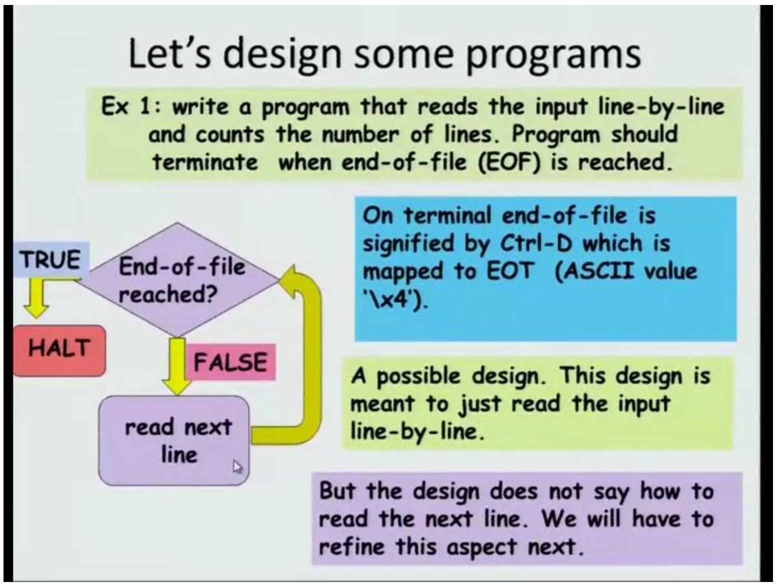
Introduction to Programming in C

Prof. Satyadev Nandakumar
Department of Computer Science and Engineering
Indian Institute of Technology, Kanpur

Lecture - 28

(Refer Slide Time: 00:14)



Let's design some programs

Ex 1: write a program that reads the input line-by-line and counts the number of lines. Program should terminate when end-of-file (EOF) is reached.

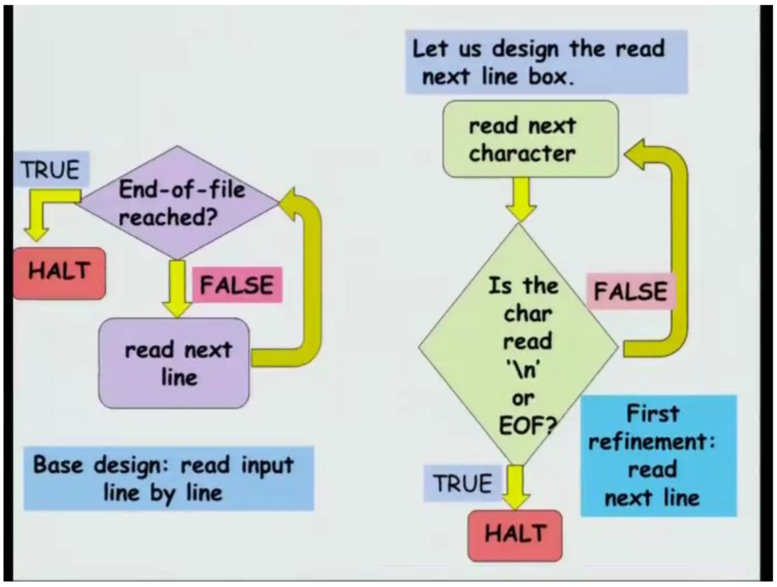
On terminal end-of-file is signified by Ctrl-D which is mapped to EOT (ASCII value '\x4').

A possible design. This design is meant to just read the input line-by-line.

But the design does not say how to read the next line. We will have to refine this aspect next.

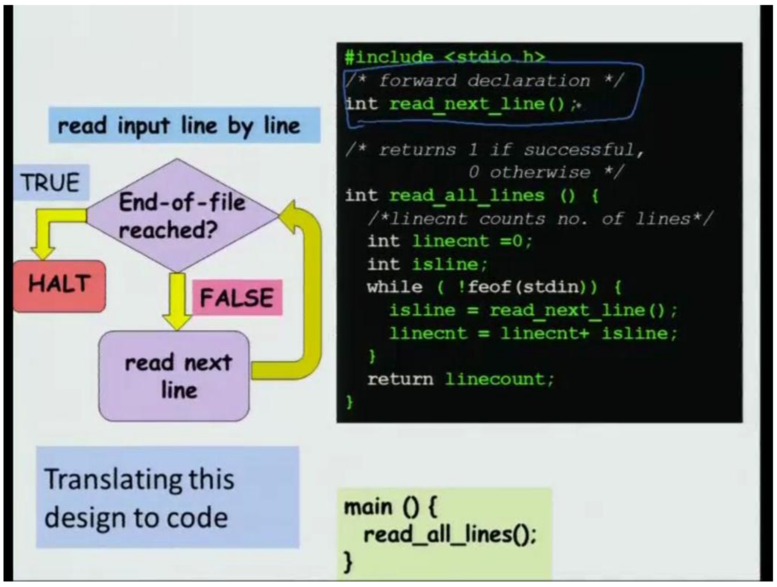
With the concepts we have seen so far, let us design a sample program. So, the… What we have… The problem that we want to solve is we want to write a program that reads the input line-by-line, and counts how many lines has the user input. Program should terminate when the end-of-file character is encountered. So, we will try to solve this problem. By the way, the end of file is a character, which you can enter using control-D if you are running Linux. So, the flowchart at the very top level can be envisioned as follows. So, we will just check has the end-of-file been reached. If the end-of-file has not been reached, you read the next line. If it is has been and check again. If the end-of-file has been reached, then you halt; otherwise you read another line. So, here is the very top-level picture of what we want to do. So, this design is just meant to read the input line-by-line. So, it is a very vague flowchart, but at the top level, this is what we want to do. So, let us say more details about how we are going to accomplish this. In particular, we want to see how we can read and put line-by-line.

(Refer Slide Time: 01:33)


Let us design the read next line box.
read next character
TRUE
End-of-file reached?
HALT
FALSE
read next line
Is the char read '\n' or EOF?
FALSE
First refinement: read next line
Base design: read input line by line
TRUE
HALT

So, here is the top-level design. And now we are going to essentially expand this box. We want to say how do we read the next line. So, let us design the read next line box. So, the read next line box, first you read a character and then you check whether the character read is new line character; that means that the user has pressed an enter. So, the line is ended at that point or the user can enter a bunch of characters; and instead of pressing enter, press control D. So, the user can enter end-of-file. If either of these are true, then the line has ended. So, you halt. Otherwise, if the character is neither new line nor end-of-file, then you read the next character. So, here is the design for the function to read the next character – next line. So, you read character-by-character; after every character, you check whether a new line or an end-of-file has been encountered. If either of them happen, then the line has ended; otherwise, you go back and read another character.

(Refer Slide Time: 02:54)


read input line by line
TRUE
End-of-file reached?
HALT
FALSE
read next line

```
#include <stdio.h>
/* forward declaration */
int read_next_line();

/* returns 1 if successful,
           0 otherwise */
int read_all_lines () {
  /*linecnt counts no. of lines*/
  int linecnt =0;
  int isline;
  while ( !feof(stdin)) {
    isline = read_next_line();
    linecnt = linecnt+ isline;
  }
  return linecount;
}
```

Translating this design to code

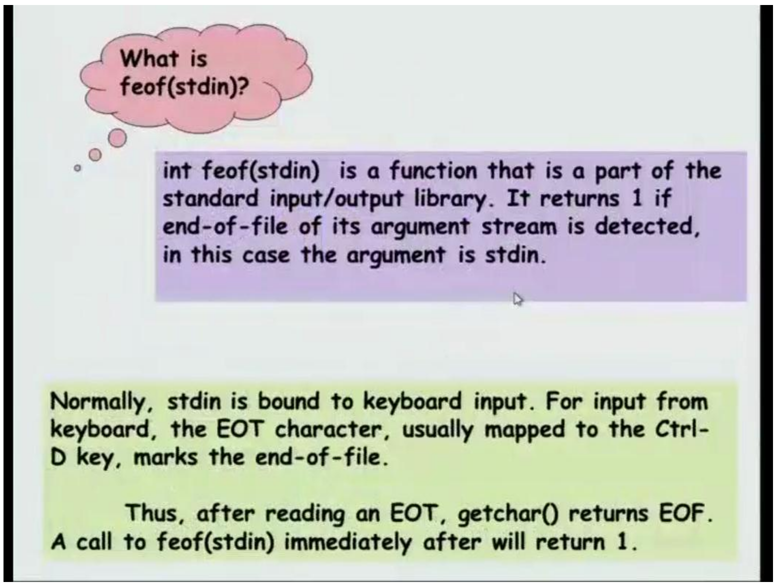
```
main () {
  read_all_lines();
}
```

So, let us start by writing the top-level function. So, let us translate the top-level function into code. So, here we will introduce a new concept called what is known as a forward declaration. So, when you define a function, you can either give the logic – the full function body when you define the function or you can just say that, here is what the function will look like; here is the type signature; basically, it is taking no arguments and it will return an integer value. And I will terminate that statement by using a semicolon; which says that, this function… I will currently just say the type of the function; I will define the function later. This is done, so that we can write a function, which uses this particular function. So, when we write a function, which uses that function, the type of the function should be known. For that we can just declare the type of the function. This is what is known as a declaration of a function.

Unless you define the function, you cannot use it; but in order for another function to just see what the function looks likes, declaration is sufficient. So, let us design the top-level function. So, we declare this function that, we will use in this function that we are about to write. So, this user function will be called read all lines. Now, in that, we will keep a line count initialized to 0; and then I will keep a flag called is line. Now, what this will do is we have to check for whether an end-of-file has been reached or not. For that, I will use the function f e o f s t d i n. We will see that in a minute. While the end-of-file has not been encountered, you say that, read next line; read next line will return a 1 if a line has been encountered; otherwise, it will return a 0. So, line count will be incremented by

1 if I read another line; otherwise, it will remain as it is. Finally, you return the number of lines read. So, this is a realization of the flowchart on the left. Now, there are a couple of things that require explanation. First is that even though the read next line function has not yet been defined, just based on the declaration, I can say that, it is going to return an integer and I can use the integer here.

(Refer Slide Time: 06:09)



What is feof(stdin)?

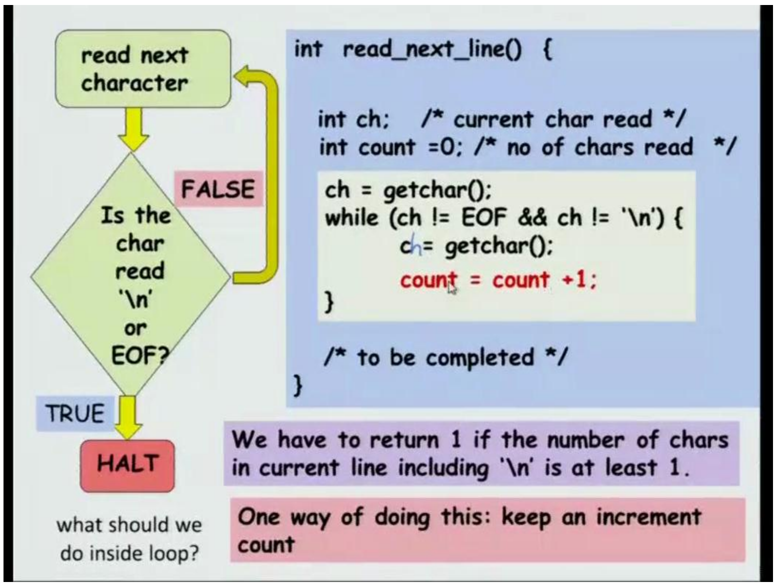
int feof(stdin) is a function that is a part of the standard input/output library. It returns 1 if end-of-file of its argument stream is detected, in this case the argument is stdin.

Normally, stdin is bound to keyboard input. For input from keyboard, the EOT character, usually mapped to the Ctrl-D key, marks the end-of-file.

Thus, after reading an EOT, getchar() returns EOF. A call to feof(stdin) immediately after will return 1.

The other thing is what do we mean by f e o f s t d i n? So, what do we mean by the function f e o f? So, f e o f s t d i n is a function that is part of the s t d i o library. We have already used other functions from that library. For example, print f and scan f. Now, the f e o f function – what it does is – it returns a function; it returns a value 1 if the end-of-file has been encountered in the input argument. So, s t d i n means that, I am using the standard input, which is the keyboard input. So, if and end-of-file has been entered via the keyboard, then f e o f s t d i n will return 1. So, s t d i n is usually the keyboard input. And usually, if the user enters the control D character, then f e o f will say 1, because end-of-file has been entered.

(Refer Slide Time: 07:13)



```
int  read_next_line()  {

   int ch;   /* current char read */
   int count =0; /* no of chars read  */
    ch = getchar();
    while (ch != EOF && ch != '\n') {
            ch= getchar();
            count = count +1;
    }

    /* to be completed */
}
```

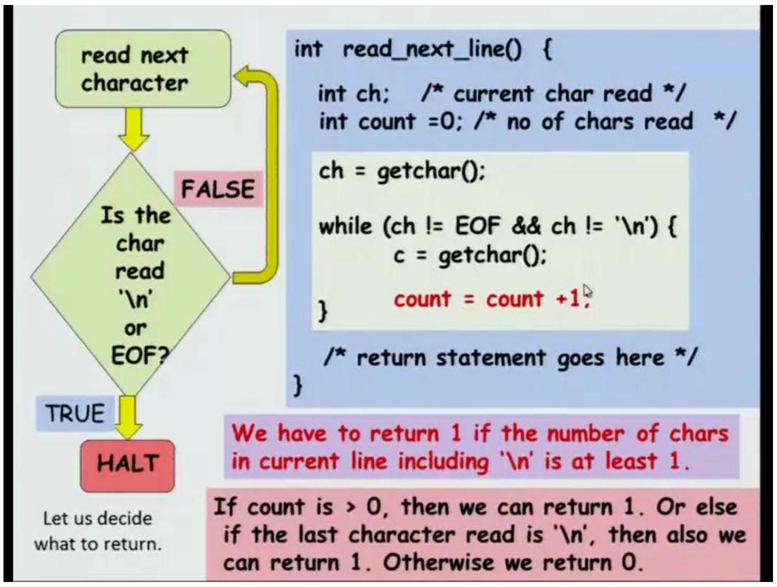
We have to return 1 if the number of chars in current line including '\n' is at least 1.

One way of doing this: keep an increment count

Now, let us design the function to read a line. We earlier wrote a function, which assumed that, there is a function, which will read the next line; and based on that, I will keep on reading lines until the end-of-file is encountered. So, we are now about to write the bottom function. So, we want to read a line. So, we have already drawn the flowchart for that. Now, let us try to make it into code. So, we have to design a few variables; we will have int ch for reading a character; we will come to that in a minute; then we will keep a count of how many characters have been read. And let us write the basic loop. So, we will just write the loop corresponding to the flowchart; ch will be getchar. So, get the next character. And while ch is… While the read character is neither end-of-file nor new line, you should keep reading characters. So, if neither of this is true, then you should read the next character, which is what the flowchart says. A slight… A small point here is that, getchar returns an integer. This is a technicality because end-of-file is negative 1.

ASCII characters if you remember, go from 0 until 127 or something like that; whereas, end-of-file is defined to be minus 1. So, because of this minus 1, you cannot keep the return value of getchar as a character; it technically has to be an integer. Now, this is a technicality. So, keep that in mind. Now, we need to do something further in the loop. So, we will complete this in a minute. So, what should we do inside the loop? This should be character. So, what is this function supposed to do overall? We have to return a 1, if the number of characters in the current line that we have read is at least 1. So, if the current line contains at least a character, then we have to return 1. For example, if the user just entered a new line, which is just press the enter key, there is a blank line. In that case, we would not say that, we have read a line, because it was a blank line. So, if there is at least one character, which is neither new line nor end-of-file in that line, we have to return a 1; otherwise, let us say we return a 0. So, one way to do that is to keep a count of the number of the characters we have read. So, for every character read, we will keep a count of every character, which is neither end-of-file nor a new line; we will keep a count of characters. So, notice the way that, the loop has been return. So, if the first character is a new line, it will not enter the loop. Hence, count remains 0. At the same time, the way the loop is returned; count will count exactly those characters, which are neither new line nor end-of-file.

(Refer Slide Time: 11:04)



```
int  read_next_line()  {
   int ch;    /* current char read */
   int count =0; /* no of chars read  */
   ch = getchar();

   while (ch != EOF && ch != '\n') {
         c = getchar();
         count = count +1;
   }
    /* return statement goes here */
}
```

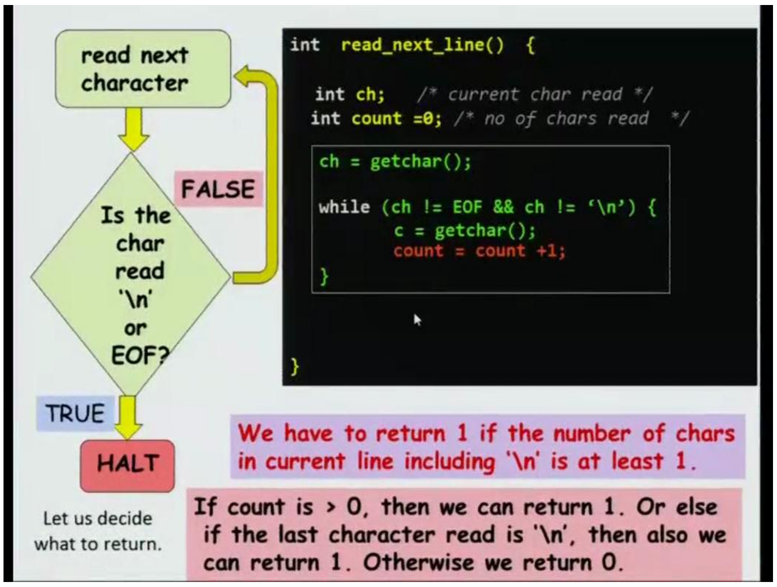
We have to return 1 if the number of chars in current line including '\n' is at least 1.

Let us decide what to return.

If count is > 0, then we can return 1. Or else if the last character read is '\n', then also we can return 1. Otherwise we return 0.

So, now, let us decide what should be the return value. We have to return a 1 if the number of characters in the current line including new line is at least 1. So, if count is greater than 0, we can return a 1. If exactly 1; if the last character was end-of-file without having any other characters, then will return a 0. So, how we do that? We can check whether at least a character has been read by just checking the value of count.

(Refer Slide Time: 11:45)



```
int  read_next_line()  {

  int ch;    /* current char read */
  int count =0; /* no of chars read  */

  ch = getchar();

  while (ch != EOF && ch != '\n') {
          c = getchar();
          count = count +1;
  }

}
```

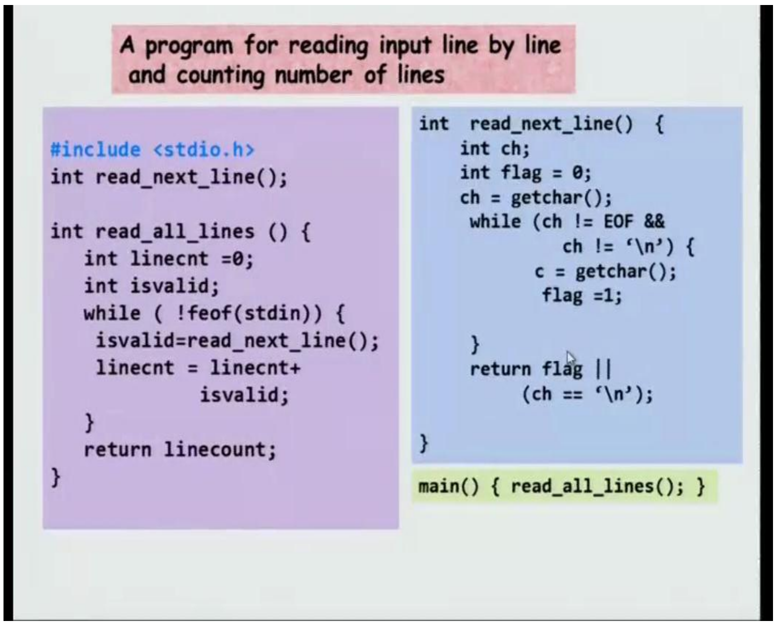
We have to return 1 if the number of chars in current line including '\n' is at least 1.

Let us decide what to return.

If count is > 0, then we can return 1. Or else if the last character read is '\n', then also we can return 1. Otherwise we return 0.

So, if count is greater than 0, then at least one character has been entered; otherwise, for example, we can also say that, if the user has just entered a blank line, then also we can say that, one more line has been entered. So, that is up to the way you want to do it; you can also take the stance that, maybe a blank line does not count as a line. If that is the case, then you do not have to do it; but in this case, let us just assume that, if at least a character has been entered, which is either a normal character on a new line, we will say that, return 1. If the only character entered in that line is end-of-file, we will say that, there is no more new line. So, what we have to do is return count greater than 0; this tells you how many non-new-line, non-end-of-file characters have been entered. So, this should be at least 1; or, there is exactly one character entered, which is a new line. So, neither these cases we will return a 1; otherwise, we will return a 0.

(Refer Slide Time: 13:03)



A program for reading input line by line and counting number of lines

```
#include <stdio.h>
int read_next_line();

int read_all_lines () {
   int linecnt =0;
   int isvalid;
   while ( !feof(stdin)) {
    isvalid=read_next_line();
    linecnt = linecnt+
             isvalid;
   }
   return linecount;
}
```

```
int  read_next_line()  {
    int ch;
    int flag = 0;
    ch = getchar();
     while (ch != EOF &&
             ch != '\n') {
           c = getchar();
            flag =1;

     }
     return flag ||
          (ch == '\n');

}
```

```
main() { read_all_lines(); }
```

So, we can put these programs together by concatenating all the code that we have written. Notice one thing that declare the function first; we use the function here. So, here is a top-level function, which will use read next line. When read all lines uses read next line; read next line has not been defined yet. So, you can go here after read all lines has been defined, you can define read next line. So, here is a function here. So, this is function 1, this is function 2, and finally you have made. Read all lines does not need any forward declaration, because when main uses read all lines, it has already been defined. That was not the case here. When read all lines used to read next line, read next line was not defined yet. That is why we needed a forward declaration. In this program, you can reorder the code such that read next line code can be written before in which case you do not need the forward declaration. But the concept of forward declaration is useful for later discussion. So, I have just introduced that.