Grupo D: Berard, Guglieri, Rubin, Sandoval

Informe:

* Muy bien
* Los casos de prueba de la Fig. 5.2 idealmente deberían correrlos con valgrind (como mínimo hacerlo ustedes) para verificar que no tengan pérdidas de memorias en las distintas entradas.

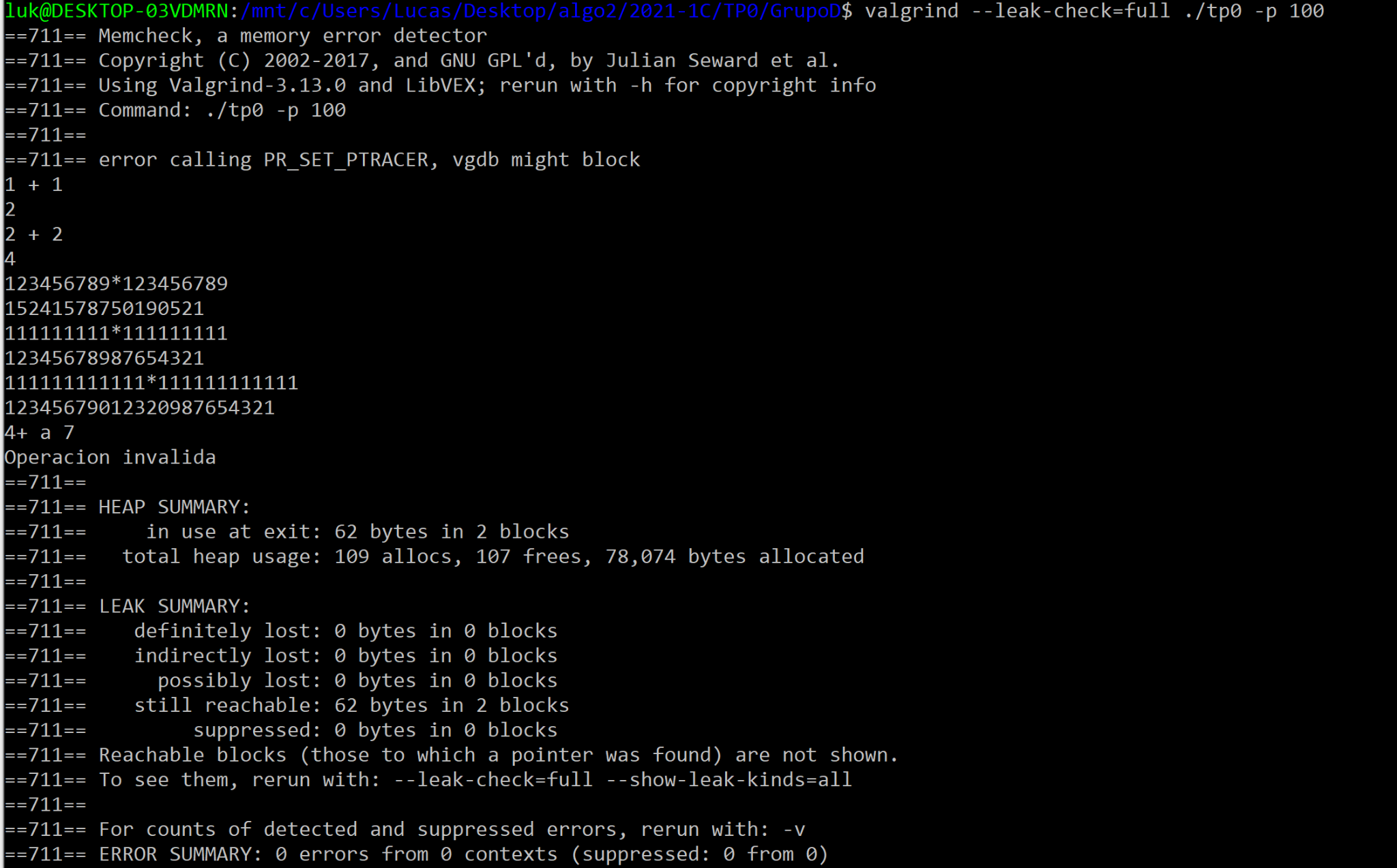
Implementación:

* Corrí con números de precisión grande y funcionó.
* No procesan bien las cuentas del estilo -10 - -15. Aca devuelven -5 cuando deberían estar devolviendo 5. No debería ser difícil de arreglar. Corregirlo para el TP1.
* Dejo captura (al final) de una pérdida de memoria que logre generar (y donde parece estar). Es un bug raro ya que si uno corre simplemente “4+ a 7” anda bien sin perder memoria. El leak se genera luego de correr una operación con números de precisión mayor a 8 únicamente. Es extraño que pase esto. Corregir para el TP1.
* También encontré perdida de memoria con la entrada “asdf + asdf asdf”. Prueben estos casos con valgrind --leak-check-full --show-leak-kinds=all ./tp0 -p 100 para máximos detalles sobre cómo localizar la pérdida.

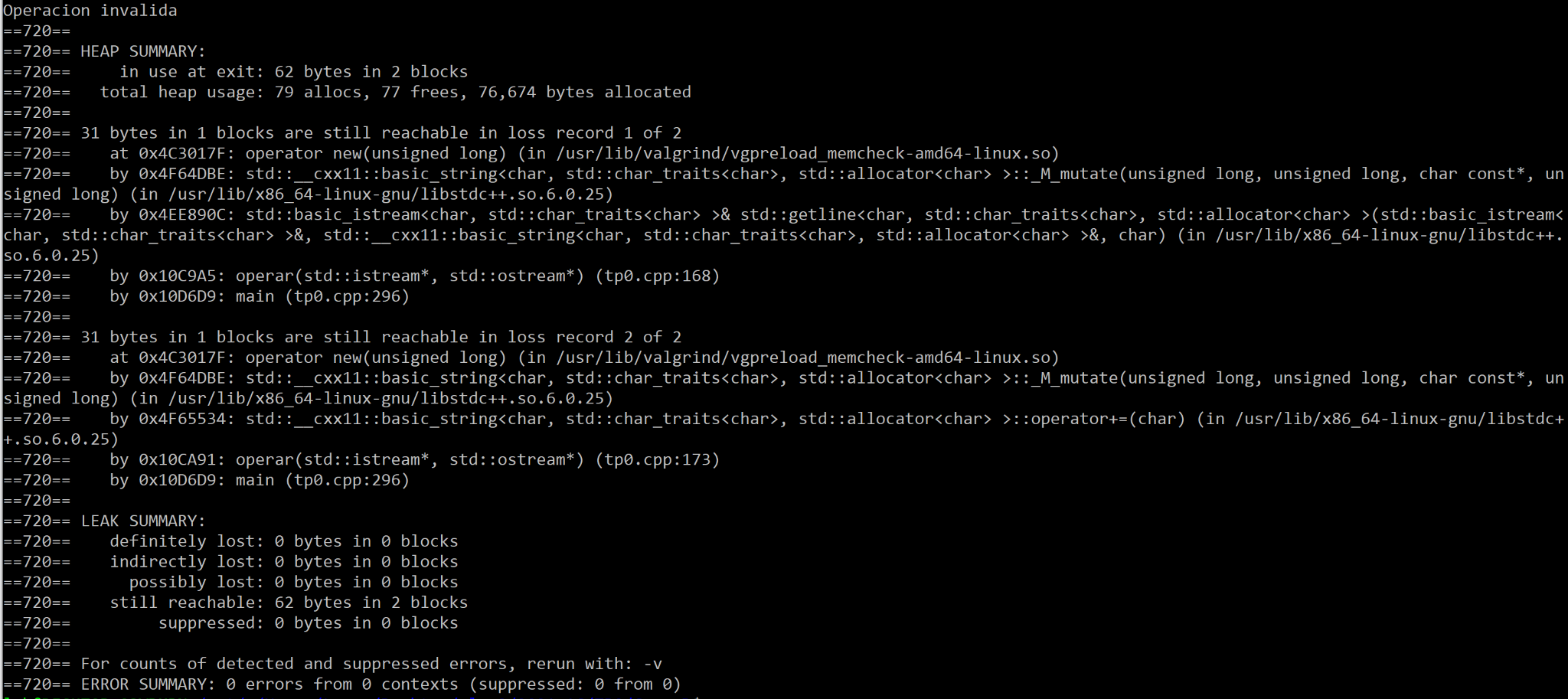
Código:

* Porque un *ushort* para la longitud de *digits*? No convendría un *ulong*? Con esto podrían representar aún más dígitos.
* No suena bien tener setters en *bignum*. ¿Por qué un número debería ser mutable y que cambie su precisión o signo? En caso de necesitarlos por conveniencia en la implementación deberían ir a la parte privada de la clase.
* No están usando el operador >> en ningún momento. Parece que solo lo pusieron “para cumplir” porque todo el trabajo lo hace el constructor de bignum que utiliza un string. Se pierden por lo tanto muchas de las ventajas del operator>>. Una posible implementación de este operador podría seguir la línea del opeartor>> de complejos que explicó Leandro al comienzo del cuatrimestre. Se podría leer dígitos hasta encontrar un carácter que no lo es. En ese momento, lo devuelven al stream, ensamblan el bignum con los datos leídos y terminan. Si en algún momento viene un carácter inesperado, marcar el estado de invalidez en el stream y salir del proceso. Hecho esto, leer las expresiones aritméticas resultaría mucho más simple, considerando que el operador >> permite saltear los espacios. En cada línea de la entrada, sólo sería necesario leer el primer bignum, luego el operador -y validarlo- y finalmente el segundo bignum. Esto debería usarse en el ciclo while donde leen los datos de entrada (que actualmente realiza *operar()*)
* Hay partes donde a in *int* le restan un *bool.* Esto es dudoso. Traten de corregirlo y evitarlo
* Sugerencia: pueden configurar un make test para que corra el programa con archivos de prueba en valgrind. Pueden hacer que compare (con un diff o algo) la salida obtenida con la salida verdadera. De esta forma automatizan el testeo de su programa.

Mem leak 1. Ocurre luego de ingresar un operación invalida luego de hacer cálculos con números de precisión grandes.



Esta captura parece mostrar dónde está el problema que genera el mem leak 1.



Abajo hay otra captura de un mem leak con este mismo problema.

Mem leak: Idem al caso de arriba.

