Slide 1

Administrivia

- Reminder: Homework 7 due Tuesday. (We need to set a “not accepted past this date/time” so I can post solutions. Should it be end of the day Thursday?)
- (If there are other homeworks you haven’t turned in, or you turned in something but you know it wasn’t right — I will give some points for anything turned in by the deadline for Homework 7.)

Slide 2

Dynamic Memory and C

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

  In most 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 3

Dynamic Memory and C

- "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 not bother with `free`, but for longer and more complicated programs, you should clean up when you can, or eventually you may run out of memory.)

Slide 4

Dynamic Memory and C, Continued

- Examples:

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

- Book recommends "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 — program to generate N numbers and sort them.

Slide 5

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 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.
- 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 9

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 = 20;
```

  - `->` 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 if time permits.) (It didn't — there will be an example on the sample programs page.)

Slide 10

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.

Minute Essay

- None — quiz.

Slide 11