Introduction au langage C

Copyleft © GUINKO Tonguim Ferdinand - IBAM - Université de Ouagadougou

6 janvier 2010

Table des matières

1 Introduction au langage C **4**

1.1 Généralités sur les langages de programmation 4

1.1.1 Définition 4

1.1.2 Catégorisation 4

1.1.2.1 Les langages de première génération 4

1.1.2.2 Les langages de deuxième génération 5

1.1.2.3 Les langages de troisième génération 6

1.1.2.4 Les langages de quatrième génération 7

1.1.3 Langages de "type ALGOL" 7

1.1.3.1 Généralités 7

1.1.3.2 Caractérisques d'ALGOL 8

1.2 Historique du langage C 8

1.2.1 Langages précurseurs 8

1.2.1.1 Langage CPL 8

1.2.1.2 Langage BCPL 9

1.2.1.3 Langage B 9

1.3 Présentation du langage C 10

1.3.1 Historique : du B au C 10

1.3.2 Normalisation 10

1.3.3 Définitions 11

1.3.3.1 Fichier source et exécutable 11

1.3.3.2 L'identificateur 11

1.3.4 Fonctionnement du compilateur 12

1.3.4.1 Généralités 12

1.3.4.2 Les compilateurs 12

1.3.4.3 Phases de compilation 12

1.3.4.4 Types de fichiers 14
1.3.5 Structure d'un programme C 14

Chapitre 1

Introduction au langage C

1.1 Généralités sur les langages de programmation

1.1.1 Définition

Un langage de programmation est un logiciel qui permet la communication entre un utilisateur et l'ordinateur, à l'aide de programmes informatiques. Un programme informatique est une suite d'instructions indiquant à un système informatique une ou plusieurs tâches à effectuer. Ces instructions doivent être exécutée dans l'ordre par un processeur, et sont généralement regroupées ainsi qu'il suit :

1. Instructions d'entrée/sortie : par exemple `PRINTF` en langage C ;
2. Instructions de calculs : addition, soustraction, multiplication, division ;
3. Instructions logiques : pour permettre à l'ordinateur de prendre des décisions : par exemple effectuer un branchement conditionnel ou une boucle ;
4. Instructions de mouvements, de recherche et d'emmagasinage de données.

1.1.2 Catégorisation

Les langages de programmation peuvent être regroupés suivant leur dégré d'évolution ou d'incorporation des principes algorithmiques dans leur sémantique. Ainsi on distingue 4 générations de langages de programmation :

1.1.2.1 Les langages de première génération

On appelle langage de première génération les `langages machine` ou `codes-machine` utilisés initialement vers 1945. Ces langages sont les plus proches du matériel du système informatique avec lequel ils interagissent. Un programme en `code-machine` est une suite d'instructions élémentaires, composées uniquement de 0 et de 1, exprimant les opérations de base que la machine peut physiquement exécuter : instructions de calcul (addition, ...) ou de traitement (*et* logique, ...), instructions d'échanges entre

la mémoire principale et l'unité de calcul ou entre la mémoire principale et une mémoire externe, des instructions de test qui permettent par exemple de décider de la prochaine instruction à effectuer.

Exemples :

– voici en code machine de l'IBM 370 l'ordre de charger 293 dans le registre "3" : 01011000 0011 00000000000100100100
– pour dire à la machine d'additionner 1084, le programmeur doit entrer : 00100000000000000000000000010111000

Ce type de code binaire est le seul que la machine puisse directement comprendre et donc réellement exécuter. Tout programme écrit dans un langage évolué devra par conséquent être d'abord traduit en code-machine avant d'être exécuté. Dans les codes-machine, les erreurs de transcriptions sont fréquentes et la tâche du programmeur fort fastidieuse. Il faut en outre que le programmeur mémorise les nombres de chaque code utilisé et garde trace des instructions et des adresses où elles doivent s'exécuter. Programmer en langage machine est donc une opération coûteuse parce que très longue et la correction des erreurs n'en finit pas ; il est aussi très compliqué de modifier les programmes. De plus, chaque ordinateur possède son propre code : les programmes écrit en langage machine ne sont donc pas portables.

1.1.2.2 Les langages de deuxième génération

La deuxième génération est caractérisée par l'apparition des langages d'assemblage vers 1950 où chaque instruction élémentaire est exprimée de façon symbolique. Un programme dit `assembleur` assure la traduction en code exécutable, c'est à dire en langage-machine. Pour palier aux difficultés de programmation en langage-machine, on imagina au début des années 1950 une notation symbolique formée de mnémoniques pour représenter les instructions et d'adresses symboliques pour localiser les casiers de la mémoire de l'ordinateur. Le programme ainsi écrit s'appelle le `programme source`. Il est nécessaire qu'il soit transposé en langage-machine pour que l'ordinateur le comprenne. Cette traduction se fait à l'aide d'un assembleur qui produit un `programme objet`. Chaque ordinateur ou chaque microprocesseur autour duquel un micro-ordinateur est construit possède son propre assembleur. Ainsi, l'assembleur des ordinateurs IBM doit correspondre aux instructions des microprocesseur Intel de la famille 8086. On économise du temps en programmant avec un assembleur. Bien qu'étant d'un niveau plus élevé que le langage machine, le langage assembleur reste un langage de bas niveau car il demeure très proche de la machine, ce qui a pour conséquence la forte dépendance du langage assembleur du type de machine. Le premier assembleur connu a été mis au point pour l'ordinateur EDSAC développé en Angleterre. Le premier assembleur commercial s'appelle SAP (Symbolic Assembly Program) ; il fonctionne sur un IBM 704 et a été mis au point par la United Aircraft Corporation.

Microsoft, en août 1979, met sur le marché un assembleur pour micro-ordinateurs à base de microprocesseur 8080/Z80.

Exemples :

- *Exemple 1 :* L'instruction de l'exemple précédent s'écrira :
  constante X = 293 charger X R3
- *Exemple 2 :* le écrit en assembleur pour le microprocesseur Intel 8088

Etiquette	**Mnémonique**	**Opérande**	**Commentaires du programmeur**
PERIOD :	CALL	CONIN	;prendre un caractère du clavier
CMP	AL,	'.'	;le comparer à un point
JNE	INST		;aller plus loin s'ils ne coincident pas
RET			;s'ils coïncident, afficher le drapeau de retenue
INST :	STC		;s'ils ne coïncident pas, libérer le drapeau de retenue
RET			

1.1.2.3 Les langages de troisième génération

La troisième génération débute en 1957 avec le 1er langage dit évolué ou encore de haut niveau : FORTRAN (acronyme de mathematical FORmula TRANslating system). Apparaissent ensuite ALGOL (ALGOrithmic Language en 1958) [1], COBOL (COmmon Business Oriented Language en 1959), BASIC (Beginner's All-purpose Symbolic Instruction Code en 1965), Pascal (1968), C (1972) C++ (1983), Java (1995) ... Les concepts employés par ces langages sont beaucoup plus riches et puissants que ceux des générations précédentes et leur formulation se rapproche du langage mathématique. Ces langages sont plus faciles à utiliser car ils sont plus faciles à apprendre, mieux documentés et plus rapides à utiliser : les coûts de programmation s'en trouvent réduits. Ces langages sont aussi construits autour de mnémoniques qui rappellent le langage naturel. Par exemple, l'ordre d'imprimer un fichier, à l'ordinateur, en langage C sera donné par l'instruction `PRINTF`. La plupart des langages de haut niveau ont été écrits pour des applications spécifiques. Cependant, les programmes écrits en ces langages doivent aussi être transposés dans le langage que reconnaît l'ordinateur.

Il existe deux méthodes pour rendre exécutables les programmes écrits en un langage de haut niveau :

[1] `http://fr.wikipedia.org/wiki/Algol_(langage)#cite_ref-0`

1. la **compilation** : l'ensemble du code source du programme est globalement traduit par un autre programme appelé compilateur, et le code-machine produit est optimisé. Ce code-machine est ensuite exécuté autant de fois qu'on le veut. Le résultat direct de la compilation est un programme objet. Seul le programme objet est exécutable. A la fin du processus de compilation, le compilateur indique les erreurs du programme, qu'il faut corriger, avant de recompiler le programme pour pouvoir l'exécuter. Seul un programme sans erreur détecté par le compilateur est exécutable.
2. l'**interprétation** : un programme (interpréteur) décode et exécute une à une et au fur et à mesure les instructions du programme source. Le résultat de l'interprétation du programme source est donc aussi un programme objet. Les erreurs sont plus faciles à corriger car elles ne nécessitent pas que le programme soit recompilé avant de l'exécuter de nouveau, et souvent, elles sont indiquées par l'ordinateur au fur et à mesure que le programmeur écrit ses lignes de programme. Les résultats de l'exécution sont fournis au fur et à mesure du déroulement du programme. Les programmes interprétés sont généralement plus lents au niveau de l'exécution que les programmes compilés, car l'ordinateur doit exécuter les instructions les unes à la suite des autres.

1.1.2.4 Les langages de quatrième génération

La quatrième génération qui commence au début des années 80 devait mettre l'outil informatique à la portée de tous, en supprimant la nécessité de l'apprentissage d'un langage évolué. Ses concepts fondamentaux sont `convivialité` et `non procéduralité` (il suffit de *dire* à la machine ce qu'on veut obtenir sans avoir à préciser comment le faire). Cet aspect a été rencontré avec les langages du type Visual qui prennent en charge l'élaboration de l'interface graphique.

1.1.3 Langages de "type ALGOL"

1.1.3.1 Généralités

ALGOL est l'acronyme de ALGorithmic Oriented language a été crée en 1958. Son objectif était de décrire algorithmiquement des problèmes de programmation. Les principales différences au niveau de la conception par rapport à ces prédécesseurs, en particulier Fortran, furent l'utilisation de blocs marqués par les marqueurs `BEGIN` et `END` et surtout la `récursivité`, concepts qui seront largement repris par ses successeurs. Malgré le fait qu'ALGOL n'ait pas atteint la popularité commerciale de Fortran et Cobol, il est considéré comme étant le plus important langage de sa génération compte tenu de l'influence qu'il a exercé sur le développement des langages de programmation. En effet, le lexique et la structure syntaxique d'ALGOL sont devenus si populaires que tous les langages conçus depuis sont étiquettés comme étant `de type algol`.

1.1.3.2 Caractérisques d'ALGOL

ALGOL est un langage typé, procédural récursif et à structure de blocs ; Il permet aussi la construction de types dynamiques. En outre, ALGOL présente les avantages suivants :
- tableau dynamiques
- mots réservés : les mots ou symboles réservés ne peuvent pas être utilisés dans le code
- types de données prédéfinis

1.2 Historique du langage C

1.2.1 Langages précurseurs

Le langage C est un langage algorithmique de la famille ALGOL, et dérive successivement des langages CPL (Combined Programming Language), BCPL (Basic Combined Programming Language), et B, d'où le nom de langage C.

Il a été conçu par Kernighan et Ritchie aux laboratoires Bell d'AT&T. Leur objectif premier était de réecrire en langage évoluée le système d'exploitation UNIX de manière à assurer sa portabilité. Il s'est montré en fait plus polyvalent et est utilisé aussi bien pour écrire des applications de calcul scientifique et de gestion. C'est un langage très utilisé dans l'industrie car il cumule les avantages d'un langage de haut-niveau (portabilité, modularité, etc...) et ceux des langages assembleurs proches du matériel

1.2.1.1 Langage CPL

CPL [2] (Combined Programming Language) a été conçu au début des années 1960 par les universités de Cambridge et de Londres ; cette collaboration est à l'origine du mot `Combined` dans le nom final du langage (qui était originellement `Cambridge Programming Language`). CPL fut un grand projet académique consistant à créer un langage englobant beaucoup de concepts. CPL devait notamment être fortement typé, avec de nombreux types comme des nombres entiers, réels, booléens, caractères, tableaux, listes... CPL a été grandement influencé par le langage ALGOL mais manqua de légereté, d'élégance et de simplicité. Il semble que personne n'ait réussi à terminer l'écriture d'un compilateur pour ce langage, qui était censé être bon à la fois pour la programmation scientifique (à la manière du FORTRAN et de l'ALGOL) et également pour la programmation commerciale (comme le COBOL). A cause de sa trop grande complexité, pour l'époque, CPL disparu sans laisser de trace dans les années 1970, et le BCPL lui succéda.

[2] `http://fr.wikipedia.org/wiki/Combined_Programming_Language`

1.2.1.2 Langage BCPL

BCPL [3] (Basic CPL mais originellement Bootstrap CPL) a été conçu à Cambridge en 1966 par Martin Richards, en réponse aux difficultés rencontrées durant les années 1960 avec son prédécesseur le language (CPL). L'année suivante il alla au MIT et écrivit un premier compilateur. BCPL est une version fortement simplifiée de CPL avec notamment un seul type de donnée, le mot *machine*, c'est à dire le nombre typique de bits que le processeur d'un ordinateur traite en une instruction machine (addition, soustraction, multiplication, copie...) La notion est devenue un peu floue avec les processeurs actuels qui peuvent traiter des données de toutes sortes de tailles. Cependant on peut raisonnablement classer les Pentium et PowerPC parmis les processeurs 32 bits, contre 64 bits pour les Alpha, Itanium ou Opteron. Du temps de BCPL on trouvait des architectures à mots de 40 bits, 36 bits, 18 bits, 16 bits...

BCPL opère sur les mots de la machine. Il est donc à la fois portable et proche du matériel, donc efficace pour la programmation système. BCPL a servi à écrire divers systèmes d'exploitation de l'époque, dont un (TripOS) qui s'est retrouvé partiellement porté sur l'Amiga pour devenir la partie AmigaDOS du système.

Mais aujourd'hui BCPL ne semble plus être utilisé que par son inventeur. C'est sans doute dû au fait que C a repris et étendu la plupart des qualités de BCPL. Ken Thompson l'a fait évoluer en le simplifiant davantage, afin de pouvoir l'utiliser sur de petits ordinateurs et en y ajoutant quelques changements pour correspondre à ses goûts personnels (par exemple réduire le nombre de caractères dans un programme), donnant ainsi naissance au langage B.

1.2.1.3 Langage B

Le langage B [4] a été créé par Ken Thompson en 1971 dans les laboratoires Bell d'AT&T. En 1969/1970, il avait écrit en assembleur la première version du système d'exploitation UNIX sur un PDP-7 contenant 8 kilo-mots de 18 bits. Le langage B est une simplification du langage BCPL, un peu forcée par les limitations du PDP-7. Mais la syntaxe très sobre du langage B (et des commandes UNIX) toute en lettres minuscules correspond surtout aux goûts personnels de Thompson.

Le langage B a été porté et utilisé sur quelques autres systèmes. Mais cette même année 1970, le succès du projet UNIX justifia l'achat d'un PDP-11. Celui-ci avait des mots machine de 16 bits mais il était aussi capable de traiter des octets (24 Ko de mémoire vive en tout) dans chacun duquel pouvait être stocké un caractère. Le langage B ne traitait que des mots machines, donc le passage de 18 à 16 bits n'était pas problématique, mais il était impossible de traiter efficacement les caractères de 8 bits. Pour bien exploiter les capacités de la machine, le langage B a donc commencé à être étendu en ajoutant un type pour les caractères ...

À partir de 1971, Dennis Ritchie [5] fit évoluer le langage B, pour répondre à ces

[3] `http://www.lysator.liu.se/c/clive-on-bcpl.html`
[4] `http://en.wikipedia.org/wiki/B_(programming_language)`
[5] `http://fr.wikipedia.org/wiki/Dennis_Ritchie`

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

1.3 Présentation du langage C

1.3.1 Historique : du B au C

Le langage C a été inventé aux Bells Labs en 1972 par Dennis Ritchie pour permettre l'écriture du système d'exploitation UNIX [6], alors développé par Ken Thompson [7] et Dennis Ritchie. Le langage C a repris la syntaxe du langage B avec un minimum de changements ; c'est un langage très utilisé dans l'industrie car il cumule les avantages d'un langage de haut-niveau (portabilité, modularité, etc...) et ceux des langages assembleurs proches du matériel.

En 1978, Brian W. Kernighan documenta abondament le langage et publia avec Dennis Ritchie le livre de référence *The C Programming Language*. On appelle souvent `C K&R` le langage tel que spécifié dans la première édition de ce livre. Dans les années qui suivirent, le langage C fut porté sur de nombreuses autres machines. Ces portages ont souvent été faits, au début, à partir du compilateur pcc de Steve Johnson, mais par la suite des compilateurs originaux furent développés indépendamment. Durant ces années, chaque compilateur C fut écrit en suivant les spécifications du K&R, mais certains ajoutaient des extensions, comme des types de données ou des fonctions supplémentaires, ou interprétaient différemment certaines parties du livre (qui n'était pas forcément très précis). À cause de cela, il fut de moins en moins facile d'écrire des programmes en C qui puissent fonctionner tels quels sur un grand nombre d'architectures.

1.3.2 Normalisation

Pour résoudre ce problème, et ajouter au C des possibilités dont le besoin se faisait sentir (et déjà existantes dans certains compilateurs), en 1989, l'organisme national de normalisation des Etats Unis d'Amérique (ANSI) normalisa le langage C. Le nom exact de la norme définie est `ANSI X3.159-1989 Programming language C`, mais elle fut (et est toujours) connue sous les dénominations `ANSI C` ou `C89`. Puis l'ANSI soumit cette norme à l'Organisation internationale de normalisation (ISO), qui l'accepta telle quelle et la publia l'année suivante sous le numéro `ISO/IEC 9899 :1990` (le document étant connu sous le nom ISO C90, ou C90).

En 1999, l'organisme ISO proposa une nouvelle version de la norme. Le nouveau document, qui au niveau de l'ISO est celui ayant autorité aujourd'hui, est `ISO/IEC 9899 :1999`, connu sous le sigle `C99` [8].

[6] `http://tonguim.free.fr/ibam/miage2_soir_20082009/chap3IntroductionLinux_pln.pdf`
[7] `http://fr.wikipedia.org/wiki/Kenneth_Thompson`
[8] `http://www.liafa.jussieu.fr/~yunes/systemes/stdlib.html`

1.3.3 Définitions

1.3.3.1 Fichier source et exécutable

Un fichier peut être défini comme une entité regroupant un ensemble d'informations, stocké sur un support physique (disque par exemple) et manipulable grâce à un système d'exploitation.

On distingue différents type de fichiers :
– des fichiers exécutables (applications) : un fichier exécutable contient du code directement interprétable et exécutable par le processeur. Sous dos, ces fichiers prennent l'extension *.exe* ou *.com*. Sous UNIX c'est une propriété donnée au fichier qui indique au système qu'il s'agit d'un exécutable.
– des fichiers binaires : ils contiennent du code machine mais ne sont pas directement exécutables : ce sont des fichiers objets.
– des fichiers texte (ASCII) [9] : ces fichiers contiennent uniquement des codes dit ASCII (codés sur 7 bits). C'est le type de fichiers le plus portable.
– des fichiers liés à une application particulière : fichiers musicaux (mp3, waw,...) images (jpeg, gif, tif, png, etc...), vidéo, traitement de texte (doc, wpd ...).

L'objectif d'un programmeur est bien sur d'arriver à générer (puis exécuter) un fichier exécutable. Ceci passe par plusieurs étapes que nous allons décrire dans le cas d'un programme en langage C :

1. la première étape consiste à écrire le programme (on parle de source) dans un fichier texte à l'aide d'un éditeur. En C, on donne l'extension **.c** à ce fichier. Ce programme est bien évidemment incompréhensible par la machine ;
2. la deuxième étape est l'étape de précompilation. Elle consiste à traiter les directives de compilation (comme l'inclusion de fichiers d'entête de bibliothèques) et génère un fichier texte qui est encore un fichier source en C ;
3. l'étape suivante est la compilation. Elle consiste à transformer les instructions du programme en langage compréhensible par le processeur (langage machine) et génère un fichier binaire dit fichier objet ; Dans certains documents les deuxième et troisième étapes ne forment qu'une seule étape et sont assimilées à la compilation ;
4. la dernière étape consiste à éffectuer l'édition de liens. Le code généré à la compilation est complété par le code des fonctions des bibliothèques utilisées. C'est seulement après cette étape que l'on génère un fichier exécutable.

En général, ces étapes sont transparentes pour l'utilisateur.

1.3.3.2 L'identificateur

L'identificateur est le nom donné aux différentes composantes d'un programme : variables, fonctions, constantes ... Il doit être formé d'au maximum 31 caractères

[9] `http://fr.wikipedia.org/wiki/American_Standard_Code_for_Information_Interchange`

pris dans l'ensemble composé des lettres a-z et A-Z, des chiffres 0-9 et du caractère spécial (soulignement).

Le C tient compte des majuscules et des minuscules. Par exemple, Index et index sont deux identificateurs différents. Un identificateur peut être un mot quelconque à l'exception. 32 mots sont réservés dans un programme en C et ne peuvent pas être des identificateurs.

Voici ces mots (inutile de les apprendre) : *auto, double, int, struct, break, else, long, switch, case, enum, register, typedef, char, extern, return, union, const, float, short, unsigned, continue, for, signed, void, default, goto, sizeof, volatile, do, if, static, while.*

1.3.4 Fonctionnement du compilateur

1.3.4.1 Généralités

Le langage C est un langage multi plateforme, c'est à dire qu'il peut fonctionner sur de nombreuses machines et sous de nombreux systèmes d'exploitation à condition d'avoir un compilateur adéquat : Le programme est écrit dans un fichier texte. Ce fichier est ensuite compilé en exécutable (par le compilateur et le linker). Pour qu'un programme fonctionne sur une autre plateforme (par ex. un autre système d'exploitation), il suffit de recompiler le programme avec un compilateur fonctionnant pour cette plateforme. C'est cette fonctionnalité qui a contribué a rendre le C aussi célèbre. Aujourd'hui la plupart des programmeurs utilisent des fonctions spécifiques de leur plateforme, ce qui rend le port du programme quasi impossible sur une autre plateforme.

1.3.4.2 Les compilateurs

1.3.4.3 Phases de compilation

La compilation est l'opération qui consiste à lire un fichier source et à le traduire dans le langage binaire du processeur. Ce langage est absolument incompréhensible par les êtres humains, cependant, il existe un langage de programmation qui permet de coder directement le binaire : il s'agit de l'assembleur présenté brièvement ci-dessus.

La compilation se fait en plusieurs phases :

En général, le processus de compilation génère un fichier binaire pour chaque fichier source. Ces fichiers binaires sont nommés fichiers objets, et porte de ce fait l'extension **.o** » (ou **.obj** dans les systèmes Microsoft). Comme un programme peut être constitué de plusieurs fichiers sources, il faut regrouper les fichiers objets pour générer le fichier exécutable du programme, fichier que l'on appelle également le fichier image en raison du fait que c'est le contenu de ce fichier qui sera chargé en mémoire par le système pour l'exécuter et qu'il contient donc l'image sur disque des instructions des programmes en cours d'exécution. L'opération de regroupement des fichiers objets pour constituer le fichier image s'appelle l'édition de liens, et elle est réalisée par un programme nommé le linker (éditeur de liens en français). Ce programme

regarde dans tous les fichiers objets les références partielles aux autres fichiers objets et, pour chaque *lien* ainsi trouvé, il complète les informations nécessaires pour en faire une référence complète. Par exemple, un fichier source peut très bien utiliser une fonctionnalité d'un autre fichier source. Comme cette fonctionnalité n'est pas définie dans le fichier source courant, une référence partielle est créée dans le fichier objet lors de sa compilation, mais il faudra tout de même la terminer en indiquant exactement comment accéder à la fonctionnalité externe. C'est le travail de l'éditeur de liens, lorsqu'il regroupe les deux fichiers objets.

Certains fichiers objets sont nécessaires pour tous les programmes. Ce sont notamment les fichiers objets définissant les fonctions de base, et les fonctions permettant d'accéder au système d'exploitation. Ces fichiers objets ont donc été regroupés dans des bibliothèques (également couramment appelées *librairies*, que l'on peut ainsi utiliser directement lors de la phase d'édition de liens. Les fichiers objets nécessaires sont alors lus dans la bibliothèque et ajoutés au programme en cours d'édition de liens. Les bibliothèques portent souvent l'extension **.a** (ou **.lib** dans les systèmes Microsoft).

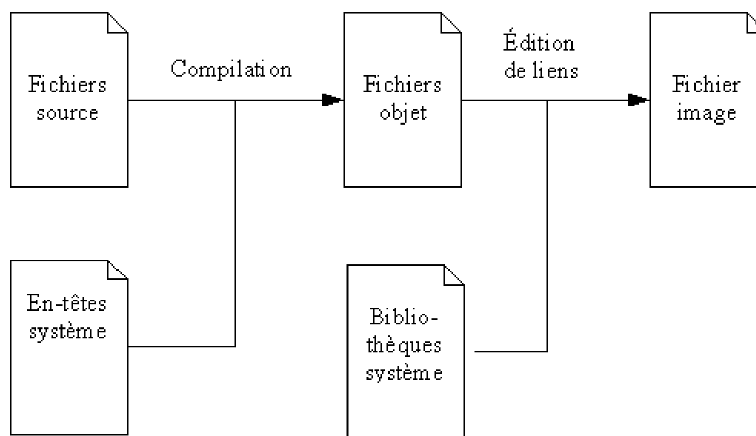

Fig. 1.1 – Etapes de compilation)

Malheureusement, la solution consistant à stocker dans des bibliothèques les fonctions les plus utilisées souffre de la duplication du code contenu dans ces bibliothèques dans tous les programmes, d'où une perte de place considérable. De plus, la mise à jour d'une bibliothèque nécessite de refaire l'édition de liens de tous les programmes qui l'utilisent, ce qui n'est pas réalisable en pratique. C'est pour cela que les bibliothèques dynamiques ont été crées : une bibliothèque dynamique n'est pas incluse dans les fichiers des exécutables qui l'utilisent, mais reste dans un fichier séparé. Les bibliothèques sont regroupés dans un répertoire bien défini du système de fichiers, ce qui permet de les partager entre différents programmes. Ainsi, la mise à jour d'une bibliothèque peut se faire sans avoir à toucher tous les programmes qui l'utilisent. Le problème est cependant que l'édition de liens reste incomplète, parce que les références aux objets des bibliothèques dynamiques sont toujours externes. Il existe donc un programme spécial, l'éditeur de liens dynamiques (*ld*, pour *Link Dynami-*

cally), qui résout les dernières références incomplètes lors du chargement de chaque programme. Les bibliothèques dynamiques portent l'extension **.so** (pour *Shared Object*, ou **.dll** dans les systèmes Microsoft (pour *Dynamic Link Library.* Évidemment, le chargement des programmes est un peu plus lent avec les bibliothèques dynamiques, puisqu'il faut réaliser l'édition de liens dynamiques lors de leur lancement. Cependant, ce processus a été optimisé, et les formats de fichiers binaires utilisés contiennent toutes les informations précalculées pour faciliter la tâche de l'éditeur de liens dynamiques. Ainsi, la différence de performance est devenue à peine décelable.

Dans le cas du langage C, une phase supplémentaire apparaît dans le processus de compilation, il s'agit du traitement du fichier par le préprocesseur C, un programme permettant d'inclure dans le fichier source les éléments référencés par les instructions situées au début du fichier source (instructions précédées du caractère #). C'est donc le préprocesseur qui ajoutera dans le fichier source la définition de la fonction `printf()` qu'il sera allé chercher dans le fichier `stdio.h` grâce à l'instruction `#include`. Les phases de compilation dans le cas d'un compilateur C sont décrites dans la figure 1.2 :

1.3.4.4 Types de fichiers

Classiquement, les fichiers sources C se divisent en deux grande catégories :

- les fichiers d'en-tête, qui contiennent les déclarations des symboles du programme ;
- les fichiers C, qui contiennent leurs définitions.

Le fait de faire cette distinction entre la déclaration (qui consiste à dire : *telle chose existe* et la définition (qui consiste à décrire la chose précédemment déclarée) permet de faire en sorte que l'on peut utiliser les fichiers objets sans avoir les fichiers sources. En effet, la déclaration permet de réaliser les références partielles dans les programmes, tandis que les fichiers objets contiennent bien entendu la définition binaire de ce qui a été déclaré.

Pour compiler un programme, on n'a donc réellement besoin que de trois types de fichiers :

- les fichiers de déclaration du système et des bibliothèques de base ;
- les fichiers des bibliothèques de base eux-mêmes ;
- les fichiers source (de déclaration et de définition) du programme à compiler.

En C, les fichiers sources de déclaration portent l'extension **.h**, et les fichiers de définition portent l'extension **.c** pour les programmes écrits en C.

1.3.5 Structure d'un programme C

Les règles fixant la structure générale d'un programme sont très souples en C. La succession des différentes sections peut ainsi varier d'un programme à un autre.

Exemple :

```
/*Calcul des puissances successives de 2*/
```

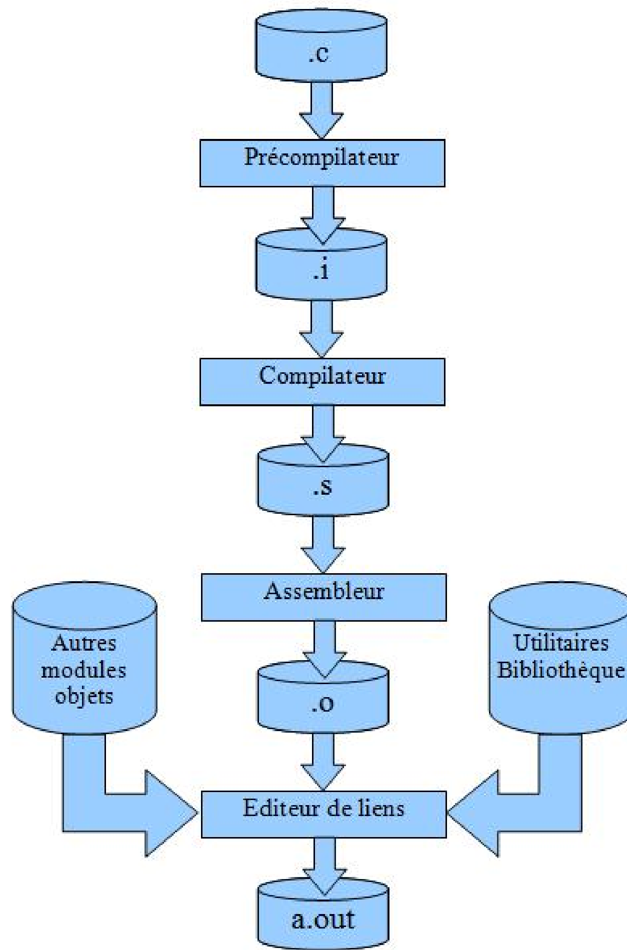

FIG. 1.2 – Etapes de compilation du langage C (http://www.dev-fr.org/index.php?topic=3278.0/)

```
#include<stdio.h>
/*---------------------------------------------------------*/
long int puissance(long int X, int Y); /* calcule x^y */
/*---------------------------------------------------------*/
int main()
 {
    long int resultat;
    int i;
    for (i=0; i<=N; i++)
    {
        resultat=puissance(2,i);
        printf("2 puissance %d =%ld\n",i,resultat);
```

```
    }
    return 0;
}
/*-----------------------------------------------------------*/
long int puissance(long int X, int Y)
{
    int i;
    long int P=1;
    for (i=0; i<Y; i++)
    P=X*P;
    return P;
}
```

On distingue dans ce programme une première zone contenant des directives de compilation (ligne précédée du caractère #). On trouve ensuite une entête de fonction (déclaration). Suit le code des fonctions composant le programme en commençant par la fonction main correspondant au programme principal. Tout programme en C commence dans une fonction appelée **main**. En général, la fonction **main** retourne au système d'exploitation un nombre indiquant un code d'erreur s'il n'a pas pu s'exécuter correctement et 0 (le chiffre 0) s'il n'y a pas d'erreur. En C, un bloc est une partie de programme délimitée par des accolades et commençant par un identificateur ou un mot clé. Une fonction est un bloc : elle est délimitée par des accolades. Voici un exemple de fonction main :

```
main ()
{
}
```

L'espace entre les parenthèses est vide car aucune variable n'est passée à la fonction main (par le système d'exploitation). L'ordre dans lequel les fonctions apparaissent dans le fichier source n'a pas beaucoup d'importance. Elles doivent seulement se trouver à la suite de leur déclaration. Par souci de clarté, en général la fonction main est placée en premier et les autres fonctions sont placées après elle.

Typographie

Le C utilise les espaces pour différencier les instructions successives et non le retour à la ligne, comme c'est le cas en langage *BASIC*, ce qui signifie qu'en C on peut écrire son programme sur une seule ligne.

Exemple :

```
main () { }
```

est équivalent à :

```
main ()
{
}
```

Un programmeur peut très bien rendre son programme difficile à comprendre.

Exemple :

```
for
(i=1;i<=10
;i++){scanf
("%d"
,i);printf
("%d\
-> %x",
i, i)
;}
```

est beaucoup plus difficile à comprendre que :

```
for(i=1; i<=10; i++)
{
scanf("%d", i);
  printf("%d -> %x");
}
```

Chaque instruction d'une fonction doit se terminer par un point-virgule. Par exemple pour une instruction `printf` :

```
printf("Bonjour...");
```

ou encore :

```
a=b+c;
```

Petit programme

Voici un petit programme commenté. Chaque fonction sera détaillée plus loin.

```
#include <stdio.h>         /* Ajoute les fonctions de stdio.h */
main ()
{               /* Début */
  printf("Bonjour,\n"); /* Affiche du texte */
  printf("Voici un petit programme en C.\n");
  return 0;
}                       /* fin de la fonction main */
```