V22.0474-001 Software Engineering
Spring 2008

Lecture 18: Debugging with gdb

Clark Barrett, New York University

Project Status Check

- Homework 7 is due on Tuesday
- On April 3, class will be held in 512 WWH
- *Homework 8 (testable working code) is due Apr. 10*
- The remaining homeworks have been posted (take a look)
- Spring Showcase (May 7)

Debugging macros

In the sample project, the header file `src/include/debug.h` together with the cpp file `src/generic/debug.cpp` contain several useful debugging macros:

- `FatalAssert(cond, msg)`
  Check `cond`. If true, continue. If false, exit with message `msg`.

- `DebugAssert(cond, msg)`
  In `DEBUG` executable, does the same as `FatalAssert`. In `OPTIMIZED` executable, does nothing (doesn't even generate any code).

- `IF_DEBUG(code)`
  Code that gets executed only in `DEBUG` executable.

- `DebugPrint(cond, msg)`
  In `DEBUG` executable, if `cond` is true, print message `msg`. In `OPTIMIZED` executable, does nothing.

Error Handling

I suggest the following breakdown of error handling:

1. *User errors* You should check for errors that can occur as the result of mistakes made by the user. Your code should print a reasonable error message and either exit gracefully, or (preferably), continue and allow the user to try again. *Exceptions* are often a good way to implement this kind of functionality.

2. *Code invariants* You should use DebugAssert (or something equivalent) to document code invariants. These are conditions that should *always* be true, even if the user does something wrong. If a code invariant fails, it indicates an unexpected program state. A good place to use assertions is to check function preconditions and postconditions.

3. *Extremely rare errors or unimplemented code* I use *FatalAssert* for extremely rare errors. These are errors that I don't expect will ever happen in practice, but if they do, I want to know about it. Some examples might be unreachable code, out of memory errors, or counter overflows. I also sometimes use *FatalAssert* for stubs that aren't functional yet.

Debugging using gdb

GDB is the *GNU Project Debugger*.

A helpful gdb reference has been posted on the web syllabus next to lecture 18.

To use gdb, you must compile with the `-g` flag. This is done automatically if you are using the default debug build from the advanced Makefile.

To start gdb, type `gdb`. To start gdb from emacs, type `M-x gdb`. Note that the emacs buffer is called `*gud*`.

Debugging using gdb

Essential Commands

- `file` *`filename`*: Load the executable located at *filename*
- `b` *`[file:]line`*: Set breakpoint at *line* (optionally, in *file*); in emacs, you can do this by moving to the line and typing `C-x SPC`
- `run` *`args`*: Run the current executable with optional arguments *args*
- `bt`: Display program stack
- `up`, `down`: Move up and down program stack
- `p` *`expr`*: Print the value of an expression *expr*
- `set` *`var=expr`*: Set *var* to new value *expr*
- `c`: Continue execution
- `n`: Next line, stepping over function calls
- `s`: Next line, stepping into function calls
- `fin`: Finish current function call

Debugging using gdb

More Useful Commands

- `info b`: List breakpoints and number of times hit
- `ignore` *`n count`*: Ignore the next *count* occurrences of breakpoint *n*
- `disable` *`n`*: Disable breakpoint *n*; similar commands: `enable`, `delete`
- `cond` *`n [expr]`*: Make breakpoint *n* conditional on *expr* (unconditional if no *expr*)
- `commands` *`n`*: Type commands to execute every time breakpoint *n* is hit; type `end` when done
- `until`: Run until execution reaches next line (useful for getting out of loops)
- `return` *`[expr]`*: Force current function to return (optionally setting return value to *expr*
- `signal` *`num`*: Continue with signal nụm
- `jump` *`line`*: Continue execution at line *line*
- `call` *`function`*: Call function *function*

Cool gdb tricks

I want to stop at a certain point, but only under certain conditions.

Cool gdb tricks

I want to stop at a certain point, but only under certain conditions.

Meet *conditional breakpoints*

1. Set breakpoint *n* at the place you want to stop
2. Type `cond n expr`, where *expr* is the condition under which you want to stop
3. For example, `cond 1 x > 100` makes breakpoint 1 conditional: execution only stops if `x` is greater than 100

Cool gdb tricks

My program crashes and I really need to see the value of a variable a few lines before it crashed, but a conditional breakpoint doesn’t work/is too slow.

Cool gdb tricks

My program crashes and I really need to see the value of a variable a few lines before it crashed, but a conditional breakpoint doesn't work/is too slow.

1. Set breakpoint *n* at the place you want to stop
2. `ignore` *`n big-number`*: tell gdb to ignore *n* a huge number of times, like 1000000
3. `run`
4. When the program crashes, type `info b` to see how many times breakpoint *n* was hit; say it was hit *c* times
5. `ignore` *`n c-1`*
6. `run`
7. Presto: you're at the breakpoint right before it crashes

Cool gdb tricks

My program crashes/assert-fails and I want to see what would have happened next.

Cool gdb tricks

My program crashes/assert-fails and I want to see what would have happened next.

1. Set a breakpoint at `debugError` in `debug.cpp` (if you are using my debugging macros; if not, set a break right before the crash somehow)
2. Use `return 0` to get to the stack frame where things went wrong
3. Set a breakpoint at the next line you want to execute (line *n*)
4. Use `jump` *n* to jump to that line
5. Continue with `c` (if there is a pending exception, you may have to use `signal 0` to continue)

Cool gdb tricks

My program crashes/assert-fails and I want to see what would have happened next.

1. Set a breakpoint at `debugError` in `debug.cpp` (if you are using my debugging macros; if not, set a break right before the crash somehow)
2. Use `return 0` to get to the stack frame where things went wrong
3. Set a breakpoint at the next line you want to execute (line *n*)
4. Use `jump` *n* to jump to that line
5. Continue with `c` (if there is a pending exception, you may have to use `signal 0` to continue)

Note: if you want to be really clever, you can skip step 1: replace `exit(1)` in `debugError` by `cerr << 0/0;`. Now, instead of exiting, the program just crashes with an arithmetic exception and leaves you in the function where it crashed.

Cool gdb tricks

There’s a bad assertion/statement where my program crashes, but I want to continue debugging without having to recompile/rerun the program.

Cool gdb tricks

There's a bad assertion/statement where my program crashes, but I want to continue debugging without having to recompile/rerun the program.

1. Set breakpoint *n* at the offending statement
2. Type `commands` *n*
3. Type `jump` *k* where *k* is the line after the offending statement
4. Type `end`
5. Continue debugging; the offending statement will automatically be skipped

Cool gdb tricks

There's a bad assertion/statement where my program crashes, but I want to continue debugging without having to recompile/rerun the program.

1. Set breakpoint *n* at the offending statement
2. Type `commands` *n*
3. Type `jump` *k* where *k* is the line after the offending statement
4. Type `end`
5. Continue debugging; the offending statement will automatically be skipped

By elaborating on this basic concept, you can write and test whole new algorithms without ever exiting your debugging session. This can be useful in a big project where it takes hours or days to get to the bug!

The debug/compile/test loop in emacs

You can edit, compile, and debug without ever leaving emacs:

1. Make some changes to your code
2. Type `M-x compile` and then enter the compile command (i.e. `make`)
3. If there are compile errors, type `C-x ‘` to automatically cycle through the errors
4. After compiling, type `M-x gdb` to start gdb (or `C-x b *gud*` to switch to the gdb buffer if it is already running)

The debug/compile/test loop in emacs

You can edit, compile, and debug without ever leaving emacs:

1. Make some changes to your code
2. Type `M-x compile` and then enter the compile command (i.e. `make`)
3. If there are compile errors, type `C-x ‘` to automatically cycle through the errors
4. After compiling, type `M-x gdb` to start gdb (or `C-x b *gud*` to switch to the gdb buffer if it is already running)

Note: the `M-x compile` command only compiles the current library. If you want to rebuild everything, you will need to do a `make` in the root project directory.

Also, unit test executables don’t automatically detect when the main library has changed. If you change code in the library but not in the unit test, you will need to remove the unit test executable and re-link.