Exercice 0 Petit exemple de compilation séparée

Dans cet exercice on applique la discipline de séparation du code décrite ci-dessus sur un tout petit exemple. Ce n'est pas très pertinent sur un si petit code, mais le but est de vous faire comprendre comment compiler ces fichiers, et de voir quelles erreurs de compilation peuvent apparaître.

Avant de commencer, téléchargez sur cahier de prépa l'archive `exo0_vide.zip`. Après extraction, vous devez avoir quatre fichiers : `a.h` , `a.c`, `test_a.c` et `Makefile`.

Question 1
Dans le fichier `a.c`, donnez la définition de la fonction nommée `affiche_a` qui prend en entrée un entier positif `n` et affiche `n` fois le caractère `'a'`. Si des instructions se trouvent déjà dans le corps de cette fonction, comme `printf("Définissez la fonction affiche_a\n");` ou `assert(false);`, supprimez-les quand vous définissez la fonction. Dé-commentez la déclaration de cette fonction dans le fichier `a.h`. Attendez pour compiler.

Question 2
Complétez la fonction `main` du fichier `test_a.c` pour qu'elle teste la fonction `affiche_a`. À nouveau supprimez les instructions devenues inutiles qui se trouvaient dans le corps de cette fonction. Attendez pour compiler.

Question 3
Essayez de compiler le fichier `a.c` comme on compilait jusqu'ici, c'est-à-dire par `gcc a.c -o a`.

Vous obtenez une erreur du compilateur `undefined reference to` `` `main` ``. Elle signifie que ce fichier ne peut être transformé en code exécutable puisqu'il n'y a pas de fonction `main`. En effet, ce code ne dit pas ce qui devrait se passer à l'exécution. Les fichiers de définition de fonction doivent être compilés différemment.

Question 4
Tapez `gcc -c a.c -o a.o`.
Si votre code est correct, rien ne s'affiche mais vous avez un fichier `a.o` dans votre dossier.

La commande `gcc -c` a généré un fichier appelé fichier objet, ici ici `a.o`, qui contient le code compilé des définitions de fonctions, mais la compilation s'est arrêtée avant de produire un fichier exécutable En particulier cette compilation admet que les fonctions déclarées sont définies quelque part, et attend la prochaine étape de compilation pour accéder à leurs définitions.

Question 5
Pour constater ce dernier point, dé-commentez dans le fichier `a.h` la déclaration de la fonction `affiche_point`. Ajoutez un appel à cette fonction dans votre définition de `affiche_a` dans `a.c`. Compilez à nouveau avec `gcc -c a.c -o a.o`. Normalement aucune erreur n'apparaît.

Question 6
Commentez la commande `#include "a.h"` dans le fichier `a.c` et compilez à nouveau ce fichier avec `gcc -c a.c -o a.o`

Cette fois une erreur (enfin un warning) apparaît, et indique notamment :
`implicit declaration of function ‘affiche_point’.`
Cela signifie que la déclaration de la fonction `affiche_point` est manquante.

Question 7

Au choix, supprimez l'appel à `affiche_point` et supprimez sa déclaration ou bien donnez une définition de `affiche_point` dans `a.c`. Dans les deux cas, dé-commentez la commande `#include "a.h"` dans le fichier `a.c`. Compilez à nouveau ce fichier avec `gcc -c a.c -o a.o`. vous ne devez plus avoir d'erreur ni de warning, et un fichier objet nommé `a.o` a du être créé.

Question 8
Puisque le fichier `test_a.c` code bien un programme (avec une fonction `main`), compilez-le avec la commande utilisée jusqu'ici : `gcc test_a.c -o test_a`.
Si vous obtenez l'erreur `implicit declaration of function 'affiche_a'`, c'est que vous avez oublié d'inclure `a.h`. Ajoutez `#include "a.h"` et recompilez.

Là vous obtenez une erreur `undefined reference to` `` `affiche_a` ``. C'est normal, la définition de cette fonction se trouve dans un autre fichier, `a.c`. Pour résoudre le problème, on pourrait ajouter `#include "a.c"` dans `test_a.c`. Cela aurait pour effet de copier le code du fichier `a.c` au début du fichier `test_a.c` avant de le compiler. Ainsi la définition de `affiche_a` serait présente, et la compilation produirait bien un exécutable `test.a`.
Cependant **on ne fera jamais ça**, car cette solution fait recompiler les fonctions définies dans `a.c` à chaque fois qu'on compile un fichier avec `#include "a.c"`. À la place on va utiliser le code déjà compilé des définitions de `a.c`, c'est-à-dire le fichier objet `a.o`.

Question 9
Tapez `gcc a.o test_a.c -o test_a`.
Vous obtenez un exécutable nommé `test_a`, lancez-le pour vérifier que tout fonctionne normalement.

Question 10
Modifiez la définition de la fonction `affiche_a` pour qu'elle affiche le même nombre de `'a'`, mais séparés par des espaces. Refaites les étapes de compilation nécessaires pour tester à nouveau votre code.

Vous avez du faire `gcc -c a.c -o a.o` puis `gcc a.o test_a.c -o test_a`. À chaque modification d'un fichier d'implémentation, il faut a minima recompiler le fichier objet correspondant, et en général recompiler le programme pour tester cette fonction. De plus, lorsqu'on aura un code plus important, les programmes de test pourront nécessiter de rassembler plusieurs fichiers objets (*i.e.* des `.o`), et l'oubli de l'un d'eux entraînera une erreur de type `undefined reference to`. C'est donc assez fastidieux. C'est pourquoi l'utilisation d'un `Makefile` peut se révéler profitable.
Un `Makefile` permet de définir des commandes qui lancent des commandes de compilation, qu'on peut voir comme des raccourcis. Par exemple,

```
a.o : a.c
  gcc -c a.c -o a.o
```

définit le raccourci `make a.o` qui lancera la commande `gcc -c a.c -o a.o`.

Question 11
Supprimez vos fichiers compilés : `a.o` et `test_a`. Tapez `make a.o` dans le terminal *Vous pouvez utiliser l'autocomplétion grâce à la touche tabulation.*

Vous voyez apparaître `gcc -c a.c -o a.o`, c'est la commande qui a été lancée. Vous pouvez aussi constater que le fichier `a.o` a bien été compilé.

Question 12
Supprimez le fichier que vous venez de compiler : `a.o`. Tapez `make test_a` dans le terminal.

Vous voyez apparaître `gcc -c a.c -o a.o`, puis `gcc test_a.c a.o -o test_a`. Ce sont les deux commandes qui ont été lancées. Vous pouvez aussi constater que les fichiers `a.o` et `test_a` ont bien été compilés. En regardant dans le fichier `Makfile` comment est définie le raccourci, vous trouvez la définition suivante :

```
test_a : a.o test_a.c
  gcc test_a.c a.o -o test_a
```

Cela peut vous surprendre, car le raccourci `make test_a` ne semble pas appeler `gcc -c a.c -o a.o`. Mais on a omis de considérer l'information placée après le `:`, elle indique quels fichiers doivent être compilés et à jour pour lancer la commande indiquée à la ligne suivante. Ici, on a précisé qu'il faut que le fichier `a.o` soit compilé et à jour pour lancer `gcc test_a.c a.o -o test_a`, on parle de dépendance entre `test_a` et `a.o`. Comme le fichier `a.o` n'existait pas, `make` a cherché à le compiler, et pour ça il a suivi ce qu'on avait donné plus haut comme instructions (en définissant le raccourci `make a.o`), à savoir `gcc -c a.c -o a.o`.

Question 13
Supprimez le fichier `test_a` seulement, et relancez `make test_a`.

Cette fois le fichier `a.o` était déjà présent et à jour, le raccourci `make test_a` a donc seulement déclenché la compilation `gcc test_a.c a.o -o test_a`.
On peut se demander à quoi sert de préciser `a.c` et `a.h` comme dépendances de `a.o`. Cela permet de préciser que le fichier `a.o` doit être recompilé si le fichier `a.c` ou le fichier `a.h` a été modifié depuis la dernière compilation de `a.o`.

Question 14
Supprimez à nouveau le fichier `test_a`, puis modifiez le fichier `a.c` et enregistrez ces modifications. Relancez `make test_a`. *Vous pouvez aussi faire la même chose en modifiant* `a.h`.

Cette fois le fichier `a.o` était déjà présent, mais il n'était pas à jour, `make` a alors cherché à le recompiler, et pour ça il a suivi ce que l'on a donné plus haut, c'est pourquoi la commande de compilation `gcc -c a.c -o a.o` a été effectuée avant `gcc test_a.c a.o -o test_a`.

Si les dépendances entre fichiers compilés sont mal définies dans le `Makefile`, il peut arriver qu'on modifie un fichier mais qu'on n'observe pas les modifications à l'exécution du programme, car il manque une re-compilation quelque part. Si ce genre de chose arrive, ou justement pour l'éviter, on peut supprimer tous les fichiers compilés, et tout recompiler à chaque fois. Pour cela, on définit un raccourci `make clean` dans le `Makefile`, qui supprime tous les fichiers d'extension `.o` (grâce à la commande `rm -f *.o`) et qui supprime un par un les fichiers exécutables (ici seulement `test_a` grâce à `rm -f test_a`).

Question 15
Supprimez tous les fichiers compilés grâce au raccourci `make clean`. Constatez que ça a marché, relancez des compilations... bref familiarisez vous avec les commandes `make`.

À retenir

- La compilation se fait en deux étapes.
- La première nécessite que tout soit bien déclaré, et si ce n'est pas la cas on obtient des erreurs de type `implicit declaration of ...`.
- La deuxième nécessite que les fonctions appelées soient bien définies, et doit avoir accès à ces définitions, si ce n'est pas le cas on obtient des erreurs de type `undefined reference to ...`.
- La première étape produit un fichier compilé intermédiaire, qu'on nomme avec l'extension `.o` par convention.
- La deuxième étape produit un fichier exécutable, qu'on nomme sans extension par convention. Pour cela, le point d'entrée est la fonction `main`. Le code qu'on donne à compiler doit de ce fait contenir une et une seule fonction `main`.
- le `Makefile` permet de définir des raccourcis qui lanceront les commandes de compilation dont on a besoin.

En pratique (sans `Makefile`)

→ on déclare les fonctions dans un fichier `xxx.h`

→ on définit toutes les fonctions déclarées dans `xxx.h` dans un fichier `xxx.c` qui inclut `xxx.h` grâce à `#include "xxx.h"`

→ on compile ces définitions via la commande `gcc -c xxx.c -o xxx.o`.

→ on teste toutes ces fonctions dans la fonction `main` d'un fichier `test_xxx.c` qui inclut lui aussi `xxx.h` grâce à `#include "xxx.h"`

→ on compile ce programme de test avec les définitions de fonctions déjà compilées via la commande `gcc xxx.o test_xxx.c -o xxx`

→ on lance le programme de test via `./xxx`