CSE 232, Lab Exercise #5

Assignment Overview

The aim of this lab is:

- do a code review to learn the Google Code style rules
- learn to debug your C++ programs using `gdb` (the GNU Project Debugger). The purpose of a debugger is to allow you to see what is going on inside your C++ program while it runs. In addition, you can use `gdb` to see what your program was doing at the moment it crashed.

Code Review

Code Style

Although the compiler can often compile very ugly source code, the primary reader of your code won't be computers, but other humans (including yourself). And in much the same way that well formatted essays are better at conveying information, well-formed code is easier to understand and debug. However, what does well-formed mean?

Conventions

Although there are a few different conventions for what exactly defines well-formed code, the important thing is to be consistent. For this assignment, we will be evaluating your Project 2 solutions with respect to the Google Style Guide. The point of this part of the lab is to evaluate (and improve) code by using a consistent style. Warning, much of the code we have shown in videos and examples may not conform to Google's standards.

Exchange your Project 2 solution with your partner. You will be editing their solution to either make if conform to the following rules or to add comments noting violations of the rules. Not all of the style guide will make sense given your incomplete mastery of C++, but we'll be focusing on the content you can apply.

Rules

Naming

- General Naming Rules
- Variable Names
- Function Names

Comments

- Comments
- Function Comments
- Implementation Comments

Formatting

- Line Length
- Spaces vs. Tabs
- Function Calls
- Horizontal Whitespace

Functions

- Parameter Ordering
- Write Short Functions
- Reference Arguments
- Horizontal Whitespace

Parting Words (copied from the end of the Google Style Guide)

Use common sense and BE CONSISTENT.

If you are editing code, take a few minutes to look at the code around you and determine its style. If they use spaces around their if clauses, you should, too. If their comments have little boxes of stars around them, make your comments have little boxes of stars around them too.

The point of having style guidelines is to have a common vocabulary of coding so people can concentrate on what you are saying, rather than on how you are saying it. We present global style rules here so people know the vocabulary. But local style is also important. If code you add to a file looks drastically different from the existing code around it, the discontinuity throws readers out of their rhythm when they go to read it. Try to avoid this.

OK, enough writing about writing code; the code itself is much more interesting. Have fun!

GDB and Debugging

Here are some of the useful actions that `gdb` can perform:

- Start your program and step through it line by line
- Make your program stop on specified conditions
- Show the values of variables used by your program
- Examine the contents of any frame on the call stack
- Set breakpoints that will stop your program when it reaches a certain point. Then you can step through part of the execution using step and next, and type continue to resume regular execution.

You will find the example programs you will debug during this lab in the lab directory.

Setting up gdb for use with C++

Before we start using `gdb`, we need to download a GDB init file (which will be called `.gdbinit`) that does two things. First, it allows us to print C++ Standard Library containers and data members from within `gdb`. Second, it allows us to debug our programs while treating the C++ constructs as the “bottom level”; in other words, when we debug a piece of a program that includes a vector, we don't want the debugger to go all the way into the code that defines the vector.

You will find this file, called `gdbinit` in the lab directory; save it to your home directory. Once saved in your home directory, please rename the file to `.gdbinit`. Note that the initial dot means the file is hidden; so once you rename it, it will not be visible by default in the graphical file manager.

Finally, debugging in Mimir is problematic. The system is setup for testing code and much of the underlying mechanisim confuses the debugger. Please do your work on x2go.

Getting started with gdb

C and C++ programs compiled with the GNU compiler and the `-g` option can be debugged using GNU's debugger `gdb` (actually, you can use `gdb` on code that is not compiled with `-g`, but unless you like trying to figure out how assembly code sequences map to your source code I wouldn't recommend doing so). Also, do not compile with an optimization flag (i.e. don't use `-O2`), or `gdb` will have a hard time mapping optimized machine code to your source code. For example:

```
$ g++ -g -std=c++11 -Wall myprog.cpp
```

To start `gdb`, invoke `gdb` on the executable file. For example:

```
$ gdb a.out
```

If your program terminates with an error, then the operating system will often dump a core file that contains information about the state of the program when it crashed. `gdb` can be used to examine the contents of a core file:

```
$ gdb core a.out
```

One good way to get started when you are trying to track down a bug, is to set breakpoints at the start of every function. In this way, you will quickly be able to determine which function has the problem. Then you can restart the program and step through the offending function line-by-line until you locate the problem exactly.

Common gdb Commands

```
gdb also understands abbreviations of commands, so you can just type up to
the unique part of a command name ("cont" for "continue", or "p" for "print")

help                   List classes of all gdb commands
help <topic>           Shows help available for topic or command

bt                  Shows stack: sequence of function calls executed so far
(or backtrace)         (good for pinpointing location of a program crash)
(or where)

up                    move up the stack
down                  move down the stack

run                    Starts program at the beginning
  run command line args

break
  break <line>           Sets breakpoint at line number <line>
  break <func-name>      Sets breakpoint at beginning of function <func-name>
    break main             Sets breakpoint at beginning of program

continue               Continues execution from breakpoint

condition <bp-num> <exp>   Sets breakpoint number <bp-num> to break only if
                           conditional expression <exp> is true
info break             Shows current breakpoints
disable [breakpoints] [bnums ...]  Disable one or more breakpoints
enable [breakpoints] [bnums ...]   Enable one or more breakpoints
clear <line>           Clears breakpoint at line number <line>
clear <func-name>      Clears breakpoint at beginning of function <func-name>
delete <bp-num>        Deletes breakpoint number <bp-num>
delete                 Deletes all breakpoints

step (or s)            Executes next line of program (steping into functions)
  step <count>           Executes next <count> lines of program
next (or n)            Like step, but treats a function call as a single
  next <count>             instruction
until <line>           Executes program until line number <line>


list                   Lists next few lines of program
  list <line>            Lists lines around line number <line> of program
  list <start> <end>     Lists line numbers <start> through <end>
  list <func-name>       Lists lines at beginning of function <func-name>

print <exp> (or inspect <exp>  Displays the value of expression <exp>

 To print in different formats:
   print/x  <exp> print the value of the expression in hexidecimal
                   (e.g. print/x 123   displays 0x7b)
   print/t  <exp> print the value of the expression in binary
                   (e.g. print/t 123   displays  1111011)
   print/d  <exp> print the value of the expression as unsigned int format
                   (e.g. print/d 0x1c displays 28)
   print/c  <exp> print the ascii value of the expression
                   (e.g. print/c 99  displays 'c')
   print   (int)<exp> print the value of the expression as signed int format
                   (e.g. print (int)'c' displays 99)
 To represent different formats in the expression (the default is int):
   0x suffix for hex: 0x1c
   0b suffix for binary: 0b101
       (e.g. print 0b101   displays 5,   print 0b101 + 3   displays 8)
   you can also re-cast expressions using C-style syntax  (int)'c'

 You can also use register values and values stored in memory locations
   in expressions:
   print $eax    # print the value stored in the eax register
   print *(int *)0x8ff4bc10  # print the int value stored at
                               memory address 0x8ff4bc10

quit                  Quits gdb
```

info commands for examining runtime and debugger state

`gdb` has a large set of info X commands for displaying information about different types of runtime state and about debugger state. Here is how to list all the info commands in help, and a description of what a few of the info commands do:

```
(gdb) help status           # lists a bunch of info X commands

(gdb) info frame            # list information about the current stack frame
(gdb) info locals           # list local variable values of current stack frame
(gdb) info args             # list argument values of current stack frame
(gdb) info registers        # list register values
(gdb) info breakpoints      # list status of all breakpoints
```

Sample gdb session

Below is output from two runs of gdb on programs from the lab directory. You will follow along by entering the commands on your own computer. Be sure you understand each step, and ask your TA if you don't understand why something is being done.

Before you do the debugging, just run the a.out and see what happens.

Run 1: badString.cpp

Looking at fn2

```
$ g++ -g -std=c++11 -Wall badString.cpp      #-- compile program with -g flag

$ gdb a.out                                  #-- invoke gdb with the executable

GNU gdb (Debian 7.7.1+dfsg-5) 7.7.1
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
...
Reading symbols from a.out...done.


(gdb) break main                            #-- set a breakpoint at the beginning
                                                of the main and the two functions
Breakpoint 1 at 0x400d87: file badString.cpp, line 35.

(gdb) break fn2
Breakpoint 2 at 0x400c72: file badString.cpp, line 11.


(gdb) run                                   #-- run the program
Starting program: /user/yourname/cse232_201701/Lab05/a.out

Breakpoint 1, main () at badString.cpp:35  # first executable line in main
35        string my_string = "abcdefg";

(gdb) list                                  #-- list the source code near the
                                                break point
30        size_t indx = str.find(ch);
```

```
31        return str.substr(indx-1, 3);
32      }
33
34      int main (){
35        string my_string = "abcdefg";
36        cout << fn2(my_string) << endl;
37        cout << fn1(my_string) << endl;
38
39      }

(gdb) list 15                             #-- list source code around line 15
10      char fn2(string str){
11        char ch = str[0];
12        for (auto i = str.size(); i>=0; --i){
13          cout << ch << endl;
14          if (str[i] < ch)
15            ch = str[i];
16        }
17        return ch;
18      }
19

(gdb) list                                 #-- list the next few lines of code
20      /*
21        make a substring of the parameter str
22        of size 3, consisting of:
23        - the smallest chararter as found by fn2
24        - the character in front of the smallest
25        - the character in back of the smallest.
26        Thus fn1("cdeaxyz") --> "eax"
27      */
28      string fn1(string str){
29        char ch = fn2(str);

(gdb) next                                 #-- execute the next instruction
36        cout << fn1(my_string) << endl;


(gdb)                                      #-- hitting Enter executes the
                                               previous command
                                               (next in this case)
Breakpoint 3, fn2 (str="abcdefg") at badString.cpp:11 # breakdpoint for fn2
11        char ch = str[0];

#-- also you can use the up and down arrows to scroll through previous commands

(gdb) print str                            #-- print str in fn2
$1 = "abcdefg"

(gdb) p ch                                 #-- p is short for the print command
$2 = 0 '\000'                              # ch in fn2 is not initialized

(gdb) print my_string                           # my_string in main, not fn2
No symbol "my_string" in current context.

(gdb) bt                               # bt == backtrace. Where in the call stack
##0  fn2 (str="abcdefg") at badString.cpp:11
#1  0x0000000000400e39 in main () at badString.cpp:36



To explain, activity goes from top to bottom. We are presently (#0) in
fn2. That was called by main (#1). We we have 2 functions running: we
are presently in fn2, which was called by main



(gdb) up
#1  0x0000000000400e39 in main () at badString.cpp:36
36        cout << fn2(my_string) << endl;      # can look at main stuff now

(gdb) print my_string
$3 = “abcdefg”                            # my_string is defined in main

(gdb) down
#0  fn2 (str="abcdefg") at badString.cpp:11
```

```
11        char ch = str[0];                    # back to fn2, where we are running

(gdb) next
12 for (auto i = str.size(); i>=0; --i){

(gdb) n                              # n is next
13          cout << ch << endl;
(gdb) n                              # just stepping through the loop
a
14          if (str[i] < ch)
(gdb) n
15            ch = str[i];
(gdb) n
12        for (auto i = str.size(); i>=0; --i){  # back to the top of the loop
(gdb) print ch                                   # how'd we do? WTH?
$4 = 0 '\000'
(gdb) print str[i]                                        # WTH?
$5 = (const char &) @0x60205f: 0 '\000'
(gdb) print i
$6 = 7
```

★
So what's the problem here? In we executed Line 15 above but it didn't change ch. Tell your TA the problem.

```
(gdb) cont                                  #-- continue the execution
Stop stepping through the loop, just run it until we hit a breakpoint or an error

Program received signal SIGSEGV, Segmentation fault.  # oopsie poopsie
0x0000000000400d19 in fn2 (str="abcdefg") at badString.cpp:14
14          if (str[i] < ch)

(gdb) bt
#0  0x0000000000400d19 in fn2 (str="abcdefg") at badString.cpp:14
#1  0x0000000000400e39 in main () at badString.cpp:36

(gdb) print ch
$7 = -128 '\200'              # there are no negative chars, so that's bad

(gdb) print i
$8 = 18446744073709547431     # ok, that is REALLY bad
```

★
So what's the problem again? Show your TA.

To Do: Read the fn2 code and fix it to run as it should. Look at the comments

Looking at fn1

Here is some debugging for fn1. I'm not explaining any of it, just showing some stuff. Notice I set a break point I wanted to change, searched the docs to find it. Command is **clear**

Also, the lines numbers are now off because I stuck a "fn2_fixed" in the code to get it to run.

```
(gdb) break main
Breakpoint 1 at 0x400e0f: file badString.cpp, line 40.
(gdb) break fn2_fixed
Breakpoint 2 at 0x400c72: file badString.cpp, line 20.
(gdb) unbreak fn2_fixed
Undefined command: "unbreak".  Try "help".
(gdb) help break
Set breakpoint at specified line or function.
```

```
break [PROBE_MODIFIER] [LOCATION] [thread THREADNUM] [if CONDITION]
PROBE_MODIFIER shall be present if the command is to be placed in a
probe point.  Accepted values are `-probe' (for a generic, automatically
guessed probe type) or `-probe-stap' (for a SystemTap probe).
LOCATION may be a line number, function name, or "*" and an address.
If a line number is specified, break at start of code for that line.
If a function is specified, break at start of code for that function.
If an address is specified, break at that exact address.
With no LOCATION, uses current execution address of the selected
stack frame.  This is useful for breaking on return to a stack frame.

THREADNUM is the number from "info threads".
CONDITION is a boolean expression.

Multiple breakpoints at one place are permitted, and useful if t---Type <return> to continue, or q <return> to quit---
heir
conditions are different.

Do "help breakpoints" for info on other commands dealing with breakpoints.
(gdb) help breakpoints
Making program stop at certain points.

List of commands:

awatch -- Set a watchpoint for an expression
break -- Set breakpoint at specified line or function
break-range -- Set a breakpoint for an address range
catch -- Set catchpoints to catch events
catch assert -- Catch failed Ada assertions
catch catch -- Catch an exception
catch exception -- Catch Ada exceptions
catch exec -- Catch calls to exec
catch fork -- Catch calls to fork
catch load -- Catch loads of shared libraries
catch rethrow -- Catch an exception
catch signal -- Catch signals by their names and/or numbers
catch syscall -- Catch system calls by their names and/or numbers
catch throw -- Catch an exception
catch unload -- Catch unloads of shared libraries
catch vfork -- Catch calls to vfork
clear -- Clear breakpoint at specified line or function
commands -- Set commands to be executed when a breakpoint is hit
---Type <return> to continue, or q <return> to quit---q
Quit
(gdb) clear
No source file specified.
(gdb) clear fn2_fixed
Deleted breakpoint 2
(gdb) break fn1
Breakpoint 3 at 0x400d8f: file badString.cpp, line 35.
Breakpoint 1, main () at badString.cpp:41
warning: Source file is more recent than executable.
41          string my_string = "abcdefg";
(gdb) n
42          cout << fn2_fixed(my_string) << endl;
(gdb) continue
Continuing.
terminate called after throwing an instance of 'std::out_of_range'
  what():  basic_string::substr: __pos (which is 18446744073709551615) > this->size() (which is 7)

Program received signal SIGABRT, Aborted.
0x00007ffff7244067 in __GI_raise (sig=sig@entry=6) at ../nptl/sysdeps/unix/sysv/linux/raise.c:56
56      ../nptl/sysdeps/unix/sysv/linux/raise.c: No such file or directory.

(gdb) bt
#0  0x00007ffff7244067 in __GI_raise (sig=sig@entry=6) at ../nptl/sysdeps/unix/sysv/linux/raise.c:56
#1  0x00007ffff7245448 in __GI_abort () at abort.c:89
#2  0x00007ffff7b31b3d in __gnu_cxx::__verbose_terminate_handler() () from /usr/lib/x86_64-linux-gnu/libstdc++.so.6
#3  0x00007ffff7b2fbb6 in ?? () from /usr/lib/x86_64-linux-gnu/libstdc++.so.6
#4  0x00007ffff7b2fc01 in std::terminate() () from /usr/lib/x86_64-linux-gnu/libstdc++.so.6
#5  0x00007ffff7b2fe19 in __cxa_throw () from /usr/lib/x86_64-linux-gnu/libstdc++.so.6
#6  0x00007ffff7b85cdf in std::__throw_out_of_range_fmt(char const*, ...) () from /usr/lib/x86_64-linux-gnu/libstdc++.so.6
#7  0x00007ffff7b91752 in std::string::substr(unsigned long, unsigned long) const ()
   from /usr/lib/x86_64-linux-gnu/libstdc++.so.6
#8  0x0000000000400e45 in fn1 (str="abcdefg") at badString.cpp:37
#9  0x0000000000400f0f in main () at badString.cpp:43
```

```
(gdb) up 9
#9  0x0000000000400f0f in main () at badString.cpp:43
43        cout << fn1(my_string) << endl;

(gdb) down 1
#8  0x0000000000400e45 in fn1 (str="abcdefg") at badString.cpp:37
37        return str.substr(indx-1, 3);

(gdb) print indx
$1 = 0
```

★
So what's wrong? Can you fix it? Show your TA

Keyboard shortcuts in gdb

gdb supports **command line completion**; by typing in a prefix you can hit `TAB` and `gdb` will try to complete the command line for you.

Also, you can give just the **unique prefix** of a command as the command and `gdb` will execute it. For example, rather than entering the command `print x`, you can just enter `p x` to print out the value of `x`.

The **up and down arrow keys** can be used to scroll through previous command lines, so you do not need to re-type them each time.

If you just hit `RETURN` at the `gdb` prompt, `gdb` will execute the **most recent previous command** again. This is particularly useful if you are stepping through the execution, then you don't have to type next each time you want to execute the next instruction, you can just type it one time and then hit `RETURN`.