PoP Série 0

H1 : La commande `make` et la compilation séparée

H2 : Lire le document sur le développement de projet et répondre aux questions théoriques (sauf questions avancées)

S'inscrire au MOOC Introduction à la programmation orientée objet (en C++).
La première série du MOOC commence la semaine prochaine.

La commande `make` et la compilation séparée.

Buts. Le but de cette série est de comprendre la raison d'être et le fonctionnement de l'outil de développement `make`, qui permet de re-générer un exécutable en ne recompilant que les fichiers sources ayant subi une modification depuis leur dernière compilation. Ce sujet est indépendant du MOOC.

Prérequis : Savoir compiler avec g++ et geany, au besoin revoir la série 1 du semestre précédent.

Exercice 1 (niveau 0) : Notions de module (interface et implémentation)

Il est souvent utile de découper un programme en plusieurs modules, et ceci pour plusieurs raisons détaillées dans le document sur le développement de projet. Nous rappelons ici seulement les plus importantes en relation avec le sujet de cette première partie, la ***compilation séparée*** :

1. un module est une entité qui peut être indépendante des autres, ce qui facilite sa compréhension.

2. chaque module peut être développé indépendamment, ce qui est indispensable lorsqu'on travaille en équipe, deux personnes ne pouvant pas éditer simultanément le même fichier.

3. chaque module peut être compilé séparément. Par conséquent, lors de la modification d'un module, seul ce module (et ceux qui en dépendent) devra être recompilé, ce qui réduit considérablement le temps nécessaire à la création de l'exécutable.

Nous appellerons **module** une entité logique composée de deux fichiers :

— **un fichier en-tête qui constitue son INTERFACE** (exemple vu en cours : `calcul.h`), contenant seulement les ***déclarations*** de symboles, de types et les prototypes de fonctions exportées.

  — Cela permet par exemple d'appeler une fonction exportée `f` dans un autre module X sans devoir connaître la définition détaillée de cette fonction `f`.

    — La seule condition à remplir est que cet autre module X contienne une directive `#include` pour récupérer le fichier en-tête qui contient le prototype de `f`.

— **un fichier source qui constitue son IMPLÉMENTATION** (exemple vu en cours : `calcul.cc`), contient la ***définition*** des fonctions. Il inclut, entre autres, le fichier en-tête du module (pour notre exemple, il inclut donc `calcul.h`).

La compilation du fichier source produit un module objet, qui est un fichier portant le même nom que le fichier source, mais avec l'extension `.o` . Jusqu'à présent ce fichier intermédiaire était invisible pour vous, car vous n'aviez qu'un seul fichier source, et les étapes de compilation et d'édition de liens étaient regroupées en une seule qui produisait directement un exécutable.

Ce ne sera plus forcément le cas dorénavant : un module objet sera généré à partir de chaque module (*compilation*), et l'exécutable final sera généré à partir de tous les modules objets (*édition des liens*).

Remarque. Avec la commande `g++`, il faut utiliser l'option `-c` pour forcer la génération d'un module objet. Par exemple : `g++ -c truc.cc` ne va pas générer de fichier exécutable, mais un module objet appelé `truc.o`.

Question : Soit un module B constitué par les fichiers `B.h` et `B.cc`. On désire appeler une fonction `g()` du module B dans le code d'une fonction `h()` du module A.

Que dois-je faire au minimum pour que l'implémentation du module A, c'est-à-dire `A.cc`, compile correctement pour produire un fichier `A.o` ?

Piste : Se concentrer sur la directive `#include` et ce qui est inclus.

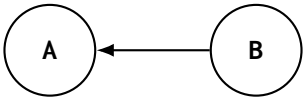
Exercice 2 (niveau 0) : Le graphe des dépendances des fichiers

En langage C++, un programme composé de plusieurs modules n'a pas nécessairement besoin d'être entièrement recompilé lors d'une modification. Uniquement les modules modifiés et **ceux qui en dépendent** doivent l'être. Cependant, cette décision n'est pas triviale. En effet, un fichier donné peut dépendre d'un ou plusieurs autres fichiers, qui peuvent aussi dépendre d'autres fichiers, etc.

Ces dépendances sont dues aux directives `#include` présentes dans les fichiers.

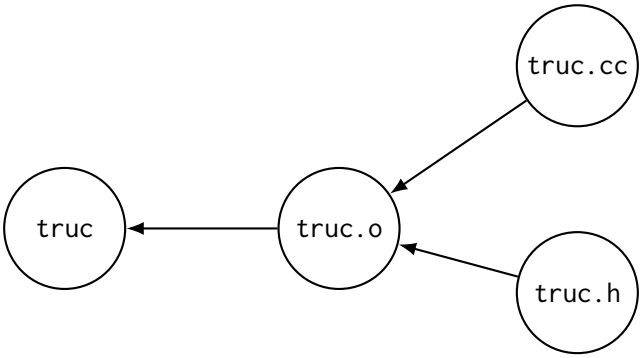
Alors qu'une modification d'un fichier source implique uniquement sa recompilation, une modification d'un fichier en-tête implique la recompilation de tous les modules qui en dépendent. Toutes ces dépendances peuvent être représentées graphiquement à l'aide d'un graphe.

Il est instructif de représenter graphiquement les dépendances entre fichiers. Si chaque fichier est représenté à l'aide d'un cercle contenant son nom, on exprime par ***une flèche orientée*** le fait que ***le fichier A dépend du fichier B*** :



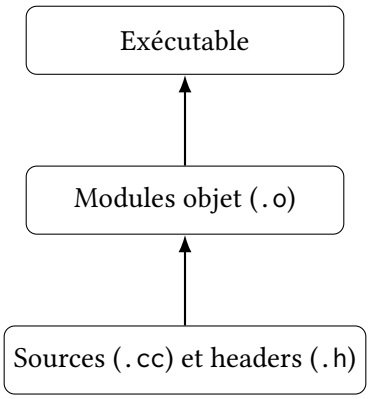
Remarque. Cette représentation n'est pas standardisée. Nous garderons toujours le sens indiqué ci-dessus pour la flèche mais parfois nous représenterons le cercle de A au-dessus de celui de B, ou à droite de celui de B.

Par exemple, soient trois fichiers `truc.h`, `truc.cc` et `truc.o`. Le module objet `truc.o` dépend des fichiers du module truc, à savoir les fichiers `truc.h` et `truc.cc`, car une modification de l'un de ces fichiers implique que le module objet soit re-généré. De plus, l'exécutable `truc` dépend du module objet `truc.o`. Sous forme de graphe, cela donne :



Il s'agit là d'un exemple simple avec un seul module, mais dès qu'il y en a plusieurs, ce graphe peut devenir assez complexe. Cependant, les dépendances présentes dans un projet peuvent se résumer à **un graphe à trois étages** :

- **l'étage de l'exécutable :** L'exécutable dépend de tous les modules objet.
- **l'étage des modules objets :** Chaque module objet (`.o`) dépend de son fichier source (`.cc`), ainsi que de tous les autres fichiers en-tête éventuellement inclus dans le fichier source.
- **l'étage des fichiers sources (`.cc`) et en-tête (`.h`) :** Ces fichiers apparaissent au même niveau dans un graphe de dépendance car *aucune opération automatique n'est faite pour préserver la cohérence entre le fichier* `.cc` *et son fichier en-tête* `.h`. Seule la personne qui programme peut et doit le faire avec son éditeur favori.



Remarque. Les dépendances vis-à-vis des fichiers en-tête standard (`iostream`, `string`, `vector`, `cmath`, ...) ne doivent pas être prises en compte dans ces dessins. En effet, ces fichiers ne sont de toute façon pas modifiables.

Exercice : Récupérez le projet qui se trouve dans le répertoire `PoP_s0_code` de l'archive "**code source calendrier**" fournie. Il s'agit d'un petit projet fictif pour illustrer les principes de la compilation séparée.

- Le module `calendrier` offre des fonctions de base.
- Le module `difference` offre la fonction qui permet de calculer la différence en jours entre deux dates.
- Le programme principal, qui appelle les fonctions de ces deux modules, se trouve dans le fichier `prog.cc` (remarquez qu'il n'y a PAS de fichier `prog.h` car c'est inutile : aucune fonction n'en est exportée).

Question : Identifiez à la main les dépendances qui existent entre les 5 fichiers source, les 3 fichiers objets et l'exécutable du projet, et dessinez le graphe des dépendances en le structurant en 3 couches (source/objet/exécutable). (Ignorez le fichier Makefile pour l'instant).

Exercice 3 (niveau 0) : La commande `make`, le fichier `Makefile` et sa syntaxe

L'outil qui, sous Linux/Unix, permet de compiler automatiquement le minimum de fichiers nécessaires à la création d'un nouvel exécutable, est la commande `make`.

Cette commande exécute un fichier, nommé `Makefile` (ou `makefile`), qui doit se trouver dans le répertoire du projet, à côté des autres fichiers source. Le `Makefile` contient la description des dépendances entre fichiers, ainsi que les actions à entreprendre pour générer un exécutable.

C'est au programmeur d'écrire le fichier `Makefile`. Il s'agit d'un fichier texte ayant *une syntaxe spécialisée pour la commande* `make`. La plupart du temps vous allez ré-utiliser un fichier `Makefile` déjà existant, que vous adapterez à vos besoins pour vos futurs projets.

Commencez par ouvrir avec **geany** le fichier `Makefile` associé au projet, et à regarder son contenu.

Un fichier `Makefile` peut contenir :

- des définitions de *variables* ;
- des définitions de *règles* ;
- des *commentaires* (lignes commençant par #).

Définitions de variables. La notion de variable est beaucoup plus limitée qu'en C++. On parle de variable « macro ». Par exemple, on déclare une variable `VAR` simplement avec le symbole `=`.

```
VAR = chaine de caracteres avec eventuellement des espaces
```

Ensuite, pour obtenir la valeur d'une variable dans une règle ou une commande il faut utiliser la syntaxe : `$(VAR)`.

Si la définition ne tient pas sur une seule ligne, il faut terminer chaque ligne (sauf la dernière !) par le symbole `\`.

Par ailleurs, une définition d'une variable peut sans problème combiner la valeur d'autres variables.

Comme en C++, l'intérêt des variables est de définir une seule fois une expression éventuellement complexe qui doit être utilisée plusieurs fois. Exemples :

```
CXXFLAGS = -Wall -std=c++11 -g
LIBS = -lglut -lGL -lGLU -lm -Llib -lglui -L/usr/X11R6/lib -lX11 -lXext -lXmu -lXi
```

En particulier, nous aurons besoin des variables `CXXFILES` et `OFILES`, qui représentent respectivement la liste des fichiers source et la liste des fichiers objet. C'est au programmeur de mettre à jour ces variables en fonction de l'évolution du projet (ajout ou suppression de fichiers sources). Exemple :

```
CXXFILES = chaine.cc graphic.cc cine.cc
```

Exercice : Identifiez les définitions de variables « macros » présentes dans le `Makefile` que vous venez d'ouvrir.

On trouve :

- la macro `CXX`, qui désigne le compilateur utilisé (ici : `g++`).
- la macro `CXXFLAGS`, qui désigne les options de compilation (ici : `-Wall`, qui permet d'obtenir des messages additionnels d'avertissement (*warnings*), signalant des erreurs non fatales, et `-std=c++11` pour le standard du C++ utilisé).
- la macro `CXXFILES`, qui liste l'ensemble des fichiers source.
- la macro `OFILES`, qui liste l'ensemble des modules objet.

Remarque. On peut aussi écrire `OFILES = $(CXXFILES:.cc=.o)`, ce qui permet de lister automatiquement l'ensemble des modules objet à partir de la liste de fichiers source (`CXXFILES`), nous évitant de taper deux fois les mêmes noms de modules. Faites-le et sauvegardez le fichier `Makefile`.

Définition de règles. Il existe plusieurs types de règles mais elles commencent toutes par indiquer une ***cible*** suivie de `:`.

Tout d'abord regardons les règles qui permettent de spécifier les ***dépendances*** entre fichiers. Par exemple :

```
FICHIER_A: FICHIER_B
```

Cela signifie que la ***cible*** FICHIER_A dépend du FICHIER_B (ceci est équivalent à un arc du graphe des dépendances). Plusieurs fichiers peuvent figurer à droite du : . Par exemple :

```
FICHIER_A: FICHIER_B FICHIER_C FICHIER_D
```

Cela signifie que le FICHIER_A dépend de FICHIER_B, FICHIER_C et FICHIER_D.

Ensuite, il est possible d'associer une (ou plusieurs) ***action(s)*** à la règle. Une action apparaît sur la ligne qui suit la règle après un caractère TAB. Cette action sera exécutée si au moins un des fichiers dont dépend la cible est plus récent que la cible elle-même (**la date et l'heure de la dernière modification des fichiers faisant foi**).

Typiquement, l'action à effectuer est d'invoquer le compilateur g++, mais toute autre commande Unix peut être invoquée (cp, rm, echo, ...). Voici un exemple avec deux actions (g++ et echo), qui sont exécutées si truc.cc est plus récent que la cible truc.o :

```
truc.o: truc.cc
        g++ -c truc.cc
        @echo "Compilation effectuee: truc.o a ete genere."
```

ATTENTION ! Le caractère TAB est absolument nécessaire au début de chaque ligne effectuant une action (ici : avant g++ et avant @echo) et pas des espaces ! La syntaxe du Makefile est très stricte, et ne pardonne aucune erreur.

Remarque. La commande echo permet d'afficher une chaîne de caractères.

Remarque. Le symbole @ qui précède la commande echo indique que la commande elle-même (c.à.d. echo) ne doit pas être affichée dans le Terminal (comme c'est le cas par défaut). Par contre le message proprement dit est bien affiché dans le Terminal, car la commande echo est bel et bien exécutée.

Bien entendu, pour un projet il y aura plusieurs règles dans le Makefile, puisque toutes les dépendances identifiées dans la section 2 doivent être exprimées. Lors de son exécution, la **première** règle figurant dans le fichier est considérée. Il s'agit généralement de la règle indiquant que le fichier exécutable dépend de tous les modules objet (.o). Pour le projet fictif, il s'agit de la règle suivante (lignes 10 et 11 du Makefile) :

```
prog: $(OFILES)
        $(CXX) $(OFILES) -o prog
```

Cette règle indique que l'exécutable prog dépend de tous les modules objet $(OFILES) et que, si l'un de ces modules objet est plus récent que l'exécutable, il faut appeler la commande g++ pour effectuer l'édition des liens entre tous les modules objet $(OFILES) et re-générer un exécutable nommé prog (vous connaissez déjà l'option -o).

Après lecture de cette règle, et avant exécution de l'éventuelle action associée, l'outil make vérifie si l'un des modules objet dépend à son tour d'autres fichiers, au cas où ces modules objet eux-mêmes devraient être mis à jour (il *remonte* le graphe des dépendances en direction du code source). Ces dépendances doivent être aussi exprimées par des règles.

On peut exprimer le fait que chaque module objet dépend de son fichier source (exemple niveau 0), cependant cela peut demander une longue liste de règles.

Heureusement il existe une règle **prédéfinie** qui lie automatiquement tout fichier objet à son fichier source. L'action qui est entreprise lorsqu'un fichier source est plus récent qu'un fichier objet est la suivante :

```
$(CXX) $(CXXFLAGS) -c $<
```

Remarque. L'expression prédéfinie $< désigne la première dépendance du fichier objet, c'est-à-dire le fichier source (.cc). De plus, l'option -c indique au compilateur de générer un module objet (.o), au lieu de générer directement un exécutable.

Remarque. Les macros CXX et CXXFLAGS doivent être définies dans le Makefile par le programmeur.

Toutes les dépendances n'ont pas encore été décrites dans le fichier Makefile. En effet, tout module objet ne dépend pas uniquement du fichier source homonyme, mais aussi de tous les fichiers en-tête (non standard) inclus par le fichier source. Voici ces règles dans le cas du projet fictif :

```
calendrier.o: calendrier.cc calendrier.h
difference.o: difference.cc calendrier.h difference.h
prog.o: prog.cc calendrier.h difference.h
```

Écrire ces règles à la main est possible mais sujet à erreurs surtout quand le nombre de modules est élevé. Heureusement on peut les générer automatiquement, comme expliqué dans la section suivante.

Exercice 4 (niveau 0) : Comment identifier automatiquement les dépendances

Cela dépend de l'environnement de programmation (système d'exploitation, compilateur, ...). Voyons comment cela marche sous Linux, avec le compilateur `g++`.

L'option `-MM` de `g++` affiche les dépendances d'un module objet (`.o`) vis-à-vis du fichier source (`.cc`) correspondant et des fichiers en-tête (`.h`), excepté les dépendances vis-à-vis des fichiers en-têtes standards. Le résultat est une liste de dépendances sous la forme standard d'une règle, vue dans l'exercice précédent.

Exercice : Faites un test en tapant la commande suivante depuis le répertoire du projet fictif :

```
g++ -MM calendrier.cc
```

Remarque. Aucune compilation n'est effectuée si l'option `-MM` est utilisée.

Exercice 5 (niveau 0) : Cibles particulières

La possibilité ci-dessus est utilisée dans le second type de cible illustrée maintenant avec la **cible depend**. Ce type de cible est appelé une ***cible particulière*** car ce n'est pas un fichier et on n'indique pas de fichiers dépendants dans la règle. Le but est que l'action associée soit automatiquement exécutée si la cible est sélectionnée. Voici un exemple simple de cible particulière :

```
ma_cible:
    @echo "Badaboum."
```

Une telle cible peut être invoquée lors de l'appel de la commande `make`. Par exemple, la commande :

```
make ma_cible
```

... exécutera l'action `echo` qui affichera le message "Badaboum".

Dans le Makefile du projet fictif, il y a deux *cibles particulières*, et qui sont très utiles :

- `depend` : génère automatiquement les règles qui représentent le graphe des dépendances.
- `clean` : efface tous les fichiers "inutiles" du projet (c.à.d. l'exécutable et les modules objet).

La cible `depend` doit être invoquée par le programmeur à chaque fois que le graphe des dépendances a changé (à cause, par exemple, d'un ajout ou d'un retrait d'un fichier). L'exécution de la cible `depend` va automatiquement mettre à jour le fichier `Makefile` lui-même, en utilisant l'option `-MM` du compilateur `g++` vue auparavant.

Exercice : Pour voir l'effet de la cible `depend` :

- ouvrez le fichier `Makefile` dans l'éditeur et supprimez les lignes 31 à 33 si elles existent. Il s'agit des lignes juste après la ligne `# DO NOT DELETE THIS LINE`
- puis **quittez l'édition du fichier `Makefile`** car il **va être modifié** automatiquement avec l'action suivante.
- lancez la commande `make depend` ; quelques lignes s'affichent dans la fenêtre terminal.
- ouvrez le nouveau fichier `Makefile` avec **geany**. Notez que trois règles ont été ajoutées en fin de fichier.

Remarque. Il n'y a pas besoin de comprendre le détail de l'action de la cible `depend`, qui est assez compliquée.

Exercice 6 (niveau 0) : Conseils pour l'utilisation pratique des `Makefile` (résumé)

Lors de la création d'un nouveau projet :

- copiez dans le répertoire du projet un ancien fichier `Makefile`
- éditez-le, et mettez à jour les macros en fonction du projet (en particulier, les macros `CXXFILES` et éventuellement `OFILES`)
- mettez à jour les règles de dépendances, en tapant make depend (ceci modifie votre fichier `Makefile` DONC il ne faut pas l'éditer en même temps qu'on exécute cette cible ; c'est une source de confusion fréquente)

Ensuite, à chaque fois que vous **modifiez** un ou plusieurs fichiers de votre projet (`.cc` ou `.h`), exécutez simplement `make` pour générer un nouvel exécutable.

Pour faire cela directement depuis geany, éditez la commande du bouton `Build` en choisissant « `Set Build Commands` » dans le menu du bouton `Build`. Il suffit alors de remplacer la commande de la ligne 2. `Build` par `make`. Ainsi vous n'avez plus besoin d'appeler directement le compilateur `g++` !

Si vous ne voulez pas changer la commande associée au bouton `Build`, l'alternative est d'indiquer une des cibles de votre `Makefile` quand vous choisissez « `Make custom target` » avec le menu du bouton `Build`.

De plus, à chaque **ajout** ou **retrait** d'un fichier (`.cc` ou `.h`) du projet, mettez à jour les macros, terminez l'édition du fichier `Makefile`, puis tapez `make depend` dans le terminal pour remettre à jour les règles de dépendances.

Si vous voulez effacer rapidement les fichiers "inutiles" (`.x` et `.o`) pour faire de la place, tapez `make clean`.

Exercice 7 (niveau 0) : Un exemple de compilation séparée

Cet exercice et son corrigé se trouvent dans le document séparé « PoP série 0 - Exercice 7 » (disponible sur moodle) et expliquent comment créer un `Makefile` pas à pas.

Exercice 8 (niveau 1)

Effectuez les étapes suivantes dans l'ordre, en partant du répertoire `PoP_s0_code` du projet :

- Fermez la fenêtre d'édition du `Makefile`, si vous étiez en train de l'éditer.
- Tapez `make clean`.
- Tapez `make depend`.
- Ouvrez, avec geany, le fichier source `prog.cc`.
- Lancez la commande `make` depuis geany (voir exercice 6) ou tapez simplement `make` dans le terminal. Observez les commandes et messages qui s'affichent dans la sous-fenêtre de geany, et qui montrent la progression, étape par étape. Combien d'appels à la commande `g++` sont effectués ? Pour chaque appel, indiquez s'il s'agit d'une compilation ou d'une édition de liens.
- Exécutez le programme (`prog`) dans le terminal et vérifiez brièvement qu'il fonctionne.
- Relancez la commande `make`. Que se passe-t-il cette fois ? Pourquoi ?
- Modifiez `prog.cc` (par ex. le message d'introduction), sauvegardez. Relancez `make`. Combien de compilations sont faites ? Pourquoi ? Exécutez le programme.
- Modifiez `difference.h` (par ex. un commentaire), sauvegardez. Relancez `make`. Combien de compilations sont faites ? Pourquoi ?
- Modifiez `calendrier.h` (par ex. un commentaire), sauvegardez. Relancez `make`. Combien de compilations sont faites ? Pourquoi ?
- Modifiez `calendrier.cc` (par ex. un commentaire), sauvegardez. Relancez `make`. Combien de compilations sont faites ? Pourquoi ?

- Effacez le fichier exécutable `prog` depuis une fenêtre Terminal, et relancez `make`. Combien de compilations sont faites ? Pourquoi ?
- Faites `make clean`, puis `make`. Combien de compilations sont faites ? Pourquoi ?

Ajoutez un module `siecle` dans le projet : créez les fichiers `siecle.h` et `siecle.cc`, et écrivez une fonction (prototype : `int siecle (int annee);`) qui retourne le siècle de l'année passée en argument (par exemple : 20, pour l'année 1971). Appelez la fonction `siecle( )` depuis le programme principal (`main`), pour afficher le siècle des deux dates fournies par l'utilisateur (par exemple : "Cette date se trouve dans le 20ème siècle"). Ensuite, mettez à jour le `Makefile` pour qu'il tienne compte du nouveau module. Enfin, générez un exécutable et testez votre programme. N'avez-vous rien oublié ?