CYB220 Lab 8 – GDB

**Due: Tue Dec 3rd, 2024, 11:59 pm.**

**Points: 50 pts**

**Turn in: this report.**

Step 1: compile the program with “-g” flag to instrument useful data for the gdb debugger.

(Two files provided for this lab, so you don’t need to do this step.)

* g++ -g -o program-g++-dbg Score\_system\_new.cpp

(version 1: compiled with g++)

* clang++ -g -o program-clang-dbg Score\_system\_new.cpp (version 2: compiled with clang++)

Step 2: Run both versions, when the score\_system program asking for comments, enter EXACT ten “A”s

Based on the average score, final grade is: F

looks good? (Yes or No)AAAAAAAA

Comments - Looks good? - AAAAAAAA

Step 3: Practice using gdb to examine variable values and their locations.

----------------------Experiment #1----------------------

Debug/run the clang version with gdb: **gdb program-clang-dbg**

**A.** Once gdb started, the first thing to do is to set up break points. Your program will stop before executing the code at the break points.

gdb commands: **break or b**

eg: break <line #> or b <function number>

break 18

b main

For the location of the variables

Because all variables are local variables (no dynamic variables), we can set up a break point at the place where all variables have been declared.

**break 18**  set up a break point on line 18

To check the variable value before the buffer overflow happen, where should we set the break point?

* at the line to get user input to comments (cin >> comments;)
* Can you find the line number and set break point there? (“list” command may be helpful)

**B.** After setting up the break points, now run the program.

gdb command: **run or r**

It should run the program and stop at the first break points (or stop and ask for user input as the program executes).

If you want to do line by line debug, use command **next (or n)**

If you want to execute the program until next break point, use command **continue (or c)**

At break point #1, we can take a look at the locations of each local variable.

For example: print (or p) &scores  gives the address of the scores array.

print &average  gives the address of the average variable.

Or use display, eg. display scores  display the value stored in scores

At break point #2, we can examine the variable values by using “print” command.

For example: print scores  prints the scores array’s elements

print sum  prints the sum variable’s value

Once fill in the “value before overflow” column, use gdb command “next” to execute the next line of code and enter the user input for comments.

Then use “print” commands to print the variable values after the buffer overflow and fill in column #5.

Here is my finished table for the clang++ version.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Variable name** | **Type** | **Location (Mem Address)** | **Value before overflow\*** | **Value after overflow\*** |
| scores[10] | int array | df20 | 32,23,32,… | 32,23,32,… |
| name[100] | char array | deb0 | jiasong | jiasong |
| number\_or\_score | int | deac | 3 | 3 |
| average | double | dea0 | 29 | 29 |
| sum | int | de9c | 87 | 1094795585 |
| grade | char | de9b | 70 ‘F’ | 65 ‘A’ |
| Comments[5] | char array | de96 | NONE | AAAAA |

\*Overflow means the buffer overflow after giving comments (see picture below).

﻿

Based on the average score, final grade is: F

looks good? (Yes or No)AAAAAAAA

Comments - Looks good? - AAAAAAAA

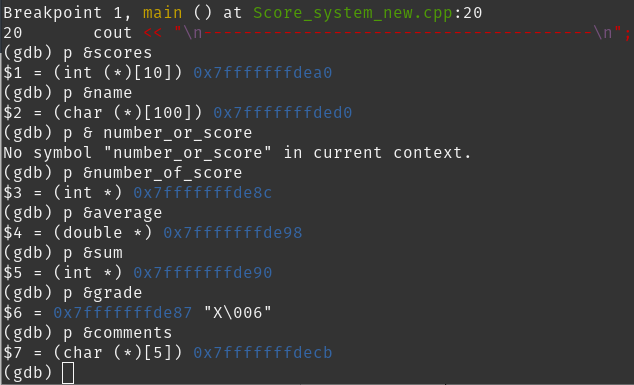
----------------------Experiment #2----------------------

(10 pts) Follow the steps before to debug the g++ version with gdb and fill in the table.

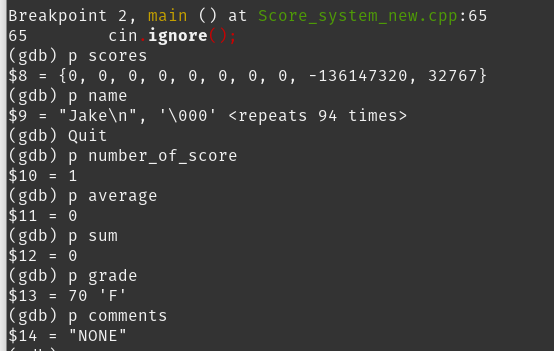
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Variable name** | **Type** | **Location (Mem Address)** | **Value before overflow\*** | **Value after overflow\*** |
| scores[10] | int array | 0x7fffffffdea0 | {0, 0, 0, 0, 0, 0, 0, 0, -136147320, 32767} | {0, 0, 0, 0, 0, 0, 0, 0, -136147320, 32767} |
| name[100] | char array | 0x7fffffffded0"Jake\n", '\000' <repeats 94 times> | "Jake\n", '\000' <repeats 94 times> | "AAAAA", '\000' <repeats 94 times> |
| number\_or\_score | int | 0x7fffffffde8c | 1 | 1 |
| average | double | 0x7fffffffde98 | 0 | 0 |
| sum | int | 0x7fffffffde90 | 0 | 0 |
| grade | char | 0x7fffffffde87 | 70 'F' | 70 'F' |
| Comments[5] | char array | 0x7fffffffdecb | "NONE" | "AAAAA |

For the g++ version:

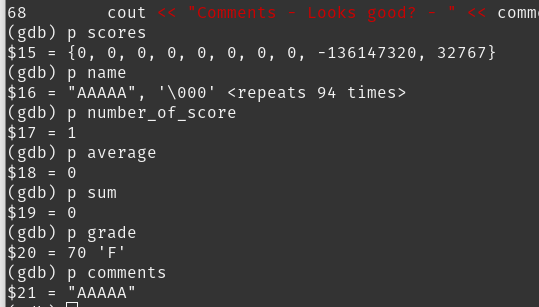
* (5pts) Required: A screenshot of where you get the locations of variables.



* (5pts) Required: A screenshot of where you get the value before overflow.



* (5pts) Required: A screenshot of where you get the value after overflow.



(5 pts)Draw a memory layout for the g++ version of program.

Memory Layout:

0x7fffffffdea0: | scores[0-9] |

-----------------------------------------------------

0x7fffffffded0: | name[0 - 99] |

-----------------------------------------------------

0x7fffffffde8c: | number\_or\_score |

-----------------------------------------------------

0x7fffffffde90: | sum |

-----------------------------------------------------

0x7fffffffde98: | average |

-----------------------------------------------------

0x7fffffffde87: | grade |

-----------------------------------------------------

0x7fffffffdecb: | Comments[0-4] |

(5 pts) Draw a memory layout for the clang++ version of program.

Memory Layout:

0x0000df20: | scores[0 - 9] |

-----------------------------------------------------

0x0000deb0: | name[0 - 99] |

-----------------------------------------------------

0x0000deac: | number\_or\_score |

-----------------------------------------------------

0x0000dea0: | average |

-----------------------------------------------------

0x0000de9c: | sum |

-----------------------------------------------------

0x0000de9b: | grade |

-----------------------------------------------------

0x0000de96: | Comments[0-4] |

(5 pts) Question: the program compiled with clang++ stores variable based on the order of declaration. What about the version of C++? Any pattern?

The g++ version also stores based on order of declaration, but goes from low addresses to high addresses. Clang++ scores from high addresses to low addresses.

(5 pts) Question: In the clang version, why did we get sum == 1094795585 after the overflow?

After the grade was overwritten, the remaining A's were stored in the sum variable, which led to a very large number.

(5 pts) Lab Summary (What have you learned in this lab? Anything interesting?)

I already knew how to use gdb, so I didn't really learn anything mind-boggling. I didn't know you could print out the addresses of variables directly, but that's really all I learned.