LESSON 7

POINTERS AND CALL BY REFERENCE

When an expression is passed as an argument to a function, a copy of the value of the expression is made, and it is this copy that is passed to the function. Suppose that var is a variable and foo( ) is a function. Then a function call such as foo(var) cannot change the value of var in the calling environment, because only a copy of the value of var is passed to foo( ). In other programming languages, however, such a function call can change the value of var in the calling environment. The mechanism that accomplishes this is known as call by reference. It is often convenient to have functions modify the values of the variables referred to in the argument list. To get the effect of call-by-reference in C, you must use pointers in the parameter list in the function definition, and pass the addresses of variables as arguments in the function call. Before the lesson explains this in detail, however, you need to understand how pointers work. In C pointers have many uses. This lesson explains how pointers are used as arguments to functions. The two lessons that follow will explain how pointer are use with arrays and strings. The lessons will show how pointers are used with structures, and later, how pointers are used with files. Pointers are used in programs to access memory and manipulate addresses. You have already seen the use of addresses as arguments to scanf( ). A function call such as scanf("%d", &var) causes an appropriate value to be stored at a particular address in memory. If var is a variable, then &var is the address, or location, in memory that contains the stored value. The address operator & is unary operator and has the same precedence and "right to left" associativity as the other unary operators. Pointer variables can be declared in programs and used to hold an addresses as values. The declaration

```
int *ptr:
```

declares ptr to be of type "pointer to int". The legal range of values for any pointer always includes the special address NULL, also defined as 0 in <stdio.h>, and a set of positive integers that are interpreted as machine addresses in a particular C environment. Some examples of assignment to the pointer ptr are:

```
ptr = &i;
ptr = 0;
ptr = NULL;     /* equivalent to ptr == 0; */
ptr = (int *) 1999;     /* an absolute address in memory */
```

In the first example think of ptr as "referring to i" or "pointing to i" or "containing the address of i". The compiler decides what address to assign the variable i. This will vary from machine to machine and may even be different for different executions on the same machine. The second and third examples are assignments of the special value 0 to the pointer ptr. In the fourth example the cast is necessary to avoid a compiler warning. In this example an actual memory address is used. This is not typical for beginning programmers. Code like this may be necessary for system programs, but rarely for elementry programs.

ADDRESSING AND DEREFERENCING

You have already seen that addresses are passed as arguments to scanf( ). The lesson will now show how a pointer can be used instead. Suppose you declare:

```
int  i, *ptr;
```

Then the statements

```
ptr = &i:
scanf("%d", ptr):
```

causes the next value from the standard input stream to be stored in ptr. But since ptr points to i, this is equivalent to storing the value at the address of i. The dereferencing or indirection operator * is unary and has the same precedence and "right to left" associativity as the other unary operators. If ptr is a pointer, then *ptr is the value of the variable that ptr points to. The name "indirection" is taken from machine language programming. The direct value of ptr is a memory location, whereas *ptr is the indirect value of ptr, namely the value at the memory location stored in ptr. . The dereferencing or indirection operator * is the inverse operator to &. The short program illustrates the distinction between a pointer value and its dereferenced value.
/* Pointer value and dereferenced value */

```
main(   )
{
  int i = 123, *ptr;
  ptr = &i;
  printf("The value of i:     %u\n", *p);
  printf ("The address of i:     %u\n", p);
}
```

The output of this program is the following:

```
The value of i:     123
The address of i:     f12345
```

The actual location of a variable in memory is system dependent. The operator * takes the value of ptr to be a memory location and returns the value stored in this location, appropriately interpreted according to the type declaration of p. In the above program, if you had wanted to initialize ptr rather than assign its value in an assignment statement, you sould have written

```
int i = 123, *ptr = &i;
```

This is an initialization of ptr, not *ptr. The variable ptr is declared to be of type "int *" and its initial value is &i

CALL BY REFERENCE

Whenever variables are passed as arguments to a function, their values are copied to the corresponding function parameters, and the variables themselves are not changed in the calling environment. This call-by-value mechanism is strictly adhered to in C. This lesson will describe how the addresses of variables can be used as arguments to functions to be able to modify the stored values of the variables in the calling environment. For a function to effect call-by-reference, pointers must be used in the parameter list in the function definition. Then, when the function is called, addresses of variables must be passed as arguments. An example program shows how to swap the values stored in two variables. Most of the work of this program is carried out by the function call to swap_ptr( ). Notice that the addresses of i and j are passed as arguments. As you shall see, this allows the function call to change the values of i and j in the calling environment. The calling environment is the function swap_ptr.

```
main( )
{
   void  swap_ptr(int *, int *);
   int i=9,  j=2;
   printf("%d %d\n", i, j);/* 9 2 is printed */
    swap_ptr (&i, &j);
    printf("%d %d\n",i,j);    /* 2 9 is printed */
}
void swap_prt (int *p, int *q)
{
   int temp;
   if (*p > *q) {
      temp = *p;
      *p = *q;
      *q = temp;
     }/* end for loop */
}/* end swap_ptr */
```

PROGRAM ANALYSIS

```
void swap_prt (int *p, int *q)
{
   int temp;
```

The parameters p and q are both of type pointer to int. The variable temp is local to this function and is of type int. which is a temporary storage location.

```
if (*p > *q) {
      temp = *p;
      *p = *q;
      *q = temp;
     }/* end for loop */
```

If the value of what is pointed to by p is greater than the value of what is pointed to by q, then the if block is executed. First, temp is assigned the value of what is pointed to by p; second, what is pointed to by p is assigned to the value of what is pointed to by q; and third, what is pointed to by q is assigned the value of temp. This has the effect of interchanging in the calling environment the stored values of what p and q are point to.

Call-by-reference is accomplished by

- Passing an address as an argument when the function is called.
- Declaring a function parameter to be a pointer.
- Using the dereferenced pointer in the function body.

PROCESSING CHARACTERS EXAMPLE

A function that uses a return statement can pass back to the calling environment a single value. If more than one value is needed in the calling environment, then addresses must be passed as arguments to the function. The next example program processes characters in this way. Here is what the program is to accomplish:

- Read characters from the input stream until EOF is encountered.
- Change any lowercase letter to an uppercase letter.
- Print three words to a line with a single space between each word.
- Count the number of characters and the number of letters printed.

For this program a word is to be a maximal sequence of characters containing no white space characters.

```
/************** main program to process characters *************/
#include <stdio.h>
#include <ctype.h>

#define NUM_WORDS       3      /* Defines the number of words per line
*/
/* The function prototype */
int  process_char (int *ptr, int *num_chars_ptr, int *num_letters_ptr);

main(   )
 {
   int ctr, num_chars = 0, num_letters = 0,

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

```
          if ( process_char (&ctr, &num_chars, &num_letters) == 1)
            putchar(ctr);
        printf("\n%s%5d\n%s%5d\n\n",
                              "Number of characters:", num_chars,
                               "Number of letters: ", num_letters);
    }/* end main program */
```

The processing of each character takes place in the function process_char(   ). Since the values of the variables ctr, num_chars, and num_letters are to be changed in the calling environment, addresses of these variables are passed as arguments to process_char(   ). Notice that ctr is an int rather than a char. This is because ctr must eventually take on the special value EOF, which is not a character. Notice also that a character gets written on the screen only if process_char(   ) returns the value 1. In the if test the value of 1 is returned as a sentential that the character has been appropriately processed and that it is ready to be print. The value 0 is used to signal that the character is not to be printed. This case will occur when contiguous white space characters occur. The nest code shows you how the function process_char(   ) executes.

```
/********************** process_char ****************?

int  process_char (int *ptr,  int *num_chars_ptr,  int *num_letters_ptr)
{
static int count = 0, last_crt = ' ';
if (isspace(last_crt) && isspace(*ptr))
          return ( 0 );
if (isalpha (*ptr)) {
          ++*num_letters_ptr;
          if (islower (*ptr))
                    *ptr = toupper (*ptr);
          }
else
          if (isspace(*ptr))
                    if (++count % NUM_WORDS == 0)
                              *ptr = '\n';
                    else
                              *ptr=' ';
++*num_chars_ptr;
last_crt = *ptr;
return ( 1 );
}/* end function process_char */
```

Before this function is analyzed, lets look at the expected output from the program. Assume the file called text.dat contains the following lines:

Now is the time

```
        for all good Programmers

code this program.
```

Notice that the file contains contiguous blanks. Now compile the code and call the executable process.exe and on the command line the input indirection operator

```
process < text.dat
```

what appears on the screen is:

```
NOW IS THE

TIME FOR ALL

GOOD PROGRAMMERS TO

CODE THIS PROGRAM.

Number of characters:  xx

Number of letters:  xx
```

ANALYSIS OF THE process_char **FUNCTION**

```
int  process_char (int *ptr,  int *num_chars_ptr,  int *num_letters_ptr)
{
static int count = 0, last_crt = ' ';
```

The parameters of the function are three pointers to int. Although you can think of ptr as a pointer to a character, its declaration must be consistent with its use in main( ). The local variables count and last_char are declared of type static so that they will only be initialized once. If declared type auto, they would be reinitialized every time the function process_char is called.

```
if (isspace(last_crt) && isspace(*ptr))
        return ( 0 );
```

If the last character seen was a white space character and the character pointed to by ptr is also a white space character, the value 0 is returned to the calling environment. The calling environment is main( ) and when this value is received: the current character is not printed.

```
if (isalpha (*ptr)) {
        ++*num_letters_ptr;
        if (islower (*ptr))
                *ptr = toupper (*ptr);
        }
```

If the character pointed to by ptr is a letter, then the function increments the value pointed to by num_letters_ptr. Also, if the value pointed to by ptr is a lowercase letter, then the function assign to the value pointed to by ptr the corresponding uppercase letter.

```
++*num_chars_ptr;
```

The increment operator ++ and the indirection operator * are both unary and associate "right-to-left". This statement is equivalent to

```
++(*num_chars_ptr);
```

The expression in the parentheses is being incremented. This expression is the dereferenced value in the calling environment, here the main function.

The next set of statements:

```
else
        if (isspace(*ptr))
                if (++count % NUM_WORDS == 0)
                        *ptr = '\n';
                else
                        *ptr=' ';
```

If the character pointed to by ptr is a white space character, then last_char cannot also be a white space character; the function has already evaluated that case. No matter what this character is, the function will print a newline or a blank, depending on the incremented value of count. Since the symbolic constant NUM_WORDS is the value 3, every third time we print a newline, and the other two times we print a blank. The effect of this is to print at most three words to a line with a single blank between them.

In this sequence

```
++*num_chars_ptr;
last_crt = *ptr;
return ( 1 );
}/* end function process_char */
```

the function increments the value pointed to by num_chars_ptr. Then the function assigns to last_char the value pointed to by ptr. The last step function returns the value 1 to the calling environment to indicate that a character is to be printed.

You will often see that p, q and r are used as identifiers for pointer variables in functions. This is because, that p standing for "pointer," and q and r are the next letters in the alphabet. In a similar fashion p1, p2, ... pn are also used as identifiers for pointer variables. Other ways to designate that an identifier is a pointer is to prepend p to a name, as in p_high and p_low, or to append ptr, as in num_chars_ptr.

An alternative declaration style for a pointer is:

```
char*  ptr;
```

which is equivalent to

```
char  *ptr;
```

Some programmers prefer this style because the * is closer to the variable type being pointed to. In the declaration, you must be careful to declare each variable, because

```
char*  p,q,r;
```

is not equivalent to

```
char  *p, *q, *r;
```