

C Programs

- Converting numbers into words
- Finding out number of words, sentences, paragraphs, lines in a text file

/* C Program to convert numbers into words...*/

/* About this C program :

1. Converts Numerical amount to Rs. & paise only.
2. Limitations... upto 99 crores.
3. Use : you can use it in your billing program etc. by adapting the 'main()' function.
4. Is self explanatory .
5. Is freeware. But do let us know by e-mail if you use the code in your programs.
6. You use this program at your own risk. It is your responsibility to check out the code / program.
7. If you have any suggestions / queries kindly use the feedback form, or e-mail : rsn@dirpune.com

*/

```
#include <stdio.h>
```

```
long float numberf=0,numberpaisef=0,numberpf2;
```

```
long int numberpaise=0 ;
```

```
long int number=0,number2=0,numrem1=0,numrem2=0;
```

```
int flagfinish=0,i=0,j=0 ;
```

```
char numberstr[50];
```

```
char rsinword[151],rsinword2[51],rsinword3[51],rsinword4[51] ;
```

```
void proc_rs3(void);
```

```
void proc_rs4(void);
```

```
long int get_therem(long int x,long int y);
```

```
long int func_rs2(long int x,long int y);
```

```
void main()
```

```
{
```

```
printf("Enter Number : ") ;
```

```
scanf("%lf",&numberf) ;
```

```
flagfinish = 0 ;
```

```
numberpaisef=(numberf*100.0);
```

```

number = (long int)(numberf) ;
numberpf2=(number)*100;
numberpaisef=numberpaisef-numberpf2+0.5;

numberpaise= (long int)(numberpaisef);
/* numberpaise = int((numberf*100) % 100 );*/

/* printf("\nNumbers : 1: %f, 2: %f 3: %f\n", numberf,numberpf2,numberpaisef) ; */

/* printf("\nNumbers : 1: %ld 2: %ld\n", number,numberpaise) ; */

/* flagfinish set to 1 so comes out of loop */

if (number > 0)
{
strcpy(rsinword,"Rupees ");

while (flagfinish == 0)
{
if (number >= 10000000 && number <= 999999999)
{
number2 = 10000000 ;
numrem1 = func_rs2(number,number2) ;
proc_rs3() ;
strcat(rsinword," Crore");
number = get_there(number,number2) ;
}

else if (number >= 100000 && number <= 9999999)
{
number2 =100000 ;
numrem1=func_rs2(number,number2) ;
proc_rs3() ;
strcat(rsinword," Lakh") ;
number = get_there(number,number2) ;
}

else if (number >= 1000 && number <= 99999)
{
number2 =1000 ;
numrem1=func_rs2(number,number2) ;
proc_rs3() ;
strcat(rsinword," Thousand") ;
number = get_there(number,number2) ;
}

else if (number >= 100 && number <= 999)
{
number2 =100 ;

```

```

numrem1=func_rs2(number,number2) ;
proc_rs3() ;
strcat(rsinword," Hundred") ;
number = get_them(number,number2) ;
}
else if (number >= 10 && number <= 99)
{
numrem1=number ;
proc_rs3() ;
flagfinish = 1 ;
}
else if (number >= 1 && number <= 9)
{
numrem1 = number ;
proc_rs3() ;
flagfinish = 1 ;
}
else if (number<=0)
{
flagfinish = 1 ;
}

} /* end of while loop */
} /* end of if construct */

if (numberpaise!=0)
{
strcat(rsinword," and Paise ") ;

if (numberpaise >= 10 && numberpaise <= 99)
{
numrem1=numberpaise;
printf("\n %d",numrem1);
proc_rs3();
flagfinish=1;
}
if (numberpaise >= 1 && numberpaise <= 9)
{
numrem1=numberpaise;
printf("\n %d",numrem1);
proc_rs3();
flagfinish=1;
}

} /* end of if */

```

```

strcat(rsinword," Only");

if (strlen(rsinword)<=50)
{
for(i = 0,j = 0 ;i < strlen(rsinword); i++,j++)
{
rsinword2[j] = rsinword[i];
}
}/* if 50 */

if ( ( strlen(rsinword)>50) && (strlen(rsinword)<=100) )
{
for(i = 0,j = 0 ;i < 50; i++,j++)
{
rsinword2[j] = rsinword[i];
}
for(i = 50,j = 0 ;i < strlen(rsinword); i++,j++)
{
rsinword3[j] = rsinword[i];
}
}/* if 100 */

if ( ( strlen(rsinword)>100) && (strlen(rsinword)<=150) )
{
for(i = 0,j = 0 ;i < 50; i++,j++)
{
rsinword2[j] = rsinword[i];
}
for(i = 50,j = 0 ;i < 100; i++,j++)
{
rsinword3[j] = rsinword[i];
}
for(i = 100,j = 0 ;i < strlen(rsinword); i++,j++)
{
rsinword4[j] = rsinword[i];
}
}/* if 150 */

printf("\n %f",numberf);
printf("\n %s",rsinword2);
printf("\n %s",rsinword3);
printf("\n %s",rsinword4);

} /* end of function main() */

```

```
long int func_rs2(long int x,long int y)
{
return (long int) x/y;
}
```

```
void proc_rs3(void)
{
switch (numrem1)
{
case 1:
case 2:
case 3:
case 4:
case 5:
case 6:
case 7:
case 8:
case 9:
numrem2=numrem1;
proc_rs4();
break;
case 10:
strcat(rsinword," Ten" );
break;
case 11:
strcat(rsinword," Eleven" );
break;
case 12:
strcat(rsinword," Twelve");
break;
case 13:
strcat(rsinword," Thirteen");
break;
case 14:
strcat(rsinword," Fourteen");
break;
case 15:
strcat(rsinword," Fifteen");
break;
case 16:
strcat(rsinword," Sixteen");
break;
case 17:
strcat(rsinword," Seventeen");
break;
case 18:
```

```
strcat(rsinword, " Eighteen");
break;
case 19:
strcat(rsinword, " Nineteen");
break;
case 20:
case 21:
case 22:
case 23:
case 24:
case 25:
case 26:
case 27:
case 28:
case 29:
strcat(rsinword, " Twenty");
if (numrem1 >= 21)
{
numrem2=numrem1 % 10;
proc_rs4();
}
break;
case 30:
case 31:
case 32:
case 33:
case 34:
case 35:
case 36:
case 37:
case 38:
case 39:
strcat(rsinword, " Thirty" );

if (numrem1 >= 31)
{
numrem2=numrem1 % 10 ;
proc_rs4() ;
}
break;
case 40:
case 41:
case 42:
case 43:
case 44:
case 45:
case 46:
```

```
case 47:
case 48:
case 49:
strcat(rsinword, " Forty" );

if (numrem1 >= 41)
{
numrem2=numrem1 % 10 ;
proc_rs4() ;
}
break;
case 50:
case 51:
case 52:
case 53:
case 54:
case 55:
case 56:
case 57:
case 58:
case 59:
strcat(rsinword, " Fifty" );

if (numrem1 >= 51)
{
numrem2=numrem1 % 10 ;
proc_rs4() ;
}
break;
case 60:
case 61:
case 62:
case 63:
case 64:
case 65:
case 66:
case 67:
case 68:
case 69:
strcat(rsinword, " Sixty" );

if (numrem1 >= 61)
{
numrem2=numrem1 % 10 ;
proc_rs4() ;
}
break;
```

```
case 70:
case 71:
case 72:
case 73:
case 74:
case 75:
case 76:
case 77:
case 78:
case 79:
strcat(rsinword, " Seventy") ;
```

```
if (numrem1 >= 71)
{
numrem2=numrem1 % 10 ;
proc_rs4() ;
}
break;
case 80:
case 81:
case 82:
case 83:
case 84:
case 85:
case 86:
case 87:
case 88:
case 89:
strcat(rsinword, " Eighty") ;
```

```
if (numrem1 >= 81)
{
numrem2=numrem1 % 10 ;
proc_rs4() ;
}
break;
case 90:
case 91:
case 92:
case 93:
case 94:
case 95:
case 96:
case 97:
case 98:
case 99:
```

```

strcat(rsinword," Ninety" );

if (numrem1 >= 91)
{
numrem2=numrem1 % 10 ;
proc_rs4() ;
}
break;
} /*switch*/

}/* end of function proc_rs3 */

void proc_rs4(void)
{
switch (numrem2)
{
case 1 : strcat(rsinword," One" ) ;
break;
case 2 : strcat(rsinword," Two");
break;
case 3 : strcat(rsinword," Three");
break;
case 4 : strcat(rsinword," Four");
break;
case 5 : strcat(rsinword," Five");
break;
case 6 : strcat(rsinword," Six");
break;
case 7 : strcat(rsinword," Seven");
break;
case 8 : strcat(rsinword," Eight");
break;
case 9 : strcat(rsinword," Nine");
break;
} /* end of switch*/

} /* end of function proc_rs4 */

```

```

long int get_therem(long int x,long int y)
{
return (long int)(x % y) ;
}

```

```
/* Program to read number of words,sentences,lines,symbols in a text file */
```

```
/*About this C program :
```

1.Displays the number of words,sentences,lines,paragraphs,symbols in a pure text file

2.The program assumes that the file path is fixed... filename should exist in this fixed file path & be 8 char.(max.) for primary name & 3 char. (max.) for the extension.

You can modify the Filepath by changing the filepath in the program below to the directory of your choice

3. If the Text file has more than 22 lines, the program pauses the display to show screenful at a time.

4. Is self explanatory .

5. Is freeware.But do let us know by e-mail if you use the code in your programs.

6. You use this program at your own risk. It is your responsibility to check out the code / program.

```
*/
```

```
#include <stdio.h>
```

```
#include <dos.h>
```

```
#include <string.h>
```

```
void main()
```

```
{
```

```
int wrdctr,sentctr,symbctr,paractr,lctr,countchars1;
```

```
int wflg,sflg,syflg,newlineflg,syflg2;
```

```
FILE *fp;
```

```
char char1;
```

```
char fname1[13],fname2[70];
```

```
wrdctr = sentctr = symbctr = paractr = lctr = countchars1 = 0;
```

```
wflg = sflg = syflg = newlineflg = syflg2= 0;
```

```
clrscr();
```

```
wflg=0;
```

```
sflg=0;
```

```
printf("Enter file [pure text file...] : ") ;
```

```
gets(fname1);
```

```
printf("The File Contents & Statistics are...\n\n");
```

```
strcpy(fname2,"c:\\");
```

```
strcat(fname2,fname1);
```

```
if ((fp = fopen(fname2,"r")) == NULL)
```

```
{
```

```
fprintf(stderr,"Error Opening File ");
```

```
getch();
```

```
exit(1);
```

```
}
```

```

countchars1 = 0;

while(!feof(fp))
{
fscanf(fp,"%c",&char1);

if (countchars1==0)
{
if(char1 == ' '|| char1 == '\t')
{
wflg = 1;
}
if(char1 == '.' || char1 == '?' || char1 == '!')
{
sflg = 1;
wflg = 1;

}
}

countchars1++;

if(char1 == ' '|| char1 == '\t')
{
if(wflg == 0)
{
wflg = 1;
wrctr++;
}
}

if(char1 == '.' || char1 == '?' || char1 == '!')
{

if(sflg == 0 && countchars1 != 0)
{
sflg = 1;
sentctr++;
if(wflg == 0)
{
wflg = 1;
wrctr++;
}
}
}
if(char1 == '\n')
{
lctr++;
}
}

```

```

if(lctr % 22 == 0)
{
printf("\n << Press any key to continue ... >>");

getch(); /*shows one screen at a time */
}
if ((sflg==1)&&(newlineflg==0))
{
paractr++;
newlineflg = 1;
}
if(wflg == 0)
{
wrctr++;
}
}

/*char1 == \" || char1 == \"' || char1 == \"\")*/

if(char1 == '(' || char1 == ')' || char1 == '[' ||
char1 == ']' || char1 == '{' || char1 == '}' ||
char1 == '+' || char1 == '-' || char1 == '/' ||
char1 == '*' || char1 == '%' || char1 == '=' ||
char1 == '>' || char1 == '<' || char1 == ':' ||
char1 == ';' || char1 == '$' || char1 == ',' ||
char1 == '#' || char1 == '^' || char1 == '|' ||
char1 == '~' || char1 == '^' || char1 == '@' ||
char1 == '\')

{
if ((wflg==0)&&(syflg==0))
{
wrctr++;
}
syflg = 1;
sybctr++;
}
if (char1=="\" || char1=="'")
{
syflg=1;
sybctr++;
if (syflg2==1)
{
wrctr++;
}
if(syflg2==0)
{
syflg2=1;
}
}
}

```

```

}

}
if (char1!="\" && char1!="'")
{
syflg2=0;
}

/*&& char1 != '\t'*/
/*char1 != \" && char1 != '\" && char1 != '\\'*/

if(char1 != ' ' && char1 != '\t' && char1 != '.' && char1 != '?' && char1 != '!')
{
if(char1 != '(' && char1 != ')' && char1 != '[' &&
char1 != ']' && char1 != '{' && char1 != '}' &&
char1 != '+' && char1 != '-' && char1 != '/' &&
char1 != '*' && char1 != '%' && char1 != '=' &&
char1 != '>' && char1 != '<' && char1 != ':' &&
char1 != ';' && char1 != '$' && char1 != ',' &&
char1 != '#' && char1 != '^' && char1 != '|' &&
char1 != '~' && char1 != `` && char1 != '@' &&
char1 != \" && char1 != '\" && char1 != '\\' && char1 != '\n')
{
wflg=0;
if (newlineflg==1)
{
newlineflg=0;
}
}
}

if(char1 != '.' && char1 != '?' && char1 != '!')
{
sflg=0;
}

printf("%c",char1);
} /* end of while loop */

if (countchars1 > 0 && wflg == 0 && sflg==0)
{
wrctr++;
lctr++;
}

fclose(fp);
printf("\nFile Statistics :\n");

```

```

printf("-----\n");
printf("\nNo. Of Characters : %d",countchars1);
printf("\nNo. Of Words : %d",wrctr);
printf("\nNo. Of Sentences : %d",sentctr);
printf("\nNo. Of Paragraphs : %d",paractr);
printf("\nNo. Of Symbols : %d",symbctr);
printf("\nNo. Of Lines : %d",lctr);

printf("\n Press any key to exit ...");

getch();

} /*end of main() */

```

QUESTION:

to write a function strend(s,t), which returns 1 if the string "t" occurs at the end of the string "s", and 0 otherwise,using pointers

```
*/
```

```
/*
```

limitations and assumptions:

1. This is program with function strend to test strend.
2. The character tested for is 't' can be easily changed to another
3. Length of string can be obtained by using strlen function but here in the function strend, it has been obtained by actual count

```
*/
```

```
#include<stdio.h>
```

```
#include<conio.h>
```

```
int strend(char *, char);
```

```
void main()
```

```
{
```

```
char str[80]="\0",ch='t';
```

```
int status;
```

```
printf("\nplease enter a string:");
```

```
gets(str);
```

```
status=strend(str,ch);
```

```
switch(status)
```

```
{
```

```
case 0:printf("\n %c NOT found as last letter", ch);
```

```
break;
```

```

case 1:printf("\n %c found as last letter", ch);
break;
}
printf("\n Press any Key ...");
getche();
}

/*
function strend takes in a character pointer(string to be searched)
and a character (character to search for)
It returns 1 if the character is the last character in string
else it returns 0

*/
int strend(char *s, char ch)
{
int length=0;
char *p;
p=s;
while(*p)
{
p++;
length++;
}
if (s[length-1]==ch) return 1;
else if (s[length-1]!=ch) return 0;
}

/*

```

QUESTION:

write a program that prints the distinct words in its input sorted into decreasing order of frequency of occurrence.print each word by its count.

```

*/

/*
limitations and assumptions:
1. the strings are separated by spaces.
2. the carriage return CR decides the end of entries
3. the string length can be varied by changing
#define MAXSTRLEN 20
4. the max number of strings can be varied by changing

```

```
# define MAXSTR 100
```

5. Program done using only standard C.

6. Bubble sort used for sorting. You can use other sorts.

```
*/
```

```
#include<stdio.h>
```

```
#include<conio.h>
```

```
#include<string.h>
```

```
#define MAXSTRLEN 20
```

```
#define MAXSTR 100
```

```
void main()
```

```
{
```

```
struct sarr2
```

```
{
```

```
char sarr1[MAXSTRLEN];
```

```
int num;
```

```
}sarray2[MAXSTR];
```

```
char ch=' ',prevch=' ',sarray[MAXSTR][MAXSTRLEN],tempstr[MAXSTRLEN];
```

```
int i=0,j=0,numstr=0,count=0,tempnum=0;
```

```
printf("\n enter : ");
```

```
do
```

```
{
```

```
ch=getche();
```

```
if (ch!='\r')
```

```
{
```

```
if(ch==' ')
```

```
{
```

```
if (prevch!=' ')
```

```
{
```

```
i++;
```

```
j=0;
```

```
}
```

```
prevch=ch;
```

```
}
```

```
else if (ch!=' ')
```

```
{
```

```
sarray[i][j]=ch;
```

```
prevch=ch;
```

```
j++;
```

```
}
```

```
}
```

```
}while(ch!='\r');
```

```
numstr=i;
```

```
if(ch=='\r') printf("\n\nEnter pressed, number of strings=%d ", numstr+1);
```

```

/*
sort sarray
*/
for(i=0;i<=numstr-1;i++)
for(j=i+1;j<=numstr;j++)
{
if (strcmp(sarray[i],sarray[j])>0)
{
strcpy(tempstr,sarray[i]);
strcpy(sarray[i],sarray[j]);
strcpy(sarray[j],tempstr);
}
}

```

```

/*
Initialise the array 2
*/
for(i=0;i<MAXSTR;i++)
{
strcpy(sarray2[i].sarr1, " \0");
sarray2[i].num=0;
}

```

```

printf("\n\n The entered sorted strings in array and counting occurences...\n");
strcpy(tempstr,sarray[0]);
count=0;
j=0;
for(i=0;i<=numstr;i++)
{
printf(" %s",sarray[i]);
if (strcmp(tempstr,sarray[i])==0)
{
count++;
strcpy(sarray2[j].sarr1,tempstr);
sarray2[j].num=count;
}
else if (strcmp(tempstr,sarray[i])!=0)
{
count=1;
j++;
strcpy(tempstr,sarray[i]);
strcpy(sarray2[j].sarr1,tempstr);
sarray2[j].num=count;
}
}

```

```

}
}

count=j;

for(i=0;i<=count-1;i++)
for(j=i+1;j<=count;j++)
{
if (sarray2[i].num<sarray2[j].num)
{
strcpy(tempstr,sarray2[i].sarr1);
strcpy(sarray2[i].sarr1,sarray2[j].sarr1);
strcpy(sarray2[j].sarr1,tempstr);

tempnum=sarray2[i].num;
sarray2[i].num=sarray2[j].num;
sarray2[j].num=tempnum;

}
}

printf("\n\n The strings and occurrences ");

for(i=0;i<=count;i++)
{
printf("\n%s = %d ",sarray2[i].sarr1,sarray2[i].num);
}

printf("\n\n Press any key ... ");
getche();
}

/*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.

```
*/
```

```
#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;
}

```

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;
}

```

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;
}

```

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;
}

```

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 embarrassing, 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;

*pnnumber = 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

```

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("%s correct (%d).\n", pos == test[i].expected ? "C" : "Inc", pos);
```

```
        if(pos != -1)  
        {  
            printf("Character found was %c\n", test[i].data[pos]);  
        }  
    }
```

```
    return 0;  
}
```

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;
    }
}

```

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;
    }
}

```

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 $-(2 \text{ to the power } (\text{wordsize} - 1))$. 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 representation 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. -1 is represented by $(2^n - 1) - (+1)$). 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 : 11111111
 One's complement: 00000000
 Subtract 1 : 11111111

Therefore, because $(n \neq 10)$ 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 $((n \neq 10) > 0)$ ' to 'while $(n \neq 10)$ '. Since any fractional part is truncated with integer division, n will eventually equal zero after successive divides by 10, and 'n $\neq 10$ ' 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;
    }
```

```
}  
}
```

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+1];
    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+1];
}
break;
}
}
s1[j] = s2[i]; /* Don't forget the null character */
}

```

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[50];  
  
    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 */
}

```

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;
}

```

```

/* 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;
}

```

Exercise 2-10. 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 = "AaBbcCD3EdFGHgIJKLhM2NOjPQRkSTIUVWfXYf0Z1";
    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

```

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;
}

```

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.
                     * (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;
}

```

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;
}

```

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.

```

/* 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)strtol(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;
}

```

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      (char)' '
#define TAB        (char)'\t'

int CalculateNumberOfSpaces(int Offset, int TabSize)
{
    return TabSize - (Offset % TabSize);
}

int main(void)
{
    char Buffer[MAX_BUFFER];
    int TabSize = 5; /* A good test value */
}

```

```

int i, j, k;

while(fgets(Buffer, sizeof(Buffer), stdin))
{
    for(i = 0; Buffer[i] != '\0'; i++)
    {
        if(Buffer[i] == TAB)
        {
            j = CalculateNumberOfSpaces(i, TabSize);
            for(k = 0; k < j; k++)
            {
                putchar(SPACE);
            }
        }
        else
        {
            putchar(Buffer[i]);
        }
    }
}

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.

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);
    }
    fflush(stdout);
    return 0;
}

```

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

```

#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;
}

```

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;
}

```

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;
    }
}
```

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;
}
```

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 ** ** * * * * * ***

```


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.

```
#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;

    for(thisidx = 0; thisidx <= MAXWORDLEN; thisidx++)
    {
        lengtharr[thisidx] = 0;
    }

    while((c = getchar()) != EOF)
    {
        if(c == ' ' || c == '\t' || c == '\n')
        {
            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;
        }
    }
}
}
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:

```
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

```

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;
}

```

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.

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.

```
#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;
}
```

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;  
}
```

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

```
#include <stdio.h>
```

```
int main(void)
```

```
{  
  int blanks, tabs, newlines;  
  int c;
```

```
  blanks = 0;  
  tabs = 0;  
  newlines = 0;
```

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

```

{
  if(c == ' ')
    ++blanks;

  if(c == '\t')
    ++tabs;

  if(c == '\n')
    ++newlines;
}
}

printf("Blanks: %d\nTabs: %d\nLines: %d\n", blanks, tabs, newlines);
return 0;
}

```

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;
}

```

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;
}

```

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;
}

```

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;
}

```

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;
}

```

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. 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;
}

```

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.

Here's a list of simple compile lines for a variety of popular compilers:

GNU C

```
gcc -W -Wall -ansi -pedantic -o hello hello.c
```

Microsoft C, up to version 5.1

```
cl -W3 -Zi -Od -AL hello.c
```

Microsoft C, version 6, Microsoft Visual C++ 1.0, 1.5

```
cl -W4 -Zi -Od -AL hello.c
```

Microsoft Visual C++ 2.0 and later

```
cl -W4 -Zi -Od hello.c
```

Turbo C++

```
tcc -A -ml hello.c (I think)
```

Borland C++, 16 bit versions

```
bcc -A -ml hello.c
```

Borland C++, 32 bit versions

```
bcc32 -A hello.c
```