Slide 1

Administrivia

- Next quiz a week from today.
- Homework 2 grades mailed (earlier today). Sample solution available on the Web, linked from the bottom of “lecture topics and assignments”.

Slide 2

`vim` Tips

- You may have discovered already that if you don’t know/remember many of the keyboard shortcuts (and `vim` is pretty much *all* keyboard shortcuts) it’s painful to use `vim`. I like text-based editors for this class because they’re easy to use remotely. There are others that may be easier to get started with, but . . .
- I think `vim` is a good editor for writing code: It does syntax highlighting of code in any language it “knows about” as well as automatic indentation. (Tidy up indentation by typing == on a line.) It also shows matching parentheses/braces, and if you put the cursor on one of those and press `%` it takes you to the match — or indicates there isn’t one. Helpful!
- If you have trouble remembering, try a “cheat sheet” of commands you want to remember.

Slide 3

`vim` Tips, Continued

- Short way to cut/copy/paste: `yy` ("yank") to copy a line. `dd` to delete a line. Both go into a buffer you can then insert with `p` or `P`. Precede the `yy` or `dd` with a number to get multiple lines. Or . . .
- You will probably like "visual mode": Put the cursor at the start of text to highlight and press `v`. Move the cursor to the end and then type `y` to copy or `d` to delete, and then use `P` to (re)insert.
- `.` repeats the most recent command (e.g., `dd`).
- You can search for text with `/`. Repeat search with `n`. Use `cw` to change a "word". Combine with `.` to do a quick repeated search-and-replace.

Slide 4

I/O Redirection Revisited

- (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!

Slide 5

Input from “Standard Input”

- 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).

Slide 6

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 7

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 8

Functions and Scope

- 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*.

Slide 9

Functions and Parameters

- 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. Notice also that actual parameters can be expressions.

Slide 10

Function Return Values

- 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.

Slide 11

Pointer Variables, Briefly

- (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.

Slide 12

Variables and Memory — Simplified View

- 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` — even assuming those are the same number of cells, which they often are but need not be).

Slide 13

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);`

  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;`

Slide 14

Example

- As an example, revise the quadratic-equation program once more . . .

- Computing and printing roots in a single function was never 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? To be continued . . .

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.