CHAPITRE I

PRÉSENTATION

1 Introduction

1.1 Histoire du C

Création

Le langage C a été créé au début des années 1970 par Dennis Ritchie et Ken Thompson alors qu'ils travaillaient au développement du système d'exploitation Unix. Son nom provient de sa parenté avec le langage B, lui-même descendant du langage BCPL. La première description des règles et de la syntaxe du langage a été faite par Brian Kernighan et Denis Ritchie en 1978 lors de la publication de leur livre *The C Programming Language* [6] ce qui a contribué à assurer son succès. Ce livre (familièrement appelé le K&R), sans être une norme stricte, a servi de spécification informelle aux programmeurs de l'époque et cette version du C, maintenant dépréciée, est appelée C K&R.

Un certain nombre de constructions du langage se sont avérées délicates à utiliser et sources de bogues : type entier implicite, pas de déclaration préalable de fonction, absence de vérification des paramètres... Ces constations ont progressivement amené les fabricants de compilateurs C à implémenter leurs propres extensions au langage. La popularité grandissante du C, associée à la diversité des extensions et des implémentations, a conduit à une standardisation internationale : règles de syntaxe, comportement du code généré, bibliothèque standard...

Normalisation

Dès 1983, l'institut national américain de normalisation (ANSI) a formé un comité afin d'arriver à une spécification standard du langage, basée sur les implémentations Unix existantes. Ce travail a abouti en 1989 avec la publication de la norme C ANSI, aussi appelée C89. Il s'agit d'un sur-ensemble du C K&R incluant les principales extensions et clarifiant les règles d'exécution. Afin de maintenir une compatibilité avec tous les anciens codes source, la syntaxe K&R a été conservée comme « possibilité » mais déconseillée. En 1990, la norme C ANSI a été adoptée par l'organisation internationale de standardisation (ISO),

avec de simples modifications de forme, sous le nom de C ISO ou C90. Les termes C ANSI, C standard, C89 ou C90 font donc référence au même langage et aux mêmes règles, respectées maintenant par tous les compilateurs.

Un premier changement est intervenu en 1995 avec la publication d'un amendement ajoutant des détails mineurs : légères corrections, fichiers d'entête supplémentaires... On parle alors de la norme « C90 amendement 1 ».

Une deuxième normalisation par l'ISO, d'ampleur plus importante, a eu lieu en 1999. Elle a ajouté de nombreuses fonctionnalités au langage (types complexes et booléens, types entiers longs étendus, caractères étendus, fonctions *inline*, tableaux à taille variable, nouveaux fichiers d'entête et nouvelles fonctions de bibliothèque...), changé certaines règles de syntaxe (déclarations de variables incluses dans le code, syntaxe des commentaires...) et amélioré la compatibilité avec les normes de calcul flottant et les possibilités d'optimisation du code par le compilateur. La compatibilité avec la norme C89/90 est quasi-totale : seules certaines libertés syntaxiques ont été bridées afin de permettre au compilateur de signaler plus facilement des écritures risquées, au prix d'une correction et recompilation de vieux codes source. Cette version du langage est connue sous le nom de C99. Son adoption a été freinée par le manque d'empressement des vendeurs à proposer des compilateurs respectant cette norme. Plus de dix ans après, certains compilateurs n'intègrent toujours pas tous les aspects du C99, ce qui pose des soucis de portabilité du code source.

La dernière version du langage (la version « standard » actuelle) a été publiée en 2011 par l'ISO sous le nom de C11. Elle ajoute des mots-clés permettant de gérer l'alignement des variables, un support des codages Unicode, de nouvelles fonctions d'entrées/sorties limitant les risques de débordement de buffer et un type générique de fonction permettant, avec un même appel, d'avoir des arguments de types différents. Elle intègre un support standard pour la programmation multithreads auparavant laissée à des bibliothèques externes.

Certains ajouts du C99, rarement implémentés par les compilateurs, ont été rendus optionnels en C11 afin de favoriser son adoption. Pour la même raison, les compilateurs peuvent proposer une implémentation partielle de la norme C11, la disponibilité des fonctionnalités pouvant être testée dans le code. Ce livre se focalisera sur la norme C89/90, encore très répandue et dont les caractéristiques sont à la base du langage C. Les différences avec la norme C99 seront systématiquement présentées et un chapitre en fin d'ouvrage détaillera plus précisément les ajouts liés aux deux nouvelles normes C99 et C11.

1.2 Présentation du langage

Le succès du langage C ne s'est jamais démenti et de nombreux logiciels sont encore écrits en C afin de profiter de ses caractéristiques : simplicité, efficacité et proximité avec la machine. On peut citer le noyau Linux, l'environnement de bureau Gnome pour les systèmes Unix, les interpréteurs PHP et perl, l'interpréteur de bytecode CPython, les logiciels de calcul numérique Mathematica et MATLAB...

Avantages

- Le C est un langage simple, ayant peu de constructions syntaxiques, et il est assez facile de maîtriser rapidement l'ensemble de ses mécanismes. Il est cependant puissant, permettant d'exprimer des structures algorithmiques et des structures de données complexes, que cela soit dans le domaine de la programmation système, le calcul numérique ou la gestion de fichiers.

- C'est un langage intégrant des concepts algorithmiques d'un niveau supérieur au simple langage machine (boucles, tests, répétitions, fonctions...), tout en permettant la manipulation d'entités de bas niveau directement liées à l'architecture sous-jacente de la machine (bits, octets, adresses mémoire...).
- Sous réserve de respecter une certaine rigueur de programmation, la portabilité des programmes écrits en C est assez bonne, les points critiques pouvant se résumer aux interactions directes avec le matériel.
- Les instructions C et ses types de données étant assez proches des instructions processeur, le code généré par les compilateurs est très efficace, ce qui en fait un langage de choix pour les applications critiques.
- De par leur origine commune, les systèmes Unix sont fortement interfacés avec le C et cela offre aux logiciels utilisant ce langage un lien direct avec le système d'exploitation et l'architecture de la machine, favorisant l'efficacité et la rapidité du code machine généré.

Indépendamment des avantages du langage, trois autres raisons font du C un apprentissage indispensable à tout programmeur :

- la manipulation des structures de données se faisant directement à partir de leurs rouages internes, la maîtrise du langage nécessitera une bonne compréhension des concepts matériels (adresses, pointeurs, taille des données...) ;
- de nombreux langages se sont inspirés de la syntaxe du C et l'apprentissage de cette dernière sera un investissement pour la maîtrise d'autres langages de programmation ;
- l'ancienneté du C et l'hégémonie qu'il a pu avoir à une époque ont produit un corpus de code absolument gigantesque et il est probable que tout informaticien aura un jour ou l'autre à maintenir et à faire évoluer un logiciel écrit en C.

Inconvénients

Malheureusement toute médaille a son revers et le C n'échappe pas aux critiques.

- La permissivité de la syntaxe[1] autorise à écrire du code efficace mais qui risque d'être totalement illisible, empêchant toute évolution ultérieure du code. Il existe d'ailleurs un concours annuel du code C fonctionnel le plus illisible possible[2].
- L'efficacité du code est aussi liée à l'absence de vérifications et de contrôles lors de l'exécution, ainsi qu'à une grande liberté laissée au compilateur. Cela signifie qu'un code correct sera performant mais qu'il est malheureusement possible d'écrire du code syntaxiquement légal mais faisant autre chose que ce que le programmeur souhaite. Cette liberté de programmation, en plus de favoriser les bogues, est aussi source de nombreuses failles de sécurité.
- La simplicité du langage a été obtenue en rejetant de nombreuses structures de données en-dehors du noyau syntaxique de base[3]. Il n'y a ainsi pas nativement de structures classiques telles que les listes, les piles, les tables de hachage, et le traitement des chaînes de caractères est assez basique. Il faudra soit développer le code correspondant à la main, soit utiliser des bibliothèques extérieures (ce qui peut poser un problème de portabilité).

1. Entre autre liée à l'époque de sa conception, quand l'espace mémoire était une denrée rare.
2. `www.ioccc.org`
3. Les détracteurs du C le décrivent comme un « assembleur poudré », sans aucun concept algorithmique de haut niveau, ni structures de données efficaces.

– L'ancienneté du langage le prive de fonctions standards d'interfaçage graphique. Là encore, il faudra avoir recours à des bibliothèques tierces qui ne seront pas forcément disponibles à l'identique sur tous les systèmes.

1.3 Composition d'un programme

Le langage C fait partie des langages dits impératifs qui permettent d'exprimer la résolution d'un problème en termes de séquences d'instructions à exécuter allant modifier l'état du programme et de la machine. Un programme C est donc composé d'une série d'instructions qui vont s'exécuter dans l'ordre à partir de la première. Certaines de ces instructions (tests, boucles, répétitions, sélections...) ont pour but d'affiner ce déroulé linéaire en proposant des alternatives d'exécution. Des variables globales forment une zone de stockage de données accessible depuis n'importe quelle instruction. Des fonctions permettent de regrouper des instructions ensemble afin de ne pas avoir à les répéter dans le code ; elles correspondent en quelque sorte à de nouvelles instructions que le programme définit une fois et utilise ensuite dans son code. Chaque fonction peut être paramétrée par des arguments et possède des variables locales lui servant à mémoriser des données pendant son exécution.

Fichiers

Un programme C est une collection de fonctions s'appelant les unes les autres. Une de ces fonctions doit avoir comme identifiant le nom `main()` ; elle sera le point d'entrée du programme par lequel commencera l'exécution. L'ensemble de ces fonctions se trouve dans un ou plusieurs fichiers source (classiquement caractérisés par un nom ayant une extension « `.c` ») dont l'un contient la fonction `main()`. À chacun de ces fichiers source peut être associé un fichier d'entête (ayant l'extension « `.h` », comme *header)* décrivant ce qui s'y trouve (en termes de fonctions et de variables globales) afin que les fonctions situées dans d'autres fichiers sources puissent y accéder correctement.

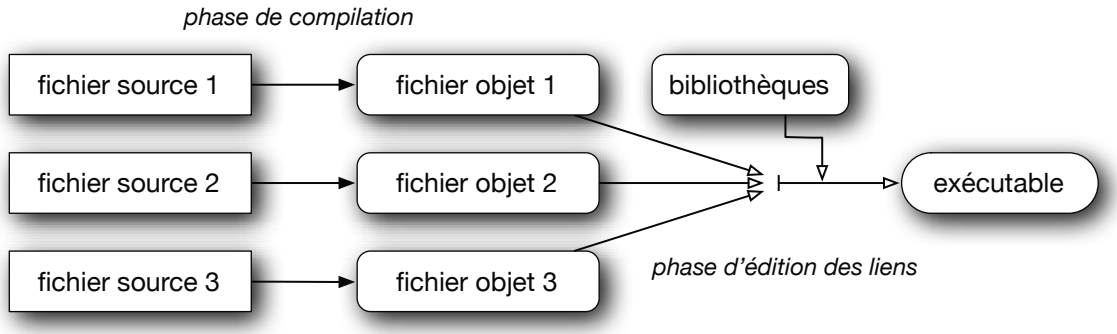

***Fig. I.1** – Organisation d'un programme*

La compilation va ensuite consister à traduire un fichier source en code machine placé dans un fichier objet. Une fois la compilation de tous les fichiers sources effectuée, la génération d'un code exécutable est possible en réunissant l'ensemble des fichiers objets du programme auxquels sont ajoutées les bibliothèques nécessaires. Ces dernières correspondent à du code déjà compilé provenant de l'extérieur et contenant des fonctions classiques auxquelles il sera fait référence dans le programme initial. Cette étape s'appelle l'édition des liens.

Compilation

Le passage d'un fichier source à un exécutable se décompose en plusieurs étapes (voir plus de détails en section X.3).

1. Le fichier source est tout d'abord lu par le préprocesseur dont la tâche consiste à effectuer un certain nombre de traitements sur le texte du programme lui-même, indépendamment de la syntaxe : définition de macros, inclusion de fichiers d'entête, compilation conditionnelle. Même si cette étape ne s'effectue jamais de manière isolée, elle ne fait pas partie stricto sensu de la compilation.
2. Le résultat est ensuite fourni au compilateur proprement dit qui va procéder à l'analyse lexicale (vérification des mots-clés), syntaxique (vérification de la grammaire du code) et sémantique du code source (vérification des types et association des objets avec leur définition). Le résultat de ces traitements est la génération d'un code intermédiaire interne proche des instructions machine.
3. Le programme issu de la compilation fait l'objet d'une optimisation : suppression du code inutile, calcul des constantes, réorganisation des boucles, allocation des registres, expansion *inline*... Cela permet d'avoir un code plus petit ou plus rapide.
4. Une fois optimisé, le code intermédiaire propre au compilateur est ensuite transformé en instructions assembleur en fonction de l'architecture cible : placement mémoire, utilisation des registres, choix des modes d'adressage...
5. Les instructions sont passées dans un assembleur qui génère un fichier objet composé d'instructions en langage machine, c'est-à-dire formé de codes numériques. Le travail de compilation s'arrête ici.
6. La dernière étape correspond à l'édition des liens qui est la réunion de tous les fichiers objets nécessaires au programme (fichiers objets du projet et bibliothèques externes) et la résolution des références croisées entre fichiers.

Environnement de programmation

L'ensemble du travail de compilation, regroupant toutes ces étapes, s'automatise assez facilement au sein des environnements de programmation. Il s'agit de logiciels permettant à l'utilisateur de réunir l'ensemble des outils utilisés lors du développement et de les piloter à partir d'une interface graphique unifiée (voir section XV.2.2).

1.4 Exemple : le « *hello, world* »

Depuis le livre de Kernighan et Ritchie, il est de tradition dans tout ouvrage de programmation d'illustrer très rapidement la syntaxe d'un langage en donnant un programme affichant la chaîne « *hello, world* » à l'écran[4].

***Listing I.1** – hello, world*

```
#include <stdio.h>

/* Voici la fonction */
main() {
    printf("hello, world\n");
}
```

4. Des variantes sont possibles : avec ou sans virgule, avec ou sans point d'exclamation ou retour à la ligne.

On peut distinguer les éléments suivants dans le code ci-dessus :

- `#include <stdio.h>` inclut le fichier d'entête standard `stdio.h` contenant les déclarations des fonctions d'entrées/sorties de la bibliothèque standard du C ;
- un commentaire est inclus entre les marques `/*` et `*/` ;
- `main()` est le nom de la fonction principale, point de début du programme ;
- les accolades `{` et `}` entourent les instructions constituant le corps de cette fonction ;
- `printf()` est un appel de fonction (située dans la bibliothèque standard) permettant d'écrire sur l'écran ;
- les guillemets `"..."` délimitent une chaîne de caractères. Dans celle-ci se trouvent les caractères à afficher, suivis d'un retour à la ligne indiqué par le caractère '`\n`' ;
- un point-virgule '`;`' termine toute instruction en C.

Pour simplifier l'écriture, la déclaration de la fonction `main()` oublie délibérément d'indiquer les éventuels paramètres de la fonction (entre les parenthèses) ainsi que le type de la valeur renvoyée par celle-ci. Cette façon de faire peut convenir ici puisque la fonction ne renvoie rien et n'a pas de paramètre mais, bien que conforme à une syntaxe C K&R, n'est pas très « propre » ni conforme à la norme C99, en raison de l'absence de précision sur le type renvoyé. Afin d'alléger les exemples, la première partie de cet ouvrage se contentera de cette déclaration de la fonction `main()` avant de présenter une définition plus correcte en section X.3.

Mise en pratique

Pour pouvoir lancer et exécuter le programme ci-dessus, il faut au moins deux outils : un éditeur de texte et un compilateur.

- L'éditeur de texte va permettre de taper et sauvegarder le code source dans un fichier texte (mais ayant une extension « `.c` »). En théorie, n'importe quel logiciel sachant éditer des textes pourrait faire l'affaire, mais il est plus pratique d'utiliser un éditeur intelligent sachant colorier le source suivant la syntaxe du C[5].
- Le compilateur va transformer le code source en code objet puis en fichier exécutable.

Ces deux programmes peuvent être récupérés et lancés séparément[6] mais le plus simple est d'utiliser un environnement de programmation (IDE) qui les réunit, avec d'autres outils, au sein d'une interface unifiée. Ces logiciels peuvent être assez complexes à maîtriser et le choix d'un IDE doit se faire en connaissance de cause. L'un des plus simples pour débuter est un IDE libre et gratuit : Code::Blocks[7]. Pour exécuter le programme du listing I.1, il faut :

- créer un nouveau projet avec Code::Blocks ;
- indiquer qu'il s'agit d'une application console[8] en C ;
- choisir le nom et l'emplacement du projet dans l'arborescence locale des fichiers ;
- Code::Blocks crée automatiquement un fichier `main.c` qu'il est possible d'éditer à la main dans la fenêtre de l'éditeur[9] ;
- une fois le code source écrit, une commande (menu principal, barre d'outils ou raccourci clavier) permet de le compiler puis de l'exécuter ;

5. Les couleurs distingueront les mots-clés, les chaînes de caractères, les commentaires...

6. Ainsi, cet ouvrage fera quelques fois référence au logiciel `gcc`, le compilateur propre aux systèmes Unix.

7. `www.codeblocks.org/`

8. C'est-à-dire nécessitant une simple fenêtre de texte pour son affichage.

9. Lorsque le projet inclut plusieurs fichiers sources, un menu dans la partie gauche de la fenêtre permet de choisir le fichier actif en cours d'édition.

– l'affichage du programme se fait alors dans une fenêtre console de l'application.

Lorsque la compilation se fait sous Unix en ligne de commande et pas avec un IDE, la compilation génère un fichier exécutable qu'il faut lancer à la main en tapant son nom comme commande dans un terminal.

1.5 Syntaxe de base du langage

Mise en page

Le format des fichiers C est libre : indentation, espacement, retour à la ligne sont laissés au choix du programmeur. La seule contrainte est de bien distinguer les mots-clés du langage en les écrivant en un seul morceau, sans espace. Bien sûr un format clair est indispensable à la lisibilité du code : indentation correcte des blocs et une instruction par ligne.

Jeu de caractères

Le seul jeu de caractères compatible avec tous les systèmes est le codage ASCII sans accent. C'est donc le choix évident pour un fichier source portable. Toute la syntaxe du langage (mots-clés, identifiants) doit utiliser ce codage. Il n'est donc pas question d'avoir des noms de variables ou de fonctions contenant des caractères accentués ou exotiques. Il est en revanche possible de les utiliser dans des chaînes de caractères mais, ces caractères exotiques étant codés différemment suivant les systèmes d'exploitation, ces chaînes risquent de ne plus être compréhensibles lors du portage du fichier sur un autre système.

Identificateurs

Chaque variable ou fonction (ainsi que les macros et types créés par le programmeur) est désignée par un identificateur qui doit respecter les règles suivantes :

- il est formé d'une suite de lettres (`A` à `Z`, `a` à `z`, sans utiliser de caractère accentué), de chiffres (`0` à `9`) ou du caractère souligné (`_`) ;
- le premier caractère de l'identificateur ne peut pas être un chiffre ;
- un identificateur commençant par le soulignement (`_`) est normalement réservé pour une utilisation par le système ; le programmeur peut utiliser ce caractère pour débuter un identificateur de son code mais il risque d'y avoir collision de nom avec celui d'une éventuelle bibliothèque extérieure ajoutée au programme ;
- les identificateurs sont sensibles à la casse (capitales/minuscules) ;
- un identificateur peut être de longueur quelconque ; cependant, la norme C impose au compilateur de tenir compte des 31 premiers caractères pour discriminer les noms internes et des 6 premiers pour les références externes : la quasi-totalité des compilateurs tiennent compte de l'intégralité de l'identificateur mais c'est une source possible de bogues dans le cas contraire.

Mots-clés

La liste ci-dessous indique les mots-clés réservés du langage (norme C89/90) qui ne peuvent pas être utilisés comme identificateur.

```
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
```

La norme C99 y a ajouté 5 mots-clés :

```
inline  restrict  _Bool  _Complex  _Imaginary
```

Bien sûr, rien n'empêche de les utiliser *à l'intérieur* d'un nom plus long : `longueur`, `integrale` ou `auPif` sont des identifiants possibles.

Commentaires

Les commentaires d'un programme se placent entre les marques `/*` et `*/` (comme dans l'exemple du listing I.1) et peuvent s'étendre sur plusieurs lignes. En revanche, il n'est pas possible d'imbriquer les commentaires. Ce sont les seuls commentaires admis par la norme C89/90. Le C99 permet des commentaires commençant par `//` et se terminant à la fin de la ligne (voir chapitre X). Cette dernière écriture est également autorisée comme extension du C89/90 par tous les compilateurs [10].

• *Tout au long de cet ouvrage, ce symbole mettra en évidence et rappellera des points cruciaux de programmation liés à des pièges du langage et des bogues classiques.*

10. Autrement dit, même sans demander la conformité avec le C99, un développeur pourra utiliser `//`, sous réserve de ne pas chercher un respect absolument strict avec le C89/90.