Les erreurs mémoires.

Les erreurs mémoires peuvent être de deux types:
- Une non libération de mémoire allouée dans le programme (appelée fuite mémoire).
- Un accès à une case mémoire non prévue (accès en lecture ou écriture).

Non libération:

Cause:
La non libération est généralement liée à la non utilisation de l'instruction *free* après avoir utilisé *malloc* (*realloc*, *strdup*, etc).

Conséquence:
Cette non utilisation de la mémoire n'impacte pas directement le fonctionnement du programme. La mémoire RAM est alors utilisée de plus en plus et peut, dans certains cas, être totalement utilisée empêchant alors l'utilisation du système d'exploitation.

Accès à une case mémoire non prévue:

Cause:
Un accès à une case mémoire non prévue peut prendre différente forme. Les plus courantes étant l'indexation d'un tableau au delà de sa taille (T[5] pour un tableau T de 5 cases par exemple), ainsi que le déférencement d'un pointeur dont l'adresse est invalide (accès à *pointeur, alors que pointeur contient NULL ou une adresse invalide).

Conséquence:
Les conséquences de ces accès ou écriture dans des zones de mémoires non prévues sont multiples. Une partie de ces erreurs sont totalement invisibles lors de l'exécution du programme ce qui les rend difficiles à détecter. Une autre partie de ces erreurs peut provoquer des comportements étranges du programme, non compatible avec l'attente d'un programme C (changement de valeur d'une variable imprévue) lors d'une écriture en dehors de la zone mémoire.
Enfin, si la zone mémoire touchée sort du cadre de la zone allouée pour le programme par le système d'exploitation le programme est stoppée brutalement indiquant une "erreur de segmentation", ou en anglais: Segmentation Fault.
Les conséquences sont généralement associées à des comportements indéterminés et peuvent varier en fonction des conditions: résultats différents suivant plusieurs lancements ou en changeant de machine.

Remarque: L'erreur de segmentation n'est que l'une manifestation possible d'une erreur mémoire.
L'absence d'une telle erreur n'indique donc pas qu'il n'y a pas d'erreur.
Sa présence par contre indique forcément une erreur mémoire.

Les erreurs mémoires sont à la fois difficile à identifier et à debuger sans l'aide d'un programme externe car l'erreur peut passer inaperçu dans le comportement du programme, celui-ci peut être altéré, et l'origine de l'erreur n'est pas indiquée.
Il existe pour cela deux programmes permettant de débuger les erreurs mémoires en indiquant l'origine de l'erreur: **gdb** et **valgrind**.

GDB

En cas d'erreur de segmentation, **gdb** permet d'indiquer la ligne à l'origine du problème et la trace d'execution associée.

Principe.

Soit un fichier exécutable du nom de [EXEC]. Le fichier exécutable doit impérativement avoir été compilé avec les options de debug (-g).

Lancer gdb (depuis le même répertoire que l'exécutable) avec
```
$ gdb ./[EXEC]
```

ou bien
```
$ gdb -tui ./[EXEC]
```
afin d'avoir en plus une interface visuelle.

Démarrez ensuite l'exécutable avec
```
(gdb) run
```

La ligne à l'origine de l'erreur mémoire s'affiche alors (voir exemple en bas d'annexe).

Si l'executable attend des **arguments** [ARGS] de la ligne de commande, la syntaxe est la suivante
```
(gdb) run [ARGS]
```

L'affichage de la **trace d'exécution** s'obtient avec la commande
```
(gdb) bt
```

Pour **quitter gdb**:
```
(gdb) quit
```

Valgrind

Valgrind est un programme permettant de vérifier si des zones mémoires sont mal utilisées ou si il y a des fuites mémoires. Il détecte d'avantages d'erreurs que **gdb** et indiquent tous les comportements étranges, même si ceux-ci n'aboutissent pas à d'erreur de segmentation.
Avant de rendre un programme, il est recommandé de toujours vérifier avec **Valgrind** si celui-ci ne détecte pas de comportement suspects.

Lancement de valgrind en ligne de commande (depuis le répertoire de l'executable):
`$ valgrind ./[EXEC]`
ou
`$ valgrind ./[EXEC] [ARGS]` *(dans le cas où il y a des arguments).*

Exemples de sorties:

Soit le programme incorrecte suivant:

```
int main()
{
    int *a;
    *a=8;

    return 0;
}
```

Cas de gdb:

```
(gdb) run
Starting program: [EXEC]

Program received signal SIGSEGV, Segmentation fault.
0x0000000000401af8 in main ()
    at [FICHIER]:LIGNE
LIGNE     *a=8;
```

Cas de valgrind:

```
$ valgrind ./[EXEC]

[...]
==5438== Invalid write of size 4 (indique écriture non permise à la ligne 41 du fichier main.c)
==5438==    at 0x401AF8: main (main.c:41)
==5438==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
[...]
==5438== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 2 from 2)
(La dernière ligne résume le nombre d'erreurs: il faut toujours atteindre 0 erreurs au moment de rendre un programme)
```