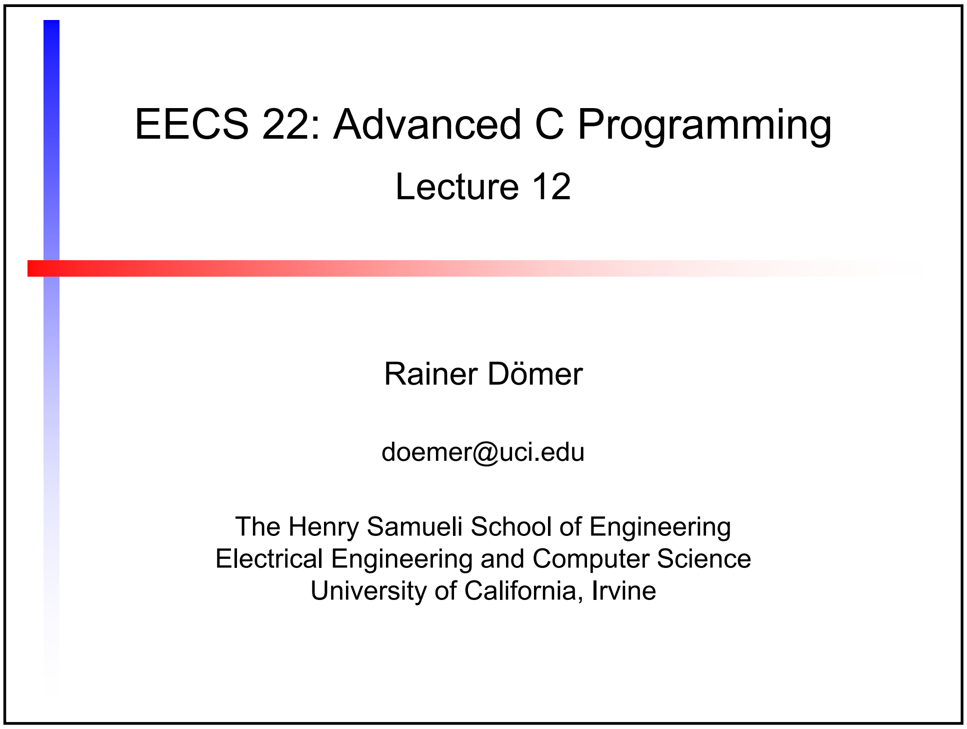


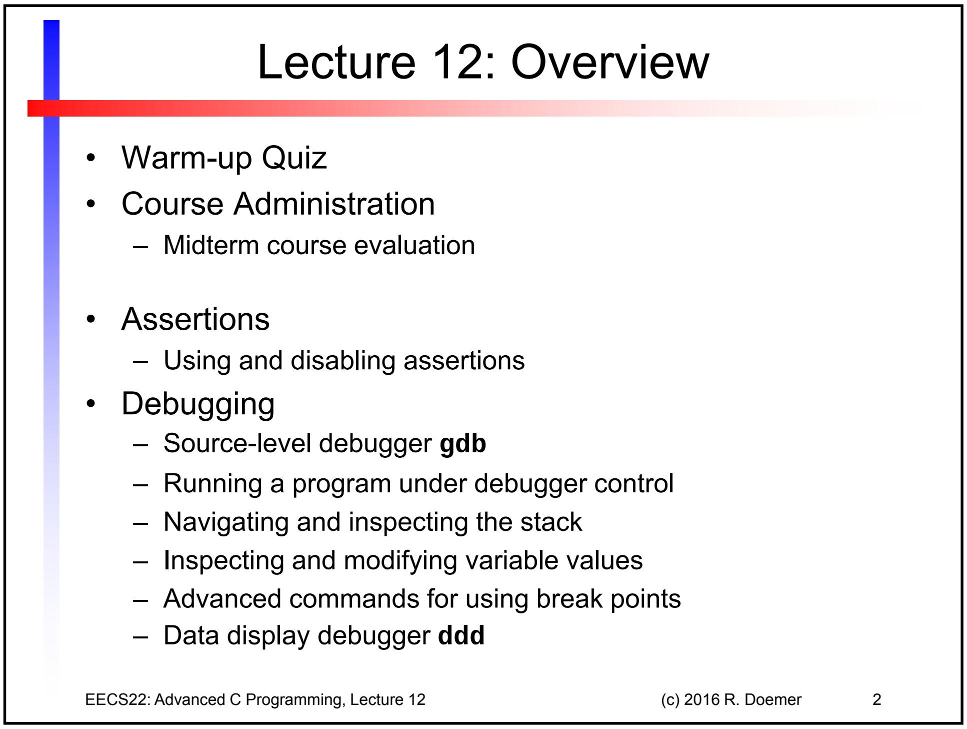
EECS 22: Advanced C Programming

Lecture 12

Rainer Dömer

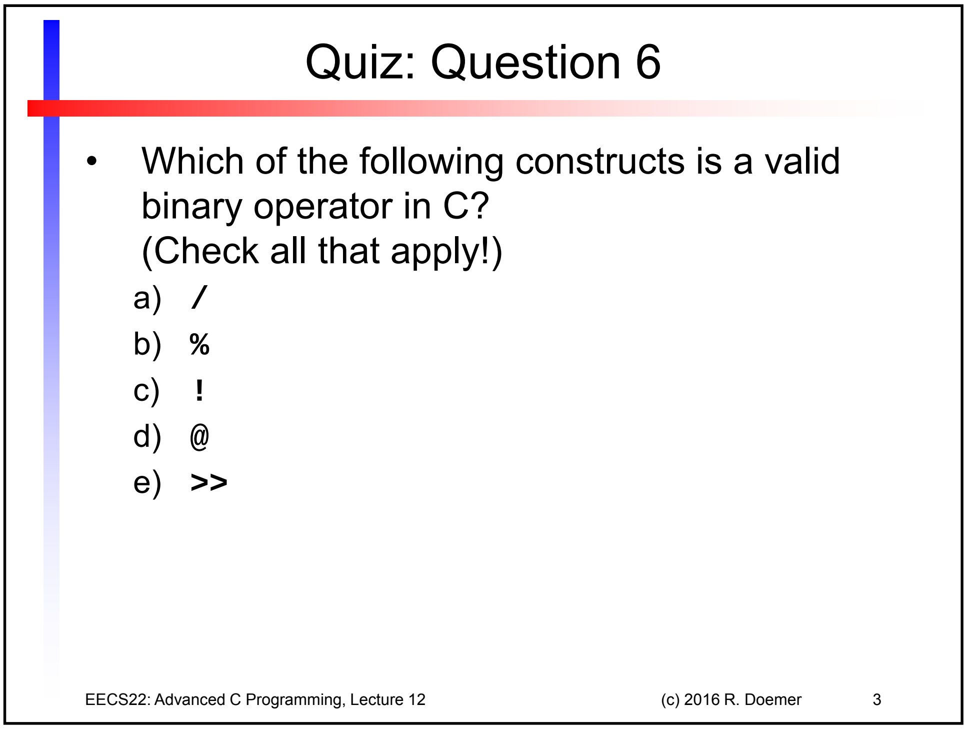
doemer@uci.edu

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



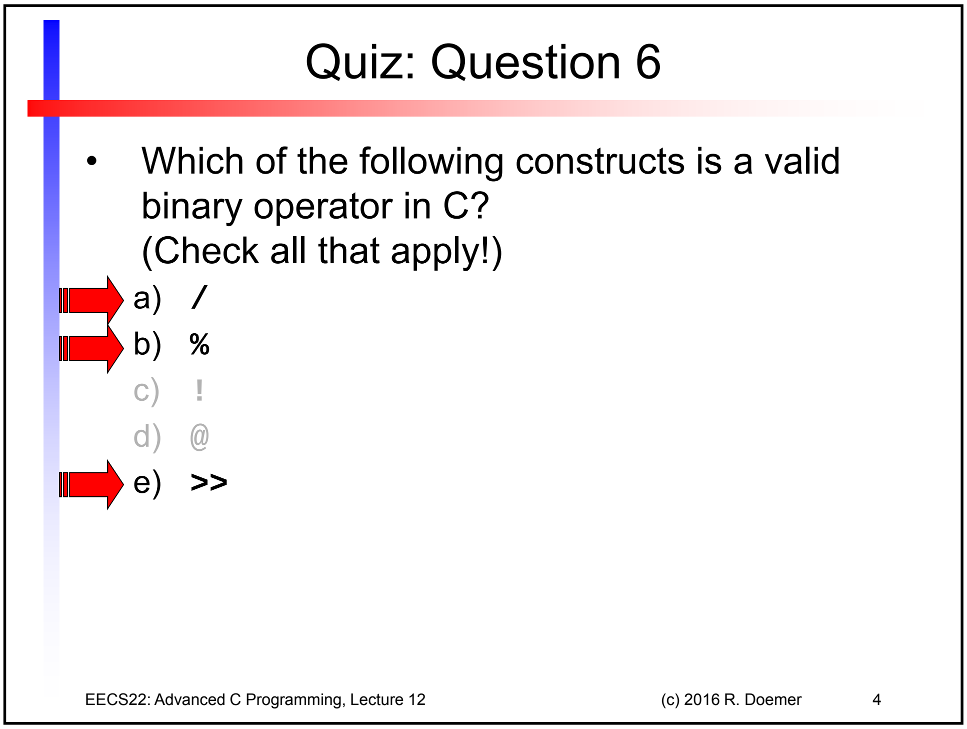
Lecture 12: Overview

- Warm-up Quiz
- Course Administration
  - Midterm course evaluation
- Assertions
  - Using and disabling assertions
- 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**



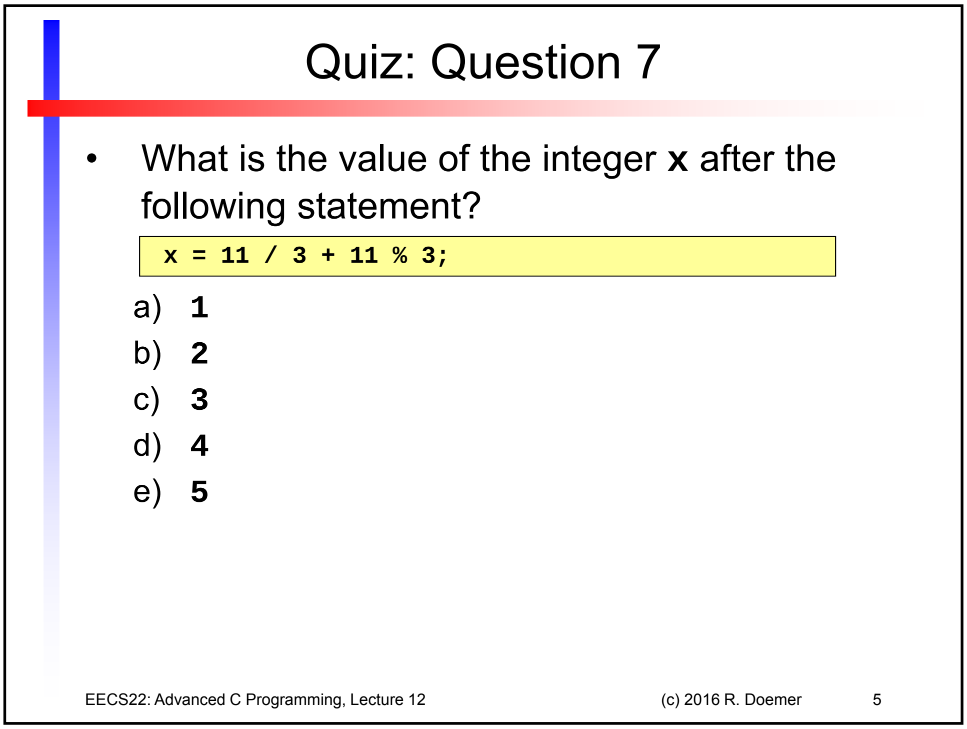
Quiz: Question 6

- Which of the following constructs is a valid binary operator in C?
  (Check all that apply!)
  a) `/`
  b) `%`
  c) `!`
  d) `@`
  e) `>>`



Quiz: Question 6

- Which of the following constructs is a valid binary operator in C?
  (Check all that apply!)
  [✓ a) `/`]
  [✓ b) `%`]
  c) `!`
  d) `@`
  [✓ e) `>>`]

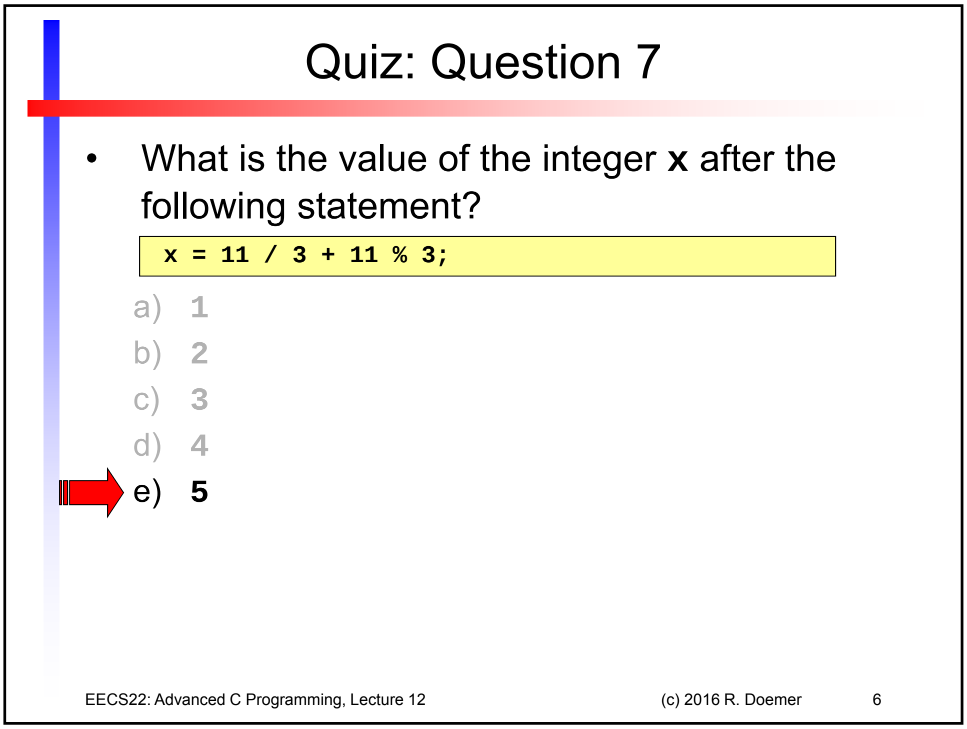


Quiz: Question 7

- What is the value of the integer **x** after the following statement?

```
x = 11 / 3 + 11 % 3;
```

a) **1**
b) **2**
c) **3**
d) **4**
e) **5**

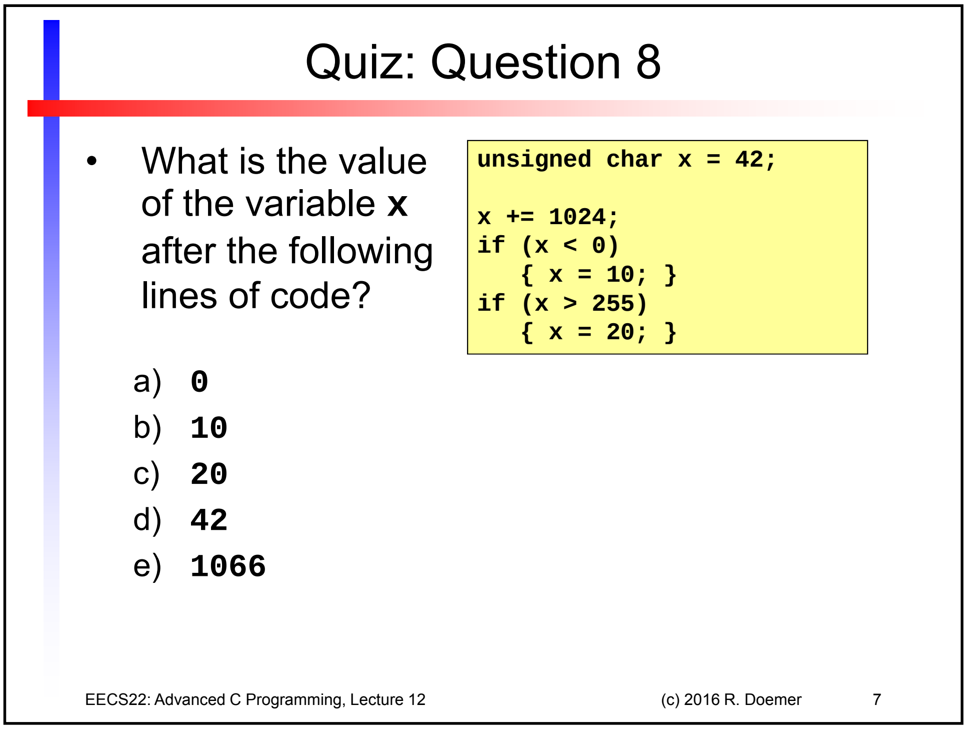


Quiz: Question 7

- What is the value of the integer **x** after the following statement?

```
x = 11 / 3 + 11 % 3;
```

a) **1**
b) **2**
c) **3**
d) **4**
➡ e) **5**



Quiz: Question 8

- What is the value of the variable **x** after the following lines of code?

```
unsigned char x = 42;

x += 1024;
if (x < 0)
   { x = 10; }
if (x > 255)
   { x = 20; }
```

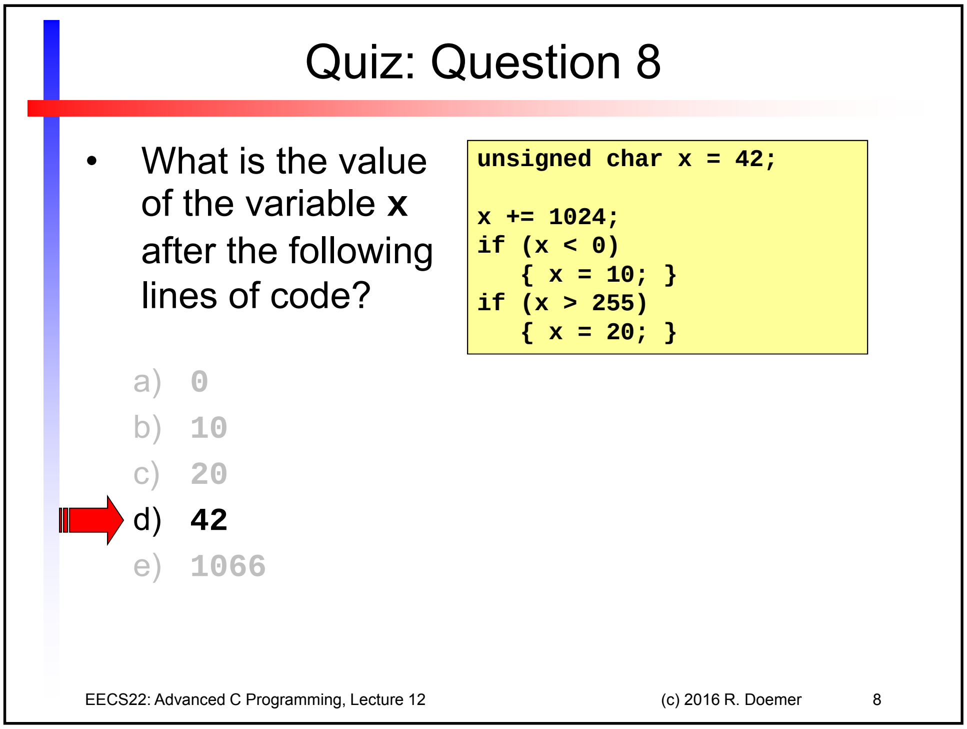
a) **0**

b) **10**

c) **20**

d) **42**

e) **1066**



Quiz: Question 8

- What is the value of the variable **x** after the following lines of code?

```
unsigned char x = 42;

x += 1024;
if (x < 0)
   { x = 10; }
if (x > 255)
   { x = 20; }
```

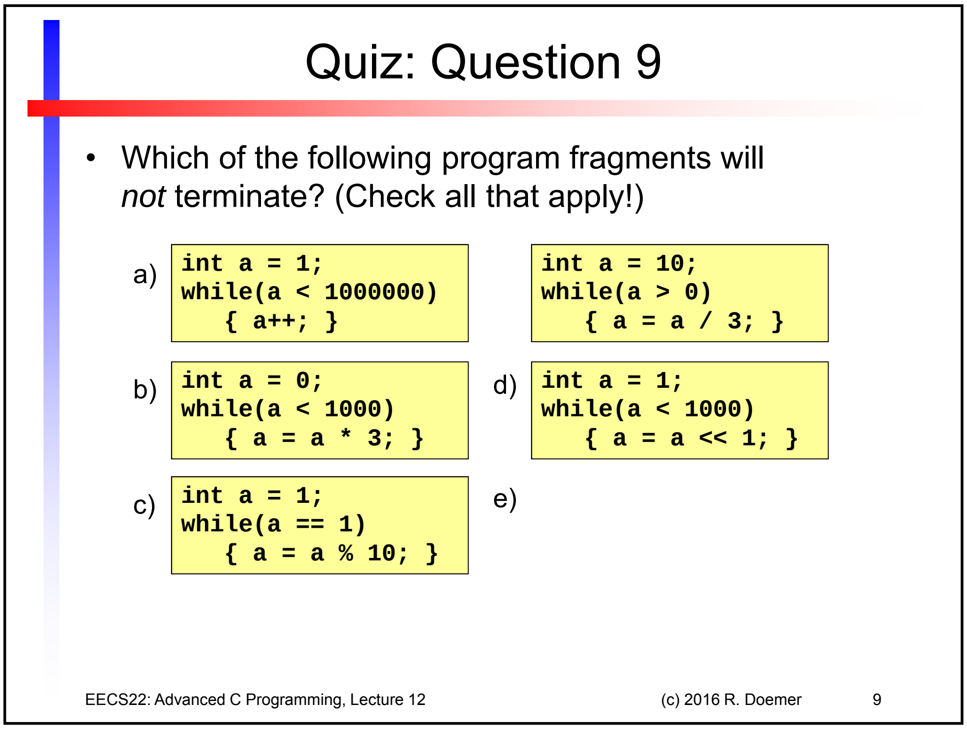
a) **0**

b) **10**

c) **20**

[✓ d) **42**]

e) **1066**



Quiz: Question 9

- Which of the following program fragments will *not* terminate? (Check all that apply!)

a)
```
int a = 1;
while(a < 1000000)
   { a++; }
```

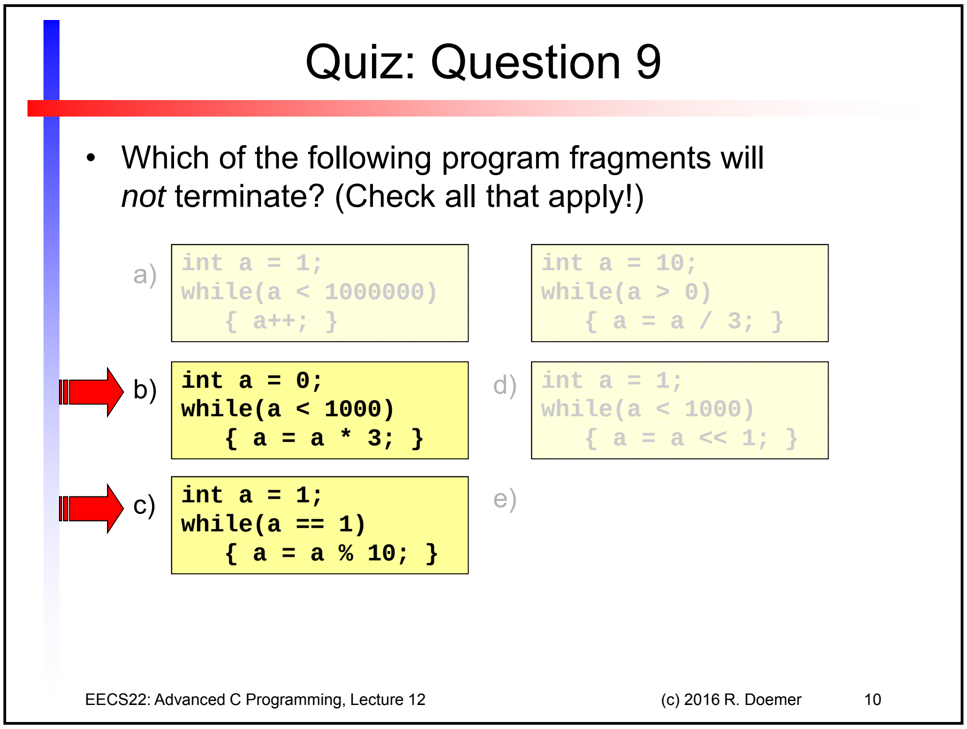
```
int a = 10;
while(a > 0)
   { a = a / 3; }
```

b)
```
int a = 0;
while(a < 1000)
   { a = a * 3; }
```

d)
```
int a = 1;
while(a < 1000)
   { a = a << 1; }
```

c)
```
int a = 1;
while(a == 1)
   { a = a % 10; }
```

e)



Quiz: Question 9

- Which of the following program fragments will *not* terminate? (Check all that apply!)

a)
```
int a = 1;
while(a < 1000000)
   { a++; }
```

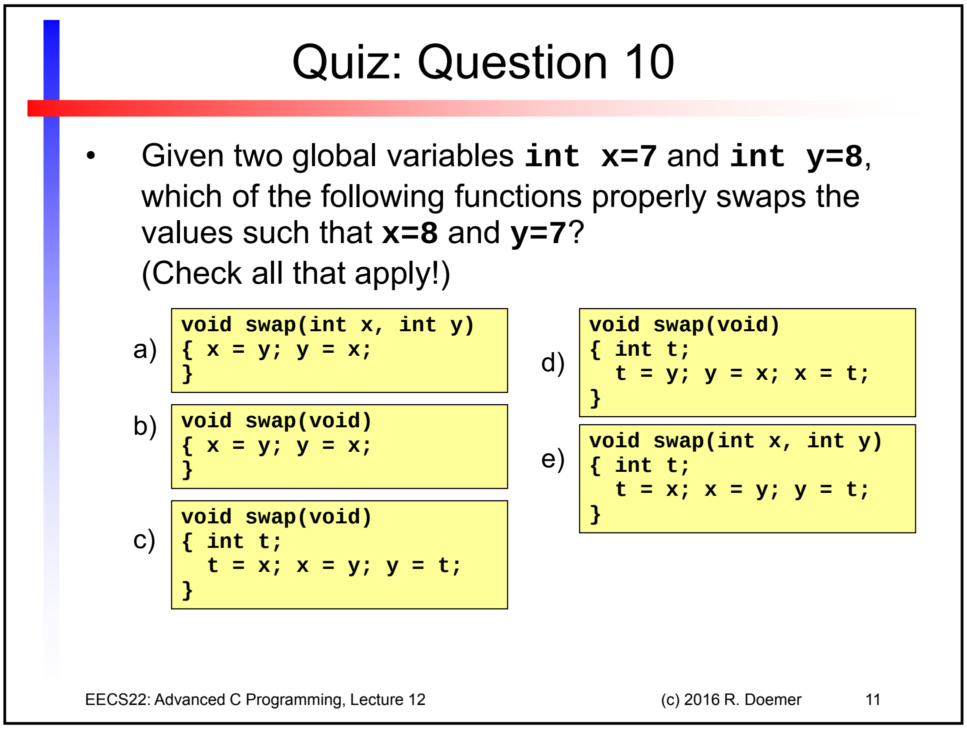
```
int a = 10;
while(a > 0)
   { a = a / 3; }
```

➡ b)
```
int a = 0;
while(a < 1000)
   { a = a * 3; }
```

d)
```
int a = 1;
while(a < 1000)
   { a = a << 1; }
```

➡ c)
```
int a = 1;
while(a == 1)
   { a = a % 10; }
```

e)



Quiz: Question 10

- Given two global variables **`int x=7`** and **`int y=8`**, which of the following functions properly swaps the values such that **`x=8`** and **`y=7`**? (Check all that apply!)

a)
```
void swap(int x, int y)
{ x = y; y = x;
}
```

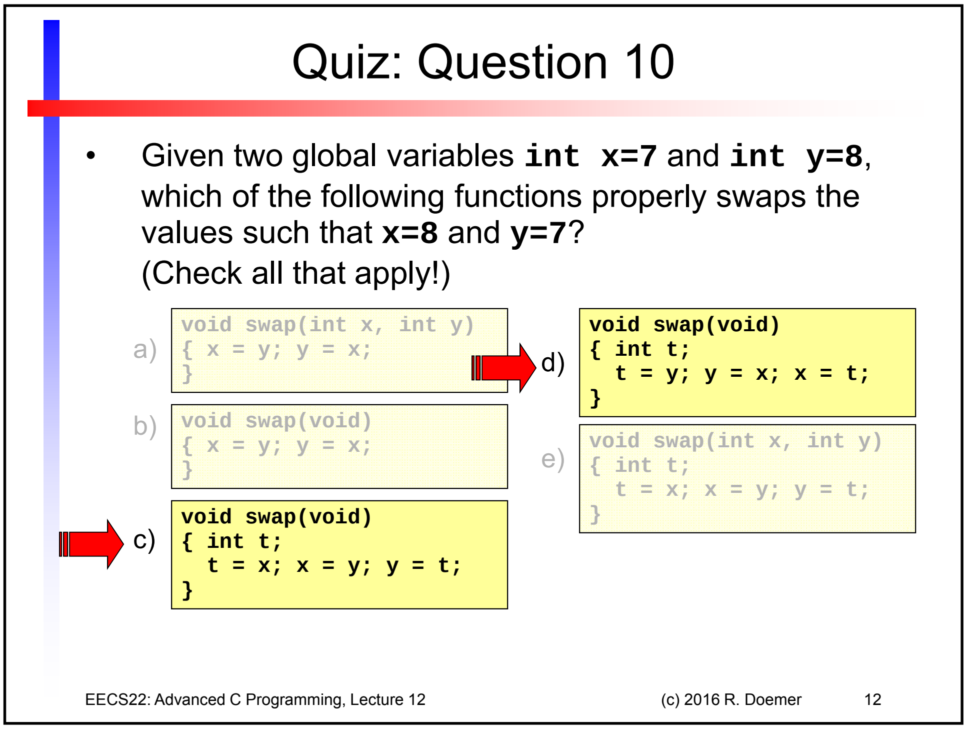
b)
```
void swap(void)
{ x = y; y = x;
}
```

c)
```
void swap(void)
{ int t;
  t = x; x = y; y = t;
}
```

d)
```
void swap(void)
{ int t;
  t = y; y = x; x = t;
}
```

e)
```
void swap(int x, int y)
{ int t;
  t = x; x = y; y = t;
}
```

EECS22: Advanced C Programming, Lecture 12 (c) 2016 R. Doemer 11



Quiz: Question 10

- Given two global variables **`int x=7`** and **`int y=8`**, which of the following functions properly swaps the values such that **`x=8`** and **`y=7`**? (Check all that apply!)

a)
```
void swap(int x, int y)
{ x = y; y = x;
}
```

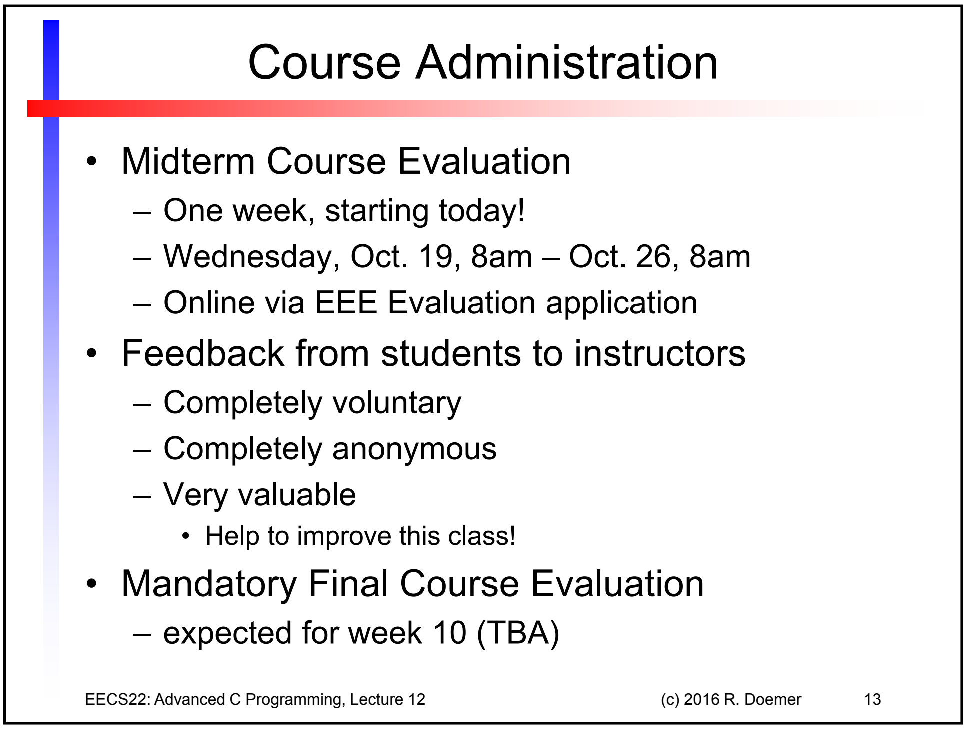
b)
```
void swap(void)
{ x = y; y = x;
}
```

c)
```
void swap(void)
{ int t;
  t = x; x = y; y = t;
}
```

d)
```
void swap(void)
{ int t;
  t = y; y = x; x = t;
}
```

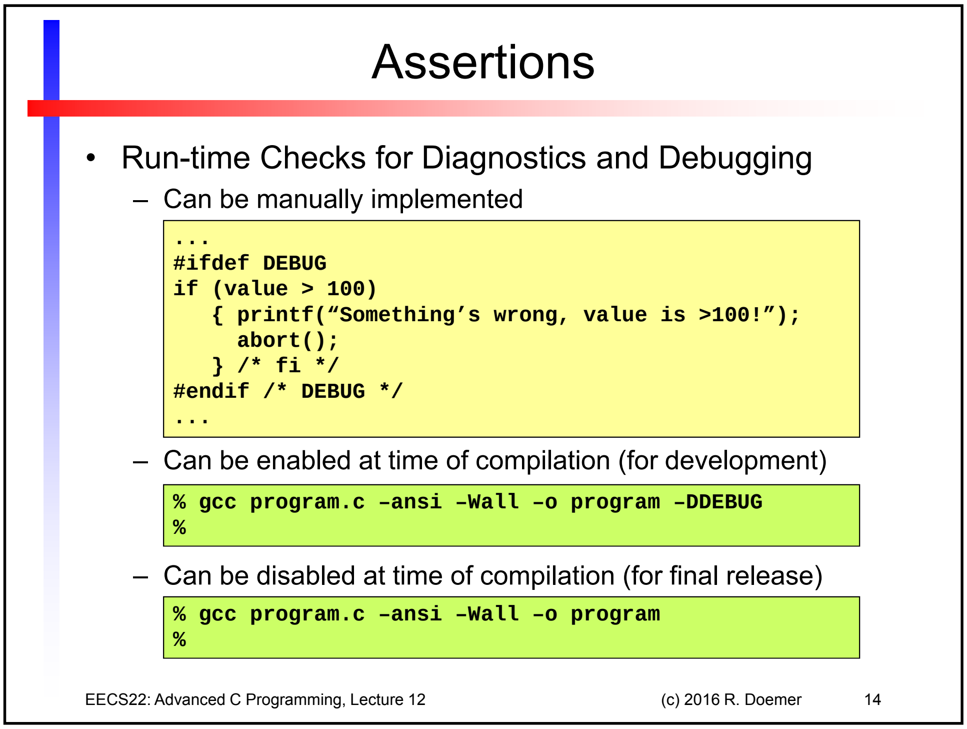
e)
```
void swap(int x, int y)
{ int t;
  t = x; x = y; y = t;
}
```

EECS22: Advanced C Programming, Lecture 12 (c) 2016 R. Doemer 12



Course Administration

- Midterm Course Evaluation
  - One week, starting today!
  - Wednesday, Oct. 19, 8am – Oct. 26, 8am
  - Online via EEE Evaluation application
- Feedback from students to instructors
  - Completely voluntary
  - Completely anonymous
  - Very valuable
    - Help to improve this class!
- Mandatory Final Course Evaluation
  - expected for week 10 (TBA)



Assertions

- Run-time Checks for Diagnostics and Debugging
  - Can be manually implemented

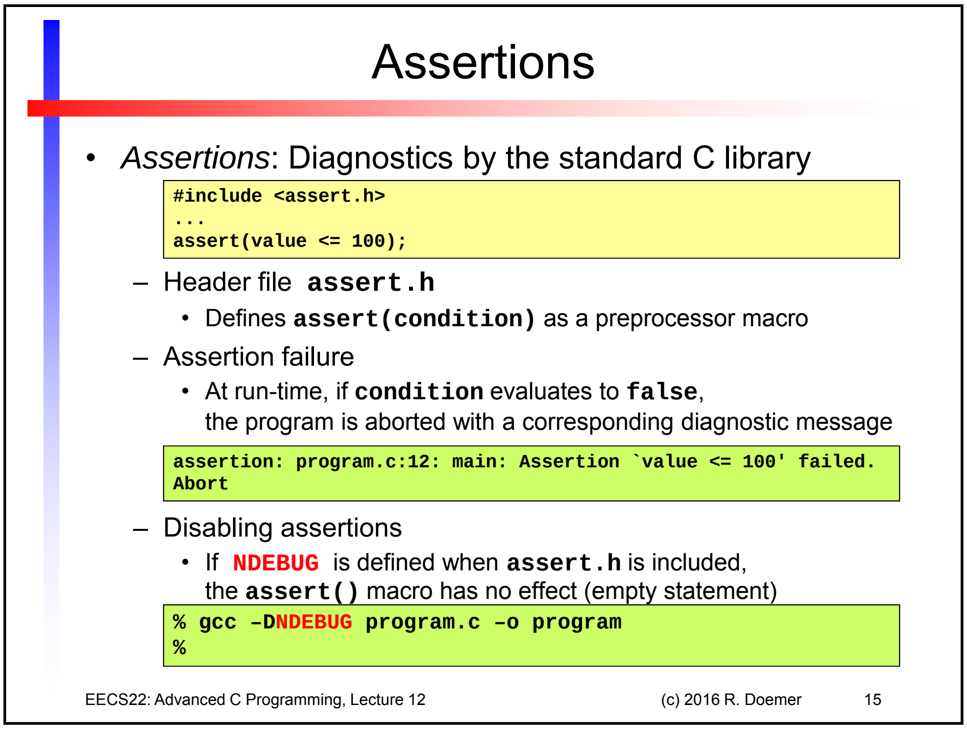
```
...
#ifdef DEBUG
if (value > 100)
   { printf(“Something’s wrong, value is >100!”);
     abort();
   } /* fi */
#endif /* DEBUG */
...
```

  - Can be enabled at time of compilation (for development)

```
% gcc program.c -ansi -Wall -o program -DDEBUG
%
```

  - Can be disabled at time of compilation (for final release)

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



Assertions

- *Assertions*: Diagnostics by the standard C library

```
#include <assert.h>
...
assert(value <= 100);
```

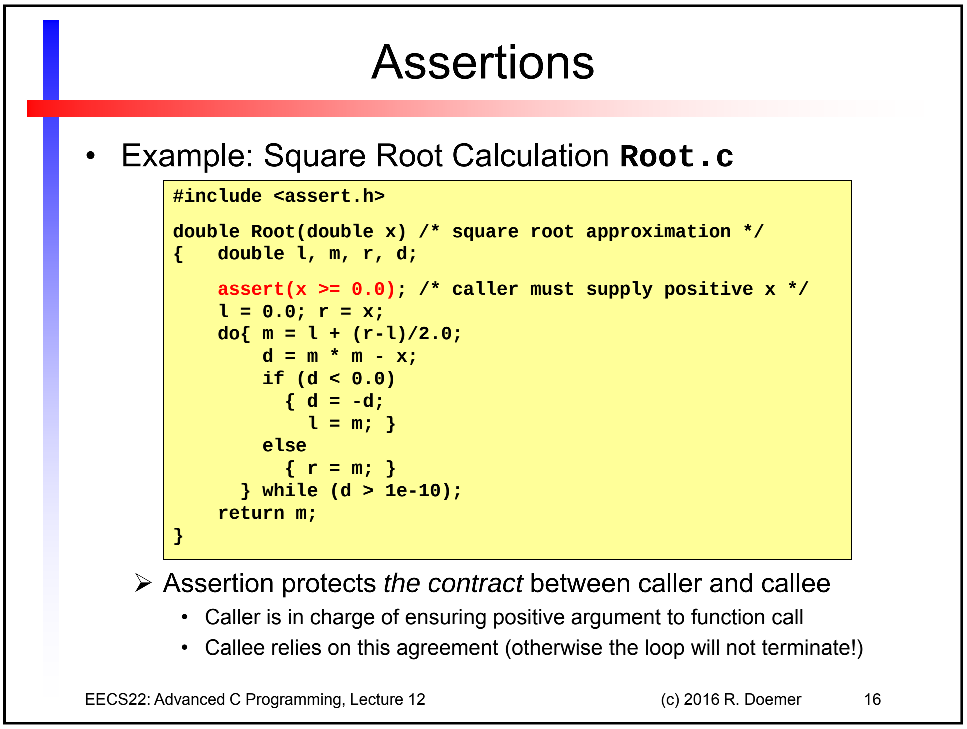
  - Header file **`assert.h`**
    - Defines **`assert(condition)`** as a preprocessor macro
  - Assertion failure
    - At run-time, if **`condition`** evaluates to **`false`**,
      the program is aborted with a corresponding diagnostic message

```
assertion: program.c:12: main: Assertion `value <= 100' failed.
Abort
```

  - Disabling assertions
    - If **`NDEBUG`** is defined when **`assert.h`** is included,
      the **`assert()`** macro has no effect (empty statement)

```
% gcc -DNDEBUG program.c -o program
%
```

EECS22: Advanced C Programming, Lecture 12 (c) 2016 R. Doemer 15



Assertions

- Example: Square Root Calculation **`Root.c`**

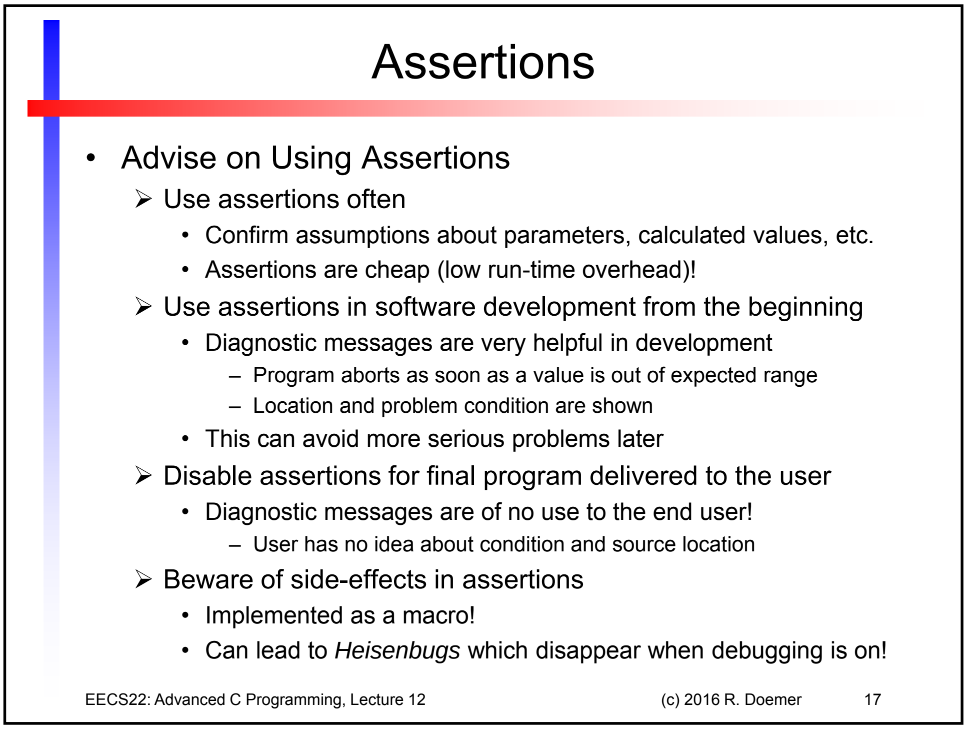
```
#include <assert.h>

double Root(double x) /* square root approximation */
{   double l, m, r, d;

    assert(x >= 0.0); /* caller must supply positive x */
    l = 0.0; r = x;
    do{ m = l + (r-l)/2.0;
        d = m * m - x;
        if (d < 0.0)
          { d = -d;
            l = m; }
        else
          { r = m; }
      } while (d > 1e-10);
    return m;
}
```

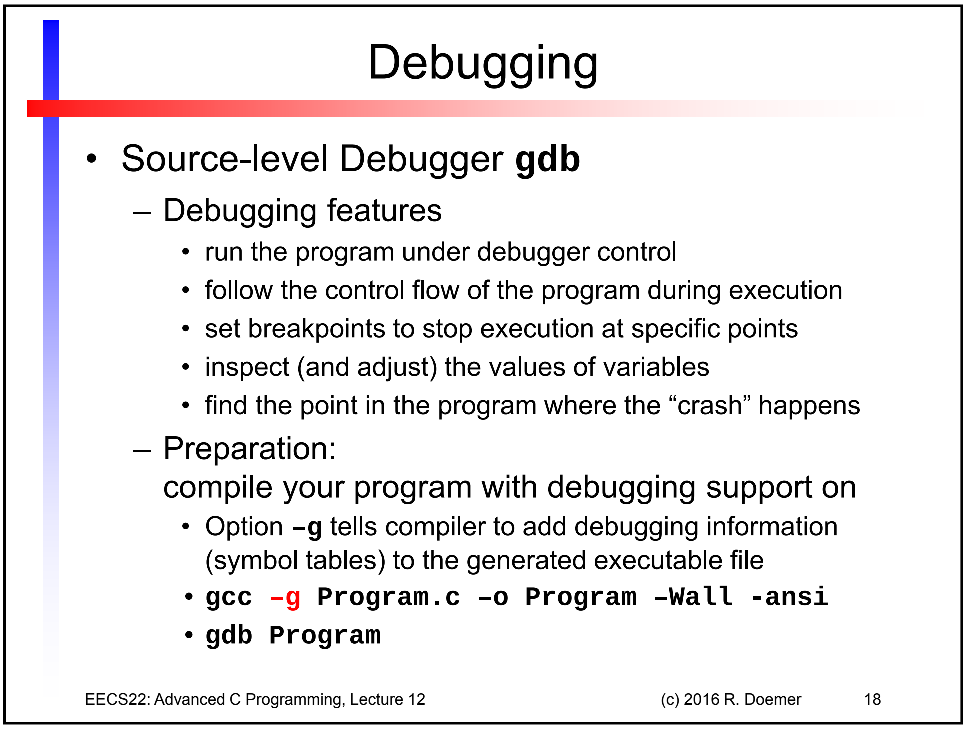
➢ Assertion protects *the contract* between caller and callee
  - Caller is in charge of ensuring positive argument to function call
  - Callee relies on this agreement (otherwise the loop will not terminate!)

EECS22: Advanced C Programming, Lecture 12 (c) 2016 R. Doemer 16



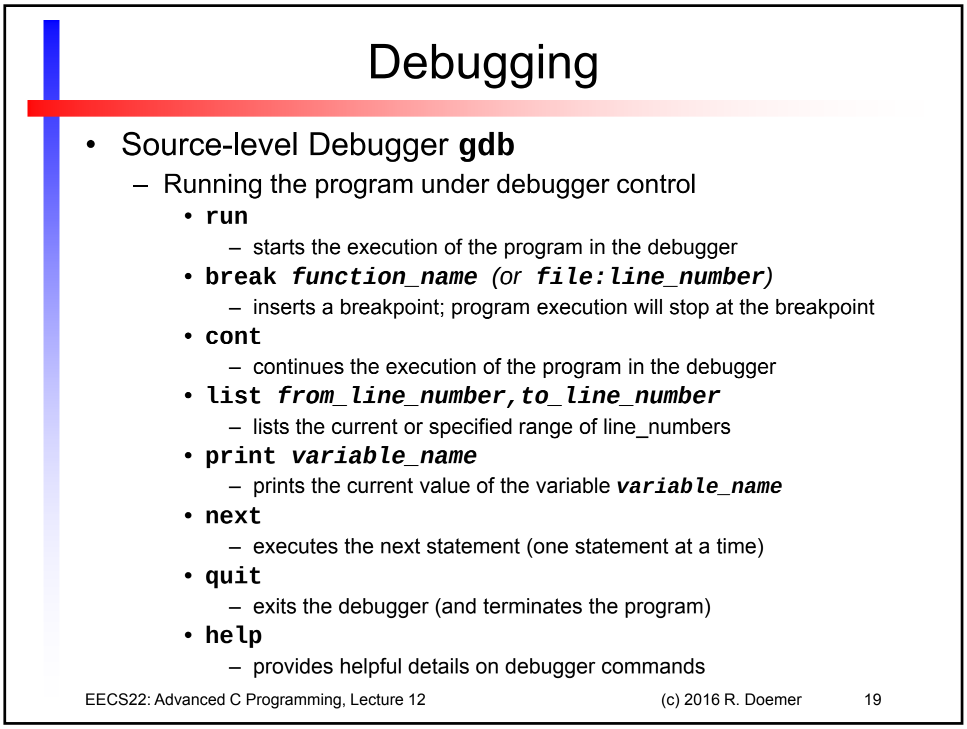
Assertions

- Advise on Using Assertions
  - Use assertions often
    - Confirm assumptions about parameters, calculated values, etc.
    - Assertions are cheap (low run-time overhead)!
  - Use assertions in software development from the beginning
    - Diagnostic messages are very helpful in development
      - Program aborts as soon as a value is out of expected range
      - Location and problem condition are shown
    - This can avoid more serious problems later
  - Disable assertions for final program delivered to the user
    - Diagnostic messages are of no use to the end user!
      - User has no idea about condition and source location
  - Beware of side-effects in assertions
    - Implemented as a macro!
    - Can lead to *Heisenbugs* which disappear when debugging is on!



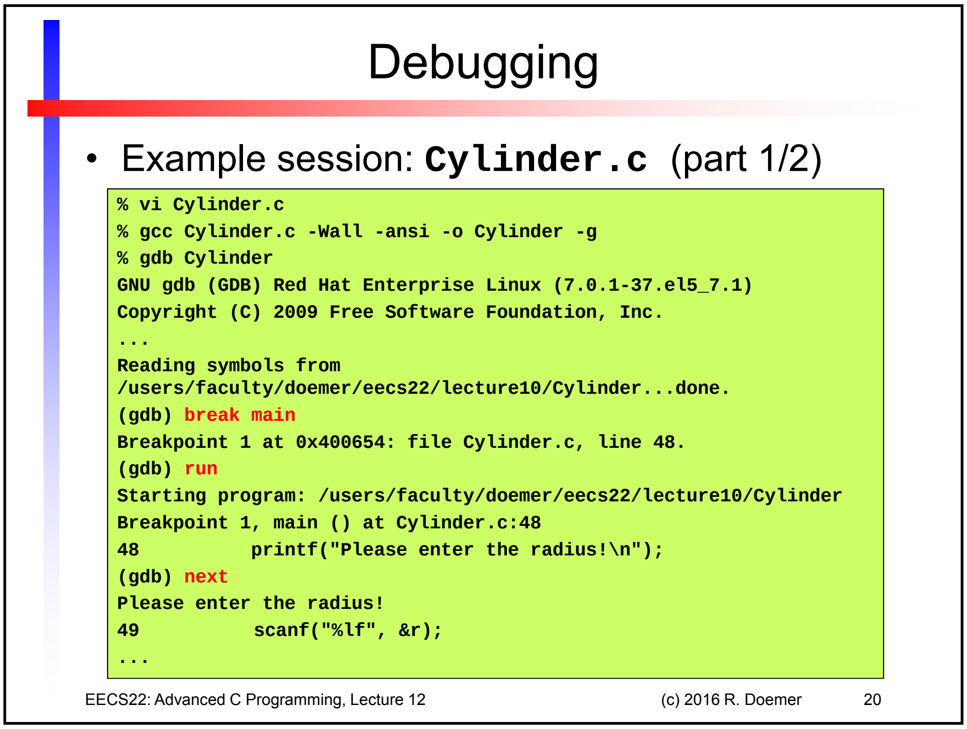
Debugging

- 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 Program.c -o Program -Wall -ansi`**
    - **`gdb Program`**



Debugging

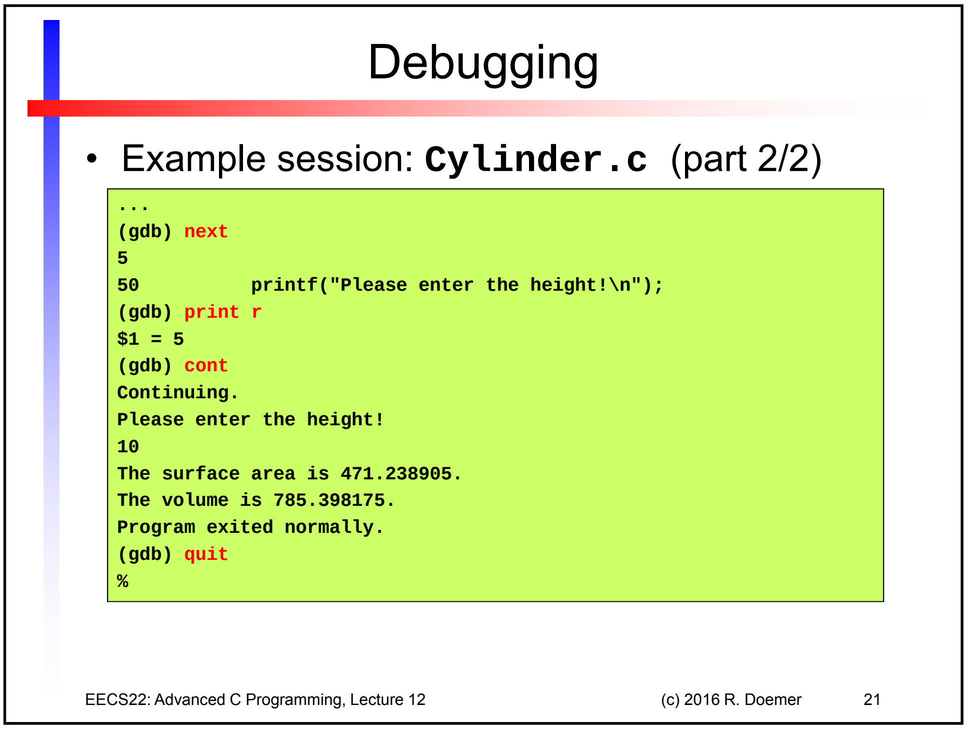
- Source-level Debugger **gdb**
  - Running the program under debugger control
    - **`run`**
      - starts the execution of the program in the debugger
    - **`break`** ***`function_name`*** *(or* ***`file:line_number`****)*
      - inserts a breakpoint; program execution will stop at the breakpoint
    - **`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



Debugging

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

```
% vi Cylinder.c
% gcc Cylinder.c -Wall -ansi -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);
...
```

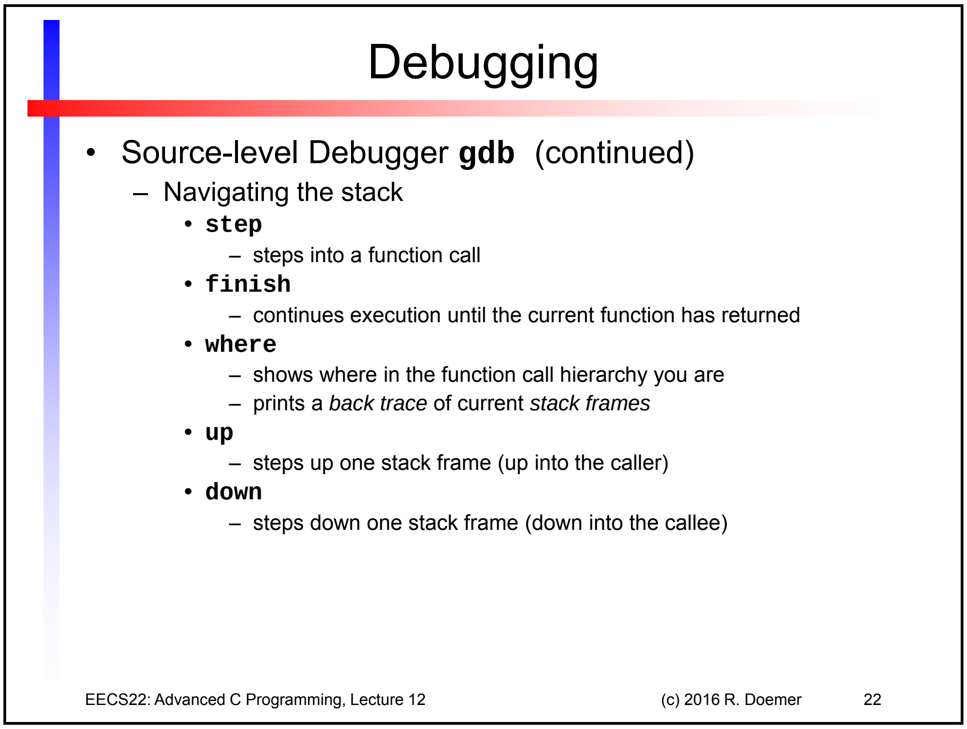


Debugging

- 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
%
```

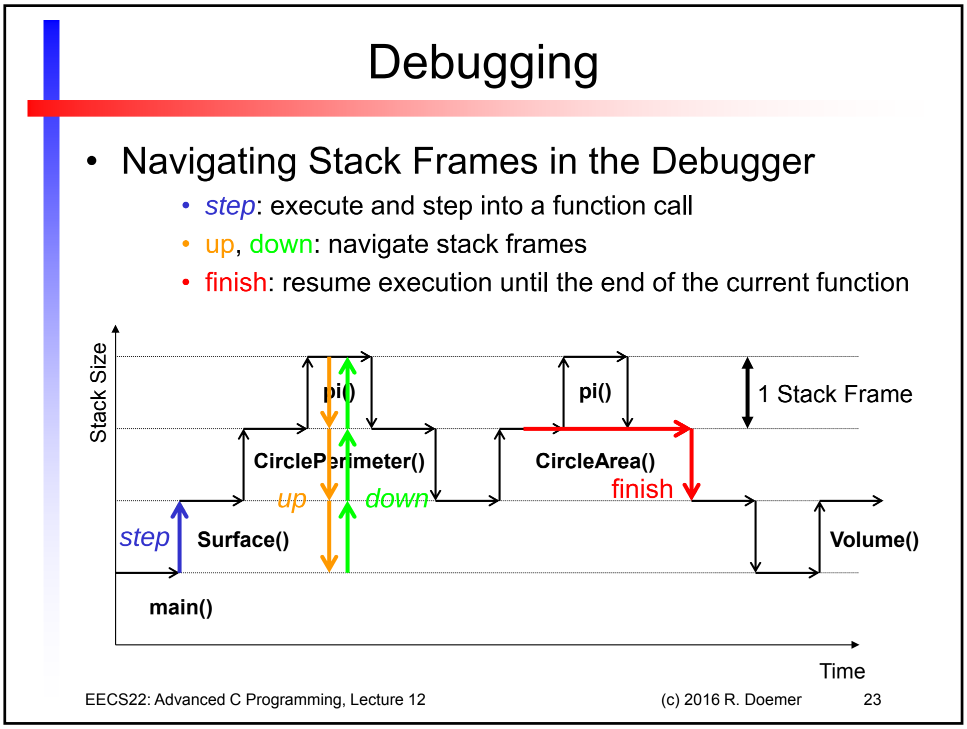
EECS22: Advanced C Programming, Lecture 12 (c) 2016 R. Doemer 21



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)

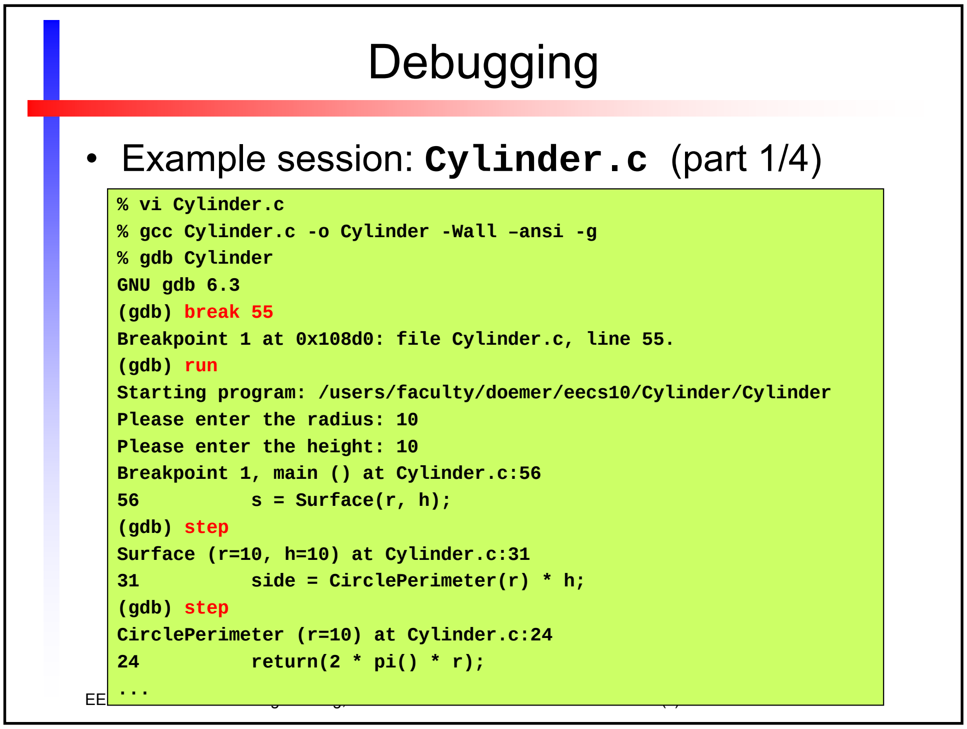
EECS22: Advanced C Programming, Lecture 12 (c) 2016 R. Doemer 22



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) 2016 R. Doemer 23

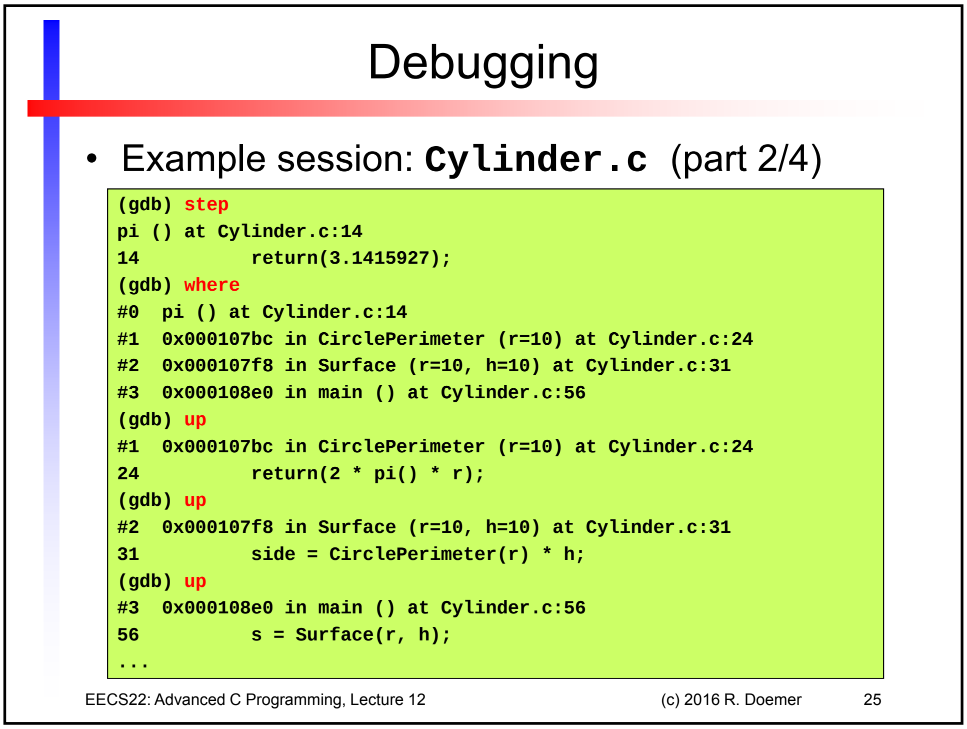


Debugging

- 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

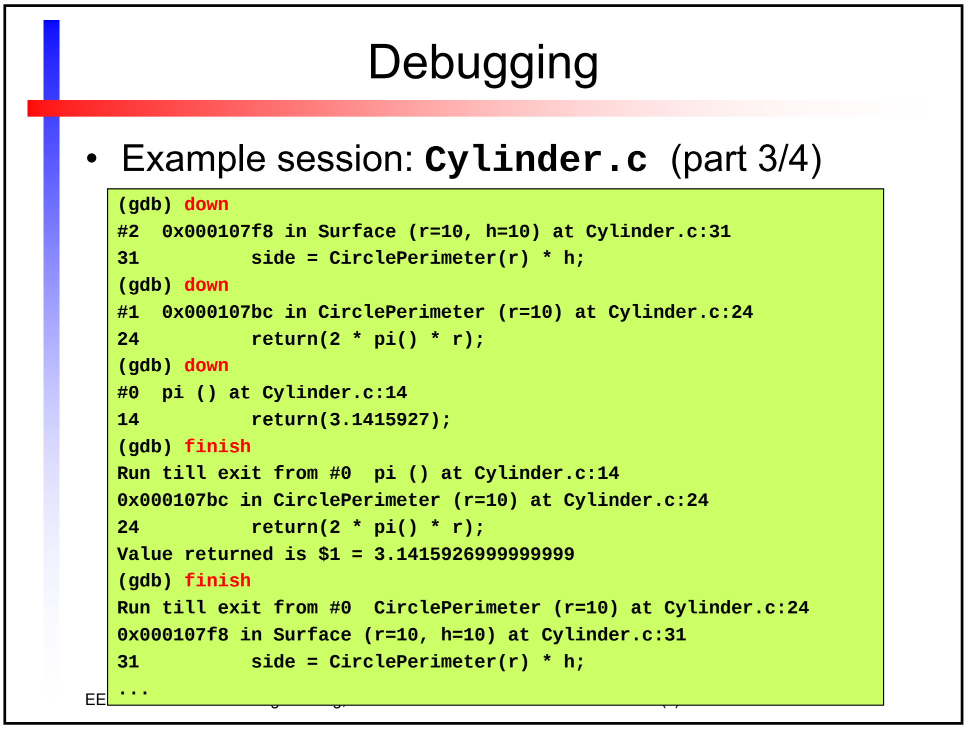


Debugging

- 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);
...
```

EECS22: Advanced C Programming, Lecture 12 (c) 2016 R. Doemer 25

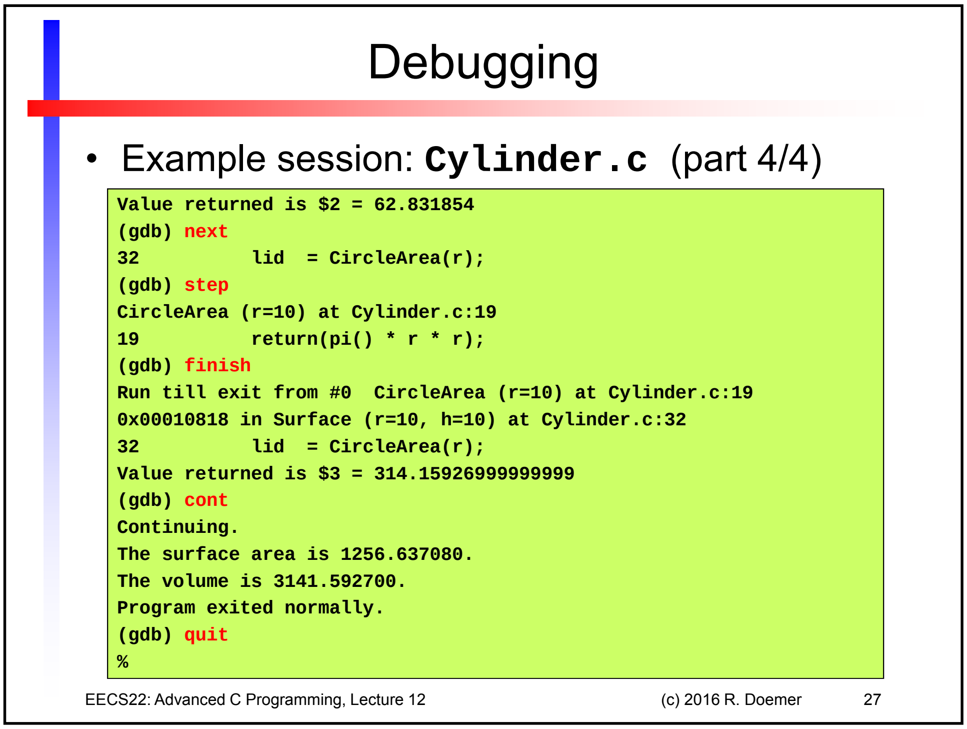


Debugging

- 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;
...
```

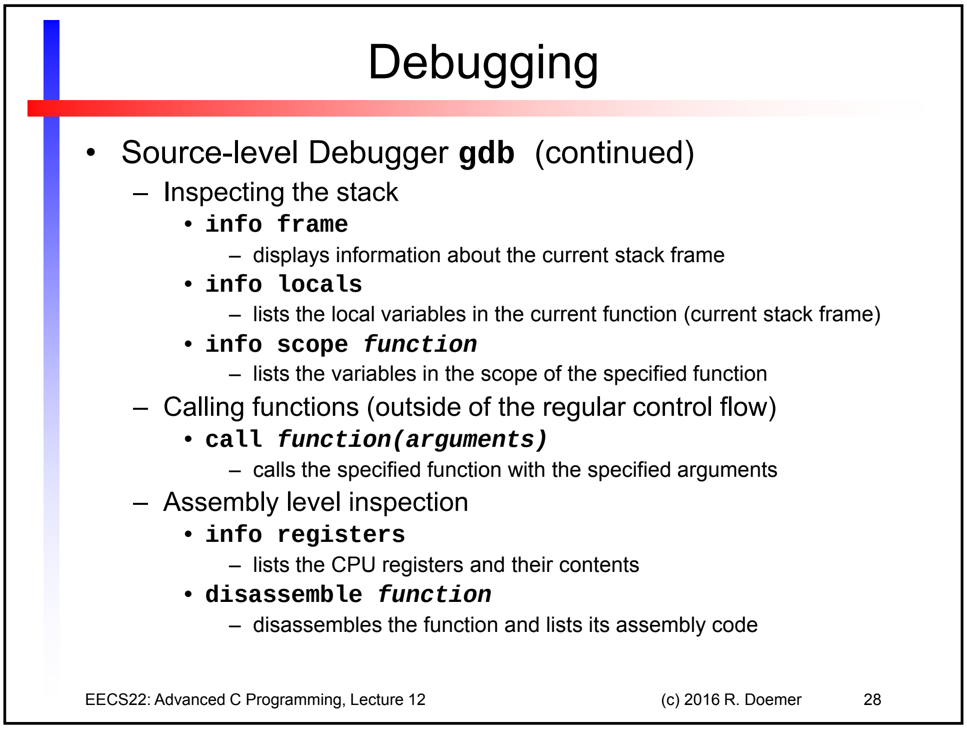
EE



Debugging

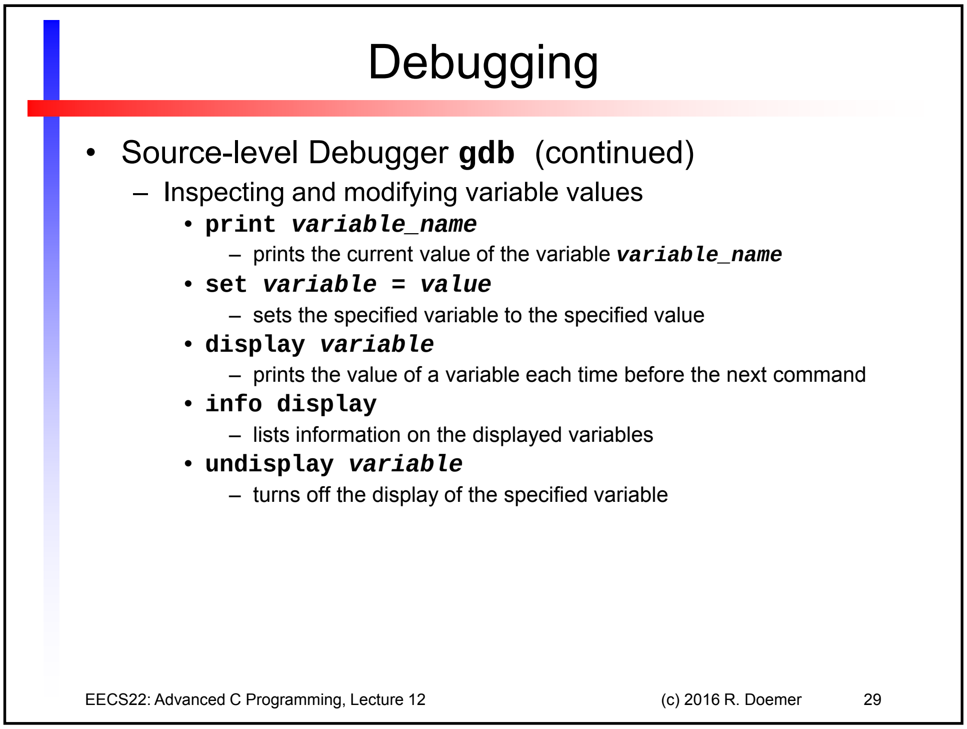
- Example session: **`Cylinder.c`** (part 4/4)

```
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
%
```



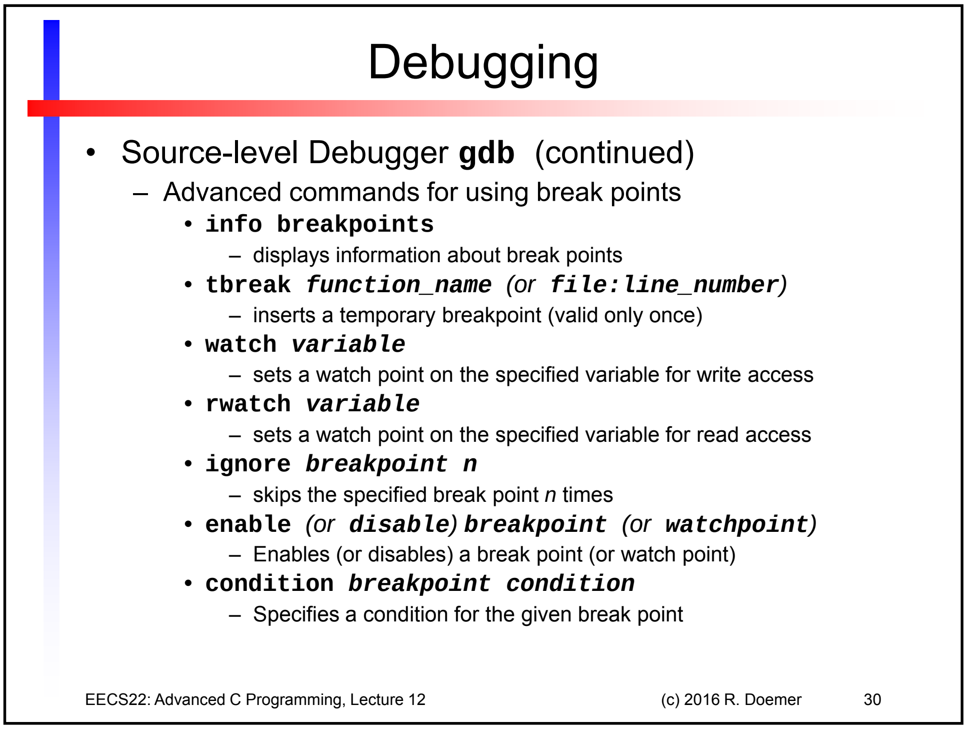
Debugging

- 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

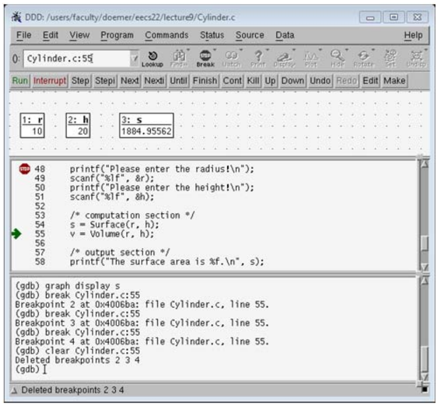


Debugging

- 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

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`**



```
48     printf("Please enter the radius!\n");
49     scanf("%lf", &r);
50     printf("Please enter the height!\n");
51     scanf("%lf", &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 0x4006ba: file Cylinder.c, line 55.
(gdb) break Cylinder.c:55
Breakpoint 3 at 0x4006ba: file Cylinder.c, line 55.
(gdb) break Cylinder.c:55
Breakpoint 4 at 0x4006ba: file Cylinder.c, line 55.
(gdb) clear Cylinder.c:55
Deleted breakpoints 2 3 4
(gdb) 
```

Deleted breakpoints 2 3 4