PA 0: Environment Setup, AddressSanitizer and GDB

Due on Friday 1/29/2021 at 11:59pm

Introduction

The objective of this programming assignment is to help you ready for other programming assignments by covering some introductory topics. 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 several compile/syntax errors and 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.

Your Task

1. **[System Setup]** Set up the development environment by choosing one of these options:
   a. Oracle VM Virtual Machine with Xubuntu (i.e., lightweight Ubuntu) system,
   b. A full Ubuntu setup with dual boot along with existing OS,
   c. Or a remote but dedicated linux machine from Amazon AWS.
   d. You can find the step-by-step guide for the first option 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 is. You must fix the compile errors first. Here is list of things to start with:
   a. Write a statement in ***Blank A*** to include the required header files.
   b. Write an access specifier in ***Blank B*** to allow variables to be used from outside of the class.
   c. Correct the statements in **Lines 15, 16, 21, 28, 29,** where a member variable of an object is accessed through a pointer variable.
3. **[Compilation with symbol table]** Next, you begin fixing the runtime errors. First, compile your program. Note that if you compile your program as usual and then launch it under gdb, the symbol table will not be loaded and all printed variable names or line numbers to locate the errors will be in some internal address format - not easy for us to make sense of. To solve this problem, you need to compile with the "-g" option (e.g., **g++**

buggy.cpp -g). In addition, it is best not to use any optimizations so that the job of AddressSanitizer as a memory-error-catcher is easier.

4. **[GDB Start/Run/Backtrace]** Once compiled, do the following:
    a. Launch the executable under gdb (i.e. **gdb buggy**)
    b. Run the program under gdb (i.e., type **run** once **gdb** is ready)
    c. When (/if) stopped because of a runtime exception, print the "stack frame" by the command **backtrace** to see how your program reached the last statement with the error.

Note that "stack frame" may contain the sequence of all function calls from your code all the way up to and inside some library functions that were called by your program. For instance, if you call function t**estfunction()** from your **main()**, and then the **testfunction()** eventually calls a library function to allocate memory, and if your program crashes from inside the alloc function, **gdb** will crash and stop at the allocation function. However, to assist you with finding the source of the error, **gdb** will also contain the entire stack frame. At level 0, there would be the alloc function, then at level 1 the **testfunction()** and finally at level 2 you will find the **main()** function. There may be more levels below the main() to further indicate which program. The nice thing is you can even navigate between these levels and investigate variables in each layer.

5. **[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 or printf).
6. **[C++ Runtime Error Fix (Null-Pointer)]** Store a value (i.e., in the `value` member variable) in mylist[i] and write a statement in Blank C to fix the segmentation error (Hint: de-reference the mylist[i] as a pointer instead of as a value).
7. **[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)
8. **[Deletion of Dynamically Allocated Memory]** The function *create_LL* dynamically allocates memory from the heap for elements (i.e. nodes) of mylist. Write code to free these allocated memory to avoid memory leak in blank D.
9. **[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.

10. **[Using IDE]** Now, repeat everything in the above list of tasks, but this time under your favorite GUI IDE. We recommend using CLion, Visual Studio Code, or anything that you like. However, make sure that you are the IDE for debugging and running in addition to writing code. That means that you must be able to use all features of gdb from the IDE's menu items at the click of your mouse, without going back to the gdb's CLI. For this, make sure that CLion/Code can locate the g++ compiler and gdb debugger and use them. You may have to do this step manually if not done automatically by the IDE.

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, gdb is a full-fledged debugger that you would need for non-memory errors as well.

For no 10, you can just include a screenshot of debugging inside your IDE. That will be adequate for this part.

```
#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)