Guion de prácticas 5

Memoria Dinámica

Abril de 2016

DECSAI

Metodología de la Programación

Curso 2015/2016

Índice

1. Definición del problema

La implementación de la clase Imagen utilizada hasta ahora, almacena los pixeles en un vector cuyo tamaño se fija en tiempo de compilación. En la definición de la clase tenemos

```
typedef unsigned char byte;

class Imagen{
  private:
  //número máximo de píxeles que podemos almacenar
  static const int MAXPIXELS = 1000000;
  byte datos[MAXPIXELS];
  int nfilas;
  int ncolumnas;
  .......
}
```

Como consecuencia de esto, cada vez que se instancia un objeto de la clase Imagen, se ”crea” un vector de tamaño `MAXPIXELS` aunque la imagen actual pueda tener unos pocos cientos.

Una manera de evitar este problema de desperdicio de memoria es ”solicitar” o ”pedir”, en tiempo de ejecución, solamente la memoria que se va a necesitar (conociendo a priori el tamaño de la imagen), utilizarla y luego ”liberarla”.

Por tanto en esta práctica haremos uso de los conceptos básicos de memoria dinámica a partir de la clase Imagen utilizada en guiones anteriores.

2. Objetivos

El desarrollo de esta práctica pretende servir a los siguientes objetivos:

- Repasar conceptos básicos memoria dinámica.
- Modificar la clase Imagen de prácticas anteriores para incluir gestión de memoria dinámica.
- Familiarizarse con el uso de celdas enlazadas en memoria dinámica.

3. Descripción

En esta práctica realizaremos dos tareas:

1. Extender la clase Imagen para que los datos se almacenen en un vector dinámico.

2. Extender la funcionalidad de `arteASCII` para cargar las cadenas contenidas en el fichero `grises.txt` en una lista enlazada de celdas, cada una de las cuales contendrá una cadena de caracteres.

4. Memoria dinámica en la clase Imagen

En primer lugar, debe hacer una copia de la estructura de directorios del guión anterior (trabajará sobre la copia).
Luego deberá realizar las siguientes tareas.

- Reimplementar la clase Imagen utilizando como estructura interna un vector dinámico.
  - Ahora, el constructor de la clase debe reservar memoria (Repase los apuntes de teoría para implementar la reserva de memoria asociada a un vector). Más concretamente, el constructor sin parámetros debe crear una imagen vacía (0 filas, 0 columnas y sin memoria reservada).
  - El constructor con parámetros debe reservar la memoria para la imagen.
  - El método `void crear(int filas, int columnas)` debe liberar la memoria que tenga la imagen, si la hubiera, antes de reservar memoria de nuevo.
  - Por último tenga en cuenta que el método de lectura debe crear la imagen antes de leer los datos.
- Implementar un método `destruir()` que permita liberar la memoria reservada. Note que al destruir la imagen, además de liberar la memoria, también debe poner el número de filas y columnas a cero. Tenga en cuenta que sólo debe liberar la memoria si tuviera reservada, es decir, destruir una imagen ya destruida o vacía no debe producir ningún efecto. Este método debe llamarse explícitamente al final del programa. Cuando se imparta en teoría el tema de clases, se verá que la manera correcta de realizar esta tarea es mediante la utilización de un método ”destructor”. Por ahora, aplicaremos esta solución de compromiso

5. Implementación con celdas enlazadas

En este mismo guion practicaremos con la implementación de celdas enlazadas en memoria dinámica tal y como se vió en el tema de memoria dinámica ampliando las funciones implementadas en la Práctica 4.

Ejercicio 1 *Reimplementar el Ejercicio 1 de la Práctica 4 (*`arteASCII`*) de la siguiente forma.*

- Utilizar como fichero principal el fichero `arteASCII2.cpp` que se proporciona con esta práctica y que sustituye al antiguo `arteASCII.cpp` de la Práctica 4.
- Implementar en el fichero `listas.cpp` la clase descrita en el fichero `listas.h` para la gestión de listas enlazadas de `string`, en la que cada celda de la lista se utiliza para almacenar una cadena de caracteres. Aunque la implementación de esta estructura de datos debe contener más métodos, sólo se implementarán los siguientes:
  - Implementar el construtor o constructores de la clase.
  - Implementar el método
    `void Lista::destruir();`
    para liberar la memoria reservada.
  - Implementar el método
    `void Lista::insertar(string cad);`
    que inserta una nueva cadena al final de la lista.
  - Implementar el método
    `string Lista::getCelda(int i) const;`
    que devuelve el string de la posición i-ésima de la lista o la cadena vacía en caso de que el valor de i sea erróneo (suponemos la cadena vacía no es un valor permitido en ninguna celda de la lista).
  - Implementar el método
    `int Lista::longitud() const;`
    que devuelve el número de celdas que contiene la lista.
  - En `listas.cpp` aparece implementado un método
    `bool Lista::leerLista(char nombrefichero[]);`
    que construye una lista de celdas enlazadas a partir de la información contenida en un fichero, cuyo nombre se pasa por parámetro y cuya estructura es la misma del fichero `grises.txt` de la Práctica 4.
- En la clase Imagen (fichero `imagen.cpp`) implementar el método

  ```
  bool Imagen::listaAArteASCII(const Lista celdas);
  ```

  que a su vez llamará al método:

  ```
  bool Imagen::aArteASCII(const char grises[],
  char arteASCII[], int maxlong);
  ```

  codificado desde la Práctica 3. La nueva implementación, en vez de utilizar una única cadena de caracteres para codificar la imagen actual a ASCII, trabajará con una lista de celdas de cadenas. El método debe generar, por cada cadena de la lista, un fichero de salida ASCII tal y como se explica en el Ejercicio 1 de la Práctica 4 pero cambiando el nombre del fichero de salida a `ascii1.txt`, `ascii2.txt`, etc. Esta función deberá iterar por cada celda de la lista y ejecutar el método *aArteASCII*.

6. Material a entregar

Cuando esté todo listo y probado debidamente con la herramienta valgrind (ver la siguiente sección) el alumno empaquetará la estructura de directorios anterior en un archivo con el nombre **practica5.zip** y lo entregará en la plataforma **decsai** en el plazo indicado. No deben entregarse archivos objeto (**.o**) ni ejecutables. Para asegurarse de esto último conviene ejecutar **make mrproper** antes de proceder al empaquetado. Los ficheros deberán estar en los directorios adecuados: include para los ficheros .h y src para los ficheros .cpp.

El alumno debe asegurarse de que ejecutando las siguientes órdenes se compila y ejecuta correctamente su proyecto:

```
unzip practica5
cd practica5
make
bin/testimagen
bin/testplano
bin/testarteASCII
bin/arteASCII2
```

Se deberá incluir un informe en pdf donde aparezcan los integrantes de la pareja que hizo el trabajo, las dificultades que hayan tenido en realizar la práctica, y capturas de pantalla mostrando que los programas funcionan correctamente. Este informe se guardará en la carpeta doc. Se comprobará con Valgrind la corrección del código.

7. Chequeo de memoria con Valgrind

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 los errores en el manejo 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.

Nota: 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.

8. Ejemplos de uso de Valgrind

Se consideran a continuación algunos ejemplos de código con problemas usuales con gestión dinámica de memoria. Se analizan para ver qué mensajes de aviso nos muestra esta herramienta en cada caso (marcados en rojo). En concreto, vamos a identificar los siguientes escenarios: Uso de memoria no inicializada, lectura y/o escritura en memoria liberada , sobrepasar los límites de un array en operación de lectura, sobrepasar los límites de un array en operación de escritura, problemas con delete sobre arrays, aviso sobre el uso masivo de memoria no dinámica

8.1. Uso de memoria no inicializada

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

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

int main(){
   int array[5];
   cout << array[3] << endl;
}
```

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:

```
==4630== Memcheck, a memory error detector
==4630== Copyright (C) 2002-2013, and GNU GPL'd, by Julian
==4630== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h
==4630== Command: ./ejemplo1
==4630==
==4630== Conditional jump or move depends on uninitialised
==4630==    at 0x4EBFE9E: std::ostreambuf_iterator<char,
==4630==    by 0x4EC047C: std::num_put<char, std::ostreambu
==4630==    by 0x4ECC21D: std::ostream& std::ostream::_M_in
==4630==    by 0x400843: main (ejemplo1.cpp:6)
==4630==  Uninitialised value was created by a stack allocation
==4630==    at 0x40082D: main (ejemplo1.cpp:4)
==4630==
==4630== Use of uninitialised value of size 8
==4630==    at 0x4EBFD83: ??? (in /usr/lib/x86_64-linux-gnu/
==4630==    by 0x4EBFEC5: std::ostreambuf_iterator<char,
==4630==    by 0x4EC047C: std::num_put<char, std::ostreambuf
==4630==    by 0x4ECC21D: std::ostream& std::ostream::_M_ins
==4630==    by 0x400843: main (ejemplo1.cpp:6)
==4630==  Uninitialised value was created by a stack allocation
==4630==    at 0x40082D: main (ejemplo1.cpp:4)
.................................................
```

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:

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

int main(){
   int array[5]={1,2,3,4,5};
   cout << array[3] << endl;
}
```

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

```
==4654== Memcheck, a memory error detector
==4654== Copyright (C) 2002-2013, and GNU GPL'd, by Julian
==4654== Using Valgrind-3.10.0.SVN and LibVEX; rerun with
==4654== Command: ./ejemplo1-ok
==4654==
4
==4654==
==4654== HEAP SUMMARY:
==4654==     in use at exit: 0 bytes in 0 blocks
==4654==   total heap usage: 0 allocs, 0 frees, 0 bytes alloc
==4654==
==4654== All heap blocks were freed -- no leaks are possible
==4654==
==4654== For counts of detected and suppressed errors,
rerun with: -v
==4654== ERROR SUMMARY: 0 errors from 0 contexts
(suppressed: 0 from 0)
```

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).

8.2. Lectura y/o escritura en memoria liberada

Supongamos ahora que el código analizado es el siguiente:

```
#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

```
==4662== Memcheck, a memory error detector
==4662== Copyright (C) 2002-2013, and GNU GPL'd, by Julian
Seward et al.
==4662== Using Valgrind-3.10.0.SVN and LibVEX; rerun with
-h for copyright info
==4662== Command: usoMemoriaLiberada
==4662==
==4662== Invalid read of size 1
==4662==    at 0x4008E2: main (usoMemoriaLiberada.cpp:24)
==4662==  Address 0x5a1d040 is 0 bytes inside a block
of size 1 free'd
==4662==    at 0x4C2C2BC: operator delete(void*) (in
/usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==4662==    by 0x4008DD: main (usoMemoriaLiberada.cpp:21)
==4662==
Caracter c: a==4662==
==4662== HEAP SUMMARY:
==4662==     in use at exit: 0 bytes in 0 blocks
==4662==   total heap usage: 1 allocs, 1 frees, 1 bytes
allocated
==4662==
==4662== All heap blocks were freed -- no leaks are possible
==4662==
==4662== For counts of detected and suppressed errors,
rerun with: -v
==4662== ERROR SUMMARY: 1 errors from 1 contexts
```

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**.

8.3. Sobrepasar los límites de un array en operación de lectura

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

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

int main(){
   int *array=new int[5];
   cout << array[10] << endl;
}
```

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

```
==4670== Memcheck, a memory error detector
==4670== Copyright (C) 2002-2013, and GNU GPL'd, by Julian
Seward et al.
```

```
==4670== Using Valgrind-3.10.0.SVN and LibVEX; rerun with
-h for copyright info
==4670== Command: ./ejemplo2
==4670==
==4670== Invalid read of size 4
==4670==    at 0x40088B: main (ejemplo2.cpp:6)
==4670==  Address 0x5a1d068 is 20 bytes after a block of
size 20 alloc'd
==4670==    at 0x4C2B800: operator new[](unsigned long) (in
/usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==4670==    by 0x40087E: main (ejemplo2.cpp:5)
==4670==
==4670==
==4670== HEAP SUMMARY:
==4670==     in use at exit: 20 bytes in 1 blocks
==4670==   total heap usage: 1 allocs, 0 frees, 20 bytes
allocated
==4670==
==4670== 20 bytes in 1 blocks are definitely
 lost in loss record 1 of 1
==4670==    at 0x4C2B800: operator new[](unsigned long) (in
/usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==4670==    by 0x40087E: main (ejemplo2.cpp:5)
==4670==
==4670== LEAK SUMMARY:
==4670==    definitely lost: 20 bytes in 1 blocks
==4670==    indirectly lost: 0 bytes in 0 blocks
==4670==      possibly lost: 0 bytes in 0 blocks
==4670==    still reachable: 0 bytes in 0 blocks
==4670==         suppressed: 0 bytes in 0 blocks
==4670==
==4670== For counts of detected and suppressed errors,
rerun with: -v
==4670== ERROR SUMMARY: 2 errors from 2 contexts
```

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

8.4. 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**):

```
#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:

```
==4678== Memcheck, a memory error detector
==4678== Copyright (C) 2002-2013, and GNU GPL'd, by Julian
Seward et al.
==4678== Using Valgrind-3.10.0.SVN and LibVEX; rerun with
-h for copyright info
==4678== Command: ./ejemplo3
==4678==
==4678== Invalid write of size 4
==4678==    at 0x400743: main (ejemplo3.cpp:7)
==4678==  Address 0x5a1d054 is 0 bytes after a block of
size 20 alloc'd
==4678==    at 0x4C2B800: operator new[](unsigned long) (in
/usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==4678==    by 0x40071E: main (ejemplo3.cpp:5)
==4678==
==4678==
==4678== HEAP SUMMARY:
==4678==     in use at exit: 20 bytes in 1 blocks
==4678==   total heap usage: 1 allocs, 0 frees,
20 bytes allocated
==4678==
==4678== 20 bytes in 1 blocks are definitely lost
in loss record 1 of 1
==4678==    at 0x4C2B800: operator new[](unsigned long) (in
/usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==4678==    by 0x40071E: main (ejemplo3.cpp:5)
==4678==
==4678== LEAK SUMMARY:
==4678==    definitely lost: 20 bytes in 1 blocks
==4678==    indirectly lost: 0 bytes in 0 blocks
==4678==      possibly lost: 0 bytes in 0 blocks
==4678==    still reachable: 0 bytes in 0 blocks
==4678==         suppressed: 0 bytes in 0 blocks
==4678==
==4678== For counts of detected and suppressed errors,
rerun with: -v
==4678== ERROR SUMMARY: 2 errors from 2 contexts
```

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:

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

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

```
}
```

De esta forma desaparecen todos los mensajes de error previos:

```
==4683== Memcheck, a memory error detector
==4683== Copyright (C) 2002-2013, and GNU GPL'd, by Julian
Seward et al.
==4683== Using Valgrind-3.10.0.SVN and LibVEX; rerun with
-h for copyright info
==4683== Command: ./ejemplo4
==4683==
==4683==
==4683== HEAP SUMMARY:
==4683==     in use at exit: 0 bytes in 0 blocks
==4683==   total heap usage: 1 allocs, 1 frees,
20 bytes allocated
==4683==
==4683== All heap blocks were freed -- no leaks are possible
==4683==
==4683== For counts of detected and suppressed errors,
rerun with: -v
==4683== ERROR SUMMARY: 0 errors from 0 contexts
```

8.5. 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:

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

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

```
==4686== Memcheck, a memory error detector
==4686== Copyright (C) 2002-2013, and GNU GPL'd, by
Julian Seward et al.
==4686== Using Valgrind-3.10.0.SVN and LibVEX; rerun
with -h for copyright info
==4686== Command: ./ejemplo5
==4686==
valgrind --leak-check=full --track-origins=yes ./ejemplo4
==4686== Mismatched free() / delete / delete []
==4686==    at 0x4C2C2BC: operator delete(void*) (in
/usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==4686==    by 0x4007AA: main (ejemplo5.cpp:9)
==4686==  Address 0x5a1d040 is 0 bytes inside a block of
```

```
size 20 alloc'd
==4686==    at 0x4C2B800: operator new[](unsigned long) (in
/usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==4686==    by 0x40076E: main (ejemplo5.cpp:5)
==4686==
==4686==
==4686== HEAP SUMMARY:
==4686==     in use at exit: 0 bytes in 0 blocks
==4686==   total heap usage: 1 allocs, 1 frees,
20 bytes allocated
==4686==
==4686== All heap blocks were freed -- no leaks are possible
==4686==
==4686== For counts of detected and suppressed errors,
rerun with: -v
==4686== ERROR SUMMARY: 1 errors from 1 contexts
```

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.

8.6. Aviso sobre el uso masivo de memoria no dinámica

Si usamos mucha memoria alojada en vectores no dinámicos, memcheck se quejará y nos dará un buen montón de errores. Por ejemplo, si con el siguiente código

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

int main(){
   int array[1000000]={1,2,3,4,5};
   cout << array[3] << endl;
}
```

ejecutamos:

`valgrind --leak-check=full --track-origins=yes ./ejemplo6`

obtenemos una larguísima salida

```
==4687== Memcheck, a memory error detector
==4687== Copyright (C) 2002-2013, and GNU GPL'd, by Julian
Seward et al.
==4687== Using Valgrind-3.10.0.SVN and LibVEX; rerun
with -h for copyright info
==4687== Command: ./ejemplo6
==4687==
==4687== Warning: client switching stacks?  SP change:
0xffefffd40 --> 0xffec2f438
==4687==          to suppress, use:
--max-stackframe=4000008 or greater
==4687== Invalid write of size 8
==4687==    at 0x40088C: main (ejemplo6.cpp:5)
==4687==  Address 0xffec2f438 is on thread 1's stack
==4687==
==4687== Invalid write of size 8
==4687==    at 0x4C313C7: memset (in /usr/lib/valgrind/vgpreload
```

```
==4687==    by 0x400890: main (ejemplo6.cpp:5)
==4687==  Address 0xffec2f440 is on thread 1's stack
.........
==4687== Warning: client switching stacks?
SP change: 0xffec2f440 --> 0xffefffd40
==4687==          to suppress, use:
--max-stackframe=4000000 or greater
==4687==
==4687== HEAP SUMMARY:
==4687==     in use at exit: 0 bytes in 0 blocks
==4687==   total heap usage: 0 allocs, 0 frees,
0 bytes allocated
==4687==
==4687== All heap blocks were freed -- no leaks are possible
==4687==
==4687== For counts of detected and suppressed errors,
rerun with: -v
==4687== ERROR SUMMARY: 499992 errors from 9 contexts
```

Afortunadamente también nos dice como solucionar este problema: **to suppress, use: –max-stackframe=4000008 or greater**. Por tanto, ejecutando

`valgrind --leak-check=full --track-origins=yes --max-stackframe=4000008 ./ejemplo6`

obtenemos una salida de ejecución correcta

```
==4697== Memcheck, a memory error detector
==4697== Copyright (C) 2002-2013, and GNU GPL'd, by
Julian Seward et al.
==4697== Using Valgrind-3.10.0.SVN and LibVEX; rerun
with -h for copyright info
==4697== Command: ./ejemplo6
==4697==
==4697==
==4697== HEAP SUMMARY:
==4697==     in use at exit: 0 bytes in 0 blocks
==4697==   total heap usage: 0 allocs, 0 frees,
0 bytes allocated
==4697==
==4697== All heap blocks were freed -- no leaks are possible
==4697==
==4697== For counts of detected and suppressed errors,
rerun with: -v
==4697== ERROR SUMMARY: 0 errors from 0 contexts
```