Langage C

par **Christian QUEINNEC**
Professeur à l'université Pierre-et-Marie-Curie

Le langage C a été défini en 1972 par Denis Ritchie, chercheur des laboratoires Bell, dans le cadre d'un projet qui visait à réécrire le système d'exploitation Unix dans un langage de haut niveau. Le langage C a été fortement influencé par son premier client : Unix, et nombre de ses caractéristiques proviennent des besoins suscités par le développement de ce système. Parmi ces caractéristiques figurent :

— la compacité d'écriture visant à minimiser la frappe des programmes (ainsi écrit-on {et} pour `begin` et `end`). La rapidité de programmation a été préférée à la lisibilité, l'inverse étant requis dans un langage comme Ada ;

— l'absence de bibliothèque obligatoire d'exécution rejetant la manipulation des entrées/sorties, la gestion de tâches, l'allocation dynamique de mémoire, etc., dans des bibliothèques extérieures, supplémentaires et optionnelles ;

— la proximité sémantique avec la machine d'exécution privilégiant systématiquement un contrôle fin de celle-ci au détriment de toute abstraction opaque. Citons en particulier un système de types débrayable, l'absence de booléens, l'absence de chaînes de caractères, une grande variété d'entiers de tailles diverses, etc. Cette caractéristique s'accompagne d'un slogan : « Trust the programmer », qui stipule le primat du programmeur sur la machine. En cas d'incompréhension du programme, le compilateur respectera le programme à la lettre plutôt que de le rejeter. C est souvent considéré comme un langage dangereux ou, à tout le moins, « sale » lorsque comparé à d'autres langages de programmation occupant d'autres niches et cherchant à optimiser d'autres critères. La norme ISO de 1990 ainsi que sa révision de 1999 en font un langage de plus en plus sûr. La santé de C montre bien son indéfectible vitalité dans ses niches propres.

C est un « assembleur portable de haut niveau », c'est-à-dire un langage de programmation de haut niveau qui permet également une manipulation précise des entités proches des processeurs. Ses caractéristiques auraient pu le réduire à n'être que le langage d'écriture d'Unix mais, le succès d'Unix aidant et comme C est disponible (par construction) sur tout Unix (puisqu'un portage d'Unix sur une machine X passe par l'écriture d'un générateur de code spécifique de C vers X), C a pu être de plus en plus choisi comme langage d'écriture d'applications générales. C peut être considéré comme un langage normal de programmation avec son lot de singularités dont les plus délicates concernent les références et l'allocation.

C occupe aujourd'hui plusieurs niches dans lesquelles sa survie n'est pas menacée. C peut être utilisé pour écrire des programmes hautement portables ou hautement spécifiques.

1. Utilisations

1.1 Assembleur portable

À la différence d'Ada, C n'est pas un langage inhéremment portable. Toutefois, une expérience trentenaire de portage d'Unix et/ou de tous ses utilitaires (quelques millions de lignes de code) sur quelques centaines de types d'ordinateurs ou de systèmes d'exploitation différents ont permis d'élaborer un sous-ensemble pragmatiquement portable de C.

Ce sous-ensemble, qui correspond au côté « assembleur portable », est utilisé pour écrire des applications ou des bibliothèques destinées à fonctionner sur de nombreuses machines. Citons dans cette catégorie les utilitaires GNU, le système de fenêtrage X Window, les bibliothèques d'accès aux bases de données, le serveur HTTP Apache, etc. Ce sous-ensemble peut également être employé comme cible de compilateur traduisant un langage de très haut niveau en un langage plus simple : C. L'existence de compilateurs C pour quasiment toute machine existante permet aux compilateurs de langages de haut niveau (Eiffel, Scheme (Bigloo), lex, yacc...) de s'affranchir des problèmes de bas niveau (allocation de registres, gestion des coprocesseurs, optimisations diverses, etc.) et de n'avoir qu'un unique compilateur à maintenir.

L'absence en C de bibliothèque d'exécution a aussi un effet bénéfique sur les applications écrites en C : aucune redevance n'est due ! En contrepartie, ni gestion élaborée de mémoire (GC pour glaneur de cellules), ni possibilité d'introspection à l'exécution ne sont procurées par défaut.

1.2 Assembleur de haut niveau

Une deuxième niche, d'intention assez opposée, est l'écriture de programmes spécifiques non portables. L'aspect « assembleur de haut niveau » est alors privilégié, permettant à la fois un contrôle fin de la machine (adresses fixées, instructions particulières, représentations en mémoire précises) et l'emploi de structures de contrôle de haut niveau (boucle `while`, fonctions), de structures de données de haut niveau (tableau, enregistrement) et d'un système de types assurant une certaine cohérence (lorsque bien employé). Cette caractéristique de C est mise en œuvre pour l'écriture de pilotes (*drivers*) ou de systèmes enfouis (*embedded systems*) comme, par exemple, l'écriture d'applications pour Palm OS.

1.3 Langage de couplage

Une dernière niche existe, de nature plus abstraite car ne conduisant pas nécessairement à du code écrit en C. Le modèle d'exécution et le modèle mémoire de C sont de plus en plus considérés comme les seuls procédés effectifs d'échange entre langages ou systèmes différents. La description d'une interface en C permet à deux langages différents (Ada et Python ou C++ et Java par exemple) de décrire comment s'appeler l'un l'autre et comment échanger des informations par passage au dénominateur commun que constitue C.

C est à la base de la compréhension des mécanismes de RPC (Remote Procedure Call) et XDR (eXternal Data Representation), à la base de tous les interstitiels (*middlewares*) tels que Corba ou DCOM qui permettent d'effectuer des appels à des fonctions lointaines, c'est-à-dire situées dans des espaces mémoires disjoints, voire même sur d'autres machines.

Pour résumer, C est un langage de programmation ayant des capacités d'**assembleur portable de haut niveau**.

De l'**assembleur**, il garde la précision et la fidélité : imaginer l'assembleur produit par un morceau de C est, aux optimisations près, trivial, alors que celui produit par un morceau de C++, de ML ou de Prolog est affaire de spécialiste. Cette fidélité, qui permet d'apprécier les coûts d'exécution et autorise une mise au point plus aisée sur un matériel nouveau, est la raison principale pour laquelle les noyaux de systèmes d'exploitation sont écrits en C et non en langages de plus haut niveau.

En terme de **portabilité**, C a depuis longtemps fait ses preuves, qui ont dépassé l'Unix d'origine. L'existence d'un noyau portable bien compris est essentielle pour les échanges interlangages qui deviennent inévitables depuis l'avènement des réseaux et de la programmation par composants.

En ce qui concerne les langages de programmation, le **haut niveau** commence avec C qui devient de fait le minimum linguistique requis, à tel point que la syntaxe des nouveaux langages (Java, Perl, Tcl) calque celle de C, et qu'aucun nouvel ordinateur n'est créé sans compilateur C.

C a été longtemps défini par le seul livre de Kernighan et Richie (K&R) [1] et par le code du compilateur générique *pcc* (*portable C compiler*) qui a longtemps été la souche unique des compilateurs C, prévenant ainsi l'apparition de dialectes. C a fait l'objet de la norme ISO 9899 (publiée en 1990, révisée en 1999), souvent nommée inappropriément ANSI C à la place de C ISO. Cette norme a clarifié de nombreux points sémantiques, a donc augmenté la taille du langage mais l'a rendu, en même temps, beaucoup plus sûr par l'ajout d'un système de types (débrayable). La dernière version de la norme C, dite C99, n'est pas encore vraiment en usage en 2001. Aussi nous cantonnerons-nous ici à la version de 1990, dite C90.

2. Compilation et macroexpansion

Le langage C est principalement mis en œuvre par **compilation**, bien qu'il existe **quelques interprètes** de C comme EiC. Le fichier est l'unité de compilation. Le compilateur travaille en plusieurs passes. Les toutes premières sont très simples, comme l'élimination des barres de fraction inverses en bout de ligne, suivie de l'élimination des commentaires (qui débutent par `/*` et s'achèvent par `*/`).

La première passe vraiment importante correspond à un macroexpanseur nommé *cpp* (*C pre-processor*). De fait, un fichier destiné à un compilateur C est un mélange de deux langages différents, c'est-à-dire un fond de C structuré par des directives de macroexpansion. Le résultat de la macroexpansion est du C pur qui sera compilé.

Le **macroexpanseur** *cpp* est un outil modeste mais général pouvant être utilisé indépendamment de C pour traiter tout type de textes.

Un macroexpanseur est un filtre qui prend un fichier en entrée, le lit et l'expanse afin de produire un autre fichier. Expanser un fichier consiste à identifier les directives de macroexpansion qui s'y trouvent et à les exécuter séquentiellement. Les directives permettent, entre autres, de réaliser des inclusions de textes, de spécifier des zones à traiter ou à ignorer (expansion conditionnelle), de définir des constantes ou des fonctions d'expansion.

Les directives débutent par un dièse # et s'achèvent en fin de ligne.

2.1 Inclusion de fichiers

La directive la plus simple est la directive permettant d'inclure un fichier. Elle peut prendre deux formes distinctes :

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

Une ligne comportant une directive `include` est remplacée par le contenu du fichier mentionné. La première forme, où le nom du fichier est entre guillemets, correspond à l'inclusion d'un fichier de l'utilisateur. La seconde forme correspond à l'inclusion d'un fichier propre au système. La différence entre ces deux formes est que le fichier est recherché dans des suites de répertoires différentes. Les fichiers inclus sont également macroexpansés. L'inclusion de fichiers est fondamentale pour C car elle permet de lutter contre la redondance d'informations : il est ainsi loisible de n'écrire une information qu'une seule fois mais de l'utiliser plusieurs fois.

2.2 Expansion conditionnelle

La portabilité d'un programme passe par l'identification des seules portions de programme qui sont à adapter pour telle ou telle machine. Ces portions gagnent à être courtes et concentrées en quelques points bien identifiés. Afin toujours de lutter contre la redondance et de ne pas dupliquer le code commun, on use de

compilation conditionnelle qui permet selon telle ou telle condition de spécifier le code à compiler. On peut également, grâce à la compilation conditionnelle, instrumenter son code pour mesurer, mettre au point, tester, tracer ses programmes. L'alternative (« si-alors-sinon ») des langages de programmation permet cet effet.

```
#if condition
  ...
#else
  ...
#endif
```

Les lignes comprises entre `if` et `else` forment la partie « alors », tandis que les lignes entre `else` et `endif` forment la partie « sinon ». Cette dernière partie peut être omise, elle est alors considérée comme vide.

Le macroexpanseur commence par évaluer la condition. Si sa valeur est vraie, alors l'alternative (c'est-à-dire toutes les lignes comprises entre `if` et `endif`) est remplacée par la partie « alors », sinon elle est remplacée par la partie « sinon ». Les lignes remplaçant l'alternative toute entière sont elles-mêmes macroexpansées.

La condition est une expression du langage du macroexpanseur qui, hormis la syntaxe, n'a que peu à voir avec le langage C. Les expressions de *cpp* sont elles-mêmes macroexpansées et ne peuvent manipuler que des entiers (longs). Un opérateur spécial existe, nommé `defined`, qui permet de tester si une constante est définie pour *cpp*. L'opérateur vaut zéro (faux) si la constante n'est pas définie et autre chose que zéro (vrai) sinon (on retrouve là le codage des booléens de C). L'argument de l'opérateur `defined` n'est bien sûr pas macroexpansé.

La directive `if` généralise les directives plus anciennes que sont `ifdef` et `ifndef`. La directive `if defined` (*`constante`*) peut remplacer la directive `ifdef` *`constante`*, tandis que similairement la directive `if ! defined` (*`constante`*) remplace `ifndef` *`constante`* où `!` est l'opérateur de négation booléenne (comme en C).

Pour rendre plus lisibles des imbrications profondes d'alternatives, existe la directive `elif` qui correspond à un `else if` mais avec un unique `endif` au lieu de deux. Il est aussi possible d'insérer des blancs entre le dièse initial et le nom de la directive afin de les apparier visuellement.

2.3 Constantes de macroexpansion

Il est possible de définir des constantes de macroexpansion.

```
#define constante lexèmes...
```

La directive `define` définit une constante de macroexpansion. Chaque ligne qui suit cette directive sera analysée et si elle contient un lexème de nom *`constante`*, celui-ci sera remplacé par les lexèmes qui le suivent dans sa définition. Lorsqu'une constante de macroexpansion a été définie, l'opérateur spécial `defined` appliqué à cette constante vaut vrai (non-zéro).

La lexémisation qu'effectue *cpp* est calquée sur celle de C de manière à assurer une certaine compatibilité. En particulier, les chaînes de caractères de C sont laissées inaltérées par *cpp*.

La directive `define` est fondamentale pour C car elle permet de résoudre le problème des constantes dérivées.

Lorsque l'on introduit une constante magique comme 132 (le nombre de caractères que pouvait imprimer en une seule ligne une imprimante rapide d'avant l'avènement des imprimantes lasers A*4*), il est usuel d'avoir des traitements qui utiliseront ce nombre ou un nombre dérivé de celui-ci.

```
#define TAILLE 132

   ...
   {
      /* un tableau de TAILLE caractères */
      char ligne[TAILLE];
      int i;
      for ( i=0 ; i<TAILLE/2 ; i++ ) {
         ligne[i] = ligne[TAILLE-1-i];
      }
      ...
```

Ainsi, un changement de valeur de constante ne sera fait qu'en un seul endroit et mettra à jour toutes les valeurs dérivées (encore un exemple de lutte contre la redondance d'informations par explicitation des dépendances). L'usage d'une constante de macroexpansion permet d'une part de satisfaire C qui impose que la déclaration d'un tableau s'effectue à l'aide d'une constante entière : `TAILLE` ne peut donc être une variable de C. D'autre part, elle permet que le code compilé des opérations mettant en œuvre `TAILLE` utilise des modes d'adressage immédiats (les plus rapides qui soient).

Une constante de macroexpansion peut être définie comme une suite vide de lexèmes auquel cas elle est définie mais est remplacée par rien du tout. La suite de lexèmes définissant une constante de macroexpansion peut n'avoir aucun sens du point de vue de C ! Le premier interprète de commande *sh* écrit par Steve Bourne usait d'un style mimant Pascal. Ce style fut assez décrié et n'est donc pas conseillé.

```
#define BEGIN {
#end    END   }

   ...
   for ( i=0, i<TAILLE/2 ; i++ )
   BEGIN
      ligne[i] = ligne[TAILLE-1-i];
   END
   ...
```

2.4 Macrofonctions

La directive `define` permet également de définir des macrofonctions lorsque le nom défini est immédiatement suivi d'une parenthèse ouvrante. Les macrofonctions permettent de réaliser des expansions paramétrées par des arguments.

L'exemple suivant montre la définition d'une macroconstante nommée `BITSTRING`. Elle représente le type des chaînes de bits que va manipuler la macrofonction `BITSTRING_ISSET` qui ne répond vrai que si le *i*ème bit de *b* est à 1.

```
#define BITSTRING unsigned long

#define BITSTRING_ISSET (b, i)\
   ( (b) & (((BITSTRING)1) << (i)) )
```

Dans cet exemple, la définition a été éclatée en deux lignes, la première est terminée par une barre de fraction inverse. Une passe d'élimination des barres de fraction inverses en bout de ligne, qui regroupe donc ces lignes logiques en une véritable ligne physique, prend place avant l'appel à *cpp*.

La macrofonction `BITSTRING_ISSET` cache les détails d'accès au *i*ème bit (par génération d'un masque que l'on conjoint logiquement avec la chaîne de bits initiale) et permet donc de raisonner à un niveau plus abstrait sans payer le coût d'un appel à une fonction de même spécification.

2.5 Autres directives

Quelques autres directives existent, comme `undef` qui défait ce que `define` avait défini. Il existe aussi quelques possibilités lexicales comme la construction d'un lexème C à partir de lexèmes *cpp* (cet opérateur de concaténation se nomme `##`) ou encore la conversion (grâce à `#`) d'un lexème de *cpp* en une chaîne de caractères pour C.

On peut également forcer une erreur lors de la macroexpansion avec la directive `error`. Enfin, pour aider à la mise au point existent des variables prédéfinies de macroexpansion. Ce sont les suivantes :

`__LINE__` s'expanse en le numéro de la ligne courante ;
`__FILE__` s'expanse en le nom du fichier macroexpansé ;
`__DATE__` s'expanse en la date courante ;
`__TIME__` s'expanse en l'heure courante ;
`__STDC__` s'expanse en la constante 1 si le compilateur est conforme à la norme ISO.

2.6 Conclusion

Le macroexpanseur *cpp* n'est pas très puissant mais s'il est bien employé, il permet de cacher certaines ressources par le biais de macroconstantes ou de macrofonctions (les fichiers *.h* qui décrivent le système contiennent les nombreux exemples de cette technique). Il permet également de factoriser des valeurs ou des fragments de code afin de lutter contre la redondance. Il importe de comprendre que cet outil, bien que toujours présent dans un compilateur C, ne peut effectuer que des substitutions textuelles simples, en amont et indépendamment du véritable compilateur. En particulier, les entités manipulées par le compilateur C (variables, fonctions, tailles de données, etc.) ne sont pas manipulables par *cpp*.

`sizeof(short) == 2` n'est pas une expression compréhensible par *cpp* puisque l'opérateur `sizeof` de C n'est défini que pour une architecture cible du compilateur et que cette architecture n'est pas connue du macroexpanseur.

3. Valeurs manipulées

C est un assembleur portable. Il manipule donc toutes les valeurs que les unités centrales savent manipuler et, inversement, ne manipule que ces valeurs. Cette intention mène à la présence d'une grande variété de nombres entiers (huit) ou flottants (trois), mais également à l'absence de booléens, de chaînes de bits et de chaînes de caractères. En revanche, les structures agrégatives tels que tableaux et enregistrements sont disponibles.

3.1 Flottants

Il existe trois types de nombres flottants en C : `float`, `double` et `long double`. Les nombres flottants sont une approximation finie commode des nombres mathématiques dits réels, ils sont toutefois dénués des bonnes qualités des nombres réels (en particulier l'associativité). Ces trois types entretiennent des rapports d'inclusion en ce sens que toute valeur de type `float` est de type `double` et que toute valeur de type `double` est de type `long double`. Ces types correspondent aux nombres flottants que savent manipuler les unités centrales ; les `long double` ont été introduits dans la norme du fait de la mise sur le marché de processeurs sachant manipuler des nombres flottants plus grands que 64 bit (on peut cependant émuler, par logiciel et de façon relativement efficace, ces grands flottants).

La précision des nombres flottants dépend de leur encodage et de leur taille. La majorité des encodages d'aujourd'hui répond à la norme IEEE 754. Le tableau **1** indique les étendues classiques.

Tableau 1 – Intervalles de variation de nombres flottants

Taille (octets)	Minimum positif	Maximum positif
4	$1{,}175 \cdot 10^{-38}$	$3{,}4 \cdot 10^{38}$
8	$2{,}22 \cdot 10^{-308}$	$1{,}79 \cdot 10^{308}$
10	10^{-4931}	10^{4932}

Le tableau **2** recense quelques tailles typiques en octets. Il illustre bien la diversité des flottants. Les plus grandes variations portent sur la taille (et le codage) des `long double`.

Tableau 2 – Tailles de nombres flottants

Unité centrale, système	`float`	`double`	`long double`
Sparc, Solaris	4	8	16
Pentium, Windows	4	8	10
Pentium, Linux	8	8	12

Les critères de choix entre ces différentes représentations sont : la précision en nombre de décimales souhaitées, l'étendue des nombres représentés (surtout en terme d'exposant), la célérité des opérations arithmétiques et leur taille de stockage en mémoire. Toutes ces caractéristiques sont définies par des constantes de macroexpansion rassemblées dans le fichier *float.h*. Il est ainsi possible d'énoncer des algorithmes conditionnalisés et/ou paramétrés (par macroexpansion) par des types flottants, une version particulière pouvant être choisie à la compilation ou à l'exécution.

C définit un certain nombre d'opérateurs binaires prédéfinis s'appliquant sur tous les types de flottants. Ce sont l'addition +, la soustraction –, la multiplication * et la division /. La bibliothèque standard contient de nombreuses fonctions comme les lignes trigonométriques ou hyperboliques, des fonctions de lecture/écriture, etc.

C est assez permissif en terme de conversion mais on y retrouve le principe de « contagion flottante » qui s'applique dès qu'une opération arithmétique binaire mélange deux natures de flottants : celui dont la représentation est la plus courte adopte le codage de l'autre des arguments. Afin d'être maître des représentations, les littéraux flottants peuvent être suffixés par un F pour signifier

float ou par un L pour signifier long double. L'absence de suffixe est interprétée comme un double.

```
float f;
double d;
long double ld;

f = 2;
d = f + 1.0F;
ld = 3.0F * d;
f = ld + 1e-20L;
```

En première ligne, l'entier 2 est converti en float puis stocké dans la variable f. En deuxième ligne, l'addition ayant deux opérandes de type float, le résultat est un float qui est converti en un double pour être stocké dans d. En troisième ligne, le produit ayant un second argument double force la conversion du littéral 3.0F en un double, le produit est alors calculé en double puis converti en un long double pour être stocké dans la variable ld.

La quatrième et dernière ligne illustre la **règle de préservation des valeurs**. Lorsqu'un grand flottant est stocké dans un petit flottant, deux cas peuvent survenir :

— le grand flottant est compris dans l'intervalle de variation des petits flottants, auquel cas sa valeur est approximée (par réduction du nombre de décimales : c'est une troncature) ;

— le grand flottant ne peut être valablement approximé, auquel cas le résultat est indéterminé et les conséquences sur le programme sont quelconques (le programme peut s'arrêter avec une erreur ou continuer avec une valeur quelconque).

Par exemple, si 1d contient 9.000000000000000001L, cette valeur sera tronquée en 9.0F (préservation de valeur). Si en revanche, ld contient 9e+1000, il n'y a pas de valeur de type float proche (la plus proche est seulement de l'ordre de 1e+38) et le résultat est indéterminé : il ne faut surtout pas continuer à calculer avec cette valeur ! Pour éviter ce problème, une conversion d'un grand flottant vers un petit flottant doit s'accompagner d'une vérification d'appartenance à l'intervalle de variation du petit flottant ou d'une preuve équivalente.

Les opérations classiques +, –, * et / sont applicables aux flottants avec des risques de sortie de l'intervalle de variation desdits flottants. La détection de ces anomalies est laissée au programmeur afin que ceux qui savent ne pas en avoir besoin ne paient pas pour l'incompétence des autres. Fort heureusement, la norme IEEE 754, universellement acceptée, a introduit la notion de NaN (*not a number*) qui permet, en cas d'anomalie, d'engendrer un résultat qui n'est pas un nombre correct. Composer un NaN avec tout autre nombre mène encore à un NaN. Pour tester le non-débordement d'un calcul, il suffit alors de tester que le résultat n'est pas un NaN.

3.2 Entiers

Il existe huit types d'entiers en C. Cette extraordinaire variété provient du souci de manipuler en C tous les entiers avec ou sans signe que les unités centrales procurent. Ces entiers sont généralement de un, deux, quatre ou huit octets. Toutes ces tailles ne sont pas nécessairement procurées par toutes les unités centrales ; ces choix dépendent de l'architecture matérielle (16, 32 ou 64 bit) ainsi que de contraintes de bus internes.

Initialement n'existait que le type int qui correspondait aux entiers tenant sur un registre de l'unité centrale sous-jacente. Puis sont nés les besoins de stocker de petits entiers sur un ou deux octets ou de manipuler des grands entiers (sur huit octets).

Les unités centrales savent manipuler des entiers avec ou sans signe. Les plus petits entiers signés sont les signed char qui tiennent sur un octet. Tous les signed char sont des signed short int (que l'on nomme plus souvent short). Ces entiers courts sont la plupart du temps représentés sur deux octets. Tous les short sont des signed int (plus fréquemment nommés des int). Les machines courantes aujourd'hui procurent généralement des int de quatre octets. Enfin, tous les int sont des signed long int (abrégés en long) qui occupent souvent huit octets.

On note cette progression ainsi :

signed char ⩽ short ⩽ int ⩽ long

Le symbole ⩽ marque l'inclusion au sens large. La norme permet donc que les long soient des int, ce qui signifie qu'ils sont encodés en le même nombre d'octets. Les tailles que nous avons ci-avant mentionnées ne doivent donc pas être tenues comme universelles mais doivent être vérifiées pour la machine et le compilateur particulier utilisés.

C est indépendant de l'encodage particulier choisi pour représenter les entiers signés : complément à 1 ou complément à 2. Ce dernier encodage étant aujourd'hui universel, il a servi de base au tableau **3** qui indique les intervalles de variation de ces entiers.

Ces nombres, adaptés à l'unité centrale vers laquelle on compile, peuvent être retrouvés dans le fichier *limits.h*. On y trouve, par exemple pour une certaine unité centrale, que UINT_MAX, le plus grand des unsigned int, est 4294967295.

Les entiers non signés sont des chaînes de bits de un, deux, quatre ou huit octets considérées comme des nombres entiers naturels, donc positifs. Les plus petits entiers non signés sont les unsigned char qui tiennent sur un octet. Tous les unsigned char sont des unsigned short int (aussi nommés des unsigned short) qui en général tiennent sur deux octets. Tous les unsigned short sont des unsigned int (aussi abrégés en unsigned). Les unsigned int ont en général la taille des registres de l'unité centrale sous-jacente : sur les stations de travail actuelles de 32 bit, un unsigned a une taille de quatre octets. Enfin, tous les unsigned sont des unsigned long int (ou unsigned long) qui peuvent, suivant les machines, avoir des tailles de quatre ou huit octets.

Quelques opérateurs sont prédéfinis pour agir sur les entiers signés ou non signés : l'addition, la soustraction binaire notée – (l'opposé est un opérateur unaire préfixe également noté –), le quotient euclidien noté / et le reste euclidien noté %. Il est peu recommandé d'utiliser quotient et reste euclidiens avec des entiers négatifs (les sémantiques de ces opérateurs ayant une fâcheuse tendance à varier de langage à langage).

Tableau 3 – Intervalles de variation d'entiers

Taille (octets)	Signé		Non signé	
	Minimum	Maximum	Minimum	Maximum
1	– 128	+ 127	0	255
2	– 32768	+ 32767	0	65535
4	– 2147483648	+ 2147483647	0	4294967295
8	– 9223372036854775808	9223372036854775807	0	18446744073709551615

Là encore, les résultats des opérations sur les entiers peuvent sortir des intervalles de variation de ces dits entiers. Aucune vérification n'est faite par le compilateur, c'est au programmeur de s'assurer de la correction de ses calculs. Toutefois, à la différence de l'arithmétique signée, l'arithmétique non signée est modulaire. L'entier non signé qui suit le plus grand entier exprimable (ce que l'on peut exprimer par `UINT_MAX+1`) est zéro. Inversement, zéro moins un redonne le plus grand entier positif non signé.

Les règles de contagion s'adaptent aux nombres entiers signés, elles s'adaptent également aux entiers non signés. Les littéraux entiers peuvent être suffixés par un `U` pour indiquer qu'ils sont non signés et/ou par un `L` pour indiquer qu'ils doivent être codés par des longs entiers. Sans suffixe, un littéral entier est un `int`. Il n'y a pas de possibilité en C de mentionner ou de calculer sur des entiers plus petits qu'un registre, c'est-à-dire plus petits que `int` ou `unsigned int`.

Les littéraux entiers peuvent être écrits en base décimale, en base octale (ils commencent alors par un 0) ou en base hexadécimale (ils débutent alors par `0x` ou `0X`).

La règle de préservation des valeurs s'applique également. Lorsqu'un grand entier (dont la représentation est grande) est stocké dans un petit entier (dont la représentation est plus petite), deux cas peuvent survenir :

— le grand entier appartient à l'intervalle de variation des petits entiers et tout se passe bien ;

— le grand entier n'appartient pas à cet intervalle et le résultat est indéterminé ; les conséquences du programme sont elles-mêmes indéterminées : le programme peut s'arrêter en signalant explicitement une erreur ou continuer sans rien dire avec une valeur quelconque !

Entiers signés et non signés de même taille ont une intersection non vide (ce sont les signés positifs) (tableau **3**). Toutefois, à taille égale, les signés négatifs ne peuvent être représentés par des non signés (positifs). Inversement, les plus grands entiers non signés n'ont pas d'équivalents signés.

Lorsque l'on mélange entiers signés et non signés, les règles de contagion stipulent que la contagion suit cet ordre :

```
signed char ⩽ unsigned char ⩽ signed short
  ⩽ unsigned short ⩽ signed int ⩽ unsigned int
  ⩽ signed long ⩽ unsigned long
```

```
signed short int ssi;
unsigned int ui;
unsigned long ul;

ul = ssi + ui;
```

Dans cette instruction, l'addition convertit son premier opérande vers un `unsigned int` (puisqu'un `signed short int` est plus petit qu'un `unsigned short int` qui est plus petit qu'un `unsigned int`) avant de réaliser l'addition qui produit un `unsigned int` qui est ensuite converti en un `unsigned long` (ce qui est toujours possible).

Mixer ces différents types d'entiers doit faire l'objet de soins attentifs. L'expression `0-1U` vaut probablement 4294967295, c'est-à-dire `UINT_MAX`, tandis que `0-1L` est – 1L.

La règle de préservation des valeurs fait que même les plus petits flottants ont un intervalle de variation plus étendu que les plus grands entiers. Ainsi, mélanger entiers et flottants force les entiers à être convertis en des flottants (avec une éventuelle perte de précision).

3.3 Chaînes de bits

À part leurs capacités arithmétiques (addition, soustraction, multiplication, quotient et reste euclidiens), les entiers non signés sont souvent utilisés comme des chaînes de 8, 16, 32 ou 64 bit sur lesquelles s'exercent les opérateurs sur chaînes de bits qui sont les opérations booléennes classiques : négation, conjonction (*and*), disjonction (*or*), disjonction exclusive (*xor*), sans oublier les divers décalages (logiques, arithmétiques mais pas cycliques) à droite ou à gauche.

C autorise également les entiers signés à se comporter comme des chaînes de bits, le premier bit étant dévolu à la représentation du signe. L'influence de ce bit se fait surtout sentir sur les décalages à droite qui répliquent le bit de signe.

```
unsigned int a = 0xCafe; /* 1100 1010 1111 1110 */
  signed int b = 0x8306; /* 1000 0011 0000 0110 */
                         /* conjonction (et)    */
   a & b                 /* 1000 0010 0000 0110 */
                         /* disjonction (ou)    */
   a | b                 /* 1100 1011 1111 1110 */
               /* disjonction exclusive (xor) */
   a ^ b                 /* 0100 1001 1111 1000 */
                         /* négation (non)      */
   ~ a                   /* 0011 0101 0000 0001 */
                         /* décalage gauche     */
   a << 6                /* 1011 1111 1000 0000 */
                  /* décalage droite logique  */
   a >> 2                /* 0011 0010 1011 1111 */
              /* décalage droite arithmétique  */
   b >> 2                /* 1110 0000 1100 0001 */
```

3.4 Booléens

Il n'y a pas de type booléen en C90. La règle est qu'une valeur ne comportant que des bits à zéro représente faux ; toute autre représentation (comportant donc au moins un bit différent de zéro) vaut vrai. On peut ainsi encoder les booléens avec les entiers signés 0 ou 1, les manipuler 64 par 64 comme des `unsigned long int` ou encore utiliser des références. Aucune représentation n'est imposée, c'est au programmeur de choisir la sienne.

Il y a des opérateurs booléens prédéfinis correspondant à la négation `!` (non), la conjonction `&&` (et) et la disjonction `||` (ou). Ces derniers opérateurs binaires ont une sémantique dite à court-circuit qui assure que lorsque la valeur du résultat final est connue, le calcul s'arrête avec ce résultat. Ainsi, lorsque le premier opérande d'une conjonction (et) est faux, la conjonction entière ne peut être que fausse et le deuxième opérande n'est pas calculé. Symétriquement, lorsque le premier opérande d'une disjonction (ou) est vrai, la disjonction entière est vraie et le deuxième opérande n'est pas calculé non plus. Les opérateurs booléens sont principalement utilisés dans les instructions qui évaluent des conditions (alternative `if-else`, boucles `while`, `for` et autres).

Il y a un type nommé `_Bool` **en C99**. C'est un type entier non signé assez grand pour prendre les valeurs 0 et 1.

3.5 Caractères

Les caractères, de type `char`, forment un type assez spécial en C. Ils tiennent généralement sur un octet (ils pouvaient tenir sur 9 bit lorsque existaient encore des machines à mots de 36 bit et

tenaient sur 64 bit sur Cray). La norme permet qu'ils soient représentés par des `signed char` ou par des `unsigned char`. Cela explique que seuls les caractères dont le codage est entre 0 et 127 sont dénués d'ambiguïté, on s'abstiendra de citer d'autres caractères que ceux-là.

Une syntaxe permet de citer des caractères dans un programme. L'expression `'a'` vaut le petit entier codant le caractère « a minuscule ». Quelques caractères spéciaux existent, comme `\r` pour « retour chariot » (abrégé en CR), `\t` pour « tabulation » ou encore `\0` pour le caractère dit `NUL` de code zéro. On trouve encore `\n` dit caractère *newline*. Ce caractère abstrait est utilisé par Unix pour marquer une fin de ligne là où DOS utilise le couple CR-LF (*carriage return-line feed*) et là où MacOS utilise le couple LF-CR. Le caractère *newline* ne pouvant être utilisé abstrait, c'est *line feed* (LF) qui a été choisi pour le représenter en Unix.

La plupart des fonctions de lecture renvoyant des résultats de type caractère mais devant aussi signaler les fins de flux (ou de fichiers) renvoient en fait un `int`, le plus petit type qui englobe tous les `char` et qui inclut au moins une valeur n'étant pas un `char` pour identifier `EOF`, la macroconstante d'expansion signifiant la fin de flux (pour *end of file*).

Conclusion sur les types de base

C a un souci constant de coller aux unités centrales et de pouvoir exprimer des calculs avec les valeurs et les instructions précises de leur jeu d'instructions. Le souci d'écrire vite des programmes conduit à l'introduction de conversions explicites automatiques entre types, demandant une rigueur accrue et une connaissance fine des machines. Toutes ces caractéristiques font de C un langage difficile à maîtriser dans tous ses détails.

3.6 Enregistrements

Un enregistrement (ou *structure* en C) permet de rassembler, en un tout manipulable, une suite de champs (ou *membres* en jargon C) nommés. Un enregistrement est, avant emploi, défini par son type grâce au mot-clé `struct`. Voici, par exemple, le type `Point` d'un point à deux dimensions dont les coordonnées sont nommées `x` et `y`.

```
struct Point {
   double x;
   double y;
};
```

Les champs d'un enregistrement peuvent être extraits par l'usuelle notation pointée. Si `p1` est de type `struct Point`, alors `p1.x` est le `double` correspondant à la première coordonnée du point `p1`. L'expression `p1.x` peut être utilisée en lecture ou en écriture.

La représentation physique d'un enregistrement se déduit aisément de la définition de son type. Les champs d'un enregistrement sont placés dans l'ordre de définition et respectent les contraintes de taille et d'alignement de l'unité centrale sous-jacente. Des octets de bourrage (*padding*) sont introduits si nécessaire pour assurer des alignements corrects. Il est également possible d'imposer la taille en bits des champs d'un enregistrement ; cette option assez rare permet d'obtenir des placements bit à bit très précis.

Comme toutes les valeurs de C (à l'exception des tableaux), les enregistrements sont copiés lors des affectations, des appels à des fonctions ou des retours de fonctions. Comme les copies peuvent être coûteuses, l'usage de références est préférable. L'exemple qui suit est donc peu recommandable et n'illustre que ces mécanismes de recopie.

```
int
est_sur_diagonale (struct Point p)
{
   return p.x == p.y;
}
   ...
   /* initialisation des champs x et y */
   struct Point p1 = {1, 1};
   struct Point p2;
                          /* copie de p1 dans p */
   if ( est_sur_diagonale(p1) ) {
      /* copie de p1 dans p2 */
      p2 = p1;
   ...
```

L'appel à la fonction `est_sur_diagonale` recopie (les octets composant) le point `p1` dans (les octets formant) le point `p`, c'est-à-dire la variable `p` de la fonction invoquée. La ligne `p2 = p1` recopie (les octets composant) le point `p1` dans (les octets formant) le point `p2`. C'est pour éviter ces copies que l'on use généralement de références permettant de partager les valeurs.

3.7 Tableaux

Un tableau est une répétition finie d'une certaine donnée (une valeur de base, un enregistrement ou même un tableau). Les types tableaux ne sont pas déclarés, ils sont créés implicitement à la déclaration des tableaux. Les éléments de tableaux sont lus ou écrits avec la syntaxe courante pour l'indexation qui use de crochets carrés. La particularité qu'a introduite C est que les éléments de tableaux sont comptés à partir de zéro.

```
int u[10][5];
struct Point v[5] = {{0, 0}, {1, 1}, {2}};
```

Les fragments précédents définissent un tableau à deux dimensions d'entiers nommé `u`, ainsi qu'un tableau monodimensionnel (un vecteur) de points. Ce dernier est initialisé – partiellement puisque seules ses trois premières coordonnées sont (partiellement) spécifiées – dans sa déclaration. Avec ces déclarations, l'expression `v[0]` désigne le premier point du tableau `v`, tandis que `v[1].x` est la première coordonnée du second point du tableau `v` (initialisée à 1). Pour un tableau monodimensionnel `t` de taille `n`, les éléments sont nommés `t[0]`, `t[1]`, `...t[n-1]`.

Les tableaux peuvent également être multidimensionnels ; ils sont toutefois considérés comme des vecteurs de vecteurs. Ils sont physiquement rangés en mémoire de telle façon que le dernier indice soit celui qui varie le plus vite (c'est l'ordre inverse de Fortran). Ainsi, le tableau `u` commence par `u[0][0]`, suivi de `u[0][1]`, etc.

C est un langage sans bibliothèque d'exécution. En conséquence, un tableau ne saurait être inspecté à l'exécution afin de connaître sa taille. La taille d'un tableau n'est pas une information structurelle toujours présente et accessible. Ainsi, les programmes qui manipulent des tableaux dont ils connaissent par ailleurs la taille ne payent pas pour les programmes qui ont négligé de stocker cette information. En conséquence, puisque les tableaux n'ont pas de taille, il n'y a pas de vérification possible d'index : le compilateur ne peut assurer que les programmes n'accèdent qu'aux seuls éléments présents dans le tableau. Cette caractéristique est à la fois dangereuse puisqu'elle rend la mémoire toute entière accessible sans garde-fou (`v[-273]` permet de lire loin, bien loin, très loin de la zone `v`) et nécessaire car elle permet de manipuler des tableaux dont la taille dynamique est quelconque.

La syntaxe utilisée en C est une **syntaxe dite en situation** : la déclaration du type mime son emploi. Cette décision (remise en cause, par exemple, par Java) ne permet malheureusement pas de décrire simplement la syntaxe d'une déclaration comme un type suivi de la mention d'un nom de variable du genre `int i` ou `struct Point p`. Une variable est déclarée dans un contexte d'emploi tel que ce dernier a un type assez simple. Ainsi, puisque v est un tableau de `struct Point`, on a que `v[5]` est une `struct Point`, ce que l'on écrit `struct Point v[5]`. D'autres exemples plus probants apparaîtront plus loin (§ 3.12).

La représentation physique d'un tableau peut nécessiter des octets additionnels de bourrage afin que l'élément répété puisse respecter les contraintes d'alignement imposées par l'unité centrale.

À la différence de tous les autres types de données, lorsqu'un tableau apparaît comme argument d'une fonction, il est partagé entre l'appelée (la fonction) et l'appelant. Toute modification que viendrait à y apporter l'appelée sera donc visible de l'appelant lorsque l'appelée renverra son résultat. Passer un enregistrement en argument conduit à sa recopie (qui peut être une source d'inefficacité certaine), toute modification que viendrait à y apporter l'appelée sera donc locale à l'appelée et invisible de l'appelant lorsque l'appelée renverra son résultat.

Les expressions permettant d'accéder à un champ d'enregistrement ou à un élément de tableau peuvent être utilisées en lecture ou en écriture, c'est-à-dire à gauche ou à droite d'un signe d'affectation. Dans ces deux cas, le sens de l'expression n'est pas tout à fait le même. En fait, l'expression désigne une zone mémoire (un certain nombre d'octets situé à une certaine adresse). Cette zone mémoire est *écrite* quand l'expression est à gauche du signe d'affectation (on parle alors de « valeur gauche » ou, en jargon, de « L-valeur »). Cette zone mémoire est *lue* quand l'expression est à droite du signe d'affectation.

3.8 Références

C ne procure pas de types récursifs permettant de définir des listes ou des arbres mais propose la notion de référence. Une référence est un objet opaque désignant une zone en mémoire. Une zone est définie par un type (permettant de connaître sa taille en octets) et une adresse la situant en mémoire. L'opérateur & appliqué à une zone, fabrique une référence sur cette zone. L'opérateur * appliqué à une référence permet de retrouver la zone référencée. L'opérateur * est l'inverse (à gauche) de & : pour toute zone z, on a *&z égale à z. Ce mécanisme, assez simple, se complique du fait qu'une zone voit sa signification dépendre de sa position par rapport au signe d'affectation.

Comme les tableaux, les variables de type référence se définissent en situation. Une variable pp de type référence sur un point est définie comme `struct Point *pp` signifiant que *pp est une `struct Point`.

```
struct Point p1;
struct Point p2[3] = { {1, 1},{2, 2},{3, 3} };
struct Point *pp = &p2[1];

p1.x = 10 * (*pp).x;
      /* p1 contient {20, ?} maintenant. */
p1.y = 10 * pp->y;
      /* p1 contient {20, 20} maintenant. */
pp->x = 3;
      /* p2[1] contient {3, 2} maintenant. */
*pp = p1;
      /* p2[1] contient {20, 20} maintenant.*/
```

La zone nommée pp référence une zone contenant deux `doubles` qui correspondent au second des points contenus dans le tableau p2. L'écriture `(*pp).x` étant un peu lourde, on lui préfère souvent l'écriture rigoureusement équivalente `pp->x`. Ces deux écritures permettent de désigner la zone nommée x dans la zone référencée par pp (la zone référencée est aussi nommée `p2[1]`).

Les deux premières affectations utilisent la zone référencée à droite du signe d'affectation, la zone est donc lue. Les deux dernières affectations l'utilisent à gauche du signe d'affectation, les zones sont donc écrites.

La constante prédéfinie `NULL` est la référence qui ne référence rien. Il est erroné de déréférencer la référence `NULL` (en général, on obtient une erreur mais la norme ne l'impose pas). Comme `NULL` est représentée par zéro, toute référence différente de `NULL` équivaut au booléen vrai.

Une référence a la taille d'une adresse de l'unité centrale sous-jacente (quatre octets pour la plupart des machines d'aujourd'hui, huit octets pour DEC alpha). Cette constance de taille permet de définir des types récursifs avec insertion de références explicites. Ainsi, le type liste d'entiers pourrait être défini comme :

```
struct ListeEntier {
   struct ListeEntier *suite;
   int entier;
};
struct ListeEntier maillon2 = {
   NULL, 123 };
struct ListeEntier maillon[2] = {
   {&maillon[1], 1},
   {&maillon2,   -42} };
struct ListeEntier *le = &maillon[0];
```

La valeur de la variable `le` est une référence sur une liste de trois entiers. Cette liste est formée de trois « maillons » contenant respectivement 1, – 42 et 123. La fin de liste est représentée par le pointeur vide `NULL`.

La déréférenciation n'est pas la seule opération possible sur les références, il existe également toute une arithmétique. Prenons le cas de la variable `le` qui référence la zone `maillon [0]` ; écrire `le+1` bâtit une nouvelle référence sur la zone `maillon [1]`. Symétriquement, `(le+2)–1` bâtit une nouvelle référence toujours sur la zone `maillon[1]`. Ainsi, addition et soustraction permettent de construire de nouvelles références translatées d'un certain nombre de termes. Il est également possible de faire la différence entre deux références et d'obtenir ainsi le nombre de termes qui les séparent : fort algébriquement, `(le+1)–le` vaut 1.

Ces possibilités arithmétiques sont valables pour toute référence. Quoique `&maillon2` ne soit pas une référence dans un tableau, l'expression `1+&maillon2` désigne la donnée de type `struct ListeEntier` qui serait un cran après `maillon2`, comme si `maillon2` était le premier terme d'un tableau de `struct ListeEntier`. L'expression `&maillon+1000` est tout autant permise et permet de lire tout l'espace mémoire du programme.

Lorsqu'un tableau est passé en argument à une fonction, son type est « dégradé » en une référence sur son premier terme (n'étant plus un tableau, cette référence peut être copiée sans problème tout en assurant que la zone référencée est bien partagée). Ainsi peut-on écrire :

```
unsigned long
longueur_liste_entiers (struct ListeEntier *l)
{
      unsigned long longueur = 0;

      while ( l != NULL ) {
         longueur++;
         l = l->suite;
      }
      return longueur ;
}

   ...
   /* Ici, maillon est dégradé en &maillon[0] */
   printf("longueur_liste_entiers: %lu\n",
         longueur_liste_entiers(maillon));
   ...
```

Les références sont typées et le système de types de C n'est pas polymorphe, il n'est donc pas possible d'écrire des fonctions d'utilité générale, il faudrait en écrire une version par type concerné. C a introduit un nouveau type de référence : `void*`, qui correspond à la notion d'adresse en mémoire. Le type `void*` désigne l'essence d'une référence sur une zone dont on ne sait rien. On ne peut rien faire d'une telle référence, on ne peut même pas la déréférencer car il n'y a aucune valeur de type `void` ! On ne peut pas plus faire de l'arithmétique avec un `void*` car on ne connaît pas la taille de ce qui est référencé !

La seule opération permise est de convertir cette référence en une référence typée. Ainsi, on peut typer la fonction `malloc` (pour *memory allocation*, cf. § 7.1) qui prend un nombre d'octets, tente d'allouer dynamiquement quelque part en mémoire une zone de cette taille et renvoyer une référence sur cette zone. L'ignorance du type de référence qu'il faut renvoyer ainsi que l'unicité de la fonction `malloc` (et non une fonction `malloc` par type existant) font que `malloc` ne peut renvoyer qu'un `void*` que l'appelant convertira en le type de son choix (en espérant qu'il ne se soit point trompé sur la taille demandée !).

Une référence typée correspond à une adresse en mémoire et la connaissance du type de ce qui est référencé (type dont on peut notamment extraire la taille afin d'implanter l'arithmétique des références). Une référence `void*` ne correspond qu'à une adresse.

3.9 Chaînes de caractères

Bien qu'il existe une syntaxe pour citer des chaînes de caractères (enserrées entre guillemets), les chaînes n'existent en fait pas en C ! On les simule par des tableaux de caractères et par une référence sur le premier caractère de ce tableau, c'est-à-dire un élément de type `char*`.

Comme C est un langage sans bibliothèque d'exécution, la taille d'une chaîne n'est pas une information structurelle. La taille est déterminée par la convention qu'un caractère dit `NUL`, de code entier zéro, figure en dernière position de ce tableau. Cette convention interdit donc d'utiliser (sans précaution) le caractère `NUL` dans une chaîne.

```
char s1[] = "Bonjour";
char* s2 = &s1[3];
char* s3 = "jour";

printf("%s", s1);       /* imprime Bonjour */
*s2 = s1[strlen(s1)+1];
printf("%s", s1);       /* imprime Bon */
```

Dans l'exemple précédent, le tableau `s1` est initialisé avec les caractères formant le mot « Bonjour ». La taille du tableau est automatiquement ajustée à 8 puisqu'un `NUL` final est inséré. La seconde ligne définit un pointeur sur le « j » de « Bonjour ». De par leur construction, ces deux premières chaînes partagent les mêmes octets composant le mot « jour ». La troisième ligne alloue quelque part un tableau de caractères initialisés avec les lettres formant le mot « jour » puis initialise la variable `s3` avec un pointeur sur le « j » de ce mot. Suivant les options du compilateur, cette occurrence de « jour » sera égale à ou différente de la précédente.

Les chaînes de caractères sont manipulées à l'aide de fonctions spécialisées. Par exemple, la fonction `strlen` calcule le nombre de caractères présents dans une chaîne, c'est-à-dire le nombre de caractères non `NUL` partant du premier caractère pointé. Le fragment précédent copie le `NUL` achevant la chaîne `s1` au début de la chaîne `s2`, raccourcissant ainsi `s2` et par là même `s1`.

L'absence de chaînes de caractères n'est pas très gênante à un niveau d'assembleur. C'est en revanche plus délicat avec l'avènement du réseau et des interstitiels comme Corba ou DCOM. En effet, `char*` ne décrit qu'un pointeur sur un unique caractère et ne dit strictement rien de plus sur les éventuels caractères qui pourraient suivre. Il faut donc distinguer, dans le cadre de ces programmations de plus haut niveau, la chaîne et le pointeur sur un unique caractère.

3.10 Union de types

Le type `union` confère à C les vertus d'un **assembleur**. Dans un système d'exploitation, le matériel impose souvent de traiter des zones dont le type est inconnu mais dont les premiers octets permettront de découvrir le type précis (que l'on songe par exemple à un paquet venant d'Internet dont il faut découvrir la nature avant de savoir quoi en faire). Certaines interfaces matérielles peuvent également être très polymorphes et, par exemple, adopter un découpage en champs dont la nature est indiquée par les premiers bits. Dans ce monde à la lisière du matériel, les valeurs existent parfois avant que leur type ne soit déterminé : il est donc nécessaire de pouvoir conférer dynamiquement un type à une zone. Le type `union` permet cela.

Voici un exemple d'emploi définissant des flottants courts :

```
union ieee754_float {
   float f;

   /* Les flottants simple-précision (32 bit)
      d'IEEE 754. */
   struct {
#if   __BYTE_ORDER == __BIG_ENDIAN
      unsigned int negative:1;
      unsigned int exponent:8;
      unsigned int mantissa:23;
#endif                        /* gros-boutien. */
#if   __BYTE_ORDER == __LITTLE_ENDIAN
      unsigned int mantissa:23;
      unsigned int exponent:8;
      unsigned int negative:1;
#endif                        /* petit-boutien. */
   } ieee;
}

union ieee754_float f1 = { 3.14 };
```

Un flottant simple conforme à la norme IEEE 754 peut être décomposé en trois champs correspondant à son signe, son exposant et sa mantisse – dans un ordre dépendant de l'ordre suivant lequel les octets de la machine sont rangés (gros-boutien : les

octets de poids fort sont rangés d'abord, et c'est le cas de pratiquement toutes les machines sauf les PC, petit-boutien : les octets de poids faibles sont rangés en premier). On peut imposer un type particulier à la zone `f1` en écrivant `f1.f` ou `f1.ieee`. Avec cette dernière expression, on peut accéder à la mantisse en écrivant `f1.ieee.mantissa`.

Une zone de type `union` a pour taille la plus grande des tailles de ses variantes.

3.11 Types énumérés

Les types énumérés permettent de définir des constantes entières nommées. Ils apportent un peu plus de sûreté et de portée dans un monde de constantes de macroexpansion. Voici un court exemple montrant la définition de deux constantes nommées ayant pour valeur 0 (implicite) et 366 (explicite).

```
enum annee {
   NORMALE,
   BISSEXTILE = 366
}
   ...
   printf("NORMALE, BISSEXTILE: %d, %d\n",
          NORMALE, BISSEXTILE);
```

Conclusion sur les valeurs manipulées

Les types ont été initialement introduits en C pour une raison fondamentale : connaître à la compilation la taille des données manipulées afin de pouvoir engendrer les instructions spécialisées pour ces tailles et pour ces types. Ce n'est que bien plus tard que le typage a commencé d'avoir des vertus supplémentaires comme celle de pouvoir vérifier une certaine cohérence des programmes relativement aux données qu'ils manipulent. Cette nouvelle qualité importante ne doit pas faire oublier que partout, le compilateur doit connaître la taille des données manipulées : lors des affectations (puisqu'il y a recopie), lors des invocations à des fonctions, pour passer d'une référence à une zone, etc.

3.12 Typage

Les règles syntaxiques de C confèrent aux déclarations une syntaxe en situation. Hélas, dès que le type comporte plus d'un opérateur (référence sur un tableau, tableau sur des références), la syntaxe s'obscurcit et nécessite de connaître les règles régissant les priorités des opérateurs.

`int i;`	entier
`float tf[10];`	vecteur de 10 flottants
`char *p;`	référence sur un caractère
`struct Point *pp[10];`	tableau de 10 références sur des Points
`struct Point *(pp[10]);`	tableau de 10 références sur des Points
`struct Point (*pp)[10]);`	référence sur un tableau de 10 Points
`int *foo(int);`	fonction prenant un entier et renvoyant une référence sur un entier

La directive `typedef` existe à la fois pour nommer des types et pour cacher leur définition précise (lorsque l'on souhaite les rendre opaques à ses clients). Le nouveau type créé peut être utilisé partout où un type prédéfini peut l'être. Ainsi peut-on écrire :

```
typedef struct Point *Point;
Struct Point p1 = { 1, 2 };
Point pp1 = &p1;

typedef Point (*PointFonction)(Point);

Point
conjugue (Point p)
{
   p->y = - p->y;
   return p;
}

  ...
  PointFonction pf = conjugue;
  pp1 = pf(pp1);
  ...
```

Le type `Point` est défini comme une référence sur une structure Point. Le type `PointFonction`, quant à lui, est défini comme une fonction prenant un `Point` et retournant un `Point`. On peut dès lors utiliser ce type pour, par exemple, définir une variable `pf` que l'on initialise avec la fonction `conjugue`.

En C, l'équivalence de type est structurelle : deux types sont égaux s'ils ont même définition. La directive `typedef` est donc un procédé d'abréviation.

Deux mots-clés supplémentaires peuvent flanquer les déclarations de types, ils se nomment `const` et `volatile`. Le mot-clé `const` assure que la variable ou le champ ainsi qualifié est constant. Le compilateur vérifie alors qu'aucune valeur gauche de ce type n'existe. Le mot-clé `volatile` au contraire désigne une zone dont le contenu peut évoluer indépendamment du programme (une cellule mémoire contenant un compteur matériel par exemple). Chaque fois que cette zone est lue, elle doit être effectivement lue et ne peut être stockée en registre (car elle deviendrait vite obsolète).

4. Expressions

En termes d'expressions, C est un langage normal.

4.1 Identificateurs

Les identificateurs en C commencent par une lettre (ou un blanc souligné) et peuvent continuer avec des lettres, des chiffres ou des blancs soulignés. Seules les trente et une premières lettres sont discriminantes pour le compilateur qui différencie majuscules et minuscules. Il y a trente-deux mots-clés réservés que l'on ne peut utiliser comme identificateur (voir encadré **1**).

En C99, les mots-clés suivants ont été ajoutés : `inline`, `restrict`, `_Bool`, `_Complex` et `_Imaginary`.

Il est également déconseillé d'utiliser des identificateurs de fonctions ou de macros prédéfinies dans des bibliothèques standards, ainsi que des noms débutant par des blancs soulignés,

sans oublier les traditionnels `asm` et `fortran` utilisés pour les relations entre C et d'autres langages.

Encadré 1 – Mots-clés réservés

`auto`	`break`	`case`	`char`
`const`	`continue`	`default`	`do`
`double`	`else`	`enum`	`extern`
`float`	`for`	`goto`	`if`
`int`	`long`	`register`	`return`
`short`	`signed`	`sizeof`	`static`
`struct`	`switch`	`typedef`	`union`
`unsigned`	`void`	`volatile`	`while`

4.2 Littéraux

Les littéraux sont les constantes entières, les flottants, les caractères ainsi que les chaînes de caractères. Ce sont tous des expressions.

4.3 Opérateurs unaires

C comporte un certain nombre d'opérateurs prédéfinis ayant diverses arités. Les opérateurs unaires figurent dans l'encadré **2**.

Encadré 2 – Opérateurs unaires

`-`	opposé
`!`	négation booléenne
`~`	négation pour chaîne de bit
`&`	référence
`*`	déréférence
`(`*type*`)`	conversion

Ces opérateurs unaires préfixent l'expression sur laquelle ils opèrent. La conversion est la plus délicate à maîtriser : elle permet de changer la représentation de nombres (avec toutes les précautions mentionnées plus haut), comme dans `(float) 1` qui renvoie `1.0`. La conversion permet également d'adapter la signature des fonctions. Enfin, la conversion autorise aussi de changer le type des références, ce qui permet de « tricher » sur la nature de ce qui est référencé. Le fragment suivant illustre la conversion d'une référence vers une structure en une référence sur son premier champ (ce que l'on peut toujours faire). On peut également effectuer la conversion inverse.

```
struct Point {
   double x;
   double y;
} p1, *pp = &p1;
int offsetXY = 1;

   ...
   double *ppd = ((double*) pp)+offsetXY;
   double x1 = *ppd;
       /* x1 == p1.y == pp->y */
   double x2 = ((struct Point*) (ppd-offsetXY))->x ;
       /* x2 == p1.x == pp->x */
```

Dans ce fragment, une référence sur la zone `p1.y` est obtenue en convertissant tout d'abord la référence sur le point `p1` en une référence sur son premier champ `p1.x`. Une incrémentation permet de déplacer cette référence sur le `double` qui suit, `ppd` est donc une référence sur la zone `p1.y`. L'affectation à `x2` illustre comment retrouver la référence sur le point `p1` à partir de `ppd`. On décrémente tout d'abord `ppd` pour retrouver une référence sur `p1.x` que l'on convertit en une référence sur le point `p1` tout entier, référence dont on extrait le champ `x`. Toutes ces conversions sont décidées par le programmeur et crues par le compilateur qui ne s'assure que de la cohérence entre deux conversions mais certainement pas de la validité des conversions : le code précédent n'est valable que pour `offsetXY` valant 1.

4.4 Opérateurs binaires

Les opérateurs binaires sont légion. Ils apparaissent dans l'encadré **3**.

Encadré 3 – Opérateurs binaires

`+`	addition
`–`	soustraction
`*`	multiplication
`/`	division (flottante ou euclidienne)
`%`	reste euclidien
`== != < > <=>=`	comparateurs génériques
`||`	ou booléen (avec court-circuit)
`&&`	et booléen (avec court-circuit)
`&`	et pour chaînes de bit
`|`	ou pour chaînes de bit
`^`	ou exclusif pour chaînes de bit
`<< >>`	décalage pour chaîne de bit

Les opérateurs de comparaison sont génériques, ils peuvent s'appliquer sur tout type ayant une base numérique : les nombres, les caractères mais aussi les références.

Les opérateurs booléens binaires sont à court-circuit, ils n'évaluent leur second argument qu'en cas d'indétermination du résultat final.

4.5 Autres opérateurs

Il existe un **opérateur ternaire** correspondant à l'alternative classique (si-alors-sinon) sous forme d'expression et non d'instruction. Cet opérateur permet d'augmenter la compacité du code engendré lorsque des expressions complexes ne diffèrent que de peu. C'est ce que montre le fragment suivant où la première ligne est explicitée par les cinq dernières :

```
printf("Il y a %d mot%s\n", n, (n>1)?"s":"");
   /* à la place de : */
if (n > 1) {
   printf("Il y a %d mots\n");
}  else {
   printf("Il y a %d mot\n");
}
```

Enfin, il existe un dernier **opérateur binaire de séquentialisation** : la virgule. Elle est principalement utilisée dans les boucles `for` pour introduire plusieurs variables de boucle. L'expression composée e_1, e_2 calcule d'abord e_1 puis renvoie la valeur de e_2.

4.6 Opérateurs sur zones

D'autres opérateurs binaires restreignent leur premier argument (celui de gauche) à une zone écrivable : ces opérateurs sont des variations autour de l'affectation notée par le signe = (encadré **4**).

Encadré 4 – Affectations

=	affectation
op=	opération binaire (*op*) puis affectation

L'affectation est, en C, une expression et non une instruction. Puisque c'est une expression, elle retourne une valeur qui est celle rangée dans la zone de gauche.

La syntaxe *op*= correspond à une élimination de sous-expression commune (que ne faisait pas le compilateur C originel). Écrire `x += 3` correspond en fait à écrire `x = x + 3`. C'est aussi vrai avec tous les autres opérateurs binaires. Ainsi, l'amusante graphie `z << = 3` est équivalente à `z = z << 3`. Attention cependant aux opérateurs non commutatifs : `y %= 5` est équivalent à `y = y % 5` et non à `y = 5 % y`.

On ne peut utiliser cette notation avec les opérateurs binaires booléens car ils sont à court-circuit.

Les compilateurs actuels sont bien meilleurs et n'ont plus besoin de cette notation. Elle s'est toutefois « médullarisée dans les parties reptiliennes des cerveaux des programmeurs C », ce qui fait que l'on s'en sert encore (ainsi qu'en Perl, Java et autres langages ayant le même substrat syntaxique). Il est donc hors de question de faire disparaître ces syntaxes si compactes et si utiles.

Deux autres variations autour de l'affectation existent encore. Ce sont les opérateurs unaires préfixes ou suffixes de l'encadré **5**.

Encadré 5 – Incrémentation/décrémentation

++	incrémentation et affectation
--	décrémentation et affectation

Ces opérateurs ont été inventés pour mettre en œuvre les facultés de pré/post-incrémentation/décrémentation que procuraient les unités centrales d'antan. L'opérateur ++ (resp. `--`) s'applique à une zone et l'incrémente (resp. la décrémente). Ainsi, l'expression `++x` est équivalente à `x += 1` où la variable `x` est incrémentée et sa nouvelle valeur est renvoyée. Écrire `x++` renvoie la valeur courante de `x` et garantit que la variable `x` sera incrémentée avant la fin de l'instruction courante. La norme introduit une notion de point de contrôle permettant de découvrir jusqu'à quand le compilateur peut au plus différer cette incrémentation.

Ces opérateurs peuvent s'appliquer à tous les types à substrat numérique ainsi qu'aux références. On peut ainsi comprendre comment l'on peut définir la fonction `strcpy` qui copie une chaîne de caractères dans une autre, selon un style éminemment propre à C.

```
char*
strcpy (char *dest, const char *src)
{
   char *d = dest;

   while (*src) {
        *d++ = *src++;
   }
   return dest;
}
```

La boucle présente dans cette fonction recopie les caractères un par un tant qu'ils ne sont pas `NUL`.

4.7 Priorités et associativités syntaxiques

Il existe seize niveaux de priorité syntaxique pour les très nombreux opérateurs de C qui, de plus, peuvent être associatifs à gauche ou à droite. En revanche, parenthéser permet de forcer l'interprétation souhaitée sans avoir à connaître et à se remémorer ces détails.

Nota : pour plus de détails, le lecteur est invité à se reporter à la bibliographie, et notamment à [2].

5. Instructions

C est un langage à instructions. Toute expression suivie d'un point-virgule est une instruction. Un point-virgule marque la fin de chaque instruction ; le point-virgule n'est pas un séparateur d'instructions comme en Pascal, c'est en C un terminateur d'instruction. Quelques instructions prédéfinies permettent de structurer des instructions en alternatives et boucles. Toujours fidèle au principe de rapidité d'écriture, C introduit de nombreux raccourcis d'écriture.

5.1 Bloc local

```
{
   definitionLocale*
   instruction*
}
```

Le bloc d'instructions regroupe en une seule instruction une série d'instructions éventuellement précédée d'une série de définitions de variables locales au bloc. Ces variables sont allouées en pile au début du bloc et sont automatiquement dépilées en sortie de bloc.

5.2 Alternative

```
if ( expression ) {
   instruction*
}  else {
   instruction*
}
```

L'alternative, c'est-à-dire l'instruction « si-alors-sinon », est très classique. On peut omettre l'alternant, c'est-à-dire la partie « sinon » qui débute avec le mot-clé `else`. Dans la syntaxe ci-avant, la conséquence et l'alternant sont représentés par des blocs d'instructions mais pourraient être réduits à n'importe quelle instruction unique.

5.3 Boucle `while`

```
while ( expression ) {
   instruction*
   /* break et continue possibles ici */
}
```

La boucle « tant-que » classique évalue sa condition (l'expression entre parenthèses suivant le mot-clé `while`). Puis, si cette condition est vraie, elle exécute son corps, enfin, elle recommence tout depuis le début. Là encore, le corps de la boucle pourrait être réduit à n'importe quelle instruction unique. Deux instructions spéciales sont possibles pour sortir immédiatement de la boucle (`break`) ou pour réitérer immédiatement la boucle (`continue`).

5.4 Boucle `do-while`

```
do {
   instruction*
   /* break et continue possibles ici */
} while (expression)
```

Cette boucle est une variation autour de la boucle tant-que. La boucle `do-while` exécute son corps puis teste la condition. Si la condition est vraie, alors la boucle toute entière est recommencée depuis le début. Les différences avec la boucle `while` sont que le corps est exécuté au moins une fois et que la condition de la boucle est testée après le corps (et non avant).

5.5 Boucle `for`

```
for (expression ; expression ; expression) {
   instruction*
   /* break et continue possibles ici */
}
```

La boucle `for` est encore une variation autour de la boucle `while`. Toutefois, comme elle met en jeu classiquement un index entre deux bornes, elle est plutôt ciblée vers les parcours de tableaux. Donnons un exemple d'une telle boucle.

```
int t[TAILLE];
int i, j;

for ( i=0, j=TAILLE-1 ; i<j ; i++, j-- ) {
   if ( t[i] > t[j] ) {
      int tmp = t[i];

      t[i] = t[j];
      t[j] = tmp;
   };
}
```

Ce fragment parcourt un tableau d'entiers nommé `t` de taille `TAILLE`. Deux variables sont utilisées : `i` et `j`, qui sont incrémentées (resp. décrémentées) jusqu'à se croiser. On peut noter l'usage de la virgule pour séparer ces effets.

Trois expressions suivent, entre parenthèses, le mot-clé `for` : une expression d'initialisation des index, une condition et une expression modifiant les index. En premier, l'expression d'initialisation est évaluée puis débute le traitement en boucle qui commence, comme la boucle `while`, par le test de la condition. Si la condition est vraie, alors le corps de la boucle est évalué, suivi de l'expression modifiant les index. Le traitement de la boucle recommence alors. Les instructions `break` et `continue` sont bien sûr possibles.

5.6 Saut

C est un assembleur et se doit de procurer les instructions natives de l'unité centrale sous-jacente. Il est donc nécessaire que le tant décrié `goto` soit présent. Toute instruction peut être précédée d'un label l'identifiant (un simple identificateur). C étant assez laxiste, il est possible de sauter à n'importe quelle étiquette de la fonction en cours dans l'irrespect le plus complet de la structure en blocs.

```
label: instruction
goto label
```

L'instruction `goto` est également bien pratique pour pallier l'insuffisance de `break` et `continue` qui ne permettent de contrôler que la boucle courante, sans possibilité de contrôle des boucles plus englobantes. Ce défaut est corrigé en Java.

5.7 Saut indexé

Une autre instruction importante de C est le saut indexé. L'instruction `switch` évalue l'expression (entre parenthèses) qui suit le mot-clé `switch`, ce qui mène à un entier (ou à un type à substrat entier comme les caractères). Cet entier permet de choisir à quel cas sauter parmi tous les cas mentionnés dans le corps du `switch` délimité par des accolades. Les cas sont explicités par le mot-clé `case` suivi d'une constante entière puis d'instructions. Un cas par défaut peut être également spécifié avec le mot-clé `default`.

```
switch (expression) {
   case constanteEntiere :
      instruction*
      /* break possible ici */
   ...
   default:
      instruction*
      /* break possible ici */
}
```

Attention, l'instruction `switch` n'est pas une généralisation n-aire de l'alternative `if`, c'est un saut indexé. Le compteur ordinal de l'unité centrale sous-jacente est modifié pour désigner l'instruction du cas visé. À partir de celle-ci, toutes les instructions qui suivent sont exécutées en séquence jusqu'à la fin de l'instruction `switch`. Comme ce n'est pas ce que l'on souhaite dans l'immense majorité des cas, on achève quasiment toujours le bloc d'instructions d'un cas par l'instruction `break`, qui signifie ici « sortir » de l'instruction `switch`. Voici un exemple :

```
int
compter_lignes (char *s)
{
   int nl_number = 0;

   while ( 1 ) {
      switch (*s) {
         case '\n' :
            nl_number++;
         default:
            s++;
            break;
         case NUL:
            return nl_number;
      }
   }
}
```

Cette fonction compte le nombre de caractères *newline* (c'est-à-dire le nombre de lignes) présents dans une chaîne de caractères. Lorsque le caractère référencé par `s` est `NUL`, la chaîne est terminée et la fonction renvoie le nombre de lignes comptées. Lorsque le caractère référencé par `s` est le caractère `\n`, le compteur `nl_number` est incrémenté ainsi que le pointeur `s` (cette instruction étant située dans le cas par défaut). Lorsque le caractère référencé est autre, seul le pointeur `s` est incrémenté. L'instruction `break` permet qu'après l'incrémentation de `s`, la boucle soit reprise en esquivant le cas de `NUL`.

Cette instruction est souvent décriée car il faut pratiquement toujours insérer un `break` après chaque cas. De plus, l'oubli d'un seul `break` est catastrophique. En revanche, modifier l'instruction `switch` pour insérer automatiquement ces instructions `break` ferait perdre définitivement et sans espoir de retour l'accès à cette instruction fondamentale en ce qui concerne l'efficacité d'exécution, qu'est le saut indexé.

6. Fonctions

Les instructions sont rassemblées en fonctions. Les fonctions factorisent des traitements et luttent ainsi contre la redondance. Les fonctions sont nommées et typées. Une fonction spécifie le type des arguments qu'elle attend et le type du résultat qu'elle renvoie. Un type spécial existe, nommé `void` (aucune valeur de ce type n'existe), pour signifier qu'une fonction ne renvoie aucun résultat (ce que l'on nomme une procédure en Pascal).

Une nouvelle instruction spéciale, nommée `return`, permet non seulement d'imposer le résultat d'une fonction mais aussi d'interrompre l'invocation de ladite fonction pour lui faire renvoyer immédiatement ce dit résultat. La fonction `compter_lignes`, précédemment donnée en exemple, illustre l'emploi d'un `return` au sein d'une boucle sans fin.

Le corps d'une fonction est un bloc pouvant, comme tout bloc, comporter des variables locales à ce bloc. Les variables (on dit aussi les paramètres) d'une fonction ont pour portée le corps de la fonction. Variables de fonctions et variables locales de bloc ont une durée de vie réduite à leur corps ou bloc. Les variables locales de bloc sont dites *automatic* en jargon et sont allouées dans la pile d'exécution. Elles sont désallouées lorsque l'on sort du bloc. Allocation et désallocation en pile sont particulièrement économiques : une seule instruction suffit à déplacer le pointeur de sommet de pile, quel que soit le nombre d'octets considérés. C'est plus cher qu'une allocation statique effectuée par le compilateur (zéro instruction à l'exécution) mais extraordinairement moins cher qu'un appel à l'allocateur dynamique en mémoire, la fonction `malloc`.

```
int
alloue_1K_et_invoque (int (*f) (void* zone))
{
   char zone[1024];

   return f((void*) &zone);
}
```

La fonction `alloue_1K_et_invoque` prend une fonction en argument sous le nom `f`, alloue un kilooctet en pile puis invoque `f` en lui fournissant cette zone allouée. Lorsque la fonction `f` renvoie un résultat, ce résultat devient celui de la fonction `alloue_1K_et_invoque` et le kilooctet est désalloué au passage. Si l'allocation en pile est peu coûteuse, elle nécessite de bien comprendre les relations qui existent entre les trois zones où peuvent résider les données :

— la zone statique allouée à la compilation pour les variables globales. Elle est présente du début jusqu'à la fin du programme. Elle est inextensible mais gratuite ;

— la zone dynamique, ou tas, allouée à la demande grâce à la fonction `malloc` (libérée grâce à la fonction `free`, à moins d'utiliser un glaneur de cellules, GC). Elle est extensible (pour autant que le système d'exploitation le permette), chère mais procure une durée de vie non contrainte ;

— la pile fluctuant au gré des invocations de fonctions. Elle est peu chère, automatiquement désallouée et, de ce fait, a une durée de vie bornée.

Ces trois cas ont été ordonnés afin d'illustrer la règle d'usage qui veut, pour simplifier la gestion des données, qu'une donnée en pile ne puisse pointer que sur des données plus anciennes en pile ou sur des données allouées en tas ou en zone statique, que les données en tas ne puissent pointer que sur les données en tas ou en zone statique. Tout écart à la règle doit faire l'objet de soins attentifs de façon à éviter les « pointeurs fous » c'est-à-dire des références sur des données ayant disparu (de la pile ou du tas). Cependant, il ne faut pas verser dans l'excès inverse : la rétention en mémoire, qui consiste à ne rien libérer. La programmation moderne fait usage d'une abondance d'objets qu'il importe de libérer si l'on veut que l'application puisse fonctionner à long terme (pendant des semaines ou des mois).

6.1 Phases de compilation

La compilation de C passe par plusieurs phases. Un programme écrit en C est généralement découpé en plusieurs fichiers dont le nom a *.c* comme suffixe. Le fichier est l'unité de compilation et conduit à un fichier compilé de suffixe *.o* (sous Unix) ou *.obj* (sous Windows). Les fichiers compilés peuvent être regroupés en bibliothèques – de suffixe *.a* (statiques) ou *.so* (dynamiques et partagées) pour Unix ou *.dll* pour Windows.

Pour éviter de devoir tout recompiler, un programme est souvent divisé en de multiples fichiers. Seuls les fichiers modifiés sont recompilés, ainsi que les fichiers qui en dépendent (par exemple, les bibliothèques les regroupant) avec des outils classiques mais toujours d'actualité, comme l'utilitaire *make*. La génération du programme exécutable final s'obtient au terme d'un processus que l'on appelle édition de liens et qui consiste à assembler les fichiers compilés nommément spécifiés, ainsi que les seuls fichiers compilés nécessaires extraits des bibliothèques nommément spécifiées. L'édition de liens est un processus bien plus rapide que la compilation, ce qui explique ce schéma de compilation/édition de liens qu'adoptent pratiquement tous les langages de programmation compilés.

Pour être rapide, l'édition de liens propre à C reste élémentaire et ne procure qu'un procédé dit de « compilation indépendante » plutôt qu'un procédé de « compilation séparée ». Un procédé de compilation séparée permet qu'un programme entier puisse être compilé d'un seul coup ou par petits morceaux avec exactement le même niveau de vérification. La compilation indépendante n'assure pas ce niveau de vérification qui doit être obtenu à l'aide de fichiers de suffixe *.h* contenant les déclarations de types communes aux fichiers devant former ensemble un exécutable cohérent.

6.2 Exportations

Un fichier compilé partiel est formé de code et de données compilés ainsi que de deux tables : l'une recensant les noms définis et l'endroit où ils le sont, l'autre recensant les noms inconnus et les endroits où ils manquent. Ce modèle assume qu'un unique espace de noms existe pour toutes les entités nommées. Cet unique espace exclut donc que deux entités différentes mais de même nom puissent y coexister.

Les noms exportés sont contrôlés au niveau de C par deux directives : `static` et `extern`. Un fichier apparaît comme une suite de déclarations ou de définitions de variables globales ou de fonctions. Lorsqu'un nom est défini, il est disponible pour le reste du fichier. Par défaut, tout nom est exporté. On peut ne pas exporter un nom en le qualifiant de `static`. Dans le cas d'une fonction que l'on souhaiterait ne pas exporter mais qui serait en récursion mutuelle avec une fonction à exporter, se pose le problème de l'ordre dans lequel définir ces fonctions. C distingue les déclarations (qui permettent d'associer un type à un nom) des définitions (qui permettent d'allouer la zone correspondant au nom possiblement avec une initialisation). Séparer la déclaration de la définition de la fonction permet de résoudre ce problème, comme l'illustre le fragment suivant :

```
#include "even.h"

/* Déclaration (ou, en jargon, prototype) */
static int odd (int n);

/* Définition et déclaration */
int
even (int n)
{
   if ( n == 0 ) {
      return 1;
   }  else {
      return odd(n-1);
   }
}

/* Définition */
/* static */ int
odd (int n)
{
   if ( n == 0 ) {
      return 0;
   }  else {
      return even(n-1);
   }
}
```

La fonction even est exportée, tandis que la fonction odd ne l'est pas. Ces deux fonctions ont besoin l'une de l'autre, aussi commence-t-on par la déclaration du type de odd (sans définition). On peut alors définir correctement even puis définir odd conformément à sa déclaration.

6.3 Variables rémanentes

La directive `static` peut également s'utiliser au sein d'un bloc pour qualifier une variable globale non exportée dont la portée est réduite à ce bloc. Ce type de variable correspond également à la notion de variable rémanente de certains autres langages de programmation puisque la variable est propre au bloc et retrouve, à tout nouvel appel, la valeur qu'elle avait à la fin du dernier appel. C'est ce que montre le fragment suivant qui assure que la fonction `initialise` n'est appelée qu'au plus une fois.

```
void
initialise (int s)
{
   static unsigned char est_deja_initialise = 0;

   if (est_deja_initialise) {
      return;
   };
   est_deja_initialise = 1;
   ...
}
```

Il faut toutefois noter que l'existence de variables globales modifiables nuit au caractère multitâche des programmes.

6.4 Directive `extern`

La directive `extern` permet d'expliciter les exportations. La bonne façon de procéder est la suivante. Lorsque l'on désire exporter une fonction ou une variable hors de son fichier de définition *`fichier.c`*, il faut associer à ce fichier un fichier d'export *`fichier.h`* contenant une directive `extern` déclarant cette fonction ou variable. Ainsi, exporter la fonction even précédemment définie passe par la création du fichier *`even.h`* contenant :

```
extern int even (int n);
```

Ce fichier doit être inclus à la fois par le fichier qui définit la fonction even (afin de vérifier que déclaration et définition sont cohérentes) et par les fichiers clients qui souhaitent utiliser la fonction even (afin de vérifier que l'usage qu'ils font de cette fonction est conforme à son type).

La directive `extern` doit être entendue comme une pure déclaration indiquant que, quelque part dans l'exécutable complet, existera une variable ou une fonction de ce type ainsi nommée.

6.5 Fonction `main`

Une fonction particulière nommée `main` sert à définir le point d'entrée d'un exécutable : l'exécution commence avec l'invocation de la fonction `main`. Celle-ci reçoit un tableau de chaînes de caractères correspondant aux arguments d'appel fournis par le système (*via*, par exemple, la ligne de commande). Comme les tableaux n'ont pas de taille en C, la fonction `main` reçoit un premier argument (traditionnellement nommé `argc`, *argument count*) indiquant le nombre de chaînes fournies dans le tableau de second argument.

```
int
main (int argc,
      char* argv[])
{
   int i;
   for (i=1 ; i<argc ; i++) {
      printf("%s", argv[i]);
      putchar(' ');
   }
   return EXIT_SUCCESS;
}
```

Cette fonction mime l'utilitaire *echo* d'Unix qui transmet ses arguments dans son flux de sortie standard.

La fonction `main` doit renvoyer au système d'exploitation un octet indiquant si le programme s'est heureusement terminé (la constante zéro nommée `EXIT_SUCCESS`) ou un octet différent de zéro pour expliquer les 255 façons de se « planter » !

7. Bibliothèques

C étant un assembleur portable, il n'impose pas de bibliothèque d'exécution. Il permet ainsi de réaliser de tout petits exécutables n'ayant que peu de dépendances vis-à-vis d'un système d'exploitation. C est ainsi apte à engendrer des programmes pour processeurs enfouis (machines à laver, contrôle de moteur, etc.).

C vient toutefois avec une « bibliothèque standard » procurant un grand nombre de fonctions usuelles. On y trouve notamment :

— *ctype.h* manipulation de caractères (`toupper` ...) ;
— *errno.h* manipulation des erreurs (`perror` ...) ;
— *assert.h* pour la mise au point ;
— *locale.h* pour particulariser la langue et certains affichages ;
— *math.h* pour les fonctions mathématiques ;
— *setjmp.h* échappements non locaux ;
— *signal.h* signaux ;
— *stdarg.h* fonctions d'arité variable ;
— *stdio.h* entrée/sorties, flux, fichiers ;
— *stdlib.h* divers ;
— *string.h* manipulation de chaînes (`strcat` ...) ;
— *time.h* manipulation de date et temps.

Ces bibliothèques sont standards et procurent les utilitaires classiques de manipulation de caractères, de chaînes, d'impression, de lecture, etc. Seules quelques-unes d'entre elles ne se retrouvent pas dans les autres langages et méritent quelques mots. Nous présenterons donc la fonction d'allocation dynamique `malloc` (§ 7.1) ainsi que les échappements non locaux : le couple `setjmp`/`longjmp` (§ 7.2).

7.1 Allocation dynamique

La fonction d'allocation dynamique `malloc` prend une taille en entrée, alloue (si possible) une zone de cette taille en mémoire et retourne une référence de type `void*` sur cette zone. Cette référence est alors convertie en une référence correctement typée sur cette même zone. La zone que renvoie `malloc` peut être utilisée pour n'importe quel type et répond donc aux plus exigeantes des contraintes d'alignement.

La mémoire ainsi acquise peut être rendue disponible au moyen de la fonction `free`. Il faut bien sûr veiller à ne pas libérer des zones encore utiles (et ainsi créer des pointeurs fous) mais à libérer les zones inutiles (sous peine de rétention indue). Des glaneurs de cellules (GC) existent bien sûr en bibliothèque pour C.

7.2 Échappements non locaux

La notion d'exception si importante dans les langages modernes n'existe pas en C car elle impose un surcoût d'implantation dans la bibliothèque d'exécution. Il existe en revanche un mécanisme de bas niveau qui peut être mis en œuvre pour réaliser un effet un peu semblable.

La fonction `setjmp` permet de marquer un contexte d'exécution sain auquel on pourrait souhaiter revenir en cas de problème. La fonction `setjmp` sauvegarde les registres de la machine, la hauteur de la pile d'exécution et quelques autres informations caractérisant l'état courant dans un *jump buffer*. Cette sauvegarde faite, la fonction `setjmp` renvoie alors la valeur zéro. On écrit souvent un fragment de C ressemblant à celui-ci :

```
jmp_buf jb;
int status;

if ( (status = setjmp(jb)) == 0 ) {
   /* suite normale effectuant le calcul
      potentiellement suspect. Celui-ci s'échappe
      en exécutant (ici ou ailleurs):          */
               longjmp(jb, code);
} else {
   /* retour avec status <- code,
      traitement du retour inopiné */
}
```

Si d'aventure, pendant le traitement de la conséquence de l'alternative, un appel à la « fonction » `longjmp` est effectué, celle-ci abandonnera son contexte d'appel pour revenir au contexte identifié par `jb`. Le second argument de `longjmp` deviendra la nouvelle valeur qui sera de nouveau affectée à la variable `status`. En quelque sorte, la fonction `setjmp` ramène un second résultat qui, par spécification de la fonction `longjmp`, ne saurait être nul et conduira donc à l'exécution de l'alternant de l'alternative qui pourra traiter et réparer le retour inopiné provoqué par `longjmp`.

On ne peut utiliser la fonction `longjmp` que pendant la durée du calcul de la conséquence de l'alternative, c'est-à-dire, en termes plus techniques, que pendant la durée où la pile d'exécution marquée est encore en mémoire. Attention, un échappement brutal doit s'accompagner d'un effort de programmation important afin de ne pas conduire à une situation confuse où des ressources ne seraient pas libérées (fichiers non fermés, zones de données non désallouées, etc.).

8. Conclusion

C est un langage de programmation de haut niveau ayant des caractéristiques d'assembleur portable. Son existence sur tout ordinateur ainsi que l'identification d'un noyau portable longuement éprouvé en font un **langage majeur de développement**.

En revanche, c'est aussi un langage ne cherchant pas à cacher l'unité centrale sous-jacente. Les références, les mécanismes d'allocation et la multiplicité de types sont de très importants concepts délicats à dominer. Les programmeurs y trouvent à la fois une très grande précision et un certain confort.

C est le langage de choix pour les **systèmes enfouis**, pour l'écriture de programmes **proches de la machine**, pour des programmes dont la **portabilité** doit être assurée ou pour **coupler des langages différents**. Pour toutes ces raisons, C restera utilisé pendant encore quelques décennies.