**Character input and output**

A character in programming refers to a single symbol – a smallest possible string. The characters in ‘C’ have no relationship with characters in the plays of Shakespeare even though those characters utter characters we could process in programming!

A character in ‘C’ is like a very small integer having one of the 256 possible values. It occupies a single byte. We code English alphabets A as 65, B as 66, Z as 90, a as 97 and so on in a coding scheme called ASCII. We have no concept of character type in Python – everything is a string there.

For the first unit, we are required to find the number of characters, number of words and number of lines in a given file. We want to simulate a program in Unix called wc – word count program.

We have not learnt as yet how to play with files in our programs. It is a bit too early in this course. We do know how to read from the keyboard. Can the operating system open a file for me and make it logically available on the keyboard? It can. This concept is called input redirection.

When we run our command cmd from the unix shell and specify <filename on the same line, our program cmd reads from the file whenever it reads from the keyboard.

**input redirection:**

$ cmd <filename

It is similarly possible to collect the output of the program which would appear on the screen in a file using output redirection.

**output redirection:**

$ cmd >filename

To complete this project, we should learn

a) how to read and display a character?

b) how to read a line?

c) how to make out when we reach the end of file?

d) how to break a given sequence of characters into words?

Let us check the program : ex1.c

To read a character from the keyboard, we could use scanf(“%c”, &x);

We could also use x = getchar();

We prefer the second as it is developed only for handling a single char and therefore more efficient even though it is not generic.

Similarly we prefer putchar(x) over printf(‘%c”, x);

This code shows how to read two characters and display them.

#if 0

scanf("%c", &x);

scanf("%c", &y);

printf("x:%c y:%c\n", x, y);

#endif

x = getchar();

y = getchar();

putchar(x);

putchar(y); printf("%d", y);

Let us try to read a line and display.

Let us examine the program ex2.c.

char ch;

while(ch != '\n')

{

ch = getchar();

putchar(ch);

}

This is a terrible code. The first time we enter the loop, ch is not initialized. If you are lucky, the loop is never entered!

In these cases, we require reading before the loop and reading at the end of the body of the loop.

ch = getchar();

while(ch != '\n')

{

putchar(ch);

}

Infinite loop unless the first character is new line!!

ch = getchar();

while(ch != '\n')

{

putchar(ch);

ch = getchar();

}

This is fine. But we can as well use assignment expression in the while.

This is ‘C’ code!!

while( (ch = getchar()) != '\n')

{

putchar(ch);

}

putchar('\n');

A few points to observe. The input from the file or the keyboard is stored in something called the buffer. That is transferred to getchar only afte enter key is pressed.

The next hurdle is to read the whole file. How do we know we have reached the end of file. ‘C’ handles this in a very interesting way. When the end of file is reached – which the operating system can make out – a particular value gets is returned by getchar. This value is decided by ‘C’ and not by the operating system. This value is associated with the name EOF. So, we keep reading until getchar returns EOF.

// file: ex3.c

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

{

putchar(ch);

}

Let us examine ex4.c – the first attempt to count the number of lines and number of characters.

char ch;

int nl = 0;

int nw = 0;

int nc = 0;

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

{

++nc;

while((ch = getchar()) != '\n')

{

++nc;;

}

++nl; ++nc;

}

//putchar('\n');

printf("# of char : %d\n", nc);

printf("# of lines : %d\n", nl);

This could work but not preferred. Observe that we are taking input in two places. What if we reach EOF in the inner loop?

It is always preferred to read at only one place.

It is always preferred to have a while with an it over

while with a while within.

This program ex5.c uses the following logic.

Count the character each time we enter the loop.

Count the lines if the character is newline.

Count the words assuming that the words on a newline or space or tab – any white space.

char ch;

int nl = 0;

int nw = 0;

int nc = 0;

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

{

++nc;

if(ch == '\n')

{

++nl;

}

if(ch == ' ' || ch == '\n' || ch == '\t')

{

++nw;

}

}

//putchar('\n');

printf("# of char : %d\n", nc);

printf("# of words : %d\n", nw);

printf("# of lines : %d\n", nl);

What if there are multiple spaces between the words? Then we are in for trouble.

Can we use a boolean variable to indicate whether we are in a word or not. If we are in a word and we reach a space, the word ends and we count. If we are not in a word and we encounter a space, we ignore it. We set in word when we encounter a non- white-space.

Let us have a look at the program ex6.c.

int inword = 0; // not in a word so far

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

{

++nc;

if(ch == '\n')

{

++nl;

}

if(inword && (ch == ' '|| ch == '\t' || ch == '\n'))

{

inword = 0; ++nw;

}

// avoid recomputation of white space concept

else if (!(ch == ' '|| ch == '\t' || ch == '\n'))

{

inword = 1;

}

}

You may compile this and run this against any text file say myfile.txt.

# ./a.out < myfile.txt

Compare with

# wc myfile.txt

Both should give the same result.

I have tried on the collection of Shakespeare’s works.

$ wc s.txt

124210 899680 5447737 s.txt

$ gcc ex6.c

$ ./a.out <s.txt

# of char : 5447737

# of words : 899680

# of lines : 124210

Here we end the unit I.

Wish you happy learning ‘C’.