University of Illinois at Urbana-Champaign
Dept. of Electrical and Computer Engineering

ECE 220: Computer Systems & Programming

C Functions and Examples

1

Signature Include Name and Types for Inputs and Outputs

Let's talk about how functions are used in **C**.

A function's **signature*** defines the function's
- **name**,
- **number and type of parameters**, and
- **type of value returned**.

*Sometimes called a function's interface.

2

Function Signature Needed to Call Function

A function's signature is
needed to call the function:

- compiler can **check number of arguments** (for each parameter, caller must pass an **argument**)
- compiler can **convert to correct types** (or refuse to do so)

3

Declare a Function to Use It, But Define It Somewhere

Function **declarations** include
- **just a signature** (followed by semicolon)
- parameter names strongly encouraged
- **used to generate calls**

Function **definitions** include
- **signature followed by code** (a compound statement)
- parameter names required
- **used to generate code for function** (also suffices to generate calls)

4

Pitfall: Leaving Out Parameter Names

A declaration without parameter names:

```
// draw a rectangle of *'s
// given height and width
int32_t draw_rectangle
    (int32_t, int32_t);
```

So … which argument is which?

Compiler cannot help: the types are identical.

Always include parameter names.

Pitfall: Leaving Out Declarations

Early **C** standards did not require declarations to call functions.

Instead used auto-conversion and defaults:

- integer arguments converted to **`int`**
- floating-point arguments converted to **`double`**
- return value defaulted to **`int`**.

These **assumptions often fail**, but compiler can not help without a signature!

C Passes Arguments to Functions by Value

C uses call by value.

Function arguments/parameters are **values of expressions**.

Copies of values are passed to a function.

- The **function owns the copies** (and may change them).
- The **function cannot change the original values**,
- even if they correspond to values of variables.

LC-3 Call Sequence Illustrates Operation of Call by Value

To understand how call by value works, recall the call sequence in LC-3:

1. Caller **evaluates arguments** (expressions) and **pushes values** onto stack.
2. **Copies** on the stack **form the parameters** portion of the function's stack frame.
3. Executing function **can modify its parameter values** (the copies).
4. When function returns, **caller discards copies** from stack (by popping them).

Write C Functions to Use What We Have Learned

Let's write a few C functions.

Our goals:
- learn how to use various operators,
- learn how use control constructs,
- understand scope and call by value, and
- think about argument checking and error handling.

First Task: Reverse Bits in an Integer

First task:
reverse the bits in a 32-bit integer.

For example (with 6 bits),
- given **001010**, produce **010100**, or
- given **110010**, produce **010011**.

What is the function signature?

Start by Developing the Function Signature

Let's **call the function `reverse_bits`**.

Argument type? **`uint32_t`**

Return type? **`uint32_t`**

Why unsigned?

Safer with bitwise operations
(avoids any special treatment of sign bit).

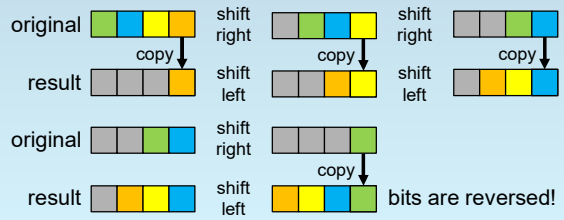
```
uint32_t reverse_bits(uint32_t arg);
```

How can we approach the problem?

A Simple Way to Reverse Bits

How about moving one bit at a time?

Copy, shift, repeat.



Declare Variables Based on Our Approach

First, we need a couple of variables…

```
uint32_t reverse_bits (uint32_t arg)
{
    int32_t i; // iteration counter
    uint32_t result = 0;
    ...
```

What about the "original" value?

Just use `arg`!

One `for` loop Covers the Reversal Algorithm

Now for the loop…

```
    for (i = 0; 32 > i; i++) {
        result = ((result << 1) |
                  (arg & 1));
        arg >>= 1;
    }
```

(By adding two shifts, we avoid writing a second bit copy operation outside the loop.)

`result` Has the Reverse Bits

All that remains is to return the reversed bits.

```
        ...
        return result;
    }
```


What Happens to `reverse_bits`' Changes to `arg`?

Let's write a call…

```
    uint32_t x = 42;
    uint32_t x_rev;
    x_rev = reverse_bits (x);
```

`reverse_bits` changes the value of `arg`!

So does x change value?

No. C uses call by value.

What Happens to `reverse_bits`' Changes to `arg`?

What about this call?

```
uint32_t arg = 42;
uint32_t arg_rev;
arg_rev = reverse_bits (arg);
```

Now does `arg` (shown) change? **No.**

`arg` in the function is **scoped in the function**.

`arg` in the code above is **scoped in the enclosing function** (not `reverse_bits`!).

Second Task: Convert Octal ASCII to 2's Complement

Second task: **convert a 5-digit octal number with a sign (all in ASCII) into a 2's complement value**.

For example,

- **"+10201"** converts to **4225** ($=8^4+2\times8^2+8^0$), and
- **"-00321"** converts to **-209** ($= -(3\times8^2+2\times8^1+8^0)$).

What is the function signature?

Start by Developing the Function Signature

Let's **call the function `convert_octal`**.

Argument types? **`six chars`**

Return type? **`int16_t`**

(because it fits!)

```
int16_t convert_octal
  (char sign, char dmax, char d2,
   char d3, char d4, char dmin);
```

How can we approach the problem?

A Method to `convert_octal` to 2's Complement

As we do in assembly:

- **subtract '0'** from each digit,
- **shift** the resulting three bits per digit left **to match the digit's place value**, and
- **add the digits up** (or OR them together).

Handle the sign last.

Declare Variables Based on Our Approach

First, we need a variable…

```
int16_t convert_octal
    (char sign, char dmax, char d2,
     char d3, char d4, char dmin)
{
    int16_t total;    // result
```


Our Variables Require a Sequence

We can not use a loop with distinct names!

```
total  = ((dmax - '0') << 12);
total += ((d2   - '0') <<  9);
total += ((d3   - '0') <<  6);
total += ((d4   - '0') <<  3);
total += ((dmin - '0') <<  0);
```

What about the sign?

`total` Has the Converted Value

Negate if necessary. Then return the result.

```
        if ('-' == sign) {
            total = -total;
        }
        return total;
    }
```


Third Task: Play a Number-Guessing Game

Third task: **play a number-guessing game with the human user**.

The computer chooses a number from 1 to 10.

The human gets a certain number of guesses.

Human wins if they can guess the number.

Human loses if they run out of guesses.

Computer gives feedback based on how close each guess is to the secret number.

What is the function signature?

Start by Developing the Function Signature

Let's **call the function `guessing_game`**.

Argument types? **design choice**

Return type? **design choice**

What information do we need to play?

secret number (1 to 10)

number of guesses allowed (> 0)

Let's choose to make both parameters.

Functions Should Rarely Return Type `void`

What about the return type?

One possibility: **`void`**

`void` is the type of "nothing" in **C**.

In other words,
- if a function returns nothing,
- the function's return type is **`void`**.

Choose void rarely.

Returning an Error Value Implies that Callers Check It

Why avoid `void`?

One piece of information returned:
- did the function call succeed?
- If function always succeeds,
- you might choose to return **`void`**.

Then others write code using your function
- say in 100 places (**call sites**).
- None of the calls check for failure.

Later, you change the function. Now it can fail.

Now what? Oops. Fix all 100 calls.

Let's Return an `int32_t` from Our Game Function

What were we talking about?

Oh, right, **`guessing_game`**.

Let's return an **`int32_t`**.

So we have

```
int32_t guessing_game
    (int32_t number,
     int32_t num_tries);
```

(As mentioned earlier, the names tell us which parameter is which.)

Each Returned Value Can Have a Different Meaning

What does the return value mean?

What are the possible outcomes?
- bad arguments (ex: -42 guesses allowed) → **2**
- human types something odd → **3**
- human wins the game → **1**
- human loses the game → **0**

Sometimes, we may identify other possibilities
- when we write the code,
- or if we need to change the code later.

Steps Needed for Our Guessing Game

How can we approach the problem?

1. Check argument values.
2. For each guess allowed,
   - prompt human for a guess,
   - read guess from keyboard, and
   - give feedback based on guess (including winning).
3. If human runs out of guesses, they lose.

Declare Variables Based on Our Approach

First, we need variables…

```
int32_t guessing_game
 (int32_t number, int32_t num_tries)
{
    int32_t guess;  // human's guess

    int32_t difference;
              // guess minus number
```


Start by Checking the Arguments

First step in the code is to check arguments:
- **number** must be in **[1,10]**, and
- **num_tries** must be at least **1**.

```
if (1 > number || 10 < number ||
    1 > num_tries) {
    return 2;
}
```

Recall that **return** terminates the function.

Next, Write the Loop Control

How many times does the loop body below execute?

```
while (0 < num_tries--) {
    // loop body
}
```

The loop body executes **num_tries times**.

Interact with the Human User

Prompt, read a number, and check for human error.

```
printf ("Enter a number: ");
if (1 != scanf ("%d", &guess)) {
  printf ("That's not a number!\n");
  return 3;
}
```

Again, **return** terminates the function.

Give Feedback for the Human's Guess

Give feedback
- based on the difference
- between the guess and the number.

```
    difference = guess - number;
    switch (difference) {
        ...
    }
} // end of the while loop
```


If Human Guesses Correctly, Return a Win

Difference of 0 means a correct guess.

```
case 0:
    printf ("You win!\n");
    return 1;
```

Again, **return** terminates the function.

So **this case needs no break**.

Cases of Positive and Negative Difference are the Same

Feedback can be the same regardless of sign.

```
case 1:
case -1:
    printf ("Close now...\n");
    break;
```

Add as many (or as few) cases as we like.

If the Human's Guess is Bad, Say So

For larger differences, use a default case.

```
default:
    printf ("No way.\n");
    break;
```

Remember that **`default`** catches all other values.

Finish the Function After the Loop Ends

What has happened if the loop ends?

Human has run out of guesses!

```
    ...
    printf ("Tough break!\n");
    return 0;
}
```


More Tasks! See the Code Online

More tasks: **PRINT_SLOT and PRINT_CENTERED**

The latter requires **C** syntax that you have yet to see in our class.

Both are included,

- along **with all of the functions discussed**,
- and a simple main that executes all,
- in one program **on the web page**.

