LESSON 4

This lesson introduces some of the basic ideas involved in character processing. The lesson discusses how characters are stored and manipulated by the C language, how characters can be treated as small integers, and how use is made of certain standard header files. To illustrate the ideas, the lesson presents simple character processing programs that accomplish useful examples. These example programs make use of the character input/output functions getchar( ) and putchar( ). For a programmer trying to master a computer language, getting data into and out of a computer is a skill that has to be developed quickly.

A number of important concepts are covered in this lesson. The use of the symbolic constant end-of-file (EOF) is explained. When getchar( ) detects an end-of-file mark, it returns the value EOF, which makes it possible for the programmer to detect when the end of a file has been reached. The use of the header file ctype.h is explained. This file provides the programmer with a set of macros that can be used to process character data. To the programmer, these macros are used in the same manner as functions are used. The use of system header files such as stdio.h allows the programmer to write portable code. The programs presented in this lesson are quite simple, and are used to explain the essential ideas of character processing.

The DATA TYPE char

The data type char is one of the fundamental types of the C language. Constants and variables of this type are used to represent characters. Each character is stored in a computer in 1 byte. A byte is composed of 8 bits. A byte composed of 8 bits is capable of storing 2^8 or 256 distinct values.

When a character is stored in a byte, the contents of that byte can be thought of as either a character or as a small integer. Although 256 distinct values can be stored in a byte, only a subset of these values represent actual printing characters. These include the lowercase letters, uppercase letters, digits, punctuation, and special characters such as + - * and %. The character set also includes the white space characters blank, tab, and newline. Examples of nonprinting characters are newline and bell. The example will illustrate the use of the bell in this lesson.

A constant of type char is written between single quotes as 'a' or 'b' or 'c'. Remember a single quote for a single character. A typical declaration for a variable of type char is

```
char c;
```

Character variables can be initialized as in the list of the example:

```
char c1 = 'A', c2 = 'B', c3 = '*';
```

A character is stored in memory in one byte according to a specific encoding. The encode in most computers is the ASCII character codes. For any other code, like EBCDIC the numeric value will be different. A character is considered to have the integer value corresponding to its ASCII encoding. There is no particular relationship between the value of the character constant representing a digit and the digit's intrinsic integer value. That is, the value of '7' is not 7. The fact that the values of 'a', 'b', 'c', and so forth, occur in ascending order is an important property of ASCII. It makes it convenient to sort characters, words, lines, etc., into lexicographical order.

In the functions printf( ) and scanf( ) a %c is used to designate the character format. For example, the statement

```
printf("%c", 'a');   /*Causes the character a to be printed */
```

causes the character constant 'a' to be printed in the format of a character. Similarly,

```
printf("%c%c%c", 'A', 'B', 'C');  /* Causes ABC to be printed */
```

causes ABC to be printed.

Constants and variables of type char can also be treated as small integers, since they have an integer ASCII value. The statement

```
printf("%d", 'a');   /*The value 97 is printed */
```

causes the value of the character constant 'a' to be printed in the format of a decimal integer. Thus 97 is printed. On the other hand the statement

```
printf("%c", 97);   /*The character a is printed */
```

causes the value of the decimal integer constant 97 to be printed in the format of a character and a is printed.

Some nonprinting and hard-to-print characters require an escape sequence. For example, the newline character is written as '\n ' in the format statement, and even though it is being described by the two characters \ and n, it represents a single ASCII character. The backslash character \ is also called the escape character and is used to escape the usual meaning of the character that follows it. The following table contains some nonprinting and hard-to-print characters.

Name Of Character	Written In C	Value
null	\0	0
backspace	\b	8
tab	\t	9
newline	\n	10
formfeed	\f	12
carrage return	\r	13
double quote	\"	34
single quote	\'	39
back slash	\\	92

The character " has to be escaped if it is used as a character in a string. An example is

```
printf("\"ABC\"");    /* "ABC" is Printed */
```

Inside single quotes one would write "', although '\"' is also accepted. In general, escaping an ordinary character has no effect. Inside a string the single quote is just an ordinary character.

```
printf("'ABC'"); /* 'ABC' is printed */
```

Another way to write a constant of type char is by means of a one, two, or three-octal-digit escape sequence as in '\007'. This character is the bell and it also can be written as '\07' or '\7', but it cannot be written as '7'.

USING getchar( ) AND putchar( )

The C language provides getchar( ) and putchar( ) for the input and output of

characters. To read a character from the keyboard getchar( ) is used. To write a character to the screen putchar( ) is used. For example, a program that prints the line "Hello C World" on the screen can be written as follows:

```
main( )
{
   putchar( 'H' );
   putchar( 'e');
   putchar( 'l' );
   putchar( 'l' );
   putchar( 'o' );
   putchar( ' ' );
   putchar( 'C' );
   putchar( ' ' );
   putchar( 'W' );
   putchar( 'o' );
   putchar( 'r' );
   putchar( 'l' );
   putchar( 'd' );
   putchar( '\n' )
}
```

This is a tedious way to accomplish the task; the use of a printf( ) statement would be much easier. In the next program getchar( ) gets a character from the input stream (keyboard) and assigns it to the variable c.
Then putchar( ) is used to print the character on the screen.

```
main( )
{
   char c;
   while (1) {
      c = getchar( );
   putchar ( c ) ;
}
```

Note that the variable c is of type char. In the next version of this program the type will be changed to int. Also, in the while loop because 1 is nonzero, as an expression it is always true. Thus the while construct

```
while (1) {



}
```

is an infinite loop.

The way to stop this program is with an interrupt, which on our system is effected by typing a control-c. However, since an interrupt is system dependent, you may have to type something different to effect it. The "delete" key is sometimes used to effect an interrupt.

This is not an acceptable program and will be rewritten in the next example.

```
#include <stdio.h>
main(   )
{
  int c;
  while (( c = getchar(   ) != EOF)  {
     putchar(c);
    putchar(c);
  }
}
```

PROGRAM ANALYSIS

#include <stdio.h>

The # symbol is a special character when in column one. The symbol is a control indicator for the preprocessor. The preprocess is another program that runs before the C compiler and

A control line of the form #include <stdio.h> tells the preprocessor to include a copy of the file named stdio.h (in this case) into the source code at that point it is invoked. This is done before passing the rest of the source code to the C compiler. The left and right chevrons around stdio.h tell the preprocessor to look for this file in the include directory. The include directory is system dependent. The file stdio.h is a standard header file supplied with the C compiler system, and this file is typically included in functions that make use of certain standard input/output constructs. The constructs are called function prototypes.

One line in the stdio.h header file is:

```
#define EOF (-1)
```

The identifier EOF is a mnemonic for "end-of-file." What is actually used to signal an end-of-file mark. The EOF mark is usually system dependent. Although the int value - 1 is often used, different systems can have different values. By including the file stdio.h and using the symbolic constant EOF, the program portable for different systems. This means that the source file can be moved to a different system and run with little or no changes.

```
int c;
```

The variable c has been declared in the program as an int rather than a char. Which character is used to signal the end-of-file, cannot be a value that represents a character. Since c is an int, it can hold all possible character values as well as the special value EOF. Although one usually thinks of a char as a "short int" type, one could also think of an int as a "long char" type.

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

The expression

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

is composed of two parts. The subexpression

```
c = getchar(   )
```

gets a value from the keyboard and assigns it to the variable c, and the value of the sub-expression takes on that value as well. The symbols != represent the "not equal" operator. As long as the value of the sub-expression c = getchar( ) is not equal to EOF, the body of the while loop is executed. Typically, an EOF value can be entered at the keyboard by typing a controi-d in UNIX (control-z in DOS) immediately following a carriage return. Again, this is system dependent. The parentheses around the subexpression c = getchar( ) are necessary. Suppose that we had typed c = getchar( ) != EOF. Because of operator precedence this is equivalent to:

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

This has the effect of getting a character from the input stream, testing to see if it is not equal to EOF, and assigning the result of the test (either 0 or 1) to the variable c.

The char data type have an underlying integer valued representation is the numeric value of their 7-bit ASCII representation. For example, the constant char 'a' has a value 97, 'b' a value of 98. If you thinks of characters as small integers or short int, then arithmetic on character can be developed and performed. The values of the letters in both the lowercase and uppercase alphabet occur in order. The expression 'a' + 1 has the value 'b', the expression 'b' + 1 has the value 'c', and the expression 'z' - 'A' has the value 25. The expression 'A' - 'a' has the same value as 'B' - 'b', which is the same as 'Z' - 'z'. Because of this, if the

variable c has the value of a lowercase letter, then the expression c + 'A' - 'a' has the value of the corresponding uppercase letter. These ideas are incorporated into the next program, which capitalizes all lowercase letters and doubles the newline characters.

```
/*Capitalize lowercase letters and double space output*/
#include <stdio.h>
main(   )
{
  int c;
  while ((c = getchar( )) != EOF)
      if ('a'  <=  c && c  <=  'z' )
         putchar(c + 'A' - 'a');
     else
       if (c == '\n')  {
          putchar( '\n' ) ',
          putchar( '\n' ) ',
       }
      else
          putchar(c);
}
```

PROGRAM ANALYSIS

The expression:

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

The function getchar( ) gets a character and assigns it to the integer variable c. As long as the value of c is not equal to EOF, the body of the while loop is executed.

In the if expression:

```
if ('a'  <=  c && c  <=  'z' )
        putchar(c + 'A' - 'a');
```

the order of the operator precedence in the expressions

```
'a' <= c && c <= 'z'
```

and

```
('a' < = c) && (c <= 'z')
```

are equivalent. The symbols <= represent the operator "less than or equal."

The sub-expression 'a' <= c tests to see if the value of the character 'a' is less than or equal to the value stored in the variable c. The sub-expression c <= 'z' tests to see if the value stored in the variable c is less than or equal to the value of the character 'z'. The symbols && represent the operator "logical and." If both sub-expressions are true, then the expression:

```
'a' <= c && c <= 'z'
```

is true; otherwise it is false. Thus the expression is true if and only if the value stored in the variable c is a lowercase letter. If the expression is true, then the statement

```
putchar(c + 'A' - 'a'):
```

is executed. This putchar( ) calculates value of the corresponding uppercase letter which is printed.

```
else
      if (c == '\n')  {
         putchar( '\n' ) ',
         putchar( '\n' ) ',
      }
```

The two equal symbols == represent the operator "equals" as in is c equal to '\n'. It is important to remember not to use one equal symbol. If c is not a lowercase letter, another test is made to see if it is equal to a newline character '\n'. If it is, the two newline characters are printed.

If the value of c is not a lowercase letter and it is not a newline character, then the character corresponding to the value c is printed. An else is always associated with the immediately preceding if.

```
else
         putchar(c);
```

This program is portable to any ASCII computer, it will not work on an EBCDIC computer. The reason for this, is that the uppercase letters are not contiguous in the EBCDIC code. The next version of this program will work on either computer.

```
/*Capitalize lowercase letters & double space */
#include <stdio.h>
```

```
#include <ctype.h>
main(   )
{
int c;
while (( c = getchar (   )) != EOF )
    if( islower(c))
      putchar(toupper(c));
    else
      if( c == '\n') {
        putchar('\n');
        putchar('\n');
      }
      else
       putchar(c);
}
```

PROGRAM ANALYSIS

The file ctype.h, along with stdio.h,

```
#include <stdio.h>

#include <ctype.h>
```

is a standard header file provided with the C system. This file contains macros, which can be used when processing characters. A macro is code that gets expanded by the preprocessor. For the purposes of this lesson we will treat the macros in ctype.h just as if they were functions. Although there are technical differences between macro and a function, they both are used in a similar fashion. The macros islower( ) and toupper( ), which are used in this program, are found in ctvpe.h.

```
while (( c = getchar(   )) != EOF)
    if( islower( c ))
         putchar( toupper( c ));
```

A character is obtained from the input stream and assigned to the variable int c. As long as the value of c is not EOF, the body of the while loop is executed. The macro islower( ) is defined in ctype.h. If c is a lowercase letter, the islower(c) has value 1, otherwise the value 0. The macro toupper( ) is defined in ctype.h. If c is a lowercase letter then toupper(c) has the value of the corresponding uppercase letter. Therefore, the if statement has the effect of testing to see whether or not c has the value of a lowercase letter. If it does, then the

corresponding uppercase letter is written on the screen. The value stored in c is not changed by invoking isupper(c) or toupper(c).

The C programmer must not know exactly how the macros in ctype.h are implemented. Along with printf( ) and scanf( ), they can be treated as a system supplied resource. The important point to remember is that by using these macros you are writing portable code that will run with any compiler, not just an ASCII compiler.

The reason to emphasize the character constructs like c + 'A' - 'a' is that the majority of C code is written for an ASCII C compilers, and even though the construct is not considered good programming practice, the programmer will eventually encounter different compilers. Since a programmer must learn to read and test code as well as develop code, this particular construct should be mastered. In order to avoid non-portable code, the programmer should master and use the macros in ctype.h.

THE MACROS IN ctype.h

The C compiler system provides a standard header file ctype.h, which contains a set of macros that are used to test or convert characters. They are made accessible by the preprocessor control line:

```
#include <ctype.h>
```

The first table contain the macros that only test a character and return an int value that is either nonzero (true) or 0 (false).

MACRO	Nonzero(true) is returned if:
isalpha(c)	c is a letter
isupper(c)	c is an upper case letter
islower(c)	c is a lower case letter
isdigit(c)	c is a digit
isxdigit(c)	c is a hexidecimal digit
isspace(c)	c is a white space character
isalnum(c)	c is a letter or a digit
ispunct(c)	c is a punctuation mark
isprint(c)	c is a printable character
iscntri(c)	c is a control character
isascii(c)	c is an ASCII code

The second table contains macros that provide conversion of characters values. The macros do not change the value stored in the variable c. The function returns the converted value and the return value is used in the program logic.

MACRO	EFFECT
toupper(c)	change c to an upper case letter
tolower(c)	change c to a lower case letter
toascii(c)	changes c to an ASCII code

The first program in this lesson, showed how every character read in can be printed out twice. This example generalizes that idea by writing a function that prints out a given character n times.

```
repeat (char c, int n)
{
    int i;
    for( i =0; i <n;  ++i )
        putchar(c);
```

```
}/* End function repeat */
```

It is important to notice that the variable c is declared as a char, not an int. This is because a test for EOF is not made in this function. Therefore, there is no need to declare c an int.

When a function is called by another function, the term invoke is used and the formal parameter list sometimes called the function signature is called the actual argument list. Suppose the function is invoked with the statement

```
repeat( 'B' - 1, 2 + 3);
```

The arguments of this function call are the two expressions 'B' -1 and 2 + 3. The values of these arguments are passed and associated with the formal parameters of the function. The effect of the function call is to print the letter A five times.

The next example is a main( ) function that can be used to test the repeat( ) function.

```
main( )
{
  int  i;
  char bell = '\007', c = 'A';
  repeat ('B' - 1, 2 + 3);
  putchar(' ');
  for (i = 0; i <9; ++i) {
    repeat (c + i, i);
    putchar('\n);
  }

  repeat(bell, 100);
  putchar( '\n' );
}
```

When the program is compiled and run, this is displayed on the screen.

```
AAAAA B CC DDD EEEE FFFFF GGGGGG
HHHHHHH IIIIIII
```

The function repeat( ) can be used to draw simple figures on the screen.
Suppose that you want to count the number of words being input at the keyboard. Using top-down design to break up the problem into small manageable pieces. To

do this you need to know the definition of a word, and you need to know when to end the task. For this program assume that words are separated by white space. Making any word a contiguous string of nonwhite space characters. The program logic will end the processing of characters when an end-of-file sentinel is encounter. The key requirement of the program is a function that detects a word. This requirement will be explained in some detail.

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

main (   )
{
  int word_cnt = 0;
  while ( found_next_word ( ) == 1 )
        ++word_cnt;
  printf ("Number of words = %d\n\n", word_cnt );
}/* end of main function */

/*****************************************************************/

found_next_word(   ) /* function to find next word */
{
  int c
  while (isspace(c = get.char(   ))) /*while loop to skip white space */
        ;
  if (c != EOF)  {              /* Program found a word */
     while ((c = getchar(   )) != EOF &&  ! isspace (c) )
          ;   /* skip everything but EOF and white space */
     return ( 1 );
  }
  return ( 0 );
}
```

PROGRAM ANALYSIS

The

```
int variable word_cnt = 0;
```

is initialized to zero.

```
while (found_next_word(   ) = 1 )
    ++word_cnt;
```

As long as the function found_next_word( ) returns the value 1, the body of the while loop is executed, causing word_cnt to be indexed by one each time through the loop.

```
printf ("Number of words = %d\n\n", word_cnt);
```

Just before exiting the program, we print out the number of words found.

```
found_next_word ( ) /* function to find next word */
{
  int c;
```

This is the beginning of the function definition for found_next_word( ). The function has no parameters in its parameter list. In the body of the function the int variable c is declared. Although the program is going to use c to take character values, c is declared c an int, not a char. Eventually c will hold the special value EOF and on some systems that value may not fit into a char.

```
while (isspace(c = get.char( ))) /*while loop to skip white space */
       ;
```

A character is received from the input stream and assigned to the variable c. The value of the sub-expression

```
c == getchar( )
```

takes on this value. As long as this value is a white space character, the body of the while loop is executed. The body of the while loop is just the empty statement. The effect of the while loop is to skip white space. The empty statement ; is displayed on a line by itself. It is good programming practice to put the semi-colon on a line by itself. If you had written

```
while (isspace(c = getchar( )));
```

the visibility of the empty statement would not be obvious.

```
if (c != EOF)  {              /* Program found a word */
    while ((c = getchar( )) != EOF &&  ! isspace (c) )
         ;   /* skip everything but EOF and white space */
    return ( 1 );
  }
```

After white space has been skipped, the value of c is either EOF or the first

"letter" of a word. If the value of c is not EOF, then a word has been found. The test expression in the while loop consists of three parts. First a character is received from the input stream and assigned to c, and the sub-expression c = getchar( ) assigns a value to c. A test is then made to see if that value is EOF. If it is, then the body of the while loop is not executed and control is passes to the next statement. If the value is not EOF, then a test is made to see if the value is a white space character. If it is, then the body of the while loop is not executed and control is passes to the next statement. If the value is not a white space character, then the body of the while loop is executed. The body is the empty statement semi-colon. The effect of this while loop is to skip everything except EOF and white space characters. This means that the word that has been found and has now been skipped.

```
return (1);
```

After a word has been found and skipped, the value 1 is returned, meaning the word was found.

```
return (0);
```

If a word was not found, then the value 0 is returned.