Ces opérateurs ne semblent pas très folichons. En effet, avec, nous ne pouvons faire que quelques tests basiques sur des nombres. Cependant, rappelez-vous : pour un ordinateur, tout n'est que nombre et comme pour le stockage des données (revoyez le début du chapitre sur les variables si cela ne vous dit rien) il est possible de ruser et d'exprimer toutes les conditions possibles avec ces opérateurs (cela vous paraîtra plus clair quand nous passerons aux exercices).

6.2.2 Résultat d'une comparaison

Comme dit dans l'extrait plus haut, une opération de comparaison va donner zéro si elle est fausse et un si elle est vraie. Pour illustrer ceci, vérifions quels sont les résultats donnés par différentes comparaisons.

```
1 int main(void)
2 {
3     printf("10 == 20 vaut %d\n", 10 == 20);
4     printf("10 != 20 vaut %d\n", 10 != 20);
5     printf("10 < 20 vaut %d\n", 10 < 20);
6     printf("10 > 20 vaut %d\n", 10 > 20);
7
8     return 0;
9 }
```

```
1 10 == 20 vaut 0
2 10 != 20 vaut 1
3 10 < 20 vaut 1
4 10 > 20 vaut 0
```

Le résultat confirme bien ce que nous avons dit ci-dessus.

6.3 Les opérateurs logiques

Toutes ces comparaisons sont toutefois un peu faibles seules car il y a des choses qui ne sont pas possibles à vérifier en utilisant une seule comparaison. Par exemple, si un nombre est situé

entre zéro et mille (inclus). Pour ce faire, il serait nécessaire de vérifier que celui-ci est supérieur ou égal à zéro **et** inférieur ou égal à mille.

Il nous faudrait donc trouver un moyen de combiner plusieurs comparaisons entre elles pour résoudre ce problème. Hé bien rassurez-vous, le langage C fournit de quoi associer plusieurs résultats de comparaisons : les **opérateurs logiques**.

6.3.1 Les opérateurs logiques de base

Il existe trois opérateurs logiques. L'opérateur « **et** », l'opérateur « **ou** », et l'opérateur de **négarion**. Les deux premiers permettent de combiner deux conditions alors que le troisième permet d'inverser le sens d'une condition.

L'opérateur « et »

L'opérateur « et » va manipuler deux conditions. Il va donner un si elles sont vraies, et zéro sinon.

Première condition	Seconde condition	Résultat de l'opérateur « et »
Fausse	Fausse	0
Fausse	Vraie	0
Vraie	Fausse	0
Vraie	Vraie	1

Cet opérateur s'écrit `&&` et s'intercale entre les deux conditions à combiner. Si nous reprenons l'exemple vu plus haut, pour combiner les comparaisons `a >= 0` et `a <= 1000`, nous devons placer l'opérateur `&&` entre les deux, ce qui donne l'expression `a >= 0 && a <= 1000`.

L'opérateur « ou »

L'opérateur « ou » fonctionne exactement comme l'opérateur « et », il prend deux conditions et les combine pour former un résultat. Cependant, l'opérateur « ou » vérifie que l'une des deux conditions (ou que les deux) est (sont) vraie(s).

Première condition	Seconde condition	Résultat de l'opérateur « ou »
Fausse	Fausse	0
Fausse	Vraie	1
Vraie	Fausse	1
Vraie	Vraie	1

Cet opérateur s'écrit `||` et s'intercale entre les deux conditions à combiner. L'exemple suivant permet de déterminer si un nombre est divisible par trois ou par cinq (ou les deux) : `(a % 3 == 0) || (a % 5 == 0)`. Notez que les parenthèses ont été placées par soucis de lisibilité.

L'opérateur de négation

Cet opérateur est un peu spécial : il manipule une seule condition et en inverse le sens.

Condition	Résultat de l'opérateur de négation
Fausse	1
Vraie	0

Cet opérateur se note `!`. Son utilité ? Simplifier certaines expressions. Par exemple, si nous voulons vérifier cette fois qu'un nombre **n'est pas** situé entre zéro et mille, nous pouvons

utiliser la condition `a >= 0 && a <= 1000` et lui appliquer l'opérateur de négation, ce qui donne `!(a >= 0 && a <= 1000)`.



Faites bien attention à l'utilisation des parenthèses ! L'opérateur de négation s'applique à l'opérande le plus proche (sans parenthèses, il s'agirait de `a`). Veuillez donc bien entourer de parenthèses l'expression concernée par la négation.



Notez que pour cet exemple, il est parfaitement possible de se passer de cet opérateur à l'aide de l'expression `a < 0 || a > 1000`. Il est d'ailleurs souvent possible d'exprimer une condition de différentes manières.

6.3.2 Évaluation en court-circuit

Les opérateurs `&&` et `||` évaluent toujours la première condition avant la seconde. Cela paraît évident, mais ce n'est pas le cas dans tous les langages de programmation. Ce genre de détail permet à ces opérateurs de disposer d'un comportement assez intéressant : **l'évaluation en court-circuit**.

De quoi s'agit-il ? Pour illustrer cette notion, reprenons l'exemple précédent : nous voulons vérifier si un nombre est compris entre zéro et mille, ce qui donne l'expression `a >= 0 && a <= 1000`. Si jamais `a` est inférieur à zéro, nous savons dès la vérification de la première condition qu'il n'est pas situé dans l'intervalle voulu. Il n'est donc pas nécessaire de vérifier la seconde. Hé bien, c'est exactement ce qu'il se passe en langage C. Si le résultat de la première condition suffit pour déterminer le résultat de l'opérateur `&&` ou `||`, alors la seconde condition n'est pas évaluée. Voilà pourquoi l'on parle d'évaluation en court-circuit.

Plus précisément, ce sera le cas pour l'opérateur `&&` si la première condition est fausse et pour l'opérateur `||` si la première condition est vraie (relisez les tableaux précédents si cela ne vous semble pas évident).

Ce genre de propriété peut-être utilisé efficacement pour éviter de faire certains calculs, en choisissant intelligemment quelle sera la première condition.

6.3.3 Combinaisons

Bien sûr, il est possible de mélanger ces opérateurs pour créer des conditions plus complexes. Voici un exemple un peu plus long (et inutile, soit dit en passant :-°).

```

1  int a = 3;
2  double b = 64.67;
3  double c = 12.89;
4  int d = 8;
5  int e = -5;
6  int f = 42;
7  int r;
8
9  r = ((a < b && b > 32) || (c < d + b || e == 0)) && (f > d);
10 printf("La valeur logique est égale à : %d\n", r);

```

Ici, la variable `r` est égale à 1, la condition est donc vraie.

Parenthèses

En regardant le code écrit plus haut, vous avez sûrement remarqué la présence de plusieurs parenthèses. Celles-ci permettent d'enlever toute ambiguïté dans les expressions créées avec des opérateurs logiques. En effet, comme pour les opérateurs mathématiques, les opérateurs logiques ont une priorité (revoyez le chapitre sur les opérations mathématiques si cela ne vous dit rien)

qui fait que l'opérateur `&&` passe *avant* l'opérateur `||`. Ainsi, le premier code est équivalent au second car l'opérateur `&&` est évalué *avant* l'opérateur `||`.

```
1 printf("%d\n", (a && b) || (c && d));
```

```
1 printf("%d\n", a && (b || c) && d );
```

Si vous souhaitez un autre résultat, il est nécessaire d'ajouter des parenthèses pour modifier la priorité par défaut, par exemple comme ceci.

`c printf("%d\n", a && (b || c) && d);` Au prochain chapitre, nous allons combiner les conditions avec une seconde notion : les **sélections**.

Au prochain chapitre, nous allons combiner les conditions avec une seconde notion: les sélections.

Comme dit au chapitre précédent, les structures de contrôle permettent de modifier le comportement d'un programme suivant la réalisation de différentes conditions. Parmi ces structures de contrôle se trouvent les **instructions de sélection** (ou **sélections** en abrégé) qui vont retenir notre attention dans ce chapitre.

Le tableau ci-dessous reprend celles dont dispose le langage C.

Structure de sélection	Action
if...	exécute une suite d'instructions si une condition est respectée.
else	exécute une suite d'instructions si une condition est respectée ou exécute une autre suite d'instructions si elle ne l'est pas.
switch	exécute une suite d'instructions différente suivant la valeur testée.

7.1 La structure if

Vous savez désormais manipuler des conditions, c'est bien, cependant l'intérêt de la chose reste assez limité pour l'instant. Rendons à présent cela plus intéressant en voyant comment exécuter un bloc d'instruction quand une ou plusieurs conditions sont respectées. C'est le rôle de l'instruction `if` et de ses consœurs.

7.1.1 L'instruction if

L'instruction `if` permet d'exécuter un bloc d'instructions si une condition est vérifiée ou de le passer si ce n'est pas le cas.

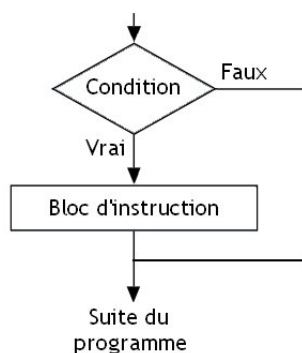


FIGURE 7.1 – Structure If

L'instruction `if` ressemble à ceci.

```

1  if (/* Condition */)
2  {
3      /* Une ou plusieurs instruction(s) */
4  }
```

Si la condition n'est pas vérifiée, le bloc d'instructions est passé et le programme recommence immédiatement à la suite du bloc d'instructions délimité par l'instruction `if`.

Si vous n'avez qu'une seule instruction à réaliser, vous avez la possibilité de ne pas mettre d'accolades.

```

1  if (/* Condition */)
2      /* Une seule instruction */
```

Cependant, nous vous conseillons de mettre les accolades systématiquement afin de vous éviter des problèmes si vous décidez de rajouter des instructions par la suite en oubliant d'ajouter des accolades. Bien sûr, ce n'est qu'un conseil, vous êtes libre de ne pas le suivre.

À présent, voyons quelques exemples d'utilisation.

Exemple 1

```

1  #include <stdio.h>
2
3
4  int main(void)
5  {
6      int a = 10;
7      int b = 20;
8
9      if (a < b)
10     {
11         printf("%d est inférieur à %d\n", a, b);
12     }
13
14     return 0;
15 }
```

```

1  10 est inférieur à 20
```

L'instruction `if` évalue l'expression logique `a < b`, conclut qu'elle est valide et exécute le bloc d'instructions.

Exemple 2

```

1  #include <stdio.h>
2
3
4  int main(void)
5  {
6      int a = 10;
7      int b = 20;
8
9      if (a > b)
10     {
11         printf("%d est supérieur à %d\n", a, b);
12     }
13
14     return 0;
15 }
```

Ce code n'affiche rien. La condition étant fausse, le bloc contenant l'appel à la fonction `printf()` est ignoré.

7.1.2 L’instruction else

Avec l’instruction `if`, nous savons exécuter un bloc d’instructions quand une condition est remplie. Toutefois, si nous souhaitons réaliser une action en cas d’échec de l’évaluation de la condition, nous devons ajouter une autre instruction `if` à la suite, comme ci-dessous.

```
1  if (a > 5)
2  {
3      /* Du code */
4  }
5
6  if (a <= 5)
7  {
8      /* Code alternatif */
9  }
```

Le seul problème, c’est qu’il est nécessaire d’ajouter une instruction `if` et d’évaluer une nouvelle condition, ce qui n’est pas très efficace et assez long à taper. Pour limiter les dégâts, le C fournit une autre instruction : `else`, qui signifie « sinon ». Celle-ci se place immédiatement après le bloc d’une instruction `if` et permet d’exécuter un bloc d’instructions alternatif si la condition testée n’est pas vérifiée. Sa syntaxe est la suivante.

```
1  if (/* Condition */)
2  {
3      /* Une ou plusieurs instructions */
4  }
5  else
6  {
7      /* Une ou plusieurs instructions */
8  }
```

Et elle doit être comprise comme ceci.

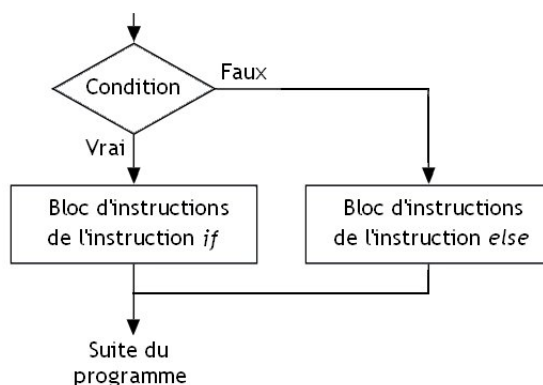


FIGURE 7.2 – Structure if else



Notez que l’instruction `else` ne possède aucune parenthèse.

7.1.3 Exemple

Supposons que nous voulions créer un programme très simple auquel nous fournissons une heure et qui indique s’il fait jour ou nuit à cette heure-là. Nous supposerons qu’il fait jour de 8 heures à 20 heures et qu’il fait nuit sinon.

```

1  int main(void)
2  {
3      int heure;
4      scanf("%d", &heure);
5
6      if (heure > 8 && heure < 20)
7      {
8          printf("Il fait jour.\n");
9      }
10     else
11     {
12         printf("Il fait nuit.\n");
13     }
14
15     return 0;
16 }

```

```

1  10
2  Il fait jour.

```

7.2 If / else if

Il est parfois nécessaire d'imbriquer plusieurs instructions `if` et `else` les unes dans les autres.

```

1  if (/* Condition */)
2  {
3      /* Du code */
4  }
5  else
6  {
7      /* Une ou plusieurs instruction(s) */
8
9      if (/* Autre condition */)
10     {
11         /* Du code */
12     }
13 }

```

Cependant, c'est assez long à écrire, d'autant plus s'il y a beaucoup d'imbrications. Pour éviter ces inconvénients, sachez qu'il est possible de combiner une instruction `else` et une instruction `if`. Les imbrications simplifiables avec une suite `else if` sont celles qui s'écrivent comme suit.

```

1  if (/* Expression logique */)
2  {
3      /* Une ou plusieurs instruction(s) */
4  }
5  else
6  {
7      if (/* Expression logique */)
8      {
9          /* Une ou plusieurs instruction(s) */
10     }
11 }

```

Faites bien attention : le bloc d'instructions du `else` doit contenir un `if`, éventuellement avec un `else`, mais rien d'autre. Celles-ci peuvent alors être simplifiées comme ceci.

```

1  if (/* Expression logique */)
2  {
3      /* Une ou plusieurs instruction(s) */
4  }
5  else if (/* Expression logique */)

```

```

6 {
7     /* Une ou plusieurs instruction(s) */
8 }

```

Comme vous pouvez le voir, nous avons « fusionné » l’instruction `else` et la seconde instruction `if`. Notez que comme il s’agit toujours d’une suite d’instructions `if` et `else`, il n’y aura qu’un seul bloc d’instructions qui sera finalement exécuté. En effet, l’ordinateur va tester la condition de l’instruction `if`, puis, si elle est fausse, celle de l’instruction `if` suivant l’instruction `else` et ainsi de suite jusqu’à ce qu’une condition soit vraie (ou jusqu’à une instruction `else` finale si elles sont toutes fausses).



Notez qu’il n’est pas obligatoire d’ajouter une instruction `else`.

```

1  #include <stdio.h>
2
3
4  int main(void)
5  {
6      int heure = 11;
7
8      if (heure < 7)
9      {
10         printf("Zzz... \n");
11     }
12     else if (heure >= 7 && heure < 12)
13     {
14         printf("C'est le matin !\n");
15     }
16     else if (heure == 12)
17     {
18         printf("Il est midi !\n");
19     }
20     else if (heure > 12 && heure < 18)
21     {
22         printf("C'est l'après-midi !\n");
23     }
24     else if (heure >= 18 && heure < 24)
25     {
26         printf("C'est le soir !\n");
27     }
28     else if (heure == 24 || heure == 0)
29     {
30         printf("Il est minuit, dormez brave gens !\n");
31     }
32     else
33     {
34         printf("Il est l'heure de réapprendre à lire l'heure !\n");
35     }
36
37     return 0;

```

```

1  11
2  On est le matin !
3  0
4  Il est minuit, dormez brave gens !
5  -2
6  Il est l'heure de réapprendre à lire l'heure !

```

7.2.1 Exercice

Imaginez que vous avez un score de jeu vidéo sous la main :

- si le score est strictement inférieur à deux mille, affichez « C’est la catastrophe! » ;
- si le score est supérieur ou égal à deux mille et que le score est strictement inférieur à cinq mille, affichez : « Tu peux mieux faire! » ;

- si le score est supérieur ou égal à cinq mille et que le score est strictement inférieur à neuf mille, affichez : « Tu es sur la bonne voie! » ;
 - sinon, affichez : « Tu es le meilleur! ».
- Au boulot !

```

1  #include <stdio.h>
2
3
4  int main(void)
5  {
6      int score;
7
8      printf("Quel est le score du joueur ? ");
9      scanf("%d", &score);
10
11     if (score < 2000)
12     {
13         printf("C'est la catastrophe !\n");
14     }
15     else if (score >= 2000 && score < 5000)
16     {
17         printf("Tu peux mieux faire !\n");
18     }
19     else if (score >= 5000 && score < 9000)
20     {
21         printf("Tu es sur la bonne voie !\n");
22     }
23     else
24     {
25         printf("Tu es le meilleur !\n");
26     }
27
28     return 0;
29 }
```

7.3 L'instruction switch

L'instruction `switch` permet de comparer la valeur d'une variable par rapport à une liste de valeurs. Techniquement, elle permet d'écrire de manière plus concise une suite d'instructions `if` et `else` qui auraient pour objectif d'accomplir différentes actions suivant la valeur d'une variable.

```

1  if (a == 1)
2  {
3      /* Instruction(s) */
4  }
5  else if (a == 2)
6  {
7      /* Instruction(s) */
8  }
9  /* Etc. */
10 else
11 {
12     /* Instruction(s) */
13 }
```

Avec l'instruction `switch`, cela donne ceci.

```

1  switch (a)
2  {
3      case 1:
4          /* Instruction(s) */
5          break;
6      case 2:
7          /* Instruction(s) */
8          break;
9  }
```

```
10  /* Etc... */
11
12  default: /* Si aucune comparaison n'est juste */
13          /* Instruction(s) à exécuter dans ce cas */
14          break;
15 }
```

Ici, la valeur de la variable *a* est comparée successivement avec chaque entrée de la liste, indiquées par le mot-clé `case`. En cas de correspondance, les instructions suivant le mot-clé `case` sont exécutées jusqu'à rencontrer une instruction `break` (nous la verrons plus en détail un peu plus tard). Si aucune comparaison n'est bonne, alors ce sont les instructions de l'entrée marquée avec le mot-clé `default` qui seront exécutées.

7.3.1 Exemple

```
1  #include <stdio.h>
2
3
4  int main(void)
5  {
6      int note;
7
8      printf("Quelle note as-tu obtenue (sur cinq) ? ");
9      scanf("%d", &note);
10
11     switch(note)
12     {
13         /* si note == 0 */
14         case 0:
15             printf("No comment.\n");
16             break;
17
18         /* si note == 1 */
19         case 1:
20             printf("Cela te fait 4/20, c'est accablant.\n");
21             break;
22
23         /* si note == 2 */
24         case 2:
25             printf("On se rapproche de la moyenne, mais ce n'est pas encore ça.\n");
26             break;
27
28         /* si note == 3 */
29         case 3:
30             printf("Tu passes.\n");
31             break;
32
33         /* si note == 4 */
34         case 4:
35             printf("Bon travail, continue ainsi !\n");
36             break;
37
38         /* si note == 5 */
39         case 5:
40             printf("Excellent !\n");
41             break;
42
43         /* si note est différente de 0, 1, 2, 3, 4 et 5 */
44         default:
45             printf("Euh... tu possèdes une note improbable...\n");
46             break;
47     }
48
49     return 0;
50 }
```



Notez que comme pour l'instruction `else`, une entrée marquée avec le mot-clé `default` n'est pas obligatoire.

7.3.2 Plusieurs entrées pour une même action

Une même suite d'instructions pour être désignée par plusieurs entrées comme le montre l'exemple suivant.

```
1  int main(void)
2  {
3      int note;
4
5      printf("Quelle note as-tu obtenue ? ");
6      scanf("%d", &note);
7
8      switch(note)
9      {
10         /* si la note est comprise entre zéro et trois inclus */
11         case 0:
12         case 1:
13         case 2:
14         case 3:
15             printf("No comment.\n");
16             break;
17
18         /* si la note est comprise entre quatre et sept inclus */
19         case 4:
20         case 5:
21         case 6:
22         case 7:
23             printf("C'est accablant.\n");
24             break;
25
26         /* si la note est comprise entre huit et neuf inclus */
27         case 8:
28         case 9:
29             printf("On se rapproche de la moyenne, mais ce n'est pas encore ça.\n");
30             break;
31
32         /* si la note est comprise entre dix et douze inclus */
33         case 10:
34         case 11:
35         case 12:
36             printf("Tu passes.\n");
37             break;
38
39         /* si la note est comprise entre treize et seize inclus */
40         case 13:
41         case 14:
42         case 15:
43         case 16:
44             printf("Bon travail, continue ainsi !\n");
45             break;
46
47         /* si la note est comprise entre dix-sept et vingt inclus */
48         case 17:
49         case 18:
50         case 19:
51         case 20:
52             printf("Excellent !\n");
53             break;
54
55         /* si la note est différente */
56         default:
57             printf("Euh... tu possèdes une note improbable...\n");
58             break;
59     }
60 }
```

```

61     return 0;
62 }

```

7.4 Plusieurs entrées sans instruction break

Si vous l'utiliserez souvent, sachez également que l'instruction `break` n'est pas obligatoire. En effet, le but de cette dernière est de sortir du `switch` et donc de ne pas exécuter les actions d'autre entrées. Toutefois, il arrive que les actions à réaliser se chevauchent entre entrées auquel cas l'instruction `break` serait plutôt mal venue.

Prenons un exemple : vous souhaitez réaliser un programme qui affiche entre 1 à dix fois la même phrase, ce nombre étant fourni par l'utilisateur. Vous pourriez écrire une suite de `if` ou différentes entrées d'un `switch` qui, suivant le nombre entré, appelleraient une fois `printf()`, puis deux fois, puis trois fois, etc. mais cela serait horriblement lourd.

Dans un tel cas, une meilleure solution consiste à appeler `printf()` à chaque entrée du `switch`, mais de ne pas terminer ces dernières par une instruction `break`.

```

1  #include <stdio.h>
2
3
4  int
5  main(void)
6  {
7      unsigned nb;
8
9      printf("Combien de fois souhaitez-vous répéter l'affichage (entre 1 à 10 fois) ? ");
10     scanf("%u", &nb);
11
12     switch (nb)
13     {
14     case 10:
15         printf("Cette phrase est répétée une à dix fois.\n");
16     case 9:
17         printf("Cette phrase est répétée une à dix fois.\n");
18     case 8:
19         printf("Cette phrase est répétée une à dix fois.\n");
20     case 7:
21         printf("Cette phrase est répétée une à dix fois.\n");
22     case 6:
23         printf("Cette phrase est répétée une à dix fois.\n");
24     case 5:
25         printf("Cette phrase est répétée une à dix fois.\n");
26     case 4:
27         printf("Cette phrase est répétée une à dix fois.\n");
28     case 3:
29         printf("Cette phrase est répétée une à dix fois.\n");
30     case 2:
31         printf("Cette phrase est répétée une à dix fois.\n");
32     case 1:
33         printf("Cette phrase est répétée une à dix fois.\n");
34     case 0:
35         break;
36     default:
37         printf("Certes, mais encore ?\n");
38         break;
39     }
40
41     return 0;
42 }

```

```

1  Combien de fois souhaitez-vous répéter l'affichage (entre 1 à 10 fois) ? 2
2  Cette phrase est répétée une à dix fois.
3  Cette phrase est répétée une à dix fois.
4
5  Combien de fois souhaitez-vous répéter l'affichage (entre 1 à 10 fois) ? 5
6  Cette phrase est répétée une à dix fois.

```

```

7 Cette phrase est répétée une à dix fois.
8 Cette phrase est répétée une à dix fois.
9 Cette phrase est répétée une à dix fois.
10 Cette phrase est répétée une à dix fois.

```

Comme vous le voyez, la phrase « Cette phrase est répétée une à dix fois » est affichée une à dix fois suivant le nombre initialement fourni. Cela est possible étant donné l'absence d'instruction `break` entre les `case` 10 à 1, ce qui fait que l'exécution du `switch` continue de l'entrée initiale jusqu'au `case` 0.

7.5 L'opérateur conditionnel

L'**opérateur conditionnel** ou **opérateur ternaire** est un opérateur particulier dont le résultat dépend de la réalisation d'une condition. Son deuxième nom lui vient du fait qu'il est le seul opérateur du langage C à requérir trois opérandes : une condition et deux expressions.

```

1 (condition) ? expression si vrai : expression si faux

```



Les parenthèses entourant la condition ne sont pas obligatoires, mais préférables.

Grosso modo, cet opérateur permet d'écrire de manière condensée une structure `if {} else {}`. Voyez par vous-mêmes.

```

1 #include <stdio.h>
2
3 int main(void)
4 {
5     int heure;
6
7     scanf("%d", &heure);
8
9     (heure > 8 && heure < 20) ? printf("Il fait jour.\n") : printf("Il fait nuit.\n");
10    return 0;
11 }

```

Il est également possible de l'écrire sur plusieurs lignes, même si cette pratique est moins courante.

```

1 (heure > 8 && heure < 20)
2 ? printf("Il fait jour.\n")
3 : printf("Il fait nuit.\n");

```

Cet opérateur peut sembler inutile de prime abord, mais il s'avère être un allié de choix pour simplifier votre code quand celui-ci requiert la vérification de conditions simples.

7.5.1 Exercice

Pour bien comprendre cette nouvelle notion, nous allons faire un petit exercice. Imaginez que nous voulions faire un mini jeu vidéo dans lequel nous affichons le nombre de coups du joueur. Seulement voilà, vous êtes maniaques du français et vous ne supportez pas qu'il y ait un « s » en trop ou en moins. Essayez de réaliser un programme qui demande à l'utilisateur d'entrer un nombre de coups puis qui affiche celui-ci correctement accordé.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int nb_coups;
6
7      printf("Donnez le nombre de coups : ");
8      scanf("%d", &nb_coups);
9      printf("Vous gagnez en %d coup%c\n", nb_coups, (nb_coups > 1) ? 's' : ' ');
10     return 0;
11 }
```

Ce programme utilise l'opérateur conditionnel pour condenser l'expression et aller plus vite dans l'écriture du code. Sans lui nous aurions dû écrire quelque chose comme ceci.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int nb_coups;
6
7      printf("Donnez le nombre de coups : ");
8      scanf("%d", &nb_coups);
9
10     if (nb_coups > 1)
11         printf("Vous gagnez en %d coups\n", nb_coups);
12     else
13         printf("Vous gagnez en %d coup\n", nb_coups);
14
15     return 0;
16 }
```

Ce chapitre a été important, il vous a permis d'utiliser les conditions ; les instructions `if` et `else` ; l'instruction `switch` et l'opérateur conditionnel. Aussi, si vous n'avez pas très bien compris ou que vous n'avez pas tout retenu, nous vous conseillons de relire ce chapitre.

Le chapitre suivant sera l'occasion de mettre en œuvre ce que vous avez appris puisqu'il s'agira de votre premier TP.

*[TP] : Travaux Pratiques

TP : déterminer le jour de la semaine

Avant de poursuivre notre périple, il est à présent temps de nous poser un instant afin de réaliser un petit exercice reprenant tout ce qui vient d'être vu.

8.1 Objectif

Votre objectif est de parvenir à réaliser un programme qui, suivant une date fournie par l'utilisateur sous la forme « jj/mm/aaaa », donne le jour de la semaine correspondant. Autrement dit, voici ce que devrait donner l'exécution de ce programme.

```

1  Entrez une date : 11/2/2015
2  C'est un mercredi
3
4  Entrez une date : 13/7/1970
5  C'est un lundi

```

8.2 Première étape

Pour cette première étape, vous allez devoir réaliser un programme qui demande à l'utilisateur un jour du mois de janvier de l'an un et qui lui précise de quel jour de la semaine il s'agit, le tout à l'aide de la méthode présentée ci-dessous.

```

1  Entrez un jour : 27
2  C'est un jeudi

```

8.2.1 Déterminer le jour de la semaine

Pour déterminer le jour de la semaine correspondant à une date, vous allez devoir partir du premier janvier de l'an un (c'était un samedi) et calculer le nombre de jours qui sépare cette date de celle fournie par l'utilisateur. Une fois ce nombre obtenu, il nous est possible d'obtenir le jour de la semaine correspondant à l'aide de l'opérateur modulo.

En effet, comme vous le savez, les jours de la semaine suivent un cycle et se répètent tous les sept jours. Or, le reste de la division entière est justement un nombre cyclique allant de zéro jusqu'au diviseur diminué de un. Voici ce que donne le reste de la division entière des chiffres 1 à 9 par 3.

```

1  1 % 3 = 1
2  2 % 3 = 2
3  3 % 3 = 0
4  4 % 3 = 1

```



```

5 5 % 3 = 2
6 6 % 3 = 0
7 7 % 3 = 1
8 8 % 3 = 2
9 9 % 3 = 0

```

Comme vous le voyez, le reste de la division oscille toujours entre zéro et deux. Ainsi, si nous attribuons un chiffre de zéro à six à chaque jour de la semaine (par exemple zéro pour samedi et ainsi de suite pour les autres) nous pouvons déduire le jour de la semaine correspondant à un nombre de jours depuis le premier janvier de l'an un.

Prenons un exemple : l'utilisateur entre la date du vingt-sept janvier de l'an un. Il y a vingt-six jours qui le sépare du premier janvier. Le reste de la division entière de vingt-six par 7 est 5, il s'agit donc d'un jeudi.

Si vous le souhaitez, vous pouvez vous aider du calendrier suivant.

```

1      Janvier 1
2 di lu ma me je ve sa
3
4      2 3 4 5 6 7 8
5      9 10 11 12 13 14 15
6     16 17 18 19 20 21 22
7     23 24 25 26 27 28 29
8     30 31

```

À présent, à vous de jouer. ;)

8.3 Correction

Alors, cela s'est bien passé ? Si oui, félicitations, si non, la correction devrait vous aider à y voir plus clair.

```

1  #include <stdio.h>
2
3
4  int
5  main(void)
6  {
7      unsigned jour;
8      int njours;
9
10     printf("Entrez un jour : ");
11     scanf("%u", &jour);
12
13     njours = (jour - 1);
14
15     switch (njours % 7)
16     {
17     case 0:
18         printf("C'est un samedi\n");
19         break;
20
21     case 1:
22         printf("C'est un dimanche\n");
23         break;
24
25     case 2:
26         printf("C'est un lundi\n");
27         break;
28
29     case 3:
30         printf("C'est un mardi\n");
31         break;
32
33     case 4:

```

```

34     printf("C'est un mercredi\n");
35     break;
36
37     case 5:
38         printf("C'est un jeudi\n");
39         break;
40
41     case 6:
42         printf("C'est un vendredi\n");
43         break;
44     }
45
46     return 0;
47 }

```

Tout d'abord, nous demandons à l'utilisateur d'entrer un jour du mois de janvier que nous affectons à la variable `jour`. Ensuite, nous calculons la différence de jours séparant le premier janvier de celui entré par l'utilisateur. Enfin, nous appliquons le modulo à ce résultat afin d'obtenir le jour de la semaine correspondant.

8.4 Deuxième étape

Bien, complexifions à présent un peu notre programme et demandons à l'utilisateur de nous fournir un jour *et* un mois de l'an un.

```

1  Entrez une date (jj/mm) : 20/4
2  C'est un mercredi

```

Pour ce faire, vous allez devoir convertir chaque mois en son nombre de jours et ajouter ensuite celui-ci au nombre de jours séparant la date entrée du premier du mois. À cette fin, vous pouvez considérer dans un premier temps que chaque mois compte trente et un jours et ensuite retrancher les jours que vous avez compté en trop suivant le mois fourni.

Par exemple, si l'utilisateur vous demande quel jour de la semaine était le vingt avril de l'an un :

- vous multipliez trente et un par trois puisque trois mois séparent le mois d'avril du mois de janvier (janvier, février et mars) ;
- vous retranchez trois jours (puisque le mois de février ne comporte que vingt-huit jours les années non bissextiles) ;
- enfin, vous ajoutez les dix-neuf jours qui séparent la date fournie du premier du mois.

Au total, vous obtenez alors cent et neuf jours, ce qui nous donne, modulo sept, le nombre quatre, c'est donc un mercredi.

À toutes fins utiles, voici le calendrier complet de l'an un.

Janvier							Février							Mars							
di	lu	ma	me	je	ve	sa	di	lu	ma	me	je	ve	sa	di	lu	ma	me	je	ve	sa	
						1			1	2	3	4	5			1	2	3	4	5	
2	3	4	5	6	7	8	6	7	8	9	10	11	12	6	7	8	9	10	11	12	
9	10	11	12	13	14	15	13	14	15	16	17	18	19	13	14	15	16	17	18	19	
16	17	18	19	20	21	22	20	21	22	23	24	25	26	20	21	22	23	24	25	26	
23	24	25	26	27	28	29	27	28	27	28	29	30	31								
30	31																				
Avril							Mai							Juin							
di	lu	ma	me	je	ve	sa	di	lu	ma	me	je	ve	sa	di	lu	ma	me	je	ve	sa	
						1	1	2	3	4	5	6	7			1	2	3	4		
3	4	5	6	7	8	9	8	9	10	11	12	13	14	5	6	7	8	9	10	11	
10	11	12	13	14	15	16	15	16	17	18	19	20	21	12	13	14	15	16	17	18	
17	18	19	20	21	22	23	22	23	24	25	26	27	28	19	20	21	22	23	24	25	
24	25	26	27	28	29	30	29	30	31	26	27	28	29	30							

18																																										
19	Juillet														Aoû t														Septembre													
20	di	lu	ma	me	je	ve	sa								di	lu	ma	me	je	ve	sa								di	lu	ma	me	je	ve	sa							
21								1	2								1	2	3	4	5	6															1	2	3			
22	3	4	5	6	7	8	9								7	8	9	10	11	12	13								4	5	6	7	8	9	10							
23	10	11	12	13	14	15	16								14	15	16	17	18	19	20								11	12	13	14	15	16	17							
24	17	18	19	20	21	22	23								21	22	23	24	25	26	27								18	19	20	21	22	23	24							
25	24	25	26	27	28	29	30								28	29	30	31															25	26	27	28	29	30				
26	31																																									
27																																										
28	Octobre														Novembre														Déc e mbr e													
29	di	lu	ma	me	je	ve	sa								di	lu	ma	me	je	ve	sa								di	lu	ma	me	je	ve	sa							
30								1								1	2	3	4	5															1	2	3					
31	2	3	4	5	6	7	8								6	7	8	9	10	11	12								4	5	6	7	8	9	10							
32	9	10	11	12	13	14	15								13	14	15	16	17	18	19								11	12	13	14	15	16	17							
33	16	17	18	19	20	21	22								20	21	22	23	24	25	26								18	19	20	21	22	23	24							
34	23	24	25	26	27	28	29								27	28	29	30															25	26	27	28	29	30	31			
35	30 31																																									

À vos claviers!

8.5 Correction

Bien, passons à la correction.

```

1  #include <stdio.h>
2
3
4  int
5  main(void)
6  {
7      unsigned jour;
8      unsigned mois;
9      int njours;
10
11      printf("Entrez une date (jj/mm) : ");
12      scanf("%u/%u", &jour, &mois);
13
14      njours = (mois - 1) * 31;
15
16      switch (mois)
17      {
18          case 12:
19              --njours;
20          case 11:
21          case 10:
22              --njours;
23          case 9:
24          case 8:
25          case 7:
26              --njours;
27          case 6:
28          case 5:
29              --njours;
30          case 4:
31          case 3:
32              njours -= 3;
33              break;
34      }
35
36      njours += (jour - 1);
37
38      switch (njours % 7)
39      {
40          case 0:
41              printf("C'est un samedi\n");
42              break;
43
44          case 1:

```

```

45     printf("C'est un dimanche\n");
46     break;
47
48     case 2:
49         printf("C'est un lundi\n");
50         break;
51
52     case 3:
53         printf("C'est un mardi\n");
54         break;
55
56     case 4:
57         printf("C'est un mercredi\n");
58         break;
59
60     case 5:
61         printf("C'est un jeudi\n");
62         break;
63
64     case 6:
65         printf("C'est un vendredi\n");
66         break;
67 }
68
69 return 0;
70 }

```

Nous commençons par demander deux nombres à l'utilisateur qui sont affectés aux variables `jours` et `mois`. Ensuite, nous multiplions le nombre de mois séparant celui entré par l'utilisateur du mois de janvier par trente et un. Après quoi, nous soustrayons les jours comptés en trop suivant le mois fourni. Enfin, nous ajoutons le nombre de jours séparant celui entré du premier du mois, comme pour la première étape.



Notez que nous avons utilisé ici une propriété intéressante de l'instruction `switch` : si la valeur de contrôle correspond à celle d'une entrée, alors les instructions sont exécutées *jusqu'à rencontrer une instruction* `break` (ou jusqu'à la fin du `switch`). Ainsi, si le mois entré est celui de mai, l'instruction `--njours` va être exécutée, puis l'instruction `njours -= 3` va également être exécutée.

8.6 Troisième et dernière étape

À présent, il est temps de réaliser un programme complet qui correspond aux objectifs du TP. Vous allez donc devoir demander à l'utilisateur une date entière et lui donner le jour de la semaine correspondant.

```

1  Entrez une date : 11/2/2015
2  C'est un mercredi

```

Toutefois, avant de vous lancer dans la réalisation de celui-ci, nous allons parler calendriers et années bissextiles.

8.6.1 Les calendriers Julien et Grégorien

Vous le savez certainement, une **année bissextile** est une année qui comporte 366 jours au lieu de 365 et qui se voit ainsi ajouter un vingt-neuf février. Ce que vous ne savez en revanche peut-être pas, c'est que la détermination des années bissextile a varié au cours du temps.

Jusqu'en 1582, date d'adoption du **calendrier Grégorien** (celui qui est en vigueur un peu près partout actuellement), c'est le **calendrier Julien** qui était en application. Ce dernier considérait

une année comme bissextile si celle-ci était multiple de quatre. Cette méthode serait correcte si une année comportait 365,25 jours. Cependant, il s'est avéré plus tard qu'une année comportait en fait 365,2422 jours.

Dès lors, un décalage par rapport au cycle terrestre s'était lentement installé ce qui posa problème à l'Église catholique pour le calcul de la date de Pâques qui glissait doucement vers l'été. Le calendrier Grégorien fût alors instauré en 1582 pour corriger cet écart en modifiant la règle de calcul des années bissextiles : il s'agit d'une année multiple de quatre *et*, s'il s'agit d'une année multiple de 100, également multiple de 400. Par exemple, les années 1000 et 1100 ne sont plus bissextiles à l'inverse de l'année 1200 qui, elle, est divisible par 400.

Toutefois, ce ne sont pas douze années bissextiles qui ont été supprimées lors de l'adoption du calendrier Grégorien (100, 200, 300, 500, 600, 700, 900, 1000, 1100, 1300, 1400, 1500), mais seulement dix afin de rapprocher la date de Pâques de l'équinoxe de printemps.

8.6.2 Mode de calcul

Pour réaliser votre programme, vous devrez donc vérifier si la date demandée est antérieure ou postérieure à l'an 1582. Si elle est inférieure ou égale à l'an 1582, alors vous devrez appliquer le calendrier Julien. Si elle est supérieure, vous devrez utiliser le calendrier Grégorien.

Pour vous aider, voici un schéma que vous pouvez suivre.

```

1  Si l'année est supérieure à 1582
2      Multiplier la différence d'années par 365
3      Ajouter au résultat le nombre d'années multiples de 4
4      Soustraire à cette valeur le nombre d'années multiples de 100
5      Ajouter au résultat le nombre d'années multiples de 400
6      Ajouter deux à ce nombre (du fait que seules dix années ont été supprimées en 1582)
7  Si l'année est inférieure ou égale à 1582
8      Multiplier la différence d'années par 365
9      Ajouter au résultat le nombre d'années multiples de 4
10
11  Au nombre de jours obtenus, ajouter la différence de jours entre
12  le mois de janvier et le mois fourni. N'oubliez pas que les mois comportent
13  trente et un ou trente jours et que le mois de février comporte pour sa
14  part vingt-huit jours sauf les années bissextiles où il s'en voit ajouter un
15  vingt-neuvième. Également, faites attention au calendrier en application pour
16  la détermination des années bissextiles !
17
18  Au résultat obtenu ajouter le nombre de jour qui sépare celui entré du premier
19  du mois.
20
21  Appliquer le modulo et déterminer le jour de la semaine.
```

À vous de jouer !

8.7 Correction

Ça va, vous tenez bon ?

```

1  #include <stdio.h>
2
3
4  int
5  main(void)
6  {
7      unsigned jour;
8      unsigned mois;
9      unsigned an;
10     int njours;
11
12     printf("Entrez une date (jj/mm/aaaa) : ");
```

```
13  scanf("%u/%u/%u", &jour, &mois, &an);
14  njours = (an - 1) * 365;
15
16  if (an > 1582) /* Calendrier Grégorien */
17  {
18      njours += ((an - 1) / 4);
19      njours -= ((an - 1) / 100);
20      njours += ((an - 1) / 400);
21      njours += 2;
22  }
23  else /* Calendrier Julien */
24      njours += ((an - 1) / 4);
25
26  njours += (mois - 1) * 31;
27
28  switch (mois)
29  {
30      case 12:
31          --njours;
32      case 11:
33      case 10:
34          --njours;
35      case 9:
36      case 8:
37      case 7:
38          --njours;
39      case 6:
40      case 5:
41          --njours;
42      case 4:
43      case 3:
44          if (an > 1582)
45          {
46              if (an % 4 == 0 && (an % 100 != 0 || an % 400 == 0))
47                  njours -= 2;
48              else
49                  njours -= 3;
50          }
51          else
52          {
53              if (an % 4 == 0)
54                  njours -= 2;
55              else
56                  njours -= 3;
57          }
58          break;
59      }
60
61
62  njours += (jour - 1);
63
64  switch (njours % 7)
65  {
66      case 0:
67          printf("C'est un samedi\n");
68          break;
69
70      case 1:
71          printf("C'est un dimanche\n");
72          break;
73
74      case 2:
75          printf("C'est un lundi\n");
76          break;
77
78      case 3:
79          printf("C'est un mardi\n");
80          break;
81
82      case 4:
83          printf("C'est un mercredi\n");
84          break;
85  }
```

```
86     case 5:
87         printf("C'est un jeudi\n");
88         break;
89
90     case 6:
91         printf("C'est un vendredi\n");
92         break;
93     }
94
95     return 0;
96 }
```

Tout d'abord, nous demandons à l'utilisateur d'entrer une date au format jj/mm/aaaa et nous attribuons chaque partie aux variables `jour`, `mois` et `an`. Ensuite, nous multiplions par 365 la différence d'années séparant l'année fournie de l'an un. Toutefois, il nous faut encore prendre en compte les années bissextiles pour que le nombre de jours obtenus soit correct. Nous ajoutons donc un jour par année bissextile en prenant soin d'appliquer les règles du calendrier en vigueur à la date fournie.

Maintenant, il nous faut ajouter le nombre de jours séparant le mois de janvier du mois spécifié par l'utilisateur. Pour ce faire, nous utilisons la même méthode que celle vue lors de la deuxième étape à *une différence près* : nous vérifions si l'année courante est bissextile afin de retrancher le bon nombre de jours (le mois de février comportant dans ce cas vingt-neuf jours et non vingt-huit).

Enfin, nous utilisons le même code que celui de la première étape.

Ce chapitre nous aura permis de faire une petite pause et de mettre en application ce que nous avons vu dans les chapitres précédents. Reprenons à présent notre route en attaquant la notion de **boucle**.

Dans ce chapitre, nous allons aborder les **boucles**. Une boucle est un moyen de répéter des instructions suivant le résultat d'une condition. Ces structures, dites **itératives**, que nous allons voir dans ce chapitre sont les suivantes.

9.1 La boucle while

La première des boucles que nous allons étudier est la boucle `while` (qui signifie « tant que »). Celle-ci permet de répéter un bloc d'instructions tant qu'une condition est remplie.

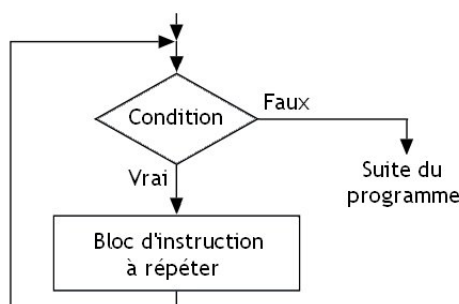


FIGURE 9.1 – Structure While

9.1.1 Syntaxe

La syntaxe de notre boucle `while` est assez simple.

```

1 while (/* Condition */)
2 {
3     /* Bloc d'instructions à répéter */
4 }
  
```

Structure itérative	Action
<i>while...</i>	répète une suite d'instructions tant qu'une condition est respectée.
<i>do... while...</i>	répète une suite d'instructions tant qu'une condition est respectée. Le groupe d'instructions est exécuté au moins une fois.
<i>for...</i>	répète un nombre fixé de fois une suite d'instructions.

Exemple

```

1  #include <stdio.h>
2
3
4  int main(void)
5  {
6      int i = 0;
7
8      while (i < 5)
9      {
10         printf("La variable i vaut %d\n", i);
11         i++;
12     }
13
14     return 0;
15 }

```

```

1  La variable i vaut 0
2  La variable i vaut 1
3  La variable i vaut 2
4  La variable i vaut 3
5  La variable i vaut 4

```

Le fonctionnement est simple à comprendre :

- Au départ, notre variable `i` vaut zéro. Étant donné que zéro est bien inférieur à cinq, la condition est vraie, le corps de la boucle est donc exécuté.
- La valeur de `i` est affichée.
- `i` est augmentée d'une unité et vaut désormais un.
- La condition de la boucle est de nouveau vérifiée.

Ces étapes vont ainsi se répéter pour les valeurs un, deux, trois et quatre. Quand la variable `i` vaudra cinq, la condition sera fausse, et l'instruction `while` sera alors passée.



Dans cet exemple, nous utilisons une variable nommée `i`. Ce nom lui vient d'une contraction du mot anglais *iterator* qui signifie que cette variable sert à l'itération (la répétition) du corps de la boucle. Ce nom est tellement court et explicite qu'il est pour ainsi dire devenu une convention de nommage en C.

9.1.2 Exercice

Essayez de réaliser un programme qui détermine si un nombre entré par l'utilisateur est premier. Pour rappel, un nombre est dit premier s'il n'est divisible que par un et par lui-même. Notez que si un nombre x est divisible par y alors le résultat de l'opération `x % y` est nul.

Indice

Pour savoir si un nombre est premier, il va vous falloir vérifier si celui-ci est uniquement divisible par un et lui-même. Dit autrement, vous allez devoir contrôler qu'aucun nombre compris entre 1 et le nombre entré (tout deux exclus) n'est un diviseur de ce dernier. Pour parcourir ces différentes possibilités, une boucle va vous être nécessaire.

Correction

```

1  #include <stdio.h>
2
3
4  int main(void)

```

```

5  {
6      int nombre;
7      int i = 2;
8
9      printf("Entrez un nombre : ");
10     scanf("%d", &nombre);
11
12     while ((i < nombre) && (nombre % i != 0))
13     {
14         ++i;
15     }
16
17     if (i == nombre)
18     {
19         printf("%d est un nombre premier\n", nombre);
20     }
21     else
22     {
23         printf("%d n'est pas un nombre premier\n", nombre);
24     }
25
26     return 0;
27 }

```

9.2 La boucle do-while

La boucle `do while` fonctionne comme la boucle `while`, à un petit détail près : elle s'exécutera toujours au moins une fois, alors qu'une boucle `while` peut ne pas s'exécuter si la condition est fausse dès le départ.

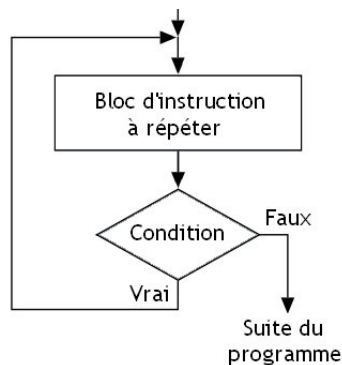


FIGURE 9.2 – Instruction do... while...

9.2.1 Syntaxe

À la différence de la boucle `while`, la condition est placée à la fin du bloc d'instruction à répéter, ce qui explique pourquoi celui-ci est toujours exécuté au moins une fois. Remarquez également la présence d'un point-virgule à la fin de l'instruction qui est obligatoire.

```

1  do
2  {
3      /* Bloc d'instructions à répéter */
4  } while (/* Condition */);

```

9.2.2 Exemple 1

Voici le même code que celui présenté avec l'instruction `while`.

```

1  #include <stdio.h>
2
3
4  int main(void)
5  {
6      int i = 0;
7
8      do
9      {
10         printf("La variable i vaut %d\n", i);
11         ++i;
12     } while (i < 5);
13
14     return 0;
15 }

```

```

1  La variable i vaut 0
2  La variable i vaut 1
3  La variable i vaut 2
4  La variable i vaut 3
5  La variable i vaut 4

```

9.2.3 Exemple 2

Comme nous vous l'avons dit plus haut, une boucle `do while` s'exécute au moins une fois.

```

1  #include <stdio.h>
2
3
4  int main(void)
5  {
6      do
7      {
8         printf("Boucle do-while\n");
9     } while (0);
10
11     return 0;

```

```

1  Boucle do-while

```

Comme vous le voyez, malgré que la condition est fausse (pour rappel, une valeur nulle correspond à une valeur fausse), le corps de la boucle est exécuté une fois puisque la condition n'est évaluée qu'*après* que le bloc d'instructions ait été parcouru.

9.3 La boucle for

9.3.1 Syntaxe

```

1  for (/* Expression 1 */ ; /* Condition */ ; /* Expression 2 */)
2  {
3      /* Instructions à répéter */
4  }

```

Une boucle `for` se décompose en trois parties :

- une expression, qui sera le plus souvent l'initialisation d'une variable ;
 - une condition ;
 - une seconde expression, qui consistera le plus souvent en l'incrément d'une variable.
- Techniquement, une boucle `for` revient en fait à écrire ceci.

```
1  /* Expression 1 */
2
3  while (/* Condition */)
4  {
5      /* Bloc d'instructions à répéter */
6      /* Expression 2 */
7  }
```

9.3.2 Exemple

Le fonctionnement de cette boucle est plus simple à appréhender à l'aide d'un exemple.

```
1  #include <stdio.h>
2
3
4  int main(void)
5  {
6      int i;
7
8      for (i = 0 ; i < 5 ; ++i)
9          printf("la variable i vaut %d\n", i);
10
11     return 0;
12 }
```

```
1  variable vaut 0
2  variable vaut 1
3  variable vaut 2
4  variable vaut 3
5  variable vaut 4
```

Ce qui, comme dit précédemment, revient exactement à écrire cela.

```
1  #include <stdio.h>
2
3
4  int main(void)
5  {
6      int i;
7
8      i = 0;
9
10     while (i < 5)
11     {
12         printf("la variable i vaut %d\n", i);
13         ++i;
14     }
15
16     return 0;
17 }
```



Notez bien que la première expression, `i = 0`, est située *en dehors* du corps de la boucle. Elle n'est donc pas évaluée à chaque tour.

Exercice

Essayez de réaliser un programme qui calcule la somme de tous les nombres compris entre un et n (n étant déterminé par vos soins). Autrement dit, pour un nombre n donné, vous allez devoir calculer $1 + 2 + 3 + \dots + (n - 2) + (n - 1) + n$.

```

1  #include <stdio.h>
2
3
4  int main (void)
5  {
6      const unsigned int n = 250;
7      unsigned int somme = 0;
8      unsigned int i;
9
10     for (i = 1; i <= n; ++i)
11         somme += i;
12
13     printf ("Somme de 1 à %u : %u\n", n, somme);
14     return 0;
15 }

```



Notez qu'il est possible de réaliser cet exercice sans boucle en calculant : $\frac{N \times (N+1)}{2}$.

9.3.3 Plusieurs compteurs

Notez que le nombre de compteurs ou de conditions n'est pas limité, comme le démontre le code suivant.

```

1  for (i = 0, j = 2 ; i < 10 && j < 12; i++, j += 2)

```

Ici, nous définissons deux compteurs `i` et `j` initialisés respectivement à zéro et deux. Le contenu de la boucle est exécuté tant que `i` est inférieur à dix et que `j` est inférieur à douze, `i` étant augmentée de une unité et `j` de deux unités à chaque tour de boucle. Le code est encore assez lisible, cependant la modération est de mise, un trop grand nombre de paramètres rendant la boucle `for` illisible.

9.4 Imbrications

Il est parfaitement possible d'**imbriquer** une ou plusieurs boucles en plaçant une boucle dans le corps d'une autre boucle.

```

1  int i;
2  int j;
3
4  for (i = 0 ; i < 1000 ; ++i)
5  {
6      for (j = i ; j < 1000 ; ++j)
7      {
8          /* Code */
9      }
10 }

```

Cela peut servir par exemple pour déterminer la liste des nombres dont le produit vaut mille.

```

1  #include <stdio.h>
2
3
4  int main(void)
5  {
6      int i;
7      int j;
8
9      for (i = 0 ; i <= 1000 ; ++i)

```

```

10 {
11     for (j = i ; j <= 1000 ; ++j)
12         if (i * j == 1000)
13             printf ("%d * %d = 1000 \n", i, j);
14 }
15
16 return 0;
17 }

```

```

1 1 * 1000 = 1000
2 2 * 500 = 1000
3 4 * 250 = 1000
4 5 * 200 = 1000
5 8 * 125 = 1000
6 10 * 100 = 1000
7 20 * 50 = 1000
8 25 * 40 = 1000

```



Vous n'êtes bien entendu pas tenu d'imbriquer des types de boucles identiques. Vous pouvez parfaitement placer, par exemple, une boucle `while` dans une boucle `for`.

9.5 Boucles infinies

Lorsque vous utilisez une boucle, il y a une chose que vous devez impérativement vérifier : elle doit pouvoir se terminer. Cela paraît évident de prime abord, pourtant il s'agit d'une erreur de programmation assez fréquente qui donne lieu à des **boucles infinies**. Soyez donc vigilants !

L'exemple le plus fréquent est l'oubli d'incrémentation de l'itérateur.

```

1 #include <stdio.h>
2
3 int main(void)
4 {
5     int i = 0;
6
7     while (i < 5)
8     {
9         printf("La variable i vaut %d\n", i);
10        /* Dubli d'incrémentation */
11    }
12
13    return 0;
14 }

```

```

1 La variable i vaut 0
2 La variable i vaut 0
3 La variable i vaut 0
4 ...

```

Ce code continuera jusqu'à ce que l'utilisateur arrête le programme.

9.6 Exercices

9.6.1 Calcul du PGCD de deux nombres

Le PGCD de deux nombres est le plus grand nombre qui peut diviser ces derniers. Par exemple, le PGCD de quinze et douze est trois et celui de vingt-quatre et dix-huit est six.

Pour le calculer, nous devons disposer de deux nombres a et b avec a supérieur à b . Ensuite, nous effectuons la division entière de a par b .

- si le reste est nul, alors nous avons terminé ;
 - si le reste est non nul, nous revenons au début en remplaçant a par b et b par le reste.
- Avec cette explication, vous avez tout ce qu'il vous faut : à vos claviers !

Correction

```

1  #include <stdio.h>
2
3
4  int main (void)
5  {
6      unsigned int a = 46;
7      unsigned int b = 42;
8      unsigned int reste = a % b;
9
10     while (reste != 0)
11     {
12         a = b;
13         b = reste;
14         reste = a % b;
15     }
16
17     printf("%d", b);
18     return 0;
19 }
```

9.6.2 Une overdose de lapins

Au treizième siècle, un mathématicien italien du nom de *Leonardo Fibonacci* posa un petit problème dans un de ses livres, qui mettait en scène des lapins. Ce petit problème mis en avant une suite de nombres particulière, nommée la [suite de Fibonnaci](#), du nom de son inventeur. Il fit les hypothèses suivantes :

- le premier mois, nous plaçons un couple de deux lapins dans un enclos ;
- un couple de lapin ne peut procréer qu'à partir du troisième mois de sa venue dans l'enclos (autrement dit, il ne se passe rien pendant les deux premiers mois) ;
- chaque couple capable de procréer donne naissance à un nouveau couple ;
- enfin, pour éviter tout problème avec la SPA¹, les lapins ne meurent jamais.

Le problème est le suivant : combien y a-t-il de couples de lapins dans l'enclos au n -ième mois ? Le but de cet exercice est de réaliser un petit programme qui fasse ce calcul automatiquement.

Indice

Allez, un petit coup de pouce : suivant l'énoncé, un couple ne donne naissance à un autre couple qu'au début du troisième mois de son apparition. Combien de couple y a-t-il le premier mois ? Un seul. Combien y en a-t-il le deuxième mois ? Toujours un seul. Combien y en a-t-il le troisième mois (le premier couple étant là depuis deux mois) ? Deux. Avec ceci, vous devriez venir à bout du problème.

Correction

```

1  #include <stdio.h>
2
3
4  int main(void)
5  {
6      int a;
7      int b;
```

1. Société Protectrice des Animaux

```

8  int nb_lapins = 1;
9  int const mois = 10;
10 int i;
11
12 a = 0;
13 b = 1;
14
15 for (i = 1; i < mois; ++i)
16 {
17     nb_lapins = a + b;
18     a = b;
19     b = nb_lapins;
20 }
21
22 printf("Au mois %d, il y a %d lapins\n", mois, nb_lapins);
23 return 0;
24 }

```

9.6.3 Des pieds et des mains pour convertir mille miles

Si vous avez déjà voyagé en Grande-Bretagne ou aux États-unis, vous savez que les unités de mesure utilisées dans ces pays sont différentes des nôtres. Au lieu de notre cher système métrique, dont les *stars* sont les centimètres, mètres et kilomètres, nos amis outre-manche et outre-atlantique utilisent le système impérial, avec ses pouces, pieds et *miles*, voire lieues et *furlongs*! Et pour empirer les choses, la conversion n'est pas toujours simple à effectuer de tête... Aussi, la lecture d'un ouvrage tel que *Le Seigneur des Anneaux*, dans lequel toutes les distances sont exprimées en unités impériales, peut se révéler pénible.

Grâce au langage C, nous allons aujourd'hui résoudre tous ces problèmes! Votre mission, si vous l'acceptez, sera d'écrire un programme affichant un tableau de conversion entre *miles* et kilomètres. Le programme ne demande rien à l'utilisateur, mais doit afficher quelque chose comme ceci.

Miles	Km
5	8
10	16
15	24
20	32
25	40
30	48

Autrement dit, le programme compte les kilomètres de cinq en cinq jusqu'à trente et affiche à chaque fois la valeur correspondante en *miles*. Un *mile* vaut exactement 1.609344 km, cependant nous allons utiliser une valeur approchée: huit-cinquièmes de kilomètre (soit 1.6km). Autrement dit, 1 *miles* = $\frac{8}{5}$ km ou (1 km = $\frac{5}{8}$ *miles*).

Correction

```

1  #include <stdio.h>
2
3
4  int main(void)
5  {
6      unsigned miles = 0;
7
8      printf("Miles\tKm\n");
9
10     do
11     {
12         ++miles;
13         printf("%u\t%u\n", miles * 5, miles * 8);
14     } while (miles * 5 < 30);
15
16     return 0;
17 }

```


9.6.4 Puissances de trois

Passons à un exercice un peu plus difficile, du domaine des mathématiques. Essayez de le faire même si vous n'aimez pas les mathématiques.

Vous devez vérifier si un nombre est une puissance de trois, et afficher le résultat. De plus, si c'est le cas, vous devez afficher l'exposant qui va avec.

Indice

Pour savoir si un nombre est une puissance de trois, vous pouvez utiliser le modulo. Attention cependant : si le reste vaut 0, le nombre n'est pas forcément une puissance de trois (par exemple, le reste de la division de 15 par 3 est nul, mais 15 n'est pas une puissance de trois).

Correction

```

1  #include <stdio.h>
2
3
4  /* Un petite explication s'impose, notamment au niveau du for. La première
5  partie qui correspond à l'initialisation de i ne devrait pas vous poser
6  trop de soucis. Ensuite, le i /= 3 sert à diviser i par trois à chaque
7  itération. Au tour de la condition, le principe est simple : tant que le
8  reste de la division de i par 3 est égal à zéro et que i est positif, on
9  incrémente l'exposant. Enfin, pour déterminer si le nombre est une puissance
10 de trois, il suffit de vérifier si i est égal à 1 (essayez avec de petits
11 nombres dans votre tête ou sur papier si vous n'êtes pas convaincu). */
12
13 int main(void)
14 {
15     int number, i;
16     int exposant = 0;
17
18     printf("Veuillez entrez un nombre : ");
19     scanf("%d", &number);
20
21     for (i = number; (i % 3 == 0) && (i > 0); i /= 3)
22     {
23         ++exposant;
24     }
25
26     /* Chaque division successive divise i par 3, donc si nous obtenons finalement
27     i == 1, c'est que le nombre est bien une puissance de 3 */
28
29     if (i == 1)
30     {
31         printf ("%d est égal à 3 ^ %d\n", number, exposant);
32     }
33     else
34     {
35         printf ("%d n'est pas une puissance de 3\n", number);
36     }
37
38     return 0;
39 }
```

9.6.5 La disparition: le retour

Connaissez-vous le roman *La Disparition*? Il s'agit d'un roman français de Georges Perec, publié en 1969. Sa particularité est qu'il ne contient *pas une seule fois* la lettre «e». On appelle ce genre de textes privés d'une lettre des *lipogrammes*. Celui-ci est une prouesse littéraire, car la lettre «e» est la plus fréquente de la langue française: elle représente une lettre sur six en

moyenne! Le roman faisant environ trois cents pages, il a sûrement fallu déployer des trésors d'inventivité pour éviter tous les mots contenant un «e».

Si vous essayez de composer un tel texte, vous allez vite vous rendre compte que vous glissez souvent des «e» dans vos phrases sans même vous en apercevoir. Nous avons besoin d'un vérificateur qui nous sermonnera chaque fois que nous écrirons un «e». C'est là que le langage C entre en scène!

Écrivez un programme qui demande à l'utilisateur de taper une phrase, puis qui affiche le nombre de «e» qu'il y a dans celle-ci. Une phrase se termine toujours par un point «.», un point d'exclamation «!» ou un point d'interrogation «?». Pour effectuer cet exercice, il sera indispensable de lire la phrase caractère par caractère.

```
1  Entrez une phrase : Bonjour, comment allez-vous ?
2  Au moins une lettre 'e' a été repérée (précisément : 2) !
```

Indice

La première chose à faire est d'afficher un message de bienvenue, afin que l'utilisateur sache quel est votre programme. Ensuite, il vous faudra lire les caractères tapés (rappelez-vous les différents formats de la fonction `scanf()`), un par un, *jusqu'à ce qu'un* point (normal, d'exclamation ou d'interrogation) soit rencontré. Dans l'intervalle, il faudra compter chaque «e» qui apparaîtra. Enfin, il faudra afficher le nombre de «e» qui ont été comptés (potentiellement aucun).

Correction

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      unsigned compteur = 0;
6      char c;
7
8      printf("Entrez une phrase se terminant par '.', '!' ou '?' : ");
9
10     do
11     {
12         scanf("%c", &c);
13         if (c == 'e' || c == 'E')
14         {
15             compteur++;
16         }
17     } while (c != '.' && c != '!' && c != '?');
18
19     if (compteur == 0)
20     {
21         printf("Aucune lettre 'e' repérée. Félicitations !\n");
22     }
23     else
24     {
25         printf("Au moins une lettre 'e' a été repérée (précisément : %d) !\n", compteur);
26     }
27
28     return 0;
29 }
```

Les boucles sont assez faciles à comprendre, la seule chose dont il faut se souvenir étant de faire attention de bien avoir une condition de sortie pour ne pas tomber dans une boucle infinie. Le prochain chapitre abordera la notion de **saut**.

Dans les chapitres précédents, nous avons vu comment modifier l'exécution de notre programme en fonction du résultat d'une ou plusieurs conditions. Ainsi, nous avons pu réaliser des tâches plus complexes que de simplement exécuter une suite d'instructions de manière linéaire.

Cette exécution non linéaire est possible grâce à ce que l'on appelle des **sauts**. Un saut correspond au passage d'un point à un autre d'un programme. Bien que cela vous ait été caché, sachez que vous en avez déjà rencontré ! En effet, une instruction `if` réalise par exemple un saut à votre insu.

```
1  if (/* Condition */) { /* Bloc */ }
2
3  /* Suite du programme */
```

Dans le cas où la condition est fausse, l'exécution du programme passe le bloc de l'instruction `if` et exécute ce qui suit. Autrement dit, il y a un **saut** jusqu'à la suite du bloc.

Dans la même veine, une boucle `while` réalise également des sauts.

```
1  while (/* Condition */) { /* Bloc à répéter */ }
2
3  /* Suite du programme */
```

Dans cet exemple, si la condition est vraie, le bloc qui suit est exécuté puis il y a un saut pour revenir à l'évaluation de la condition. Si en revanche elle est fausse, comme pour l'instruction `if`, il y a un saut au-delà du bloc d'instructions.

Tous ces sauts sont cependant automatiques et vous sont cachés. Dans ce chapitre, nous allons voir comment réaliser manuellement des sauts à l'aide de trois instructions : `break`, `continue` et `goto`.

10.1 L'instruction `break`

Nous avons déjà rencontré l'instruction `break` lors de la présentation de l'instruction `switch`, cette dernière permettait de quitter le bloc d'un `switch` pour reprendre immédiatement après. Cependant, l'instruction `break` peut également être utilisée au sein d'une boucle pour stopper son exécution (autrement dit pour effectuer un saut au-delà du bloc à répéter).

10.1.1 Exemple

Le plus souvent, une instruction `break` est employée pour sortir d'une itération lorsqu'une condition (différente de celle contrôlant l'exécution de la boucle) est remplie. Par exemple, si nous souhaitons réaliser un programme qui détermine le plus petit diviseur commun de deux nombres, nous pouvons utiliser cette instruction comme suit.

```

1
Entrez deux nombres : 112 567 le plus petit diviseur de 112 et 567
est 7
Entrez deux nombres : 13 17

```

Comme vous le voyez, la condition principale permet de progresser parmi les diviseurs possibles alors que la seconde détermine si la valeur courante de `i` est un diviseur commun. Si c'est le cas, l'exécution de la boucle est stoppée et le résultat affiché. Dans le cas où il n'y a aucun diviseur commun, la boucle s'arrête lorsque le plus petit des deux nombres est atteint.

10.2 L'instruction continue

L'instruction `continue` permet d'arrêter l'exécution de l'itération courante. Autrement dit, celle-ci vous permet de retourner (sauter) directement à l'évaluation de la condition.

Exemple

Afin d'améliorer un peu l'exemple précédent, nous pourrions passer les cas où le diviseur testé est un multiple de deux (puisque si un des deux nombres n'est pas divisible par deux, il ne peut pas l'être par quatre, par exemple).

Ceci peut s'exprimer à l'aide de l'instruction `continue`.

```

1  #include <stdio.h>
2
3
4  int main(void) { int a; int b; int i; int min;
5
6      printf("Entrez deux nombres : "); scanf("%d %d", &a, &b);
7      min = (a < b) ? a : b;
8
9      for (i = 2; i <= min; ++i) { if (i != 2 && i % 2 == 0)
10         { printf("je passe %d\n", i);
11           continue; } if (a % i == 0 && b % i == 0)
12         { printf("le plus petit diviseur
13           de %d et %d est %d\n", a, b, i);
14           break; } }
15
16     return 0; }

```

```

1  je passe 4 je passe 6 le plus petit diviseur de 112 et 567 est 7

```



Dans le cas de la boucle `for`, l'exécution reprend à l'évaluation de sa deuxième expression (ici `++i`) et non à l'évaluation de la condition (qui a lieu juste après). Il serait en effet mal venu que la variable `i` ne soit pas incrémentée lors de l'utilisation de l'instruction `continue`. Notez bien que les instructions `break` et `continue` n'affecte que l'exécution de la boucle dans laquelle elles sont situées. Ainsi, si vous utilisez l'instruction `break` dans une boucle imbriquée dans une autre, vous sortirez de la première, mais pas de la seconde.

```

1  #include <stdio.h>
2
3
4  int main(void) { int i; int j;
5

```

```

6  for (i = 0 ; i <= 1000 ; ++i) { for (j = i ; j <= 1000 ; ++j) { if
7      (i * j == 1000) { printf ("%d * %d = 1000 \n", i, j);
8          break; /* Quitte la boucle courante, mais pas la
9              première. */ } } }
10
11  return 0; }

```

```

1  1 * 1000 = 1000 2 * 500 = 1000 4 * 250 = 1000 5 * 200 = 1000 8 * 125
2  = 1000 10 * 100 = 1000 20 * 50 = 1000 25 * 40 = 1000

```

10.3 L'instruction goto

Nous venons de voir qu'il était possible de réaliser des sauts à l'aide des instructions `break` et `continue`. Cependant, d'une part ces instructions sont confinées à une boucle ou à une instruction `switch` et, d'autre part, la destination du saut nous est imposée (la condition avec `continue`, la fin du bloc d'instructions avec `break`).

L'instruction `goto` permet de sauter à un point précis du programme que nous aurons déterminé à l'avance. Pour ce faire, le langage C nous permet de marquer des instructions à l'aide d'étiquettes (*labels* en anglais). Une étiquette n'est rien d'autre qu'un nom choisis par nos soins suivi du caractère `:`. Généralement, par soucis de lisibilité, les étiquettes sont placées en retrait des instructions qu'elles désignent.

10.4 Exemple

Reprenons (encore) l'exemple du calcul du plus petit commun diviseur. Ce dernier aurait pu être écrit comme suit à l'aide d'une instruction `goto`.

```

1  #include <stdio.h>
2
3
4  int main(void) { int a; int b; int i; int min;
5
6      printf("Entrez deux nombres : "); scanf("%d %d", &a, &b);
7      min = (a < b) ? a : b;
8
9      for (i = 2; i <= min; ++i) { if (a % i == 0 && b % i == 0)
10         { goto trouve; } }
11
12     return 0; trouve: printf("le plus petit diviseur
13     de %d et %d est %d\n", a, b, i);
14     return 0; }

```

Comme vous le voyez, l'appel à la fonction `printf()` a été marqué avec une étiquette nommée `trouve`. Celle-ci est utilisée avec l'instruction `goto` pour spécifier que c'est à cet endroit que nous souhaitons nous rendre si un diviseur commun est trouvé. Vous remarquerez également que nous avons désormais deux instructions `return`, la première étant exécutée dans le cas où aucun diviseur commun n'est trouvé.

10.5 Le dessous des boucles

Maintenant que vous savez cela, vous devriez être capable de réécrire n'importe quelle boucle à l'aide de cette instruction. En effet, une boucle ne consiste jamais qu'en deux sauts : un vers une condition et l'autre vers l'instruction qui suit le corps de la boucle. Ainsi, les deux codes suivants sont équivalents.

```
1  #include <stdio.h>
2
3
4  int main(void) { int i = 0;
5
6      while (i < 5) { printf("La variable i vaut %d\n", i);
7          i++; }
8
9      return 0; }
```

```
1  #include <stdio.h>
2
3
4  int main(void) { int i = 0;
5
6
7      condition: if (i < 5) { printf("La variable i vaut %d\n", i);
8          i++; goto condition; }
9
10     return 0; }
```

10.6 Goto Hell ?

Bien qu'utile dans certaines circonstances, sachez que l'instruction `goto` est fortement décriée, principalement pour deux raisons :

- mise à part dans des cas spécifiques, il est possible de réaliser la même action de manière plus claire à l'aide de structures de contrôles ;
- l'utilisation de cette instruction peut amener votre code à être plus difficilement lisible et, dans les pire cas, en faire un [code spaghetti](#).

À vrai dire, elle est aujourd'hui surtout utilisée dans le cas de la gestion d'erreur, ce que nous verrons plus tard dans ce cours. Aussi, en attendant, nous vous conseillons d'éviter son utilisation.

Dans le chapitre suivant, nous aborderons la notion de **fonction**.

Nous avons découvert beaucoup de nouveautés dans les chapitres précédents et nos programmes commencent à grossir. C'est pourquoi il est important d'apprendre à les découper en **fonctions**.

11.1 Qu'est-ce qu'une fonction ?

Le concept de fonction ne vous est pas inconnu : `printf()`, `scanf()`, et `main()` sont des **fonctions**.

? Mais qu'est-ce qu'une fonction exactement et quel est leur rôle exactement ?

Une fonction est :

- une suite d'instructions ;
- marquée à l'aide d'un nom (comme une variable finalement) ;
- qui a vocation à être exécutée à plusieurs reprises ;
- qui rassemble des instructions qui permettent d'effectuer une tâche précise (comme afficher du texte à l'écran, calculer la racine carrée d'un nombre, etc).

Pour mieux saisir leur intérêt, prenons un exemple concret.

```
1  #include <stdio.h>
2
3
4  int main(void)
5  {
6      int a;
7      int b;
8      int i;
9      int min;
10
11     printf("Entrez deux nombres : ");
12     scanf("%d %d", &a, &b);
13     min = (a < b) ? a : b;
14
15     for (i = 2; i <= min; ++i)
16     {
17         if (a % i == 0 && b % i == 0)
18         {
19             printf("Le plus petit diviseur de %d et %d est %d\n", a, b, i);
20             break;
21         }
22     }
23
24     return 0;
25 }
```


Ce code, repris du chapitre précédent, permet de calculer le plus petit commun diviseur de deux nombres donnés. Imaginons à présent que nous souhaitions faire la même chose, mais avec deux paires de nombres. Le code ressemblerait alors à ceci.

```

1  #include <stdio.h>
2
3
4  int main(void)
5  {
6      int a;
7      int b;
8      int i;
9      int min;
10
11     printf("Entrez deux nombres : ");
12     scanf("%d %d", &a, &b);
13     min = (a < b) ? a : b;
14
15     for (i = 2; i <= min; ++i)
16     {
17         if (a % i == 0 && b % i == 0)
18         {
19             printf("Le plus petit diviseur de %d et %d est %d\n", a, b, i);
20             break;
21         }
22     }
23
24     printf("Entrez deux autres nombres : ");
25     scanf("%d %d", &a, &b);
26     min = (a < b) ? a : b;
27
28     for (i = 2; i <= min; ++i)
29     {
30         if (a % i == 0 && b % i == 0)
31         {
32             printf("Le plus petit diviseur de %d et %d est %d\n", a, b, i);
33             break;
34         }
35     }
36
37     return 0;
38 }
```

Comme vous le voyez, ce n'est pas très pratique : nous devons recopier les instructions de calcul deux fois, ce qui est assez dommage et qui plus est source d'erreurs. C'est ici que les fonctions entre en jeu en nous permettant par exemple de rassembler les instructions dédiées au calcul du plus petit diviseur commun en un seul point que nous solliciterons autant de fois que nécessaire.



Oui, il est aussi possible d'utiliser une boucle pour éviter la répétition, mais l'exemple aurait été moins parlant.

11.2 Définir et utiliser une fonction

Pour définir une fonction, nous allons devoir donner quatre informations sur celle-ci :

- son **nom** : les règles sont les mêmes que pour les variables ;
- son **corps** (son contenu) : le bloc d'instructions à exécuter ;
- son **type de retour** : le type du résultat de la fonction ;
- d'éventuels **paramètres** : des valeurs reçues par la fonction lors de l'appel.

La syntaxe est la suivante.

```

1  type nom(paramètres)
2  {
```

```

3      /* Corps de la fonction */
4  }

```

Prenons un exemple en créant une fonction qui affiche « bonjour ! » à l'écran.

```

1  #include <stdio.h>
2
3
4  void bonjour(void)
5  {
6      printf("Bonjour !\n");
7  }
8
9
10 int main(void)
11 {
12     bonjour();
13     return 0;
14 }

```

Comme vous le voyez, la fonction se nomme « bonjour » et est composée d'un appel à `printf()`. Reste les deux mots-clés `void` :

- dans le cas du type de retour, il spécifie que la fonction ne retourne rien ;
- dans le cas des paramètres, il spécifie que la fonction n'en reçoit aucun (cela se manifeste lors de l'appel : il n'y a rien entre les parenthèses).

11.2.1 Le type de retour

Le type de retour permet d'indiquer deux choses : si la fonction retourne une valeur et le type de cette valeur.

```

1  #include <stdio.h>
2
3
4  int deux(void)
5  {
6      return 2;
7  }
8
9
10 int main(void)
11 {
12     printf("Retour : %d\n", deux());
13     return 0;

```

```

1  Retour : 2

```

Dans l'exemple ci-dessus, la fonction `deux()` est définie comme retournant une valeur de type `int`. Vous retrouvez l'instruction `return`, une instruction de saut (comme `break`, `continue` et `goto`). Ce `return` arrête l'exécution de la fonction courante et provoque un retour (techniquement, un saut) vers l'appel à cette fonction qui se voit alors attribuer la valeur de retour (s'il y en a une). Autrement dit, dans notre exemple, l'instruction `return 2` stoppe l'exécution de la fonction `deux()` et ramène l'exécution du programme à l'appel qui vaut désormais 2, ce qui donne finalement `printf("Retour : %d\n", 2)`.

11.2.2 Les paramètres

Un paramètre sert à fournir des informations à la fonction lors de son exécution. La fonction `printf()` par exemple récupère ce qu'elle doit afficher dans la console à l'aide de paramètres. Ceux-ci sont définis de la même manière que les variables si ce n'est que les définitions sont séparées par des virgules.

```

1 type nom(type paramètre1, type paramètre2, ...)
2 {
3     /* Corps de la fonction */
4 }

```



Vous pouvez utiliser un maximum de trente et un paramètres, toutefois nous vous conseillons de vous limiter à *cinq* afin de conserver un code concis et lisible.

Maintenant que nous savons tout cela, nous pouvons réaliser une fonction qui calcul le plus petit commun diviseur entre deux nombres et ainsi simplifier l'exemple du dessus.

```

1 #include <stdio.h>
2
3
4 int ppcd(int a, int b)
5 {
6     int min = (a < b) ? a : b;
7     int i;
8
9     for (i = 2; i <= min; ++i)
10         if (a % i == 0 && b % i == 0)
11             return i;
12
13     return 0;
14 }
15
16
17 int main(void)
18 {
19     int a;
20     int b;
21     int resultat;
22
23     printf("Entrez deux nombres : ");
24     scanf("%d %d", &a, &b);
25     resultat = ppcd(a, b);
26
27     if (resultat != 0)
28         printf("Le plus petit diviseur de %d et %d est %d\n", a, b, resultat);
29
30     printf("Entrez deux autres nombres : ");
31     scanf("%d %d", &a, &b);
32     resultat = ppcd(a, b);
33
34     if (resultat != 0)
35         printf("Le plus petit diviseur de %d et %d est %d\n", a, b, resultat);
36
37     return 0;
38 }

```

Plus simple et plus lisible, non ?



Remarquez la présence de deux instructions `return` dans la fonction `ppcd()`. La valeur zéro est retournée afin d'indiquer l'absence d'un diviseur commun.

11.3 Les arguments et les paramètres

À ce stade, il est important de préciser qu'un paramètre est propre à une fonction, il n'est *pas* utilisable en dehors de celle-ci. Par exemple, la variable `a` de la fonction `ppcd()` n'a aucun rapport avec la variable `a` de la fonction `main()`.

Voici un autre exemple plus explicite à ce sujet.

```

1  #include <stdio.h>
2
3
4  void fonction(int nombre)
5  {
6      ++nombre;
7      printf("Variable nombre dans `fonction' : %d\n", nombre);
8  }
9
10
11 int main(void)
12 {
13     int nombre = 5;
14
15     fonction(nombre);
16     printf("Variable nombre dans `main' : %d\n", nombre);
17     return 0;

```

```

1  Variable nombre dans `fonction' : 6
2  Variable nombre dans `main' : 5

```

Comme vous le voyez, les deux variables `nombre` sont bel et bien distinctes. En fait, lors d'un appel de fonction, vous spécifiez des **arguments** à la fonction appelée. Ces arguments ne sont rien d'autres que des expressions dont les résultats seront ensuite affectés aux différents **paramètres** de la fonction.



Notez bien cette différence car elle est très importante : un argument est une *expression* alors qu'un paramètre est une *variable*.

Ainsi, la valeur de la variable `nombre` de la fonction `main()` est passée en argument à la fonction `fonction()` et est ensuite affectée au paramètre `nombre`. La variable `nombre` de la fonction `main()` n'est donc en rien modifiée.

11.4 Les prototypes

Jusqu'à présent, nous avons toujours défini notre fonction *avant* la fonction `main()`. Cela paraît de prime abord logique (nous définissons la fonction avant de l'utiliser), cependant cela est surtout indispensable. En effet, si nous déplaçons la définition après la fonction `main()`, le compilateur se retrouve dans une situation délicate : il est face à un appel de fonction dont il ne sait rien (nombres d'arguments, type des arguments et type de retour). Que faire ? Hé bien, il serait possible de stopper la compilation, mais ce n'est pas ce qui a été retenu, le compilateur va considérer que la fonction retourne une valeur de type `int` et qu'elle reçoit un nombre indéterminé d'arguments.

Toutefois, si cette décision à l'avantage d'éviter un arrêt de la compilation, elle peut en revanche conduire à des problèmes lors de l'exécution si cette supposition du compilateur s'avère inadéquate. Or, il serait pratique de pouvoir définir les fonctions dans l'ordre que nous souhaitons sans se soucier de qui doit être défini avant qui.

Pour résoudre ce problème, il est possible de **déclarer** une fonction à l'aide d'un **prototype**. Celui-ci permet de spécifier le type de retour de la fonction, son nombre d'arguments et leur type, mais ne comporte pas le corps de cette fonction. La syntaxe d'un prototype est la suivante.

```

1  type nom(paramètres);

```

Ce qui donne par exemple ceci.

```

1  #include <stdio.h>
2
3  void bonjour(void);
4
5
6  int main(void)
7  {
8      bonjour();
9      return 0;
10 }
11
12
13 void bonjour(void)
14 {
15     printf("Bonjour !\n");
16 }

```



Notez bien le point-virgule à la fin du prototype qui est obligatoire.



Étant donné qu'un prototype ne comprends pas le corps de la fonction qu'il déclare, il n'est pas obligatoire de préciser le nom des paramètres de celles-ci. Ainsi, le prototype suivant est parfaitement correct.

```

1  int ppcd(int, int);

```

11.5 Variables globales et classes de stockage

11.5.1 Les variables globales

Il arrive parfois que l'utilisation de paramètres ne soit pas adaptée et que des fonctions soient amenées à travailler sur des données qui doivent leur être communes. Prenons un exemple simple : vous souhaitez compter le nombre d'appels de fonction réalisé durant l'exécution de votre programme. Ceci est impossible à réaliser, sauf à définir une variable dans la fonction `main()`, la passé en argument de chaque fonction et de faire en sorte que chaque fonction retourne sa valeur augmentée de un, ce qui est très peu pratique.

À la place, il est possible de définir une variable dite « **globale** » qui sera utilisable par toutes les fonctions. Pour définir une variable globale, il vous suffit de définir une variable *en dehors de tout bloc*, autrement dit en dehors de toute fonction.

```

1  #include <stdio.h>
2
3  void fonction(void);
4
5  int appels = 0;
6
7
8  void fonction(void)
9  {
10     ++appels;
11 }
12
13
14 int main(void)
15 {
16     fonction();
17     fonction();
18     printf("Ce programme a réalisé %d appel(s) de fonction\n", appels);
19     return 0;
20 }

```

1 Ce programme a réalisé 2 appel(s) de fonction

Comme vous le voyez, nous avons simplement placé la définition de la variable *appels* en dehors de toute fonction et *avant* toute définition de fonction de sorte qu'elle soit partagée entres-elles.



Le terme « global » est en fait un peu trompeur étant donné que la variable n'est pas globale au programme, mais tout simplement disponible pour toutes les fonctions du fichier dans lequel elle est située. Ce terme est utilisé en opposition aux paramètres et variables des fonctions qui sont dits « **locaux** ».



N'utilisez les variables globales que lorsque cela vous paraît *vraiment* nécessaire. Ces dernières étant utilisables dans un fichier entier (voire dans plusieurs, nous le verrons un peu plus tard), elles ont tendances à rendre la lecture du code plus difficile.

11.6 Les classes de stockage

Les variables locales et les variables globales ont une autre différence de taille : leur **classe de stockage**. La classe de stockage détermine (entre autre) la **durée de vie** d'un objet, c'est-à-dire le temps durant lequel celui-ci existera en mémoire.

11.6.1 Classe de stockage automatique

Les variables locales sont par défaut de classe de stockage **automatique**. Cela signifie qu'elles sont allouées automatiquement à chaque fois que le bloc auquel elles appartiennent est exécuté et qu'elles sont détruites une fois son exécution terminée.

```

1  int ppcd(int a, int b)
2  {
3      int min = (a < b) ? a : b;
4      int i;
5
6      for (i = 2; i <= min; ++i)
7          if (a % i == 0 && b % i == 0)
8              return i;
9
10     return 0;
11 }
```

Par exemple, à chaque fois que la fonction `ppcd()` est appelée, les variables `a`, `b`, `min` et `i` sont allouées en mémoires et détruites à la fin de l'exécution de la fonction.

11.6.2 Classe de stockage statique

Les variables globales sont *toujours* de classe de stockage **statique**. Ceci signifie qu'elles sont allouées au début de l'exécution du programme et sont détruites à la fin de l'exécution de celui-ci. En conséquence, elles conservent leur valeur tout au long de l'exécution du programme.

Également, à l'inverse des autres variables, celles-ci sont initialisées à zéro si elles ne font pas l'objet d'une initialisation. L'exemple ci-dessous est donc correct et utilise deux variables valant zéro.

```

1  #include <stdio.h>
2
3  int a;
4  double b;
5
6
7  int main(void)
8  {
9      printf("%d, %f\n", a, b);
10     return 0;
11 }

```

```

1  0, 0.000000

```

Petit bémol tout de même : étant donné que ces variables sont créées au début du programme, elles ne peuvent être initialisées qu'à l'aide de *constantes*. La présence de variables au sein de l'expression d'initialisation est donc proscrite.

```

1  #include <stdio.h>
2
3  int a = 20; /* Correct */
4  double b = a; /* Incorrect */
5
6
7  int main(void)
8  {
9      printf("%d, %f\n", a, b);
10     return 0;
11 }

```

11.6.3 Modification de la classe de stockage

Il est possible de modifier la classe de stockage d'une variable automatique en précédant sa définition du mot-clé `static` afin d'en faire une variable statique.

```

1  #include <stdio.h>
2
3
4  int compteur(void)
5  {
6      static int n;
7
8      return ++n;
9  }
10
11
12 int main(void)
13 {
14     compteur();
15     printf("n = %d\n", compteur());
16     return 0;
17 }

```

```

text n = 2

```

11.7 Exercices

11.7.1 Afficher un rectangle

Le premier exercice que nous vous proposons consiste à afficher un rectangle dans la console. Voici ce que devra donner l'exécution de votre programme.

```

1  Donnez la longueur : 5
2  Donnez la largeur : 3
3
4  ***
5  ***
6  ***
7  ***
8  ***

```

11.7.2 Correction

```

1  #include <stdio.h>
2
3  void rectangle(int, int);
4
5
6  int main(void)
7  {
8      int longueur;
9      int largeur;
10
11     printf("Donnez la longueur : ");
12     scanf("%d", &longueur);
13     printf("Donnez la largeur : ");
14     scanf("%d", &largeur);
15     printf("\n");
16     rectangle(longueur, largeur);
17     return 0;
18 }
19
20
21 void rectangle(int longueur, int largeur)
22 {
23     int i;
24     int j;
25
26     for (i = 0; i < longueur; i++)
27     {
28         for (j = 0; j < largeur; j++)
29             printf("*");
30
31         printf("\n");
32     }
33 }

```



Vous pouvez aussi essayer d'afficher le rectangle dans l'autre sens

.

11.8 Afficher un triangle

Même principe, mais cette fois-ci avec un triangle (rectangle). Le programme devra donner ceci.

```

1  Donnez un nombre : 5
2
3  *
4  **
5  ***
6  ****
7  *****

```

Bien entendu, la taille du triangle variera en fonction du nombre entré.

11.8.1 Correction

```

1  #include <stdio.h>
2
3  void triangle(int);
4
5
6  int main(void)
7  {
8      int nombre;
9
10     printf("Donnez un nombre : ");
11     scanf("%d", &nombre);
12     printf("\n");
13     triangle(nombre);
14     return 0;
15 }
16
17 void triangle(int nombre)
18 {
19     int i;
20     int j;
21
22     for (i = 0; i < nombre; i++)
23     {
24         for (j = 0; j <= i; j++)
25             printf("*");
26
27         printf("\n");
28     }
29 }
```

11.9 En petites coupures ?

Pour ce dernier exercice, vous allez devoir réaliser un programme qui reçoit en entrée une somme d'argent et donne en sortie la plus petite quantité de coupures nécessaires pour reconstituer cette somme.

Pour cet exercice, vous utiliserez les coupures suivantes :

- des billets de 100€;
- des billets de 50€;
- des billets de 20€;
- des billets de 10€;
- des billets de 5€;
- des pièces de 2€;
- des pièces de 1€;

Ci dessous un exemple de ce que devra donner votre programme une fois terminé.

```

1  Entrez une somme : 285
2  2 billet(s) de 100.
3  1 billet(s) de 50.
4  1 billet(s) de 20.
5  1 billet(s) de 10.
6  1 billet(s) de 5.
```

11.9.1 Correction

```

1  #include <stdio.h>
2
3
4  int coupure_inferieure(int valeur)
5  {
6      switch (valeur)
7      {
8          case 100:
```

```
9         return 50;
10
11     case 50:
12         return 20;
13
14     case 20:
15         return 10;
16
17     case 10:
18         return 5;
19
20     case 5:
21         return 2;
22
23     case 2:
24         return 1;
25
26     default:
27         return 0;
28     }
29 }
30
31
32 void coupure(int somme)
33 {
34     int valeur;
35     int nb_coupure;
36
37     valeur = 100;
38
39     while (valeur != 0)
40     {
41         nb_coupure = somme / valeur;
42
43         if (nb_coupure > 0)
44         {
45             if (valeur >= 5)
46                 printf("%d billet(s) de %d.\n", nb_coupure, valeur);
47             else
48                 printf("%d pièce(s) de %d.\n", nb_coupure, valeur);
49
50             somme -= nb_coupure * valeur;
51         }
52
53         valeur = coupure_inferieure(valeur);
54     }
55 }
56
57
58 int main(void)
59 {
60     int somme;
61
62     printf("Entrez une somme : ");
63     scanf("%d", &somme);
64     coupure(somme);
65     return 0;
66 }
```

Le prochain chapitre sera l'occasion de mettre en pratique ce que nous venons de voir à l'aide d'un second TP.

TP : une calculatrice basique

Après tout ce que vous venez de découvrir, il est temps de faire une petite pause et de mettre en pratique vos nouveaux acquis. Pour ce faire, rien de tel qu'un exercice récapitulatif : réaliser une calculatrice basique.

12.1 Objectif

Votre objectif sera de réaliser une calculatrice basique pouvant calculer une somme, une soustraction, une multiplication, une division, le reste d'une division entière, une puissance, une factorielle, le PGCD et le PPCD.

Celle-ci attendra une entrée formatée suivant la [notation polonaise inverse](#). Autrement dit, les opérandes d'une opération seront entrés *avant* l'opérateur, par exemple comme ceci pour la somme de quatre et cinq : `4 5 +`.

Elle devra également retenir le résultat de l'opération précédente et déduire l'utilisation de celui-ci en cas d'omission d'un opérande. Plus précisément, si l'utilisateur entre par exemple `5 +`, vous devrez déduire que le premier opérande de la somme est le résultat de l'opération précédente (ou zéro s'il n'y en a pas encore eu).

Chaque opération se verra attribuer un symbole ou une lettre, comme suit :

- addition : `+`;
- soustraction : `-`;
- multiplication : `*`;
- division : `/`;
- reste de la division entière : `%`;
- puissance : `^`;
- factorielle : `!`;
- PGCD : `g`;
- PPCD : `p`.

Le programme doit s'arrêter lorsque la lettre « q » est spécifiée comme opération (avec ou sans opérande).

12.2 Préparation

12.2.1 Précisions concernant scanf



Pourquoi utiliser la notation polonaise inverse et non l'écriture habituelle ?

Parce qu'elle va vous permettre de bénéficier d'une caractéristique intéressante de la fonction `scanf()` : sa valeur de retour. Nous anticipons un peu sur les chapitres suivants, mais sachez que la fonction `scanf()` retourne une valeur entière qui correspond au nombre de conversions réussies. Une conversion est réussie si ce qu'entre l'utilisateur correspond à l'indicateur de conversion.

Ainsi, si nous souhaitons récupérer un entier à l'aide de l'indicateur `d`, la conversion sera réussie si l'utilisateur entre un nombre (par exemple 2) alors qu'elle échouera s'il entre une lettre ou un signe de ponctuation.

Grâce à cela, vous pourrez détecter facilement s'il manque ou non un opérande pour une opération.



Lorsqu'une conversion échoue, la fonction `scanf()` arrête son exécution. Aussi, s'il y avait d'autres conversions à effectuer après celle qui a avorté, elles ne seront pas réalisées

```
1 double a;
2 double b;
3 char op;
4
5 scanf("%lf %lf %c", &a, &b, &op);
```

Dans le code ci-dessus, si l'utilisateur entre `7 *`, la fonction `scanf()` retournera 1 et n'aura lu que le nombre 7. Il sera nécessaire de l'appeler une seconde fois pour que le symbole `*` soit récupéré.



Petit bémol tout de même : les symboles `+` et `-` sont considérés comme des débuts de nombre valables (puisque vous pouvez par exemple entrer -2). Dès lors, si vous souhaitez additionner ou soustraire un nombre au résultat de l'opération précédente, vous devrez doubler ce symbole. Pour ajouter cinq cela donnera donc : `5 ++`.

12.2.2 Les puissances

Pour élever un nombre à une puissance donnée (autrement dit, pour calculer x^y), nous allons avoir besoin d'une nouvelle partie de la bibliothèque standard dédiée aux fonctions mathématiques de base. Le fichier d'en-tête de la bibliothèque mathématique se nomme `<math.h>` et contient, entre autre, la déclaration de la fonction `pow()`.

```
1 #include <math.h>
2
3 int main() {
4     double x, y;
5     // ...
6 }
```

Cette dernière prends deux arguments : la base et l'exposant.



L'utilisation de la bibliothèque mathématique requiert d'ajouter l'option `-lm` lors de la compilation, par exemple comme ceci : `gcc -lm main.c`

12.2.3 La factorielle

La factorielle d'un nombre est égal au produit des nombres entiers *positifs et non nuls* inférieurs ou égaux à ce nombre. La factorielle de quatre équivaut donc à `1 * 2 * 3 * 4`, donc vingt-quatre. Cette fonction n'est pas fournie par la bibliothèque standard, il vous faudra donc la programmer par vous-même (pareil pour le PGCD et le PPCD que nous avons vus dans les chapitres précédents).



Par convention, la factorielle de zéro est égale à un.

12.2.4 Exemple d'utilisation

```

1  > 5 6 +
2  11.000000
3  > 4 *
4  44.000000
5  > 2 /
6  22.000000
7  > 5 2 %
8  1.000000
9  > 2 5 ^
10 32.000000
11 > 1 ++
12 33.000000
13 > 5 !
14 120.000000

```

12.3 Derniers conseils

Nous vous conseillons de récupérer les nombres sous forme de `double`. Cependant, gardez bien à l'esprit que certaines opérations ne peuvent s'appliquer qu'à des entiers : le reste de la division entière, la factorielle, le PGCD et le PPCD. Il vous sera donc nécessaire d'effectuer des conversions.

Également, notez bien que la factorielle ne s'applique qu'à *un seul* opérande à l'inverse de toutes les autres opérations.

Bien, vous avez à présent toutes les cartes en main : au travail !

12.4 Correction

Alors ? Pas trop secoué ? Bien, voyons à présent la correction.

```

1  #include <math.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  unsigned long pgcd(unsigned long, unsigned long);
6  unsigned long ppdc(unsigned long, unsigned long);
7  unsigned long factorielle(unsigned long);
8
9
10 unsigned long pgcd(unsigned long a, unsigned long b)
11 {
12     unsigned long r = a % b;
13
14     while (r != 0)
15     {
16         a = b;
17         b = r;
18         r = a % b;
19     }
20
21     return b;
22 }
23
24
25 unsigned long ppdc(unsigned long a, unsigned long b)
26 {
27     unsigned long i;
28     unsigned long min = (a < b) ? a : b;
29

```

```

30     for (i = 2; i <= min; ++i)
31         if (a % i == 0 && b % i == 0)
32             return i;
33
34     return 0;
35 }
36
37
38 unsigned long factorielle(unsigned long a)
39 {
40     unsigned long i;
41     unsigned long r = 1;
42
43     for (i = 2; i <= a; ++i)
44         r *= i;
45
46     return r;
47 }
48
49
50 int
51 main(void)
52 {
53     double a;
54     double b;
55     double res = 0;
56     int n;
57     char op;
58
59     while (1)
60     {
61         printf("> ");
62         n = scanf("%lf %lf %c", &a, &b, &op);
63
64         if (n <= 1)
65         {
66             scanf("%c", &op);
67             b = a;
68             a = res;
69         }
70         if (op == 'q')
71             break;
72
73         switch (op)
74         {
75             case '+':
76                 res = a + b;
77                 break;
78
79             case '-':
80                 res = a - b;
81                 break;
82
83             case '*':
84                 res = a * b;
85                 break;
86
87             case '/':
88                 res = a / b;
89                 break;
90
91             case '%':
92                 res = (unsigned long)a % (unsigned long)b;
93                 break;
94
95             case '^':
96                 res = pow(a, b);
97                 break;
98
99             case '!':
100                 res = factorielle((n == 0) ? a : b);
101                 break;
102

```

```
103     case 'g':
104         res = pgcd(a, b);
105         break;
106
107     case 'p':
108         res = ppcd(a, b);
109         break;
110     }
111
112     printf("%lf\n", res);
113 }
114 return 0;
115 }
```

Commençons par la fonction `main()`. Nous définissons plusieurs variables :

- `a` et `b`, qui représentent les éventuels opérandes fournis ;
- `res`, qui correspond au résultat de la dernière opération réalisée (ou zéro s'il n'y en a pas encore eu) ;
- `n`, qui est utilisée pour retenir le retour de la fonction `scanf()` ; et
- `op`, qui retient l'opération demandée.

Ensuite, nous entrons dans une boucle infinie (la condition étant toujours vraie puisque valant un) où nous demandons à l'utilisateur d'entrer l'opération à réaliser et les éventuels opérandes. Nous vérifions ensuite si un seul opérande est fourni ou aucune (ce qui se déduit, respectivement, d'un retour de la fonction `scanf()` valant un ou zéro). Si c'est le cas, nous appelons une seconde fois `scanf()` pour récupérer l'opérateur. Puis, la valeur de `a` est attribuée à `b` et la valeur de `res` à `a`.

Si l'opérateur utilisé est `q`, alors nous quittons la boucle et par la même occasion le programme. Notez que nous n'avons pas pu effectuer cette vérification dans le corps de l'instruction `switch` qui suit puisque l'instruction `break` nous aurait fait quitter celui-ci et non la boucle.

Enfin, nous réalisons l'opération demandée au sein de l'instruction `switch`, nous stockons le résultat dans la variable `res` et l'affichons. Remarquez que l'utilisation de conversions explicites n'a été nécessaire que pour le calcul du reste de la division entière. En effet, dans les autres cas (par exemple lors de l'affectation à la variable `res`), il y a des conversions implicites.



Nous avons utilisé le type `unsigned long` lors des calculs nécessitant des nombres entiers afin de disposer de la plus grande capacité possible et parce que l'usage de nombres négatifs n'a pas beaucoup d'intérêt dans ce cadre. Cependant, l'usage de nombres non signés n'est obligatoire que pour la fonction factorielle (puisque celle-ci n'opère que sur des nombres strictement positifs).

Ce chapitre nous aura permis de revoir la plupart des notions des chapitres précédents. Dans le chapitre suivant, nous verrons comment découper nos projets en plusieurs fichiers.

Ce chapitre est la suite directe de celui consacré aux fonctions : nous allons voir comment découper nos projets en plusieurs fichiers. En effet, même si l'on découpe bien son projet en fonctions, ce dernier est difficile à relire si tout est contenu dans le même fichier. Ce chapitre a donc pour but de vous apprendre à découper vos projets efficacement.

13.1 Portée et masquage

13.1.1 La notion de portée

Avant de voir comment diviser nos programmes en plusieurs fichiers, il est nécessaire de vous présenter une notion importante, celle de **portée**. La portée d'une variable ou d'une fonction est la partie du programme où cette dernière est utilisable. Il existe plusieurs types de portées, cependant nous n'en verrons que deux :

- au niveau d'un bloc ;
- au niveau d'un fichier.

Au niveau d'un bloc

Une portée au niveau d'un bloc signifie qu'une variable n'est utilisable, visible que de sa déclaration jusqu'à la fin du bloc dans lequel elle est déclarée. Illustration.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      {
6          int nombre = 3;
7          printf("%d\n", nombre);
8      }
9
10     /* Incorrect ! */
11     printf("%d\n", nombre);
12     return 0;
13 }
```

Dans ce code, la variable `nombre` est déclarée dans un sous-bloc. Sa portée est donc limitée à ce dernier et elle ne peut pas être utilisée en dehors.

13.1.2 Au niveau d'un fichier

Une portée au niveau d'un fichier signifie qu'une variable n'est utilisable, visible, que de sa déclaration jusqu'à la fin du fichier dans lequel elle est déclarée. En fait, il s'agit de la portée

des variables « globales » dont nous avons parlé dans le chapitre sur les fonctions.

```

1  #include <stdio.h>
2
3  int nombre = 3;
4
5  int triple(void)
6  {
7      return nombre * 3;
8  }
9
10 int main(void)
11 {
12     nombre = triple();
13     printf("%d\n", nombre);
14     return 0;
15 }
```

Dans ce code, la variable `nombre` a une portée au niveau du fichier et peut par conséquent être aussi bien utilisée dans la fonction `triple()` que dans la fonction `main()`.

13.2 La notion de masquage

En voyant les deux types de portées, vous vous êtes peut-être posé la question suivante : que se passe-t-il s'il existe plusieurs variables et/ou plusieurs fonctions de même nom ? Hé bien, cela dépend de la portée de ces dernières. Si elles ont la même portée comme dans l'exemple ci-dessous, alors le compilateur sera incapable de déterminer à quelle variable ou à quelle fonction le nom fait référence et, dès lors, retournera une erreur.

```

1  int main(void)
2  {
3      int nombre = 10;
4      int nombre = 20;
5
6      return 0;
7  }
```

En revanche, si elles ont des portées différentes, alors celle ayant la portée la plus faible sera privilégiée, on dit qu'elle **masque** celle(s) de portée plus élevée. Autrement dit, dans l'exemple qui suit, c'est la variable du bloc de la fonction `main()` qui sera affichée.

```

1  #include <stdio.h>
2
3  int nombre = 10;
4
5  int main(void)
6  {
7      int nombre = 20;
8
9      printf("%d\n", nombre);
10     return 0;
11 }
```

Notez que nous disons : « celle(s) de portée plus élevée » car les variables déclarées dans un sous-bloc ont une portée plus faible que celle déclarée dans un bloc supérieur. Ainsi, le code ci-dessous est parfaitement valide et affichera 30.

```

1  #include <stdio.h>
2
3  int nombre = 10;
4
5  int main(void)
```

```

6 {
7     int nombre = 20;
8
9     if (nombre == 20)
10    {
11        int nombre = 30;
12
13        printf("%d\n", nombre);
14    }
15
16    return 0;
17 }

```

13.3 Diviser pour mieux régner

13.3.1 Les fonctions

Dans l'extrait précédent, nous avons, entre autres, créé une fonction `triple()` que nous avons placée dans le même fichier que la fonction `main()`. Essayons à présent de les répartir dans deux fichiers distincts. Pour ce faire, il vous suffit de créer un second fichier avec l'extension « `.c` ». Dans notre cas, il s'agira de « `main.c` » et de « `autre.c` ».

```

1  /* Fichier autre.c */
2
3  int triple(int nombre)
4  {
5      return nombre * 3;
6  }

```

```

1  /* Fichier main.c */
2
3  int main(void)
4  {
5      int nombre = triple(3);
6      return 0;
7  }

```

La compilation se réalise de la même manière qu'auparavant, si ce n'est qu'il vous est nécessaire de spécifier les deux noms de fichier : `zcc main.c autre.c`. À noter que vous pouvez également utiliser une forme raccourcie : `zcc *.c`, où `*.c` correspond à tous les fichiers portant l'extension « `.c` » du dossier courant.

Si vous testez ce code, vous aurez droit à un bel avertissement de votre compilateur du type « *implicit declaration of function 'triple'* ». Quel est le problème ? Le problème est que la fonction `triple()` n'est pas déclarée dans le fichier `main.c` et que le compilateur ne la connaît donc pas lorsqu'il compile le fichier. Pour corriger cette situation, nous devons déclarer la fonction en signalant au compilateur que cette dernière se situe dans un autre fichier. Pour ce faire, nous allons inclure le prototype de la fonction `triple()` dans le fichier `main.c` en le précédant du mot-clé `extern`, qui signifie que la fonction est externe au fichier.

```

1  /* Fichier autre.c */
2
3  int triple(int nombre)
4  {
5      return nombre * 3;

```

```

1  /* Fichier main.c */
2
3  extern int triple(int nombre);
4
5  int main(void)
6  {

```

```

7   int nombre = triple(3);
8
9   return 0;
10 }
```

En terme technique, on dit que la fonction `triple()` est **définie** dans le fichier « autre.c » (car c'est là que se situe le corps de la fonction) et qu'elle est **déclarée** dans le fichier « main.c ». Sachez qu'une fonction ne peut être définie qu'*une seule et unique fois*.

Pour information, notez que le mot-clé `extern` est facultatif devant un prototype (il est implicitement inséré par le compilateur). Nous vous conseillons cependant de l'utiliser, dans un soucis de clarté et de symétrie avec les déclarations de variables (voyez ci-dessous).

13.3.2 Les variables

La même méthode peut être appliquée aux variables, mais *uniquement à celle ayant une portée au niveau d'un fichier*. Également, à l'inverse des fonctions, il est plus difficile de distinguer une définition d'une déclaration de variable (elles n'ont pas de corps comme les fonctions). La règle pour les différencier est qu'une déclaration sera précédée du mot-clé `extern` alors que la définition non. C'est à vous de voir dans quel fichier vous souhaitez définir la variable, mais elle ne peut être définie qu'*une seule et unique fois*. Enfin, sachez que seule la définition peut comporter une initialisation. Ainsi, cet exemple est tout à fait valide.

```

1  /* Fichier autre.c */
2
3  int nombre = 10;    /* Une définition */
4  extern int autre;   /* Une déclaration */
```

```

1  /* Fichier main.c */
2  extern int nombre;  /* Une déclaration */
3  int autre = 10;     /* Une définition */
```

Alors que celui-ci, non.

```

1  /* Fichier autre.c */
2
3  int nombre = 10;    /* Il existe une autre définition */
4  extern int autre = 10; /* Une déclaration ne peut pas comprendre une initialisation */
```

```

1  /* Fichier main.c */
2
3  int nombre = 20;    /* Il existe une autre définition */
4  int autre = 10;     /* Une définition */
```

13.4 On m'aurait donc menti ?

Nous vous avons dit plus haut qu'il n'était possible de définir une variable ou une fonction qu'une seule fois, mais en fait, ce n'est pas tout à fait vrai ... Il est possible de rendre une variable (ayant une portée au niveau d'un fichier) ou une fonction locale à un fichier en précédant sa définition du mot-clé `static`. De cette manière, la variable ou la fonction est interne au fichier où elle est définie et n'entre pas en conflit avec les autres variables ou fonctions locales à d'autres fichiers. La contrepartie est que la variable ou la fonction ne peut être utilisée que dans le fichier où elle est définie (c'est assez logique). Ainsi, l'exemple suivant est tout à fait correct et affichera 20.

```

1  /* Fichier autre.c */
2  static int nombre = 10;
```

```

1  /* Fichier main.c */
2  #include <stdio.h>
3
4  static int nombre = 20;
5
6  int main(void)
7  {
8      printf("%d\n", nombre);
9      return 0;
10 }
```



Ne confondez pas l'utilisation du mot-clé `static` visant à modifier la classe de stockage d'une variable automatique avec celle permettant de limiter l'utilisation d'une variable globale à un seul fichier !

13.5 Les fichiers d'en-têtes

Pour terminer ce chapitre, il ne nous reste plus qu'à voir les fichiers d'en-têtes.

Jusqu'à présent, lorsque vous voulez utiliser une fonction ou une variable définie dans un autre fichier, vous insérez sa déclaration dans le fichier ciblé. Seulement voilà, si vous utilisez dix fichiers et que vous décidez un jour d'ajouter ou de supprimer une fonction ou une variable ou encore de modifier une déclaration, vous vous retrouvez Gros-Jean comme devant et vous êtes bon pour modifier les dix fichiers, ce qui n'est pas très pratique...

Pour résoudre ce problème, on utilise des fichiers d'en-têtes (d'extension « .h »). Ces derniers contiennent conventionnellement des déclarations de fonctions et de variables et sont inclus via la directive `#include` dans les fichiers qui utilisent les fonctions et variables en question.



Les fichiers d'en-têtes n'ont pas besoin d'être spécifiés lors de la compilation, ils seront automatiquement inclus.

La structure d'un fichier d'en-tête est généralement de la forme suivante.

```

1  #ifndef CONSTANCE_H
2  #define CONSTANCE_H
3
4  /* Les déclarations */
5
6  #endif
```

Les directives du préprocesseur sont là pour éviter les inclusions multiples : vous devez les utiliser pour chacun de vos fichiers d'en-têtes. Vous pouvez remplacer `CONSTANTE` par ce que vous voulez, le plus simple et le plus fréquent étant le nom de votre fichier, par exemple `AUTRE_H` si votre fichier se nomme « autre.h ». Voici un exemple d'utilisation de fichiers d'en-têtes.

```

1  /* Fichier d'en-tête autre.h */
2
3  #ifndef AUTRE_H
4  #define AUTRE_H
5
6  extern int triple(int nombre);
7
8  #endif
```

```

1  /* Fichier source autre.c */
2
3  #include "autre.h"
```

```
4
5 int triple(int nombre)
6 {
7     return nombre * 3;
8 }
```

```
1 /* Fichier source main.c */
2
3 #include "autre.h"
4
5 int main(void)
6 {
7     int nombre = triple(3);
8     return 0;
```

Plusieurs remarques à propos de ce code :

- dans la directive d'inclusion, les fichiers d'en-têtes sont entre guillemets et non entre crochets comme les fichiers d'en-têtes de la bibliothèque standard ;
- les fichiers sources et d'en-têtes correspondants portent le même nom ;
- nous vous conseillons d'inclure le fichier d'en-tête dans le fichier source correspondant (dans mon cas « autre.h » dans « autre.c ») afin d'éviter des problèmes de portée.

Dans le chapitre suivant, nous aborderons un point essentiel que nous verrons en deux temps : la gestion d'erreurs.

Dans les chapitres précédents, nous vous avons présenté des exemples simplifiés afin de vous familiariser avec le langage. Aussi, nous avons pris soin de ne pas effectuer de vérifications quant à d'éventuelles rencontres d'erreurs.

Mais à présent, c'est fini! Vous disposez désormais d'un bagage suffisant pour affronter la dure réalité d'un programmeur : des fois, il y a des trucs qui foirent et il est nécessaire de le prévoir. Nous allons voir comment dans ce chapitre.

14.1 Détection d'erreurs

La première chose à faire pour gérer d'éventuelles erreurs lors de l'exécution, c'est avant tout de les détecter. Par exemple, quand vous exécutez une fonction et qu'une erreur a lieu lors de son exécution, celle-ci doit vous prévenir d'une manière ou d'une autre. Et elle peut le faire de deux manières différentes.

14.1.1 Valeurs de retour

Nous l'avons vu dans les chapitres précédents : certaines fonctions, comme `scanf()`, retournent un nombre (souvent un entier) alors qu'elles ne calculent pas un résultat comme la fonction `pow()` par exemple. Vous savez dans le cas de `scanf()` que cette valeur représente le nombre de conversions réussies, cependant cela va plus loin que cela : cette valeur vous signifie si l'exécution de la fonction s'est bien déroulée.

Scanf

En fait, la fonction `scanf()` retourne le nombre de conversions réussies ou un nombre inférieur si elles n'ont pas toutes été réalisées ou, enfin, *un nombre négatif en cas d'erreur*.

Ainsi, si nous souhaitons récupérer deux entiers et être certains que `scanf()` les a récupérés, nous pouvons utiliser le code suivant.

```
1  #include <stdio.h>
2
3
4  int main(void)
5  {
6      int x;
7      int y;
8
9      printf("Entrez deux nombres : ");
10
11     if (scanf("%d %d", &x, &y) == 2)
12         printf("Vous avez entre : %d et %d\n", x, y);
```



```

13
14     return 0;
15 }

```

```

1  Entrez deux nombres : 1 2
2  Vous avez entre : 1 et 2
3
4  Entrez deux nombres : 1 a

```

Comme vous pouvez le constater, le programme n'exécute pas l'affichage des nombres dans le dernier cas, car `scanf()` n'a pas réussi à réaliser deux conversions.

Main

Maintenant que vous savez cela, regarder bien votre fonction `main()`.

```

1  int main(void)
2  {
3      return 0;
4  }

```

Vous ne voyez rien qui vous interpelle ? :)


Oui, vous avez bien vu, elle retourne un entier qui, comme pour `scanf()`, sert à indiquer la présence d'erreur. En fait, il y a deux valeurs possibles :

- `EXIT_SUCCESS` (ou zéro, cela revient au même), qui indique que tout s'est bien passé ; et
- `EXIT_FAILURE`, qui indique un échec du programme.


Ces deux constantes sont définies dans l'en-tête `<stdlib.h>`.

Les autres fonctions

Sachez que `scanf()`, `printf()` et `main()` ne sont pas les seules fonctions qui retournent des entiers, en fait quasiment toutes les fonctions de la bibliothèque standard le font.

 Ok, mais je fais comment pour savoir ce que retourne une fonction ?

À l'aide de la documentation. Vous disposez de [la norme](#) (enfin, du brouillon de celle-ci) qui reste la référence ultime, sinon vous pouvez également utiliser un moteur de recherche avec la requête `man nom_de_fonction` afin d'obtenir les informations dont vous avez besoin.

 Si vous êtes anglophobe, une traduction française de diverses descriptions est disponible à [cette adresse](#), vous les trouverez à la section trois.

14.1.2 Variable globale `errno`

Le retour des fonctions est un vecteur très pratique pour signaler une erreur. Cependant, il n'est pas toujours utilisable. En effet, nous avons vu lors du second TP la fonction mathématique `pow()`. Or, cette dernière utilise *déjà* son retour pour transmettre le résultat d'une opération. Comment faire dès lors pour signaler un problème ?

Une première idée serait d'utiliser une valeur particulière, comme zéro par exemple. Toutefois, ce n'est pas satisfaisant puisque, dans le cas de la fonction `pow()`, elle peut parfaitement retourner zéro lors d'un fonctionnement normal. Que faire alors ?

Dans une telle situation, il ne reste qu'une seule solution : utiliser un autre canal, en l'occurrence une variable globale. La bibliothèque standard fournit une variable globale nommée `errno` (elle est déclarée dans l'en-tête `<errno.h>`) qui permet à différentes fonctions d'indiquer une erreur en modifiant la valeur de celle-ci.



Une valeur de zéro indique qu'aucune erreur n'est survenue.

Les fonctions mathématiques recourent abondamment à cette fonction. Prenons l'exemple suivant.

```

1  #include <errno.h>
2  #include <stdio.h>
3
4
5  int main(void)
6  {
7      double x;
8
9      errno = 0;
10     x = pow(-1, 0.5);
11
12     if (errno == 0)
13         printf("x = %f\n", x);
14
15     return 0;
16 }
```

L'appel revient à demander le résultat de l'expression $-1^{\frac{1}{2}}$, autrement dit, de cette expression : $\sqrt{-1}$, ce qui est impossible dans l'ensemble des réels. Aussi, la fonction `pow()` modifie la variable `errno` pour vous signifier qu'elle n'a pas pu calculer l'expression demandée.

Une petite précision concernant ce code et la variable `errno` : celle-ci doit *toujours* être mise à zéro *avant* d'appeler une fonction qui est susceptible de la modifier, ceci afin de vous assurer qu'elle ne contient pas la valeur qu'une autre fonction lui a assignée auparavant. Imaginez que vous ayez précédemment appelé la fonction `pow()` et que cette dernière a échoué, si vous l'appellez à nouveau, la valeur de `errno` sera toujours celle assignée lors de l'appel précédent.



Notez que la bibliothèque standard ne prévoit en fait que deux valeurs d'erreur possibles pour `errno` : `EDOM` (pour le cas où le résultat d'une fonction mathématique est impossible) et `ERANGE` (en cas de dépassement de capacité, nous y reviendrons plus tard). Ces deux constantes sont définies dans l'en-tête `<errno.h>`.

14.2 Prévenir l'utilisateur

Savoir qu'une erreur s'est produite, c'est bien, le signaler à l'utilisateur, c'est mieux. Ne laissez pas votre utilisateur dans le vide, s'il se passe quelque chose, dites le lui.

```

1  #include <stdio.h>
2
3
4  int main(void)
5  {
6      int x;
7      int y;
8
9      printf("Entrez deux nombres : ");
10
11     if (scanf("%d %d", &x, &y) == 2)
12         printf("Vous avez entré : %d et %d\n", x, y);
13     else
14         printf("Vous devez saisir deux nombres !\n");
15
16     return 0;
17 }
```

```
1  Entrez deux nombres : a b
2  Vous devez saisir deux nombres !
3
4  Entrez deux nombres : 1 2
5  Vous avez entre : 1 et 2
```

Simple, mais tellement plus agréable.

14.3 Un exemple d'utilisation des valeurs de retour

Maintenant que vous savez tout cela, il vous est possible de modifier le code utilisant la fonction `scanf()` pour vérifier si celle-ci a réussi et, si ce n'est pas le cas, préciser à l'utilisateur qu'une erreur est survenue *et* quitter la fonction `main()` en retournant la valeur `EXIT_FAILURE`.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4
5  int main(void)
6  {
7      int x;
8      int y;
9
10     printf("Entrez deux nombres : ");
11
12     if (scanf("%d %d", &x, &y) != 2)
13     {
14         printf("Vous devez saisir deux nombres !\n");
15         return EXIT_FAILURE;
16     }
17
18     printf("Vous avez entré : %d et %d\n", x, y);
19     return 0;
20 }
```

Ceci nous permet de réduire un peu la taille de notre code en éliminant directement les cas d'erreurs.

Bien, vous voilà à présent fin prêt pour la deuxième partie du cours et ses *vrais* exemples. Plus de pitié donc : gare à vos fesses si vous ne vérifiez pas le comportement des fonctions que vous appelez !

Deuxième partie

Agrégats, mémoire et fichiers

Dans ce chapitre, nous allons aborder une notion centrale du langage C : les pointeurs.

Les pointeurs constituent ce qui est appelé une **fonctionnalité bas niveau**, c'est-à-dire un mécanisme qui nécessite de connaître quelques détails sur le fonctionnement d'un ordinateur pour être compris et utilisé correctement. Dans le cas des pointeurs, il s'agira surtout de disposer de quelques informations sur la mémoire vive.

15.1 Présentation

Avant de vous présenter le concept de pointeur, un petit rappel concernant la mémoire s'impose (n'hésitez pas à relire le chapitre sur les variables si celui-ci s'avère insuffisant).

Souvenez-vous : toute donnée manipulée par l'ordinateur est stockée dans sa **mémoire**, plus précisément dans une de ses différentes mémoires (registre(s), mémoire vive, disque(s) dur(s), etc.). Cependant, pour utiliser une donnée, nous avons besoin de savoir où elle se situe, nous avons besoin d'une **référence** vers cette donnée. Dans la plupart des cas, cette référence est en fait une **adresse mémoire** qui indique la position de la donnée dans la mémoire vive.

15.1.1 Les pointeurs

Si l'utilisation des références peut être implicites (c'est par exemple le cas lorsque vous manipulez des variables), il est des cas où elle doit être explicite. C'est à cela que servent les **pointeurs** : ce sont des variables dont le contenu est une adresse mémoire (une référence, donc).

15.1.2 Utilité des pointeurs

Techniquement, il y a trois utilisations majeures des pointeurs en C :

- le passage de références à des fonctions ;
- la manipulation de données complexes ;
- l'allocation dynamique de mémoire.

Passage de références à des fonctions

Rappelez-vous du chapitre sur les fonctions : lorsque vous fournissez un argument lors d'un appel, la valeur de celui-ci est affectée au paramètre correspondant, paramètre qui est une variable propre à la fonction appelée. Toutefois, il est parfois souhaitable de modifier une variable de la fonction *appelante*. Dès lors, plutôt que de passer la valeur de la variable en argument, c'est une référence vers celle-ci qui sera envoyée à la fonction.

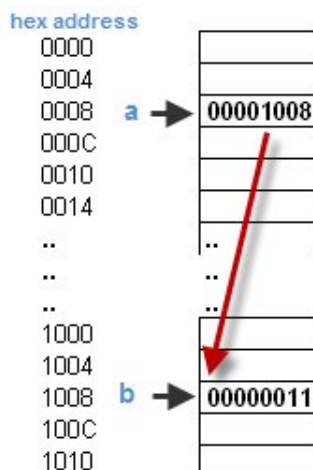


FIGURE 15.1 – Exemple, avec une variable a qui est un pointeur sur une variable b

Manipulation de données complexes

Jusqu'à présent, nous avons manipulé des données simples : `int`, `double`, `char`, etc. Cependant, le C nous permet également d'utiliser des données plus complexes qui sont en fait des **agrégats** (un regroupement si vous préférez) de données simples. Or, il n'est possible de manipuler ces agrégats qu'en les parcourant données simples par données simples, ce qui requiert de disposer d'une référence vers les données qui le composent.



Nous verrons les agrégats plus en détails lorsque nous aborderons les structures et les tableaux.

15.1.3 L'allocation dynamique de mémoire

Il n'est pas toujours possible de savoir quelle quantité de mémoire sera utilisée par un programme. En effet, si vous prenez le cas d'un logiciel de dessin, ce dernier ne peut pas prévoir quelle sera la taille des images qu'il va devoir manipuler. Pour palier à ce problème, les programmes recourent au mécanisme de **l'allocation dynamique de mémoire** : ils demandent de la mémoire au système d'exploitation lors de leur exécution. Pour que cela fonctionne, le seul moyen est que le système d'exploitation fournisse au programme une référence vers la zone allouée.

15.2 Déclaration et initialisation

La syntaxe pour déclarer un pointeur est la suivante.

```
1 type *nom_du_pointeur;
```

Par exemple, si nous souhaitons créer un pointeur sur `int` (c'est-à-dire un pointeur pouvant stocker l'adresse d'un objet de type `int`) et que nous voulons le nommer « ptr », nous devons écrire ceci.

```
1 int *ptr;
```

L'astérisque peut être entourée d'espaces et placée n'importe où entre le type et l'identificateur. Ainsi, les trois définitions suivantes sont identiques.

```

1 int *ptr;
2 int * ptr;
3 int* ptr;

```



Notez bien qu'un pointeur est toujours typé. Autrement dit, vous aurez toujours un pointeur sur (ou vers) un objet d'un certain type (`int`, `double`, `char`, etc.).

15.2.1 Initialisation

Un pointeur, comme une variable, ne possède pas de valeur par défaut, il est donc important de l'initialiser pour éviter d'éventuels problèmes. Pour ce faire, il est nécessaire de recourir à l'**opérateur d'adressage** (ou de référencement) : `&` qui permet d'obtenir l'adresse d'un objet. Ce dernier se place derrière l'objet dont l'adresse souhaite être obtenue. Par exemple comme ceci.

```

1 int a = 10;
2 int *p;
3
4 p = &a;

```

Ou, plus directement, comme cela.

```

1 int a = 10;
2 int *p = &a;

```

Faites bien attention à ne pas mélanger différents types de pointeurs ! Un pointeur sur `int` n'est pas le même qu'un pointeur sur `long` ou qu'un pointeur sur `double`. De même, n'affectez l'adresse d'un objet qu'à un pointeur du même type.



```

1 int a;
2 double b;
3 int *p = &b; /* faux */
4 int *q = &a; /* correct */
5 double *r = p; /* faux */

```

15.2.2 Pointeur nul

Vous souvenez-vous du chapitre sur la gestion d'erreur ? Dans ce dernier, nous vous avons dit que, le plus souvent, les fonctions retournaient une valeur particulière en cas d'erreur. *Quid* de celles qui retournent un pointeur ? Existe-t-il une valeur spéciale qui puisse représenter une erreur ou bien sommes-nous condamnés à utiliser une variable globale comme `errno` ?

Heureusement pour nous, il existe un cas particulier : les pointeurs nuls. Un pointeur nul est tout simplement un pointeur contenant une adresse invalide. Cette adresse invalide dépend de votre système d'exploitation, mais elle est la même pour tous les pointeurs nuls. Ainsi, deux pointeurs nuls ont une valeur égale.

Pour obtenir cette adresse invalide, il vous suffit de convertir explicitement zéro vers le type de pointeur voulu. Ainsi, le pointeur suivant est un pointeur nul.

```

1 int *p = (int *)0;

```




Rappelez-vous qu'il y a conversion implicite vers le type de destination dans le cas d'une affectation. La conversion est donc superflue dans ce cas-ci.

La constante NULL

Afin de clarifier un peu les codes sources, il existe une constante définie dans l'en-tête `<stddef.h>` : `NULL`. Celle-ci peut être utilisée partout où un pointeur nul est attendu *sauf comme argument de la fonction `printf()`* (nous verrons pourquoi plus tard dans ce cours).

```
1 int *p = NULL; /* Un pointeur nul. */
```

15.3 Utilisation

15.3.1 Indirection (ou déréférencement)

Maintenant que nous savons récupérer l'adresse d'un objet et l'affecter à un pointeur, voyons le plus intéressant : accéder à cet objet ou le modifier via le pointeur. Pour y parvenir, nous avons besoin de l'**opérateur d'indirection** (ou de déréférencement) : `*`.



Le symbole `*` n'est pas celui de la multiplication ?

Si, c'est aussi le symbole de la multiplication. Toutefois, à l'inverse de l'opérateur de multiplication, l'opérateur d'indirection ne prends qu'un seul opérande (il n'y a donc pas de risque de confusion).

L'opérateur d'indirection attend un pointeur comme opérande et se place juste derrière celui-ci. Une fois appliqué, ce dernier nous donne accès à la valeur de l'objet référencé par le pointeur, aussi bien pour la lire que pour la modifier.

Dans l'exemple ci-dessous, nous accédons à la valeur de la variable `a` via le pointeur `p`.

```
1 int a = 10;
2 int *p = &a;
3
4 printf("a = %d\n", *p);
```

```
1 a = 10
```

À présent, modifions la variable `a` à l'aide du pointeur `p`.

```
1 int a = 10;
2 int *p = &a;
3
4 *p = 20;
5 printf("a = %d\n", *p);
```

```
1 a = 20
```



Comme pour n'importe quelle variable, il est possible de déclarer un pointeur comme constant. Cependant, puisqu'un pointeur référence un objet, il peut également être déclaré comme un pointeur vers un objet constant. Pour ce faire, la position du mot-clé `const` est importante. Si le mot-clé est devant l'identificateur et derrière le symbole `*`, alors il s'agit d'un pointeur constant.

```
1 int * const ptr; /* Un pointeur constant sur int. */
```

Si le mot-clé est devant le symbole `*` et derrière le type référencé, alors il s'agit d'un pointeur vers un objet constant.



```
int const *ptr; /* Pointeur sur int constant. */
```

Enfin, ces deux notations peuvent être combinées pour créer un pointeur constant vers un objet constant.

```
1 int const * const ptr; /* Pointeur constant sur int constant. */
```

15.3.2 Passage comme argument

Voici un exemple de passage de pointeurs en arguments d'une fonction.

```
1 #include <stdio.h>
2
3 void test(int *pa, int *pb)
4 {
5     *pa = 10;
6     *pb = 20;
7 }
8
9
10 int main(void)
11 {
12     int a;
13     int b;
14     int *pa = &a;
15     int *pb = &b;
16
17     test(&a, &b);
18     test(pa, pb);
19     printf("a = %d, b = %d\n", a, b);
20     printf("a = %d, b = %d\n", *pa, *pb);
21     return 0;
22 }
```

```
1 a = 10, b = 20
2 a = 10, b = 20
```

Remarquez que les appels `test(&a, &b)` et `test(pa, pb)` réalisent la même opération.

15.4 Retour de fonction

Pour terminer, sachez qu'une fonction peut également retourner un pointeur. Cependant, faites attention : *l'objet référencé par le pointeur doit toujours exister au moment de son utilisation* ! L'exemple ci-dessous est donc incorrect étant donnée que la variable `n` est de classe de stockage automatique et qu'elle n'existe donc plus après l'appel à la fonction `ptr()`.

```
1 #include <stdio.h>
2
3
4 int *ptr(void)
5 {
6     int n;
7 }
```

```

8   return &n;
9   }
10
11
12  int main(void)
13  {
14      int *p = ptr();
15
16      *p = 10;
17      printf("%d\n", *p);
18      return 0;
19  }

```

L'exemple devient correct si `n` est de classe de stockage statique.

15.5 Pointeur de pointeur

Au même titre que n'importe quel autre objet, un pointeur a lui aussi une adresse. Dès lors, il est possible de créer un objet pointant sur ce pointeur : un pointeur de pointeur.

```

1  int a = 10;
2  int *pa = &a;
3  int **pp = &pa;

```

Celui-ci s'utilise de la même manière qu'un pointeur si ce n'est qu'il est possible d'opérer deux indirections : une pour atteindre le pointeur référencé et une seconde pour atteindre la variable sur laquelle pointe le premier pointeur.

```

1  #include <stdio.h>
2
3
4  int main(void)
5  {
6      int a = 10;
7      int *pa = &a;
8      int **pp = &pa;
9
10     printf("a = %d\n", **pp);
11     return 0;
12 }

```



Ceci peut continuer à l'infini pour concevoir des pointeurs de pointeur de pointeur de pointeur de... Bref, vous avez compris le principe.

15.6 Pointeurs génériques et affichage

Le type `void`

Vous avez déjà rencontré le mot-clé `void` lorsque nous avons parlé des fonctions, ce dernier permet d'indiquer qu'une fonction n'utilise aucun paramètre et/ou ne retourne aucune valeur. Toutefois, nous n'avons pas tout dit à son sujet : `void` est en fait un type, au même titre que `int` ou `double`. o_O



Et il représente quoi ce type, alors ?

Hum... rien (d'où son nom). :-°

En fait, il s'agit d'un type dit « **incomplet** », c'est à dire que la taille de ce dernier n'est pas

calculable et qu'il n'est pas utilisable dans des expressions. Quel est l'intérêt de la chose me direz-vous ? Permettre de créer des pointeurs « **génériques** » (ou « universels »).

En effet, nous venons de vous dire qu'un pointeur devait toujours être typé. Cependant, cela peut devenir gênant si vous souhaitez créer une fonction qui doit pouvoir travailler avec n'importe quel type de pointeur (nous verrons un exemple très bientôt). C'est ici que le type `void` intervient : un pointeur sur `void` est considéré comme un pointeur générique, ce qui signifie qu'il peut référencer n'importe quel type d'objet.

En conséquence, il est possible d'affecter n'importe quelle adresse d'objet à un pointeur sur `void` et d'affecter un pointeur sur `void` à n'importe quel autre pointeur (et inversement).

```
1 int a;  
2 double b;  
3 void *p;  
4 double *r;  
5  
6 p = &a; /* correct */  
7 p = &b; /* correct */  
8 r = p; /* correct */
```

15.7 Afficher une adresse

Il est possible d'afficher une adresse à l'aide de l'indicateur de conversion `p` de la fonction `printf()`. Ce dernier attend en argument un pointeur sur `void`. Vous voyez ici l'intérêt d'un pointeur générique : un seul indicateur suffit pour afficher tous les types de pointeurs.

Notez que l'affichage s'effectue le plus souvent en hexadécimal.

```
1 int a;  
2 int *p = &a;  
3  
4 printf("%p == %p\n", (void *)&a, (void *)p);
```

Tant que nous y sommes, profitons en pour voir quelle est l'adresse invalide de notre système.

```
1 printf("%p\n", (void *)0);
```

Oui, le plus souvent, il s'agit de zéro.

15.8 Exercice

Pour le moment, tout ceci doit sans doute vous paraître quelques peu abstrait et sans doute inutile. Toutefois, rassurez-vous, cela vous semblera plus clair après les chapitres suivants.

En attendant, nous vous proposons un petit exercice mettant en pratique les pointeurs : programmez une fonction nommée « swap », dont le rôle est d'échanger la valeur de deux variables de type `int`. Autrement dit, la valeur de la variable « a » doit devenir celle de « b » et la valeur de « b », celle de « a ».

15.9 Correction

```
1 #include <stdio.h>  
2  
3 void swap(int *, int *);  
4  
5
```

```
6 void swap(int *pa, int *pb)
7 {
8     int tmp;
9
10    tmp = *pa;
11    *pa = *pb;
12    *pb = tmp;
13 }
14
15
16 int main(void)
17 {
18     int a = 10;
19     int b = 20;
20
21     swap(&a, &b);
22     printf("a = %d, b = %d\n", a, b);
23     return 0;
24 }
```

Nous en avons fini avec les pointeurs, du moins, pour le moment.

En effet, les pointeurs sont omniprésents en langage C et nous n'avons pas fini d'en entendre parler. Mais pour l'heure, nous allons découvrir une des fameuses données complexes dont nous avons parlé en début de chapitre : les **structures**.

Dans le chapitre précédent, nous vous avons dit que les pointeurs étaient entre autres utiles pour manipuler des données complexes. Les structures sont les premières que nous allons étudier.

Une **structure** est un regroupement de plusieurs objets, de types différents ou non. *Grosso modo*, une structure est finalement une boîte qui regroupe plein de données différentes.

16.1 Définition, initialisation et utilisation

16.1.1 Définition d'une structure

Une structure étant un regroupement d'objets, la première chose à réaliser est la description de celle-ci (techniquement, sa **définition**), c'est-à-dire préciser de quel(s) objet(s) cette dernière va se composer.

La syntaxe de toute définition est la suivante.

```
1 struct étiquette
2 {
3     /* Objet(s) composant(s) la structure. */
4 };
```

Prenons un exemple concret : vous souhaitez demander à l'utilisateur deux mesures de temps sous la forme heure(s) :minute(s) :seconde(s).milliseconde(s) et lui donner la différence entre les deux en seconde. Vous pourriez utiliser six variables pour stocker ce que vous fournit l'utilisateur, toutefois cela reste assez lourd. À la place, nous pourrions représenter chaque mesure à l'aide d'une structure composée de trois objets : un pour les heures, un pour les minutes et un pour les secondes.

```
1 struct temps {
2     unsigned heures;
3     unsigned minutes;
4     double secondes;
5 };
```

Comme vous le voyez, nous avons donné un nom (plus précisément, une **étiquette**) à notre structure : « temps ». Les règles à respecter sont les mêmes que pour les noms de variable et de fonction.

Pour le reste, la composition de la structure est décrite à l'aide d'une suite de déclarations de variable. Ces différentes déclarations constituent les **membres** ou **champs** de la structure. Notez bien qu'il s'agit de *déclarations* et non de définitions, l'utilisation d'initialisations est donc exclue.

Enfin, notez la présence d'un point-virgule *obligatoire* à la fin de la définition de la structure.



Une structure ne peut pas comporter plus de cent vingt-sept membres.

16.1.2 Définition d'une variable de type structure

Une fois notre structure décrite, il ne nous reste plus qu'à créer une variable de ce type. Pour ce faire, la syntaxe est la suivante.

```
1 struct étiquette identificateur;
```

La méthode est donc la même que pour définir n'importe quelle variable, si ce n'est que le type de la variable est précisé à l'aide du mot-clé `struct` et de l'étiquette de la structure. Avec notre exemple de la structure `temps`, cela donne ceci.

```
1 #include <stdio.h>
2
3 struct temps {
4     unsigned heures;
5     unsigned minutes;
6     double secondes;
7 };
8
9
10 int main(void)
11 {
12     struct temps t;
13
14     return 0;
15 }
```

16.1.3 Initialisation

Comme pour n'importe quelle autre variable, il est possible d'initialiser une variable de type structure dès sa définition. Toutefois, à l'inverse des autres, l'initialisation s'effectue à l'aide d'une liste fournissant une valeur pour chaque membre de la structure.

L'exemple ci-dessous initialise le membre `heures` à 1, `minutes` à 45 et `secondes` à 30.560.

```
1 struct temps t = { 1, 45, 30.560 };
```



Dans le cas où vous ne fournissez pas un nombre suffisant de valeurs, les membres oubliés seront initialisés à zéro ou, s'il s'agit de pointeurs, seront des pointeurs nuls.

16.1.4 Accès à un membre

L'accès à un membre d'une structure se réalise à l'aide de la variable de type structure et de l'opérateur `.` suivi du nom du champ visé.

```
1 variable.membre
```

Cette syntaxe peut être utilisée aussi bien pour obtenir la valeur d'un champ que pour en modifier le contenu. L'exemple suivant effectue donc la même action que l'initialisation présentée précédemment.

```
1 t.heures = 1;
2 t.minutes = 45;
3 t.secondes = 30.560;
```

16.1.5 Exercice

Afin d'assimiler tout ceci, voici un petit exercice.

Essayez de réaliser ce qui a été décrit plus haut : demandez à l'utilisateur de vous fournir deux mesures de temps sous la forme heure(s) :minute(s) :seconde(s).milliseconde(s) et donnez lui la différence en seconde entre celles-ci.

Voici un exemple d'utilisation.

```
1 Première mesure (hh:mm:ss.xxx) : 12:45:50.640
2 Deuxième mesure (hh:mm:ss.xxx) : 13:30:35.480
3 Il y a 2684.840 seconde(s) de différence.
```

Correction

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 struct temps {
5     unsigned heures;
6     unsigned minutes;
7     double secondes;
8 };
9
10
11
12 int main(void)
13 {
14     struct temps t1;
15     struct temps t2;
16
17     printf("Première mesure (hh:mm:ss) : ");
18
19     if (scanf("%u:%u:%lf", &t1.heures, &t1.minutes, &t1.secondes) != 3)
20     {
21         printf("Mauvaise saisie\n");
22         return EXIT_FAILURE;
23     }
24
25     printf("Deuxième mesure (hh:mm:ss) : ");
26
27     if (scanf("%u:%u:%lf", &t2.heures, &t2.minutes, &t2.secondes) != 3)
28     {
29         printf("Mauvaise saisie\n");
30         return EXIT_FAILURE;
31     }
32
33     t1.minutes += t1.heures * 60;
34     t1.secondes += t1.minutes * 60;
35     t2.minutes += t2.heures * 60;
36     t2.secondes += t2.minutes * 60;
37
38     printf("Il y a %.3f seconde(s) de différence.\n",
39           t2.secondes - t1.secondes);
40     return 0;
41 }
```

16.2 Structures et pointeurs

Certains d'entre vous s'en étaient peut-être doutés : s'il existe un objet d'un type, il doit être possible de créer un pointeur vers un objet de ce type. Si oui, sachez que vous aviez raison. :)

```
1 struct temps *p;
```

La définition ci-dessus crée un pointeur `p` vers un objet de type `struct temps`.

16.2.1 Accès via un pointeur

L'utilisation d'un pointeur sur structure est un peu plus complexe que celle d'un pointeur vers un type de base. En effet, il y a deux choses à gérer : l'accès via le pointeur et l'accès à un membre. Intuitivement, vous combineriez sans doute les opérateurs `*` et `.` comme ceci.

```
1 *p.heures = 1;
```

Toutefois, cette syntaxe ne correspond pas à ce que nous voulons car l'opérateur `.` s'applique *prioritairement* à l'opérateur `*`. Autrement dit, le code ci-dessus accède au champ `heures` et tente de lui appliquer l'opérateur d'indirection, ce qui est incorrect puisque le membre `heures` est un entier non signé.

Pour résoudre ce problème, nous devons utiliser des parenthèses afin que l'opérateur `.` soit appliqué *après* le déréférencement, ce qui donne la syntaxe suivante.

```
1 (*p).heures = 1;
```

Cette écriture étant un peu lourde, le C fournit un autre opérateur qui combine ces deux opérations : l'opérateur `->`.

```
1 p->heures = 1;
```

Le code suivant initialise donc la structure `t` via le pointeur `p`.

```
1 struct temps t;
2 struct temps *p = &t;
3
4 p->heures = 1;
5 p->minutes = 45;
6 p->secondes = 30.560;
```

16.2.2 Adressage

Il est important de préciser que l'opérateur d'adressage peut s'appliquer aussi bien à une structure qu'à un de ses membres. Ainsi, dans l'exemple ci-dessous, nous définissons un pointeur `p` pointant sur la structure `t` et un pointeur `q` pointant sur le champ `heures` de la structure `t`.

```
1 struct temps t;
2 struct temps *p = &t;
3 int *q = &t.heures;
```

16.2.3 Pointeurs sur structures et fonctions

Indiquons enfin que l'utilisation de pointeurs est particulièrement propice dans le cas du passage de structures à des fonctions. En effet, rappelez-vous, lorsque vous fournissez un argument lors d'un appel de fonction, la valeur de celui-ci est affectée au paramètre correspondant. Cette règle s'applique également aux structures : la valeur de chacun des membres est copiée une à une. Dès lors, si la structure passée en argument comporte beaucoup de champs, la copie risque d'être longue. L'utilisation d'un pointeur évite ce problème puisque seule la valeur du pointeur (autrement dit, l'adresse vers la structure) sera copiée et non toute la structure.

16.3 Portée et déclarations

Dans les exemples précédents, nous avons toujours placés notre définition de structure en dehors de toute fonction. Cependant, sachez que celle-ci peut être circonscrite à un bloc de sorte de limiter sa **portée**, comme pour les définitions de variables et les déclarations de variables et fonctions.

Dans l'exemple ci-dessous, la structure `temps` ne peut être utilisée que dans le bloc de la fonction `main()` et les éventuels sous-blocs qui la composent.

```

1  int main(void)
2  {
3      struct temps {
4          unsigned heures;
5          unsigned minutes;
6          double secondes;
7      };
8
9      struct temps t;
10
11     return 0;
12 }
```

Notez qu'il est possible de combiner une définition de structure et une définition de variable. En effet, une définition de structure n'étant rien d'autre que la définition d'un nouveau type, celle-ci peut être placée là où est attendu un type dans une définition de variable. Avec l'exemple précédent, cela donne ceci.

```

1  int main(void)
2  {
3      struct temps {
4          unsigned heures;
5          unsigned minutes;
6          double secondes;
7      } t1;
8      struct temps t2;
9
10     return 0;
11 }
```

Il y a trois définitions dans ce code : celle du type `struct temps`, celle de la variable `t1` et celle de la variable `t2`. Après sa définition, le type `struct temps` peut tout à fait être utilisé pour définir d'autres variables de ce type, ce qui est le cas de `t2`.



Notez qu'il est possible de condenser la définition du type `struct temps` et de la variable `t1` sur une seule ligne, comme ceci.

```

1  struct temps { unsigned heures; unsigned minutes; double secondes; } t1;
```

S'agissant d'une définition de variable, il est également parfaitement possible de l'initialiser.

```

1  int main(void)
2  {
3      struct temps {
4          unsigned heures;
5          unsigned minutes;
6          double secondes;
7      } t1 = { 1, 45, 30.560 };
8      struct temps t2;
9
10     return 0;
11 }
```

Enfin, précisons que l'étiquette d'une structure peut être omise lors de sa définition. Néanmoins, cela ne peut avoir lieu que si la définition de la structure est combinée avec une définition de variable. C'est assez logique étant donné qu'il ne peut pas être fait référence à cette définition à l'aide d'une étiquette.

```

1  int main(void)
2  {
3      struct {
4          unsigned heures;
5          unsigned minutes;
6          double secondes;
7      } t;
8
9      return 0;
10 }
```

16.3.1 Déclarations

Jusqu'à présent, nous avons parlé de définitions de structures, toutefois, comme pour les variables et les fonctions, il existe également des déclarations de structures. Une déclaration de structure est en fait une définition sans le corps de la structure.

```

1  struct temps;
```

Quel est l'intérêt de la chose me direz-vous ? Résoudre deux types de problèmes : les structures interdépendantes et les structures comportant un ou des membres qui sont des pointeurs vers elle-même.

16.3.2 Les structures interdépendantes

Deux structures sont interdépendantes lorsque l'une comprend un pointeur vers l'autre et inversement.

```

1  struct a {
2      struct b *p;
3  };
4
5  struct b {
6      struct a *p;
7  };
```

Voyez-vous le problème ? Si le type `struct b` ne peut être utilisé qu'après sa définition, alors le code ci-dessus est faux et le cas de structures interdépendantes est insoluble. Heureusement, il est possible de **déclarer** le type `struct b` afin de pouvoir l'utiliser *avant* sa définition.

✗ Une déclaration de structure crée un type dit **incomplet** (comme le type `void`). Dès lors, il ne peut pas être utilisé pour définir une variable (puisque les membres qui composent la structure sont inconnus). Ceci n'est utilisable que pour définir des pointeurs.

Le code ci-dessous résout le « problème » en déclarant le type `struct b` avant son utilisation.

```

1  struct b;
2
3  struct a {
4      struct b *p;
5  };
6
7  struct b {
8      struct a *p;
9  };
```

Nous avons entouré le mot « problème » de guillemets car le premier code que nous vous avons montré n'en pose en fait aucun et compile sans sourciller. :-°

En fait, afin d'éviter ce genre d'écritures, le langage C prévoit l'ajout de **déclarations implicites**. Ainsi, lorsque le compilateur rencontre un pointeur vers un type de structure qu'il ne connaît pas, il ajoute implicitement une déclaration de cette structure juste avant. Ainsi, le premier et le deuxième code sont équivalents si ce n'est que le premier comporte une déclaration implicite et non une déclaration explicite.

16.3.3 Structure qui pointe sur elle-même

Le deuxième cas où les déclarations de structure s'avèrent nécessaires est celui d'une structure qui comporte un pointeur vers elle-même.

```
1 struct a {
2     struct a *p;
3 };
```

De nouveau, si le type `struct a` n'est utilisable qu'après sa définition, c'est grillé. Toutefois, comme pour l'exemple précédent, le code est correct, mais pas tout à fait pour les mêmes raisons. Dans ce cas ci, le type `struct a` est connu car nous sommes en train de le définir. Techniquement, dès que la définition commence, le type est déclaré.

 Notez que, comme les définitions, les déclarations (implicites ou non) ont une portée.

16.4 Un peu de mémoire

Savez-vous comment sont représentées les structures en mémoire ?

Les membres d'une structure sont placés les uns après les autres en mémoire. Par exemple, prenons cette structure.

```
1 struct exemple
2 {
3     double flottant;
4     char lettre;
5     unsigned entier;
6 };
```

Si nous supposons qu'un `double` a une taille de huit octets, un `char` de un octet, et un `unsigned int` de quatre octets, voici ce que devrait donner cette structure en mémoire.

 Les adresses spécifiées dans le schéma sont fictives et ne servent qu'à titre d'illustration.

16.4.1 L'opérateur sizeof

Voyons à présent comment déterminer cela de manière plus précise, en commençant par la taille des types. L'opérateur `sizeof` permet de connaître la taille en multiplètes (*bytes* en anglais) de son opérande. Cet opérande peut être soit un type (qui doit alors être entre parenthèses), soit une expression (auquel cas les parenthèses sont facultatives).

Le résultat de cet opérateur est de type `size_t`. Il s'agit d'un type entier *non signé* défini dans l'en-tête `<stddef.h>` qui est capable de contenir la taille de n'importe quel objet. L'exemple ci-dessous utilise l'opérateur `sizeof` pour obtenir la taille des types de bases.

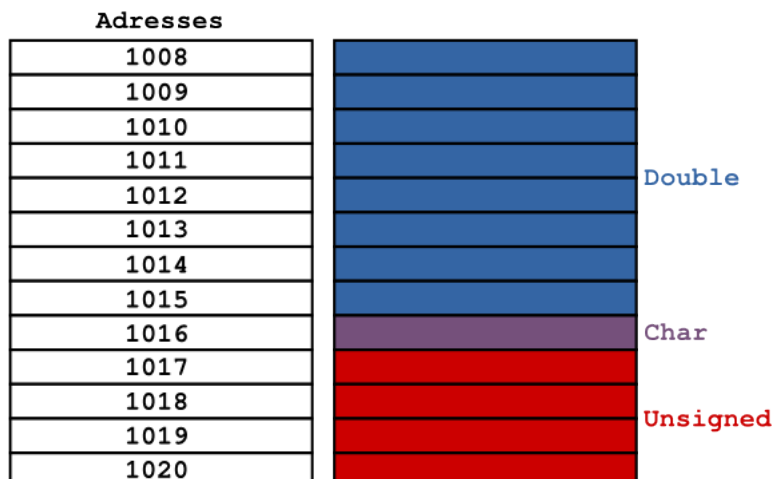


FIGURE 16.1 – Représentation en mémoire de la structure

```

1  #include <stddef.h>
2  #include <stdio.h>
3
4
5  int main(void)
6  {
7      double f;
8
9      printf("char: %u\n", (unsigned)sizeof(char));
10     printf("short: %u\n", (unsigned)sizeof(short));
11     printf("int : %u\n", (unsigned)sizeof(int));
12     printf("long : %u\n", (unsigned)sizeof(long));
13     printf("float : %u\n", (unsigned)sizeof(float));
14     printf("double : %u\n", (unsigned)sizeof(double));
15     printf("long double : %u\n", (unsigned)sizeof(long double));
16
17     printf("int : %u\n", (unsigned)sizeof 5);
18     printf("double : %u\n", (unsigned)sizeof f);
19     return 0;
20 }

```

```

1  char : 1
2  short : 2
3  int : 4
4  long : 8
5  float : 4
6  double : 8
7  long double : 16
8  int : 4
9  double : 8

```

i Le type `char` a toujours une taille de un multiplié.

Malheureusement pour nous, la fonction `printf()` ne fournit pas d'indicateur de conversion pour le type `size_t`¹. Dès lors, nous devons recourir à une conversion explicite afin de pouvoir utiliser un autre indicateur. En l'occurrence, nous avons choisi le type `unsigned int` étant donné que la taille des types de base est très petite et qu'il n'y a en conséquence pas de risque de dépasser sa capacité.

1. Depuis la norme C99, il existe un indicateur de conversion `zu` qui permet d'afficher une expression de type `size_t`.

Remarquez que les parenthèses ne sont pas obligatoires dans le cas où l'opérande de l'opérateur `sizeof` est une expression (dans notre exemple : 5 et `f`).



Il est parfaitement possible que vous n'obteniez pas les mêmes valeurs que nous, celles-ci dépendent de votre machine.

Ceci étant fait, voyons à présent ce que donne la taille de la structure présentée plus haut. En toute logique, elle devrait être égale à la somme des tailles de ses membres, chez nous : 13 (8 + 1 + 4).

```

1  #include <stddef.h>
2  #include <stdio.h>
3
4  struct exemple
5  {
6      double flottant;
7      char lettre;
8      unsigned int entier;
9  };
10
11
12  int main(void)
13  {
14      printf("struct exemple : %u\n", (unsigned)sizeof(struct exemple));
15      return 0;
16  }
```

```

1  struct exemple : 16
```

Ah ! Il semble que nous avons loupé quelque chose... :-°

Pourquoi obtenons nous seize et non treize, comme attendu ? Pour répondre à cette question, nous allons devoir plonger un peu dans les entrailles de notre machine.

16.4.2 Alignement en mémoire

Le processeur et la mémoire vive communiquent entre eux à l'aide d'un canal appelé un **bus mémoire**. Les communications à travers le bus mémoire s'opèrent à l'initiative du processeur qui dispose d'instructions spécifiques pour déplacer des données de la mémoire vers un registre et inversement. Ce canal dispose d'une capacité de transport limitée et ne peut en conséquence transmettre qu'un nombre fixé de multiplètes.

Toutefois, bien que la mémoire vive soit composée de multiplètes, le processeur utilise *toujours* la capacité maximale du bus et lit ou écrit dans celle-ci par blocs de la taille du bus. Dès lors, le processeur ne va pas voir la mémoire vive comme une suite de multiplètes, mais comme une suite de blocs de la taille du bus qui sont le plus souvent appelés des **mots**.

Quel rapport avec notre structure me direz-vous ? Supposons que notre processeur lise des blocs de quatre octets à la fois. Si notre structure faisait treize octets, voici comment le processeur la verrait.

Le premier champ est réparti sur plusieurs mots et remplit complètement ces mots. Cette situation ne pose pas de problèmes : soit le processeur charge la donnée en plusieurs fois (ce que la plupart des processeurs savent faire automatiquement), soit le compilateur prévoit plusieurs instructions de chargements.

Par contre, le membre de type `unsigned int` est problématique : vu que sa taille est égale à celle d'un mot, son transfert devrait être réalisé en une seule fois. Toutefois, dans notre exemple, le membre est à califourchon sur deux mots. De ce fait, pour obtenir sa valeur, le processeur devrait :

- charger le troisième mot et en extraire les bons octets ;

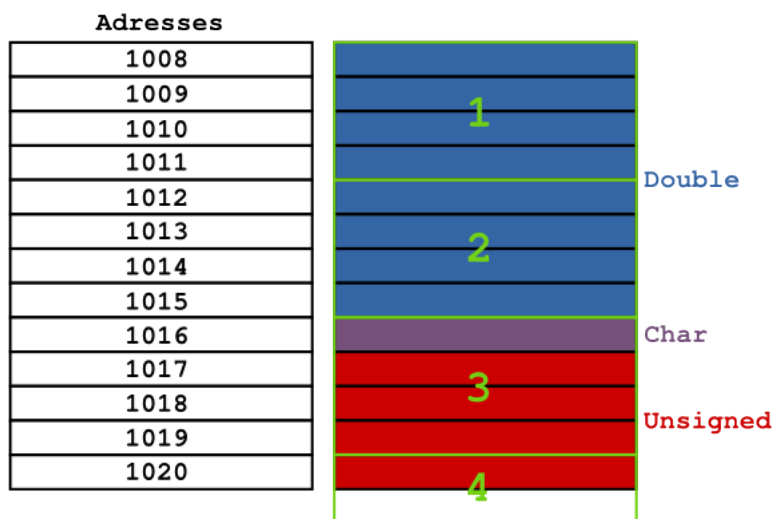


FIGURE 16.2 – Vue de la mémoire par le processeur

- charger le quatrième mot et en extraire le bon octet ;
- et, enfin, combiner le tout pour obtenir la valeur correcte.

C'est plus lent, sans compter que quelques rares processeurs ne gèrent pas ce genre d'opérations et se contentent de signaler une erreur.

❓ Mais, comment le processeur sait-il que le dernier champ est à cheval sur deux mots ?

À cause de son adresse. Dans notre exemple, le membre de type `unsigned int` est situé à l'adresse 1017. Étant donné qu'un `unsigned int` a une taille de quatre octets et que le processeur lit des mots de quatre octets, il sait que si une donnée de ce type n'est pas à une adresse multiple de quatre, alors elle est forcément à califourchon sur deux mots.

Pour éviter ces problèmes, les compilateurs ajoutent des multipliants dit « de bourrage » (*padding* en anglais), afin d'**aligner** les données sur les bonnes adresses. Dans le cas de notre structure, le compilateur a ajouté trois octets de bourrage juste après le membre de type `char` afin que le dernier champ soit forcément à une adresse multiple de quatre.

16.4.3 La macrofonction `offsetof`

Il vous est possible de connaître les **contraintes d'alignement** d'un type (c'est-à-dire le nombre dont doivent être multiple les adresses d'un type) à l'aide de la **macrofonction** `offsetof`² qui est définie dans l'en-tête `<stddef.h>` (nous verrons plus tard ce qu'est une macrofonction lorsque nous aborderons le préprocesseur). Cette dernière attend deux arguments : une définition de structure ou un type de structure et le nom d'un membre de celle-ci. Elle retourne le nombre de multipliants qui précède ce champ au sein de la structure. Son retour est de type `size_t`, comme pour l'opérateur `sizeof`.

Étant donné que le type `char` prends un multipliant, si le premier champ d'une structure est de ce type alors le membre suivant sera forcément précédé du nombre de multipliants de bourrage nécessaire pour qu'il soit bien aligné. Ainsi, le code suivant vous donne les contraintes d'alignement pour chaque type de base.

2. La norme C11 a introduit un nouvel opérateur `_Alignof` (et un synonyme `alignof` fourni par l'en-tête `<stdalign.h>`) qui donne les contraintes d'alignement du type de son opérande. Il s'utilise de la même manière que l'opérateur `sizeof` et retourne lui aussi une valeur entière de type `size_t`.

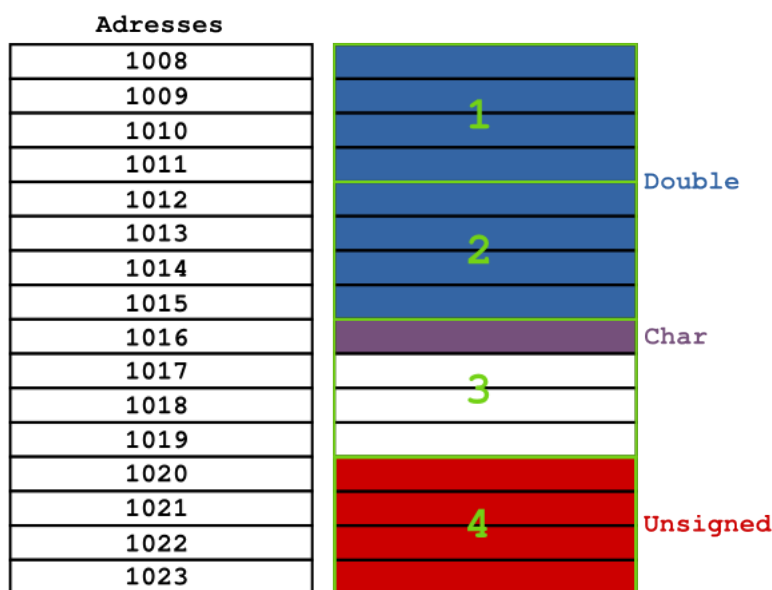


FIGURE 16.3 – La structure correctement alignée

```

1  #include <stddef.h>
2  #include <stdio.h>
3
4
5  int main(void)
6  {
7      printf("short: %u\n", (unsigned)offsetof(struct { char c; short n; }, n));
8      printf("int : %u\n", (unsigned)offsetof(struct { char c; int n; }, n));
9      printf("long : %u\n", (unsigned)offsetof(struct { char c; long n; }, n));
10     printf("float : %u\n", (unsigned)offsetof(struct { char c; float f; }, f));
11     printf("double : %u\n", (unsigned)offsetof(struct { char c; double f; }, f));
12     printf("long double : %u\n", (unsigned)offsetof(struct { char c; long double f; }, f));
13     return 0;
14 }

```

```

1  short: 2
2  int : 4
3  long : 8
4  float : 4
5  double : 8
6  long double : 16

```



Le type `char` ayant une taille de un octet, il peut *toujours* être contenu dans un mot. Il n'a donc pas de contraintes d'alignement.

Pour chaque type, nous définissons une structure composée d'un premier membre de type `char` et d'un second membre d'un type dont nous souhaitons connaître les contraintes d'alignement. Cette définition est utilisée comme premier argument de la macrofonction `offsetof()`, le deuxième étant le nom du second membre de la structure. Pour le reste, le nombre retourné étant de type `size_t`, nous opérons à nouveau une conversion vers le type `unsigned int`.

Les structures en elles-mêmes ne sont pas compliquées à comprendre, mais l'intérêt est parfois plus difficile à saisir. Ne vous en faites pas, nous aurons bientôt l'occasion de découvrir des cas où les structures se trouvent être bien pratiques. En attendant, n'hésitez pas à relire le chapitre s'il vous reste des points obscurs.

Continuons notre route et découvrons à présent un deuxième type de données complexes : les **tableaux**.

Poursuivons notre tour d’horizon des données complexes avec les **tableaux**.

Comme les structures, les tableaux sont des regroupements de plusieurs objets. Cependant, à l’inverse de celles-ci, les tableaux regroupe des données de *même type* et de manière *contiguë* (ce qui exclut la présence de multipléts de bourrage).

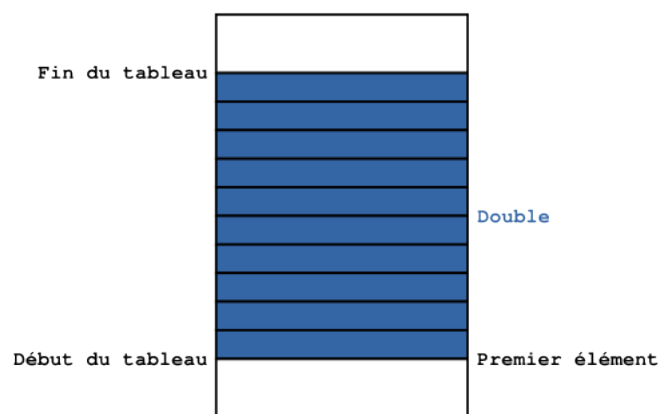


FIGURE 17.1 – Représentation d’un tableau en mémoire

Un tableau est donc un gros bloc de mémoire de *taille finie* qui commence à une adresse déterminée : celle de son premier élément.

17.1 Les tableaux simples (à une dimension)

17.1.1 Définition

La définition d’un tableau nécessite trois informations :

- le type des éléments du tableau (rappelez-vous : un tableau est une suite de données de *même type*) ;
- le nom du tableau (en d’autres mots, son identificateur) ;
- **la longueur** du tableau (le nombre d’éléments qui le composent). Cette dernière doit *obligatoirement* être une constante entière.

1 `type identificateur[longueur];`

Comme vous le voyez, la syntaxe de la déclaration d'un tableau est similaire à celle d'une variable, la seule différence étant qu'il vous est nécessaire de préciser le nombre d'éléments entre crochets à la suite de l'identificateur du tableau.

Ainsi, si nous souhaitons par exemple définir un tableau contenant vingt `int`, nous devons procéder comme suit.

```
1 int tab[20];
```

17.1.2 Initialisation

Comme pour les variables, il est possible d'initialiser un tableau ou, plus précisément, tout ou partie de ses éléments.

Initialisation avec une longueur explicite

L'initialisation se réalise de la même manière que pour les structures, c'est-à-dire à l'aide d'une liste d'initialisation.

```
1 int tab[3] = { 1, 2, 3 };
```

L'exemple ci-dessus initialise les trois membres du tableau avec les valeurs 1, 2 et 3.



Comme pour les structures, dans le cas où vous ne fournissez pas un nombre suffisant de valeurs, les éléments oubliés seront initialisés à zéro ou, s'il s'agit de pointeurs, seront des pointeurs nuls.

Pour un tableau de structures, la liste d'initialisation comportera elle-même une liste d'initialisation pour chaque structure composant le tableau.

```
1 struct temps tab[2] = { { 12, 45, 50.6401 }, { 13, 30, 35.480 } } ;
```



Notez que, inversement, une structure peut comporter des tableaux comme membres.

Initialisation avec une longueur implicite

Lorsque vous initialisez un tableau, il vous est permis d'omettre la longueur de celui-ci, car le compilateur sera capable d'en déterminer la taille en comptant le nombre d'éléments présents dans la liste d'initialisation. Ainsi, l'exemple ci-dessous est correct et définit un tableau de trois `int` valant respectivement 1, 2 et 3.

```
1 int tab[] = { 1, 2, 3 };
```

17.1.3 Accès aux éléments d'un tableau

L'accès aux éléments d'un tableau se réalise à l'aide d'un **indice**, un nombre entier correspondant à la position de chaque élément dans le tableau (premier, deuxième, troisième, etc). Cependant, il y a une petite subtilité : *les indices commencent toujours à zéro*.

Ceci tient au fait que l'accès aux différents éléments est réalisé à l'aide de l'adresse du premier élément à laquelle est ajouté l'indice (qui doit donc être nul pour conserver l'adresse du premier élément). Étant donné que tous les éléments ont la même taille et se suivent en mémoire,

Indice	Adresse de l'élément
0	1008 (1008 + 0)
1	1012 (1008 + 4)
2	1016 (1008 + 8)
3	1020 (1008 + 12)
4	1024 (1008 + 16)
5	1028 (1008 + 20)
...	...

Indice	Adresse de l'élément
0	1008 (1008 + (0 * 4))
1	1012 (1008 + (1 * 4))
2	1016 (1008 + (2 * 4))
3	1020 (1008 + (3 * 4))
4	1024 (1008 + (4 * 4))
5	1028 (1008 + (5 * 4))
...	...

leurs adresses peuvent effectivement se calculer à l'aide de l'adresse du premier élément et d'un décalage par rapport à celle-ci (l'indice, donc).

Prenons un exemple avec un tableau composé de `int` (ayant une taille de quatre octets) et dont le premier élément est placé à l'adresse 1008. Si vous déterminez à la main les adresses de chaque élément, vous obtiendrez ceci.

En fait, il est possible de reformuler ceci à l'aide d'une multiplication entre l'indice et la taille d'un `int`.

Nous pouvons désormais formaliser mathématiquement tout ceci en posant T la taille d'un élément du tableau, i l'indice de cet élément, et A l'adresse de début du tableau (l'adresse du premier élément, donc). L'adresse de l'élément d'indice i s'obtient en calculant $A + T \times i$. Ceci étant posé, voyons à présent comment mettre tout cela en œuvre en C.

Le premier élément

Pour commencer, nous avons besoin de l'adresse du premier élément du tableau. Celle-ci s'obtient en fait d'une manière plutôt contre-intuitive : lorsque vous utilisez une variable de type tableau dans une expression, celle-ci est convertie implicitement en un pointeur *constant* sur son premier élément. Comme vous pouvez le constater dans l'exemple qui suit, nous pouvons utiliser la variable `tab` comme nous l'aurions fait s'il s'agissait d'un pointeur.

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      int tab[3] = { 1, 2, 3 };
6
7      printf("Premier élément : %d\n", *tab);
8      return 0;
9  }
```

```

1  Premier élément : 1
```

Notez que comme il s'agit d'une conversion implicite vers un pointeur constant, il n'est pas possible d'affecter une valeur à une variable de type tableau. Ainsi, le code suivant est incorrect.

```

1  int t1[3];
2  int t2[3];
3
4  t1 = t2; /* Incorrect. */

```

La règle de conversion implicite comprends néanmoins deux exceptions : l'opérateur `&` et l'opérateur `sizeof`.

Lorsqu'il est appliqué à une variable de type tableau, l'opérateur `&` produit comme résultat l'adresse du premier élément du tableau. Si vous exécutez le code ci-dessous, vous constaterez que les deux expressions donnent un résultat identique.

```

1  #include <stdio.h>
2
3
4  int main(void)
5  {
6      int tab[3];
7
8      printf("%p == %p\n", (void *)tab, (void *)&tab);
9      return 0;
10 }

```

Dans le cas où une expression de type tableau est fournie comme opérande de l'opérateur `sizeof`, le résultat de celui-ci sera bien la taille totale du tableau (en multipliants) et non la taille d'un pointeur.

```

1  #include <stdio.h>
2
3
4  int main(void)
5  {
6      int tab[3];
7      int *ptr;
8
9      printf("sizeof tab = %u\n", (unsigned)sizeof tab);
10     printf("sizeof ptr = %u\n", (unsigned)sizeof ptr);
11     return 0;
12 }

```

```

1  sizeof tab = 12
2  sizeof ptr = 8

```

Cette propriété vous permet d'obtenir le nombre d'éléments d'un tableau à l'aide de l'expression suivante.

```

1  sizeof tab / sizeof tab[0]

```

Les autres éléments

Pour accéder aux autres éléments, il va nous falloir ajouter la position de l'élément voulu à l'adresse du premier élément et ensuite utiliser l'adresse obtenue. Toutefois, recourir à la formule présentée au-dessus ne marchera pas car, en C, les pointeurs sont typés. Dès lors, lorsque vous additionnez un nombre à un pointeur, le compilateur multiplie automatiquement ce nombre par la taille du type d'objet référencé par le pointeur. Ainsi, pour un tableau de `int`, l'expression `tab + 1` est implicitement convertie en `tab + sizeof(int)`.

Voici un exemple affichant la valeur de tous les éléments d'un tableau.

```

1  #include <stdio.h>
2
3
4  int main(void)
5  {
6      int tab[3] = { 1, 2, 3 };
7
8      printf("Premier élément : %d\n", *tab);
9      printf("Deuxième élément : %d\n", *(tab + 1));
10     printf("Troisième élément : %d\n", *(tab + 2));
11     return 0;
12 }

```

```

1  Premier élément : 1
2  Deuxième élément : 2
3  Troisième élément : 3

```

L'expression `*(tab + i)` étant quelque peu lourde, il existe un opérateur plus concis pour réaliser cette opération : l'opérateur `[]`. Celui-ci s'utilise de cette manière.

```

1  expression[indice]

```

Ce qui est équivalent à l'expression suivante.

```

1  *(expression + indice)

```

L'exemple suivant est donc identique au précédent.

```

1  #include <stdio.h>
2
3
4  int main(void)
5  {
6      int tab[3] = { 1, 2, 3 };
7
8      printf("Premier élément : %d\n", tab[0]);
9      printf("Deuxième élément : %d\n", tab[1]);
10     printf("Troisième élément : %d\n", tab[2]);
11     return 0;
12 }

```

17.1.4 Parcours et débordement

Une des erreurs les plus fréquente en C consiste à dépasser la taille d'un tableau, ce qui est appelé un cas de **débordement** (*overflow* en anglais). En effet, si vous tentez d'accéder à un objet qui ne fait pas partie de votre tableau, vous réalisez un accès mémoire non autorisé, ce qui provoquera un comportement indéfini. Cela arrive généralement lors d'un parcours de tableau à l'aide d'une boucle.

```

1  #include <stdio.h>
2
3
4  int main(void)
5  {
6      int tableau[5] = {784, 5, 45, -12001, 8};
7      int somme = 0;
8      unsigned i;
9
10     for (i = 0; i <= 5; ++i)
11         somme += tableau[i];
12
13     printf("%d\n", somme);
14     return 0;
15 }

```

Le code ci-dessus est volontairement erroné et tente d'accéder à un élément qui se situe au-delà du tableau. Ceci provient de l'utilisation de l'opérateur `<=` à la place de l'opérateur `<` ce qui entraîne un tour de boucle avec `i` qui est égal à 5, alors que le dernier indice du tableau doit être quatre.



N'oubliez pas : les indices d'un tableau commencent *toujours* à zéro. En conséquence, les indices valides d'un tableau de n éléments vont de 0 à $n - 1$.

17.1.5 Tableaux et fonctions

Passage en argument

Étant donné qu'un tableau peut être utilisé comme un pointeur sur son premier élément, lorsque vous passer un tableau en argument d'une fonction, celle-ci reçoit un pointeur vers le premier élément du tableau. Le plus souvent, il vous sera nécessaire de passer également la taille du tableau afin de pouvoir le parcourir.

Le code suivante utilise une fonction pour parcourir un tableau d'entier et afficher la valeur de chacun de ses éléments.

```

1  #include <stdio.h>
2
3
4  void affiche_tableau(int *tab, unsigned taille)
5  {
6      unsigned i;
7
8      for (i = 0; i < taille; ++i)
9          printf("tab[%u] = %d\n", i, tab[i]);
10 }
11
12
13 int main(void)
14 {
15     int tab[5] = { 2, 45, 67, 89, 123 };
16
17     affiche_tableau(tab, 5);
18     return 0;
19 }
```

```

1  tab[0] = 2
2  tab[1] = 45
3  tab[2] = 67
4  tab[3] = 89
5  tab[4] = 123
```

Notez qu'il existe une syntaxe alternative pour déclarer un paramètre de type tableau héritée du langage B (voyez la dernière section)



```
void affiche_tableau(int tab[], unsigned taille)
```

Toutefois, nous vous conseillons de recourir à la première écriture, cette dernière étant plus explicite.

Retour de fonction

De la même manière que pour le passage en argument, retourner un tableau revient à retourner un pointeur sur le premier élément de celui-ci. Toutefois, n'oubliez pas les problématiques de

classe de stockage ! Si vous retournez un tableau de classe de stockage automatique, vous fournissez à la fonction appelante un pointeur vers un objet qui n'existe plus (puisque l'exécution de la fonction appelée est terminée).

```

1  #include <stdio.h>
2
3
4  int *tableau(void)
5  {
6      int tab[5] = { 1, 2, 3, 4, 5 };
7
8      return tab;
9  }
10
11
12 int main(void)
13 {
14     int *p = tableau(); /* Incorrect. */
15
16     printf("%d\n", p[0]);
17     return 0;
18 }
```

17.2 La vérité sur les tableaux

Nous vous avons dit qu'une variable de type tableau pouvait être utilisée comme un pointeur constant sur son premier élément. Cependant, ce n'est pas tout à fait vrai.

17.2.1 Un peu d'histoire

Le prédécesseur du langage C était le langage B. Lorsque le développement du C a commencé, un des objectifs était de le rendre autant que possible compatible avec le B, afin de ne pas devoir (trop) modifier les codes existants (un code écrit en B pourrait ainsi être compilé avec un compilateur C sans ou avec peu de modifications). Or, en B, un tableau se définissait comme suit.

```
1 auto tab[3];
```



Le langage B était un langage non typé, ce qui explique l'absence de type dans la définition. Le mot-clé `auto` (toujours présent en langage C, mais devenu obsolète) servait à indiquer que la variable définie était de classe de stockage automatique.

Toutefois, à la différence du langage C, cette définition crée un tableau de trois éléments *et* un pointeur initialisé avec l'adresse du premier élément. Ainsi, pour créer un pointeur, il suffisait de définir une variable comme un tableau de taille nulle.

```
1 auto ptr[];
```

Le langage C, toujours en gestation, avait repris ce mode de fonctionnement. Cependant, les structures sont arrivées et les problèmes avec. En effet, prenez ce bout de code.

```

1  #include <stdio.h>
2
3  struct exemple {
4      int tab[3];
5  };
6
7
```



```

8  struct exemple exemple_init(void)
9  {
10     struct exemple init = { { 1, 2, 3 } };
11
12     return init;
13 }
14
15
16 int main(void)
17 {
18     struct exemple s = exemple_init();
19
20     printf("%d\n", s.tab[0]);
21     return 0;
22 }

```

La fonction `exemple_init()` retourne une structure qui est utilisée pour initialiser la variable de la fonction `main()`. Dans un tel cas, comme pour n'importe quelle variable, le contenu de la première structure sera intégralement copié dans la deuxième. Le souci, c'est que si une définition de tableau crée un tableau *et* un pointeur initialisé avec l'adresse du premier élément de celui-ci, alors il est nécessaire de modifier le champ `tab` de la structure `s` lors de la copie sans quoi son champ `tab` pointerait vers le tableau de la structure `init` (qui n'existera plus puisque de classe de stockage automatique) et non vers le sien. Voilà qui complexifie la copie de structures, particulièrement si sa définition comprend plusieurs tableaux possiblement imbriqués...

Pour contourner ce problème, les concepteurs du langage C ont imaginé une solution (tordue) qui est à l'origine d'une certaine confusion dans l'utilisation des tableaux : une variable de type tableau *ne sera plus un pointeur*, mais sera *convertie en un pointeur sur son premier élément lors de son utilisation*.

17.2.2 Conséquences de l'absence d'un pointeur

Étant donné qu'il n'y a plus de pointeur alloué, la copie de structures s'en trouve simplifiée et peut être réalisée sans opération particulière (ce qui était l'objectif recherché).

Toutefois, cela entraîne une autre conséquence : il n'est plus possible d'assigner une valeur à une variable de type tableau, seuls ses éléments peuvent se voir affecter une valeur. Ainsi, le code suivant est incorrect puisqu'il n'y a aucun pointeur pour recevoir l'adresse du premier élément du tableau `t2`.

```

1  int t1[3];
2  int t2[3];
3
4  t1 = t2; /* Incorrect. */

```

Également, puisqu'une variable de type tableau n'est plus un pointeur, celle-ci n'a pas d'adresse. Dès lors, lorsque l'opérateur `&` est appliqué à une variable de type tableau, le résultat sera l'adresse du premier élément du tableau puisque seuls les éléments du tableau ont une adresse.

17.3 Les tableaux multidimensionnels

Jusqu'à présent, nous avons travaillé avec des tableaux linéaires, c'est-à-dire dont les éléments se suivaient les uns à la suite des autres. Il s'agit de tableaux dit à une dimension ou **unidimensionnels**.

Cependant, certaines données peuvent être représentées plus simplement sous la forme de tableaux à deux dimensions (autrement dit, organisées en lignes et en colonnes). C'est par exemple le cas des images (non vectorielles) qui sont des matrices de pixels ou, plus simplement, d'une grille de Sudoku qui est organisée en neuf lignes et en neuf colonnes.

Le langage C vous permet de créer et de gérer ce type de tableaux dit **multidimensionnels** (en fait, des tableaux de tableaux) et ce, bien au-delà de deux dimensions.

17.3.1 Définition

La définition d'un tableau multidimensionnel se réalise de la même manière que celle d'un tableau unidimensionnel si ce n'est que vous devez fournir la taille des différentes dimensions.

Par exemple, si nous souhaitons définir un tableau de `int` de vingt lignes et trente-cinq colonnes, nous procéderons comme suit.

```
1 int tab[20][35];
```

De même, pour un tableau de `double` à trois dimensions.

```
1 double tab[3][4][5];
```

17.4 Initialisation

17.4.1 Initialisation avec une longueur explicite

Comme pour les tableaux de structures, l'initialisation d'un tableau multidimensionnel s'effectue à l'aide d'une liste d'initialisation comprenant elle-même des listes d'initialisations.

```
1 int t1[2][2] = { { 1, 2 }, { 3, 4 } };
2 int t2[2][2][2] = { { { 1, 2 }, { 3, 4 } }, { { 5, 6 }, { 7, 8 } } };
```



Comme pour les tableaux unidimensionnel, dans le cas où vous ne fournissez pas un nombre suffisant de valeurs, les éléments omis seront initialisés à zéro ou, s'il s'agit de pointeurs, seront des pointeurs nuls.

17.4.2 Initialisation avec une longueur implicite

Lorsque vous initialisez un tableau multidimensionnel, il vous est permis d'omettre la taille de la *première dimension*. La taille des autres dimensions doit en revanche être spécifiée, le compilateur ne pouvant déduire la taille de toutes les dimensions.

```
1 int t1[][2] = { { 1, 2 }, { 3, 4 } };
2 int t2[][2][2] = { { { 1, 2 }, { 3, 4 } }, { { 5, 6 }, { 7, 8 } } };
```

17.4.3 Utilisation

Techniquement, un tableau multidimensionnel est un tableau dont les éléments sont eux-mêmes des tableaux. Dès lors, vous avez besoin d'autant d'indices qu'il y a de dimensions. Par exemple, pour un tableau à deux dimensions, vous avez besoin d'un premier indice pour accéder à l'élément souhaité du premier tableau, mais comme cet élément est lui-même un tableau, vous devez utiliser un second indice pour sélectionner un élément de celui-ci. Illustration.

```
1 #include <stdio.h>
2
3
4 int main(void)
5 {
6     int tab[2][2] = { { 1, 2 }, { 3, 4 } };
```

```

7
8     printf("tab[0][0] = %d\n", tab[0][0]);
9     printf("tab[0][1] = %d\n", tab[0][1]);
10    printf("tab[1][0] = %d\n", tab[1][0]);
11    printf("tab[1][1] = %d\n", tab[1][1]);
12    return 0;
13 }

```

```

1 tab[0][0] = 1
2 tab[0][1] = 2
3 tab[1][0] = 3
4 tab[1][1] = 4

```

17.4.4 Représentation en mémoire

Techniquement, les données d'un tableau multidimensionnel sont stockées les unes à côté des autres en mémoire : elles sont rassemblées dans un tableau à une seule dimension. Si les langages comme le FORTRAN mémorisent les colonnes les unes après les autres (*column-major order* en anglais), le C mémorise les tableaux lignes par lignes (*row-major order*).

1	2	3
4	5	6
7	8	9

FIGURE 17.2 – Exemple de tableau en deux dimensions

1	4	7	2	5	8	3	6	9
---	---	---	---	---	---	---	---	---

FIGURE 17.3 – Column-major order

1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---

FIGURE 17.4 – Row-major order

Le calcul d'adresse à effectuer est une généralisation du calcul vu au chapitre précédent.

17.4.5 Parcours

Le même exemple peut être réalisé à l'aide de deux boucles imbriquées afin de parcourir le tableau.

```

1 #include <stdio.h>
2
3
4 int main(void)
5 {
6     int tab[2][2] = { { 1, 2 }, { 3, 4 } };

```

```

7   unsigned i;
8   unsigned j;
9
10  for (i = 0; i < 2; ++i)
11      for (j = 0; j < 2; ++j)
12          printf("tab[%u][%u] = %d\n", i, j, tab[i][j]);
13
14  return 0;
15 }

```

17.4.6 Tableaux multidimensionnels et fonctions

Passage en argument

Souvenez-vous : sauf exceptions, un tableau est converti en un pointeur sur son premier élément. Dès lors, qu’obtenons-nous lors du passage d’un tableau à deux dimensions en argument d’une fonction ? Le premier élément du tableau est un tableau, donc un pointeur sur... Un tableau (hé oui). :p

La syntaxe d’un pointeur sur tableau est la suivante.

```

1   type (*identificateur)[taille];

```

Vous remarquerez la présence de parenthèses autour du symbole `*` et de l’identificateur afin de signaler au compilateur qu’il s’agit d’un pointeur sur un tableau et non d’un tableau de pointeurs. Également, notez que la taille du tableau référencé doit être spécifiée. En effet, sans celle-ci, le compilateur ne pourrait pas opérer correctement le calcul d’adresses puisqu’il ne connaîtrait pas la taille des éléments composant le tableau référencé par le pointeur.

La même logique peut-être appliquée pour créer des pointeur sur des tableaux de tableaux.

```

1   type (*identificateur)[N][M];

```

Et ainsi de suite jusqu’à ce que mort s’en suive... :-°

L’exemple ci-dessous illustre ce qui vient d’être dit en utilisant une fonction pour afficher le contenu d’un tableau à deux dimensions de `int`.

```

1   #include <stdio.h>
2
3
4   void affiche_tableau(int (*tab)[2], unsigned n, unsigned m)
5   {
6       unsigned i;
7       unsigned j;
8
9       for (i = 0; i < n; ++i)
10          for (j = 0; j < m; ++j)
11              printf("tab[%u][%u] = %d\n", i, j, tab[i][j]);
12  }
13
14
15  int main(void)
16  {
17      int tab[2][2] = { { 1, 2 }, { 3, 4 } };
18
19      affiche_tableau(tab, 2, 2);
20      return 0;
21  }

```

Retour de fonction

Même remarque que pour les tableaux unidimensionnels : attention à la classe de stockage ! Pour le reste, nous vous laissons admirer la syntaxe particulièrement dégoûtante d’une fonction retournant un pointeur sur un tableau de deux `int`.

```
1  #include <stdio.h>
2
3
4  int (*tableau(void))[2] /* Ouh ! Que c'est laid ! */
5  {
6      int tab[2][2] = { { 1, 2 }, { 3, 4 } };
7
8      return tab;
9  }
10
11
12 int main(void)
13 {
14     int (*p)[2] = tableau(); /* Incorrect. */
15
16     printf("%d\n", p[0][0]);
17     return 0;
18 }
```

17.5 Exercices

17.5.1 Somme des éléments

Réalisez une fonction qui calcule la somme de tous les éléments d'un tableau de `int`.

```
1  int somme(int *tab, unsigned taille)
2  {
3      unsigned i;
4      int res = 0 ;
5
6      for (i = 0; i < taille; ++i)
7          res += tab[i];
8
9      return res;
10 }
```

17.5.2 Maximum et minimum

Créez deux fonctions : une qui retourne le plus petit élément d'un tableau de `int` et une qui renvoie le plus grand élément d'un tableau de `int`.

```
1  int minimum(int *tab, unsigned taille)
2  {
3      unsigned i;
4      int min = tab[0];
5
6      for (i = 1; i < taille; ++i)
7          if (tab[i] < min)
8              min = tab[i];
9
10     return min;
11 }
12
13
14 int maximum(int *tab, unsigned taille)
15 {
16     unsigned i;
17     int max = tab[0];
18
19     for (i = 1; i < taille; ++i)
20         if (tab[i] > max)
21             max = tab[i];
22
23     return max;
24 }
```

17.5.3 Recherche d'un élément

Construisez une fonction qui teste la présence d'une valeur dans un tableau de `int`. Celle-ci retournera 1 si un ou plusieurs éléments du tableau sont égaux à la valeur recherchée, 0 sinon.

```
1  int find(int * tab, unsigned taille, int val)
2  {
3      unsigned i;
4
5      for (i = 0; i < taille; ++i)
6          if (tab[i] == val)
7              return 1;
8
9      return 0;
10 }
```

17.5.4 Inverser un tableau

Produisez une fonction qui inverse le contenu d'un tableau (le premier élément devient le dernier, l'avant dernier le deuxième et ainsi de suite).

Indice

Pensez à la fonction `swap()` présentée dans le chapitre sur les pointeurs.

Correction

```
1  void swap(int *pa, int *pb)
2  {
3      int tmp;
4
5      tmp = *pa;
6      *pa = *pb;
7      *pb = tmp;
8  }
9
10
11 void invert(int *tab , unsigned taille)
12 {
13     unsigned i;
14
15     for (i = 0; i < (taille / 2); ++i)
16         swap(tab + i , tab + taille - 1 - i);
17 }
18 }
```

17.5.5 Produit des lignes

Composez une fonction qui calcul le produit de la somme des éléments de chaque ligne d'un tableau de `int` à deux dimensions (ce tableau comprend cinq lignes et cinq colonnes).

```
1  int produit(int (*tab)[5])
2  {
3      unsigned i;
4      unsigned j;
5      int res = 1;
6
7      for (i = 0; i < 5; ++i)
8      {
9          int tmp = 0;
10
11         for (j = 0; j < 5; ++j)
12             tmp += tab[i][j];
```

```

13     res *= tmp;
14 }
15
16     return res;
17 }
18

```

17.5.6 Triangle de Pascal

Les triangles de Pascal sont des objets mathématiques amusants. Voici une petite animation qui vous expliquera le fonctionnement de ceux-ci.

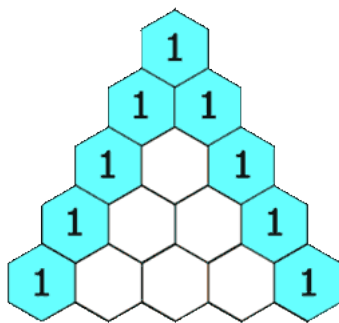


FIGURE 17.5 – Explication des triangles de Pascal en image

Votre objectif va être de réaliser un programme qui affiche un triangle de Pascal de la taille souhaitée par l'utilisateur. Pour ce faire, nous allons diviser le triangle en lignes afin de pouvoir le représenter sous la forme d'un tableau à deux dimensions. Chaque élément du tableau se verra attribué soit une valeur du triangle soit zéro pour signaler qu'il n'est pas utilisé.

La première chose que nous allons faire est donc définir un tableau à deux dimensions (nous fixerons la taille des dimensions à dix) dont tous les éléments sont initialisés à zéro. Ensuite, nous allons demander à l'utilisateur d'entrer la taille du triangle qu'il souhaite obtenir (celle-ci ne devra pas être supérieure aux dimensions du tableau).

À présent, passons à la fonction de création du triangle de Pascal. Celle-ci devra mettre en œuvre l'algorithme suivant.

```

1  N = taille du triangle de Pascal fournie par l'utilisateur
2
3  Mettre la première case du tableau à 1
4
5  Pour i = 1, i < N, i = i + 1
6      Mettre la première case de la ligne à 1
7
8      Pour j = 1, j < i, j = j + 1
9          La case [i,j] prend la valeur [i - 1, j - 1] + [i - 1, j]
10
11     Mettre la dernière case de la ligne à 1

```

Enfin, il vous faudra écrire une petite fonction pour afficher le tableau ainsi créé. Bon courage !

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4
5  void triangle_pascal(int (*tab)[10], unsigned taille)
6  {
7      unsigned i;
8      unsigned j;
9

```

```

10     tab[0][0] = 1;
11
12     for (i = 1; i < taille; ++i)
13     {
14         tab[i][0] = 1;
15
16         for (j = 1; j < i; ++j)
17             tab[i][j] = tab[i - 1][j - 1] + tab[i - 1][j];
18
19         tab[i][i] = 1;
20     }
21 }
22
23
24 void print_triangle(int (*tab)[10], unsigned taille)
25 {
26     unsigned i;
27     unsigned j;
28     int sp;
29
30     for (i = 0; i < taille; ++i)
31     {
32         for (sp = taille - 1 - i; sp > 0; --sp)
33             printf(" ");
34
35         for (j = 0; j < taille; ++j)
36             if (tab[i][j] != 0)
37                 printf("%d ", tab[i][j]);
38
39         printf("\n");
40     }
41 }
42
43
44 int main(void)
45 {
46     int tab[10][10] = { { 0 } };
47     unsigned taille;
48
49     printf("Taille du triangle: ");
50
51     if (scanf("%u", &taille) != 1)
52     {
53         printf("Mauvaise saisie\n");
54         return EXIT_FAILURE;
55     }
56     else if (taille > 10)
57     {
58         printf("La taille ne doit pas être supérieure à 10\n");
59         return EXIT_FAILURE;
60     }
61
62     triangle_pascal(tab, taille);
63     print_triangle(tab, taille);
64     return 0;
65 }

```

Dans le chapitre suivant, nous aborderons un nouveau type d'agrégat, un peu particulier puisqu'il se base sur les tableaux : les **chaînes de caractères**.

Dans ce chapitre, nous allons apprendre à manipuler du texte ou, en langage C, des **chaînes de caractères**

18.1 Qu'est-ce qu'une chaîne de caractères ?

Dans le chapitre sur les variables, nous avons mentionné le type `char`. Pour rappel, nous vous avons dit que le type `char` servait surtout au stockage de caractères, mais que comme ces derniers étaient stockés dans l'ordinateur sous forme de nombres, il était également possible d'utiliser ce type pour mémoriser des nombres.

Le seul problème, c'est qu'une variable de type `char` ne peut stocker qu'une seule lettre, ce qui est insuffisant pour stocker une phrase ou même un mot. Si nous voulons mémoriser un texte, nous avons besoin d'un outil pour rassembler plusieurs lettres dans un seul objet, manipulable dans notre langage. Cela tombe bien, nous en avons justement découvert un au chapitre précédent : les tableaux.

C'est ainsi que le texte est géré en C : sous forme de tableaux de `char` appelés **chaînes de caractères** (*strings* en anglais).

18.1.1 Représentation en mémoire

Néanmoins, il y a une petite subtilité. Une chaîne de caractères est un plus qu'un tableau : c'est un objet à part entière qui doit être manipulable directement. Or, ceci n'est possible que si nous connaissons sa taille.

Avec une taille intégrée

Dans certains langages de programmation, les chaînes de caractères sont stockées sous la forme d'un tableau de `char` auquel est adjoint un entier pour indiquer sa longueur. Plus précisément, lors de l'allocation du tableau, le compilateur réserve un élément supplémentaire pour conserver la taille de la chaîne. Ainsi, il est aisé de parcourir la chaîne et de savoir quand la fin de celle-ci est atteinte. De telles chaînes de caractères sont souvent appelées des *Pascal strings*, s'agissant d'une convention apparue avec le langage de programmation Pascal.

Avec une sentinelle

Toutefois, une telle technique limite la taille des chaînes de caractères à la capacité du type entier utilisé pour mémoriser la longueur de la chaîne. Dans la majorité des cas, il s'agit d'un `unsigned char`, ce qui donne une limite de 255 caractères maximum sur la plupart des machines. Pour ne pas subir cette limitation, les concepteurs du langage C ont adopté une autre solution :

la fin de la chaîne de caractères sera indiquée par un caractère spécial, en l'occurrence zéro (noté `'\0'`). Les chaînes de caractères qui fonctionnent sur ce principe sont appelées *null terminated strings*, ou encore *C strings*.

Cette solution a toutefois deux inconvénients :

- la taille de chaque chaîne doit être calculée en la parcourant jusqu'au caractère nul ;
- le programmeur doit s'assurer que chaque chaîne qu'il construit se termine bien par un caractère nul.

18.2 Définition, initialisation et utilisation

18.2.1 Définition

Définir une chaîne de caractères, c'est avant tout définir un tableau de `char`, ce que nous avons vu au chapitre précédent. L'exemple ci-dessous définit un tableau de vingt-cinq `char`.

```
1 char tab[25];
```

18.2.2 Initialisation

Il existe deux méthodes pour initialiser une chaîne de caractères :

- de la même manière qu'un tableau ;
- à l'aide d'une chaîne de caractères littérale.

Avec une liste d'initialisation

Initialisation avec une longueur explicite

Comme pour n'importe quel tableau, l'initialisation se réalise à l'aide d'une liste d'initialisation. L'exemple ci-dessous définit donc un tableau de vingt-cinq `char` et initialise les sept premiers avec la suite de lettres « Bonjour ».

```
1 char chaine[25] = { 'B', 'o', 'n', 'j', 'o', 'u', 'r' };
```

Étant donné que seule une partie des éléments sont initialisés, les autres sont implicitement mis à zéro, ce qui nous donne une chaîne de caractères valides puisqu'elle est bien terminée par un caractère nul. Faites cependant attention à ce qu'il y ait *toujours* de la place pour un caractère nul.

Initialisation avec une longueur implicite

Dans le cas où vous ne spécifiez pas de taille lors de la définition, il vous faudra ajouter le caractère nul à la fin de la liste d'initialisation pour obtenir une chaîne valide.

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

Avec une chaîne littérale

Bien que tout à fait valide, cette première solution est toutefois assez fastidieuse. Aussi, il en existe une seconde : recourir à une **chaîne de caractères littérales** pour initialiser un tableau. Une chaîne de caractères littérales est une suite de caractères entourée par le symbole `"`. Nous en avons déjà utilisé auparavant comme argument des fonctions `printf()` et `scanf()`.

Techniquement, une chaîne littérale est un tableau de `char` terminé par un caractère nul. Elles peuvent donc s'utiliser comme n'importe quel autre tableau. Si vous exécutez le code ci-dessous,

vous remarquerez que l'opérateur `sizeof` retourne bien le nombre de caractères composant la chaîne littérale (n'oubliez pas de compter le caractère nul) et que l'opérateur `printf` peut effectivement leur être appliqué.

```
1 #include <stdio.h>
2
3
4 int main(void)
5 {
6     printf("%u\n", (unsigned)sizeof "Bonjour");
7     printf("%c\n", "Bonjour"[3]);
8     return 0;
9 }
```

```
1 8
2 j
```

Ces chaînes de caractères littérales peuvent également être utilisées à la place des listes d'initialisation. En fait, il s'agit de la troisième et dernière exception à la règle de conversion implicite des tableaux.

Initialisation avec une longueur explicite

Dans le cas où vous spécifiez la taille de votre tableau, faites bien attention à ce que celui-ci dispose de suffisamment de place pour accueillir la chaîne entière, c'est-à-dire les caractères qui la composent *et* le caractère nul.

```
1 char chaine[25] = "Bonjour";
```

Initialisation avec une longueur implicite

L'utilisation d'une chaîne littérale pour initialiser un tableau dont la taille n'est pas spécifiée vous évite de vous soucier du caractère nul puisque celui-ci fait partie de la chaîne littérale.

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

Utilisation de pointeurs

Nous vous avons dit que les chaînes littérales n'étaient rien d'autre que des tableaux de `char` terminés par un caractère nul. Dès lors, comme pour n'importe quel tableau, il vous est loisible de les référencer à l'aide de pointeurs.

```
1 char *ptr = "Bonjour";
```

Néanmoins, les chaînes littérales sont des *constantes*, il vous est donc impossible de les modifier. L'exemple ci-dessous est donc incorrect.

```
1 int main(void)
2 {
3     char *ptr = "bonjour";
4
5     ptr[0] = 'B'; /* Incorrect. */
6     return 0;
7 }
```



Notez bien la différence entre les exemples précédents qui initialisent un tableau avec le contenu d'une chaîne littérale (il y a donc copie de la chaîne littérale) et cet exemple qui initialise un pointeur avec l'adresse du premier élément d'une chaîne littérale.

18.2.3 Utilisation

Pour le reste, une chaîne de caractères s'utilise comme n'importe quel autre tableau. Aussi, pour modifier son contenu, il vous faudra accéder à ses éléments un à un.

```

1  #include <stdio.h>
2
3
4  int main(void)
5  {
6      char chaine[25] = "Bonjour";
7
8      printf("%s\n", chaine);
9      chaine[0] = 'A';
10     chaine[1] = 'u';
11     chaine[2] = ' ';
12     chaine[3] = 'r';
13     chaine[4] = 'e';
14     chaine[5] = 'v';
15     chaine[6] = 'o';
16     chaine[7] = 'i';
17     chaine[8] = 'r';
18     chaine[9] = '\0'; /* N'oubliez pas le caractère nul ! */
19     printf("%s\n", chaine);
20     return 0;
21 }
```

```

1  Bonjour
2  Au revoir
```

18.3 Afficher et récupérer une chaîne de caractères

Les chaînes littérales n'étant rien d'autre que des tableaux de `char`, il vous est possible d'utiliser des chaînes de caractères là où vous employiez des chaînes littérales. Ainsi, les deux exemples ci-dessous afficheront la même chose.

```

1  #include <stdio.h>
2
3
4  int main(void)
5  {
6      char chaine[] = "Bonjour\n";
7
8      printf(chaine);
9      return 0;
10 }
```

```

1  #include <stdio.h>
2
3
4  int main(void)
5  {
6      printf("Bonjour\n");
7      return 0;
8  }
```

```

1  Bonjour
```

Toutefois, les fonctions `printf()` et `scanf()` disposent d'un indicateur de conversion vous permettant d'afficher ou de demander une chaîne de caractères : `%s`.

18.3.1 Printf

L'exemple suivant illustre l'utilisation de cet indicateur de conversion avec `printf()` et affiche la même chose que les deux codes précédents.

```

1  #include <stdio.h>
2
3
4  int main(void)
5  {
6      char chaine[] = "Bonjour";
7
8      printf("%s\n", chaine);
9      return 0;
10 }
```

```

1  Bonjour
```

18.3.2 Scanf

Le même indicateur de conversion peut être utiliser avec `scanf()` pour demander à l'utilisateur d'entrer une chaîne de caractères. Cependant, un problème se pose : étant donné que nous devons créer un tableau de taille finie pour accueillir la saisie de l'utilisateur, nous devons impérativement limiter la longueur des données que nous fournit l'utilisateur.

Pour éviter ce problème, il est possible de spécifier une taille maximale à la fonction `scanf()`. Pour ce faire, il vous suffit de placer un nombre entre le symbole `%` et l'indicateur de conversion `s`. L'exemple ci-dessous demande à l'utilisateur d'entrer son prénom (limité à 254 caractères) et affiche ensuite celui-ci.



La fonction `scanf()` ne décompte pas le caractère nul final de la limite fournie ! Il vous est donc nécessaire de lui indiquer la taille de votre tableau diminuée de un.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4
5  int main(void)
6  {
7      char chaine[255];
8
9      printf("Quel est votre prénom ? ");
10
11     if (scanf("%254s", chaine) != 1)
12     {
13         printf("Erreur lors de la saisie\n");
14         return EXIT_FAILURE;
15     }
16
17     printf("Bien le bonjour %s !\n", chaine);
18     return 0;
19 }
```

```

1  Quel est votre prénom ? Albert
2  Bien le bonjour Albert !
```

Chaîne de caractères avec des espaces

Sauf qu'en fait, l'indicateur `s` signifie : « la plus longue suite de caractère ne comprenant pas d'espaces » (les espaces étant ici entendu comme une suite d'un ou plusieurs des caractères suivant : `' '`, `'\f'`, `'\n'`, `'\r'`, `'\t'`, `'\v'`)...

Autrement dit, si vous entrez « Bonjour tout le monde », la fonction `scanf()` va s'arrêter au mot « Bonjour », ce dernier étant suivi par un espace.



Comment peut-on récupérer une chaîne complète alors ?

Eh bien, il va falloir nous débrouiller avec l'indicateur `c` de la fonction `scanf()` et réaliser nous même une fonction employant ce dernier au sein d'une boucle. Ainsi, nous pouvons par exemple créer une fonction recevant un pointeur sur `char` et la taille du tableau référencé qui lit un caractère jusqu'à être arrivé à la fin de la ligne ou à la limite du tableau.



Et comment sait-on que la lecture est arrivée à la fin d'une ligne ?

La fin de ligne est indiquée par le caractère `\n`.
Avec ceci, vous devriez pouvoir construire une fonction adéquate.
Allez, hop, au boulot et faites gaffe aux retours d'erreur !

```

1  #include <stdio.h>
2
3
4  int lire_ligne(char *chaine, size_t max)
5  {
6      size_t i;
7      char c;
8
9      for (i = 0; i < max - 1; ++i)
10     {
11         if (scanf("%c", &c) != 1)
12             return 0;
13         else if (c == '\n')
14             break;
15
16         chaine[i] = c;
17     }
18
19     chaine[i] = '\0';
20     return 1;
21 }
22
23
24 int main(void)
25 {
26     char chaine[255];
27
28     printf("Quel est votre prénom ? ");
29
30     if (lire_ligne(chaine, sizeof chaine))
31         printf("Bien le bonjour %s !\n", chaine);
32
33     return 0;
34 }
```

```

1  Quel est votre prénom ? Charles Henri
2  Bien le bonjour Charles Henri !
```



Gardez bien cette fonction sous le coude, nous allons en avoir besoin pour la suite.

18.4 Lire et écrire depuis et dans une chaîne de caractères

S'il est possible d'afficher et récupérer une chaîne de caractères, il est également possible de lire depuis une chaîne et d'écrire dans une chaîne. À cette fin, deux fonctions qui devraient vous sembler familières existent : `sprintf()` et `sscanf()`.

18.4.1 La fonction `sprintf`

```
1 int sprintf(char *chaine, char *format, ...);
```



Les trois petits points à la fin du prototype de la fonction signifient que celle-ci attend un nombre variable d'arguments. Nous verrons ce mécanisme plus en détail dans la troisième partie du cours.

La fonction `sprintf()` est identique à la fonction `printf()` mise à part que celle-ci écrit les données produites dans une chaîne de caractères au lieu de les afficher à l'écran. Celle-ci retourne le nombre de caractères écrit (sans compter le caractère nul final!) ou bien un nombre négatif en cas d'erreur.

Cette fonction peut vous permettre, entre autres, d'écrire un nombre dans une chaîne de caractères.

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     char chaine[16];
6     int n = 64;
7
8     sprintf(chaine, "%d", n);
9     printf("%s\n", chaine);
10    return 0;
11 }
```

```
1 64
```



La fonction `sprintf()` n'effectue *aucune* vérification quant à la taille de la chaîne de destination, vous devez donc vous assurer qu'elle dispose de suffisamment de place pour accueillir la chaîne finale (caractère nul compris!).

Comment dès lors s'assurer qu'il n'y aura aucun débordement ? Malheureusement, il n'est pas possible de spécifier une taille comme avec l'indicateur `s` de la fonction `scanf()`, aussi, deux solutions s'offrent à vous :

- vérifier que le nombre en question ne dépasse pas un certain seuil ;
- compter la quantité de chiffres composant le nombre avant d'appeler `sprintf()`.

Ainsi, l'exemple ci-dessous ne pose pas de problèmes puisque nous savons que si le nombre est inférieur ou égal à 999 999 999, il n'excèdera pas neuf caractères (*n'oubliez pas de compter le caractère nul final!*).

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4
5 int main(void)
6 {
7     char chaine[10];
8     long n;
```



```

9
10     printf("Entrez un nombre : ");
11
12     if (scanf("%ld", &n) != 1 || n > 999999999L)
13     {
14         printf("Erreur lors de la saisie\n");
15         return EXIT_FAILURE;
16     }
17
18     sprintf(chaine, "%ld", n);
19     printf("%s\n", chaine);
20     return 0;
21 }

```

```

1  Entrez un nombre : 890765456789
2  Erreur lors de la saisie
3
4  Entrez un nombre : 5678
5  5678

```

18.4.2 La fonction sscanf

```

1  int sscanf(char *chaine, char *format, ...);

```

La fonction `sscanf()` est identique à la fonction `scanf()` si ce n'est que celle-ci extrait les données depuis une chaîne de caractères plutôt qu'en provenance d'une saisie de l'utilisateur. Cette dernière retourne le nombre de conversions réussies *ou* un nombre inférieur si elles n'ont pas toutes été réalisées *ou* enfin un nombre négatif en cas d'erreur.

Voici un exemple d'utilisation.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4
5  int main(void)
6  {
7      char chaine[10];
8      int n;
9
10     if (sscanf("5 abcd", "%d %9s", &n, chaine) != 2)
11     {
12         printf("Erreur lors de l'examen de la chaîne\n");
13         return EXIT_FAILURE;
14     }
15
16     printf("%d %s\n", n, chaine);
17     return 0;
18 }

```

```

1  5 abcd

```



Notez que la fonction `sscanf()` ne souffre pas du même problème que `scanf()` en ce qui concerne de potentiels caractères non lus, nous y reviendrons un peu plus tard.

18.5 Les classes de caractères

Pour terminer cette partie théorique sur une note un peu plus légère, sachez que la bibliothèque standard fournit un en-tête `<ctype.h>` qui permet de classer les caractères. Onze fonctions sont ainsi définies

```

1 int isalnum(int c);
2 int isalpha(int c);
3 int iscntrl(int c);
4 int isdigit(int c);
5 int isgraph(int c);
6 int islower(int c);
7 int isprint(int c);
8 int ispunct(int c);
9 int isspace(int c);
10 int isupper(int c);
11 int isxdigit(int c);

```

Chacune d'entre elles attend en argument un caractère et retourne un nombre positif ou zéro suivant que le caractère fourni appartienne ou non à la catégorie déterminée par la fonction.

Fonction	Catégorie	Description	Par défaut
isupper	Majuscule	Détermine si le caractère entré est une lettre majuscule	'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y' ou 'Z'
islower	Minuscule	Détermine si le caractère entré est une lettre minuscule	'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y' ou 'z'
isdigit	Chiffre décimal	Détermine si le caractère entré est un chiffre décimal	'0', '1', '2', '3', '4', '5', '6', '7', '8' ou '9'
isxdigit	Chiffre hexadécimal	Détermine si le caractère entré est un chiffre hexadécimal	'0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F', 'a', 'b', 'c', 'd', 'e' ou 'f'
isspace	Espace	Détermine si le caractère entré est un espace	' ', '\f', '\n', '\r', '\t' ou '\v'
iscntrl	Contrôle	Détermine si le caractère est un caractère dit « de contrôle »	'\0', '\a', '\b', '\f', '\n', '\r', '\t' ou '\v'
ispunct	Ponctuation	Détermine si le caractère entré est un signe de ponctuation	'!', '"', '#', '%', '&', '(', ')', '*', '+', ',', '-', '.', ':', ';', '<', '=', '>', '?', '[', '\', ']', '^', '_', '{', '<barre droite>', '}' ou '~'
isalpha	Alphabétique	Détermine si le caractère entré est une lettre alphabétique	Les deux ensembles de caractères de <code>islower()</code> et <code>isupper()</code>
isalnum	Alphanumérique	Détermine si le caractère entré est une lettre alphabétique ou un chiffre décimal	Les trois ensembles de caractères de <code>islower()</code> , <code>isupper()</code> et <code>isdigit()</code>
isgraph	Graphique	Détermine si le caractère est représentable graphiquement	Tout sauf l'ensemble de <code>iscntrl()</code> et l'espace (' ')
isprint	Affichable	Détermine si le caractère est « affichable »	Tout sauf l'ensemble de <code>iscntrl()</code>

 La suite `<barre_droite>` symbolise le caractère `␣`.

Le tableau ci-dessous vous présente chacune de ces onze fonctions ainsi que les ensembles de caractères qu'elles décrivent. La colonne « par défaut » vous détail leur comportement en cas d'utilisation de la *locale* `C`. Nous reviendrons sur les *locales* dans la troisième partie de ce cours ; pour l'heure, considérez que ces fonctions ne retournent un nombre positif que si un des caractères de leur ensemble « par défaut » leur est fourni en argument.

L'exemple ci-dessous utilise la fonction `isdigit()` pour déterminer si l'utilisateur a bien entré une suite de chiffres.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4
5 /* lire_ligne() */
6
7

```

```

8  int main(void)
9  {
10     char suite[255];
11     unsigned i;
12
13     if (!lire_ligne(suite, sizeof suite))
14     {
15         printf("Erreur lors de la saisie.\n");
16         return EXIT_FAILURE;
17     }
18
19     for (i = 0; suite[i] != '\0'; ++i)
20         if (!isdigit(suite[i]))
21         {
22             printf("Veuillez entrer une suite de chiffres.\n");
23             return EXIT_FAILURE;
24         }
25
26     printf("C'est bien une suite de chiffres.\n");
27     return 0;
28 }

```

```

1  122334
2  C'est bien une suite de chiffres.
3
4  5678a
5  Veuillez entre une suite de chiffres.

```



Notez que cet en-tête fournit également deux fonctions : `tolower()` et `toupper()` retournant respectivement la version minuscule ou majuscule de la lettre entrée. Dans le cas où un caractère autre qu'une lettre est entré (ou que celle-ci est déjà en minuscule ou en majuscule), la fonction retourne celui-ci.

18.6 Exercices

18.6.1 Palindrome

Un palindrome est un texte identique lorsqu'il est lu de gauche à droite et de droite à gauche. Ainsi, le mot *radar* est un palindrome, de même que les phrases *engage le jeu que je le gagne* et *élu par cette crapule*. Normalement, il n'est pas tenu compte des accents, trémas, cédilles ou des espaces. Toutefois, pour cet exercice, nous nous contenterons de vérifier si un mot donné est un palindrome.

```

1  int palindrome(char *s)
2  {
3      size_t len = 0;
4      size_t i;
5
6      while (s[len] != '\0')
7          ++len;
8
9      for (i = 0; i < len; ++i)
10         if (s[i] != s[len - 1 - i])
11             return 0;
12
13     return 1;
14 }

```

18.6.2 Compter les parenthèses

Écrivez un programme qui lit une ligne et vérifie que chaque parenthèse ouvrante est bien refermée par la suite.

```
1 Entrez une ligne : printf("%u\n", (unsigned)sizeof(int);
2 Il manque des parenthèses.
```

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 /* lire_ligne() */
5
6
7 int main(void)
8 {
9     char s[255];
10    char *t;
11    long n = 0;
12
13    printf("Entrez une ligne : ");
14
15    if (!lire_ligne(s, sizeof s))
16    {
17        printf("Erreur lors de la saisie.\n");
18        return EXIT_FAILURE;
19    }
20
21    for (t = s; *t != '\0'; ++t)
22        if (*t == '(')
23            ++n;
24        else if (*t == ')')
25            --n;
26
27    if (n == 0)
28        printf("Le compte est bon.\n");
29    else
30        printf("Il manque des parenthèses.\n");
31
32    return 0;
33 }
```

Le chapitre suivant sera l'occasion de mettre en pratique ce que nous avons vu dans les chapitres précédents à l'aide d'un TP.

TP : l'en-tête `<string.h>`

Les quatre derniers cours ayant été riche en nouveautés, posons nous un instant pour mettre en pratique tout cela.

19.1 Préparation

Dans le chapitre précédent, nous vous avons dit qu'une chaîne de caractères se manipulait comme un tableau, à savoir en parcourant ses éléments un à un. Cependant, si cela s'arrêtait là, cela serait assez gênant. En effet, la moindre opération sur une chaîne nécessiterait d'accéder à ses différents éléments, que ce soit une modification, une copie, une comparaison, etc. Heureusement pour nous, la bibliothèque standard fournit une suite de fonction nous permettant de réaliser plusieurs opérations de base sur des chaînes de caractères. Ces fonctions sont déclarées dans l'en-tête `<string.h>`.

L'objectif de ce TP sera de réaliser une version pour chacune des fonctions de cet en-tête qui vont vous être présentées. :)

19.1.1 strlen

```
1 size_t strlen(char *chaine);
```

La fonction `strlen()` vous permet de connaître la taille d'une chaîne fournie en argument. Celle-ci retourne une valeur de type `size_t`. Notez bien que la longueur retournée ne comprend *pas* le caractère nul. L'exemple ci-dessous affiche la taille de la chaîne « Bonjour ».

```
1 #include <stdio.h>
2 #include <string.h>
3
4
5 int main(void)
6 {
7     printf("Longueur : %u\n", (unsigned)strlen("Bonjour"));
8     return 0;
9 }
```

```
1 Longueur : 7
```

19.1.2 strcpy

```
1 char *strcpy(char *destination, char *source);
```

La fonction `strcpy()` copie le contenu de la chaîne `source` dans la chaîne `destination`, caractère nul compris. La fonction retourne l'adresse de `destination`. L'exemple ci-dessous copie la chaîne « Au revoir » dans la chaîne `chaine`.

```

1  #include <stdio.h>
2
3
4  int main(void)
5  {
6      char chaine[25] = "Bonjour\n";
7
8      strcpy(chaine, "Au revoir");
9      printf("%s\n", chaine);
10     return 0;
11 }
```

```

1  Au revoir
```



La fonction `strcpy()` n'effectue *aucune* vérifications. Vous devez donc vous assurer que la chaîne de destination dispose de suffisamment d'espace pour accueillir la chaîne qui doit être copiée (caractère nul compris!).

19.1.3 strcat

```

1  char *strcat(char *destination, char *source);
```

La fonction `strcat()` ajoute le contenu de la chaîne `source` à celui de la chaîne `destination`, caractère nul compris. La fonction retourne l'adresse de `destination`. L'exemple ci-dessous ajoute la chaîne « tout le monde » au contenu de la chaîne `chaine`.

```

1  #include <stdio.h>
2  #include <string.h>
3
4
5  int main(void)
6  {
7      char chaine[25] = "Bonjour";
8
9      strcat(chaine, " tout le monde");
10     printf("%s\n", chaine);
11     return 0;
12 }
```

```

1  Bonjour tout le monde
```



Comme `strcpy()`, la fonction `strcat()` n'effectue *aucune* vérifications. Vous devez donc vous assurer que la chaîne de destination dispose de suffisamment d'espace pour accueillir la chaîne qui doit être ajoutée (caractère nul compris!).

19.1.4 strcmp

```

1  int strcmp(char *chaine1, char *chaine2);
```

La fonction `strcmp()` compare deux chaînes de caractères. Cette fonction retourne :

- une valeur positive si la première chaîne est « plus grande » que la seconde ;
- zéro si elles sont égales ;
- une valeur négative si la seconde chaîne est « plus grande » que la première.



Ne vous inquiétez pas au sujet des valeurs positives et négatives, nous y reviendront en temps voulu lorsque nous aborderons les notions de jeux de caractères et d'encodages dans la troisième partie du cours. En attendant, effectuez simplement une comparaison entre les deux caractères qui diffèrent.

```

1  #include <stdio.h>
2  #include <string.h>
3
4
5  int main(void)
6  {
7      char chaine1[] = "Bonjour";
8      char chaine2[] = "Au revoir";
9
10     if (strcmp(chaine1, chaine2) == 0)
11         printf("Les deux chaînes sont identiques\n");
12     else
13         printf("Les deux chaînes sont différentes\n");
14
15     return 0;
16 }
```

```

1  Les deux chaînes sont différentes
```

19.1.5 strchr

```

1  char *strchr(char *chaine, int ch);
```

La fonction `strchr()` recherche la présence du caractère `ch` dans la chaîne `chaine`. Si celui-ci est rencontré, la fonction retourne l'adresse de la première occurrence de celui-ci au sein de la chaîne. Dans le cas contraire, la fonction renvoie un pointeur nul.

```

1  #include <stddef.h>
2  #include <stdio.h>
3  #include <string.h>
4
5
6  int main(void)
7  {
8      char chaine[] = "Bonjour";
9      char *p;
10
11     p = strchr(chaine, 'o');
12
13     if (p != NULL)
14         printf("La chaîne '%s' contient la lettre %c\n", chaine, *p);
15
16     return 0;
17 }
```

```

1  La chaîne "Bonjour" contient la lettre o
```

19.1.6 strpbrk

```

1  char *strpbrk(char *chaine, char *liste);
```

La fonction `strpbrk()` recherche la présence d'un des caractères de la chaîne `liste` dans la chaîne `chaine`. Si un de ceux-ci est rencontré, la fonction retourne l'adresse de la première occurrence au sein de la chaîne. Dans le cas contraire, la fonction renvoie un pointeur nul.


```

1  #include <stddef.h>
2  #include <stdio.h>
3  #include <string.h>
4
5
6  int main(void)
7  {
8      char chaine[] = "Bonjour";
9      char *p;
10
11     p = strpbrk(chaine, "aeiouy");
12
13     if (p != NULL)
14         printf("La première voyelle de la chaîne '%s' est : %c\n", chaine, *p);
15
16     return 0;
17 }

```

```

1  La première voyelle de la chaîne 'Bonjour' est : o

```

19.1.7 strstr

```

1  char *strstr(char *chaine1, char *chaine2);

```

La fonction `strstr()` recherche la présence de la chaîne `chaine2` dans la chaîne `chaine1`. Si celle-ci est rencontrée, la fonction retourne l'adresse de la première occurrence de celle-ci au sein de la chaîne. Dans le cas contraire, la fonction renvoie un pointeur nul.

```

1  #include <stddef.h>
2  #include <stdio.h>
3  #include <string.h>
4
5
6  int main(void)
7  {
8      char chaine[] = "Bonjour";
9      char *p;
10
11     p = strstr(chaine, "jour");
12
13     if (p != NULL)
14         printf("La chaîne '%s' contient la chaîne '%s'\n", chaine, p);
15
16     return 0;
17 }

```

```

1  La chaîne 'Bonjour' contient la chaîne 'jour'

```

Ceci étant dit, à vous de jouer. ;)



Par convention, nous commencerons le nom de nos fonctions par la lettre « x » afin d'éviter des collisions avec ceux de l'en-tête `<string.h>`.

19.1.8 strlen

```

1  size_t strlen(char *chaine)
2  {
3      size_t i;
4
5      for (i = 0; chaine[i] != '\0'; ++i)
6          ;
7
8      return i;
9  }

```

19.1.9 strcpy

```

1 char *strcpy(char *destination, char *source)
2 {
3     size_t i;
4
5     for (i = 0; source[i] != '\0'; ++i)
6         destination[i] = source[i];
7
8     destination[i] = '\0' /* N'oubliez pas le caractère nul final ! */
9     return destination;
10 }

```

19.1.10 strcat

```

1 char *strcat(char *destination, char *source)
2 {
3     size_t i;
4
5     while (*destination != '\0')
6         ++destination;
7
8     for (i = 0; source[i] != '\0'; ++i)
9         destination[i] = source[i];
10
11     destination[i] = '\0'; /* N'oubliez pas le caractère nul final ! */
12     return destination;
13 }

```

19.1.11 strcmp

```

1 int strcmp(char *chaine1, char *chaine2)
2 {
3     while (*chaine1 == *chaine2)
4     {
5         if (*chaine1 == '\0')
6             return 0;
7
8         ++chaine1;
9         ++chaine2;
10    }
11
12    return (*chaine1 < *chaine2) ? -1 : 1;
13 }

```

19.1.12 strchr

```

1 char *strchr(char *chaine, int ch)
2 {
3     while (*chaine != '\0')
4         if (*chaine == ch)
5             return chaine;
6         else
7             ++chaine;
8
9     return NULL;
10 }

```

19.1.13 strpbrk

```

1 char *strpbrk(char *chaine, char *liste)
2 {
3     while (*chaine != '\0')
4     {
5         char *p = liste;
6
7         while (*p != '\0')

```

```

8      {
9          if (*chaine == *p)
10             return chaine;
11
12         ++p;
13     }
14
15     ++chaine;
16 }
17
18 return NULL;
19 }

```

19.1.14 strstr

```

1 char *strstr(char *chaine1, char *chaine2)
2 {
3     while (*chaine1 != '\0')
4     {
5         char *s = chaine1;
6         char *t = chaine2;
7
8         while (*s != '\0' && *t != '\0')
9         {
10             if (*s != *t)
11                 break;
12
13             ++s;
14             ++t;
15         }
16
17         if (*t == '\0')
18             return chaine1;
19
20         ++chaine1;
21     }
22
23     return NULL;
24 }

```

19.2 Pour aller plus loin : strtok

Pour les plus aventureux d'entre-vous, nous vous proposons de réaliser en plus la mise en œuvre de la fonction `strtok()` qui est un peu plus complexe que ses congénères. :)

```

1 char *strtok(char *chaine, char *liste);

```

La fonction `strtok()` est un peu particulière : elle divise la chaîne `chaine` en une suite de sous-chaînes délimitée par *un* des caractères présent dans la chaîne `liste`. En fait, cette dernière remplace les caractères de la chaîne `liste` par des caractères nuls dans la chaîne `chaine` (elle modifie donc la chaîne `chaine` !) et renvoie les différentes sous-chaînes ainsi créées au fur et à mesure de ses appels.

Lors des appels subséquents, la chaîne `chaine` doit être un pointeur nul afin de signaler à `strtok()` que nous souhaitons la sous-chaîne suivante. S'il n'y a plus de sous-chaîne ou qu'aucun caractère de la chaîne `liste` n'est trouvé, celle-ci retourne un pointeur nul.

L'exemple ci-dessous tente de scinder la chaîne « un, deux, trois » en trois sous-chaînes.

```

1 #include <stddef.h>
2 #include <stdio.h>
3 #include <string.h>
4
5
6 int main(void)

```

```

7 {
8     char chaine[] = "un, deux, trois";
9     char *p;
10    unsigned i;
11
12    p = strtok(chaine, ", ");
13
14    for (i = 0; p != NULL; ++i)
15    {
16        printf("%u : %s\n", i, p);
17        p = strtok(NULL, ", ");
18    }
19
20    return 0;
21 }

```

```

1 0 : un
2 1 : deux
3 2 : trois

```

La fonction `strtok()` étant un peu plus complexe que les autres, voici une petite marche à suivre pour réaliser cette fonction.

- regarder si la chaîne fournie (ou la sous-chaîne suivante) contient, à *son début*, des caractères présent dans la seconde chaîne. Si oui, ceux-ci doivent être passés;
- si la fin de la (sous-)chaîne est atteinte, retourner un pointeur nul;
- parcourir la chaîne fournie (ou la sous-chaîne courante) jusqu'à rencontrer un des caractères composant la seconde chaîne. Si un caractère est rencontré, le remplacer par un caractère nul, conserver la position actuelle dans la chaîne et retourner la sous-chaîne ainsi créée (*sans* les éventuels caractères passés au début);
- si la fin de la (sous-)chaîne est atteinte, retourner la (sous-)chaîne (*sans* les éventuels caractères passés au début).



Vous aurez besoin d'une variable de classe de stockage statique pour réaliser cette fonction. Également, l'instruction `goto` pourra vous être utile.

19.2.1 Correction

```

1 char *xstrtok(char *chaine, char *liste)
2 {
3     static char *dernier;
4     char *base = (chaine != NULL) ? chaine : dernier;
5     char *s;
6     char *t;
7
8     if (base == NULL)
9         return NULL;
10
11    separateur_au_debut:
12    for (t = liste; *t != '\0'; ++t)
13        if (*base == *t)
14        {
15            ++base;
16            goto separateur_au_debut;
17        }
18
19    if (*base == '\0')
20    {
21        dernier = NULL;
22        return NULL;
23    }
24
25    for (s = base; *s != '\0'; ++s)
26        for (t = liste; *t != '\0'; ++t)
27            if (*s == *t)

```

```
28     {
29         *s = '\\0';
30         dernier = s + 1;
31         return base;
32     }
33
34     dernier = NULL;
35     return base;
36 }
```

le chapitre suivant, nous aborderons le mécanisme de l'**allocation dynamique de mémoire**.

Il est à présent temps d'aborder la troisième et dernière utilisation majeure des pointeurs : l'allocation dynamique de mémoire.

Comme nous vous l'avons dit dans le chapitre sur les pointeurs, il n'est pas toujours possible de savoir quelle quantité de mémoire sera utilisée par un programme. Par exemple, si vous demandez à l'utilisateur de vous fournir un tableau, vous devrez lui fixer une limite, ce qui pose deux problèmes :

- la limite en elle-même, qui ne convient peut-être pas à votre utilisateur ;
- l'utilisation excessive de mémoire du fait que vous réservez un tableau d'une taille fixée à l'avance.

De plus, si vous utilisez un tableau de classe de stockage statique, alors cette quantité de mémoire superflue sera inutilisable jusqu'à la fin de votre programme.

Or, un ordinateur ne dispose que d'une quantité limitée de mémoire vive, il est donc important de ne pas en réserver abusivement. L'allocation dynamique permet de réserver une partie de la mémoire vive inutilisée pour stocker des données et de libérer cette même partie une fois qu'elle n'est plus nécessaire.

20.1 La notion d'objet

Jusqu'à présent, nous avons toujours recouru au système de types et de variables du langage C pour stocker nos données sans jamais vraiment nous soucier de ce qui se passait « en-dessous ».

Il est à présent temps de lever une partie de ce voile en abordant la notion d'**objet**.

En C, un objet est une zone mémoire pouvant contenir des données et est composé d'une suite contiguë d'un ou plusieurs multipliets. En fait, tous les types du langage C manipulent des objets. La différence entre les types tient simplement en la manière dont ils répartissent les données au sein de ces objets, ce qui est appelé leur **représentation** (celle-ci sera abordée dans la troisième partie du cours). Ainsi, la valeur 1 n'est pas représentée de la même manière dans un objet de type `int` que dans un objet de type `double`.

Un objet étant une suite contiguë de multipliets, il est possible d'en examiner le contenu en lisant ses multipliets un à un. Ceci peut se réaliser en C à l'aide de l'adresse de l'objet et d'un pointeur sur `unsigned char`, le type `char` du C ayant *toujours* la taille d'un multipliet. Notez qu'il est *impératif* d'utiliser la version non signée du type `char` afin d'éviter des problèmes de conversions.

L'exemple ci-dessous affiche les multipliets composant un objet de type `int` et un objet de type `double` en hexadécimal.

```
1 #include <stdio.h>
2
3
```

```

4  int main(void)
5  {
6      int n = 1;
7      double f = 1.;
8      unsigned char *byte;
9      unsigned i;
10
11     byte = (unsigned char *)&n;
12
13     for (i = 0; i < sizeof n; ++i)
14         printf("%x ", byte[i]);
15
16     printf("\n");
17     byte = (unsigned char *)&f;
18
19     for (i = 0; i < sizeof f; ++i)
20         printf("%x ", byte[i]);
21
22     printf("\n");
23     return 0;
24 }

```

```

1  1 0 0 0
2  0 0 0 0 0 0 f0 3f

```



Il se peut que vous n'obteniez pas le même résultat que nous, ce dernier dépend de votre machine.

Comme vous le voyez, la représentation de la valeur 1 n'est pas du tout la même entre le type `int` et le type `double`.

20.2 Malloc et consueurs

La bibliothèque standard fournit trois fonctions vous permettant d'allouer de la mémoire : `malloc()`, `calloc()` et `realloc()` et une vous permettant de la libérer : `free()`. Ces quatre fonctions sont déclarées dans l'en-tête `<stdlib.h>`.

20.2.1 malloc

```

1  void *malloc(size_t taille);

```

La fonction `malloc()` vous permet d'allouer un objet de la taille fournie en argument (qui représente un nombre de multiplés) et retourne l'adresse de cet objet sous la forme d'un pointeur générique. En cas d'échec de l'allocation, elle retourne un pointeur nul.



Vous devez *toujours* vérifier le retour d'une fonction d'allocation afin de vous assurer que vous manipulez bien un pointeur valide.

Allocation d'un objet

Dans l'exemple ci-dessous, nous réservons un objet de la taille d'un `int`, nous y stockons ensuite la valeur dix et l'affichons. Pour cela, nous utilisons un pointeur sur `int` qui va se voir affecter l'adresse de l'objet ainsi alloué et qui va nous permettre de le manipuler comme nous le ferions s'il référençait une variable de type `int`.

```

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

```

```

3
4
5 int main(void)
6 {
7     int *p;
8
9     p = malloc(sizeof(int));
10
11     if (p == NULL)
12     {
13         printf("Échec de l'allocation\n");
14         return EXIT_FAILURE;
15     }
16
17     *p = 10;
18     printf("%d\n", *p);
19     return 0;
20 }

```

```

1 10

```

Allocation d'un tableau

Pour allouer un tableau, vous devez réserver un bloc mémoire de la taille d'un élément multiplié par le nombre d'éléments composant le tableau. L'exemple suivant alloue un tableau de dix `int`, l'initialise et affiche son contenu.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4
5 int main(void)
6 {
7     int *p;
8     unsigned i;
9
10    p = malloc(sizeof(int) * 10);
11
12    if (p == NULL)
13    {
14        printf("Échec de l'allocation\n");
15        return EXIT_FAILURE;
16    }
17
18    for (i = 0; i < 10; ++i)
19    {
20        p[i] = i * 10;
21        printf("p[%u] = %d\n", i, p[i]);
22    }
23
24    return 0;
25 }

```

```

1 p[0] = 0
2 p[1] = 10
3 p[2] = 20
4 p[3] = 30
5 p[4] = 40
6 p[5] = 50
7 p[6] = 60
8 p[7] = 70
9 p[8] = 80
10 p[9] = 90

```



Autrement dit et de manière plus générale : pour allouer dynamiquement un objet de type **T**, il vous faut créer un pointeur sur le type **T** qui conservera son adresse.



La fonction `malloc()` n'effectue *aucune* initialisation, le contenu du bloc alloué est donc indéterminé.

20.2.2 free

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

La fonction `free()` libère le bloc précédemment alloué par une fonction d'allocation dont l'adresse est fournie en argument. Dans le cas où un pointeur nul lui est fourni, elle n'effectue aucune opération.



Retenez bien la règle suivante : à chaque appel à une fonction d'allocation doit correspondre un appel à la fonction `free()`.

Dès lors, nous pouvons compléter les exemples précédents comme suit.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4
5 int main(void)
6 {
7     int *p;
8
9     p = malloc(sizeof(int));
10
11     if (p == NULL)
12     {
13         printf("Échec de l'allocation\n");
14         return EXIT_FAILURE;
15     }
16
17     *p = 10;
18     printf("%d\n", *p);
19     free(p);
20     return 0;
21 }
```

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4
5 int main(void)
6 {
7     int *p;
8     unsigned i;
9
10    p = malloc(sizeof(int) * 10);
11
12    if (p == NULL)
13    {
14        printf("Échec de l'allocation\n");
15        return EXIT_FAILURE;
16    }
17
18    for (i = 0; i < 10; ++i)
19    {
20        p[i] = i * 10;
21        printf("p[%u] = %d\n", i, p[i]);
22    }
23
24    free(p);
25    return 0;
26 }
```



Remarquez que même si le deuxième exemple alloue un tableau, il n'y a bien eu qu'une seule allocation. Un seul appel à la fonction `free()` est donc nécessaire.

20.2.3 calloc

```
1 void *calloc(size_t nombre, size_t taille);
```

La fonction `calloc()` attend deux arguments : le nombre d'éléments à allouer et la taille de chacun de ces éléments. Techniquement, elle revient au même que d'appeler `malloc()` comme suit.

```
1 malloc(nombre * taille);
```

à un détail près : *le bloc de mémoire ainsi alloué est initialisé à zéro.*



Faites attention : cette initialisation n'est pas similaire à celle des variables de classe de stockage statique ! À l'inverse de ces dernières qui seront soit initialisées à zéro, soit seront des pointeurs nuls, l'initialisation réalisée par la fonction `calloc()` ne s'applique qu'aux entiers ou aux chaînes de caractères (nous verrons cela plus en détails dans la troisième partie du cours).

20.2.4 realloc

```
1 void *realloc(void *p, size_t taille);
```

La fonction `realloc()` libère un bloc de mémoire précédemment alloué, en réserve un nouveau de la taille demandée et copie le contenu de l'ancien objet dans le nouveau. Dans le cas où la taille demandée est inférieure à celle du bloc d'origine, le contenu de celui-ci sera copié à hauteur de la nouvelle taille. À l'inverse, si la nouvelle taille est supérieure à l'ancienne, l'excédant n'est pas initialisé.

La fonction attend deux arguments : l'adresse d'un bloc précédemment alloué à l'aide d'une fonction d'allocation et la taille du nouveau bloc à allouer. Elle retourne l'adresse du nouveau bloc ou un pointeur nul en cas d'erreur.

L'exemple ci-dessous alloue un tableau de dix `int` et utilise `realloc()` pour agrandir celui-ci à vingt `int`.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4
5 int main(void)
6 {
7     int *p;
8     int *tmp;
9     unsigned i;
10
11     p = malloc(sizeof(int) * 10);
12
13     if (p == NULL)
14     {
15         printf("Échec de l'allocation\n");
16         return EXIT_FAILURE;
17     }
18
19     for (i = 0; i < 10; ++i)
20         p[i] = i * 10;
21
```

```

22     tmp = realloc(p, sizeof(int) * 20);
23
24     if (tmp == NULL)
25     {
26         printf("Échec de l'allocation\n");
27         return EXIT_FAILURE;
28     }
29
30     p = tmp;
31
32     for (i = 10; i < 20; ++i)
33         p[i] = i * 10;
34
35     for (i = 0; i < 20; ++i)
36         printf("p[%u] = %d\n", i, p[i]);
37
38     free(p);
39     return 0;
40 }

```

```

1  p[0] = 0
2  p[1] = 10
3  p[2] = 20
4  p[3] = 30
5  p[4] = 40
6  p[5] = 50
7  p[6] = 60
8  p[7] = 70
9  p[8] = 80
10 p[9] = 90
11 p[10] = 100
12 p[11] = 110
13 p[12] = 120
14 p[13] = 130
15 p[14] = 140
16 p[15] = 150
17 p[16] = 160
18 p[17] = 170
19 p[18] = 180
20 p[19] = 190

```

Remarquez que nous avons utilisé une autre variable, `tmp`, pour vérifier le retour de la fonction `realloc()`. En effet, si nous avions procédé comme ceci.

```

1  p = realloc(p, sizeof(int) * 20);

```

il nous aurait été impossible de libérer le bloc mémoire référencé par `p` en cas d'erreur puisque celui-ci serait devenu un pointeur nul. Il est donc *impératif* d'utiliser une seconde variable afin d'éviter des *fuites de mémoire*.

20.3 Les tableaux multidimensionnels

L'allocation de tableaux multidimensionnels est un petit peu plus complexe que celles des autres objets. Techniquement, il existe deux solutions : l'allocation d'un seul bloc de mémoire (comme pour les tableaux simples) et l'allocation de plusieurs tableaux eux-mêmes référencés par les éléments d'un autre tableau.

20.3.1 Allocation en un bloc

Comme pour un tableau simple, il vous est possible d'allouer un bloc de mémoire dont la taille correspond à la multiplication des longueurs de chaque dimension, elle-même multipliée par la taille d'un élément. Toutefois, cette solution vous contraint à effectuer une partie du calcul d'adresse vous-même puisque vous allouez en fait un seul tableau.

L'exemple ci-dessous illustre ce qui vient d'être dit en allouant un tableau à deux dimensions de trois fois trois `int`.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4
5  int main(void)
6  {
7      int *p;
8      unsigned i;
9      unsigned j;
10
11     p = malloc(3 * 3 * sizeof(int));
12
13     if (p == NULL)
14     {
15         printf("Échec de l'allocation\n");
16         return EXIT_FAILURE;
17     }
18
19     for (i = 0; i < 3; ++i)
20         for (j = 0; j < 3; ++j)
21         {
22             p[(i * 3) + j] = (i * 3) + j;
23             printf("p[%u][%u] = %d\n", i, j, p[(i * 3) + j]);
24         }
25
26     free(p);
27     return 0;
28 }
```

```

1  p[0][0] = 0
2  p[0][1] = 1
3  p[0][2] = 2
4  p[1][0] = 3
5  p[1][1] = 4
6  p[1][2] = 5
7  p[2][0] = 6
8  p[2][1] = 7
9  p[2][2] = 8
```

Comme vous le voyez, une partie du calcul d'adresse doit être effectué en multipliant le premier indice par la longueur de la première dimension, ce qui permet d'arriver à la bonne « ligne ». Ensuite, il ne reste plus qu'à sélectionner le bon élément de la ligne à l'aide du second indice.

Bien qu'un petit peu plus complexe quant à l'accès aux éléments, cette solution à l'avantage de n'effectuer qu'une seule allocation de mémoire.

20.3.2 Allocation de plusieurs tableaux

La seconde solution consiste à allouer plusieurs tableaux plus un autre qui les référencera. Dans le cas d'un tableau à deux dimensions, cela signifie allouer un tableau de pointeurs dont chaque élément se verra affecter l'adresse d'un tableau également alloué dynamiquement. Cette technique nous permet d'accéder aux éléments des différents tableaux de la même manière que pour un tableau multidimensionnel puisque nous utilisons cette fois plusieurs tableaux.

L'exemple ci-dessous revient au même que le précédent, mais utilise le procédé qui vient d'être décrit. Notez que puisque nous réservons un tableau de pointeurs sur `int`, l'adresse de celui-ci doit être stockée dans un pointeur de pointeur sur `int` (rappelez-vous la règle générale lors de la présentation de la fonction `malloc()`).

```

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

```
3
4
5 int main(void)
6 {
7     int **p;
8     unsigned i;
9     unsigned j;
10
11     p = malloc(3 * sizeof(int *));
12
13     if (p == NULL)
14     {
15         printf("Échec de l'allocation\n");
16         return EXIT_FAILURE;
17     }
18
19     for (i = 0; i < 3; ++i)
20     {
21         p[i] = malloc(3 * sizeof(int));
22
23         if (p[i] == NULL)
24         {
25             printf("Échec de l'allocation\n");
26             return EXIT_FAILURE;
27         }
28     }
29
30     for (i = 0; i < 3; ++i)
31         for (j = 0; j < 3; ++j)
32         {
33             p[i][j] = (i * 3) + j;
34             printf("p[%u][%u] = %d\n", i, j, p[i][j]);
35         }
36
37     for (i = 0; i < 3; ++i)
38         free(p[i]);
39
40     free(p);
41     return 0;
42 }
```

Si cette solution permet de faciliter l'accès aux différents éléments, elle présente toutefois l'inconvénient de réaliser plusieurs allocations et donc de nécessiter plusieurs appels à la fonction `free()`.

Ce chapitre nous aura permis de découvrir la notion d'objet ainsi que le mécanisme de l'allocation dynamique. Dans le chapitre suivant, nous verrons comment manipuler les **fichiers**.

Jusqu'à présent, nous n'avons manipulé que des données en mémoire, ce qui nous empêchait de les stocker de manière permanente. Dans ces deux chapitres, nous allons voir comment conserver des informations de manière permanente à l'aide des fichiers.

21.1 Les fichiers

En informatique, un fichier est un ensemble d'informations stockées sur un support, réuni sous un même nom et manipulé comme une unité.

21.1.1 Extension de fichier

Le contenu de chaque fichier est lui-même organisé suivant un format qui dépend des données qu'il contient. Il en existe une pléthore pour chaque type de données :

- audio : Ogg, MP3, MP4, FLAC, Wave, etc. ;
- vidéo : Ogg, WebM, MP4, AVI, etc. ;
- documents : ODT, DOC et DOCX, XLS et XLSX, PDF, Postscript, etc.

Afin d'aider l'utilisateur, ces formats sont le plus souvent indiqués à l'aide d'une **extension** qui se traduit par un suffixe ajouté au nom de fichier. Toutefois, cette extension est purement *indicative* et *facultative*, elle n'influence *en rien* le contenu du fichier. Son seul objectif est d'aider à déterminer le type de contenu d'un fichier.

21.1.2 Système de fichiers

Afin de faciliter leur localisation et leur gestion, les fichiers sont classés et organisés sur leur support suivant un **système de fichiers**. C'est lui qui permet à l'utilisateur de répartir ses fichiers dans une arborescence de dossiers et de localiser ces derniers à partir d'un chemin d'accès.

Le chemin d'accès

Le chemin d'accès d'un fichier est une suite de caractères décrivant la position de celui-ci dans le système de fichiers. Un chemin d'accès se compose au minimum du nom du fichier visé et au maximum de la suite de tous les dossiers qu'il est nécessaire de traverser pour l'atteindre depuis le **répertoire racine**. Ces éventuels dossiers à traverser sont séparés par un caractère spécifique qui est `/` sous Unixöide et `\` sous Windows.

La **racine** d'un système de fichier est le point de départ de l'arborescence des fichiers et dossiers. Sous Unixöides, il s'agit du répertoire `/` tandis que sous Windows, chaque lecteur est une racine (comme `C:` par exemple). Si un chemin d'accès commence par la racine, alors

celui-ci est dit **absolu** car il permet d'atteindre le fichier depuis n'importe quelle position dans l'arborescence. Si le chemin d'accès ne commence pas par la racine, il est dit **relatif** et ne permet de parvenir au fichier que depuis un point précis dans la hiérarchie de fichiers.

Ainsi, si nous souhaitons accéder à un fichier nommé « `texte.txt` » situé dans le dossier « `documents` » lui-même situé dans le dossier « `utilisateur` » qui est à la racine, alors le chemin d'accès absolu vers ce fichier serait `/utilisateur/documents/texte.txt` sous Unixöide et `C:\utilisateur\documents\texte.txt` sous Windows (à supposer qu'il soit sur le lecteur `C:`). Toutefois, si nous sommes déjà dans le dossier « `utilisateur` », alors nous pouvons y accéder à l'aide du chemin relatif `documents/texte.txt` ou `documents\texte.txt`. Néanmoins, ce chemin relatif n'est utilisable que si nous sommes dans le dossier « `utilisateur` ».

Métadonnées

Également, le système de fichier se charge le plus souvent de conserver un certain nombre de données concernant chaque fichier comme :

- sa taille ;
- ses droits d'accès (les utilisateurs autorisés à le manipuler) ;
- la date de dernier accès ;
- la date de dernière modification ;
- etc.

21.2 Les flux : un peu de théorie

La bibliothèque standard vous fournit différentes fonctions pour manipuler les fichiers, toutes déclarées dans l'en-tête `<stdio.h>`. Toutefois, celles-ci manipulent non pas des fichiers, mais des **flux** de données en provenance ou à destination de fichiers. Ces flux peuvent être de deux types :

- des flux de textes qui sont des suites de caractères terminées par un caractère de fin de ligne (`\n`) et formant ainsi des lignes ;
- des flux binaires qui sont des suites de multiplets.

21.2.1 Pourquoi utiliser des flux ?

Pourquoi recourir à des flux plutôt que de manipuler directement des fichiers nous demanderez-vous ? Pour deux raisons : les disparités entre système d'exploitation quant à la représentation des lignes et la lourdeur des opérations de lecture et d'écriture.

Disparités entre systèmes d'exploitation

Les différents systèmes d'exploitation ne représentent pas les lignes de la même manière :

- sous Mac OS (avant Mac OS X), la fin d'une ligne était indiquée par le caractère `\r` ;
- sous Windows, la fin de ligne est indiquée par la suite de caractères `\r\n` ;
- sous Unixöides (GNU/Linux, *BSD, Mac OS X, Solaris, etc.), la fin de ligne est indiquée par le caractère `\n`.

Aussi, si nous manipulons directement des fichiers, nous devrions prendre en compte ces disparités, ce qui rendrait notre code nettement plus pénible à rédiger. En passant par les fonctions de la bibliothèque standard, nous évitons ce casse-tête car celle-ci remplace le ou les caractères de fin de ligne par un `\n` (ce que nous avons pu expérimenter avec les fonctions `printf()` et `scanf()`) et inversement.

Lourdeur des opérations de lecture et d'écriture

Lire depuis un fichier ou écrire dans un fichier signifie le plus souvent accéder au disque dur. Or, rappelez-vous, il s'agit de la mémoire la plus lente d'un ordinateur. Dès lors, si pour lire une

ligne il était nécessaire de récupérer les caractères un à un depuis le disque dur, cela prendrait un temps fou.

Pour éviter ce problème, les flux de la bibliothèque standard recourent à un mécanisme appelé la **temporisation** ou **mémorisation** (*buffering* en anglais). La bibliothèque standard fournit deux types de temporisation :

- la temporisation par blocs ;
- et la temporisation par lignes.

Avec la **temporisation par blocs**, les données sont récupérées depuis le fichier et écrites dans le fichier sous forme de blocs d'une taille déterminée. L'idée est la suivante : plutôt que de lire les caractères un à un, nous allons demander un bloc de données d'une taille déterminée que nous conserverons en mémoire vive pour les accès suivants. Cette technique est également utilisée lors des opérations d'écritures : les données sont stockées en mémoire jusqu'à ce qu'elles atteignent la taille d'un bloc. Ainsi, le nombre d'accès au disque dur sera limité à la taille totale du fichier à lire (ou des données à écrire) divisée par la taille d'un bloc.

La **temporisation par lignes**, comme son nom l'indique, se contente de mémoriser une ligne. Techniquement, celle-ci est utilisée lorsque le flux est associé à un périphérique interactif comme un terminal. En effet, si nous appliquons la temporisation par blocs pour les entrées de l'utilisateur, ce dernier devrait entrer du texte jusqu'à atteindre la taille d'un bloc. Pareillement, nous devrions attendre d'avoir écrit une quantité de données égale à la taille d'un bloc pour que du texte soit affiché à l'écran. Cela serait assez gênant et peu interactif... À la place, c'est la temporisation par lignes qui est utilisée : les données sont stockées jusqu'à ce qu'un caractère de fin de ligne soit rencontré (`\n`, donc) ou jusqu'à ce qu'une taille maximale soit atteinte.

La bibliothèque standard vous permet également de ne pas temporiser les données en provenance ou à destination d'un flux.

21.2.2 `stdin`, `stdout` et `stderr`

Par défaut, trois flux *de texte* sont ouverts lors du démarrage d'un programme et sont déclarés dans l'en-tête `<stdio.h>` : `stdin`, `stdout` et `stderr`.

- `stdin` correspond à l'**entrée standard**, c'est-à-dire le flux depuis lequel vous pouvez récupérer les informations fournies par l'utilisateur.
- `stdout` correspond à la **sortie standard**, il s'agit du flux vous permettant de transmettre des informations à l'utilisateur qui seront le plus souvent affichées dans le terminal.
- `stderr` correspond à la **sortie d'erreur standard**, c'est ce flux que vous devez privilégier lorsque vous souhaitez transmettre des messages d'erreurs ou des avertissements à l'attention de l'utilisateur (nous verrons comment l'utiliser un peu plus tard dans ce chapitre). Comme pour `stdout`, ces données sont le plus souvent affichées dans le terminal.

Les flux `stdin` et `stdout` sont temporisés par lignes (sauf s'ils sont associés à des fichiers au lieu de périphériques interactifs, auxquels cas ils seront temporisés par blocs) tandis que le flux `stderr` est *au plus* temporisé par lignes (ceci afin que les informations soient transmises le plus rapidement possible à l'utilisateur).

21.3 Ouverture et fermeture d'un flux

21.4 La fonction `fopen`

```
1 FILE *fopen(char *chemin, char *mode);
```

La fonction `fopen()` permet d'ouvrir un flux. Celle-ci attend deux arguments : un **chemin d'accès** vers un fichier qui sera associé au flux et un **mode** qui détermine le type de flux (texte ou binaire) et la nature des opérations qui seront réalisées sur le fichier via le flux (lecture,

écriture ou les deux). Elle retourne un pointeur vers un flux en cas de succès et un pointeur nul en cas d'échec.

Le mode

Le mode est une chaîne de caractères composée d'une ou plusieurs lettres qui décrit le type du flux et la nature des opérations qu'il doit réaliser.

Cette chaîne commence *obligatoirement* par la seconde de ces informations. Il existe six possibilités reprises dans le tableau ci-dessous.

Mode	Type(s) d'opération(s)	Effets
<code>r</code>	Lecture	Néant
<code>r+</code>	Lecture et écriture	Néant
<code>w</code>	Écriture	Si le fichier n'existe pas, il est créé. Si le fichier existe, son contenu est effacé.
<code>w+</code>	Lecture et écriture	<i>Idem</i>
<code>a</code>	Écriture	Si le fichier n'existe pas, il est créé. Place les données à la fin du fichier
<code>a+</code>	Lecture et écriture	<i>Idem</i>

TABLE 21.1 – Les modes d'accès à un fichier

Par défaut, les flux sont des flux de textes. Pour obtenir un flux binaire, il suffit d'ajouter la lettre `b` à la fin de la chaîne décrivant le mode.

Exemple

Le code ci-dessous tente d'ouvrir un fichier nommé « texte.txt » en lecture seule dans le dossier courant. Notez que dans le cas où il n'existe pas, la fonction `fopen()` retournera un pointeur nul (seul le mode `r` permet de produire ce comportement).

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4
5  int main(void)
6  {
7      FILE *fp;
8
9      fp = fopen("texte.txt", "r");
10
11     if (fp == NULL)
12     {
13         printf("Le fichier texte.txt n'a pas pu être ouvert\n");
14         return EXIT_FAILURE;
15     }
16
17     printf("Le fichier texte.txt existe\n");
18     return 0;
19 }
```

21.4.1 La fonction fclose

```

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

La fonction `fclose()` termine l'association entre un flux et un fichier. S'il reste des données temporisées, celles-ci sont écrites. La fonction retourne zéro en cas de succès et `EOF` en cas d'erreur.



`EOF` est une constante définie dans l'en-tête `<stdio.h>` et est utilisée par les fonctions déclarées dans ce dernier pour indiquer soit l'arrivée à la fin d'un fichier (nous allons y venir) soit la survenance d'une erreur. La valeur de cette constante est *toujours* un entier *négatif*.

Nous pouvons désormais compléter l'exemple précédent comme suit.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4
5  int main(void)
6  {
7      FILE *fp;
8
9      fp = fopen("texte.txt", "r");
10
11     if (fp == NULL)
12     {
13         printf("Le fichier texte.txt n'a pas pu être ouvert\n");
14         return EXIT_FAILURE;
15     }
16
17     printf("Le fichier texte.txt existe\n");
18
19     if (fclose(fp) == EOF)
20     {
21         printf("Erreur lors de la fermeture du flux\n");
22         return EXIT_FAILURE;
23     }
24
25     return 0;
26 }
```



Veillez qu'à chaque appel à la fonction `fopen()` corresponde un appel à la fonction `fclose()`.

21.5 Écriture vers un flux de texte

21.5.1 Écrire un caractère

```

1  int putc(int ch, FILE *flux);
2  int fputc(int ch, FILE *flux);
3  int putchar(int ch);
```

Les fonctions `putc()` et `fputc()` écrivent un caractère dans un flux. Il s'agit de l'opération d'écriture la plus basique sur laquelle reposent toutes les autres fonctions d'écriture. Ces deux fonctions retournent soit le caractère écrit, soit `EOF` si une erreur est rencontrée. La fonction `putchar()`, quant à elle, est identique aux fonctions `putc()` et `fputc()` si ce n'est qu'elle écrit dans le flux `stdout`.



Techniquement, `putc()` et `fputc()` sont identiques, si ce n'est que `putc()` est en fait le plus souvent une macrofonction. Étant donné que nous n'avons pas encore vu de quoi il s'agit, préférez utiliser la fonction `fputc()` pour l'instant.

L'exemple ci-dessous écrit le caractère « C » dans le fichier « texte.txt ». Étant donné que nous utilisons le mode `w`, le fichier est soit créé s'il n'existe pas, soit vidé de son contenu s'il existe (revoyez le tableau des modes à la section précédente si vous êtes perdus).

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4
5  int main(void)
6  {
7      FILE *fp;
8
9      fp = fopen("texte.txt", "w");
10
11     if (fp == NULL)
12     {
13         printf("Le fichier texte.txt n'a pas pu être ouvert\n");
14         return EXIT_FAILURE;
15     }
16     if (fputc('C', fp) == EOF)
17     {
18         printf("Erreur lors de l'écriture d'un caractère\n");
19         return EXIT_FAILURE;
20     }
21     if (fclose(fp) == EOF)
22     {
23         printf("Erreur lors de la fermeture du flux\n");
24         return EXIT_FAILURE;
25     }
26
27     return 0;

```

21.5.2 Écrire une ligne

```

1  int fputs(char *ligne, FILE *flux);
2  int puts(char *ligne);

```

La fonction `fputs()` écrit une ligne dans le flux `flux`. La fonction retourne un nombre positif ou nul en cas de succès et `EOF` en cas d'erreurs. La fonction `puts()` est identique si ce n'est qu'elle ajoute automatiquement un caractère de fin de ligne et qu'elle écrit sur le flux `stdout`.



Maintenant que nous savons comment écrire une ligne dans un flux précis, nous allons pouvoir diriger nos messages d'erreurs vers le flux `stderr` afin que ceux-ci soient affichés le plus rapidement possible.

L'exemple suivant écrit le mot « Bonjour » suivi d'un caractère de fin de ligne au sein du fichier « texte.txt ».

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4
5  int main(void)
6  {
7      FILE *fp;
8
9      fp = fopen("texte.txt", "w");
10
11     if (fp == NULL)
12     {
13         fputs("Le fichier texte.txt n'a pas pu être ouvert\n", stderr);
14         return EXIT_FAILURE;
15     }
16     if (fputs("Bonjour\n", fp) == EOF)
17     {
18         fputs("Erreur lors de l'écriture d'une ligne\n", stderr);
19         return EXIT_FAILURE;
20     }
21     if (fclose(fp) == EOF)
22     {

```

```

23     fputs("Erreur lors de la fermeture du flux\n", stderr);
24     return EXIT_FAILURE;
25 }
26
27 return 0;
28 }

```



La norme ^a vous garanti qu'une ligne peut contenir jusqu'à 254 caractères (caractère de fin de ligne inclus). Aussi, veuillez à ne pas écrire de ligne d'une taille supérieure à cette limite.

a. Programming Language C, X3J11/88-090, § 4.9.2, Streams, al. 4

21.5.3 La fonction fprintf

```

1 int fprintf(FILE *flux, char *format, ...);

```

La fonction `fprintf()` est la même que la fonction `printf()` si ce n'est qu'il est possible de lui spécifier sur quel flux écrire (au lieu de `stdout` pour `printf()`). Elle retourne le nombre de caractères écrits ou une valeur négative en cas d'échec.

21.6 Lecture depuis un flux de texte

21.6.1 Récupérer un caractère

```

1 int getc(FILE *flux);
2 int fgetc(FILE *flux);
3 int getchar(void);

```

Les fonctions `getc()` et `fgetc()` sont les exacts miroirs des fonctions `putc()` et `fputc()` : elles récupèrent un caractère depuis le flux fourni en argument. Il s'agit de l'opération de lecture la plus basique sur laquelle reposent toutes les autres fonctions de lecture. Ces deux fonctions retournent soit le caractère lu, soit `EOF` si la fin de fichier est rencontrée *ou* si une erreur est rencontrée. La fonction `getchar()`, quant à elle, est identique à ces deux fonctions si ce n'est qu'elle récupère un caractère depuis le flux `stdin`.



Comme `putc()`, la fonction `getc()` est le plus souvent une macrofonction. Utilisez donc plutôt la fonction `fgetc()` pour le moment.

L'exemple ci-dessous lit un caractère provenant du fichier `texte.txt`.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4
5 int main(void)
6 {
7     FILE *fp;
8     int ch;
9
10    fp = fopen("texte.txt", "r");
11
12    if (fp == NULL)
13    {
14        fprintf(stderr, "Le fichier texte.txt n'a pas pu être ouvert\n");
15        return EXIT_FAILURE;
16    }

```

```

17     if ((ch = fgetc(fp)) != EOF)
18         printf("%c\n", ch);
19     if (fclose(fp) == EOF)
20     {
21         fprintf(stderr, "Erreur lors de la fermeture du flux\n");
22         return EXIT_FAILURE;
23     }
24
25     return 0;
26 }

```



Notez que nous utilisons une affectation comme premier opérande de l'opérateur `!=`. Une affectation étant une expression en C, ce genre d'écriture est tout à fait valide. Vous en rencontrerez fréquemment comme expression de contrôle de boucles.

21.6.2 Récupérer une ligne

```

1 char *fgets(char *tampon, int taille, FILE *flux);

```

La fonction `fgets()` lit une ligne depuis le flux `flux` et la stocke dans le tableau `tampon`. Cette dernière lit au plus un nombre de caractères égal à `taille` diminué de un afin de laisser la place pour le caractère nul, qui est automatiquement ajouté. Dans le cas où elle rencontre un caractère de fin de ligne : *celui-ci est conservé au sein du tableau*, un caractère nul est ajouté et la lecture s'arrête.

La fonction retourne l'adresse du tableau `tampon` en cas de succès et un pointeur nul si la fin du fichier est atteinte *ou si une erreur est survenue*.

L'exemple ci-dessous réalise donc la même opération que le code précédent, mais en utilisant la fonction `fgets()`.



Étant donné que la norme nous garanti qu'une ligne peut contenir jusqu'à 254 caractères (caractère de fin de ligne inclus), nous utilisons un tableau de 255 caractères pour les contenir (puisque'il est nécessaire de prévoir un espace pour le caractère nul).

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      char buf[255];
6      FILE *fp;
7
8      fp = fopen("texte.txt", "r");
9
10     if (fp == NULL)
11     {
12         fprintf(stderr, "Le fichier texte.txt n'a pas pu être ouvert\n");
13         return EXIT_FAILURE;
14     }
15     if (fgets(buf, sizeof buf, fp) != NULL)
16         printf("%s\n", buf);
17     if (fclose(fp) == EOF)
18     {
19         fprintf(stderr, "Erreur lors de la fermeture du flux\n");
20         return EXIT_FAILURE;
21     }
22
23     return 0;
24 }

```

Toutefois, il y a un petit problème : la fonction `fgets()` conserve le caractère de fin de ligne qu'elle rencontre. Dès lors, nous affichons deux retours à la ligne : celui contenu dans la chaîne `buf` et celui affiché par `printf()`. Aussi, il serait préférable d'en supprimer un, de préférence celui de la chaîne de caractères. Pour ce faire, nous pouvons faire appel à une petite fonction (que nous appellerons `chomp()` en référence à la fonction éponyme du langage Perl) qui se chargera de remplacer le caractère de fin de ligne par un caractère nul.

```
1 void chomp(char *s)
2 {
3     while (*s != '\n' && *s != '\0')
4         ++s;
5
6     if (*s == '\n')
7         *s = '\0';
8 }
```

21.6.3 La fonction fscanf

```
1 int fscanf(FILE *flux, char *format, ...);
```

La fonction `fscanf()` est identique à la fonction `scanf()` si ce n'est qu'elle récupère les données depuis le flux fourni en argument (au lieu de `stdin` pour `scanf()`).



Le flux `stdin` étant le plus souvent mémorisé par lignes, ceci vous explique pourquoi nous lisons les caractères restant après un appel à `scanf()` jusqu'à rencontrer un caractère de fin de ligne : pour vider le tampon du flux `stdin`.

La fonction `fscanf()` retourne le nombre de conversions réussies (voire zéro, si aucune n'est demandée ou n'a pu être réalisée) ou `EOF` si une erreur survient *avant* qu'une conversion n'ait eu lieu.

21.7 Écriture vers un flux binaire

21.7.1 Écrire un multiplet

```
1 int putc(int ch, FILE *flux);
2 int fputc(int ch, FILE *flux);
```

Comme pour les flux de texte, il vous est possible de recourir aux fonctions `putc()` et `fputc()`. Dans le cas d'un flux binaire, ces fonctions écrivent un multiplet (sous la forme d'un `int` converti en `unsigned char`) dans le flux spécifié.

21.7.2 Écrire une suite de multiplats

```
1 size_t fwrite(void *ptr, size_t taille, size_t nombre, FILE *flux);
```

La fonction `fwrite()` écrit le tableau référencé par `ptr` composé de `nombre` éléments de `taille` multiplats dans le flux `flux`. Elle retourne une valeur égale à `nombre` en cas de succès et une valeur inférieure en cas d'échec.

L'exemple suivant écrit le contenu du tableau `tab` dans le fichier `binaire.bin`. Dans le cas où un `int` fait 4 octets, 20 octets seront donc écrits.

```
1 #include <stddef.h>
2 #include <stdio.h>
3 #include <stdlib.h>
```

```

4
5
6 int main(void)
7 {
8     int tab[5] = { 1, 2, 3, 4, 5 };
9     const size_t n = sizeof tab / sizeof tab[0];
10    FILE *fp;
11
12    fp = fopen("binaire.bin", "wb");
13
14    if (fp == NULL)
15    {
16        fprintf(stderr, "Le fichier binaire.bin n'a pas pu être ouvert\n");
17        return EXIT_FAILURE;
18    }
19    if (fwrite(&tab, sizeof tab[0], n, fp) != n)
20    {
21        fprintf(stderr, "Erreur lors de l'écriture du tableau\n");
22        return EXIT_FAILURE;
23    }
24    if (fclose(fp) != EOF)
25    {
26        fprintf(stderr, "Erreur lors de la fermeture du flux\n");
27        return EXIT_FAILURE;
28    }
29
30    return 0;
31 }

```

21.8 Lecture depuis un flux binaire

21.8.1 Lire un multiplet

```

1 int getc(FILE *flux);
2 int fgetc(FILE *flux);

```

Lors de la lecture depuis un flux binaire, les fonctions `fgetc()` et `getc()` permettent de récupérer un multiplet (sous la forme d'un `unsigned char` converti en `int`) depuis un flux.

21.8.2 Lire une suite de multiplets

```

1 size_t fread(void *ptr, size_t taille, size_t nombre, FILE *flux);

```

La fonction `fread()` est l'inverse de la fonction `fwrite()` : elle lit `nombre` éléments de `taille` multiplets depuis le flux `flux` et les stocke dans l'objet référencé par `ptr`. Elle retourne une valeur égale à `nombre` en cas de succès ou une valeur inférieure en cas d'échec.

Dans le cas où nous disposons du fichier `binaire.bin` produit par l'exemple de la section précédente, nous pouvons reconstituer le tableau `tab`.

```

1 #include <stddef.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4
5
6 int main(void)
7 {
8     int tab[5] = { 1, 2, 3, 4, 5 };
9     const size_t n = sizeof tab / sizeof tab[0];
10    FILE *fp;
11    unsigned i;
12
13    fp = fopen("binaire.bin", "rb");
14
15    if (fp == NULL)

```

```
16 {
17     fprintf(stderr, "Le fichier binaire.bin n'a pas pu être ouvert\n");
18     return EXIT_FAILURE;
19 }
20 if (fread(&tab, sizeof tab[0], n, fp) != n)
21 {
22     fprintf(stderr, "Erreur lors de la lecture du tableau\n");
23     return EXIT_FAILURE;
24 }
25 if (fclose(fp) != EOF)
26 {
27     fprintf(stderr, "Erreur lors de la fermeture du flux\n");
28     return EXIT_FAILURE;
29 }
30
31 for (i = 0; i < n; ++i)
32     printf("tab[%u] = %d\n", i, tab[i]);
33
34 return 0;
35 }
```

Dans le chapitre suivant, nous continuerons notre découverte des fichiers en attaquant quelques notions plus avancées : la gestion d'erreur et de fin de fichier, le déplacement au sein d'un flux, la temporisation et les subtilités liées aux flux ouverts en lecture et écriture.

Dans ce chapitre, nous allons poursuivre notre lancée et vous présenter plusieurs points un peu plus avancés en rapport avec les fichiers et les flux.

22.1 Détection d'erreurs et fin de fichier

Lors de la présentation des fonctions `fgetc()`, `getc()` et `fgets()` vous aurez peut-être remarqué que ces fonctions utilisent un même retour pour indiqué soit la rencontre de la fin du fichier, soit la survenance d'une erreur. Du coup, comment faire pour distinguer l'un et l'autre cas ?

Il existe deux fonctions pour clarifier une telle situation : `feof()` et `ferror()`.

22.1.1 La fonction feof

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

La fonction `feof()` retourne une valeur non nulle dans le cas où la fin du fichier associé au flux spécifié est atteinte.

22.1.2 La fonction ferror

```
1 int ferror(FILE *flux);
```

La fonction `ferror()` retourne une valeur non nulle dans le cas où une erreur s'est produite lors d'une opération sur le flux visé.

22.1.3 Exemple

L'exemple ci-dessous utilise ces deux fonctions pour déterminer le type de problème rencontré par la fonction `fgets()`.

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     char buf[255];
6     FILE *fp;
7
8     fp = fopen("texte.txt", "r");
9
10    if (fp == NULL)
11    {
12        fprintf(stderr, "Le fichier texte.txt n'a pas pu être ouvert\n");
```

```

13     return EXIT_FAILURE;
14 }
15 if (fgets(buf, sizeof buf, fp) != NULL)
16     printf("%s\n", buf);
17 else if (ferror(fp))
18 {
19     fprintf(stderr, "Erreur lors de la lecture\n");
20     return EXIT_FAILURE;
21 }
22 else
23 {
24     fprintf(stderr, "Fin de fichier rencontrée\n");
25     return EXIT_FAILURE;
26 }
27 if (fclose(fp) == EOF)
28 {
29     fprintf(stderr, "Erreur lors de la fermeture du flux\n");
30     return EXIT_FAILURE;
31 }
32
33 return 0;
34 }

```

De manière imagée, un fichier peut être vu comme une longue bande magnétique depuis laquelle des données sont lues ou sur laquelle des informations sont écrites. Ainsi, au fur et à mesure que les données sont lues ou écrites, nous avançons le long de cette bande.

Toutefois, il est parfois nécessaire de se rendre directement à un point précis de cette bande (par exemple à la fin) afin d'éviter des lectures inutiles pour parvenir à un endroit souhaité. À cet effet, la bibliothèque standard fournit plusieurs fonctions.

22.1.4 La fonction `ftell`

```

1 long ftell(FILE *flux);

```

La fonction `ftell()` retourne la position actuelle au sein du fichier sous forme d'un `long`. Elle retourne un nombre négatif en cas d'erreur. Dans le cas des flux *binaires*, cette valeur correspond au nombre de multipliants séparant la position actuelle du début du fichier.



Lors de l'ouverture d'un flux, la position courante correspond au début du fichier, *sauf* pour les modes `a` et `a+` pour lesquels la position initiale est soit le début, soit la fin du fichier.

22.1.5 La fonction `fseek`

```

1 int fseek(FILE *flux, long distance, int repere);

```

La fonction `fseek()` permet d'effectuer un déplacement d'une distance fournie en argument depuis un repère donné. Elle retourne zéro en cas de succès et une autre valeur en cas d'échec. Il existe trois repères possibles :

- `SEEK_SET` qui correspond au début du fichier ;
- `SEEK_CUR` qui correspond à la position courante ;
- `SEEK_END` qui correspond à la fin du fichier.

Cette fonction s'utilise différemment suivant qu'elle opère sur un flux de texte ou sur un flux binaire.

Les flux de texte

Dans le cas d'un flux de texte, il y a deux possibilités :

- la distance fournie est nulle ;
- la distance est une valeur fournie par un précédent appel à la fonction `ftell()` et le repère est `SEEK_SET`.

Les flux binaires

Dans le cas d'un flux binaire, seuls les repères `SEEK_SET` et `SEEK_CUR` peuvent être utilisés. La distance correspond à un nombre de multiplés à passer depuis le repère fourni.



Dans la cas des modes `a` et `a+`, un déplacement a lieu à la fin du fichier *avant chaque opération d'écriture* et ce, *qu'il y ait eu déplacement auparavant ou non*.

22.1.6 La fonction rewind

```
1 int fseek(FILE *flux, long distance, int repere);
```

La fonction `rewind()` vous ramène au début du fichier (autrement dit, elle rembobine la bande).



Si vous utilisez un flux ouvert en lecture *et* écriture vous *devez* appeler une fonction de déplacement entre deux opérations de nature différentes ou utiliser la fonction `fflush()` (présentée dans l'extrait suivant) entre une opération d'écriture et de lecture. Si vous ne souhaitez pas vous déplacer, vous pouvez utiliser l'appel `fseek(flux, 0, SEEK_CUR)` afin de respecter cette condition sans réellement effectuer un déplacement.

22.2 La temporisation

Dans le chapitre précédent, nous vous avons précisé que les flux étaient le plus souvent temporisés afin d'optimiser les opérations de lecture et d'écriture sous-jacentes. Dans cette section, nous allons nous pencher un peu plus sur cette notion.

22.2.1 Introduction

Nous vous avons dit auparavant que deux types de temporisations existaient : la temporisation par lignes et celle par blocs. Une des conséquences logiques de cette temporisation est que les fonctions de lecture/écriture récupèrent les données et les inscrivent dans ces tampons. Ceci peut paraître évident, mais cela peut avoir des conséquences parfois surprenantes si ce fait est oublié ou inconnu.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4
5 int main(void)
6 {
7     char nom[64];
8     char prenom[64];
9
10    printf("Quel est votre nom ? ");
11
12    if (scanf("%63s", nom) != 1)
13    {
14        fprintf(stderr, "Erreur lors de la saisie\n");
```

```

15     return EXIT_FAILURE;
16 }
17
18 printf("Quel est votre prénom ? ");
19
20 if (scanf("%63s", prenom) != 1)
21 {
22     fprintf(stderr, "Erreur lors de la saisie\n");
23     return EXIT_FAILURE;
24 }
25
26 printf("Votre nom est %s\n", nom);
27 printf("Votre prénom est %s\n", prenom);
28 return 0;
29 }

```

```

1  Quel est votre nom ? Charles Henri
2  Quel est votre prénom ? Votre nom est Charles
3  Votre prénom est Henri

```

Comme vous le voyez, le programme ci-dessus réalise deux saisies, mais si l'utilisateur entre par exemple « Charles Henri », il n'aura l'occasion d'entrer des données qu'une seule fois. Ceci est dû au fait que l'indicateur `s` récupère une suite de caractères exempte d'espaces (ce qui fait qu'il s'arrête à « Charles ») et que la suite « Henri » *demeure dans le tampon du flux `stdin`*. Ainsi, lors de la deuxième saisie, il n'est pas nécessaire de récupérer de nouvelles données depuis le terminal puisqu'il y en a déjà en attente dans le tampon, d'où le résultat obtenu.

Le même problème peut se poser si par exemple les données fournies ont une taille supérieure par rapport à l'objet qui doit les accueillir.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4
5  int main(void)
6  {
7      char chaine1[16];
8      char chaine2[16];
9
10     printf("Un morceau : ");
11
12     if (fgets(chaine1, sizeof chaine1, stdin) == NULL)
13     {
14         printf("Erreur lors de la saisie\n");
15         return EXIT_FAILURE;
16     }
17
18     printf("Un autre morceau : ");
19
20     if (fgets(chaine2, sizeof chaine2, stdin) == NULL)
21     {
22         printf("Erreur lors de la saisie\n");
23         return EXIT_FAILURE;
24     }
25
26     printf("%s ; %s\n", chaine1, chaine2);
27     return 0;

```

```

1  Un morceau : Une chaîne de caractères, vraiment, mais alors vraiment trop longue
2  Un autre morceau : Une chaîne de ; caractères, vr

```

Ici, la chaîne entrée est trop importante pour être contenue dans `chaine1`, les données non lues sont alors conservées dans le tampon du flux `stdin` et lues lors de la seconde saisie (qui ne lit par l'entière non plus).

22.2.2 Intérargir avec la temporisation

Vider un tampon

Si la temporisation nous évite des coûts en terme d'opérations de lecture/écriture, il nous est parfois nécessaire de passer outre cette mécanique pour vider manuellement le tampon d'un flux.

Opération de lecture

Si le tampon contient des données qui proviennent d'une opération de lecture, celles-ci peuvent être abandonnées soit en appelant une fonction de positionnement soit en lisant les données, tout simplement. Il y a toutefois un bémol avec la première solution : les fonctions de positionnement ne fonctionnent pas dans le cas où le flux ciblé est lié à un périphérique « interactif », c'est-à-dire le plus souvent un terminal.

Autrement dit, pour que l'exemple précédent recoure bien à deux saisies, il nous est nécessaire de vérifier que la fonction `fgetc()` a bien lu un caractère `\n` (qui signifie que la fin de ligne est atteinte et donc celle du tampon s'il s'agit du flux `stdin`). Si ce n'est pas le cas, alors il nous faut lire les caractères restants jusqu'au `\n` final (ou la fin du fichier).

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5
6  int vider_tampon(FILE *fp)
7  {
8      int c;
9
10     do
11         c = fgetc(fp);
12     while (c != '\n' && c != EOF);
13
14     return ferror(fp) ? 0 : 1;
15 }
16
17
18 int main(void)
19 {
20     char chaine1[16];
21     char chaine2[16];
22
23     printf("Un morceau : ");
24
25     if (fgetc(chaine1, sizeof chaine1, stdin) == NULL)
26     {
27         printf("Erreur lors de la saisie\n");
28         return EXIT_FAILURE;
29     }
30     if (strchr(chaine1, '\n') == NULL)
31         if (!vider_tampon(stdin))
32         {
33             fprintf(stderr, "Erreur lors de la vidange du tampon.\n");
34             return EXIT_FAILURE;
35         }
36
37     printf("Un autre morceau : ");
38
39     if (fgetc(chaine2, sizeof chaine2, stdin) == NULL)
40     {
41         printf("Erreur lors de la saisie\n");
42         return EXIT_FAILURE;
43     }
44     if (strchr(chaine2, '\n') == NULL)
45         if (!vider_tampon(stdin))
46         {
47             fprintf(stderr, "Erreur lors de la vidange du tampon.\n");

```

```

48         return EXIT_FAILURE;
49     }
50
51     printf("%s ; %s\n", chaine1, chaine2);
52     return 0;
53 }

```

```

1  Un morceau : Une chaîne de caractères vraiment, mais alors vraiment longue
2  Un autre morceau : Une autre chaîne de caractères
3  Une chaîne de ; Une autre chaî

```

Opération d'écriture

Si, en revanche, le tampon comprend des données en attente d'écriture, il est possible de forcer celle-ci soit à l'aide d'une fonction de positionnement, soit à l'aide de la fonction `fflush()`.

```

1  int fflush(FILE *flux);

```

Celle-ci vide le tampon du flux spécifié et retourne zéro en cas de succès ou `EOF` en cas d'erreur.



Notez bien que cette fonction ne peut être employée que pour vider un tampon contenant des données en attente d'écriture.

Modifier un tampon

Techniquement, il ne vous est pas possible de modifier directement le contenu d'un tampon, ceci est réalisé par les fonctions de la bibliothèque standard au gré de vos opérations de lecture et/ou d'écriture. Il y a toutefois une exception à cette règle : la fonction `ungetc()`.

```

1  int ungetc(int ch, FILE *flux);

```

Cette fonction est un peu particulière : elle place le caractère `ch` dans le tampon du flux `flux`. Ce caractère pourra être lu lors d'un appel ultérieur à une fonction de lecture. Elle retourne le caractère ajouté en cas de succès et `EOF` en cas d'échec.

Cette fonction est très utile dans le cas où les actions d'un programme dépendent du contenu d'un flux. Imaginez par exemple que votre programme doit déterminer si l'entrée standard contient une suite de caractères ou un nombre et doit ensuite afficher celui-ci. Vous pourriez utiliser `getchar()` pour récupérer le premier caractère et déterminer s'il s'agit d'un chiffre. Toutefois, le premier caractère du flux est alors lu et cela complique votre tâche pour la suite... La fonction `ungetc()` vous permet de résoudre ce problème en remplaçant ce caractère dans le tampon du flux `stdin`.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  static void afficher_nombre(void);
5  static void afficher_chaine(void);
6
7
8  static void afficher_nombre(void)
9  {
10     double f;
11
12     if (scanf("%lf", &f) == 1)
13         printf("Vous avez entré le nombre : %f\n", f);
14     else

```

```

15     fprintf(stderr, "Erreur lors de la saisie\n");
16 }
17
18
19 static void afficher_chaine(void)
20 {
21     char chaine[255];
22
23     if (scanf("%254s", chaine) == 1)
24         printf("Vous avez entré la chaine : %s\n", chaine);
25     else
26         fprintf(stderr, "Erreur lors de la saisie\n");
27 }
28
29
30 int main(void)
31 {
32     int ch;
33
34     ch = getchar();
35
36     if (ungetc(ch, stdin) == EOF)
37     {
38         fprintf(stderr, "Impossible de remplacer un caractère\n");
39         return EXIT_FAILURE;
40     }
41
42     switch (ch)
43     {
44     case '0':
45     case '1':
46     case '2':
47     case '3':
48     case '4':
49     case '5':
50     case '6':
51     case '7':
52     case '8':
53     case '9':
54         afficher_nombre();
55         break;
56
57     default:
58         afficher_chaine();
59         break;
60     }
61
62     return 0;
63 }

```



La fonction `ungetc()` ne vous permet de remplacer qu'*un seul caractère* avant une opération de lecture.

22.3 Flux ouverts en lecture et écriture

Pour clore ce chapitre, un petit mot sur le cas des flux ouverts en lecture *et* en écriture (soit à l'aide des modes `r+`, `w+`, `a+` et leurs équivalents binaires).

Si vous utilisez un tel flux, vous *devez* appeler une fonction de déplacement entre deux opérations de nature différentes ou utiliser la fonction `fflush()` entre une opération d'écriture et de lecture. Si vous ne souhaitez pas vous déplacer, vous pouvez utiliser l'appel `fseek(flux, 0L, SEEK_CUR)` afin de respecter cette condition sans réellement effectuer un déplacement.



Vous l'aurez sans doute compris : cette règle impose en fait de vider le tampon du flux entre deux opérations de natures différentes.


```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4
5  int main(void)
6  {
7      char chaine[255];
8      FILE *fp;
9
10     fp = fopen("texte.txt", "w+");
11
12     if (fp == NULL)
13     {
14         fprintf(stderr, "Le fichier texte.txt n'a pas pu être ouvert.\n");
15         return EXIT_FAILURE;
16     }
17     if (fputs("Une phrase.\n", fp) == EOF)
18     {
19         fprintf(stderr, "Erreur lors de l'écriture d'une ligne.\n");
20         return EXIT_FAILURE;
21     }
22
23     /* Retour au début : la condition est donc remplie. */
24
25     if (fseek(fp, 0L, SEEK_SET) != 0)
26     {
27         fprintf(stderr, "Impossible de revenir au début du fichier.\n");
28         return EXIT_FAILURE;
29     }
30     if (fgets(chaine, sizeof chaine, fp) == NULL)
31     {
32         fprintf(stderr, "Erreur lors de la lecture.\n");
33         return EXIT_FAILURE;
34     }
35
36     printf("%s\n", chaine);
37
38     if (fclose(fp) == EOF)
39     {
40         fputs("Erreur lors de la fermeture du flux\n", stderr);
41         return EXIT_FAILURE;
42     }
43
44     return 0;

```

```

1  Une phrase.

```

Dans le chapitre suivant, nous ferons une petite pause dans notre apprentissage afin de réaliser un jeu de *Puissance 4*.

Le **préprocesseur** est un programme qui réalise des traitements sur le code source *avant* que ce dernier ne soit réellement compilé. Globalement, il a trois grands rôles :

- réaliser des **inclusions** (la fameuse directive `#include`) ;
- définir des **macros** qui sont des substituts à des morceaux de code. Après le passage du préprocesseur, tous les appels à ces macros seront remplacés par le code associé ;
- permettre la **compilation conditionnelle**, c'est-à-dire de moduler le contenu d'un fichier source suivant certaines conditions.

23.1 Les inclusions

Nous avons vu dès le début du cours comment inclure des fichiers d'en-tête avec la directive `#include`, sans toutefois véritablement expliquer son rôle. Son but est très simple : inclure le contenu d'un fichier dans un autre. Ainsi, si nous nous retrouvons par exemple avec deux fichiers comme ceux-ci avant la compilation.

```
1  /* Fichier d'en-tête fichier.h */
2
3  #ifndef FICHIER_H
4  #define FICHIER_H
5
6  extern int glob_var;
7
8  extern void f1(int);
9  extern long f2(double, char);
10
11 #endif
```

```
1  /* Fichier source fichier.c */
2
3  #include "fichier.h"
4
5  void f1(int arg)
6  {
7      /* du code */
8  }
9
10 long f2(double arg, char c)
11 {
12     /* du code */
13 }
```

Après le passage du préprocesseur et avant la compilation à proprement parler, le code obtenu sera le suivant :

```

1  /* Fichier source fichier.c */
2
3  extern int glob_var;
4
5  extern void f1(int arg);
6  extern long f2(double arg, char c);
7
8  void f1(int arg)
9  {
10     /* du code */
11 }
12
13 long f2(double arg, char c)
14 {
15     /* du code */
16 }

```

Nous pouvons voir que les déclarations contenues dans le fichier « fichier.h » ont été incluses dans le fichier « fichier.c » et que toutes les directives du préprocesseur (les lignes commençant par le symbole `#`) ont disparu.



Vous pouvez utiliser l'option `-E` lors de la compilation pour requérir uniquement l'utilisation du préprocesseur. Ainsi, vous pouvez voir ce que donne votre code *après* son passage.

23.2 Les macroconstantes

Comme nous l'avons dit dans l'introduction, le préprocesseur permet la définition de macros, c'est-à-dire de substituts à des morceaux de code. Une macro est constituée des éléments suivants.

- La directive `#define`.
- Le nom de la macro qui, par convention, est souvent écrit en majuscules. Notez que vous pouvez choisir le nom que vous voulez, à condition de respecter les mêmes règles que pour les noms de variable ou de fonction.
- Une liste *optionnelle* de paramètres.
- La définition, c'est-à-dire le code par lequel la macro sera remplacée.

Dans le cas particulier où la macro n'a pas de paramètre, on parle de **macroconstante** (ou constante de préprocesseur). Leur substitution donne toujours le même résultat (d'où l'adjectif « constante »).

23.2.1 Substitutions de constantes

Par exemple, pour définir une macroconstante `TAILLE` avec pour valeur `100`, nous utiliserons le code suivant.

```

1  #define TAILLE 100

```

L'exemple qui suit utilise cette macroconstante afin de ne pas multiplier l'usage de constantes entières. À chaque fois que la macroconstante `TAILLE` est utilisée, le préprocesseur remplacera celle-ci par sa définition, à savoir `100`.

```

1  #include <stdio.h>
2  #define TAILLE 100
3
4  int main(void)
5  {
6      int variable = 5;

```

```

7
8  /* On multiplie par TAILLE */
9  variable *= TAILLE;
10 printf("Variable vaut : %d\n", variable);
11
12 /* On additionne TAILLE */
13 variable += TAILLE;
14 printf("Variable vaut : %d\n", variable);
15 return 0;
16 }

```


Ce code sera remplacé, après le passage du préprocesseur, par celui ci-dessous.

```

1  int main(void)
2  {
3      int variable = 5;
4
5      variable *= 100;
6      printf("Variable vaut : %d\n", variable);
7      variable += 100;
8      printf("Variable vaut : %d\n", variable);
9      return 0;
10 }

```

Nous n'avons pas inclus le contenu de `<stdio.h>` car celui-ci est trop long et trop compliqué pour le moment. Néanmoins, l'exemple permet d'illustrer le principe des macros et surtout de leur avantage : il suffit de changer la définition de la macroconstante `TAILLE` pour que le reste du code s'adapte.

 D'accord, mais je peux aussi très bien utiliser une variable constante, non ?

Dans certains cas (comme dans l'exemple ci-dessus), utiliser une variable constante donnera le même résultat. Toutefois, une variable nécessitera le plus souvent de réserver de la mémoire et, si celle-ci doit être partagée par différents fichiers, de jouer avec les déclarations et les définitions.

De plus, les macroconstantes peuvent être employées pour définir la longueur d'un tableau, alors que ceci est impossible avec une variable, même constante.

Notez qu'il vous est possible de définir une macroconstante lors de la compilation à l'aide de l'option `-D`.



```
zcc -DTAILLE=100
```

Ceci recient à définir une macroconstante `TAILLE` au début de chaque fichier qui sera substituée par `100`.

23.2.2 Substitutions d'instructions

Si les macroconstantes sont souvent utilisées en vue de substituer des constantes, il est toutefois aussi possible que leur définition comprenne des suites d'instructions. L'exemple ci-dessous définit une macroconstante `BONJOUR` qui sera remplacée par `puts("Bonjour !")`.

```

1  #include <stdio.h>
2
3  #define BONJOUR puts("Bonjour !")
4
5  int main(void)
6  {
7      BONJOUR;
8      return 0;
9  }

```

Notez que nous n'avons pas placé de point-virgule dans la définitions de la macroconstante, tout d'abord afin de pouvoir en placer un lors de son utilisation, ce qui est plus naturel et, d'autre part, pour éviter des doublons qui peuvent s'avérer facheux.

En effet, si nous ajoutons un point virgule à la fin de la définition, il ne faut pas en rajouter un lors de l'appel, sous peine de risquer des erreurs de syntaxe.

```
1  #include <stdio.h>
2
3  #define BONJOUR puts("Bonjour !");
4  #define AUREVOIR puts("Au revoir !");
5
6  int main(void)
7  {
8      if (1)
9          BONJOUR; /* erreur ! */
10     else
11         AUREVOIR;
12
13     return 0;
14 }
```

Le code donnant en réalité ceci.

```
1  int main(void)
2  {
3      if (1)
4          puts("Bonjour !");
5      else
6          puts("Au revoir !");
7
8      return 0;
9  }
```

Ce qui est incorrect. Pour faire simple : le corps d'une condition ne peut comprendre qu'une instruction, or `puts("Bonjour !");` en est une, le point-virgule seul (`;`) en est une autre. Dès lors, il y a une instruction entre le `if` et le `else`, ce qui n'est pas permis. L'exemple ci-dessous pose le même problème en étant un peu plus limpide.


```
1  #include <stdio.h>
2
3
4  int main(void)
5  {
6      if (1)
7          puts("Bonjour !");
8
9      puts("Je suis une instruction mal placee !");
10
11     else
12         puts("Au revoir !");
13
14     return 0;
15 }
```

23.2.3 Macros dans d'autres macros

La définition d'une macro peut parfaitement comprendre une ou plusieurs autres macros (qui, à leur tour, peuvent également comprendre des macros dans leur définition et ainsi de suite). L'exemple suivant illustre ceci en définissant une macroconstante `NB_PIXELS` correspondant à la multiplication entre les deux macroconstantes `LONGUEUR` et `HAUTEUR`.

```
1  #define LONGUEUR 1024
2  #define HAUTEUR 768
3  #define NB_PIXELS (LONGUEUR * HAUTEUR)
```

La règle suivante permet d'éviter de répéter les opérations de remplacement « à l'infini » :

 Une macro n'est pas substituée si elle est utilisée dans sa propre définition.

Ainsi, dans le code ci-dessous, `LONGUEUR` donnera `LONGUEUR * 2 / 2` et `HAUTEUR` sera substituée par `HAUTEUR / 2 * 2`. En effet, lors de la substitution de la macroconstante `LONGUEUR`, la macroconstante `HAUTEUR` est remplacée par sa définition : `LONGUEUR * 2`. Or, comme nous sommes en train de substituer la macroconstante `LONGUEUR`, `LONGUEUR` sera laissé tel quel. Le même raisonnement peut être appliqué pour la macroconstante `HAUTEUR`.

```
1 #define LONGUEUR (HAUTEUR / 2)
2 #define HAUTEUR (LONGUEUR * 2)
```

23.2.4 Définition sur plusieurs lignes

La définition d'une macro doit normalement tenir sur une seule ligne. C'est-à-dire que dès que la fin de ligne est atteinte, la définition est considérée comme terminée. Ainsi, dans le code suivant, la définition de la macroconstante `BONJOUR_AUREVOIR` ne comprend en fait que `puts("Bonjour");`.

```
1 #define BONJOUR_AUREVOIR puts("Bonjour");
2 puts("Au revoir")
```

Pour que cela fonctionne, nous sommes donc contraints de la rédiger comme ceci.


```
1 #define BONJOUR_AUREVOIR puts("Bonjour"); puts("Au revoir")
```

Ce qui est assez peu lisible et peu pratique si notre définition doit comporter une multitude d'instructions ou, pire, des blocs d'instructions... Heureusement, il est possible d'indiquer au préprocesseur que plusieurs lignes doivent être fusionnées. Cela se fait à l'aide du symbole `\`, que nous plaçons en fin de ligne.

```
1 #define BONJOUR_AUREVOIR puts("Bonjour");\
2 puts("Au revoir")
```

Lorsque le préprocesseur rencontre une barre oblique inverse (`\`) suivie d'une fin de ligne, celles-ci sont supprimées et la ligne qui suit est fusionnée avec la ligne courante.

Notez bien qu'*aucun espace* n'est ajouté durant l'opération !



```
1 #define BONJOUR_AUREVOIR\
2 puts("Bonjour");\
3 puts("Au revoir")
```

Le code ci-dessus est donc incorrect car il donne en vérité ceci.

```
1 #define BONJOUR_AUREVOIRputs("Bonjour");puts("Au revoir")
```



L'utilisation de la fusion ne se limite pas aux définitions de macros, il est possible de l'employer n'importe où dans le code source. Bien que cela ne soit pas obligatoire (une instruction pouvant être répartie sur plusieurs lignes), il est possible d'y recourir pour couper une ligne un peu trop longue tout en indiquant cette césure de manière claire.

```
1 printf("Une phrase composée de plusieurs résultats à présenter : %d, %f, %f, %d, %d\n", \
```



```
a, b, c, d, e, f);
```

Si vous souhaitez inclure un bloc d'instructions au sein d'une définition, ne perdez pas de vue que celui-ci constitue une instruction et donc que vous retombez sur le problème exposé plus haut.

```

1  #include <stdio.h>
2
3  #define BONJOUR\
4      {\
5          puts("Bonjour !");\
6      }
7  #define AUREVOIR\
8      {\
9          puts("Au revoir !");\
10     }
11
12 int main(void)
13 {
14     if (1)
15         BONJOUR; /* erreur ! */
16     else
17         AUREVOIR;
18
19     return 0;
20 }
```

Une solution fréquente à ce problème consiste à recourir à une boucle `do {} while` avec une condition nulle (de sorte qu'elle ne soit exécutée qu'une seule fois), celle-ci ne constituant qu'une seule instruction *avec* le point-virgule final.

```

1  #include <stdio.h>
2
3  #define BONJOUR\
4      do {\
5          puts("Bonjour !");\
6      } while(0)
7  #define AUREVOIR\
8      do {\
9          puts("Au revoir !");\
10     } while(0)
11
12 int main(void)
13 {
14     if (1)
15         BONJOUR; /* ok */
16     else
17         AUREVOIR;
18
19     return 0;
20 }
```

Définition nulle

Sachez que le corps d'une macro peut parfaitement être vide.

```
1  #define MACRO
```

Dans un tel cas, la macro ne sera tout simplement pas substituée par quoi que ce soit. Bien que cela puisse paraître étrange, cette technique est souvent utilisée, notamment pour la compilation conditionnelle que nous verrons bientôt.

23.2.5 Annuler une définition

Enfin, une définition peut être annulée à l'aide de la directive `#undef` en lui spécifiant le nom de la macro qui doit être détruite.

```
1 #define TAILLE 100
2 #undef TAILLE
3
4 /* TAILLE n'est à présent plus utilisable. */
```

23.3 Les macrofonctions

Pour l'instant, nous n'avons manipulé que des macroconstantes, c'est-à-dire des macros n'employant pas de paramètres. Comme vous vous en doutez, une **macrofonction** est une macro qui accepte des paramètres et les emploie dans sa définition.

```
1
```

Pour ce faire, le nom de la macro est suivi de parenthèses comprenant le nom des paramètres séparés par une virgule. Chacun d'eux peut ensuite être utilisé dans la définition de la macrofonction et sera remplacé par la suite fournie en argument lors de l'appel à la macrofonction.



Notez bien que nous n'avons pas parlé de « valeur » pour les arguments. En effet, n'importe quelle suite de symboles peut-être passée en argument d'une macrofonction, y compris du code. C'est d'ailleurs ce qui fait la puissance du préprocesseur.

Illustrons ce nouveau concept avec un exemple : nous allons écrire deux macros : `EUR` qui convertira une somme en euro en francs (français) et `FRF` qui fera l'inverse. Pour rappel, un euro équivaut à 6,55957 francs français.

```
1 #include <stdio.h>
2
3 #define EUR(x) ((x) / 6.55957)
4 #define FRF(x) ((x) * 6.55957)
5
6 int main(void)
7 {
8     printf("Dix francs français valent %f euros.\n", EUR(10));
9     printf("Dix euros valent %f francs français.\n", FRF(10));
10     return 0;
11 }
```

```
1 Dix francs français valent 1.524490 euros.
2 Dix euros valent 65.595700 francs français.
```

Appliquons encore ce concept avec un deuxième exercice : essayez de créer la macro `MIN` qui renvoie le minimum entre deux nombres.

```
1 #include <stdio.h>
2
3 #define MIN(a, b) ((a) < (b) ? (a) : (b))
4
5 int main(void)
6 {
7     printf("Le minimum entre 16 et 32 est %d.\n", MIN(16, 32));
8     printf("Le minimum entre 2+9+7 et 3*8 est %d.\n", MIN(2+9+7, 3*8));
9     return 0;
10 }
```

```
1 Le minimum entre 16 et 32 est 16.
2 Le minimum entre 2+9+7 et 3*8 est 18.
```




Remarquez que nous avons utilisés des expressions composées lors de la deuxième utilisation de la macrofonction `MIN`.

23.3.1 Priorité des opérations

Quelque chose vous a peut-être frappé dans les corrections : pourquoi écrire `(x)` et pas simplement `x` ?

En fait, il s'agit d'une protection en vue d'éviter certaines ambiguïtés. En effet, si l'on n'y prend pas garde, on peut par exemple avoir des surprises dues à la priorité des opérateurs. Prenons l'exemple d'une macro `MUL` qui effectue une multiplication.

```
1 #define MUL(a, b) (a * b)
```

Tel quel, le code peut poser des problèmes. En effet, si nous appelons la macrofonction comme ceci.

```
1 MUL(2+3, 4+5)
```

Nous obtenons comme résultat 19 (la macro sera remplacée par `2 + 3 * 4 + 5`) et non 45, qui est le résultat attendu. Pour garantir la bonne marche de la macrofonction, nous devons rajouter des parenthèses.

```
1 #define MUL(a, b) ((a) * (b))
```

Dans ce cas, nous obtenons bien le résultat souhaité, c'est-à-dire 45 (`((2 + 3) * (4 + 5))`).



Nous vous conseillons de rajouter des parenthèses en cas de doute pour éviter toute erreur.

23.3.2 Les effets de bords

Pour finir, une petite mise en garde : évitez d'utiliser plus d'une fois un paramètre dans la définition d'une macro en vue d'éviter de multiplier d'éventuels **effets de bord**.



Des effets de quoi ?

Un effet de bord est une modification du contexte d'exécution. Vous voilà bien avancé nous direz-vous... En fait, vous en avez déjà rencontré, l'exemple le plus typique étant une affectation.

```
1 a = 10;
```

Dans cet exemple, le contexte d'exécution du programme (qui comprends ses variables) est modifié puisque la valeur d'une variable est changée. Ainsi, imaginez que la macro `MUL` soit appelée comme suit.

```
1 MUL(a = 10, a = 20)
```

Après remplacement, celle-ci donnerait l'expression suivante : `((a = 10) * (a = 20))` qui est assez problématique... En effet, quelle sera la valeur de `a`, finalement ? 10 ou 20 ? Ceci est impossible à dire sans fixer une règle d'évaluation et... la norme n'en prévoit aucune dans ce cas ci.

Aussi, pour éviter ce genre de problèmes tordus, veillez à n'utiliser chaque paramètre qu'une seule fois.

23.4 Les directives conditionnelles

Le préprocesseur dispose de cinq directives conditionnelles : `#if`, `#ifdef`, `#ifndef`, `#elif`, et `#else`. Ces dernières permettent de conserver ou non une portion de code en fonction de la validité d'une condition (si la condition est vraie, le code est gardé sinon il est passé). Chacune de ces directives (ou suite de directives) doit être terminée par une directive `#endif`.

23.4.1 Les directives `#if`, `#elif` et `#else`

Les directives `#if` et `#elif`

Les directives `#if` et `#elif`, comme l'instruction `if`, attendent une expression conditionnelle ; toutefois, celle-ci sont plus restreintes étant donné que nous sommes dans le cadre du préprocesseur. Ce dernier se contentant d'effectuer des substitutions, il n'a par exemple aucune connaissance des mots-clés du langage C ou des variables qui sont employées.

Ainsi, les conditions ne peuvent comporter que des expressions *entières* (ce qui exclut les nombres flottants et les pointeurs) et *constantes* (ce qui exclut l'utilisation d'opérateurs à effets de bord comme `=`). Aussi, les mots-clés et autres identificateurs (hormis les macros) sont présents dans la définition *sont ignorés* (plus précisément, ils sont remplacés par `0`).

L'exemple ci-dessous explicite ceci.

```
1 #if 1.89 > 1.88 /* Incorrect, ce ne sont pas des entiers */
2 #if sizeof(int) == 4 /* Équivalent à 0(0) == 4, 'sizeof' et 'int' étant des mots-clés */
3 #if (a = 20) == 20 /* Équivalent à (0 = 20) == 4, 'a' étant un identificateur */
```

Techniquement, seuls les opérateurs suivants peuvent être utilisés : `+` (binaire ou unaire), `-` (binaire ou unaire), `*`, `/`, `%`, `==`, `!=`, `<=`, `<`, `>`, `>=`, `!`, `&&`, `||`, `?:`, l'opérateur ternaire et les opérateurs de manipulations des *bits*, que nous verrons au chapitre suivant.



Bien entendu, vous pouvez également utiliser des parenthèses afin de régler la priorité des opérations.

L'opérateur `defined`

Le préprocesseur fournit un opérateur supplémentaire utilisable dans les conditions : `defined`. Celui-ci prend comme opérande un nom de macro et retourne 1 ou 0 suivant que ce nom correspond ou non à une macro définie.

```
1 #if defined TAILLE
```

Celui-ci est fréquemment utilisé pour produire des programmes portables. En effet, chaque système et chaque compilateur définissent généralement une ou des macroconstantes qui lui sont propres. En vérifiant si une de ces constantes existe, nous pouvons déterminer sur quelle plate-forme et avec quel compilateur nous compilons et adapter le code en conséquence.



Notez que ces constantes sont propres à un système d'exploitation et/ou compilateur donné, elles ne sont donc pas spécifiées par la norme du langage C.

La directive `#else`

La directive `#else` quant à elle, se comporte comme l'instruction éponyme.

Exemple

Voici un exemple d'utilisation.

```

1  #include <stdio.h>
2
3  #define A 2
4
5  int main(void)
6  {
7      #if A < 0
8          puts("A < 0");
9      #elif A > 0
10         puts("A > 0");
11     #else
12         puts("A == 0");
13     #endif
14     return 0;
15 }
```

```

1  A > 0
```

Notez que les directives conditionnelles peuvent être utilisées à la place des commentaires en vue d'empêcher la compilation d'un morceau de code. L'avantage de cette technique par rapport aux commentaires est qu'elle vous évite de produire des commentaires imbriqués.



```

1  #if 0
2      /* On multiplie par TAILLE */
3      variable *= TAILLE;
4      printf("Variable vaut : %d\n", variable);
5  #endif
```

23.4.2 Les directives #ifdef et #ifndef

Ces deux directives sont en fait la contraction, respectivement, de la directive `#if defined` et `#if !defined`. Si vous n'avez qu'une seule constante à tester, il est plus rapide d'utiliser ces deux directives à la place de leur version longue.

Protection des fichiers d'en-tête

Ceci étant dit, vous devriez à présent être en mesure de comprendre le fonctionnement des directives jusqu'à présent utilisées dans les fichiers en-têtes.

```

1  #ifndef CONSTANCE_H
2  #define CONSTANCE_H
3
4  /* Les déclarations */
5
6  #endif
```

La première directive vérifie que la macroconstante `CONSTANTE_H` n'a pas encore été définie. Si ce n'est pas le cas, elle est définie (ici avec une valeur nulle) et le bloc de la condition (comprenant le contenu du fichier d'en-tête) est inclus. Si elle a déjà été définie, le bloc est sauté et le contenu du fichier n'est ainsi pas à nouveau inclus.



Pour rappel, ceci est nécessaire pour éviter des problèmes d'inclusions multiples, par exemple lorsqu'un fichier A inclut un fichier B, qui lui-même inclut le fichier A.

Avec ce chapitre, vous devriez pouvoir utiliser le préprocesseur de manière basique et éviter plusieurs de ses pièges fréquents. Toutefois, si vous souhaitez aller plus loin et approfondir son utilisation, nous vous conseillons la lecture [du cours de Pouet_forever](#).

Après ce que nous venons de découvrir, voyons comment mettre tout cela en musique à l'aide d'un exercice récapitulatif : réaliser un *Puissance 4*.

24.1 Première étape : le jeu

Dans cette première partie, vous allez devoir réaliser un « Puissance 4 » pour deux joueurs *humains*. Le programme final devra ressembler à ceci.

```

1      1  2  3  4  5  6  7
2      +---+---+---+---+---+---+
3      | | | | | | | |
4      +---+---+---+---+---+---+
5      | | | | | | | |
6      +---+---+---+---+---+---+
7      | | | | | | | |
8      +---+---+---+---+---+---+
9      | | | | | | | |
10     +---+---+---+---+---+---+
11     | | | | | | | |
12     +---+---+---+---+---+---+
13     | | | | | | | |
14     +---+---+---+---+---+---+
15     1  2  3  4  5  6  7
16
17     Joueur 1 : 1
18
19     1  2  3  4  5  6  7
20     +---+---+---+---+---+---+
21     | | | | | | | |
22     +---+---+---+---+---+---+
23     | | | | | | | |
24     +---+---+---+---+---+---+
25     | | | | | | | |
26     +---+---+---+---+---+---+
27     | | | | | | | |
28     +---+---+---+---+---+---+
29     | | | | | | | |
30     +---+---+---+---+---+---+
31     | 0 | | | | | | |
32     +---+---+---+---+---+---+
33     1  2  3  4  5  6  7
34
35     Joueur 2 : 7
36
37     1  2  3  4  5  6  7
38     +---+---+---+---+---+---+
39     | | | | | | | |
40     +---+---+---+---+---+---+
41     | | | | | | | |
42     +---+---+---+---+---+---+

```

```

43 | | | | | | |
44 +---+---+---+---+---+---+
45 | | | | | | |
46 +---+---+---+---+---+---+
47 | | | | | | |
48 +---+---+---+---+---+---+
49 | 0 | | | | | | X |
50 +---+---+---+---+---+---+
51 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
52
53 Joueur 1 :

```

Pour ceux qui ne connaissent pas le jeu Puissance 4, ce dernier se joue à l'aide d'une grille verticale de sept colonnes sur six lignes. Chaque joueur dispose de vingt et un jetons d'une couleur (le plus souvent, rouge et jaune traduit dans notre exemple par les caractères « O » et « X ») et place ceux-ci au sein de la grille à tour de rôle.

Pour gagner le jeu, un joueur doit aligner quatre jetons verticalement, horizontalement ou en oblique. Il s'agit donc du même principe que le Morpion à une différence prêt : la grille est *verticale* ce qui signifie que les jetons tombent au fond de la colonne choisie par le joueur. Pour plus de détails, je vous renvoie à [l'article dédié sur Wikipédia](#).

Maintenant que vous savez cela, il est temps de passer à la réalisation de celui-ci.

Bon travail!

24.2 Correction

Bien, l'heure de la correction est venue.

```

1  #include <ctype.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  #define P4_COLONNES (7)
6  #define P4_LIGNES (6)
7
8  #define J1_JETON ('O')
9  #define J2_JETON ('X')
10
11 #define ACT_ERR (0)
12 #define ACT_JOUER (1)
13 #define ACT_NOUVELLE_SAISIE (2)
14 #define ACT_QUITTER (3)
15
16 #define STATUT_OK (0)
17 #define STATUT_GAGNE (1)
18 #define STATUT_EGALITE (2)
19
20 struct position
21 {
22     int colonne;
23     int ligne;
24 };
25
26 static void affiche_grille(void);
27 static void calcule_position(int, struct position *);
28 static unsigned calcule_nb_jetons_depuis_vers(struct position *, int, int, char);
29 static unsigned calcule_nb_jetons_depuis(struct position *, char);
30 static int coup_valide(int);
31 static int demande_action(int *);
32 static int grille_complete(void);
33 static void initialise_grille(void);
34 static int position_valide(struct position *);
35 static int statut_jeu(struct position *pos, char);
36 static unsigned umax(unsigned, unsigned);
37 static int vider_tampon(FILE *);
38
39 static char grille[P4_COLONNES][P4_LIGNES];

```

```

40
41
42 static void affiche_grille(void)
43 {
44     /*
45      * Affiche la grille pour le ou les joueurs.
46      */
47
48     int col;
49     int lgn;
50
51     putchar('\n');
52
53     for (col = 1; col <= P4_COLONNES; ++col)
54         printf(" %d ", col);
55
56     putchar('\n');
57     putchar('+');
58
59     for (col = 1; col <= P4_COLONNES; ++col)
60         printf("----+");
61
62     putchar('\n');
63
64     for (lgn = 0; lgn < P4_LIGNES; ++lgn)
65     {
66         putchar('|');
67
68         for (col = 0; col < P4_COLONNES; ++col)
69             if (isalpha(grille[col][lgn]))
70                 printf(" %c |", grille[col][lgn]);
71             else
72                 printf(" %c |", ' ');
73
74         putchar('\n');
75         putchar('+');
76
77         for (col = 1; col <= P4_COLONNES; ++col)
78             printf("----+");
79
80         putchar('\n');
81     }
82
83     for (col = 1; col <= P4_COLONNES; ++col)
84         printf(" %d ", col);
85
86     putchar('\n');
87 }
88
89
90 static void calcule_position(int coup, struct position *pos)
91 {
92     /*
93      * Traduit le coup joué en un numéro de colonne et de ligne.
94      */
95
96     int lgn;
97
98     pos->colonne = coup;
99
100    for (lgn = P4_LIGNES - 1; lgn >= 0; --lgn)
101        if (grille[pos->colonne][lgn] == ' ')
102        {
103            pos->ligne = lgn;
104            break;
105        }
106 }
107
108
109 static unsigned calcule_nb_jetons_depuis_vers(struct position *pos, int dpl_hrz, int dpl_vrt, char jeton)
110 {
111     /*
112      * Calcule le nombre de jetons adjacents identiques depuis une position donnée en se

```



```

113     * déplaçant de `dpl_hrz` horizontalement et `dpl_vrt` verticalement.
114     * La fonction s'arrête si un jeton différent ou une case vide est rencontrée ou si
115     * les limites de la grille sont atteintes.
116     */
117
118     struct position tmp;
119     unsigned nb = 1;
120
121     tmp.colonne = pos->colonne + dpl_hrz;
122     tmp.ligne = pos->ligne + dpl_vrt;
123
124     while (position_valide(&tmp))
125     {
126         if (grille[tmp.colonne][tmp.ligne] == jeton)
127             ++nb;
128         else
129             break;
130
131         tmp.colonne += dpl_hrz;
132         tmp.ligne += dpl_vrt;
133     }
134
135     return nb;
136 }
137
138
139 static unsigned calcule_nb_jetons_depuis(struct position *pos, char jeton)
140 {
141     /*
142     * Calcule le nombre de jetons adjacents en vérifiant la colonne courante,
143     * de la ligne courante et des deux obliques courantes.
144     * Pour ce faire, la fonction calcule_nb_jeton_depuis_vers() est appelé à
145     * plusieurs reprises afin de parcourir la grille suivant la vérification
146     * à effectuer.
147     */
148
149     unsigned max;
150
151     max = calcule_nb_jetons_depuis_vers(pos, 0, 1, jeton);
152     max = umax(max, calcule_nb_jetons_depuis_vers(pos, 1, 0, jeton) + \
153     calcule_nb_jetons_depuis_vers(pos, -1, 0, jeton) - 1);
154     max = umax(max, calcule_nb_jetons_depuis_vers(pos, 1, 1, jeton) + \
155     calcule_nb_jetons_depuis_vers(pos, -1, -1, jeton) - 1);
156     max = umax(max, calcule_nb_jetons_depuis_vers(pos, 1, -1, jeton) + \
157     calcule_nb_jetons_depuis_vers(pos, -1, 1, jeton) - 1);
158
159     return max;
160 }
161
162
163 static int coup_valide(int col)
164 {
165     /*
166     * Si la colonne renseignée est inférieur ou égal à zéro
167     * ou que celle-ci est supérieure à la longueur du tableau
168     * ou que la colonne indiquée est saturée
169     * alors le coup est invalide.
170     */
171
172     if (col <= 0 || col > P4_COLONNES || grille[col - 1][0] != ' ')
173         return 0;
174
175     return 1;
176 }
177
178
179 static int demande_action(int *coup)
180 {
181     /*
182     * Demande l'action à effectuer au joueur courant.
183     * S'il entre un chiffre, c'est qu'il souhaite jouer.
184     * S'il entre la lettre « Q » ou « q », c'est qu'il souhaite quitter.
185     * S'il entre autre chose, une nouvelle saisie sera demandée.

```

```

186     */
187
188     char c;
189     int ret = ACT_ERR;
190
191     if (scanf("%d", coup) != 1)
192     {
193         if (scanf("%c", &c) != 1)
194         {
195             fprintf(stderr, "Erreur lors de la saisie\n");
196             return ret;
197         }
198
199         switch (c)
200         {
201             case 'Q':
202             case 'q':
203                 ret = ACT_QUITTER;
204                 break;
205             default:
206                 ret = ACT_NOUVELLE_SAISIE;
207                 break;
208         }
209     }
210     else
211         ret = ACT_JOUER;
212
213     if (!vider_tampon(stdin))
214     {
215         fprintf(stderr, "Erreur lors de la vidange du tampon.\n");
216         ret = ACT_ERR;
217     }
218
219     return ret;
220 }
221
222
223 static int grille_completeness(void)
224 {
225     /*
226      * Détermine si la grille de jeu est complète.
227      */
228
229     unsigned col;
230     unsigned lgn;
231
232     for (col = 0; col < P4_COLONNES; ++col)
233         for (lgn = 0; lgn < P4_LIGNES; ++lgn)
234             if (grille[col][lgn] == ' ')
235                 return 0;
236
237     return 1;
238 }
239
240
241 static void initialise_grille(void)
242 {
243     /*
244      * Initialise les caractères de la grille.
245      */
246
247     unsigned col;
248     unsigned lgn;
249
250     for (col = 0; col < P4_COLONNES; ++col)
251         for (lgn = 0; lgn < P4_LIGNES; ++lgn)
252             grille[col][lgn] = ' ';
253 }
254
255
256 static int position_valide(struct position *pos)
257 {
258     /*

```

```

259     * Vérifie que la position fournie est bien comprise dans la grille.
260     */
261
262     int ret = 1;
263
264     if (pos->colonne >= P4_COLONNES || pos->colonne < 0)
265         ret = 0;
266     else if (pos->ligne >= P4_LIGNES || pos->ligne < 0)
267         ret = 0;
268
269     return ret;
270 }
271
272
273 static int statut_jeu(struct position *pos, char jeton)
274 {
275     /*
276      * Détermine s'il y a lieu de continuer le jeu ou s'il doit être
277      * arrêté parce qu'un joueur a gagné ou que la grille est complète.
278      */
279
280     if (grille_complete())
281         return STATUT_EGALITE;
282     else if (calcule_nb_jetons_depuis(pos, jeton) >= 4)
283         return STATUT_GAGNE;
284
285     return STATUT_OK;
286 }
287
288
289 static unsigned umax(unsigned a, unsigned b)
290 {
291     /*
292      * Retourne le plus grand des deux arguments.
293      */
294
295     return (a > b) ? a : b;
296 }
297
298
299 static int vider_tampon(FILE *fp)
300 {
301     /*
302      * Vide les données en attente de lecture du flux spécifié.
303      */
304
305     int c;
306
307     do
308         c = fgetc(fp);
309     while (c != '\n' && c != EOF);
310
311     return ferror(fp) ? 0 : 1;
312 }
313
314
315 int main(void)
316 {
317     int statut;
318     char jeton = J1_JETON;
319
320     initialise_grille();
321     affiche_grille();
322
323     while (1)
324     {
325         struct position pos;
326         int action;
327         int coup;
328
329         printf("Joueur %d : ", (jeton == J1_JETON) ? 1 : 2);
330
331         action = demande_action(&coup);

```

```

332
333     if (action == ACT_ERR)
334         return EXIT_FAILURE;
335     else if (action == ACT_QUITTER)
336         return 0;
337     else if (action == ACT_NOUVELLE_SAISIE || !coup_valide(coup))
338     {
339         fprintf(stderr, "Vous ne pouvez pas jouer à cet endroit\n");
340         continue;
341     }
342
343     calcule_position(coup - 1, &pos);
344     grille[pos.colonne][pos.ligne] = jeton;
345     affiche_grille();
346     statut = statut_jeu(&pos, jeton);
347
348     if (statut != STATUT_OK)
349         break;
350
351     jeton = (jeton == J1_JETON) ? J2_JETON : J1_JETON;
352 }
353
354 if (statut == STATUT_GAGNE)
355     printf("Le joueur %d a gagné\n", (jeton == J1_JETON) ? 1 : 2);
356 else if (statut == STATUT_EGALITE)
357     printf("Égalité\n");
358
359 return 0;
360 }

```

Le programme commence par initialiser la grille (tous les caractères la composant sont des espaces au début, symbolisant une case vide) et l’afficher.

Ensuite, il est demandé au premier joueur d’effectuer une action. Ceci est réalisé via la fonction `demande_action()` qui attend du joueur soit qu’il entre un numéro de colonne, soit la lettre `Q` ou `q`. Si le joueur entre autre chose ou que la colonne indiquée est invalide, une nouvelle saisie lui est demandée.

Dans le cas où le numéro de colonne est valable, la fonction `calcule_position()` est appelée afin de calculer le numéro de ligne et de stocker celui-ci et le numéro de colonne dans une structure. Après quoi la grille est mise à jour en ajoutant un nouveau jeton et à nouveau affichée.

La fonction `statut_jeu()` entre alors en action et détermine si quatre jetons adjacents sont présents ou non ou si la grille est complète. Ceci est notamment réalisé à l’aide de la fonction `calcule_nb_jetons_depuis()` qui fait elle-même appel à la fonction `calcule_nb_jetons_depuis_vers()`. Si aucune des deux conditions n’est remplie, c’est reparti pour un tour sinon la boucle est quittée et le programme se termine.

La fonction `calcule_nb_jetons_depuis_vers()` mérite que l’on s’attarde un peu sur elle. Celle-ci reçoit une position dans la grille et se déplace suivant les valeurs attribuées à `dpl_hrz` (soit « déplacement horizontal ») et `dpl_vrt` (pour « déplacement vertical »). Celle-ci effectue son parcours aussi longtemps que des jetons identiques sont rencontrés et que le déplacement a lieu dans la grille. Elle retourne ensuite le nombre de jetons identiques rencontrés (notez que nous commençons le décompte à un puisque, par définition, la position indiquée est celle qui vient d’être jouée par un des joueurs).

La fonction `calcule_nb_jetons_depuis()` appelle à plusieurs reprises la fonction `calcule_nb_jetons_depuis_vers()` afin d’obtenir le nombre de jetons adjacents de la colonne courante (déplacement de un vers le bas), de la ligne (deux déplacements : un vers la gauche et un vers la droite) et des deux obliques (deux déplacements pour chacune d’entre-elles).

Notez qu’afin d’éviter de nombreux passages d’arguments, nous avons employé la variable globale `grille` (dont le nom est explicite).



Nous insistons sur le fait que nous vous présentons *une* solution parmi d'autres. Ce n'est pas parce que votre façon de procéder est différente qu'elle est incorrecte.

24.3 Deuxième étape : une petite IA

24.3.1 Introduction

Dans cette seconde partie, votre objectif va être de construire une petite intelligence artificielle. Votre programme va donc devoir demander si un ou deux joueurs sont présents et jouer à la place du second joueur s'il n'y en a qu'un seul.

Afin de vous aider dans la réalisation de cette tâche, nous allons vous présenter un algorithme simple que vous pourrez mettre en œuvre. Le principe est le suivant : pour chaque emplacement possible (il y en aura toujours au maximum sept), nous allons calculer combien de pièces composeraient la ligne si nous jouions à cet endroit *sans tenir compte de notre couleur* (autrement dit, le jeton que nous jouons est considéré comme étant des *deux* couleurs). Si une valeur est plus élevée que les autres, l'emplacement correspondant sera choisi. Si en revanche il y a plusieurs nombres égaux, une des cases sera choisie « au hasard » (nous y reviendrons).

Cet algorithme connaît toutefois une exception : s'il s'avère lors de l'analyse qu'un coup permet à l'ordinateur de gagner, alors le traitement s'arrêtera là et le mouvement est immédiatement joué.

Par exemple, dans la grille ci-dessous, il est possible de jouer à sept endroits (indiqués à l'aide d'un point d'interrogation).

1		1	2	3	4	5	6	7
2	+	+	+	+	+	+	+	+
3					?			
4	+	+	+	+	+	+	+	+
5					O	?		
6	+	+	+	+	+	+	+	+
7					O	X	?	
8	+	+	+	+	+	+	+	+
9					O	O	X	
10	+	+	+	+	+	+	+	+
11			?	?	X	O	O	?
12	+	+	+	+	+	+	+	+
13		?	X	O	O	X	X	X
14	+	+	+	+	+	+	+	+
15		1	2	3	4	5	6	7

Si nous appliquons l'algorithme que nous venons de décrire, nous obtenons les valeurs suivantes.

1		1	2	3	4	5	6	7
2	+	+	+	+	+	+	+	+
3					4			
4	+	+	+	+	+	+	+	+
5					O	2		
6	+	+	+	+	+	+	+	+
7					O	X	2	
8	+	+	+	+	+	+	+	+
9					O	O	X	
10	+	+	+	+	+	+	+	+
11			2	2	X	O	O	3
12	+	+	+	+	+	+	+	+
13		2	X	O	O	X	X	X
14	+	+	+	+	+	+	+	+
15		1	2	3	4	5	6	7

Le programme jouera donc dans la colonne quatre, cette dernière ayant la valeur la plus élevée (ce qui tombe bien puisque l'autre joueur gagnerait si l'ordinateur jouait ailleurs :p).



Pour effectuer le calcul, vous pouvez vous aider des fonctions `long_colonne()`, `long_ligne()` et `long_oblique()` de la correction précédente.

24.3.2 Tirage au sort

Nous vous avons dit que si l'ordinateur doit choisir entre des valeurs identiques, ce dernier devrait en choisir une « au hasard ». Cependant, un ordinateur est en vérité incapable d'effectuer cette tâche (c'est pour cela que nous employons des guillemets). Pour y remédier, il existe des algorithmes qui permettent de produire des suites de nombres dits *pseudo-aléatoires*, c'est-à-dire qui s'approchent de suites aléatoires sur le plan statistique.

La plupart de ceux-ci fonctionnent à partir d'une *graine*, c'est-à-dire un nombre de départ qui va déterminer tout le reste de la suite. C'est ce système qui a été choisi par la bibliothèque standard du C. Deux fonctions vous sont dès lors proposées : `srand()` et `rand()` (elles sont déclarées dans l'en-tête `<stdlib.h>`).

```
1 void srand(unsigned int seed);
2 int rand(void);
```

La fonction `srand()` est utilisée pour initialiser la génération de nombres à l'aide de la graine fournie en argument (un entier non signé en l'espèce). Le plus souvent vous ne l'appellerez qu'une seule fois au début de votre programme sauf si vous souhaitez réinitialiser la génération de nombres.

La fonction `rand()` retourne un nombre pseudo-aléatoire compris entre zéro et `RAND_MAX` (qui est une constante également définie dans l'en-tête `<stdlib.h>`) suivant la graine fournie à la fonction `srand()`.

Toutefois, il y a un petit bémol : étant donné que l'algorithme de génération se base sur la graine fournie pour construire la suite de nombres, une même graine donnera *toujours la même suite* ! Dès lors, comment faire pour que les nombres générés ne soient pas identiques entre deux exécutions du programme ? Pour que cela soit possible, nous devrions fournir un nombre différent à `srand()` à chaque fois, mais comment obtenir un tel nombre ?

C'est ici que la fonction `time()` (déclarée dans l'en-tête `<time.h>`) entre en jeu.

```
1 time_t time(time_t *t);
```

La fonction `time()` retourne la date actuelle sous forme d'une valeur de type `time_t` (qui est un nombre entier ou flottant). Le plus souvent, il s'agit du nombre de secondes écoulé depuis une date fixée arbitrairement appelée *Epoch* en anglais. Cette valeur peut également être stockée dans une variable de type `time_t` dont l'adresse lui est fournie en argument (un pointeur nul peut lui être envoyé si cela n'est pas nécessaire). En cas d'erreur, la fonction retourne la valeur `-1` convertie vers le type `time_t`.

Pour obtenir des nombres différents à chaque exécution, nous pouvons donc appeler `srand()` avec le retour de la fonction `time()` converti en entier non signé. L'exemple suivant génère donc une suite de trois nombres pseudo-aléatoires différents à chaque exécution (en vérité à chaque exécution espacée d'une seconde).

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4
5
6 int main(void)
7 {
8     time_t t;
```

```

9
10     if (time(&t) == (time_t)-1)
11     {
12         fprintf(stderr, "Impossible d'obtenir la date courante\n");
13         return EXIT_FAILURE;
14     }
15
16     srand((unsigned)t);
17     printf("1 : %d\n", rand());
18     printf("2 : %d\n", rand());
19     printf("3 : %d\n", rand());
20     return 0;
21 }

```

Tirer un nombre dans un intervalle donné

Il est possible de générer des nombres dans un intervalle précis en procédant en deux temps. Tout d'abord, il nous est nécessaire d'obtenir un nombre compris entre zéro inclus et un exclus. Pour ce faire, nous pouvons diviser le nombre pseudo-aléatoire obtenu par la plus grande valeur qui peut être retournée par la fonction `rand()` augmentée de un. Celle-ci nous est fournie via la macroconstante `RAND_MAX` qui est définie dans l'en-tête `<stdlib.h>`.

```

1 double nb_aleatoire(void)
2 {
3     return rand() / (RAND_MAX + 1.);
4 }

```

Notez que nous avons bien indiqué que la constante `1.` est un nombre flottant afin que le résultat de l'opération soit de type `double`.

Ensuite, il nous suffit de multiplier le nombre obtenu par la différence entre le maximum et le minimum augmenté de un et d'y ajouter le minimum.

```

1 int nb_aleatoire_entre(int min, int max)
2 {
3     return nb_aleatoire() * (max - min + 1) + min;
4 }

```

Afin d'illustrer ce qui vient d'être dit, le code suivant affiche trois nombres pseudo-aléatoires compris entre zéro et dix inclus.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4
5
6 static double nb_aleatoire(void)
7 {
8     return rand() / (RAND_MAX + 1.);
9 }
10
11 static int nb_aleatoire_entre(int min, int max)
12 {
13     return nb_aleatoire() * (max - min + 1) + min;
14 }
15
16
17 int main(void)
18 {
19     time_t t;
20
21     if (time(&t) == (time_t)-1)
22     {
23         fprintf(stderr, "Impossible d'obtenir la date courante\n");
24     }

```

```

25     return EXIT_FAILURE;
26 }
27
28 srand((unsigned)t);
29 printf("1 : %d\n", nb_aleatoire_entre(0, 10));
30 printf("2 : %d\n", nb_aleatoire_entre(0, 10));
31 printf("3 : %d\n", nb_aleatoire_entre(0, 10));
32 return 0;
33 }

```

À présent, c'est à vous !

24.4 Correction

```

1  #include <ctype.h>
2  #include <stddef.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  #define P4_COLONNES (7)
7  #define P4_LIGNES (6)
8
9  #define J1_JETON ('O')
10 #define J2_JETON ('X')
11
12 #define ACT_ERR (0)
13 #define ACT_JOUER (1)
14 #define ACT_NOUVELLE_SAISIE (2)
15 #define ACT_QUITTER (3)
16
17 #define STATUT_OK (0)
18 #define STATUT_GAGNE (1)
19 #define STATUT_EGALITE (2)
20
21 struct position
22 {
23     int colonne;
24     int ligne;
25 };
26
27 static void affiche_grille(void);
28 static void calcule_position(int, struct position *);
29 static unsigned calcule_nb_jetons_depuis_vers(struct position *, int, int, char);
30 static unsigned calcule_nb_jetons_depuis(struct position *, char);
31 static int coup_valide(int);
32 static int demande_action(int *);
33 static int demande_nb_joueur(void);
34 static int grille_complete(void);
35 static int ia(void);
36 static void initialise_grille(void);
37 double nb_aleatoire(void);
38 int nb_aleatoire_entre(int, int);
39 static int position_valide(struct position *);
40 static int statut_jeu(struct position *pos, char);
41 static unsigned umax(unsigned, unsigned);
42 static int vider_tampon(FILE *);
43
44 static char grille[P4_COLONNES][P4_LIGNES];
45
46
47 static void affiche_grille(void)
48 {
49     /*
50      * Affiche la grille pour le ou les joueurs.
51      */
52
53     int col;
54     int lgn;
55
56     putchar('\n');

```



```

57
58     for (col = 1; col <= P4_COLONNES; ++col)
59         printf(" %d ", col);
60
61     putchar('\n');
62     putchar('+');
63
64     for (col = 1; col <= P4_COLONNES; ++col)
65         printf("----+");
66
67     putchar('\n');
68
69     for (lgn = 0; lgn < P4_LIGNES; ++lgn)
70     {
71         putchar('|');
72
73         for (col = 0; col < P4_COLONNES; ++col)
74             if (isalpha(grille[col][lgn]))
75                 printf(" %c |", grille[col][lgn]);
76             else
77                 printf(" %c |", ' ');
78
79         putchar('\n');
80         putchar('+');
81
82         for (col = 1; col <= P4_COLONNES; ++col)
83             printf("----+");
84
85         putchar('\n');
86     }
87
88     for (col = 1; col <= P4_COLONNES; ++col)
89         printf(" %d ", col);
90
91     putchar('\n');
92 }
93
94
95 static void calcule_position(int coup, struct position *pos)
96 {
97     /*
98      * Traduit le coup joué en un numéro de colonne et de ligne.
99      */
100
101     int lgn;
102
103     pos->colonne = coup;
104
105     for (lgn = P4_LIGNES - 1; lgn >= 0; --lgn)
106         if (grille[pos->colonne][lgn] == ' ')
107         {
108             pos->ligne = lgn;
109             break;
110         }
111 }
112
113
114 static unsigned calcule_nb_jetons_depuis_vers(struct position *pos, int dpl_hrz, int dpl_vrt, char jeton)
115 {
116     /*
117      * Calcule le nombre de jetons adjacents identiques depuis une position donnée en se
118      * déplaçant de `dpl_hrz` horizontalement et `dpl_vrt` verticalement.
119      * La fonction s'arrête si un jeton différent ou une case vide est rencontrée ou si
120      * les limites de la grille sont atteintes.
121      */
122
123     struct position tmp;
124     unsigned nb = 1;
125
126     tmp.colonne = pos->colonne + dpl_hrz;
127     tmp.ligne = pos->ligne + dpl_vrt;
128
129     while (position_valide(&tmp))

```

```

130     {
131         if (grille[tmp.colonne][tmp.ligne] == jeton)
132             ++nb;
133         else
134             break;
135
136         tmp.colonne += dpl_hrz;
137         tmp.ligne += dpl_vrt;
138     }
139
140     return nb;
141 }
142
143
144 static unsigned calcule_nb_jetons_depuis(struct position *pos, char jeton)
145 {
146     /*
147      * Calcule le nombre de jetons adjacents en vérifiant la colonne courante,
148      * de la ligne courante et des deux obliques courantes.
149      * Pour ce faire, la fonction calcule_nb_jeton_depuis_vers() est appelé à
150      * plusieurs reprises afin de parcourir la grille suivant la vérification
151      * à effectuer.
152      */
153
154     unsigned max;
155
156     max = calcule_nb_jetons_depuis_vers(pos, 0, 1, jeton);
157     max = umax(max, calcule_nb_jetons_depuis_vers(pos, 1, 0, jeton) + \
158         calcule_nb_jetons_depuis_vers(pos, -1, 0, jeton) - 1);
159     max = umax(max, calcule_nb_jetons_depuis_vers(pos, 1, 1, jeton) + \
160         calcule_nb_jetons_depuis_vers(pos, -1, -1, jeton) - 1);
161     max = umax(max, calcule_nb_jetons_depuis_vers(pos, 1, -1, jeton) + \
162         calcule_nb_jetons_depuis_vers(pos, -1, 1, jeton) - 1);
163
164     return max;
165 }
166
167
168 static int coup_valide(int col)
169 {
170     /*
171      * Si la colonne renseignée est inférieure ou égale à zéro
172      * ou que celle-ci est supérieure à la longueur du tableau
173      * ou que la colonne indiquée est saturée
174      * alors le coup est invalide.
175      */
176
177     if (col <= 0 || col > P4_COLONNES || grille[col - 1][0] != ' ')
178         return 0;
179
180     return 1;
181 }
182
183
184 static int demande_action(int *coup)
185 {
186     /*
187      * Demande l'action à effectuer au joueur courant.
188      * S'il entre un chiffre, c'est qu'il souhaite jouer.
189      * S'il entre la lettre « Q » ou « q », c'est qu'il souhaite quitter.
190      * S'il entre autre chose, une nouvelle saisie sera demandée.
191      */
192
193     char c;
194     int ret = ACT_ERR;
195
196     if (scanf("%d", coup) != 1)
197     {
198         if (scanf("%c", &c) != 1)
199         {
200             fprintf(stderr, "Erreur lors de la saisie\n");
201             return ret;
202         }
203     }

```

```

203
204     switch (c)
205     {
206     case 'Q':
207     case 'q':
208         ret = ACT_QUITTER;
209         break;
210     default:
211         ret = ACT_NOUVELLE_SAISIE;
212         break;
213     }
214 }
215 else
216     ret = ACT_JOUER;
217
218 if (!vider_tampon(stdin))
219 {
220     fprintf(stderr, "Erreur lors de la vidange du tampon.\n");
221     ret = ACT_ERR;
222 }
223
224 return ret;
225 }
226
227
228 static int demande_nb_joueur(void)
229 {
230     /*
231      * Demande et récupère le nombre de joueurs.
232      */
233
234     int njeueur = 0;
235
236     while (1)
237     {
238         printf("Combien de joueurs prennent part à cette partie ? ");
239
240         if (scanf("%d", &njeueur) != 1 && ferror(stdin))
241         {
242             fprintf(stderr, "Erreur lors de la saisie\n");
243             return 0;
244         }
245         else if (njeueur != 1 && njeueur != 2)
246             fprintf(stderr, "Plait-il ?\n");
247         else
248             break;
249
250         if (!vider_tampon(stdin))
251         {
252             fprintf(stderr, "Erreur lors de la vidange du tampon.\n");
253             return 0;
254         }
255     }
256
257     return njeueur;
258 }
259
260
261 static int grille_complete(void)
262 {
263     /*
264      * Détermine si la grille de jeu est complète.
265      */
266
267     unsigned col;
268     unsigned lgn;
269
270     for (col = 0; col < P4_COLONNES; ++col)
271         for (lgn = 0; lgn < P4_LIGNES; ++lgn)
272             if (grille[col][lgn] == ' ')
273                 return 0;
274
275     return 1;

```

```

276 }
277
278
279 static int ia(void)
280 {
281     /*
282      * Fonction mettant en œuvre l'IA présentée.
283      * Assigne une valeur pour chaque colonne libre et retourne ensuite le numéro de
284      * colonne ayant la plus haute valeur. Dans le cas où plusieurs valeurs égales sont
285      * générées, un numéro de colonne est « choisi au hasard » parmi celles-ci.
286      */
287
288     unsigned meilleurs_col[P4_COLONNES];
289     unsigned nb_meilleurs_col = 0;
290     unsigned max = 0;
291     unsigned col;
292
293     for (col = 0; col < P4_COLONNES; ++col)
294     {
295         struct position pos;
296         unsigned longueur;
297
298         if (grille[col][0] != ' ')
299             continue;
300
301         calcule_position(col, &pos);
302         longueur = calcule_nb_jetons_depuis(&pos, J2_JETON);
303
304         if (longueur >= 4)
305             return col;
306
307         longueur = umax(longueur, calcule_nb_jetons_depuis(&pos, J1_JETON));
308
309         if (longueur >= max)
310         {
311             if (longueur > max)
312             {
313                 nb_meilleurs_col = 0;
314                 max = longueur;
315             }
316
317             meilleurs_col[nb_meilleurs_col++] = col;
318         }
319     }
320
321     return meilleurs_col[nb_aleatoire_entre(0, nb_meilleurs_col - 1)];
322 }
323
324
325 static void initialise_grille(void)
326 {
327     /*
328      * Initialise les caractères de la grille.
329      */
330
331     unsigned col;
332     unsigned lgn;
333
334     for (col = 0; col < P4_COLONNES; ++col)
335         for (lgn = 0; lgn < P4_LIGNES; ++lgn)
336             grille[col][lgn] = ' ';
337 }
338
339
340 static unsigned umax(unsigned a, unsigned b)
341 {
342     /*
343      * Retourne le plus grand des deux arguments.
344      */
345
346     return (a > b) ? a : b;
347 }
348

```

```

349
350 double nb_aleatoire(void)
351 {
352     /*
353      * Retourne un nombre pseudo-aléatoire compris entre zéro inclus et un exclus.
354      */
355
356     return rand() / ((double)RAND_MAX + 1.);
357 }
358
359
360 int nb_aleatoire_entre(int min, int max)
361 {
362     /*
363      * Retourne un nombre pseudo-aléatoire entre `min` et `max` inclus.
364      */
365
366     return nb_aleatoire() * (max - min + 1) + min;
367 }
368
369
370 static int position_valide(struct position *pos)
371 {
372     /*
373      * Vérifie que la position fournie est bien comprise dans la grille.
374      */
375
376     int ret = 1;
377
378     if (pos->colonne >= P4_COLONNES || pos->colonne < 0)
379         ret = 0;
380     else if (pos->ligne >= P4_LIGNES || pos->ligne < 0)
381         ret = 0;
382
383     return ret;
384 }
385
386
387 static int statut_jeu(struct position *pos, char jeton)
388 {
389     /*
390      * Détermine s'il y a lieu de continuer le jeu ou s'il doit être
391      * arrêté parce qu'un joueur a gagné ou que la grille est complète.
392      */
393
394     if (grille_complete())
395         return STATUT_EGALITE;
396     else if (calcule_nb_jetons_depuis(pos, jeton) >= 4)
397         return STATUT_GAGNE;
398
399     return STATUT_OK;
400 }
401
402
403 static int vider_tampon(FILE *fp)
404 {
405     /*
406      * Vide les données en attente de lecture du flux spécifié.
407      */
408
409     int c;
410
411     do
412         c = fgetc(fp);
413     while (c != '\n' && c != EOF);
414
415     return ferror(fp) ? 0 : 1;
416 }
417
418
419 int main(void)
420 {
421     int statut;

```

```

422     char jeton = J1_JETON;
423     int njeueur;
424
425     initialise_grille();
426     affiche_grille();
427     njeueur = demande_nb_joueur();
428
429     if (!njeueur)
430         return EXIT_FAILURE;
431
432     while (1)
433     {
434         struct position pos;
435         int action;
436         int coup;
437
438         if (njeueur == 1 && jeton == J2_JETON)
439         {
440             coup = ia();
441             printf("Joueur 2 : %d\n", coup + 1);
442             calcule_position(coup, &pos);
443         }
444         else
445         {
446             printf("Joueur %d : ", (jeton == J1_JETON) ? 1 : 2);
447             action = demande_action(&coup);
448
449             if (action == ACT_ERR)
450                 return EXIT_FAILURE;
451             else if (action == ACT_QUITTER)
452                 return 0;
453             else if (action == ACT_NOUVELLE_SAISIE || !coup_valide(coup))
454             {
455                 fprintf(stderr, "Vous ne pouvez pas jouer à cet endroit\n");
456                 continue;
457             }
458
459             calcule_position(coup - 1, &pos);
460         }
461
462         grille[pos.colonne][pos.ligne] = jeton;
463         affiche_grille();
464         statut = statut_jeu(&pos, jeton);
465
466         if (statut != STATUT_OK)
467             break;
468
469         jeton = (jeton == J1_JETON) ? J2_JETON : J1_JETON;
470     }
471
472     if (statut == STATUT_GAGNE)
473         printf("Le joueur %d a gagné\n", (jeton == J1_JETON) ? 1 : 2);
474     else if (statut == STATUT_EGALITE)
475         printf("Égalité\n");
476
477     return 0;
478 }

```

Le programme demande désormais combien de joueurs sont présents. Dans le cas où ils sont deux, le programme se comporte de la même manière que précédemment. S'il n'y en a qu'un seul, la fonction `ia()` est appelée lors du tour du deuxième joueur. Cette fonction retourne un numéro de colonne après analyse de la grille. Notez également que les fonctions `time()` et `srand()` sont appelées au début afin d'initialiser la génération de nombre pseudo-aléatoires.

La fonction `ia()` agit comme suit :

- les numéros des colonnes ayant les plus grandes valeurs calculées sont stockés dans le tableau `meilleurs_col`;
- si la colonne est complète, celle-ci est passée;
- si la colonne est incomplète, la fonction `calcule_position_depuis()` est appelée afin de détermi-

ner combien de pièces seraient adjacentes si un jeton était posé à cet endroit. S'il s'avère que l'ordinateur peut gagner, la fonction retourne immédiatement le numéro de la colonne courante ;

- un des numéros de colonne présent dans le tableau `meilleurs_col` est tiré « au hasard » et est retourné.

24.5 Troisième et dernière étape : un système de sauvegarde/-restauration

Pour terminer, nous allons ajouter un système de sauvegarde/restauration à notre programme.

Durant une partie, un utilisateur doit pouvoir demander à sauvegarder le jeu courant ou de charger une ancienne partie en lieu et place d'entrer un numéro de colonne.

À vous de voir comment organiser les données au sein d'un fichier et comment les récupérer depuis votre programme.

24.6 Correction

```

1  #include <ctype.h>
2  #include <stddef.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <string.h>
6
7  #define P4_COLONNES (7)
8  #define P4_LIGNES (6)
9
10 #define J1_JETON ('O')
11 #define J2_JETON ('X')
12
13 #define ACT_ERR (0)
14 #define ACT_JOUER (1)
15 #define ACT_NOUVELLE_SAISIE (2)
16 #define ACT_SAUVEGARDER (3)
17 #define ACT_CHARGER (4)
18 #define ACT_QUITTER (5)
19
20 #define STATUT_OK (0)
21 #define STATUT_GAGNE (1)
22 #define STATUT_EGALITE (2)
23
24 #define MAX_NOM (255)
25
26 struct position
27 {
28     int colonne;
29     int ligne;
30 };
31
32 static void affiche_grille(void);
33 static void calcule_position(int, struct position *);
34 static unsigned calcule_nb_jetons_depuis_vers(struct position *, int, int, char);
35 static unsigned calcule_nb_jetons_depuis(struct position *, char);
36 static int charger_jeu(char *nom, char *, int *);
37 static int coup_valide(int);
38 static int demande_action(int *);
39 static int demande_fichier(char *, size_t);
40 static int demande_nb_joueur(void);
41 static int grille_complete(void);
42 static int ia(void);
43 static void initialise_grille(void);
44 double nb_aleatoire(void);
45 int nb_aleatoire_entre(int, int);
46 static int position_valide(struct position *);
47 static int sauvegarder_jeu(char *nom, char, int);

```

```

48 static int statut_jeu(struct position *pos, char);
49 static unsigned umax(unsigned, unsigned);
50 static int vider_tampon(FILE *);
51
52 static char grille[P4_COLONNES][P4_LIGNES];
53
54
55 static void affiche_grille(void)
56 {
57     /*
58      * Affiche la grille pour le ou les joueurs.
59      */
60
61     int col;
62     int lgn;
63
64     putchar('\n');
65
66     for (col = 1; col <= P4_COLONNES; ++col)
67         printf(" %d ", col);
68
69     putchar('\n');
70     putchar('+');
71
72     for (col = 1; col <= P4_COLONNES; ++col)
73         printf("----+");
74
75     putchar('\n');
76
77     for (lgn = 0; lgn < P4_LIGNES; ++lgn)
78     {
79         putchar('|');
80
81         for (col = 0; col < P4_COLONNES; ++col)
82             if (isalpha(grille[col][lgn]))
83                 printf(" %c |", grille[col][lgn]);
84             else
85                 printf(" %c |", ' ');
86
87         putchar('\n');
88         putchar('+');
89
90         for (col = 1; col <= P4_COLONNES; ++col)
91             printf("----+");
92
93         putchar('\n');
94     }
95
96     for (col = 1; col <= P4_COLONNES; ++col)
97         printf(" %d ", col);
98
99     putchar('\n');
100 }
101
102
103 static void calcule_position(int coup, struct position *pos)
104 {
105     /*
106      * Traduit le coup joué en un numéro de colonne et de ligne.
107      */
108
109     int lgn;
110
111     pos->colonne = coup;
112
113     for (lgn = P4_LIGNES - 1; lgn >= 0; --lgn)
114         if (grille[pos->colonne][lgn] == ' ')
115         {
116             pos->ligne = lgn;
117             break;
118         }
119 }
120

```



```

121
122 static unsigned calcule_nb_jetons_depuis_vers(struct position *pos, int dpl_hrz, int dpl_vrt, char jeton)
123 {
124     /*
125      * Calcule le nombre de jetons adjacents identiques depuis une position donnée en se
126      * déplaçant de `dpl_hrz` horizontalement et `dpl_vrt` verticalement.
127      * La fonction s'arrête si un jeton différent ou une case vide est rencontrée ou si
128      * les limites de la grille sont atteintes.
129      */
130
131     struct position tmp;
132     unsigned nb = 1;
133
134     tmp.colonne = pos->colonne + dpl_hrz;
135     tmp.ligne = pos->ligne + dpl_vrt;
136
137     while (position_valide(&tmp))
138     {
139         if (grille[tmp.colonne][tmp.ligne] == jeton)
140             ++nb;
141         else
142             break;
143
144         tmp.colonne += dpl_hrz;
145         tmp.ligne += dpl_vrt;
146     }
147
148     return nb;
149 }
150
151
152 static unsigned calcule_nb_jetons_depuis(struct position *pos, char jeton)
153 {
154     /*
155      * Calcule le nombre de jetons adjacents en vérifiant la colonne courante,
156      * de la ligne courante et des deux obliques courantes.
157      * Pour ce faire, la fonction calcule_nb_jetons_depuis_vers() est appelé à
158      * plusieurs reprises afin de parcourir la grille suivant la vérification
159      * à effectuer.
160      */
161
162     unsigned max;
163
164     max = calcule_nb_jetons_depuis_vers(pos, 0, 1, jeton);
165     max = umax(max, calcule_nb_jetons_depuis_vers(pos, 1, 0, jeton) + \
166     calcule_nb_jetons_depuis_vers(pos, -1, 0, jeton) - 1);
167     max = umax(max, calcule_nb_jetons_depuis_vers(pos, 1, 1, jeton) + \
168     calcule_nb_jetons_depuis_vers(pos, -1, -1, jeton) - 1);
169     max = umax(max, calcule_nb_jetons_depuis_vers(pos, 1, -1, jeton) + \
170     calcule_nb_jetons_depuis_vers(pos, -1, 1, jeton) - 1);
171
172     return max;
173 }
174
175
176 static int charger_jeu(char *nom, char *jeton, int *njourneur)
177 {
178     /*
179      * Charge une partie existante depuis le fichier spécifié.
180      */
181
182     FILE *fp;
183     unsigned col;
184     unsigned lgn;
185
186     fp = fopen(nom, "r");
187
188     if (fp == NULL)
189     {
190         fprintf(stderr, "Impossible d'ouvrir le fichier %s\n", nom);
191         return 0;
192     }
193     else if (fscanf(fp, "%c%d", jeton, njourneur) != 2)

```

```

194     {
195         fprintf(stderr, "Impossible de récupérer le joueur et le nombre de joueurs\n");
196         return 0;
197     }
198
199     for (col = 0; col < P4_COLONNES; ++col)
200         for (lgn = 0; lgn < P4_LIGNES; ++lgn)
201             {
202                 if (fscanf(fp, "%c", &grille[col][lgn]) != 1)
203                     {
204                         fprintf(stderr, "Impossible de récupérer la grille\n");
205                         return 0;
206                     }
207             }
208
209     if (fclose(fp) != 0)
210     {
211         fprintf(stderr, "Impossible de fermer le fichier %s\n", nom);
212         return 0;
213     }
214
215     return 1;
216 }
217
218
219 static int coup_valide(int col)
220 {
221     /*
222      * Si la colonne renseignée est inférieure ou égal à zéro
223      * ou que celle-ci est supérieure à la longueur du tableau
224      * ou que la colonne indiquée est saturée
225      * alors le coup est invalide.
226      */
227
228     if (col <= 0 || col > P4_COLONNES || grille[col - 1][0] != ' ')
229         return 0;
230
231     return 1;
232 }
233
234
235 static int demande_action(int *coup)
236 {
237     /*
238      * Demande l'action à effectuer au joueur courant.
239      * S'il entre un chiffre, c'est qu'il souhaite jouer.
240      * S'il entre la lettre « Q » ou « q », c'est qu'il souhaite quitter.
241      * S'il entre autre chose, une nouvelle saisie sera demandée.
242      */
243
244     char c;
245     int ret = ACT_ERR;
246
247     if (scanf("%d", coup) != 1)
248     {
249         if (scanf("%c", &c) != 1)
250             {
251                 fprintf(stderr, "Erreur lors de la saisie\n");
252                 return ret;
253             }
254
255         switch (c)
256         {
257             case 'Q':
258             case 'q':
259                 ret = ACT_QUITTER;
260                 break;
261
262             case 'C':
263             case 'c':
264                 ret = ACT_CHARGER;
265                 break;
266

```

```

267     case 'S':
268     case 's':
269         ret = ACT_SAUVEGARDER;
270         break;
271
272     default:
273         ret = ACT_NOUVELLE_SAISIE;
274         break;
275     }
276 }
277 else
278     ret = ACT_JOUER;
279
280 if (!vider_tampon(stdin))
281 {
282     fprintf(stderr, "Erreur lors de la vidange du tampon.\n");
283     ret = ACT_ERR;
284 }
285
286 return ret;
287 }
288
289
290 static int demande_fichier(char *nom, size_t max)
291 {
292     /*
293      * Demande et récupère un nom de fichier.
294      */
295
296     char *nl;
297
298     printf("Veuillez entrer un nom de fichier : ");
299
300     if (fgets(nom, max, stdin) == NULL)
301     {
302         fprintf(stderr, "Erreur lors de la saisie\n");
303         return 0;
304     }
305
306     nl = strchr(nom, '\n');
307
308     if (nl == NULL && !vider_tampon(stdin))
309     {
310         fprintf(stderr, "Erreur lors de la vidange du tampon.\n");
311         return 0;
312     }
313
314     if (nl != NULL)
315         *nl = '\0';
316
317     return 1;
318 }
319
320
321 static int demande_nb_joueur(void)
322 {
323     /*
324      * Demande et récupère le nombre de joueurs.
325      */
326
327     int njeueur = 0;
328
329     while (1)
330     {
331         printf("Combien de joueurs prennent part à cette partie ? ");
332
333         if (scanf("%d", &njeueur) != 1 && ferror(stdin))
334         {
335             fprintf(stderr, "Erreur lors de la saisie\n");
336             return 0;
337         }
338         else if (!vider_tampon(stdin))
339         {

```

```

340         fprintf(stderr, "Erreur lors de la vidange du tampon.\n");
341         return 0;
342     }
343
344     if (njourer != 1 && njourer != 2)
345         fprintf(stderr, "Plait-il ?\n");
346     else
347         break;
348 }
349
350 return njourer;
351 }
352
353
354 static int grille_complete(void)
355 {
356     /*
357      * Détermine si la grille de jeu est complète.
358      */
359
360     unsigned col;
361     unsigned lgn;
362
363     for (col = 0; col < P4_COLONNES; ++col)
364         for (lgn = 0; lgn < P4_LIGNES; ++lgn)
365             if (grille[col][lgn] == ' ')
366                 return 0;
367
368     return 1;
369 }
370
371
372 static int ia(void)
373 {
374     /*
375      * Fonction mettant en œuvre l'IA présentée.
376      * Assigne une valeur pour chaque colonne libre et retourne ensuite le numéro de
377      * colonne ayant la plus haute valeur. Dans le cas où plusieurs valeurs égales sont
378      * générées, un numéro de colonne est « choisi au hasard » parmi celles-ci.
379      */
380
381     unsigned meilleurs_col[P4_COLONNES];
382     unsigned nb_meilleurs_col = 0;
383     unsigned max = 0;
384     unsigned col;
385
386     for (col = 0; col < P4_COLONNES; ++col)
387     {
388         struct position pos;
389         unsigned longueur;
390
391         if (grille[col][0] != ' ')
392             continue;
393
394         calcule_position(col, &pos);
395         longueur = calcule_nb_jetons_depuis(&pos, J2_JETON);
396
397         if (longueur >= 4)
398             return col;
399
400         longueur = umax(longueur, calcule_nb_jetons_depuis(&pos, J1_JETON));
401
402         if (longueur >= max)
403         {
404             if (longueur > max)
405             {
406                 nb_meilleurs_col = 0;
407                 max = longueur;
408             }
409
410             meilleurs_col[nb_meilleurs_col++] = col;
411         }
412     }

```

```

413     return meilleurs_col[nb_aleatoire_entre(0, nb_meilleurs_col - 1)];
414 }
415
416
417 static void initialise_grille(void)
418 {
419     /*
420      * Initialise les caractères de la grille.
421      */
422
423     unsigned col;
424     unsigned lgn;
425
426     for (col = 0; col < P4_COLONNES; ++col)
427         for (lgn = 0; lgn < P4_LIGNES; ++lgn)
428             grille[col][lgn] = ' ';
429 }
430
431
432 static unsigned umax(unsigned a, unsigned b)
433 {
434     /*
435      * Retourne le plus grand des deux arguments.
436      */
437
438     return (a > b) ? a : b;
439 }
440
441
442 double nb_aleatoire(void)
443 {
444     /*
445      * Retourne un nombre pseudo-aléatoire compris entre zéro inclus et un exclus.
446      */
447
448     return rand() / ((double)RAND_MAX + 1.);
449 }
450
451
452 int nb_aleatoire_entre(int min, int max)
453 {
454     /*
455      * Retourne un nombre pseudo-aléatoire entre `min` et `max` inclus.
456      */
457
458     return nb_aleatoire() * (max - min + 1) + min;
459 }
460
461
462 static int position_valide(struct position *pos)
463 {
464     /*
465      * Vérifie que la position fournie est bien comprise dans la grille.
466      */
467
468     int ret = 1;
469
470     if (pos->colonne >= P4_COLONNES || pos->colonne < 0)
471         ret = 0;
472     else if (pos->ligne >= P4_LIGNES || pos->ligne < 0)
473         ret = 0;
474
475     return ret;
476 }
477
478
479 static int sauvegarder_jeu(char *nom, char jeton, int njeueur)
480 {
481     /*
482      * Sauvegarde la partie courant dans le fichier indiqué.
483      */
484
485

```

```

486 FILE *fp;
487 unsigned col;
488 unsigned lgn;
489
490 fp = fopen(nom, "w");
491
492 if (fp == NULL)
493 {
494     fprintf(stderr, "Impossible d'ouvrir le fichier %s\n", nom);
495     return 0;
496 }
497 else if (fprintf(fp, "%c%d", jeton, njeueur) < 0)
498 {
499     fprintf(stderr, "Impossible de sauvegarder le joueur courant\n");
500     return 0;
501 }
502
503 if (fputc('|', fp) == EOF)
504 {
505     fprintf(stderr, "Impossible de sauvegarder la grille\n");
506     return 0;
507 }
508
509 for (col = 0; col < P4_COLONNES; ++col)
510 {
511     for (lgn = 0; lgn < P4_LIGNES; ++lgn)
512     {
513         if (fprintf(fp, "%c|", grille[col][lgn]) < 0)
514         {
515             fprintf(stderr, "Impossible de sauvegarder la grille\n");
516             return 0;
517         }
518     }
519 }
520
521 if (fputc('\n', fp) == EOF)
522 {
523     fprintf(stderr, "Impossible de sauvegarder la grille\n");
524     return 0;
525 }
526
527 if (fclose(fp) != 0)
528 {
529     fprintf(stderr, "Impossible de fermer le fichier %s\n", nom);
530     return 0;
531 }
532
533 printf("La partie a bien été sauvegardée dans le fichier %s\n", nom);
534 return 1;
535 }
536
537 static int statut_jeu(struct position *pos, char jeton)
538 {
539     /*
540      * Détermine s'il y a lieu de continuer le jeu ou s'il doit être
541      * arrêté parce qu'un joueur a gagné ou que la grille est complète.
542      */
543
544     if (grille_complete())
545         return STATUT_EGALITE;
546     else if (calcule_nb_jetons_depuis(pos, jeton) >= 4)
547         return STATUT_GAGNE;
548
549     return STATUT_OK;
550 }
551
552
553 static int vider_tampon(FILE *fp)
554 {
555     /*
556      * Vide les données en attente de lecture du flux spécifié.
557      */
558

```

```

559
560     int c;
561
562     do
563         c = fgetc(fp);
564     while (c != '\n' && c != EOF);
565
566     return ferror(fp) ? 0 : 1;
567 }
568
569
570 int main(void)
571 {
572     static char nom[MAX_NOM];
573     int statut;
574     char jeton = J1_JETON;
575     int njeueur;
576
577     initialise_grille();
578     affiche_grille();
579     njeueur = demande_nb_joueur();
580
581     if (!njeueur)
582         return EXIT_FAILURE;
583
584     while (1)
585     {
586         struct position pos;
587         int action;
588         int coup;
589
590         if (njeueur == 1 && jeton == J2_JETON)
591         {
592             coup = ia();
593             printf("Joueur 2 : %d\n", coup + 1);
594             calcule_position(coup, &pos);
595         }
596         else
597         {
598             printf("Joueur %d : ", (jeton == J1_JETON) ? 1 : 2);
599             action = demande_action(&coup);
600
601             if (action == ACT_ERR)
602                 return EXIT_FAILURE;
603             else if (action == ACT_QUITTER)
604                 return 0;
605             else if (action == ACT_SAUVEGARDER)
606             {
607                 if (!demande_fichier(nom, sizeof nom) || !sauvegarder_jeu(nom, jeton, njeueur))
608                     return EXIT_FAILURE;
609                 else
610                     return 0;
611             }
612             else if (action == ACT_CHARGER)
613             {
614                 if (!demande_fichier(nom, sizeof nom) || !charger_jeu(nom, &jeton, &njeueur))
615                     return EXIT_FAILURE;
616                 else
617                 {
618                     affiche_grille();
619                     continue;
620                 }
621             }
622             else if (action == ACT_NOUVELLE_SAISIE || !coup_valide(coup))
623             {
624                 fprintf(stderr, "Vous ne pouvez pas jouer à cet endroit\n");
625                 continue;
626             }
627
628             calcule_position(coup - 1, &pos);
629         }
630
631         grille[pos.colonne][pos.ligne] = jeton;

```

```
632     affiche_grille();
633     statut = statut_jeu(&pos, jeton);
634
635     if (statut != STATUT_OK)
636         break;
637
638     jeton = (jeton == J1_JETON) ? J2_JETON : J1_JETON;
639 }
640
641 if (statut == STATUT_GAGNE)
642     printf("Le joueur %d a gagné\n", (jeton == J1_JETON) ? 1 : 2);
643 else if (statut == STATUT_EGALITE)
644     printf("Égalité\n");
645
646 return 0;
647 }
```

À présent, durant la partie, un joueur peut entrer la lettre « s » (en minuscule ou en majuscule) afin d'effectuer une sauvegarde de la partie courante au sein d'un fichier qu'il doit spécifier. Les données sont sauvegardées sous forme de texte avec, dans l'ordre : le joueur courant, le nombre de joueurs et le contenu de la grille. Une fois la sauvegarde effectuée, le jeu est stoppé.

Il peut également entrer la lettre « c » (de nouveau en minuscule ou en majuscule) pour charger une partie précédemment sauvegardée.

Dans le chapitre suivant, nous verrons comment rendre nos programmes plus robustes en approfondissant la gestion d'erreur.

Jusqu'à présent, nous vous avons présenté les bases de la gestion d'erreurs en vous parlant des retours de fonctions et de la variable globale `errno`. Dans ce chapitre, nous allons approfondir cette notion afin de réaliser des programmes plus robustes.

25.1 Gestion de ressources

25.1.1 Allocation de ressources

Dans cette deuxième partie, nous avons découvert l'allocation dynamique de mémoire et la gestion de fichiers. Ces deux sujets ont un point en commun : il est nécessaire de passer par une fonction d'allocation et par une fonction de libération lors de leur utilisation. Il s'agit d'un processus commun en programmation appelé la **gestion de ressources**. Or, celle-ci pose un problème particulier dans le cas de la gestion d'erreurs. En effet, regardez de plus près ce code simplifié permettant l'allocation dynamique d'un tableau multidimensionnel de trois fois trois `int`.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4
5  int main(void)
6  {
7      int **p;
8      unsigned i;
9
10     p = malloc(3 * sizeof(int *));
11
12     if (p == NULL)
13     {
14         fprintf(stderr, "Échec de l'allocation\n");
15         return EXIT_FAILURE;
16     }
17
18     for (i = 0; i < 3; ++i)
19     {
20         p[i] = malloc(3 * sizeof(int));
21
22         if (p[i] == NULL)
23         {
24             fprintf(stderr, "Échec de l'allocation\n");
25             return EXIT_FAILURE;
26         }
27     }
28
29     /* ... */
```

```
30     return 0;
31 }
```

Vous ne voyez rien d'anormal dans le cas de la seconde boucle ? Celle-ci quitte le programme en cas d'échec de la fonction `malloc()`. Oui, mais nous avons *déjà* fait appel à la fonction `malloc()` auparavant, ce qui signifie que si nous quittons le programme à ce stade, nous le ferons *sans avoir libéré certaines ressources* (ici de la mémoire).

Seulement voilà, comment faire cela sans rendre le programme bien plus compliqué et bien moins lisible ? C'est ici que l'utilisation de l'instruction `goto` devient pertinente et d'une aide précieuse. La technique consiste à placer la libération de ressources en fin de fonction et de se rendre au bon endroit de cette zone de libération à l'aide de l'instruction `goto`. Autrement dit, voici ce que cela donne avec le code précédent.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4
5  int main(void)
6  {
7      int **p;
8      unsigned i;
9      int status = EXIT_FAILURE;
10
11     p = malloc(3 * sizeof(int *));
12
13     if (p == NULL)
14     {
15         fprintf(stderr, "Échec de l'allocation\n");
16         goto fin;
17     }
18
19     for (i = 0; i < 3; ++i)
20     {
21         p[i] = malloc(3 * sizeof(int));
22
23         if (p[i] == NULL)
24         {
25             fprintf(stderr, "Échec de l'allocation\n");
26             goto liberer_p;
27         }
28     }
29
30     status = 0;
31
32     /* Zone de libération */
33 liberer_p:
34     while (i > 0)
35     {
36         --i;
37         free(p[i]);
38     }
39
40     free(p);
41 fin:
42     return status;
43 }
```

Comme vous le voyez, nous avons placé deux étiquettes : une référençant l'instruction `return` et une autre désignant la première instruction de la zone de libération. Ainsi, nous quittons la fonction en cas d'échec de la première allocation, mais nous libérons les ressources auparavant dans le cas des allocations suivantes. Notez que nous avons ajouté une variable `status` afin de pouvoir retourner la bonne valeur suivant qu'une erreur est survenue ou non.

25.1.2 Utilisation de ressources

Si le problème se pose dans le cas de l'allocation de ressources, il se pose également dans le cas de leur utilisation.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4
5  int main(void)
6  {
7      FILE *fp;
8
9      fp = fopen("texte.txt", "w");
10
11     if (fp == NULL)
12     {
13         fprintf(stderr, "Le fichier texte.txt n'a pas pu être ouvert\n");
14         return EXIT_FAILURE;
15     }
16     if (fputs("Au revoir !\n", fp) == EOF)
17     {
18         fprintf(stderr, "Erreur lors de l'écriture d'une ligne\n");
19         return EXIT_FAILURE;
20     }
21     if (fclose(fp) == EOF)
22     {
23         fprintf(stderr, "Erreur lors de la fermeture du flux\n");
24         return EXIT_FAILURE;
25     }
26
27     return 0;
28 }
```

Dans le code ci-dessus, l'exécution du programme est stoppée en cas d'erreur de la fonction `fputs()`. Cependant, l'arrêt s'effectue alors qu'une ressource n'a pas été libérée (en l'occurrence le flux `fp`). Aussi, il est nécessaire d'appliquer la solution présentée juste avant pour rendre ce code correct.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4
5  int main(void)
6  {
7      FILE *fp;
8      int status = EXIT_FAILURE;
9
10     fp = fopen("texte.txt", "w");
11
12     if (fp == NULL)
13     {
14         fprintf(stderr, "Le fichier texte.txt n'a pas pu être ouvert\n");
15         goto fin;
16     }
17     if (fputs("Au revoir !\n", fp) == EOF)
18     {
19         fprintf(stderr, "Erreur lors de l'écriture d'une ligne\n");
20         goto fermer_flux;
21     }
22     if (fclose(fp) == EOF)
23     {
24         fprintf(stderr, "Erreur lors de la fermeture du flux\n");
25         goto fin;
26     }
27
28     status = 0;
29
30     fermer_flux:
31     fclose(fp);
```

```

32 fin:
33     return status;
34 }

```

Nous attirons votre attention sur deux choses :

- En cas d'échec de la fonction `fclose()`, le programme est arrêté immédiatement. Étant donné que c'est la fonction `fclose()` qui pose problème, il n'y a pas lieu de la rappeler (notez que cela n'est toutefois pas une erreur de l'appeler une seconde fois).
- Le retour de la fonction `fclose()` n'est pas vérifié en cas d'échec de la fonction `fputs()` étant donné que nous sommes *déjà* dans un cas d'erreur.

25.2 Fin d'un programme

Vous le savez, l'exécution de votre programme se termine lorsque celui-ci quitte la fonction `main()`. Toutefois, que se passe-t-il exactement lorsque celui-ci s'arrête ? En fait, un petit bout de code appelé **épilogue** est exécuté afin de réaliser quelques tâches. Parmi celles-ci figure la vidange et la fermeture de tous les flux encore ouvert. Une fois celui-ci exécuté, la main est rendue au système d'exploitation qui, le plus souvent, se chargera de libérer toutes les ressources qui étaient encore allouées.



Mais ?! Vous venez de nous dire qu'il était nécessaire de libérer les ressources avant l'arrêt du programme. Du coup, pourquoi s'amuser à appeler les fonctions `fclose()` ou `free()` alors que l'épilogue ou le système d'exploitation s'en charge ?

Pour quatre raisons principales :

1. Afin de libérer des ressources pour les autres programmes. En effet, si vous ne libérez les ressources allouées qu'à la fin de votre programme alors que celui-ci n'en a plus besoin, vous gaspillez des ressources qui pourraient être utilisées par d'autres programmes.
2. Pour être à même de détecter des erreurs, de prévenir l'utilisateur en conséquence et d'éventuellement lui proposer des solutions.
3. En vue de rendre votre code plus facilement modifiable par après et d'éviter les oublis.
4. Parce qu'il n'est pas garanti que les ressources seront effectivement libérées par le système d'exploitaion.



Aussi, de manière générale :

- considérez que l'objectif de l'épilogue est de fermer *uniquement* les flux `stdin`, `stdout` et `stderr` ;
- ne comptez *pas* sur votre système d'exploitation pour la libération de quelques ressources que ce soit (et notamment la mémoire) ;
- libérez vos ressources le plus tôt possible, autrement dit dès que votre programme n'en a plus l'utilité.

25.2.1 Terminaison normale

Le passage par l'épilogue est appelée la **terminaison normale** du programme. Il est possible de s'y rendre directement en appelant la fonction `exit()` qui est déclarée dans l'en-tête `<stdlib.h>`.

```

1 void exit(int status);

```

Appeler cette fonction revient au même que de quitter la fonction `main()` à l'aide de l'instruction `return`. L'argument attendu est une expression entière identique à celle fournie comme opérande de l'instruction `return`. Ainsi, il vous est possible de mettre fin directement à l'exécution de votre programme *sans* retourner jusqu'à la fonction `main()`.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4
5  void fin(void)
6  {
7      printf("C'est la fin du programme\n");
8      exit(0);
9  }
10
11
12  int main(void)
13  {
14      fin();
15      printf("Retour à la fonction main\n");
16      return 0;
17  }
```

```

1  C'est la fin du programme
```

Comme vous le voyez l'exécution du programme se termine une fois que la fonction `exit()` est appelée. La fin de la fonction `main()` n'est donc *jamais* exécutée.

25.2.2 Terminaison anormale

Il est possible de terminer l'exécution d'un programme *sans passer par l'épilogue* à l'aide de la fonction `abort()` (définie également dans l'en-tête `<stdlib.h>`). Dans un tel cas, il s'agit d'une **terminaison anormale** du programme.

```

1  void abort(void);
```

Une terminaison anormale signifie qu'une condition *non prévue* par le programmeur est survenue. Elle est donc à distinguer d'une terminaison normale survenue suite à une erreur qui, elle, était prévue (comme une erreur lors d'une saisie de l'utilisateur). De manière générale, la terminaison anormale est utilisée lors de la phase de développement d'un logiciel afin de faciliter la détection d'erreurs de programmation, celle-ci entraînant le plus souvent la production d'une **image mémoire** (*core dump* en anglais) qui pourra être analysée à l'aide d'un **débogueur** (*debugger* en anglais).



Une image mémoire est en fait un fichier contenant l'état des registres et de la mémoire d'un programme lors de la survenance d'un problème.

25.3 Les assertions

25.3.1 La macrofonction assert

Une des utilisation majeure de la fonction `abort()` est réalisée via la macrofonction `assert()` (définie dans l'en-tête `<assert.h>`). Cette dernière est utilisée pour placer des tests à certains points d'un programme. Dans le cas où un de ces tests s'avère faux, un message d'erreur est affiché (ce dernier comprend la condition dont l'évaluation est fausse, le nom du fichier et la ligne courante) après quoi la fonction `abort()` est appelée afin de produire une image mémoire.

Cette technique est très utile pour détecter rapidement des erreurs au sein d'un programme lors de sa phase de développement. Généralement, les assertions sont placées en début de fonction afin de vérifier un certains nombres de conditions. Par exemple, si nous reprenons la fonction `long_colonne()` du TP sur le Puissance 4.

```

1 static unsigned long_colonne(unsigned joueur, unsigned col, unsigned ligne, unsigned char (*grille)[6])
2 {
3     unsigned i;
4     unsigned n = 1;
5
6     for (i = ligne + 1; i < 6; ++i)
7     {
8         if (grille[col][i] == joueur)
9             ++n;
10        else
11            break;
12    }
13
14    return n;
15 }
```

Nous pourrions ajouter quatre assertions vérifiant si :

- le numéro du joueur est un ou deux ;
- le numéro de la colonne est compris entre zéro et six inclus ;
- le numéro de la ligne est compris entre zéro et cinq ;
- le pointeur `grille` n'est pas nul.

Ce qui nous donne le code suivant.

```

1 static unsigned long_colonne(unsigned joueur, unsigned col, unsigned ligne, unsigned char (*grille)[6])
2 {
3     unsigned i;
4     unsigned n = 1;
5
6     assert(joueur == 1 || joueur == 2);
7     assert(col < 7);
8     assert(ligne < 6);
9     assert(grille != NULL);
10
11    for (i = ligne + 1; i < 6; ++i)
12    {
13        if (grille[col][i] == joueur)
14            ++n;
15        else
16            break;
17    }
18
19    return n;
20 }
```

Ainsi, si nous ne respectons pas une de ces conditions pour une raison ou pour une autre, l'exécution du programme s'arrêtera et nous aurons droit à un message du type (dans le cas où la première assertion échoue).

```

1 a.out: main.c:71: long_colonne: Assertion 'joueur == 1 || joueur == 2' failed.
2 Aborted
```

Comme vous le voyez, le nom du programme est indiqué, suivi du nom du fichier, du numéro de ligne, du nom de la fonction et de la condition qui a échoué. Intéressant, non ? :)

25.3.2 Suppression des assertions

Une fois votre programme développé et dûment testé, les assertions ne vous sont plus vraiment utiles étant donné que celui-ci fonctionne. Les conserver alourdirait l'exécution de votre

programme en ajoutant des vérifications. Toutefois, heureusement, il est possible de supprimer ces assertions *sans modifier votre code* en ajoutant simplement l'option `-DNDEBUG` lors de la compilation.

```
1 $ zcc -DNDEBUG main.c
```

25.4 Les fonctions `strerror` et `perror`

Vous le savez, certaines (beaucoup) de fonctions standards peuvent échouer. Toutefois, en plus de signaler à l'utilisateur qu'une de ces fonctions a échoué, cela serait bien de lui spécifier *pourquoi*. C'est ici qu'entre en jeu les fonctions `strerror()` et `perror()`.

```
1 char *strerror(int num);
```

La fonction `strerror()` (déclarée dans l'en-tête `<string.h>`) retourne une chaîne de caractères correspondant à la valeur entière fournie en argument. Cette valeur sera en fait *toujours* celle de la variable `errno`. Ainsi, il vous est possible d'obtenir plus de détails quand une fonction standard rencontre un problème. L'exemple suivant illustre l'utilisation de cette fonction en faisant appel à la fonction `fopen()` afin d'ouvrir un fichier qui n'existe pas (ce qui provoque une erreur).

```
1 #include <errno.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5
6
7 int main(void)
8 {
9     FILE *fp;
10
11     fp = fopen("nawak.txt", "r");
12
13     if (fp == NULL)
14     {
15         fprintf(stderr, "fopen: %s\n", strerror(errno));
16         return EXIT_FAILURE;
17     }
18
19     fclose(fp);
20     return 0;
21 }
```

```
1 fopen: No such file or directory
```

Comme vous le voyez, nous avons obtenu des informations supplémentaires : la fonction a échoué parce que le fichier « `nawak.txt` » n'existe pas.



Notez que les messages d'erreur retournés par la fonction `strerror()` sont le plus souvent en anglais.

```
1 void perror(char *message);
```

La fonction `perror()` (déclarée dans l'en-tête `<stdio.h>`) écrit sur le flux d'erreur standard (`stderr`, donc) la chaîne de caractères fournie en argument, suivie du caractère `:`, d'un espace et du retour de la fonction `strerror()` avec comme argument la valeur de la variable `errno`. Autrement dit, cette fonction revient au même que l'appel suivant.


```
1 fprintf(stderr, "message: %s\n", strerror(errno));
```

L'exemple précédent peut donc également s'écrire comme suit.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4
5  int main(void)
6  {
7      FILE *fp;
8
9      fp = fopen("nawak.txt", "r");
10
11     if (fp == NULL)
12     {
13         perror("fopen");
14         return EXIT_FAILURE;
15     }
16
17     fclose(fp);
18     return 0;
```

```
1  fopen: No such file or directory
```

Voici qui clôture la deuxième partie de ce cours qui aura été riche en nouvelles notions. N'hésitez pas à reprendre certains passages avant de commencer la troisième partie qui vous fera plonger sous le capot.

Troisième partie

Notions avancées

La représentation des types

Dans le chapitre sur l'allocation dynamique, nous avons effleuré la notion de représentation en parlant de celle d'objet. Pour rappel, la représentation d'un type correspond à la manière dont les données sont réparties en mémoire, plus précisément comment les multipliets et les *bits* les composant sont agencés et utilisés.

Dans ce chapitre, nous allons plonger au coeur de ce concept et vous exposer la représentation des types.

26.1 La représentation des entiers

26.1.1 Les entiers non signés

La représentation des entiers non signés étant la plus simple, nous allons commencer par celle-ci. :)

Dans un entier non signé, les différents *bits* correspondent à une puissance de deux. Plus précisément, le premier correspond à 2^0 , le second à 2^1 , le troisième à 2^2 et ainsi de suite jusqu'au dernier *bit* composant le type utilisé. Pour calculer la valeur d'un entier non signé, il suffit donc d'additionner les puissances de deux correspondant aux bits à 1.

Pour illustrer notre propos, voici un tableau comprenant la représentation de plusieurs nombres au sein d'un objet de type `unsigned char` (ici sur un octet).

Nombre	Représentation							
1	0	0	0	0	0	0	0	1
	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	$2^0=1$							
3	0	0	0	0	0	0	1	1
	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	$2^0+2^1=3$							
42	0	0	1	0	1	0	1	0
	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	$2^1+2^3+2^5=42$							
255	1	1	1	1	1	1	1	1
	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	$2^0+2^1+2^2+2^3+2^4+2^5+2^6+2^7=255$							

Nombre	Représentation en signe et magnitude							
1	0	0	0	0	0	0	0	1
	+	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	$2^0=1$							
1	0	0	0	0	0	0	0	1
	-	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	$-2^0=-1$							

Nombre	Représentation en complément à un							
1	0	0	0	0	0	0	0	1
	+	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	$2^0=1$							
-1	1	1	1	1	1	1	1	0
	$-/2^7$	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	$-(2^7-2^6-2^5-2^4-2^3-2^2-2^1)=-1$							

26.1.2 Les entiers signés

Les choses se compliquent un peu avec les entiers signés. En effet, il nous faut représenter une information supplémentaire : le signe de la valeur.

La représentation en signe et magnitude

La première solution qui vous est peut-être venue à l'esprit serait de réserver un *bit*, par exemple le dernier, pour représenter le signe. Ainsi, la représentation est identique à celle des entiers non signés si ce n'est que le dernier *bit* est réservé pour le signe (ce qui diminue donc en conséquence la valeur maximale représentable). Cette méthode est appelée **représentation en signe et magnitude**.

Cependant, si la représentation en signe et magnitude paraît simple, elle n'est en vérité pas facile à mettre en œuvre au sein d'un processeur car elle implique plusieurs vérifications (notamment au niveau du signe) qui se traduisent par des circuits supplémentaires et un surcoût en calcul. De plus, elle laisse deux représentations possibles pour le zéro (`0000 0000` et `1000 0000`), ce qui est gênant pour l'évaluation des conditions.

La représentation en complément à un

Dès lors, comment pourrions nous faire pour simplifier nos calculs en évitant des vérifications liées au signe ? Eh bien, sachant que le maximum représentable dans notre exemple est `255` (soit `1111 1111`), il nous est possible de représenter chaque nombre négatif comme la différence entre le maximum et sa valeur absolue. Par exemple `-1` sera représenté par `255 - 1`, soit `254` (`1111 1110`). Cette représentation est appelée **représentation en complément à un**.

Ainsi, si nous additionnons `1` et `-1`, nous n'avons pas de vérifications à faire et obtenons le maximum, `255`. Toutefois, ceci implique, comme pour la représentation en signe et magnitude, qu'il existe deux représentations pour le zéro : `0000 0000` et `1111 1111`.



Notez qu'en fait, cette représentation revient à inverser tous les *bits* d'un nombre positif pour obtenir son équivalent négatif. Par exemple, si nous inversons `1` (`0000 0001`), nous obtenons bien `-1` (`1111 1110`) comme ci-dessus.

Par ailleurs, il subsiste un second problème : dans le cas où deux nombres négatifs sont additionnés, le résultat obtenu n'est pas valide. En effet, `-1 + -1` nous donne `1111 1110 + 1111 1110`

Nombre	Représentation en complément à deux							
1	0	0	0	0	0	0	0	1
	+	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	$2^0=1$							
-1	1	1	1	1	1	1	1	0
	$-/2^7$	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	$-(256-2^7-2^6-2^5-2^4-2^3-2^2-2^1-2^0)=-1$							

soit `1 1111 1100` ; autrement dit, comme nous travaillons sur huit *bits* pour nos exemples, `-3`... Mince !

Pour résoudre ce problème, il nous faut reporter la dernière retenue (soit ici le dernier *bit* que nous avons ignoré) au début (ce qui revient à ajouté un) ce qui nous permet d'obtenir `1111 1101`, soit `-2`.

La représentation en complément à deux

Ainsi est apparue une troisième représentation : celle en **complément à deux**. Celle-ci conserve les qualités de la représentation en complément à un, mais lui corrige certains défauts. En fait, elle représente les nombres négatifs de la même manière que la représentation en complément à un, si ce n'est que la soustraction s'effectue entre le maximum augmenté de un (soit `256` dans notre cas) et non le maximum.

Ainsi, par exemple, la représentation en complément à deux de `-1` est `256 - 1`, soit `255` (`1111 1111`).



Remarquez que cette représentation revient finalement à inverser tous les *bits* d'un nombre positif et à augmenter le résultat de un en vue d'obtenir son équivalent négatif. Par exemple, si nous inversons les *bits* du nombre `1` nous obtenons `1111 1110` et si nous lui ajoutons un, nous avons bien `1111 1111`.

Cette fois ci, l'objectif recherché est atteint ! :)

En effet, si nous additionnons `1` et `-1` (soit `0000 0001` et `1111 1111`) et que nous ignorons la retenue, nous obtenons bien zéro. De plus, il n'y a plus de cas particulier de retenue à déplacer comme en complément à un, puisque, par exemple, la somme `-1 + -2`, soit `1111 1111 + 1111 1110` donne `1 1111 1101`, autrement dit `-3` *sans la retenue*. Enfin, nous n'avons plus qu'une seule représentation pour le zéro.



Point de norme (1) La norme ^a du langage C ne précise pas quelle représentation doit être utilisée pour les entiers signés. Elle impose uniquement qu'il s'agisse d'une de ces trois. Cependant, dans les faits, il s'agit presque toujours de la représentation en complément à deux.

a. la norme C89 ne précise pas les représentations possibles pour les types entiers (Programming Language C, X3J11/88-090, § A.6.3.4, Integers, al. 1), mais les normes suivantes bien. Voyez à cet égard : ISO/IEC 9899 :TC3, doc. [N1256](#), § 6.2.6.2, Integer types, p. 38 et ISO/IEC 9899 :201x, doc. [N1570](#), § 6.2.6.2, Integer types, p. 45.



Point de norme (2) Notez que chacune de ces représentations disposent toutefois d'une suite de *bits* qui est susceptible de ne pas représenter une valeur et de produire une erreur en cas d'utilisation ^a.

— Pour les représentations en signe et magnitude et en complément à deux, il s'agit de la suite où tous les *bits* sont à zéro et le *bit* de signe à un : `1000 0000` ;

- Pour la représentation en complément à un, il s'agit de la suite où tous les *bits* sont à un, y compris le *bit* de signe : `1111 1111`.

Dans le cas des représentations en signe et magnitude et en complément à un, il s'agit des représentations possibles pour le « zéro négatif ». Pour la représentation en complément à deux, cette suite est le plus souvent utilisée pour représenter un nombre négatif supplémentaire (dans le cas de `1000 0000`, il s'agira de `-128`). Toutefois, même si ces représentations peuvent être utilisées pour représenter une valeur valide, ce n'est pas forcément le cas.



Néanmoins, rassurez-vous, ces valeurs ne peuvent être produites dans le cas de calculs ordinaires, sauf survenance d'une condition exceptionnelle comme un dépassement de capacité (nous en parlerons bientôt). Vous n'avez donc pas de vérifications supplémentaires à effectuer à leur égard. Évitez simplement d'en produire une délibérément, par exemple en l'affectant directement à une variable.

a. La norme C89 est muette sur ce point, mais les normes suivantes sont plus loquaces : ISO/IEC 9899 :TC3, doc. [N1256](#), § 6.2.6.1 , General, al. 5, pp. 37-38 et ISO/IEC 9899 :201x, doc. [N1570](#), § 6.2.6.1, General, al. 5, p. 44.

26.1.3 Les bits de bourrages

Il est important de préciser que tous les *bits* composant un type entier ne sont pas forcément utilisés pour représenter la valeur qui y est stockée. Nous l'avons vu avec le *bit* de signe, qui ne correspond pas à une valeur. La norme prévoit également qu'il peut exister des *bits* de bourrages, et ce *aussi bien pour les entiers signés que pour les entiers non signés* à l'exception du type `char`. Ceux-ci peuvent par exemple être employés pour maintenir d'autres informations (par exemple : le type de la donnée stockée, ou encore un *bit de parité* pour vérifier l'intégrité de celle-ci).



Par conséquent, il n'y a pas forcément une corrélation parfaite entre le nombre de *bits* composant un type entier et la valeur maximale qui peut y être stockée.



Sachez toutefois que les *bits* de bourrages au sein des types entiers sont assez rares, les architectures les plus courantes n'en emploient pas.

26.2 La représentation des flottants

La représentation des types flottants amène deux difficultés supplémentaires :

- la gestion de nombres réels, c'est-à-dire potentiellement composés d'une partie entière et d'une suite de décimales ;
- la possibilité de stocker des nombres de différents ordres de grandeur, entre 10^{-37} et 10^{37} .

26.2.1 Première approche

Avec des entiers

Une première solution consisterait à garantir une précision à 10^{-37} en utilisant deux nombres entiers : un, signé, pour la partie entière et un, non signé, pour stocker la suite de décimales.

Cependant, si cette approche a le mérite d'être simple, elle a le désavantage d'utiliser *beaucoup* de mémoire. En effet, pour stocker un entier de l'ordre de 10^{37} , il serait nécessaire d'utiliser un peu moins de 128 *bits*, soit 16 octets (et donc environ 32 octets pour la représentation globale). Un tel coût est inconcevable, même à l'époque actuelle.

Nombre	Représentation							
1	0	0	0	1	0	0	0	0
	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}
	$2^0=1$							
1,5	0	0	0	1	1	0	0	0
	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}
	$2^{-1}+2^0=1,5$							
0,875	0	0	0	1	1	1	1	0
	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}
	$2^{-3}+2^{-2}+2^{-1}=0,875$							

Une autre limite provient de la difficulté d'effectuer des calculs sur les nombres flottants avec une telle représentation : il faudrait tenir compte du passage des décimales vers la partie entière et inversement.

Avec des puissances de deux négatives

Une autre représentation possible consiste à attribuer des puissances de deux *négatives* à une partie des *bits*. Les calculs sur les nombres flottants obtenus de cette manière sont similaires à ceux sur les nombres entiers. Le tableau ci-dessous illustre ce concept en divisant un octet en deux : quatre *bits* pour la partie entière et quatre *bits* pour la partie fractionnaire.

Toutefois, notre problème de capacité persiste : il nous faudra toujours une grande quantité de mémoire pour pouvoir stocker des nombres d'ordre de grandeur aussi différents.

26.2.2 Une histoire de virgule flottante

Par conséquent, la solution qui a été retenue historiquement consiste à réserver quelques *bits* qui contiendront la valeur d'un exposant. Celui-ci sera ensuite utilisé pour déterminer à quelles puissances de deux correspondent les *bits* restants. Ainsi, la virgule séparant la partie entière de sa suite décimales est dite « flottante », car sa position est ajustée par l'exposant. Toutefois, comme nous allons le voir, ce gain en mémoire ne se réalise pas sans sacrifice : la précision des calculs va en pâtir.

Le tableau suivant utilise deux octets : un pour stocker un exposant sur sept *bits* avec un *bit* de signe ; un autre pour stocker la valeur du nombre. L'exposant est lui aussi signé et est représenté en signe et magnitude (par souci de facilité). Par ailleurs, cet exposant est attribué au premier *bit* du deuxième octet ; les autres correspondent à une puissance de deux chaque fois inférieure d'une unité.

Les quatre premiers exemples n'amènent normalement pas de commentaires particuliers ; néanmoins, les deux derniers illustrent deux problèmes posés par les nombres flottants.

Les approximations

Les nombres réels mettent en évidence la difficulté (voir l'impossibilité) de représenter une suite de décimales à l'aide d'une somme de puissances de deux. En effet, le plus souvent, les valeurs ne peuvent être qu'approchées et obtenues suite à des arrondis. Dans le cas de 0,00001, la somme $2^{-24} + 2^{-23} + 2^{-22} + 2^{-20} + 2^{-19} + 2^{-17}$ donne en vérité 0,000010907649993896484375 qui, une fois arrondie, donnera bien 0,00001.

Les pertes de précision

Dans le cas où la valeur à représenter ne peut l'être entièrement avec le nombre de *bits* disponible (autrement dit, le nombre de puissances de deux disponible), la partie la moins

Nombre	Signe	Exposant							Bits du nombre							
1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
	+	+	2^5	2^4	2^3	2^2	2^1	2^0	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}
	+	0							$2^0=1$							
-1	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
	-	+	2^5	2^4	2^3	2^2	2^1	2^0	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}
	-	0							$-2^0=-1$							
0,5	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
	+	+	2^5	2^4	2^3	2^2	2^1	2^0	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}
	+	0							$2^{-1}=0,5$							
10	0	0	0	0	0	1	1	1	1	0	1	0	0	0	0	0
	+	+	2^5	2^4	2^3	2^2	2^1	2^0	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}
	+	$2^1+2^0=3$							$2^3+2^1=10$							
0,00001	0	1	0	1	0	0	1	0	1	0	1	1	0	1	1	1
	+	-	2^5	2^4	2^3	2^2	2^1	2^0	2^{-17}	2^{-18}	2^{-19}	2^{-20}	2^{-21}	2^{-22}	2^{-23}	2^{-24}
	+	$-2^1+2^4=-17$							$2^{-24}+2^{-23}+2^{-22}+2^{-21}+2^{-20}+2^{-19}+2^{-18}+2^{-17}=0,00001$							
32769	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0
	+	+	2^5	2^4	2^3	2^2	2^1	2^0	2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^9	2^8
	+	$2^0+2^1+2^2+2^3=15$							$2^{15}=32768$							

significative sera abandonnée et, corrélativement, de la précision.

Ainsi, dans notre exemple, si nous souhaitons représenter le nombre 32769, nous avons besoin d'un exposant de 15 afin d'obtenir 32768. Toutefois, vu que nous ne possédons que de 8 *bits* pour représenter notre nombre, il nous est impossible d'attribuer l'exposant 0 à un des *bits*. Cette information est donc perdue et seule la valeur la plus significative (ici 32768) est conservée.

26.2.3 Formalisation

L'exemple simplifié que nous vous avons montré illustre dans les grandes lignes la représentation des nombres flottants. Cependant, vous vous en doutez, la réalité est un peu plus complexe. De manière plus générale, un nombre flottant est représenté à l'aide : d'un signe, d'un exposant et d'une **mantisse** qui est en fait la suite de puissances qui sera additionnée.

Par ailleurs, nous avons utilisés une suite de puissance de deux, mais il est parfaitement possible d'utiliser une autre base. Beaucoup de calculatrices utilisent par exemple des suites de puissances de dix et non de deux.



À l'heure actuelle, les nombres flottants sont presque toujours représentés suivant la norme [IEEE 754](#) qui utilise une représentation en base deux.

26.3 La représentation des pointeurs

Cet extrait sera relativement court : la représentation des pointeurs est indéterminée. Sur la plupart des architectures, il s'agit en vérité d'entiers non signés, mais il n'y a absolument aucune garantie à ce sujet.

26.4 Ordre des multipléts et des bits

Jusqu'à présent, nous vous avons montré les représentations binaires en ordonnant les *bits* et les multipléts par puissance de deux croissantes. Cependant, s'il s'agit d'une représentation

Nombre	Représentation gros-boutiste															
1	Octet de poids fort								Octet de poids faible							
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
	2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
$2^0=1$																

Nombre	Représentation petit-boutiste															
1	Octet de poids faible								Octet de poids fort							
	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^9	2^8
$2^0=1$																

possible, elle n'est pas la seule. En vérité, l'ordre des multipliets et des *bits* peut varier d'une machine à l'autre.

26.4.1 Ordre des multipliets

Dans le cas où un type est composé de plus d'un multipliet (ce qui, à l'exception du type `char`, est pour ainsi dire toujours le cas), ceux-ci peuvent être agencés de différentes manières. L'ordre des multipliets d'une machine est appelé son **boutisme** (*endianness* en anglais).

Il en existe principalement deux : le gros-boutisme et le petit-boutisme.

Le gros-boutisme

Sur une architecture gros-boutiste, les multipliets sont ordonnés par **poids** décroissant.

Le poids d'un *bit* se détermine suivant la puissance de deux qu'il représente : plus elle est élevée, plus le *bit* a un poids important. Le *bit* représentant la puissance de deux la plus basse est appelé de *bit* de **poids faible** et celui correspondant à la puissance la plus grande est nommé *bit* de **poids fort**. Pour les multipliets, le raisonnement est identique : son poids est déterminé par celui des *bits* le composant.

Autrement dit, une machine gros-boutiste place les multipliets comprenant les puissance de deux les plus élevées en premier (nos exemple précédents utilisaient donc cette représentation).

Le petit-boutisme

Une architecture petit-boutiste fonctionne de manière inverse : les multipliets sont ordonnés par poids croissant.

Ainsi, si nous souhaitons stocker le nombre `1` dans une variable de type `unsigned short` (qui sera, pour notre exemple, d'une taille de deux octets), nous aurons les deux résultats suivants, selon le boutisme de la machine.



Le boutisme est relativement transparent du point de vue du programmeur, puisqu'il s'agit d'une propriété de la mémoire. En pratique, de tels problèmes ne se posent que lorsqu'on accède à la mémoire à travers des types différents de celui de l'objet stocké initialement (par exemple via un pointeur sur `char`). En particulier, la communication entre plusieurs machines doit les prendre en compte.

26.4.2 Ordre des bits

Cependant, ce serait trop simple si le problème en restait là... :-°
Malheureusement, l'ordre des *bits* varie également d'une machine à l'autre et ce, *indépendamment de l'ordre des multipliets*. Comme pour les multipliets, il existe différentes possibilités, mais les

Nombre	Représentation gros-boutiste pour les multipléts et petit-boutiste pour les <i>bits</i>															
1	Octet de poids fort								Octet de poids faible							
	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7
	$2^0=1$															

Nombre	Représentation petit-boutiste pour les multipléts et petit-boutiste pour les <i>bits</i>															
1	Octet de poids faible								Octet de poids fort							
	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}
	$2^0=1$															

plus courante sont le gros-boutisme et le petit-boutisme. Ainsi, il est par exemple possible que la représentation des multipléts soit gros-boutiste, mais que celle des *bits* soit petit-boutiste (et inversement).

Notez toutefois que, le plus souvent, une représentation petit-boutiste ou gros-boutiste s'applique aussi bien aux octets qu'aux *bits*.



| Par ailleurs, sachez qu'à quelques exceptions près, l'ordre du stockage des *bits* n'apparaît pas en C. En particulier, les opérateurs de manipulation de *bits*, vus au chapitre suivant, n'en dépendent pas.

26.4.3 Applications

Connaître le boutisme d'une machine

Le plus souvent, il est possible de connaître le boutisme employé pour les *multipléts* à l'aide du code suivant. Ce dernier stocke la valeur 1 dans une variable de type `unsigned short`, et accède à son premier octet à l'aide d'un pointeur sur `unsigned char`. Si celui-ci est nul, c'est que la machine est gros-boutiste, sinon, c'est qu'elle est petit-boutiste.

```

1  #include <stdio.h>
2
3
4  int main(void)
5  {
6      unsigned short n = 1;
7      unsigned char *p = (unsigned char *)&n;
8
9      if (*p != 0)
10         printf("Je suis petit-boutiste.\n");
11     else
12         printf("Je suis gros-boutiste.\n");
13
14     return 0;
15 }
```



Notez bien que cette technique n'est pas entièrement fiable. En effet, rappelez-vous : d'une part, les deux boutismes présentés, s'ils sont les plus fréquents, ne sont pas les seuls et, d'autre part, la présence de *bits* de bourrage au sein du type entier, bien que rare, pourrait fausser le résultat.

Le boutisme des constantes octales et hexadécimales

Suivant ce qui vient de vous être présenter, peut-être vous êtes vous posé la question suivante : si les boutismes varient d'une machine à l'autre, alors que vaut finalement la constante `0x01` ? En effet, suivant les boutismes employés, celle-ci peut valoir 1 ou 16.

Heureusement, cette question est réglée par la norme¹ : le boutisme ne change *rien* à l'écriture des constantes octales et hexadécimales, elle est toujours réalisée suivant la convention classique des chiffres de poids fort en premier. Ainsi, la constante `0x01` vaut 1 et la constante `0x8000` vaut 32768 (en non signé).

De même, les indicateurs de conversion `x`, `X` et `o` affichent toujours leurs résultats suivant cette écriture.

```
1 printf("%02x\n", 1);
```

```
1 01
```

26.5 Les fonctions memset, memcpy, memmove et memcmp

26.5.1 La fonction memset

```
1 void *memset(void *obj, int val, size_t taille);
```

La fonction `memset()` initialise les `taille` premiers multiplats de l'objet référencé par `obj` avec la valeur `val` (qui est convertie vers le type `unsigned char`). Cette fonction est très utile pour (ré)initialiser les différents éléments d'un agrégats sans devoir recourir à une boucle.

```
1 #include <stdio.h>
2 #include <string.h>
3
4
5 int main(void)
6 {
7     int tab[] = { 10, 20, 30, 40, 50 };
8     int i;
9
10    memset(tab, 0, sizeof tab);
11
12    for (i = 0; i < sizeof tab / sizeof tab[0]; ++i)
13        printf("%d : %d\n", i, tab[i]);
14
15    return 0;
16 }
```

```
1 0 : 0
2 1 : 0
3 2 : 0
4 3 : 0
5 4 : 0
```



Faites attention ! Manipuler ainsi les objets *byte* par *byte* nécessite de connaître leur représentation. Dans un souci de portabilité, on ne devrait donc pas utiliser `memset`.

1. Programming Language C, X3J11/88-090, § 3.1.3.2, Integer constants, al. 3 Pour terminer ce chapitre, il nous reste à vous présenter quatre fonctions que nous avons passé sous silence lors de notre présentation de l'en-tête `<string.h>` : `memset()`, `memcpy()`, `memmove()` et `memcmp()`. Bien qu'elles soient définies dans cet en-tête, ces fonctions ne sont pas véritablement liées aux chaînes de caractères et opèrent en vérité sur les multiplats composant les objets. De telles modifications impliquant la représentation des types, nous avons attendu ce chapitre pour vous en parler.



sur des pointeurs ou des flottants. De même, pour éviter d'obtenir les représentations problématiques dans le cas d'entiers signés, il est conseillé de n'employer cette fonction que pour mettre ces derniers à zéro.

26.5.2 Les fonctions memcpy et memmove

```
1 void *memcpy(void *destination, void *source, size_t taille);
2 void *memmove(void *destination, void *source, size_t taille);
```

Les fonction `memcpy()` et `memmove()` copient toutes deux les `taille` premiers multiplètes de l'objet pointé par `source` vers les premiers multiplètes de l'objet référencé par `destination`. La différence entre ces deux fonctions est que `memcpy()` ne doit être utilisée qu'avec deux objets qui ne se chevauchent pas (autrement dit, les deux pointeurs ne doivent pas accéder ou modifier une même zone mémoire ou une partie d'une même zone mémoire) alors que `memmove()` ne souffre pas de cette restriction.

```
1 #include <stdio.h>
2 #include <string.h>
3
4
5 int main(void)
6 {
7     int n = 10;
8     int m;
9
10    memcpy(&m, &n, sizeof n);
11    printf("m = n = %d\n", m);
12
13    memmove(&n, ((unsigned char *)&n) + 1, sizeof n - 1);
14    printf("n = %d\n", n);
15    return 0;
16 }
```

```
1 m = n = 10
2 n = 0
```

L'utilisation de `memmove()` amène quelques précisions : nous copions ici les trois derniers multiplètes de l'objet `n` vers ses trois premiers multiplètes. Étant donné que notre machine est petit-boutiste, les trois derniers multiplètes sont à zéro et la variable `n` a donc pour valeur finale zéro.

26.5.3 La fonction memcmp

```
1 int memcmp(void *obj1, void *obj2, size_t taille);
```

La fonction `memcmp()` compare les `taille` premiers multiplètes des objets référencés par `obj1` et `obj2` et retourne un nombre entier inférieur, égal ou supérieur à zéro suivant que la valeur du dernier multiplète comparé (convertie en `unsigned char`) de `obj1` est inférieure, égales ou supérieure à celle du dernier multiplète comparé de `obj2`.

```
1 #include <stdio.h>
2 #include <string.h>
3
4
5 int main(void)
6 {
7     int n = 10;
8     int m = 10;
```

```

9      int o = 20;
10
11     if (memcmp(&n, &m, sizeof n) == 0)
12         printf("n et m sont égaux\n");
13
14     if (memcmp(&n, &o, sizeof n) != 0)
15         printf("n et o ne sont pas égaux\n");
16
17     return 0;
18 }

```

```

1  n et m sont égaux
2  n et o ne sont pas égaux

```



Dit autrement, la fonction `memcmp()` retourne zéro si les deux portions comparées ont la même représentation.

En résumé

1. les entiers non signés sont représentés sous forme d'une somme de puissance de deux ;
2. il existe trois représentations possibles pour les entiers signés, mais la plus fréquente est celle en complément à deux ;
3. les types entiers peuvent contenir des *bits* de bourrage *sauf* le type `char` ;
4. les types flottants sont représentés sous forme d'un signe, d'un exposant et d'une mantisse, mais sont susceptibles d'engendrer des approximations et des pertes de précision ;
5. la représentation des pointeurs est indéterminée ;
6. le boutisme utilisé dépend de la machine cible ;
7. les constantes sont toujours écrites suivant la représentation gros-boutiste.

Nous avons abordés les limites des types à plusieurs reprises dans les chapitres précédents, mais nous ne nous sommes pas encore véritablement arrêté sur ce point. Il est à présent temps pour nous de vous présenter les implications de ces limites.

27.1 Les limites des types

Au début de ce cours, nous vous avons précisés que tous les types avaient des bornes et qu'en conséquence, ils ne pouvaient stocker qu'un intervalle défini de valeurs. Nous vous avons également indiqué que la norme du langage imposait des minimas pour ces intervalles qui, pour rappel, sont les suivants.

Toutefois, mise à part pour le type `char`, les bornes des types vont le plus souvent au-delà de ces minimums garantis. Cependant, pour pouvoir manipuler nos types dans leur limites, encore faut-il les connaître. Heureusement pour nous, la norme nous fournit deux en-têtes définissant des macroconstantes correspondant aux limites des types entiers et flottants : `<limits.h>` et `<float.h>`.

27.1.1 L'en-tête `<limits.h>`



Remarquez qu'il n'y a pas de macroconstantes pour le minimum des types entiers non signés puisque celui-ci est *toujours* zéro.

À ces macroconstantes, il faut également ajouter `CHAR_BIT` qui vous indique le nombre *bits* composant un multiplét. Celle-ci a pour valeur minimale huit, c'est-à-dire qu'en C, un multiplét

Type	Minimum	Maximum
signed char	-127	127
unsigned char	0	255
short	-32 767	32 767
unsigned short	0	65 535
int	-32 767	32 767
unsigned int	0	65 535
long	-2 147 483 647	2 147 483 647
unsigned long	0	4 294 967 295
float	-1×10^{37}	1×10^{37}
double	-1×10^{37}	1×10^{37}
long double	-1×10^{37}	1×10^{37}

Type	Minimum	Maximum
signed char	SCHAR_MIN	SCHAR_MAX
unsigned char	0	UCHAR_MAX
short	SHRT_MIN	SHRT_MAX
unsigned short	0	USHRT_MAX
int	INT_MIN	INT_MAX
unsigned int	0	UINT_MAX
long	LONG_MIN	LONG_MAX
unsigned long	0	ULONG_MAX

ne peut avoir moins de huit *bits*.

L'exemple suivant utilise ces macroconstantes et affiche leur valeur.

```

1  #include <limits.h>
2  #include <stdio.h>
3
4
5  int
6  main(void)
7  {
8      printf("Un multiplet se compose de %d bits.\n", CHAR_BIT);
9      printf("signed char : min = %d ; max = %d.\n", SCHAR_MIN, SCHAR_MAX);
10     printf("unsigned char : min = 0 ; max = %u.\n", UCHAR_MAX);
11     printf("short : min = %d ; max = %d.\n", SHRT_MIN, SHRT_MAX);
12     printf("unsigned short : min = 0 ; max = %u.\n", USHRT_MAX);
13     printf("int : min = %d ; max = %d.\n", INT_MIN, INT_MAX);
14     printf("unsigned int : min = 0 ; max = %u.\n", UINT_MAX);
15     printf("long : min = %ld ; max = %ld.\n", LONG_MIN, LONG_MAX);
16     printf("unsigned long : min = 0 ; max = %lu.\n", ULONG_MAX);
17     return 0;
18 }
```

```

1  Un multiplet se compose de 8 bits.
2  signed char : min = -128 ; max = 127.
3  unsigned char : min = 0 ; max = 255.
4  short : min = -32768 ; max = 32767.
5  unsigned short : min = 0 ; max = 65535.
6  int : min = -2147483648 ; max = 2147483647.
7  unsigned int : min = 0 ; max = 4294967295.
8  long : min = -9223372036854775808 ; max = 9223372036854775807.
9  unsigned long : min = 0 ; max = 18446744073709551615.
```



Bien entendu, les valeurs obtenues dépendent de votre machine, celles-ci sont simplement données à titre d'illustration.

27.1.2 L'en-tête <float.h>

Les types flottants amènent une subtilité supplémentaire : ceux-ci disposent en vérité de *quatre* bornes : le plus grand nombre représentable, le plus petit nombre représentable (qui est l'opposé du précédent), la plus petite partie fractionnaire représentable (autrement dit, le nombre le plus proche de zéro représentable) et son opposé. Dit autrement, les bornes des types flottants peuvent être schématisées comme suit.

```

1  -MAX          -MIN    0    MIN          MAX
2  +-----+    +    +-----+
```

Où **MAX** représente le plus grand nombre représentable et **MIN** le plus petit nombre représentable avant zéro. Étant donné que deux des bornes ne sont finalement que l'opposé des autres, deux macroconstantes sont suffisantes pour chaque type flottant.

Type	Maximum	Minimum de la partie fractionnaire
float	FLT_MAX	FLT_MIN
double	DBL_MAX	DBL_MIN
long double	LDBL_MAX	LDBL_MIN

```

1  #include <float.h>
2  #include <stdio.h>
3
4
5  int
6  main(void)
7  {
8      printf("float : min = %e ; max = %e.\n", FLT_MIN, FLT_MAX);
9      printf("double : min = %e ; max = %e.\n", DBL_MIN, DBL_MAX);
10     printf("long double : min = %Le ; max = %Le.\n", LDBL_MIN, LDBL_MAX);
11     return 0;
12 }
```

```

1  float : min = 1.175494e-38 ; max = 3.402823e+38.
2  double : min = 2.225074e-308 ; max = 1.797693e+308.
3  long double : min = 3.362103e-4932 ; max = 1.189731e+4932.
```

27.2 Les dépassements de capacité

Nous savons à présent comment obtenir les limites des différents types de base, c'est bien. Toutefois, nous ignorons toujours *quand* la limite d'un type est dépassée et ce qu'il se passe dans un tel cas.

Le franchissement de la limite d'un type est appelé un **dépassement de capacité** (*overflow* ou *underflow* en anglais, suivant que la limite maximale ou minimale est outrepassée). Un dépassement survient lorsqu'une opération produit un résultat dont la valeur n'est pas représentable par le type de l'expression ou lorsqu'une valeur est convertie vers un type qui ne peut la représenter.

Ainsi, dans l'exemple ci-dessous, l'expression `INT_MAX + 1` provoquent un dépassement de capacité, de même que la conversion (implicite) de la valeur `INT_MAX` vers le type `signed char`.

1



Gardez en mémoire que des conversions implicites ont lieu lors des opérations. Revoyez le chapitre sur les opérations mathématiques si cela vous paraît lointain.



D'accord, mais que se passe-t-il dans un tel cas ?

C'est ici que le bât blesse : ce n'est pas précisé. :-°
Ou, plus précisément, cela dépend des situations.

27.2.1 Dépassement lors d'une opération

Les types entiers

Les entiers signés

Si une opération travaillant avec des entiers signés dépasse la capacité du type du résultat, le comportement est indéfini. Dans la pratique, il y a trois possibilités :

1. La valeur boucle, c'est-à-dire qu'une fois une limite atteinte, la valeur revient à la seconde. Cette solution est la plus fréquente et s'explique par le résultat de l'arithmétique en complément à deux. En effet, à supposer qu'un `int` soit représenté sur 32 *bits* :

— `INT_MAX + 1` se traduit par

`0111 1111 1111 1111 1111 1111 1111 1111 + 0000 0000 0000 0000 0000 0000 0000 0001`, ce qui donne `1000 0000 0000 0000 0000 0000 0000 0000` soit `INT_MIN` ; et

— `INT_MIN - 1` se traduit par

`1000 0000 0000 0000 0000 0000 0000 0000 + 1111 1111 1111 1111 1111 1111 1111 1111` ce qui donne `0111 1111 1111 1111 1111 1111 1111 1111` soit `INT_MAX`.

2. La valeur sature, c'est-à-dire qu'une fois une limite atteinte, la valeur reste bloquée à celle-ci.
3. Le processeur considère l'opération comme invalide et stoppe l'exécution du programme.

Les entiers non signés

Si une opération travaillant avec des entiers non signés dépasse la capacité du type du résultat, alors il n'y a proprement parler *pas* de dépassement et la valeur boucle. Autrement dit, l'expression `UINT_MAX + 1` donne *toujours* zéro.

Les types flottants

Si une opération travaillant avec des flottants dépasse la capacité du type du résultat, alors, comme pour les entiers signés, le comportement est indéfini. Toutefois, les flottants étant le plus souvent représenté suivant la norme IEEE 754, le résultat sera souvent le suivant :

1. En cas de dépassement de la borne maximale ou minimale, le résultat sera égal à l'infini (positif ou négatif).
2. En cas de dépassement de la limite de la partie fractionnaire, le résultat sera arrondi à zéro.

Néanmoins, gardez à l'esprit que ceci *n'est pas garanti* par la norme. Il est donc parfaitement possible que votre programme soit tout simplement arrêté.

27.2.2 Dépassement lors d'une conversion

Valeur entière vers entier signé

Si la conversion d'une valeur entière vers un type signé produit un dépassement, le résultat n'est pas déterminé.

Valeur entière vers entier non signé

Dans le cas où la conversion d'une valeur entière vers un type non signé, la valeur boucle, comme précisé dans le cas d'une opération impliquant des entiers non signés.

Valeur flottante vers entier

Lors d'une conversion d'une valeur flottante vers un type entier (signé ou non signé), la partie fractionnaire est ignorée. Si la valeur restante n'est pas représentable, le résultat est indéfini (*y compris* pour les types non signés, donc).

Valeur entière vers flottant

Si la conversion d'une valeur entière vers un type flottant implique un dépassement, le comportement est indéfini.



Notez que si le résultat n'est pas *exactement* représentable (rappelez-vous des pertes de précision) la valeur sera approchée, mais cela ne constitue pas un dépassement.

Valeur flottante vers flottant

Enfin, si la conversion d'une valeur flottante vers un autre type flottant produit un dépassement, le comportement est indéfini.



Comme pour le cas d'une conversion d'un type entier vers un type flottant, si le résultat n'est pas *exactement* représentable, la valeur sera approchée.

27.3 Gérer les dépassements entiers

Nous savons à présent ce qu'est un dépassement, quand ils se produisent et dans quel cas ils sont problématiques (en vérité, potentiellement tous, puisque même si la valeur boucle, ce n'est pas forcément ce que nous souhaitons). Reste à présent pour nous à les gérer, c'est-à-dire les détecter et les éviter. Malheureusement pour nous, la bibliothèque standard ne nous fournit aucun outil à cet effet, mise à part les macroconstantes qui vous ont été présentées. Aussi il va nous être nécessaire de réaliser nos propres outils.

27.3.1 Préparation

Avant de nous lancer dans la construction de ces fonctions, il va nous être nécessaire de déterminer leur forme. En effet, si, intuitivement, il vous vient sans doute à l'esprit de créer des fonctions prenant deux paramètres et fournissant un résultat, reste le problème de la gestion d'erreur.

```
1 int safe_add(int a, int b);
```

En effet, comment précise-t-on à l'utilisateur qu'un dépassement a été détecté, toutes les valeurs du type `int` pouvant être retournées ? Eh bien, pour résoudre ce souci, nous allons recourir dans les exemples qui suivent à la variable `errno`, à laquelle nous affecteront la valeur `ERANGE` en cas de dépassement. Ceci implique qu'elle soit mise à zéro avant tout appel à ces fonctions (revoyez le premier chapitre sur la gestion d'erreur si cela ne vous dit rien).

Par ailleurs, afin que ces fonctions puissent éventuellement être utilisées dans des suites de calculs, nous allons faire en sorte qu'elle retourne la borne maximale ou minimale en cas de dépassement.

Ceci étant posé, passons à la réalisation de ces fonctions. :)

27.3.2 L'addition

Afin d'empêcher un dépassement, il va nous falloir le détecter et ceci, bien entendu, à l'aide de condition qui ne doivent pas elle-même produire de dépassement. Ainsi, ce genre de condition est à proscrire :

```
if (a + b > INT_MAX).
```

Pour le cas d'une addition deux cas de figures se présentent :

1. `b` est positif et il nous faut alors vérifier que la somme ne dépasse pas le maximum.
2. `b` est négatif et il nous faut dans ce cas déterminer si la somme dépasse ou non le minimum.

Ceci peut être contrôlé en vérifiant que `a` n'est pas supérieur ou inférieur à, respectivement, `MAX - b` et `MIN - b`. Ainsi, il nous est possible de rédiger une fonction comme suit.

```

1  int safe_add(int a, int b)
2  {
3      int err = 1;
4
5      if (b >= 0 && a > INT_MAX - b)
6          goto overflow;
7      else if (b < 0 && a < INT_MIN - b)
8          goto underflow;
9
10     return a + b;
11 underflow:
12     err = -1;
13 overflow:
14     errno = ERANGE;
15     return err > 0 ? INT_MAX : INT_MIN;
16 }

```

Contrôlons à présent son fonctionnement.

```

1  #include <errno.h>
2  #include <limits.h>
3  #include <stdio.h>
4  #include <string.h>
5
6  /* safe_add() */
7
8
9  static void test_add(int a, int b)
10 {
11     errno = 0;
12     printf("%d + %d = %d", a, b, safe_add(a, b));
13     printf(" (%s).\n", strerror(errno));
14 }
15
16
17 int main(void)
18 {
19     test_add(INT_MAX, 1);
20     test_add(INT_MIN, -1);
21     test_add(INT_MIN, 1);
22     test_add(INT_MAX, -1);
23     return 0;
24 }

```

```

1  2147483647 + 1 = 2147483647 (Numerical result out of range).
2  -2147483648 + -1 = -2147483648 (Numerical result out of range).
3  -2147483648 + 1 = -2147483647 (Success).
4  2147483647 + -1 = 2147483646 (Success).

```

27.3.3 La soustraction

Pour la soustraction, le principe est le même que pour l'addition si ce n'est que les tests doivent être quelques peu modifiés. En guise d'exercice, essayez de trouver la solution par vous-même. ;)

```

1  int safe_sub(int a, int b)
2  {
3      int err = 1;
4
5      if (b >= 0 && a < INT_MIN + b)
6          goto underflow;
7      else if (b < 0 && a > INT_MAX + b)
8          goto overflow;
9
10     return a - b;
11 underflow:
12     err = -1;

```

```

13 overflow:
14     errno = ERANGE;
15     return err > 0 ? INT_MAX : INT_MIN;
16 }

```

❓ Mais ?! Pourquoi ne pas simplement faire `safe_add(a, -b)` ?

Bonne question !

Cela a de quoi surprendre de prime à bord, mais l'opérateur de négation `-` peut produire un dépassement. En effet, souvenez-vous : la représentation en complément à deux ne dispose que d'une seule représentation pour zéro, du coup, il reste une représentation qui est possiblement invalide où seul le *bit* de signe est à un. Cependant, le plus souvent, cette représentation est utilisée pour fournir un nombre négatif supplémentaire. Or, si c'est le cas, il y a une asymétrie entre la borne inférieure et supérieure et la négation de la limite inférieure produira un dépassement. Il nous est donc nécessaire de prendre ce cas en considération.

27.3.4 La négation

À ce sujet, pour réaliser cette opération, deux solutions s'offrent à nous :

1. Vérifier si l'opposé de la borne supérieure est plus grand que la borne inférieure (autrement dit que `-INT_MAX > INT_MIN`) et, si c'est le cas, vérifier que le paramètre n'est pas la borne inférieure ; ou
2. Vérifier `sub(0, a)` où `a` est l'opérande qui doit subir la négation.

Notre préférence allant à la seconde solution, notre fonction sera donc la suivante.

```

1 int safe_neg(int a)
2 {
3     return safe_sub(0, a);
4 }

```

27.3.5 La multiplication

Pour la multiplication, cela se corse un peu, notamment au vu de ce qui a été dit précédemment au sujet de la représentation en complément à deux. Procédons par ordre :

1. Tout d'abord, si l'un des deux opérands est positif, nous devons vérifier que le minimum n'est pas atteint (puisque le résultat sera négatif) et, si les deux sont positifs, que le maximum n'est pas dépassé. Toutefois, vu que nous allons recourir à la division pour le déterminer, nous devons en plus vérifier que le diviseur n'est pas nul.

```

1 if (b > 0)
2 {
3     if (a > 0 && a > INT_MAX / b)
4         goto overflow;
5     else if (a < 0 && a < INT_MIN / b)
6         goto underflow;
7 }

```

Ensuite, si l'un des deux opérands est négatif, nous devons effectuer la même vérification que précédemment et, si les deux sont négatifs, vérifier que le résultat ne dépasse pas le maximum. Toutefois, nous devons également faire attention à ce que les opérands ne soient pas nuls ainsi qu'au cas de dépassement possible par l'expression `INT_MIN / -1` (en cas de représentation en complément à deux).

```

1  else if (b < 0)
2  {
3      if (a < 0 && a < INT_MAX / b)
4          goto overflow;
5      else if ((-INT_MAX > INT_MIN && b < -1) && a > INT_MIN / b)
6          goto underflow;
7  }

```

Ce qui nous donne finalement la fonction suivante.

```

1  int safe_mul(int a, int b)
2  {
3      int err = 1;
4
5      if (b > 0)
6      {
7          if (a > 0 && a > INT_MAX / b)
8              goto overflow;
9          else if (a < 0 && a < INT_MIN / b)
10             goto underflow;
11      }
12      else if (b < 0)
13      {
14          if (a < 0 && a < INT_MAX / b)
15              goto overflow;
16          else if ((-INT_MAX > INT_MIN && b < -1) && a > INT_MIN / b)
17              goto underflow;
18      }
19
20      return a * b;
21  underflow:
22      err = -1;
23  overflow:
24      errno = ERANGE;
25      return err > 0 ? INT_MAX : INT_MIN;
26  }

```

27.3.6 La division

Rassurez-vous, la division est nettement moins pénible à vérifier. En fait, il y a un seul cas problématique : si les deux opérandes sont `INT_MIN` et `-1` et que l'opposé du maximum est inférieur à la borne minimale.

```

1  int safe_div(int a, int b)
2  {
3      if (-INT_MAX > INT_MIN && b == -1 && a == INT_MIN)
4      {
5          errno = ERANGE;
6          return INT_MIN;
7      }
8
9      return a / b;
10 }

```

27.3.7 Le modulo

Enfin, pour le modulo, les mêmes règles que celles de la division s'appliquent.

```

1  int safe_mod(int a, int b)
2  {
3      if (-INT_MAX > INT_MIN && b == -1 && a == INT_MIN)
4      {
5          errno = ERANGE;
6          return INT_MIN;
7      }

```

```

8
9     return a % b;
10 }

```

27.4 Gérer les dépassements flottants

La tâche se complique un peu avec les flottants, puisqu'en plus de tester les limites supérieures (cas d'*overflow*), il faudra aussi s'intéresser à celles de la partie fractionnaire (cas d'*underflow*).

27.4.1 L'addition

Comme toujours, il faut prendre garde à ce que les codes de vérification ne provoquent pas eux-mêmes des dépassements. Pour implémenter l'opération d'addition, on commence donc par se ramener au cas plus simple où on connaît le signe des opérandes `a` et `b`.

- Si `a` et `b` sont de signes opposés (par exemple : `a >= 0` et `b >= 0`, alors il ne peut pas se produire d'*overflow*.
- Si `a` et `b` sont de même signes, alors il ne peut pas se produire d'*underflow*.

Dans le premier cas, tester l'*underflow* revient à tester si `a + b` est censé (dans l'arithmétique usuelle) être compris entre `-DBL_MIN` et `DBL_MIN`. Il faut cependant exclure 0 de cet intervalle, de manière à ce que `1.0 + (-1.0)`, par exemple, ne soit pas considéré comme un *underflow*. Si par exemple `a >= 0` (et dans ce cas `b <= 0`), il suffit alors de tester si `-DBL_MIN - a < b` et `a < DBL_MIN - b`.

Dans le deuxième cas, il reste à vérifier l'absence d'*overflow*. Si par exemple `a` et `b` sont tous deux positifs, il s'agit de vérifier que `a <= DBL_MAX - b`.

```

1  /* Retourne a+b. */
2  /* En cas d'overflow, +-DBL_MAX est retourné (suivant le signe de a+b) */
3  /* En cas d'underflow, +- DBL_MIN est retourné */
4  /* errno est fixé en conséquence */
5  double safe_addf(double a, double b)
6  {
7      int positive_result = a > -b;
8      double ret_value = positive_result ? DBL_MAX : -DBL_MAX;
9
10     if (a == -b)
11         return 0;
12     else if ((a < 0) != (b < 0))
13     {
14         /* On se ramène au cas où a <= 0 et b >= 0 */
15         if (a <= 0)
16         {
17             double t = a;
18             a = b;
19             b = t;
20         }
21         if (-DBL_MIN - a < b && a < DBL_MIN - b)
22             goto underflow;
23     }
24     else
25     {
26         int negative = 0;
27
28         /* On se ramène au cas où a et b sont positifs */
29         if (a < 0)
30         {
31             a = -a;
32             b = -b;
33             negative = 1;
34         }
35
36         if (a > DBL_MAX - b)
37             goto overflow;
38

```



```

39     if (negative)
40     {
41         a = -a;
42         b = -b;
43     }
44 }
45
46 return a + b;
47
48 underflow:
49     ret_value = positive_result ? DBL_MIN : -DBL_MIN;
50 overflow:
51     errno = ERANGE;
52     return ret_value;
53 }

```

Quelques tests :

```

1  static void test_addf(double a, double b)
2  {
3      errno = 0;
4      printf("%e + %e = %e", a, b, safe_addf(a, b));
5      printf(" (%s)\n", strerror(errno));
6  }
7
8  int main(void)
9  {
10     test_addf(1, 1);
11     test_addf(5, -6);
12     test_addf(DBL_MIN, -3./2*DBL_MIN); /* underflow */
13     test_addf(DBL_MAX, 1);
14     test_addf(DBL_MAX, -DBL_MAX);
15     test_addf(DBL_MAX, DBL_MAX); /* overflow */
16     test_addf(-DBL_MAX, -1./2*DBL_MAX); /* overflow */
17     return 0;
18 }
19
20

```

Exemple de résultat :

On peut faire deux remarques sur la fonction `safe_addf` ci-dessus :

- `addf(DBL_MAX, 1)` peut ne pas être considéré comme un *overflow* du fait des pertes de précision.
- à l'inverse, si un *underflow* se déclare, cela ne veut pas forcément dire qu'il y a une erreur irrécupérable dans l'algorithme. Il est possible que les pertes de précision dans les représentations des flottants les causent elles-mêmes.



Pour être tout à fait rigoureux, il faut remarquer que des expressions telles que `-DBL_MIN - a` et `DBL_MAX - b` (avec `a >= 0` et `b >= 0`) peuvent théoriquement générer respectivement un *overflow* et un *underflow*. Cela ne peut cependant se produire que sur des implémentations marginales des flottants, puisque cela signifierait que la partie fractionnaire (la mantisse) est représentée sur plus de bits que la différence entre les valeurs extrémales de l'exposant. Nous avons donc omis ces vérifications pour des raisons de concision.

27.4.2 La soustraction

Pour notre plus grand bonheur, étant donné qu'il n'y a pas d'asymétrie entre les bornes, la soustraction revient à faire `safe_addf(a, -b)`.

27.4.3 La multiplication

Heureusement, la multiplication est un peu plus simple que l'addition ! Comme tout à l'heure, on va essayer de contrôler la valeur des arguments de manière à savoir dans quel cas on se situe potentiellement (*underflow* ou *overflow*).

Ici, ce n'est plus le signe de `a` et de `b` qui va nous intéresser : au contraire, on va même le gommer en utilisant la valeur absolue. Cependant, la position de `a` et de `b` par rapport à +1 et -1 est essentielle :

- si `fabs(b) <= 1`, `a * b` peut produire un *underflow* mais pas d'*overflow*.
- si `fabs(b) >= 1`, `a * b` peut produit un *overflow* mais pas d'*underflow*.



La fonction `fabs()` est définie dans l'en-tête `<math.h>` et, comme son nom l'indique, retourne la valeur absolue de son argument. N'oubliez pas à cet égard de préciser l'option `-lm` lors de la compilation.

Ainsi, dans le premier cas, on vérifiera si `fabs(a) < DBL_MIN / fabs(b)`. L'expression `DBL_MIN / fabs(b)` ne peut ni produire d'*underflow* (puisque `fabs(b) <= 1`) ni d'*overflow* (puisque `DBL_MIN <= fabs(b)`). De même, dans le deuxième cas, la condition pourra s'écrire `fabs(a) > DBL_MAX / fabs(b)`.

```

1 double safe_mul(double a, double b)
2 {
3     int different_signs = (a < 0) != (b < 0);
4     double ret_value = different_signs ? -DBL_MAX : DBL_MAX;
5
6     if (fabs(b) < 1)
7     {
8         if (fabs(a) < DBL_MIN / fabs(b))
9             goto underflow;
10    }
11    else
12    {
13        if (fabs(a) > DBL_MAX / fabs(b))
14            goto overflow;
15    }
16
17    return a * b;
18
19 underflow:
20     ret_value = different_signs ? -DBL_MIN : DBL_MIN;
21 overflow:
22     errno = ERANGE;
23     return ret_value;
24 }
```

Quelques tests :

```

1 static void test_add(double a, double b)
2 {
3     errno = 0;
4     printf("%e + %e = %e", a, b, safe_addf(a, b));
5     printf(" (%s)\n", strerror(errno));
6 }
7
8 int main(void)
9 {
10    test_add(1, 1);
11    test_add(5, -6);
12    test_add(DBL_MIN, -3./2*DBL_MIN); /* underflow */
13    test_add(DBL_MAX, 1);
14    test_add(DBL_MAX, -DBL_MAX);
15    test_add(DBL_MAX, DBL_MAX); /* overflow */
16    test_add(-DBL_MAX, -1./2*DBL_MAX); /* overflow */
17    return 0;
18 }
```

Exemple de résultat :

```

1  -1.000000e+00 * -1.000000e+00 = 1.000000e+00 (Success)
2  2.000000e+00 * 3.000000e+00 = 6.000000e+00 (Success)
3  1.797693e+308 * 2.000000e+00 = 1.797693e+308 (Numerical result out of range)
4  2.225074e-308 * 5.000000e-01 = 2.225074e-308 (Numerical result out of range)
5  -3.337611e-308 * 6.666667e-01 = -2.225074e-308 (Success)
6  -8.988466e+307 * 1.500000e+00 = -1.348270e+308 (Success)
7  -8.988466e+307 * 3.000000e+00 = -1.797693e+308 (Numerical result out of range)

```

27.4.4 La division

La division ressemble sensiblement à la multiplication. On traite cependant en plus le cas où le dénominateur est nul.

```

1  double safe_divf(double a, double b)
2  {
3      int different_signs = (a < 0) != (b < 0);
4      double ret_value = different_signs ? -DBL_MAX : DBL_MAX;
5
6      if (b == 0)
7      {
8          errno = EDOM;
9          return ret_value;
10     }
11     else if (fabs(b) < 1)
12     {
13         if (fabs(a) > DBL_MAX * fabs(b))
14             goto overflow;
15     }
16     else
17     {
18         if (fabs(a) < DBL_MIN * fabs(b))
19             goto underflow;
20     }
21
22     return a / b;
23
24 underflow:
25     ret_value = different_signs ? -DBL_MIN : DBL_MIN;
26 overflow:
27     errno = ERANGE;
28     return ret_value;
29 }

```

Ce chapitre aura été fastidieux, n'hésitez pas à vous poser après sa lecture et à tester les différents codes fournis afin de correctement appréhender les différentes limites.

En résumé

1. Les limites des différents types sont fournies par des macroconstantes définies dans les en-tête `<limits.h>` et `<float.h>` ;
2. Les types flottants disposent de quatre limites et non de deux ;
3. Un dépassement de capacité se produit soit quand une opération produit un résultat hors des limites du type de son expression ou lors de la conversion d'une valeur vers un type qui ne peut représenter celle-ci ;
4. En cas de dépassement, le comportement est indéfini *sauf* pour les calculs impliquant des entiers non signés et pour les conversions d'un type entier vers un type entier non signé, auquel cas la valeur boucle.
5. La bibliothèque standard ne fournit aucun outils permettant de détecter et/ou de gérer ces dépassements.

Nous avons vu dans le chapitre précédent la représentation des différents type et notamment celle des types entiers. Ce chapitre est en quelque sorte le prolongement de celui-ci puisque nous allons à présent voir comment manipuler directement les *bits* à l'aide des opérateurs de manipulation des *bits* et des champs de *bits*.

28.1 Les opérateurs de manipulation des bits

28.1.1 Présentation

Le langage C définit six opérateurs permettant de manipuler les *bits* :

- l'opérateur « et » : `&` ;
- l'opérateur « ou inclusif » : `|` ;
- l'opérateur « ou exclusif » : `^` ;
- l'opérateur de négation ou de complément : `~` ;
- l'opérateur de décalage à droite : `>>` ;
- l'opérateur de décalage à gauche : `<<`.



Veillez à ne pas confondre les opérateurs de manipulations des *bits* « et » (`&`) et « ou inclusif » (`|`) avec leurs homologues « et » (`&&`) et « ou » (`||`) logiques. Il s'agit d'opérateurs totalement différents au même titre que les opérateurs d'affectation (`=`) et d'égalité (`==`). De même, l'opérateur de manipulations des *bits* « et » (`&`) n'a pas de rapport avec l'opérateur d'adressage (`&`), ce dernier n'utilisant qu'un opérande.



Notez que tous ces opérateurs travaillent uniquement sur des *entiers*. | Si néanmoins vous souhaitez modifier la représentation d'un type flottant (ce que nous vous déconseillons), vous pouvez accéder à ses *bits* à l'aide d'un pointeur sur `unsigned char` (revoyez [le chapitre sur l'allocation dynamique](#) si cela ne vous dit rien).

28.1.2 Les opérateurs « et », « ou inclusif » et « ou exclusif »

Chacun de ces trois opérateurs attend deux opérandes entiers et produit un nouvel entier en appliquant une table de vérité à chaque paire de *bits* formée à partir des *bits* des deux nombres de départ. Plus précisément :

- L'opérateur « et » (`&`) donnera 1 si les deux *bits* de la paire sont à 1 ;
- L'opérateur « ou inclusif » (`|`) donnera 1 si *au moins* un des deux *bits* de la paire est à 1 ;

Bit 1	Bit 2	Opérateur « et »	Opérateur « ou inclusif »	Opérateur « ou exclusif »
0	0	0	0	0
1	0	0	1	1
0	1	0	1	1
1	1	1	1	0

— L'opérateur « ou exclusif » (\oplus) donnera 1 si *un seul* des deux *bits* de la paire est à 1.

Ceci est résumé dans le tableau ci-dessous, donnant le résultat des trois opérateurs pour chaque paires de *bits* possibles.

L'exemple ci-dessous utilise ces trois opérateurs. Comme vous le voyez, les *bits* des deux opérandes sont pris un à un pour former des paires et chacune d'elle se voit appliquer la table de vérité correspondante afin de produire un nouveau nombre entier.

```

1  #include <stdio.h>
2
3
4  int main(void)
5  {
6      int a = 0x63; /* 0x63 == 99 == 0110 0011 */
7      int b = 0x2A; /* 0x2A == 42 == 0010 1010 */
8
9      /* 0110 0011 & 0010 1010 == 0010 0010 == 0x22 == 34 */
10     printf("%2X\n", a & b);
11     /* 0110 0011 | 0010 1010 == 0110 1011 == 0x6B == 107 */
12     printf("%2X\n", a | b);
13     /* 0110 0011 ^ 0010 1010 == 0100 1001 == 0x49 == 73 */
14     printf("%2X\n", a ^ b);
15     return 0;
16 }
```

```

1  22
2  6B
3  49
```

28.1.3 L'opérateur de négation ou de complément

L'opérateur de négation a un fonctionnement assez simple : il inverse les *bits* de son opérande (les uns deviennent des zéros et les zéros des uns).

```

1  #include <stdio.h>
2
3
4  int
5  main(void)
6  {
7      unsigned a = 0x7F; /* 0111 1111 */
8
9      /* ~0111 1111 == 1000 0000 */
10     printf("%2X\n", ~a);
11     return 0;
12 }
```

```

1  FFFFFFFF
```



Notez que *tous* les *bits* ont été inversés, d'où le nombre élevé que nous obtenons puisque les *bits* de poids forts ont été mis à un. Ceci nous permet par ailleurs de constater que, sur notre machine, il y a visiblement quatre octets qui sont utilisés pour représenter

! la valeur d'un objet de type `unsigned int`.

⊗ N'oubliez pas que les représentations entières *signées* ont chacune une représentation qui est susceptible d'être invalide. Les opérateurs de manipulation des *bits* vous permettant de modifier directement la représentation, vous devez éviter d'obtenir ces dernières.

Ainsi, les exemples suivants sont susceptibles de produire des valeurs incorrectes (à supposer que la taille du type `int` soit de quatre octets sans *bits* de bourrages).

```
1  /* Invalide en cas de représentation en complément à deux ou signe et magnitude */
2  int a = ~0x7FFFFFFF;
3  /* Idem */
4  int b = 0x00000000 | 0x80000000;
5  /* Invalide en cas de représentation en complément à un */
6  int c = ~0;
7  /* Idem */
8  int d = 0x11110000 ^ 0x00001111;
```

i Notez toutefois que les entiers *non signés*, eux, ne subissent pas ces restrictions.

28.1.4 Les opérateurs de décalage

Les opérateurs de décalage, comme leur nom l'indique, décalent la valeur des *bits* d'un objet d'une certaine quantité, soit vers la gauche (c'est-à-dire vers le *bit* de poids fort), soit vers la droite (autrement dit, vers le *bit* de poids faible). Ils attendent deux opérandes : le nombre dont les *bits* doivent être décalés et la grandeur du décalage.

⊗ Un décalage ne peut être négatif ni être supérieur *ou égal* au nombre de *bits* composant l'objet décalé. Ainsi, si le type `int` utilise 32 *bits* (sans *bits* de bourrage), le décalage ne peut être plus grand que 31.

L'opérateur de décalage à gauche

L'opérateur de décalage à gauche translate la valeur des *bits* vers le *bit* de poids forts. Les *bits* de poids faibles perdant leur valeur durant l'opération sont mis à zéro. Techniquement, l'opération de décalage à gauche revient à calculer la valeur de l'expression $a \times 2^y$.

```
1  #include <stdio.h>
2
3
4  int
5  main(void)
6  {
7      /* 0000 0001 << 2 == 0000 0100 */
8      int a = 1 << 2;
9      /* 0010 1010 << 2 == 1010 1000 */
10     int b = 42 << 2;
11
12     printf("a = %d, b = %d\n", a, b);
13     return 0;
14 }
```

```
1  a = 4, b = 168
```

Opérateur combiné	Équivalent à
variable &= nombre	variable = variable & nombre
variable = nombre	variable = variable nombre
variable ^= nombre	variable = variable ^ nombre
variable «= nombre	variable = variable « nombre
variable »= nombre	variable = variable » nombre

✗ La première opérande ne peut être un nombre négatif.

! L'opération de décalage à gauche revenant à effectuer une multiplication, celle-ci est soumise au risque de dépassement de capacité que nous verrons au chapitre suivant.

L'opérateur de décalage à droite

L'opérateur de décalage à droite translate la valeur des *bits* vers le *bit* de poids faible. Dans le cas où la première opérande est un entier *non signé* ou un entier signé *positif*, les *bits* de poids forts perdant leur valeur durant l'opération sont mis à zéro. Si en revanche il s'agit d'un nombre signé *négatif*, les *bits* perdant leur valeur se voient mis à zéro ou un suivant la machine cible. Techniquement, l'opération de décalage à droite revient à calculer la valeur de l'expression $a / 2^y$.

```

1  #include <stdio.h>
2
3
4  int
5  main(void)
6  {
7      /* 0001 0000 >> 2 == 0000 0100 */
8      int a = 16 >> 2;
9      /* 0010 1010 >> 2 == 0000 1010 */
10     int b = 42 >> 2;
11
12     printf("a = %d, b = %d\n", a, b);
13     return 0;
14 }
```

```

1  a = 4, b = 10
```

i Dans le cas où une valeur est translatée au-delà du *bit* de poids faible, elle est tout simplement perdue

28.1.5 Opérateurs combinés

Enfin, sachez que, comme pour les opérations arithmétiques usuelles, les opérateurs de manipulation des *bits* disposent d'opérateurs combinés réalisant une affectation et une opération.

28.2 Masques et champs de bits

28.2.1 Les masques

Une des utilisations fréquentes des opérateurs de manipulations des *bits* est l'emploi de **masques**. Un masque est un ensemble de *bits* appliqué à un second ensemble *de même taille*

lors d'une opération de manipulation des *bits* (plus précisément, uniquement les opérations `&`, `|` et `^`) en vue soit de sélectionner un sous-ensemble, soit de modifier un sous-ensemble.

Modifier la valeur d'un bit

Mise à zéro

Cette définition doit probablement vous paraître quelque peu abstraite, aussi, prenons un exemple.

```
1 unsigned short n;
```

Nous disposons d'une variable `n` de type `unsigned short` (que nous supposons composées de deux octets pour nos exemples) et souhaiterions mettre le *bit* de poids fort à zéro.

Une solution consiste à appliquer les opérateurs de manipulation des *bits* afin d'obtenir la valeur voulue. Étant donné que nous désirons mettre un *bit* à zéro, nous pouvons déjà abandonner l'opérateur `|` au vu de sa table de vérité. Également, l'opérateur `^` ne convient pas tout à fait puisqu'il inverserait la valeur du *bit* au lieu de la mettre à zéro. Il nous reste donc l'opérateur `&`.

Avec cet opérateur, il nous est possible d'utiliser une valeur qui nous donnera le bon résultat. Cette valeur, de même taille que celle de la variable `n`, est précisément un masque qui va « cacher », « masquer » une partie de la valeur.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4
5 int main(void)
6 {
7     unsigned short n;
8
9     if (scanf("%hx", &n) != 1)
10    {
11        perror("scanf");
12        return EXIT_FAILURE;
13    }
14
15    printf("%X\n", n & 0x7FFF);
16    return 0;
17 }
```

```
1 8FFF
2 FFF
3
4 7F
5 7F
```

Comme vous le voyez, l'opérateur `&` peut être utilisé pour sélectionner une partie de la valeur de `n` en mettant à un les *bits* que nous souhaitons garder (en l'occurrence tous sauf le *bit* de poids fort) et les autres à zéro.

Mise à un

À l'inverse, les opérateurs de manipulation des *bits* peuvent être employés pour mettre un ou plusieurs *bits* à un. Dans ce cas, c'est l'opérateur `&` qui ne convient pas au vu de sa table de vérité.

Si nous reprenons notre exemple précédent et que nous souhaitons modifier la valeur de la variable `n` de sorte de mettre à un le *bit* de signe, nous pouvons procéder comme suit.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
```



```

4
5 int main(void)
6 {
7     unsigned short n;
8
9     if (scanf("%hx", &n) != 1)
10    {
11        perror("scanf");
12        return EXIT_FAILURE;
13    }
14
15    printf("%X\n", n | 0x8000);
16    return 0;
17 }

```

```

1 FFF
2 8FFF
3
4 7F
5 807F

```

Comme vous le voyez, l'opérateur `|` peut être utilisé de la même manière dans ce cas ci à l'aide d'un masque dont les *bits* qui doivent être mis à un sont... à un.

28.2.2 Les champs de bits

Mise en situation

Une autre utilisation des opérateurs de manipulation des *bits* est le compactage de données entières.

Imaginons que nous souhaitions stocker la date courante sous la forme de trois entiers : l'année, le mois et le jour. La première solution qui vous viendra à l'esprit sera probablement de recourir à une structure, par exemple comme celle ci-dessous, ce qui est un bon réflexe.

```

1 struct date {
2     unsigned char jour;
3     unsigned char mois;
4     unsigned short annee;
5 };

```

Toutefois, nous gaspillons finalement de la mémoire avec ce système. En effet, techniquement, nous aurions besoin de 12 *bits* pour stocker l'année (afin d'avoir un peu de marge jusque l'an 4095 :p), 4 pour le mois et 5 pour le jour, ce qui nous fait un total de 21 *bits* contre 32 pour notre structure (à supposer que le type `short` fasse deux octets et le type `char` un octet), sans compter les multipliants de bourrage (revoyez le chapitre sur les structures si cela ne vous dit rien).

Ceci n'est pas gênant dans la plupart des cas, mais cela peut le devenir si la mémoire disponible vient à manquer ou si cette structure est amenée à être créée un grand nombre de fois.

Avec les opérateurs de manipulations des *bits*, il nous est possible de stocker ces trois champs dans un tableau de trois `unsigned char` afin d'économiser de la place.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4
5 static void modifie_jour(unsigned char *date, unsigned jour)
6 {
7     /* Nous stockons le jour (cinq bits). */
8     date[0] |= jour;
9 }

```

```

10
11
12 static void modifie_mois(unsigned char *date, unsigned mois)
13 {
14     /* Nous ajoutons les trois premiers bits du mois après ceux du jour. */
15     date[0] |= (mois & 0x07) << 5;
16     /* Puis le bit restant dans le second octet. */
17     date[1] |= (mois >> 3);
18 }
19
20
21 static void modifie_annee(unsigned char *date, unsigned annee)
22 {
23     /* Nous ajoutons sept bits de l'année après le dernier bit du mois. */
24     date[1] |= (annee & 0x7F) << 1;
25     /* Et ensuite les cinq restants. */
26     date[2] |= (annee) >> 7;
27 }
28
29
30 static unsigned calcule_jour(unsigned char *date)
31 {
32     return date[0] & 0x1F;
33 }
34
35
36 static unsigned calcule_mois(unsigned char *date)
37 {
38     return (date[0] >> 5) | ((date[1] & 0x1) << 3);
39 }
40
41
42 static unsigned calcule_annee(unsigned char *date)
43 {
44     return (date[1] >> 1) | (date[2] << 7);
45 }
46
47
48 int
49 main(void)
50 {
51     unsigned char date[3] = { 0 }; /* Initialisation à zéro. */
52     unsigned jour, mois, annee;
53
54     printf("Entrez une date au format jj/mm/aaaa : ");
55
56     if (scanf("%u/%u/%u", &jour, &mois, &annee) != 3) {
57         perror("fscanf");
58         return EXIT_FAILURE;
59     }
60
61     modifie_jour(date, jour);
62     modifie_mois(date, mois);
63     modifie_annee(date, annee);
64     printf("Le %u/%u/%u\n", calcule_jour(date), calcule_mois(date), calcule_annee(date));
65     return 0;
66 }

```

```

1 Entrez une date au format jj/mm/aaaa : 31/12/2042
2 Le 31/12/2042

```

Cet exemple amène quelques explications. Une fois les trois valeurs récupérées, il nous les compacter dans le tableau d'`unsigned char` :

1. Pour le jour, c'est assez simple, nous incorporons ses cinq *bits* à l'aide de l'opérateur `|` (les trois éléments du tableau étant à zéro au début, cela ne pose pas de problème).
2. Pour le mois, seuls trois *bits* étant encore disponibles, il va nous falloir répartir ceux-ci sur deux éléments du tableau. Tout d'abord, nous sélectionnons les trois premiers *bits* à l'aide du masque `0x07`, nous les décalons ensuite de cinq *bits* vers la gauche (afin de ne

pas écraser les cinq *bits* du jour) et, enfin, nous les ajoutons à l'aide de l'opérateur `||`. Le dernier *bit* est lui stocké dans le second élément et est sélectionné à l'aide d'un décalage vers la droite afin d'éliminer les trois premiers *bits* (qui ont déjà été traité).

3. Pour l'année, nous utilisons la même technique que pour le mois : nous sélectionnons les sept premiers *bits* à l'aide du masque `0x7F`, les décalons d'un *bit* vers la gauche en vue de ne pas écraser le *bit* du mois et les intégrons avec l'opérateur `||`. Les cinq *bits* restants sont ensuite insérés en recourant préalablement à un décalage de sept *bit* vers la droite.

Présentation

Comme vous le voyez, si nous gagnons effectivement de la place en mémoire, nous y perdons en temps de calcul et, plus important, notre code est nettement plus complexe. C'est la raison pour laquelle cette méthode n'est employée que dans le cas de contraintes particulières.

Bien entendu, nous pourrions recourir à des fonctions ou à des macrofonctions pour simplifier la lecture du code, toutefois, nous ne ferions que reporter la difficulté de compréhension sur ces dernières. Heureusement, en vue de simplifier ce type d'écritures, le langage C met à notre disposition les **champs de bits**.

Un champ de *bits* est une structure *composée exclusivement* de champs de type `int` ou `unsigned int` dont la taille en *bits* de chacun est précisée. Cette taille *ne peut être* supérieure à la taille en *bits* du type `int`. L'exemple ci-dessous définit une structure composée de trois champs de *bits*, `a`, `b` et `c` de respectivement un, deux et trois *bits*.

```

1 struct champ_de_bits
2 {
3     unsigned a : 1;
4     unsigned b : 2;
5     unsigned c : 3;
6 };

```

Ainsi, notre exemple précédent peut être réécrit comme ceci.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4
5  struct date {
6      unsigned jour : 5;
7      unsigned mois : 4;
8      unsigned annee : 12;
9  };
10
11
12  int
13  main(void)
14  {
15      struct date date;
16      unsigned jour, mois, annee;
17
18      printf("Entrez une date au format jj/mm/aaaa : ");
19
20      if (scanf("%u/%u/%u", &jour, &mois, &annee) != 3) {
21          perror("fscanf");
22          return EXIT_FAILURE;
23      }
24
25      date.jour = jour;
26      date.mois = mois;
27      date.annee = annee;
28
29      printf("Le %u/%u/%u\n", date.jour, date.mois, date.annee);
30      return 0;
31  }

```



Les champs de *bits* ne disposent pas d'une adresse et ne peuvent en conséquence se voir appliquer l'opérateur d'adressage. Par ailleurs, nous vous conseillons de ne les employer que pour stocker des nombres non signés, le support des nombres signés n'étant pas garanti par la norme.

Si vous avez poussé la curiosité jusqu'à vérifier la taille de cette structure, il y a de forte chance pour que celle-ci équivaille à celle du type `int`. En fait, il s'agit de la méthode la plus courante pour conserver les champs de *bits* : ils sont stockés dans des suites de `int`. Dès lors, si vous souhaitez économiser de la place, faites en sorte que les données à stocker coïncident le plus possible avec la taille d'un ou plusieurs objets de type `int`.



Gardez à l'esprit que le compactage de données et les champs de *bits* répondent à des besoins particulier et complexifient votre code. Dès lors, ne les utilisez que lorsque cela semble réellement se justifier.

28.3 Les drapeaux

Une autre utilisation régulière des opérateurs de manipulation des *bits* consiste en la gestion des **drapeaux**. Un drapeau correspond en fait à un *bit* qui est soit « levé », soit « baissé » dans l'objectif d'indiquer si une situation est vraie ou fausse.

Supposons que nous souhaitions fournir à une fonction un nombre et plusieurs de ses propriétés, par exemple s'il est pair, s'il s'agit d'une puissance de deux et s'il est premier. Nous pourrions bien entendu lui fournir quatre paramètres, mais cela fait finalement beaucoup pour simplement fournir un nombre et, foncièrement, trois *bits*.

À la place, il nous est possible d'employer un entier dont trois *bits* seront utilisés pour représenter chaque condition. Par exemple, le premier indiquera si le nombre est pair, le second s'il s'agit d'une puissance de deux et le troisième s'il est premier.

```

1 void traitement(int nombre, unsigned char drapeaux)
2 {
3     if (drapeaux & 0x01) /* Si le nombre est pair */
4     {
5         /* ... */
6     }
7     if (drapeaux & 0x02) /* Si le nombre est une puissance de deux */
8     {
9         /*... */
10    }
11    if (drapeaux & 0x04) /* Si le nombre est premier */
12    {
13        /*... */
14    }
15 }
16
17
18 int main(void)
19 {
20     int nombre;
21     unsigned char drapeaux;
22
23     nombre = 2;
24     drapeaux = 0x01 | 0x02; /* 0000 0011 */
25     traitement(nombre, drapeaux);
26     nombre = 17;
27     drapeaux = 0x04; /* 0000 0100 */
28     traitement(nombre, drapeaux);
29     return 0;
30 }
```

Comme vous le voyez, nous utilisons l'opérateur `||` pour combiner plusieurs drapeaux et l'opérateur `&` pour déterminer si un drapeau est levé ou non.



Notez que, chaque drapeau représentant un *bit*, ceux-ci correspondent toujours à des puissances de deux.

Voilà qui est plus efficace, mais en somme assez peu lisible... En effet, il serait bon de préciser dans notre code à quoi correspond chaque drapeaux. Pour ce faire, nous pouvons recourir au préprocesseur afin de clarifier un peu tout cela.

```

1  #define PAIR      (1 << 0)
2  #define PUISSANCE (1 << 1)
3  #define PREMIER  (1 << 2)
4
5
6  void traitement(int nombre, unsigned char drapeaux)
7  {
8      if (drapeaux & PAIR) /* Si le nombre est pair */
9      {
10         /* ... */
11     }
12     if (drapeaux & PUISSANCE) /* Si le nombre est une puissance de deux */
13     {
14         /*... */
15     }
16     if (drapeaux & PREMIER) /* Si le nombre est premier */
17     {
18         /*... */
19     }
20 }
21
22
23 int main(void)
24 {
25     int nombre;
26     unsigned char drapeaux;
27
28     nombre = 2;
29     drapeaux = PAIR | PUISSANCE; /* 0000 0011 */
30     traitement(nombre, drapeaux);
31     nombre = 17;
32     drapeaux = PREMIER; /* 0000 0100 */
33     traitement(nombre, drapeaux);
34     return 0;
35 }
```

Voici qui est mieux.

Pour terminer, remarquez qu'il s'agit d'un bon cas d'utilisation des champs de *bits*, chacun d'entre eux représentant alors un drapeau.

```

1  struct propriete
2  {
3      unsigned pair : 1;
4      unsigned puissance : 1;
5      unsigned premier : 1;
6  }
7
8
9  void traitement(int nombre, struct propriete prop)
10 {
11     if (prop.pair) /* Si le nombre est pair */
12     {
13         /* ... */
14     }
15     if (prop.puissance) /* Si le nombre est une puissance de deux */
16     {
17         /*... */
18     }
19 }
```

```

18     }
19     if (prop.premier) /* Si le nombre est premier */
20     {
21         /*... */
22     }
23 }
24
25
26 int main(void)
27 {
28     int nombre;
29     struct propriete prop = { 0 }; /* Initialisation à zéro. */
30
31     nombre = 2;
32     prop.pair = 1;
33     prop.puissance = 1;
34     traitement(nombre, prop);
35     memset(&prop, 0, sizeof prop); /* Mise à zéro. */
36     nombre = 17;
37     drapeaux = PREMIER; /* 0000 0100 */
38     traitement(nombre, drapeaux);
39     return 0;
40 }

```

28.4 Exercices

28.4.1 Obtenir la valeur maximale d'un type non signé

Maintenant que nous connaissons la représentation des nombres non signés ainsi que les opérateurs de manipulation des *bits*, vous devriez pouvoir trouver comment obtenir la plus grande valeur représentable par le type `unsigned int`.

Indice

Rappelez-vous : dans la représentation des entiers non signés, chaque *bit* représente une puissance de deux.

Solution

```

1  #include <stdio.h>
2
3
4  int main(void)
5  {
6      printf("%u\n", ~0U);
7      return 0;
8  } Cette technique n'est valable que pour les entiers
9  non signés, la représentation où tous les bits sont à un
10 étant potentiellement invalide dans le cas des entiers signés
11 (représentation en complément à un).

```



Notez bien que nous avons utilisé le suffixe `U` afin que le type de la constante `0` soit `unsigned int` et non `int` (n'hésitez pas à revoir le chapitre relatif aux opérations mathématiques si cela ne vous dit rien).



Cette technique n'est valable que pour les entiers *non signés*, la représentation où tous les *bits* sont à un étant potentiellement invalide dans le cas des entiers signés (représentation en complément à un).

28.4.2 Afficher la représentation en base deux d'un entier

Vous le savez, il n'existe pas de format de la fonction `printf()` qui permet d'afficher la représentation binaire d'un nombre. Pourtant, cela pourrait s'avérer bien pratique dans certains cas, même si la représentation hexadécimale est disponible.

Dans ce second exercice, votre tâche sera de réaliser une fonction capable d'afficher la représentation binaire d'un `unsigned int` en *gros-boutiste*.

Indice

Pour afficher la représentation gros-boutiste, il va vous falloir commencer par afficher le *bit* de poids de fort suivit des autres. Pour ce faire, vous allez avoir besoin d'un masque dont seul ce *bit* sera à un. Pour ce faire, vous pouvez vous aider de l'exercice précédent.

Solution

```

1  #include <stdio.h>
2
3
4  void affiche_bin(unsigned n)
5  {
6      unsigned mask = ~(~0U >> 1);
7      unsigned i = 0;
8
9      while (mask > 0)
10     {
11         if (i != 0 && i % 4 == 0)
12             putchar(' ');
13
14         putchar((n & mask) ? '1' : '0');
15         mask >>= 1;
16         ++i;
17     }
18
19     putchar('\n');
20 }
21
22
23 int main(void)
24 {
25     affiche_bin(1);
26     affiche_bin(42);
27     return 0;
28 }
```

```

1  0000 0000 0000 0000 0000 0000 0000 0001
2  0000 0000 0000 0000 0000 0000 0010 1010
```

L'expression `~(~0U >> 1)` nous permet d'obtenir un masque où seul le *bit* de poids fort est à un. Nous pouvons ensuite l'employer successivement en décalant le *bit* à un de sorte d'obtenir la représentation binaire d'un entier *non signé*.



À nouveau, faites bien attention que ceci n'est valable que pour les entiers *non signés*, une représentation dont tous les *bits* sont à un ou dont seul le *bit* de poids fort est à un étant possiblement incorrecte dans le cas des entiers signés.

28.4.3 Déterminer si un nombre est une puissance de deux

Vous le savez : les puissances de deux ont la particularité de n'avoir qu'un seul bit à un, tous les autres étant à zéro. Toutefois, elles ont une autre propriété : si l'on soustrait un à une puissance de deux n , tous les *bits* précédents celui de la puissance seront mis à un (par exemple `0000 1000 - 1 == 0000 0111`). En particulier, on remarque que n et $n - 1$ n'ont aucun bit à 1 en commun. Réciproquement, si n n'est pas une puissance de 2, alors le bit à 1 le plus fort est aussi à 1 dans $n - 1$. par exemple `0000 1010 - 1 == 0000 1001`).

Sachant cela, il nous est possible de créer une fonction très simple déterminant si un nombre est une puissance de 2 ou non.

```

1  #include <stdio.h>
2
3
4  int puissance_de_deux(unsigned int n)
5  {
6      return n != 0 && (n & (n - 1)) == 0;
7  }
8
9
10 int main(void)
11 {
12     if (puissance_de_deux(256))
13         printf("256 est une puissance de deux\n");
14     else
15         printf("256 n'est pas une puissance de deux\n");
16
17     if (puissance_de_deux(48))
18         printf("48 est une puissance de deux\n");
19     else
20         printf("48 n'est pas une puissance de deux\n");
21
22     return 0;
23 }
```

```

1  256 est une puissance de deux
2  48 n'est pas une puissance de deux
```

En résumé

1. Le C fournit six opérateurs de manipulation des *bits* ;
2. Ces derniers travaillant directement sur la représentation des entiers, il est *impératif* d'éviter d'obtenir des représentations potentiellement invalides dans le cas des entiers signés ;
3. L'utilisation de masques permet de sélectionner ou modifier une portion d'un nombre entier ;
4. Les champs de *bits* permettent de stocker des entiers de taille arbitraire, mais doivent *toujours* avoir une taille inférieure à celle du type `int`. Par ailleurs, ils n'ont pas d'adresse et ne supportent pas forcément le stockage d'entiers signés ;
5. Les drapeaux constituent une solution élégante pour stocker des états binaires (« vrai » ou « faux »).

Jeux de caractères et encodages

Lorsque nous vous avons présenté les chaînes de caractères, nous vous avons précisé que celle-ci étaient des tableaux de `char` terminé par un caractère nul, en vous laissant sous-entendre qu'à chaque « caractère » (`'a'`, `','`, `'1'`, etc.) correspondait un `char`. Toutefois, ce n'est pas tout à fait exacte ou, plus précisément, ce n'est pas toujours vrai.

Dans ce chapitre, nous allons découvrir comment les chaînes de caractères sont réellement représentées et pour quelles raisons cette représentation est susceptible de varier.

29.1 Les jeux de caractères et les encodages

29.1.1 Introduction



Avant de poursuivre ce chapitre, nous vous invitons à lire au moins les deux premiers chapitres du [cours de Maëlan sur les encodages](#). Ceux-ci constitueront une base solide sur laquelle nous nous appuierons dans la suite.

29.1.2 Ce que dit la norme

Maintenant que les notions de jeux de caractères et d'encodages vous sont connues, voyons comment celles-ci s'agencent en C. La norme définit deux jeux de caractères¹ :

1. Le jeu de caractères source qui, comme son nom ne l'indique pas, correspond au jeu de caractères vers lequel votre fichier source va être converti. Il ne s'agit donc pas de l'encodage de votre fichier source (que vous déterminez à l'aide de votre éditeur de texte), mais d'un encodage interne au compilateur.
2. Le jeu de caractères d'exécution qui correspond à celui utilisé par votre système (il est par exemple utilisé par votre console) vers lequel votre programme sera finalement traduit. Ce dernier dépend de la *locale* employée (nous y reviendrons un peu plus tard dans ce chapitre).



Autrement dit, il y a possiblement deux conversions lors de la compilation : une qui a lieu du jeu de caractères de vos fichiers sources vers le jeu employé en interne par le compilateur et une depuis le jeu de caractères du compilateur vers celui du système.

1. Programming Language C, X3J11/88-090, § 2.2.1, Character sets

29.1.3 Caractères garantis

La norme précise que ces deux jeux comprennent au minimum les caractères suivants.

1	A	B	C	D	E	F	G	H	I	J	K	L	M
2	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
3	a	b	c	d	e	f	g	h	i	j	k	l	m
4	n	o	p	q	r	s	t	u	v	w	x	y	z
5	0	1	2	3	4	5	6	7	8	9			
6	!	"	#	%	&	'	()	*	+	,	-	.
7	;	<	=	>	?	[^]	~	{		}	~

À ceux-ci s'ajoutent l'espace, les tabulations horizontales et verticales et le saut de page.

Il est également précisé que les points de codes des dix chiffres ('0', '1', '2', '3', '4', '5', '6', '7', '8' et '9') doivent se suivre de manière croissante.

Jeu de caractère source

De plus, le jeu de caractères source doit comprendre un ou plusieurs caractères permettant d'indiquer la fin d'une ligne de texte.

Jeu de caractère d'exécution

Enfin, le jeu de caractères d'exécution doit comprendre le caractère nul, le caractère d'appel, l'espacement arrière, le retour chariot et le saut de ligne.

Certes, mais encore ?



C'est super cette description détaillée, mais ça m'apporte quoi de savoir ça ? J'en fais quoi, moi, de vos deux jeux ?

Pour résumer, la norme nous décrit ici les caractères qui peuvent être employés dans vos fichiers sources et les caractères que votre système doit supporter au minimum.

Ceci est impératif pour assurer d'une part la compilation de vos programmes et la bonne exécution de ceux-ci. En effet, imaginer par exemple que le compilateur utilise en interne un jeu de caractères ne comprenant pas le caractère `p`, vous serez bien ennuyé ensuite pour faire appel à `printf()`... De même, il serait fort gênant que votre console ne sache pas afficher les retours à la ligne.

Toutefois, cela a également une seconde conséquence : si vous utilisez un autre caractère que ceux énumérés ci-dessus (par exemple un « e » accent, un caractère cyrillique ou un idéogramme japonais), la norme ne vous garantit pas d'une part que la compilation réussisse (la conversion du jeu utilisé par vos fichiers sources vers celui du compilateur pourrait par exemple échouer) et, d'autre part, que ceux-ci seront supportés par votre système (si ce n'est pas le cas, cela se traduira le plus souvent par un affichage chaotique).

Aussi, dans un souci de portabilité, il est nécessaire de se contenter le plus souvent de ces derniers, ce qui exclut donc l'emploi (ou à tout le moins l'emploi correct) de la plupart des langues du monde à l'exception de l'anglais et du latin.



1. Cette restriction doit toutefois être relativisée puisque la plupart des compilateurs utilisent l'UTF-8 en interne, de même que les systèmes d'exploitations modernes. De plus, cette restriction n'a pas d'objet pour les commentaires puisque ceux-ci sont ignorés lors de la compilation.
2. Par ailleurs, sachez qu'il existe pas mal de solutions permettant de contourner cette limitation. À ce sujet, si cela vous intéresse, nous vous recommandons [GNU gettext](#) qui est une solution libre et gratuite très utilisée sous GNU/Linux et *BSD.



Les plus attentifs (ou fourbes, c'est selon) d'entre-vous se rappelleront sans doute que plusieurs des codes présentés dans ce cours comportent des caractères accentués et... oui, nous n'avons pas respecté la norme sur ce point. Mais bon, un exemple n'est-il pas plus agréable avec ?

29.2 Les caractères larges

29.2.1 Mise en situation

Si nous devons tâcher de respecter la norme en n'employant pas de caractères en dehors de ceux garantis, cela n'est pas vrai pour les utilisateurs de nos programmes qui, eux, ne se gêneront pas pour utiliser ceux supportés par leur système (c'est d'ailleurs bien là l'intérêt de choisir la langue de son système).

Or, jusqu'à présent, nous sommes toujours partis du principe que les caractères entrés tenaient sur un seul `char`, ce qui n'est pas toujours vrai, comme vous avez pu le voir au travers du cours de Maëlan.

En fait, il s'agit du comportement *par défaut* des fonctions de la bibliothèque standard. Chaque `char` est supposé représenter un caractère et une chaîne de caractères est censée n'être qu'une suite de `char` finie par un zéro.

Il est possible de s'en rendre compte à l'aide du code suivant.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5
6  int main(void)
7  {
8      char chaine[255];
9      char *nl;
10
11     if (fgets(chaine, sizeof chaine, stdin) == NULL)
12     {
13         perror("fgets");
14         return EXIT_FAILURE;
15     }
16
17     nl = strchr(chaine, '\n');
18
19     if (nl != NULL)
20         *nl = '\0';
21
22     printf("Longueur : %u\n", (unsigned)strlen(chaine));
23     return 0;
24 }
```

```

1  Bonjour
2  Longueur : 7
3
4  Élégamment trouvé
5  Longueur : 19
```

Comme vous le voyez, la taille de la chaîne « Élégamment trouvé » n'est pas celle attendue dans notre cas (notre exemple emploie l'UTF-8 comme encodage) car ce sont les multipliets (les `char`, donc) qui ont été comptés et non les caractères. La bonne réponse aurait dû être 16.

29.2.2 Les caractères larges

Afin de résoudre ce problème, la norme C89 a introduit les **caractères larges**. Pour ce faire, un nouveau type a été introduit : le type `wchar_t` (pour *wide character*), défini dans l'en-tête `<stddef.h>`. Celui-ci n'est rien d'autre qu'un type entier (signé ou non signé) capable de représenter le point de code le plus élevé supporté par le système.

L'objectif recherché est de traduire une chaîne de caractères classique recourant à un encodage avec un nombre variable de multipliets (comme l'UTF-8 ou l'ISO-2022) vers une chaîne de caractères larges dont chacun représentera exactement un caractère. Dans le cas de chaînes de caractères en UTF-8 par exemple, celles-ci seront le plus souvent converties en UTF-16 ou en UTF-32.

À cet effet, plusieurs fonctions de conversions sont mises à notre disposition et sont définies dans l'en-tête `<stdlib.h>`. Toutefois, aucune fonction de traitement de ces chaînes n'est fournie, c'est-à-dire que celles-ci doivent être manipulées « à la main » (il n'y a donc par exemple pas de fonction du type `strlen()` qui manipule une chaîne de `wchar_t`).



En vérité, les normes suivantes du langage C (à commencer par un amendement adopté en 1994) ont ajouté des fonctions de traitements des chaînes de caractères larges ainsi que d'autres fonctions de conversions dites « réentrantes » (c'est-à-dire qui peuvent être appelées simultanément par plusieurs fils d'exécutions ou *threads* en anglais). Toutefois, nous ne les aborderons pas dans ce cours, d'une part parce que celui-ci se fonde sur la norme C89 et, d'autre part, parce que leur présentation mériterait plusieurs chapitres à elle seule.

Les fonctions `mbtowc` et `wctomb`

```
1 int mbtowc(wchar_t *destination, const char *chaine, size_t max);
2 int wctomb(char *destination, wchar_t source);
```

La fonction `mbtowc()` (pour *multibyte character to wide character*) convertit une suite de multipliets en un caractère large (qu'elle stocke dans l'objet référencé par `destination`) en lisant au plus `max` multipliets depuis la chaîne `source`. Elle retourne le nombre de multipliets utilisés pour produire le caractère large ou `-1` en cas d'erreur.

La fonction `wctomb()` (pour *wide character to multibyte character*) effectue l'opération inverse : elle convertit le caractère large `source` en une suite de multipliets qui sera stockée dans le tableau `destination`. Elle retourne le nombre de multipliets produits en cas de succès et `-1` en cas d'erreur.

La fonction `mbtowc`

L'exemple ci-dessous lit une ligne depuis l'entrée standard et convertit la première suite de multipliets représentant un caractère en un caractère large.

```
1 #include <locale.h>
2 #include <stddef.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <string.h>
6
7
8 int main(void)
9 {
10     char chaine[255];
11     wchar_t wc;
12     char *nl;
13     int n;
14
15     if (setlocale(LC_CTYPE, "") == NULL)
```

```

16     {
17         perror("setlocale");
18         return EXIT_FAILURE;
19     }
20     if (fgets(chaine, sizeof chaine, stdin) == NULL)
21     {
22         perror("fgets");
23         return EXIT_FAILURE;
24     }
25
26     nl = strchr(chaine, '\n');
27
28     if (nl != NULL)
29         *nl = '\0';
30
31     n = mbtowlc(&wc, chaine, MB_CUR_MAX);
32
33     if (n <= 0)
34     {
35         perror("mbtowlc");
36         return EXIT_FAILURE;
37     }
38
39
40     printf("%d multiplet(s) a(ont) été lu pour produire la valeur %u.\n", \
41           n, (unsigned)wc);
42     return 0;
43 }

```

```

1  Élégamment trouvé
2  2 multiplet(s) a(ont) été lu pour produire la valeur 201.
3
4  ASCII
5  1 multiplet(s) a(ont) été lu pour produire la valeur 65.

```



Les résultats obtenus dépendent bien entendu du jeu de caractères utilisé par votre système. Si ce dernier n'emploie pas l'Unicode ou n'utilise pas l'UTF-8 comme encodage, la sortie du programme peut être différente.

Comme vous le voyez, étant donné que notre système emploie de l'UTF-8, deux multipléts ont été lus depuis la chaîne `chaine` pour construire la valeur du caractère `É`, soit 201 (qui correspond à son point de code dans le jeu de caractères Unicode). Le caractère `A` étant quant à lui représenté sur un seul multiplet en UTF-8, la lecture d'un seul suffit pour obtenir sa valeur, à savoir 65.

L'en-tête `<locale.h>` a été ajouté en vue d'utiliser la fonction `setlocale()` dont nous parlerons dans la prochaine section. Sachez pour l'instant qu'elle doit être appelée *avant* d'utiliser les fonctions de conversions.

`MB_CUR_MAX` est une macro (elle est éfinie dans l'en-tête `<stdlib.h>`) dont la valeur est déterminée par la *locale* courante (nous y reviendrons lorsque nous aborderons la fonction `setlocale()`) et correspond au nombre maximum de multipléts nécessaires pour construire un caractère large dans la locale actuelle.

La fonction `wctomb`

Comme nous vous l'avons dit, la fonction `wctomb()` effectue l'exact inverse de la fonction `mbtowc()`. L'exemple suivant tente de convertir le caractère large `É` en la suite de multipléts correspondante.

```

1  #include <limits.h>
2  #include <locale.h>
3  #include <stddef.h>
4  #include <stdio.h>
5  #include <stdlib.h>
6
7

```

```

8  int main(void)
9  {
10     char tab[MB_LEN_MAX];
11     int i;
12     int n;
13
14     if (setlocale(LC_CTYPE, "") == NULL)
15     {
16         perror("setlocale");
17         return EXIT_FAILURE;
18     }
19
20     n = wctomb(tab, L'É');
21
22     if (n <= 0)
23     {
24         perror("wctomb");
25         return EXIT_FAILURE;
26     }
27
28     for (i = 0; i < n; ++i)
29         printf("%x ", (unsigned char)tab[i]);
30
31     putchar('\n');
32     return 0;
33 }

```

1 c3 89

Ici, nous essayons de convertir le caractère large `É` (notez qu’une constante de type caractère large peut être définie en la faisant précédé de la lettre `U` ou `L`) en une suite de multiplète qui sera stockée dans le tableau `tab`.

La taille du tableau `tab` a été fixée à `MB_LEN_MAX`, une macroconstante définie dans l’en-tête `<limits.h>` correspondant à la plus grande suite de multiplète pouvant représenter un caractère sur le système. Notez bien la différence avec la macro `MB_CUR_MAX` qui se limite à la *locale* courante. Par ailleurs, la valeur de `MB_CUR_MAX` dépendant des appels à la fonction `setlocale()`, elle ne peut être utilisée pour déterminer la taille d’un tableau lors de sa définition.

Une fois la conversion effectuée, nous affichons la valeur des différents multiplète en hexadécimal (remarquez la conversion en `unsigned char` afin d’éviter l’affichage de nombres négatifs).



Dans le cas où vous insérez une constante de type caractère large dans votre code source comme `L'É'`, les mêmes restrictions s’appliquent que pour les caractères simples : si le caractère en question ne fait pas partie du jeu de caractères source ou d’exécution, le résultat n’est pas déterminé par la norme. Une telle pratique est donc à proscrire également, pour les mêmes motifs que précédemment.

Convertir une chaîne complète

Bien entendu, ces deux fonctions peuvent être utilisées en combinaison avec une boucle en vue de convertir une chaîne de caractères entière (c’est même là tout l’intérêt de la chose).

```

1  #include <locale.h>
2  #include <stddef.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <string.h>
6
7
8  int main(void)
9  {
10     char mbs[255];

```

```

11  wchar_t wcs[255];
12  char *nl;
13  char *pc;
14  int n;
15  int i;
16
17  if (setlocale(LC_CTYPE, "") == NULL)
18  {
19      perror("setlocale");
20      return EXIT_FAILURE;
21  }
22  if (fgets(mbs, sizeof mbs, stdin) == NULL)
23  {
24      perror("fgets");
25      return EXIT_FAILURE;
26  }
27
28  nl = strchr(mbs, '\n');
29
30  if (nl != NULL)
31      *nl = '\0';
32
33  pc = mbs;
34
35  for (i = 0; (n = mbtowc(&wcs[i], pc, MB_CUR_MAX)) > 0; ++i)
36  {
37      if (*pc == '\0')
38          break;
39
40      pc += n;
41  }
42
43  if (*pc != '\0')
44  {
45      perror("mbtowc");
46      return EXIT_FAILURE;
47  }
48
49  for (i = 0; wcs[i] != L'\0'; ++i)
50      printf("%u ", (unsigned)wcs[i]);
51
52  putchar('\n');
53  return 0;
54 }

```

```

1  Éléphant trouvé
2  201 108 233 103 97 109 101 110 116 32 116 114 111 117 118 233

```

Comme vous le voyez, nous appelons la fonction `mbtowc()` tant que celle-ci ne rencontre pas une erreur ou que nous ne rencontrons pas le caractère de fin de chaîne (ce dernier devant également être converti, cette seconde condition est placée au sein de la boucle). Ensuite, suivant le nombre de multipliets lus par `mbtowc()`, nous augmentons la valeur du pointeur `pc` afin de référencer les prochains caractères à lire. À la sortie de la boucle, nous vérifions que le pointeur `pc` pointe bien vers le caractère nul sans quoi cela signifie que la fonction `mbtowc()` a rencontré une erreur. Enfin, nous parcourons la chaîne large `wcs` pour afficher les différents points de code des caractères la composant.



Notez que comme la comparaison `wcs[i] != L'\0'` porte sur le caractère large `wcs[i]`, nous avons fait du second opérande un caractère large également. Le caractère nul étant garanti de faire partie du jeu de caractères d'exécution, cela ne pose pas de problèmes

Les fonction mbstowcs et wctombs

```
1 size_t mbstowcs(wchar_t *destination, char *source, size_t max);
2 size_t wctombs(char *destination, wchar_t *source, size_t max);
```

C'est chouette de pouvoir employer des boucles, mais cela reste assez fastidieux... Heureusement pour nous, il existe des fonctions qui se chargent de le faire pour nous. ^^

Les fonctions `mbstowcs()` et `wctombs()` convertissent une chaîne de caractères vers une chaîne de caractères larges et inversement. Elles stockent les `max` premiers caractères produits dans la chaîne `destination`.

Les deux fonctions retournent le nombre de caractères convertis (hors caractère nul final) ou `(size_t)-1` en cas d'erreurs.

Ci-dessous, un exemple de conversion recourant à la fonction `mbstowcs()` qui nous permet de rendre notre programme initial correct. Notez que nous avons dû réaliser notre propre version de `strlen()` afin de calculer la longueur de la chaîne de caractères larges obtenues.

```
1 #include <locale.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4
5
6 static size_t
7 wchar_len(wchar_t *wcs)
8 {
9     size_t i = 0;
10
11     while (wcs[i] != L'\0')
12         ++i;
13
14     return i;
15 }
16
17
18 int main(void)
19 {
20     wchar_t wcs[255];
21     char mbs[255];
22     char *nl;
23
24     if (setlocale(LC_CTYPE, "") == NULL)
25     {
26         perror("setlocale");
27         return EXIT_FAILURE;
28     }
29     if (fgets(mbs, sizeof mbs, stdin) == NULL)
30     {
31         perror("fgets");
32         return EXIT_FAILURE;
33     }
34
35     nl = strchr(mbs, '\n');
36
37     if (nl != NULL)
38         *nl = '\0';
39
40     if (mbstowcs(wcs, mbs, sizeof mbs) == (size_t)-1)
41     {
42         perror("mbstowcs");
43         return EXIT_FAILURE;
44     }
45
46     printf("Nombre de multiplats : %u\n", (unsigned)strlen(mbs));
47     printf("Nombre de caractères : %u\n", (unsigned)wchar_len(wcs));
48     return 0;
49 }
```

```
1 Élégamment trouvé
```

```

2 Nombre de multipléts : 19
3 Nombre de caractères : 16

```

La fonction mblen

```

1 int mblen(char *chaine, size_t max);

```

La fonction `mblen()` lit au plus `max` multipléts de la chaîne `chaine`. Si ceux-ci forment un caractère large valide, elle retourne le nombre de multipléts qui seront utilisés pour le composer. Dans le cas contraire, elle retourne soit `0` (si le premier caractère est le caractère nul) ou `-1` (la suite de multipléts ne correspond pas à un caractère large).

```

1 #include <locale.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4
5
6 int
7 main(void)
8 {
9     int n;
10
11     if (setlocale(LC_CTYPE, "") == NULL)
12     {
13         perror("setlocale");
14         return EXIT_FAILURE;
15     }
16
17     n = mblen("Élégamment trouvé", MB_CUR_MAX);
18
19     if (n > 0)
20         printf("Le prochain caractère large sera composé de %d multipléts(s).\n", n);
21
22     return 0;
23 }

```

```

1 Le prochain caractère large sera composé de 2 multipléts(s).

```

29.2.3 Des fonctions dans tous leurs états

Il est important de vous préciser une chose en rapport avec ces fonctions : elles disposent d'un **état** interne. En vérité, ceci est essentiellement important dans le cas où vous utilisez un jeu de caractères avec état comme l'ISO-2022-JP. Dans un tel cas, les fonctions de conversions doivent mémoriser la dernière séquence d'échappement afin d'effectuer correctement leur travail, ce qu'elles réalisent à l'aide de variables internes. Or, si une erreur est rencontrée ou si une autre suite de multipléts leur est donnée en cours de route, le résultat risque de s'en trouver compromis.

Dès lors, il est nécessaire de réinitialiser cet état interne après la rencontre d'une erreur ou lors d'un changement de données à traiter. Cela se réalise très simplement en fournissant comme argument une chaîne nulle à l'une des fonctions `mbtowc()`, `wctomb()` ou `mblen()`. Dans un tel cas, l'état interne est remis à zéro et une valeur nulle est retournée si le jeu de caractères courant n'utilise pas d'état et un nombre strictement positif sinon.###

29.3 Internationalisation et localisation

Cela est resté finalement assez discret jusqu'à ce chapitre, mais en y regardant de plus près, les programmes que nous avons réalisés sont en fait destinés à un environnement anglophone. En effet, prenez par exemple les entrées : si nous souhaitons fournir un nombre flottant à notre programme, nous devons utiliser le point comme séparateur entre la partie entière et décimale.

Or, dans certains pays, on pourrait vouloir utiliser la virgule à la place. Cela nous paraît moins étrange étant donné que les constantes flottantes sont écrites de cette manière en C, mais il n'en va pas de même pour nos utilisateurs.

Ce qu'il faudrait finalement, c'est que nos programmes puissent s'adapter aux usages, coutumes et langues de notre utilisateur, ce que nous avons entrevu dans la section précédente.

29.3.1 L'internationalisation

L'**internationalisation** (parfois abrégée « i18n ») est un procédé par lequel un programme est rendu capable de s'adapter aux préférences linguistiques et régionales d'un utilisateur.

29.3.2 La localisation

La **localisation** (parfois abrégée « l10n ») est une opération par laquelle un programme internationalisé se voit fournir les informations nécessaires pour s'adapter aux préférences linguistiques et régionales d'un utilisateur.

29.3.3 La fonction `setlocale`

De manière générale, les programmes que nous avons conçus jusqu'ici étaient déjà partiellement internationalisés, car la bibliothèque standard du langage C l'est dans une certaine mesure. Toutefois, nous n'avons jamais recouru à un processus de localisation pour que ceux-ci s'adaptent à nos usages. Nous vous le donnons en mille : la localisation en C s'effectue à l'aide de... la fonction `setlocale()`.

```
1 char *setlocale(int categorie, char *localisation);
```

Cette fonction attends deux arguments : une catégorie et la localisation qui doit être employée pour cette catégorie.

Les catégories

La bibliothèque standard du C divise la localisation en plusieurs catégories, plus précisément cinq :

1. La catégorie `LC_COLLATE` qui modifie le comportement des fonctions `strcoll()` et `strxfrm()` ;
2. La catégorie `LC_CTYPE` qui adapte le comportement des fonctions de conversions que nous venons de voir, ainsi que les fonctions de l'en-tête `<ctype.h>` ;
3. La catégorie `LC_MONETARY` qui influence le comportement de la fonction `localeconv()` ;
4. La catégorie `LC_NUMERIC` qui altère le comportement des fonctions `*printf()` et `*scanf()` ainsi que des fonctions de conversions des chaînes de caractères (`atof()`, `strtod()`, etc.) en ce qui concerne les nombres flottants ;
5. La catégorie `LC_TIME` qui change le comportement de la fonction `strftime()`.

Enfin, la catégorie `LC_ALL` (qui n'en est pas vraiment une) représente toutes les catégories en même temps. Nous ne attarderons toutefois que sur `LC_CTYPE` et `LC_NUMERIC` dans cette section, les fonctions affectées par les autres catégories n'ayant pas été présentées à ce stade.

Les localisations

La bibliothèque standard prévoit deux localisations possibles :

1. La localisation `"C"` qui correspond à celle par défaut. Celle-ci utilise les usages anglophones et part du principe que les caractères employés se limitent à ceux garantis par la norme et qu'ils sont tous représentés sur un multipllet ;

2. La localisation `""` (une chaîne vide) qui correspond à celle utilisée par votre système.

Il est également possible de fournir un pointeur nul comme localisation, auquel cas la localisation actuelle est retournée.

```

1  #include <locale.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5
6  int main(void)
7  {
8      char *s;
9
10     s = setlocale(LC_ALL, NULL);
11     puts(s);
12
13     if (setlocale(LC_ALL, "") == NULL)
14     {
15         perror("setlocale");
16         return EXIT_FAILURE;
17     }
18
19     s = setlocale(LC_ALL, NULL);
20     puts(s);
21     return 0;
22 }
```

```

1  C
2  fr_BE.UTF-8
```

Comme vous le voyez, la localisation de départ est bien `C`.



La forme que prend la localisation dépend de votre système. Sous unixoïdes et dans notre exemple, elle prend la forme de la langue en minuscule (au format [ISO 639](#)) suivie d'un tiret bas et du pays en majuscule (au format [ISO 3166-1](#)) et, éventuellement, terminée par un point et par l'encodage utilisé.

29.3.4 Exemple

Nous avons déjà eu l'occasion d'expérimenter la modification de la localisation de la catégorie `LC_CTYPE`. Ainsi, nous avons pu préciser aux fonctions de conversion que la traduction devait s'opérer depuis et vers le jeu de caractères d'exécution complet et non uniquement le sous-ensemble défini par la norme.

L'exemple ci-dessous modifie la localisation de la catégorie `LC_NUMERIC` pour que les fonctions `scanf()` et `printf()` adaptent leur gestion et affichage des nombres flottants.

```

1  #include <locale.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5
6  int main(void)
7  {
8      double f;
9
10     if (setlocale(LC_NUMERIC, "") == NULL)
11     {
12         perror("setlocale");
13         return EXIT_FAILURE;
14     }
15
16     printf("Veuillez entrer un nombre flottant : ");
```

```
17
18     if (scanf("%lf", &f) != 1)
19     {
20         perror("scanf");
21         return EXIT_FAILURE;
22     }
23
24     printf("Vous avez entré : %f.\n", f);
25     return 0;
26 }
```

```
1  Veuillez entrer un nombre flottant : 45,5
2  Vous avez entré : 45,500000.
3
4  Veuillez entrer un nombre flottant : 45.5
5  Vous avez entré : 45,000000.
```

Comme vous le voyez, après l'appel à `setlocale()`, seule la virgule est considérée comme séparateur de la partie entière et de la partie décimale.

En résumé

1. La norme définit deux jeux de caractères : le jeu de caractère source, utilisé en interne par le compilateur, vers lequel vos fichiers sources sont convertis et le jeu de caractère d'exécution, utilisé par votre système, vers lequel la conversion finale aura lieu.
2. Mise à part les caractères garantis par la norme, les caractères supportés par les jeux de caractères source et d'exécution sont indéterminés.
3. Le type `wchar_t` et les fonctions de conversion associées permettent de construire des chaînes de caractères larges à partir de chaîne simple, ce qui est particulièrement utile lorsque le jeu employé par le système encode les caractères sur un nombre variable de multipliants (comme l'ISO-2022 ou l'UTF-8).
4. Ces fonctions de conversions disposent d'un état interne qui doit être réinitialisé après une erreur ou avant chaque changement de données à traiter.
5. La fonction `setlocale()` permet de modifier la localisation de certaines fonctions de la bibliothèque standard.

Jusqu'à présent, nous avons toujours employé le préprocesseur pour définir des constantes au sein de nos codes. Toutefois, une solution un peu plus commune existe pour les constantes entières : les **énumérations**.

30.1 Définition

Une énumération se définit à l'aide du mot-clé `enum` suivi du nom de l'énumération et de ses membres.

```
1 enum naturel { ZERO, UN, DEUX, TROIS, QUATRE, CINQ };
```

La particularité de cette définition est qu'elle crée en vérité deux choses : un type dit « énuméré » `enum naturel` et des constantes dites « énumérées » `ZERO`, `UN`, `DEUX`, etc. Le type énuméré ainsi produit peut être utilisé de la même manière que n'importe quel autre type. Quant aux constantes énumérées, il s'agit de constantes entières.

Certes me direz-vous, mais que valent ces constantes ? *Eh* bien, à défaut de préciser leur valeur, chaque constante énumérée se voit attribuer la valeur de celle qui la précède augmentée de un, sachant que la première constante est mise à zéro. Dans notre cas donc, la constante `ZERO` vaut zéro, la constante `UN` un et ainsi de suite jusqu'à cinq.

L'exemple suivant illustre ce qui vient d'être dit.

```
1 #include <stdio.h>
2
3 enum naturel { ZERO, UN, DEUX, TROIS, QUATRE, CINQ };
4
5
6 int main(void)
7 {
8     enum naturel n = ZERO;
9
10    printf("n = %d.\n", (int)n);
11    printf("UN = %d.\n", UN);
12    return 0;
13 }
```

```
1 n = 0.
2 UN = 1.
```



Notez qu'il n'est pas obligatoire de préciser un nom lors de la définition d'une énumération. Dans un tel cas, seules les constantes énumérées sont produites.



```
enum { ZERO, UN, DEUX, TROIS, QUATRE, CINQ };
```

Toutefois, il est possible de préciser la valeur de certaines constantes (voire de toutes les constantes) à l'aide d'une affectation.

```
1 enum naturel { DIX = 10, ONZE, DOUZE, TREIZE, QUATORZE, QUINZE };
```

Dans un tel cas, la règle habituelle s'applique : les constantes sans valeur se voit attribuer celle de la constante précédente augmentée de un et celle dont la valeur est spécifiée sont initialisées avec celle-ci. Dans le cas ci-dessus, la constante `DIX` vaut donc dix, la constante `ONZE` onze et ainsi de suite jusque quinze. Notez que le code ci-dessous est parfaitement équivalent.

```
1 enum naturel { DIX = 10, ONZE = 11, DOUZE = 12, TREIZE = 13, QUATORZE = 14, QUINZE = 15 };
```

Types entiers sous-jacents

Vous aurez sans doute remarqué que, dans notre exemple, nous avons converti la variable `n` vers le type `int`. Cela tient au fait qu'un type énuméré est un type entier (ce qui est logique puisqu'il est censé stocker des constantes entières), mais que le type sous-jacent n'est pas déterminé (cela peut donc être `char`, `short`, `int` ou `long`) et dépend entre autre des valeurs devant être contenues. Ainsi, une conversion s'impose afin de pouvoir utiliser un format d'affichage correct.

Pour ce qui est des constantes énumérées, c'est plus simple : elles sont toujours de type `int`.

30.2 Utilisation

Dans la pratique, les énumérations servent essentiellement à fournir des informations supplémentaires via le typage, par exemple pour les retours d'erreurs. En effet, le plus souvent, les fonctions retournent un entier pour préciser si leur exécution s'est bien déroulée. Toutefois, indiquer un retour de type `int` ne fourni pas énormément d'information. Un type énuméré prend alors tout son sens.

La fonction `vider_tampon()` du dernier T.P. s'y prêterait par exemple bien.

```
1 enum erreur { E_OK, E_ERR };
2
3
4 static enum erreur vider_tampon(FILE *fp)
5 {
6     int c;
7
8     do
9         c = fgetc(fp);
10    while (c != '\n' && c != EOF);
11
12    return ferror(fp) ? E_ERR : E_OK;
13 }
```

De cette manière, il est plus clair à la lecture que la fonction retourne le statut de son exécution.

Dans la même idée, il est possible d'utiliser un type énuméré pour la fonction `statut_jeu()` (également employée dans la correction du dernier T.P.) afin de décrire plus amplement son type de retour.

```
1 enum statut { STATUT_OK, STATUT_GAGNE, STATUT_EGALITE };
2
3
```

```

4 static enum statut statut_jeu(struct position *pos, char jeton)
5 {
6     if (grille_complete())
7         return STATUT_EGALITE;
8     else if (calcule_nb_jetons_depuis(pos, jeton) >= 4)
9         return STATUT_GAGNE;
10
11     return STATUT_OK;
12 }

```

Dans un autre registre, un type énuméré peut être utilisé pour contenir des drapeaux. Par exemple, la fonction `traitement()` présentée dans le chapitre relatif aux opérateurs de manipulation des *bits* peut être réécrite comme suit.

```

1 enum drapeau {
2     PAIR = 0x00,
3     PUISSANCE = 0x01,
4     PREMIER = 0x02
5 };
6
7
8 void traitement(int nombre, enum drapeau drapeaux)
9 {
10     if (drapeaux & PAIR) /* Si le nombre est pair */
11     {
12         /* ... */
13     }
14     if (drapeaux & PUISSANCE) /* Si le nombre est une puissance de deux */
15     {
16         /*... */
17     }
18     if (drapeaux & PREMIER) /* Si le nombre est premier */
19     {
20         /*... */
21     }
22 }

```

En résumé

1. Sauf si le nom de l'énumération n'est pas renseignée, une définition d'énumération crée un type énuméré et des constantes énumérées;
2. Sauf si une valeur leur est attribuée, la valeur de chaque constantes énumérées est celle de la précédente augmentée de un et celle de la première est zéro.
3. Le type entier sous-jacent à un type énuméré est indéterminé; les constantes énumérées sont de type `int`.

Dans la deuxième partie de ce cours nous vous avons présenté la notion d'agrégat qui recouvrait les tableaux et les structures. Toutefois, nous avons passé sous silence un dernier agrégat plus discret moins utilisé : les **unions**.

31.1 Définition

Une union est, à l'image d'une structure, un regroupement d'objet de type *différents*. La nuance, et elle est de taille, est qu'une union est un agrégat qui ne peut contenir qu'*un seul* de ses membres à la fois. Autrement dit, une union peut accueillir la valeur de n'importe lequel de ses membres, mais un seul à la fois.

Concernant la définition, elle est identique à celle d'une structure si ce n'est que le mot-clé `union` est employé en lieu et place de `struct`.

```
1 union type
2 {
3     int entier;
4     double flottant;
5     void *pointeur;
6     char lettre;
7 };
```

Le code ci-dessus définit une union `type` pouvant contenir un objet de type `int` ou de type `double` ou de type pointeur sur `void` ou de type `char`. Cette possibilité de ne stocker qu'un objet à la fois est traduite par le résultat de l'opérateur `sizeof`.

```
1 #include <stdio.h>
2
3 union type
4 {
5     int entier;
6     double flottant;
7     void *pointeur;
8     char lettre;
9 };
10
11
12 int main(void)
13 {
14     printf("%u.\n", sizeof (union type));
15     return 0;
16 }
```

1 8.

Dans notre cas, la taille de l'union correspond à la taille du plus grand type stocké à savoir les types `void *` et `double` qui font huit octets. Ceci traduit bien l'impossibilité de stocker plusieurs objets à la fois.



Notez que, comme les structures, les unions peuvent contenir des *bits* de bourrage, mais uniquement à leur fin.

Pour le surplus, une union s'utilise de la même manière qu'une structure et l'accès aux membres s'effectue à l'aide des opérateurs `.` et `->`.

31.2 Utilisation

Étant donné leur singularité, les unions sont rarement employées. Leur principal intérêt est de réduire l'espace mémoire utilisé là où une structure ne le permet pas.

Par exemple, imaginez que vous souhaitiez construire une structure pouvant accueillir plusieurs types possibles, par exemple des entiers et des flottants. Vous aurez besoin de trois champs : un indiquant quel type est stocké dans la structure et deux permettant de stocker soit un entier soit un flottant.

```
1 struct nombre
2 {
3     unsigned entier : 1;
4     unsigned flottant : 1;
5     int e;
6     double f;
7 };
```

Toutefois, vous gaspiller ici de la mémoire puisque seul un des deux objets sera stockés. Une union est ici la bienvenue afin d'économiser de la mémoire.

```
1 struct nombre
2 {
3     unsigned entier : 1;
4     unsigned flottant : 1;
5     union
6     {
7         int e;
8         double f;
9     } u;
10 };
```

Le code suivant illustre l'utilisation de cette construction.

```
1 #include <stdio.h>
2
3 struct nombre
4 {
5     unsigned entier : 1;
6     unsigned flottant : 1;
7     union
8     {
9         int e;
10        double f;
11    } u;
12 };
13
14 static void affiche_nombre(struct nombre n)
15 {
16     if (n.entier)
17         printf("%d\n", n.u.e);
18     else if (n.flottant)
19         printf("%f\n", n.u.f);
20 }
```

```

20     printf("%f\n", n.u.f);
21 }
22
23
24 int main(void)
25 {
26     struct nombre a = { 0 };
27     struct nombre b = { 0 };
28
29     a.entier = 1;
30     a.u.e = 10;
31     b.flottant = 1;
32     b.u.f = 10.56;
33
34     affiche_nombre(a);
35     affiche_nombre(b);
36     return 0;
37 }

```

```

1  10
2  10.560000

```

Une autre utilisation fréquente des unions est de permettre de modifier l'alignement d'un objet ou, plus précisément, d'augmenter l'alignement d'un objet. En fait, il s'agit d'une des conséquences de l'union : étant donné qu'elle doit pouvoir stocker n'importe lequel de ses membres, son alignement doit être le plus élevé parmi celui de ses membres.



Pour rappel, l'alignement d'un type peut être connu à l'aide de la macrofonction `offsetof()` et d'un type structure de la forme `struct { char c; type x; };`.

Ainsi, si nous souhaitons aligner un objet de type `int` de la même manière qu'un objet de type `double`, il nous suffit de construire une union qui comprendra les deux types. Le code suivant vérifie ce qui vient d'être dit.

```

1  #include <stddef.h>
2  #include <stdio.h>
3
4  union nombre
5  {
6      int e;
7      double f;
8  };
9
10
11 int main(void)
12 {
13     printf("int : %u\n", (unsigned)offsetof(struct { char c; int n; }, n));
14     printf("union nombre : %u\n", (unsigned)offsetof(struct { char c; union nombre n; }, n));
15     return 0;
16 }

```

```

1  int : 4
2  union nombre : 8

```

Dans la même veine, il est ainsi possible de connaître l'alignement le plus strict parmi les types natifs en construisant une union comportant les types les plus contraignants, à savoir : le type `long`, le type `long double` et le type `void *`.

```

1  #include <stddef.h>
2  #include <stdio.h>
3
4  union align
5  {

```

```
6     long e;  
7     long double f;  
8     void *p;  
9 };  
10  
11  
12 int main(void)  
13 {  
14     printf("union align : %u\n", (unsigned)offsetof(struct { char c; union align n; }, n));  
15     return 0;  
16 }  
  
1 union align : 16
```

Comme pour l'exemple précédent, cette technique peut être utilisée pour imposer l'alignement le plus strict à un objet ayant une contrainte d'alignement plus faible. Gardez bien ceci en mémoire, nous y reviendrons lors du T.P. final.

En résumé

1. Une union est un agrégat regroupant des objets de types différents, mais ne pouvant en stocker qu'un seul à la fois;
2. Comme les structures, les unions peuvent comprendre des *bits* de bourrage, mais uniquement à leur fin;
3. Une union acquiert l'alignement le plus strict parmi ceux de ses membres.

Ce chapitre sera relativement court et pour cause, nous allons aborder un petit point du langage C, mais qui a toute son importance : les **définitions de type**.

32.1 Définition et utilisation

Une définition de type permet, comme son nom l'indique, de définir un type, c'est-à-dire d'en produire un nouveau ou, plus précisément, de créer un *alias* (un synonyme si vous préférez) d'un type existant. Une définition de type est identique à une déclaration de variable, si ce n'est que celle-ci doit être précédée du mot-clé `typedef` (pour *type definition*) et que l'identificateur ainsi choisi désignera un type et non une variable.

Ainsi, le code ci-dessous définit un nouveau type `entier` qui sera un *alias* pour le type `int`.

```
1 typedef int entier;
```

Le synonyme ainsi créé peut être utilisé au même endroit que n'importe quel autre type.

```
1 #include <stdio.h>
2
3 typedef int entier;
4
5
6 entier main(void)
7 {
8     entier a = 10;
9
10    printf("%d.\n", a);
11    return 0;
12 }
```



D'accord, mais cela me sert à quoi de créer un synonyme pour un type existant ?

Les définitions de type permettent en premier lieu de raccourcir certaines écritures, notamment afin de s'affranchir des mots-clés `struct`, `union` et `enum` (bien que, ceci constitue une perte d'information aux yeux de certains).

```
1 #include <stdio.h>
2
3 struct position
4 {
5     int lgn;
6     int col;
```

```

7  };
8
9  typedef struct position position;
10
11
12  int main(void)
13  {
14      position pos;
15
16      pos.lgn = 1;
17      pos.col = 1;
18      printf("%d, %d.\n", pos.lgn, pos.col);
19      return 0;
20  }

```

Notez qu'une définition de type étant une déclaration, il est parfaitement possible de combiner la définition de la structure et la définition de type (comme pour une variable, finalement).

```

1  typedef struct position
2  {
3      int lgn;
4      int col;
5  } position;

```

Également, dans le même sens que ce qui a été dit au sujet des énumérations, une définition de type peut être employée pour donner plus d'informations via le typage. C'est une manière de désigner plus finement le contenu d'un type en ne se contentant pas d'une information plus générale comme « un entier » ou « un flottant ».

Par exemple, nous aurions pu définir un type `ligne` et un type `colonne` afin de donner plus d'information sur le contenu de nos variables.

```

1  typedef short ligne;
2  typedef short colonne;
3
4  struct position
5  {
6      ligne lgn;
7      colonne col;
8  };

```

De même, les définitions de type sont couramment utilisées afin de créer des abstractions. Nous en avons vu un exemple avec l'en-tête `<time.h>` qui définit le type `time_t`. Celui-ci permet de ne pas devoir modifier la fonction `time()` et son utilisation suivant le type qui est sous-jacent. Peu importe que `time_t` soit un entier ou un flottant, la fonction `time()` s'utilise de la même manière.

Enfin, les définitions de type permettent de résoudre quelques points de syntaxe tordus.

En résumé

1. Une définition de type permet de créer un synonyme d'un type existant (en ce compris un autre *alias*);

Les pointeurs de fonction

À ce stade, vous pensiez sans doute avoir fait le tour des pointeurs, que ces derniers n'avaient plus de secrets pour vous et que vous maîtrisiez enfin tous leurs aspects ainsi que leur syntaxe parfois déroutante ? *Eh bien, pas encore !*

Il reste un dernier type de pointeur (et non des moindres) que nous avons vu jusqu'ici : les **pointeurs de fonction**.

33.1 Déclaration et initialisation

Jusqu'à maintenant, nous avons manipulé des pointeurs sur objet, c'est-à-dire des adresses vers des zones mémoires contenant des *données* (des entiers, des flottants, des structures, etc.). Toutefois, il est également possible de référencer des *instructions* et ceci est réalisé en C à l'aide des pointeurs de fonction.

Un pointeur de fonction se définit à l'aide d'une syntaxe mélangeant celle des pointeurs sur tableau et celles des prototypes de fonction. Sans plus attendre, voici ci-dessous la définition d'un pointeur sur une fonction retournant un `int` et attendant un `int` comme argument.

```
1 int (*pf)(int);
```

Comme vous le voyez, il est nécessaire, tout comme les pointeurs sur tableau, d'entourer le symbole `*` et l'identificateur de parenthèses, ici afin d'éviter que cette déclaration ne soit vue comme un prototype et non comme un pointeur de fonction. Autre particularité : le type de retour, le nombre d'arguments et leur type doivent également être spécifiés.

33.1.1 Initialisation



Ok... Et je lui affecte comment l'adresse d'une fonction, moi, à ce machin ? D'ailleurs, elles ont une adresse, les fonctions ?

Oui et comme d'habitude, cela est réalisé à l'aide de l'opérateur `&`. \sim
En fait, dans le cas des fonctions, il n'est pas obligatoire de recourir à cet opérateur, ainsi, les deux syntaxes suivantes sont correctes.

```
1 int (*pf)(int);
2
3 pf = &fonction;
4 pf = fonction;
```


Ceci est dû, à l'image des tableaux, à une conversion implicite : sauf s'il est l'opérande de l'opérateur `&`, un identificateur de fonction est converti en un pointeur sur cette fonction. L'utilisation de l'opérateur `&` est donc facultative, mais elle a le mérite de clarifier un peu les choses. Pour cette raison, nous utiliserons cette syntaxe dans la suite de ce cours.

33.2 Utilisation

33.2.1 Déréférencement

Un pointeur de fonction s'emploie de la même manière qu'un pointeur classique, si ce n'est que l'opérateur `*` et l'identificateur doivent à nouveau être entre parenthèses. Pour le reste, la liste des arguments suit l'expression déréférencée, comme pour un appel de fonction classique.

```

1  #include <stdio.h>
2
3
4  static int triple(int a)
5  {
6      return a * 3;
7  }
8
9
10 int main(void)
11 {
12     int (*pt)(int) = &triple;
13
14     printf("%d.\n", (*pt)(3));
15     return 0;
16 }
```

```

1  9.
```

Toutefois, particularité des fonctions oblige, sachez que le déréférencement *n'est pas nécessaire*. Ceci à cause de la conversion implicite expliquée précédemment : un identificateur de fonction est, sauf s'il est l'opérande de l'opérateur `&`, converti en un pointeur sur cette fonction. L'appel `triple(3)` cache donc en fait un pointeur de fonction qui, comme vous le voyez, n'est *pas* déréférencé.



Heu... Mais pourquoi l'expression `(*pt)(3)` ne provoque-t-elle pas une erreur si c'est un pointeur de fonction qui est nécessaire lors d'un appel ?

Parce que la conversion implicite aura lieu juste après le déréférencement. Eh oui, déréférencé un pointeur de fonction, c'est un peu reculer pour mieux sauter : nous obtenons une expression équivalente à un identificateur de fonction qui sera ensuite convertie en un pointeur de fonction. Les deux expressions suivantes sont donc équivalentes.

```

1  triple(3);
2  (*pt)(3);
```

L'intérêt d'employer le déréférencement est purement syntaxique : cela permet de distinguer des appels effectués via des pointeurs des appels de fonction classiques.

33.2.2 Passage en argument

Comme n'importe quel pointeur, un pointeur de fonction peut être passé en argument d'une autre fonction (c'est d'ailleurs tout l'intérêt de ceux-ci, comme nous le verrons bientôt). Pour ce faire, il vous suffit d'employer la même syntaxe que pour une déclaration.

```

1  #include <stdio.h>
2
3
4  static int triple(int a)
5  {
6      return a * 3;
7  }
8
9
10 static int quadruple(int a)
11 {
12     return a * 4;
13 }
14
15
16 static void affiche(int a, int (*pf)(int))
17 {
18     printf("%d.\n", (*pf)(a));
19 }
20
21
22 int main(void)
23 {
24     affiche(3, &triple);
25     affiche(3, &quadruple);
26     return 0;
27 }

```

```

1  9.
2  12.

```



La fonction `affiche()` ci-dessus est ce que l'on appelle une *fonction de rappel* (*callback function* en anglais), c'est-à-dire une fonction faisant appel à une autre à l'aide de l'adresse qui lui est fournie en argument.

33.2.3 Retour de fonction

Dans l'autre sens, il est possible de retourner un pointeur de fonction à l'aide d'une syntaxe... un peu lourde. :-°

```

1  #include <stddef.h>
2  #include <stdio.h>
3
4
5  static void affiche_pair(int a)
6  {
7      printf("%d est pair.\n", a);
8  }
9
10
11 static void affiche_impair(int a)
12 {
13     printf("%d est impair.\n", a);
14 }
15
16
17 static void (*affiche(int a))(int)
18 {
19     if (a % 2 == 0)
20         return &affiche_pair;
21     else
22         return &affiche_impair;
23 }
24
25

```

```

26
27 int main(void)
28 {
29     void (*pf)(int);
30     int a = 2;
31
32     pf = affiche(a);
33     (*pf)(a);
34     return 0;
35 }

```

```

1 2 est pair.

```

Comme pour une variable de type pointeur de fonction, le symbole `*` doit être entouré de parenthèses ainsi que l'identificateur qui le suit. Toutefois, lorsqu'il s'agit du type de retour d'une fonction, la liste des arguments doit également être placée entre ces parenthèses.

Dans cet exemple, la fonction `affiche()` attend un `int` et retourne un pointeur sur une fonction ne retournant rien et utilisant un argument de type `int`. Suivant si `a` est pair ou impair, la fonction `affiche()` retourne l'adresse de la fonction `affichage_pair()` ou `affichage_impair()` qui est recueillie par le pointeur `pf` de la fonction `main()`.

33.3 Pointeurs de fonction et pointeurs génériques

Vous le savez, le type `void` est employé en C pour produire un pointeur générique qui peut se voir assigner n'importe quel type de pointeur et être converti vers n'importe quel type de pointeurs. Cette définition est toutefois incomplète car il doit en fait être précisé que cela ne fonctionne que pour des pointeurs sur des *objets*. Le code ci-dessous est donc incorrect.

```

1 int (*pf)(int);
2 void *p;
3
4 pf = p; /* Faux. */
5 p = pf; /* Faux également. */

```

Pareillement, une conversion explicite d'un pointeur sur un objet vers ou depuis un pointeur sur fonction est interdite (ou, plus précisément, son résultat est indéterminé). Ceci exclut donc l'utilisation de l'indicateur `p` de la fonction `printf()` pour afficher un pointeur de fonction.

```

1 printf("%p.\n", (void *)pf); /* Faux. */

```

Toutefois, les pointeurs de fonction disposent de leur propre pointeur « générique ». Nous utilisons ici les guillemets car il ne l'est pas tout à fait puisqu'il peut notamment être utilisé à l'inverse d'un pointeur sur `void`. Un pointeur « générique » de fonction se déclare comme un pointeur de fonction, mais en ne spécifiant que le type de retour.

```

1 int (*pf)();

```

Un tel pointeur peut se voir assigner n'importe quel pointeur sur fonction du moment que le type de retour de celui-ci est identique au sien. Inversement, ce pointeur « générique » peut être affecté à un autre pointeur sous la même condition. Dans notre cas, le type de retour doit donc être `int`.

```

1 int (*f)(int, int);
2 int (*g)(char, char, double);
3 void (*h)(void);
4 int (*pf)();

```

```

5  pf = f; /* Ok. */
6  pf = g; /* Ok. */
7  pf = h; /* Faux car le type de retour de `h` est `void`. */
8  f = pf; /* Ok. */

```



Il existe cependant une exception supplémentaire : une fonction à nombre variables d'arguments ne peut pas être affectée à un tel pointeur, même si le type de retour est identique. Nous verrons bientôt de quoi il s'agit, mais pour l'heure, sachez que les fonctions de la famille de `printf()` et de `scanf()` sont concernées par cette règle.

33.3.1 La promotion des arguments



Hé là, minute papillon ! Il se passe quoi si j'utilise le pointeur `pf` avec une fonction qui attend normalement des arguments ?

Excellente question !

Vous vous en doutez : les arguments peuvent toujours être envoyé à la fonction référencée. Cependant, il y a une subtilité. Étant donné qu'un pointeur « générique » de fonction ne fournit aucune information quant aux arguments, le compilateur ne peut pas convertir ceux-ci vers le type attendu par la fonction. Ainsi, si vous fournissez un `int` et que la fonction attend un `char`, le `int` ne sera pas converti vers le type `char` par le compilateur.

Toutefois, dans un tel cas, plusieurs conversions implicites sont appliquées afin de limiter les types possibles (on parle de « promotion des arguments ») :

1. Un argument de type entier de rang inférieur ou égal à celui du type `int` (soit `char` et `short` le plus souvent) est converti vers le type `int` (ou `unsigned int` si le type `int` ne peut pas représenter toutes les valeurs du type d'origine).
2. Un argument de type `float` est converti vers le type `double`.



Ceci signifie qu'une fonction appelée à l'aide d'un pointeur « générique » de fonction ne pourra *jamais* recevoir des arguments de type `char`, `short` ou `float`.

Illustration.

```

1  #include <stdio.h>
2
3
4  static float triple(float f)
5  {
6      return 3.F * f;
7  }
8
9
10 static short quadruple(short n)
11 {
12     return 4 * n;
13 }
14
15
16 int main(void)
17 {
18     float (*pt)() = &triple;
19     short (*pq)() = &quadruple;
20
21     printf("triple = %f.\n", (*pt)(3.F)); /* Faux. */
22     printf("quadruple = %d.\n", (*pq)(2)); /* Faux. */
23     return 0;
24 }

```

33.3.2 Les pointeurs nuls

L'absence de conversions par le compilateur dans le cas où aucune information n'est fournie par rapport aux arguments pose un problème particulier dans le cas des pointeurs nuls et tout spécialement lors de l'usage de la macroconstante `NULL`.

Rappelez-vous : un pointeur nul est construit en convertissant, soit explicitement, soit implicitement, zéro (entier) vers un type pointeur. Or, étant donné que le compilateur n'effectuera aucune conversion implicite dans notre cas, nous ne pouvons compter que sur les conversions explicites.

Et c'est ici que le bât blesse : la macroconstante `NULL` a deux valeurs possibles : `(void *)0` ou `0`, le choix étant laissé aux différents systèmes. La première ne pose pas de problème, mais la seconde en pose un plutôt gênant : c'est un `int` qui sera passé comme argument et non un pointeur nul.

Dès lors, lorsque vous employez un pointeur « générique » de fonction, vous devez recourir à une conversion explicite si vous souhaitez produire un pointeur nul.

```
1 static void affiche(char *chaine)
2 {
3     if (chaine != NULL)
4         puts(chaine);
5 }
6
7 /* ... */
8
9 void (*pf)() = &affiche;
10
11 (*pf)(NULL); /* Faux. */
12 (*pf)(0); /* Faux. */
13 (*pf)((char*)0); /* Ok. */
```

En résumé

1. Un pointeur de fonction permet de stocker une référence vers une fonction ;
2. Il n'est pas nécessaire d'employer l'opérateur `&` pour obtenir l'adresse d'une fonction ;
3. Le déréférencement n'est pas obligatoire lors de l'utilisation d'un pointeur de fonction ;
4. Une fonction employant un pointeur vers une autre fonction reçu en argument est appelée une fonction de rappel (*callback function* en anglais) ;
5. Le recours à une structure permet d'éviter les problèmes de définitions récursives ;
6. Il est possible d'utiliser un pointeur « générique » de fonction en ne fournissant aucune information quant aux arguments lors de sa définition ;
7. Un pointeur « générique » de fonction peut être converti vers ou depuis n'importe quel autre type de pointeur de fonction du moment que le type de retour reste identique ;
8. Lors de l'utilisation d'un pointeur « générique » de fonction, les arguments transmis sont promus, mais aucune conversion implicite n'est réalisée par le compilateur ;
9. Lors de l'utilisation d'un pointeur « générique » de fonction, un pointeur nul ne peut être fourni qu'à l'aide d'une conversion explicite.

Les fonctions à nombre variable d'arguments

Dans le chapitre précédent, il a été question des pointeurs de fonction et, notamment, des pointeurs « génériques » de fonction. Néanmoins, ces derniers ne permettent pas d'expliquer le fonctionnement des fonctions `printf()` et `scanf()`. C'est ce que nous allons voir dans ce chapitre en étudiant les **fonctions à nombre variable d'arguments**.

34.1 Présentation

Une fonction à nombre variable d'arguments est, comme son nom l'indique, une fonction capable de recevoir et de manipuler un nombre variable d'arguments, mais en plus, et cela son nom ne le spécifie pas, potentiellement de types différents.

Une telle fonction se définit en employant comme *dernier* paramètre une suite de trois points appelée une ellipse indiquant que des arguments supplémentaires peuvent être transmis.

```
1 void affiche_suite(int n, ...);
```

Le prototype ci-dessus déclare une fonction recevant un `int` et, éventuellement, un nombre indéterminé d'autres arguments. La partie précédant l'ellipse décrit donc les paramètres attendus par la fonction et qui, comme pour n'importe quelle autre fonction, *doivent* lui être transmis.



L'ellipse ne peut être placée qu'à la fin de la liste des paramètres.

Une fois définie, la fonction peut être appelée en lui fournissant zéro ou plusieurs arguments supplémentaires.

```
1 affiche_suite(10, 20);  
2 affiche_suite(10, 20, 30, 40, 50, 60, 70, 80, 90, 100);  
3 affiche_suite(10);
```



Comme pour les pointeurs génériques de fonction, le nombre et le types des arguments est inconnu du compilateur (c'est d'ailleurs bien le but de la manœuvre). Dès lors, les mêmes règles de promotion s'appliquent ainsi que les problèmes qui y sont inhérent, notamment la problématique des pointeurs nuls. N'employer donc pas la macroconstante `NULL` pour fournir un pointeur nul comme argument optionnel.

34.2 L'en-tête <stdarg.h>

Une fois le prototype de la fonction déterminé, encore faut-il pouvoir manipuler ces arguments supplémentaires au sein de sa définition. Pour ce faire, la bibliothèque standard nous fournit trois macrofonctions : `va_start()`, `va_arg()` et `va_end()` définies dans l'en-tête `<stdarg.h>`. Ces trois macrofonctions attendent comme premier argument une variable de type `va_list` défini dans le même en-tête.

Afin d'illustrer leur fonctionnement, nous vous proposons directement un exemple mettant en œuvre une fonction `affiche_suite()` qui reçoit comme premier paramètre le nombre d'entiers qui vont lui être transmis et les affiche ensuite tous, un par ligne.

```
1  #include <stdarg.h>
2  #include <stdio.h>
3
4
5  void affiche_suite(int nb, ...)
6  {
7      va_list ap;
8
9      va_start(ap, nb);
10
11     while (nb > 0)
12     {
13         int n;
14
15         n = va_arg(ap, int);
16         printf("%d.\n", n);
17         --nb;
18     }
19
20     va_end(ap);
21 }
22
23
24 int main(void)
25 {
26     affiche_suite(10, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100);
27     return 0;
28 }
```

```
1  10.
2  20.
3  30.
4  40.
5  50.
6  60.
7  70.
8  80.
9  90.
10 100.
```

La macrofonction `va_start` initialise le parcours des paramètres optionnels. Elle attend deux arguments : une variable de type `va_list` et le nom du dernier paramètre obligatoire de la fonction courante (dans notre cas `nb`). Il est impératif de l'appeler avant toute opération sur les paramètres optionnels.

La macrofonction `va_arg()` retourne le paramètre optionnel suivant en considérant celui-ci comme de type `type`. Elle attend deux arguments : une variable de type `va_list` précédemment initialisée par la macrofonction `va_start()` et le type du paramètre optionnel suivant.

Enfin, la fonction `va_end()` met fin au parcours des arguments optionnels. Elle doit *toujours* être appelée après un appel à la macrofonction `va_start()`.



La macrofonction `va_arg()` n'effectue *aucune* vérification ! Cela signifie que si vous renseignez un type qui ne correspond pas au paramètre optionnel suivant ou si vous tentez de récupérer un paramètre optionnel qui n'existe pas, vous allez au devant de comportements indéfinis.

Étant donné cet état de fait, il est impératif de pouvoir déterminer le nombre d'arguments optionnels envoyé à la fonction. Dans notre cas, nous avons opté pour l'emploi d'un paramètre obligatoire indiquant le nombre d'arguments optionnels envoyés.



Mais ?! Et si l'utilisateur de la fonction se plante en ne précisant pas le bon nombre ou le bon type d'arguments ?

Eh bien... C'est foutu.

Il s'agit là d'une lacune similaire à celle des pointeurs « génériques » de fonction : étant donné que le type et le nombre des arguments optionnels est inconnu, le compilateur ne peut effectuer aucune vérification ou conversion.

34.3 Méthodes pour déterminer le nombre et le type des arguments

De manière générale, il existe trois grandes méthodes pour gérer le nombre et le type des arguments optionnels.

34.3.1 Les chaînes de formats

Cette méthode, vous la connaissez déjà, il s'agit de celle employée par les fonctions `printf()`, `scanf()` et consœurs. Elle consiste à décrire le nombre et le type des arguments à l'aide d'une chaîne de caractères comprenant des indicateurs.

34.3.2 Les suites d'arguments de même type

La seconde solution ne peut être utilisée que si tous les arguments optionnels sont de même type. Elle consiste soit à indiquer le nombre d'arguments transmis, soit à utiliser un délimiteur. La fonction `affiche_suite()` recourt par exemple à un paramètre pour déterminer le nombre de paramètres optionnels qu'elle a reçu.

Un délimiteur pourrait par exemple être un pointeur nul dans le cas d'une fonction similaire affichant une suite de chaîne de caractères.

```

1  #include <stdarg.h>
2  #include <stddef.h>
3  #include <stdio.h>
4
5
6  static void affiche_suite(char *chaine, ...)
7  {
8      va_list ap;
9
10     va_start(ap, chaine);
11
12     do
13     {
14         puts(chaine);
15         chaine = va_arg(ap, char *);
16     } while(chaine != NULL);
17
18     va_end(ap);
19 }
```



```

20
21
22 int main(void)
23 {
24     affiche_suite("un", "deux", "trois", (char *)0);
25     return 0;
26 }

```

```

1 un
2 deux
3 trois

```

34.3.3 Emploi d'un pivot

La dernière pratique consiste à recourir à un paramètre « pivot » qui fera varier le parcours des paramètres optionnels en fonction de sa valeur. Notez qu'un type énuméré se prête très bien pour ce genre de paramètre. Par exemple, la fonction suivante affiche un `int` ou un `double` suivant la valeur du paramètre `type`.

```

1  #include <stdarg.h>
2  #include <stdio.h>
3
4  enum type { TYPE_INT, TYPE_DOUBLE };
5
6
7  static void affiche(enum type type, ...)
8  {
9      va_list ap;
10
11      va_start(ap, type);
12
13      switch (type)
14      {
15          case TYPE_INT:
16              printf("Un entier : %d.\n", va_arg(ap, int));
17              break;
18
19          case TYPE_DOUBLE:
20              printf("Un flottant : %f.\n", va_arg(ap, double));
21              break;
22      }
23
24      va_end(ap);
25  }
26
27
28  int main(void)
29  {
30      affiche(TYPE_INT, 10);
31      affiche(TYPE_DOUBLE, 3.14);
32      return 0;
33  }

```

```

1  Un entier : 10.
2  Un flottant : 3.140000.

```

En résumé

1. Une fonction à nombre variable d'arguments est capable de recevoir et de manipuler une suite indéterminée d'arguments optionnels de types potentiellement différents.
2. L'ellipse ne peut être placée qu'à la fin de la liste des paramètres et doit impérativement être précédée d'un paramètre non optionnel.

3. La macrofonction `va_arg()` n'effectue *aucune* vérification, il est donc impératif de contrôler le nombre et le types des arguments reçus, par exemple à l'aide d'une chaîne de formats ou d'un paramètre décrivant le nombre d'arguments transmis.

T.P. : un allocateur statique de mémoire

Dans ce dernier chapitre, nous allons mettre en œuvre une partie des notions présentées dans cette partie afin de réaliser un allocateur statique de mémoire.

35.1 Objectif

Votre objectif sera de parvenir à réaliser un petit allocateur statique de mémoire en mettant en œuvre deux fonctions : `static_malloc()` et `static_free()` comparables aux fonctions `malloc()` et `free()` de la bibliothèque standard. Toutefois, afin d'éviter d'entrer dans les méandres de l'allocation dynamique de mémoire, ces fonctions fourniront de la mémoire préalablement allouée statiquement.

35.2 Première étape : allouer de la mémoire

Pour commencer, vous allez devoir construire une fonction `static_malloc()` dont le prototype sera le suivant.

```
1 void *static_malloc(size_t taille);
```

À la lumière de la fonction `malloc()`, celle-ci reçoit une taille en multiplète et retourne un pointeur vers un bloc de mémoire d'au moins la taille demandée ou un pointeur nul s'il n'y a plus de mémoire disponible.

35.2.1 Mémoire statique

Afin de réaliser ses allocations, la fonction `static_malloc()` ira piocher dans un bloc de mémoire statique. Celui-ci consistera simplement en un tableau de `char` de classe de stockage statique d'une taille prédéterminée. Pour cet exercice, nous partirons avec un bloc de un mébimultiplète, soit 1.048.576 multiplètes (1024×1024).

```
1 static char mem[1048576UL];
```

Alignement

Toutefois, retourner un bloc de `char` de la taille demandée ne suffit pas. En effet, si nous allouons par exemple treize multiplètes, disons pour une chaîne de caractères, puis quatre multiplètes pour un `int`, nous allons retourner une adresse qui ne respecte potentiellement pas l'alignement requis par le type `int`. Étant donné que la fonction `static_malloc()` ne connaît pas les contraintes

d'alignements que doit respecter l'objet qui sera stocké dans le bloc qu'elle fournit, elle doit retourner un bloc respectant les contraintes les plus strictes. Dit autrement, la taille de chaque bloc devra être un multiple de l'alignement le plus rigoureux.

Cet alignement peut être connu à l'aide d'une union comprenant les types les plus contraignants et de la macrofonction `offsetof()`, comme précisé dans le chapitre sur les unions.

```

1 union align
2 {
3     long e;
4     long double f;
5     void *p;
6 };

```

Mais... ce n'est pas tout ! Le tableau alloué statiquement doit lui aussi être aligné suivant l'alignement le plus sévère. En effet, si celui-ci commence à une adresse non multiple de cet alignement, notre stratégie précédente tombe à l'eau. Pour ce faire, il nous suffit d'inclure le tableau dans une union avec comme autre membre l'union décrite ci-dessus.

```

1 union reserve
2 {
3     union align align
4     char mem[1048576UL]
5 };
6
7 static union reserve reserve;

```

Avec ceci, vous devriez pouvoir réaliser la fonction `static_malloc()` sans encombre. Hop hop ! Au travail !

35.3 Correction

```

1 #include <assert.h>
2 #include <stddef.h>
3 #include <stdio.h>
4
5 #define MEM_SIZE (1024UL * 1024UL)
6 #define ALIGNEMENT(type) (offsetof(struct { char c; type t; }, t))
7
8 union align
9 {
10     long e;
11     long double f;
12     void *p;
13 };
14
15 union reserve
16 {
17     union align align;
18     char mem[MEM_SIZE];
19 };
20
21 static union reserve reserve;
22
23 static size_t calcule_multiple_align(size_t);
24 static void *static_malloc(size_t);
25
26
27 static size_t calcule_multiple_align(size_t a)
28 {
29     /*
30      * Calcule le plus petit multiple de l'alignement maximal égal ou supérieur à `a`.
31      */
32
33     static size_t align_max = ALIGNEMENT(union align);
34     size_t multiple = a / align_max;

```

```

35     return ((a % align_max == 0) ? multiple : multiple + 1) * align_max;
36 }
37
38
39
40 static void *static_malloc(size_t taille)
41 {
42     /*
43      * Alloue un bloc de mémoire au moins de taille `taille`.
44      */
45
46     static size_t alloue = 0UL;
47     void *ret;
48
49     assert(taille > 0);
50     taille = calcule_multiple_align(taille);
51
52     if (MEM_SIZE - alloue < taille)
53     {
54         fprintf(stderr, "Il n'y a plus assez de mémoire disponible.\n");
55         return NULL;
56     }
57
58     ret = &reserve.mem[alloue];
59     alloue += taille;
60     return ret;
61 }

```

La fonction `calcule_multiple_align()` détermine le plus petit multiple de l'alignement maximal égal ou supérieur à `a`. Celle-ci nous permet d'« arrondir » la taille demandée de sorte que les blocs alloués soit toujours correctement alignés.

La macrofonction `ALIGNEMENT()` calcule l'alignement d'un type donné en recourant à la macrofonction `offsetof()` comme expliqué précédemment.

La fonction `static_malloc()` utilise une variable statique : `alloue` qui représente la quantité de mémoire déjà allouée. Dans le cas où la taille demandée (arrondie au multiple de l'alignement qui lui est égal ou supérieur) n'est pas disponible, un pointeur nul est retourné. Sinon, la variable `alloue` est mise à jour et un pointeur vers la position courante du tableau `reserve.mem` est retourné.

Notez que nous avons utilisé une assertion afin d'éviter qu'une taille nulle ne soit fournie.

35.4 Deuxième étape : libérer de la mémoire

Pouvoir allouer de la mémoire, c'est bien, mais pouvoir en libérer, ce serait mieux. Parce que pour le moment, dès que l'on arrive à la fin du tableau, que la mémoire précédemment allouée soit encore utilisée ou non, c'est grillé. :-°

Toutefois, pour mettre en place un système de libération de la mémoire, il va nous falloir changer un peu de tactique.

35.4.1 Ajout d'un en-tête

Tout d'abord, si nous voulons libérer des blocs puis les réutiliser, il nous faut conserver d'une manière ou d'une autre leur taille. En effet, sans cette information, il nous sera impossible de les réaffecter.

Pour ce faire, il nous est possible d'ajouter un en-tête lors de l'allocation d'un bloc, autrement dit un objet (par exemple un entier de type `size_t`) qui précédera le bloc alloué et contiendra la taille du bloc.

```

1  +-----+-----+
2  | En-tête | Bloc alloué |
3  +-----+-----+

```

Ainsi, lors de l'allocation, il nous suffit de transmettre à l'utilisateur le bloc alloué sans son en-tête à l'aide d'une expression du type `(char *)ptr + sizeof (size_t)` et, à l'inverse, lorsque l'utilisateur souhaite libérer un bloc, de le récupérer à l'aide d'une expression comme `(char *)ptr - sizeof (size_t)`. Notez que cette technique nous coûte un peu plus de mémoire puisqu'il est nécessaire d'allouer le bloc *et* l'en-tête.

35.4.2 Les listes chaînées

Mais, ce n'est pas tout. Pour pouvoir retrouver un bloc libéré, il nous faut également un moyen pour les conserver et en parcourir la liste. Afin d'atteindre cet objectif, nous allons employer une structure de donnée appelée une **liste chaînée**. Celle-ci consiste simplement en une structure comprenant des données ainsi qu'un pointeur vers une structure du même type.

```
1 struct list
2 {
3     struct list *suivante;
4 };
```

Ainsi, il est possible de créer des *chaînes* de structure, chaque maillon de la chaîne référençant le suivant. L'idée pour notre allocateur va être de conserver une liste des blocs libérés, liste qu'il parcourera en vue de rechercher un bloc de taille suffisante *avant* d'allouer de la mémoire provenant de la réserve. De cette manière, il nous sera possible de réutiliser de la mémoire précédemment allouée avant d'aller puiser dans la réserve.

Nous aurons donc a priori besoin d'une liste comprenant une référence vers le bloc libéré et une référence vers le bloc suivant (il ne nous est pas nécessaire d'employer un membre pour la taille du bloc puisque l'en-tête la contient).

```
1 struct bloc
2 {
3     void *p;
4     struct bloc *suivant;
5 };
```

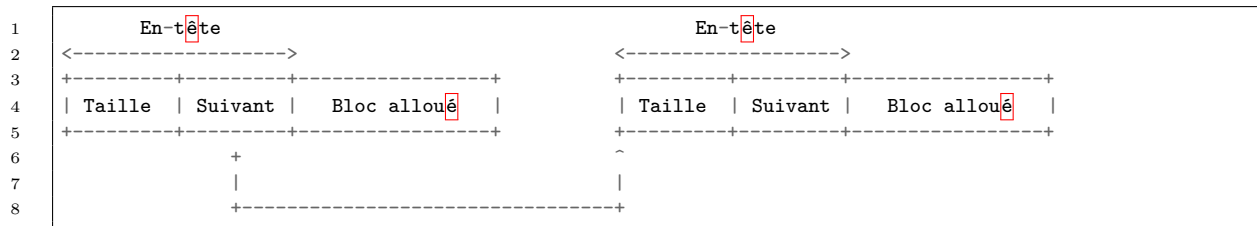
Le serpent qui se mange la queue

Cependant, avec une telle technique, nous risquons d'entrer dans un cercle vicieux. En effet, imaginons qu'un utilisateur libère un bloc de 64 multiplats. Pour l'ajouter à la liste des blocs libérés, nous avons besoin d'espace pour stocker une structure `bloc`, nous allons donc allouer un peu de mémoire. De plus, si par la suite ce bloc est réutilisé, la structure `bloc` ne nous est plus utile, sauf que si nous la libérons, nous allons devoir allouer une autre structure `bloc` pour référencé... une structure `bloc` (à moins que la structure ne s'autoréférence). Voilà qui n'est pas très optimal.

À la place, il nous est possible de recourir à une autre stratégie : inclure la structure `bloc` dans l'en-tête ou, plus précisément, son champ `suivant`, la référence vers le bloc n'étant plus nécessaire dans ce cas. Autrement dit, notre en-tête sera le suivant.

```
1 struct bloc
2 {
3     size_t taille;
4     struct bloc *suivant;
5 };
```

Lors de la libération du bloc, il nous suffit d'employer le champ `suivant` de l'en-tête pour ajouter le bloc à la liste des blocs libres.



Étant donné que l'en-tête précède le bloc alloué, il est impératif que sa taille soit « arrondie » de sorte que les données qui seront stockées dans le bloc respectent l'alignement le plus strict.

Avec ceci, vous devriez pouvoir réaliser une fonction `static_free()` et modifier la fonction `static_malloc()` en conséquence.

35.5 Correction

```

1  #include <assert.h>
2  #include <stddef.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  #define MEM_SIZE (1024UL * 1024UL)
7  #define ALIGNEMENT(type) (offsetof(struct { char c; type t; }, t))
8  #define TAILLE_EN_TETE (offsetof(struct { struct bloc b; union align u; }, u))
9
10 union align
11 {
12     long e;
13     long double f;
14     void *p;
15 };
16
17 union reserve
18 {
19     union align align;
20     char mem[MEM_SIZE];
21 };
22
23 struct bloc
24 {
25     size_t taille;
26     struct bloc *suivant;
27 };
28
29
30 static union reserve reserve;
31 static struct bloc *libre;
32
33 static void bloc_init(struct bloc *, size_t);
34 static size_t calcule_multiple_align(size_t);
35 static struct bloc *recherche_bloc_libre(size_t);
36 static void static_free(void *);
37 static void *static_malloc(size_t);
38
39
40 static void bloc_init(struct bloc *b, size_t taille)
41 {
42     /*
43      * Initialise les membres d'une structure `bloc`.
44      */
45
46     b->taille = taille;
47     b->suivant = NULL;
48 }

```



```

49
50
51 static size_t calcule_multiple_align(size_t a)
52 {
53     /*
54      * Calcule le plus petit multiple de l'alignement maximal égal ou supérieur à `a`.
55      */
56
57     static size_t align_max = ALIGNEMENT(union align);
58     size_t multiple = a / align_max;
59
60     return ((a % align_max == 0) ? multiple : multiple + 1) * align_max;
61 }
62
63
64 static struct bloc *recherche_bloc_libre(size_t taille)
65 {
66     /*
67      * Recherche un bloc libre ou une suite de bloc libres dont la taille
68      * est au moins égale à `taille`.
69      */
70
71     struct bloc *bloc = libre;
72     struct bloc *precedent = NULL;
73     struct bloc *ret = NULL;
74
75     while (bloc != NULL)
76     {
77         if (bloc->taille >= taille)
78         {
79             if (precedent != NULL)
80                 precedent->suivant = bloc->suivant;
81             else
82                 libre = bloc->suivant;
83
84             ret = bloc;
85             break;
86         }
87
88         precedent = bloc;
89         bloc = bloc->suivant;
90     }
91
92     return ret;
93 }
94
95
96 static void *static_malloc(size_t taille)
97 {
98     /*
99      * Alloue un bloc de mémoire au moins de taille `taille`.
100     */
101
102     static size_t alloue = 0UL;
103     void *ret;
104
105     assert(taille > 0);
106     taille = calcule_multiple_align(taille);
107     ret = recherche_bloc_libre(taille);
108
109     if (ret == NULL)
110     {
111         if (MEM_SIZE - alloue < TAILLE_EN_TETE + taille)
112         {
113             fprintf(stderr, "Il n'y a plus assez de mémoire disponible.\n");
114             return NULL;
115         }
116
117         ret = &reserve.mem[alloue];
118         alloue += TAILLE_EN_TETE + taille;
119         bloc_init(ret, taille);
120     }
121

```

```

122     return (char *)ret + TAILLE_EN_TETE;
123 }
124
125
126 static void static_free(void *ptr)
127 {
128     /*
129      * Ajoute le bloc fourni à la liste des blocs libres.
130      */
131
132     struct bloc *bloc = libre;
133     struct bloc *nouveau;
134
135     if (ptr == NULL)
136         return;
137
138     nouveau = (struct bloc *)((char *)ptr - TAILLE_EN_TETE);
139
140     while (bloc != NULL && bloc->suivant != NULL)
141         bloc = bloc->suivant;
142
143     if (bloc == NULL)
144         libre = nouveau;
145     else
146         bloc->suivant = nouveau;
147 }

```

Désormais, la fonction `static_malloc()` fait d'abord appel à la fonction `recherche_bloc_libre()` avant d'allouer de la mémoire de la réserve. Comme son nom l'indique, cette fonction parcourt la liste des blocs libres à la recherche d'un bloc d'une taille égale ou supérieure à celle demandée. Cette liste est matérialisée par la variable globale `libre` qui est un pointeur sur une structure de type `bloc`. Cette variable correspond au premier maillon de la liste et est employée pour initialiser le parcours.

Si aucun bloc libre n'est trouvé, alors la fonction `static_malloc()` pioche dans la réserve un bloc de la taille demandée augmentée de la taille de l'en-tête. Ces deux tailles sont « arrondies » au multiple égal ou directement supérieur de l'alignement le plus contraignant. Étant donné que la taille de l'en-tête est fixe, nous avons représenté sa taille « arrondie » à l'aide de la macroconstante `TAILLE_EN_TETE` et de la macrofonction `offsetof()`. Une fois le tout alloué, l'en-tête est « retiré » en ajoutant la taille de l'en-tête au pointeur référant la mémoire allouée et le bloc demandé est retourné.

La fonction `static_free()` se rend à la fin de la liste et y ajoute le bloc qui lui est fourni en argument. Pour ce faire, elle « récupère » l'en-tête en soustrayant la taille de l'en-tête au pointeur fourni en argument. Notez également que cette fonction n'effectue aucune opération dans le cas où un pointeur nul lui est fourni (tout comme la fonction `free()` standard), ce qui peut être intéressant pour simplifier la gestion d'erreur dans certains cas.

35.6 Troisième étape : fragmentation et défragmentation

Bien, notre allocateur recycle à présent les blocs qu'il a précédemment alloués, c'est une bonne chose. Toutefois, un problème subsiste : la fragmentation de la mémoire allouée, autrement dit sa division en une multitude de petits blocs.

S'il s'agit d'un effet partiellement voulu (nous allouons par petits blocs pour préserver la réserve), il peut avoir des conséquences fâcheuses non désirées. En effet, imaginez que nous ayons alloué toute la mémoire sous forme de blocs de 16, 32 et 64 multipliants, si même tous ces blocs sont libres, notre allocateur retournera un pointeur nul dans le cas d'une demande de par exemple 80 multipliants... Voilà qui est plutôt gênant.

Une solution consiste à défragmenter la liste des blocs libres, c'est-à-dire fusionner plusieurs blocs pour en reconstruire d'autre avec une taille plus importante. Dans notre cas, nous allons mettre en œuvre ce système lors de la recherche d'un bloc libre : désormais, nous allons regarder

si un bloc est d'une taille suffisante *ou* si *plusieurs* blocs, une fois fusionnés, seront de taille suffisante.

35.6.1 Fusion de blocs

Toutefois, une fusion de blocs n'est possible que si ceux-ci sont adjacents, c'est-à-dire s'ils se suivent en mémoire. Plus précisément, l'adresse suivant le premier bloc à fusionner doit être celle de début du second (autrement dit `(char *)ptr_bloc1 + taille_bloc1 == (char *)ptr_bloc2`).

Néanmoins, il ne nous est pas possible de vérifier cela facilement si notre liste de blocs libres n'est pas un minimum triée. En effet, sans tri, il nous serait nécessaire de parcourir toute la liste à la recherche d'éventuels blocs adjacents au premier, puis, de faire de même pour le deuxième et ainsi de suite, ce qui n'est pas particulièrement efficace.

À la place, il nous est possible de trier notre liste par adresses croissantes (ou décroissantes, le résultat sera le même) de sorte que si un bloc n'est pas adjacent au suivant, la recherche peut être immédiatement arrêtée pour ce bloc ainsi que tous ceux qui lui étaient adjacents. Ce tri peut être réalisé simplement lors de l'insertion d'un nouveau bloc libre en plaçant celui-ci correctement dans la liste à l'aide de comparaisons : s'il a une adresse inférieure à celle d'un élément de la liste, il est placé avant cet élément, sinon, le parcours continue.

En effet, deux pointeurs peuvent tout à fait être comparés du moment que ceux-ci référencent un même objet ou un même agrégat (c'est notre cas ici puisqu'ils référenceront tous des éléments du tableau `mem` de l'union `reserve`) et qu'ils sont du même type (une conversion explicite vers le type pointeur sur `char` sera donc nécessaire comme explicité auparavant).



N'oubliez pas que si deux ou plusieurs blocs sont fusionnés, il n'y a plus besoin que d'un seul en-tête, les autres peuvent donc être comptés comme de la mémoire utilisable.

35.7 Correction

```

1  #include <assert.h>
2  #include <stddef.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  #define MEM_SIZE (1024UL * 1024UL)
7  #define ALIGNEMENT(type) (offsetof(struct { char c; type t; }, t))
8  #define TAILLE_EN_TETE (offsetof(struct { struct bloc b; union align u; }, u))
9
10 union align
11 {
12     long e;
13     long double f;
14     void *p;
15 };
16
17 union reserve
18 {
19     union align align;
20     char mem[MEM_SIZE];
21 };
22
23 struct bloc
24 {
25     size_t taille;
26     struct bloc *suivant;
27 };
28
29 struct suite
30 {
31     struct bloc *precedent;
32     struct bloc *premier;

```

```

33     struct bloc *dernier;
34     size_t taille;
35 };
36
37
38 static union reserve reserve;
39 static struct bloc *libre;
40
41 static void bloc_init(struct bloc *, size_t);
42 static size_t calcule_multiple_align(size_t);
43 static struct bloc *recherche_bloc_libre(size_t);
44 static void static_free(void *);
45 static void *static_malloc(size_t);
46
47
48 static void bloc_init(struct bloc *b, size_t taille)
49 {
50     /*
51      * Initialise les membres d'une structure `bloc`.
52      */
53
54     b->taille = taille;
55     b->suivant = NULL;
56 }
57
58
59 static size_t calcule_multiple_align(size_t a)
60 {
61     /*
62      * Calcule le plus petit multiple de l'alignement maximal égal ou supérieur à `a`.
63      */
64
65     static size_t align_max = ALIGNEMENT(union align);
66     size_t multiple = a / align_max;
67
68     return ((a % align_max == 0) ? multiple : multiple + 1) * align_max;
69 }
70
71
72 static struct bloc *recherche_bloc_libre(size_t taille)
73 {
74     /*
75      * Recherche un bloc libre ou une suite de bloc libres dont la taille
76      * est au moins égale à `taille`.
77      */
78
79     struct suite suite = { 0 };
80     struct bloc *bloc = libre;
81     struct bloc *precedent = NULL;
82     struct bloc *ret = NULL;
83
84     while (bloc != NULL)
85     {
86         if (bloc->taille >= taille)
87         {
88             if (precedent != NULL)
89                 precedent->suivant = bloc->suivant;
90             else
91                 libre = bloc->suivant;
92
93             ret = bloc;
94             break;
95         }
96         else if (suite.dernier != NULL && (char *)suite.dernier + TAILLE_EN_TETE \
97             + suite.dernier->taille == (char *)bloc)
98         {
99             suite.dernier = bloc;
100             suite.taille += TAILLE_EN_TETE + bloc->taille;
101         }
102         else
103         {
104             suite.precedent = precedent;
105             suite.premier = bloc;

```

```

106         suite.dernier = bloc;
107         suite.taille = bloc->taille;
108     }
109
110     if (suite.taille >= taille)
111     {
112         if (suite.precedent != NULL)
113             suite.precedent->suivant = suite.dernier->suivant;
114         else
115             libre = suite.dernier->suivant;
116
117         suite.premier->taille = suite.taille;
118         ret = suite.premier;
119         break;
120     }
121
122     precedent = bloc;
123     bloc = bloc->suivant;
124 }
125
126 return ret;
127 }
128
129
130 static void *static_malloc(size_t taille)
131 {
132     /*
133      * Alloue un bloc de mémoire au moins de taille `taille`.
134      */
135
136     static size_t alloue = 0UL;
137     void *ret;
138
139     assert(taille > 0);
140     taille = calcule_multiple_align(taille);
141     ret = recherche_bloc_libre(taille);
142
143     if (ret == NULL)
144     {
145         if (MEM_SIZE - alloue < TAILLE_EN_TETE + taille)
146         {
147             fprintf(stderr, "Il n'y a plus assez de mémoire disponible.\n");
148             return NULL;
149         }
150
151         ret = &reserve.mem[alloue];
152         alloue += TAILLE_EN_TETE + taille;
153         bloc_init(ret, taille);
154     }
155
156     return (char *)ret + TAILLE_EN_TETE;
157 }
158
159
160 static void static_free(void *ptr)
161 {
162     /*
163      * Ajoute le bloc fourni à la liste des blocs libres.
164      */
165
166     struct bloc *bloc = libre;
167     struct bloc *precedent = NULL;
168     struct bloc *nouveau;
169
170     if (ptr == NULL)
171         return;
172
173     nouveau = (struct bloc *)((char *)ptr - TAILLE_EN_TETE);
174
175     while (bloc != NULL)
176     {
177         if (bloc < nouveau)
178         {

```

```

179     if (precedent != NULL)
180         precedent->suivant = nouveau;
181     else
182         libre = nouveau;
183
184
185     nouveau->suivant = bloc;
186     break;
187 }
188
189 precedent = bloc;
190 bloc = bloc->suivant;
191 }
192
193 if (bloc == NULL)
194     libre = nouveau;
195 }

```

La fonction `static_free()` a été aménagée afin que les adresses des différents blocs soient triées par ordre croissant. Si le nouveau bloc libéré a une adresse inférieure à un autre bloc, il est placé avant lui, sinon il est placé à la fin de la liste.

Quant à la fonction `recherche_bloc_libre()`, elle emploie désormais une structure `suite` qui comprend un pointeur vers le bloc précédent le début de la suite (champ `precedent`), un pointeur vers le premier bloc de la suite (champ `premier`), un pointeur vers le dernier bloc de la suite (champ `dernier`) et la taille totale de la suite (champ `taille`).

Dans le cas où cette structure n'a pas encore été modifiée ou que le dernier bloc de la suite n'est pas adjacent à celui qui le suit, le premier et le dernier blocs de la suite deviennent le bloc courant, la taille est réinitialisée à la taille du bloc courant et le bloc précédent la suite est... celui qui précède le bloc courant (ou un pointeur nul s'il n'y en a pas).

Si la suite atteint une taille suffisante, alors les blocs la composant sont fusionnés et le nouveau bloc ainsi construit est retourné.

Ce chapitre clôt cette troisième partie ainsi que ce cours. N'hésitez pas à revoir l'un ou l'autre passage qui vous ont semblé difficiles ou flous avant de prendre le large vers de nouveaux horizons.

Table des figures

2.1	L'éditeur d'images vectorielles Inkscape est un programme graphique	18
3.1	Exemple : on demande à notre mémoire de sélectionner la case mémoire d'adresse 1002 et on récupère son contenu (ici, 17).	24
7.1	Structure If	55
7.2	Structure if else	57
9.1	Structure While	75
9.2	Instruction do... while...	77
15.1	Exemple, avec une variable a qui est un pointeur sur une variable b	122
16.1	Représentation en mémoire de la structure	136
16.2	Vue de la mémoire par le processeur	138
16.3	La structure correctement alignée	139
17.1	Représentation d'un tableau en mémoire	141
17.2	Exemple de tableau en deux dimensions	150
17.3	Column-major order	150
17.4	Row-major order	150
17.5	Explication des triangles de Pascal en image	154

Liste des tableaux

3.1	Correspondance base deux-base dix	22
3.2	Adresse mémoire	24
3.3	Les 7 types de base	25
3.4	Les limites des types	26
3.5	Les identificateurs	27
3.6	Les nombres de 1 à 16 en binaire, octal, décimal et hexadécimal	30
4.1	Liste des indicateurs de conversion	35
4.2	Les nombres de 1 à 16 en binaire, octal, décimal et hexadécimal.	36
21.1	Les modes d'accès à un fichier	188

Table des matières

Avant-propos	1
Remerciements	1
Sommaire	4
I Les bases du langage C	5
1 Introduction à la programmation	7
1.1 Avant-propos	7
1.1.1 Esprit et but du tutoriel	7
1.1.2 À qui est destiné ce cours ?	7
1.2 Aller plus loin	8
1.3 La programmation, qu'est-ce que c'est ?	8
1.3.1 Les programmes expliqués en long, en large et en travers	8
1.4 Le langage C	10
1.4.1 L'histoire du C	10
1.4.2 Pourquoi apprendre le C ?	10
1.5 La norme	11
1.6 L'algorithmique	12
1.6.1 Le pseudo-code	12
2 Rencontre avec le C	13
2.1 Windows	13
2.1.1 Le compilateur	13
2.2 L'éditeur de texte	14
2.3 Introduction à la ligne de commande	14
2.4 GNU/Linux, *BSD et autres Unixöides	15
2.4.1 Le compilateur	15
2.4.2 Configuration	15
2.4.3 L'éditeur de texte	16
2.4.4 Introduction à la ligne de commande	16
2.5 Mac OS X	17
2.5.1 Le compilateur	17
2.5.2 Configuration	18

2.5.3	L'éditeur de texte	18
2.5.4	Introduction à la ligne de commande	18
2.6	Notre cible	18
2.7	Première rencontre	19
2.8	Les commentaires	20
3	Les variables	21
3.1	Qu'est-ce qu'une variable ?	21
3.1.1	Codage des informations	21
3.1.2	Binaire	22
3.2	La mémoire	22
3.2.1	Adresse mémoire	23
3.2.2	Les variables	24
3.3	Déclarer une variable	25
3.3.1	Les types	25
3.3.2	Les identificateurs	26
3.3.3	Déclaration et initialisation	27
3.3.4	Affectation	29
3.4	Les représentations octales et hexadécimales	30
3.5	Constantes octales et hexadécimales	31
4	Manipulations basiques des entrées/sorties	33
4.1	Les sorties	33
4.1.1	Première approche	33
4.1.2	Les formats	34
4.1.3	Précision des nombres flottants	36
4.1.4	Les caractères spéciaux	36
4.1.5	Sur plusieurs lignes	37
4.2	Interagir avec l'utilisateur	37
5	Les opérations mathématiques	41
5.1	Les opérations mathématiques de base	41
5.2	Division et modulo	41
5.3	Utilisation	42
5.4	La priorité des opérateurs	43
5.5	Les expressions	43
5.6	Type d'une expression	44
5.6.1	Suffixes	44
5.6.2	Exemple	44
5.7	Conversions de types	45
5.7.1	Perte d'informations	45
5.7.2	Deux types de conversions	45
5.8	Sucre syntaxique	46
5.9	Les opérateurs combinés	46
5.9.1	L'incrément et la décrémentation	47
5.10	Exercices	48
6	Tests et conditions	49
6.1	Les booléens	49
6.2	Les opérateurs de comparaison	50
6.2.1	Comparaisons	50
6.2.2	Résultat d'une comparaison	50

6.3	Les opérateurs logiques	50
6.3.1	Les opérateurs logiques de base	51
6.3.2	Évaluation en court-circuit	52
6.3.3	Combinaisons	52
7	Les sélections	55
7.1	La structure if	55
7.1.1	L'instruction if	55
7.1.2	L'instruction else	57
7.1.3	Exemple	57
7.2	If / else if	58
7.2.1	Exercice	59
7.3	L'instruction switch	60
7.3.1	Exemple	61
7.3.2	Plusieurs entrées pour une même action	62
7.4	Plusieurs entrées sans instruction break	63
7.5	L'opérateur conditionnel	64
7.5.1	Exercice	64
8	TP : déterminer le jour de la semaine	67
8.1	Objectif	67
8.2	Première étape	67
8.2.1	Déterminer le jour de la semaine	67
8.3	Correction	68
8.4	Deuxième étape	69
8.5	Correction	70
8.6	Troisième et dernière étape	71
8.6.1	Les calendriers Julien et Grégorien	71
8.6.2	Mode de calcul	72
8.7	Correction	72
9	Les boucles	75
9.1	La boucle while	75
9.1.1	Syntaxe	75
9.1.2	Exercice	76
9.2	La boucle do-while	77
9.2.1	Syntaxe	77
9.2.2	Exemple 1	77
9.2.3	Exemple 2	78
9.3	La boucle for	78
9.3.1	Syntaxe	78
9.3.2	Exemple	79
9.3.3	Plusieurs compteurs	80
9.4	Imbrications	80
9.5	Boucles infinies	81
9.6	Exercices	81
9.6.1	Calcul du PGCD de deux nombres	81
9.6.2	Une overdose de lapins	82
9.6.3	Des pieds et des mains pour convertir mille miles	83
9.6.4	Puissances de trois	84
9.6.5	La disparition: le retour	84

10 Les sauts	87
10.1 L'instruction break	87
10.1.1 Exemple	87
10.2 L'instruction continue	88
10.3 L'instruction goto	89
10.4 Exemple	89
10.5 Le dessous des boucles	89
10.6 Goto Hell?	90
11 Les fonctions	91
11.1 Qu'est-ce qu'une fonction?	91
11.2 Définir et utiliser une fonction	92
11.2.1 Le type de retour	93
11.2.2 Les paramètres	93
11.3 Les arguments et les paramètres	94
11.4 Les prototypes	95
11.5 Variables globales et classes de stockage	96
11.5.1 Les variables globales	96
11.6 Les classes de stockage	97
11.6.1 Classe de stockage automatique	97
11.6.2 Classe de stockage statique	97
11.6.3 Modification de la classe de stockage	98
11.7 Exercices	98
11.7.1 Afficher un rectangle	98
11.7.2 Correction	99
11.8 Afficher un triangle	99
11.8.1 Correction	100
11.9 En petites coupures?	100
11.9.1 Correction	100
12 TP : une calculatrice basique	103
12.1 Objectif	103
12.2 Préparation	103
12.2.1 Précisions concernant scanf	103
12.2.2 Les puissances	104
12.2.3 La factorielle	104
12.2.4 Exemple d'utilisation	105
12.3 Derniers conseils	105
12.4 Correction	105
13 Découper son projet	109
13.1 Portée et masquage	109
13.1.1 La notion de portée	109
13.1.2 Au niveau d'un fichier	109
13.2 La notion de masquage	110
13.3 Diviser pour mieux régner	111
13.3.1 Les fonctions	111
13.3.2 Les variables	112
13.4 On m'aurait donc menti?	112
13.5 Les fichiers d'en-têtes	113

14 La gestion d'erreurs (1)	115
14.1 Détection d'erreurs	115
14.1.1 Valeurs de retour	115
14.1.2 Variable globale errno	116
14.2 Prévenir l'utilisateur	117
14.3 Un exemple d'utilisation des valeurs de retour	118
 II Agrégats, mémoire et fichiers	 119
15 Les pointeurs	121
15.1 Présentation	121
15.1.1 Les pointeurs	121
15.1.2 Utilité des pointeurs	121
15.1.3 L'allocation dynamique de mémoire	122
15.2 Déclaration et initialisation	122
15.2.1 Initialisation	123
15.2.2 Pointeur nul	123
15.3 Utilisation	124
15.3.1 Indirection (ou déréférencement)	124
15.3.2 Passage comme argument	125
15.4 Retour de fonction	125
15.5 Pointeur de pointeur	126
15.6 Pointeurs génériques et affichage	126
15.7 Afficher une adresse	127
15.8 Exercice	127
15.9 Correction	127
 16 Les structures	 129
16.1 Définition, initialisation et utilisation	129
16.1.1 Définition d'une structure	129
16.1.2 Définition d'une variable de type structure	130
16.1.3 Initialisation	130
16.1.4 Accès à un membre	130
16.1.5 Exercice	131
16.2 Structures et pointeurs	131
16.2.1 Accès via un pointeur	132
16.2.2 Adressage	132
16.2.3 Pointeurs sur structures et fonctions	132
16.3 Portée et déclarations	133
16.3.1 Déclarations	134
16.3.2 Les structures interdépendantes	134
16.3.3 Structure qui pointe sur elle-même	135
16.4 Un peu de mémoire	135
16.4.1 L'opérateur sizeof	135
16.4.2 Alignement en mémoire	137
16.4.3 La macrofonction offsetof	138

17 Les tableaux	141
17.1 Les tableaux simples (à une dimension)	141
17.1.1 Définition	141
17.1.2 Initialisation	142
17.1.3 Accès aux éléments d'un tableau	142
17.1.4 Parcours et débordement	145
17.1.5 Tableaux et fonctions	146
17.2 La vérité sur les tableaux	147
17.2.1 Un peu d'histoire	147
17.2.2 Conséquences de l'absence d'un pointeur	148
17.3 Les tableaux multidimensionnels	148
17.3.1 Définition	149
17.4 Initialisation	149
17.4.1 Initialisation avec une longueur explicite	149
17.4.2 Initialisation avec une longueur implicite	149
17.4.3 Utilisation	149
17.4.4 Représentation en mémoire	150
17.4.5 Parcours	150
17.4.6 Tableaux multidimensionnels et fonctions	151
17.5 Exercices	152
17.5.1 Somme des éléments	152
17.5.2 Maximum et minimum	152
17.5.3 Recherche d'un élément	153
17.5.4 Inverser un tableau	153
17.5.5 Produit des lignes	153
17.5.6 Triangle de Pascal	154
18 Les chaînes de caractères	157
18.1 Qu'est-ce qu'une chaîne de caractères ?	157
18.1.1 Représentation en mémoire	157
18.2 Définition, initialisation et utilisation	158
18.2.1 Définition	158
18.2.2 Initialisation	158
18.2.3 Utilisation	160
18.3 Afficher et récupérer une chaîne de caractères	160
18.3.1 Printf	161
18.3.2 Scanf	161
18.4 Lire et écrire depuis et dans une chaîne de caractères	163
18.4.1 La fonction sprintf	163
18.4.2 La fonction sscanf	164
18.5 Les classes de caractères	164
18.6 Exercices	166
18.6.1 Palindrome	166
18.6.2 Compter les parenthèses	166
19 TP : l'en-tête <string.h>	169
19.1 Préparation	169
19.1.1 strlen	169
19.1.2 strcpy	169
19.1.3 strcat	170
19.1.4 strcmp	170
19.1.5 strchr	171

19.1.6	strpbrk	171
19.1.7	strstr	172
19.1.8	strlen	172
19.1.9	strcpy	173
19.1.10	strcat	173
19.1.11	strcmp	173
19.1.12	strchr	173
19.1.13	strpbrk	173
19.1.14	strstr	174
19.2	Pour aller plus loin : strtok	174
19.2.1	Correction	175
20	L'allocation dynamique	177
20.1	La notion d'objet	177
20.2	Malloc et consœurs	178
20.2.1	malloc	178
20.2.2	free	180
20.2.3	calloc	181
20.2.4	realloc	181
20.3	Les tableaux multidimensionnels	182
20.3.1	Allocation en un bloc	182
20.3.2	Allocation de plusieurs tableaux	183
21	Les fichiers (1)	185
21.1	Les fichiers	185
21.1.1	Extension de fichier	185
21.1.2	Système de fichiers	185
21.2	Les flux : un peu de théorie	186
21.2.1	Pourquoi utiliser des flux ?	186
21.2.2	stdin, stdout et stderr	187
21.3	Ouverture et fermeture d'un flux	187
21.4	La fonction fopen	187
21.4.1	La fonction fclose	188
21.5	Écriture vers un flux de texte	189
21.5.1	Écrire un caractère	189
21.5.2	Écrire une ligne	190
21.5.3	La fonction fprintf	191
21.6	Lecture depuis un flux de texte	191
21.6.1	Récupérer un caractère	191
21.6.2	Récupérer une ligne	192
21.6.3	La fonction fscanf	193
21.7	Écriture vers un flux binaire	193
21.7.1	Écrire un multiplet	193
21.7.2	Écrire une suite de multiplets	193
21.8	Lecture depuis un flux binaire	194
21.8.1	Lire un multiplet	194
21.8.2	Lire une suite de multiplets	194

22 Les fichiers (2)	197
22.1 Détection d'erreurs et fin de fichier	197
22.1.1 La fonction feof	197
22.1.2 La fonction ferror	197
22.1.3 Exemple	197
22.1.4 La fonction ftell	198
22.1.5 La fonction fseek	198
22.1.6 La fonction rewind	199
22.2 La temporisation	199
22.2.1 Introduction	199
22.2.2 Intérargir avec la temporisation	201
22.3 Flux ouverts en lecture et écriture	203
23 Le préprocesseur	205
23.1 Les inclusions	205
23.2 Les macroconstantes	206
23.2.1 Substitutions de constantes	206
23.2.2 Substitutions d'instructions	207
23.2.3 Macros dans d'autres macros	208
23.2.4 Définition sur plusieurs lignes	209
23.2.5 Annuler une définition	211
23.3 Les macrofonctions	211
23.3.1 Priorité des opérations	212
23.3.2 Les effets de bords	212
23.4 Les directives conditionnelles	213
23.4.1 Les directives #if, #elif et #else	213
23.4.2 Les directives #ifdef et #ifndef	214
24 TP : un Puissance 4	217
24.1 Première étape : le jeu	217
24.2 Correction	218
24.3 Deuxième étape : une petite IA	224
24.3.1 Introduction	224
24.3.2 Tirage au sort	225
24.4 Correction	227
24.5 Troisième et dernière étape : un système de sauvegarde/restauration	234
24.6 Correction	234
25 La gestion d'erreur (2)	245
25.1 Gestion de ressources	245
25.1.1 Allocation de ressources	245
25.1.2 Utilisation de ressources	247
25.2 Fin d'un programme	248
25.2.1 Terminaison normale	248
25.2.2 Terminaison anormale	249
25.3 Les assertions	249
25.3.1 La macrofonction assert	249
25.3.2 Suppression des assertions	250
25.4 Les fonctions strerror et perror	251

III	Notions avancées	253
26	La représentation des types	255
26.1	La représentation des entiers	255
26.1.1	Les entiers non signés	255
26.1.2	Les entiers signés	256
26.1.3	Les bits de bourrages	258
26.2	La représentation des flottants	258
26.2.1	Première approche	258
26.2.2	Une histoire de virgule flottante	259
26.2.3	Formalisation	260
26.3	La représentation des pointeurs	260
26.4	Ordre des multipléts et des bits	260
26.4.1	Ordre des multipléts	261
26.4.2	Ordre des bits	261
26.4.3	Applications	262
26.5	Les fonctions memset, memcpy, memmove et memcmp	263
26.5.1	La fonction memset	263
26.5.2	Les fonctions memcpy et memmove	264
26.5.3	La fonction memcmp	264
27	Les limites des types	267
27.1	Les limites des types	267
27.1.1	L'en-tête <limits.h>	267
27.1.2	L'en-tête <float.h>	268
27.2	Les dépassements de capacité	269
27.2.1	Dépassement lors d'une opération	269
27.2.2	Dépassement lors d'une conversion	270
27.3	Gérer les dépassements entiers	271
27.3.1	Préparation	271
27.3.2	L'addition	271
27.3.3	La soustraction	272
27.3.4	La négation	273
27.3.5	La multiplication	273
27.3.6	La division	274
27.3.7	Le modulo	274
27.4	Gérer les dépassements flottants	275
27.4.1	L'addition	275
27.4.2	La soustraction	276
27.4.3	La multiplication	277
27.4.4	La division	278
28	Manipulation des bits	279
28.1	Les opérateurs de manipulation des bits	279
28.1.1	Présentation	279
28.1.2	Les opérateurs « et », « ou inclusif » et « ou exclusif »	279
28.1.3	L'opérateur de négation ou de complément	280
28.1.4	Les opérateurs de décalage	281
28.1.5	Opérateurs combinés	282
28.2	Masques et champs de bits	282
28.2.1	Les masques	282
28.2.2	Les champs de bits	284

28.3 Les drapeaux	287
28.4 Exercices	289
28.4.1 Obtenir la valeur maximale d'un type non signé	289
28.4.2 Afficher la représentation en base deux d'un entier	290
28.4.3 Déterminer si un nombre est une puissances de deux	291
29 Jeux de caractères et encodages	293
29.1 Les jeux de caractères et les encodages	293
29.1.1 Introduction	293
29.1.2 Ce que dit la norme	293
29.1.3 Caractères garantis	294
29.2 Les caractères larges	295
29.2.1 Mise en situation	295
29.2.2 Les caractères larges	296
29.2.3 Des fonctions dans tous leurs états	301
29.3 Internationalisation et localisation	301
29.3.1 L'internationalisation	302
29.3.2 La localisation	302
29.3.3 La fonction setlocale	302
29.3.4 Exemple	303
30 Les énumérations	305
30.1 Définition	305
30.2 Utilisation	306
31 Les unions	309
31.1 Définition	309
31.2 Utilisation	310
32 Les définitions de type	313
32.1 Définition et utilisation	313
33 Les pointeurs de fonction	315
33.1 Déclaration et initialisation	315
33.1.1 Initialisation	315
33.2 Utilisation	316
33.2.1 Déréférencement	316
33.2.2 Passage en argument	316
33.2.3 Retour de fonction	317
33.3 Pointeurs de fonction et pointeurs génériques	318
33.3.1 La promotion des arguments	319
33.3.2 Les pointeurs nuls	320
34 Les fonctions à nombre variable d'arguments	321
34.1 Présentation	321
34.2 L'en-tête <stdarg.h>	322
34.3 Méthodes pour déterminer le nombre et le type des arguments	323
34.3.1 Les chaînes de formats	323
34.3.2 Les suites d'arguments de même type	323
34.3.3 Emploi d'un pivot	324

35 T.P. : un allocateur statique de mémoire	327
35.1 Objectif	327
35.2 Première étape : allouer de la mémoire	327
35.2.1 Mémoire statique	327
35.3 Correction	328
35.4 Deuxième étape : libérer de la mémoire	329
35.4.1 Ajout d'un en-tête	329
35.4.2 Les listes chaînées	330
35.5 Correction	331
35.6 Troisième étape : fragmentation et défragmentation	333
35.6.1 Fusion de blocs	334
35.7 Correction	334
Index	339
Table des figures	339
Liste des tableaux	341
Table des matières	353