Lecture 12

CSE 110

20 July 1992

Today we'll cover the things that you still don't know that you need to know in order to do the assignment.

1 The `NULL` Pointer

For each pointer type, there is one special value of that type which represents a pointer which does not point anywhere at all. This value is called the `NULL`*pointer*. The `NULL` pointer has one and only one interesting property: If x is an object of type <foo>, and therefore `&`x is pointer to the object x, with type <pointer to foo>, then the value of `&`x is *not* the `NULL` pointer.

The way you represent the `NULL` pointer in your code is by writing `NULL`. `NULL` is actually a macro, but we won't discuss what it's a macro for until later, because it's confusing.

Caution: To not confuse the `NULL` pointer with the `NUL` character. The `NULL` pointer is a pointer, which happens not to point anywhere. The `NUL` character is a character, which is conventionally used to mark the end of a string. It's unfortunate that the names are so similar, but the two things have nothing at all to do with one another.

Many functions which return pointer values return the `NULL` pointer to indicate that something went wrong. For example, you might have a function, `malloc`, which takes an argument `n`, somehow finds `n` contiguous bytes of space on the backboard, reserves them, and returns a pointer to the first byte. `malloc` could return the `NULL` pointer to indicate that there weren't `n` free bytes of space left on the blackboard. You could check the return value from `malloc` to see if you had run out of space:

```
char *buf;
buf = malloc(1000);
if (buf == NULL) {
  printf("Out of memory.\n");
  abort();
}
. . .
```

You know that if `malloc` succeeded in finding the memory it was looking for and returned a pointer to that memory, the value of that pointer would be different from `NULL`. If the return value compares equal to `NULL`, we know that `malloc` didn't succeed.

2 Input Sources and Output Sinks

When you call an input function such as `scanf` or `getchar`, the input comes from a place called the *standard input*. Normally, the standard input it attached to the keyboard so that `scanf` and tt getchar read from the keyboard.

If you run your program under MS-DOS, you can arrange to have the standard input connected somewhere else, such as to a file. For example, to run the command 'foo' with the standard input connected to a file, you enter the command line `foo < input_file`. When `foo` calls `scanf` or `getchar`, the data that `scanf` or `getchar` reads comes from the file `input_file` instead of from the keyboard. This is awfully handy—it means your program doesn't have to know anything about files in order to operate on files.

Similarly, `printf` sends its output to the *standard output*, which is normally attached to the screen so that `printf`'s output appears on the screen. But you can redirect the standard output to a file also: `foo > output_file` attaches `foo`'s standard output to the file `output_file`, and everything that `printf` writes will go into the file instead of to the screen.

Your program is completely incognizant that anything different is happening when it gets run with its standard input or output redirected, and that's good, because it means you don't have to write any extra code to handle it.

2.1 A File Copy Utility

This trivial program copies data from the standard input to the standard output: (`putchar`'s argmuent is a character, which it writes out on the standard output; `putchar(c)` is identical to `printf("%c", c)`.)

```
#include <stdio.h>

int main(void)
{
  int c;

  while ((c = getchar()) != EOF)
    putchar(c);
  return 0;
}
```

Let's call it `scopy`, for 'simple copy'. If you run `scopy`, it echoes everything you type back at you, and perhaps that doesn't sem too useful. But you can use it to copy files: `scopy < source_file > destination_file` copies the data in `source_file` into `destination_file`.

3 Operating on a Particular File

Sometimes, though, you want to read data from or write data to a particular file. For example, the compiler needs to write its output into a file with a particular name. How do we do that?

3.1 Opening a File

First you have to ask the operating system to *open* the file for you. Opening a file means that you notify the operating system that you want to use the file. The operating system checks to make sure that the file you've named exists, and that you have permission from the file's owner to read or write the file. It sets up variables in its own blackboard space that keep track of how much data you've read from the file and what part of the file the next byte is supposed to come from. Sometimes it does other things too—on UNIX the operating system arranges that if someone erases a file that you're reading, the file doesn't actually go away until you're done.

The way you open a file is with the `fopen` function. `fopen` accepts two arguments: The first a string containing the name of the file you want to open, and the second is a string that says whether you want to read the file or write it. For example:

```
fopen("cse110.log", "r");
```

says to open the file `cse110.log` for reading. (`r` means 'read'.)

`fopen`'s return value is a peculiar type: It's a pointer to an object called a `FILE`, which contains some data from the file, information about whether you've reached `EOF` in the file, and other things that the standard I/O functions need to know to work properly. You need this `FILE *` value later on to tell the computer what file you want to read or write from. In some sense, the `FILE *` 'represents' the file.

`FILE *` values are often called *streams*.

The prototype for `fopen` looks like this:

```
FILE * fopen(char *filename, char *type);
```

If `fopen` can't open the file for some reason, it returns the `NULL` pointer.

3.2 Reading from a File with `getc`

Once you've got the file open, you can operate on it with functions that are very much like the ones you're used to. For example, `getc` is a function which takes one argument, a `FILE *`. It is exactly like `getchar` except that it reads its character from the source represented by the `FILE *` that is its argument, instead of from the standard input. Here's code to print out the contents of the file `mydata.txt` onto the standard output:

```
#include <stdio.h>
#define FILENAME "mydata.txt"

int main(void)
{
  FILE * the_file;
  int c;

  the_file = fopen(FILENAME, "r");
  if (the_file == NULL) {
    printf("Couldn't open the file %s.\n", FILENAME);
    return 1;
  }

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

  return 0;
}
```

Actually we left out a detail here: `getc` and `getchar` return `EOF` for two reasons: because there was no more data for them to read, or because there was some kind of error in reading the data. (For example, the disk failed in the middle.) We really should have checked whether the `getc` above was returning `EOF` for end-of-file or for an error.

3.3 Minor Standard I/O Functions

The function `ferror` takes a stream as its argument and returns **true** if there's been an error on the stream, **false** otherwise. Similarly, the function `feof` takes a stream as its argument and returns **true** if you've tried to read beyond the end of the data on that stream, and **false** otherwise.

We should replace the `reutrn 0;` in the code above with

```
  if (ferror(the_file)) {
    printf("Read error on file %s.\n", FILENAME);
    return 1;
  } else {
    return 0;
  }
```

to detect and report read errors while reading the file.

3.4 Stream Versions of `printf` and `scanf`

The funciton `fprintf` is just like `printf`, except it has an extra argmuent: The first argument to `fprintf` is a stream to write its output to. The second argument is a format string just like `printf`'s first argument, and the remaining arguments are values to fill into the conversions in the format string, just like `printf`'s remaining arguments. So, for example,

```
fprintf(the_file, "The value of %s is %d.\n", "fooey", fooey);
```

is exactly like

```
printf("The value of %s is %d.\n", "fooey", fooey);
```

except that the `fprintf` writes the output to the file represented by `the_file`, while the `printf` writes it to the standard output.

Similarly, there is an `fscanf` function, which is just like `scanf`. Similarly, `getchar` has `getc` and `putchar` has `putc`.

3.5 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)`.

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

3.6 Closing a File

When you're done with a file, you must *close* it. This tells the operating system that you are done using the file. That way the operating system knows that it can forget all the details about the file that it was keeping track of for you, such as how much data you'd read out of it.

To close a stream, you call `fclose`. `fclose`'s argument is the stream you want to close. Once a stream is closed, you can't read from it or write to it any more. `fclose` returns 0 if it succeeds and `EOF` if it fails.

4 Command-Line Arguments

When you run a program from MS-DOS, you can give it argmuents. For example, when you run the `copy` command, you give it arguments that say what files you want copied and where you want them copied to. These command-line arguments get passed in to `main` when the operating system calls `main`.

If we were writing `copy` in C, we'd need some way to examine the command-line argumetns so that we'd know which files to copy.

4.1 `main`'s Header

There are exactly two legal ways to write `main`'s header.

```
int main(void)
```

we already know about—it means we're going to ignore the command-line arguments. The other header `main` can have is

```
int main(int argc, char **argv)
```

When the program gets run, the operating system collects the arguments together. Each argument gets put into a `NUL`-terminated array of characters.

The operating system gets the address of each of these arrays. These addresses have type <pointer to `char`>. It takes the addresses and assembles them into another array, an <array of ¶>2`char`. This is called the *argument vector*.

The operating system then takes the address of the first element in the argument vector. The first element is a <pointer to `char`>, and so the address of

the first element has type <pointer to ¶>2`char`. Theoperating system arranges that this value, the address of the argument vector, gets passed to `main` as its second argument, which is conventionally called `argv` for 'argument vector', even though it's really a pointer to the argument vector itself.

`main` has the usual problem: It's got `argv`, a pointer to the first element of an array, so it can find the elements of the array with no problem. But how does it know when to stop?

`main`'s first argument is the *argument count*, conventionally called `argc`. It's an <`int`>. It says how many arguments there are and therefore how long the argument vector is.

4.2 A Thousand Words about `argv` and `argc`

Here's a picture:

4.3 `argc` and `argv` in Practice

It takes a while to get comfortable enough with pointers to understand what's going on with `argv`. In the meantime, just remember this: `argv[0]` is a string which contains the program's first argument. `argv[1]` is a string which contains the program's second argument, and so on, up to `argv[argc-1]`, which is a string which contains the program's last argmuent.

4.4 The `echo` Program

Here's a program called `echo`, which prints out its command-line arguments onto the standard output:

```
#include <stdio.h>

int main(int argc, char **argv)
{
  int i;

  for (i=0; i<argc; i++)
    printf("%s ", argv[i]);

  printf("\n");
  return 0;
}
```

4.5 The `type` Program

Here's a program which types out the contents of the files named on its command line. It's like the MS-DOS `type` command.

```
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++ ) {

    /* 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);

}
```

Some notes: We used `continue`, which we haven't seen before. When the computer encounters a `continue` statement, it immediately starts the next iteration of the smallest loop it's in. Inside of `while` or `do–while` loop, `continue` skips the rest of the loop body and jumps right to the test. In a `for` statement, `continue` is the same, except it evaluates the update expression before skipping to the test. We use it here becauseif we can't open a file, we don't want to bother with the rest of the loop, which is for reading a file; we want to go on and try the next file immediately. `continue` starts the next pass thorugh the loop immediately, without executing the rest of the loop body.

We use the variable `num_errors` to keep track of the number of problems we've had with the files so far, and return it when we're done. Note that this is consistent with the convention of returning 0 if your program was successful and nonzero if it encountered errors.

`argv[0]` conventionally contains the name of the command that's being run—in this case, `type`. If we entered the command

```
type foo bar baz
```

to type out the files `foo`, `bar`, and `baz`, then `argv[0]` would be `type`, and `argv[1]` would be `foo`. That's why we start opening files with the one named in `argv[1]`.

Note that we're always careful to close files with `fclose` when we're done. most systems enforce a limit on the number of files you can have open at once, so we need to recycle our streams when we're done using them.

Notice that we printed out error messages to the standard error stream instead of to the standard output; that's so the error messages don't get mixed up with the output if the user redirectes the standard output into a file.