Initiation à la programmation en C

la compilation

Antoine Miné

Année 2006–2007

Site du cours: `http://www.di.ens.fr/~mine/enseignement/prog2006/`

Le C est un langage compilé. La programmation en C nécessite donc l'emploi d'un outil spécialisé appelé **compilateur C**, dont nous expliquons ici le maniement.

La section 1 couvre les opérations élémentaires. Sa lecture est suffisante pour réaliser les premiers TPs. La section 2 décrit quelques utilisations plus avancées. Elle peut être sautée en première lecture. Cette fiche comporte également des tables de références utiles : quelques raccourcis pour Emacs (Fig. 1, p. 2), ainsi qu'une table des options de compilation utiles (Fig. 8, p. 13).

Lors des TP, nous utiliserons le compilateur GCC de GNU installé sur toutes les stations Unix des machines élèves (salles S, T, etc.). Toutefois, les principes décrits seront également valables pour les autres compilateurs C et même des compilateurs pour d'autres langages. Si vous travaillez chez vous et n'êtes pas sûr de posséder un compilateur C, la section 2.8 est pour vous.

1 Utilisation élémentaire

Pour travailler, il nous faut trois outils :
- un éditeur de texte pour taper le fichier source,
- un compilateur C pour transformer le source en exécutable,
- une fenêtre de terminal pour lancer les ordres de compilation et d'exécution.

1.1 L'éditeur

Chacun à sa préférence, mais Emacs, XEmacs, gVim ou Vim sont conseillés. Ces éditeurs sont spécialement adaptés à la composition de programmes. En particulier, leur fonction de coloration syntaxique met en évidence les entités lexicales du C (mots-clés, constantes, commentaires, etc.) par un code de couleur. Une autre fonction utile est l'indentation automatique : l'éditeur décalera chaque ligne vers la droite à chaque ouverture d'un nouveau bloc ou pour chaque instruction dépendant d'une construction `if`, `for`, etc. Pour plus d'informations, nous renvoyons le lecteur aux nombreuses documentations existantes (par exemple, sur la page des tuteurs `http://www.tuteurs.ens.fr/unix/editeurs/`). Nous rappelons, en figure 1, une courte liste de raccourcis Emacs utiles.

1.2 Le fichier source

C'est un fichier contenant le texte de votre programme. Par convention, on utilise l'extension `.c`[1]. Lancez un terminal et tapez :[2][3]

[1] Ceci permet à l'éditeur et au compilateur de le reconnaître pour ce qu'il est : le source d'un programme écrit en langage C.

[2] Par convention, le dollar `$` symbolise l'invite de commande. Elle ressemble en réalité plutôt à `clipper-~ $`. Vous ne tapez donc que les lignes qui commencent par `$`, en omettant le `$`.

[3] Le `&` indique qu'il faut lancer Emacs en parallèle. Ceci permet de continuer à taper des commandes dans le terminal sans attendre d'avoir quitté Emacs. Si vous avez oublié le `&`, vous pouvez toujours vous rattraper en

raccourci [a]	effet
C-x C-f	ouvre ou créé un fichier
C-x C-s	enregistre le fichier
C-x C-w	enregistre le fichier sous un nouveau nom
C-x C-c	quitte Emacs
C-g	interrompt l'action courante[b]
C-_	annule la dernière action
C-x 2	coupe la fenêtre en deux pour afficher plusieurs fichiers
C-x 1	annule tous les découpages et n'affiche qu'une sous-fenêtre
C-x b	sélectionne le fichier ouvert affiché dans la (sous-)fenêtre courante
M-g	va à un numéro de ligne spécifié
C-s	recherche un morceau de texte
Tab	(re)indente automatiquement la ligne courante
clic gauche	sélectionne (cliquer puis déplacer)
clic milieu	copie la sélection à l'endroit pointé

[a] Par convention C-x représente la séquence Ctrl+x. Taper C-x C-f revient donc à maintenir Ctrl appuyée tandis qu'on presse x puis f. De même M-x signifie Méta+x. La touche Méta est symbolisée par un carré sur les Suns. Sur les PCs, c'est la touche Alt.

[b] Très utile quand Emacs ne semble pas répondre. Cela signifie généralement qu'il est bloqué dans un mode spécial et attend des informations complémentaire de votre part. N'hésitez pas alors à presser C-g plusieurs fois pour sortir de tous les modes spéciaux imbriqués et enfin revenir au mode "édition" classique que vous connaissez.

FIG. 1 – Quelques raccourcis clavier et souris utiles dans Emacs.

```
$ mkdir essai
$ cd essai
$ emacs essai1.c &
```

Tapez ensuite le programme suivant dans Emacs et sauvegardez-le (C-x C-s) :

essai1.c

```
#include <stdio.h>

int main()
{
  printf("Bonjour le monde!\n");
  return 0;
}
```

1.3 Le compilateur

Avant d'être exécuté, le source doit être transformé en langage machine (seul compréhensible directement pas l'ordinateur). C'est le rôle du compilateur. On utilisera ici le compilateur GCC, invoqué par la commande gcc. Dans son utilisation la plus élémentaire, GCC ne prend qu'un argument : le nom du fichier à compiler. Si tout se passe bien, **rien** n'est affiché à l'écran et un fichier exécutable a.out est généré. Pour enfin voir se dérouler notre programme, il faut exécuter ce fichier a.out[4] :

```
$ gcc essai1.c
$ ./a.out
Bonjour le monde!
```

tapant dans le terminal Ctrl+Z (interruption du programme en cours) puis la commande bg (mis en tâche de fond du programme interrompu).

[4] Taper a.out ne suffit généralement pas car le *shell* ne cherche les fichiers exécutables quand dans les répertoires par défaut (configurables par la variable d'environnement PATH). Il est donc indispensable de préciser le répertoire où trouver a.out. Le répertoire courant étant noté simplement . cela donne ./a.out.

1.4 Les erreurs

Si le programme est incorrect, un ou plusieurs messages d'erreurs apparaissent. Voici un exemple de tel programme :

essai2.c
```
#include <stdo.h>

int main()
{
  printf("Bonjour le monde!\n");
  return 0;
}
```

et la session correspondante :

```
$ gcc essai1.c
essai2.c:1:18: error: stdo.h: No such file or directory
essai2.c: In function 'main':
essai2.c:5: warning: incompatible implicit declaration of built-in function
'printf'
```

L'erreur est indiqué par la ligne `error:` (la signification de `warning:` est expliquée en 2.2). En cas d'erreur, la compilation échoue et aucun fichier `a.out` n'est généré[5].

Le mot-clé `error:` est suivi d'un message explicatif, et précédé de la position de l'erreur dans le source : nom de fichier (`essai2.c`), numéro de ligne (`1`) et parfois de colonne (`18`). Notez que l'indication de position est parfois approximative (en particulier en cas d'erreur de syntaxe : GCC pourra continuer à lire après l'erreur, pensant que la suite du source donnera un sens à son début, pour jeter l'éponge un peu plus loin quand il est vraiment perdu.) Comme une erreur est parfois la cause d'une cascade d'autres erreurs, il vaut mieux les corriger progressivement, en commençant par la première détectée.

Les causes d'erreurs sont variés. Voici quelques exemples :
- fichiers introuvables : `No such file or directory`
- erreurs de syntaxe : `expected 'XXX' before 'XXX'`
- erreurs de type :
  `invalid operands to XXX`
  `incompatible type for argument X of 'XXX'`
- variable ou fonction non déclarée ou non définie (la différence est expliquée en 2.2) :
  `'XXX' undeclared`
  `undefined reference to 'XXX'`

1.5 Les avertissements

GCC affiche parfois des messages marqués `warning:`. Contrairement aux erreurs, ces avertissements ne sont pas fatals et n'empêchent pas la génération du `a.out`. Toutefois, ils attirent l'attention du programmeur sur des constructions dangereuses. Il est fortement conseillé d'utiliser les options `-Wall` et `-Wextra` qui activent tous les messages d'avertissement. Par exemple, si le programme contient le fragment

essai3.c (extrait)
```
  if (x=2) { ... }
```

la compilation avec sans `-Wall` ne dira rien, celle avec `-Wall` indiquera :

```
$ gcc essai3.c -Wall
essai3.c:5: warning: suggest parentheses around assignment used as truth value
```

[5] Attention. Il peut rester un fichier `a.out` issu d'une compilation antérieure réussie.

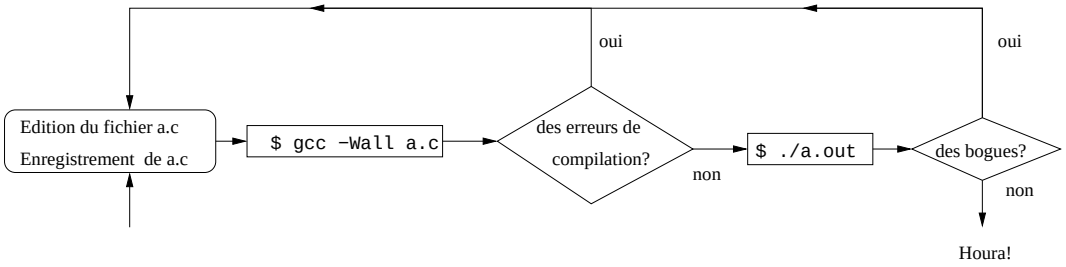

FIG. 2 – Cycle de développement d'un programme C.

En effet, j'ai confondu le test d'égalité `x==2` avec l'affectation `x=2`. Le résultat est un programme tout à fait correct, mais qui ne fait pas du tout ce que je veux[6]. Notez que `gcc -Wall` découvre l'affectation mal placée, mais ne suggère pas la bonne correction. (C'est déjà pas si mal...) L'option `-Wall` ajoute également des messages relatifs aux problèmes de typage, qui sont d'une aide précieuse. L'option `-Wextra` complémente `-Wall` en y ajoutant des avertissements qui, bien que considérés comme moins importants, donnent des informations bien utiles sur la qualité du programme.

1.6 Le cycle de développement

La figure 2 décrit (naïvement) une séance de programmation.

> On emploiera systématiquement les options `-Wall` et `-Wextra` et on s'efforcera d'obtenir une compilation qui n'affiche aucun message d'avertissement.

Mais, même après avoir supprimé toutes les erreurs et avertissements, il peut rester des bogues dans le programme. Deux cas fréquents :

- le programme s'interrompt avec un message semblable à `Segmentation fault` ou `Floating-point exception` : le programme a tenté une opération invalide comme un accès à une zone de mémoire invalide ou une division par zéro,
- le programme ne "plante" pas mais ne fait pas ce qu'on attend de lui.

Le débogage étant un sujet important, une fiche lui sera entièrement consacré.

2 Utilisation avancée

2.1 Pourquoi la compilation ?

Langages machines et langages de haut niveau. Dans l'ordinateur, c'est le microprocesseur qui a lourde tâche d'exécuter les programmes. Celui-ci ne comprend qu'un langage très rudimentaire appelé *langage machine* ou *binaire*. Comme son nom l'indique, il est formé de séquences de nombres et est réservé à la machine. Programmer directement en binaire, ou même en langage d'assemblage (version légèrement plus évoluée, "pour humain" où les instructions sont représentées par des sigles de trois ou quatre lettres), n'est pas une mince affaire. Une opération aussi simple qu'une addition ou une écriture dans un fichier doit être décomposée en de multiples instructions élémentaires. Afin de vous faire un peu peur, la figure 3 montre ce que donnerait le programme `essai1.c` traduit en langage d'assemblage et en langage machine. De plus, le langage machine ayant peu de gardes fous, il est facile de faire des erreurs (bogues). Enfin, chaque microprocesseur ayant son propre langage binaire incompatible, l'opération de *portage* d'un programme à un autre type d'ordinateur se résume à une pure et simple réécriture.

On utilise donc des langages de haut niveau (comme le C) qui sont (relativement) indépendants du microprocesseur, ont une syntaxe claire (pour un humain) et permettent de décrire succinctement des calculs complexes sans (trop) se perdre dans les détails.

[6]Son effet est le suivant. La valeur 2 est stockée dans la variable `x`. Puis, la valeur de l'expression `x=2`, c'est à dire 2, est interprétée comme valeur de vérité. 2 étant non nul, il s'agit de la valeur *vrai*, donc la branche de la conditionnelle est toujours prise !

```
    .section .rodata
.LC0:
    .string  "Bonjour le monde!"
    .text
.globl main
    .type    main, @function
main:
    pushq    %rbp
    movq     %rsp, %rbp
    movl     $.LC0, %edi
    call     puts
    movl     $0, %eax
    leave
    ret
```

Version assembleur (extrait).
Obtenue par `gcc -S essai1.c`.

```
55 48 89 e5 bf e8 05 40
00 e8 22 ff ff ff b8 00
00 00 00 c9 c3 90 90 90
      ...
42 6f 6e 6a 6f 75 72 20
6c 65 20 6d 40 05 f4 6f
6e 64 65 21 00
```

Version langage machine (hexadécimal).
Obtenue par `objdump -Ds ./a.out`.

FIG. 3 – Le programme `essai1.c` après compilation sur un PC Linux (microprocesseur intel Code Duo 2).

Compilation et interprétation. Nous l'avons dit, l'ordinateur ne peut exécuter que des programmes binaire. Une solution est donc d'utiliser un compilateur pour transformer le texte source en programme binaire exécutable. Il existe une alternative, employée par certains langages : l'interprétation. Au lieu de compiler une fois pour toute le programme source, l'interpréteur lit le source ligne par ligne et exécute les instructions au fur et à mesure qu'il les comprend. Comme l'interpréteur découvre les commandes au fur et à mesure, il peut être utilisé en mode interactif : les commandes ne sont pas lues depuis un fichier mais directement au clavier. Vous connaissez déjà un interpréteur : le *shell* ou "interpréteur de commandes" qui exécute les commandes tapées dans le terminal.

La compilation a de nombreux avantages par rapport à l'interprétation. L'analyse syntaxique et les vérifications en tout genre sont effectuées une fois pour toutes (à la compilation) et non à chaque exécution du programme. L'exécution du programme est donc beaucoup plus rapide. Le compilateur ayant à sa disposition le code source complet, il lui est possible d'effectuer des optimisations globales afin de rendre le code encore plus efficace, tandis que l'interpréteur ne peut raisonner que localement. Pour la même raison, le compilateur découvrira tout de suite toutes les erreurs de syntaxe, de typage, et de symboles non définis. Dans un langage interprété, ce type d'erreurs peut passer longtemps inaperçu et apparaître subitement, quand une partie du programme encore jamais testée est exécutée pour la première fois. Un autre avantage de la compilation est la possibilité de créer un programme composite dont les différents morceaux sont écrits dans des langages différents (voir 2.3).

Distribution. Notons finalement qu'un programme binaire issu d'une compilation n'a besoin ni du code source originel ni du compilateur pour s'exécuter. Il peut donc être distribué tel quel. Rappelons, toutefois, que le binaire n'est pas portable : il ne s'exécutera pas sur une machine d'un autre type. Il est donc poli de fournir le programme source pour que chacun puisse le (re)compiler à sa guise. Certains considèrent que la distribution de leur code source dévoilerait leur "secret de fabrication" et choisissent donc de ne distribuer *que* le binaire, illisible sauf pour l'ordinateur du bon modèle.

2.2 Les trois phases de la compilation

Jusqu'à présent, on a appelé "compilation" la transformation d'un fichier `.c` en `a.out`. En fait, cette transformation se décompose en trois phases :
- le pré-traitement (ou *preprocessing*),
- la compilation proprement dite,
- la liaison (ou édition de liens).

Ces phases sont schématisées dans la figure 4. Le pré-traitement effectue quelques transformations simples mais utiles sur le source C. La compilation fait la plus grosse partie du travail :

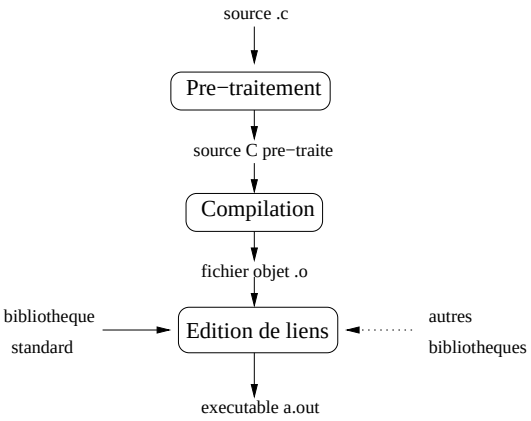

FIG. 4 – Les trois phases de GCC.

il transforme le source C pré-traité en fichier objet contenant le code binaire de chaque fonction. Un tel fichier objet n'est pas encore un programme autonome : il peut référencer des fonctions inconnues. L'édition de liens produit enfin un exécutable final `a.out` autonome[7] à partir de plusieurs fichiers : le (ou les, voir 2.3) fichier(s) objet(s) issu(s) de la compilation, la bibliothèque standard C ainsi que les bibliothèques demandées explicitement par l'utilisateur (voir 2.2.3).

Le simple fait d'appeler `gcc essai1.c` a pour effet de lancer ces trois phases successivement.

2.2.1 Le pré-traitement

La phase de pré-traitement s'occupe uniquement des instructions contenant `#`, appelées justement *directives de pré-traitement*. Nous décrivons trois exemples essentiels : `#include`, `#define`, et `#if`.

- Quand le pré-traiteur rencontre une ligne `#include <toto.h>`, il la remplace par le contenu d'un fichier `toto.h` qu'il va chercher dans une liste de répertoires prédéfinis (contenant généralement `/usr/include`). `#include "toto.h"` fonctionne de manière similaire, sauf que le fichier `toto.h` est d'abord cherché dans le répertoire courant.
- Quand le pré-traiteur rencontre `#define VAR val`, il remplace toutes les occurrences suivantes de `VAR` par `val`. Si `val` est omis, `VAR` sera remplacée par `1`. `VAR` peut contenir des arguments entre parenthèses, un peu comme une fonction C. Ainsi :

```
#define MIN(a,b) ( (a) < (b) ? (a) : (b) )
x = 12+MIN(10,f(2));
```

  donnera, après pré-traitement[8] :

```
x = 12+( (10) < (f(2)) ? (10) : (f(2)) );
```

- Enfin, quand le pré-traiteur rencontre des lignes encadrées entre `#if` *expr* et `#endif`, il évalue l'expression *expr*. Si celle-ci s'évalue en *vrai* (*i.e.*, différent de zéro), le code encadré est laissé inchangé. Sinon, il est supprimé. C'est la *compilation conditionnelle*. Attention toutefois, seul un sous-ensemble très restreint des expressions C est autorisé (constantes, constantes symboliques, opérateurs arithmétiques, logiques et de comparaison ; uniquement sur les entiers). Il est possible de préciser une alternative par `#if` ··· `#else` ··· `#endif`. Enfin, `#ifdef` *var* teste simplement si la constante symbolique est définie. Voici un exemple :

```
/* #define DEBUG */ /* à dé-commenter pour le débogage */
int toto(int c)
{
```

[7] Ce n'est pas tout à fait vrai, comme expliqué en 2.2.3.

[8] Notez que le pré-traiteur opère par recopie syntaxique. L'appel à la fonction `f(2)` se trouve donc dupliqué. Cela peut poser un problème si `f` est lente ou si elle a des *effets de bords* (par exemple un affichage sur écran ou une modification de variable globale).

```
#ifdef DEBUG
  printf("toto est appelé avec x=%i!\n",x);
#endif
  ...
}
```

En résumé, le pré-traitement transforme un source avec directives de pré-traitement en source sans directives. On comprend maintenant pourquoi ces directives ont une syntaxe très différent des autres instructions C : il ne s'agit tout simplement pas du même langage.

Le pré-traitement est influencé par quelques options de GCC. L'option de compilation `-I` *rep* permet d'ajouter *rep* à la liste des répertoires qui considérés quand le pré-traiteur cherche le fichier à #inclure (l'espace entre `-I` et *rep* est optionnel). L'option `-DVAR=val` définit la constante symbolique `VAR` comme ayant la valeur `val` (l'espace entre `-D` et `VAR` est optionnel). Ceci est équivalent à insérer `#define VAR val` en tête du source. Comme avec un `#define` classique, `VAR` peut contenir des arguments. `-DVAR` correspond à `#define VAR` et est donc équivalent à `-DVAR=1`.

Pour les curieux. L'option `-E` de GCC permet d'afficher sur l'écran le résultat du pré-traitement (elle coupe également les phases suivantes, ce qui fait qu'aucun `a.out` n'est généré). La documentation complète du pré-traiteur est disponible au format Info, accessible par la commande `info cpp`.

2.2.2 La compilation

La compilation effectue les opérations suivantes :
- analyse syntaxique du source,
- vérification de la cohérence des types des variables, fonctions et expressions,
- génération du code assembleur pour chaque fonction et variable globale,
- conversion de l'assembleur vers le binaire (grâce au programme externe `as`).

Le deuxième point est particulièrement important. Il permet de lever certaines ambiguïtés sur la signification du programme (par exemple, `x=y/z;` correspond à une division entière ou flottante selon le type de `y` et de `z`) et de détecter beaucoup d'erreurs (par exemple, une fonction appelée avec trop peu d'arguments). Pour effectuer cette vérification, le compilateur a besoin de savoir le type de chaque variable et le prototype (nombre et type des arguments et type de retour) de chaque fonction. Chaque bibliothèque vient donc avec son lot de *fichiers en-têtes*, d'extension `.h`, qui ne sont rien d'autre que des fichiers sources C contenant des définitions de type et des déclarations de variables et de fonctions. Une directive `#include` suffit alors pour importer ces en-têtes en vertu du recopiage effectué par le pré-traiteur.

L'option `-Wall` permet, entre autres, de détecter les types et prototypes manquants. Il s'agit souvent d'un avertissement et non d'une erreur fatale. La norme stipule en effet que si un prototype manque, celui-ci doit être synthétisé... avec des résultats catastrophiques car le compilateur devine souvent mal. Avec l'option `-Wall`, GCC a au moins le mérite d'avertir le programmeur qu'il s'apprête à faire n'importe quoi.

Le résultat de la compilation est un fichier objet, d'extension `.o`, contenant des objets (fonctions ou variables compilées). Ce fichier objet n'est généralement pas visible, car détruit juste après l'édition de liens (sauf compilation séparée, voir 2.3).

2.2.3 L'édition de liens

Les opérations de pré-traitement et de compilation n'agissent que sur un seul fichier à la fois. La dernière phase, l'édition de liens, combine plusieurs "conteneurs" pour construire un exécutable autonome. Les conteneurs sont des fichiers objets (d'extension `.o`) ou des bibliothèques (d'extension `.a` ou `.so`). Une bibliothèque particulière est toujours considérée : la bibliothèque standard C, contenue dans le fichier `/usr/lib/libc.so`. D'autres bibliothèques peuvent être liées au programme en ajoutant une option de la forme `-l`*bibli* à la commande GCC (il n'y a pas d'espace entre le `-l` et le *bibli*). `-l`*bibli* indique qu'il faut chercher dans un répertoire standard

(généralement `/usr/lib`) un fichier nommé `libbibli.so` ou `libbibli.a`. Notez qu'il n'y a pas d'espace entre le `-l` et le *bibli*, et que ni le préfixe `lib`, ni l'extension de fichier ne doit être précisé. Par exemple `-lm` inclue la bibliothèque mathématique contenant les fonctions `sin`, `cos`, etc. et correspondant au fichier `/usr/lib/libm.so`. L'option `-L`*rep* ajoute *rep* à la liste des répertoires où l'éditeur de liens cherchera les bibliothèques demandées.

L'éditeur de liens s'assure qu'il n'y a pas de référence indéfinie C'est à dire que, si un conteneur définit un objet `a` qui référence un objet `b`, alors la définition de `b` doit être disponible (dans ce conteneur ou un autre). Par référence, on entend : une fonction qui en appelle une autre, une fonction qui utilise une variable globale ou une variable globale qui est initialisé à partir de la valeur d'une autre. L'éditeur de liens s'assure également qu'une fonction `main` est bien définie. Enfin, il ordonne et recopie les différents objets pour construire un fichier binaire `a.out` autonome.

Les bibliothèques dynamiques. J'ai un peu menti en prétendant que le binaire était autonome. En effet, la plus part du temps un fichier objet est lié avec des bibliothèques *dynamiques* (remarquables par leur extension `.so`). Dans ce cas, les objets ne sont pas copiés dans l'exécutable, mais restent référencés. La dernière étape de liaison se fait alors automatiquement au début de l'exécution du programme. Si la bibliothèque n'est plus disponible à ce moment là, c'est la catastrophe ! Cela semble contraignant mais a un certain nombre d'avantages (gain de place disque et en mémoire, possibilité de mettre à jour une bibliothèque sans re-lier toutes les applications, etc.).

Pour aller plus loin. La phase d'édition de liens est effectuée par le programme `ld` appelé automatiquement par GCC sur un fichier objet temporaire issu de la compilation. L'esprit curieux tapera donc `man ld` ou `info ld`. On indique également la commande `objdump -tT` *fichier* qui permet d'afficher la table des objets (définis et référencés) d'un fichier objet, bibliothèque ou exécutable. Enfin, `ldd` *fichier* indique de quelles bibliothèques dynamiques dépend un exécutable ou une autre bibliothèque dynamique.

2.2.4 Exemple

Voici un exemple très simple :

maths.c

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

int main()
{
  printf("sin(pi/4)=%f!\n",sin(M_PI/4));
  return 0;
}
```

compilé puis exécute comme suit :

```
% gcc -Wall -Wextra maths.c -lm
% ./a.out
sin(pi/4)=0.707107!
```

- Les directives `#include` sont indispensables lors de la phase de compilation pour connaître les prototypes de `printf` et de `sin` ainsi que la constante symbolique `M_PI`.
- L'option `-lm` est indispensable lors de l'édition de liens pour retrouver la définition de `sin` dans `/usr/lib/libm.so`.
- Aucune option `-l` n'est nécessaire pour inclure la définition de `printf` car celle-ci est contenue dans la bibliothèque standard C.

2.3 Compilation séparée

Il est souvent souhaitable de décomposer un gros programme en plusieurs fichiers sources `.c`. Cela permet de :

```c
#include <stdio.h>

extern int cube(int);

int main()
{
  printf("10 au cube = %i\n",cube(10));
  return 0;
}
```

main.c

```c
int carre(int x)
{
  return x*x;
}

int cube(int x)
{
  return x*carre(x);
}
```

utils.c

Fig. 5 – Programme multi-fichiers.

- faciliter l'édition,
- faciliter la programmation à plusieurs (chacun a la charge de maintenir un fichier),
- adopter des techniques de programmation modulaire,
- accélérer les recompilations.

Sources multiples. La figure 5 donne un exemple de programme composé de deux sources : `main.c` définit la fonction principale et utilise la fonction `cube` définie dans `utils.c`. Celui-ci se compile simplement en précisant à GCC la liste des fichiers sources :

```
$ gcc -Wall -Wextra utils.c main.c
$ ./a.out
10 au cube = 1000
```

On note un point important : la présence dans `main.c` de la ligne `extern int cube(int);` Comme expliqué en 2.2.2, chaque fichier `.c` va être compilé *séparément* en un fichier objet. Pour bien compiler `main.c`, GCC a besoin de connaître le prototype de la fonction `cube` appelée par `main`. Le mot-clé `extern` indique que la fonction est définie dans un autre fichier (il n'est pas toujours indispensable mais, dans le doute, autant le mettre). De même, il est possible d'accéder à une variable globale définie dans un autre fichier, à condition d'insérer dans le fichier courant une copie de sa déclaration, sans initialiseur mais avec le mot-clé `extern` ajouté.

Fichiers en-têtes. On souhaite éviter autant que possible d'avoir à recopier le prototype d'une fonction dans chaque fichier source qui l'utilise (par paresse, mais aussi pour faciliter les mises à jour). Pour cela, on crée un fichier en-tête `x.h` associé à chaque fichier source `x.c`. Il contiendra le prototype des fonctions de `x.c` à exporter. Cet en-tête sera inclus grâce à `#include "x.h"` par toutes les sources C qui souhaitent utiliser les fonctions fournies par `x`. Notez que le nom de fichier est entre guillemets pour indiquer au pré-traiteur qu'il faut le chercher dans le répertoire courant. Bien que les en-têtes soient des sources C, on adopte par convention l'extension `.h` indiquant qu'ils sont destinés à être inclus et non compilés.

La figure 6 montre ce que cela donne sur notre exemple précédent. L'appel à GCC reste inchangé : `gcc -Wall -Wextra utils.c main.c`. On ne compile que les fichiers `.c`. Notez que `utils.c` inclut le fichier `utils.h`. Ceci permet au compilateur de vérifier, lors de la compilation de `utils.c`, que la *définition* de `cube` et sa *déclaration* externe sont compatibles. Nous augmentons ainsi nos chance de détecter une erreur de type.

Dans la pratique, un fichier en-tête contiendra des définitions de type et de constantes symboliques, ainsi que des déclarations `extern` (sans définition) de variables globales et de fonctions (prototypes). Parfois, une seule en-tête regroupe des déclarations de toute une bibliothèque cohérente composée de nombreux fichiers `.c`. Une utilisation habile des en-têtes permet d'obtenir un programme modulaire. Chaque module est composé d'un fichier `.h` (l'*interface*) et d'un ou plusieurs fichiers `.c` (l'*implantation*). L'interface décrit la liste des variables et fonctions que le module s'engage à fournir et comment les utiliser (grâce aux types et prototypes). Elle cache les détails d'implantation en ne donnant pas la liste des fonctions internes et en laissant certain types abstraits. Il est bien pratique, surtout quand on programme à plusieurs, de pouvoir comprendre ce que fait un morceau de programme sans trop chercher à savoir *comment* il le fait : il suffit de

```c
#include <stdio.h>
#include "utils.h"

int main()
{
  printf("10 au cube = ");
  printf("%i\n",cube(10));
  return 0;
}
```

main.c

```c
extern int cube(int x);
```

utils.h

```c
#include "utils.h"

int carre(int x)
{
  return x*x;
}

int cube(int x)
{
  return x*carre(x);
}
```

utils.c

FIG. 6 – Programme multi-fichiers avec en-tête.

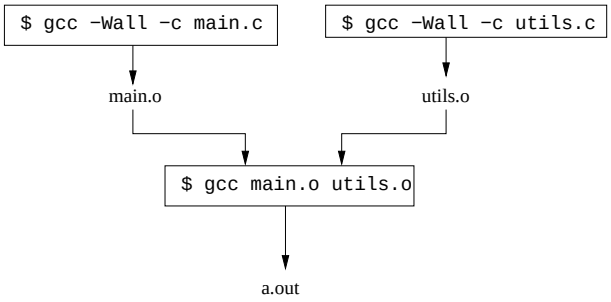

FIG. 7 – Compilation séparée.

ne lire que l'interface. Ceci permet aussi de remplacer une implantation par une autre, de même interface, sans toucher au reste du programme. De plus, l'isolation protège (un peu) le module des fausses manœuvres. La modularité facilite la maintenance des gros programmes et permet de réutiliser des morceaux relativement indépendants dans de nouveaux programmes.

Compilation séparée. Il est possible de décomposer une compilation en deux étapes : génération de fichiers objets (c'est à dire, pré-traitement et compilation) puis édition de liens. D'un côté, l'option `-c` désactive l'édition de liens et force GCC a générer des fichiers objets `.o` (sinon, on l'a vu, ils n'ont qu'une existence éphémère). D'un autre côté, il est possible de passer à GCC une liste de fichiers `.o` lors de l'édition de liens. Voici ce que donne une telle décomposition sur l'exemple de la figure 6 (voir aussi la figure 7) :

```
% gcc -Wall -Wextra -c utils.c
% gcc -Wall -Wextra -c main.c
% gcc utils.o main.o
```

Notez que les options `-Wall` et `-Wextra` étant des options de compilation et non de liaison, il est inutile de les utiliser à la dernière étape. Elles doivent par contre être précisées lors de la compilation de chaque fichier avec `-c`. Il en est de même que les options `-O` d'optimisation (2.4), `-g` de débogage (2.5) et `-D` et `-I` de pré-traitement (2.2.1).

Supposons maintenant que seul `main.c` ait changé depuis la dernière compilation. Pour mettre à jour l'exécutable, il n'est pas nécessaire de reconstruire `utils.o` et on peut se contenter de :

```
                    (utils.o est à jour)
% gcc -Wall -Wextra -c main.c
% gcc utils.o main.o
```

Dans un programme contenant des centaines de fichiers sources, le gain en temps de compilation peut être important. La section 2.6 expliquera de plus comment automatiser cette mise à jour.

2.4 Options d'optimisation

L'optimisation permet d'améliorer la rapidité du programme binaire généré, au détriment du temps de compilation. Les principales options contrôlant l'optimisation sont : `-O` (équivalente à `-O1`), `-O2` et `-O3`. Elles correspondent à des optimisations de plus en plus agressives, et donc, une compilation de plus en plus lente. Si on ne précise pas d'option, le mode par défaut est équivalent à `-O0`, c'est à dire, pas d'optimisation. Ce n'est pas la peine d'essayer `-O4` ou plus car GCC n'implante pas encore de niveau d'optimisation plus élevé que `-O3` !

Pour certains programmes, l'optimisation peut réduire sensiblement le temps d'exécution (par exemple, un programme de calcul numérique). Pour d'autres, elle est totalement inutile (par exemple, un programme qui copie des fichiers ; sa vitesse est limitée par celle du disque dur et non celle du microprocesseur). Enfin, notons qu'il existe certaines techniques de programmation qui facilitent le travail de l'optimiseur, mais nous ne rentrerons pas dans les détails.

Pour aller plus loin. La liste précise des optimisations disponibles est documentée dans `man gcc`. On note en particulier qu'il existe de nombreuses options activant chacune une optimisation élémentaire. `-O`x agit comme une "super-option" qui active plusieurs optimisations élémentaires à la fois. D'autres options, plus obscures, ne fonctionnent que sur certains types de programmes tandis qu'elles ralentissent les autres (pour plus de sûreté, elles ne sont activées par aucune super-option). Attention, la liste des options évolue au fur et à mesure des versions.

2.5 Options de débogage et de profilage

L'option `-g` enrichit le programme binaire d'informations qui facilitent le débogage. Il s'agit de correspondance entre numéro de ligne du source et position dans l'exécutable, ainsi que le nom, le type et la position en mémoire des variables. Toutes ces informations sont normalement perdues lors d'une compilation classique car elles ne sont pas nécessaires à l'exécution du programme final. Ces informations ne sont exploitables que par des outils spéciaux tels que GDB et Valgrind.

Il est fortement déconseillé de combiner l'option `-g` avec une option d'optimisation (`-O` à `-O3`). En effet, l'optimisation déplace, transforme et combine des instructions provenant de différentes lignes du source et supprime les mises à jours inutiles des variables. Les informations données par `-g` ne correspondent plus alors à la réalité de l'exécution, ce qui donne des résultats étranges sous GDB.

L'option `-pg` instrumente le programme binaire pour que l'exécution génère des informations de profilage, utiles pour l'optimisation. Chaque exécution génère alors un fichier intitulé `gmon.out` qui indique le temps pris par l'exécution de chaque fonction. Ce fichier n'est exploitable que par un outil spécial tel que Gprof.

Nous ne dirons rien de plus ici sur le débogage et le profilage. Une fiche entière sera consacrée à l'utilisation de GDB, Valgrind et Gprof.

2.6 Le gestionnaire de compilation make

L'outil make permet d'automatiser la compilation de programmes. Il est particulièrement utile dans le cas de programmes multi-fichiers. make lit un fichier spécial appelé `Makefile` décrivant le projet : les fichiers à compiler et comment les compiler. Voici un exemple de `Makefile` adapté à la compilation séparée du programme de la figure 5 :

Makefile

```
a.out: main.o utils.o
	gcc main.o utils.o

main.o: main.c
	gcc main.c -c -Wall -Wextra

utils.o: utils.c
	gcc utils.c -c -Wall -Wextra
```

Ce `Makefile` est composé de trois *règles* de la forme :

but : *prérequis*
commande

indiquant que pour fabriquer le fichier *but*, il est nécessaire de disposer des fichiers *prérequis* et d'exécuter *commande*. **Attention :** la ligne *commande* de chaque règle doit obligatoirement commencer par une tabulation. Des espaces ne suffisent pas.

Quand on tape `make`, l'outil make cherche à construire le but de la première règle, ici `a.out`. En examinant cette règle, make sait qu'il doit au préalable construire `main.o` et `utils.o`. Supposons que ni `main.o` ni `utils.o` n'existent. make cherchera donc des règles dont les buts sont `main.o` et `utils.o`. Ces règles existent bien et ne dépendent d'aucune autre règle (les fichiers `.c` sont donnés et non construits). make commence donc par exécuter ces règles : `gcc main.c -Wall -Wextra` et `gcc utils.c -Wall -Wextra`. Ensuite, il remonte à la règle initiale, dont les prérequis sont maintenant disponibles, et exécute `gcc main.o utils.o`.

En fait, make est un peu plus malin et n'exécute pas de commande inutile. Pour cela, il compare la date de dernière modification du fichier but à celle de tous les prérequis. Si aucun prérequis n'est plus récent que le but, la commande ne sera pas exécutée. Avant d'effectuer cette comparaison, make s'assure quand même que chaque prérequis est lui-même à jour, c'est à dire, a une date de modification ultérieure à celle de tous ses prérequis, et ainsi de suite. Par exemple, si `utils.c` est modifié, make exécutera, dans l'ordre, `gcc utils.c -Wall -Wextra` puis `gcc main.o utils.o`.

Écrire un `Makefile` pour le projet de la figure 6 est un tout petit peu plus complexe. Il ne faut pas oublier que les fichiers `.o` dépendent aussi de l'en-tête `utils.h` :

Makefile

```
a.out: main.o utils.o
	gcc main.o utils.o

main.o: main.c utils.h
	gcc main.c -c -Wall -Wextra

utils.o: utils.c utils.h
	gcc utils.c -c -Wall -Wextra
```

Afin de simplifier la maintenance d'un `Makefile`, on utilise souvent des variables. Celles-ci se déclarent par `VAR = val` et s'utilisent comme ceci : `$(VAR)`. Voici un exemple qui utilise des variables (très standard) `CC` et `CFLAGS` pour indiquer le compilateur et les options de compilations, `HEADERS` et `OBJS` pour indiquer la liste des en-têtes et des fichiers objets, et `PROJET` pour le nom du fichier exécutable à générer (option `-o` de l'édition de liens).

Makefile

```
CC = gcc
CFLAGS = -Wall -Wextra
HEADERS = utils.h
OBJS = main.o utils.o
PROJET = toto

$(PROJET): $(OBJS)
	$(CC) $(OBJS) -o $(PROJET)

main.o: main.c $(HEADERS)
	$(CC) $(CFLAGS) -c main.c

utils.o: utils.c $(HEADERS)
	$(CC) $(CFLAGS) -c utils.c
```

Il est enfin possible de factoriser les règles pour `main.o` et `utils.o` par une règle générique indiquant comment générer un fichier `.o` à partir d'un fichier `.c`, de la manière suivante :

option	effet	voir
`-Wall`	active des avertissements sur les constructions dangereuses	1
`-Wextra`	active des avertissements supplémentaires	
	vivement conseillés !	
`-o` *fichier*	indique que le fichier généré doit s'appeler *fichier* et non `a.out`	
`-l`*bibli*	lie avec une bibliothèque externe, pas d'espace entre `-l` et *bibli*	2.2.3
	(*e.g.*, `-lm` pour la bibliothèque mathématique)	
`-c`	désactive l'étape de liaison, seule la compilation aura lieu,	2.3
	un fichier objet `.o` sera généré à la place d'un exécutable	
`-O -O2 -O3`	génère du code optimisé, `-O3` étant le plus optimisé	2.4
`-g`	facilite le débogage avec GDB	2.5
	ne pas utiliser avec l'optimisation `-O` ... `-O3`	
`-pg`	permet le profilage avec Gprof	2.5
`-I`*répertoire*	indique où trouver les sources référencés par `#include "`*fichier*`"`	2.2.1
`-L`*répertoire*	indique où trouver les bibliothèques référencés par `-l`*bibli*	2.2.3
`-D`*cst*`=`*valeur*	équivalent à `#define` *cst valeur* en tête de source	2.2.1

FIG. 8 – Quelques options utiles de GCC.

Makefile
```
CC = gcc
CFLAGS = -Wall -Wextra
HEADERS = utils.h
OBJS = main.o utils.o
PROJET = toto

$(PROJET): $(OBJS)
        $(CC) $(OBJS) -o $(PROJET)

%.o: %.c $(HEADERS)
        $(CC) $(FLAGS) -c $*.c
```

Le caractère `%` représente un nom générique (ici, le nom d'un fichier source sans son extension). Dans la commande associée, la variable `$*` reprend la valeur courante de `%`. Notez qu'on a complètement séparé dans ce `Makefile` la partie spécifique au programme (variables) des règles génériques valables pour tout programme C.

Pour aller plus loin. La syntaxe complète des `Makefile`s est décrite dans la page Info de make, accessible par `info make`.

2.7 Récapitulatif

La figure 8 résume les options de compilation et de liaison vues dans ce document.

Pour aller plus loin. La liste des options de GCC peut s'obtenir en tapant `man gcc`. Comme GCC appelle automatiquement le pré-traiteur et l'éditeur de liens, on trouvera regroupées dans cette page les options spécifiques aux trois phases de la compilation. Attention, la liste est très longue ! En effet, outre les options génériques, GCC possèdes des options spécifiques au langage compilé (puisqu'il compile le C mais aussi le C++, l'Objective C, le Java, l'Ada, le Fortran) et des options qui dépendent de la machine (puisque GCC est capable de compiler pour un grand nombre d'architectures très différentes). La page de Man sera donc surtout utile comme référence pour retrouver la signification d'une option particulière. Notez qu'elle ne décrit ni le langage C, ni la bibliothèque standard C.

La page Info, accessible par `info gcc`, reprend ces options de manière plus détaillée. Elle y ajoute la description des diverses extensions au langage C propres à GCC et précise certains choix laissés à la discrétion du compilateur par la norme C.

2.8 Les autres compilateurs

Le compilateur GCC est déjà installé sur toutes la machines Unix (Suns et PCs) des salles élèves. Ce n'est pas forcément le cas sur votre ordinateur personnel, et il vous faudra alors installer un compilateur C : GCC ou autre.

Tous les concepts vus dans cette fichent restent valables pour les autres compilateurs. Seuls changent :
- le nom de la commande utilisée pour compiler ;
- certaines options pointues du compilateur ; celles de la figure 8 (sauf `-Wall` et `-Wextra`) sont par contre relativement universelles ;
- les messages d'erreur et d'avertissement (si la notion précise d'erreur est fixée par la norme, le message lui-même varie ; la notion d'avertissement est, par contre, laissée à la discrétion du compilateur) ;
- sous Windows, les fichiers objets, bibliothèques dynamiques et exécutables ont respectivement l'extension `.obj`, `.dll` et `.exe` (au lieu de `.o`, `.so` et `.out`) ;
- certains utilitaires mentionnés dans les sections les plus pointues (`ldd`, `objdump`, etc.) ne sont pas forcément disponibles.

2.8.1 GCC

Étant donné que les TPs se font avec GCC, on encourage vivement à installer GCC aussi chez vous pour ne pas être dépaysé. Un avantage de GCC est qu'il est disponible sur pratiquement tous les ordinateurs et tous les systèmes d'exploitation.

Sous Linux. Normalement, GCC est déjà installé. Si ce n'est pas le cas, la procédure dépend fortement de la distribution choisie, chacune ayant son propre gestionnaire de paquets. Il est possible que GCC soit déjà installé mais pas les en-têtes de la bibliothèque standard C. Ceux-ci se trouvent parfois dans un paquet à part, nommé `libc-devel`, qu'il faut installer.

Sous Mac OS X. Le logiciel *XCode*, anciennement *Developer's tools*, contient une version de GCC modifiée par Apple. Celui-ci n'est pas installé par défaut mais est fourni sur les CDs ou DVDs du système avec chaque ordinateur.

Sous Windows. Il existe deux distributions contenant GCC pour Windows :
- Le projet "minimaliste" MinGW `http://www.mingw.org/`. Pour plus de confort, on installera le paquet MinGW (compilateur) puis MSYS (terminal en ligne de commande).
- Le projet Cygwin `http://www.cygwin.com/` qui propose, en plus de GCC, la plus part des outils de GNU et Linux pour Windows.

2.8.2 Autres compilateurs

Sous Unix. L'histoire du C est indissociable de celle d'Unix, c'est pourquoi tous les fabricants d'Unix professionnels (Sun, Digital, Silicon Graphics, etc.) fournissent leur propre compilateur C avec leur système. Traditionnellement, ce compilateur est lancé par la commande `cc`. Ainsi, les stations Sun de l'École ont deux compilateurs différents : `cc` de Sun et `gcc` de GNU. Ailleurs, il se peut que seul `cc` soit installé. Les systèmes Unix libres (Linux, FreeBSd, etc) ou dérivés (Mac OS X) utilisent quant à eux le compilateur libre GCC. Les deux commandes `cc` et `gcc` sont alors identiques.

Sous Windows. Le logiciel Visual Studio C++ de Microsoft inclue un compilateur C. Certaines versions allégées (*Express*) sont disponibles gratuitement sur `http://msdn.microsoft.com/vstudio/express/`. Le compilateur est intégré dans environnement graphique de développement comprenant également un éditeur, un débogueur, un gestionnaire de projets, etc. Il est également possible de l'utiliser en ligne de commande.