Projet MyMake

Programmation C et Réseau

TP à rendre pour le vendredi 14 octobre 2022 avant 20h00
À faire en binômes

1 Introduction

`make` est un logiciel permettant de construire de façon automatique des fichiers. L'usage le plus fréquent de `make` est la compilation de fichiers sources en fichiers binaires exécutables. `make` utilise une description d'un graphe de dépendance entre les fichiers à construire et leurs prérequis. Cette description est contenue dans un fichier, qui est par défaut nommé `Makefile`. Ainsi, `make` peut reconstruire un fichier en un nombre minimal d'étapes.

Prenons l'exemple d'un projet contenant 100 fichiers sources C, compilés individuellement en 100 fichiers objets, avant d'être combinés en 1 fichier exécutable pendant l'édition des liens. La première compilation nécessitera de compiler l'ensemble des fichiers. Par la suite, si un unique fichier source est modifié, on peut re-compiler tout le projet uniquement en compilant de nouveau le fichier source modifié, puis en effectuant l'édition des liens (ce qui est typiquement beaucoup plus rapide que tout recompiler !). `make` permet d'automatiser ce processus, en utilisant les dépendances entres les fichiers et les dates de dernière modifications des fichiers.

Pour plus de détails sur `make` et des exemples plus complets, on peut se référer à : `https://fr.wikipedia.org/wiki/Make`

L'objectif de ce projet est d'implémenter une version simplifée de `make`, que l'on appelera par la suite `mymake`.

2 Organiser son code

Il est recommandé de découper son code en modules, de façon à gagner en lisibilité et à réduire la taille des différents fichiers. N'hésitez donc pas à structurer votre code en le découpant en modules. **La qualité du découpage sera prise en compte pour l'évaluation**.

3 Description de `mymake`

Vous devez écrire un programme nommé `mymake`. Comme `make`, `mymake` permet d'automatiser la construction de fichiers, par exemple en compilant des fichiers exécutables à partir de fichiers sources. Néanmois `mymake` propose moins de fonctionnalité. La suite du sujet décrit les fonctionnalités attendues. Toute autre fonctionnalités de `make` peut faire l'objet d'extensions possibles.

Il est attendu que l'exécution de `mymake` commence par lire un fichier nommé "`Makefile`", situé dans le même dossier que l'exécutable. Ce fichier contiendra une description du graphe de dépendance permettant d'automatiser la construction. Ensuite, le programme va effectuer un parcours du graphe de dépendance et construire les fichiers nécessaires. `mymake` utilisera la date de dernière modification des fichiers pour déterminer les fichiers nécessitant une reconstruction. La syntaxe du fichier `Makefile` est décrite dans la section suivante.

3.1 Syntaxe du fichier `Makefile`

Le fichier `Makefile` est constitué d'une liste de *règles*. Chaque *règle* est constitué de trois éléments : une *cible*, une liste de *prérequis*, et une liste de *commandes*. Une règle est représentée de la façon suivante :

```
<cible> ":" <prerequis 1> <prerequis 2> ... <prerequis n>
  <commande 1>
  <commande 2>
  ...
  <commande n>
```

Note importante: chaque ligne de commande commence par une caractère tabulation (`'\t'`).

Chaque règle se lit de la façon suivante : pour construire le fichier `<cible>`, il faut d'abord construire tous les fichiers `<prerequis 1>`, `<prerequis 2>`, ..., `<prerequis n>`. Une fois que c'est fait, on peut construire le fichier `<cible>` en exécutant les commandes `<commande 1>`, `<commande 2>`, ..., `<commande n>`.

Un `Makefile` est ensuite constitué d'une liste de règles, situées les unes à la suite des autres. Par exemple, voici le `Makefile` permettant de construire le module `point` vu en cours.

```
main: main.o point.o
  gcc main.o point.o -o main

main.o: main.c point.h
  gcc -c main.c

point.o: point.c point.h
  gcc -c point.c
```

Ce `Makefile` est donc constitué de 3 règles.

- Pour construire l'exécutable `main`, il faut construire les fichiers objets `main.o` et `point.o`. Une fois que c'est fait, on peut construire l'exécutable avec la commande `gcc main.o point.o -o main`.
- Pour construire le fichier objet `main.o`, il faut construire les fichiers sources `main.c` et `point.h`. Une fois que c'est fait, on peut construire l'exécutable avec la commande `gcc -c main.c`.
- La construction du fichier objet `point.o` est similaire.

On notera que les fichiers sources (`main.c`, `point.c` et `point.h`) ne sont pas pas associée à une règle. En effet, ils ne nécessitent pas d'être reconstruits par `mymake`. Du point de vue de `mymake`, construire un fichier qui n'est la cible d'aucune règle ne nécessite aucune opération.

3.2 Utilisation de `mymake`

On attend que `mymake` puisse être utilisé de la façon suivante :

```
./mymake <cible>
```

L'exécution doit construire la cible `<cible>`. Par exemple, `./mymake main` doit construire la cible `main`. Si le nom de la cible est omis, c'est alors la cible de la première règle qui est construite (dans l'exemple, ça serait la cible `main`). On notera qu'en dehors de cela, l'ordre des règles dans le fichier `Makefile` n'a pas d'importance.

▷ **Question 1:** Le dossier `test/` fourni contient un exemple de projet. Proposer un fichier `Makefile` permettant de compiler ce projet. Vous pouvez tester ce `Makefile` avec la commande `make`.

Note: cet exemple de projet n'est là que pour tester pour version de ***`mymake`*** *tout au long des questions. Ce code n'est donc pas à compléter.*

4 Structures de données.

Dans cette partie, on s'intéressera à la définition de plusieurs structures. Pour chacune d'entre elles, on veillera à découper le code en modules. Un module se compose de deux fichiers : un fichier `.h` contenant les déclarations de types et de fonctions, et un fichier `.c` contenant des implémentations associées. Pensez à vous inspirer du modele `point` de l'aide mémoire !

▷ **Question 2:** Définir un module permettant de manipuler une règle en mémoire. Pour rappel, une règle est composée d'un nom de cible, d'un ensemble de prérequis et d'un ensemble de commandes. Définir une structure représentant une règle. Définir des fonctions permettant de manipuler une règle. On attend (entre autres) des fonctions permettant d'allouer et de libérer en mémoire une règle.

▷ **Question 3:** Définir un module permettant de manipuler un ensemble de règles. Définir une structure représentant un ensemble de règles. Définir des fonctions permettant de manipuler un ensemble de règles. On attend (entre autres) des fonctions permettant d'allouer et de libérer en mémoire un ensemble de règles. On attend aussi une fonction permettant d'ajouter une règle à l'ensemble, et une fonction permettant de trouver une règle à partir d'un nom de cible. Pour cette dernière, on se contentera d'une version naïve avec une recherche linéaire.

▷ **Question 4:** Maintenant que votre projet contient plusieurs modules, écrivez un fichier `Makefile` permettant de compiler votre projet. Dans un premier temps, vous pouvez utiliser `make` pour compiler votre projet. *(Une fois le projet terminé, vous pourrez utiliser* ***`mymake`****.)*

5 Lecture du fichier `Makefile`.

Dans cette partie, on s'intéresse à la lecture du fichier `Makefile`. Le but est d'écrire une fonction qui va lire le fichier `Makefile` et renvoyer un ensemble de règles correspondant au contenu du fichier. Pour vous aider, on propose l'algorithme suivant, que vous êtes libre de suivre ou non.

- Ouvrir le fichier `Makefile` et lire ses lignes une par une avec la fonction `getline`.
- Pour chaque ligne :
  - Si la ligne **ne** commence **pas** par le caractère tabulation, on découpe la ligne autour des caractères blancs (`' '`, `'\t'`, ...) ainsi que du caractère `':'`. Ce découpage peut se faire avec l'aide de la fonction `strtok`. Puis, on ajouter à l'ensemble de règles une nouvelle règle donc le nom de cible et les prérequis résultent de ce découpage. Cette nouvelle règle devient la *règle courante*.
  - Si la ligne commence par le caractère tabulation et qu'il y une *règle courante*, alors le restant de la ligne est ajouté aux commandes de la règle courante.

Note: la gestion d'erreurs en cas de fichier `Makefile` mal formé peut être fastidieuse. Ainsi, on fera l'hypothèse que les fichiers `Makefile` sont toujours *bien formés. Évitez donc de passer trop de temps à essayer de gérer les différents cas d'erreur.*

▷ **Question 5:** Écrire une fonction implémentant la lecture du fichier `Makefile`.

6 Construction de cible

Dans cette partie, on propose d'implémenter l'algorithme permettant de générer une cible, à partir d'un ensemble de règles. On implémentera dans un premier temps une version naïve, qui reconstruit toujours l'ensemble des cibles nécessaire, puis une seconde version qui utilise la date de dernière modification d'un fichier pour déterminer si sa construction est nécessaire.

Au cours de cette partie, vous trouverez peut-être pertinent d'ajouter de nouvelles fonctions aux modules déjà existant, vous prendrez soin alors de l'indiquer.

6.1 Version naïve

Dans cette partie, on ne s'intéresse pas aux dates de dernière modification. On propose de nouveau un algorithme décrivant la construction d'une cible. Vous êtes libre de le suivre ou non.

Pour construire la cible `c` :

- S'il existe une règle dont le nom de cible est `c`, alors:
  - Alors, construire récursivement chacune des cibles correspondant aux prérequis. Puis exécuter chacune des commandes. On peut faire cela à l'aide de la fonction `system`.
- Sinon (aucune règle ne correpond) :
  - Si `c` est un fichier, il n'y a rien à faire pour construire la cible.
  - Sinon, terminer avec un message d'erreur.

Note: cet algorithme ne termine pas si le graphe de dépendance contient un cycle. Il conviendrait donc de s'assurer au préalable que le graphe ne contient aucun cycle. On laisse ce problème en extension.

▷ **Question 6:** Implémenter une fonction `make_naive`, qui prend en paramètre un ensemble de règles et un nom de cible, et ne renvoyant rien. Cette fonction doit effectuer la construction de la cible.

6.2 Version avec date de dernière modification

On cherche maintenant à réduire le nombre de cibles à reconstruire. Pour cela, on part de l'observation suivante : pour chaque règle, si aucun des prérequis n'a été modifié depuis la dernière construction, il est inutile de reconstruire la cible.

▷ **Question 7:** Implémenter une fonction `make` prenant en compte les dates de dernière modification. Pour déterminer cette date, on utilisera la fonction `stat`. Vous prendrez soin de détailler l'algorithme que vous utilisez.

7 Programme complet

▷ **Question 8:** Combinez tous les modules et les fonctions précédentes pour construire un programme complet. En particulier, vous utiliserez les paramètre de la fonction `main` pour répondre aux attentes de la section 3.2

▷ **Question 9:** Assurez-vous que vous pouvez maintenant utiliser `mymake` pour compiler votre projet !

8 Extension

Nous avons maintenant une version minimale qui imite un sous-ensemble de l'outil `make`. Pour aller un peu plus loin, on attend l'implémentation d'une (**et une seule !**) extension. Le choix de cette extension est libre, mais nous proposons quelques idées ci-dessous :

- Améliorer la fonction de recherche dans un ensemble de règles de la section 4. On pourra par exemple utiliser une recherche par dichotomie dans un ensemble de règles triées par ordre lexicographique.
- Construire en mémoire la structure de graphe de dépendance. Cela nécessitera la création d'une nouvelle structure (et d'un module).
- Le graphe décrit dans le fichier `Makefile` doit nécessairement être acyclique. Implémenter une détection de cycle avant la construction des cibles.
- `make` propose des variables automatiques, comme `$@`, `$<` ou `$^` (il y en a d'autres) qui font respectivement référence au nom de la cible, au premier prérequis, et à la liste des prérequis d'une règle. Ces symboles peuvent être utilisés dans les commandes. Pour l'exemple de `Makefile` de la section 3.1, cela donnerait :

```
main: main.o point.o
  gcc $^ -o $@

main.o: main.c point.h
  gcc -c $<

point.o: point.c point.h
  gcc -c $<
```

  Modifier votre programme pour gérer ces variables automatiques.
- À la place de la date de dernière modification d'un fichier pour déterminer si sa reconstruction est nécessaire, on peut utiliser un système de *hash*. L'idée est la suivante : chaque fois que l'on utilise un fichier en prérequis, on calcule un hash du fichier, que l'on sauvegarde. Par la suite, on peut ensuite déterminer si le fichier a été modifié en calculant son hash et en le comparant au hash précedemment sauvegardé.
- *(hors programme et difficile!)* Lorsque l'on passe le paramètre `-j <n>` où `n` est un entier (par exemple `-j 4`), `make` construit les différents cibles en parallèle. `make` lance ainsi des processus en parallèle à l'aide des fonction `fork`, `exec` et `wait`, en se limitant à un nombre maximum de `n` processus.
- Toute autre idée est la bienvenue !

Rendu et attendus

Un rapport par binôme (ou trinôme si nombre impair) est attendu. Les rapports et le source des programmes doivent être envoyés dans une archive `tar` compressée par mail aux **deux** adresses suivantes: `remi.hutin@ens-rennes.fr` et `nicolas.bailluet@ens-rennes.fr`. Le rapport doit être rendu au format pdf, et l'intégralité de votre programme doit être écrit en C. Si vous avez utilisé git comme conseillé, vous pouvez donner l'adresse de ce git à la place de l'archive (en vous assurant que nous y avons accès).

Vous porterez un soin particulier à l'écriture du code. *Pensez à nettoyer pour ne pas rendre un brouillon.* Le code doit être commenté et bien écrit pour être facilement lisible. Il est rappelé que l'on écrit un programme pour que d'autres humains puissent le lire, et (accidentellement seulement) pour que les machines puissent l'exécuter. Nous compilerons votre programme avec quelques `-W???` bien choisis pour une première vérification syntaxique, mais nous ne l'exécuterons pas. Nous le lirons en revanche attentivement. Quelques conseils se trouvent sur wikipédia[1].

Votre rapport doit apporter une réponse claire et détaillée à chaque question du sujet, sans reprendre trop de code. La note finale tiendra compte à la fois de votre rapport et de votre code. Relisez-vous avant d'envoyer votre travail! Votre rapport doit comporter (au moins) les parties suivantes :

Introduction : quelques mots pour présenter ce projet (ce qu'on va faire et pourquoi c'est intéressant).

Une réponse construite pour chaque question : expliquer vos observations, pourquoi ça se passe comme ça, quel est le mécanisme observé, à quoi il sert, pourquoi on se pose cette question... Soyez pédagogiques!

[1] Notions basiques sur la lisibilité d'un code C : `http://fr.wikibooks.org/wiki/Conseils_de_codage_en_C/Lisibilit%E9_des_sources`.

Synthèse : une conclusion sur vos observations, expérimentations et résultats, etc. Éventuellement, si vous en avez, des commentaires sur ce qui pourrait être amélioré pour l'année prochaine ou des idées de bonus que vous auriez voulu implémenter dans ce projet.

Bibliographie: Donnez la liste de toutes les sources (sites, livres ou individus) qui vous ont aidé, avec quelques mots de ce que vous en avez retiré. Attention, la frontière est mince entre *l'oubli* de certaines sources et le plagiat. N'oubliez rien, ne trichez pas.

Par tricher, nous entendons notamment :

- Rendre le travail de quelqu'un d'autre avec votre nom dessus ;
- Obtenir une réponse par Google™ ou autre et mettre votre nom dessus ;
- Récupérer du code et ne changer que les noms de variables et fonctions ou leur ordre avant de mettre votre nom dessus (*"moving chunks of code around is like moving food around on your plate to diguise the fact that you havn't eated all your brussel sprouts"*) ;
- Permettre à un collègue de *s'inspirer* de votre travail. Assurez vous que votre répertoire de travail n'est lisible que par vous même.

Il est plus que très probable que nous détections les tricheries s'il y en a. Chacun a son propre style de programmation, et personne ne code la même chose de la même manière. De plus, il existe des programmes très efficaces pour détecter les similarités douteuses entre copies (MOSS, `http://theory.stanford.edu/~aiken/moss/`). En cas de litige grave, seul un historique progressif de vos travaux (comme en offre git) constitue une preuve de votre innocence. *Commit soon, commit often.*

En revanche, il est conseillé de discuter du projet et d'échanger des idées avec vos collègues. Pour ne pas franchir la ligne jaune, **ne lisez jamais de code écrit par un autre groupe, et ne permettez pas aux autres groupes de lire votre code**. Vous ne pouvez rendre que du code écrit par vous-même, et vous devez détailler brievement vos sources d'inspiration sur internet dans la partie bibliographie de votre rapport. Soyez spécifique: si par exemple vous avez trouvé des réponses sur Stack Overflow, pointez les pages spécifiques utilisées.

Votre travail est à rendre pour le **vendredi 14 octobre 2022 avant 20h** . **Tout retard sera sanctionné : un point en moins par heure de retard entamée**.