

Slide 1

Administrivia

- Reminder: Homework 3 due Wednesday.
- Next quiz a week from today.
- Sample solution for Homework 2 available on the Web, linked from the bottom of “lecture topics and assignments”.

Slide 2

Minute Essay From Last Lecture

- Most people said functions in C looked a little like something they'd seen before in Matlab or Python, except it seems like a bit more work in C. Not surprising — many things do.

I/O Redirection Revisited

Slide 3

- (Now that you've written at least one program, time to revisit this idea.)
- You may notice that when I'm being careful I talk about getting input "from standard input" (rather than "from the keyboard") and writing "to standard output" (rather than "to the screen")?
- Why? Command-line-oriented programs can get input from a variety of sources and can send output to a variety of destinations.

This is part of what makes the environment potentially powerful. I use it to semi-automate grading of your programs!

Input from "Standard Input"

Slide 4

- From the keyboard — you know how to do this.
- From a file — use `<`, e.g.,

```
./a.out < input1.txt
```

Useful if input can be many lines.
- From a "pipe", meaning from output of another program, e.g.,

```
echo in1 in2 | ./a.out
```

Useful as a way to repeat testing with the same inputs (since this compound command will be in the shell's history and thus easy to repeat).

Output to “Standard Output”

- To the screen — you know how to do this.
- To a file — use `>`, e.g.,

```
./a.out > output1.txt
```

Useful if you want to keep the output, e.g., to compare with what you expect.
Overwrites output file, or use `>>` to append.
- To a “pipe”, meaning to input of another program, e.g.,

```
./a.out | less
```

to “page through” output
or

```
./a.out | tee out1.txt
```

to have output show on screen and also be saved in file.

Slide 5

Functions in C — Recap

- Functions in C (as in other programming languages) are a way to break up a big problem into more manageable pieces and also to avoid duplication of code/effort.
- The basic idea is similar to mathematical functions (something that transforms input(s) to output), but functions in C (again as in many — though not all! — programming languages) can have “side effects”.

Slide 6

Functions in C, Recap Continued

Slide 7

- In C, unlike in many other languages, before you can call a function the compiler has to know about it. Two ways to do this:
 - Put the definition of the function before the call to it. Okay for short functions but not a great way to organize a program in general (“top-down” idea. And obviously(?) won’t work for library functions.
 - Include a “function declaration” describing the function’s parameters and return type before the call. The actual definition (code) can then be later in the file or in some other file. That line `#include <stdio.h>?` it includes declarations for `scanf`, `printf`, etc.

Functions and Scope

Slide 8

- In addition to a type and a name, each variable has a *scope* in which it’s valid. Variables declared inside a function can be used only within that function. Variables declared outside all functions can be used anywhere — *global variables* — though this is almost always a bad idea.
- One result — variables with the same name in different functions *are different variables*.

Functions and Parameters

Slide 9

- We said last time that functions have *parameters*. Another word for them is *arguments* (you will see this in some compiler error messages). More terminology:
 - *Formal parameters* are the parameters as viewed from the function — can think of these as additional variables whose scope is the function.
 - *Actual parameters* are the values with which the function is called.
- When a function is called, actual parameters are copied to formal parameters — “pass by value” — meaning that changes made in the function to its copies are not reflected in the calling program’s copies. Note also that actual parameters can be expressions.

Function Return Values

Slide 10

- Most functions return a value (but only one); it’s the value of the expression following the keyword `return`, in the function definition. The type of this value is given as part of the function definition. If you don’t want to return anything, can make this `void`. If you want to return two things? must use “pointer variables” (addresses).
- Function calls are expression, so they have a value — whatever is returned by the function.

Pointer Variables, Briefly

Slide 11

- (Normally we wouldn't do this just yet, but the textbook lets this cat out of the bag, and it *does* help in understanding `scanf`.)
- Motivation: Some functions need to return multiple values. In higher-level languages there are ways to do this via return values, but it's more trouble in C and not often done. Instead, you can make use of parameters declared as "pointer variables" — meaning that what is copied is . . . Well, back up a step.

Variables and Memory — Simplified View

Slide 12

- A crucial component of computer hardware is the "memory" (meaning random-access memory, not disk!). A good-enough-for-now approximation models this as a list of numbered locations/cells, each consisting of a fixed number of bits. An "address" is an index into this list; the corresponding bits are its "contents".
- Variables in programs correspond to one or more of these cells, and we can talk about the "address" of the variable (the index of the first cell) and its "value" (contents of the cell, interpreted based on the variable's type — e.g., the same bits mean one thing for a C `int` and another thing for a C `float` — assuming those are the same number of cells, which they often are but need not be).

Pointer Variables, Continued

- C programs that need to return multiple values can declare some parameters as “pointers”, as in this example:

```
int divide(int a, int b, int * quotient, int * remainder);
```

Slide 13

The * indicates that what is to be copied to the function is not a *value* but an *address*.

- To call such a function, you must provide an address. More than one way to do this, but for now the one we know about is the name of a variable preceded by the “address of” operator &. (“Aha!”?)
- Within the function, you can change the value at the address specified by this kind of parameter using the “dereference” operator * — e.g.,

```
*quotient = a/b;
```

Example

- As an example, revise the quadratic-equation program once more ...
- Computing and printing roots in a single function isn't a great design choice (my opinion, but probably shared by others) but was all we could do without pointer variables.
- Now that we have them, we can make use of them to write a function that just computes the roots, so a calling program can decide whether to print them or do something else with them (maybe use them in another calculation?).
But then what about cases where there are no (real-valued) solutions?
Could have function return how many roots there are.

Slide 14

Minute Essay

- None — quiz.
- Quiz rules:
 - Okay to consult textbook, course Web site, your own work (notes, programs, etc.).
 - Not okay to use computer for any purpose other than browsing the above.

Slide 15