ENSTA Paris
2023-2024

Résolution de problèmes algorithmiques – IN103

Alexandre Chapoutot

Feuille d'exercices 1

Objectif(s)

- ★ Comprendre la compilation séparée
- ★ Écrire un Makefile

Exercice 1 – Compilation séparée

Allez dans le répertoire `exo1`. Vous y trouverez plusieurs fichiers
- `bubble_sort.c`
- `my_prog.c`
- `simulation.c`

Question 1
Donnez la ligne de commande pour compiler le fichier `my_prog.c` pour générer un exécutable nommé `my_prog.x`.

Question 2
Compilez le fichier `my_prog.c`, que se passe-t-il ? Essayez d'expliquer l'erreur.

Question 3
Compilez maintenant en utilisant les deux fichiers `bubble_sort.c` et `my_prog.c`. Essayez d'expliquer ce qui se passe.

Question 4
Supprimez le fichier exécutable `my_prog.x` et recompilez le programme comme précédemment en ajoutant l'option `-Werror`. Que se passe-t-il ?

Question 5
En utilisant la directive du pré-processeur `#include` qui inclue le contenu d'un fichier dans un autre, utilisez cette directive pour inclure le fichier `bubble_sort.c` dans le fichier `my_prog.c`. Recompilez comme précédemment en utilisant les deux fichiers `bubble_sort.c` et `my_prog.c`. Que se passe-t-il ? Essayez d'expliquer ce qui se passe.

Question 6
Les précédentes questions permettent de mettre en avant les concepts de
- **déclaration** de fonctions, l'étape consistant à donner le *prototype* (aussi appelé *signature*) d'une fonction.
- **définition** de fonctions, l'étape consistante à donner le *corps* (c'est-à-dire le code) de la fonction.

En langage C, les déclarations de fonctions sont stockés dans des fichiers `.h` et les définitions de fonctions dans des fichiers `.c`.

Définissez un fichier `bubble_sort.h` contenant la déclaration de fonction `bubble_sort` et l'inclure dans le fichier `my_prog.c`. Utilisez la ligne de compilation avec les deux fichiers `bubble_sort.c` et `my_prog.c`. Que se passe-t-il ?

Exercice 2 – Deboggage de programmes

Allez dans le répertoire `exo2`, dans lequel plusieurs fichiers sont présents. Ces fichiers permettent de calculer une base orthornormale de vecteurs à l'aide de l'algorithme de Gram-Schmidt [1].

Plusieurs erreurs ont été introduites dans les fichiers ce qui empêchent la compilation. Après une tentative de compilation, suivez les messages d'erreur pour les corriger.

Question 1
Donnez la ligne de commande pour compiler ces fichiers en un programme nommé `my_prog.x`

Question 2
Quelle est la première erreur qui apparaît dans le processus de compilation ? Comment pouvons-nous la corriger ?

Question 3
Après avoir corrigé l'erreur précédente, relancer le processus de compilation. Quelle est la nouvelle (première) erreur qui survient ? Comment pouvons nous la corriger ?

Question 4 – Pour aller plus loin (Optionnel)
Note erreur qui n'est pas générée sous Linux mais uniquement sous Mac OS. Après avoir corrigé l'erreur précédente, relancer le processus de compilation en ajoutant l'option `-std=c99 -pedantic` (révision du langage C datant de 1999 et une application stricte de la norme). Quelle la nouvelle (première) erreur survient ? Comment pouvons nous la corriger ?

Exercice 3 – Makefile
Allez dans le répertoire `exo3` dans lequel vous trouverez un ensemble de fichiers permettant de calculer les racines d'un polynôme du second degré à cœfficients réels. Plus précisément, nous avons les fichiers suivants :

- `complex.h` et `complex.c` qui contiennent respectivement les déclarations et les définitions des fonctions manipulant des nombres complexes ;
- `solve.h` et `solve.c` qui contiennent respectivement les déclarations et les définitions des fonctions qui permettent de résoudre une équation du second degré ;
- `my_prog.c` qui contient la fonction principale.

L'objectif de cet exercice est d'écrire un fichier `Makefile` qui permet d'automatiser la compilation. Différentes versions seront écrites pour introduire progressivement les constructions des fichiers `Makefile`.

Question 1
Avant d'écrire le `Makefile` donnez les lignes de compilations pour générer

- le fichier `complex.o` ;
- le fichier `solve.o` ;
- le fichier `my_prog.o` ;
- le programme final `my_prog.x` qui nécessite l'utilisation des fichiers précédents.

Question 2
Pour rappel un fichier `Makefile` est constitué de règles de la forme :

```
cible: dépendances
      actions
```

Créez un fichier `Makefile` avec plusieurs règles pour générer les différents fichiers `complex.o`, `solve.o`, `my_prog.o` et `my_prog.x`

Exécutez le Makefile en lançant la commande `make`. Que se passe-t-il ?

Question 3
Dans le fichier `Makefile` ajoutez une règle `clean` qui supprime les fichiers `*.o` générés par la compilation. Ajoutez également une règle nommée `realclean` qui dépend de la règle `clean` et qui supprime le fichier `my_prog.x`

Question 4
Un fichier `Makefile` autorise l'utilisation de variables. Des variables spécifiques `CC` ou `CFGLAS` permettent de renseigner quel commande à utiliser pour compiler les programmes écrits en langage C et quelles options sont à utiliser lors de la compilation. Ces variables permettent également de factoriser les règles pour ne pas dupliquer l'information.

Modifiez les directives du fichier `Makefile` pour utiliser les variables `CC` et `CFLAGS`.

Question 5 – Pour allez plus loin (Optionnel)
La génération de fichiers `*.o` à partir de fichiers `*.c` suivent un même schéma. Dans les fichier `Makefile` il est possible

1. https://fr.wikipedia.org/wiki/Algorithme_de_Gram-Schmidt

de définir des “pattern” de règles. En particulier, le caractère % est un joker pour représenter le nom d'un fichier. Par exemple, `%.o: %.c` indique une règle dont le produit (un fichier .o) dépend d'un fichier .c qui a le même nom (mais pas la même extension).

Une conséquence d'utilisation des ces patterns de règles de compilation est la difficulté de nommer les fichiers qui sont impliqués dans la production d'une règle implicite. Il y a plusieurs variables qui existent pour cela :

— `$@` correspond à l'élément associé à une cible d'une règle ;
— `$<` correspond au premier élément de la liste des dépendances d'une règle ;
— `$^` correspond à la liste des dépendances d'une règle.

Modifiez le fichier `Makefile` pour rendre générique la production de fichiers `.o`

Question 6 – Pour allez plus loin (Optionnel)

Pour continuer l'utilisation des variables dans les fichiers `Makefile` il est possible de transformer une liste de noms de fichier. Par exemple, il est possible de transformer une liste liste de fichiers `.c` en une liste de fichier `.o`.

La construction à utiliser est la fonction `patsubst` qui à la signature suivante : `\$(patsubst pattern,replacement,text)`. Son fonctionnement est (traduction de la documentation [2])

> Trouve les mots séparés par des espaces dans la variable `text` qui correspondent au modèle donné par `pattern` et les remplace par le contenu de `replacement`. Ici, le motif peut contenir un '%' qui agit comme un caractère générique, correspondant à n'importe quel nombre de caractères dans un mot. Si le remplacement contient également un '%', le '%' est remplacé par le texte qui correspond au '%' du modèle donné par `pattern`.

Modifiez le fichier `Makefile` en créant une variable `SRC` qui contient la liste des fichiers `.c` et en créant une variable `OBJ` qui est générée à partir de la variable `SRC` qui correspond aux fichiers `.o`

POUR S'ENTRAÎNER À LA MAISON

Exercice 1 – Rendre un programme modulaire

Allez dans le répertoire `exo1`, vous y trouverez un fichier `simulation.c`.

Question 1

Transformez le fichier `simulation.c` en plusieurs fichiers : `heun-euler.h`, `heun-euler.c` et garder seulement dans le fichier `simulation.c` la fonction `main`.

APPROFONDISSEMENT

Exercice 1 – Masquer des fonctions de la libc – version 1

Dans cet exercice, nous allons masquer des définitions de fonctions la bibliothèque standard. Attention toutes les fonctions de la bibliothèque standard peuvent être aussi définies comme des macros à paramètres. Il convient donc de prendre quelques précautions d'usage. Nous travaillons dans le répertoire `exo4` pour cet exercice.

Question 1 – Préliminaires

Dans un fichier `plus.c`, écrivez une macro à paramètre nommée `my_plus(a,b)` qui s'évalue en `a+b`. Essayer ensuite d'y définir une fonction `int my_plus(int a, int b)`. Que se passe-t-il à la compilation ?

Question 2

Définissez dans un fichier `my_malloc.c` une fonction `void* malloc(size_t)` qui affiche la valeur du paramètre de type `size_t` et retourne la valeur `NULL`.

Question 3

Définissez, dans un fichier `my_main.c`, une fonction `main` qui fait un appel à la fonction `malloc`.

Question 4

Transformez les deux fichiers `my_main.c` et `my_malloc.c` en fichiers objets (`my_main.o` et `my_malloc.o`).

2. https://www.gnu.org/software/make/manual/html_node/Text-Functions.html

Question 5
Créez un programme `my_main` en liant uniquement `my_main.o`.

Question 6
Créez un programme `my_main2` en liant `my_main.o` et `my_malloc.o`.

Question 7
Comparez les deux exécutions de ces deux programmes, comment les expliquez-vous ?

Exercice 2 – Masquer des fonctions de la libc – version 2
Nous travaillons dans le répertoire `exo5` pour cet exercice.

Dans l'exercice précedent, nous avons vu comment, lors de l'édition des liens d'un programme, le masquage des fonctions de la bibliothèque standard C était possible en donnant une définition propre des fonctions. Cependant, la technique utilisée ne peut s'appliquer qu'aux programmes que nous liions nous-mêmes.

Une autre technique permet de masquer les fonctions utilisées par un programme lors du chargement de bibliothèques dynamiques, sans avoir à modifier le programme.

Dans cet exercice, une modification de la commande `date` va être réalisée pour lui faire indiquer une autre date que la date courante, par exemple, le 1er janvier 2030, ce qui donnerait

```
% date
mar jan  1 00:00:00 CET 2030
% date -I
2030-01-01
```

Pour réaliser cette modification, il est nécessaire de
- trouver la fonction utilisée par la commande `date` retournant la date du système
- écrire notre propre version de cette fonction retournant la date choisie
- créer une bibliothèque dynamique contenant cette fonction
- pré-charger cette bibliothèque lors de l'exécution de la commande `date` afin que notre fonction remplace celle du système.

Question 1 – Inspection

Plusieurs outils de mise au point sont accessibles sous linux pour étudier les programmes :
- `strace` : intercepte et enregistre les appels système lancés par un processus
- `ltrace` : intercepte et enregistre les appels à des bibliothèques dynamiques qui sont appelées par un processus
- `objdump` : affiche des informations provenant de fichiers objets

Utilisez ces fonctions pour découvrir quelle fonction est utilisée par la commande `date` pour récupérer la date du système. Pensez à utiliser la commande `grep` sur la sortie de ces commandes (le mot-clé `time` semble être une bonne piste)

Question 2
Selon les versions installées, la commande `date` peut utiliser la fonction `clock_gettime`, `time` ou `gettimeofday`. En fonction du résultat à la question précédente, écrivez une version de la fonction dans un fichier `mydate.c`. Pour information, la date du 1er janvier 2030 est associée au nombre 1893452400.

Question 3
Générez une bibliothèque dynamique à partir de votre fonction précédente. Pour cela, il suffit d'ajouter l'option `-shared` au compilateur `gcc` et de générer un fichier avec l'extension `.so`

Question 4 – Préchargement de bibliothèques dynamiques
Cherchez sur la page de manuel de `ld.so` le nom d'une variable d'environnement qui peut servir à forcer la commande `date` à utiliser votre bibliothèque. Utilisez-la pour lancer la commande `date`.