Corrigé TP9

[a] La sortie obtenue pour l'exécution de `./arguments` est :

```
corentin@gazelle:GDB $ ./arguments
Erreur de segmentation (core dumped)
```

[b] D'après la trace d'exécution de gdb :

```
(gdb) run 17 42
Starting program: /home/corentin/w/uga/INF203/poly/Semaine9/TP9/GDB/arguments 17 42
Argument 1, valeur 17
Argument 2, valeur 42

Program received signal SIGSEGV, Segmentation fault.
0x08048451 in convertir (s=0x0) at arguments.c:10
10 while (s[i] != '\0') {
```

On constate que l'erreur de segmentation est survenue dans la fonction `convertir` alors que son argument `char * s` valait `0x0`.

L'instruction `s[i]` de la ligne 10 correspond dans ce cas là à un accès à la ième case du tableau à l'adresse `s` qui vaut 0, c'est à dire NULL. On comprend que l'on a tenté d'accéder à un tableau qui n'existe pas !

[c] Grâce aux commandes `up` et `print`, on comprends que la situation est la suivante :

Nombre d'argument	2
argc	3
argv[0]	"./arguments"
argv[1]	"17"
argv[2]	"42"

On voit dans la trace de gdb que l'on tente d'accéder à `argv[3]`. On comprends alors que la boucle qui appelle la fonction `convertir` pour chaque argument est mal paramétrée : son index `i` va de 1 à 3 alors qu'il devrait aller de 1 à 2. Il faut donc corriger la condition de sortie de boucle du for comme cela :

```
for (i=1; i<argc; i++) {
```

[d] On exécute simplement (sans argument) le programme `erreur`. On obtient l'erreur suivante :

```
corentin@gazelle:GDB $ ./erreur
Exception en point flottant (core dumped)
```

(ou en anglais : `floating point exception (core dumped) ./erreur`)

En exécutant le programme avec gdb, on apprend que l'erreur a eu lieu dans le fichier `erreur.c` à la ligne 11 dans la fonction `main` qui n'avait pas d'argument. La mention : `Arithmetic exception` nous met sur la voie d'une erreur de calcul.

Avec la commande `print` (racourcit "p"), on constate qu'au moment de l'exception, la valeur de `i` était de 1. En regardant de plus près le fichier erreur.c, on constate alors les choses suivantes :

- L'indentation du fichier est douteuse
- Les variables q et r sont initialisée à l'aide d'une syntaxe à n'utiliser sous aucun prétexte

— La ligne 11 effectue une division par zéro si i vaut 1

On comprends donc qu'il faut corriger la condition de sortie du for pour que i aille de n à 1 au lieu de n à 0.

[e] La différence entre la commande `print` et la commande `display` est que cette dernière affiche la variable à chaque commande gdb entrée ultérieurement. C'est donc très pratique pour suivre l'évolution d'une variable au cours du programme.

[f] Explication des commandes `next` et `step` :

next Exécute l'instruction et passe à la suivante. Passe "au dessus" des appels de fonction en exécutant toutes les instructions qu'elles contiennent.

step Comme next, sauf si l'instruction est un appel de fonction. Dans ce cas, se place à la première instruction de celle-ci. On dit que gdb "descend" dans la fonction.

[g] La commande `cont`, pour "continue", reprend l'exécution normale du programme jusqu'au prochain breakpoint ou jusqu'à la fin du programme si aucun breakpoint n'est rencontré.

[h] Tous les nombres de type

$$2^n * 42$$

donnent

$$x == 42$$

au bout d'un certain nombre d'itérations. Par exemple : `42`, `84`, `168`.

[i] On déclare la variable représentant la ligne découpée ainsi :

```
char *ligne_courante_decoupee[TAILLE_MAX_LIGNE];
```

On reconnait la même syntaxe que pour `argv` que nous manipulons souvent : les crochets `[...]` signifient que nous avons à faire à un tableau, et le type `char *` nous indique qu'il s'agit d'un tableau de pointeurs vers un `char`, donc un tableau de chaînes de caractère.

[j] Une implémentation possible pour la fonction `decouper_ligne` :

```
void decouper_ligne(char *ligne, char *ligne_decoupee[]) {
   int i=0, nb_word = 0;
   int skip_sep = 0, skip_word = 0;

   while (ligne[i] != '\0') {
      if (ligne[i] == ' ' || ligne[i] == '\t') {
         skip_word = 0;
         if (skip_sep) {
            i++;
            continue;
         }

         ligne[i] = '\0';
         skip_sep = 1;
      } else {
         skip_sep = 0;
         if (skip_word) {
            i++;
            continue;
         }

         ligne_decoupee[nb_word] = &ligne[i];
         skip_word = 1;

         nb_word++;
      }
      i++;
   }

   ligne_decoupee[nb_word] = NULL;
}
```

[k] Il s'agit de rechercher une chaîne de caractères dans les fichiers .c et .h du répertoire courant. C'est bien évidemment la commande `grep` qui nous vient à l'esprit !

```
grep executer *.c *.h
```

La commande donne rapidement les fichiers et les numéros de ligne où apparait le mot "executer". On voit que les occurrences correspondent à la déclaration, l'implémentation et l'appel de la fonction `executer_ligne_decoupee`.