13.15 Write a function to validate data input giving an inventory list in the following format:

Column Numbers	Information	Valid Range
1 to 5	Item number	1 to 99999
6 to 8	Supplier code	222 to 888
7 to 10	Quantity	1 to 9999
11 to 16	Cost/unit	≥ 5.0 and ≤ 500.0

Invalid data should be written out with an appropriate message.

14. Processing Character Strings

Learning Objectives

In this chapter we will learn:

1. How to declare, read, and write characters
2. How to declare and manipulate strings of characters
3. How to develop general purpose functions to manipulate strings and how they are used

So far we have considered only the simplest data types available in C. These data types are sufficient for numeric applications. The greatest strength of C is the rich data types and data structures provided by the language. This enables easy development of programs in many application areas such as manipulation of strings of character, lists, records and files. These applications are primarily non-numeric. We will introduce in this chapter a data type called *char* for characters and illustrate operations on strings of characters.

14.1 THE CHARACTER DATA TYPE

Individual characters such as a, b, c, etc. can be stored in variable names. If a variable name is to store a character then it is declared to be of type *char*. For example, if we declare

```
char letter;
```

then the variable name latter may store any character available in the character set of the computer installation. Character sets are not, unfortunately, standardised. They vary from machine to machine. There is a trend towards standardization using the character set suggested by the American National Standards Institute (ASCII character set). A programmer in an installation should find out what is implemented in the particular C system.

When a variable is declared as *char* only a *single* character can be stored in it. For example, we may write:

```
letter = 'y';
```

The character to be stored in the variable name is enclosed by apostrophes (single quotes). A declaration as *char* implies that the variable name is a scalar data type which can store only one character and not a string of characters. Thus:

```
char letter ;
letter = "xyz"
```

is illegal as letter is a scalar. A string is written in C as "xyz". We will describe later how such strings are treated.

Characters are stored internally in a computer as a coded set of binary digits which have

positive decimal integer equivalents. These codes can be ordered in an ascending sequence called a *collating sequence*. The ASCII collating sequence and codes is given in Table 14.1.

Table 14.1 ASCII Codes for Characters

Left Digits	Right Digit →	0	1	2	3	4	5	6	7	8	9
3					!	"	#	$	%	&	'
4		(	)	*	+	,	−	.	/	0	1
5		2	3	4	5	6	7	8	9	:	;
6		<	=	>	?	@	A	B	C	D	E
7		F	G	H	I	J	K	L	M	N	O
8		P	Q	R	S	T	U	V	W	X	Y
9		Z	[	\	]	^	_	`	a	b	c
10		d	e	f	g	h	i	j	k	l	m
11		n	o	p	q	r	s	t	u	v	w
12		x	y	z	{	|	}	~			

Observe that the codes for upper and lower case letters are different. Thus *A* is different from *a*. Note also that there are two representations for digits. If a digit 4 is stored in a variable name of type *integer* then it is converted to binary and can be used in arithmetic expressions. If it is stored in a variable name of type *char* then it is coded as an ASCII character and has the code 52. It can be used in arithmetic expressions but will be taken as 52.

Codes 00 to 31 and code 127 represent special control characters which cannot be printed. Code 00 represents null and code 32 represents a blank character. The code for Z, for example, is 90 (in decimal) and its binary representation in the computer would be 1011010.

In C language the ASCII equivalent of characters is stored as an integer. We can manipulate these integers using arithmetic operations. We can find out the ASCII equivalent of a character by printing its value as shown below:

```
char letter;
int p;
letter = 'a';
p = letter ;
printf("%d\n", p);
```

The value printed will be 97.

14.2 MANIPULATING STRINGS OF CHARACTERS

In most applications a string of characters rather than a single character is necessary. As an individual variable name can store only one character we need an array to store strings of characters. For example, the declaration:

```
char name[6];
```

declares name as an array of length 6 where each element of name stores a character. If we write

```
for (i = 0, i < = 5, i ++)
    scanf("%c", &name[i]);
```

and the data input from the keyboard is:

LADDUS

then we will have the following stored:

name[0] = 'L', name [1] = 'A', name[2] = 'D', name [3] = 'D', name[4] = 'U', and name[5] = 'S'

In the format string in scanf, %c is used to specify a character. The letter c in %c is lower case c.

The string LADDUS can also be read using the statement:

scanf ("%s", name);

where name should be declared an array. In the format string %s is used to specify a *string*. Observe that before name the ampersand & is *not* written as name itself is the address of the element name [0]. The specification %s in the format, states that a string is stored in the array name.

If data input from the keyboard is:

LADDUS \n (\n is carriage return)

then we will have as before name [0] = 'L'; name [1] = 'A', name [2] = 'D', name [3] = 'D', name [4] = 'U' and name [5] = 'S'.

There is one danger in using %s specification. If the string being read has a blank then scanf assumes that the string is terminated and stops reading.

Thus if we write:

scanf ("%s", buffer);

and the input string is

HE CAME

then buffer[0] = 'H' and buffer[1] = 'E'. The string CAME will not be read by the statement. If we write

scanf("%s%s", buff_1, buff_2);

then HE will be stored in array buff_1 and CAME in array buff_2.

Strings may be displayed by using a printf function with %s specification. Thus if we write

printf("name displayed is :%s \n", name);

then the display will be:

name displayed is : LADDUS

If a string HE CAME HOME is stored in the array buff_3 the statement

printf ("Display of buff_3:%s\n", buff_3);

will display it. Observe that blanks within a string does not upset printf function.

In C an individual character is represented by using a pair of single quote marks to enclose the character. If we write:

s[1] = 'P';

then the character P is stored in s[1]. If we write

s[1] = "P"; /* This is wrong */

then the right hand side is taken as a *string*. A string is a sequence of characters and is automatically terminated by the end of string character '\0' by the C translator. Thus we require two elements in an array to store a string. (Even though \0 looks like two characters, a back slash and 0, C language treats it as a *single* end of string character.)

To illustrate the use of character data type, Example Program 14.1 gives a small program to count the number of 'A's in a name. Observe in this program that name is declared as an

```
/* Example Program 14.1 */
/* Program to count number of A's in a set of
   names */
#include <stdio.h>

main()
{
  char name[20];
  int i, countA = 0;
  for (i = 0; i < 20; ++i)
    {
      scanf("%c", &name[i]);
      printf("%c", name[i]);
      if (name[i] == '\n')
        break;
      else
        if (name[i] == 'A')
          ++countA;
    } /* End of for loop */
  printf("Number of A's in name = %d\n", countA);
} /* End of Main */
```

Program 14.1 Number of 'A's in a name.

array with 20 characters. These 20 characters should include the end of line character '\n'. When a name is typed on the terminal we do not type \0 at the end of the name. A carriage return is taken as end of line. If we do not press carriage return after typing the name then the program will continue in the *for* loop 20 times and will read the blanks, if any, at the end of the name also. A blank (or white space as it is sometimes called) is also a character (A null character is "absence of a character" and is represented by '\0' which is also used as end of string character).

Returning to Example Program 14.1, in the *for* loop scanf reads one character at a time from the standard input and stores it in name [0], name[1] ...etc., upto name [19]. It is printed by printf which is the next statement. The *if* statement following printf checks if the character read is an end of line character. If it is, then it executes *break* and leaves the loop. Otherwise it checks if the character is an 'A'. If it is 'A' then it increments the count of A.

C language also provides some library functions to read characters from a terminal and display them on a terminal. These functions are:

int getchar(void)

This function reads a character from the input and returns it. Observe that the returned

value is of type *int* even though a character is read. This need not concern us. If we write

```
char in_char ;
in_char = getchar( );
```

then a character read from input will be stored in the character variable in_char.

The companion function to display a character on the terminal is:

```
void putchar(char p);
```

If we write:

```
char out_char;
putchar (out_char);
```

then a character stored in out_char will be displayed on the terminal.

We show the use of these functions in Example Program 14.2.

```
/* Example Program 14.2 */
/* Illustrating using getchar() and putchar() */
#include <stdio.h>

main()
{
  char name[20];
  int i, countA = 0;
  for (i = 0; i < 20; ++i)
    {
      name[i] = getchar();
      putchar(name[i]);
      if (name[i] == '\n')
       break;
      else
       if (name[i] == 'A')
         ++countA;
    } /* End of for loop */
  printf("\nNumber of A's in name = %d\n", countA);
} /* End of main */
```

Program 14.2 Use of getchar ( ) and putchar ( ).

We will now develop a few routines to illustrate basic operations with strings. The routines are:

1. Read a string and store it.
2. Find the length of a string (i.e. the number of characters in a string).
3. Copy a specified portion of a string.
4. Delete specified characters from a string.
5. Concatenate a string at the left of a given string.
6. Concatenate a string at the right of a given string.

7. Delete occurrence of a substring from a string.
8. Insert a string in specified position of a given string.

These functions are useful in many complex string processing problems.

As C does not provide a dynamically changing array size we will provide 81 characters as maximum length of a string. This choice is based on the fact that most terminals have 80 characters per line. The 81st character could be the character equivalent of the code when the return key of a terminal is pressed.

Example Program 14.3 reads a line from a terminal and stores it in an array named

```
/* Example Program 14.3 */
/* A function to read a line and store it in a
   buffer Another function to print a line is also given */
#include <stdio.h>
void read_line(char s[]);
void print_line(char s[]);
main()
{
  char buffer[81];
  read_line(buffer);
  print_line(buffer);
} /* End of main */
void read_line(char string[])
{
  int i = 0;
  string[i] = getchar();
  while (string[i] != '\n')
    {
      string[++i] = getchar();
    } /* End of while */
  if(i == 0)
    {
      printf("Empty string\n");
      return;
    }
  string[i] = '\0';  /* Overwrites '\n' character */
  return;
} /* End of read_line */
void print_line(char out_string[])
{
  int i = 0;
  while(out_string[i] != '\0')
    {
      putchar(out_string[i]);
      ++i;
    } /* End while */
} /* End of print_line */
```

Program 14.3 Function to read a line.

buffer. If there is an attempt to read an empty line an appropriate message is given by the function. Examining the function read_line (char_string[ ]) we see that in its body we initialize i to 0, get a character from the standard input unit and store it in string[0]. In the *while* loop this character is checked and if it is '\n' the *while* loop is exited and a message printed. If the character is not '\n' the body of the while loop is repeated. The next character is read from the input and assigned to string[1]. Reading a character and storing it in string[i] is repeated until end of line is encountered. (End of line is indicated by the character corresponding to return key on the terminal which is pressed after typing a line.) When the end of line is encountered control leaves the *while* loop.

Observe that if the input string is empty then the first character itself will be '\n'. Control leaves the loop and an error message is printed. If i > 0 then the statements in the body of the loop would have been executed at least once. Control leaves the *while* loop when it reads a '\n' character. We replace '\n' character by the end of string character '\0' using the statement string [i] = '\0'. Control now leaves the *function*.

The *function* print_line (char out_string[ ]) displays the given string.

Observe the while loop in the *function* body. It displays characters one after one until end of string character is encountered.

Example Program 14.4 is a *function* to find the length of a string. The program is quite simple to understand.

```
/* Example Program 14.4 */
/* A function to find length of a string */
/* length of string does not include '\0' character */
int length(char string[])
{
 int i = 0, len;
 while (string[i] != '\0')
   ++i;
 if (i == 0)
   len = 0;
 else
   len = i;
 return(len);
} /* End of function length */
```

Program 14.4 Finding length of a string.

Example Program 14.5 is a *function* which copies a specified number of characters from a given position of a string and places it in another string. Example Program 14.6 is a *function* to delete all the occurrences of a specified character from a string and store the compressed string back in the given string.

Referring to Example 14.6 we see that in the first *for loop* in the *function* a character in the given_string is copied to the temp_string only if it is not to be deleted. If it is to be deleted it is not copied in the temp_string. Index *i* is incremented to read the next character in given_string. The index used for temp_string is *j* and it is incremented within the loop. After creating this temporary string it is written back in the given_string. At the end of the compressed string '\0' character is placed. Observe how the program is written using all the *functions* developed so far in this chapter.

```
/* Example Program 14.5 */
/* Copy into new_string, substring of given_string
   from (from_posn) the given number of characters */

void copy_substring(char given_string[],
                    int from_posn,
                    int no_char, char new_string[])
{
  int i;
  for (i = 0; i < no_char, ++i)
    new_string[i] = given_string[from_posn -1 + i];
  /* [from_posn -1 + i] is used as given_string
     index starts with 0 */
  new_string[i] = '\0';
} /* End of copy_substring */
```

Program 14.5 Copying substring into a string.

```
/* Example Program 14.6 */
/* Deleting specified characters in a string and
   compressing it */
#include <stdio.h>
void read_line(char s[]);
void print_line(char s[]);
int length(char s[]);
void delete_compress_str(char s[], char deleted);
main()
{
  char d, string[81];
  /* d is the character to be deleted */
  read_line(string);
  d = getchar();
  delete_compress_str(string, d);
  print_line(string);
  printf("\n");
} /* End of main */
/* Insert functions read-line, print-line and
   length here */
void delete_compress_str(char given_string[],
                         char given_char)
{
  int i = 0, j = 0, len, g_length;
  char temp_string[81];
  g_length = length(given_string);
  for(i = 0; i <= g_length; ++i)
    if (given_string[i] != given_char)
      {
        temp_string[j] = given_string[i];
        ++j;
      }
  temp_string[j] = '\0';
  len = length(temp_string);
  for(i = 0; i <= len; ++i)
    given_string[i] = temp_string[i];
  given_string[i + 1] = '\0';
} /* End of compress_string */
```

Program 14.6 Deleting characters from a string.

Example Program 14.7 concatenates (i.e. appends) a string to the right of a given string and prints the concatenated string. Observe that this program uses the functions already developed. The function concatenate_right is traced in Table 14.2.

```
/* Example Program 14.7 */
/* Concatenating a string to the right of the given
   string */
#include <stdio.h>
void read_line(char s[]);
void print_line(char s[]);
int length(char s[]);
void concatenate_right(char s[], char r[]);

main()
{
 char given_string[81], new_string[81];

 read_line(given_string);
 read_line(new_string);
 concatenate_right(given_string, new_string);
 print_line(given_string);
 printf("\n");
} /* End of main */

void concatenate_right(char g_str[], char str[])
     /* Concatenate a string to the right of the
        given string */
{
  int i, given_length, str_length;
  given_length = length(g_str);
  str_length = length(str);
  for (i = given_length;
       i <= given_length + str_length; ++i)
    g_str[i] = str[i - given_length];
}  /* End of concatenate right */

/* Functions read_line, print_line and length are
   included here */
```

Program 14.7 Concatenating a string to the right.

Example Program 14.8 concatenates a string to the left of a given string. This function is more tricky as we have to move the given_string right a number of positions equal to the string which is to come on its left. We cannot write given_string [i + 1] = given_string[i] to shift the given_string right as what was originally in given_string [i + 1] will be over-written. Table 14.3 is a trace of this function which the student is urged to study carefully.

Table 14.2 Trace of function concatenate_right

i	0 1 2 3 4	
g_string	a b c d \0	given_length = 4(\0 not counted)
str	x y z \0	str_length = 3(\0 not counted)
for(i = 4;i <= 7; ++i) g_string[i] = str[i – given_length];		
i	0 1 2 3 4 5 6 7	
g_string	a b c d x y z \0	
str	x y z \0	

```
/* Example Program 14.8 */
/* Program to concatenate a string to the left of
   a given string */
#include <stdio.h>
void read_line(char s[]);
void print_line(char s[]);
int length(char s[]);
void left_concatenate(char s[], char r[]);

main()
{
  char given_string[81], conc_string[81];
  read_line(given_string);
  read_line(conc_string);
  left_concatenate(given_string, conc_string);
  print_line(given_string);
  printf("\n");
}  /* End of Main */

/* Place read_line, print_line and length functions
   here */

/* Concatenate string to the left of given_string */
void left_concatenate(char g_str[], char str[])
{
 int i, g_length, str_length;
 char temp[81];
 g_length = length(g_str);
 str_length = length(str);
 for (i = 0; i < str_length; ++i)
    temp[i] = str[i];
 for (i = 0; i <= g_length; ++i)
    temp[str_length + i] = g_str[i];
 for (i = 0; i < str_length + g_length; ++i)
   g_str[i] = temp[i];
} /* End of concatenate */
```

Program 14.8 Concatenating a string to the left.

Table 14.3 Trace of function left_concatenate

i	0	1	2	3	4			
g_string	a	b	c	d	\0			g_length = 4
str	x	y	\0					str_length = 2
for (i = 0; i < 2; ++i) temp[i] = str[i];								
	0	1	2					
temp	x	y	\0					
for (i = 0; i <= 4; ++i) temp[str_length + i] = g_str[i]								
i	0	1	2	3	4	5	6	
g_str	a	b	c	d	\0			
temp	x	y	a	b	c	d	\0	
for (i = 0; i < 6, ++i) g_str[i] = temp[i];								
i	0	1	2	3	4	5	6	
g_str	x	y	a	b	c	d	\0	
str	x	y	\0					

Example Program 14.9 is to delete a specified number of characters from a given string from a specified position. After deletion the given string is shortened and the rest of the string to its end is filled with null characters.

The last program in this section is to insert a string in a given string from a specified position. We would demonstrate how to use previously developed functions to do this job. The strategy we will use is given in Algorithm 14.1. This algorithm is written as a C program in Example Program 14.10.

Algorithm 14.1: Insert string in a given string

Given: given_string, insertion_string, position from which to insert the insertion_string in given string.

Procedure

Step 1: Copy given_string from beginning to (insert–pos) to a temp_string

Step 2: Delete the copied part of the given_string from the given_string and compress the given_string

Step 3: Right concatenate insertion_string to temp_string

Step 4: Right concatenate given_string (as at end of step 2) to temp_string

Step 5: Store back temp_string in given_string

```
/* Example Program 14.9 */
/* Function to delete characters from specified
   position */
#include <stdio.h>
void read_line(char s[]);
void print_line(char s[]);
int length(char s[]);
void delete_chars(char s[], int from, int no_of_char);
main()
{
  char given_string[81];
  int from_position, no_of_chars;
  read_line(given_string);
  scanf("%d %d", &from_position, &no_of_chars);
  delete_chars(given_string, from_position,
               no_of_chars);
  print_line(given_string);
  printf("\n");
} /* End of main */
/* Place read_line, print_line, length functions here */

void delete_chars(char given_string[],
                  int from_pos, int no_char)
{
 int g_length, i;
 g_length = length(given_string);
 if ((no_char > g_length) || (from_pos > g_length))
   {
     printf("Error in input parameters\n");
     return;
   }
 for (i = from_pos; i <= g_length - no_char; ++i)
   given_string[i] = given_string[i + no_char];
/* This for loop puts null characters to end of
   given string */
 for (i = g_length - no_char + 1; i <= g_length; ++i)
   given_string[i] = '\0';
} /* End of delete chars */
```

Program 14.9 Deleting characters from a specified position.

14.3 SOME STRING PROCESSING EXAMPLES

Example 14.1

As the first example we will write a program to rearrange a set of names with last name first. We assume that the names are given in one of the following forms:

First name *blank* or . Middle name *blank* or . Last name

```
/* Example Program 14.10 */
/* Function to insert string */
#include <stdio.h>
int length(char s[]);
void read_line(char string[]);
void print_line(char out_string[]);
void copy_substring(char s[], int a, int p.
                    char temp[]);
void delete_chars(char s[], int from, int to);
void concatenate_right(char s[], char ins[]);
void insert_string(char s[], char ins[], int from);

main()
{
  char given_string[81], insertion_string[81];
  int from;

  read_line(given_string);
  read_line(insertion_string);
  scanf("%d", &from);
  insert_string(given_string, insertion_string, from);
  print_line(given_string);  printf("\n");
} /* End of main */

void insert_string(char given_string[],
                   char insertion_string[],
                   int insert_position)
{
  int len;
  char temp_string[81];
  /*  Copy to temp_string position 0 to
      (insert_position -1) of given string */
  copy_substring(given_string, 1,
                 insert_position - 1, temp_string);
  delete_chars(given_string, 0, insert_position - 1);
  concatenate_right(temp_string, insertion_string);
  concatenate_right(temp_string, given_string);
  len = length(temp_string);
  copy_substring(temp_string, 1, len, given_string);
} /* End of insert_string */
```

Program 14.10 Inserting a string.

First name *blank* or . Last name
First name
First name *blank* or . Middle initial *blank* or . Last name
First initial *blank* or . Middle name *blank* or . Last name
First initial *blank* or . Middle initial *blank* or . Last name etc.

For example:

Given names	*Rearranged names*
RAM KUMAR GUPTA	GUPTA RAM KUMAR
S. RAJU	RAJU S.
ARVIND	ARVIND
RAM K. VEPA	VEPA RAM K.
V. RAM KUMAR	KUMAR V. RAM
A.V. GANESH	GANESH A.V.
V.K.R.V. RAO	RAO V.K.R.V.

The strategy of the algorithm is given as Algorithm 14.2.

Algorithm 14.2: Algorithm to rearrange a name

Step 1: Read name_string

Step 2: Scan name_string from right to left and find the number of characters from end of name to the first occurrence a '.' or *blank*. Call it j.

Step 3: Copy these j characters to a temp_string

Step 4: Concatenate (length of name – j) characters of name_string to the right of temp_string.

Step 5: Copy temp_string to name_string

Step 6: Print name_string

This algorithm is implemented as Example Program 14.11. Observe the use of functions developed earlier in the program. Observe the use of the function exit ( ) provided by C language. This function returns control to main ( ).

Example 14.2

An identifier in a language is defined as a string with 8 or less characters. Any character beyond 8 are ignored. It must start with an upper case letter. All other characters in the string should be either an upper case letter or a digit. A function is to be developed which returns a 1 if it is a legal identifier and a 0 if it is not legal. An appropriate error message should be printed.

The function is given in Example Program 14.12. Observe the use of character class tests isupper and isdigit which are available as a library in C. The character class tests are defined in the library <ctype.h>. The tests available in this library are given in Appendix V.

Example 14.3

A paragraph is given which has more than one line. Each line may have up to 80 characters and is terminated by an end of line character '\n'. It is required to find the number of words in the paragraph. A word is not split between adjacent lines. Words are separated by one blank space or a full stop. The average number of letters per word is also to be computed. Algorithm 14.3 is developed to do this task.

Example Program 14.13 implements the following algorithm.

```
/* Example Program 14.11 */
/* Rearranging names */
#include <stdio.h>
void read_line(char s[]);
void print_line(char s[]);
void copy_substring(char g_s[], int f_p,
                    int no_ch, char s[]);
void delete_chars(char g_s[], int f_p, int no_ch);
void concatenate_left(char g_s[], char s[]);
void concatenate_right(char g_s[], char s[]);
int length(char s[]);

main()
{
  char name_string[81], temp_string[81];
  int i, p, j, n_length;
  read_line(name_string);
  print_line(name_string);
  printf("\n");
  n_length = length(name_string);
  /* This for loop scans name from right to left
     and stops when a blank or a '.' is encountered or
     string ends. It counts the number of
     characters scanned */
  for (i = n_length; (i != 0) &&
       (name_string[i] != ' ') &&
       (name_string[i] != '.'); --i)
    ++j;
  /* At the end of the for loop j contains the
     number of characters from the end of name to
     the first occurrence of a '.' or a blank */
  p = n_length - j;
  /* p is the number of characters in name_string
     from the beginning to the last blank or a '.' */
  if (p == 0)
    {
     print_line(name_string);
     exit(0);
    }
  /* print name and exit if the person has no more
     initials or first name. Otherwise copy into
     temp_string j characters from position p */
  copy_substring(name_string, p + 2, n_length,
                 temp_string);
  delete_chars(name_string, p + 2, n_length - p);
  concatenate_left(name_string, " ");
  concatenate_right(temp_string, name_string);
  copy_substring(temp_string, 1, n_length + 1,
                 name_string);
  name_string[n_length + 2] = '\0';
  print_line(name_string);
  printf("\n");
} /* End main */

/* Insert functions length, read_line, print_line,
   copy_substring, delete_chars, concatenate_right,
   concatenate_left */
```

Program 14.11 Rearranging names.

```
/* Example Program 14.12 */
/* Checking whether an identifier is syntactically
   correct */
#include <stdio.h>
#include <ctype.h>
int check_identifier(char name[]);
main()
{
  char n[20];
  scanf("%s", n);
  if (check_identifier(n))
    printf("Identifier legal\n");
  else
    printf("Identifier illegal\n");
} /* End of main */

int check_identifier(char name[])
{
  int i, ok;
  if (!(isupper(name[0])))
    {
      printf("First Character not an uppercase letter\n");
      return(0);
    }
  for (i = 1; (i <= 7) && (name[i] != '\0'); ++i)
    {
      if (isupper(name[i]))
       ok = 1;
      else
       if (isdigit(name[i]))
         ok = 1;
      else
         {
           printf("%dth character illegal\n", i);
           ok = 0;
           break;
         }
    }
  return(ok);
} /* End of check_identifier */
```

Program 14.12 Checking identifier syntax.

Algorithm 14.3: Finding average number of letters in word

```
Read a line from the input into a buffer;
While there are more lines in input do
begin
  number of lines = no. of lines + 1 ;
  repeat
    Scan buffer from left to right till a blank or
    a . is encountered
    Increment word count
    Accumulate the number of characters per word
  until end of buffer is reached
  Read a line
end
Find average characters per word.
```

Example 14.4

We will now develop a program to convert a Roman numeral to its decimal equivalent. A set of Roman numerals is given one per line. End of the set is indicated by a line with '\n' character only. Assume that the length of a Roman numeral is less than 10. Table 14.4 gives the Roman symbols and their decimal equivalents.

Table 14.4 Roman and Decimal Numerals

Roman	M	C	L	X	V	I
Decimal	1000	100	50	10	5	1

Algorithm 14.4 gives the method of converting a Roman numeral string R_1, R_2, R_3, ..., R_n to decimal.

Algorithm 14.4: Roman to decimal conversion

```
Decimal equivalent = 0;
Read a character from the input string into R1;
V1 = Value of R1
    Read next character into R2;
while (R2 is not end of line character) do
    V2 = Value of R2;
    if(V1 > V2)
      Add V1 to decimal equivalent;
    else
      { Subtract V1 from decimal equivalent;
      Replace R1 by R2;
      Replace V1 by V2; }
    Read next character into R2;
end while
Print decimal equivalent;
```

```
/* Example Program 14.13 */
/* Counting words in sentences. Finding average
   length of words */
#include <stdio.h>
void read_line(char s[]);
int length(char s[]);
void print_line(char out_string[]);
main()
{
  int i, letters, count_word = 0, sum_length = 0,
  average_word_length, no_of_lines = 0;
  char buffer[81];
  read_line(buffer);
  print_line(buffer);printf("\n");
  /* while lines remain in input */
  while(length(buffer) != 0)
    {
      ++no_of_lines;
      printf("no lines %d\n", no_of_lines);
      i = 0;
      letters = 0;
      /* while end of line is not reached */
      while ( buffer[i] != '\0' )
        {
          /* while end of word is not reached */
          if ( (buffer[i] == ' ' ) ||
               (buffer[i] == '.' ) )
            {
              ++count_word;
              sum_length += letters;
              letters = 0;
            }
          else
            ++letters;
          ++i;
        }
      /* last word in a line counted */
      ++count_word;
      sum_length += letters;
      printf("Count wd = %d, sum_len = %d\n",
             count_word, sum_length);
      read_line(buffer);
      printf("buf_len = %d\n", length(buffer));
      print_line(buffer);printf("\n");
    } /* end of outer-most while loop */

  printf("The number of lines = %d\n", no_of_lines);
  printf("Total number of words = %d\n", count_word);
  average_word_length = sum_length / count_word;
  printf("Average length of a word = %d\n",
         average_word_length);
} /* End of main */

/* Place functions length, read_line, print_line
   here */
```

Program 14.13 Finding average word length.

Example Program 14.14 is traced with the input string CMXXIV. The trace is given in Table 14.5.

```
/* Example Program 14.14 */
/* Program follows algorithm given in text */
#include <stdio.h>
int value(char roman);
main()
{
  int decimal_equivalent = 0, dec_1, dec_2;
  char rom_1, rom_2;
  rom_1 = getchar();
  dec_1 = value(rom_1);
  rom_2 = getchar();
  while ((rom_2 != '\n'))
    {
      dec_2 = value(rom_2);
      if (dec_1 >= dec_2)
        decimal_equivalent += dec_1;
      else
        decimal_equivalent -= dec_1;
      dec_1 = dec_2;
      rom_2 = getchar();
    } /* End of while */
  decimal_equivalent += dec_1;
  printf("Decimal equivalent = %d\n",
         decimal_equivalent);
} /* End of main */

int value(char roman)
{
  switch(roman)
    {
    case 'M':  return(1000); break;
    case 'C':  return(100); break;
    case 'L':  return(50); break;
    case 'X':  return(10); break;
    case 'V':  return(5); break;
    case 'I':  return(1); break;
    default:   printf("Error in roman numeral\n"); break;
    } /* End of switch */
}
```

Program 14.14 Conversion roman to decimal.

14.4 INPUT AND OUTPUT OF STRINGS

We saw that strings may be input using a scanf function. In this case the input string must not have a blank character. Thus we cannot read the string such as "Ram Kumar" and store it in an array using scanf function. This string can be read by using getchar ( ) function. It

Table 14.5 Trace of Example Program 14.14

Input string CMXXIV				
rom-1	dec-1	rom-2	dec-2	decimal equivalent
C	100	M	1000	– 100
M	1000	X	10	900
X	10	X	10	910
X	10	I	1	920
I	1	V	5	919
V	5	'\n'		924

is, however, cumbersome to do so as we have to write an explicit loop. C provides a *function* gets ( ) to read a string. For reading a string we write:

```
char buffer[80];
gets(buffer);
```

The *gets* function reads a string terminated by the end of line (carriage return) character '\n' and stores it in buffer. When it stores the string in a buffer it replaces the '\n' character by the end of string character '\0'. If the *gets* function reads a NULL string (just a carriage return without any character in the string) it returns NULL. If it encounters an error then also it returns NULL.

The companion function *puts (buffer)* writes the contents of buffer upto '\0' on to the standard output unit. We illustrate the use of these functions in Example Program 14.15. This

```
/* Example Program 14.15 */
/* Illustrates use of gets and puts functions */
#include <stdio.h>
main()
{
  char buffer[80], n_buffer[80];
  int i, j;
  while(gets(buffer) != NULL)
    {
      printf("String read is :\n");
      puts(buffer);
      i = 0; j = 0;
      while(buffer[i] != '\0')
        {
          if (buffer[i] != ' ')
            {
              n_buffer[j] = buffer[i];
              ++j;
            }
          ++i;
        }
      n_buffer[j] = '\0';
      printf("String with blanks squeezed out\n");
      puts(n_buffer);
    } /* End of while */
} /* End of main */
```

Program 14.15 Use of gets and puts functions.

program reads a string and prints it as it is and also with all blanks in the string removed. This program reads one string after another from the input till it encounters a carriage return with no characters.

C language provides in a library many useful functions for manipulating strings. These functions are given in Appendix IV with an explanation of their formal arguments and what computation they carry out.

EXERCISES

14.1 Write a program to find whether a character string is a Palindrome. A Palindrome reads the same whether it is read from left to right or right to left. (ABBA is a Palindrome).

14.2 Write a program to delete all vowels from a sentence. Assume that the sentence is not more than 80 characters long.

14.3 Write a program to count the number of words in a sentence.

14.4 Write a program which will read a line and squeeze out all blanks from it and output the line with no blanks.

14.5 Write a program which will read a line and delete from it all occurrences of the word 'the'.

14.6 Write a program to encrypt a sentence using the strategy of replacing a letter by the next letter in its collating sequence. Thus every A will be replaced by B, every B by C and so on and finally Z will be replaced by A. Blanks are left undisturbed.

14.7 Write a program which takes a set of names of individuals and abbreviates the first, middle and other names except the last name by their first letter. For example: RAMA RAO would become R. RAO. SURESH KUMAR SHARMA would become S.K. SHARMA.

14.8 Write a program to read a sentence and substitute every occurrence of the word 'POST' by the word 'DAK'. Use the following sentence to test your program.

Input line: "The postman came from the post office bringing post"
Output line: "The dakman came from the dak office bringing dak"

14.9 Write a program to convert a hexadecimal number to decimal. For example the decimal equivalent of

$$\begin{aligned} AC8D &= A * 16^3 + C * 16^2 + 8 * 16^1 + D * 16^0 \\ &= 10 * 16^3 + 12 * 16^2 + 8 * 16 + 13 \\ &= 44173 \end{aligned}$$

14.10 Write a program to convert decimal numbers to hexadecimal. For example the hexadecimal equivalent of 43919 is AB8F.

14.11 Write a program to count the number of occurrences of any two vowels in succession in a line of text. For example, in the following sentence:

"Please allow a studious girl to read behavioural science"

such occurrences are:

ea, io, ou, ea, io, ou, ie.

Observe that in a word such as studious we have counted "io" and "ou" as two separate occurrences of two consecutive vowels.

14.12 Write a program to arrange a set of names in alphabetic order. The sorting is to be on the first two characters of the last name. For example, given the list:

Ramaswamy R.
Arumugam B.
Agarwal K.
Sarma A.B.
Bagchi D.R.

the sorted list should be

Agarwal K.
Arumugam B.
Bagchi D.R.
Ramaswamy R.
Sarma A.B.

15. Enumerated Data Types and Stacks

Learning Objectives

In this chapter we will learn:

1. The definition and use of a data type called enumerated data type
2. How new data type names can be obtained by using a feature called typedef
3. The use of typedef to create a data type named Boolean which is very useful in programming
4. How to create a data structure called stack and define operations necessary to use a stack effectively
5. Applications of stack

C provides a number of data types and structures which are normally not provided in other popular programming languages such as Fortran and Cobol. The motivation for providing these is to ease program development and readability.

The standard scalar data types provided by C are integer, real and character. In this chapter we will first consider another scalar data type known as enumerated scalar data type. The primary use of this data type is to enhance program understandability. A subsidiary advantage is the possibility of automatic checking of input data at execution time.

By a data structure we mean a collection of similar types of data making up a composite. Such a composite or structure has some special properties which make it easy to develop algorithms. The only data structure we have encountered so far is the array structure. We have used arrays to store vectors, strings of characters, tables of values, etc. In this chapter we will introduce a data structure known as a *stack* which has a number of applications in programming. C does not provide this data structure as a standard type. We will simulate this structure with an array structure and define procedures for operations relevant to this structure. We will then illustrate their use with a number of examples.

15.1 ENUMERATED DATA TYPE

A variable declared as an enumerated data type can assume a value defined by a set of identifiers. For example, one may define

```
enum subject {math, phy, chem, ta, esc, hss};
```

which defines a new data type named *subject* which can assume one of the six values specified within braces. The word *enum* is a keyword and is an abbreviation of enumerated. In order to declare a variable name *course* to be of the enumerated *type subject* we write:

```
enum subject course;
```

Observe that we again use the keyword *enum* to specify that the variable name *course* is an enumerated type. The type itself is then stated as *subject*. The identifier *course* cannot assume any value other than that within the enumeration. If it does, an error message is given by the C compiler. There must be a blank space between *enum* and *subject* and another blank between *subject* and *course*. The rules for forming type name and the enumerated identifiers are the same as for any identifier.

Some more examples of enumerated data types are given below:

```
enum card_suit {clubs, diamonds, hearts, spades};
enum card_suit given_card;
enum day_of_week {mon, tue, wed, thu, fri, sat, sun};
enum day_of_week week_day, holiday;
enum colour {violet, indigo, blue, green, yellow, orange, red};
enum colour paint, tile;
enum designation {ldc, udc, asst, suptd};
enum designation employee, staff;
```

We can use the enumerated variable in any statement. For example if there is a variable name *course* of enumerated type *subject* then:

```
if (course==chem)
    credit = 15;
```

will assign 15 to credit if the course happens to be chem. An enumerated variable can also be used in a switch statement as shown below:

```
switch (course)
  { case math : credit = 20; break;
    case chem : credit = 15; break;
    case phy : credit = 18; break;
    case ta : credit = 10; break;
    case esc : credit = 16; break;
    case hss : credit = 9; break;
    default: printf(" error in course value \n");
    break;
  } /* End switch */
```

The enumerated data types are not part of what the hardware provides in a computer such as integers and reals. C includes a set of routines to implement these types. The implementation details need not concern a programmer. The primary idea used in implementation is to assign integer values starting from 0 to the successive components of the enumerated type. Thus, for example, in

```
enum day_of_week {mon, tue, wed, thu, fri, sat, sun};
```

0 will be assigned to mon, 1 to tue, 2 to wed, 3 to thu, 4 to fri, 5 to sat and 6 to sun. This fact allows the use of enumerated variables in statements such as in *switch*, *if* and *for*.

We could declare a variable name such as day_of_week as integer and use 0, 1, 2 etc. for the different days of the week. However, the use of enumerated type and assignment of meaningful name enhances the understandability of C programs.

We will now illustrate the use of enumerated type with an example.

Example 15.1

A college requires a student to take five subjects each semester. The subjects are: mathematics, physics, chemistry, computing and english. Each subject has a specified weightage. At the end of the semester letter grades are assigned to each subject for each student. The letter grades are: A, B, C, D, F and I. It is required to find at the end of each semester the Grade Point Average (GPA) for each student using the following definition:

GPA = Sum of(Points corresponding to letter grade obtained by student * Course weight)/ Sum of (course_weight)

The points for grades A, B, C, D, F are respectively 10, 8, 6, 4 and 2. The point for grade I is 0. This grade is given to a student who has not completed the course. Thus when an I grade is encountered this course should be ignored in GPA calculation, that is, the weight of this course is not added.

We design the program as follows. We first decide how to represent data. As there are a finite number of grades and subjects, we choose enumerated data types to represent these. Thus we define:

```
enum grade {A, B, C, D, F, I };
enum subject {math, phy, chem, comp, english};
enum grade student_grade ;
enum subject course;
```

The weights for each subject and the points for each letter grade are stored in two arrays. An array is convenient to use as a summation with subject as index is needed to find GPA. The values for weights for each subject and points for each letter grade are read as data. This allows the program to accommodate any changes in these values if required.

Each student's performance is given in the form:

roll_no, grades in math, phy, chem, comp and english.

The order of these grades is to be *strictly* followed. One line per student is used in the input. The end of students is indicated by a line with 000 for roll_no. An example data is the following:

```
                3385BACFI
                5452ABICD
                .....          ← other data
Last line       0000XXXXX
```

The strategy used is given in Algorithm 15.1.

Algorithm 15.1

Algorithm to compute student grade

1. Store course weights in an array
2. Store points for grades in another array
3. Read a student's record

4. While student's records remain do

```
{ if (grade != I)
    {total_points = total_points +
                    points for grade * weight of course;
   total_weight = total_weight +
                    course weight;
}
```

5. GPA of student = total_points/total_weight

This algorithm is converted to Example Program 15.1. In Example Program 15.1 observe

```
/* Example Program 15.1 */
/* Student grade example - Use of enumerated
   data-type */
#include <stdio.h>
main()
{
  enum grade {A, B, C, D, F, I};
  enum subject {phy, chem, math, comp, english};
  enum subject course;
  enum grade stud_grade;
  int roll_no, total_points = 0, total_weight = 0,
  weight[5], points[6], error_code;
  float grade_point_avge;
  char temp;
  for (course = phy; course <= english; ++course)
    scanf("%d", &weight[course]);
  for (stud_grade = A; stud_grade <= I; ++stud_grade)
    scanf("%d", &points[stud_grade]);
  /* A typical student record is 3683ABDIC */
  scanf("%d", &roll_no);
  printf("%d  ", roll_no);
  while(roll_no != 0)
    {
     for (course = phy; course <= english; ++course)
      {
        temp = getchar();
        putchar(temp);
        switch(temp)
          {
           case 'A': stud_grade = A; break;
           case 'B': stud_grade = B; break;
           case 'C': stud_grade = C; break;
           case 'D': stud_grade = D; break;
           case 'F': stud_grade = F; break;
           case 'I': stud_grade = I; break;
             default : printf("Error in grade\n");
             error_code = 1; break;
          } /* End of switch */
```

```
        /* If there is error in input goto next record */
        if (error_code == 1)
          continue;
        if (stud_grade != I)
          {
            total_points += points[stud_grade] *
             weight[course];
            total_weight += weight[course];
          }
       } /* End of for loop which processes one
            student's grade */

      grade_point_avge = (float)total_points /
       (float)total_weight;
      printf(" GPA = %4.2f\n", grade_point_avge);
      /* end of processing one student's record */
      scanf("%d", &roll_no);
      printf("%d  ", roll_no);
      total_points = 0; total_weight = 0;
    } /* End of while */
  printf("End of Processing all student records\n");
} /* End of main */
```

Program 15.1 Students grade calculation.

that an enumerated variable name is used as the index in a *for* loop. This is allowed as enumerated types are internally represented by the C compiler as integers. Observe that a data record is typed in the form:

3683ABDIC

No blank should separate 3683 from ABDIC. There are no blanks separating A from B, B from D etc.

When the record is read by the statement:

scanf("%d", &roll_no);

the integer 3683 is stored in roll_no and the scanning of input line stops. The next statement in the *for* loop

temp = getchar ( ) ;

reads the first character from the input, A in this case and stores it in temp. It is important to understand that getchar( ) stores A as a character in temp. In other words temp now contains 'A'. The symbol A in the enumerated data type grade is not to be confused with the character 'A'. A in the enumerated type is a code for 0. Thus we need the *switch* statement to assign the enumerated data type values to stud_grade. It would be an *error* to write:

stud_grade = temp ;

as temp is of type *char* and stud_grade is an enumerated type.

The *for* loop:

for (course = phy; course <= english; ++course)

First starts with course = phy, finds the grade obtained by the student in this course calculates points obtained by the student by multiplying the points for the course by the weight of the course and adds it to total_points. It then accumulates the weights of course. This is repeated for the other courses taken by the student. Observe that the order in which the grades are typed in the input line must strictly follow the order phy, chem, math, comp and english which is the order used in enumeration.

After the *for* loop is complete, the next student's record is read and processed. This is repeated till the last record is reached. The last record has 000 as roll_no which is used as the termination condition in the while loop.

15.2 CREATING NEW DATA TYPE NAMES

We can create new data type names in C language using a facility called *typedef*. For example, the declaration

```
typedef int Boolean ;
```

makes the name Boolean a synonym of int. The type Boolean can be used in declarations in exactly the same way as int. If we declare

```
Boolean end_of_file, error_found ;
```

then end_of_file and error_found are actually int.

Similarly the declaration

```
typedef char Letter ;
```

makes Letter a new type name.

We will use typedef later in the book to define more complex data types. Observe that we have used capital letter as the first letter of data type name declared using typedef to distinguish it from built-in types.

It must be understood that typedef declaration does not add any new data type; it merely renames an existing data type which is useful in program understanding and in program portability.

We will illustrate the use of typedef in Example Program 15.2.

Example 15.2

Given the date of birth of an undergraduate student in the form:

```
day    month    year
```

for example, 250370,

a program is required to check if the date is a "reasonable" date and if it is so, then write the date in the expanded form:

```
Day    Name of the month    Year
```

For example, write 25 MARCH 1970 corresponding to the input date 250370. We will use the following criteria to determine "reasonableness":

1. If the month is January, March, May, July, August, October or December then 1 <= day <= 31.

```
/* Example Program 15.2 */
#include <stdio.h>
#define TRUE 1
#define FALSE 0
typedef int Boolean;
Boolean leap_year(int year);
Boolean day_ok_for_month(int day, int month, int year);
void print_date(int day, int month, int year);
int day_for_month[12] = {31, 28, 31, 30, 31, 30, 31,
                         31, 30, 31, 30, 31};
main()
{
  int date, day, month, year, approx_age, current_year,
  roll_no, upper_age, lower_age;
  Boolean error_found;
  scanf("%d %d %d", &current_year, &upper_age, &lower_age);
  while(scanf("%d %d", &roll_no, &date) != EOF)
    /* date in the form ddmmyy e.g  050791 */
    {
      year = date % 100;
      month = (date/100) % 100;
      day = date/10000;
      approx_age = current_year - year;
      if ((approx_age > upper_age) ||
          (approx_age < lower_age))
        {
          error_found = TRUE;
          printf("Error in age, roll_no = %d\n", roll_no);
          continue;
        }
      if ((month > 12) || (month < 1))
        {
          error_found = TRUE;
          printf("Error in month, roll_no = %d\n", roll_no);
          continue;
        }
      if ((day <= 31) && (day >= 1))
        day_ok_for_month(day, month, year);
      else
        {
          error_found = TRUE;
          printf("Error in day of month, roll %d\n",
                  roll_no);
          continue;
        }
      if (! day_ok_for_month)
        {
          error_found = TRUE;
          printf("Error in day of month, roll_no = %d\n",
                  roll_no);
          continue;
        }
      printf("%d ", roll_no);
      print_date(day, month, year);
    } /* End of while */
} /* End of main */
```

```
Boolean day_ok_for_month(int day, int month, int year)
{
  if (day <= day_for_month[month])
    return(TRUE);
  else
    if ((month == 2) && leap_year(year) && (day <=29))
      return(TRUE);
    else
      return(FALSE);
} /* End of day_ok_for_month */
Boolean leap_year(int year)
{
  if ((year % 100 == 0) && (year % 400 == 0))
    return(TRUE); /* it is a leap year */
  else
    if (year % 4 == 0)
      return(TRUE); /* it is a leap year */
    else
      return(FALSE); /* it is not a leap year */
} /* End of leap-year */
void print_date(int day, int month, int year)
{
  printf("%2d ", day);
  switch(month)
    {
    case 1:
      printf("January"); break;
    case 2:
      printf("February"); break;
    case 3:
      printf("March"); break;
    case 4:
      printf("April"); break;
    case 5:
      printf("May"); break;
    case 6:
      printf("June"); break;
    case 7:
      printf("July"); break;
    case 8:
      printf("August"); break;
    case 9:
      printf("September"); break;
    case 10:
      printf("October"); break;
    case 11:
      printf("November"); break;
    case 12:
      printf("December"); break;
    } /* End of switch */
  printf(" %d\n", year);
} /* End of print_date */
```

Program 15.2 Checking date of birth of a student.

2. If the month is April, June, September or November then 1<= day <= 30.

3. If the month is February then 1 <= day <= 29 if the year is a leap year. If the year is divisible by 100 then it should be divisible by 400 for it to be a leap year. For example the year 1900 is *not* a leap year but the year 2000 is a leap year.

Example Program 15.2 implements the date checking program. Observe that the input data is typed in the form

4642 250370

with a blank between roll_no and date.

The program first reads the current_year which is given as data. It then reads the first student record. The end of student records is indicated when the end of file is reached.

The *while* loop does the processing of one record. The first *if* statement in the *while* loop checks whether the year of birth of student is reasonable. The criterion used is that a student in a college must be between the upper and lower age limits set by the college authorities. If the student's age is not within these bounds then the Boolean variable error_found is set to TRUE and an error message is printed. As a *continue* statement is the next one, the rest of the statements are not executed and control returns to *while* to process the next record.

After this, the value given for month is checked. If it is not correct an error message is again given and control returns to process the next record. The checking of day value is a little more complicated as the day value depends on the month and the year. The rules given in the statement of the problem are used in the function day_ok_for_month (day, month, year). Observe that this function uses another function for finding if the year is a leap year. The maximum days in each month is stored in the global array day_for_month. As this data is invariant it is initialized and stored as a global array. This array is used by the function day_ok_for_month to check whether the day exceeds the maximum allowed days for a month. If the month is February, then the leap_year function is invoked to check if day < 28 for a non leap year. Finally the function print_date is invoked to print the date in the format required in the problem specification.

15.3 A STACK

In an array structure any element may be retrieved by specifying the appropriate subscript. We may also store an element anywhere in the array. As opposed to this, *a stack* is a structure with a group of cells in which only the top cell is accessible. Thus only the element at the top of the stack may be retrieved. Any new element to be stored in the stack is also stored at the top. Figure 15.1 illustrates this.

The store operation is called *Push* and the retrieve operation *Pop*. Because of the nature of storing and retrieving elements at the same end of the stack the element accessed is the last one stored. Hence a stack is also known as a *last-in-first-out(LIFO)* store.

One may visualize the operation of a stack with the following analogy. Imagine a dish washer in a hotel washing plates and stacking, that is, placing them one on top of another. When a waiter wants to take a plate to serve a customer he will take a plate from the top of the stack. This plate is the most recently washed plate. When the cleaner washes another plate he will place it on top of the stack.

Given the stack of Fig. 15.1(b) if we execute three Pop operations the elements will be retrieved (removed) from the stack in the order 'A', 'X' and '+'. If we now Push ('P'), Push ('Q') the element 'Q' will be inserted on top of the stack. A Pop will now remove 'Q'. This

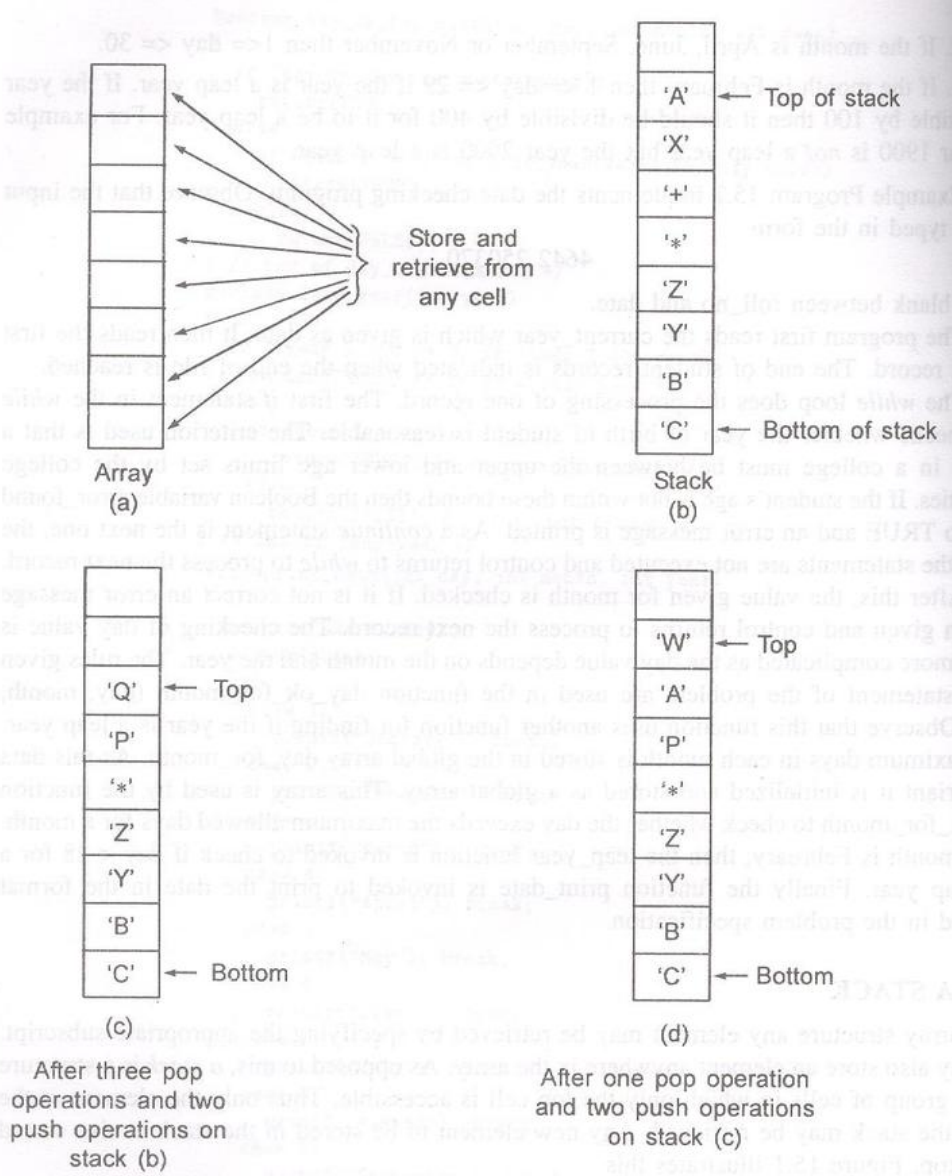


Fig. 15.1 Array and stack data structures.

last-in-first-out retrieval property of a stack is extremely useful in a number of programming situations. These situations arise in programming language translation such as evaluating arithmetic expressions using arithmetic operator precedence rules, implementing procedure calls and returns, etc.

We will illustrate the use of a stack with an example in the next section.

15.4 SIMULATION OF A STACK

In C the operations on a stack can be simulated using an array. In this section we will develop declarations to implement a stack and functions to simulate the operations Push and Pop on a stack.

A stack is declared as an array with an index which will point to the top of the stack. We will assume that each cell of the stack stores a character. (In general a cell can store anything that can be stored in a cell of an array). We will also declare variables of type Boolean to indicate whether the stack is full or empty. The declarations are given as Example Program 15.3 and, are global.

```
/* Example Program 15.3 */
/* These are defined as global outside main() */
#define STACK_SIZE   80
#define FALSE        0
#define TRUE         1
char stack[STACK_SIZE];
typedef int Boolean;
Boolean stack_empty, stack_full;
int top_of_stack;

/* top_of_stack is the array index of stack[]. It specifies
   from where a data can be stored or retrieved */

/* Functions defining push and pop operations for a stack */

void push (char pushed_char)
{
  if (stack_full)
    {
      printf("Error - Stack is full\n");
      return;
    }
  else
    {
      ++top_of_stack;
      stack[top_of_stack] = pushed_char;
      stack_empty = FALSE;
    }

  if (top_of_stack == STACK_SIZE)
    stack_full = TRUE;
} /* End of push */
char pop()
{
  char temp;

  if (stack_empty)
    {
      printf("Error - Stack is empty\n");
      return(' ');
    }
  else
    {
      temp = stack[top_of_stack];
      --top_of_stack;
      if (top_of_stack <= 0)
        stack_empty = TRUE;
      return(temp);
    }
} /* End of pop */
```

Program 15.3 Stack declaration and operations.

We will simulate the operation Push by the function push. The input to the function is the pushed character. The current index giving top_of_stack and status of stack, namely, whether it is full or empty are available through global variables. The function first checks if the stack is already full. If it is full it gives a message that the pushed character cannot be stored and aborts the procedure. If the stack is not full then it increments the top_of_the_stack index by 1 and stores the pushed character at the top of the stack. As the stack now contains a character, stack_empty is assigned a value FALSE. The pointer value is now compared with its maximum allowed value determined by the stack size. If it is above this value then the variable stack_full is assigned a TRUE value.

The operation Pop from a stack is simulated by function pop. The popped character is the output of this function. The function first checks if the stack is empty. If it is empty then nothing can be retrieved. A message is given and the function is aborted. If the stack is not empty then a character from the top of the stack is popped. The index top_of_stack is decreased by 1 to point to the 'new' top element in the stack.

As a character has been popped the stack can be empty. The index value is compared with its minimum allowed value which is one. The minimum value is 1 because a push increments top_of_stack before storing data in stack. If it is below 1 then the variable stack_empty is assigned a TRUE value.

15.5 APPLICATIONS OF STACK

In this section we will consider as the first example the translation of an algebraic expression written in the usual notation (namely with arithmetic operators appearing between operands and using parentheses to indicate precedence of operations) into another form in which operators follow the operands on which they operate and no parentheses are used. Consider, for example, the expression:

$$(A - B) * C$$

This is the usual notation and is known as the *infix* notation. In this notation operators appear between the operands on which they operate. Parentheses impose a precedence of operation.

The same expression may be written in what is known as *Polish postfix* notation as shown below:

$$AB - C*$$

This notation is called Polish as it was developed by the Polish logician J. Lukasiewicz and post_fix as the operators appear after the operands. In this notation we interpret the expression by scanning it from left to right. The minus sign is applied to the two operands immediately preceding it. After carrying it out we continue scanning till we reach the next operator. When it is found it is applied to the two quantities preceding it. In this case the two quantities are A – B and C to which the multiplication operator * is applied giving (A – B) * C.

Consider the Polish postfix expression

$$ABC - /$$

Scanning from left to right, we stop when we reach – and apply it to the two immediately

preceding operands obtaining (B – C). Continuing the scan we encounter/ and apply it to A and (B – C) getting A/(B – C). We give below some more examples of infix and postfix expressions:

Infix expression	*Postfix expression*
A + B – C + D	AB + C – D +
(A + B)/(C + D)	AB + CD + /
A + B * C – D	ABC * + D –
A + B * (C + D * (E + F))	ABCDEF + * + * +
(A/B) * (C * F + (A – D) * E)	AB/CF * AD – E * + *

The Polish postfix form is very important as this form is used by compilers to evaluate expressions.

In this section we will first write a program to do the infix to postfix conversion. We have made the following assumptions:

(i) Only capital letters are allowed as variable names

(ii) No constants are allowed in the expression

(iii) We restrict ourselves to the four arithmetic operators +, –, * and/. Unary – is excluded.

(iv) We assume that there are no errors in the infix expression.

We have made the above assumptions to present the essence of the method without getting lost in details.

The conversion method assigns a precedence or rank to each operator. This determines where an operator will enter the postfix expression. This is necessary to implement the convention that multiplications and divisions are performed before additions and subtractions and to account for parentheses if they are present in the infix expression. The precedence numbers are:

Operators	* /	+ –	(	)
Precedence numbers	4	3	2	1

The method uses a stack described in the last section. The method of conversion from infix to postfix expression is as follows:

1. Scan the infix expression from left to right.

2. If the scanned character is an operand enter it in the postfix expression. If it is not an operand then push it into the stack. Before pushing it in, however, the precedence of the operator at the top of the stack is compared with that of the present operator. If the operator at the top of the stack has a higher or same precedence as the present operator, the operator at the top of the stack is popped and appended to the postfix expression. This is repeated and more operators are popped out of the stack till the operator at the top of the stack has a lower precedence.

3. Whenever a right parenthesis is found in the infix expression it is ignored. At this time the stack will invariably have at its top the matching left parenthesis. This is popped out of the stack and ignored.

In developing the program we first do the following:

1. Use the functions developed in the last section to simulate the push and pop operations in a stack.

2. Write a function to assign precedence to the operators.

3. The infix expression is stored in an array and scanned from left to right.

The algorithm may be briefly written as Algorithm 15.2.

Algorithm 15.2

Algorithm to convert infix expression to postfix expression.

Given: An array which stores the string corresponding to the infix expression. Initially stack is empty. Stack pointer initialized to 0.

1. Scan infix_string from left to right one character at a time.
2. While characters are left in the infix_string

```
{Switch (scanned_character)
  letter: Place scanned_char in postfix string;
          break;
  operator:
          if stack is empty
             push the scanned character in the stack
             break;
          else
             repeat
                Pop character from stack
                   if the precedence of the character on top of stack is higher than
                      that of the scanned character
                   then
                      place the character from top of stack in the postfix string
             until stack is empty or the precedence of the popped character from
                      stack becomes lower than that of the scanned character
          if precedence of character in stack is low
             Push character popped from stack back into it.
             Push scanned character into the stack.
          else
             Push scanned character into the stack.
             break;
  left_paren:
             Push scanned character into stack;
             break;
  right_paren:
             Pop characters from stack and put them in post_fix string till matching
                left parenthesis is found.
             break;
} /* End of while */
```

3. Pop out any operators left in stack and append them to the post_fix string.

A program corresponding to this algorithm is given as Example Program 15.4. In this program we have used the stack declarations and the push and pop functions of Example

```
/* Example Program 15.4 */
/* Conversion of infix to postfix expression */
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#define TRUE         1
#define FALSE        0
#define STACK_SIZE   80
char stack[STACK_SIZE];
typedef int Boolean;
Boolean stack_empty, stack_full;
enum symbol_type {letter, operator, left_paren, right_paren};
int top_of_stack;

char pop(void);
void push(char g_ch);
int precedence(char s);
enum symbol_type nature_of(char sc_ch);

main()
{
  int i, k, in_str_length, len_postfix;
  char scanned_char, infix_string[80], postfix_string[80],
  stack_char;
  Boolean stack_prec_low, match_found, error_flag;
  enum symbol_type {letter, operator, left_paren, right_paren};

  /* Initialise variables */

  stack_prec_low = TRUE;
  stack_empty = TRUE;
  stack_full = FALSE;
  top_of_stack = 0;    /* Stack empty */
  gets(infix_string);
  in_str_length = strlen(infix_string);
  scanned_char = infix_string[0];
  i = 0; k = 0;

 while ( i < in_str_length )
  {
    switch (nature_of(scanned_char))
      {
      case letter:
        postfix_string[k] = scanned_char;
        ++k;
        break;

      case operator:
        if (stack_empty)
```

```
        {
          push(scanned_char);
          break;
        }
      else
        {
          do
            {
              stack_char = pop();

              if (precedence(stack_char) >=
                  precedence(scanned_char))
                {
                  postfix_string[k] = stack_char;
                  ++k;
                  stack_prec_low = FALSE;
                }
              else
                stack_prec_low = TRUE;
            }
          while ((!stack_prec_low) && (!stack_empty));
        }

      if (stack_prec_low)
        {
          push(stack_char);
          push(scanned_char);
        }
      else
        push(scanned_char);
      break;  /* End of case: operator */

    case left_paren:
      push(scanned_char);
      break;

    case right_paren:
      do
        {
          stack_char = pop();
          if (stack_char == '(')
            match_found = TRUE;
          else
            {
              postfix_string[k] = stack_char;
              ++k;
              match_found = FALSE;
            }
        } /* End of do while */
      while ((!match_found) && (!stack_empty));

      if (!match_found)
```

```
    {
      printf("Matching left parenthesis not found\n");
      error_flag = TRUE;
      exit(1);
    }
  break; /* End of case: right_paren */

        default:
          error_flag = TRUE;
          printf("Error in scanned character\n");
          exit(1);
          break;
        } /* End of switch statement */

      ++i;
      scanned_char = infix_string[i];
}
  /* End of while (scanned_char != '\0') */

  /* Input expression scanning is now over */
  /* Take out all the operators left in the stack */

  while (!stack_empty)
    {
      stack_char = pop();
      postfix_string[k] = stack_char;
      ++k;
    } /* End of while */

  postfix_string[k] = '\0';
  /* Length of postfix string is k-1 */
  len_postfix = k-1;
  printf("Input Expression\n");

  for (i = 0; i <= in_str_length; ++i)
    putchar(infix_string[i]);
  putchar('\n');
  printf("Postfix Expression\n");

  for (i = 0; i <= k-1; ++i)
    putchar(postfix_string[i]);
  putchar('\n');
} /* End of main */
  enum symbol_type nature_of(char scanned_char)
  {
    if (isupper(scanned_char))
      return(letter);

    if ((scanned_char == '-') || (scanned_char == '+') ||
        (scanned_char == '*') || (scanned_char == '/'))
      return(operator);
```

```
    if (scanned_char == '(')
      return(left_paren);

    if (scanned_char == ')')
      return(right_paren);
  } /* End of nature_of (scanned_char) */


  int precedence(char given_char)
  {
    switch(given_char)
      {
      case '*': return(4); break;
      case '/': return(4); break;
      case '+': return(3); break;
      case '-': return(3); break;
      case '(': return(2); break;
      case ')': return(1); break;
      default: printf("Error in given_char\n");
        break;
      } /* End of switch */
  } /* End of precedence */

  /* Include functions char pop(void) and
     void push(char s) here */
```

Program 15.4 Conversion of infix expression to postfix expression.

Program 15.3. We have used a function nature_of (char scanned_char) to classify the characters in the infix string as letter, operator, left_paren or right_paren.

The precedence is also determined in a simple function which implements the operator precedence table using a *switch* statement. In the *while* loop in the main function, actions appropriate to each character type in infix_string as stated in Algorithm 15.2 are implemented. The rest of the program is self-explanatory.

Having obtained a postfix expression corresponding to an infix expression we will now develop a program to carry out the arithmetic operations to evaluate the value of the arithmetic expression. A stack is used again to aid the evaluation procedure. The stack will now be used to store real numbers rather than characters. Given a postfix expression, Algorithm 15.3 is a procedure to evaluate the value of the expression.

Algorithm 15.3: Evaluates an arithmetic expression from a postfix expression

1. Store in an array the values (real numbers) corresponding to each variable name.

While characters remain in postfix string do.
{ Scan the postfix string from left to right. If a scanned character is a letter (namely, a variable name) then push its *value* into a stack.

If the scanned character is an arithmetic operator then pop stack. Assign the value popped to operand 2. Pop the stack again and assign the value popped to operand 1. As operands are popped out from stack in "last-in-first-out" discipline the first value popped is operand 2 and the next one is operand 1.

If operator is:

* : Accumulator = operand 1 * operand 2;
/ : Accumulator = operand 1 / operand 2;
+ : Accumulator = operand 1 + operand 2;
– : Accumulator = operand 1 – operand 2;

Push result in Accumulator into stack;
} End of while

For example, consider the infix expression (A – B) * C. The corresponding postfix expression is AB – C*. If the values of A, B, C are respectively 2.5, 1.4 and 3.0 then applying algorithm 15.3 we obtain:

1. A = 2.5, B = 1.4, C = 3.0

2. Scanning AB – C* we encounter first A. Thus the value of A, namely, 2.5 is pushed into stack. Next B is encountered and its value 1.4 is pushed in the stack.

3. The next character scanned is an arithmetic operator, namely, –. The stack is popped and the value assigned to operand 2. Thus operand 2 = 1.4. The stack is popped again and value popped assigned to operand 1. Thus operand 1 = 2.5.

4. As the operator is – we compute

Accumulator = operand 1 – operand 2
= (2.5 – 1.4) = 1.1

5. 1.1 is pushed into stack.

6. The next character scanned is C. Thus its value 3.0 is pushed into stack.

7. The next character scanned is *. The stack is popped and operand 2 = 3.0. The stack is popped again and operand 1 = 1.1

8. As the operator is * we compute

Accumulator = operand 1 * operand 2
= 1.1 * 3.0 = 3.3

9. The algorithm now terminates as there are no more characters in the postfix expression.

Example Program 15.5 implements Algorithm 15.3. The function store_values reads and stores in a memory array values of variable names given as input in the form:

variable name real number variable name real number

For example: A 2.5 B 1.4 C 3.0

The stack and operations of Push and Pop are simulated by push_num and pop_num. In this example the cells in the stack store real numbers. The *function* calculate implements steps in the while loop of Algorithm 15.3. Observe that memory [ ] where values of A, B etc. are stored is global.

There are many more applications of stacks in programming. We will discuss some more applications later in this book.

```
/* Example Program 15.5 */
/* Program to calculate an expression from postfix notation */
/* Postfix string is given as one input line values
   of identifiers given in next line
   <identifier> blank <value> blank <identifier> <blank> ...
   This line terminated by a *
   For example A 25 B 1.4 C 3.0 */
#include <ctype.h>
#include <string.h>
#include <stdio.h>
#define TRUE      1
#define FALSE     0
#define STACK_SIZE  80

float num_stack[STACK_SIZE];
int num_st_top;
typedef int Boolean;
Boolean num_st_empty, num_st_full;
float memory[26];

void read_values(void);
float calculate(char postfix[]);
float pop_num(void);
void push_num(float num);

main()
{
  float result;
  char postfix_string[80];
  /* This reads the given postfix string */
  gets(postfix_string);
  /* This reads the second line in input and assigns
     values to identifiers and stores in mem */
  read_values();
  result = calculate(postfix_string);
  printf("The result of evaluation = %f\n", result);
} /* End of main */
  /* Values typed in the form   A 2.5 B 1.4 C 3.0 */
  void read_values()
  {
    float value;
    char temp;
    int location;
    scanf("%c %f", &temp, &value);
    while(temp != '*')
      {
        location = temp - 65;
        memory[location] = value;
```

```
    /* ASCII code of A is 65. If A is read its value is
       stored ASCII code for B, C, D, .. are 66, 67,
       68, .. respectively. If P is read, its location
       is 80 - 65 = 15 in memory code of Z = 90 maximum
       value of location = 25 */
       scanf("%c %f", &temp, &value);
     } /* End of while loop */
} /* End of read_values */

float calculate(char postfix_exp[])
{
  float operand_1, operand_2, acc;
  int i = 0, j;
  char scanned_char;
  while (i < strlen(postfix_exp))
    {
      scanned_char = postfix_exp[i];
      if (isupper(scanned_char))
        {
          j = scanned_char - 65;
          push_num(memory[j]);
        }
      else
        {
          operand_1 = pop_num();
          operand_2 = pop_num();
          switch(scanned_char)
            {
            case '*':
              acc = operand_1 * operand_2;
              break;
    /* Remember values in stack is pulled from top */
            case '/':
              acc = operand_2 / operand_1;
              break;
    /* Remember values in stack is pulled from top */
            case '-':
              acc = operand_2 - operand_1;
              break;
            case '+':
              acc = operand_1 + operand_2;
              break;
            default:
              printf("Incorrect operator symbol\n");
              exit(1);
              break;
            } /* End switch */
          push_num(acc);
```

```
        } /* End of else part of if */
        ++i;
      } /* End of while */
    return(pop_num());
  } /* End of calculate */

  float pop_num()
  {
    float temp;
    if (num_st_empty)
      {
        printf("Error - Stack is empty\n");
        exit(-1);
      }

    else
      {
        --num_st_top;
        if (num_st_top <= 0)
          num_st_empty = TRUE;
        temp = num_stack[num_st_top];
        return(temp);
      }
  } /* End of pop */

  void push_num(float num_in_st)
  {
    if (num_st_full)
      {
        printf("Error - Numeric stack full\n");
        exit(-1);
      }
    else
      {
        num_stack[num_st_top] = num_in_st;
        ++num_st_top;
        num_st_empty = FALSE;
      }
    if (num_st_top == STACK_SIZE)
      num_st_full = TRUE;
  } /* End of push_num */
```

Program 15.5 Calculating value of a postfix expression.

EXERCISES

15.1 Rewrite Example Program 12.3 (which illustrates a tabulation technique) using enumerated scalar data type.

15.2 Rewrite Example Program 12.4 using enumerated scalar data types to represent various responses to questions. Discuss the advantages and disadvantages of this approach as compared to the approach used in Chapter 12.

15.3 A Company pays normal wage for work during week days from Monday to Friday and 1.5 times wage for work on Saturday and Sunday. Given data in the following form:

Employee Number, Wage/hour, hours worked on Monday, hours on Tuesday, ..., hours on Sunday.

Write a program to write out the Employee number and weekly wage. Use enumerated data type in your program.

15.4 Write a program to check whether a given string is a valid real number. Use the syntax rules for real numbers given in Chapter 5.

15.5 In the Fortran 77 language variable names are defined as follows:

Integer variable name: A string of up to 6 alphanumeric characters. The first character must be I, J, K, L, M or N. The other characters may be either letters or digits.

Real variable name: A string of up to 6 alphanumeric characters. The first character may be any letter except I, J, K, L, M, N. The other characters may be any of the 26 letters or digits. Write a program to check whether a given string is a valid integer variable name, real variable name or valid as a variable name.

15.6 Write a program to read a line of text and delete all the vowels from it.

15.7 Write a program to determine if an input character string is of the form X*a*Y. X is a string of arbitrary length using only the characters A and B. For example, X may be ABBAB. Y is a string which is the reverse of X. Thus for the string X given above Y is BABBA. *a* is any arbitrary character which is not A or B. For example given ABAACABAB the program should write a message that this string is invalid. For the string ABBABCBABBA the program should write a message that it is valid. Use a stack.

15.8 Use the Push and Pop operations on a stack to do the following:

(i) Assign to a variable name Y the value of the third element from the top of the stack and keep the stack undisturbed.

(ii) Given an arbitrary integer *n* pop out the top *n* elements. A message should be given if an unusual condition is encountered.

(iii) Assign to a variable name Y the value of the third element from the bottom of the stack and keep the stack undisturbed.
(*Hint* : You may use a temporary stack in this case)

15.9 Transform the following expressions to postfix form:

(i) (A + B) * (C – D)/(E – F)

(ii) A – B/C – D/(E – F)

(iii) (A – B)/C * ((X – Y)/(X + Y))

15.10 Transform the following postfix expression to infix form:

(i) AB + C –

(ii) ABC * –

(iii) AB – C * DEF – * –

15.11 Write a program to read a postfix string and convert it to the infix form. The infix form should use parentheses as needed.

15.12 Write a program to read a parenthesised infix expression and do the following:

(i) Check if the left and right parentheses match.

(ii) If they match then remove all excess superfluous parentheses and create an infix string with minimum number of parentheses.

15.13 A queue is a data structure in which a new element is inserted at the back of the queue and an element is retrieved from the front (the other end) of the queue. For example, given a queue,

ABCDEFXYZ

back front

a retrieval operation will retrieve Z. An insert operation will insert an element behind A. Write a program to:

(i) Define a data structure for a queue of characters using an array.

(ii) Write functions to add an element to a queue and to retrieve an element from the queue.

The functions must have parameters to indicate queue full, queue empty conditions and set pointers for adding and retrieving elements.

15.14 A *dequeue* is an ordered set of elements in which elements may be inserted or retrieved from either end. Using an array simulate a *dequeue* of characters and the operations *retrieveleft, retrieveright, insertleft, insertright.* Exceptional conditions such as dequeue full or empty should be indicated. Two pointers (namely, left and right) are needed in this simulation.

16. Structures

Learning objectives

In this chapter we will learn:

1. Representation of related data items as a structure
2. Applications of structures
3. Use of arrays in structures

There are many applications where it is convenient to treat related data items as one entity. Individual components of such an entity would usually be of different data types. Such an entity is called a *structure* and the related data items used in it are called its *components*. For example, consider items in a store. Each item in the store may be described by its code, current stock and its price. Each item can be described by the definition

```
struct item_in_store
    {   int item_code;
        int qty_in_stock;
        float price;
    };
```

The above definition is called a *structure definition*. In this definition a name is given to the structure. The name of the above structure is item_in_store. It is similar to the name we gave to enumerated scalar data type in the last chapter. The name tells the compiler that three locations in memory should be reserved for each data of *type* item_in_store. The first location will store an integer, the second location also an integer and the third a floating point number.

The following declaration declares soap and hair_oil to be of *type*: item_in_store.

```
struct item_in_store soap, hair_oil;
```

We can use typedef to give a name to a structure. For example,

```
typedef struct date
    {   int day;
        int month;
        int year;
    } Date;
Date birth_day, marriage_day;
```

The use of typedef names the structure as Date and declares birth_day and marriage_day as the structure specified. Observe that date and Date are different names as C distinguishes between upper and lower case letters. Observe that in the name of structure given by typedef

we use an upper case letter as the first letter. This is a good practice to distinguish it as a structure name.

```
typedef struct comp_no
    {   float real_part;
        float imag_part;
    } Complex_number;
Complex_number x, y;
```

We have defined above a structure Complex_number which has two components, one called real_part and the other imag_part. It declares the variable names *x* and *y* to be of *type* Complex_number.

```
typedef struct isbn_code
    {   int country_code;
        int publisher_code;
        int serial_no;
        char check_char;
    } ISBN;
ISBN pascal_book, fortran_book;
```

The above definition defines structure isbn_code as one which has 4 components, three of which are of *type int* and the fourth of *type char*. It names the structure ISBN.

The variable names pascal_book, fortran_book are declared to be type ISBN.

The general syntax of structure definition is:

```
struct <structure_identifier>
    {   type_name <identifier>;
        type_name <identifier>;
        .
        .
        type_name <identifier>;
    };
```

Observe that semicolons must terminate the definition of each component identifier of the structure. A semicolon *must* also be present after the right braces which closes the struct definition.

The syntax of declaration of a variable_name as of *type* structure is:

```
struct <structure identifier> <variable identifier>;
```

The syntax rules for structure identifier and variable identifier are the same as for any identifier. The types used within a structure definition may be any *type*: int, float, char, array, enum, and even struct.

16.1 USING STRUCTURES

Let us consider the variable name soap which is declared to be of *type* struct item_in_store. If we want to find the price of soap we use the notation: soap.price. Similarly the item_code and qty_in_stock of soap may be found by using the notation soap.item_code, soap.qty_in_stock. In fact the storage assignment for storing information about soap is shown in Table 16.1.

Table 16.1 Storage of struct soap item_in_store.

soap	soap.item_code	2542	int
	soap.qty_in_stock	95	int
	soap.price	3.50	float

In general the syntax is:

struct_variable name.struct component identifier

No blank should be left before and after the .(dot) which is used between the variable name and its component identifier. For example:

soap. item_code

is an error as blank spaces are there. The correct specification is:

soap.item_code

If suppose a shop stocks a number of brands of soap and we want to find the number of different brands of soaps and the total value of soaps stored in the shop, it is done by Example Program 16.1.

```
/* Example Program 16.1 */
/* Illustrates use of struct declaration and use */
#include <stdio.h>
typedef struct item_in_store
{
  int item_code;
  int qty_in_stock;
  float price;
} Item;

main()
{
  Item soap;
  int no_of_brands = 0;
  float total_soap_value = 0;
    while (scanf("%d %d %f", &soap.item_code,
              &soap.qty_in_stock,
              &soap.price) != EOF)
    {
      total_soap_value += soap.qty_in_stock *
       soap.price;
      ++no_of_brands;
    } /* End of while */

  printf("Number of brands of soap in store = %d\n",
        no_of_brands);
  printf("Total value of soaps stocked = Rs %f\n",
        total_soap_value);
} /* end of main */
```

Program 16.1 Structure definition and use.

Observe in Example Program 16.1 that the structure definition follows #include and before main( ). As the structure definition is a *type* definition it is usually global. In the main program we declare soap as of *type* Item. Remember that Item is a synonym for struct item_in_store. Observe that we read each component of the structure soap in the scanf statement with appropriate format declaration. The *while* loop is self-explanatory.

We can use structures in functions. We illustrate this in Example Program 16.2. These

```
/* Example Program 16.2 */
/* Defining a structure for complex numbers */
#include <stdio.h>
typedef struct complex_number
{
  float real;
  float imag;
} Complex;
Complex sum(Complex m, Complex n);
Complex product(Complex m, Complex n);
Complex quotient(Complex m, Complex n);

main()
{
  Complex x, y, z, p, q, r, d, e, f;
  scanf("%f %f %f %f", &x.real, &x.imag, &y.real,
       &y.imag);
  /* Sum of Complex numbers x and y */
  z = sum(x, y);
  printf("Sum of complex numbers x and y \n");
  printf("%f + i%f\n", z.real, z.imag);
  scanf("%f %f %f %f", &p.real, &p.imag, &q.real,
       &q.imag);
  /* Product of complex numbers p and q */
  r = product(p, q);
  printf("Product of complex numbers p and q\n");
  printf("%f + i%f\n", r.real, r.imag);
  scanf("%f %f %f %f", &d.real, &d.imag, &e.real,
       &e.imag);
  /* Quotient of complex numbers d and e */
  f = quotient(d, e);
  printf("Quotient of complex numbers d and e \n");
  printf("%f + i%f\n",
        f.real, f.imag);
} /* End of main */

Complex sum(Complex m, Complex n)
{
  Complex p;
  p.real = m.real + n.real;
  p.imag = m.imag + n.imag;
  return(p);
} /* End of sum */
```

```
Complex product(Complex m, Complex n)
{
  Complex p;
  p.real = m.real * n.real - m.imag * n.imag;
  p.imag = m.real * n.imag + m.imag * n.real;
  return(p);
} /* End of product */

Complex quotient(Complex m, Complex n)
{
  Complex p;
  float denom;
  denom = n.real*n.real + n.imag*n.imag;
  p.real = (m.real*n.real + m.imag*n.imag) / denom;
  p.imag = (m.imag*n.real - m.real*n.imag) / denom;
  return(p);
} /* End of quotient */
```

Program 16.2 Complex numbers and operations.

functions are implemented to perform the operations of addition, multiplication, and division of complex numbers. The definition of these operations are:

Given a complex number a + ib ; *a* is called the real part, and *b* the imaginary part; *i* represents square root of – 1.

Given two complex numbers a + ib and c + id the definition of sum, product and quotient are:

sum: $(a + ib) + (c + id) = (a + c) + i(b + d)$
product: $(a + ib) * (c + id) = (ac - bd) + i\ (ad + bc)$
quotient: $(a + ib) / (c + id) = \{ac + bd/(c^2 + d^2)\} + i\ \{bc - ad/(c^2 + d^2)\}$

Example Program 16.2 implements the sum, product and quotient functions for complex numbers. Observe that each function has structure variable names as formal arguments. The function *type* is defined as a structure and thus the result returned to the function name is the specified structure. Observe that in the body of the function we have declared a variable *p* as a structure and assign calculated values to p.real and p.imag. Having found the two component values, the structure *p* is returned to the function name. We could have attempted to write

sum(Complex m, Complex n, Complex p)

with the intention of getting the result back in the structure *p*. This is not allowed. Result cannot be returned to a structure name used as a formal argument.

As the last example in this section, we will reconsider the example given in Chapter 11 (Example 11.7). The problem is: given today's date and the date an employee joined a job to find if he/she has completed one year service. We will now use a structure and a function to solve the problem.

The structure used is Date which is defined in Example Program 16.3. The main advantage of using a structure in this case is a uniform data type definition of the two dates of relevance in this problem. The rest of Example Program 16.3 is self-explanatory.

```
/* Example Program 16.3 */
/* Illustrates use of structure in function */
#include <stdio.h>
#define TRUE   1
#define FALSE  0
typedef struct date
{
  int day;
  int month;
  int year;
} Date;
typedef int Boolean;
Boolean one_year_service(Date today,
                         Date joining_day);

main()
{
  Date today, join;
  int emp_no;
  scanf("%d %d %d", &today.day, & today.month,
       &today.year);
  while (scanf("%d %d %d %d", &emp_no, &join.day,
               &join.month, &join.year) != EOF)
    {
      if (one_year_service(today, join))
       printf("Emp No = %d Completed one year service\n",
              emp_no);
      else
       printf("Emp No = %d Not completed one year service\n",
              emp_no);
    } /* End while */
} /* End of main */

Boolean one_year_service(Date today, Date joining_date)
{
  int diff_day, diff_month, diff_year;
  diff_day = today.day - joining_date.day;
  diff_month = today.month - joining_date.month;
  diff_year = today.year - joining_date.year;
  if ((diff_year > 1) || (diff_year == 1) &&
      (diff_month > 0) ||
      ((diff_year == 1) && (diff_month == 0) &&
       (diff_day >= 0)))
    return(TRUE);
  else
    return(FALSE);
} /* End one_year_service */
```

Program 16.3 Use of structure in function.

16.2 USE OF STRUCTURES IN ARRAYS AND ARRAYS IN STRUCTURES

So far we have considered arrays in which the individual elements were scalars. Elements of an array can also be structures. Consider the structure item_in_store described at the beginning of the chapter. The structure is

```
typedef struct item_in_store
    {   int item_code;
        int qty_in_stock;
        float price;
    } Item;

Item inventory [100];
```

The above declaration states that the array inventory has 100 components each of which is a structure. The storage will be as shown in Table 16.2.

Table 16.2 Illustrates an Array whose Elements are Structures

	int item_code	int qty_in_stock	float price
inventory[0]			
inventory[1]			
.			
.			
.			
inventory[99]			

An individual element is referred to as:

```
inventory[2].price
```

which gives the price of inventory[2].

We illustrate the use of this array in Example Program 16.4 which lists items out_of_stock in the store and computes the total value of inventory kept in the store.

Observe in Example Program 16.4 the definition of struct as global (i.e. outside main). The maximum size of inventory is declared as 100. The actual number of items in the inventory is read in as data. The *for* loop reads one record corresponding to one inventory item and processes it. If an item is out of stock (that is, if inventory[i].qty_in_stock == 0) its code is printed out. It is not included in calculating the inventory value. The inventory value is accumulated in the *for* loop. Finally the inventory_value is rounded to the nearest rupee and printed out.

It is possible to have arrays as components of structures. For example, if we want the name of items besides their item_code we may define a structure.

```
typedef struct item_in_store
    {   int item_code;
        char item_name[20];
        int qty_in_stock;
        float price;
    } Item;
```

```
/* Example Program 16.4 */
/* Illustrates use of structures as array elements */
#include <stdio.h>
typedef struct item_in_store
{
  int item_code;
  int qty_in_stock;
  float price;
} Item;

main()
{
  Item inventory[100];
  int i, no_of_items, value_of_inventory;
  float inv_value = 0;
  scanf("%d", &no_of_items);
  /* No_of_items in store. It must be less than
     100 the maximum size of inventory */
  for (i = 0; i < no_of_items; ++i)
    {
      scanf("%d %d %f", &inventory[i].item_code,
           &inventory[i].qty_in_stock,
           &inventory[i].price);
      if (inventory[i].qty_in_stock == 0)
       printf("Item number = %d out of stock\n",
            inventory[i].item_code);
      else
       inv_value += inventory[i].qty_in_stock *
         inventory[i].price;
    } /* End of reading all inventory records */
  value_of_inventory = inv_value + 0.5;  /* rounding */
  printf("Inventory value to nearest rupee = %d\n",
       value_of_inventory);
} /* End of main */
```

Program 16.4 Use of structure as array element.

If in Example Program 16.4 we want to print the name of inventory items which are out of stock we may modify the *if* statement in the program. This is given in Fig. 16.1.

```
if ( inventory[i].qty_in_stock == 0 )
{
  printf ( "The following item is out of stock\n" );
  printf ("Item code is %d\n", inventory[i].item_code );
  puts ( inventory[i].item_name );
  printf ( "\n" );
}
else
  inv_value += inventory[i].qty_in_stock
  * inventory[i].price;
```

Fig. 16.1 Use of an array in a structure.

(We have assumed that the name of the inventory item has already been read in by a scanf statement.)

It is also possible to define structures within structures. We illustrate this with an example. Suppose it is required to describe suppliers who supply items to a store. This may be described by the structure:

```
typedef struct supp_record
    {   int supp_code;
        char supp_name[20];
        struct supp_address
            {   char street[30];
                char city [20];
                int pin_code;
            }
    } Supplier_rec; /*End of structure supplier */
Supplier_rec supplier[100];
```

Observe that the address of the supplier is declared as a structure within the structure supp_record. We will now write an instruction to illustrate how such a structure can be used. This instruction is to find out the name of all suppliers whose address has the Pin Code 560001 and is given in Fig. 16.2.

```
for ( i = 0; i < no_of_suppliers; ++i )
  {
    if ( supplier[i].supp_address.pin_code == 560001 )
      {
        puts ( supplier[i].supp_name );
        printf ( "\n" );
      }
  }
```

Fig. 16.2 Use of structure within a structure.

We have specified two structures in this section. One of them describes item in an inventory and the other suppliers. We will now see how they can be used together in a program. Suppose it is required to find out the address of all suppliers who supply a particular item to a store. In the current structure describing an item_in_store there is no reference to who supplies the item. Similarly there is no information in supp_record about what items the supplier is capable of supplying. Thus it is not possible to find out who could supply an item which is out of stock in a store. We will alter the item_in_store structure to include the supp_code of all suppliers who supply this item. The altered item_in_store structure is shown below:

```
typedef struct item_in_store
    {   int item_code;
        char item_name[20];
        int qty_in_stock;
        float price;
        int supp_code[3];
    } Item;
```

In the above structure we have allowed for three suppliers for each item_in_store. An array with three components stores these three suppliers' codes.

We now give a program segment in Fig. 16.3 to print the name and address of all suppliers who can supply individual items which are out of stock in the store. Observe in this

```
Item inventory[100];
for ( i = 0; i < no_of_items; ++i )
  {
    if ( inventory[i].qty_in_stock == 0 )
      {
        for ( j = 0; j < 3; ++j )
          search_key[j] == inventory[i].supp_code[j];
        for ( k = 0; k < no_of_suppliers; ++k )
          for ( m = 0; m < 3; ++m )
            if ( supplier[k].supp_code == search_key[m] )
              print_supplier_address ( supplier, k );
      } /* End of finding suppliers for inventory[i] */
  } /* End of search of all items in inventory */
```

Fig. 16.3 Printing name and address of all suppliers who can supply a specified out-of-stock item.

program that if inventory item 4, for example, is out of stock, then search_key [1], [2] and [3] will store the codes of the three suppliers for item 4. Using these three codes we now search all the supplier records to find the three suppliers who have these codes and immediately print their names and addresses.

The function which prints the address of a supplier is given in Fig. 16.4. Observe that to access an array of a structure within a structure we write

supplier[k].supp_address.street

```
void print_supplier_address ( Supplier_rec supplier[],
                              int k )
{
  puts ( supplier[k].supp_name );
  printf( "\n" );
  puts ( supplier[k].supp_address.street );
  printf( "\n" );
  puts ( supplier[k].supp_address.city );
  printf ( "\n %d \n", supplier[k].supp_address.
                                    pin_code ) ;
} /* end of function print_supplier_address */
```

Fig. 16.4 Printing the name and address of a supplier from a given supplier structure.

This will find the street name in the supplier_address of the k^{th} supplier. Observe that the identifier supp_code is used in two structures item_in_store and supp_record. This is allowed as these names by themselves have no meaning unless qualified by the structure identifier.

Structures are used in files and have many applications. We will return to this topic after we discuss what are known as *pointers* in the next chapter.

EXERCISES

16.1 Create a structure to specify data on students given below:

Roll no., Name, Department, Course, Year of joining

A typical student's data will be

1456 S. Raghavan C.S. B.E. 1991

Assume that there are not more than 500 students in the college.

(i) Write a function to print names of all students who joined in a particular year.

(ii) Write a function to print the data on a student whose roll number is given.

16.2 Create a structure to specify data on customers in a bank. The data to be stored is:

Acct. no, Name, Balance in account.

Assume maximum of 200 customers in the bank.

(i) Write a function to print the Acct. no and name of each customer with balance below Rs. 100.

If a customer gives a request for withdrawal or deposit it is given in the form:

Acct. no, amount, (1 for deposit, 2 for withdrawal)

(ii) Write a program to give a message, "the balance is insufficient for the specified withdrawal".

16.3 Create a structure for items in a store with the following data:

Part no., Part name, Qty. on hand, Re-order level, Re-order quantity, Supplier code.

Assume there are 100 items in the store:

(i) Write a program to print details of items whose stock is below the re-order level.

(ii) Write a function to automatically re-order items whose stock is below the re-order level.

16.4 Create a structure to store employee's data with the following information:

Employee's no., Employee's name, Employee's pay, date of joining (which is itself a structure)

(i) It is decided to increase the pay as per the following rules:

Pay <= Rs. 2000	: 15 % increase
Pay <= Rs. 5000 but > Rs. 2000	: 10 % increase
Pay > Rs. 5000	: no increase

Write a program to do this. (Assume there are 200 employees.)

(ii) Write a program to print details of employees who have completed 20 years service.

16.5 A scooter company has serial numbers of scooters starting from AA0 to FF9. The other characteristics of scooters to be specified in a structure are:

Year of manufacture, Colour, Horse power.

(i) Specify a structure to store information corresponding to a scooter.

(ii) Write a program to retrieve information on scooters with serial numbers between BB0 and CC9.

17. Pointer Data Type and its Applications

Learning Objectives

In this chapter we will learn:

1. The distinction between the contents of a variable and the address of a variable
2. How to declare variable names to store addresses also known as pointers
3. The use of the operator & to find the address of a variable name
4. The use of operator * to retrieve contents of a variable, given its address
5. Calling functions by passing addresses of formal arguments to it
6. The use of an array name as a pointer

Pointer data type is an important data type in C as it is used extensively by professional C programmers. Beginners find it difficult to understand what is a pointer data type and how it is used. We will explain these ideas first.

17.1 POINTER DATA TYPE

When we declare a variable name x as type integer we tell the compiler that a location in memory where an integer can be stored should be found and it should be given a name x.

Thus writing

```
int x ;
```

will pick a memory box (i.e., a location in memory) and give it a name x. The variable name is a symbolic name for the address of the memory box. This is shown in Fig. 17.1. The numeric value of the address corresponding to x is shown as 2568 in Fig. 17.1. If we write $x = 32$ then the integer 32 is stored as contents of x as shown in Fig. 17.2.

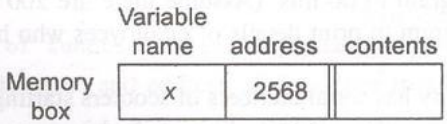


Fig. 17.1 Effect of declaring a variable name.

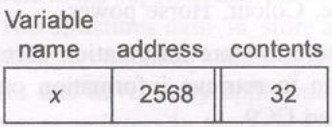


Fig. 17.2 A variable name with a value assigned to it.

We have already encountered the operator & which was used in scanf function. The operator & is called an address operator. If we write:

$$\&x$$

the operator & tells the compiler to find the numeric value of the address of a memory box whose symbolic name is x. We see from Fig. 17.2 that the address corresponding to x is 2568. Thus $\&x$ is 2568. If we write:

$$y = \&x ;$$

then 2568 which is an *address* is stored in y. When a variable stores an address we declare that variable as a *pointer data type*.

Thus y is declared as:

```
int *y ;
```

This declaration says that y will store the address of an integer variable name.

In this case y is the address of the integer variable name x. Fig. 17.3 shows what happens when we write

$$y = \&x ;$$

Variable name	address	contents
y	8468	2568

Fig. 17.3 Effect of writing $y = \&x$.

The address allocated by the compiler to store y is shown in Fig. 17.3 as 8468 (Observe that it has no relation to 2568 which is the address of x).

If we write:

$$z = *y ;$$

* is taken as an operator which is called an *indirection operator*.

The indirection operator * asks the compiler to:

1. Read the contents of y. (As y is declared as of *type* pointer its contents will be an address.)
2. Go to the address found in Step 1 and retrieve its contents
3. Store the value retrieved in Step 2 in z.

From Fig. 17.3 we see that 2568 is stored in y. It is interpreted as an address. The contents of this address is 32 as shown in Fig. 17.2. Thus 32 will be stored in z. This is shown in Fig. 17.4.

Variable name	address	contents
z	6426	32

Fig. 17.4 Effect of assignment $z = *y$.

Pictorially we have shown the sequence of operations described in Fig. 17.5.

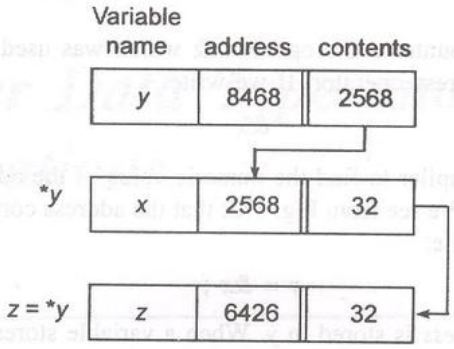


Fig. 17.5 Sequence of operations when we write $z = *y$.

Observe that in a statement * is taken as an indirection operator. The indirection operator can operate only on addresses. Thus the operand of an indirection operation should be declared to be a pointer data type. The symbol * used in the *declaration* has a different meaning from the * used in a *statement*. In the declaration it only gives information that *y* will store an address whereas in a statement it is an operator commanding that the address stored in *y* should be retrieved and used.

We illustrate in Example Program 17.1 the use of the indirection operator * and pointer type data declaration.

```
/* Example Program 17.1 */
/* Illustrating pointer data type */
#include <stdio.h>

main()
{
  char p, z, w;
  char *address_of_p;
  p = 'A';
  address_of_p = &p;
  z = *address_of_p;
  /* Look up address of p find contents and
     assign to z */
  w = *(&p);
  putchar(p);
  putchar(z);
  putchar(w);
} /* End of main */
```

```
Output:
AAA
```

Program 17.1 Illustrating pointer data type.

The sequence of operations $*(\&x)$ is meaningful. However, the sequence $\&(*x)$ is meaningless. $*x$ commands that the address of x should be found and contents of that address which will be a value is to be retrieved. & is a command to find the address of a value. This is meaningless. Address is meaningful only for a variable name.

17.2 POINTERS AND ARRAYS

An array in C is declared as:

```
int a[5] = {8, 6, 4, 9, 11} ;
```

which states that *a* is an array of integers and that the array has 5 components a[0], a[1], a[2], a[3] and a[4]. The array is stored in consecutive addresses in memory as shown in Table 17.1.

Table 17.1 Illustrating Storage of an Array

name	address	contents
a[0]	2688	8
a[1]	2689	6
a[2]	2690	4
a[3]	2691	9
a[4]	2692	11

If we write:

```
int *x ;
x = &a[0] ;
```

it will result in

```
x = 2688 ;
```

If we write instead

```
x = a ;
```

then also *x* will be 2688. The reason is that in the C language the *name* of an array variable is taken as the *address* of its zeroth component. Thus *a* is the pointer to the array a[5]. Using this fact we can write Example Program 17.2 to store characters in an array and display them.

```
/* Example Program 17.2 */
/* Storing characters in an array & displaying
   them */
#include <stdio.h>

main()
{
  char *ptr;
  char b[5];
  for (ptr = b; ptr < b+5; ++ptr)
    {
      *ptr = getchar();
      putchar(*ptr);
    }
  putchar('\n');
}
```

Program 17.2 Illustrating that array name is pointer.

We could have done this also by using subscripts. What we have done above is only as an illustration.

Example Program 17.3 reverses a character string and prints it.

```
/* Example Program 17.3 */
/* Use of array name as pointer */
#include <stdio.h>
main()
{
  char magic[] = "ABRACADABRA";
  char *addr;
  addr = magic + 11;
  while (--addr >= magic)
    putchar(*addr);
  putchar('\n');
}
```

Program 17.3 Reversing string.

In Example Program 17.3 the variable name magic stores the starting address of the string magic. One is added to the address of the *last* character in the array magic and stored in addr. The *while* loop prints the characters in reverse. Observe the use of – –addr for comparison in the *while* loop.

17.3 POINTERS AND FUNCTIONS

Pointers can be passed as arguments to a function and can also be returned to the name of the function. We will show a small application of this with an example. The problem is to write a function to interchange values of two integers *a* and *b*. In other words if a = 25 and b = 38 the function should, after doing its work, give a = 38 and b = 25.

We now write a function:

```
void interchange (int x, int y)
```

with *x* and *y* as formal arguments. The function is shown below:

```
void interchange (int x, int y)
{   int temp;
    temp = x;
    x = y;
    y = temp;
} /* End of interchange (x, y) */
```

If we now write in the main program

```
a = 25; b = 38;
interchange (a, b);
```

then 25 is copied into *x* and 38 to *y* which are the formal arguments of the function interchange. The function interchanges the values of *x* and *y*. However, *a* and *b will not be* interchanged as the values interchanged by the function are not sent back to the locations where *a* and *b* are stored. The statement interchange (a, b) does not get them back. This is illustrated in

Fig. 17.6. Observe that the address of x and y are not known outside the function in which x and y are used.

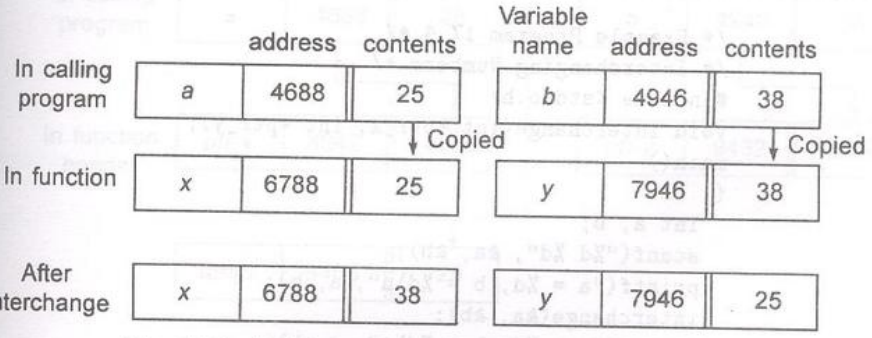


Fig. 17.6 Relation between calling and called functions.

We can circumvent this problem by using an array as the formal argument in the *function* interchange. We have seen in Chapter 13 that values computed in functions can be returned to its formal arguments only if they are arrays. We rewrite the *function* interchange as shown below:

```
void interchange (int x [ ]);
{   int temp;
    temp = x[0];
    x[0] = x[1];
    x[1] = temp;
} /* End of interchange */
```

We can call the function with another array *a*

```
interchange (a) ;
```

where *a* is declared as: int a[2]

Remember that the argument *a* in interchange(a) is in fact a pointer (i.e. an address) and not a value. Thus when interchange is called with *a*, its *address* is taken as the address of *x*. In other words array *a* and array *x* occupy the *same locations in memory*. They have different symbolic names which are in fact synonyms. Thus when x[0] and x[1] are interchanged by the *function* interchange a[0] and a[1] also get automatically interchanged. This is illustrated in Fig. 17.7.

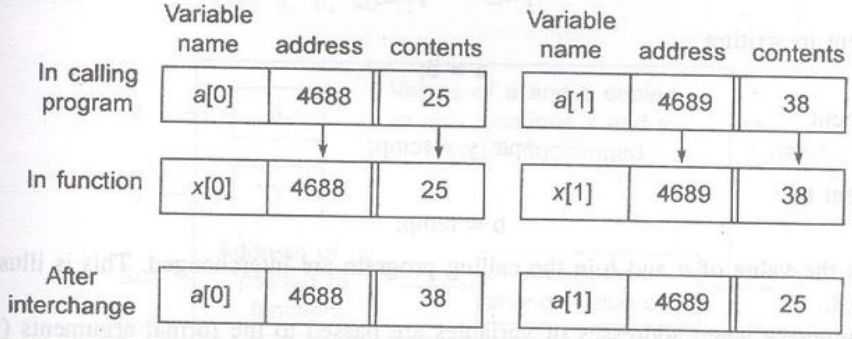


Fig. 17.7 Illustrating the fact that addresses of the actual argument are passed to the formal argument when the argument is an array.

We can use this idea of sending addresses rather than values to the formal arguments of the *function*. The *function* interchange is redefined as shown in Example Program 17.4. In

```
/* Example Program 17.4 */
/* Interchanging Numbers */
#include <stdio.h>
void interchange(int *ptr_x, int *ptr_y);
main()
{
  int a, b;
  scanf("%d %d", &a, &b);
  printf("a = %d, b = %d\n", a, b);
  interchange(&a, &b);
  printf("a = %d, b = %d\n", a, b);
}

void interchange(int *ptr_x, int *ptr_y)
{
  int temp;
  temp = *ptr_x;
  *ptr_x = *ptr_y;
  *ptr_y = temp;
} /* End of interchange */
```

Program 17.4 Interchanging numbers.

this program the function interchange has as its formal arguments two pointer variables, ptr_x, ptr_y respectively. In the body of the function in the statement

```
temp = *ptr_x ;
```

the operator * commands that the address contained in ptr_x (namely address of *a*) should be referred and the contents of this address read and stored in temp. Thus this statement is equivalent to writing

```
temp = *(&a) = a;
```

The next statement

```
*ptr_x = *ptr_y;
```

is equivalent to writing

```
a = b;
```

The statement

```
*ptr_y = temp;
```

is equivalent to

```
b = temp;
```

Thus the value of *a* and *b* in the calling program are interchanged. This is illustrated in Fig. 17.8.

In summary when addresses of variables are passed to the formal arguments (which should be of type pointer) of a function, the formal and actual arguments occupy the same locations in memory. Thus the values of the formal variables of function are changed by

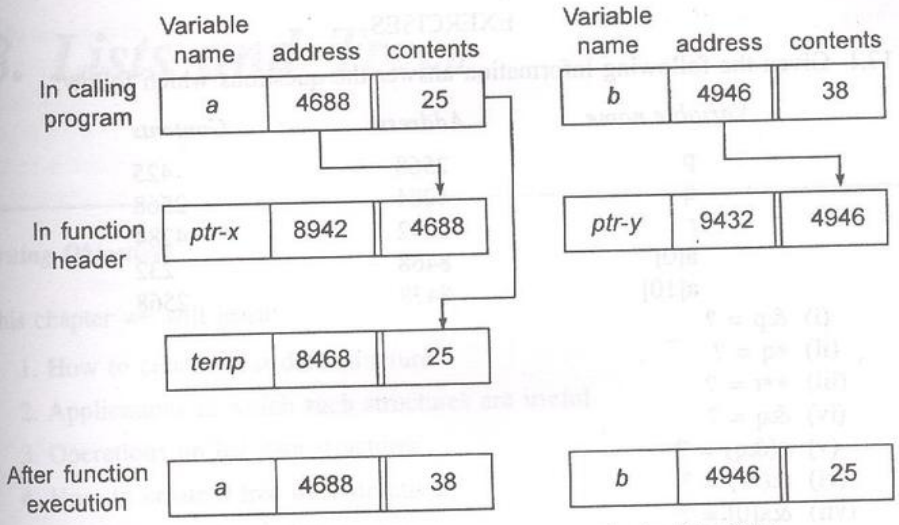


Fig. 17.8 Passing pointers as arguments to functions.

operations within the function. Figure 17.9 illustrates what happens when the formal parameters of a function are a mixture of normal variables and pointer variables.

```
Function definition

int f_name( int x, float y, char *ptr )
{
.................
.................
return();
.................
.................
return();
} /* End of function */

Calling function

int a; float b; char c;
f_name( a, b, &c );
```

Statements in the body of the function

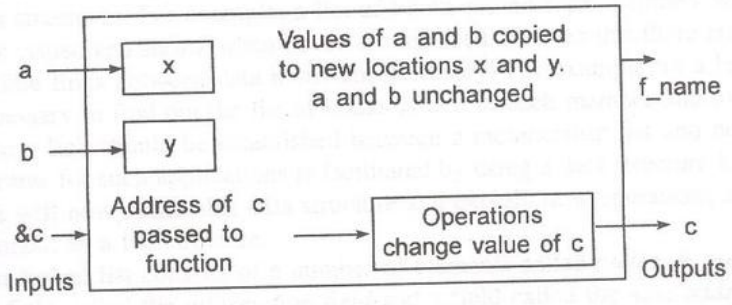


Fig. 17.9 Effect of using pointer variable as an argument of function.

EXERCISES

17.1 Given the following information answer the questions which follow:

Variable name	*Address*	*Contents*
p	2568	425
q	4284	2568
r	6242	4284
a[0]	8468	232
a[10]	8478	2568

(i) &p = ?
(ii) *q = ?
(iii) **r = ?
(iv) &q = ?
(v) *(&q) = ?
(vi) &(*r) = ?
(vii) &a[0] = ?
(viii) a = ?
(ix) a[0] = ?
(x) &a[5] = ?
(xi) a[10] = ?
(xii) *a[10] = ?

17.2 Given three variables *x*, *y*, *z* write a function to circularly shift right their values. In other words if x = 5, y = 8, z = 10 after circular shift y = 5, z = 8 and x = 10. Call the function with variables *a*, *b*, *c* to circularly shift their values.

17.3 Given an array *p* of size 5. Write a function to shift it circularly left by two positions. Thus p[0] = 5, p[1] = 3, p[2] = 8, p[3] = 9, p[4] = 6 then after the shift p[0] = 8, p[1] = 9, p[2] = 6, p[3] = 5 and p[4] = 3. Call this function with a (3 × 5) matrix and get its rows left shifted.

17.4 Write a function to solve a quadratic equation $ax^2 + bx + c = 0$. The input to the function are the values *a*, *b*, *c* and the outputs of the function should be stored in variable names *p*, *q* appropriately declared.

17.5 Write a function which will take as its input a matrix mat of size (m × n) (m < 10, n < 5) and return the same matrix with all its elements replaced by absolute values.

18. Lists and Trees

Learning Objectives

In this chapter we will learn:

1. How to create a list data structure
2. Applications in which such structures are useful
3. Operations on list data structures
4. How to create a tree data structure
5. Applications of tree data structures

18.1 LIST DATA STRUCTURE

The two data structures we have encountered so far for representing collections of data items are arrays and structures. Retrieval of information from any array is simple. It is, however, necessary to specify the size of the array before it is used. Thus situations could arise where the specified storage is too much,or too little. For example we used arrays to represent strings of characters in Chapter 14. When characters were to be inserted in the string then we had to move to the right all the characters appearing after the insertion point. The array size should be sufficient to accommodate the expanded string. If a substring is to be deleted from a string then again characters on the right of deleted substring have to be moved left. The string would now become smaller and part of the array would thus be unused and wasted. The procedures to delete and insert would also involve fair amount of book keeping of array index to move characters. Thus in applications which require frequent editing of strings of characters it would be useful to have a data structure in which we can dynamically allocate space as needed and also release unused space. Further, operations of insertion and deletion should be simplified.

There are also other applications of computers in which it is necessary to have dynamically changing data structures. For example, a list of books issued from a library would grow as new books are issued and shrink when books are returned. Besides this there are applications in which flexible links between data items are necessary. For example, in a lending library it may be necessary to find out the list of books issued to each member and overdue books. In such a case a link should be established between a membership list and book issue list. Writing programs for such applications is facilitated by using a data structure known as a *list structure*. We will now define this data structure and explain how operations are carried out on data organized as a list structure.

A linear linked list consists of a number of elements called *nodes*. A node consists of two fields, a field called the *information field* and a field called the *next address field*. The information field holds the actual information in a list element. The next address field holds

the address of the next node in the list. In other words it establishes a link to the next element in the list. The next address which is used to access the next node is known as a *pointer*. The entire linear linked list is referred to and accessed by a *pointer* which points to (i.e., it contains the address of) the first node of the list. This pointer is external to the list and is a *list identifier* which contains the address of the first element of the list. The last node of the list is specified by storing a special symbol known as NULL in its next address field. The structure of a list is shown in Fig. 18.1.

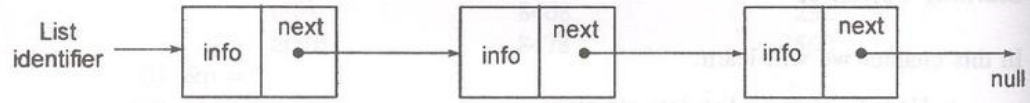


Fig. 18.1 A list structure.

C provides a method of creating a list with a structure and a *pointer data type*. For example, if a node of a list is to be defined in C it may be done by the following declaration:

```
typedef struct node
                {   char info;
                    struct node *next_node;
                }   List_node;

List_node   *list_of_char, n1;
```

The declaration defines a structure named node. This structure has two components. The first is of type char. It tells that info will store a character. The next component is a structure of type node. Observe that we have defined a structure (which refers to itself) within a structure. This is allowed in C. This declaration states that a pointer (namely, an address) named next_node will point to another structure of the type node. Following this we have declared list_of_char as a pointer which will point to (i.e. contain the address of) a data of type List_node. A node named n1 is also declared as of type List_node. If we write the following statement:

```
*list_of_char = &n1;
n1.info = 'X';
```

the effect will be to create a node n1 pointed to by list_of_char. This is shown in Fig. 18.2.

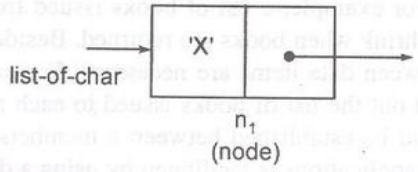


Fig. 18.2 Creating a node in a list-of-char.

We will now create a list of three nodes shown in Fig. 18.3. We have named the nodes n1, n2 and n3.

A program for doing this is given as Example Program 18.1. Examining this program

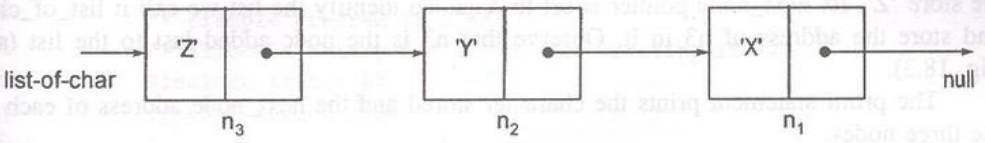


Fig. 18.3 A list with three nodes to be created.

```
/* Example Program 18.1 */
/* Creation of a list */
#include <stdio.h>
typedef struct node
{
  char info;
  struct node *next_node;
} List_node;

main()
{
  List_node *list_of_char, n1, n2, n3;
  /* Store 'X' in n1 */
  n1.info = 'X';
  n1.next_node = NULL;
  /* n1 is the only node in the list now */
  n2.info = 'Y';
  n2.next_node = &n1;
  /* Node n2 is created and linked to n1 */
  n3.info = 'Z';
  n3.next_node = &n2;
  list_of_char = &n3;
  /* End of creation of list */
  printf("node3 = %c %x, node2 = %c %x\n",
        n3.info, n3.next_node, n2.info,
        n2.next_node);
  printf("node1 = %c %x, header addr = %x\n",
        n1.info, n1.next_node, list_of_char);
} /* End of main */

Output of Example Program 18.1 :

node3 = Z 7fffe080, node2 = Y 7fffe088
node1 = X 0, header addr = 7fffe078
```

Program 18.1 Creation of a list with three nodes.

we see that we declare list_of_char as a pointer to List_node and n1, n2, n3 as of type List_node. We store in n1.info 'X' and store NULL in its address field. NULL is a symbol defined in <stdio.h> and is used to indicate the end of a list. Thus we now have a list with one node. We now take node n2 and store in its information field 'Y'. In its next address field we store the address of n1 (&n1), thereby linking n2 to n1. Finally in n3

we store 'Z'. Its next_node pointer is set to &n2. To identify the list we call it list_of_char and store the address of n3 in it. Observe that n3 is the node added last to the list (see Fig. 18.3).

The printf statement prints the character stored and the next_node address of each of the three nodes.

The problem with Example Program 18.1 is the need to declare all the nodes n1, n2, n3 at the beginning. As the main purpose of lists is to store variable number of nodes we must find a better method of creating a list without having to name and declare them before creating the list. We will do this next.

In order to do this we need a C function provided in the C library. This function is called *malloc* (which is an abbreviation for memory allocate). We will also use the unary operator *sizeof*. We will define these next.

The function *malloc* allocates at run time number of bytes needed to store a node. It has one argument which specifies the number of bytes to be allocated. If the function successfully allocates the requested storage it will return the address of the requested storage. If it is unable to provide the requested storage (e.g., when there is no space left in memory), it will return NULL. As *malloc* returns an address, this address must be cast to the type of data structure it must point to. For example, if we want to allocate one int cell and set a pointer to this address we do the following:

```
int *ptr;
ptr = (int *) malloc (sizeof(int));
```

The argument of malloc is sizeof (int). The operator *sizeof* returns the number of bytes needed to store a data of the *type* specified. The function malloc now returns the address where these bytes can be stored. (int *) casts this address to be of *type* pointer to an integer. ptr is declared to be an address for storing a data of type int by the declaration int *ptr. Thus we now set ptr to point to the address of the correct type of argument needed.

If we want to allocate a location in memory to store a data of *type* struct node, we do the following:

```
struct node *ptr;
ptr = (struct node *) malloc(sizeof(struct node));
```

The function *malloc* provides the address of locations in the memory to store ptr. Thus by using malloc we can create lists of indefinite length.

We will now rewrite Example Program 18.1 as Example Program 18.2 using *malloc* function. In Example Program 18.2 temp and list_of_char are declared as pointers to struct_node. In the main program an address is allocated to temp by the function *malloc*. A character is read by the getchar ( ) function. It is stored in the char field of the structure whose address is temp by the statement

```
(*temp).info = a_char;
```

As statements of the type (*ptr).info occur very often in C a shorter notation is provided. This notation is ptr –> info which is identical in its effect to (*ptr).info. Thus the statement (*temp).info may be written as: temp –> info. The symbol –> is a minus sign followed by the greater than symbol. The next node field of temp is made NULL. We have now created a list with one node. To add more nodes to the list we first store the address of

```
/* Example Program 18.2 */
/* Allocate memory for a list and creating it */
#include <stdio.h>
typedef struct node
{
  char info;
  struct node *next_node;
} List_node;
List_node *temp, *list_of_char, *print_node;

main()
{
  char a_char;
  temp = (List_node *)malloc(sizeof(List_node));
  a_char = getchar();
  (*temp).info = a_char;
  /* The symbol (*temp).info has another notation
     temp->info where -> is a minus sign followed
     by a > sign. This symbol is a short hand
     notation and is extensively used */
  temp->next_node = NULL;
  list_of_char = temp;
  while ((a_char = getchar()) != '\n')
    {
      temp = (List_node *)malloc(sizeof(List_node));
      temp->info = a_char;
      temp->next_node = list_of_char;
      list_of_char = temp;
    } /* End of while */
  print_node = list_of_char;
  while (print_node != NULL)
    {
      printf("Char = %c, Link Address = %x\n",
             print_node->info,
             print_node->next_node);
      print_node = print_node->next_node;
    } /* End of while */
} /* End of main */
```

```
Output of Example Program 18.2 :

Char = P, Link Address = 1834
Char = W, Link Address = 1824
Char = Z, Link Address = 1814
Char = Y, Link Address = 1804
Char = X, Link Address = 0
```

Program 18.2 Creating a list using malloc function.

temp in list_of_char; read another character from input and enter the *while* loop. In the *while* loop a new node address is obtained using malloc function and is stored in temp. The character read is stored in this node and the node is linked to the existing list by the statement

```
temp->next_node = list_of_char;
```

The new node address is stored in list_of_char, another character is read and the *while* loop is re-entered. This procedure is repeated till the end of input is reached. Outside the loop the list is printed. The steps in creating the linked list is shown in Fig. 18.4.

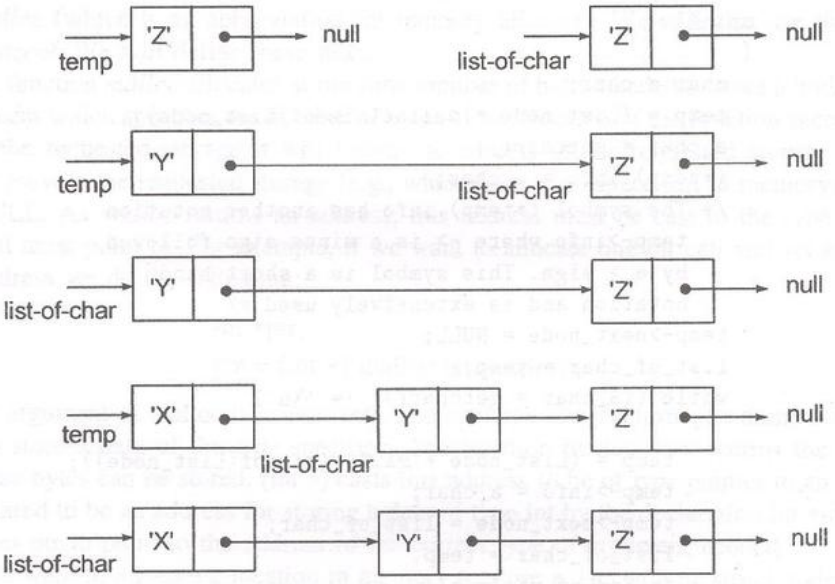


Fig. 18.4 Steps in creation of a linked list.

18.2 MANIPULATION OF A LINEARLY LINKED LIST

In this section we will illustrate with a running example various manipulations which may be performed on lists. The program is briefly described below:

Example 18.1

A lending library keeps a list of issued books. Each item in the issue list has the following information. Accession number of the book, issue code which is either I, B or L and the identification code of the member to whom issued. The issue code I is used if issued to an individual member, B if sent for binding and L if issued to another library. As the number of books issued will be continuously changing we will use a list structure for issued books.

In Example Program 18.3 we will create a list of issued books. The following operations have to be normally performed on this list:

1. If a member asks for a book it is necessary to search the list of issued books to find its status.

```
/*  Example Program 18.3 */
/*  Function to create a book list */
#include <stdio.h>
typedef struct book
{
  int acc_no;
  char issue_code;
  struct book *next_book;
} Book;
Book * add_book(int book_no, char code,
                Book *list_of_books);

main()
{
  int no_of_book;
  char code_of_book;
  Book *book_list = NULL;
  scanf("%d %c", &no_of_book, &code_of_book);
  while (no_of_book != 0)
    {
      book_list = add_book(no_of_book, code_of_book,
                           book_list);
      scanf("%d %c", &no_of_book, &code_of_book);
    }
} /* End of main */

Book * add_book(int book_no, char code,
                Book *list_of_books)
{
  Book *new_book;
  new_book = (Book *)malloc(sizeof(Book));
  new_book->acc_no = book_no;
  new_book->issue_code = code;
  new_book->next_book = list_of_books;
  list_of_books = new_book;
  return(list_of_books);
}
```

Program 18.3 Creating a list of books.

2. When a book is issued to a member this should be added to the list of issued books.

3. When a book is returned by a member it should be deleted from the list of issued books.

In Example Program 18.3 the function add_book appends an issued book to an existing list_of_books. An integer variable book_no, a character variable code and a pointer variable list_of_books are formal arguments of the function add_book. In the function, a node to add a new book to be issued is allocated by the function malloc. In this node called new_book the book number and issue code are stored. The next_book address is made list_of_books and

this address is returned to add_book function. As *list_of_books is initialized to NULL when add_book is called the first time next_node address will be NULL. The next time the function is invoked the address returned will be used thereby linking this new book node to the existing list.

Observe that add_book function is a pointer type. This is necessary as a pointer is returned to the function name. Example Program 18.4 is a function to search the list of issued

```
/* Example Program 18.4 */
/* Function to search a list for a book */
/* List of books created by Example Program 18.3
   and address list_of_books is assumed available */
#include <stdio.h>
#define TRUE     1
#define FALSE    0
typedef struct book
{
  int acc_no;
  char issue_code;
  struct book *next_book;
} Book;
typedef int Boolean;

Boolean found_book(int book_no, Book *list_of_books)
{
  Book *temp;
  temp = list_of_books;
  while (temp != NULL)
    {
      if (temp->acc_no == book_no)
        return(TRUE);
      else
        temp = temp->next_book;
    } /* End of while */
  return(FALSE);
} /* end of found_book */
```

Program 18.4 Searching a list of books.

books to locate a book with specified accession number. In this function the *while* loop follows the pointers in the list until the specified book is found or the end of the list is reached. The address of the header list_of_books is assumed available from Example Program 18.3.

Example Program 18.5 is a function to delete a book from the list. When a book is deleted from the list then the node corresponding to the book should be removed from the list. Assuming that the node marked current node in Fig. 18.5 is to be removed then the link address of the previous node should be replaced by the link address of the current node. This will make the previous node point to the next node "short circuiting" the current node. After this operation the link address field of the current node may be set to NULL. Further, the

```
/* Example Program 18.5 */
/* Function delete_book */
typedef struct book
{
  int acc_no;
  char issue_code;
  struct book next_book;
} Book;
Book *list_of_books;

Book *delete_book(int no_of_book, Book *list_of_books)
{
  Book *prev_book, *current_book;
  if (list_of_books->acc_no == no_of_book)
    {
      list_of_books = list_of_books->next_book;
      return(list_of_books);
    }
  prev_book = list_of_books;
  current_book = prev_book->next_book;
  while (current_book != NULL)
    {
      if (current_book->acc_no == no_of_book)
        {
          prev_book->next_book =
            current_book->next_book;
          free(current_book);
          return(list_of_books);
        }
      else
        {
          prev_book = current_book;
          current_book = current_book->next_book;
        }
    } /* End of while */
  printf("Book not in list\n");
  return(list_of_books);
} /* end of delete_book */
```

Program 18.5 Deleting a book from list-of-books.

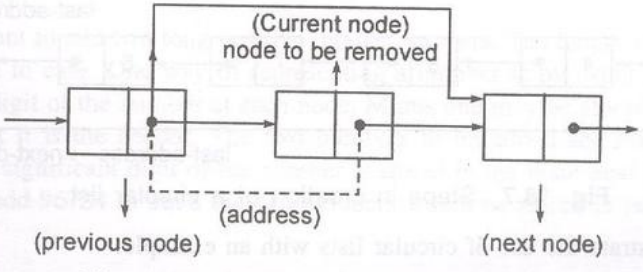


Fig. 18.5 Deletion of a node from a list.

library function *free* may be used to free the pointer to the node so that it maybe reused. The library function *free* takes as its formal argument a pointer and releases that pointer. An exception to this strategy occurs if the very first node (namely, the header node) is to be removed. In this case there is no "previous" node. Thus the header pointer is moved to point to the next node.

18.3 CIRCULAR AND DOUBLY LINKED LISTS

One of the shortcomings of a linear linked list is that having reached a node in the list we cannot go back to preceding nodes unless their addresses are stored. Suppose we make a small change in the structure of a list so that the next address field of the last node in the list is not NULL but is the address of the first node of the list, then we have what is called a *circular list* (see Fig. 18.6). In such a structure we can reach any node from any other node in the list. Along with this there is a danger of endlessly looping round the list. To prevent this the header node, namely, the first node in the list is made distinct by storing in its information field an element which distinguishes it from the other nodes in the list. Such a

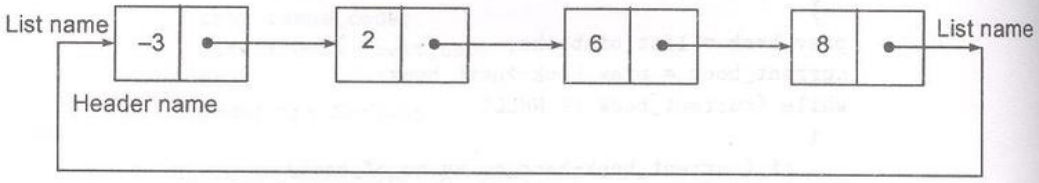


Fig. 18.6 A circular list of positive integers.

node is called a *sentinel node*. A sentinel node of a circular list of positive integers may store a negative integer. A circular list of integers is created by Example Program 18.6. Observe that the function create_circ_list returns the address of the header of the list to main. The method of creation of the list is illustrated in Fig. 18.7.

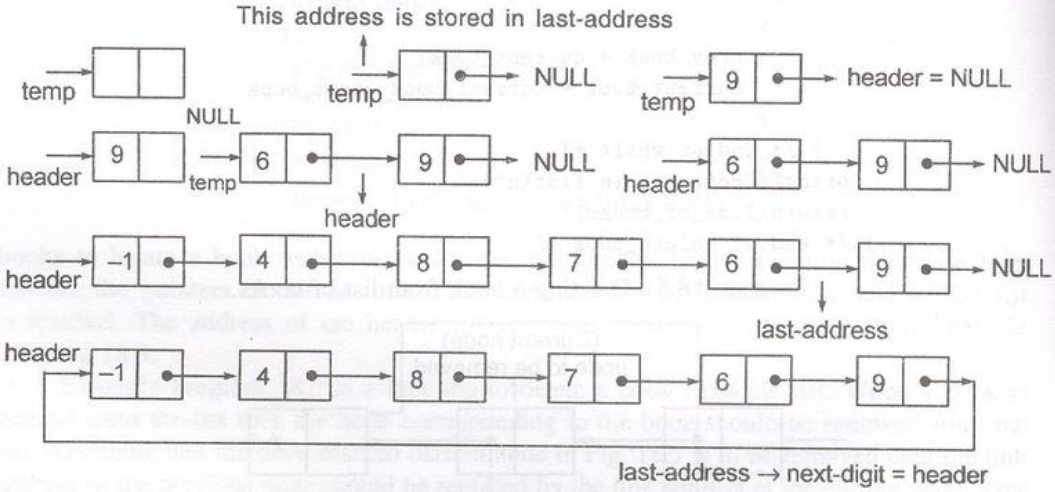


Fig. 18.7 Steps in creation of a circular list.

We will illustrate the use of circular lists with an example.

```
/* Example program 18.6 */
#include <stdio.h>
typedef struct dig_node
{
  int number;
  struct dig_node *next_digit;
} Digit_node;
Digit_node *create_circ_list(Digit_node *k);

main()
{
  Digit_node *list_head = NULL;
  list_head = create_circ_list(list_head);
} /* End of main */

Digit_node *create_circ_list(Digit_node *header)
{
  Digit_node *temp, *last_address;
  int digit;
  header = NULL;
  temp = (Digit_node *)malloc(sizeof(Digit_node));
  last_address = temp;
  while (scanf("%d", &digit)!= EOF)
    {
      temp->number = digit;
      temp->next_digit = header;
      header = temp;
      /* If true link header to last_address */
      if (digit < 0)
          last_address->next_digit = header;
      else
        temp =
          (Digit_node *)malloc(sizeof(Digit_node));
    } /* End of while */
  return(header);
} /* End of create_circ_list */
```

Program 18.6 Creating a circular list.

Example 18.2

Suppose we want to add two long positive integer numbers. The length of each number may vary from case to case. One way of representing a number is by using a list structure. We can store one digit of the number at each node. Minus one may be stored in the header node to indicate that it is the header. The two numbers to be added are stored in two circular lists. The least significant digit of the number is stored in the node next to the header. Thus if we want to add 96784 to 3678 the two numbers would be stored in two lists as shown in Fig. 18.8.

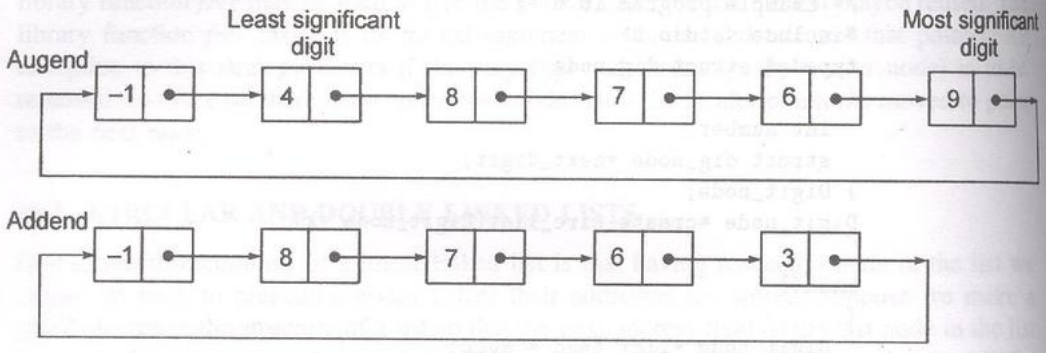


Fig. 18.8 Circular lists for augend and addend to be added.

Once the two lists are created the digits in the corresponding nodes (beginning from the least significant digit) of augend and addend are added and stored in a new list named result. The procedure to do this is given in Example Program 18.7. Comments are included to make the program self_explanatory. Observe that by using circular lists we are able to position the pointers of the augend and addend at the header of the respective lists.

```
/* Example Program 18.7 */
/* Program to add two integers */
#include <stdio.h>
typedef struct digit_node
{
  int number;
  struct digit_node *next_digit;
} Digit_node;

Digit_node *create_circ_list(Digit_node *header);
void print_circ_list(Digit_node *header);
Digit_node *Add_numbers(Digit_node *result_addr,
                        Digit_node *addend,
                        Digit_node *augend);

main()
{
  Digit_node *and_head, *aug_head, *result_head;
  and_head = create_circ_list(and_head);
  printf("Printing from main addend list\n");
  print_circ_list(and_head->next_digit);
  aug_head = create_circ_list(aug_head);
  printf("Printing from augend list\n");
  print_circ_list(aug_head->next_digit);
  result_head = Add_numbers(result_head, and_head,
                            aug_head);
  printf("Printing from main result list\n");
  print_circ_list(result_head->next_digit);
} /* End of main */
```

```
Digit_node *Add_numbers(Digit_node *result_addr,
                        Digit_node *addend,
                        Digit_node *augend)
{
  Digit_node *temp, *last_address, *header, *operand;
  int sum, carry;
  /* The following three statements initialize a
     circular list to store the result */
  temp = (Digit_node *)malloc(sizeof(Digit_node));
  last_address = temp;
/* Pointers of augend and addend lists are moved
   to point to the least significant digits */
addend = addend->next_digit;
augend = augend->next_digit;
carry = 0;
while ((addend->number >= 0) && (augend->number >= 0))
  {
    /* Addition of digits of addend and augend begins */
    sum = addend->number + augend->number + carry;
    temp->number = sum % 10;
    carry = sum / 10;
    /* After addition node is linked to the
       previous node */
    temp->next_digit = header;
    header = temp;
    /* A new node for result is created. Pointers
       of addend and augend are advanced */
    temp = (Digit_node *)malloc(sizeof(Digit_node));
    addend = addend->next_digit;
    augend = augend->next_digit;
  } /* End of while */
/* If end of either addend or augend list is reached then
   the operand is taken from the appropriate list */
if (addend->number < 0)
  operand = augend;
else
  operand = addend;
/* Carry if any is added to the operand and the
   result is stored in new list */
while (operand->number >= 0)
  {
    sum = operand->number + carry;
    operand = operand->next_digit;
    temp->number = sum % 10;
    carry = sum / 10;
    temp->next_digit = header;
    header = temp;
    temp = (Digit_node *)malloc(sizeof(Digit_node));
 } /* End of while */
```

```
    /* Carry if any is stored in the result list */
    if (carry == 1)
      {
        temp->number = carry;
        temp->next_digit = header;
        header = temp;
        temp = (Digit_node *)malloc(sizeof(Digit_node));
      }
    /* -1 is stored in the header node of the result */
    temp->number = -1;
    temp->next_digit = header;
    last_address->next_digit = temp;
    result_addr = temp;
    return(result_addr);
} /* End of function Add_numbers */

Digit_node *create_circ_list(Digit_node *header)
{
    Digit_node *temp, *last_address;
    int digit;
    header = NULL;
    temp = (Digit_node *)malloc(sizeof(Digit_node));
    /* Store address of node created at the beginning */
    last_address = temp;
    while (scanf("%d", &digit)!= EOF)
      {
        /* In the following statements read a digit
           and store it in the node already created.
           Set the next address held of the
           node = previously created node address
           (namely, header) */
        temp->number = digit;
        temp->next_digit = header;
        header = temp;
        printf("%d %d\n", digit, header);

        /* If digit < 0 link header to last_address */
        if (digit < 0)
          {
            last_address->next_digit = header;
            printf("Last link addr = %d\n", header);
          }
        else
          temp = (Digit_node *)malloc(sizeof(Digit_node));
        if (digit < 0)
          break;
      } /* End of while */
    return(header);
} /* End of create_circ_list */
```

```
void print_circ_list(Digit_node *header)
{
  Digit_node *temp;
  int digit;
  temp = header;
  digit = temp->number;
  while (digit >= 0)
    {
      printf("Addr = %d, digit = %d\n", temp, digit);
      temp = temp->next_digit;
      digit = temp->number;
    }
} /* End of print_circ_list */
```

Program 18.7 Adding two numbers represented as circular lists.

18.4 A DOUBLY LINKED CIRCULAR LIST

A doubly linked circular list consists of a number of nodes where each node is not only linked to the node to its right but also to the node to its left. In other words a node has a pointer giving the address of the next sequential node and another pointer giving the address of the previous node.

A doubly linked circular list is illustrated in Fig. 18.9. In this example, the information in each node is assumed to be a positive digit. We will identify one node as a header node

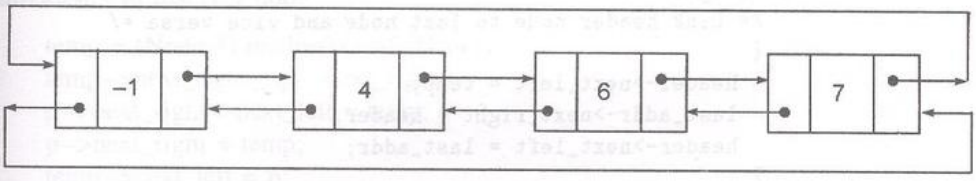


Fig. 18.9 A doubly linked circular list.

and store a – 1 in it to distinguish it as the header. A list of this type may be described by the declaration given in Example Program 18.8.

The primary advantage of a doubly linked list is that we can traverse it in both directions. Another advantage is the ease with which an arbitrary node may be deleted. Deletion is easy as links to both the previous node and the next node exist at all nodes.

Example Program 18.8 illustrates the creation of a doubly linked circular list. The sequence of creation of the list is illustrated in Fig. 18.10 with input data 2 4 6 –1. This sequence is obtained by tracing the procedure given in Example Program 18.8 step by step.

Observe that in a doubly linked list the following relation holds for any node pointer

p = p–>next_left–>next_right = p–>next_right–>next_left;

```
/* Example program 18.8 */
/* Function to create a doubly linked list */
#include <stdio.h>
typedef struct node
{
  int info;
  struct node *next_right;
  struct node *next_left;
} Node;
Node *header;

Node * create_doub_link_list(Node *header)
{
  Node *temp, *last_addr;
  int digit;
  header = NULL;
  temp = (Node *)malloc(sizeof(struct node));
  /* Store address of first node created */
  last_addr = temp;
  while (scanf("%d", &digit) != EOF)
    {
      temp->info = digit;
      temp->next_right = header;
      header = temp;
      if (digit < 0)
        /* Link header node to last node and vice versa */
        {
          header->next_left = temp;
          last_addr->next_right = header;
          header->next_left = last_addr;
        }
      else
        {
          temp = (Node *)malloc(sizeof(struct node));
          header->next_left = temp;
        }
    } /* End of while */
} /* End of create_doub_link_list */
```

Program 18.8 Function to create a circular doubly linked list.

Deletion of a node is quite easy in a doubly linked list as both the left and right pointers of a node are available in the node itself. For example, if a node containing a specified information is to be removed from the list of Fig. 18.10 it may be done by the procedure given as Example Program 18.9.

If a node is to be inserted to the right of a specified node p it may be done by the following statements:

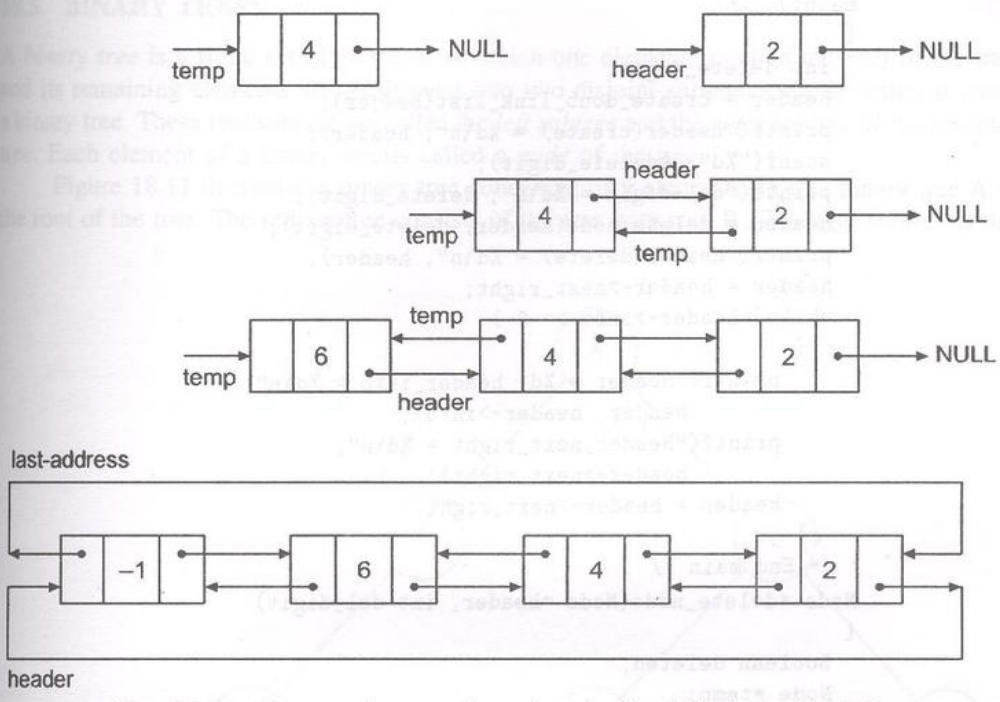


Fig. 18.10 Progressive creation of a doubly linked circular list.

Statements to insert a node

```
temp = (Node *) malloc(sizeof (Node));
temp->next_right = p->next_right;
p->next_right->next_left = temp;
p->next_right = temp;
temp->next_left = p;
temp->info = digit;
```

```
/* Example Program 18.9 */
#include <stdio.h>
#define TRUE 1
#define FALSE 0
typedef struct node
{
  int info;
  struct node *next_right;
  struct node *next_left;
} Node;
Node *header;
typedef unsigned int Boolean;
Node *delete_node(Node *header, int del_digit);
Node *create_doub_link_list(Node *header);
```

```
main()
{
 int delete_digit;
 header = create_doub_link_list(header);
 printf("header(create) = %d\n", header);
 scanf("%d", &delete_digit);
 printf("del-digit = %d\n", delete_digit);
 header = delete_node(header, delete_digit);
 printf("header(delete) = %d\n", header);
 header = header->next_right;
 while( header->info >= 0 )
  {
   printf("Header = %d, header_info = %d\n",
          header, header->info);
   printf("Header_next_right = %d\n",
          header->next_right);
   header = header->next_right;
  }
} /* End main */
Node *delete_node(Node *header, int del_digit)
{
 Boolean deleted;
 Node *temp;
 deleted = FALSE;
 temp = header->next_right;
 while((!deleted) && (temp->info >= 0))
  {
   if (temp->info == del_digit)
    {
     temp->next_right->next_left =
       temp->next_left;
     temp->next_left->next_right =
       temp->next_right;
     deleted = TRUE;
    }
   else
    temp = temp->next_right;
  }
 if (deleted)
  printf("Node deleted\n");
 else
  printf("Node not found and not deleted\n");
 return(header);
} /* End of delete_node */

/* Include function Node *create_doub_link_list here */
```

Program 18.9 Deleting a node from a doubly linked list.

18.5 BINARY TREES

A *binary tree* is a finite set of elements of which one element is called the *root* of the tree and its remaining elements are partitioned into two disjoint subsets, each of which is itself a binary tree. These two subsets are called the *left subtree* and the *right subtree* of the original tree. Each element of a binary tree is called a *node* of the tree.

Figure 18.11 illustrates a binary tree consisting of 9 elements. In this binary tree A is the root of the tree. The left subtree consists of the tree with root B. The right subtree is the

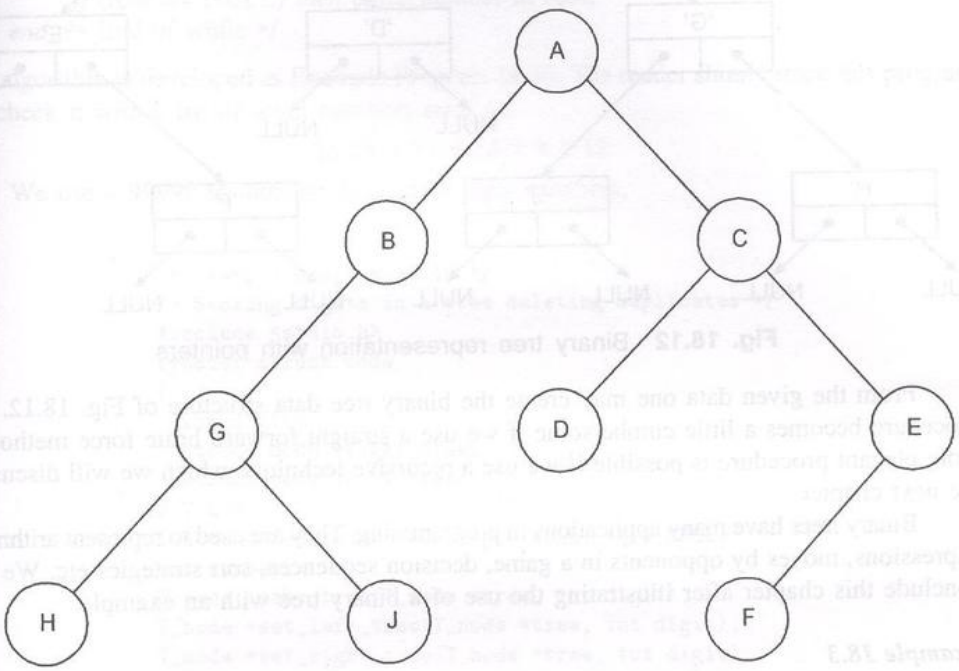


Fig. 18.11 A binary tree.

tree with root C. B has an empty right subtree and its left subtree has root G. C has a left subtree consisting of the single element D. This element which has no further trees emanating from it is known as a *leaf* node. The right subtree of C has E as its root element. E has an empty right subtree and its left subtree is the leaf node F. The node G has a left subtree with a leaf node H and a right subtree with leaf node J. The binary tree of Fig. 18.11 may be represented using pointers as shown in Fig. 18.12. A node of the tree may be described by the following declaration:

```
struct node
        {  char info;
           struct node *left_tree;
           struct node *right_tree;
        };
struct node *tree_node;
```

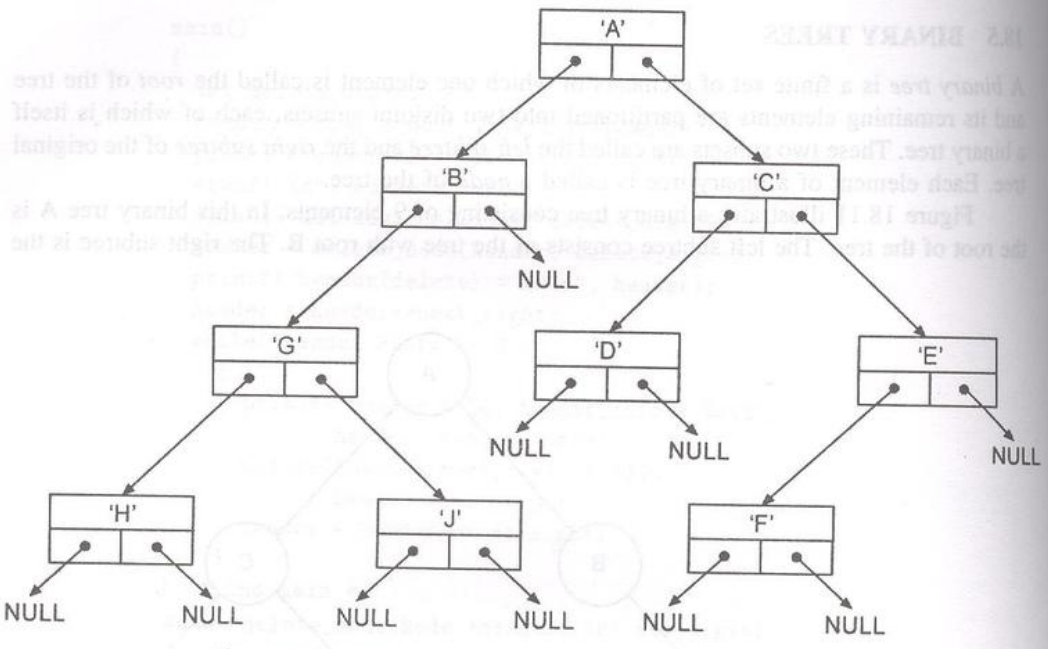


Fig. 18.12 Binary tree representation with pointers.

From the given data one may create the binary tree data structure of Fig. 18.12. The procedure becomes a little cumbersome if we use a straight forward brute force method. A more elegant procedure is possible if we use a recursive technique which we will discuss in the next chapter.

Binary trees have many applications in programming. They are used to represent arithmetic expressions, moves by opponents in a game, decision sequences, sort strategies etc. We will conclude this chapter after illustrating the use of a binary tree with an example.

Example 18.3

A procedure to pick all duplicate numbers in a set of numbers is to be developed. A straight forward method would be to pick a number and compare it with the rest of the numbers in the list. This, however, needs N^2 comparisons for a set of N numbers. A binary tree based algorithm reduces the number of comparisons to a considerable extent. The algorithm is sketched below:

Algorithm to pick duplicate numbers from a list

```
Read(number);
Root node content = number;
Right subtree = null;
Left subtree = null;
while there is data do
  begin
    Read(number);
    Compare number with contents of root node;
```

```
    repeat
      if match then
        declare as duplicate and
        set duplicate found as true
      else
        if number < root then
          root = left subtree root;
        else root = right subtree root;
    until duplicate found or (root == NULL);
    if (root == NULL) then place number in root;
end; /* End of while */
```

This algorithm is developed as Example Program 18.10. The reader should trace this program and check it with a list of input numbers such as:

16 18 3 11 9 18 7 9 3 12

We use – 99999 to indicate the end of input numbers.

```
/* Example Program 18.10 */
/* Storing digits in a tree deleting duplicates */
#include <stdio.h>
typedef struct node
{
  int digit;
  struct node *right_tree;
  struct node *left_tree;
} T_node;
T_node *tree_node= NULL, *p = NULL, *q = NULL;
int number;
T_node *make_tree(T_node *tree, int digit);
T_node *set_left_tree(T_node *tree, int digit);
T_node *set_right_tree(T_node *tree, int digit);

main()
{
  /* Read a number and store in root */
  scanf("%d", &number);
  tree_node = make_tree(tree_node, number);
  while( scanf("%d", &number) != EOF )
    {
      q = tree_node;
      p = tree_node;
      while ((number != p->digit) && (q != NULL))
        {
          p = q;
          if (number < p->digit)
            q = p->left_tree;
          else
            q = p->right_tree;
        } /* End of inner while */
```

```
        if (number == p->digit)
          printf("Number %d is a duplicate\n", number);
        else
          /* Insert number to the right or left of p */
          {
            if (number < p->digit)
              p = set_left_tree(p, number);
            else
              p = set_right_tree(p, number);
          } /* End of else clause */
      } /* End of outer while */
} /* End of main */

T_node *make_tree(T_node *tree_node, int info)
{
  tree_node = (T_node *)malloc(sizeof(T_node));
  tree_node->digit = info;
  tree_node->left_tree = NULL;
  tree_node->right_tree = NULL;
  return(tree_node);
} /* End of make_tree */

T_node *set_left_tree(T_node *tree_node, int info)
{
  T_node *temp;
  if (tree_node->left_tree != NULL)
    printf("Illegal set_left_tree function use\n");
  else
    {
      temp = make_tree(temp, info);
      tree_node->left_tree = temp;
    }
  return(tree_node);
} /* End of set_left_tree */

T_node *set_right_tree(T_node *tree_node, int info)
{
  T_node *temp;
  if (tree_node->right_tree != NULL)
    printf("Illegal set_left_tree function use\n");
  else
    {
      temp = make_tree(temp, info);
      tree_node->right_tree = temp;
    }
  return(tree_node);
} /* End of set_right_tree */
```

Program 18.10 Storing digits in a tree after deleting duplicates.

EXERCISES

18.1 Simulate a stack using a list structure. Write functions to Push and Pop in the stack.

18.2 Simulate a queue data structure with a list structure. A queue is a first-in-first-out data structure. Write procedures to add an element to the queue and retrieve an element from the top of the queue.

18.3 A dequeue is a data structure in which elements may be inserted at either end and removed from either end. The two ends of the dequeue are called left and right. Use a list structure to simulate a dequeue. Write functions to remove elements from left, remove from right, insert at left, insert at right. The functions should take care of operations when the dequeue becomes empty.

18.4 Write routines to do the following in a linear sequential list:

(i) Append an element to the tail of the list.
(ii) Concatenate two lists.
(iii) Reverse a list so that the last element becomes the first and vice versa.
(iv) Insert a node as the n^{th} node of the list.
(v) Copy a list into another.
(vi) Count the number of nodes in a list.
(vii) Delete all even nodes in the list.
(viii) Interchange the n^{th} and k^{th} nodes of a list.

18.5 Create a data structure which would represent the record of each member of a library. The record should contain membership number, name, borrower category and a part which would contain a list of books borrowed by a member.

18.6 Write a function which would scan the member data structure and print out the books borrowed by a specified member.

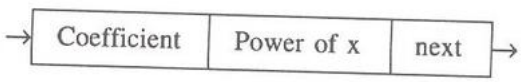
18.7 Write a function which would print out the names of all members who have borrowed more than 6 books.

18.8 Write a function to merge two sorted lists.

18.9 Define a list data structure to facilitate the symbolic manipulation of polynomials. For example a term of the polynomial

$$a_n x^n + a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \ldots + a_1 x^1 + a_0$$

may be represented by the node



(i) Write a routine to add two polynomials represented by two lists.
(ii) Write another routine to symbolically differentiate a polynomial and obtain a list which represents the derivative polynomial.

18.10 A sparse vector is a vector in which more than 80 percent of the components are zero. Represent a sparse vector by a circular list.

18.11 Write a routine to add two sparse vectors represented by two lists.

18.12 Repeat Exercise 18.3 using doubly linked circular list.

18.13 Design doubly linked circular lists to store the following sparse matrix. Pictorially show the lists to represent this matrix.

$$\begin{matrix} 2 & 0 & 0 & 4 \\ 0 & 0 & 5 & 0 \\ 0 & 7 & 0 & 6 \\ 1 & 4 & 0 & 0 \end{matrix}$$

Each node of the list would have the row and column index of the matrix element and the value of the element.

18.14 Write a program to add two n × n sparse matrices and store the result in a third matrix.

18.15 Write a program to count the number of nodes in a binary tree.

19. Recursion

Learning Objectives

In this chapter we will learn:

1. The concept of recursion
2. The advantages and disadvantages of using recursion
3. Applications of recursion in programming

In Chapter 13 we asked the question: Can a function call itself? The answer to this question is yes. In this chapter we will discuss when and how this feature, called *recursion*, is to be used in programming. We will also answer some of the other questions raised in Chapter 13 regarding rules which govern the calling of functions by other functions in C language.

19.1 RECURSIVE FUNCTIONS

Recursion is the name given to the technique of defining a function or a process in terms of itself. The best known example of a recursively defined function is the factorial function defined as follows:

Factorial Function:

$0! = 1$
$n! = n\ (n - 1)!$

Here n! is defined in terms of (n – 1)!, which is in turn defined in terms of (n – 2)! and so on till we reach 0! which is explicitly defined to have a value 1. Any recursive definition must have an explicit definition for some value or values of the argument(s); otherwise the definition would be circular. Some more recursively defined functions are:

Fibonacci numbers:

(i) Fib(0) = 0; Fib(1) = 1;
(ii) Fib (n + 1) = Fib(n) + Fib(n – 1);

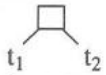
Arithmetic Expression:

(i) <Arithmetic Expression>: = <Variable Name>
(ii) <Arithmetic Expression>: = <Arithmetic Expression> op <Arithmetic Expression>
where op is an arithmetic operator + – * or /.

A tree structure:

(i) □ is a tree

(ii) If t_1 and t_2 are trees, then



is a tree.

A List:

(i) A node is a list

(ii) A list concatenated to a list is a list.

The power of recursion lies in the possibility of defining an infinite set of objects by finite statements. Similarly a recursive program can define an infinite number of computations even though it may not have any explicit loops. Recursive algorithms are appropriate when the problems to be solved or the data structure to be processed are already defined recursively.

If a function R contains an explicit reference to itself, then it is said to be *directly recursive.* If R contains a reference to another function S which contains a reference to R, then R is said to be *indirectly recursive.*

The following function is a recursive definition of the factorial function.

Algorithm 19.1: A recursive function for factorial

```
int factorial(int n)
    {   int result;
        if(n == 0)
            return(1);
        else
            { result = n * factorial(n – 1);
              return(result);
            }
    } /* End of factorial */
```

Functions normally have locally defined objects such as variables, constants, types and other functions. These have no existence or meaning outside the function. Each time a function is invoked recursively a new set of objects are created. Although they have the same names their values have to be preserved till the end of recursion. This puts an extra load on the translator of the language. It is this provision which distinguishes a language that provides recursion from one which does not. For example, the popular languages Fortran 77, BASIC and Cobol do not provide recursion whereas Fortran 90 and Pascal allow recursion. If recursion is required to solve a problem then a BASIC programmer has to explicitly set up the necessary mechanism whereas in C it is not necessary.

19.2 RECURSION VERSUS ITERATION

As mentioned in the last section recursive algorithms are appropriate when the problem to be solved or the data structure to be processed are defined recursively. This does not mean that in such cases recursive algorithms are necessarily the best way of solving these problems. Let

us, for example, consider two algorithms to compute Fibonacci numbers. Fibonacci numbers are recursively defined as follows:

$$fib(0) = 0;\ fib(1) = 1$$
$$fib(n + 1) = fib(n) + fib(n - 1)$$

A recursive algorithm is given below as Algorithm 19.2.

Algorithm 19.2: Recursive algorithm to compute Fibonacci numbers

```
int fib (int n)
    { int result;
      if (n == 0)
          return(0);
      else if(n == 1)
          return(1);
      else
      {
          result = fib(n - 1) + fib(n - 2);
          return(result);
      }
    } /* End of fib */
```

If n = 5 then this algorithm will activate itself 15 times as shown in Fig. 19.1. The value of fib(5) is calculated working backwards starting from the leaf nodes of the tree of Fig. 19.1 and ending at the root node.

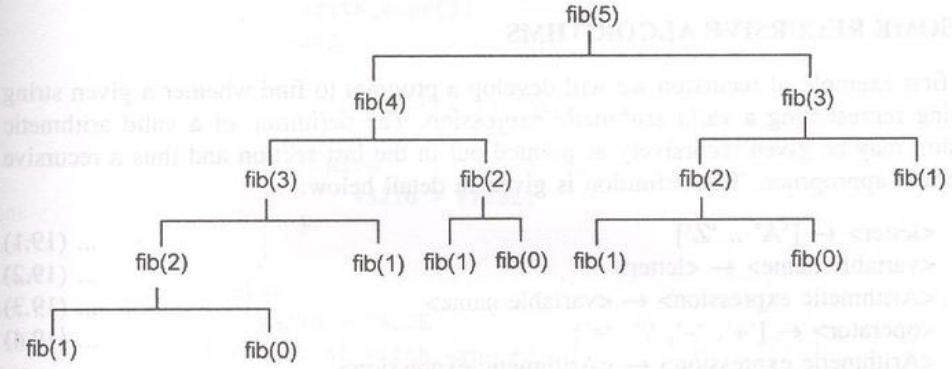


Fig. 19.1 The calls made to fib (*n*) when *n* = 5.

It is clear that the total number of calls to fib and consequently computing it will grow exponentially with *n*. Each call requires reservation of storage for values and links. Such a program to solve the problem needs extra resources. In contrast to this, Fibonacci numbers may be generated iteratively as illustrated in the function of Algorithm 19.3.

In this iterative algorithm the number of computations performed increases linearly with *n*. It is thus much more efficient compared to the recursive program. Observe, however, that the recursive program is concise and "elegant" and closely follows the problem definition.

In spite of the elegance of the recursive methods of programming it is necessary to explore alternative methods based on iteration due to considerations of efficiency. In general a thumb rule one should follow is to use iteration if there is an *obvious* iterative algorithm to solve the problem. If one is forced to mimic recursion by using stacks and operations on stacks then recursion is a better technique to use.

Algorithm 19.3: An iterative algorithm to calculate Fibonacci numbers

```
int fib(int n)
    { int i, prev, current, next;
        if (n == 0)
          return(0);
        else
          if(n == 1)
              return(1);
          else
            { prev = 0; current = 1;
              for (i = 2, i <= n, ++i)
                  { next = prev + current;
                    prev = current;
                    current = next;
                  }
            }
          return(current);
    } /* End of fib */
```

19.3 SOME RECURSIVE ALGORITHMS

As the first example of recursion we will develop a program to find whether a given string is a string representing a valid arithmetic expression. The definition of a valid arithmetic expression may be given recursively as pointed out in the last section and thus a recursive algorithm is appropriate. The definition is given in detail below:

<letter> ← ['A' .. 'Z'] ... (19.1)
<variable name> ← <letter> ... (19.2)
<Arithmetic expression> ← <variable name> ... (19.3)
<operator> ← ['+', '–', '/', '*'] ... (19.4)
<Arithmetic expression> ← <Arithmetic expression>
    <operator> <Arithmetic expression> ... (19.5)

The algorithm faithfully follows the above definition. A symbol is read. We proceed further if it is not the end of line. If it is a letter then the next symbol is read. If it is end of line then we declare the string a valid arithmetic expression as per Eq. (19.2). If the symbol read is not the end of line then the symbol should be an operator as defined by (19.4). If it is an operator we call arith_expn to check if the rest of the string is an arithmetic expression. If it is not an operator then the string is not a valid arithmetic expression. The program is given as Example Program 19.1.

```
/* Example Program 19.1 */
#include <stdio.h>
#define TRUE    1
#define FALSE   0
typedef unsigned int Boolean;
char symbol;
Boolean valid;
void arith_expr(void);
Boolean is_operator(char s);

main()
{
  valid = FALSE;
  arith_expr();
  if (valid)
    printf("Valid arithmetic expression\n");
  else
    printf("Invalid expression\n");
} /* End of main */

void arith_expr(void)
{
  symbol = getchar();
  if ((symbol >= 'A') && (symbol <= 'Z'))
    {
      symbol = getchar();
      if (is_operator(symbol))
       arith_expr();
      else
       {
         if (symbol == '\n')
           valid = TRUE;
         else
           valid = FALSE;
       }
    }
  else
    valid = FALSE;
} /* End of arith_expr */

Boolean is_operator(char s)
{
  if ((s == '+') || (s == '*') || (s == '-')
      || (s == '/'))
    return(TRUE);
  else
    return(FALSE);
} /* End of is_operator */
```

Program 19.1 Checking validity of arithmetic expression.

As the second example we will consider again the example of picking duplicate numbers from a set of numbers. We solved this problem in the last chapter without using recursion. We will solve it now using recursion.

The strategy we will adopt would be to read each number and place it at an appropriate node of a tree, if it is not already in the tree. A procedure to do this is evolved remembering that the basic definition of a tree data structure is recursive. Thus the tree can be built recursively. This is given as Algorithm 19.4.

Algorithm 19.4: Building a tree

```
Buildtree (struct tree *node, int number);
    {if (number > value at node)
      then if right branch of tree is empty
          then {
                  Create a node;
                  Place number in it;
                  Attach node to tree;
               }
          else Buildtree (right branch, number)
      else if (number < value at node)
          then if leftbranch of tree is empty
              then {
                      Create a node;
                      Place number in it;
                      Attach node to tree;
                   }
              else Buildtree (leftbranch, number)
          else Write ('Number is a duplicate')
    } /* End of algorithm 19.4 */
```

This procedure may be used in a program to detect duplicates as given in Algorithm 19.5.

Algorithm 19.5: Program to detect duplicates

```
Read an input value;
Create a root node for a tree and
place it in the node;
While data remain in input
    { Read(input value);
      Buildtree(rootnode, input value);
    }
```

These algorithms are converted as Example Program 19.2. The program has been traced with the following data to illustrate the way the tree is built.

Input data: 15 25 30 25 12 22 14 28

The trace is shown as Fig. 19.2. The student should trace the program and verify the tree building procedure.

```
/* Example Program 19.2 */
/* Traverse a tree in-order left-root-right */
#include <stdio.h>
typedef struct tree
{
  int value;
  struct tree *right;
  struct tree *left;
} Tree;
Tree *root_node;
int inp_value;
void build_tree(Tree *t, int n);
void trav_in_order(Tree *node);
main()
{
  scanf("%d", &inp_value);
  /* Create root of tree and store value read */
  root_node = (Tree *)malloc(sizeof(Tree));
  root_node->value = inp_value;
  root_node->right = NULL;
  root_node->left = NULL;
  while (scanf("%d", &inp_value) != EOF)
    build_tree(root_node, inp_value);
} /* End of main */
void build_tree(Tree *node, int number)
{
  Tree *new_node;
  if (number > node->value)
    {
      if (node->right == NULL)
        /* If terminal node on right branch */
        {
          /* Create node and link to right */
          new_node = (Tree *)malloc(sizeof(Tree));
          new_node->value = number;
          new_node->left = NULL;
          new_node->right = NULL;
          node->right = new_node;
        }
      else
        build_tree(node->right, number);
    }
  else
    {
      if (number < node->value)
        if (node->left == NULL)
          /* If terminal node on left branch */
          {
            /* Create node and link to left */
            new_node->value = number;
            new_node->left = NULL;
            new_node->right = NULL;
            node->left = new_node;
          }
```

```
            else
              build_tree(node->left, number);
          else
            printf("duplicate number = %d\n", number);
        }
    } /* End of build_tree */

    void trav_in_order(Tree *node)
    {
      if (node != NULL)
        {
          /* Visit left node */
          trav_in_order(node->left);
          printf("%d ", node->value);
          /* Visit right node */
          trav_in_order(node->right);
        }
    } /* End of trans_in_order */
```

Program 19.2 Building a tree removing duplicate integers.

19.4 TREE TRAVERSAL ALGORITHMS

The tree was generated by Algorithm 19.4 systematically keeping smaller numbers in a left subtree and larger numbers in a right subtree. This tree has the property that the contents of each node in the left subtree of a node t are less than the contents of t. Thus if we travel through all the nodes of the tree in the order left node, rootnode, rightnode starting from the left most node of the tree, we will pass through the numbers in ascending sequence. This method of travelling through the nodes in the order left, root, right is known as *in order traversal of the tree.*

We can specify this traversal recursively as:

1. Traverse the left subtree *inorder*
2. Visit the root
3. Traverse the right subtree *inorder*

The above method may be expressed as Algorithm 19.6.

Algorithm 19.6: Inorder tree traversal to print numbers in ascending order

```
Travinorder(struct tree *node)
  { if node <> nil then
      { Travinorder (node->left);
        Write (node->value, ' ');
        Travinorder (node->right)
      }
  }
```

This algorithm is converted to Example Program 19.3.

With the data given for Example Program 19.2 the order in which nodes will be "visited" and values printed is shown in Fig. 19.3. If the numbers in the list are to be printed in descending order then we would again traverse the tree in order but from right to left.

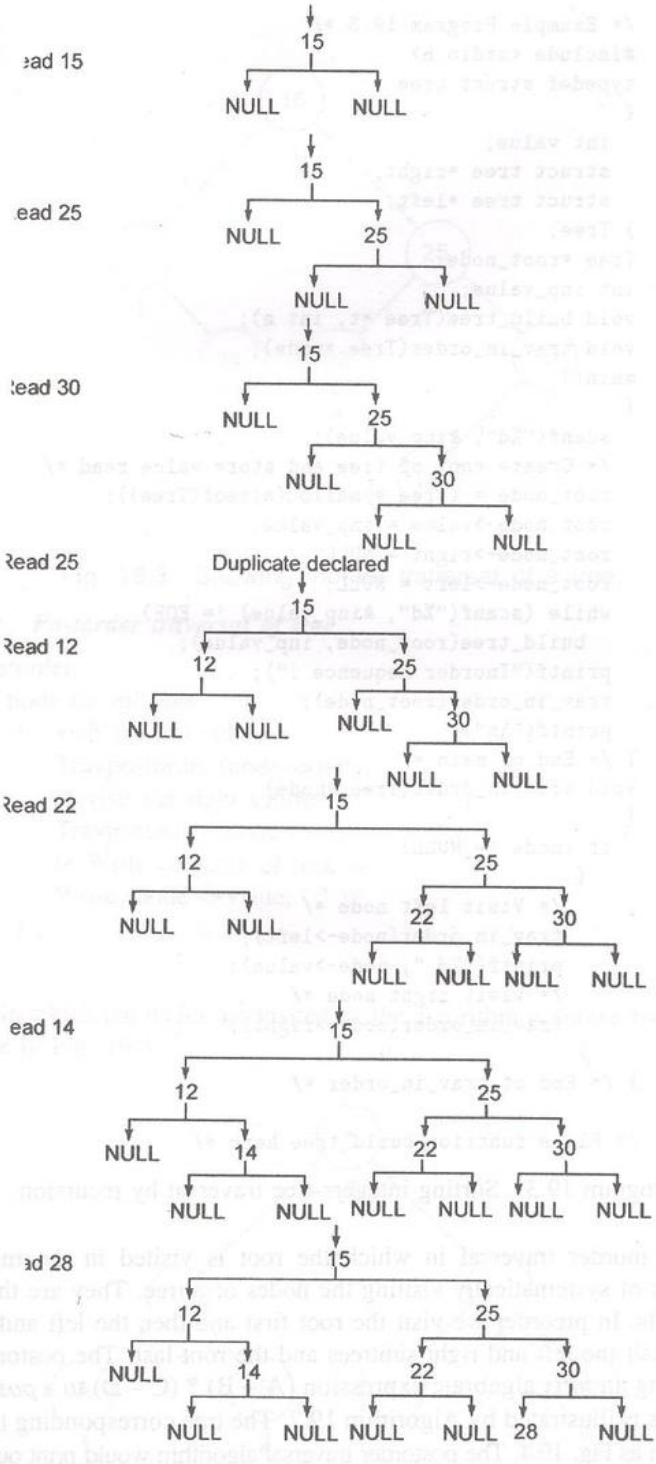


Fig. 19.2 Illustrating recursive tree building.