Université Gustave Eiffel — INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

L3 Info

TD Prog.C n°4

Traceur d'allocation
part.2 & 3 : modularisation

Ce TP constitue la 2° partie du projet (`libmtrack.so`).

2°) Modularisation

Dans cette seconde étape, il va s'agir de sortir tout ce que l'on peut du programme en cours (`track_02.c`) et transférer tout ce code dans un module à part.

Ensuite (version 4 et suivantes) le programme principal (celui à analyser) ne devra plus contenir aucune référence au traceur : ce sera au compilateur de faire le «branchement»

▸ **Exercice 1. (création du module `mtrack_03.[h|c]`)→ `track_03.c`**

La version 3 du programme principal (`track_03.c`) devrait alors se réduire à la fonction principale – exactement la même que dans la version 2 – plus un appel au `header mtrack_03.h`.

Ce `header` doit se réduire au *strict minimum* : 3 lignes + `«inclusion guard»`

→ ça devrait prendre 10 mn. tout au plus ...

La compilation devra bien sûr être adaptée en conséquence (cf. CM sur la modularisation). Elle peut toujours être effectuée en 1 seule passe même si le recours à un fichier `Makefile` devrait considérablement aider..

▸ **Exercice 2. (retour à la version initiale)→ `track_04.c` identique à `track_00.c`**

A partir de cette version 4, le programme principal `track_04.c` doit retrouver exactement la forme qu'il avait au départ `track_00.c` .

On ne doit plus y trouver aucun référence au traceur : les appels aux fonctions d'allocation / libération revienne sur les originaux `malloc/free`

☞ c'est la chaîne de compilation qui va devoir se charger de «brancher» le traceur (dont le code est dans le module `mtrack_04.[h|c|o]`)

- le corps du module `mtrack_04.c` est strictement identique `mtrack_03.c` : simple copie.
- le header du module `mtrack_04.h` est presque identique à la version précédente : il a juste besoin de 2 macros pour « masquer» les clônes dans les programmes appelant (en l'occurence `track_00.c`) :

  ```
  #define malloc(size) _mon_clone_de_malloc(size)
  #define free(ptr)    _mon_clone_de_free(ptr)
  ```

  ☞ c'est donc le préprocesseur qui remplacera `malloc` et `free` par leur clône dans la source à analyser.

☞ attention : à partir de cette version, il devient interdit d'inclure `mtrack_0*.h` dans `mtrack_0*.c`. Dans les versions précédentes ça ne servait déjà à rien, mais maintenant ce serait une catastrophe à cause des macros qui viendraient «clôner les clônes».

- chaîne de compilation : puisque le programme principal `track_04.c` ne doit plus faire référence au traceur, l'appel à `mtrack_04.h` doit être remis à la charge du compilateur
  ☞ cf. option(s) `-include path/header.h` ou `-Ipath -include header.h`

☞ attention : pour les mêmes raisons que précédemment, la compilation en 1 seule passe n'est plus possible. Il faut nécessairement la faire en 3 passes, avec des options différentes pour le module et pour le programme principal.

ⓐ pour le traceur :

```
xterm
$>gcc -c mtrack_04.c -o mtrack_04.o
```

ⓑ pour la source à traiter :

```
xterm
$>gcc -include mtrack_04.h -c main.c -o main.o
```

☞ étape ③(édition de liens) : pas de changement

```
xterm
$>gcc mtrack_04.o main.o -o main_04
```

3°) Informations contextuelles

Dans cette troisième étape, un peu plus technique, il va s'agir de faire en sorte que le traceur engrange aussi quelques informations contextuelles sur le code analysé.

En particulier, il sera fort pratique par la suite (analyse de programme plus gros, modulaires...) de connaître, pour chaque appel tracé, à quelle ligne de quel fichier / fonction à eut lieu cet appel. Il ne faut oublier le propos initial : c'est un outil de déboguage de code qui doit pouvoir traiter un "projet" modulaire.

Par exemple, la version 5 de notre traceur, toujours sur le même code source (`track_05.c = track_00.c`) devra produire quelque chose comme (format à votre convenance) :

```
xterm
$>./track_05
-activate tracker - [VERS.5]-
-----------------
in file <src/track_05.c> function <main> line <015> - (call#01) - malloc(1)->0x215aab0
in file <src/track_05.c> function <main> line <016> - (call#02) - malloc(1)->0x215aad0
in file <src/track_05.c> function <main> line <017> - (call#01) - free(0x215aab0)
in file <src/track_05.c> function <main> line <019> - (call#02) - free(0x215aad1) - ERROR : illlegal address -> ignored
.....
.....
$>
```

Quelques étapes intermédiaires sont nécessaires pour en arriver là.

«predefined macros» (__FILE__ ,__func__ & __LINE__)

Tous les standards C/C++ connaissent un certain nombre de «macros» prédéfinies, de la forme `__MACRO__` (double underscore) que l'on a rarement l'occasion d'utiliser en pratique mais que l'on trouve assez courrament dans les sources de la `libC`.

Les deux qui nous intéressent ici sont :

- `__FILE__` : donne le nom du fichier (source !) courant, sous la forme d'une chaîne de caractères
- `__LINE__` : donne le numéro de ligne où apparait la macro sous la forme d'un entier

La 3° macro utile est :

- `__func__` : donne le nom de la fonction courante, sous la forme d'une chaîne de caractères.

Elle est un peu différente : elle ne fait pas partie des «predefined macros» du standard mais est définie dans <stddef.h> mais le principe est exactement le même

src/predefmacros.c

```
01| #include <stdio.h>
02|
03| int main(void)
04| {
05|   fprintf(stdout,"file : %s\n,__FILE__"
06|   fprintf(stdout,"line : %d\n,__LINE__"
07|   fprintf(stdout,"func : %s\n,__func__"
08|   return 0;
09| }
```

xterm

```
$> ./predefmacros
file : src/predefmacros.c
line : 6
func : main
$>
```

Avant de pouvoir mettre ça en œuvre sur le traceur, il faut quelques étapes explicatives....

Même si c'est très simple à écrire, les principes en action ne sont pas évidents puisqu'ils exploitent certaines propriétés du langage (macros) vis-à-vis de la chaîne de compilation (pré-processing) :

- on veut les informations contextuelles (fichier, fonction, ligne) sur le code à analyser (pas sur les lignes du traceur)

☞ les appels aux «predefined macros» doivent donc apparaître dans le code source (pour avoir la bonne fonction, à la bonne ligne du bon fichier), mais les informations recueillies (ce fichier, cette ligne, cette fonction) doivent être traitées et exploitées dans le traceur (table d'enregistrement)

☞ il faut donc que notre traceur aille écrire cette requête dans le code source , la récupère, et l'intègre dans la cellule correspondante du traceur....

☞ mais on ne veut pas avoir à modifier le code source manuellement.

Donc là encore, il faut déléguer le travail au préprocesseur en jouant sur la définition et l'usage des macros.

puisque les fonctions «clônes» sont maintenant cachées par des macros permettant de les assimiler aux fonctions originales, rien ne nous empêche de modifier leur prototype.

☞ par exemple, pour le clône de malloc :

clone de malloc dans <memtrack_05.h>

```
void* _mon_clone_de_malloc(char* fich, const char* fonc, int line, size_t size);
#define malloc(size) _mon_clone_de_malloc(__FILE__,__func__,__LINE__, size)
```

Appliqué sur un programme test, l'étape ① de préprocessing donnerait :

xterm

```
$> cpp -include ./memtrack_05.h truc.c -o truc_pp.c
```

truc.c

```
01| #include <stdlib.h>
02|
03| int main(void)
04| {
05|     void *ptr = malloc(1);
06|     /* ... suite ... */
```

cpp ⟶

truc_pp.c

```
01| void* malloc(size_t size); /* importé de <stdlib.h>, mais inutile... */
02|
03| /* importé de <memtrack_05.h> par cpp */
04| void* _mon_clone_de_malloc(char* , const char* , int , size_t );
05|
06| int main(void)
07| {
08|     void *ptr = _mon_clone_de_malloc("truc.c","main",8, 1);
09|     /* ... suite ... */
```

☞ Ce sont les bonnes informations de contexte qui sont transmises au traceur par le clône.

Le principe serait exactement le même pour toutes les fonctions à clôner.

▸ **Exercice 3. (informations contextuelles)→** `memtrack_05.{h,c}`

Pour cette avant dernière étape, il faut intégrer les mécanismes décrits ci-dessus dans votre traceur.

Les informations contextuelles récoltées par les clônes dans le code à analyser (qui pourra être modulaire, ne l'oublions pas) devront être intégrées dans les "cellules" de traçage.

Ainsi, pour tout bloc mémoire ..., on devra accèder aux informations de contexte (fichier, ligne, fonction) d'allocation et de libération.

Remarque 1 : toutes les chaînes de caractères récoltées via les macros `__FILE__` et `__func__` sont définies "en dur" quelque part dans la mémoire et existent tant que le processus est actif : chaque appel à `__FILE__` crée donc une nouvelle chaîne automatiquement (ce sont donc, paradoxalement, des «macros variables»).

☞ de simple pointeurs suffisent pour y accéder : il n'y a pas de mémoire supplémentaire à créer/libérer, ni de chaînes à copier.

Remarque 2 : outre les informations qu'elles apportent au développeur (lorsqu'il s'agit de tracer une erreur, par exemple), la récolte et le stockage de ces *contextes* peuvent aider à détecter des défauts de conception sur des programmes modulaires un peu complexe.

☞ par exemple, un bloc mémoire alloué (appel à `malloc`) dans un module et libéré (appel à `free`) dans un autre est peut-être révélateur d'une faiblesse de conception.

▸ **Exercice 4. (Version finale : passage du traceur en lib. partagée)→** `libmtrack.{h,so}`

La toute dernière étape de ce «projet» consiste à transformer notre traceur en *bibliothèque partagée* et à mettre en place quelques mécanismes permettant de l'activer simplement sur n'importe quel code.

Là encore, l'essentiel se passe au niveau des directives de compilation. Tout est décrit dans les divers documents de cours à votre disposition.

Pour ce qui est de l'accès à cette bibliothèque, c'est à dire faire en sorte qu'elle soit facilement «branchable» sur n'importe quel programme que l'on souhaite analyser, cela passera également par le définition de quelques variables d'environnement système.

Ceci sera vu ultérieurement.