TP 2 et 3

Introduction à C

Partie A : prise en main de gcc et makefile

L'objectif de cette partie est de vous familiariser avec le langage C et sa chaîne de développement basée sur le compilateur **`gcc`**, et de vous apprendre à utiliser des makefile lorsque vous développez des programmes en C.

1. Programme C formé d'un seul module (20 minutes)

1. Créez un répertoire sur votre compte utilisateur, et copiez-y (en utilisant la commande **`cp -r`**) le contenu du répertoire `/share/l3info/syr1/tp_c`
2. Placez-vous dans le répertoire copié et essayez de compiler le programme **`hms.c`**, à l'aide de la commande

   ```
   gcc –Wall nomfichier.c
   ```

   Vérifiez la signification de l'option `Wall`...
3. Editez le code source du programme, que constatez-vous ? Corrigez l'indentation du programme, ainsi que ses erreurs. ~~Pour corriger l'indentation, vous pouvez utiliser la commande **`indent`**, à condition d'avoir fermé votre éditeur~~.
4. Relancez une compilation, afin de vous assurer de la correction du programme, et à l'aide de la commande **`ls -ail`**, retrouvez le nom du fichier produit lors de cette étape.

   La commande précédente en (4) a généré un fichier **`a.out`** qui est l'exécutable correspondant au programme **`hms.c`**.

   On peut l'exécuter par la commande **`./a.out`**.

   Si on souhaite préciser un nom d'exécutable particulier, on utilise l'option **`-o`** de `gcc` :

   ```
   gcc –o nomprog –Wall nomfichier.c
   ```

   La commande produit alors un exécutable appelé **`nomprog`** (sous Linux, les exécutables n'ont pas l'extension **`.exe`** comme c'est le cas sous DOS/Window). L'exécution de ce programme se fait alors simplement par la commande :

   ```
   ./nomprog
   ```
5. Recompilez le programme de manière à produire un exécutable nommé **`hms`**, et vérifiez son bon fonctionnement en l'exécutant dans le terminal de commande.

2. Programme C formé de plusieurs modules (20 minutes)

En règle générale, un module est formé d'un fichier .c contenant le code source du module, et d'un fichier d'entête (extension .h), qui permet d'exporter les définitions des identificateurs du module (données, fonctions, etc.).

Lorsque l'on compile un module, on utilise l'option **-c** du compilateur ; celui-ci produit alors un fichier intermédiaire (fichier objet d'extension **`.o`**) qui n'est pas directement exécutable.

Créez un répertoire sur votre compte utilisateur, et copiez-y (en utilisant la commande **`cp -r`**) le contenu du répertoire `/share/l3info/syr1/tp_liste`

Placez-vous ensuite dans le répertoire nouvellement crée, celui-ci contient les codes source (partiels) d'un TP sur la mise en œuvre de listes chaînées en C.

1. Lancez la compilation de **`src/test_list.c`** par la commande ci-dessous.

   ```
   gcc –c src/test_list.c
   ```

   Celle-ci produit une erreur de compilation, d'où provient cette erreur ?

2. Comment corriger cette erreur ?

- **Indice n°1**: on dispose d'un fichier d'entête **`list.h`** qui définit les variables et les identificateurs exportés respectivement par le module **`list.c`**, et de la directive #**`include <fichier.h>`**.
- **Indice n°2**: il faut parfois indiquer au compilateur où trouver les fichiers d'entête, pour cela on utilise l'option `–I` de `gcc`. Par exemple `gcc –I./include` indique qu'il faut aller chercher les fichiers entêtes `.h` dans le répertoire `./include`..

3 Une fois le problème résolu, compilez (avec l'option –c) le fichier `src/list.c` ? Est-il exécutable ?

4 Reliez maintenant les deux modules pour obtenir le programme exécutable, à l'aide de la commande ci-dessous :

```
gcc –o test_list test_list.o list.o
```

5 Exécutez ce programme par la commande ci-dessous, et observez les résultats sur la sortie standard

```
./test_list
```

6 Le message *segmentation fault* indique que le programme a exécuté un accès mémoire à une adresse illégale, soit à cause d'un débordement de tableau, ou de l'utilisation d'un pointeur mal ou non initialisé.

7 Recompilez le module `test_list.c` en utilisant l'option `–Wall` qui demande au compilateur d'afficher tous les messages d'avertissement (*warning*) repérés lors de la compilation du module. Qu'en déduisez-vous ?

8 En utilisant les résultats de la question précédente, corrigez les erreurs présentes dans le module `test_list.c` et vérifiez ensuite le bon fonctionnement du programme (pour répondre à cette question il peut-être utile de lire la description fournie en partie B. du sujet).

3. Utilisation de l'outil make (40 minutes)

Chaque fois que l'on modifie un des fichiers source (.c ou .h), il faut recompiler les modules dépendant de ces fichiers, et refaire l'édition de lien pour obtenir un exécutable à jour. Cette tâche est très fastidieuse à faire à la main, surtout quand le programme comporte de nombreux modules (à titre d'exemple, le noyau du système d'exploitation Linux contient plusieurs centaines de modules C).

Il est bien sûr possible de tout recompiler chaque fois, mais ce genre d'approche atteint rapidement ses limites dès lors que l'on travaille sur des programmes conséquents, car le temps de compilation devient prohibitif.

Le programme **`make`** dont le fonctionnement a été vu en cours permet justement d'automatiser cette tâche, en ne recompilant à chaque fois que ce qui doit l'être.

1. Complétez le fichier `makefile` du répertoire `tp_liste` de manière à permettre l'assemblage et l'édition de lien du programme vu en 2 ; celui-ci devra s'assurer :
   - Que les fichiers `.o` produits lors de la compilation soient stockés dans le répertoire `obj` prévu à cet effet,
   - Que le programme exécutable soit stocké dans `./bin` sous le nom `test_list`
   - Que les fichiers objets et l'exécutable soient à jour par rapport à leur code source `.c` et aux fichiers d'entêtes (`.h`) dont ils dépendent.
2. Vérifiez le bon fonctionnement de votre `makefile` en modifiant un des fichiers puis en relançant le `makefile` à l'aide de la commande **`make`**.
3. Complétez dans le fichier `makefile` l'entrée `clean` permettant d'effacer tous les fichiers objets (**`*.o`**) ainsi que l'exécutable produit.
4. Complétez dans le fichier `makefile` l'entrée `listing` permettant de produire un fichier **pdf** à partir des sources des fichiers .c et h, en utilisant la séquence de commandes :

   ```
   a2ps -o listing.ps include/list.h src/list.c src/test_list.c
   ps2pdf listing.ps
   rm listing.ps
   ```

5. Vérifiez le bon fonctionnement de cette règle en l'appelant par la commande :

   ```
   make listing
   ```

Partie B : listes chaînées en C

1. Module de mise en œuvre de listes chaînées

On s'intéresse à une mise en œuvre de listes chaînées en langage C, similaire à celle étudiée en TD. On rappelle que le type liste est représenté à l'aide d'une structure `list_elem_t` dont la définition est donnée :

```
typedef struct s_list {
   int  value;              // identifiant du maillon
   struct s_list * next;    // pointeur sur le maillon suivant
} list_elem_t;
```

Cette représentation est illustrée par la Fig. 1 ci-dessous.

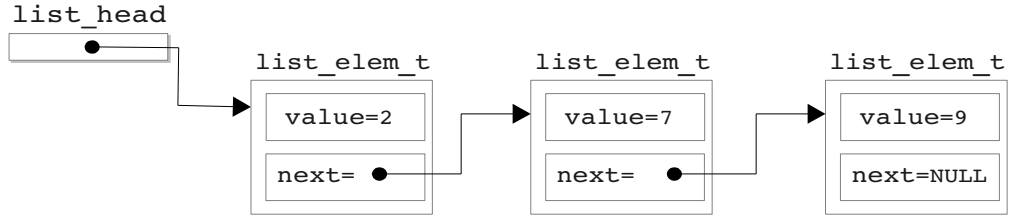

Figure 1. Représentation en mémoire d'une liste chaînée.

Le module `list.c` fournit les fonctions suivantes étudiées en TD (pour une description plus détaillée des spécifications, veuillez vous reporter aux sources du module `list.c`). Les fonctions `insert_head`, `create_element` et `free_element` sont déjà implémentées.

- `list_elem_t * create_element(int val)` : crée un nouveau maillon, dont le champ `next` vaut `NULL`, et dont le champ `value` vaut l'entier `val` passé en paramètre. La fonction retourne `NULL` en cas d'échec, sinon un pointeur sur le nouveau maillon.
- `void free_element(list_elem_t * l)` : libère la mémoire allouée à un maillon
- `int insert_head(list_elem_t * * l, int val)` ajoute un élément en tête de la liste (**`*l`** désignant cette tête de liste). A l'issu de l'exécution, `*l` pointe sur la nouvelle tête de liste. La fonction retourne 0 en cas de réussite, -1 en cas d'erreur.
- `int insert_tail(list_elem_t * * l, int val)` ajoute un élément en queue de la liste (**`*l`** désignant cette tête de liste). A l'issu de l'exécution, `*l` pointe sur la tête de liste. La fonction retourne 0 en cas de réussite, -1 en cas d'erreur.
- `int find_element(list_elem_t * l, int pos)` : retourne le maillon à la position pos dans la liste (le 1er élément est à la position 0) La fonction retourne un pointeur sur le maillon en cas de réussite, NULL en cas d'échec
- `int remove_element(list_elem_t** l, int val)` : supprime de la liste le premier élément de la liste de valeur val (**`*l`** désignant la tête de liste) et libère l'espace mémoire utilisé par le maillon ainsi supprimé. La fonction retourne
  - 0 en cas de réussite (un maillon de valeur `val` a été trouvé dans la liste),
  - -1 en cas d'échec (pas de maillon de valeur `val` trouvé dans la liste),
- `void reverse_list(list_elem_t** l)` : modifie la liste en *renversant* l'ordre de ses éléments (le premier devient le dernier, le second l'avant dernier, etc.).

  ☞ Attention à ne pas créer de « fuite mémoire »

Chacune de ces fonctions doit effectuer **un seul parcours** de la liste,
soit directement, soit par appel d'une autre fonction.

2. Module de test des listes chaînées

Vous disposez également d'un programme de test `test_list`, situé dans le sous-répertoire `./bin` . Pour le lancer, il vous suffit d'écrire `./bin/test_list`.

Ce programme permet (grâce à des commandes clavier) d'ajouter/supprimer des éléments de la liste. Les commandes sont résumées ci-dessous :

- `'t'` : ajout en tête de liste d'une chaîne saisie au clavier
- `'q'` : ajout en queue de liste d'une chaîne saisie au clavier
- `'f'` : recherche (et affichage) de l'élément n°i de la liste
- `'s'` : suppression d'un élément de valeur donnée
- `'r'` : inversion de la liste
- `'x'` : fin du programme

Le contenu de la liste est affiché à l'issue de chaque opération sous la forme :

```
[valeur_1]->[valeur_2]->…->[valeur_n]
```

Par ailleurs le programme signale les éventuelles fuites mémoire causées par les mises en œuvre des fonctions.

2. Utilisation du débogueur DDD

Si on souhaite mettre au point un programme à l'aide d'un « débogueur », il faut le compiler (ainsi que tous ses modules) avec l'option **`-g`** comme indiqué ci-dessous

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

Pour éviter d'avoir à modifier à chaque fois toutes les règles du makefile ; on peut définir des variables, qui permettent plus de généricité. Par exemple, ajouter au début du makefile la ligne

```
CFLAGS = -Wall -g
```

permet de définir une variable **`CFLAGS`** à laquelle on pourra ensuite faire référence, en écrivant **`$(CFLAGS)`**, dans les définitions de règles du makefile, comme dans l'exemple ci-dessous :

```
gcc $(CFLAGS) -c module1.c
```

1. Modifiez le `makefile` de manière à ce que les appels à **`gcc`** se fassent avec l'option **-g** (ou pas) suivant la valeur affectée à **`CFLAGS`** en début de `makefile`.
2. Relancez une compilation, expliquez pourquoi le `makefile` ne refait pas une mise à jour des modules objets et du programme exécutable ?
3. Comment résoudre le problème ?

Le débogueur **DDD** est une interface graphique qui facilite l'utilisation du débogueur **gdb** de la chaîne **gcc**., il se lance par la commande **ddd** ou **ddd test** (si **test** est le programme a tester).

Vous trouverez un manuel d'utilisation à l'adresse suivante :

http://hiko-seijuro.developpez.com/articles/ddd

3. Réalisation

3.1. Travail à effectuer

1. Complétez le fichier `list.c` qui fournit la mise en œuvre des listes chaînées (on rappelle en particulier qu'il faut prendre le plus grand soin dans la gestion des pointeurs, par exemple toujours s'assurer que la variable `list_elem_t* l` est différente de `NULL` avant d'écrire par exemple `l->next`).

   > Rappel : Chacune de ces fonctions doit effectuer **un seul parcours de la liste**, soit directement, soit par appel d'une autre fonction.

2. Vérifiez le bon fonctionnement de votre programme en ajoutant au **`makefile`** une cible **`tester`** qui teste de manière non-interactive (vous utiliserez pour cela les redirections d'entrées-sorties) le programme **`test_list`** en utilisant un fichier de jeu de tests `commandes.txt` que vous compléterez.

3.2. Conseils

1. Lisez bien les spécifications des fonctions (détaillées dans le fichier `list.c`) avant de vous jeter la tête la première dans le codage.
2. Assurez vous que vos sources compilent sans provoquer le moindre message de warning (rappelez vous que les compilateurs C sont très permissifs). Ce qui est considéré comme un warning en C provoquerait vraisemblablement une erreur de compilation en JAVA
3. Lorsque l'on manipule un pointeur sur une structure, on accède à ses champs non pas par l'opérateur '`.`' mais par l'opérateur '`->`'.