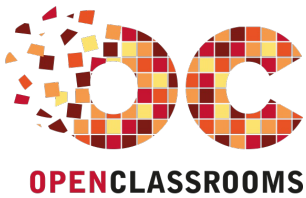
Débuguer facilement avec Valgrind

Par Tados



www.openclassrooms.com

Dernière mise à jour le 24/06/2011

Sommaire

Débuguer facilement avec Valgrind

Par  Tados

Mise à jour : 24/06/2011
Difficulté : Facile Durée d'étude : 30 minutes

Valgrind est un programme créé en 2000 par Julian Seward qui a depuis été rejoint par d'autres programmeurs. Il a été conçu principalement pour les programmes écrits en C et C++ et ne fonctionne pas sur Windows. Vous pouvez avoir la liste des plateformes supportées ici.

Valgrind possède plusieurs outils dont memcheck qui permet de :

- vérifier les accès en lecture et en écriture ;
- contrôler les fuites de mémoire ;
- vérifier que l'on n'utilise aucune variable non initialisée.

Tout ceci permet notamment de débusquer très facilement les fameuses erreurs de segmentation qui en ont terrorisés plus d'un. C'est alléchant n'est-ce pas ?

Valgrind permet aussi de faire du profilage de code (callgrind), du cache (cachegrind) et du tas (massif), ainsi que du débugage d'application multi-threadée (helgrind). Toutefois ce tutoriel ce concentre sur sa fonctionnalité première qui est de contrôler l'utilisation de la mémoire avec memcheck.
Sommaire du tutoriel :

Comment utiliser valgrind ?

Comme je l'ai dit en introduction, valgrind a été principalement conçu pour les programmes écrits en C ou en C++. C'est particulièrement le cas pour l'utilisation que l'on va en faire dans ce tutoriel. Je vais quant à moi utiliser le langage C.

Hello, world !

Prenons un premier programme juste pour comprendre comment on utilise valgrind :

Code : C

```c
#include<stdio.h>

int main(void)
{
    printf("Hello, world !\n");
    return 0;
```

```
}
```

Simple comme bonjour me direz-vous ! Et bien oui, mais le but ici n'est pas de faire compliqué.

Compilons-le :

Code : Console

```
gcc -o test0 test0.c
```

Voila, maintenant exécutons-le avec valgrind :

Code : Console

```
valgrind ./test0
```

Voila ce que j'obtiens chez moi :

Code : Console

```
==9029== Memcheck, a memory error detector
==9029== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==9029== Using Valgrind-3.5.0-Debian and LibVEX; rerun with -
h for copyright info
==9029== Command: ./test
==9029==
Hello, world !
==9029==
==9029== HEAP SUMMARY:
==9029==     in use at exit: 0 bytes in 0 blocks
==9029==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==9029==
==9029== All heap blocks were freed -- no leaks are possible
==9029==
==9029== For counts of detected and suppressed errors, rerun with: -v
==9029== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 12 from 7)
```

Ça, c'est exactement ce que vous voudrez obtenir quand vous testerez vos propres programmes.
Le numéro à gauche indique le numéro du processus, vous n'en aurez probablement pas besoin.
On remarque également la dernière ligne :

Code : Console

```
==9029== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 12 from 7)
```

Cette ligne est un résumé du nombre d'erreurs qui ont été détectées. Ne faites pas attention aux erreurs « supprimées », elle ne nous concernent pas.

HEAP summary

Code : Console

```
==9029== HEAP SUMMARY:
==9029==     in use at exit: 0 bytes in 0 blocks
```

```
==9029==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==9029==
==9029== All heap blocks were freed -- no leaks are possible
```

Il s'agit d'un résumé de l'usage du tas. Lorsque vous utilisez l'allocation dynamique, vous allouez sur le tas.

Fuite de mémoire

Code : Console

```
==9029==     in use at exit: 0 bytes in 0 blocks
```

Cette ligne nous informe sur la quantité de mémoire allouée dynamiquement et non libérée à la fin du programme. Ici, bien évidemment, comme rien n'a été alloué, il ne peut pas vraiment y avoir de fuite de mémoire... 

Usage de la mémoire

Code : Console

```
==9029==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
```

Cette ligne informe de l'utilisation qui a été faite de l'allocation dynamique. Le nombre d'allocations qui on été faites, le nombre de libérations, et la quantité totale de mémoire qui a été demandée en tout au cours de l'exécution du programme.

Invalid read/write, partie 1

À présent nous allons essayer des programmes volontairement erronés pour voir quelles sortes de messages peut nous donner valgrind. 

Déréférençons NULL

Code : C

```c
#include<stdlib.h>

int main(void)
{
    int *p = NULL;
*p = 0;
    return 0;
}
```

En voila un très joli bug. Compilons vite ça !

Code : Console

```
gcc -o test02 test02.c
```

Voila, maintenant exécutons-le, mais sans valgrind :

Code : Console

```
tados@tados-laptop:valgrind_tests$ ./test02
Erreur de segmentation
```

Et PAF ! Erreur de segmentation ! Ça plante et on n'est pas avancé.
Mais essayons avec valgrind :

Code : Console

```
tados@tados-laptop:valgrind_tests$ valgrind ./test02
==9239== Memcheck, a memory error detector
==9239== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==9239== Using Valgrind-3.5.0-Debian and LibVEX; rerun with -h for copyright info
==9239== Command: ./test02
==9239==
==9239== Invalid write of size 4
==9239==    at 0x80483F9: main (in /home/tados/Documents/SdZ/valgrind_tests/test02)
==9239==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
==9239==
==9239==
==9239== Process terminating with default action of signal 11 (SIGSEGV)
==9239==  Access not within mapped region at address 0x0
==9239==    at 0x80483F9: main (in /home/tados/Documents/SdZ/valgrind_tests/test02)
==9239==  If you believe this happened as a result of a stack
==9239==  overflow in your program's main thread (unlikely but
==9239==  possible), you can try to increase the size of the
==9239==  main thread stack using the --main-stacksize= flag.
==9239==  The main thread stack size used in this run was 8388608.
==9239==
==9239== HEAP SUMMARY:
==9239==     in use at exit: 0 bytes in 0 blocks
==9239==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==9239==
==9239== All heap blocks were freed -- no leaks are possible
==9239==
==9239== For counts of detected and suppressed errors, rerun with: -v
==9239== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 12 from 7)
Erreur de segmentation
```

N'ayez pas peur ! On retrouve tout ce qu'il y avait avant, on a juste une petite tartine au milieu mais c'est pas pire que ce que l'on peut avoir en compilant. De plus, comme avec les erreurs de compilation, il faut commencer par la première.

Code : Console

```
==9239== Invalid write of size 4
==9239==    at 0x80483F9: main (in /home/tados/Documents/SdZ/valgrind_tests/test02)
==9239==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
```

Isolée, cette erreur ne fait plus peur du tout : le programme a simplement essayé d'écrire à un endroit où il n'avait pas le droit depuis la fonction main.

OK, c'est un bon indice mais, me direz-vous :
« Mais si ma fonction main fait 400 lignes de long ? »
Et bien je vous dirais qu'il ne faut pas faire des fonctions de 400 lignes de long !
« Mais si c'est pas moi qui l'ai écrite cette fonction de 400 lignes de long ? »
Ha ! alors là c'est différent. C'est vrai, j'avoue, c'est un indice plutôt maigre, alors voila ce qu'on va faire...

Compiler en débug

Oui, il faut compiler en débug, avec l'option `-g`.

Compiler en débug permet de garder ce que l'on appelle les informations de débugage. Et ce n'est pas pour rien ! Essayons-donc :

Code : Console

```
tados@tados-laptop:valgrind_tests$ gcc -o test02 test02.c -g
tados@tados-laptop:valgrind_tests$ valgrind ./test02
==9283== Memcheck, a memory error detector
==9283== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==9283== Using Valgrind-3.5.0-Debian and LibVEX; rerun with -
h for copyright info
==9283== Command: ./test02
==9283==
==9283== Invalid write of size 4
==9283==    at 0x80483F9: main (test02.c:6)
==9283==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
==9283==
==9283==
==9283== Process terminating with default action of signal 11 (SIGSEGV)
==9283==  Access not within mapped region at address 0x0
==9283==    at 0x80483F9: main (test02.c:6)
==9283==  If you believe this happened as a result of a stack
==9283==  overflow in your program's main thread (unlikely but
==9283==  possible), you can try to increase the size of the
==9283==  main thread stack using the --main-stacksize= flag.
==9283==  The main thread stack size used in this run was 8388608.
==9283==
==9283== HEAP SUMMARY:
==9283==     in use at exit: 0 bytes in 0 blocks
==9283==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==9283==
==9283== All heap blocks were freed -- no leaks are possible
==9283==
==9283== For counts of detected and suppressed errors, rerun with: -v
==9283== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 12 from 7)
Erreur de segmentation
```

Là encore, isolons l'erreur du reste de la trace :

Code : Console

```
==9283== Invalid write of size 4
==9283==    at 0x80483F9: main (test02.c:6)
==9283==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
```

Voila qui est mieux ! On sait à présent que l'erreur de segmentation a lieu dans la fonction main, à la 6° ligne du fichier test02.c ! Jetons un petit coup d'œil :

Code : C

```
#include<stdlib.h>

int main(void)
{
    int *p = NULL;
*p = 0;
```

```
    return 0;
}
```

C'est pas merveilleux ça ? 

Essayons en lecture

On va maintenant faire la même erreur, déréférencer le pointeur `NULL`, mais pour essayer de lire cette fois. Et hop, un autre petit programme foireux : 

Code : C

```c
#include<stdlib.h>

int main(void)
{
    int i;
    int *p = NULL;
i = *p;
    return 0;
}
```

Code : Console

```
tados@tados-laptop:valgrind_tests$ gcc -o test03 test03.c -g
tados@tados-laptop:valgrind_tests$ valgrind ./test03
==9459== Memcheck, a memory error detector
==9459== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==9459== Using Valgrind-3.5.0-Debian and LibVEX; rerun with -
h for copyright info
==9459== Command: ./test03
==9459==
==9459== Invalid read of size 4
==9459==    at 0x80483CE: main (test03.c:7)
==9459==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
==9459==
==9459==
==9459== Process terminating with default action of signal 11 (SIGSEGV)
==9459==  Access not within mapped region at address 0x0
==9459==    at 0x80483CE: main (test03.c:7)
==9459==  If you believe this happened as a result of a stack
==9459==  overflow in your program's main thread (unlikely but
==9459==  possible), you can try to increase the size of the
==9459==  main thread stack using the --main-stacksize= flag.
==9459==  The main thread stack size used in this run was 8388608.
==9459==
==9459== HEAP SUMMARY:
==9459==     in use at exit: 0 bytes in 0 blocks
==9459==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==9459==
==9459== All heap blocks were freed -- no leaks are possible
==9459==
==9459== For counts of detected and suppressed errors, rerun with: -v
==9459== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 12 from 7)
Erreur de segmentation
```

Et voila !

Code : Console

```
==9459== Invalid read of size 4
==9459==    at 0x80483CE: main (test03.c:7)
==9459==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
```

Lecture invalide à la 7° ligne.

À votre prochaine erreur de segmentation, pas de panique ! Un petit coup de valgrind et le tour est joué !
Et ce n'est pas fini, valgrind a plus d'un tour dans son sac !

Invalid read/write, partie 2

On va continuer à faire des accès interdits pour voir différents messages d'erreur. Vous allez voir, il suffit de lire ce que nous écrit valgrind, tout est limpide. Je ne prendrai d'ailleurs plus la peine d'isoler les erreurs, les surligner suffit amplement.

Dépassons, dépassons.

Allez ! Un petit dépassement de tableau pour voir !

Code : C

```c
#include<stdlib.h>

int main(void)
{
    int *p = malloc(3 * sizeof *p);
    if (p != NULL)
    {
p[3] = 0;
        free(p);
    }
    return 0;
}
```

Code : Console

```
tados@tados-laptop:valgrind_tests$ gcc -o test04 test04.c -g
tados@tados-laptop:valgrind_tests$ valgrind ./test04
==12044== Memcheck, a memory error detector
==12044== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==12044== Using Valgrind-3.5.0-Debian and LibVEX; rerun with -
h for copyright info
==12044== Command: ./test04
==12044==
==12044== Invalid write of size 4
==12044==    at 0x804843B: main (test04.c:8)
==12044==  Address 0x4185034 is 0 bytes after a block of size 12 alloc'd
==12044==    at 0x4023C1C: malloc (vg replace_malloc.c:195)
==12044==    by 0x8048428: main (test04.c:5)
==12044==
==12044==
==12044== HEAP SUMMARY:
==12044==     in use at exit: 0 bytes in 0 blocks
==12044==   total heap usage: 1 allocs, 1 frees, 12 bytes allocated
==12044==
==12044== All heap blocks were freed -- no leaks are possible
==12044==
==12044== For counts of detected and suppressed errors, rerun with: -v
==12044== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 12 from 7)
```

On a écrit, à la 8° ligne du fichier test04.c, à un endroit situé juste après un bloc de 12 *bytes* alloué à la 5° ligne du fichier test04.c. Et oui, lorsque l'on essaye d'accéder à une zone mémoire proche d'une zone allouée, valgrind s'en rend compte et nous le signale. Oui j'ai dit « proche », pas « au-delà ». Regardez :

Code : C

```c
#include<stdlib.h>

int main(void)
{
    int *p = malloc(3 * sizeof *p);
    if (p != NULL)
    {
p[-1] = 0;
        free(p);
    }
    return 0;
}
```

Code : Console

```
tados@tados-laptop:valgrind_tests$ gcc -o test05 test05.c -g
tados@tados-laptop:valgrind_tests$ valgrind ./test05
==12073== Memcheck, a memory error detector
==12073== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==12073== Using Valgrind-3.5.0-Debian and LibVEX; rerun with -
h for copyright info
==12073== Command: ./test05
==12073==
==12073== Invalid write of size 4
==12073==    at 0x804843B: main (test05.c:8)
==12073==  Address 0x4185024 is 4 bytes before a block of size 12 alloc'd
==12073==    at 0x4023C1C: malloc (vg replace_malloc.c:195)
==12073==    by 0x8048428: main (test05.c:5)
==12073==
==12073==
==12073== HEAP SUMMARY:
==12073==     in use at exit: 0 bytes in 0 blocks
==12073==   total heap usage: 1 allocs, 1 frees, 12 bytes allocated
==12073==
==12073== All heap blocks were freed -- no leaks are possible
==12073==
==12073== For counts of detected and suppressed errors, rerun with: -v
==12073== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 12 from 7)
```

On a écrit, à la 8° ligne du fichier test05.c, à un endroit situé juste avant un bloc de 12 *bytes* alloué à la 5° ligne du fichier test05.c. Limpide, je vous dis !

Utilisons un pointeur libéré

Code : C

```c
#include<stdlib.h>

int main(void)
{
    int *p = malloc(sizeof *p);
    if (p != NULL)
```

```
    {
        free(p);
*p = 1;
    }
    return 0;
}
```

Code : Console

```
tados@tados-laptop:valgrind_tests$ gcc -o test06 test06.c -g
tados@tados-laptop:valgrind_tests$ valgrind ./test06
==11890== Memcheck, a memory error detector
==11890== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==11890== Using Valgrind-3.5.0-Debian and LibVEX; rerun with -
h for copyright info
==11890== Command: ./test06
==11890==
==11890== Invalid write of size 4
==11890==    at 0x8048444: main (test06.c:9)
==11890==  Address 0x4185028 is 0 bytes inside a block of size 4 free'd
==11890==    at 0x4023836: free (vg replace malloc.c:325)
==11890==    by 0x804843F: main (test06.c:8)
==11890==
==11890==
==11890== HEAP SUMMARY:
==11890==     in use at exit: 0 bytes in 0 blocks
==11890==   total heap usage: 1 allocs, 1 frees, 4 bytes allocated
==11890==
==11890== All heap blocks were freed -- no leaks are possible
==11890==
==11890== For counts of detected and suppressed errors, rerun with: -v
==11890== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 12 from 7)
```

Et voila ! On nous dis exactement où on a essayé d'accéder à de la mémoire libérée, mais aussi et surtout où elle a été préalablement libérée. Et voila comment un bug difficile à trouver devient clair comme de l'eau de roche. 

Memory leaks

À présent on va s'attaquer à un autre problème : les fuites de mémoire.

Oublions de libérer de la mémoire

Dans le prochain programme, on va non seulement oublier de libérer de la mémoire, mais on va même faire pire. On va carrément perdre trace de l'addresse d'un bloc alloué dynamiquement.

Code : C

```
#include<stdlib.h>

int main(void)
{
    int *p = malloc(sizeof *p);
    if (p != NULL)
        free(p);
    p = malloc(sizeof *p);
p = NULL;
    return 0;
}
```

On compile et on exécute :

Code : Console

```
tados@tados-laptop:valgrind_tests$ gcc -o test07 test07.c -g
tados@tados-laptop:valgrind_tests$ valgrind ./test07
==9565== Memcheck, a memory error detector
==9565== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==9565== Using Valgrind-3.5.0-Debian and LibVEX; rerun with -
h for copyright info
==9565== Command: ./test07
==9565==
==9565==
==9565== HEAP SUMMARY:
==9565==     in use at exit: 4 bytes in 1 blocks
==9565==   total heap usage: 2 allocs, 1 frees, 8 bytes allocated
==9565==
==9565== LEAK SUMMARY:
==9565==    definitely lost: 4 bytes in 1 blocks
==9565==    indirectly lost: 0 bytes in 0 blocks
==9565==      possibly lost: 0 bytes in 0 blocks
==9565==    still reachable: 0 bytes in 0 blocks
==9565==         suppressed: 0 bytes in 0 blocks
==9565== Rerun with --leak-check=full to see details of leaked memory
==9565==
==9565== For counts of detected and suppressed errors, rerun with: -v
==9565== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 12 from 7)
```

Tiens ! Pas d'erreur. On retrouve le *heap summary*. On a alloué en tout 8 *bytes* en deux fois et, à la fin du programme, un bloc de 4 *bytes* n'avait pas été libéré.

Un autre résumé fait son apparition : le « *leak summary* ». C'est un résumé des fuites de mémoire de votre programme. Vous ne serez probablement coupable que de pertes definitives ou indirectes. Lorsque l'on perd la trace d'un pointeur vers une liste par exemple, celui-ci est définitivement perdu. Les nœuds de la liste, eux, sont indirectement perdus.

Et comment corrige-t-on cela ?

C'est bien de savoir qu'il y a des fuites, mais il faut plus que ça si l'on veut les corriger. En effet, mais valgrind nous invite à utiliser l'option `--leak-check=full` pour avoir plus de détails, ne nous privons pas ! Avec cette option, les fuites de mémoire sont signalées de la même manière que les erreurs de lecture ou d'écriture. Voyez plutôt :

Code : Console

```
tados@tados-laptop:valgrind_tests$ valgrind --leak-check=full ./test07
==9630== Memcheck, a memory error detector
==9630== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==9630== Using Valgrind-3.5.0-Debian and LibVEX; rerun with -
h for copyright info
==9630== Command: ./test07
==9630==
==9630==
==9630== HEAP SUMMARY:
==9630==     in use at exit: 4 bytes in 1 blocks
==9630==   total heap usage: 2 allocs, 1 frees, 8 bytes allocated
==9630==
==9630== 4 bytes in 1 blocks are definitely lost in loss record 1 of 1
==9630==    at 0x4023C1C: malloc (vg replace_malloc.c:195)
==9630==    by 0x804844B: main (test07.c:8)
==9630==
==9630== LEAK SUMMARY:
==9630==    definitely lost: 4 bytes in 1 blocks
==9630==    indirectly lost: 0 bytes in 0 blocks
```

```
==9630==      possibly lost: 0 bytes in 0 blocks
==9630==    still reachable: 0 bytes in 0 blocks
==9630==         suppressed: 0 bytes in 0 blocks
==9630==
==9630== For counts of detected and suppressed errors, rerun with: -v
==9630== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 12 from 7)
```

Encore une fois, on ne peut pas faire plus clair : un bloc de 4 bytes a été définitivement perdu dans la fonction main, à la 8° ligne du fichier test04.c ... Attends, à la 8° ligne ? 

Code : C

```c
#include<stdlib.h>

int main(void)
{
    int *p = malloc(sizeof *p);
    if (p != NULL)
        free(p);
p = malloc(sizeof *p);
    p = NULL;
    return 0;
}
```

Et oui, on nous signale où la mémoire a été allouée, pas où elle a été perdue. Mais c'est tout de même pas mal ! Vous n'allez quand même pas vous plaindre non ? 

Conditional jump or move depends on uninitialised value(s)

Valgrind permet également de détecter un autre type d'erreurs liées cette fois aux variables non initialisées. Valgrind signale en effet une erreur lorsqu'on utilise, comme condition, une valeur non initialisée ou dérivée d'une valeur non initialisée. Mais rien ne vaut un exemple.

Oublions d'initialiser

Code : C

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

int main(void)
{
    int i;
    int j;
if (j)
    {
        printf("Hello, world !\n");
    }
    return 0;
}
```

Je sais, je n'utilise pas la variable `i`, vous allez comprendre dans une petite minute. 

Code : Console

```
tados@tados-laptop:valgrind_tests$ gcc -o test08 test08.c -g
tados@tados-laptop:valgrind_tests$ valgrind ./test08
==10090== Memcheck, a memory error detector
```

```
==10090== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==10090== Using Valgrind-3.5.0-Debian and LibVEX; rerun with -
h for copyright info
==10090== Command: ./test08
==10090==
==10090== Conditional jump or move depends on uninitialised value(s)
==10090==    at 0x80483F2: main (test05.c:8)
==10090==
Hello, world !
==10090==
==10090== HEAP SUMMARY:
==10090==     in use at exit: 0 bytes in 0 blocks
==10090==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==10090==
==10090== All heap blocks were freed -- no leaks are possible
==10090==
==10090== For counts of detected and suppressed errors, rerun with: -v
==10090== Use --track-
origins=yes to see where uninitialised values come from
==10090== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 12 from 7)
```

Et voila, on utilise une variable non initialisée à la 8° ligne. On remarque aussi que valgrind nous conseille d'utiliser une option, `--track-origins=yes` en l'occurence. Valgrind consomme plus de mémoire avec cette option. Essayons-la.

Code : Console

```
tados@tados-laptop:valgrind_tests$ valgrind --track-origins=yes ./test08
==10112== Memcheck, a memory error detector
==10112== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==10112== Using Valgrind-3.5.0-Debian and LibVEX; rerun with -
h for copyright info
==10112== Command: ./test08
==10112==
==10112== Conditional jump or move depends on uninitialised value(s)
==10112==    at 0x80483F2: main (test08.c:8)
==10112==  Uninitialised value was created by a stack allocation
==10112==    at 0x80483EA: main (test08.c:5)
==10112==
Hello, world !
==10112==
==10112== HEAP SUMMARY:
==10112==     in use at exit: 0 bytes in 0 blocks
==10112==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==10112==
==10112== All heap blocks were freed -- no leaks are possible
==10112==
==10112== For counts of detected and suppressed errors, rerun with: -v
==10112== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 12 from 7)
```

Voila, on sait que la valeur initialisée vient de la pile, c'est-à-dire d'une variable allouée automatiquement. Regardons la 5° ligne qui nous est indiquée.

Code : C

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

int main(void)
{
    int i;
    int j;
    if (j)
    {
        printf("Hello, world !\n");
    }
```

```
    return 0;
}
```

Et oui, valgrind ne nous montre pas exactement de quelle variable il s'agit mais juste le bloc où elle a été définie (et ce même si cette variable est définie au milieu du bloc). Mais vous avouerez que c'est déjà pas mal.

Et avec l'allocation dynamique ?

Et bien regardons ça tout de suite.

Code : C

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

int main(void)
{
    int *i = malloc(sizeof *i);
if (*i)
    {
        printf("Hello, world !\n");
    }
    return 0;
}
```

Code : Console

```
tados@tados-laptop:valgrind_tests$ gcc -o test09 test09.c -g
tados@tados-laptop:valgrind_tests$ valgrind --track-origins=yes ./test06
==10241== Memcheck, a memory error detector
==10241== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==10241== Using Valgrind-3.5.0-Debian and LibVEX; rerun with -
h for copyright info
==10241== Command: ./test09
==10241==
==10241== Conditional jump or move depends on uninitialised value(s)
==10241==    at 0x8048435: main (test09.c:7)
==10241==  Uninitialised value was created by a heap allocation
==10241==    at 0x4023C1C: malloc (vg replace_malloc.c:195)
==10241==    by 0x8048428: main (test09.c:6)
==10241==
==10241==
==10241== HEAP SUMMARY:
==10241==     in use at exit: 4 bytes in 1 blocks
==10241==   total heap usage: 1 allocs, 0 frees, 4 bytes allocated
==10241==
==10241== LEAK SUMMARY:
==10241==    definitely lost: 4 bytes in 1 blocks
==10241==    indirectly lost: 0 bytes in 0 blocks
==10241==      possibly lost: 0 bytes in 0 blocks
==10241==    still reachable: 0 bytes in 0 blocks
==10241==         suppressed: 0 bytes in 0 blocks
==10241== Rerun with --leak-check=full to see details of leaked memory
==10241==
==10241== For counts of detected and suppressed errors, rerun with: -v
==10241== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 12 from 7)
```

Dans le cas de l'allocation dynamique, valgrind nous dit explicitement où a été allouée la variable fautive.

La pile d'appel ?

Qu'est-ce que la pile d'appel ? En voila une bonne question. Sans y répondre, disons que cela nous permet de savoir dans quelle fonction on est, par quelle fonction celle-ci a été appelée, et cætera... Et justement, cette information est extrêmement importante pour débuguer. Mettons que l'on ait une fonction qui renvoie le maximum de deux entiers :

Code : C

```
int myMmax(int a, int b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

C'est bien de savoir qu'une valeur non initialisée a été utilisée dans cette condition. Mais savoir par où elle est arrivée, c'est mieux. Que faut-il faire pour avoir cette information ? Rien. 

Exemple

Code : C

```
int myMax(int a, int b)
{
    if (a > b)
        return a;
    else
        return b;
}

int myAbs(int a)
{
    return myMax(-a, a);
}

void foo(int *i)
{
    *i = myAbs(*i);
}

int main(void)
{
    int i;
foo(&i);
    return 0;
}
```

Code : Console

```
tados@tados-laptop:valgrind_tests$ gcc -o test10 test10.c -g
tados@tados-laptop:valgrind_tests$ valgrind --track-origins=yes ./test10
==10630== Memcheck, a memory error detector
==10630== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==10630== Using Valgrind-3.5.0-Debian and LibVEX; rerun with -
h for copyright info
==10630== Command: ./test10
==10630==
==10630== Conditional jump or move depends on uninitialised value(s)
```

```
==10630==    at 0x80483BD: myMax (test10.c:3)
==10630==    by 0x80483E4: myAbs (test10.c:11)
==10630==    by 0x80483F7: foo (test10.c:16)
==10630==    by 0x8048404: main (test10.c:22)
==10630==  Uninitialised value was created by a stack allocation
==10630==    at 0x80483EA: main (test10.c:20)
==10630==
==10630==
==10630== HEAP SUMMARY:
==10630==     in use at exit: 0 bytes in 0 blocks
==10630==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==10630==
==10630== All heap blocks were freed -- no leaks are possible
==10630==
==10630== For counts of detected and suppressed errors, rerun with: -v
==10630== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 12 from 7)
```

Voila, on sait d'où elle vient et par où elle est venue. De plus, cette pile d'appel est affichée pour toutes les erreurs signalées par valgrind. Je vous le dis, la vie est belle ! 

Limitations

Voila, vous avez un aperçu de ce que peu faire valgrind, et c'est déjà énorme ! 

Un petit mot sur les optimisations du compilateur. Il est globalement conseillé de les utiliser le moins possible car elles sont susceptibles notamment de fausser les numéros de ligne ou encore, très rarement, de faire apparaître des faux positifs. Enfin, il faut savoir que memcheck ne peut pas détecter toutes les erreurs que vous êtes capable de commettre.

Code : C

```c
int main(void)
{
    int array[3];
array[3] = 0;
    return 0;
}
```

Code : Console

```
tados@tados-laptop:valgrind_tests$ gcc -o test11 test11.c -g
tados@tados-laptop:valgrind_tests$ valgrind ./test11
==12532== Memcheck, a memory error detector
==12532== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==12532== Using Valgrind-3.5.0-Debian and LibVEX; rerun with -
h for copyright info
==12532== Command: ./test11
==12532==
==12532==
==12532== HEAP SUMMARY:
==12532==     in use at exit: 0 bytes in 0 blocks
==12532==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==12532==
==12532== All heap blocks were freed -- no leaks are possible
==12532==
==12532== For counts of detected and suppressed errors, rerun with: -v
==12532== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 12 from 7)
```

Et oui, hélas, si l'on dépasse un tableau alloué automatiquement, memcheck n'y voit que du feu. Il existe bien l'option `--tool=exp-ptrcheck` qui permet de faire ce genre de vérifications, mais ptrcheck demeure encore un outil moins robuste que memcheck. La preuve, il ne parvient toujours pas à détecter cette même erreur :

Code : Console

```
tados@tados-laptop:valgrind_tests$ valgrind --tool=exp-ptrcheck ./test11
==12579== exp-ptrcheck, a heap, stack & global array overrun detector
==12579== NOTE: This is an Experimental-Class Valgrind Tool
==12579== Copyright (C) 2003-2009, and GNU GPL'd, by OpenWorks Ltd et al.
==12579== Using Valgrind-3.5.0-Debian and LibVEX; rerun with -
h for copyright info
==12579== Command: ./test11
==12579==
==12579==
==12579== For counts of detected and suppressed errors, rerun with: -v
==12579== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Vous voyez que valgrind est un outil très simple à utiliser et extrêmement puissant. Une fois que vous y avez goûté, c'est difficile de s'en passer.

Maintenant, les erreurs de segmentation ne vous feront plus peur !