Slide 1

## Administrivia

- Next homework will be on the Web probably tomorrow. I will send mail.

Slide 2

## A Little About `make` — Review

- Motivation: Most programming languages allow you to compile programs in pieces (“separate compilation”). This makes sense when working on a large program — when you change something, just recompile parts that are affected.
- Idea behind `make` — have computer figure out what needs to be recompiled and issue right commands to recompile it.

Slide 3

## Makefiles

- First step in using `make` is to set up “makefile” 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 4

## 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 -std=c99
```

- Or you could use

```
OPT = -O
CFLAGS = -Wall -pedantic -std=c99 $(OPT)
```

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

Slide 5

## Dynamic Memory and C

- With the C89 standard, you had to decide when you compiled the program how big to make things, particularly arrays — a significant limitation.
- Variable-length arrays in C99 standard help with that, but don't solve all related problems:

  In many implementations, space is obtained for them on "the stack", an area of memory that's limited in size.

  You can return a pointer from a function, *but* not to one of the function's local variables (because these local variables cease to exist when you return from the function).

Slide 6

## Dynamic Memory and C, Continued

- "Dynamic allocation" of memory gets around these limitations — allows us to request memory of whatever size we want (well, up to limitations on total memory the program can use) and have it stick around until we give it back to the system.

  (The trick here is that most implementations differentiate between two areas of memory, a "stack" used for local variables, and a "heap" used for dynamic memory allocation. Usually the former is more limited in size.)
- To request memory, use `malloc`. To return it to the system, use `free`. (For short simple programs you can skip this, but not good practice, since in "real" programs you may eventually run out of memory.)
- Python and Scala hide most of this from you — allocating space for objects is automatic/hidden, and space is reclaimed by automatic garbage collection. Makes for easier programming but possibly-unpredictable performance.

Slide 7

## Dynamic Memory and C, Continued

- Examples:

```
int * nums = malloc(sizeof(int) * 100);
char * some_text = malloc(sizeof(char) *
20);
free(nums);
```

- Some books/resources recommend "casting" value returned by `malloc`. Other references recommend the opposite! But you should check the value — if `NULL`, system was not able to get that much memory.
- (Example — improved sort program.)

Slide 8

## Function Pointers

- You know from more-abstract languages that there are situations in which it's useful to have method parameters that are essentially code. Some languages make that easy (functions are "first-class objects") and others don't, but almost all of them provide some way to do it, since it's so useful — e.g., providing a "less-than" function for a generic sort.
- In C, you do this by explicitly passing a pointer to the function.

Slide 9

## Function Pointers in C

- The type of a function pointer includes information about the number and types of parameters, plus the return type.
- Example — last parameter to library function `qsort` (in its `man` page). Call this by providing, in your code, a function with declaration

```
int my_compare(const char *, const char *);
```

  and using `my_compare` as the last parameter to `qsort`.
- (Example — improved sort program.)

Slide 10

## Pointers Versus Arrays — One More Thing

- We've said that pointers and arrays are in most contexts equivalent. One potential benefit of this is that it can make it easier to work with substrings, subarrays, etc.
- Example — one more revision of the string-length example, using recursion.

Slide 11

## Minute Essay

- TBA

Slide 12

## Minute Essay Answer

- TBA