Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

L3 Info

TD Prog.C n°6

Traceur d'allocation
partie 3 : informations contextuelles & finalisation

Ce TP constitue la 3° et dernière partie du mini-projet «Traceur».
A ce stade, le module `mtrack[h,o]` *permet de tracer et analyser la gestion mémoire d'un code cible en se greffant sur celui-ci à la compilation.*
Il ne manque que quelques éléments sur le "contexte" d'appel des fonctions à tracer pour que ses fonctionalités soient complètes (par rapport au cahier de charges initial).
Une fois cette étape réalisée, le code produit pourra être considéré comme finalisé et passer en «déploiement», c'est-à-dire ici prendre la forme d'une bibliothèque partagée correctement installée et facilement utilisable.

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 a eu 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 courramment 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 1. (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 2. (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. Celà est à faire à la toute fin, lorsque le traceur est totalement finalisé.

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.

Le plus simple est de placer le bibliothèque dans des répertoires "pseudo-standard" :

- `$(HOME)/.local/include/mtrack` pour le 'header' `mtrack.h`
- `$(HOME)/.local/lib/mtrack` pour la lib. `libmtrack.so`
- dans le fichier de configuration `$(HOME)/.bashrc`, ajouter les lignes d'instruction suivantes :

.bashrc
```
export LOCINCPATH=$(HOME)/.local/include
export LOCLIBPATH=$(HOME)/igm/.local/lib
# variables d'environnement pour Makefile
export incMTRACK='-include $(LOCINCPATH)/mtrack/mtrack.h'
export libMTRACK='-L$(LOCLIBPATH)/mtrack -lmtrack'
# export path for 'ld' (directories linker)
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$LOCLIBPATH/mtrack
```

  ☞ penser à "actualiser" les modifications : `$> bash` dans la fenêtre de compilation en cours.

- dans le `Makefile` de compilation des fichiers à tester ajuster l'appel à la lib par quelque chose comme :

Makefile
```
ifeq ($(MT),1)
  PFLAGS += $(incMTRACK)
  LFLAGS += $(libMTRACK)
endif
```

☞ il suffira alors de compiler vos test avec `$> make test MT=1` pour que le traceur soit activé.

désormais, ces quelques lignes ajoutées à vos `Makefile` suffiront à activer le traceur sur n'importe quel code.