Programmation C
TP n° 2 : gdb, struct

I don't like debuggers. Never have, probably never will. I use gdb all the time, but I tend to use it not as a debugger, but as a disassembler on steroids that you can program.

— *Linus Torvalds (2000)*

Le debugger `gdb` offre de très nombreuses fonctionnalités facilitant la détection d'erreurs dans les programmes, notamment leur exécution pas-à-pas, l'introduction de points d'arrêts, l'observation de valeurs de variables, etc. Les exercices de cette séance vous permettront de découvrir les plus élémentaires [1].

Exercice 1 : Prise en main de `gdb`

1. Dans un fichier `exo1.c`, recopier le code ci-dessous avec la même mise en page. Sa boucle `for` accumule dans deux variables `sum` et `prd`, respectivement la somme et le produit des entiers de `1` à `(SUP - 1)`, où `SUP` est une constante définie en début de programme.

```c
#define SUP 10
int main(){
    int i, sum = 0, prd = 1;
    for (i = 1; i < SUP; i++){
         sum += i;
         prd *= i;
    }
    return 0;
}
```

2. Compilez ce code. Dans le même répertoire, lancez `gdb` puis entrez la commande `file exo1` chargeant en mémoire l'exécutable produit. La commande `list` (en abrégé `l`) suivie d'un numéro de ligne affiche le code source de votre exécutable autour de cette ligne (les 5 précédentes, la ligne, les 5 suivantes).

3. La commande `run` (en abrégé `r`) exécute le programme jusqu'à sa terminaison (ou jusqu'à la rencontre d'un point d'arrêt, voir plus bas). Vous pouvez l'utiliser à tout moment pour relancer le programme.

4. L'ajout d'un point d'arrêt se fait à l'aide de la commande `break` (en abrégé `b`) suivie d'un numéro de ligne. Entrez cette commande suivie du numéro de ligne de l'instruction `sum += i;` (en principe, 5). Relancez le programme : l'exécution devrait être suspendue à cette ligne, immédiatement avant l'exécution de son instruction (`Breakpoint 1, main () at exo1.c:5`).

5. L'affichage de la valeur courante d'une expression se fait à l'aide la commande `print` (en abrégé `p`). Entrez la commande `print i` (affichant en principe la valeur `1`) puis `print sum` (affichant `0` – la valeur de `sum` n'a pas encore été modifiée).

6. La commande `next` (en abrégé `n`) permet d'exécuter l'instruction suivante du programme. Entrez deux fois `next` (ou `n`) : l'exécution devrait revenir à la boucle de la ligne 4. Entrez une nouvelle fois `next`. Réaffichez les valeurs de `i`, `sum` et `prd`.

1. La commande `help` de `gdb` pourra également vous être utile.

7. La commande `continue` (en abrégé `c`) reprend l'exécution du programme à partir de l'instruction courante, jusqu'à la rencontre d'un point d'arrêt. Entrez cette commande.

8. On peut ajouter un nombre quelconque de points d'arrêts. Ajoutez-en un à la ligne 6, et observez l'effet de `continue` (`c`). La commande `info break` (en abrégé `i b`) affiche la liste des points d'arrêt courants, ainsi que le nombre de fois où chacun a été atteint.

9. La commande `delete` (en abrégé `d`) suivie d'un numéro de point d'arrêt permet de supprimer ce point d'arrêt (ou, sans argument, tous). Supprimer le premier point d'arrêt créé ci-dessus. Redemandez la liste des points d'arrêt, observez l'effet de `continue`.

10. Plutôt que d'afficher des valeurs d'expressions par `print`, on peut aussi demander l'affichage automatique de ces valeurs à chaque rencontre d'un point d'arrêt, à l'aide de la commande `display` (en abrégé `disp`). Entrez successivement les commandes suivantes : `display i`, `display sum`, et `display prd`, et observez l'effet de `continue`.

11. Ajoutez un point d'arrêt à la ligne du `return 0`. Replacez un point d'arrêt à la ligne 5, relancez le programme (`run` ou `r`) et observez l'exécution de votre programme (`continue` ou `c`) jusqu'à sa terminaison.

Notez que le contenu d'un tableau peut être observé sous `gdb` : il suffit de faire suivre `print` ou `display` du nom du tableau. Notez également que ces commandes n'acceptent que des expressions dont les variables ont déjà été déclarées. Vous pouvez donc, pour débugger un programme sous `gdb` : ajouter un point d'arrêt à la ligne du return final ; lancer le programme une première fois ; ajouter des points d'arrêts et les `display` nécessaires, relancer le programme, tenter de comprendre l'erreur, rajouter des points d'arrêts et des affichages si nécessaire, etc.

Exercice 2 : Fractions avec `struct`

Dans cet exercice, on se propose de définir un type avec `struct` afin de représenter les rationnels sous forme de fractions. Pensez à tester votre code à chaque question, par exemple en utilisant les fonctions d'affichage de `gdb`.

1. À l'aide de `struct`, définissez un type de structure `frac` avec deux champs entiers `num` et `den` de type `long int` qui représentent respectivement le numérateur et le dénominateur de la fraction. Utilisez le mot clé `typedef` afin de créer l'alias `frac` pour le type `struct frac` et rendre ainsi votre code plus lisible.

2. Écrivez une fonction d'en-tête `struct frac build(long int n, long int d)` qui prend en arguments deux entiers `n` et `d` et qui retourne la fraction $\frac{n}{d}$.

La fonction définie à la question précédente à un défaut évident : elle permet de construire des fractions avec 0 comme dénominateur ! Afin de pallier ce problème, on se propose d'utiliser la fonction `assert` de la bibliothèque standard. Une commande de la forme `assert(b)` évalue la condition logique `b`. Si la condition est fausse, le programme interrompt son exécution en affichant un message d'erreur. Pour vous servir de cette fonction, il faut ajouter au début de votre programme la ligne suivante :

```
1       #include <assert.h>
```

3. Modifiez le code de la fonction `build` afin de provoquer une erreur si l'on essaye de construire une fraction dont le dénominateur est `0`. Ensuite, dans le `main`, créez un tableau `ex_fractions` de fractions qui contient les fractions $\frac{1}{1}$, $\frac{1}{2}$, $\frac{2}{4}$, $\frac{-9}{3}$, $\frac{8}{-20}$, $\frac{-5}{-1}$, $\frac{1}{-3}$.

4. Écrivez une fonction d'en-tête `int eq(frac f, frac g)` qui renvoie `1` si les deux fractions sont égales. On rappelle que deux fractions $\frac{a}{b}, \frac{c}{d}$ sont égales si et seulement si $a * d = c * b$.

5. Écrivez une fonction `int isInteger(frac f)` qui renvoie `1` si la fraction `f` est un entier (c'est-à-dire peut être mis sous la forme $\frac{n}{1}$ où n est un entier) et 0 sinon.

6. Écrivez les fonctions suivantes qui calculent la somme, la soustraction et la multiplication de fractions.

```
frac sum(frac f, frac g) // somme
frac sub(frac f, frac g) // soustraction
frac mul(frac f, frac g) // multiplication
```

7. Écrivez une fonction d'en-tête `frac reduce(frac f)` qui renvoie la fraction `f` sous forme irréductible. Pour cela, on pourra d'abord coder la fonction `long pgcd(long a, long b)` qui calcule le pgcd des deux entiers `a` et `b`. On rappelle que l'algorithme d'Euclide pour calculer le pgcd de deux entiers *positifs* a et b est (en pseudo-code) :

```
x <- a
y <- b
while (y !=0){
  r <- reste de la division euclidienne de x par y
  x <- y
  y <- r
}
return x
```

Vous devez également faire en sorte que lorsque la fraction renvoyée par `reduce` est négative, le signe apparaisse au numérateur et non pas au dénominateur. Testez vos fonctions sur les fractions de `ex_fractions`.

On souhaite maintenant définir un type qui représente un point (rationnel) dans le plan, c'est-à-dire un point dont les coordonnées peuvent être représentées par des fractions.

8. Définissez un type de structure `point` avec deux champs `x` et `y` de type `frac`, qui représentent les coordonnées du point, puis utilisez `typedef` afin de créer l'alias `point` pour le type `struct point`.

9. Écrivez une fonction `int eqp(point p1, point p2)` qui renvoie `1` si les deux points ont les mêmes coordonnées et `0` sinon.

10. Écrivez une fonction d'en-tête `double dist(point p1, point p2)` qui prend en argument deux points et calcule leur distance euclidienne en tant que valeur de type `double`.

On rappelle que pour utiliser la fonction `sqrt` de la bibliothèque standard, il faut rajouter `#include <math.h>` en haut de votre programme et il faut compiler avec l'option `-lm`. Pour cela, vous devez rajouter la ligne

```
LDLIBS=-lm
```

au début de votre ficher `Makefile`.

On rappelle également que pour convertir une variable `n` de type entier en double, il faut utiliser l'expression `(double)n`.