Qu'est ce qu'un projet C++ ?

Essentiellement, c'est un ensemble de fichiers `.hpp` et `.cpp` qui, lors des différentes étapes de la compilation, permettent de fabriquer un (et parfois plusieurs) exécutable(s).

Comme vous le savez, compiler *globalement* un projet consiste à compiler *individuellement* chacun de ses composants puis à procéder à leur édition de liens, le tout dans un ordre particulier : les compilations d'abord (dans un ordre a priori quelconque) puis l'édition de liens (qui produit l'exécutable).

Vous savez également que si vous modifiez un fichier source `truc.cpp`, il va falloir recompiler ce dernier pour produire le nouveau fichier objet `truc.o`, puis refabriquer l'exécutable, et ce dans cet ordre. Vous savez que si vous modifiez un fichier `truc`.hpp, il va falloir recompiler chacun des fichiers qui l'incluent. Vous savez également qu'il n'est pas utile de recompiler d'autres fichiers sources que vous n'avez pas touché... puisqu'ils n'ont pas changé !

L'outil `make`

Imaginez maintenant qu'un outil automatisé soit capable de déterminer quels fichiers ont changé et lesquels n'ont pas changé... il serait capable de lancer les compilation automatiquement ! Cet outil existe : c'est la commande `make` (sous Unix) ! Sous Windows, elle se nomme `mingw32-make`, mais c'est la même commande.

La commande `make` est capable d'automatiser n'importe quelle compilation, indépendamment du (ou des) langage(s) de programmation utilisé(s), puisqu'elle ne regarde à aucun moment le contenu réel des fichiers.

Le fichier `Makefile`

Le fichier `Makefile` (avec une majuscule, c'est important sous Linux et Mac OS X) est un fichier de texte (comme l'est aussi un fichier C++), éditable avec Notepad++ ou gedit. Vous devez y taper un certain nombre d'indications décrivant votre projet C++.

Ce fichier `Makefile` comporte deux types de lignes différents :

- des lignes décrivant une dépendance entre les fichiers (par exemple, le fichier `ratio.o` dépend des fichiers `ratio.cpp` et `ratio.hpp`, c'est-à-dire qu'il faut refabriquer le premier si l'un des deux autres au moins a été modifié ;
- des lignes de compilation qui indiquent comment refabriquer le fichiers lorsque la dépendance (ci-dessus) l'a indiqué.

Pour commencer à parler sur quelque chose de concret, lisez l'exemple suivant :

```
tp7.exe : tp7.o Ratio.o
--->| g++ -Wall -o tp7.exe tp7.o Ratio.o

Ratio.o : Ratio.cpp Ratio.hpp
--->| g++ -Wall -c Ratio.cpp

tp7.o : tp7.cpp Ratio.hpp
--->| g++ -Wall -c tp7.cpp

clean :
--->| del -f tp7.exe *.o
```

Exemple de fichier `Makefile`.

Les lignes de dépendances comportent un « `:` » qui sépare la *cible* (en anglais, *target*) (à gauche), c'est-à-dire le fichier qui sera produit, de ses dépendances (*prerequisites*), c'est-à-dire de la liste des fichiers sources nécessaires à sa production, et dont un changement quelconque va nécessiter de refabriquer la cible.

Sous chaque ligne de dépendance se trouve une ligne de compilation, reconnaissable au fait qu'elle commence par un caractère tabulation (indiqué ci-dessus par "`--->|`" dans l'exemple ; attention, vous devez pas mettre des tirets + supérieur à + barre verticale mais bel et bien un caractère tabulation !). Cette ligne de compilation est identique à celles que vous utilisez depuis le début des TP de C++.

?

Par exemple, le but `Ratio.o` dépend de `Ratio.cpp` et `Ratio.hpp`, et il déclenche pour reconstruire `Ratio.o` la commande "`g++ -c Ratio.cpp`".

Attention, vous devrez vous assurer que votre éditeur de texte ne convertit pas les tabulations en un ou plusieurs espaces sinon la commande `make` ne saura pas identifier les lignes de compilation !

Le premier but que vous indiquez (dans l'exemple, c'est `tp7.exe`) est le but par défaut : celui qui sera déclenché si vous tapez « `make` » (tout court). Vous pouvez également déclencher un autre but explicitement : par exemple, taper « `make Ratio.o` » procédera à toutes les actions nécessaires pour produire le fichier `Ratio.o`. Souvent, quand plusieurs exécutables sont à produire, on crée un but artificiel nommé `all` qui permet de les lister tous.

Vous vous demandez peut être comment la commande `make` sait si un fichier a changé ? C'est simple : elle regarde la date de dernière modification de chaque fichier d'une ligne de dépendance. Ainsi pour une dépendance de la forme "`A : B C D`", `make` va vérifier que la date de dernière modification de `A` est plus récente que celles de `B`, `C` ou `D`. Ainsi, si on modifie disons `C`, la date de `C` sera plus récente que celle de `A` et `make` saura qu'il faut refaire `A`. Pour cette raison, il est parfois important de s'assurer que votre ordinateur a son horloge bien réglée ! C'est en particulier important lorsque vous travaillez sur un disque en réseau, sur plusieurs postes, voire sur un disque externe ou une clé USB. Il est en effet assez courant d'entendre `make` se plaindre qu'un fichier (créé sur une autre machine par exemple) a une date de dernière modification qui est dans le futur (de la machine sur laquelle vous êtes) !

Le but "`all`" dont on a parlé plus haut est un peu particulier en ce sens qu'il ne correspond pas *réellement* à un fichier dans le répertoire courant. Du coup, comme `make` ne peut pas trouver la date de dernière modification de ce fichier (puisqu'il n'existe pas !) alors la dépendance indiquée est *toujours* considérée comme non-satisfaite, et est reconstruite systématiquement. C'est également le cas du but "`clean`" qui se trouve en fin de l'exemple, et dont l'intérêt est expliqué plus bas.

Vous devriez comprendre que contrairement à un script (.bat sous Windows ou script shell sous Unix) qui referait la *totalité* de la compilation à chaque fois, vous gagnez un temps considérable en ne faisant *que ce qui est nécessaire*... surtout si le projet est gros !

La ligne de compilation est une commande shell quelconque : il n'est pas obligé que ce soit `g++` qui soit déclenché. Cela peut être n'importe quelle commande. Vous pouvez même déclencher plusieurs lignes de commande à condition de bien les débuter par le caractère tabulation. Ainsi, à la fin du fichier, comme on le fait souvent, on a rajouté un but « `clean` » qui permet de déclencher l'effacement de l'exécutable et des fichiers objets, afin de ne laisser que les fichiers sources. Pour le déclencher, il suffit de taper la commande « `make clean` ». C'est très pratique pour faire du ménage, avant peut être de recompiler tout à neuf en tapant « `make` » ou « `make all` ». Évidemment, la commande « `del` » indiquée ici n'existe que sous Windows, et il faut lui substituer la commande « `rm -f` » (l'option `-f` permet de forcer l'effacement) si vous êtes sous Linux.

Attention : si vous vous trompez dans les dépendances (la plupart du temps, en en oubliant), alors `make` risque de ne pas recompiler tout ce qu'il faut ! N'ayez donc pas une confiance aveugle en ce que vous avez tapé dans le `Makefile` ! Les erreurs de dépendances sont assez subtiles à trouver et corriger. Schématiquement, pensez à indiquer pour chaque `.cpp` qu'il dépend de tous les fichiers `.hpp` que vous incluez (les vôtres, entre guillemets, pas ceux du système, entre inférieur-supérieur, car ils ne changent jamais !).

Allons plus loin...

Vous pouvez fort bien réaliser vos `Makefile` maintenant grâce aux consignes simples qui viennent de vous être données. Cependant, on peut également profiter de certaines fonctionnalités supplémentaires...

Variables du Makefile

Nous pouvons par exemple déclarer des variables pour le `Makefile` afin de centraliser certaines choses. L'exemple précédent pourrait ainsi devenir :

?

```
CXX=g++
CXXFLAGS=-Wall -g
RM=del
#RM=rm -f

tp7.exe : tp7.o Ratio.o
--->| $(CXX) -o tp7.exe tp7.o Ratio.o

Ratio.o : Ratio.cpp Ratio.hpp
--->| $(CXX) $(CXXFLAGS) -c Ratio.cpp

tp7.o : tp7.cpp Ratio.hpp
--->| $(CXX) $(CXXFLAGS) -c tp7.cpp

clean :
--->| $(RM) tp7.exe *.o
```

Exemple de fichier `Makefile` amélioré.

Nous avons utilisé trois variables : l'une `CXX` contient le nom du compilateur, ce qui est parfois pratique quand on en a plusieurs différents, la deuxième `CXXFLAGS` contient les différentes options de compilation qu'on choisit de marquer à chaque compilation (ici, `-Wall` pour forcer le maximum de détection d'erreurs et `-g` pour permettre le débogage plus facilement, voir les explications sur `gdb`). Et la dernière `RM` contient le nom de la commande à utiliser pour effacer un fichier (il suffit de permuter le commentaire de début de ligne pour adopter la version Unix ou Windows de la commande).

Dépendances génériques

Il est possible d'indiquer à `make` des règles de dépendance et de compilation génériques, c'est-à-dire correspondant à un motif de nom de fichier (par exemple tous les fichiers portant l'extension `.cpp`) plutôt qu'à un seul fichier. Par exemple, les lignes suivantes permettent la compilation de n'importe quel fichier `.cpp` en le fichier correspondant `.o` qui dépend du fichier `.cpp` (forcément !) et d'une série de fichiers `.h` (les mêmes pour tous, ce qui est parfois excessif, et déclenchera des compilations inutiles) :

```
CXX=gcc
CXXFLAGS=-Wall -g
INCLUDES=truc1.hpp truc2.hpp truc3.hpp

.cpp .o : $(INCLUDES)
--->| $(CXX) $(CXXFLAGS) -c $<
```

Exemple de règle générique de `Makefile`.

Le but `.cpp.o` s'applique à tout fichier `.cpp` à partir duquel on doit produire un `.o`. La commande `make` remplace automatiquement la pseudo-variable « `$<` » par le nom du premier fichier source de la liste de dépendance (par exemple `truc.cpp`). Il existe une grande quantité de pseudo-variables comparables. Pour en avoir un aperçu, lisez la documentation de `make` (voir plus bas).

Vous pouvez également utiliser quelques autres pseudo-variables :

- `$@` est la cible de la règle (celle indiquée avant les deux-points) ;
- `$^` est la liste de *toutes* les dépendances d'une cible (et pas seulement la première comme `$<`).

Attention : n'abusez pas des règles génériques si vous ne les maîtrisez pas : on arrive vite à ne plus savoir qui fait quoi !

Génération automatique des dépendances

Il est également possible de générer une liste de dépendances automatiquement avec `g++` (ou `gcc` en C) avec l'option `-M`. Par exemple, la ligne de commande "`g++ -M truc.cpp`" va produire à l'écran un fichier qui liste toutes les dépendances (les inclusions) du fichier `truc.cpp`. Dans cette liste, se trouvent *absolument tous* les fichiers include, y-compris ceux fournis avec le système d'exploitation ou le compilateur... qui changent r ? ment ! L'option `-MM` permet de ne lister que les inclusions locales, c'est-à-dire les fichiers inclus entre guillemets.

Rien n'empêche ensuite de rediriger la sortie de cette liste vers un fichier (traditionnellement un fichier appelé `.depend` sous Unix pour en faire, à

cause du "." initial, un fichier caché) et de l'inclure dans le `Makefile` (puisque l'inclusion existe aussi pour `make`).

```
# Inclusion de la liste des dépendances :
include .depend
# Pour recréer ce fichier, taper dans une console :
# g++ -MM *.cpp >.depend
```

Exemple de `Makefile` incluant une liste de dépendances créée systématiquement

Vous devrez donc régénérer les dépendances *manuellement* quand c'est nécessaire, c'est-à-dire à chaque fois que vous ajoutez un fichier *.cpp ou que vous ajoutez ou supprimez une inclusion dans ces mêmes fichiers.

Compléments sur `make` et les `Makefile`

- La définition de Wikipedia fr : http://fr.wikipedia.org/wiki/GNU_Make ;
- Un petit cours sur make : http://gl.developpez.com/tutoriel/outil/makefile/ ;
- Un howto en français http://iihm.imag.fr/blanch/howtos/GNUMake.html ;
- La documentation de GNU make (en anglais) : http://www.gnu.org/software/make/manual/.

Au delà des `Makefile`

D'autres outils existent pour automatiser la compilation. On peut citer :

- `autoconf` + `automake` : ils permettent de régler votre programme source pour l'adapter à la machine sur laquelle a lieu la compilation. Cet ensemble de scripts, issu de très nombreuses années de développement, est extrêmement puissant... mais difficile à comprendre ! Les documentations ne suivent pas toujours le rythme du développement, et l'architecture générale du système en plusieurs fichiers rend son usage souvent assez obscur ! Ces outils produisent un script Bourne shell nommé `configure` qui lorsqu'il est lancé teste de (très) nombreuses choses, et finalement produit le fichier `Makefile`, que vous n'avez plus qu'à utiliser en tapant `make`.
- `smake` : combine des fonctionnalités de `make` et de `autoconf` + `automake`.
- `cmake` : un peu le même principe que `smake` et désormais très utilisé (et un tutoriel pour débuter).
- `qmake` : automatise la génération du `Makefile` pour un projet utilisant la bibliothèque Qt utilisée pour faire des applications portables (Windows, Linux, Mac OS X). Cet outil est particulièrement facile d'utilisation.

Enfin, d'autres outils (Visual .NET, Netbeans, Eclipse, etc.) comportent leur propre gestion de projet, et s'apparentent donc à un `Makefile`... sans en être un ! Malgré leur apparente simplicité due à l'interface graphique, les utiliser bien n'est pas si simple puisque cela revient à savoir comment exprimer les dépendances.

Conclusion

Vous pouvez maintenant utiliser un `Makefile` pour vos TP.

Modifié le: lundi 12 février 2024, 09:20

◄ Afficher les accents sous Windows

Aller à…

FunIO : fonctions de lecture pour réaliser de petits jeux sur la console ►

?