M2 FEADéP – Programmation C
Outils pour la Programmation en C

Bruno FERRES
École Normale Supérieure de Lyon

Préambule

Dans ce TP, on se propose de prendre en main différents outils qui facilitent grandement le développement (ainsi que le déverminage) en langage C. Les outils en questions ne sont pas des pré-requis pour l'agrégation, mais ils sont d'une grande utilité pour écrire du code propre, sûr et efficace, et peuvent même vous être utile pour exposer le fonctionnement de votre code.
Voici une introduction sommaire aux outils présentés dans ce TP :

— `GNU Make` : `make` permet, de façon générale, de définir des règles pour la génération de fichiers, en explicitant les dépendances de génération afin d'éviter de re-générer des fichiers dont les sources n'ont pas été modifiées.
Les `Makefile` sont notamment très utilisés pour définir des règles de compilation dans des projets complexes, et peuvent vous aider à automatiser la compilation dans vos projets, et ce à un moindre coût d'apprentissage.

— `valgrind` : il s'agit d'un outil permettant d'analyser les allocations et accès mémoire de votre code.
L'outil permet notamment de vérifier que toute mémoire allouée a été correctement désallouée, mais son utilisation peut également pointer des accès mémoires erronés, et donc potentiellement dangereux. *En effet, ce n'est pas parce que vous ne rencontrez aucune SEGFAULT que tous vos accès mémoires sont licites !.*

— `time` : il s'agit d'un utilitaire de base du système, qui permet de mesurer le temps d'exécution d'un programme.
Son utilisation n'est pas spécialement difficile, mais peut permettre notamment de comparer les performances de différentes variations d'implémentation — ce qui peut amener de nouveaux arguments pour justifier ces choix, ou ouvrir une discussion. *Parfois, une analyse statique de la complexité algorithmique de votre implémentation n'est pas suffisante pour motiver vos choix. En effet, il est souvent nécessaire de prendre en compte les détails d'implémentations (accès au cache, matériel sous-jacent, ...) pour avoir une idée des réelles performances de votre code. Un outil comme* `time` *peut mettre en avant ce genre de détails.*

— `gbd` : `gbd` est un des outils les plus courants pour le déverminage en C. Il permet notamment de dérouler l'exécution d'un programme, pas à pas, afin d'analyser l'état du système, et détecter les erreurs dans le code source.
Si vous voulez utiliser `gdb`, il est nécessaire d'ajouter le flag `-g` à la compilation, afin de demander au compilateur de générer des informations supplémentaires dans l'exécutable produit. Ces informations incluent notamment les symboles de debug, ce qui permet de faire la correspondance entre l'exécutable produit et le code source.

1 Introduction à `Makefile`

Dans cette partie, on présente une utilisation sommaire de `make` pour la compilation de sources en C, et également pour des démonstrations de code. Notamment, `make` peut être utilisé pour éviter de rappeler `gcc` pour chaque compilation — pour peu que vous utilisiez des *flags* particuliers, oublier une option dans un appel à `gcc` peut vite arriver !

Rapide présentation : La syntaxe d'une règle `make` est la suivante :

```
my_target: my_dep1 my_dep2
    my_rule
```

Les trois éléments majeurs qui constituent une règle sont donc :

- la **cible** : c'est souvent le fichier qui sera généré par la règle, mais ça peut également être un simple mot clef permettant de lancer une action précise (voir Question 5.1)
- les **dépendances** : ce sont des fichiers (ou d'autres règles) qui sont utilisés pour la génération de la cible actuelle.
- les **règles de génération** : il s'agit des commandes à exécuter pour générer la cible. Souvent, pour de la compilation, il s'agira d'un appel à votre compilateur préféré (`gcc`, `clang`, ...) avec les bons arguments (notamment les *flags*).

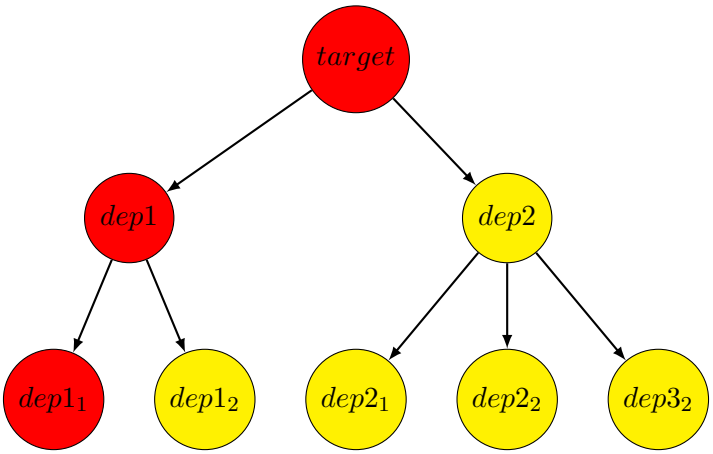

FIGURE 1 – Exemple de résolution d'un arbre de dépendance

Gestion des dépendances L'un des intérêts d'utiliser un `Makefile` est de ne regénérer que les fichiers pour lesquelles des dépendances ont étés modifiées depuis la dernière génération. Par exemple, en Figure 1, la cible *target* possède deux dépendances *dep1* et *dep2*, qui possèdent elles même leurs propres dépendances. Si l'on modifie la dépendance

$dep1_1$, uniquement les dépendances en rouge seront recompilées, ce qui évite notamment de recompiler la cible *dep2*.
Cela permet en général d'éviter des temps de compilation excessifs — attention cependant, si un fichier n'est pas marqué comme dépendance, le modifier n'aura donc parfois aucun impact sur le prochain appel à `make`.

Exercice 1

Dans le répertoire `make` du TP, regarder la première partie du fichier `Makefile`.
Celle-ci comporte notamment **3 règles** :

- `hello_world_1`
- `hello_world_2`
- `bash_hello`
- il y a également une règle `all`, mais nous y reviendrons plus tard...

Pour invoquer ces règles, on peut taper (depuis le répertoire dans lequel se situe le fichier `Makefile`) la commande `make <TARGET>`, pour chacune des règles existantes.
A noter, si vous tapez `make` et que vous utilisez l'auto-complétion, vous pourrez lister l'ensemble des règles disponibles !

Question 1.1:
Exécuter les quatres commandes suivantes :

1. `make hello_world_1`
2. `make hello_world_2`
3. `make bash_hello`
4. `make`

Que remarquez vous par rapport à cette dernière commande ? En regardant le fichier `Makefile` plus en détail, en déduire à quoi sert la règle implicite `all`.

Question 1.2:
Analysez les différences entre les règles `hello_world_1` et `hello_world_2`. Que pouvez vous en déduire sur le rôle des variables implicites `$` et `$<` ?

Exercice 2

Regardez maintenant la seconde partie du `Makefile` (ainsi que le fichier `hello_thread.c`).
Celle-ci contient deux nouvelles règles : `hello_thread_1` et `hello_thread_2`.

Question 2.1:
Essayez de lancer `make hello_thread_1`. Que se passe-t-il ?
Lancez ensuite `make hello_thread_2`. Que se passe-t-il ?

Exercice 3

Regardez la troisième partie du `Makefile`, ainsi que les fichier `hello_mod.c` et `mod.c`.
Il y a deux nouvelles règles : `hello_mod_1` et `hello_mod_2`.

Question 3.1:
Essayez d'exécuter les deux règles, et expliquer la différence. Que pouvez vous en déduire de la variable automatique `$?` ?

Question 3.2:
Essayez de relancer la commande `make hello_mod_2`. Que remarquez vous ? Ajoutez un commentaire dans le fichier `hello_mod.c`, et réessayez. Expliquer le comportement de `make` dans ce cas. Que pouvez vous en déduire sur la gestion des dépendances par `make` (par exemple, `hello_mod.c` et `mod.c` pour la règle `hello_mod`) ?

Exercice 4

Regardez enfin la quatrième partie du `Makefile`, ainsi que le fichier `fact.c`.
Il y a deux nouvelles règles : `fact` et `fact_test`.

Question 4.1:
Tout d'abord, expliquez l'utilisation du type `int64_t` dans ce code.

Question 4.2:
Vous pourrez noter que `fact_test` a pour unique dépendance `fact`. A quoi cela sert-t-il ?

Question 4.3:
En outre, la règle `fact_test` peut vous intéresser particulièrement pour vos démonstrations. En effet, vous pouvez créer de telles règles pour chaque cas de démonstration que vous souhaites utiliser dans vos restitutions.
Lancez la commande `make fact_test` : que remarquez vous ? Expliquer le comportement du programme.
Utiliser de telles règles vous permettra de ne pas vous emmêler dans les paramètres à utiliser pour chaque cas, et vous évitera de montrer le fonctionnement d'un code que vous aurez oublié de recompiler... Je ne peux donc que vous recommander d'utiliser ce genre de mécanisme — ou, en tout cas, de précisément noter quels tests peuvent être lancés pour quelle démonstration.

Exercice 5

Il existe enfin une dernière règle dans le `Makefile` : `clean`.

Question 5.1:
Listez le contenu du répertoire courant avec `ls`, et lancez la commande `make clean`. Que

remarquez vous ?

Question 5.2:
Relancez la même commande. Que remarquez vous, par rapport au comportement explicité en fin de la question 3.2 ?
Cette règle est déclarée dans la section `.PHONY` : cela signifie qu'elle n'est pas une règle de génération — plus précisément, cela signifie que `make` ne vérifie pas si les dépendances d'une telle règle ont changé avant de relancer la règle. Cela est en pratique utilisé pour lancer inconditionnellement certaines règles, ce qui est notamment pratique pour les règles de nettoyage des répertoires de travail !

Remarque : Une utilisation plus avancée de `GNU Make` consiste à générer des fichiers objets `.o` séparés pour chaque fichier source `.c`, et à déléguer la tâche de l'édition des liens à une autre règle du `Makefile` (voir Figure 2). Cela permet notamment de séparer les différentes étapes du flots de compilation, de traiter de façon différente la compilation de certains fichiers sources, ainsi que d'autres opérations qui sont parfois nécessaires lorsqu'on compile un projet suffisamment complexe.

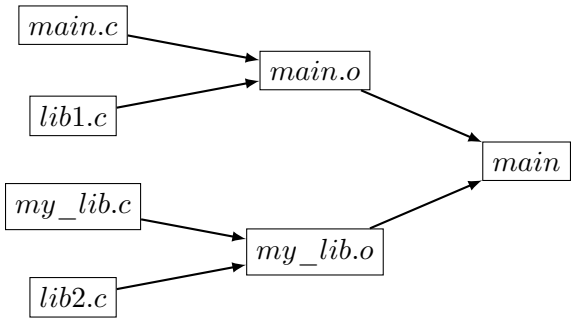

FIGURE 2 – Exemple de compilation d'un *main* C

2 Introduction à `gdb` : une structure d'arbre malencontreuse

Comme précisé en préambule, `gdb` est un outil très pratique pour le déverminage. Afin de vous en convaincre, on se propose d'analyser un code source comportant des *bugs*, d'abord avec les moyens que vous trouverez, ensuite en utilisant un outil éprouvé.
Les sources pour cette section d'introduction sont disponibles dans le répertoire `tree` — en effet, on va s'intéresser à une implémentation de structure d'arbre binaire de recherche ! On fournit également un `Makefile`, pour faciliter la compilation dans les différents exercices.

Exercice 6

On fournit — très gentiment, il est vrai — une première implémentation dans le fichier source `tree.c`.
Vous pouvez compiler cette première version avec `make tree` (ou même `make`).

Question 6.1:
La première tentative d'exécution de l'exécutable fourni devrait vous convaincre qu'il y a (au moins) un problème dans le code source. Tentez d'identifier l'origine des *bugs*, de la manière qui vous semble la plus adapté, et de corriger le code pour qu'il fonctionne.

Question 6.2:
En réalité, c'est exactement à cela que sert l'outil `gdb` — exécuter un programme pas à pas, afin d'en inspecter l'état, et d'identifier les *bugs*. Son utilisation est simple [1] : `gdb <exec>` — attention cependant à bien utiliser le *flag* `-g` à la compilation !
Réessayez l'exercice précédent, mais cette fois ci en utilisant l'outil adapté. Pour se faire, on propose de réaliser les actions suivantes :

1. placer des *breakpoints* aux endroits qui vous semblent opportuns (par exemple, au hasard, à l'entrée de la fonction `insert`).
2. lancer l'exécution du programme avec `run`.
3. exécuter pas à pas le programme après avoir atteint le premier `breakpoint`. À chaque pas, afficher le contenu des variables qui vous semblent intéressantes (par exemple, toujours au hasard, utilisez les commandes `p tree` et `p *tree` pour afficher respectivement la valeur du pointeur `tree` et de son contenu).

Vous devriez arriver (bien plus facilement !) à identifier les erreurs dans le code, et à les corriger.

Question 6.3:
Ce n'est pas parce qu'un code fonctionne qu'il n'est pas buggé... (la métaphore « tomber en marche » est bien plus fréquente qu'on ne le croit).

1. On fournit en annexe un petit guide des commandes disponibles.

On se propose maintenant d'analyser la gestion de la mémoire de notre programme. Pour ce faire, on va utiliser un autre outil bien pratique : `valgrind`.
Son utilisation est encore plus simple : `valgrind ./<exec>`.
Lancez `valgrind` sur l'exécutable non buggé, et analysez la sortie de l'outil. Que pouvez vous en déduire ? Comment utiliser `valgrind` pour améliorer le programme ?

Exercice 7

En fait, une autre version — bien heureusement, non buggée ! — est également fournie dans le fichier `tree_working.c.gpg`. Maintenant que vous avez bien travaillé pour déverminer ce premier programme, on se propose d'utiliser `gdb` pour exécuter pas à pas ce second programme, afin de mieux prendre en main l'outil [2].
Cette nouvelle version est fournie chiffrée, afin d'éviter que vous ne vous en inspiriez pour le premier exercice... Vous pouvez désormais y accéder, en utilisant la commande `gpg -d tree_working.c.gpg -o tree_working.c` — le mot de passe est fourni en annexe.

Question 7.1:
Vous devriez pouvoir compiler la version corrigée en utilisant la règle `make tree_working` du `Makefile` (en utilisant évidemment le mot de passe adéquat).
Une fois fait, on se propose donc de lancer `gdb` sur ce nouvel exécutable, et de réaliser les opérations suivantes :

1. placer un *breakpoint* au début de la fonction `insert`. Pour varier un peu, essayez de faire ça en utilisant la syntaxe `b tree_working.c:xxx`, `xxx` représentant le numéro de la ligne où on souhaite placer un *breakpoint*. Cette option est notamment bien pratique pour placer des *breakpoints* à des endroits précis d'un programme, notamment lorsque ceux-ci ne correspondent pas à des appels de fonction.
2. lancer l'exécution du programme avec `run`.
3. exécuter pas à pas le programme après avoir atteint le premier `breakpoint`. À chaque pas, afficher le contenu de `tree`, de `(*tree)` et de `(**tree)`.

Question 7.2:
L'analyse du contenu de `(**tree)` lors d'un appel bien choisi à `insert` devrait vous rappeler quelque chose : il s'agit du type `node` qui nous permet de stocker notre arbre ! En particulier, vous avez donc accès aux valeurs des champs `(*tree)->left` et `(*tree)->right`.
Admettons que la valeur du fils gauche `left` soit `0xdeadc0de` (attention, si la valeur en question est `0x0`, cela signifie que `(*tree)` n'a pas de fils gauche. Regardez donc lors d'un autre appel à `insert` !). Que se passe-t-il si vous affichez le contenu de `(*0xdeadc0de)` ?

2. `gdb` est parfois également un outil bien pratique pour dérouler un programme dans un but de démonstration. Si cela vous semble adéquat, je ne peux que vous recommander de vous appuyer sur cet outil pour présenter le comportement de votre code, sur certains exemples bien choisis.

Question 7.3:

Il se trouve que `gdb` est un outil très intelligent : il affiche le contenu de la mémoire en fonction des informations de typage qu'il possède. C'est notamment la raison pour laquelle il affiche le contenu de `(**tree)` sous la forme d'une structure correspondant au type `node`. Que se passe-t-il si, cette fois ci, vous *castez* l'un des fils de `tree` dans son type (pour ce faire, utilisez `p (node)(*0xdeadc0de)`) ?
Comparez la structure obtenue avec le résultat de `p (*(*tree)->left)`. Que remarquez vous ?

Question 7.4: (facultatif)

Vous pouvez désormais, si vous le souhaitez, dérouler d'autres fonctions, et analyser le contenu de la mémoire en différents points du programme.
Une telle utilisation de `gdb` est parfois très pratique pour mieux comprendre comment vos programmes sont représentés en mémoire !

3 Un nouvel espoir ? (les tableaux.)

On se propose maintenant d'implémenter cette structure d'arbre d'une façon différente. Une implémentation classique de ce genre de structure s'appuie sur l'utilisation d'un tableau, plutôt que celle d'une structure récursive.

Exercice 8

Cet exercice a pour vocation d'ouvrir une discussion sur les choix d'implémentation d'une telle structure — le plus important étant que vous soyez capable de justifier l'utilisation d'une implémentation par rapport à une autre avec de solides arguments.

Question 8.1:
Comment représenter un arbre dans un tel tableau ? Quelle(s) remarque(s) pouvez vous faire sur son contenu ? Quelle(s) conditions sont nécessaires sur l'arbre ? Quelles sont les principales différences avec l'approche précédente ?

Question 8.2:
Implémentez une telle structure dans un autre fichier. En particulier, vous implémenterez les fonctions suivantes sur votre structure, fonctionnellement équivalentes à celle implémentées dans les exercices précédents :

- `insert`
- `free_tree`
- `depth_print`

N'hésitez pas également à démontrer leurs utilisations sur différents exemples bien choisis. Au besoin, utilisez `make`, `gbd`, `valgrind`, ou tout autre outil qui vous semble pratique !

Exercice 9 (Facultatif)

On se propose maintenant de comparer expérimentalement les performances des deux approches. Pour se faire, on va utiliser la commande `time`. Son utilisation est très simple : `time ./<exec>`.

Question 9.1:
Utilisez `time` sur vos deux approches. Que remarquez vous ?

Question 9.2:
Afin de se rendre vraiment compte des différences notables entre les deux approches, il est nécessaire de faire ces mesures sur des charges plus importantes. Pour chaque approche, écrivez un exécutable plus réaliste, qui génère un arbre conséquent (par exemple, un millier de données ? 10000 ?). Relancez `time` sur chaque approche, et comparez.

Mot de passe : *jaimegdb*

Récapitulatif des commandes `gdb`

Commande	Argument(s)	Effet
`b; br;`	*<function name>*	Ajout d'un *breakpoint* à l'entrée de la fonction.
`break`	*<fichier.c :line>*	Ajout d'un *breakpoint* à la ligne du fichier source.
`r; run`	-	Lance l'exécution du programme.
`c; continue`	-	Continue l'exécution (après un *breakpoint*).
`p; print`	*variable*	Affiche le contenu de la variable.
`s; step`	-	Avance l'exécution d'une instruction [3].
`n; next`	-	Avance l'exécution d'une instruction.
`bt; backtrace`	-	Affiche la pile d'appel actuelle.

On rappelle également que pour pouvoir utiliser efficacement `gdb`, il est nécessaire d'ajouter le *flag* `-g` à la compilation.

3. La différence entre `step` et `next` se fait au niveau des appels de fonction : `step` rentre dans le corps de la fonction, tandis que `next` progresse à l'instruction suivante.