TP 1 : La compilation séparée, les fichiers include, les makefiles, l'utilisation d'un deboggeur

1 La compilation séparée

Il est possible en langage C d'utiliser la compilation séparée et de construire ses propres bibliothèques.

Commençons par un exemple simple. Nous allons utiliser un programme principal qui fait appel à une fonction `hello_world`. présente dans un autre fichier C.

Le programme principal va donc donner la chose suivante :

```
// princ1.c

#include <stdio.h>

extern void hello_world();

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

Le second fichier lui contient la définition de la fonction `hello_world`.

```
// hello1.c

#include <stdio.h>

void hello_world()
{
  printf("Hello World\n");
}
```

1. Créez les 2 fichiers ci-dessus (`princ1.c` et `hello1.c`).
2. Tentez de compiler `princ1.c` à l'aide de la commande `gcc -Wall princ1.c`. Que constatez-vous ? Pourquoi ? Meme question pour le fichier `hello1.c`.
3. Réalisez une compilation **séparée** de chaque fichier à l'aide de la commande `gcc -Wall -c`. Que constatez-vous ? Quel fichier est généré ?
4. Réalisez de nouveau une compilation séparée en enlevant dans `princ1.c` la ligne `extern void hello_world`. Que constatez-vous ? Le fichier objet est-il généré ?
5. Réalisez maintenant une édition des liens complète (génération du fichier binaire) à l'aide de la commande

   ```
   gcc -Wall princ1.o hello1.o -o princ1
   ```

   On peut également réaliser la compilation séparée ainsi que l'édition des liens en utilisant la commande

   ```
   gcc -Wall princ1.c hello1.c -o princ1
   ```

6. Modifiez le fichier `princ1.c` pour y inclure volontairement une erreur. Au lieu d'appeler la fonction `hello_world`, appelez la fonction `bye_world`. A votre avis, où va se produire l'erreur ? Lors de la compilation séparée ? Lors de l'édition de liens ?

2 Les fichiers `include`

En général, pour fabriquer une bibliothèque, on utilise à la fois un fichier d'implémentation `.c` et un fichier d'interface `.h`. Le fichier d'interface contient juste les en-têtes des fonctions (ainsi qu'éventuellement des déclarations de types et de constantes), tandis que le fichier d'implémentation contient les définitions complètes des fonctions.

1. Ecrivez les fichiers d'interface et d'implémentation contenant 3 fonctions. La première (`Affiche`), affiche la chaîne de caractères qui lui est passée en paramètre, la seconde (`Fact`) calcul et renvoie le factoriel de l'entier qui lui est passé en paramètre et la troisième (`Date`) vous affiche la date et l'heure du jour (on appelera pour cela la commande `/bin/date` à l'aide de la fonction `system`).
2. Testez ces trois fonctions depuis un programme principal que vous écrirez.

3 Les inclusions multiples

Prenons un exemple de code plus compliqué, dans lequel une bibiothèque est utilisée (elle est composée des 2 fichiers d'interface et d'implémentation) ainsi qu'un fichier `.h` destiné à contenir un ensemble de types et constantes utiles pour le programme complet.

1. Créez le fichier `constantes.h` qui contient la constante `MAXNOM` et le type `eleve`. Ce type est une structure composée de 2 champs : le nom de l'élève (chaîne de caractères) et sa promo (un entier).
2. Créez la bibliothèque composèe des fichiers `eleves.h` et `eleves.c`. qui définit les fonctions `init_eleve` et `affiche_eleve`. Dans le fichier `eleves.h`, vous aurez besoin d'inclure le fichier `constantes.h`.
3. Ecrivez maintenant un programme principal qui utilise cette bibliothèque. En particulier, dans ce programme, incluez le fichier d'en-tête de la bibliothèque (`eleve.h`) mais aussi le le fichier de constantes, puisque vous avez à manipuler la structure `eleve` dans ce programme. Que constatez-vous ? Pourquoi ?
4. La résolution du problèmes des inclusions multiples se fait en insérant systématiquement dans tout fichier .h la directive suivante du pro-processeur :

   ```
   #ifndef CONSTANTES_H
   #define CONSTANTES_H

   // code de constantes.h

   #endif
   ```

   Grâce à cette directive, le fichier `constantes.h` n'est inclus qu'une seule fois. Plus précisément, s'il est inclus 2 fois, il sera inclus **vide** à la seconde fois.
   La macro `CONSTANTES_H` est définie arbitrairement par le programmeur. En général, cette constante est liée au nom du fichier dans lequel la macro.

4 L'outil `Make` et les fichiers `makefile` (Merci B. Lussier)

Make est un outil de programmation, dont une version est disponible pour les systèmes Unix sous la licence GNU (c'est-à-dire libre à l'utilisa-

tion, la copie et la redistribution). Il permet d'automatiser des tâches, telles que la compilation de programmes ou l'archivage de fichiers. Pour fonctionner, Make a besoin d'un certain nombre d'informations qu'il cherchera par défaut dans un fichier nommé `makefile` ou `Makefile` (il est ainsi recommandé d'utiliser ces noms de fichiers). Le but de cette section est de donner les informations nécessaires à la création de fichiers `Makefile` pour la compilation de projets C peu complexes.

4.1 Syntaxe

Un fichier `Makefile` est constitué de plusieurs règles, chacune définie par la syntaxe suivante (une tabulation *(tab)* doit nécessairement précéder chaque commande) :

```
cible1 : [dépendance1] [dépendance2] ...
(tab) [commande1]
(tab) [commande2]
...

cible2 : [dépendanceA] [dépendanceB] ...
(tab) [commandeA]
(tab) [commandeB]
...
```

La commande shell `make cible1` entraîne l'évaluation successive de chacune de ses dépendances (`dépendance1`, `dépendance2`, etc...). Pour une compilation par exemple, Make compare la date de modification de la cible avec celle de chaque dépendance : si au moins une dépendance s'avère plus récente que la cible, les commandes de la cible (`commande1`, `commande2`, etc...) sont exécutées.

Si une dépendance est la cible d'une autre règle du `Makefile`, cette cible est à son tour évaluée, et, si nécessaire, ses commandes sont exécutées avant celles de la cible originale. Par exemple, si `dépendance2` est égale à `cible2` (et si une dépendance de `cible2` est plus récente que `cible2`), `make cible1` entraînera l'exécution successive des commandes `commandeA`, `commandeB`, etc... puis `commande1`, `commande2`, etc...

Si une cible ne contient pas de dépendance, ses commandes sont exécutées à chaque appel de la cible.

Pour éviter d'écrire de longues commandes ou une longue liste de dépendances sur la même ligne, on peut utiliser le caractère de continuation de ligne \.

4.2 Exemple simple

On considère un programme `etudiant_INSA.c` qui utilise les couples de fichiers (`.c` et `.h`) : `traîner_sur_internet` et `manger_des_pâtes`. Le fichier `Makefile` suivant permet de compiler l'ensemble des fichiers par la commande shell `make etudiant_INSA`.

```
etudiant_INSA : etudiant_INSA.o \
                traîner_sur_internet.o manger_des_pâtes.o
(tab) gcc etudiant_INSA.o traîner_sur_internet.o \
                manger_des_pâtes.o -o etudiant_INSA

etudiant_INSA.o : etudiant_INSA.c
(tab) gcc -c etudiant_INSA.c

traîner_sur_internet.o : traîner_sur_internet.c
(tab) gcc -c traîner_sur_internet.c

manger_des_pâtes.o : manger_des_pâtes.c
(tab) gcc -c manger_des_pâtes.c
```

4.3 Cibles conventionnelles

Les cibles `default`, `all`, `clean` et `install` se retrouvent couramment dans un `Makefile`. C'est souvent une bonne idée de les implanter, lorsqu'elles sont applicables (`make install`, par exemple, n'est généralement pas nécessaire dans un TP).

- `default` : décrit les cibles évaluées et exécutées par la commande shell `make` appelée sans argument,
- `all` : réalise l'ensemble des cibles décrites par le fichier `Makefile` (par exemple, compile l'ensemble des fichiers d'un projet C),
- `clean` : supprime les fichiers créés par l'exécution de Make, par exemple pour repartir d'un environnement "propre" (en C, `make clean` supprime généralement les fichiers objets et les exécutables)
- `install` : installe une application ou un programme, idéalement au bon endroit (le problème vient souvent de ce que le "bon endroit" dépend du système qu'on utilise).

4.4 Règles génériques

Une règle générique ne mentionne pas de cibles particulières, mais porte sur un ensemble de cibles. En C, une règle générique permet généralement

de décrire la compilation de fichiers sources `.c` en fichiers objets `.o`. Les variables automatiques `$^` et `$<` sont souvent utilisées pour cela : `$^` représente l'ensemble des dépendances de la règle, et `$<` représente uniquement la première dépendance. `$@` est une autre variable automatique souvent utilisée, représentant le nom de la cible dans une règle.

Ci-dessous, un exemple de règle générique, créant les fichiers objets à partir des fichiers sources de même nom. La variable automatique `%` y est utilisée pour représenter la même valeur du côté de la cible et du côté des dépendances.

```
%.o : %.c
(tab) gcc -c $<
```

4.5 Exemple plus concret de `Makefile`

En reprenant le programme `etudiant_INSA` et les fichiers dont il dépend `traîner_sur_internet` et `manger_des_pâtes`, on obtient :

```
default : etudiant_INSA

all : etudiant_INSA

clean :
(tab) rm *.o etudiant_INSA

etudiant_INSA : etudiant_INSA.o \
                traîner_sur_internet.o manger_des_pâtes.o
(tab) gcc etudiant_INSA.o traîner_sur_internet.o \
                manger_des_pâtes.o -o $@

%.o : %.c
(tab) gcc -c $<

traîner_sur_internet.o : traîner_sur_internet.h

manger_des_pâtes.o : manger_des_pâtes.h
```

4.6 Exercice

En reprenant les fichiers que vous avez écrits dans la section 3, écrivez le fichier `makefile` qui vous permet de compiler automatiquement vos fichiers et de générer le programme binaire correspondant.

5 Utilisation du deboggeur `gdb`

Un deboggeur est un programme qui permet l'analyse de programmes binaires en cours d'exécution. Il est ainsi possible, lors de l'exécution d'un programme, de le stopper temporairement pour analyser la valeur de certaines variables, le contenu de certaines adresses, etc.

Pour utiliser un deboggeur sous Unix, il faut au préalable compiler le programme que l'on souhaite debogger avec l'option `-g` :

```
gcc -Wall -g toto.c -o toto
```

Ensuite, il suffit de lancer le deboggeur sur le programme binaire de la façon suivante :

```
gdb toto
```

Voici à présent quelques commandes usuelles du deboggeur :

La pose de points d'arrêt

- `(b)reak [line]` place un point d'arrêt à la ligne indiquée.
- `(b)reak [fonc]` place un point d'arrêt à la fonction spécifiée.
- `info break` indique où sont définis les point d'arrêts.
- `clear [line|fonc]` supprime un point d'arrêt.

Exécution du programme

- `run < file` lance le programme avec une redirection de l'entrée standard.
- `(n)ext` exécute une instruction sans rentrer dans le code des fonctions.
- `(s)tep` exécute une instruction en entrant dans le code des fonctions.
- `(c)ount` continue l'exécution du programme.
- `jump [line]` saute a la ligne indiquée (modifie le compteur ordinal).
- `(l)ist` liste le code source du programme.

Examen des données

- `(p)rint [exp]` affiche la valeur de l'expression.
- `(p)rint [*tab@num]` affiche num valeurs du tableau `tab`.
- `display [exp]` affiche la valeur de l'expression après chaque arrêt.
- `undisplay [num]` supprime un display.
- `set [exp=value]` modifie la valeur d'une variable.

Nous allons à présent utiliser le deboggeur au travers d'exemples simples. Soit le programme suivant :

```
#include <stdio.h>

int main()
{
  int i;
  int j=200;

  for (i=0;i<10;i++) {
    j--;
  }
  return 0;
}
```

1. Codez le programme ci-dessus et posez un point d'arrêt à l'intérieur de la boucle de façon à visualiser à chaque tour de boucle la valeur de la variable `j`.
2. Ajoutez la déclaration `int tab[10];` ainsi que la ligne `tab[i]=i;` dans la boucle . Visualisez à l'aide du debogger le remplissage du tableau à chaque itération de la boucle.
3. Ajoutez un pointeur de type `int * p;` qui pointe à chaque itération de la boucle sur l'élément du tableau `tab` qui vient d'être modifié. Visualisez cet element.