Optimisation de programme

Lundi 29 Janvier 2018

Michael FRANÇOIS

francois@esiea.fr

https://francois.esiea.fr

Optimisation de programme

Optimisation de programme

• **Optimisation de programme :** processus général consistant à apporter des modifications progressives à un programme dans le but d'obtenir une version plus efficace.

• Pourquoi optimiser un programme ?
  - dans le cas d'une utilisation fréquente ;
  - utilisation sur de grands ensembles de données.

• La meilleure façon d'améliorer les performances d'un algorithme est d'affiner le code graduellement par transformations successives.

Remarque : avant de se lancer sérieusement dans une optimisation de programme, il faut d'abord connaître le système d'exploitation et l'environnement de programmation utilisés.

• La première étape dans l'implémentation d'un algorithme, consiste à écrire d'abord une version de travail de l'algorithme dans sa forme la plus **simple** :

- ce qui fournit une base pour affinements et améliorations.

• Il faudra ensuite identifier la **boucle principale** et essayer de minimiser le nombre d'instructions qu'elle contient :

- pour cela, on peut lancer le programme et repérer ainsi les instructions exécutées le plus souvent.

• Il faudra examiner chaque instruction de la boucle principale, afin de savoir si elle est vraiment **indispensable** ou **non**.

- L'implémentation d'un algorithme est un processus cyclique :
  - on conçoit le programme ;
  - on élimine ses erreurs ;
  - on découvre ses propriétés ;
  - puis on affine la mise en œuvre jusqu'à obtenir les performances requises.
- Dans ce processus, l'analyse mathématique n'est pas à négliger car, elle peut suggérer des algorithmes susceptibles de donner les meilleures performances pour un problème donné :
  - après l'implémentation, on peut toujours vérifier si le résultat est celui attendu.

Optimisation au niveau du code

Considérons le code C suivant :

```c
#include <stdio.h>

int puissance(int Val, int P)
{
 int i, Pui;
 Pui=1;
 for (i=1; i<=P; i++)
 {
   Pui = Pui * Val;
 }
 return Pui;
}

int main (int argc, char ** argv)
{
 int i, Val, P;
 Val = 2; P=15;
 for (i=0; i<puissance(Val, P); i++)
 {
   printf("i = %d\n", i);
 }
 return 0;
}
```

Que fait exactement ce code ?
Comment l'optimiser ?

Éviter la répétition inutile d'appels de fonctions

CODE 1

```c
#include <stdio.h>

int puissance(int Val, int P)
{
 int i, Pui;
 Pui=1;
 for (i=1; i<=P; i++)
 {
   Pui = Pui * Val;
 }
 return Pui;
}

int main (int argc, char ** argv)
{
 int i, Val, P;
 Val = 2; P=15;
 for (i=0; i<puissance(Val, P); i++)
 {
   printf("i = %d\n", i);
 }
 return 0;
}
```

CODE 1 (amélioré)

```c
#include <stdio.h>

int puissance(int Val, int P)
{
 int i, Pui;
 Pui=1;
 for (i=1; i<=P; i++)
 {
   Pui = Pui * Val;
 }
 return Pui;
}

int main (int argc, char ** argv)
{
 int i, Val, P, fin;
 Val = 2; P=15;
 fin = puissance(Val, P);
 for (i=0; i<fin; i++)
 {
   printf("i = %d\n", i);
 }
 return 0;
}
```

Remarque : le code exécute la fonction `puissance()` à chaque itération, ce qui est inutile car la valeur de fin de boucle (*i.e.* 2^{15}) ne change pas.

Considérons le code C suivant :

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

int main (int argc, char ** argv)
{
 int i, min, max, TAB[1000];
 srand(time(NULL));
 for (i=0; i<1000; i++)
 {TAB[i] = rand() % 100000;}
 min = TAB[0]; max=TAB[0];
 for (i=1; i<1000; i++)
 {
   if (TAB[i]<min){min=TAB[i];}
 }

 for (i=1; i<1000; i++)
 {
   if (TAB[i]>max){max=TAB[i];}
 }
 printf("min=%d, max=%d\n", min,max);
 return 0;
}
```

Que fait exactement ce code ?
Comment l'optimiser ?

Minimiser les parcours de tableaux

CODE 2

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

int main (int argc, char ** argv)
{
 int i, min, max, TAB[1000];
 srand(time(NULL));
 for (i=0; i<1000; i++)
 {TAB[i] = rand() % 100000;}
 min = TAB[0]; max=TAB[0];
 for (i=1; i<1000; i++)
 {
   if (TAB[i]<min){min=TAB[i];}
 }

 for (i=1; i<1000; i++)
 {
   if (TAB[i]>max){max=TAB[i];}
 }
 printf("min=%d, max=%d\n", min,max);
 return 0;
}
```

CODE 2 (amélioré)

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

int main (int argc, char ** argv)
{
 int i, min, max, TAB[1000], Elt;
 srand(time(NULL));
 for (i=0; i<1000; i++)
 {TAB[i] = rand() % 100000;}
 min = TAB[0]; max=min;
 for (i=1; i<1000; i++)
 {
   Elt = TAB[i];
   if (Elt<min){min=Elt;}
   if (Elt>max){max=Elt;}
 }
 printf("min=%d, max=%d\n", min,max);
 return 0;
}
```

Remarque : pour calculer le min et le max des valeurs stockées dans un tableau, il est inutile de parcourir plusieurs fois le tableau.
Regrouper les boucles quand c'est possible !!!

Déroulement de boucle

CODE 3

```c
#include <stdio.h>

int main (int argc, char ** argv)
{
 int i, TAB[100000];

 for (i=0; i<100000; i++)
 {
   TAB[i] = i;
 }
 return 0;
}
```

CODE 3 (amélioré)

```c
#include <stdio.h>

int main (int argc, char ** argv)
{
 int i, TAB[100000];

 for (i=0; i<100000; )
 {
   TAB[i] = i++;
   TAB[i] = i++;
   TAB[i] = i++;
   TAB[i] = i++;
   TAB[i] = i++;
 }
 return 0;
}
```

Remarque : le fait de retirer `i++` comme 3ème argument du for est très dangereux car, on peut déborder au delà de la taille requise. Ici, cette technique permet de diminuer le nombre d'instructions et de sauts. Dans le cas où la taille est très grande, il y a optimisation.

Considérons le code C suivant :

```c
#include <stdio.h>

int main(int argc, char** argv)
{
  int i;
  for (i=0; i<1000000; i++)
  {
    if (i==0 || i==500000 || i==999999)
    {
      printf("%c", '$');
    }
    else
    {
      printf("%c", '-');
    }
  }
  printf("\n");
  return 0;
}
```

Que fait exactement ce code ?
Comment l'optimiser ?

Minimiser les tests inutiles dans une boucle

CODE 4

```c
#include <stdio.h>

int main(int argc, char** argv)
{
  int i;
  for (i=0; i<1000000; i++)
  {
    if (i==0 || i==500000 || i==999999)
    {
      printf("%c", '$');
    }
    else
    {
      printf("%c", '-');
    }
  }
  printf("\n");
  return 0;
}
```

CODE 4 (amélioré)

```c
#include <stdio.h>

int main(int argc, char** argv)
{
  int i;
  printf("%c", '$');
  for (i=1; i<500000; i++)
  {
    printf("%c", '-');
  }
  printf("%c", '$');
  for (i=500001; i<999999; i++)
  {
    printf("%c", '-');
  }
  printf("%c\n", '$');
  return 0;
}
```

Remarque : Dans le cas où la réussite d'un test "maîtrisé" est très rare, il ne faut pas garder ce test dans une boucle car il est trop coûteux. Il faut isoler l'exécution de ce test, quitte à diviser en deux la boucle. On réalise un gain de temps surtout dans le cas où la taille du tableau est très grande.

Considérons le code C suivant :

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

int Test (int N)
{
  int i, res=1;
  for (i=2; i<N; i++)
  {
    if ((N % i) == 0)
    {
      res = 0;
    }
  }
  return res;
}

int main(int argc, char ** argv)
{
   printf("%d\n", Test(15439));
   return 0;
}
```

Que fait exactement ce code ?
Comment l'optimiser ?

Amélioration 1 (ajouter un `break`) :

CODE 5

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

int Test (int N)
{
  int i, res=1;
  for (i=2; i<N; i++)
  {
    if ((N % i) == 0)
    {
      res = 0;
    }
  }
  return res;
}

int main(int argc, char ** argv)
{
   printf("%d\n", Test(15439));
   return 0;
}
```

CODE 5 (amélioré)

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

int Test (int N)
{
  int i, res=1;
  for (i=2; i<N; i++)
  {
    if ((N % i) == 0)
    {
      res=0;
      break;
    }
  }
  return res;
}


int main(int argc, char ** argv )
{

  printf("%d\n", Test(15439));
  return 0;
}
```

Remarque : le mot clé `break` permet d'interrompre la boucle `for` une fois qu'un facteur est trouvé. Mais quand on ne trouve pas de facteur, la boucle ira jusqu'à N.

Amélioration 2 (ajouter un `return`) :

CODE 5

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

int Test (int N)
{
  int i, res=1;
  for (i=2; i<N; i++)
  {
    if ((N % i) == 0)
    {
      res = 0;
    }
  }
  return res;
}

int main(int argc, char ** argv)
{
   printf("%d\n", Test(15439));
   return 0;
}
```

CODE 5 (amélioré)

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

int Test (int N)
{
  int i, res=1;
  for (i=2; i<N; i++)
  {
    if ((N % i) == 0)
    {
      return 0;
    }
  }
  return res;
}


int main(int argc, char ** argv )
{

  printf("%d\n", Test(15439));
  return 0;
}
```

Remarque : on peut retourner 0 dès lors qu'un facteur est trouvé. Mais quand on ne trouve pas de facteur, la boucle ira jusqu'à N.

Amélioration 3 (limiter la fin de boucle) :

CODE 5

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

int Test (int N)
{
  int i, res=1;
  for (i=2; i<N; i++)
  {
    if ((N % i) == 0)
    {
      res = 0;
    }
  }
  return res;
}

int main(int argc, char ** argv)
{
   printf("%d\n", Test(15439));
   return 0;
}
```

CODE 5 (amélioré)

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

int Test (int N)
{
  int i, res=1, fin=sqrt(N);
  for (i=2; i<=fin; i++)
  {
    if ((N % i) == 0)
    {
      res = 0 ;
    }
  }
  return res;
}

int main(int argc, char ** argv )
{
  printf("%d\n", Test(15439));
  return 0;
}
```

Remarque : pour tester la primalité d'un nombre il est inutile de tester les facteurs qui sont plus grands que la racine carrée du nombre.

Amélioration 4 (mixte) :

CODE 5

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

int Test (int N)
{
  int i, res=1;
  for (i=2; i<N; i++)
  {
    if ((N % i) == 0)
    {
      res = 0;
    }
  }
  return res;
}

int main(int argc, char ** argv)
{
   printf("%d\n", Test(15439));
   return 0;
}
```

CODE 5 (amélioré)

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

int Test (int N)
{
  int i, fin=sqrt(N);
  for (i=2; i<=fin; i++)
  {
    if ((N % i) == 0)
    {
      return 0 ;
    }
  }
  return 1;
}

int main(int argc, char ** argv )
{
  printf("%d\n", Test(15438));
  return 0;
}
```

Remarque : le `return 0` évite d'aller jusqu'à $\sqrt{N}$ dans le cas où on trouve un facteur. Et même dans le cas où on n'en trouve pas, la boucle n'ira pas au delà de $\sqrt{N}$.

Optimisation au niveau du compilateur

Niveaux d'optimisation du compilateur gcc

- Une grande variété de niveaux d'optimisation est disponible via gcc.
- Il y a des options d'optimisations qui réduisent la taille du code machine résultant. D'autres essaient de produire un code plus rapide, en augmentant potentiellement sa taille.
- On distingue 5 niveaux d'optimisations (-O) :
  - -O0
  - -O1
  - -O2
  - -O3
  - -Os

Niveaux d'optimisation -O0

• -O0 $\Longrightarrow$ (lettre O suivie d'un zéro), ce niveau désactive complètement l'optimisation. C'est le comportement par défaut de GCC.

• Les exécutables produits gardent leurs tables de symboles et d'autres informations. Ceci a pour conséquence d'alourdir le fichier exécutable. On peut retirer ces données en utilisant l'option -s (*i.e.* strip);

Niveaux d'optimisation -O1

- -O1 $\Longrightarrow$ premier niveau d'optimisation.
- Le compilateur va essayer de générer un code plus rapide et plus léger sans prendre plus de temps au moment de la compilation.
- C'est le niveau classique d'optimisation.

Niveaux d'optimisation -O2

- -O2 $\Longrightarrow$ optimisation un peu au dessus du niveau -O1.
- C'est le niveau recommandé d'optimisation, sauf si vous avez des besoins spécifiques.
- Avec ce niveau le compilateur augmente le temps de compilation, mais surtout les performances du code généré.

Niveaux d'optimisation -O3

- -O3 $\Longrightarrow$ optimise encore plus que -O2.
- Il s'agit du plus haut niveau d'optimisation possible, mais aussi plus risqué.
- Ce niveau active toutes les optimisations spécifiées par -O2 plus d'autres.

Niveaux d'optimisation -Os

- -Os $\Longrightarrow$ optimisation concernant la taille du code.
- Ce niveau active toutes les optimisations spécifiées par -O2 qui n'ont pas d'impacts négatifs sur la taille du code. Elle effectue également d'autres optimisations conçues pour réduire la taille du code.
- L'utilisation de -Os n'est pas très recommandé, sauf pour des besoins très ciblés.

Valgrind

Valgrind

• **Valgrind :** développé en 2000 par Julian Seward, cet outil libre permet d'analyser principalement des programmes écrits en C/C++.

• Environnement supportés : GNU/Linux, Mac OS X.

• Il possède plusieurs modules notamment *Memcheck* permettant de vérifier que :

- les valeurs ou pointeurs utilisés sont initialisés ;
- les zones mémoires libérées ou non allouées ne sont accédées ;
- une zone mémoire n'est pas doublement libérée ;
- la libération des mémoires allouées.

• On trouve également des modules comme : *Cachegrind, Massif, Callgrind, helgrind, etc.*

Comment utiliser Valgrind ?

• Considérons le programme Hello world suivant :

```
#include<stdio.h>

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

• Ce programme affiche simplement le message `Hello world !` sur la sortie standard.

```
$ gcc -o EXEC test.c
$ valgrind ./EXEC
==22464== Memcheck, a memory error detector
==22464== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==22464== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==22464== Command: ./EXEC
==22464==
Hello world !
==22464==
==22464== HEAP SUMMARY:
==22464==     in use at exit: 0 bytes in 0 blocks
==22464==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==22464==
==22464== All heap blocks were freed -- no leaks are possible
==22464==
==22464== For counts of detected and suppressed errors, rerun with: -v
==22464== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Remarque : le numéro à gauche "22464" indique le numéro du processus, donc pas forcément nécessaire pour l'analyse du programme.

HEAP SUMMARY : (Résumé de l'usage du tas en cas d'allocation dynamique)

```
==22464== HEAP SUMMARY:
==22464==     in use at exit: 0 bytes in 0 blocks
==22464==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==22464==
==22464== All heap blocks were freed -- no leaks are possible
```

• `in use at exit:` indique la quantité de mémoire allouée dynamiquement et non libérée à la fin du programme. Ici pas de fuite de mémoire puisque rien n'a été alloué.

• `total heap usage:` indique l'utilisation faite de l'allocation dynamique. Le nombre d'allocations effectué, de libérations, et la quantité totale de mémoire allouée pendant l'exécution du programme.

ERROR SUMMARY : (Résumé du nombre d'erreurs détectées)

```
==22464== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Exemple sur un petit programme erroné

• Considérons le programme suivant :

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

int main(void)
{
    int * POINTEUR = NULL;
    *POINTEUR = 94;
    return 0;
}
-------------------------------------------------------
```

• Ce programme tente d'affecter la valeur 94 au contenu (type `int`) de la case mémoire indexée par le pointeur `POINTEUR`.

```
$ gcc -o EXEC test.c
$ ./EXEC
Segmentation fault (core dumped)
$ gcc -o EXEC test.c -g
$ valgrind ./EXEC
==23520== Memcheck, a memory error detector
==23520== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==23520== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==23520== Command: ./EXEC
==23520==
==23520== Invalid write of size 4
==23520==    at 0x4004FD: main (test.c:7)
==23520==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
==23520==
==23520==
==23520== Process terminating with default action of signal 11 (SIGSEGV)
==23520==  Access not within mapped region at address 0x0
==23520==    at 0x4004FD: main (test.c:7)
==23520==  If you believe this happened as a result of a stack
==23520==  overflow in your program's main thread (unlikely but
==23520==  possible), you can try to increase the size of the
==23520==  main thread stack using the --main-stacksize= flag.
==23520==  The main thread stack size used in this run was 8388608.
==23520==
==23520== HEAP SUMMARY:
==23520==     in use at exit: 0 bytes in 0 blocks
==23520==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==23520==
==23520== All heap blocks were freed -- no leaks are possible
==23520==
==23520== For counts of detected and suppressed errors, rerun with: -v
==23520== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
Segmentation fault (core dumped)
```

Remarque : compiler avec '-g' permet d'obtenir les infos de débogage. Ainsi, on constate que l'erreur de segmentation a lieu dans la fonction `main` à la 7ème ligne (*POINTEUR = 94;). Que s'est-il passé ?

Un autre exemple de programme erroné

• Considérons le programme suivant :

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

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

• Ce programme tente d'affecter à la variable `VAL`, la valeur (type `int`) possédée par le pointeur.

```
$ gcc -o EXEC test.c
$ ./EXEC
Segmentation fault (core dumped)
$ gcc -o EXEC test.c -g
$ valgrind ./EXEC
==23358== Memcheck, a memory error detector
==23358== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==23358== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==23358== Command: ./EXEC
==23358==
==23358== Invalid read of size 4
==23358==    at 0x4004FD: main (test.c:8)
==23358==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
==23358==
==23358==
==23358== Process terminating with default action of signal 11 (SIGSEGV)
==23358==  Access not within mapped region at address 0x0
==23358==    at 0x4004FD: main (test.c:8)
==23358==  If you believe this happened as a result of a stack
==23358==  overflow in your program's main thread (unlikely but
==23358==  possible), you can try to increase the size of the
==23358==  main thread stack using the --main-stacksize= flag.
==23358==  The main thread stack size used in this run was 8388608.
==23358==
==23358== HEAP SUMMARY:
==23358==     in use at exit: 0 bytes in 0 blocks
==23358==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==23358==
==23358== All heap blocks were freed -- no leaks are possible
==23358==
==23358== For counts of detected and suppressed errors, rerun with: -v
==23358== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
Segmentation fault (core dumped)
```

Remarque : lecture invalide à la 8ème ligne (VAL = * POINTEUR;) dans la fonction `main`. Pourquoi ?

Un autre exemple de programme erroné

- Considérons le programme suivant :

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

int main(int argc, char** argv)
{
  int i, TAB[10];
  for (i = 0; i < 9; i++)
  {
    TAB[i] = i;
  }
  for (i = 0; i < 10; i++)
  {
    printf("%d ", TAB[i]);
  }
  printf("\n");
  return 0;
}
```

- Ce programme initialise et affiche les éléments d'un tableau. Sauf que la dernière valeur du tableau n'a pas été initialisée.

```
$ gcc -o EXEC test.c -Wall -g
$ valgrind ./EXEC
==27737== Memcheck, a memory error detector
==27737== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==27737== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==27737== Command: ./EXEC
==27737==
==27737== Conditional jump or move depends on uninitialised value(s)
==27737==    at 0x4E8147E: vfprintf (vfprintf.c:1660)
==27737==    by 0x4E8B388: printf (printf.c:33)
==27737==    by 0x4005CD: main (test.c:13)
==27737==
==27737== Use of uninitialised value of size 8
==27737==    at 0x4E8093B: _itoa_word (_itoa.c:179)
==27737==    by 0x4E845E6: vfprintf (vfprintf.c:1660)
==27737==    by 0x4E8B388: printf (printf.c:33)
==27737==    by 0x4005CD: main (test.c:13)
==27737==
==27737== Conditional jump or move depends on uninitialised value(s)
==27737==    at 0x4E80945: _itoa_word (_itoa.c:179)
==27737==    by 0x4E845E6: vfprintf (vfprintf.c:1660)
==27737==    by 0x4E8B388: printf (printf.c:33)
==27737==    by 0x4005CD: main (test.c:13)
==27737==
==27737== Conditional jump or move depends on uninitialised value(s)
==27737==    at 0x4E84632: vfprintf (vfprintf.c:1660)
==27737==    by 0x4E8B388: printf (printf.c:33)
==27737==    by 0x4005CD: main (test.c:13)
==27737==
==27737== Conditional jump or move depends on uninitialised value(s)
==27737==    at 0x4E81549: vfprintf (vfprintf.c:1660)
==27737==    by 0x4E8B388: printf (printf.c:33)
==27737==    by 0x4005CD: main (test.c:13)
==27737==
==27737== Conditional jump or move depends on uninitialised value(s)
==27737==    at 0x4E815CC: vfprintf (vfprintf.c:1660)
==27737==    by 0x4E8B388: printf (printf.c:33)
==27737==    by 0x4005CD: main (test.c:13)
==27737==
0 1 2 3 4 5 6 7 8 15
==27737==
==27737== HEAP SUMMARY:
==27737==     in use at exit: 0 bytes in 0 blocks
==27737==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==27737==
==27737== All heap blocks were freed -- no leaks are possible
==27737==
==27737== For counts of detected and suppressed errors, rerun with: -v
==27737== Use --track-origins=yes to see where uninitialised values come from
==27737== ERROR SUMMARY: 8 errors from 6 contexts (suppressed: 0 from 0)
```

Le problème vient de l'utilisation d'une variable qui n'a pas été initialisée. Valgrind indique que l'accès à cette variable se fait à la ligne 13 via l'instruction : `printf("%d ", TAB[i]);`
on peut utiliser l'option : `--track-origins=yes`, pour obtenir des informations supplémentaires sur la provenance de la variable non-initialisée.

```
$ gcc -o EXEC test.c -g -Wall
$ valgrind --track-origins=yes ./EXEC
==28461== Memcheck, a memory error detector
==28461== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==28461== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==28461== Command: ./EXEC
==28461==
==28461== Conditional jump or move depends on uninitialised value(s)
==28461==    at 0x4E8147E: vfprintf (vfprintf.c:1660)
==28461==    by 0x4E8B388: printf (printf.c:33)
==28461==    by 0x4005CD: main (test.c:13)
==28461==  Uninitialised value was created by a stack allocation
==28461==    at 0x40057D: main (test.c:5)
==28461==
==28461== Use of uninitialised value of size 8
==28461==    at 0x4E8093B: _itoa_word (_itoa.c:179)
==28461==    by 0x4E845E6: vfprintf (vfprintf.c:1660)
==28461==    by 0x4E8B388: printf (printf.c:33)
==28461==    by 0x4005CD: main (test.c:13)
==28461==  Uninitialised value was created by a stack allocation
==28461==    at 0x40057D: main (test.c:5)
```

Remarque : on voit que la variable non-initialisée provient d'une allocation sur la pile, lors de la déclaration du tableau `TAB[10]`.

Un exemple de programme contenant une faille

- Considérons le programme suivant :

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

int main(int argc, char** argv)
{
  char * str = malloc(sizeof(char)*5);
  print{"Saisir une chaîne :\n"}
  gets(str);
  printf("%s\n",str);
  return 0;
}
-----------------------------------------------------
```

- Ce programme attend que l'utilisateur donne une chaîne de caractères (sans préciser la taille) pour l'afficher en ligne. Le problème est que seulement 5 cases mémoires ont été allouées. Valgrind peut ne pas détecter cette faille pendant l'exécution.

Exemple d'exécution sans dépassement de cases mémoires réservées :

```
$ valgrind ./EXEC
==23234== Memcheck, a memory error detector
==23234== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==23234== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==23234== Command: ./EXEC
==23234==
Saisir une chaîne :
algo
algo
==23234==
==23234== HEAP SUMMARY:
==23234==     in use at exit: 5 bytes in 1 blocks
==23234==   total heap usage: 1 allocs, 0 frees, 5 bytes allocated
==23234==
==23234== LEAK SUMMARY:
==23234==    definitely lost: 5 bytes in 1 blocks
==23234==    indirectly lost: 0 bytes in 0 blocks
==23234==      possibly lost: 0 bytes in 0 blocks
==23234==    still reachable: 0 bytes in 0 blocks
==23234==         suppressed: 0 bytes in 0 blocks
==23234== Rerun with --leak-check=full to see details of leaked memory
==23234==
==23234== For counts of detected and suppressed errors, rerun with: -v
==23234== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Remarque : pas d'erreurs car la taille de la chaîne saisie par l'utilisateur est conforme. Pourtant le programme présente une faiblesse.

Exemple d'exécution avec dépassement de cases mémoires réservées :

```
$ valgrind ./EXEC
==29052== Memcheck, a memory error detector
==29052== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==29052== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==29052== Command: ./EXEC
==29052==
Saisir un chaîne :
algos
==29052== Invalid write of size 1
==29052==    at 0x4EA6FA3: gets (iogets.c:64)
==29052==    by 0x4005EF: main (test.c:8)
==29052==  Address 0x51fd045 is 0 bytes after a block of size 5 alloc'd
==29052==    at 0x4C2AB80: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==29052==    by 0x4005D5: main (test.c:6)
==29052==
==29052== Invalid read of size 1
==29052==    at 0x4C2E0F4: strlen (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==29052==    by 0x4EA78BB: puts (ioputs.c:36)
==29052==    by 0x4005FB: main (test.c:9)
==29052==  Address 0x51fd045 is 0 bytes after a block of size 5 alloc'd
==29052==    at 0x4C2AB80: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==29052==    by 0x4005D5: main (test.c:6)
==29052==
algos
==29052==
==29052== HEAP SUMMARY:
==29052==     in use at exit: 5 bytes in 1 blocks
==29052==   total heap usage: 1 allocs, 0 frees, 5 bytes allocated
==29052==
==29052== LEAK SUMMARY:
==29052==    definitely lost: 5 bytes in 1 blocks
==29052==    indirectly lost: 0 bytes in 0 blocks
==29052==      possibly lost: 0 bytes in 0 blocks
==29052==    still reachable: 0 bytes in 0 blocks
==29052==         suppressed: 0 bytes in 0 blocks
==29052== Rerun with --leak-check=full to see details of leaked memory
==29052==
==29052== For counts of detected and suppressed errors, rerun with: -v
==29052== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

Remarque : 2 erreurs. Des zones mémoires non-réservées tentées d'être accessibles en lecture/écriture. Pour ce cas, l'erreur a été détectée.

Bibliographie

- R. ERRA, "Cours 2 d'informatique", 1A-S2 2013-2014 ESIEA.

- http://openclassrooms.com/courses/debuguer-facilement-avec-valgrind
- http://pages.cs.wisc.edu/~bart/537/valgrind.html