Uso de la herramienta VALGRIND

Metodología de la programación, doble grado, 2013-2014

Contenido:

1 ¿Qué es?

Valgrind es una plataforma de análisis del código. Contiene un conjunto de herramientas que permiten detectar problemas de memoria y también obtener datos detallados de la forma de funcionamiento (rendimiento) de un programa. Es una herramienta de libre distribución que puede obtenerse en: `valgrind.org`. ¡No está disponible para Windows! Algunas de las herramientas que incorpora son:

- **memcheck**: detecta errores en el uso de la memoria dinámica
- **cachegrind**: permite mejorar la rapidez de ejecución del código
- **callgrind**: da información sobre las llamadas a métodos producidas por el código en ejecución
- **massif**: ayuda a reducir la cantidad de memoria usada por el programa.

Nosotros trabajaremos básicamente con la opción de chequeo de problemas de memoria. Esta es la herramienta de uso por defecto. El uso de las demás se indica mediante la opción **tool**. Por ejemplo, para obtener información sobre las llamadas a funciones y métodos mediante **callgrind** haríamos:

```
valgrind --tool=callgrind .........
```

Quizás la herramienta más necesaria sea la de chequeo de memoria (por eso es la opción por defecto). La herramienta **memchek** presenta varias opciones de uso (se consideran aquí únicamente las más habituales):

- **leak-check**: indica al programa que muestre las fugas de memoria al finalizar la ejecución del programa. Los posibles valores para este argumento son **no**, **summary**, **yes** y **full**
- **undef-value-errors**: controla si se examinan los errores debidos a variables no inicializadas (sus posibles valores son **no** y **yes**, siendo este último el valor por defecto)
- **track-origins**: indica si se controla el origen de los valores no inicializados (con **no** y **yes** como posibles valores, siendo el primero el valor por defecto). El valor de este argumento debe estar en concordancia con el del argumento anterior (no tiene sentido indicar que no se desea información sobre valores no inicializados e indicar aquí que se controle el origen de los mismos)

Una forma habitual de lanzar la ejecución de esta herramienta es la siguiente (observad que el nombre del programa y sus posibles argumentos van al final de la línea):

```
valgrind --leak-check=full --track-origins=yes ./programa
```

Esta forma de ejecución ofrece información detallada sobre los posibles problemas en el uso de la memoria dinámica requerida por el programa. Iremos considerando algunos ejemplos para ver la salida obtenida en varios escenarios. Es importante tener en cuenta que es preciso compilar los programas con la opción **-g** para que se incluya información de depuración en el ejecutable.

2 Chequeo de memoria: algunos problemas frecuentes

Se consideran a continuación algunos ejemplos de código con problemas usuales en código con gestión dinámica de memoria. Se analizan para ver qué mensajes de aviso nos muestra esta herramienta en cada caso.

2.1 Uso de memoria no inicializada

Imaginemos que el siguiente código se encuentra en un archivo llamado **ejemplo1.cpp**.

```
1 #include <iostream>
2 using namespace std;
3
4 int main(){
5    int array[5];
6    cout << array[3] << endl;
7 }
```

Compilamos mediante la sentencia:

```
g++ -g -o ejemplo1 ejemplo1.cpp
```

Si ejecutamos ahora **valgrind** como hemos visto antes:

```
valgrind --leak-check=full --track-origins=yes ./ejemplo1
```

se obtiene un informa bastante detallado de la forma en que el programa usa la memoria dinámica. Parte del informe generado es:

```
full --track-origins=yes ./ejemplo1
==13420== Memcheck, a memory error detector
==13420== Copyright (C) 2002-2011, and GNU GPL'd, by Julian Seward et al.
==13420== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
==13420== Command: ./ejemplo1
==13420==
==13420== Conditional jump or move depends on uninitialised value(s)
==13420==    at 0x4EC4F16: std::ostreambuf_iterator<char, std::char_traits<char> > std::num_put<char, std::o
==13420==    by 0x4EC514C: std::num_put<char, std::ostreambuf_iterator<char, std::char_traits<char> > >::do_
==13420==    by 0x4EC8015: std::ostream& std::ostream::_M_insert<long>(long) (in /usr/lib/x86_64-linux-gnu/l
==13420==    by 0x400922: main (ejemplo1.cpp:6)
==13420==  Uninitialised value was created by a stack allocation
==13420==    at 0x40090C: main (ejemplo1.cpp:4)
==13420==
==13420== Use of uninitialised value of size 8
==13420==    at 0x4EBA343: ??? (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.17)
==13420==    by 0x4EC4F37: std::ostreambuf_iterator<char, std::char_traits<char> > std::num_put<char, std::o
.................................................
```

Si nos fijamos en los mensajes que aparecen en el informe anterior (**Conditional jump or move depends on uninitialised values - use of uninitialised value of size 8**) se alude a que los valores del array no han sido inicializados y que se pretende usar el contenido de una posición no inicializada. Si arreglamos el código de forma conveniente y ejecutamos de nuevo, veremos que desaparecen los mensajes de error:

```
1 #include <iostream>
2 using namespace std;
3
4 int main(){
5    int array[5]={1,2,3,4,5};
6    cout << array[3] << endl;
7 }
```

valgrind El informe de que todo ha ido bien es el siguiente:

```
==13504== Memcheck, a memory error detector
==13504== Copyright (C) 2002-2011, and GNU GPL'd, by Julian Seward et al.
==13504== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
==13504== Command: ./ejemplo1-ok
==13504==
==13504==
==13504== HEAP SUMMARY:
==13504==     in use at exit: 0 bytes in 0 blocks
==13504==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==13504==
==13504== All heap blocks were freed -- no leaks are possible
==13504==
==13504== For counts of detected and suppressed errors, rerun with: -v
==13504== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 2 from 2)
```

No debemos hacer caso a la indicación final de usar la opción **-v** (modo **verbose**). Si se usa se generaría una salida mucho más extensa que nos informa de errores de la propia librería de **valgrind** o de las librerías aportadas por C++ (lo que no nos interesa, ya que no tenemos posibilidad de reparar sus problemas).

2.2 Lectura y/o escritura en memoria liberada

Supongamos ahora que el código analizado es el siguiente:

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

using namespace std;

int main(void){
    // Se reserva espacio para p
    char *p = new char;

    // Se da valor
    *p = 'a';

    // Se copia el caracter en c
    char c = *p;

    // Se muestra
    cout << "Caracter c: " << c;

    // Se libera el espacio
    delete p;

    // Se copia el contenido de p (YA LIBERADO) en c
    c = *p;
    return 0;
}
```

El análisis de este código ofrece el siguiente informe valgrind

```
==2408== Memcheck, a memory errvalgrindor detector
==2408== Copyright (C) 2002-2012, and GNU GPL'd, by Julian Seward et al.
==2408== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
==2408== Command: ./usoMemoriaLiberada
==2408==
==2408== Invalid read of size 1
==2408==    at 0x4008E2: main (usoMemoriaLiberada.cpp:24)
==2408==  Address 0x5a1a040 is 0 bytes inside a block of size 1 free'd
==2408==    at 0x4C2BADC: operator delete(void*) (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==2408==    by 0x4008DD: main (usoMemoriaLiberada.cpp:21)
==2408==
==2408==
==2408== HEAP SUMMARY:
==2408==     in use at exit: 0 bytes in 0 blocks
==2408==   total heap usage: 1 allocs, 1 frees, 1 bytes allocated
==2408==
==2408== All heap blocks were freed -- no leaks are possible
==2408==
==2408== For counts of detected and suppressed errors, rerun with: -v
==2408== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 2 from 2)
```

El informe indica explícitamente la línea del código en que se produce el error (línea 21): lectura inválida (sobre memoria ya liberada). En esta línea se muestra que se ha hecho la liberación sobre el puntero **p**.

2.3 Sobrepasar los límites de un array en operación de lectura

Veremos ahora el mensaje obtenido cuando se sobrepasan los límites de un array:

```
1 #include<iostream>
2 using namespace std;
3
4 int main(){
5    int *array=new int[5];
6    cout << array[10] << endl;
7 }
```

Entre los mensajes de error aparece ahora el siguiente texto (parte del informe completo generado):

```
==4474== Memcheck, a memory error detector
==4474== Copyright (C) 2002-2012, and GNU GPL'd, by Julian Seward et al.
==4474== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
==4474== Command: ./ejemplo2
==4474== Parent PID: 1529
==4474==
==4474== Invalid read of size 4
==4474==    at 0x40088B: main (ejemplo2.cpp:6)
==4474==  Address 0x5a1a068 is 20 bytes after a block of size 20 alloc'd
==4474==    at 0x4C2AFE7: operator new[](unsigned long) (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux
==4474==    by 0x40087E: main (ejemplo2.cpp:5)
==4474==
==4474==
==4474== HEAP SUMMARY:
==4474==     in use at exit: 20 bytes in 1 blocks
==4474==   total heap usage: 1 allocs, 0 frees, 20 bytes allocated
==4474==
==4474== 20 bytes in 1 blocks are definitely lost in loss record 1 of 1
==4474==    at 0x4C2AFE7: operator new[](unsigned long) (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux
==4474==    by 0x40087E: main (ejemplo2.cpp:5)
==4474==
==4474== LEAK SUMMARY:
==4474==    definitely lost: 20 bytes in 1 blocks
==4474==    indirectly lost: 0 bytes in 0 blocks
==4474==      possibly lost: 0 bytes in 0 blocks
==4474==    still reachable: 0 bytes in 0 blocks
==4474==         suppressed: 0 bytes in 0 blocks
==4474==
==4474== For counts of detected and suppressed errors, rerun with: -v
==4474== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 2 from 2)
```

Observad el error: (**Invalid read of size 4** ocurrido en relación al espacio de memoria reservado en la línea 5).

2.4 Sobrepasar los límites de un array en operación de escritura

También es frecuente exceder los límites del array para escribir en una posición que ya no le pertenece (alta probabilidad de generación de **core**):

```cpp
#include<iostream>
using namespace std;

int main(){
   int *array=new int[5];
   for(int i=0; i <= 5; i++){
      array[i]=i;
   }
}
```

Además del problema mencionado con anterioridad, en este programa no se libera el espacio reservado al finalizar. La información ofrecida por **valgrind** ayudará a solucionar todos los problemas mencionados:

```
==14144== Memcheck, a memory error detector
==14144== Copyright (C) 2002-2011, and GNU GPL'd, by Julian Seward et al.
==14144== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
==14144== Command: ./ejemplo3
==14144==
==14144== Invalid write of size 4
==14144==    at 0x400732: main (ejemplo3.cpp:7)
==14144==  Address 0x5a06054 is 0 bytes after a block of size 20 alloc'd
==14144==    at 0x4C2AAA4: operator new[](unsigned long) (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linu
==14144==    by 0x40070D: main (ejemplo3.cpp:5)
==14144==
==14144==
==14144== HEAP SUMMARY:
==14144==     in use at exit: 20 bytes in 1 blocks
==14144==   total heap usage: 1 allocs, 0 frees, 20 bytes allocated
==14144==
==14144== 20 bytes in 1 blocks are definitely lost in loss record 1 of 1
==14144==    at 0x4C2AAA4: operator new[](unsigned long) (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linu
==14144==    by 0x40070D: main (ejemplo3.cpp:5)
==14144==
==14144== LEAK SUMMARY:
==14144==    definitely lost: 20 bytes in 1 blocks
==14144==    indirectly lost: 0 bytes in 0 blocks
==14144==      possibly lost: 0 bytes in 0 blocks
==14144==    still reachable: 0 bytes in 0 blocks
==14144==         suppressed: 0 bytes in 0 blocks
==14144==
==14144== For counts of detected and suppressed errors, rerun with: -v
==14144== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 2 from 2)
```

Observad los dos mensajes de error indicados: escritura inválida de tamaño 4 (tamaño asociado al entero) y en el resumen de memoria usada (leak summary) se indica que hay memoria perdida (20 bytes formando parte de un bloque: 5×4 bytes). Con esta información es fácil arreglar estos problemas:

```cpp
#include<iostream>
using namespace std;

int main(){
   int *array=new int[5];
   for(int i=0; i < 5; i++){
      array[i]=i;
   }
```

```
9     delete [] array;
10 }
```

De esta forma desaparecen todos los mensajes de error previos:

```
==14218== Memcheck, a memory error detector
==14218== Copyright (C) 2002-2011, and GNU GPL'd, by Julian Seward et al.
==14218== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
==14218== Command: ./ejemplo4
==14218==
==14218==
==14218== HEAP SUMMARY:
==14218==     in use at exit: 0 bytes in 0 blocks
==14218==   total heap usage: 1 allocs, 1 frees, 20 bytes allocated
==14218==
==14218== All heap blocks were freed -- no leaks are possible
==14218==
==14218== For counts of detected and suppressed errors, rerun with: -v
==14218== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 2 from 2)
```

2.5 Problemas con delete sobre arrays

Otro problema habitual al liberar el espacio de memoria usado por un array suele consistir en olvidar el uso de los corchetes. Esto genera un problema de uso de memoria, ya que no se indica que debe eliminarse un array. El código y los mensajes correspondientes de **valgrind** aparecen a continuación:

```
1  #include<iostream>
2  using namespace std;
3
4  int main(){
5     int *array=new int[5];
6     for(int i=0; i < 5; i++){
7        array[i]=i;
8     }
9     delete array;
10 }
```

```
==14364== Memcheck, a memory error detector
==14364== Copyright (C) 2002-2011, and GNU GPL'd, by Julian Seward et al.
==14364== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
==14364== Command: ./ejemplo5
==14364==
==14364== Mismatched free() / delete / delete []
==14364==    at 0x4C2A44B: operator delete(void*) (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==14364==    by 0x40078E: main (ejemplo5.cpp:9)
==14364==  Address 0x5a06040 is 0 bytes inside a block of size 20 alloc'd
==14364==    at 0x4C2AAA4: operator new[](unsigned long) (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linu
==14364==    by 0x40074D: main (ejemplo5.cpp:5)
==14364==
==14364==
==14364== HEAP SUMMARY:
==14364==     in use at exit: 0 bytes in 0 blocks
==14364==   total heap usage: 1 allocs, 1 frees, 20 bytes allocated
==14364==
==14364== All heap blocks were freed -- no leaks are possible
==14364==
==14364== For counts of detected and suppressed errors, rerun with: -v
==14364== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 2 from 2)
```

El mensaje relevante aquí es **Mismatched free() / delete / delete []**. Indica que no hizo la liberación de espacio (reservado en la línea 5) de forma correcta.

3 Análisis de rendimiento: callgrind

El análisis de rendimiento consiste en la obtención de medidas sobre los recursos consumidos por un programa durante su tiempo de ejecución: qué métodos son los que se han ejecutado y qué tiempo de ejecución se ha gastado en cada uno de ellos. Esto permite detectar cuáles son los métodos más críticos y cuyo funcionamiento debe ser mejorado. El análisis y visualización de resultados se facilita si se instala además la herramienta kcachegrind.

Imaginemos queremos depurar el rendimiento de la práctica del viajante de comercio. La forma de recolectar información sobre su rendimiento, mediante **valgrind**, es la siguiente (asumiendo que el archivo **berlin52.tsp** se encuentra en el directorio básico de la práctica):

```
valgrind --tool=callgrind ./bin/tsp berlin52.tsp
```

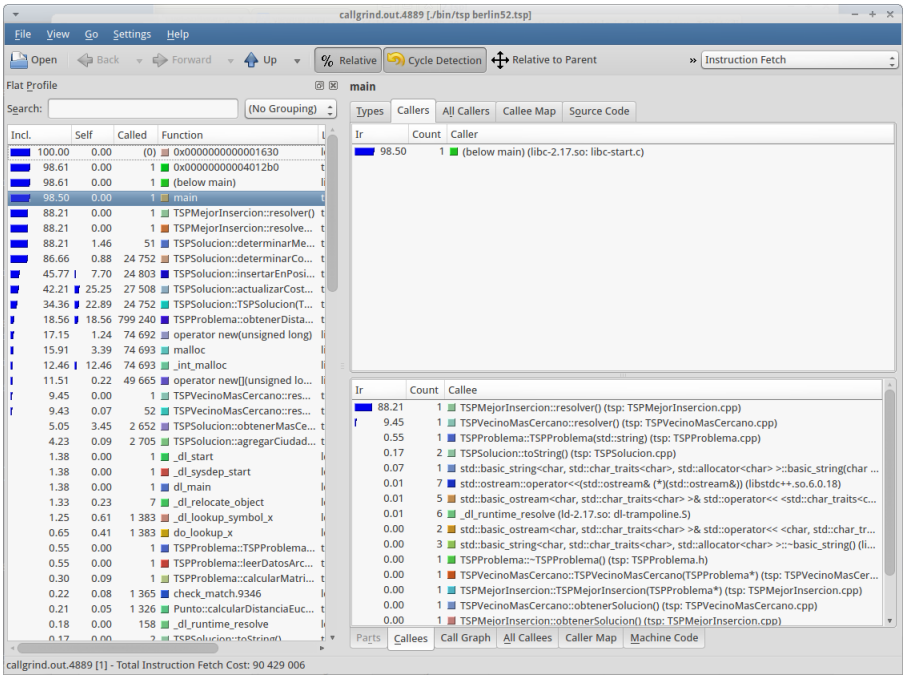
Finaliza la ejecución del programa (si todo va bien) y toda la información recogida sobre el rendimiento del sistema se almacena en un archivo llamado

```
calgrind.out.XXXX
```

donde los cuatro últimos caracteres son números que permiten separar archivos generados en ejecuciones diferentes. La mejor forma de visualizar los resultados es mediante la citada herramienta **kcachegrind**. Su ejecución se produce de la forma siguiente:

```
kcachegrind calgrind.out.4889
```

donde 4889 son los 4 dígitos generados para el archivo de datos de la ejecución de interés. Esto hace aparecer la siguiente ventana:



En la ventana de la izquierda aparece desglosada la lista general de llamadas producidas junto con una estimación del tiempo total de ejecución usado por cada método (función). Se aprecia que el método **resolver()** de la clase **TSPMejorInsercion** ha consumido el 88% del tiempo de ejecución. También puede visualizarse un árbol con las llamadas realizadas: