Slide 1

Administrivia

- Reminder: Homework 7 due Tuesday.

Slide 2

Text Strings in C, A Little More

- A significant problem in working with strings is that there's no natural maximum size, so you have to decide how big to make the array of characters you will use to hold one — and then be sure you don't try to put in too many characters.
- Some library functions let you say how big the array is; some don't. *Always* be as careful as you can when working with strings; trying to store a string in an array not big enough is a source of "buffer overflows", which can lead to program crashes and more subtle problems, including security risks.
- Example — revisit the "change case" example, but prompt for filenames.

Slide 3

Arrays of Text Strings and Command-Line Arguments

- If you can have arrays of `int` and `char` and so forth — can you have arrays of text strings? Sure! They look like two-dimensional arrays of `char`, or like arrays of `char *`.
- Further, this is how C programs get input "from the command line" (e.g., when you write `gcc myprogram.c`, `gcc` somehow gets `myprogram.c`, right?):

  `main` can also be defined as

  ```
  int main(int argc, char * argv[]) { .... }
  ```

  where `argc` is the number of arguments, plus one, and `argv` is an array of strings containing the arguments. Example — let's write a simple "echo" program.

Slide 4

One More Topic — User-Defined Types

- So far we've only talked about representing very simple types — numbers, characters, text strings, arrays, and pointers. You might ask whether there are ways to represent more complex objects (e.g., a "money" object to represent dollars and cents — useful since floating-point is inexact for decimal fractions).
- Most/many programming languages (C included) do let you do that, in various ways . . .

Slide 5

User-Defined Types in C — `typedef`

- `typedef` just provides a way to give a new name to an existing type, e.g.:

```
typedef charptr char *;
```

- This can make your code more readable, or allow you to isolate things that might be different on different platforms (e.g., whether to use `float` or `double` in some application) in a single place.

Slide 6

User-Defined Types in C — `enum`

- In C (and in some other programming languages) an *enumeration* or an *enumerated type* is just a way of specifying a small range of values, e.g.

```
enum basic_color { red, green, blue, yellow };
enum basic_color color = red;}
```

  This can make code more readable, and sometimes combines nicely with `switch` constructs.

- Under the hood, C enumerated types are really just integers, though, and they can be ugly to work with in some ways (e.g., no nice way to do I/O with them).

Slide 7

User-Defined Types in C — `struct`

- More complex (interesting?) types can be defined with `struct`, which lets you define a new type as a collection of other types.
- One way to define uses `typedef`:

```
typedef struct {
    int dollars;
    int cents;
} money;
money bank_balance;
```

- Another way doesn't:

```
struct money {
    int dollars;
    int cents;
};
struct money bank_balance;
```

Slide 8

User-Defined Types in C — `struct`, Continued

- Either way you define a `struct`, how you access its fields is the same:

  `.` if what you have is a `struct` itself:

```
struct money bank_balance;
bank_balance.dollars = 100;
bank_balance.cents = 100;
```

  `->` if what you have is a pointer to a `struct`:

```
struct money * bank_balance_ptr = &bank_balance;
bank_balance_ptr->dollars = 100;
bank_balance_ptr->cents = 100;
```

- (Short example.)

Slide 9

User-Defined Types in C — `union`

- For completeness, we should mention that C also provides a way of defining a structure that can contain one of several alternatives ("this OR that", as opposed to the "this AND that" of `struct`) — `union`.
- See the discussion in the textbook for more about this; it can be useful, but can also make code more difficult to understand.

Slide 10

Minute Essay

- About the textbook — what did you like? what did you not like?