CS 246 Spring 2018 - Debugging Tutorial

May 25, 2018

Summary

1 Let your tools help you

While this is more error prevention than debugging, there are certain classes of errors that the compiler, the C++ programming language, and some libraries can help you catch. Program defensively wherever possible.

1.1 Compiler

- Use `-Wall` when compiling at a minimum; consider also using `-Wextra` and `-Wpedantic`.

1.2 C++

- Use `const` liberally.
- Use `override` keyword when writing polymorphic code.

1.3 Libraries

Take advantages of any features that a library may provide to help you catch mistakes. For example, in the Standard Template Library (STL), use `std::vector::at()` rather than `std::vector::operator[]` since it will raise the exception `std::out_of_range` if you try to index the vector out of bounds.

2 Print statements

In order to be able to debug your code, you first need to understand what it is doing, the changes in its state, and where it is when things go wrong. An easy way to accomplish this is by putting print statements at strategic locations so you can track control flow.

- Always print to standard error and close files.
- Put print statements at the beginning and end of routines.
- If a variable is changing unexpectedly, print its value wherever it might change.
- Comment out, rather than remove, debug print statements until you are *sure* the program works.

You may find it helpful to use a preprocessor macro to simplify your print statements. This one allows an optional message to be printed beforehand, which can be used to label the expression to be printed. (Note that either the label or the expression can be omitted.) A particularly nice feature is that it automatically includes for the macro call the following information:

- the name of the executing function,
- the file name, and
- the line number.

See `~/cs246/1185/tutorials/debugging/examples/debugOutput1.cc`.

```
#include <iostream>
#define DPRT( title, expr ) \
{ std::cerr << #title "\t\"" << __PRETTY_FUNCTION__ << "\" " \
<< expr << " in " << __FILE__ << " at line " \
<< __LINE__ << std::endl; }
```

It will still be tedious to comment out all uses of the macro. An even better approach is to use the preprocessor to turn off the output. (This is much easier if you're using makefiles since you'd just need to modify the `CXXFLAGS` variable and recompile.)

See example `~/cs246/1185/tutorials/debugging/examples/debugOutput2.cc`.

Note that for hand-marking questions, it would be a good idea to remove the `DPRT` commands so that the markers can more easily see what you're doing. Use `egrep` to make sure you don't miss any occurrences!

3 Assertions

Assertions can significantly increase the execution cost of your program, and not all things can (or should) be checked with assertions. However, the strength of using assertions is that if you violate one, your program is stopped immediately, and you know precisely where.

- Assert your easy-to-check preconditions, such as `ptr != nullptr` or containers/strings not being empty.
- Compile with `-DNDEBUG` removes all assertions from your program.
- Never put computations needed by your program in your assertion. If you remove the assertions, then your program will work differently.
- Use comma-expressions to add documentation to your assertion message.

- Sometimes useful to assert that will never reach/execute a particular statement, just to make sure you're catching all cases. In that case, `assert( false );`

See `~/cs246/1185/tutorials/debugging/examples/assertion.cc` for an example of writing an assertion.

```
#include <cassert>

unsigned int stoppingDistance( Car * carPtr ) {
    assert( carPtr != nullptr );
    ...
    assert( ("Internal error", distance > 0) );
    ...
}

$ ./a.out
a.out: assertion.cc:12: unsigned int stoppingDistance(Car*):
   Assertion `("Internal error", distance > 0)' failed.
Aborted
```

4 Debugger `gdb`

`gdb`, the GNU Debugger, is the C++ debugger installed on the student environment. It may not be the newest tool out there, but it provides all of the basic features that more modern debuggers have, so learning how to use it will be beneficial in the long run for more than this course. It is interactive and effectively allows you to dynamically add and remove debug print statements as well as look at your run-time stack. In order to use it, you need to compile and link your code with the `-g` flag.

Note that GDB and Emacs *can* work well together, allowing you to see the source code about to be executed whenever the program execution is paused. (Currently, need to use `gud-gdb` rather than just `gdb` in `emacs`.)

4.1 First example: finding an infinite loop

4.1.1 Running your first program in gdb

- Compile `backwardsCat.cc` using the command: `g++14 -g backwardsCat.cc -o backwardsCat`
- Run GDB on the executable: `gdb ./backwardsCat`
- You will see the prompt `(gdb)`.
- Use the command `help` (or `h`) to see what types of commands are available. (Note that you can use the command `quit`, or `q`, or `Ctrl-d` to exit GDB.)
- Use the command `run` (or `r`) to run your program.
- The program is now waiting for your input, so type something and then press `Ctrl-d` at the start of a new line when you're done to signal end-of-file.
- Uh oh! The program is no longer responding, so it's probably in an infinite loop! Press `Ctrl-c` to terminate the program, which returns you to the GDB prompt.

4.1.2 Displaying basic diagnostic information and navigating the call stack

Notice that unlike when running a program directly from the command line, `Ctrl-c` didn't end the program; it just returned us to the GDB prompt. We now have the opportunity to ask GDB to help us identify what has gone wrong.

- You can use the `backtrace` (or `bt`) command to see the list of stack frames that the program was executing when you aborted the execution. (You will want to skip the frames that are executing library code, and focus on the ones in your own code.)

- You can use the commands `up` or `down` to move up or down the stack; alternatively, you could enter a specific frame by using the command `frame` (or `f`) followed by the frame number, where 0 is the most recent frame.

- Once you're in the desired frame, you can use the `list` (or `l`) command to list 10 lines of code at a time.

- Let's take a look at the values of the variables in the `reverse` function. You can use `print` (or `p`) to print the contents of the various variables. In fact, if the variable you're trying to print is a pointer, you can print its dereferenced value as `p *ptr` or `p ptr->next`. We can even print `s.size()` to see that `finish` is set correctly. Everything looks reasonable, until we check the value of `posn`, which is huge!

4.1.3 Navigating a running program using breakpoints and stepping

In order to determine the source of the problem, we should run our program more carefully, observing its behaviour carefully.

- First, stop the current execution of the program using the `kill` or `k` command.

- This time, we want to see what is causing the infinite loop. It's probably `reverse`'s fault based on the value of `posn`, a local variable, so let's run that loop more carefully. First, let GDB know that we want to pay special attention to `reverse`. We will use the `break` command, whose short-form is `b`, to tell GDB to set a breakpoint by specifying: `break reverse`. (You can also set breakpoints at lines of code using `break fileName:lineNumber`.)

- Start running the program again.

- This time GDB stops when `reverse` is called. We want to carefully watch its execution, so we will `step` (or `s`) to continue. (Alternatively, `next` or `n` can be used to step *over* the execution of a line or function, which executes the function without stopping.)

- Once we're inside the loop, we should print out the value of `posn`, but it looks correct. Let's run the loop a few times, but since manually printing `posn` every time is tedious, use the `display` (or `d`) command to ask GDB to print its value after every step.

- A few more steps should let us identify the problem. As a temporary fix, we can change `start` to 1 to see if the program is almost correct after this change. Use `set variable start = 1`, and then type `continue` (or `c`) to see whether the program is otherwise correct.

- The code runs properly until `posn` reaches 0 and then 1 is subtracted from it.

4.2 Second example: Finding out why the value of a variable is wrong (bookstock.cc)

Next, let's consider a different program, `bookstock.cc`. It sets up a copy of Homer's "Odyssey" with 10 copies in stock. We want to modify the contents of the `AntiqueBook` that `odysseyStock` contains to first have 20 copies, and then change the book information to be for the "Illiad". However, running the program shows us that the value of `odysseyStock.numCopies()` is now wrong after changing the book's stock information. Unfortunately, not only does the program not crash, but if we use `valgrind`, it doesn't indicate that there are any memory leaks or errors.

4.2.1 Detecting more obscure bugs using watchpoints

- First, let's double-check the contents of `odysseyStock`. Set a breakpoint right before the second `std::cout`, which is line 75, and run the program.
- Try printing `odysseyStock.numCopies()`. The value should be `20`, but it isn't.
- If we look at the intervening code, we aren't obviously changing the number of copies anywhere. We could step through the program with the values of `odysseyStock.numCopies()` displayed, but for a larger program, this could be very inefficient and time-consuming. Instead, we'd rather let GDB tell us when the value changes.
- Kill the execution of the program and delete the breakpoint by using the command `delete break 1`.
- Set a breakpoint on `main`. Run the program and, once it's paused, step until `odysseyStock` is created, then set a watchpoint via `watch odysseyStock.numCopies_`.
- Now resume the program using continue. Notice that the program stops when `watch odysseyStock.numCopies_` is assigned the correct value. So far, so good!
- Continue again. Whoops! The program also stops when we call `AntiqueBook::value(int value)`. This is the problem!

4.3 More commands of interest

There are a number of commands that we didn't have the chance to use in these short examples.

- The `undisplay`, (or `undisp` or `delete display`) command stops a variable from displaying.
- The `delete` (or `del` or `d`) command can be used to remove breakpoints, watchpoints and so forth. To delete breakpoints, you'll need to know their number. You can see all of your breakpoints using the `info breakpoints` command.
- Breakpoints can also be temporarily disabled (and subsequently reenabled) using the `disable` and `enable` commands.
- The `finish` command continues until the end of the current function.
- The `step` and `next` commands can optionally be supplied with a number of lines to step.
- The `call` command calls the function given as an argument. For example, `call f()`.
- Pressing enter (i.e. supplying a blank command to GDB) repeats the previous command.

You can also put a set of commands into a file, and load them when you start GDB so that you don't have to keep entering them every time you restart GDB.

5 Memory checker `valgrind`

`valgrind`[1] is an open-source, free set of tools. We are primarily concerned with its memory error detection capabilities, such as ability to detect memory leaks or dangling pointers.

- Use of `valgrind` can significantly slow your program; but, it may help you detect errors that are otherwise hard to find.
- However, be cautious in its use since it may state that there are errors that don't actually exist.
- In order to use it, you need to compile and link your code with the `-g` flag.

```
$ cat test.cc
int main() {}
$ g++14 -g test.cc
$ valgrind ./a.out
==139225== Memcheck, a memory error detector
==139225== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==139225== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==139225== Command: ./a.out
==139225==
==139225==
==139225== HEAP SUMMARY:
==139225==     in use at exit: 0 bytes in 0 blocks
==139225==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==139225==
==139225== All heap blocks were freed -- no leaks are possible
==139225==
==139225== For counts of detected and suppressed errors, rerun with: -v
==139225== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

The key bit of information here is that 0 bytes were allocated and 0 were freed, thus no leaks are possible. Note, however, that just having the same number of bytes allocated as freed doesn't mean that you don't have a memory leak. What we want to see is that all heap blocks were freed. We'll be focusing on this part of the `valgrind` report in our subsequent examples. One other issue we'll need to be aware of is that not all C++ libraries free their memory, and neither does the C++ runtime.

```
$ cat test.cc
int main() {
    int * p = new int;
    delete p;
}
$ g++14 -g test.cc
$ valgrind ./a.out
==68954== Memcheck, a memory error detector
==68954== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==68954== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==68954== Command: ./a.out
```

[1] According to `http://valgrind.org/docs/manual/faq.html`, the name `valgrind` is pronounced as: "The "Val" as in the word "value". The "grind" is pronounced with a short 'i' – ie. "grinned" (rhymes with "tinned") rather than "grined" (rhymes with "find")."

```
==68954==
==68954==
==68954== HEAP SUMMARY:
==68954==     in use at exit: 72,704 bytes in 1 blocks
==68954==   total heap usage: 2 allocs, 1 frees, 72,708 bytes allocated
==68954==
==68954== LEAK SUMMARY:
==68954==    definitely lost: 0 bytes in 0 blocks
==68954==    indirectly lost: 0 bytes in 0 blocks
==68954==      possibly lost: 0 bytes in 0 blocks
==68954==    still reachable: 72,704 bytes in 1 blocks
==68954==         suppressed: 0 bytes in 0 blocks
==68954== Rerun with --leak-check=full to see details of leaked memory
==68954==
==68954== For counts of detected and suppressed errors, rerun with: -v
==68954== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Note that there was only one allocation, but `valgrind` reports two since the C++ runtime isn't freeing its memory.

See `~cs246/1185/tutorials/debugging/examples/valgrind1.cc`.

Try running `valgrind` with `-leak-check=full`, which **must** be specified before the executable name.

Let's look at some common errors, focusing on the `valgrind` error messages: See `~cs246/1185/tutorials/debugging/examples/valgrind2.cc`.

6 Trapping session information

There may be times when you need to record what is happening in your shell session. For example, you have multiple pages of compiler error messages, but since those are often cumulative i.e. one error triggers others, you normally want to focus upon and fix the first few, recompile, and then see what is left. Or you want to document exactly what happens for a variety of program executions, with and without command-line arguments and I/O redirection.

While it may be possible to just use I/O redirection, there's a convenient tool that you can use called `script`. Invoked without parameters, it traps all further interactions in a text file with the name `typescript` until you type `exit` to end the session. You can then easily browse, search, or edit the file. See `man script` for some of the options available.

Note, however, that it traps *everything*, include backspaces, so that you may want to run the output through `scriptfix` to clean it up first. Also, depending upon how your aliases are set, they may not be available in the `script` session. If you put them in your `~/.bashrc` file, they should still be available.

```
Script started on Fri 01 Jun 2018 10:01:19 AM EDT
$ cat t
abc
def
$ g++14 -g backwardsCat.cc
c
$ ./a.out < t
Enter some text:
^C
$ gdb ./a.out
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
```

```
Copyright (C) 2016 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".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./a.out...done.
(gdb) break reverse
Breakpoint 1 at 0x4013ef: file backwardsCat.cc, line 8.
(gdb) run < t
Starting program: /u1/ctkierst/cs246/1185/tutorials/debugging/examples/a.out < t
Enter some text:

Breakpoint 1, reverse (s="def", start=0, finish=18446744073709551615) at backwar
8 std::string reverse(std::string s, size_t start = 0, size_t finish = std::stri
(gdb) n
9   if (finish == std::string::npos) finish = s.size();
(gdb)
10   std::string answer;
(gdb)
11   for (auto posn = finish - 1; posn >= start; --posn) {
(gdb) disp answer
1: answer = ""
(gdb) disp posn
2: posn = 3
(gdb) n
13     if (posn < s.size()) {
1: answer = ""
2: posn = 2
(gdb)
14       answer += s[posn];
1: answer = ""
2: posn = 2
(gdb)
11   for (auto posn = finish - 1; posn >= start; --posn) {
1: answer = "f"
2: posn = 2
(gdb)
13     if (posn < s.size()) {
1: answer = "f"
2: posn = 1
(gdb)
14       answer += s[posn];
```

```
1: answer = "f"
2: posn = 1
(gdb)
11    for (auto posn = finish - 1; posn >= start; --posn) {
1: answer = "fe"
2: posn = 1
(gdb)
13      if (posn < s.size()) {
1: answer = "fe"
2: posn = 0
(gdb)
14        answer += s[posn];
1: answer = "fe"
2: posn = 0
(gdb)
11    for (auto posn = finish - 1; posn >= start; --posn) {
1: answer = "fed"
2: posn = 0
(gdb)
13      if (posn < s.size()) {
1: answer = "fed"
2: posn = 18446744073709551615
(gdb) q
A debugging session is active.

Inferior 1 [process 122351] will be killed.

Quit anyway? (y or n) y
$ exit
exit

Script done on Fri 01 Jun 2018 10:02:18 AM EDT
```