Un livre de Wikilivres.

Programmation C

Nous vous proposons, par ce livre, d'apprendre la programmation en langage C. Dans ce livre, nous supposerons que le lecteur a des notions de base en programmation, algorithmique et structures de données ; si vous n'en avez pas ou si vous voulez réviser des notions pendant l'apprentissage du C, vous pouvez consulter les Wikilivres associés : Programmation, Algorithmique et Structures de données.

Le langage C permet d'appréhender des principes informatiques de bas niveau, liés à l'architecture d'un ordinateur, comme la gestion de la mémoire. Si la connaissance de ces principes peut aider à comprendre certains côtés du langage C, ils ne sont cependant pas nécessaires pour lire ce Wikilivre, les concepts étant expliqués chaque fois que cela est nécessaire.

Ce livre, en constante évolution, est issu d'un projet collaboratif.

Bonne lecture !

> **Remarque :** *Note à l'attention des lecteurs de la version imprimable*
>
> Pour des raisons techniques, les notes de bas de pages sont rassemblées en fin d'ouvrage dans la version imprimable.

Voici une description rapide des différents chapitres qui constituent ce livre :

Dans cette partie, nous étudierons les concepts de base du langage C, sa syntaxe ainsi que la manière dont un programme est structuré.

Types de base

Ensuite, nous présenterons les différents types de base que fournit le langage C. Ces types étant pour l'essentiel des nombres (entiers, réels et complexes), ce chapitre traitera aussi de l'arithmétique associée à ces types. Les types caractères, qui sont la base de nombreux traitements, et ont un rôle particulier dans ce langage, seront aussi définis.

Classe de stockage

Le C permet de construire des constantes, par le biais de *classes de stockages*. Celles-ci permettent de définir plus généralement des types dérivés des types de base et seront étudiés dans ce chapitre.

Opérateurs

Nous étudierons ensuite les opérateurs permettant de travailler sur les types de base, ainsi que les règles générales qui structurent les expressions en C.

Structures de contrôle - tests

La notion d'expression définie, nous aborderons les moyens de contrôler le flux d'exécution d'un programme. Tout d'abord, les tests de type *Si...Alors...Fin Si* seront étudiés.

Structures de contrôle - itérations

Dans la suite du chapitre précédent, nous étudierons les structure d'itérations, ou boucles, de type *Pour*, *Tant que* ou dérivés.

Fonctions et procédures

Comment le C prend en charge les fonctions et procédures.

Tableaux

Comment le C gère les tables de données.

Pointeurs

Les pointeurs sont une spécificité du C, souvent mal comprise. Ce chapitre définira et montrera comment bien utiliser les pointeurs, ainsi que les erreurs à éviter dans leur manipulation.

Types avancés - structures, unions, énumérations

Explications sur les types avancés du C.

Préprocesseur

Le préprocesseur est un « langage » au-dessus du C, que l'utilisateur rencontre pour la première fois dans l'utilisation des directives d'inclusion d'en-têtes de la bibliothèque standard. Cependant, il est capable de bien plus, et fournit des outils comme les macros ou l'inclusion conditionnelle, qui sont l'objet de ce chapitre.

Bibliothèque standard

Cette partie a pour but de familiariser le lecteur avec la « bibliothèque standard » du C. Celle-ci propose un ensemble de fonctions, mais aussi de types ou de variables permettant d'effectuer des opérations spécifiques, comme les traitements de fichiers, les calculs mathématiques, ou la gestion des chaînes des caractères.

Chaînes de caractères

Ce chapitre traite de toutes les fonctions permettant d'effectuer des traitements sur les chaînes de caractères : concaténation, copie, recherche de caractères, etc.

Entrées/sorties

Ce chapitre traite des interactions avec l'utilisateur.

Erreurs

La bibliothèque standard utilise un mécanisme unique pour la gestion d'erreur, qui permet à un programme de déterminer précisément la cause de l'échec d'une fonction de la bibliothèque standard.

Mathématiques

Ce chapitre détaille les fonctions mathématiques fournies par la bibliothèque standard.

Gestion de la mémoire

La bibliothèque standard fournit un moyen d'allouer et de libérer dynamiquement de la mémoire au cours de l'exécution du programme. Cette fonctionnalité est surtout utilisée pour résoudre des problèmes dont le volume des données à traiter varie d'une exécution à l'autre.

Gestion des signaux

Un signal est une interruption logicielle reçue par un processus. Ces interruptions fournissent un moyen de communication entre le système et un programme, pour informer le programme de situations d'erreur (utilisation d'une zone de mémoire invalide, par exemple), ou informer un processus d'événements asynchrones (fin d'une opération d'entrée/sortie). Le C permet d'envoyer et de traiter la réception de ces signaux.

Conclusion

Introduction

Historique

Naissance

Le langage C a été inventé aux Bells Labs en 1972 par Dennis Ritchie pour permettre l'écriture du système d'exploitation UNIX, alors développé par Ken Thompson et Dennis Ritchie.

Le système d'exploitation UNIX, né à la fin des années 1960 - début des années 1970, a été écrit directement en langage assembleur pour les machines auxquelles il était destiné. Si le langage assembleur permettait d'écrire un tel système, il n'en était pas moins peu aisé à utiliser. Un tel langage est en effet particulier à un type de processeur, ce qui fait que tout le système devait être réécrit pour le faire fonctionner sur une nouvelle architecture. Cela fait que son principal créateur, Ken Thompson, souhaita rapidement utiliser un langage plus évolué pour réécrire UNIX.

Parmi les langages disponibles à l'époque, BCPL (pour *Basic Combined Programming Language*, qui est une simplification de CPL), créé par Martin Richards en 1966, était intéressant. Sans entrer dans des descriptions détaillées, BCPL est un langage simple, procédural, et non typé. Sa simplicité permettait de créer facilement des compilateurs BCPL sur les machines de l'époque, où les ressources étaient très limitées (le premier ordinateur utilisé par Ken Thompson pour lancer Unix était un PDP-7, qui disposait d'une mémoire de 4000 mots de 18 bits, soit moins de 9 Ko). Ken Thompson l'a fait évoluer pour concevoir le langage B, qu'il a implémenté sur les premières machines UNIX. Cependant, certaines limitations du langage B ont fait qu'UNIX n'a pu être réécrit dans ce langage.

À partir de 1971, Dennis Ritchie fit évoluer B, pour répondre à ces problèmes. À l'image des programmeurs qui incrémentent les versions de leurs programmes, Ritchie « incrémenta » la lettre B pour appeler le nouveau langage C. Cette évolution se « stabilisa » vers 1973, année à partir de laquelle UNIX et les utilitaires systèmes d'UNIX ont été réécrits avec succès en C.

Développement

Par la suite en 1978, Brian W. Kernighan documenta très activement le langage, pour finalement publier avec Ritchie le livre de référence *The C Programming Language*. On appelle souvent **C K&R** le langage tel que spécifié dans la première édition de ce livre.

Dans les années qui suivirent, le langage C fut porté sur de nombreuses autres machines. Ces portages ont souvent été faits, au début, à partir du compilateur *pcc* de Steve Johnson, mais par la suite des compilateurs originaux furent développés indépendamment. Durant ces années, chaque compilateur C fut écrit en suivant les spécifications du K&R, mais certains ajoutaient des *extensions*, comme des types de données ou des fonctions supplémentaires, ou interprétaient différemment certaines parties du livre (qui n'était pas forcément très précis). À cause de cela, il fut de moins en moins facile d'écrire des programmes en C qui puissent fonctionner tels quels sur un grand nombre d'architectures.

Normalisation

Pour résoudre ce problème, et ajouter au C des possibilités dont le besoin se faisait sentir (et déjà existantes dans certains compilateurs), en 1989, l'organisme national de normalisation des USA (ANSI) normalisa le C. Le nom exact de cette norme est **ANSI X3.159-1989** ***Programming Language C***, mais elle fut (et est toujours) connue sous les dénominations *ANSI C* ou *C89*. Puis l'ANSI soumit cette norme à l'Organisation internationale de normalisation (ou ISO), qui l'accepta telle quelle et la publia l'année suivante sous le numéro **ISO/IEC 9899:1990** (le document étant connu sous le nom *ISO C90*, ou *C90*). Le besoin de normalisation de l'époque étant extrêmement pressant, pratiquement tous les éditeurs de compilateurs C de l'époque se sont mis à modifier leurs compilateurs pour supporter la norme (et même n'ont pas attendu la publication officielle pour lancer ces modifications), et de fait la

quasi totalité des implémentations existant à ce jour comprennent ces normes. L'ISO publia dans les années qui suivirent deux ensembles de correctifs, et un *amendement* pour la gestion des caractères internationaux.

Étant donnée la grande diversité des implémentations existants au moment de l'élaboration de la norme, et le principe de négociation qui base les processus de normalisation, la norme C est un compromis. Son but était principalement double : d'une part assurer le plus possible la portabilité du code C, pour simplifier le portage de programmes C d'une implémentation à une autre (ce qui était souvent un cauchemar à réaliser), d'autre part donner une certaine liberté aux éditeurs pour proposer des extensions spécifiques. Pour ce faire, la norme a défini des *niveaux de conformité* pour les programmes C et les implémentations, allant du programme *strictement conforme*, que toute implémentation doit accepter, et qui fonctionnera exactement de la même manière, aux programmes dépendant d'extensions.

On peut noter que le livre de Kernighan et Ritchie a été republié dans une seconde édition, pour refléter les changements du C89.

Enfin, en 1999, l'organisme ISO proposa une nouvelle version de la norme, qui reprenait quelques bonnes idées du langage C++ (voir plus bas). Il ajouta aussi le type `long long` d'une taille minimale de 64 bits, les types complexes, l'initialisation des structures avec des champs nommés, parmi les modifications les plus visibles. Le nouveau document, qui au niveau de l'ISO est celui ayant autorité aujourd'hui, est **ISO/IEC 9899:1999**, connu sous le sigle *C99*. Deux ensembles de correctifs à cette version ont été publiés jusqu'à présent. Au sens strict, la dénomination *ISO C* correspond donc actuellement à la norme de 1999 corrigée, mais l'usage est tel qu'il est préférable de toujours préciser de quelle version on parle (89/90 ou 99), et c'est ce qui sera fait dans cet ouvrage lorsque la distinction sera nécessaire. Il est à noter que, si la norme C90 a été largement adoptée par les éditeurs de compilateurs C, très peu implémentent la version C99. En effet, le besoin des utilisateurs s'est fait moins pressant, et l'effort nécessaire pour rendre les compilateurs conformes a rarement été mené à terme.

L'activité du groupe de travail de la norme C est, depuis, portée plus sur la « maintenance » du langage qu'à de véritables « évolutions ». La prochaine version, pour l'instant dénommée officieusement *C1X* et **ISO/IEC 9899:201x** officiellement, n'est pas encore publiée et ne sera pas étudiée dans ce livre[1].

La norme C spécifie la syntaxe du langage ainsi qu'une bibliothèque de fonctions simple. Cette bibliothèque est moins fournie que dans d'autres langages, mais cela est dû au domaine d'utilisation très varié de ce langage. En particulier, pour assurer une portabilité maximale du langage entre les implémentations embarquées, les ordinateurs de type PC et les supercalculateurs (par exemple), certaines fonctions n'ont pas été acceptées dans la norme. On peut citer, entre autres, la programmation parallèle, les communications entre processus, la communication réseau ou les interfaces graphiques...

Cela fait qu'en parallèle de cette normalisation du langage C, certaines extensions ont elles aussi été standardisées, voire normalisées. Ainsi, par exemple, des fonctions spécifiques aux systèmes UNIX, sur lesquels ce langage est toujours très populaire, et qui n'ont pas été intégrées dans la norme du langage C, ont servi à définir une partie de la norme POSIX.

Dans ce livre, nous étudierons surtout le langage C tel que défini dans les normes ISO citées ci-dessus. La norme C99 n'étant pas implémentée par tous les compilateurs, et beaucoup de programmes existants étant développés en C90, les différences entre les deux normes seront indiquées à chaque fois que nécessaire. Toutefois, nous illustrerons certains chapitres avec des exemples d'extensions courantes qu'un développeur C peut rencontrer, en particulier concernant les fonctions qui ne sont pas fournies par la norme C elle-même. Mais ces exemples d'extensions seront mineurs. D'autres Wikilivres étudient plus en détails de telles extensions, vous pouvez les trouver en regardant les livres listés sur la page Catégorie:C.

Norme et documents de référence

La norme qui définit le langage C est élaborée au niveau international par un groupe de travail (*working group*) de l'ISO. Ce groupe de travail fait partie plus précisément du comité technique commun à l'ISO et à la Commission Electrotechnique Internationale, le Joint Technical Commitee 1, qui élabore les normes internationales concernant les technologies de l'information. Il se décompose en sous-comités, donc le SC22, responsable des langages informatiques. Le SC22 rassemble plusieurs groupes de travail, chacun portant sur un langage spécifique, ou sur des

considérations indépendantes des langages. On peut ainsi citer, par exemple :

- JTC1/SC22/WG9 pour Ada ;
- JTC1/SC22/WG14 pour le C ;
- JTC1/SC22/WG21 pour le C++.

Comme pour toute norme internationale, c'est l'ISO qui décide des modalités de sa distribution. Comme la grande majorité des normes ISO, celle-ci n'est pas en libre distribution, mais en vente[2]. Le lecteur désirant acheter un exemplaire de la norme peut le faire sur le site officiel de l'ISO, par exemple.

Les corrections, ou questions, concernant la norme sont diffusées individuellement par le WG14 sur leur site sous le nom de *Defect Report* (DR), et rassemblées pour publication officielle dans des *rectificatifs techniques* ou *Technical Corrigenda* (TC). Deux ont été publiés pour le C99, en 2001 et 2004.

Par ailleurs, les *brouillons* (ou *drafts*) de la norme appartiennent au groupe de travail, qui peut décider de les publier. C'est le cas des documents n869, qui est le dernier brouillon disponible avant la publication de C99, et n1124, correspondant au C99 auquel ont été ajoutés les TC1 et TC2 (le TC3 ayant été publié en 2007).

On pourra noter que, tout comme l'ANSI l'a fait pour le C89, le WG14 a publié un *Rationale*, qui est un commentaire de la norme C99. Ces deux textes (en anglais, toujours) peuvent être intéressants pour comprendre les motivations de certains choix dans l'évolution du langage.

Tous ces documents sont disponibles sur le site officiel du groupe de travail (les URLs et références se trouvent dans la bibliographie à la fin de ce livre).

Toutefois, il faut bien noter qu'une norme est un texte très technique, qui sert de référence, et est très loin d'être un texte pédagogique. Un débutant fera mieux de lire un ouvrage comme ce wikilivre pour apprendre le C que lire directement la norme, au risque d'être rapidement découragé. Le présent ouvrage est construit de manière à ne pas avoir à recourir au texte de la norme, et n'y fera référence que rarement, mais le lecteur intéressé et averti pourra y vérifier des points complexes qui ne seront pas détaillés ici.

C et C++

Ce livre ne porte pas sur le C++, mais ces deux langages étant d'apparence très proches, il convient, pour éviter la confusion au lecteur désireux d'apprendre l'un ou l'autre, de préciser qu'ils sont très différents et que, si la connaissance de l'un peut aider à l'apprentissage de l'autre, *ils ne doivent pas être confondus*. L'idée du langage qui est devenu C++ a été lancée par Bjarne Stroustrup en 1979, dans un souci de faire évoluer le C de manière à le rendre plus « robuste ». L'apport le plus visible de cette évolution est que le C++ est orienté objet, alors que le C est procédural, mais ce n'est pas le seul.

Le langage C++ s'est créé *à partir* du C puis, rapidement, *en parallèle* à lui, et en 1998 est devenu une norme ISO au même titre que le C. Les groupes de travail respectifs du C et du C++ communiquent régulièrement dans le but d'assurer une certaine compatibilité entre les deux langages. De fait, un certain nombre d'idées issues des réflexions autour du C++ se sont vues intégrées dans la norme C99[3]. Cependant, la croyance populaire selon laquelle le C est un sous-ensemble du C++, ou qu'utiliser un compilateur C++ pour un programme C ne pose pas de problème, est fausse, et à plus d'un titre :

- un code source strictement conforme du point de vue du C peut être invalide au sens du C++[4] ;
- un code source qui est à la fois strictement conforme pour le C et le C++ peut avoir un comportement différent dans ces deux langages[5].

Durant l'apprentissage du C, le lecteur devra donc faire attention. Un certain nombre d'éditeurs, mettant à profit la grande proximité des deux langages, distribuent ensemble un compilateur C et un compilateur C++, en appelant parfois l'ensemble *C/C++*, ce qui entraîne une grande confusion chez les utilisateurs, qui ont parfois du mal à savoir quel est le langage dans lequel ils travaillent. L'utilisateur doit donc apprendre comment fonctionne son implémentation, pour déterminer quel(s) langage(s), norme(s) et extension(s) elle supporte, et comment le faire fonctionner dans l'un ou l'autre mode.

Une liste de compilateurs, avec les supports des normes, est référencée en Bibliographie.

Pourquoi apprendre le langage C ?

Avec son approche bas-niveau, le C permet d'obtenir des programmes très optimisés, pratiquement autant que s'ils avaient été écrits directement en assembleur. Avec un peu d'effort, il est même possible d'utiliser une approche orientée objet, au prix d'une certaine rigueur que le langage et les compilateurs, dans une certaine mesure, sont très loin d'imposer.

Les systèmes d'exploitation pour ordinateur de bureau les plus répandus actuellement sont Windows de Microsoft, Mac OS X d'Apple, et GNU/Linux. Ils sont tous trois écrits en langage C. Pourquoi ? Parce que les systèmes d'exploitation tournent directement au dessus du matériel de la machine. Il n'y a pas de couche plus basse pour gérer leurs requêtes. À l'origine, les systèmes d'exploitation étaient écrits en assembleur, ce qui les rendait rapides et performants. Toutefois, écrire un OS en assembleur est une tâche pénible et cela produit du code qui ne peut s'exécuter que sur une seule architecture processeur, comme l'Intel X86 ou l'AMD 64. Écrire un OS dans un langage de plus haut niveau, comme le langage C, permet au programmeur de porter son système d'exploitation sur une autre architecture sans avoir à tout réécrire.

Mais pourquoi utiliser le C et non Java, Basic ou Perl ? Principalement à cause de la gestion de la mémoire. À la différence de la plupart des autres langages de programmation, le langage C permet au programmeur de gérer la mémoire de la manière qu'il aurait choisie s'il avait utilisé l'assembleur. Les langages comme le Java et le Perl permettent au programmeur de ne pas avoir à se soucier de l'allocation de la mémoire et des pointeurs. C'est en général un point positif, car il est assez pénible et inutile de devoir se soucier de la gestion de la mémoire lorsqu'on écrit un programme de haut niveau comme un rapport sur les résultats trimestriels.

Kenneth Thompson (à gauche) et Dennis Ritchie (à droite), les créateurs du langage C

Cependant lorsqu'on parle d'écrire un programme de bas niveau comme la partie du système d'exploitation qui s'occupe d'envoyer la chaîne d'octets correspondant à notre rapport trimestriel depuis la mémoire de l'ordinateur vers le *buffer* de la carte réseau afin de l'envoyer vers une imprimante réseau, avoir un accès direct à la mémoire est fondamental — ce qui est impossible à faire dans un langage comme Java par exemple. Les compilateurs C produisent de plus très souvent un code rapide et performant.

Est-ce vraiment si merveilleux que le langage C soit un langage si répandu ? Par effet domino, la génération suivante de programmes suit la tendance de ses ancêtres. Les systèmes d'exploitation écrits en C ont toujours des librairies écrites en C. Ces bibliothèques système sont à leur tour utilisées pour écrire des bibliothèques de plus haut niveau (comme OpenGL ou GTK) et le programmeur de ces bibliothèques décide souvent d'utiliser le même langage que celui utilisé par ces bibliothèques système. Les développeurs d'applications utilisent ces bibliothèques de haut niveau pour écrire des traitements de texte, des jeux, les lecteurs multimédia, etc... La plupart d'entre eux choisiront d'utiliser pour leur programme le même langage que les bibliothèques de haut niveau. Et le schéma se reproduit à l'infini...

Bases du langage

Bonjour !

L'un des plus petits programmes possible en langage C est :

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

(Les numéros de lignes ont été ajoutés pour la lisibilité, mais ne font pas partie du programme.)

Ce programme a pour seul but d'afficher le texte « Bonjour ! » suivi d'un retour à la ligne. Voici la description de chaque ligne qui le compose :

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

Inclusion de l'en-tête nommé `<stdio.h>`. Il est lié à la gestion des entrées et sorties (*STanDard Input Output*, entrées/sorties standards) et contient, entre autres, la déclaration de la fonction `printf` permettant d'afficher du texte formaté.

```
3.  int main(void)
```

Définition de la fonction principale du programme, `main` (*principal* en anglais). Il s'agit de la fonction appelée au démarrage du programme, tout programme en C doit donc la définir. La partie entre parenthèses spécifie les paramètres que reçoit le programme : ici, le programme n'utilisera aucun paramètre, ce qui est spécifié en C avec le mot-clé `void` (*vide*). La partie à gauche du nom de la fonction `main` spécifie le type renvoyé par la fonction : la fonction `main` est définie par la norme C comme renvoyant une valeur entière, de type `int` (pour *integer*, entier), au système d'exploitation.

```
4.  {
```

Début de la définition de la fonction `main`.

```
5.      printf("Bonjour !\n");
```

Appel de la fonction `printf` (*print formatted*, affichage formaté, qui sera expliquée dans le chapitre dédié à cette fonction). La chaîne « Bonjour ! » va être affichée, elle sera suivie d'un retour à la ligne, représenté en C par `\n`.

```
6.      return 0;
```

La fonction `main` a été définie comme retournant (*to return*) une valeur de type `int`, on renvoie donc une telle valeur. Par convention, la valeur 0 indique au système d'exploitation que le programme s'est terminé normalement.

```
7.  }
```

Fin de la définition de la fonction `main`.

Ce premier exemple ne détaille évidemment pas tout ce qui concerne la fonction `main`, ou `printf`, par exemple. La suite de l'ouvrage précisera les points nécessaires dans les chapitres appropriés (voir par exemple le paragraphe *la fonction main* du chapitre *Fonctions et procédures*, ou le paragraphe *Sorties formatées* du chapitre *Entrées/sorties*).

Compilation

Pour exécuter un programme C, il faut préalablement le compiler[?].

Tapez le code source du programme *Bonjour !* dans un éditeur de texte[?] et sauvegardez-le sous le nom `bonjour.c` (l'extension `.c` n'est pas obligatoire, mais est usuelle pour un fichier source en C). Ouvrez une fenêtre de commandes (terminal sous Unix, commandes MS-DOS sous Windows), et placez-vous dans le répertoire où est sauvegardé votre fichier. Pour compiler avec le compilateur standard de votre système, il faut taper la commande :

```
cc bonjour.c
```

D'autres compilateurs pour le langage C peuvent être utilisés. La compilation du programme produira un fichier exécutable, appelé `a.out` sous Unix, et `a.exe` sous Windows. Il peut être exécuté en tapant `./a.out` sous Unix, et `a` sous Windows.

Éléments de syntaxe

Identificateurs

Les identificateurs commencent par une lettre ou le caractère souligné ("_") et peuvent contenir des lettres, des chiffres et le caractère souligné (cependant, les identificateurs commençant par deux caractères soulignés, ou un caractère souligné suivi d'une majuscule sont réservés par l'implémentation, et ne doivent pas être utilisés dans un programme « ordinaire »). **Tous** les mots-clés ainsi que les symboles (variables, fonctions, champs, etc.) sont **sensibles à la casse des lettres**. Quelques exemples d'identificateurs valides :

```
toto
_coin_coin76
```

Mots réservés du langage

Le langage, dans la norme C90, possède 32 mots réservés (ou mots-clés) qui ne peuvent pas être utilisés comme identificateurs. La norme C99 en a ajouté cinq ; la norme C11 en a de nouveau apporté 7. Le tableau ci-dessous les liste tous, avec la mention *« C99 »* ou *« C11 »* pour indiquer ceux définis uniquement dans les versions correspondantes ou ultérieures :

auto	break	case	char	const	continue	default	do
double	else	enum	extern	float	for	goto	if
inline *(C99)*	int	long	register	restrict *(C99)*	return	short	signed
sizeof	static	struct	switch	typedef	union	unsigned	void
volatile	while	_Alignas *(C11)*	_Alignof *(C11)*	_Atomic *(C11)*	_Bool *(C99)*	_Complex *(C99)*	_Generic *(C11)*
_Imaginary *(C99)*	_Noreturn *(C11)*	_Static_assert *(C11)*	_Thread_local *(C11)*				

Un programme C90 peut par exemple définir `inline` comme un identifiant de variable ou de fonction mais cela est très déconseillé, car il sera invalide au sens de la norme C99. Une mise à jour du compilateur peut alors l'empêcher de compiler le code source en question. Pour éviter de telles situations, il est préférable de considérer *tous* les mots-clés du tableau précédent comme étant réservés, quels que soient le compilateur et la norme utilisés.

Commentaires

Les commentaires commencent par `/*` et se terminent par `*/`, ils ne peuvent pas être imbriquées :

```
/* ceci est un commentaire */
/*
 ceci est aussi un commentaire
 */
/* /* ceci est encore un commentaire */
```

⚠ Ce code contient **une erreur volontaire** !

```
/* /* */ ceci n'est pas un commentaire */
```

Inclure des commentaires pertinents dans un programme est un art subtil. Cela nécessite un peu de pratique pour savoir guider les lecteurs et attirer leur attention sur certaines parties délicates du code. Pour faire court, on ne saurait trop que rappeler ce célèbre vers de Nicolas Boileau, qui disait que « ce que l'on conçoit bien, s'énonce clairement et les mots pour le dire arrivent aisément » (Art Poétique, 1674). Ou encore, pour adopter une approche plus pragmatique : si un programmeur a du mal à commenter (expliquer, *spécifier*) le fonctionnement d'un passage de code, c'est que le code est mal conçu et qu'il serait bénéfique de le réécrire plus lisiblement.

Attention! Les commentaires **ne devraient pas** être employés pour désactiver certaines parties du code. Cette technique dissuade un programmeur d'en inclure puisque le langage C ne permet pas d'imbriquer les commentaires. Pour cela, le préprocesseur dispose d'instructions dédiées, qui permettent heureusement de désactiver du code en laissant les commentaires.

Le `//` ajoute la possibilité de placer un commentaire d'une seule ligne, à partir du `//` jusqu'à la fin de la ligne :

```
// commentaire
instruction; // autre commentaire
```

Cependant, ces commentaires « à la C++ » sont apparus dans C99, et ne sont pas permis en C90.

Instructions

Les *instructions* se terminent par un point-virgule (`;`), on peut placer autant d'instructions que l'on veut sur une même ligne (même si ce n'est pas conseillé pour la lisibilité du code). Les *blocs d'instructions* commencent par une accolade ouvrante ({) et se terminent par une accolade fermante (}). **Les instructions doivent obligatoirement être déclarées dans une fonction** : il est impossible d'appeler une fonction pour initialiser une variable globale par exemple (contrairement au C++).

```
/* une instruction */
i = 1;

/* plusieurs instructions sur la même ligne */
i = 1; j = 2; printf("bonjour\n");

/* un bloc */
{
        int i;
        i = 5;
}

/* l'instruction vide */
;
```

Dans la mesure où le compilateur ne se soucie pas des blancs (espaces et retours à la ligne), vous pouvez formater votre code comme vous l'entendez. Il y a beaucoup de religions concernant les styles d'indentation, mais pour faire court et éviter les guerres saintes, on ne saurait trop conseiller que d'utiliser le même nombre de blancs par niveau d'imbrication de bloc.

À noter que l'instruction vide étant valide en C, on peut donc pratiquement rajouter autant de point-virgules que l'on veut. Le point-virgule ne sert pas uniquement à marquer la fin des instructions, les compilateurs l'utilisent généralement comme caractère de synchronisation, suite à une erreur dans un programme source. En fait, en général, lorsqu'une erreur est détectée, les compilateurs ignorent tout jusqu'au prochain point-virgule. Ce qui peut avoir des conséquences assez dramatiques, comme dans l'exemple suivant :

⚠ Ce code contient **une erreur volontaire** !

```
int traite_arguments(int nb, char * argv[])
{
        /* ... */
        return 0
}

int main(int nb, char * argv[])
{
        int retour;
        retour = traite_arguments(nb, argv);
        /* ... */
}
```

On notera l'absence de point-virgule à la fin de l'instruction `return` à la fin de la fonction `traite_arguments`. Ce que la plupart des compilateurs feront dans ce cas sera d'ignorer tout jusqu'au prochain point-virgule. On se rend compte du problème : on a sauté une déclaration de fonction (avec ses deux paramètres) et une déclaration de variable. Ce qui veut dire qu'une cascade d'erreurs va suivre suite à l'oubli... d'un seul caractère (`;`) !

Déclarations de variables

```
T var1, var2, ..., varN;
```

Cette ligne déclare les variables *var1*, *var2*, ..., *varN* de type *T*. Une variable est dite *locale*, si elle est définie à l'intérieur d'une fonction et *globale* si définie en-dehors.

Par exemple :

```
int jour; /* 'jour' est une variable globale, de type entier */

int main(void)
{
    double prix; /* 'prix' est une variable locale à la fonction 'main', de type réel */
    return 0;
}
```

Variables locales

Les variables locales (aussi appelées *automatiques*) ne sont visibles que dans le bloc dans lequel elles sont définies, et n'existent que durant ce bloc. Par exemple :

⚠ Ce code contient **une erreur volontaire** !

```
int fonction(int n)
{
    int i; /* i est visible dans toute la fonction */
    i = n + 1;
    {  /* début d'un nouveau bloc */
        int j; /* j est visible dans le nouveau bloc *seulement* */
        j = 2 * i; /* i est accessible ici */
    }
    i = j; /* ERREUR : j n'est plus accessible */
    return i;
}
```

Le code précédent accède à `i` depuis un bloc contenu dans le bloc où `i` a été défini, ce qui est normal. Puis il essaye d'accéder à la valeur de la variable `j` *en-dehors* du bloc où elle à été définie, ce qui est une erreur.

Les variables locales ne sont pas initialisées automatiquement et contiennent donc, après leur déclaration, une valeur aléatoire. Utiliser une telle valeur peut causer un comportement aléatoire du programme. Le programme suivant essaye d'afficher la valeur d'une telle variable, et on ne peut savoir à l'avance ce qu'il va afficher. Si on le lance plusieurs fois, il peut afficher plusieurs fois la même valeur aussi bien qu'il peut afficher une valeur différente à chaque exécution :

⚠ Ce code contient **une erreur volontaire** !

```c
#include <stdio.h>

int main(void)
{
    int n;
    printf("La variable n vaut %d\n", n);
    return 0;
}
```

Avant la normalisation ISO C99, les déclarations de variables locales devaient obligatoirement être placées juste au début d'un bloc et s'arrêtaient à la première instruction rencontrée. Si une déclaration est faite au-delà, une erreur sera retournée. Suivant la norme C99, les déclarations peuvent se trouver n'importe où dans un bloc. Par exemple, le code suivant est correct suivant la norme C99, mais pas suivant la norme C90, car la variable a est définie après l'instruction `puts("Bonjour !");` :

```c
int ma_fonction(int n)
{
    puts("Bonjour !");
    int a;  /* Valide en C99, invalide en C90 */
    /* autre chose... */
    return a;
}
```

Variables globales

Une attention particulière doit être portée aux variables globales, souvent source de confusion et d'erreur. Utilisez des noms explicites, longs si besoin et limitez leur usage au seul fichier où elles sont déclarées, en les déclarants statiques.

Les variables globales sont initialisées avant que la fonction `main` s'exécute. Si le programmeur ne fournit pas de valeur initiale explicitement, chaque variable reçoit une valeur par défaut suivant son type :

- un nombre (entier, réel ou complexe) est initialisé à 0 ;
- les pointeurs sont initialisés à `NULL` ;
- pour une structure, l'initialisation se fait récursivement, chaque membre étant initialisé suivant les mêmes règles ;
- pour une union, le premier membre est initialisé suivant ces règles.

Pour initialiser explicitement une variable globale, on ne peut utiliser que des constantes ; en particulier, on ne peut appeler de fonction, contrairement à d'autres langages.

⚠ Ce code contient **une erreur volontaire** !

```c
int jour_courant(void)
{
```

```
   /* retourne le n° du jour courant */
}

int jour = jour_courant(); /* ERREUR ! */

int main(void)
{
   /* ... */
   return 0;
}
```

Le code précédent ne pourra pas compiler, car on ne peut appeler directement `jour_courant()` pour initialiser `jour`. Si vous avez besoin d'un tel comportement, une solution est de faire l'initialisation explicite dans `main` :

```
int jour_courant(void)
{
   /* retourne le n° du jour courant */
}

int jour; /* 'jour' est initialisé à 0 *avant* le début de 'main' */

int main(void)
{
   jour = jour_courant();
   /* ... */
   return 0;
}
```

Traitements des ambiguïtés

Le C utilise un mécanisme élégant pour lever des constructions syntaxiques en apparence ambiguës. L'analyse des mots (lexèmes, *token* en anglais) se fait systématiquement de la gauche vers la droite. Si plusieurs lexèmes peuvent correspondre à une certaine position, le **plus grand** aura priorité. Considérez l'expression valide suivante :

```
a+++b;
```

Ce genre de construction hautement illisible et absconse est bien évidemment à éviter, mais illustre parfaitement bien ce mécanisme. Dans cet exemple, le premier lexème trouvé est bien sûr l'identificateur 'a' puis le compilateur a le choix entre l'opérateur unaire '++', ou l'opérateur binaire '+'. Le plus grand étant le premier, c'est celui-ci qui aura priorité. L'instruction se décompose donc en :

```
(a ++) + b;
```

Types de base

Le C est un langage typé statiquement : chaque variable, chaque constante et chaque expression, a un type défini à la compilation. Le langage lui-même fournit des types permettant de manipuler des nombres (entiers, réels ou complexes) ou des caractères (eux-mêmes étant manipulés comme des entiers spéciaux), et permet de construire des types plus complexes à partir de ces premiers, par exemple en groupant des données de même type en tableaux, ou des données de types différents dans des structures. Dans ce chapitre, nous étudierons les types de base fournis par le C, l'étude des types complexes étant faite dans la suite du livre.

Entiers

Il y a cinq types de variables entières (« integer » en anglais) :

- `char` ;
- `short int`, ou plus simplement `short` ;
- `int` ;
- `long int`, ou `long` ;
- `long long int`, ou `long long` (ce type a été ajouté depuis la norme C99).

Comme évoqué en introduction, le type caractère `char` est particulier, et sera étudié en détail plus bas.

Les types entiers peuvent prendre les modificateurs `signed` et `unsigned` qui permettent respectivement d'obtenir un type signé ou non signé. Ces modificateurs ne changent pas la taille des types. Le langage ne définit pas exactement leur tailles, mais définit un domaine de valeurs minimal pour chacun.

Représentation des entiers signés

La norme C tient compte des anciennes représentations des nombres signés telles que le **signe+valeur absolue** et le **complément à 1**. Ces deux représentations sont brièvement expliquées ci-dessous :

Signe + valeur absolue (SVA)
: Un bit contient le signe du nombre (par ex : 0 pour +, 1 pour -), les autres bits sont utilisés pour la valeur absolue. On peut donc représenter 0 de deux manières : +0 (000...000) ou -0 (100..000). Sur N bits on peut donc représenter tout nombre entre $-(2^{N-1}-1)$ *(111...111)* et $+(2^{N-1}-1)$ *(011...111)*.

Complément à 1 (CPL1)
: Les bits des nombres négatifs sont inversés. On peut donc représenter 0 de deux manières : +0 (000...000) ou -0 (111..111). Sur N bits on peut donc représenter tout nombre entre $-(2^{N-1}-1)$ *(100...000)* et $+(2^{N-1}-1)$ *(011...111)*.

Ces deux représentations peuvent représenter la valeur nulle de deux manières différentes.

La représentation moderne des nombres négatifs utilise le **complément à 2 (CPL2)** qui consiste à représenter les nombres négatifs comme le complément à 1 et en ajoutant 1. Sur un nombre fixe de bits, la valeur 0 n'a qu'une seule représentation : +0 (000...000) et -0 (111...111 + 1 = (1)000...000) ont deux représentations identiques. Sur N bits on peut donc représenter tout nombre entre $-(2^{N-1})$ *(100...000)* et $+(2^{N-1}-1)$ *(011...111)*. Cette représentation possède donc un domaine plus large.

Le tableau ci-dessous donne le domaine des valeurs quelle que soit la représentation utilisée (SVA, CPL1 ou CPL2) :

Domaines de valeurs minimaux des types entiers (C90 et C99) quelle que soit sa représentation (SVA, CPL1 ou CPL2)

Type	Taille	Borne inférieure	Borne inférieure (formule)	Borne supérieure	Borne supérieure (formule)
`signed char`	≥ 8 bits	-128	$-(2^7)$	+127	2^7-1
`unsigned char`	≥ 8 bits	0	0	+255	2^8-1
`short`	≥ 16 bits	-32 767	$-(2^{15}-1)$	+32 767	$2^{15}-1$
`unsigned short`	≥ 16 bits	0	0	+65 535	$2^{16}-1$
`int`	≥ 16 bits	-32 767	$-(2^{15}-1)$	+32 767	$2^{15}-1$
`unsigned int`	≥ 16 bits	0	0	+65 535	$2^{16}-1$
`long`	≥ 32 bits	-2 147 483 647	$-(2^{31}-1)$	+2 147 483 647	$2^{31}-1$
`unsigned long`	≥ 32 bits	0	0	+4 294 967 295	$2^{32}-1$
`long long` *(C99)*	≥ 64 bits	-9 223 372 036 854 775 807	$-(2^{63}-1)$	+9 223 372 036 854 775 807	$2^{63}-1$
`unsigned long long` *(C99)*	≥ 64 bits	0	0	+18 446 744 073 709 551 615	$2^{64}-1$

Cette table signifie qu'un programme peut utiliser sans problème une variable de type `int` pour stocker la valeur $2^{15}-1$, quel que soit le compilateur ou la machine sur laquelle va tourner le programme.

Par contre, une implémentation C *peut* fournir des domaines de valeurs plus larges que ceux indiqués au-dessus :

- Les domaines indiqués pour les nombres signés dans le tableau précédent sont ceux d'une implémentation par **complément à 1**, ou par **signe et valeur absolue**. Pour **le complément à 2**, la borne inférieure est de la forme -2^{N-1}, ce qui autorise une valeur supplémentaire (ex : `int` de -2^{15} à $+2^{15}-1$, soit de -32 768 à +32 767),
- un `int` implémenté sur 32 bits pourrait aller de $-(2^{31}-1)$ à $2^{31}-1$, par exemple[6], et un programme tournant sur une telle implémentation peut alors utiliser ces valeurs, mais il perdrait en portabilité.

Domaines de valeurs des types entiers représentés en complément à 2 (CPL2)

Type	Borne inférieure	Borne inférieure (formule)	Borne supérieure	Borne supérieure (formule)
`signed char`	-128	$-(2^7)$	+127	2^7-1
`unsigned char`	0	0	+255	2^8-1
`short`	-32 768	$-(2^{15})$	+32 767	$2^{15}-1$
`unsigned short`	0	0	+65 535	$2^{16}-1$
`int`	-32 768	$-(2^{15})$	+32 767	$2^{15}-1$
`unsigned int`	0	0	+65 535	$2^{16}-1$
`long`	-2 147 483 648	$-(2^{31})$	+2 147 483 647	$2^{31}-1$
`unsigned long`	0	0	+4 294 967 295	$2^{32}-1$
`long long` *(C99)*	-9 223 372 036 854 775 808	$-(2^{63})$	+9 223 372 036 854 775 807	$2^{63}-1$
`unsigned long long` *(C99)*	0	0	+18 446 744 073 709 551 615	$2^{64}-1$

Par ailleurs, une relation d'ordre entre ces domaines de valeurs est garantie ; qui peut être exprimée ainsi :

```
domaine(char) ≤ domaine(short) ≤ domaine(int) ≤ domaine(long) ≤ domaine(long lon
```

Cela signifie que toutes les valeurs possibles pour une variable du type `char` sont aussi utilisables pour les autres types ; mais aussi que, par exemple, une valeur valide pour le type `int` peut ne pas être représentable dans une variable de type `short`.

Si vous ne savez pas quel type donner à une variable de type entier, le type `int` est par défaut le meilleur choix (à condition que votre donnée ne dépasse pas $2^{15}-1$) : ce type est la plupart du temps représenté au niveau matériel par un « mot machine », c'est-à-dire qu'il est adapté à la taille que la machine peut traiter directement (il fait usuellement 32 bits sur un PC 32 bits, par exemple). Cela permet un traitement plus rapide par le matériel. De plus, beaucoup de bibliothèques (que ce soit celle fournie par le langage C ou d'autres) utilisent ce type pour passer des entiers, ce qui fait que l'utilisation de ces bibliothèques sera plus aisée.

Par ailleurs, un utilisateur peut connaître les domaines de valeurs exacts de sa machine en utilisant l'en-tête `<limits.h>`.

Portabilité apportée par C99

L'incertitude sur l'intervalle de valeur de chaque type en fonction de la machine peut s'avérer extrêmement gênante, pour ne pas dire rédhibitoire. En effet, certains programmes peuvent nécessiter un type de données de taille fixe et cependant être destinés à être portables. Pour ces programmes, les types entiers du C ne sont pas suffisants. *Beaucoup* d'extensions ont été rajoutées pour définir explicitement des types entiers à intervalle fixe (8, 16, 32 bits...) à partir des types de base, avec une nomenclature loin d'être homogène d'un compilateur à l'autre (ce qui, loin de résoudre le problème, ne faisait que le déplacer).

La norme ISO C99 décide une bonne fois pour toute de définir, dans l'en-tête <stdint.h>, plusieurs nouveaux types où *N* représente un nombre entier définissant la taille requise en bit :

- des types implémentés optionnellement sur certaines architectures (à éviter ?) ;
  - entiers signés ou non et de longueur *N* exacte : `uint`*N*`_t` et `int`*N*`_t` ;
  - entiers pouvant contenir un pointeur : `intptr_t` et `uintptr_t` ;
- des types requis par toutes les architectures respectant la norme C99 ;

- entiers devant être plus grand que *N* bits au moins : `int_least`*N*`_t` et `uint_least`*N*`_t` ;
- entiers rapides à calculer et plus grand que *N* bits au moins : `int_fast`*N*`_t` et `uint_fast`*N*`_t` ;
- plus grand entier : `intmax_t` et `uintmax_t`.

Cet en-tête définit aussi des constantes pour les valeurs minimales et maximales de chaque type.

L'include <inttypes.h> définit les constantes symboliques à utiliser pour imprimer ces nouveaux types avec les fonctions de la famille de printf (PRIxxx) et les lire avec celles de scanf (SCNxxx).

Constantes numériques entières

Il existe différentes suites de caractères qui sont reconnues comme étant des constantes numériques entières :

- un nombre en notation décimale : une suite de chiffres (0-9) ;
- le caractère « 0 » suivi d'un nombre en notation octale : une suite de chiffres compris entre 0 et 7 ;
- les caractères « 0x » suivi d'un nombre en notation hexadécimale : une suite de chiffres et des lettres a, b, c, d, e, f (ou A, B, C, D, E, F).

Par défaut, une constante numérique entière est de type `int` et, si sa valeur est trop grande pour le type `int`, elle prend celle du type « plus grand » suffisant. Comme les domaines de valeurs des types peuvent varier suivant la machine, le type effectif d'une constante peut lui aussi varier. Cela peut s'avérer problématique lors de passage de paramètres à des fonctions à nombre variable d'arguments, par exemple. À cause de cela, il est recommandé de forcer le type de la constante en le **postfixant** des attributs suivants :

- `U` : la constante est non-signée (voir la section promotion pour comprendre les implications) ;
- `L` : la constante est de type `long` au lieu de `int` ;
- `LL` est une nouveauté C99 pour les constantes de type `long long`.

`U` peut être combiné à `L` et `LL` pour obtenir les types `unsigned long` et `unsigned long long`, respectivement. Lorsqu'une constante est suffixée, mais que sa valeur est trop grande pour le type demandé, le même processus de recherche de type « assez grand » est utilisé[7].

Débordement

Sur une machine donnée, un type entier a un domaine de valeurs fixe. Considérons qu'on travaille sur un PC en 32 bits, en complément à deux : sur un tel ordinateur, le type `int` varie souvent de -2^{31} à $2^{31}-1$. Cela permet de manipuler sans problème des valeurs dans ce domaine. Par contre, si on utilise des valeurs hors du domaine, par exemple 2^{32}, et qu'on essaye de la stocker dans une variable de type `int` sur une telle machine, que se passe-t-il ?

La réponse dépend du type:

- Si on essaye d'enregistrer une valeur hors domaine dans une variable de type *signé* (comme dans l'exemple), la conversion n'est pas définie par le langage. Cela signifie que tout peut arriver.
- Si on essaye d'enregistrer une valeur hors domaine dans une variable de type *non signé* (`unsigned long`, par exemple), la conversion se fait modulo la valeur maximale représentable par ce type + 1.

Exemples : On suppose que le type `unsigned char` est codé sur 8 bits. Alors une valeur de ce type peut aller de 0 à 2^8-1, soit 255. Par la règle précédente, les conversions se feront donc modulo 255 + 1, soit 256. On considère le programme suivant :

```
#include <stdio.h>

int main(void)
{
    unsigned char c = 300;
    /* %hhu sert à dire à printf() qu'on veut afficher un unsigned char */
    printf("La variable c vaut %hhu.\n", c);
    c = -5;
    printf("La variable c vaut %hhu.\n", c);
    return 0;
```

```
}
```

Le résultat du programme est :

```
La variable c vaut 44.
La variable c vaut 251.
```

En effet, on a 300 - 256 = 44 et -5 + 256 = 251.

De même :

```
#include <stdio.h>

int main(void)
{
    signed char c = 300;
    /* %hhd sert à dire à printf() qu'on veut afficher un signed char */
    printf("La variable c vaut %hhd.\n", c);
    c = -300;
    printf("La variable c vaut %hhd.\n", c);
    return 0;
}
```

Le résultat du programme peut alors être :

```
La variable c vaut 44.
La variable c vaut -44.
```

Sur une telle machine, les types signés sont traités de la même manière que les types non signés.

Si vous essayez de compiler ces deux programmes, votre compilateur pourra détecter les débordements et vous en avertir (GCC le fait).

Réels

Les nombres réels ne pouvant tous être représentés, sont approximés par des nombres à virgule flottante. Comme dans le cas des entiers, il existe plusieurs types de nombre à virgule flottante. En voici la liste triée par précision croissante :

- **float** ;
- **double** ;
- **long double**.

La norme C90 était assez floue concernant les nombres à virgule flottante, leurs représentations, la précision des opérations, etc., ce qui fait que c'était un des domaines où la conception de programmes utilisant les nombres flottants était chose peu aisée. Le C99 a clarifié les choses en précisant qu'une implémentation C devait respecter la norme **IEC 60559:1989** ***Arithmétique binaire en virgule flottante pour systèmes à microprocesseur***. Cette norme (dérivée de IEEE 754) définit des formats de données pour les nombres à virgule flottante, ainsi que des opérations et fonctions sur ces nombres. Elle garantit, entre autres :

- que certains types de données auront toujours le même format ;
- et que les calculs effectués sur un type de donnée donneront toujours le même résultat.

Elle définit de plus comment sont gérés les cas exceptionnels, comme les infinis, les *NaN* (pour *Not a Number*, résultant par exemple de la division 0/0), etc.

Les types flottants du C correspondent aux type IEC 60559 de la manière suivante :

- **float** : simple précision
- **double** : double précision
- **long double** : suivant l'implémentation, soit la double précision étendue, soit un type non-IEC 60559 (mais de précision au moins égale à **double**), soit double précision.

Constantes réelles

Une suite de caractères représente une constante à virgule flottante si :

- c'est une suite de chiffres séparée par un caractère « point », cette séparation pouvant s'effectuer à n'importe quel endroit de la suite (`0.0`, `.0` et `0.` représentent tous les trois la valeur 0 de type `double`) ;
- un *nombre* suivi d'un caractère « e » suivi d'un entier.

Dans le deuxième cas, le *nombre* peut être soit un entier, soit un réel du premier cas.

Les constantes sont de type `double` par défaut. Pour demander le type `float`, il faut la suffixer par `f` ou `F`, et pour le type `long double` par `l` ou `L`.

Arithmétique

Une attention particulière doit être portée sur la précision des types réels. Ces différents types ne font qu'approximer l'ensemble des nombres réels, avec une précision finie. Des erreurs d'arrondis sont à prévoir, ce qui est très problématique pour des domaines qui n'en tolèrent pas (notamment pour les applications financières, il est conseillé de ne pas utiliser de calcul en virgule flottante).

Le type `float` en particulier a une précision minimale, qui est bien souvent insuffisante. Voici un exemple classique d'erreur à ne pas faire qui illustre les problèmes liés à la précision des types flottants :

```c
#include <stdio.h>

int main(void)
{
    float i = 0;
    int j;
    for (j = 0; j < 1000; j ++)
    {
        i += 0.1;
    }
    printf("i = %f\n", i);
    return 0;
}
```

Le résultat est 99,999046, ce qui montre que la précision du type `float` est en général mauvaise, d'autant plus que le nombre 0,1 n'est pas représentable de manière exacte en binaire[8]. Il est ainsi conseillé d'utiliser le type `double` à la place de `float` autant que possible. Dans ce cas de figure, il est préférable d'éviter les accumulations d'erreurs infinitésimales, en réécrivant le code de la manière suivante :

```c
#include <stdio.h>

int main(void)
{
    float i;
    int j;
    for (j = 0; j < 1000; j++)
    {
        i = j * 0.1;
    }
    printf("i = %f\n", i);
    return 0;
```

```
}
```

C'est probablement plus coûteux, car on effectue un millier de multiplications au lieu d'un millier d'additions, mais cela permet d'avoir une précision nettement meilleure que le code précédent.

Pour plus d'informations sur ce domaine, un wikilivre Arithmétique flottante est disponible.

Caractères

À l'origine, le type permettant de représenter un caractère est `char`. Même si un `char` n'est plus toujours suffisant aujourd'hui pour représenter un caractère quelconque.

Ce type est un peu plus particulier que les autres, d'une part parce que sa taille définit l'unité de calcul pour les quantités de mémoire (et donc pour les tailles des autres types du langage) et d'autre part son domaine de valeur peut grandement varier de manière relativement inattendue.

Par définition, la taille du type `char`, notée `sizeof(char)`, vaut toujours 1. Cependant, il faut faire attention : contrairement à ce qu'on pense souvent, un `char` au sens du C ne vaut pas toujours un *octet*. Il occupera au minimum 8 bits, mais il existe des architectures, relativement spécialisées il est vrai, ayant des `char` de 9 bits, de 16 bits, voire plus. Même si, dans une large majorité des cas, les compilateurs utilisent des `char` de 8 bits, à la fois par simplicité (les machines modernes fonctionnent généralement en 8, 16, 32 ou 64 bits) et pour éviter des problèmes de portabilité de code (beaucoup de codes C existants reposent sur l'hypothèse que les `char` font 8 bits, et risqueraient de ne pas marcher sur une autre architecture)[9]. Par simplification, nous utiliserons donc le terme *octet* la plupart du temps dans la suite de ce wikilivre.

Un autre piège de ce type est qu'il peut être **de base** (c'est-à-dire implicitement) `signed` ou `unsigned`, au choix du compilateur, ce qui peut s'avérer dangereux (c'est-à-dire difficile à maitriser). Considérez le code suivant :

```
/* Ce code peut ne pas fonctionner avec certains compilateurs */
char i;
for (i = 100; i >= 0; i --)
{
    /* ... */
}
```

Dans cet exemple, l'instruction `for` permet de faire itérer les valeurs entières de `i` de 100 à 0, incluses. On pourrait naïvement penser **optimiser** en utilisant un type `char`. Sauf que si ce type est implicitement `unsigned`, la condition `i >= 0` sera toujours vraie, et tout ce que vous obtiendrez est une boucle infinie. Normalement, tout bon compilateur devrait vous avertir que la condition est toujours vraie et donc vous permettre de corriger en conséquence, plutôt que perdre des heures en débugage.

Constantes représentant un caractère

Une constante représentant un caractère (de type char) est délimitée par des apostrophes, comme par exemple `'a'`. En fait pour le C, les caractères ne sont ni plus ni moins que des nombres entiers (ils sont de type `int`, mais leur valeur tiendra dans un type `char`), les deux étant parfaitement interchangeables. Cependant, comme dit plus haut, il ne s'agit généralement pas là des caractères Unicode que nous manipulons tous les jours, mais de caractère-octet dont la portée est réduite.

Un petit exemple :

```
#include <stdio.h>

int main(void)
```

```c
{
    printf("Sur votre machine, la lettre 'a' a pour code %d.\n", 'a');
    return 0;
}
```

Le programme précédent donnera le résultat suivant dans un environnement ASCII :

```
Sur votre machine, la lettre 'a' a pour code 97.
```

Il est tout à fait autorisé d'écrire `'a' * 2` ou `'a' - 32`.

Cette soustraction permet de convertir une minuscule ASCII en majuscule si le codage ASCII est utilisé, mais ne le permet pas si un autre codage est utilisé (comme ISO-8859-1 ou UTF-8).

Valeur des caractères-octets

La valeur représentée par cette constante est néanmoins dépendante des conventions de codages de caractères employées. À l'origine ces conventions étaient connues du système ; aujourd'hui, même si dans une écrasante majorité des cas, on se retrouvera avec un jeu de caractères augmentant l'ASCII et donc l'ISO-646. La code ASCII définit 96 glyphes de caractères portant les numéros 32 à 126, bien loin des milliers de caractères nécessaires pour les logiciels fonctionnant sur la planète entière et dans des langues nombreuses[10]. Cet ensemble est à peine suffisant pour couvrir l'anglais alors que plusieurs langues latines étaient visées à l'époque où l'ASCII et l'ISO-646 ont été définis, si bien que de nombreuses extensions sont par la suite apparues.

Par exemple le caractère « œ » (ligature du o et du e) a pour valeur :

- 189 dans le jeu de caractères ISO-8859-15 (principalement utilisé pour les langues latines d'Europe, sous Unix) ;
- 156 sur certaines variantes du jeu de caractères Windows 1252 (principalement utilisé pour les langues latines d'Europe, sous Windows) ;
- 0xc5, 0x93 (deux octets, donc deux `char`) en l'UTF-8 ;
- 207 avec l'encodage Mac Roman (Mac OS 9 et antérieur) ;
- Et n'a pas d'équivalent ni en ISO-8859-1 ni en ASCII (le caractère 189 est le symbole « ½ », le 207 est le « Ï » et le 156 n'est pas utilisé).

Retenons simplement qu'Unicode se développe sur Internet, mais que des logiciels plus anciens ou embarqués peuvent fonctionner avec des jeux de caractères huit bits, plus limités. Il s'agit en fait d'une problématique concernant des sujets plus vastes, comme l' *internationalisation*, la *portabilité* et l' *interopérabilité*.

Le langage C reste relativement agnostique à ce niveau : les caractères sont des octets et les chaînes, une simple suite d'octets terminée par 0. Il laisse au système et au développeur le soin d'interpréter et de traiter les octets comme il se doit. Ceci peut être fait à l'aide de bibliothèques appropriées. Si on trouve ici où là de telles listes de bibliothèques [11], il est préférable de connaître le langage C avant de les utiliser.

Cet agnosticisme du langage C lui a permis de s'adapter aux nombreuses évolutions des conventions de codages des caractères.

Caractères non graphiques

Certains caractères ne sont pas graphiques. Par définition, on ne peut pas donc les écrire dans un code source de manière visible. Le langage C adopte la convention suivant pour désigner certains d'entre eux de manière littérale :

Constante	Caractère
`'\''`	une apostrophe
`'\"'`	un guillemet
`'\?'`	un point d'interrogation
`'\\'`	un backslash

`'\a'`	un signal sonore (ou visuel)
`'\b'`	un espace arrière
`'\f'`	saut au début de la page suivante
`'\n'`	saut de ligne
`'\r'`	un retour chariot
`'\t'`	une tabulation
`'\v'`	une tabulation verticale

De plus, on peut écrire n'importe quelle valeur de caractère avec les expressions suivantes :

- `'\x`*`HH`*`'`, où chaque *`H`* représente un chiffre hexadécimal correspondant au code du caractère. Par exemple, `'\x61'` représente le caractère `'a'` (minuscule) en ASCII (car 97 = 6 * 16 + 1) ;
- `"\xc5\x93"` représente le caractère œ en UTF-8.
- `'\o`*`OO`*`'`, où chaque *`O`* représente un chiffre octal correspondant au code du caractère.

Trigraphe

Le langage C date d'une époque où le standard de codage des caractères était l'ISO-646, un standard encore plus ancien et plus incomplet que l'ASCII. Les caractères absent de l'ISO-646 pouvaient être simulés par une séquence de trois caractères.

L'ASCII s'étant répandu, cette fonctionnalité tombée en désuétude est en général à éviter, car très souvent inutile, et peut causer des bugs incompréhensibles au programmeur non averti. Néanmoins, de très rares programmes peuvent utiliser ce genre de fonctionnalités. Un trigraphe est simplement une suite de trois caractères dans le code source qui sera remplacée par un seul.

Cette fonctionnalité a été ajoutée au C pour supporter les architectures (systèmes) dont l'alphabet ne dispose pas de certains caractères qui sont nécessaires dans la syntaxe du C, comme les dièses ou les accolades.

Les substitutions suivantes sont faites *partout* dans le code source (y compris les chaînes de caractères) :

Trigraphe	Caractère
`??=`	`#`
`??(`	`[`
`??)`	`]`
`??<`	`{`
`??>`	`}`
`??/`	`\`
`??'`	`^`
`??!`	`|`
`??-`	`~`

Voici une manière de rendre illisible un programme utilisant les trigraphes :

```
??=include <stdio.h>

int main(void)
??<
   puts("Bonjour !");
   return 0;
??>
```

Par défaut, la plupart des compilateurs désactivent les trigraphes, au cas où vous ne seriez pas encore dissuadé de les

utiliser.

Chaîne de caractères

Une chaîne de caractère, comme son nom l'indique, est une suite de caractères avec la particularité d'avoir un caractère nul (0) à la fin. Une chaîne de caractère est en fait implémentée en C avec un tableau de type `char`.

Ce chapitre vous donne les rudiments sur les chaînes de caractère pour une première approche. Pour en savoir plus sur

- les rares fonctions de chaînes de caractères, voir Programmation C/Chaînes de caractères
- les tableaux & pointeurs : Programmation_C/Tableaux#Cha.C3.AEnes_de_caract.C3.A8res

```
Programmation_C/Tableaux#Chaînes_de_caractères
```

Structure

En mémoire, une chaîne de caractère est représenté dans le langage C comme un tableau d'octet dont les valeurs dépendent de l'encodage utilisé ici, Unicode.

***Amélie* en Unicode formes NFC et NFD**

Caractère représenté	A	m	é		l	i	e	'\0'
Unicode NFC	0041	006d	00e9		006c	0069	0065	0
Unicode UTF-8 NFC	41	6d	c3 a9		6c	69	65	0
Unicode NFD	0041	006d	0065	0301	006c	0069	0065	0
Unicode UTF-8 NFD	41	6d	65	cc 81	6c	69	65	0
Unicode NFD	A	m	e	◌́	l	i	e	'\0'

Chaîne littérale

On appelle une chaîne littérale la manière dont est définie une constante chaîne dans un code sources. Sous sa forme la plus simple, on déclare une chaîne comme une suite de caractères entre guillemets (double quote) :

```
"Ceci est une chaîne de caractère";
""; /* Chaîne vide */
```

Si la chaîne est trop longue, on peut aussi la couper sur plusieurs lignes :

```
"Ceci est une chaîne de caractère, " /* pas de ; */
"déclarée sur plusieurs lignes.";
```

Deux chaînes côtes à côtes (modulo les espaces et les commentaires) seront concaténées par le compilateur. De plus on peut utiliser le caractère barre oblique inverse (\) pour annuler la signification spéciale de certains caractères ou utiliser des caractères spéciaux (C.f liste ci-dessus).

```
"Une chaîne avec des \"guillemets\" et une barre oblique (\\)\n";
```

Il est aussi possible d'utiliser la notation hexadécimale et octale pour décrire des caractéres dans la chaîne, il faut néanmoins faire attention avec la notation hexadécimale, car la définition peut s'étendre sur plus de 2 caractéres. En fait, tous les caractères suivant le `\x` et appartenant à l'ensemble `"0123456789abcdefABCDEF"` seront utilisés pour déterminer la valeur du caractère (du fait que les caractères d'une chaîne peuvent faire plus qu'un octet). Cela peut produire certains effets de bords comme :

```
/* Ce code contient un effet de bord inattendu */
"\x00abcdefghijklmnopqrstuvzxyz"
```

Ce code n'insèrera pas un caractère 0 au début de la chaine, mais toute la séquence `"\x00abcdef"` servira à calculer la valeur du caractère (même si le résultat sera tronqué pour tenir dans un type `char`). On peut éviter cela en utilisant la concaténation de chaînes constantes :

```
"\x00" "abcdefghijklmnopqrstuvzxyz"
```

Enfin, les chaînes de caractères faisant appel aux concepts de pointeur, tableau et de zone mémoire statique, leur utilisation plus poussée sera décrite dans la section dédiée aux tableaux.

À noter la représentation **obsolète** des chaînes de caractères multilignes. À éviter dans la mesure du possible :

```
/* Ce code est obsolete et a eviter */
"Ceci est une chaîne de caractère,\
déclarée sur plusieurs lignes";
```

Booléens

Le langage (jusqu'à la norme C99) ne fournit pas de type booléen. La valeur entière 0 prend la valeur de vérité *faux* et toutes les autres valeurs entières prennent la valeur de vérité *vrai*.

La norme C99 a introduit le type `_Bool`, qui peut contenir les valeurs 0 et 1. Elle a aussi ajouté l'en-tête `<stdbool.h>`, qui définit le type `bool` qui est un raccourci pour `_Bool`, et les valeurs `true` et `false`[12]. Néanmoins, ces nouveautés du C99 ne sont pas très utilisées, les habitudes ayant été prises d'utiliser 0 et *différent de zéro* pour les booléens en C.

Nota : toute expression utilisant des opérateurs booléens (voir opérateurs), retourne 1 si l'expression est vraie et 0 si elle est fausse, ce qui rend quasiment inutile l'usage du type booléen.

Vide

En plus de ces types, le langage C fournit un autre type, `void` qui représente *rien*, le *vide*. Il n'est pas possible de déclarer une variable de type `void`. Nous verrons l'utilité de ce type lorsque nous parlerons de fonctions et de

pointeurs.

Classes de stockage

Il est possible de construire des types dérivés des types de base du langage en utilisant plusieurs combinaisons, deux étant illustrées dans ce chapitre: les classes de stockage et les qualificateurs.

Classe de stockage

Le langage C permet de spécifier, avant le type d'une variable, un certain nombre de *classes de stockage* :

- `auto` : pour les variables locales ;
- `extern` : déclare une variable sans la définir ;
- `register` : demande au compilateur de faire tout son possible pour utiliser un registre processeur pour cette variable ;
- `static` : rend une définition de variable persistante.

Les classes `static` et `extern` sont, de loin, les plus utilisées. `register` est d'une utilité limitée, et `auto` est maintenant obsolète.

Une variable, ou un paramètre de fonction, ne peut avoir qu'au plus une classe de stockage.

Classe 'static'

L'effet de la classe 'static' dépend de l'endroit où l'objet est déclaré :

- *Objet local à une fonction* : la valeur de la variable sera persistante entre les différents appels de la fonction. La variable ne sera visible que dans la fonction, mais ne sera pas réinitialisée à chaque appel de la fonction. L'intérêt est de garantir une certaine encapsulation, afin d'éviter des usages multiples d'une variable globale. Qui plus est, cela permet d'avoir plusieurs fois le même nom, dans des fonctions différentes.

Exemple :

```
#include <stdio.h>

void f(void)
{
    static int i = 0; /* i sera initialisée à 0 à la compilation seulement */
    int j = 0; /* j sera initialisée à chaque appel de f */;
    i++;
    j++;
    printf("i vaut %d et j vaut %d.\n", i, j);
}

int main(void)
{
    f();
    f();
    f();
    return 0;
}
```

Résultat d'exécution du code ci dessus :

```
i vaut 1 et j vaut 1.
i vaut 2 et j vaut 1.
i vaut 3 et j vaut 1.
```

- *Objet global et fonction* : comme une variable globale est déjà persistante, le mot-clé `static` aura pour effet de limiter la portée de la variable ou de la fonction au seul fichier où elle est déclarée, toujours dans le but de garantir un certain niveau d'encapsulation.

Une variable de classe statique est initialisée au moment de la compilation à zéro par défaut (contrairement aux variables dynamiques qui ont une valeur initiale indéterminée). Elle peut être initialisée explicitement à n'importe quelle valeur *constante*.

Classe 'extern'

`extern` permet de *déclarer* une variable sans la *définir*. C'est utile pour la compilation séparée, pour définir une variable ou une fonction dans un fichier, en permettant à des fonctions contenues dans d'autres fichiers d'y accéder.

Toutes les variables globales et fonctions qui ne sont pas déclarées (ou définies) `static` sont externes par défaut.

`static` et `extern` sont employés pour distinguer, dans un fichier C, les objets et fonctions « publics » , qui pourront être accessibles depuis d'autres fichiers, de ceux qui sont « privés » et ne doivent être utilisés que depuis l'intérieur du fichier.

Classe 'register'

L'usage de ce mot clé est utile dans un contexte de logiciel embarqué. Indique que la variable devrait être stockée dans un registre du processeur. Cela permet de gagner en performance par rapport à des variables qui seraient stockées dans un espace mémoire beaucoup moins rapide, comme une pile placée en mémoire vive.

Ce mot-clé a deux limitations principales :

- Les registres du processeur sont limités. Leur nombre peut varier en fonction du processeur. Sur un PC (d'architecture AMD64?), il sont au nombre de 13, dont seulement 4 servent au stockage (EAX,EBX,ECX,EDX). Il est donc inutile de déclarer une structure entière ou un tableau avec le mot clé `register`.
- Qui plus est, les variables placées dans des registres sont forcément locales à des fonctions ; on ne peut pas définir une variable globale en tant que registre.

Aujourd'hui, ce mot-clé est déconseillé sauf pour des cas particuliers, les compilateurs modernes sachant généralement mieux que le programmeur comment optimiser et quelles variables placer dans les registres.

```c
#include <stdio.h>

int main(void)
{
    register short i, j;
    for (i = 1; i < 1000; ++i)
    {
        for(j = 1; j < 1000; ++j)
        {
            printf("\n %d %d", i, j);
        }
    }
    return 0;
}
```

Classe 'auto'

Cette classe est un héritage du langage B. En C, ce mot-clé sert pour les variables locales à une fonction non-statiques, dites aussi *automatiques*. Mais une variable déclarée localement à une fonction sans qualificateur `static` étant implicitement automatique, ce mot-clé est inutile en C.

Qualificateurs

Le C définit trois qualificateurs pouvant influer sur une variable :

- `const` : pour définir une variable dont la valeur ne devrait jamais changer ;
- `restrict` : permet une optimisation pour la gestion des pointeurs ;
- `volatile` : désigne une variable pouvant être modifiée notamment par une source externe indépendante du programme.

Une variable, ou un paramètre de fonction, peut avoir aucun, un, deux, ou les trois qualificateurs (certaines combinaisons n'auraient que peu de sens, mais sont autorisées).

Qualificateur 'const'

La classe `const` ne déclare pas une vraie constante, mais indique au compilateur que la valeur de la variable ne doit pas changer. Il est donc impératif d'assigner une valeur à la déclaration de la variable, sans quoi toute tentative de modification ultérieure entrainera une erreur de la part du compilateur :

⚠ Ce code contient **une erreur volontaire** !

```
const int i = 0;
i = 1; /* erreur*/
```

En fait, le mot-clé `const` est beaucoup plus utilisé avec des pointeurs. Pour indiquer qu'on ne modifie pas l'objet pointé, il est bon de spécificier le mot-clé `const` :

```
void fonction( const char * pointeur )
{
        pointeur[0] = 0;      /* erreur*/
        pointeur    = "Nouvelle chaine de caractères";
}
```

Dans cet exemple, on indique que l'objet pointé ne sera pas modifié. Pour indiquer que la valeur elle-même du pointeur est constante, il faut déclarer la variable de la sorte :

⚠ Ce code contient **une erreur volontaire** !

```
char * const pointeur = "Salut tout le monde !";
pointeur = "Hello world !"; /* erreur*/
```

Encore plus subtil, on peut mélanger les deux :

⚠ Ce code contient **plusieurs erreurs volontaires** !

```
const char * const pointeur = "Salut tout le monde !";
pointeur = "Hello world !"; /* erreur*/
pointeur[0] = 0; /* erreur*/
```

Cette dernière forme est néanmoins rarement usitée. En outre ce dernier exemple présente un autre problème qui est

la modification d'une chaine de caractères « en dur », qui sont la plupart du temps placées dans la section lecture seule du programme et donc inaltérables.

Qualificateur 'volatile'

Ce mot-clé sert à spécifier au compilateur que la variable peut être modifiée à son insu. Cela annule toute optimisation que le compilateur pourrait faire, et l'oblige à procéder à chaque lecture ou écriture dans une telle variable tel que le programmeur l'a écrit dans le code. Ceci a de multiples utilisations :

- pour les coordonnées d'un pointeur de souris qui seraient modifiées par un autre programme ;
- pour la gestion des signaux (voir Gestion des signaux) ;
- pour de la programmation avec de multiples fils d'exécution qui doivent communiquer entre eux ;
- pour désigner des registres matériels qui peuvent être accédés depuis un programme C (une horloge, par exemple), mais dont la valeur peut changer indépendamment du programme ;
- etc.

On peut combiner `const` et `volatile` dans certaines situations. Par exemple :

```
extern const volatile int horloge_temps_reel;
```

déclare une variable entière, qu'on ne peut modifier à partir du programme, mais dont la valeur peut changer quand même. Elle pourrait désigner une valeur incrémentée régulièrement par une horloge interne.

Qualificateur 'restrict'

Introduit par C99, ce mot-clé s'applique aux déclarations de pointeurs uniquement. Avec *restrict*, le programmeur certifie au compilateur que le pointeur déclaré sera le seul à pointer sur une zone mémoire. Cela permettra au compilateur d'effectuer des optimisations qu'il n'aurait pas pu *deviner* autrement. Le programmeur ne doit pas *mentir* sous peine de problèmes...

Par exemple :

```
int* restrict pZone;
```

Opérateurs

Les opérateurs du C

Les opérateurs du C permettent de former des expressions, expressions qui diront quoi faire à votre programme. On peut voir un programme C comme étant composé de trois catégories d'instructions :

- Les **déclarations** et **définitions** (variables, fonctions, types) : déclarent et définissent les objets que pourront manipuler le programme.
- Les **expressions** : manipulent les déclarations, via les opérateurs.
- Les **instructions** : manipulent les expressions pour leur donner une signification particulière (test, boucle, saut, ...).

Les déclarations de variables ont en partie été décrites dans les chapitres précédents. Les expressions seront en partie décrites dans celui-ci (les opérateurs liés aux pointeurs, tableaux, structures et fonctions seront décrit dans des chapitres dédiés), et les instructions seront décrites au cours des chapitres suivants.

Les expressions en C sont en fait très génériques, elles peuvent inclure des opérations arithmétiques classiques

(addition, soustraction, division, modulo, ...), des expressions booléennes (OU logique, ET logique, OU exclusif, ...), des comparaisons (égalité, inégalité, différence, ...) et même des affectations (copie, auto-incrément, ...). Toutes ces opérations peuvent s'effectuer en une seule expression, il suffit d'appliquer les bons opérateurs sur les bons opérandes.

Les opérateurs binaires et ternaires utilisent une notation infixe (l'opérateur se trouve entre les 2 ou 3 opérandes). Les opérateurs unaires s'écrivent de manière préfixé (avant l'opérande), à l'exception des opérateurs ++ et -- qui peuvent s'écrire de manière préfixée ou suffixée (avec une différence subtile, décrite ci-après).

La **priorité** (quel opérateur est appliqué avant, en l'absence de parenthèses explicite) et l'**associativité** (dans quel ordre sont traités les arguments des opérateurs ayant la même priorité) sont résumées dans la table suivante (par ordre décroissant de priorité - les opérateurs décrits dans un autre chapitre ont un lien dédié) :

opérateur	parité	associativité	description
`( )`		gauche vers la droite (GD)	parenthésage
`() [] . ->`		GD	appel de fonction, index de tableau, membre de structure, pointe sur membre de structure
`!`	unaire	droite vers la gauche (DG)	négation booléenne
`~`	unaire	DG	négation binaire
`++ --`	unaire	DG	incrémentation et décrémentation
`-`	unaire	DG	opposé
`(`*`type`*`)`	unaire	DG	opérateur de transtypage (cast)
`*`	unaire	DG	opérateur de déréférençage
`&`	unaire	DG	opérateur de référençage
`sizeof`	unaire	DG	fournit la taille en nombre de "char" de l'expression (souvent en octet mais pas toujours, mais sizeof(char) == 1 par définition, voir Caractères)
`* / %`	binaire	GD	multiplication, division, modulo (reste de la division)
`+ -`	binaire	GD	addition, soustraction
`>> <<`	binaire	GD	décalages de bits
`> >= < <=`	binaire	GD	comparaisons
`== !=`	binaire	GD	égalité/différence
`&`	binaire	GD	**et** binaire
`^`	binaire	GD	**ou** exclusif binaire
`|`	binaire	GD	**ou** inclusif binaire
`&&`	binaire	GD	**et** logique avec séquencement
`||`	binaire	GD	**ou** logique avec séquencement
`? :`	ternaire	DG	**si...alors...sinon**
`= += -= *= /= %= ^= &= |= >>= <<=`	binaire	DG	affectation
`,`	binaire	GD	séquencement

Post/pré incrémentation/décrémentation

C'est un concept quelque peu sibyllin du C. Incrémenter ou décrémenter de un est une opération extrêmement courante. Écrire à chaque fois '`variable = variable + 1`' peut-être très pénible à la longue. Le langage C a donc introduit des opérateurs raccourcis pour décrémenter ou incrémenter n'importe quel type atomique (gérable directement par le processeur : c'est à dire pas par un tableau, ni une structure ou une union). Il s'agit des opérateurs '++' et '--', qui peuvent être utilisés de manière préfixée ou suffixée (avant ou après la variable).

Utilisé de manière préfixée, l'opérateur incrémente/décrémente la variable, puis retourne la valeur de celle-ci. En fait, les expressions `(++E)` et `(--E)` sont équivalentes respectivement aux expressions `(E+=1)` et `(E-=1)`.

Par contre, utilisé de manière suffixée, l'opérateur retourne la valeur originale avant de modifier la valeur de la variable.

```
int i = 0, j;

j = i++; /* j vaut  0 et i vaut  1 */
j = --i; /* j vaut 0 et i vaut 0 */
```

Il est important de noter que le langage ne fait que garantir qu'une variable post-incrémentée ou post-décrémentée acquiert sa nouvelle valeur entre le résultat de l'opérateur et le prochain point de séquencement atteint (généralement la fin d'une instruction). Mais on ne sait pas vraiment quand entre les deux. Ainsi, si l'objet sur lequel s'applique un tel opérateur apparaît plusieurs fois avant un point de séquencement, alors le résultat est **imprévisible** et son comportement peut changer simplement en changeant les options d'un même compilateur :

⚠ Ce code contient **plusieurs erreurs volontaires** !

```
/* Le code qui suit est imprévisible */
int i = 0;
i = i++;
```

⚠ Ce code contient **une erreur volontaire** !

```
/* Et celui-là d'après vous ? */
int i = 0;
i = ++i;
```

Contrairement à ce à quoi vous pouviez vous attendre, le bout de code ci-dessous n'est pas mieux défini que celui d'au-dessus. La valeur de i y est modifiée une première fois lors de son incrémentation, puis une deuxième fois par l'opérateur d'affectation =. Or, un autre moyen de générer un comportement indéfini est de modifier la valeur d'une variable plusieurs fois entre deux points de séquencement.

Promotion entière

La promotion entière, à ne pas confondre avec la conversion automatique de type, fait que tous les types plus petits ou égaux à `int` (`char`, `short`, champs de bits, type énuméré) sont convertis (promus) en `int` ou `unsigned` avant toute opération. Ainsi, dans l'exemple suivant, `a + b` est calculé avec des `int` et le résultat est de type `int` :

```
short sum(short a, short b) {
        return a + b; /* équivaut à : return (int)a + (int)b; */
}
```

Le compilateur peut émettre un avertissement du fait que le résultat `int` est converti en `short` pour être retourné, et cette conversion peut causer une perte de précision (par exemple si `int` a une largeur de 32 bits et `short` de 16 bits).

La promotion se fait vers `unsigned` lorsqu'un `int` ne peut pas représenter le type promu.

Conversion automatique

La conversion est un mécanisme qui permet de convertir implicitement les nombres dans le format le plus grand utilisé dans l'expression.

L'exemple classique est le mélange des nombres réels avec des nombres entiers. Par exemple l'expression '`2 / 3.`' est de type double et vaudra effectivement deux tiers (0,666666...). L'expression '`2 * a / 3`' calculera les deux tiers de la variable 'a', et arrondira automatiquement à l'entier par défaut, les calculs ne faisant intervenir que des instructions sur les entiers (en supposant que 'a' soit un entier). À noter que l'expression '`2 / 3 * a`' vaudra... toujours zéro !

Plus subtile, l'expression '`2 * a / 3.`', en supposant toujours que *a* soit un entier, effectuera une multiplication entière entre 2 et a, puis promouvra le résultat en réel (`double`) puis effectuera la division réelle avec trois, pour obtenir un résultat réel lui-aussi.

Enfin un cas où les promotions peuvent surprendre, c'est lorsqu'on mélange des entiers signés et non-signés, plus particulièrement dans les comparaisons. Considérez le code suivant :

```
/* Ce code contient un effet de bord sibyllin */
unsigned long a = 23;
signed char b = -23;

printf( "a %c b\n", a < b ? '<' : (a == b ? '=' : '>') );
```

De toute évidence `a` est supérieur à `b` dans cet exemple. Et pourtant, si ce code est exécuté sur certaines architectures, il affichera `a < b`, justement à cause de cette conversion.

Tout d'abord dans la comparaison `a < b`, le type de `a` est « le plus grand », donc `b` est promu en `unsigned long`. C'est en fait ça le problème : -23 est une valeur négative, et la conversion d'une valeur négative en un type non signé se fait modulo la valeur maximale représentable par le type non signé + 1. Si la valeur maximale pour le type `unsigned long` est 2^{32}-1, alors -23 est converti modulo 2^{32}, et devient 4294967273 (soit 2^{32}-23). Dans ce cas là effectivement 23 < 4294967273, d'où ce résultat surprenant.

À noter qu'on aurait eu le même problème si le type de `b` était `signed long`. En fait les types signés sont promus en non-signés, s'il y a au moins une variable non-signée dans l'expression, et que le type non-signé est "plus grand" que le type signé.

Dans ce cas présent, il faut effectuer soi-même le transtypage :

```
printf( "a %c b\n", (long) a < b ? '<' : ((long)a == b ? '=' : '>') );
```

La norme définit précisément les conversions arithmétiques et les promotions entières, se référer à elle pour les détails.

Évaluation des expressions booléennes

Le C ne possède pas de type booléen dédié[13]. Dans ce langage, n'importe quelle valeur différente de zéro est considérée vraie, zéro étant considéré comme faux. Ce qui veut dire que n'importe quelle expression peut être

utilisée à l'intérieur des tests (entier, réels, pointeurs, tableaux, etc.). Cela peut conduire à des expressions pas toujours très claires, comme :

```
int a;
a = une_fonction();
if (a)
{
    /* ... */
}
```

Ce type d'écriture simplifiée a ses adeptes, mais elle peut aussi s'écrire de la manière suivante:

```
int a;
a = une_fonction();
if (a != 0)
{
    /* ... */
}
```

Cette seconde écriture, sémantiquement équivalente à la première, rend explicite le test qui est effectué (on compare la valeur de la variable a à zéro).

Par ailleurs, cette absence de type booléen rend aussi valide le code suivant:

```
int a = 0;
int b = 2;

if (a = b)
{
        /* Le code qui suit sera toujours exécuté ... */
}
```

Dans cet exemple, il y a un seul caractère = entre a et b, donc on ne teste pas si a est égal à b, mais on **affecte** la valeur de la variable b à la variable a. Le résultat de l'expression étant 2, elle est donc toujours considérée vraie. Ce genre de raccourci est en général à éviter. En effet, il est facile de faire une faute de frappe et d'oublier de taper un caractère =. Comme le code résultant est toujours valide au sens du C, cette erreur peut ne pas être vue immédiatement. Pour aider les développeurs à détecter ce genre d'erreurs, de nombreux compilateurs émettent un avertissement quand un tel code leur est présenté. Une manière de faire taire ces avertissements, *une fois qu'on s'est assuré qu'il ne s'agit pas d'une erreur*, est de mettre entre parenthèses l'affectation:

```
/* On veut faire une affectation ici. */
/* doublement des parenthèses pour supprimer l'avertissement du compilateur*/
if ((a = b))
{
    /* ... */
}
```

Comme un développeur devant maintenir un programme contenant un test `if (a = b) { /* ... */ }` se demandera si le développeur précédent n'a pas fait une faute de frappe, il est préférable d'éviter autant que possible ce genre de situations et, s'il est nécessaire (ce qui a très peu de chances d'arriver), de la commenter. En effet, même un code `if ((a = b)) { /* ... */ }` non commenté doit être lu avec attention, car le développeur précédent peut avoir ajouté les parenthèses juste pour faire taire le compilateur, sans se rendre compte d'une erreur. Dans tous les cas, la manière la plus sûre est de décomposer ainsi:

```
a = b;
if (a != 0)
{
    /* ... */
}
```

Une autre technique classique, lorsqu'une comparaison fait intervenir une constante, est de mettre la constante à gauche. De cette manière, si la comparaison se transforme par mégarde en affectation, cela provoquera une erreur à la compilation :

```
if (0 == b)
{
    /* Une instruction "0 = b" ne passerait pas */
}
```

Les opérateurs logiques de comparaisons (`&&` et `||`, similaires sémantiquement à leur équivalent binaire `&` et `|`) évaluent leurs opérandes en **circuit court**. Dans le cas du ET logique (`&&`), si l'opérande gauche s'évalue à faux, on sait déjà que le résultat du ET sera faux et donc ce n'est pas la peine d'évaluer l'opérande droite. De la même manière si l'opérande gauche d'un OU logique (`||`) est évalué à vrai, le résultat sera aussi vrai et donc l'évaluation de l'opérande droite est inutile. Ceci permet d'écrire des expressions de ce genre, sans générer d'exception de l'unité arithmétique du processeur :

```
if (z != 0 && a / z < 10)
{
    printf("Tout va bien\n");
}
```

Et voila !

On réduit :

```
if (string)
 len = strlen(string);
else
 len = 0;
```

par

```
len = string ? strlen(string) : 0;
```

C'est à dire que *len* égale si *string* à la longueur de *string* si non 0 ;)

L'ordinateur calcule très vite et très bien, c'est pour ça que l'utiliser reste un atout.

Les calculs ne sont pour lui que des instructions simples et utilisant peu de ressource (mul, div, add, sub, inc, xor, ...).

Structures de contrôle - tests

Les tests permettent d'effectuer des opérations différentes suivant qu'une condition est vérifiée ou non.

Test if

Les sauts conditionnels permettent de réaliser une instruction si une condition est vérifiée. Si la condition n'est pas vérifiée, l'exécution se poursuit séquentiellement.

Syntaxe

Première forme :

```
/* Ce code est volontairement incomplet */
    if (condition)
        instruction
    instruction suivante
```

Si la *condition* est vérifiée, alors on exécute l'instruction, sinon on exécute l'instruction suivante.

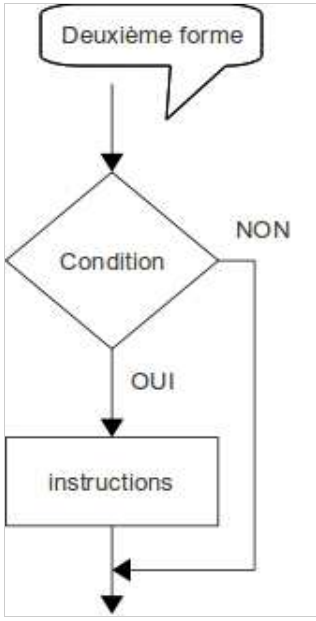
```
/* Ce code est volontairement incomplet*/
    if (condition) instruction; /* ... */
    instruction suivante; /* ... */ ;
```

Deuxième forme :

```
/* Ce code est volontairement incomplet */
    if (condition)
        instructions
    instruction suivante
```

Si la *condition* est vérifiée, alors on exécute le *bloc* d'instructions, sinon on exécute exécute l'instruction suivante.

```
/* Ce code est volontairement incomplet */
    if (condition)
    {
       instruction 1; /* ... */
       instruction 2; /* ... */
    }
    instruction suivante; /* ... */ ;
```



Deuxième forme

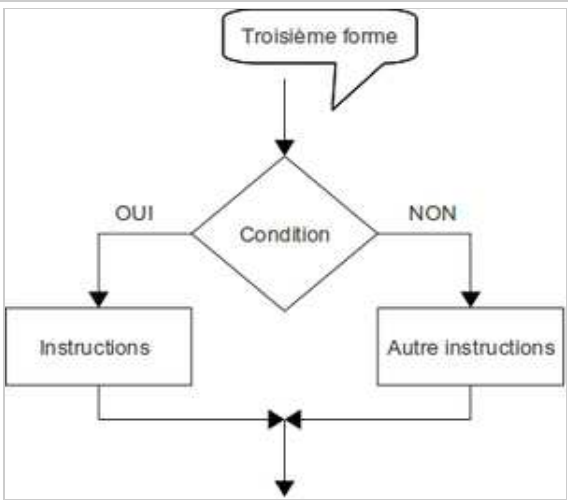
Troisième forme :

```
/* Ce code est volontairement incomplet */
    if (condition)
        instructions
    else
        autre instructions
```

```
    instruction suivante; /* ... */ ;
```

Si la *condition* est vérifiée, alors on exécute le *bloc* d'instructions, sinon on exécute l'autre bloc.

```
    /* Ce code est volontairement incomplet*/
    if (condition)
    {
       instruction 1; /* ... */
       instruction 2; /* ... */
    } else {
       instruction 3; /* ... */
       instruction 4; /* ... */
    }
    instruction suivante; /* ... */ ;
```



Troisième forme

Quatrième forme :

```
    /* Ce code est volontairement incomplet*/
    if (condition 1)
        instructions
    else if (condition 2)
        autre instructions
    else
        autre instructions
    instruction suivante; /* ... */ ;
```

Si la *condition* est vérifiée, alors on exécute le *bloc* d'instructions, sinon on exécute l'autre *if* comme la troisième forme.

```
/* Ce code est volontairement incomplet */
    if (condition 1)
    {
       instruction 1; /* ... */
       instruction 2; /* ... */
    } else if (condition 2) {
       instruction 3; /* ... */
       instruction 4; /* ... */
    } else {
       instruction 5; /* ... */
       instruction 6; /* ... */
    }
    instruction suivante; /* ... */ ;
```

Exemple de code :

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

int main(void)
{
    int a;
    a = 3; /* assignation d'une valeur pour l'exemple*/

    if (a >= 5)
        printf("Le nombre a est supérieur ou égal à 5 \n");
    else
        printf("Le nombre a est inférieur à 5 \n");
    return 0;
```

```
}
```

D'ailleurs, lorsqu'un **if** contient une instruction (`for`, `do`, `while`, `switch`, `goto`, `return`, etc.) il est conseillé de la mettre entre accolades.

Test switch

Cette instruction permet de tester si une expression coïncide avec un certain nombre de constantes, et d'exécuter une action par défaut dans le cas où aucune valeur ne correspond à celle de l'expression. Cela permet un traitement beaucoup plus efficace et lisible qu'une succession de `if/else` imbriqués. Insistons sur le terme *constante* : il est en effet impossible d'utiliser des expressions dont la valeur n'est pas connue à la *compilation* (c'est à dire de variable dans les instructions `case`).

Syntaxe

```
switch (expression)
{
    case valeur1:
        bloc1
    case valeur2:
        bloc2
    /*...*/
    case valeurN:
        blocN
    default:
        blocD
}
```

Compare la valeur de l'*expression* à celles de *valeur1*, *valeur2*, ..., *valeurN*. En cas d'égalité entre *expression* et *valeurI* les blocs sont exécutés séquentiellement à partir de *blocI*, et ce jusqu'à la fin de l'instruction *switch*. Si *expression* est égale à *valeur2*, dans cet exemple, les blocs *bloc2* à *blocN* et même *blocD* seront exécutés. Pour empêcher ce comportement on utilise l'instruction *break*, que l'on peut placer à n'importe quel endroit pour sortir (aller à la fin) de l'instruction *switch*. En général, on retrouve plus fréquemment l'instruction *switch* écrite de la sorte :

```
switch (expression)
{
    case valeur1:
        bloc1
        break;
    case valeur2:
        bloc2
        break;
    /*...*/
    case valeurN:
        blocN
        /* pas de break; */
    default:
        blocD
}
```

C'est en fait tellement rare de ne pas mettre de *break* entre les différents cas, qu'il est conseillé de mettre un commentaire pour les cas où cette instruction est délibérément omise, ça permet de bien signaler au lecteur qu'il ne s'agit pas d'un oubli.

Expression conditionnelle

```
condition ? expression_si_vrai : expression_si_faux ;
```

Exemple

```
#include <stdio.h>

int main(int argc, char * argv[])
{
    /*
        program                     argc=1  argv[0]="program"
        program argument            argc=2  argv[0]="program"  argv[1]="argument"
        program argument argument   argc=3  argv[0]="program"  argv[1]="argument"  argv[2]="argument"
        ...
    */
    printf("%s\n", argc < 2 ? "Vous n'avez pas donné d'argument." : "Vous avez donné au moins un argument.");
    return 0;
}
```

Ce mini-programme teste si le nombre d'arguments passé à `main` est inférieur à 2 avec l'expression `argc < 2` et renvoie `"vous n'avez pas donné d'argument"` si l'expression est vraie et `"vous avez donné au moins un argument"` sinon. Le résultat de l'évaluation est alors passé à la fonction `printf` qui affiche le résultat. (si `argc` vaut 2 alors il y a 1 argument : voir "la fonction main" dans le chapitre "fonctions")

On note le manque de clarté de l'exemple: l'opérateur ternaire ne doit être utilisé que dans de rares cas, c'est à dire lorsque son utilisation ne se fait pas au détriment de la lisibilité du code.

Structures de contrôle - itérations

Boucle for

Syntaxe

```
for (initialisation ; condition ; itération)
    bloc
```

Une boucle **for** commence par l' *initialisation*, puis exécute le *bloc* de code tant que la *condition* est vérifiée et en appelant l'*itération* après chaque *bloc*.

On peut tout de suite remarquer que la boucle **for** du C est beaucoup plus puissante qu'une boucle **for** *classique*, dans le sens où elle ne correspond pas forcément à un nombre fini de tours de boucle.

La norme C99 permet maintenant de déclarer des variables dans la partie initialisation de la boucle **for**. Ces variables ne peuvent qu'être de classe **automatic** ou **register**. Leur portée est limitée à la boucle. Elles n'interfèrent pas avec les variables de même nom déclarées en dehors de la boucle. Leur utilisation pourrait conduire à de possibles optimisations du code par le compilateur.

Exemples

Exemple 1

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

En C99 :

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

Cet exemple est une boucle **for** *normale*, l'indice de boucle *i* est incrémenté à chaque tour et le bloc ne fait qu'afficher l'indice. Ce code affiche donc les 10 premiers entiers en partant de 0. Le bloc de code étant réduit à une instruction, on aurait pu écrire :

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

Exemple 2

```c
int i = 0;
for( ; i != 1 ; i = rand() % 4)
{
        printf("je continue\n");
}
```

Ce deuxième exemple montre plusieurs choses :

1. l'initialisation de la boucle a été omise, en fait n'importe laquelle des trois parties peut l'être (et même les trois à la fois, ce qui résulte en une boucle infinie) ;
2. l'expression itération est **réellement** une expression, on peut y faire appel à des fonctions ;
3. le nombre d'itérations n'est pas fixe, le programme affichera « je continue » un nombre (pseudo-)aléatoire de fois.

Remarque : `for(;;)` crée une boucle infinie.

Boucle while

Syntaxe

```c
while (condition)
    bloc
```

Une boucle **while** exécute le *bloc* d'instructions tant que la *condition* est vérifiée.

Exemple

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

Cet exemple est le même que celui vu précédement pour la boucle **for**, il affiche les dix premiers entiers à partir de 0.

Boucle do-while

Syntaxe

```
do
    bloc
while (condition);
```

Cette boucle est une légère variante de la précédente. Elle exécute donc le bloc au moins une fois, et ce jusqu'à ce que la condition soit fausse (ou tant que la condition est vraie).

Exemple 1

```
#include <stdio.h>

int main(void)
{
    int i = 0;
    do
    {
        printf("%d\n", i);
        i = i + 1;
    } while (i < 10);
    return 0;
}
```

Exemple identique à celui de la boucle while au dessus, on affiche les nombres de 0 jusqu'à 9.

Exemple 2

```
#include <stdio.h>

int main(void)
{
    char c;
    do
    {
        printf("Veuillez taper la lettre 'o'\n");
        scanf("%c", &c);
    } while (c != 'o');
    return 0;
}
```

Cette fois on voit l'utilité de la boucle do-while, la variable est initialisée dans la boucle. On doit tester sa valeur à la fin de l'itération. Tant que l'utilisateur n'aura pas tapé la lettre o, le programme redemandera de taper la lettre.

Exemple 3

http://fr.wikibooks.org/wiki/Discussion:Programmation_C
/Pointeurs#Exemple_Complet_sur_les_boucles.2C_les_tableaux_et_les_pointeurs

Arrêt et continuation des boucles

Il arrive fréquemment qu'en évaluant un test à l'intérieur d'une boucle, on aimerait arrêter brutalement la boucle ou alors « sauter » certains cas non significatifs. C'est le rôle des instructions `break` et `continue`.

- `break` permet de sortir immédiatement d'une boucle `for`, `while` ou `do-while`. À noter que si la boucle se trouve elle-même dans une autre boucle, seule la boucle où l'instruction `break` se trouvait est stoppée.
- `continue` permet de recommencer la boucle depuis le début du bloc. Dans le cas de la boucle `for`, le bloc lié à l'incrémentation sera exécuté, puis dans tous les cas, la condition sera testée.

Saut inconditionnel (goto)

Cette instruction permet de continuer l'exécution du programme à un autre endroit, **dans la même fonction**. On l'utilise de la manière suivante :

```
goto label;
```

Où *label* est un identificateur quelconque. Cet identificateur devra être défini quelque part dans la même fonction, avec la syntaxe suivante :

```
label:
```

Ce label peut être mis à n'importe quel endroit dans la fonction, y compris avant ou après le `goto`, ou dans un autre bloc que là où sont utilisés la ou les instructions `goto` pointant vers ce label. Les labels doivent être uniques au sein d'une même fonction, mais on peut les réutiliser dans une autre fonction.

Un cas où l'emploi d'une instruction `goto` est tout à fait justifié, par exemple, est pour simuler le déclenchement d'une exception, pour sortir rapidement de plusieurs boucles imbriquées.

```
void boucle(void)
{
    while (1)
    {
        struct Evenement_t * event;

        attend_evenement();

        while ((event = retire_evenement()))
        {
            switch (event->type) {
            case FIN_TRAITEMENT:
                goto fin;
            /* ... */
            }
        }
    }
    fin:
```

```
    /* ... */
}
```

Dans ce cas de figure, il serait possible d'utiliser aussi un booléen qui indique que le traitement est terminé, pour forcer la sortie des boucles. L'utilisation de `goto` dans cet exemple ne nuisant pas à la lisibilité du programme et étant tout aussi efficace, il est tout à fait légitime de l'utiliser.

Il faut cependant noter que cette instruction est souvent considérée comme à éviter. En effet, elle provoque un saut dans le code, qui se « voit » beaucoup moins bien que les structures de contrôle comme `if` ou `while`, par exemple. De fait, la pratique a montré qu'un abus de son utilisation rend un code source rapidement peu compréhensible, et difficilement maintenable. L'article de Edsger W. Dijsktra *Go to considered harmful* (http://www.acm.org/classics/oct95/) [archive] anglais, datant de 1968, est l'exemple le plus connu de cette critique, qui a donné naissance à la programmation structurée.

Fonctions et procédures

Définition

Le code suivant définit une fonction `fonction` renvoyant une valeur de type `type_retour` et prenant N arguments, `par1` de type `type1`, `par2` de type `type2`, etc.

```
type_retour fonction(type1 par1, type2 par2, /* ..., */ typeN parN)
{
        /* Déclarations de variables ... */
        /* Instructions ... */
}
```

L'exécution d'une fonction se termine soit lorsque l'accolade fermante est atteinte, soit lorsque le mot clef `return` est rencontré. La valeur renvoyée par une fonction est donnée comme paramètre à `return`. Une procédure est une fonction renvoyant `void`, dans ce cas `return` est appelé sans paramètre.

Les passages des arguments aux fonctions se font toujours *par valeur*. Si on veut modifier la valeur d'un argument passé en paramètre à une fonction, en dehors de cette même fonction, il faut utiliser des pointeurs.

Déclaration par prototype

Le prototype d'une fonction correspond simplement à son en-tête (tout ce qui précède la première accolade ouvrante). C'est-à-dire son nom, son type de retour et les types des différents paramètres. Cela permet au compilateur de vérifier que la fonction est appelée avec le bon nombre de paramètres et surtout avec les bons types. La ligne suivante *déclare* la fonction `fonction`, mais sans la *définir* :

```
type_retour nom_fonction(type1, type2, /* ..., */ typeN);
```

À noter que les noms des paramètres peuvent être omis et que la déclaration **doit se terminer** par un point-virgule (;), sans quoi vous pourrez vous attendre à une cascade d'erreurs.

Absence des paramètres

Avant la normalisation par l'ANSI, il était possible de faire une déclaration partielle d'une fonction, en spécifiant son type de retour, mais pas ses paramètres:

```
int f();
```

Cette déclaration ne dit rien sur les éventuels paramètes de la fonction `f`, sur leur nombre ou leur type, au contraire de :

```
int g(void);
```

qui précise que la fonction `g` ne prend aucun argument.

Cette déclaration partielle laissait au compilateur le soin de compléter la déclaration lors de l'appel de la fonction, ou de sa définition. On perd donc un grand intérêt des prototypes. Mais à cause de *l'immense* quantité de code existant qui se reposait sur ce comportement, l'ANSI (puis le WG14) n'ont pas interdit de tels programmes, mais ont déclaré dès le C90 que cette construction est *obsolète*.

Évaluation des arguments

La norme du langage **ne spécifie pas** l'ordre d'évaluation des arguments. Il faut donc faire particulièrement attention aux effets de bords.

⚠ Ce code contient **une erreur volontaire** !

```
#include <stdio.h>

int somme(int a, int b)
{return a + b;
}

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

Voici un premier exemple. Lors de l'appel de la fonction `somme`, si l'expression `++i` est évaluée avant l'expression `i`, alors le programme affichera *2*. Si, au contraire, c'est l'expression `i` qui est évaluée avant l'expression `++i`, alors le programme affichera *1*.

⚠ Ce code contient **une erreur volontaire** !

```
#include <stdio.h>
int fonction(int, int);
int g(void);
int h(void);

int test(void)
{
    return fonction(g(), h());
}
```

Dans cet autre exemple, les expressions `g()` et `h()` pouvant être évaluées dans n'importe quel ordre, on ne peut pas savoir laquelle des fonctions `g` et `h` sera appelée en premier. Si l'appel de ces fonctions provoque des effets de bord

(affichage de messages, modification de variables globales...), alors le comportement du programme est imprévisible. Pour pallier à ce problème, il faut imposer l'ordre d'appel :

```
#include <stdio.h>
int fonction(int, int);
int g(void);
int h(void);

int test(void)
{
    int a,b;
    a = g();
    b = h();
    return fonction(a, b);
}
```

Nombre variable d'arguments

Une fonctionnalité assez utile est d'avoir une fonction avec un nombre variable d'arguments, comme la fameuse fonction `printf()`. Pour cela, il suffit de déclarer le prototype de la fonction de la manière suivante :

Déclaration

```
#include <stdarg.h>

void ma_fonction(type1 arg1, type2 arg2, ...)
{
}
```

Dans l'exemple ci-dessus, les points de suspension ne sont pas un abus d'écriture, mais bel et bien une notation C pour indiquer que la fonction accepte d'autres arguments. L'exemple est limité à deux arguments, mais il est bien sûr possible d'en spécifier autant qu'on veut. C'est dans l'unique but de ne pas rendre ambigüe la déclaration, qu'aucun abus d'écriture n'a été employé.

L'inclusion de l'en-tête `<stdarg.h>` n'est nécessaire que pour traiter les arguments à l'intérieur de la fonction. La première remarque que l'on peut faire est qu'une fonction à nombre variable d'arguments contient **au moins un** paramètre fixe. En effet la déclaration suivante est invalide :

⚠ Ce code contient **une erreur volontaire** !

```
void ma_fonction(...);
```

Accès aux arguments

Pour accéder aux arguments situés après le dernier argument fixe, il faut utiliser certaines fonctions (ou plutôt macros) de l'en-tête `<stdarg.h>` :

```
void va_start (va_list ap, last);
type va_arg (va_list ap, type);
void va_end (va_list ap);
```

`va_list` est un type opaque dont on n'a pas à se soucier. On commence par l'initialiser avec `va_start`. Le paramètre `last` doit correspondre au nom du dernier argument fixe de la fonction, ou alors tout bon compilateur retournera au moins un avertissement.

Vient ensuite la collecte *minutieuse* des arguments. Il faut bien comprendre qu'à ce stade, le langage n'offre **aucun** moyen de savoir comment sont structurées les données (c'est à dire leur type). Il faut absolument définir une convention, laissée à l'imagination du programmeur, pour pouvoir extraire les données correctement.

Qui plus est, il faut être extrêmement vigilant lors de la récupération des paramètres, à cause de la promotion des types entiers ou réels. En effet, les entiers sont systématiquement promus en `int`, sauf si la taille du type est plus grande, auquel cas le type est inchangé. Pour les réels, le type `float` est promu en `double`, alors que le type `long double` est inchangé. C'est pourquoi ce genre d'instruction n'a aucun sens dans une fonction à nombre variable d'arguments :

⚠ Ce code contient **une erreur volontaire** !

```
char caractere = va_arg(list, char);
```

Il faut obligatoirement récupérer un entier de type `char`, comme étant un entier de type `int`.

Exemple de convention

Un bon exemple de convention est la fonction `printf()` elle même. Elle utilise un spécificateur de format qui renseigne à la fois le nombre d'arguments qu'on s'attend à trouver mais aussi le type de chacun. D'un autre coté, analyser un spécificateur de format est relativement rébarbatif, et on n'a pas toujours besoin d'une artillerie aussi lourde.

Une autre façon de faire, relativement répandue, est de ne passer que des couples (type, objet), où `type` correspond à un code représentant un type (une énumération par exemple) et `objet` le contenu de l'objet lui-même (`int`, pointeur, `double`, etc.). On utilise alors un code spécial (généralement 0) pour indiquer la fin des arguments, ou alors un des paramètres pour indiquer combien il y en a. Un petit exemple complet :

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

enum va_list{
        TYPE_FIN, TYPE_ENTIER, TYPE_REEL, TYPE_CHAINE
};

void affiche(FILE * out, ...)
{
        va_list list;
        int     type;

        va_start(list, out);

        while ((type = va_arg(list, int)))
        {
                switch (type)
                {
                case TYPE_ENTIER: fprintf(out, "%d", va_arg(list, int)); break;
                case TYPE_REEL:   fprintf(out, "%g", va_arg(list, double)); break;
                case TYPE_CHAINE: fprintf(out, "%s", va_arg(list, char *)); break;
                }
        }
        fprintf(out, "\n");
        va_end(list);
}

int main(int nb, char * argv[])
{
        affiche(stdout, TYPE_CHAINE, "Le code ascii de 'a' est ", TYPE_ENTIER, 'a', 0);
```

```
        affiche(stderr, TYPE_CHAINE, "2 * 3 / 5 = ", TYPE_REEL, 2. * 3 / 5, 0);

        return 0;
}
```

L'inconvénient de ce genre d'approche est de ne pas oublier le marqueur de fin. Dans les deux cas, il faut être vigilant avec les conversions implicites, notamment dans le second cas. À noter que la conversion (transtypage) explicite des types en une taille inférieure à celle par défaut (`int` pour les entiers ou `double` pour les réels) **ne permet pas** de contourner la promotion implicite. Même écrit de la sorte:

```
affiche(stderr, TYPE_CHAINE, "2 * 3 / 5 = ", TYPE_REEL, (float) (2. * 3 / 5), 0);
```

Le résultat transmis au cinquième paramètre sera quand même promu implicitement en type `double`.

Fonction *inline*

Il s'agit d'une extension ISO C99, qui à l'origine vient du C++. Ce mot clé doit se placer avant le type de retour de la fonction. Il ne s'agit que d'une indication, le compilateur peut ne pas honorer la demande, notamment si la fonction est récursive. Dans une certaine mesure, les fonctionnalités proposées par ce mot clé sont déjà prises en charge par les instructions du préprocesseur. Beaucoup préfèreront passer par une macro, essentiellement pour des raisons de compatibilité avec d'anciens compilateurs ne supportant pas ce mot clé, et quand bien même l'utilisation de macro est souvent très délicat.

Le mot clé `inline` permet de s'affranchir des nombreux défauts des macros, et de réellement les utiliser comme une fonction normale, c'est à dire surtout sans effets de bord. À noter qu'il est préférable de classifier les fonctions `inline` de manière statique. Dans le cas contraire, la fonction sera aussi déclarée comme étant accessible de l'extérieur, et donc définie comme une fonction normale.

En la déclarant `static inline`, un bon compilateur devrait supprimer toute trace de la fonction et seulement la mettre *in extenso* aux endroits où elle est utilisée. Ceci permettrait à la limite de déclarer la fonction dans un fichier en-tête, bien qu'il s'agisse d'une pratique assez rare et donc à éviter. Exemple de déclaration d'une fonction `inline` statique :

```
static inline int max(int a, int b)
{
        return (a > b) ? a : b;
}
```

La fonction main

Nous allons revenir ici sur la fonction `main`, présente dans chaque programme. Cette fonction est le point d'entrée du programme. La norme définit deux prototypes, qui sont donc portables:

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

Le premier prototype est plus "général" : il permet de récupérer des paramètres au programme. Le deuxième existe pour des raisons de simplicité, quand on ne veut pas traiter ces arguments.

Si ces deux prototypes sont portables, une implémentation peut néanmoins définir un autre prototype pour `main`, ou

spécifier une autre fonction pour le démarrage du programme. Cependant ces cas sont plus rares (et souvent spécifiques à du C embarqué).

Paramètres de ligne de commande

La fonction `main` prend deux paramètres qui permettent d'accéder aux paramètres passés au programme lors de son appel. Le premier, généralement appelé `argc` (*argument count*), est le nombre de paramètres qui ont été passés au programme. Le second, `argv` (*argument vector*), est la liste de ces paramètres. Les paramètres sont stockés sous forme de chaîne de caractères, `argv` est donc un tableau de chaînes de caractères, ou un pointeur sur un pointeur sur `char`. `argc` correspond au nombre d'éléments de ce tableau.

La première chaîne de caractères, dont l'adresse est dans `argv[0]`, contient le nom du programme. Le premier paramètre est donc `argv[1]`. Le dernier élément du tableau, `argv[argc]`, est un pointeur nul.

Valeur de retour

La fonction `main` retourne toujours une valeur de type entier. L'usage veut qu'on retourne 0 (ou `EXIT_SUCCESS`) si le programme s'est déroulé correctement, ou `EXIT_FAILURE` pour indiquer qu'il y a eu une erreur (Les macros `EXIT_SUCCESS` et `EXIT_FAILURE` étant définies dans l'en-tête `<stdlib.h>`). Il est possible par le programme appelant de récupérer ce code de retour, et de l'interpréter comme bon lui semble.

Exemple

Voici un petit programme très simple qui affiche la liste des paramètres passés au programme lorsqu'il est appelé:

```c
#include <stdio.h>

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

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

        return 0;
}
```

On effectue une boucle sur `argv` à l'aide de `argc`. Enregistrez-le sous le nom `params.c` puis compilez-le (`cc params.c -o params`). Vous pouvez ensuite l'appeler ainsi:

```
./params hello world ! # sous Unix
params.exe hello world ! # sous MS-DOS ou Windows
```

Vous devriez voir en sortie quelque chose comme ceci (*paramètre 0* varie selon le système d'exploitation):

```
paramètre 0 : ./params
paramètre 1 : hello
paramètre 2 : world
paramètre 3 : !
```

Fonctions en C pré-ANSI

Absence de prototype lors de l'appel d'une fonction

Avant la normalisation C89, on pouvait appeler une fonction sans disposer ni de sa définition, ni de sa déclaration. Dans ce cas, la fonction était *implicitement* déclarée comme retournant le type `int`, et prenant un nombre indéterminé de paramètres.

```
/* Aucune déclaration de g() n'est visible. */

void f(void)
{
    g();  /* Déclaration implicite: extern int g() */
}
```

À cause de la grande quantité de code existant à l'époque qui reposait sur ce comportement, le C90 a conservé cette possibilité. Cependant, elle a été retirée de la norme C99, et est à éviter même lorsqu'on travaille en C90.

En effet, c'est plus qu'une bonne habitude de programmation de s'assurer que *chaque* fonction utilisée dans un programme ait son prototype déclaré **avant** qu'elle ne soit définie ou utilisée. C'est d'autant plus indispensable lorsque les fonctions sont définies et utilisées dans des fichiers différents.

Ancienne notation

À titre anecdotique, ceci est la façon « historique » de définir une fonction, avant que le prototypage ne fut utilisé. Cette notation est interdite depuis C90.

```
type_retour fonction(par1, par2, ..., parN)
type1 par1;
type2 par2;
...
typeN parN;
{
        /* Déclarations de variables ... */
        /* Instructions ... */
}
```

Au lieu de déclarer les types à l'intérieur même de la fonction, ils sont simplement décrits après la fonction et avant la première accolade ouvrante. À noter que `type_retour` *pouvait* être omis, et dans ce cas valait par défaut `int`.

Fichiers

Exécuter des fichiers

Les primitives sont *execl*, *execlp*, *execle*, *exect*, *execv*, *execvp*[14].

L'exemple ci-dessous lance un fichier *test.exe* sur un bureau de Windows 7 :

```
#include <unistd.h>

int main()
{
     int fichier;
     fichier = execl ( "c:\\Users\\public\\Desktop\\test.exe", "test.exe", ".", (char*)0);
}
```

Renommer et supprimer des fichiers

Utiliser les fonctions *rename* et *remove*[15].

Copier des fichiers

Il faut copier le contenu du premier dans le second[16].

Tableaux

Il existe deux types de tableaux : les tableaux statiques, dont la taille est connue à la compilation, et les tableaux dynamiques, dont la taille est connue à l'exécution. Nous ne nous intéresserons pour l'instant qu'aux tableaux statiques, les tableaux dynamiques seront présentés avec les pointeurs.

Syntaxe

Déclaration

```
T tableau[N];
```

Déclare un *tableau* de *N* éléments, les éléments étant de type *T*. *N* doit être un nombre connu à la compilation, N ne peut pas être une variable. Pour avoir des tableaux dynamiques, il faut passer par l'allocation mémoire.

Cette dernière restriction a été levée par la norme C99 qui a apporté le type tableau à longueur variable appelés **Variable Length Array** (**VLA**). il est maintenant possible de déclarer un tableau de type VLA par :

```
T tableau[expr];
```

où *expr* est une expression entière, calculée à l'exécution du programme. Ce type de tableau **ne peut pas** appartenir à la classe de stockage **static** ou **extern**. Sa portée est limitée au bloc dans lequel il est défini. Par rapport à un allocation dynamique, les risques de fuite mémoire sont supprimés.

Accès aux éléments

L'exemple précédent a permis de déclarer un tableau à N éléments. En C, ces éléments sont indexés de 0 à N-1, et le i^e élément peut être accédé de la manière suivante (où `i` est supposé déclaré comme une variable entière ayant une valeur entre 0 et N-1):

```
tableau[i]
```

Note: La syntaxe suivante est équivalente à la précédente:

```
i[tableau]
```

Bien qu'autorisée par le C, elle va à l'encontre de ce à quoi nombre de programmeurs sont habitués dans d'autres langages, c'est pourquoi elle est **très peu** utilisée. Comme, de plus, elle n'apporte aucun réel avantage par rapport à la première syntaxe, elle est simplement à éviter, au profit de la précédente.

Exemples

```
int i;
int tableau[10]; /* déclare un tableau de 10 entiers */

for (i = 0; i < 10; i++) /* boucle << classique >> pour le parcours d'un tableau */
{
    tableau[i] = i;  /* chaque case du tableau reçoit son indice comme valeur */
}
```

⚠ Ce code contient **une erreur volontaire** !

```
int tableau[1]; /* déclare un tableau d'un entier */

tableau[10] = 5; /* accède à l'élément d'indice 10 (qui ne devrait pas exister) */

printf("%d\n", tableau[10]);
```

Ce deuxième exemple, non seulement peut compiler (le compilateur peut ne pas détecter le dépassement de capacité), mais peut aussi s'exécuter et afficher le « bon » résultat. **Le langage C n'impose pas à une implémentation de vérifier les accès, en écriture comme en lecture, hors des limites d'un tableau** ; il précise explicitement qu'un tel code a un *comportement indéfini*, donc que n'importe quoi peut se passer. En l'occurrence, ce code peut très bien marcher comme on pourrait l'attendre, i.e. afficher `5`, ou causer un arrêt du programme avec erreur (si la zone qui correspondrait au 11ème élément du tableau est hors de la mémoire allouée au processus, le système d'exploitation peut détecter une tentative d'accès invalide à une zone mémoire, ce qui peut se traduire par une « erreur de segmentation » qui termine le programme), ou encore corrompre une autre partie de la mémoire du processus (dans le cas où le pseudo-11ème élément correspondrait à une partie de la mémoire du processus), ce qui peut modifier son comportement ultérieur de manière très difficile à prévoir.

Ce code est donc un exemple d'un type de bogue très courant en langage C. S'il est facile à détecter dans ce code très court, il peut être très difficile à identifier dans du code plus complexe, où on peut par exemple essayer d'accéder à un indice `i` d'un tableau, la valeur de `i` pouvant varier suivant un flot d'exécution complexe, ou alors essayer de copier le contenu d'un tableau dans un autre, sans vérifier la taille du tableau de destination (ce dernier cas est connu sous le nom de *débordement de tampon*, *dépassement de capacité*, ou encore *buffer overflow* en anglais). **Il est donc très important de s'assurer que tous les accès au contenu de tableaux, en lecture comme en écriture, se font dans les limites de ses bornes.**

Tableaux à plusieurs dimensions

Les tableaux vus pour l'instant étaient des tableaux à une dimension (*ie* : des tableaux à un seul indice), il est possible de déclarer des tableaux possédant un nombre aussi grand que l'on veut de dimensions. Par exemple pour déclarer un tableau d'entiers à deux dimensions :

```
int matrice[10][5];
```

Pour accèder aux éléments d'un tel tableau, on utilise une notation similaire à celle vue pour les tableaux à une dimension :

```
matrice[0][0] = 5;
```

affecte la valeur 5 à la case d'indice (0,0) du tableau.

Avec le type VLA (Variable Length Array), introduit par C99, il est possible de définir un tableau à plusieurs dimensions dont les bornes ne sont connues qu'à l'exécution et non à la compilation :

```
static void vlaDemo(int taille1, int taille2)
{
        // Declaration du tableau VLA
        int table[taille1][taille2];

        // ...
```

Initialisation des tableaux

Il est possible d'initialiser directement les tableaux lors de leur déclaration :

```
int tableau[5] = { 1 , 5 , 45 , 3 , 9 };
```

initialise le *tableau* d'entiers avec les valeurs fournies entre accolades (`tableau[0] = 1;`, `tableau[1] = 5;`, etc.)

À noter que si on ne spécifie aucune taille entre les crochets du tableau, le compilateur la calculera automatiquement pour contenir tous les éléments. La déclaration ci-dessus aurait pu s'écrire plus simplement :

```
int tableau[] = { 1 , 5 , 45 , 3 , 9 };
```

Si on déclare un tableau de taille fixe, mais qu'on l'initialise avec moins d'éléments qu'il peut contenir, les éléments restant seront mis à zéro. On utilise cette *astuce* pour initialiser rapidement un tableau à zéro :

```
int tableau[512] = {0};
```

Cette technique est néanmoins à éviter, car dans ce cas le tableau sera stocké en entier dans le code du programme, faisant grossir artificiellement la taille du programme exécutable. Alors qu'en ne déclarant qu'une taille et l'initialisant à la main au début du programme, la plupart des formats d'exécutable sont capables d'optimiser en ne stockant que la taille du tableau dans le programme final.

Néanmoins, cette syntaxe est aussi utilisable pour les tableaux à plusieurs dimensions :

```
int matrice[2][3] = { { 1 , 2 , 3 } , { 4 , 5 , 6 } };
```

Ce qui peut aussi s'écrire :

```
int matrice[2][3] = { 1 , 2 , 3 , 4 , 5 , 6 };
```

À noter que si on veut aussi utiliser l'adaptation dynamique, il faut quand même spécifier une dimension :

```
int identite[][3] = { { 1 , 0 , 0 } , { 0 , 1 , 0 } , { 0 , 0 , 1 } };
/* Ou plus "simplement" */
int identite[][3] = { { 1 } , { 0 , 1 } , { 0 , 0 , 1 } };
```

Conversion des noms de tableaux en pointeurs

A quelques exceptions près (c.f. ci-dessous), un nom de tableau apparaissant dans une expression C sera automatiquement converti en un pointeur vers son premier élément lors de l'évaluation de cette expression : si `tab` est le nom d'un tableau d'entiers, le *nom de tableau* `tab` sera converti dans le type pointeur vers `int` dans *l'expression* tab, et la valeur de cette expression sera l'adresse du début de la zone mémoire allouée pour le stockage des éléments du tableau, i.e. l'adresse de son premier élément. Le code suivant est par exemple valide :

```
int tab[10];
int *p, *q;
p = tab;
q = &(tab[0]);
```

La déclaration `int tab[10]` réserve une zone mémoire suffisante pour stocker 10 variables de type `int`, désignées par `tab[0]`, `tab[1]`..., `tab[9]`. Dans l'instruction `p = tab`, l'expression `tab` est de type pointeur vers `int`, et la valeur affectée à `p` est l'adresse de la première case du tableau. Cette adresse est aussi l'adresse de la variable `tab[0]` : dans la seconde instruction, l'adresse de `tab[0]` est affectée à `q` via l'opérateur de prise d'adresse `&`. Après la seconde instruction, `p` et `q` ont la même valeur, et les instructions `*p = 1`, `*q = 1` ou `tab[0] = 1` deviennent opératoirement équivalentes.

Exceptions à la règle de conversion

Noter que dans la déclaration `int tab[10]`, le nom `tab` ne désigne pas en lui-même un pointeur mais bien un *tableau*, de type *tableau d'entiers à 10 éléments*, même si ce nom est converti en pointeur dans la plupart des contextes. Les exceptions à la règle de conversion interviennent :

- lorsque le nom du tableau est opérande de l'opérateur unaire `&`,
- lorsqu'il est argument de l'opérateur `sizeof`,
- lorsqu'il est argument d'un pré ou d'une post-incrémentation ou décrémentation (`--`,`++`)
- lorsqu'il constitue la partie gauche d'une affectation,
- lorsqu'il est suivi de l'opérateur d'accès à un champ.

Dans chaque cas, le nom du tableau gardera le sens d'une variable de type tableau, c'est-à-dire conservera son type initial :

- dans l'expression `&tab`, la sous-expression `tab` conserve son type "tableau d'entiers à 10 éléments", et l'expression elle-même est de type "pointeur vers tableaux d'entiers à 10 éléments". Le code suivant est par exemple valide :

```
int tab[10];  /* tableau d'entiers à 10 éléments */
int (*p)[10]; /* pointeur vers les tableaux d'entiers à 10 éléments */
p = &tab;
```

Le code suivant est en revanche incorrect :

⚠ Ce code contient **plusieurs erreurs volontaires** !

```c
int tab[10];
int *p;
p = &tab; /* incorrect. '&tab' n'est pas du bon type  */
```

- dans `sizeof tab` la taille calculée sera celle de `tab` considéré encore comme une variable de type "tableau d'entiers à 10 éléments", soit 10 fois la taille d'un `int`.
- dans `tab++` , `tab--` , `--tab` ou `++tab` , la sous-expression `tab` n'est ni une expression numérique ni un pointeur, mais bien une variable de type tableau : elle n'est donc ni incrémentable, ni décrémentable, et l'expression globale ne sera jamais compilable,
- dans l'instruction `tab = expr` où `expr` est une expression quelconque, `tab` est toujours considéré comme une variable de type "tableau d'entiers à 10 éléments"... la règle de conversion des noms de tableaux et les restrictions sur les conversion explicites font qu'il est *impossible* que `expr` soit de même type : cette affectation générera toujours une erreur de typage, et ne sera jamais compilable,
- dans l'expression `tab.champ` où `champ` est un nom de champ quelconque, `tab` est de type tableau, i.e. ne possède aucun champ, donc cette expression ne sera jamais compilable.

Le cas où `tab` est membre gauche d'une affectation montre que les valeurs de type tableau ne sont jamais manipulables de manière directe : il est par exemple impossible d'affecter un tableau à un autre tableau - l'instruction `tab_1 = tab_2` est toujours incompilable, à cause de la conversion en pointeur de `tab_2` - ou de comparer structurellement deux tableaux par une comparaison simple - dans `tab_1 == tab_2` , les deux noms seront convertis en pointeur, et la comparaison effectuée sera celle des adresses.

Cas des paramètres de fonctions

Tout paramètre de fonction déclaré comme étant de type "tableau d'éléments de type T" est automatiquement converti en un pointeur vers T à la compilation. Les deux écritures suivantes sont équivalentes :

version avec tableaux

```c
void f(int t[])
{
   /* ... */
}
void g(int m[][10])
{
   /* ... */
}
```

version avec pointeurs

```c
void f(int *t)
{
   /* ... */
}
void g(int (*m)[10])
{
   /* ... */
}
```

Dans la première version de `f` et `g` , les paramètres `t` et `m` sont bien des pointeurs, conformément à l'équivalence avec la seconde version, et sont donc réaffectables dans le corps de ces fonctions. La taille en la première dimension de `t` et de `m` , même si elle est spécifiée dans le code (e.g. `void g(int m[10][10])`), sera de toute manière effacée à la compilation : il ne s'agit alors que d'un simple commentaire pour le programmeur.

Cette équivalence est cohérente avec la règle de conversion des noms de tableaux : lors d'un appel de la forme `f(tab)` où `tab` est un tableau d'entiers, le nom `tab` sera converti en pointeur vers `int` , qui est bien le type du paramètre de `f` dans la seconde version.

Cas des tableaux externes

Dans le cas des déclarations externes, et contrairement au cas des paramètres de fonctions, il n'y a *pas* équivalence entre la notation par tableaux et la notation par pointeurs. Le programme suivant est en général compilable, mais incorrect :

⚠ Ce code contient **une erreur volontaire** !

fichier1.c	**fichier2.c**

```c
#include <stdio.h>
int tab[] = {42};

void f(void)
{
  printf("fichier1 : tab = %p\n", tab);
}
```

```c
#include <stdio.h>
extern int *tab;
void f(void)

int main(int nb, char *argv[] )
{
    f();
    printf("fichier2 : tab = %p\n", tab);
    return 0;
}
```

Les valeurs affichées pour `tab` dans `main` et dans `f` seront en général distinctes. L'erreur se situe dans la déclaration externe de `tab` située dans le fichier `fichier2.c`. Il aurait fallu écrire la déclaration suivante, spécifiant correctement le type de `tab` dans `fichier2.c` comme étant celui d'un tableau et non d'un pointeur (mais ne spécifiant pas sa taille, le compilateur n'effectuant de toute manière aucun test de dépassement) :

```c
extern int tab[];
```

L'absence de message d'erreur dans la compilation de la première version de ce programme n'est due qu'à l'insuffisance de la vérification de la cohérence globale du typage lors de la phase de liaison : il est plus généralement possible de déclarer une variable externe de n'importe quel autre type que son type réel, sans que l'incohérence globale du typage soit nécessairement signalée par le compilateur. Aucune norme ne spécifiant le comportement du programme résultant, ce comportement est, de fait, indéfini.

Tableaux VLA passés à une fonction

En C99, il est possible de passer à une fonction un tableau de type VLA (Variable Length Array). Il faut passer les tailles des dimensions en premier.

Exemple de fonction recevant en argument un tableau de type VLA à deux dimensions :

```c
static void vlaPrint(int taille1, int taille2, int table[taille1][taille2])
{
        // Ecriture du tableau sur stdout
        for (int i=0; i< taille1; i++)
        {
                for (int j=0; j< taille2; j++)
                {
                        (void)printf("table[%d][%d] = %d\n", i, j, table[i][j]);
                }
        }
}
```

Chaînes de caractères

Comme il a été dit tout au long de cet ouvrage, les chaînes de caractères sont des tableaux particuliers. En déclarant une chaîne de caractères on peut soit la manipuler en tant que pointeur soit en tant que tableau. Considérez les déclarations suivantes :

```c
char * chaine1 = "Ceci est une chaine";
char chaine2[] = "Ceci est une autre chaine";
```

Bien que se manipulant exactement de la même façon, les opérations permises sur les deux variables ne sont pas tout à fait les mêmes. Dans le premier cas on déclare un pointeur sur une chaîne de caractères statique, dans le second cas, un tableau (alloué soit sur la pile si la variable est déclarée dans une fonction ou soit dans le segment global si la variable est globale) de taille suffisante pour contenir tous les caractères de la chaîne affectée (incluant le caractère nul).

Le second cas est donc une notation abrégée pour `char chaine2[] = { 'C', 'e', 'c', 'i', ' ', ..., 'e', '\0' };`. Dans tous les autres cas (ceux où une chaîne de caractères **ne sert pas** à initialiser un tableau), **la chaîne déclarée est statique** (les données sont persistantes entre les différents appels de fonctions), et sur certaines architectures, pour ne pas dire toutes, elle est même en **lecture seule**. En effet l'instruction `chaine1[0] = 'c';` peut provoquer un accès illégal à la mémoire. En fait le compilateur peut optimiser la gestion des chaînes en regroupant celles qui sont identiques. C'est pourquoi il est préférable de classifier les pointeurs sur chaîne de caractères avec le mot clé `const`.

On comprend aisément que si `chaine2` est alloué sur la pile (déclaré dans une fonction), la valeur **ne pourra pas** être utilisée comme valeur de retour. Tandis que la valeur de `chaine1` pourra être retournée même si c'est une variable locale.

Pointeurs

Dans cette section, nous allons présenter un mécanisme permettant de manipuler les adresses, les **pointeurs**. Un pointeur a pour valeur l'adresse d'un objet C d'un type donné (un pointeur est typé). Ainsi, un pointeur contenant l'adresse d'un entier sera de type *pointeur vers entier*.

Usage et déclaration

L'opérateur **&** permet de connaitre l'adresse d'une variable, on dira aussi la référence. Toute déclaration de variable occupe un certain espace dans la mémoire de l'ordinateur. La référence permet de savoir où cet emplacement se trouve. En simplifiant à l'extrême, on peut considérer la mémoire d'un ordinateur comme une gigantesque table d'octets. Quand on déclare une variable de type `int`, elle sera allouée à un certain emplacement (ou dit autrement : un indice, une adresse ou une référence) dans cette table. Un pointeur permet simplement de stocker une référence, il peut donc être vu comme un nombre allant de 0 à la quantité maximale de mémoire dont dispose votre ordinateur (moins un, pour être exact).

Un pointeur occupera habituellement toujours la même taille (occupera la même place en mémoire), quelque soit l'objet se trouvant à cet emplacement. Il s'agit en général de la plus grande taille directement gérable par le processeur : sur une architecture 32bits, elle sera de 4 octets, sur une architecture 64bits, 8 octets, etc. Le type du pointeur ne sert *qu'à* renseigner comment sont organisées les données suivant l'adresse référencée par le pointeur. Ce code, par exemple, affiche la référence d'une variable au format hexadécimal :

```
int i;
printf("%p\n", &i);
```

Pouvoir récupérer l'adresse n'a d'intérêt que si on peut manipuler l'objet pointé. Pour cela, il est nécessaire de pouvoir déclarer des pointeurs, ou dit autrement un objet pouvant contenir des références. Pour cela on utilise l'étoile (*) entre le type et le nom de la variable pour indiquer qu'il s'agit d'un pointeur :

```
T * pointeur, * pointeur2, /* ..., */ * pointeurN;
```

Déclare les variables *pointeur*, *pointeur2*, ..., *pointeurN* de type pointeur vers le type *T*. À noter la bizarrerie du langage à vouloir associer l'étoile à la variable et non au type, qui oblige à répéter l'étoile pour chaque variable.

```
/* Ce code contient une déclaration volontairement confuse */
int * pointeur, variable;
```

Cet exemple de code déclare un *pointeur sur un entier de type* `int` et une variable de type `int`. Dans un vrai programme, il est rarement possible d'utiliser des noms aussi triviaux, aussi il est recommandé de séparer la déclaration des variables de celles des pointeurs (ou d'utiliser l'instruction `typedef`, qui, elle, permet d'associer l'étoile au type), la lisibilité du programme sera légèrement améliorée.

Il est essentiel de bien comprendre ce qui a été déclaré dans ces exemples. Chaque pointeur peut contenir une référence sur un emplacement de la mémoire (un indice dans notre fameuse table). On peut obtenir une référence (ou un indice) avec l'opérateur **&** (ou allouer une référence soi-même avec des *fonctions* dédiées, c.f la section suivante). Cet opérateur transforme donc une *variable* de type `T` en un pointeur de type `T *`. Insistons sur le terme variable, car évidemment des expressions telles que '`&23435`' ou '`&(2 * a / 3.)`' n'ont aucun sens, dans la mesure où les constantes et expressions du langage n'occupent aucun emplacement susceptible d'intéresser votre programme.

Ce code, par exemple, affiche la référence d'une variable dans un format défini par l'implémentation (qui peut être hexadécimal, ou une combinaison "segment:offset", par exemple) :

```
int i;

printf("%p\n", &i);
```

Il ne faut pas oublier que, comme toutes les variables locales en C, un pointeur est à l'origine non initialisé. Une bonne attitude de programmation est de s'assurer que lorsqu'il ne pointe pas vers un objet valide, sa valeur est mise à zéro (ou `NULL`, qui est déclaré entre autre dans `<stdio.h>`).

L'arithmétique des pointeurs

L'arithmétique associée aux pointeurs est sans doute ce qui a valu au C sa réputation « d'assembleur plus compliqué et plus lent que l'assembleur ». On peut très vite construire des expressions incompréhensibles avec les opérateurs disponibles. Dans la mesure du possible, il est conseillé de se limiter à des expressions simples, quitte à les décomposer, car la plupart des compilateurs savent très bien optimiser un code C.

Déréférencement

Il s'agit de l'opération la plus simple sur les pointeurs. Comme son nom l'indique, il s'agit de l'opération réciproque au référencement (`&`). L'opérateur associé est l'étoile (`*`), qui est aussi utilisé pour déclarer un type pointeur. Cet opérateur permet donc de transformer un pointeur de type *T* *, en un objet de type *T*, les opérations affectant l'objet pointé :

```
int variable = 10;
int * pointeur = &variable;

*pointeur = 20; /* Positionne 'variable' à 20 */
```

Ici, `pointeur` contient une adresse valide, celle de `variable` ; son déréférencement est donc possible. Par contre, si `pointeur` était une variable locale non initialisée, son déréférencement provoquerait à coup sûr un arrêt brutal de

votre programme.

Vous obtiendrez le même résultat, si `pointeur` est initialisé à `NULL`. Cette adresse est invalide et toute tentative de déréférencement se soldera par un arrêt du programme.

Arithmétique de base

L'arithmétique des pointeurs s'apparente à celle des entiers, mais il est important de comprendre la distinction entre ces deux concepts.

Les opérations arithmétiques permises avec les pointeurs sont :

- Addition / soustraction d'une valeur entière à un pointeur (on avance / recule d'un nombre de cases mémoires égal à la taille du type *T*) : le résultat est donc un pointeur, de même type que le pointeur de départ. Il faut bien faire attention avec ce genre d'opération à ne pas sortir du bloc mémoire, car le C n'effectuera aucun test pour vous. Considérez l'exemple suivant :

```
/* Parcours les éléments d'un tableau */
int tableau[N];
int * p;

for (p = tableau; p < &tableau[N]; p ++)
{
     /* ... */
}
```

  Normalement un tableau de N cases permet d'être itéré sur les indices allant de 0 à N - 1, inclusivement. L'expression `&tableau[N]` fait référence la case mémoire non allouée immédiatement après le plus grand indice, donc potentiellement source de problème. Toutefois, par exception pour le premier indice après le plus grand, C garantit que le résultat de l'expression soit bien défini. Bien sûr, il ne faut pas déréférencer ce pointeur.

  À noter qu'à l'issue de la boucle, p pointera sur la N+1ème case du tableau, donc hors de l'espace alloué. Le C autorise tout à fait ce genre de pratique, il faut juste faire attention à ne pas déréférencer le pointeur à cet endroit.

- Soustraction de deux pointeurs de même type (combien d'objet de type *T* y a t-il entre les deux pointeurs) : le résultat est donc un **entier**, de type `ptrdiff_t`.

  ⚠ Ce code contient **une erreur volontaire** !

```
int       autre_tableau[3];
int       tableau[10];
int *     p     = &tableau[5];  /* p pointe sur le 6e élément du tableau */
int *     q     = &tableau[3];  /* q pointe sur le 4e élément du tableau */
ptrdiff_t diff1 = p - q;        /* diff1 vaut 2 */
ptrdiff_t diff2 = q - p;        /* diff2 vaut -2 */

q = &autre_tableau[2];
ptrdiff_t dif3 = p - q; /* Erreur ! */
```

  Dans cet exemple, les deux premières soustractions sont définies, car p et q pointent sur des éléments du même tableau. La troisième soustraction est indéfinie, car on utilise des adresses d'éléments de tableaux différents.

  Notons que l'opérateur `[]` s'applique toujours à une opérande de type entier et une autre de type pointeur. Lorsqu'on écrit `tableau[i]`, il y a en fait une conversion de tableau à pointeur avec l'application de l'opérateur `[]`. On peut donc bien sûr utiliser l'opérateur `[]` avec un pointeur pour opérande :

```
int   a;
int b;
```

```
int * p = &a; /* On peut accéder à la valeur de 'a' via 'p[0]' ou '*p' */

/* p[1] est indéfini - n'espérez pas accéder à la valeur de b depuis l'adresse de a */
```

Arithmétique avec effet de bord

C'est sans doute ce qui a donné des sueurs froides à des générations de programmeurs découvrant le C : un usage « optimisé » de la priorité des opérateurs, le tout imbriqué dans des expressions à rallonge. Par exemple '`while( *d++ = *s++ );`', pour copier une chaine de caractères.

En fait, en décomposant l'instruction, c'est nettement plus simple qu'il ne parait. Par exemple :

```
int i;
int * entier;

/* ... */

i = *entier++; /* i = *(entier++); */
```

Dans ce cas de figure, l'opérateur d'incrémentation ayant priorité sur celui de déréférencement, c'est celui-ci qui sera appliqué en premier. Comme il est postfixé, l'opérateur ne prendra effet qu'à la fin de l'expression (donc de l'affectation). La variable *i* sera donc tout simplement affectée de la valeur pointée par *entier* et après cela le pointeur sera incrémenté. Voici les différents effets suivant les combinaisons de ces deux opérateurs :

```
i = *++entier; /* Incrémente d'abord le pointeur, puis déréférence la nouvelle adresse pointée */
i = ++*entier; /* Incrémente la valeur pointée par "entier", puis affecte le résultat à "i" */
i = (*entier)++; /* Affecte la valeur pointée par "entier" et incrémente cette valeur */
```

On peut évidemment complexifier les expressions à outrance, mais privilégier la compacité au détriment de la clarté et de la simplicité dans un hypothétique espoir d'optimisation est une erreur de débutant à éviter.

Le pointeur `void *`

Ce pointeur est un cas particulier. Il permet de pointer sur un type quelconque. Il est notemment utilisé dans la fonction malloc():

```
void * malloc(int n);
```

En pratique, il faut penser à transformer ce pointeur pour qu'il devienne utilisable, même si certains compilateurs acceptent de l'utiliser directement:

```
// Allocation avec conversion
int * p; // Pointeur p sur le type int
p = (int *) malloc(sizeof(int) * 10); // Allocation de 10 int, soit 20 octets
*p = 4; // Modification

// Allocation sans conversion
int * p; // Pointeur p sur le type int
p = malloc(sizeof(int) * 10); // Allocation de 10 int, soit 20 octets
*p = 4; // Modification
```

Tableaux dynamiques

Un des intérêts des pointeurs et de l'allocation dynamique est de permettre de décider de la taille d'une variable au moment de l'exécution, comme par exemple pour les tableaux. Ainsi pour allouer un tableau de n entiers (n étant connu à l'exécution), on déclare une variable de type pointeur sur entier à laquelle on alloue une zone mémoire correspondant à n entiers :

```
int * alloue_tableau(int n, size_t taille)
{
    return malloc(n * taille);
}

/* Ailleurs dans le programme */
int * tableau = alloue_tableau(256, sizeof *tableau);
if (tableau  != NULL)
{
    /* opérations sur le tableau */
    /* ... */
    free( tableau );
}
```

Cet exemple alloue un tableau de 256 cases. Bien que la variable soit un pointeur, il est dans ce cas permis d'accéder aux cases de 0 à 255, soit entre les adresses `&tableau[0]` et `&tableau[255]`, incluses.

Tableaux dynamiques à deux dimensions

Tout comme on pouvait allouer des tableaux statiques à plusieurs dimensions, on peut allouer des tableaux dynamiques à plusieurs dimensions. Pour ce faire, on commence là-aussi par déclarer un pointeurs approprié : un pointeur sur des pointeurs (etc.) sur des types. Pour déclarer un tableau dynamique d'entiers à deux dimensions :

```
int ** matrice;
```

L'allocation d'un tel objet va se dérouler en plusieurs étapes (une par étoile), on alloue d'abord l'espace pour un tableau de pointeurs vers entier. Ensuite, on alloue pour chacun de ces tableaux l'espace pour un tableau d'entiers. Si on veut une matrice 4x5 :

```
#define LIGNES 4
#define COLONNES 5
int i;
int ** matrice = malloc(sizeof *matrice * LIGNES);

for (i = 0; i < LIGNES; i++)
{
        matrice[i] = malloc(sizeof **matrice * COLONNES);
}
```

Il ne faut jamais oublier de libérer la mémoire allouée précédemment. Ainsi, à tout appel de `malloc` doit correspondre un appel de `free` Pour libérer l'espace alloué ci-dessus, on procède de manière inverse, en commençant par libérer chacune des lignes du tableau, puis le tableau lui même :

```
for(i = 0; i < LIGNES; i++)
{
        free(matrice[i]);
}
free(matrice);
```

Tableaux dynamiques à trois dimensions

Voici maintenant un exemple d'une fonction créant un tableau à trois dimensions avec surtout, et c'est très important, les tests des valeurs de retour des fonctions `malloc` :

```
int *** malloc_3d(int nb_tableau, int lignes, int colonnes)
{
    int i;
    int j;

    int***t = malloc(sizeof(*t) * nb_tableau);
    /* première dimension */
    if (t==NULL)
        {
            printf ("Impossible d'initialiser avec malloc\n" );
            exit (-1);
        }

    for (i=0;i< nb_tableau;i++) {
        t[i] = malloc(sizeof(**t) * lignes);
        /* deuxième dimension */
        if (t[i]==NULL) {
            printf ("Impossible d'initialiser avec malloc\n" );
            exit (-1);
        }
        for (j=0;j<lignes;j++) {
            /* troisième dimension */
            t[i][j] = malloc(sizeof(***t) * colonnes);
            if (t[i][j]==NULL)
                {
                    printf ("Impossible d'initialiser avec malloc\n" );
                    exit (-1);
                }
        }
    }
    return t;
}
```

Utilisation des pointeurs sur des tableaux particuliers

Il est possible avec un pointeur de lire/parcourir les éléments d'une structure. Chaque éléments d'une structure utilise un espace qui permet de calculer des déplacements.

```
#include <stdio.h>
#include <string.h>
struct s{
 int a;
 int b;
 char *s;
}s;
int
main(void)
{
 struct s st={1,1,"salut"};
 void *p=&st;
 printf ("%s\n",
          (char *)((void **)p)[2]);
 /*<==OU==>*/
 p += 2*sizeof(int);
 printf ("%s\n",
          (char *)((void **)p)[0]);
 /*<=DE LA MEME MANIERE=>*/
 p -= 2*sizeof(int);
 /*====================================*/
 memset (p,-1,
         2*sizeof(int));
 printf ("%i,%i\n",
          st.a, st.b);
```

```
  memset ((int *)p,0,
          2*sizeof(int));
  printf ("%i,%i\n",
           st.a, st.b);
  ((int *)p)[0] = 1;
  ((int *)p)[1] = 0;
  printf ("%i,%i\n",
           st.a, st.b);
  printf ("%s\n",st.s);
  return 0;
 }
```

Vous trouverez un exemple complet sur les tableaux et les pointeur ici[1] (http://fr.wikibooks.org/wiki/Discussion:Programmation_C/Pointeurs#Exemple_Complet_sur_les_boucles.2C_les_tableaux_et_les_pointeurs)

Utilisation des pointeurs pour passer des paramètres par adresse

Toutes les variables en C, à l'exception des tableaux, sont passés par valeurs aux paramètres des fonctions. C'est à dire qu'une copie est effectuée sur la pile d'appel. Si bien que toutes les modifications de la variable effectuées dans la fonction seront perdues une fois de retour à l'appelant. Or, il y a des cas où l'on aimerait bien pouvoir modifier une variable passée en paramètre et que ces modifications perdurent dans la fonction appelante. C'est un des usages des paramètres par adresse : permettre la modification d'une variable de l'appelant, comme dans l'exemple suivant :

```
#include <stdio.h>
/* Ce code échange le contenu de deux variables */
void echange(int * a, int *b)
{
        int tmp=0,
                  *t[3]={&tmp,a,b};
        unsigned short int i;
        for (i=0;i<3;i++)
             *t[i] = *t[(i!=2)*(i+1)];
        /*tmp = *a;  <=> *t[0]=*t[1] */
        /**a = *b;  <=>  *t[1]=*t[2] */
        /**b = tmp; <=>  *t[2]=*t[0] */
}

int main(void)
{
    int a = 5;
    int b = 2;

    printf("a = %d, b = %d.\n", a, b);

    /* On passe à 'echange' les adresses de a et b. */
    echange(&a, &b);

    printf("a = %d, b = %d.\n", a, b);

    echange(&a, &b);

    printf("a = %d, b = %d.\n", a, b);

    return 0;
}
```

Ce passage par adresse est extrêmement répandu pour optimiser la quantité de données qui doit transiter sur la pile d'appel (qui est, sur beaucoup de systèmes, de taille fixe). En fait, même si la variable ne doit pas être modifiée, on utilise quand même un passage par adresse, juste pour éviter la copie implicite des variables autres que les tableaux. Ceci est particulièrement intéressant avec les structures, puisque celles-ci ont tendance à être assez imposantes, et cela ne nuit pas trop la lisibilité du programme.

Tableaux dynamiques et passage de tableaux comme arguments d'une fonction

Comme on vient de le voir, l'intérêt principal d'une allocation dynamique est de pouvoir lancer son programme sans connaître la taille du tableau qu'on utilisera; celle-ci sera établie en cours de fonctionnement. Cependant, un autre intérêt du tableau dynamique est de pouvoir être passé comme argument d'une fonction pour des **tableaux à plusieurs dimensions**.

En reprenant la fonction malloc_3d() vue précédemment, on peut écrire :

```
int fonction_3d (int ***tab);

int main(int argc, char **argv)
{
   /* Tableau dynamique créé avec des malloc() */
   int ***tab = malloc_3d(2, 10, 10);

   fonction_3d (tab);

    return EXIT_SUCCESS;
}
/* fonction recevant le tableau en 3d */
void fonction_3d (int ***tab)
{
  /* On peut utiliser ici la notation tab[i][j][k] en veillant à
     ce que i, j, k ne sortent pas des bornes du tableau -> dans
     cet exemple tab[3][9][9] est illégal alors que tab[1][3][7]
     peut être utilisé.
     On ne pourra pas savoir si on sort du tableau !
  */
}
```

Pointeurs vers fonctions

Les pointeurs vers les fonctions sont un peu spéciaux, parce qu'ils n'ont pas d'arithmétique associée (car une telle arithmétique n'aurait pas beaucoup de sens). Les opérations permises avec les pointeurs sur fonctions sont en fait relativement limitées :

```
type_retour (*pointeur_fonction)(liste_paramètres);
```

Déclare *pointeur_fonction*, un pointeur vers une fonction prenant *liste_paramètres* comme paramètres et renvoyant *type_retour*. Le parenthésage est ici obligatoire, sans quoi l'étoile se rattacherait au type de retour. Pour faire pointer un pointeur vers une fonction, on utilise une affectation « normale » :

```
pointeur_fonction = &fonction;
/* Qui est en fait équivalent à : */
pointeur_fonction = fonction;
```

Où *fonction* est compatible avec le pointeur (mêmes paramètres et valeur de retour). Une fois que le pointeur pointe vers une fonction, on peut appeler cette fonction :

```
(*pointeur_fonction)(paramètres);
/* Ou plus simplement, mais moins logique syntaxiquement */
pointeur_fonction(paramètres);
```

Exemple simple d'utilisation de pointeur de fonction avec retour

On fait une comparaison de deux entiers (5 et 4) via un pointeur sur une fonction de comparaison.

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

int comparaison_plus_grand_que(int a, int b)
{
        return a > b ;
}

int main(int argc, char * argv[])
{
        int taille = 10;
        int (*p_comparaison)(int,int); // pointeur de fonction
        int a = 5, b = 4;

        p_comparaison = comparaison_plus_grand_que;
        if((*p_comparaison)(a,b)) // appel de la fonction via le pointeur
                printf("%d est plus grand que %d\n", a ,b);
        else
                printf("%d est plus petit que %d\n", a ,b);

        return EXIT_SUCCESS;
}
```

Pointeurs de fonctions

Exemple sans pointeur

On écrit une fonction graphique.

Cette fonction dessine f.

```
//INITIALISATION :
f est une FORME GRAPHIQUE
f := "_/-\_"
//.
//DECLARATION DE LA FONCTION dessine_f :
fonction dessine_f()
 dessine f
fin de fonction
```

Cela donne le résultat :

```
_/-\_
```

Avec la fonction f qui est fixe, dessine_f() ne sait dessiner que la forme "_/-_".

Si on remplace la valeur intrinsèque de f par un pointeur de fonction, la fonction pourra dessiner n'importe quelle forme.

Même exemple en utilisant un pointeur de fonction

En C, le nom d'une fonction est un pointeur. On peut l'utiliser comme argument :

```
//INITIALISATION :
g, h sont des FORMES GRAPHIQUES
g := "/-\/-\"
h := "--\/--"
//.
//DECLARATION DE LA FONCTION dessine_f :
fonction dessine_f(P_f)
 dessine P_f()
fin de fonction
//.
dessine_f(g)  donnera :"/-\/-\"
dessine_f(h)  donnera :"--\/--"
```

Voir **Exemple graphique (avec Gnuplot)** ci-dessous.

Exemple numérique

- Testé sous Code Block (Windows,Linux).
- Passer deux pointeurs de fonctions à une fonction.

Code source

- Ici on passe les deux fonctions f et g à la fonction f1_o_f2().
- La même fonction peut calculer gof, fog et fof...
- On peut remarquer que les pointeurs de fonctions ont les mêmes types arguments que les fonctions qu'ils vont recevoir.

```c
/* ------------------------------ */
#include    <stdio.h>
#include     <math.h>
/* ------------------------------ */


/* ------ Fonction f ------------ */
double f(double x){return( pow(x,2.));}
/* ------------------------------ */
char  feq[] = "x**2";
/* ------------------------------ */

/* ------ Fonction g ------------ */
double g(double x){return(2.0*x + 3.0);}
/* ------------------------------ */
char  geq[] = "2.0*x + 3.0";
/* ------------------------------ */


/* -Fonction fog  (g suivie de f)-*/
double f1_o_f2(
double (*P_f1)(double x),/* Pointeur pour la première fonction */
double (*P_f2)(double x),/* Pointeur pour la deuxième fonction */
double a
)
{
 return((*P_f1)( ((*P_f2)(a))) );
}
/* ------------------------------ */
```

```
/* ------------------------------ */
int main(void)
{
double a = 2.0;

 printf(" f : x-> %s\n", feq);
 printf(" g : x-> %s\n", geq);
 printf(" \n\n");


   printf(" f(g(%.0f)) = %6.1f\n", a, f1_o_f2(f,g,a));
   printf(" g(f(%.0f)) = %6.1f\n", a, f1_o_f2(g,f,a));
   printf(" f(f(%.0f)) = %6.1f\n", a, f1_o_f2(f,f,a));


 printf("\n\n Press return to continue.\n");
 getchar();

 return 0;
}
```

Résultat ;

```
f : x-> x**2
g : x-> 2.0*x + 3.0
```

```
f(g(2)) = 49.0
g(f(2)) = 11.0
f(f(2)) = 16.0
```

```
Press return to continue.
```

Exemple graphique (avec Gnuplot)

- Testé sous Code Block (Windows,Linux).
- Passer un pointeurs de fonctions à une fonction.
- Vous pourrez approfondir ce travail en analyse avec le cours de Wikiversité: Mathc Home Edition (http://fr.wikiversity.org/wiki/Mathc_Home_Edition) [archive].

Code source

- La fonction Gplt() dessine f(x) et g(x)...
- On peut remarquer que les pointeurs de fonctions ont les mêmes types arguments que les fonctions qu'ils vont recevoir.

```
/* ------------------------------ */
#include    <stdio.h>
#include     <math.h>
/* ------------------------------ */
```

```
/* --- Dessinons f et g --------- */
double f(double x){return(  pow(x,2.));}
double g(double x){return(2.0*x + 3.0);}
/* ------------------------------ */


/* Le fichier de points: [a,f(a)] */
void Gplt(
double (*P_f)(double x)
)
{
FILE   *fp;
double   a;

        fp = fopen("data","w");
   for(a = -5.0; a <= 5.0; a += 0.3)
       fprintf(fp," %6.3f   %6.3f\n",
                     a,      ((*P_f)(a)) );
 fclose(fp);
}


/* ------------------------------ */
int main(void)
{

 printf("f) Dans gnuplot -> plot \"data\" ");
    Gplt(f);
 getchar();

 printf("g) Dans gnuplot -> plot \"data\" ");
    Gplt(g);


 printf("\n\n Press return to continue.\n");
 getchar();

 return 0;
}
```

Tableau de pointeurs de fonctions

Premier exemple

Nous avons des fonctions semblables. Nous voulons les associer pour pouvoir les manipuler dans des boucles.

Nous allons créer un **tableau de pointeurs de fonctions**.

Je vais utiliser les fonctions trigonométriques prédéfinies pour faciliter la lecture.

Déclaration d'un tableau de pointeurs de fonctions

```
double (*TrigF[6])(double x) = {cos,sin,tan,atan,asin,acos};
```

- Toutes les fonctions ont la même forme.
- double FUNCTION(double)
- Le tableau à la même forme que les fonctions.
- double ARRAY(double)
- Il y a six fonctions. (0,1,2,3,4,5)= {cos,sin,tan,atan,asin,acos}.

Exemple d'un appel

```
   cos(.5) = TrigF[0](.5)
```

Exemple à tester

```
/* ------------------------------ */
#include    <stdio.h>
#include     <math.h>
/* ------------------------------ */

int main(void)
{
double (*TrigF[6])(double x) = {cos,sin,tan,atan,asin,acos};

double x= .5;
int    i=  0;

 printf(" Nous avons declare un tableau "\
        " de pointeurs de fonctions.\n  "\
        " J'ai utilise ici les fonctions predefinie du c.\n");


  printf("       cos(%.1f)  = %.3f  \n",  x,     cos(x));
  printf(" TrigF[%d](%.1f)) = %.3f\n\n",i,x,TrigF[i](x));

 printf(" Press return to continue");
 getchar();

 return 0;
}
```

Application

- Créer un tableau de valeurs des fonctions trigonométriques.
- Imprimer le résultat dans cette ordre (sin,cos,tan,acos,asin,atan)
- Pour .1 <= x <+ .5
- Ce travail pourrait être récupéré dans une matrice.

Résultat dans un fichier

```
/* ------------------------------ */
#include    <stdio.h>
#include     <math.h>
/* ------------------------------ */

int main(void)
{
FILE   *fp = fopen("list.txt","w");

double (*TrigF[6])(double x) = {atan,asin,acos,tan,cos,sin};

int    i= 6;
```

```
 double x= .1;

 fprintf(fp,"   x || sin    cos    tan    acos   asin   atan \n");

      for(;x<=.5;x+=.1)
         {
          fprintf(fp," %.1f ||",x);
          for(i=6;i;)
             fprintf(fp," %.3f ",TrigF[--i](x));
          fprintf(fp,"\n");
         }

  fclose(fp);

  printf("\n\n Ouvrir le fichier list.txt\n");
  getchar();

  return 0;
 }
```

Le résultat :

```
  x || sin    cos    tan    acos   asin   atan
0.1 || 0.100  0.995  0.100  1.471  0.100  0.100
0.2 || 0.199  0.980  0.203  1.369  0.201  0.197
0.3 || 0.296  0.955  0.309  1.266  0.305  0.291
0.4 || 0.389  0.921  0.423  1.159  0.412  0.381
0.5 || 0.479  0.878  0.546  1.047  0.524  0.464
```

```
Remarque :
* Attention à l'ordre des fonctions dans la déclaration du tableau.
* double (*TrigF[6])(double x) = {atan,asin,acos,tan,cos,sin};
```

```
Au démarage :
* La décrémentation ce fait dans le tableau. TrigF[--i](x)
* Il entre 6 dans le tableau.
* 6 est décrémenté -> 5 (avant l'appel de la fonction --i)
* La sixième fonctions est appelé (Sin).
* La numéro cinq. :)
```

```
Au final :
* Il entre UN dans le tableau.
* UN est décrémenté -> 0
* La première fonctions est appelé (atan).
* La numéro zéro. :))
```

```
* i est égal à zéro en rentrant dans la boucle.
* Le cycle est cassé. :(
```

Avec résultat à l'écran

```
 /* ------------------------------ */
 #include    <stdio.h>
 #include     <math.h>
 /* ------------------------------ */
```

```
int main(void)
{
double (*TrigF[6])(double x) = {atan,asin,acos,tan,cos,sin};

int    i= 6;
double x= .1;

     for(;x<=.5;x+=.1)
       {
         printf("\n");
         for(i=6;i;) printf(" %.3f ",TrigF[--i](x));
       }

 printf("\n\n Press return to continue.\n");
 getchar();

 return 0;
}
```

Deuxième exemple

Nous voulons créer la fonction **Derivate** capable de calculer la dérivé première et seconde d'une fonction, en utilisant un **tableau de pointeurs de fonctions**.

Déclaration d'un tableau de pointeurs de fonctions

- Voir listing en fin de page.

```
double (*Derivate[3])(double (*P_f)(double x),double a,double h) = {fx,Dx_1,Dx_2};
```

- Toutes les fonctions (fx,Dx_1,Dx_2) ont la même forme.
- double FUNCTION(double (*P_f)(double x) double double)
- Le tableau à la même forme que les fonctions.
- double ARRAY(double (*P_f)(double x) double double)

- Il y a trois fonctions. (0,1,2)= {fx, Dx_1, Dx_2}.

- La fonction fx donne f.
- Supprimer cette fonction et travailler sur deux fonctions.
- Réfléchissez.

Exemple d'un appel

```
     f(x)=Derivate[0](f,x,0.)
```

- Derivate[0] donne f(x).
- Voir la fonction fx() la première fonction du tableau.
- h = 0 dans cet appel parce qu'il n'est pas utilisé (voir code de fx())

Exemple à tester

```
/* ------------------------------ */
#include    <stdio.h>
#include     <math.h>
/* ------------------------------ */


/* ------ Fonction f ------------ */
double f(double x){return( pow(x,2.));}
/* ------------------------------ */
char  feq[] = "x**2";
/* ------------------------------ */

/* ------ Fonction g ------------ */
double g(double x){return(
 pow(cos(x),2.)+sin(x)+x-3);}
/* ------------------------------ */
char  geq[] = "cos(x)**2+sin(x)+x-3";
/* ------------------------------ */


/* ------------------------------ */
double fx(
double (*P_f)(double x),
double a,
double h
)
{
 return( ((*P_f)(a)) );
}


/* ------------------------------
 f'(a) = f(a+h) - f(a-h)
          -------------
              2h
   ------------------------------ */
double Dx_1(
double (*P_f)(double x),
double a,
double h
)
{
 return( ( ((*P_f)(a+h))-((*P_f)(a-h)) ) / (2.*h) );
}


/* ------------------------------
 f''(a) = f(a+h) - 2 f(a) + f(a-h)
           ----------------------
                     h**2
   ------------------------------- */
double Dx_2(
double (*P_f)(double x),
double a,
double h
)
{
 return( (((*P_f)(a+h))-2*((*P_f)(a))+((*P_f)(a-h))) / (h*h) );
}


/* ------------------------------ */
int main(void)
{
double (*Derivate[3])(double (*P_f)(double x),
                      double a,
                      double h) = {fx,Dx_1,Dx_2};
double x = 2;
double h = 0.001;

printf("\n\n");

printf("  f(%.3f) = %.3f = %.3f \n",x,f(x),       Derivate[0](f,x,0.));
```

```
printf(" f'(%.3f) = %.3f = %.3f \n",x,Dx_1(f,x,h),Derivate[1](f,x,h));
printf("f''(%.3f) = %.3f = %.3f \n",x,Dx_2(f,x,h),Derivate[2](f,x,h));

printf("\n\n");

printf("  g(%.3f) = %.3f = %.3f \n",x,g(x),       Derivate[0](g,x,0.));
printf(" g'(%.3f) = %.3f = %.3f \n",x,Dx_1(g,x,h),Derivate[1](g,x,h));
printf("g''(%.3f) = %.3f = %.3f \n",x,Dx_2(g,x,h),Derivate[2](g,x,h));

    printf("\n\n Press return to continue.");

 getchar();

 return 0;
}
```

Types avancés - structures, unions, énumérations

Structures

```
struct ma_structure {
    type1 champ1;
    type2 champ2;
...
    typeN champN;
} var1, var2, ..., varM;
```

Déclare une structure (ou enregistrement) *ma_structure* composé de N champs, *champ1* de type *type1*, *champ2* de type *type2*, etc. On déclare, par la même occasion, M variables de type *struct ma_structure*.

Accès aux champs

L'accès aux champs d'une structure se fait avec un point :

```
struct complexe {
    int reel;
    int imaginaire;
} c;

c.reel = 1;
c.imaginaire = 2;
```

Initialisation

Il y a plusieurs façons d'initialiser une variable de type structure :

- En initialisant les champs un à un :

```
struct t_mastruct {
    char ch;
    int nb;
    float pi;
};
t_mastruct variable;
variable.ch = 'a';
variable.nb = 12345;
variable.pi = 3.141592;
```

Cette façon est néanmoins pénible lorsqu'il y a beaucoup de champs.

- À la déclaration de la variable :

```
struct t_mastruct {
    char ch;
    int nb;
    float pi;
};
t_mastruct variable = { 'a', 12345, 3.141592 };
```

Les valeurs des champs sont assignés dans l'ordre où ils sont déclarés. S'il manque des initialisations, les champs seront initialisés à 0. L'inconvénient, c'est qu'on doit connaitre l'ordre où sont déclarés les champs, ce qui peut être tout aussi pénible à retrouver, et peut causer des plantages lorsque la définition de la structure est modifiée.

- Une nouveauté du C99 permet d'initialiser certains champs à la déclaration de la variable, en les nommant :

```
struct t_mastruct {
    char ch;
    int nb;
    float pi;
};
t_mastruct variable = { .pi = 3.141592, .ch = 'a', .nb = 12345 };
```

Les champs non initialisés seront mis à leur valeur par défaut.

Manipulation

La seule opération prise en charge par le langage est la copie, lors des affectations ou des passages de paramètres à des fonctions. Toutes les autres opérations sont à la charge du programmeur, notamment la comparaison d'égalité (cf. section suivante).

Alignement et bourrage (padding)

Il s'agit d'un concept relativement avancé, mais qu'il est bien de connaitre pour agir en connaissance de cause. Lorsqu'on déclare une structure, on pourrait naïvement croire que les champs se suivent les uns à la suite des autres en mémoire. Or, il arrive souvent que des octets soient intercalés entre les champs.

Considérez la structure suivante:

```
struct ma_structure {
    char  champ1;      /*  8 bits */
    int   champ2;      /* 32 bits */
    char  champ3;      /*  8 bits */
};
```

On pourrait penser que cette structure occupera 6 octets en mémoire, et pourtant, sur une bonne partie des compilateurs, on obtiendrait une taille plus proche des 12 octets.

En fait, les compilateurs insèrent quasiment toujours des octets entre les champs pour pouvoir les *aligner* sur des adresses qui correspondent à des mots machines. Cela est dû à une limitation de la plupart des processeurs, qui ne peuvent lire des « mots » de plus d'un octet que s'ils sont alignés sur un certain adressage (alors qu'il n'y a pas de contrainte particulière pour lire un seul octet, si le type `char` est défini sur 8bit).

En se représentant la mémoire comme un tableau continu, on peut tracer le dessin suivant:

```
   | bloc N        | bloc N + 1    | bloc N + 2    |
---+---------------+---------------+---------------+---
   | a | b | c | d | e | f | g | h | i | j | k | l |
---+---------------+---------------+---------------+---
```

Les cases `a`, `b`, `c`, ... représentent des octets, et les *blocs* des sections de 32 bits. Si on suppose qu'une variable de type `ma_structure` doive être placée en mémoire à partir du bloc numéro N, alors un compilateur pourra, pour des raisons de performance, placer `champ1` en `a`, `champ2` de `e` à `h`, et `champ3` en `i`. Cela permettrait en effet d'accéder simplement à `champ2`: le processeur fournit des instructions permettant de lire ou d'écrire directement le bloc N + 1. Dans ce cas, les octets de `b` à `d` ne sont pas utilisés; on dit alors que ce sont des octets *de bourrage* (ou *padding* en anglais). Un autre compilateur (ou le même, appelé avec des options différentes) peut aussi placer `champ2` de `b` à `e`, et `champ3` en `f`, pour optimiser l'utilisation mémoire. Mais alors il devra générer un code plus complexe lors des accès à `champ2`, le matériel ne lui permettant pas d'accéder en une seule instruction aux 4 octets `b` à `e`.

En fait il faut garder à l'esprit que toutes les variables suivent cette contrainte: aussi bien les variables locales aux fonctions, les champs de structures, les paramètres de fonctions, etc.

L'existence d'octets de bourrage ainsi que leur nombre sont non seulement dépendants de l'architecture, mais aussi du compilateur. Cela dit, il est toujours possible de connaître la « distance » (*offset*) d'un champ par rapport au début de la structure, et ce, de manière portable. Pour cela il existe une macro, déclarée dans l'entête `<stddef.h>`:

```
size_t offsetof(type, champ);
```

La valeur renvoyée est le nombre de `char` (i.e. d'octets la plupart du temps), entre le début de la structure et celui du champ. Le premier argument attendu est bel et bien le *type* de la structure et non une variable de ce type, ni un pointeur vers une variable de ce type. Pour s'en souvenir, il suffit de savoir que beaucoup d'implémentations d'`offsetof` utilisent une arithmétique de ce genre:

```
size_t distance = (size_t) &((type *)NULL)->champ;
```

Si `type` était un pointeur, il faudrait faire un déréférencement supplémentaire (ou éviter l'étoile dans la macro). À noter que, même si cette macro peut s'avérer contraignante (notamment lorsqu'on ne dispose que de type pointeur), il est quand même préférable de l'utiliser pour des raisons de portabilité.

Voici un petit exemple d'utilisation de cette macro:

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

struct ma_structure {
    char  champ1;
    int   champ2;
    char  champ3;
};

int main(void)
{
    /* en C99 */
    /*
    printf("L'offset de 'champ2' vaut %zu.\n",
           offsetof(struct ma_structure, champ2));
    */

    /* en C90 */
    printf("L'offset de 'champ2' vaut %lu.\n",
```

```
            (unsigned long) offsetof(struct ma_structure, champ2));
    return 0;
}
```

Sur une architecture 32 bits, vous obtiendrez très certainement la réponse:

```
L'offset de 'champ2' vaut 4.
```

En fait toute cette section était pour souligner le fait qu'il est difficilement portable de comparer les structures comme des blocs binaires (via la fonction `memcmp` par exemple), car il n'y a aucune garantie que ces octets de bourrage soient initialisés à une certaine valeur. De la même manière, il est sage de prendre quelques précautions avant de transférer cette structure à l'extérieur du programme (comme un fichier, un tube de communication ou une socket IP). En général, il préférable de traiter la structure champ par champ, pour ce genre d'opérations.

Mais comme rien n'est toujours totalement négatif, les conséquences du bourrage offre énormément de souplesse : imaginons 2 structures:

```
struct str {
    char *string;
};
struct str_ok {
    char *string;
    size_t len;
};
```

quelle que soit la structure utilisé, on peut avec l'adresse et la cast (struct str *) accéder à la chaine.

```
int main (void)
{
    struct str s = { "chaine" };
    struct str_ok sok = { "chaine", 0 };
    void *p = &sok;
    if (((struct str *)p)->string)
    {
        sok.len = strlen(sok.string);
        printf("%s\n", ((struct str *)p)->string);
    }
    /* en revanche: Peut-être très dangereux ! */
    p = &s;
    printf("%s\n",((struct str_ok *)p)->string);
    return 0;
}
```

1. ↑ Le lecteur intéressé pourra récupérer la dernière version de travail sur la page "projects" (http://www.open-std.org/jtc1/sc22/wg14/www/projects) [archive] du site du WG14.
2. ↑ Certaines normes sont rendues disponibles gratuitement et en libre téléchargement sur internet, mais leur nombre est relativement faible par rapport à l'ensemble des normes existantes.
3. ↑ On peut citer comme exemple les commentaires de fin de ligne, les macros à nombre variable d'arguments, les déclarations de symboles à n'importe quel endroit et le mot clé `inline` pour pallier la précarité des macros
4. ↑ L'exemple le plus simple serait la définition d'une variable ayant pour identifieur un mot réservé du C++, comme `int class;`.
5. ↑ Le lecteur curieux pourra se référer à cet article (http://david.tribble.com/text/cdiffs.htm) [archive] de David R. Tribble, en anglais, qui énumère les différences entre C99 et C++98.
6. ↑ La norme C contraint les domaines de valeurs des types signés entre $-(2^{N-1}-1)$ et $2^{N-1}-1$, où N est un entier quelconque, et les types non signés entre 0 et $2^{N}-1$.
7. ↑ La liste des types successifs utilisés pour déterminer le type d'une constante entière a changé entre C90 et C99. Cela n'a, quasiment tout le temps, aucune incidence, sauf pour le cas d'une constante de valeur trop

grande pour le type `long` : en C90, le type `unsigned long` est utilisé, alors qu'en C99 le type `long long` sera utilisé. Une telle valeur sera alors signée dans un cas, et non signée dans l'autre. Dans ce cas, utiliser un suffixe adapté (`UL`, `LL` ou `ULL`) permet de s'assurer du type de la constante.

8. ↑ $0{,}1_{10} = 0.0001100110011..._2$
9. ↑ La confusion est surtout née de l'utilisation du mot *byte* pour désigner la taille du type *char* dans la norme, alors que ce mot est utilisé dans presque tous les autres domaines de l'informatique pour désigner un octet.
10. ↑ En occident, en-dehors des codages compatibles avec ASCII, on ne rencontre guère en pratique que la famille de jeux de caractères EBCDIC, utilisée sur des systèmes IBM et quelques mainframes.
11. ↑ Le site Unicode publie une liste de bibliothèque de gestion des caractères et chaînes Unicodes. http://www.unicode.org/resources/libraries.html
12. ↑ Ces trois expressions sont définies sous formes de macro pour le préprocesseur.
13. ↑ La norme C99 a ajouté le type `_Bool`, mais celui-ci est peu utilisé, l'usage des types entiers pour représenter les booléens en C étant très répandu (voir Booléens).
14. ↑ http://www.lipn.fr/~cerin/SE/SeETlangC.pdf
15. ↑ http://www.siteduzero.com/tutoriel-3-14052-lire-et-ecrire-dans-des-fichiers.html#ss_part_4
16. ↑ http://c.developpez.com/faq/?page=fichiers#FICHIERS_copier

Pointeurs vers structures

Il est (bien entendu) possible de déclarer des variables de type pointeur vers structure :

```
struct ma_struct * ma_variable;
```

Comme pour tout pointeur, on doit allouer de la mémoire pour la variable avant de l'utiliser :

```
ma_variable = malloc( sizeof(struct ma_struct) );
```

L'accès aux champs peut se faire comme pour une variable de type structure « normale » :

```
(* ma_variable).champ
```

Ce cas de figure est en fait tellement fréquent qu'il existe un raccourci pour l'accès aux champs d'un pointeur vers structure :

```
ma_variable->champ
```

Unions

Une union et un enregistrement se déclarent de manière identique :

```
union type_union
{
    type1 champ1;
    type2 champ2;
    /* ... */
    typeN champN;
};
```

```
/* Déclaration de variables */
union type_union var1, var2, /* ... */ varM;
```

Toutefois, à la différence d'un enregistrement, les N champs d'une instance de cette union occupent le même emplacement en mémoire. Modifier l'un des champ modifie donc tous les champs de l'union. Typiquement, une union s'utilise lorsqu'un enregistrement peut occuper plusieurs fonctions bien distinctes et que chaque fonction ne requiert pas l'utilisation de tous les champs.

Voici un exemple ultra classique d'utilisation des unions, qui provient en fait de la gestion des événements du système X-Window, très répandu sur les systèmes Unix :

```
union _XEvent
{
    int type;

    XAnyEvent xany;
    XKeyEvent xkey;
    XButtonEvent xbutton;
    XMotionEvent xmotion;
    XCrossingEvent xcrossing;
    XFocusChangeEvent xfocus;
    XExposeEvent xexpose;
    /* ... */
    XErrorEvent xerror;
    XKeymapEvent xkeymap;
    long pad[24];
};
```

La déclaration a juste été un peu raccourcie pour rester lisible. Les types `X*Event` de l'union sont en fait des structures contenant des champs spécifiques au message. Par exemple, si `type` vaut `ButtonPress`, seules les valeurs du champ `xbutton` seront significatives. Parmi ces champs, il y a `button` qui permet de savoir quel bouton de la souris à été préssé :

```
    XEvent ev;
    XNextEvent(display, &ev);

    switch (ev.type) {
    case ButtonPress:
        printf("Le bouton %d a été pressé.\n", ev.xbutton.button);
        break;
    default:
        printf("Message type %d\n", ev.type);
    }
```

En offrant une souplesse nécessaire à la manipulation de l'objet, la gestion de la mémoire en C peut aider à réduire les erreurs de programmation lié à la structure de la mémoire (erreur de segmentation/segfault/"Argh!!!"):

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

struct s1 {
    char *string;
} s1;

struct s2 {
    char   *string;
    size_t len;
    int    ______;
    int    setnull;
} s2;

union str {
    struct s2 s_;
    struct s1 _s;
```

```
}str;

typedef void * STR_s1;
typedef void * STR_s2;

STR_s2
string_analyse(STR_s1 *s)
{
    static union str us, init_us = {{NULL,0,0,0}};
    memcpy((void *)&us, (void *)&init_us, sizeof(union str));
    if (!((struct s1 *)s)->string)
    {
        return &us._s;
    }
    us.s_.len = strlen(((struct s1 *)s)->string);
    if (!us.s_.len)
    {
        us.s_.setnull = 1;
        return &us._s;
    }
    us.s_.string = ((struct s1 *)s)->string;
    return &us.s_;
}

int main(void)
{
    struct s1 s = {NULL}, s_ = {"salut"}, s__ = {""};
    void *p;
    p = string_analyse((void *)&s);
    printf("null? %s,%lu,%i\n",
        ((struct s2 *)p)->string, ((struct s2 *)p)->len, ((struct s2 *)p)->setnull);
    p = string_analyse((void *)&s_);
    printf("null? %s,%lu,%i\n",
        ((struct s2 *)p)->string, ((struct s2 *)p)->len, ((struct s2 *)p)->setnull);
    p = string_analyse((void *)&s__);
    printf("null? %s,%lu,%i\n",
        ((struct s2 *)p)->string, ((struct s2 *)p)->len, ((struct s2 *)p)->setnull);
    printf("Réalisé en toute sécurité...\n"
        "Même pour un petit vaisseau serial killer du monde d'en haut.\n");
    return 0;
}
```

Tout en permettant l'optimisation:

```
struct s1{
    size_t len;
    char   *str;
} s1;

union s {
    struct s1  dbl[2];
    void       *perform[5];
    struct s1  simple;
};
```

void *	void *	void *	void *	void *
size_t	char *	size_t	char *	---
size_t	char *	------		---
n oct	n oct	n oct	n oct	2int=n oct

ce qui signifie que:

```
int main(void)
{
    const char *end[2] = {", ", "\n"};
    union s init = {{{5,"Salut"}, {3,"toi"}}}, us;
```

```
    void *p;
    memset((void *)&us, 0,40);
    memcpy((void *)&us,
            (void *)&init,
            2 * sizeof(struct s1));
    for (p = &us;
        ((union s *)p)->perform[0] != (void *)0;
        p += 16)
        printf("%s%s",
                (char *)((union s *)p)->perform[1],
                end[
                     !((char *)((union s *)p)->perform[2])]
                );
    return 0;
}
```

Définitions de synonymes de types (typedef)

Le langage C offre un mécanisme assez pratique pour définir des synonymes de types. Il s'agit du mot-clé `typedef`.

```
typedef un_type synonyme_du_type;
```

Contrairement aux langages à typage fort comme le C++, le C se base sur les types atomiques pour décider de la compatibilité entre deux types. Dit plus simplement, la définition de nouveaux types est plus un mécanisme d'alias qu'une réelle définition de type. Les deux types sont effectivement parfaitement interchangeables. À la limite on pourrait presque avoir les mêmes fonctionnalités en utilisant le préprocesseur C, bien qu'avec ce dernier vous aurez certainement beaucoup de mal à sortir de tous les pièges qui vous seront tendus.

Quelques exemples

```
typedef unsigned char octet;
typedef double matrice4_4[4][4];
typedef struct ma_structure * ma_struct;
typedef void (*gestionnaire_t)( int );

/* Utilisation */
octet nombre = 255;
matrice4_4 identite = { {1,0,0,0}, {0,1,0,0}, {0,0,1,0}, {0,0,0,1} };
ma_struct pointeur = NULL;
gestionnaire_t fonction = NULL;
```

Ce mot clé est souvent utilisé conjointement avec la déclaration des structures, pour ne pas devoir écrire à chaque fois le mot clé `struct`. Elle permet aussi de **grandement** simplifier les prototypes de fonctions qui prennent des pointeurs sur des fonctions en argument, ou retournent de tels types. Il est conseillé dans de tels cas de définir un type synonyme, plutôt que de l'écrire in extenso dans la déclaration du prototype. Considérez les deux déclarations :

```
/* Déclaration confuse */
void (*fonction(int, void (*)(int)))(int);

/* Déclaration claire avec typedef */
typedef void (*handler_t)(int);

handler_t fonction(int, handler_t);
```

Les vétérans auront reconnu le prototype imbitable de l'appel système `signal()`, qui permet de rediriger les signaux (interruption, alarme périodique, erreur de segmentation, division par zéro, ...).

Énumérations

```
enum nom_enum { val1, val2, ..., valN };
```

Les symboles *val1*, *val2*, ..., *valN* pourront être utilisés littéralement dans la suite du programme. Ces symboles sont en fait remplacés par des entiers lors de la compilation. La numérotation commençant par défaut à 0, s'incrémentant à chaque déclaration. Dans l'exemple ci-dessus, *val1* vaudrait 0, *val2* 1 et *valN* N-1.

On peut changer à tout moment la valeur d'un symbole, en affectant au symbole, la valeur **constante** voulue (la numérotation recommençant à ce nouvel indice). Par exemple :

```
enum Booleen { Vrai = 1, Faux = 0 };

/* Pour l'utiliser */
enum Booleen variable = Faux;
```

Ce qui est assez pénible en fait, puisqu'il faut à chaque fois se souvenir que le type `Booleen` est dérivé d'une énumération. Il est préférable de simplifier les déclarations, grâce à l'instruction *typedef* :

```
typedef enum { Faux, Vrai } Booleen;

/* Pour l'utiliser */
Booleen variable = Faux;
```

Type incomplet

Pour garantir un certain degré d'encapsulation, il peut être intéressant de masquer le contenu d'un type complexe, pour éviter les usages trop « optimisés » de ce type. Pour cela, le langage C permet de déclarer un type sans indiquer explicitement son contenu.

```
struct ma_structure;

/* Plus loin dans le code */
struct ma_structure * nouvelle = alloue_objet();
```

Les différents champs de la structure n'étant pas connus, le compilateur ne saura donc pas combien de mémoire allouer. On ne peut donc utiliser les types incomplets qu'en tant que pointeur. C'est pourquoi, il est pratique d'utiliser l'instruction `typedef` pour alléger les écritures :

```
typedef struct ma_structure * ma_struct;

/* Plus loin dans le code */
ma_struct nouvelle = alloue_objet();
```

Cette construction est relativement simple à comprendre, et proche d'une conception objet. À noter que le nouveau type défini par l'instruction `typedef` peut très bien avoir le même nom que la structure. C'est juste pour éviter les ambiguités, qu'un nom différent a été choisi dans l'exemple.

Un autre cas de figure relativement classique, où les types incomplets sont très pratiques, ce sont les structures s'auto-référençant, comme les listes chainées, les arbres, etc.

```
struct liste
{
    struct liste * suivant;
    struct liste * precedant;
    void *         element;
};
```

Ou de manière encore plus tordue, plusieurs types ayant des références croisées :

```
struct type_a;
struct type_b;

struct type_a
{
    struct type_a * champ1;
    struct type_b * champ2;
    int             champ3;
};

struct type_b
{
    struct type_a * ref;
    void *          element;
};
```

Préprocesseur

Le préprocesseur est un langage de macro qui est analysé, comme son nom l'indique, avant la compilation. En fait, c'est un langage complètement indépendant, il est même théoriquement possible de l'utiliser par dessus un autre langage que le C. Cette indépendance fait que le préprocesseur ignore totalement la structure de votre programme, les directives seront toujours évaluées de haut en bas.

Ces directives commencent toutes par le symbole dièse (#), suivi d'un nombre quelconque de blancs (espace ou tabulation), suivi du nom de la directive en minuscule. Les directives doivent être déclarées sur une ligne dédiée. Les noms standards de directives sont :

- `define` : définit un motif de substitution.
- `undef` : retire un motif de substitution.
- `include` : inclusion de fichier.
- `ifdef`, `ifndef`, `if`, `else`, `elif`, `endif` : compilation conditionnelle.
- `pragma` : extension du compilateur.
- `error` : émettre un message d'erreur personnalisé et stopper la compilation.

Variables de substitution et macros

Déclarations de constantes

Les *variables de substitution* du préprocesseur fournissent un moyen simple de nommer des constantes. En effet :

```
#define CONSTANTE valeur
```

permet de substituer *presque partout* dans le code source qui suit cette ligne la suite de caractères « CONSTANTE » par la *valeur*. Plus précisément, la substitution se fait partout, à l'exception des caractères et des chaines de caractères. Par exemple, dans le code suivant :

```
#define TAILLE 100

printf("La constante TAILLE vaut %d\n", TAILLE);
```

La substitution se fera sur la deuxième occurrence de TAILLE, mais pas la première. Le préprocesseur transformera ainsi l'appel à `printf` :

```
printf("La constante TAILLE vaut %d\n", 100);
```

Le préprocesseur procède à des traitements sur le code source, sans avoir de connaissance sur la structure de votre programme. Dans le cas des variables de substitution, il ne sait faire qu'un remplacement de texte, comme le ferait un traitement de texte. On peut ainsi les utiliser pour n'importe quoi, des constantes, des expressions, voire du code plus complexe ; le préprocesseur n'ira pas vérifier si la valeur de remplacement est une expression C ou non. Dans l'exemple précédent, nous avons utilisé `TAILLE` pour remplacer la constante 100, mais toute suite de caractères peut être définie. Par exemples:

```
#define PREFIXE "erreur:"
#define FORMULE (1 + 1)

printf(PREFIXE "%d\n", FORMULE);
```

Ici, l'appel à `printf` sera transformé ainsi :

```
printf("erreur:" "%d\n", (1 + 1));
```

Une définition de constantes peut s'étendre sur plusieurs lignes. Pour cela, il faut que le **dernier** caractère de la ligne soit une barre oblique inverse ('\'). On peut ainsi définir une macro qui est remplacée par un bout de code plus complexe :

```
#define BLOC_DEFAUT           \
    default:                  \
        puts("Cas interdit."); \
        break;
```

Cette variable de substitution permet ainsi d'ajouter le cas par défaut à un `switch`.

Historiquement, les programmeurs avaient pour habitude d'utiliser des capitales pour distinguer les déclarations du préprocesseur et les minuscules pour les noms de symboles (fonctions, variables, champs, etc.) du compilateur. Ce n'est pas une règle à suivre impérativement, mais elle améliore la lisibilité des programmes.

Il est possible de définir plusieurs fois la même « CONSTANTE ». Le compilateur n'émettra un avertissement que si les deux valeurs ne concordent pas.

En fait, dans la section précédente, il était fait mention d'un mécanisme relativement similaire : les énumérations. On peut légitimement se demander ce que ces énumérations apportent en plus par rapport aux directives du préprocesseur. En fait, on peut essentiellement souligner que :

- Lors des cas multiples (`switch`), le compilateur peut vérifier que l'ensemble des cas couvre l'intervalle de valeur du type, et émettre un avertissement si ce n'est pas le cas. Ce qui est évidemment impossible à faire avec des `#define`.
- Là où l'utilité est plus flagrante, c'est lors du débogage. Un bon débogueur peut afficher le nom de l'élément énuméré, au lieu de simplement une valeur numérique, ce qui rends un peu moins pénible ce processus déjà très rébarbatif à la base, surtout lorsque le type en question est une structure avec des dizaines, pour ne pas dire une centaine, de champs.
- Certains compilateurs (`gcc` pour ne citer que le plus connu) n'incluent pas par défaut les symboles du préprocesseur avec l'option standard de débogage (`-g`), principalement pour éviter de faire exploser la taille de l'exécutable. Si bien que dans un débogueur il est souvent impossible d'afficher la valeur associée à une constante du préprocesseur autrement qu'en fouillant dans les sources. À moins d'avoir un environnement plutôt évolué, cette limitation peut s'avérer très pénible.

Déclarations automatiques

Le langage C impose que le compilateur définisse un certain nombre de constantes. Sans énumérer toutes celles spécifiques à chaque compilateur, on peut néanmoins compter sur :

- `__FILE__` (`char *`) : une chaîne de caractères représentant le nom de fichier dans lequel on se trouve. Pratique pour diagnostiquer les erreurs.
- `__LINE__` (`int`) : le numéro de la ligne en cours dans le fichier.
- `__DATE__` (`char *`) : la date en cours (incluant le jour, le mois et l'année).
- `__TIME__` (`char *`) : l'heure courante (HH:MM:SS).
- `__STDC__` (`int`) : cette constante est en général définie si le compilateur suit les règles du C ANSI (sans les spécificités du compilateur). Cela permet d'encadrer des portions de code non portables et fournir une implémentation moins optimisée, mais ayant plus de chance de compiler sur d'autres systèmes.

Extensions

Chaque compilateur *peut* définir d'autres macro, pourvu qu'elles restent dans les conventions de nommage que la norme leur réserve. Les lister toutes ici est impossible et hors-sujet, mais on peut citer quelques unes, à titre d'exemple, que le lecteur pourra rencontrer régulièrement dans du code. Il va de soi que le fait qu'elles soient définies ou non, et leur éventuelle valeur, est entièrement dépendant du compilateur.

- Détection du système d'exploitation :
  - `_WIN32` ou `__WIN32__` (Windows)
  - `linux` ou `__linux__` (Linux)
  - `__APPLE__` ou `__MACH__` (Apple Darwin)
  - `__FreeBSD__` (FreeBSD)
  - `__NetBSD__` (NetBSD)
  - `sun` ou `__SVR4` (Solaris)
- Visual C++ : `_MSC_VER`
- Compilateur gcc :
  - Version : `__GNUC__`, `__GNUC_MINOR__`, `__GNUC_PATCHLEVEL__`.
- Compilateurs Borland :
  - `__TURBOC__` (version de Turbo C ou Borland C)
  - `__BORLANDC__` (version de Borland C++ Builder)
- Divers : `__MINGW32__` (MinGW), `__CYGWIN__` ou `__CYGWIN32__` (Cygwin),

Déclaration de macros

Une macro est en fait une constante qui peut prendre un certain nombre d'arguments. Les arguments sont placés

entre parenthèses après le nom de la macro **sans espaces**, par exemple :

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

La première macro prend deux arguments et « retourne » le maximum entre les deux. La deuxième est plus subtile, elle échange la valeur des deux arguments (qui doivent être des variables entières), sans passer par une variable temporaire, et ce avec le même nombre d'opérations.

La macro va en fait remplacer toutes les occurrences de la chaîne « MAX » et de ses arguments par « x > y ? x : y ». Ainsi, si on appelle la macro de cette façon :

```
printf("%d\n", MAX(4,6));
```

Elle sera remplacée par :

```
printf("%d\n", 4 > 6 ? 4 : 6);
```

Il faut bien comprendre qu'il ne s'agit que d'une substitution de texte, qui **ne tient pas** compte de la structure du programme. Considérez l'exemple suivant, qui illustre une erreur très classique dans l'utilisation des macros :

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

i = 2 * MAX(4,6); /* Sera remplacé par : i = 2 * 4 > 6 ? 4 : 6; */
```

L'effet n'est pas du tout ce à quoi on s'attendait. Il est donc important de bien parenthéser les expressions, justement pour éviter ce genre d'effet de bord. Il aurait mieux fallu écrire la macro `MAX` de la manière suivante :

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

En fait dès qu'une macro est composée d'autre chose qu'un élément atomique (un lexème, ou un *token*) il est bon de le mettre entre parenthèses, notamment les arguments qui peuvent être des expressions utilisant des opérateurs ayant des priorités arbitraires.

Il est a noter que l'emploi systématique de parenthèses ne protège pas contre tous les effets de bord. En effet :

```
MAX(x++,y); /* Sera remplacé par : ((x++) > (y) ? (x++) : (y)) */
```

Du coup, x est incrémenté de 2 et non pas de 1. La qualité et la performance des compilateurs C modernes fait que l'utilisation de fonctions inline est le plus souvent préférable.

Suppression d'une définition

Il arrive qu'une macro/constante soit déjà définie, mais qu'on aimerait quand même utiliser ce nom avec une autre

valeur. Pour éviter un avertissement du préprocesseur, on doit d'abord supprimer l'ancienne définition, puis déclarer la nouvelle :

```
#undef symbole
#define symbole nouvelle_valeur
```

Cette directive supprime le symbole spécifié. À noter que même pour les macros avec arguments, il suffit juste de spécifier le nom de cette macro. Qui plus est, supprimer une variable de substitution inexistante ne provoquera aucune erreur.

Transformation en chaîne de caractères

Le préprocesseur permet de transformer une **expression** en chaîne de caractères. Cette technique ne fonctionne donc qu'avec des macros ayant au moins un argument. Pour transformer n'importe quel argument de la macro en chaîne de caractères, il suffit de préfixer le nom de l'argument par le caractère dièse ('#'). Cela peut être utile pour afficher des messages de diagnostique très précis, comme dans l'exemple suivant :

```
#define assert(condition) \
if( (condition) == 0 ) \
{ \
        puts( "La condition '" #condition "' a échoué" ); \
        exit( 1 );            \
}
```

À noter qu'il n'existe pas de mécanisme simple pour transformer la *valeur* de la macro en chaîne de caractères. C'est le cas classique des constantes numériques qu'on aimerait souvent transformer en chaîne de caractères : le préprocesseur C n'offre malheureusement rien de vraiment pratique pour effectuer ce genre d'opération.

Concaténation d'arguments

Il s'agit d'une facilité d'écriture pour simplifier les tâches répétitives. En utilisant l'opérateur ##, on peut concaténer deux expressions :

```
#define version(symbole) symbole ## _v123

int version(variable); /* Déclare "int variable_v123;" */
```

Déclaration de macros à nombre variable d'arguments

Ceci est une nouveauté du C99. La déclaration d'une macro à nombre variable d'arguments est en fait identique à une fonction, sauf qu'avec une macro on ne pourra pas traiter les arguments supplémentaires un à un. Ces paramètres sont en fait traités comme un tout, via le symbole `__VA_ARGS__`. Exemple :

```
#define debug(message, ...) fprintf( stderr, __FILE__ ":%d:" message "\n", __LINE__, __VA_ARGS__ )
```

Il y a une restriction qui ne saute pas vraiment aux yeux dans cet exemple, c'est que **les points de suspension doivent obligatoirement être remplacés par au moins un argument**, ce qui n'est pas toujours très pratique. Malheureusement le langage C n'offre aucun moyen pour contourner ce problème pourtant assez répandu.

À noter, une **extension du compilateur gcc**, qui permet de s'affranchir de cette limitation en rajoutant l'opérateur ## à __VA_ARGS__ :

```
/* Extension de gcc pour utiliser la macro sans argument */
#define debug(message, ...) fprintf( stderr, __FILE__ ":%d:" message "\n", __LINE__, ##__VA_ARGS__ )
```

Ces macros peuvent être utilisées par exemple pour des traitements autour de la fonction `printf`, voir par exemple Variadic Macros (http://www.redhat.com/docs/manuals/enterprise/RHEL-3-Manual/cpp/variadic-macros.html) [archive] anglais.

Exemples

Les macro du préprocesseur sont souvent utilisées dans des situations où une fonction, éventuellement `inline`, aurait le même effet, avec une meilleure robustesse. Cependant, il y a des usages pour lesquels les macros ne peuvent être remplacés par des fonctions.

Par exemple, pour afficher la taille des types C, on écrirait un programme comme le suivant :

```
#include <stdio.h>

int main(void)
{
    printf("sizeof(char) = %zd.\n", sizeof(char));
    printf("sizeof(short) = %zd.\n", sizeof(short));
    printf("sizeof(int) = %zd.\n", sizeof(int));
    /* ...*/
    return 0;
}
```

Écrire un tel programme peut être fatiguant, et il est très facile de faire des erreurs en recopiant les lignes. Ici, une fonction ne pourrait simplifier l'écriture. En effet, il faudrait passer à la fonction le nom du type, pour l'afficher dans la chaîne "sizeof(XXX) = ", et la taille. On devrait donc donner deux arguments à la fonction, pour l'appeler ainsi:

```
print_taille("char", sizeof(char));
```

Ce qui ne serait pas beaucoup plus simple, surtout que la fonction elle-même serait plus complexe (utilisation de fonctions `str*` pour inclure le premier paramètre dans la chaîne de format). Ici, le préprocesseur fournit une solution beaucoup plus simple, à l'aide de l'opérateur # :

```
#include <stdio.h>

#define PRINTF_SIZEOF(type) printf("sizeof(" #type ") = %zd.\n", sizeof(type))

int main(void)
{
    PRINTF_SIZEOF(char);
    PRINTF_SIZEOF(short);
    PRINTF_SIZEOF(int);
    return 0;
}
```

Une technique similaire peut être utilisée pour conserver à l'affichage le nom de constantes ou d'énumérations :

```
#include <stdio.h>
enum etat_t { Arret, Demarrage, Marche, ArretEnCours };

#define CASE_ETAT(etat) case etat: printf("Dans l'état " #etat "\n"); break;

void affiche_etat(etat_t etat)
{
  switch (etat)
  {
     CASE_ETAT(Arret)
     CASE_ETAT(Demarrage)
     CASE_ETAT(Marche)
     CASE_ETAT(ArretEnCours)
     default:
         printf("Etat inconnu (%d).\n", etat);
         break;
  }
}
```

Ces exemples sont tirés d'un message (http://groups.google.fr/group/comp.lang.c/msg/19f07545672ba9e7) [archive] de Bill Godfrey sur comp.lang.c.

Compilation conditionnelle

Les tests permettent d'effectuer de la compilation conditionnelle. La directive `#ifdef` permet de savoir si une constante ou une macro est définie. Chaque déclaration `#ifdef` doit obligatoirement se terminer par un `#endif`, avec éventuellement une clause `#else` entre. Un petit exemple classique :

```
#ifdef DEBUG
/* S'utilise : debug( ("Variable x = %d\n", x) ); (double parenthésage) */
#define debug(x) printf x
#else
#define debug(x)
#endif
```

L'argument de la directive `#ifdef` doit obligatoirement être un symbole du préprocesseur. Pour utiliser des expressions un peu plus complexes, il y a la directive `#if` (et `#elif`, contraction de `else if`). Cette directive utilise des expressions semblable à l'instruction `if()` : si l'expression évaluée est différente de zéro, elle sera considéré comme vraie, et comme fausse si l'expression s'évalue à 0.

On peut utiliser l'addition, la soustraction, la multiplication, la division, les opérateurs binaires (&, |, ^, ~, <<, >>), les comparaisons et les connecteurs logiques (&& et ||). Ces derniers sont évalués en circuit court comme leur équivalent C. Les opérandes possibles sont les nombres entiers et les caractères, ainsi que les macros elle-mêmes, qui seront remplacés par leur valeur.

À noter, l'opérateur spécial `defined` qui permet de tester si une macro est définie (et qui renvoit donc un booléen). Exemple :

```
#if defined(DEBUG) && defined(NDEBUG)
#error DEBUG et NDEBUG sont définis !
#endif
```

Cette directive est très pratique pour désactiver des pans entier de code sans rien y modifier. Il suffit de mettre une expression qui vaudra toujours zéro, comme :

```
#if 0
/* Vous pouvez mettre ce que vous voulez ici, tout sera ignoré, même du code invalide */
:-)
#endif
```

Un autre avantage de cette technique, est que les directives de compilations peuvent être imbriquées, contrairement aux commentaires. Dans ce cas, s'il y avait eu d'autres directives `#if` / `#endif` (correctement balancées), la clause `#if 0` ne s'arrêterait pas au premier `#endif` rencontré, tandis que cela aurait été le cas avec des commentaires.

À noter qu'au niveau du préprocesseur **il est impossible d'utiliser les opérateurs du C**. Notamment l'opérateur `sizeof`, dont le manque aura fait grincer les dents à des générations de programmeurs, sera en fait interprété comme étant une macro. Il faut bien garder à l'esprit que le préprocesseur C est **totalement** indépendant du langage C.

Inclusion de fichiers

Il s'agit d'une autre fonctionnalité massivement employée dans toute application C qui se respecte. Comme son nom l'indique, la directive `#include` permet d'inclure in extenso le contenu d'un autre fichier, comme s'il avait été écrit en lieu et place de la directive. On peut donc voir ça comme de la factorisation de code, ou plutôt de *déclarations*. On appelle de tels fichiers, des fichiers en-têtes (*header files*) et on leur donne en général l'extension `.h`.

Il est rare d'inclure du code dans ces fichiers (*définitions* de variables ou de fonctions), principalement parce que ces fichiers sont destinés à être inclus dans plusieurs endroits du programme. Ces définitions seraient donc définies plusieurs fois, ce qui, dans le meilleur des cas, seraient du code superflu, et dans le pire, pourraient poser des problèmes à l'édition des liens.

On y place surtout des déclarations de type, macros, prototypes de fonctions relatif à **un module**. On en profite aussi pour documenter toutes les fonctions publiques, leurs paramètres, les valeurs renvoyées, les effets de bords, la signification des champs de structures et les pré/post conditions (quel état doit respecter la fonction avant/après son appel). Dans l'idéal on devrait pouvoir comprendre le fonctionnement d'un module, simplement en lisant son fichier en-tête.

La directive `#include` prend en fait un argument : le nom du fichier que vous voulez inclure. Ce nom doit être soit mis entre guillemets doubles ou entre balises (< >). Cette différence affecte simplement l'ordre de recherche du fichier. Dans le premier cas, le fichier est recherché dans le répertoire où le fichier contenant la directive se trouve, puis le préprocesseur regarde à des endroits préconfigurés. Dans le second cas, il regardera seulement dans les endroits préconfigurés. Les endroits préconfigurés sont le répertoire *include* par défaut (par exemple `/usr/include/`, sous Unix) et ceux passés explicitement en paramètre au compilateur.

En fait l'argument de la directive `#include` peut aussi être une macro, dont la valeur est un argument valide (soit une chaîne de caractères, soit un nom entre balises < et >) :

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

Exemple

- `fichier.h`

```
void affiche( void );
```

- `fichier.c`

```
#include "fichier.h"

int main( void )
{
        affiche();
        return 0;
}
```

C'est comme si `fichier.c` avait été écrit :

```
void affiche( void );

int main( void )
{
        affiche();
        return 0;
}
```

Protection contre les inclusions multiples

À noter un problème relativement récurrent avec les fichiers en-têtes : il s'agit des inclusions multiples. À mesure qu'une application grandit, il arrive fréquemment qu'un fichier soit inclus plusieurs fois à la suite d'une directive `#include`. Quand bien même les déclarations sont identiques, définir deux types avec le même nom n'est pas permis en C. On peut néanmoins s'en sortir avec cette technique issue de la nuit des temps :

```
#ifndef H_MON_FICHIER
#define H_MON_FICHIER

/* Mettre toutes les déclarations ici */

#endif
```

C'est un problème tellement classique, que le premier réflexe lors de l'écriture d'un tel fichier est de rajouter ces directives. `H_MON_FICHIER` est bien-sûr à adapter à chaque fichier, l'essentiel est que le nom soit unique dans toute l'application. Habituellement on utilise le nom du fichier, mis en majuscule, avec les caractères non-alphabétiques remplacés par des soulignés.

Avertissement et message d'erreur personnalisés

Il peut être parfois utile d'avertir l'utilisateur que certaines combinaisons d'options sont dangereuses ou carrément invalides. Le préprocesseur dispose d'une directive pour effectuer cela:

```
#error "message d'erreur"
```

Cette directive affichera en fait le message tel quel, qui peut évidemment contenir n'importe quoi, y compris des caractères réservés, sans qu'on ait besoin de le mettre entre guillemets ("). Après émission du message, la compilation s'arrêtera.

À noter que certains compilateurs comme GCC proposent aussi la directive:

```
#warning "message d'avertissement"
```

qui permet d'afficher un message sans arrêter la compilation.

Bibliothèque standard

La bibliothèque standard du langage C peut paraître relativement pauvre par rapport à d'autres langages tout en un plus récents comme Python, Ruby, Perl, C# ou Java. Conçue avant tout avec un souci de portabilité, et en ayant en tête les contraintes matérielles limitées de certaines plate-formes auxquelles le C est *aussi* destiné, vous obtenez avec cela le plus petit dénominateur commun qui puisse être porté sur le plus grand nombre de plateformes. Les contraintes qu'ont subies l'ANSI, et le WG14 après lui, portant principalement sur les contraintes de portabilité sur certaines architectures parfois « exotiques », et la très forte contrainte de ne pas « casser » du code existant reposant sur un comportement déjà établi, voire normalisé, font que certains points connus pour être complexes, voire peu souhaitables, sont restés dans le langage et la bibliothèque standard.

Concevoir une application avec les seules fonctions présentes dans cette bibliothèque nécessite une très grande rigueur. Le WG14 ne s'étant pas fixé pour but d'étendre la bibliothèque standard de manière importante, il est plus que conseillé de se tourner vers des bibliothèques de plus haut niveau, afin d'éviter de réinventer inutilement la roue. Il en existe heureusement beaucoup, mais décrire ne serait-ce que ce qui existe est hors de la portée de cet ouvrage.

La bibliothèque standard permet toutefois de faire des traitements usuels avec peu d'efforts, pour peu qu'on ait conscience des dangers et des pièges qui sont parfois tendus. Les sections qui suivent permettront de voir un peu plus clair dans les méandres parfois très sombres où s'aventure le C.

Voir aussi

Cours sur Wikiversity sur l'utilisation des fonctions standards du langage C.

Chaînes de caractères

Le langage C offre quelques facilités d'écritures pour simuler les chaînes de caractères à l'aide de tableaux. En plus de cela, certaines fonctions de la bibliothèque standard (et les autres) permettent de faciliter leur gestion. À la différence d'un tableau, les chaînes de caractères respectent une convention : se terminer par le caractère nul, `'\0'` (*antislash-zero*). Ainsi, pour construire une chaîne de caractères « à la main », il ne faut pas oublier ce caractère final.

On peut noter que, par rapport à ce qui est disponible dans d'autres langages, les fonctions de la bibliothèque standard sont peu pratiques à utiliser. On peut avancer sans trop de risque qu'une des plus grandes lacunes du C provient de sa gestion précaire des chaînes de caractères, pourtant massivement employées dans tout programme digne de ce nom. Pour faire court, on ne saurait trop conseiller de soit reprogrammer les fonctions ci-dessous, d'utiliser une bibliothèque externe ou de faire preuve de paranoïa avant leur utilisation.

On notera en particulier que la bibliothèque du langage C, assez ancienne n'apporte pas de fonction particulière pour traiter les spécificités de nouveaux standards tels qu'Unicode. Elle est toutefois assez générique pour effectuer certains opérations basiques. Pour une utilisation plus avancée, un bibliothèque Unicode peut avoir un intérêt.

Les fonctions permettant de manipuler les chaînes de caractères se trouvent dans l'entête `<string.h>`', ainsi pour les utiliser il faut ajouter la commande préprocesseur :

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

Comparaison de chaînes

```
int strcmp(const char * chaine1, const char * chaine2);
int strncmp(const char * chaine1, const char * chaine2, size_t longueur);
```

Compare les chaînes *chaine1* et *chaine2* et renvoie un entier :

- négatif, si *chaine1* est inférieure à *chaine2* (avant dans l'ordre alphabétique) ;
- nul, si *chaine1* est égale à *chaine2* (.i.e. *chaine1* et *chaine2* sont identiques) ;
- positif, si *chaine1* est supérieur à *chaine2* (après dans l'ordre alphabétique) .

première remarque : lorsque les deux chaînes sont égales, `strcmp` renvoie 0, qui a la valeur de vérité *faux*. Pour tester l'égalité entre deux chaînes, il faut donc écrire soit `if (strcmp(chaine1, chaine2) == 0) ...`, soit `if (!strcmp(chaine1, chaine2)) ...` mais surtout pas `if (strcmp(chaine1, chaine2)) ...` qui teste si deux chaînes sont différentes !

seconde remarque : l'opérateur ==, dans le cas de pointeurs, teste si les adresses sont égales. Noter `chaine1 == chaine2`, si *chaine1* et *chaine2* sont des `char *` revient à tester si les deux chaînes pointent sur la même zone mémoire et non pas à tester l'égalité de leur contenu.

troisième remarque : la comparaison effectuée est une comparaison binaire. Deux chaînes canoniquement équivalentes au sens Unicode sont donc ici considérées différentes.

quatrième remarque : la comparaison n'est pas basée sur un alphabet mais sur un codage de caractère. Ceci conduit à un tri particulier des caractères accentuées, majuscules et minuscules.

À noter l'existence de deux fonctions de comparaisons de chaîne, insensibles à la casse des caractères et s'adaptant à la localisation en cours, fonctionnant sur le même principe que `strcmp()` et `strncmp()`, mais dont l'origine provient des systèmes BSD :

```
int strcasecmp(const char * chaine1, const char * chaine2);
int strncasecmp(const char * chaine1, const char * chaine2, size_t longueur);
```

Longueur d'une chaîne

```
size_t strlen(const char * chaine);
```

Renvoie la longueur de la *chaine* en octets, sans compter le marqueur de fin de chaîne `'\0'`.

exemple : `strlen("coincoin")` renvoie 8.

Concaténation et copie de chaînes

```
char * strcpy (char *destination, const char *source);
char * strncpy (char *destination, const char *source, size_t n);
```

```
char * strcat (char *destination, const char *source);
char * strncat (char *destination, const char *source, size_t n);
```

`strcpy` copie le contenu de *source* à l'adresse mémoire pointé par *destination*, incluant le caractère nul final. `strcat` concatène la chaîne *source* à la fin de la chaîne *destination*, en y rajoutant aussi le caractère nul final. Toutes ces fonctions renvoient le pointeur sur la chaîne *destination*.

Pour *strncpy*, on notera que:

- si la chaîne *source* a **moins de n caractères non nuls**, *strncpy* complètera la chaîne *destination* avec des caractères nuls;
- si la chaîne *source* a **n caractères non nuls ou plus**, la fonction **n'insèrera pas de caractère nul** à la fin de la chaine *destination* (i.e. *destination* ne sera pas une chaîne de caractères valide).

Il faut être très prudent lors des copies ou des concaténations de chaînes, car les problèmes pouvant survenir peuvent être très pénibles à diagnostiquer. L'erreur « classique » est de faire une copie dans une zone mémoire non réservée ou trop petite, comme dans l'exemple suivant, *a priori* anodin:

⚠ Ce code contient **une erreur volontaire** !

```
char *copie_chaine(const char *source)
{
    char *chaine = malloc(strlen(source));

    if (chaine != NULL)
    {
        strcpy(chaine, source);
    }
    return chaine;
}
```

Ce code *a priori* correct provoque pourtant l'écriture d'un caractère dans une zone non allouée. Les conséquences de ce genre d'action sont totalement imprévisibles, pouvant au mieux passer inaperçue, ou au pire écraser des structures de données critiques dans la gestion des blocs mémoires et engendrer des accès mémoire illégaux lors des prochaines tentatives d'allocation ou de libération de bloc, i.e. le cauchemar de tout programmeur. Dans cet exemple il aurait bien évidemment fallu écrire :

```
        char * chaine = malloc( strlen(source) + 1 );
```

Une autre erreur, beaucoup plus fréquente hélas, est de copier une chaîne dans un tableau de caractères local, sans se soucier de savoir si ce tableau est capable d'accueillir la nouvelle chaîne. Les conséquences peuvent ici être beaucoup plus graves qu'un simple accès illégal à une zone mémoire globale.

Un écrasement mémoire (*buffer overflow*) est considéré comme un défaut de sécurité relativement grave, puisque, sur un grand nombre d'architectures, un « attaquant » bien renseigné sur la structure du programme peut effectivement lui faire exécuter du code arbitraire. Ce problème vient de la manière dont les variables locales aux fonctions et certaines données internes sont stockées en mémoire. Comme le C n'interdit pas d'accéder à un tableau en dehors de ses limites, on pourrait donc, suivant la qualité de l'implémentation, accéder aux valeurs stockées au-delà des déclarations de variables locales. En fait, sur un grand nombre d'architectures, les variables locales sont placées dans un espace mémoire appelé *pile*, avec d'autres informations internes au système, comme *l'adresse de retour de la fonction*, c'est-à-dire l'adresse de la prochaine instruction à exécuter après la fin de la fonction. En s'y prenant bien, on peut donc écraser cette valeur pour la remplacer par l'adresse d'un autre bout de code, qui donnerait l'ordre d'effacer le disque dur, par exemple ! Si en plus le programme possède des privilèges, les résultats peuvent être assez catastrophiques. Ainsi décrit, le problème semble complexe : il faudrait que l'attaquant puisse insérer dans une zone mémoire de l'application un bout de code qu'il a lui-même écrit, et qu'il arrive à écrire l'adresse de ce bout

de code là où le système s'attend à retrouver l'adresse de retour de la fonction. Cependant, ce genre d'attaque est aujourd'hui *très* courant, et les applications présentant ce genre d'erreur deviennent très rapidement la cible d'attaques.

Voici un exemple très classique, où ce genre d' *exploit* peut arriver :

⚠ Ce code contient **une erreur volontaire** !

```
int traite_chaine(const char *ma_chaine)
{
    char tmp[512];

    strcpy(tmp, ma_chaine);

    /* ... */
}
```

Ce code, hélas plus fréquent qu'on ne le pense, est à **bannir**. Parmi différentes méthodes, on peut éviter ce problème en ne copiant qu'un certain nombre des premiers caractères de la chaîne, avec la fonction `strncpy` :

```
char tmp[512];
strncpy(tmp, ma_chaine, sizeof(tmp));
tmp[sizeof(tmp) - 1] = '\0';
```

On notera l'ajout explicite du caractère nul, si *ma_chaine* est plus grande que la zone mémoire *tmp*. La fonction `strncpy` ne rajoutant hélas pas, dans ce cas, de caractère nul. C'est un problème tellement classique que toute application C reprogramme en général la fonction `strncpy` pour prendre en compte ce cas de figure (voir la fonction POSIX `strdup`, ou `strlcopy` utilisée par le système d'exploitation OpenBSD, par exemple)

Le langage n'offrant que très peu d'aide, la gestion correcte des chaînes de caractères est un problème à ne pas sous-estimer en C.

Recherche dans une chaîne

```
char * strchr(const char * chaine, int caractère);
char * strrchr(const char * chaine, int caractère);
```

Recherche le *caractère* dans la *chaine* et renvoie la position de la première occurrence dans le cas de `strchr` et la position de la dernière occurrence dans le cas de `strrchr`.

```
char * strstr(const char * meule_de_foin, const char * aiguille);
```

Recherche l'*aiguille* dans la *meule de foin* et renvoie la position de la première occurrence.

Traitement des blocs mémoire

La bibliothèque `<string.h>` contient encore quelques fonctions pour la manipulation de zone brute de mémoire. Ces fonctions, préfixées par `mem`, sont les équivalents des fonctions `str*` pour des zones de mémoire qui ne sont pas des chaînes de caractères. Cela peut être des tableaux non terminés par la valeur 0, comme des tableaux pouvant

contenir la valeur 0 avant la fin, par exemple. Elles permettent aussi de traiter des structures (par exemple pour copier les données d'une structure dans une autre), ou des zones de mémoires allouées dynamiquement (par exemple pour initialiser une zone mémoire allouée par `malloc`).

Comme, au contraire des fonctions `str*`, elles n'ont pas de délimiteur de fin de tableau, il faut leur donner en paramètre la taille de ces tableaux (cette taille étant en *bytes* au sens strict, bien que souvent en octets).

```
void memcpy( void * destination, const void * source, size_t longueur );
```

Copie '`longueur`' octet(s) de la zone mémoire '`source`' dans '`destination`'. Vous devez bien sûr vous assurer que la chaine destination ait suffisament de place, ce qui est en général plus simple dans la mesure où l'on connait la longueur.
Attention au récouvrement des zones : si `destination + longueur < source` alors `source >= destination`.

```
void memmove( void * destination, const void * source, size_t longueur )
```

Identique à la fonction `memcpy()`, mais permet de s'affranchir totalement de la limitation de recouvrement.

```
void memset( void * memoire, int caractere, size_t longueur );
```

Initialise les '`longueur`' premiers octets du bloc '`memoire`', par la valeur convertie en type `char` de '`caractere`'.
Cette fonction est souvent employée pour mettre à zéro tous les champs d'une structure ou d'un tableau :

```
struct MonType_t mem;
memset( &mem, 0, sizeof mem );
```

```
int memcmp( const void * mem1, const void * mem2, size_t longueur );
```

Compare les '`longueur`' premiers octets des blocs '`mem1`' et '`mem2`'. Renvoie les codes suivants :

- < 0 : `mem1` < `mem2`
- = 0 : `mem1` == `mem2`
- > 0 : `mem1` > `mem2`

```
void * memchr( const void * memoire, int caractere, size_t longueur );
```

Recherche dans les '`longueur`' premiers octets du bloc '`memoire`', la valeur convertie en type `char` de '`caractere`'.
Renvoie un pointeur sur l'emplacement où le caractère a été trouvé, ou `NULL` si rien n'a été trouvé.

```
void * memrchr( const void * memoire, int caractere, size_t longueur );
```

Pareil que `memchr()`, mais commence par la fin du bloc '`memoire`'.

Entrées/sorties

Les fonctions d'entrées/sorties sont celles qui vous permettent de communiquer avec l'extérieur, c'est-à-dire, la console, les fichiers, tubes de communication, socket IP, etc ... Pour utiliser ces fonctions, il faut inclure l'en-tête `<stdio.h>`, avec la directive d'inclusion :

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

Manipulation de fichiers

En C, les fichiers ouverts sont représentés par le type `FILE`, qui est un type *opaque* : on ne connaît pas la nature réelle du type, mais seulement des fonctions pour le manipuler. Ainsi, on ne peut créer directement de variables de type `FILE`, seules les fonctions de la bibliothèque standard peuvent créer une variable de ce type, lors de l'ouverture d'un fichier. Ces données sont donc uniquement manipulées par des pointeurs de type `FILE *`.

Ce type est un *flux* de données, qui représente des fichiers, mais peut aussi représenter toute autre source ou destination de données. L'en-tête `<stdio.h>` fournit trois flux que l'on peut utiliser « directement » :

- `stdin`, l'entrée standard ;
- `stdout`, la sortie standard ;
- `stderr`, la sortie standard des erreurs.

Souvent, *l'entrée standard* envoie au programme les données issues du clavier, et les *sorties standard* envoient les données que le programme génère à l'écran. Mais d'où viennent et où vont ces données dépend étroitement du contexte et de l'implémentation ; la bibliothèque standard fournit le type `FILE` comme une abstraction pour les manipuler tous de la même manière, ce qui libère le programmeur de certains détails d'implémentations, et permet à l'utilisateur d'un programme d'employer (suivant son implémentation) aussi bien son clavier qu'un fichier comme entrée standard.

Ouverture

```
FILE * fopen(const char * restrict chemin, const char * restrict mode)
```

Ouvre le fichier désigné par le *chemin* et renvoie un nouveau flux de données pointant sur ce fichier. L'argument *mode* est une chaîne de caractères désignant la manière dont on veut ouvrir le fichier :

- **r** : ouvre le fichier en lecture, le flux est positionné au début du fichier ;
- **r+** : ouvre le fichier en lecture/écriture, le flux est positionné au début du fichier ;
- **w** : ouvre le fichier en écriture, supprime toutes les données si le fichier existe et le crée sinon, le flux est positionné au début du fichier ;
- **w+** : ouvre le fichier en lecture/écriture, supprime toutes les données si le fichier existe et le crée sinon, le flux est positionné au début du fichier ;
- **a** : ouvre le fichier en écriture, crée le fichier s'il n'existe pas, le flux est positionné à la fin du fichier ;
- **a+** : ouvre le fichier en lecture/écriture, crée le fichier s'il n'existe pas, le flux est positionné à la fin du fichier.

Résumé des modes

mode	lecture	écriture	crée le fichier	vide le fichier	position du flux
r	X				début
r+	X	X			début
w		X	X	X	début
w+	X	X	X	X	début
a		X	X		fin
a+	X	X	X		fin

Lorsqu'un fichier est ouvert en écriture, les données qui sont envoyées dans le flux ne sont pas directement écrites sur le disque. Elles sont stockées dans un *tampon*, une zone mémoire de taille finie. Lorsque le tampon est plein, les données sont purgées (*flush*), elles sont écrites dans le fichier. Ce mécanisme permet de limiter les accès au système de fichiers et donc d'accélérer les opérations sur les fichiers.

À noter une particularité des systèmes Microsoft Windows, est de traiter différemment les fichiers textes, des fichiers binaires. Sur ces systèmes, le caractère de saut de ligne est en fait composé de deux caractères (CR, puis LF, de code ASCII respectif 13 et 10, ou `'\r'` et `'\n'` écrit sous forme de caractère C). Lorsqu'un fichier est ouvert en mode texte (mode par défaut), **toute séquence CRLF lue** depuis le fichier sera convertie en LF, et tout caractère LF écrit sera en fait précédé d'un caractère CR supplémentaire. Si le fichier est ouvert en mode binaire, aucune conversion n'aura lieu.

Ce genre de comportement issu d'un autre âge, est en fait bien plus agaçant que réellement utile. Le premier réflexe est en général de désactiver ce parasitage des entrées/sorties, tant il est pénible. Pour cela deux cas de figure :

1. **Vous avez accès au nom du fichier** : donc vous pouvez utiliser la fonction `fopen()`. Dans ce cas de figure, il suffit de rajouter la lettre `b` au mode d'ouverture:

```
fopen("fichier.txt", "rb"); /* Ouverture sans conversion */
fopen("fichier.txt", "wb"); /* L'écriture d'un '\n' n'entrainera pas l'ajout d'un '\r' */
```

2. **Vous n'avez pas accès au nom du fichier** (par exemple `stdout` ou `stdin`). Il existe une fonction **spécifique** à Microsoft Windows, non portable sur d'autres systèmes, mais qui peut s'avérer nécessaire:

```
/* Ce code qui suit n'est pas portable */
#include <fcntl.h>

setmode(descripteur, O_BINARY);
```

   Histoire de compliquer encore un petit peu, cette fonction travaille avec les descripteurs de fichier, plutôt que les flux stdio. On peut tout de même s'en sortir avec cette fonction faisant partie de la bibliothèque standard:

```
int fileno(FILE *);
```

   Par exemple pour mettre la sortie standard en mode binaire:

```
setmode(fileno(stdout), O_BINARY);
```

Fermeture

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

Dissocie le *flux* du fichier auquel il avait été associé par `fopen`. Si le fichier était ouvert en écriture, le tampon est vidé. Cette fonction renvoie 0 si la fermeture s'est bien passée (notamment la purge des zones en écriture), ou `EOF` en cas d'erreur (voir le paragraphe sur la gestion d'erreurs).

Suppression

```
int remove(const char * path);
```

Supprime le fichier ou le répertoire nommé '`path`'. La fonction renvoie 0 en cas de réussite et une valeur non nulle en cas d'erreur, ce qui peut inclure :

- un répertoire n'est pas vide ;
- vous n'avez pas les permissions pour effacer le fichier (média en lecture seule) ;
- le fichier est ouvert ;
- etc.

Renommage (ou déplacement)

```
int rename(const char * ancien_nom, const char * nouveau_nom);
```

Cette fonction permet de renommer l'ancien fichier ou répertoire nommé '`ancien_nom`' par '`nouveau_nom`'. Elle peut aussi servir a déplacer un fichier, en mettant le chemin absolu ou relatif du nouvel emplacement dans '`nouveau_nom`'.

La fonction renvoie 0 si elle réussie et une valeur non nulle en cas d'erreur.

Les causes d'erreur dépendent de l'implémentation, et peuvent être:

- vous tentez d'écraser un répertoire par un fichier ;
- vous voulez écraser un répertoire non vide ;
- vous n'avez pas les permissions suffisantes ;
- les deux noms ne sont pas sur la même partition ;
- etc.

Déplacement dans le flux

```
int fseek( FILE * flux, long deplacement, int methode );
long ftell( FILE * flux );
```

`fseek` permet de se déplacer à une position arbitraire dans un *flux*. Cette fonction renvoie 0 en cas de réussite.

deplacement indique le nombre d'octet à avancer (ou reculer si ce nombre est négatif) à partir du point de référence (*methode*) :

- SEEK_SET : le point de référence sera le début du fichier.
- SEEK_CUR : le point de référence sera la position courante dans le fichier.

- SEEK_END : le point de référence sera la fin du fichier.

`ftell` permet de savoir à quelle position se trouve le curseur (ce depuis le début).

En cas d'erreur, ces deux fonctions renvoient -1.

Plusieurs remarques peuvent être faites sur ces deux fonctions :

1. Sur une machine 32bits pouvant gérer des fichiers d'une taille de 64bits (plus de 4Go), ces fonctions sont limite inutilisables, du fait qu'un type `long` sur une telle architecture est limité à 32bits. On mentionnera les fonctions `fseeko()` et `ftello()` qui utilisent le type opaque `off_t`, à la place du type `int`, à l'image des appels systèmes. Ce type `off_t` est codé sur 64bits sur les architectures le supportant et 32bits sinon. La disponibilité de ces fonctions est en général limités aux systèmes Unix, puisque dépendantes de la spécification Single Unix (SUS).
2. Il faut bien sûr que le périphérique où se trouve le fichier supporte une telle opération. Dans la terminologie Unix, on appelle cela un périphérique en mode bloc. À la différence des périphériques en mode caractère (`stdin`, `stdout`, tube de communication, connexion réseau, etc ...) pour lesquels ces appels échoueront.

Synchronisation

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

Cette fonction purge toutes les zones mémoires en attente d'écriture et renvoie 0 si tout s'est bien passé, ou `EOF` en cas d'erreur. Si `NULL` est passé comme argument, tous les flux ouverts en écriture seront purgés.

À noter que cette fonction **ne permet pas** de purger les flux ouverts en lecture (Pour répondre à une question du genre « Voulez-vous effacer ce fichier (o/n) ? »). Une instruction de ce genre sera au mieux ignorée, et au pire provoquera un comportement indéterminé :

⚠ Ce code contient **une erreur volontaire** !

```
fflush( stdin );
```

Pour effectuer une purge des flux ouverts en lecture, il faut passer par des appels systèmes normalisés dans d'autres documents (POSIX), mais dont la disponibilité est en général dépendante du système d'exploitation.

Sorties formatées

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

Ces fonctions permettent d'écrire des données formatées dans :

- la sortie standard pour `printf` ;
- un *flux* pour `fprintf` ;
- une *chaîne* de caractères pour `sprintf`.

En retour elle indique le nombre de caractères qui a été écrit à l'écran, dans le flux ou la zone mémoire (**caractère nul non compris** pour `sprintf`).

Bien que cela ait déjà été traité dans la section dédiée aux chaines de caractères, il faut faire très attention avec la fonction `sprintf()`. Dans la mesure où la fonction n'a aucune idée de la taille de la zone mémoire transmise, il faut s'assurer qu'il n'y aura pas de débordements. Mieux vaut donc utiliser la fonction `snprintf()`, qui permet de limiter explicitement le nombre de caractère à écrire.

À noter que `snprintf()` devrait toujours retourner la taille de la chaine à écrire, indépendamment de la limite fixée par le paramètre `taille`. Le conditionnel reste de mise, car beaucoup d'implémentations de cette fonction se limitent à retourner le nombre de caractères *écrit*, c'est à dire en s'arrêtant à la limite le cas échéant.

Type de conversion

Mis à part l'« endroit » où écrivent les fonctions, elles fonctionnent exactement de la même manière, nous allons donc décrire leur fonctionnement en prenant l'exemple de `printf`.

L'argument *format* est une chaîne de caractères qui détermine ce qui sera affiché par `printf` et sous quelle forme. Cette chaîne est composée de texte « normal » et de séquences de contrôle permettant d'inclure des variables dans la sortie. Les séquences de contrôle commencent par le caractère « % » suivi d'un caractère parmi :

- **d** ou **i** pour afficher un entier signé au format décimal (`int`) ;
- **u** pour un entier non signé au format décimal ;
- **x** ou **X** pour afficher un entier au format hexadécimal (avec les lettres "abcdef" pour le format 'x' et "ABCDEF" avec le format 'X') ;
- **f** pour afficher un réel (`double`) avec une précision fixe ;
- **e** pour afficher un réel (`double`) en notation scientifique ;
- **g** effectue un mixe de 'f' et de 'e' suivant le format le plus approprié ;
- **c** pour afficher en tant que caractère ;
- **s** pour afficher une chaine de caractère C standard ;
- **p** pour afficher la valeur d'un pointeur, généralement sous forme hexadécimale. Suivant le compilateur, c'est l'équivalent soit à `"%08x"`, ou alors à `"0x%08x"`. ;
- **n** ce n'est pas un format d'affichage et l'argument associé doit être de type `int *` et être une référence valide. La fonction stockera dans l'entier pointé par l'argument le nombre de caractères écrit jusqu'à maintenant ;
- **%** pour afficher le caractère '%'.

Contraindre la largeur des champs

Une autre fonctionnalité intéressante du spécificateur de format est que l'on peut spécifier sur combien de caractère les champs seront alignés. Cette option se place entre le '%' et le format de conversion et se compose d'un signe '-' optionnel suivit d'un nombre, éventuellement d'un point et d'un autre nombre (`[-]<nombre>[.<nombre>]`). Par exemple: `%-30.30s`.

Le premier nombre indique sur combien de caractère se fera l'alignement. Si la valeur convertie est plus petite, elle sera alignée sur la droite, ou la gauche si un signe moins est présent au début. Si la valeur est plus grande que la largeur spécifiée, le contenu s'étendra au-delà, décalant tout l'alignement. Pour éviter ça, on peut spécifier un deuxième nombre au delà duquel le contenu sera tronqué. Quelques exemples:

```
printf("%10s",     "Salut");               /* " Salut" */
printf("%-10s",    "Salut");               /* "Salut " */
printf("%10s",     "Salut tout le monde"); /* "Salut tout le monde" */
printf("%10.10s", "Salut tout le monde"); /* "Salut tout" */
```

Contraindre la largeur des champs numériques

On peut aussi paramétrer la largeur du champ, en spécifiant `*` à la place. Dans ce cas, en plus de la valeur à afficher (i.e. **1234**), il faut donner **avant** un entier de type `int` pour dire sur combien de caractères l'alignement se fera (i.e. **10**) :

```
printf("%-*d", 10, 1234); /* "1234 " */
printf("%*d",  10, 1234); /* " 1234" */
```

À noter que pour le formattage de nombres entiers, la limite « dure » du spécificateur de format est sans effet, pour éviter de facheuses erreurs d'interprétation. On peut toutefois utiliser les extensions suivantes :

- `0` : Si un zéro est présent dans le spécificateur de largeur, le nombre sera aligné avec zéros au lieu de blancs.
- `+` : Si un signe plus est présent avec le spécificateur de largeur, le signe du nombre sera affiché tout le temps (0 est considéré comme positif).
- (espace) : Si le nombre est positif, un blanc sera mis avant, pour l'aligner avec les nombres négatifs.

Exemples :

```
printf("%+010d",     543);        /* "+000000543"  */
printf("%-+10d",     543);        /* "+543 "       */
printf("%-+10d",     1234567890); /* "+1234567890" */
printf("%-+10.10d", 1234567890); /* "+1234567890" */
printf("%08x",       543);        /* "0000021f"    */
```

Contraindre la largeur des champs réels

Pour les réels, la limite « dure » sert en fait à indiquer la précision voulue après la virgule :

```
printf("%f",   3.1415926535); /* "3.141593"   */
printf("%.8f", 3.1415926535); /* "3.14159265" */
```

Spécifier la taille de l'objet

Par défaut, les entiers sont présuposés être de type `int`, les réels de type `double` et les chaines de caractères de type `char *`. Il arrive toutefois que les types soient plus grands (et non plus petits à cause de la **promotion des types**, c.f. paragraphes opérateurs et fonction à nombre variable d'arguments), le spécificateur de format permet donc d'indiquer la taille de l'objet en ajoutant les attributs suivants *avant* le caractère de conversion :

- hh : indique que l'entier est un `[un]signed char` au lieu d'un `[unsigned] int` ;
- h : indique que l'entier est de type `[unsigned] short` au lieu d'un `[unsigned] int` ;
- l : pour les entiers, le type attendu ne sera plus `int` mais `long int` et pour les chaines de caractères, il sera de type `wchar_t *` (c.f section chaînes de caractères).
- ll : cet attribut ne concerne que les types entiers, où le type attendu sera `long long int`.
- L : pour les types réels, le type attendu sera `long double`.
- z pour afficher une variable de type *size_t*.

Pour résumer les types d'arguments attendus en fonction de l'indicateur de taille et du type de conversion :

Format	**Attributs de taille**						**Autres attributs (rarement utilisés)**		
	aucun	**hh**	**h**	**l** *(elle)*	**ll** *(elle-elle)*	**L**	**j**	**z**	**t**
n	int *	signed char *	short *	long *	long long *		intmax_t *	size_t *	ptrdiff_t *
d, i, o, x, X	int	signed char	short	long	long long		intmax_t	size_t	ptrdiff_t

u	unsigned int	unsigned char	unsigned short	unsigned long	unsigned long long		uintmax_t	size_t	ptrdiff_t
s	char *			wchar_t *					
c	int			wint_t					
p	void *								
a, A, e, E, f, F, g, G	double					long double			

`hh` et `ll` sont des nouveautés de C99. On notera qu'avec l'attribut `hh` et les formats `n`, `d`, `i`, `o`, `x` ou `X`, le type est `signed char` et non `char`. En effet, comme vu dans le chapitre Types de base, le type `char` peut être signé ou non, suivant l'implémentation. Ici, on est sûr de manipuler le type caractère signé.

Quelques exemples :

```
signed char nb;

printf("%d%hhn", 12345, &nb); /* Affichage de "12345" et nb vaudra 5 */
printf("%ls", L"Hello world!"); /* "Hello world!" */
```

Arguments positionnels

Il s'agit d'une fonctionnalité relativement peu utilisée, mais qui peut s'avérer très utile dans le cadre d'une application internationnalisée. Considérez le code suivant (tiré du manuel de `gettext`) :

```
printf( gettext("La chaine '%s' a %zu caractères\n"), s, strlen(s) );
```

`gettext` est un ensemble de fonctions permettant de manipuler des catalogues de langues. La principale fonction de cette bibliothèque est justement `gettext()`, qui en fonction d'une chaine de caractère retourne la chaine traduite selon la locale en cours (où celle passée en argument si rien n'a été trouvé).

Une traduction en allemand du message précédant, pourrait donner : `"%d Zeichen lang ist die Zeichenkette '%s'"`

On remarque d'emblée que les spécificateurs de format sont inversés par rapport à la chaine originale. Or l'ordre des arguments passés à la fonction `printf()` sera toujours le même. Il est quand même possible de s'en sortir avec les arguments positionnels. Pour cela, il suffit d'ajouter à la suite du caractère `%` un nombre, suivi d'un signe `$`. Ce nombre représente le numéro de l'argument à utiliser pour le spécificateur, en commençant à partir de 1. Un petit exemple :

```
char * s = "Bla bla";
printf("La chaine %2$s a %1$zu caractères\n", strlen(s), s); /* "La chaine Bla bla a 7 caractères" */
```

À noter que si **un** des arguments utilise la référence positionnelle, **tous les autres arguments** devront faire évidemment de même, sous peine d'avoir un comportement imprévisible.

Écriture par bloc ou par ligne

Il s'agit d'une fonctionnalité relativement pointue de la bibliothèque `stdio`, mais qui peut expliquer certains comportement en apparence étrange (notamment avec les systèmes POSIX). Les réglages par défaut étant bien faits,

il y a peu de chance pour que vous ayez à vous soucier de cet aspect, si ce n'est à titre de curiosité.

En règle générale les flux de sortie ouvert par via la bibliothèque `stdio` sont gérés par bloc, ce qui veut dire qu'une *écriture* (via `printf()`, `fprintf()` ou `fwrite()`) ne sera pas systématiquement répercutée dans le fichier associé.

Cela dépend en fait du type d'objet sur lequel les écritures se font :

- Un terminal : les écritures se feront par ligne, ou si les lignes sont plus grandes qu'une certaine taille (4Ko en général), l'écriture se fera par bloc. Les flux en écriture seront aussi purgés si on tente de lire des données depuis le même terminal.
- Autre (fichiers, connexion réseau, tubes de communication) : les écritures se feront par bloc, indépendamment des lignes.
- Flux d'erreur (`stderr`) : écriture immédiate.

C'est ce qui fait qu'un programme affichant des messages à intervalle régulier (genre une seconde), affichent ces lignes une à une sur un terminal, et par bloc de plusieurs lignes lorsqu'on redirige sa sortie vers un programme de mise en page (comme `more`), avec une latence qui peut s'avérer génante. C'est ce qui fait aussi qu'une instruction comme `printf("Salut tout le monde");` n'affichera en général rien, car il n'y a pas de retour à la ligne.

En fait ce comportement peut être explicitement réglé, avec cette fonction :

```
int setvbuf(FILE * restrict flux, char * restrict mem, int mode, size_t taille);
```

Cette fonction doit être appelée juste **après** l'ouverture du flux et **avant** la première écriture. Les arguments ont la signification suivante :

- `flux` : Le flux `stdio` pour lequel vous voulez changer la méthode d'écriture.
- `mem` : Vous pouvez transmettre une zone mémoire qui sera utilisée pour stocker les données avant d'être écrites dans le fichier. Vous pouvez aussi passer la valeur `NULL`, dans ce cas les fonctions `stdio`, appelleront la fonction `malloc()` lors de la *première* écriture.
- `mode` : indique comment se feront les écritures :
  - `_IONBF` : écriture immédiate, pas de stockage temporaire.
  - `_IOLBF` : écriture par ligne.
  - `_IOFBF` : par bloc.
- `taille` : La taille de la zone mémoire transmise ou à allouer.

La fonction `setvbuf()` renvoie 0 si elle réussi, et une valeur différente de zéro dans le cas contraire (en général le paramètre `mode` est invalide).

Cette fonctionnalité peut être intéressante pour les programmes générant des messages sporadiques. Il peut effectivement s'écouler un temps arbitrairement long avant que le bloc mémoire soit plein, si cette commande est redirigée vers un autre programme, ce qui peut s'avérer assez dramatique pour des messages signalant une avarie grave. Dans ce cas, il est préférable de forcer l'écriture par ligne (ou immédiate), plutôt que de faire suivre systématiquement chaque écriture de ligne par un appel à `fflush()`, avec tous les risques d'oubli que cela comporte.

Quelques remarques pour finir

La famille de fonctions `printf()` permet donc de couvrir un large éventail de besoins, au prix d'une pléthore d'options pas toujours faciles à retenir.

Il faut aussi faire attention au fait que certaines implémentations de `printf()` tiennent compte de la localisation pour les conversions des nombres réels (virgule ou point comme séparateur décimal, espace ou point comme séparateurs des milliers, etc.). Ceci peut être gènant lorsqu'on veut retraiter la sortie de la commande. Pour désactiver la localisation, on peut utiliser la fonction `setlocale()`:

```c
#include <locale.h>

/* ... */
setlocale( LC_ALL, "C" );
printf( ... );
setlocale( LC_ALL, "" );
```

Entrées formatées

La bibliothèque `stdio` propose quelques fonctions très puissantes pour saisir des données depuis un flux quelconque. Le comportement de certaines fonctions (`scanf` notamment) peut paraitre surprenant de prime abord, mais s'éclaircira à la lumière des explications suivantes.

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

Ces trois fonctions permettent de lire des données formatées provenant de :

- l'entrée standard pour `scanf` ;
- un *flux* pour `fscanf` ;
- une *chaîne* de caractères pour `sscanf`.

L'argument *format* ressemble aux règles d'écriture de la famille de fonction `printf`, cependant les arguments qui suivent ne sont plus des variables d'entrée mais des variables de sortie (ie : l'appel à `scanf` va modifier leur valeur, il faut donc passer une référence).

Ces fonctions retournent le nombre d'arguments correctement lus depuis le format, qui peut être inférieur ou égal au nombre de spécificateurs de format, et même nul.

Format de conversion

Les fonctions `scanf()` analysent le spécificateur de format et les données d'entrée, en les comparant caractère à caractère et s'arrêtant lorsqu'il y en a un qui ne correspond pas. À noter que les blancs (espaces, tabulations et retour à la ligne) dans le spécificateur de format ont une signification spéciale : à un blanc de la chaine *format* peut correspondre un nombre quelconque de blanc dans les données d'entrée, y compris aucun. D'autres part, il est possible d'insérer des séquences spéciales, commençant par le caractère '%' et à l'image de `printf()`, pour indiquer qu'on aimerait récupérer la valeur sous la forme décrite par le caractère suivant le '%' :

- **s** : extrait la chaîne de caractères, **en ignorant les blancs initiaux et ce jusqu'au prochain blanc**. L'argument correspondant doit être de type `char *` et pointer vers un bloc mémoire suffisamment grand pour contenir la chaîne et son caractère terminal.
- **d** : extrait un **nombre décimal signé** de type `int`, ignorant les espaces se trouvant éventuellement avant le nombre.
- **i** : extrait un **nombre** (de type `int`) hexadécimal, si la chaine commence par "`0x`", octal si la chaine commence par "`0`" et décimal sinon. Les éventuels espaces initiaux seront ignorés.
- **f** : extrait un **nombre réel**, en sautant les blancs, de type `float`.
- **u** : lit un **nombre décimal non-signé**, sans les blancs, de type `int`.
- **c** : lit un **caractère** (de type char), y compris un blanc.
- **[]** : lit une **chaîne de caractères** qui doit faire partie de l'ensemble entre crochets. Cet ensemble est une énumération de caractère. On peut utiliser le tiret ('-') pour grouper les déclarations (comme "`0-9`" ou "`a-z`"). Pour utiliser le caractère spécial ']' dans l'ensemble, il doit être placé en *première position* et, pour utiliser le tiret comme un caractère normal, il doit être mis à la fin. Pour indiquer que l'on veut tous les caractères sauf ceux de l'ensemble, on peut utiliser le caractère '^' en première position. À noter que `scanf` terminera toujours la chaîne par 0 et que, contrairement au spécificateur `%s`, les blancs ne seront pas ignorés.

- **n** : Comme pour la fonction `printf()`, ce spécificateur de format permet de stocker dans l'entier correspondand de type `int`, le **nombre de caractères lus** jusqu'à présent.

Contraindre la largeur

Comme pour la fonction `printf()`, il est possible de contraindre le nombre de caractères à lire, en ajoutant ce nombre juste avant le caractère de conversion. Dans le cas des chaînes, c'est même une obligation, dans la mesure où `scanf()` ne pourra pas ajuster l'espace à la volée.

Exemple :

```
/* Lit une chaîne de caractères entre guillemets d'au plus 127 caractères */
char tmp[128];

if (fscanf(fichier, "Config = \"%127[^\"]\"", tmp ) == 1)
{
    printf("L'argument associé au mot clé 'Config' est '%s'\n", tmp);
}
```

Cet exemple est plus subtil qu'il ne paraît. Il montre comment analyser une structure relativement classique de ce qui pourrait être un fichier de configuration de type "MotClé=Valeur". Ce format spécifie donc qu'on s'attend à trouver le mot clé "Config", en ignorant éventuellement les blancs initiaux, puis le caractère '=', entouré d'un nombre quelconque de blancs, éventuellement aucun. À la suite de cela, on doit avoir un guillemet (""), puis au plus 127 caractères autres que que les guillemets, qui seront stockés dans la zone mémoire `tmp` (qui sera terminée par 0, d'où l'allocation d'un caractère supplémentaire). Le guillemet final est là pour s'assurer, d'une part, que la longueur de la chaîne est bien inférieure à 127 caractère et, d'autre part, que le guillemet n'a pas été oublié dans le fichier.

En cas d'erreur, on peut par exemple ignorer tous les caractères jusqu'à la ligne suivante.

Ajuster le type des arguments

On peut aussi ajuster le type des arguments en fonction des attributs de taille :

Format	**Attributs de taille**				
	aucun	**hh**	**h**	**l** *(elle)*	**ll** *(elle-elle)*
d, i, n	int *	char *	short *	long *	long long *
u	unsigned int *	unsigned char *	unsigned short *	unsigned long *	unsigned long long *
s, c, [ ]	char *				
f	float *			double *	long double *

Ainsi pour lire la valeur d'un entier sur l'entrée standard, on utilisera un code tel que celui ci :

```
#include <stdio.h>

int main(void)
{
    int i;

    printf("Entrez un entier : ");
    scanf("%d", &i);
    printf("la variable i vaut maintenant %d\n", i);
    return 0;
}
```

Les appels à `printf` ne sont pas indispensables à l'exécution du `scanf`, mais permettent à l'utilisateur de

comprendre ce qu'attend le programme (et ce qu'il fait aussi).

Conversions muettes

La fonction `scanf` reconnait encore un autre attribut qui permet d'effectuer la conversion, mais sans retourner la valeur dans une variable. Il s'agit du caractère étoile '*', qui remplace l'éventuel attribut de taille.

En théorie, la valeur de retour ne devrait pas tenir compte des conversions muettes.

Exemple:

```
int i, j;
sscanf("1 2.434e-308 2", "%d %*f %d", &i, &j); /* i vaut 1 et j vaut 2 */
```

Quelques remarques pour finir

La fonction `scanf()` n'est pas particulièrement adaptée pour offrir une saisie conviviale à l'utilisateur, même en peaufinant à l'extrême les spécificateurs de format. En général, il faut s'attendre à une gestion **très rudimentaire** du clavier, avec très peu d'espoir d'avoir ne serait-ce que les touches directionnelles pour insérer du texte à un endroit arbitraire.

Qui plus est, lors de saisie de texte, les terminaux fonctionnent en mode bloc : pour que les données soient transmises à la fonction de lecture, il faut que l'utilisateur confirme sa saisie par entrée. Même les cas les plus simple peuvent poser problèmes. Par exemple, il arrive souvent qu'on ne veuille saisir qu'un caractère pour répondre à une question du genre `"Écraser/Annuler/Arrêter ? (o|c|s)"`, avant d'écraser un fichier. Utiliser une des fonctions de saisie nécessite de saisir d'abord un caractère, ensuite de valider avec la touche *entrée*. Ce qui peut non seulement être très pénible s'il y a beaucoup de questions, mais aussi *risqué* si on ne lit les caractères qu'un à un. En effet, dans ce dernier cas, si l'utilisateur entre une chaine de plusieurs caractères, puis valide sa saisie, les caractères non lus seront disponibles pour des lectures ultérieures, répondant de ce fait automatiquement aux questions du même type.

Il s'agit en fait de deux opérations en apparence simples, mais impossible à réaliser avec la bibliothèque C standard :

1. Purger les données en attente de lecture, pour éviter les réponses « automatiques ».
2. Saisir des caractères sans demande de confirmation.

Ces fonctionnalités sont hélas le domaine de la gestion des terminaux POSIX et spécifiées dans la norme du même nom.

On l'aura compris, cette famille de fonction est plus à l'aise pour traiter des fichiers, ou tout objet nécessitant le moins d'interaction possible avec l'utilisateur. Néanmoins dans les cas les plus simples, ce sera toujours un pis-aller.

À noter, que contrairement à la famille de fonction `printf()`, `scanf()` n'est pas sensible à la localisation pour la saisie de nombres réels.

Entrées non formatées

Pour saisir des chaines de caractères indépendemment de leur contenu, on peut utiliser les fonctions suivantes :

```
char * fgets(char * restrict chaine, int taille_max, FILE * restrict flux);
int fgetc( FILE * restrict flux);
int ungetc( int octet, FILE * flux );
```

La fonction `fgets()` permet de saisir une ligne complète dans la zone mémoire spécifiée, en évitant tout débordement. Si la ligne peut être contenue dans le bloc, elle contiendra le caractère de saut de ligne (`'\n'`), en plus du caractère nul. Dans le cas contraire, la ligne sera tronquée, et la suite de la ligne sera obtenue à l'appel suivant. Si la fonction a pu lire au moins un caractère, elle retournera la chaine transmise en premier argument, ou `NULL` s'il n'y a plus rien à lire.

Un exemple de lecture de ligne arbitrairement longue est fournie dans le livre Exercices en langage C (énnoncé et solution).

La fonction `fgetc()` permet de ne saisir qu'un caractère depuis le flux spécifié. À noter que la fonction renvoie bien un entier de type `int` et non de type `char`, car en cas d'erreur (y compris la fin de fichier), cette fonction renvoie `EOF` (défini à -1 en général).

À noter que cette fonction est incapable de traiter des fichiers mixtes (binaire et texte) depuis un descripteur en mode caractère (accès séquentiel). D'une part la fonction ne renvoyant pas le nombre d'octets lus (ce qui aurait facilement réglé le problème) et d'autre part, ftell() ne fonctionnant pas sur de tels descripteurs, il faudra reprogrammer `fgets` pour gérer ce cas de figure.

Entrées/sorties brutes

Les fonctions suivantes permettent d'écrire ou de lire des quantités arbitraires de données depuis un flux. Il faut faire attention à la portabilité de ces opérations, notamment lorsque le flux est un fichier. Dans la mesure où lire et écrire des structures binaires depuis un fichier nécessite de gérer l'alignement, le bourrage, l'ordre des octets pour les entiers (*big endian*, *little endian*) et le format pour les réels, il est souvent infiniment plus simple de passer par un format texte.

Sortie

```
size_t fwrite(const void * buffer, size_t taille, size_t nombre, FILE * flux);
```

Entrée

```
size_t fread(void * buffer, size_t taille, size_t nombre, FILE * flux);
```

Lit *nombre* éléments, chacun de taille *taille*, à partir du *flux* et stocke le résultat dans le *buffer*. Renvoie le nombre d'éléments correctement lus.

Gestion des erreurs

Qui dit entrées/sorties dit forcément une pléthore de cas d'erreurs à gérer. C'est souvent à ce niveau que se distinguent les « bonnes » applications des autres : fournir un comportement cohérent face à ces situations exceptionnelles. Dans la description des fonctions précédentes, il est fait mention que, en cas d'erreur, un code spécial est retourné par la fonction. De plus, on dispose de la fonction `int ferror( FILE * );`, qui permet de savoir si une erreur a été déclenchée sur un fichier lors d'un appel antérieur à une fonction de la bibliothèque standard.

Si ces informations permettent de savoir s'il y a eu une erreur, elles ne suffisent pas à connaître la cause de l'erreur. Pour cela, la bibliothèque `stdio` repose sur la variable globale *errno*, dont l'utilisation est décrite dans le chapitre sur la gestion d'erreur.

Erreurs

Le langage C fournit un en-tête spécialisé pour la gestion des erreurs : `<errno.h>`. Cet en-tête déclare notamment une variable globale `errno`, et un certain nombre de codes d'erreur, qui permettent aux fonctions de la bibliothèque standard de reporter précisément la cause d'une erreur.

Utilisation

Pour inclure l'en-tête de son fichier source, il faut ajouter la ligne :

```
#include <errno.h>
```

On peut alors utiliser la variable `errno`, de type `int`, pour traiter les erreurs[1].

Lorsqu'on veut utiliser `errno` pour déterminer la cause d'un échec, il faut d'abord s'assurer que la fonction a bel et bien échoué. Le C laissant une certaine liberté dans la manière de signaler un échec, il n'y a pratiquement aucun mécanisme universel qui permette de détecter une telle situation, chaque fonction étant presque un cas particulier. Cependant, une pratique relativement répandue est de retourner un code spécifique, en général en dehors de l'intervalle de ce qu'on attend. Par exemple, lorsque la fonction est censée allouer un objet et retourne un pointeur, une erreur est souvent signalée en retournant un pointeur nul, et la variable `errno` décrit plus en détail la nature de l'erreur.

Il est nécessaire de placer `errno` à 0 *avant* d'utiliser la fonction qui peut échouer, car les fonctions de la bibliothèque standard ne sont pas obligées de la mettre à zéro en cas de succès. Si on ne la réinitialisait pas « manuellement », on pourrait « voir » le résultat d'une erreur causée par un appel de fonction antérieur. La procédure à suivre est donc la suivante :

- Mettre `errno` à 0 ;
- Appeler la fonction (de la bibliothèque standard) que l'on souhaite ;
- Vérifier si elle a échoué ;
  - Si c'est le cas : la valeur de `errno` est disponible pour traiter l'erreur ;
  - Sinon : procéder à la suite du traitement.

Affichage d'un message d'erreur

Une fois qu'on est sûr que la variable `errno` contient une valeur pertinente, il est indispensable de déterminer le traitement à effectuer, et le cas échéant présenter un message tout aussi significatif à l'utilisateur. Rien n'est plus frustrant pour un utilisateur qu'un message aussi abscons que : "`fopen("/tmp/tmpab560f9d4", "w") : erreur 28`", alors qu'un message écrit sous la forme "`impossible de créer le fichier /tmp/tmpab560f9d4: plus de place disponible sur le périphérique`" sera plus facilement compris par l'utilisateur.

Énumérer toutes les valeurs possibles d'`errno` pour leur associer un message peut être relativement pénible, surtout si on doit le faire dans chaque application. Heureusement qu'une fonction toute prête existe :

```
#include <string.h>
char * strerror(int code);
```

Cette fonction permet de connaître la signification textuelle d'une valeur de `errno`. À noter que le code de retour est une chaine statique, dont il est sage de présupposer la durée de vie la plus courte possible. N'essayez pas non plus de modifier le contenu de la chaîne, affichez-la directement ou copiez-la dans une zone temporaire.

Un autre intérêt de passer par cette fonction est que les messages retournés sont adaptés à la localisation du système. Ce qui est loin d'être négligeable, car cela retire au programmeur le souci de traduire les messages d'erreurs dus à la bibliothèque standard dans toutes les langues possibles (chaque implémentation pouvant avoir ses propres codes d'erreur spécifiques, la maintenance d'une telle traduction serait un cauchemar).

Une autre fonction, `perror`, permet d'envoyer le message correspondant à errno sur le flux des erreurs stderr.

```
#include <errno.h>
void perror(const char *chaine);
```

Le paramètre *chaine*, s'il est différent de NULL, est une chaine de caractère passée par le programmeur et affichée avant le message d'erreur.

Exemple

Pour illustrer cela, voici un exemple, qui tente de convertir une chaine de caractères en un nombre. Le nombre donné, 2^{64}, peut ne pas être dans le domaine de valeur du type `unsigned long` (qui peut se limiter à 2^{32} - 1), suivant l'architecture. Dans ce cas, on utilise `errno` pour afficher un message d'erreur approprié à l'utilisateur.

```
#include <stdio.h>  /* puts(), printf(), NULL */
#include <stdlib.h> /* strtoul() */
#include <limits.h> /* ULONG_MAX */
#include <string.h> /* strerror() */
#include <errno.h>  /* errno */

static void teste(const char *nombre)
{
    unsigned long res;

        /* On réinitialise errno */
    errno = 0;

    /* Appel de strtoul : conversion d'une chaine en un entier unsigned long*/
    res = strtoul(nombre, NULL, 10);

    /* On détecte une erreur de strtoul quand elle renvoie ULONG_MAX _et_
     * que errno est non nul. En effet, si on lui passe en entrée la
     * représentation de ULONG_MAX, la conversion se fait sans erreur
     * et errno reste à 0.
     */
    if (res == ULONG_MAX && errno != 0)
    {
        /* Il y a eu une erreur ! */
        (void)fprintf(stderr, "Impossible de convertir le nombre '%s': %s.\n",
                      nombre, strerror(errno));
    }
    else
    {
       (void)printf("La conversion a été effectuée, et la valeur est: %lu\n", res);
    }
}

int main(void)
{
    /* 2 puissance 8 : sera toujours accepte */
    teste("256");

    /* 2^64 - 1 : peut echouer suivant la machine */
    teste("18446744073709551615");
    return EXIT_SUCCESS;
}
```

Si le type `unsigned long` est codé sur 32 bits, on peut obtenir:

```
La conversion a été effectuée, et la valeur est: 256
```

```
Impossible de convertir le nombre '18446744073709551615': la valeur est en dehor
```

Notons bien qu'il est nécessaire de placer `errno` à 0 *avant* d'utiliser `strtoul`, car elle pourrait contenir une valeur non nulle, même au lancement du programme.

Mathématiques

Pour pouvoir utiliser les fonctions mathématiques, il faut utiliser l'en-tête `<math.h>`, ainsi que `<errno.h>` pour gérer les erreurs :

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

Comme pour un certain nombre de fonctions de la bibliothèques standard, il est en effet nécessaire d'utiliser `<errno.h>` pour détecter l'erreur d'une fonction mathématique (voir le chapitre sur la gestion d'erreurs pour voir comment utiliser `<errno.h>`).

Note : Sur certains compilateurs comme GCC, il est nécessaire d'ajouter durant l'édition des liens une option pour que la « librairie mathématique » soit liée au programme. Pour GCC, cette option est `-lm`. Sans cette option, le programme pourra compiler, mais le résultat à l'exécution sera surprenant...

Exponentiations

```
double exp ( double x );
double pow ( double x, double y );
```

exp calcule *e* élevé à la puissance de *x* (e^x) où *e* est la base des logarithmes naturels (ln(e) = 1). *pow* calcule la valeur de *x* élévé à la puissance *y* (x^y).

Erreurs

La fonction *pow* peut déclencher l'erreur suivante :

- **EDOM** : *x* est négatif, et *y* n'est pas un entier.

Logarithmes

```
double log ( double x );
double log10 ( double x );
```

log calcule le logarithme néperien de *x* (noté généralement ln(x) en mathématiques). *log10* calcule le logarithme à base 10 de *x*.

Erreurs

- **EDOM** : *x* est négatif ;
- **ERANGE** : *x* est nul.

Racine carrée

```
double sqrt ( double x );
```

Renvoie la racine carrée de *x*.

Erreurs

- **EDOM** : *x* est négatif.

Sinus, cosinus, tangente

```
double sin ( double x );
double cos ( double x );
double tan ( double x );
```

Note : les angles retournés sont en radians (intervalle -pi/2 à pi/2).

Arc sinus, arc cosinus

```
double asin ( double x );
double acos ( double x );
```

Erreurs

- **EDOM** : *x* est inférieur à -1 ou supérieur à 1.

Arc tangente

```
double atan ( double x );
double atan2 ( double y, double x );
```

Gestion de la mémoire

La gestion dynamique de la mémoire en C se fait à l'aide de principalement deux fonctions de la bibliothèque standard :

- `malloc`, pour l'allocation dynamique de mémoire ;
- `free`, pour la libération de mémoire préalablement allouée avec `malloc`.

Deux autres fonctions permettent de gérer plus finement la mémoire :

- `calloc`, pour allouer dynamiquement de la mémoire, comme `malloc`, qui a préalablement été initialisée à 0 ;
- `realloc`, pour modifier la taille d'une zone mémoire déjà allouée.

Ces fonctions sont déclarées dans l'en-tête `<stdlib.h>`.

Gestion dynamique de la mémoire

Les déclarations de variables en C et dans beaucoup d'autres langages ont une limitation très importante : la taille des variables doit être connue à la compilation. Cela pose problème quand on ne sait qu'à l'exécution le nombre de données qu'on doit traiter. Pour résoudre ce problème, et pouvoir décider durant l'exécution d'un programme du nombre de variables à créer, il faudra nécessairement passer par de l'allocation dynamique de mémoire. Avec ce système, le programmeur dispose de fonctions qui permettent de demander au système une zone mémoire d'une certaine taille, qu'il pourra utiliser comme il le souhaite. En C, ces fonctions sont disponibles dans l'en-tête `<stdlib.h>`.

L'un des principaux avantages qu'offre le langage C est sa capacité à fournir au programmeur un contrôle poussé sur la gestion de la mémoire. Cette liberté nécessite néanmoins une grande rigueur, tant les problèmes pouvant survenir sont nombreux et souvent difficiles à diagnostiquer. On peut dire sans prendre beaucoup de risque que la plupart des erreurs de programmation en C, ont pour origine une mauvaise utilisation des fonctions de gestion de la mémoire. Il ne faut pas sous-estimer la difficulté de cette tâche. Autant cela est trivial pour un programme de quelques centaines de lignes, autant cela peut devenir un casse-tête quand ledit programme a subi des changements conséquents, pas toujours fait dans les règles de l'art.

La manière dont la mémoire physique d'un ordinateur est conçue, ainsi que la façon dont le système d'exploitation la manipule, sont très variables. Cependant, un modèle assez classique consiste à découper la mémoire en segments, segments dont on garde les références dans des tables de pages : c'est le modèle de segmentation / pagination. Ce modèle offre beaucoup d'avantages par rapport à un accès purement linéaire. Décrire son fonctionnement en détail est hors de la porté de cet ouvrage, mais on pourra noter tout de même :

- Indépendance totale de l'espace d'adressage entre les processus : un processus ne peut pas accéder à la mémoire d'un autre processus. C'est pourquoi transmettre la valeur d'un pointeur à un autre processus ne servira en général à rien, car le second processus ne pourra jamais accéder à l'emplacement pointé.
- Gestion fine de la mémoire : les segments sont accédés via plusieurs niveaux d'indirection dans les tables de pages. Cela permet de mettre virtuellement les segments n'importe où dans la mémoire ou même sur un système de fichier. Dans la pratique, éparpiller trop les segments (fragmenter) réduit significativement les performances.

Au niveau des inconvénients, on citera essentiellement un problème de performances. Plusieurs niveaux d'indirection impliquent de multiple lectures en mémoire, extrêmement pénalisant en terme de temps d'exécution, au point où des caches sont nécessaires pour garantir des vitesses acceptables. Même si RAM veut dire mémoire à accès aléatoire, il faut bien garder à l'esprit qu'un accès purement séquentiel (adresse mémoire croissante) peut être jusqu'à cent fois plus rapide qu'une méthode d'accès qui met sans cesse à défaut ce système de cache.

Un processus peut donc demander au système de réserver pour son usage exclusif un secteur mémoire de taille déterminée. Il peut également demander au système de modifier la taille d'une zone précédemment réservée ou de la libérer s'il n'en a plus l'utilité.

Fonctions fournies par <stdlib.h>

`malloc`: allocation de mémoire

La fonction la plus simple d'allocation mémoire est `malloc` :

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

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

Elle prend en argument la *taille* que l'on veut allouer et renvoie un pointeur vers une zone mémoire allouée ou un pointeur nul si la demande échoue (la cause d'erreur la plus courante étant qu'il n'y a plus assez d'espace mémoire disponible). Trois remarques peuvent être faites :

1. `malloc` renvoie une valeur de type `void *`, il n'est pas nécessaire de faire une conversion explicite (cela est nécessaire en C++) ;
2. l'argument *taille* est de type `size_t`, l'appel à la fonction `malloc` devrait toujours se faire conjointement à un appel à l'opérateur `sizeof`.
3. La zone mémoire allouée, si l'appel réussit, n'est pas initialisée. C'est une erreur de conception grave que d'accéder au bloc mémoire en s'attendant à trouver une certaine valeur (0 par exemple).

Par exemple, si on souhaite réserver une zone mémoire pour y allouer un entier :

```
/* Déclaration et initialisation */
int *ptr = NULL;

/* Allocation */
ptr = malloc(sizeof(int));

/* On vérifie que l'allocation a réussi. */
if (ptr != NULL)
{
    /* Stockage de la valeur "10" dans la zone mémoire pointée par ptr */
    *ptr = 10;
}
else
{
    /* décider du traitement en cas d'erreur */
}
```

Si on souhaite allouer de l'espace pour une structure de données plus complexe :

```
typedef struct{
    double b;
    int a;
    MaStructure *suivant;
} MaStructure;

/* Déclaration et initialisation */
MaStructure *ptr = NULL;

/* Allocation */
ptr = (MaStructure *)malloc(sizeof(MaStructure));

if (ptr != NULL)
{
    /* Initialisation de la structure nouvellement créée */
    ptr->a = 10;
    ptr->b = 3.1415;
    ptr->suivant = NULL;
}
```

`free`: libération de mémoire

Le C ne possède pas de mécanisme de ramasse-miettes, la mémoire allouée dynamiquement par un programme doit donc être explicitement libérée. La fonction `free` permet de faire cette libération.

```
void free( void * pointeur );
```

La fonction prend en argument un pointeur vers une zone mémoire précédemment allouée par un appel à malloc, calloc ou realloc et libère la zone mémoire pointée.

Exemple :

```
/* Déclaration et initialisation */
int *ptr = NULL;

/* Allocation */
ptr = malloc(sizeof(int));

/* On vérifie que l'allocation a réussi. */
if (ptr != NULL)
{
    /* ... utilisation de la zone allouée ... */

    /* Libérer la mémoire utilisée */
    free(ptr);
    ptr = NULL; /* Pour éviter les erreurs */
}
else
{
    /* décider du traitement en cas d'erreur */
}
```

L'utilisation du pointeur après libération de la zone allouée (ou la double libération d'une même zone mémoire) est une erreur courante qui provoque des résultats imprévisibles. Il est donc conseillé :

- d'attribuer la valeur nulle (`NULL`) au pointeur juste après la libération de la zone pointée, et à toute autre pointeur faisant référence à la même adresse,
- de tester la valeur nulle avant toute utilisation d'un pointeur.

De plus, donner à `free` l'adresse d'un objet qui n'a pas été alloué par une des fonctions d'allocation cause un comportement indéfini.

`calloc` : allocation avec initialisation à 0

La fonction `calloc` permet d'allouer une zone mémoire dont tous les bits seront initialisés à 0. Son prototype est légèrement différent de celui de `malloc`, et est pratique pour l'allocation dynamique de tableaux.

Syntaxe :

```
void * calloc(size_t nb_element, size_t taille);
```

De manière similaire à `malloc`, `calloc` retourne un pointeur de type `void*` pointant une zone de *`nb_element*taille`* octets allouée en mémoire, dont tous les bits seront initialisés à 0, ou retourne un pointeur nul en cas d'échec.

Exemple :

```
/* allouer un tableau de 5 entiers */
int* ptr = calloc ( 5, sizeof(int) );
```

Le pointeur contient l'adresse du premier élément du tableau :

```
*ptr = 3;   /* premier entier : 3  */
```

Le pointeur peut être utilisé comme un tableau classique pour accéder aux éléments qu'il contient :

```
ptr[0] = 3; /* équivaut à  *ptr = 3;       */
ptr[1] = 1; /* équivaut à  *(ptr+1) = 1;   */
ptr[2] = 4; /* équivaut à  *(ptr+2) = 4;   */
```

Notez que `calloc` place tous les *bits* à zéro, mais que ce n'est pas nécessairement une représentation valide pour un pointeur nul ni pour le nombre zéro en représentation flottante. Ainsi, pour initialiser à zéro un tableau de `double` de manière portable, par exemple, il est nécessaire d'assigner la valeur `0.0` à chaque élément du tableau. Étant donné qu'on initialise chaque élément « manuellement », on peut dans ce cas utiliser `malloc` plutôt que `calloc` (la première étant normalement beaucoup plus rapide que la seconde).

`realloc`

Il arrive fréquemment qu'un bloc alloué n'ait pas la taille suffisante pour accueillir de nouvelles données. La fonction `realloc` est utilisée pour changer (agrandir ou réduire) la taille d'une zone allouée par `malloc`, `calloc`, ou `realloc`.

Syntaxe :

```
void * realloc(void * ancien_bloc, size_t nouvelle_taille);
```

`realloc` tentera de réajuster la taille du bloc pointé par *ancien_bloc* à la nouvelle taille spécifiée. À noter :

- si *nouvelle_taille* vaut zéro, l'appel est équivalent à `free(ancien_bloc)`.
- si *ancien_bloc* est nul, l'appel est équivalent à `malloc(nouvelle_taille)`.

- En cas de succès, `realloc` alloue un espace mémoire de taille *nouvelle_taille*, copie le contenu pointé par le paramètre *pointeur* dans ce nouvel espace (en tronquant éventuellement si la nouvelle taille est inférieure à la précédente), puis libère l'espace pointé et retourne un pointeur vers la nouvelle zone mémoire.

- En cas d'échec, cette fonction ne libère pas l'espace mémoire actuel, et retourne une adresse nulle.

Notez bien que `realloc` ne peut que modifier des espaces mémoires qui ont été alloués par `malloc`, `calloc`, ou `realloc`. En effet, autoriser `realloc` à manipuler des espaces mémoires qui ne sont pas issus des fonctions de la bibliothèque standard pourrait causer des erreurs, ou des incohérences graves de l'état du processus. En particulier, les tableaux, automatiques comme statiques, ne peuvent être passés à `realloc`, comme illustré par le code suivant :

⚠ Ce code contient **une erreur volontaire** !

```
void f(void)
{
    int tab[10];
    /* ... */
    int *ptr = realloc(tab, 20 * sizeof(int));
    /* ... */
}
```

Lorsque `realloc` reçoit la valeur de `tab`, qui est un pointeur sur le premier élément (i.e. `&tab[0]`), il ne peut la traiter, et le comportement est indéfini. Sur cet exemple, il est facile de voir l'erreur, mais dans l'exemple suivant, la situation est plus délicate :

```c
#include <stdint.h> /* pour SIZE_MAX */
#include <stdlib.h>

/* 'double' essaye de doubler l'espace mémoire pointé par ptr.
 *
 * En cas de succès, la valeur renvoyée est un pointeur vers le nouvel espace mémoire, et l'ancienne
 * valeur de ptr est invalide.
 * En cas d'échec, l'espace pointé par ptr n'est pas modifié, et la valeur NULL est renvoyée.
 */
void *double(void *ptr, size_t n)
{
    void *tmp = NULL;
    if ((ptr != NULL) && (n != 0) && (n <= SIZE_MAX / 2))
    {
        tmp = realloc(ptr, 2 * n);
    }
    return tmp;
}
```

La fonction `double` en elle-même ne comporte pas d'erreur, mais elle peut causer des plantages suivant la valeur de `ptr` qui lui est passée. Pour éviter des erreurs, il faudrait que la documentation de la fonction précise les contraintes sur la valeur de `ptr`... et que les programmeurs qui l'utilisent y fassent attention.

On peut aussi noter que, quand `realloc` réussit, le pointeur renvoyé peut très bien être égal au pointeur initial, ou lui être différent. En particulier, il n'y a aucune garantie que, quand on diminue la taille de la zone mémoire, il le fasse « sur place ». C'est très probable, car c'est ce qui est le plus facile et rapide à faire du point de vue de l'implémentation, mais rien ne l'empêche par exemple de chercher un autre espace mémoire disponible qui aurait exactement la taille voulue, au lieu de garder la zone mémoire initiale.

On peut noter le test `(n <= SIZE_MAX / 2)`. Il permet d'éviter un *débordement entier* : si n était supérieur à cette valeur, le produit `n * 2` devrait avoir une valeur supérieure à `SIZE_MAX`, qui est la plus grande valeur représentable par le type `size_t`. Lorsque cette valeur est passée à `realloc`, la conversion en `size_t`, qui est un type entier non signé, se fera modulo `SIZE_MAX + 1`, et donc la fonction recevra une valeur différente de celle voulue. Si le test n'était pas fait, `double` pourrait ainsi retourner à l'appelant une zone mémoire de taille inférieure à celle demandée, ce qui causerait des bogues. Ce genre de bogue (non spécifique à `realloc`) est très difficile à détecter, car n'apparaît que lorsque l'on atteint des valeurs limites, ce qui est assez rare, et le problème peut n'être visible que bien longtemps après que l'erreur de calcul soit faite.

Gestion d'erreur

Pour gérer correctement le cas d'échec, on ne peut faire ainsi :

⚠ Ce code contient **une erreur volontaire** !

```c
int *ptr = malloc(10 * sizeof(int));
if (ptr != NULL)
{
    /* ... */

    /* On se rend compte qu'on a besoin d'un peu plus de place. */
    ptr = realloc(ptr, 20 * sizeof(int));
    /* ... */
}
```

En effet, si `realloc` échoue, la valeur de `ptr` est alors nulle, et on aura perdu la référence vers l'espace de taille `10 * sizeof(int)` qu'on a déjà alloué. Ce type d'erreur s'appelle une *fuite mémoire*. Il faut donc faire ainsi :

```c
int *ptr = malloc(10 * sizeof(int));
```

```
if (ptr != NULL)
{
    /* ... */

    /* On se rend compte qu'on a besoin d'un peu plus de place. */
    int *tmp = realloc(ptr, 20 * sizeof(int));
    if (tmp == NULL)
    {
        /* Exemple de traitement d'erreur minimaliste */
        free(ptr);
        return EXIT_FAILURE;
    }
    else
    {
        ptr = tmp;
        /* On continue */

    }
}
```

Exemple

`realloc` peut être utilisé quand on souhaite boucler sur une entrée dont la longueur peut être indéfinie, et qu'on veut gérer la mémoire assez finement. Dans l'exemple suivant, on suppose définie une fonction `lire_entree` qui :

- lit sur une entrée quelconque (par exemple `stdin`) un entier, et renvoie 1 si la lecture s'est bien passée ;
- renvoie 0 si aucune valeur n'est présente sur l'entrée, ou en cas d'erreur.

Cette fonction est utilisée pour construire un tableau d'entiers issus de cette entrée. Comme on ne sait à l'avance combien d'éléments on va recevoir, on augmente la taille d'un tableau au fur et à mesure avec `realloc`.

```
/* Fonction qui utilise 'lire_entree' pour lire un nombre indéterminé
 * d'entiers.
 * Renvoie l'adresse d'un tableau d'entiers (NULL si aucun entier n'est lu).
 * 'taille' est placé au nombre d'éléments lus (éventuellement 0).
 * 'erreur' est placée à une valeur non nulle en cas d'erreur, à une valeur nulle sinon.
 */
int *traiter_entree(size_t *taille, int *erreur)
{
    size_t max = 0;   /* Nombre d'éléments utilisables */
    size_t i = 0;     /* Nombre d'éléments utilisés */
    int *ptr = NULL;  /* Pointeur vers le tableau dynamique */
    int valeur;       /* La valeur lue depuis l'entrée */
    *erreur = 0;

    while (lire_entree(&valeur) != 0)
    {
        if (i >= max)
        {
            /* Il n'y a plus de place pour stocker 'valeur' dans 'ptr[i]' */
            max = max + 10;
            int *tmp = realloc(ptr, max * sizeof(int));
            if (tmp == NULL)
            {
                /* realloc a échoué : on sort de la boucle */
                *erreur = 1;
                break;
            }
            else
            {
                ptr = tmp;
            }
        }
        ptr[i] = valeur;
        i++;
    }
    *taille = i;
    return ptr;
}
```

Ici, on utilise `max` pour se souvenir du nombre d'éléments que contient la zone de mémoire allouée, et `i` pour le nombre d'éléments effectivement utilisés. Quand on a pu lire un entier depuis l'entrée, et que `i` vaut `max`, on sait qu'il

n'y a plus de place disponible et qu'il faut augmenter la taille de la zone de mémoire. Ici, on incrémente la taille `max` de 10 à chaque fois, mais il est aussi possible de la multiplier par 2, ou d'utiliser toute autre formule. On utilise par ailleurs le fait que, quand le pointeur envoyé à `realloc` est nul, la fonction se comporte comme `malloc`.

Le choix de la formule de calcul de `max` à utiliser chaque fois que le tableau est rempli résulte d'un compromis :

- augmenter `max` peu à peu permet de ne pas gaspiller trop de mémoire, mais on appellera `realloc` très souvent.
- augmenter très vite `max` génère relativement peu d'appels à `realloc`, mais une grande partie de la zone mémoire peut être perdue.

Une allocation mémoire est une opération qui peut être coûteuse en terme de temps, et un grand nombre d'allocations mémoire peut fractionner l'espace mémoire disponible, ce qui alourdit la tâche de l'allocateur de mémoire, et au final peut causer des pertes de performance de l'application. Aucune formule n'est universelle, chaque situation doit être étudiée en fonction de différents paramètres (système d'exploitation, capacité mémoire, vitesse du matériel, taille habituelle/maximale de l'entrée...). Toutefois, l'essentiel est bien souvent d'avoir un algorithme qui marche, l'optimisation étant une question secondaire. Dans une telle situation, utilisez d'abord une méthode simple (incrémentation ou multiplication par une constante), et n'en changez que si le comportement du programme devient gênant.

Problèmes et erreurs classiques

Défaut d'initialisation d'un pointeur

Pour éviter des erreurs, un pointeur devrait **toujours** être initialisé lors de sa déclaration ; soit à NULL, soit avec l'adresse d'un objet, soit avec la valeur de retour d'une fonction « sûre » comme `malloc`. Méditons sur l'exemple suivant :

⚠ Ce code contient **une erreur volontaire** !

```
/* Déclaration sans initialisation */
int *ptr;
int var;

/* On stocke dans var la valeur de la zone mémoire pointée par ptr*/
var = *ptr;
```

Ce code va compiler! Si on est chanceux l'exécution de ce code provoquera une erreur à la troisième étape lorsqu'on déréférence `ptr`. Si on l'est moins, ce code s'exécute sans souci et stocke dans `var` une valeur aléatoire, ce qui pourrait provoquer une erreur lors d'une utilisation ultérieure de la variable `var` alors que l'erreur réelle se situe bien plus en amont dans le code! Une variable (automatique) de type pointeur est en effet comme toutes les autres variables : si elle n'est pas initialisée explicitement, son contenu est indéfini ; son déréférencement peut donc causer n'importe quel comportement.

Lorsqu'on déclare une variable ou qu'on alloue une zone mémoire, on ne dispose d'**aucune** garantie sur le contenu de cette zone ou de cette variable. Initialiser systématiquement les variables dès leur déclaration, en particulier lorsqu'il s'agit de pointeurs, fait partie des bons réflexes à prendre et pour la plupart des applications ne dégrade pas le temps d'exécution de manière significative. Ce type d'erreur peut être extrêmement difficile à localiser à l'intérieur d'un programme plus complexe.

Référence multiple à une même zone mémoire

La recopie d'un pointeur dans un autre n'alloue pas une nouvelle zone. La zone est alors référencée par deux pointeurs. Un problème peut survenir si on libère la zone allouée sans réinitialiser **tous** les pointeurs correspondants :

⚠ Ce code contient **une erreur volontaire** !

```c
int* ptr = malloc(sizeof(int)); // allouer une zone mémoire pour un entier
int* ptr2 = ptr; // ptr2 pointe la même zone mémoire

*ptr = 5;
printf("%d\n", *ptr2 ); // affiche 5

/* libération */
free( ptr );
ptr = NULL;

*ptr2 = 10; /* <- résultat imprévisible (plantage de l'application, ...) */
```

En règle générale, il faut éviter que plusieurs variables pointent la même zone mémoire allouée.

L'utilisation d'un outil de vérification statique permet de détecter ce genre d'erreur.

Fuite mémoire

La perte du pointeur associé à un secteur mémoire rend impossible la libération du secteur à l'aide de free. On qualifie cette situation de *fuite mémoire*, car des secteurs demeurent réservés sans avoir été désalloués. Cette situation perdure jusqu'à la fin du programme principal. Voyons l'exemple suivant :

⚠ Ce code contient **une erreur volontaire** !

```c
int i = 0;       // un compteur
int* ptr = NULL; // un pointeur
while (i < 1001) {
    ptr = malloc(sizeof(int)); // on écrase la valeur précédente de ptr par une nouvelle
    if (ptr != NULL)
    {
       /* traitement ....*/
    }
}
```

À la sortie de cette boucle on a alloué un millier d'entiers soit environ 2000 octets, que l'on ne peut pas libérer car le pointeur ptr est écrasé à chaque itération (sauf la dernière). La mémoire disponible est donc peu à peu grignotée jusqu'au dépassement de sa capacité. Ce genre d'erreurs est donc à proscrire pour des processus tournant en boucle pendant des heures, voire indéfiniment. Si son caractère cumulatif la rend difficile à détecter, il existe de nombreux utilitaires destinés à traquer la moindre fuite.

Gestion des signaux

Les signaux permettent une communication, assez sommaire, entre le système et un processus, ou entre différents processus. Cette communication est sommaire, car un *signal* ne porte qu'une seule information: son numéro, de type `int`. Un processus peut aussi s'envoyer un signal à lui-même.

Ces signaux sont envoyés de manière *asynchrone*: lorsqu'un processus reçoit un signal, son exécution est interrompue, et une fonction spécifique, dite *gestionnaire de signal*, est appelée, avec en paramètre le numéro du signal reçu, pour traiter l'événement. Lorsque cette fonction se termine, le processus reprend là où il s'était arrêté.

C'est à chaque processus de déterminer ce qu'il fait quand il reçoit un signal de numéro donné, en définissant un gestionnaire de signal pour tous les signaux qu'il le souhaite. Vous l'aurez remarqué, dans tous les exemples de ce livre rencontrés jusqu'à présent, nous ne nous sommes pas occupés de savoir quels signaux nous pourrions recevoir, et comment il fallait les traiter. En l'absence de précisions dans nos programmes, l'implémentation fournira en effet un gestionnaire de signal par défaut, qui le plus souvent se contentera d'arrêter le programme.

Les fonctions suivantes, fournies par l'en-tête `<signal.h>`, sont utilisées dans la gestion des signaux :

- `signal()` pour définir un gestionnaire de signal;
- et `raise()` pour envoyer un signal au processus courant.

Le C définit ces deux fonctions, et pas plus, alors que les signaux peuvent faire beaucoup plus... En particulier, aucune fonction pour envoyer un signal à un autre processus n'est définie. Cela est dû au fait que, sur certains systèmes, la notion de processus n'existe pas, et une seule tâche s'exécute. Dans de telles situations, il serait aberrant de demander à un compilateur C pour un tel système de fournir des moyens de communications inter-processus ! Par contre, les communications inter-processus étant chose courante dans de nombreux autres systèmes, des extensions fournissent des moyens de le faire (voir par exemple la fonction `kill()` définie dans POSIX, cf. le chapitre Gestion des signaux du livre Programmation POSIX).

Définir un gestionnaire de signal

La fonction

```
void (*signal(int sig, void (*func)(int)))(int);
```

permet de définir un gestionnaire de signal. La signature de cette fonction étant un peu complexe, on peut la simplifier en utilisant un `typedef`:

```
typedef void (*t_handler)(int);

t_handler signal(int sig, t_handler func);
```

Le comportement de signal() varie selon les versions d'Unix, et a aussi varié au cours du temps dans les différentes versions de Linux. Évitez de l'utiliser : utilisez plutôt sigaction(2).

Ainsi, on voit mieux comment fonctionne cette fonction (notez que le type `t_handler` n'est pas défini par la norme, et n'est ici que pour clarifier la syntaxe). Elle prend deux paramètres:

- le numéro du type de signal à traiter;
- et un pointeur vers la fonction de gestion du signal.

Le premier paramètre est le numéro du signal. La norme définit un faible nombre de signaux, par des macros dans l'en-tête `<signal.h>`, par exemple `SIGSEGV` qui indique un accès illégal à la mémoire (*SEGmentation Violation*), et laisse à l'implémentation la liberté de définir d'autres types de signaux.

Pour le deuxième paramètre, on peut donner trois valeurs possibles:

- `SIG_IGN` (macro définie dans `<signal.h>`): tout signal ayant le numéro `sig` sera ignoré (i.e. rien ne se passera, le programme continue de s'exécuter);
- `SIG_DFL` (id.): le gestionnaire de signal par défaut sera mis en place pour ce type de signal.
- un pointeur vers une fonction de type `t_handler`: cette fonction sera appelée lorsqu'un signal de type `sig` sera reçu par le processus.

Cette fonction renvoie la valeur du dernier gestionnaire de signal pour le numéro donné (qui peut être `SIG_IGN` ou `SIG_DFL`).

En cas d'erreur, la fonction renvoie `SIG_ERR` et place la variable `errno` à une valeur significative.

Les gestionnaires de signaux

Les gestionnaires de signaux sont des fonctions au prototype simple :

```
void fonction(int);
```

Elles prennent un argument de type `int`, qui est le numéro du signal, et ne renvoient rien. En effet, ces fonctions étant appelées de manière asynchrone, leur éventuelle valeur de retour ne serait récupérée, et encore utilisée, par personne...

Comme on utilise `signal` pour associer à chaque numéro de signal une fonction qui va gérer les signaux de ce numéro, on pourrait penser qu'il n'est pas nécessaire de donner ce numéro en paramètre à la fonction. Cependant cela s'avère pratique, car on peut ainsi définir une seule fonction qui peut gérer plusieurs types de signaux, et dont le comportement variera en fonction du signal à traiter.

Voici un exemple simple de définition et de mise en place d'un gestionnaire de signaux, pour le type de signal `SIG_FPE`, qui concerne des exceptions lors des calculs en virgule flottante (*Floating Point Exception*):

Le comportement de signal() varie selon les versions d'Unix, et a aussi varié au cours du temps dans les différentes versions de Linux. Évitez de l'utiliser : utilisez plutôt sigaction(2).

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

void sig_fpe(int sig)
{
    /* ... */
}

int main(void)
{
    if (signal(SIG_FPE, sig_fpe) == SIG_ERR)
    {
        puts("Le gestionnaire de signal pour SIG_FPE n'a pu etre defini.");
    }
    else
    {
        puts("Le gestionnaire de signal pour SIG_FPE a pu etre defini.");
    }
    /* ... */
    return 0;
}
```

Conclusion

Le C est un langage plein de paradoxes. Pouvant aussi bien s'accommoder d'applications bas niveau que proposer des interfaces relativement proches des méthodes orientées objets, ce langage a de quoi séduire un large public. Pourtant, force est de constater qu'il a fait souffrir beaucoup de personnes et pas seulement les programmeurs. Issu de l'époque où la mémoire et les capacités de calcul étaient encore des denrées rares et chères, les pionniers de la programmation ont trop souvent privilégié le côté bas niveau de ce langage. Il en résulta des applications difficilement maintenables, abusant des aspects les plus complexes ou non-portables qu'offre le C pour économiser le moindre cycle processeur. C'est de cette époque que sa réputation « d'assembleur plus compliqué et plus lent que l'assembleur » s'est forgée, éclipsant ses aspects de plus haut niveau, qui pourtant méritent une meilleure estime.

Les fonctions de la bibliothèque standard et surtout le langage lui-même contiennent beaucoup de « pièges », notamment par les comportements indéfinis, ou dépendants de l'implémentation, qui rendent des bogues parfois très difficiles à découvrir. Le programmeur doit comprendre que le C lui fournit de grandes possibilités, mais qu'en retour il lui demande une rigueur d'autant plus grande. Apprivoiser ce langage peut nécessiter une longue phase d'apprentissage. Nous espérons que cet ouvrage vous aura permis d'appréhender le C avec plus de sérénité et de casser certains mythes qui ont décidément la vie bien dure.

Il existe beaucoup de bibliothèques qui proposent au programmeur C des outils pour étendre la bibliothèque standard, et permettent ainsi la gestion d'interfaces graphiques, la programmation réseau, le gestion de l'internationalisation, etc. Celles-ci sont très riches, et bien trop nombreuses pour être abordées dans ce livre. Wikilivres propose des livres étudiant certaines de ces bibliothèques. Pour les autres, Internet et ses moteurs de recherche restent de loin les meilleurs outils pour trouver ce dont vous aurez nécessairement besoin.

Le langage C a su s'adapter, d'Unix aux téléphones mobiles, de l'ISO-646 à Unicode. Le C est aujourd'hui un langage encore très utilisé. C'est *LE* langage de développement de la plupart des micro-contrôleurs actuels. Il a remplacé l'assembleur pour les petites applications. Pour les « grosses » applications tournant sur des OS (Windows, Linux, etc…), il est concurrencé par d'autres langages comme son dérivé le C++, un langage objet encore plus subtil mais permettant tout de même le développement d'applications complexes.

Voir aussi

- Autotools en C

Notes

1. ↑ `errno` peut être définie par une macro ou un identificateur. N'essayez donc pas de récupérer son adresse.

Bibliographie

Vous trouverez ici des références d'ouvrages ou de sites internet ayant servi à l'élaboration de ce wikilivre, ou pouvant être d'intérêt pour une connaissance plus approfondie du C.

Site officiel du WG14 et références normatives

- Le site officiel du WG14 (http://www.open-std.org/jtc1/sc22/wg14/) [archive] anglais. Contient de nombreuses informations sur le C, dont les dernières propositions en discussion pour l'évolution de la norme. Attention toutefois, ces documents sont globalement très techniques. On peut noter la présence du *Rationale* pour la norme C99, qui lui est abordable.
  - Le TC3 n'est pas disponible sur le site du WS, mais sur celui de l'ISO (http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50510) [archive].
- *Rationale* (http://www.lysator.liu.se/c/rat/title.html) [archive] de la norme ANSI X3.159-1989 anglais (absent du site du WG14)
- La page (http://clc-wiki.net/wiki/C_standardisation:ISO) [archive] du wiki du groupe de discussion comp.lang.c référençant les documents normatifs et les moyens d'acquérir la norme anglais.

Livres

- Le « K&R »:
  - Brian W. Kernighan et Dennis M. Ritchie. *Le Langage C: Norme ANSI*, 2e éd., Dunod, (ISBN 2-100-05116-4) [présentation en ligne (http://c.developpez.com/livres/#L2100487345) [archive]]
  - anglais Brian W. Kernighan et Dennis M. Ritchie. *The C Programming Language, Second Edition*. Prentice Hall, Inc., 1988. (ISBN 0-13-110362-8) [présentation en ligne (http://cm.bell-labs.com/cm/cs/cbook/) [archive]]
- Le site Developpez.com (http://www.developpez.com) [archive] propose des critiques de livres sur le C ici (http://c.developpez.com/livres) [archive] (la page contient aussi des livres sur le C++).

Compilateurs C

- Le wiki de comp.lang.c liste ici (http://clc-wiki.net/wiki/C_resources:Compilers) [archive] un certain nombre de compilateurs en précisant leur support des normes C90 et C99 anglais.

Autres ressources

Tutoriels

- Les bases du langage C (http://neofutur.net/langage_c/examples_language_C/langage_C_les_bases/langage_C_bases.html) [archive] par William Waisse.

Histoire du C

- The Development of the C Language (http://cm.bell-labs.com/cm/cs/who/dmr/chist.html) [archive] par Dennis Ritchie anglais.
- Historique de C (http://marc.mongenet.ch/Articles/C/index.html) [archive] par Marc Mongenet.

Licence

Licence de documentation libre GNU

Récupérée de « http://fr.wikibooks.org/w/index.php?title=Programmation_C/Version_imprimable&oldid=445563 »