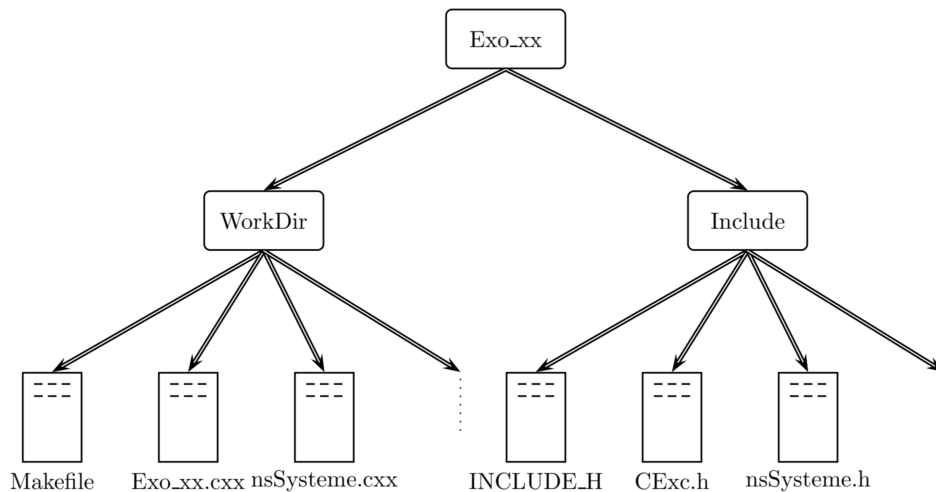
Programmation Système, TD 1

Préambule

NB : L'ensemble du TD se réfère au document de cours disponible sur AMETICE et intitulé ≪ Intro ≫.

Afin de préparer l'environnement au sein duquel seront réalisés tous les TD et TP, un nouveau répertoire sera créé pour chaque TD et/ou TP, répertoire au sein duquel sera créé pour chaque exercice un nouveau répertoire associé : chacun de ces répertoires associés sera nommé `Exo_xx` où `xx` désignera le numéro de l'exercice. A l'intérieur de chacun de ces répertoires, deux nouveaux répertoires seront créés :

- l'un, appelé `Include`, sera destiné à recevoir scripts, en-têtes, fichiers de paramètrage
- l'autre, appelé `WorkDir`, sera le répertoire de travail contenant codes sources et `makefile`



Préparation de l'environnement de travail :

- dans votre répertoire personnel créer un répertoir `ProgSys` contenant un sous-répertoire `TP_1`
- à l'intérieur de ce sous-répertoire créer le répertoire `Exo_01`
- à l'intérieur du répertoire `Exo_01`, créer deux sous-répertoires `Include` et `WorkDir`
- à chaque nouvel exercice, dupliquer le précédent répertoire `Exo_n` en le renommant `Exo_n+1`, de façon à disposer dans le répertoire `TP_1` de la suite de sous-répertoires `Exo_01`, `Exo_02`,..., qui seront structurés de manière identique.

Des questions vous sont posées tout au long du TP : Les réponses sont à rendre sur une feuille de papier libre comportant votre nom et votre groupe.

Compilation séparée, make et Makefiles de base

Lorsque le code source d'un projet de programmation se trouve dans plusieurs fichiers, on doit pouvoir les compiler séparément. L'idée est, que si l'on ne modifie qu'un ou deux fichiers, on souhaite n'avoir à recompiler que ces fichiers et ceux qui en dépendent. La commande `make`, qui opère sur un fichier de script dénommé “Makefile” ou “makefile”, permet de réaliser compilation et édition de liens en mode séparé.

Exercice 1. Récupérer les fichiers `File1.cxx`, `File2.cxx` et `notreEntete.h` sur AMETICE, étudier leur code ; placer `File1.cxx` et `File2.cxx` dans le répertoire `WorkDir`, placer `notreEntete.h` dans le répertoire `Include` ;

- placez-vous dans ce `WorkDir`, utilisé comme répertoire de travail ;
- tentez de compiler ≪ normalement ≫ en ligne de commande dans le shell :
  `g++ -o File1.exec File1.cxx File2.cxx`
  et lire le message d'erreur ;
- compilez alors à partir de la ligne de commande :
  `g++ -o File1.exec File1.cxx File2.cxx -I../Include`
- exécutez le programme `File1.exec` et lisez deux entiers au clavier.

Question 1 : Expliquer en une phrase pourquoi la commande de compilation ≪ normale ≫ n'a pas fonctionné.

En étudiant l'*arbre de dépendances* de l'exécutable `File1.exec`, on constate que

- si on crée une mise à jour du fichier `NotreEntete.h` (par exemple *via* la commande `touch NotreEntete.h`), il est alors nécessaire de recompiler l'ensemble des fichiers, ce qui apparaît normal ;
- si, par contre, on modifie le fichier `File1.cxx`, il apparaît souhaitable de ne recompiler que ce fichier avant de procéder à une nouvelle *édition de liens* entre `File1.o`, `File2.o`, `NotreEnTete.h` pour générer à nouveau l'exécutable `File1.exec`.

Question 2 : De fait, en quoi est-il normal de recompiler l'ensemble des fichiers dans le premier cas et uniquement `File1.cxx` *dans le second cas ?*

On dispose d'un outil astucieux, `make`, permettant de tenir compte de la chronologie de modification des fichiers afin de ne mettre à jour que ce qui est nécessaire. Il suffit d'établir les règles de dépendances dans un fichier particulier, nommé au choix `makefile` ou `Makefile`, qui sera lu par la commande `make`. Le principe de l'exécution de `make` est d'évaluer d'abord la premiere règle rencontrée, ou celle dont le nom est spécifié en argument de l'appel de `make`. L'évaluation d'une règle se fait recursivement. Si un fichier de dépendances est lui même un fichier *cible* d'une autre règle, cette règle est à son tour évaluée.

Exercice 2. Placez-vous dans le répertoire `WorkDir`. Dans un fichier appelé `makefile` ouvert *via* l'éditeur de texte de votre choix, expliciter les dépendances permettant de créer l'exécutable `File1.exec` et les fichiers objets, `File1.o`, `File2.o`. On rappelle que la syntaxe d'une règle est la suivante (attention à la tabulation, obligatoire) :

$$\langle \text{cible} \rangle : \quad \langle \text{dépendances} \rangle$$
$$\Longleftrightarrow \langle \text{opérations} \rangle$$

Une fois le fichier `makefile` constitué :

- lancer la commande `make` (qui prendra par défaut le fichier `makefile` constitué) et observer ce qui se passe ;
- si tout s'est bien passé, lancer dans le shell la commande `ls -lct` afin d'examiner les dates de modification des fichiers ; *Question 3 : Quel est précisément l'intérêt des options* `-lct` *de la commande* `ls` *?*

— effectuer dans le shell la commande `touch File1.cxx` suivie à nouveau de la commande `ls -lct` ; *Question 4 : Que constatez-vous par rapport à l'étape précédente ?*
— relancer la commande `make` ; vous constaterez que seules les lignes
```
g++ -c -I../Include File1.cxx
g++ -o File1.exec File1.o File2.o
```
sont exécutées par `make`, car les fichiers cibles `File1.o` et `File1.exec` sont les seuls devant être mis à jour ;

Exercice 3. Le fichier `makefile` peut contenir des commandes du shell sans nécessiter obligatoirement la présence de dépendances : ceci permet (par exemple) d'ajouter au `makefile` des commandes de nettoyage :
— ajouter à la suite des cibles existantes dans votre `makefile` une cible intitulée « clean », suivie d'une liste de dépendances vide, et ayant pour unique opération associée la commande `rm *.o File1.exec`
— dans le shell, lancer la commande `make clean`, suivie de la commande `ls -l` pour vérifier ce qui s'est passé ; *Question 5 : Précisément, quel sont les fichiers restant dans le répertoire ?*
— une fois ce nettoyage effectué, générer à nouveau dans le répertoire l'exécutable `File1.exec`

Makefiles « génériques », constitution de bibliothèques

L'outil `make` intègre la notion de variable qui permet de rendre les traitements plus génériques et donc plus flexibles. Par ailleurs, on peut imaginer enrichir le fichier `File2.cxx` avec d'autres fonctions et en faire ainsi une bibliothèque utilisable avec d'autres programmes : il s'agira alors de constituer une *archive* qui sera générée avec l'outil ad-hoc `ar` afin de pouvoir la passer en tant que bibliothèque au compilateur/linker *via* l'option `-lSys`.

Exercice 4. Par étapes : (1) constitution d'un `makefile` générique (c'est-à-dire capable de fonctionner avec d'autres programmes que `File1`, (2) constitution d'une bibliothèque de calcul à partir du fichier source `Fichier2.cxx`... Pour l'étape 1 :
— Remplacer la commande `g++ -c -I../include` partout dans le `makefile` avec l'appel de variable `$(COMPILER)`, et définir cette variable tout en haut du `makefile` ainsi :
```
COMPILER = g++ -c -I../Include
```
Cette ligne est seule, sans seconde ligne, ni TAB, etc... Relancer ensuite `make clean`, puis `make` pour constater que tout fonctionne comme auparavant. *Question 6 : Quel est précisément l'intérêt de la variable* `$(COMPILER)` *?*
— Afin de rendre le `makefile` encore plus générique, on souhaite pouvoir passer à `make` le nom d'un programme quelconque *via* une nouvelle variable, en tapant depuis le shell l'appel : `make nom=UnProgrammeQuelconque` (par exemple) et obtenir ainsi l'exécutable `UnProgrammeQuelconque.exec`. Il faut alors décrire les cibles et dépendances à partir de la variable `nom`. Dans le `makefile` courant, remplacer la règle ayant pour cible `File1.o` par :
```
$(nom).o : $(nom).cxx ../Include/notreEntete.h
        $(COMPILER) $(nom).cxx
```
— Remplacer alors dans les autres règles du `makefile` toutes les occurrences de `File1` par `$(nom)`
— Nettoyer le répertoire, puis générer à nouveau l'exécutable `File1.exec` *Question 7 : Qu'avez-vous à taper en ligne de commande dans le shell pour cela ?*

Le `makefile` obtenu à ce stade permet d'utiliser les fonctions disponibles dans `File2.cxx` avec tout autre programme qui sera compilé en le passant *via* la variable `nom`. A ce stade néanmoins, `File2` ne représente pas encore une bibliothèque distribuable puisqu'il est nécessaire que sa source reste accessible pour pouvoir procéder à une édition de liens avec le programme qui voudrait l'utiliser. Pour en faire une archive utilisable en tant que bibliothèque, il est donc nécessaire de procéder une fois pour toute à sa compilation avant d'en faire une archive reconnue par `g++`. En l'occurence, et par convention, tout intitulé d'archive doit obligatoirement comporter dans son nom le préfixe `lib` et le suffixe `.a` ; par exemple si nous décidons d'inclure dans le nom de notre bibliothèque le mot `Sys` (pour `Sys`tème), notre bibliotèque s'intitulera forcément `libSys.a` pour pouvoir être reconnue par `g++` Ainsi, et pour faire de `File2` une bibliothèque, dans le `makefile` courant :

- dans la règle servant à fabriquer `File2.o`, remplacer cette cible par `libSys.a` (la règle sert donc à fabriquer la bibliothèque) ;
- dans les *opérations* associées à cette règle
  - compiler `File2.cxx` avec `$(COMPILER)`
  - faire suivre d'un ≪ `;` ≫ pour séparer de la commande suivante
  - ajouter la commande d'archivage : `ar -cqs libSys.a File2.o ; rm File2.o`

  Question 8 : A quoi servent les options `-cqs` *de la commande* `ar` *?*
- pour pouvoir procéder à l'édition de liens avec l'archive constituée dans le `makefile` courant, remplacer toute occurence `File2.o` par `libSys.a` dans la règle ayant pour cible `$(nom).exec`
- dans cette même règle, remplacer l'opération d'édition de liens par l'opération `g++ -s -o $(nom).exec $(nom).o -L. -lSys`

  Question 9 : Quel est le sens des options `-L` *et* `-lSys` *associées à cet appel de* `g++`*, et pourquoi le nom complet de l'archive ≪* `libSys.a` *≫ ne suit-il pas l'option* `-l` *?*
- Refaire l'ensemble des tests (`make clean`, `make nom=File1`, etc...), et des appels à `ls -l` pour voir ce qui est apparu dans le répertoire courant.

Question 10 : Réaliser une liste avec des fichiers/répertoires apparus. Dessiner l'arbre de dépendances de l'exécutable `File1.exec`

Epilogue

Le code source de projets téléchargés pour Unix/Linux comporte toujours un outil de configuration s'adaptant au type de système Unix et d'architecture sur lequel il doit être compilé, générant ensuite un makefile permettant la compilation et l'édition de liens pour le projet. Voir ;o) `man make`.