

# EECS 22: Advanced C Programming

## Lecture 8 (TuTh)

Rainer Dömer

doemer@uci.edu

The Henry Samueli School of Engineering  
Electrical Engineering and Computer Science  
University of California, Irvine

## Part 1: Overview

- Warm-up Quiz
- Shared Libraries
  - Static object file archives
  - Dynamically loaded shared objects
- Make and Makefile
  - Rules
  - Dependencies
  - Application example **PhotoLab2**
    - Modules **FileIO, Age, Main**
    - **Makefile**
  - Advanced features

## Quiz: Question 1


- Which Linux command shows you the path to the current directory?
  - a) `cd`
  - b) `pwd`
  - c) `dir`
  - d) `ls`
  - e) `list`

EECS22: Advanced C Programming, Lecture 11

(c) 2017 R. Doemer

3

## Quiz: Question 1

- Which Linux command shows you the path to the current directory?
  - a) `cd`
  -  b) `pwd`
  - c) `dir`
  - d) `ls`
  - e) `list`

EECS22: Advanced C Programming, Lecture 11

(c) 2017 R. Doemer

4

## Quiz: Question 2


- Which of the following Linux commands renames file “homework1.c” into “text1.c”?
  - a) `rn text1.c homework1.c`
  - b) `rn homework1.c text1.c`
  - c) `rm text1.c homework1.c`
  - d) `mv homework1.c text1.c`
  - e) `mv text1.c homework1.c`

EECS22: Advanced C Programming, Lecture 11

(c) 2017 R. Doemer

5

## Quiz: Question 2

- Which of the following Linux commands renames file “homework1.c” into “text1.c”?
  - a) `rn text1.c homework1.c`
  - b) `rn homework1.c text1.c`
  - c) `rm text1.c homework1.c`
  -  d) `mv homework1.c text1.c`
  - e) `mv text1.c homework1.c`

EECS22: Advanced C Programming, Lecture 11

(c) 2017 R. Doemer

6

### Quiz: Question 3


- What is C *not*?
  - a) a structured programming language
  - b) a object-oriented programming language
  - c) a compiled programming language
  - d) a high-level programming language
  - e) a portable programming language

EECS22: Advanced C Programming, Lecture 11

(c) 2017 R. Doemer

7

### Quiz: Question 3

- What is C *not*?
  - a) a structured programming language
  -  b) a object-oriented programming language
  - c) a compiled programming language
  - d) a high-level programming language
  - e) a portable programming language

EECS22: Advanced C Programming, Lecture 11

(c) 2017 R. Doemer

8

## Quiz: Question 4

- What is the meaning of the following code fragment?


```
/* printf("C programming is great!\n") */
```

- a) it prints "C programming is boring!"
- b) it prints "C programming is great!"
- c) it is a syntax error because a semicolon is missing after the `printf()` statement
- d) it is the main function of the C program
- e) it is a comment ignored by the compiler

## Quiz: Question 4

- What is the meaning of the following code fragment?

```
/* printf("C programming is great!\n") */
```

- a) it prints "C programming is boring!"
- b) it prints "C programming is great!"
- c) it is a syntax error because a semicolon is missing after the `printf()` statement
- d) it is the main function of the C program
-  e) it is a comment ignored by the compiler

## Quiz: Question 5

- What is true about of the following compiler call? (Check all that apply!)

```
% gcc HelloWorld.c -Wall -ansi -o HelloWorld
```

- a) the GNU C Compiler is called to generate an executable program called `HelloWorld`
- b) the compiler will print warning and/or error messages about any non-ANSI compliance in the code
- c) the compiler will ignore all warnings
- d) the compiler will read the file `HelloWorld.c`
- e) the compiler will overwrite the `HelloWorld` file if it already exists

EECS22: Advanced C Programming, Lecture 11

(c) 2017 R. Doemer

11

## Quiz: Question 5

- What is true about of the following compiler call? (Check all that apply!)

```
% gcc HelloWorld.c -Wall -ansi -o HelloWorld
```

- a) the GNU C Compiler is called to generate an executable program called `HelloWorld`
- b) the compiler will print warning and/or error messages about any non-ANSI compliance in the code
- c) the compiler will ignore all warnings
- d) the compiler will read the file `HelloWorld.c`
- e) the compiler will overwrite the `HelloWorld` file if it already exists

EECS22: Advanced C Programming, Lecture 11

(c) 2017 R. Doemer

12

## Shared Libraries

- When a set of modules is useful for multiple applications, a *shared library* can be built for reusing the functionality provided by the modules
  - 1) *Static library*: Archive of object files (linked at compile-time)
    - Examples: `libc.a`, `libM.a`
    - Archive tools `ar` and `ranlib`
      - can combine multiple object files into linker archives
  - 2) *Dynamic library*: Shared object file (loaded at run-time)
    - Examples: `libc.so`, `libM.so`
    - Compiler `gcc`
      - can create position-independent code (`-fPIC`)
      - can create shared libraries (`-shared`)
      - can create executable files that load shared object files at run-time (`-Lpath -Xlinker -R -Xlinker path`)

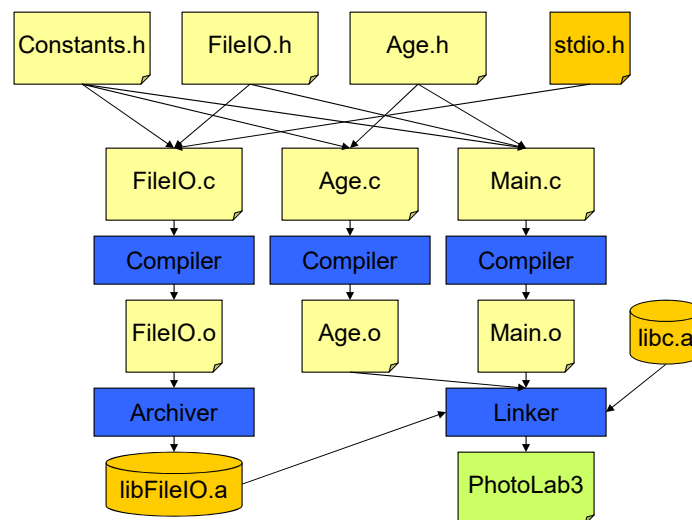
EECS22: Advanced C Programming, Lecture 11

(c) 2017 R. Doemer

13

## Shared Libraries

- Example: Shared `FileIO` Library for `PhotoLab3`



EECS22: Advanced C Programming, Lecture 11

(c) 2017 R. Doemer

14

## Shared Libraries

- Example: Shared `FileIO` Library for `PhotoLab3`
- Option 1: Static shared library
  - Preparing the archive file:

```
% vi Constants.h
% vi FileIO.h
% vi FileIO.c
% gcc -c FileIO.c -o FileIO.o -Wall -ansi -std=c99
% ar rc libFileIO.a FileIO.o
% ranlib libFileIO.a
%
```

- Building and compiling the application:

```
% vi Age.h
% vi Age.c
% vi Main.c
% gcc -c Age.c -o Age.o -Wall -ansi -std=c99
% gcc -c Main.c -o Main.o -Wall -ansi -std=c99
% gcc Main.o Age.o -lFileIO -L. -o PhotoLab3
% ./PhotoLab3
%
```

## Shared Libraries

- Example: Shared `FileIO` Library for `PhotoLab3`
- Option 2: Dynamically loaded shared library
  - Preparing the shared object file:

```
% gcc -c -fPIC FileIO.c -o FileIO.o -Wall -ansi -std=c99
% gcc -shared -fPIC -o libFileIO.so FileIO.o
%
```

- Building and compiling the application:

```
% gcc -c Age.c -o Age.o -Wall -ansi -std=c99
% gcc -c Main.c -o Main.o -Wall -ansi -std=c99
% gcc Main.o Age.o -lFileIO -L.
  -Xlinker -R -Xlinker . -o PhotoLab3
% ./PhotoLab3
% ldd PhotoLab3
    linux-vdso.so.1 => (0x00007fff07550000)
    libFileIO.so => ./libFileIO.so (0x00007f2ddf1ac000)
    libc.so.6 => /lib64/libc.so.6 (0x00000035f5000000)
    /lib64/ld-linux-x86-64.so.2 (0x00000035f4c00000)
%
```



## Make and Makefile

- Building an application from multiple source files requires multiple compiler calls
  - Typing compiler calls manually is tedious and error-prone
    - Automation can help: build scripts!
- Linux tool **make**
  - reads compilation rules from a **Makefile** (script)
  - automatically executes necessary build commands
- **Makefile** consists of a set of rules
- Rules consist of
  - *Target* (usually a file)
  - *Dependencies* (typically source files)
  - *Command(s)* to generate the target from the sources

EECS22: Advanced C Programming, Lecture 11

(c) 2017 R. Doemer

17

## Make and Makefile

- Example Rule:

```
HelloWorld: HelloWorld.c
    gcc -Wall -ansi -std=c99 HelloWorld.c -o HelloWorld
```

Target **HelloWorld**

- depends on source file **HelloWorld.c**
- can be built by executing the listed command
  - Note: Command line starts with a *horizontal tabulator* (TAB)!

- Compilation: Make!

```
% vi HelloWorld.c
% vi Makefile
% make
gcc -Wall -ansi -std=c99 HelloWorld.c -o HelloWorld
% ./HelloWorld
Hello World!
%
```

EECS22: Advanced C Programming, Lecture 11

(c) 2017 R. Doemer

18

## Make and Makefile

- **Makefile** with Multiple Rules
  - **make**
    - Builds the first target (executes first rule)
  - **make target**
    - Builds the specified target (executes specified rule)
- **Rules and Dependencies**
  - Rules are applied *if and only if* the target file...
    - ... does not exist, or
    - ... is out of date
      - Target file is older than any of its dependencies
      - Every file has a time stamp of its last modification time, so **make** can determine what needs to be re-built
  - Rules are applied *recursively*
    - If a dependency is missing or is not up-to-date, **make** builds it first!

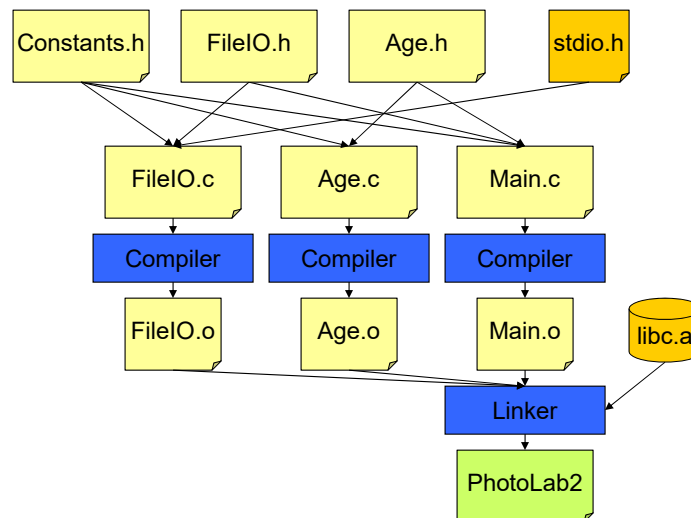
EECS22: Advanced C Programming, Lecture 11

(c) 2017 R. Doemer

19

## Make and Makefile

- **Making PhotoLab2**



EECS22: Advanced C Programming, Lecture 11

(c) 2017 R. Doemer

20

## Make and Makefile

- Example: Basic Makefile for PhotoLab2

```
# Makefile: PhotoLab2

PhotoLab2: FileIO.o Age.o Main.o
    gcc -Wall -ansi -std=c99 FileIO.o Age.o Main.o -o PhotoLab2

FileIO.o: FileIO.c FileIO.h Constants.h
    gcc -Wall -ansi -std=c99 -c FileIO.c -o FileIO.o

Age.o: Age.c Age.h Constants.h
    gcc -Wall -ansi -std=c99 -c Age.c -o Age.o

Main.o: Main.c Constants.h FileIO.h Age.h
    gcc -Wall -ansi -std=c99 -c Main.c -o Main.o
```

## Make and Makefile

- Example session: `make PhotoLab2`

```
% vi Constants.h
% vi FileIO.h
% vi FileIO.c
% vi Age.h
% vi Age.c
% vi Main.c
% vi Makefile
% make
gcc -Wall -ansi -std=c99 -c FileIO.c -o FileIO.o
gcc -Wall -ansi -std=c99 -c Age.c -o Age.o
gcc -Wall -ansi -std=c99 -c Main.c -o Main.o
gcc -Wall -ansi -std=c99 FileIO.o Age.o Main.o -o PhotoLab2
% ./PhotoLab2
% vi Age.c
% make
gcc -Wall -ansi -std=c99 -c Age.c -o Age.o
gcc -Wall -ansi -std=c99 FileIO.o Age.o Main.o -o PhotoLab2
% ./PhotoLab2
```

## Advanced Makefile

- Commands issued by **make**
  - Command line must start with a (horizontal) TAB character
    - Spaces are not recognized!
  - Multiple commands are executed in order
    - Long command lines can be wrapped to the next line by a backslash (\) immediately followed by a newline
  - Commands are echoed to the standard output
    - Echo can be suppressed with a @ prefix before the command
  - Commands are executed as shell commands
    - The **sh** shell is used (similar to **bash** in Linux)
  - Return value of command determines success
    - Return value 0 indicates success (no errors)
    - Return value not equal to 0 indicates error
  - Execution of commands stops with the first error
    - Errors can be ignored with a - prefix before the command

EECS22: Advanced C Programming, Lecture 11

(c) 2017 R. Doemer

23

## Advanced Makefile

- Advanced Features
  - Makefile variables
 

```
DEBUG = -g -DDEBUG
CFLAGS = -Wall -ansi -std=c99 $(DEBUG)
...
Program: Program.c Program.h
gcc $(CFLAGS) Program.c -o Program
```
  - Dummy targets (aka. *pseudo* or *phony* targets)
 

```
# default target
all: PhotoLab2

clean:
    rm -f *.o
    rm -f PhotoLab2
```
  - Many more features are available...
    - **man make**

EECS22: Advanced C Programming, Lecture 11

(c) 2017 R. Doemer

24

## Advanced Makefile

- Advanced Makefile for PhotoLab2 (part 1/2)

```
# Makefile: PhotoLab2
# 10/19/17 RD

# variable definitions
CC      = gcc
DEBUG   = -g -DDEBUG
#DEBUG  = -O2 -DNDEBUG
CFLAGS  = -Wall -ansi -std=c99 $(DEBUG) -c
LFLAGS  = -Wall $(DEBUG)

# convenience targets
all: PhotoLab2

clean:
    rm -f *.o
    rm -f PhotoLab2

test: PhotoLab2
    ./PhotoLab2

...
```

EECS22: Advanced C Programming, Lecture 11

(c) 2017 R. Doemer

25

## Advanced Makefile

- Advanced Makefile for PhotoLab2 (part 2/2)

```
...

# compilation rules
FileIO.o: FileIO.c FileIO.h Constants.h
    $(CC) $(CFLAGS) FileIO.c -o FileIO.o

Age.o: Age.c Age.h Constants.h
    $(CC) $(CFLAGS) Age.c -o Age.o

Main.o: Main.c Constants.h FileIO.h Age.h
    $(CC) $(CFLAGS) Main.c -o Main.o

PhotoLab2: FileIO.o Age.o Main.o
    $(CC) $(LFLAGS) FileIO.o Age.o Main.o -o PhotoLab2
```

EECS22: Advanced C Programming, Lecture 11

(c) 2017 R. Doemer

26

## Advanced Makefile

- Example session: `make PhotoLab2`

```
...
% vi Makefile
% make clean
rm -f *.o
rm -f PhotoLab2
% make test
gcc -Wall -ansi -std=c99 -c FileIO.c -o FileIO.o
gcc -Wall -ansi -std=c99 -c Age.c -o Age.o
gcc -Wall -ansi -std=c99 -c Main.c -o Main.o
gcc -g FileIO.o Age.o Main.o -o PhotoLab2
./PhotoLab2
% vi Age.c
% make test
gcc -Wall -ansi -std=c99 -c Age.c -o Age.o
gcc -g FileIO.o Age.o Main.o -o PhotoLab2
./PhotoLab2
%
```

## Part 2: Overview

- Debugging
  - Source-level debugger `gdb`
  - Running a program under debugger control
  - Navigating and inspecting the stack
  - Inspecting and modifying variable values
  - Advanced commands for using break points
  - Data display debugger `ddd`

## Debugging

- Source-level Debugger `gdb`
  - Debugging features
    - run the program under debugger control
    - follow the control flow of the program during execution
    - set breakpoints to stop execution at specific points
    - inspect (and adjust) the values of variables
    - find the point in the program where the “crash” happens
  - Preparation:
    - compile your program with debugging support on
    - Option `-g` tells compiler to add debugging information (symbol tables) to the generated executable file
    - `gcc -g Prog.c -o Prog -Wall -ansi -std=c99`
    - `gdb Prog`

EECS22: Advanced C Programming, Lecture 12

(c) 2017 R. Doemer

29

## Debugging

- Source-level Debugger `gdb`
  - Running the program under debugger control
    - `run`
      - starts the execution of the program in the debugger
    - `break function_name (or file:line_number)`
      - inserts a breakpoint; program execution will stop at the breakpoint
    - `cont`
      - continues the execution of the program in the debugger
    - `list from_line_number, to_line_number`
      - lists the current or specified range of line\_numbers
    - `print variable_name`
      - prints the current value of the variable `variable_name`
    - `next`
      - executes the next statement (one statement at a time)
    - `quit`
      - exits the debugger (and terminates the program)
    - `help`
      - provides helpful details on debugger commands

EECS22: Advanced C Programming, Lecture 12

(c) 2017 R. Doemer

30

## Debugging

- Example session: `Cylinder.c` (part 1/2)

```
% vi Cylinder.c
% gcc Cylinder.c -Wall -ansi -std-c99 -o Cylinder -g
% gdb Cylinder
GNU gdb (GDB) Red Hat Enterprise Linux (7.0.1-37.el5_7.1)
Copyright (C) 2009 Free Software Foundation, Inc.
...
Reading symbols from
/users/faculty/doemer/eecs22/lecture10/Cylinder...done.
(gdb) break main
Breakpoint 1 at 0x400654: file Cylinder.c, line 48.
(gdb) run
Starting program: /users/faculty/doemer/eecs22/lecture10/Cylinder
Breakpoint 1, main () at Cylinder.c:48
48     printf("Please enter the radius!\n");
(gdb) next
Please enter the radius!
49     scanf("%lf", &r);
...
```

EECS22: Advanced C Programming, Lecture 12

(c) 2017 R. Doemer

31

## Debugging

- Example session: `Cylinder.c` (part 2/2)

```
...
(gdb) next
5
50     printf("Please enter the height!\n");
(gdb) print r
$1 = 5
(gdb) cont
Continuing.
Please enter the height!
10
The surface area is 471.238905.
The volume is 785.398175.
Program exited normally.
(gdb) quit
%
```

EECS22: Advanced C Programming, Lecture 12

(c) 2017 R. Doemer

32



## Debugging

- Source-level Debugger `gdb` (continued)
  - Navigating the stack
    - `step`
      - steps into a function call
    - `finish`
      - continues execution until the current function has returned
    - `where`
      - shows where in the function call hierarchy you are
      - prints a *back trace* of current *stack frames*
    - `up`
      - steps up one stack frame (up into the caller)
    - `down`
      - steps down one stack frame (down into the callee)

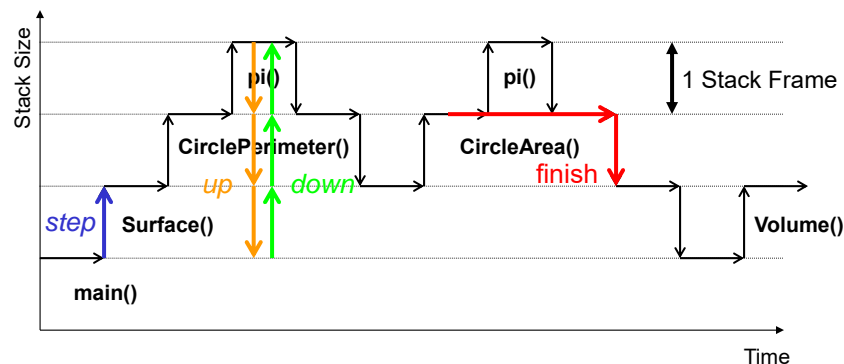
EECS22: Advanced C Programming, Lecture 12

(c) 2017 R. Doemer

33

## Debugging

- Navigating Stack Frames in the Debugger
  - `step`: execute and step into a function call
  - `up`, `down`: navigate stack frames
  - `finish`: resume execution until the end of the current function



EECS22: Advanced C Programming, Lecture 12

(c) 2017 R. Doemer

34

## Debugging

- Example session: `Cylinder.c` (part 1/4)

```
% vi Cylinder.c
% gcc Cylinder.c -o Cylinder -Wall -ansi -g
% gdb Cylinder
GNU gdb 6.3
(gdb) break 55
Breakpoint 1 at 0x108d0: file Cylinder.c, line 55.
(gdb) run
Starting program: /users/faculty/doemer/eecs10/Cylinder/Cylinder
Please enter the radius: 10
Please enter the height: 10
Breakpoint 1, main () at Cylinder.c:56
56      s = Surface(r, h);
(gdb) step
Surface (r=10, h=10) at Cylinder.c:31
31      side = CirclePerimeter(r) * h;
(gdb) step
CirclePerimeter (r=10) at Cylinder.c:24
24      return(2 * pi() * r);
...
EE
```

## Debugging

- Example session: `Cylinder.c` (part 2/4)

```
(gdb) step
pi () at Cylinder.c:14
14      return(3.1415927);
(gdb) where
#0  pi () at Cylinder.c:14
#1  0x000107bc in CirclePerimeter (r=10) at Cylinder.c:24
#2  0x000107f8 in Surface (r=10, h=10) at Cylinder.c:31
#3  0x000108e0 in main () at Cylinder.c:56
(gdb) up
#1  0x000107bc in CirclePerimeter (r=10) at Cylinder.c:24
24      return(2 * pi() * r);
(gdb) up
#2  0x000107f8 in Surface (r=10, h=10) at Cylinder.c:31
31      side = CirclePerimeter(r) * h;
(gdb) up
#3  0x000108e0 in main () at Cylinder.c:56
56      s = Surface(r, h);
...
EE
```

## Debugging

- Example session: `Cylinder.c` (part 3/4)

```
(gdb) down
#2 0x000107f8 in Surface (r=10, h=10) at Cylinder.c:31
31      side = CirclePerimeter(r) * h;
(gdb) down
#1 0x000107bc in CirclePerimeter (r=10) at Cylinder.c:24
24      return(2 * pi() * r);
(gdb) down
#0 pi () at Cylinder.c:14
14      return(3.1415927);
(gdb) finish
Run till exit from #0 pi () at Cylinder.c:14
0x000107bc in CirclePerimeter (r=10) at Cylinder.c:24
24      return(2 * pi() * r);
Value returned is $1 = 3.1415926999999999
(gdb) finish
Run till exit from #0 CirclePerimeter (r=10) at Cylinder.c:24
0x000107f8 in Surface (r=10, h=10) at Cylinder.c:31
31      side = CirclePerimeter(r) * h;
...
EE
```

## Debugging

- Example session: `Cylinder.c` (part 4/4)

```
Value returned is $2 = 62.831854
(gdb) next
32      lid = CircleArea(r);
(gdb) step
CircleArea (r=10) at Cylinder.c:19
19      return(pi() * r * r);
(gdb) finish
Run till exit from #0 CircleArea (r=10) at Cylinder.c:19
0x00010818 in Surface (r=10, h=10) at Cylinder.c:32
32      lid = CircleArea(r);
Value returned is $3 = 314.15926999999999
(gdb) cont
Continuing.
The surface area is 1256.637080.
The volume is 3141.592700.
Program exited normally.
(gdb) quit
%
```

## Debugging

- Source-level Debugger `gdb` (continued)
  - Inspecting the stack
    - `info frame`
      - displays information about the current stack frame
    - `info locals`
      - lists the local variables in the current function (current stack frame)
    - `info scope function`
      - lists the variables in the scope of the specified function
  - Calling functions (outside of the regular control flow)
    - `call function(arguments)`
      - calls the specified function with the specified arguments
  - Assembly level inspection
    - `info registers`
      - lists the CPU registers and their contents
    - `disassemble function`
      - disassembles the function and lists its assembly code

## Debugging

- Source-level Debugger `gdb` (continued)
  - Inspecting and modifying variable values
    - `print variable_name`
      - prints the current value of the variable `variable_name`
    - `set variable = value`
      - sets the specified variable to the specified value
    - `display variable`
      - prints the value of a variable each time before the next command
    - `info display`
      - lists information on the displayed variables
    - `undisplay variable`
      - turns off the display of the specified variable

## Debugging

- Source-level Debugger `gdb` (continued)
  - Advanced commands for using break points
    - `info breakpoints`
      - displays information about break points
    - `tbreak function_name (Or file:line_number)`
      - inserts a temporary breakpoint (valid only once)
    - `watch variable`
      - sets a watch point on the specified variable for write access
    - `rwatch variable`
      - sets a watch point on the specified variable for read access
    - `ignore breakpoint n`
      - skips the specified break point  $n$  times
    - `enable (Or disable) breakpoint (Or watchpoint)`
      - Enables (or disables) a break point (or watch point)
    - `condition breakpoint condition`
      - Specifies a condition for the given break point

EECS22: Advanced C Programming, Lecture 12

(c) 2017 R. Doemer

41

## Debugging

- Data Display Debugger `ddd`
  - Graphical frontend for `gdb`
    - Requires *X forwarding* and corresponding client (e.g. *Xming* in addition to *Putty*)
  - Provides menu bar and command buttons
  - Displays separate work windows
    - Graphical display area for data structures
    - Source code browser
    - Assembly code browser
    - Command line interface
  - Example: `Cylinder.c`

```

ddd: /users/faculty/doemer/eecs22/lectures/Cylinder.c
File Edit View Program Commands Status Source Data Help
0 Cylinder.c:55
Run| Interrupt| Step| Step| Next| Next| Until| Finish| Cont| Kill| Up| Down| Undo| Redo| Edit| Make
1: r 2: h 3: s
10: 20 1884.9582
48 printf("Please enter the radius\n");
49 scanf("%f", &r);
50 printf("Please enter the height\n");
51 scanf("%f", &h);
52
53 /* computation section */
54 s = Surface(r, h);
55 v = Volume(r, h);
56
57 /* output section */
58 printf("The surface area is %f.\n", s);

(gdb) graph display s
(gdb) break Cylinder.c:55
Breakpoint 2 at 0x4008ba: file Cylinder.c, line 55.
(gdb) break Cylinder.c:55
Breakpoint 3 at 0x4008ba: file Cylinder.c, line 55.
(gdb) break Cylinder.c:55
Breakpoint 4 at 0x4008ba: file Cylinder.c, line 55.
(gdb) clear Cylinder.c:55
Deleted breakpoints 2 3 4
(gdb) ]
Deleted breakpoints 2 3 4
  
```

EECS22: Advanced C Programming, Lecture 12

(c) 2017 R. Doemer

42