TP1: Compilation et Makefile

La compilation, qu'est-ce que c'est?

Contrairement à des langages interprétés comme Python, un programme en C doit au préalable être **compilé**, c'est-à-dire qu'à partir du **code source** nous allons générer un **code machine** stocké dans un **exécutable**. C'est ce code machine qui permettra d'indiquer au processeur les instructions à exécuter.

Bien sur, ce n'est pas vraiment nous qui allons générer l'exécutable, mais plutot un programme appelé **compilateur**. Un compilateur très répandu est gcc, nous l'utiliserons donc par la suite.

> Par abus de langage, "compilation" désigne bien souvent tout le processus de transformation du code source en code machine. Cependant les choses sont un peu plus compliqué en pratique.

Compilation avec gcc

Ouvrez un editeur de texte et recopiez le programme suivant:

```c
#include <stdio.h>

int
main(void)
{
    printf("Bonjour!\n");
    return 0;
}
```

Sauvegardez ce fichier que vous appelerez `bonjour.c`. Ouvrez ensuite un terminal, placez-vous dans le dossier contenant votre fichier nouvellement créé, puis exécutez la commande suivante:

```
$ gcc bonjour.c
```

Affichez maintenant le contenu de votre dossier. Un fichier `a.out` a normalement été produit! C'est votre exécutable. Pour l'exécuter, tapez la commande suivante:

```
$ ./a.out
Bonjour!
```

Il est possible de renommer l'exécutable. Pour cela il faut rajouter l'option `-o <nom>`.

```
$ gcc bonjour.c -o bonjour
```

Il est également possible d'utiliser plusieurs fichiers pour générer notre exécutable. Créez un fichier `main.c` dans lequel vous copierez votre fonction `main`.

```
int
main(void)
{
    bonjour();
    return 0;
}
```

Modifiez maintenant votre fichier `bonjour.c` afin qu'il contienne le code suivant.

```
#include <stdio.h>

void
bonjour()
{
    printf("Bonjour!\n");
}
```

Compilez les deux fichiers avec la syntaxe suivante:

```
$ gcc main.c bonjour.c -o bonjour
```

Une erreur doit normalement apparaitre... Que se passe-t-il selon vous?

Créez maintenant un fichier `bonjour.h` et ajoutez-y l'unique ligne suivante:

```
void bonjour();
```

Rajoutez ensuite la ligne `#include "bonjour.h"` au début du fichier `main.c` ainsi qu'à la deuxième ligne du fichier `bonjour.c`. Relancer la compilation. Normalement, les choses doivent mieux se passer!

> Il est également possible de compiler séparement les différents fichiers source puis de les lier. Pour cela, il faut utiliser l'option `-c` qui génère des **fichiers objet** avec l'extension `.o`. On peut ensuite passer à la phase d'édition de liens.
>
> ```
> $ gcc -c bonjour.c
> $ gcc -c main.c
> $ gcc main.o bonjour.o -o bonjour
> ```
>
> Compiler séparément les fichiers peut se révéler utile si l'on ne modifie qu'un seul fichier et que l'on veut s'épargner la recompilation "pour rien" des autres.

Nous allons maintenant ajouter quelques options supplémentaires:

```
$ gcc -std=c11 -Wall -Wextra -O0 bonjour.c main.c -o bonjour
```

Voici quelques explication concernant les options proposées:

- `-std=c11`: permet de préciser le standard du langage C que nous souhaitons utiliser. En effet, la syntaxe et les fonctionnalités du langage évolue avec le temps. Ici nous avons choisi la norme

C11, qui inclut notamment, par rapport à la norme C99, la programmation multi-thread ainsi qu'une meilleur gestion de l'encodage UNICODE.
- `-Wall`: cette option permet d'afficher des avertissements sur un certain nombre d'erreurs de programmation. Votre programme pourra tout de même compiler mais la présence d'avertissement peut impliquer un comportement erroné lors de l'exécution de votre programme.
- `-Wextra`: d'autres avertissements, que nous détaillerons ultérieurement.
- `-O0`: désactive toute forme d'optimisation que votre compilateur pourrait utiliser.

Utilisation de make et premiers Makefiles

L'outil `make` permet d'automatiser le processus de compilation. Il est associé à un fichier appelé `Makefile`, qui écrit l'ensemble des **règles de compilation**.

Une règle se présente sous cette forme:

```
cible: dépendances
    commande
```

Par exemple, si on souhaite compiler notre fichier `bonjour.c`, la règle sera:

```
bonjour.o: bonjour.c bonjour.h
    gcc -Wall -c bonjour.c
```

On peut ainsi obtenir une première version de notre Makefile.

```
main: main.o bonjour.o
    gcc -Wall main.o bonjour.o -o main
main.o: main.c bonjour.h
    gcc -c main.c
bonjour.o: bonjour.c bonjour.h
    gcc -c bonjour.c
```

Cependant cette première version ne nous apporte pas grand chose par rapport à un simple script shell (excepté le fait qu'un fichier non modifié ne sera pas re-compilé).

Heureusement, `make` est bien plus puissant que ca. Nous avons notamment la possibilité d'utiliser des ***variables** ainsi que des **règles implicites**. Voici une deuxième version possible de notre Makefile.

```
CC=gcc
OBJ=main.o bonjour.o

main: $(OBJ)
main.o: main.c bonjour.h
bonjour.o: bonjour.c bonjour.h

.PHONY : clean
clean:
    rm -f main *.o
```

Le programme `make` va déduire de lui-même les commandes à passer au compilateur. On peut même aller encore plus loin.

```
# on précise le compilateur à utiliser
CC=gcc

# on précise la liste des fichiers objets
OBJ=main.o bonjour.o

# on précise les options de compilation
CFLAGS=-std=c11 -Wall -Wextra -pedantic -ggdb

main: $(OBJ)
    $(CC) $(CFLAGS) $^ -o $@

# Règle implicite pour la génération des fichiers objet.
%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@ -M

.PHONY : clean
clean:
    rm -f main *.o
```

Le point technique réside surtout dans la définition de la règle implicite. On précise que la règle a générer les fichiers cibles dont le nom terminera par `.o` à partir des fichiers dont le nom termine par `.c`. On utilise également des variables automatiques: `$<` sélectionne le premier fichier listé dans les prérequis, `$@` le nom du fichier cible et `$^` l'ensemble des fichiers prérequis. On remarquera aussi l'option `-M` qui permet de gérer les dépendances aux différents fichiers d'en-tête (.h).

A vous de coder!

Commencez par créer 4 fichiers:

- un fichier `tp1_main.c` qui contiendra votre fonction `main`
- un fichier `tp1_exos.c` qui contiendra vos exercices
- un fichier `tp1_exos.h` qui contiendra la déclaration des fonctions implémentées dans `tp1_exos.c`.
- un Makefile

Reprenez ensuite les exercices du td1 et implémentez les différentes fonctions.