# Lecture 13

CSE 110

22 July 1992

We just spent a long time learning a lot of ins and outs of the C language, and guess what? We're almost done. There's a very short list of language features that we haven't already seen, and some of those we won't bother with. In other words: You know the language.

The class is supposed to concern itself primarily with how to program, and secondarily with the C language itself, and now we're finally getting to studying programming instead of C.

Accordingly, we spent some time working on writing a simple program: `type` Complete code for `type` appears in the notes for lecture 12, section 4.5.

The point of working on this in class is to argue about style and documentation, to think about issues of formatting, and most important, program structure and how to write a good program.

## 1 `type`, Episode 1

The `type` program is a fundamental MS-DOS utility.[1] When you run it, you give it one or more filenames as arguments. `type` gets the contents of each file in turn and writes that data to the standard output.

This has at least uses:

- You can use `type` to view a file by dumping its contents to the screen.
- You can use `type` to concatenate[2] two files, by redirecting the standard

[1] It's a fundamental utility in nearly all environments. For example, under UNIX, `type` is called `cat`.

[2] *concatenate* means to stick things together in a row. For example, if we concatenate the words 'foo' and 'bar' we get the word 'foobar'.

output of `type` into a destination file.

## 1.1 Notes About the Design

We're writing this program with the *top-down design* method, which is one of very few effective techniques for writing a real program.[3]

Top-down design means that we start by thinking about what our program has to do, in general terms: "It will have to open a file, read the data from the file, write the data to the standard output, and close the file. It will have to do that over and over for each file named on the command line." Some of the sentences in the description correspond obviously with C constructs: 'repeat over and over' means a `while`. `do`–`while`, or `for` loop. Other sentences get turned into functions, such as `read_data_from_file()`.

This description becomes our function `main`. Then we start writing that functions that we called from `main` that aren't written yet: `read_data_from_file()`, for example. If a function is simple and we can see right away how to write it, we just go ahead; if it seems more complicated, we can top-down design it in the same way.

Functions let us break a big problem into little problems. Each function solves a little problem; then all we have to do is put the functions together. If the little problems turn out to be too big to handle, we can break them down into even littler problems.

## 1.2 What we got Done Today

```
int main(int argc, char **argv)
{
  /* index of the name of the file we're working on now. */
  int i;
  FILE *cur_file;


  for ( i=1;  i < argc; i++ ) {
```

[3]Other techniques include: The *bottom-up* approach, where you implement a few fundamental primitive functions, and then some more complicated routines that depend on those, and then finally build up the program from the building blocks you've written; the *Mongolian hordes* technique, where you just dive in and write the first thing that comes into your head. The Mongolian hordes technique is not particularly effective.

```
    /* open a file and handle errors */
    cur_file = open_file(argv[i]);
    if (cur_file == NULL)
      continue;

    /* Get data from file, print data and handle errors*/
    spew(cur_file);

    /* close file and handle errors */
    /* postpone: announce that file is finished */
  }

  /* return success or failure code */
}

FILE * open_file(char *filename)
{
  FILE *file;

  file = fopen(filename, "r");

  if (file == NULL) {
    fprintf(stderr, "I couldn't open file %s!.\n", filename);
  }

  return file;
}

int /*?*/ spew(FILE *current)
{
  int c;

  while ((c = getc(current)) != EOF)
    putchar(c);

}
```

### 1.3 Some Notes About the Design Process

Our original plan was to have `open_file` handle errors and return 0 or 1 to `main` to indicate success or failure. But then we realized that won't work—on success, `open_file` gets a `FILE *` from `fopen`, and it *must* return this `FILE *` to `main` so that `main` can pass it off to the other functions that need it. The function to read the data from the file will need this value, as will the function that closes the file again.

So `open_file` must return a `FILE *` on success. The natural value to have it return on failure is that same one that `fopen` itself returns on failure: `NULL`. Furthermore, `open_file` isn't in a position to do any cleanup if `fopen` fails; `main` will need to skip that file completely and not try to read it or close it, but `open_file` can't do anything about that except warn `main` that something went wrong. So `open_file` ended up being a rather small function. That's all right. It's much easier to merge a small function back into the place where it's called and eliminate the call than it is to separate out a big block of code from one function and put it into a new function. Real programs often have one-line functions.

Our original plan was to have one function to read data in from the file and one to write it out again. But then we realized that since all we ever do with the data we read in is write it out again right away, we should put both things together and have one function (`spew`) which reads the data and writes it out immediately.

I want to push very hard on the idea that designs and programs do not spring forth out of your head fully-formed, like Athena from the forehead of Zeus. You get an idea, and then discover it doesn't work right, and then you fix it, like we did here. And if it doesn't work, the you can try something else.

## 2 Miscellaneous Things We Discussed

Actually writing a program brought up a few issues, some of them about design and others about language features.

### 2.1 Predefined Streams and the Standard Error Output

When you run your program, three stream variables are already set up. `stdin` is a value of type `FILE *`, which represents the standard input; similarly `stdout` represents the standard output. Doing `getc(stdin)` is *exactly* the same as

doing `getchar()`—in fact, `getchar()` is usually a macro which expands to `getc(stdin)`. Similarly `printf` is usually a one-line function which calls

```
fprintf(stdout, ...);
```

to do the real work.

The third predefined stream is called `stderr`, which is short for *standard error output*. It's pretty much equivalent to `stdout`—it's normally attached to the screen, so that things you write to `stderr` appear on the screen. But it's not identical with `stdout`, and if the user redirects the standard output into a file by putting `> file` on the command line, the standard error is *not* redirected—it stays attached to the screen.

`stderr` is there so that you have a place to write error messages. If you used `printf` to write your error messages, they'd go onto the standard output, and might wind up in a file and never be seen. But if you write them with

```
fprintf(stderr, "Error in line %d.\n", lineno);
```

instead, they go onto the screen no matter where the regular output is going, and the user can see them.

## 2.2 `continue`

`continue` is covered in yesterday's notes.

It may seem like I keep pulling these things out of my hat, but really we're near the end. The only control structures we haven't seen yet are `switch` and `goto`. We won't do `goto`. I kept thinking I would bring up `switch` when we needed it (like `continue`), but it just hasn't come up yet.

## 2.3 Don't Ring the Bell

If you ring the bell on the computer, everyone in the room hears it.

Therefore, the question you have to ask yourself before writing a program that rings a bell to signal a certain event is, "Is this event so important that everyone in the room needs to know about it?"

Usually, the answer is 'no'.

## 2.4 Don't Worry About the Output Device

One person said we should read the input and write the output in 80-character chunks, because the screen is probably 80 characters wide.

This is silly. The input might be something like:

```
short
words
stars
groak
fubar
flesh
```

in which case the fact that the screen is 80 characters wide is completely irrelevant—the lines in the input are only 5 characters each. On the other hand, maybe the lines in the input are very long, 2000 or 300 characters.

The specification for the program calls for "Write the data to the standard output."

Whenever you start thinking about things like the screen width, you have to step back and ask if the screen width is really relevant. The way to ask this is to say to yourself, 'Does it make sense to run my program on a terminal that prints the output on paper?' In this case the answer is 'yes'. Sure, you might want to type out the contents of some files, even if you're typing out on paper. Ask yourself 'Would it make sense to run my program if the screen were 1,065 characters wide?' Sure, why not? We're just typing out files.

Even though *our* displays are 80 characters wide, it's not relevant to the program. So don't put it in.