Foire aux questions sur le langage C

19 avril 2003

Résumé

Ce document regroupe les questions les plus fréquemment posées (avec les réponses) du groupe de discussion francophone news:fr.comp.lang.c sur le langage C. Cette FAQ (voir la question 2.1, page 9) est basée sur celle de news:comp.lang.c maintenue par Steve Summit (http://www.eskimo.com/~scs/C-faq/top.html).

Une version HTML de ce document est disponible à l'adresse suivante : http://www.isty-info.uvsq.fr/~rumeau/fclc/.

Une version texte de ce document est disponible : http://www.isty-info.uvsq.fr/~rumeau/fclc/fclc.txt. Les sources XML sont également récupérables ici : http://www.isty-info.uvsq.fr/~rumeau/fclc/pack.tar.gz.

Une version PDF de la FAQ est à nouveau disponible ici : http://www.isty-info.uvsq.fr/~rumeau/fclc/fclc.pdf. Enfin, il existe aussi une version poscript : http://www.isty-info.uvsq.fr/~rumeau/fclc/fclc.ps.

Même si ce document n'est plus si récent que ça, il est probable qu'il y traine encore quelques coquilles ou erreurs. Aussi, si vous en trouvez, n'hésitez pas à les indiquer aux mainteneurs (voir la question 2.5, page 10).

Derniers changements

- Version **2.15**, le **18/04/2003**
  Quelques modifications mineures et corrections de liens morts.
- Version **2.14**, le **18/01/2003**
  Corrections dans le source XML
  Une version PDF est à nouveau disponible
  Modification de la question 14.3, page 75
- Version **2.13**, le **11/11/2002**
  Correction du code de la question 14.12, page 78
  Ajout de la question 14.8, page 77
  Correction des questions 3.10, page 19 et 15.10, page 84.
- Version **2.12**, le **27/08/2002**
  Correction de l'exemple dans la question 14.2, page 74.
- Version **2.11**, le **03/07/2002**
  Précisions sur la question 13.9, page 66 Et quelques corrections typographiques.
- Version **2.10**, le **09/04/2002**
  Compléments sur la question 3.6, page 14
  Ajout de la question 5.8, page 30
  Ajout de la question 11.10, page 56
  Ajout de la question 9.10, page 50
  De nombreuses autres corrections et modifications.

Table des matières

Chapitre 1

Copyright (Droits de reproduction)

1.1 Copyright de la FAQ de comp.lang.c

« The comp.lang.c FAQ list is Copyright 1990-1999 by Steve Summit. Content from the book *C Programming FAQs : Frequently Asked Questions* is made available here by permission of the author and the publisher as a service to the community. It is intended to complement the use of the published text and is protected by international copyright laws. The content is made available here and may be accessed freely for personal use but may not be republished without permission. »

1.2 Qu'en est-il de ce document ?

- Steve Summit a rédigé la version anglo-saxonne sur laquelle nous nous sommes basés. On en a traduit de longs passages, réorganisé, annoté, repensé une partie et réécrit complétement d'autres paragraphes, ce avec son consentemment.
- Les auteurs comme les contributeurs de cette FAQ ne garantissent rien, ni que les conseils donnés ici fonctionnent, ni que le code compile et fonctionne correctement, ni que votre machine ne va pas s'autodétruire instantanément après lecture de ce message. Autrement dit, vous êtes entièrement responsable de ce que vous faites des informations données ici. En cas de pépin, vous ne pouvez vous en prendre qu'à vous même.
- Les copies conformes et versions intégrales de ce document sont autorisées sur tout support pour peu que cette notice soit préservée. Une utilisation commerciale devra faire l'objet d'une autorisation préalable, notamment de Steve Summit.

Chapitre 2

Introduction

2.1 Qu'est-ce qu'une FAQ ?

C'est une *Foire Aux Questions* (les anglos-saxons disent *Frequent Asked Questions* ou questions fréquemment posées). Elle regroupe les réponses aux questions récurrentes sur news:fr.comp.lang.c.

2.2 Qui la maintient ?

Elle est rédigée à l'initiative de Guillaume Rumeau (mailto:guillaume.rumeau@wanadoo.fr).

2.3 Qui y contribue ?

Les rédacteurs de la FAQ sont :
– Guillaume Rumeau (mailto:guillaume.rumeau@wanadoo.fr),
– Thomas Pornin (mailto:Thomas.Pornin@ens.fr),
– Pascal Cabaud (mailto:pascal.cabaud@wanadoo.fr),
mais elle ne saurait exister sans les contributions de :

« After », « EpSylOn », « ironfil », « MacLord », Edgar Bonnet, Erwan David, Thomas Deniau, Emmanuel Delahaye, Gabriel Dos Reis, Laurent Dupuis, Horst Kraemer, Antoine Leca, Vincent Lefèvre, Fabien Le Lez, Éric Lévénez, Serge Paccalin, Yves Roman, Michel Simian, et tous les autres...

Merci aussi à Thomas Baruchel pour ces nombreuses corrections orthographiques et typographiques, et à Stéphane Muller pour son script python qui permet d'avoir des versions postscript et pdf de la FAQ.

et n'oublions pas bien sûr Steve Summit.

Remarque

Si vous pensez qu'un nom a été injustement omis, n'hésitez pas à nous le faire savoir, nous nous ferons un plaisir de l'ajouter.

2.4 Où puis-je la trouver ?

à la page de Guillaume Rumeau : http://www.isty-info.uvsq.fr/~rumeau/fclc/ et deux fois par mois sur news:fr.comp.lang.c et news:fr.usenet.reponses.

2.5 J'ai trouvé une erreur !

Ce document étant rédigé par des humains, il peut contenir des erreurs. Vous êtes vivement invités à les signaler soit en postant sur news:fr.comp.lang.c soit en écrivant à l'un des (aux) rédacteurs.

(voir aussi 2.3, page 9)

2.6 Et mes questions ?

Il faut commencer par les reformuler le plus précisément possible (par exemple les reformuler en questions moins ouvertes) ; décrire les problèmes aide beacoup.

Quand on commence à avoir une idée, chercher les mots clefs décrivant le problème à l'aide de moteurs de recherche, par exemple http://www.google.com/. Regardez aussi ce que sait faire http://www.copernic.com/fr.

Si vous n'avez toujours pas de reponse, analysez les différents forums disponibles, et posez la question dans celui qui vous semblera le plus adapté ; si il y a plusieurs questions qui semblent pouvoir aller dans plusieurs forums, rédigez plusieurs messages (il y aura plus de réponses, car les réponses seront moins longues à écrire ...).

Si la question est très technique et n'obtient pas de réponse valable, faites l'effort de la re-rédiger en anglais et/ou demandez gentiment une traduction si vous n'êtes pas sûr de votre anglais, il y a suffisament de francophones sur les groupes internationaux pour qu'une bonne âme vous rende service.

Avant de poster (sur news:fr.comp.lang.c), mieux vaut se demander si :
- la question porte-t-elle sur le C ISO ? *cf.* 3.7, page 15.
- la réponse est-elle dans la doc' ? dans mon bouquin ? Bonne lecture !
- la réponse est-elle dans la FAQ ? Bonne lecture !
- la réponse a-t-elle déjà été donnée ? Pour le savoir, il faut chercher sur http://groups.google.com/.

Comme il est dit et redit quotidiennement sur news:fr.comp.lang.c, le forum ne traite ni de graphisme, ni de *Windows*.

À ce sujet, rappelons que la FAQ de news:fr.comp.os.ms-windows.programmation est disponible ici : http://www.usenet-fr.news.eu.org/fr.usenet.reponses/comp/os/faq-winprog.html.

(Merci à Emmanuel Delahaye pour l'URL de copernic et à Antoine Leca de m'avoir prêté sa plume ;-)).

2.7 Dois-je poster sur fr.comp.lang.c ?

Ainsi que le stipule la charte du groupe, les seules questions traitées sur news:fr.comp.lang.c portent sur le C ISO. Pour ce qui a trait avec la programmation *Be*, *DOS*, *Mac*, *Unix*, *Windows*, il y a des forums dédiés.

En particulier,
– news:fr.comp.os.ms-windows.programmation pour la programmation sous *Windows*,
– news:fr.comp.sys.mac.programmation pour la programmation sous *MacOS*, *MacOS X*.
Pour les autres systèmes, il n'y a pas de forum spécifique dans la hierarchie news:fr.*. Pour ce qui concerne les *Unix*, news:fr.comp.os.unix et/ou news:fr.comp.os.bsd (ou encore news:news:fr.comp.sys.next) devraient faire l'affaire.

Dans la hiérarchie internationale, on trouve :
– news:comp.os.msdos.programmer pour *DOS*,
– news:comp.unix.programmer pour *Unix*,
– news:comp.sys.be.programmer pour *Be*,
– news:comp.os.os2.programmer pour *OS/2*,
Enfin, en ce qui concerne la programmation graphique, il existe désormais un forum dédié à ce sujet dans la hiérarchie fr : news:fr.comp.graphisme.programmation

2.8 Comment poster sur fr.comp.lang.c ?

En n'envoyant pas de pièce jointe, en postant en texte ASCII. Les accents (ISO-8859-1) sont acceptés. Outre les pièces jointes, le HTML, le *quoted printable* et tout ce qui n'est pas *text/plain* est à proscrire (pas mal de serveurs de *news* supprimeront ces contributions indésirables sans autre forme de procès).

Plus la question est précise, meilleure sera la réponse. On peut utiliser des balises dans le sujet, une liste est maintenue par Alexandre Lenoir, disponible là : http://www.planete.net/~alenoir/fcsm.html. Ne donnez pas de noms propres dans les sujets !

Il est de bon ton de poster les morceaux de code problématique. Veillez à poster du code qui compile (sauf s'il ne s'agit que de quelques lignes). Si le programme est trop gros, donnez une URL où on peut le lire.

Pour répondre, veillez à ne citer que le strict nécessaire et à répondre après le message ou passage cité.

Je vous rappelle qu'il est très impoli, pour des raisons évidentes, de demander une réponse par mail ; et en général vous n'aurez pas satisfaction. Si vraiment vous y tenez, tentez votre chance avec le service fourni ici : http://francette.net/bdr.html (je n'ai pas testé).

2.9 Comment comprendre le langage utilisé sur fr.comp.lang.c ?

Ceci n'est pas une FAQ sur Usenet mais sur le langage C donc on va faire court :

– `OEQLC` signifie « Où Est la Question sur Langage C» autrement dit, votre question est hors-sujet (*off topic*),
– `OT` ou `HS` signifie « Hors-Sujet » autrement dit, c'est du bruit,
– `RTFM` signifie « *Read The F*cking Manual* » autrement dit, c'est dans tout bouquin digne de ce nom (voir 3.9, page 18 et 3.10, page 19). Pour les âmes sensibles, on traduit parfois par « *Read The Fine Manual* », ou en français par « *Regarde Ton Fichu Manuel* ».

Chapitre 3

Le langage C

3.1 Qu'est-ce que le langage C ?

Donner une définition du C est assez difficile, je vous propose celle-ci (avec laquelle je me suis le moins fait insulté `;-)`) :

C'est un langage structuré, généralement compilé, de haut niveau.

Pour le « haut niveau », cela dépend un peu du point de vue en fait. À la fin des années 70, les réfractaires disaient de lui que c'était « encore un assembleur ». D'autres le trouvent donc de bas niveau ... On vous renvoie à l'introduction et aux avant-propos de K&R (*cf.* 3.9, page 18).

Il dispose d'une bibliothèque standard (normes ANSI, ISO, IEEE, AFNOR,...) permettant un minimum d'interactions avec la machine. Il a été normalisé (*cf.* 3.7, page 15) ce qui permet de recompiler un source (n'utilisant que la bibliothèque standard) sur n'importe quelle machine disposant d'un compilateur respectueux de la norme.

Cett dernière ne porte que sur le langage proprement dit et sur le contenu de la bibliothèque standard. Cette dernière ne contient que le strict minimum pour interagir avec la machine. Ainsi, peut-on manipuler du texte et des fichiers, déterminer le temps de calcul ou gérer la mémoire mais guère plus. Le reste est à la charge du programmeur, ou des bibliothèques spécifiques du système.

Au vu de ce qui précède, on peut donc donner cette autre définition :

Le C est un langage de programmation dont la structure est proche de la machine de Von Neumann.

Donc ce qui importe, ce n'est pas tant qu'on puisse recompiler notre programme type sans modification sur toutes les plateformes, c'est la « sémantique ». Le fait d'écrire en C standard n'implique nullement que le programme soit portable en ce sens qu'il a la même sémantique sur tous les compilateurs. Il est important de noter que la définition de C définit une machine abstraite « paramétrée » – les paramètres variant d'un compilateur à un autre.

Rappelons ici encore que le langage ne gère ni la souris, ni l'écran, ni votre store électrique (pardon, nucléaire) dernier cri. Tout cela est du ressort de votre OS.

Voir aussi 3.5, page 14.

Merci à Emmanuel Delahaye, Gabriel Dos Reis, Vincent Lefèvre, Thomas Pornin et tous les autres pour la rédaction de cet article.

3.2 À quoi ça sert ?

À créer ses propres logiciels, selon ses besoins, de manière portable *ie.* indépendemment de la machine et du système d'exploitation.

3.3 D'où vient le C ?

Il a été créé par Brian Kernighan et Dennis Ritchie, sur les cendres du BCPL (*Basic Combined Programming Language* de Martin Richard, le BCPL étant une simplification du CPL, *Cambridge Programming Language*) et du B (langage expérimental de Ken Thompson), dans les années 70 lorsqu'ils écrivaient *Unix*.

Voir aussi 3.7, page 15.

3.4 Que peut-on faire en C ?

On peut tout faire ou presque. On peut créer son propre système d'exploitation (les *Unix-like* sont encore écrits en C pour la majorité), son interface graphique, sa base de données, son *driver* (pilote) pour la dernière machine à café USB, *etc.*

Tout compilateur C est fourni avec une bibliothèque de fonctions, en principe standard.

Voir aussi 3.5, page 14.

3.5 Portabilité, matériel, système ...

Si la norme du C permet une bonne portabilité, il faut noter que l'on peut faire des choses parfaitement dépendantes de la cible. Ainsi, les *Unix* sont-ils écrits en C. Mais point n'est besoin d'utiliser des API[1] exotiques pour perdre en portabilité, présupposer qu'un `char` fait 8 bits est l'exemple le plus flagrant (*cf.* 16.2, page 85). Un autre exemple classique est de croire que toutes les machines supportent l'*ASCII* ...

3.6 Et le C++ dans tout ça ?

Le C++ est un langage à objets basé sur le C. Il y a des différences suffisantes pour qu'il s'agisse d'un autre langage, ayant son forum propre : `news:fr.comp.lang.c++`.

[1] Application Programming Interface i.e. interface de programmation d'applications littéralement ; autrement-dit bibliothèque.

Il existe un autre langage à objets basé sur le C. Il s'agit d'Objective-C. Il est principalement utilisé dans *Mac OS X* et dans *GNUStep*. Son forum est news:fr.comp.lang.objective-c.

3.7 ISO, ANSI, K&R, ..., *qu'es aquo* ?

Les origines

Le langage C est le fruit des efforts conjoints de Brian Kernighan, Denis Ritchie et Ken Thompson. Le dernier dirigeait le projet de réécriture de *Multics* (un OS multi-utilisateurs, projet abandonné) et il voulait créer un OS qui soit portable répondant à ses attentes, permettant un accès simple aux périphériques. De fil en aiguille, c'est devenu un système largement indépendant du *hardware*. À l'époque les plateformes étaient toutes si différentes qu'il était commun de donner ses sources et la moindre recompilation d'une machine à une autre demandait souvent un effort de portage.

Pour faire ce système, il en est venu à penser *Unix* de sorte que les deux seuls éléments qui dépendent du *hardware* se résument au strict minimum à savoir le compilateur et le noyau. Il lui fallait un langage d'assez bas niveau et simple. Se basant sur le BCPL, Thompson a donc mis au point le langage B (pour *Unics*, 1969-1972) puis Ritchie l'a amélioré pour en faire le langage C (*Unix*, 1973).

Voir aussi 3.3, page 14 et surtout le papier de Dennis Ritchie : http://cm.bell-labs.com/cm/cs/who/dmr/chist.html pour plus de détails.

La popularité du C tient alors autant à sa simplicité qu'à la pénétration d'*Unix* (et donc l'apparition de compilateurs C sur les machines *ie.* la nouvelle portabilité des programmes). Des compilateurs sont alors assez vite apparus sur d'autres plateformes qu'*Unix*, contribuant ainsi à la diffusion (un peu anarchique) du langage.

En 1978, Brian W. Kernighan et Denis M. Ritchie ont publié *The C Programming Language* (ISBN :0131101633) (On trouve tous les numéros ISBN des différentes éditions sur http://cm.bell-labs.com/cm/cs/cbook/index.html).

Dès lors, les compilateurs ont commencé à suivre les recommandations et indications des auteurs. Cet ouvrage a fait office de norme pendant longtemps, le langage qui y est décrit s'appelle le C K&R, en référence aux 2 auteurs.

La normalisation

Devant la popularité du C, l'*American National Standard Institut* (http://www.ansi.org) charge en 1983 le comité X3J11 de standardiser le langage C. On parle à ce moment-là de C pré-ANSI. Après un processus long et complexe, le travail du comité a finalement été approuvé : le 14 décembre 1989, le standard ANSI X3.159-1989 (ou C89) est né. Il est publié au printemps 1990.

Entre-temps, en 1988, durant la période de travail du comité, la deuxième édition du K&R a été publiée (ISBN : 0131103709). Elle a été complètement réécrite et on y a ajouté des exemples et des exercices afin de clarifier l'implémentation de certaines constructions complexes du langage.

Dans sa plus grande partie, le standard C ANSI de 1989 officialise les pratiques existantes,

en ajoutant quelques nouveautés provenant du C++ comme les prototypes de fonctions et le support de jeux de caractères internationaux (notamment les très controversées séquences trigraphes). Le standard C ANSI décrit aussi les routines pour le support des bibliothèques d'exécution du C.

L'*International organization for standardization* (http://www.iso.ch) a adopté en 1990 ce standard en tant que standard international sous le nom de ISO/IEC 9899 :1990 (ou C90). Ce standard ISO remplace le précédent standard ANSI (C89) même à l'intérieur des USA, car rappelons-le, ANSI est une organisation nationale, et non internationale comme l'ISO. Aux USA, on parle alors de ANSI/ISO 9899-1990 [1992], en France de ISO/CEI 9899 :1990.

Détails sur la normalisation

Durant les années 1990, lorsqu'on parle de C89 (C ANSI) ou de C90 (C ISO), ce sont deux appellations différentes pour en fait une seule et même norme. Il existe aussi une norme européenne et une française (AFNOR) dont on entend beaucoup moins parler, qui sont aussi semblables à la norme ISO. Noter que la norme française (AFNOR) est parfaitement identique à l'ISO, l'AFNOR se contentant de publier l'ISO (en anglais) en guise de norme française (le site du groupe de normalisation du C à l'AFNOR : http://forum.afnor.fr/afnor/WORK/AFNOR/GPN2/Z65B/, ne pas se fier aux apparences, il s'agit du groupe sur le C, la page est un peu vieille et le groupe — réduit passé un temps à Antoine Leca — a été recueilli par le groupe chargé du C++).

Les standards ISO, en tant que tel, sont sujets à des révisions, par la diffusion de *« Technical Corrigenda »* [2] et de *« Normative Addenda »*[3]. C'est ainsi qu'en 1995, le *Normative Addendum* 1 (NA1) (http://www.lysator.liu.se/c/na1.html) parfois appelé *Amendment 1 (AM1)* fut approuvé en 1995. Il ajouta environ 50 pages de spécifications diverses concernant notamment de nouvelles fonctions dans la bibliothèque standard pour l'internationalisation, et les séquences digraphes pour le jeu de caractères ISO 646, autorisant ainsi les terminaux ne possédant pas certains caractères à utiliser une écriture alternative (`<%` `%>` pour `{` et `}` ou encore `<` `:` `:>` pour `[` et `]`).

Peu de temps après, toujours en 1995, le *Technical Corrigendum 1 (TCOR1)* (http://anubis.dkuug.dk/JTC1/SC22/WG14/www/docs/tc1.htm) fut approuvé et modifia le standard ISO en environ 40 points, la plupart d'entre eux étant des corrections mineures ou des clarifications. En 1996, on publia aussi le TCOR2 (http://anubis.dkuug.dk/JTC1/SC22/WG14/www/docs/tc2.htm) qui apporta des changements encore plus mineurs que le TCOR1. TCOR2 reformulait certains points abscons.

À partir de 1997, on désigne par C95 l'ensemble des documents TCOR1, TCOR2 et AMD1 et de la norme C90. Le terme C95 est utilisé dans le *« rationale »* de C99.

En fait ce n'est pas directement l'ISO qui travaille sur les standards, elle ne fait que les approuver conjointement avec l'IEC (http://www.iec.ch). C'est pour cette raison que les standards ISO du C commencent par ISO/IEC... De plus, l'ISO charge des comités techniques de travailler dans tels et tels domaines. En l'occurrence, le JTC1 (http://www.jtc1.org) est le comité spécialisé dans le domaine informatique. Le JTC1 à son tour répartit le travail dans plusieurs sous-comités : celui qui nous intéresse est le SC22,

[2] « Rectificatif technique ».
[3] « Amendement ».

dont le but est la standardisation des technologies de l'information. Or le SC22 lui-même est subdivisé en *Working Groups*, le WG14 étant celui qui est en relation avec le C.

Finalement, c'est le ISO/IEC JTC1/SC22/WG14 qui rédige la norme ISO du C, le SC2 approuve alors le projet final (FDIS), puis le transmet au JTC1 qui approuve la nouvelle norme ISO.

En réalité, le groupe de travail WG14 est composé d'organismes nationaux — tels ANSI (le plus actif), AFNOR, BSI, CSA, DS, ... — représentants les pays prenant part à la normalisation (je vous passe les détails de pays votants, observateurs, et autres). Chacun de ces organismes travaille sur le langage. L'ANSI dispose elle aussi d'un comité spécialisé dans le domaine informatique : le X3 (http://www.x3.org), qui depuis 1996 s'appelle NCITS (prononcez *insights* en anglais) (http://www.ncits.org) pour *National comittee for Information Technology Standards*. Lui aussi dispose de comités techniques qui travaillent chacun dans un domaine particulier : le J11 (http://www.ncits.org/tc\protect\T1\textunderscorehome/j11.htm) a en charge le langage C. C'est donc le X3J11 qui développe la norme C ANSI aux USA, et qui travaille avec le WG14.

Il se trouve qu'en 1993, lors des réunions bi-annuelles entre le WG14 (ISO) et le X3J11 (ANSI), tout le monde s'est accordé pour dire,

– que la révision (prévue dans les textes ISO) de 1995 ne se ferait pas, et
– que la révision de 2000 environ aboutirait à une nouvelle version de la norme (le futur C9X).

L'idée a alors émergée de créer un nouveau standard du C qui regrouperait le C90, le NA1, le TCOR1 et le TCOR2, apporterait d'autres modifications afin de maintenir le C en phase avec les techniques de programmation d'aujourd'hui, et qui minimiserait les incompatibilités avec le C++, sans pour autant vouloir transformer le C en C++. Ce projet de nouveau standard du C a pris le nom de code C9X avec l'intention qu'il serait publié dans la fin des années 1990 (http://anubis.dkuug.dk/JTC1/SC22/WG14/www/charter.html).

Vers la fin de la normalisation de C99, SC22/WG21 – le groupe de travail qui s'occupe de C++ – a adressé une requête formelle (par l'intermédiaire du bureau SC22) à SC22/WG14 pour documenter les éventuelles incompatibilités introduites par C99 par rapport à C++ (SC22/WG21 l'avait fait par rapport à C90). SC22/WG14 a répondu qu'il n'avait ni le temps nécessaire ni la compétence pour faire cela.

Cependant, un tel travail (inspiré partiellement de ce que C++ a déjà fait) a été entrepris à titre personel par David Tribble dont la contribution se trouve ici : http://www.david.tribble.com/text/cdiffs.htm.

Tout au long du projet C9X, des « *drafts* » (brouillons) du projet sont distribués afin que tout le monde puisse donner son avis et, le cas échéant, revoir certaines parties. Le dernier *draft* disponible est le document n869 (http://www.dkuug.dk/jtc1/sc22/wg14/www/docs/n869/) datant de janvier 1999. Ce document est celui le plus proche de la norme et que l'on peut obtenir gratuitement : il définit C9X, le projet de la norme.

Dernières nouvelles

Très récemment, le 1er décembre 1999, la norme officielle a été adoptée par l'ISO sous le

nom de ISO/IEC 9899 :1999, ou plus simplement C99. Elle a aussi été publiée par l'ANSI, qui travaille conjointement avec l'ISO, sous le nom de ANSI/ISO/IEC 9899-1999, mais que l'on appelle C2k.

À l'heure actuelle, C99, qui est équivalent à C2k, n'est pas encore totalement supportée par les compilateurs. Il faut un certain temps pour implémenter toutes les nouvelles fonctionnalités de C99. Tout ce dont on peut être sûr, c'est que n'importe quel bon compilateur supporte au moins la norme C90.

Voici d'ailleurs au passage quelques unes des nouveautés de C99 :
– tableau à longueur variable,
– support des nombres complexes grâce à `complex.h`,
– types `long long int` et `unsigned long long int` d'au moins 64 bits,
– famille de fonctions `vscanf`,
– les fameux commentaires à-la-C++ `//`,
– les familles de fonctions `snprintf`,
– le type booléen,
– etc.
Théoriquement, ISO révise les normes tous les cinq (5) ans. Les groupes de travail n'ont pas besoin d'attendre les cinq ans avant de commencer à travailler sur les éventuelles extensions. Cependant le travail de normalisation prend un certain temps — pensez par exemple que ANSI a débuté le travail de normalisation de C89 en 1983 et n'a fini qu'en 1989. Il est possible qu'un C04 soit publié en 2004, ce serait C99 augmenté de quelques amendements. Le travail de normalisation est long et pénible par moment.

Un grand merci à « After » pour le brouillon de cet article et à Gabriel Dos Reis, Éric Lévénez et Antoine Leca pour leurs relectures avisées.

3.8 De quoi ai-je besoin pour programmer en C ?

D'un éditeur de texte basique, d'un compilateur (voir 4.2, page 22), d'un bon bouquin (voir 3.9, page 18), d'un débogueur et de beaucoup de patience. En principe, compilateur, bibliothèque(s), (débogueurs) et doc' sont fournis ensembles.

Pour apprendre le C, il vous faudra un bon compilateur, de la doc' papier, un bon dictionnaire d'anglais et un stock d'aspirine `;-)`

3.9 Quel(s) bouquin(s) ?

Le livre que tout programmeur C se doit de connaître et d'avoir sur son bureau est Kernighan B.W. & Ritchie D.M. (1997), *Le langage C Norme ANSI*, 2ème édition, Dunod, Paris.

On trouvera les exercices corrigés du précédent dans : Tondo C.L. & Gimpel S.E. (2000), *Exercices corrigés sur le langage C*, Dunod, Paris.

En complément, Braquelaire J.-P. (2000), *Méthodologie de la programmation en C, Bibliothèque standard, API POSIX*, 3ème édition, Dunod, Paris. sera une excellente ressource.

Enfin, citons l'excellent KERNIGHAN B.W. & PIKE R., (1999) *The Pratice of Programming*, Addison-Wesley, Reading. (http://cm.bell-labs.com/cm/cs/tpop/index.html).

Il arrive souvent au programmeur de devoir résoudre des problèmes d'Algorithmique. Il peut se reporter à la bible en la matière : KNUTH D.E. (1997), *The Art of Computer Programming*, third edition, Addison-Wesley, Reading. (communément abrégé en TAoCP). Il y a aussi : SEDGEWICK R. (1991), *Algorithmes en langage C*, InterÉditions, Paris. ou cet autre : LOUDON K. (2000), *Maîtriser les Algorithmes en C*, O'Reilly, Paris.

Une bonne introduction à l'Analyse Numérique en C est : PRESS W.H., FLANNERY B.P., TEUKOLSKY S.A., VETTERLING W.T. (1992), *Numerical Recipes in C, The Art of Scientific Computing*, second edition, Cambridge University Press. (communément abrégé en NR, PFTW ou *Numerical Recipes* selon). C'est disponible en ligne (voir 4.5, page 23).

Il existe aussi : ENGELN-MÜLLGES G. & UHLIG F. (1996), *Numerical Algorithms with C*, Springer, Berlin. (fourni avec les sources et *djgpp*, pour plateformes *Wintel*, sur CD).

3.10 Où trouver...

de la doc' ?

Là par exemple : http://cm.bell-labs.com/cm/cs/who/dmr/, c'est la page de Denis RITCHIE. Il y a aussi celle de Brian KERNIGHAN : http://cm.bell-labs.com/cm/cs/who/bwk/.

On trouve des cours de C sur le *web* en français sur les sites universitaires. Ainsi, on peut citer : ftp://ftp.ltam.lu/TUTORIEL/COURS-C/COURS-C.ZIP, http://www.enseignement.polytechnique.fr/profs/informatique/Eric.Goubault/poly/cours.ps.gz, http://www.loria.fr/~mermet/CoursC/coursC.ps, http://www.enseignement.polytechnique.fr/profs/informatique/Jean-Jacques.Levy/poly/polyx-cori-levy.ps.gz, http://www-inf.int-evry.fr/COURS/COURSC/

On lira aussi très attentivement : ftp://ftp.laas.fr/pub/ii/matthieu/c-superflu/c-superflu.pdf qui contient tout ce qu'il faut savoir pour commencer à programmer proprement en C. Il contient aussi une grosse bibliographie.

Un CD complet en ligne sur le C : http://www.infop6.jussieu.fr/cederoms/Videoc2000/

Les sources du bouquin de BRAQUELAIRE (dernière édition : la 3ème, 2ème tirage...) : http://dept-info.labri.u-bordeaux.fr/~achille/MPC-3/2T/MPC-3-2t.tar.gz

Les sources du bouquin de LOUDON : http://www.editions-oreilly.fr/archives/algoc.zip.

La bibliothèque standard : http://www.dinkumware.com/htm\protect\T1\textunderscorecl/index.html#Table-of-Contents

Le IOCCC est un concours de *hackers* qui récompense chaque année le pire programme C : http://www.ioccc.org/index.html

Un *freezzine* en anglais : http://www.gmonline.demon.co.uk/cscene/

Sur les sites universitaires on trouve toujours des cours en ligne, du code, *etc* ...

Signalons aussi (même si c'est hors-sujet) le fichier ftp://ftp.laas.fr/pub/ii/matthieu/tpp/tpp.ps.gz qui explique comment utiliser *make*. Je recommande aussi vivement l'utilisation d'outils tels que *CVS* : http://www.cvshome.org/docs/ http://www.idealx.org/fr/doc/cvs/cvs.html http://matrix.samizdat.net/serveurs/cvs/tut\protect\T1\textunderscorecvs.html

la norme ?

Là : http://wwwold.dkuug.dk/jtc1/sc22/open/n2794/n2794.txt On peut aussi l'acheter soit auprès de l'AFNOR (très cher) soit en ligne aux USA (environ 20 $ US), ça se passe ici : http://www.cssinfo.com/ncitsgate.html ou encore là http://webstore.ansi.org/ansidocstore/product.asp?sku=ANSI%2FISO%2FIEC+9899%2D1999

(Merci à Antoine Leca).

la FAQ de comp.lang.c ?

Là : http://www.eskimo.com/~scs/C-faq/top.html

les pages des manuels Unix en français ?

On peut trouver les pages de manules en Français pour la plupart des Unix, sous la forme de packages du système. Pour Linux, vous les trouverez ici : http://perso.club-internet.fr/ccb/ ou encore là : http://www.delafond.org/traducmanfr/

Le C et les CGI ?

Tout ce qui concerne le C à propos des CGI est là : http://www.chez.com/nospam/cgi.html Il y a aussi une FAQ ici : http://www.htmlhelp.com/faq/cgifaq.html

(informations bienvenues à mailto:pascal.cabaud@wanadoo.fr)

La Programmation Objet en C ?

Oui, on peut programmer Orienté Objet en C. Voici un document qui présente ces techniques : http://ldeniau.home.cern.ch/ldeniau/html/oopc/oopc.html

Chapitre 4

Outils, environnement de développement et autres gadgets

Note

Cette section a du mal à vivre sans l'aide des lecteurs. Par soucis d'équité et pour limiter un peu le volume d'informations, on s'en tient aux logiciels, programmes, codes, *etc.* libres, gratuits ou du domaine public.

4.1 Environnements de développement intégrés

Apple

Sur *Mac OS*, *MPW* combiné avec le terrible *MacsBug* vous donneront entière satisfaction. Pour *Mac OS X*, Project Builder est l'IDE de choix utilisant GNU CC. http://developer.apple.com/tools/projectbuilder/

Unix

Sur *Unix-like*, *vi[m]* et *[x]emacs*, combinés avec *make*, *GNU CC* et *GNU DB* forment un environnement de développement intégré.

Wintel

Borland fournit l'environnement *Turbo* complet avec éditeur, débogueur, *etc.* Mais c'est passablement vieux : la version 2.0 distribuée sur les sites de *Borland* date de 1988, il s'agit donc d'un compilateur pré-ANSI (*cf.* 3.7, page 15). Si l'on tient à utiliser du matériel *Borland*, *Turbo C++* 1.01 sera plus confortable et plus conforme à la norme (en ayant récupéré l'archive *BC20P2.ZIP* qui traîne un peu partout).

Mieux vaut se rabattre sur le couple *djgpp-RHIDE* (tentez votre chance ici : http://www.rhide.com), qui d'ailleurs existe sous Linux. Il y a aussi *Source Navigator* disponible ici : http://sources.redhat.com/sourcenav/.

On pourra aussi utiliser *NTemacs* (*cf.* http://www.linux-france.org/article/appli/emacs/Gnus+Emacs/Windows/emacs.html).

Signalons enfin l'existence d'un IDE assez complet, gratuit et open-source : *Dev-C++* ; il

est basé sur *MinGW* (mais peut aussi utiliser *Cygwin*) pour la compilation et le debugger. On le retrouvera à l'adresse : http://www.bloodshed.net/devcpp.html.

(voir aussi 4.2, page 22).

4.2 Compilateurs

Le choix dépend du système d'exploitation et du processeur cible.

Apple

Pour les machines sous *Mac OS*, le compilateur d'*Apple* est gratuit, il est là : ftp://ftp.apple.com/devworld/Tool\protect\T1\textunderscoreChest/Core\protect\T1\textunderscoreprotect\T1\textunderscoreOS\protect\T1\textunderscoreTools/MPW\protect\T1\textunderscoreetc./. Pour les machines sous *Mac OS X*, le compilateur est *GNU CC*

Unix

Pour *Unix-like*, le *GNU C Compiler* (*alias GNU CC* ou *gcc*) est fourni avec toute distribution **BSD* ou *GNU/Linux*.

Wintel

Pour machines *Wintel*, il existe un portage du célèbre *GNU C Compiler* : http://www.delorie.com/djgpp/.

Un autre portage existe pour *Windows* : http://sources.redhat.com/cygwin/.

MinGW (Minimalist-GnuWin32), encore un portage *Win32* de *gcc*. Frustre mais puissant, optimise bien, adapté si l'on connaît bien l'environnement de programmation *UNIX* : http://mingw.sourceforge.net/

Borland a aussi mis en téléchargement gratuit certains de ses compilateurs là : http://www.borland.com/bcppbuilder/freecompiler/ ou là : http://www.borland.fr/download/compilateurs/.

Il y a aussi *lcc-win32*, qui est un dérivé du *lcc* originel de Chris Fraser et David Hanson. *lcc-win32* vient avec un éditeur intégré, et contient ce qu'il faut comme documentation et bibliothèques pour ouvrir des fenêtres sous *Windows*. Il est plus léger en termes de consommation mémoire et de CPU que *GNU CC* ; il produit un code décent (mais néanmoins pas optimal) : http://www.cs.virginia.edu/~lcc-win32/

4.3 Débogueurs

Apple

Allez là : http://devworld.apple.com/tools/debuggers/. Pour *Mac OS X*, le débogueur fourni avec le système est *GNU DB*.

Unix

GNU DB (*alias gdb*) est fourni avec les distributions **BSD* et *GNU/Linux*.

Wintel

À part celui de l'environnement *Turbo* (*cf.* 4.1, page 21), regardez du côté de http://sources.redhat.com/insight/.

4.4 Graphisme

Apple

Pour *Mac OS classique*, il vous faut récupérer *QuickDraw*, fournie par *Apple* : http://devworld.apple.com/macos/quickdraw.html Pour *Mac OS X*, l'environnement graphique s'appelle *Quartz* et il utilise *OpenGL* : http://developer.apple.com/techpubs/macosx/CoreTechnologies/graphics/Quartz2D/quartz2d.html

Unix

Pour environnement *Unix*, il existe un environnement graphique libre, reproduisant le système *X11* (dit aussi *X-Window*). Il est fourni avec toute distribution **BSD* ou *GNU/Linux*. Allez le voir sur http://www.xfree86.org/.

Wintel

Pour du graphisme sous *DOS* avec *djgpp*, il y a *Allegro* qui se trouve là : http://www.talula.demon.co.uk/allegro

4.5 Bibliothèques

Calcul scientifique

Pour du calcul scientifique, ici : http://www.netlib.org/ (mis à jour très régulièrement).

Ajay Shah maintenait un index de codes libres et/ou du domaine public implémentant diverses routines de calcul. On le trouve par exemple là : ftp://ftp.math.psu.edu/pub/FAQ/numcomp-free-c mais ça date.

Numerical Recipes in C

Là : http://www.ulib.org/webRoot/Books/Numerical\protect\T1\textunderscoreRecipes/bookcpdf.html. Les sources ont été incluses dans la distribution *Linux* pour les plate-formes à base de *PowerPC* : « *LinuxPPC 1999* ». Les sources ont disparu des miroirs et « *LinuxPPC 2000* » ne l'intègre plus mais les miroirs ont encore (en septembre 2000) le *package* de binaires (pour PPC uniquement) et certains sont prêts à envoyer les sources.

Générateurs de nombres pseudo-aléatoires

(Les anglo-saxons disent *Pseudo-Random Numbers Generators* et abrègent en PRNG)

Par exemple là : http://www.io.com/~ritter/NETLINKS.HTM#RandomnessLinks, http://burtleburtle.net/bob/rand/isaacafa.html ou http://random.mat.sbg.ac.at/. Voir aussi http://www.netlib.org/ et cherchez enfin *r250*, *RANLIB* et *FSULTRA*.

Calcul multi-précision

On trouve le code des fonctions `quad` de *BSD* quelque part là : ftp://ftp.uu.net/systems/unix/bsd-sources/ dans un répertoire `src/lib/libc/quad/`.

Il y a aussi la bibliothèque *GNU MP* (GNU Multiple Precision ou GMP) que l'on trouve ici : http://www.swox.com/gmp/.

Enfin, il y a le *package MIRACL* (voir http://indigo.ie/~mscott/).

Les sources de la bibliothèque standard

Le livre Plauger P.J. (1994), *La bibliothèque C standard* chez Masson, réimplémente la plupart des fonctions. Attention, les sources ne sont pas libres. On peut aussi consulter les sources des *Unix-based* libres **BSD* ou encore celles du projet *GNU*.

(informations bienvenues à mailto:pascal.cabaud@wanadoo.fr).

4.6 Outils divers

Notez que la majorité des outils cités ici ont initialement été écrits pour des plateformes *Unix*. Il existe assez souvent des portages sur d'autres *OS*, que l'on cite parfois. Il arrive fréquemment que je ne donne pas d'URL ; une recherche par exemple avec le merveilleux http://www.google.com/ vous donnera des liens à ne plus savoir qu'en faire ...

Certains sont là : ftp://gatekeeper.dec.com/pub/usenet/comp.sources.unix/ ou sur le mirroir : ftp://ftp.uu.net/usenet/comp.sources.unix/.

Pour les outils *GNU*, voir sur le site du projet : http://www.gnu.org/.

Indentation et mise en forme

Cherchez *cb*, *enscript*, *indent*, *lgrind*, *vgrind*.

Vérification de sources

Cherchez *lint* et regardez http://www.elirion.com/lintpage.html. On trouve *splint*(anciennement *LCLint*) là : ftp://www.splint.org.

Génération de références croisées

Cherchez *cflow*, *cxref*, *calls*, *cscope*, *xscope*, ou *ixfw*. C'est pratique pour voir les graphes de dépendances des fonctions.

Dégénération de code C

Cherchez *obfus*, *shroud*, ou *opqcp*. Ce gadget sert à rendre un source parfaitement illisible.

Interprétation des déclarations

Cherchez *cdecl* dans le `volume 14`, à http://www.uu.net/usenet/comp.sources.unix/. Ça sert essentiellement à comprendre des déclarations ou à en générer.

Génération de dépendances

Cherchez *makedepend* ou tentez *cc -M* ou *cpp -M*.

Gestion de projets

Cherchez *GNU make*. Voir aussi 3.10, page 19.

Contrôle de versions

Cherchez *CVS*, *RCS* et *SCCS*. Voir aussi 3.10, page 19.

Traducteurs Fortran/C et Pascal/C

Cherchez *ftoc* et *f2c* pour le Fortran. On trouve *f2c* sur http://www.netlib.org/. Il existe une version pour *MacOS* : http://www.alumni.caltech.edu/~igormt/Mac\protect\T1\textunderscoreF2C.html. Pour le Pascal, ça se trouve là : ftp://csvax.cs.caltech.edu/pub/p2c-1.20.tar.Z. Cherchez aussi *ptoc*.

Calcul d'unités

Cherchez *ccount*, *Metre*, *lcount*, *csize*, http://www.qucis.queensu.ca/.

Calcul du nombre de lignes d'un fichier source

Utilisez les commandes *wc* ou `grep -c ";"`.

Génération de prototypes

Cherchez *cproto* (http://www.vex.net/~cthuang/cproto/) et *cextract*.

Gestion correcte des appels à malloc

Cherchez *dbmalloc*, *MallocDebug*, `JMalloc.c` et `JMalloc.h`, zieutez la page http://www.cs.colorado.edu/homes/zorn/public\protect\T1\textunderscorehtml/MallocDebug.html et regardez : ftp://ftp.crpht.lu/pub/sources/memdebug. Ceci fait, il vous reste à compiler et installer *dmalloc* provenant de là : http://dmalloc.com/.

Génération de documentation à partir de code commenté

Il existe divers outils dont *doc++* et *doxygen*. Le dernier gère le format LaTeX2e, HTML et *nroff* et se trouve ici : http://www.stack.nl/~dimitri/doxygen/.

Voir aussi 4.7, page 25.

Préprocesseur sélectif

Cherchez *unifdef*, *rmifdef* et *scpp*. Ils suppriment les directives `#ifdef` inutiles, rendant ainsi un code plus lisible.

Gestion des expressions régulières

Le projet *GNU* propose *rx*. Il y a aussi *regexp* sur ftp://ftp.cs.toronto.edu/pub/regexp.shar.Z.

Profileur

Le projet *GNU* développe *gprof* (*cf.* 16.1, page 85). Il existe aussi *FunctionCheck* : http://www710.univ-lyon1.fr/~yperret/fnccheck/profiler.html

4.7 Où trouver du code ?

Là : http://www.gnu.org par exemple. Les logiciels *GNU*, les *Unix-like* *BSD* et *GNU/Linux* sont distribués (ils sont *libres*) avec les sources. La pluparts d'entre eux étant écrits en C, cela en fait une très grande collection de code.

On trouvera du code (du domaine public) là : ftp://ftp.cdrom.com. Il y a aussi ftp://ftp.brokersys.com/pub/snippets et sur le *web* à http://www.brokersys.com/snippets/ ou http://www.ping.de/sites/systemcodes/ ou encore par *ftp* par exemple à ftp:

//ftp.funet.fi/pub/languages/C/Publib/.

Les serveurs ftp://ftp.uu.net/, ftp://archive.umich.edu/, ftp://oak.oakland.edu/, ftp://sumex-aim.stanford.edu/, et ftp://wuarchive.wustl.edu/, hébergent une grande quantité de logiciels, code et documentation libres et/ou gratuits. le serveur princicpal du projet *GNU* est ftp://prep.ai.mit.edu/.

Voir aussi 4.6, page 24 et 4.5, page 23.

(informations bienvenues à mailto:pascal.cabaud@wanadoo.fr)

Chapitre 5

Déclarations et initialisations

5.1 Quels types utiliser ?

Il existe en C plusieurs types de nombres entiers. Si vous avez à gérer de grands nombres, il faut utiliser le type `long`. Avec la norme C99 (*cf.* 3.7, page 15), un type `long long` est disponible. Pour des nombres de petites tailles, et si la place mémoire manque, c'est le type `short` qu'il faut prendre.

Le type `char` peut parfois être utilisé comme très petit entier. Mais cela doit être évité au maximum. En effet, outre les problèmes de signe, le code généré peut être plus complexe et risque finalement de prendre plus de place et de faire perdre du temps. Même les `short` font parfois perdre du temps.

Dans les autres cas, `int` est bien adapté.

Le mot clé `unsigned` est à utiliser pour les nombres positifs, si vous avez des problèmes dus aux débordements ou pour les traitements par bits.

Le choix entre `float` et `double` ne se pose pas. On devrait toujours utiliser `double`, sauf si on a vraiment des contraintes de mémoire (*cf.* 11.1, page 54 et 11.10, page 56).

5.2 Comment définir une structure qui pointe sur elle-même ?

Il y a plusieurs façons correctes de le faire. Le problème est que dans la définition de la structure avec un `typedef`, le type n'est pas encore défini. Voici la manière la plus simple, où pour contourner le problème, on ajoute un *tag* à la structure.

```
typedef struct node {
    char * item;
    struct node * next;
} node_t;
```

Une autre solution consiste à déclarer le type de la structure avant sa définition, avec un pointeur.

```
typedef struct node * node_p;
typedef struct node {
    char * item;
    node_p next;
} node_t;
```

Cette construction récursive est utilisée pour obtenir des listes chaînées ou des arborescences.

5.3 Comment déclarer une variable globale ?

Sauf dans des cas précis, vous devriez éviter d'utiliser des variables globales.

Si vous y tenez vraiment, la meilleure solution est de déclarer la variable dans un fichier `xxx.c`, et de mettre la déclaration `extern` dans un `xxx.h` associé. Ceci évitera des redéfinitions de la variable lors des inclusions d'en-têtes.

Le mot clé `static` permet d'avoir des variables locales persistantes, ce qui peut être une bonne alternative aux globales.

5.4 Quelle est la différence entre `const` et `#define` ?

Cela n'a rien à voir. Le `#define` permet de définir une macro, alors que le mot clé `const` indique que l'objet qualifié est protégé en écriture.

Lors de la compilation, la première phase est effectuée par le pré-processeur qui remplace les macros par leurs valeurs. C'est un simple copier/coller. Une variable déclarée `const` reste quant à elle une variable, mais on ne peut lui affecter une valeur que lors de l'initialisation. Après, elle est n'est plus modifiable.

Le mot-clé `const` permet aussi au compilateur d'effectuer des optimisations en plaçant par exemple la variable dans un registre.

Il est à noter qu'avec la nouvelle norme C99, il est possible de déclarer un tableau dont la taille est donnée par une variable `const`. Auparavant, il fallait utiliser une macro.

Voir aussi les questions 13.7, page 65 et 5.5, page 28.

5.5 Comment utiliser `const` avec des pointeurs ?

Le mot-clé `const` permet de protéger une variable de modifications ultérieures. Une variable constante n'est modifiable qu'une fois, lors de l'initialisation.

Un pointeur étant une variable comme les autres, `const` s'utilise de la même façon. Voici un exemple :

```
const char * sz1;
```

```
char const * sz2;
char * const sz3;
char const * const sz4;
```

Les variables `sz1` et `sz2` sont des pointeurs sur objet constant de type `char`. La variable `sz3` est un pointeur constant sur un objet (non constant) de type `char`. Enfin, `sz4` est un pointeur constant sur un objet constant de type `char`.

Un petit « amusement » pour terminer, que signifie la déclaration `const char * (* f)(char * const * s) ;` ?

Voir aussi la question 5.4, page 28.

5.6 Comment bien initialiser ses variables ?

Les variables globales, ou déclarées `static`, sont initialisées automatiquement lors de leur définition. Si aucune valeur n'est spécifiée, c'est un zéro qui est pris (suivant le type de la variable, `0`, `0.0` ou `NULL`).

Ce n'est pas le cas pour les variables automatiques (les autres). Il est donc nécessaire de le faire « à la main ».

Les variables allouées dynamiquement avec `malloc` ne le sont pas non plus. On pourra utiliser `calloc` qui initialise les variables allouées. Il est à noter que `calloc` met les bits à `0` comme le ferait `memset`. Cette initialisation est valide pour les types entiers (`char`, `short`, `int` et `long`), mais non portable pour les pointeurs et les flottants.

Une bonne méthode qui initialise correctement les variables suivant leur type est celle-ci :

```
{
    type a[10]={0};
    struct s x ={0};
}
```

Avec la variante dynamique :

```
[static] const struct s x0 ={0};
{
    struct s *px =malloc(sizeof *px);
    *px=x0;
}
```

5.7 Comment déclarer un tableau de fonctions ?

Deux cas se présentent. Si toutes les fonctions ont le même prototype, il suffit de faire ainsi :

```
extern char * f(int, int); /* une fonction                    */
char * (*fp[N])(int, int); /* Un tableau de N fonctions       */
char * sz;
fp[4] = f;                 /* affectation de f dans le tableau */
sz = fp[4](42, 12);        /* utilisation                      */
```

Si les fonctions ont des prototypes différents, il faut déclarer un tableau de fonctions génériques. Les fonctions génériques n'existent pas à proprement parler. Il faut utiliser une fonction sans argument spécifié et retournant un `int`.

```
int (*fp[N])(); /* Un tableau de N fonctions quelconques*/
```

Cela « capte » la plupart des cas, sauf les fonctions au nombre d'arguments variables, comme `printf`.

Voir aussi la question 7.4, page 39.

5.8 Comment connaître le nombre d'éléments d'un tableau ?

On peut utiliser une macro de ce type là :

```
#define NELEMS(n) (sizeof(n) / sizeof *(n))
```

5.9 Quelle est la différence entre `char a[]` et `char * a` ?

Il faut bien se rappeler qu'en C, un tableau n'est pas un pointeur, même si à l'usage ça y ressemble beaucoup.

`char a[]` déclare un tableau de `char` de taille inconnue (type incomplet). Dès l'initialisation, par

```
char a[] = "Hello";
```

`a` se transforme en `char[6]`, soit un tableau de six `char` (type complet).

Il arrive souvent de voir la confusion entre « tableau » et « pointeur constant » (moi même je la fais parfois ;-))

Ce « pointeur constant » est inspiré par K&R, 1ère édition. C'était une métaphore malheureuse de K&R qui voulait exprimer qu'un tableau se *comporte* en général comme une « valeur (rvalue) du type pointeur », et bien entendu une valeur est toujours constante. Par cette métaphore ils ont essayé d'expliquer pourquoi on ne pouvait pas prendre l'adresse d'un tableau.

Malheureusement ceci n'explique pas pourquoi `sizeof a` donne la taille du tableau et non la taille d'un pointeur.

Sur ce sujet, la norme est plus précise en disant qu'un tableau dans une *expression* — sauf dans `&a` ou `sizeof a` ou dans des initialisations par des chaînes littérales "xyz" — est *converti* automatiquement en une valeur du type pointeur qui pointe sur l'élément initial du tableau (merci à Horst KRAEMER pour ces précisions).

Voir aussi la question 7.1, page 37.

5.10 Peut-on déclarer un type sans spécifier sa structure ?

Plus précisément, est-il possible de définir un type de données dont on veut cacher à l'utilisateur de la bibliothèque l'implémentation ?

Oui c'est possible, en encapsulant ce type dans une structure. Il y a au moins deux méthodes. La première est de définir une structure (publique) qui contient un pointeur `void *`, qui pointe vers une variable du type privé. Ce type privé est défini avec les fonctions, dans un `.c`. Il faut alors prévoir des fonctions d'initialisation et de destruction des objets.

Une autre solution consiste à déclarer une structure qui contient une donnée du type à protéger, et à accéder aux types par des pointeurs uniquement. Voici un exemple :

```
/* data.h */
struct Data_s ;
typedef struct Data_s Data_t ;

extern Data_t * DataNew(int x, int y);
extern int DataFonction(Data_t * this);

/* data.c */
#include <stdio.h>
#include <stdlib.h>
#include "data.h"

struct Data_s {
    int x;
    int y;
};

Data_t * DataNew(int x , int y) {
    Data_t * pData;

    pData = malloc(sizeof *pData) ;
    if (pData) {
        pData->x = x;
        pData->y = y;
    }
```

```
    return pData;
}

int DataFonction(Data_t * this) {
    if (!this)
        return 0;
    return (this->x * this->x) + (this->y * this->y);
}

/* main.c */
#include <stdio.h>
#include "data.h"

int main(void) {
    Data_t *psData;

    psData = DataNew(3, 4);
    printf("%d\n", DataFonction(psData));

    return 0;
}
```

(Merci à Yves ROMAN pour cet exemple.)

Chapitre 6

Structures, unions, énumérations

6.1 Quelle est la différence entre `struct` et `typedef struct` ?

```
struct x1 { ... };
typedef struct { ... } x2;
```

La première écriture déclare un *tag* de structure `x1`. La deuxième déclare un nouveau type nommé `x2`.

La principale différence est l'utilisation. La deuxième écriture permet un peu plus l'abstraction de type. Cela permet de cacher le véritable type derrière `x2`, l'utilisateur n'étant pas sensé savoir que c'est une structure.

```
struct x1 v1;
x2 v2;
```

6.2 Une structure peut-elle contenir un pointeur sur elle-même ?

Oui.

Voir aussi la question 5.2, page 27.

6.3 Comment implémenter des types cachés (abstraits) en C ?

L'une des bonnes façons est d'utiliser des pointeurs sur des structures qui ne sont pas publiquement définies. On peut en plus cacher les pointeurs par des `typedef`.

Voir aussi la question 5.10, page 31.

6.4 Peut-on passer des structures en paramètre de fonctions ?

Oui, c'est parfaitement autorisé. Toutefois, rappelons que les paramètres en C sont passés par valeurs et copiés dans la pile. Pour des grosses structures, il est préférable de passer un pointeur dessus, et éventuellement un pointeur constant.

6.5 Comment comparer deux structures ?

Il n'existe pas en C d'opérateur ou de fonction pour comparer deux structures. Il faut donc le faire à la main, champs par champs.

Une comparaison bit à bit n'est pas portable, et risque de ne pas marcher, en raison du *padding* (alignement sur certains octets).

6.6 Comment lire/écrire des structures dans des fichiers ?

Il faut utiliser les fonctions `fread` et `fwrite`. Attention : les fichiers obtenus ne sont pas portables.

Une méthode plus portable consiste à enregistrer les structures dans un fichier texte.

6.7 Peut-on initialiser une union ?

La norme prévoit d'initialiser le premier membre d'une union. Pour le reste, ce n'est pas standard. En C99, on peut initialiser les champs d'une union :

```
union { /* ... */ } u = { .any_member = 42 };
```

6.8 Quelle est la différence entre une énumération et des `#define` ?

Il y a peu de différences.

L'un des avantages de l'énumération est que les valeurs numériques sont assignées automatiquement. De plus, une énumération se manipule comme un type de données. Certains programmeurs reprochent aux énumérations de réduire le contrôle qu'ils ont sur la taille des variables de type énumération.

6.9 Comment récupérer le nombre d'éléments d'une énumération ?

Ceci n'est possible de façon automatique que si les valeurs se suivent.

```
typedef enum { A, B, C, D} type_e;
```

Dans cette énumération, les valeurs sont données par le compilateur dans l'ordre croissant, à partir de 0 et avec un pas de 1. Ainsi, le nombre d'éléments de `type_e` est `D + 1`.

On peut rajouter un élément à l'énumération qui donne directement le nombre d'éléments :

```
typedef enum { RED, BLUE, GREEN, YELLOW, NB_COLOR} color_e;
```

le nombre d'éléments de `color_e` est donc `NB_COLOR`.

Si l'on est obligé, pour des raisons diverses et variées, de fixer d'autres valeurs aux constantes, alors cette solution ne marche pas. On peut toujours rajouter un champs dans l'énumération et fixer manuellement sa valeur.

6.10 Comment imprimer les valeurs symboliques d'une énumération ?

On ne peut pas le faire simplement. Il faut écrire une fonction qui le fait. Un problème qui se pose alors est la maintenance, car une modification des valeurs de l'énumération entraîne la nécessité d'une mise à jour de cette fonction.

Voici un code qui limite les problèmes de mise à jour :

```
/* Fichier foo.itm */
ITEM(FOO_A)
ITEM(FOO_B)
ITEM(FOO_C)
/**/

/* Fichier foo.h */
#ifndef FOO_H
#define FOO_H
#define ITEM(a) a,
typedef enum {
#include "foo.itm"
    FOO_NB
} foo_t;
#undef ITEM

#define ITEM(a) #a,
const char * const aFoo[] = {
#include "foo.itm"
};
```

```
#undef ITEM
#endif
/**/

/* Fichier foo.c */
#include <stdio.h>
#include "foo.h"
int main(void) {
    foo_t foo;
    for (foo = FOO_A; foo < FOO_NB; foo++) {
        printf("foo=%d ('%s')\n",foo, aFoo[foo]);
    }

    return 0;
}
```

Merci à Emmanuel Delahaye pour cet exemple.

Chapitre 7

Tableaux et pointeurs

7.1 Quelle est la différence entre un tableau et un pointeur ?

Un tableau n'est pas un pointeur. Un tableau est une zone mémoire pouvant contenir `N` éléments consécutifs de même type. Un pointeur est une zone mémoire qui contient l'adresse d'une autre zone mémoire. Toutefois, dans un grand nombre de cas, tout se passe comme si c'était la même chose.

À ce titre, il faut bien faire la différence entre `a[i]` pour un tableau et `ap[i]` pour un pointeur. Voici un exemple :

```
char a[] = "Bonjour";
char *ap = "Au revoir";
```

L'expression `a[3]` signifie que l'on accède aux quatrième élément du tableau. `ap[3]` signifie que l'on accède à la zone mémoire pointée par `(ap+3)`. Autrement dit, `a[3]` est l'objet situé 3 places après `a[0]` (`a` est le tableau *entier*), alors que `ap[3]` est l'objet situé 3 places après l'objet pointé par `ap`. Dans l'exemple, `a[3]` vaut `'j'` et `ap[3]` vaut `'r'`.

Voir aussi la question 5.9, page 30.

7.2 Comment passer un tableau à plusieurs dimensions en paramètre d'une fonction ?

Ce n'est pas si facile. La règle de base est qu'il faut connaître la taille des `N-1` dernières dimensions. Pour un tableau à deux dimensions, la deuxième doit être connue, et la fonction doit être déclarée ainsi :

```
int f1(int a[][NCOLUMNS]);
            /* a est un tableau a deux dimensions (cf. remarque) */
int f2(int (*ap)[NCOLUMNS]);
            /* ap est un pointeur sur un tableau  */
```

Si elle n'est pas connue, il faut passer la taille du tableau en paramètre (ligne ET colonne) et un pointeur sur le tableau :

```
int f(int * a, int nrows, int ncolumns);
```

On accède aux éléments du tableau ainsi :

```
a[i * ncolumns + j] /* element de la ieme ligne
                     * et de la jeme colonne */
```

Une remarque : Dans une déclaration de paramètre

```
int f1(int a[][NCOLUMNS])
```

ou

```
int f1(int a[42][NCOLUMNS])
```

`a` est un pointeur sur `int[NCOLUMS]` malgré l'écriture. La déclaration est interprétée comme

```
int (*a) [NCOLUMS]
```

Ainsi les déclarations de `f1` et `f2` dans l'exemple initial sont exactement les mêmes.

7.3 Comment allouer un tableau à plusieurs dimensions ?

La première solution est d'allouer un tableau de pointeurs, puis d'initialiser chacun de ces pointeurs par un tableau dynamique.

```
#include <stdlib.h>

int ** a = malloc(nrows * sizeof *a );
for(i = 0; i < nrows; i++)
   a[i] = malloc(ncolumns * sizeof *(a[i]));
```

Dans la vraie vie, le retour de `malloc` doit être vérifié.

Une autre solution est de simuler un tableau multi-dimensions avec une seule allocation :

```
int *a = malloc(nrows * ncolumns * sizeof *a);
```

L'accès aux éléments se fait par :

```
a[i * ncolumns + j] /* element de la ieme ligne
                     * et de la jeme colonne */
```

7.4 Comment définir un type pointeur de fonction ?

On utilise `typedef`, comme pour n'importe quel autre type. Voici un exemple :

```
int f(char * sz); /* une fonction                 */
int (*pf)(char *);/* un pointeur sur une fonction */
typedef int (*pf_t)(char *);
                  /* un type pointeur sur fonction*/
```

Il est toutefois préférable de ne pas cacher le pointeur dans un `typedef`. La solution suivante est plus jolie :

```
typedef int (f_t)(char *); /* un type fonction        */
f_t * pf;                  /* un pointeur sur ce type */
```

On l'utilise alors de cette façon :

```
pf = f;
int ret = pf("Merci pour cette reponse");
```

Voir aussi la question 5.7, page 29.

7.5 Que vaut (et signifie) la macro `NULL` ?

`NULL` est une macro qui représente une valeur spéciale pour désigner un pointeur nul lorsque converti au type approprié. Elle est définie dans `<stddef.h>` ou dans `<stdio.h>`.

La valeur réelle de `NULL` est dépendante de l'implémentation, et n'est pas nécessairement un pointeur, ni de type pointeur. Des valeurs possibles sont `((void *)0)` ou `0`.

`NULL` permet de distinguer les pointeurs valides des pointeurs invalides. Par exemple, `malloc` renvoie une valeur comparable à `NULL` quand elle échoue.

Voir aussi la question 12.3, page 59.

7.6 Que signifie l'erreur « `NULL`-*pointer assignment* » ?

Cela signifie que vous avez essayé d'accéder à l'adresse 0 de la mémoire. Vous avez probablement déréférencé un pointeur `NULL`, ou oublié de tester la valeur retour d'une fonction, avant de l'utiliser.

7.7 Comment imprimer un pointeur ?

La seule manière prévue par la norme pour imprimer correctement un pointeur est d'utiliser la fonction `printf` avec le code de format `%p`. Le pointeur doit être d'abord *casté* en un pointeur générique `void *`.

```
char * p;
printf("Pointeur p avant initialisation: %p\n",
                                          (void *)p);
```

7.8 Quelle est la différence entre `void *` et `char *` ?

Le premier est un pointeur générique, qui peut recevoir l'adresse de n'importe quel type d'objet. Le second est un pointeur sur un caractère, généralement utilisé pour les chaînes.

Avant la norme ANSI, le type `void` n'existait pas. C'était donc `char *` qui était utilisé pour faire des pointeurs génériques. Depuis la norme, ce n'est plus valide. De nombreux programmeurs ont toutefois gardé cette habitude, notamment dans le *cast* de fonctions comme `malloc` (*cf.* 12.1, page 58).

Chapitre 8

Chaînes de caractères

8.1 Comment comparer deux chaînes ?

Pour comparer deux chaînes entre elles, il faut utiliser la fonction `strcmp`, et non l'opérateur `==`. Celui-ci comparera les pointeurs entre eux, ce qui n'est probablement pas ce qui est voulu !

```
const char * sz = "non";
if(strcmp(sz, "oui") == 0) {
    /* sz et "oui" sont egaux */
}
```

Il existe aussi la fonction `strncmp` qui permet de contrôler la longueur de comparaison.

8.2 Comment recopier une chaîne dans une autre ?

Il faut utiliser la fonction `strcpy`, et non l'opérateur d'affectation `=`. Il faut s'assurer que l'on dispose d'espace suffisant dans la chaîne cible avant d'utiliser `strcpy`, qui ne fait aucun contrôle de débordement.

Pour une copie plus sécurisée, on préférera la fonction `strncpy`.

8.3 Comment lire une chaîne au clavier ?

Il y a de nombreuses fonctions qui le font. Le plus sûr est d'utiliser `fgets`.

```
char tab[20];
fgets(tab, sizeof tab, stdin);
```

Voici un exemple complet d'utilisation propre de `fgets` pour la lecture d'une ligne, avec un contrôle d'erreurs. Cette fonction est fournie par Emmanuel Delahaye.

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

int get_line(char *buf, size_t size) {
   int ret; /* 0=Ok 1=Err 2=Incomplet
             * On peut aussi definir des constantes. (Macros, enum...)
             */

   if (fgets(buf, size, stdin) != NULL) {
      char *p = strchr(buf, '\n'); /* search ... */
      if (p != NULL) {
         *p = 0; /* ... and kill */
         ret = 0;
      }
      else {
         ret = 2;
      }
   }
   else {
      ret = 1;
   }
   return ret;
}
```

Il peut être intéressant dans certains cas de faire un traitement particulier dans le cas 2 (lecture incomplète), comme vider le buffer du flux ou redimentionner la zone de réception (voir la question 14.5, page 76).

La fonction `gets` est à proscrire, car il n'y a aucun contrôle de débordement, ce qui peut engendrer de nombreux bugs (*stack overflow*) (*cf.* 8.7, page 43).

8.4 Comment obtenir la valeur numérique d'un `char` (et vice-versa) ?

En C, un `char` est un petit entier. Il n'y a donc aucune conversion à faire. Quand on a un `char`, on a aussi sa valeur, et vice-versa.

8.5 Que vaut `sizeof(char)` ?

Un `char` vaut et vaudra toujours 1 indépendamment de l'implémentation. En effet, les tailles d'allocation en C se calculent en `char` (`size_t`). Or, un `char` a une taille de 1 `char`. Donc

```
sizeof(char) == (size_t) 1
```

Pour autant, un `char` ne fait pas forcément 8 bits (un octet) (voir aussi 16.2, page 85).

8.6 Pourquoi `sizeof('a')` ne vaut pas 1 ?

En C, les caractères constants sont des `int`, et non des `char`. Ainsi,

```
sizeof('a') == sizeof(int)
```

ce qui peut valoir 2 ou 4 sur votre machine, ou autre chose.

8.7 Pourquoi ne doit-on jamais utiliser `gets` ?

Serge PACCALIN donne l'exemple suivant :

```
#include <stdio.h>

int main(void)
{
   char chaine[16];

   printf("Tapez votre nom :\n");
   gets(chaine);
   printf("Vous vous appelez %s.\n",chaine);
   return 0;
}
```

Quand on tape une chaîne de plus de 15 caractères, rien ne va plus : `gets` accepte sans broncher la chaîne mais lorsqu'il est question de la stocker dans

```
char chaine[16]
```

on peut obtenir un magnifique *Segmentation fault (core dumped)*. Dans certains cas, avec certains compilateurs sur certaines machines et certains OS, il est possible que ça passe. Mais attention ! C'est un leurre, et le changement de cible démontrera la malfaçon.

8.8 Pourquoi ne doit-on *presque* jamais utiliser `scanf` ?

`scanf` est une fonction de la bibliothèque standard, qui est souvent la première que l'on apprend pour lire des données au clavier. Cette fonction n'est pas plus dangereuse qu'une autre, à condition de bien savoir l'utiliser, ce qui n'est pas donné à tout le monde.

Par exemple, regardez le programme suivant donné par Serge PACCALIN :

```
#include <stdio.h>

int main(void) {
   int val = 0;

   while (val != 1){
      printf("Tapez un nombre"
             "(1 pour arreter le programme) :\n");
      scanf(" %d",&val);
      printf("Vous avez saisi %d.\n",val);
   }
   return 0;
}
```

Et il explique : « Quand le programme demande un nombre, taper `"toto"` suivi de la touche Entrée. Le programme part en boucle parce que tous les `scanf` successifs butent sur `"toto"` qui reste indéfiniment dans `stdin`. »

Dans la plupart des cas, l'utilisation de la fonction `fgets` sera plus simple et moins risquée.

`scanf` est une fonction qui peut être utilisée dans certaines conditions, et à condition de bien savoir ce que l'on fait. L'utilisation de `scanf` pour la lecture de nombres (entiers ou flottants) est l'une des plus acceptable. Par contre, la lecture d'une chaîne avec `scanf` sans contrôle de format est aussi dangereux que `gets`.

```
scanf("%s", astring) ;
```

est donc a proscrire.

`scanf` n'est préférable à `fgets` que dans le cas ou l'on veut lire mot par mot et non ligne par ligne, *et* conserver le reste dans le buffer du flux d'entrée. Sinon, lire un mot avec `fgets` ne pose pas de problème et est même plus simple.

```
scanf("%4s", astring) ;
```

Cette construction, par exemple, ne pose pas les problèmes cités plus haut (si le contrôle d'erreurs est fait), et est parfois utile.

Voir aussi les questions 8.7, page 43 et 8.3, page 41

Chapitre 9

Fonctions et prototypes

9.1 Pour commencer ...

Il y a trois notions :
– déclaration,
– définition,
– prototype.
La déclaration d'une fonction, c'est annoncer que tel identificateur correspond à une fonction, qui renvoie tel type. La définition d'une fonction est une déclaration où, en plus, on donne le code de la fonction elle-même. Le prototype est une déclaration de fonction où le type des arguments est également donné.

Par exemple :

```
int f();    /* declaration de f(), renvoyant un int, pas de prototype  */

int f(void);/* declaration de f(), renvoyant un int, prototype (0 arg) */

int f(void) /* definition de f() avec declaration avec prototype       */
{
    return 42;
}

int f(x)    /* definition de f() avec declaration sans prototype       */
int x;
{
    return x;
}

int f()     /* definition de f() avec declaration sans prototype       */
{
    return 42;
}
```

Ce qui n'est pas possible :
– avoir une définition sans déclaration,
– avoir un prototype sans déclaration.
Ce qui est autorisé :
– appeler une fonction déclarée, qu'elle ait un prototype ou pas.
Ce qui était autorisé en C90 mais ne l'est plus en C99 :
– appeler une fonction non déclarée ; l'appel valait déclaration dite « implicite », sans prototype et avec type de retour `int`.
Ce qui est encore autorisé en C99 mais disparaîtra bientôt :
– déclarer/définir une fonction sans prototype.
Ce qu'il faut faire quand on veut programmer lisiblement, en détectant les bugs et en gardant du code maintenable et compatible avec le futur :
– utiliser des déclarations et définitions avec prototype.

9.2 Qu'est-ce qu'un prototype ?

Un prototype est une signature de fonction. Comme tout objet en C, une fonction doit être déclarée avant son utilisation. Cette déclaration est le prototype de la fonction. Le prototype doit indiquer au compilateur le nom de la fonction, le type de la valeur de retour et le type des paramètres (sauf pour les fonctions à arguments variables, comme `printf`. (*cf.* 9.6, page 48).

```
int fa(int a, char const * const b);
int fb(int, char const * const);
```

Les noms de paramètre sont optionnels, mais il est fortement conseillé de les laisser. Cela donne une bonne indication sur leurs rôles.

Les fonctions de la bibliothèque ont également leur prototype. Avant l'utilisation de celles-ci, il faut inclure les fichiers d'en-tête contenant les prototypes. Par exemple, le prototype de `malloc` se trouve dans `stdlib.h`.

Certains préfèrent ajouter le mot clé `extern` au prototype, afin de rester cohérent avec la déclaration des variables globales.

Voir aussi les questions 9.3, page 46, 12.1, page 58, 13.5, page 64 et 14.17, page 80.

9.3 Où déclarer les prototypes ?

Un prototype de fonction doit être déclaré avant l'utilisation de la fonction. Pour une plus grande lisibilité, mais aussi pour simplifier la maintenance du code, il est conseillé de regrouper tous les prototypes d'un module (fichier `xxx.c`) dans un en-tête (<`xxx.h`>). Ce dernier n'a plus alors qu'à être inclus dans le code qui utilise ces fonctions. C'est le cas des fonctions de la bibliothèque standard.

Voir aussi les questions 13.5, page 64 et 9.10, page 50.

9.4 Quels sont les prototypes valides de `main` ?

La fonction `main` renvoie toujours un `int`. Les prototypes valides sont :

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

Tout autre prototype n'est pas du tout portable et ne doit jamais être utilisé (même s'il est accepté par votre compilateur).

En particulier, vous ne devez pas terminer la fontion `main` sans retourner une valeur positive (non nulle en cas d'erreur). Les valeurs de retour peuvent être `0`, `EXIT_SUCCESS` ou `EXIT_FAILURE`.

On pourra aussi rencontrer (sous *Unix*) le prototype suivant :

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

dans le but d'utiliser les variables d'environnement du *shell* actif. Ce n'est ni portable ni standard, d'autant plus que les fonctions `getenv`, `setenv` et `putenv` le sont et suffisent largement.

Enfin, rappelons que le prototype suivant

```
int main () ;
```

est parfaitement valide en C++ (et est synonyme du premier présenté ici), mais ne l'est pas en C.

9.5 Comment `printf` peut recevoir différents types d'arguments ?

`printf` est une fonction à nombre variable de paramètres. Son prototype est le suivant :

```
int printf(const char * format, ...); /* C 90 */
int printf(const char * restrict format, ...); /* C 99 */
```

Le type et le nombre des paramètres n'est pas défini dans le prototype, c'est le traitement effectué dans la fonction qui doit les vérifier.

Pour utiliser cette fonction, il est donc impératif d'inclure l'en-tête `<stdio.h>`.

Pour écrire une fonction de ce type, lire la question suivante (9.6, page 48).

9.6 Comment écrire une fonction à un nombre variable de paramètres ?

La bibliothèque standard fournit des outils pour faciliter la gestion de ce type de fonctions. On les trouve dans l'en-tête `<stdarg.h>`.

Le prototype d'une fonction à nombre variable de paramètres doit contenir au moins un paramètre explicite, puis se termine par `...` Exemple :

```
int f(int nombre, ...);
```

Il faut, d'une façon ou d'une autre, passer dans les paramètres le nombre d'arguments réellement transmis. On peut le faire en donnant ce nombre explicitement (comme `printf`), ou passer la valeur `NULL` en dernier.

Attention toutefois avec la valeur `NULL` dans ce cas. En effet, `NULL` n'est pas nécessairement une valeur du type pointeur mais une valeur qui *donne* un pointeur nul si elle est affectée ou passée ou comparée à un type pointeur. Le passage d'une valeur à un paramètre n'est pas une affectation à un pointeur mais une affectation qui obéit aux lois spéciales pour les paramètres à nombre variable (ou pour les paramètres d'une fonction sans prototype). Les lois de promotion pour les types arithmétiques sont appliquées). Si `NULL` est défini par

```
#define NULL 0
```

alors `(int)0` est passé à la fonction. Si un pointeur n'a pas la même taille qu'un `int` ou si un pointeur nul n'est pas représenté par « tous les bits 0 » le passage d'un 0 ne passe donc *pas* de pointeur nul. La méthode portable est donc

```
f(toto,titi,(void*)NULL);
```

ou

```
f(toto,titi,(void*)0);
```

C'est le seul cas où il faut caster NULL parce qu'il ne s'agit pas d'un contexte syntactique « de pointeur », seulement d'un contexte « de pointeur par contrat ».

Après cela, les fonctions `va_start`, `va_arg` et `va_end` permettent de parcourir la liste des paramètres.

Voici un petit exemple :

```
#include <stdarg.h>
```

```
int vexemple(int nombre, ...){
    va_list argp;
    int i;
    int total = 0;

    if(nombre < 1)
            return 0;

    va_start(argp, nombre);
    for (i = 0; i < nombre; i++) {
        total += va_arg(argp, int);
    }
    va_end(argp);

    return total;
}
```

Merci à Horst KRAEMER pour ces remarques.

9.7 Comment modifier la valeur des paramètres d'une fonction ?

En C, les paramètres sont passés par valeur. Dans la plupart des implémentations, cela se fait par une copie dans la pile. Lors du retour de la fonction, ces valeurs sont simplement dépilées, et les modifications éventuelles sont perdues. Pour pallier cela, il faut simuler un passage des paramètres par référence, en passant un pointeur sur les variables à modifier. Voici l'exemple classique de l'échange des valeurs entre deux entiers :

```
void echange(int * a, int * b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

9.8 Comment retourner plusieurs valeurs ?

Le langage C ne permet pas aux fonctions de renvoyer plusieurs objets. Une solution consiste à passer l'adresse des objets à modifier en paramètre. Une autre solution consiste à renvoyer une structure, ou un pointeur sur une structure qui contient l'ensemble des valeurs. Généralement, quand on a ce genre de choses à faire, c'est qu'il se cache une structure de données que l'on n'a pas identifiée. La pire des solutions est d'utiliser des variables globales.

9.9 Peut-on, en C, imbriquer des fonctions ?

Non, on ne peut pas. Les concepteurs ont jugé cela trop compliqué à mettre en oeuvre (portée des variables, gestion de la pile etc.). Certaines implémentations, comme *GNU CC* le supportent toutefois. Ceci dit, on peut très bien s'en passer, en utilisant des pointeurs sur les structures de données à partager, ou en utilisant des pointeurs de fonctions.

9.10 Qu'est-ce qu'un en-tête ?

.

Un en-tête est un ensemble de déclarations, définitions et prototypes nécessaires pour compiler et pour utiliser un module. Par exemple, pour utiliser les fonctions d'entrées/sorties de la bibliothèque standard, il est nécessaire d'inclure dans son programme l'en-tête `<stdio.h>`.

Par abus, on parle souvent de fichier d'en-tête, car historiquement, et encore aujourd'hui pour de nombreuses implémentations, ces en-têtes sont des fichiers. C'est également le cas pour les en-têtes personnels. Toutefois, la norme n'exige pas que les en-têtes standards soient des fichiers à proprement parlé.

Chapitre 10

Expressions

10.1 Le type Booléen existe-t-il en C ?

Oui, il existe un type booléen en C. C'est un ajout de la dernière version de la norme (C99). Il s'agit du type `_Bool` défini dans `<stdbool.h>`. Cet en-tête contient également les définitions de `true` et `false`. Une macro `bool` est souvent définie comme équivalent à `_Bool`.

Rappelons également qu'en C, une valeur est « fausse » si elle est nulle (ou équivalent), et « vraie » sinon. Les définitions de `true` et `false` suivent cette règle. Ainsi, la valeur entière de `true` est `1` et celle de `false` est `0`.

10.2 Un pointeur `NULL` est-il assimilé à une valeur fausse ?

Oui, la valeur `NULL` est apparentée à un `0`. Les écritures `if(p != NULL)` et `if(p)` sont donc équivalentes. De même, `if(p == NULL)` est équivalent à `if( !p)`.

Voir aussi les questions 10.1, page 51 et 7.5, page 39.

10.3 Que donne l'opérateur `!` sur un nombre négatif ?

L'opérateur `!` sur un nombre négatif donne bien ce que l'on attend, à savoir `0`.

Voir aussi la question 10.1, page 51.

10.4 Que vaut l'expression `a[i] = i++` ?

Ce genre d'expression fait partie des « *undefined behaviour* », ou comportement indéfini. Cela signifie que le résultat d'une telle opération dépend du compilateur. L'opérateur `++` modifie la valeur de `i`, alors que celle-ci est utilisée ailleurs dans l'expression. C'est ce que l'on appelle un « effet de bords ».

10.5 Pourtant, `i++` vaut `i` ?

Effectivement, `i++` vaut `i`, avant l'incrémentation. Toutefois, rien ne dit dans quel sens est calculée l'expression `a[i] = i++`. Est-ce le `i++` qui est évalué avant le `a[i]`, ou le contraire ? On n'en sait rien, c'est pourquoi l'on dit que c'est un comportement indéfini.

10.6 En est-il de même pour `i++ * i++` ?

Oui. La norme ne précise pas pour les opérateurs binaires dans quel ordre les opérandes sont évalués.

10.7 Peut-on utiliser les parenthèses pour forcer l'ordre d'évaluation d'une expression ?

Pas en général. Les parenthèses ne donnent qu'un ordre partiel d'évaluation, entre les opérateurs. Voici un petit exemple :

```
a = f() + g() * h();
b = (f() + g()) * h();
```

Les parenthèses dans la deuxième expression modifient l'ordre d'évaluation de l'addition et de la multiplication. Par contre, l'ordre dans lequel seront évaluées les fonctions est indéfini, dans l'une ou l'autre des deux expressions. Cela ne dépend que du compilateur.

Pour forcer un ordre d'évaluation, il faut utiliser une écriture séquentielle, avec des variables temporaires.

```
tf = f();
tg = g();
th = h();
b = (tf + tg) * th;
```

On a alors un comportement parfaitement défini sur toutes les cibles, quel que soit le compilateur.

10.8 Qu'en est-il des opérateurs logiques `&&` et `||` ?

Ces deux opérateurs forment une exception à la règle. La norme prévoit que les opérandes de ces deux opérateurs soient évalués de gauche à droite. De plus, l'évaluation s'arrête dès que le résultat est connu.

10.9 Comment sont évaluées les expressions comprenant plusieurs types de variables ?

Le principe est simple. Si les variables sont de types différents, il y aura un *cast* implicite vers le type le plus précis. On parle de promotion. Voici les règles de base :
– Si l'un des opérandes est un `long double`, l'autre est converti en un `long double`.
– Sinon, si l'un des opérandes est un `double`, l'autre est converti en un `double`.
– Sinon, si l'un des opérandes est un `float`, l'autre est converti en un `float`.
– Sinon, les opérandes de types `char` et `short` sont convertis en `int`.
– Enfin, si l'un des opérandes est un `long`, l'autre est converti en un `long`.
En C99, il faut rajouter le type `long long`.

Cela se complique dans le cas d'opérandes `unsigned`. Les comparaisons entre valeurs signées et non signées dépendent de la machine et de la taille des différents types.

10.10 Qu'est-ce qu'une *lvalue* ?

lvalue est un terme qui est utilisé pour définir les expressions que l'on peut mettre à gauche d'une affection. Toute variable modifiable est une *lvalue*.

Quand le compilateur prévient que le membre gauche d'une affectation n'est pas une *lvalue*, c'est souvent parce que l'on ne voulait pas faire une affectation, mais une comparaison (*cf.* 15.5, page 82).

Chapitre 11

Nombres en virgule flottante

11.1 J'ai un problème quand j'imprime un nombre réel.

Sur la plupart des architectures, les nombres réels (dits flottants) sont représentés en base 2, comme pour les entiers. Ainsi, le nombre 3.1 ne peut s'écrire exactement en base 2. La représentation binaire est un arrondi qui dépend de la précision du codage des flottants, et des choix du compilateur. De plus, avec une fonction comme `printf`, le nombre à imprimer est converti en base 2 puis reconverti en base 10, ce qui augmente encore les imprécisions.

Il est préférable d'utiliser les `double`, qui ont une précision supérieure aux `float`, sauf si l'économie de mémoire est vraiment critique. Voir à ce sujet la question 11.10, page 56.

11.2 Pourquoi mes extractions de racines carrées sont erronées ?

Assurez-vous d'avoir inclus `math.h`, d'avoir correctement déclaré les autres fonctions renvoyant des `double`. Une autre fonction de la bibliothèque standard avec laquelle il faut faire attention est `atof`, dans `stdlib.h`.

11.3 J'ai des erreurs de compilation avec des fonctions mathématiques

Il faut s'assurer d'avoir *linké* (lié) son code avec la bibliothèque mathématique. Par exemple, sous *Unix*, vous devez généralement passer l'option `-lm` à la fin de la ligne de commande.

11.4 Mes calculs flottants me donnent des résultats étranges et/ou différents selon les plateformes

Pour commencer, relisez 11.1, page 54.

Si le problème est plus complexe, il convient de se rappeler que les ordinateurs utilisent des formats de représentation des flottants qui ne permettent pas des calculs exacts. Pertes de précision, accumulation d'erreurs et autres anomalies sont le lot commun du numéricien.

Rappelez-vous qu'aucun calcul sur des flottants n'a de chance d'être exact, en particulier, n'utilisez jamais `==` entre deux flottants. Ces problèmes ne sont pas spécifiques au C.

Dans certains problèmes, une solution peut être d'introduire un petit paramètre de relaxation, par exemple `#define EPS 1e-10`, puis de multiplier l'un des termes (judicieusement choisi) de vos calculs par `(1 + EPS)`.

Pour plus de renseignements, on se reportera par exemple aux *Numerical Recipes* ou à *Numerical Algorithms with C* (cf. 3.9, page 18).

11.5 Comment simuler == entre des flottants ?

Étant donné qu'il y a perte de précision très vite, pour comparer deux valeurs flottantes, on teste si elles sont assez proches. Plutôt que d'écrire une horreur du genre :

```
double a, b;
/* ... */
if (a == b) /* HORREUR ! */
    /* ... */
```

on écrira :

```
#include <math.h>
/* ... */
double a, b;
/* ... */
if (fabs (a - b) <= epsilon * fabs (a) )
    /* ... */
```

où l'on aura judicieusement choisi `epsilon` (*non-nul !*).

11.6 Comment arrondir des flottants ?

La méthode la plus simple et la plus expéditive est `(int)(x + 0.5)`. Cette technique ne fonctionne pas correctement pour les nombres négatifs aussi vaut-il mieux utiliser

```
(int)(x < 0 ? x - 0.5 : x + 0.5)
```

11.7 Pourquoi le C ne dispose-t-il pas d'un opérateur d'exponentiation ?

Parce que certains processeurs ne disposent pas d'une telle instruction. Il existe une fonction `pow` déclarée dans `math.h` bien que la multiplication soit préférable pour de petits exposants.

11.8 Comment obtenir *Pi* ?

Parfois une constante prédéfinie `M_PI` est déclarée dans `math.h` mais ce n'est pas standard aussi vaut-il mieux calculer *pi* soi-même *via* `4 * atan (1.0)`.

11.9 Qu'est-ce qu'un `NaN` ?

« *NaN* `is Not a Number` », ce qui signifie « Ce n'est pas un nombre ». Un `NaN` est un nombre flottant qui est le résultat d'une opération non conforme, par exemple `0/0`. Lorsqu'un `NaN` est produit, la plupart des architectures produisent une interruption (ou un signal) qui termine le programme, au moment de l'utilisation de celui-ci. Il est parfois possible de vérifier si un nombre est `NaN`. Un bon test est celui-ci :

```
#define isNaN(x) ((x) != (x))
```

Certains compilateurs fournissent des facilités quant à la gestion des `NaN`. *GCC* fournit dans la bibliothèque mathématique (`math.h`) les fonctions `isnan`, `isinf` et `finite`.

11.10 Faut-il préférer les `double` aux `float` ?

La vitesse de traitement d'un `double` n'est pas forcément plus longue qu'un `float`, cela dépend du compilateur (de ses options) et du processeur. Ainsi avec l'exemple suivant, en remplaçant le `typedef` par `float` ou `double`, on s'aperçoit que sur un Pentium ou un PowerPC, le `double` est plus rapide à calculer que le `float` tout en ayant une précision plus grande.

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

typedef float reel;     /* float ou double */
```

```
int main(void) {
    long i ;
    reel d = 3.0 ;

    for (i = 0; i < 100000000; i++) {
        d = cos(d) ;
    }

    (void)printf("%f\n", d);
    return 0;
}
```

Le C comprend des instructions mathématiques pour traiter les float directement au lieu de toujours passer par des double depuis la dernière norme (C99). Par exemple il existe `cosf` en plus de `cos`. En faisant des essais on s'aperçoit que dans notre exemple, le `cosf` appliqué à un `float` devient aussi rapide que le `cos` appliqué à un `double`.

En conclusion, nous pouvons dire qu'il est préférable d'utiliser des `double` à la place des `float`, sauf lorsque la place mémoire devient critique.

Chapitre 12

Allocation dynamique

12.1 Doit-on ou ne doit-on pas caster `malloc` ?

Cette question est probablement celle qui revient le plus souvent dans la discussion. Et à chaque fois, elle engendre une longue discussion.

Certains intervenants pensent que caster la valeur de retour de `malloc` est inutile, voire dangereux. En effet, `malloc` renvoie un `void *`. Or, en C, un pointeur `void *` est implicitement casté lors d'une affectation vers le type de la variable affectée. Bien sûr, expliciter le cast n'est pas interdit, et est parfois utile. Toutefois, *caster* le retour de `malloc` risque de cacher au compilateur l'oubli du prototype de `malloc`. Ce prototype se trouve dans le fichier d'en-tête `<stdlib.h>`. Sans lui, `malloc` sera par défaut une fonction retournant un `int` et dont les paramètres seront du type des arguments passés, ce qui peut provoquer de sérieux bugs.

La véritable erreur est l'oubli du fichier d'en-tête `<stdlib.h>`, et non pas le cast de `malloc` en lui même. Mais le cast de `malloc` risque de cacher au compilateur cette erreur. À noter qu'il existe des outils de vérification de code et des options sur la plupart des compilateurs[1] qui permettent de détecter ce genre d'erreur.

D'autres intervenants jugent qu'il faille tout de même caster le retour de `malloc`, afin de conserver une compatibilité avec d'anciens compilateurs pré-ANSI, ou pour intégrer plus facilement le code avec C++. Evidemment, les programmeurs avertis sauront dans quelles situations il est utile ou non de caster les `void *`.

Voir aussi la question 7.8, page 40

12.2 Comment allouer proprement une variable ?

Le plus portable et le plus simple est de faire ainsi :

```
var_t * ma_var = malloc(N * sizeof *ma_var);
```

[1] pour GCC par exemple, l'option -Wall active -Wmissing-prototypes -Wmissing-declarations

Si le type de la variable change, l'allocation est toujours valide. À noter que l'on ne *caste* pas le retour de `malloc`

Voir la question 12.1, page 58 à ce sujet, ainsi que la question 12.10, page 60.

12.3 Pourquoi mettre à `NULL` les pointeurs après un `free` ?

La fonction `free` libère l'espace mémoire pointé par le pointeur en question. Mais la valeur de celui-ci ne peut-être changée, car en C les arguments sont passés par valeur aux fonctions.

La variable pointeur contient après le `free` une adresse invalide. Son utilisation peut entraîner de sérieux embêtements. Pour éviter cela, une bonne solution consiste à affecter la valeur `NULL` au pointeur après l'appel à `free`.

Il existe aussi certaines implémentations de l'allocation dynamique qui fonctionnent en *Garbage Collector*, c'est-à-dire, que la mémoire n'est réellement libérée que lorsque le pointeur est mis à `NULL`.

Dans tous les cas, cela permet de tester facilement la validité des pointeurs.

12.4 Pourquoi `free` ne met pas les pointeurs à `NULL` ?

Rappelons que les paramètres des fonctions sont passés par valeur (ou par copie). Ainsi, pour modifier la valeur du pointeur, il faudrait passer un pointeur sur le pointeur, ce qui compliquerait l'utilisation de `free`. Mais ce n'est pas le cas, il faut donc le faire soi-même.

12.5 Quelle est la différence entre `malloc` et `calloc` ?

Pratiquement, `calloc` est équivalent à :

```
/* p = calloc(m, n); */
p = malloc(m * n);
memset(p, 0, m * n);
```

Chaque élément est initialisé à `0`. Ce `0` est un « tout bit à zéro ». La valeur des éléments n'est pas forcément valide, suivant leur type.

Voir aussi la question 5.6, page 29.

12.6 Que signifie le message « assignment of pointer from integer » quand j'utilise `malloc` ?

Cela signifie que vous avez oublié d'inclure le fichier `stdlib.h`.

Voir à ce sujet la question 12.1, page 58.

12.7 Mon programme plante à cause de `malloc`, cette fonction est-elle buggée ?

Il est assez facile de corrompre les structures de données internes de `malloc`. Les sources les plus plausibles du problème sont :

– l'emploi de `malloc(strlen(s))` au lieu de `malloc(strlen(s)+1)`.
– la libération d'un pointeur deux fois.

Il y en a d'autres...

Voir aussi les questions 12.2, page 58 et 12.8, page 60.

12.8 Que signifient les erreurs « *segmentation fault* » et « *bus error* » ?

Cela signifie que vous avez essayé d'accéder à une zone mémoire non autorisée. C'est souvent l'utilisation d'un pointeur non initialisé ou `NULL` qui en est la cause. Ce genre d'erreur peut aussi provenir d'une mauvaise allocation (*cf.* 12.7, page 60 et 12.2, page 58) ou de l'oubli du `0` en fin de chaîne.

12.9 Doit-on libérer explicitement la mémoire avant de quitter un programme ?

Oui, car tous les systèmes ne le font pas d'eux-mêmes.

12.10 Du bon usage de `realloc`

La fonction `realloc` permet de modifier la taille de l'espace mémoire alloué à une variable. Elle est souvent utilisée pour augmenter cette taille.

Rappelons que la mémoire allouée par `malloc`, `calloc` et `realloc` est fournie sous la forme d'une zone continue (en un seul bloc). Or, il peut arriver que la nouvelle taille demandée dépasse l'espace disponible derrière la zone initiale. Dans ce cas, la fonction `realloc` alloue une nouvelle zone ailleur, là ou il y a de la place, et y recopie les données initiales. L'ancienne zone est alors libérée.

C'est pourquoi `realloc` renvoie un pointeur sur la nouvelle zone mémoire, même si l'augmentation de taille (ou la réduction) a pu se faire sur place. Bien sûr, comme `malloc`, `realloc` peut échouer.

Voici pour résumer une bonne manière d'utiliser `realloc` :

```
#include <stdlib.h> /* pour realloc() et free() */

/* ... */

int * var = NULL ;
var = malloc(sizeof * var * 42) ;
if (!var) {
    /* gestion des erreurs */
}

/* ... */

int * tmp = NULL ;
tmp = realloc(var, 84) ;
if (tmp) {
    var = tmp ;
}
else {
    /* gestion de l'erreur */
}
```

Chapitre 13

Le pré-processeurs

13.1 Quel est le rôle du préprocesseur ?

Le préprocesseur interprète les directives qui commencent par `#`. Principalement, ces directives permettent d'inclure d'autres fichiers (via `#include`) et de définir des macros (via `#define`) qui sont remplacées lors de la compilation.

Chaque directive de compilation commence par un `#` situé en début de ligne (mais éventuellement précédé par des espaces, des tabulations ou des commentaires) et se termine en fin de ligne.

Le préprocesseur est également responsable de la reconnaissance des trigraphes, des *backslashs* terminaux, et de l'ablation des commentaires.

13.2 Qu'est-ce qu'un trigraphe ?

Dans les temps anciens, les ordinateurs n'utilisaient pas ASCII ; chaque machine avait son propre jeu de caractères. L'ISO a défini un jeu de caractères supposés présents sur toutes les machines, c'est l'*Invariant Code Set ISO 646-1983*. Ce jeu de caractères ne comporte pas certains caractères intéressants, tels que les accolades et le *backslash*. Aussi, le standard C89 a introduit les trigraphes, séquences de trois caractères commençant par `??`. Il existe neuf séquences remplacées par le préprocesseur. Ce remplacement a lieu avant toute autre opération, et agit également dans les commentaires, les chaînes constantes, etc.

Les trigraphes sont, de fait, rarement utilisés. On les voit apparaître occasionnellement et par erreur, quand on écrit ça :

```
printf("Kikoo ??!\n");
```

Le trigraphe est remplacé par un *pipe*, donc ce code affiche ceci :

```
Kikoo |
```

De toutes façons, le redoublement du point d'interrogation est de mauvais goût.

Les compilateurs modernes soient ne reconnaissent plus les trigraphes, soit émettent des avertissements quand ils les rencontrent.

13.3 À quoi sert un *backslash* en fin de ligne ?

Après le remplacement des trigraphes, le préprocesseur recherche tous les caractères *backslash* situés en fin de ligne ; chaque occurrence de ce caractère est supprimée, de même que le retour à la ligne qui le suit. Ceci permet d'unifier plusieurs lignes en une seule.

Ce comportement est pratique pour écrire des macros ou des chaînes de caractères sur plusieurs lignes :

```
printf("Hello\
 World !\n");
```

Pour les chaînes de caractères, on peut aussi écrire plusieurs chaînes côte à côte, et le compilateur les unifiera (mais pas le préprocesseur : pour lui, ce seront deux chaînes à la suite l'une de l'autre). C'est une technique qui doit être généralement préférée. On écrira donc pour l'exemple précédent :

```
printf("Hello"
       "World !\n");
```

13.4 Quelles sont les formes possibles de commentaires ?

Un commentaire en C commence par `/*` et se termine par `*/`, éventuellement plusieurs lignes plus loin. Les commentaires ne s'imbriquent pas.

On peut aussi utiliser la compilation conditionnelle, comme ceci :

```
#if 0
    /* ceci est ignore */
#endif /* 0 */
```

Dans ce cas, il faut que ce qui est ignoré soit une suite de *tokens* valide (le préprocesseur va quand même les regarder, afin de trouver le `#endif`). Ceci veut dire qu'il ne faut pas de chaîne non terminées. Ce genre de commentaire n'est pas adapté à du texte, à cause des apostrophes.

La nouvelle norme du C (C99) permet d'utiliser les commentaires du C++ : ils commencent par `//` et se terminent en fin de ligne. Ce type de commentaire n'est pas encore supporté partout, donc mieux vaut ne pas s'en servir si on veut faire du code vraiment portable, même si avant C99, des compilateurs acceptaient ce type de commentaires.

13.5 Comment utiliser `#include` ?

`#include` comporte trois formes principales :

```
#include <fichier>
#include "fichier"
#include tokens
```

La première forme recherche le fichier indiqué dans les répertoires système ; on peut les ajuster soit via des menus (dans le cas des compilateurs avec une interface graphique), soit en ligne de commande. Sur un système *Unix*, le répertoire système classique est `/usr/include/`. Une fois le fichier trouvé, tout se passe comme si son contenu était tel quel dans le code source, là où se trouve le `#include`.

La deuxième forme recherche le fichier dans le répertoire courant. Si le fichier ne s'y trouve pas, il est ensuite cherché dans les répertoires systèmes, comme dans la première forme.

La troisième forme, où ce qui suit le `#include` ne correspond pas à une des deux formes précédentes, commence par effectuer tous les remplacements de macros dans la suite de tokens, et le résultat doit être d'une des deux formes précédentes.

Si le fichier n'est pas trouvé, c'est une erreur, et la compilation s'arrête. On notera que si on se sert d'habitude de `#include` pour inclure des fichiers d'en-tête (tels que `stdio.h`), ce n'est pas une obligation.

13.6 Comment éviter l'inclusion multiple d'un fichier ?

Il arrive assez souvent qu'un fichier inclus en incluse un autre, qui lui-même en inclut un autre, etc. On peut arriver à des boucles, qui peuvent conduire à des redoublements de déclarations (donc des erreurs, pour `typedef` par exemple), voire des boucles infinies (le compilateur finissant par planter).

Pour cela, le moyen le plus simple est de « protéger » chaque fichier par une construction de ce genre :

```
#ifndef FOO_H_
#define FOO_H_
/* ici, contenu du fichier */
#endif /* FOO_H_ */
```

Ainsi, même si le fichier est inclus plusieurs fois, son contenu ne sera actif qu'une fois. Certains préprocesseurs iront même jusqu'à reconnaître ces structures, et ne pas lire le fichier si la macro de protection (ici `FOO_H_`) est encore définie.

Il y a eut divers autres moyens proposés, tels que `#import` ou `#pragma once`, mais ils ne sont pas standards, et encore moins répandus.

13.7 Comment définir une macro ?

On utilise `#define`. La forme la plus simple est la suivante :

```
#define FOO  3 + 5
```

Après cette déclaration, toute occurrence de l'identificateur `FOO` est remplacée par son contenu (ici `3 + 5`). Ce remplacement est syntaxique et n'a pas lieu dans les chaînes de caractères, ou dans les commentaires (qui n'existent déjà plus, de toutes façons, à cette étape).

On peut définir un contenu vide. La macro sera remplacée, en cas d'utilisation, par rien. Le contenu de la macro est une suite de *tokens* du C, qui n'a pas à vouloir dire quelque chose. On peut définir ceci, c'est valide :

```
#define FOO  (({/coucou+}[\}+ "zap" 123
```

La tradition est d'utiliser les identificateurs en majuscules pour les macros ; rien n'oblige cependant à appliquer cet usage. Les règles pour les identificateurs de macros sont les mêmes que pour celles du langage.

13.8 Comment définir une macro avec des arguments ?

On fait ainsi :

```
#define FOO(x, y)   ((x) + (x) * (y))
```

Ceci définit une macro qui attend deux arguments ; notez qu'il n'y a pas d'espace entre le nom de la macro (`FOO`) et la parenthèse ouvrante. Toute invocation de la macro par la suite est remplacée par son contenu, les arguments l'étant aussi. Ainsi, ceci :

```
FOO(bla, "coucou")
```

devient ceci :

```
((bla) + (bla) * ("coucou"))
```

(ce qui ne veut pas dire grand'chose en C, mais le préprocesseur n'en a cure). Le premier argument est remplacé deux fois, donc, s'il a des effets de bord (appel d'une fonction, par exemple), ces effets seront présents deux fois.

Si la macro est invoquée sans arguments, elle n'est pas remplacée. Cela permet de définir une macro sensée remplacer une fonction, mais en conservant la possibilité d'obtenir un pointeur sur la fonction. Ainsi :

```
int min(int x, int y) { return x < y ? x : y; }
#define min(x, y)   ((x) < (y) ? (x) : (y))
min(3, 4);    /* invocation de la macro */
(min)(3, 4);  /* invocation de la fonction, via le pointeur */
```

C'est une erreur d'invoquer une macro à argument avec un nombre incorrect d'arguments. En C99, on peut utiliser des arguments vides ; en C89, c'est flou et mieux vaut éviter.

13.9 Comment faire une macro avec un nombre variable d'arguments ?

Ce n'est possible qu'en C99. On utilise la construction suivante :

```
#define error(l, ...)    { \
        fprintf(stderr, "line %d: ", l); \
        fprintf(stderr, __VA_ARGS__); \
    }
```

Ceci définit une macro, qui attend au moins un argument ; tous les arguments supplémentaires sont concaténés, avec leurs virgules de séparation, et on peut les obtenir en utilisant `__VA_ARGS__`. Ainsi, ceci :

```
error(5, "boo: '%s'\n", bla)
```

sera remplacé par ceci :

```
{ fprintf(stderr, "line %d: ", 5); \
  fprintf(stderr, "boo: '%s'\n", bla); }
```

Ce mécanisme est supporté par les dernières versions de la plupart des compilateurs C actuellement développés ; mieux vaut l'éviter si le code doit aussi fonctionner avec des compilateurs un peu plus anciens.

Il existe aussi des extensions sur certains compilateurs. Par exemple, sous GCC, le code suivant est équivalent à l'exemple précédent :

```
#define error(l, format...)    { \
        fprintf(stderr, "line %d: ", l); \
        fprintf(stderr, format); \
    }
```

Une autre méthode consiste à utiliser le parenthèsage des arguments :

```
#define PRINTF(s) printf s
...
PRINTF (("Vitesse du vent %d m/s", v));
```

13.10 Que font les opérateurs # et ## ?

L'opérateur `#` permet de transformer un argument d'une macro en une chaîne de caractères. On fait ainsi :

```
#define BLA(x)    printf("l'expression '%s' retourne %d\n", #x, x);
BLA(5 * x + y);
```

ce qui donne le résultat suivant :

```
printf("l'expression '%s' retourne %d\n", "5 * x + y", 5 * x + y);
```

Les éventuelles chaînes de caractères et backslashs dans l'argument sont protégés par des *backslashes*, afin de constituer une chaîne valide.

L'opérateur `##` effectue la concaténation de deux *tokens*. Si le résultat n'est pas un *token* valide, alors c'est une erreur ; mais certains préprocesseurs sont peu stricts et se contentent de re-séparer les *tokens*. On l'utilise ainsi :

```
#define FOO(x, y)   x ## y
FOO(bar, qux)();
```

qui donne ceci :

```
barqux();
```

13.11 Une macro peut-elle invoquer d'autres macros ?

Oui. Mais il est prévu un mécanisme qui empêche les boucles infinies.

Tout d'abord, les invocations de macros ne sont constatées que lors de l'utilisation de la macro, pas lors de sa définition. Si on fait ceci :

```
#define FOO    BAR
#define BAR    100
```

alors on obtient bien `100`, pas `BAR`.

Si la macro possède des arguments, chaque fois que cet argument est utilisé (sans être précédé d'un `#` ou précédé ou suivi d'un `##`), il est d'abord examiné par le préprocesseur, qui, s'il y reconnaît une macro, la remplace. Une fois les arguments traités, le préprocesseur les implante à leur place dans la suite de *tokens* générés par la macro, et gère les opérateurs `#` et `##`.

À la suite de cette opération, le résultat est de nouveau examiné pour rechercher d'autres remplacements de macros ; mais si une macro est trouvée, alors qu'on est dans le remplacement de ladite, cette macro n'est pas remplacée. Ceci évite les boucles infinies.

Je sais, c'est compliqué. Quelques exemples :

```
#define FOO    coucou BAR
#define BAR    zoinx FOO
FOO
```

`FOO` est remplacée par `coucou BAR`, et le `BAR` résultant est remplacé par `zoinx FOO`. Ce `FOO` n'est pas remplacé, parce qu'on est dans le remplacement de `FOO`. Donc, on obtient `coucou zoinx FOO`.

Un autre exemple, plus tordu :

```
#define FOO(x)   x(5)
FOO(FOO);
```

La macro `FOO` est invoquée ; elle attend un argument, qui est `FOO`. Cet argument est d'abord examiné ; il y a `FOO` dedans, mais non suivi d'une parenthèse ouvrante (l'argument est examiné tout seul, indépendamment de ce qui le suit lors de son usage), donc le remplacement n'a pas lieu. Ensuite, l'argument est mis en place, et on obtient `FOO(5)`. Ce résultat est réexaminé ; cette fois, `FOO` est bien invoquée avec un argument, mais on est dans le deuxième remplacement, à l'intérieur de la macro `FOO`, donc on ne remplace pas. Le résultat est donc : `FOO(5) ;`

Si vous voulez utiliser ce mécanisme, allez lire une douzaine de fois la documentation du *GNU cpp*, et surtout le paragraphe 12 de l'annexe A du Kernighan & Ritchie (2ème édition).

13.12 Comment redéfinir une macro ?

On peut redéfinir à l'identique une macro ; ceci est prévu pour les fichiers d'en-tête inclus plusieurs fois. Mais il est en général plus sain de protéger ses fichiers contre l'inclusion multiple (*cf.* 13.6, page 64).

Redéfinir une macro avec un contenu ou des arguments différents, est une erreur. Certains préprocesseurs laxistes se contentent d'un avertissement. La bonne façon est d'abord d'indéfinir la macro via un `#undef`. Indéfinir une macro qui n'existe déjà pas, n'est pas une erreur.

13.13 Que peut-on faire avec `#if` ?

`#if` permet la compilation conditionnelle. L'expression qui suit le `#if` est évaluée à la compilation, et, suivant son résultat, le code qui suit le `#if` jusqu'au prochain `#endif`, `#elif` ou `#else` est évalué, ou pas. Quand le code n'est pas évalué, les directives de préprocesseur ne le sont pas non plus ; mais les `#if` et similaires sont néanmoins comptés, afin de trouver la fin de la zone non compilée.

Lorsque le préprocesseur rencontre un `#if`, il :
– récupère l'expression qui suit le `#if`
– remplace les `defined MACRO` et `defined(MACRO)` par la constante `1` si la macro nommée est définie, `0` sinon
– remplace les macros qui restent dans l'expression
– remplace par la constante `0` tous les identificateurs qui restent
– évalue l'expression
L'expression ne doit comporter que des constantes entières (donc, éventuellement, des constantes caractères), qui sont promues au type `(unsigned) long` (en C89) ou `(u)intmax_t` (en C99). Les flottants, les pointeurs, l'accès à un tableau, et surtout l'opérateur `sizeof` ne sont pas utilisables par le préprocesseur.

Il n'est pas possible de faire agir un `#if` suivant `sizeof(long)`, pour reprendre un *desiderata* fréquent. Par ailleurs, les constantes de type caractère n'ont pas forcément la même valeur pour le préprocesseur et pour le compilateur.

13.14 Qu'est-ce qu'un `#pragma` ?

C'est une indication pour le compilateur. Le préprocesseur envoie cette directive sans la modifier. Le standard C89 ne prévoit aucune directive standard, mais le préprocesseur comme le compilateur sont sensés ignorer les directives inconnues.

Le C99 définit trois `#pragma` qui permettent d'ajuster le comportement du compilateur, quant au traitement des nombres flottants et complexes.

13.15 Qu'est-ce qu'un `#assert` ?

C'est une extension gérée par *GNU* et (au moins) le compilateur *Sun* (*Workshop Compiler*, pour *Solaris*). C'est une sorte d'alternative à `#ifdef`, avec une syntaxe plus agréable. C'est à éviter, car non standard.

13.16 Comment définir proprement une macro qui comporte plusieurs *statements* ?

On peut ouvrir un bloc, car tout *statement* est remplaçable, en C, par un bloc, mais cela pose des problèmes avec le `;` terminal du *statement*. La manière recommandée est la suivante :

```
#define foo(x)   do { f(x); printf("coucou\n"); } while (0)
```

On peut ainsi l'utiliser comme ceci :

```
if (bar) foo(1); else foo(2);
```

Si on avait défini `foo` sans le `do` et le `while (0)`, le code ci-dessus aurait provoqué une erreur de compilation, car le `else` serait séparé du `if` par deux statements : le bloc et le statement vide, terminé par le point-virgule.

13.17 Comment éviter les effets de bord ?

Les macros peuvent être dangereuses si l'on ne fait pas attention aux effets de bord. Par exemple si l'on a le code suivant :

```
#define MAX   3 + 5

int i = MAX;
```

La variable `i` vaudra bien 8, mais si on utilise la macro `MAX` ainsi :

```
int i = MAX * 2;
```

La variable `i` ne vaudra pas 16, mais 13. Pour éviter ce genre de comportement, il faut écrire la macro ainsi :

```
#define MAX   (3 + 5)
```

Dans certains cas, une macro représente une expression C complète. Il est alors plus cohérent de placer des parenthèses vides pour simuler une fonction. Et dans ce cas il ne faut pas la terminer par un point virgule, et

```
#define PRTDEBUG()   (void)printf("Coucou\n")
```

ainsi, on pourra utiliser la macro par :

```
if (i == 10) {
   PRTDEBUG();
}
else {
   i++;
}
```

Quand une macro a des arguments il faut faire attention à la façon de les utiliser. Ainsi la macro :

```
#define CALCUL(x, y)   (x + y * 2)
```

a des effets de bord suivant la façon de l'utiliser :

```
int i = CALCUL(3, 5);
```

donnera bien un résultat de 13, alors que le même résultat serait attendu avec :

```
int i = CALCUL(3, 2 + 3);
```

qui donne 11. Pour éviter cela, il suffit de placer des parenthèses sur les arguments de la macro :

```
#define CALCUL(x, y)   ((x) + (y) * 2)
```

Un effet de bord qui ne peut être contourné survient quand la macro utilise plusieurs fois une variable :

```
#define MAX(x, y)   ((x) > (y) ? (x) : (y))
```

Si on utilise la macro comme cela :

```
i = MAX(j, k);
```

On obtiendra un résultat correct, alors qu'avec :

```
i = MAX(j++, k++);
```

une des variables `j` ou `k` sera incrémentée 2 fois. Pour éviter ce genre de comportement, il faut remplacer la macro par une fonction, de préférence inline (C99) :

```
inline int max(int x, int y)
{
    return x > y ? x : y;
}
```

En règle générale, quand on utilise une fonction avec un nom en majuscule (comme MAX), on s'attend à ce que ce soit en fait une macro, avec les effets de bord qui en découlent. Alors que si le nom est en minuscule, il s'agit sûrement d'une véritable fonction. Cette règle n'est hélas pas générale et donc il convient de vérifier le véritable type d'une fonction si l'on ne veut pas être surpris lors de son utilisation.

13.18 Le préprocesseur est-il vraiment utile ?

La réponse est oui. Il existe un mouvement qui voudrait faire disparaître le préprocesseur, en remplaçant les `#define` par des variables constantes, et avec un quelconque artifice syntaxique pour importer les déclarations d'un fichier d'entête.

Il s'avère qu'on a vraiment besoin d'un mécanisme polymorphe (comme une fonction qui accepterait plusieurs types différents), et que seules les macros apportent ce mécanisme en C. Les anti-préprocesseurs acharnés parlent d'adopter le mécanisme des templates du C++, mais ça ne risque pas d'arriver de sitôt.

Dans la vie de tous les jours, l'utilisation du préprocesseur, avec quelques `#include` et des `#define` sans surprise, ne pose pas de problème particulier, ni de maintenance, ni de portabilité.

13.19 Approfondir le sujet.

Outre le Kernighan & Ritchie, qui comporte quelques pages très bien faites sur le sujet, on peut lire la documentation au format `info` du *GNU cpp* ; elle est assez partiale

dans certains cas (condamnation explicite des trigraphes, par exemple) mais assez riche en enseignements.

Pour le reste, chaque compilateur C vient avec un préprocesseur, et il existe quelques préprocesseurs indépendants (un écrit par Dennis RITCHIE en personne, qu'on voit inclus dans *lcc*, et aussi *ucpp*, une oeuvre à moi : http://www.di.ens.fr/~pornin/ucpp/).

Chapitre 14

Fonctions de la bibliothèque

14.1 Comment convertir un nombre en une chaîne de caractères ?

Il suffit d'utiliser `sprintf`.

```
char sz[4];
sprintf(sz, "%d", 123);
```

Pour convertir un `double` ou un `long`, il faut utiliser `%f` ou `%ld`.

Le problème de `sprintf` est qu'il faut réserver assez de place pour la chaîne résultat. Ainsi le code

```
char sz[6];
sprintf(sz, "%u", i);
```

marchera sur des machines où `i` est un entier 16 bits, mais il plantera si `i` est un entier plus grand (> 99999).

En C99, la fonction snprintf permet d'éviter les débordements :

```
char sz[4];
n = snprintf(sz, sizeof sz, "%d", 123);
if (n < 0 || n >= sizeof sz)
    erreur();
```

14.2 Comment convertir une chaîne en un nombre ?

Si le nombre espéré est un entier, il faut utiliser la fonction `strtol`. Elle convertit une chaîne en un entier long, dans une base donnée.

Si le nombre est un réel (`float` ou `double`), alors la fonction `strtod` fera très bien l'affaire.

```
char test[] = "  -123.45e+2";
char * err = NULL;

errno = 0;
double result = strtod(test, &err);

if (err == test) {
    printf("Erreur de conversion :\n");
}
else if (errno == ERANGE) {
    printf("Depassement :\n");
}
else {
    printf("Conversion reussie :\n");
    if(*err == '\0') {
        printf("Pour toute la chaine\n");
    }
}
```

Si le nombre est un `long double` (C99 seulement) alors la fonction `strtold` est à préférer.

14.3 Comment découper une chaîne ?

La fonction `strtok` est faite pour ça !

```
char sz1[] = "this is,an   example ; ";
char sz2[] = ",; ";
char *p;

p = strtok(sz1, sz2);
if (p != NULL) {
    puts(p);
    while ((p = strtok(NULL, sz2)) != NULL) {
        puts(p);
    }
}
```

Attention, la fonction `strtok` souffre au moins des problèmes / caractéristiques suivants :
- Elle fusionne les délimiteurs adjacents. En cas d'utilisation d'une virgule comme séparateur, "a,,b,c" est séparée en trois éléments et non quatre.
- Le caractère du délimiteur est perdu, car il est remplacé par un caractère nul (0).

– Elle modifie la chaine qu'elle analyse. C'est un défaut, car cela oblige à faire une copie de la chaine en cas d'utilisation ultérieure. Cela signifie aussi que l'on ne peut pas séparer une chaine littérale avec.
– On ne peut utiliser qu'un appel de cette fonction à la fois. Si une séquence de `strok` est en cours, et qu'une autre démarre, l'état de la première est perdue. Ce n'est pas grave pour les petits programmes, mais il est facile de se perdre dans la hiérarchie des fonctions imbriquées dans des programmes plus importants. En d'autre termes, `strtok` brise les principes de l'encapsulation.

Dans des cas simples, on pourra utiliser la fonction `strchr`.

14.4 Pourquoi ne jamais faire `fflush(stdin)` ?

La fonction `fflush` a un comportement défini uniquement sur les flux ouverts en écriture tels que `stdout`. Il est possible que sur votre système, appliquer cette fonction à `stdin` soit possible, mais c'est alors une extension non standard. Le comportement est indéterminé, et imprévisible.

Il faut bien comprendre que `stdin` n'est pas forcément relié au clavier, mais peut être rattaché à un réseau, un fichier, etc.

14.5 Comment vider le buffer associé à `stdin` ?

Une bonne manière est de lire sur le flux tant qu'il n'est pas vide, avec les fonctions habituelles comme `fgets` ou `getchar`. Voici un exemple avec cette dernière :

```
c = getchar();
if (c != '\n')
    while ( (getchar()) !=  '\n') {
};
```

Ce morceau de code permet de lire un caractère, et vide ce qui peut rester dans le buffer, notamment le `'\n'` final.

14.6 Pourquoi mon `printf` ne s'affiche pas ?

Le flux standard `stdout`, sur lequel écrit `printf` est bufferisé. C'est à dire que les caractères sont écrits dans un tampon (une zone mémoire). Lorque celui-ci est plein, ou lorsqu'une demande explicite est faite, il est vidé dans le flux proprement dit (sur l'écran généralement). Tant que le buffer n'est pas vidé, rien ne s'affiche.

Pour vider le buffer, il y a trois possibilités :

– Le buffer est plein
– Il est vidé explicitement par l'appel de la fonction `fflush` (*cf.* 14.4, page 76)
– La chaîne de caractères se *termine* par un `'\n'`

14.7 Comment obtenir l'heure courante et la date ?

Il faut simplement utiliser les fonctions `time`, `ctime` et/ou `localtime`, qui contrairement à leurs noms donnent l'heure et la date.

Voici un petit exemple :

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

int main(void) {
    time_t now;
    time(&now);

    printf("Il est %.24s.\n", ctime(&now));

    return 0;
}
```

14.8 Comment faire la différence entre deux dates ?

Il faut simplement utiliser la fonction `difftime`. Cette fonction prend deux `time_t` en paramètres et renvoie un `double`.

14.9 Comment construire un générateur de nombres aléatoires ?

Ce n'est pas possible. La bibliothèque standard inclut un générateur pseudo-aléatoire, la fonction `rand`.

Toutefois, l'implémentation dépend du système, et celle-ci n'est généralement pas très bonne (en terme de résultats statistiques). Si `rand` ne vous suffit pas (simulation numérique ou cryptologie), il vous faudra regarder du coté de bibliothèques mathématiques, dont de nombreuses se trouvent sur Internet. En particulier, on consultera les paragraphes 7-0 et 7-1 des *Numerical Recipes in C* (*cf.* 4.5, page 23) et le volume 2 de TAoCP (*cf.* 3.9, page 19).

14.10 Comment obtenir un nombre pseudo-aléatoire dans un intervalle ?

La méthode la plus simple à faire,

```
rand() % N
```

qui renvoie un nombre entre `0` et `N-1` est aussi la moins bonne. En effet, les bits de poids faibles ont une distribution très peu aléatoire. Par exemple, le bit de poids le plus faible a une distribution qui peut être celle-ci sur un mauvais générateur :

`0 1 0 1 0 1 0 1 0 1 ...`

Voici la méthode préconisée dans *Numerical Recipes* (*cf.*4.5, page 23) :

```
(int)((double)rand() / ((double)RAND_MAX + 1) * N)
```

`RAND_MAX` est défini dans `stdlib.h`, et `N` doit être plus petit que `RAND_MAX`.

14.11 À chaque lancement de mon programme, les nombres pseudo-aléatoires sont toujours les mêmes ?

C'est normal, et c'est fait exprès. Pour contrer cela, il faut utiliser une graine pour le générateur qui change à chaque lancement du programme. C'est la fonction `srand` qui s'en charge.

On peut utiliser l'heure système, avec `time`, de la facon suivante :

```
srand(time(NULL));
```

Notez qu'il est peu utile d'appeler la fonction `srand` plus d'une fois par programme.

14.12 Comment savoir si un fichier existe ?

En C ISO, le seul moyen de savoir si un fichier existe, c'est d'essayer de l'ouvrir.

```
{
   FILE *fp = fopen ("fichier.txt", "r");

   if (fp == NULL)
   {
      fputs ("Le fichier n'existe pas,\n"
             "ou vous n'avez pas les droits necessaires\n"
             "ou il est inaccessible en ce moment\n"
             , stderr);
   }
   else
   {
      /* ... operation sur le fichier */

      fclose(fp);
```

```
    }
}
```

Dans la norme *POSIX*, il existe la fonction `access`, mais certains systèmes n'implémentent pas cette interface.

14.13 Comment connaître la taille d'un fichier ?

Malheureusement, les fonctions `stat` et `fstat` de la norme *POSIX* ne sont pas reprises dans la norme ISO. La seule solution standard est d'utiliser `fseek` et `ftell`.

Toutefois, cela ne marche pas pour les très gros fichiers (supérieurs à `LONG_MAX`).

14.14 Comment lire un fichier binaire proprement ?

Il faut ouvrir le fichier en mode « binaire », en passant la chaîne `"rb"` en mode d'ouverture à la fonction `fopen`. Cela évite les transformations inopportunes et les problèmes des caractères de contrôle.

De même, pour écrire dans un fichier binaire, on utilise le mode `"wb"`.

14.15 Comment marquer une pause dans un programme ?

Il n'y a pas de fonction standard pour cela. Il existe toutefois la fonction `sleep` en *POSIX*, elle provoque une attente passive pour une durée donnée en secondes.

14.16 Comment trier un tableau de chaînes ?

La fonction `qsort` est une bonne fonction de tri, qui implémente le *Quick Sort*. Le plus simple est de donner un exemple :

```
/* Fonction qui compare deux pointeurs
   vers des chaines pour qsort */
int pstrcmp(const void * p1, const void * p2){
    return strcmp(*(char * const *)p1, *(char * const *)p2);
}
```

Les paramètres doivent être des pointeurs génériques pour `qsort`. `p1` et `p2` sont des pointeurs sur des chaînes. Un tableau de chaînes doit être pris au sens d'un tableau de pointeurs vers des `char *`.

L'appel à `qsort` ressemble alors à :

```
qsort(tab, sizeof tab, sizeof *tab, pstrcmp);
```

14.17 Pourquoi j'ai des erreurs sur les fonctions de la bibliothèque, alors que j'ai bien inclus les entêtes ?

Les en-têtes (les `.h`) ne contiennent que les prototypes des fonctions. Le code proprement-dit de ces fonctions se trouve dans des fichiers objets. Ce code doit être « lié » au tien. Cela est fait par un éditeur de liens.

Pour certaines fonctions, il faut spécifier explicitement à l'éditeur de liens où il peut les trouver (et ce particulièrement pour les fonctions non-standard).

Par exemple, sous Unix, pour utiliser les fonctions mathématiques, il faut généralement lier le programme avec la bibliothèque adéquate :

```
cc -lm monfic.o -o monprog
```

Chapitre 15

Styles

15.1 Comment bien programmer en C ?

La chose la plus importante est de commenter un programme. Il ne s'agit pas de décrire en français tout ce que fait chaque ligne de code, mais de préciser le fonctionnement des opérations complexes, d'expliquer le rôle des variables, de dire à quoi servent les fonctions. Choisir des noms de variables et de fonctions explicites est une bonne façon de commenter un programme.

Tout morceau de code qui n'est pas standard doit être abondamment commenté afin de rendre le portage vers d'autres cibles le moins fastidieux possible, l'idéal étant d'utiliser des macros.

Il est également important de bien structurer son programme en modules, puis en fonctions. Certains vont jusqu'à dire qu'une fonction ne doit pas dépasser la taille d'un écran.

Les déclarations et prototypes doivent être regroupés dans des fichiers d'en-têtes, avec les macros.

Enfin, il est très important de bien présenter le code, avec une indentation judicieuse et des sauts de ligne. (*cf.* 15.2, page 81) Il est généralement admis que les lignes ne doivent pas dépasser 80 caractères.

Pour le reste, c'est une histoire de goût.

15.2 Comment indenter proprement du code ?

L'indentation du code est une chose essentielle pour la lisibilité. Certaines personnes utilisent des tabulations, ce qui est une mauvaise habitude. La largeur de ces tabulations varie d'un éditeur à un autre. Des éditeurs remplacent les tabulation par un nombre d'espaces fixe et d'autres encore utilisent des tabulations de taille variable. Ne parlons pas des imprimantes ou des lecteurs de *news*... Tout ceci rend l'utilisation des tabulations difficile.

Pour éviter tout problème, et améliorer la lisibilité du code, il faut utiliser uniquement des espaces. Un éditeur correct doit pouvoir générer un nombre d'espaces fixe (voire une

indentation automatique) lorsqu'on appuye sur la touche `<TAB>` (ou autre raccourci). Personnellement, je règle à 4 espaces par tabulation.

15.3 Quel est le meilleur style de programmation ?

Comme vous vous en doutez, il n'y en a pas. Le plus important est d'en avoir un, et de le suivre. Quand on utilise un type d'indentation ou un style de nommage, il faut l'utiliser dans tout le programme (voire dans tout ses programmes). C'est la régularité qui donne la lisibilité.

Il existe des styles de programmations fréquemment utilisés en C, comme les styles *K&R* ou *GNU*. Le style *K&R* est le style « historique », et c'est pourquoi il est très utilisé. Le style *GNU* est utilisé pour tous les projets de la *Free Software Fundation*.

Sur le site FTP `ftp://caramba.cs.tu-berlin.de`, le repertoire */pub/doc/style* contient quelques documents intéressants sur la question.

15.4 Qu'est-ce que la notation hongroise ?

C'est une convention de nommage des objets, inventée par Charles Simonyi. Le principe est de faire précéder le nom des variables par un identificateur de type. Par exemple, une chaîne de caractère représentant un nom sera nommée `szname`, `sz` signifiant « *string zero* », ou chaîne terminée par un `'\0'`.

Personnellement, je ne trouve pas cette convention toujours pratique. Le nom de la variable doit avant tout refléter son rôle.

15.5 Pourquoi certains écrivent-ils `if(0==x)` et non `if(x==0)` ?

Il arrive souvent que l'on écrive `=` au lieu de `==`. Comme `0` n'est pas une *lvalue* (*cf.* 10.10, page 53), cette étourderie provoquera une erreur, simple à détecter. Dans le même genre, on peut écrire `while (0 == x)`.

Certains compilateurs préviennent lorsque l'on fait une affectation là ou est attendu un test. C'est le cas de *GNU CC*, avec l'option *-Wall*.

Lorsque l'on écrit `while ( c = fct() )`, certains compilateurs râlent en croyant que l'on s'est trompé entre le `=` et le `==`. Pour éviter cela, il suffit de rajouter un paire de parenthèses.

```
while ( (c= fct()) ) {
    /* ... */
}
```

15.6 Pourquoi faut-il mettre les '{' et '}' autour des boucles ?

C'est une précaution contre les erreurs du genre :

```
for(i = 0; i < N; i++);
    tab[i] = i;
```

De plus, cela permet une plus grande simplicité dans l'évolution du code. En effet, les programmes ont tendance à s'épaissir avec le temps.

15.7 Pourquoi certains disent-ils de ne jamais utiliser les `goto` ?

Le `goto` est une instruction qui permet de casser l'aspect structuré d'un programme. Des `goto` mal utilisés permettent de rendre un code totalement illisible (code *spaghetti*), d'autant plus qu'avec les structures de boucles traditionnelles, on peut souvent s'en passer.

Toutefois, il arrive parfois que l'utilisation d'un `goto` rende le code plus propre. C'est le cas, par exemple, des sorties de boucles imbriquées en cas d'erreur. Cela rejoint le cas plus général des gestions d'exceptions internes.

Poser comme règle de ne *jamais* utiliser les `goto` est une absurdité. Par contre, avertir le programmeur de l'utiliser avec parcimonie, et avec beaucoup de précautions me semble une bonne chose.

15.8 Pourquoi commenter un `#endif` ?

`#endif` ne peut être suivi par autre chose qu'un commentaire. On commente donc pour savoir à quelle directive il correspond :

```
#if FOO
/* du code ou des directives */
#ifdef BAR
/* encore du code ou des directives */
#endif /* BAR */
/* encore du code ou des directives */
#endif /* FOO */
```

15.9 Où trouver de la doc sur les différents styles ?

Le document suivant contient des règles de base à suivre pour programmer proprement : ftp://ftp.laas.fr/pub/ii/matthieu/c-superflu/c-superflu.pdf

Sachez également qu'il existe un programme *indent* issu de *BSD* qui réindente automatiquement du code, suivant un style donné. Les options classiques de la version *GNU* de cet utilitaire sont `-kr` (pour le style décrit dans *K&R*), `-gnu` (pour le style utilisé dans les projets *GNU*) ou encore `-orig` (pour le style *BSD*).

Sous *Unix*, on trouve également la commande *cb* avec l'option `-sj` pour avoir le style *K&R*.

15.10 Comment bien structurer son programme ?

Il n'y a pas de réponse définitive, mais voici une piste :

- Suite à l'analyse (conception) il est possible de découper le projet en une multitude de systèmes et de sous-systèmes.
- Chaque sous-système fait l'objet d'un module composé d'un fichier source(xxx.c) et d'un fichier d'en-tête (xxx.h).
- Les systèmes sont organisés hiérarchiquement (arbre) de façon à éviter les dépendances croisées.

Evidement, tout ça est peu théorique. De plus, il existes des techniques avancées (ADT, callback) qui permettent les appels croisés propres. (c à d sans dépendances croisées). L'avantage est que chaque module est testable individuellement (et le plus souvent réutilisable).

La phase suivante du développement est l'intégration. Elle consiste à mettre la colle qui va bien pour faire tenir tout les modules ensembles et à tester le fonctionnement de l'ensemble.

Nous ne traitons pas ici des différentes méthodes d'analyse et de conception. Rappelons toutefois les 5 phases (en 'V') de développement d'un projet :

```
1 - Spécification      5 - Validation
 \                    /
  2 - Conception    4 - Integration
   \               /
    3 - Codage et test unitaire
```

Voir à ce sujet les questions 9.3, page 46 et 13.5, page 64.

Merci à Emmanuel Delahaye pour cette réponse.

Chapitre 16

Autres

16.1 Comment rendre un programme plus rapide ?

Il y a deux raisons possibles à la « lenteur » d'un programme.

La première vient de l'écriture du code lui-même, les entrées/sorties, les allocations dynamiques, de nombreux appels de fonctions, des boucles, *etc.* Le programmeur a généralement peu intérêt à modifier son code, tout au plus pourra-t-il remplacer les petites fonctions le plus souvent appelées par des macros et tenter de limiter les boucles. On pourra aussi améliorer les entrées/sorties et les allocations si c'est possible. Il reste enfin les options de compilation sur lesquelles on peut jouer.

L'autre raison vient de la complexité théorique des algorithmes utilisés. Dans ce dernier cas, il faut chercher un meilleur algorithme. Cet aspect est développé par exemple dans http://www.enseignement.polytechnique.fr/profs/informatique/Jean-Jacques.Levy/poly/polyx-cori-levy.ps.gz

Il existe des outils de *profilage* de programmes. Il s'agit de compiler les sources avec une bibliothèque puis de lancer le programme. On lance alors un lociciel associé à la bibliothèque. Le résultat est un fichier où est détaillé le temps passé dans chaque fonction, le nombre d'appels, *etc.* Sur *Unix-like*, le projet *GNU* propose *gprof* (*cf.* 4.6, page 24).

Rappelons tout de même que la vitesse d'exécution d'un programme (hors problèmes d'algorithmique) est peu souvent critique, et qu'il est bien plus important de fournir un code lisible.

16.2 Quelle est la différence entre *byte* et *octet* ?

L'unité de mémoire élémentaire du C est le *byte*. Le *byte* au sens anglo-saxon et donc pas l'*octet* au sens francophone (caractère).

En fait un *byte* correspond à un caractère non-signé (`unsigned char`), lequel peut prendre plus (ou moins) de 8 bits. En principe, en français, on parle dans ce cas de *multiplet* (et peut-être bientôt de *codet*) comme traduction officielle de *byte* dans ce sens.

(Merci à Antoine Leca).

16.3 Peut-on faire une gestion d'exceptions en C ?

Oui, c'est possible, en utilisant le couple `setjmp`/`longjmp`.

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

jmp_buf env;

long fact(long x) {
    long i, n;

    if (x < 0)
        longjmp(env, 1);
    for (n = 1, i = 2; i <= x; i ++)
        n *= i;
    return n;
}

long comb(long k, long n) {
    if (k < 0 || n < 1 || k > n)
        longjmp(env, 2);
    return fact(n) / (fact(k) * fact(n - k));
}

int main(int argc, char *argv[]) {
    if (argc < 3) {
        fprintf(stderr, "pas assez d'arguments\n");
        return EXIT_FAILURE;
    }
    if (setjmp(env)) {
        fprintf(stderr, "erreur de calcul\n");
        return EXIT_FAILURE;
    }
    printf("%ld\n", comb(strtol(argv[1], 0, 0),
           strtol(argv[2], 0, 0)));
    return 0;
}
```

Voilà un programme qui calcule le coefficient binomial des deux arguments ; `main` appelle `comb` qui appelle `fact`. Ces fonctions vérifient un peu les arguments, et on voudrait renvoyer une erreur en cas de problème ; mais :

– Je renvoie déjà le résultat, il faudrait qu'il y ait des résultats « impossibles » pour y coder les erreurs ; c'est le cas ici (le résultat est toujours supérieur ou égal à 1) mais ça demande une analyse mathématique pas toujours facile.

– Je ne veux pas tester les codes d'erreurs à chaque invocation d'une fonction. Cela peut devenir lourd syntaxiquement, et ça consomme du temps CPU.

`setjmp` sauvegarde l'état du programme au moment de l'appel, et renvoie `0`. `longjmp` remplace le contenu de la pile d'exécution par la sauvegarde, et le programme se trouve à nouveau à l'endroit de l'appel à `setjmp`. Celle-ci renvoie alors une valeur passée en paramètre de `longjmp` (dans l'exemple, `1` pour une erreur dans `fact` et `2` pour une erreur dans `comb`).

La méthode présentée ici est assez rustique. Il existe des mécanismes de POO[1] en C bien plus évolués. Vous pouvez à ce sujet allez voir l'excellent document : http://ldeniau.home.cern.ch/ldeniau/html/oopc/oopc.html. Allez voir aussi le document suivant : http://cern.ch/Laurent.Deniau/html/exception/exception.html.

Voir aussi la question 3.10, page 19.

16.4 Comment gérer le numéro de version de mon programme ?

Serge Paccalin propose la chose suivante :

```
#define STR_(a) #a
#define STR(a)  STR_(a)
#define VERSION 4
printf("This is version " STR(VERSION) " of the program\n");
```

En effet, quelque chose comme :

```
#define STR(a) #a
#define VERSION a
printf("This is version " STR(VERSION) " of the program\n");
```

fait intervenir la concaténation des chaînes trop tôt ce qui fait que le résultat de cette dernière séquence renvoie :

```
This is version VERSION of the program
```

16.5 Pourquoi ne pas mettre de '_' devant les identifiants ?

Well well well, ce n'est pas si facile.

Les vrais identificateurs réservés sont :

– les mots clés tels que `if`, `for`, `switch`, `long`, ...
– les identificateurs commençant par deux '_'
– les identificateurs commençant par un '_' suivi d'une lettre *majuscule*.

[1] Programmation Orienté Objet

Ensuite, il y a les headers standards et la bibliothèque. Les headers sont libres d'utiliser des identificateurs commençant par un '_' et suivis d'une lettre minuscule, comme '_liste', mais c'est pour définir quelque chose qui a un « file scope », c'est-à-dire une portée globale à la « translation unit » (le fichier source C et les fichiers qu'il inclut). Donc, globalement, on ne doit pas s'en servir pour définir quelque chose qui a ce « file scope ».

Ça veut dire quoi ? Que les choses suivantes sont interdites :

```
typedef int _foo;
struct _bar {
    int x;
    char * y;
};
void _f(void);
```

En revanche, les choses suivantes sont autorisées :

```
void f(int _x[]) {
    typedef int _foo;
    struct _bar {
        int x;
        char *y;
    };
    extern void _g(void);
}
struct qux {
    long _truc;
};
```

J'attire l'attention du public ébahi sur les quatre points suivants :

– En définissant des identificateurs à portée réduite (bloc, prototype, fonction), on peut masquer des identificateurs définis par les headers standards, identificateurs qui pouvaient intervenir dans des macros définies dans lesdits headers ; comme toutes les fonctions standards peuvent être surchargées par des macros, à l'intérieur d'un bloc où on a joué à définir un identificateur de type `_foo`, toute utilisation d'une facilité fournie par un header peut déclencher un « undefined behaviour » (par exemple, l'ordinateur utilise spontanément son modem pour téléphoner à la belle-mère du programmeur et l'inviter à venir dîner à la maison).
– Le `extern void _g(void) ;` explicite le fait que la restriction est sur le scope et pas le linkage. Bien entendu, il faut que la fonction `_g()` soit définie quelque part, avec un external linkage, ce qui ne peut pas se faire en C standard. Donc cette déclaration, quoique valide, doit avoir un pendant dans une autre translation unit, qui ne peut pas être fabriqué de façon standard. Pour compléter, rajoutons que la définition n'a besoin d'exister que si on se sert effectivement de la fonction. Donc on est en fait autorisé à faire une déclaration inutile qui pourrait faire planter des implémentations non conformes mais courantes. How useful.

– Quand bien même on aurait le droit, rares sont les implémentations complètement conformes de ce point de vue. Par pur pragmatisme, on évite donc de jouer avec des identificateurs commençant par un '_' suivi d'une lettre minuscule.
– Chaque header apporte ses propres restrictions ; par exemple, `<ctype.h>` peut déclarer n'importe quel identificateur qui commence par "is" ou "to" suivi d'une lettre minuscule. Ces identificateurs sont réservés pour ce qui est de l'external linkage, ce qui veut dire que même si on n'inclut pas `<ctype.h>`, on ne doit pas définir, entre autres, de variable globale "iszoinx" qui ne soit pas protégé par le mot clé `static`.

16.6 À quoi peut servir un cast en `(void)` ?

Il y a principalement deux utilités à caster une expression en `(void)`.

La première utilisation est pour indiquer explicitement au compilateur qu'une valeur est ignorée, comme au retour d'une fonction. Par exemple, il arrive souvent d'utiliser la fonction `printf` sans utiliser ni même tester la valeur de retour. Ecrire l'appel à `printf` ainsi

```
(void)printf("%s\n", "Un message à la con") ;
```

est une manière de dire au compilateur, et aux lecteurs du code, que je sais que `printf` renvoie une valeur, mais que j'ai décidé de l'ignorer. Cela peut être utile pour des utilitaires de vérification de code, comme *lint*.

La seconde utilisation est dans des définitions de macro. Voici un exemple :

```
#undef the_truc
#ifdef __TRUC__DEPENDANT__
#  define the_truc(a) ((void)0)
#else
#  define the_truc(a) /* du code */
#endif
```

Ainsi, `the_truc(a)` est utilisable là ou une expression est requise, comme ici :

```
i = (the_truc(a), 5) ;
```

Avec une définition de `the_truc` comme ceci,

```
#  define the_truc(a)
```

il y aurait une erreur à la compilation.

Chapitre 17

En guise de conclusion

```
--- Hou là...
--- J'ai écrit ça moi ?
--- Toujours est-il que « quelqu'un » l'a écrit.
    Tant pis, maintenant tu écris la doc'.
--- Oh non !
--- Bon alors tu débogues.
--- Ca va, ça va, j'écris la doc' ...
--- OK, ensuite tu débogues.
--- ...
```

Index