

# CS256—Advanced Programming

Lab 1, 18 Jan.

Due: 21 Jan.

Jim Rogers

`jrogers@cs.earlham.edu`

Spring 2011

## Objectives

1. To (re)gain familiarity with the emacs editor and the Gnu C++ compiler (g++) on the CS Department unix machines.
2. To gain experience with Pre- and Post-conditions, loop invariants and the assert mechanism and the Gnu debugger (gdb).

## Procedure

We will be using the Earlham Computing Services (ECS) Mac lab machines as X-terminals to connect to the Computer Science Department (CS) unix/X-windows machines.

### Logging onto ACL machines from a Mac box.

1. Log into the Mac using your ECS userid and password (not your CS userid and password).
2. Double-click the ACL script icon the bottom dock. This will open Parallels Desktop and start up an ubuntu linux virtual machine, running in parallel with Mac OS-X, which is configured to request a host from the CS server `quark.cs.earlham.edu`.
3. If Parallels notifies you that it is checking for updates (or has an update ready to install) select **Cancel**. (Do **not** let it install any updates.)
4. `quark` will respond with a “chooser” widget, offering you a choice of three hosts. Select the first one (unless you have some particular reason to dislike it) and click **accept** (or just double click the host name).
5. `quark` will pass you to the host you selected. This will present you with a “login” widget. Login using your CS userid and password.
6. Your windows manager (`vtwm` unless you have changed it) will then run. For a tutorial on interacting with the `vtwm` see the class web.

7. When you are done, kill `vtwm`. (Typically `Exit` in the left-click-on-root menu.) This will log you out of the ACL machine and bring the chooser back up. (Don't do this yet.)
8. To exit the ubuntu virtual machine select the *Shutdown* button in the top tab. For safety, you should let the virtual shutdown completely before quitting Parallels.
9. To quit Parallels you can either select the *Parallels*→*Quit* menu item or just type `<Splat-Q>`.
10. Don't forget to log off of the Mac as well.

### Obtaining and editing the source code

11. For your own sanity, you should create a sub-directory in your home directory to keep your CS256 class work in. In the `Little Shell` window, type:<sup>1</sup>

```
bash$ mkdir cs256
bash$ chmod go-rx cs256
bash$ cd cs256
```

The first of the commands creates the directory `cs256`. (`mkdir` should be read “make directory”.) The second changes the access modes so that others (`go` means “others in your group” and “others”—access modes for “others” are sometimes referred to as “world” access modes) cannot read the directory or access the files in it (`-rx` means “disallow reading or using” the directory). (`chmod` should be read “change modes”.) Finally `cd cs256` moves you into the new directory to work. (`cd` should be read “change directory”.)

12. The source code for this lab is provided in my class directory (which is world readable). You should copy this into your directory

```
bash$ cp ~/jrogers/cs256/lab1/div.cpp .
```

Don't miss the `.` at the end of the line or the tilde (`~`) at the beginning. (`cp` should be read “copy”. The `.` should be read “here”. The `~/jrogers/` refers to my home directory. Just `~` by itself or `~<your-user-id>` would refer to your own home directory.)

13. Start up `emacs` (in the left-click-on-root menu) and open `div.cpp`. (Don't forget it is in your `cs256` directory. You should `Find` the file `~/cs256/div.cpp`.) Since the source is provided for you, you don't need to edit it yet, but I strongly recommend compiling under `emacs` since it makes it easier to track your compilation errors.

---

<sup>1</sup>A note on typefaces. In the examples, the shell's prompt is `bash$`, your prompt may differ. Other program output is also rendered in the `typewriter` font. Input from the user (that would be you) is rendered in *slanted typewriter* font.

14. I recommend that you enable **syntax highlighting** in your source buffers. This will use color to emphasize the syntax of your program. (It’s a great way to find missing ‘’-marks.) This should already be enabled for you, but if it isn’t you can enable it from the `Options` menu.

## Compiling

15. To compile the program, select `Compile...` from the `C++` menu.
16. `emacs` will prompt you with the default compiler command line in the “mini-buffer” at the bottom of the window:

```
Compile command: g++ -Wall -ggdb div.cpp -o div
```

This is the actual command line it will use to compile the file.<sup>2</sup> (You can use this command yourself in a shell window.) You can modify this command if you need to (and you often times will). To accept it just type `(Return)`. Deciphering: “`g++`” is the name of the compiler (also known as `c++`, don’t confuse it with `gcc`—the Gnu C compiler). “`-Wall`” asks for all Warnings. While it takes a little time to get used to deciphering the compiler’s warnings, they are generally quite good at pinning down problems with your code. You should **always** compile with the `-Wall` option and should **never** simply ignore a warning you do not understand. “`-ggdb`” asks for code supporting the `gdb` debugger to be included in the output file. “`div.cpp`” is the name of the source program. And, finally, “`-o div`” asks for the compiled program to be saved in the file `div`. This should be the same as the base filename (the part before the “`.cpp`”) of the source program in order for the debugger to find it.

17. If there are errors (there shouldn’t be here), you can let `emacs` locate the error for you using `Next Error` from the `C++` menu. You can do the same thing by middle-clicking<sup>3</sup> on the line displaying the error in the `*compilation*` buffer. One of the advantages of compiling under `emacs` is that if you modify the source file and then select `Compile...` again, it will notice that you have not saved your changes before compiling and ask you if you would like to.
18. Once you have successfully compiled the program you can run it by typing, in the shell window,

```
bash$ ./div
```

The `./`, again, refers to the current directory. `unix` interprets this as a request to find the file and attempt to execute it. The `./` is necessary because, well-configured systems do not automatically look in the current directory for executable files.

---

<sup>2</sup>If your command line differs from this, your user initialization files are probably not up to date. Ask me or one of the TAs to help you correct this.

<sup>3</sup>On the Macs, a three button mouse is simulated holding the `(Splat)` key down while clicking for middle-click (Mouse-2) and holding the `(Option)` key while clicking for right-click (Mouse-3). (Yes, these seem backwards to me too.)

## Using gdb

19. To start the debugger on your program you can select **Debugger...** from the **C++** menu. This will open a buffer for the debugger and prompt you with the default debugger command line (there may be some minor variations in the verbiage):<sup>3</sup>

```
Run gdb (like this):  gdb --annotate=3 div
```

Again, you should accept this with **(Return)**. This will split your window into five sub-windows, something like:

```
File Edit Options Buffers Tools Gud Complete In/Out Signals Help
p p* [GDB icons]
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Found
License GPLv3+: GNU GPL version 3 or 1
This is free software: you are free to
There is NO WARRANTY, to the extent pe
and "show warranty" for details.
This GDB was configured as "i486-linux
(gdb)
** *gud-div* Bot L9 (Debugger-u:%* *locals of div* A11 L1 (Local
|*****
|* File: div.cpp *
|* jrogers, Jan 2003 *
|* *
|* A driver for the intdiv integer division function *
|* The subject program for Lab 1 of CS-256, Spring 2003. *
|* *
|*****/
|#include <stdlib.h>
|#include <iostream>
|#include <cassert>
|using namespace std;
|
|*****
|* Function: int intdiv(int numerator, int denominator) *
|* *
|* *
|* Preconditions: numerator >= 0 ... *
|*****
|div.cpp Top L1 (C++/1 Abbrev)
|No stack.
|No breakpoints or watchpoints.
-u:%* *stack frames of div* A11 L1 -u:%* *breakpoints of div* A11 L1
```

Your source file will be open in the middle window. The upper-left window is the gdb interaction window (which is named, for obscure reasons, `*gud-div*`. This is the window that will initially be selected (i.e., the window in which your cursor will be active). You can switch to a different window by left-clicking the mouse in a blank section of its mode line (the bottom border of the window). The other three windows will come into play as we proceed.<sup>4</sup>

<sup>4</sup>You can browse the manual for `emacs`'s gdb graphical interface by selecting **Search Documentation**→**Look**

These windows will be easier to use if you make the whole `emacs` window wider. You can do this either by left-clicking on the side of the frame and dragging it or by left-clicking on the resize button, the rightmost button on the tab, and dragging the mouse over the side of the frame.

20. Set a breakpoint at the `intdiv` function:

```
(gdb) break intdiv
Breakpoint 1 at 0x80489d6: file div.cpp, line 28. (gdb)
```

The first number is the number of the breakpoint. The `0x...` is the hexadecimal address of the corresponding machine instruction. (The actual number may vary.) The `line` is the line number in the source, which may vary depending on whether one or both of us has modified the program.

The breakpoint will also show up in the bottom-right window, the `breakpoints` window.

```
Num Type Disp Enb Address What
1 breakpoint keep y 0x080489d6 in intdiv(int, int) at div.cpp:28
```

(Note that the line will likely extend past the right edge of the window, which will be indicated with a `→` at the end of the line. You can scroll the line by left-clicking in it and moving the cursor to the right along the line.)

You can also set a break point at a particular line in the source program that is being displayed in the middle window by left-clicking beside that line on the gray border between the scrollbar and the rest of the window. If you scroll the `div.cpp` window down to show the `intdiv` function you should see a red dot in this gray border beside the first line of the function.

21. To run the program type (in the `*gud-div*` window):

```
(gdb) run
Starting program: /clients/users/<your-name-here>/cs256/div
Numerator:
```

Initially, the program stops waiting for input. Enter appropriate data:

```
52
Denominator:
17
```

```
Breakpoint 1, intdiv(int,int) (numerator=52, denominator=17) at div.cpp:28
(gdb)
```

---

up `Subject` in `User Manual` from the `Help` menu, entering `GDB` (all caps) at the `Subject` to look up: prompt and then left-clicking on the `GDB Graphical Interface` Menu item at the bottom of the buffer. But you should **not** do that now. (If you have already done it, then just reselect the `*gud-div*` buffer from the `Buffers` menu to get back to where you belong.)

The program stops at the first line of the function, which is the first local declaration, and the `emacs` moves the mark in the source buffer to the corresponding line of the source code. At this point `gdb` is in control and is waiting for a command.

**Remember: when you see the (gdb) prompt you must enter a gdb command; when the cursor just sits on the left with no prompt (or with your program's prompt) the program is either waiting for input or working (perhaps looping infinitely?).**

22. You can step through the program one line at a time using:

```
(gdb) next
(gdb)
```

This will move the mark in the source buffer to the next line. Note, also, that the values of the local variable `quotient` and `product` are displayed in the upper-right window, the `locals` window. Before we executed the first line of the function, which declares them, their value was uninitialized, which is to say it was whatever value happened to have been left in the memory. When we executed that line they got their initial value, 0.

23. You can examine the value of variables, even of expressions using `print`:

```
(gdb) print numerator
$1 = 52
(gdb) print numerator/denominator
$2 = 3
(gdb)
```

24. The values in the `locals` window will be updated as you step through the program. You can also ask `gdb` to display the values of variables every time it stops with `display`:

```
(gdb) display product
1: product = 0
(gdb) display quotient
2: quotient = 0
(gdb) next
2: quotient = 0
1: product = 0
(gdb)
```

(This is redundant when running `gdb` under `emacs` but will be useful if you run it from the command line, i.e., directly in a shell window.)

25. Continue to enter `next` until `quotient` gets incremented. When at the (gdb) prompt you can repeat the last `gdb` command by simply typing `<Return>`. You can recall lines you entered previously using `<Control><Up>` or `<Splat>P`.

...

```
(gdb) next
2: quotient = 0
1: product = 0
(gdb) next
2: quotient = 1
1: product = 0
(gdb)
```

You should be at the `product += denominator;` line of the source program. Note that when gdb “breaks” (not to be confused with being broken), the displayed line has not yet been executed; it is about to be.

26. You can remove a breakpoint by deleting it by number.

```
(gdb) delete 1
(gdb)
```

You can also delete a breakpoint by left-clicking on it in the `breakpoints` window and typing `D` (capital ‘D’).

27. The `finish` command runs the program until the current function returns:

```
(gdb) finish
Run till exit from #0 intdiv(int,int) (numerator=52, denominator=17) at div.cpp:44
0x08048b84 in main () at div.cpp:80
Value returned is $3 = 3
(gdb)
```

The mark in the source buffer should be at

```
cout << num << "/" << den << " = " << intdiv( num, den ) << endl;
```

We are actually in the middle of this line, having just returned from the function call. You can set a breakpoint by line number as well. Set a breakpoint at the current line. The line number is included in the message that gdb prints when it breaks. It is also displayed in the status line at the bottom of the `div.cpp` buffer as `L##`. (For me it is 80, for you it may be something different.)

```
(gdb) break 80
Breakpoint 2 at 0x8048b72: file div.cpp, line 80.
(gdb)
```

28. You can return control to the program (until the next breakpoint) with:

```
(gdb) cont
Continuing.
52/17 = 3
Another?
```

29. Say ‘y’ and enter new data. The program should break at the `cout` line.

```
y
Numerator
17
Denominator
52
```

```
Breakpoint 2, main () at div.cpp:80
(gdb)
```

30. There is a `step` command which is similar to `next` except that it will step “into” the body of a function—if the current line invokes a function, it will go to the first line of that function. (Fortunately, this will skip over most library functions, such as ‘<<’.)

```
(gdb) step
intdiv(int,int) (numerator=17, denominator=52) at div.cpp:28
2: quotient = -1073744668
1: product = 1074739082
(gdb)
```

(The values for `quotient` and `product` will be arbitrary, since they have not yet been initialized.) Note that we have stopped at the entry to `intdiv`—not because of the breakpoint (which we deleted) but, rather, because we have stepped into the function. Step once more, to execute the initialization of `quotient` and `product`:

```
(gdb) step
2: quotient = 0
1: product = 0
(gdb)
```

31. A **watchpoint** is like a breakpoint set on a variable or expression. It breaks the program whenever the value of the variable or expression changes.

```
(gdb) watch product
Hardware watchpoint 3: product
(gdb) cont
Continuing.
Hardware watchpoint 3: product
```

```
Old value = 0
New value = 52
intdiv (numerator=17, denominator=52) at div.cpp:36
2: quotient = 1
1: product = 52
(gdb)
```

The displayed line is actually the next line to be executed, which will be the next line in the flow of the program **following** the update of `product`. Here that is the head of the `while` loop, since the update of `product` is the last line of the body of the loop.

32. Delete the watchpoint (it should be number 3).

```
(gdb) delete 3
(gdb)
```

33. Continue, reply 'y' to the "Another?" prompt, and ask for the quotient of 100 and 27.

```
(gdb) cont
Continuing.
17/52 = 0
Another?
y
Numerator
100
Denominator
27
```

```
Breakpoint 2, main () at div.cpp:80
(gdb)
```

34. Step into the function again.

```
(gdb) step
intdiv (numerator=100, denominator=27) at div.cpp:28
2: quotient = -1073744668
1: product = 10738174720
(gdb)
```

35. By 'watch'ing well chosen expressions you can watch what happens to your loop invariants as the loop body progresses.

```
(gdb) watch product == quotient * denominator
Hardware watchpoint 5: product == quotient * denominator
(gdb) cont
Continuing.
Watchpoint 4: product == quotient * denominator

Old value = false
New value = true
intdiv (numerator=100, denominator=27) at div.cpp:29
```

```
2: quotient = 0
1: product = 0
(gdb)
```

Note that the initial values for `product`, `quotient` and `denominator` make the invariant true.

```
(gdb) cont
Continuing.
Hardware watchpoint 4: product == quotient * denominator

Old value = true
New value = false
intdiv (numerator=100, denominator=27) at div.cpp:44
2: quotient = 1
1: product = 0
(gdb)
```

When `quotient` is updated it becomes false.

```
(gdb) cont
Continuing.
Hardware watchpoint 4: product == quotient * denominator

Old value = false
New value = true
intdiv (numerator=100, denominator=27) at div.cpp:36
2: quotient = 1
1: product = 27
(gdb)
```

Only to become true again when `product` is also updated. And thus the invariant is first disturbed and subsequently restored during the body of the loop.

36. You can delete all breakpoints with:

```
(gdb) delete
delete
Delete all breakpoints? (y or n) y
(gdb)
```

37. Continue the program and allow it to terminate.

```
(gdb) cont
Continuing.
100/27 = 3
```

```
Another?
```

```
n
```

```
Program exited normally.
```

```
(gdb)
```

38. At this point you can either **run** the program again or exit the debugger with **quit**.
39. Explore the debugger. Try setting other breakpoints and watchpoints. Try printing arbitrary expressions. What does *until* do?

## Interrupting lost programs

One of the great advantages to running your program under **gdb** when you are debugging it is that you can use it to figure out where your program has gone when it wanders off into oblivion.

40. Run the `div` program again. This time enter `100` for the numerator and `0` for the denominator.

```
(gdb) run
Starting program: /clients/users/<your-name-here>/cs256/div
Numerator:
100
Denominator:
0
```

Note that there is no **(gdb)** prompt—the program is off being busier than it has any right to be.

41. To find out what is going on, interrupt the program by selecting **BREAK** from the **Signals** menu. (You can also just type **<Ctrl>-c <Ctrl>-c**.<sup>5</sup>)

```
Program received signal SIGINT, Interrupt.
(numerator=100, denominator=0) at div.cpp:41
2: quotient = -1793786562
1: product = 0
```

The line number and the value of `quotient` will likely vary, but the value of `quotient` should look kind of suspicious. If all we ever do to it is to add 1, how did it get negative? (In your case it may not be negative but it is likely to be very large if it is not.)

42. To see what is going on, go through the loop one line at a time using **next** (or **step**).

---

<sup>5</sup>Note that **<Ctrl>-c** by itself is the key you use to interrupt your program when running from the command line. Most of the control keys you can use to signal your program from the command line can be sent to your program under **gdb** by typing **<Ctrl>-c** followed by the usual key. In this case **<Ctrl>-c** itself can be sent with **<Ctrl>-c <Ctrl>-c**.

```
(gdb) next
2: quotient = -1793786562
1: product = 0
(gdb) next
2: quotient = -1793786561
1: product = 0
(gdb) next
2: quotient = -1793786561
1: product = 0
(gdb)
```

43. At this point you should have a pretty good idea of what is going on. There is, in fact, a precondition missing from this function. Figure out what it is.
44. You need to patch the program and recompile it. The cleanest way to do this is to open a new frame (**New Frame** from the **File** menu), select the source buffer in that frame (**div.cpp** from the **Buffers** menu), do your editing there and then recompile (**Compile...** from the **C++** menu). If you do this directly in the source window of the frame you are running **gdb** in, the **\*compilation\*** buffer will replace the **\*gud-div\*** buffer. You can get it back by selecting it in the **Buffers** menu.

In patching **div.cpp**, remember to add your new precondition to those listed in the documentation as well as adding an **assert** of the precondition to the beginning of the function body.

45. Return to the **\*gud-div\*** window and run the program again.

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
'/clients/users/<your-name-here>/cs256/div' has changed; re-reading symbols.
Starting program: /clients/users/<your-name-here>/cs256/div
Numerator:
```

Note that **gdb** recognizes that the executable has changed and reloads it.

Enter **100** and **0** for the **Numerator** and **Denominator** again.

```
100
Denominator:
0
div: div.cpp:30: int intdiv(int, int): Assertion '<your-precondition-here>' failed.
Program received signal SIGABRT, Aborted.
0xb7f4d410 in __kernel_vsyscall ()
(gdb)
```

The message about the assertion failing you will get whether you are running under **gdb** or not. The advantage of being under **gdb**, here, is that you can use it to find out why the calling program failed to satisfy the precondition before calling the function.

46. To find out how we got to where we are we will explore the program's stack which is displayed in the `*stack frames` of `div*` window:

```
#0 0xb7f4d410 in __kernel_vsyscall ()
#1 0xb7cf1085 in raise () from /lib/tls/i686/cmov/libc.so.6
#2 0xb7cf2a01 in abort () from /lib/tls/i686/cmov/libc.so.6
#3 0xb7cea10e in __assert_fail () from /lib/tls/i686/cmov/libc.so.6
#4 0x08048a38 in intdiv (numerator=100, denominator=0) at div.cpp:30
#5 0x08048bae in main () at div.cpp:81
```

You can also ask `gdb` to display this using the `backtrace` (you can abbreviate this `bt`) command:

```
(gdb) bt
#0 0xb7f4d410 in __kernel_vsyscall ()
#1 0xb7cf1085 in raise () from /lib/tls/i686/cmov/libc.so.6
#2 0xb7cf2a01 in abort () from /lib/tls/i686/cmov/libc.so.6
#3 0xb7cea10e in __assert_fail () from /lib/tls/i686/cmov/libc.so.6
#4 0x08048a38 in intdiv (numerator=100, denominator=0) at div.cpp:30
#5 0x08048bae in main () at div.cpp:81 (gdb)
```

This lists every function your program has called on the way to this crash starting with the `__kernel_vsyscall` function (at #0) that actually killed it and working its way back to the `main` function which is the function the system calls to start your program. The numbers represent **stack frames** the place where the variables local to corresponding function are stored. You can look at what was going on in any one of these functions by selecting its stack frame. The last (i.e., top) few are not interesting, since they are just tracking the inevitable path to death. The ones that are interesting are #4—the offending call to `intdiv`—and the one just below it, since that is the place that `intdiv` was called without properly satisfied preconditions.

47. You can select this frame by clicking on it in the `stack frames` window or by using the `fr` command:

```
(gdb) fr 5
#5 0x08048bae in main () at div.cpp:81
(gdb)
```

(The line number may be different for you, since we have edited the source program.)

The mark in the source buffer is again at

```
cout << num << "/" << den << " = " << intdiv( num, den ) << endl;
```

48. You can then examine the state of the variables that are being passed to `intdiv`.

```
(gdb) print num
$1 = 100
(gdb) print den
$2 = 0
(gdb)
```

49. In this way you can figure out why the precondition was not met.

### Print out and typescripts

50. To print a file from emacs, select one of the print commands from the **File** menu. I recommend using **Postscript Print Buffer (B+W)**. This will preserve the syntax highlighting, but will do the highlighting so it shows up well on a black and white printer. To print only a portion of a buffer (your `*gud-div*` buffer, for example), select a region of the buffer by left-clicking and dragging the mouse over the region (the window will scroll for you) or by setting the mark at one end of the region (using `<Ctrl>-<space>` or `<F1>`) and positioning the point (the cursor) at the other end, and then selecting **Postscript Print Region (B+W)** from the **File** menu.
51. To print a file from the command line use either:

```
bash$ a2ps filename
```

or

```
bash$ a2ps -1 filename
```

(where filename is the name of the file).

The first of these will, by default, print the file two-up (two pages per page) which is good for working copies and your own records. The second command will print a single page per page, which is good for copies you are turning in to be graded. **Generally, you should print your source files with `a2ps -1 filename`. This is because `a2ps` includes more information in the page headings than the emacs print commands do.**

52. By default, these will print to your default printer, most likely the printer in the Math/CS lounge. To get these to print out to the printer in the Mac lab, you should use

```
bash$ a2ps -Pd224 filename
```

53. You don't have this option when printing from emacs, which will always print to your default printer. You can set your default printer to d224 (**before** starting emacs) with

```
bash$ export PRINTER=d224
```

54. You can print out transcripts of your sessions by generating a typescript:

```
bash$ script
Script started, output file is typescript
bash$
```

This will copy everything that shows up in your shell window into the file “typescript”. (This is literally everything—including backspaces and arrow keys. These don’t show up well at all, so try to avoid them.) When you are done type

```
bash$ exit
exit

Script done, output file is typescript
bash$
```

You can then print typescript as above.

## Assignment

Please submit three things:

1. A printed copy of your source code with the new precondition and assertion .
2. A transcript of a (short) gdb session in which you cause an assert associated with a precondition to fail.
3. Either a transcript of a (short) gdb session (perhaps the same one) in which you cause an assert associated with a post-condition to fail or a (brief) explanation of why it can’t be done.

This document is based, in part, on John Howell’s lab *How to create and run a program* for CS-35 and also on Michael Main and Walter Savitch’s example *Data Structures—Lab Exercise 2*.