Week 5 Part I

Kyle Dewey

Overview

- Exam will be back Thursday
- New office hour
- More on functions
- File I/O
- Project #2

Office Hour

More on Functions

Recap...

- Consider a function `foo` that takes an `int` and a `char` and returns a `double`
- The function prototype for this looks like:

```
double foo( int, char );
```

Recap...

- Consider a function `foo` that takes an `int` and a `char` and returns a `double`
- Lets say it adds them and multiplies the result by `2.5`
- The function definition looks like:

```
double foo( int x, char y ) {
  return ( x + y ) * 2.5;
}
```

Questions

- Why are function prototypes needed?
- Where do function prototypes go?

Relationship to Variables

- Many similarities
- Variable declaration shares similarities to function prototypes
  - Sometimes called function declaration

```
double foo( int, char );
int bar;
...
```

Relationship to Variables

- Function declaration (function prototypes) are like variable declaration
- Function definition is like variable initialization
    - Though the values (i.e. the function definitions) can never be changed

Relationship to Variables

- Function names have the same rules as variable names (i.e. can’t start with a number, etc.)
- Can actually have variables that hold pointers to functions

Definition and Use

- Function prototypes go at the top of a file
- Function definitions can be anywhere in a file

```
#include <stdio.h>
int min( int, int );
int main();
int min( int x, int y ) {
  if ( x < y )
    return x;
  else
    return y;
}
int main() {
  int a, b;
  scanf( “%i %i”, &a, &b );
  printf( “%i\n”, min( a, b ) );
  return 0;
}
```

Calling a Function

- To make a function do work, we must **call** it
- A function call is **not** the same as a function definition
    - A function can be defined only once
    - A function can be called as many times as we want
    - Building a car versus driving a car

Function Call Semantics

- Say we have the following function definition:

```
int min( int x, int y ) {
  if ( x < y )
    return x;
  else
    return y;
}
```

Function Call Semantics

- Say we call this function like so:

```
int min( int x, int y ) {
  if ( x < y )
    return x;
  else
    return y;
}

int main() {
  int z = min( 5, 6 );
}
```

Function Call Semantics

- Semantically, this is equivalent to:

```
int main() {
  // int z = min( 5, 6 );
  int z;
  int x = 5;
  int y = 6;
  if ( x < y )
    z = x;
  else
    z = y;
}
```

Key Insight

- Function parameters are treated just like variables being declared and initialized

```
int main() {
  // int z = min( 5, 6 );
  int z;
  int x = 5;
  int y = 6;
  if ( x < y )
    z = x;
  else
    z = y;
}
```

One Property

- Function arguments are **copies** of what was passed, not what was passed itself
- This is called “call-by-value”

Call-by-Value

```
void changeIt( int x ) {
  x = 10;
}

int main() {
  int y = 1;
  changeIt( y );
  // what does y equal?
}
```

Call-by-Value

```
void changeIt( int x ) {
  x = 10;
}

int main() {
  int y = 1;
  // changeIt( y )
  int x = 10;
  // what does y equal?
}
```

Back to `scanf`

- `scanf` needs the addresses of the variables that will hold what was read in
- This is precisely because of call-by-value
  - We want to change the value of the variable itself, **not** a copy of the variable

Key Insight

- Function parameters are treated just like variables being declared and initialized

```
int main() {
  // int z = min( 5, 6 );
  int z;
  int x = 5;
  int y = 6;
  if ( x < y )
    z = x;
  else
    z = y;
}
```

A Second Property

- Type coercion occurs

```
int asInt( double x ) {
  return x;
}

int main() {
  int y = asInt( 5.5 );
}
```

A Second Property

- Type coercion occurs

```
int asInt( double x ) {
  return x;
}

int main() {
  // int y = asInt( 5.5 );
  double x = 5.5;
  int y = x;
}
```

Function Inputs / Outputs

- When a function **takes** a value, the value is an input (parameter / argument)
- The function's **return value** is whatever the function returned (an output)
  - `void` functions do not return values

Function Calls

- For non-`void` functions, a function call acts like an expression
- The function call returns whatever the output of the function was

Function Calls

```
int max( int x, int y ) {
  if ( x > y )
    return x;
  else
    return y;
}

int main() {
  int y = max( 4, 5 ) * 7 + 3;
}
```

Function parameters vs. `scanf`

- Reading in an input (`scanf`) is **not** the same as taking a parameter
  - `scanf`: get an input from the user
  - Parameter: get an input from within the program
- The parameter approach is far more flexible

```
int max( int x, int y ) {
 if ( x > y )
  return x;
 else
  return y;
}

int maxScanf() {
 int x, y;
 scanf( “%i %i”, &x, &y );
 if ( x > y )
  return x;
 else
  return y;
}
```

```
int max( int x, int y ) {
  if ( x > y )
    return x;
  else
    return y;
}

int maxScanf() {
  int x, y;
  scanf( “%i %i”, &x, &y );
  return max( x, y );
}
```

Function Outputs

- Printing out an output (`printf`) is **not** the same as returning a value
  - `printf`: print to the user via a terminal
  - Returning: output a value wherever the function is called
- Returning is far more flexible

```
int max( int x, int y ) {
  if ( x > y )
    return x;
  else
    return y;
}

void maxPrintf( int x, int y ) {
  if ( x > y )
    printf( “%i\n”, x );
  else
    printf( “%i\n”, y );
}
```

```
int max( int x, int y ) {
  if ( x > y )
    return x;
  else
    return y;
}

void maxPrintf( int x, int y ) {
  printf( “%i\n”, max( x, y ) );
}
```

Flexibility

- Functions are far more reusable than `printf` / `scanf`
  - Input / output can be changed later
  - `printf` / `scanf` always refer to the terminal

Example

- We want to define a function that takes the max of 4 integers
- First with `scanf` / `printf`

```
void max2() {
  int a, b;
  scanf( “%i %i”, &a, &b );
  if ( a > b )
    printf( “%i\n”, a );
  else
    printf( “%i\n”, b );
}
```

```
void max4() {
  int a, b, c, d;
  scanf( “%i %i %i %i”,
         &a, &b, &c, &d );
  if ( a >= b && a >= c && a >= d )
    printf( “%i\n”, a );
  else if ( b >= a && b >= c && b >= d )
    printf( “%i\n”, b );
  else if ( c >= a && c >= b && c >= d )
    printf( “%i\n”, c );
  else
    printf( “%i\n”, d );
}
```

Example

- We want to define a function that takes the max of 4 integers
- Now with parameters / return values

```
int max2( int a, int b ) {
  if ( a > b )
    return a;
  else
    return b;
}
```

```
void max2() {
  int a, b;
  scanf( “%i %i”, &a, &b );
  if ( a > b )
    printf( “%i\n”, a );
  else
    printf( “%i\n”, b );
}
```

```
int max4( int a, int b, int c, int d ) {
  return max2( max2( a, b ),
               max2( c, d ) );
}
```

Code Difference

- Using `printf` / `scanf`: 21 lines
- Without `printf` / `scanf`: 10 lines
  - Plus it's more flexible
  - Can be adapted to behave just like with `printf` / `scanf` with fewer lines!

The `main` Function

- Entry point for code outside of `ch`
- This function is called with command line arguments
- Should `return 0` on program success, or `return <nonzero>` on program failure

Command Line Arguments

- The arguments specified to a program on the command line
- For example:
    - `emacs foo.txt`
    - `foo.txt` is a command-line argument to `emacs`

```
int max( int, int );
int main( int argc, char** argv );

int main( int argc, char** argv ) {
  printf( “%i\n”, max( 5, 2 ) );
  return 0;
}

int max( int x, int y ) {
  if ( x > y )
    return x;
  else
    return y;
}
```

File Input / Output

File I/O

- Many programs manipulate files
  - `cat`: read a file
  - `emacs`: read & write to a file
  - `cp`: read from one file (source) and write to another (destination)

Terminal vs. Files

- Reading to / writing from either is very similar
- Main difference: files stay on the system, but terminal output does not usually stay
  - i.e. when you close the window, the files remain but the terminal output's gone

Terminal vs. Files

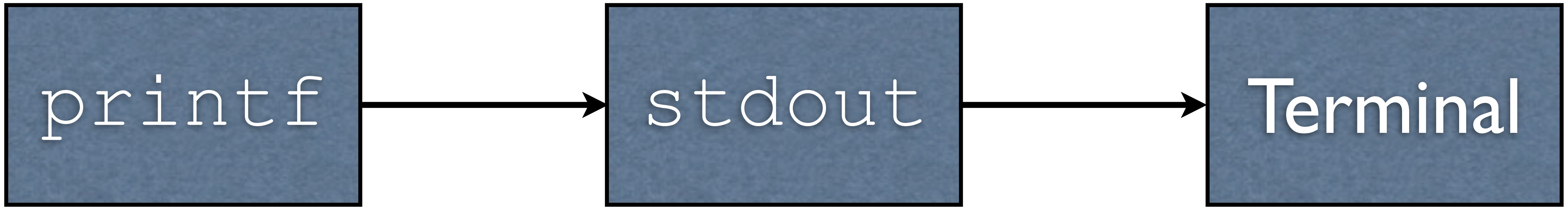
- The following functions behave on files:
  - `fscanf`
  - `fprintf`
  - `getc`
  - `putc`
- Sound familiar?

Difference

- These functions also require where they are reading from / writing to
  - `printf` always writes to the terminal, but `fprintf` can write anywhere
  - `scanf` always reads from the terminal, but `fscanf` can read anywhere

`printf` Revisited

- Technically, `printf` does not write to the terminal
  - It writes to `stdout` (standard output)
  - `stdout` is usually (but not always!) the terminal



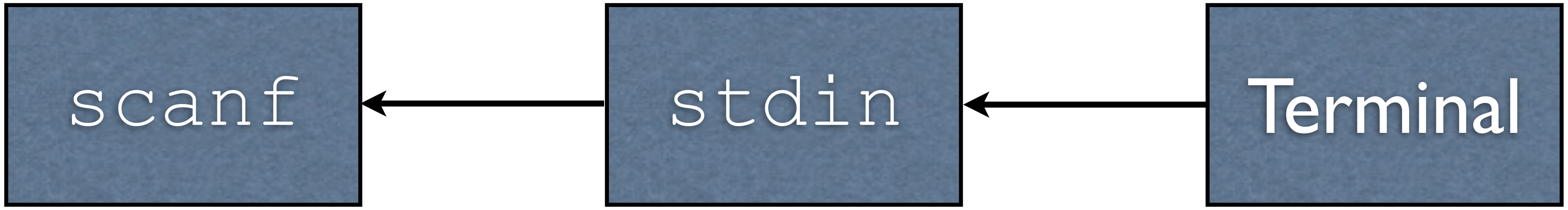
printf / fprintf

- These snippets do the exact same thing

```
printf( “hello” );
...
fprintf( stdout, “hello” );
```

scanf Revisited

- Technically, scanf does not read from the terminal
    - It reads from stdin (standard input)
    - stdin is usually (but not always!) the terminal



scanf / fscanf

- These snippets do the exact same thing

```
int x;
scanf( “%i”, &x );
...
int x;
fscanf( stdin, “%i”, &x );
```

getc / putc

- More equivalences

```
int c = getchar();
...
int c = getc( stdin );
```

```
putchar( ‘a’ );
...
putc( ‘a’, stdout );
```

stdin / stdout

- These are **file pointers** of type FILE*
- All these functions take file pointers

fopen

- Your own file pointers can be made by opening a file
- `fopen` is the tool for this

```
FILE* file = fopen( “file.txt”, “r” );
fprintf( file, “Hello” );
...
```

`fopen`

- First argument: the name of the file to open
- Second argument: what to open the file for
  - "`r`": read only. File must exist.
  - "`w`": write only. If a file with the same name already exists it will be deleted and overwritten.
- Return value: file pointer, or the special constant `NULL` if failure occurs

fclose

- When done with a file, call `fclose` on it
- Note that operations can be performed only on open files
    - If files aren’t open, the operations fail

```
FILE* file = fopen( “file.txt”, “r” );
fprintf( file, “Hello” );
fclose( file );
```

makeHelloFile.c,
catHelloFile.c

Techniques for Reading

- The data may need to be formatted in a certain way
    - i.e. if we read in a dictionary of words, how do we know when one word ends and another begins? When we are out of words? How many words there are?

Techniques for Reading

- We could specify the number of words beforehand
- We could separate each word by a letter that is in no word (such as a newline)
- Could end the words with some special non-word identifier
- Files all end with the special character `EOF` (end of file)

Techniques for Reading

```
3
foo
bar
baz
;;;
```

Techniques for Reading

- For more examples, see the additional materials
    - `p3_4.c`, `p3_5.c` (with corresponding `sensor1.txt`), `p3_6.c` (with corresponding `sensor2.txt`), `p3_7.c` (with corresponding `sensor3.txt`), `p3_8.c` (with corresponding `waves.txt`)

Project #2