PA 0: Environment Setup, AddressSanitizer and GDB

Due on Sunday 1/19/2020 at 11:59pm

Introduction

The objective of this programming assignment is to help you ready for other programming assignments by covering introductory subjects. Through this assignment, you will complete setting up your own development environment where you will work until the end of the semester, recap the basics of C/C++ programming and learn using AddressSanitizer and GDB to debug your programs. You are given a simple program, *buggy.cpp* about a linked list data structure that has, as the name suggests, several bugs that you are going to find out with the help of AddressSanitizer. Once you have corrected the program, submit a report documenting your findings and the process that you followed. More about the report coming up in the following.

Exercises

1. **[System Setup]** Set up the development environment by choosing one of these options: AWS, Oracle VM Virtual Machine with “Xubuntu” (i.e., lightweight Ubuntu) system, or a full Ubuntu setup. You can find the detailed guideline under the resources tab of the course website. Once you have your linux box ready, install various packages (e.g., g++, gdb, clion, AddressSanitizer) using standard commands (i.e., sudo apt-get install <package-name>)
2. **[C++ Compilation]** Compile your buggy C++ program to have an executable program called “buggy”, (not a.out). Note that buggy.cpp does not compile as it is. You must fix the compile errors first. Here is list of things to start with:
3. Write a statement in ***Blank A*** to include the required header files.
4. Write an access specifier in ***Blank B*** to allow variables to be used by outside of a class.
5. Correct the statements in **Lines 15, 16, 21, 28, 29,** where a member variable of an object is accessed through a pointer variable.
6. **[Compilation with symbol table]** From now, you begin fixing the runtime errors. First, compile your program. Note that if you compile your program as usual, the symbol table will not be loaded by default making the debugger messages not so informative to point out where the error is. To solve this problem, you need to compile with the “-g” option (e.g., g++ buggy.cpp -g). In addition, it is best not using any optimizations so that job of AddressSanitizer as a memory-error-catcher is easier.
7. **[GDB Start/Run/Backtrace]** Once compiled, (1) start gdb with the executable program name (i.e. buggy) and (2) run the program inside gdb. The gdb may stop after dumping a segmentation fault error. (3) Print a backtrace to see how your program reached the last statement with the error.
8. **[GDB Breakpoint/Print]** The backtrace summary shows that the program stops at the statement in Line 15 (you can see the line numbers in any editor). Let’s dig into what causes the segmentation fault error. (1) Set a breakpoint in Line 15, (2) run your program inside gdb. When you see a question, “Start it from the beginning? (y or n)”, choose y. Then, your program starts again from the beginning and stop at the breakpoint. (3) Then, print what is stored in mylist[i] by using a gdb command (neither cout nor printf).
9. **[C++ Runtime Error Fix (Null-Pointer)]** Write a value stored in mylist[i] and write a statement in Blank C to fix the segmentation error (Hint: de-refence the mylist[i] as a pointer instead of as a value).
10. **[C++ Runtime Error Fix (non-NULL garbage value Pointer)]** Compile the program again and run it inside gdb. You still face another segmentation fault in function *sum\_LL*, which you need to fix and explain. (Hint: The second part of function *create\_LL* should be corrected as well)
11. **[Dynamically Allocated Memory Deletion]** The function *create\_LL* dynamically allocates memory from the heap for elements (i.e. nodes) of mylist. Write code to free these allocate memory to avoid memory leak in blank D.
12. **[AddressSanitizer]** Now start the whole thing from scratch, but this time with AddressSanitizer instead of gdb. It is easy to get started with this binary analysis tool. Just compile your program with an additional option “-fsanitize=address”, run your program as is and see how it can quickly find the potential memory errors at even more ease. Note that this is a break-at-first-error type tool which makes your program exit upon finding the first possible cause of memory corruption. Therefore, you need to run your program a few times to catch all the errors.

**Report**

Submit a report with the corrected code on ecampus. The report should document the steps you followed to debug each problem precisely. Note that you should include the debugging steps under 2 modes of debugging: gdb and AddressSanitizer. Also note that AddressSanitizer is not really a debugger, rather a binary analysis tool that can point to potential errors. On the other hand, you would need gdb in this course for non-memory errors that are plentiful as well.

|  |
| --- |
| #include <iostream>   //Blank A  class node {   //Blank B      int val;      node\* next;  };    void create\_LL(vector<node\*>& mylist, int node\_num){      mylist.assign(node\_num, NULL);      //create a set of nodes      for (int i = 0; i < node\_num; i++) {          //Blank C          mylist[i].val = i;          mylist[i].next = NULL;      }      //create a linked list      for (int i = 0; i < node\_num; i++) {          mylist[i].next = mylist[i+1];      }  }  int sum\_LL(node\* ptr) {      int ret = 0;      while(ptr) {          ret += ptr.val;          ptr = ptr.next;      }      return ret;  }  int main(int argc, char \*\* argv){      const int NODE\_NUM = 3;      vector<node\*> mylist;      create\_LL(mylist, NODE\_NUM);      int ret = sum\_LL(mylist[0]);      cout << "The sum of nodes in LL is " << ret << endl;      //Step4: delete nodes      //Blank D  } |

Table 1. Sample Buggy Code (buggy.cpp)