Guion de prácticas

Uso de valgrind para memoria dinámica

Abril de 2018

UNIVERSIDAD DE GRANADA

ETSIIT Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

DECSAI

Metodología de la Programación

Curso 2017/2018

Índice

1. Descripción **5**

2. Chequeo de memoria con Valgrind **5**

3. Ejemplos de uso de Valgrind **6**
- 3.1. Uso de memoria no inicializada 6
- 3.2. Lectura y/o escritura en memoria liberada 9
- 3.3. Sobrepasar los límites de un array en operación de lectura de memoria 11
- 3.4. Sobrepasar los límites de un array en operación de escritura en memoria 13
- 3.5. Problemas con delete sobre arrays 16
- 3.6. Aviso sobre el uso masivo de memoria no dinámica 17

4. Práctica a entregar: gestionar bigramas **19**
- 4.1. Corrección de la práctica 19

5. Apendice 1. Código para la práctica **21**
- 5.1. Bigrama.h 21

1. Descripción

En esta práctica aprenderemos a utilizar la herramienta `valgrind` para diagnosticar varios problemas, entre otros, los problemas de mala gestión de memoria dinámica en un programa.

2. 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 **memcheck** 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 preciso compilar los programas con la opción **-g** para que se incluya información de depuración en el ejecutable.

3. 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, todos ellos relacionados con el acceso a memoria para leer y/o escribir un valor en una variable: uso de memoria no inicializada, lectura y/o escritura en memoria ya liberada, sobrepasar los límites de un array en operación de lectura, sobrepasar los límites de un array en operación de lectura o escritura, problemas con delete sobre arrays, aviso sobre el uso masivo de memoria no inicializada.

3.1. Uso de memoria no inicializada

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

Listing 1: ejemplo1.cpp

```
1 #include <iostream>
2 using namespace std;
3
4 int main(){
5    int array[5];
6    cout << array[3] << endl; // Memoria NO inicializada
7 }
```

Compilamos mediante la sentencia:

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

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

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

se obtiene un informe 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**) 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:

Listing 2: ejemplo1–ok.cpp

```
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; // Memoria SI inicializada
7 }
```

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 es necesario 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).

3.2. Lectura y/o escritura en memoria liberada

Supongamos ahora que el código analizado es el siguiente:

Listing 3: usoMemoriaLiberada.cpp

```
1  #include <cstdio>
2  #include <cstdlib>
3  #include <iostream>
4
5  using namespace std;
6
7  int main(void){
8      // Se reserva espacio para p
9      char *p = new char;
10
11     // Se da valor
12     *p = 'a';
13
14     // Se copia el caracter en c
15     char c = *p;
16
17     // Se muestra
18     cout << "Caracter c: " << c;
19
20     // Se libera el espacio
21     delete p;
22
23     // Se copia el contenido de p (YA LIBERADO) en c
24     c = *p;
25     return 0;
26 }
```

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 24): lectura inválida (sobre memoria ya liberada). En esta línea se muestra que se ha hecho la liberación sobre el puntero **p**.

3.3. Sobrepasar los límites de un array en operación de lectura de memoria

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

Listing 4: ejemplo2.cpp

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

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). Observad también el error relativo a memoria reservada pero no liberada (20 bytes).

3.4. Sobrepasar los límites de un array en operación de escritura en memoria

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

Listing 5: ejemplo3.cpp

```
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; // Escritura en memoria fuera de
8    }              // limite valido cuando i == 5
9 }
```

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 $\times$ 4 bytes). Con esta información es fácil arreglar estos problemas:

Listing 6: ejemplo4.cpp

```
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 }
```

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
```

3.5. 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:

Listing 7: ejemplo5.cpp

```
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; // Liberacion de memoria incorrecta
10 }
```

```
==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.

3.6. 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

Listing 8: ejemplo6.cpp

```
1 #include <iostream>
2 using namespace std;
3
4 int main(){
5    int array[1000000]={1,2,3,4,5};
6    cout << array[3] << endl;
7 }
```

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
```

4. Práctica a entregar: gestionar bigramas

Un bigrama es cualquier secuencia de dos caracteres consecutivos que aparecen en un texto determinado a la cual suele añadirse la frecuencia con la que esta secuencia aparece en el texto. Por ejemplo, el siguiente texto:

"Hola a todas las personas"

contendría los siguientes bigramas, con sus frecuencias asociadas (no se diferencian mayúsculas ni se contemplan los separadores como los espacios en blanco, ni las palabras de una única letra):

{"ho",1}, {"ol",1}, {"la",2}, {"to",1}, {"od",1}, {"da",1}, {"as",3}, {"pe",1}, {"er",1}, {"rs",1}, {"so",1}, {"on",1}

Se pide construir un programa para gestionar diferentes secuencias de bigramas, en el que cada secuencia contiene exactamente los mismos bigramas, aunque puedan estar en distinto orden y con distintas frecuencias. El programa deberá leer dos secuencias de bigramas, precedidas ambas por el número de bigramas que contienen, almacenarlas en memoria dinámica y calcular una nueva secuencia de bigramas en memoria dinámica a partir de las anteriores, la cual contiene los mismos bigramas que las dos secuencias precedentes, y en la que la frecuencia de cada bigrama es la suma de las frecuencias del mismo bigrama en las dos secuencias leídas. El programa deberá mostrar la secuencia resultado en orden ascendente de frecuencias.

Descargue el fichero **OrdenaBigramas_nb.zip**, descomprímalo y ábralo en NetBeans o desde la línea de comandos con su editor favorito. Complete el código y validelo con el fichero de datos **ordenabigramas.txt** que se encuentra, como siempre, en la carpeta **data**/.

```
$$ cat data/ordenabigramas.txt
30
de 822
do 401
co 369
da 288
ca 276
...  más datos ...
```

Complete el código y compruebe tanto los datos de salida como el buen uso de la memoria dinámica.

4.1. Corrección de la práctica

En este caso, se ha incluido un objetivo **test** en el makefile que ejecuta el programa, redirecciona la entrada desde el fichero de datos de validación y le pasa el valgrind. El resultado debería ser el que muestra la Figura 1 tanto si es desde la línea de comandos como si es desde dentro de NetBeans.

```
$$ make tests
valgrind --leak-check=full dist/Debug/GNU-Linux/ordenabigramas < data/ordenabigramas.txt
==15096== Memcheck, a memory error detector
==15096== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==15096== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==15096== Command: dist/Debug/GNU-Linux/ordenabigramas
==15096==
Introduce el número de bigramas del primer vector: Introduce el número de bigramas del
segundo vector:
Contenido de v1
Lista de 30 bigramas:
de-822, do-401, co-369, da-288, ca-276, ci-268, di-203, ha-190, ba-181, ce-166, cu-157,
gu-91, ho-91, go-87, ga-76, bi-70, fu-69, fa-62, hi-59, he-47, be-42, bu-41, fi-39, gi-38,
du-37, fe-34, bo-28, ge-26, hu-23, fo-19,
Contenido de v2
Lista de 30 bigramas:
he-4058, ha-1217, hi-870, ce-805, be-655, ho-615, de-552, do-491, co-461, fo-410, ca-365,
ge-339, di-322, bu-243, ga-227, bo-216, go-203, fi-190, fe-177, gu-137, fu-122, fa-117,
bi-116, cu-116, da-115, gi-96, ba-86, du-75, hu-62, ci-55,
Sumando v1 y v2 en v3 ...

Contenido de v3
Lista de 30 bigramas:
hu-85, du-112, gi-134, fa-179, bi-186, fu-191, fe-211, gu-228, fi-229, bo-244, ba-267,
cu-273, bu-284, go-290, ga-303, ci-323, ge-365, da-403, fo-429, di-525, ca-641, be-697,
ho-706, co-830, do-892, hi-929, ce-971, de-1374, ha-1407, he-4105,
==15096==
==15096== HEAP SUMMARY:
==15096==     in use at exit: 72,704 bytes in 1 blocks
==15096==   total heap usage: 6 allocs, 5 frees, 78,544 bytes allocated
==15096==
==15096== LEAK SUMMARY:
==15096==    definitely lost: 0 bytes in 0 blocks
==15096==    indirectly lost: 0 bytes in 0 blocks
==15096==      possibly lost: 0 bytes in 0 blocks
==15096==    still reachable: 72,704 bytes in 1 blocks
==15096==         suppressed: 0 bytes in 0 blocks
==15096== Reachable blocks (those to which a pointer was found) are not shown.
==15096== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==15096==
==15096== For counts of detected and suppressed errors, rerun with: -v
==15096== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

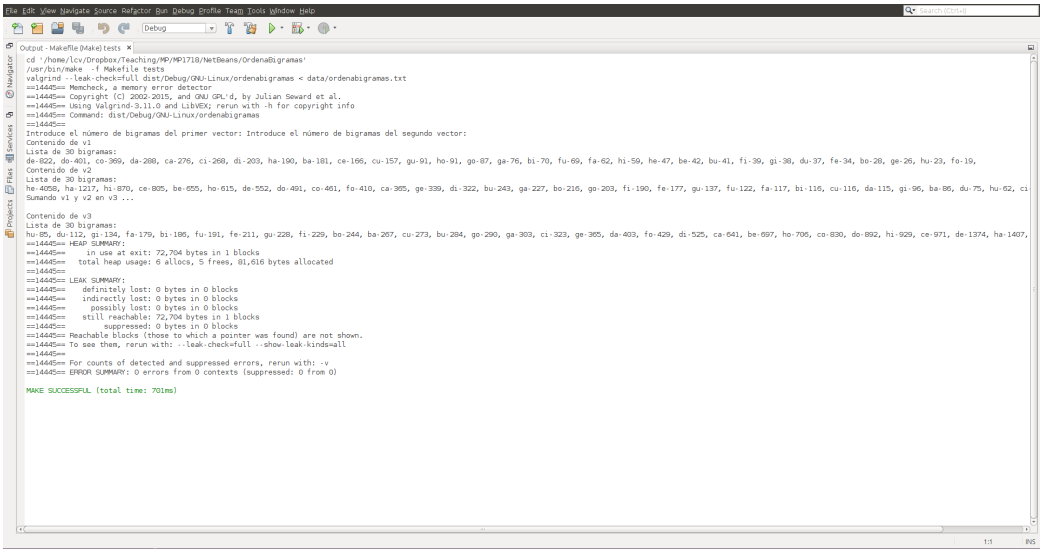

Figura 1: Ejecución del programa, redirección de la entrada de datos y validación con valgrind desde la línea de comandos y desde dentro de Netbeans

5. Apendice 1. Código para la práctica

5.1. Bigrama.h

```
/**
 * @file Bigrama.h
 * @author DECSAI
 */

#ifndef BIGRAMA_H
#define BIGRAMA_H

#include <iostream>
/**
 * @class Bigrama
 * @brief Gestión individual de bigramas: pares de letras consecutivas de un
 * texto y su frecuencia asociada dentro del texto
 */
class Bigrama {
public:
    /// @brief Constructor base
    Bigrama();

    /**
     * @brief Devuelve el bigrama almacenado
     * @return el bigrama
     */
    const char* getBigrama() const;

    /**
     * @brief Devuelve la frecuencia almacenada
     * @return la frecuencia
     */
    int getFrecuencia() const;

    /**
     * @brief Actualiza el bigrama, siempre que sea de la longitud adecuada,
     *  en otro caso, no se actualiza, para evitar desbordamientos del vector
     * @param cadena El nuevo valor del bigrama
     */
    void setBigrama(const char cadena[]);

    /**
     * @brief Actualiza la frecuencia almacenada
     * @param frec La nueva frecuencia
     */
    void setFrecuencia(int frec);


private:
    char _bigrama[3]; /// Bigrama almacenado, incluyendo el '\0'. Ojo a desbordamientos
    int _frecuencia; /// Frecuencia almacenada
};

/**
 * @brief Ordena un vector de bigramas de forma ascendente por frecuencias
 * @param v El vector de bigramas
 * @param n Tamaño de @a v
 */
void ordenaAscFrec(Bigrama *v, int n);

/**
 * @brief Ordena un vector de bigramas de forma ascendente por bigramas
 * @param v El vector de bigramas
 * @param n Tamaño de @a v
 */
void ordenaAscBigr(Bigrama *v, int n);

/**
 * @brief Imprime un vector de bigramas
 * @param v El vector de bigramas
 * @param n Tamaño de @a v
 */
void imprimeBigramas(const Bigrama *v, int n);

/**
 * @brief Suma dos listas de bigramas y devuelve el resultado
 * @param v1 Primer vector de bigramas
 * @param nv1 Tamaño de @a v1
 * @param v2 Segundo vector de bigramas
 * @param nv2 Tamaño de @a v2
 * @param res Vector resultado creado en memoria dinámica
 * @param nres Tamaño de @a res
 * @pre @a v1 y @a v2 deben tener el mismo tamaño y los mismos bigramas aunque las frecuencias sean
     distintas
 */
void sumaBigramas(const Bigrama *v1, int nv1, const Bigrama *v2, int nv2, Bigrama *&res, int & nres);

#endif /* BIGRAMA_H */
```