Le but de ce TP est de vous familiariser avec la chaîne de compilation et l'outil `make`.

1 Compilation d'un fichier simple

Recopiez le programme suivant dans un fichier `tp1_1.c`. **Attention** recopiez le exactement tel qu'il est. Il contient volontairement une erreur, que nous allons dépister par la suite.

```
1  #include <stdio.h>
2
3  int fonction_a(int param)
4  {
5    return 42 + param;
6  }
7
8  void fonction_b(int param)
9  {
10    if (param = 0)
11       printf("Le paramètre est nul\n");
12    else
13       printf("Le retour de la fonction_a est %d\n", fonction_a(param));
14 }
15
16 int main(void)
17 {
18   fonction_b(10);
19   fonction_a(20);
20   return 0;
21 }
```

La méthode la plus simple pour compiler le programme est d'utiliser la commande

```
$ gcc tp1_1.c
```

Q 1. Exécutez cette commande. Quel est le nom du fichier généré à la fin du processus de compilation ?

Pour changer le nom du fichier généré, on utilise l'option `-o`

Q 2. Compilez à nouveau votre programme mais en utilisant cette nouvelle option pour que le fichier de destination se nomme `tp1_1`.

Nous allons maintenant utiliser des options qui forcent le compilateur à mener une analyse plus poussée de votre fichier source. Le but est de vous aider à trouver un certain nombre d'erreurs **à la compilation**.

Q 3. Modifiez la ligne 18 du programme pour passer la valeur 0 en paramètre à `fonction_b`. Que ce passe-t'il si vous compilez et exécutez votre programme ? Pourquoi ?

Q 4. Compilez maintenant votre programme avec les options `-Wall -W -Werror`, que remarquez vous ? Comment corriger le programme ?

L'erreur que nous avons remarquée est une erreur assez courante (il suffit d'une simple faute de frappe). De nombreuses autres erreurs courant seront détectées en utilisant ces options de compilation.

Nous pouvons voir ici l'intérêt des ces option qui active plus d'analyse sur le fichier source. Dorénavant, nous utiliserons **toujours** ces options.

2 Séparation du programme en plusieurs fichiers

Nous allons maintenant voir comment compiler plusieurs fichiers en un seul programme.

Q 5. Créez deux fichiers `tp1_2.c` et `tp1_3.c`. Coupez puis coller les fonctions `fonction_a` et `fonction_b` dans les fichiers `tp1_2.c` et `tp1_3.c` respectivement. Laissez la fonction `main` dans le fichier d'origine.

Pour compiler plusieurs fichiers nous avons deux solutions :

1. compiler le tout en une seule commande ;
2. effectuer les trois premières étapes de la compilation (préprocesseur, compilation, assemblage) séparément sur chacun des fichiers, puis effectuer l'édition de liens avec les fichiers objets résultants.

L'avantage de la première solution est sa simplicité. Il suffit d'une seule commande pour compiler notre programme constitué de trois fichiers :

```
gcc -Wall -Werror -W -o tp1 tp1*.c
```

Cependant, si notre programme est constitué de beaucoup de fichiers, cette compilation complète peut être très longue. À titre d'exemple, le noyau linux est constitué de 21629 fichiers * et peut mettre plusieurs dizaines de minutes à compiler, même sur une bonne machine †. Dans ce cas, la deuxième solution est plus adaptée, car il suffit de compiler uniquement le(s) fichier(s) modifié(s) séparément puis de faire l'édition de liens (on échappe néanmoins pas à une première compilation complète).

2.1 Méthode simple

Avant d'utiliser la deuxième méthode, nous allons corriger nos sources pour que la compilation simple de nos trois fichiers se passe correctement.

Q 6. Compilez les fichiers à l'aide de la commande donnée précédemment. Que ce passe-t'il ? Pouvez vous expliquer chacune des erreurs ?

Q 7. Corrigez ces erreurs en déclarant les prototypes des fonctions `fonction_a` et `fonction_b` aux emplacements adéquats.

Vous avez du, normalement, déclarer le prototype de la fonction `fonction_a` dans deux fichiers. Cette pratique est sujette aux erreurs de recopie ou de copier/coller et implique de devoir modifier les prototypes des deux fichiers si on change la fonction `fonction_a` (par exemple en lui ajoutant un paramètre).

Pour éviter cette duplication des prototypes, on les regroupe généralement dans un fichier entête qui porte le même nom que le fichier source qui contient la fonction, mais utilisant l'extension `.h`. Ce fichier `.h` peut alors être inclus dans les fichiers qui utilisent la fonction à l'aide de la directive préprocesseur `#include`.

Q 8. Créez un fichier `tp1_2.h` et placez y le prototype de la fonction `fonction_a`. Modifiez les autres fichiers en remplaçant le prototype de cette fonction par l'inclusion de votre nouveau fichier :

```
#include "tp1_2.h"
```

Notez l'utilisation du caractère `"` plutôt que des chevrons (`< >`) comme à la ligne 1 du programme d'origine. Lorsque les chevrons sont utilisés, le préprocesseur ne cherche les fichiers que dans une liste de répertoires contenant les fichiers d'inclusion des librairies standard ‡. Lorsque que les guillemets sont utilisés, le préprocesseur cherche également dans le répertoire courant. Dans les deux cas, le nom de fichier est un chemin relatif.

2.2 Compilation séparée

Nous allons maintenant utiliser la seconde méthode. L'option de compilation utilisée pour se limiter aux trois premières étapes de compilation (préprocesseur, compilation, assemblage) est l'option `-c`. Il demande à `gcc` de fabriquer un fichier dont l'extension est `.o`, appelé communément fichier *objet*. Il contient le code machine des fonctions situées dans le fichier compilé. Il porte par défaut le même nom que le fichier C, l'extension étant modifiée de `.c` à `.o`.

Pour compiler notre programme de cette façon, il faut exécuter `gcc` trois fois, une pour chaque fichier :

```
$ gcc -Wall -Werror -W -c tp1_1.c
$ gcc -Wall -Werror -W -c tp1_2.c
$ gcc -Wall -Werror -W -c tp1_3.c
```

Enfin, il faut *réunir* les trois fichiers objets générés en un exécutable à l'aide de l'édition de liens. Pour cette étape, on peut utiliser `gcc` avec uniquement l'option `-o`, en lui passant simplement en paramètre le nom des fichiers objets :

```
$ gcc -o tp1 tp1_1.o tp1_2.o tp1_3.o
```

Évidemment, il faut maintenant taper quatre commandes pour compiler. Si on revient sur l'exemple du noyau linux, il faudrait 21630 commande pour compiler ! Il faut donc automatiser tout ça.

∗. Dans sa version 4.2. Ce nombre compte tous les fichiers C dans l'arborescence du noyau (noyau en lui même + modules)

†. Core i7 2.4Ghz, SSD, 32G RAM

‡. ainsi que des répertoires ajoutés par l'option `-I` à la compilation

3 L'outil `make`

L'outil `make` permet d'automatiser la construction de fichiers. L'intérêt principal est que cet outil permet de décrire des dépendances entre les fichiers sources et les fichiers à construire. Grâce à cela, l'outil `make` est capable de ne reconstruire que les fichiers destinations dont les sources ont été modifiées.

Q 9. Lisez attentivement la courte introduction à l'outil `make` disponible à cette adresse : `http://cristal.univ-lille.fr/~hauspie/make.html`.

Q 10. Écrivez un fichier `Makefile` permettant de compiler notre programme.

Par la suite, vous devrez utiliser `make` pour tous les TP. Un TP qui ne compile pas avec un appel simple à la commande `make` ou qui compile sans les optiones `-Wall -Werror -W` ne sera pas validé.