Slide 1

Administrivia

- Reminder: Homework 5 due next week. At least one more homework but possibly only one.

Slide 2

Minute Essay From Last Lecture

- Many people sort of got the key point (see “answer” slide).

Slide 3

A Little About the C Preprocessor

- C logically divides the process of producing an executable into distinct phases. First phase is "preprocessing".
- Preprocessing makes use of "preprocessor directives", which start with a #.
- Examples you've seen — `#include` to include information about library functions, `#define` to define constants.

Slide 4

A Little More About the C Preprocessor

- Other functionality includes macros and "conditional compilation":
- Macros can be used to do a kind of generic programming. Simple example:

```
#define max(a, b) ((a) > (b) ? (a) : (b))
```

  More complex macros can be used to generate multiple lines of code.
- Conditional compilation is often used to tailor library or other code to specific environments. Also allows writing `.h` files that can be included more than once without harm.
- More in chapter 14, some beyond the scope of this course. Focus is on relatively simple text manipulation.

Slide 5

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, such as one can do with classes in object-oriented languages.
- The answer is "yes, sort of" — C doesn't provide nearly as much syntactic help with object-oriented programming, but you can get something of the same effect. But first, some simpler user-defined types . . .

Slide 6

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 7

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 8

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 — something like a class in an object-oriented language, but with no methods and no way to hide fields/variables.

- Two versions of syntax (next slide) . . .

Slide 9

User-Defined Types in C — `struct`

- 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 10

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;
```

Slide 11

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 discussion in textbook about this; it can be useful, but can also make code more difficult to understand.

Slide 12

User-Defined Types and Library Code

- Library code often makes use of "opaque" types (e.g., `FILE`).
- Implementing this often involves separating functionality into interface (`.h` file containing type definitions, function declarations) and implementation (`.c` file containing function definitions.
- This leads into . . .

Slide 13

Separate Compilation — Review

- C (like many languages) lets you split large programs into multiple source-code files. Typical to put function declarations (headers), constants, etc., in file ending `.h`, function definitions (code) in file ending `.c`. Compilation process can be separated into "compile" (convert source to object code) and "link" (combine object and library code to make executable) steps.
- UNIX utility `make` can help manage compilation process. Can also be useful as a convenient way to always compile with preferred options. Idea behind `make` — have computer figure out what needs to be recompiled and issue right commands to recompile it.

Slide 14

Makefiles

- First step in using `make` is to set up "makefile" with "rules" describing how files that make up your program (source, object, executable, etc.) depend on each other and how to update the ones that are generated from others. Normally call this file `Makefile` or `makefile`.

  Simple example on sample programs page.
- When you type `make`, `make` figures out (based on files' timestamps) which files need to be recreated and how to recreate them.

Slide 15

Defining Rules

- Define dependencies for a rule by giving, for each "target", list of files it depends on.
- Also give the list of commands to be used to recreate target.

NOTE!: Lines containing commands must start with a tab character. Alleged paraphrase from an article by Brian Kernighan on the origins of UNIX:

> The tab in makefile was one of my worst decisions, but I just wanted to do something quickly. By the time I wanted to change it, twelve (12) people were already using it, and I didn't want to disrupt so many people.

Slide 16

Useful Command-Line Options

- `make` without parameters makes the first "target" in the makefile. `make foo` makes `foo`.
- `make -n` just tells you what commands would be executed — a "dry run".
- `make -f otherfile` uses `otherfile` as the makefile.

Slide 17

"Phony" Targets

- Normally targets are files to create (e.g., executables), but they don't have to be. So you can package up other things to do . . .
- Example — many makefiles contain code to clean up, e.g.:

```
clean:
        -rm *.o main
```

To use — `make clean`.

Slide 18

Variables in Makefiles

- You can also define variables, e.g.:
  - List of object files needed to create an executable. Then use this list to specify dependencies, command.
  - Pathname for a command, options to be used for all compiles, etc.
- Example:

```
objs = main.o foo.o
CFLAGS = -Wall -pedantic
main:   $(objs)
        gcc $(CFLAGS) -o main $(objs)
```

Slide 19

Predefined Implicit Rules

- `make` already knows how to "make" some things — e.g., `foo` or `foo.o` from `foo.c`.
- In applying these rules, it makes use of some variables, which you can override.
- A simple but useful makefile might just contain:

```
CFLAGS = -Wall -pedantic -O
```

- Or you could use

```
CFLAGS = -Wall -pedantic $(OPT)
OPT = -O
```

and then optionally override the `-O` by saying, e.g., `make OPT=-g foo`.

Slide 20

Minute Essay

- Had you seen `make` in another course or elsewhere?