Bonnes pratiques de développement et débogage de programme en langage C et Fortran

Sylvain Ferrand

CMAP

Octobre 2007

Table des matières

Remarques générales

Le temps de débogage d'un programme peut représenter jusqu'à 80% du temps de developpement

$\longrightarrow$ Mieux vaut connaitre quelque astuces pour les éviter et savoir les traquer!

Les bonnes pratiques

Utilisez des conventions de nommage pour les variables, les constantes et les fonctions

- Constantes et variables globales en MAJUSCULE
- Eviter les noms trop proches les uns des autres
- Utiliser des noms courts et explicites
  - Pour une variable de stockage : que contient-elle ?
  - Pour un tableau : que contient une case ?
  - ...

Les bonnes pratiques

Utilisez des conventions de nommage pour les variables, les constantes et les fonctions

- Constantes et variables globales en MAJUSCULE
- Eviter les noms trop proches les uns des autres
- Utiliser des noms courts et explicites
  - Pour une variable de stockage : que contient-elle ?
  - Pour un tableau : que contient une case ?
  - ...

Les bonnes pratiques

Utilisez des conventions de nommage pour les variables, les constantes et les fonctions

- Constantes et variables globales en MAJUSCULE
- Eviter les noms trop proches les uns des autres
- Utiliser des noms courts et explicites
  - Pour une variable de stockage : que contient-elle ?
  - Pour un tableau : que contient une case ?
  - ...

Les bonnes pratiques

Utilisez des conventions de nommage pour les variables, les constantes et les fonctions

- Constantes et variables globales en MAJUSCULE
- Eviter les noms trop proches les uns des autres
- Utiliser des noms courts et explicites
  - Pour une variable de stockage : que contient-elle ?
  - Pour un tableau : que contient une case ?
  - ...

Le 'coding style'

Pensez à bien indenter votre code
Voici un exemple, en style Kernighan et Ritchie

```
void truc()
{
     if (x > y) {
          faire_un_truc();
     }
     fin();
}
```

Le 'coding style' (suite)

Commentez raisonnablement et lisiblement,

- Bien décrire les arguments des fonctions
- Prévoir l'exploitation par un outil de documentation automatique (ex: ROBOdoc)
- Eviter d'indiquer le fonctionnemement du code dans les commentaire.
- Eviter de paraphraser le code

Le 'coding style' (suite)

Commentez raisonnablement et lisiblement,

- Bien décrire les arguments des fonctions
- Prévoir l'exploitation par un outil de documentation automatique (ex: ROBOdoc)
- Eviter d'indiquer le fonctionnemement du code dans les commentaire.
- Eviter de paraphraser le code

Le 'coding style' (suite)

Commentez raisonnablement et lisiblement,

- Bien décrire les arguments des fonctions
- Prévoir l'exploitation par un outil de documentation automatique (ex: ROBOdoc)
- Eviter d'indiquer le fonctionnemement du code dans les commentaire.
- Eviter de paraphraser le code

Le 'coding style' (suite)

Commentez raisonnablement et lisiblement,

- Bien décrire les arguments des fonctions
- Prévoir l'exploitation par un outil de documentation automatique (ex: ROBOdoc)
- Eviter d'indiquer le fonctionnemement du code dans les commentaire.
- Eviter de paraphraser le code

Le 'coding style' (suite)

Commentez raisonnablement et lisiblement,

- Bien décrire les arguments des fonctions
- Prévoir l'exploitation par un outil de documentation automatique (ex: ROBOdoc)
- Eviter d'indiquer le fonctionnemement du code dans les commentaire.
- Eviter de paraphraser le code

Des bugs classiques

- Les débordements de tableau
- Les problèmes d'initialisation (généralement signalés par le compilateur)
- Les problèmes liés aux conversions implicites dans les expressions mathématiques
- Les typos :
  - Commentaires mal fermés :

```
a=b; /* this is a bug
c=d; /* c=d will never happen */
```

  - Confusions entre test et assignation en C

```
if(a=b) c;      /* a always equals b */
```

Des bugs classiques

- Les débordements de tableau
- Les problèmes d'initialisation (généralement signalés par le compilateur)
- Les problèmes liés aux conversions implicites dans les expressions mathématiques
- Les typos :
  - Commentaires mal fermés :

```
a=b; /* this is a bug
c=d; /* c=d will never happen */
```

  - Confusions entre test et assignation en C

```
if(a=b) c;          /* a always equals b */
```

Des bugs classiques

- Les débordements de tableau
- Les problèmes d'initialisation (généralement signalés par le compilateur)
- Les problèmes liés aux conversions implicites dans les expressions mathématiques
- Les typos :
  - Commentaires mal fermés :

```
a=b; /* this is a bug
c=d; /* c=d will never happen */
```

  - Confusions entre test et assignation en C

```
if(a=b) c;      /* a always equals b */
```

Des bugs classiques

- Les débordements de tableau
- Les problèmes d'initialisation (généralement signalés par le compilateur)
- Les problèmes liés aux conversions implicites dans les expressions mathématiques
- Les typos :
  - Commentaires mal fermés :

```
a=b; /* this is a bug
c=d; /* c=d will never happen */
```

  - Confusions entre test et assignation en C

```
if(a=b) c;          /* a always equals b */
```

Eviter les structures obsolètes du Fortran

- Banir le typage implicite, utilisez `implicite none`
- Eviter les instructions `do label...continue` et préférer les boucles F90 (mais universelles): `do...enddo` et `dowhile...enddo`
- Eviter les `if` logiques et toute les structures utilisant des étiquettes. Utilisez la structure `if...then...endif`
- Oublier les instructions archaïques `goto, continue, entry, equivalence`

Eviter les structures obsolètes du Fortran

- Banir le typage implicite, utilisez `implicite none`
- Eviter les instructions `do label...continue` et préférer les boucles F90 (mais universelles): `do...enddo` et `dowhile...enddo`
- Eviter les `if` logiques et toute les structures utilisant des étiquettes. Utilisez la structure `if...then...endif`
- Oublier les instructions archaïques `goto, continue, entry, equivalence`

Eviter les structures obsolètes du Fortran

- Banir le typage implicite, utilisez `implicite none`
- Eviter les instructions `do label...continue` et préférer les boucles F90 (mais universelles): `do...enddo` et `dowhile...enddo`
- Eviter les `if` logiques et toute les structures utilisant des étiquettes. Utilisez la structure `if...then...endif`
- Oublier les instructions archaïques `goto, continue, entry, equivalence`

Eviter les structures obsolètes du Fortran

- Banir le typage implicite, utilisez `implicite none`
- Eviter les instructions `do label...continue` et préférer les boucles F90 (mais universelles): `do...enddo` et `dowhile...enddo`
- Eviter les `if` logiques et toute les structures utilisant des étiquettes. Utilisez la structure `if...then...endif`
- Oublier les instructions archaïques `goto, continue, entry, equivalence`

Les options de compilation

GCC

- `-Wall` Active tous les avertissements standards
- `-Wextra` Encore d'autres options d'avertissement

gfortran

- `-Wall` Active tous les avertissements standards
- `-Wimplicit` Avertissement sur les typages implicites

ifort

- `-warn all` Active tous les avertissements standards

Table des matières

Table des matières

Qu'est ce qu'un débogueur?

Le débogueur est destiné à aider le programmeur à détecter des bogues dans un programme.
Il permet en général d'exécuter le programme tout en permettant de :

- exécuter le programme pas à pas
- contrôler l'état de la mémoire et des variables
- agir sur la mémoire
- définir des points d'arrêts
- ...

Panorama des debugeurs du marché

Sous Windows

(Microsoft) **Visual Studio Debugger (codeview)**

Sous Linux et unix

gdb, dbx, ddd
(Intel) **IDB** *syntaxe compatible dbx et gdb*
Purify, valgrind

Débugueurs parrallèles

(Totalview technologie) **Totalview**
(Allinea) **Distributed Debugging Tool (DDT)**
(Portland) **pgdbg**
(Microsoft) **Visual studio**

Table des matières

Le GNU Debugger "GDB"

- Le débogueur standard du projet GNU
- Fonctionne sur un grand nombre d'UNIX et d'architechtures
- Logiciel en ligne de commande souvent appellé par d'autre programmes (Eclipse, DDD)
- Deux modes d'utilisation
  - Déboggage d'un programme en cours d'exécution
  - Déboggage post-mortem (fichier core)

Le GNU Debugger "GDB"

- Le débogueur standard du projet GNU
- Fonctionne sur un grand nombre d'UNIX et d'architechtures
- Logiciel en ligne de commande souvent appellé par d'autre programmes (Eclipse, DDD)
- Deux modes d'utilisation
  - Déboggage d'un programme en cours d'exécution
  - Déboggage post-mortem (fichier core)

Le GNU Debugger "GDB"

- Le débogueur standard du projet GNU
- Fonctionne sur un grand nombre d'UNIX et d'architechtures
- Logiciel en ligne de commande souvent appellé par d'autre programmes (Eclipse, DDD)
- Deux modes d'utilisation
  - Déboggage d'un programme en cours d'exécution
  - Déboggage post-mortem (fichier core)

Le GNU Debugger "GDB"

- Le débogueur standard du projet GNU
- Fonctionne sur un grand nombre d'UNIX et d'architechtures
- Logiciel en ligne de commande souvent appellé par d'autre programmes (Eclipse, DDD)
- Deux modes d'utilisation
  - Déboggage d'un programme en cours d'exécution
  - Déboggage post-mortem (fichier core)

Premiers pas avec GDB

- Compiler avec l'option -g pour inclure les symboles de débogage (dans un makefile, mettre en général `CFLAGS=-g`)
- lancer le programme via gdb : `gdb ./a.out`

```
[sylvain@pyramides sources]$ gdb ./a.out
(gdb) list 1,12
1       #include <stdio.h>
2
3       int main( int argc, char *argv[] )
4       {
5         int x, y;
6
7         printf("Bonjour!\n");
8         x=2;
9         y=3;
10        printf("Le produit x * y vaut %d \n", x * y);
11        return 0;
12      }
(gdb)
```

une session GDB simple

```
(gdb) run
Starting program: /home/sylvain/enseignement/cours_debug/sources/a.out
Bonjour!
Le produit de x et y vaut 6

Program exited normally.
(gdb) break 9
Breakpoint 1 at 0x80483e3: file exemple1.c, line 9.
(gdb) run
Starting program: /home/sylvain/enseignement/cours_debug/sources/a.out
Bonjour!

Breakpoint 1, main (argc=1, argv=0xbff05f74) at exemple1.c:9
9         y=3;
(gdb) print x
$1 = 2
(gdb) set variable x = 12
(gdb) print x
$3 = 12
(gdb) next
10        printf("Le produit de x et y vaut %d \n", x * y);
(gdb) next
Le produit de x et y vaut 36
11        return 0;
(gdb)
```

Les commandes utiles de GDB

- `run` *`command-line-arguments`* Exécute le programme avec les arguments spécifiés
- `break` *`place`* Place un point d'arrêt au nom de fonction indiqué ou au numéro de ligne
- `help` *`command`*
- `step` Exécute l'instruction de la ligne courante et s'arrête à la suivante
- `next` Idem, mais exécute la fonction si c'est un appel de fonction
- `finish` Poursuit l'exécution jusqu'à la fin de la fonction
- `Continue` Poursuit l'exécution jusqu'à un point d'arrêt
- `print` *`E`* Imprime la valeur de la variable ou de l'expression
- `quit` Quitte gdb

Table des matières

DDD

DDD (Data Display Debugger) est une interface qui appelle un compilateur de bas niveau GDB, DBX et idb.
Il permet de visualiser simultanément :

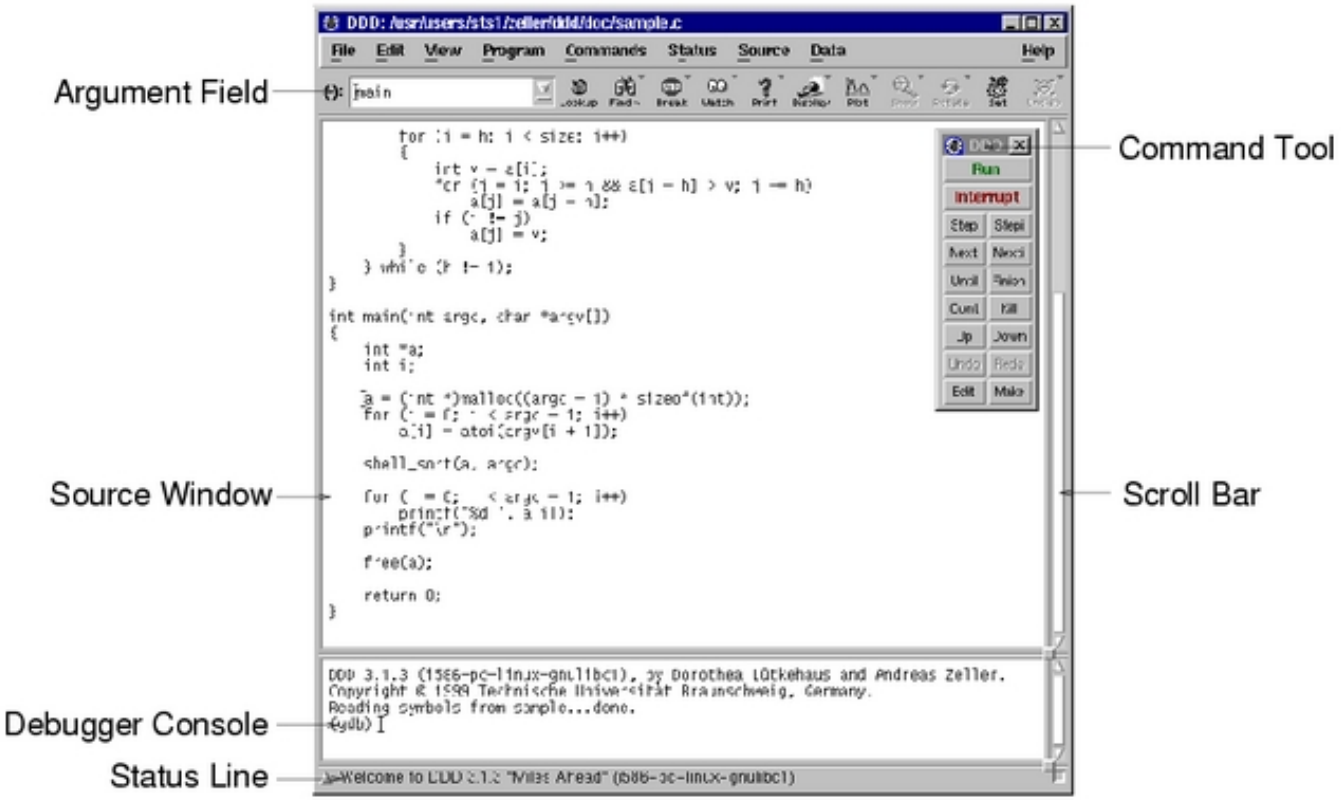
- le code source
- l'état de la mémoire et les structures de données sous forme graphique

Choix du débogueur de bas niveau

Si vous lancez `ddd` sans argument, vous utiliserez par defaut le débogueur **gdb**.
Pour lancer DDD avec idb (debuggeur Intel), il faut utiliser la commande :

```
ddd -debugger "/full/path/to/idb -gdb"
```

Présentation de l'interface de DDD



Initial DDD Window

Premières manipulations avec DDD

Dans le shell :

- Compilation : `gcc -g test.c`
- Exécuter ddd avec l'exécutable en argument : `ddd ./a.out`

Une fois DDD lancé :

- Pour placer un point d'arrêt : Double clic devant la ligne
- Pour exécuter le programme : **Program => Run**
- Pour visualiser une variable : Placer la souris sur la variable
- Pour afficher une variable : Double clic sur le nom de variable
- Pour fixer une valeur : Sélectionner la variable puis **set**

Visualiser un tableau alloué statiquement (dans la pile)

En Fortran :

```
integer x(10)
```

En C :

```
int x[10]
```

Un simple double clic sur la variable affiche les 10 éléments du tableau.

Visualiser un tableau alloué dynamiquement (dans le tas)

En C :

```
int * dyn;
dyn = (int*) calloc (10,sizeof(int));
```

`Dyn` est un pointeur :

`graph display *dyn` affiche la valeur du premier élément

`graph display *dyn@10` affiche les 10 éléments

En fortran 90 :

```
INTEGER, DIMENSION(:), ALLOCATABLE ::  a
ALLOCATE( a(10) )
```

Avec ifort puis idb (debugueur intel), `display a` affiche correctement le tableau. Par contre gfortran+gdb ne permettent pas de le visualiser actuellement.

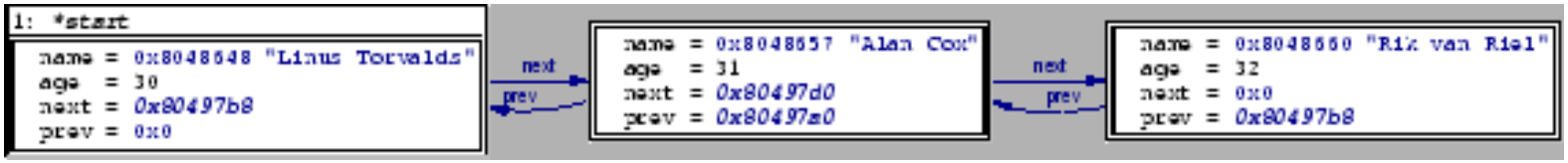
Visualiser une liste chaînée (Programme : liste.c)

```
int main() {
  typedef struct person_struct {
    char* name;
    int age;
    struct person_struct *next;
    struct person_struct *prev;
  } person_t;

  person_t *start, *pers, *temp;
  char *names[] = {"Linus Torvalds", "Alan Cox", "Rik van Riel"};
  int ages[] = {30, 31, 32};
  int count;  /* Temporary counter */

  start = (person_t*)malloc(sizeof(person_t));
  start->name = names[0];
  start->age = ages[0];
  start->prev = NULL;
  start->next = NULL;
  pers = start;

  for (count=1; count < 3; count++) {
    temp = (person_t*)malloc(sizeof(person_t));
    temp->name = names[count];
    temp->age = ages[count];
    pers->next = temp;
    temp->prev = pers;
    pers = temp;
  }
  temp->next = NULL;
  return 0;
}
```

Liste chaînée

Exécutons ce programme dans le débogueur :

- Plaçons un point d'arrêt avant la boucle `for`.
- Visualisons la variable `start`
- Continuons dans la boucle avec **next**
- Continuons à visualiser la liste chaînée en cliquant sur les adresses `next`



Fonctions avancées

Voici d'autres fonctions de DDD

- Visualiser la pile des appels des fonctions : **Status** puis **Backtrace**
- Tracer des graphiques avec les données : Sélectionner une variable tableau puis cliquer **plot**
- Points d'arrêt conditionnels : Sur un point d'arrêt, clic droit **propriété**. La syntaxe est celle des tests en C

TP - Un peu de pratique

Observez le programme `dddbug.c`. Ce programme prend un nombre indéfini d'arguments numériques et les affiche triés par ordre croissant. Cependant ce programme est buggé...

- Essayez de mettre en évidence le problème (indication : essayez des grands nombres)
- Déroulez le programme dans DDD pour comprendre ce qui se passe

Pour aller plus loin - Analyse d'un buffer overflow

Le dépassement de tampon est un bug très classique pouvant avoir des conséquences graves pour la sécurité du système en permettant d'exécuter du code avec les droits de l'utilisateur.

La pile est une partie de la mémoire disponible pour stocker les variables locales pour un programme.
C'est une file LIFO où s'empilent les variables. Si une variable déborde sur la pile, il est alors possible de modifier d'autres variables ou l'adresse de retour d'une fonction.

Stack buffer overflow

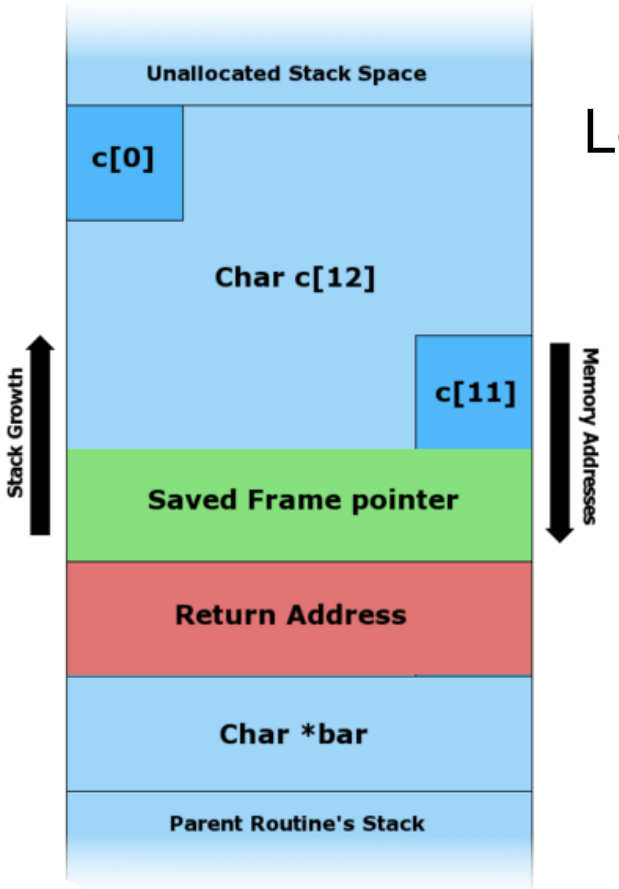
Voici une example de code apparemment anodin mais présentant un danger d'overflow :

```
void foo (char *bar)
{
   char  c[12];
   strcpy(c, bar);  // no bounds checking...
}

int main (int argc, char **argv)
{
   foo(argv[1]);
}
```

Remarquez que l'on ne vérifie pas la taille de la chaîne prise en argument du programme.

Stack buffer overflo - le principe

Voici la structure de la pile lors de l'exécution de la fonction `foo()` :

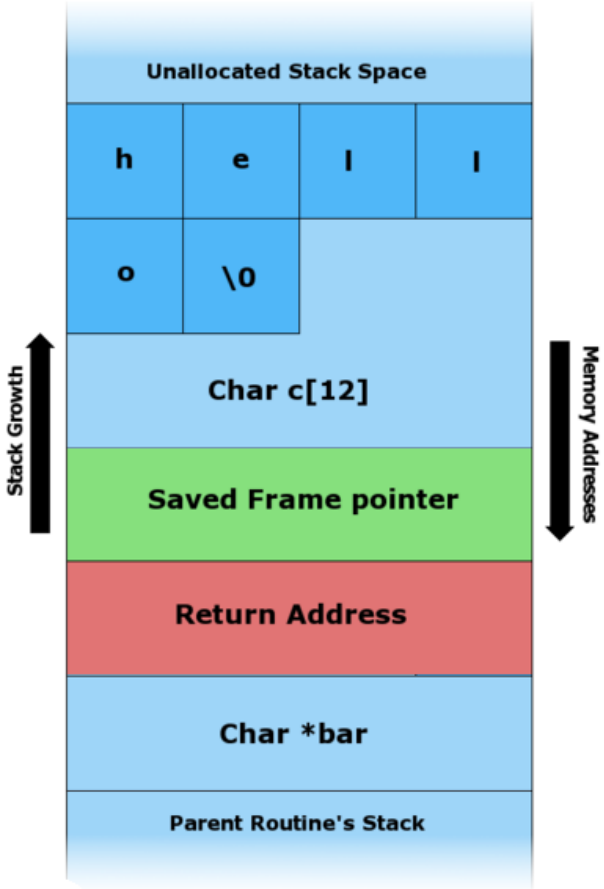


Le système place sur la pile :

- d'abord les arguments de la fonction (ici un pointeur)
- puis l'adresse de retour vers la fonction appelante
- puis l'adresse du début précédant de la trame (=la portion de pile en cours)
- puis les variables locales

Stack buffer overflow - le principe

Essayons d'exécuter le programme avec différents arguments



- Si la chaîne passée en argument est courte (jusqu'à 11 caratères), le programme s'exécute sans erreur
- Si on augmente la longueur de la chaîne on obtient le message `segmentation fault`. La chaîne copiée par `strcpy` peut alors écraser d'éventuelles variables, les arguments de la fonction, jusqu'à l'adresse de retour.

Stack buffer overflow - Avec DDD

Exécutons le programme `overflow.c` dans DDD.

- Placez un point d'arrêt sur le `printf` de la fonction
- Visualisez les variables `a`, `c` et `bar` pour 11 puis 12 puis 20 fois la lettre 'A' passées en argument
- Observez l'état du registre `eip` avec **Status/registers** en sortant de la fonction. Ce registre contient l'adresse de retour de la fonction récupérée sur la pile.
- Il est possible de visualiser l'état de la mémoire avec **data/memory** en choisissant par exemple 40 octets à partir de `*c`

Table des matières

Une autre approche : Valgrind

Valgrind est un ensemble d'outils pour le débogage de programmes.
Le module le plus important est **Memcheck**.
Valgrind exécute le programme dans une machine virtuelle pour en analyser le fonctionnement en cours d'exécution.
Memcheck contrôle en particulier :

- que l'on n'utilise pas de valeurs ou de pointeurs non initialisés
- que l'on ne lit pas de zones mémoires libérées
- que l'on ne lit pas de zones mémoires en dehors de ce que l'on a alloué
- que l'on n'oublie pas de libérer la mémoire allouée.

Exemple d'utilisation - Détecter une fuite de mémoire

- Compilez le programme suivant avec l'option -g

```
#include <stdlib.h>
int main()
{
    char *x = malloc(100); /* or, in C++, "char *x = new char[100] */
    return 0;
}
```

- Exécutez le programme dans valgrind

```
[sylvain@pyramides sources]$ valgrind --tool=memcheck ./exemple
==2330== 100 bytes in 1 blocks are definitely lost in loss record 1 of 1
==2330==    at 0x1B900DD0: malloc (vg_replace_malloc.c:131)
==2330==    by 0x804840F: main (example1.c:5)
```

- Valgrind a détecté un block de 100 octets non libérés

Exemple d'utilisation - Accés en dehors d'un tableau

- Compilez le programme suivant avec l'option -g

```
#include <stdlib.h>

int main()
{
    char *x = malloc(10);
    x[10] = 'a';
    return 0;
}
```

- Exécutez le programme dans valgrind ... Commentez
- Attention la même erreur avec un tableau statique (alloué dans la pile) ne sera pas détectée!

Exemple d'utilisation - Utilisation de variables non initialisées

- Compilez le programme suivant avec l'option -g

```
#include <stdio.h>

int main()
{
    int x;
    if(x == 0)
    {
        printf("X is zero");
    }
return 0;
}
```

- Exécutez le programme dans valgrind ... Commentez

Références

- Linux kernel coding style `http://kerneltrap.org/files/Jeremy/CodingStyle.txt`
- ROBOdoc `http://www.xs4all.nl/~rfsber/Robo/robodoc.html`
- GDB manual : `http://sourceware.org/gdb/documentation/`
- DDD manual : `http://www.gnu.org/manual/ddd/html_mono/ddd.html`
- Un tutorial Totalview : `http://www.llnl.gov/computing/tutorials/totalview/`
- Buffer overflow sur wikipédia (attention à l'erreur dans le diagramme de la pile) : `http://en.wikipedia.org/wiki/Stack_buffer_overflow`