Outline

- Project Status Check
- Review: Sample Project and Advanced Makefiles
- Debugging with \texttt{gdb}
- Checking line coverage with \texttt{gcov}

Project Status Check

Project Assignments

1. Proposal
2. Requirements
3. Design
4. Design Review
5. Revised Requirements and Design
6. Testing
7. Final Report

Project Presentations

1. Requirements
2. Design
3. Testing
4. Final Presentation

Upcoming assignments

- Revised Requirements and Design documents are due Monday
- \textit{You must have code that another group can test by one week from Monday (April 18).}
Review

Sample Project

- Check it out:
  ```bash
cvs -d /home/barrett/public/repository co sample_project
  ```
- Browse it online:
  Click on **Sample Project** from the course web page.

To build the sample project on a CIMS sun machine, edit `Makefile.local` and change the `TOP` variable to point to your local working copy.

Then type `make`.

---

Review

TousethisMakefilesystemforyourowncode,youmustdothefollowing:

- Mimic the directory structure of the sample project:
  ```
  project.project/bin.project/src,
  project/src/include.project/src/main,
  project/src/module1,project/src/module2,...
  ```
- Copy files from the sample project: `Makefile, Makefile.local, Makefile.std, bin/libmerge, src/Makefile`
- Edit `Makefile.local` and change the options as appropriate for your project
- Edit `src/Makefile` to contain your header files and modules
- Create a `Makefile` for each module similar to the ones in the sample project (note the difference between a `library` module such as `expression` and an `executable` module such as `main`).

---

Review: Advanced Makefiles

What can you do with an advanced Makefile?

- **Easy to add new source files.** Module files are listed in `src/module/Makefile`. Public header files (in `src/include`) are listed in `src/Makefile`. New modules must be added to module list in `src/Makefile`.

- **Support for multiple platforms.** To compile for a different platform (i.e. object files, libraries, and executables are stored separately), change the `PLATFORM` variable in `Makefile.local` or type `make PLATFORM=newPlatform`.

- **Support for multiple build-types.** Default build is `debug`. To compile the optimized version, uncomment the `OPTIMIZED=1` line in `Makefile.local` or type `make OPTIMIZED=1`.

---

Review: Advanced Makefiles

**Support for unit testing.**

1. Create test directory `src/module/test`
2. Write unit test in `src/module/test/test.cpp`
3. Create local `src/module/test/Makefile` (see example in `sample_project/src/expression/test`)
4. Type `make` in `src/module/test` to build unit test

**Support for source searching in `emacs`**

- To find a class declaration:
  1. Move the cursor onto the class name
  2. Type `M-`
  3. Make sure `emacs` has the right class name and press return
  4. Give path to tags table: `project/` and press return
  5. To go back to where you were, use `M-x b`

- To search through all files in your project
  1. Type `M-x tags-search`
  2. Enter the expression to search for
  3. If you haven’t before, give the path to the tags table: `project/`
  4. Use `M-`, to “find next”
**Review: Debugging macros**

In the sample project, the header file `src/include/debug.h` contains 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.

---

**Debugging using gdb**

**GDB** is the GNU Project Debugger.

A helpful gdb reference has been posted on the web syllabus under lecture 20.

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
- `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

**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 `num`
- `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.

Meet conditional breakpoints

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

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

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

1. Set a breakpoint at \texttt{debugError} in \texttt{debug.cpp} (if you are using my debugging macros; if not, set a break right before the crash somehow)
2. Use \texttt{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 \texttt{n})
4. Use \texttt{jump \text{n}} to jump to that line
5. Continue with \texttt{c} (if there is a pending exception, you may have to use \texttt{signal 0} to continue)

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

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

1. Set breakpoint \texttt{n} at the offending statement
2. Type \texttt{commands n}
3. Type \texttt{jump k} where \texttt{k} is the line after the offending statement
4. Type \texttt{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 \texttt{M-x compile} and then enter the compile command (i.e. \texttt{make})

3. If there are compile errors, type \texttt{C-x `} to automatically cycle through the errors

4. After compiling, type \texttt{M-x gdb} to start gdb (or \texttt{C-x b *gud*} to switch to the gdb buffer if it is already running)

Note: the \texttt{M-x compile} command only compiles the current library. If you want to rebuild everything, you will need to do a \texttt{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.

Using gcov to test code coverage

gcov works with g++ to figure out which lines of your program have been executed.

To use gcov, you must do the following:

1. Compile source files with \texttt{-fprofile-arcs -ftest-coverage}

2. Run the executable

3. For each source file \texttt{source.cpp} whose coverage you want to examine, type \texttt{gcov source.cpp}

4. Output from gcov is in the file \texttt{source.cpp.gcov}

Using the advanced Makefile from the sample project (get the latest version), you can do step one automatically by typing \texttt{make GCOV=1}.

To additionally get information on branches, use \texttt{gcov -b source.cpp}.

Using gcov as part of your testing strategy is a good idea.