

Source Level Debugging

- A source level debugger has a number of useful features that facilitates debugging problems associated with executing your program.
- You have to create symbolic table information within the executable by using the `-g` option when compiling with `gcc` or `g++`.
- This symbolic table information includes the correspondances between
 - statements in the source code and locations of instructions in the executable
 - variables in the source code and locations in the data areas of the executable

GDB: Gnu DeBugger

- GDB is a line oriented debugger where actions are issued by typing in commands.
- It can be invoked for executables compiled by `gcc` or `g++` with the `-g` option.
- General capabilities:
 - Starting/exiting your program from the debugger.
 - Stopping and continuing the execution of your program.
 - Examining the state of your program.
 - Changing state in your program.

Starting/Exiting GDB

- Bring up the `gdb` debugger by typing:
`gdb executable_name`
- Initiate your executable by using the command:
`run [command-line arguments]`
 - Command line arguments can include anything that would normally appear after the executable name on the command line.
 - run-time options, filenames, I/O redirection, etc.
 - When no arguments are specified, then `gdb` uses the arguments specified for the previous `run` command during the current `gdb` session.
- Exit the `gdb` debugger by typing:
`quit`

Stopping and Continuing the Execution of Your Program in GDB

- Setting and deleting breakpoints.
- Execution stepping and continuing.

Setting and Deleting Breakpoints

- Can set a breakpoint to stop:
 - at a particular source line number or function
 - when a specified condition occurs
- General form.
`break [linenum | function] [if condition]`
- Specifying a break command without any arguments causes a breakpoint to be set at the current line where execution has stopped.
- Can delete a breakpoint:
`delete breakpoint_number`

Examples of Setting and Deleting Breakpoints

```
(gdb) break                # sets breakpoint at current line
                           # number
(gdb) break 74             # sets breakpoint at line 74 in the
                           # current file
(gdb) break foo            # sets breakpoint when entering
                           # function foo
(gdb) break 60 if i == 10  # sets breakpoint at line 60 in the
                           # current file if the variable i has
                           # the value 10
(gdb) delete 3             # remove the third breakpoint
(gdb) delete               # remove all breakpoints
```

Execution Stepping in GDB

- Can step to the next statement or into a function that is being invoked.
- General form. n indicates the number of steps to perform or until the program encounters a breakpoint or terminates execution. By default only one step is performed.
`step [n]`
- Can step to the next statement and across any function that is being invoked.
- General form. n again is the number of steps.
`next [n]`

Continuing Execution in GDB

- You can continue execution in *gdb* up to encountering the next breakpoint or program termination.
- General form. The n indicates to ignore the current breakpoint at which the debugger has stopped $n-1$ times. If not specified, then n is 1 by default.
`cont [n]`

Continuing Execution Until Leaving a Loop

- The *until* command executes your program until it reaches a source line greater than the current line. If the program is not executing a backward jump, then it has the same effect as the *next* command.
- This command is useful for continuing execution on a backward jump until the program leaves a loop.

Examining the State of Your Program in GDB

- Listing source code.
- Printing the values of expressions.
- Displaying the values of expressions.
- Printing the trace of function calls.
- Switching context within the trace.

Listing Source Code

- You can list source code at a specified line or function.
- General form. You can optionally specify a filename before a line number or function name. You can specify a range of line numbers. By default ten lines are listed. If no arguments are specified, then ten lines are listed from the current program location or last line that was listed.

```
list [[filename:]linenum[,linenum] | [filename:]functionname]
```

Listing Source Code Examples

```
(gdb) list                # list 10 lines from current
                           # location
(gdb) list 72              # list lines 67..76 (10 lines
                           # around the specified line) in
                           # the current file
(gdb) list cal.c:55        # list lines 50..59 in cal.c
(gdb) list 80,95           # list lines 80..95
(gdb) list main            # list starting at function main
(gdb) list cal.c:jan1     # list starting at function jan1
                           #in file cal.c
```

Printing the Values of Expressions in GDB

- One can print the value of legal expressions. One can also specify the format for the value to be printed.
- General form. The *fmt* character can be d (decimal), u (unsigned), f (float), c (character), o (octal), C(character), and x (hexadecimal). When no expression is specified, it prints the value of the last expression specified in a print command. If no *fmt* is given, then it uses the type associated with the expression.

```
print[/fmt] [expression]
```

Example Print Commands

```
print i           # prints the value of variable i
print a[i]        # prints the value of a[i]
print/x a[i]      # prints a[i] in hexadecimal
print a[i]-b[i]   # prints the value of a[i]-b[i]
print a           # prints the values in the array a
print p           # prints the address contained in pointer p
print *p          # prints the value pointed to by p
print p->next     # prints a field of the struct pointed to by p
```

Displaying the Values of Expressions in GDB

- The display command is similar to the print command, but the expression is evaluated and printed after each time you step or continue in gdb.
- General form. All current expressions to be displayed will be reprinted if you do not give an expression as an argument.

```
display[/fmt] [expression]
```

Undisplaying Expressions in GDB

- Use the undisplay command to remove expressions from being displayed.
- General form. Undisplay with no argument removes all display expressions.

```
undisplay [display_number]
```

Printing the Trace of Function Calls in GDB

- You can print the trace of function calls corresponding to the activation records that were made to reach the current routine.
- This trace shows the names of the routines, the values of the arguments passed to each routine, and the line number last executed in each routine.
- One has the option to only print information for the last n calls. A negative n will print the first n calls.
- General form.
where $[n]$

Switching Context within the Trace

- One can switch contexts between functions up or down the call trace. This feature is handy for viewing the values of local variables not in the current function.
- General forms. By default n is 1.
up $[n]$ # go back n caller contexts
down $[n]$ # go forward n caller contexts

Changing State in Your Program

- You can assign values to variables during a debugging session. This can allow a user to test a potential fix without changing the source code and recompiling.
- General form.
set variable = expression
- Examples
set i = 5
set a[i] = 10

Making Calls from GDB

- You can make calls from the gdb prompt. It is very useful to write diagnostic routines that will print the values in dynamically allocated data structures since these structures are often difficult to access using the print command directly. Calls to functions can also be used to change the state of the program.
- General form.
call function_name(args)

Other Useful Features

- Can run a program that has a segmentation fault to easily find out exactly where the program dies.
- Can type *cntrl-C* within GDB when a program does not terminate to find out where the endless loop occurs.

Command Shortcuts

- You do not have to type the complete command. You only have to type enough characters to ensure that the command name is unique.
 - (gdb) r < in.dat # same as “run < in.dat”
 - (gdb) p n # same as “print n”
 - (gdb) l foo # same as “list foo”
 - (gdb) c # same as “cont”
 - (gdb) n # same as “next”
- Likewise you can apply command and identifier name completion using the tab character, just as you do command and filename completion in tcsh.

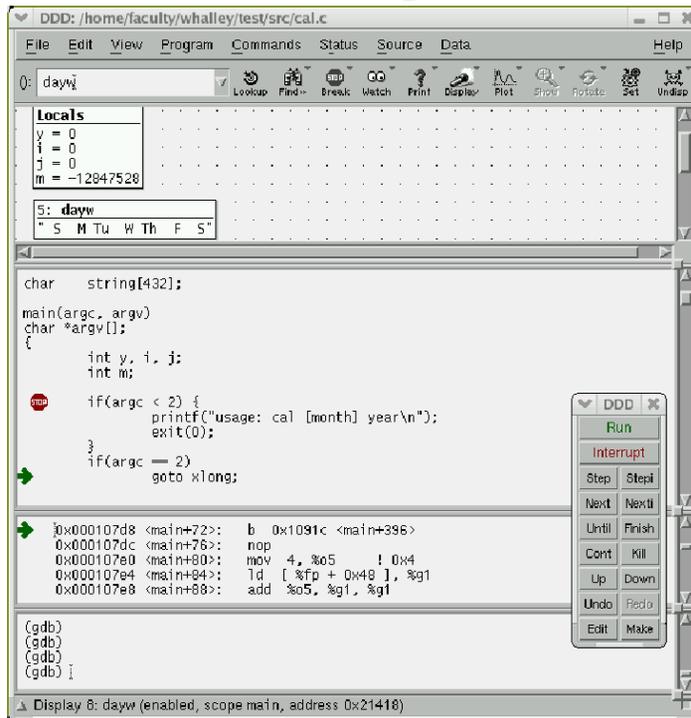
DDD: Data Display Debugger

- DDD is a graphical front-end for GDB and other command-line debuggers.
- From DDD you can invoke all of the commands available in GDB.
- In addition, it has a graphical user interface that:
 - Shows the source code where the program execution has currently stopped.
 - Allows you to select commonly specified options with the mouse.

DDD Interface

- DDD displays four different windows.
 - Data window to display variables.
 - Source window to display source code.
 - Machine code window to display disassembled machine code.
 - GDB console where conventional gdb commands can be typed.
- DDD also has other panels which include common commands that can be selected with a mouse click.

DDD Snapshot



Using the Console Window

- This console is setup based on the type of line oriented debugger that is being used.
- For the gdb console, any gdb command can be entered.
- Commonly used gdb commands can also be issued by clicking on the panel.

Using the DDD Source Window

- Can set a breakpoint by using the right mouse button and positioning the cursor to the left of a source code line.
- Can display the value of a variable by moving the mouse cursor over a variable.
- Can highlight a variable and select to print or display its value by using the options at the top.

Using the DDD Data Window

- To have a variable with its value appear in the data window as a display:
 - A user can highlight a variable in the source window and then click on the display button.
 - A user can double click on a variable in the source window.

Diff (11.1)

- The diff Unix utility compares two files and displays the differences between the two files. The differences are displayed with an ed-like notation indicating what changes are needed to modify the first file to make it similar to the second.
- Diff is very useful in shell scripts to detect differences between expected output and actual output.

Diff Output (11.1)

- Diff output consists of a list of changes.
- General form consists of a sequence of:
commands
lines
- Commands are of the form (a for append, c for change, and d for delete).
linenums [acd] linenums
- Lines from the first file are preceded by “<”.
Lines from the second file are preceded by “>”.

Diff Example

<u>tmp1.txt</u>	<u>tmp2.txt</u>	<u>tmp3.txt</u>
cat	cat	dog
dog	mouse	mouse
mouse		cow

```
% diff tmp1.txt tmp2.txt
```

```
2d1  
< dog
```

```
% diff tmp1.txt tmp3.txt
```

```
1d0  
< cat  
3a3  
> cow
```

```
% diff tmp2.txt tmp3.txt
```

```
1c1  
< cat  
---  
> dog  
2a3  
>cow
```

Patch (20.9)

- Patch is a Unix utility that takes diff output and applies the commands to make the first file have the same contents as the second.
- Updates to free software are typically accomplished using patch. Often the differences between versions of files is much smaller than the files themselves.

Gvimdiff

- The utility (g)vimdiff allows you to edit two or three files and visualize the differences.
- Changed and new lines are highlighted.
- Deleted lines are also depicted.
- This display is updated as changes are made to any of the files.
- General form.

```
gvimdiff file1 file2 [file3]
```

Cmp

- The cmp Unix utility just returns a status indicating if the files differ.

<u>Exit Status</u>	<u>Meaning</u>
0	The files were identical.
1	The files differed.
2	An error occurred.

- The cmp utility is often used when comparing two binary files and is typically faster than diff.

Configuration Management Systems

- Configuration management systems provide:
 - backup of versions of files
 - control access to shared files
- Best to keep only source files under configuration management as long as the other files can be automatically regenerated.

Brief History of Popular Free Configuration Management Systems

- SCCS (Source Code Control System)
 - Popular configuration management system.
 - Problem is that it stores the original file and builds the new versions by applying the changes.
- RCS (Revision Control System)
 - More efficient in that it stores only the changes and the current file is the complete one.
- CVS (Concurrent Version System)
 - Allows more than one person to edit the same file at the same time and allows merges.

Brief History of Popular Free Configuration Management Systems (cont.)

- SVN (SubVersion)
 - Changes are tracked per-change rather than per-file.
- Some other software configuration management systems support a decentralized approach rather than using a centralized repository.

Commonly Used SVN Commands

```
svnadmin create repository
# Creates a directory using the repository
# (repo) name. Use a full path as the repo
# name and place it in a different directory
# from the files you wish to put under SVN.
# Only needs to be done once.

svn checkout file://repository
# Allows a user to work in the repository.
# Only needs to be done once for each user.

svn add filename # Adds a file to the repo. File will be added
# on the next commit.

svn delete filename
# Deletes a file from the repo.

svn commit # Updates the repo with latest files in the
# cwd. You will be prompted for a comment
# describing the latest commit.
```

Commonly Used SVN Commands (cont.)

```
svn status # Shows status of files in repository and cwd
# by the first column of each listed file.
# ' ' indicates file has no changes.
# 'M' indicates file has been modified.
# '?' indicates file not under version control.
# 'A' indicates file has been added.

svn diff [filename] # Shows differences between file in cwd and
# repo.

svn log filename # Shows log messages about the specified
# file.

svn update [revnum]
# Updates current working directory (cwd)
# with latest files in the repo. Can be used to
# get up-to-date files when working with
# other users or to go back to a previous
# revision.
```

SVN Identification and Log Information

- Each time you perform an *svn commit* command, you are prompted for a message (comment) giving an overview of that change. Each version is identified by a revision number.
- You can see messages about the history of the revisions of a file by using the *svn log* command.
- You can go back to an older version by specifying a revision number when using the *svn update* command.

Finding More Information about SVN

- You can get online help by typing:
`svn help command_name`
- You can also look at tutorials that are available on the web.
<http://www.thegeekstuff.com/2011/04/svn-command-examples/>

Gprof

- The gprof Unix utility produces an execution profile of the call graph of a program.
- The profile is useful for finding out where most of the time is spent during the execution of your program.
- A developer can use this information to tune the time-consuming portions of a long-running program.

Using Gprof

- You can have a program instrumented to collect data that can be processed by *gprof* by using the `-pg` option when compiling with *gcc* or *g++*.
`g++ -pg filename.cpp`
- A *gmon.out* file will be produced as a side effect of the execution of your instrumented program.
- You can obtain the profile from the *gmon.out* file by running the following command:
`gprof -b`