INF3173-241-TP1 - Compilation automagique

Ce TP vise à réaliser l'outil `justmake` qui compile des sources en C sans avoir à écrire un fichier `Makefile`.

Étape 1 : Spécifications de base

Votre programme doit réaliser la fonctionnalité suivante.

- On liste les fichiers source ayant l'extension `.c` contenu dans un répertoire spécifié en argument. La recherche n'est pas récursive.
- On compile chaque fichier source pour produire un fichier objet ayant l'extension `.o`. Par exemple, le fichier source `foo.c` produit le fichier objet `foo.o`.
- Finalement, on fait l'édition de lien de tous les fichiers objets pour produire l'exécutable final. Le nom de l'exécutable par défaut est `programme.exe` et peut être modifié par l'utilisateur en argument.
- Tous les fichiers produits par l'outil (les fichiers objets et l'exécutable final) doivent être écrits dans un répertoire de compilation disinct des sources. Ce répertoire est par défaut `jmlbuild`. Si le répertoire de compilation n'existe pas, il est créé. Vous pouvez assumer que le répertoire parent existe (et donc pas besoin de créer des répertoires récursivement). Il est une bonne pratique de séparer les sources et les fichiers générés pour bien les distinguer et éviter d'ajouter des fichiers générés dans un gestionnaire de révision, par exemple.
- Il doit être possible de spécifier en ligne de commande des options de compilation et d'édition de liens (comme des librairies) avec les options `-c` et `-l`. Voir l'exemple de compilation de tetris ci-bas.

Les répertoires `demo` et `tetris` sont fournis pour tester.

Les options de `justmake` sont les suivantes:

```
-j nombre de processus maximum en exécution simultanément (par
défaut 1)
-k activer keep_going (par défaut 0, utilisé à l'étape 2)
-s répertoire des sources (obligatoire)
-b répertoire de compilation (par défaut jmbuild)
-c cflags (options passées au compilateur)
-l libs (librairies passées à l'éditeur de lien)
-p nom du programme (par défaut programme.exe)
```

Voici un exemple d'exécution de base pour compiler les deux exemples. À noter que vous avez besoin de librairies supplémentaires pour compiler le jeu.

```
# Compilation du programme justmake (vous pouvez choisir n'importe
quel répertoire de compilation)
cmake -G Ninja -S . -B build
```

```
cmake --build build

# Le chemin du programme compilé est: build/bin/justmake
# Astuce: ajouter justmake dans le PATH du shell courant facilite
l'utilisation
source build/env.sh

# Compilation demo
justmake -s demo -b build-demo

# Exécution demo
./build-demo/programme.exe

# Librairies requises pour compiler tetris
sudo apt install pkg-config libsdl2-dev libglew-dev

# Compilation du jeu tetris
# on utilise pkg-config pour spécifier les librairies facilement
justmake -s tetris -b build-tetris -p tetris.exe -c "$(pkg-config --
cflags glew sdl2 opengl)" -l "$(pkg-config --libs glew sdl2 opengl)
-lm"

# exécution
./build-tetris/tetris.exe
```

> Note: Techniquement, il ne serait pas nécessaire de recompiler le programme si les sources et les entêtes n'ont pas été modifiés depuis la dernière compilation. Par simplicité pour ce travail, on recompile toujours le programme au complet, même si les fichiers sources n'ont pas changés.

Étape 2 : Compilation multiprocesseur

En utilisant tous les processeurs de l'ordinateur, on peut réduire le temps de la compilation par rapport à compiler les fichier un à la fois. Pour cela, on spécifie en ligne de commande le nombre de processus de compilation à lancer au maximum simultanément. Par exemple, si on spécifie `-j 4`, alors on doit lancer jusqu'à 4 processus de compilation en même temps. Autrement dit, on lance un processus si la limite n'est pas atteinte. Si la limite est atteinte, on doit attendre qu'un processus se termine avant de démarrer le suivant. Cette procédure est répétée tant qu'il y a des fichiers à compiler.

Si une erreur de compilation survient avec un fichier source, l'option `keep_going` indique comment on gère l'erreur. Si `keep_going` est vrai, alors on continue à compiler les autres fichiers, mais on ne doit pas faire l'édition de lien. Si `keep_going` est faux, alors on doit cesser de lancer de nouvelles commandes, sans toutefois interrompre les commandes déjà lancées. Dans tous les cas, on doit attendre que toutes les commandes déjà lancées se terminent.

Détails d'implémentation

Tout votre travail doit être entièrement réalisé dans le fichier `src/jml.c`. Vous pouvez modifier, créer des fonction ou des structures comme vous le voulez dans ce fichier. Cependant, l'interface `jml.h` ne doit pas être modifiée, sinon la correction pourrait échouer.

Le traitement des arguments est fourni (dans le fichier `src/main.c`), ainsi que des utilitaires pour traiter les chaines et construire des commandes. Pour cela, référez-vous au fichier `jml.h` qui contient le détail des fonctions fournies.

Voici les fonctions que vous devez implémenter:

- jml_main : fonction principale (les options sont déjà traitées et fournies)
- jml_listdir : lister les fichiers source
- jml_command_exec_one : exécuter une tâche unique
- jml_command_exec_all : exécuter une liste de tâche

Se référer au fichier source, qui contient la documentation et des indications supplémentaires.

Vous avez également accès à 2 librairies pour vous aider dans le développement. Ces deux librairies sont fournies. Ne modifiez pas les fichiers de ces librairies, ni aucun autre fichier.

- cwalk : librairie de manipulation de chemins d'accès
- list: librairie liste chainée

Validation

Des tests sont fournis pour vérifier le fonctionnement général. La vérification se fait avec la commande `sleep` et en mesurant le délai d'exécution, pour donner une indication si l'exécution fonctionne comme attendu. Aussi, on vérifie que les erreurs de commandes sont gérées correctement.

Les tests sont réalisés avec Catch2 dans le fichier `test/test_justmake.cpp`. Vous n'avez pas besoin d'écrire des tests, mais vous pouvez les modifier ou en ajouter selon votre choix, mais ils ne seront pas considérés pour la correction.

L'exécutable des tests est `bin/test_justmake`. Vous pouvez exécuter les tests en ligne de commande, ou directement depuis QtCreator.

Les tests unitaires ne vérifient pas que votre programme fonctionne dans son ensemble. Vous devez tester manuellement la compilation d'un programme réel, tel que le projet demo et le jeu tetris.

Vous pouvez vérifier les fuites de mémoire une fois l'implémentation complète et que les tests passent. Par exemple:

```
valgrind --leak-check=full justmake -s demo
valgrind --leak-check=full test_justmake
```

Correction sur Hopper

Votre travail sera corrigé sur le système Hopper. Le système vous permet de former votre équipe pour ce TP. Faire l'archive avec `make remise` ou `ninja remise`, puis envoyer l'archive produite sur le serveur de correction. Votre note sera attribuée automatiquement. Vous pouvez soumettre votre travail plusieurs fois et la meilleure note sera conservée. D'autres détails et instructions pour l'utilisation de ce système seront fournis.

Barème de correction (sujet à ajustements)

- Lister les fichiers sources: 10
- Exécution séquentielle: 30
- Exécution multiprocesseur: 30
- Fonctionnement global correct: 30
- Respect du style (voir fichier .clang-format): pénalité max 10
- Qualité (fuite mémoire, gestion d'erreur, avertissements, etc): pénalité max 10
- Total sur 100 points

Le non-respect de la mise en situation pourrait faire en sorte que des tests échouent. Il est inutile de modifier les tests pour les faire passer artificiellement, car un jeu de test privé est utilisé. Se référer aux résultats de test pour le détail des éléments vérifiés et validés. En cas de problème, contactez votre enseignant.

Bon travail !

Note sur les logiciels externes

Le code intègre les librairies et les programmes suivants.

- https://github.com/catchorg/Catch2
- https://github.com/likle/cwalk
- https://github.com/superjer/tinyc.games
- https://github.com/andyleejordan/c-list