Langage C
TP N°8

Itheri Yahiaoui & Stéphane Cormier

Exercice 1 : « make et makefile »

Soit les code C suivants :

```
// fichier fcts.h
/* choix_menu */
int choix_menu() ;
/* lire_data */
void lire_data(char txt[]) ;
/* print_data */
void print_data(char txt[]) ;
/* lire_dec retourne un entier entre 0 et 4 */
int lire_dec() ;
```

```
// fichier fcts.c
#include "fcts.h"
#include <stdio.h>
#include <string.h>
int choix_menu() {
    int ok ;
    int choix ;
    ok = 0 ;
    while (! ok) {
     printf("Choix (1 ou 2):\n")
;
     scanf("%d", &choix) ;
     ok=((choix == 1)||(choix ==
2));
    }
    return choix ; }
```

```
void lire_data (char txt[]) {
    printf("Saisir une chaine de 5 caractères
    exactement :\n") ;
    scanf("%s", txt) ;
}
void print_data(char txt[]) {
    printf("Voici le résultat : ") ;
    printf("%s\n", txt) ;
}
int lire_dec() {
    int dec ;
    printf("Donner un entier :\n") ;
    scanf("%d", &dec) ;
    dec = dec % 4 ;
    return dec ;
}
```

```
/* fichier fct1.h */
/* on decale les 5 premieres
lettres et si ce n'est pas des
lettres on remplace par ?
txt contient au moins 5
caracteres */

void fct1(char txt[]) ;
```

```
/* fichier fct1.c */
#include "fct1.h"
void fct1(char txt[]) {
 int i ;
 for (i=0 ; i<5 ; i++) {
 if ((txt[i] >= 'a')&&(txt[i] <= 'z')) {
   txt[i] = 'a' + ((txt[i] - 'a') + 11) % 26 ;
 }
 else if ((txt[i] >= 'A')&&(txt[i] <= 'Z'))
{
   txt[i] = 'A' + ((txt[i] - 'A') + 11) % 26 ;
 }
 else {
       txt[i] = '?' ;
 }
 }
```

```
/* fichier fct2.h */
/* permutation des 5 premiers
caracteres de txt
txt contient au moins 5
caracteres */
void fct2(char txt[]) ;




}
```

```
/* fichier fct2.c */
#include "fct2.h"
#include "fcts.h"
void fct2(char txt[]) {
char c ;
int dec ;
int i, j ;
   dec = lire_dec() ;
   c = txt[0] ;
   j = 0 ;
   for (i=0 ; i<3 ; i++) {
       txt[j] = txt[(j+dec) % 4] ;
       j = (j+dec) % 4 ;
       }
   txt[j] = c ;}
```

```c
/* fichier main.c */
#include "fct1.h"
#include "fcts.h"
int main() {
  int choix ;
  char texte[6] ;
  choix = choix_menu() ;
  lire_data(texte) ;
    switch (choix) {
      case 1 :
            fct1(texte) ;
            break ;
      case 2 :
            fct2(texte) ;
            break ;
      default :
            break ;
      }
      print_data(texte) ;
return 0 ;
}
```

Compiler séparément tous les fichiers :

- fcts.c
- fct1.c
- fct2.c
- main.c

Normalement, les trois premières compilations se passent bien, mais la dernière produit un message d'erreur. Quelle est l'erreur générée et pourquoi ?

Modifier main.c pour permettre la compilation. Modifier également les fichiers.h pour éviter les inclusions multiples et les redéclarations.

Générer les fichiers .o correspondants aux fichiers précédents et utiliser gcc pour créer le a.out qu'on nommera ***principal.***

Soit le fichier Makefile suivant :

```
principal : main.o fct1.o fct2.o fcts.o
	@echo creation  :
	gcc -o principal main.o fct1.o fct2.o fcts.o
	@echo ------------------------------

main.o : main.c fct1.h fct2.h fcts.h
	@echo creation de main.o :
	gcc main.c
	@echo ------------------------------

fct1.o : fct1.c fct1.h
	@echo creation de fct1.o :
	gcc fct1.c
	@echo ------------------------------

fct2.o : fct2.c fct2.h fcts.h
	@echo creation de fct2.o :
	gcc fct2.c
	@echo ------------------------------

fcts.o: fcts.c fcts.h
	@echo creation de fcts.o :
	gcc fcts.c
	@echo ------------------------------
```

- Lancer le programme make. Cela fonctionne-t-il sinon faire les modifications nécessaires.
- Renommer votre makefile en principal0.mak, et lancer make en indiquant le fichier correspondant avec l'option « -f ».
- En suivant les différentes étapes de 1 à 9 du lien « https://gl.developpez.com/tutoriel/outil/makefile/ » créer différents fichiers makefile nommés :
    - principal1.mak,
    - principal2.mak,
    - …,
    - principal8.mak
    - principal9.mak.
- Pour chaque makefile créé, lancer « make » et assurer vous de la bonne compilation et de la création de votre exécutable.

Rmq : Attention à la syntaxe précise du makefile (utilisation des tabulation espace et \)

Exercice 2 : « Utilisation de gdb »

Écrire l'exemple C ci-dessous qui contient des erreurs. Il devrait calculer et afficher (n!)

```
# include <stdio.h>
int main() {
          int i, num, j;

          printf("Entrez le nombre n :");

          scanf("%d", &num);

          for (i=1; i<num; i++)
                   j=j*i;
          printf("La factorielle de d est %d \n", num, j);
}
```

1. **Compiler le programme (cc factoriel.c)**
Tester avec n = 3. Vous n'obtenez pas bien-sûr le bon résultat car l'algorithme est faux (ce qui est normal).

2. **Lancer gdb**

   a) Pour utiliser gdb, compilez le programme avec l'option de débogage –g : cc -g factoriel.c
   b) Lancez gdb avec : gdb a.out (si vous n'avez pas indiqué un fichier de sortie sinon gdb nom_programme)

3. **Configurer un point d'arrêt dans le programme C**
   Syntaxe : break line_number
   Autres formats:
   - break [nom_fichier]: numéro_ligne
   - break [nom_fichier]: nom_fonction

Cela permet de placer des points d'arrêt dans le programme C, où l'on suspecte des erreurs. Lors de l'exécution du programme, le débogueur s'arrête au point d'arrêt et invite à procéder au débogage.

Placer un point d'arrêt dans le programme, lors de l'exécution du programme le débuggeur s'arrête au niveau du point d'arrêt et donne le prompt pour pouvoir déboguer.

break 10 par exemple

Regarder le résultat obtenu.

4. **Exécuter le programme C dans le débogueur gdb**
   Syntaxe : run [args] args permet de placer des arguments de ligne de commande.

À partir de gdb, la commande run et le programme s'exécute jusqu'au premier breakpoint et donne l'invite de commande pour le débogage. Vérifier l'affichage fourni (gdb affiche la ligne correspondante au breakpoint). On peut ensuite utiliser les différentes commandes de gdb pour déboguer le programme.

5. **Affichage des valeurs de variables dans le débogueur gdb**
   Syntaxe : print {variables}

Exemple :

```
print i
print j
print num
```

Tester l'affichage de ces valeurs dans gdb.

Dans le programme précédent, la variable j n'est pas initialisée d'où des valeurs qui n'ont pas de sens.

Résoudre ce problème en initialisant la variable j à 1, compiler le programme et l'exécuter.
Même après ce correctif, le programme ne fonctionne toujours pas (valeur erronée).

Donc, placer un break en ligne 10.

6. Autres commandes de gdb

Il existe trois types d'opérations dans gdb quand un programme s'arrête à un point d'arrêt.

Ils permettent de continuer jusqu'au prochain point d'arrêt, ensuite en pas à pas, ou en poursuivant les prochaines lignes du programme :

- c ou continue : le débogueur continuera à s'exécuter jusqu'au prochain point d'arrêt.
- n ou next: le débogueur exécutera la ligne suivante en une seule instruction.
- s ou step: identique à next, mais ne traite pas une fonction comme une instruction unique, mais entre dans la fonction et l'exécute ligne par ligne.

Vérifier le programme de factorielle avec le continue par exemple. Quel est le problème rencontré ? Le corriger.

Raccourcis de commande gdb

Les raccourcis suivants pour la plupart des opérations fréquentes de gdb :

- l – list
- p - print
- c - continue
- s – step
- ENTER: appuyer sur la touche Entrée pour exécuter à nouveau la commande précédemment exécutée.

Autres commandes de gdb

- l **commande** : Utiliser la commande gdb l ou list pour imprimer le code source en mode débogage. Utiliser l numéro de ligne pour afficher un numéro de ligne spécifique (ou) l fonction pour afficher une fonction spécifique.
- **bt : backtrack** – Affiche la pile d'exécution.
- **help** : Affiche l'aide sur un sujet particulier de gdb - help TOPICNAME.
- **quit** : Quitter le débogueur gdb.

Remarque 1 : on peut également introduire des « watchpoint », à l'aide de la commande watch, qui permet d'interrompre l'exécution lorsque la valeur d'une variable est modifiée: on « surveille » la variable.

Remarque 2 :

Résumé rapide

quit (q)	quitter gdb
run (r)	lancer l'exécution
break,watch,clear,delete (b, v, cl, d)	introduire un point d'arrêt, ou "surveiller" une variable
step, next, continue (s, n ,c)	avancer d'un pas (en entrant ou pas dans les sous-fonctions, relancer jusqu'au prochain point d'arrêt
print, backtrace, list (p, bt, l)	afficher la valeur d'une variable, la pile d'exécution, afficher l'endroit où l'on se trouve dans le code

Exercice 3 : « gdb avec des arguments de ligne de commande »

```c
#include <stdio.h>
#include <stdlib.h>

int conversion(char *s) {
   int somme, i;
   /* Algorithme de Horner */
   somme = 0;
   i = 0;
   while (s[i] != '\0') {
     somme = 10 * somme + s[i] - '0' ;
     i++;
   }
   return somme;
}
```

```c
int main(int argc, char *argv[]) {

  int i;
  int valeur, somme=0;

  for (i=1; i<=argc; i++) {
     valeur = conversion(argv[i]);
     somme = somme + valeur;
     printf("Argv %d, valeur %d\n", i,
valeur);
  }
  printf("Somme : %d\n", somme);
  return 0;
}
```

Le programme affiche à l'écran la valeur (entière) de chacun des arguments passés par la ligne de commande ainsi que la somme de ces valeurs. Compiler le programme avec l'option de débogage.
Exécuter le programme.
On obtient une erreur.

Lancer gdb avec le fichier de sortie précédemment créé.
On se retrouve dans l'interface de gdb. Exécuter run 19 54 (par exemple).
Le programme s'interrompt lors de l'erreur de segmentation. On retourne alors dans l'interface de gdb avec différentes informations :
— l'indication d'une erreur de segmentation;
— la fonction où s'est produite et la valeur de ses arguments ;
— la ligne en cours d'exécution, ayant provoqué l'erreur

Quelle était la fonction en cours d'exécution au moment de l'erreur ? Quelle était la valeur de son argument ?
Pourquoi la ligne en cours d'exécution a-t-elle provoqué une erreur de segmentation ?

gdb permet de remonter dans les appels de fonctions en cascade. La commande est up.

gdb indique le numéro de la ligne de la fonction main d'appel de la fonction en cours d'exécution.

Afficher
- ***la valeur problématique avec print argv[i]***
- ***la valeur de l'indice de boucle : print i***
- ***le détail des arguments transmis au programme :***
    - ***print argc***
    - ***print arg[…]***

Exercice 4 : « Utilisation de gprof »

Le profilage permet de déterminer les portions chronophages du code d'un programme et qui doivent donc être réécrites pour une accélération de l'exécution. Le profilage permet de gagner du temps sur la vitesse d'exécution.

L'outil de profilage GNU «gprof» est utilisable avec quelques restrictions :
- Avoir le profilage activé lors de la compilation
- Exécuter le programme pour obtenir des données de profilage
- Activer gprof sur ces données

Voici le code que nous allons utiliser :

```
//Programme principal
// main.c
#include <stdio.h>

void sous_f1(void);
void sous_sous_f1(void);
void sous_sous_sous_f1(void);

void f1(void)
{
     printf("\n Fonction f1 \n");
     int i = 0;
     for(;i<0xffffffff;i++);  // permet de
prendre un peu de temps machine
     sous_f1();  //appel  de  la  fonction
sous_f1
     return;
}

static void f2(void)
{
     printf("\n Fonction f2 \n");
     int i = 0;
     for(;i<0xffffffaa;i++);
     return;
}

int main(void)
{
     printf("\n Programme principal \n");
     int i = 0;

     for(;i<0xffffff;i++);
     f1(); // appel de f1
     f2();  // appel de f2

     return 0; }
```

```
// Fichier de fonctions
// fonctions.c


#include<stdio.h>

void sous_sous_sous_f1(void)
{
     printf("\n    Fonction    sous_sous_sous_f1
\n");
     int i = 0;
     for(;i<0xffffffcc;i++);     /*  permet  de
prendre un peu de temps machine*/
     return;
}

void sous_sous_f1(void)
{
     printf("\n Fonction sous_sous_f1 \n");
     int i = 0;
     for(;i<0xffffffdd;i++);     /*  permet  de
prendre un peu de temps machine*/
     sous_sous_sous_f1();
     return;
}

void sous_f1(void)
{
     printf("\n Fonction sous_f1()\n");
     int i = 0;
     for(;i<0xffffffee;i++);
     sous_sous_f1() ;
     return;
}
```

1. Activer le profilage

On active le profilage avec l'option '-pg' lors de la compilation : Consulter le manuel de gcc pour cette option.
Compiler avec les options suivantes :
-Wall pour activer tous les warnings (consulter le manuel de gcc pour cette option)
-pg pour le profilage

```
gcc -Wall -pg main.c fonctions.c -o test
```

2. Exécution du code

Tester le fichier binaire généré. Il va afficher les différents printf prévus dans les fonctions et sous-fonctions. Vérifier ensuite les fichiers présents dans le répertoire d'exécution.
Rmq : Est apparu un fichier à l'extension .out qui contient les informations utiles au profilage.

3. Exécution de gprof

Lancer gprof ainsi :
gprof nom_exececutable nom_fichier_genere_a_etape_2 > analyse.txt

Comprendre les informations de profilage :

Toutes les informations de profilage sont présentes dans le fichier « analyse.txt ».
Ouvrir ce fichier texte.
Ce fichier est découpé en 2 parties :

- Un profil (flat profile)
- Un graphe d'appel (call graph)

Toutes les explications sont fournies dans chacune de ces sections et clairement détaillées dans le fichier.

Personnaliser la sortie de gprof en utilisant des flags particuliers

a) Suppression de l'affichage de fonctions déclarées statiquement (privées) à l'aide de –a

Reprendre votre fichier de sortie .out précédent et tester avec gprof et l'option –a
gprof - a nom_exececutable nom_fichier_genere_a_etape_2 > analyse2.txt
Consulter le fichier analyse2.txt. Que constate-t-on ?

b) Suppression de l'affichage détaillé à l'aide de –b

Tester avec cette option. Comparer avec analyse.txt
gprof - a nom_exececutable nom_fichier_genere_a_etape_2 > analyse2.txt

c) Afficher que le profil à l'aide de –p

Tester avec cette option. Comparer avec analyse.txt

d) Afficher les informations liées à une fonction spécifique dans un profil

gprof -nom_fonction -b nom_exececutable nom_fichier_genere_a_etape_2 > analyse3.txt

e) Suppression du profil en sortie en utilisant –P

f) En utilisant –Pnom_fonction

Exclut la fonction nom_fonction de la sortie

g) Afficher que le graphe d'appel à l'aide de –q et que les information d'une fonction avec - qNom_Fct

Tester l'option –q et visualiser le résultat en sortie.