Pointer Stew

This is a puzzle that uses pointers and arrays in a complex manner. If you completely understand how this works, you definitely know your C pointers and arrays.

You can follow this by looking at the slides in pointerstew-slides.pdf.

The Program

Line numbers are included for reference; they don’t appear in the source code, of course.

```c
#include <stdio.h>
char *c[] = {
    "ENTER",
    "NEW",
    "POINT",
    "FIRST"
};
char **cp[] = { c+3, c+2, c+1, c }; char ***cpp = cp;
int main(void)
{
    printf("%s", **++ cpp);
    printf("%s ", *--*++ cpp +3);
    printf("%s", *cpp[-2]+3);
    printf("%s
", cpp[-1][-1]+1);
    return (0);
}
```

Analysis

Slide 1 is a graphical representation of the initial values of variables and pointers. What follows begins at line 12.

**Line 12: *++cpp**

Here, cp points to cp. As cp is an array of pointers to pointers to characters, the “++” changes cpp to point to cp + 1 (see slide 2). Then the first dereference (“*”) is to c + 2 (see slide 3), and the second dereference (“*”) is to *(c + 2), or c[2] (see slide 4). When printed, the printf dereferences the argument, which is c[2], printing the string that c[2] points to, which is “POINT” (see slide 5).

So the printf on line 12 prints the string POINT with no trailing newline.

After this, cp points to cp + 1. The other variables are unchanged. Slide 6 shows this configuration.

**Line 13: *cpp[-2]+3**

First, we apply the rules of precedence to parenthesize this expression. This produces “*(++(c[-2]))+3”.

Now, cp points to cp + 1. After applying the “++” operator, cpp points to cp + 2 (see slide 7). Then the first dereference (“*”) is to c + 1, and applying the decrement operator “--” changes the entry in the location cp + 2 to be c + 1 - 1, or c (see slide 8). The second dereference (“*”) thus is *c, or c[0], or the string “ENTER”. Adding 3 to this value takes us to c[0] + 3, which is the string “ER” (see slide 9).

So the printf on line 13 prints the string ER with a trailing blank and no trailing newline.

After this, cpp points to cp + 2 and cp[2] points to c. The other variables are unchanged. Slide 10 shows this configuration.

**Line 14: *cp[-2] +3**

Again, we fully parenthesize this to get *(cp[-2])+3.

As cp points to cp + 2, the dereference “*cp[-2]” is to *(cp + 2 - 2), or *cp (see slide 11), or c + 3. Then the dereference “*” takes us to *(c + 3) (see slide 12), or c[3], or the string “FIRST”. Adding 3 to this takes us to c[3] + 3, or which is the string “ST” (see slide 13).

So the printf on line 14 prints the string ST with no trailing newline. Slide 14 shows the configuration after this line.
Line 15: cpp[1][-1]+1

As cpp still points to cp + 2, the dereference “cpp[-1]” is to *(cp + 2 - 1), or *(cp+1) (see slide 15), or c + 2. Then the next “[−1]” takes us to *(c + 2 - 1), or *(c + 1), or c[1] (see slide 16), or the string “NEW”. Adding 1 to this takes us to c[1] + 1, or which is the string “EW” (see slide 17).

So the printf on line 15 prints the string EW with a trailing newline.

Result

So the result of this program is the line

POINTER STEW

Credit

This problem is from Alan Feuer’s excellent book The C Puzzle Book (Addison-Wesley Professional, Boston, MA; ©1998; ISBN 078-5342604610). This document has a slightly modified version by Matt Bishop. Only changes necessary to get it to compile without warnings were made. The C code analyzed above is as in the original.