**"The C Programming Language", 2nd edition, Kernighan and Ritchie**

**Answer to Exercise 1-1**

*Run the "hello, world" program on your system. Experiment with leaving out parts of the program, to see what error messages you get.*   
  
Murphy's Law dictates that there is no single correct answer to the very first exercise in the book. Oh well. Here's a "hello world" program:

**#include <stdio.h>**

**int** main(**void**)

{

printf("hello, world\n");

**return** **0**;

}

As you can see, I've added a return statement, because main always returns int, and it's good style to show this explicitly.

**Answer to Exercise 1-2**

*Experiment to find out what happens when printf 's argument string contains \c, where c is some character not listed above.*   
  
By 'above', the question is referring to:   
\n (newline)   
\t (tab)   
\b (backspace)   
\" (double quote)   
\\ (backslash) We have to tread carefully here, because using a non-specified escape sequence invokes [undefined behaviour](file:///D:\Study\计算机\C程序设计语言（第二版，中英文版%20习题解答，Kernighan%20著）\练习答案\undefined.html). The following program attempts to demonstrate all the legal escape sequences, not including the ones already shown (except \n , which I actually need in the program), and not including hexadecimal and octal escape sequences.

**#include <stdio.h>**

**int** main(**void**)

{

printf("Audible or visual alert. \a\n");

printf("Form feed. \f\n");

printf("This escape, \r, moves the active position to the initial position of the current line.\n");

printf("Vertical tab \v is tricky, as its behaviour is unspecified under certain conditions.\n");

**return** **0**;

}

**Answer to Exercise 1-3**

*Modify the temperature conversion program to print a heading above the table.*

**#include <stdio.h>**

**int** main(**void**)

{

**float** fahr, celsius;

**int** lower, upper, step;

lower = **0**;

upper = **300**;

step = **20**;

printf("F C\n\n");

fahr = lower;

**while**(fahr <= upper)

{

celsius = (**5.0** / **9.0**) \* (fahr - **32.0**);

printf("%3.0f %6.1f\n", fahr, celsius);

fahr = fahr + step;

}

**return** **0**;

}

**Answer to Exercise 1-4**

*Write a program to print the corresponding Celsius to Fahrenheit table.*

**#include <stdio.h>**

**int** main(**void**)

{

**float** fahr, celsius;

**int** lower, upper, step;

lower = **0**;

upper = **300**;

step = **20**;

printf("C F\n\n");

celsius = lower;

**while**(celsius <= upper)

{

fahr = (**9.0**/**5.0**) \* celsius + **32.0**;

printf("%3.0f %6.1f\n", celsius, fahr);

celsius = celsius + step;

}

**return** **0**;

}

**Answer to Exercise 1-5**

*Modify the temperature conversion program to print the table in reverse order, that is, from 300 degrees to 0.*   
  
This version uses a while loop:

**#include <stdio.h>**

**int** main(**void**)

{

**float** fahr, celsius;

**int** lower, upper, step;

lower = **0**;

upper = **300**;

step = **20**;

printf("C F\n\n");

celsius = upper;

**while**(celsius >= lower)

{

fahr = (**9.0**/**5.0**) \* celsius + **32.0**;

printf("%3.0f %6.1f\n", celsius, fahr);

celsius = celsius - step;

}

**return** **0**;

}

This version uses a for loop:

**#include <stdio.h>**

**int** main(**void**)

{

**float** fahr, celsius;

**int** lower, upper, step;

lower = **0**;

upper = **300**;

step = **20**;

printf("C F\n\n");

**for**(celsius = upper; celsius >= lower; celsius = celsius - step)

{

fahr = (**9.0**/**5.0**) \* celsius + **32.0**;

printf("%3.0f %6.1f\n", celsius, fahr);

}

**return** **0**;

}

Chris Sidi notes that Section 1.3 Has a short For statement example, and "Based on that example, I think the solution to 1.5:   
a) should do fahr to celsius conversion (whereas the solutions on your page do celsius to fahr)   
b) should be similar to the example and as small." He offers this solution:

**#include <stdio.h>**

/\* print Fahrenheit-Celsius table \*/

**int**

main()

{

**int** fahr;

**for** (fahr = **300**; fahr >= **0**; fahr = fahr - **20**)

printf("%3d %6.1f\n", fahr, (**5.0**/**9.0**)\*(fahr-**32**));

**return** **0**;

}

**Answer to Exercise 1-6**

*Verify that the expression getchar() != EOF is 0 or 1.*

/\* This program prompts for input, and then captures a character

\* from the keyboard. If EOF is signalled (typically through a

\* control-D or control-Z character, though not necessarily),

\* the program prints 0. Otherwise, it prints 1.

\*

\* If your input stream is buffered (and it probably is), then

\* you will need to press the ENTER key before the program will

\* respond.

\*/

**#include <stdio.h>**

**int** main(**void**)

{

printf("Press a key. ENTER would be nice :-)\n\n");

printf("The expression getchar() != EOF evaluates to %d\n", getchar() != EOF);

**return** **0**;

}

**Answer to Exercise 1-7**

*Write a program to print the value of EOF .*

**#include <stdio.h>**

**int** main(**void**)

{

printf("The value of EOF is %d\n\n", EOF);

**return** **0**;

}

**Exercise 1-8**

*Write a program to count blanks, tabs, and newlines.*

**#include <stdio.h>**

**int** main(**void**)

{

**int** blanks, tabs, newlines;

**int** c;

**int** done = **0**;

**int** lastchar = **0**;

blanks = **0**;

tabs = **0**;

newlines = **0**;

**while**(done == **0**)

{

c = getchar();

**if**(c == **' '**)

++blanks;

**if**(c == **'\t'**)

++tabs;

**if**(c == **'\n'**)

++newlines;

**if**(c == EOF)

{

**if**(lastchar != **'\n'**)

{

++newlines; /\* this is a bit of a semantic stretch, but it copes

\* with implementations where a text file might not

\* end with a newline. Thanks to Jim Stad for pointing

\* this out.

\*/

}

done = **1**;

}

lastchar = c;

}

printf("Blanks: %d\nTabs: %d\nLines: %d\n", blanks, tabs, newlines);

**return** **0**;

}

**Exercise 1-9**

*Write a program to copy its input to its output, replacing each string of one or more blanks by a single blank.*

**#include <stdio.h>**

**int** main(**void**)

{

**int** c;

**int** inspace;

inspace = **0**;

**while**((c = getchar()) != EOF)

{

**if**(c == **' '**)

{

**if**(inspace == **0**)

{

inspace = **1**;

putchar(c);

}

}

/\* We haven't met 'else' yet, so we have to be a little clumsy \*/

**if**(c != **' '**)

{

inspace = **0**;

putchar(c);

}

}

**return** **0**;

}

Chris Sidi writes: "instead of having an "inspace" boolean, you can keep track of the previous character and see if both the current character and previous character are spaces:"

**#include <stdio.h>**

/\* count lines in input \*/

**int**

main()

{

**int** c, pc; /\* c = character, pc = previous character \*/

/\* set pc to a value that wouldn't match any character, in case

this program is ever modified to get rid of multiples of other

characters \*/

pc = EOF;

**while** ((c = getchar()) != EOF) {

**if** (c == **' '**)

**if** (pc != **' '**) /\* or if (pc != c) \*/

putchar(c);

/\* We haven't met 'else' yet, so we have to be a little clumsy \*/

**if** (c != **' '**)

putchar(c);

pc = c;

}

**return** **0**;

}

Stig writes: "I am hiding behind the fact that break is mentioned in the introduction"!

**#include <stdio.h>**

**int** main(**void**)

{

**int** c;

**while** ((c = getchar()) != EOF) {

**if** (c == **' '**) {

putchar(c);

**while**((c = getchar()) == **' '** && c != EOF)

;

}

**if** (c == EOF)

**break**; /\* the break keyword is mentioned

\* in the introduction...

\* \*/

putchar(c);

}

**return** **0**;

}

**Exercise 1-10**

*Write a program to copy its input to its output, replacing each tab by \t , each backspace by \b , and each backslash by \\ . This makes tabs and backspaces visible in an unambiguous way.*

**Category 0**

Gregory Pietsch pointed out that my solution was actually Category 1. He was quite right. Better still, he was kind enough to submit a Category 0 solution himself. Here it is:

/\* Gregory Pietsch <gkp1@flash.net> \*/

/\*

\* Here's my attempt at a Category 0 version of 1-10.

\*

\* Gregory Pietsch

\*/

**#include <stdio.h>**

**int** main()

{

**int** c, d;

**while** ( (c=getchar()) != EOF) {

d = **0**;

**if** (c == **'\\'**) {

putchar(**'\\'**);

putchar(**'\\'**);

d = **1**;

}

**if** (c == **'\t'**) {

putchar(**'\\'**);

putchar(**'t'**);

d = **1**;

}

**if** (c == **'\b'**) {

putchar(**'\\'**);

putchar(**'b'**);

d = **1**;

}

**if** (d == **0**)

putchar(c);

}

**return** **0**;

}

**Category 1**

This solution, which I wrote myself, is the sadly discredited Cat 0 answer which has found a new lease of life in Category 1.

**#include <stdio.h>**

**#define ESC\_CHAR '\\'**

**int** main(**void**)

{

**int** c;

**while**((c = getchar()) != EOF)

{

**switch**(c)

{

**case** **'\b'**:

/\* The OS on which I tested this (NT) intercepts \b characters. \*/

putchar(ESC\_CHAR);

putchar(**'b'**);

**break**;

**case** **'\t'**:

putchar(ESC\_CHAR);

putchar(**'t'**);

**break**;

**case** ESC\_CHAR:

putchar(ESC\_CHAR);

putchar(ESC\_CHAR);

**break**;

**default**:

putchar(c);

**break**;

}

}

**return** **0**;

}

**Exercise 1-11**

*How would you test the word count program? What kinds of input are most likely to uncover bugs if there are any?*   
  
It sounds like they are really trying to get the programmers to learn how to do a unit test. I would submit the following:   
  
0. input file contains zero words   
1. input file contains 1 enormous word without any newlines   
2. input file contains all white space without newlines   
3. input file contains 66000 newlines   
4. input file contains word/{huge sequence of whitespace of different kinds}/word   
5. input file contains 66000 single letter words, 66 to the line   
6. input file contains 66000 words without any newlines   
7. input file is /usr/dict contents (or equivalent)   
8. input file is full collection of moby words   
9. input file is binary (e.g. its own executable)   
10. input file is /dev/nul (or equivalent)   
  
66000 is chosen to check for integral overflow on small integer machines.

Dann suggests a followup exercise 1-11a: write a program to generate inputs (0,1,2,3,4,5,6)   
  
  
  
I guess it was inevitable that I'd receive a solution for this followup exercise! Here is Gregory Pietsch's program to generate Dann's suggested inputs:

**#include <assert.h>**

**#include <stdio.h>**

**int** main(**void**)

{

FILE \*f;

**unsigned** **long** i;

**static** **char** \*ws = " \f\t\v";

**static** **char** \*al = "abcdefghijklmnopqrstuvwxyz";

**static** **char** \*i5 = "a b c d e f g h i j k l m "

"n o p q r s t u v w x y z "

"a b c d e f g h i j k l m "

"n o p q r s t u v w x y z "

"a b c d e f g h i j k l m "

"n\n";

/\* Generate the following: \*/

/\* 0. input file contains zero words \*/

f = fopen("test0", "w");

assert(f != NULL);

fclose(f);

/\* 1. input file contains 1 enormous word without any newlines \*/

f = fopen("test1", "w");

assert(f != NULL);

**for** (i = **0**; i < ((**66000**ul / **26**) + **1**); i++)

fputs(al, f);

fclose(f);

/\* 2. input file contains all white space without newlines \*/

f = fopen("test2", "w");

assert(f != NULL);

**for** (i = **0**; i < ((**66000**ul / **4**) + **1**); i++)

fputs(ws, f);

fclose(f);

/\* 3. input file contains 66000 newlines \*/

f = fopen("test3", "w");

assert(f != NULL);

**for** (i = **0**; i < **66000**; i++)

fputc(**'\n'**, f);

fclose(f);

/\* 4. input file contains word/

\* {huge sequence of whitespace of different kinds}

\* /word

\*/

f = fopen("test4", "w");

assert(f != NULL);

fputs("word", f);

**for** (i = **0**; i < ((**66000**ul / **26**) + **1**); i++)

fputs(ws, f);

fputs("word", f);

fclose(f);

/\* 5. input file contains 66000 single letter words,

\* 66 to the line

\*/

f = fopen("test5", "w");

assert(f != NULL);

**for** (i = **0**; i < **1000**; i++)

fputs(i5, f);

fclose(f);

/\* 6. input file contains 66000 words without any newlines \*/

f = fopen("test6", "w");

assert(f != NULL);

**for** (i = **0**; i < **66000**; i++)

fputs("word ", f);

fclose(f);

**return** **0**;

}

**Exercise 1-12**

Write a program that prints its input one word per line.

**#include <stdio.h>**

**int** main(**void**)

{

**int** c;

**int** inspace;

inspace = **0**;

**while**((c = getchar()) != EOF)

{

**if**(c == **' '** || c == **'\t'** || c == **'\n'**)

{

**if**(inspace == **0**)

{

inspace = **1**;

putchar(**'\n'**);

}

/\* else, don't print anything \*/

}

**else**

{

inspace = **0**;

putchar(c);

}

}

**return** **0**;

}

**Exercise 1-13**

*Write a program to print a histogram of the lengths of words in its input. It is easy to draw the histogram with the bars horizontal; a vertical orientation is more challenging.*

/\* This program was the subject of a thread in comp.lang.c, because of the way it handled EOF.

\* The complaint was that, in the event of a text file's last line not ending with a newline,

\* this program would not count the last word. I objected somewhat to this complaint, on the

\* grounds that "if it hasn't got a newline at the end of each line, it isn't a text file".

\*

\* These grounds turned out to be incorrect. Whether such a file is a text file turns out to

\* be implementation-defined. I'd had a go at checking my facts, and had - as it turns out -

\* checked the wrong facts! (sigh)

\*

\* It cost me an extra variable. It turned out that the least disturbing way to modify the

\* program (I always look for the least disturbing way) was to replace the traditional

\* while((c = getchar()) != EOF) with an EOF test actually inside the loop body. This meant

\* adding an extra variable, but is undoubtedly worth the cost, because it means the program

\* can now handle other people's text files as well as my own. As Ben Pfaff said at the

\* time, "Be liberal in what you accept, strict in what you produce". Sound advice.

\*

\* The new version has, of course, been tested, and does now accept text files not ending in

\* newlines.

\*

\* I have, of course, regenerated the sample output from this program. Actually, there's no

\* "of course" about it - I nearly forgot.

\*/

**#include <stdio.h>**

**#define MAXWORDLEN 10**

**int** main(**void**)

{

**int** c;

**int** inspace = **0**;

**long** lengtharr[MAXWORDLEN + **1**];

**int** wordlen = **0**;

**int** firstletter = **1**;

**long** thisval = **0**;

**long** maxval = **0**;

**int** thisidx = **0**;

**int** done = **0**;

**for**(thisidx = **0**; thisidx <= MAXWORDLEN; thisidx++)

{

lengtharr[thisidx] = **0**;

}

**while**(done == **0**)

{

c = getchar();

**if**(c == **' '** || c == **'\t'** || c == **'\n'** || c == EOF)

{

**if**(inspace == **0**)

{

firstletter = **0**;

inspace = **1**;

**if**(wordlen <= MAXWORDLEN)

{

**if**(wordlen > **0**)

{

thisval = ++lengtharr[wordlen - **1**];

**if**(thisval > maxval)

{

maxval = thisval;

}

}

}

**else**

{

thisval = ++lengtharr[MAXWORDLEN];

**if**(thisval > maxval)

{

maxval = thisval;

}

}

}

**if**(c == EOF)

{

done = **1**;

}

}

**else**

{

**if**(inspace == **1** || firstletter == **1**)

{

wordlen = **0**;

firstletter = **0**;

inspace = **0**;

}

++wordlen;

}

}

**for**(thisval = maxval; thisval > **0**; thisval--)

{

printf("%4d | ", thisval);

**for**(thisidx = **0**; thisidx <= MAXWORDLEN; thisidx++)

{

**if**(lengtharr[thisidx] >= thisval)

{

printf("\* ");

}

**else**

{

printf(" ");

}

}

printf("\n");

}

printf(" +");

**for**(thisidx = **0**; thisidx <= MAXWORDLEN; thisidx++)

{

printf("---");

}

printf("\n ");

**for**(thisidx = **0**; thisidx < MAXWORDLEN; thisidx++)

{

printf("%2d ", thisidx + **1**);

}

printf(">%d\n", MAXWORDLEN);

**return** **0**;

}

Here's the output of the program when given its own source as input:

113 | \*

112 | \*

111 | \*

110 | \*

109 | \*

108 | \*

107 | \*

106 | \*

105 | \*

104 | \*

103 | \*

102 | \*

101 | \*

100 | \*

99 | \*

98 | \*

97 | \*

96 | \*

95 | \*

94 | \* \*

93 | \* \*

92 | \* \*

91 | \* \*

90 | \* \*

89 | \* \*

88 | \* \*

87 | \* \*

86 | \* \*

85 | \* \*

84 | \* \*

83 | \* \*

82 | \* \*

81 | \* \*

80 | \* \*

79 | \* \*

78 | \* \*

77 | \* \*

76 | \* \*

75 | \* \*

74 | \* \*

73 | \* \*

72 | \* \*

71 | \* \*

70 | \* \*

69 | \* \*

68 | \* \*

67 | \* \*

66 | \* \*

65 | \* \*

64 | \* \*

63 | \* \* \*

62 | \* \* \*

61 | \* \* \*

60 | \* \* \*

59 | \* \* \*

58 | \* \* \*

57 | \* \* \*

56 | \* \* \*

55 | \* \* \*

54 | \* \* \*

53 | \* \* \*

52 | \* \* \* \*

51 | \* \* \* \*

50 | \* \* \* \*

49 | \* \* \* \*

48 | \* \* \* \*

47 | \* \* \* \*

46 | \* \* \* \*

45 | \* \* \* \*

44 | \* \* \* \*

43 | \* \* \* \* \*

42 | \* \* \* \* \*

41 | \* \* \* \* \*

40 | \* \* \* \* \*

39 | \* \* \* \* \*

38 | \* \* \* \* \*

37 | \* \* \* \* \*

36 | \* \* \* \* \*

35 | \* \* \* \* \*

34 | \* \* \* \* \*

33 | \* \* \* \* \*

32 | \* \* \* \* \*

31 | \* \* \* \* \*

30 | \* \* \* \* \* \*

29 | \* \* \* \* \* \*

28 | \* \* \* \* \* \* \*

27 | \* \* \* \* \* \* \*

26 | \* \* \* \* \* \* \*

25 | \* \* \* \* \* \* \* \*

24 | \* \* \* \* \* \* \* \*

23 | \* \* \* \* \* \* \* \*

22 | \* \* \* \* \* \* \* \* \*

21 | \* \* \* \* \* \* \* \* \*

20 | \* \* \* \* \* \* \* \* \*

19 | \* \* \* \* \* \* \* \* \*

18 | \* \* \* \* \* \* \* \* \*

17 | \* \* \* \* \* \* \* \* \*

16 | \* \* \* \* \* \* \* \* \*

15 | \* \* \* \* \* \* \* \* \*

14 | \* \* \* \* \* \* \* \* \* \*

13 | \* \* \* \* \* \* \* \* \* \*

12 | \* \* \* \* \* \* \* \* \* \*

11 | \* \* \* \* \* \* \* \* \* \*

10 | \* \* \* \* \* \* \* \* \* \*

9 | \* \* \* \* \* \* \* \* \* \* \*

8 | \* \* \* \* \* \* \* \* \* \* \*

7 | \* \* \* \* \* \* \* \* \* \* \*

6 | \* \* \* \* \* \* \* \* \* \* \*

5 | \* \* \* \* \* \* \* \* \* \* \*

4 | \* \* \* \* \* \* \* \* \* \* \*

3 | \* \* \* \* \* \* \* \* \* \* \*

2 | \* \* \* \* \* \* \* \* \* \* \*

1 | \* \* \* \* \* \* \* \* \* \* \*

+---------------------------------

1 2 3 4 5 6 7 8 9 10 >10

**Exercise 1-14**

*Write a program to print a histogram of the frequencies of different characters in its input.*   
  
Naturally, I've gone for a vertical orientation to match exercise 13. I had some difficulty ensuring that the printing of the X-axis didn't involve cheating. I wanted to display each character if possible, but that would have meant using isprint(), which we haven't yet met. So I decided to display the value of the character instead. (The results below show the output on an ASCII system - naturally, a run on an EBCDIC machine would give different numbers.) I had to jump through a few hoops to avoid using the % operator which, again, we haven't yet met at this point in the text.

**#include <stdio.h>**

/\* NUM\_CHARS should really be CHAR\_MAX but K&R haven't covered that at this stage in the book \*/

**#define NUM\_CHARS 256**

**int** main(**void**)

{

**int** c;

**long** freqarr[NUM\_CHARS + **1**];

**long** thisval = **0**;

**long** maxval = **0**;

**int** thisidx = **0**;

**for**(thisidx = **0**; thisidx <= NUM\_CHARS; thisidx++)

{

freqarr[thisidx] = **0**;

}

**while**((c = getchar()) != EOF)

{

**if**(c < NUM\_CHARS)

{

thisval = ++freqarr[c];

**if**(thisval > maxval)

{

maxval = thisval;

}

}

**else**

{

thisval = ++freqarr[NUM\_CHARS];

**if**(thisval > maxval)

{

maxval = thisval;

}

}

}

**for**(thisval = maxval; thisval > **0**; thisval--)

{

printf("%4d |", thisval);

**for**(thisidx = **0**; thisidx <= NUM\_CHARS; thisidx++)

{

**if**(freqarr[thisidx] >= thisval)

{

printf("\*");

}

**else** **if**(freqarr[thisidx] > **0**)

{

printf(" ");

}

}

printf("\n");

}

printf(" +");

**for**(thisidx = **0**; thisidx <= NUM\_CHARS; thisidx++)

{

**if**(freqarr[thisidx] > **0**)

{

printf("-");

}

}

printf("\n ");

**for**(thisidx = **0**; thisidx < NUM\_CHARS; thisidx++)

{

**if**(freqarr[thisidx] > **0**)

{

printf("%d", thisidx / **100**);

}

}

printf("\n ");

**for**(thisidx = **0**; thisidx < NUM\_CHARS; thisidx++)

{

**if**(freqarr[thisidx] > **0**)

{

printf("%d", (thisidx - (**100** \* (thisidx / **100**))) / **10** );

}

}

printf("\n ");

**for**(thisidx = **0**; thisidx < NUM\_CHARS; thisidx++)

{

**if**(freqarr[thisidx] > **0**)

{

printf("%d", thisidx - (**10** \* (thisidx / **10**)));

}

}

**if**(freqarr[NUM\_CHARS] > **0**)

{

printf(">%d\n", NUM\_CHARS);

}

printf("\n");

**return** **0**;

}

Here's the output of the program when given its own source as input:

474 | \*

473 | \*

472 | \*

471 | \*

470 | \*

469 | \*

468 | \*

467 | \*

466 | \*

465 | \*

464 | \*

463 | \*

462 | \*

461 | \*

460 | \*

459 | \*

458 | \*

457 | \*

456 | \*

455 | \*

454 | \*

453 | \*

452 | \*

451 | \*

450 | \*

449 | \*

448 | \*

447 | \*

446 | \*

445 | \*

444 | \*

443 | \*

442 | \*

441 | \*

440 | \*

439 | \*

438 | \*

437 | \*

436 | \*

435 | \*

434 | \*

433 | \*

432 | \*

431 | \*

430 | \*

429 | \*

428 | \*

427 | \*

426 | \*

425 | \*

424 | \*

423 | \*

422 | \*

421 | \*

420 | \*

419 | \*

418 | \*

417 | \*

416 | \*

415 | \*

414 | \*

413 | \*

412 | \*

411 | \*

410 | \*

409 | \*

408 | \*

407 | \*

406 | \*

405 | \*

404 | \*

403 | \*

402 | \*

401 | \*

400 | \*

399 | \*

398 | \*

397 | \*

396 | \*

395 | \*

394 | \*

393 | \*

392 | \*

391 | \*

390 | \*

389 | \*

388 | \*

387 | \*

386 | \*

385 | \*

384 | \*

383 | \*

382 | \*

381 | \*

380 | \*

379 | \*

378 | \*

377 | \*

376 | \*

375 | \*

374 | \*

373 | \*

372 | \*

371 | \*

370 | \*

369 | \*

368 | \*

367 | \*

366 | \*

365 | \*

364 | \*

363 | \*

362 | \*

361 | \*

360 | \*

359 | \*

358 | \*

357 | \*

356 | \*

355 | \*

354 | \*

353 | \*

352 | \*

351 | \*

350 | \*

349 | \*

348 | \*

347 | \*

346 | \*

345 | \*

344 | \*

343 | \*

342 | \*

341 | \*

340 | \*

339 | \*

338 | \*

337 | \*

336 | \*

335 | \*

334 | \*

333 | \*

332 | \*

331 | \*

330 | \*

329 | \*

328 | \*

327 | \*

326 | \*

325 | \*

324 | \*

323 | \*

322 | \*

321 | \*

320 | \*

319 | \*

318 | \*

317 | \*

316 | \*

315 | \*

314 | \*

313 | \*

312 | \*

311 | \*

310 | \*

309 | \*

308 | \*

307 | \*

306 | \*

305 | \*

304 | \*

303 | \*

302 | \*

301 | \*

300 | \*

299 | \*

298 | \*

297 | \*

296 | \*

295 | \*

294 | \*

293 | \*

292 | \*

291 | \*

290 | \*

289 | \*

288 | \*

287 | \*

286 | \*

285 | \*

284 | \*

283 | \*

282 | \*

281 | \*

280 | \*

279 | \*

278 | \*

277 | \*

276 | \*

275 | \*

274 | \*

273 | \*

272 | \*

271 | \*

270 | \*

269 | \*

268 | \*

267 | \*

266 | \*

265 | \*

264 | \*

263 | \*

262 | \*

261 | \*

260 | \*

259 | \*

258 | \*

257 | \*

256 | \*

255 | \*

254 | \*

253 | \*

252 | \*

251 | \*

250 | \*

249 | \*

248 | \*

247 | \*

246 | \*

245 | \*

244 | \*

243 | \*

242 | \*

241 | \*

240 | \*

239 | \*

238 | \*

237 | \*

236 | \*

235 | \*

234 | \*

233 | \*

232 | \*

231 | \*

230 | \*

229 | \*

228 | \*

227 | \*

226 | \*

225 | \*

224 | \*

223 | \*

222 | \*

221 | \*

220 | \*

219 | \*

218 | \*

217 | \*

216 | \*

215 | \*

214 | \*

213 | \*

212 | \*

211 | \*

210 | \*

209 | \*

208 | \*

207 | \*

206 | \*

205 | \*

204 | \*

203 | \*

202 | \*

201 | \*

200 | \*

199 | \*

198 | \*

197 | \*

196 | \*

195 | \*

194 | \*

193 | \*

192 | \*

191 | \*

190 | \*

189 | \*

188 | \*

187 | \*

186 | \*

185 | \*

184 | \*

183 | \*

182 | \*

181 | \*

180 | \*

179 | \*

178 | \*

177 | \*

176 | \*

175 | \*

174 | \*

173 | \*

172 | \*

171 | \*

170 | \*

169 | \*

168 | \*

167 | \*

166 | \*

165 | \*

164 | \*

163 | \*

162 | \*

161 | \*

160 | \*

159 | \*

158 | \*

157 | \*

156 | \*

155 | \*

154 | \*

153 | \*

152 | \*

151 | \*

150 | \*

149 | \*

148 | \*

147 | \*

146 | \*

145 | \*

144 | \*

143 | \*

142 | \*

141 | \*

140 | \*

139 | \*

138 | \*

137 | \*

136 | \*

135 | \*

134 | \*

133 | \*

132 | \*

131 | \*

130 | \*

129 | \*

128 | \*

127 | \*

126 | \*

125 | \*

124 | \*

123 | \*

122 | \*

121 | \*

120 | \*

119 | \*

118 | \*

117 | \*

116 | \*

115 | \*

114 | \*

113 | \*

112 | \*

111 | \*

110 | \*

109 | \* \*

108 | \* \*

107 | \* \*

106 | \* \*

105 | \* \*

104 | \* \*

103 | \* \*

102 | \* \*

101 | \* \*

100 | \* \*

99 | \* \*

98 | \* \*

97 |\*\* \*

96 |\*\* \*

95 |\*\* \*

94 |\*\* \*

93 |\*\* \*

92 |\*\* \*

91 |\*\* \*

90 |\*\* \*

89 |\*\* \*

88 |\*\* \*

87 |\*\* \*

86 |\*\* \*

85 |\*\* \*

84 |\*\* \*

83 |\*\* \*

82 |\*\* \*

81 |\*\* \*

80 |\*\* \*

79 |\*\* \*

78 |\*\* \*

77 |\*\* \*

76 |\*\* \*

75 |\*\* \*

74 |\*\* \*

73 |\*\* \*

72 |\*\* \*

71 |\*\* \* \*

70 |\*\* \* \*

69 |\*\* \* \*

68 |\*\* \* \*

67 |\*\* \* \*

66 |\*\* \* \*

65 |\*\* \* \*

64 |\*\* \* \*

63 |\*\* \* \*

62 |\*\* \* \*

61 |\*\* \* \*

60 |\*\* \* \*

59 |\*\* \* \* \*

58 |\*\* \* \* \*

57 |\*\* \* \* \*

56 |\*\* \* \* \*

55 |\*\* \* \* \*

54 |\*\* \* \* \*

53 |\*\* \* \* \*

52 |\*\* \* \* \*

51 |\*\* \*\* \* \*

50 |\*\* \*\* \* \*

49 |\*\* \*\* \*\*\*

48 |\*\* \*\* \*\*\*

47 |\*\* \*\* \*\*\*

46 |\*\* \*\* \*\*\*

45 |\*\* \*\* \*\*\*

44 |\*\* \*\* \*\*\*

43 |\*\* \* \*\* \*\*\*

42 |\*\* \* \* \* \*\* \*\*\*

41 |\*\* \* \* \* \*\* \*\*\*

40 |\*\* \*\* \* \* \* \*\* \*\*\*

39 |\*\* \*\* \* \* \* \* \*\* \*\*\*

38 |\*\* \*\* \* \* \* \* \*\* \*\*\*

37 |\*\* \*\* \* \* \* \* \*\* \*\*\* \*

36 |\*\* \*\* \* \* \* \* \*\* \*\*\* \*

35 |\*\* \*\* \* \* \* \* \*\* \*\*\* \*

34 |\*\* \*\* \* \* \* \* \*\* \*\*\* \*

33 |\*\* \*\* \* \* \* \* \*\* \*\*\* \*

32 |\*\* \*\* \* \* \* \* \*\* \* \*\*\* \*

31 |\*\* \*\* \* \* \* \* \*\* \* \*\*\* \*

30 |\*\* \*\* \* \* \* \* \*\* \* \*\*\* \*

29 |\*\* \*\* \* \* \* \* \*\* \* \*\*\* \*

28 |\*\* \* \*\* \* \* \*\*\* \*\* \* \* \*\*\* \*

27 |\*\* \* \*\* \* \* \* \*\*\* \*\* \* \* \*\*\* \*

26 |\*\* \* \*\* \* \* \* \*\*\* \*\* \* \* \*\*\* \*

25 |\*\* \* \*\* \* \* \* \*\*\* \*\* \* \* \*\*\* \*

24 |\*\* \* \*\* \* \* \* \*\*\* \*\* \* \* \*\*\* \*

23 |\*\* \* \*\* \* \* \* \*\*\* \*\* \* \* \*\*\* \*

22 |\*\* \* \*\* \* \* \* \*\*\* \*\* \* \* \*\*\* \*

21 |\*\* \* \*\* \* \* \* \* \*\*\* \*\* \* \* \*\*\* \* \*

20 |\*\* \* \*\* \* \* \* \* \*\*\* \*\* \* \* \*\*\* \* \* \* \*

19 |\*\* \* \*\* \* \* \* \* \*\*\* \*\* \* \* \*\*\* \* \* \* \*

18 |\*\* \* \*\* \* \* \* \* \* \*\*\* \*\* \* \* \*\*\* \* \* \* \*

17 |\*\* \* \*\* \* \* \* \* \* \*\*\* \*\* \* \* \*\*\* \* \* \* \*

16 |\*\* \* \*\* \* \* \* \* \* \*\*\* \*\* \* \*\* \*\*\* \* \* \* \*

15 |\*\* \* \*\* \* \* \* \* \* \* \* \*\*\* \*\* \* \*\* \*\*\* \* \* \* \*

14 |\*\* \* \*\* \* \* \* \* \*\* \* \* \* \*\* \*\*\* \*\* \* \*\*\* \*\*\* \* \* \* \*

13 |\*\* \* \*\* \* \* \* \* \*\* \* \*\* \*\*\* \*\* \*\*\* \*\* \* \*\*\* \*\*\* \* \* \* \*

12 |\*\* \* \*\* \* \* \* \*\*\*\* \* \*\* \*\*\* \*\* \*\*\* \*\* \* \*\*\* \*\*\* \* \* \* \*

11 |\*\* \* \*\* \* \* \* \*\*\*\* \* \*\* \*\*\* \* \*\*\* \*\*\* \*\* \* \*\*\*\*\*\*\* \* \* \* \*

10 |\*\* \* \*\* \* \* \* \*\*\*\* \* \*\* \*\*\* \* \*\*\* \*\*\* \*\* \* \*\*\*\*\*\*\* \* \* \* \*

9 |\*\* \* \*\* \* \* \* \*\*\*\* \* \*\* \*\*\* \* \*\*\* \*\*\* \*\* \* \*\*\*\*\*\*\* \* \* \* \*

8 |\*\* \* \*\* \* \* \*\*\*\*\*\* \* \*\* \*\*\* \* \*\*\* \*\*\* \*\* \* \*\*\*\*\*\*\* \* \* \* \*

7 |\*\* \* \*\* \* \*\* \*\*\*\*\*\* \* \*\* \*\*\* \* \*\*\* \*\*\*\* \*\* \*\*\*\*\*\*\*\*\* \* \* \* \*

6 |\*\* \* \*\* \* \*\*\* \*\*\*\*\*\* \* \*\* \*\*\* \*\*\*\*\* \*\*\*\* \*\* \*\*\*\*\*\*\*\*\* \* \* \* \*

5 |\*\* \* \* \*\*\*\*\*\* \*\*\* \*\*\*\*\*\* \* \*\* \*\*\* \*\*\*\*\* \*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\* \* \* \* \*

4 |\*\* \* \* \*\*\*\*\*\* \*\*\* \*\*\*\*\*\* \* \*\* \*\*\* \*\*\*\*\* \*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\*\* \* \* \*

3 |\*\* \* \* \*\*\*\*\*\* \*\*\* \*\*\*\*\*\* \* \*\* \*\*\* \*\*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\*\* \* \* \*

2 |\*\* \*\*\* \*\*\*\*\*\* \*\*\* \*\*\*\*\*\* \* \*\* \*\*\* \*\*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\*\* \* \* \*

1 |\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

+----------------------------------------------------------------------

0000000000000000000000000000000000000000000000111111111111111111111111

1333333344444444445555566666677777788889999999000000000111111111122222

0234578901234567890234901257902578923581235789012345789012345678901345

**Answer to Exercise 1-15, page 27**

*Rewrite the temperature conversion program of Section 1.2 to use a function for conversion.*

**#include <stdio.h>**

**float** FtoC(**float** f)

{

**float** c;

c = (**5.0** / **9.0**) \* (f - **32.0**);

**return** c;

}

**int** main(**void**)

{

**float** fahr, celsius;

**int** lower, upper, step;

lower = **0**;

upper = **300**;

step = **20**;

printf("F C\n\n");

fahr = lower;

**while**(fahr <= upper)

{

celsius = FtoC(fahr);

printf("%3.0f %6.1f\n", fahr, celsius);

fahr = fahr + step;

}

**return** **0**;

}

**Answer to Exercise 1-16, page 30**

*Revise the main routine of the longest-line program so it will correctly print the length of arbitrarily long input lines, and as much as possible of the text.*

/\* This is the first program exercise where the spec isn't entirely

\* clear. The spec says, 'Revise the main routine', but the true

\* length of an input line can only be determined by modifying

\* getline. So that's what we'll do. getline will now return the

\* actual length of the line rather than the number of characters

\* read into the array passed to it.

\*/

**#include <stdio.h>**

**#define MAXLINE 1000 /\* maximum input line size \*/**

**int** getline(**char** line[], **int** maxline);

**void** copy(**char** to[], **char** from[]);

/\* print longest input line \*/

**int** main(**void**)

{

**int** len; /\* current line length \*/

**int** max; /\* maximum length seen so far \*/

**char** line[MAXLINE]; /\* current input line \*/

**char** longest[MAXLINE]; /\* longest line saved here \*/

max = **0**;

**while**((len = getline(line, MAXLINE)) > **0**)

{

printf("%d: %s", len, line);

**if**(len > max)

{

max = len;

copy(longest, line);

}

}

**if**(max > **0**)

{

printf("Longest is %d characters:\n%s", max, longest);

}

printf("\n");

**return** **0**;

}

/\* getline: read a line into s, return length \*/

**int** getline(**char** s[], **int** lim)

{

**int** c, i, j;

**for**(i = **0**, j = **0**; (c = getchar())!=EOF && c != **'\n'**; ++i)

{

**if**(i < lim - **1**)

{

s[j++] = c;

}

}

**if**(c == **'\n'**)

{

**if**(i <= lim - **1**)

{

s[j++] = c;

}

++i;

}

s[j] = **'\0'**;

**return** i;

}

/\* copy: copy 'from' into 'to'; assume 'to' is big enough \*/

**void** copy(**char** to[], **char** from[])

{

**int** i;

i = **0**;

**while**((to[i] = from[i]) != **'\0'**)

{

++i;

}

}

Chris Sidi, however, was not convinced - he thought this answer was "too easy", so he checked with bwk, who agreed. Chris writes: "Looks like Mr. Kernighan meant for "main routine" in Exercise 1-16 to refer to function main(), saying your solution of modifying getline() is "too easy." :) (Though I think your solution shouldn't be removed from the Answers web site, just complimented with another one that only modifies main())"   
  
Cue Mr "386sx", riding to the rescue on a white horse...

/\* Exercise 1-16 \*/

**#include <stdio.h>**

**#define MAXLINE 20**

**int** getline(**char** s[], **int** lim);

**void** copy(**char** to[], **char** from[]);

**int** main(**void**)

{

**char** line[MAXLINE];

**char** longest[MAXLINE];

**char** temp[MAXLINE];

**int** len, max, prevmax, getmore;

max = prevmax = getmore = **0**;

**while**((len = getline(line, MAXLINE)) > **0**)

{

**if**(line[len - **1**] != **'\n'**)

{

**if**(getmore == **0**)

copy(temp, line);

prevmax += len;

**if**(max < prevmax)

max = prevmax;

getmore = **1**;

}

**else**

{

**if**(getmore == **1**)

{

**if**(max < prevmax + len)

{

max = prevmax + len;

copy(longest, temp);

longest[MAXLINE - **2**] = **'\n'**;

}

getmore = **0**;

}

**else** **if**(max < len)

{

max = len;

copy(longest, line);

}

prevmax = **0**;

}

}

**if**(max > **0**)

{

printf("%s", longest);

printf("len = %d\n", max);

}

**return** **0**;

}

**int** getline(**char** s[], **int** lim)

{

**int** c, i;

**for**(i = **0**;

i < lim - **1** && ((c = getchar()) != EOF && c != **'\n'**);

++i)

s[i] = c;

**if**(c == **'\n'**)

{

s[i] = c;

++i;

}

**else** **if**(c == EOF && i > **0**)

{

/\* gotta do something about no newline preceding EOF \*/

s[i] = **'\n'**;

++i;

}

s[i] = **'\0'**;

**return** i;

}

**void** copy(**char** to[], **char** from[])

{

**int** i;

i = **0**;

**while**((to[i] = from[i]) != **'\0'**)

++i;

}

**Answer to Exercise 1-17, page 31**

*Write a program to print all input lines that are longer than 80 characters.*

**#include <stdio.h>**

**#define MINLENGTH 81**

**int** readbuff(**char** \*buffer) {

size\_t i=**0**;

**int** c;

**while** (i < MINLENGTH) {

c = getchar();

**if** (c == EOF) **return** -**1**;

**if** (c == **'\n'**) **return** **0**;

buffer[i++] = c;

}

**return** **1**;

}

**int** copyline(**char** \*buffer) {

size\_t i;

**int** c;

**int** status = **1**;

**for**(i=**0**; i<MINLENGTH; i++)

putchar(buffer[i]);

**while**(status == **1**) {

c = getchar();

**if** (c == EOF)

status = -**1**;

**else** **if** (c == **'\n'**)

status = **0**;

**else**

putchar(c);

}

putchar(**'\n'**);

**return** status;

}

**int** main(**void**) {

**char** buffer[MINLENGTH];

**int** status = **0**;

**while** (status != -**1**) {

status = readbuff(buffer);

**if** (status == **1**)

status = copyline(buffer);

}

**return** **0**;

}

**Answer to Exercise 1-18, page 31**

*Write a program to remove all trailing blanks and tabs from each line of input, and to delete entirely blank lines.*

/\* K&R2 1-18 p31: Write a program to remove trailing blanks and tabs

from each line of input, and to delete entirely blank lines.

The program specification is ambiguous: does "entirely blank lines"

mean lines that contain no characters other than newline, or does

it include lines composed of blanks and tabs followed by newline?

The latter interpretation is taken here.

This implementation does not use any features not introduced in the

first chapter of K&R2. As a result, it can't use pointers to

dynamically allocate a buffer to store blanks that it has seen, so

it must limit the number of blanks that are allowed to occur

consecutively. (This is the value of MAXQUEUE, minus one.)

It is intended that this implementation "degrades gracefully."

Even though a particular input might have 1000 or more blanks or

tabs in a row, causing a problem for a single pass, multiple passes

through the file will correct the problem. The program signals the

need for such an additional pass by returning a failure code to the

operating system. (EXIT\_FAILURE isn't mentioned in the first

chapter of K&R, but I'm making an exception here.) \*/

**#include <stdio.h>**

**#include <stdlib.h>**

**#define MAXQUEUE 1001**

**int** advance(**int** pointer)

{

**if** (pointer < MAXQUEUE - **1**)

**return** pointer + **1**;

**else**

**return** **0**;

}

**int** main(**void**)

{

**char** blank[MAXQUEUE];

**int** head, tail;

**int** nonspace;

**int** retval;

**int** c;

retval = nonspace = head = tail = **0**;

**while** ((c = getchar()) != EOF) {

**if** (c == **'\n'**) {

head = tail = **0**;

**if** (nonspace)

putchar(**'\n'**);

nonspace = **0**;

}

**else** **if** (c == **' '** || c == **'\t'**) {

**if** (advance(head) == tail) {

putchar(blank[tail]);

tail = advance(tail);

nonspace = **1**;

retval = EXIT\_FAILURE;

}

blank[head] = c;

head = advance(head);

}

**else** {

**while** (head != tail) {

putchar(blank[tail]);

tail = advance(tail);

}

putchar(c);

nonspace = **1**;

}

}

**return** retval;

}

Chris Sidi writes:

Ben,

I thought your solution to 1-18 was really neat (it didn't occur to me

when I was doing the exercise), the way it degrades gracefully and

multiple passes can get rid of huge blocks of whitespace.

However, if there is a huge block of non-trailing whitespace (eg "A",2000

spaces, "B\n") your program returns an error when there's not a need for

it. And if someone were to use your program till it passes it will loop

infinitely:

$ perl -e 'print "A"," "x2000,"B\n";' > in

$ until ./a.out < in > out; do echo failed, running another pass; cp out

in; done

failed, running another pass

failed, running another pass

failed, running another pass

[snip]

Below I have added a variable spaceJustPrinted to your program and check

to see if the spaces printed early are trailing. I hope you like the

minor improvement. (Though I can understand if you don't give a [1] :))

[1] expletive deleted - RJH.

/\* K&R2 1-18 p31: Write a program to remove trailing blanks and tabs

from each line of input, and to delete entirely blank lines.

The program specification is ambiguous: does "entirely blank lines"

mean lines that contain no characters other than newline, or does

it include lines composed of blanks and tabs followed by newline?

The latter interpretation is taken here.

This implementation does not use any features not introduced in the

first chapter of K&R2. As a result, it can't use pointers to

dynamically allocate a buffer to store blanks that it has seen, so

it must limit the number of blanks that are allowed to occur

consecutively. (This is the value of MAXQUEUE, minus one.)

It is intended that this implementation "degrades gracefully."

Even though a particular input might have 1000 or more trailing

blanks or tabs in a row, causing a problem for a single pass,

multiple passes through the file will correct the problem. The

program signals the need for such an additional pass by returning a

failure code to the operating system. (EXIT\_FAILURE isn't mentioned

in the first chapter of K&R, but I'm making an exception here.) \*/

**#include <stdio.h>**

**#include <stdlib.h>**

**#define MAXQUEUE 1001**

**int** advance(**int** pointer)

{

**if** (pointer < MAXQUEUE - **1**)

**return** pointer + **1**;

**else**

**return** **0**;

}

**int** main(**void**)

{

**char** blank[MAXQUEUE];

**int** head, tail;

**int** nonspace;

**int** retval;

**int** c;

**int** spaceJustPrinted; /\*boolean: was the last character printed whitespace?\*/

retval = spaceJustPrinted = nonspace = head = tail = **0**;

**while** ((c = getchar()) != EOF) {

**if** (c == **'\n'**) {

head = tail = **0**;

**if** (spaceJustPrinted == **1**) /\*if some trailing whitespace was printed...\*/

retval = EXIT\_FAILURE;

**if** (nonspace) {

putchar(**'\n'**);

spaceJustPrinted = **0**; /\* this instruction isn't really necessary since

spaceJustPrinted is only used to determine the

return value, but we'll keep this boolean

truthful \*/

nonspace = **0**; /\* moved inside conditional just to save a needless

assignment \*/

}

}

**else** **if** (c == **' '** || c == **'\t'**) {

**if** (advance(head) == tail) {

putchar(blank[tail]); /\* these whitespace chars being printed early

are only a problem if they are trailing,

which we'll check when we hit a \n or EOF \*/

spaceJustPrinted = **1**;

tail = advance(tail);

nonspace = **1**;

}

blank[head] = c;

head = advance(head);

}

**else** {

**while** (head != tail) {

putchar(blank[tail]);

tail = advance(tail);

}

putchar(c);

spaceJustPrinted = **0**;

nonspace = **1**;

}

}

/\* if the last line wasn't ended with a newline before the EOF,

we'll need to figure out if trailing space was printed here \*/

**if** (spaceJustPrinted == **1**) /\*if some trailing whitespace was printed...\*/

retval = EXIT\_FAILURE;

**return** retval;

}

**Answer to Exercise 1-19, page 31**

*Write a function reverse(s) that reverses the character string s . Use it to write a program that reverses its input a line at a time.*

**#include <stdio.h>**

**#define MAX\_LINE 1024**

**void** discardnewline(**char** s[])

{

**int** i;

**for**(i = **0**; s[i] != **'\0'**; i++)

{

**if**(s[i] == **'\n'**)

s[i] = **'\0'**;

}

}

**int** reverse(**char** s[])

{

**char** ch;

**int** i, j;

**for**(j = **0**; s[j] != **'\0'**; j++)

{

}

--j;

**for**(i = **0**; i < j; i++)

{

ch = s[i];

s[i] = s[j];

s[j] = ch;

--j;

}

**return** **0**;

}

**int** getline(**char** s[], **int** lim)

{

**int** c, i;

**for**(i = **0**; i < lim - **1** && (c = getchar()) != EOF && c != **'\n'**; ++i)

{

s[i] = c;

}

**if**(c == **'\n'**)

{

s[i++] = c;

}

s[i] = **'\0'**;

**return** i;

}

**int** main(**void**)

{

**char** line[MAX\_LINE];

**while**(getline(line, **sizeof** line) > **0**)

{

discardnewline(line);

reverse(line);

printf("%s\n", line);

}

**return** **0**;

}

**Answer to Exercise 1-20, page 34**

Thanks to Rick Dearman for pointing out that my solution used fgets() which has not been introduced by page 34. I've fixed the solution to use K&R's getline() function instead. Further thanks to Roman Yablonovsky who, in Oct 2000, pointed out that the solution was buggy, and hinted at a fix. Basically, the problem he spotted was that my solution failed to keep track of the cumulative effect of multiple tabs in a single line. I've adopted his fix (which was in fact also slightly buggy, but I've fixed that too).   
  
*Write a program detab that replaces tabs in the input with the proper number of blanks to space to the next tab stop. Assume a fixed set of tab stops, say every* n *columns. Should* n *be a variable or a symbolic parameter?*

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <string.h>**

**#define MAX\_BUFFER 1024**

**#define SPACE ' '**

**#define TAB '\t'**

**int** CalculateNumberOfSpaces(**int** Offset, **int** TabSize)

{

**return** TabSize - (Offset % TabSize);

}

/\* K&R's getline() function from p29 \*/

**int** getline(**char** s[], **int** lim)

{

**int** c, i;

**for**(i = **0**; i < lim - **1** && (c = getchar()) != EOF && c != **'\n'**; ++i)

s[i] = c;

**if**(c == **'\n'**)

{

s[i] = c;

++i;

}

s[i] = **'\0'**;

**return** i;

}

**int** main(**void**)

{

**char** Buffer[MAX\_BUFFER];

**int** TabSize = **5**; /\* A good test value \*/

**int** i, j, k, l;

**while**(getline(Buffer, MAX\_BUFFER) > **0**)

{

**for**(i = **0**, l = **0**; Buffer[i] != **'\0'**; i++)

{

**if**(Buffer[i] == TAB)

{

j = CalculateNumberOfSpaces(l, TabSize);

**for**(k = **0**; k < j; k++)

{

putchar(SPACE);

l++;

}

}

**else**

{

putchar(Buffer[i]);

l++;

}

}

}

**return** **0**;

}

In answer to the question about whether *n* should be variable or symbolic, I'm tempted to offer the answer 'yes'. :-) Of course, it should be variable, to allow for modification of the value at runtime, for example via a command line argument, without requiring recompilation.

**Answer to Exercise 1-21, page 34**

*Write a program entab that replaces strings of blanks with the minimum number of tabs and blanks to achieve the same spacing. Use the same stops as for detab . When either a tab or a single blank would suffice to reach a tab stop, which should be given preference?*   
  
Rick Dearman's Cat 0 solution:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

KnR 1-21

--------

Write a program "entab" which replaces strings of

blanks with the minimum number of tabs and blanks

to achieve the same spacing.

Author: Rick Dearman

email: rick@ricken.demon.co.uk

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**#include <stdio.h>**

**#define MAXLINE 1000 /\* max input line size \*/**

**#define TAB2SPACE 4 /\* 4 spaces to a tab \*/**

**char** line[MAXLINE]; /\*current input line\*/

**int** getline(**void**); /\* taken from the KnR book. \*/

**int**

main()

{

**int** i,t;

**int** spacecount,len;

**while** (( len = getline()) > **0** )

{

spacecount = **0**;

**for**( i=**0**; i < len; i++)

{

**if**(line[i] == **' '**)

spacecount++; /\* increment counter for each space \*/

**if**(line[i] != **' '**)

spacecount = **0**; /\* reset counter \*/

**if**(spacecount == TAB2SPACE) /\* Now we have enough spaces

\*\* to replace them with a tab

\*/

{

/\* Because we are removing 4 spaces and

\*\* replacing them with 1 tab we move back

\*\* three chars and replace the ' ' with a \t

\*/

i -= **3**; /\* same as "i = i - 3" \*/

len -= **3**;

line[i] = **'\t'**;

/\* Now move all the char's to the right into the

\*\* places we have removed.

\*/

**for**(t=i+**1**;t<len;t++)

line[t]=line[t+**3**];

/\* Now set the counter back to zero and move the

\*\* end of line back 3 spaces

\*/

spacecount = **0**;

line[len] = **'\0'**;

}

}

printf("%s", line);

}

**return** **0**;

}

/\* getline: specialized version \*/

**int** getline(**void**)

{

**int** c, i;

**extern** **char** line[];

**for** ( i=**0**;i<MAXLINE-**1** && ( c=getchar()) != EOF && c != **'\n'**; ++i)

line[i] = c;

**if**(c == **'\n'**)

{

line[i] = c;

++i;

}

line[i] = **'\0'**;

**return** i;

}

Stefan Farfeleder's Cat 1 solution:

/\* 1-21.c \*/

**#include <stdio.h>**

**#define TABSTOP 4**

**int** main(**void**)

{

size\_t spaces = **0**;

**int** ch;

size\_t x = **0**; /\* position in the line \*/

size\_t tabstop = TABSTOP; /\* get this from the command-line

\* if you want to \*/

**while** ((ch = getchar()) != EOF)

{

**if** (ch == **' '**)

{

spaces++;

}

**else** **if** (spaces == **0**) /\* no space, just printing \*/

{

putchar(ch);

x++;

}

**else** **if** (spaces == **1**) /\* just one space, never print a tab \*/

{

putchar(**' '**);

putchar(ch);

x += **2**;

spaces = **0**;

}

**else**

{

**while** (x / tabstop != (x + spaces) / tabstop)

/\* are the spaces reaching behind the next tabstop ? \*/

{

putchar(**'\t'**);

x++;

spaces--;

**while** (x % tabstop != **0**)

{

x++;

spaces--;

}

}

**while** (spaces > **0**) /\* the remaining ones are real space \*/

{

putchar(**' '**);

x++;

spaces--;

}

putchar(ch); /\* now print the non-space char \*/

x++;

}

**if** (ch == **'\n'**)

{

x = **0**; /\* reset line position \*/

}

}

**return** **0**;

}

**Answer to Exercise 1-22, page 34**

*Write a program to "fold" long input lines into two or more shorter lines after the last non-blank character that occurs before the n -th column of input. Make sure your program does something intelligent with very long lines, and if there are no blanks or tabs before the specified column.*

**Category 1 Solution**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

KnR 1-22

--------

Write a program that wraps very long lines of input

into two or more shorter lines.

Author: Rick Dearman

email: rick@ricken.demon.co.uk

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**#include <stdio.h>**

**#define MAXLINE 1000 /\* max input line size \*/**

**char** line[MAXLINE]; /\*current input line\*/

**int** getline(**void**); /\* taken from the KnR book. \*/

**int**

main()

{

**int** t,len;

**int** location,spaceholder;

**const** **int** FOLDLENGTH=**70**; /\* The max length of a line \*/

**while** (( len = getline()) > **0** )

{

**if**( len < FOLDLENGTH )

{

}

**else**

{

/\* if this is an extra long line then we

\*\* loop through it replacing a space nearest

\*\* to the foldarea with a newline.

\*/

t = **0**;

location = **0**;

**while**(t<len)

{

**if**(line[t] == **' '**)

spaceholder = t;

**if**(location==FOLDLENGTH)

{

line[spaceholder] = **'\n'**;

location = **0**;

}

location++;

t++;

}

}

printf ( "%s", line);

}

**return** **0**;

}

/\* getline: specialized version \*/

**int** getline(**void**)

{

**int** c, i;

**extern** **char** line[];

**for** ( i=**0**;i<MAXLINE-**1** && ( c=getchar()) != EOF && c != **'\n'**; ++i)

line[i] = c;

**if**(c == **'\n'**)

{

line[i] = c;

++i;

}

line[i] = **'\0'**;

**return** i;

}

**Answer to Exercise 1-23, page 34**

*Write a program to remove all comments from a C program. Don't forget to handle quoted strings and character constants properly. C comments do not nest.*   
  
  
This was the first exercise to be posted as a fun "competition" on comp.lang.c, on 1 June 2000. As a result, there was a small flurry of submissions. Not all of them are completely working solutions. See the very end of this page for a test program which breaks most of them. :-)

**Category 0 Solutions**

From Rick Dearman   
  
Now handles "/\* comment in string \*/" correctly, but does not remove the comment from

**return** /\* comment inside return statement \*/ **0**;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

"Write a program to remove all comments from a C program.

Don't forget to handle quoted strings and character

constants properly. C comments do not nest."

Author: Rick Dearman (rick@ricken.demon.co.uk)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**#include <stdio.h>**

**#define MAXLINE 1000 /\* max input line size \*/**

**char** line[MAXLINE]; /\*current input line\*/

**int** getline(**void**); /\* taken from the KnR book. \*/

**int**

main()

{

**int** in\_comment,len;

**int** in\_quote;

**int** t;

in\_comment = in\_quote = t = **0**;

**while** ((len = getline()) > **0** )

{

t=**0**;

**while**(t < len)

{

**if**( line[t] == **'"'**)

in\_quote = **1**;

**if**( ! in\_quote )

{

**if**( line[t] == **'/'** && line[t+**1**] == **'\*'**)

{

t=t+**2**;

in\_comment = **1**;

}

**if**( line[t] == **'\*'** && line[t+**1**] == **'/'**)

{

t=t+**2**;

in\_comment = **0**;

}

**if**(in\_comment == **1**)

{

t++;

}

**else**

{

printf ("%c", line[t]);

t++;

}

}

**else**

{

printf ("%c", line[t]);

t++;

}

}

}

**return** **0**;

}

/\* getline: specialized version \*/

**int** getline(**void**)

{

**int** c, i;

**extern** **char** line[];

**for** ( i=**0**;i<MAXLINE-**1** && ( c=getchar()) != EOF && c != **'\n'**; ++i)

line[i] = c;

**if**(c == **'\n'**)

{

line[i] = c;

++i;

}

line[i] = **'\0'**;

**return** i;

}

From Ben Pfaff   
  
This version is a bugfix for the code var/'\2'

/\* K&R2 1-23: Write a program to remove all comments from a C program.

Don't forget to handle quoted strings and character constants

properly. C comments do not nest.

This solution does not deal with other special cases, such as

trigraphs, line continuation with \, or <> quoting on #include,

since these aren't mentioned up 'til then in K&R2. Perhaps this is

cheating.

Note that this program contains both comments and quoted strings of

text that looks like comments, so running it on itself is a

reasonable test. It also contains examples of a comment that ends

in a star and a comment preceded by a slash. Note that the latter

will break C99 compilers and C89 compilers with // comment

extensions.

Interface: The C source file is read from stdin and the

comment-less output is written to stdout. \*\*/

**#include <stdio.h>**

**int**

main(**void**)

{

**#define PROGRAM 0**

**#define SLASH 1**

**#define COMMENT 2**

**#define STAR 3**

**#define QUOTE 4**

**#define LITERAL 5**

/\* State machine's current state, one of the above values. \*/

**int** state;

/\* If state == QUOTE, then ' or ". Otherwise, undefined. \*/

**int** quote;

/\* Input character. \*/

**int** c;

state = PROGRAM;

**while** ((c = getchar()) != EOF) {

/\* The following cases are in guesstimated order from most common

to least common. \*/

**if** (state == PROGRAM || state == SLASH) {

**if** (state == SLASH) {

/\* Program text following a slash. \*/

**if** (c == **'\*'**)

state = COMMENT;

**else** {

putchar(**'/'**);

state = PROGRAM;

}

}

**if** (state == PROGRAM) {

/\* Program text. \*/

**if** (c == **'\''** || c == **'"'**) {

quote = c;

state = QUOTE;

putchar(c);

}

**else** **if** (c == "/\*"[**0**])

state = SLASH;

**else**

putchar(c);

}

}

**else** **if** (state == COMMENT) {

/\* Comment. \*/

**if** (c == "/\*"[**1**])

state = STAR;

}

**else** **if** (state == QUOTE) {

/\* Within quoted string or character constant. \*/

putchar(c);

**if** (c == **'\\'**)

state = LITERAL;

**else** **if** (c == quote)

state = PROGRAM;

}

**else** **if** (state == SLASH) {

}

**else** **if** (state == STAR) {

/\* Comment following a star. \*/

**if** (c == **'/'**)

state = PROGRAM;

**else** **if** (c != **'\*'**)

state = COMMENT;

}

**else** /\* state == LITERAL \*/ {

/\* Within quoted string or character constant, following \. \*/

putchar(c);

state = QUOTE;

}

}

**if** (state == SLASH)

putchar(**'/'** //\*\*/

**1**);

**return** **0**;

}

/\*

Local variables:

compile-command: "checkergcc -W -Wall -ansi -pedantic knr123-0.c -o knr123-0"

End:

\*/

From Lew Pitcher

/\* Lew Pitcher <lpitcher@yesic.com> \*/

/\*/

\*\* derem - remove C comments

\*\*

\*\* (attempt to solve K&R Exercise **1**-**22**)

\*\*

\*\* As I only have v1 copy of K&R, I cannot

\*\* be sure what is covered in K&R ANSI chapter **1**.

\*\* So, I restrict myself to the components covered

\*\* in K&R v1 chapter **1**, but modified **for** requisite ANSI

\*\* features (**int** main() and **return** value).

\*\*

\*\* Components covered in v1 K&R chapter **1** include:

\*\* **while** (), **for** (), **if** () **else**

\*\* getchar(), putchar(), EOF

\*\* character constants, character escapes

\*\* strings

\*\* array subscripting

\*\*

\*\* Not directly covered are

\*\* string subscripting ( "/\*"[**0**] )

\*\* initializers ( **int** state = PROGRAM; )

\*\*/

/\*/\*/

**#include <stdio.h>**

**#define PROGRAM 0**

**#define BEGIN\_COMMENT 1**

**#define COMMENT 2**

**#define END\_COMMENT 3**

**#define QUOTE 4**

**int** main(**void**)

{

**int** this\_char, quote\_char;

**int** state;

state = PROGRAM;

**while** ((this\_char = getchar()) != EOF)

{

**if** (state == PROGRAM)

{

**if** (this\_char == **'/'**)

state = BEGIN\_COMMENT;

**else** **if** ((this\_char == **'"'**) || (this\_char == **'\''**))

{

state = QUOTE;

putchar(quote\_char = this\_char);

}

**else** putchar(this\_char);

}

**else** **if** (state == BEGIN\_COMMENT)

{

**if** (this\_char == **'\*'**)

state = COMMENT;

**else**

{

putchar(**'/'**); /\* for the '/' of the comment \*/

**if** (this\_char != **'/'**)

{

state = PROGRAM;

putchar(this\_char);

}

**else** state = COMMENT; /\* stuttered \*/

}

}

**else** **if** (state == QUOTE)

{

putchar(this\_char);

**if** (this\_char == **'\\'**)

putchar(getchar()); /\* escaped character \*/

**else** **if** (this\_char == quote\_char)

state = PROGRAM;

}

**else** **if** (state == COMMENT)

{

**if** (this\_char == **'\*'**)

state = END\_COMMENT;

}

**else** **if** (state == END\_COMMENT)

{

**if** (this\_char == **'/'**)

state = PROGRAM;

**else** **if** (this\_char != **'\*'**) /\* stuttered \*/

state = COMMENT;

}

}

**return** **0**;

}

From Gregory Pietsch

/\* Gregory Pietsch <gkp1@flash.net> \*/

**#include <stdio.h>**

**char** p[] =

"0/!10\"040\'050.001/011\*!21\"/41\'/51./02\*!32.!23/ "

"03\*!33.!24\"004\\064.045\'005\\075.056.047.05";

**int** main(){**int** c,i,d;**char** s,n;s=**'0'**;**while**((c=getchar())

!=EOF){d=**0**;**for**(i=**0**;p[i]!=**'\0'**&&d==**0**;i=i+**4**){**if**(p[i]==s&&

(p[i+**1**]==c||p[i+**1**]==**'.'**)){**if**(p[i+**2**]==**'0'**)putchar(c);**else**

**if**(p[i+**2**]==**'/'**){putchar(**'/'**);putchar(c);}**else** **if**(p[i+**2**]

==**' '**)putchar(**' '**);n=p[i+**3**];d=**1**;}}s=n;}**return** **0**;}

**Category 1 Solutions**

From Ben Pfaff (again)   
  
This version has the var/'\2' bug fix.

/\* K&R2 1-23: Write a program to remove all comments from a C program.

Don't forget to handle quoted strings and character constants

properly. C comments do not nest.

This solution does not deal with other special cases, such as

trigraphs, line continuation with \, or <> quoting on #include,

since these aren't mentioned up 'til then in K&R2. Perhaps this is

cheating.

Note that this program contains both comments and quoted strings of

text that looks like comments, so running it on itself is a

reasonable test. It also contains examples of a comment that ends

in a star and a comment preceded by a slash. Note that the latter

will break C99 compilers and C89 compilers with // comment

extensions.

Interface: The C source file is read from stdin and the

comment-less output is written to stdout. \*\*/

**#include <stdio.h>**

**int**

main(**void**)

{

/\* State machine's current state. \*/

**enum** {

PROGRAM,

SLASH,

COMMENT,

STAR,

QUOTE,

LITERAL

} state;

/\* If state == QUOTE, then ' or ". Otherwise, undefined. \*/

**int** quote;

state = PROGRAM;

**for** (;;) {

**int** c = getchar();

**if** (c == EOF) {

**if** (state == SLASH)

putchar(**'/'** //\*\*/

**1** / **1** /**'\1'**);

**break**;

}

**switch** (state) {

**case** SLASH:

/\* Program text following a slash. \*/

**if** (c == "/\*"[**1**]) {

state = COMMENT;

**break**;

}

putchar(**'/'**);

state = PROGRAM;

/\* Fall through. \*/

**case** PROGRAM:

/\* Program text. \*/

**if** (c == **'\''** || c == **'"'**) {

quote = c;

state = QUOTE;

putchar(c);

}

**else** **if** (c == "/\*"[**0**])

state = SLASH;

**else**

putchar(c);

**break**;

**case** COMMENT:

/\* Comment. \*/

**if** (c == **'\*'**)

state = STAR;

**break**;

**case** STAR:

/\* Comment following a star. \*/

**if** (c == **'/'**)

state = PROGRAM;

**else** **if** (c != **'\*'**) {

state = COMMENT;

putchar (**' '**);

}

**break**;

**case** QUOTE:

/\* Within quoted string or character constant. \*/

putchar(c);

**if** (c == **'\\'**)

state = LITERAL;

**else** **if** (c == quote)

state = PROGRAM;

**break**;

**case** LITERAL:

/\* Within quoted string or character constant, following \. \*/

putchar(c);

state = QUOTE;

**break**;

**default**:

abort();

}

}

**return** **0**;

}

/\*

Local variables:

compile-command: "checkergcc -W -Wall -ansi -pedantic knr123.c -o knr123"

End:

\*/

From Chris Torek

/\* torek@elf.bsdi.com (Chris Torek) \*/

/\*

"Write a program to remove all comments from a C program. Don't forget

to handle quoted strings and character constants properly. C comments do

not nest."

Well, what the heck. I mailed this a day or two ago, but here is

the posted version. I modified the problem a bit: it removes

comments from full ANSI C89 or C99 programs, handling trigraphs

and \-newline sequences. It attempts to preserve any trigraphs in

the output, even while examining them in the "C code" as their

translated characters. (I am not sure why I bothered doing all of

them, when only ??/ matters here.) It keeps output line numbers in

sync with input line numbers, so that if the output is compiled,

any error messages will refer back to the proper input source line.

Lightly tested.

\*/

**#include <assert.h>**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <string.h>**

/\*

\* This flag controls whether we do trigraph processing.

\*/

**int** trigraphs = **1**;

/\*

\* This flag controls whether a comment becomes "whitespace" (ANSI C)

\* or "nothing at all" (some pre-ANSI K&R C compilers).

\*/

**int** whitespace = **1**;

/\*

\* This flag controls whether we do C89 or C99. (C99 also handles C++.)

\*/

**int** c99;

/\*

\* These are global so that options() can get at them, and for later

\* error messages if needed.

\*/

**const** **char** \*inname, \*outname;

**int** options(**const** **char** \*, **char** \*\*);

**void** usage(**void**);

**void** process(FILE \*, FILE \*);

**#ifdef \_\_GNUC\_\_**

**void** panic(**const** **char** \*) \_\_attribute\_\_((noreturn));

**#else**

**void** panic(**const** **char** \*);

**#endif**

**int** main(**int** argc, **char** \*\*argv) {

**int** i;

FILE \*in, \*out;

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

**if** (argv[i][**0**] == **'-'**)

i += options(argv[i] + **1**, argv + i + **1**);

**else** **if** (inname == NULL)

inname = argv[i];

**else**

usage();

}

**if** (inname != NULL) {

**if** ((in = fopen(inname, "r")) == NULL) {

fprintf(stderr, "cannot open %s for reading\n", inname);

exit(EXIT\_FAILURE);

}

} **else** {

inname = "stdin";

in = stdin;

}

**if** (outname != NULL) {

**if** ((out = fopen(outname, "w")) == NULL) {

fprintf(stderr, "cannot open %s for writing\n",

outname);

exit(EXIT\_FAILURE);

}

} **else** {

outname = "stdout";

out = stdout;

}

process(in, out);

fclose(in);

fclose(out);

exit(EXIT\_SUCCESS);

}

/\*

\* This scans for -o type options. Options that have an argument

\* can either take it immediately or as a subsequent argument (e.g.,

\* -ofoo means the same thing as -o foo). We return 0 for "handled

\* them normally", 1 for "handled them normally but needed more

\* arguments".

\*

\* Currently this function is more powerful than really needed, but

\* if we ever decide to have more arguments...

\*/

**int** options(**const** **char** \*afterdash, **char** \*\*moreargs) {

**int** nmore = **0**, c;

**while** ((c = \*afterdash++) != **'\0'**) {

**if** (c == **'o'**) {

**if** (\*afterdash) {

outname = afterdash;

afterdash = "";

} **else** **if** (moreargs[nmore] != NULL)

outname = moreargs[nmore++];

**else**

usage();

} **else** **if** (c == **'t'**)

trigraphs = **0**;

**else** **if** (c == **'w'**)

whitespace = **0**;

**else** **if** (c == **'9'**)

c99 = **1**;

**else**

usage();

}

**return** nmore;

}

**void** usage(**void**) {

fprintf(stderr, "usage: uncomment [-9tw] [-o outfile] [infile]\n");

exit(EXIT\_FAILURE); /\* ??? \*/

}

/\*

\* States, level 0:

\* normal

\* trigraph processing: Q1 Q2 (for ??x)

\*

\* States, level 1:

\* backslash-newline processing: BACK (seen \, may consume NL)

\*

\* States, level 2:

\* normal

\* character constant: CC (seen '), CCBACK (seen \ inside CC)

\* string constant: SC, SCBACK

\* comment: SLASH, COMM, COMMSTAR (for /, in-comment, & seen-star)

\* C99: SLASHSLASH

\*/

**enum** l0state {

L0\_NORMAL,

L0\_Q1, L0\_Q2

};

**enum** l1state {

L1\_NORMAL,

L1\_BACK

};

**enum** l2state {

L2\_NORMAL,

L2\_CC, L2\_CCBACK,

L2\_SC, L2\_SCBACK,

L2\_SLASH, L2\_COMM, L2\_COMMSTAR,

L2\_SLASHSLASH

};

**struct** state {

FILE \*in;

**enum** l0state l0state;

**int** npushback;

**char** pushback[**4**];

**char** pushorig[**4**]; /\* nonzero => trigraph pushback \*/

**int** lastgetc;

**int** lineno;

};

/\*

\* Set up "initial" state.

\*/

**static** **void** state0(**struct** state \*sp, FILE \*in) {

sp->in = in;

sp->l0state = L0\_NORMAL;

sp->npushback = **0**;

sp->lastgetc = **0**;

sp->lineno = **1**;

}

**static** **void** pushback(**struct** state \*sp, **int** c, **char** origc) {

assert(sp->npushback < **sizeof** sp->pushback);

sp->pushback[sp->npushback] = c;

sp->pushorig[sp->npushback++] = origc;

}

/\*

\* Get a character, doing trigraph processing. Set \*origc to 0 for normal

\* characters, or the actual input character pre-trigraph-mapping

\* for trigraph input.

\*

\* As a side effect, this can wind up getting up to 3 characters, maybe

\* stuffing two of them into the pushback buffer sp->buf[]. It also bumps

\* sp->lineno when a previously-read newline has been passed over.

\*/

**static** **int** getl0char(**struct** state \*sp, **char** \*origc) {

**int** c, newc;

**enum** l0state state;

state = sp->l0state;

\*origc = **0**;

**while** ((c = getc(sp->in)) != EOF) {

**if** (sp->lastgetc == **'\n'**)

sp->lineno++;

sp->lastgetc = c;

**switch** (state) {

**case** L0\_NORMAL:

/\* ? => get another character; otherwise we are ok \*/

**if** (c == **'?'**) {

state = L0\_Q1;

**continue**;

}

assert(sp->l0state == L0\_NORMAL);

**return** c;

**case** L0\_Q1:

/\* ?? => get another character \*/

**if** (c == **'?'**) {

state = L0\_Q2;

**continue**;

}

/\* ?X => return ?, look at X later \*/

pushback(sp, c, **0**);

sp->l0state = L0\_NORMAL;

**return** **'?'**;

**case** L0\_Q2:

/\*

\* ??X, where X is trigraph => map

\* ??X, where X is non-trigraph => tricky

\* ??? => also tricky

\*/

**switch** (c) {

**case** **'='**:

newc = **'#'**;

**break**;

**case** **'('**:

newc = **'['**;

**break**;

**case** **'/'**:

newc = **'\\'**;

**break**;

**case** **')'**:

newc = **']'**;

**break**;

**case** **'\''**:

newc = **'^'**;

**break**;

**case** **'<'**:

newc = **'{'**;

**break**;

**case** **'!'**:

newc = **'|'**;

**break**;

**case** **'>'**:

newc = **'}'**;

**break**;

**case** **'?'**:

/\*

\* This one is slightly tricky. Three '?'s

\* mean that the '?' we read two characters

\* ago gets returned, and the two remaining

\* '?'s leave us in Q2 state.

\*/

sp->l0state = L0\_Q2;

**return** **'?'**;

**default**:

/\*

\* This one returns the first ?, leaves

\* the second ? to be re-examined, and

\* leaves the last character to be re-examined.

\* In any case we are back in "normal" state.

\*/

pushback(sp, c, **0**);

pushback(sp, **'?'**, **0**);

sp->l0state = L0\_NORMAL;

**return** **'?'**;

}

/\* mapped a trigraph char -- return new char \*/

\*origc = c;

sp->l0state = L0\_NORMAL;

**return** newc;

**default**:

panic("getl0char state");

}

}

sp->lastgetc = EOF;

**return** EOF;

}

**void** warn(**struct** state \*, **const** **char** \*);

**void** process(FILE \*in, FILE \*out) {

**enum** l1state l1state = L1\_NORMAL;

**enum** l2state l2state = L2\_NORMAL;

**int** c, pendnls;

**char** origc, backc;

**struct** state state;

state0(&state, in);

pendnls = **0**;

backc = **0**; /\* defeat gcc warning \*/

/\*

\* Slight sort-of-bug: files ending in \ cause two "final" getc()s.

\*/

**do** {

**if** (state.npushback) {

c = state.pushback[--state.npushback];

origc = state.pushorig[state.npushback];

} **else** **if** (trigraphs) {

c = getl0char(&state, &origc);

} **else** {

c = getc(in);

origc = **0**;

**if** (state.lastgetc == **'\n'**)

state.lineno++;

state.lastgetc = c;

}

/\*

\* Do backslash-newline processing.

\*/

**switch** (l1state) {

**case** L1\_NORMAL:

**if** (c == **'\\'**) {

l1state = L1\_BACK;

backc = origc;

**continue**;

}

**break**;

**case** L1\_BACK:

/\*

\* If backc is nonzero here, the backslash that

\* got us into this state was spelled ??/ --

\* if we eat a newline (and hence the backslash),

\* we forget that the eaten newline was spelled

\* this way. This is sort of a bug, but so it goes.

\*/

l1state = L1\_NORMAL;

**if** (c == **'\n'**) {

pendnls++;

**continue**;

}

**if** (c != EOF)

pushback(&state, c, origc);

c = **'\\'**;

origc = backc;

**break**;

**default**:

panic("bad l1state");

}

/\*

\* Now ready to do "C proper" processing.

\*/

**#define SYNCLINES() while (pendnls) putc('\n', out), pendnls--**

**#define OUTPUT(ch, tri) ((tri) ? fprintf(out, "??%c", tri) : putc(ch, out))**

**#define COPY() OUTPUT(c, origc)**

**switch** (l2state) {

**case** L2\_NORMAL:

**switch** (c) {

**case** **'\''**:

l2state = L2\_CC;

**break**;

**case** **'"'**:

l2state = L2\_SC;

**break**;

**case** **'/'**:

l2state = L2\_SLASH;

**continue**;

**default**:

**break**;

}

SYNCLINES();

**if** (c != EOF)

COPY();

**break**;

**case** L2\_CC:

**switch** (c) {

**case** EOF:

warn(&state, "EOF in character constant");

**break**;

**case** **'\n'**:

warn(&state, "newline in character constant");

**break**;

**case** **'\\'**:

l2state = L2\_CCBACK;

**break**;

**case** **'\''**:

l2state = L2\_NORMAL;

**break**;

**default**:

**break**;

}

**if** (c != EOF)

COPY();

**break**;

**case** L2\_CCBACK:

**switch** (c) {

**case** EOF:

warn(&state, "EOF in character constant");

**break**;

**case** **'\n'**:

warn(&state, "newline in character constant");

**break**;

**default**:

**break**;

}

l2state = L2\_CC;

**if** (c != EOF)

COPY();

**break**;

**case** L2\_SC: /\* much like CC \*/

**switch** (c) {

**case** EOF:

warn(&state, "EOF in string constant");

**break**;

**case** **'\n'**:

warn(&state, "newline in string constant");

**break**;

**case** **'\\'**:

l2state = L2\_SCBACK;

**break**;

**case** **'"'**:

l2state = L2\_NORMAL;

**break**;

**default**:

**break**;

}

**if** (c != EOF)

COPY();

**break**;

**case** L2\_SCBACK:

**switch** (c) {

**case** EOF:

warn(&state, "EOF in string constant");

**break**;

**case** **'\n'**:

warn(&state, "newline in string constant");

**break**;

**default**:

**break**;

}

l2state = L2\_SC;

**if** (c != EOF)

COPY();

**break**;

**case** L2\_SLASH:

**if** (c == **'\*'**)

l2state = L2\_COMM;

**else** **if** (c99 && c == **'/'**)

l2state = L2\_SLASHSLASH;

**else** {

SYNCLINES();

OUTPUT(**'/'**, **0**);

**if** (c != **'/'**) {

**if** (c != EOF)

COPY();

l2state = L2\_NORMAL;

}

}

**break**;

**case** L2\_COMM:

**switch** (c) {

**case** **'\*'**:

l2state = L2\_COMMSTAR;

**break**;

**case** **'\n'**:

pendnls++;

**break**;

**case** EOF:

warn(&state, "EOF inside comment");

**break**;

}

**break**;

**case** L2\_COMMSTAR:

**switch** (c) {

**case** **'/'**:

l2state = L2\_NORMAL;

/\*

\* If comments become whitespace,

\* and we have no pending newlines,

\* must emit a blank here.

\*

\* The comment text is now all eaten.

\*/

**if** (whitespace && pendnls == **0**)

putc(**' '**, out);

SYNCLINES();

**break**;

**case** **'\*'**:

/\* stay in L2\_COMMSTAR state \*/

**break**;

**case** EOF:

warn(&state, "EOF inside comment");

**break**;

**case** **'\n'**:

pendnls++;

/\* FALLTHROUGH \*/

**default**:

l2state = L2\_COMM;

}

**break**;

**case** L2\_SLASHSLASH:

**switch** (c) {

**case** EOF:

/\* ??? do we really care? \*/

warn(&state, "EOF inside //-comment");

**break**;

**case** **'\n'**:

l2state = L2\_NORMAL;

pendnls++; /\* cheesy, but... \*/

SYNCLINES();

**default**:

**break**;

}

**break**;

**default**:

panic("bad l2state");

}

} **while** (c != EOF);

SYNCLINES();

}

**void** warn(**struct** state \*sp, **const** **char** \*msg) {

fprintf(stderr, "uncomment: %s(%d): %s\n", inname, sp->lineno, msg);

}

**void** panic(**const** **char** \*msg) {

fprintf(stderr, "panic: %s\n", msg);

abort();

exit(EXIT\_FAILURE);

}

From Chris Mears   
  
Here's Chris's updated version, without the bugs (says he). **:-)**

/\*

\* C comment stripper.

\*

\* Strips comments from C or C++ code.

\*/

**#include <stdio.h>**

**enum** state\_t { normal, string, character, block\_comment, line\_comment};

**enum** token\_t { none, backslash, slash, star, tri1, tri2, tri\_backslash};

**static** **int** print\_mode(**enum** state\_t s)

{

**return** (s == normal || s == string || s == character);

}

**void** cstrip(FILE \*infile, FILE \*outfile)

{

**int** ch;

**int** comment\_newline = **0**;

**enum** state\_t state = normal;

**enum** token\_t token = none;

**enum** token\_t last\_token = none;

**if** (!infile || !outfile || (infile == outfile)) {

**return**;

}

**while** ((ch = fgetc(infile)) != EOF) {

**switch** (ch) {

**case** **'/'**:

**if** (token == tri2) {

token = tri\_backslash;

**if** (print\_mode(state))

fputc(ch, outfile);

} **else** **if** (state == string || state == character) {

fputc(ch, outfile);

token = slash;

} **else** **if** (state == block\_comment && token == star) {

state = normal;

token = none;

/\* Replace block comments with whitespace. \*/

**if** (comment\_newline) {

fputc(**'\n'**, outfile);

} **else** {

fputc(**' '**, outfile);

}

} **else** **if** (state == normal && token == slash) {

state = line\_comment;

token = slash;

} **else** {

token = slash;

}

**break**;

**case** **'\\'**:

**if** (state == normal && token == slash)

fputc(**'/'**, outfile);

**if** (print\_mode(state))

fputc(ch, outfile);

**if** (token == backslash || token == tri\_backslash) {

token = none;

} **else** {

last\_token = token;

token = backslash;

}

**break**;

**case** **'"'**:

**if** (state == normal && token == slash)

fputc(**'/'**, outfile);

**if** (state == string && token != backslash)

state = normal;

**else** **if** (state == normal && token != backslash)

state = string;

**if** (print\_mode(state))

fputc(ch, outfile);

token = none;

**break**;

**case** **'\''**:

**if** (state == normal && token == slash)

fputc(**'/'**, outfile);

**if** (state == character && token != backslash)

state = normal;

**else** **if** (state == normal && token != backslash)

state = character;

**if** (print\_mode(state))

fputc(ch, outfile);

token = none;

**break**;

**case** **'\n'**:

/\* This test is independent of the others. \*/

**if** (state == block\_comment)

comment\_newline = **1**;

**if** (state == normal && token == slash)

fputc(**'/'**, outfile);

**if** (token == backslash || token == tri\_backslash)

token = last\_token;

**else** **if** (state == line\_comment &&

token != backslash) {

state = normal;

token = none;

} **else** {

token = none;

}

**if** (print\_mode(state))

fputc(ch, outfile);

**break**;

**case** **'\*'**:

**if** (state == normal && token == slash) {

state = block\_comment;

token = none;

comment\_newline = **0**;

} **else** {

token = star;

}

**if** (print\_mode(state))

fputc(ch, outfile);

**break**;

**case** **'?'**:

**if** (state == normal && token == slash)

fputc(**'/'**, outfile);

**if** (token == tri1) {

token = tri2;

} **else** **if** (token == tri2) {

token = tri2; /\* retain state \*/

} **else** {

/\* We might need the last token if this

\* trigraph turns out to be a backslash.

\*/

last\_token = token;

token = tri1;

}

**if** (print\_mode(state))

fputc(ch, outfile);

**break**;

**default**:

**if** (state == normal && token == slash)

fputc(**'/'**, outfile);

**if** (print\_mode(state))

fputc(ch, outfile);

token = none;

**break**;

} /\* switch \*/

} /\* while \*/

**return**;

}

/\* Small driver program. \*/

**int** main(**void**)

{

cstrip(stdin, stdout);

**return** **0**;

}

Here's a critique of the above, sent in by Rick Litherland. (Please note: when Rick posted this, I hadn't yet posted Chris Mears's updated version of the code.)

(Since I find it hard to pick the solution number out of KRX12300.C at a glance, I'll refer to the solutions as uncomment00, uncomment01, and so on.)

[Rick - KR means K&R. X means eXercise. 1 means Chapter 1. 23 means exercise 23. The next digit is the category number - 0 == Cat 0 (ANSI C89, with code restricted to what K&R have discussed at this point in the book). The final digit is the solution number. 0 is the first I received in that category, 1 is the second, and so on. (RJH)]

uncomment03 (Gregory Pietsch)

===========

I can find only one possible flaw in this, namely that it does not allow for a slash in program text being immediately followed by a quotation mark. One could reasonably argue that this is not a flaw at all, because that would never happen in sensible code. On the other hand, it can happen in legal code, as demonstrated by the following complete (if useless) program.

**#include <stdio.h>**

**int** main(**void**)

{

/\* print the number three \*/

printf("%d\n", **6**/**'\2'**);

/\* remember to return a value from main \*/

**return** **0**;

}

When this is fed to uncomment03, the output is

**#include <stdio.h>**

**int** main(**void**)

{

printf("%d\n", **6**/**'\2'**);

/\* remember to return a value from main \*/

**return** **0**;

}

Clearly, uncomment03 realises that the second comment is too important to remove. Um, sorry, that was a feeble excuse for a joke. What's happening is that uncomment03 doesn't recognise the beginning of the character constant '\2', so it takes the closing quote as the start of a "character constant" that is never terminated. The peculiar idiom 6/'\2' for 3 can be replaced by the even more brain-damaged 6/"\2"[0] with the same effect. Since uncomment03 is table-driven, it's easy to make it recognise these situations by adding two new rules to the table.

/\* modified krx12303.c \*/

**#include <stdio.h>**

**char** p[] =

"0/!10\"@40\'@50.@01/@11\*!2"

"1\"/41\'/5" /\* added by RAL \*/

"1./02\*!32.!23/ 03\*!33.!24\"@04\\@64.@45\'@05\\@75.@56.@47.@5";

**int** main(){**int** c,i,d;**char** s,n;s=**'0'**;**while**((c=getchar())

!=EOF){d=**0**;**for**(i=**0**;p[i]!=**'\0'**&&d==**0**;i=i+**4**){**if**(p[i]==s&&

(p[i+**1**]==c||p[i+**1**]==**'.'**)){**if**(p[i+**2**]==**'@'**)putchar(c);**else**

**if**(p[i+**2**]==**'/'**){putchar(**'/'**);putchar(c);}**else** **if**(p[i+**2**]

==**' '**)putchar(**' '**);n=p[i+**3**];d=**1**;}}s=n;}**return** **0**;}

/\* end of modified krx12303.c \*/

uncomment02 (Lew Pitcher)

===========

uncomment11 (Chris Torek)

===========

These have the same problem (or non-problem, according to your point of view) as uncomment03. If it were regarded as a problem, it could probably be fixed quite easily, though not (I think) as neatly as with uncomment03; I haven't looked at these carefully enough to be sure.

uncomment01, uncomment10 (Ben Pfaff)

=========== ===========

An oversight has the effect that if a slash in program text is followed by anything other than a star or another slash, the following character is dropped. For example, with input

**int** a = **4**/**2**;

the output is

**int** a = **4**/;

The correction is the same in both cases; replace

/\* Program text following a slash. \*/

**if** (c == **'\*'**)

state = COMMENT;

**else** {

putchar(**'/'**);

**if** (c != **'/'**)

state = PROGRAM;

}

by

/\* Program text following a slash. \*/

**if** (c == **'\*'**)

state = COMMENT;

**else** {

putchar(**'/'**);

**if** (c != **'/'**) {

putchar(c);

state = PROGRAM;

}

}

After this, these programs will have the same problem (or not) as the previous three.

uncomment12 (Chris Mears)

===========

This is a completely different kettle of fish. If you run this with Ben Pfaff's solution as input, the output is quite bizarre; some comments have just their initial and final slashes removed, for instance. I've managed to find two things contributing to this. The first is illustrated by the input

**int** c = **'/'**;

with output

**int** c = **''**;

This can be fixed by changing the lines

**case** **'/'**:

**if** (state == string) {

to

**case** **'/'**:

**if** (state == string || state == character) {

However, with or without this change, the input

char \*p = "\\"; /\* This is not a comment. \*/

is left unchanged. What happens is that the closing quote of the string literal isn't recognised as such because of the preceding backlash, despite the backslash before that. The handling of backslashes is split between three cases (at least), and is complicated enough that I don't feel competent to propose a remedy.

This program breaks most of the above submissions:

/\* krx123tp.c - a test program to serve as input to krx123\*.c

\*

\* This is a shameless copy of Ben Pfaff's solution, to which I have

\* added a few extra statements to further test the candidate programs

\* for this exercise. As Ben says, this program already contains lots

\* of examples of comments and not-quite-comments. I've just made it

\* a little tougher.

\*

\*/

/\* K&R2 1-23: Write a program to remove all comments from a C program.

Don't forget to handle quoted strings and character constants

properly. C comments do not nest.

This solution does not deal with other special cases, such as

trigraphs, line continuation with \, or <> quoting on #include,

since these aren't mentioned up 'til then in K&R2. Perhaps this is

cheating.

Note that this program contains both comments and quoted strings of

text that looks like comments, so running it on itself is a

reasonable test. It also contains examples of a comment that ends

in a star and a comment preceded by a slash. Note that the latter

will break C99 compilers and C89 compilers with // comment

extensions.

Interface: The C source file is read from stdin and the

comment-less output is written to stdout. \*\*/

**#include <stdio.h>**

**int**

main(**void**)

{

/\* State machine's current state. \*/

**enum** {

PROGRAM,

SLASH,

COMMENT,

STAR,

QUOTE,

LITERAL

} state;

/\* If state == QUOTE, then ' or ". Otherwise, undefined. \*/

**int** quote;

state = PROGRAM;

**for** (;;) {

**int** c = getchar();

**if** (c == EOF) {

**if** (state == SLASH)

putchar(**'/'** //\*\*/

**1** / **1** /**'\1'**);

**break**;

}

**if**(**0**)

printf("%d\n", **6**/**'\2'**);

/\* line of code, and comment, added by RJH 10 July 2000 \*/

**switch** (state) {

**case** SLASH:

/\* Program text following a slash. \*/

**if** (c == "/\*"[**1**]) {

state = COMMENT;

**break**;

}

putchar(**'/'**);

state = PROGRAM;

/\* Fall through. \*/

**case** PROGRAM:

/\* Program text. \*/

**if** (c == **'\''** || c == **'"'**) {

quote = c;

state = QUOTE;

putchar(c);

}

**else** **if** (c == "/\*"[**0**])

state = SLASH;

**else**

putchar(c);

**break**;

**case** COMMENT:

/\* Comment. \*/

**if** (c == **'\*'**)

state = STAR;

**break**;

**case** STAR:

/\* Comment following a star. \*/

**if** (c == **'/'**)

state = PROGRAM;

**else** **if** (c != **'\*'**) {

state = COMMENT;

putchar (**' '**);

}

**break**;

**case** QUOTE:

/\* Within quoted string or character constant. \*/

putchar(c);

**if** (c == **'\\'**)

state = LITERAL;

**else** **if** (c == quote)

state = PROGRAM;

**break**;

**case** LITERAL:

/\* Within quoted string or character constant, following \. \*/

putchar(c);

state = QUOTE;

**break**;

**default**:

abort();

}

}

**return** /\* this comment added by RJH 10 July 2000 \*/ **0**;

}

/\*

Local variables:

compile-command: "checkergcc -W -Wall -ansi -pedantic knr123.c -o knr123"

End:

\*/

**Answer to Exercise 1-24, page 34**

*Write a program to check a C program for rudimentary syntax errors like unbalanced parentheses, brackets and braces. Don't forget about quotes, both single and double, escape sequences, and comments. (This program is hard if you do it in full generality.)*   
  
Rick Dearman's Category 0 solution:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

KnR 1-24

--------

Write a program to check the syntax of a C program

for matching {} () "" '' []

Author: Rick Dearman

email: rick@ricken.demon.co.uk

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**#include <stdio.h>**

**#define MAXLINE 1000 /\* max input line size \*/**

**char** line[MAXLINE]; /\*current input line\*/

**int** getline(**void**); /\* taken from the KnR book. \*/

**int**

main()

{

**int** len=**0**;

**int** t=**0**;

**int** brace=**0**, bracket=**0**, parenthesis=**0**;

**int** s\_quote=**1**, d\_quote=**1**;

**while** ((len = getline()) > **0** )

{

t=**0**;

**while**(t < len)

{

**if**( line[t] == **'['**)

{

brace++;

}

**if**( line[t] == **']'**)

{

brace--;

}

**if**( line[t] == **'('**)

{

parenthesis++;

}

**if**( line[t] == **')'**)

{

parenthesis--;

}

**if**( line[t] == **'\''**)

{

s\_quote \*= -**1**;

}

**if**( line[t] == **'"'**)

{

d\_quote \*= -**1**;

}

t++;

}

}

**if**(d\_quote !=**1**)

printf ("Mismatching double quote mark\n");

**if**(s\_quote !=**1**)

printf ("Mismatching single quote mark\n");

**if**(parenthesis != **0**)

printf ("Mismatching parenthesis\n");

**if**(brace != **0**)

printf ("Mismatching brace mark\n");

**if**(bracket != **0**)

printf ("Mismatching bracket mark\n");

**if**( bracket==**0** && brace==**0** && parenthesis==**0** && s\_quote == **1** && d\_quote == **1**)

printf ("Syntax appears to be correct.\n");

**return** **0**;

}

/\* getline: specialized version \*/

**int** getline(**void**)

{

**int** c, i;

**extern** **char** line[];

**for** ( i=**0**;i<MAXLINE-**1** && ( c=getchar()) != EOF && c != **'\n'**; ++i)

line[i] = c;

**if**(c == **'\n'**)

{

line[i] = c;

++i;

}

line[i] = **'\0'**;

**return** i;

}

Stefan Farfeleder's Category 1 solution:

/\* 1-24.c \*/

**#include <stdio.h>**

**#include <stdlib.h>**

**#define MAX\_STACK 1024**

**enum**

{

CODE, /\* nothing of the following \*/

COMMENT, /\* inside a comment \*/

QUOTE1, /\* inside '' \*/

QUOTE2 /\* inside "" \*/

};

**int** main(**void**)

{

**int** ch;

**int** state = CODE;

**char** stack[MAX\_STACK];

size\_t top = **0**; /\* points to the top of the stack :-) \*/

size\_t line = **1**;

**int** error = **0**; /\* for ok-message \*/

**while** ((ch = getchar()) != EOF)

{

**if** (ch == **'\n'**)

{

line++;

}

**switch** (state)

{

**case** CODE:

**if** (ch == **'\''**)

{

state = QUOTE1;

}

**else** **if** (ch == **'"'**)

{

state = QUOTE2;

}

**else** **if** (ch == **'/'**)

{

**int** second = getchar();

**if** (second == **'\*'**)

{

state = COMMENT;

}

**else**

{

ungetc(second, stdin);

}

}

**else** **if** (ch == **'('** || ch == **'['** || ch == **'{'**)

{

**if** (top < MAX\_STACK)

{

stack[top++] = ch;

}

**else**

{

printf("Stack too small!\n");

**return** EXIT\_FAILURE; /\* exit gracefully :-) \*/

}

}

**else** **if** (ch == **')'** || ch == **']'** || ch == **'}'**)

{

**if** (top == **0**) /\* found closing brace but stack is empty \*/

{

printf("Line %lu: Closing '%c' found without "

"counterpart.\n", (**unsigned** **long**)line, ch);

error = **1**;

}

**else**

{

**char** open = stack[--top];

**if** ((ch == **')'** && open != **'('**) ||

(ch == **']'** && open != **'['**) ||

(ch == **'}'** && open != **'{'**))

{

printf("Line %lu: Closing '%c' does not match "

"opening '%c'.\n", (**unsigned** **long**)line, ch, open);

error = **1**;

}

}

}

**break**;

**case** COMMENT:

**if** (ch == **'\*'**)

{

**int** second = getchar();

**if** (second == **'/'**)

{

state = CODE;

}

**else**

{

ungetc(second, stdin);

}

}

**break**;

**case** QUOTE1:

**if** (ch == **'\\'**)

{

(**void**)getchar(); /\* an escaped char inside '' throw it away \*/

}

**else** **if** (ch == **'\''**)

{

state = CODE;

}

**break**;

**case** QUOTE2:

**if** (ch == **'\\'**)

{

(**void**)getchar(); /\* an escaped char inside "" throw it away \*/

}

**else** **if** (ch == **'"'**)

{

state = CODE;

}

**break**;

}

}

**if** (state == COMMENT)

{

printf("Code ends inside comment!\n");

}

**else** **if** (state == QUOTE1)

{

printf("Code ends inside single quotes!\n");

}

**else** **if** (state == QUOTE2)

{

printf("Code ends inside double quotes!\n");

}

**else** **if** (top == **0** && error == **0**)

{

printf("Code seems to be ok.\n");

}

**if** (top > **0**) /\* still something in the stack \*/

{

size\_t i;

**for** (i = **0**; i < top; i++)

{

printf("Opening '%c' found without counterpart.\n", stack[i]);

}

}

**return** **0**;

}

Stig Brautaset's Cat 1 solution:

/\* This is my first rudimentary C syntax checker. It checks for syntax errors,

\* like closing a set of brackets using the wrong type. It is not \*very\* good

\* at it, but it does not bother about comments, and it does know something

\* about escape sequences and character strings/constants.

\*

\* It uses a simple static stack to keep track of the braces, and it also uses

\* a stack to keep track of the errors on each line. Someday I might change

\* that to use a queue for the error-tracking, because as it is now, it outputs

\* the rightmost error on the line first, and then it steps leftwards (if there

\* is more than one error on each line).

\*

\* I might also implement my dynamically allocated stack and queue implementa-

\* tions, so that running out of space in the stack is not an issue. I might

\* also skip it, since it has little to do with the exercise in question.

\*

\* The program is especially bad at error-recovery. If it finds an error, (or

\* something it believes to be an error) subsequent errors reported might be a

\* bit dubious.

\*/

**#include <stdio.h>**

**#define MAXVAL 1000**

**#define MAXLINE 1000**

**typedef** **struct** {

**int** top;

**int** val[MAXVAL];

**int** pos[MAXVAL];

} stackstr;

/\* very simple stack push function \*/

**int** push(stackstr \*stk, **int** foo, **int** bar)

{

**if** (stk->top == MAXVAL) {

printf("stack overflow. NOT putting more values on the stack.\n");

**return** **1**;

}

stk->val[stk->top] = foo;

stk->pos[stk->top] = bar;

stk->top++;

**return** **0**;

}

/\* very simple function to pop values off a stack \*/

**int** pop(stackstr \*stk, **int** \*foo, **int** \*bar)

{

**if** (stk->top == **0**) {

**return** **1**;

}

stk->top--;

\*foo = stk->val[stk->top];

\*bar = stk->pos[stk->top];

**return** **0**;

}

/\* we go through the input one line at a time, and this function

\* gets the line to test

\*/

**int** getline(**char** \*s, **int** lim)

{

**int** i, c;

**for** (i = **0**; i < lim - **1** && (c = getchar()) != EOF && c != **'\n'**; i++)

\*(s + i) = c;

**if** (c == **'\n'**)

\*(s + i++) = c;

\*(s + i) = **'\0'**;

**return** i;

}

**void** scanline(stackstr \*stk, stackstr \*errstk, **char** \*s, **int** len)

{

**int** i, c, d, foo;

**static** **int** string = **0**, comment = **0**, isconst = **0**, escape = **0**;

**for** (i = **0**; i < len; i++) {

c = \*(s + i);

**if** (!comment) {

**if** (c == **'\\'**) { /\* we have an escape \*/

/\* test for a valid escape sequence \*/

**if** ((d = \*(s + ++i)) == **'\\'** || d == **'n'** || d == **'0'** || d == **'r'** || d == **'?'**

|| d == **'t'** || d == **'\''** || d == **'\"'** || d == **'b'** || d == **'x'**) {

**continue**; /\* ok, valid escape sequence -- don't bother about it \*/

} **else** {

push(errstk, **5**, i); /\* illigal escape sequence \*/

}

} **else** **if** (c == **'\"'**) { /\* is it a text string then? \*/

**if** (!string)

string = **1**;

**else**

string = **0**;

} **else** **if** (c == **'\''**) { /\* is it a constant? \*/

**if** (!isconst)

isconst = **1**;

**else**

isconst = **0**;

}

}

**if** (!isconst && !string && !comment && c == **'/'**) {

**if** ((d = \*(s + ++i)) == **'\*'**)

comment = **1**;

} **else** **if** (comment && c == **'\*'**) {

**if** ((d = \*(s + ++i)) == **'/'**) {

comment = **0**;

**continue**; /\* done with the comment stuff -- start over \*/

}

}

/\* only bother about ({[ ]})'s that's not in

\* a string, constant or comment

\*/

**if** (!isconst && !string && !comment) {

**if** (c == **'('** || c == **'{'** || c == **'['**) {

push(stk, c, **0**);

} **else** **if** (c == **']'** || c == **'}'** || c == **')'**) {

**if** (pop(stk, &d, &foo)) {

push(errstk, **4**, i);

}

**if** (c == **')'** && d != **'('**) {

push(stk, d, **0**);

push(errstk, **1**, i);

} **else** **if** (c == **']'** && d != **'['**) {

push(stk, d, **0**);

push(errstk, **2**, i);

} **else** **if** (c == **'}'** && d != **'{'**) {

push(stk, d, **0**);

push(errstk, **3**, i);

}

}

}

}

}

/\* print errors on the line (if there were any) \*/

**void** print\_err(stackstr \*errstk, **int** lineno)

{

**int** errno, pos;

/\* yes I know... this way the errors come "backwards" :) \*/

**while** (!pop(errstk, &errno, &pos)) {

printf("on line number %d: ", lineno);

**switch**(errno) {

**case** **1**:

printf("closing unopened parantheses, column %d\n", pos+**1**);

**break**;

**case** **2**:

printf("closing unopened square bracket, column %d\n", pos+**1**);

**break**;

**case** **3**:

printf("closing unopened curly braces, column %d\n", pos+**1**);

**break**;

**case** **4**:

printf("trying to close unopened block/control structure, column %d\n", pos+**1**);

**break**;

**case** **5**:

printf("illigal escape sequence, column %d\n", pos+**1**);

**break**;

**default**:

printf("undeterminable error\n");

**break**;

}

}

}

**int** main(**void**)

{

stackstr errstk = {**0**}, stk = {**0**};

**int** c, linenbr = **0**, errcount = **0**, linelen;

**char** line[MAXLINE];

**while** ((linelen = getline(line, MAXLINE)) > **0**) {

linenbr++;

scanline(&stk, &errstk, line, linelen);

**if** (errstk.top) {

print\_err(&errstk, linenbr);

errcount++;

}

}

**if** (errcount)

printf("%d lines contained error(s)\n", errcount);

**else**

printf("Well, \*I\* didn't find any syntax errors, but don't take my word for it...:)\n");

**return** **0**;

}

**Answer to Exercise 2-1, page 36**

*Write a program to determine the ranges of char , short , int , and long variables, both signed and unsigned , by printing appropriate values from standard headers and by direct computation. Harder if you compute them: determine the ranges of the various floating-point types.*

**#include <stdio.h>**

**#include <limits.h>**

**int**

main ()

{

printf("Size of Char %d\n", CHAR\_BIT);

printf("Size of Char Max %d\n", CHAR\_MAX);

printf("Size of Char Min %d\n", CHAR\_MIN);

printf("Size of int min %d\n", INT\_MIN);

printf("Size of int max %d\n", INT\_MAX);

printf("Size of long min %ld\n", LONG\_MIN); /\* RB \*/

printf("Size of long max %ld\n", LONG\_MAX); /\* RB \*/

printf("Size of short min %d\n", SHRT\_MIN);

printf("Size of short max %d\n", SHRT\_MAX);

printf("Size of unsigned char %u\n", UCHAR\_MAX); /\* SF \*/

printf("Size of unsigned long %lu\n", ULONG\_MAX); /\* RB \*/

printf("Size of unsigned int %u\n", UINT\_MAX); /\* RB \*/

printf("Size of unsigned short %u\n", USHRT\_MAX); /\* SF \*/

**return** **0**;

}

**Answer to Exercise 2-2, page 42**

Exercise 2-2 discusses a for loop from the text. Here it is:

**for**(i=**0**; i<lim-**1** && (c=getchar()) != **'\n'** && c != EOF; ++i)

s[i] = c;

*Write a loop equivalent to the for loop above without using && or || .*

**#include <stdio.h>**

**#define MAX\_STRING\_LENGTH 100**

**int** main(**void**)

{

/\*

for (i = 0; i < lim-1 && (c=getchar()) != '\n' && c != EOF; ++i)

s[i] = c;

\*/

**int** i = **0**,

lim = MAX\_STRING\_LENGTH,

c;

**char** s[MAX\_STRING\_LENGTH];

**while** (i < (lim - **1**))

{

c = getchar();

**if** (c == EOF)

**break**;

**else** **if** (c == **'\n'**)

**break**;

s[i++] = c;

}

s[i] = **'\0'**; /\* terminate the string \*/

**return** **0**;

}

Here's a Category 1 solution from Craig Schroeder, which is not so much exegetic as - um - cute. :-)

**#include <stdio.h>**

**#define lim 80**

**int** main()

{

**int** i, c;

**char** s[lim];

/\* There is a sequence point after the first operand of ?: \*/

**for**(i=**0**; i<lim-**1** ? (c=getchar()) != **'\n'** ? c != EOF : **0** : **0** ; ++i)

s[i] = c;

**return** s[i] ^= s[i]; /\* null terminate and return. \*/

}

**Answer to Exercise 2-3, page 46**

*Write the function htoi(s) , which converts a string of hexadecimal digits (including an optional 0x or 0X) into its equivalent integer value. The allowable digits are 0 through 9, a through f, and A through F .*   
  
Here's my solution:

/\* Write the function htoi(s), which converts a string of hexadecimal

\* digits (including an optional 0x or 0X) into its equivalent integer

\* value. The allowable digits are 0 through 9, a through f, and

\* A through F.

\*

\* I've tried hard to restrict the solution code to use only what

\* has been presented in the book at this point (page 46). As a

\* result, the implementation may seem a little naive. Error

\* handling is a problem. I chose to adopt atoi's approach, and

\* return 0 on error. Not ideal, but the interface doesn't leave

\* me much choice.

\*

\* I've used unsigned int to keep the behaviour well-defined even

\* if overflow occurs. After all, the exercise calls for conversion

\* to 'an integer', and unsigned ints are integers!

\*/

/\* These two header files are only needed for the test driver \*/

**#include <stdio.h>**

**#include <stdlib.h>**

/\* Here's a helper function to get me around the problem of not

\* having strchr

\*/

**int** hexalpha\_to\_int(**int** c)

{

**char** hexalpha[] = "aAbBcCdDeEfF";

**int** i;

**int** answer = **0**;

**for**(i = **0**; answer == **0** && hexalpha[i] != **'\0'**; i++)

{

**if**(hexalpha[i] == c)

{

answer = **10** + (i / **2**);

}

}

**return** answer;

}

**unsigned** **int** htoi(**const** **char** s[])

{

**unsigned** **int** answer = **0**;

**int** i = **0**;

**int** valid = **1**;

**int** hexit;

**if**(s[i] == **'0'**)

{

++i;

**if**(s[i] == **'x'** || s[i] == **'X'**)

{

++i;

}

}

**while**(valid && s[i] != **'\0'**)

{

answer = answer \* **16**;

**if**(s[i] >= **'0'** && s[i] <= **'9'**)

{

answer = answer + (s[i] - **'0'**);

}

**else**

{

hexit = hexalpha\_to\_int(s[i]);

**if**(hexit == **0**)

{

valid = **0**;

}

**else**

{

answer = answer + hexit;

}

}

++i;

}

**if**(!valid)

{

answer = **0**;

}

**return** answer;

}

/\* Solution finished. This bit's just a test driver, so

\* I've relaxed the rules on what's allowed.

\*/

**int** main(**void**)

{

**char** \*endp = NULL;

**char** \*test[] =

{

"F00",

"bar",

"0100",

"0x1",

"0XA",

"0X0C0BE",

"abcdef",

"123456",

"0x123456",

"deadbeef",

"zog\_c"

};

**unsigned** **int** result;

**unsigned** **int** check;

size\_t numtests = **sizeof** test / **sizeof** test[**0**];

size\_t thistest;

**for**(thistest = **0**; thistest < numtests; thistest++)

{

result = htoi(test[thistest]);

check = (**unsigned** **int**)strtoul(test[thistest], &endp, **16**);

**if**((\*endp != **'\0'** && result == **0**) || result == check)

{

printf("Testing %s. Correct. %u\n", test[thistest], result);

}

**else**

{

printf("Testing %s. Incorrect. %u\n", test[thistest], result);

}

}

**return** **0**;

}

And here's Marshall's:

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <string.h>**

**#include <ctype.h>**

**long** hchartoi (**char** hexdig, **int** pos); /\* converts a hex char to decimal knowing its 0 based place value \*/

**long** htoi (**char** hexstring[]); /\* converts a string of hex bits to integer ... \*/

**int** main(**void**)

{

**char** \*endp = NULL;

**char** \*test[] =

{

"F00",

"bar",

"0100",

"0x1",

"0XA",

"0X0C0BE",

"abcdef",

"123456",

"0x123456",

"deadbeef",

"zog\_c"

};

**long** **int** result;

**long** **int** check;

size\_t numtests = **sizeof** test / **sizeof** test[**0**];

size\_t thistest;

**for**(thistest = **0**; thistest < numtests; thistest++)

{

result = htoi(test[thistest]);

check = strtol(test[thistest], &endp, **16**);

**if**((\*endp != **'\0'** && result == -**1**) || result == check)

{

printf("Testing %s. Correct. %ld\n", test[thistest], result);

}

**else**

{

printf("Testing %s. Incorrect. %ld\n", test[thistest], result);

}

}

**return** **0**;

}

**long** htoi (**char** s[])

{

**char** \*p = &s[strlen(s)-**1**];

**long** deci = **0**, dig = **0**;

**int** pos = **0**;

**while** (p >= s) {

**if** ((dig = hchartoi(\*p, pos)) < **0** ) {

printf("Error\n");

**return** -**1**;

}

deci += dig;

--p;

++pos;

}

**return** deci;

}

/\* convert hex char to decimal value \*/

**long** hchartoi (**char** hexdig, **int** pos)

{

**char** hexdigits[] = "0123456789ABCDEF";

**char** \*p = &hexdigits[**0**];

**long** deci = **0**;

**int** i;

**while** (\*p != toupper(hexdig) && deci < **16**) {

++p;

++deci;

}

**if** (\*p == toupper(hexdig)) {

**for** (i = **0**; i < pos; i++)

deci \*= **16**;

**return** deci;

}

**return** -**1**;

}

**Answer to Exercise 2-4, page 48**

*Write an alternate version of squeeze(s1,s2) that deletes each character in the string s1 that matches any character in the string s2 .*

/\*

\* Exercise 2-4 Page 48

\*

\* Write an alternate version of squeeze(s1,s2) that deletes each

\* character in s1 that matches any character in the string s2.

\*

\*/

/\* squeeze2: delete all characters occurring in s2 from string s1. \*/

**void** squeeze2(**char** s1[], **char** s2[])

{

**int** i, j, k;

**int** instr2 = **0**;

**for**(i = j = **0**; s1[i] != **'\0'**; i++)

{

instr2 = **0**;

**for**(k = **0**; s2[k] != **'\0'** && !instr2; k++)

{

**if**(s2[k] == s1[i])

{

instr2 = **1**;

}

}

**if**(!instr2)

{

s1[j++] = s1[i];

}

}

s1[j] = **'\0'**;

}

/\* test driver \*/

**#include <stdio.h>**

**#include <string.h>**

**int** main(**void**)

{

**char** \*leftstr[] =

{

"",

"a",

"antidisestablishmentarianism",

"beautifications",

"characteristically",

"deterministically",

"electroencephalography",

"familiarisation",

"gastrointestinal",

"heterogeneousness",

"incomprehensibility",

"justifications",

"knowledgeable",

"lexicographically",

"microarchitectures",

"nondeterministically",

"organizationally",

"phenomenologically",

"quantifications",

"representationally",

"straightforwardness",

"telecommunications",

"uncontrollability",

"vulnerabilities",

"wholeheartedly",

"xylophonically", /\* if there is such a word :-) \*/

"youthfulness",

"zoologically"

};

**char** \*rightstr[] =

{

"",

"a",

"the",

"quick",

"brown",

"dog",

"jumps",

"over",

"lazy",

"fox",

"get",

"rid",

"of",

"windows",

"and",

"install",

"linux"

};

**char** buffer[**32**];

size\_t numlefts = **sizeof** leftstr / **sizeof** leftstr[**0**];

size\_t numrights = **sizeof** rightstr / **sizeof** rightstr[**0**];

size\_t left = **0**;

size\_t right = **0**;

**for**(left = **0**; left < numlefts; left++)

{

**for**(right = **0**; right < numrights; right++)

{

strcpy(buffer, leftstr[left]);

squeeze2(buffer, rightstr[right]);

printf("[%s] - [%s] = [%s]\n", leftstr[left], rightstr[right], buffer);

}

}

**return** **0**;

}

**Answer to Exercise 2-5, page 48**

*Write the function any(s1,s2) , which returns the first location in the string s1 where any character from the string s2 occurs, or -1 if s1 contains no characters from s2 . (The standard library function strpbrk does the same job but returns a pointer to the location.)*   
  
Here is my solution, which is very simple but quite naive and inefficient. It has a worst-case time complexity of O(nm) where n and m are the lengths of the two strings.

/\*

\* Exercise 2-5 Page 48

\*

\* Write the function any(s1,s2), which returns the first location

\* in the string s1 where any character from the string s2 occurs,

\* or -1 if s1 contains no characters from s2. (The standard library

\* function strpbrk does the same job but returns a pointer to the

\* location.)

\*

\*/

**int** any(**char** s1[], **char** s2[])

{

**int** i;

**int** j;

**int** pos;

pos = -**1**;

**for**(i = **0**; pos == -**1** && s1[i] != **'\0'**; i++)

{

**for**(j = **0**; pos == -**1** && s2[j] != **'\0'**; j++)

{

**if**(s2[j] == s1[i])

{

pos = i;

}

}

}

**return** pos;

}

/\* test driver \*/

/\* We get a helpful boost for testing from the question text, because we are

\* told that the function's behaviour is identical to strpbrk except that it

\* returns a pointer instead of a position. We use this fact to validate the

\* function's correctness.

\*/

**#include <stdio.h>**

**#include <string.h>**

**int** main(**void**)

{

**char** \*leftstr[] =

{

"",

"a",

"antidisestablishmentarianism",

"beautifications",

"characteristically",

"deterministically",

"electroencephalography",

"familiarisation",

"gastrointestinal",

"heterogeneousness",

"incomprehensibility",

"justifications",

"knowledgeable",

"lexicographically",

"microarchitectures",

"nondeterministically",

"organizationally",

"phenomenologically",

"quantifications",

"representationally",

"straightforwardness",

"telecommunications",

"uncontrollability",

"vulnerabilities",

"wholeheartedly",

"xylophonically",

"youthfulness",

"zoologically"

};

**char** \*rightstr[] =

{

"",

"a",

"the",

"quick",

"brown",

"dog",

"jumps",

"over",

"lazy",

"fox",

"get",

"rid",

"of",

"windows",

"and",

"install",

"linux"

};

size\_t numlefts = **sizeof** leftstr / **sizeof** leftstr[**0**];

size\_t numrights = **sizeof** rightstr / **sizeof** rightstr[**0**];

size\_t left = **0**;

size\_t right = **0**;

**int** passed = **0**;

**int** failed = **0**;

**int** pos = -**1**;

**char** \*ptr = NULL;

**for**(left = **0**; left < numlefts; left++)

{

**for**(right = **0**; right < numrights; right++)

{

pos = any(leftstr[left], rightstr[right]);

ptr = strpbrk(leftstr[left], rightstr[right]);

**if**(-**1** == pos)

{

**if**(ptr != NULL)

{

printf("Test %d/%d failed.\n", left, right);

++failed;

}

**else**

{

printf("Test %d/%d passed.\n", left, right);

++passed;

}

}

**else**

{

**if**(ptr == NULL)

{

printf("Test %d/%d failed.\n", left, right);

++failed;

}

**else**

{

**if**(ptr - leftstr[left] == pos)

{

printf("Test %d/%d passed.\n", left, right);

++passed;

}

**else**

{

printf("Test %d/%d failed.\n", left, right);

++failed;

}

}

}

}

}

printf("\n\nTotal passes %d, fails %d, total tests %d\n",

passed,

failed,

passed + failed);

**return** **0**;

}

Here's a much better solution, by Partha Seetala. This solution has a worst- case time complexity of only O(n + m) which is considerably better.   
  
It works in a very interesting way. He first defines an array with one element for each possible character in the character set, and then takes the *second* string and 'ticks' the array at each position where the second string contains the character corresponding to that position. It's then a simple matter to loop through the first string, quitting as soon as he hits a 'ticked' position in the array.

**#include <stdio.h> /\* for NULL \*/**

**int** any(**char** \*s1, **char** \*s2)

{

**char** array[**256**]; /\* rjh comments

\* (a) by making this char array[256] = {0}; the first loop becomes unnecessary.

\* (b) for full ANSIness, #include <limits.h>, make the array unsigned char,

\* cast as required, and specify an array size of UCHAR\_MAX + 1.

\* (c) the return statements' (parentheses) are not required.

\*/

**int** i;

**if** (s1 == NULL) {

**if** (s2 == NULL) {

**return**(**0**);

} **else** {

**return**(-**1**);

}

}

**for**(i = **0**; i < **256**; i++) {

array[i] = **0**;

}

**while**(\*s2 != **'\0'**) {

array[\*s2] = **1**;

s2++;

}

i = **0**;

**while**(s1[i] != **'\0'**) {

**if** (array[s1[i]] == **1**) {

**return**(i);

}

i++;

}

**return**(-**1**);

}

/\* test driver by Richard Heathfield \*/

/\* We get a helpful boost for testing from the question text, because we are

\* told that the function's behaviour is identical to strpbrk except that it

\* returns a pointer instead of a position. We use this fact to validate the

\* function's correctness.

\*/

**#include <string.h>**

**int** main(**void**)

{

**char** \*leftstr[] =

{

"",

"a",

"antidisestablishmentarianism",

"beautifications",

"characteristically",

"deterministically",

"electroencephalography",

"familiarisation",

"gastrointestinal",

"heterogeneousness",

"incomprehensibility",

"justifications",

"knowledgeable",

"lexicographically",

"microarchitectures",

"nondeterministically",

"organizationally",

"phenomenologically",

"quantifications",

"representationally",

"straightforwardness",

"telecommunications",

"uncontrollability",

"vulnerabilities",

"wholeheartedly",

"xylophonically",

"youthfulness",

"zoologically"

};

**char** \*rightstr[] =

{

"",

"a",

"the",

"quick",

"brown",

"dog",

"jumps",

"over",

"lazy",

"fox",

"get",

"rid",

"of",

"windows",

"and",

"install",

"linux"

};

size\_t numlefts = **sizeof** leftstr / **sizeof** leftstr[**0**];

size\_t numrights = **sizeof** rightstr / **sizeof** rightstr[**0**];

size\_t left = **0**;

size\_t right = **0**;

**int** passed = **0**;

**int** failed = **0**;

**int** pos = -**1**;

**char** \*ptr = NULL;

**for**(left = **0**; left < numlefts; left++)

{

**for**(right = **0**; right < numrights; right++)

{

pos = any(leftstr[left], rightstr[right]);

ptr = strpbrk(leftstr[left], rightstr[right]);

**if**(-**1** == pos)

{

**if**(ptr != NULL)

{

printf("Test %d/%d failed.\n", left, right);

++failed;

}

**else**

{

printf("Test %d/%d passed.\n", left, right);

++passed;

}

}

**else**

{

**if**(ptr == NULL)

{

printf("Test %d/%d failed.\n", left, right);

++failed;

}

**else**

{

**if**(ptr - leftstr[left] == pos)

{

printf("Test %d/%d passed.\n", left, right);

++passed;

}

**else**

{

printf("Test %d/%d failed.\n", left, right);

++failed;

}

}

}

}

}

printf("\n\nTotal passes %d, fails %d, total tests %d\n",

passed,

failed,

passed + failed);

**return** **0**;

}

**Answer to Exercise 2-6, page 49**

*Write a function setbits(x,p,n,y) that returns x with the n bits that begin at position p set to the rightmost n bits of y, leaving the other bits unchanged.*   
  
This one's scary.

**#include <stdio.h>**

**unsigned** setbits(**unsigned** x, **int** p, **int** n, **unsigned** y)

{

**return** (x & ((~**0** << (p + **1**)) | (~(~**0** << (p + **1** - n))))) | ((y & ~(~**0** << n)) << (p + **1** - n));

}

**int** main(**void**)

{

**unsigned** i;

**unsigned** j;

**unsigned** k;

**int** p;

**int** n;

**for**(i = **0**; i < **30000**; i += **511**)

{

**for**(j = **0**; j < **1000**; j += **37**)

{

**for**(p = **0**; p < **16**; p++)

{

**for**(n = **1**; n <= p + **1**; n++)

{

k = setbits(i, p, n, j);

printf("setbits(%u, %d, %d, %u) = %u\n", i, p, n, j, k);

}

}

}

}

**return** **0**;

}

**Answer to Exercise 2-7, page 49**

*Write a function invert(x,p,n) that returns x with the n bits that begin at position p inverted (i.e., 1 changed into 0 and vice versa), leaving the others unchanged.*

**unsigned** invert(**unsigned** x, **int** p, **int** n)

{

**return** x ^ (~(~**0**U << n) << p);

}

/\* main driver added, in a hurry while tired, by RJH. Better test driver suggestions are welcomed! \*/

**#include <stdio.h>**

**int** main(**void**)

{

**unsigned** x;

**int** p, n;

**for**(x = **0**; x < **700**; x += **49**)

**for**(n = **1**; n < **8**; n++)

**for**(p = **1**; p < **8**; p++)

printf("%u, %d, %d: %u\n", x, n, p, invert(x, n, p));

**return** **0**;

}

**Answer to Exercise 2-8, page 49**

*Write a function rightrot(x,n) that returns the value of the integer x rotated to the right by n bit positions.*   
  
Greg's Cat 0 solution

**unsigned** rightrot(**unsigned** x, **unsigned** n)

{

**while** (n > **0**) {

**if** ((x & **1**) == **1**)

x = (x >> **1**) | ~(~**0**U >> **1**);

**else**

x = (x >> **1**);

n--;

}

**return** x;

}

/\* main driver added, in a hurry while tired, by RJH. Better test driver suggestions are welcomed! \*/

**#include <stdio.h>**

**int** main(**void**)

{

**unsigned** x;

**int** n;

**for**(x = **0**; x < **700**; x += **49**)

**for**(n = **1**; n < **8**; n++)

printf("%u, %d: %u\n", x, n, rightrot(x, n));

**return** **0**;

}

Here's Bob Wightman's Cat 1 solution:

/\* K&R exercise 2-8

It is class 1 due to the /sizeof/ operator (CHAR\_BIT is introduced with

<limits.h> in Chapter 1). I could have used the conditional operator but

thought that this is clearer.

Notes:

1. Implicit int removed (not absolutely necessary but...)

2. Checks for the size of the shift and reduces it to the range 0 -

(number of bits in an int) - 1 This is to avoid right shifting the

number into oblivion.

3. If either the value or the shift is zero then nothing need to be done

to the parameter so just return it.

\*/

**unsigned** **int** rightrot(**unsigned** **int** x, **unsigned** **int** n)

{

/\* calculate number of bits in type \*/

size\_t s = **sizeof**(x) \* CHAR\_BIT;

size\_t p;

/\* limit shift to range 0 - (s - 1) \*/

**if**(n < s)

p = n;

**else**

p = n % s;

/\* if either is zero then the original value is unchanged \*/

**if**((**0** == x) || (**0** == p))

**return** x;

**return** (x >> p) | (x << (s - p));

}

/\* Driver based on yours but runs the shift values beyond the size of an

unsigned integer on any system \*/

**int** main(**void**)

{

**unsigned** **int** val;

**unsigned** **int** pos;

**unsigned** **int** max = **sizeof** (pos) \* CHAR\_BIT + **1**;

**for**(val = **0**; val < **700**; val += **49**)

{

**for**(pos = **0**; pos < max; ++pos)

{

printf("%u, %d: %u\n", x, n, rightrot(val, pos));

}

}

}

**Answer to Exercise 2-9, page 51**

*In a two's complement number system, x &= (x-1) deletes the rightmost 1-bit in x . Explain why. Use this observation to write a faster version of bitcount .*   
  
bitcount is written on p.50 as this:

/\* bitcount: count 1 bits in x \*/

**int** bitcount(**unsigned** x)

{

**int** b;

**for** (b = **0**; x != **0**; x >>= **1**)

**if** (x & **01**)

b++;

**return** b;

}

Answer: If x is odd, then (x-1) has the same bit representation as x except that the rightmost 1-bit is now a 0. In this case, (x & (x-1)) == (x-1). If x is even, then the representation of (x-1) has the rightmost zeros of x becoming ones and the rightmost one becoming a zero. Anding the two clears the rightmost 1-bit in x and all the rightmost 1-bits from (x-1). Here's the new version of bitcount:

/\* bitcount: count 1 bits in x \*/

**int** bitcount(**unsigned** x)

{

**int** b;

**for** (b = **0**; x != **0**; x &= (x-**1**))

b++;

**return** b;

}

**Answer to Exercise 2-10, page 52**

*Rewrite the function lower, which converts upper case letters to lower case, with a conditional expression instead of if-else .*

/\*

Exercise 2-10. Rewrite the function lower, which converts upper case letters

to lower case, with a conditional expression instead of if-else.

Assumptions : by conditional expression they mean an expression involving a ternary operator.

Author: Bryan Williams

\*/

**#include <stdio.h>**

**#include <string.h>**

**#define TEST**

**#define ORIGINAL 0**

**#define SOLUTION 1**

**#define PORTABLE\_SOLUTION 0**

/\*

ok, the original routine we are trying to convert looks like this..

\*/

**#if ORIGINAL**

/\* lower: convert c to lower case; ASCII only \*/

**int** lower(**int** c)

{

**if**(c >= **'A'** && c <= **'Z'**)

**return** c + **'a'** - **'A'**;

**else**

**return** c;

}

**#endif**

/\*

the natural solution for simply making this a conditional (ternary) return instead of an

if ... else ...

\*/

**#if SOLUTION**

/\* lower: convert c to lower case; ASCII only \*/

**int** lower(**int** c)

{

**return** c >= **'A'** && c <= **'Z'** ? c + **'a'** - **'A'** : c;

}

**#endif**

/\*

the more portable solution, requiring string.h for strchr but keeping the idea of a

conditional return.

\*/

**#if PORTABLE\_SOLUTION**

/\* lower: convert c to lower case \*/

**int** lower(**int** c)

{

**char** \*Uppercase = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

**char** \*Lowercase = "abcdefghijklmnopqrstuvwxyz";

**char** \*p = NULL;

**return** NULL == (p = strchr(Uppercase, c)) ? c : \*(Lowercase + (p - Uppercase));

}

**#endif**

/\*

ok, this bit is just a test driver... exclude as required

\*/

**#ifdef TEST**

**int** main(**void**)

{

**char** \*Tests = "AaBbcCD3EdFGHgIJKLhM2NOjPQRkSTlUVWfXYf0Z1";

**char** \*p = Tests;

**int** Result = **0**;

**while**(**'\0'** != \*p)

{

Result = lower(\*p);

printf("[%c] gives [%c]\n", \*p, Result);

++p;

}

/\* and the obligatory boundary test \*/

Result = lower(**0**);

printf("'\\0' gives %d\n", Result);

**return** **0**;

}

#endif

**Answer to Exercise 3-1, page 58**

*Our binary search makes two tests inside the loop, when one would suffice (at the price of more tests outside). Write a version with only one test inside the loop and measure the difference in run-time.*   
  
  
Paul Griffiths' solution (krx30100.c):

/\* Solution by Paul Griffiths (paul@paulgriffiths.demon.co.uk) \*/

/\*

EX3\_1.C

=======

Suggested solution to Exercise 3-1

\*/

**#include <stdio.h>**

**#include <time.h>**

**int** binsearch(**int** x, **int** v[], **int** n); /\* Original K&R function \*/

**int** binsearch2(**int** x, **int** v[], **int** n); /\* Our new function \*/

**#define MAX\_ELEMENT 20000**

/\* Outputs approximation of processor time required

for our two binary search functions. We search for

the element -1, to time the functions' worst case

performance (i.e. element not found in test data) \*/

**int** main(**void**) {

**int** testdata[MAX\_ELEMENT];

**int** index; /\* Index of found element in test data \*/

**int** n = -**1**; /\* Element to search for \*/

**int** i;

clock\_t time\_taken;

/\* Initialize test data \*/

**for** ( i = **0**; i < MAX\_ELEMENT; ++i )

testdata[i] = i;

/\* Output approximation of time taken for

100,000 iterations of binsearch() \*/

**for** ( i = **0**, time\_taken = clock(); i < **100000**; ++i ) {

index = binsearch(n, testdata, MAX\_ELEMENT);

}

time\_taken = clock() - time\_taken;

**if** ( index < **0** )

printf("Element %d not found.\n", n);

**else**

printf("Element %d found at index %d.\n", n, index);

printf("binsearch() took %lu clocks (%lu seconds)\n",

(**unsigned** **long**) time\_taken,

(**unsigned** **long**) time\_taken / CLOCKS\_PER\_SEC);

/\* Output approximation of time taken for

100,000 iterations of binsearch2() \*/

**for** ( i = **0**, time\_taken = clock(); i < **100000**; ++i ) {

index = binsearch2(n, testdata, MAX\_ELEMENT);

}

time\_taken = clock() - time\_taken;

**if** ( index < **0** )

printf("Element %d not found.\n", n);

**else**

printf("Element %d found at index %d.\n", n, index);

printf("binsearch2() took %lu clocks (%lu seconds)\n",

(**unsigned** **long**) time\_taken,

(**unsigned** **long**) time\_taken / CLOCKS\_PER\_SEC);

**return** **0**;

}

/\* Performs a binary search for element x

in array v[], which has n elements \*/

**int** binsearch(**int** x, **int** v[], **int** n) {

**int** low, mid, high;

low = **0**;

high = n - **1**;

**while** ( low <= high ) {

mid = (low+high) / **2**;

**if** ( x < v[mid] )

high = mid - **1**;

**else** **if** ( x > v[mid] )

low = mid + **1**;

**else**

**return** mid;

}

**return** -**1**;

}

/\* Implementation of binsearch() using

only one test inside the loop \*/

**int** binsearch2(**int** x, **int** v[], **int** n) {

**int** low, high, mid;

low = **0**;

high = n - **1**;

mid = (low+high) / **2**;

**while** ( low <= high && x != v[mid] ) {

**if** ( x < v[mid] )

high = mid - **1**;

**else**

low = mid + **1**;

mid = (low+high) / **2**;

}

**if** ( x == v[mid] )

**return** mid;

**else**

**return** -**1**;

}

Colin Barker's solution (krx30101.c):

/\* Solution by Colin Barker (colin.barker@wanadoo.fr)

\* using the driver from the solution by Paul Griffiths.

\*/

/\*

EX3\_1.C

=======

Suggested solution to Exercise 3-1

\*/

**#include <stdio.h>**

**#include <time.h>**

**int** binsearch(**int** x, **int** v[], **int** n); /\* Original K&R function \*/

**int** binsearch2(**int** x, **int** v[], **int** n); /\* Our new function \*/

**#define MAX\_ELEMENT 20000**

/\* Outputs approximation of processor time required

for our two binary search functions. We search for

the element -1, to time the functions' worst case

performance (i.e. element not found in test data) \*/

**int** main(**void**) {

**int** testdata[MAX\_ELEMENT];

**int** index; /\* Index of found element in test data \*/

**int** n = -**1**; /\* Element to search for \*/

**int** i;

clock\_t time\_taken;

/\* Initialize test data \*/

**for** ( i = **0**; i < MAX\_ELEMENT; ++i )

testdata[i] = i;

/\* Output approximation of time taken for

100,000 iterations of binsearch() \*/

**for** ( i = **0**, time\_taken = clock(); i < **100000**; ++i ) {

index = binsearch(n, testdata, MAX\_ELEMENT);

}

time\_taken = clock() - time\_taken;

**if** ( index < **0** )

printf("Element %d not found.\n", n);

**else**

printf("Element %d found at index %d.\n", n, index);

printf("binsearch() took %lu clocks (%lu seconds)\n",

(**unsigned** **long**) time\_taken,

(**unsigned** **long**) time\_taken / CLOCKS\_PER\_SEC);

/\* Output approximation of time taken for

100,000 iterations of binsearch2() \*/

**for** ( i = **0**, time\_taken = clock(); i < **100000**; ++i ) {

index = binsearch2(n, testdata, MAX\_ELEMENT);

}

time\_taken = clock() - time\_taken;

**if** ( index < **0** )

printf("Element %d not found.\n", n);

**else**

printf("Element %d found at index %d.\n", n, index);

printf("binsearch2() took %lu clocks (%lu seconds)\n",

(**unsigned** **long**) time\_taken,

(**unsigned** **long**) time\_taken / CLOCKS\_PER\_SEC);

**return** **0**;

}

/\* Performs a binary search for element x

in array v[], which has n elements \*/

**int** binsearch(**int** x, **int** v[], **int** n) {

**int** low, mid, high;

low = **0**;

high = n - **1**;

**while** ( low <= high ) {

mid = (low+high) / **2**;

**if** ( x < v[mid] )

high = mid - **1**;

**else** **if** ( x > v[mid] )

low = mid + **1**;

**else**

**return** mid;

}

**return** -**1**;

}

**int** binsearch2(**int** x, **int** v[], **int** n)

{

**int** low, high, mid;

low = -**1**;

high = n;

**while** (low + **1** < high) {

mid = (low + high) / **2**;

**if** (v[mid] < x)

low = mid;

**else**

high = mid;

}

**if** (high == n || v[high] != x)

**return** -**1**;

**else**

**return** high;

}

Andrew Tesker's solution (krx30102.c):

/\* Andrew Tesker

\*

\* krx30102.c

\*/

**#include <stdio.h>**

/\* find x in v[] \*/

**int** binsearch(**int** x, **int** v[], **int** n);

/\*

The main is here for the purpose of a built in test

\*/

**int** main(**void**)

{

**int** test[]={**1**,**3**,**5**,**7**,**9**,**11**,**13**};

**int** i;

**for**(i=(**sizeof**(test)/**sizeof**(**int**))-**1**; i>=**0**; --i)

printf("looking for %d. Index=%d\n",test[i],binsearch(test[i], test, **sizeof**(test)/**sizeof**(\*test)));

**return** **0**;

}

/\* n = size of array v \*/

**int** binsearch(**int** x, **int** v[], **int** n)

{

**int** low, high, mid;

low = **0**;

high = n-**1**;

**while**(low < high) {

mid = (low+high)/**2**;

**if**(x <= v[mid])

high=mid;

**else**

low = mid+**1**;

}

**return** (x == v[low])?low : -**1**;

}

**Answer to Exercise 3-2, page 60**

*Write a function escape(s,t) that converts characters like newline and tab into visible escape sequences like \n and \t as it copies the string t to s . Use a switch . Write a function for the other direction as well, converting escape sequences into the real characters.*

/\*

EX3\_2.C

=======

Suggested solution to Exercise 3-2

\*/

**#include <stdio.h>**

**void** escape(**char** \* s, **char** \* t);

**void** unescape(**char** \* s, **char** \* t);

**int** main(**void**) {

**char** text1[**50**] = "\aHello,\n\tWorld! Mistakee\b was \"Extra 'e'\"!\n";

**char** text2[**51**];

printf("Original string:\n%s\n", text1);

escape(text2, text1);

printf("Escaped string:\n%s\n", text2);

unescape(text1, text2);

printf("Unescaped string:\n%s\n", text1);

**return** **0**;

}

/\* Copies string t to string s, converting special

characters into their appropriate escape sequences.

The "complete set of escape sequences" found in

K&R Chapter 2 is used, with the exception of:

\? \' \ooo \xhh

as these can be typed directly into the source code,

(i.e. without using the escape sequences themselves)

and translating them is therefore ambiguous. \*/

**void** escape(**char** \* s, **char** \* t) {

**int** i, j;

i = j = **0**;

**while** ( t[i] ) {

/\* Translate the special character, if we have one \*/

**switch**( t[i] ) {

**case** **'\n'**:

s[j++] = **'\\'**;

s[j] = **'n'**;

**break**;

**case** **'\t'**:

s[j++] = **'\\'**;

s[j] = **'t'**;

**break**;

**case** **'\a'**:

s[j++] = **'\\'**;

s[j] = **'a'**;

**break**;

**case** **'\b'**:

s[j++] = **'\\'**;

s[j] = **'b'**;

**break**;

**case** **'\f'**:

s[j++] = **'\\'**;

s[j] = **'f'**;

**break**;

**case** **'\r'**:

s[j++] = **'\\'**;

s[j] = **'r'**;

**break**;

**case** **'\v'**:

s[j++] = **'\\'**;

s[j] = **'v'**;

**break**;

**case** **'\\'**:

s[j++] = **'\\'**;

s[j] = **'\\'**;

**break**;

**case** **'\"'**:

s[j++] = **'\\'**;

s[j] = **'\"'**;

**break**;

**default**:

/\* This is not a special character, so just copy it \*/

s[j] = t[i];

**break**;

}

++i;

++j;

}

s[j] = t[i]; /\* Don't forget the null character \*/

}

/\* Copies string t to string s, converting escape sequences

into their appropriate special characters. See the comment

for escape() for remarks regarding which escape sequences

are translated. \*/

**void** unescape(**char** \* s, **char** \* t) {

**int** i, j;

i = j = **0**;

**while** ( t[i] ) {

**switch** ( t[i] ) {

**case** **'\\'**:

/\* We've found an escape sequence, so translate it \*/

**switch**( t[++i] ) {

**case** **'n'**:

s[j] = **'\n'**;

**break**;

**case** **'t'**:

s[j] = **'\t'**;

**break**;

**case** **'a'**:

s[j] = **'\a'**;

**break**;

**case** **'b'**:

s[j] = **'\b'**;

**break**;

**case** **'f'**:

s[j] = **'\f'**;

**break**;

**case** **'r'**:

s[j] = **'\r'**;

**break**;

**case** **'v'**:

s[j] = **'\v'**;

**break**;

**case** **'\\'**:

s[j] = **'\\'**;

**break**;

**case** **'\"'**:

s[j] = **'\"'**;

**break**;

**default**:

/\* We don't translate this escape

sequence, so just copy it verbatim \*/

s[j++] = **'\\'**;

s[j] = t[i];

}

**break**;

**default**:

/\* Not an escape sequence, so just copy the character \*/

s[j] = t[i];

}

++i;

++j;

}

s[j] = t[i]; /\* Don't forget the null character \*/

}

**Answer to Exercise 3-3, page 63**

*Write a function expand(s1,s2) that expands shorthand notations like a-z in the string s1 into the equivalent complete list abc...xyz in s2 . Allow for letters of either case and digits, and be prepared to handle cases like a-b-c and a-z0-9 and -a-z . Arrange that a leading or trailing - is taken literally.*

/\*

EX3\_3.C

=======

Suggested solution to Exercise 3-3

\*/

**#include <stdio.h>**

**#include <string.h>**

**void** expand(**char** \* s1, **char** \* s2);

**int** main(**void**) {

**char** \*s[] = { "a-z-", "z-a-", "-1-6-",

"a-ee-a", "a-R-L", "1-9-1",

"5-5", NULL };

**char** result[**100**];

**int** i = **0**;

**while** ( s[i] ) {

/\* Expand and print the next string in our array s[] \*/

expand(result, s[i]);

printf("Unexpanded: %s\n", s[i]);

printf("Expanded : %s\n", result);

++i;

}

**return** **0**;

}

/\* Copies string s2 to s1, expanding

ranges such as 'a-z' and '8-3' \*/

**void** expand(**char** \* s1, **char** \* s2) {

**static** **char** upper\_alph[**27**] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

**static** **char** lower\_alph[**27**] = "abcdefghijklmnopqrstuvwxyz";

**static** **char** digits[**11**] = "0123456789";

**char** \* start, \* end, \* p;

**int** i = **0**;

**int** j = **0**;

/\* Loop through characters in s2 \*/

**while** ( s2[i] ) {

**switch**( s2[i] ) {

**case** **'-'**:

**if** ( i == **0** || s2[i+**1**] == **'\0'** ) {

/\* '-' is leading or trailing, so just copy it \*/

s1[j++] = **'-'**;

++i;

**break**;

}

**else** {

/\* We have a "range" to extrapolate. Test whether

the two operands are part of the same range. If

so, store pointers to the first and last characters

in the range in start and end, respectively. If

not, output and error message and skip this range. \*/

**if** ( (start = strchr(upper\_alph, s2[i-**1**])) &&

(end = strchr(upper\_alph, s2[i+**1**])) )

;

**else** **if** ( (start = strchr(lower\_alph, s2[i-**1**])) &&

(end = strchr(lower\_alph, s2[i+**1**])) )

;

**else** **if** ( (start = strchr(digits, s2[i-**1**])) &&

(end = strchr(digits, s2[i+**1**])) )

;

**else** {

/\* We have mismatched operands in the range,

such as 'a-R', or '3-X', so output an error

message, and just copy the range expression. \*/

fprintf(stderr, "EX3\_3: Mismatched operands '%c-%c'\n",

s2[i-**1**], s2[i+**1**]);

s1[j++] = s2[i-**1**];

s1[j++] = s2[i++];

**break**;

}

/\* Expand the range \*/

p = start;

**while** ( p != end ) {

s1[j++] = \*p;

**if** ( end > start )

++p;

**else**

--p;

}

s1[j++] = \*p;

i += **2**;

}

**break**;

**default**:

**if** ( s2[i+**1**] == **'-'** && s2[i+**2**] != **'\0'** ) {

/\* This character is the first operand in

a range, so just skip it - the range will

be processed in the next iteration of

the loop. \*/

++i;

}

**else** {

/\* Just a normal character, so copy it \*/

s1[j++] = s2[i++];

}

**break**;

}

}

s1[j] = s2[i]; /\* Don't forget the null character \*/

}

**Answer to Exercise 3-4, page 64**

Wayne Lubin's query involved Paul's discussion of two's complement. The text has now been corrected (by Paul).   
  
*In a two's complement number representation, our version of itoa does not handle the largest negative number, that is, the value of n equal to -(2 to the power (wordsize - 1)) . Explain why not. Modify it to print that value correctly regardless of the machine on which it runs.*   
  
Exercise 3-4 explanation: There are a number of ways of representing signed integers in binary, for example, signed-magnitude, excess-M, one's complement and two's complement. We shall restrict our discussion to the latter two. In a one's complement number representation, the binary represenation of a negative number is simply the binary representation of its positive counterpart, with the sign of all the bits switched. For instance, with 8 bit variables:

SIGNED BINARY UNSIGNED

25 00011001 25

-25 11100110 230

127 01111111 127

-127 10000000 128

The implications of this are (amongst others) that there are two ways of representing zero (all zero bits, and all one bits), that the maximum range for a signed 8-bit number is -127 to 127, and that negative numbers are biased by (2^n - 1) (i.e. -I is represented by (2^n - 1) - (+I). In our example, so:

Bias = 2^8 - 1 = 255 = 11111111

Subtract 25 = 00011001

Equals = 11100110

In a two's complement representation, negative numbers are biased by 2^n, e.g.:

Bias = 2^8 = 100000000

Subtract 25 = 00011001

Equals = 11100111

In other words, to find the two's complement representation of a negative number, find the one's complement of it, and add one. The important thing to notice is that the range of an 8 bit variable using a two's complement representation is -128 to 127, as opposed to -127 to 127 using one's complement. Thus, the absolute value of the largest negative number cannot be represented (i.e. we cannot represent +128). Since the itoa() function in Chapter 3 handles negative numbers by reversing the sign of the number before processing, then adding a '-' to the string, passing the largest negative number will result it in being translated to itself:

-128 : 10000000

One's complement: 01111111

Subtract 1 : 10000000

Therefore, because (n /= 10) will be negative, the do-while loop will run once only, and will place in the string a '-', followed by a single character, (INT\_MIN % 10 + '0'). We can remedy these two bugs in the following way: 1 - change 'while ((n /= 10) > 0)' to 'while (n /= 10)'. Since any fractional part is truncated with integer division, n will eventually equal zero after successive divides by 10, and 'n /= 10' will evaluate to false sooner or later. 2 - change 'n % 10 + '0'' to 'abs(n % 10) + '0'', to get the correct character. EX3\_4.C shows the revised function, which will run correctly regardless of the number representation.

/\*

EX3\_4.C

=======

Suggested solution to Exercise 3-4

\*/

**#include <stdlib.h>**

**#include <stdio.h>**

**#include <limits.h>**

**void** itoa(**int** n, **char** s[]);

**void** reverse(**char** s[]);

**int** main(**void**) {

**char** buffer[**20**];

printf("INT\_MIN: %d\n", INT\_MIN);

itoa(INT\_MIN, buffer);

printf("Buffer : %s\n", buffer);

**return** **0**;

}

**void** itoa(**int** n, **char** s[]) {

**int** i, sign;

sign = n;

i = **0**;

**do** {

s[i++] = abs(n % **10**) + **'0'**;

} **while** ( n /= **10** );

**if** (sign < **0**)

s[i++] = **'-'**;

s[i] = **'\0'**;

reverse(s);

}

**void** reverse(**char** s[]) {

**int** c, i, j;

**for** ( i = **0**, j = strlen(s)-**1**; i < j; i++, j--) {

c = s[i];

s[i] = s[j];

s[j] = c;

}

}

**Answer to Exercise 3-5, page 64**

*Write the function itob(n,s,b) that converts the integer n into a base b character representation in the string s . In particular, itob(n,s,16) formats n as a hexadecimal integer in s .*

/\*

EX3\_5.C

=======

Suggested solution to Exercise 3-5

\*/

**#include <stdlib.h>**

**#include <stdio.h>**

**void** itob(**int** n, **char** s[], **int** b);

**void** reverse(**char** s[]);

**int** main(**void**) {

**char** buffer[**10**];

**int** i;

**for** ( i = **2**; i <= **20**; ++i ) {

itob(**255**, buffer, i);

printf("Decimal 255 in base %-2d : %s\n", i, buffer);

}

**return** **0**;

}

/\* Stores a string representation of integer n

in s[], using a numerical base of b. Will handle

up to base-36 before we run out of digits to use. \*/

**void** itob(**int** n, **char** s[], **int** b) {

**static** **char** digits[] = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ";

**int** i, sign;

**if** ( b < **2** || b > **36** ) {

fprintf(stderr, "EX3\_5: Cannot support base %d\n", b);

exit(EXIT\_FAILURE);

}

**if** ((sign = n) < **0**)

n = -n;

i = **0**;

**do** {

s[i++] = digits[n % b];

} **while** ((n /= b) > **0**);

**if** (sign < **0**)

s[i++] = **'-'**;

s[i] = **'\0'**;

reverse(s);

}

/\* Reverses string s[] in place \*/

**void** reverse(**char** s[]) {

**int** c, i, j;

**for** ( i = **0**, j = strlen(s)-**1**; i < j; i++, j--) {

c = s[i];

s[i] = s[j];

s[j] = c;

}

}

**Answer to Exercise 3-6, page 64**

*Write a version of itoa that accepts three arguments instead of two. The third argument is a minimum field width; the converted number must be padded with blanks on the left if necessary to make it wide enough.*

/\*

EX3\_6.C

=======

Suggested solution to Exercise 3-6

\*/

**#include <stdio.h>**

**#include <limits.h>**

**void** itoa(**int** n, **char** s[], **int** width);

**void** reverse(**char** s[]);

**int** main(**void**) {

**char** buffer[**20**];

itoa(INT\_MIN, buffer, **7**);

printf("Buffer:%s\n", buffer);

**return** **0**;

}

**void** itoa(**int** n, **char** s[], **int** width) {

**int** i, sign;

**if** ((sign = n) < **0**)

n = -n;

i = **0**;

**do** {

s[i++] = n % **10** + **'0'**;

printf("%d %% %d + '0' = %d\n", n, **10**, s[i-**1**]);

} **while** ((n /= **10**) > **0**);

**if** (sign < **0**)

s[i++] = **'-'**;

**while** (i < width ) /\* Only addition to original function \*/

s[i++] = **' '**;

s[i] = **'\0'**;

reverse(s);

}

**void** reverse(**char** s[]) {

**int** c, i, j;

**for** ( i = **0**, j = strlen(s)-**1**; i < j; i++, j--) {

c = s[i];

s[i] = s[j];

s[j] = c;

}

}

**Answer to Exercise 4-1, page 71**

*Write the function strrindex(s,t) , which returns the position of the rightmost occurrence of t in s , or -1 if there is none.*

/\* Test driver by Richard Heathfield

\* Solution (strrindex function) by Rick Dearman

\*/

**#include <stdio.h>**

/\* Write the function strrindex(s,t), which returns the position

\*\* of the rightmost occurrence of t in s, or -1 if there is none.

\*/

**int** strrindex( **char** s[], **char** t )

{

**int** i;

**int** count = -**1**;

**for**(i=**0**; s[i] != **'\0'**; i++)

{

**if**(s[i] == t)

{

count = i;

}

}

**return** count;

}

**typedef** **struct** TEST

{

**char** \*data;

**char** testchar;

**int** expected;

} TEST;

**int** main(**void**)

{

TEST test[] =

{

{"Hello world", **'o'**, **7**},

{"This string is littered with iiiis", **'i'**, **32**},

{"No 'see' letters in here", **'c'**, -**1**}

};

size\_t numtests = **sizeof** test / **sizeof** test[**0**];

size\_t i;

**char** ch = **'o'**;

**int** pos;

**for**(i = **0**; i < numtests; i++)

{

pos = strrindex(test[i].data, test[i].testchar);

printf("Searching %s for last occurrence of %c.\n",

test[i].data,

test[i].testchar);

printf("Expected result: %d\n", test[i].expected);

printf("%sorrect (%d).\n", pos == test[i].expected ? "C" : "Inc", pos);

**if**(pos != -**1**)

{

printf("Character found was %c\n", test[i].data[pos]);

}

}

**return** **0**;

}

**Answer to Exercise 4-2, page 73**

*Extend atof to handle scientific notation of the form 123.45e-6 where a floating-point number may be followed by e or E and an optionally signed exponent.*

/\*

\*\* Written by Dann Corbit as K&R 2, Exercise 4-2 (Page 73).

\*\* Keep in mind that this is \*JUST\* a student exercise, and is

\*\* light years away from being robust.

\*\*

\*\* Actually, it's kind of embarassing, but I'm too lazy to fix it.

\*\*

\*\* Caveat Emptor, not my fault if demons fly out of your nose,

\*\* and all of that.

\*/

**#include <ctype.h>**

**#include <limits.h>**

**#include <float.h>**

**#include <signal.h>**

**#include <stdio.h>**

**int** my\_atof(**char** \*string, **double** \*pnumber)

{

/\* Convert char string to double data type. \*/

**double** retval;

**double** one\_tenth = **0.1**;

**double** ten = **10.0**;

**double** zero = **0.0**;

**int** found\_digits = **0**;

**int** is\_negative = **0**;

**char** \*num;

/\* Check pointers. \*/

**if** (pnumber == **0**) {

**return** **0**;

}

**if** (string == **0**) {

\*pnumber = zero;

**return** **0**;

}

retval = zero;

num = string;

/\* Advance past white space. \*/

**while** (isspace(\*num))

num++;

/\* Check for sign. \*/

**if** (\*num == **'+'**)

num++;

**else** **if** (\*num == **'-'**) {

is\_negative = **1**;

num++;

}

/\* Calculate the integer part. \*/

**while** (isdigit(\*num)) {

found\_digits = **1**;

retval \*= ten;

retval += \*num - **'0'**;

num++;

}

/\* Calculate the fractional part. \*/

**if** (\*num == **'.'**) {

**double** scale = one\_tenth;

num++;

**while** (isdigit(\*num)) {

found\_digits = **1**;

retval += scale \* (\*num - **'0'**);

num++;

scale \*= one\_tenth;

}

}

/\* If this is not a number, return error condition. \*/

**if** (!found\_digits) {

\*pnumber = zero;

**return** **0**;

}

/\* If all digits of integer & fractional part are 0, return 0.0 \*/

**if** (retval == zero) {

\*pnumber = zero;

**return** **1**; /\* Not an error condition, and no need to

\* continue. \*/

}

/\* Process the exponent (if any) \*/

**if** ((\*num == **'e'**) || (\*num == **'E'**)) {

**int** neg\_exponent = **0**;

**int** get\_out = **0**;

**long** index;

**long** exponent = **0**;

**double** getting\_too\_big = DBL\_MAX \* one\_tenth;

**double** getting\_too\_small = DBL\_MIN \* ten;

num++;

**if** (\*num == **'+'**)

num++;

**else** **if** (\*num == **'-'**) {

num++;

neg\_exponent = **1**;

}

/\* What if the exponent is empty? Return the current result. \*/

**if** (!isdigit(\*num)) {

**if** (is\_negative)

retval = -retval;

\*pnumber = retval;

**return** (**1**);

}

/\* Convert char exponent to number <= 2 billion. \*/

**while** (isdigit(\*num) && (exponent < LONG\_MAX / **10**)) {

exponent \*= **10**;

exponent += \*num - **'0'**;

num++;

}

/\* Compensate for the exponent. \*/

**if** (neg\_exponent) {

**for** (index = **1**; index <= exponent && !get\_out; index++)

**if** (retval < getting\_too\_small) {

get\_out = **1**;

retval = DBL\_MIN;

} **else**

retval \*= one\_tenth;

} **else**

**for** (index = **1**; index <= exponent && !get\_out; index++) {

**if** (retval > getting\_too\_big) {

get\_out = **1**;

retval = DBL\_MAX;

} **else**

retval \*= ten;

}

}

**if** (is\_negative)

retval = -retval;

\*pnumber = retval;

**return** (**1**);

}

/\*

\*\* Lame and evil wrapper function to give the exercise the requested

\*\* interface. Dann Corbit will plead innocent to the end.

\*\* It's very existence means that the code is not conforming.

\*\* Pretend you are a C library implementer, OK? But you would fix

\*\* all those bleeding gaps, I am sure.

\*/

**double** atof(**char** \*s)

{

**double** d = **0.0**;

**if** (!my\_atof(s, &d))

{

**#ifdef DEBUG**

fputs("Error converting string in [sic] atof()", stderr);

**#endif**

raise(SIGFPE);

}

**return** d;

}

**#ifdef UNIT\_TEST**

**char** \*strings[] = {

"1.0e43",

"999.999",

"123.456e-9",

"-1.2e-3",

"1.2e-3",

"-1.2E3",

"-1.2e03",

"cat",

"",

**0**

};

**int** main(**void**)

{

**int** i = **0**;

**for** (; \*strings[i]; i++)

printf("atof(%s) = %g\n", strings[i], atof(strings[i]));

**return** **0**;

}

#endif

**Answer to Exercise 4-3, page 79**

*Given the basic framework, it's straightforward to extend the calculator. Add the modulus ( % ) operator and provisions for negative numbers.*   
  
In Bob's words: "Here's my attempt Adding the modulus is easily done by another case in main and using the fmod function. The standard library has been mentioned at this point so it should be valid to use this for type 0 compliance. math.h should be added to the list of #includes for fmod."

**int** main(**void**)

{

**int** type;

**double** op2;

**char** s[MAXOP];

**int** flag = TRUE;

**while**((type = Getop(s)) != EOF)

{

**switch**(type)

{

/\* other cases snipped for brevity \*/

**case** **'%'**:

op2 = pop();

**if**(op2)

push(fmod(pop(), op2));

**else**

printf("\nError: Division by zero!");

**break**;

}

}

**return** EXIT\_SUCCESS;

}

Bob goes on to say: "Deal with unary minus when retrieving tokens. This is based on the fact that a unary minus will have no intervening space between it and its operand."

/\* Getop: get next operator or numeric operand. \*/

**int** Getop(**char** s[])

{

**#define PERIOD '.'**

**int** i = **0**;

**int** c;

**int** next;

/\* Skip whitespace \*/

**while**((s[**0**] = c = getch()) == **' '** || c == **'\t'**)

;

s[**1**] = **'\0'**;

/\* Not a number but may contain a unary minus. \*/

**if**(!isdigit(c) && c != PERIOD && c != **'-'**)

**return** c;

**if**(c == **'-'**)

{

next = getch();

**if**(!isdigit(next) && next != PERIOD)

{

**return** c;

}

c = next;

}

**else**

{

c = getch();

}

**while**(isdigit(s[++i] = c))

c = getch();

**if**(c == PERIOD) /\* Collect fraction part. \*/

**while**(isdigit(s[++i] = c = getch()))

;

s[i] = **'\0'**;

**if**(c != EOF)

unGetch(c);

**return** NUMBER;

}

**Answer to Exercise 4-4, page 79**

*Add commands to print the top element of the stack without popping, to duplicate it, and to swap the top two elements. Add a command to clear the stack.*

**#include<stdlib.h>**

**#include<stdio.h>**

**#include<ctype.h>**

**#include<math.h>**

**#define MAXOP 100**

**#define NUMBER 0**

**#define TRUE 1**

**#define FALSE 0**

/\* This programme is a basic calculator.

Extra cases have been added to:

1. Show the top item of the stack without permanently popping it.

2. Swap the top two items on the stack.

3. Duplicate the top item on the stack.

4. Clear the stack.

I have used functions for each of the new cases rather than have the

code inline in order to limit the physical size of the switch block.

In anticipation of the following exercise the following characters have

been used for the operations (in the same order as above): ? ~ # !

rather than use alphabetic characters.

It is actually rather difficult to be original in this exercise.

This is exercise 4-4 from Kernighan & Ritchie, page 79.

\*/

**int** Getop(**char** s[]);

**void** push(**double** val);

**double** pop(**void**);

**void** showTop(**void**);

**void** duplicate(**void**);

**void** swapItems(**void**);

**void** clearStack();

**int** main(**void**)

{

**int** type;

**double** op2;

**char** s[MAXOP];

**int** flag = TRUE;

**while**((type = Getop(s)) != EOF)

{

**switch**(type)

{

**case** NUMBER:

push(atof(s));

**break**;

**case** **'+'**:

push(pop() + pop());

**break**;

**case** **'\*'**:

push(pop() \* pop());

**break**;

**case** **'-'**:

op2 = pop();

push(pop()- op2);

**break**;

**case** **'/'**:

op2 = pop();

**if**(op2)

push(pop() / op2);

**else**

printf("\nError: division by zero!");

**break**;

**case** **'%'**:

op2 = pop();

**if**(op2)

push(fmod(pop(), op2));

**else**

printf("\nError: division by zero!");

**break**;

**case** **'?'**:

showTop();

**break**;

**case** **'#'**:

duplicate();

**break**;

**case** **'~'**:

swapItems();

**break**;

**case** **'!'**:

clearStack();

**case** **'\n'**:

printf("\n\t%.8g\n", pop());

**break**;

**default**:

printf("\nError: unknown command %s.\n", s);

**break**;

}

}

**return** EXIT\_SUCCESS;

}

**#define MAXVAL 100**

**int** sp = **0**; /\* Next free stack position. \*/

**double** val[MAXVAL]; /\* value stack. \*/

/\* push: push f onto stack. \*/

**void** push(**double** f)

{

**if**(sp < MAXVAL)

val[sp++] = f;

**else**

printf("\nError: stack full can't push %g\n", f);

}

/\*pop: pop and return top value from stack.\*/

**double** pop(**void**)

{

**if**(sp > **0**)

**return** val[--sp];

**else**

{

printf("\nError: stack empty\n");

**return** **0.0**;

}

}

**void** showTop(**void**)

{

**if**(sp > **0**)

printf("Top of stack contains: %8g\n", val[sp-**1**]);

**else**

printf("The stack is empty!\n");

}

**void** duplicate(**void**)

{

**double** temp = pop();

push(temp);

push(temp);

}

**void** swapItems(**void**)

{

**double** item1 = pop();

**double** item2 = pop();

push(item1);

push(item2);

}

/\* pop only returns a value if sp is greater than zero. So setting the

stack pointer to zero will cause pop to return its error \*/

**void** clearStack(**void**)

{

sp = **0**;

}

**int** getch(**void**);

**void** unGetch(**int**);

/\* Getop: get next operator or numeric operand. \*/

**int** Getop(**char** s[])

{

**int** i = **0**;

**int** c;

**int** next;

/\* Skip whitespace \*/

**while**((s[**0**] = c = getch()) == **' '** || c == **'\t'**)

;

s[**1**] = **'\0'**;

/\* Not a number but may contain a unary minus. \*/

**if**(!isdigit(c) && c != **'.'** && c != **'-'**)

**return** c;

**if**(c == **'-'**)

{

next = getch();

**if**(!isdigit(next) && next != **'.'**)

{

**return** c;

}

c = next;

}

**else**

c = getch();

**while**(isdigit(s[++i] = c))

c = getch();

**if**(c == **'.'**) /\* Collect fraction part. \*/

**while**(isdigit(s[++i] = c = getch()))

;

s[i] = **'\0'**;

**if**(c != EOF)

unGetch(c);

**return** NUMBER;

}

**#define BUFSIZE 100**

**char** buf[BUFSIZE];

**int** bufp = **0**;

/\* Getch: get a ( possibly pushed back) character. \*/

**int** getch(**void**)

{

**return** (bufp > **0**) ? buf[--bufp]: getchar();

}

/\* unGetch: push character back on input. \*/

**void** unGetch(**int** c)

{

**if**(bufp >= BUFSIZE)

printf("\nUnGetch: too many characters\n");

**else**

buf[bufp++] = c;

}

**Answer to Exercise 4-5, page 79**

*Add access to library functions like sin , exp , and pow . See <math.h> in Appendix B, Section 4.*

**#include<stdlib.h>**

**#include<stdio.h>**

**#include<ctype.h>**

**#include<math.h>**

**#include <string.h>**

**#define MAXOP 100**

**#define NUMBER 0**

**#define IDENTIFIER 1**

**#define TRUE 1**

**#define FALSE 0**

/\*

The new additions deal with adding functions from math.h to the

calculator.

In anticipation of the following exercise the code deals with an

identifier in the following manner:

If the identifier is recognised as one of the supported mathematical

functions then that function from the library is called. If the

identifier is not one of the supported functions, even if it is a

valid function from math.h it is ignored.

The main changes are the introduction of another define value

(IDENTIFIER) along with its associated case in the switch statement.

Getop has also been changed to deal with reading in alphabetical

characters.

This is exercise 4-5 from Kernighan & Ritchie, page 79.

\*/

**int** Getop(**char** s[]);

**void** push(**double** val);

**double** pop(**void**);

**void** showTop(**void**);

**void** duplicate(**void**);

**void** swapItems(**void**);

**void** clearStack();

**void** dealWithName(**char** s[]);

**int** main(**void**)

{

**int** type;

**double** op2;

**char** s[MAXOP];

**int** flag = TRUE;

**while**((type = Getop(s)) != EOF)

{

**switch**(type)

{

**case** NUMBER:

push(atof(s));

**break**;

**case** IDENTIFIER:

dealWithName(s);

**break**;

**case** **'+'**:

push(pop() + pop());

**break**;

**case** **'\*'**:

push(pop() \* pop());

**break**;

**case** **'-'**:

op2 = pop();

push(pop()- op2);

**break**;

**case** **'/'**:

op2 = pop();

**if**(op2)

push(pop() / op2);

**else**

printf("\nError: division by zero!");

**break**;

**case** **'%'**:

op2 = pop();

**if**(op2)

push(fmod(pop(), op2));

**else**

printf("\nError: division by zero!");

**break**;

**case** **'?'**:

showTop();

**break**;

**case** **'#'**:

duplicate();

**break**;

**case** **'~'**:

swapItems();

**break**;

**case** **'!'**:

clearStack();

**case** **'\n'**:

printf("\n\t%.8g\n", pop());

**break**;

**default**:

printf("\nError: unknown command %s.\n", s);

**break**;

}

}

**return** EXIT\_SUCCESS;

}

**#define MAXVAL 100**

**int** sp = **0**; /\* Next free stack position. \*/

**double** val[MAXVAL]; /\* value stack. \*/

/\* push: push f onto stack. \*/

**void** push(**double** f)

{

**if**(sp < MAXVAL)

val[sp++] = f;

**else**

printf("\nError: stack full can't push %g\n", f);

}

/\*pop: pop and return top value from stack.\*/

**double** pop(**void**)

{

**if**(sp > **0**)

**return** val[--sp];

**else**

{

printf("\nError: stack empty\n");

**return** **0.0**;

}

}

**void** showTop(**void**)

{

**if**(sp > **0**)

printf("Top of stack contains: %8g\n", val[sp-**1**]);

**else**

printf("The stack is empty!\n");

}

/\*

Alternatively:

void showTop(void)

{

double item = pop();

printf("Top of stack contains: %8g\n", item);

push(item);

}

\*/

**void** duplicate(**void**)

{

**double** temp = pop();

push(temp);

push(temp);

}

**void** swapItems(**void**)

{

**double** item1 = pop();

**double** item2 = pop();

push(item1);

push(item2);

}

**void** clearStack(**void**)

{

sp = **0**;

}

/\* deal with a string/name this may be either a maths function or for

future exercises: a variable \*/

**void** dealWithName(**char** s[])

{

**double** op2;

**if**( **0** == strcmp(s, "sin"))

push(sin(pop()));

**else** **if**( **0** == strcmp(s, "cos"))

push(cos(pop()));

**else** **if** (**0** == strcmp(s, "exp"))

push(exp(pop()));

**else** **if**(!strcmp(s, "pow"))

{

op2 = pop();

push(pow(pop(), op2));

}

**else**

printf("%s is not a supported function.\n", s);

}

**int** getch(**void**);

**void** unGetch(**int**);

/\* Getop: get next operator or numeric operand. \*/

**int** Getop(**char** s[])

{

**int** i = **0**;

**int** c;

**int** next;

/\*size\_t len;\*/

/\* Skip whitespace \*/

**while**((s[**0**] = c = getch()) == **' '** || c == **'\t'**)

;

s[**1**] = **'\0'**;

**if**(isalpha(c))

{

i = **0**;

**while**(isalpha(s[i++] = c ))

c = getch();

s[i - **1**] = **'\0'**;

**if**(c != EOF)

unGetch(c);

**return** IDENTIFIER;

}

/\* Not a number but may contain a unary minus. \*/

**if**(!isdigit(c) && c != **'.'** && c != **'-'**)

**return** c;

**if**(c == **'-'**)

{

next = getch();

**if**(!isdigit(next) && next != **'.'**)

{

**return** c;

}

c = next;

}

**else**

c = getch();

**while**(isdigit(s[++i] = c))

c = getch();

**if**(c == **'.'**) /\* Collect fraction part. \*/

**while**(isdigit(s[++i] = c = getch()))

;

s[i] = **'\0'**;

**if**(c != EOF)

unGetch(c);

**return** NUMBER;

}

**#define BUFSIZE 100**

**char** buf[BUFSIZE];

**int** bufp = **0**;

/\* Getch: get a ( possibly pushed back) character. \*/

**int** getch(**void**)

{

**return** (bufp > **0**) ? buf[--bufp]: getchar();

}

/\* unGetch: push character back on input. \*/

**void** unGetch(**int** c)

{

**if**(bufp >= BUFSIZE)

printf("\nUnGetch: too many characters\n");

**else**

buf[bufp++] = c;

}

**Answer to Exercise 4-6, page 79**

***Add commands for handling variables. (It's easy to provide twenty-six variables with single-letter names.) Add a variable for the most recently printed value.***

**#include <stdlib.h>**

**#include <stdio.h>**

**#include <ctype.h>**

**#include <math.h>**

**#include <string.h>**

**#define MAXOP 100**

**#define NUMBER 0**

/\* 4-6 these are new for this exercise\*/

**#define IDENTIFIER 1**

**#define ENDSTRING 2**

/\* 4-6 end of new stuff \*/

**#define TRUE 1**

**#define FALSE 0**

**#define MAX\_ID\_LEN 32**

**#define MAXVARS 30**

/\*

The new additions deal with adding variables to the calculator.

If the identifier is recognised as one of the supported mathematical

functions then that function from the library is called. If the

identifier is not one of the supported functions, even if it is a

valid function from math.h it is ignored.

This is a class 1 solution as it uses structures which are not

introduced until Chapter 6. This allows the use of "normal" names for

variables rather than the suggested single letter though any

identifier is limited to 31 characters.

The main changes are:

1. The introduction of two more define values (IDENTIFIER,

ENDSTRING) along with associated cases in the switch statement.

2. Getop has also been changed to deal with reading in alphabetical

characters and coping with the '=' sign.

3. A structure to hold the variable name and value.

4. Another case in the switch statement to deal with the '=' sign.

5. Altering the clearStack function to clear the array of structs as

well as the stack.

6. The '<' operator now prints the last accessed variable.

Improvements:

The code could be made class 0 by the use of "parallel" arrays for the

names and values rather than a struct but this would be messy and is

the situation that structs were made for.

The use of a binary tree together with dynamically allocated memory

would allow the arbitrary limit of 30 variables to be avoided. This

would still be a class 1 solution.

This is exercise 4-6 from Kernighan & Ritchie, page 79.

\*/

/\* 4-6 this is new for this program \*/

**struct** varType{

**char** name[MAX\_ID\_LEN];

**double** val;

};

/\* 4-6 End of new stuff \*/

**int** Getop(**char** s[]);

**void** push(**double** val);

**double** pop(**void**);

**void** showTop(**void**);

**void** duplicate(**void**);

**void** swapItems(**void**);

/\* 4-6 this is new for this program \*/

/\* Changed clearStack(void) to clearStacks(struct varType var[])\*/

**void** clearStacks(**struct** varType var[]);

**void** dealWithName(**char** s[], **struct** varType var[]);

**void** dealWithVar(**char** s[], **struct** varType var[]);

**int** pos = **0**;

**struct** varType last;

/\* 4-6 End of new stuff \*/

**int** main(**void**)

{

**int** type;

**double** op2;

**char** s[MAXOP];

**struct** varType var[MAXVARS];

/\* Use the new function here \*/

clearStacks(var);

**while**((type = Getop(s)) != EOF)

{

**switch**(type)

{

**case** NUMBER:

push(atof(s));

**break**;

**case** IDENTIFIER:

dealWithName(s, var);

**break**;

**case** **'+'**:

push(pop() + pop());

**break**;

**case** **'\*'**:

push(pop() \* pop());

**break**;

**case** **'-'**:

op2 = pop();

push(pop()- op2);

**break**;

**case** **'/'**:

op2 = pop();

**if**(op2)

push(pop() / op2);

**else**

printf("\nError: division by zero!");

**break**;

**case** **'%'**:

op2 = pop();

**if**(op2)

push(fmod(pop(), op2));

**else**

printf("\nError: division by zero!");

**break**;

**case** **'?'**:

showTop();

**break**;

**case** **'#'**:

duplicate();

**break**;

**case** **'~'**:

swapItems();

**break**;

**case** **'!'**:

clearStacks(var);

**break**;

**case** **'\n'**:

printf("\n\t%.8g\n", pop());

**break**;

/\* 4-6 this is new for this program \*/

**case** ENDSTRING:

**break**;

**case** **'='**:

pop();

var[pos].val = pop();

last.val = var[pos].val;

push(last.val);

**break**;

**case** **'<'**:

printf("The last variable used was: %s (value == %g)\n",

last.name, last.val);

**break**;

/\* 4-6 End of new stuff \*/

**default**:

printf("\nError: unknown command %s.\n", s);

**break**;

}

}

**return** EXIT\_SUCCESS;

}

**#define MAXVAL 100**

**int** sp = **0**; /\* Next free stack position. \*/

**double** val[MAXVAL]; /\* value stack. \*/

/\* push: push f onto stack. \*/

**void** push(**double** f)

{

**if**(sp < MAXVAL)

val[sp++] = f;

**else**

printf("\nError: stack full can't push %g\n", f);

}

/\*pop: pop and return top value from stack.\*/

**double** pop(**void**)

{

**if**(sp > **0**)

{

**return** val[--sp];

}

**else**

{

printf("\nError: stack empty\n");

**return** **0.0**;

}

}

**void** showTop(**void**)

{

**if**(sp > **0**)

printf("Top of stack contains: %8g\n", val[sp-**1**]);

**else**

printf("The stack is empty!\n");

}

/\*

Alternatively:

void showTop(void)

{

double item = pop();

printf("Top of stack contains: %8g\n", item);

push(item);

}

\*/

**void** duplicate(**void**)

{

**double** temp = pop();

push(temp);

push(temp);

}

**void** swapItems(**void**)

{

**double** item1 = pop();

**double** item2 = pop();

push(item1);

push(item2);

}

/\* 4-6 this is new for this program \*/

/\* Altered to clear both the main stack and that of the variable

structure \*/

**void** clearStacks(**struct** varType var[])

{

**int** i;

/\* Clear the main stack by setting the pointer to the bottom. \*/

sp = **0**;

/\* Clear the variables by setting the initial element of each name

to the terminating character. \*/

**for**( i = **0**; i < MAXVARS; ++i)

{

var[i].name[**0**] = **'\0'**;

var[i].val = **0.0**;

}

}

/\* a string/name may be either a maths function or a variable \*/

**void** dealWithName(**char** s[], **struct** varType var[])

{

**double** op2;

**if**(!strcmp(s, "sin"))

push(sin(pop()));

**else** **if**(!strcmp(s, "cos"))

push(cos(pop()));

**else** **if** (!strcmp(s, "exp"))

push(exp(pop()));

**else** **if**(!strcmp(s, "pow"))

{

op2 = pop();

push(pow(pop(), op2));

}

/\* Finally if it isn't one of the supported maths functions we have a

variable to deal with. \*/

**else**

{

dealWithVar(s, var);

}

}

/\* Our identifier is not one of the supported maths function so we have

to regard it as an identifier. \*/

**void** dealWithVar(**char** s[], **struct** varType var[])

{

**int** i = **0**;

**while**(var[i].name[**0**] != **'\0'** && i < MAXVARS-**1**)

{

**if**(!strcmp(s, var[i].name))

{

strcpy(last.name, s);

last.val = var[i].val;

push(var[i].val);

pos = i;

**return**;

}

i++;

}

/\* variable name not found so add it \*/

strcpy(var[i].name, s);

/\* And save it to the last variable \*/

strcpy(last.name, s);

push(var[i].val);

pos = i;

}

/\* 4-6 End of new stuff \*/

**int** getch(**void**);

**void** unGetch(**int**);

/\* Getop: get next operator or numeric operand. \*/

**int** Getop(**char** s[])

{

**int** i = **0**;

**int** c;

**int** next;

/\* Skip whitespace \*/

**while**((s[**0**] = c = getch()) == **' '** || c == **'\t'**)

{

;

}

s[**1**] = **'\0'**;

**if**(isalpha(c))

{

i = **0**;

**while**(isalpha(s[i++] = c ))

{

c = getch();

}

s[i - **1**] = **'\0'**;

**if**(c != EOF)

unGetch(c);

**return** IDENTIFIER;

}

/\* Not a number but may contain a unary minus. \*/

**if**(!isdigit(c) && c != **'.'** && c != **'-'**)

{

/\* 4-6 Deal with assigning a variable. \*/

**if**(**'='** == c && **'\n'** == (next = getch()))

{

unGetch(**'\0'**);

**return** c;

}

**if**(**'\0'** == c)

**return** ENDSTRING;

**return** c;

}

**if**(c == **'-'**)

{

next = getch();

**if**(!isdigit(next) && next != **'.'**)

{

**return** c;

}

c = next;

}

**else**

{

c = getch();

}

**while**(isdigit(s[++i] = c))

{

c = getch();

}

**if**(c == **'.'**) /\* Collect fraction part. \*/

{

**while**(isdigit(s[++i] = c = getch()))

;

}

s[i] = **'\0'**;

**if**(c != EOF)

unGetch(c);

**return** NUMBER;

}

**#define BUFSIZE 100**

**int** buf[BUFSIZE];

**int** bufp = **0**;

/\* Getch: get a ( possibly pushed back) character. \*/

**int** getch(**void**)

{

**return** (bufp > **0**) ? buf[--bufp]: getchar();

}

/\* unGetch: push character back on input. \*/

**void** unGetch(**int** c)

{

**if**(bufp >= BUFSIZE)

printf("\nUnGetch: too many characters\n");

**else**

buf[bufp++] = c;

}

**Answer to Exercise 4-7, page 79**

*Write a routine ungets(s) that will push back an entire string onto the input. Should ungets know about buf and bufp , or should it just use ungetch ?*

/\* K&R Exercise 4-7 \*/

/\* Steven Huang \*/

**#include <string.h>**

**#include <stdio.h>**

**#define BUFSIZE 100**

**char** buf[BUFSIZE]; /\* buffer for ungetch \*/

**int** bufp = **0**; /\* next free position in buf \*/

**int** getch(**void**) /\* get a (possibly pushed back) character \*/

{

**return** (bufp > **0**) ? buf[--bufp] : getchar();

}

**void** ungetch(**int** c) /\* push character back on input \*/

{

**if**(bufp >= BUFSIZE)

printf("ungetch: too many characters\n");

**else**

buf[bufp++] = c;

}

/\*

ungets() actually takes a little bit of thought. Should the

first character in "s" be sent to ungetch() first, or should

it be sent last? I assumed that most code calling getch()

would be of this form:

char array[...];

int i;

while (...) {

array[i++] = getch();

}

In such cases, the same code might call ungets() as:

ungets(array);

and expect to repeat the while loop to get the same string

back. This requires that the last character be sent first

to ungetch() first, because getch() and ungetch() work with

a stack.

To answer K&R2's additional question for this problem,

it's usually preferable for something like ungets() to just

build itself on top of ungetch(). This allows us to change

ungetch() and getch() in the future, perhaps to use a linked

list instead, without affecting ungets().

\*/

**void** ungets(**const** **char** \*s)

{

size\_t i = strlen(s);

**while** (i > **0**)

ungetch(s[--i]);

}

**int** main(**void**)

{

**char** \*s = "hello, world. this is a test.";

**int** c;

ungets(s);

**while** ((c = getch()) != EOF)

putchar(c);

**return** **0**;

}

**Answer to Exercise 4-8, page 79**

*Suppose there will never be more than one character of pushback. Modify getch and ungetch accordingly.*

/\* K&R Exercise 4-8 \*/

/\* Steven Huang \*/

**#include <stdio.h>**

**int** buf = EOF; /\* buffer for ungetch \*/

**int** getch(**void**) /\* get a (possibly pushed back) character \*/

{

**int** temp;

**if** (buf != EOF) {

temp = buf;

buf = EOF;

} **else** {

temp = getchar();

}

**return** temp;

}

**void** ungetch(**int** c) /\* push character back on input \*/

{

**if**(buf != EOF)

printf("ungetch: too many characters\n");

**else**

buf = c;

}

**int** main(**void**)

{

**int** c;

**while** ((c = getch()) != EOF) {

**if** (c == **'/'**) {

putchar(c);

**if** ((c = getch()) == **'\*'**) {

ungetch(**'!'**);

}

}

putchar(c);

}

**return** **0**;

}

**Answer to Exercise 4-12, page 88**

*Adapt the ideas of printd to write a recursive version of atoi ; that is, convert an integer into a string by calling a recursive routine.*

/\*

itoa() is non-standard, but defined on p.64 as having this prototype:

void itoa(int n, char s[])

Instead of this, I thought I'd use a different prototype (one I got from

the library manual of one of my compilers) since it includes all of the

above:

char \*itoa(int value, char \*digits, int base);

Description: The itoa() function converts an integer value into an

ASCII string of digits. The base argument specifies the number base for

the conversion. The base must be a value in the range [2..36], where 2

is binary, 8 is octal, 10 is decimal, and 16 is hexadecimal. The buffer

pointed to by digits must be large enough to hold the ASCII string of

digits plus a terminating null character. The maximum amount of buffer

space used is the precision of an int in bits + 2 (one for the sign and

one for the terminating null).

Returns: digits, or NULL if error.

\*/

**#include <stdlib.h>**

**char** \*utoa(**unsigned** value, **char** \*digits, **int** base)

{

**char** \*s, \*p;

s = "0123456789abcdefghijklmnopqrstuvwxyz"; /\* don't care if s is in

\* read-only memory

\*/

**if** (base == **0**)

base = **10**;

**if** (digits == NULL || base < **2** || base > **36**)

**return** NULL;

**if** (value < (**unsigned**) base) {

digits[**0**] = s[value];

digits[**1**] = **'\0'**;

} **else** {

**for** (p = utoa(value / ((**unsigned**)base), digits, base);

\*p;

p++);

utoa( value % ((**unsigned**)base), p, base);

}

**return** digits;

}

**char** \*itoa(**int** value, **char** \*digits, **int** base)

{

**char** \*d;

**unsigned** u; /\* assume unsigned is big enough to hold all the

\* unsigned values -x could possibly be -- don't

\* know how well this assumption holds on the

\* DeathStation 9000, so beware of nasal demons

\*/

d = digits;

**if** (base == **0**)

base = **10**;

**if** (digits == NULL || base < **2** || base > **36**)

**return** NULL;

**if** (value < **0**) {

\*d++ = **'-'**;

u = -value;

} **else**

u = value;

utoa(u, d, base);

**return** digits;

}

**Answer to Exercise 4-13, page 88**

*Write a recursive version of the function reverse(s) , which reverses the string s in place.*

/\*

EXERCISE 4-13 Gregory Pietsch

\*/

**static** **void** swap(**char** \*a, **char** \*b, size\_t n)

{

**while** (n--) {

\*a ^= \*b;

\*b ^= \*a;

\*a ^= \*b;

a++;

b++;

}

}

**void** my\_memrev(**char** \*s, size\_t n)

{

**switch** (n) {

**case** **0**:

**case** **1**:

**break**;

**case** **2**:

**case** **3**:

swap(s, s + n - **1**, **1**);

**break**;

**default**:

my\_memrev(s, n / **2**);

my\_memrev(s + ((n + **1**) / **2**), n / **2**);

swap(s, s + ((n + **1**) / **2**), n / **2**);

**break**;

}

}

**void** reverse(**char** \*s)

{

**char** \*p;

**for** (p = s; \*p; p++)

;

my\_memrev(s, (size\_t)(p - s));

}

**Answer to Exercise 4-14, page 91**

*Define a macro swap(t,x,y) that interchanges two arguments of type t . (Block structure will help.)*   
  
Here are Greg's solutions for Cat 0 and Cat 1:

/\* EXERCISE 4-14 Gregory Pietsch \*/

/\* conditional compilation added by RJH \*/

**#ifdef CATEGORY\_0**

**#define swap(t,x,y) do{t z=x;x=y;y=z}while(0)**

**#else**

**#ifdef CATEGORY\_1**

/\*

This works if I can use the assignment operator on type t.

I didn't know if I was allowed to use sizeof or not and still remain

Level 0, otherwise this one is better:

\*/

**#define swap(t,x,y) \**

**do { \**

**(unsigned char \*)a=(unsigned char \*)(&(x)); \**

**(unsigned char \*)b=(unsigned char \*)(&(y)); \**

**size\_t i = sizeof(t); \**

**while (i--) { \**

**\*a ^= \*b; \**

**\*b ^= \*a; \**

**\*a ^= \*b; \**

**a++; \**

**b++; \**

**} \**

**} while (0)**

**#endif**

**#endif**

/\* editor's note: sizeof is first mentioned on p91, after this exercise,

\* and is not explained properly until p135, so it can be used in

\* Category 0 solutions only for exercises 6-1 onward.

\*/

...and here is a lively entry for Category 0, from Lars, which uses token pasting to derive a name for the temporary variable:

/\*

\* Solution to exercise 4-14 in K&R2, page 91:

\*

\* Define a macro swap(t,x,y) that interchanges two arguments of type t.

\* (Block structure will help.)

\*

\* Feel free to modify and copy, if you really must, but preferably not.

\* This is just an exercise in preprocessor mechanics, not an example of

\* how it should really be used. The trickery is not worth it to save three

\* lines of code.

\*

\* To exchange the values of two variables we need a temporary variable and

\* this one needs a name. Any name we pick, the user of the macro might also

\* use. Thus, we use the preprocessor argument concatenation operator ## to

\* create the name from the actual variable names in the call. This guarantees

\* that the result won't be either of the actual arguments. In order to

\* make sure the result also does not fall into the implementation's name

\* space, we prefix the name with something safe.

\*

\* Lars Wirzenius <liw@iki.fi>

\*/

**#include <stdio.h>**

**#define swap(t, x, y) \**

**do { \**

**t safe ## x ## y; \**

**safe ## x ## y = x; \**

**x = y; \**

**y = safe ## x ## y; \**

**} while (0)**

**int** main(**void**) {

**int** ix, iy;

**double** dx, dy;

**char** \*px, \*py;

ix = **42**;

iy = **69**;

printf("integers before swap: %d and %d\n", ix, iy);

swap(**int**, ix, iy);

printf("integers after swap: %d and %d\n", ix, iy);

dx = **123.0**;

dy = **321.0**;

printf("doubles before swap: %g and %g\n", dx, dy);

swap(**double**, dx, dy);

printf("integers after swap: %g and %g\n", dx, dy);

px = "hello";

py = "world";

printf("pointers before swap: %s and %s\n", px, py);

swap(**char** \*, px, py);

printf("integers after swap: %s and %s\n", px, py);

**return** **0**;

}

...and here is yet another solution from Gregory:

**#define swap(t,x,y) \**

**do { \**

**(unsigned char \*)\_0=(unsigned char \*)(&(x)); \**

**(unsigned char \*)\_1=(unsigned char \*)(&(y)); \**

**unsigned long \_2 = (unsigned long) \**

**((unsigned char \*)(&(x)+1) \**

**- (unsigned char \*)(&(x))); \**

**while (\_2--) { \**

**\*\_0 ^= \*\_1; \**

**\*\_1 ^= \*\_0; \**

**\*\_0 ^= \*\_1; \**

**\_0++; \**

**\_1++; \**

**} \**

**} while (0)**

**Answer to Exercise 5-1, page 97**

*As written, getint treats a + or - not followed by a digit as a valid representation of zero. Fix it to push such a character back on the input.*   
  
Here is Greg's solution:

**#include <ctype.h>**

**int** getch(**void**);

**void** ungetch(**int**);

/\* getint: get next integer from input into \*pn \*/

**int** getint(**int** \*pn)

{

**int** c, sign, sawsign;

**while** (isspace(c = getch())) /\* skip white space \*/

;

**if** (!isdigit(c) && c != EOF && c != **'+'** && c != **'-'**) {

ungetch(c); /\* it's not a number \*/

**return** **0**;

}

sign = (c == **'-'**) ? -**1** : **1**;

**if** (sawsign = (c == **'+'** || c == **'-'**))

c = getch();

**if** (!isdigit(c)) {

ungetch(c);

**if** (sawsign)

ungetch((sign == -**1**) ? **'-'** : **'+'**);

**return** **0**;

}

**for** (\*pn = **0**; isdigit(c); c = getch())

\*pn = **10** \* \*pn + (c - **'0'**);

\*pn \*= sign;

**if** (c != EOF)

ungetch(c);

**return** c;

}

**Answer to Exercise 5-2, page 97**

*Write getfloat , the floating-point analog of getint . What type does getfloat return as its function value?*   
  
Here is Chris's solution:

/\*

\* Exercise 5-2 from The C Programming Language, 2nd edition, by Kernighan

\* and Ritchie.

\*

\* "Write getfloat, the floating-point analog of getint. What type does

\* getfloat return as its function value?"

\*/

/\*

\* Here's the getint function, from section 5.2:

\*/

**#include <ctype.h>**

**#include <stdio.h>**

**int** getch(**void**);

**void** ungetch(**int**);

/\* getint: get next integer from input into \*pn \*/

**int** getint(**int** \*pn)

{

**int** c, sign;

**while** (isspace(c = getch())) /\* skip white space \*/

;

**if** (!isdigit(c) && c != EOF && c != **'+'** && c != **'-'**) {

ungetch(c); /\* it is not a number \*/

**return** **0**;

}

sign = (c == **'-'**) ? -**1** : **1**;

**if** (c == **'+'** || c == **'-'**)

c = getch();

**for** (\*pn = **0**; isdigit(c); c = getch())

\*pn = **10** \* \*pn + (c - **'0'**);

\*pn \*= sign;

**if** (c != EOF)

ungetch(c);

**return** c;

}

/\*

\* The getch and ungetch functions, from section 4.3, are also required.

\*/

**#include <stdio.h>**

**#define BUFSIZE 100**

**char** buf[BUFSIZE]; /\* buffer for ungetch \*/

**int** bufp = **0**; /\* next free position in buf \*/

**int** getch(**void**) /\* get a (possibly pushed-back) character \*/

{

**return** (bufp > **0**) ? buf[--bufp] : getchar();

}

**void** ungetch(**int** c) /\* push character back on input \*/

{

**if** (bufp >= BUFSIZE)

printf("ungetch: too many characters\n");

**else**

buf[bufp++] = c;

}

/\*

\* The getfloat function.

\*

\* Reads the next number from input, and puts it into \*fp. Returns EOF for

\* end of file, zero if the next input is not a number, and a positive

\* value of the input contains a valid number.

\*

\* Based heavily on the getint function from K&R2.

\*/

**#include <ctype.h>**

**#include <math.h>**

**int** getfloat(**float** \*fp)

{

**int** ch;

**int** sign;

**int** fraction;

**int** digits;

**while** (isspace(ch = getch())) /\* skip white space \*/

;

**if** (!isdigit(ch) && ch != EOF && ch != **'+'**

&& ch != **'-'** && ch != **'.'**) {

ungetch(ch);

**return** **0**;

}

sign = (ch == **'-'**) ? -**1** : **1**;

**if** (ch == **'+'** || ch == **'-'**) {

ch = getch();

**if** (!isdigit(ch) && ch != **'.'**) {

**if** (ch == EOF) {

**return** EOF;

} **else** {

ungetch(ch);

**return** **0**;

}

}

}

\*fp = **0**;

fraction = **0**;

digits = **0**;

**for** ( ; isdigit(ch) || ch == **'.'** ; ch = getch()) {

**if** (ch == **'.'**) {

fraction = **1**;

} **else** {

**if** (!fraction) {

\*fp = **10** \* \*fp + (ch - **'0'**);

} **else** {

\*fp = \*fp + ((ch - **'0'**) / pow(**10**, fraction));

fraction++;

}

digits++;

}

}

\*fp \*= sign;

**if** (ch == EOF) {

**return** EOF;

} **else** {

ungetch(ch);

**return** (digits) ? ch : **0**;

}

}

/\*

\* Test module.

\*/

**#include <stdio.h>**

**int** main(**void**)

{

**int** ret;

**do** {

**float** f;

fputs("Enter a number: ", stdout);

fflush(stdout);

ret = getfloat(&f);

**if** (ret > **0**) {

printf("You entered: %f\n", f);

}

} **while** (ret > **0**);

**if** (ret == EOF) {

puts("Stopped by EOF.");

} **else** {

puts("Stopped by bad input.");

}

**return** **0**;

}

...and here is Greg's solution:

/\* Gregory Pietsch <gkp1@flash.net> Exercise 5-2 dated 2001-01-08 \*/

**#include <ctype.h>**

**#include <limits.h>**

/\* also uses getch and ungetch from Section 4.3 \*/

/\* number of significant digits in a double \*/

**#define SIG\_MAX 32**

/\* store double in d; return next character \*/

**int** getfloat(**double** \*d)

{

**const** **char** point = **'.'**; /\* localeconv->decimal\_point[0]; \*/

**int** c;

**char** buf[SIG\_MAX], sign, sawsign, sawe, sawesign, esign;

**double** x;

**static** **double** fac[] = {**0.0**, **1.0e8**, **1.0e16**, **1.0e24**, **1.0e32**};

**double** dpow;

**int** ndigit, nsig, nzero, olead, opoint, n;

**char** \*pc;

**long** lo[SIG\_MAX / **8** + **1**], lexp;

**long** \*pl;

/\* skip white space \*/

**while** (isspace(c = getch()))

;

**if** (sawsign = (c == **'-'** || c == **'+'**)) {

sign = c;

c = getch();

} **else**

sign = **'+'**;

olead = -**1**;

opoint = -**1**;

ndigit = **0**;

nsig = **0**;

nzero = **0**;

**while** (c != EOF) {

**if** (c == point) {

**if** (**0** <= opoint)

**break**; /\* already seen point \*/

**else**

opoint = ndigit;

} **else** **if** (c == **'0'**) {

/\* saw a zero \*/

nzero++;

ndigit++;

} **else** **if** (!isdigit(c))

**break**; /\* found nondigit \*/

**else** {

/\* got a nonzero digit \*/

**if** (olead < **0**)

olead = nzero;

**else** {

/\* deliver zeros \*/

**for** ( ; **0** < nzero && nsig < SIG\_MAX; --nzero)

buf[nsig++] = **0**;

}

++ndigit;

/\* deliver digit \*/

**if** (nsig < SIG\_MAX)

buf[nsig++] = (c - **'0'**);

}

c = getch();

}

**if** (ndigit == **0**) {

/\* no digits? \*/

\*d = **0.0**;

**if** (c != EOF)

ungetch(c);

**if** (**0** <= opoint) {

/\* saw point \*/

ungetch(c = point);

}

**if** (sawsign) {

/\* saw sign \*/

ungetch(c = sign);

}

**return** c;

}

/\* skip trailing digits \*/

**for** ( ; **0** < nsig && buf[nsig - **1**] == **0**; --nsig)

;

/\* compute significand \*/

pc = buf;

pl = &(lo[nsig >> **3**]);

**for** (\*pl = **0**, n = nsig; **0** < n; --n) {

**if** ((n & **7**) == **0**)

/\* start new sum \*/

\*--pl = \*pc++;

**else**

\*pl = \*pl \* **10** + \*pc++;

}

**for** (\*d = (**double**)(lo[**0**]), n = **0**; ++n <= (nsig >> **3**); )

**if** (lo[n] != **0**)

\*d += fac[n] \* (**double**)(lo[n]);

/\* fold in any explicit exponent \*/

lexp = **0**;

**if** (c == **'e'** || c == **'E'**) {

/\* we have an explicit exponent \*/

sawe = c;

c = getch();

**if** (sawesign = (c == **'+'** || c == **'-'**)) {

esign = c;

c = getch();

} **else**

esign = **'+'**;

**if** (!isdigit(c)) {

/\* ill-formed exponent \*/

**if** (c != EOF)

ungetch(c);

**if** (sawesign)

ungetch(c = esign);

c = sawe;

} **else** {

/\* get exponent \*/

**while** (isdigit(c)) {

/\* get explicit exponent digits \*/

**if** (lexp < **100000**)

lexp = lexp \* **10** + (c - **'0'**);

/\* else overflow \*/

c = getch();

}

**if** (esign == **'-'**)

lexp = -lexp;

}

}

**if** (c != EOF)

ungetch(c);

**if** (opoint < **0**)

lexp += ndigit - nsig;

**else**

lexp += opoint - olead - nsig;

/\* this is where I pray I don't lose precision \*/

esign = (lexp < **0**) ? **'-'** : **'+'**;

/\* if anyone has a better way of handling overflow, tell me \*/

**if** (lexp < SHRT\_MIN)

lexp = SHRT\_MIN;

**if** (lexp > SHRT\_MAX)

lexp = SHRT\_MAX;

**if** (lexp < **0**)

lexp = -lexp;

**if** (lexp != **0**) {

dpow = (esign == **'-'**) ? **0.1** : **10.0**;

**while** (lexp != **0**) {

/\* form 10.0 to the lexp power \*/

**if** ((lexp & **1**) != **0**) /\* lexp is positive \*/

\*d \*= dpow;

lexp >>= **1**;

dpow \*= dpow;

}

}

/\* if there was a minus sign in front, negate \*d \*/

**if** (sign == **'-'**)

\*d = -(\*d);

**return** c;

}

**Answer to Exercise 5-3, page 107**

*Write a pointer version of the function strcat that we showed in Chapter 2: strcat(s,t) copies the string t to the end of s .*

/\* ex 5-3, p107 \*/

**#include <stdio.h>**

**void** strcpy(**char** \*s, **char** \*t)

{

**while**(\*s++ = \*t++);

}

**void** strcat(**char** \*s, **char** \*t)

{

**while**(\*s)

{

++s;

}

strcpy(s, t);

}

**int** main(**void**)

{

**char** testbuff[**128**];

**char** \*test[] =

{

"",

"1",

"12",

"123",

"1234"

};

size\_t numtests = **sizeof** test / **sizeof** test[**0**];

size\_t thistest;

size\_t inner;

**for**(thistest = **0**; thistest < numtests; thistest++)

{

**for**(inner = **0**; inner < numtests; inner++)

{

strcpy(testbuff, test[thistest]);

strcat(testbuff, test[inner]);

printf("[%s] + [%s] = [%s]\n", test[thistest], test[inner], testbuff);

}

}

**return** **0**;

}

Give nineteen programmers a spec, and you'll get at least twenty completely different programs. As a tiny example of this, here's a totally different solution, by [Bryan Williams.](mailto:gfd34@dial.pipex.com)

/\*

Exercise 5-3. Write a pointer version of the function strcat that we showed in

Chapter 2: strcat(s,t) copies the string t to the end of s.

implementation from chapter 2:

/ \* strcat: concatenate t to end of s; s must be big enough \* /

void strcat(char s[], char t[])

{

int i, j;

i = j = 0;

while (s[i] != '\0') / \* find end of s \* /

i++;

while ((s[i++] = t[j++]) != '\0') / \* copy t \* /

;

}

Author : Bryan Williams

\*/

/\* strcat: concatenate t to end of s; s must be big enough; pointer version \*/

**void** strcat(**char** \*s, **char** \*t)

{

/\* run through the destination string until we point at the terminating '\0' \*/

**while**(**'\0'** != \*s)

{

++s;

}

/\* now copy until we run out of string to copy \*/

**while**(**'\0'** != (\*s = \*t))

{

++s;

++t;

}

}

**#define DRIVER 6**

**#if DRIVER**

**#include <stdio.h>**

**int** main(**void**)

{

**char** S1[**8192**] = "String One";

**char** S2[**8192**] = "String Two";

printf("String one is (%s)\n", S1);

printf("String two is (%s)\n", S2);

strcat(S1, S2);

printf("The combined string is (%s)\n", S1);

**return** **0**;

}

#endif

**Answer to Exercise 5-4, page 107**

*Write the function strend(s,t) , which returns 1 if the string t occurs at the end of the string s , and zero otherwise.*

/\*

Exercise 5-4. Write the function strend(s,t), which returns 1 if the string t

occurs at the end of the string s, and zero otherwise.

Author : Bryan Williams

\*/

**int** strlen(**char** \*s) /\* added by RJH; source: K&R p99 \*/

{

**int** n;

**for**(n = **0**; \*s != **'\0'**; s++)

{

n++;

}

**return** n;

}

**int** strcmp(**char** \*s, **char** \*t) /\* added by RJH; source: K&R p106 \*/

{

**for**(;\*s == \*t; s++, t++)

**if**(\*s == **'\0'**)

**return** **0**;

**return** \*s - \*t;

}

**int** strend(**char** \*s, **char** \*t)

{

**int** Result = **0**;

**int** s\_length = **0**;

**int** t\_length = **0**;

/\* get the lengths of the strings \*/

s\_length = strlen(s);

t\_length = strlen(t);

/\* check if the lengths mean that the string t could fit at the string s \*/

**if**(t\_length <= s\_length)

{

/\* advance the s pointer to where the string t would have to start in string s \*/

s += s\_length - t\_length;

/\* and make the compare using strcmp \*/

**if**(**0** == strcmp(s, t))

{

Result = **1**;

}

}

**return** Result;

}

**#include <stdio.h>**

**int** main(**void**)

{

**char** \*s1 = "some really long string.";

**char** \*s2 = "ng.";

**char** \*s3 = "ng";

**if**(strend(s1, s2))

{

printf("The string (%s) has (%s) at the end.\n", s1, s2);

}

**else**

{

printf("The string (%s) doesn't have (%s) at the end.\n", s1, s2);

}

**if**(strend(s1, s3))

{

printf("The string (%s) has (%s) at the end.\n", s1, s3);

}

**else**

{

printf("The string (%s) doesn't have (%s) at the end.\n", s1, s3);

}

**return** **0**;

}

**Answer to Exercise 5-5, page 107**

*Write versions of the library functions strncpy , strncat , and strncmp , which operate on at most the first n characters of their argument strings. For example, strncpy(s,t,n) copies at most n characters of t to s . Full descriptions are in Appendix B.*   
  
Note: Lars uses EXIT\_FAILURE in his test code, but not in the actual solution code. As far as I can tell, then, this is a Category 0 solution.

/\*

\* Solution to exercise 5-5 in K&R2, page 107:

\*

\* Write versions of the library functions strncpy, strncat,

\* and strncmp, which operate on at most the first n characters

\* of their argument strings. For example, strncpy(s,t,n) copies

\* at most n characters of t to s. Full descriptions are in

\* Appendix B.

\*

\* Note that the description in the exercise is not precise. Here are

\* descriptions from Appendix B (though one should really follow the

\* descriptions in the standard):

\*

\* char \*strncpy(s,ct,n) copy at most n characters of string ct

\* to s, return s. Pad with '\0's is ct

\* has fewer than n characters.

\* char \*strncat(s,ct,n) concatenate at most n characters of

\* string ct to string s, terminate s with

\* '\0'; return s.

\* int strncmp(cs,ct,n) compare at most n characters of string

\* cs to string ct; return <0 if cs<ct,

\* 0 if cs==ct, or >0 if cs>ct.

\*

\* Further note that the standard requires strncmp to compare the

\* characters using unsigned char internally.

\*

\* Implementation note: since the function names are reserved by the

\* standard, I've used the prefix `liw\_'. This also allows me to check

\* the functions against the standard library versions. For each library

\* function, I've written a test function that tests a particular test

\* case. Where appropriate, the test functions use internal buffers that

\* are of size MAX\_BUF; at least some of the test cases should be longer

\* to test all boundary conditions.

\*

\* Feel free to modify, copy, and use as you wish.

\*

\* Lars Wirzenius <liw@iki.fi>

\*/

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <string.h>**

**#define MAX\_BUF 16**

**char** \*liw\_strncpy(**char** \*s, **const** **char** \*ct, size\_t n) {

**char** \*p;

p = s;

**for** (; n > **0** && \*ct != **'\0'**; --n)

\*p++ = \*ct++;

**for** (; n > **0**; --n)

\*p++ = **'\0'**;

**return** s;

}

**char** \*liw\_strncat(**char** \*s, **const** **char** \*ct, size\_t n) {

**char** \*p;

p = s;

**while** (\*p != **'\0'**)

++p;

**for** (; n > **0** && \*ct != **'\0'**; --n)

\*p++ = \*ct++;

\*p = **'\0'**;

**return** s;

}

**int** liw\_strncmp(**const** **char** \*cs, **const** **char** \*ct, size\_t n) {

**while** (n > **0** && \*cs == \*ct && \*cs != **'\0'**) {

++cs;

++ct;

--n;

}

**if** (n == **0** || \*cs == \*ct)

**return** **0**;

**if** (\*(**unsigned** **char** \*) cs < \*(**unsigned** **char** \*) ct)

**return** -**1**;

**return** **1**;

}

**void** test\_ncpy(**const** **char** \*str) {

**char** std\_buf[MAX\_BUF];

**char** liw\_buf[MAX\_BUF];

memset(std\_buf, **0x42**, **sizeof**(std\_buf));

strncpy(std\_buf, str, **sizeof**(std\_buf));

memset(liw\_buf, **0x42**, **sizeof**(liw\_buf));

liw\_strncpy(liw\_buf, str, **sizeof**(liw\_buf));

**if** (memcmp(std\_buf, liw\_buf, **sizeof**(std\_buf)) != **0**) {

fprintf(stderr, "liw\_strncpy failed for <%s>\n", str);

exit(EXIT\_FAILURE);

}

}

**void** test\_ncat(**const** **char** \*first, **const** **char** \*second) {

**char** std\_buf[MAX\_BUF];

**char** liw\_buf[MAX\_BUF];

memset(std\_buf, **0x69**, **sizeof**(std\_buf));

strcpy(std\_buf, first);

strncat(std\_buf, second, **sizeof**(std\_buf) - strlen(std\_buf) - **1**);

memset(liw\_buf, **0x69**, **sizeof**(liw\_buf));

strcpy(liw\_buf, first);

liw\_strncat(liw\_buf, second, **sizeof**(liw\_buf) - strlen(liw\_buf) - **1**);

**if** (memcmp(std\_buf, liw\_buf, **sizeof**(std\_buf)) != **0**) {

fprintf(stderr, "liw\_strncat failed, <%s> and <%s>\n",

first, second);

exit(EXIT\_FAILURE);

}

}

**void** test\_ncmp(**const** **char** \*first, **const** **char** \*second) {

size\_t len;

**int** std\_ret, liw\_ret;

**if** (strlen(first) < strlen(second))

len = strlen(second);

**else**

len = strlen(first);

std\_ret = strncmp(first, second, len);

liw\_ret = liw\_strncmp(first, second, len);

**if** ((std\_ret < **0** && liw\_ret >= **0**) || (std\_ret > **0** && liw\_ret <= **0**) ||

(std\_ret == **0** && liw\_ret != **0**)) {

fprintf(stderr, "liw\_strncmp failed, <%s> and <%s>\n",

first, second);

exit(EXIT\_FAILURE);

}

}

**int** main(**void**) {

test\_ncpy("");

test\_ncpy("a");

test\_ncpy("ab");

test\_ncpy("abcdefghijklmnopqrstuvwxyz"); /\* longer than MAX\_BUF \*/

test\_ncat("", "a");

test\_ncat("a", "bc");

test\_ncat("ab", "cde");

test\_ncat("ab", "cdefghijklmnopqrstuvwxyz"); /\* longer than MAX\_BUF \*/

test\_ncmp("", "");

test\_ncmp("", "a");

test\_ncmp("a", "a");

test\_ncmp("a", "ab");

test\_ncmp("abc", "ab");

printf("All tests pass.\n");

**return** **0**;

}

**Exercise 5-6, page 107**

Greg supplied a fresh version of this answer (which supersedes the old answer, so I've removed it) on 29 Jan 2001.

*Rewrite appropriate programs from earlier chapters and exercises with pointers instead of array indexing. Good possibilities include getline (Chapters 1 and 4), atoi , itoa , and their variants (Chapters 2, 3, and 4), reverse (Chapter 3), and strindex and getop (Chapter 4).*

/\* Gregory Pietsch ex. 5-6 dated 2001-01-29 \*/

**#include <ctype.h>**

**#include <stdio.h>**

**#include <stdlib.h>**

/\* getline: get line into s, return length \*/

**int** getline(**char** \*s, **int** lim)

{

**char** \*p;

**int** c;

p = s;

**while** (--lim > **0** && (c = getchar()) != EOF && c != **'\n'**)

\*p++ = c;

**if** (c == **'\n'**)

\*p++ = c;

\*p = **'\0'**;

**return** (**int**)(p - s);

}

/\* atoi: convert s to an integer

\*

\* Here's the easy way:

\* int atoi(char \*s){return (int)strtoul(s, NULL, 10);}

\* But I'll behave...

\*/

**int** atoi(**char** \*s)

{

**int** n, sign;

**while** (isspace(\*s))

s++;

sign = (\*s == **'+'** || \*s == **'-'**) ? ((\*s++ == **'+'**) ? **1** : -**1**) : **1**;

**for** (n = **0**; isdigit(\*s); s++)

n = (n \* **10**) + (\*s - **'0'**); /\* note to language lawyers --

\* the digits are in consecutive

\* order in the character set

\* C90 5.2.1

\*/

**return** sign \* n;

}

/\* Shamelessly copied from my 4-12 answer

itoa() is non-standard, but defined on p.64 as having this prototype:

void itoa(int n, char s[])

Instead of this, I thought I'd use a different prototype (one I got from

the library manual of one of my compilers) since it includes all of the

above:

char \*itoa(int value, char \*digits, int base);

Description: The itoa() function converts an integer value into an

ASCII string of digits. The base argument specifies the number base for

the conversion. The base must be a value in the range [2..36], where 2

is binary, 8 is octal, 10 is decimal, and 16 is hexadecimal. The buffer

pointed to by digits must be large enough to hold the ASCII string of

digits plus a terminating null character. The maximum amount of buffer

space used is the precision of an int in bits + 2 (one for the sign and

one for the terminating null).

Returns: digits, or NULL if error.

\*/

**char** \*utoa(**unsigned** value, **char** \*digits, **int** base)

{

**char** \*s, \*p;

s = "0123456789abcdefghijklmnopqrstuvwxyz"; /\* don't care if s is in

\* read-only memory

\*/

**if** (base == **0**)

base = **10**;

**if** (digits == NULL || base < **2** || base > **36**)

**return** NULL;

**if** (value < (**unsigned**) base) {

digits[**0**] = s[value];

digits[**1**] = **'\0'**;

} **else** {

**for** (p = utoa(value / ((**unsigned**)base), digits, base);

\*p;

p++);

utoa( value % ((**unsigned**)base), p, base);

}

**return** digits;

}

**char** \*itoa(**int** value, **char** \*digits, **int** base)

{

**char** \*d;

**unsigned** u; /\* assume unsigned is big enough to hold all the

\* unsigned values -x could possibly be -- don't

\* know how well this assumption holds on the

\* DeathStation 9000, so beware of nasal demons

\*/

d = digits;

**if** (base == **0**)

base = **10**;

**if** (digits == NULL || base < **2** || base > **36**)

**return** NULL;

**if** (value < **0**) {

\*d++ = **'-'**;

u = -((**unsigned**)value);

} **else**

u = value;

utoa(u, d, base);

**return** digits;

}

/\* reverse, shamelessly copied from my 4-13 answer \*/

**static** **void** swap(**char** \*a, **char** \*b, size\_t n)

{

**while** (n--) {

\*a ^= \*b;

\*b ^= \*a;

\*a ^= \*b;

a++;

b++;

}

}

**void** my\_memrev(**char** \*s, size\_t n)

{

**switch** (n) {

**case** **0**:

**case** **1**:

**break**;

**case** **2**:

**case** **3**:

swap(s, s + n - **1**, **1**);

**break**;

**default**:

my\_memrev(s, n / **2**);

my\_memrev(s + ((n + **1**) / **2**), n / **2**);

swap(s, s + ((n + **1**) / **2**), n / **2**);

**break**;

}

}

**void** reverse(**char** \*s)

{

**char** \*p;

**for** (p = s; \*p; p++)

;

my\_memrev(s, (size\_t)(p - s));

}

/\* strindex: return index of t in s, -1 if not found \*/

/\* needed strchr(), so here it is: \*/

**static** **char** \*strchr(**char** \*s, **int** c)

{

**char** ch = c;

**for** ( ; \*s != ch; ++s)

**if** (\*s == **'\0'**)

**return** NULL;

**return** s;

}

**int** strindex(**char** \*s, **char** \*t)

{

**char** \*u, \*v, \*w;

**if** (\*t == **'\0'**)

**return** **0**;

**for** (u = s; (u = strchr(u, \*t)) != NULL; ++u) {

**for** (v = u, w = t; ; )

**if** (\*++w == **'\0'**)

**return** (**int**)(u - s);

**else** **if** (\*++v != \*w)

**break**;

}

**return** -**1**;

}

/\* getop \*/

**#define NUMBER '0' /\* from Chapter 4 \*/**

**int** getop(**char** \*s)

{

**int** c;

**while** ((\*s = c = getch()) == **' '** || c == **'\t'**)

;

\*(s + **1**) = **'\0'**;

**if** (!isdigit(c) && c != **'.'**)

**return** c; /\* not a number \*/

**if** (isdigit(c)) /\* collect integer part \*/

**while** (isdigit(\*++s = c = getch()))

;

**if** (c == **'.'**) /\* collect fraction part \*/

**while** (isdigit(\*++s = c = getch()))

;

\*++s = **'\0'**;

**if** (c != EOF)

ungetch(c);

**return** NUMBER;

}

/\* Is there any more? \*/

**Answer to Exercise 5-7, page 110**

*Rewrite readlines to store lines in an array supplied by main , rather than calling alloc to maintain storage. How much faster is the program?*

/\* K&R Exercise 5-7 \*/

/\* Steven Huang \*/

**#include <stdio.h>**

**#include <string.h>**

**#include <stdlib.h>**

**#include <time.h>**

**#define TRUE 1**

**#define FALSE 0**

**#define MAXLINES 5000 /\* maximum number of lines \*/**

**#define MAXLEN 1000 /\* maximum length of a line \*/**

**char** \*lineptr[MAXLINES];

**char** lines[MAXLINES][MAXLEN];

/\* K&R2 p29 \*/

**int** getline(**char** s[], **int** lim)

{

**int** c, i;

**for** (i = **0**; i < lim - **1** && (c = getchar()) != EOF && c != **'\n'**; i++)

s[i] = c;

**if** (c == **'\n'**) {

s[i++] = c;

}

s[i] = **'\0'**;

**return** i;

}

/\* K&R2 p109 \*/

**int** readlines(**char** \*lineptr[], **int** maxlines)

{

**int** len, nlines;

**char** \*p, line[MAXLEN];

nlines = **0**;

**while** ((len = getline(line, MAXLEN)) > **0**)

**if** (nlines >= maxlines || (p = malloc(len)) == NULL)

**return** -**1**;

**else** {

line[len - **1**] = **'\0'**; /\* delete the newline \*/

strcpy(p, line);

lineptr[nlines++] = p;

}

**return** nlines;

}

**int** readlines2(**char** lines[][MAXLEN], **int** maxlines)

{

**int** len, nlines;

nlines = **0**;

**while** ((len = getline(lines[nlines], MAXLEN)) > **0**)

**if** (nlines >= maxlines)

**return** -**1**;

**else**

lines[nlines++][len - **1**] = **'\0'**; /\* delete the newline \*/

**return** nlines;

}

**int** main(**int** argc, **char** \*argv[])

{

/\* read things into cache, to be fair. \*/

readlines2(lines, MAXLINES);

**if** (argc > **1** && \*argv[**1**] == **'2'**) {

puts("readlines2()");

readlines2(lines, MAXLINES);

} **else** {

puts("readlines()");

readlines(lineptr, MAXLINES);

}

**return** **0**;

}

Steven writes: "Unfortunately, the follow-up question here on which version is faster is difficult to determine on my machine, because the difference is very small. I can call malloc() one million times in under a second - this suggests that the conventional wisdom that malloc() is slow and should be avoided may need some more adjustment."   
  
[Editor's note: That's probably because malloc is actually taking memory requests to the system as infrequently as possible, so that most of the calls invoke little more than pointer arithmetic. This suggests that the conventional wisdom may be based on real world programs, rather than artificial "how many mallocs per second can I do" benchmarks. :-) ]   
  
[This space reserved for Steven's right of reply!]

**Exercise 5-8**

*There is no error-checking in day\_of\_year or month\_day. Remedy this defect.*

/\*

\* A solution to exercise 5-8 in K&R2, page 112:

\*

\* There is no error checking in day\_of\_year or month\_day. Remedy

\* this defect.

\*

\* The error to check for is invalid argument values. That is simple, what's

\* hard is deciding what to do in case of error. In the real world, I would

\* use the assert macro from assert.h, but in this solution I take the

\* approach of returning -1 instead. This is more work for the caller, of

\* course.

\*

\* I have selected the year 1752 as the lowest allowed year, because that

\* is when Great Britain switched to the Gregorian calendar, and the leap

\* year validation is valid only for the Gregorian calendar.

\*

\* Lars Wirzenius <liw@iki.fi>

\*/

**#include <stdio.h>**

**static** **char** daytab[**2**][**13**] = {

{**0**, **31**, **28**, **31**, **30**, **31**, **30**, **31**, **31**, **30**, **31**, **30**, **31**},

{**0**, **31**, **29**, **31**, **30**, **31**, **30**, **31**, **31**, **30**, **31**, **30**, **31**},

};

/\* day\_of\_year: set day of year from month & day \*/

**int** day\_of\_year(**int** year, **int** month, **int** day)

{

**int** i, leap;

**if** (year < **1752** || month < **1** || month > **12** || day < **1**)

**return** -**1**;

leap = (year%**4** == **0** && year%**100** != **0**) || year%**400** == **0**;

**if** (day > daytab[leap][month])

**return** -**1**;

**for** (i = **1**; i < month; i++)

day += daytab[leap][i];

**return** day;

}

/\* month\_day: set month, day from day of year \*/

**int** month\_day(**int** year, **int** yearday, **int** \*pmonth, **int** \*pday)

{

**int** i, leap;

**if** (year < **1752** || yearday < **1**)

**return** -**1**;

leap = (year%**4** == **0** && year%**100** != **0**) || year%**400** == **0**;

**if** ((leap && yearday > **366**) || (!leap && yearday > **365**))

**return** -**1**;

**for** (i = **1**; yearday > daytab[leap][i]; i++)

yearday -= daytab[leap][i];

\*pmonth = i;

\*pday = yearday;

**return** **0**;

}

/\* main: test day\_of\_year and month\_day \*/

**int** main(**void**)

{

**int** year, month, day, yearday;

**for** (year = **1970**; year <= **2000**; ++year) {

**for** (yearday = **1**; yearday < **366**; ++yearday) {

**if** (month\_day(year, yearday, &month, &day) == -**1**) {

printf("month\_day failed: %d %d\n",

year, yearday);

} **else** **if** (day\_of\_year(year, month, day) != yearday) {

printf("bad result: %d %d\n", year, yearday);

printf("month = %d, day = %d\n", month, day);

}

}

}

**return** **0**;

}

**Exercise 5-9, page 114**

*Rewrite the routines day\_of\_year and month\_day with pointers instead of indexing.*   
  
  
  
Here's Lars's solution: @br @br

/\*

\* A solution to exercise 5-9 in K&R2, page 114:

\*

\* Rewrite the routines day\_of\_year and month\_day with pointers

\* instead of indexing.

\*

\* Lars Wirzenius <liw@iki.fi>

\*/

**#include <stdio.h>**

**static** **char** daytab[**2**][**13**] = {

{**0**, **31**, **28**, **31**, **30**, **31**, **30**, **31**, **31**, **30**, **31**, **30**, **31**},

{**0**, **31**, **29**, **31**, **30**, **31**, **30**, **31**, **31**, **30**, **31**, **30**, **31**},

};

/\* original versions, for comparison purposes \*/

**int** day\_of\_year(**int** year, **int** month, **int** day)

{

**int** i, leap;

leap = (year%**4** == **0** && year%**100** != **0**) || year%**400** == **0**;

**for** (i = **1**; i < month; i++)

day += daytab[leap][i];

**return** day;

}

**void** month\_day(**int** year, **int** yearday, **int** \*pmonth, **int** \*pday)

{

**int** i, leap;

leap = (year%**4** == **0** && year%**100** != **0**) || year%**400** == **0**;

**for** (i = **1**; yearday > daytab[leap][i]; i++)

yearday -= daytab[leap][i];

\*pmonth = i;

\*pday = yearday;

}

/\* pointer versions \*/

**int** day\_of\_year\_pointer(**int** year, **int** month, **int** day)

{

**int** i, leap;

**char** \*p;

leap = (year%**4** == **0** && year%**100** != **0**) || year%**400** == **0**;

/\* Set `p' to point at first month in the correct row. \*/

p = &daytab[leap][**1**];

/\* Move `p' along the row, to each successive month. \*/

**for** (i = **1**; i < month; i++) {

day += \*p;

++p;

}

**return** day;

}

**void** month\_day\_pointer(**int** year, **int** yearday, **int** \*pmonth, **int** \*pday)

{

**int** i, leap;

**char** \*p;

leap = (year%**4** == **0** && year%**100** != **0**) || year%**400** == **0**;

p = &daytab[leap][**1**];

**for** (i = **1**; yearday > \*p; i++) {

yearday -= \*p;

++p;

}

\*pmonth = i;

\*pday = yearday;

}

**int** main(**void**)

{

**int** year, month, day, yearday;

year = **2000**;

month = **3**;

day = **1**;

printf("The date is: %d-%02d-%02d\n", year, month, day);

printf("day\_of\_year: %d\n", day\_of\_year(year, month, day));

printf("day\_of\_year\_pointer: %d\n",

day\_of\_year\_pointer(year, month, day));

yearday = **61**; /\* 2000-03-01 \*/

month\_day(year, yearday, &month, &day);

printf("Yearday is %d\n", yearday);

printf("month\_day: %d %d\n", month, day);

month\_day\_pointer(year, yearday, &month, &day);

printf("month\_day\_pointer: %d %d\n", month, day);

**return** **0**;

}

And here's Greg's: @br @br

/\* Gregory Pietsch - gkp1@flash.net \*/

/\* Given the problem, I thought that this would be a better

\* description of daytab.

\*/

**static** **int** \*daytab = {

**0**,

**31**,

**31**+**28**,

**31**+**28**+**31**,

**31**+**28**+**31**+**30**,

**31**+**28**+**31**+**30**+**31**,

**31**+**28**+**31**+**30**+**31**+**30**,

**31**+**28**+**31**+**30**+**31**+**30**+**31**,

**31**+**28**+**31**+**30**+**31**+**30**+**31**+**31**,

**31**+**28**+**31**+**30**+**31**+**30**+**31**+**31**+**30**,

**31**+**28**+**31**+**30**+**31**+**30**+**31**+**31**+**30**+**31**,

**31**+**28**+**31**+**30**+**31**+**30**+**31**+**31**+**30**+**31**+**30**,

**0**,

**31**,

**31**+**29**,

**31**+**29**+**31**,

**31**+**29**+**31**+**30**,

**31**+**29**+**31**+**30**+**31**,

**31**+**29**+**31**+**30**+**31**+**30**,

**31**+**29**+**31**+**30**+**31**+**30**+**31**,

**31**+**29**+**31**+**30**+**31**+**30**+**31**+**31**,

**31**+**29**+**31**+**30**+**31**+**30**+**31**+**31**+**30**,

**31**+**29**+**31**+**30**+**31**+**30**+**31**+**31**+**30**+**31**,

**31**+**29**+**31**+**30**+**31**+**30**+**31**+**31**+**30**+**31**+**30**,

};

/\* is it a leap year? (assume it's my calendar, the Gregorian) \*/

**int** leap(**int** year)

{

**return** ((year % **4**) == **0**)

&& (((year % **100**) != **0**)

|| (year % **400**) == **0**)));

}

/\* day\_of\_year: set day of year from month & day \*/

**int** day\_of\_year(**int** year, **int** month, **int** day)

{

**return** \*(daytab + ((month - **1**) + (leap(year) \* **12**))) + day;

}

/\* month\_day: set month, day from day of year \*/

**void** month\_day(**int** year, **int** yearday, **int** \*pmonth, **int** \*pday)

{

**int** m, ly;

ly = leap(year);

**if** (yearday < **1** || yearday > (**365** + ly))

**return**; /\* no real error checking \*/

m = leap(year) ? **23** : **11**;

**while** (\*(daytab + m) > yearday)

m--;

**if** (pmonth)

\*pmonth = (m % **12**) + **1**;

**if** (pday)

\*pday = yearday - (\*(daytab + m));

}

**Answer to Exercise 8-6, page 189**

*The standard library function calloc(n,size) returns a pointer to n objects of size size , with the storage initialized to zero. Write calloc , by calling malloc or by modifying it.*

/\*

Exercise 8.6. The standard library function calloc(n, size) returns a pointer to n objects

of size size, with the storage initialised to zero. Write calloc, by calling

malloc or by modifying it.

Author: Bryan Williams

\*/

**#include <stdlib.h>**

**#include <string.h>**

/\*

Decided to re-use malloc for this because :

1) If the implementation of malloc and the memory management layer changes, this will be ok.

2) Code re-use is great.

\*/

**void** \*mycalloc(size\_t nmemb, size\_t size)

{

**void** \*Result = NULL;

/\* use malloc to get the memory \*/

Result = malloc(nmemb \* size);

/\* and clear the memory on successful allocation \*/

**if**(NULL != Result)

{

memset(Result, **0x00**, nmemb \* size);

}

/\* and return the result \*/

**return** Result;

}

/\* simple test driver, by RJH \*/

**#include <stdio.h>**

**int** main(**void**)

{

**int** \*p = NULL;

**int** i = **0**;

p = mycalloc(**100**, **sizeof** \*p);

**if**(NULL == p)

{

printf("mycalloc returned NULL.\n");

}

**else**

{

**for**(i = **0**; i < **100**; i++)

{

printf("%08X ", p[i]);

**if**(i % **8** == **7**)

{

printf("\n");

}

}

printf("\n");

free(p);

}

**return** **0**;

}

**Answer to Exercise 5-10, page 118**

*Write the program expr , which evaluates a reverse Polish expression from the command line, where each operator or operand is a separate argument. For example,   
expr 2 3 4 + \*   
evaluates 2 X (3 + 4).*   
  
Note: Lars uses EXIT\_FAILURE on error. As far as I can tell, this is the only thing which makes this a Category 1, rather than Category 0, solution.

/\*

\* Solution to exercise 5-10 in K&R2:

\*

\* Write the program expr, which evaluates a reverse Polish expression

\* from the command line, where each operator or operand is a separate

\* argument. For example,

\*

\* expr 2 3 4 + \*

\*

\* evaluates 2\*(3+4).

\*

\* This is very similar to the program in 4.3 (and should ideally have been

\* a modification of that).

\*

\* Feel free to modify and copy freely.

\*

\* Lars Wirzenius <liw@iki.fi>

\*/

**#include <ctype.h>**

**#include <stdio.h>**

**#include <stdlib.h>**

**#define STACK\_SIZE 1024**

**double** stack[STACK\_SIZE];

**int** stack\_height = **0**;

**void** panic(**const** **char** \*msg) {

fprintf(stderr, "%s\n", msg);

exit(EXIT\_FAILURE);

}

**void** push(**double** value) {

**if** (stack\_height == STACK\_SIZE)

panic("stack is too high!");

stack[stack\_height] = value;

++stack\_height;

}

**double** pop(**void**) {

**if** (stack\_height == **0**)

panic("stack is empty!");

**return** stack[--stack\_height];

}

**int** main(**int** argc, **char** \*\*argv) {

**int** i;

**double** value;

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

**switch** (argv[i][**0**]) {

**case** **'\0'**:

panic("empty command line argument");

**break**;

**case** **'0'**:

**case** **'1'**:

**case** **'2'**:

**case** **'3'**:

**case** **'4'**:

**case** **'5'**:

**case** **'6'**:

**case** **'7'**:

**case** **'8'**:

**case** **'9'**:

push(atof(argv[i]));

**break**;

**case** **'+'**:

push(pop() + pop());

**break**;

**case** **'-'**:

value = pop();

push(pop() - value);

**break**;

**case** **'\*'**:

push(pop() \* pop());

**break**;

**case** **'/'**:

value = pop();

push(pop() / value);

**break**;

**default**:

panic("unknown operator");

**break**;

}

}

printf("%g\n", pop());

**return** **0**;

}

**Answer to Exercise 5-11, page 118**

*Modify the programs entab and detab (written as exercises in Chapter 1) to accept a list of tab stops as arguments. Use the default tab settings if there are no arguments.*   
  
  
Here's detab...

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

detab.c - Source code for the detab command

AUTHOR: Gregory Pietsch

DESCRIPTION:

detab - expand tabs into spaces

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* include files \*/

**#include <stdio.h>**

**#include <string.h>**

/\* macros \*/

**#define NO\_ARG 0**

**#define REQUIRED\_ARG 1**

**#define OPTIONAL\_ARG 2**

/\* types \*/

/\* GETOPT\_LONG\_OPTION\_T: The type of long option \*/

**typedef** **struct** GETOPT\_LONG\_OPTION\_T {

**char** \*name; /\* the name of the long option \*/

**int** has\_arg; /\* one of the above macros \*/

**int** \*flag; /\* determines if getopt\_long() returns a

\* value for a long option; if it is

\* non-NULL, 0 is returned as a function

\* value and the value of val is stored in

\* the area pointed to by flag. Otherwise,

\* val is returned. \*/

**int** val; /\* determines the value to return if flag is

\* NULL. \*/

} GETOPT\_LONG\_OPTION\_T;

**typedef** **enum** GETOPT\_ORDERING\_T {

PERMUTE,

RETURN\_IN\_ORDER,

REQUIRE\_ORDER

} GETOPT\_ORDERING\_T;

/\* globally-defined variables \*/

**char** \*optarg = NULL;

**int** optind = **0**;

**int** opterr = **1**;

**int** optopt = **'?'**;

/\* statically-defined variables \*/

**static** **char** \*program\_name;

/\* if nonzero, it means tab every x characters \*/

**static** **unsigned** **long** tab\_every = **8**;

/\* -i: only handle initial tabs/spaces \*/

**static** **int** flag\_initial = **0**;

/\* expand tabs into spaces \*/

**static** **int** flag\_expand = **1**;

**static** **unsigned** **long** \*tab\_stop\_list = NULL;

**static** size\_t num\_tab\_stops = **0**;

**static** size\_t num\_tab\_stops\_allocked = **0**;

**static** **int** show\_help = **0**;

**static** **int** show\_version = **0**;

**static** **char** \*shortopts = "it:";

**static** GETOPT\_LONG\_OPTION\_T longopts[] =

{

{"initial", NO\_ARG, NULL, **'i'**},

{"tabs", REQUIRED\_ARG, NULL, **'t'**},

{"help", NO\_ARG, &show\_help, **1**},

{"version", NO\_ARG, &show\_version, **1**},

{NULL, **0**, **0**, **0**}

};

/\* functions \*/

/\* reverse\_argv\_elements: reverses num elements starting at argv \*/

**static** **void** reverse\_argv\_elements(**char** \*\*argv, **int** num)

{

**int** i;

**char** \*tmp;

**for** (i = **0**; i < (num >> **1**); i++) {

tmp = argv[i];

argv[i] = argv[num - i - **1**];

argv[num - i - **1**] = tmp;

}

}

/\* permute: swap two blocks of argv-elements given their lengths \*/

**static** **void** permute(**char** \*\*argv, **int** len1, **int** len2)

{

reverse\_argv\_elements(argv, len1);

reverse\_argv\_elements(argv, len1 + len2);

reverse\_argv\_elements(argv, len2);

}

/\* is\_option: is this argv-element an option or the end of the option

list? \*/

**static** **int** is\_option(**char** \*argv\_element, **int** only)

{

**return** ((argv\_element == NULL)

|| (argv\_element[**0**] == **'-'**)

|| (only && argv\_element[**0**] == **'+'**));

}

/\* getopt\_internal: the function that does all the dirty work \*/

**static** **int** getopt\_internal(**int** argc, **char** \*\*argv, **char** \*shortopts,

GETOPT\_LONG\_OPTION\_T \* longopts, **int** \*longind, **int**

only)

{

GETOPT\_ORDERING\_T ordering = PERMUTE;

**static** size\_t optwhere = **0**;

size\_t permute\_from = **0**;

**int** num\_nonopts = **0**;

**int** optindex = **0**;

size\_t match\_chars = **0**;

**char** \*possible\_arg = NULL;

**int** longopt\_match = -**1**;

**int** has\_arg = -**1**;

**char** \*cp;

**int** arg\_next = **0**;

/\* first, deal with silly parameters and easy stuff \*/

**if** (argc == **0** || argv == NULL || (shortopts == NULL && longopts ==

NULL))

**return** (optopt = **'?'**);

**if** (optind >= argc || argv[optind] == NULL)

**return** EOF;

**if** (strcmp(argv[optind], "--") == **0**) {

optind++;

**return** EOF;

}

/\* if this is our first time through \*/

**if** (optind == **0**)

optind = optwhere = **1**;

/\* define ordering \*/

**if** (shortopts != NULL && (\*shortopts == **'-'** || \*shortopts == **'+'**)) {

ordering = (\*shortopts == **'-'**) ? RETURN\_IN\_ORDER :

REQUIRE\_ORDER;

shortopts++;

}

**else**

ordering = (getenv("POSIXLY\_CORRECT") != NULL) ? REQUIRE\_ORDER :

PERMUTE;

/\*

\* based on ordering, find our next option, if we're at the

beginning of

\* one

\*/

**if** (optwhere == **1**) {

**switch** (ordering) {

**case** PERMUTE:

permute\_from = optind;

num\_nonopts = **0**;

**while** (!is\_option(argv[optind], only)) {

optind++;

num\_nonopts++;

}

**if** (argv[optind] == NULL) {

/\* no more options \*/

optind = permute\_from;

**return** EOF;

} **else** **if** (strcmp(argv[optind], "--") == **0**) {

/\* no more options, but have to get `--' out of the way

\*/

permute(argv + permute\_from, num\_nonopts, **1**);

optind = permute\_from + **1**;

**return** EOF;

}

**break**;

**case** RETURN\_IN\_ORDER:

**if** (!is\_option(argv[optind], only)) {

optarg = argv[optind++];

**return** (optopt = **1**);

}

**break**;

**case** REQUIRE\_ORDER:

**if** (!is\_option(argv[optind], only))

**return** EOF;

**break**;

}

}

/\* we've got an option, so parse it \*/

/\* first, is it a long option? \*/

**if** (longopts != NULL

&& (memcmp(argv[optind], "--", **2**) == **0**

|| (only && argv[optind][**0**] == **'+'**))

&& optwhere == **1**) {

/\* handle long options \*/

**if** (memcmp(argv[optind], "--", **2**) == **0**)

optwhere = **2**;

longopt\_match = -**1**;

possible\_arg = strchr(argv[optind] + optwhere, **'='**);

**if** (possible\_arg == NULL) {

/\* no =, so next argv might be arg \*/

match\_chars = strlen(argv[optind]);

possible\_arg = argv[optind] + match\_chars;

match\_chars = match\_chars - optwhere;

}

**else**

match\_chars = (possible\_arg - argv[optind]) - optwhere;

**for** (optindex = **0**; longopts[optindex].name != NULL; optindex++)

{

**if** (memcmp(argv[optind] + optwhere,

longopts[optindex].name,

match\_chars) == **0**) {

/\* do we have an exact match? \*/

**if** (match\_chars == (**int**)

(strlen(longopts[optindex].name))) {

longopt\_match = optindex;

**break**;

}

/\* do any characters match? \*/

**else** {

**if** (longopt\_match < **0**)

longopt\_match = optindex;

**else** {

/\* we have ambiguous options \*/

**if** (opterr)

fprintf(stderr, "%s: option `%s' is

ambiguous "

"(could be `--%s' or `--%s')\n",

argv[**0**],

argv[optind],

longopts[longopt\_match].name,

longopts[optindex].name);

**return** (optopt = **'?'**);

}

}

}

}

**if** (longopt\_match >= **0**)

has\_arg = longopts[longopt\_match].has\_arg;

}

/\* if we didn't find a long option, is it a short option? \*/

**if** (longopt\_match < **0** && shortopts != NULL) {

cp = strchr(shortopts, argv[optind][optwhere]);

**if** (cp == NULL) {

/\* couldn't find option in shortopts \*/

**if** (opterr)

fprintf(stderr,

"%s: invalid option -- `-%c'\n",

argv[**0**],

argv[optind][optwhere]);

optwhere++;

**if** (argv[optind][optwhere] == **'\0'**) {

optind++;

optwhere = **1**;

}

**return** (optopt = **'?'**);

}

has\_arg = ((cp[**1**] == **':'**)

? ((cp[**2**] == **':'**) ? OPTIONAL\_ARG : REQUIRED\_ARG)

: NO\_ARG);

possible\_arg = argv[optind] + optwhere + **1**;

optopt = \*cp;

}

/\* get argument and reset optwhere \*/

arg\_next = **0**;

**switch** (has\_arg) {

**case** OPTIONAL\_ARG:

**if** (\*possible\_arg == **'='**)

possible\_arg++;

**if** (\*possible\_arg != **'\0'**) {

optarg = possible\_arg;

optwhere = **1**;

}

**else**

optarg = NULL;

**break**;

**case** REQUIRED\_ARG:

**if** (\*possible\_arg == **'='**)

possible\_arg++;

**if** (\*possible\_arg != **'\0'**) {

optarg = possible\_arg;

optwhere = **1**;

}

**else** **if** (optind + **1** >= argc) {

**if** (opterr) {

fprintf(stderr, "%s: argument required for option `",

argv[**0**]);

**if** (longopt\_match >= **0**)

fprintf(stderr, "--%s'\n",

longopts[longopt\_match].name);

**else**

fprintf(stderr, "-%c'\n", \*cp);

}

optind++;

**return** (optopt = **':'**);

}

**else** {

optarg = argv[optind + **1**];

arg\_next = **1**;

optwhere = **1**;

}

**break**;

**case** NO\_ARG:

**if** (longopt\_match < **0**) {

optwhere++;

**if** (argv[optind][optwhere] == **'\0'**)

optwhere = **1**;

}

**else**

optwhere = **1**;

optarg = NULL;

**break**;

}

/\* do we have to permute or otherwise modify optind? \*/

**if** (ordering == PERMUTE && optwhere == **1** && num\_nonopts != **0**) {

permute(argv + permute\_from, num\_nonopts, **1** + arg\_next);

optind = permute\_from + **1** + arg\_next;

}

**else** **if** (optwhere == **1**)

optind = optind + **1** + arg\_next;

/\* finally return \*/

**if** (longopt\_match >= **0**) {

**if** (longind != NULL)

\*longind = longopt\_match;

**if** (longopts[longopt\_match].flag != NULL) {

\*(longopts[longopt\_match].flag) =

longopts[longopt\_match].val;

**return** **0**;

}

**else**

**return** longopts[longopt\_match].val;

}

**else**

**return** optopt;

}

**int** getopt\_long(**int** argc, **char** \*\*argv, **char** \*shortopts,

GETOPT\_LONG\_OPTION\_T \* longopts, **int** \*longind)

{

**return** getopt\_internal(argc, argv, shortopts, longopts, longind, **0**);

}

**void** help(**void**)

{

puts( "OPTIONS" );

puts( "" );

puts( "-i, --initial When shrinking, make"

" initial spaces/tabs on a line tabs" );

puts( " and expand every other"

" tab on the line into spaces." );

puts( "-t=tablist, "

"Specify list of tab stops. "

"Default is every 8 characters." );

puts( "--tabs=tablist, "

"The parameter tablist is a list"

" of tab stops separated by" );

puts( "-tablist "

"commas; if no commas are present,"

" the program will put a" );

puts( " "

"tab stop every x places, "

"with x being the number in the" );

puts( " parameter." );

puts( "" );

puts( "--help Print usage message"

" and exit successfully." );

puts( "" );

puts( "--version Print version "

"information and exit successfully." );

}

**void** version(**void**)

{

puts( "detab - expand tabs into spaces" );

puts( "Version 1.0" );

puts( "Written by Gregory Pietsch" );

}

/\* allocate memory, die on error \*/

**void** \*xmalloc(size\_t n)

{

**void** \*p = malloc(n);

**if** (p == NULL) {

fprintf(stderr, "%s: out of memory\n", program\_name);

exit(EXIT\_FAILURE);

}

**return** p;

}

/\* reallocate memory, die on error \*/

**void** \*xrealloc(**void** \*p, size\_t n)

{

**void** \*s;

**if** (n == **0**) {

**if** (p != NULL)

free(p);

**return** NULL;

}

**if** (p == NULL)

**return** xmalloc(n);

s = realloc(p, n);

**if** (s == NULL) {

fprintf(stderr, "%s: out of memory\n", program\_name);

exit(EXIT\_FAILURE);

}

**return** s;

}

/\* Determine the location of the first character in the string s1

\* that is not a character in s2. The terminating null is not

\* considered part of the string.

\*/

**char** \*xstrcpbrk(**char** \*s1, **char** \*s2)

{

**char** \*sc1;

**char** \*sc2;

**for** (sc1 = s1; \*sc1 != **'\0'**; sc1++)

**for** (sc2 = s2;; sc2++)

**if** (\*sc2 == **'\0'**)

**return** sc1;

**else** **if** (\*sc1 == \*sc2)

**break**;

**return** NULL; /\* terminating nulls match \*/

}

/\* compare function for qsort() \*/

**int** ul\_cmp(**const** **void** \*a, **const** **void** \*b)

{

**unsigned** **long** \*ula = (**unsigned** **long** \*) a;

**unsigned** **long** \*ulb = (**unsigned** **long** \*) b;

**return** (\*ula < \*ulb) ? -**1** : (\*ula > \*ulb);

}

/\* handle a tab stop list -- assumes param isn't NULL \*/

**void** handle\_tab\_stops(**char** \*s)

{

**char** \*p;

**unsigned** **long** ul;

size\_t len = strlen(s);

**if** (xstrcpbrk(s, "0123456789,") != NULL) {

/\* funny param \*/

fprintf(stderr, "%s: invalid parameter\n", program\_name);

exit(EXIT\_FAILURE);

}

**if** (strchr(s, **','**) == NULL) {

tab\_every = strtoul(s, NULL, **10**);

**if** (tab\_every == **0**)

tab\_every = **8**;

}

**else** {

tab\_stop\_list = xrealloc(tab\_stop\_list,

(num\_tab\_stops\_allocked += len) \* (**sizeof**(**unsigned**

**long**)));

**for** (p = s; (p = strtok(p, ",")) != NULL; p = NULL) {

ul = strtoul(p, NULL, **10**);

tab\_stop\_list[num\_tab\_stops++] = ul;

}

qsort(tab\_stop\_list, num\_tab\_stops, **sizeof**(**unsigned** **long**),

ul\_cmp);

}

}

**void** parse\_args(**int** argc, **char** \*\*argv)

{

**int** opt;

**do** {

**switch** ((opt = getopt\_long(argc, argv, shortopts, longopts,

NULL))) {

**case** **'i'**: /\* initial \*/

flag\_initial = **1**;

**break**;

**case** **'t'**: /\* tab stops \*/

handle\_tab\_stops(optarg);

**break**;

**case** **'?'**: /\* invalid option \*/

fprintf(stderr,"For help, type:\n\t%s --help\n",

program\_name);

exit(EXIT\_FAILURE);

**case** **1**:

**case** **0**:

**if** (show\_help || show\_version) {

**if** (show\_help)

help();

**if** (show\_version)

version();

exit(EXIT\_SUCCESS);

}

**break**;

**default**:

**break**;

}

} **while** (opt != EOF);

}

/\* output exactly n spaces \*/

**void** output\_spaces(size\_t n)

{

**int** x = n; /\* assume n is small \*/

printf("%\*s", x, "");

}

/\* get next highest tab stop \*/

**unsigned** **long** get\_next\_tab(**unsigned** **long** x)

{

size\_t i;

**if** (tab\_stop\_list == NULL) {

/\* use tab\_every \*/

x += (tab\_every - (x % tab\_every));

**return** x;

}

**else** {

**for** (i = **0**; i < num\_tab\_stops && tab\_stop\_list[i] <= x; i++);

**return** (i >= num\_tab\_stops) ? **0** : tab\_stop\_list[i];

}

}

/\* the function that does the dirty work \*/

**void** tab(FILE \* f)

{

**unsigned** **long** linelength = **0**;

**int** c;

**int** in\_initials = **1**;

size\_t num\_spaces = **0**;

**unsigned** **long** next\_tab;

**while** ((c = getc(f)) != EOF) {

**if** (c != **' '** && c != **'\t'** && num\_spaces > **0**) {

/\* output spaces and possible tabs \*/

**if** (flag\_expand

|| (flag\_initial && !in\_initials)

|| num\_spaces == **1**) {

/\* output spaces anyway \*/

output\_spaces(num\_spaces);

linelength += num\_spaces;

num\_spaces = **0**;

}

**else**

**while** (num\_spaces != **0**) {

next\_tab = get\_next\_tab(linelength);

**if** (next\_tab > **0** && next\_tab <= linelength +

num\_spaces) {

/\* output a tab \*/

putc(**'\t'**, stdout);

num\_spaces -= (next\_tab - linelength);

linelength = next\_tab;

}

**else** {

/\* output spaces \*/

output\_spaces(num\_spaces);

linelength += num\_spaces;

num\_spaces = **0**;

}

}

}

**switch** (c) {

**case** **' '**: /\* space \*/

num\_spaces++;

**break**;

**case** **'\b'**: /\* backspace \*/

/\* preserve backspaces in output; decrement length for

tabbing

\* purposes

\*/

putc(c, stdout);

**if** (linelength > **0**)

linelength--;

**break**;

**case** **'\n'**: /\* newline \*/

putc(c, stdout);

in\_initials = **1**;

linelength = **0**;

**break**;

**case** **'\t'**: /\* tab \*/

next\_tab = get\_next\_tab(linelength + num\_spaces);

**if** (next\_tab == **0**) {

**while** ((next\_tab = get\_next\_tab(linelength)) != **0**) {

/\* output tabs \*/

putc(**'\t'**, stdout);

num\_spaces -= (next\_tab - linelength);

linelength = next\_tab;

}

/\* output spaces \*/

output\_spaces(num\_spaces);

num\_spaces = **0**;

putc(**'\t'**, stdout);

linelength += num\_spaces + **1**;

}

**else**

num\_spaces = next\_tab - linelength;

**break**;

**default**:

putc(c, stdout);

in\_initials = **0**;

linelength++;

**break**;

}

}

}

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

{

**int** i;

FILE \*fp;

**char** \*allocked\_argvs = xmalloc(argc + **1**);

**char** \*\*new\_argv = xmalloc((argc + **1**) \* **sizeof**(**char** \*));

**char** \*p;

program\_name = argv[**0**];

memset(allocked\_argvs, **0**, argc + **1**);

**for** (i = **0**; i < argc; i++) {

p = argv[i];

**if** (isdigit(p[**1**])) {

new\_argv[i] = xmalloc(strlen(p) + **2**);

sprintf(new\_argv[i], "-t%s", p + **1**);

allocked\_argvs[i] = **1**;

}

**else**

new\_argv[i] = p;

}

new\_argv[argc] = NULL;

parse\_args(argc, new\_argv);

**if** (optind == argc)

tab(stdin);

**else** {

**for** (i = optind; i < argc; i++) {

**if** (strcmp(argv[i], "-") == **0**)

fp = stdin;

**else** {

fp = fopen(argv[i], "r");

**if** (fp == NULL) {

fprintf(stderr, "%s: can't open %s\n",

argv[**0**], argv[i]);

abort();

}

}

tab(fp);

**if** (fp != stdin)

fclose(fp);

}

}

/\* free everything we can \*/

**for** (i = **0**; i < argc; i++)

**if** (allocked\_argvs[i])

free(new\_argv[i]);

free(allocked\_argvs);

**if** (tab\_stop\_list != NULL)

free(tab\_stop\_list);

**return** EXIT\_SUCCESS;

}

/\* END OF FILE detab.c \*/

Here's entab...

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

entab.c - Source code for the detab command

AUTHOR: Gregory Pietsch

DESCRIPTION:

entab - shrinks spaces into tabs

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* include files \*/

**#include <stdio.h>**

**#include <string.h>**

/\* macros \*/

**#define NO\_ARG 0**

**#define REQUIRED\_ARG 1**

**#define OPTIONAL\_ARG 2**

/\* types \*/

/\* GETOPT\_LONG\_OPTION\_T: The type of long option \*/

**typedef** **struct** GETOPT\_LONG\_OPTION\_T {

**char** \*name; /\* the name of the long option \*/

**int** has\_arg; /\* one of the above macros \*/

**int** \*flag; /\* determines if getopt\_long() returns a

\* value for a long option; if it is

\* non-NULL, 0 is returned as a function

\* value and the value of val is stored in

\* the area pointed to by flag. Otherwise,

\* val is returned. \*/

**int** val; /\* determines the value to return if flag is

\* NULL. \*/

} GETOPT\_LONG\_OPTION\_T;

**typedef** **enum** GETOPT\_ORDERING\_T {

PERMUTE,

RETURN\_IN\_ORDER,

REQUIRE\_ORDER

} GETOPT\_ORDERING\_T;

/\* globally-defined variables \*/

**char** \*optarg = NULL;

**int** optind = **0**;

**int** opterr = **1**;

**int** optopt = **'?'**;

/\* statically-defined variables \*/

**static** **char** \*program\_name;

/\* if nonzero, it means tab every x characters \*/

**static** **unsigned** **long** tab\_every = **8**;

/\* -i: only handle initial tabs/spaces \*/

**static** **int** flag\_initial = **0**;

/\* don't expand tabs into spaces \*/

**static** **int** flag\_expand = **0**;

**static** **unsigned** **long** \*tab\_stop\_list = NULL;

**static** size\_t num\_tab\_stops = **0**;

**static** size\_t num\_tab\_stops\_allocked = **0**;

**static** **int** show\_help = **0**;

**static** **int** show\_version = **0**;

**static** **char** \*shortopts = "it:";

**static** GETOPT\_LONG\_OPTION\_T longopts[] =

{

{"initial", NO\_ARG, NULL, **'i'**},

{"tabs", REQUIRED\_ARG, NULL, **'t'**},

{"help", NO\_ARG, &show\_help, **1**},

{"version", NO\_ARG, &show\_version, **1**},

{NULL, **0**, **0**, **0**}

};

/\* functions \*/

/\* reverse\_argv\_elements: reverses num elements starting at argv \*/

**static** **void** reverse\_argv\_elements(**char** \*\*argv, **int** num)

{

**int** i;

**char** \*tmp;

**for** (i = **0**; i < (num >> **1**); i++) {

tmp = argv[i];

argv[i] = argv[num - i - **1**];

argv[num - i - **1**] = tmp;

}

}

/\* permute: swap two blocks of argv-elements given their lengths \*/

**static** **void** permute(**char** \*\*argv, **int** len1, **int** len2)

{

reverse\_argv\_elements(argv, len1);

reverse\_argv\_elements(argv, len1 + len2);

reverse\_argv\_elements(argv, len2);

}

/\* is\_option: is this argv-element an option or the end of the option

list? \*/

**static** **int** is\_option(**char** \*argv\_element, **int** only)

{

**return** ((argv\_element == NULL)

|| (argv\_element[**0**] == **'-'**)

|| (only && argv\_element[**0**] == **'+'**));

}

/\* getopt\_internal: the function that does all the dirty work \*/

**static** **int** getopt\_internal(**int** argc, **char** \*\*argv, **char** \*shortopts,

GETOPT\_LONG\_OPTION\_T \* longopts, **int** \*longind, **int**

only)

{

GETOPT\_ORDERING\_T ordering = PERMUTE;

**static** size\_t optwhere = **0**;

size\_t permute\_from = **0**;

**int** num\_nonopts = **0**;

**int** optindex = **0**;

size\_t match\_chars = **0**;

**char** \*possible\_arg = NULL;

**int** longopt\_match = -**1**;

**int** has\_arg = -**1**;

**char** \*cp;

**int** arg\_next = **0**;

/\* first, deal with silly parameters and easy stuff \*/

**if** (argc == **0** || argv == NULL || (shortopts == NULL && longopts ==

NULL))

**return** (optopt = **'?'**);

**if** (optind >= argc || argv[optind] == NULL)

**return** EOF;

**if** (strcmp(argv[optind], "--") == **0**) {

optind++;

**return** EOF;

}

/\* if this is our first time through \*/

**if** (optind == **0**)

optind = optwhere = **1**;

/\* define ordering \*/

**if** (shortopts != NULL && (\*shortopts == **'-'** || \*shortopts == **'+'**)) {

ordering = (\*shortopts == **'-'**) ? RETURN\_IN\_ORDER :

REQUIRE\_ORDER;

shortopts++;

}

**else**

ordering = (getenv("POSIXLY\_CORRECT") != NULL) ? REQUIRE\_ORDER :

PERMUTE;

/\*

\* based on ordering, find our next option, if we're at the

beginning of

\* one

\*/

**if** (optwhere == **1**) {

**switch** (ordering) {

**case** PERMUTE:

permute\_from = optind;

num\_nonopts = **0**;

**while** (!is\_option(argv[optind], only)) {

optind++;

num\_nonopts++;

}

**if** (argv[optind] == NULL) {

/\* no more options \*/

optind = permute\_from;

**return** EOF;

} **else** **if** (strcmp(argv[optind], "--") == **0**) {

/\* no more options, but have to get `--' out of the way

\*/

permute(argv + permute\_from, num\_nonopts, **1**);

optind = permute\_from + **1**;

**return** EOF;

}

**break**;

**case** RETURN\_IN\_ORDER:

**if** (!is\_option(argv[optind], only)) {

optarg = argv[optind++];

**return** (optopt = **1**);

}

**break**;

**case** REQUIRE\_ORDER:

**if** (!is\_option(argv[optind], only))

**return** EOF;

**break**;

}

}

/\* we've got an option, so parse it \*/

/\* first, is it a long option? \*/

**if** (longopts != NULL

&& (memcmp(argv[optind], "--", **2**) == **0**

|| (only && argv[optind][**0**] == **'+'**))

&& optwhere == **1**) {

/\* handle long options \*/

**if** (memcmp(argv[optind], "--", **2**) == **0**)

optwhere = **2**;

longopt\_match = -**1**;

possible\_arg = strchr(argv[optind] + optwhere, **'='**);

**if** (possible\_arg == NULL) {

/\* no =, so next argv might be arg \*/

match\_chars = strlen(argv[optind]);

possible\_arg = argv[optind] + match\_chars;

match\_chars = match\_chars - optwhere;

}

**else**

match\_chars = (possible\_arg - argv[optind]) - optwhere;

**for** (optindex = **0**; longopts[optindex].name != NULL; optindex++)

{

**if** (memcmp(argv[optind] + optwhere,

longopts[optindex].name,

match\_chars) == **0**) {

/\* do we have an exact match? \*/

**if** (match\_chars == (**int**)

(strlen(longopts[optindex].name))) {

longopt\_match = optindex;

**break**;

}

/\* do any characters match? \*/

**else** {

**if** (longopt\_match < **0**)

longopt\_match = optindex;

**else** {

/\* we have ambiguous options \*/

**if** (opterr)

fprintf(stderr, "%s: option `%s' is

ambiguous "

"(could be `--%s' or `--%s')\n",

argv[**0**],

argv[optind],

longopts[longopt\_match].name,

longopts[optindex].name);

**return** (optopt = **'?'**);

}

}

}

}

**if** (longopt\_match >= **0**)

has\_arg = longopts[longopt\_match].has\_arg;

}

/\* if we didn't find a long option, is it a short option? \*/

**if** (longopt\_match < **0** && shortopts != NULL) {

cp = strchr(shortopts, argv[optind][optwhere]);

**if** (cp == NULL) {

/\* couldn't find option in shortopts \*/

**if** (opterr)

fprintf(stderr,

"%s: invalid option -- `-%c'\n",

argv[**0**],

argv[optind][optwhere]);

optwhere++;

**if** (argv[optind][optwhere] == **'\0'**) {

optind++;

optwhere = **1**;

}

**return** (optopt = **'?'**);

}

has\_arg = ((cp[**1**] == **':'**)

? ((cp[**2**] == **':'**) ? OPTIONAL\_ARG : REQUIRED\_ARG)

: NO\_ARG);

possible\_arg = argv[optind] + optwhere + **1**;

optopt = \*cp;

}

/\* get argument and reset optwhere \*/

arg\_next = **0**;

**switch** (has\_arg) {

**case** OPTIONAL\_ARG:

**if** (\*possible\_arg == **'='**)

possible\_arg++;

**if** (\*possible\_arg != **'\0'**) {

optarg = possible\_arg;

optwhere = **1**;

}

**else**

optarg = NULL;

**break**;

**case** REQUIRED\_ARG:

**if** (\*possible\_arg == **'='**)

possible\_arg++;

**if** (\*possible\_arg != **'\0'**) {

optarg = possible\_arg;

optwhere = **1**;

}

**else** **if** (optind + **1** >= argc) {

**if** (opterr) {

fprintf(stderr, "%s: argument required for option `",

argv[**0**]);

**if** (longopt\_match >= **0**)

fprintf(stderr, "--%s'\n",

longopts[longopt\_match].name);

**else**

fprintf(stderr, "-%c'\n", \*cp);

}

optind++;

**return** (optopt = **':'**);

}

**else** {

optarg = argv[optind + **1**];

arg\_next = **1**;

optwhere = **1**;

}

**break**;

**case** NO\_ARG:

**if** (longopt\_match < **0**) {

optwhere++;

**if** (argv[optind][optwhere] == **'\0'**)

optwhere = **1**;

}

**else**

optwhere = **1**;

optarg = NULL;

**break**;

}

/\* do we have to permute or otherwise modify optind? \*/

**if** (ordering == PERMUTE && optwhere == **1** && num\_nonopts != **0**) {

permute(argv + permute\_from, num\_nonopts, **1** + arg\_next);

optind = permute\_from + **1** + arg\_next;

}

**else** **if** (optwhere == **1**)

optind = optind + **1** + arg\_next;

/\* finally return \*/

**if** (longopt\_match >= **0**) {

**if** (longind != NULL)

\*longind = longopt\_match;

**if** (longopts[longopt\_match].flag != NULL) {

\*(longopts[longopt\_match].flag) =

longopts[longopt\_match].val;

**return** **0**;

}

**else**

**return** longopts[longopt\_match].val;

}

**else**

**return** optopt;

}

**int** getopt\_long(**int** argc, **char** \*\*argv, **char** \*shortopts,

GETOPT\_LONG\_OPTION\_T \* longopts, **int** \*longind)

{

**return** getopt\_internal(argc, argv, shortopts, longopts, longind, **0**);

}

**void** help(**void**)

{

puts( "OPTIONS" );

puts( "" );

puts( "-i, --initial When shrinking, make"

" initial spaces/tabs on a line tabs" );

puts( " and expand every other"

" tab on the line into spaces." );

puts( "-t=tablist, "

"Specify list of tab stops. "

"Default is every 8 characters." );

puts( "--tabs=tablist, "

"The parameter tablist is a list"

" of tab stops separated by" );

puts( "-tablist "

"commas; if no commas are present,"

" the program will put a" );

puts( " "

"tab stop every x places, "

"with x being the number in the" );

puts( " parameter." );

puts( "" );

puts( "--help Print usage message"

" and exit successfully." );

puts( "" );

puts( "--version Print version "

"information and exit successfully." );

}

**void** version(**void**)

{

puts( "detab - expand tabs into spaces" );

puts( "Version 1.0" );

puts( "Written by Gregory Pietsch" );

}

/\* allocate memory, die on error \*/

**void** \*xmalloc(size\_t n)

{

**void** \*p = malloc(n);

**if** (p == NULL) {

fprintf(stderr, "%s: out of memory\n", program\_name);

exit(EXIT\_FAILURE);

}

**return** p;

}

/\* reallocate memory, die on error \*/

**void** \*xrealloc(**void** \*p, size\_t n)

{

**void** \*s;

**if** (n == **0**) {

**if** (p != NULL)

free(p);

**return** NULL;

}

**if** (p == NULL)

**return** xmalloc(n);

s = realloc(p, n);

**if** (s == NULL) {

fprintf(stderr, "%s: out of memory\n", program\_name);

exit(EXIT\_FAILURE);

}

**return** s;

}

/\* Determine the location of the first character in the string s1

\* that is not a character in s2. The terminating null is not

\* considered part of the string.

\*/

**char** \*xstrcpbrk(**char** \*s1, **char** \*s2)

{

**char** \*sc1;

**char** \*sc2;

**for** (sc1 = s1; \*sc1 != **'\0'**; sc1++)

**for** (sc2 = s2;; sc2++)

**if** (\*sc2 == **'\0'**)

**return** sc1;

**else** **if** (\*sc1 == \*sc2)

**break**;

**return** NULL; /\* terminating nulls match \*/

}

/\* compare function for qsort() \*/

**int** ul\_cmp(**const** **void** \*a, **const** **void** \*b)

{

**unsigned** **long** \*ula = (**unsigned** **long** \*) a;

**unsigned** **long** \*ulb = (**unsigned** **long** \*) b;

**return** (\*ula < \*ulb) ? -**1** : (\*ula > \*ulb);

}

/\* handle a tab stop list -- assumes param isn't NULL \*/

**void** handle\_tab\_stops(**char** \*s)

{

**char** \*p;

**unsigned** **long** ul;

size\_t len = strlen(s);

**if** (xstrcpbrk(s, "0123456789,") != NULL) {

/\* funny param \*/

fprintf(stderr, "%s: invalid parameter\n", program\_name);

exit(EXIT\_FAILURE);

}

**if** (strchr(s, **','**) == NULL) {

tab\_every = strtoul(s, NULL, **10**);

**if** (tab\_every == **0**)

tab\_every = **8**;

}

**else** {

tab\_stop\_list = xrealloc(tab\_stop\_list,

(num\_tab\_stops\_allocked += len) \* (**sizeof**(**unsigned**

**long**)));

**for** (p = s; (p = strtok(p, ",")) != NULL; p = NULL) {

ul = strtoul(p, NULL, **10**);

tab\_stop\_list[num\_tab\_stops++] = ul;

}

qsort(tab\_stop\_list, num\_tab\_stops, **sizeof**(**unsigned** **long**),

ul\_cmp);

}

}

**void** parse\_args(**int** argc, **char** \*\*argv)

{

**int** opt;

**do** {

**switch** ((opt = getopt\_long(argc, argv, shortopts, longopts,

NULL))) {

**case** **'i'**: /\* initial \*/

flag\_initial = **1**;

**break**;

**case** **'t'**: /\* tab stops \*/

handle\_tab\_stops(optarg);

**break**;

**case** **'?'**: /\* invalid option \*/

fprintf(stderr,"For help, type:\n\t%s --help\n",

program\_name);

exit(EXIT\_FAILURE);

**case** **1**:

**case** **0**:

**if** (show\_help || show\_version) {

**if** (show\_help)

help();

**if** (show\_version)

version();

exit(EXIT\_SUCCESS);

}

**break**;

**default**:

**break**;

}

} **while** (opt != EOF);

}

/\* output exactly n spaces \*/

**void** output\_spaces(size\_t n)

{

**int** x = n; /\* assume n is small \*/

printf("%\*s", x, "");

}

/\* get next highest tab stop \*/

**unsigned** **long** get\_next\_tab(**unsigned** **long** x)

{

size\_t i;

**if** (tab\_stop\_list == NULL) {

/\* use tab\_every \*/

x += (tab\_every - (x % tab\_every));

**return** x;

}

**else** {

**for** (i = **0**; i < num\_tab\_stops && tab\_stop\_list[i] <= x; i++);

**return** (i >= num\_tab\_stops) ? **0** : tab\_stop\_list[i];

}

}

/\* the function that does the dirty work \*/

**void** tab(FILE \* f)

{

**unsigned** **long** linelength = **0**;

**int** c;

**int** in\_initials = **1**;

size\_t num\_spaces = **0**;

**unsigned** **long** next\_tab;

**while** ((c = getc(f)) != EOF) {

**if** (c != **' '** && c != **'\t'** && num\_spaces > **0**) {

/\* output spaces and possible tabs \*/

**if** (flag\_expand

|| (flag\_initial && !in\_initials)

|| num\_spaces == **1**) {

/\* output spaces anyway \*/

output\_spaces(num\_spaces);

linelength += num\_spaces;

num\_spaces = **0**;

}

**else**

**while** (num\_spaces != **0**) {

next\_tab = get\_next\_tab(linelength);

**if** (next\_tab > **0** && next\_tab <= linelength +

num\_spaces) {

/\* output a tab \*/

putc(**'\t'**, stdout);

num\_spaces -= (next\_tab - linelength);

linelength = next\_tab;

}

**else** {

/\* output spaces \*/

output\_spaces(num\_spaces);

linelength += num\_spaces;

num\_spaces = **0**;

}

}

}

**switch** (c) {

**case** **' '**: /\* space \*/

num\_spaces++;

**break**;

**case** **'\b'**: /\* backspace \*/

/\* preserve backspaces in output; decrement length for

tabbing

\* purposes

\*/

putc(c, stdout);

**if** (linelength > **0**)

linelength--;

**break**;

**case** **'\n'**: /\* newline \*/

putc(c, stdout);

in\_initials = **1**;

linelength = **0**;

**break**;

**case** **'\t'**: /\* tab \*/

next\_tab = get\_next\_tab(linelength + num\_spaces);

**if** (next\_tab == **0**) {

**while** ((next\_tab = get\_next\_tab(linelength)) != **0**) {

/\* output tabs \*/

putc(**'\t'**, stdout);

num\_spaces -= (next\_tab - linelength);

linelength = next\_tab;

}

/\* output spaces \*/

output\_spaces(num\_spaces);

num\_spaces = **0**;

putc(**'\t'**, stdout);

linelength += num\_spaces + **1**;

}

**else**

num\_spaces = next\_tab - linelength;

**break**;

**default**:

putc(c, stdout);

in\_initials = **0**;

linelength++;

**break**;

}

}

}

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

{

**int** i;

FILE \*fp;

**char** \*allocked\_argvs = xmalloc(argc + **1**);

**char** \*\*new\_argv = xmalloc((argc + **1**) \* **sizeof**(**char** \*));

**char** \*p;

program\_name = argv[**0**];

memset(allocked\_argvs, **0**, argc + **1**);

**for** (i = **0**; i < argc; i++) {

p = argv[i];

**if** (isdigit(p[**1**])) {

new\_argv[i] = xmalloc(strlen(p) + **2**);

sprintf(new\_argv[i], "-t%s", p + **1**);

allocked\_argvs[i] = **1**;

}

**else**

new\_argv[i] = p;

}

new\_argv[argc] = NULL;

parse\_args(argc, new\_argv);

**if** (optind == argc)

tab(stdin);

**else** {

**for** (i = optind; i < argc; i++) {

**if** (strcmp(argv[i], "-") == **0**)

fp = stdin;

**else** {

fp = fopen(argv[i], "r");

**if** (fp == NULL) {

fprintf(stderr, "%s: can't open %s\n",

argv[**0**], argv[i]);

abort();

}

}

tab(fp);

**if** (fp != stdin)

fclose(fp);

}

}

/\* free everything we can \*/

**for** (i = **0**; i < argc; i++)

**if** (allocked\_argvs[i])

free(new\_argv[i]);

free(allocked\_argvs);

**if** (tab\_stop\_list != NULL)

free(tab\_stop\_list);

**return** EXIT\_SUCCESS;

}

/\* END OF FILE entab.c \*/

**Answer to Exercise 5-13, page 118**

*Write the program tail, which prints the last n lines of its input. By default, n is 10, say, but it can be changed by an optional argument, so that*

*tail -n*

*prints the last n lines. The program should behave rationally no matter how unreasonable the input or the value of n. Write the program so it makes the best use of available storage; lines should be stored as in the sorting program of Section 5.6, not in a two-dimensional array of fixed size.*   
Gregory Pietsch's solution

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

tail.c - Source code for the tail command

AUTHOR: Gregory Pietsch <gkp1@flash.net>

DESCRIPTION:

tail prints the last part of each file on the command line (10 lines by

default); it reads from standard input if no files are given or when a

filename of `-' is encountered. If more than one file is given, it

prints a header consisting of the file's name enclosed in `==>' and `<==' before

the output for each file.

There are two option formats for tail: the new one, in which numbers are

arguments to the option letters; and the old one, in which the number

precedes any option letters. In this version, the old format is barely

supported. Supporting it fully is left as an exercise to the reader ;-).

GNU's -f (or --follow) option is not supported. With that option, the

program loops forever on the assumption that the file being tailed is

growing. I couldn't figure out how to determine if the program is reading

from a pipe in ANSI C; this option is ignored if reading from a pipe.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* include files \*/

**#include <ctype.h>**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <string.h>**

/\* macros \*/

**#define NO\_ARG 0**

**#define REQUIRED\_ARG 1**

**#define OPTIONAL\_ARG 2**

/\* how many characters will fill one's tail (literally) \*/

**#define TAIL\_BUFFER\_SIZE 16384**

/\* how much for a string buffer \*/

**#define TAIL\_STRING\_BUFFER\_SIZE 256**

/\* need MIN \*/

**#ifndef MIN**

**#define MIN(x,y) ((x)<(y)?(x):(y))**

**#endif**

/\* types \*/

**typedef** **enum** VERBOSITY\_T {

NEVER,

SOMETIMES,

ALWAYS

} VERBOSITY\_T;

**typedef** **struct** LINE\_QUEUE\_EL\_T {

**char** \*s;

**struct** LINE\_QUEUE\_EL\_T \*next;

} LINE\_QUEUE\_EL\_T;

**typedef** **struct** LINE\_QUEUE\_T {

**struct** LINE\_QUEUE\_EL\_T \*first;

**struct** LINE\_QUEUE\_EL\_T \*last;

**unsigned** **long** num\_elements;

} LINE\_QUEUE\_T;

/\* GETOPT\_LONG\_OPTION\_T: The type of long option \*/

**typedef** **struct** GETOPT\_LONG\_OPTION\_T {

**char** \*name; /\* the name of the long option \*/

**int** has\_arg; /\* one of the above macros \*/

**int** \*flag; /\* determines if getopt\_long() returns a

\* value for a long option; if it is

\* non-NULL, 0 is returned as a function

\* value and the value of val is stored in

\* the area pointed to by flag. Otherwise,

\* val is returned. \*/

**int** val; /\* determines the value to return if flag is

\* NULL. \*/

} GETOPT\_LONG\_OPTION\_T;

**typedef** **enum** GETOPT\_ORDERING\_T {

PERMUTE,

RETURN\_IN\_ORDER,

REQUIRE\_ORDER

} GETOPT\_ORDERING\_T;

/\* globally-defined variables \*/

**char** \*optarg = NULL;

**int** optind = **0**;

**int** opterr = **1**;

**int** optopt = **'?'**;

/\* statically-defined variables \*/

**static** **int** show\_help = **0**;

**static** **int** show\_version = **0**;

**static** **char** \*shortopts = "c:l:n:qv";

**static** GETOPT\_LONG\_OPTION\_T longopts[] =

{

{"bytes", REQUIRED\_ARG, NULL, **'c'**},

{"lines", REQUIRED\_ARG, NULL, **'n'**},

{"quiet", NO\_ARG, NULL, **'q'**},

{"silent", NO\_ARG, NULL, **'q'**},

{"verbose", NO\_ARG, NULL, **'v'**},

{"help", NO\_ARG, &show\_help, **1**},

{"version", NO\_ARG, &show\_version, **1**},

{NULL, **0**, **0**, **0**}

};

**static** **char** \*program\_name;

**static** **int** flag\_bytes = **0**;

**static** VERBOSITY\_T flag\_verbosity = SOMETIMES;

**static** **unsigned** **long** number = **0**;

**static** **int** flag\_skip = **0**;

/\* functions \*/

/\* reverse\_argv\_elements: reverses num elements starting at argv \*/

**static** **void** reverse\_argv\_elements(**char** \*\*argv, **int** num)

{

**int** i;

**char** \*tmp;

**for** (i = **0**; i < (num >> **1**); i++) {

tmp = argv[i];

argv[i] = argv[num - i - **1**];

argv[num - i - **1**] = tmp;

}

}

/\* permute: swap two blocks of argv-elements given their lengths \*/

**static** **void** permute(**char** \*\*argv, **int** len1, **int** len2)

{

reverse\_argv\_elements(argv, len1);

reverse\_argv\_elements(argv, len1 + len2);

reverse\_argv\_elements(argv, len2);

}

/\* is\_option: is this argv-element an option or the end of the option list? \*/

**static** **int** is\_option(**char** \*argv\_element, **int** only)

{

**return** ((argv\_element == NULL)

|| (argv\_element[**0**] == **'-'**)

|| (only && argv\_element[**0**] == **'+'**));

}

/\* getopt\_internal: the function that does all the dirty work \*/

**static** **int** getopt\_internal(**int** argc, **char** \*\*argv, **char** \*shortopts,

GETOPT\_LONG\_OPTION\_T \* longopts, **int** \*longind, **int** only)

{

GETOPT\_ORDERING\_T ordering = PERMUTE;

**static** size\_t optwhere = **0**;

size\_t permute\_from = **0**;

**int** num\_nonopts = **0**;

**int** optindex = **0**;

size\_t match\_chars = **0**;

**char** \*possible\_arg = NULL;

**int** longopt\_match = -**1**;

**int** has\_arg = -**1**;

**char** \*cp;

**int** arg\_next = **0**;

/\* first, deal with silly parameters and easy stuff \*/

**if** (argc == **0** || argv == NULL || (shortopts == NULL && longopts == NULL))

**return** (optopt = **'?'**);

**if** (optind >= argc || argv[optind] == NULL)

**return** EOF;

**if** (strcmp(argv[optind], "--") == **0**) {

optind++;

**return** EOF;

}

/\* if this is our first time through \*/

**if** (optind == **0**)

optind = optwhere = **1**;

/\* define ordering \*/

**if** (shortopts != NULL && (\*shortopts == **'-'** || \*shortopts == **'+'**)) {

ordering = (\*shortopts == **'-'**) ? RETURN\_IN\_ORDER : REQUIRE\_ORDER;

shortopts++;

}

**else**

ordering = (getenv("POSIXLY\_CORRECT") != NULL) ? REQUIRE\_ORDER :

PERMUTE;

/\*

\* based on ordering, find our next option, if we're at the beginning of

\* one

\*/

**if** (optwhere == **1**) {

**switch** (ordering) {

**case** PERMUTE:

permute\_from = optind;

num\_nonopts = **0**;

**while** (!is\_option(argv[optind], only)) {

optind++;

num\_nonopts++;

}

**if** (argv[optind] == NULL) {

/\* no more options \*/

optind = permute\_from;

**return** EOF;

} **else** **if** (strcmp(argv[optind], "--") == **0**) {

/\* no more options, but have to get `--' out of the way \*/

permute(argv + permute\_from, num\_nonopts, **1**);

optind = permute\_from + **1**;

**return** EOF;

}

**break**;

**case** RETURN\_IN\_ORDER:

**if** (!is\_option(argv[optind], only)) {

optarg = argv[optind++];

**return** (optopt = **1**);

}

**break**;

**case** REQUIRE\_ORDER:

**if** (!is\_option(argv[optind], only))

**return** EOF;

**break**;

}

}

/\* we've got an option, so parse it \*/

/\* first, is it a long option? \*/

**if** (longopts != NULL

&& (memcmp(argv[optind], "--", **2**) == **0**

|| (only && argv[optind][**0**] == **'+'**))

&& optwhere == **1**) {

/\* handle long options \*/

**if** (memcmp(argv[optind], "--", **2**) == **0**)

optwhere = **2**;

longopt\_match = -**1**;

possible\_arg = strchr(argv[optind] + optwhere, **'='**);

**if** (possible\_arg == NULL) {

/\* no =, so next argv might be arg \*/

match\_chars = strlen(argv[optind]);

possible\_arg = argv[optind] + match\_chars;

match\_chars = match\_chars - optwhere;

}

**else**

match\_chars = (possible\_arg - argv[optind]) - optwhere;

**for** (optindex = **0**; longopts[optindex].name != NULL; optindex++) {

**if** (memcmp(argv[optind] + optwhere,

longopts[optindex].name,

match\_chars) == **0**) {

/\* do we have an exact match? \*/

**if** (match\_chars == (**int**)(strlen(longopts[optindex].name))) {

longopt\_match = optindex;

**break**;

}

/\* do any characters match? \*/

**else** {

**if** (longopt\_match < **0**)

longopt\_match = optindex;

**else** {

/\* we have ambiguous options \*/

**if** (opterr)

fprintf(stderr, "%s: option `%s' is ambiguous "

"(could be `--%s' or `--%s')\n",

argv[**0**],

argv[optind],

longopts[longopt\_match].name,

longopts[optindex].name);

**return** (optopt = **'?'**);

}

}

}

}

**if** (longopt\_match >= **0**)

has\_arg = longopts[longopt\_match].has\_arg;

}

/\* if we didn't find a long option, is it a short option? \*/

**if** (longopt\_match < **0** && shortopts != NULL) {

cp = strchr(shortopts, argv[optind][optwhere]);

**if** (cp == NULL) {

/\* couldn't find option in shortopts \*/

**if** (opterr)

fprintf(stderr,

"%s: invalid option -- `-%c'\n",

argv[**0**],

argv[optind][optwhere]);

optwhere++;

**if** (argv[optind][optwhere] == **'\0'**) {

optind++;

optwhere = **1**;

}

**return** (optopt = **'?'**);

}

has\_arg = ((cp[**1**] == **':'**)

? ((cp[**2**] == **':'**) ? OPTIONAL\_ARG : REQUIRED\_ARG)

: NO\_ARG);

possible\_arg = argv[optind] + optwhere + **1**;

optopt = \*cp;

}

/\* get argument and reset optwhere \*/

arg\_next = **0**;

**switch** (has\_arg) {

**case** OPTIONAL\_ARG:

**if** (\*possible\_arg == **'='**)

possible\_arg++;

**if** (\*possible\_arg != **'\0'**) {

optarg = possible\_arg;

optwhere = **1**;

}

**else**

optarg = NULL;

**break**;

**case** REQUIRED\_ARG:

**if** (\*possible\_arg == **'='**)

possible\_arg++;

**if** (\*possible\_arg != **'\0'**) {

optarg = possible\_arg;

optwhere = **1**;

}

**else** **if** (optind + **1** >= argc) {

**if** (opterr) {

fprintf(stderr, "%s: argument required for option `",

argv[**0**]);

**if** (longopt\_match >= **0**)

fprintf(stderr, "--%s'\n", longopts[longopt\_match].name);

**else**

fprintf(stderr, "-%c'\n", \*cp);

}

optind++;

**return** (optopt = **':'**);

}

**else** {

optarg = argv[optind + **1**];

arg\_next = **1**;

optwhere = **1**;

}

**break**;

**case** NO\_ARG:

**if** (longopt\_match < **0**) {

optwhere++;

**if** (argv[optind][optwhere] == **'\0'**)

optwhere = **1**;

}

**else**

optwhere = **1**;

optarg = NULL;

**break**;

}

/\* do we have to permute or otherwise modify optind? \*/

**if** (ordering == PERMUTE && optwhere == **1** && num\_nonopts != **0**) {

permute(argv + permute\_from, num\_nonopts, **1** + arg\_next);

optind = permute\_from + **1** + arg\_next;

}

**else** **if** (optwhere == **1**)

optind = optind + **1** + arg\_next;

/\* finally return \*/

**if** (longopt\_match >= **0**) {

**if** (longind != NULL)

\*longind = longopt\_match;

**if** (longopts[longopt\_match].flag != NULL) {

\*(longopts[longopt\_match].flag) = longopts[longopt\_match].val;

**return** **0**;

}

**else**

**return** longopts[longopt\_match].val;

}

**else**

**return** optopt;

}

**int** getopt\_long(**int** argc, **char** \*\*argv, **char** \*shortopts,

GETOPT\_LONG\_OPTION\_T \* longopts, **int** \*longind)

{

**return** getopt\_internal(argc, argv, shortopts, longopts, longind, **0**);

}

**void** help(**void**)

{

puts( "OPTIONS" );

puts( "" );

puts( "-c N, --bytes N Print last N bytes. "

"N is a nonzero integer," );

puts( " optionally followed by one of "

"the following" );

puts( " characters:" );

puts( "" );

puts( " b 512-byte blocks." );

puts( " k 1-kilobyte blocks." );

puts( " m 1-megabyte blocks." );

puts( "" );

puts( "-N, -l N, -n N, Print last N lines." );

puts( "--lines N" );

puts( "" );

puts( "-q, --quiet, Never print filename headers. "

"Normally, filename" );

puts( "--silent headers are printed if and only"

" if more than one file" );

puts( " is given on the command line." );

puts( "" );

puts( "-v, --verbose Always print filename headers." );

puts( "" );

puts( "--help Print usage message and exit successfully.");

puts( "" );

puts( "--version Print version"

" information and exit successfully." );

}

**void** version(**void**)

{

puts( "tail - output the last part of files" );

puts( "Version 1.0" );

puts( "Written by Gregory Pietsch" );

}

/\* allocate memory, die on error \*/

**void** \*xmalloc(size\_t n)

{

**void** \*p = malloc(n);

**if** (p == NULL) {

fprintf(stderr, "%s: out of memory\n", program\_name);

exit(EXIT\_FAILURE);

}

**return** p;

}

/\* reallocate memory, die on error \*/

**void** \*xrealloc(**void** \*p, size\_t n)

{

**void** \*s;

**if** (n == **0**) {

**if** (p != NULL)

free(p);

**return** NULL;

}

**if** (p == NULL)

**return** xmalloc(n);

s = realloc(p, n);

**if** (s == NULL) {

fprintf(stderr, "%s: out of memory\n", program\_name);

exit(EXIT\_FAILURE);

}

**return** s;

}

/\* get string duplicate \*/

**char** \*xstrdup(**char** \*s)

{

**char** \*p = xmalloc(strlen(s) + **1**);

strcpy(p, s);

**return** p;

}

/\* queue stuff - get fresh queue \*/

LINE\_QUEUE\_T \*lq\_create(**void**)

{

LINE\_QUEUE\_T \*lq = xmalloc(**sizeof** LINE\_QUEUE\_T);

lq->first = NULL;

lq->last = NULL;

lq->num\_elements = **0**;

**return** lq;

}

/\* put an item onto the queue \*/

**void** lq\_enq(LINE\_QUEUE\_T \* lq, **char** \*s)

{

LINE\_QUEUE\_EL\_T \*lq\_el = xmalloc(**sizeof** LINE\_QUEUE\_EL\_T);

lq\_el->s = xstrdup(s);

lq\_el->next = NULL;

**if** (lq->first == NULL && lq->last == NULL) {

/\* first element \*/

lq->first = lq->last = lq\_el;

lq->num\_elements = **1**;

}

**else** {

/\* tack onto end \*/

lq->last->next = lq\_el;

lq->last = lq\_el;

lq->num\_elements++;

}

}

/\* take an item off the queue \*/

**char** \*lq\_deq(LINE\_QUEUE\_T \* lq)

{

**char** \*s;

LINE\_QUEUE\_EL\_T \*lq\_el;

**if** (lq->first == NULL)

**return** NULL;

lq\_el = lq->first;

s = lq\_el->s;

**if** (lq->first == lq->last)

lq->first = lq->last = NULL;

**else**

lq->first = lq->first->next;

free(lq\_el);

lq->num\_elements--;

**return** s;

}

/\* output number lines -- this function is tough because I can only

\* use fseek() to rewind a text stream (See ISO C 7.9.9.2 if you don't

\* believe me).

\*/

**void** tail\_lines(FILE \* f)

{

**char** buffer[TAIL\_BUFFER\_SIZE];

size\_t num\_read;

**int** last\_is\_nl = **0**;

**unsigned** **long** num\_skipped = **0**;

**int** c;

LINE\_QUEUE\_T \*lq = NULL;

**char** \*s;

size\_t s\_size = **0**;

size\_t s\_allocked = **0**;

**char** \*p;

**if** (flag\_skip) {

/\* skip a bunch of lines, output everything else \*/

**while** ((c = getc(f)) != EOF && num\_skipped < number) {

**if** (c == **'\n'**)

num\_skipped++;

}

**while** ((num\_read = fread(buffer, **1**, TAIL\_BUFFER\_SIZE, f)) != **0**)

{

fwrite(buffer, **1**, num\_read, stdout);

last\_is\_nl = (buffer[num\_read - **1**] == **'\n'**);

}

**if** (!last\_is\_nl)

fputc(**'\n'**, stdout);

}

**else** {

lq = lq\_create();

s = xmalloc(TAIL\_STRING\_BUFFER\_SIZE);

s\_allocked = TAIL\_STRING\_BUFFER\_SIZE;

**while** ((c = getc(f)) != EOF) {

/\* add to s, if not at eof or end of line \*/

**if** (c != **'\n'**) {

**if** (s\_size == s\_allocked - **1**) {

s\_allocked += TAIL\_STRING\_BUFFER\_SIZE;

s = xrealloc(s, s\_allocked);

}

s[s\_size++] = c;

}

**else** {

/\* enqueue s, possibly dequeueing if we don't need a

line \*/

s[s\_size] = **'\0'**;

lq\_enq(lq, s);

**if** (lq->num\_elements > number)

free(lq\_deq(lq));

s\_size = **0**;

}

}

**while** (lq->num\_elements != **0**) {

/\* print out strings \*/

p = lq\_deq(lq);

puts(p);

free(p);

}

free(s);

free(lq);

}

}

/\* output number characters, or skip over number characters \*/

**void** tail\_chars(FILE \* f)

{

**char** buffer[TAIL\_BUFFER\_SIZE];

size\_t num\_read;

**int** last\_is\_nl = **0**;

**long** lnum = number;

**if** (flag\_skip)

fseek(f, lnum, SEEK\_SET);

**else**

fseek(f, -lnum, SEEK\_END);

**while** ((num\_read = fread(buffer, **1**, TAIL\_BUFFER\_SIZE, f)) != **0**) {

fwrite(buffer, **1**, num\_read, stdout);

last\_is\_nl = (buffer[num\_read - **1**] == **'\n'**);

}

**if** (!last\_is\_nl)

fputc(**'\n'**, stdout);

}

**void** parse\_args(**int** argc, **char** \*\*argv)

{

**int** opt;

**char** \*p;

**int** flag\_found\_number = **0**;

**int** verbosity\_changed = **0**;

**do** {

**switch** ((opt = getopt\_long(argc, argv, shortopts, longopts, NULL))) {

**case** **'c'**: /\* print bytes \*/

**if** (flag\_found\_number) {

fprintf(stderr, "%s: invalid arguments\s", program\_name);

abort();

}

flag\_bytes = **1**;

p = optarg;

**if** (\*p == **'+'**) {

flag\_skip = **1**;

p++;

}

**for** (number = **0**;

isdigit(\*p);

number = number \* **10** + (\*p++ - **'0'**));

**switch** (\*p) {

**case** **'b'**: /\* 512-byte blocks \*/

number \*= **512**;

**break**;

**case** **'k'**: /\* kilobyte blocks \*/

number \*= **1024**;

**break**;

**case** **'m'**: /\* megabyte blocks \*/

number \*= **1048576**;

**break**;

**default**:

**break**;

}

flag\_found\_number = **1**;

**break**;

**case** **'l'**:

**case** **'n'**: /\* lines \*/

**if** (flag\_found\_number) {

fprintf(stderr, "%s: invalid arguments\s", program\_name);

abort();

}

flag\_bytes = **0**;

p = optarg;

**if** (\*p == **'+'**) {

flag\_skip = **1**;

p++;

}

number = strtoul(p, NULL, **10**);

flag\_found\_number = **1**;

**break**;

**case** **'q'**: /\* quiet \*/

**if** (verbosity\_changed) {

fprintf(stderr, "%s: invalid arguments\s", program\_name);

abort();

}

verbosity\_changed = **1**;

flag\_verbosity = NEVER;

**break**;

**case** **'v'**: /\* verbose \*/

**if** (verbosity\_changed) {

fprintf(stderr, "%s: invalid arguments\s", program\_name);

abort();

}

verbosity\_changed = **1**;

flag\_verbosity = ALWAYS;

**break**;

**case** **'?'**: /\* invalid option \*/

fprintf(stderr, "For help, type:\n\t%s --help\n", program\_name);

exit(EXIT\_FAILURE);

**case** **1**:

**case** **0**:

**if** (show\_help || show\_version) {

**if** (show\_help)

help();

**if** (show\_version)

version();

exit(EXIT\_SUCCESS);

}

**break**;

**default**:

**break**;

}

} **while** (opt != EOF);

**if** (flag\_found\_number == **0** || number == **0**) {

/\* didn't find anything, so set default \*/

flag\_bytes = **0**;

number = **10**;

}

}

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

{

**int** i;

**int** j;

**unsigned** **long** ul;

**char** \*\*new\_argv = xmalloc((argc + **1**) \* (**sizeof**(**char** \*)));

**char** \*allocked\_argvs = xmalloc(argc + **1**);

**char** \*p;

**char** \*s;

**char** \*t;

FILE \*f;

**int** flag\_plus = **0**;

memset(allocked\_argvs, **0**, argc + **1**);

new\_argv[**0**] = program\_name = argv[**0**];

/\* deal with silly old-format arguments \*/

**for** (i = **1**, j = **1**; i < argc; i++) {

p = argv[i];

flag\_plus = **0**;

/\* handle options first \*/

**if** (\*p == **'-'** || \*p == **'+'**) {

**if** (isdigit(p[**1**]) || p[**1**] == **'+'** || \*p == **'+'**) {

/\* rearrange p \*/

s = xmalloc(strlen(p) + **3**);

t = s;

\*t++ = **'-'**;

**if** (\*p == **'-'**)

p++;

ul = **0**;

**if** (\*p == **'+'**) {

flag\_plus = **1**;

p++;

}

**while** (isdigit(\*p)) {

ul = ul \* **10** + (\*p - **'0'**);

p++;

}

**if** (strchr(p, **'q'**) != NULL)

\*t++ = **'q'**;

**if** (strchr(p, **'v'**) != NULL)

\*t++ = **'v'**;

**if** (strpbrk(p, "cbkm") != NULL)

\*t++ = **'c'**;

**if** (strchr(p, **'l'**) != NULL)

\*t++ = **'l'**;

**if** (strchr(p, **'n'**) != NULL || t[-**1**] == **'-'**)

\*t++ = **'n'**;

**if** (flag\_plus)

\*t++ = **'+'**;

sprintf(t, "%lu", ul);

t += strlen(t);

**if** (strchr(p, **'b'**) != NULL)

\*t++ = **'b'**;

**if** (strchr(p, **'k'**) != NULL)

\*t++ = **'k'**;

**if** (strchr(p, **'m'**) != NULL)

\*t++ = **'m'**;

\*t = **'\0'**;

new\_argv[j] = s;

allocked\_argvs[j++] = **1**;

}

**else**

new\_argv[j++] = argv[i];

}

}

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

/\* handle file names \*/

p = argv[i];

**if** (\*p != **'-'**)

new\_argv[j++] = p;

}

new\_argv[argc] = NULL;

parse\_args(argc, new\_argv);

**if** (optind == argc

|| (optind == argc - **1** && strcmp(argv[optind], "-") == **0**)) {

/\* no more argv-elements, tail stdin \*/

**if** (flag\_verbosity == ALWAYS)

puts("==> standard input <==");

flag\_bytes ? tail\_chars(stdin) : tail\_lines(stdin);

}

**else** **if** (optind == argc - **1**) {

/\* one file \*/

f = fopen(new\_argv[optind], flag\_bytes ? "rb" : "r");

**if** (f == NULL) {

fprintf(stderr, "%s: Can't open file %s\n",

program\_name, new\_argv[optind]);

abort();

}

**if** (flag\_verbosity == ALWAYS)

printf("==> %s <==\n", new\_argv[optind]);

flag\_bytes ? tail\_chars(f) : tail\_lines(f);

fclose(f);

}

**else** {

/\* multiple files \*/

**for** (i = optind; i < argc; i++) {

**if** (strcmp(new\_argv[i], "-") == **0**) {

f = stdin;

**if** (flag\_verbosity != NEVER)

puts("==> standard input <==");

}

**else** {

f = fopen(new\_argv[i], flag\_bytes ? "rb" : "r");

**if** (f == NULL) {

fprintf(stderr, "%s: can't open %s\n",

argv[**0**], argv[i]);

abort();

}

**if** (flag\_verbosity != NEVER)

printf("==> %s <==\n", new\_argv[i]);

}

flag\_bytes ? tail\_chars(f) : tail\_lines(f);

**if** (f != stdin)

fclose(f);

}

}

/\* free all we can \*/

**for** (i = **1**; i <= argc; i++)

**if** (allocked\_argvs[i])

free(new\_argv[i]);

free(allocked\_argvs);

**return** EXIT\_SUCCESS;

}

/\* END OF FILE tail.c \*/

Steven Huang's solution

/\* K&R Exercise 5-13 \*/

/\* Steven Huang \*/

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <string.h>**

**#define DEFAULT\_NUM\_LINES 10**

**#define MAX\_LINE\_LEN 1000**

/\*

Points of interest for a novice:

1. atoi() has a normally annoying property of not being able to

tell the caller conclusively whether the input was bad ("abc")

or it was really zero ("0"), because it returns 0 for both

cases. Here, we exploit that property, because we only want

to accept options in the form of "-n".

2. Try to understand how this program deals with input that

doesn't even have as many lines as the line\_ptrs[] array.

That is, how does this program degenerate into just displaying

everything it read? (Hint: what does it mean when line\_ptrs[x]

is NULL?)

3. Using modulo arithmetic on an index to a circular array is

a common and useful technique. Try to understand the range

of values that current\_line (and j, later) will take. In

particular, why shouldn't we just do this:

for (i = 0; i < num\_lines; i++)

if (line\_ptrs[i])

printf("%s", line\_ptrs[i]);

4. Why do we still use a "%s" to display what's inside line\_ptrs,

rather than just:

printf(line\_ptrs[i]);

5. There is a bug in this program, where you see:

numlines = -numlines;

When will this break?

\*/

/\* K&R2 p29 \*/

**int** getline(**char** s[], **int** lim)

{

**int** c, i;

**for** (i = **0**; i < lim - **1** && (c = getchar()) != EOF && c != **'\n'**; i++)

s[i] = c;

**if** (c == **'\n'**)

s[i++] = c;

s[i] = **'\0'**;

**return** i;

}

/\* duplicates a string \*/

**char** \*dupstr(**const** **char** \*s)

{

**char** \*p = malloc(strlen(s) + **1**);

**if** (p)

strcpy(p, s);

**return** p;

}

**int** main(**int** argc, **char** \*argv[])

{

**int** num\_lines = DEFAULT\_NUM\_LINES;

**char** \*\*line\_ptrs;

**char** buffer[MAX\_LINE\_LEN];

**int** i;

**unsigned** j, current\_line;

**if** (argc > **1**) {

/\*

We use a little trick here. The command line parameter should be

in the form of "-n", where n is the number of lines. We don't

check for the "-", but just pass it to atoi() anyway, and then

check if atoi() returned us a negative number.

\*/

num\_lines = atoi(argv[**1**]);

**if** (num\_lines >= **0**) {

fprintf(stderr, "Expected -n, where n is the number of lines\n");

**return** EXIT\_FAILURE;

}

/\* Now make num\_lines the positive number it's supposed to be. \*/

num\_lines = -num\_lines;

}

/\* First, let's get enough storage for a list of n pointers... \*/

line\_ptrs = malloc(**sizeof** \*line\_ptrs \* num\_lines);

**if** (!line\_ptrs) {

fprintf(stderr, "Out of memory. Sorry.\n");

**return** EXIT\_FAILURE;

}

/\* and make them all point to NULL \*/

**for** (i = **0**; i < num\_lines; i++)

line\_ptrs[i] = NULL;

/\* Now start reading \*/

current\_line = **0**;

**do** {

getline(buffer, **sizeof** buffer);

**if** (!feof(stdin)) {

**if** (line\_ptrs[current\_line]) {

/\* there's already something here \*/

free(line\_ptrs[current\_line]);

}

line\_ptrs[current\_line] = dupstr(buffer);

**if** (!line\_ptrs[current\_line]) {

fprintf(stderr, "Out of memory. Sorry.\n");

**return** EXIT\_FAILURE;

}

current\_line = (current\_line + **1**) % num\_lines;

}

} **while** (!feof(stdin));

/\* Finished reading the file, so we are ready to print the lines \*/

**for** (i = **0**; i < num\_lines; i++) {

j = (current\_line + i) % num\_lines;

**if** (line\_ptrs[j]) {

printf("%s", line\_ptrs[j]);

free(line\_ptrs[j]);

}

}

**return** EXIT\_SUCCESS;

}

**Answer to Exercise 5-14, page 121**

*Modify the sort program to handle a -r flag, which indicates sorting in reverse (decreasing) order. Be sure that -r works with -n.*

/\* K&R Exercise 5-14 \*/

/\* Steven Huang \*/

**#include <stdio.h>**

**#include <string.h>**

**#include <stdlib.h>**

**#define TRUE 1**

**#define FALSE 0**

**#define MAXLINES 5000 /\* maximum number of lines \*/**

**char** \*lineptr[MAXLINES];

**#define MAXLEN 1000 /\* maximum length of a line \*/**

**int** reverse = FALSE;

/\* K&R2 p29 \*/

**int** getline(**char** s[], **int** lim)

{

**int** c, i;

**for** (i = **0**; i < lim - **1** && (c = getchar()) != EOF && c != **'\n'**; i++)

s[i] = c;

**if** (c == **'\n'**) {

s[i++] = c;

}

s[i] = **'\0'**;

**return** i;

}

/\* K&R2 p109 \*/

**int** readlines(**char** \*lineptr[], **int** maxlines)

{

**int** len, nlines;

**char** \*p, line[MAXLEN];

nlines = **0**;

**while** ((len = getline(line, MAXLEN)) > **0**)

**if** (nlines >= maxlines || (p = malloc(len)) == NULL)

**return** -**1**;

**else** {

line[len - **1**] = **'\0'**; /\* delete the newline \*/

strcpy(p, line);

lineptr[nlines++] = p;

}

**return** nlines;

}

/\* K&R2 p109 \*/

**void** writelines(**char** \*lineptr[], **int** nlines)

{

**int** i;

**for** (i = **0**; i < nlines; i++)

printf("%s\n", lineptr[i]);

}

**int** pstrcmp(**const** **void** \*p1, **const** **void** \*p2)

{

**char** \* **const** \*s1 = reverse ? p2 : p1;

**char** \* **const** \*s2 = reverse ? p1 : p2;

**return** strcmp(\*s1, \*s2);

}

**int** numcmp(**const** **void** \*p1, **const** **void** \*p2)

{

**char** \* **const** \*s1 = reverse ? p2 : p1;

**char** \* **const** \*s2 = reverse ? p1 : p2;

**double** v1, v2;

v1 = atof(\*s1);

v2 = atof(\*s2);

**if** (v1 < v2)

**return** -**1**;

**else** **if** (v1 > v2)

**return** **1**;

**else**

**return** **0**;

}

**int** main(**int** argc, **char** \*argv[])

{

**int** nlines;

**int** numeric = FALSE;

**int** i;

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

**if** (\*argv[i] == **'-'**) {

**switch** (\*(argv[i] + **1**)) {

**case** **'n'**: numeric = TRUE; **break**;

**case** **'r'**: reverse = TRUE; **break**;

**default**:

fprintf(stderr, "invalid switch '%s'\n", argv[i]);

**return** EXIT\_FAILURE;

}

}

}

**if** ((nlines = readlines(lineptr, MAXLINES)) >= **0**) {

qsort(lineptr, nlines, **sizeof**(\*lineptr), numeric ? numcmp : pstrcmp);

writelines(lineptr, nlines);

**return** EXIT\_SUCCESS;

} **else** {

fputs("input too big to sort\n", stderr);

**return** EXIT\_FAILURE;

}

}

**Answer to Exercise 6-1, page 136**

*Our version of getword does not properly handle underscores, string constants, comments, or preprocessor control lines. Write a better version.*

/\* K&R 6-1: "Our version of getword() does not properly handle

underscores, string constants, or preprocessor control lines.

Write a better version."

This is intended to be a solution to K&R 6-1 in "category 0" as

defined by the official rules given on Richard Heathfield's "The C

Programming Language Answers To Exercises" page, found at

http://users.powernet.co.uk/eton/kandr2/index.html.

For more information on the language for which this is a lexical

analyzer, please see the comment preceding getword() below.

Note that there is a small modification to ungetch() as defined by

K&R. Hopefully this lies within the rules. \*/

/\* knr61.c - answer to K&R2 exercise 6-1.

Copyright (C) 2000 Ben Pfaff <blp@gnu.org>.

This program is free software; you can redistribute it and/or

modify it under the terms of the GNU General Public License as

published by the Free Software Foundation; either version 2 of the

License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but

WITHOUT ANY WARRANTY; without even the implied warranty of

MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU

General Public License for more details.

You should have received a copy of the GNU General Public License

along with this program; if not, write to the Free Software

Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA

02111-1307, USA. \*/

**#include <ctype.h>**

**#include <limits.h>**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <string.h>**

/\* Tokens. Other non-whitespace characters self-represent themselves

as tokens. \*/

**enum** token

{

TOK\_ID = UCHAR\_MAX + **1**, /\* Identifier. \*/

TOK\_STRING, /\* String constant. \*/

TOK\_CHAR, /\* Character constant. \*/

TOK\_EOF /\* End of file. \*/

};

**enum** token getword (**char** \*word, **int** lim);

**static** **int** skipws (**void**);

**static** **int** getstelem (**char** \*\*, **int** \*, **int**);

**static** **int** getch (**void**);

**static** **void** ungetch (**int**);

**static** **void** putch (**char** \*\*, **int** \*, **int**);

/\* Main program for testing. \*/

**int**

main (**void**)

{

ungetch (**'\n'**);

**for** (;;)

{

**char** word[**64**];

**enum** token token;

/\* Get token. \*/

token = getword (word, **sizeof** word);

/\* Print token type. \*/

**switch** (token)

{

**case** TOK\_ID:

printf ("id");

**break**;

**case** TOK\_STRING:

printf ("string");

**break**;

**case** TOK\_CHAR:

printf ("char");

**break**;

**case** TOK\_EOF:

printf ("eof\n");

**return** **0**;

**default**:

printf ("other");

word[**0**] = token;

word[**1**] = **'\0'**;

**break**;

}

/\* Print token value more or less unambiguously. \*/

{

**const** **char** \*s;

printf ("\t'");

**for** (s = word; \*s != **'\0'**; s++)

**if** (isprint (\*s) && \*s != **'\''**)

putchar (\*s);

**else** **if** (\*s == **'\''**)

printf ("\\'");

**else**

/\* Potentially wrong. \*/

printf ("\\x%02x", \*s);

printf ("'\n");

}

}

}

/\* Parses C-like tokens from stdin:

- Parses C identifiers and string and character constants.

- Other characters, such as operators, punctuation, and digits

not part of identifiers are considered as tokens in

themselves.

- Skip comments and preprocessor control lines.

Does not handle trigraphs, line continuation with \, or numerous

other special C features.

Returns a token type. This is either one of TOK\_\* above, or a single

character in the range 0...UCHAR\_MAX.

If TOK\_ID, TOK\_STRING, or TOK\_CHAR is returned, WORD[] is filled

with the identifier or string value, truncated at LIM - 1

characters and terminated with '\0'.

For other returned token types, WORD[] is indeterminate. \*/

**enum** token

getword (**char** \*word, **int** lim)

{

**int** beg\_line, c;

**for** (;;)

{

beg\_line = skipws ();

c = getch ();

**if** (!beg\_line || c != **'#'**)

**break**;

/\* Skip preprocessor directive. \*/

**do**

{

c = getch ();

**if** (c == EOF)

**return** TOK\_EOF;

}

**while** (c != **'\n'**);

ungetch (**'\n'**);

}

**if** (c == EOF)

**return** TOK\_EOF;

**else** **if** (c == **'\_'** || isalpha ((**unsigned** **char**) c))

{

**do**

{

putch (&word, &lim, c);

c = getch ();

}

**while** (isalnum ((**unsigned** **char**) c) || c == **'\_'**);

ungetch (c);

**return** TOK\_ID;

}

**else** **if** (c == **'\''** || c == **'"'**)

{

**int** quote = c;

word[**0**] = **'\0'**;

**while** (getstelem (&word, &lim, quote))

;

**return** quote == **'\''** ? TOK\_CHAR : TOK\_STRING;

}

**else**

**return** (**unsigned** **char**) c;

}

/\* Skips whitespace and comments read from stdin.

Returns nonzero if a newline was encountered, indicating that we're

at the beginning of a line. \*/

**static** **int**

skipws (**void**)

{

/\* Classification of an input character. \*/

**enum** class

{

CLS\_WS = **0**, /\* Whitespace. \*/

CLS\_BEG\_CMT, /\* Slash-star beginning a comment. \*/

CLS\_END\_CMT, /\* Star-slash ending a comment. \*/

CLS\_OTHER, /\* None of the above. \*/

CLS\_IN\_CMT = **4** /\* Combined with one of the above,

indicates we're inside a comment. \*/

};

/\* Either 0, if we're not inside a comment,

or CLS\_IN\_CMT, if we are inside a comment. \*/

**enum** class in\_comment = **0**;

/\* Have we encountered a newline outside a comment? \*/

**int** beg\_line = **0**;

**for** (;;)

{

**int** c; /\* Input character. \*/

**enum** class class; /\* Classification of `c'. \*/

/\* Get an input character and determine its classification. \*/

c = getch ();

**switch** (c)

{

**case** **'\n'**:

**if** (!in\_comment)

beg\_line = **1**;

/\* Fall through. \*/

**case** **' '**: **case** **'\f'**: **case** **'\r'**: **case** **'\t'**: **case** **'\v'**:

class = CLS\_WS;

**break**;

**case** **'/'**:

/\* Outside a comment, slash-star begins a comment. \*/

**if** (!in\_comment)

{

c = getch ();

**if** (c == **'\*'**)

class = CLS\_BEG\_CMT;

**else**

{

ungetch (c);

c = **'/'**;

class = CLS\_OTHER;

}

}

**else**

class = CLS\_OTHER;

**break**;

**case** **'\*'**:

/\* Inside a comment, star-slash ends the comment. \*/

**if** (in\_comment)

{

c = getch ();

**if** (c == **'/'**)

class = CLS\_END\_CMT;

**else**

{

ungetch (c);

class = CLS\_OTHER;

}

}

**else**

class = CLS\_OTHER;

**break**;

**default**:

/\* Other characters. \*/

**if** (c == EOF)

**return** **0**;

class = CLS\_OTHER;

}

/\* Handle character `c' according to its classification

and whether we're inside a comment. \*/

**switch** (class | in\_comment)

{

**case** CLS\_WS:

**case** CLS\_WS | CLS\_IN\_CMT:

**case** CLS\_OTHER | CLS\_IN\_CMT:

**break**;

**case** CLS\_BEG\_CMT:

in\_comment = CLS\_IN\_CMT;

**break**;

**case** CLS\_OTHER:

ungetch (c);

**return** beg\_line;

**case** CLS\_END\_CMT | CLS\_IN\_CMT:

in\_comment = **0**;

**break**;

**case** CLS\_BEG\_CMT | CLS\_IN\_CMT:

**case** CLS\_END\_CMT:

**default**:

printf ("can't happen\n");

**break**;

}

}

}

/\* Get a character inside a quoted string or character constant.

QUOTE is ' for a character constant or " for a quoted string.

\*WORDP points to a string being constructed that has \*LIMP bytes

available. \*/

**static** **int**

getstelem (**char** \*\*wordp, **int** \*limp, **int** quote)

{

**int** c;

/\* Handle end-of-quote and EOF. \*/

c = getch ();

**if** (c == quote || c == EOF)

**return** **0**;

/\* Handle ordinary string characters. \*/

**if** (c != **'\\'**)

{

putch (wordp, limp, c);

**return** **1**;

}

/\* We're in a \ escape sequence.

Get the second character. \*/

c = getch ();

**if** (c == EOF)

**return** **0**;

/\* Handle simple single-character escapes. \*/

{

**static** **const** **char** escapes[] = {"''??\"\"\\\\a\ab\bf\fn\nr\rt\tv\v"};

**const** **char** \*cp = strchr (escapes, c);

**if** (cp != NULL)

{

putch (wordp, limp, cp[**1**]);

**return** **1**;

}

}

/\* Handle hexadecimal and octal escapes.

This also handles invalid escapes by default,

doing nothing useful with them.

That's okay because invalid escapes generate undefined behavior. \*/

{

**unsigned** **char** v = **0**;

**if** (c == **'x'** || c == **'X'**)

**for** (;;)

{

**static** **const** **char** hexits[] = "0123456789abcdef";

**const** **char** \*p;

c = getch ();

p = strchr (hexits, tolower ((**unsigned** **char**) c));

**if** (p == NULL)

**break**;

v = v \* **16** + (p - hexits);

}

**else**

{

**int** i;

**for** (i = **0**; i < **3**; i++)

{

v = v \* **8** + (c - **'0'**);

c = getch ();

**if** (c < **'0'** || c > **'7'**)

**break**;

}

}

putch (wordp, limp, v);

ungetch (c);

}

**return** **1**;

}

/\* Capacity of putback buffer. \*/

**#define BUFSIZE 100**

/\* Putback buffer. \*/

**char** buf[BUFSIZE];

/\* Number of characters in putback buffer. \*/

**int** bufp = **0**;

/\* Retrieves and returns a character from stdin or from the putback

buffer.

Returns EOF if end of file is encountered. \*/

**int**

getch (**void**)

{

**return** bufp > **0** ? buf[--bufp] : getchar ();

}

/\* Stuffs character C into the putback buffer.

From the caller's perspective, fails silently if the putback buffer

is full. \*/

**void**

ungetch (**int** c)

{

**if** (c == EOF)

**return**;

**if** (bufp >= BUFSIZE)

printf ("ungetch: too many characters\n");

**else**

buf[bufp++] = c;

}

/\* Stuffs character C into buffer \*WORDP, which has \*LIMP bytes

available.

Advances \*WORDP and reduces \*LIMP as appropriate.

Drops the character on the floor if it would overflow the buffer.

Ensures that \*WORDP is null terminated if possible. \*/

**static** **void**

putch (**char** \*\*wordp, **int** \*limp, **int** c)

{

**if** (\*limp > **1**)

{

\*(\*wordp)++ = c;

(\*limp)--;

}

**if** (\*limp > **0**)

\*\*wordp = **'\0'**;

}

/\*

Local variables:

compile-command: "checkergcc -W -Wall -ansi -pedantic knr61.c -o knr61"

End:

\*/

**Answer to Exercise 6-3, page 143**

Bug (noticed by John W Krahn) fixed 11 June 2002. The noise word list was broken because it contained out-of-order data. I fixed this, and made the program more generally useful, by performing all string comparisons without regard to case.   
  
*Write a cross-referencer that prints a list of all words in a document, and, for each word, a list of the line numbers on which it occurs. Remove noise words like "the", "and," and so on.*

/\* Write a cross-referencer program that prints a list of all words in a

\* document, and, for each word, a list of the line numbers on which it

\* occurs. Remove noise words like "the", "and," and so on.

\*/

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <string.h>**

**#include <ctype.h>**

/\* no such thing as strdup, so let's write one

\*

\* supplementary question: why did I call this function dupstr,

\* rather than strdup?

\*

\*/

**char** \*dupstr(**char** \*s)

{

**char** \*p = NULL;

**if**(s != NULL)

{

p = malloc(strlen(s) + **1**);

**if**(p)

{

strcpy(p, s);

}

}

**return** p;

}

/\* case-insensitive string comparison \*/

**int** i\_strcmp(**const** **char** \*s, **const** **char** \*t)

{

**int** diff = **0**;

**char** cs = **0**;

**char** ct = **0**;

**while**(diff == **0** && \*s != **'\0'** && \*t != **'\0'**)

{

cs = tolower((**unsigned** **char**)\*s);

ct = tolower((**unsigned** **char**)\*t);

**if**(cs < ct)

{

diff = -**1**;

}

**else** **if**(cs > ct)

{

diff = **1**;

}

++s;

++t;

}

**if**(diff == **0** && \*s != \*t)

{

/\* the shorter string comes lexicographically sooner \*/

**if**(\*s == **'\0'**)

{

diff = -**1**;

}

**else**

{

diff = **1**;

}

}

**return** diff;

}

**struct** linelist

{

**struct** linelist \*next;

**int** line;

};

**struct** wordtree

{

**char** \*word;

**struct** linelist \*firstline;

**struct** wordtree \*left;

**struct** wordtree \*right;

};

**void** printlist(**struct** linelist \*list)

{

**if**(list != NULL)

{

printlist(list->next);

printf("%6d ", list->line);

}

}

**void** printtree(**struct** wordtree \*node)

{

**if**(node != NULL)

{

printtree(node->left);

printf("%18s ", node->word);

printlist(node->firstline);

printf("\n");

printtree(node->right);

}

}

**struct** linelist \*addlink(**int** line)

{

**struct** linelist \*new = malloc(**sizeof** \*new);

**if**(new != NULL)

{

new->line = line;

new->next = NULL;

}

**return** new;

}

**void** deletelist(**struct** linelist \*listnode)

{

**if**(listnode != NULL)

{

deletelist(listnode->next);

free(listnode);

}

}

**void** deleteword(**struct** wordtree \*\*node)

{

**struct** wordtree \*temp = NULL;

**if**(node != NULL)

{

**if**(\*node != **'\0'**)

{

**if**((\*node)->right != NULL)

{

temp = \*node;

deleteword(&temp->right);

}

**if**((\*node)->left != NULL)

{

temp = \*node;

deleteword(&temp->left);

}

**if**((\*node)->word != NULL)

{

free((\*node)->word);

}

**if**((\*node)->firstline != NULL)

{

deletelist((\*node)->firstline);

}

free(\*node);

\*node = NULL;

}

}

}

**struct** wordtree \*addword(**struct** wordtree \*\*node, **char** \*word, **int** line)

{

**struct** wordtree \*wordloc = NULL;

**struct** linelist \*newline = NULL;

**struct** wordtree \*temp = NULL;

**int** diff = **0**;

**if**(node != NULL && word != NULL)

{

**if**(NULL == \*node)

{

\*node = malloc(**sizeof** \*\*node);

**if**(NULL != \*node)

{

(\*node)->left = NULL;

(\*node)->right = NULL;

(\*node)->word = dupstr(word);

**if**((\*node)->word != NULL)

{

(\*node)->firstline = addlink(line);

**if**((\*node)->firstline != NULL)

{

wordloc = \*node;

}

}

}

}

**else**

{

diff = i\_strcmp((\*node)->word, word);

**if**(**0** == diff)

{

/\* we have seen this word before! add this line number to

\* the front of the line number list. Adding to the end

\* would keep them in the right order, but would take

\* longer. By continually adding them to the front, we

\* take less time, but we pay for it at the end by having

\* to go to the end of the list and working backwards.

\* Recursion makes this less painful than it might have been.

\*/

newline = addlink(line);

**if**(newline != NULL)

{

wordloc = \*node;

newline->next = (\*node)->firstline;

(\*node)->firstline = newline;

}

}

**else** **if**(**0** < diff)

{

temp = \*node;

wordloc = addword(&temp->left, word, line);

}

**else**

{

temp = \*node;

wordloc = addword(&temp->right, word, line);

}

}

}

**if**(wordloc == NULL)

{

deleteword(node);

}

**return** wordloc;

}

/\* We can't use strchr because it's not yet been discussed, so we'll

\* write our own instead.

\*/

**char** \*char\_in\_string(**char** \*s, **int** c)

{

**char** \*p = NULL;

/\* if there's no data, we'll stop \*/

**if**(s != NULL)

{

**if**(c != **'\0'**)

{

**while**(\*s != **'\0'** && \*s != c)

{

++s;

}

**if**(\*s == c)

{

p = s;

}

}

}

**return** p;

}

/\* We can't use strtok because it hasn't been discussed in the text

\* yet, so we'll write our own.

\* To minimise hassle at the user end, let's modify the user's pointer

\* to s, so that we can just call this thing in a simple loop.

\*/

**char** \*tokenise(**char** \*\*s, **char** \*delims)

{

**char** \*p = NULL;

**char** \*q = NULL;

**if**(s != NULL && \*s != **'\0'** && delims != NULL)

{

/\* pass over leading delimiters \*/

**while**(NULL != char\_in\_string(delims, \*\*s))

{

++\*s;

}

**if**(\*\*s != **'\0'**)

{

q = \*s + **1**;

p = \*s;

**while**(\*q != **'\0'** && NULL == char\_in\_string(delims, \*q))

{

++q;

}

\*s = q + (\*q != **'\0'**);

\*q = **'\0'**;

}

}

**return** p;

}

/\* return zero if this word is not a noise word,

\* or non-zero if it is a noise word

\*/

**int** NoiseWord(**char** \*s)

{

**int** found = **0**;

**int** giveup = **0**;

**char** \*list[] =

{

"a",

"an",

"and",

"be",

"but",

"by",

"he",

"I",

"is",

"it",

"off",

"on",

"she",

"so",

"the",

"they",

"you"

};

**int** top = **sizeof** list / **sizeof** list[**0**] - **1**;

**int** bottom = **0**;

**int** guess = top / **2**;

**int** diff = **0**;

**if**(s != NULL)

{

**while**(!found && !giveup)

{

diff = i\_strcmp(list[guess], s);

**if**(**0** == diff)

{

found = **1**;

}

**else** **if**(**0** < diff)

{

top = guess - **1**;

}

**else**

{

bottom = guess + **1**;

}

**if**(top < bottom)

{

giveup = **1**;

}

**else**

{

guess = (top + bottom) / **2**;

}

}

}

**return** found;

}

/\*

\* Argh! We can't use fgets()! It's not discussed until page 164.

\* Oh well... time to roll our own again...

\*/

**char** \*GetLine(**char** \*s, **int** n, FILE \*fp)

{

**int** c = **0**;

**int** done = **0**;

**char** \*p = s;

**while**(!done && --n > **0** && (c = getc(fp)) != EOF)

{

**if**((\*p++ = c) == **'\n'**)

{

done = **1**;

}

}

\*p = **'\0'**;

**if**(EOF == c && p == s)

{

p = NULL;

}

**else**

{

p = s;

}

**return** p;

}

/\*

\* Ideally, we'd use a clever GetLine function which expanded its

\* buffer dynamically to cope with large lines. Since we can't use

\* realloc, and because other solutions would require quite hefty

\* engineering, we'll adopt a simple solution - a big buffer.

\*

\* Note: making the buffer static will help matters on some

\* primitive systems which don't reserve much storage for

\* automatic variables, and shouldn't break anything anywhere.

\*

\*/

**#define MAXLINE 8192**

**int** main(**void**)

{

**static** **char** buffer[MAXLINE] = {**0**};

**char** \*s = NULL;

**char** \*word = NULL;

**int** line = **0**;

**int** giveup = **0**;

**struct** wordtree \*tree = NULL;

**char** \*delims = " \t\n\r\a\f\v!\"%^&\*()\_=+{}[]\\|/,.<>:;#~?";

**while**(!giveup && GetLine(buffer, **sizeof** buffer, stdin) != NULL)

{

++line;

s = buffer;

**while**(!giveup && (word = tokenise(&s, delims)) != NULL)

{

**if**(!NoiseWord(word))

{

**if**(NULL == addword(&tree, word, line))

{

printf("Error adding data into memory. Giving up.\n");

giveup = **1**;

}

}

}

}

**if**(!giveup)

{

printf("%18s Line Numbers\n", "Word");

printtree(tree);

}

deleteword(&tree);

**return** **0**;

}

**Answer to Exercise 6-4, page 143**

*Write a program that prints the distinct words in its input sorted into decreasing order of frequency of occurrence. Precede each word by its count.*   
  
Bryan's solution is, as far as I can tell, Category 1 only because he uses EXIT\_SUCCESS and EXIT\_FAILURE .

/\*

Chapter 6. Structures

Write a program that prints out the distinct words in its

input sorted into decreasing order of frequency of occurrence.

Precede each word by its count.

Author: Bryan Williams

\*/

**#include <stdlib.h>**

**#include <stdio.h>**

**#include <string.h>**

**#include <assert.h>**

**typedef** **struct** WORD

{

**char** \*Word;

size\_t Count;

**struct** WORD \*Left;

**struct** WORD \*Right;

} WORD;

/\*

Assumptions: input is on stdin, output to stdout.

Plan: read the words into a tree, keeping a count of how many we have,

allocate an array big enough to hold Treecount (WORD \*)'s

walk the tree to populate the array.

qsort the array, based on size.

printf the array

free the array

free the tree

free tibet (optional)

free international shipping!

\*/

**#define SUCCESS 0**

**#define CANNOT\_MALLOC\_WORDARRAY 1**

**#define NO\_WORDS\_ON\_INPUT 2**

**#define NO\_MEMORY\_FOR\_WORDNODE 3**

**#define NO\_MEMORY\_FOR\_WORD 4**

**#define NONALPHA "1234567890 \v\f\n\t\r+=-\*/\\,.;:'#~?<>|{}[]`!\"£$%^&()"**

**int** ReadInputToTree(WORD \*\*DestTree, size\_t \*Treecount, FILE \*Input);

**int** AddToTree(WORD \*\*DestTree, size\_t \*Treecount, **char** \*Word);

**int** WalkTree(WORD \*\*DestArray, WORD \*Word);

**int** CompareCounts(**const** **void** \*vWord1, **const** **void** \*vWord2);

**int** OutputWords(FILE \*Dest, size\_t Count, WORD \*\*WordArray);

**void** FreeTree(WORD \*W);

**char** \*dupstr(**char** \*s);

**int** main(**void**)

{

**int** Status = SUCCESS;

WORD \*Words = NULL;

size\_t Treecount = **0**;

WORD \*\*WordArray = NULL;

/\* Read the words on stdin into a tree \*/

**if**(SUCCESS == Status)

{

Status = ReadInputToTree(&Words, &Treecount, stdin);

}

/\* Sanity check for no sensible input \*/

**if**(SUCCESS == Status)

{

**if**(**0** == Treecount)

{

Status = NO\_WORDS\_ON\_INPUT;

}

}

/\* allocate a sufficiently large array \*/

**if**(SUCCESS == Status)

{

WordArray = malloc(Treecount \* **sizeof** \*WordArray);

**if**(NULL == WordArray)

{

Status = CANNOT\_MALLOC\_WORDARRAY;

}

}

/\* Walk the tree into the array \*/

**if**(SUCCESS == Status)

{

Status = WalkTree(WordArray, Words);

}

/\* qsort the array \*/

**if**(SUCCESS == Status)

{

qsort(WordArray, Treecount, **sizeof** \*WordArray, CompareCounts);

}

/\* walk down the WordArray outputting the values \*/

**if**(SUCCESS == Status)

{

Status = OutputWords(stdout, Treecount, WordArray);

}

/\* free the word array \*/

**if**(NULL != WordArray)

{

free(WordArray);

WordArray = NULL;

}

/\* and free the tree memory \*/

**if**(NULL != Words)

{

FreeTree(Words);

Words = NULL;

}

/\* Error report and we are finshed \*/

**if**(SUCCESS != Status)

{

fprintf(stderr, "Program failed with code %d\n", Status);

}

**return** (SUCCESS == Status ? EXIT\_SUCCESS : EXIT\_FAILURE);

}

**void** FreeTree(WORD \*W)

{

**if**(NULL != W)

{

**if**(NULL != W->Word)

{

free(W->Word);

W->Word = NULL;

}

**if**(NULL != W->Left)

{

FreeTree(W->Left);

W->Left = NULL;

}

**if**(NULL != W->Right)

{

FreeTree(W->Right);

W->Right = NULL;

}

}

}

**int** AddToTree(WORD \*\*DestTree, size\_t \*Treecount, **char** \*Word)

{

**int** Status = SUCCESS;

**int** CompResult = **0**;

/\* safety check \*/

assert(NULL != DestTree);

assert(NULL != Treecount);

assert(NULL != Word);

/\* ok, either \*DestTree is NULL or it isn't (deep huh?) \*/

**if**(NULL == \*DestTree) /\* this is the place to add it then \*/

{

\*DestTree = malloc(**sizeof** \*\*DestTree);

**if**(NULL == \*DestTree)

{

/\* horrible - we're out of memory \*/

Status = NO\_MEMORY\_FOR\_WORDNODE;

}

**else**

{

(\*DestTree)->Left = NULL;

(\*DestTree)->Right = NULL;

(\*DestTree)->Count = **1**;

(\*DestTree)->Word = dupstr(Word);

**if**(NULL == (\*DestTree)->Word)

{

/\* even more horrible - we've run out of memory in the middle \*/

Status = NO\_MEMORY\_FOR\_WORD;

free(\*DestTree);

\*DestTree = NULL;

}

**else**

{

/\* everything was successful, add one to the tree nodes count \*/

++\*Treecount;

}

}

}

**else** /\* we need to make a decision \*/

{

CompResult = strcmp(Word, (\*DestTree)->Word);

**if**(**0** < CompResult)

{

Status = AddToTree(&(\*DestTree)->Left, Treecount, Word);

}

**else** **if**(**0** > CompResult)

{

Status = AddToTree(&(\*DestTree)->Left, Treecount, Word);

}

**else**

{

/\* add one to the count - this is the same node \*/

++(\*DestTree)->Count;

}

} /\* end of else we need to make a decision \*/

**return** Status;

}

**int** ReadInputToTree(WORD \*\*DestTree, size\_t \*Treecount, FILE \*Input)

{

**int** Status = SUCCESS;

**char** Buf[**8192**] = {**0**};

**char** \*Word = NULL;

/\* safety check \*/

assert(NULL != DestTree);

assert(NULL != Treecount);

assert(NULL != Input);

/\* for every line \*/

**while**(NULL != fgets(Buf, **sizeof** Buf, Input))

{

/\* strtok the input to get only alpha character words \*/

Word = strtok(Buf, NONALPHA);

**while**(SUCCESS == Status && NULL != Word)

{

/\* deal with this word by adding it to the tree \*/

Status = AddToTree(DestTree, Treecount, Word);

/\* next word \*/

**if**(SUCCESS == Status)

{

Word = strtok(NULL, NONALPHA);

}

}

}

**return** Status;

}

**int** WalkTree(WORD \*\*DestArray, WORD \*Word)

{

**int** Status = SUCCESS;

**static** WORD \*\*Write = NULL;

/\* safety check \*/

assert(NULL != Word);

/\* store the starting point if this is the first call \*/

**if**(NULL != DestArray)

{

Write = DestArray;

}

/\* Now add this node and it's kids \*/

**if**(NULL != Word)

{

\*Write = Word;

++Write;

**if**(NULL != Word->Left)

{

Status = WalkTree(NULL, Word->Left);

}

**if**(NULL != Word->Right)

{

Status = WalkTree(NULL, Word->Right);

}

}

**return** Status;

}

/\*

CompareCounts is called by qsort. This means that it gets pointers to the

data items being compared. In this case the data items are pointers too.

\*/

**int** CompareCounts(**const** **void** \*vWord1, **const** **void** \*vWord2)

{

**int** Result = **0**;

WORD \* **const** \*Word1 = vWord1;

WORD \* **const** \*Word2 = vWord2;

assert(NULL != vWord1);

assert(NULL != vWord2);

/\* ensure the result is either 1, 0 or -1 \*/

**if**((\*Word1)->Count < (\*Word2)->Count)

{

Result = **1**;

}

**else** **if**((\*Word1)->Count > (\*Word2)->Count)

{

Result = -**1**;

}

**else**

{

Result = **0**;

}

**return** Result;

}

**int** OutputWords(FILE \*Dest, size\_t Count, WORD \*\*WordArray)

{

**int** Status = SUCCESS;

size\_t Pos = **0**;

/\* safety check \*/

assert(NULL != Dest);

assert(NULL != WordArray);

/\* Print a header \*/

fprintf(Dest, "Total Words : %lu\n", (**unsigned** **long**)Count);

/\* Print the words in descending order \*/

**while**(SUCCESS == Status && Pos < Count)

{

fprintf(Dest, "%10lu %s\n", (**unsigned** **long**)WordArray[Pos]->Count, WordArray[Pos]->Word);

++Pos;

}

**return** Status;

}

/\*

dupstr: duplicate a string

\*/

**char** \*dupstr(**char** \*s)

{

**char** \*Result = NULL;

size\_t slen = **0**;

/\* sanity check \*/

assert(NULL != s);

/\* get string length \*/

slen = strlen(s);

/\* allocate enough storage \*/

Result = malloc(slen + **1**);

/\* populate string \*/

**if**(NULL != Result)

{

memcpy(Result, s, slen);

\*(Result + slen) = **'\0'**;

}

**return** Result;

}

**Answer to Exercise 6-5, page 145**

*Write a function undef that will remove a name and definition from the table maintained by lookup and install .*

**int** undef(**char** \* name) {

**struct** nlist \* np1, \* np2;

**if** ((np1 = lookup(name)) == NULL) /\* name not found \*/

**return** **1**;

**for** ( np1 = np2 = hashtab[hash(name)]; np1 != NULL;

np2 = np1, np1 = np1->next ) {

**if** ( strcmp(name, np1->name) == **0** ) { /\* name found \*/

/\* Remove node from list \*/

**if** ( np1 == np2 )

hashtab[hash(name)] = np1->next;

**else**

np2->next = np1->next;

/\* Free memory \*/

free(np1->name);

free(np1->defn);

free(np1);

**return** **0**;

}

}

**return** **1**; /\* name not found \*/

}

Gregory Pietsch's solution

**void** undef(**char** \*s)

{

**struct** nlist \*np1, \*np2;

**unsigned** hashval = hash(s);

**for** (np1 = hashtab[hashval], np2 = NULL;

np1 != NULL;

np2 = np1, np1 = np1->next)

**if** (strcmp(s, np1->name) == **0**) {

/\* found a match \*/

free(np1->name);

free(np1->defn);

**if** (np2 == NULL)

/\* at the beginning? \*/

hashtab[hashval] = np1->next;

**else**

/\* in the middle or at the end? \*/

np2->next = np1->next;

free(np1);

**return**;

}

}

**Answer to Exercise 7-1, page 153**

*Write a program that converts upper case to lower or lower case to upper, depending on the name it is invoked with, as found in argv[0].*

/\* This program converts its input to upper case

\* (if argv[0] begins with U or u) or lower case.

\* If argc is 0, it prints an error and quits.

\*/

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <ctype.h>**

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

{

**int** (\*convcase[**2**])(**int**) = {toupper, tolower};

**int** func;

**int** result = EXIT\_SUCCESS;

**int** ch;

**if**(argc > **0**)

{

**if**(toupper((**unsigned** **char**)argv[**0**][**0**]) == **'U'**)

{

func = **0**;

}

**else**

{

func = **1**;

}

**while**((ch = getchar()) != EOF)

{

ch = (\*convcase[func])((**unsigned** **char**)ch);

putchar(ch);

}

}

**else**

{

fprintf(stderr, "Unknown name. Can't decide what to do.\n");

result = EXIT\_FAILURE;

}

**return** result;

}

Here's a category 1 solution from Bryan Williams...

/\*

Exercise 7-1. Write a program that converts upper case to lower case or lower case to upper,

depending on the name it is invoked with, as found in argv[0].

Assumptions: The program should read from stdin, until EOF, converting the output to stdout

appropriately.

The correct outputs should be :

Program Name Output

lower stdin with all caps converted to lower case

upper stdin with all lowercase characters converted to uppercase

[anything else] helpful message explaining how to use this

Author : Bryan Williams

\*/

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <ctype.h>**

**#define SUCCESS 0**

**#define NO\_ARGV0 1**

**#define BAD\_NAME 2**

**int** main(**int** argc, **char** \*argv[])

{

**int** ErrorStatus = SUCCESS;

**int** (\*convert)(**int** c) = NULL;

**int** c = **0**;

/\* check that there were any arguments \*/

**if**(SUCCESS == ErrorStatus)

{

**if**(**0** >= argc)

{

printf("Your environment has not provided a single argument for the program name.\n");

ErrorStatus = NO\_ARGV0;

}

}

/\* check for valid names in the argv[0] string \*/

**if**(SUCCESS == ErrorStatus)

{

**if**(**0** == strcmp(argv[**0**], "lower"))

{

convert = tolower;

}

**else** **if**(**0** == strcmp(argv[**0**], "upper"))

{

convert = toupper;

}

**else**

{

printf("This program performs two functions.\n");

printf("If the executable is named lower then it converts all the input on stdin to lowercase.\n");

printf("If the executable is named upper then it converts all the input on stdin to uppercase.\n");

printf("As you have named it %s it prints this message.\n", argv[**0**]);

ErrorStatus = BAD\_NAME;

}

}

/\* ok so far, keep looping until EOF is encountered \*/

**if**(SUCCESS == ErrorStatus)

{

**while**(EOF != (c = getchar()))

{

putchar((\*convert)(c));

}

}

/\* and return what happened \*/

**return** SUCCESS == ErrorStatus ? EXIT\_SUCCESS : EXIT\_FAILURE;

}

**Answer to Exercise 7-2, page 155**

*Write a program that will print arbitrary input in a sensible way. As a minimum, it should print non-graphic characters in octal or hexadecimal according to local custom, and break long text lines.*

/\* Use -o for octal output, -x for hexadecimal

\*/

**#include <stdio.h>**

**#define OCTAL 8**

**#define HEXADECIMAL 16**

**void** ProcessArgs(**int** argc, **char** \*argv[], **int** \*output)

{

**int** i = **0**;

**while**(argc > **1**)

{

--argc;

**if**(argv[argc][**0**] == **'-'**)

{

i = **1**;

**while**(argv[argc][i] != **'\0'**)

{

**if**(argv[argc][i] == **'o'**)

{

\*output = OCTAL;

}

**else** **if**(argv[argc][i] == **'x'**)

{

\*output = HEXADECIMAL;

}

**else**

{

/\* Quietly ignore unknown switches, because we don't want to

\* interfere with the program's output. Later on in the

\* chapter, the delights of fprintf(stderr, "yadayadayada\n")

\* are revealed, just too late for this exercise.

\*/

}

++i;

}

}

}

}

**int** can\_print(**int** ch)

{

**char** \*printable = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890 !\"#%&'()\*+,-./:;<=>?[\\]^\_{|}~\t\f\v\r\n";

**char** \*s;

**int** found = **0**;

**for**(s = printable; !found && \*s; s++)

{

**if**(\*s == ch)

{

found = **1**;

}

}

**return** found;

}

**int** main(**int** argc, **char** \*argv[])

{

**int** split = **80**;

**int** output = HEXADECIMAL;

**int** ch;

**int** textrun = **0**;

**int** binaryrun = **0**;

**char** \*format;

**int** width = **0**;

ProcessArgs(argc, argv, &output);

**if**(output == HEXADECIMAL)

{

format = "%02X ";

width = **4**;

}

**else**

{

format = "%3o ";

width = **4**;

}

**while**((ch = getchar()) != EOF)

{

**if**(can\_print(ch))

{

**if**(binaryrun > **0**)

{

putchar(**'\n'**);

binaryrun = **0**;

textrun = **0**;

}

putchar(ch);

++textrun;

**if**(ch == **'\n'**)

{

textrun = **0**;

}

**if**(textrun == split)

{

putchar(**'\n'**);

textrun = **0**;

}

}

**else**

{

**if**(textrun > **0** || binaryrun + width >= split)

{

printf("\nBinary stream: ");

textrun = **0**;

binaryrun = **15**;

}

printf(format, ch);

binaryrun += width;

}

}

putchar(**'\n'**);

**return** **0**;

}

**Answer to Exercise 7-3, page 156**

*Revise minprintf to handle more of the other facilities of printf .*

/\* Gregory Pietsch - K&R2 Exercise 7-3 - 2001-01-28 gkp@flash.net \*/

**#include <stdarg.h>**

**#include <stdio.h>**

/\* minprintf: minimal printf with variable argument list \*/

**void** minprintf(**char** \*fmt, ...)

{

va\_list ap;

**char** \*p, \*sval;

**int** ival;

**double** dval;

**unsigned** uval;

va\_start(ap, fmt); /\* make ap point to the first unnamed arg \*/

**for** (p = fmt; \*p; p++) {

**if** (\*p != **'%'**) {

putchar(\*p);

**continue**;

}

**switch** (\*++p) {

**case** **'d'**:

**case** **'i'**:

ival = va\_arg(ap, **int**);

printf("%d", ival);

**break**;

**case** **'c'**:

ival = va\_arg(ap, **int**);

putchar(ival);

**break**;

**case** **'u'**:

uval = va\_arg(ap, **unsigned** **int**);

printf("%u", uval);

**break**;

**case** **'o'**:

uval = va\_arg(ap, **unsigned** **int**);

printf("%o", uval);

**break**;

**case** **'x'**:

uval = va\_arg(ap, **unsigned** **int**);

printf("%x", uval);

**break**;

**case** **'X'**:

uval = va\_arg(ap, **unsigned** **int**);

printf("%X", uval);

**break**;

**case** **'e'**:

dval = va\_arg(ap, **double**);

printf("%e", dval);

**break**;

**case** **'f'**:

dval = va\_arg(ap, **double**);

printf("%f", dval);

**break**;

**case** **'g'**:

dval = va\_arg(ap, **double**);

printf("%g", dval);

**break**;

**case** **'s'**:

**for** (sval = va\_arg(ap, **char** \*); \*sval; sval++)

putchar(\*sval);

**break**;

**default**:

putchar(\*p);

**break**;

}

}

va\_end(ap);

}

/\* end of function \*/

**Answer to Exercise 7-6, page 165**

*Write a program to compare two files, printing the first line where they differ.*   
  
Here's Rick's solution:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

KnR 7-6

--------

Write a program to compare two files and print the

first line where they differ.

Author: Rick Dearman

email: rick@ricken.demon.co.uk

Note: This program prints ALL the lines that are

different using the <> indicators used by

the unix diff command. However this program

will not cope with something as simple as a

line being removed.

In reality the program would be more useful

if it searched forward for matching lines.

This would be a better indicator of the simple

removal of some lines.

This has lead me to track down a version of the

"diff" command available on GNU/Linux systems.

for more information go to the web site at:

www.gnu.org

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**#include <stdio.h>**

**#include <string.h>**

**#define MAXLINE 1000**

**void** diff\_line( **char** \*lineone, **char** \*linetwo, **int** linenumber )

{

**if**(strcmp (lineone, linetwo) < **0** || strcmp (lineone, linetwo) > **0**)

printf( "%d<%s\n%d>%s\n", linenumber, lineone, linenumber, linetwo);

}

**int** main(**int** argc, **char** \*argv[] )

{

FILE \*fp1, \*fp2;

**char** fp1\_line[MAXLINE], fp2\_line[MAXLINE];

**int** i;

**if** ( argc != **3** )

{

printf("differ fileone filetwo\n");

exit(**0**);

}

fp1 = fopen( argv[**1**], "r" );

**if** ( ! fp1 )

{

printf("Error opening file %s\n", argv[**1**]);

}

fp2 = fopen( argv[**2**], "r" );

**if** ( ! fp2 )

{

printf("Error opening file %s\n", argv[**2**]);

}

i = **0**;

**while** ( (fgets(fp1\_line, MAXLINE, fp1) != NULL)

&& (fgets(fp2\_line, MAXLINE, fp2) != NULL))

{

diff\_line( fp1\_line, fp2\_line, i );

i++;

}

**return** **0**;

}

and here's "flippant squirrel"'s solution:

/\* Exercise 7-6 - write a program to compare two files, printing the first line

\* where they differ

\*

\* Note : I amended this a bit...if a file is shorter than the other, but is identical

\* up to that point, the program prints out "EOF" as the string that's not equal.

\*

\*/

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <string.h>**

**#define BUFF\_SIZE 1000**

/\* uses fgets, removes the '\n' at the end of the string if it exists \*/

**char** \*safegets(**char** \*buffer, **int** length, FILE \*file)

{

**char** \*ptr;

**int** len;

**if** (buffer != NULL)

{

ptr = fgets(buffer, length, file);

**if** (ptr != NULL)

{

len = strlen(buffer);

**if** (len > **0**)

{

**if** (buffer[len - **1**] == **'\n'**)

{

buffer[len - **1**] = **'\0'**;

}

}

}

**return** ptr;

}

**return** NULL;

}

**int** main(**int** argc, **char** \*argv[])

{

FILE \*leftFile, \*rightFile;

**char** buff1[BUFF\_SIZE], buff2[BUFF\_SIZE];

**char** \*ptr1, \*ptr2;

**unsigned** **long** lineNum = **0**;

**if** (argc < **3**)

{

fprintf(stderr, "Usage : 7\_6 <path to file> <path to file>\n");

**return** **0**;

}

**if** (!(leftFile = fopen(argv[**1**], "r")))

{

fprintf(stderr, "Couldn't open %s for reading\n", argv[**1**]);

**return** **0**;

}

**if** (!(rightFile = fopen(argv[**2**], "r")))

{

fprintf(stderr, "Couldn't open %s for reading\n", argv[**2**]);

fclose(leftFile); /\* RJH 10 Jul 2000 \*/

**return** **0**;

}

/\* read through each file, line by line \*/

ptr1 = safegets(buff1, BUFF\_SIZE, leftFile);

ptr2 = safegets(buff2, BUFF\_SIZE, rightFile);

++lineNum;

/\* stop when either we've exhausted either file's data \*/

**while** (ptr1 != NULL && ptr2 != NULL)

{

/\* compare the two lines \*/

**if** (strcmp(buff1, buff2) != **0**)

{

printf("Difference:\n");

printf("%lu\t\"%s\" != \"%s\"\n", lineNum, buff1, buff2);

**goto** CleanUp;

}

ptr1 = safegets(buff1, BUFF\_SIZE, leftFile);

ptr2 = safegets(buff2, BUFF\_SIZE, rightFile);

++lineNum;

}

/\*

\* if one of the files ended prematurely, it definitely

\* isn't equivalent to the other

\*/

**if** (ptr1 != NULL && ptr2 == NULL)

{

printf("Difference:\n");

printf("%lu\t\"%s\" != \"EOF\"\n", lineNum, buff1);

}

**else** **if** (ptr1 == NULL && ptr2 != NULL)

{

printf("Difference:\n");

printf("%lu\t\"EOF\" != \"%s\"\n", lineNum, buff2);

}

**else**

{

printf("No differences\n");

}

CleanUp:

fclose(leftFile);

fclose(rightFile);

**return** EXIT\_SUCCESS;

}

**Answer to Exercise 7-8, page 165**

*Write a program to print a set of files, starting each new one on a new page, with a title and a running page count for each file.*

/\* K&R Exercise 7-8 \*/

/\* Steven Huang \*/

/\*

Limitation: This program doesn't wrap long lines.

\*/

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <assert.h>**

**#define LINES\_PER\_PAGE 10**

**#define TRUE 1**

**#define FALSE 0**

**void** print\_file(**char** \*file\_name)

{

FILE \*f;

**int** page\_number = **1**;

**int** line\_count;

**int** c;

**int** new\_page = TRUE;

assert(file\_name != NULL);

**if** ((f = fopen(file\_name, "r")) != NULL) {

**while** ((c = fgetc(f)) != EOF) {

**if** (new\_page) {

/\* print out the header \*/

printf("[%s] page %d starts\n", file\_name, page\_number);

new\_page = FALSE;

line\_count = **1**;

}

putchar(c);

**if** (c == **'\n'** && ++line\_count > LINES\_PER\_PAGE) {

/\* print out the footer \*/

printf("[%s] page %d ends\n", file\_name, page\_number);

/\* skip another line so we can see it on screen \*/

putchar(**'\n'**);

new\_page = TRUE;

page\_number++;

}

}

**if** (!new\_page) {

/\* file ended in the middle of a page, so we still need to

print a footer \*/

printf("[%s] page %d ends\n", file\_name, page\_number);

}

/\* skip another line so we can see it on screen \*/

putchar(**'\n'**);

fclose(f);

}

}

**int** main(**int** argc, **char** \*argv[])

{

**int** i;

**if** (argc < **2**) {

fputs("no files specified\n", stderr);

**return** EXIT\_FAILURE;

}

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

print\_file(argv[i]);

}

**return** EXIT\_SUCCESS;

}

**Answer to Exercise 7-9, page 168**

*Functions like isupper can be implemented to save space or to save time. Explore both possibilities.*   
  
This question is best left to an essay rather than code, so here's my take: The easiest way to implement the eleven is() functions in C90's version of <ctype.h> is via a table lookup. If UCHAR\_MAX is 255, then a table would take up around 514 8-bit bytes and still have room for five more is() functions. In modern programs, this is a miniscule expense of both space and time since a mere table lookup doesn't cost a whole lot (although space may be a priority for embedded systems). Additionally, since the is() functions of <ctype.h> are locale-dependent and therefore subject to locale-specific whims, a table could more easily be modified than modifying hard calculations. Consider the following three implementations of isupper() : Implementation #1:

**int** isupper(**int** c)

{

**return** (c >= **'A'** && c <= **'Z'**);

}

Implementation #2:

**int** isupper(**int** c)

{

**return** (strchr("ABCDEFGHIJKLMNOPQRSTUVWXYZ", c) != NULL);

}

Implementation #3:

/\* Presumably, \_UP is a power of 2 and

\* \_Ctype is a table

\*/

**int** isupper(**int** c)

{

**return** ((\_Ctype[(**unsigned** **char**)c] & \_UP) != **0**);

}

Implementation #1 fails in EBCDIC and implementation #2 fails in a locale that adds more upperspace characters than the ones mentioned. Implementation #3, however, suggests that \_Ctype[] can be modified to accommodate new uppercase characters.

**Answer to Exercise 8-1, page 174**

Ron Scott has also sent me a solution to this exercise. Once he has granted me permission to display it here, I will post it on this site.   
*Rewrite the program cat from Chapter 7 using read , write , open and close instead of their standard library equivalents. Perform experiments to determine the relative speeds of the two versions.*

/\*

Andrew Tesker

ucat.c

a version of cat using UNIX system access

\*/

**#include <stdio.h>**

**#include <fcntl.h>**

**#define BUFSIZE 1024**

**int** main(**int** argc, **char** \*argv[])

{

**int** fd1;

**void** filecopy(**int** f, **int** t);

**if**(argc == **1**)

filecopy(**0**, **1**);

**else** {

**while**(--argc > **0**)

**if**(( fd1 = open(\*++argv, O\_RDONLY, **0**)) == -**1**) {

printf("unix cat: can't open %s\n", \*argv);

**return** **1**;

}

**else** {

filecopy(fd1, **1**);

close(fd1);

}

}

**return** **0**;

}

**void** filecopy(**int** from, **int** to)

{

**int** n;

**char** buf[BUFSIZE];

**while**((n=read(from, buf, BUFSIZE)) > **0** )

write(to, buf, n);

}

**Answer to Exercise 8-3, page 179**

*Design and write \_flushbuf , fflush , and fclose .*

/\* Editor's note: Gregory didn't supply a main() for this. Normally, in these situations,

\* I'd supply one myself, so that you can easily run and test the code. But, in this case,

\* I wouldn't know where to start! If anyone wants to fill the gap, please let me know.

\* Thanks.

\* RJH, 28 June 2000

\*/

**#include <stdio.h>**

/\* on p.176 \*/

**#include "syscalls.h"**

/\* or stdlib.h \*/

/\* \_flushbuf - flush a buffer

\* According to the code on p. 176, \_flushbuf

\* is what putc calls when the buffer is full.

\* EOF as the character causes everything to

\* be written -- I don't tack on the EOF.

\*/

**int** \_flushbuf(**int** c, FILE \*f)

{

**int** num\_written, bufsize;

**unsigned** **char** uc = c;

**if** ((f->flag & (\_WRITE|\_EOF|\_ERR)) != \_WRITE)

**return** EOF;

**if** (f->base == NULL && ((f->flag & \_UNBUF) == **0**)) {

/\* no buffer yet \*/

**if** ((f->base = malloc(BUFSIZ)) == NULL)

/\* couldn't allocate a buffer, so try unbuffered \*/

f->flag |= \_UNBUF;

**else** {

f->ptr = f->base;

f->cnt = BUFSIZ - **1**;

}

}

**if** (f->flag & \_UNBUF) {

/\* unbuffered write \*/

f->ptr = f->base = NULL;

f->cnt = **0**;

**if** (c == EOF)

**return** EOF;

num\_written = write(f->fd, &uc, **1**);

bufsize = **1**;

} **else** {

/\* buffered write \*/

**if** (c != EOF)

f->ptr++ = uc;

bufsize = (**int**)(f->ptr - f->base);

num\_written = write(f->fd, fp->base, bufsize);

f->ptr = f->base;

f->cnt = BUFSIZ - **1**;

}

**if** (num\_written == bufsize)

**return** c;

**else** {

f->flag |= \_ERR;

**return** EOF;

}

}

/\* fflush \*/

**int** fflush(FILE \*f)

{

**int** retval;

**int** i;

retval = **0**;

**if** (f == NULL) {

/\* flush all output streams \*/

**for** (i = **0**; i < OPEN\_MAX; i++) {

**if** ((\_iob[i]->flag & \_WRITE) && (fflush(\_iob[i]) == -**1**))

retval = -**1**;

}

} **else** {

**if** ((f->flag & \_WRITE) == **0**)

**return** -**1**;

\_flushbuf(EOF, f);

**if** (f->flag & \_ERR)

retval = -**1**;

}

**return** retval;

}

/\* fclose \*/

**int** fclose(FILE \*f)

{

**int** fd;

**if** (f == NULL)

**return** -**1**;

fd = f->fd;

fflush(f);

f->cnt = **0**;

f->ptr = NULL;

**if** (f->base != NULL)

free(f->base);

f->base = NULL;

f->flag = **0**;

f->fd = -**1**;

**return** close(fd);

}

**Answer to Exercise 8-4, page 179**

*The standard library function   
int fseek(FILE \*fp, long offset, int origin)   
is identical to lseek except that fp is a file pointer instead of a file descriptor and the return value is an int status, not a position. Write fseek . Make sure that your fseek coordinates properly with the buffering done for the other functions of the library.*   
  
Here's Gregory's first solution:

/\* Gregory Pietsch -- My category 0 solution to 8-4 \*/

**int** fseek(FILE \*f, **long** offset, **int** whence)

{

**if** ((f->flag & \_UNBUF) == **0** && base != NULL) {

/\* deal with buffering \*/

**if** (f->flag & \_WRITE) {

/\* writing, so flush buffer \*/

fflush(f); /\* from 8-3 \*/

} **else** **if** (f->flag & \_READ) {

/\* reading, so trash buffer \*/

f->cnt = **0**;

f->ptr = f->base;

}

}

**return** (lseek(f->fd, offset, whence) < **0**);

}

...and here's his second, which is considerably more comprehensive:

/\*

[The following solution is in the zip file as krx80401.c - RJH (ed.) ]

EXERCISE 8-4

I thought I'd improve 8-4 too. I'm trying my best to get this as close

to ISO C as possible given the restrictions that I'm under. (A real

implementation would have fsetpos() borrow some of the same code.)

\*/

/\* Gregory Pietsch -- My category 0 solution to 8-4 \*/

**#define SEEK\_SET 0**

**#define SEEK\_CUR 1**

**#define SEEK\_END 2**

**int** fseek(FILE \*f, **long** offset, **int** whence)

{

**int** result;

**if** ((f->flag & \_UNBUF) == **0** && base != NULL) {

/\* deal with buffering \*/

**if** (f->flag & \_WRITE) {

/\* writing, so flush buffer \*/

**if** (fflush(f))

**return** EOF; /\* from 8-3 \*/

} **else** **if** (f->flag & \_READ) {

/\* reading, so trash buffer --

\* but I have to do some housekeeping first

\*/

**if** (whence == SEEK\_CUR) {

/\* fix offset so that it's from the last

\* character the user read (not the last

\* character that was actually read)

\*/

**if** (offset >= **0** && offset <= f->cnt) {

/\* easy shortcut \*/

f->cnt -= offset;

f->ptr += offset;

f->flags &= ~\_EOF; /\* see below \*/

**return** **0**;

} **else**

offset -= f->cnt;

}

f->cnt = **0**;

f->ptr = f->base;

}

}

result = (lseek(f->fd, offset, whence) < **0**);

**if** (result == **0**)

f->flags &= ~\_EOF; /\* if successful, clear EOF flag \*/

**return** result;

}

**Answer to Exercise 8-6, page 189**

*The standard library function calloc(n,size) returns a pointer to n objects of size size , with the storage initialized to zero. Write calloc , by calling malloc or by modifying it.*

/\*

Exercise 8.6. The standard library function calloc(n, size) returns a pointer to n objects

of size size, with the storage initialised to zero. Write calloc, by calling

malloc or by modifying it.

Author: Bryan Williams

\*/

**#include <stdlib.h>**

**#include <string.h>**

/\*

Decided to re-use malloc for this because :

1) If the implementation of malloc and the memory management layer changes, this will be ok.

2) Code re-use is great.

\*/

**void** \*mycalloc(size\_t nmemb, size\_t size)

{

**void** \*Result = NULL;

/\* use malloc to get the memory \*/

Result = malloc(nmemb \* size);

/\* and clear the memory on successful allocation \*/

**if**(NULL != Result)

{

memset(Result, **0x00**, nmemb \* size);

}

/\* and return the result \*/

**return** Result;

}

/\* simple test driver, by RJH \*/

**#include <stdio.h>**

**int** main(**void**)

{

**int** \*p = NULL;

**int** i = **0**;

p = mycalloc(**100**, **sizeof** \*p);

**if**(NULL == p)

{

printf("mycalloc returned NULL.\n");

}

**else**

{

**for**(i = **0**; i < **100**; i++)

{

printf("%08X ", p[i]);

**if**(i % **8** == **7**)

{

printf("\n");

}

}

printf("\n");

free(p);

}

**return** **0**;

}