CS29206 Systems Programming Laboratory
Spring 2024
Lab Assignment: 4
Date: 29–Feb–2024

Fixing memory-related problems using valgrind

You are given a zip file `A4.zip` that contains the source code of a library, an application program based on the library, and a few other things. If you build the library, and compile and run the code (under *valgrind*), you notice two problems. First, the code does not run properly. This is because of a bug in the source code of the library. This bug does not result in memory losses, but leads to uninitialized memory. Second, the application program is very sloppy with its memory management, and leads to memory losses of several kinds. Running the application program under *valgrind* will point out both the problems. Your task is to identify the sources of the problems, and repair both the library and the application program so that all the above memory-related problems are eliminated. The various components of `A4.zip` are now described.

The *listutils* library (*listutils.h* and *listutils.c*)

The library makes an implementation of the ADT *sorted list* (of integers). In order to store such a list, a data type *slist* is defined. This data type is a structure consisting of two fields: the size *n* of the list, and an int pointer *item*. The pointer *item* is dynamically allocated memory to store exactly *n* int variables. The dynamic array stores the list in sorted (ascending) order. Duplicates are allowed in our sorted lists. The empty list has $n = 0$ and *item* = `NULL`. The library implements the following functions on *slist* data types.

```
slist listinit ( ) ;
```
This function returns the empty list. You must initialize an *slist* before passing it to any *listutils* function. If you use an uninitialized *slist*, you are very likely to encounter a segmentation fault. Our application program never does that. Note, however, that any *slist* returned by a *listutils* function is appropriately structured, that is, you may store a returned *slist* in an uninitialized *slist* structure. This structure can now be supplied as inputs to other functions without any danger of segmentation faults.

```
void listkill ( slist * ) ;
void listfree ( slist * ) ;
```
These two functions are to be used to convert an initialized *slist* to the empty list. The function *listkill* is unclean in terms of memory loss, whereas the function *listfree* is clean in that respect.

```
slist listcopy ( slist ) ;
```
This function returns a copy of the input list. The *item* array of the copy is allocated fresh memory, and leaves the input *slist* unchanged. Later, you can use the input *slist* and the returned *slist* independently of one another.

```
slist listadd ( slist , int ) ;
slist listsub ( slist , int ) ;
```
These functions are respectively for inserting and deleting an element in an *slist*. The input *slist* is not modified. A fresh *slist* is created as a result of the insertion or the deletion operation, and is returned. Insertion returns a sorted list with exactly one extra item (duplicates are allowed). Deletion returns a copy of the input list if the element to be deleted is not present in the list. Otherwise, the output list is a copy of the input list with only one instance of the element missing.

```
void listprn ( slist ) ;
```
Print an *slist* (sorted printing with duplicates, if any).

The library further defines a *listop* data type. In order to understand this, let us think of a sequence T of m *slist* operations done on our lists. The result of the i-th operation is stored in an *slist* `%i` for $1 \leqslant i \leqslant m$. Also, assume that `%0` is the empty list. For $i \neq j$, we have two different *slist* structures `%i` and `%j`. Their *item* arrays are allocated disjoint memory.

The data type *listop* is a structure of three integers. The first field *operation* is the code of an *slist* operation (the codes of the permitted operations are *#define*'d in *listutils.h*). This is followed by two arguments *arg*1 and *arg*2. An argument is the index of an *slist* appearing earlier or an integer to insert or delete. If an operation does not require any of these arguments, that argument can be set to 0 (or –1 or left undefined). For example, insertion has code 5. Therefore the i-th *listop* storing *operation* = 5, *arg*1 = j, and *arg*2 = 3 is meant for setting `%i` = `%j` + 3 (insert the integer 3 to `%j` and store the result in `%i`). We must have $j < i$. After this operation, `%i` will contain the new list, whereas `%j` will remain unaffected.

A file *ops.txt* storing 10 operations with the implied meanings are given below. This file is used in the sample given below. We assume that `%0` is the empty list.

Line Number	Line	Meaning	Effects(s)
0	10	The number of *slist* operations is $m = 10$	
1	5 0 4	`%1` = `%0` + 4	`%1` = { 4 }
2	3 0	Free `%0` and copy to `%2`	`%0` = { } and `%2` = { }
3	5 1 2	`%3` = `%1` + 2	`%3` = { 2, 4 }
4	1	Initialize `%4` to the empty list	`%4` = { }
5	5 4 7	`%5` = `%4` + 7	`%5` = { 7 }
6	5 5 6	`%6` = `%5` + 6	`%6` = { 6, 7 }
7	5 6 3	`%7` = `%6` + 3	`%7` = { 3, 6, 7 }
8	2 7	Kill `%7` and copy to `%8`	`%7` = { } and `%8` = { }
9	5 3 4	`%9` = `%3` + 4	`%9` = { 2, 4, 4 }
10	6 9 4	`%10` = `%9` – 4	`%10` = { 2, 4 }

The library provides the following function for reading the operations from a text file (like *ops.txt*).

```
listop *freadops ( char * ) ;
```

The input to this function is a file name. If the file contains m operations (written at the beginning of the file), then a dynamically allocated array of $m + 1$ *listop* structures is returned. The first m entries of this array are the m operations for computing the lists `%i` with $1 \leqslant i \leqslant m$. The last entry stores `ENDOFOPS` that null terminates the array. Since m is not stored in the array, this entry indicates that there are no further operations.

The application program (*listapp.c*)

This program takes a command-line argument standing for the name of the operations file (like *ops.txt*). It first calls *freadops* on this file to store the sequence of operations in a (null-terminated) array *OPS* of *listop* structures. It then sets `%0` to the empty list (this is not coming from any operation), and enters a loop to generate and print `%i` for $1 \leqslant i \leqslant m$ as per the operations listed in *OPS*. The lists generated are stored in a table T of *slist* structures. For each operation, an appropriate library function is called. When a special operation called `ENDOFOPS` is encountered in *OPS*, the loop is broken, and the program terminates. The application program makes no attempts to free any memory allocated during its running.

A random operation generator (*genops.c*)

The example operations file used in this write-up is supplied as *OPS.txt*. This is for your initial working on this assignment. For generating larger samples, you can use the code *genops.c* provided in `A4.zip`.

This program takes an (optional) command-line argument: the number *m* of operations. Without this argument, the default value *m* = 100 is used. A random list of *m* operations is written in the file *ops.txt*.

Makefile

A makefile is supplied in `A4.zip` to relieve you from any compilation effort. You can simply run

```
make runapp
```

to do all the jobs. This will create the library (*liblistutils.so*), compile the application program *listapp.c*, and finally run the compiled application under *valgrind*. It, however, requires the input operations file to have the name *ops.txt*. This filename is hard-coded in the makefile.

The operations generator *genops.c* is not compiled or run by *make runapp*, so no (new) *ops.txt* file is created in this process. In order to experiment with larger (or smaller) samples, use the following.

```
make genops
./genops 1000
```

Some additional targets are *library*, *app*, and *clean*. Read the makefile to know what they do.

What you have to do

Copy *OPS.txt* to *ops.txt*, do *make runapp*, and see the problems. Repair the problems.

- First, repair all the memory-loss issues. This needs looking into *listapp.c* only.
- Then, look at the source code of the library, and repair the error messages "*uninitialized memory*" belched out by *valgrind*. There is only one error in *listutils.c*.
- Do **not** change any file other than *listapp.c* and *listutils.c*. Your submission will be compiled under the other original files (*listutils.h*, *Makefile*, and so on) supplied to you in `A4.zip`.
- Do **not** change any data type or function prototype.
- Do **not** introduce any new variables in any of the two C files mentioned above.
- Do **not** rewrite any code. In *listapp.c*, add suitable *free*() statements, and replace bad library calls by good library calls. No other changes are permitted. In *listutils.c*, repair only the bug. Well, you need to add exactly one extra statement somewhere. Do only that, and nothing else.
- Manipulate any *slist* data **only** using the library calls. For example, you must **not** call *free*() on the *item* array of any *slist*. Use a library function that does this for you. This is how you should use any ADT implementation (even if you know the internals of that ADT implementation).

Submit to the Moodle server (before it is too late) **only** your updated files *listapp.c* and *listutils.c* (as individual C source files with those names, not as zip file(s)). You will get no credit if you submit any other files and/or do not strictly follow the **not** directives given above.

Sample Output

First, use the given *OPS.txt* to see what problems you have. The memory problems are highlighted in red.

```
$ cp OPS.txt ops.txt
$ make runapp
gcc -I. -Wall -fPIC -c -o listutils.o listutils.c
gcc -shared -o liblistutils.so listutils.o
gcc -o listapp -Wall -I. -L. -Wl,-rpath=. listapp.c -llistutils
valgrind ./listapp ops.txt
==13667== Memcheck, a memory error detector
==13667== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==13667== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==13667== Command: ./listapp ops.txt
==13667==
%0 = { }
$ %0 + 4
==13667== Conditional jump or move depends on uninitialised value(s)
==13667== at 0x48E1958: __vfprintf_internal (vfprintf-internal.c:1687)
==13667== by 0x48CBD3E: printf (printf.c:33)
==13667== by 0x484B5DF: listprn (in /home/abhij/IITKGP/course/lab/SPL/Spring24/prog/A4/liblistutils.so)
==13667== by 0x109778: main (in /home/abhij/IITKGP/course/lab/SPL/Spring24/prog/A4/listapp)
==13667==
==13667== Use of uninitialised value of size 8
==13667== at 0x48C569B: _itoa_word (_itoa.c:179)
==13667== by 0x48E1574: __vfprintf_internal (vfprintf-internal.c:1687)
==13667== by 0x48CBD3E: printf (printf.c:33)
==13667== by 0x484B5DF: listprn (in /home/abhij/IITKGP/course/lab/SPL/Spring24/prog/A4/liblistutils.so)
==13667== by 0x109778: main (in /home/abhij/IITKGP/course/lab/SPL/Spring24/prog/A4/listapp)
==13667==
==13667== Conditional jump or move depends on uninitialised value(s)
==13667== at 0x48C56AD: _itoa_word (_itoa.c:179)
==13667== by 0x48E1574: __vfprintf_internal (vfprintf-internal.c:1687)
==13667== by 0x48CBD3E: printf (printf.c:33)
==13667== by 0x484B5DF: listprn (in /home/abhij/IITKGP/course/lab/SPL/Spring24/prog/A4/liblistutils.so)
==13667== by 0x109778: main (in /home/abhij/IITKGP/course/lab/SPL/Spring24/prog/A4/listapp)
==13667==
==13667== Conditional jump or move depends on uninitialised value(s)
==13667== at 0x48E2228: __vfprintf_internal (vfprintf-internal.c:1687)
==13667== by 0x48CBD3E: printf (printf.c:33)
==13667== by 0x484B5DF: listprn (in /home/abhij/IITKGP/course/lab/SPL/Spring24/prog/A4/liblistutils.so)
==13667== by 0x109778: main (in /home/abhij/IITKGP/course/lab/SPL/Spring24/prog/A4/listapp)
==13667==
==13667== Conditional jump or move depends on uninitialised value(s)
==13667== at 0x48E16EE: __vfprintf_internal (vfprintf-internal.c:1687)
==13667== by 0x48CBD3E: printf (printf.c:33)
==13667== by 0x484B5DF: listprn (in /home/abhij/IITKGP/course/lab/SPL/Spring24/prog/A4/liblistutils.so)
==13667== by 0x109778: main (in /home/abhij/IITKGP/course/lab/SPL/Spring24/prog/A4/listapp)
==13667==
%1 = { 0 }
$ free(%0)
%2 = { }
$ %1 + 2
==13667== Conditional jump or move depends on uninitialised value(s)
==13667== at 0x484B3B4: listadd (in /home/abhij/IITKGP/course/lab/SPL/Spring24/prog/A4/liblistutils.so)
==13667== by 0x10967E: main (in /home/abhij/IITKGP/course/lab/SPL/Spring24/prog/A4/listapp)
==13667==
%3 = { 0, 0 }
$ init()
%4 = { }
$ %4 + 7
%5 = { 0 }
$ %5 + 6
%6 = { 0, 0 }
$ %6 + 3
%7 = { 0, 0, 0 }
$ kill(%7)
%8 = { }
$ %3 + 4
%9 = { 0, 0, 0 }
$ %9 - 4
==13667== Conditional jump or move depends on uninitialised value(s)
==13667== at 0x484B479: listsub (in /home/abhij/IITKGP/course/lab/SPL/Spring24/prog/A4/liblistutils.so)
==13667== by 0x1096FD: main (in /home/abhij/IITKGP/course/lab/SPL/Spring24/prog/A4/listapp)
==13667==
%10 = { 0, 0, 0 }
==13667==
==13667== HEAP SUMMARY:
==13667== in use at exit: 448 bytes in 9 blocks
==13667== total heap usage: 16 allocs, 7 frees, 6,280 bytes allocated
==13667==
==13667== LEAK SUMMARY:
==13667== definitely lost: 12 bytes in 1 blocks
==13667== indirectly lost: 0 bytes in 0 blocks
==13667== possibly lost: 132 bytes in 1 blocks
==13667== still reachable: 304 bytes in 7 blocks
==13667== suppressed: 0 bytes in 0 blocks
==13667== Rerun with --leak-check=full to see details of leaked memory
==13667==
==13667== Use --track-origins=yes to see where uninitialised values come from
==13667== For lists of detected and suppressed errors, rerun with: -s
==13667== ERROR SUMMARY: 84 errors from 7 contexts (suppressed: 0 from 0)
```

After you repair the problems, your output should look as follows.

```
$ make runapp
gcc -shared -o liblistutils.so listutils.o
gcc -o listapp -Wall -I. -L. -Wl,-rpath=. listapp.c -llistutils
valgrind ./listapp ops.txt
==14041== Memcheck, a memory error detector
==14041== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==14041== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==14041== Command: ./listapp ops.txt
==14041==
%0 = { }
$ %0 + 4
%1 = { 4 }
$ free(%0)
%2 = { }
$ %1 + 2
%3 = { 2, 4 }
$ init()
%4 = { }
$ %4 + 7
%5 = { 7 }
$ %5 + 6
%6 = { 6, 7 }
$ %6 + 3
%7 = { 3, 6, 7 }
$ kill(%7)
%8 = { }
$ %3 + 4
%9 = { 2, 4, 4 }
$ %9 - 4
%10 = { 2, 4 }
==14041==
==14041== HEAP SUMMARY:
==14041== in use at exit: 0 bytes in 0 blocks
==14041== total heap usage: 16 allocs, 16 frees, 6,276 bytes allocated
==14041==
==14041== All heap blocks were freed -- no leaks are possible
==14041==
==14041== For lists of detected and suppressed errors, rerun with: -s
==14041== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Notice that the exact outputs you see on your machine may be different from what is shown in the above sample transcript. The errors will appear anyway.