CSc 352: Basic C Programming

What's new relative to Java?

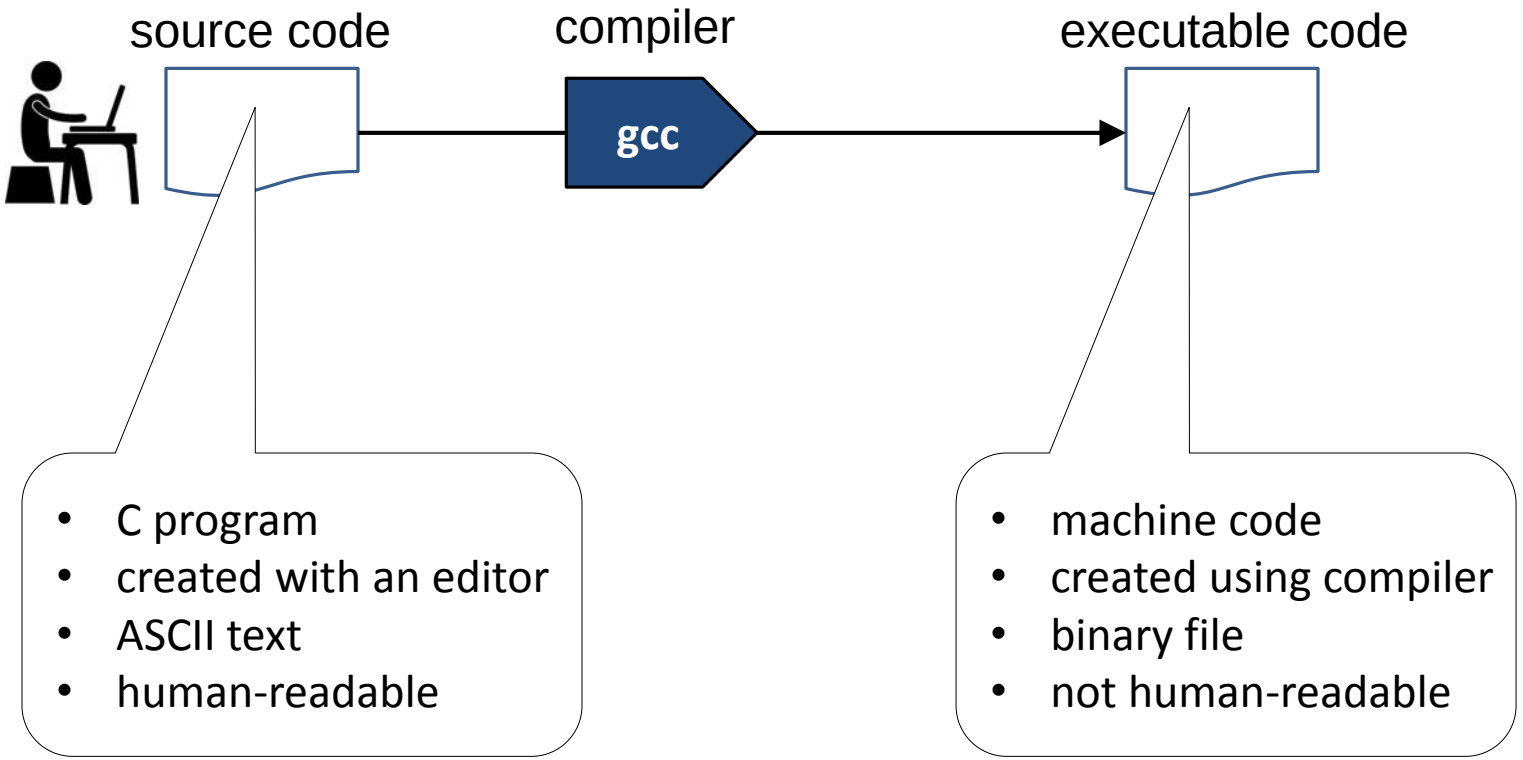
- Some syntactic differences
  - a lot of the basic syntax is similar
    - assignment, conditionals, loops, etc.
- The biggest differences are conceptual
  - procedural rather than object-oriented
    - no notion of classes and inheritance hierarchies
  - much closer to the machine
    - debugging sometimes requires thorough understanding of what's going on at the machine level
  - explicit dynamic memory management (malloc, free)
  - pointers

What's new relative to Java?

- No Garbage Collection
- No array boundary protection
- You have much more control
  - This means you can write faster programs
  - This also means code can be hard to debug and security vulnerabilities are much more likely
- C is generally compiled to "machine code" (java programs require java "machine" to be installed)

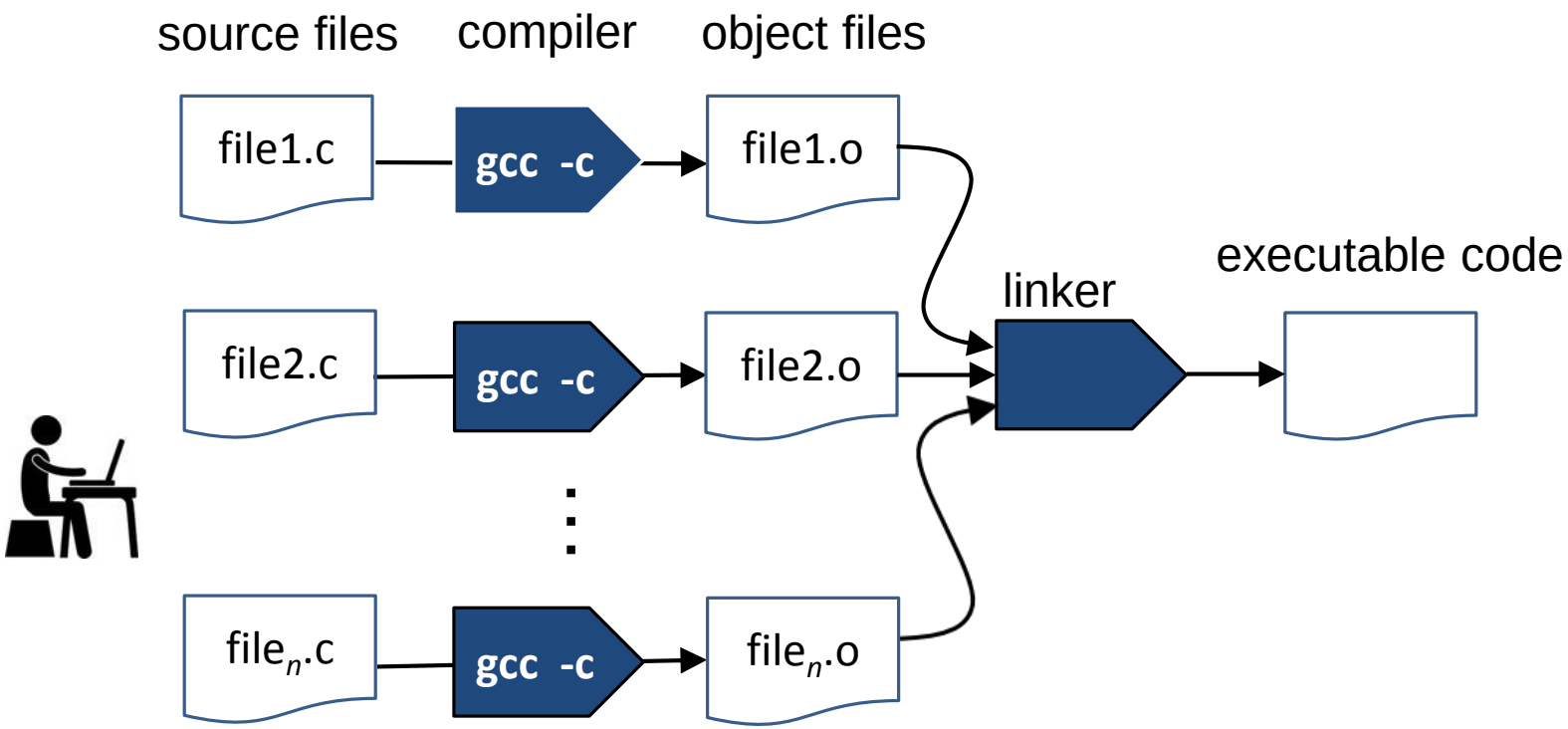
The program development process

Simple programs (single source file)



The program development process

More complex programs (many source files)



gcc

- The c compiler we'll be using on lectura in this class is gcc.
- gcc can create an executable file from a text file containing C code (called a source file)
  - The source code file should be name <file name>.c
  - The ".c" ending tells users that the file contains C source code
- gcc has MANY possible options, but in this class there are a couple we will almost always use:

gcc

- -Wall
  - this option tells gcc to print out all warning messages
  - your code must show no warnings when compiled with this option
- -o <file name>
  - This tells gcc to name the executable file <file name>
  - generally you will name the executable the same name as the source code file without the “.c”

gcc

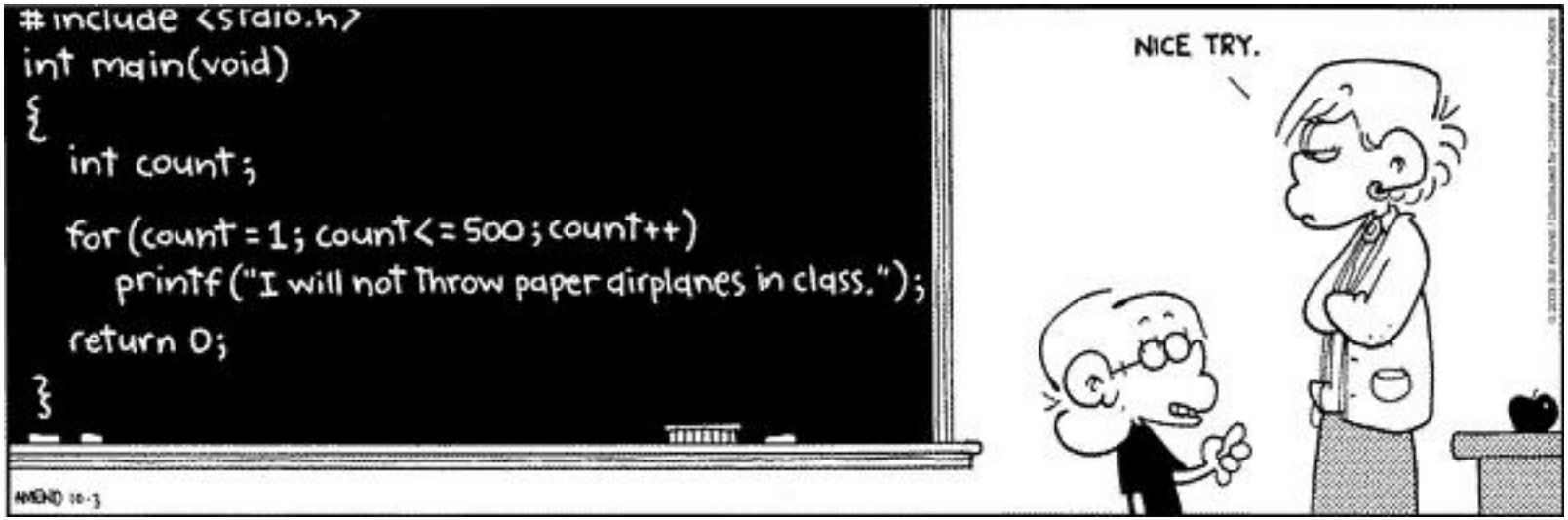
- For example, if you have a C source code file called project2.c, then typing

  ```
  gcc –Wall –o project2 project2.c
  ```

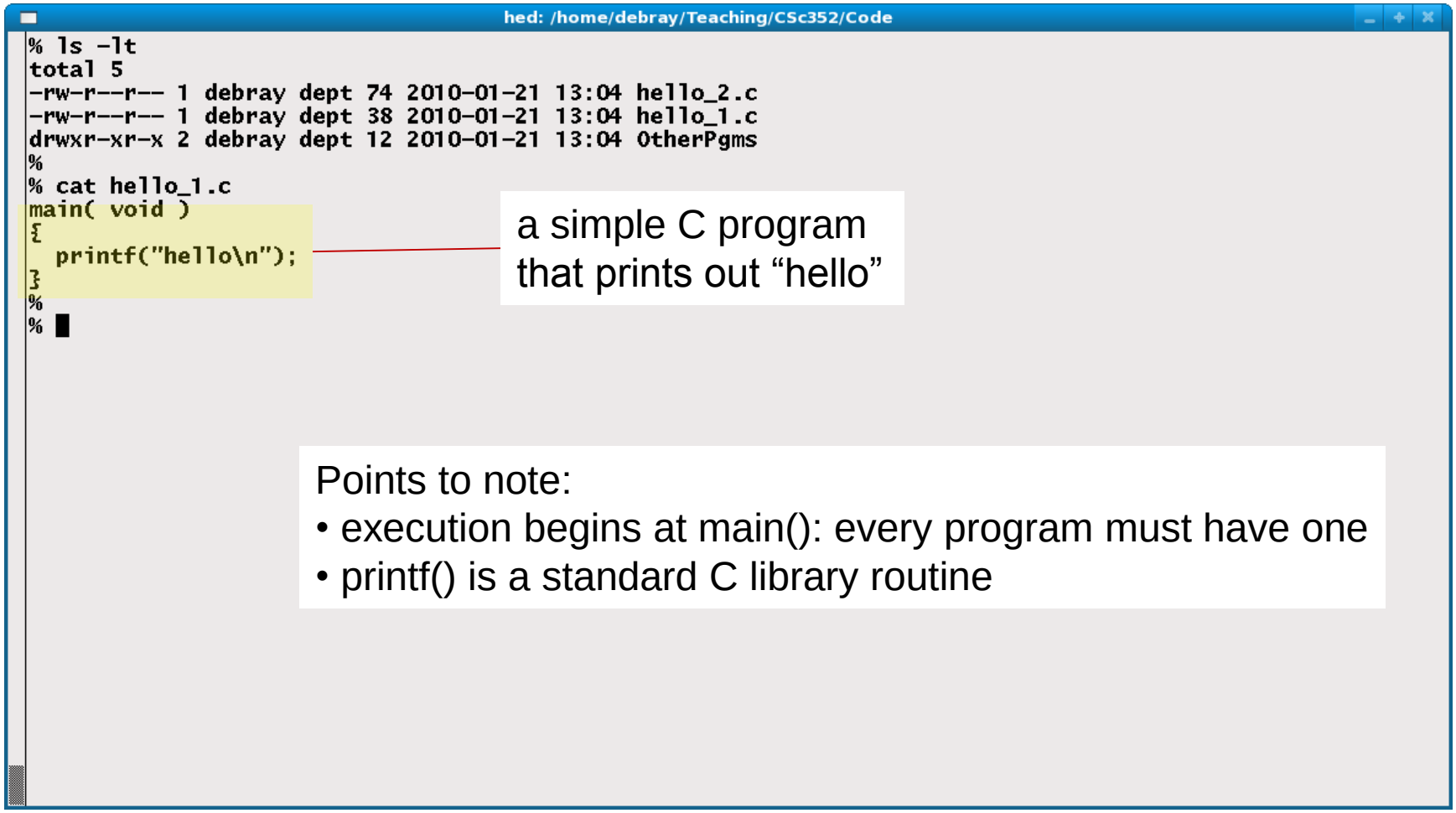
  will hopefully create an executable file called project2

- If the -o option is omitted, gcc will name the executable file a.out

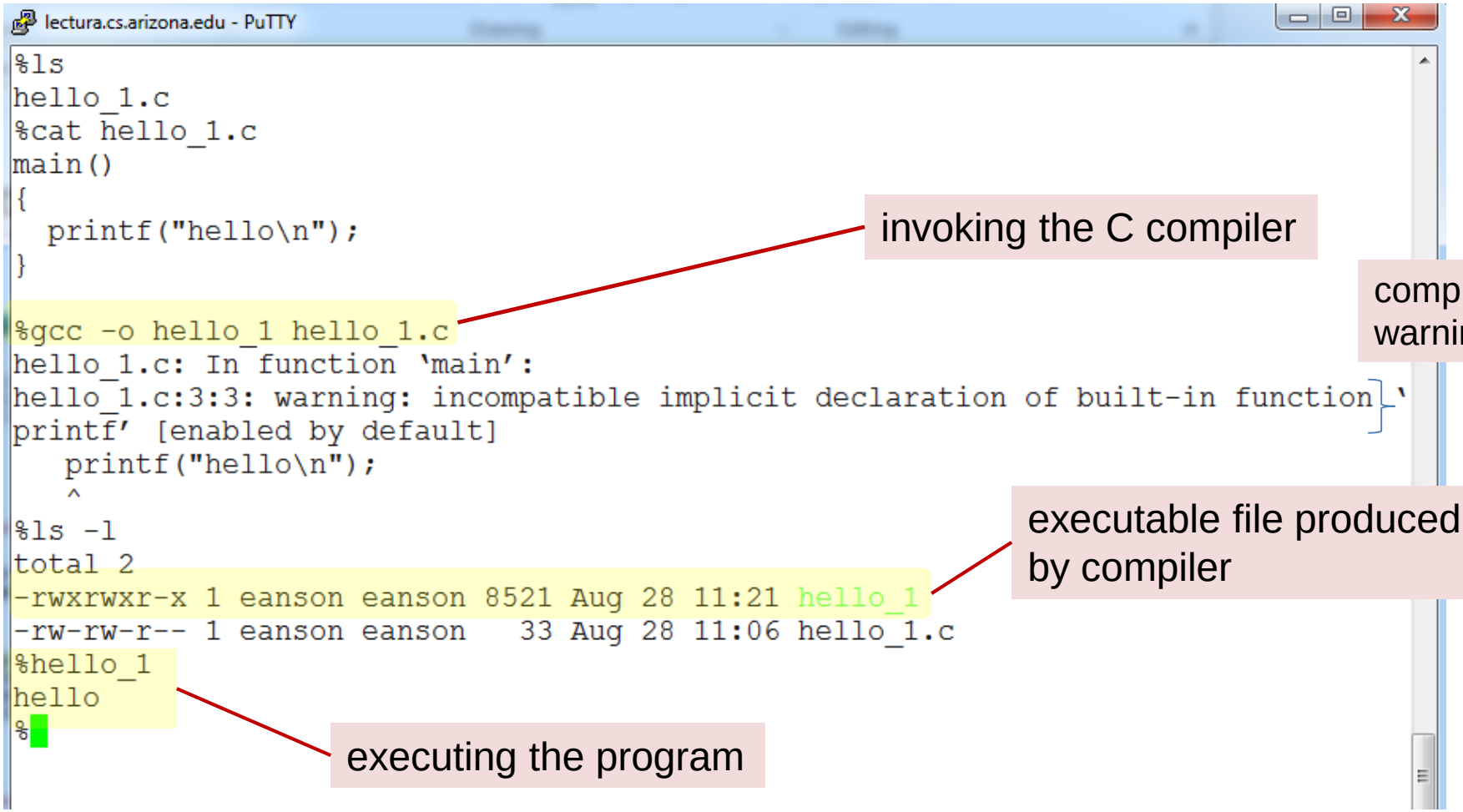
A simple program



A simple program

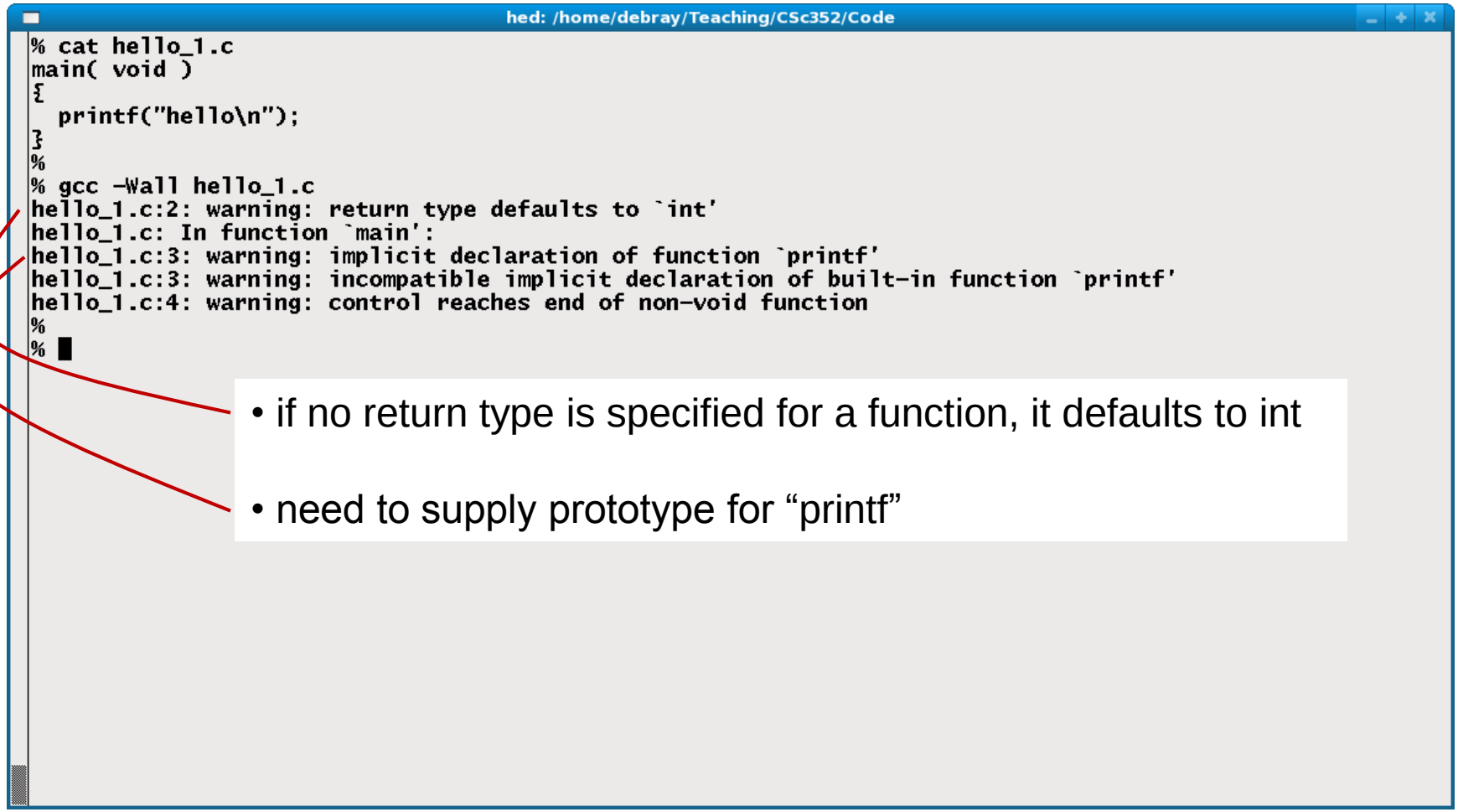


A simple program…

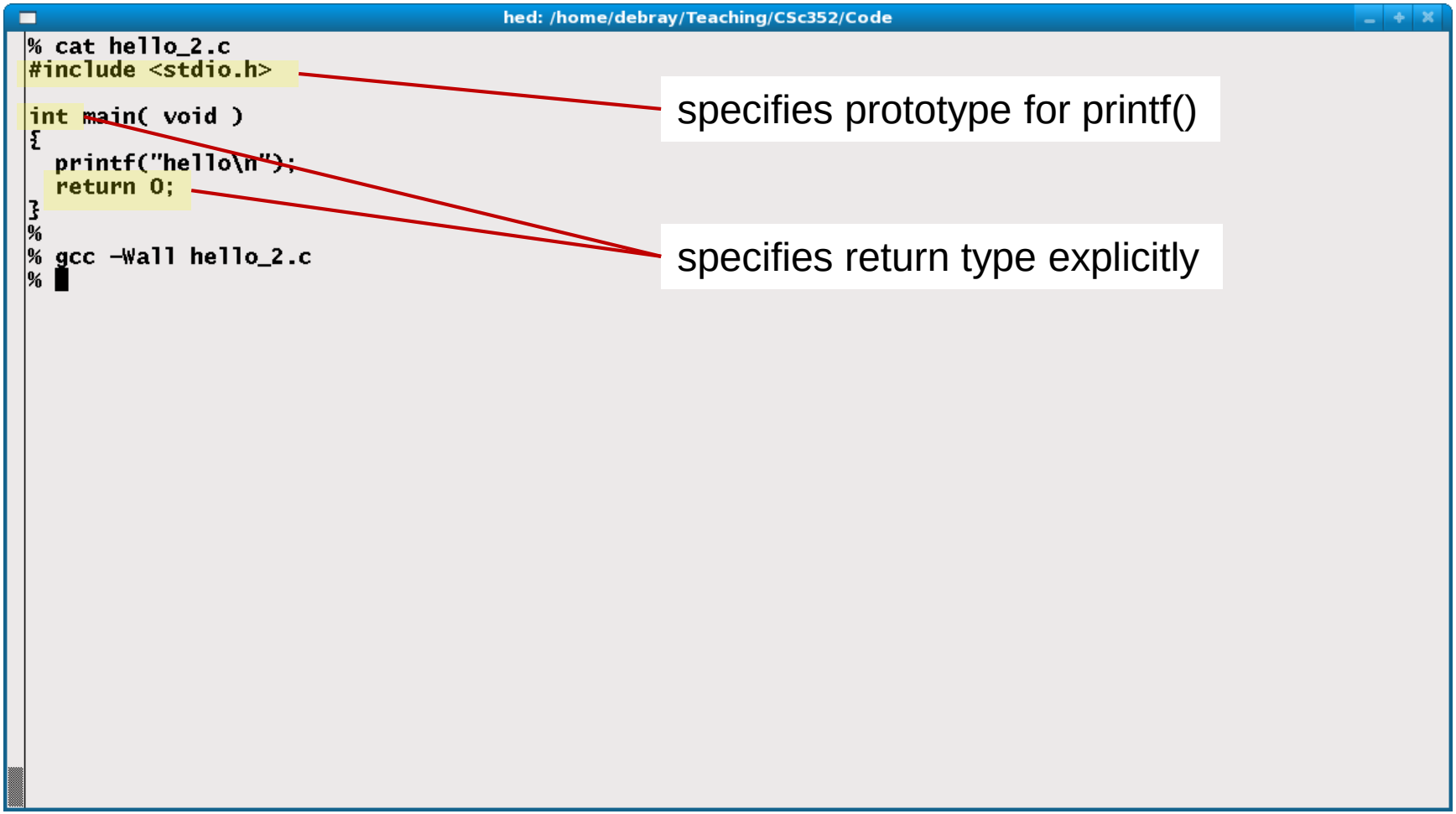


Gcc options: -Wall

gcc –Wall generates warnings on questionable constructs



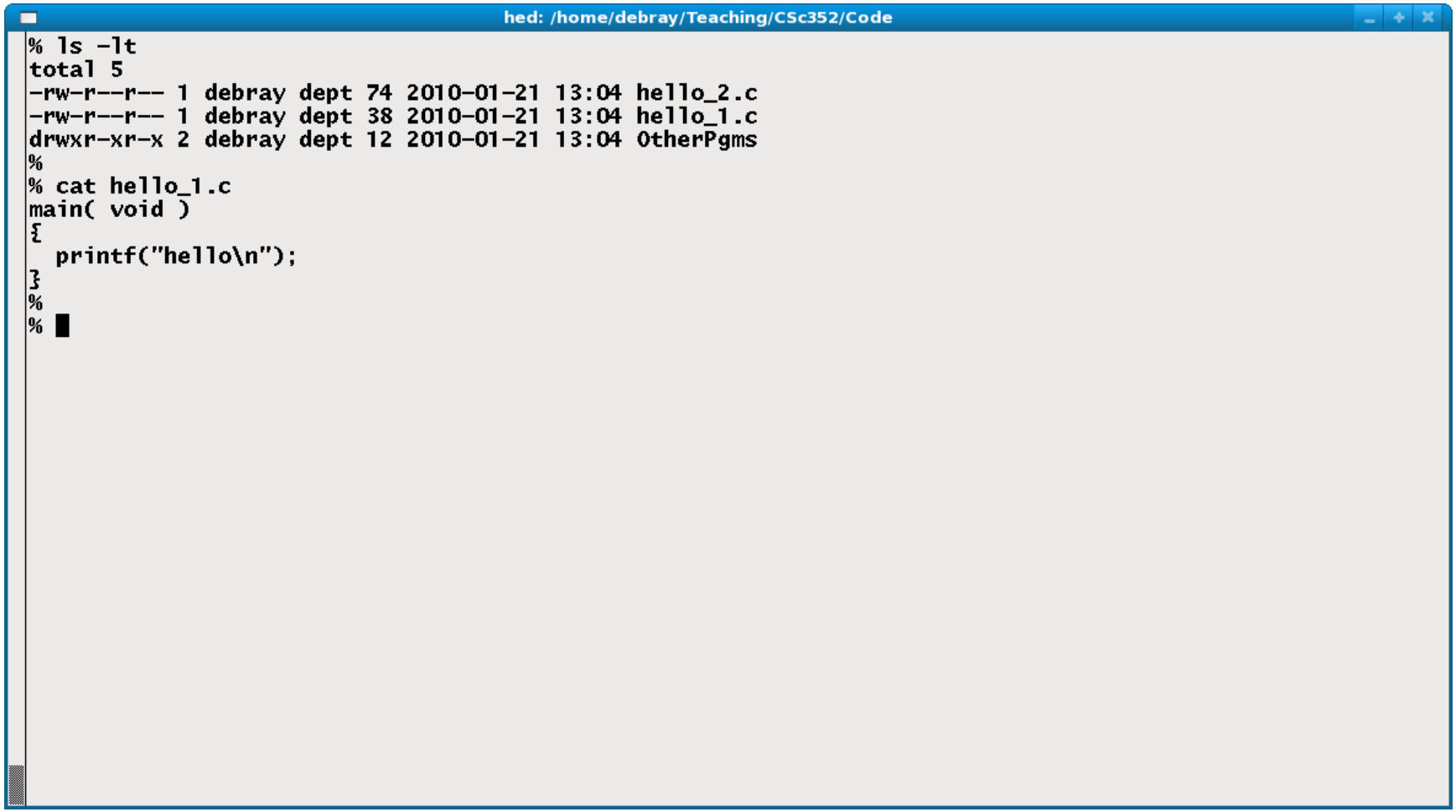
Fixing the compiler warnings



Summary

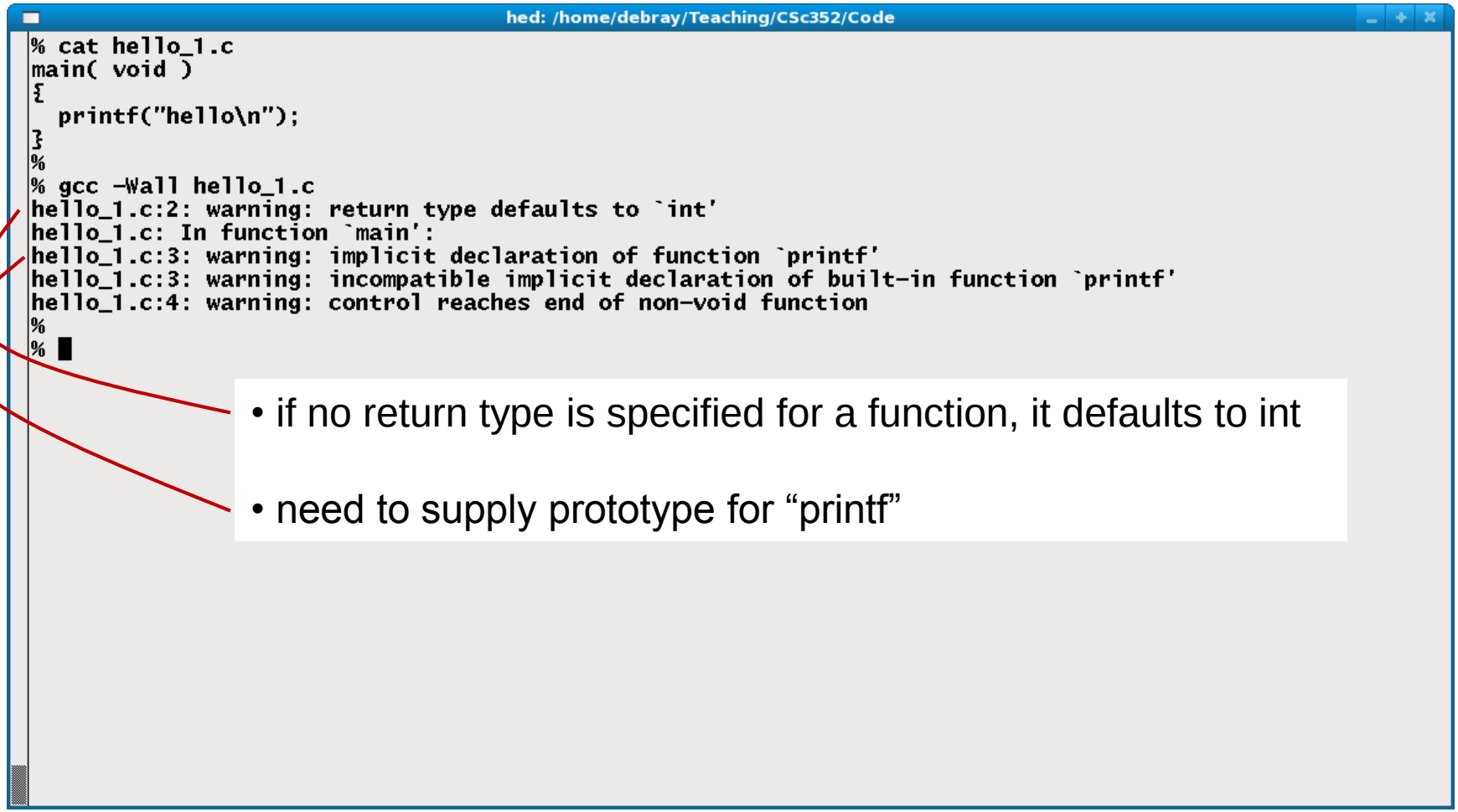
- execution starts at main()
  - every program should have one
- the return type for a function defaults to int
  - should specify explicitly: good style
- need to supply prototypes for functions imported from elsewhere (e.g., standard libraries)
  - specified using **“#include** …”

A simple program revisited

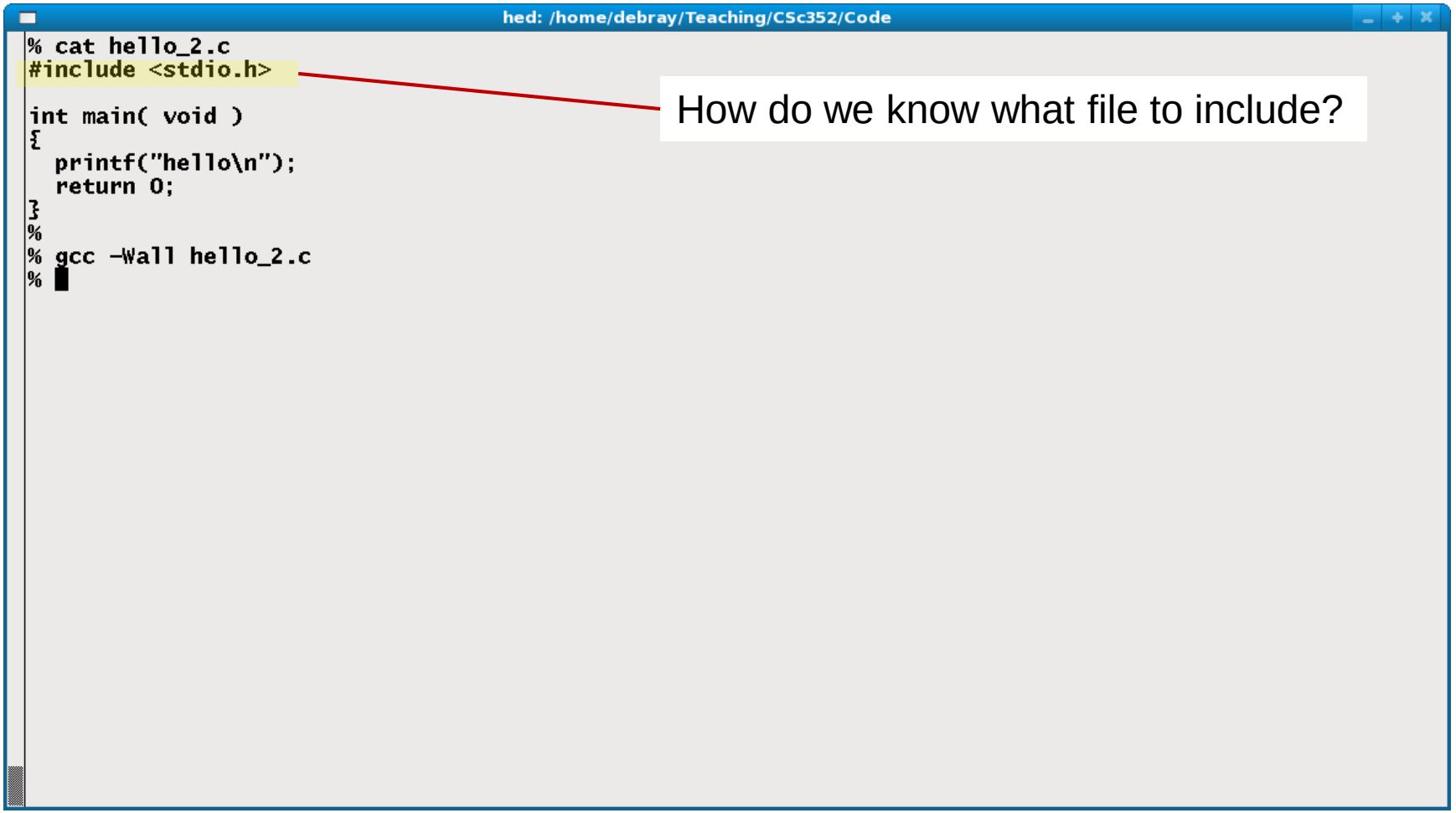


Gcc options: -Wall

gcc –Wall generates warnings on questionable constructs



Fixing the compiler warnings



The `man` command

- The **man** command displays documentation for commands (and more). Here is an abridged example—the "man page" for **cat**:

```
% man cat
CAT(1)                                    User Commands
CAT(1)


NAME
       cat - concatenate files and print on the standard output

SYNOPSIS
       cat [OPTION]... [FILE]...

DESCRIPTION
       Concatenate FILE(s), or standard input, to standard output.

       -A, --show-all
              equivalent to -vET
       ...

       With no FILE, or when FILE is -, read standard input.
```

Manual sections

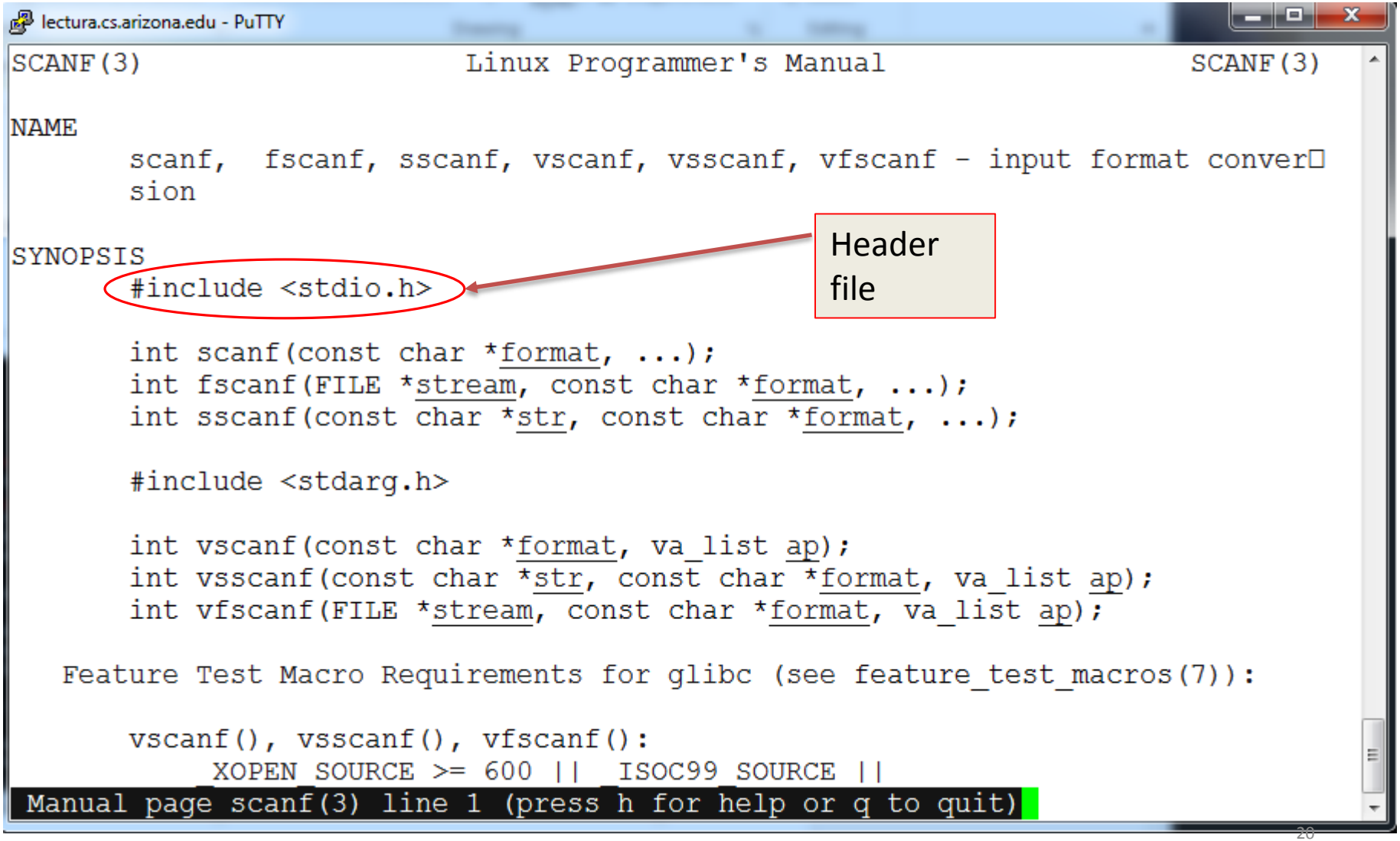
The UNIX "manual" is divided into these sections: (from `man man`)

1 User commands
2 System calls (functions provided by the kernel)
3 Library calls (functions within program libraries)
4 Special files (usually found in **/dev**)
5 File formats and conventions eg **/etc/passwd**
6 Games
7 Miscellaneous (including macro packages and conventions), e.g.
man(7), **groff(7)**
8 System administration commands (usually only for root)
9 Kernel routines [Non standard]

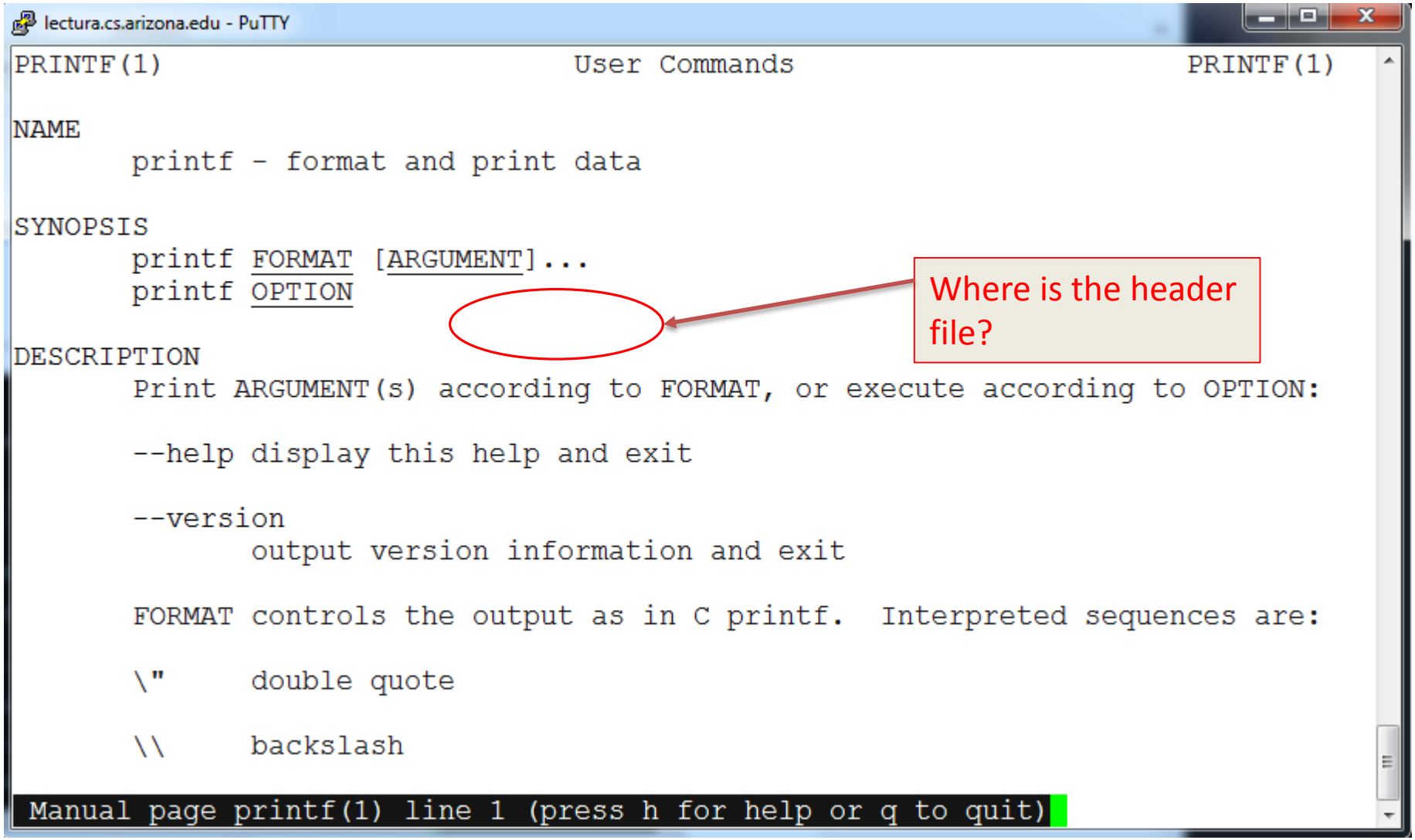
Recall that **man cat** showed `CAT(1)`. That "`(1)`" tells us that `cat` is a user command.

man malloc shows `MALLOC(3)`. That "`(3)`" tells us that `malloc` is a library function.

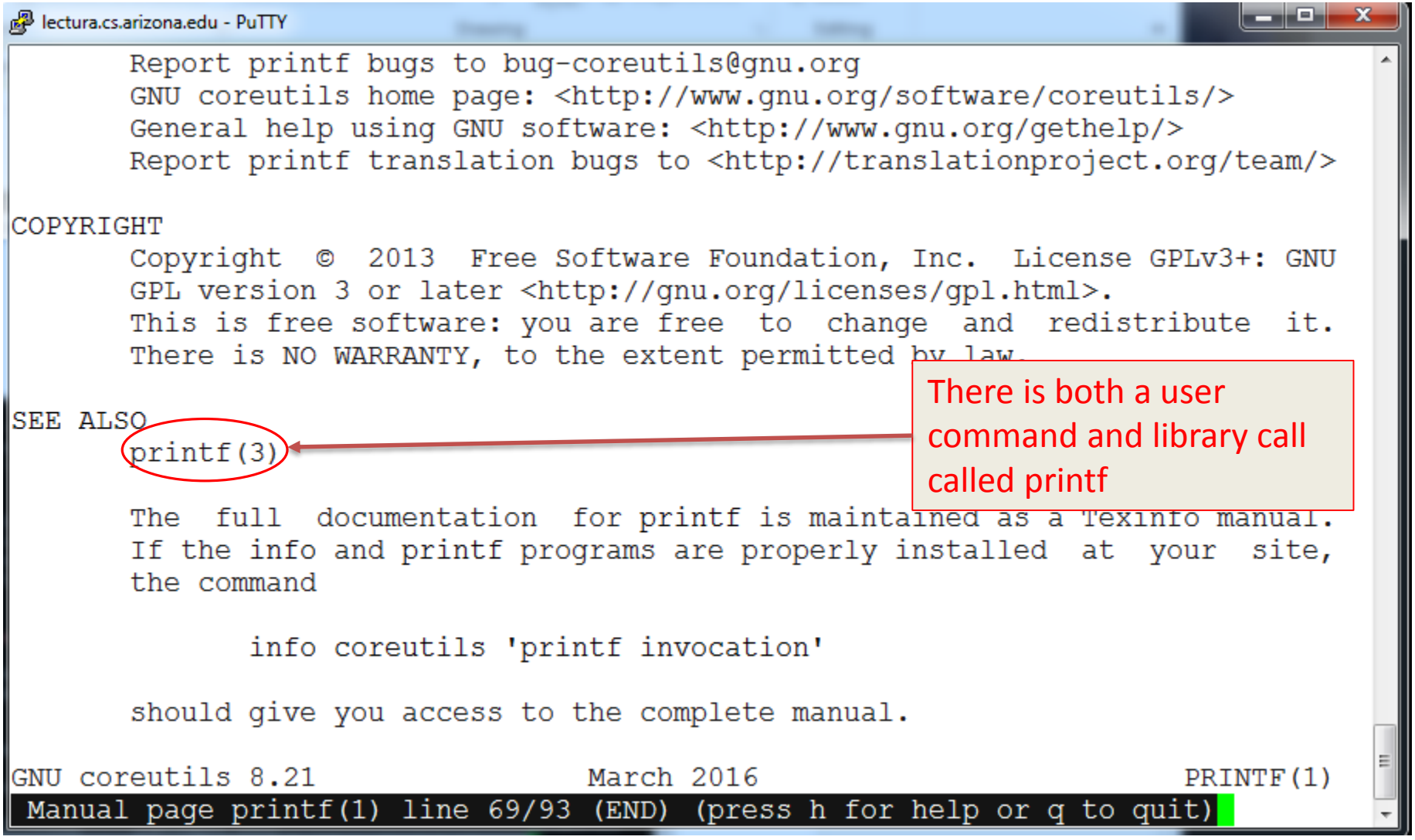
man scanf


lectura.cs.arizona.edu - PuTTY
SCANF(3) Linux Programmer's Manual SCANF(3)
NAME
scanf, fscanf, sscanf, vscanf, vsscanf, vfscanf - input format conver
sion
SYNOPSIS
#include <stdio.h>
int scanf(const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
int sscanf(const char *str, const char *format, ...);
#include <stdarg.h>
int vscanf(const char *format, va_list ap);
int vsscanf(const char *str, const char *format, va_list ap);
int vfscanf(FILE *stream, const char *format, va_list ap);
Feature Test Macro Requirements for glibc (see feature_test_macros(7)):
vscanf(), vsscanf(), vfscanf():
_XOPEN_SOURCE >= 600 || _ISOC99_SOURCE ||
Manual page scanf(3) line 1 (press h for help or q to quit)
Header file

man printf


lectura.cs.arizona.edu - PuTTY
PRINTF(1) User Commands PRINTF(1)
NAME
printf - format and print data
SYNOPSIS
printf FORMAT [ARGUMENT]...
printf OPTION
DESCRIPTION
Print ARGUMENT(s) according to FORMAT, or execute according to OPTION:
--help display this help and exit
--version
output version information and exit
FORMAT controls the output as in C printf. Interpreted sequences are:
\" double quote
\\ backslash
Manual page printf(1) line 1 (press h for help or q to quit)
Where is the header file?

man printf bottom

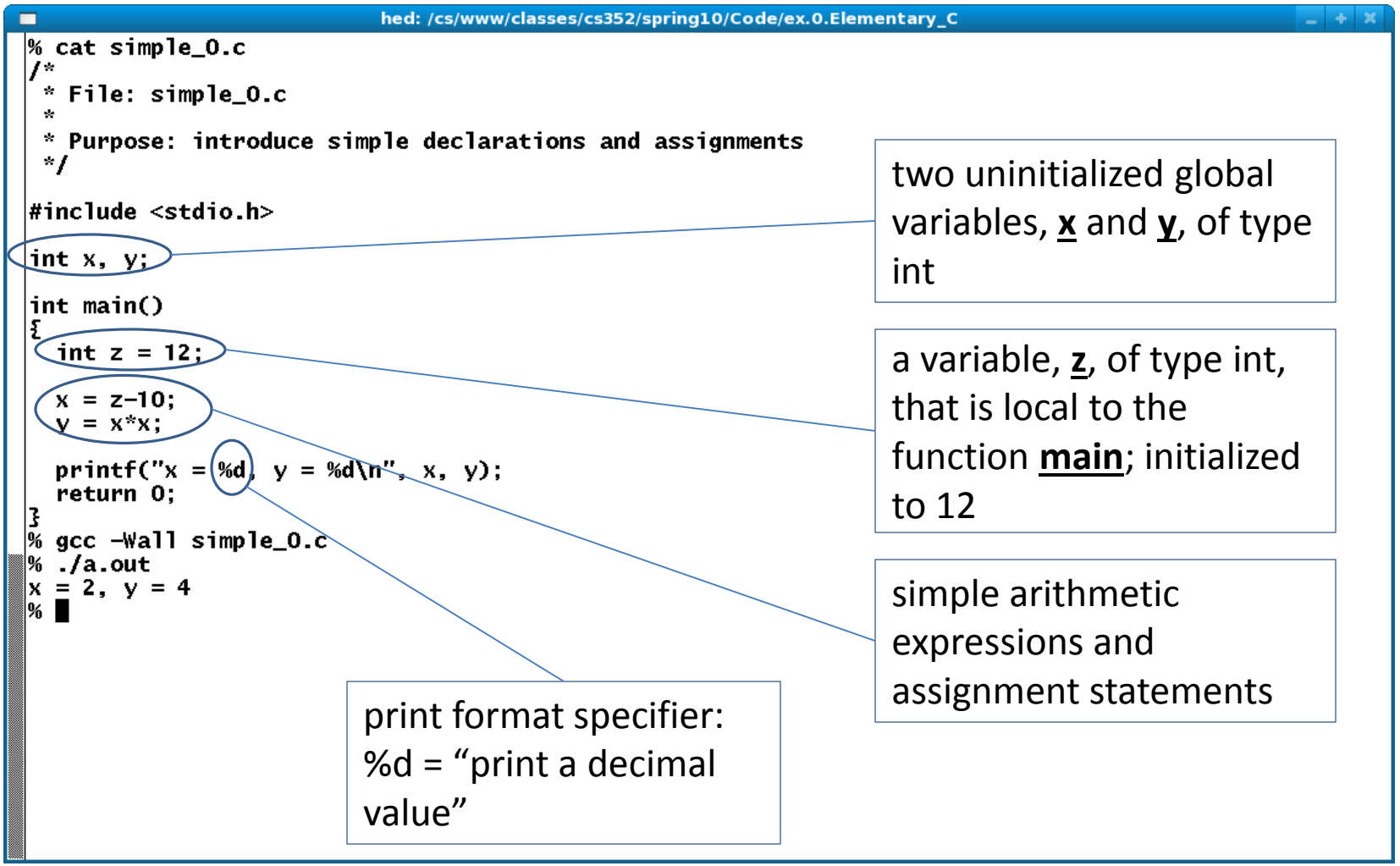


man 3 printf

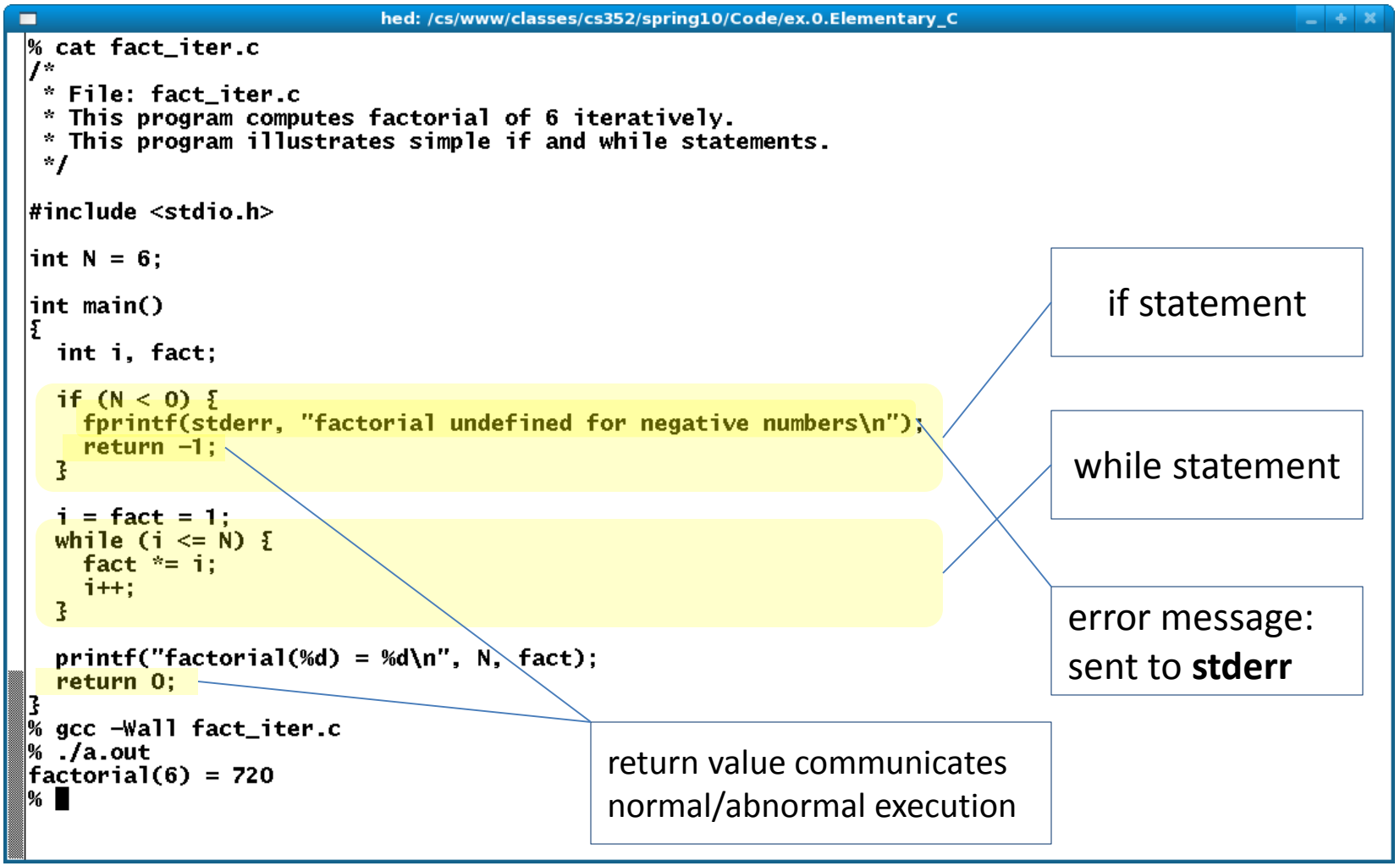
lectura.cs.arizona.edu - PuTTY

```
PRINTF(3)                  Linux Programmer's Manual                 PRINTF(3)

NAME
       printf,   fprintf,  sprintf,  snprintf,  vprintf,  vfprintf,  vsprintf,
       vsnprintf - formatted output conversion

SYNOPSIS
       #include <stdio.h>

       int printf(const char *format, ...);
       int fprintf(FILE *stream, const char *format, ...);
       int sprintf(char *str, const char *format, ...);
       int snprintf(char *str, size_t size, const char *format, ...);

       #include <stdarg.h>

       int vprintf(const char *format, va_list ap);
       int vfprintf(FILE *stream, const char *format, va_list ap);
       int vsprintf(char *str, const char *format, va_list ap);
       int vsnprintf(char *str, size_t size, const char *format, va_list ap);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

 Manual page printf(3) line 1 (press h for help or q to quit)
```

Simple declarations and statements



Simple conditionals, while loops



For loops

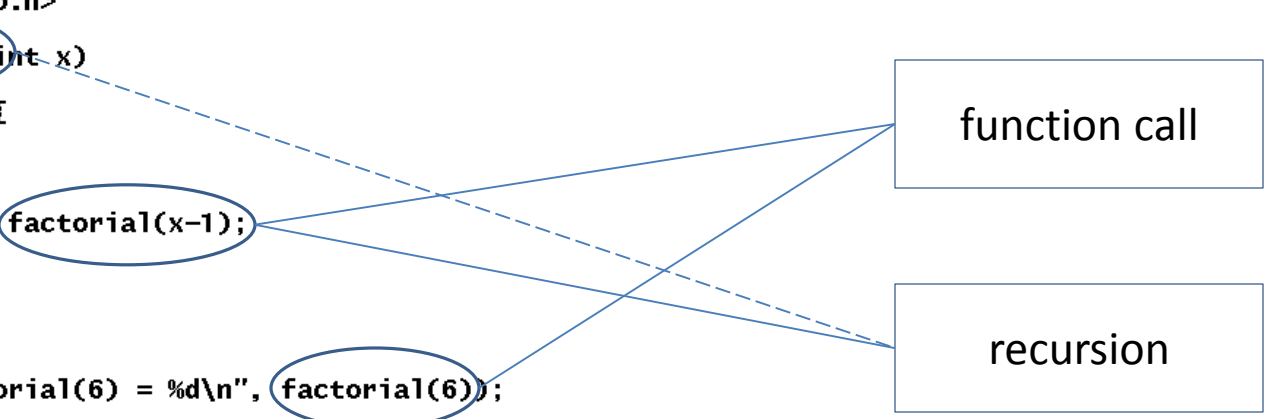
hed: /cs/www/classes/cs352/spring10/Code/ex.0.Elementary_C

```
% cat fact_iter-1.c
/*
 * File: fact_iter-1.c
 * This program computes factorial of 6 iteratively.
 * This program illustrates simple for statements.
 */

#include <stdio.h>

int N = 6;

int main()
{
  int i, fact;

  if (N < 0) {
    fprintf(stderr, "factorial undefined for negative numbers\n");
    return -1;
  }

  for (i = 1, fact = 1; i <= N; i++) {
    fact *= i;
  }

  printf("factorial(%d) = %d\n", N, fact);
  return 0;
}
% gcc -Wall fact_iter-1.c
% ./a.out
factorial(6) = 720
%
```

Function calls, recursion

hed: /cs/www/classes/cs352/spring10/Code/ex.0.Elementary_C

```
% cat fact_rec.c
/*
 * File: fact_rec.c
 * This program computes factorial of 6 recursively.
 * This program illustrates (recursive) function calls.
 */

#include <stdio.h>

int factorial(int x)
{
  if (x == 0) {
    return 1;
  }
  else {
    return x * factorial(x-1);
  }
}

int main()
{
  printf("factorial(6) = %d\n", factorial(6));
  return 0;
}
%
% gcc -Wall fact_rec.c
% ./a.out
factorial(6) = 720
%
```



Formatted Output: printf()

text: Ch. 3
Sec. 3.1

- takes a variable no. of arguments:
  - printf("…*fmtStr*…", arg_1, arg_2, …, arg_n)
    - "… *fmtStr*…" is a string that can contain *conversion specifiers*
    - the no. of conversion specifiers should be equal to n
    - "regular" (non-%) characters in *fmtStr* written out unchanged
  - each conversion specifier is introduced by '%'
    - this is followed by additional (optional) characters specifying how wide the output is to be, precision, padding, etc.
    - the conversion specifier indicates how the specified value is to be interpreted:
      - **d** = decimal integer, e.g.: printf("value = **%d**\n", n);
      - **x** = hex integer, e.g.: printf("hex value of %d is **%x**\n", x, x);
      - **f** = floating point number, e.g.: printf("area = **%f**\n", A);

Function calls (cont'd)

hed: /cs/www/classes/cs352/spring10/Code/ex.0.Elementary_C

```
% cat fact_rec.c
/*
 * File: fact_rec.c
 * This program computes factorial of 6 recursively.
 * This program illustrates (recursive) function calls.
 */

#include <stdio.h>

int factorial(int x)
{
  if (x == 0) {
    return 1;
  }
  else {
    return x * factorial(x-1);
  }
}

int main()
{
  printf("factorial(6) = %d\n", factorial(6));
  return 0;
}
%
% gcc -Wall fact_rec.c
% ./a.out
factorial(6) = 720
%
```

What happens when this printf() is called?

- the arguments are evaluated (as in all C function calls)
  - **factorial(6)** evaluated
- when factorial() returns, the printf is executed:
  - **printf(" … ", 720)**
- This causes the string

  factorial(6) = 720

  to be printed out

Formatted Input: scanf()

text: Ch. 3
Sec. 3.2

- takes a variable no. of arguments:
  - scanf("…*fmtStr*…", arg_1, arg_2, …, arg_n)
    - "… *fmtStr*…" is a string that can contain ***conversion specifiers***
    - the no. of conversion specifiers should be equal to n
    - arg_i are *locations* where values that are read in should be placed
    - each conversion specifier is introduced by '%'
      - similar to conversions for printf
- execution behavior:
  - uses format string to read in as many input values from stdin as it can
  - return value indicates the no. of values it was able to read
    - return value of EOF (-1) indicates no further input ("end of file").

scanf()

- Specifying where to put an input value:
  - in general we need to provide a pointer to the location where the value should be placed
    - for a scalar variable **X**, this is typically written as **&X**
  - **Examples:**
    - scanf("%d", &**n**) : read a decimal value into a variable **n**
    - scanf("%d %d %d", &**x**, &**y**, &**z**) : read three decimal values and put them into variables **x**, **y**, **z** respectively
      - suppose the input contains the values 12, 3, 71, 95, 101. Then:
        - » x ← 12; y ← 3; z ← 71; return value = 3
      - suppose the input contains the values 19, 23. Then:
        - » x ← 19; y ← 23; z is unassigned; return value = 2

Exercise

Which of these are correct?

```
int x;
printf("%d\n", x);
```

```
int x, y;
printf("%d", x, y);
```

```
int x, y, z;
printf("%d %d %d", x, y, z);
```

```
int x, y;
printf(%d, %d, x, y);
```

```
int f(int n);
printf("f(%d)= %d\n", n, f(n));
```

```
int x, y;
printf("%d %d\n", x);
```

Exercise

Which of these are correct?

`int x;` `printf("%d\n", x);`	✓
`int x, y;` `printf("%d", x, y);`	X wrong no. of arguments
`int x, y, z;` `printf("%d %d %d", x, y, z);`	✓
`int x, y;` `printf("%d, %d", x, y);`	X format specifier needs to be a string " ... "
`int f(int n);` `printf("f(%d)= %d\n", n, f(n));`	✓
`int x, y;` `printf("%d %d\n", x);`	X wrong no. of arguments

Exercise

Which of these are correct?

```
int x;
scanf("%d\n", x);
```

```
int x, y;
scanf("%d %d", &x, &y);
```

```
int x, y, z;
scanf("%d %d %d", &x, &y, &z);
```

```
int x, y;
scanf(%d, %d, &x, &y);
```

```
int f(int n);
scanf("f(%d)= %d\n", &n, &f(n));
```

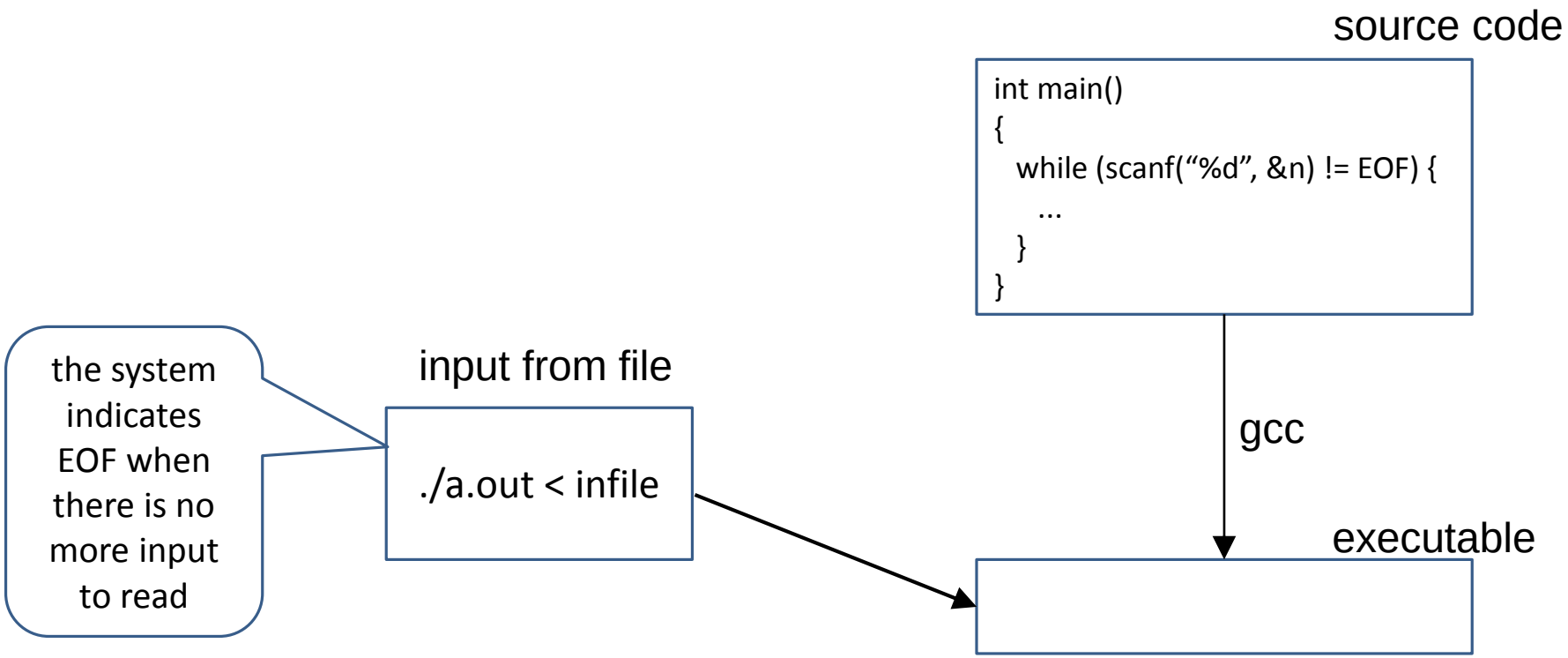
```
int x, y;
scanf("%d %d\n", &x);
```

Exercise

Which of these are correct?

`int x;` `scanf("%d\n", x);`	✗ need &x
`int x, y;` `scanf("%d %d", &x, &y);`	✓
`int x, y, z;` `scanf("%d %d %d", &x, &y, &z);`	✓
`int x, y;` `scanf(%d, %d, &x, &y);`	✗ format specifier needs to be a string " ... "
`int f(int n);` `scanf("f(%d)= %d\n", &n, &f(n));`	✗ &f(n) doesn't make sense
`int x, y;` `scanf("%d %d\n", &x);`	✗ wrong no. of arguments

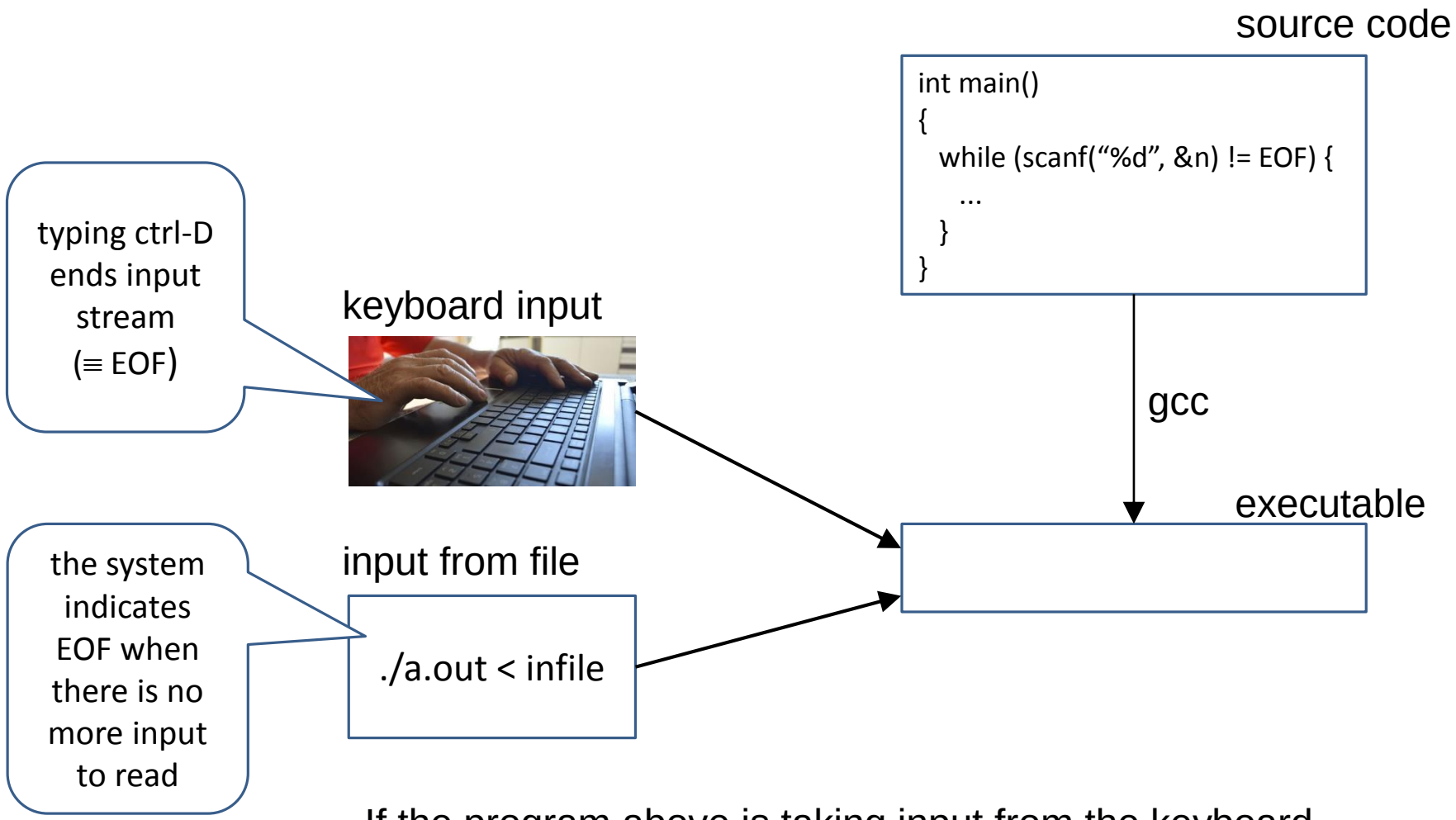
Reading inputs: ^D and end-of-file



Assuming the input is good, the program will above will terminate after all of the data from the file infile is read.

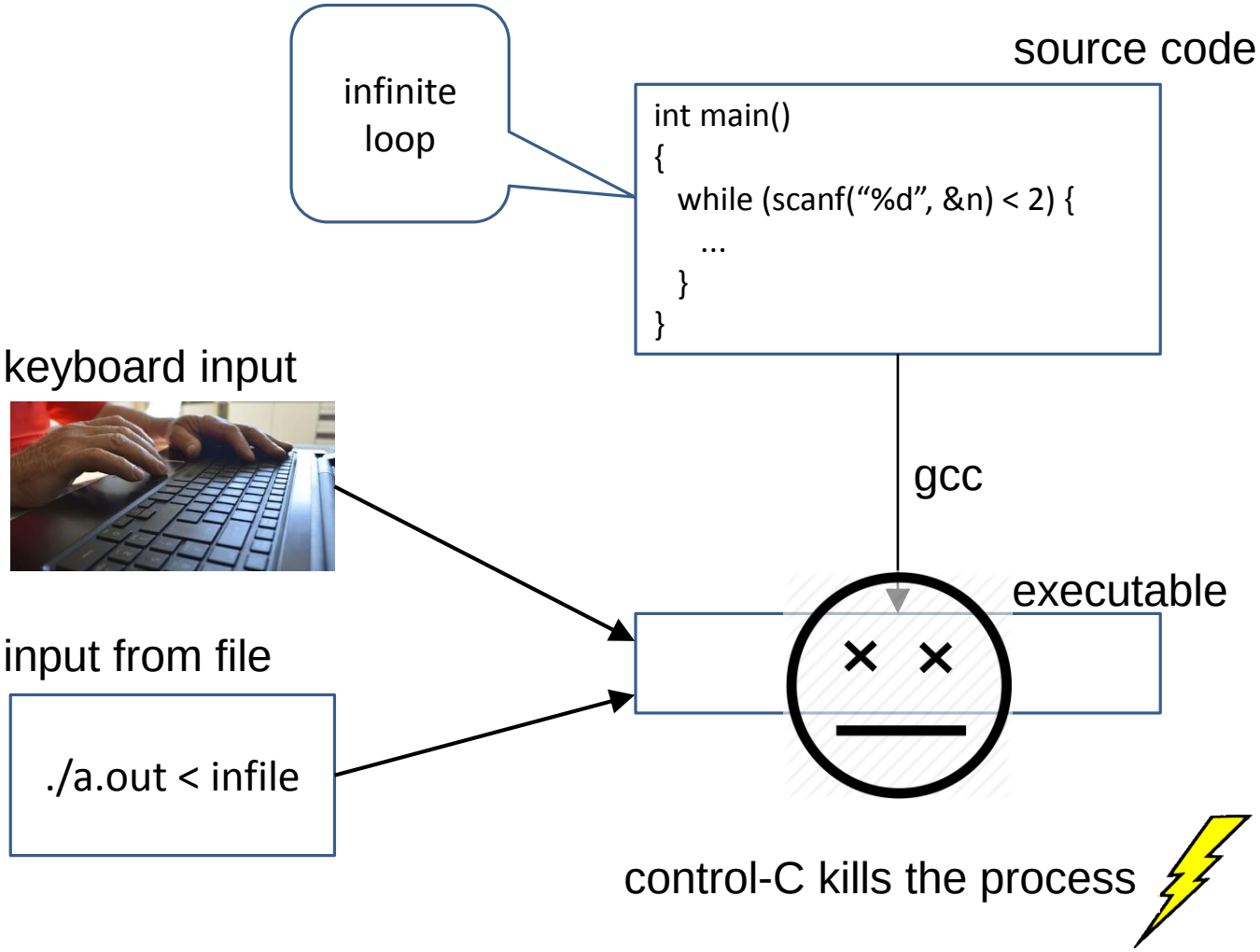
How do you indicate the end of file (EOF) when input comes from the keyboard?

Reading inputs: ^D and end-of-file



If the program above is taking input from the keyboard, pressing ctrl-d will send an EOF causing the loop condition to fail, and the program to exit

Reading inputs: ^C, ^D, and end-of-file



Assignment: *l*-values

hed: /home/debray/tmp

```
% cat pgm1.c
#include <stdio.h>

int x;

int main() {
  x = 0;
  x++;

  printf("x = %d\n", x);
  return 0;
}
% gcc -Wall pgm1.c
% ./a.out
x = 1
%
```

hed: /home/debray/tmp

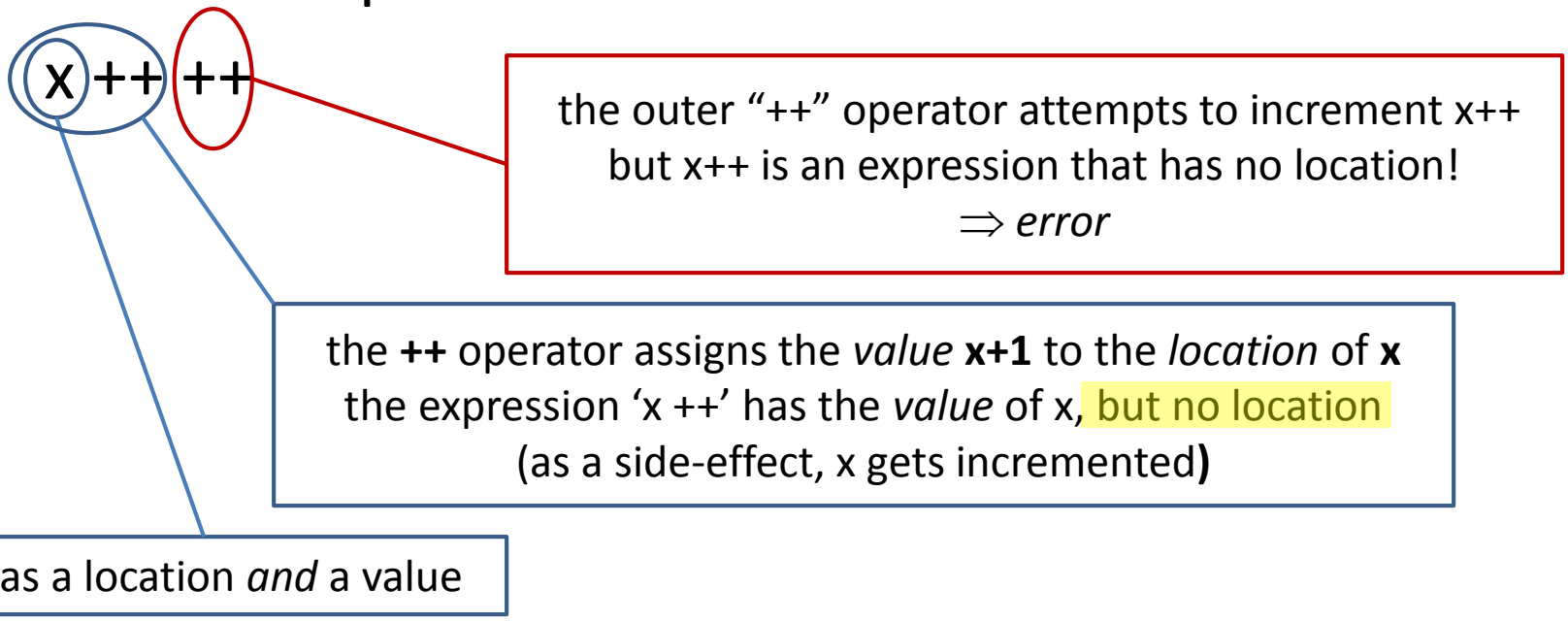
```
% cat pgm2.c
#include <stdio.h>

int x;

int main() {
  x = 0;
  x++ ++;

  printf("x = %d\n", x);
  return 0;
}
% gcc -Wall pgm2.c
pgm2.c: In function `main':
pgm2.c:7:7: error: lvalue required as
increment operand
%
```

ice?

what's going on?

Assignment: *l*-values

- The left-hand-side (destination) of an assignment has to be a *location*
  - this is referred to as an "*l*-value"
- Consider an expression



What can be an *l*-value?

- *l*-values:
  - names of variables of arithmetic type (int, char, float, etc.)
  - array elements: $x[y]$
  - also:
    - **struct**s, **union**s, pointers, etc. (to be discussed later)
    - operations involving pointers (to be discussed later)
- Not *l*-values:
  - functions
  - result of an assignment
  - value returned by a function call

Expressions and Statements

- A statement is a command to be executed when a program is run
  - printf(“hello\n”);
  - x = 5;
- An expression is something which computes to a value
  - 5
  - x + y
  - x == 8
- In c, all expressions can be statements just by adding ;
  - 5;
  - x + y;

Expressions and Statements

- In c many things you might not expect are expressions like assignment
  - x = 8 /* evaluates to the value 8 */
- Loosely, a ***side effect*** is an action that happens while evaluating an expression
  - scanf("%d", &i)
    - evaluates to the number of variables assigned values | side effect: stores value in i
  - x++
    - evaluates to x | side effect: stores the value of x + 1 in x
  - x = 8
    - evaluates to 8 | side effect stores the value 8 in x

Expressions and Statements

- So the assignment = in c is really an operator that returns the value to the right and has a side effect of storing that value in the variable to the left
- This is why x = y = z = 1 is legal in c. The assignment operator is evaluated from left to right.
- a = b += c++ - d + --e / -f
  - is legal, well defined, but BAD programming (DON'T DO IT) Would you want to evaluate it?

Expressions and Statements

```
a = 5;
c = (b = a + 2) - (a = 1);
```

This is legal but NOT well defined. Sometimes it will evaluate to 6 and sometimes 2. Even on the same machine.

My Most Common c Error

```
if (x = 5) {   . . . }
```

I had meant `x == 5`, but I typed it wrong. But in c, this is still legal.

x = 5 stores the value of 5 in x and evaluates to 5

5 is considered “true” by c (everything not 0 is true), so the if statement ALWAYS executes.

Fortunately, the –Wall option for gcc will warn about this error.

Primitive data types: Java vs. C

- C provides some numeric types not available in Java
  - unsigned
- The C language provides only a "minimum size" guarantee for primitive types
  - the actual size may vary across processors and compilers
- Originally C did not have a boolean type
  - faked it with **int**s: 0 ≡ *false*; non-0 ≡ *true*
    - hence code of the form:

```
if ( (x = getnum()) ) { … } // if value read is nonzero
```

  - C99 provides some support for booleans

Primitive numeric types: Java vs. C

Java	C	C size	Comments
byte	unsigned char	typically (and at least) 8 bits	
	signed char	typically (and at least) 8 bits	
	char	typically (and at least) 8 bits	signedness is implementation dependent
short	unsigned short int	typically (and at least) 16 bits	
	signed short int	same as unsigned short	
int	unsigned int	16 or 32 bits	the "natural" size for the machine
	signed int	*same as* unsigned int	
?	unsigned long int	typically (and at least) 32 bits	
	signed long int	== unsigned long	
long	unsigned long long int	typically (and at least) 64 bits	
	signed long long int	== unsigned long long	

Note: the keywords in gray may be omitted

Signed vs. unsigned values

- Essential idea:
  - "signed" : the highest bit of the value is interpreted as the sign bit (0 = +ve; 1 = –ve)
  - "unsigned" : the highest bit not interpreted as sign bit
- For an *n*-bit value:
  - signed: value ranges from -2^{n-1} to $+2^{n-1}-1$
  - unsigned: value ranges from 0 to $2^{n}-1$
- Right-shift operator (>>) may behave differently
  - *unsigned values*: 0s shifted in on the left
  - *(signed) negative values*: bit shifted in is implementation dependent

Booleans

- Originally, C didn't have a separate **boolean** type
  - truth values were **int**s: 0 = *false*; non-0 = *true*
  - still commonly used in programming
- C99 provides the type **_Bool**
  - **_Bool** is actually an (unsigned) integer type, but can only be assigned values 0 or 1, e.g.:

```
_Bool flag;
flag = 5;    /* flag is assigned the value 1 */
```

- C99 also provides boolean macros in stdbool.h:

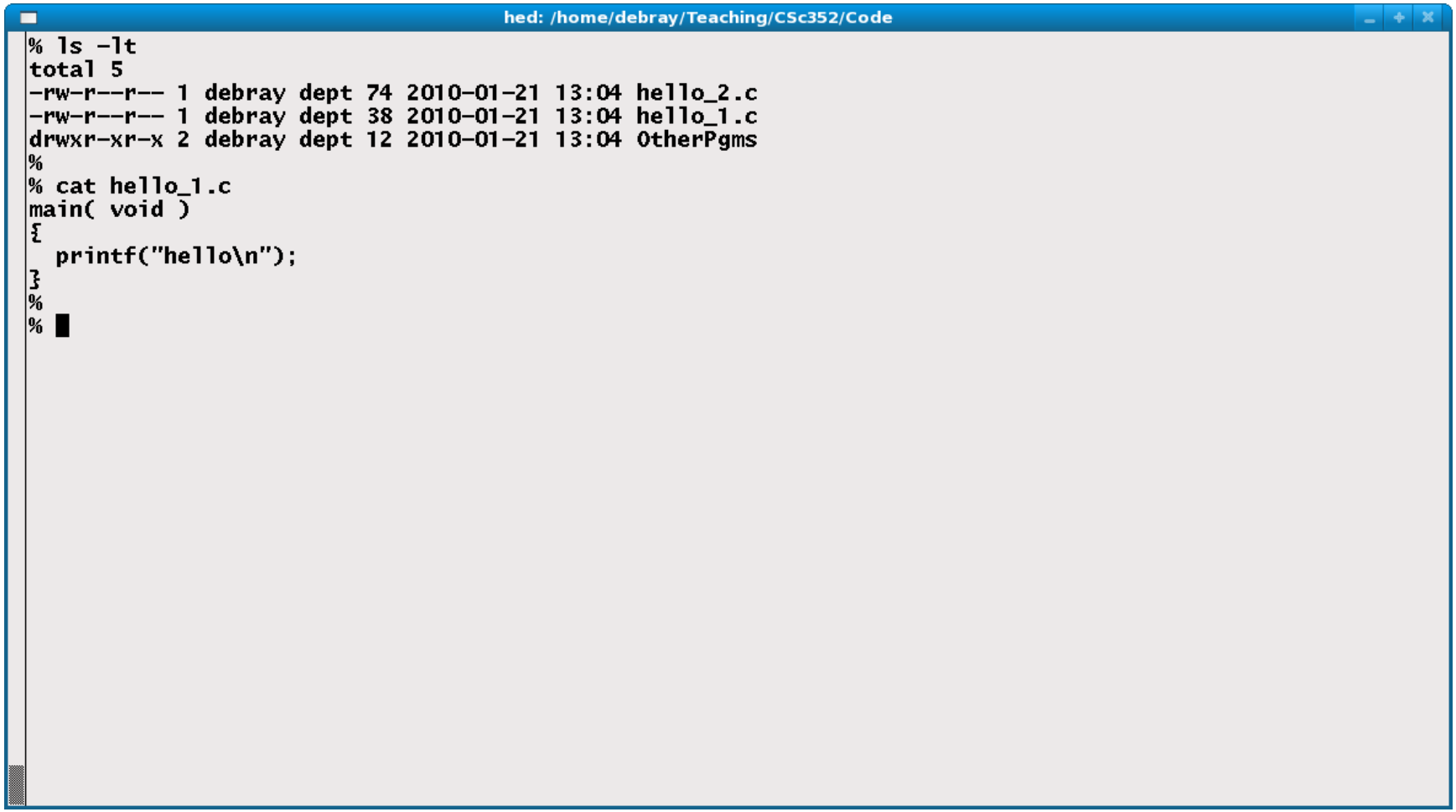
```
#include <stdbool.h>
bool flag;    /* same as _Bool flag */
flag = true;
```

Arithmetic operators: Java vs. C

- Most of the common operators in C are as in Java
  - e.g.: +, ++, +=, -, -=, *, /, %, >>, <<, &&, ||, …
- C doesn't have operators relating to objects:

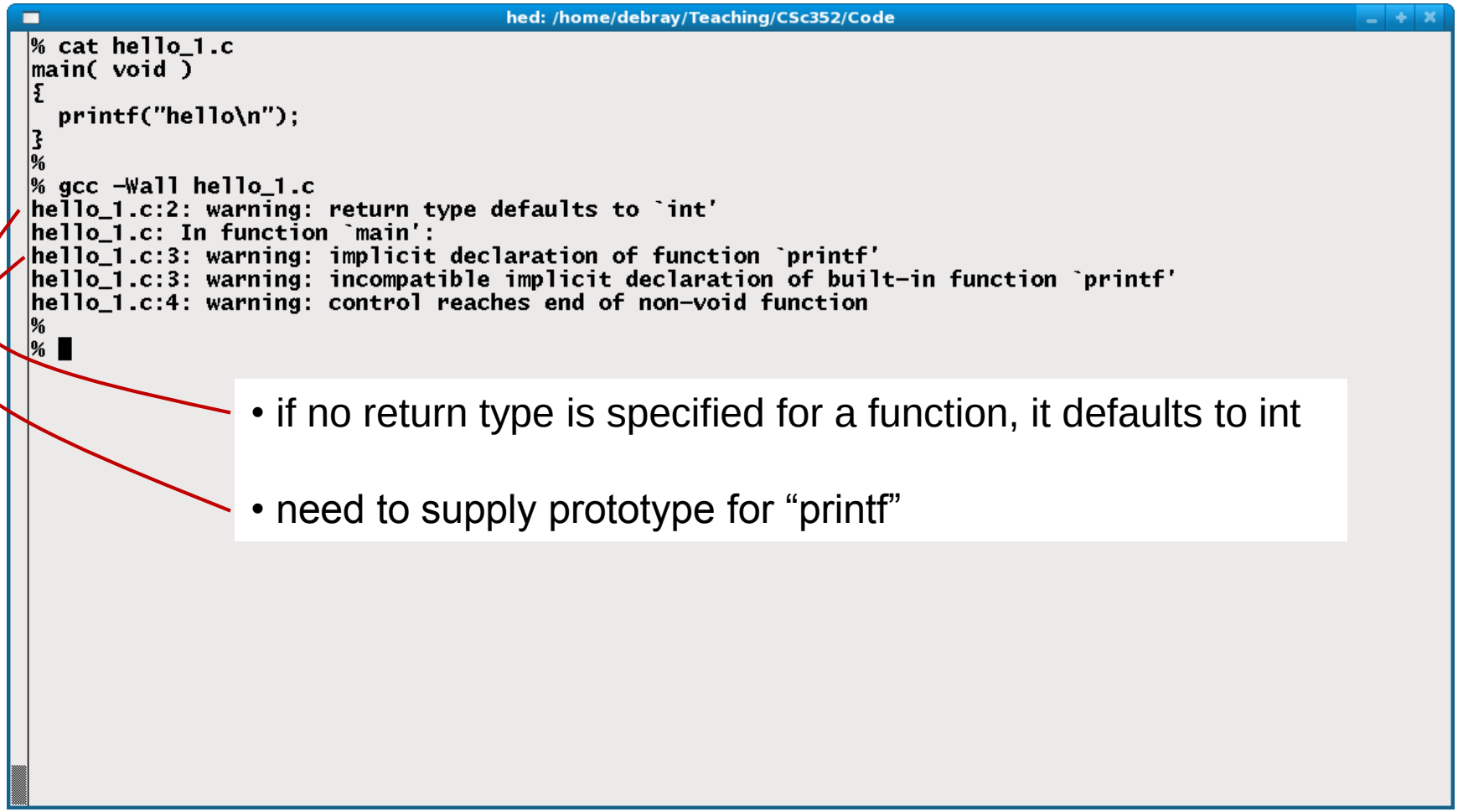
  new, instanceof
- C doesn't have >>> (and >>>=)
  - use >> on unsigned type instead

prototypes – recall:

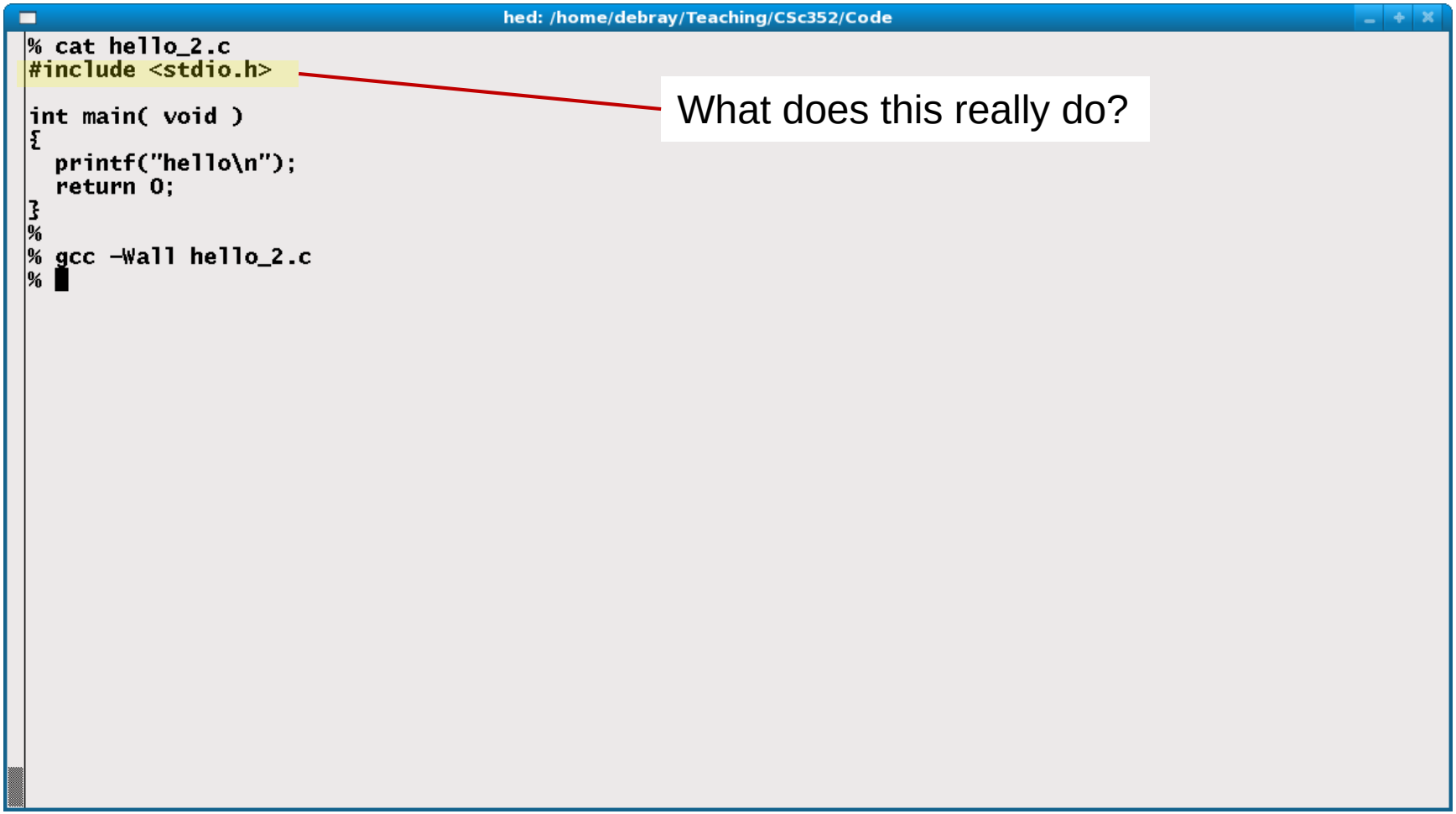


warnings when compiled

gcc –Wall generates warnings on questionable constructs



Fixing the compiler warnings



prototypes

```
#include <stdio.h>

int main()
{
  int a, b, c;

  scanf("%d%d", &a, &b);
  c = addInt(a, b);
  printf("%d + %d = %d\n",a,b,c);

  return 0;
}

int addInt(int a, int b)
{
  return a + b;
}
```

What happens when we try to compile this simple program?

prototypes (cont)

- When we try to compile the program this happens:

```
eanson@lectura:~/inClass$ gcc -Wall adder.c
adder.c: In function 'main':
adder.c:8:3: warning: implicit declaration of function 'addInt' [-Wimplicit-func
tion-declaration]
```

This looks a lot like the warning we got when we didn't have the #include <stdio.h>

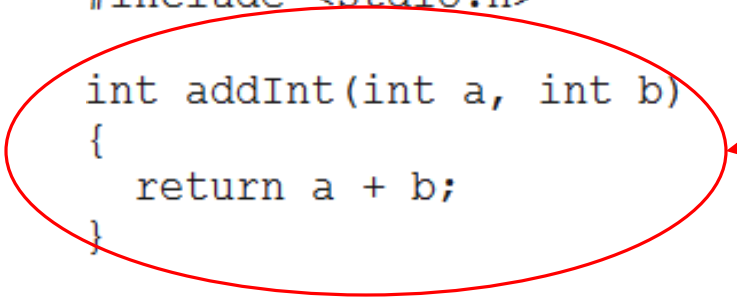
prototypes

```
#include <stdio.h>

int main()
{
  int a, b, c;

  scanf("%d%d", &a, &b);
  c = addInt(a, b);
  printf("%d + %d = %d\n",a,b,c);

  return 0;
}

int addInt(int a, int b)
{
  return a + b;
}
```

When the compiler sees this, it wants to know what addInt is.

Fix One

```
eanson@lectura:~/inClass$ cat fixAdder.c
#include <stdio.h>

int addInt(int a, int b)
{
  return a + b;
}

int main()
{
  int a, b, c;

  scanf("%d%d", &a, &b);
  c = addInt(a, b);
  printf("%d + %d = %d\n",a,b,c);

  return 0;
}


eanson@lectura:~/inClass$ gcc -Wall fixAdder.c
eanson@lectura:~/inClass$
```



Defined before it is used.

Fix Two

```
eanson@lectura:~/inClass$ cat proto.c
#include <stdio.h>

int addInt(int, int);

int main()
{
  int a, b, c;

  scanf("%d%d", &a, &b);
  c = addInt(a, b);
  printf("%d + %d = %d\n",a,b,c);

  return 0;
}

int addInt(int a, int b)
{
  return a + b;
}

eanson@lectura:~/inClass$ gcc -Wall proto.c
eanson@lectura:~/inClass$
```

prototype added

Functions from special libraries

- Some library code is not linked in by default
  - Examples: **sqrt**, **ceil**, **sin**, **cos**, **tan**, **log**, … [math library]
  - requires specifying to the compiler/linker that the math library needs to be linked in
    - you do this by adding "**–lm**" at the end of the compiler invocation:

gcc –Wall foo.c –lm

linker command to add math library

- Libraries that need to be linked in explicitly like this are indicated in the man pages

void

- I’ve said in C there are no procedures just functions.
- Every function returns a value, but what if I don’t need/want a value returned?
- For example, suppose I have a function that just prints “hi”.
- I can declare that function as type “void”

void

```
eanson@lectura:~/inClass$ cat voider.c
#include <stdio.h>

void sayHi()
{
  printf("HI!\n");

  return;
}

int main()
{
  sayHi();

  return 0;
}
eanson@lectura:~/inClass$ gcc -Wall voider.c
eanson@lectura:~/inClass$ a.out
HI!
eanson@lectura:~/inClass$
```

void

```
eanson@lectura:~/inClass$ cat voider.c
#include <stdio.h>

void sayHi()
{
  printf("HI!\n");

  return;
}

int main()
{
  sayHi();

  return 0;
}
eanson@lectura:~/inClass$ gcc -Wall voider.c
eanson@lectura:~/inClass$ a.out
HI!
eanson@lectura:~/inClass$
```

Function returns no value

void

```
eanson@lectura:~/inClass$ cat voider.c
#include <stdio.h>

void sayHi()
{
  printf("HI!\n");

  return;
}

int main()
{
  sayHi();

  return 0;
}
eanson@lectura:~/inClass$ gcc -Wall voider.c
eanson@lectura:~/inClass$ a.out
HI!
eanson@lectura:~/inClass$
```

Function also has no parameters.

void

```
eanson@lectura:~/inClass$ cat voider.c
#include <stdio.h>

void sayHi(void)
{
  printf("HI!\n");

  return;
}

int main()
{
  sayHi();

  return 0;
}
eanson@lectura:~/inClass$ gcc -Wall voider.c
eanson@lectura:~/inClass$ a.out
HI!
eanson@lectura:~/inClass$
```

Optional way to indicate that the function also has no parameters.

printing to stderr

- The assignments ask you to print error messages to standard error.
- To do this you use fprintf
- fprintf works like printf except that printf always prints to stdout and you tell fprintf where to print:
- To print an error:
  fprintf(stderr,"I'm an error\n");

printing to stderr

- Note
  printf("Hi!\n");
- and
  fprintf(stdout,"Hi\n");
- Do exactly the same thing.

Returning from an error

- Remember that in the assignments, when your program exits, if it has seen an error it should not return 0.
- The catch is sometimes your program will continue to run after getting a bad input. It must remember it has seen the error and return the correct value.

Characters and strings

- A character is of type **char** (signed or unsigned)
  - C99 provides support for “wide characters”
- **strings** :
  - an array of characters 
  - terminated by a 0 character (“NUL” : written ‘\0’)
  - Not a predefined type in C; string operations are provided through libraries

 C arrays don’t do bound-checking

  - careless programming can give rise to memory-corruption errors

Example of string manipulation

hed: /cs/www/classes/cs352/spring10/Code/ex.0.Elementary_C

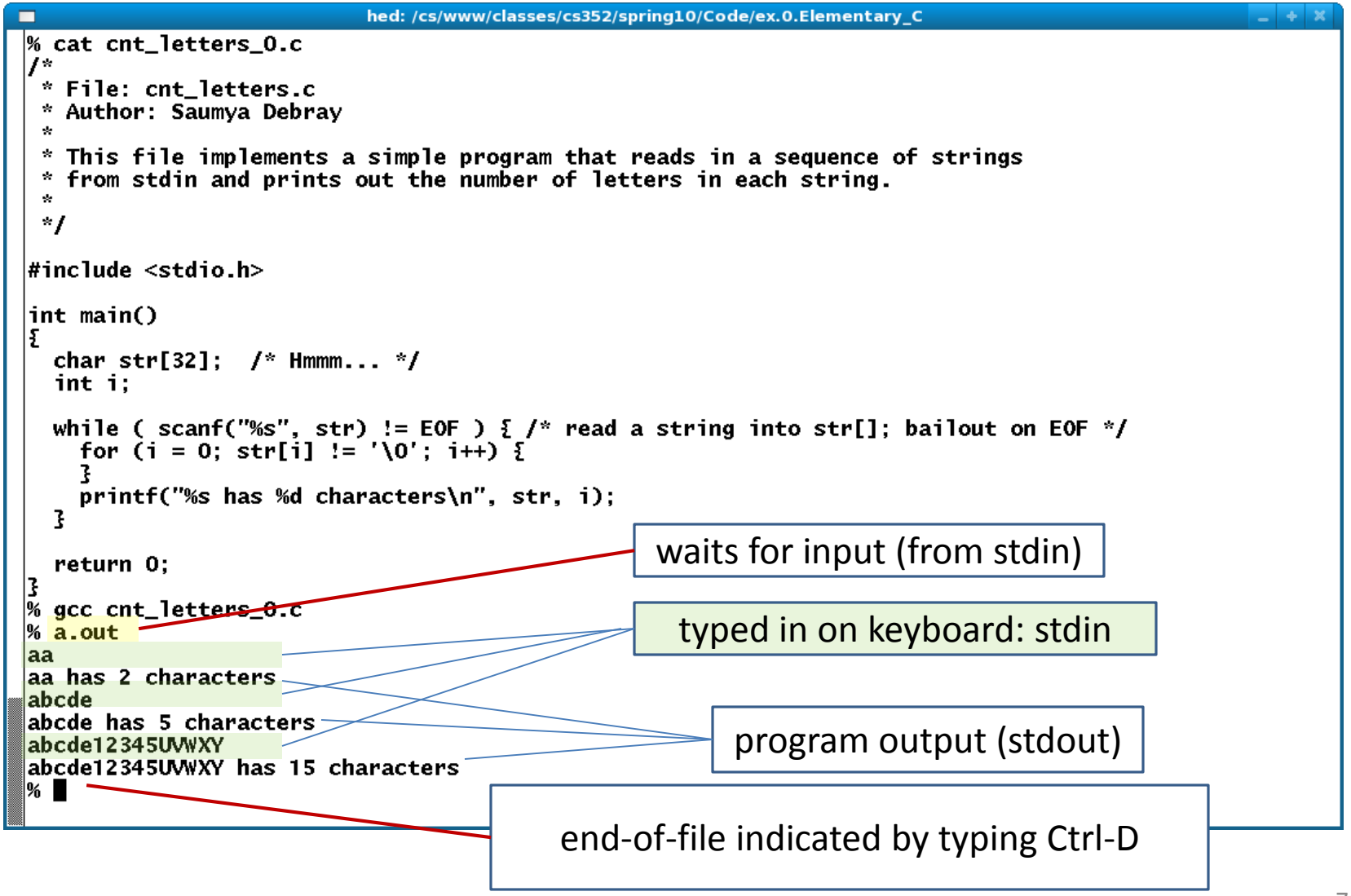
```
% cat cnt_letters_0.c
/*
 * File: cnt_letters.c
 * Author: Saumya Debray
 *
 * This file implements a simple program that reads in a sequence of strings
 * from stdin and prints out the number of letters in each string.
 *
 */

#include <stdio.h>

int main()
{
  char str[32]; /* Hmmm... */
  int i;

  while ( scanf("%s", str) != EOF ) { /* read a string into str[]; bailout on EOF */
    for (i = 0; str[i] != '\0'; i++) {
    }
    printf("%s has %d characters\n", str, i);
  }

  return 0;
}
% gcc cnt_letters_0.c
% a.out
aa
aa has 2 characters
abcde
abcde has 5 characters
abcde12345UVWXY
abcde12345UVWXY has 15 characters
%
```

declares an array of chars

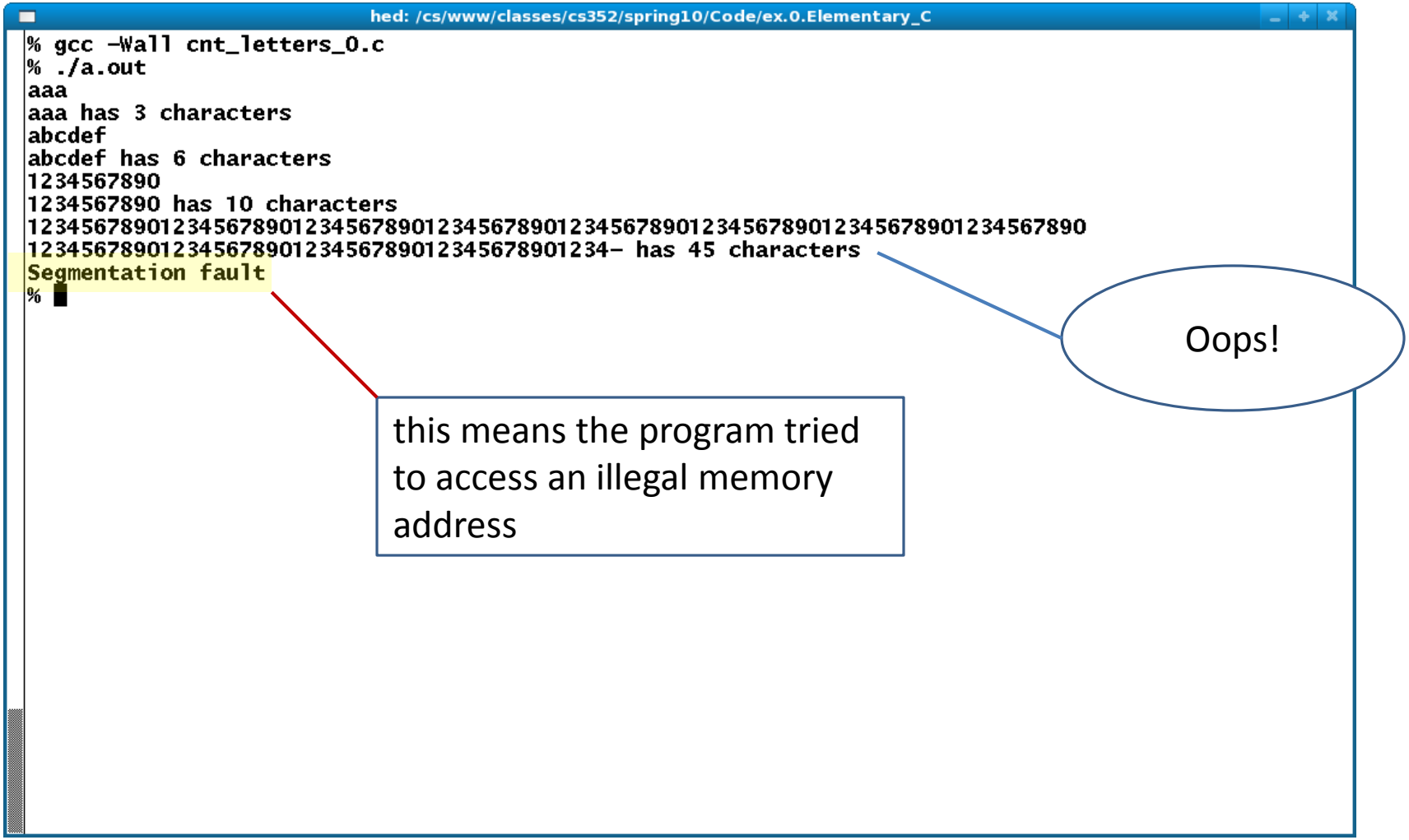
the input analog to printf:
"read a value into **str**"
(%s = "string")

print out a string (%s)

More strings…



More strings…



More detail

- So “strings” are arrays of characters (that end with a NULL (‘\0’))
- But what is a character?
- A character is a byte of data.
- This can be viewed as a (small) number or as a symbol from the ASCII table.
- To specify a character in C use single quotes:
  char c;
  c = ‘b’;

- So even though characters can be thought of as letters, digits, or symbols, you can also use addition, subtraction, multiplication, etc on them.
- You can also compare them:
  ‘A’ < ‘G’ evaluates to true.
- The uppercase letters and lower case letters run in a sequence.
- i.e.: ‘c’ + 1 == ‘d’

- How can we use this to write a test if a character c is a capitol letter?

- How can we use this to write a test if a character c is a upper case letter?

```
if (c >= ‘A’ && c <= ‘Z’) {
// if c is an uppercase letter then we go here
}
```

- Let’s write a function to convert all the lowercase letters of a string to upper case.

The completed program

```
eanson@lectura:~/inClass$ cat upperExample.c
#include <stdio.h>

/* void upper(char c[]) takes a string and converts all lower case letters
 * to uppercase letters.
*/
void upper(char c[]) {
  int i;
  for (i=0; c[i] != '\0'; ++i)
    if (c[i] >= 'a' && c[i] <= 'z')     //c[i] is lower case letter
      c[i] = c[i] - 'a' + 'A';
  return;
}

int main() {
  char str[64];
  scanf("%s", str);
  upper(str);
  printf("%s\n",str);
  return 0;
}
eanson@lectura:~/inClass$ gcc -Wall upperExample.c
eanson@lectura:~/inClass$ a.out
What4Is!This?
WHAT4IS!THIS?
eanson@lectura:~/inClass$
```