Langage C

Architecture et environnement de programmation

Thomas Guyet

AGROCAMPUS-OUEST, Centre de Rennes

Version 1.0, Dernière modification : 18 avril 2012

Table des matières

1 Architecture des ordinateurs **4**
1.1 Codage binaire de l'information . . . 4
1.1.1 Bits et mots mémoires . . . 5
1.1.2 Codage des nombres entiers . . . 5
1.1.3 Codage des nombres entiers relatifs . . . 6
1.1.4 Codage des nombres à virgule flottante (simple précision) . . . 7
1.1.5 Représentation des textes . . . 10
1.1.6 Remarque sur la sémantique du binaire . . . 12
1.2 Architecture de von Neumann . . . 12
1.2.1 Présentation de l'architecture de von Neumann . . . 13
1.2.2 Image de l'organisation de la mémoire . . . 14
1.2.3 Image du fonctionnement d'un processeur . . . 14
1.3 Compilation . . . 17
1.4 Correction des exercices . . . 18

2 Environnement de programmation **20**
2.1 Programmer en environnement Linux (sans IDE) . . . 20
2.1.1 Éditeurs . . . 20
2.1.2 Compilateurs `gcc`, `g++` . . . 21
2.1.3 Débuggage avec `gdb` . . . 22
2.1.4 Debuggage avec `valgrind` . . . 25
2.1.5 Profilage avec `gprof` . . . 25
2.2 Programmer avec `Code::Blocks` . . . 26
2.2.1 Les IDE pour le développement en C . . . 26
2.2.2 Premier projet de `Code::Blocks` . . . 27
2.2.3 Fonctionnalités de `Code::Blocks` . . . 29

3 Complément de compilation en C **31**
3.1 Introduction à la compilation . . . 31
3.1.1 Notions d'analyse syntaxique . . . 32
3.1.2 Notion d'optimisation du code . . . 33
3.2 Le pré-processeur . . . 34

3.2.1 Exemple de l'effet du pré-processeur lors de la compilation 35
3.2.2 Utilisation des `define` dans les `.h` 35
3.3 Compilation séparée 36
3.3.1 Compilation d'un fichier `.c` en fichier objet avec `gcc` 36
3.3.2 Édition de liens entre plusieurs fichiers `.o` 37
3.4 Compilation rapide d'un fichier unique 37
3.5 Utilisation d'un Makefile 38
3.5.1 Un exemple simple 38
3.5.2 Utilisation de variables dans le Makefile 39
3.6 Options avancées de compilation 40

CHAPITRE

1 Architecture des ordinateurs

Le langage C est un langage dit de “bas niveau”, c’est-à-dire que ce qui est écris dans un programme en C est très proche de ce que fait le processeur lors de l’exécution du programme. J’entends par là que lors qu’on écrit une instruction C, on écrit quasiment ce que va faire le processeur lors de l’exécution du programme. Ce langage se différencie donc très fortement des langages tels que Java, R ou Python pour lesquels les instructions écrites peuvent être très “loin” de la façon dont cela se déroulera sur le processeur.
Ceci se traduit également sur les objets qui sont manipulés (les variables). En C, il est facilement possible de connaître quel est le codage informatique qui est fait d’une variable, alors qu’en R, par exemple, cette connaissance est loin d’être immédiate (et en pratique nécessite beaucoup de connaissance en C).
Il est donc très utile de comprendre le fonctionnement d’un processeur. S’il n’est pas nécessaire de comprendre tous les détails, des images très générales de son fonctionnement pourront se montrer très utiles par la suite. C’est ce à quoi s’intéresse l’architecture des ordinateurs

Definition 1 - Architecture des ordinateurs

L’architecture des ordinateurs (ou architecture des processeurs) décrit :
- (architecture matérielle, interne) l’agencement de composants électroniques ainsi que leurs interactions,
- (architecture externe) le modèle de fonctionnement du processeur.

L’architecture comprend notamment la donnée d’un **jeu d’instructions**, d’un ensemble de **registres** visibles par le programmeur, d’une **organisation de la mémoire** et des entrées sorties, des modalités d’un éventuel support multiprocesseurs, etc. (*source Wikipedia*)

1.1 Codage binaire de l’information

Tout l’art de l’informatique consiste à faire traiter des informations pour un ordinateur. Mais ces informations sont d’une nature particulière, elles sont numériques : toutes les informations sont échantillonnées et, en particulier, les nombres sont approximés. Techniquement, nous allons voir que ces nombres sont codés en utilisant un codage binaire.
Dans les cursus de programmation avancé, l’étude de l’algorithmique est toujours associé à l’étude des structures de données. Pour des données complexes (par exemple, un annuaire de personnes, une chaîne d’ADN, ...), il est très souvent possible de représenter l’information de plusieurs manière sous une forme numérique. En conséquent, les algorithmes de traitement de cette information seront également différents. Pour réaliser une tâche, le choix d’une bonne structure de données et d’un bon algorithme va de pair.

Mais les choix de la structure de l'information touchent également les informations de bas niveau telles que la représentation des nombres. Dans la suite de ce chapitre, on comprendra que la récente apparition des processeurs de calcul sur 64 bits induit des représentations des données différentes des processeurs 32 bits et que cela à des conséquences pratiques, en particulier, pour les applications de calcul numérique pour lesquels la représentation des nombres à un effet très important.
Dans la suite de cette section, on présente la représentation des données élémentaires.

1.1.1 Bits et mots mémoires

Definition 2 - Bit (Binary Digit)

Un bit est un chiffre qui peut prendre deux valeurs 0 ou 1.

Électroniquement, seuls les bits sont représentés dans la mémoire de l'ordinateur et les circuits électroniques ne sont capables que de faire des opérations sur les bits : c'est-à-dire donner des valeurs de bits (0 ou 1) en fonction de la valeur d'autres bits ... Les informations sont représentées, ou encore codées, en mémoire comme une suite de bits. Pour faciliter la conception des circuits électroniques, la suite de bits n'est pas de longueur arbitraire mais plutôt d'une longueur prédéfinie multiple d'une puissance de 2 (pour des raisons d'optimisation des traitements). On parle alors de mots mémoires, ou mots.

Definition 3 - Mot

Un mot, ou un mot mémoire, est une suite de bits d'une longueur prédéfinie (puissance de 2) pour une architecture électronique.

Exemple 1

Pour un processeur dit "32 bits", les mots utilisés par les processeurs sont de taille 32, et pour les architectures dites "64 bits", les mots sont de taille 64.

Si on veut qu'un ordinateur fasse des opérations sur des nombres, il faut arriver à traduire cette opération sous la forme d'opérations sur des bits. La première étape consiste d'abord à être capable de représenter des objets complexes tels que des nombres ou des caractères sous la formes de bits. C'est ce à quoi s'intéresse le codage de l'information.
Un codage n'étant jamais parfait, il faut bien être conscient que cette étape préliminaire du codage de l'information sous forme binaire (discrétisée, et de taille finie) est la source de beaucoup d'incapacités fondamentales des outils informatiques.
Dans la suite de cette section, nous présentons le codage pour les **types de bases** : nombres entiers, nombres relatifs, nombre à virgule flottante et chaînes de caractères.

1.1.2 Codage des nombres entiers

Prenons le cas d'un mot mémoire de 16 bits. Pour traduire un mot m sous la forme d'un nombre entier, il faut lire les bits de droite à gauche. Le bit de poids faible (le plus à droite) vaut 2^0, le suivant

vaut 2^1, le suivant vaut 2^2 ... Pour obtenir la valeur v du nombre entier, il faut sommer les valeurs des bits à 1. On note m_i la valeur (1 ou 0) du ième bit à partir de la droite dans le mot m.

$$v = \sum_{i=0}^{i=15} m_i.2^i$$

Se codage du bit donnant le bit de poids faible à droite est dit "little endian" et s'oppose au codage des nombres en "big endian". Le "little endian" est le plus fréquemment rencontré dans les ordinateurs usuels et je me tiendrai à ce codage pour la suite.

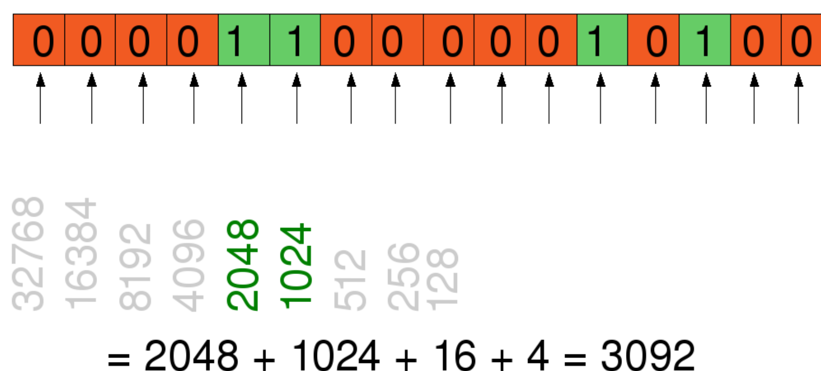


FIGURE 1.1 – test

Exercice 1 ()

Question a) Donner le nombre entier correspondant au code binaire suivant : 10101011

Question b) Coder sur 8 bits les entiers 205, 27 et 13

Les limites du codage des nombres entiers sur 16 bits sont donc :
– Valeur minimum : 0
– Valeur maximum : 131072

On pourra remarquer que plus la longueur du mot est importante, plus la limite supérieure des entiers codables dans l'ordinateur est haute. Si vous travaillez avec un ordinateur 16 bits, il pourra bien évidemment compter au delà de 131072, mais cela nécessitera pour lui plus de travail que si vous restez dans cette limite. Ensuite, pour des applications très bas niveau (celle où il est nécessaire de programmer des opérations sur des bits comme par exemple le développement de driver de périphériques), ces limites sont prégnantes.

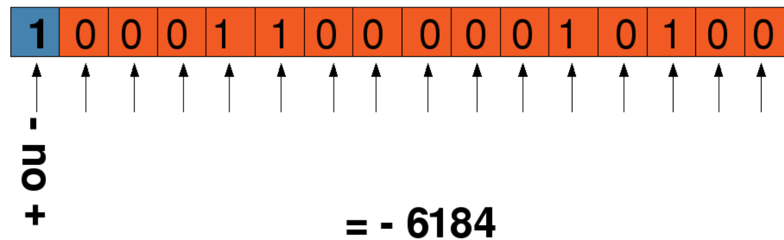
Lorsqu'un calcul numérique conduit à un résultat théorique qui dépasse la limite de codage d'un nombre on parle de "débordement". En langage C, rien n'interdit les débordements sur les nombres entiers, en pratique, la suite du programme continuera mais sera erronée. Par exemple, rien ne limite la réalisation de l'opération 131072+131072, mais la valeur calculée sera une valeur entre 0 et 131072 (elle dépendra du processeur ...).

1.1.3 Codage des nombres entiers relatifs

Pour les entiers relatifs, il faut être capable de coder le signe d'un nombre. Pour cela, on se sert du bit de poids fort (le plus à gauche dans un codage de type "little endian") pour coder la valeur d'un entier

relatif : **négatif si il est égal à 1 et positif sinon**.
Pour obtenir la valeur v du nombre entier, il faut sommer les valeurs des bits à 1. On note m_i la valeur (1 ou 0) du ième bit à partir de la droite dans le mot m.

$$v = (-1)^{m_{15}} \sum_{i=0}^{i=14} m_i.2^i$$



Exercice 2 ()

Question a) Donner le nombre entier relatif correspondant au code binaire suivant : `1010 1011`. Remarque par rapport à l'exercice 1 ?

Question b) Coder sur 8 bits les entiers relatifs suivants : 205, 27, -13 et -77. Remarques par rapport à l'exercice 2 ?

1.1.4 Codage des nombres à virgule flottante (simple précision)

Un nombre à virgule flottante est une représentation d'un nombre à virgule (*e.g.* 1,32). A la fin des années 70 chaque ordinateur avait sa propre représentation interne pour les nombres à virgule flottante. Or, l'arithmétique à virgule flottante possède certaines subtilités que le constructeur moyen ne maîtrisait pas forcément et certaines machines effectuaient certaines opérations de manière incorrecte. Pour remédier à cette situation, l'IEEE proposa un standard non seulement pour permettre les échanges de données en virgule flottante entre ordinateurs, mais aussi pour fournir un modèle rodé aux constructeurs, dont le fonctionnement était correct et maîtrisé. De nos jours, pratiquement tous les constructeurs ont des processeurs ou des coprocesseurs dédiés qui effectuent des calculs en virgule flottante et qui emploient la représentation au standard IEEE 754.
La norme les nombres à virgule flottante à *simple précision* (mot de 32 bits) et à *double précision* (mot de 64 bits).

Nombres à simple précision ou *float*

La représentation des nombres à virgule flottante simple précision en utilise une écriture sous la forme exponentielle suivante :

$$v = (-1)^S.m.2^{E-127}$$

Pour les float simple précision, les 32 bits sont organisés de la sorte (de droite à gauche) :
- 23 bits de mantisse (m) représentant la fraction du nombre. Les 23 bits code un nombre entiers M qui permet de réécrire la mantisse en notation décimale sous la forme "1,M"

– 8 bits d'exposant (E) dont la valeur entière est entre 0 et 128. Dans la formule, il faut noter la présence du "-127" de sorte à ce que l'exposant peut être négatif pour coder de très petits nombres.
– 1 bit de signe (S) (bit de poids fort)

Exemple 2

– $1 = 2^0 \times (1+0)$
Le bit de signe sera 0, l'exposant, en code relatif à 127 sera représenté par 127 = 01111111, et le significande vaut 1, ce qui résulte en une mantisse dont tous les bits sont à O. La représentation IEEE simple precision IEEE 754 du nombre 1 est donc :

$$Code(1) = \begin{array}{llll} 0011 & 1111 & 1000 & 0...0 \\ seee & eeee & emmm & m...m \end{array}$$

– $0.5 = 2^{-1} \times (1+0)$
Le bit de signe est 0, l'exposant, en code relatif à 127 est représenté par 127 - 1 = 01111110, et le significande vaut 1, ce qui résulte en une mantisse dont tous les bits sont à 0. La représentation IEEE simple precision IEEE 754 du nombre 0.5 est donc :

$$Code(0.5) = \begin{array}{llll} 0011 & 1111 & 0000 & 0...0 \\ seee & eeee & emmm & m...m \end{array}$$

– $1.5 = 2^0 \times (1+2^{-1})$
Le bit de signe est 0, l'exposant, en code relatif à 127 est représenté par 127 = 01111111, et le significande vaut 1.1, ce qui résulte en une mantisse dont le premier bit est à 1 et les 22 suivants à 0. La représentation IEEE simple precision IEEE 754 du nombre 1.5 est donc :

$$Code(1.5) = 0011 \quad 1111 \quad 1100 \quad 0...0$$

Certains codes sont conservés pour des valeurs particulières :

Nombre	signe	exposant	mantisse
Zéro	0/1	0	0
infini	0/1	FF	0
NaN (not a number)	0/1	FF	tout sauf 0

En arithmétique à virgule flottante on peut obtenir un résultat valable, ou alors rencontrer un problème de dépassement par valeur supérieure (*overflow*) lorsque le résultat est trop grand pour pouvoir être représenté, ou par valeur inférieure (*underflow*) lorsque le résultat est trop petit. Il peut même advenir une impossibilité de donner une valeur (*NaN*).
– **Dépassement par valeur inférieure** : Cette situation arrive lorsqu'un résultat est trop petit pour pouvoir être représenté. Le standard IEEE 754 résout partiellement le problème en autorisant dans ce cas une représentation dénormalisée. Une représentation dénormalisée est caractérisée par le fait d'avoir un code d'exposant complètement nul, ce qui est interprété comme une indication du fait que le bit de poids fort de la mantisse, implicite, est cette fois à 0 au lieu d'être à 1. De cette façon, le plus petit nombre "exprimable" est :

$$2^{-127} \times 2^{-23} = 2^{-150} \approx 10^{-45}$$

Cependant, il faut remarquer que plus le nombre représenté est petit, moins sa mantisse comportera de bits significatifs. Ce schéma permet une approche "douce" du phénomène de dépassement par valeur inférieure, en sacrifiant la précision lorsqu'un résultat est trop petit pour admettre une représentation normalisée.

– **Dépassement par valeurs supérieures** : Le dépassement par valeurs supérieures ne peut pas être traité comme le dépassement par valeurs inférieures, et est indiqué par un code d'exposant dont tous les bits sont à 1, suivi par une mantisse dont tous les bits sont à 0. Ceci est interprété comme représentant l'infini. L'infini peut être positif ou négatif, en fonction de la valeur du bit de signe. L'infini peut être utilisé dans les calculs et les résultats correspondent au sens commun :

$$inf + inf = inf$$
$$\forall x,\ x/inf = 0$$
$$\forall x,\ x/0 = inf$$

– **NaN : Not a Number** : Cependant, certaines opérations peuvent ne conduire à aucun résultat exprimable, comme

$$inf/inf = ?$$
$$0 \times inf = ?$$

Le résultat de telles opération est alors indiqué par un autre code spécial : le code d'exposant a tous les bits à 1, suivi par une mantisse non nulle. Le "nombre" correspondant est appelé **NaN** (Not a Number) : c'est un non-nombre. **NaN** doit être différencié de **NULL** dont il sera question de pointeurs.

Les caractéristiques des nombres à virgules flottantes simple précision sont les suivantes :

– Plus petit nombre normalisé : 2^{-126}
– Plus grand nombre normalisé : presque 2^{128}
– Intervalle utile : approximativement 10^{-38} à 10^{38}
– La précision des nombres flottants n'est pas uniforme (*cf.* Figure 1.2) : pour les nombres proches de zéro, il y a beaucoup de précision, et plus on s'éloigne de zéro plus on perd en précision moyenne.

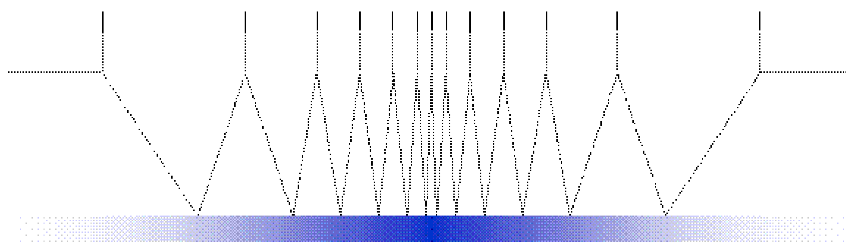


FIGURE 1.2 – Illustration de densité des nombres à virgule flottante (barres verticales en haut) dans IR (graduation continue en bas) (Source `http://www.macaulay.ac.uk/fearlus/floating-point/fpreal.gif`)

Exemple 3

Supposons que l'on cherche la valeur du float suivant exprimé en binaire :

11000010110101011010000000000000

On procède ainsi :

- *Le bit de signe est positionné : c'est un négatif.*
- *On a 10000101 = 133 qui est la valeur de l'exposant biaisé. 133 - 127 = 6 est la valeur de l'exposant vrai.*
- *La lecture de la mantisse donne (avec le bit sous-entendu égal à 1 en tête, puisqu'il s'agit bien d'un nombre normalisé) : 1,10101011010000000000000.*

L'exposant est 6, il faut donc décaler la mantisse de 6 colonnes vers la gauche.
Le nombre est donc égal à :

$$-1,1010101101 \times 2^6 = -1101010,1101 = -(2+8+32+64+0,5+0,25+0,0625)$$

$$= -106,8125$$

IEEE 754 double précision

Le principe de codage des nombres à virgule en double précision est exactement le même, mais on dispose cette fois de 64 bits pour coder le signe, l'exposant (sur 11 bits avec un codage relatif à 1023) et la mantisse (sur 52 bits). En accroissant fortement la taille de la mantisse, on gagne surtout en précision de calcul (et peu en plage de nombres).
Les caractéristiques des nombres à virgules flottantes double précision sont les suivantes :

- Plus petit nombre normalisé : 2^{-1022}
- Plus grand nombre normalisé : presque 2^{1024}
- Intervalle utile : approximativement 10^{-308} à 10^{308}

Definition 4 - Précision

La notion de précision renvoie à la capacité de différencier finement deux nombres. Plus un ordinateur est précis, plus les nombres qu'il sera capable de distinguer seront proches de 0, et moins il fera d'arrondies dans les calculs.

Avec un codage en simple précision, un ordinateur ne peut pas faire la différence entre 0 et 2^{-126}. Tous les nombres entre ces deux valeurs, par exemple 2^{-200} seront donc arrondies à l'une de ces valeurs. Par contre, en passant dans un codage double, les trois valeurs, 0, 2^{-126} et 2^{-200} seront bien distinguées. En passant en double, il y a un gain de précision.
Cette notion de précision est fondamentale dans les problèmes de calcul numérique où les arrondies dans les calculs s'accumulent. C'est le cas par exemple des calculs pour les prévisions météo ou bien les simulations numérique d'écoulement, de rayonnement, ... pour les quelques de nombreuses opérations numériques sont réalisées.

1.1.5 Représentation des textes

En informatique, un texte doit être vu comme une simple suite de caractères pour lequel il n'y a aucune mise en forme. Nous voyons donc d'abord la représentation des caractères, puis celle des textes.

Représentation des caractères

Le codage des caractères associe un nombre entier à un caractère contenu dans un alphabet. Le problème informatique du codage en mémoire des caractères est donc très simplifié puisqu'il va utiliser le codage des nombres entiers. La représentation de base pour un caractère est l'octet (8 bits).
Il reste néanmoins à faire la correspondance entre un nombre entier, codé en mémoire, et un caractère alphabétique. Ceci est possible grâce à des **tables de caractères**.
La table de correspondance la plus connue est la table ASCII. Cette table n'utilise que 127 valeurs numériques codées sur 7 bits. Le 8ème bit est nul. Schématiquement, la table ASCII est organisée comme suit :

- 0 à 31 : caractères non imprimables (saut de ligne, son de cloche, ...)
- 32 : espace
- puis, les chiffres arabes, les lettres latines majuscules puis les minuscules
- et enfin, les signes de ponctuation sont répartis dans la table

Autres tables usuelles :

- Latin-1 : basée sur la table ASCII, elle utilise les 127 caractères restant pour coder les caractères accentués et les caractères allemands.
- UTF-8 : ce codage permet de représenter les milliers de caractères du répertoire universel, il est compatible ASCII, et conçu pour son interopérabilité (communications facilité entre systèmes). Dans ce codage, la taille en mémoire d'un caractère varie entre 1 à 4 octets.
- Windows-1225 : une des nombreuses tables de caractères Windows (à éviter ...)
- tables pour les caractères mandarins, arabes, cyriliques, ...

Exercice 3 (Codage des caractères) *En consultant sur internet une table ASCII, répondre aux questions suivantes :*

Question a) à quels caractères ASCII correspondent les codes suivants

- 01110010
- 10010110
- 00110110

Question b) Pour les codes précédents, quelle(s) différence(s) y aurait-il avec un table "Latin-1" ? avec une table "cyrilique" ?

Question c) Représenter en binaire les caractères suivants : "f", "EOF" (ou "EOT") et " ?"

Représentation des chaînes de caractères, les textes

Lorsqu'on pense à un texte, un informaticien parle de chaîne de caractères.

Definition 5 - Chaîne de caractères

Une chaîne de caractères est une suite ordonnée de caractères se terminant par un caractère `nul`. En anglais, on emploie le terme *string*.

Dans la mémoire de l'ordinateurs, une chaîne de caractères est un espace contigüe de mots mémoires qui est organisé dans l'ordre des caractères dans la chaîne. À partir du moment où vous savez qu'un

caractère ASCII est codé sur un mot de 8 bits, le texte en mémoire est simplement une suite de mots de 8 bits. Et lorsque vous tombez sur un mot contenant 8 zéros (le caractère `nul`), alors vous arrêter le lire la chaîne de bits.
On peut alors représenter une chaîne de caractères comme une suite de bits, mais qui s'étend beaucoup plus largement en mémoire. En pratique, cette représentation est sans intérêt, nous n'y reviendrons jamais.

Exemple 4

En pratique, la chaîne de caractères "toto" est codée par 5 mots : 4 pour le texte et 1 pour le caractères `nul`*. Avec des mots de 8 bits, une telle chaîne de caractère prendra donc* $5 \times 8 = 40$ *bits.*
En consultant la table ASCII, on sait que 'o' se code 01101111 *et que 't' se code* 01110100*, on a alors en mémoire :*

0111010001101111011101000110111100000000

Pour un fichier texte simple (c'est-à-dire non mis en forme avec word ou autre logiciel de traitement de texte), c'est la même chose. Le fichier est enregistré comme une suite de caractères codés sur des mots de 8 bits. En revanche, le caractère de fin du fichier n'est pas le caractère `nul`, mais le caractères `EOF` (ou `EOT`).

Exercice 4 () *Extraite le texte de la chaîne de caractères ASCII suivante :*

011010110110000101100111011011100010000100000000101001

1.1.6 Remarque sur la sémantique du binaire

Le binaire n'est qu'un codage d'une information structurée. Si vous ne connaissez pas à l'avance la structure de l'information, il est très difficile d'extraire du sens d'une suite de bits.
C'est le sens d'un format de fichier : il définit la façon dont est codé l'information dans un fichier. Si vous connaissez le type de fichier auquel vous avez à faire, il vous sera possible d'en extraire l'information.

1.2 Architecture de von Neumann

L'architecture de von Neumann désigne un modèle de fonctionnement d'un micro-processeur (et *a fortiori* d'un ordinateur). Cette architecture a été proposée en 1936. Elle était très innovante par son principe de représenter en mémoire à la fois les données traitées par l'ordinateur et les programmes, c'est à dire la description des traitements. Jusque là, programmes et données étaient des objets différents. Les travaux théoriques de A. Turing ont permis cette avancée conceptuelle et les travaux de von Neumann (et d'autres également) ont permis une concrétisation de ces idées.
Tout d'abord, il va nous falloir poser quelques définitions.

Definition 6 - Programme

Un programme est une *suite d'instructions* qui traite des données en entrée et produit des données en sortie.

Definition 7 - Instruction

Une instruction est une *action élémentaire* que peut réaliser un processeur.

Definition 8 - Processus

C'est *une* exécution d'un programme sur un processeur.

Pour faire une image culinaire, un processus est à un programme ce que la préparation d'un plat est à une recette.
Tout le travail d'un programmeur est de bien connaître les instructions qui peuvent être faites par une machine pour construire une suite d'instructions qui vont faire réaliser au processeur la tâche qu'il s'est fixée.
On dépeind ici l'architecture d'un ordinateur de manière très simplifiée. Les principes sont présentés en laissant volontairement de côté bon nombre de détails et en sous-entendant que tous les processeurs suivent ces fonctionnements. Des variantes importantes peuvent être rencontrées en pratique. Pour plus d'information, il faudra se référer aux cours de systèmes et réseaux.

1.2.1 Présentation de l'architecture de von Neumann

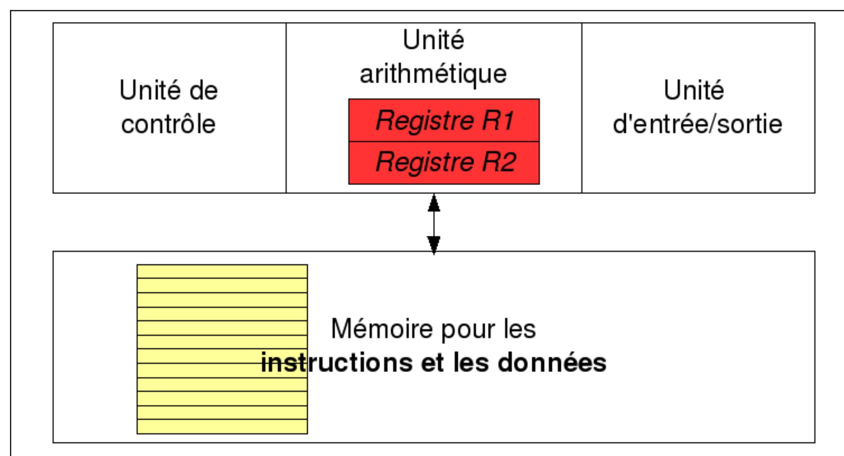
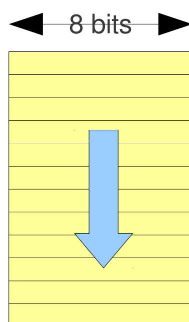


FIGURE 1.3 – Illustration de l'architecture de von Neumann

L'architecture de von Neumann décompose l'ordinateur en 4 parties distinctes :
- l'**unité arithmétique et logique** (UAL ou ALU en anglais) ou unité de traitement : son rôle est d'effectuer les opérations de base : addition, soustraction, ... ;
- l'**unité de contrôle**, chargée du séquençage des opérations ;
- la **mémoire** qui contient à la fois les données et le programme qui dira à l'unité de contrôle quels calculs faire sur ces données ;
- les **dispositifs d'entrée-sortie**, qui permettent de communiquer avec le monde extérieur.

Les registres sont des petites mémoires sur lesquelles l'ALU applique ses opérations de base. Lorsque l'ALU fait une addition, il fait l'addition des nombres (codés en binaire) contenus dans le registre 1 et le registre 2. Le résultat de l'addition est retourné dans l'un de ces deux registres.

1.2.2 Image de l'organisation de la mémoire



Je commence par préciser que la mémoire dont il est question dans l'architecture de Von Neumann est la mémoire RAM. Il ne s'agit pas du disque dur dont il n'est pas question ici.
La mémoire peut être représentée comme une longue bande orientée d'emplacements mémoire. À une granularité fine, on peut voir la mémoire comme une longue bande d'emplacement pouvant contenir des bits (0 ou 1), à une granularité plus importante, on peut la voir comme une langue bande d'emplacement de mots mémoires. La taille des mots mémoires utilisé est une caractéristique du processeur utilisé : c'est la taille des registres qui définit la taille des mots d'un processeur.
Cette bande est orientée. L'orientation impose un sens de lecture par le processeur. Par conséquent, chaque emplacement mémoire peut être identifié par sa position dans la bande. On parle d'adresse mémoire.

Definition 9 - Adresse mémoire

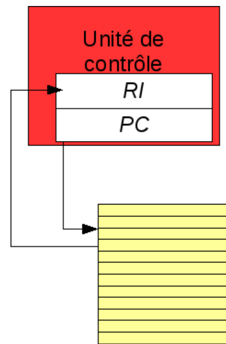
C'est un nombre qui localise un mot dans la bande mémoire.

Prenons l'exemple d'un processeur 8 bits. On représente sa mémoire comme une grande bande de largeur 8 bits. Chaque emplacement peut contenir :
– une donnée codée sur 8 bits parmi les types de base : un nombre entier, un nombre en virgule flottante ou un caractère.
– la description d'une instruction
Les instructions et les données ne sont pas mélangées de manière anarchique. Les données sont regroupées ensembles (par exemple, pour les chaînes de caractères on a vu qu'un texte était une suite contigüe de mots de 8 bits). De même, dans la mesure où les programmes sont des suites d'instructions, la séquence des instructions est traduite par la suite des instructions dans la bande.

1.2.3 Image du fonctionnement d'un processeur

Descriptif général du fonctionnement d'un processeur

Décrire le fonctionnement d'un processeur, c'est décrire comment le processeur fait pour exécuter un processus à partir de la description d'un programme en mémoire.
L'unité de contrôle est au centre de l'exécution d'un processus par le processeur. il nous faut donc donner quelques détails sur sa constitution. L'unité de contrôle contient 2 registres :
– le registre PC (program counter) : il contient l'adresse de l'instruction courante
– le registre RI (registre d'instruction) : il contient le code binaire de l'instruction à faire réaliser par l'ALU



À un instant donné de l'exécution du processus, PC pointe sur un emplacement mémoire. Pour réaliser l'instruction suivante, les opérations suivantes sont effectuées par le processeur :

1. PC est incrémenté (*i.e.* qu'il passe à l'emplacement suivant dans la bande mémoire)
2. Le contenu de l'emplacement pointé par PC est recopié dans RI
3. L'instruction est décodée par l'unité de contrôle et ce dernier prépare l'ALU
4. L'ALU effectue l'instruction, par exemple :
   - Les registres R1 et R2 sont chargés à partir de mots en mémoires
   - Une opération arithmétique ou logique traite les registres
   - Le contenu d'un registre est copié dans un emplacement mémoire
   - L'ALU effectue l'instruction, par exemple :
5. Retour à 1.

L'ensemble de ces 5 étapes constitue un cycle du processeur. Un processeur cadencé à 1GHz réalisera 1 million de cycle de la sorte par seconde.

Codage du jeu d'instructions

Definition 10 - Jeu d'instructions

Le jeu d'instructions est le descriptif de toutes les instructions élémentaires que peut réaliser un processeur et le codage binaire de ces instructions.

Concrètement, chaque instruction est câblée électroniquement sur la puce électronique. Le jeu d'instructions est donc spécifique d'un processeur. Lorsqu'une instruction est câblée électroniquement elle sera réalisée très rapidement. Ceci explique pourquoi il y a des processeurs spécifiques pour certaines tâches. Par exemple, si les processeurs des cartes graphiques sont réputées pour leur efficacité sur les opérations sur des matrices, c'est que les opérations usuelles sur les matrices sont câblées électroniquement. Alors que sur un processeur généraliste (sans opération matricielle dans le jeu d'instructions) il faudra tout coder à partir d'opérations sur des scalaires. Les résultats seront médiocres par rapport à la puce électronique spécialisée.
En général, un processeur sait faire les opérations suivantes :
- Consulter ou modifier un état de la mémoire ou des registres
- Déclencher une opération entrée-sortie (communication avec l'extérieur, en particulier l'utilisateur humain)
- Modifier la séquence d'instructions, par exemple en sautant à différents endroits du programme : en avant (pour les conditions), en arrière (pour les boucles)

– Faire des opérations arithmétiques et logiques : additions, soustraction, incrémentation, ...

Le jeu d'instructions est donc un catalogue d'opérations élémentaires. Mais en plus, c'est un codage des instructions. Par exemple, pour le code instruction binaire 10011101, le processeur réalise l'addition des deux registres. Pour certaines instructions, une partie du mot de l'instruction est réservée à un paramètre (par exemple une adresse mémoire).

Lorsqu'un programme est écrit en utilisant des codes instructions spécifique à un processeur, on dit qu'il s'agit d'un programme en langage assembleur.

Definition 11 - Langage assembleur

Le langage assembleur est la "langue maternelle" d'un processeur. Un programme écrit en langage assembleur est spécifique à un processeur. On parle parfois de "code machine". On parle également de "langage machine".
Un fichier écris en assembleur est un **programme exécutable** par un processeur.

Exemple de jeu d'instructions et de programme assembleur

Voici une liste réduite d'instructions élémentaires pour un processeur 16 bits (inspirée d'un processeur Intel8080)

Chaque instruction est codée sur 16 bits. Les 5 premiers bits codent l'opération, le 6eme et 7eme bit désigne l'utilisation du registre 1 ou 2, et les 9 bits restant servent à coder une adresse mémoire pouvant être utilisé dans une opération.

Le jeu d'instructions comporte 6 opérations :

Instruction	Code opération	Nom mnémonique
Chargement Registre->Mémoire	00001	`LOAD`
Copie Registre->Mémoire	00010	`STORE`
Addition mémoire/registre à registre	00011	`ADD`
Afficher le contenu de la mémoire	00100	`PRINT`
Test le registre et branchement	00101	`ADD`
Fin	00101	`END`

Exemple de mot mémoire codant une instruction :

- 0000110000010100 : `LOAD R1, @(0 0001 0100)` Chargement dans le registre 1 (`R1` du contenu de l'emplacement mémoire 20, écrit en mnémonic
- 0001101000010100 : `ADD R2, @(0 0001 0100)` Additionne `R2` et le contenu de l'adresse 20 (place le résultat dans le registre `R2`)
- 0001111000010100 : `ADD R1,R2` Additionne `R1` et `R2` (place le résultat dans `R1`). L'adresse ne sert ici à rien !
- 001011000000110 : `BZ R1, @(0 0001 0100)` Test si `R1` est nul, si oui, mettre `PC` à 6.

Exemple 5 - Exemple de programme (assembleur)

```
LOAD R1, @20
BZ R1, FIN
ADD R1, @(21)
```

```
     STORE R1, @(22)
     PRINT @(22)
FIN  END
```

Les explications précédentes ne permettent certainement pas de comprendre entièrement le programme. Il s'agit juste d'une illustration d'un programme assembleur. En particulier, les expressions entre parenthèses désigne des adressages indirectes (notion non-présentée, mais indispensable pour compléter correctement un jeux d'instructions). Voici néanmoins une description de ce que fait ce programme :
La première instruction charge le contenu de l'adresse mémoire dans le registre R1. À la ligne suivante, il s'agit d'un branchement conditionnel, si R1 est non nul, alors le programme passe directement à la la dernière ligne signalant la fin du programme. Dans le cas contraire, le processeur exécute les trois dernières instructions. D'abord, on demande au processeur d'ajouter le contenu du registre R1 avec le contenu de l'adresse 21 de la mémoire. La description des instructions devrait préciser où est mis le résultat de cette opération, par exemple dans le registre R1. Le contenu du registre est ensuite sauvegardé dans l'emplacement 22 de la mémoire. Finalement, le contenu de cette adresse est affiché.

1.3 Compilation

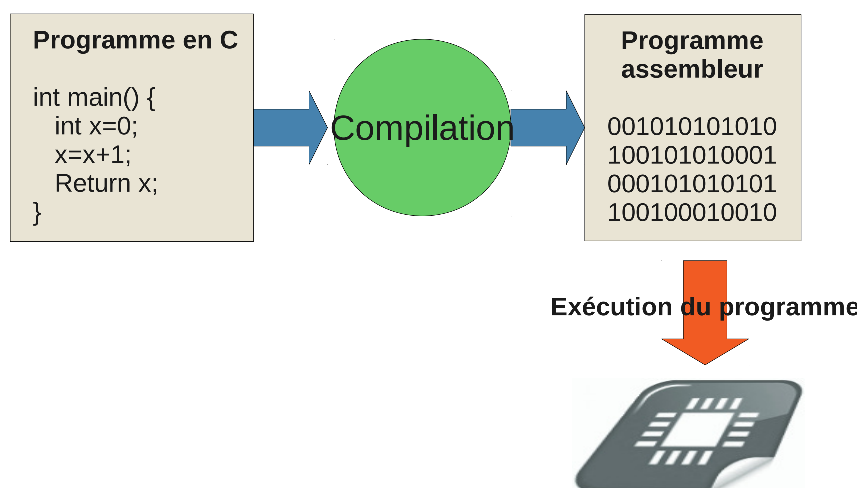
Il n'est pas raisonnable de programmer ses programmes en assembleur :

- Un programme en assembleur n'est pas tendre avec le cerveau du programmeur, même chevronné, puisqu'il nécessite d'avoir en permanence une vision de l'ensemble du code ;
- Programmer en assembleur est dangereux ... personne n'est à l'abri d'erreurs, et les erreurs en assembleurs sont difficiles à corriger et aux conséquences souvent inattendues ;
- Le temps nécessaire pour programmer une application, même simple serait démentiel ;
- Tout les efforts de programmation devraient être démultipliés pour porter les applications sur tous les processeurs du marché !

Il est donc nécessaire d'utiliser un langage de programmation plus évolué qui permette de résoudre tous ces problèmes. On parle alors de langage de "haut niveau". Plus le langage sera proche de la pensée du programmeur et s'éloignera des couches matérielles, plus il sera de haut niveau. Par exemple, des langages comme Java, python ou R sont de très haut niveau : les instructions qui sont programmées sont très loin de la réalité matérielle du processeur. Le langage C est un langage intermédiaire : il résout tous les problèmes sus mentionnés, mais reste un langage de programmation basique (ne compter pas trouver une fonction qui calcule une ACP ! Si vous voulez calculer une ACP, il faudra programmer l'algorithme où utiliser l'implémentation d'un collègue).
Parmi les langages de programmation, on distingue trois types de langage : les langages machine, *i.e.* l'assembleur, *cf.* définition 1.2.3 (le seul que la machine comprend), les langages compilés et les langages interprétés.

Definition 12 - Langage compilé

Un langage compilé est un langage de programmation pour lequel une architecture matérielle dispose d'un **compilateur** qui peut traduire un programme du langage compilé dans le langage machine. Le programme traduit par le compilateur peut alors être exécuté sur la machine.
Le résultat de la compilation est un programme



Definition 13 - Langage interprété

Un langage interprété est un langage de programmation pour lequel une architecture matérielle dispose d'un programme capable d'interpréter "en direct" le programme écrit en langage interprété. L'interpréteur agit comme une couche intermédiaire entre le langage machine et le langage interprété.

Le langage C est un langage compilé (*cf.* Figure 1.3).
La compilation va bientôt devenir votre quotidien ... vous programmez, vous compilez et vous exécutez ! Voilà ce qui vous attend ... Alors autant bien s'entendre avec son compilateur si on veut éviter la crise de nerfs. Et pour bien s'entendre avec lui, il est intéressant de mieux comprendre comment il fonctionne.
Il existe de très nombreux compilateurs : sous Windows, on peut citer par exemple **Visual C++** (Microsoft), **C++ Builder** (Borland), et sous Linux vous disposez des compilateurs `gcc` ou `g++` qui sont d'excellents compilateurs libres. Dans le cadre de ce cours, on n'utilisera que de g++ (qui peut également être utilisé sous Windows). Tous fonctionnent sur les mêmes principes.

Pour plus d'informations sur la compilation d'un programme en C, voir le chapitre 3.

1.4 Correction des exercices

Solution à l'exercice 1

Question a) *171*

Question b) *11001101, 00011011, 00001101*

Solution à l'exercice 2

Question a) *-43, c'est pas pareil à cause du bit de poid fort qui est utilisé comme bit de signe.*

Question b) *impossible de coder 205 !, 00011011, 10001101, 11001101*

Solution à l'exercice 3

Question a) *Traduire les valeurs binaire en nombre et regarder la table ASCII pour faire la correspondance :*
- 01110010 -> 114 -> 'r'
- 10010110 -> 150 -> hors table ascii !
- 00110110 -> 54 -> '6' (caractère '6' !!)

Question b) *Pas de changement pour le premier et le troisième, par contre le second s'interprète différement dans une table Latin-1 ou Cyrilique ... voir les tables !*

Question c) *Démarche inverse ...*
- 'f' -> 102 -> 01100110
- 'EOF' -> 4 -> 00000100
- '?' -> 63 -> 00111111

Solution à l'exercice 4 On segmente cette chaîne binaire en mot de 8 bits, et on traduit un à un :

1. 01101011 -> 107 -> 'G'
2. 01100001 -> 97 -> 'a'
3. 01100111 -> 103 -> 'g'
4. 01101110 -> 110 -> 'n'
5. 01100101 -> 101 -> 'e'
6. 00100001 -> 33 -> '!'
7. 00000000 -> caractère `nul` : fin de la chaîne
8. 00101001 -> peu importe !!!

La solution est donc "Gagne !".

CHAPITRE

2 Environnement de programmation

L'environnement de programmation désigne l'ensemble des outils qui sont à disposition pour permettre à un développeur de mener à bien sa tâche. Je distingue quatre types d'outils :

- le compilateur : c'est-à-dire l'outil fournit par le système d'exploitation pour compiler vos programmes C en programmes exécutables sur votre ordinateur (*cf.* chapitre 3),
- les éditeurs : c'est-à-dire les outils dans lesquels écrire vos programmes en C. Il est existe des très basiques jusqu'à des très avancés (appelé IDE pour *Interactive Development Environment*),
- les outils de débuggage/profilage : c'est-à-dire des outils qui vont vous permettre d'analyser votre code soit pour le corriger soit pour l'améliorer,
- les outils annexes : ceux qui n'ont pas de lien direct avec la programmation mais qui peuvent être utiles ...

Il se trouve que le langage C est le langage utilisé pour le développement du système d'exploitation Linux. Il est par conséquent plus facile de développer dans ce langage dans cet environnement. Je commencerai donc par présenter les outils du système Linux. Ensuite, je reviens sur les environnements de programmation que vous pourrez installer sur les autres systèmes d'exploitation usuels (Windows ou Mac). N'étant pas pas familier de ces derniers, les informations seront moins fournies.

2.1 Programmer en environnement Linux (sans IDE)

Avant de vous aventurer dans le développement sous Linux, il est plus pratique de savoir se servir d'un terminal de commandes.

2.1.1 Éditeurs

Tout éditeur de texte peut servir à écrire un programme C puisqu'un programme ce n'est qu'un texte brut (avec un syntaxe bien particulière). Néanmoins, rien ne sert de se compliquer la vie à utiliser des éditeurs tels que `vi` [1].

Les propriétés attendues d'un éditeur de texte pour la programmation sont les suivantes :

- la **coloration syntaxique** : INDISPENSABLE ! elle permet de mettre en valeur par des couleurs spécifiques, à la fois, les mots clés et syntaxes particulières. Certaines colorations permettent également de mettre en valeurs des erreurs de grammaire (oublis d'un `{` par exemple). La coloration syntaxique permet au programmeur de faciliter la lecture d'un programme et souvent de repérer des erreurs de frappes etc.
- l'**auto-indentation** : cette propriété permet de demander à l'éditeur d'indenter son programme. Cette fonctionnalité est pratique pour faciliter la vie des jeunes programmeurs (car rien ne vaut une bonne indentation dès la programmation). Certaines auto-identation se font à la frappe (le curseur se place à la bonne position lors d'une nouvelle ligne) et d'autres se font à posteriori.

1. `vi` est l'éditeur de texte le plus basique que je connaisse mais vraiment trop peu intuitif pour être utilisé ...

- la **complétion** : cette propriété offre la possibilité au programmeur de faire afficher toutes la variables ou fonctions disponibles qui commence par les premières lettres d'un mot déjà tapées. Cette fonctionnalité s'adresse soit aux bons programmeurs qui ont appris à se servir de cette fonctionnalité pour être plus efficaces, soit aux débutants qui s'en servent comme d'un aide mémoire.

Quelques éditeurs intéressants disponibles sur la plupart des systèmes d'exploitation Linux :

- `scite`
- `gedit`
- `kile`
- `emacs` (usage un peu particulier pour un débutant)

Tous ces éditeurs sont en fait des éditeurs généralistes : ils sont capables de colorer syntaxiquement différents langages de programmation. Ils “savent” quel langage utiliser à partir de l'extension du fichier que vous ouvrez.

2.1.2 Compilateurs `gcc`, `g++`

Les compilateurs sous Linux se fait au travers de la console de commande (ou terminal de commande).

Vous pouvez lancer un terminal à l'aide du menu `Applications>Outils Systèmes>Terminal`. Dans la mesure où vous aurez souvent besoin des terminaux, il est intéressant de vous faire un raccourcis sur le bureau pour les lancer plus rapidement. Pour cela, glissez l'icone du menu sur le bureau. Au lancement, le répertoire courant du terminal correspond à la racine de votre espace personnel. Vous pouvez vous en rendre compte en listant le contenu de ce répertoire (commande `ls`). Vous devriez y trouver les documents accessibles sous Windows par le volume `Z:`.

Pour vous déplacer dans vos répertoires, il faut utiliser la commande `cd`. Rendez-vous alors dans le répertoire où se trouve le fichier que vous voulez compiler.

Compilation avec `gcc`

La commande de compilation la plus simple pour un fichier `source.c` (sans dépendance et contenant un **main**) est de la forme :

```
$ gcc -o source source.c
```

ou

```
$ gcc source.c -o source
```

Cette commande va “construire” un nouveau fichier `source` (le nom du fichier généré par le compilateur est donné après l'argument `-o`) que vous pourrez exécuter ainsi pour tester votre programme :

```
$ ./source
```

Pour plus d'informations sur la commande `gcc` ou `g++`, voir le chapitre 3 sur les détails de la procédure de compilation.

Le compilateur `g++` est un compilateur pour `C++`. Il permet la compilation également des programmes `C` mais parfois avec de légère différence sur l'acceptation des syntaxes.

Automatiser la compilation avec `make`

Lorsque les programmes comprennent un grand nombre de fichier, l'étape de compilation est assez délicate. On ne peut pas prendre son temps à compiler tous les fichiers à chaque fois qu'une petite modification est réalisée, mais il faut néanmoins tenir compte des dépendances existantes entre les sources pour que les fichiers qui sont influencés par la modification soient recompilés. Bref, c'est un casse tête et tout cela peut s'automatiser avec un outil magique qui s'appelle `make`.
Un `Makefile` est un fichier qui va contenir un descritif des instructions de compilation des fichiers d'un programme et des dépendances entre ces fichiers. Ainsi, en se référant aux dates de dernière modification d'un fichier source, la commande `make` qui utilise le `Makefile`, va savoir quels sont les fichiers à recompiler, et comment le faire.
L'étape de création d'un `Makefile` prend quelques minutes, mais ensuite le gain de temps pour chaque compilation est considérable.

Les détails sur la création d'un `Makefile` sont donnés dans la section 3.5 de ce poly.

2.1.3 Débuggage avec `gdb`

L'une des difficultés de la programmation est que la conception du code est faite en amont de son exécution. Une fois que vous avez créer votre code, vous compilez puis lancez votre programme sans "voir" les détails de ce qu'il a fait.
Le débuggage ou débogage est la tâche qui consiste à diagnostiquer et réparer une erreur dans un programme, un "bug" ! Un bug (de l'anglais bug, "insecte" [2]) est, en informatique, un défaut de logique dans un programme informatique qui provoque un calcul ou un traitement incorrect à l'origine de dysfonctionnements et de pannes.
L'erreur peut provenir d'une erreur de conception du programme ou d'une erreur dans la traduction du programme dans le langage choisi (typiquement, une erreur de frappe).
Pour identifier un bug, il est plus facile de faire **exécuter le programme pas-à-pas**. C'est à dire qu'on veut pouvoir maîtriser la succession des instructions pour s'attarder sur chaque instruction de votre code. On fait exécuter une ligne de code, puis on regarde ce qu'il s'est passé, puis une autre ligne de code, etc.
Il est possible de faire cela grâce à des outils de débuggage.

Présentation succincte de `gdb`

L'outil `gdb` [3] est un programme de débuggage pour les programmes en C. Il est puissant, mais l'interface est totalement en ligne de commande, c'est-à-dire avec une invite en texte et toujours au travers du terminal de commande. Dans certains IDE, `gdb` est interfacé pour faciliter son utilisation.

Si votre programme fait une erreur à l'exécution et affiche un très peu explicite `segmentation fault`, vous devez trouver d'où vient l'erreur. Pour cela, il faut :

1. Recompiler votre programme avec l'option `-g`,
2. Exécuter de nouveau votre programme avec `gdb`.

Pour obtenir des informations de trace sur votre programme `titi.c` avec le debugger, vous devez le compiler avec l'option `-g` ainsi :

```
$ gcc -g titi.c -o titi
```

2. Pour l'histoire, il semble que ce terme est apparue à cause d'un problème survenu dans un programme à cause d'un insecte collé sur une carte perforée

3. `gdb` est l'acronyme de Gnu DeBugger

L'utilisation de cette option `-g` va demander au compilateur d'ajouter des informations dans le code qui sont utilisées pour tracer le déroulement du programme.
Vous pouvez alors utiliser le debugger de la manière suivante :

```
$ gdb titi
```

`gdb` charge tout ce dont il a besoin... et présente une invite `(gdb)`.
Une nouvelle invite de commande se présente à vous :

```
(gdb)
```

Pour exécuter le programme, taper `run` puis valider votre commande avec la touche Entrer. Le programme devrait se lancer et s'arrêter exactement à l'endroit de votre erreur. La ligne de votre programme sur laquelle s'est arrêter l'exécution est affiché par `gdb`.

Une situation pratique

Je peux mettre un "breakpoint" (un point d'arrêt dans le programme) dans la fonction critique de mon programme. Disons que cette fonction s'appelle `tutu`. je tape sous `gdb` :

```
(gdb) b tutu
```

`b` veut dire `breakpoint`. `gdb` me répond :

```
Breakpoint 1 at 0x80de41c: file titi.c, line 1309.
```

Je peux spécifier plutôt un arrêt à une ligne (ex : 134) de mon programme :

```
(gdb) b titi.c:134
```

Maintenent je lance mon programme sous `gdb` :

```
(gdb) run
```

On peut aussi taper `run bibi` pour donner en argument à `titi` la chaine '`bibi`' (ex : nom de fichier).
`gdb` me répond :

```
Starting program: /home/benoit/src/titi bibi
```

Pouf, le programme stoppe sur le `breakpoint`. On peut alors maîtriser l'exécution du programme pas-à-pas.
Par exemple, on avance d'une ligne dans le programme avec la commande `n` ou `next` et pour entrer dans les sous-fonctions, on fait un `step` ou `s`. Notez qu'il n'est pas possible d'allez en arrière dans l'exécution du programme. Si finalement, vous vouliez regarder ce qui c'est passé un peu avant, alors il faut arrêter l'exécution en cours (avec, par exemple, un `Ctrl+C`) et placer un `breakpoint` un peu plus haut dans le programme et relancer le débuggage.

```
(gdb) n
```

`gdb` me répond en m'affichant la ligne du programme en cours [4] :

4. Attention, il faut que votre fichier de programme soit effectivement celui du programme compilé que vous débugger, sinon il risque que les lignes ne vont correspondre autre chose que ce qui se passe

```
1350 if (Array1[i]!=i)
```

Et en plus, il est possible de connaître les valeurs des variables en cours d'utilisation. Disons que je veux vérifier la valeur de `i`, je fais alors :

```
(gdb) print i
```

gdb me répond : `$1 = 6`. Le 1 veut dire que c'est la première fois que j'invoque `i` ...qui est égal à 6. Il est également possible d'utiliser la commande `disp`, qui permet d'afficher la valeur de la variable à chaque pas (sans qu'on en demande l'affichage explicite).
Bon, continuons l'exécution avec la commande `continue` ou `c`.

```
(gdb) c
Continuing.
```

On peut interrompre l'exécution pour retrouver l'invite en tapant `Ctrl-c`. Sinon le programme continue son exécution jusqu'à la fin de celui-ci, ou jusqu'à tomber sur un nouveau `breakpoint` ou bien ...
Et hop ! plaf ! boum !, alors que tout ce passait bien l'erreur apparaît ! `gdb` s'arrête et nous invite à réfléchir... Tout d'abord, il faut localiser l'endroit du programme où l'exécution s'est interrompue. Pour cela, On peut taper `where` c'est-à-dire "où suis-je dans le programme ?". Le plus efficace est d'afficher la pile des appels des fonctions pour mieux zoomer sur l'endroit de l'erreur. On peut ainsi savoir comment on est arrivé à l'endroit actuel du programme :

```
(gdb) bt
```

`bt` veut dire `backtrace`. De là, on peut monter ou descendre dans les appels de fonctions :
- `up` permet de monter dans le contexte de la fonction appelante.
- `down` permet de descendre dans vers la fonction appelée.

Cela nous permet donc de se trouver dans le contexte de n'importe quelle fonction liée à l'erreur. On peut imprimer les variables que l'on veut dans le contexte courant avec la commande `print` ou `disp`. En comparant les valeurs contenues dans les variables et les valeurs théoriquement attendues, vous pouvez ainsi trouver l'erreur.
Pour quitter `gdb`, il suffit de taper `q`, et valider par Enter. `gdb` a de nombreuses commandes. pour en savoir plus, il suffit de taper `h` ou `help` et il vous propose une liste de thèmes. Tapez alors :

```
(gdb) h 'le thème voulu'
```

Remarque 2 - De l'art du débuggage

Le débuggage est un art, il nécessite de l'intuition et une très bonne connaissance de la programmation et du langage. Seule l'expérience de la programmation vous rendra efficace dans cette tâche.
Il est beaucoup plus facile d'apprendre à programmer qu'à débugger ... alors essayer de faire le moins d'erreur de programmation !

Pour devenir un pro de `gdb`, visitez le site web de référence de gdb . Là vous saurez tout sur `gdb` ! Voilà... Mais surtout n'oubliez jamais de compiler avec l'option de debuggage... `-g` pour `gcc`.

2.1.4 Debuggage avec `valgrind`

`valgrind` est un outil de profilage et de débuggage mémoire. Il permet de détecter des problèmes de gestion mémoire dans des programmes écrits en `C/C++`.
Une des premiers problèmes est la fuite de mémoire ... Une fuite mémoire est un problème qui survient lorsque le programme alloue de la mémoire et oublie de la libérer quand on n'en n'a plus besoin, voire se retrouver à ne plus être en mesure de la libérer (voir la partie du cours sur l'allocation mémoire pour comprendre le problème).

Vous pouvez consulter la page suivante pour en savoir plus : http ://www.unixgarden.com.

2.1.5 Profilage avec `gprof`

La commande `gprof` produit un compte-rendu d'exécution pour les programmes `C`. Pour utiliser la commande `gprof` il faut compiler le programme avec l'option de profilage `-pg`.

- À la fin de l'exécution, on obtient un fichier de données, `gmon.out`, que `gprof` peut analyser. On obtient alors deux compte-rendus : l'arbre d'appels (*call-graph profile*) : il contient la liste des routines en ordre décroissant du temps cpu consommé ainsi que les routines qui sont appeleées à partir d'elles. On peut ainsi connaitre quelles sont les routines mères qui appellent le plus souvent une routine donnée et quelles sont les routines filles qui sont appelées le plus fréquemment par une routine particulière.
- le profil plat (*flat profile*) de l'usage CPU : la liste des routines avec le nombre d'appels.

Les options pratiques de `gprof` sont :

- `-b` : supprime l'en-tête en cas de traitement ultérieur des sorties ;
- `-s` : génère un fichier résumé, appelé `gmon.sum` ;
- `-z` : affiche tous les symboles, même ceux qui n'ont eu aucun appel ou bien un temps total nul.

Procédure pour profiler votre code :

1. compilation et édition des liens avec l'option de profilage `-pg` ;
2. exécution classique ;
3. analyse du fichier de profilage en utilisation la commande `gprof`.

La première sortie de `gprof` est un classement décroissant (du temps cpu consommé) des fonctions appelées dans le programme : fonctions appartenant au programme et fonctions provenant de librairies du système. Par exemple l'extrait suivant :

```
% time  cumulative  self     calls  self     total    name
        seconds     seconds         ms/call  ms/call
36.9    92.23       92.23    87253  1.06     1.06     .saxpy [4]
29.9    167.06      74.83    45751  1.64     1.64     .pmv [5]
22.0    222.15      55.09    61502  0.90     0.90     .prodscal [6]
6.1     237.48      15.33    10000  1.53     3.17     .scdmb [7]
5.1     250.19      12.71    10000  1.27     21.85    .gradconj [3]
0.0     250.22      0.03                              .__mcount [8]
0.0     250.25      0.03                              .qincrement [9]
0.0     250.27      0.02     1      20.00    20.00    .fctfx [10]
0.0     250.28      0.01     6      1.67     1.67     .nrmerr [11]
0.0     250.28      0.00     20000  0.00     0.00     .fctft [12]
0.0     250.28      0.00     252    0.00     0.00     ._sigsetmask [13]
0.0     250.28      0.00     170    0.00     0.00     .pthread_mutex_lock [14]
```

- La colonne `% time` indique le pourcentage du temps cpu total consommé par la routine courante.

- La colonne `cumulative seconds` représente la somme partielle des temps cpu du haut de la liste jusqu'à la routine courante.
- La colonne `self seconds` représente le temps cpu de la routine courante.
- La colonne `calls` indique le nombre d'appels de la routine courante.
- La colonne `self ms/call` indique le temps cpu moyen (en milisecondes) consommé par appel à la routine courante, temps exclusif (ne contient pas la consommation des routines qu'elle appelle) uniquement.
- La colonne `total ms/call` indique le temps total (inclusif + exclusif) consommé par appel à la routine courante.

Dans le cas de la fonction `pmv`, il y a 45751 appels à cette fonction pour un total de 74.83 secondes d'utilisation du cpu, soit 1.64ms par appel de fonction.

2.2 Programmer avec `Code::Blocks`

2.2.1 Les IDE pour le développement en C

Il existe plusieurs IDE pour la programmation en C/C++. Généralement, ces outils ne sont pas spécifiques à C, mais plutôt à C++. Néanmoins, ils nous conviennent parfaitement. Les plus connus sont :

- `VisualC++` : l'environnement de développement en C/C++ de Microsoft et totalement dédié à Windows. Il embarque un compilateur spécifique à Windows. C'est un outil très complet, richement documenté mais très cher.
- `BorlandC++` : concurrent de `VisualC++` pendant longtemps, il est maintenant en perte de vitesse. Il fut également payant et cher pendant longtemps. Actuellement ??
- `DevCpp` : environnement de développement intégré utilisant le compilateur g++ de MINGW (une console linux sous Windows). C'est un outil gratuit.
  (`http://www.bloodshed.net/devcpp.html`)
- `Code::Blocks` : idem `DevCpp` (`http://www.codeblocks.org`).
- `Eclipse-CDT` : environnement de développement totalement paramétrable à ses besoins. Il est aussi riche qu'il est difficile à utiliser. Je ne le conseille pas aux débutants. Totalement en Java, il nécessite également d'avoir un ordinateur performant. C'est un outil gratuit.

Le point négatif de ces outils est leur complexité. S'il semble pratique de les utiliser, ils ne sont pas nécessairement intuitifs ni même faciles à utiliser. Ceci est d'autant plus vrai que les développements que nous sommes amener à réaliser sont légers.

Dans mon usage personnel, je n'utilise pas d'IDE pour développer en C. Je m'en sers uniquement en C++ lorsque les projets commencent à être importants (>10000 lignes de code). Depuis que je m'y suis familiarisé, j'utilise principalement `Eclipse-CDT`. Mais je continue à faire beaucoup de débuggage en ligne de commandes.

Mon choix de `Code::Blocks` est motivé par :

- c'est un outil "Cross-Plateform" : vous pouvez aussi bien l'installer sous Windows, sous Linux et également sous Mac (!). En particulier, vous pouvez continuer à travailler chez vous sans avoir à installer Linux,
- la possibilité de faire simplement les opérations nécessaires à ce cours, en particulier toute la partie de compilation,
- la richesse suffisante de cet outil,
- l'intégration du débugger.

2.2.2 Premier projet de `Code::Blocks`

Lors du premier lancement de `Code::Blocks`, il se peut qu'il vous demande des informations sur le compilateur. Avec une installation par défaut (avec MINGW) vous ne devez disposer que d'un seul compilateur (celui de MINGW), vous avez donc juste à dire OK !

Création du projet

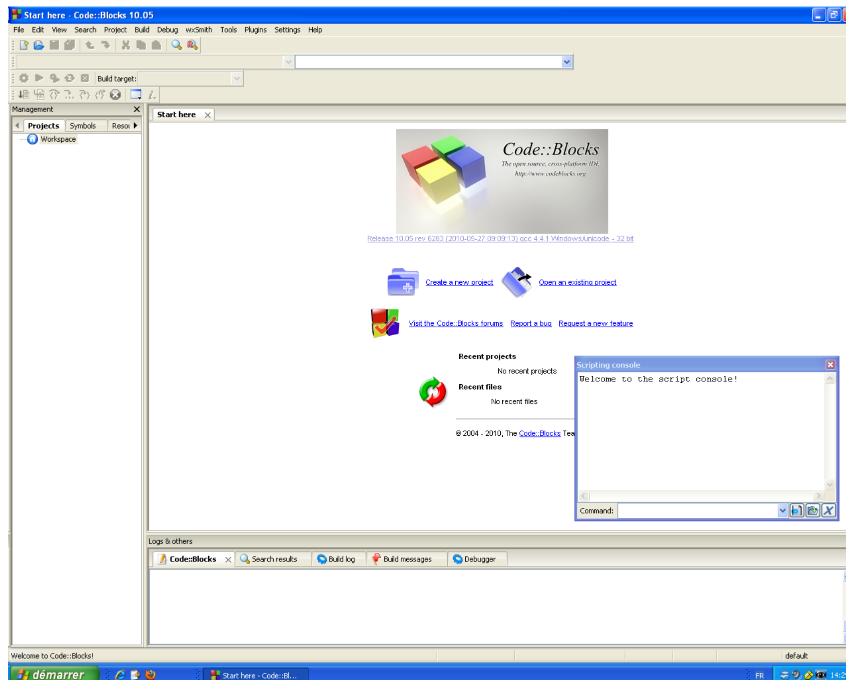


FIGURE 2.1 – Interface d'accueil de `Code::Blocks`

Lorsque vous lancer `Code::Blocks`, vous arrivez sur la fenêtre d'accueil illutrée dans la Figure 2.1. Commencez peut être par faire un peu de ménage dans les fenêtres qui sont affichées : fermer l'onglet “start here” et la console de scripts.
Pour commencer à travailler, créer un nouveau projet :

1. cliquer sur “nouveau projet” au centre de l'interface,
2. sélectionner ensuite “Empty project” pour créer un projet vide,
3. répondez aux questions relatives à l'emplacement du projet
   – nom du projet (par ex. “Test_LangageC”)
   – “folder to create project in” : donner un répertoire où enregistrer votre projet
4. répondez aux questions relatives au compilateur (doit être “GNU GCC”, ne rien modifier)

Ensuite, il vous faut créer des fichiers de programme. Pour cela :

1. aller dans le menu `File>New empty file`
2. répondre “Oui” à l'insertion du fichier au projet
3. donner un nom à votre fichier (par exemple “`test.c`”) et valider
4. sélectionner seulement “Debug” (et pas “Release”) dans la fenêtre suivante et valider. La configuration “Debug” correspond à la compilation d'un programme qui vous permettra de débugger plus facilement. C'est la version de compilation utilisé pendant la réalisation d'un projet.

> Lorsque votre programme est finalisé, vous pourriez passer à un mode de compilation “Release” dans lequel le programme sera plus efficace. Dans ce cours, il n’est pas nécessaire de travailler sur deux versions ...

Et voilà tout est prêt pour écrire votre programme. Par exemple, le petit programme ci-dessous :

```c
#include <stdio.h>

int main() {
    printf("hello world\n");
    return 0;
}
```

Il est possible de (de-)zoomer sur le code (changement de taille de la police) en utilisant `Ctrl+Roulette souris` sur la zone du code.

> **! Attention !**
>
> À chaque nouveau programme, il faut créer un nouveau projet !!
> Vous pouvez travailler sur plusieurs projets en même temps dans `Code::Blocks`.

Vous pouvez également créer un projet de type “application console” qui vous créera automatiquement le premier fichier du `main`. Pour l’insertion des fichiers suivants, il faudra procéder de la même manière que décrit ci-dessus.

Compilation et exécution d’un programme

Dans la Figure 2.2, `Code::blocks` comporte deux projets `Test_LangageC` et `exo2` visualisé dans la liste de gauche. Le projet actif est celui dont le nom est en gras (`Test_LangageC` sur l’illustration). Dans les sources, on voit qu’un seul fichier “`Test.c`” fait partie du projet.

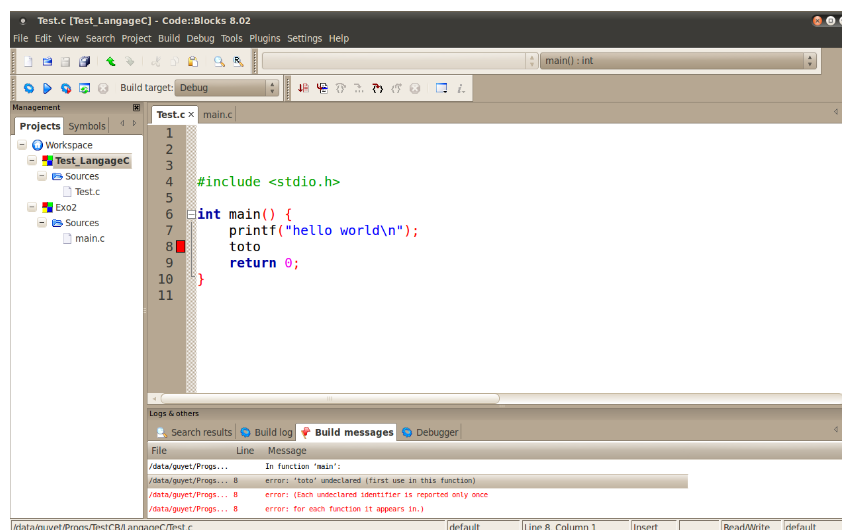


FIGURE 2.2 – Interface d’accueil de `Code::Blocks`.

Pour compiler le programme, il suffit de cliquer sur l’icône en forme d’engrenage (icône à gauche sur la Figure 2.3), en haut à gauche ou de faire `Ctrl+F9`. La fenêtre du bas indique les éventuels message

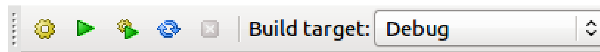


FIGURE 2.3 – Barre de menu pour la compilation dans `Code::Blocks`.

d'erreurs (en rouge). En double-cliquant sur les éléments de cette liste vous déplacerez le curseur à l'emplacement de l'erreur dans le code (également repérée par un carré rouge).
N'oubliez pas d'enregistrer vos fichiers après chaque modification pour qu'elle soit prise en compte dans la compilation !

Une fois les erreurs corrigées et la compilation réussie, il suffit de cliquer sur l'icône en forme de flèche pour exécuter le programme (second icône sur la Figure 2.3) ou de faire `Ctrl+F10`.
La touche de raccourcis `F9` permet de compiler et d'exécuter (l'exécution n'étant faite que si aucune erreur n'est trouvée dans le code). Vous pouvez faire la même chose à partir du troisième icône de la Figure 2.3. Le dernier icône actif de la barre de menu permet de faire une reconstruction totale du projet (supprime les compilations intermédiaire et recommence de zéro). Le dernier icône sert à stopper un programme en court d'exécution.

Il est également possible d'exécuter votre programme depuis la console Windows (oui ! ça existe aussi !!). Pour cela, il faut ouvrir la console Windows :

1. aller dans le menu Windows, et utiliser l'icône "Exécuter" (ou faire `Ctrl+R` depuis le bureau)
2. entrer la commande `cmd`
3. valider

Une fois dans la console, vous pouvez naviguer dans les répertoires avec les commandes `cd` et `dir` (équivalent de `ls`). Une fois positionné dans le répertoire de travail (les exécutables sont créés dans le répertoire `debug/bin` de votre projet `Code::Blocks`), il suffit taper le nom de l'exécutable (qui doit avoir le nom de votre projet), par exemple : `Test_LangageC.exe` ... et c'est partis.

2.2.3 Fonctionnalités de `Code::Blocks`

La première fonctionnalité dont vous bénéficiez est l'indentation automatique. Il y a deux moyens de bénéficier de l'indentation automatique dans `Code::Blocks` :
- au fil de la frappe : lorsque vous ajouter une nouvelle ligne, le curseur se place automatiquement à la bonne position.
- a posteriori : en allant dans le menu `Plugins>Source code formatter`, le logiciel indente automatiquement tout votre code.

Voici quelques unes des fonctionnalités de `Code::Blocks` dont je vous laisse découvrir l'usage (en fonction de vos besoins) :
- fonctionnalité de parcours de code par les fonctions, les symboles, ...
- auto-complétion,
- parcours des symboles (variables et fonctions).

Débuggage interactif

Il est possible de faire du débuggage interactif simplement dans `Code::Blocks`. Pour cela, vous pouvez utiliser le menu dédié à l'exécution pas à pas d'un programme (*cf.* Figure 2.4). De gauche à droite, les icônes permettent (en gras, les plus importants) :
- **Continue** : lance le programme en mode "debug" ou relance la suite du programme après un arrêt pendant l'exécution pas à pas,

- **Run to cursor** : lance le programme en arrêtant l'exécution à la position du curseur,
- **Next** : exécute l'instruction C suivante (*i.e.* généralement l'instruction sur la ligne du curseur)
- Next instruction : exécute l'instruction *assembleur* suivante (pas utile !)
- **Step** : fait la même chose que Next, sauf si l'instruction est une fonction, dans ce cas, le programme "rentre" dans la fonction,
- Step out : sort de la fonction courante (exécution de la fin des instructions de la fonction),
- **Stop** : arrête l'exécution programme.

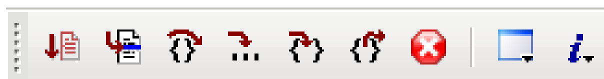


FIGURE 2.4 – Barre de menu pour le débuggage dans `Code::Blocks`.

Lors de l'arrêt d'un programme, il est possible de visualiser le contenu des variables dans la fenêtre *Watches* qui peut être ouverte à partir du huitième icône (liste déroulante d'outils de visualisation). Finalement, il est possible d'ajouter des points d'arrêt (*breakpoint*) dans votre programme en utilisant la touche `F5`. Cette touche ajoute ou supprime un *breakpoint* à la ligne où se trouve le curseur. Ce *breakpoint* imposera au débuggeur de s'arrêter juste avant cette ligne.

Dans l'illustration 2.2.3, on visualise un court programme pour lequel on a mis un *breakpoint* à la ligne 9 (visualisé par un rond rouge). Le débuggeur a été lancé. La flèche jaune visualise là où en est l'exécution du programme : actuellement, c'est à la ligne 11 (juste avant l'instruction de cette ligne). Sur la droite, la fenêtre des *Watches* permet de connaître le contenu des variables **p** et **a**.

```
main.c
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      double p = 56;
7      int a=0;
8
9      p=45;
10
11     p = p * 2;
12     return 0;
13 }
14
```

Watches
Local variables
p = 45
a = 0
Function Arguments

CHAPITRE

3 Complément de compilation en C

Dans cette partie, on apporte quelques précisions sur les étapes de compilation spécifiques au langage C. La Figure 3.1 illustre le schéma général de la compilation d'un programme C utilisant plusieurs fichiers `.c` et `.h`.
On présente également l'utilisation approfondie du compilateur `gcc` en ligne de commande ainsi que l'usage des `Makefile`.

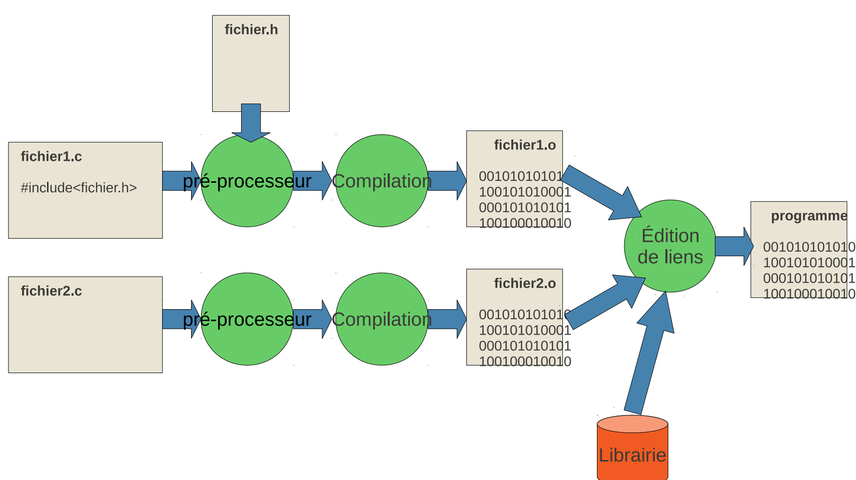


FIGURE 3.1 – Schéma général de la compilation d'un programme C

Un programme en C est un ensemble de fichiers texte contenant les programmes. On distinguera par la suite deux types de fichiers :
– les fichiers `.c` : ils contiennent les implémentations des fonctions,
– les fichiers `.h` : ils contiennent uniquement les déclarations de fonctions et de variables qui peuvent être utilisé par plusieurs fichiers `.c`. **Ces fichiers ne doivent pas être compilés**.

3.1 Introduction à la compilation

Lors de la compilation, il y a quatre tâches distinctes :
– La **vérification syntaxique** regarde si le programme est syntaxiquement correct,
– La **compilation** construit des bouts de code assembleurs à partir de vos programmes,
– L'**assemblage** construit du code machine à partir du code assembleur (je néglige cette étape par la suite),
– L'**édition de liens** lie vos bouts de code ensemble, avec éventuellement d'autres bouts de codes extérieurs que vous auriez importé (sous la forme de librairies).

Des erreurs spécifiques peuvent apparaître à chacune de ces étapes.

3.1.1 Notions d'analyse syntaxique

Lorsque vous écrivez un programme, il y a tout un tas de *règles syntaxiques* auxquelles le programmeur doit se contraindre. Ces règles permettent aux compilateurs de comprendre sans ambiguïté ce que le programmeur souhaitait que le programme fasse. Ces conventions sont donc indispensables à l'inter-compréhension de la machine et du programmeur. Avant de pouvoir passer à l'étape de compilation proprement dite, le compilateur vérifie donc que la syntaxe élémentaire est correcte.

En langage C, il vérifie, par exemple :

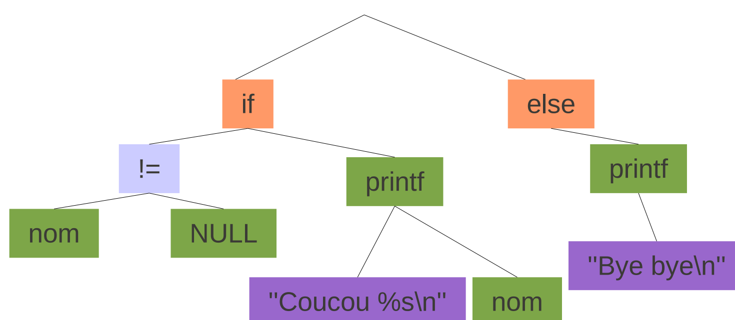
– que toutes les instructions finissent par un point virgule,
– que toutes les fonctions auxquels il est fait référence ont été déclarées (existent),
– que les accolades, les parenthèses, les guillemets ouverts sont bien fermées,
– que les structures de contrôle sont bien utilisées,
– ...

Pour comprendre le programme, le compilateur découpe votre fichier en lexème, c'est-à-dire en unité syntaxique et représente votre programme sous la forme d'un arbre (*cf.* Figure 3.2). Pour cela, il s'appuie sur les séparateurs : l'espace, les parenthèses, les accolades, les guillemets, les virgules, les points virgules.

Ensuite, il vérifie que chacun des lexèmes correspond à un mot qu'il connaît soit parce que c'est un mot clé du langage C, soit parce que c'est un mot que vous avez défini (*table des symboles*), soit parce que c'est un mot qui fait partie d'une librairie que vous avez explicitement importé (avec la directive de préprocessing **include**, *cf.* section 3.2).

Plus finement, il vérifie que ces lexèmes vont bien ensemble (on parle un peu à tort d'*analyse sémantique*, il s'agit plutôt d'une analyse grammaticale). Par exemple, il faut vérifier que après le mot clé `if` il y a bien une parenthèse pour contenir le test.

Finalement, les dernières vérification, mais pas moins les moins importantes, sont les vérifications de type. Le langage C est un langage "fortement typé", c'est à dire que toutes les variables, toutes les fonctions, tous les paramètres ont un type et que le compilateur fait la vérification de la bonne correspondance des types. Par exemple, une fois qu'on sait qu'il y a un élément entre les parenthèse du `if`, il faut faire la vérification qu'il s'agit bien d'un résultat booléen et pas du texte ou autre chose.




```c
if( nom != NULL ) {
  // Commentaire
  printf("Coucou %s\n", nom);
} else {
  printf("Bye bye\n");
}
```

FIGURE 3.2 – Exemple d'analyse syntaxique. En bas : le programme C, en haut : représentation du programme sous la forme d'un arbre par analyse syntaxique.

Si une erreur de syntaxe est trouvée, alors la compilation est arrêtée et des messages d'erreur vous invite de manière assez `explicite` à corriger celles-ci. Le compilateur peut même vous suggérer des modifications.

Remarque 3 - Lisez les messages d'erreur

Les messages d'erreurs sont explicites ! ! Au début, il peut être difficile de les lire, mais à force de les voir vous devriez apprendre à les reconnaître et à corriger vous même vos erreurs !

Remarque 4 - Utiliser le compilateur régulièrement pour la syntaxe !

Le compilateur peut également vous aider à écrire votre programme au fur et à mesure. Dans la mesure où il vous informe de la correction de votre syntaxe, n'hésitez pas à faire appel à lui (plutôt qu'à l'enseignant) pour vous donner un coup de main ...

Même en cas d'erreur, l'analyse syntaxique peut continuer afin d'identifier toutes les erreurs syntaxiques en une seule lecture du fichier. Vous pouvez ainsi vous retrouver avec des centaines d'erreur assez rapidement ! Il ne faut pas paniquer ... En règle général, il vaut mieux commencer par traiter les premières erreurs qui ont été identifiées, il y a des chances que si le début a été mal lu, le compilateur n'a plus rien compris à la suite et qu'il a trouvé plein de fautes "imaginaires".
D'autres messages d'information peuvent indiquer des "*Warning*". Ceci ne bloque pas la compilation, mais il est conseillé de les corriger également. Même si ils ne gênent pas la compilation, les *warnings* peuvent avoir des conséquences sur le bon déroulement de l'exécution du programme. Rien ne vaut un programme nickel d'un point de vue du compilateur.

! Attention !

Ce n'est pas parce qu'un programme est syntaxiquement correct qu'il est "bon" ! En effet, généralement, un programme doit faire quelque chose, une syntaxe correcte ne garantis en rien que ce que fait le programme est conforme aux spécifications.

Je ne préfère pas aborder ici les autres types d'erreur qui peuvent être rencontrées et qui sont plus complexes à généraliser :
- erreur d'édition de liens ;
- erreurs d'exécution.

3.1.2 Notion d'optimisation du code

Un compilateur est un programme très complexe qui est capable de "comprendre" votre code et parfois de le remplacer en utilisant des astuces qui exploitent les spécificités de votre architecture matérielle. On parle alors d'optimisation.
Exemple d'optimisation qui pourra être faite de manière transparente pour le programmeur :
Voici une version non-optimisée qui nécessite de faire autant d'appel à la fonction **getLimite**() que de tour de boucle :

```
for(int i=0; i<getLimite(); i++) {
  //...
}
```

Et voici, une version optimisée, avec un seul appel à la fonction **getLimite**() :

```
int l=getLimite();
for(int i=0; i<l; i++) {
  //...
}
```

Il ne s'agit là que d'un exemple simple, mais c'est avec de petites modifications que le compilateurs est capable d'obtenir des améliorations impressionnantes.
Très souvent, il est même recommandé de laisser le compilateur faire les optimisations plutôt que de les faire soit même. Le compilateur est généralement plus efficace que l'humain (programmeur moyen) pour trouver des optimisations, et les programmes restent ainsi lisibles, plus facilement compréhensibles.

3.2 Le pré-processeur

La compilation d'un fichier `.c` se fait en deux étapes (*cf.* Figure 3.3) :
- le pré-processeur apporte quelques modifications à votre fichier pour le préparer à la compilation,
- la compilation construit un fichier binaire (objet ou programme).

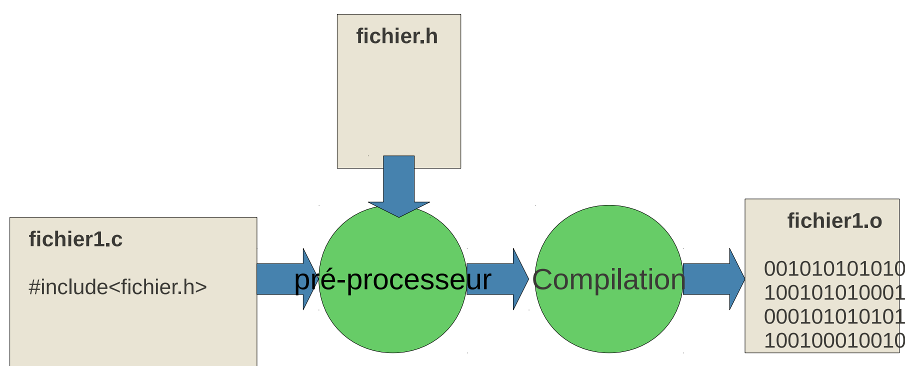


FIGURE 3.3 – Compilation d'un fichier C

Le pré-processeur traite toutes instructions qui commencent par le caractère **#** dans un programme C. Les instructions qui peuvent être données au pré-processeur sont les suivantes :
- **#include** `"fichier.h"` : demande au préprocesseur de copier le contenu du fichier fichier.h à cet emplacement. Cette instruction au pré-processeur est très fréquemment utilisé pour importer des fonctions décrites dans un autre fichier.
- **#define** TOTO : déclare la variable `TOTO` (les variables du préprocesseur sont très souvent en majuscule)
- **#define** FNCT(x)... : déclare une macro `FNCT` avec une variable `x`
- **#ifndef** TOTO ... **#endif** : si la variable `TOTO` n'est pas définie, alors le code contenu jusqu'au **#endif** sera compilé (sinon, il sera totalement ignoré)
- **#define** TOTO 10 : déclare la variable `TOTO` avec la chaîne de caractères 10. à chaque fois que le préprocesseur rencontrera cette variable, il la remplacera par 10 *avant la compilation* (il effectue simplement un remplacement du texte). Ceci est un moyen efficace de définir des constantes utiles dans un programme (par exemple la valeur de `Pi`).

3.2.1 Exemple de l'effet du pré-processeur lors de la compilation

Le petit programme 3.2.1 utilise des commandes spécifiques au pré-processeur. En utilisant la commande de compilation suivante (avec l'option -E, on ne fait fonctionner que le pré-processeur :

$ gcc -E -o *exemple_compilation_pp.i exemple_compilation_pp.c*

Le résultat est de cette commande est illustré par le programme 3.2.1, pour lequel certains remplacements ont été réalisés. On constate, d'une part, que la variable de préprocesseur **LIMITE** a été remplacée par sa valeur 10 dans le for et, d'autre part, que la macro **VALEUR_ABSOLUE** a été remplacée "intelligemment" (*i.e.* toutes les valeurs de x ont été remplacée par la valeur du paramètre effectif).

```
#include <stdio.h>
#define LIMITE 10
#define VALEUR_ABSOLUE(x) ( ((x) < 0)?-(x):(x) )

int main() {
  int i=0;
  for(i=-LIMITE; i< LIMITE; i++) {
    int val = VALEUR_ABSOLUE(i*i*i);
    printf("%d\n", val);
  }
  return 0;
}
```

```
int main() {
 int i=0;
 for(i=-10; i< 10; i++) {
  int val = ( ((i*i*i) < 0)?-(i*i*i):(i*i*i) );
  printf("%d\n", val);
 }
 return 0;
}
```

En pratique, le programme contient beaucoup de nouvelles lignes au début de fichier que je ne fais pas apparaître, vous pourrez menez l'expérience par vous même pour les voir, mais elles ont peu d'intérêt.

3.2.2 Utilisation des `define` dans les `.h`

Voici maintenant un exemple d'illustration du fonctionnement du préprocesseur qui est très utilisé dans l'écriture de fichiers `.h` propres. Je recommande donc de faire de même dans tout vos `.h`.

```
/***
* Exemple de fichier .h
***/
#ifndef FICHIER_H
#define FICHIER_H
 ...
#endif
```

Dans cet exemple, on indique que si la variable **FICHIER_H** n'est pas définit, on la définie et on ajoute (à la place des ...) les instructions en C spécifique à ce fichier. Cette façon de procéder permet de s'assurer que le préprocesseur n'insérera qu'une seule fois le contenu de ce fichier `.h` avant la compilation d'un `.c`. Lorsque le nombre de fichiers croît, il est facile de se retrouver avec des dépendances cycliques entre les fichiers `.c` et `.h` de sorte que le préprocesseur insère deux fois un même `.h`. Cela

pose des problèmes de compilation par la suite. Ici, si le fichier est appelé une seconde fois, par le préprocesseur, le fait que la variable **FICHIER_H** soit déjà définie va bloquer l'insertion du contenu du fichier.
Pour chaque fichier `.h`, il faut un nom de variable différent.
Le préprocesseur supprime également tous les commentaires (// et /* ~~~*/) qui sont utiles au programmeur, mais inutiles pour l'ordinateur.

3.3 Compilation séparée

Jusque là, nous nous sommes principalement intéressé à des programmes ne contenant qu'un unique fichier. Pour de gros programmes, il n'est pas raisonnable d'imaginer que tout ce passe dans un seul fichier. Si les programmes sont réparties dans plusieurs fichiers, il faut avoir recours à la compilation séparée. Le problème de la décomposition d'un programme en plusieurs fichiers sera abordé dans la partie sur la "décomposition" des programmes. Je présente ici le problème de la compilation séparée.

Remarque 5

Je commence par faire remarquer que la gestion de la compilation avec plusieurs fichiers est généralement transparente avec l'utilisation d'un IDE comme `Code::Blocks`.

Commençons par donner les deux grandes étapes nécessaires à la compilation séparée (*cf.* Figures 3.1) :

1. tous les fichiers `.c` sont **compilés** indépendamment sous la forme de fichiers objets (`.o`),
2. tous les fichiers objets (`.o`) sont assemblés par une étape d'**édition de liens**. Dans cette étape, les liens sont également effectué avec des librairies de fonctions extérieures à votre code mais que vous utilisez.

3.3.1 Compilation d'un fichier `.c` en fichier objet avec `gcc`

Pour compiler un fichier `.c` contenant un programme écrit en langage C sous la forme d'un fichier objet, il faut utiliser l'option `-c` de `gcc`. `gcc` est un compilateur qui s'exécute en ligne de commande sous Linux. Consulter la partie du cours sur l'utilisation de l'environnement de programmation Linux pour bien comprendre les explications suivantes.
Un fichier objet, extension `.o`, est un bout de programme compilé. Il ne peut pas être directement exécuté par le processeur, mais contient la traduction des programmes dans le langage machine. L'édition de liens se chargera *simplement* de mettre ensemble tous les fichiers `.o` et à indiquer par où il faut commencer l'exécution du programme.
Commande permettant de compiler un fichier C à la suivante :

```
$ gcc -c -o fichier.o fichier.c
```

ou

```
$ gcc -c fichier.c
```

La ligne de commande se comprend ainsi :
– `-c` : option qui indique de ne faire que compiler le programme (pas d'édition de lien)

– `-o fichier.o` : option pour imposer le nom du fichier binaire qui est construit (par défaut, il prend le même nom que le fichier `.c` et remplace juste l'extension en `.o`)
– `fichier.c` : nom du fichier à compiler (toujours à la fin de la ligne de commande)

Vous noterez qu'il n'est fait référence à aucun fichier `.h`. En effet, le compilateur ira chercher ces fichiers si le préprocesseur en a trouvé. Mais où va-t-il chercher les fichiers `.h` dans l'arborescence de fichier ? Par défaut, le compilateur les cherche dans le répertoire courant, et dans un certain nombre de répertoires usuels (définie lors de la configuration du système), par exemple `/usr/include`. Si vous souhaitez ajouter des répertoires dans lesquels le compilateur aura besoin d'aller identifier des `.h`, alors, il est nécessaire de lui indiquer explicitement à l'aide de l'option `-I`.
Par exemple :

```
$ gcc -c -o fichier.o -I ./monrepertoire/ fichier.c
```

Le résultat de la compilation est un fichier binaire qui est la traduction dans le langage machine du programme écrit en C. Pour visualiser le résultat, vous pouvez essayer d'ouvrir le fichier sous un éditeur de texte ... comme c'est du binaire, ça ne marche pas, ou bien avec une visualisation du contenu binaire :

```
$ od -a fichier.o
$ od -x fichier.o
```

3.3.2 Édition de liens entre plusieurs fichiers `.o`

Une fois que tous vos fichiers `.c` ont été compilés en fichier `.o`, vous pouvez passer à l'édition de liens. La commande pour faire l'édition de liens entre trois fichiers `.o` est la suivante :

```
$ gcc -o monprogramme fichier1.o fichier2.o fichier3.o
```

La ligne de commande se comprend ainsi :
– `-c` : contrairement à la compilation, **il n'y a pas d'option -c ici.**
– `-o monprogramme` : le fichier binaire généré s'appellera monprogramme, en l'absence de cette fonction, le fichier généré s'appellera `a.out`.
– `fichier1.o ...` : à la fin de la ligne de commande, il faut préciser tous les fichiers `.o` que vous souhaitez lier entre eux.

De même que pour l'importation de fichier `.h` lors de la compilation, vous pourriez avoir besoin de préciser explicitement l'utilisation de librairies et où trouver ces libraires. Il faut alors utiliser les options `-L` et `-l`. Je ne détaille pas ce point ici, vous pouvez consulter la documentation de gcc pour plus d'informations.
Exemple d'utilisation de la librairie de fonctionnalités mathématiques :

```
$ gcc -o monprogramme -lm fichier1.o fichier2.o fichier3.o
```

3.4 Compilation rapide d'un fichier unique

Dans le cadre de ce cours, il est possible que vos programmes tiennent dans un seul et même fichier. Il n'est alors pas nécessaire de faire deux commandes pour compiler le programme. La commande suivante réalise d'un coup les deux étapes (compilation + édition de liens) :

```
$ gcc -o monprogramme fichier.c
```

3.5 Utilisation d'un Makefile

Un `Makefile` est un fichier, nommé `Makefile` (sans extension), qui décrit la proédure de compilation qui doit être automatisée. Ce fichier est utilisé par la commande `make`.

3.5.1 Un exemple simple

Prenons l'exemple de trois fichiers :
- `fonctions.h` : un fichier contenant les descriptions de quelques fonctions utilent au programme,
- `fonctions.c` : il contient les implémentations des fonctions de `fonctions.h`. Ce fichier fait appel à `fonctions.h`,
- `main.c` : contient la fonction **main**(), c'est le programme principal. Nous supposons qu'il fait appel à `fonctions.h`.

La Figure 3.4 illustre les dépendances entre les fichiers et les étapes de compilation à effectuer.

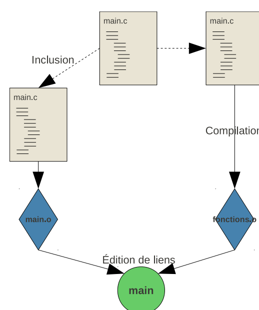


FIGURE 3.4 – Ordre de compilation et dépendances

Supposons que nous fassions une modification du fichier `main.c`, alors, il suffit de compiler de nouveau le fichier `main.c` et de refaire l'édition en lien. En revanche, si vous modifier le fichier `fonctions.h`, alors tous les fichiers doivent être compilés. En utilisant les dates des fichiers, la commande `make` est capable de déterminer les bons fichiers à compiler à partir des dépendances entre fichiers décrites dans le `Makefile`.

Pour cet exemple, le `Makefile` (assez simple) aura l'allure suivante :

```
1  all : main
2
3  main: main.o fonctions.o
4    gcc -o main main.o fonctions.o
5
6  fonction.o: fonctions.c fonctions.h
7    gcc -c -o fonctions.o fonctions.c
8
9  main.o: main.c fonctions.h
10   gcc -c -o main.o main.c
11
12 clean:
13   rm -f main.o fonctions.o main
```

Ce fichier s'interprète à partir de motifs de règles qui ont l'allure suivante :

```
cible: dependance1 dependance2 ...
       commande
       commande
```

Dans cette règle, `cible` indique le nom fichier qui sera produit par l'exécution de règle. Sur la même ligne, et séparé de " :" avec la cible, on donne toutes les dépendances nécessaire à la production de la cible. Sur les lignes en-dessous, décalées par une tabulation [1], on donne la (ou les) commande qu'il faut exécuter (typiquement un gcc).
Dans le `Makefile` exemple, la première règle `all` désigne ce qui doit être réalisé par défaut. On retrouve ensuite la règle de `main` qui réalise l'édition de liens et les deux règles pour les compilations de `main.c` et `fonctions.c`. Finalement on donne également une règle nommée `clean` pour faciliter la suppression de tous les fichiers compilés. Pour la règle de `fonctions.o`, on a comme dépendance le fichier `fonctions.c` et également `fonctions.h` qui permettra de savoir qu'il faut réappliquer cette règle si l'un de ces fichiers a été modifié. La commande réalisé lors du déclenchement de cette règle est simplement un appel à `gcc`. Pour la règle `clean`, il n'y a pas de dépendance et la commande exécuté sera la commande système `rm` (suppression de fichier).

Une fois ce fichier enregistré (dans le même répertoire que les fichiers sources), il s'utilise ainsi :
- pour compiler ou recompiler votre programme `main`, tapez dans la console :
  `...$ make`
- pour déclencher la règle `clean`, faire :
  `...$ make clean`

Une fois le `Makefile` en place, on ne se pose plus de question sur la compilation !! C'est très pratique !

3.5.2 Utilisation de variables dans le Makefile

L'intérêt de `make` ne s'arrête pas là. Il est possible d'utiliser des variables dans le `Makefile` qui vont vous faciliter la vie. L'intérêt principal est de pouvoir facilement changer toutes les options de compilation sans avoir à modifier toutes les règles.
Le cas le plus usuel est le suivant. On retrouve exactement les mêmes règles que précédemment, mais, on a ajouté aux commandes de compilation une variable `$(FLAGS)`. Cette variable sera remplacée par la valeur `-O2 -Wall`, définie en haut du fichier. On peut modifier facilement la valeur de cette variable (en `-g` par exemple). Dans ce cas, la modification s'appliquera aux deux commandes de compilation.

```
1  FLAGS= -O2 -Wall
2
3  all : main
4
5  main: main.o fonctions.o
6    gcc -o main main.o fonctions.o
7
8  fonction.o: fonctions.c fonctions.h
9    gcc $(FLAGS) -c -o fonctions.o fonctions.c
10
11 main.o: main.c fonctions.h
12   gcc $(FLAGS) -c -o main.o main.c
13
14 clean:
15   rm -f main.o fonctions.o main
```

Il restera alors à enregistrer le `Makefile`, puis à faire un `make clean` pour supprimer les anciennes versions compilées avec en `-O2` et à recompiler l'ensemble avec un `make`. On peut ainsi facilement passer d'une compilation pour le débuggage (option `-g`) à une compilation finale (option `-O2`).

Beaucoup plus de fonctionnalité existent pour les `Makefile`, mais leur maîtrise n'est pas justifié dans le cadre de ce cours.

1. Attention, il faut impérativement utiliser une tabulation. Les espaces ne vont pas fonctionner.

3.6 Options avancées de compilation

Lors de la compilation vous pouvez ajouter des options qui vont modifier le comportement du compilateur.
`-Wall` (*Warning all*) : L'ajout de cette option indique au compilateur qu'on veut qu'il nous informe de tous les *warnings* qu'il a pu trouver. Cette option assure que votre code est nickel pour la compilation.
Un autre grande classe d'option concerne le choix de l'optimisation à faire sur le code :

– options `-g`, `-g3`, `-pg` : correspond à la compilation en mode "debug" et de "profiling"
– option `-O1`, `-O2`, `-O3` : correspond au niveau global d'optimisation. Plus vous augmenterez le niveau d'optimisation, plus la compilation du code prendra du temps et utilisera de la mémoire. (l'option `-O3` utilise des fonctionnalités d'optimisation de code plus poussées, mais elles peuvent avec des conséquences importantes sur la traduction des programmes, et mieux vaut ne pas les utiliser sans en maîtriser les finesses.

Pour que les outils de débuggage (`gdb` et `valgrind`) fonctionnent, il faut impérativement que les programmes soient compilées avec l'option `-g`. En utilisant cette option, le compilateur va insérer tout un tas de code qui va permettre aux outils de bien comprendre ce qui se passe et ainsi vous aider à trouver les erreurs. Bien évidemment, toutes ces insertions nuisent aux performances de votre code, mais peu importe, vous l'utiliser dans une phase préliminaire. L'option `-pg` permet d'ajouter de nouvelles informations dans le code qui vont permettre d'utiliser l'outil `gprof`.
Quand le code fonctionne bien, il est temps de compiler une vraie version (version "Release"). Et même pourquoi pas une version optimisée en recompilant tout le code avec une option du type `-O`.Le niveau `-O2` est le niveau généralement requis pour l'optimisation d'un code finalisé.