Functions in C

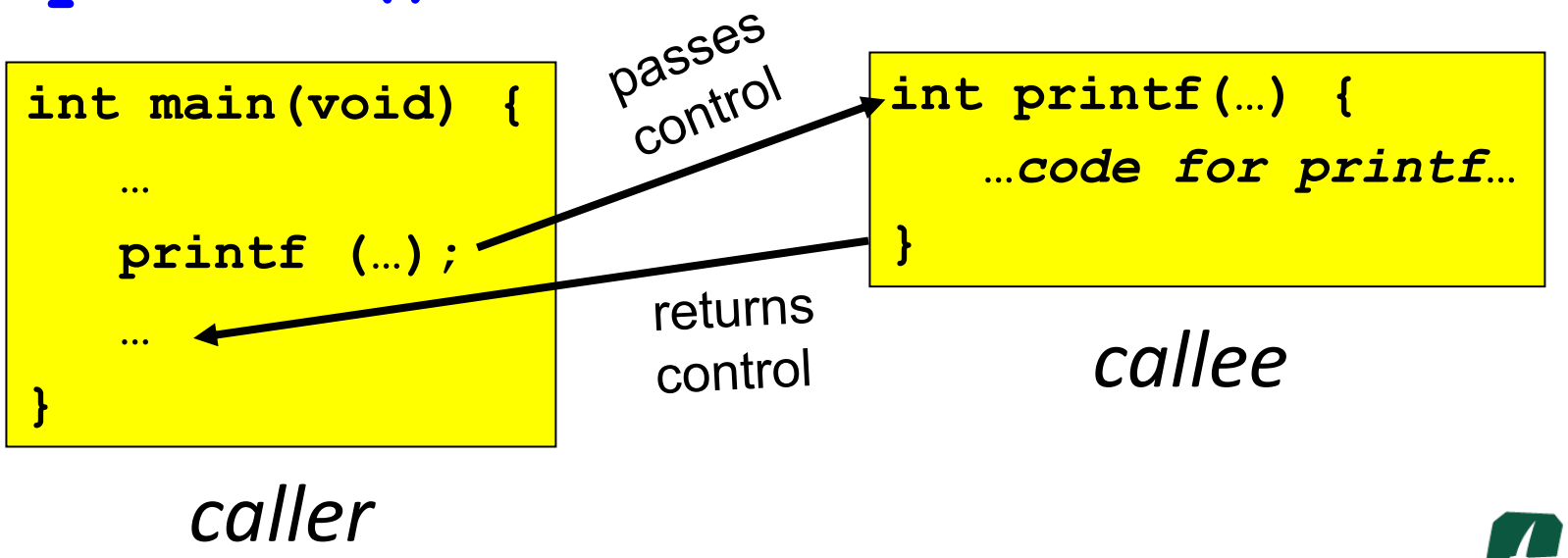
ITSC 2181: Introduction to Computer Systems
UNC Charlotte
College of Computing and Informatics

Functions in C

- **Functions** are also called *subroutines* or *procedures*
- One part of a program *calls* (or invokes the execution of) the function

(see `functions.c` in *Code Samples and Demonstrations* in *Canvas*)

Example: **`printf()`**



Are Functions Necessary?

Alternative: just copy the source code of **`printf()`** into the caller, everywhere it is called.

```
int main(void) {
    …
    …code for printing something…
    …
    …code for printing something else…
    …
    …code for printing something else…
    …
}
```

This is called ***inlining*** the function code. Usually **not** the best solution.

Reasons to Use Functions

- Functions **improve modularity**
  - reduce duplication, inconsistency
  - improve readability, easier to understand
  - simplify debugging
    - test parts – unit testing
    - then the whole – system/functional testing
- Allows creation of **libraries** of useful "building blocks" for common processing tasks

Function Return Values

- The **simplest** possible function has no return value and no input parameters. For example:
- Useful?

```
void abort (void)
```

- The next simplest case: value returned, but no input parameters. For example:

```
char getchar (void)
int rand (void)
clock_t clock (void)
```

What Values Can a Function Return?

- The **datatype** of a function can be **any** of:
  - integer or floating point number
  - `struct`s and unions
  - enumerated constants
  - `void`
  - pointers to any of the above (more on this later)
- Each function's type should be **declared before use**

How Many Values Returned?

- A function can return **at most one value**
- What if you need a function to return **multiple** results?
- Example: you provide the radius and height of a cylinder to a function, and want to get back…

1. surface area

and

2. volume of the cylinder

How Many … (cont'd)

- Choice #1: make the return type a ***struct***

```
typedef struct { //similar to an object
      int area;  // first field
      int vol;   // second field
} mystruct;

mystruct ans;
mystruct cyl (int , int );

int main(void) {
   …
   ans = cyl (r, h);
}
```

How Many … (cont'd)

- Choice **#2**: use ***global* variables**
  - global variables are ***visible*** to (and can be updated by) **all** functions

```
double area, vol;
void cyl (int , int );

int main(void) {
  …
  cyl (r, h);
}
```

```
void cyl (int r, int h)
{
  area = h * (2 * PI * r);
  vol = h * (r * r * PI);
}
```

☠ *common source of bugs* ☠
use of global variables

(see **`cylinder.c`** in *Code Samples and Demonstrations* in *Canvas*)

How Many … (cont'd)

- Choice #3: pass parameters by reference **using pointers**, instead of by value
  - allows them to be updated by the function
- Example: *later, when we talk about pointers…*

Function Side Effects

- Besides the value returned, these are things that *may be* changed by the execution of the function
- Examples
  - input to or output by the computer
  - changes to the state of the computer system
  - changes to global variables
  - changes to input parameters (using pointers)
- There are **problems** with side effects; *we'll come back to this...*

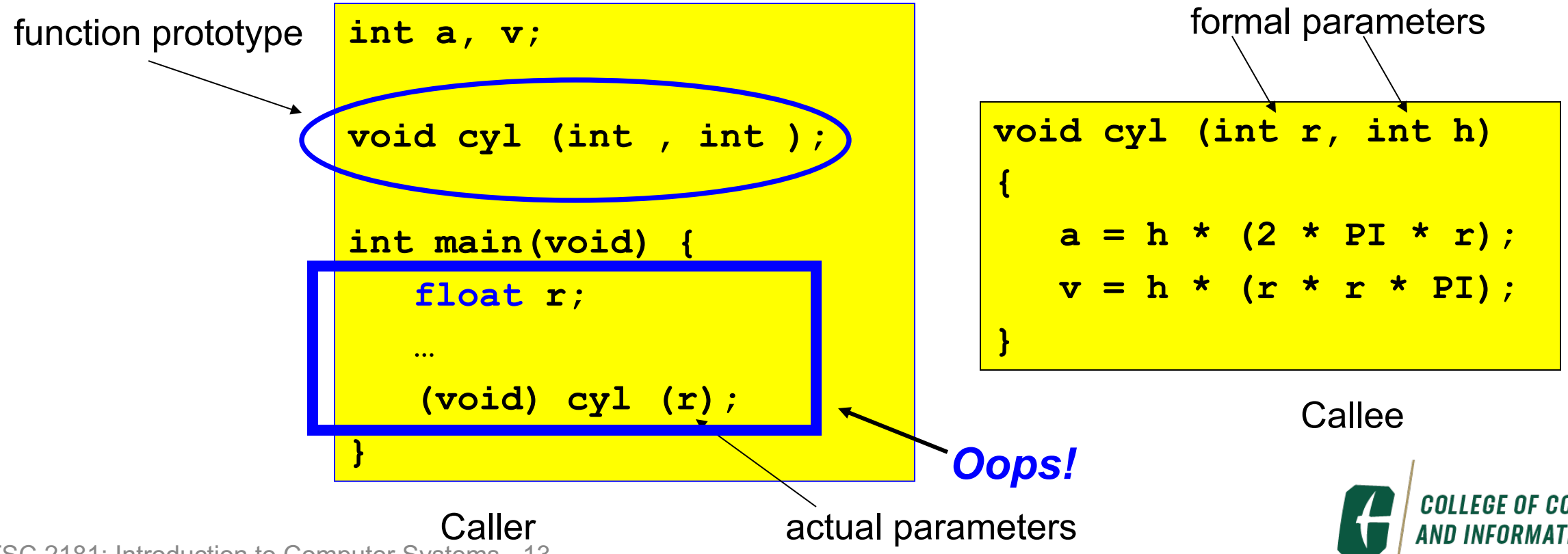
☠ *common source of bugs* ☠
side effects in functions and expressions

Input Parameters of a Function

- Often called ***arguments*** of the function
- Two types
  - ***formal* or *abstract*** – parameter declarations in the function definition
  - ***actual* or *concrete*** – the actual values passed to the function at run time
- If **no** input parameters to the function, leave empty, or use the `void` keyword

Input Parameters of a Function (cont'd)

- The **number and value** of actual parameters should match the number and type of formal parameters



Parameter Passing

- Parameters are passed using ***call-by-value***
  - i.e., a **copy of the parameter value** is made and provided to the function
- Any changes the function makes to this (copied) value have **no effect** on the caller's variables

Input Parameters (cont'd)

Example:

```
float a, v;
void main ( )
{
    int r, h;
    ...
    (void) cylbigger (r, h);
    ...
```

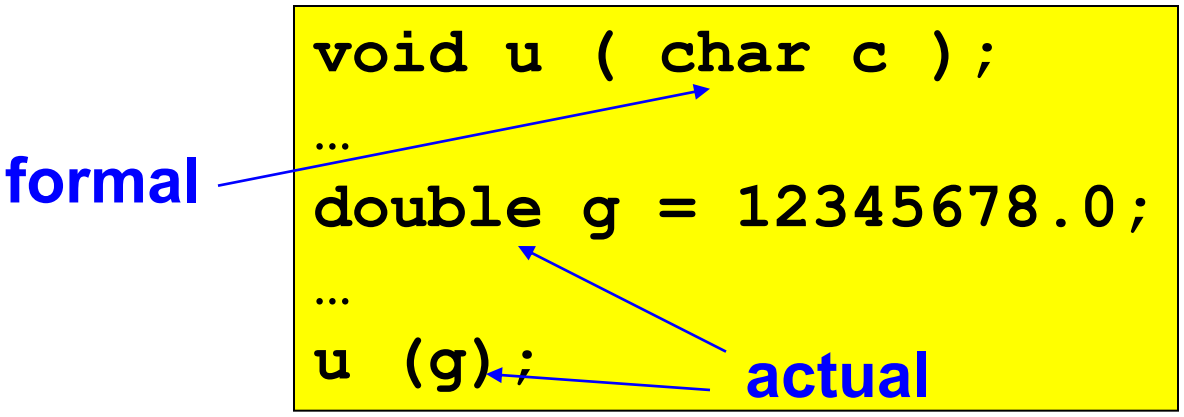
(see **functions.c** in *Code Samples and Demonstrations* in *Canvas*)

```
void cylbigger (int r, int h)
{
    r = 2 * r;
    h = 2 * h;
    a = h * (2 * PI * r);
    v = h * (r * r * PI);
}
```

does not change caller's variables **r** and **h**

Types for Function Arguments

In C, an **implicit type conversion** occurs if **actual** argument type is different from **formal** argument type




```
g = 12345678.0
c = 78
```

☠ *common source of bugs* ☠
overlooking type differences in parameters

no compiler warnings!

Advice: more predictable if you cast it yourself

(see `implicit_conversion.c` in *Code Samples and Demonstrations* in *Canvas*)

Must Declare Function **Before** Use

Program **with** compilation errors

```
#include <stdio.h>

int main (void)
{
    float w, x, y;
    …
    w = f(x, y);
    …
}

float f (float x, float y)
{
    …
}
```

Program **without** compilation errors

```
#include <stdio.h>

float f (float x, float y)
{
    …
}

int main (void)
{
    float w, x, y;
    …
    w = f(x, y);
    …
}
```

(see `order.c` in *Code Samples and Demonstrations* in *Canvas*)

Why should this make a difference?

Declare Before... (cont'd)

- Approaches
  1. (unusual) locate the **function definition** at the beginning of the source code file, or…
  2. **(usual)** put a ***function prototype*** at the beginning of the source code (actual function definition can appear anywhere)

Declare Before... (cont'd)

Program **without** compilation errors

```
#include <stdio.h>

float f (float , float );

int main (void)
{
    float w, x, y;
    …
    w = f(x, y);
    …
}

float f (float x, float y)
{
    …
}
```

← function prototype

(see `order.c` in *Code Samples and Demonstrations* in *Canvas*)

Side Effects, Again

- Q: If a variable is referenced **multiple times** in a single statement, and modified (by side effects) one of those times, do the other references see the side effect?
- Examples:

```
x = 1;
b = --x && x;
```

```
a = 2;
b = ++a;
c = a + a;
```

```
a = 2;
if (a++)
    b = a;
```

```
a = 2;
b = ++a + a;
```

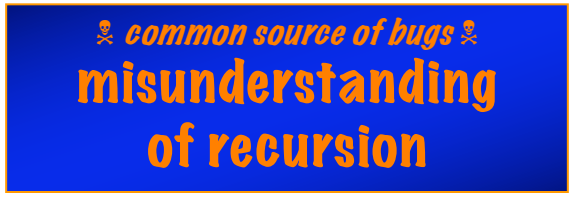
```
a = 2;
b = f( ++a, a);
```

```
a = 2;
b = ++a, c = a;
```

```
a = 2;
x = (++a > 2) ? a : 5;
```

Recursion

- What about `f()` calling `f()` ???
- A powerful and flexible way to iteratively compute a value
  - *although this idea seems modest, recursion is one of the most important concepts in computer science*
- Each iteration must temporarily store some input or intermediate values while waiting for the results of recursion to be returned

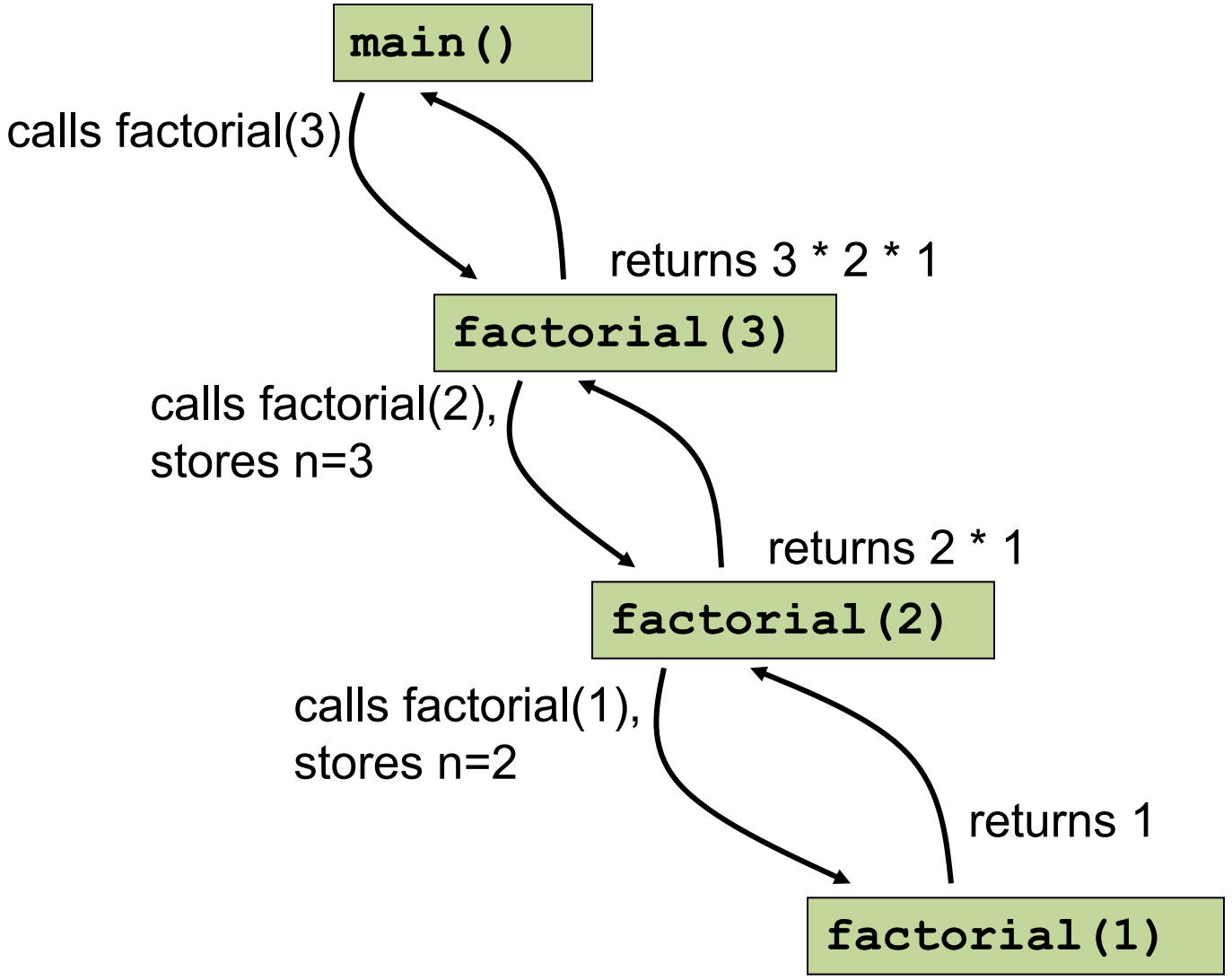


Recursion Example

```
...
int main (void)
{ …
   int n = 3;
   w = factorial( n );
…
}

int factorial(int n)
{
   if (n == 1)
      return 1;
   else
      return n * factorial(n-1);
}
```

Recursion Example… (cont'd)



Recursion ... (etc)

- What does the function
  f(*n*) = f(*n*-1) + f(*n*-2) (and f(1) == f(0) == 1) return for n = 5?

```
long long int f (long long int n)
{
  if ((n == 1) || (n == 0))
    return 1;
  else
    return (f(n-1) + f(n-2));
}
```

(see `fib.c` in *Code Samples and Demonstrations* in *Canvas*)

what function is this? any problems if n = 50?
code it and try!

Recursion or Iteration?

- Every recursion can be **rewritten** as a combination of
  1. a **loop** (iteration), **plus…**
  2. **storage** (a stack) for intermediate values

How Big Should A Function Be?

- **Too small** (100 line program, 20 functions)???
- **Too large** (10,000 line program with 2 functions)???
- Just right ? (Linux recommendations)
  - "Functions should ... do just one thing...[and] fit on one or two screenfuls of text"
  - "... the number of local variables [for a function] .... shouldn't exceed 5-10"

Top-Down Programming in C

- Procedural programming languages encourage a way of structuring your programs:
  - start with the basics
  - then progressively fill in the details
- Ex.: writing a web browser
  - how does one get started on a large program like this?

The C Standard Library

- Small set of useful functions, standardized on all platforms
- Definitions are captured in **24** header files
- Includes functions to do tasks such as:
  - Input/output processing: **`<stdio.h>`**
  - String handling: **`<string.h>`**
  - Mathematical computations: **`<math.h>`**
  - Memory management: **`<stdlib.h>`**
  - Generating random numbers: **`<stdlib.h>`**
  - Date and time processing: **`<time.h>`**

References

- S. J. Matthews, T. Newhall and K. C. Webb, *Dive into Systems*, Version 1.2. Free online textbook, available at: https://diveintosystems.org/book/

- K. N. King, *C Programming: A Modern Approach*, 2nd Edition. W. W. Norton & Company. 2008.

- D.S. Malik, *C++ Programming: From Problem Analysis to Program Design*, Seventh Edition. Cengage Learning. 2014.