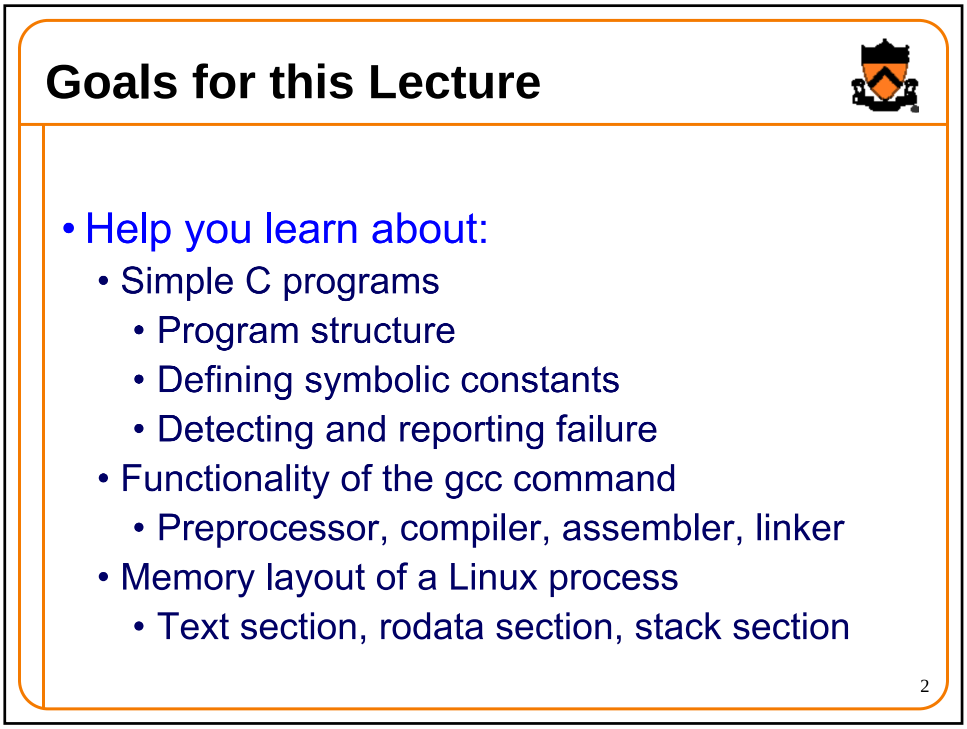
Simple C Programs



Goals for this Lecture

- Help you learn about:
  - Simple C programs
    - Program structure
    - Defining symbolic constants
    - Detecting and reporting failure
  - Functionality of the gcc command
    - Preprocessor, compiler, assembler, linker
  - Memory layout of a Linux process
    - Text section, rodata section, stack section

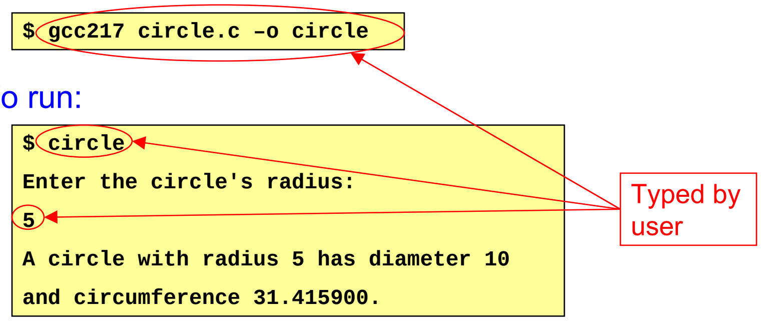
“Circle” Program

- File circle.c:

```
#include <stdio.h>

int main(void)

/* Read a circle's radius from stdin, and compute and write its
   diameter and circumference to stdout.  Return 0. */

{
   int radius;
   int diam;
   double circum;
   printf("Enter the circle's radius:\n");
   scanf("%d", &radius);
   diam = 2 * radius;
   circum = 3.14159 * (double)diam;
   printf("A circle with radius %d has diameter %d\n",
      radius, diam);
   printf("and circumference %f.\n", circum);
   return 0;
}
```

Building and Running

- To build (preprocess, compile, assemble, and link):



```
$ gcc217 circle.c –o circle
```

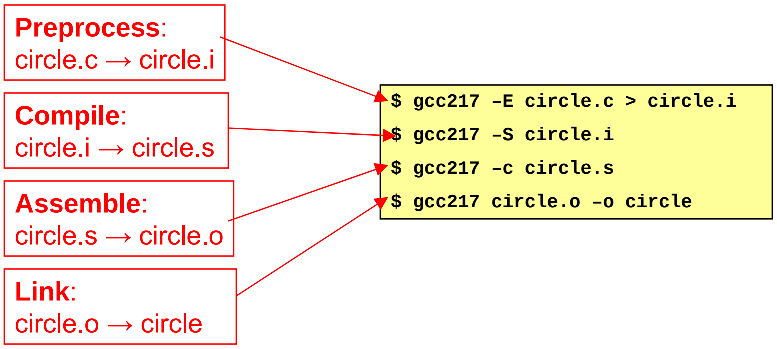
- To run:

```
$ circle

Enter the circle's radius:

5

A circle with radius 5 has diameter 10

and circumference 31.415900.
```

Typed by user

Steps in the Build Process

- To build one step at a time:



Preprocess:
circle.c → circle.i

Compile:
circle.i → circle.s

Assemble:
circle.s → circle.o

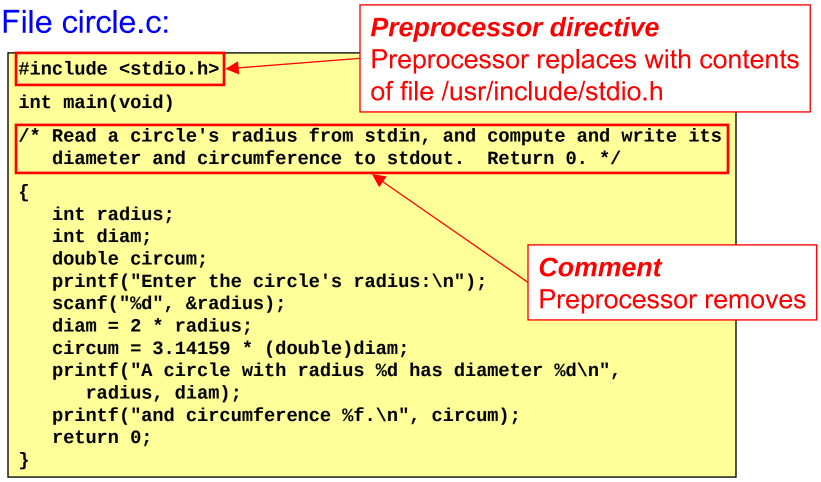
Link:
circle.o → circle

```
$ gcc217 –E circle.c > circle.i
$ gcc217 –S circle.i
$ gcc217 –c circle.s
$ gcc217 circle.o –o circle
```

- Why build one step at a time?
  - Helpful for learning how to interpret error messages
  - Permits partial builds (described later in course)

The Preprocessor’s View

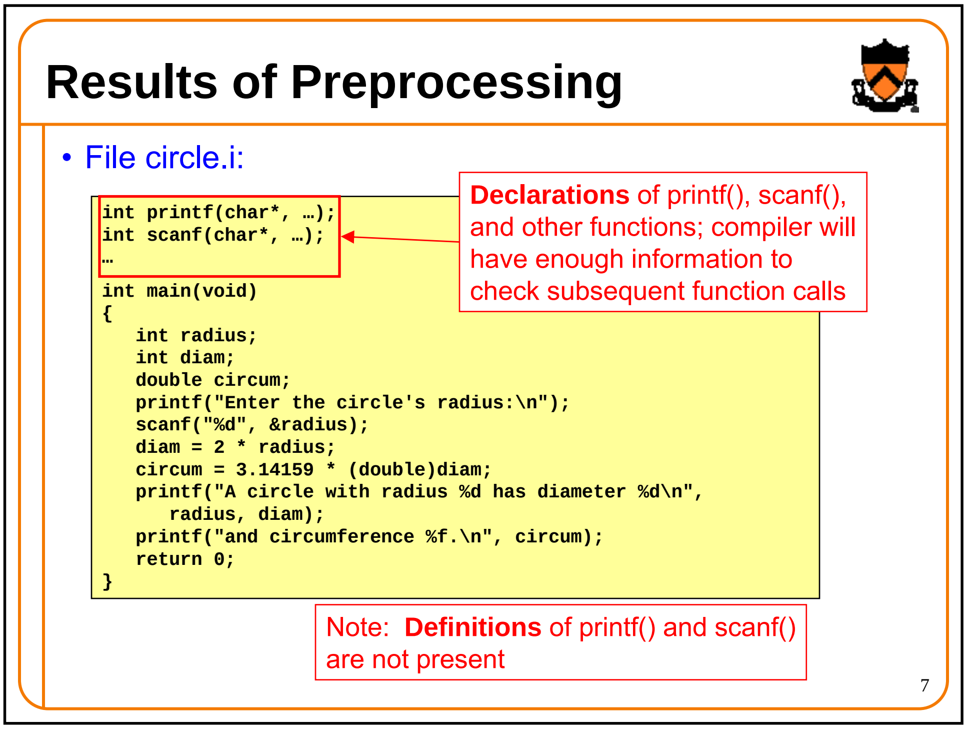
- File circle.c:



```
#include <stdio.h>
int main(void)
/* Read a circle's radius from stdin, and compute and write its
   diameter and circumference to stdout.  Return 0. */
{
   int radius;
   int diam;
   double circum;
   printf("Enter the circle's radius:\n");
   scanf("%d", &radius);
   diam = 2 * radius;
   circum = 3.14159 * (double)diam;
   printf("A circle with radius %d has diameter %d\n",
      radius, diam);
   printf("and circumference %f.\n", circum);
   return 0;
}
```

Preprocessor directive
Preprocessor replaces with contents of file /usr/include/stdio.h

Comment
Preprocessor removes



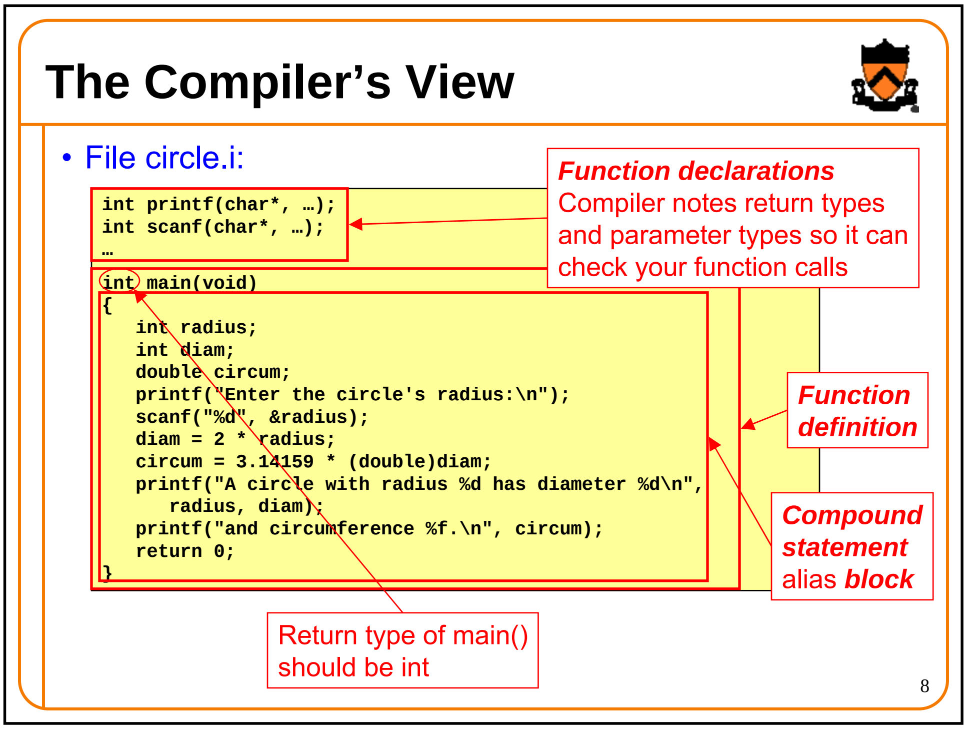
Results of Preprocessing

- File circle.i:

```
int printf(char*, …);
int scanf(char*, …);
…
int main(void)
{
   int radius;
   int diam;
   double circum;
   printf("Enter the circle's radius:\n");
   scanf("%d", &radius);
   diam = 2 * radius;
   circum = 3.14159 * (double)diam;
   printf("A circle with radius %d has diameter %d\n",
      radius, diam);
   printf("and circumference %f.\n", circum);
   return 0;
}
```

Declarations of printf(), scanf(), and other functions; compiler will have enough information to check subsequent function calls

Note: **Definitions** of printf() and scanf() are not present



The Compiler's View

- File circle.i:

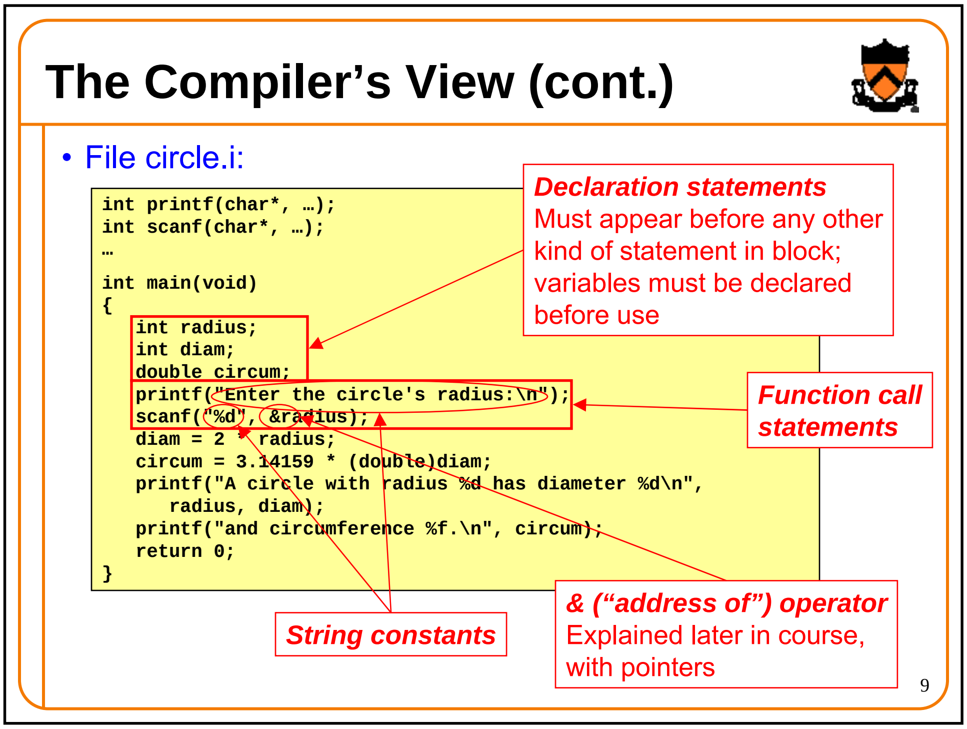
```
int printf(char*, …);
int scanf(char*, …);
…
int main(void)
{
   int radius;
   int diam;
   double circum;
   printf("Enter the circle's radius:\n");
   scanf("%d", &radius);
   diam = 2 * radius;
   circum = 3.14159 * (double)diam;
   printf("A circle with radius %d has diameter %d\n",
      radius, diam);
   printf("and circumference %f.\n", circum);
   return 0;
}
```

Function declarations Compiler notes return types and parameter types so it can check your function calls

Function definition

Compound statement alias ***block***

Return type of main() should be int



The Compiler’s View (cont.)

- File circle.i:

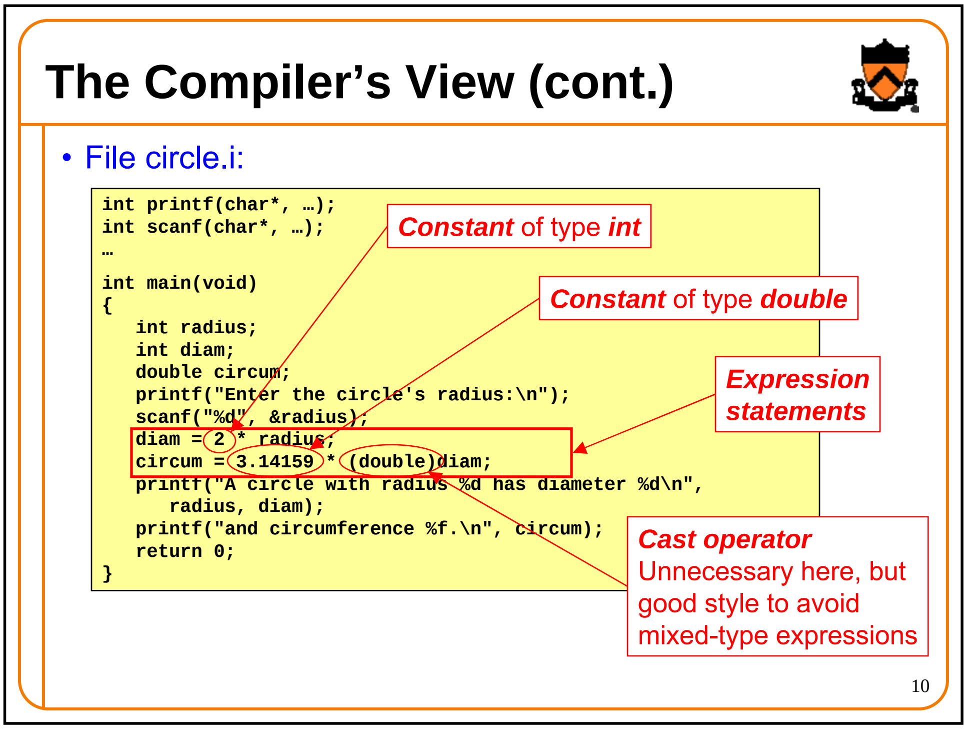
```
int printf(char*, …);
int scanf(char*, …);
…
int main(void)
{
   int radius;
   int diam;
   double circum;
   printf("Enter the circle's radius:\n");
   scanf("%d", &radius);
   diam = 2 * radius;
   circum = 3.14159 * (double)diam;
   printf("A circle with radius %d has diameter %d\n",
      radius, diam);
   printf("and circumference %f.\n", circum);
   return 0;
}
```

Declaration statements
Must appear before any other kind of statement in block; variables must be declared before use

Function call statements

String constants

& (“address of”) operator
Explained later in course, with pointers



The Compiler’s View (cont.)

- File circle.i:

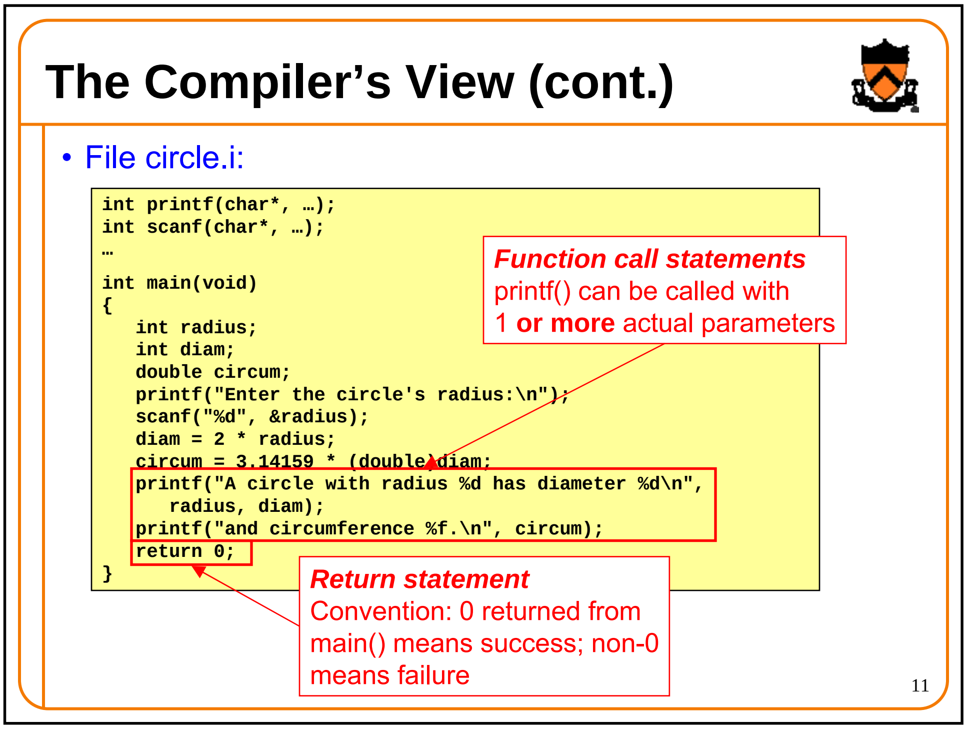
```
int printf(char*, …);
int scanf(char*, …);
…
int main(void)
{
   int radius;
   int diam;
   double circum;
   printf("Enter the circle's radius:\n");
   scanf("%d", &radius);
   diam = 2 * radius;
   circum = 3.14159 * (double)diam;
   printf("A circle with radius %d has diameter %d\n",
      radius, diam);
   printf("and circumference %f.\n", circum);
   return 0;
}
```

Constant of type ***int***

Constant of type ***double***

Expression statements

Cast operator
Unnecessary here, but good style to avoid mixed-type expressions



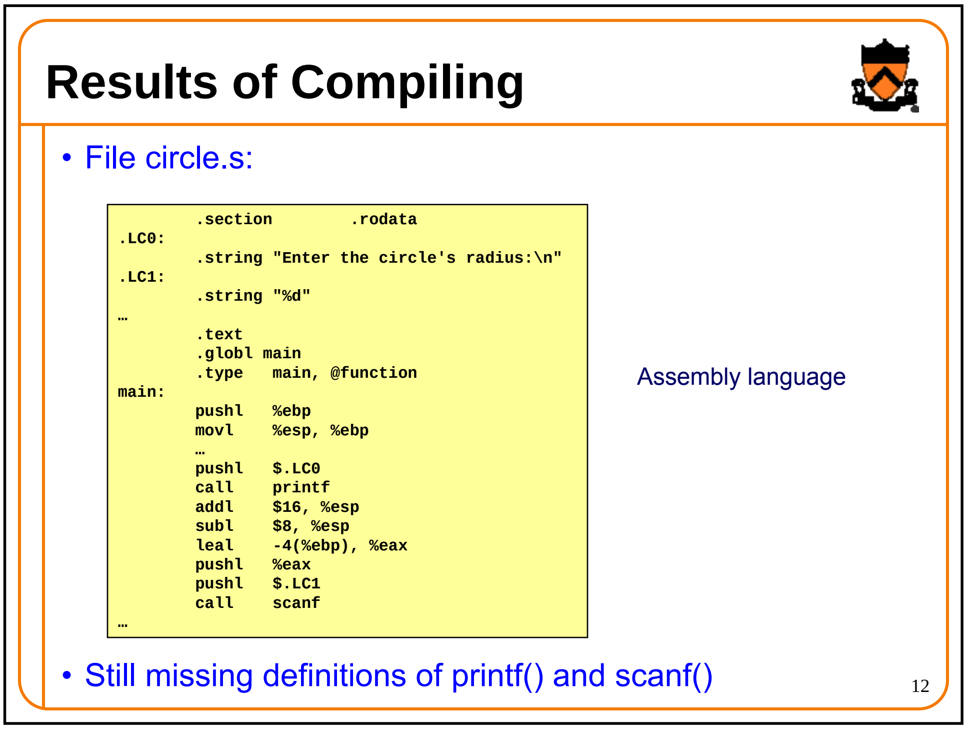
The Compiler’s View (cont.)

- File circle.i:

```
int printf(char*, …);
int scanf(char*, …);
…
int main(void)
{
   int radius;
   int diam;
   double circum;
   printf("Enter the circle's radius:\n");
   scanf("%d", &radius);
   diam = 2 * radius;
   circum = 3.14159 * (double)diam;
   printf("A circle with radius %d has diameter %d\n",
      radius, diam);
   printf("and circumference %f.\n", circum);
   return 0;
}
```

Function call statements
printf() can be called with
1 **or more** actual parameters

Return statement
Convention: 0 returned from main() means success; non-0 means failure



Results of Compiling

- File circle.s:

```
        .section        .rodata
.LC0:
        .string "Enter the circle's radius:\n"
.LC1:
        .string "%d"
…
        .text
        .globl main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        …
        pushl   $.LC0
        call    printf
        addl    $16, %esp
        subl    $8, %esp
        leal    -4(%ebp), %eax
        pushl   %eax
        pushl   $.LC1
        call    scanf
…
```

Assembly language

- Still missing definitions of printf() and scanf()

The Assembler's View

- File circle.s:

```
        .section        .rodata
.LC0:
        .string "Enter the circle's radius:\n"
.LC1:
        .string "%d"
…
        .text
        .globl main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        …
        pushl   $.LC0
        call    printf
        addl    $16, %esp
        subl    $8, %esp
        leal    -4(%ebp), %eax
        pushl   %eax
        pushl   $.LC1
        call    scanf
…
```

Assembly language

- Assembler translates assembly language into machine language
  - Details provided in 2nd half of course

Results of Assembling

- File circle.o:

```
Listing omitted
Not human-readable
```

Machine language

- Object file
- Still missing definitions of printf() and scanf()

The Linker’s View

- File circle.o:

```
Listing omitted
Not human-readable
```

Machine language

- The linker:
  - Observes that
    - Code in circle.o **calls** printf() and scanf()
    - Code in circle.o **does not define** printf() or scanf()
  - Fetches machine language definitions of printf() and scanf() from standard C library (/usr/lib/libc.a on hats)
  - Merges those definitions with circle.o to create…

Results of Linking

- File circle:

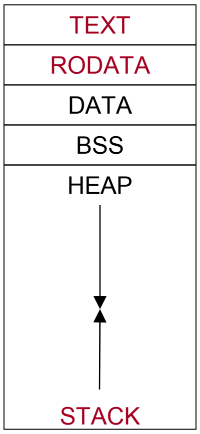
```
Listing omitted
Not human-readable
```

Machine language

- Complete executable binary file

Run-Time View

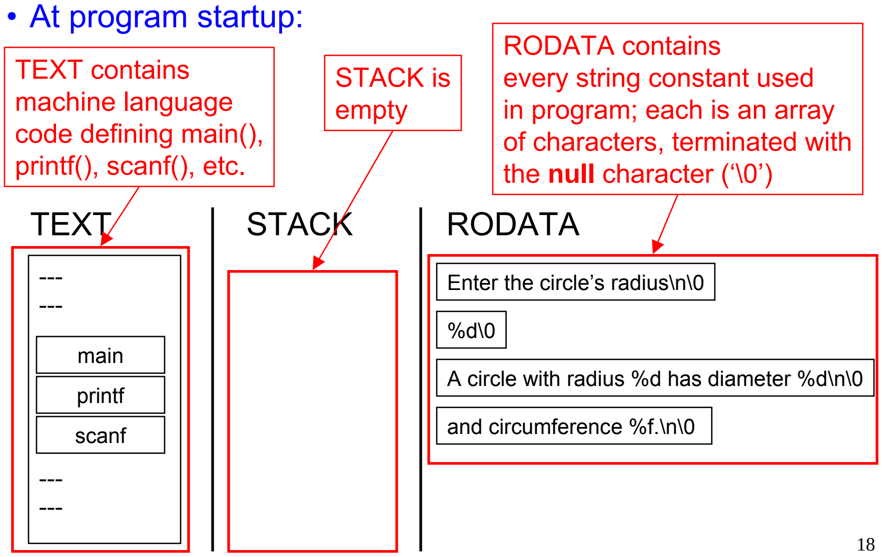
- At run-time, memory devoted to program is divided into **sections**:



- TEXT (read-only)
  - Stores executable machine language instructions
- RODATA (read-only)
  - Stores read-only data, esp. string constants
- STACK (read/write)
  - Stores values of local variables
- Other sections described later in course

Run-Time View: Startup

- At program startup:


TEXT contains machine language code defining main(), printf(), scanf(), etc.

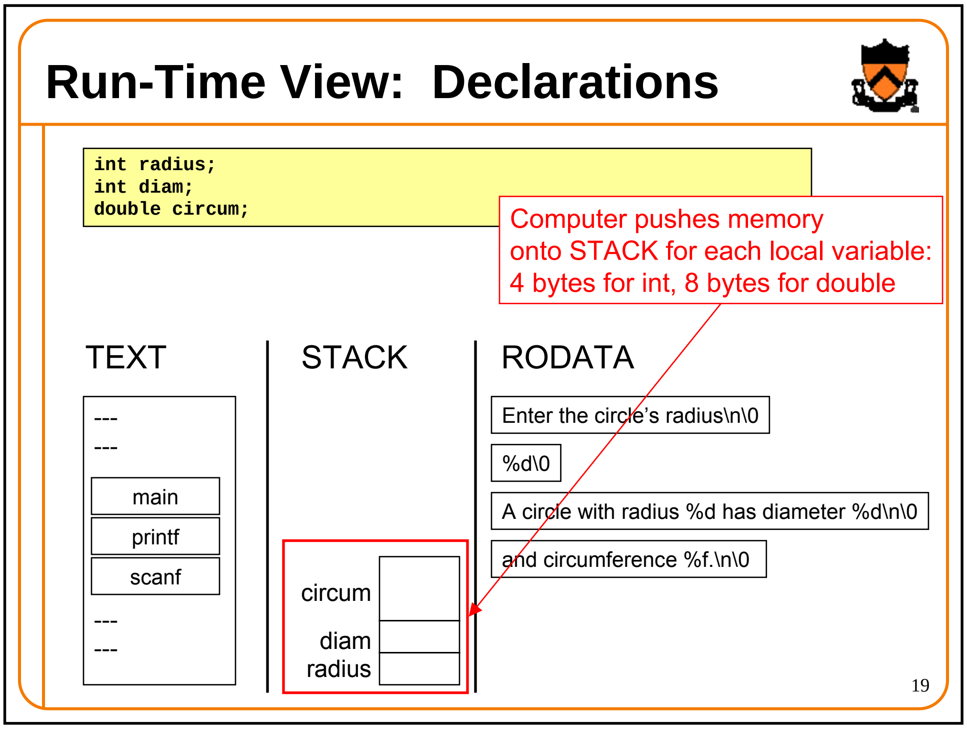
STACK is empty

RODATA contains every string constant used in program; each is an array of characters, terminated with the **null** character ('\0')

TEXT

---
main
printf
scanf

STACK
RODATA
Enter the circle's radius\n\0
%d\0
A circle with radius %d has diameter %d\n\0
and circumference %f.\n\0



Run-Time View: Declarations

```
int radius;
int diam;
double circum;
```

Computer pushes memory onto STACK for each local variable: 4 bytes for int, 8 bytes for double

TEXT

---
main
printf
scanf

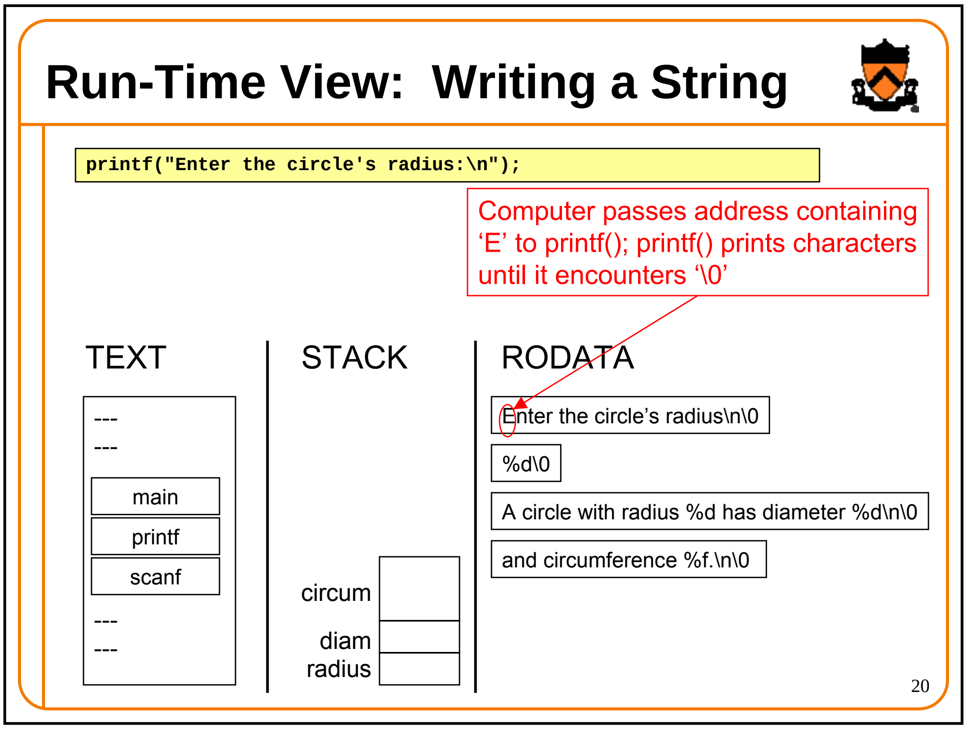
STACK

circum
diam
radius

RODATA

Enter the circle’s radius\n\0
%d\0
A circle with radius %d has diameter %d\n\0
and circumference %f.\n\0

19



Run-Time View: Writing a String

```
printf("Enter the circle's radius:\n");
```

Computer passes address containing ‘E’ to printf(); printf() prints characters until it encounters ‘\0’

TEXT

---
main
printf
scanf

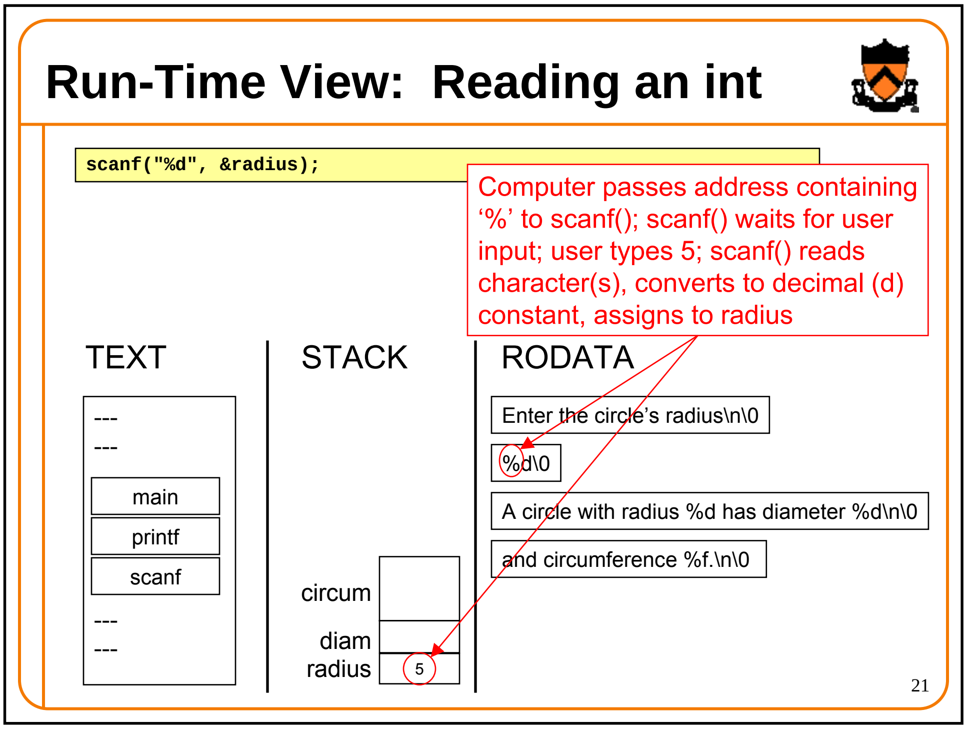
STACK

circum
diam
radius

RODATA

Enter the circle’s radius\n\0
%d\0
A circle with radius %d has diameter %d\n\0
and circumference %f.\n\0

20



Run-Time View: Reading an int

```
scanf("%d", &radius);
```

Computer passes address containing ‘%’ to scanf(); scanf() waits for user input; user types 5; scanf() reads character(s), converts to decimal (d) constant, assigns to radius

TEXT

---
main
printf
scanf

STACK

circum
diam
radius 5

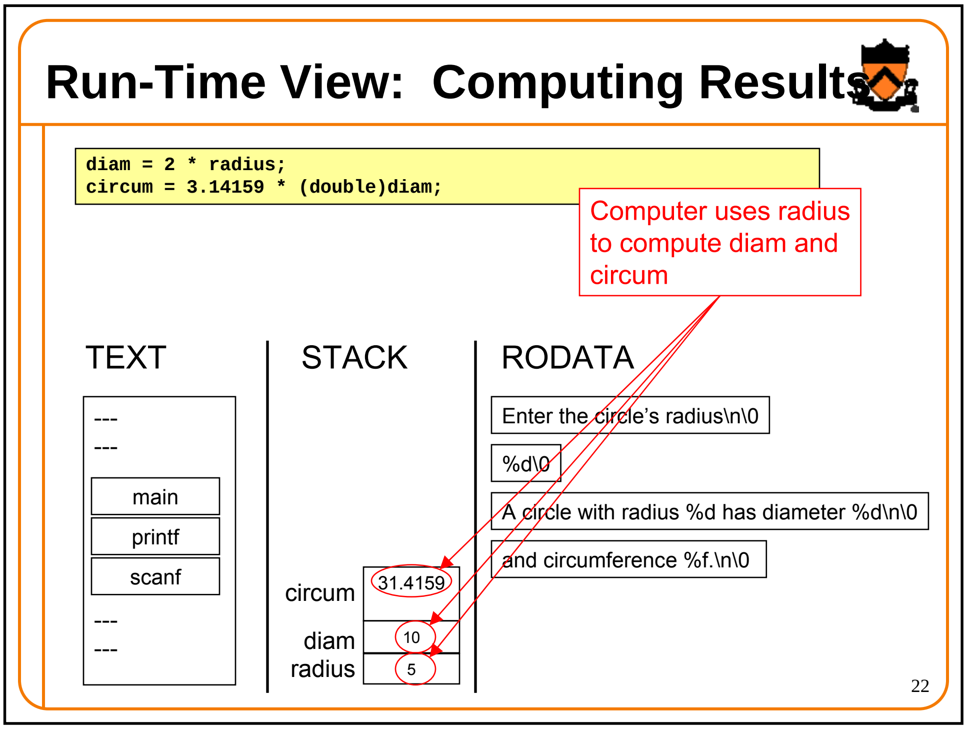
RODATA

Enter the circle’s radius\n\0

%d\0

A circle with radius %d has diameter %d\n\0

and circumference %f.\n\0



Run-Time View: Computing Results

```
diam = 2 * radius;
circum = 3.14159 * (double)diam;
```

Computer uses radius to compute diam and circum

TEXT

---
main
printf
scanf

STACK

circum 31.4159
diam 10
radius 5

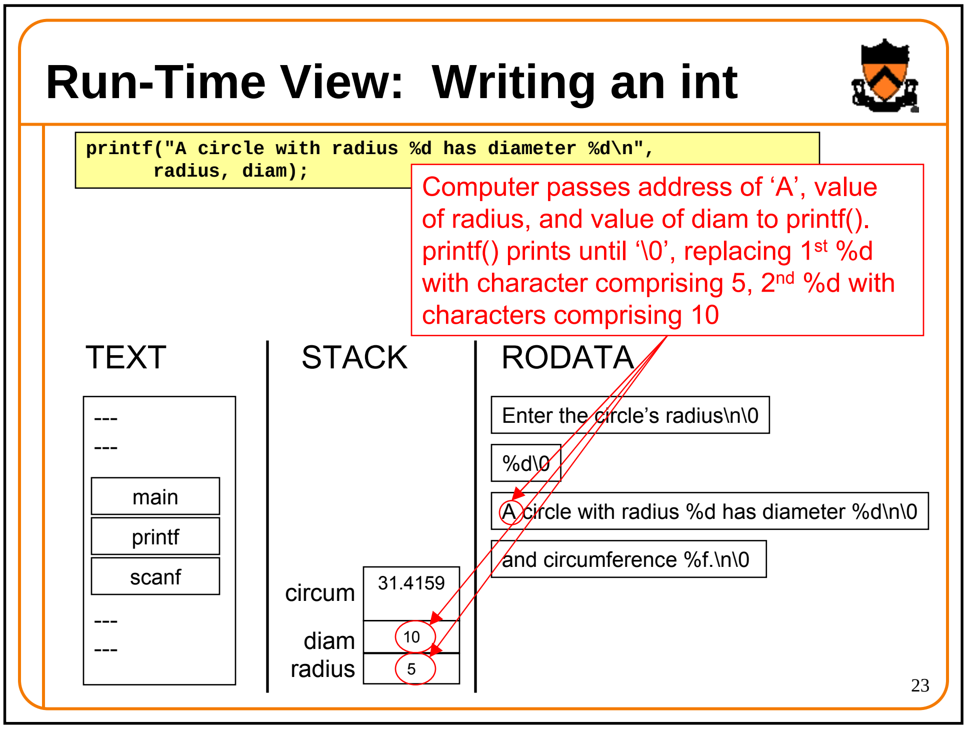
RODATA

Enter the circle’s radius\n\0

%d\0

A circle with radius %d has diameter %d\n\0

and circumference %f.\n\0



Run-Time View: Writing an int

```
printf("A circle with radius %d has diameter %d\n",
       radius, diam);
```

Computer passes address of ‘A’, value of radius, and value of diam to printf(). printf() prints until ‘\0’, replacing 1st %d with character comprising 5, 2nd %d with characters comprising 10

TEXT

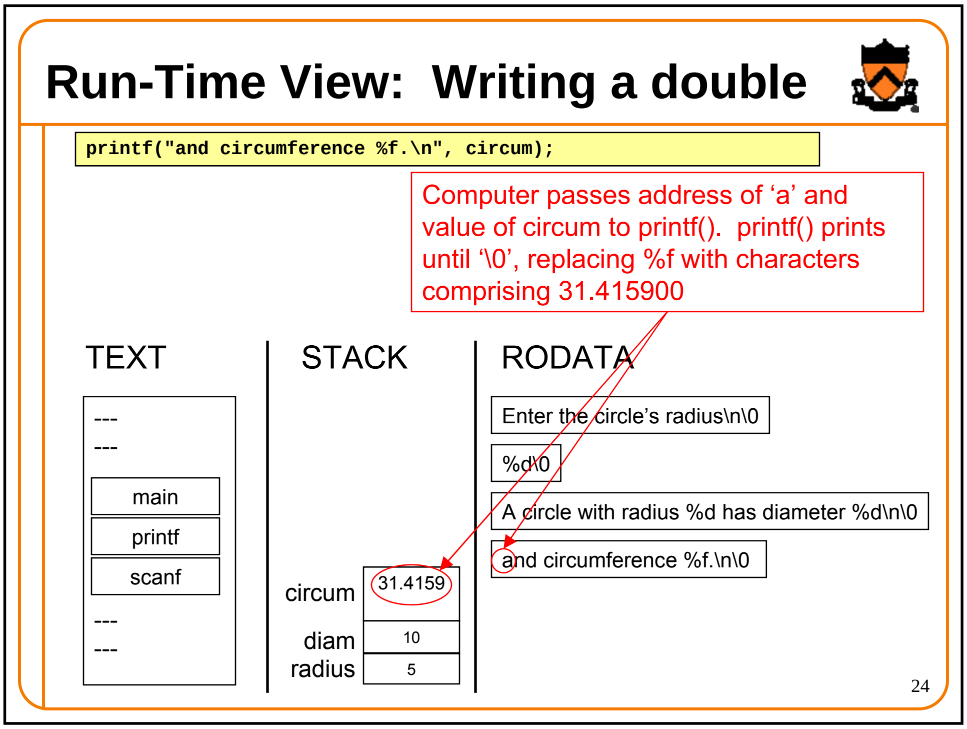
---
main
printf
scanf

STACK

circum 31.4159
diam 10
radius 5

RODATA

Enter the circle’s radius\n\0
%d\0
A circle with radius %d has diameter %d\n\0
and circumference %f.\n\0



Run-Time View: Writing a double

```
printf("and circumference %f.\n", circum);
```

Computer passes address of ‘a’ and value of circum to printf(). printf() prints until ‘\0’, replacing %f with characters comprising 31.415900

TEXT

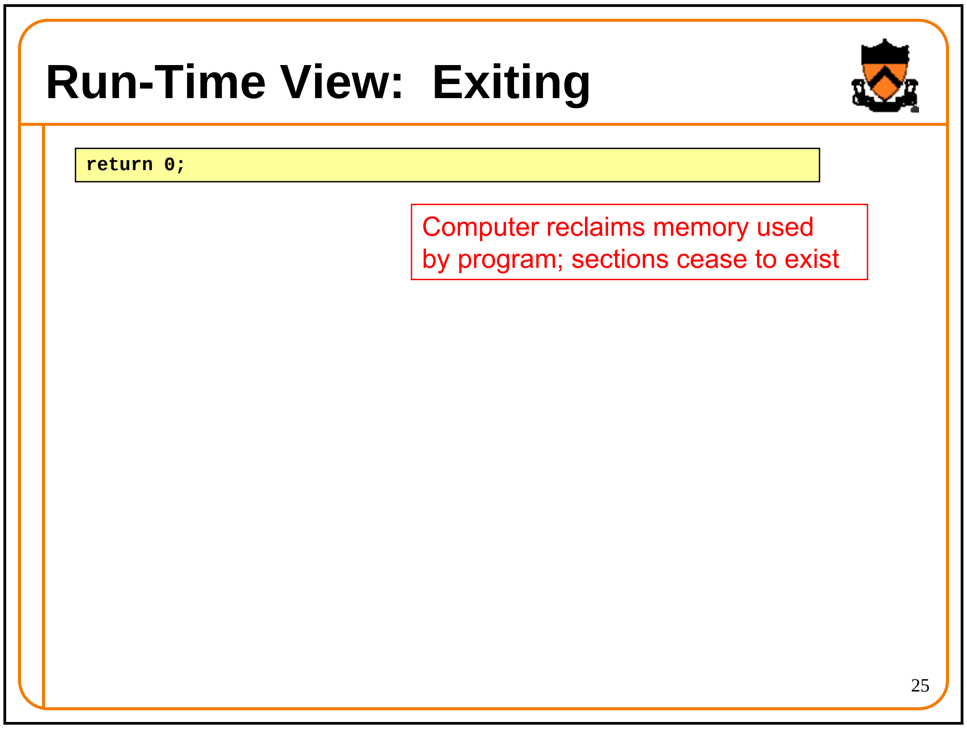
---
main
printf
scanf

STACK

circum 31.4159
diam 10
radius 5

RODATA

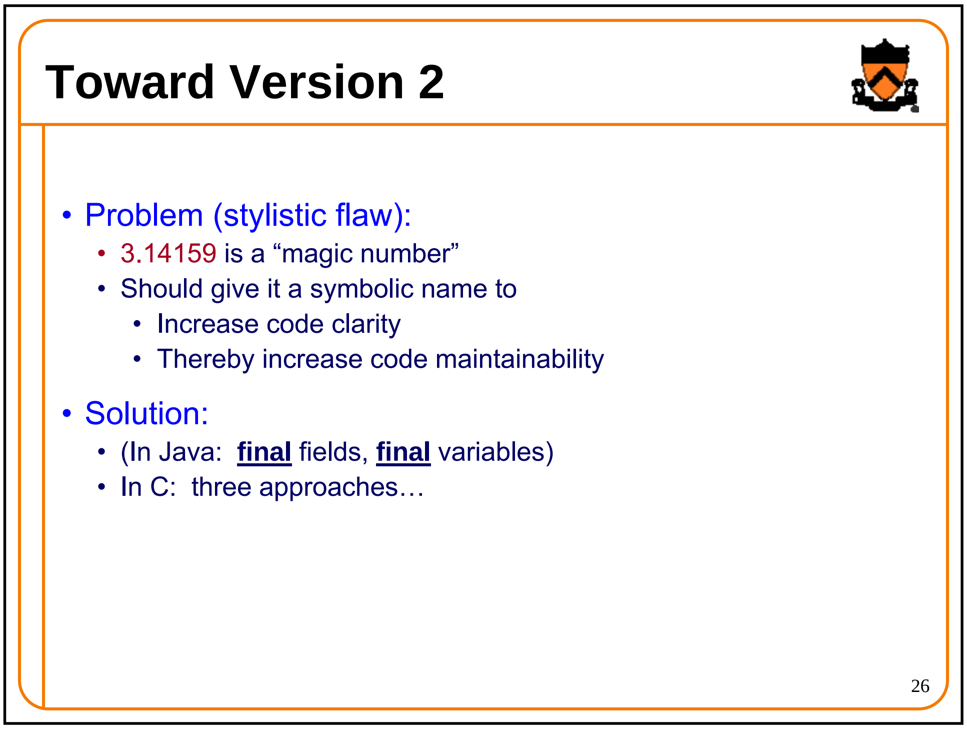
Enter the circle’s radius\n\0
%d\0
A circle with radius %d has diameter %d\n\0
and circumference %f.\n\0



Run-Time View: Exiting

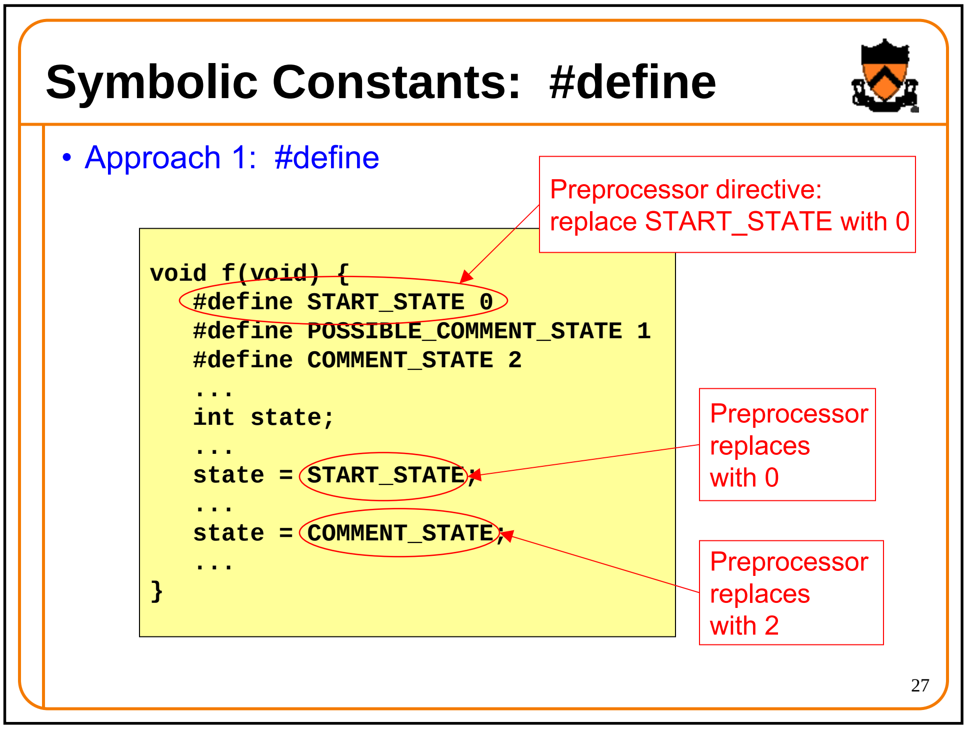
```
return 0;
```

Computer reclaims memory used by program; sections cease to exist



Toward Version 2

- Problem (stylistic flaw):
  - 3.14159 is a “magic number”
  - Should give it a symbolic name to
    - Increase code clarity
    - Thereby increase code maintainability
- Solution:
  - (In Java: **final** fields, **final** variables)
  - In C: three approaches…



Symbolic Constants: #define

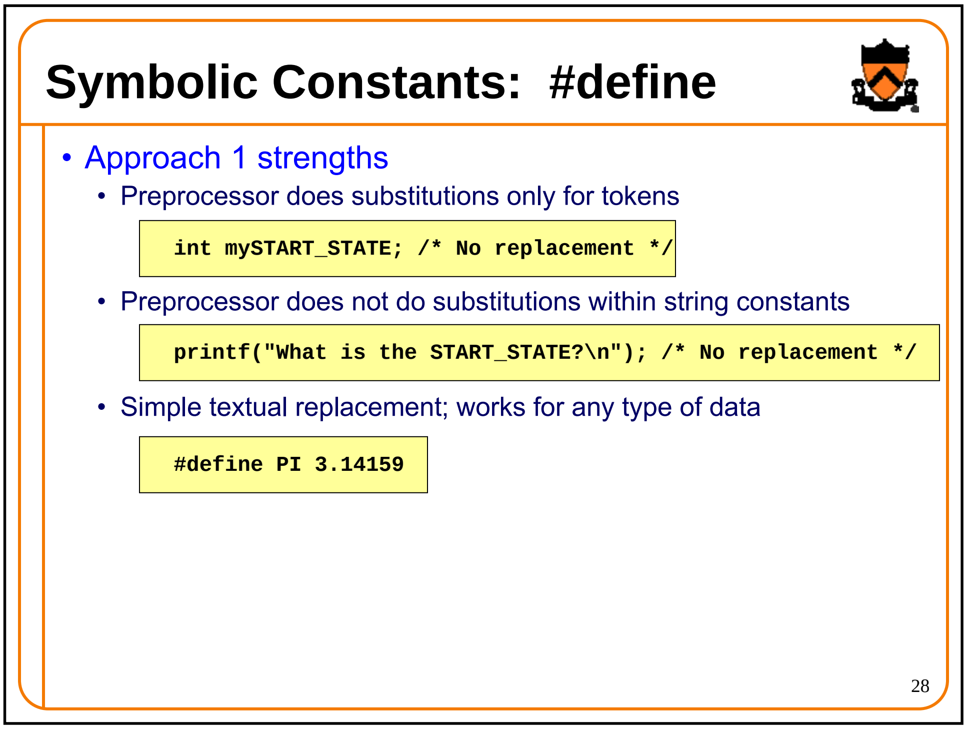
- Approach 1: #define

```
void f(void) {
   #define START_STATE 0
   #define POSSIBLE_COMMENT_STATE 1
   #define COMMENT_STATE 2
   ...
   int state;
   ...
   state = START_STATE;
   ...
   state = COMMENT_STATE;
   ...
}
```

Preprocessor directive: replace START_STATE with 0

Preprocessor replaces with 0

Preprocessor replaces with 2



Symbolic Constants: #define

- Approach 1 strengths
  - Preprocessor does substitutions only for tokens

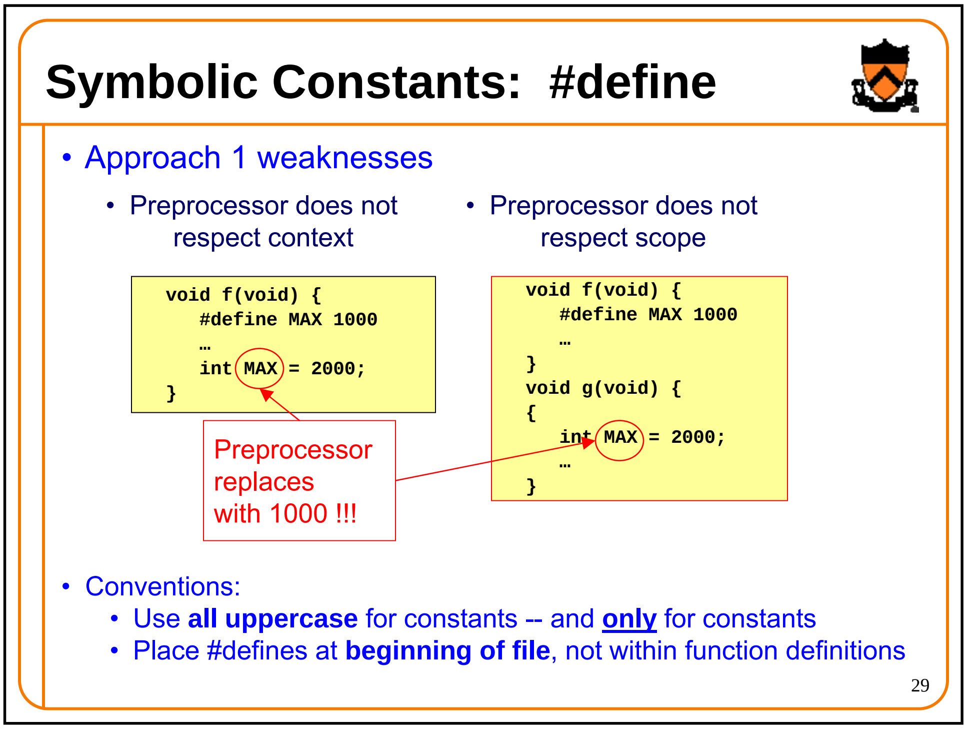
```
int mySTART_STATE; /* No replacement */
```

  - Preprocessor does not do substitutions within string constants

```
printf("What is the START_STATE?\n"); /* No replacement */
```

  - Simple textual replacement; works for any type of data

```
#define PI 3.14159
```



Symbolic Constants: #define

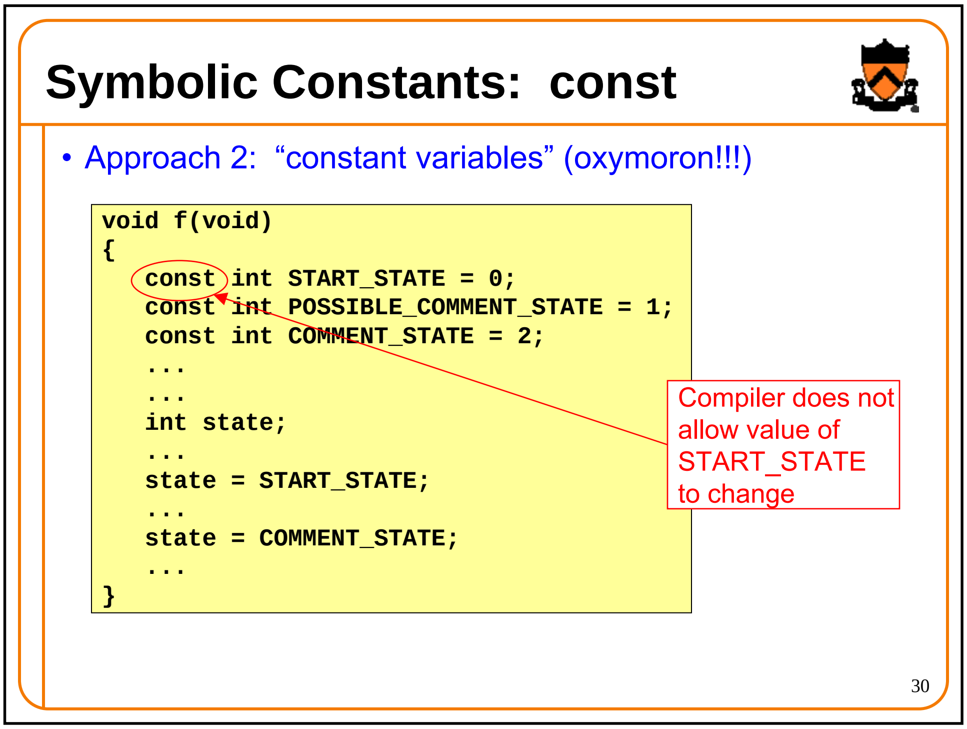
- Approach 1 weaknesses
  - Preprocessor does not respect context
  - Preprocessor does not respect scope

```
void f(void) {
   #define MAX 1000
   …
   int MAX = 2000;
}
```

```
void f(void) {
   #define MAX 1000
   …
}
void g(void) {
{
   int MAX = 2000;
   …
}
```

Preprocessor replaces with 1000 !!!

- Conventions:
  - Use **all uppercase** for constants -- and **only** for constants
  - Place #defines at **beginning of file**, not within function definitions

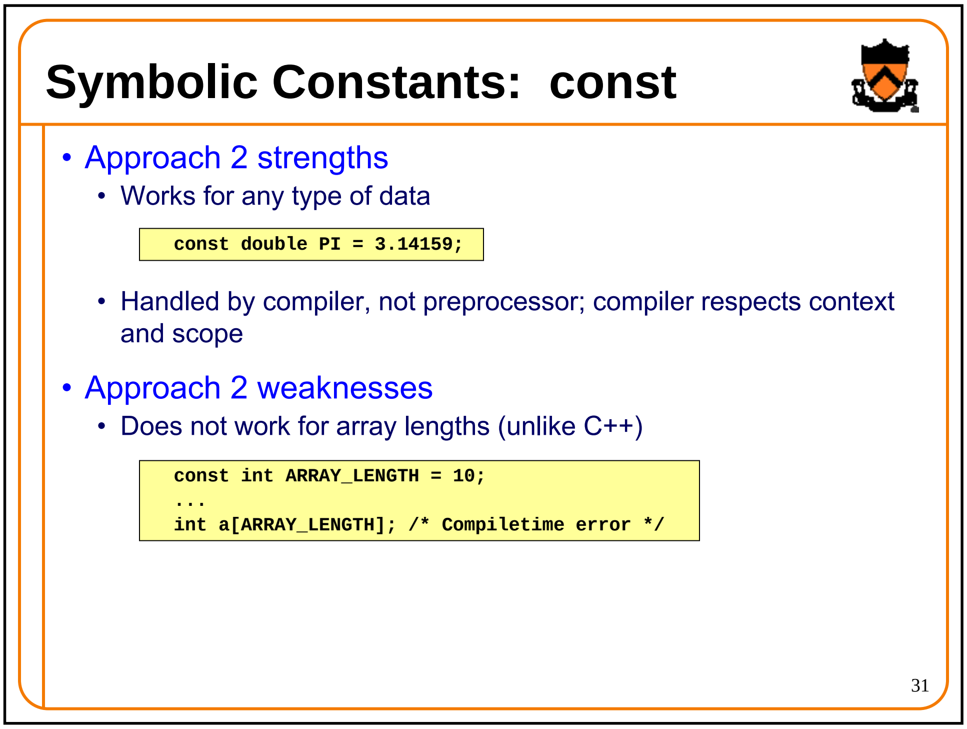


Symbolic Constants: const

- Approach 2: “constant variables” (oxymoron!!!)

```
void f(void)
{
   const int START_STATE = 0;
   const int POSSIBLE_COMMENT_STATE = 1;
   const int COMMENT_STATE = 2;
   ...
   ...
   int state;
   ...
   state = START_STATE;
   ...
   state = COMMENT_STATE;
   ...
}
```

Compiler does not allow value of START_STATE to change



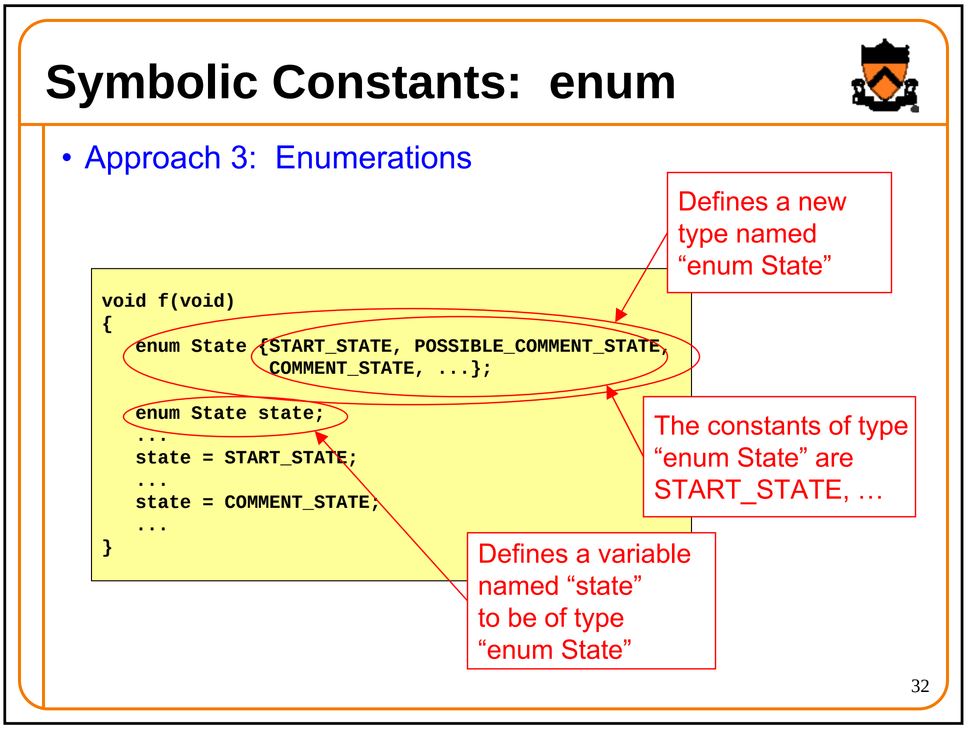
Symbolic Constants: const

- Approach 2 strengths
  - Works for any type of data

```
const double PI = 3.14159;
```

  - Handled by compiler, not preprocessor; compiler respects context and scope
- Approach 2 weaknesses
  - Does not work for array lengths (unlike C++)

```
const int ARRAY_LENGTH = 10;
...
int a[ARRAY_LENGTH]; /* Compiletime error */
```



Symbolic Constants: enum

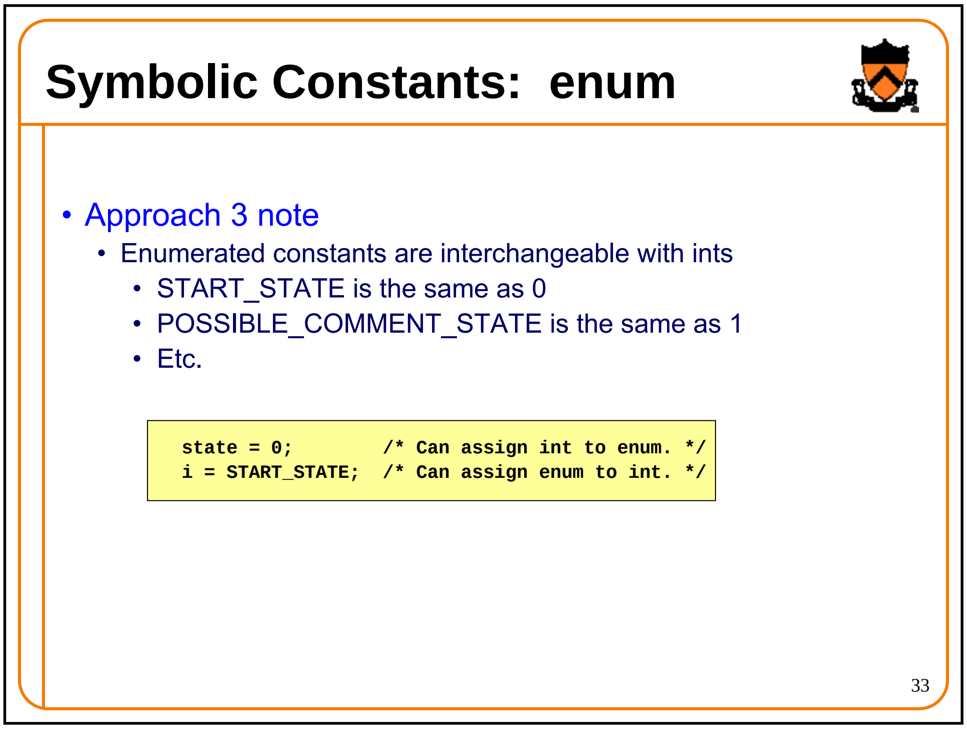
- Approach 3: Enumerations

```
void f(void)
{
   enum State {START_STATE, POSSIBLE_COMMENT_STATE,
               COMMENT_STATE, ...};
   enum State state;
   ...
   state = START_STATE;
   ...
   state = COMMENT_STATE;
   ...
}
```

Defines a new type named “enum State”

The constants of type “enum State” are START_STATE, …

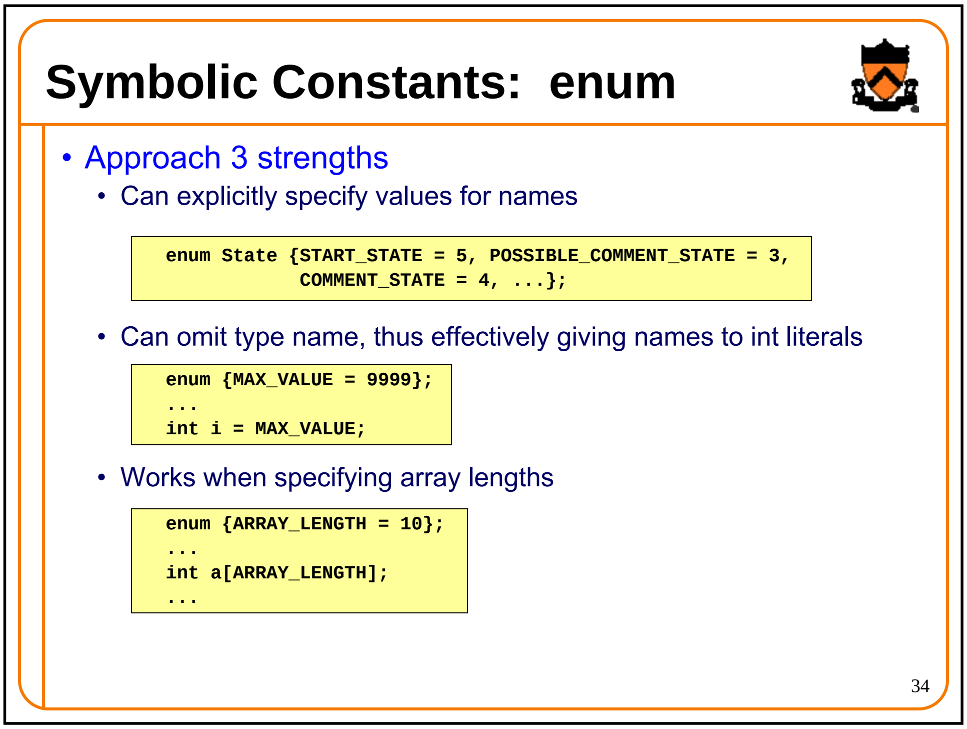
Defines a variable named “state” to be of type “enum State”



Symbolic Constants: enum

- Approach 3 note
  - Enumerated constants are interchangeable with ints
    - START_STATE is the same as 0
    - POSSIBLE_COMMENT_STATE is the same as 1
    - Etc.

```
state = 0;        /* Can assign int to enum. */
i = START_STATE;  /* Can assign enum to int. */
```



Symbolic Constants: enum

- Approach 3 strengths
  - Can explicitly specify values for names

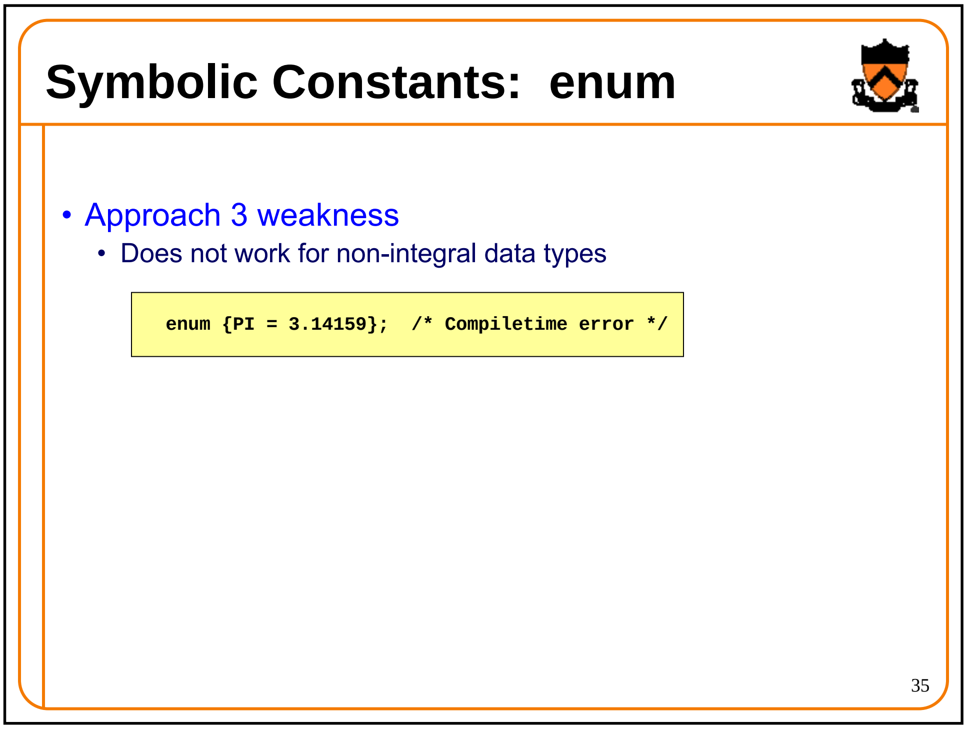
```
enum State {START_STATE = 5, POSSIBLE_COMMENT_STATE = 3,
            COMMENT_STATE = 4, ...};
```

  - Can omit type name, thus effectively giving names to int literals

```
enum {MAX_VALUE = 9999};
...
int i = MAX_VALUE;
```

  - Works when specifying array lengths

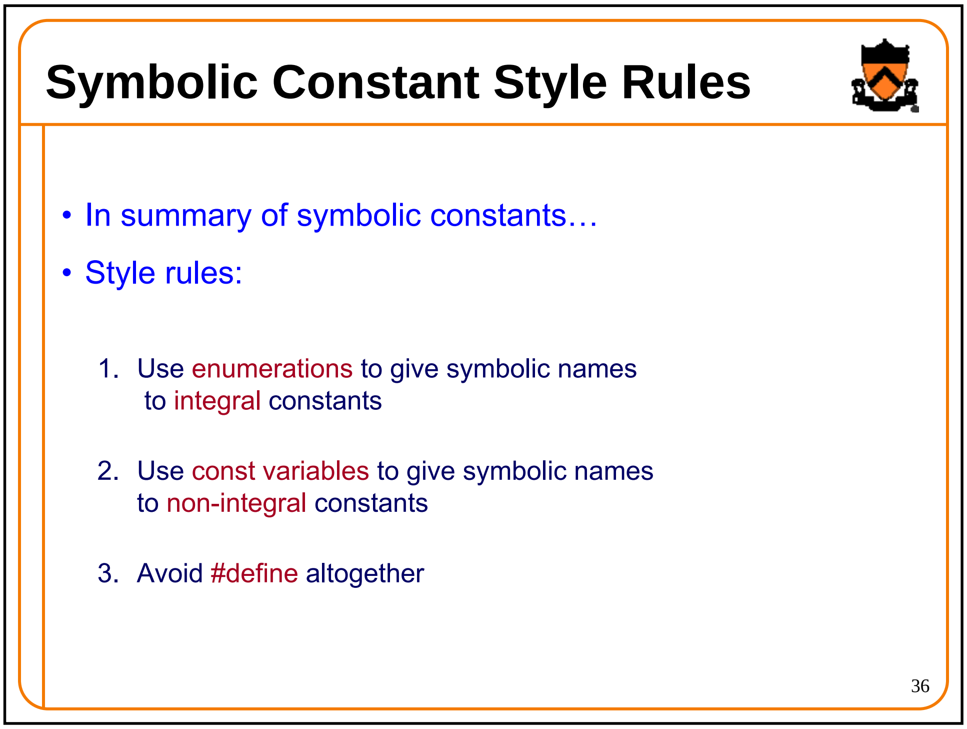
```
enum {ARRAY_LENGTH = 10};
...
int a[ARRAY_LENGTH];
...
```



Symbolic Constants: enum

- Approach 3 weakness
  - Does not work for non-integral data types

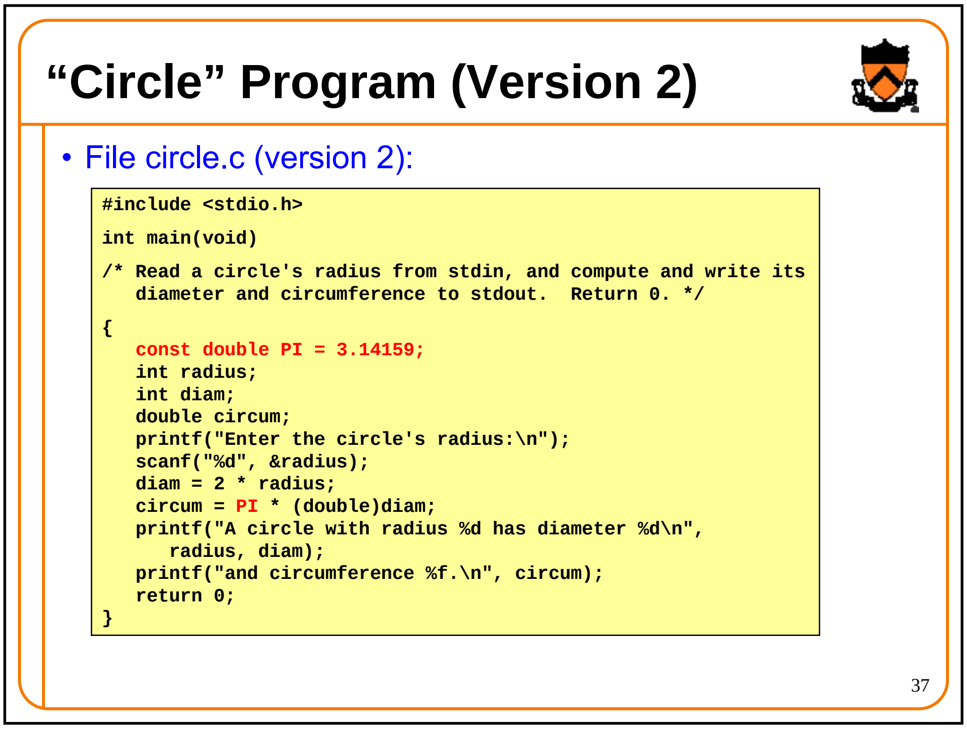
```
enum {PI = 3.14159};  /* Compiletime error */
```



Symbolic Constant Style Rules

- In summary of symbolic constants…
- Style rules:

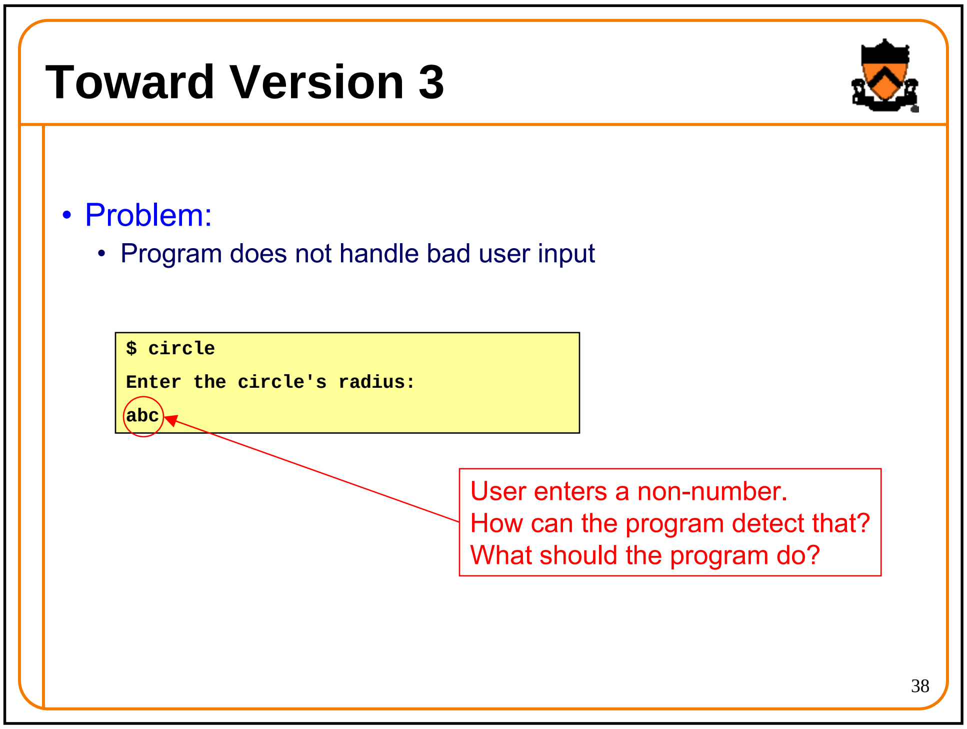
1. Use enumerations to give symbolic names to integral constants
2. Use const variables to give symbolic names to non-integral constants
3. Avoid #define altogether



“Circle” Program (Version 2)

- File circle.c (version 2):

```
#include <stdio.h>

int main(void)

/* Read a circle's radius from stdin, and compute and write its
   diameter and circumference to stdout.  Return 0. */

{
   const double PI = 3.14159;
   int radius;
   int diam;
   double circum;
   printf("Enter the circle's radius:\n");
   scanf("%d", &radius);
   diam = 2 * radius;
   circum = PI * (double)diam;
   printf("A circle with radius %d has diameter %d\n",
      radius, diam);
   printf("and circumference %f.\n", circum);
   return 0;
}
```

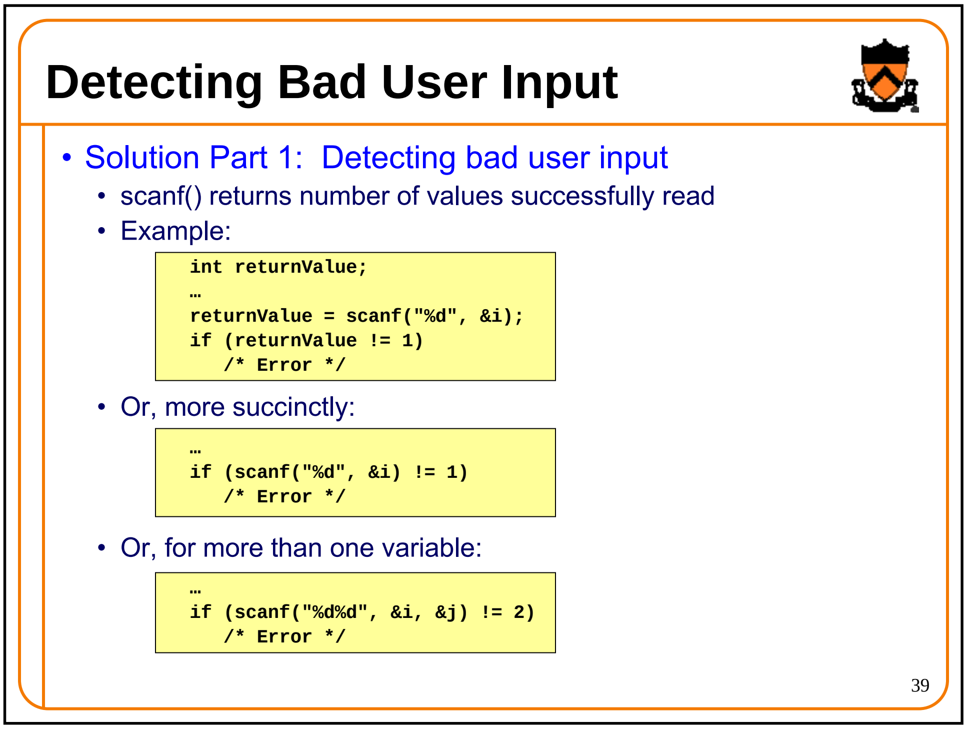


Toward Version 3

- Problem:
  - Program does not handle bad user input

```
$ circle

Enter the circle's radius:

abc
```

User enters a non-number.
How can the program detect that?
What should the program do?



Detecting Bad User Input

- Solution Part 1: Detecting bad user input
  - scanf() returns number of values successfully read
  - Example:

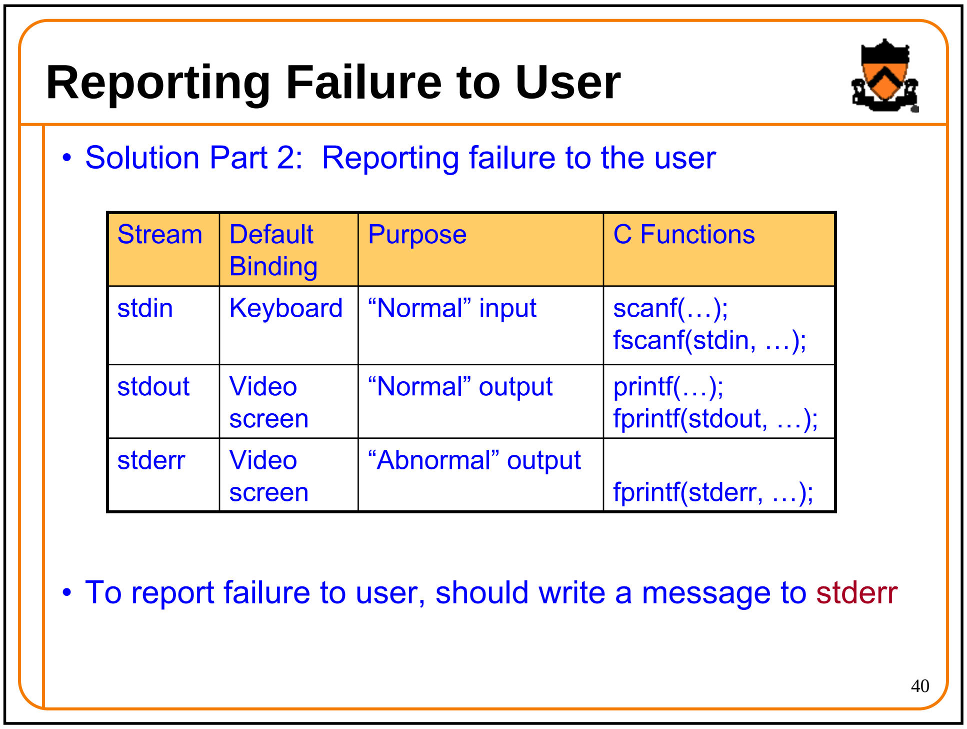
```
int returnValue;
…
returnValue = scanf("%d", &i);
if (returnValue != 1)
   /* Error */
```

  - Or, more succinctly:

```
…
if (scanf("%d", &i) != 1)
   /* Error */
```

  - Or, for more than one variable:

```
…
if (scanf("%d%d", &i, &j) != 2)
   /* Error */
```



Reporting Failure to User

- Solution Part 2: Reporting failure to the user

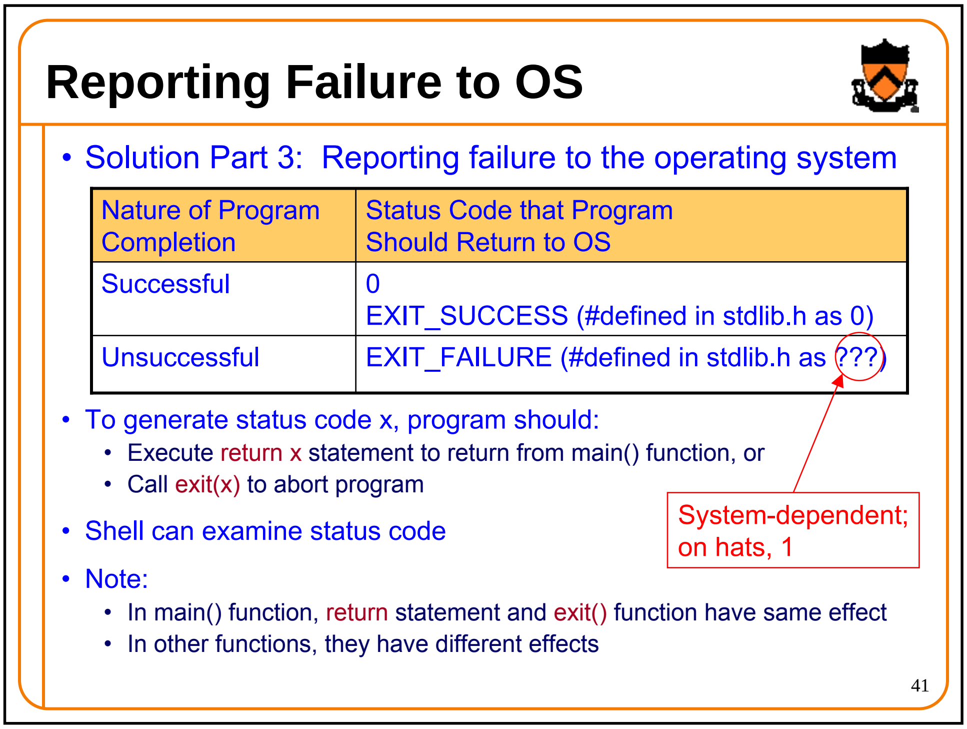
Stream	Default Binding	Purpose	C Functions

stdin	Keyboard	“Normal” input	scanf(…);
			fscanf(stdin, …);

stdout	Video screen	“Normal” output	printf(…);
			fprintf(stdout, …);

stderr	Video screen	“Abnormal” output	fprintf(stderr, …);

- To report failure to user, should write a message to stderr



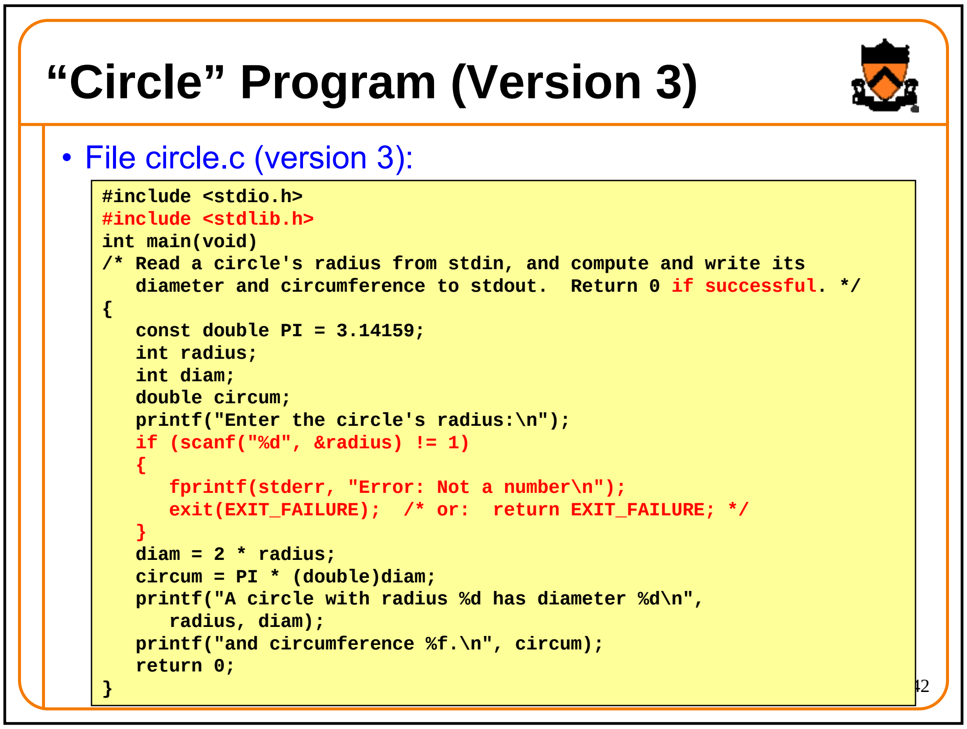
Reporting Failure to OS

- Solution Part 3: Reporting failure to the operating system

Nature of Program Completion	Status Code that Program Should Return to OS
Successful	0 EXIT_SUCCESS (#defined in stdlib.h as 0)
Unsuccessful	EXIT_FAILURE (#defined in stdlib.h as ???)

- To generate status code x, program should:
  - Execute return x statement to return from main() function, or
  - Call exit(x) to abort program
- Shell can examine status code
- Note:
  - In main() function, return statement and exit() function have same effect
  - In other functions, they have different effects

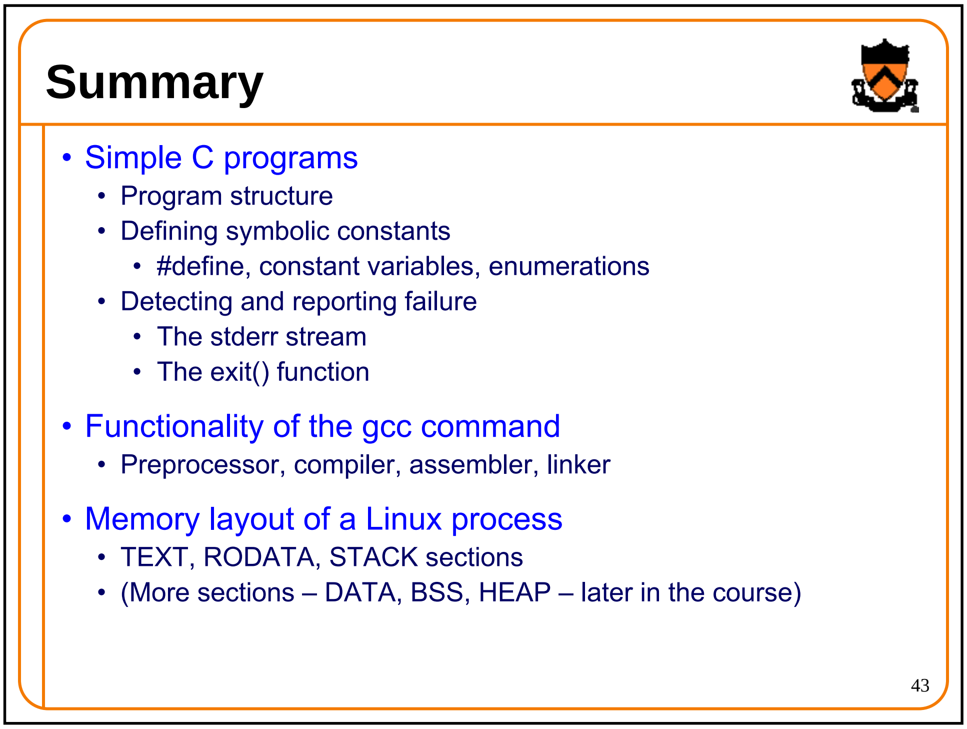
System-dependent; on hats, 1



“Circle” Program (Version 3)

- File circle.c (version 3):

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
/* Read a circle's radius from stdin, and compute and write its
   diameter and circumference to stdout.  Return 0 if successful. */
{
   const double PI = 3.14159;
   int radius;
   int diam;
   double circum;
   printf("Enter the circle's radius:\n");
   if (scanf("%d", &radius) != 1)
   {
      fprintf(stderr, "Error: Not a number\n");
      exit(EXIT_FAILURE);  /* or:  return EXIT_FAILURE; */
   }
   diam = 2 * radius;
   circum = PI * (double)diam;
   printf("A circle with radius %d has diameter %d\n",
      radius, diam);
   printf("and circumference %f.\n", circum);
   return 0;
}
```



Summary

- Simple C programs
  - Program structure
  - Defining symbolic constants
    - #define, constant variables, enumerations
  - Detecting and reporting failure
    - The stderr stream
    - The exit() function
- Functionality of the gcc command
  - Preprocessor, compiler, assembler, linker
- Memory layout of a Linux process
  - TEXT, RODATA, STACK sections
  - (More sections – DATA, BSS, HEAP – later in the course)