Erik Hammar & Markus Beronius

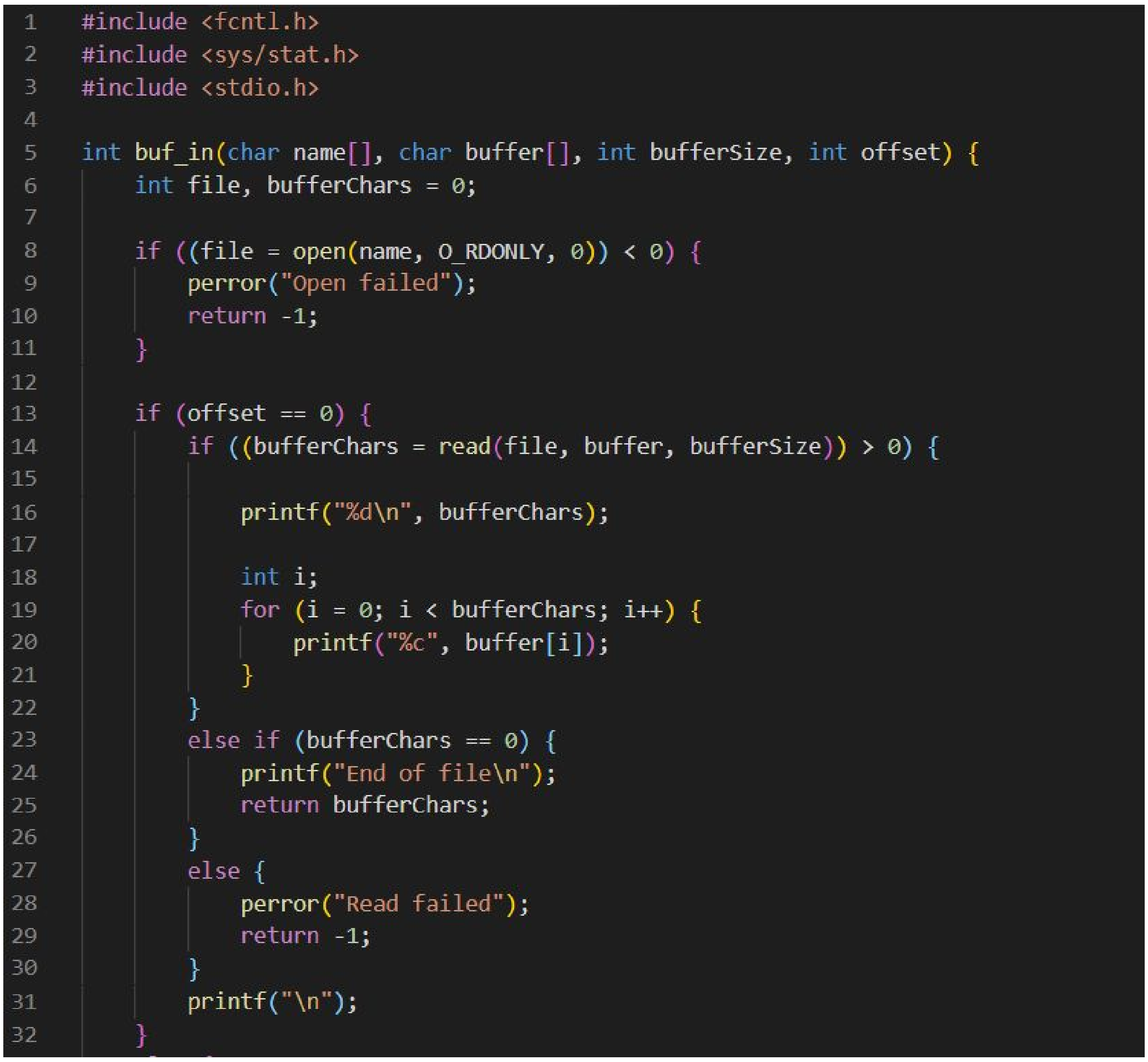
Input-output operations and buffers

The task at hand was to create create a character buffer with size b = 16 bytes, and functions to read and write b bytes at a time between two different files. The buffer was implemented to mimic a bus between two memory locations. Part 1 was to code the function which reads b bytes from a file and stores them in the buffer and we called the function *buf_in*. Part 2 was to implement two functions. One which writes the b bytes in the buffer into a separate file, called *buf_out*. The other function was to flush the remaining bytes in the buffer, for example if the buffer was not filled fully, into the file, called *buf_flush*. Following are code snippets and relevant descriptions of the integral parts of the functions.

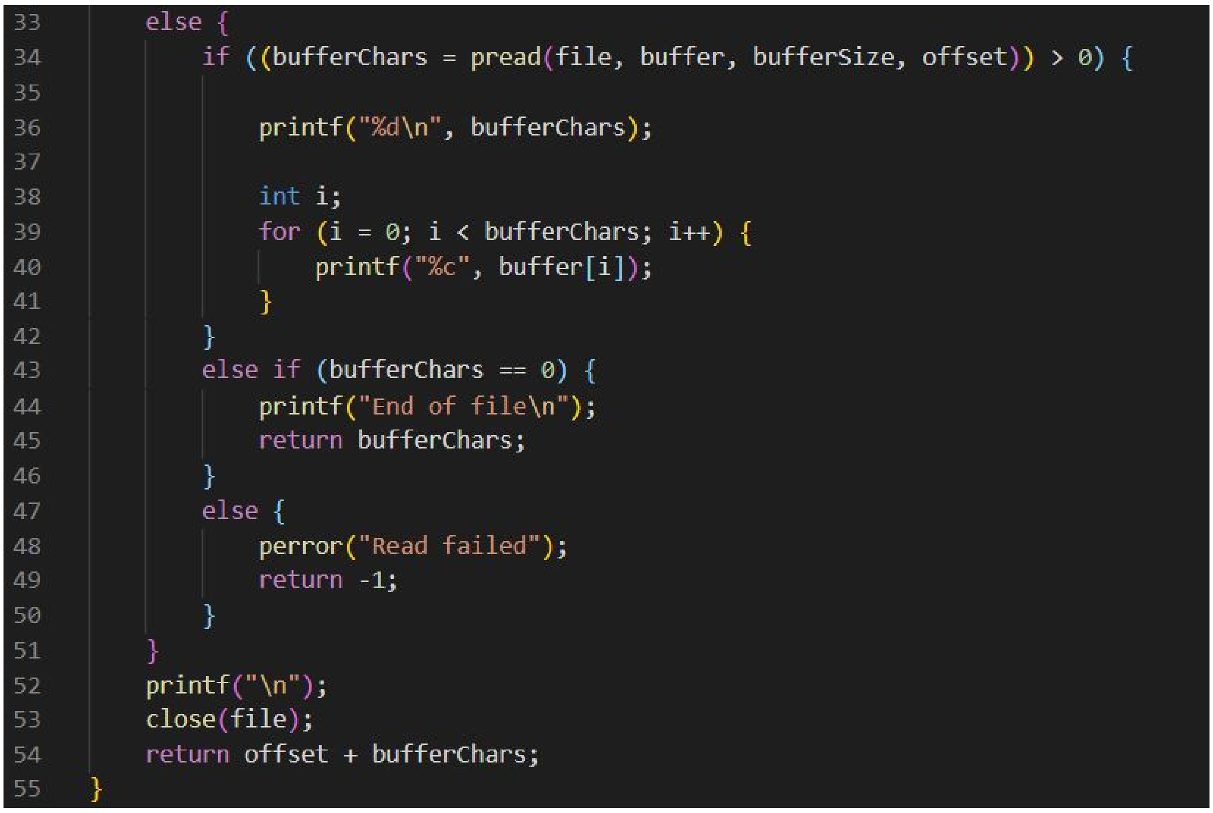
buf_in

@input *name[]*: Source file to read
buffer[]: char array to store bytes in after being read
bufferSize: int determining how many bytes *read* should read.
offset: int determining from what byte *read* should start reading from
@return Value to be used as next iterations *offset*, or amount of bytes read in the last iteration.

Line 8: Opens text file to read.
Line 13: Check to see if it has read the file previously (and shouldn't start from the beginning).
Line 14: If the file has not been read yet, reads the first 16 bytes (from *bufferSize*) and stores the data in the buffer.
NOTE: *read* (and *pread*) returns number of bytes read. On line 16 we print the value for easier troubleshooting.
Line 23: *bufferChars* = 0 means that *read* no bytes, i.e. the file was empty. (*printf* in this statement should say something like "file was empty", rather than "End of file" for more clarity)
Line 33: If the file has been read previously (*if (offset* == *0*) fails) the program enters here.
Line 34: *pread* is identical to *read* with the exception that it takes in a fourth argument (offset) telling it from what byte to start reading the file.
Line 43: Same as line 23, except this time it means no new bytes were read and the program has reached the end of the file.
Line 54: Return integer to represent the next iterations offset. E.g. if after the first iteration and 16 bytes were read, the next iteration would have an offset of (0 + 16), meaning the program would read the file starting with the 17th byte.



```c
#include <fcntl.h>
#include <sys/stat.h>
#include <stdio.h>

int buf_in(char name[], char buffer[], int bufferSize, int offset) {
    int file, bufferChars = 0;

    if ((file = open(name, O_RDONLY, 0)) < 0) {
        perror("Open failed");
        return -1;
    }

    if (offset == 0) {
        if ((bufferChars = read(file, buffer, bufferSize)) > 0) {

            printf("%d\n", bufferChars);

            int i;
            for (i = 0; i < bufferChars; i++) {
                printf("%c", buffer[i]);
            }
        }
        else if (bufferChars == 0) {
            printf("End of file\n");
            return bufferChars;
        }
        else {
            perror("Read failed");
            return -1;
        }
        printf("\n");
    }
```

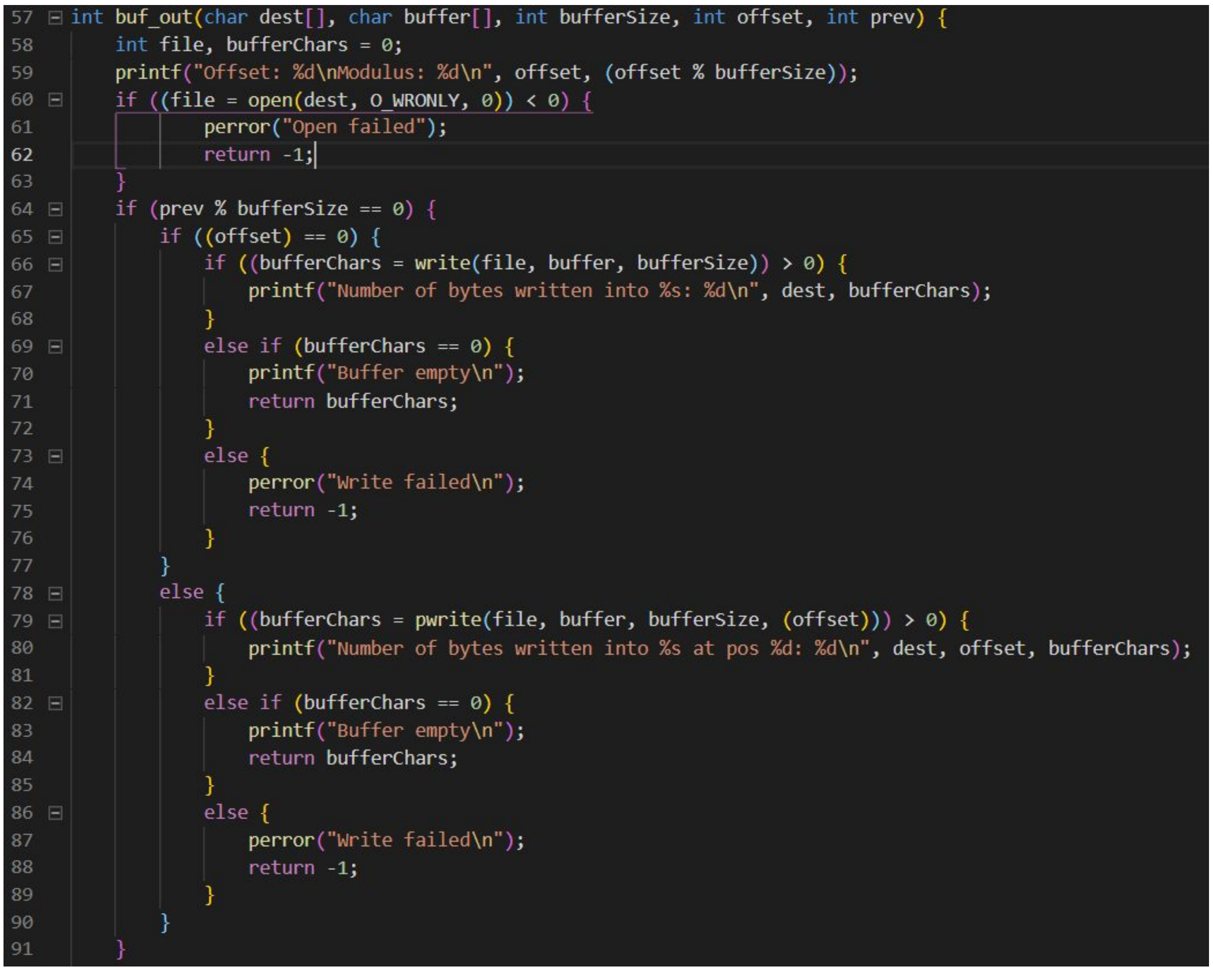


```c
    else {
        if ((bufferChars = pread(file, buffer, bufferSize, offset)) > 0) {

            printf("%d\n", bufferChars);

            int i;
            for (i = 0; i < bufferChars; i++) {
                printf("%c", buffer[i]);
            }
        }
        else if (bufferChars == 0) {
            printf("End of file\n");
            return bufferChars;
        }
        else {
            perror("Read failed");
            return -1;
        }
    }
    printf("\n");
    close(file);
    return offset + bufferChars;
}
```

buf_out

@input *name[]*: Source file to written to
buffer[]: char array where bytes to be written are stored
bufferSize: int determining how many bytes *write* should write.
offset: int determining from what byte *write* should start writing from
prev: Inte used in *buf_flush* to help determine how many bytes to be flushed
@return Value to be used as next iterations *offset*, or amount of bytes read in the last iteration.

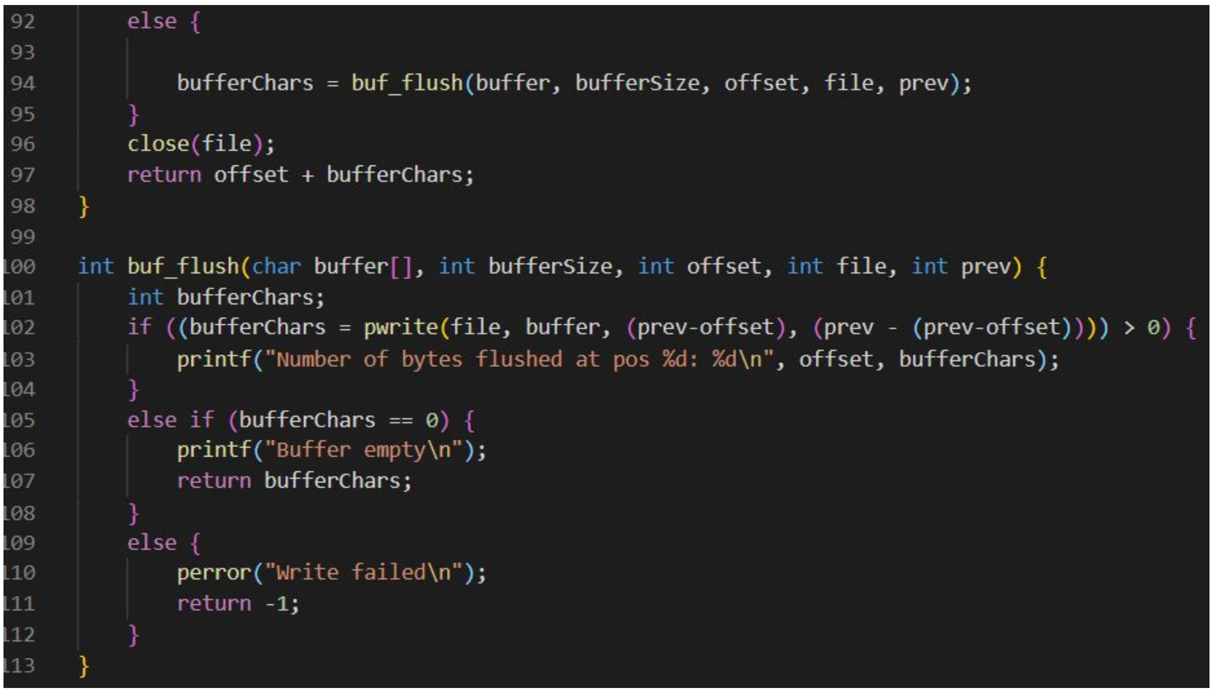
Line 60: Opens text file to write into.
Line 64: Checks if the number of new bytes in the buffer is b = 16 by comparing modulus of *bufferSize* and number of bytes read previously to zero. If it is zero, i.e. the buffer is full, we write the entire buffer into the text file. If it is not zero then it goes down to *buf_flush*.
Line 65: Checks if the file is empty, if it is empty then it writes into position 0 in the text file.
Line 66: Uses system function *write* to write out the buffer contents into the text file.
Line 69 - 77: Error handling
Line 79: Writes b bytes from *buffer* into the text file at specific position.
Line 82 - 88: Error handling
Line 94: If the buffer is not filled completely with new bytes then *buf_flush* flushes those bytes into the text file.



```c
57 int buf_out(char dest[], char buffer[], int bufferSize, int offset, int prev) {
58     int file, bufferChars = 0;
59     printf("Offset: %d\nModulus: %d\n", offset, (offset % bufferSize));
60     if ((file = open(dest, O_WRONLY, 0)) < 0) {
61         perror("Open failed");
62         return -1;
63     }
64     if (prev % bufferSize == 0) {
65         if ((offset) == 0) {
66             if ((bufferChars = write(file, buffer, bufferSize)) > 0) {
67                 printf("Number of bytes written into %s: %d\n", dest, bufferChars);
68             }
69             else if (bufferChars == 0) {
70                 printf("Buffer empty\n");
71                 return bufferChars;
72             }
73             else {
74                 perror("Write failed\n");
75                 return -1;
76             }
77         }
78         else {
79             if ((bufferChars = pwrite(file, buffer, bufferSize, (offset))) > 0) {
80                 printf("Number of bytes written into %s at pos %d: %d\n", dest, offset, bufferChars);
81             }
82             else if (bufferChars == 0) {
83                 printf("Buffer empty\n");
84                 return bufferChars;
85             }
86             else {
87                 perror("Write failed\n");
88                 return -1;
89             }
90         }
91     }
```

buf_flush

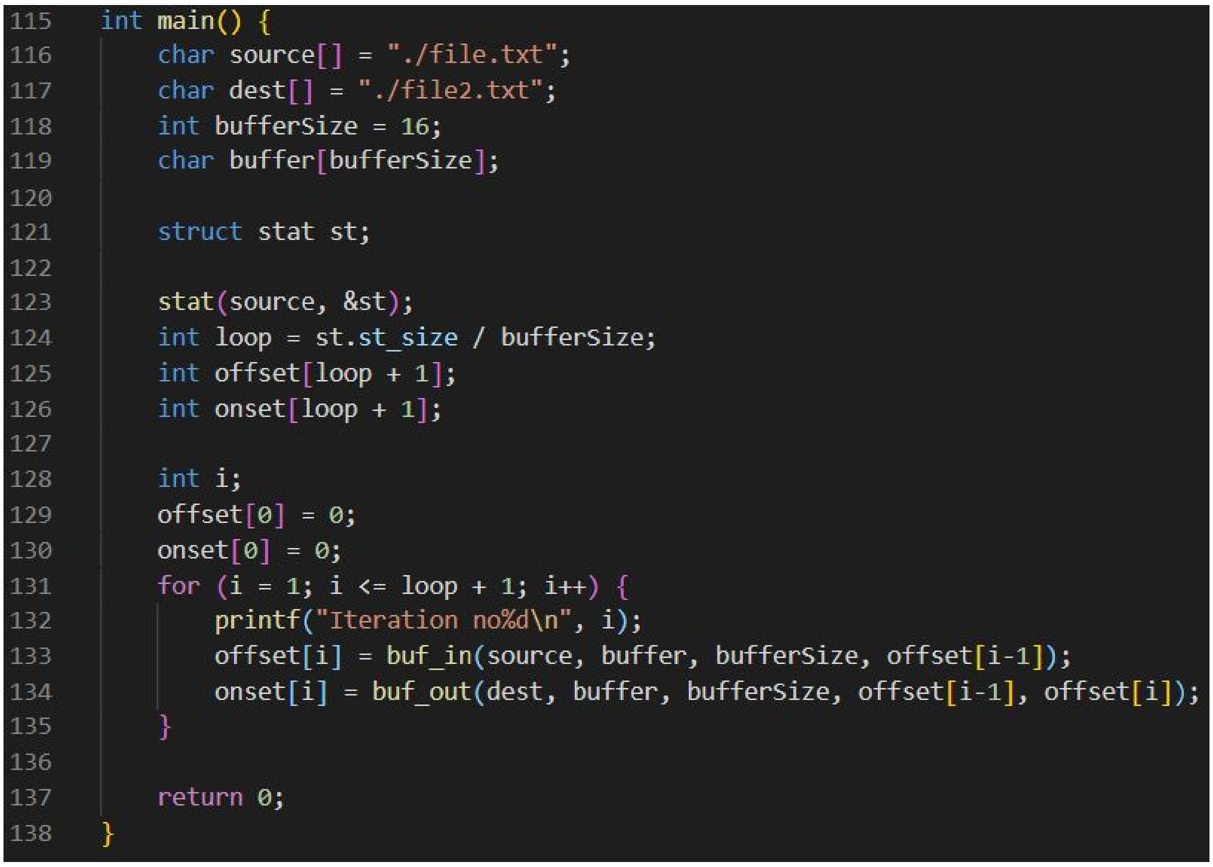
Line 102: Uses system call *pwrite* to flush the rest of the bytes. We calculate how many bytes to write and where in the text file to write them.



```
 92      else {
 93
 94          bufferChars = buf_flush(buffer, bufferSize, offset, file, prev);
 95      }
 96      close(file);
 97      return offset + bufferChars;
 98  }
 99
100  int buf_flush(char buffer[], int bufferSize, int offset, int file, int prev) {
101      int bufferChars;
102      if ((bufferChars = pwrite(file, buffer, (prev-offset), (prev - (prev-offset)))) > 0) {
103          printf("Number of bytes flushed at pos %d: %d\n", offset, bufferChars);
104      }
105      else if (bufferChars == 0) {
106          printf("Buffer empty\n");
107          return bufferChars;
108      }
109      else {
110          perror("Write failed\n");
111          return -1;
112      }
113  }
```

main

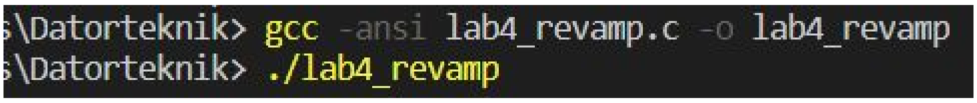
Line 121 & 123: Create a struct to be able to get the size of the text file.
Line 124: Uses the size of the text file divided by b to get the number of loops needed to get the entire text file.
Line 125: Create an array with the size equal to the number of buffer loads needed to read the text file.
131: A for-loop iterates the number of times needed to read the entire text file. In each iteration it calls *buf_in* and reads b bytes from text file1, and *buf_out* and writes out those bytes into text file2.



```
115  int main() {
116      char source[] = "./file.txt";
117      char dest[] = "./file2.txt";
118      int bufferSize = 16;
119      char buffer[bufferSize];
120  
121      struct stat st;
122  
123      stat(source, &st);
124      int loop = st.st_size / bufferSize;
125      int offset[loop + 1];
126      int onset[loop + 1];
127  
128      int i;
129      offset[0] = 0;
130      onset[0] = 0;
131      for (i = 1; i <= loop + 1; i++) {
132          printf("Iteration no%d\n", i);
133          offset[i] = buf_in(source, buffer, bufferSize, offset[i-1]);
134          onset[i] = buf_out(dest, buffer, bufferSize, offset[i-1], offset[i]);
135      }
136  
137      return 0;
138  }
```

Terminal

In the command terminal we use *gcc -ansi* to compile with the ansi flag, and then *./* to run the object file.



```
s\Datorteknik> gcc -ansi lab4_revamp.c -o lab4_revamp
s\Datorteknik> ./lab4_revamp
```