PROGRAMMING IN C

A Step-by-Step Guide to Programming in C Master the Fundamentals and Unlock Your Coding Potential Today.



FOR BTECH/BCA/MCA/BSC.IT & DIPLOMA STUDENTS

Unit 1 Introduction to C Programming

Structure:

1.1 Introduction

Objectives

1.2 Features of C and its Basic Structure

1.3 Simple C programs

1.4 Constants

Integer Constants

Real Constants

Character Constants

String Constants

Backslash Character Constants

1.5 Concept of an Integer and Variable

1.6 Rules for naming Variables and assigning values to variables 1.7 Summary

1.8 Terminal Questions

1.9 Answers to Self Assessment Questions

1.10 Answers to Terminal Questions

1.11 Exercises

1.1 Introduction

C is a general-purpose, structured programming language. Its instructions consist of terms that resemble algebraic expressions, augmented by certain English keywords such as **if, else, for, do** and **while**. C was the offspring of the 'Basic Combined Programming Language' (BPCL) called B, developed in the 1960's at Cambridge University. B language was modified by Dennis Ritchie and was implemented at Bell laboratories in 1972. The new language was named C. Since it was developed along with the UNIX operating system, it is strongly associated with UNIX. This operating system, which was also developed at Bell laboratories, was coded almost entirely in C.

In C, the *type* of a variable determines what kinds of values it may take on. The type of an object determines the set of values it can have and what operations can be performed on it. This is a fairly formal, mathematical definition of what a type is, but it is traditional (and meaningful). There are several implications to remember:
1.The "set of values" is finite. C's int type cannot represent *all* of the integers; its float type cannot represent *all* floating-point numbers.

2.When you're using an object (that is, a variable) of some type, you may have to remember what values it can take on and what operations you can perform on it. For example, there are several operators which play

with the binary (bit-level) representation of integers, but these operators are not meaningful for and may not be applied to floating-point operands. 3.When declaring a new variable and picking a type for it, you have to
keep in mind the values and operations you'll be needing.

This unit will introduce you to C programming with its various features, structure and how the C programs are constructed with simple examples. **Objectives:**
At studying this unit, you should be able to:

discuss the features of C programming language
explain the basic structure of a C program
write simple programs in C
construct and use the concept of Constants and Integers
use the variable and its declaration in C

1.2 Features of C and its Basic Structure

C is characterized by the ability to write very concise source programs, due in part to the large number of operators included within the language.

It has a relatively small instruction set, though actual implementations include extensive library functions which enhance the basic instructions. The language encourages users to write additional library functions of their own. Thus, the features and capabilities of the language can easily be extended by the user.

C compilers are commonly available for computers of all sizes. The compilers are usually compact, and they generate object programs that are small and highly efficient when compared with programs compiled from other high-level languages.

Another important characteristic of C is that its programs are highly portable, even more so than with other high-level languages. The reason for this is that C relegates most computer dependent features to its library functions. Thus, every version of C is accompanied by its own set of library functions, which are written for the particular characteristics of the host computer.

A C program can be viewed as a group of building blocks called functions. A function is a subroutine that may include one or more statements designed to perform a specific task. To write a C program we first create functions and then put them together. A C program may contain one or more sections shown in Fig. 1.1.

The documentation section consists of a set of *comment* (remarks) lines giving the name of the program, the author and other details which the programmer would like to use later. Comments may appear anywhere within a program, as long as they are placed within the delimiters /* and */ (e.g., /*this is a comment*/). Such comments are helpful in identifying the program's principal features or in explaining the underlying logic of various program features.

Documentation section Link section
Definition section
Global declaration section main() function section {

Declaration part Executable part
}

Subprogram section

Function 1
Function 2
.
.
Function n

Figure 1.1: Sections in a C program

The link section provides instructions to the compiler to link functions from the system library. The definition section defines all symbolic constants.

There are some variables that are used in more than one function. Such variables are called *global* variables and are declared in the global declaration section that is outside of all the functions.

Every C program must have one **main** function section. This section contains two parts, declaration part and executable part. The declaration part declares all the variables used in the executable part. There is at least one statement in the executable part. These two parts must appear between opening and closing braces ({ and }). The program execution begins at the opening brace and ends at the closing brace. The closing brace of the main function section is the logical end of the program. All statements in the declaration and executable parts end with a semicolon(;).

The subprogram section contains all the user-defined functions that are called in the **main** function. User-defined functions are generally placed immediately after the **main** function, although they may appear in any order.

All sections, except the **main** function section may be absent when they are not required.

Self Assessment Questions

1.Using C language programmers can write their own library functions.
(True/False)

2.C is a _____ level programming language.

3.The documentation section contains a set of _____ lines.

4.Every C program must have one **main()** function. (True/False)

1.3 Simple C Programs

Let us see a simple c program given below: #include <stdio.h>
main()

{
printf("Hello, world!\n");
return 0;
If you have a C compiler, the first thing to do is figure out how to type this program in and compile it and run it and see where its output went.

The first line is practically boilerplate; it will appear in almost all programs we write. It asks that some definitions having to do with the “Standard I/O Library" be included in our program; these definitions are needed if we are to call the library function **printf** correctly.
The second line says that we are defining a function named **main**. Most of the time, we can name our functions anything we want, but the function name **main** is special: it is the function that will be “called" first when our program starts running. The empty pair of parentheses indicates that our **main** function accepts no *arguments*, that is, there isn't any information which needs to be passed in when the function is called.
The braces { and } surround a list of statements in C. Here, they surround the list of statements making up the function **main**.
The line
printf("Hello, world!\n");
is the first statement in the program. It asks that the function **printf** be called; **printf** is a library function which prints

formatted output. The parentheses surround **printf**'s argument list: the information which is handed to it which it should act on. The semicolon at the end of the line terminates the statement.

printf 's first (and, in this case, only) argument is the string which it should print. The string, enclosed in double quotes (""), consists of the words “Hello, world!" followed by a special sequence: \n. In strings, any two-character sequence beginning with the backslash \ represents a single special character. The sequence \n represents the “ 'new line" character, which prints a carriage return or line feed or whatever it takes to end one line of output and move down to the next. (This program only prints one line of output, but it's still important to terminate it.)

The second line in the **main** function is

return 0;

In general, a function may return a value to its caller, and **main** is no exception. When **main** returns (that is, reaches its end and stops functioning), the program is at its end, and the return value from **main** tells the operating system (or whatever invoked the program that **main** is the main function of) whether it succeeded or not. By convention, a return value of 0 indicates success.

Program 1.1: Area of a circle

Here is an elementary C program that reads in the radius of a circle, calculates the area and then writes the calculated result.

```
#include <stdio.h> /* Library file access */
/* program to calculate the area of a circle */ /* Title (Comment) */ main() /* Function heading */
{
float radius, area; /*Variable declarations */
printf(“Radius=?”); /* Output statement(prompt) */
```

```
scanf(“%f”, &radius); /* Input statement */
area=3.14159*radius*radius; /* Assignment statement */
printf(“Area=%f”,area); /* Output statement */
}
```

Program 1.2: Print a few numbers

Here is a program to illustrate a simple loop.

```
#include <stdio.h>
/* print a few numbers, to illustrate a simple loop */
main()
{
int i;
for(i = 0; i < 10; i = i + 1) /* Looping statement */
printf("i is %d\n", i);
return 0;
```

Program 1.3: Program to add two numbers

```
#include <stdio.h>
main()
{
int i,j,k; // Defining variables
i = 6; // Assign values
j = 8;
```

```
k = i + j;
printf("sum of two numbers is %d \n",k); // Printing results }
```

Self Assessment Questions

5.The information that needs to be passed in when a function is called is _____.

6.The main() function doesn't return any value. (True/False)

1.4 Constants

Constants in C refer to fixed values that do not change during the execution of a program. C supports several types of constants as illustrated in Fig. 1.1.

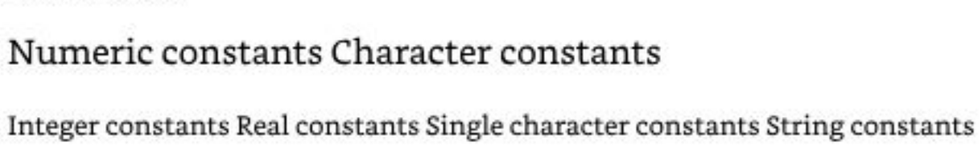


Fig. 1.1

1.4.1 Integer constants An integer constant refers to a sequence of digits. There are three types of integers, namely *decimal, octal and hexadecimal.*

Decimal integers consist of a set of digits, 0 through 9, preceded by an optional – or +.

Examples: 12, -546, 0, 354647, +56

An *octal* integer constant consists of any combination of digits from the set 0 through 7, with a leading 0.

Examples: 045, 0, 0567

A sequence of digits preceded by 0x or 0X is considered as *hexadecimal* integer. They may also include alphabets A through F or a through f. The letters A through F represent numbers 10 through 15.

Examples: 0X6, 0x5B, 0Xbcd, 0X

The largest integer value that can be stored is machine-dependent. It is 32767 on 16-bit machines and 2,147,483,647 on 32-bit machines. It is also possible to store larger integer constants on these machines by appending *qualifiers* such as U, L and UL to the constants.

Examples: 54637U or 54637u (unsigned integer)
65757564345UL or 65757564345ul (unsigned long integer)
7685784L or 7685784l (long integer)

Program 1.1: Program to represent integer constants on a 16-bit computer

```
/* Integer numbers on 16-bit machine */
main()
{
printf("Integer values\n\n");
printf("%d %d %d\n", 32767,32767+1,32767+10);
printf("\n");
printf("Long integer values\n\n");
printf("%ld %ld %ld\n", 32767L, 32767L+1L, 32767L+10L);
}
```

Type and execute the above program and observe the output

1.4.2 Real constants

The numbers containing fractional parts like 67.45 are called real (or floating point) constants.
Examples: 0.0045, -8.5, +345.678
A real number may also be expressed in *exponential (scientific)* notation. The general form is:
mantissa **e** *exponent*

The mantissa is either a real number expressed in decimal notation or an integer. The exponent is an integer number with an optional plus or minus sign. The letter **e** separating the mantissa and the exponent can be written in either lowercase or uppercase.

Examples: 04e4, 12e-2, -1.3E-2
7500000000 may be written as 7.5E9 or 75E8.

Floating point constants are normally represented as double-precision quantities. However, the suffixes f or F may be used to force single precision and l or L to extend double-precision further.

1.4.3 Character constants

A single character constant (or simple character constant) contains a single character enclosed within a pair of single quote marks.
Examples: '6', 'X', ';'

Character constants have integer values known as ASCII values. For example, the statement
printf("%d", 'a');

would print the number 97, the ASCII value of the letter a. Similarly, the statement

printf("%c", 97);
would print the letter a.

1.4.4 String constants

A string constant is a sequence of characters enclosed within double quotes. The characters may be letters, numbers, special characters and blank space.
Examples: "Hello!", "1947", "5+3"

1.4.5 Backslash character constants

C supports some special backslash character constants that are used in output functions. A list of such backslash character constants is given in Table 1.1. Note that each one of them represents one character, although they consist of two characters. These character combinations are called *escape sequences*.

Table 1.1: Backslash character constants

Constant	Meaning
'\b'	back space
'\f'	form feed
'\n'	new line
'\r'	carriage return
'\t'	horizontal tab
'\v'	vertical tab
'\''	single quote
'\" '	double quote

‘\\’ backslash
‘\0’ null

Self Assessment Questions

7.An octal integer constant consists of any combination of digits from the set _____ through _____.
8.A sequence of digits preceded by 0x or 0X is considered as _____ integer.
9.A string constant is a sequence of _____ enclosed within double quotes.

1.5 Concept of an Integer and Variable

Integers are whole numbers with a range of values supported by a particular machine. Generally, integers occupy one word of storage, and since the word sizes of machines vary (typically, 16 or 32 bits) the size of an integer that can be stored depends on the computer. If we use 16 bit word length, the size of the integer value is limited to the range -32768 to +32767 (that is,
-215 to +2 15 -1). A signed integer uses one bit for sign and 15 bits for the magnitude of the number. Similarly, a 32 bit word length can store an integer ranging from -2,147,483,648 to 2,147,483,647.

In order to provide some control over the range of numbers and storage space, C has three classes of integer storage, namely **short int, int ,** and **long int,** in both signed and unsigned forms. For example, **short int** represents fairly small integer values and requires half the amount of storage as a regular **int number** uses. A signed integer uses one bit for sign and 15 bits for the magnitude of the number, therefore, for a 16 bit machine, the range of unsigned integer

numbers will be from 0 to 65,535. We declare **long** and **unsigned integers** to increase the range of values. The use of qualifier **signed** on integers is optional because the default declaration assumes a signed number. The Table 1.2 shows all the allowed combinations of basic types and qualifiers and their size and range on a 16- bit machine.

Table 1.2: Basic types, qualifiers and their size

Type	Size (bits)	Range
int or signed int	16	-32,768 to 32,767
unsigned int	16	0 to 65535
short int or signed short int	8	-128 to 127
unsigned short int	8	0 to 255
long int or signed long int	32	-2,147,483,648 to 2,147,483,647
unsigned long int	32	0 to 4,294,967,295

Informally, a *variable* (also called an *object*) is a place where you can store a value so that you can refer to it unambiguously. A variable needs a name. You can think of the variables in your program as a set of boxes, each with a label giving its name; you might imagine that storing a value “in" a variable consists of writing the value on a slip of paper and placing it in the box.

Now let us see how integers and variables are declared. A *declaration* tells the compiler the name and type of a variable you'll be using in your program. In its simplest form, a declaration consists of the type, the name of the variable, and a terminating semicolon:

```
int i;
```

The above statement declares an integer variable i.

```
long int i1, i2;
```

We can also declare several variables of the same type in one declaration, separating them with commas as shown above.

The placement of declarations is significant. You can't place them just anywhere (i.e. they cannot be interspersed with the other statements in your program). They must either be placed at the beginning of a function, or at the beginning of a brace-enclosed block of statements, or outside of any function. Furthermore, the placement of a declaration, as well as its storage class, controls several things about its *visibility* and *lifetime*, as we'll see later.
You may wonder *why* variables must be declared before use. There are two reasons:

1.It makes things somewhat easier on the compiler; it knows right away what kind of storage to allocate and what code to emit to store and manipulate each variable; it doesn't have to try to intuit the programmer's intentions.

2.It forces a bit of useful discipline on the programmer: you cannot introduce variables wherever you wish; you must think about them enough to pick appropriate types for them. (The compiler's error messages to you, telling you that you apparently forgot to declare a variable, are as often helpful as they are a nuisance: they're helpful when they tell you that you misspelled a variable, or forgot to think about exactly how you were going to use it.)

Most of the time, it is recommended to write one declaration per line. For the most part, the compiler doesn't care what order declarations are in. You can order the declarations alphabetically, or in the order that they're used, or to put related declarations next to each other. Collecting all variables of the same type together on one line essentially orders declarations by type, which isn't a very useful order (it's only slightly more useful than random order).

A declaration for a variable can also contain an initial value. This *initializer* consists of an equal sign and an expression, which is usually a single constant:

```
int i = 1;
int i1 = 10, i2 = 20;
```

Self Assessment Questions

10.The size of the Integers in C language is same in all the machines. (True/False)

11.A _____ is a place where we can store values.

12.Size of int is _____ bits.

13.Which of the following tells the compiler the name and type of a variable you'll be using in your program? (declaration)

(a) declaration

(b) variables

(c) integers

(d) assignments

14.The _____ consists of an equal sign and an expression, which is usually a single constant. (initializer)

15.A single declaration statement can contain variables of different types. (True/False)

1.6 Rules for naming Variables and assigning values to variables

Within limits, you can give your variables and functions any names you want. These names (the formal term is “identifiers") consist of letters, numbers, and underscores. For our purposes, names must begin with a letter. Theoretically, names can be as long as you want, but extremely long ones get tedious to type after a while, and the compiler is not required to keep track of extremely long ones perfectly. (What this means is that if you were to name a variable, say, *supercalafragalisticespialidocious*, the compiler might get lazy and pretend that you'd named it *supercalafragalisticespialidocio*, such that if you later misspelled it *supercalafragalisticespialidociouz*, the compiler wouldn't

catch your mistake. Nor would the compiler necessarily be able to tell the difference if for some perverse reason you deliberately declared a second variable named *supercalafragalisticespialidociouz*.)
The capitalization of names in C is significant: the variable names *variable*, *Variable*, and *VARIABLE* (as well as silly combinations like variAble) are all distinct.

A final restriction on names is that you may not use *keywords* (the words such as **int** and **for** which are part of the syntax of the language) as the names of variables or functions (or as identifiers of any kind).

Now let us see how you can assign values to variables. The assignment operator = assigns a value to a variable. For example,
x = 1;

sets x to 1, and
a = b;
sets a to whatever b's value is. The expression
i = i + 1;
is, as we've mentioned elsewhere, the standard programming idiom for increasing a variable's value by 1: this expression takes i's old value, adds 1 to it, and stores it back into i. (C provides several “shortcut" operators for modifying variables in this and similar ways, which we'll meet later.)

We've called the = sign the “assignment operator" and referred to “assignment expressions" because, in fact, = *is* an operator just like + or -. C does not have “assignment statements"; instead, an assignment like a = b is an expression and can be used wherever any expression can appear. Since it's an expression, the assignment a = b has a value, namely, the same value that's assigned to a. This value can then be used in a larger expression; for example, we might write
c = a = b;

Which is equivalent to?

c = (a = b);

and assigns b's value to both a and c. (The assignment operator, therefore, groups from right to left.) Later we'll see other circumstances in which it can be useful to use the value of an assignment expression.

It's usually a matter of style whether you initialize a variable with an

initializer

in its declaration or with an assignment expression near where you first use it. That is, there's no particular difference between

int a = 10;

and

int a;

/* later... */

a = 10;

Self Assessment Questions

16.In C, variable names are case sensitive. (True/False)

17.A variable name in C consists of letters, numbers and _____.

1.7 Summary

C is a general-purpose, structured programming language. Its instructions consist of terms that resemble algebraic expressions, augmented by certain English keywords such as **if, else, for, do** and **while**. C is characterized by the ability to write very concise source programs, due in part to the large number of operators included within the language.

Every C program consists of one or more functions, one of which must be called **main.** The program will always begin by executing the **main** function. Additional function definitions may precede or follow **main**.

Integers are whole numbers with a range of values supported by a particular machine. Generally, integers occupy one word of storage, and since the word sizes of machines vary (typically, 16 or 32 bits) the size of an integer that can be stored depends on the computer. A *variable* (also called an *object*) is a place where you can store a value. A *declaration* tells the compiler the name and type of a variable you'll be using in your program. The assignment operator = assigns a value to a variable.

1.8 Terminal Questions

1.Explain the history of C language.
2.What are the advantages of C language?
3.What are the major components of a C program?
4.What significance is attached to the function **main**?
5.What are arguments? Where do arguments appear within a C program?
6.Distinguish between signed and unsigned integers.
7.What are the components of declaration statement?
8.State the rules for naming a variable in C.
9.What is assignment operator and what are its uses?

1.9 Answers to Self Assessment Questions

1. True
2. High
3. Comment
4. True
5. Arguments
6. False
7.(0, 7)
8. Hexadecimal
9. characters
10. False
11. Variable
12. 16
13.(a) declaration
14. initializer
15. False
16. True
17. Underscores

1.10 Answers to Terminal Questions

1.C was developed by Dennis Ritchie and came after B in the year 1972. (Refer Section 1.1 for more details)
2.C is fast, efficient and structured. (Refer to section 1.2 for more details)
3.Documentation section, Link section, Definition section, Global declaration section, main() function section, Subprogram section
4.**main()** is the function that will be “called" first when our program starts running.
5.The arguments are symbols that represent information being passed between the function and other parts of the program. They appear in the function heading.
6.A signed integer uses one bit for sign and remaining bits for the magnitude of the number, whereas an unsigned integer uses all the bits to represent magnitude.
7.A declaration consists of the type, the name of the variable, and a terminating semicolon.
8.Variables (the formal term is “identifiers") consist of letters, numbers, and underscores. The capitalization of names in C is significant. You may not use *keywords* (the words such as int and for which are part of the syntax of the language) as the names of variables or functions (or as identifiers of any kind).
9.The assignment operator (=) assigns a value to a variable.

1.11 Exercises

1.Explain the basic structure of a C program with an example.
2.What are the different steps in executing a C program?

3.Write a C program to convert Celsius to Fahrenheit and vice versa.

4.Write a program in C to add, subtract, multiply any 3 numbers.

Unit 2 Operators and Expressions

Structure:

2.1 Introduction

Objectives

2.2 Arithmetic Operators

2.3 Unary Operators

2.4 Relational and Logical Operators

2.5 The Conditional Operator

2.6 Library Functions

2.7 Bitwise Operators

2.8 The Increment and Decrement Operators

2.9 The Size of Operator

2.10 Precedence of operators

2.11 Summary

2.12 Terminal Questions

2.13 Answers to Self Assessment Questions

2.14 Answers to Terminal Questions

2.15 Exercises

2.1 Introduction

In the previous unit, you learned about the various features and structure of C programs. You also learned how the variables in C are declared. In this unit, you will learn about the operators that are available in C and how the expressions can be formed to get the solutions of any problems.

C supports a rich set of operators. An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations. Operators are used in programs to manipulate data and variables. They usually form a part of the mathematical or logical expressions.

C operator can be classified into a number of categories. They include:

1.Arithmetic operators
2.Unary operator
3.Relational operators
4.Logical operators
5.Conditional operator
6.Bitwise operators
7.Increment and Decrement operators

Objectives:

After studying this subject, you should be able to: explain different categories of operators
use operators on many operands
distinguish precedence and associativity of operators explain library functions and their use
write small programs using different types of operators

2.2 Arithmetic Operators

The basic operators for performing arithmetic are the same in many computer languages:
+ addition
- subtraction
* multiplication
/ division
% modulus (remainder)
The - operator can be used in two ways: to subtract two numbers (as in a - b), or to negate one number (as in -a + b or a + -b).
When applied to integers, the division operator / discards any remainder, so 1 / 2 is 0 and 7 / 4 is 1. But when either operand is a floating-point quantity (a real number), the division operator yields a floating-point result, with a potentially nonzero fractional part. So 1 / 2.0 is 0.5, and 7.0 / 4.0 is 1.75. The *modulus* operator % gives you the remainder when two integers are divided: 1 % 2 is 1; 7 % 4 is 3. (The modulus operator can only be applied to integers.)

An additional arithmetic operation you might be wondering about is exponentiation. Some languages have an exponentiation operator (typically ^or **), but C doesn't. (To square or cube a number, just multiply it by itself.)

Multiplication, division, and modulus all have higher *precedence* than addition and subtraction. The term “precedence" refers to how “tightly" operators bind to their operands (that is, to the things they operate on). In mathematics, multiplication has higher precedence than addition, so 1 + 2 * 3 is 7, not 9. In other words, 1 + 2 * 3 is equivalent to 1 + (2 * 3). C is the same way.

All of these operators “group" from left to right, which means that when

two
or more of them have the same precedence and participate next to each other in an expression, the evaluation conceptually proceeds from left to right. For example, 1 - 2 - 3 is equivalent to (1 - 2) - 3 and gives -4, not +2. (“Grouping" is sometimes called *associativity*, although the term is used somewhat differently in programming than it is in mathematics. Not all C operators group from left to right; a few groups from right to left.)

Whenever the default precedence or associativity doesn't give you the grouping you want, you can always use explicit parentheses. For example, if you want to add 1 to 2 and then multiply the result by 3, you could write (1 + 2) * 3.

Program 2.1: Program that shows the use of integer arithmetic to convert a given number of days into months and days.

```
/* Program to convert days to months and days */
main()
{
int months, days;
printf(“Enter days\n”);
scanf(“%d”,&days);
months=days/30;
days=days%30;
printf(“Months=%d Days=%d”, months,days);
}
```

Self Assessment Questions

1.What is the value of the arithmetic expression: 14 % 3 + 7 % 2 (a)1 (b) 2 (c)3 (d)4

2._____ operator can be only applied to integers.

2.3 Unary Operator

A unary operator acts upon a single operand to produce a new value.

Unary Minus

The most well known unary operator is minus, where a minus sign precedes a constant, variable or expression. In C, all numeric constants are positive. Therefore, a negative number is actually a positive constant preceded by a unary minus, for example:

-3

2.4 Relational and Logical operators

An if statement like

if(x > max)

max = x;

is perhaps deceptively simple. Conceptually, we say that it checks whether the condition x > max is “true" or “false". The mechanics underlying C's conception of “true" and “false," however, deserve some explanation. We need to understand how true and false values are represented, and how they are interpreted by statements like if.

As far as C is concerned, a true/false condition can be represented as an integer. (An integer can represent many values; here we care about only two values: “true" and “false." The study of mathematics involving only

two
values is called Boolean algebra, after George Boole, a mathematician who refined this study.) In C, “false" is represented by a value of 0 (zero), and “true" is represented by any value that is nonzero. Since there are many nonzero values (at least 65,534, for values of type int), when we have to pick a specific value for “true," we'll pick 1.
The *relational operators* such as <, <=, >, and >= are in fact operators, just like +, -, *, and /. The relational operators take two values, look at them, and “return" a value of 1 or 0 depending on whether the tested relation was true or false. The complete set of relational operators in C is:
< less than
<= less than or equal
>greater than
>= greater than or equal
== equal
!= not equal
For example, 1 < 2 is true(1), 3 > 4 is false(0), 5 == 5 is true(1), and 6 != 6 is

false(0).

The equality-testing operator is ==, not a single =, which is assignment. If you accidentally write
if(a = 0)
(and you probably will at some point; everybody makes this mistake), it will *not* test whether a is zero, as you probably intended. Instead, it will *assign* 0 to a, and then perform the “true" branch of the if statement if a is *non*zero. But a will have just been assigned the value 0, so the “true" branch will never be taken! (This could drive you crazy while

debugging -you wanted to do something if a was 0, and after the test, a *is* 0, whether it was supposed to be or not, but the “true" branch is nevertheless not taken.)

The relational operators work with arbitrary numbers and generate true/false values. You can also combine true/false values by using the *Boolean operators(also called the logical operators)*, which take true/false values as operands and compute new true/false values. The three Boolean operators are:
&& AND
|| OR
!NOT (takes one operand, “*unary*")
The && (“and") operator takes two true/false values and produces a true (1)result if both operands are true (that is, if the left-hand side is true **and** the right-hand side is true). The || (“or") operator takes two true/false values and produces a true (1) result if either operand is true. The ! (“not") operator takes a single true/false value and negates it, turning false to true and true to false (0 to 1 and nonzero to 0). The logical operators && and || are used when we want to test more than one condition and make decisions.

For example, to test whether the variable i lies between 1 and 10, you might use
if(1 < i && i < 10)
...

Here we're expressing the relation “i is between 1 and 10" as “1 is less than i **and** i is less than 10."
It's important to understand why the more obvious expression if(1 < i < 10) /* WRONG */
would not work. The expression 1 < i < 10 is parsed by the compiler analogously to 1 + i + 10. The expression 1 + i + 10 is parsed as (1 + i) + 10 and means “add 1 to i, and then add the result to 10." Similarly, the expression 1 < i < 10 is parsed as (1 < i) < 10 and means “see if 1 is less than i, and then see if the result is less than 10." But in this case, “the result" is 1 or

0, depending on whether i is greater than 1. Since both 0 and 1 are less than 10, the expression 1 < i < 10 would *always* be true in C, regardless of the value of i!

Relational and Boolean expressions are usually used in contexts such as an if statement, where something is to be done or not done depending on some condition. In these cases what's actually checked is whether the expression representing the condition has a zero or nonzero value. As long as the expression is a relational or Boolean expression, the interpretation is just what we want. For example, when we wrote
if(x > max)

the > operator produced a 1 if x was greater than max, and a 0 otherwise. The if statement interprets 0 as false and 1 (or any nonzero value) as true.

But what if the expression is not a relational or Boolean expression? As far as C is concerned, the controlling expression (of conditional statements like if) can in fact be *any* expression: it doesn't have to "look like" a Boolean expression; it doesn't have to contain relational or logical operators. All C looks at (when it's evaluating an if statement, or anywhere else where it needs a true/false value) is whether the expression evaluates to 0 or nonzero. For example, if you have a variable x, and you want to do something if x is nonzero, it's possible to write
if(x)

statement

and the statement will be executed if x is nonzero (since nonzero means "true").
This possibility (that the controlling expression of an if statement doesn't have to "look like" a Boolean expression) is both useful and potentially confusing. It's useful when you have a variable or a function that is "conceptually Boolean," that is, one that you consider to hold a true or false (actually nonzero or zero) value. For example, if you have a variable

verbose which contains a nonzero value when your program should run in verbose mode and zero when it should be quiet, you can write things like if(verbose)
printf("Starting first pass\n");
and this code is both legal and readable, besides which it does what you want. The standard library contains a function isupper() which tests whether a character is an upper-case letter, so if c is a character, you might write if(isupper(c))
...
Both of these examples (verbose and isupper()) are useful and readable. However, you will eventually come across code like
if(n)
average = sum / n;
where n is just a number. Here, the programmer wants to compute the average only if n is nonzero (otherwise, of course, the code would divide by 0), and the code works, because, in the context of the if statement, the trivial expression n is (as always) interpreted as “true" if it is nonzero, and “false" if it is zero.

“Coding shortcuts" like these can seem cryptic, but they're also quite common, so you'll need to be able to recognize them even if you don't choose to write them in your own code. Whenever you see code like if(x)
or
if(f())
where x or f() do not have obvious “Boolean" names, you can read them as “if x is nonzero" or “if f() returns nonzero."

Self Assessment Questions

3.The logical operators _____ and _____ are used when we want to test more than one condition and make decisions.

4.State whether the following statement is correct or not. (Correct/Incorrect)

```
if(a<4<c)
b=c;
```

2.5 The Conditional Operator

The Conditional operator (ternary operator) pair "?:" is available in C to construct conditional expressions of the form

expr1?*expr2*:*expr3*

where *expr1*, *expr2* and *expr3* are expressions.

The operator ? : works as follows: *expr1* is evaluated first. If it is nonzero(true), then the expression *expr2* is evaluated and becomes the value of the expression. If *expr1* is false, *expr3* is evaluated and its value becomes the value of the expression. For example, consider the following statements:

```
a=100;
b=200;
c=(a>b)?a:b;
```

In this example, c will be assigned the value of b. This can be achieved using the if..else statements as follows:

```
if(a>b)
c=a;
else
c=b;
```

2.6 Library Functions

The C language is accompanied by a number of *library functions or built in functions* that carry out various commonly used operations or calculations. There are library functions that carry out standard input/output operations, functions that perform operations on characters, functions that perform operations on strings and functions that carry out various mathematical calculations.

Functionally similar library functions are usually grouped together as object programs in separate library files.
A library function is accessed simply by writing the function name, followed by a list of arguments that represent information being passed to the function. A function that returns a data item can appear anywhere within an expression in place of a constant or an identifier. A function that carries out operations on data items but does not return anything can be accessed simply by writing the function name.
A typical set of library functions will include a large number of functions that are common to most C compilers, such as those shown in table 2.1.

Function abs(i)
ceil(d)

cos(d) exp(d) fabs(d) floor(d)

getchar()
log(d)
pow(d1,d2) putchar(c) rand()
sin(d)
sqrt(d)

tan(d)
toascii(c)
tolower(c) toupper(c)

Purpose
Return the absolute value of (i is integer)

Round up to the next integer value(the smallest integer that is greater than or equal to d)

Return the cosine of d
Raise e to the power d(e=Naperian constant)
Return the absolute value of d(d is double)

Round down to the next integer value(the largest integer that does not exceed d)
Enter a character from the standard input device

Return the natural logarithm of d
Return d1 raised to the power d2
Send a character to the standard output device
Return a random positive integer
Return sine of d
Return the square root of d
Return the tangent of d
Convert value of argument to ASCII
Convert letter to lowercase
Convert letter to uppercase
Table 2.1

Program 2.2: Program to convert lowercase to uppercase

#include <stdio.h> /* Input/Output functions are available in stdio.h */ #include<ctype.h> /* Character functions are available in the file ctype.h */ main()
/* read a lowercase character and print its uppercase equivalent */ {
int lower, upper;
lower=getchar();
upper=toupper(lower);
putchar(upper);

}

Program 2.3: Program to illustrate the use of library functions

#include<stdio.h>
#include<ctype.h>
#include<math.h> /* Mathematical functions are available in math. h*/ main()
{
int i=-10, e=2, d=10;
float rad=1.57;
double d1=2.0, d2=3.0;
printf("%d\n", abs(i));
printf("%f\n", sin(rad));
printf("%f\n", cos(rad));
printf("%f\n", exp(e));
printf("%d\n", log(d));
printf("%f\n", pow(d1,d2));
}

Execute the above program and observe the result

Self Assessment Questions

5._____ are built-in functions that carry out various
commonly used operations or calculations

6.Fill in the blanks as indicated by alphabets. The value of (a) floor(5.8), (b)floor(-5.8), (c) ceil(5.8) and (d) ceil(-5.8) are _____ (a) _____, ____ (b) _____, ____ (c) _____ and _____ (d) _____ respectively.

2.7 Bitwise Operators

The *bitwise operators* &, |, ^, and ~ operate on integers thought of as binary numbers or strings of bits. The & operator is bitwise AND, the | operator is bitwise OR, the ^operator is bitwise exclusive-OR (XOR), and the ~ operator is a bitwise negation or complement. (&, |, and ^are “binary" in that they take two operands; ~ is unary.) These operators let you work with the individual bits of a variable; one common use is to treat an integer as a set of single-bit *flags*. You might define the 3rd bit as the “verbose" flag bit by defining

#define VERBOSE 4
Then you can “turn the verbose bit on" in an integer variable flags by executing
flags = flags | VERBOSE;
and turn it off with
flags = flags & ~VERBOSE;
and test whether it's set with
if(flags & VERBOSE)

The left-shift and right-shift operators << and >> let you shift an integer left or right by some number of bit positions; for example, value << 2 shifts value left by two bits.

The **comma operator** can be used to link the related expressions together. The expressions are executed one after the other. The most common use for comma operators is when you want multiple variables controlling a for loop, for example:

for(i = 0, j = 10; i < j; i++, j--)

Self Assessment Questions

7._____ let you work with the individual bits of a variable;

one common use is to treat an integer as a set of single-bit flags. 8.Fill in the blanks as indicated by alphabets. If flag1=5, flag2=8, then (a)flag1&flag2, (b) flag1|flag2, (c) ~flag1 and (d) flag1f^lag2 is computed to be _____ (a) _____, ____ (b) _____, ____ (c)_____ and _____ (d) _____ respectively.

2.8 Increment and Decrement Operators

When we want to add or subtract constant 1 to a variable, C provides a set of shortcuts: the *autoincrement* and *autodecrement* operators. In their simplest forms, they look like this:

++i *add 1 to* i

--j *subtract 1 from* j

These correspond to the forms i = i + 1 and j = j - 1. They are also equivalent to the short hand forms i+=1 and j-=1. C has a set of 'shorthand' assignment operators of the form:

v op=exp;
where v is a variable, exp is an expression and op is a C binary arithmetic operator.

The assignment statement
v op=exp;
is equivalent to
v= v op(exp);
Example:
x+=y+1;
This is same as the statement
x=x+(y+1);
The ++ and -- operators apply to one operand (they're *unary* operators). The expression ++i adds 1 to i, and stores the incremented result back in i. This means that these operators don't just compute new values; they also modify the value of some variable. (They share this property -- modifying some variable -- with the assignment operators; we can say that these operators all have *side effects*. That is, they have some effect, on the side, other than just computing a new value.)

The incremented (or decremented) result is also made available to the rest of the expression, so an expression like
k = 2 * ++i

means “add one to i, store the result back in i, multiply it by 2, and store *that* result in k." (This is a pretty meaningless expression; our actual uses of ++ later will make more sense.)

Both the ++ and -- operators have an unusual property: they can be used in two ways, depending on whether they are written to the left or the right of the variable they're operating on. In either case, they increment or decrement

the variable they're operating on; the difference concerns whether it's the old or the new value that's "returned" to the surrounding expression. The *prefix* form ++i increments i and returns the incremented value. The *postfix* form i++ increments i, but returns the *prior*, nonincremented value. Rewriting our previous example slightly, the expression k = 2 * i++

means "take i's old value and multiply it by 2, increment i, store the result of the multiplication in k."
The distinction between the prefix and postfix forms of ++ and -- will probably seem strained at first, but it will make more sense once we begin using these operators in more realistic situations.
For example,

```
a[i] = c;
i = i + 1;
```

using the ++ operator, we could simply write this as

```
a[i++] = c;
```

We wanted to increment i *after* deciding which element of the array to store into, so the postfix form i++ is appropriate.
Notice that it only makes sense to apply the ++ and -- operators to variables (or to other "containers," such as a[i]). It would be meaningless to say something like

```
1++
```

or

```
(2+3)++
```

The ++ operator doesn't just mean "add one"; it means "add one *to a variable*" or "make a variable's value one more than it was before." But (1+2) is not a variable, it's an expression; so there's no place for ++ to store the incremented result.
Another unfortunate example is

```
i = i++;
```

which some confused programmers sometimes write, presumably because they want to be extra sure that i is

incremented by 1. But i++ all by itself is sufficient to increment i by 1; the extra (explicit) assignment to i is unnecessary and in fact counterproductive, meaningless, and incorrect. If you want to increment i (that is, add one to it, and store the result back in i), either use

```
i = i + 1;
```

or

```
i += 1;
```

or

```
++i;
```

or

```
i++;
```

Did it matter whether we used ++i or i++ in this last example? Remember, the difference between the two forms is what value (either the old or the new) is passed on to the surrounding expression. If there is no surrounding expression, if the ++i or i++ appears all by itself, to increment i and do nothing else, you can use either form; it makes no difference. (Two ways that an expression can appear "all by itself," with "no surrounding expression," are when it is an expression statement terminated by a semicolon, as above, or when it is one of the controlling expressions of a for loop.) For example, both the loops

```
for(i = 0; i < 10; ++i)
printf("%d\n", i);
```

and

```
for(i = 0; i < 10; i++)
printf("%d\n", i);
```

will behave exactly the same way and produce exactly the same results. (In real code, postfix increment is probably more common, though prefix definitely has its uses, too.)

Self Assessment Questions

9.Increment and Decrement operators are binary operators. (True/False)

10.The prefix form to increment i is _____ whereas the postfix form to increments i is _____.

2.9 The size of Operator

The **sizeof** is a compile time operator and, when used with an operand, it returns the number of bytes the operand occupies. The operand may be a variable, a constant or a data type qualifier.

Examples:

m=sizeof(sum);

n=sizeof(long int);

k=sizeof(235L);

The size of operator is normally used to determine the lengths of arrays and structures when their sizes are not known to the programmer. It is also used to allocate memory space dynamically to variables during execution of a program.

Program 2.4: Program to illustrate the use of size of operator

```
#include<stdio.h>
main()
{
int i=10;
```

```
printf("integer: %d\n", sizeof(i);
}
```

The above program might generate the following output:

integer: 2

Thus we see that this version of C allocates 2 bytes to each integer quantity. **Program 2.5: Program to illustrate arithmetic operators**

```
#include<stdio.h>
main()
{
int a, b, c, d;
a=10;
b=15;
c=++a-b;
printf("a=%d b=%d c=%d\n", a, b, c);
d=b++ +a;
printf("a=%d b=%d d=%d\n", a, b, d);
printf("a/b=%d\n", a/b);
printf("a%%b=%d\n", a%b);
printf("a*=b=%d\n", a*=b);
printf("%d\n", (c>d)?1:0);
printf("%d\n", (c<d)?1:0);
}
```

Execute the above program and observe the result.

2.10 Precedence of Operators

The precedence of C operators dictates the order of evaluation within an expression. The precedence of the operators introduced here is summarised in the Table 2.2. The highest precedence operators are given first.

Operators Associativity
( ) -> . left to right
!~ + - ++ -- & *right to left
*/ %left to right
+-left to right
< <= > >= left to right
== != left to right
& left to right
| left to right
&& left to right
|| right to left
= *= /= %= += -= right to left

Table 2.2

Where the same operator appears twice (for example *) the first one is the unary version.
Program 2.6: A program to illustrate evaluation of expressions #include<stdio.h>
main()

```
/* Evaluation of expressions */
{
float a, b, c, x, y, z;
a=20;
b=2;
c=-23;
x = a + b / ( 3 + c * 4 - 1);
y = a – b / (3 + c) * ( 4 – 1);
z= a – ( b / ( 3 + c ) * 2 ) – 1;
printf( “x=%f\n”, x);
printf(“y=%f\n”, y);
printf(“z=%f\n”, z);
}
```

Execute the above program and observe the result.

Program 2.7: Program to convert seconds to minutes and seconds

```
#include <stdio.h>
#define SEC_PER_MIN 60 // seconds in a minute
int main(void)
{
int sec, min, left;

printf("Convert seconds to minutes and seconds!\n"); printf("Enter the number of seconds you wish to convert.\n");
scanf("%d", &sec); *number of seconds is read in
```

```
min = sec / SEC_PER_MIN; *truncated number of minutes left = sec % SEC_PER_MIN; *number of seconds left over
printf("%d seconds is %d minutes, %d seconds.\n", sec, min, left);
return 0;
}
```

2.11 Summary

C supports a rich set of operators. An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations. Operators are used in programs to manipulate data and variables. A binary operator acts on two operands. A unary operator acts upon a single operand to produce a new value. Multiplication, division, and modulus all have higher precedence than addition and subtraction. Relational and Boolean expressions are usually used in contexts such as an if statement, where something is to be done or not done depending on some condition. The C language is accompanied by a number of library functions or built in functions that carry out various commonly used operations or calculations.

The **size of** operator is normally used to determine the lengths of arrays and structures when their sizes are not known to the programmer. It is also used to allocate memory space dynamically to variables during execution of a program. Associativity is the order in which consecutive operations within the same precedence group are carried out.

2.12 Terminal questions

1.If i=10 and j=12, what are the values of c and d after executing the following program segment:

```
i++;
c=j++ + i;
```

d=++i + j++;

2.Suppose that x, y and z are integer variables which have been assigned

the values 2, 3 and 4, respectively. Evaluate the following expression and determine the value of x.

x *= -2 * (y + z) / 3

3.Suppose that i is an integer variable whose value is 7 and c is a character variable that represents the character 'w', evaluate the
following logical expression:

(i>=6) && (c=='w')

4.Suppose that i is an integer variable whose value is 7 and f is a floating
–point variable whose value is 8.5. Evaluate the following expression:

(i + f) %4

5.What is meant by associativity?

2.13 Answers to Self Assessment Questions

1.3

2.% (modulus)

3.&&, ||

4. Incorrect

5.Library functions

6.(a) 5, (b) -6, (c) 6, (d) -5

7.Bitwise operator

8.8. (a) 0, (b) 13 , (c) 10 , (d) 13

9. False

10.++i, i++

2.14 Answers to Terminal Questions

1 c=23 and

. d=25 -8

2

Page No.: 34

3. true

4.Given expression is invalid because a floating point variable can not be used in a modulus operation.

5.Associativity is the order in which consecutive operations within the same precedence group are carried out.

2.15 Exercises

1.Suppose a=3, b=5, c=8 and d=4, give the output of the following: a) x=a*b-c/d+4 b) z=a-b/d*c+10

2.Suppose i=5, j=8, k=10, then , what is the output of the following: a)x=a++ -jb)y=k++ *j—

3.What is the precedence of operators? How expressions are evaluated using the precedences?

4.Suppose a=7, b=11, find the output of the following:

a) x=(a>b)?b:a b) x=(a<b)?a:b

5.Explain the use of bitwise operators with suitable examples.

Unit 3 Data Types and Input/Output Operators

Structure:

3.1 Introduction

Objectives

3.2 Floating-point Numbers

Converting Integers to Floating-point and vice-versa Mixed-mode Expressions

3.3 The type cast Operator

3.4 The type **char**

3.5 Keywords

3.6 Character Input and Output

3.7 Formatted input and output

3.8 The **gets()** and **puts()** functions

3.9 Interactive Programming

3.10 Summary

3.11 Terminal Questions

3.12 Answers to Self Assessment Questions

3.13 Answers to Terminal Questions

3.14 Exercises

3.1 Introduction

In the previous unit, you learned about the operators that a C programming language supports. You also learned how the operators are used in the expressions of C programs. In this unit, you will learn about the data types that are supported in C. You will also study about the Input/output operators which makes C as the most efficient and powerful programming language. Integer is one of the fundamental data types. All C compilers support four fundamental data types, namely integer (**int**), character (**char**), floating point (**float**), and double-precision floating point (**double**). Like integer data type, other data types also offer extended data types such as **long double** and
signed char.

C supports a rich set of operators. We have already used several of them, such as =, +, -, *, / and %. An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations. Operators are used in programs to manipulate data and variables. They usually form a part of the mathematical or logical expressions.Page No.: 36 It is possible to combine operands of different data types in arithmetic expressions. Such expressions are called mixed-mode arithmetic expressions.

C language is accompanied by some library functions to handle input/output(I/O) operations. In this unit we will make use of six I/O functions : getchar(), putchar(), scanf(), printf(), gets() and puts(). These functions are used to transfer the information between the computer and the standard input/output devices. Throughout this course we assume that keyboard is the standard input device and the user screen is the standard output device. The first two functions, getchar() and putchar(), allow single character to be transferred into and out of the computer; the functions scanf() and printf() permit the transfer of single character, numerical values and strings; the functions gets() and puts() facilitate the input and output of strings. These functions can be accessed within a program by including the header file stdio.h.

Objectives:
After studying this unit, you should be able to:

apply the concept of real numbers in C and characters in C
combine different data types and form more complicated arithmetic expressions

to transfer a character, numerical value and a string between the computer and I/O devices
to write programs using I/O functions to handle single character, numerical values and strings

3.2 Floating-point Numbers

Floating point (or real) numbers are stored in 32 bit (on all 16 bit and 32 bit machines), with 6 digits of precision. Floating point numbers are defined in C by the keyword **float.** When the accuracy provided by a **float** number is not sufficient, the type **double** can be used to define the number. A **double** data type number uses 64 bits giving a precision of 14 digits. These are known as double precision numbers. To extend the precision further, we may use **long double** which uses 80 bits. The table 3.1 shows all the allowed combinations of floating point numbers and qualifiers and their size and range on a 16-bit machine.

Table 3.1: Floating point numbers and qualifiers and their size and range on a 16-bit machine.

Type Size (bits) Range
Float 32 3.4E-38 to 3.4E+38
Double 64 1.7E-308 to 1.7E+308
long double 80 3.4E-4932 to 1.1E+4932

Program 3.1: The following program illustrates typical declarations, assignments and values stored in various types of variables.

```
main()
{
/* .....DECLARATIONS.....*/
float x, p;
double y, q;
unsigned k;
/* .....DECLARATIONS AND ASSIGNMENTS.....*/
int m=54321;
long int n=1234567890;
/*.....ASSIGNMENTS.....*/
x = 1.234567890000;
y = 9.87654321;
k = 54321;
p=q=1.0;
/*.....PRINTING.....*/
printf("m=%d\n",m);
printf("n=%ld\n",n);
printf("x=%.12lf\n",x);
printf("x=%f\n",x);
printf("y=%.12lf\n",y);
printf("y=%lf\n",y);
```

```
printf("k=%u p= %f q=%.12lf\n",k,p,q);
}
```

Output

```
m = -11215
n = 1234567890
x = 1.234567880630
x = 1.234568
y = 9.876543210000
y = 9.876543
k = 54321 p = 1.000000 q= 1.000000000000
```

Program 3.2: Program to calculate the average of N numbers

```
#define N 10 /* SYMBOLIC CONSTANT */
main()
{

int count; /* DECLARATION OF float sum, average, number; VARIABLES */ sum = 0; / * INITIALIZATION OF

count = 0; VARIABLES*/
while (count<N)
{
scanf("%f", &number);
sum = sum + number;
count = count + 1;
```

```
}
average = sum / N;
printf(“N = % d Sum = %f “, N, sum);
printf(“Average = %f”, average);
```

Output

```
1
2.3
4.67
1.42
7
3.67
4.08
2.2
4.25
8.21
N= 10 Sum= 38.799999 Average= 3.880000
```

Program 3.3: Program to compute the roots of a quadratic equation

```
#include <math.h>
main()
{
float a,b,c,discriminant, root1, root2;
printf(“input the values of a,b and c\n”);
```

```
scanf (“%f %f %f”, &a, &b, &c);
discriminant = b * b – 4 * a *c;
if (discriminant<0)
printf(“roots are imaginary\n”);
else
{
root1 = (-b + sqrt(discriminant)) / (2 * a);
root2 = (-b - sqrt(discriminant)) / (2 * a);
printf (“Root1 = %5.2f \n Root2 = %5.2f \n”, root1, root2);
}
}
```

Output

```
input the values of a,b and c
2 4 -16
Root1 = 2.00
Root2 = -4.00
input the values of a,b and c
1 2 3
roots are imaginary
```

3.2.1 Converting Integers to Floating-point and vice-versa

C permits mixing of constants and variables of different types in an expression, but during evaluation it adheres to very strict rules of type conversion. We know that the computer considers one operator at a time, involving two operands.

If the operands are of different types, the 'lower' type is automatically converted to the 'higher' type before the operation proceeds. The result is of higher type.
Given below is the sequence of rules that are applied while evaluating expressions.

All **short** type are automatically converted to **int ;** then
1.If one of the operands is **long double,** the other will be converted to **long double** and the result will be **long double**;
2.else, if one of the operands is **double**, the other will be converted to **double** and the result will be **double**;
3.else, if one of the operands is **float**, the other will be converted to **float** and the result will be **float**;
4.else, if one of the operands is **unsigned long int**, the other will be

converted to **unsigned long int** and the result will be **unsigned long int**;
5.else if one of the operands is **long int** and the other is **unsigned int**, then:

if **unsigned int** can be converted to **long int**, the **unsigned int** operand will be converted as such and the result will be **long int**; else, both operands will be converted to **unsigned long int** and the

result will be **unsigned long int**;
6.else, if one of the operands is **long int** , the other will be converted to **long int** and the result will be **long int**;
7.else, if one of the operands is **unsigned int** , the other will be converted to **unsigned int** and the result will be **unsigned int**;
The final result of an expression is converted to type of the variable on the left of the assignment sign before assigning the value to it. However, the following changes are introduced during the final assignment:
1.**float** to **int** causes truncation of the fractional part.
2.**double** to **float** causes rounding of digits.
3.**long int** to **int** causes dropping of the excess higher order bits **3.2.2 Mixed-mode Expressions**

When one of the operands is real and the other is integer, the expression is called a mixed-mode arithmetic expression. If either operand is of the real type, then only the real operation is performed and the result is always real number. Thus
25 / 10.0 = 2.5
Whereas 25 / 10 =2
1.When the accuracy provided by a **float** number is not sufficient, the

type **long float** can be used to define the number. (True/False) 2.A **double** data type uses _____ bits.
3.If the operands are of different data types, the 'lower' type is automatically converted to the 'higher' type before the operation proceeds. (True/False)

4.During the final assignment _____ to int causes dropping of the excess higher order bits.
5.The value of the expression 22.0/10 is _____.

3.3 The type cast Operator

C performs type conversions automatically. However, there are instances when we want to force a type conversion in a way that is different from the automatic conversion. Consider, for example, the calculation of ratio of doctors to engineers in a town.
Ratio = doctor_number / engineer _number
Since doctor _number and engineer_number are declared as integers in the program, the decimal part of the result of the division would be lost and Ratio would represent a wrong figure. This problem can be solved by converting locally one of the variables to the floating point as shown below: Ratio = (**float**) doctor_number / engineer _number
The operator (**float**) converts the doctor_number to floating point for the purpose of evaluation of the expression. Then using the rule of automatic conversion, the division is performed in floating point mode, thus retaining the fractional

part of the result. Note that in no way does the operator (**float**) affect the value of the variable doctor_number. And also, the type of doctor_number remains as **int** in the other parts of the program.

The process of such local conversion is known as casting a value. The general form of cast is:
(type-name) expression
where type-name is one of the standard C data types. The expression may be a constant, variable or an expression. The Table 3.2 shows some examples of casts and their actions:

Table 3.2: Use of Casts

Example Action

X=(**int**) 8.5 8.5 is converted to integer by truncation. A=(**int**) **21.3** / (**int**) 4.5 Evaluated as 21/4 and the result would be 5. B=(**double**) sum/n Division is done in floating point mode. Y= (**int**) (a+b) The result of a+b is converted to integer. Z= (**int**) a+b a is converted to integer and then added to b. P=cos((**double**)x) Converts x to double before using it.

Program 3.4: The following program shows the use of casts

```
main()
{
/* Program to find average of two integers */
float avg;
int n=2,n1,n2;
printf("enter any 2 numbers\n");
scanf("%d %d",&n1,&n2);
avg=(n1+n2)/(float)n;
printf(" their average is\n",avg);
}
```

Casting can be used to round-off a given value. Consider the following statement:

X= (**int**) (y+0.5);

If y is 37.7, y+0.5 is 38.2 and on casting, the result becomes 38, the value that is assigned to X. Of course, the expression, being cast is not changed. When combining two different types of variables in an expression, never assume the rules of automatic conversion. It is always a good practice to explicitly force the conversion. It is more safer and more portable. For example, when **y** and **p** are double and **m** is **int**, the following two statements are equivalent.

y = p + m;

y = p + (double)m;

However, the second statement is preferable. It will work the same way on all machines and is more readable.

6.Casting can be used to round-off a given value. (True/False)

7.The value of A in the expression A=(int)11.35/(int)14.5 is _____. 8.If the value of X is 35.2, then the value of A in the expression: A = (int)(X+0.5); is _____.

3.4 The type char

A single character can be defined as a character(**char**) type data. Characters are usually stored in 8 bits (one byte) of internal storage. The qualifier **signed** or **unsigned** may be explicitly applied to **char.** While **unsigned char**s have values between 0 and 255, **signed char**s have values from -128 to 127.

A character constant is formed by enclosing the character within a pair of single quote marks. So 'b', '.' and '5' are all valid examples of character constants. Note that a character constant, which is a single character enclosed in single quotes is different from a character string, which is any number of characters enclosed in double quotes.

The format characters %c can be used in a printf statement to display the value of a char variable at the terminal.

Program 3.5: The following program illustrates how to use char data type.

```
#include<stdio.h>
main()
{
char c='A';
int a=65;
printf("%c\n", c);
printf("%d\n", c);
printf("%c\n",a);
}
```

Output

```
A
65
A
```

Note that with the format characters %d, the ASCII number of the character is displayed. With the format character %c, the character corresponding to the given ASCII number is displayed.

Self Assessment Questions

9.What is the format character to display the value of a char variable?

10.The output of the following C statement:

printf("%c", 70); is _____.

3.5 Keywords

Keywords are the reserved words of a programming language. All the keywords have fixed meanings and these meanings cannot be changed. Keywords serve as basic building blocks for program statements. The list of all keywords in ANSI C are listed in the Table 3.3.

auto
break
case
char
const
continue default
do

Table 3.3: ANSI C Keywords

double int
else long
enum register
extern return
float short
for signed
goto sizeof
if static
struct
switch
typedef union

unsigned void
volatile
while

All keywords must be written in lowercase. Some compilers may use additional keywords that must be identified from the C manual.

Self Assessment Questions

11.All keywords must be written in _____.

12.**default** is not a valid keyword in C. (True/False)

3.6 Character Input and Output

The most basic way of reading input is by calling the function **getchar(). getchar()** reads one character from the "standard input," which is usually the user's keyboard. **getchar()** returns (rather obviously) the character it reads, or, if there are no more characters available, the special value EOF ("end of file"). This value will be assigned within the stdio.h file. Typically, EOF will be assigned the value -1, but this may vary from one compiler to another.

The syntax of the **getchar()** function is written as
character variable= getchar()

where *character variable* refers to some previously declared character variable.
Example:

```
char c;
...
```

c=getchar();
The first statement declares that c is a character-type variable. The second statement causes a single character to be entered from the keyboard and then assign to c.

A companion function is **putchar(),** which writes one character to the “standard output." (The standard output is usually the user's screen).

The syntax of the **putchar()** function is written as
putchar(character variable)

where *character variable* refers to some previously declared character variable.
Example:
char c;
…
putchar(c);
The first statement declares that c is a character-type variable. The second statement causes the current value of c to be transmitted to the user monitor where it will be displayed.

Using these two functions, we can write a very basic program to copy the input, a character at a time, to the output:

Program 3.6: Program to copy the input, a character at a time, to the output

```
#include <stdio.h>
/* copy input to output */
main()
{
```

```
int c;
c = getchar();
while(c != EOF)
{
putchar(c);
c = getchar();
}
return 0;
}
```

Execute the above program and observe the result.

It reads one character, and if it is not the EOF code, enters a while loop, printing one character and reading another, as long as the character read is not EOF. A **char** variable could hold integers corresponding to character set values, and that an **int** could hold integers of more arbitrary values (from 32768 to + 32767). Since most character sets contain a few hundred characters (nowhere near 32767), an **int** variable can in general comfortably hold all **char** values, and then some. Therefore, there's nothing wrong with declaring c as an **int.** But in fact, it's important to do so, because **getchar()** can return every character value, *plus* that special, non-character value EOF, indicating that there are no more characters. Type **char** is only guaranteed to be able to hold all the character values; it is *not* guaranteed to be able to hold EOF value without possibly mixing it up with some actual character value. Therefore, you should always remember to use an **int** for anything you assign **getchar()**'s return value to.

When you run the character copying program, and it begins copying its input (you're typing) to its output (your screen), you may find yourself wondering how to stop it. It stops when it receives end-of-file (EOF), but how do you send EOF? The answer depends on what kind of computer you're using. On Unix and Unix-related systems, it's almost always control-D. On MS-DOS machines, it's control-Z followed by the RETURN key.

(Note, too, that the character you type to generate an end-of-file condition from the keyboard is *not* the same as the special EOF value returned by **getchar().** The EOF value returned by **getchar()** is a code indicating that the input system has detected an end-of-file condition, whether it's reading the keyboard or a file or a magnetic tape or a network connection or anything else. In a disk file, at least, there is not likely to be any character in the file corresponding to EOF; as far as your program is concerned, EOF indicates the absence of any more characters to read.)
Another excellent thing to know when doing any kind of programming is how to terminate a runaway program. If a program is running forever waiting for input, you can usually stop it by sending it an end-of-file, as above, but if it's running forever *not* waiting for something, you'll have to take more drastic measures. Under Unix, control-C (or, occasionally, the DELETE key) will terminate the current program, almost no matter what. Under MS-DOS, control-C or control-BREAK will sometimes terminate the current program.

Self Assessment Questions

13.getchar() function is an output function.(True/False)

14.In order to stop reading the input character, you can use a value called _____.

3.7 Formatted Input and Output

Input data can be entered into the computer from a standard input device by means of the standard C library function **scanf()**. This function can be used to enter any combination of numerical values, single character and strings. The function returns the number of data items that have been entered successfully.

The syntax of scanf function is as follows:
scanf(control string, arg1, arg2, ...argn)

where *control string* refers to a string containing certain required formatting information, and *arg1, arg2,..., argn* are arguments that represent the individual input data items. The arguments represent *pointers* that indicate *addresses* of the data items within the computer's memory.
The *control string* consists of control characters, whitespace characters, and non-whitespace characters. The control characters are preceded by a % sign, and are listed in Table 3.4.

Table 3.4: Control characters and its explanations
Control Character Explanation

%c a single character
%d a decimal integer
%i an integer
%e, %f, %g a floating-point number
%o an octal number
%s a string
%x a hexadecimal number
%p a pointer

%n an integer equal to the number of characters read so far %u an unsigned integer
%[] a set of characters
%% a percent sign

scanf() reads the input, matching the characters from format. When a control character is read, it puts the value in the next variable. Whitespaces (tabs, spaces, etc) are skipped. Non-whitespace characters are matched to the input, then discarded. If a number comes between the % sign and the control character, then only that many characters will be entered into the variable. If **scanf()** encounters a set of characters, denoted by the %[] control character, then any characters found within the brackets are read

into the variable. The return value of **scanf()** is the number of variables that were successfully assigned values, or EOF if there is an error.

Program 3.7: Program to use scanf() to read integers, floats, characters and strings from the user.

```
#include<stdio.h>
main()
{
int i;
float f;
char c;
char str[10];
scanf("%d %f %c %s", &i, &f, &c, str);
printf("%d %f %c %s", i, f, c, str);
}
```

Execute this program and observe the result.

Note that for a **scanf()** function, the addresses of the variable are used as the arguments for an **int, float** and a **char** type variable. But this is not true for a string variable because a string name itself refers to the address of a string variable.

A *s-control* character is used to enter strings to string variables. A string that includes whitespace characters can not be entered. There are ways to work with strings that include whitespace characters. One way is to use the **getchar()** function within a loop. Another way is to use **gets()** function which will be discussed later.

It is also possible to use the **scanf()** function to enter such strings. To do so, the s-control character must be replaced by a sequence of characters enclosed in square brackets, designated as [...]. Whitespace characters may be included within the brackets, thus accommodating strings that contain such characters.

Example:

```
#include<stdio.h>
main()
{
char str[80];
...
scanf("%[ ABCDEFGHIJKLMNOPQRST]", str);
...
}
```

This example illustrates the use of the scanf() function to enter a string consisting of uppercase letters and blank spaces. Please note that if you want to allow lowercase letters to be entered, all the lowercase letters (i.e. from az) must be included in the list of control string.

Formatted Output

Output data can be written from the computer onto a standard output device using the library function **printf().** This function can be used to output any combination of numerical values, single characters and strings. It is similar to the input function **scanf(),** except that its purpose is to display data rather than enter into the computer.

The syntax of the **printf** function can be written as follows:

printf(control string, arg1, arg2, ..., argn)

where *control string* refers to a string that contains formatting information, and arg1, arg2, ..., argn are arguments that represent the individual output data items. The arguments can be written as constants, single variable or array names, or more complex expressions.
Examples:
printf("Hello, world!\n");
printf("i is %d\n", i);
printf("%d", 10);
printf("%d", i+j);

The first statement simply displays the string given as argument to the **printf()** function. In the second statement, **printf()** function replaces the two characters %d with the value of the variable i. In the third statement the argument to be printed is a constant and in the fourth, the argument is an expression.

There are quite a number of format specifiers for printf(). Some of them are listed in Table 3.5.

Table 3.5: Format specifiers for printf().

%d Print an int argument in decimal
%ld print a long int argument in decimal
%c print a character
%s print a string
%f print a float or double argument
%e same as %f, but use exponential notation %g use %e or %f, whichever is better
%o print an int argument in octal (base 8)
%x print an int argument in hexadecimal (base 16) %% print a single %

It is also possible to specify the width and precision of numbers and strings as they are inserted ; For example, a notation like %3d means to print an **int** in a field at least 3 spaces wide; a notation like %5.2f means to print a **float** or

double in a field at least 5 spaces wide, with two places to the right of the decimal.)
To illustrate with a few more examples: the call

printf("%c %d %f %e %s %d%%\n", '3', 4, 3.24, 66000000, "nine", 8); would print
3 4 3.240000 6.600000e+07 nine 8%
The call
printf("%d %o %x\n", 100, 100, 100);
would print
100 144 64
Successive calls to **printf()** just build up the output a piece at a time, so the calls
printf("Hello, ");
printf("world!\n");
would also print Hello, world! (on one line of output).

Earlier we learned that C represents characters internally as small integers corresponding to the characters' values in the machine's character set (typically ASCII). This means that there isn't really much difference between a character and an integer in C; most of the difference is in whether we choose to interpret an integer as an integer or a character. **printf** is one place where we get to make that choice: %d prints an integer value as a string of digits representing its decimal value, while %c prints the character corresponding to a character set value. So the lines
char c = 'A';
int i = 97;
printf("c = %c, i = %d\n", c, i);

would print c as the character A and i as the number 97. But if, on the other hand, we called
printf("c = %d, i = %c\n", c, i);

we'd see the decimal value (printed by %d) of the character 'A', followed by the character (whatever it is) which happens to have the decimal value 97.

You have to be careful when calling **printf().** It has no way of knowing how many arguments you've passed it or what their types are other than by looking for the format specifiers in the format string. If there are more format specifiers (that is, more % signs) than the arguments, or if the arguments have the wrong types for the format specifiers, **printf()** can misbehave badly, often printing nonsense numbers or (even worse) numbers which mislead you into thinking that some other part of your program is broken.

Because of some automatic conversion rules which we haven't covered yet, you have a small amount of latitude in the types of the expressions you pass as arguments to **printf().** The argument for %c may be of type **char** or **int,** and the argument for %d may be of type **char** or **int.** The string argument for %s may be a string constant, an array of characters, or a pointer to some characters. Finally, the arguments corresponding to %e, %f, and %g may be of types **float** or **double.** But other combinations do *not* work reliably: %d will not print a **long int** or a **float** or a **double;** %ld will not print an **int;** %e, %f, and %g will not print an **int.**

Self Assessment Questions

15.The _____ string consists of control characters, whitespace characters, and non-whitespace characters.

16.The control string used to read a hexadecimal character is

_____.

17.scanf() functions needs address of the data item to be read as the argument. (True/False)

18. The output of the following statement is _____. printf("%d %o %x\n", 64, 10, 75);

19. To print an int argument in octal, you can use _____ format string.

20. The output of the following program segment is _____. int a=97;
printf("%c", a);

3.8 The gets() and puts() functions

gets() and **puts()** functions facilitate the transfer of strings between the computer and the standard input/output devices. Each of these functions accepts a single argument. The argument must be a data item that represents a string (an array of characters). The string may include whitespace characters. In the case of **gets()**, the string will be entered from the keyboard, and will terminate with a newline character (i.e. a string will end when the user presses the RETURN key).

Example: Reading and writing a line of text.

```
#include<stdio.h>
main()
{
char line[80];
gets(line);
puts(line);
}
```

This program uses **gets()** and **puts()** functions rather than **scanf()** and **printf(),** to transfer the line of text into and out of the computer. **Self Assessment Questions**
21.**gets()** is a formatted input statement. (True/False)

22.The argument for a **gets()** and **puts()** functions are _____ variables.

23.Using **gets()** function, you cannot include whitespace characters in the input string. (True/False)

3.9 Interactive Programming

Creating interactive dialog between the computer and the user is a modern style of programming. These dialogs usually involve some form of questionanswer interaction, where the computer asks the questions and the user provides the answer, or vice versa.

In C, such dialogs can be created by alternate use of the **scanf()** and **printf()** functions.

Program 3.8: Program to calculate the simple interest

```
#include<stdio.h>
main()
{
/* Sample interactive program*/
float principle, rate, time, interest;
printf(“ Please enter the principle amount: “);
scanf(“%f”, &principle);
printf(“ Please enter the rate of interest: “);
scanf(“%f”, &rate);
printf(“ Please enter the period of deposit: “);
scanf(“%f”, &time);
interest=principle*rate*time/100.0;
printf(“Principle=%7.2f\n”, principle);
```

```
printf("Rate of interest=%5.2f\n",rate);
printf("Period of deposit=%5.2f\n", time);
printf("Interest=%7.2f\n", interest);
}
```

Execute the above program and observe the result.

3.10 Summary

Floating point(or real) numbers are stored in 32 bit (on all 16 bit and 32 bit machines), with 6 digits of precision. Floating point numbers are defined in C by the keyword **float.** When the accuracy provided by a **float** number is not sufficient, the type **double** can be used to define the number. Characters are usually stored in 8 bits (one byte) of internal storage. Like integer data type other data types also offer extended data types such as **long double** and **signed char.** C permits mixing of constants and variables of different types in an expression, but during evaluation it adheres to very strict rules of type conversion. When one of the operands is real and the other is integer, the expression is called a mixed-mode arithmetic expression. There are instances when we want to force a type conversion in a way that is different from the automatic conversion. That is, by using type cast operator. All keywords have fixed meanings and these meanings cannot be changed.

getchar(), putchar(), scanf(), printf(), gets() and **puts()** are the commonly used input/output functions in C. These functions are used to transfer of information between the computer and the standard input/output devices. **getchar()** and **putchar()** are the two functions to read and write single character. **scanf()** and **printf()** are the two formatted input/output functions. These functions can handle characters, numerical values and strings as well. **gets()** and **puts()** functions are used to handle strings. **scanf(), printf(), gets()** and **puts()** functions are used in interactive programming.

3.11 Terminal Questions

1.Which of the following arithmetic expressions are valid? If valid, give the value of the expression; otherwise give reason.

a)7.5 % 3b)14 % 3 + 7 %2

c)21 % (int) 4.5d)15.25 + - 5.0

2.Find errors, if any, in the following declaration statements:

Int x;

float letter, DIGIT;

double = p, q

exponent alpha, beta;

m,n,z:INTEGER

short char c;

long int m; count;

long float temp;

3.What would be the value of x after execution of the following statements?

int x, y = 10;

char z = 'a';

x = y + z;

4.The _____ **chars** have values from -128 to 127.

5.What are the commonly used input/output functions in C? How are they accessed?

6.Distinguish between **getchar()** and **putchar()** functions?

7.When entering a string using **scanf()** function, how can a single string which includes whitespace characters be entered?

8.Distinguish between **gets()** and **scanf()** functions.

9.A C program contains the following statements:

```
#include<stdio.h>
int i, j, k;
```

Write an appropriate **scanf()** function to enter numerical values for i, j and k assuming

a)The values for i, j and k will be decimal integers

b)The value for i will be a decimal integer, j an octal integer and k a hexadecimal integer.

c)The values for i and j will be hexadecimal integers and k will be an octal integer.

3.12 Answers to Self Assessment Questions

1. False

2.64

3. True

4.long int

5. 2.2

6. True

7.0

8.35

9.%c

10. False

11. Lowercase

12. False

13. False

14. EOF

15. Control

16. %x

17. True

18.64, 12, 4B

19. %o

20.a

21. False

22. String

23. False

3.13 Answers to Terminal Questions

1. a) invalid, because % can be used only with integers.
b)valid, answer is 3

c)valid, answer is 1

d)valid, answer is 10.25

2. Errors in the following statements

i)Int x;

Can be written as

int x;

ii)double = p, q

Can be written as

double p,q;

iii)exponent alpha, beta;

There is no **data type** exponent in C.

iv) m,n,z:INTEGER

Can be written as

int m,n,z;

v)short char c;

There is no data type **short char** in C.

vi)long int m; count;

Can be written as

long int m,count;

vii)long float temp;

There is no data type **long float** in C

1073 signed

4

· Page No.: 58

5.The commonly used input/output functions in C are: **getchar(), putchar(), scanf(), printf(), gets()** and **puts().** These functions can be accessed within a program by including the header file stdio.h. 6.**getchar()** function is used to accept a single character from the

keyboard and **putchar()** function is used to display single character on the user's screen.
7.By using control string %[ ].
8.gets() is not the formatted input function but the scanf() function is a formatted input function.
9.a) scanf("%d %d %d", &i, &j, &k);
b)scanf("%d %o %x", &i, &j, &k);
c)scanf("%x %x %o", &i, &j, &k);

3.14 Exercises

1.Represent the following numbers using scientific notation:
a) 0.001 b)-1.5

2.Represent the following scientific numbers into decimal notation: a) 1.0E+2 b) 0.001E-2
3.What is unsigned char? Explain.
4.What is short char? Explain.

5.Distinguish between float and double data types.
6.Write a program to print the factors of a given number.

7.Given the length of a side, write a C program to compute surface area and volume of a cube.
8.Write a program to reverse a number and find sum of the digits.

9.Write a program to print the multiplication table for any given number. 10.Write a program to check whether a given number is palindrome.

Unit 4 Control Statements and Decision Making

Structure:

4.1 Introduction

Objectives

4.2 The **goto** statement

4.3 The **if** statement

The **if-else** statement

Nesting of **if** statements

4.4 The conditional expression

4.5 The **switch** statement

4.6 The **while** loop

4.7 The **do...while** loop

4.8 The **for** loop

The nesting of **for** loops
4.9 The **break** statement and **continue** statement
4.10 Summary
4.11 Terminal Questions
4.12 Answers to Self Assessment Questions
4.13 Answers to Terminal Questions
4.14 Exercises

4.1 Introduction

In the previous unit, you studied about the data types that are supported in C and the types of Input/output operators available in C. This unit will enable you to use the various types of control statements for making a decision in C.Statements are the “steps" of a program. Most statements compute and assign values or call functions, but we will eventually meet several other kinds of statements as well. By default, statements are executed in sequence, one after another. We can, however, modify that sequence by using *control flow constructs* such that a statement or group of statements is executed only if some condition is true or false. This involves a kind of decision making to see whether a particular condition has occurred or not and then direct the computer to execute certain statements accordingly.
C language possesses such decision making capabilities and supports the following statements known as the *control* or *decision making* statements.

if statement
switch statement
goto statement
conditional operator statement

You will also study about the loops in this unit. Loops generally consist of two parts: one or more control expressions which control the execution of the loop, and the body, which is the statement or set of statements which is executed over and over.

As far as C is concerned, a true/false condition can be represented as an integer. (An integer can represent many values; here we care about only two values: “true" and “false." The study of mathematics involving only two values is called Boolean algebra, after George Boole, a mathematician who refined this study.) In C, “false" is represented by a value of 0 (zero), and “true" is represented by any value that is nonzero. Since there are many nonzero values (at least 65,534, for values of type int), when we have to pick a specific value for “true," we'll pick 1.

Do…while loop is used in a situation where we need to execute the body of the loop before the test is performed. The for loop is used to execute the body of the loop for a specified number of times. The break statement is used to exit any loop.

Objectives:

After studying this unit, you should be able to:

control the flow of execution of statements using two-way decision and multipath decision.

branch unconditionally from one point to another in the program.

evaluate the conditional expressions.

repeat the execution of statements by checking the condition before the loop body is executed and by checking the condition at the end of the loop.

exit from the loop depending on some condition.

break the current iteration and continue with next iteration of loop.

4.2 The *goto* statement

C supports the **goto** statement to branch unconditionally from one point to another in the program. Although it may

not be essential to use the **goto** statement in a highly structured language like C, there may be occasions when the use of **goto** might be desirable.

The **goto** requires a *label* in order to identify the place where the branch is to be made. A *label* is any valid variable name, and must be followed by a colon. The *label* is placed immediately before the statement where the control is to be transferred. The general forms of **goto** and *label* statements are shown below:

goto *label*; *label:*
---------- statement;
---------- ----------
---------- ----------
label: ----------
statement; **goto** *label;*

(i)Forward jump(ii)Backward jump

The *label* can be anywhere in the program either before the **goto** or after the **goto** *label;* statement.
During execution of the program when a statement like

goto first;

is met, the flow of control will jump to the statement immediately following the label **first.** This happens unconditionally.
Note that a **goto** breaks the normal sequential execution of the program. If the *label* is before the statement **goto** *label;* a loop will be formed and some statements will be executed repeatedly. Such a jump is known as a *backward jump.* On

the other hand, if the *label* is placed after the **goto** *label;* some statements will be skipped and the jump is known as the *forward jump.*

A **goto** is often used at the end of a program to direct the control to go to the input statement, to read further data. Consider the following example:

Program 4.1: Program showing unconditional branching

```
main()

{
double a, b;
read:
printf("enter the value of a\n");
scanf("%f", &a);
if (a<0) goto read;
b=sqrt(a);
printf("%f %f \n",a, b);
goto read;
}
```

This program is written to evaluate the square root of a series of numbers read from the terminal. The program uses two **goto** statements, one at the end, after printing the results to transfer the control back to the input statements and the other to skip any further computation when the number is negative.

Due to the unconditional **goto** statement at the end, the control is always transferred back to the input statement. In fact, this program puts the computer in a permanent loop known as an *infinite* loop.

Self Assessment Questions

1.The goto requires a _____ in order to identify the place where the branch is to be made.

2.*goto* is an unconditional branching statement. (True/False)

4.3 The *if* statement

The simplest way to modify the control flow of a program is with an if statement, which in its simplest form looks like this:

```
if(x > max)
max = x;
```

Even if you didn't know any C, it would probably be pretty obvious that what happens here is that if x is greater than max, x gets assigned to max. (We'd use code like this to keep track of the maximum value of x we'd seen--for each new x, we'd compare it to the old maximum value max, and if the new value was greater, we'd update max.)

More generally, we can say that the syntax of an if statement is: if(*expression*)
statement

where *expression* is any expression and *statement* is any statement.

What if you have a series of statements, all of which should be executed together or not at all depending on whether some condition is true? The answer is that you enclose them in braces:

if(*expression*)

{

statement 1;

statement 2;

statement n;

}

As a general rule, anywhere the syntax of C calls for a statement, you may write a series of statements enclosed by braces. (You do not need to, and should not, put a semicolon after the closing brace, because the series of statements enclosed by braces is not itself a simple expression statement.) **Program 4.2: Program to calculate the absolute value of an integer** # include < stdio.h >

void main ( )

{

int number;

printf (“Type a number:”);

scanf (“%d”, & number);

if (number < 0) /* check whether the number is a negative number */ number = - number; /* If it is negative then convert it into positive. */ printf (“The absolute value is % d \n”, number);

}

4.3.1 The *if-else* statement

An **if** statement may also optionally contain a second statement, the **“else** clause," which is to be executed if the condition is not met. Here is an example:

```
if(n > 0)
average = sum / n;
else {
printf("can't compute average\n");
average = 0;
}
```

The first statement or block of statements is executed if the condition *is* true, and the second statement or block of statements (following the keyword else) is executed if the condition is *not* true. In this example, we can compute a meaningful average only if n is greater than 0; otherwise, we print a message saying that we cannot compute the average. The general syntax of an **if** statement is therefore

if(*expression*)
statement(s)
else
statement(s)

(if there are more than one statements, they should be enclosed within braces).

Program 4.3: To find whether a number is negative or positive #include < stdio.h >

```
void main ( )
{
int num;
printf (“Enter the number”);
scanf (“%d”, &num);
if (num < 0)
printf (“The number is negative”)
```

```
else
printf (“The number is positive”);
}
```

4.3.2 Nesting of *if* statements

It's also possible to nest one **if** statement inside another. (For that matter, it's in general possible to nest any kind of statement or control flow construct within another.) For example, here is a little piece of code which decides roughly which quadrant of the compass you're walking into, based on an x value which is positive if you're walking east, and a y value which is positive if you're walking north:

```
if(x > 0)
{
if(y > 0)
printf("Northeast.\n");
else printf("Southeast.\n");
}
else {
if(y > 0)
printf("Northwest.\n");
else printf("Southwest.\n");
}
```

When you have one **if** statement (or loop) nested inside another, it's a very good idea to use explicit braces {}, as shown, to make it clear (both to you and to the compiler) how they're nested and which **else** goes with which **if**. It's also a good idea to indent the various levels, also as shown, to make the code more readable to humans. Why do both? You

use indentation to make the code visually more readable to yourself and other humans, but the compiler doesn't pay attention to the indentation (since all whitespace is essentially equivalent and is essentially ignored). Therefore, you also have to make sure that the punctuation is right.

Here is an example of another common arrangement of **if** and **else.** Suppose we have a variable **grade** containing a student's numeric grade, and we want to print out the corresponding letter grade. Here is the code that would do the job:

```
if(grade >= 90)
printf("A");
else if(grade >= 80)
printf("B");
else if(grade >= 70)
printf("C");
else if(grade >= 60)
printf("D");
else printf("F");
```

What happens here is that exactly one of the five **printf** calls is executed, depending on which of the conditions is true. Each condition is tested in turn, and if one is true, the corresponding statement is executed, and the rest are skipped. If none of the conditions is true, we fall through to the last one, printing “F".

In the cascaded **if/else/if/else/...** chain, each **else** clause is another **if** statement. This may be more obvious at first if we reformat the example, including every set of braces and indenting each **if** statement relative to the previous one:

```
if(grade >= 90)
{
printf("A");
```

```
}
else {
if(grade >= 80)
{
printf("B");
}
else {
if(grade >= 70)

{ printf("C");
} else {
if(grade >= 60)
{
printf("D");
}
else {
printf("F");
}

}}
```

By examining the code this way, it should be obvious that exactly one of the **printf** calls is executed, and that whenever one of the conditions is found true, the remaining conditions do not need to be checked and none of the later statements within the chain will be executed. But once you've convinced yourself of this and learned to recognize the idiom, it's generally preferable to arrange the statements as in the first example, without trying to indent each successive **if** statement one tabstop further out.

Program 4.4: Program to print the largest of three numbers

```
#include<stdio.h>
main()
{
int a,b,c,big;
printf (“Enter three numbers”);

scanf (“%d %d %d”, &a, &b, &c);
if (a>b) // check whether a is greater than b if true then
if(a>c) // check whether a is greater than c
big = a ; // assign a to big
else big = c ; // assign c to big
else if (b>c) // if the condition (a>b) fails check whether b is greater than c big = b ; // assign b to big
else big = c ; // assign C to big
printf (“Largest of %d,%d&%d = %d”, a,b,c,big);
}
```

Self Assessment Questions:

3.The series of statements enclosed by braces after the expression in simple *if* statement is itself a simple expression statement. (True/False) 4.In the cascaded if/else/if/else/... chain, each else clause is another _____ statement.

4.4 The conditional expression

The conditional operator (?:) takes three operands. It tests the result of the first operand and then evaluates one of the other two operands based on the result of the first. Consider the following example:
E1 ? E2 : E3

If expression *E1* is nonzero (true), then *E2* is evaluated, and that is the value of the conditional expression. If *E1* is 0 (false), *E3* is evaluated, and that is the value of the conditional expression. Conditional expressions associate from right to left. In the following example, the conditional operator is used to get the minimum of x and y:

a = (x < y) ? x : y; /* a= min(x, y) */
There is a sequence point after the first expression (*E1*). The following example's result is predictable, and is not subject to unplanned side effects: i++ > j ? y[i] : x[i];
The conditional operator does not produce a lvalue. Therefore, a statement such as
a ? x : y = 10 is not valid.

4.5 The *switch* statement

The **switch case** statements are a substitute for long **if** statements that compare a variable to several "integral" values ("integral" values are simply values that can be expressed as an integer, such as the value of a **char**). The basic format for using **switch case** is outlined below. The value of the variable given into **switch** is compared to the value following each of the cases, and when one value matches the value of the variable, the computer continues executing the program

from that point.

```
switch ( <variable> ) {
case this-value:
Code to execute if <variable> == this-value
break;
case that-value:
Code to execute if <variable> == that-value
break;
...
default:

Code to execute if <variable> does not equal the value following any of the cases
break;
```

The condition of a **switch** statement is a value. The **case** says that if it has the value of whatever is after that **case** then do whatever follows the colon. The break is used to **break** out of the **case** statements. **break** is a keyword that breaks out of the code block, usually surrounded by braces, which it is in. In this case, **break** prevents the program from falling through and executing the code in all the other **case** statements. An important thing to note about the **switch** statement is that the **case** values may only be constant integral expressions. It isn't legal to use **case** like this: int a = 10;

```
int b = 10;
int c = 20;
switch ( a ) {
case b:
/* Code */
break;
```

```
case c:
/* Code */
break;
default:
/* Code */
break;
}
```

The **default** case is optional, but it is wise to include it as it handles any unexpected cases. It can be useful to put some kind of output to alert you to the code entering the **default** case if you don't expect it to. Switch statements serve as a simple way to write long **if** statements when the requirements are met. Often it can be used to process input from a user.

Example: Below is a sample program, in which not all of the proper functions are actually declared, but which shows how one would use switch in a program.

```
#include <stdio.h>
void playgame();
void loadgame();
void playmultiplayer();
int main()
{
int input;
printf( "1. Play game\n" );
printf( "2. Load game\n" );
```

```
printf( "3. Play multiplayer\n" );
printf( "4. Exit\n" );
printf( "Selection: " );
scanf( "%d", &input );
switch ( input ) {
case 1: /* Note the colon, not a semicolon */
playgame();

break;
case 2:
loadgame();
break;
case 3:
playmultiplayer();
break;
case 4:
printf( "Thanks for playing!\n" );
break;
default:
printf( "Bad input, quitting!\n" );
break;
}
getchar();
}
```

This program will compile, but cannot be run until the undefined functions are given bodies, but it serves as a model (albeit simple) for processing input. If you do not understand this then try mentally putting in **if** statements for the **case** statements. **default** simply skips out of the **switch case** construction and allows the program to terminate naturally. If you do not like that, then you can make a loop around the whole thing to have it wait for valid input. You could easily make a few small functions if you wish to test the code.

Self Assessment Questions:

5.The conditional operator does not produce a lvalue. (True/False)

6.The condition of a **switch** statement is a _____.

7.Switch statement is an unconditional branching statement. (True/False)

4.6 The *while* loop

Loops generally consist of two parts: one or more *control expressions* which control the execution of the loop, and the *body*, which is the statement or set of statements which is executed over and over.

The most basic *loop* in C is the **while** loop. A **while** loop has one control expression, and executes as long as that expression is true. Here before executing the body of the loop, the condition is tested. Therefore it is called an **entry-controlled** loop. The following example repeatedly doubles the number 2 (2, 4, 8, 16, ...) and prints the resulting numbers as long as they are less than 1000:

```
int x = 2;

while(x < 1000)
{
```

```
printf("%d\n", x);
x = x * 2;
}
```

(Once again, we've used braces {} to enclose the group of statements which are to be executed together as the body of the loop.)

The general syntax of a **while** loop is
while(*expression*)
statement(s)
A **while** loop starts out like an **if** statement: if the condition expressed by the *expression* is true, the *statement* is executed. However, after executing the statement, the condition is tested again, and if it's still true, the statement is executed again. (Presumably, the condition depends on some value which is changed in the body of the loop.) As long as the condition remains true, the body of the loop is executed over and over again. (If the condition is false right at the start, the body of the loop is not executed at all.) As another example, if you wanted to print a number of blank lines, with the variable **n** holding the number of blank lines to be printed, you might use code like this:

```
while(n > 0)
{
printf("\n");
n = n - 1;
}
```

After the loop finishes (when control “falls out" of it, due to the condition being false), n will have the value 0.
You use a **while** loop when you have a statement or group of statements which may have to be executed a number of times to complete their task. The controlling expression represents the condition “the loop is not done" or “there's

more work to do." As long as the expression is true, the body of the loop is executed; presumably, it makes at least some progress at its task. When the expression becomes false, the task is done, and the rest of the program (beyond the loop) can proceed. When we think about a loop in this way, we can see an additional important property: if the expression evaluates to “false" before the very first trip through the loop, we make *zero* trips through the loop. In other words, if the task is already done (if there's no work to do) the body of the loop is not executed at all. (It's always a good idea to think about the “boundary conditions" in a piece of code, and to make sure that the code will work correctly when there is no work to do, or when there is a trivial task to do, such as sorting an array of one number. Experience has shown that bugs at boundary conditions are quite common.)

Program 4.5: Program to find largest of n numbers

```
main()
{
int num, large, n, i;

clrscr();
printf("enter number of numbers \n");
scanf(“%d”,&n);
large=0;
i=0;
while(i<n)
{
printf("\n enter number "); scanf(“%d”, &num);
if(large<num)
large=num;
i++;
```

```
}
printf("\n large = %d", large); }
```

Program 4.6: Program to evaluate sine series sin(x)=x-x^3/3!+x^5/5!x^7/7!+----- depending on accuracy

```
# include<stdio.h>
# include <math.h>
void main()
{
int n, i=1,count;
float acc, x, term, sum;
printf("enter the angle\n");
scanf("%d", &x);
x=x*3.1416/180.0;
printf("\nenter the accuracy)";
scanf("%f", &acc);
sum=x;
term=x;
while ((fabs(term))>acc)
{
term=-term*x*x/((2*i)*(2*i+1));
sum+=term;
i++;
}
```

```
printf"\nsum of sine series is %f", sum);
}
```

Self Assessment Questions

8.A _____ loop starts out like an **if** statement .

9.*while* is an entry-controlled loop. (True/False)

4.7 The do...while loop

The **do...while** loop is used in a situation where we need to execute the body of the loop before the test is performed. Therefore, the body of the loop may not be executed at all if the condition is not satisfied at the very first attempt. Where as while loop makes a test of condition before the body of the loop is executed.

For above reasons **while** loop is called an **entry-controlled loop** and **do..while** loop is called an **exit-controlled loop**.

do while loop takes the following form:

```
do
{
Body of the loop
}
while ( expression);
```

On reaching the **do** statement , the program proceeds to evaluate the body of the loop first. At the end of the loop, the conditional expression in the **while** statement is evaluated. If the expression is true, the program continues to evaluate the body of the loop once again. This process continues as long as the expression is true. When the expression becomes

false, the loop will be terminated and the control goes to the statement that appears immediately after the **while** statement.

On using the **do** loop, the body of the loop is always executed at least once irrespective of the expression.

Program 4.7: A program to print the multiplication table from 1 x 1 to 10 x 10 as shown below using do-while loop.

1 2 3 4 10
2 4 6 8 20
3 6 9 12 30
4 40
. .
. .
10 100

```
// Program to print multiplication table
main()
{
int rowmax=10,colmax=10,row,col,x;
printf(" Multiplication table\n");
printf("..................................\n");
row=1;
do
{
col=1;
```

```
do
{
x=row*col;
printf(“%4d”, x);
col=col+1;
}
while (col<=colmax);
printf(“\n”);;
row=row+1;
}
while(row<=rowmax);
Printf("..... \ n");
}
```

Self Assessment Questions:

10.On using the _____, the body of the loop is always executed at least once irrespective of the expression.

11.*do…while* is an entry-controlled loop. (True/False)

4.8 The *for* loop

The **for** loop is used to repeat the execution of set of statements for a fixed number of times. The **for** loop is also an **entry-controlled loop.**

Generally, the syntax of a for loop is
for(expr1; expr2; expr3) statement(s)
(Here we see that the **for** loop has three control expressions. As always, the statement can be a brace-enclosed block.)

Many loops are set up to cause some variable to step through a range of values, or, more generally, to set up an initial condition and then modify some value to perform each succeeding loop as long as some condition is true. The three expressions in a **for** loop encapsulate these conditions: expr1 sets up the initial condition, expr 2 tests whether another trip through the loop should be taken, and expr3 increments or updates things after each trip through the loop and prior to the next one. Consider the following :

```
for (i = 0; i < 10; i = i + 1)
printf("i is %d\n", i);
```

In the above example, we had i = 0 as expr1, i < 10 as expr2 , i = i + 1 as expr3, and the call to **printf** as statement, the body of the loop. So the loop began by setting i to 0, proceeded as long as i was less than 10, printed out i's value during each trip through the loop, and added 1 to i between each trip through the loop.

When the compiler sees a for loop, first, expr1 is evaluated. Then, expr2 is evaluated, and if it is true, the body of the loop (statement) is executed. Then, expr3 is evaluated to go to the next step, and expr2 is evaluated again, to see if there is a next step. During the execution of a for loop, the sequence is:
expr1
expr2
statement
expr3
expr2

statement
expr3
...
expr2
statement
expr3
expr2
The first thing executed is expr1. expr3 is evaluated after every trip through the loop. The last thing executed is always expr2, because when expr2 evaluates false, the loop exits.

All three expressions of a for loop are optional. If you leave out expr1, there simply is no initialization step, and the variable(s) used with the loop had better have been initialized already. If you leave out expr2, there is no test, and the default for the for loop is that another trip through the loop should be taken (such that unless you break out of it some other way, the loop runs forever). If you leave out expr3, there is no increment step.
The semicolons separate the three controlling expressions of a for loop. (These semicolons, by the way, have nothing to do with statement terminators.) If you leave out one or more of the expressions, the semicolons remain. Therefore, one way of writing a deliberately infinite loop in C is
for(;;)
...
It's also worth noting that a for loop can be used in more general ways than

the simple, iterative examples we've seen so far. The “control variable" of a for loop does not have to be an integer, and it does not have to be incremented by an additive increment. It could be “incremented" by a multiplicative factor (1, 2, 4, 8, ...) if that was what you needed, or it could be a floating-point variable, or it could be another type of variable which we haven't met yet which would step, not over numeric values, but over the elements of an array or other data

structure. Strictly speaking, a **for loop** doesn't have to have a “control variable" at all; the three expressions can be anything, although the loop will make the most sense if they are related and together form the expected initialize, test, increment sequence.

The powers-of-two example using **for** is:

```
int x;
for(x = 2; x < 1000; x = x * 2)
printf("%d\n", x);
```

There is no earth-shaking or fundamental difference between the **while** and for **loops**. In fact, given the general for loop

```
for(expr1; expr2; expr3)
statement
```

you could usually rewrite it as a while loop, moving the initialize and increment expressions to statements before and within the loop: expr1;

```
while(expr2)
{
statement
expr3;
}
```

Similarly, given the general while loop

```
while(expr)
statement
```

you could rewrite it as a for loop:

```
for(; expr; )
statement
```

Another contrast between the **for** and **while** loops is that although the test expression (expr2) is optional in a **for** loop, it is required in a **while** loop. If you leave out the controlling expression of a **while** loop, the compiler will complain about a syntax error. (To write a deliberately infinite **while** loop, you have to supply an expression which is always nonzero. The most obvious one would simply be while(1) .)

If it's possible to rewrite a **for** loop as a **while** loop and vice versa, why do they both exist? Which one should you choose? In general, when you choose a for loop, its three expressions should all manipulate the same variable or data structure, using the initialize, test, increment pattern. If they don't manipulate the same variable or don't follow that pattern, wedging them into a **for** loop buys nothing and a **while** loop would probably be clearer. (The reason that one loop or the other can be clearer is simply that, when you see a **for** loop, you expect to see an idiomatic initialize/ test/ increment of a single variable, and if the **for** loop you're looking at doesn't end up matching that pattern, you've been momentarily misled.) **Program 4.8: A Program to find the factorial of a number**

```
void main()
{
int M,N;
long int F=1;
clrscr();
printf(“enter the number\n”)";
scanf(“%d”,&N);
if(N<=0)
F=1;
else
{
for(M=1;M<=N;M++)
F*=M;
```

}
printf("the factorial of the number is %ld",F); getch();
}

Self Assessment Questions

12.**for** loop is an **exit-controlled** loop. (True/False)

13.The "control variable" of a **for** loop does not have to be an integer. (True/False)

4.8.1 The Nesting of for loops

Nesting of **for** loops, that is, one **for** statement within another **for** statement, is allowed in C. For example, two loops can be nested as follows: .....

.....

for(i=1;i<10;i++)

{

.....

.....

for(j=1;j<5;j++)

{

.....

.....

}

.....

.....

}

.....

.....

4.9 The *break* statement and continue statement

The purpose of ***break*** statement is to break out of a loop (**while**, **do while**, or **for** loop) or a **switch** statement. When a **break** statement is encountered inside a loop, the loop is immediately exited and the program continues with the statement immediately following the loop. When the loops are nested, the **break** would only exit from the loop containing it. That is, the **break** would exit only a single loop.

Syntax : break;

Program 4.9: Program to illustrate the use of *break* statement. void main ( )

```
{
int x;
for (x=1; x<=10; x++)
{
if (x==5)
break; /*break loop only if x==5 */
printf("%d", x);
}
printf ("\nBroke out of loop");
```

```
printf( “at x =%d“);
}
```

The above program displays the numbers from 1to 4 and prints the message “Broke out of loop when 5 is encountered.

The *continue statement*

The ***continue*** statement is used to continue the next iteration of the loop by skipping a part of the body of the loop (for, **do/while** or **while** loops). The *continue* statement does not apply to a **switch**, like a ***break*** statement. Unlike the **break** which causes the loop to be terminated, the **continue**, causes the loop to be continued with the next iteration after skipping any statements in between.

Syntax: continue;

Program 4.10: Program to illustrate the use of *continue* statement. void main ( ) {

```
int x;
for (x=1; x<=10; x++)
{ if (x==5)
continue; /* skip remaining code in loop
only if x == 5 */
printf (“%d\n”, x);
}
printf(“\nUsed continue to skip”);
}
```

The above program displays the numbers from 1to 10, except the number 5.

Program 4.11: Program to sum integers entered interactively

```
#include <stdio.h>
int main(void)
{
long num;
long sum = 0L; /* initialize sum to zero */
int status;

printf("Please enter an integer to be summed. "); printf("Enter q to quit.\n");
status = scanf("%ld", &num);
while (status == 1)
{
sum = sum + num;
printf("Please enter next integer to be summed. "); printf("Enter q to quit.\n");
status = scanf("%ld", &num);
}
printf("Those integers sum to %ld.\n", sum); return 0;
}
```

4.10 Summary

In C by default, statements are executed in sequence, one after another. We can, however, modify that sequence

by using ***control flow constructs.*** C language possesses decision making capabilities and supports the following statements known as the *control* or *decision making* statements: ***goto, if, switch.*** The **goto** statement is used to branch unconditionally from one point to another in the program. The simplest way to modify the control
flow of a program is with an **if** statement. **switch** statements serve as a simple way to write long **if** statements when the requirements are met.

The most basic *loop* in C is the **while** loop. A **while** loop has one control expression, and executes as long as that expression is true. **do..while loop** is used in a situation where we need to execute the body of the loop before the test is performed. The **for** loop is used to execute the set of statements repeatedly for a fixed number of times. It is an entry controlled loop. **break** statement is used to exit any loop. Unlike the **break** which causes the loop to be terminated, the **continue**, causes the loop to be continued with the next iteration after skipping any statements in between.

4.11 Terminal Questions

1.Consider the following program segment:

```
x=1;
y=1;

if (n>0)
x=x+1;
y=y-1;
printf("%d %d",x,y);
```

What will be the values of x and y if n assumes a value of (a) 1 and (b)0.

2.Rewrite the following without using compound relation:

```
if (grade<=59 && grade>=50)
second = second +1;
```

3.Write a program to check whether an input number is odd or even.

4.Write the **output** that will be generated by the following C program: void main()

```
{
int i=0, x=0;
while (i<20)
{
if (i%5 == 0)
{
x+=i;
printf("%d\t", i);
}
i++;
}
printf("\nx=%d"; x);
}
```

5.Write the output that will be generated by the following C program:

```
void main()
{
int i=0, x=0;
do
{
if (i%5 == 0)
{
x++;
printf("%d\t", x);
}
++i;
} while (i<20);
printf("\nx=%d", x);

}
```

4.12 Answers to Self Assessment Questions

1 labe

. l

2 true

. false Page No.: 84_3

.

4.if

5. true

6. value

7. false

8. while

9. true

10. do…while

11. false

12. false

13. true

4.13 Answers to Terminal Questions

1. (a) x=2, y=0

(b)x=1, y=0

2. if (grade<=59)

if (grade>=50)

second = second+1;

3. void main()

{

int no;

printf(“enter a number\n”);

scanf(“%d”,&no);

if (no%2==0)

printf(“even number\n”); else printf(“odd number\n”);
}

4. 0 5 10 15 x = 30
5. 1 2 3 4 x = 4

4.14 Exercises

1.Explain different types of **if** statements with examples.
2.Explain the syntax of **switch** statement with an example.
3.Write a program to check whether a given number is odd or even using **switch** statement.
4.Write a program to find the smallest of 3 numbers using **if-else** statement.
5.Write a program to find the roots of a quadratic equation.
6.Compare the following statements
a)*while* and *do…while*
b)*break* and *continue*

7.Write a program to compute the sum of digits of a given number using **while** loop.

8.Write a program that will read a positive integer and determine and print its binary equivalent using **do…while** loop.
9.The numbers in the sequence
1 1 2 3 5 8 13 ……..

are called Fibonacci numbers. Write a program using **do...while** loop to calculate and print the first n Fibonacci numbers.

10.Find errors, if any, in each of the following segments. Assume that all the variables have been declared and assigned values.

(a)

```
while (count !=10);
{
count = 1;
sum = sum + x;
count = count + 1;
}
```

(b)

```
do;
total = total + value;
scanf("%f", &value);
while (value ! =999);
```

11.Write programs to print the following outputs using **for** loops. (a)1 b) 1

2 22 2

3 3 3 3 3 3

4 4 4 4 4 4 4 4

12.Write a program to read the age of 100 persons and count the number of persons in the age group 50 to 60. Use **for** and **continue** statements.

13.Write a program to print the multiplication table using nested **for** loops.

Unit 5 Functions

Structure:

5.1 Introduction

Objectives

5.2 Function Basics

5.3 Function Prototypes

5.4 Recursion

5.5 Function Philosophy

5.6 Summary

5.7 Terminal Questions

5.8 Answers for Self Assessment Questions

5.9 Answers for Terminal Questions

5.10 Exercises

5.1 Introduction

In the previous unit, you studied about the control statements and its usage in C. You also studied how those control statements are helpful in making decisions when various types of conditions and options are available in the problem. In this unit, you will study about what a function is.

A function is a “black box" that we've locked part of our program into. The idea behind a function is that it *compartmentalizes* part of the program, and in particular, that the code within the function has some useful properties:

It performs some well-defined task, which will be useful to other parts of the program. It might be useful to other programs as well; that is, we might be able to reuse it (and without having to rewrite it).

The rest of the program doesn't have to know the details of how the function is implemented. This can make the rest of the program easier to think about. The function performs its task *well*. It may be written to do a little more than is required by the first program that calls it, with the anticipation that the calling program (or some other program) may later need the extra functionality or improved performance. (It's important that a finished function do its job well, otherwise there might be a reluctance to call it, and it therefore might not achieve the goal of reusability.)
By placing the code to perform the useful task into a function, and simply calling the function in the other parts of the program where the task must be performed, the rest of the program becomes clearer: rather than having some large, complicated, difficult-to-understand piece of code repeated wherever the task is being performed, we have a single simple function call, and the name of the function reminds us which task is being performed.

Since the rest of the program doesn't have to know the details of how the function is implemented, the rest of the program doesn't care if the function is re-implemented later, in some different way (as long as it continues to perform its same task, of course!). This means that one part of the program can be rewritten, to improve performance or add a new feature (or simply to fix a bug), without having to rewrite the rest of the program.
Functions are probably the most important weapon in our battle against software complexity. You'll want to learn when it's appropriate to break processing out into functions (and also when it's not), and *how* to set up function interfaces to best achieve the qualities mentioned above: reusability, information hiding, clarity, and maintainability.

Objectives:

After studying this unit, you should be able to:

explain the importance of functions

implement the concepts of formal arguments and actual arguments explain function declaration(function prototypes) and function definition use the concept of recursion

explain how the concept of functions reduces software complexity

5.2 Function Basics

A function is a self-contained program segment that carries out some specific, well-defined task. Every C program contains one or more functions. One of these functions must be called **main**. Program execution will always begin by carrying out the instructions in main. Additional functions will be subordinate to main, and perhaps to one another. So what defines a function? It has a *name* that you call it by, and a list of zero or more *arguments* or *parameters*. Parameters (also called formal parameters) or arguments are the special identifiers through which information can be passed to the function. A function has a *body* containing the actual instructions (statements) for carrying out the task the function is supposed to perform; and it may give you back a *return value*, of a particular type.

In general terms, the first line can be written as

data-type name(data-type parameter 1, data-type parameter 2, …, data-type parameter n)

Example 5.1: Here is a very simple function, which accepts one argument, multiplies it by 4, and hands that value back.

```
int multbyfour(int x)
{
```

```
int retval;
retval = x * 4;
return retval;
}
```

On the first line we see the return type of the function (int), the name of the function (multbyfour), and a list of the function's arguments, enclosed in parentheses. Each argument has both a name and a type; **multbyfour** accepts one argument, of type **int**, named x. The name x is arbitrary, and is used only within the definition of **multbyfour**. The caller of this function only needs to know that a single argument of type **int** is expected; the caller does not need to know what name the function will use internally to refer to that argument. (In particular, the caller does not have to pass the value of a variable named x.)

Next we see, surrounded by the familiar braces, the body of the function itself. This function consists of one declaration (of a local variable *retval*) and two statements. The first statement is a conventional expression statement, which computes and assigns a value to *retval*, and the second statement is a *return* statement, which causes the function to return to its caller, and also specifies the value which the function returns to its caller.

In general term, a *return* statement is written as
return expression
The *return* statement can return the value of any expression, so we don't really need the local *retval* variable; this function can also be written as

```
int multbyfour(int x)
{
return x * 4;
}
```

How do we call a function? We've been doing so informally since day one, but now we have a chance to call one that we've written, in full detail. The arguments in the function call are referred to as *actual arguments or actual parameters,* in contrast to the formal arguments that appear in the first line of the function definition.

Here is a tiny skeletal program to call multbyfour:

```
#include <stdio.h>
extern int multbyfour(int);
int main()
{
int i, j;
i = 5;
j = multbyfour(i);
printf("%d\n", j);
return 0;
}
```

This looks much like our other test programs, with the exception of the new line

```
extern int multbyfour(int);
```

This is an *external function prototype declaration*. It is an external declaration, in that it declares something which is defined somewhere else. (We've already seen the defining instance of the function **multbyfour**, but may be the compiler hasn't seen it yet.) The function prototype declaration contains the three pieces of information about the function that a caller needs to know: the function's name, return type, and argument type(s). Since we don't care what name the **multbyfour** function will use to refer to
its first argument, we don't need to mention it. (On the other hand, if a function takes several arguments, giving them

names in the prototype may make it easier to remember which is which, so names may optionally be used in function prototype declarations.) Finally, to remind us that this is an external declaration and not a defining instance, the prototype is preceded by the keyword **extern**.

The presence of the function prototype declaration lets the compiler know that we intend to call this function, **multbyfour**. The information in the prototype lets the compiler generate the correct code for calling the function, and also enables the compiler to check up on our code (by making sure, for example, that we pass the correct number of arguments to each function we call).
Down in the body of main, the action of the function call should be obvious: the line
j = multbyfour(i);

calls B, passing it the value of i as its argument. When **multbyfour** returns, the return value is assigned to the variable j. (Notice that the value of main's local variable i will become the value of **multbyfour's** parameter x; this is absolutely not a problem, and is a normal sort of affair.)

This example is written out in “longhand," to make each step equivalent. The variable i isn't really needed, since we could just as well call j = multbyfour(5);

And the variable j isn't really needed, either, since we could just as well call printf("%d\n", multbyfour(5));

Here, the call to **multbyfour** is a subexpression which serves as the second argument to **printf**. The value returned by **multbyfour** is passed immediately to **printf**. (Here, as in general, we see the flexibility and
generality of expressions in C. An argument passed to a function may be an arbitrarily complex subexpression, and a function call is itself an expression which may be embedded as a subexpression within arbitrarily complicated surrounding expressions.)

We should say a little more about the mechanism by which an argument is passed down from a caller into a function. Formally, C is *call by value*, which means that a function receives *copies* of the values of its arguments. We can illustrate this with an example. Suppose, in our implementation of **multbyfour**, we had gotten rid of the unnecessary **retval** variable like this:

```
int multbyfour(int x)
{
x = x * 4;
return x;
}
```

We might wonder, if we wrote it this way, what would happen to the value of the variable i when we called
j = multbyfour(i);

When our implementation of **multbyfour** changes the value of x, does that change the value of i up in the caller? The answer is no. x receives a copy of i's value, so when we change x we don't change i.

However, there is an exception to this rule. When the argument you pass to a function is not a single variable, but is rather an array, the function does *not* receive a copy of the array, and it therefore *can* modify the array in the caller. The reason is that it might be too expensive to copy the entire array, and furthermore, it can be useful for the function to write into the caller's array, as a way of handing back more data than would fit in the function's single return value. We will discuss more about passing arrays as arguments to a function later.
There may be several different calls to the same function from various places within a program. The actual arguments may differ from one function call to another. Within each function call, however, the actual arguments must

correspond to the formal arguments in the function definition; i.e the actual arguments must match in number and type with the corresponding formal arguments.

Program 5.1: A program to find the largest of three integers

```
#include<stdio.h>
main()
{
int x, y, z, w;
/* read the integers */
int max(int, int);
printf("\nx= ");
scanf("%d", &x);
printf("\ny= ");
scanf("%d", &y);
printf("\nz= ");
scanf("%d", &z);
/* Calculate and display the maximum value */
w= max(x,y);
printf("\n\nmaximum=%d\n", max(z,w));
}
int max(int a, int b)
{
int c;
c=(a>=b)?a:b;
```

```
return c;
}
```

Please execute this program and observe the result.

Function calls can span several levels within a program; function A can call function B, which can call function C and so on.

Program 5.2: Program to check whether a given integer is a perfect square or not.

```
#include<stdio.h>
main()
{
int psquare(int);
int num;
printf(“ Enter the number:”);
scanf(“%d”, &num);

if(psquare(num)) /* main() calls the function psquare() */ {
printf(“%d is a perfect square\n”);
else
printf(“%d is not a perfect square\n”);
}
}

int psquare(int x)
{
```

```
int positive(int);
float sqr;
if(positive(x)) /* psquare() in turn calls the function positive() */ {
sqr=sqrt(x));
if(sqr-int(sqr))==0)
return 1;
else
return 0;
}
int positive(int m)
{
if(m>0)
return 1;
else return 0;
}
```

Execute the above program and observe the result.

In the above program the **main** function calls the function psquare() and it in turn calls the function positive() to check whether the number to be checked for perfect square is a positive or not before checking.

The **return** statement can be absent altogether from a function definition, though this is generally regarded as a poor programming practice. If a function reaches end of the block without encountering a return statement, control simply reverts back to the calling portion of the program without returning any information. Using an empty return statement(without the accompanying expressions) is recommended.

Example 5.2: The following function accepts two integers and determines the larger one, which is then written out. The function doesn't return any information to the calling program.

```
void max(int x, int y)
{
int m;
m=(x>=y)?x:y;
printf(" The larger integer is=%d\n", m);
return;
}
```

Self Assessment Questions

1.The function main() is optional in a C program. (True/False)

2.If the function is defined elsewhere (not in the same program where it is called), the function prototype must be preceded by the keyword _____.

3.The arguments that appear in function definition are called

_____ arguments whereas the arguments that appear when the function is called are the _____ arguments.

4.The return data type, function name and the list of formal parameters enclosed in brackets are separated by _____.

5.3 Function Prototypes

In modern C programming, it is considered good practice to use prototype declarations for all functions that you call.

As we mentioned, these prototypes help to ensure that the compiler can generate correct code for calling the functions, as well as allowing the compiler to catch certain mistakes you might make.

In general terms, a function prototype can be written as
data-type name(type1, type2, …, type n)

Examples:
int sample(int, int) or int sample(int a, int b);
float fun(int, float) or float fun(int a, float b);
void demo(void); Here **void** indicates function neither return any value to the caller nor it has any arguments.

If you write the function definition after the definition of its caller function, then the prototype is required in the caller, but the prototype is optional if you write the definition of the function before the definition of the caller function. But it is good programming practice to include the function prototype wherever it is defined.

If prototypes are a good idea, and if we're going to get in the habit of writing function prototype declarations for functions we call that we've written (such as multbyfour), what happens for library functions such as **printf?** Where are their prototypes? The answer is in that boilerplate line

#include <stdio.h>

we've been including at the top of all of our programs. stdio.h is conceptually a file full of external declarations and other information pertaining to the “Standard I/O" library functions, including **printf**. The #include directive arranges all of the declarations within **stdio.h** that are considered by the compiler, rather as if we'd typed them all in ourselves. Somewhere within these declarations is an external function prototype declaration for **printf**, which satisfies the rule

that there should be a prototype for each function we call. (For other standard library functions we call, there will be other “header files" to include.) Finally, one more thing about external function prototype declarations: we've said that the distinction between external declarations and defining instances of normal variables hinges on the presence or absence of the keyword **extern**. The situation is a little bit different for functions. The “defining instance" of a function is the function, including its body (that is, the brace-enclosed list of declarations and statements implementing the function). An external declaration of a function, even without the keyword **extern**, looks nothing like a function declaration. Therefore, the keyword **extern** is optional in function prototype declarations. If you wish, you can write
int multbyfour(int);
and this is just like an external function prototype declaration as extern int multbyfour(int);
(In the first form, without the **extern**, as soon as the compiler sees the semicolon, it knows it's not going to see a function body, so the declaration can't be a definition.) You may want to stay in the habit of using **extern** in all external declarations, including function declarations, since “extern = external declaration" is an easier rule to remember.

Program 5.3: Program to illustrate that the function prototype is optional in the caller function. The program is to convert a character from lowercase to uppercase.

```
#include<stdio.h>
char lower_to_upper(char ch) /* Function definition precedes main()*/ {
char c;
c=(ch>='a' && ch<='z') ? ('A'+ch-'a'):ch;
return c;
}

main()
{
char lower, upper;
```

```
/* char lower_to_upper(char lower); */ /* Function prototype is optional here*/
printf("Please enter a lowercase character:");
scanf("%c", &lower);
upper=lower_to_upper(lower);
printf("\nThe uppercase equivalent of %c is %c\n", lower, upper); }
```

Self Assessment Questions

5.Function prototype is also called _____.

6.The function prototypes are optional. (True/False)

7.The function prototypes of the library functions is in the corresponding _____ files.

8.The function prototype for the function fun() called in main() below is _____ .

```
main()
{ double x, y, z; …
z=fun(x, y); …

}
```

5.4 Recursion

Recursion is a process by which a function calls itself repeatedly, until some specified condition has been met. The process is used for repetitive computations in which each action is stated in terms of a previous result. Many repetitive problems can be written in this form.

In order to solve a problem recursively, two conditions must be satisfied. First, the problem must be written in a recursive form, and the second, the problem statement must include a stopping condition.

Example 5.3: Factorial of a number. Suppose we wish to calculate the factorial of a positive integer, n. We would normally express this problem as n!=1 x 2 x 3 x … x n.

This can also be written as n!=n x (n-1)!. This is the recursive statement of the problem in which the desired action(the calculation of n!) is expressed in terms of a previous result (the value of (n-1)! which is assumed to be known). Also, we know that 0!=1 by definition. This expression provides stopping condition for the recursion.

Thus the recursive definition for finding factorial of positive integer n can be written as:

fact(n)={ 1 if n=0
n x fact(n-1) otherwise}

Program 5.4: Program to find factorial of a given positive integer #include<stdio.h>

```
main()
{
int n;
long int fact(int);
/* Read in the integer quantity*/
scanf("%d", &n);
/*calaculate and display the factorial*/
printf("n!=%ld\n", fact(n));
}
long int fact(int n)
```

```
{
if(n==0)
return(1);
else
return (n*fact(n-1));
}
```

Please execute this program and observe the result.

Example 5.4: The Towers of Hanoi. The Towers of Hanoi is a game played with three poles and a number of different sized disks. Each disk has a hole in the center, allowing it to be stacked around any of the poles. Initially, the disks are stacked on the leftmost pole in the order of decreasing size, i.e, the largest on the bottom, and the smallest on the top as illustrated in Figure 5.1.

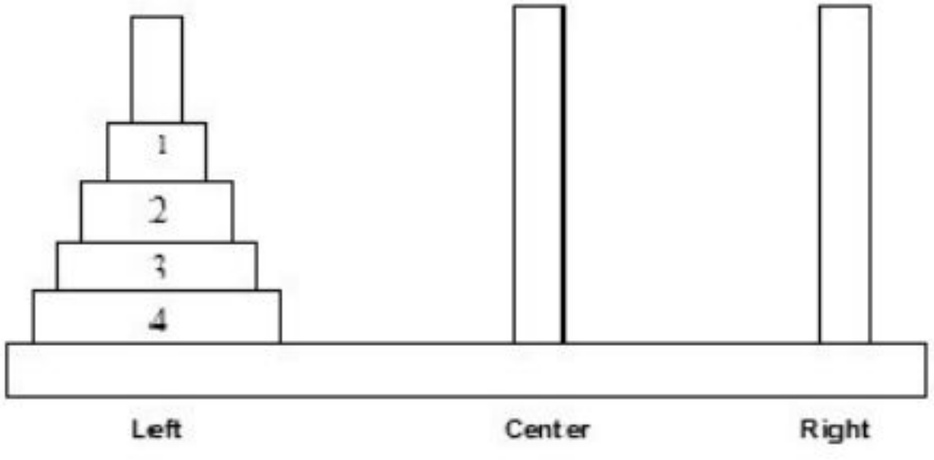


Figure 5.1: Illustration for Tower of Hanoi

The aim of the game is to transfer the disks from the leftmost pole to the rightmost pole, without ever placing a larger disk on top of a smaller disk. Only one disk may be moved at a time, and each disk must always be placed around one of the poles.

The general strategy is to consider one of the poles to be the origin, and another to be the destination. The third pole will be used for intermediate storage, thus allowing the disks to be moved without placing a larger disk over a smaller one. Assume there are n disks, numbered from smallest to largest as in Figure 5.1. If the disks are initially stacked on the left pole, the problem of moving all n disks to the right pole can be stated in the following recursive manner:

1.Move the top n-1 disks from the left pole to the center pole.

2.Move the nth disk(the largest disk) to the right pole.

3.Move the n-1 disks on the center pole to the right pole.

The problem can be solved for any value of n greater than 0(n=0 represents a stopping condition).

In order to program this game, we first label the poles, so that the left pole is represented as L, the center pole as C and the right pole as R. Let us refer the individual poles with the char-type variables **from**, **to** and **temp** for the origin, destination and temporary storage respectively.

Program 5.5: Recursive Program to solve Towers of Hanoi problem.

```
#include<stdio.h>
main()
{
void Recursive_Hanoi(int, char, char, char);
int n;
printf(" Towers of Hanoi\n\n");
printf(" How many disks?");
scanf("%d", &n);
printf("\n");
Recusrive_Hanoi(n, 'L', 'R', 'C');
}
```

```
void Recursive_Hanoi(int n, char from, char to, char temp)
{
/* Transfer n disks from one pole to another */
/* n= number of disks
from=origin
to=destination
temp=temporary storage */
{
if(n>0){
/* move n-1 disks from origin to temporary */ Recursive_Hanoi(n-1, from, temp, to);

/* move nth disk from origin to destination */ printf(" Move disk %d from %c to %c\n", n, from, to);

/* move n-1 disks from temporary to destination */
Recursive_Hanoi(n-1, temp, to, from);
}
return;
}
```

Please execute this program and observe the result

Self Assessment Questions

9._____ is a process by which a function calls itself repeatedly, until some specified condition is satisfied

10.A stopping condition must be there in a recursive definition.

(True/False)

11.The output of the following program is _____.

```
#include<stdio.h>
main()
{
int n=5;
int fun(int n);
printf("%d\n", fun(n));
}
int fun(int n)
{
if(n==0)
return 0;
else
return (n+fun(n-1));

}
```

5.5 Function Philosophy

What makes a good function? The most important aspect of a good "building block" is that have a single, well-defined task to perform. When you find that a program is hard to manage, it's often because it has not been designed and

broken up into functions cleanly. Two obvious reasons for moving code down into a function are because:
1.It appeared in the main program several times, such that by making it a

function, it can be written just once, and the several places where it used to appear can be replaced with calls to the new function.

2.The main program was getting too big, so it could be made (presumably) smaller and more manageable by lopping part of it off and making it a
function.

These two reasons are important, and they represent significant benefits of well-chosen functions, but they are not sufficient to automatically identify a good function. As we've been suggesting, a good function has at least these two additional attributes:

3.It does just one well-defined task, and does it well.
4.Its interface to the rest of the program is clean and narrow.
Attribute 3 is just a restatement of two things we said above. Attribute 4 says

that you shouldn't have to keep track of too many things when calling a function. If you know what a function is supposed to do, and if its task is simple and well-defined, there should be just a few pieces of information you have to give it to act upon, and one or just a few pieces of information which it returns to you when it's done. If you find yourself having to pass lots and lots of information to a function, or remember details of its internal implementation to make sure that it will work properly this time, it's often a sign that the function is not sufficiently well-defined. (A poorly-defined function may be an arbitrary chunk of code that was ripped out of a main program that was getting too big, such that it essentially has to have access to all of that main function's local variables.)

The whole point of breaking a program up into functions is so that you don't have to think about the entire program at once; ideally, you can think about just one function at a time. We say that a good function is a “black box," which is supposed to suggest that the “container" it's in is opaque – callers can't see inside it (and the function inside can't see out). When you call a function, you only have to know what it does, not how it does it. When you're *writing* a function, you only have to know what it's supposed to do, and you don't have to know why or under what circumstances its caller will be calling it. (When designing a function, we should perhaps think about the callers just enough to ensure that the function we're designing will be easy to call, and that we aren't accidentally setting things up so that callers will have to think about any internal details.)

Some functions may be hard to write (if they have a hard job to do, or if it's hard to make them do it truly well), but that difficulty should be compartmentalized along with the function itself. Once you've written a “hard" function, you should be able to sit back and relax and watch it do that hard work on call from the rest of your program. It should be pleasant to notice (in the ideal case) how much easier the rest of the program is to write, now that the hard work can be deferred to this workhorse function.

(In fact, if a difficult-to-write function's interface is well-defined, you may be able to get away with writing a quick-and-dirty version of the function first, so that you can begin testing the rest of the program, and then go back later and rewrite the function to do the hard parts. As long as the function's original interface anticipated the hard parts, you won't have to rewrite the rest of the program when you fix the function.)

The functions are important for far more important reasons than just saving typing. Sometimes, we'll write a function which we only call once, just because breaking it out into a function makes things clearer and easier.

If you find that difficulties pervade a program, that the hard parts can't be buried inside black-box functions and then forgotten about; if you find that there are hard parts which involve complicated interactions among multiple functions, then the program probably needs redesigning.

For the purposes of explanation, we've been seeming to talk so far only about “main programs" and the functions they call and the rationale behind moving some piece of code down out of a “main program" into a function. But in reality, there's obviously no need to restrict ourselves to a two-tier scheme. Any function we find ourselves writing will often be appropriately written in terms of sub-functions, sub-sub-functions, etc.

Program 5.6: Program to create a function that types 65 asterisks in a row

```
/* letterhead1.c */
#include <stdio.h>
#define NAME "MEGATHINK, INC."
#define ADDRESS "10 Megabuck Plaza"
#define PLACE "Megapolis, CA 94904"
#define LIMIT 65
void starbar(void); /* prototype the function */

int main(void)
{
starbar();
printf("%s\n", NAME);
printf("%s\n", ADDRESS);
printf("%s\n", PLACE);
```

```
starbar(); /* use the function */
return 0;
}

void starbar(void) /* define the function */
{
int count;
for (count = 1; count <= LIMIT; count++)
putchar('*');
putchar('\n');
}
```

Self Assessment Questions

12.By modularizing the problem into different sub problems. Each sub problem can be implemented as a _____.

13.The main purpose of function is to save typing time. (True/False)

5.6 Summary

A function is a self-contained program segment that carries out some specific, well-defined task. When you find that a program is hard to manage, it's often because it has not been designed and broken up into functions cleanly. A function is a “black box" that we've locked part of our program into. The idea behind a function is that it *compartmentalizes* part of the program. The function main() is must in every C program. The function prototype is nothing but the function declaration. Recursion is a process by which a function calls itself repeatedly, until some specified condition has been met.

5.7 Terminal Questions

1.What is the significance of the keyword 'void'?
2.What is the difference between function declaration and function definition?
3.Write a recursive function to find sum of even numbers from 2 to 10.
4.Write a recursive definition to find gcd of two numbers.
5.Write a recursive definition to find nth fibonacci number. The Fibonacci series forms a sequence of numbers in which each number is equal to the sum of the previous two numbers. In other words,

Fi=Fi-1 + Fi-2

where F refers to the ithi Fibonacci number. The first two Fibonacci numbers are 0 and 1, i.e, F1=0, F2=1;
6.What is the output of the following program ?

```
#include<stdio.h>
main()
{
int m, count;
int fun(int count);
for(count=1;count<=10;count++) {
m=fun(count);
printf("%d", m);
}
```

```
}
int fun(int n)
{
int x;
x= n*n;
return x;

}
```

5.8 Answers for Self Assessment Questions

1. False
2. extern.
3.formal, actual
4. comma.
5.Function declaration
6. False
7. header
8.double fun(double, double);
9. Recursion.
10. True
11. 15
12. function
13. False

5.9 Answers for Terminal Questions

1.‘void’ is the keyword used to specify that the function doesn’t return any value. It can also be used to specify the absence of arguments.
2.Function declaration is a direction to the compiler that what type of data is returned by the function, the function name and about the arguments where as the function definition is actually writing the body of the function along with the function header.
3. #include<stdio.h>
main()
{
int n=10;
int fun(int n);
printf(“%d”, fun(n));
}
int fun(int n)
{
if(n>0) return (n+fun(n-2));
}
4.gcd(m,n)= {m or n if m=n
GCD(m, m-n) if m>n
GCD(n,m) if m<n }
5. fib(i)= { 0 if i=1
1 if i=2

fib(i-1)+fib(i-2) otherwise}

6. Square of the integers from 1 to 10 is displayed.

5.10 Exercises

1.Suppose function F1 calls function F2 within a C program. Does the order of function definitions make any difference? Explain.

2.When a program containing recursive function calls is executed, how are the local variables within the recursive function interpreted?

3.Express the following algebraic formula in a recursive form:

Y = (x1+x2+...+xn)

4.Write a function that will allow a floating point number to be raised to an integer power.

5.Write a function to swap two numbers using pass by value technique. What is the drawback of the function?

Unit 6 Storage Classes

Structure:

6.1 Introduction

Objectives

6.2 Storage Classes and Visibility

6.3 Automatic or local variables

6.4 Global variables

6.5 Static variables
6.6 External variables
6.7 Summary
6.8 Terminal Questions
6.9 Answers for Self Assessment Questions
6.10 Answers for Terminal Questions
6.11 Exercises

6.1 Introduction

In the previous unit, you studied about functions. You found out that how functions can be used to break the large problems into small problems and then solve it. You studied how functions can be repeatedly called and used again and again. In this unit, you will study about the types of storage classes that are used in C. You will study how these storage classes are useful in making the C language a very powerful computing language.

Variables are channels of communication within a program. You set a variable to a value at one point in a program, and at another point (or points) you read the value out again. The two points may be in adjoining statements, or they may be in widely separated parts of the program. How long does a variable last? How widely separated can the setting and fetching parts of the program be, and how long after a variable is set does it persist? Depending on the variable and how you're using it, you might want different answers to these questions. For example, in some situations it may be desirable to introduce certain “global” variables that are recognized throughout the entire program (or within major portions of the program, e.g. two or more functions). Such variables are defined differently than the usual “local” variables, which are recognized only within a single function.

We will also consider the issue of static variables which can retain their values, so that the function can be reentered later and the computation resumed.

Finally, we may be required to develop a large, multifunction program in terms of several independent files, with few functions defined within each file. In such programs, the individual functions can be defined and accessed locally within a single file, or globally within multiple files.

Objectives:

After studying this unit, you should be able to:

implement the concept of storage classes and visibility of variables explain the difference between automatic variables, global variables, static variables and external variables.

compile and execute a program made up of more than one source files.

6.2 Storage Classes and Visibility

There are two ways to categorize variables: by *data type*, and by *storage class*. Data type refers to the type of information represented by a variable, for example, integer number, floating-point number, character etc. Storage class refers to the persistence of a variable and its scope within the program, that is, the portion of the program over which the variable is recognized.

The following types of storage-class specifications in C are discussed in this unit: **global**, **automatic** or **local**, **static,** and **extern**. The exact procedure for establishing a storage class for a variable depends upon the particular storage class, and the manner in which the program is organized, (i.e. single file vs. multiple file).

The *visibility* of a variable determines how much of the rest of the program can access that variable. You can arrange that a variable is visible only within one part of one function, or in one function, or in one source file, or anywhere in the program.

Why would you want to limit the visibility of a variable? For maximum flexibility, wouldn't it be handy if all variables were potentially visible everywhere? As it happens, that arrangement would be *too* flexible: everywhere in a program, you would have to keep track of the names of all the variables declared anywhere else in the program, so that you didn't accidentally re-use one. Whenever a variable had the wrong value by mistake, you'd have to search the entire program for the bug, because any statement in the entire program could potentially have modified that variable. You would constantly be stepping all over yourself by using a common variable name like i in two parts of your program, and having one snippet of code accidentally overwrite the values being used by another part of the code.

Self Assessment Questions

1.The visibility of a variable determines how much of the rest of the program can access that variable. (True/False)

2._____ class refers to the persistence of a variable and its scope within the program, that is, the portion of the program over which the variable is recognized.

3.Visibility provides security for your data used in a program. (True/False)

6.3 Automatic or local variables

A variable declared within the braces {} of a function is visible only within that function; variables declared within functions are called *local variables*. Their scope is confined to that function. You can use the keyword **auto** to declare

automatic variables, but, however it is optional. If another function somewhere else declares a local variable with the same name, it's a different variable entirely, and the two don't clash with each other. If an automatic variable is not initialized in some manner, however, its initial value will be unpredictable and contains some garbage value.

Program 6.1: Program to find factorial of a number

```
#include<stdio.h>
main()
{
auto int n; /* Here the keyword auto is optional */
long int fact(int);
printf("read the integer n:");
scanf("%d", &n);
printf("\nn!=%ld", fact(n) );
}
long int fact(auto int n) /* n is local to the function fact() and auto is optional*/ {
auto int i; /* Here the keyword auto is optional */
auto long int factorial=1; /* Here the keyword auto is optional */ while(n>0)
{
factorial=factorial*n;
n=n-1;
}
return factorial;
}
```

An automatic variable doesn't retain its value once control is transferred out of its defining function. Therefore, any value assigned to an automatic variable within a function will be lost once the function is exited.

Self Assessment Questions

4.The scope of an automatic variable is in _____ in which it is declared.

5.Does an automatic variable retain its value once control is transferred out of its defining function? (Yes/No)

6.The key word **auto** is must in the declaration of automatic variables. (True/False)

6.4 Global Variables

A variable declared outside of any function is a *global variable*, and it is potentially visible anywhere within the program. You use global variables when you *do* want to use the variable in any part of the program. When you declare a global variable, you will usually give it a longer, more descriptive name (not something generic like i) so that whenever you use it you will remember that it's the same variable everywhere. The values stored in global variables persist, for as long as the program does. (Of course, the values can in general still be overwritten, so they don't necessarily persist forever.)

Program 6.2: Program to find average length of several lines of text

```
#include<stdio.h>
/* Declare global variables outside of all the functions*/
```

```
int sum=0; /* total number of characters */
int lines=0; /* total number of lines */
main()
{
int n; /* number of characters in given line */
float avg; /* average number of characters per line */
void linecount(void); /* function declaraction */
float cal_avg(void);
printf("Enter the text below:\n");
while((n=linecount())>0) {
sum+=n;
++lines;
}

avg=cal_avg();
printf("\nAverage number of characters per line: %5.2f", avg);
}
void linecount(void)
{
/* read a line of text and count the number of characters */
char line[80];
int count=0;
while((line[count]=getchar())!='\n')
++count;
```

```
return count;
}
float cal_avg(void)
{
/* compute average and return*/
return (float)sum/lines;
}
```

In the above program the variables sum and lines are globally declared and hence they could be used in both the functions main() and cal_avg()

Self Assessment Questions

7.The variables declared in the main() function are the global variables. (True/False)

8.The global variables are more secured than the automatic variables in a program. (True/False)

6.5 Static Variables

Static variables are defined within individual functions and therefore have the same scope as automatic variables, i.e. they are local to the functions in which they are declared. Unlike automatic variables, however, static variables retain their values throughout the life of the program. As a result, if a function is exited and then reentered later, the static variables defined within that function will retain their previous values. This feature allows functions to retain

information permanently throughout the execution of a program. Static variables can be utilized within the function in the same manner as other variables. They cannot be accessed outside of their defining function.

In order to declare a static variable the keyword **static** is used as shown below:
static int count;
You can define automatic or static variables having the same name as global variables. In such situations the local variables will take precedence over the global variables, though the values of global variables will be unaffected by any manipulation of the local variables.

Initial values can be included in static variable declaration. The rules associated with the initialization remain same as the initialization of automatic or global variables. They are:
1.The initial values must be constants, not expressions.
2.The initial values are assigned to their respective variables at the

beginning of the program execution. The variables retain these values throughout the life of the program, unless different values are assigned during the course of computation.
3.Zeros will be assigned to all static variables whose declarations do not include explicit initial values.

Program 6.3: Program to generate Fibonacci numbers.

```
#include<stdio.h>
main()
{
int count, n;
long int fib(int);
```

```
printf("\n How many Fibonacci numbers?");
scanf("%d\n", &n);
for(count=1;count<=n;count++)
{
printf("\ni=%d F=%ld", count, fib(count));
}

long int fib(int count)
{
/* calculate a Fibonacci number using the formula if i=1, F=0; if i=2, F=1, and F=F1+F2 for i>=3 */
static long int f1=0, f2=1; /* declaration of static variables */ long int f;
if (count==1)
f=0;
else if (count==2)
f=1;
else
f=f1+f2;
f2=f1;

f1=f; /* f1 and f2 retain their values between different calls of the function*/
return f;
}
```

Self Assessment Questions

9.The scope of static variables and automatic variables is the same.

(True/False)

10._____ variables retain their values throughout the life of the program. As a result, if a function is exited and then reentered later, the static variables defined within that function will retain their previous values.

11.By default, a static variable is initialized to _____.

6.6 External Variables

It is possible to split a function up into several source files, for easier maintenance. When several source files are combined into one program the compiler must have a way of correlating the variables which might be used to communicate between the several source files. Furthermore, if a variable is going to be useful for communication, there must be exactly one of it: you wouldn't want one function in one source file to store a value in one variable named **externvar**, and then have another function in another source file read from a *different* variable named **externvar**. Therefore, a variable should have exactly one *defining instance*, in one place in one source file. If the same variable is to be used anywhere else (i.e. in some other source file or files), the variable is declared in those other file(s) with an *external declaration*, which is not a defining instance. The external declaration says the compiler that the variable will be used in this source file but defined in some other source file. Thus the compiler doesn't allocate space for that variable with this source file.

To make a variable as an external declaration, which is defined somewhere else; you precede it with the keyword **extern**:

extern int j;

Program 6.4: Program to illustrate the concept of external variables.

Type and save the following program in a source file called

externvariables.h

```
int principle=10000;
float rate=5.5;
int time=2;
float interest;
```

Type and save the following program in a separate source file called **demoexternvar.c**

```
#include<stdio.h>
#include “externvariables.h” /* the source file where the external variables are defined should be included here.*/

main()
{
/* external declarations of the variables which are defined in externvariables.h */
extern int principle;
extern float rate;
extern int time;
extern float interest;
/*compute interest*/
interest= principle*rate*time/100.0;
printf(“Interest=%f\n”, interest);
}
```

Compile demoexternvar.c and execute the program.

The concept of external storage class can be extended to functions also. A source file can access a function defined in any other source file provided the source file is included within the source file where you access that function.

Program 6.5: Program to illustrate the concept of external functions. Type and save the following program in a file externfunction.h

```
void output(void)
{
printf(“ Hi, Manipal!\n”);
return;
}
```

Type and save the following program in a separate source file called **demoexternfun.c**

```
#include<stdio.h>
#include “ externfunction.h”
extern void output(void);
main()
{
output();
}
```

Compile and execute the above program and observe the result. However, the keyword **extern** is optional in some C compilers.

Self Assessment Questions

12.The main purpose of using external variables is to access the same variable in different _____ files.

13.Compiler doesn't allocate memory for an external variable where it is accessed. (True/False)

14.Global variables and external variables have the same scope. (True/False)

Example 6.1: Here is an example demonstrating almost everything we've seen so far:

```
int globalvar = 1;
extern int anotherglobalvar;
static int privatevar;
f()
{
int localvar;
int localvar2 = 2;
static int persistentvar;
}
```

Here we have six variables, three declared outside and three declared inside of the function f().

globalvar is a global variable. The declaration we see is its defining instance (it happens also to include an initial value). **globalvar** can be used anywhere in this source file, and it could be used in other source files, too (as long as corresponding external declarations are issued in those other source files).

anotherglobalvar is a second global variable. It is *not* defined here; the defining instance for it (and its initialization) is somewhere else.

privatevar is a "private" global variable. It can be used anywhere within this source file, but functions in other source files cannot access it, even if they try to issue external declarations for it. (If other source files try to declare a global variable called **"privatevar"**, they'll get their own; they won't be sharing this one.) Since it has static duration and

receives no explicit initialization, privatevar will be initialized to 0.
localvar is a local variable within the function f(). It can be accessed only within the function f(). (If any other part of the program declares a variable named **"localvar"**, that variable will be distinct from the one we're looking at here.) **localvar** is conceptually "created" each time f() is called, and disappears when f() returns. Any value which was stored in **localvar** last time f() was running will be lost and will not be available next time f() is called. Furthermore, since it has no explicit initializer, the value of **localvar** will in general be garbage each time f() is called.

localvar2 is also local, and everything that we said about **localvar** applies to it, except that since its declaration includes an explicit initializer, it will be initialized to 2 each time f() is called.

Finally, **persistentvar** is again local to f(), but it *does* maintain its value between calls to f(). It has static duration but no explicit initializer, so its initial value will be 0.

The term *declaration* is a general one which encompasses defining instances and external declarations; defining instances and external declarations are two different kinds of declarations. Furthermore, either kind of declaration suffices to inform the compiler of the name and type of a particular variable (or function). If you have the defining instance of a global variable in a source file, the rest of that source file can use that variable without having to issue any external declarations. It's only in source files where the defining instance hasn't been seen that you need external declarations.

You will sometimes hear a defining instance referred to simply as a "definition," and you will sometimes hear an external declaration referred to simply as a "declaration." These usages are mildly ambiguous, in that you can't tell out of context whether a "declaration" is a generic declaration (that might be a defining instance or an external declaration) or whether it's an external declaration that specifically is not a defining instance. Similarly, there are other

constructions that can be called “definitions" in C, namely the definitions of preprocessor macros, structures, and typedefs etc.

Program 6.6: Program to illustrate the hiding of variables in blocks /*

```
hiding.c -- variables in blocks */
#include <stdio.h>
int main()
{
int x = 30;
printf("x in outer block: %d\n", x);
{
int x = 77; /* new x, hides first x */
printf("x in inner block: %d\n", x);
}
printf("x in outer block: %d\n", x);
while (x++ < 33)
{
int x = 100; /* new x, hides first x */
x++;
printf("x in while loop: %d\n", x);
}
return 0;
}
```

6.7 Summary

Variables are channels of communication within a program. Storage class refers to the persistence of a variable and its scope within the program, that is, the portion of the program over which the variable is recognized. The scope of a local or automatic variable is confined to the function where it is defined. A global variable is potentially visible anywhere within the program in which it is defined. Static variables retain their values throughout the life of the program. As a result, if a function is exited and then reentered later, the static variables defined within that function will retain their previous values. The external variable declaration says the compiler that the global variable will be used in this source file but defined in some other source file.

6.8 Terminal Questions

1.List some of the storage classes available in C.
2.What is the use of header file? Is the use of header file absolutely necessary?
3.What is the difference between declaration and definition of function?
4.What is the significance of external declaration?
5.How can you justify that variables are channels of communication in a program?

6.9 Answers to Self Assessment Questions

1. True

2. Storage

3. True

4.The function in which it is declared.

5.No

6. False

7. False

8. False

9. True

10. Static.

11. Zero

12. source

13. True

14. False

6.9 Answers for Terminal Questions

1.automatic, global, static, extern

2.Header files are used to define some variables and functions separately in a library. Built-in header files are absolutely necessary if you want to access the variables and functions defined in them.

3.Declaration is nothing but the prototype that contains the type of returned data, name of the function and type of the arguments. But the definition contains the function header and the body of the function.

4.The external declaration says the compiler that the variable will be used in this source file but defined in some other source file.

5.You set a variable to a value at one point in a program, and at another point (or points) you read the value out again. Thus the transfer of information from one point of the program to another is nothing but the communication.

6.10 Exercises

1.Distinguish between the following
i.Global and local variables
ii.Automatic and static variables
iii.Global and extern variables

2.Write a program to count the number of times a function is called using static variables.
3.Write a function prime that returns 1 if its argument is a prime number and returns zero Otherwise.
4.Write a function that will round a floating point number to an indicated decimal place. For example, the number 12.456 would yield the value
12.46 when it is rounded off to two decimal places.

5.Write a program to illustrate the concept of extern variables.

Unit 7 Arrays and Strings

Structure:
7.1 Introduction

Objectives
7.2 One Dimensional Arrays

Passing Arrays to Functions
7.3 Multidimensional Arrays
7.4 Strings
7.5 Summary
7.6 Terminal Questions
7.7 Answers for Self Assessment Questions
7.8 Answers for Terminal Questions
7.9 Exercises

7.1 Introduction

In the previous unit, you studied about the various types of storage classes that are used in C. You studied how those storage classes are used in different situations in C. In this unit, you will study about the arrays and strings. You will learn how arrays and strings are formed and manipulated.

Many applications require processing of multiple data items that have common characteristics. In such situations it is always convenient to place the data items into an array, where they will share the same name. An array is a collection of similar type of elements. All elements in the array are referred with the array name. Since arrays hold a group of data, it is very easy to perform looping and arithmetic operations on group of data. This chapter covers the processing of both one-dimensional and two-dimensional arrays.

Objectives:
After studying this unit, you should be able to:

declare, initialize and process one-dimensional and two-dimensional arrays
explain about strings and how to process them
describe the library functions available in C to process strings

7.2 One Dimensional Arrays

So far, we've been declaring simple variables: the declaration
int i;

declares a single variable, named i, of type **int**. It is also possible to declare an *array* of several elements. The declaration
int a[10];

declares an array, named a, consisting of ten elements, each of type **int**. Simply speaking, an array is a variable that can hold more than one value. You specify which of the several values you're referring to at any given time by using a numeric *subscript*. (Arrays in programming are similar to vectors or matrices in mathematics.) We can represent the array a above with a picture like this:

a:
[0] [1] [2] [3] [4] [5] [6] [7] [8] [9]

In C, arrays are *zero-based*: the ten elements of a 10-element array are numbered from 0 to 9. The subscript which specifies a single element of an array is simply an integer expression in square brackets. The first element of the array is a[0], the second element is a[1], etc. You can use these “array subscript expressions" anywhere you can use the name of a simple variable, for example:

```
a[0] = 10;

a[1] = 20;
a[2] = a[0] + a[1];
```

Notice that the subscripted array references (i.e. expressions such as a[0] and a[1]) can appear on either side of the assignment operator.

The subscript does not have to be a constant like 0 or 1; it can be any integral expression. For example, it's common to loop over all elements of an array:

```
int i;
for(i = 0; i < 10; i = i + 1)
a[i] = 0;
```

This loop sets all ten elements of the array a to 0.

Arrays are a real convenience for many problems, but there is not a lot that C will do with them for you automatically. In particular, you can neither set all elements of an array at once nor assign one array to another; both of the assignments

```
a = 0; /* WRONG */
```

and

```
int b[10];
b = a; /* WRONG */
```

are illegal.
To set all of the elements of an array to some value, you must do so one by

one, as in the loop example above. To copy the contents of one array to another, you must again do so one by one:
int b[10];

for(i = 0; i < 10; i = i + 1)
b[i] = a[i];
Remember that for an array declared
int a[10];
there is no element a[10]; the topmost element is a[9]. This is one reason that zero-based loops are also common in C.
Note that the for loop for(i = 0; i < 10; i = i + 1)
...
does just what you want in this case: it starts at 0, the number 10 suggests (correctly) that it goes through 10 iterations, but the less-than comparison means that the last trip through the loop has i set to 9. (The comparison i <= 9 would also work, but it would be less clear and therefore poorer style.) In the little examples so far, we've always looped over all 10 elements of the

sample array a. It's common, however, to use an array that's bigger than necessarily needed, and to use a second variable to keep track of how many elements of the array are currently in use. For example, we might have an integer variable

int na; /* number of elements of a[] in use */
Then, when we wanted to do something with a (such as print it out), the loop would run from 0 to na, not 10 (or whatever a's size was):

```
for(i = 0; i < na; i = i + 1)
printf("%d\n", a[i]);
```

Naturally, we would have to ensure that na's value was always less than or equal to the number of elements actually declared in a.
Arrays are not limited to type **int**; you can have arrays of **char** or **double** or any other type.
Here is a slightly larger example of the use of arrays. Suppose we want to investigate the behavior of rolling a pair of dice. The total roll can be anywhere from 2 to 12, and we want to count how often each roll comes up. We will use an array to keep track of the counts: a[2] will count how many times we've rolled 2, etc.

We'll simulate the roll of a die by calling C's random number generation function, rand(). Each time you call rand(), it returns a different, pseudorandom integer. The values that rand() returns typically span a large range, so we'll use C's *modulus* (or "remainder") operator % to produce random numbers in the range we want. The expression rand() % 6 produces random numbers in the range 0 to 5, and rand() % 6 + 1 produces random numbers in the range 1 to 6.

Program 7.1: Program to simulate the roll of a die

```
#include <stdio.h>
#include <stdlib.h>
main()
{
int i;
int d1, d2;
int a[13]; /* uses [2..12] */
```

```
for(i = 2; i <= 12; i = i + 1) a[i] = 0;

for(i = 0; i < 100; i = i + 1) {
d1 = rand() % 6 + 1;
d2 = rand() % 6 + 1;
a[d1 + d2] = a[d1 + d2] + 1; }

for(i = 2; i <= 12; i = i + 1) printf("%d: %d\n", i, a[i]);

return 0;
}
```

We include the header <stdlib.h> because it contains the necessary declarations for the rand() function. We declare the array of size 13 so that its highest element will be a[12]. (We're wasting a[0] and a[1]; this is no great loss.) The variables d1 and d2 contain the rolls of the two individual dice; we add them together to decide which cell of the array to increment, in the line

```
a[d1 + d2] = a[d1 + d2] + 1;
```

After 100 rolls, we print the array out. Typically, we'll see mostly 7's, and relatively few 2's and 12's.

7.2.1 Passing Arrays to Functions

An array name can be used as an argument to a function, thus permitting the entire array to be passed to the function. To pass an array to a function, the array name must appear by itself, without brackets or subscripts, as an actual argument within the function call. The corresponding formal argument is written in the same manner, though it must be declared as an array within the formal argument declarations. When declaring a one-dimensional array as a formal

argument, the array name is written with a pair of empty square brackets. The size of the array is not specified within the formal argument declaration.

Program 7.2: The following program illustrates the passing of an array from the main to a function. This program is used to find the average of n floating point numbers.

```
#include<stdio.h>
main()
{
int n, i;
float avg;
float list[100];
float average(int, float[]); /* function prototype */
printf("How many numbers:");
scanf("%d",&n);
printf(" Enter the numbers:");
for(i=1;i<=n;i++)
scanf("%f", &list[i]);
avg=average(n, list); /* Here list and n are actual arguments */
printf("Average=%f\n", avg);
}

float average(int a, float x[ ])
{
float avg;
float sum=0;
```

```
int i;
for(i=0;i<a;i++)

sum=sum+x[i]; /* find sum of all the numbers */ avg=sum/a; /* find average */
return avg;
}
```

Self Assessment Questions

1.In C, an array subscript starts from _____

2.An array name is a pointer. (True/False)

3.Will there be a compilation error for the following program segment?(Yes/No)

```
int a[5] = {1, 2, 3, 4, 5};
int b[5] = {5, 4, 3, 2, 1};
int c[5][5];
...
c=a+b;
...
```

7.3 Multidimensional Arrays

The C language allows arrays of any dimension to be defined. In this section, we will take a look at two-dimensional arrays. One of the most natural applications for a two-dimensional array arises in the case of a matrix. In C, the two-dimensional matrix can be declared as follows:

int array[3][6];

Following is the way of declaring as well as initializing two-dimensional arrays. int array[3][6] = {

{4,5,6,7,8,9},

{1,5,6,8,2,4},

{0,4,4,3,1,1}

};

Such arrays are accessed like so:

array[1][4]= -2;

if (array[2][1] > 0) {

printf ("Element [2][1] is %d", array[2][1]);

}

Remember that, like ordinary arrays, two-dimensional arrays are numbered

from 0. Therefore, the array above has elements from array[0][0] to array[2][5].

Program 7.3: Program to add two matrices

```
#include <stdio.h>
main()
{
int a[5][5], b[5][5], c[5][5];
int i, j, m, n;
printf("Enter the order of the matrices:");
scanf("%d%d", &m, &n);
printf(" Enter the elements of A matrix:\n");
for(i=0;i<m;i++)
```

```
for(j=0;j<n;j++)
scanf("%d", &a[i][j]);
printf("Enter the elements of B matrix:\n");
for(i=0;i<m;i++)
for(j=0;j<n;j++)
scanf("%d", &b[i][j]);
/* Add the matrices */
for(i=0;i<m;i++)
for(j=0;j<n;j++)
c[i][j] = a[i][j]+b[i][j];
/* Print the sum */
printf("The sum of matrices:\n");
for(i=0;i<m;i++)
{
for(j=0;j<n;j++)
printf("%d\t", c[i][j]);
printf("\n");
}
}
```

Multidimensional arrays are processed in the same manner as onedimensional arrays, on an element-by-element basis. However, some care is required when passing multidimensional arrays to a function. In particular, the formal argument declarations within the function definition must include explicit size specifications in all of the subscript positions except the first. These size specifications must be consistent with the corresponding size specifications in

the calling program. The first subscript position may be written as an empty pair of square brackets, as with a one-dimensional array. The corresponding function prototypes must be written in the same manner. But while calling the function the array name may be passed as the actual argument as in the case of one-dimensional arrays. E.g:

```
void process_array (int [][6]); /* function prototype */
void process_array (int array[][6])/*function definition */
{
...
}
```

Self Assessment Questions

4.In a two-dimensional matrix, the first subscript in the declaration specifies number of _____.

5.A two-dimensional array is considered as an array of one-dimensional arrays. (True/False)

7.4 Strings

Strings in C are represented by arrays of characters. The end of the string is marked with a special character, the *null character*, which is simply the character with the value 0. (The null character has no relation except in name to the *null pointer*. In the ASCII character set, the null character is named NULL.) The null or string-terminating character is represented by another character escape sequence, \0.

Because C has no built-in facilities for manipulating entire arrays (copying them, comparing them, etc.), it also has very few built-in facilities for manipulating strings.

In fact, C's only truly built-in string-handling is that it allows us to use *string constants* (also called *string literals*) in our code. Whenever we write a string, enclosed in double quotes, C automatically creates an array of characters for us, containing that string, terminated by the \0 character. For example, we can declare and define an array of characters, and initialize it with a string constant:

```
char string[ ] = "Hello, world!";
```

In this case, we can leave out the dimension of the array, since the compiler can compute it for us based on the size of the initializer (14, including the terminating \0). This is the only case where the compiler sizes a string array for us, however; in other cases, it will be necessary that *we* decide how big the arrays and other data structures we use to hold strings are.

To do anything else with strings, we must typically call functions. The C library contains a few basic string manipulation functions, and to learn more about strings, we'll be looking at how these functions might be implemented.

Since C never lets us assign entire arrays, we use the **strcpy** function to copy one string to another:

```
#include <string.h>
char string1[ ] = "Hello, world!";
char string2[20];
strcpy(string2, string1);
```

The destination string is **strcpy's** first argument, so that a call to **strcpy** mimics an assignment expression (with the destination on the left-hand side). Notice that we had to allocate string2 big enough to hold the string that would be copied to it. Also, at the top of any source file where we're using the standard library's string-handling functions (such as **strcpy**) we must include the line

```
#include <string.h>
```

which contains external declarations for these functions.

Since C won't let us compare entire arrays, either, we must call a function to do that, too. The standard library's **strcmp** function compares two strings, and returns 0 if they are identical, or a negative number if the first string is alphabetically “less than" the second string, or a positive number if the first string is “greater." (Roughly speaking, what it means for one string to be “less than" another is that it would come first in a dictionary or telephone book, although there are a few anomalies.) Here is an example: char string3[] = "this is";
char string4[] = "a test";
if(strcmp(string3, string4) == 0)
printf("strings are equal\n");
else printf("strings are different\n");

This code fragment will print “strings are different". Notice that **strcmp** does *not* return a Boolean, true/false, zero/nonzero answer, so it's not a good idea to write something like
if(strcmp(string3, string4))
...

because it will behave backwards from what you might reasonably expect. (Nevertheless, if you start reading other people's code, you're likely to come across conditionals like if(strcmp(a, b)) or even if(!strcmp(a, b)). The first does something if the strings are unequal; the second does something if they're equal. You can read these more easily if you pretend for a moment that **strcmp's** name were **strdiff**, instead.)
Another standard library function is **strcat**, which concatenates strings. It does *not* concatenate two strings together and give you a third, new string; what it really does is append one string onto the end of another. (If it gave you a new string, it would have to allocate memory for it somewhere, and the standard library string functions generally never do that for you automatically.) Here's an example:

```
char string5[20] = "Hello, ";
char string6[] = "world!";
printf("%s\n", string5);
strcat(string5, string6);
printf("%s\n", string5);
```

The first call to **printf prints** “Hello, ", and the second one prints “Hello, world!", indicating that the contents of string6 have been tacked on to the end of string5. Notice that we declared string5 with extra space, to make room for the appended characters.

If you have a string and you want to know its length (perhaps so that you can check whether it will fit in some other array you've allocated for it), you can call **strlen**, which returns the length of the string (i.e. the number of characters in it), not including the \0:

```
char string7[ ] = "abc";
int len = strlen(string7);
printf("%d\n", len);
```

Finally, you can print strings out with **printf** using the %s format specifier, as we've been doing in these examples already (e.g. printf("%s\n", string5);).

Since a string is just an array of characters, all of the string-handling functions we've just seen can be written quite simply, using no techniques more complicated than the ones we already know. In fact, it's quite instructive to look at how these functions might be implemented. Here is a version of **strcpy**:

```
mystrcpy(char dest[ ], char src[ ])
{
int i = 0;
```

```
while(src[i] != '\0')
{
dest[i] = src[i];
i++;
}
dest[i] = '\0';
```

We've called it mystrcpy instead of strcpy so that it won't clash with the version that's already in the standard library. Its operation is simple: it looks at characters in the src string one at a time, and as long as they're not \0, assigns them, one by one, to the corresponding positions in the dest string. When it's done, it terminates the dest string by appending a \0. (After exiting the while loop, i is guaranteed to have a value one greater than the subscript of the last character in src.) For comparison, here's a way of writing the same code, using a for loop:

```
for(i = 0; src[i] != '\0'; i++)
dest[i] = src[i];
dest[i] = '\0';
```

Yet a third possibility is to move the test for the terminating \0 character out of the for loop header and into the body of the loop, using an explicit if and break statement, so that we can perform the test after the assignment and therefore use the assignment inside the loop to copy the \0 to dest, too: for(i = 0; ; i++)

```
{
dest[i] = src[i];
if(src[i] == '\0')
break;
}
```

(There are in fact many, many ways to write strcpy. Many programmers like to combine the assignment and test, using an expression like (dest[i] = src[i]) != '\0')

Here is a version of strcmp:

```
mystrcmp(char str1[ ], char str2[ ])
{
int i = 0;

while(1)
{
if(str1[i] != str2[i])
return str1[i] - str2[i];
if(str1[i] == '\0' || str2[i] == '\0') return 0;
i++;
}
}
```

Characters are compared one at a time. If two characters in one position differ, the strings are different, and we are supposed to return a value less than zero if the first string (str1) is alphabetically less than the second string. Since characters in C are represented by their numeric character set values, and since most reasonable character sets assign values to characters in alphabetical order, we can simply subtract the two differing characters from each other: the expression str1[i] - str2[i] will yield a negative result if the i'th character of str1 is less than the corresponding character in str2. (As it turns out, this will behave a bit strangely when comparing upper and lower-case letters, but it's the traditional approach, which the standard versions of strcmp tend to use.) If the characters are the same, we continue

around the loop, *unless* the characters we just compared were (both) \0, in which case we've reached the end of both strings, and they were both equal. Notice that we used what may at first appear to be an infinite loop--the controlling expression is the constant 1, which is always true. What actually happens is that the loop runs until one of the two return statements breaks out of it (and the entire function). Note also that when one string is longer than the other, the first test will notice this (because one string will contain a real character at the [i] location, while the other will contain \0, and these are not equal) and the return value will be computed by subtracting the real character's value from 0, or vice versa. (Thus the shorter string will be treated as “less than" the longer.)
Finally, here is a version of strlen:

```
int mystrlen(char str[ ])
{
int i;
for(i = 0; str[i] != '\0'; i++)
{ }
return i;
```

In this case, all we have to do is find the \0 that terminates the string, and it turns out that the three control expressions of the for loop do all the work; there's nothing left to do in the body. Therefore, we use an empty pair of braces { } as the loop body. Equivalently, we could use a *null statement*, which is simply a semicolon:

```
for(i = 0; str[i] != '\0'; i++)
;
```

Everything we've looked at so far has come out of C's standard libraries. As one last example, let's write a substr function, for extracting a substring out of a larger string. We might call it like this:

```
char string8[ ] = "this is a test";
char string9[10];
substr(string9, string8, 5, 4);
printf("%s\n", string9);
```

The idea is that we'll extract a substring of length 4, starting at character 5 (0-based) of string8, and copy the substring to string9. Just as with strcpy, it's our responsibility to declare the destination string (string9) big enough. Here is an implementation of substr. Not surprisingly, it's quite similar to strcpy:

```
substr(char dest[ ], char src[ ], int offset, int len)
{
int i;
for(i = 0; i < len && src[offset + i] != '\0'; i++)
dest[i] = src[i + offset];
dest[i] = '\0';
}
```

If you compare this code to the code for mystrcpy, you'll see that the only differences are that characters are fetched from src[offset + i] instead of src[i], and that the loop stops when len characters have been copied (or when the src string runs out of characters, whichever comes first). When working with strings, it's important to keep firmly in mind the differences between characters and strings. We must also occasionally remember the way characters are represented, and about the relation between character values and integers.

As we have had several occasions to mention, a character is represented internally as a small integer, with a value depending on the character set in use. For example, we might find that 'A' had the value 65, that 'a' had the value 97 and that '+' had the value 43. (These are, in fact, the values in the ASCII character set, which most computers use. However, you don't need to learn these values, because the vast majority of the time, you use character constants to refer to

characters, and the compiler worries about the values for you. Using character constants in preference to raw numeric values also makes your programs more portable.)

As we may also have mentioned, there is a big difference between a character and a string, even a string which contains only one character (other than the \0). For example, 'A' is *not* the same as "A". To drive home this point, let's illustrate it with a few examples.

If you have a string:
char string[ ] = "hello, world!";
you can modify its first character by saying
string[0] = 'H';
(Of course, there's nothing magic about the first character; you can modify any character in the string in this way. Be aware, though, that it is not always safe to modify strings in-place like this) Since you're replacing a character, you want a character constant, 'H'. It would *not* be right to write string[0] = "H"; /* WRONG */

because "H" is a string (an array of characters), not a single character. (The destination of the assignment, string[0], is a char, but the right-hand side is a string; these types don't match.)

On the other hand, when you need a string, you must use a string. To print a single newline, you could call
printf("\n");

It would *not* be correct to call
printf('\n'); /* WRONG */
printf always wants a string as its first argument. (As one final example, putchar wants a single character, so putchar('\n') would be correct, and putchar("\n") would be incorrect.)

We must also remember the difference between strings and integers. If we treat the character '1' as an integer, perhaps by saying

```
int i = '1';
```

we will probably *not* get the value 1 in i; we'll get the value of the character '1' in the machine's character set. (In ASCII, it's 49.) When we do need to find the numeric value of a digit character (or to go the other way, to get the digit character with a particular value) we can make use of the fact that, in any character set used by C, the values for the digit characters, whatever they are, are contiguous. In other words, no matter what values '0' and '1' have, '1' - '0' will be 1 (and, obviously, '0' - '0' will be 0). So, for a variable c holding some digit character, the expression

```
c - '0'
```

gives us its value. (Similarly, for an integer value i, i + '0' gives us the corresponding digit character, as long as 0 <= i <= 9.)

Just as the character '1' is not the integer 1, the string "123" is not the integer 123. When we have a string of digits, we can convert it to the corresponding integer by calling the standard function **atoi**:

```
char string[] = "123";
int i = atoi(string);
int j = atoi("456");
```

Self Assessment Questions:

6.Will there be a compilation error for the following program? (Yes/No). char str1[10];

```
str1 =”Hello, world”;
printf(“%s”, str1);
```

7.The library function used to copy one string to another is _____.
8.The library function *atoi* can be used for any string. (True/False)

7.5 Summary

An array is a variable that can hold more than one value. In C, arrays are zero-based. An array name can be used as an argument to a function, thus permitting the entire array to be passed to the function. The C language allows arrays of any dimension to be defined. One of the most natural applications for a two-dimensional array arises in the case of a matrix. Strings in C are represented by arrays of characters. C has built in library functions to perform some operations on strings.

7.6 Terminal Questions

1.Write a program for 10 times summation of square of a number.
2.How many elements can the array in the following declaration accommodate?
int a[3][4][5];
3.Is the following array declaration and initialization correct?
int a[2][2]={1,2,3,4};
4.When you pass an array as a parameter to a function, the entire array is copied and is available to function. Justify whether this statement is true or false.
5.Write a Program that uses loops for array processing.

7.7 Answers for Self Assessment Questions

1.0

2. True

3. Yes

4. rows

5. true

6.Compilation error

7. strcpy

8. false

7.8 Answers for Terminal Questions

1. #include<stdio.h>

main()

{

int i=0, sum=0, x; printf('Enter a number:");

scanf("%d", &x);

while(i<10)

{

sum+=x*x;

```
i++;
}
printf(“Sum=%d”, sum);
}
```

2 60

. Yes, It is correct.

3 Statement is False.

. **// loops for array processing**

```
4 #include <stdio.h>
. #define SIZE 10
5 #define PAR 72
. int main(void)
{
int index, score[SIZE];
int sum = 0;
float average;
printf("Enter %d golf scores:\n", SIZE);
for (index = 0; index < SIZE; index++)
scanf("%d", &score[index]); */read in the ten scores
printf("The scores read in are as follows:\n");
for (index = 0; index < SIZE; index++)
printf("%5d", score[index]); */verify input
printf("\n");
```

```
for (index = 0; index < SIZE; index++)
sum += score[index]; */add them up
average = (float) sum / SIZE; */ time-honored method printf("Sum of scores = %d, average = %.2f\n", sum, average);
printf("That's a handicap of %.0f.\n", average - PAR);
return 0;
}
```

7.9 Exercises

1.Write a program to count the number of vowels and consonants in a given string.
2.Write a program to arrange a list of numbers in ascending order.
3.Write a program to multiply two matrices.
4.Write a program to rewrite a given string in the alphabetical order.
5.Write a program to transpose a given matrix.

Unit 8 Pointers – I

Structure:

8.1 Introduction
Objectives
8.2 Basics of Pointers
8.3 Pointers and One-dimensional Arrays

Pointer Arithmetic
Pointer Subtraction and Comparison

Similarities between Pointers and One-dimensional Arrays
8.4 Summary
8.5 Terminal Questions
8.6 Answers to Self Assessment Questions
8.7 Answers to Terminal Questions
8.8 Exercises

8.1 Introduction

In the previous unit, you studied about the arrays and strings. You studied how the arrays and strings are declared and manipulated in making C a more easy and powerful programming language. In this unit, you will study about pointer which is another useful topic that makes C a most powerful programming language. You will study about the basics of pointer and pointer arithmetic.

A pointer is a variable that points at, or refers to, another variable. That is, if we have a pointer variable of type “pointer to int, “it might point to the int variable i, or to any one of the locations of the int array a. Given a pointer variable, we can ask questions like, “what’s the value of the variable that this pointer points to?”.

Why would we want to have a variable that refers to another variable? Why not just use that other variable directly? Pointers are used frequently in C, as they have number of useful applications. For example, pointers can be used to pass information back and forth between a function and its reference point. In particular, pointers provide a way to return

multiple data items from a function via function arguments. Pointers also permit references to other functions to be specified as arguments to a given function. This has the effect of passing functions as arguments to the given function.

Pointers are also closely associated with the arrays and therefore provide an alternate way to access individual array elements. Pointers make the program efficient when we use pointers for accessing the individual array elements. In this unit, you will study about the basics of pointers and its usage in one dimensional array.

Objectives:
After studying this unit, you should be able to:
implement pointers in your program
write a program related to arrays and using a pointer with it
solve and illustrate the use of pointer arithmetics in C
point out the similarities between pointers and one dimensional arrays

8.2 Basics of Pointers

A pointer is a variable which contains the address in memory of another variable. We can have a pointer to any variable type. Before studying pointers, you should mind certain symbols that are frequently used in implementing the pointers. We use the symbol **&** that gives the "address of a variable" and is known as the **unary** or **monadic** operator. We use other symbol * that gives the "contents of an object ***pointed to*** by a pointer" and we call it the **indirection** or **dereference** operator.

The first things to do with pointers are to declare a pointer variable, set it to point somewhere, and finally manipulate the value that it points to. A simple pointer declaration has the following general format:

*datatype *variablename*

where *datatype* represents the type of the data to which the pointer *variablename* points to. In simple terms, the *variablename* holds the address of the value of type *datatype*.

For example,
int *ip;

This declaration looks like our earlier declarations, with one obvious difference: that is the asterisk. The asterisk means that ip, the variable we're declaring, is not of type int, but rather of type pointer-to-int. (Another way of looking at it is that *ip, which as we'll see is the value pointed to by ip, will be an int.)

We may think of setting a pointer variable to point to another variable as a two-step process: first we generate a pointer to that other variable, and then we assign this new pointer to the pointer variable. We can say (but we have to be careful when we're saying it) that a pointer variable has a value, and that its value is “pointer to that other variable”. This will make more sense when we see how to generate pointer values.
Pointers (that is, pointer values) are generated with the “address-of” operator &, which we can also think of as the “pointer-to” operator. We demonstrate this by declaring (and initializing) an int variable i, and then setting ip to point to it:
int i = 5;
ip = &i;
The assignment expression ip = &i; contains both parts of the “two-step process”: &i generates a pointer to i, and the assignment operator assigns the new pointer to (that is, places it “in”) the variable ip. Now ip “points to” i, which we can illustrate with this picture:



i is a variable of type int, so the value in its box is a number, 5. ip is a variable of type pointer-to-int, so the "value" in its box is an arrow pointing at another box. Referring once again back to the "two-step process" for setting a pointer variable: the & operator draws us the arrowhead pointing at i's box, and the assignment operator =, with the pointer variable ip on its left, anchors the other end of the arrow in ip's box.

We discover the value pointed to by a pointer using the "contents-of" operator, *. Placed in front of a pointer, the * operator accesses the value pointed to by that pointer. In other words, if ip is a pointer, then the expression *ip gives us whatever it is that's in the variable or location pointed to by ip. For example, we could write something like

```
printf("%d\n", *ip);
```

which would print 5, since ip points to i, and i is (at the moment) 5.

(You may wonder how the asterisk * can be the pointer contents-of operator when it is also the multiplication operator. There is no ambiguity here: it is the multiplication operator when it sits between two variables, and it is the contents-of operator when it sits in front of a single variable. The situation is analogous to the minus sign: between two variables or expressions it's the subtraction operator, but in front of a single operator or expression it's the negation operator. Technical terms you may hear for these distinct roles are *unary* and *binary*: a *binary* operator applies to two operands, usually on either side of it, while a *unary* operator applies to a single operand.)

The contents-of operator * does not merely fetch values through pointers; it can also *set* values through pointers. We can write something like *ip = 7;

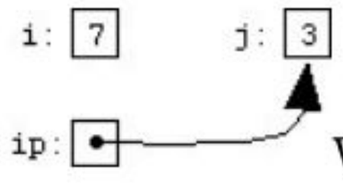
which means “set whatever ip points to 7.” Again, the * tells us to go to the location pointed to by ip, but this time, the location isn't the one to fetch from -- we're on the left-hand sign of an assignment operator, so *ip tells us the location to store *to*. (The situation is no different from array subscripting expressions such as a[3] which we've already seen appearing on both sides of assignments.)

The result of the assignment *ip = 7 is that i's value is changed to 7, and the picture changes to:

 If we called printf("%d\n", *ip) again, it would now print 7.

At this point, you may be wonder, if we wanted to set i to 7, why didn't we do it directly? We'll begin to explore that next, but first let's notice the difference between changing a pointer (that is, changing what variable it points to) and changing the value at the location it points to. When we wrote *ip = 7, we changed the value pointed to by ip, but if we declare another variable j:

int j = 3;
and write
ip = &j;
we've changed ip itself. The picture now looks like this:



We have to be careful when we say that a pointer assignment changes “what the pointer points to.” Our

earlier assignment

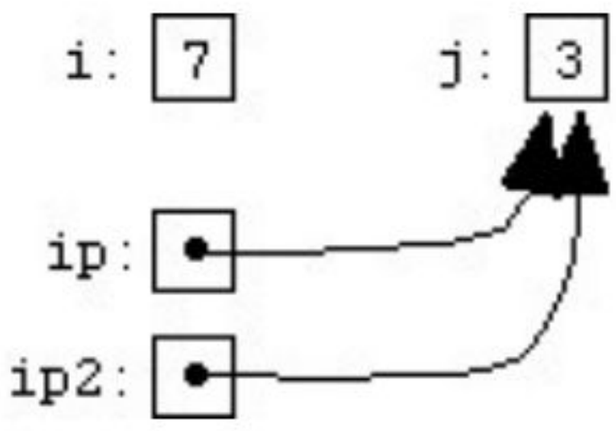
*ip = 7;
changed the value pointed to by ip, but this more recent assignment ip = &j;

has changed what *variable* ip points to. It's true that “what ip points to” has changed, but this time, it has changed for a different reason. Neither i (which is still 7) nor j (which is still 3) has changed. (What has changed is ip's value.) If we again call

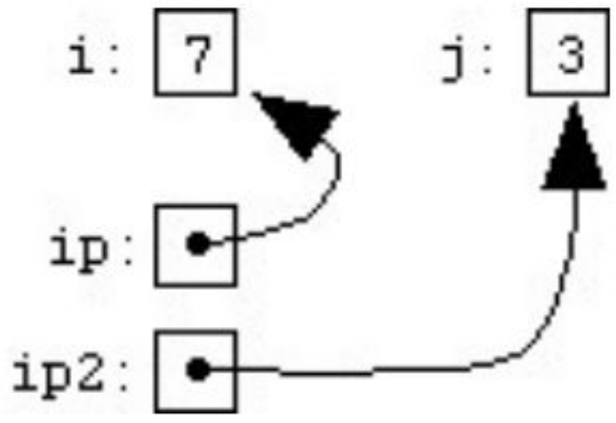
printf("%d\n", *ip);
this time it will print 3.

We can also assign pointer values to other pointer variables. If we declare a second pointer variable:
int *ip2;

then we can say
ip2 = ip;
Now ip2 points where ip does; we've essentially made a “copy” of the arrow:



Now, if we set ip to point back to i again: ip = &i;
the two arrows point to different places:



We can now see that the two assignments
ip2 = ip;
and
*ip2 = *ip;

do two very different things. The first would make ip2 again point to where ip points (in other words, back to i again). The second would store, at the location pointed to by ip2, a copy of the value pointed to by ip; in other words (if ip and ip2 still point to i and j respectively) it would set j to i's value, or 7.

It's important to keep very clear in your mind the distinction between *a pointer* and *what it points to*. You can't mix them. You can't “set ip to 5” by writing something like

ip = 5; /* WRONG */

5 is an integer, but ip is a pointer. You probably wanted to "set *the value pointed to by* ip to 5," which you express by writing

```
*ip = 5;
```

Similarly, you can't "see what ip is" by writing

```
printf("%d\n", ip); /* WRONG */
```

Again, ip is a pointer-to-int, but %d expects an int. To print *what* ip *points to*, use

```
printf("%d\n", *ip);
```

Finally, a few more notes about pointer declarations. The * in a pointer declaration is related to, but different from, the contents-of operator *. After we declare a pointer variable

```
int *ip;
```

the expression

```
ip = &i
```

sets what ip points to (that is, which location it points to), while the expression

```
*ip = 5
```

sets the value of the location pointed to by ip. On the other hand, if we declare a pointer variable and include an initializer:

```
int *ip3 = &i;
```

we're setting the initial value for ip3, which is where ip3 will point, so that initial value is a pointer. (In other words, the * in the declaration int *ip3 = &i; is not the contents-of operator, it's the indicator that ip3 is a pointer.) If you have a pointer declaration containing an initialization, and you ever have occasion to break it up into a simple declaration and a conventional assignment, do it like this:

int *ip3;
ip3 = &i;

Don't write
int *ip3;
*ip3 = &i;

or you'll be trying to mix pointer and the value to which it points Also, when we write
int *ip;

although the asterisk affects ip's type, it goes with the identifier name ip, not with the type int on the left. To declare two pointers at once, the declaration looks like
int *ip1, *ip2;
Some people write pointer declarations like this:
int* ip;
This works for one pointer, because C essentially ignores whitespace. But if you ever write
int* ip1, ip2; /* PROBABLY WRONG */

it will declare one pointer-to-int ip1 and one *plain* int ip2, which is probably not what you meant.
What is all of this good for? If it was just for changing variables like i from 5 to 7, it would not be good for much. What it's good for, among other things, is when for various reasons we don't know exactly which variable we want to change.
Program 8.1: A simple program to illustrate the relationship between two integer variables, their corresponding addresses and their associated pointers.
#include<stdio.h>
main()

```
{
int x=5;
int y;
int *px; /* pointer to an integer */
int *py; /* pointer to an integer */
px=&x; /* assign address of x to px */
y=*px; /* assign value of x to y */
py=&y; /* assign address of y to py */
printf("\nx=%d &x=%u px=%u *px=%d", x, &x, px, *px);
printf("\ny=%d &y=%u py=%u *py=%d", y, &y, py, *py);
}
```

Execute this program and observe the result.

Self Assessment Questions

1.Dereference operator is also known as _____.

2.Pointer is a variable containing address of another variable. (True/False)

3.State whether the following statements are correct: (Correct/Incorrect) int a, b;

b=&a;

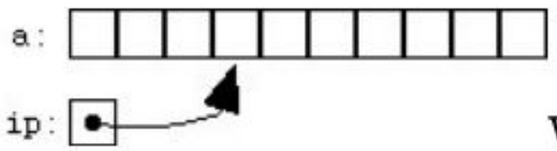
4. The contents-of operator * does not merely fetch values through pointers; it can also *set* values through pointers. (True/False)

8.3 Pointers and One-dimensional Arrays

Pointers do not have to point to single variables. They can also point at the cells of an array. For example, we can write

```
int *ip;
int a[10];
ip = &a[3];
```

and we would end up with ip pointing at the fourth cell of the array a (remember, arrays are 0-based, so a[0] is the first location). We could illustrate the situation like this:



We'd use this ip just like the one in the previous section: *ip gives us what ip points to, which in this case will be the value in a[3].

8.3.1 Pointer Arithmetic

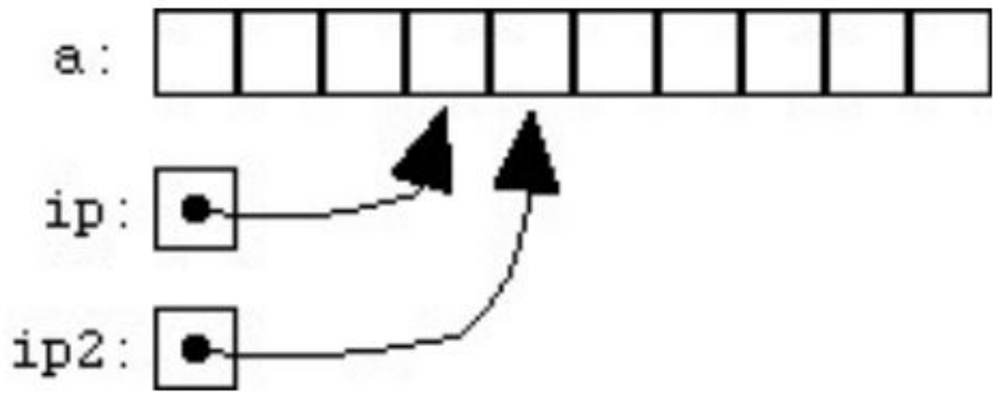
Once we have a pointer pointing into an array, we can start doing *pointer arithmetic*. Given that ip is a pointer to a[3], we can add 1 to ip:

```
ip + 1
```

What does it mean to add one to a pointer? In C, it gives a pointer to the cell one farther on, which in this case is a[4]. To make this clear, let's assign this new pointer to another pointer variable:

```
ip2 = ip + 1;
```

Now the picture looks like this:



If we now do
*ip2 = 4;

we've set a[4] to 4. But it's not necessary to assign a new pointer value to a pointer variable in order to use it; we could also compute a new pointer value and use it immediately:

*(ip + 1) = 5;

In this last example, we've changed a[4] again, setting it to 5. The parentheses are needed because the unary “contents of" operator * has higher *precedence* (i.e., binds more tightly than) the addition operator. If we wrote *ip + 1, without the parentheses, we'd be fetching the value pointed to by ip, and adding 1 to that value. The expression *(ip + 1), on the other hand, accesses the value one past the one pointed to by ip.
Given that we can add 1 to a pointer, it's not surprising that we can add and subtract other numbers as well. If ip still points to a[3], then
*(ip + 3) = 7;

sets a[6] to 7, and
*(ip - 2) = 4;
sets a[1] to 4.

Up above, we added 1 to ip and assigned the new pointer to ip2, but there's no reason we can't add one to a pointer, and change the same pointer: ip = ip + 1;

Now ip points one past where it used to (to a[4], if we hadn't changed it in the meantime). The shortcuts work for pointers, too: we could also increment a pointer using

```
ip += 1;
```

or

```
ip++;
```

Of course, pointers are not limited to ints. It's quite common to use pointers to other types, especially char.
Example 8.1: Here is a program segment to compare two strings, character by character using pointers.

```
char *p1 = &str1[0], *p2 = &str2[0];
while(1)
{
if(*p1 != *p2)
return *p1 - *p2;
if(*p1 == '\0' || *p2 == '\0')
return 0;
p1++;
p2++;
}
```

The auto increment operator ++ (like its companion, --) makes it easy to do two things at once. We've seen idioms like a[i++] which accesses a[i] and simultaneously increments i, leaving it referencing the next cell of the array a.We

can do the same thing with pointers: an expression like *ip++ lets us access what ip points to, while simultaneously incrementing ip so that it points to the next element. The preincrement form works, too: *++ip increments ip, then accesses what it points to. Similarly, we can use notations like *ip-- and *--ip.

Example 8.2: Here is a program segment to copy a string to another using pointer:

```
char *dp = &dest[0], *sp = &src[0];
while(*sp != '\0')
*dp++ = *sp++;
*dp = '\0';
```

(One question that comes up is whether the expression *p++ increments p or what it points to. The answer is that it increments p. To increment what p points to, you can use (*p)++.)

When you're doing pointer arithmetic, you have to remember how big the array the pointer points into is, so that you don't ever point outside it. If the array a has 10 elements, you can't access a[50] or a[-1] or even a[10] (remember, the valid subscripts for a 10-element array run from 0 to 9). Similarly, if a has 10 elements and ip points to a[3], you can't compute or access ip + 10 or ip - 5. (There is one special case: you can, in this case, compute, but not access, a pointer to the nonexistent element just beyond the end of the array, which in this case is &a[10]. This becomes useful when you're doing pointer comparisons, which we'll look at next.)

Program 8.2: Program to illustrate the relationship between an array and pointer.

```
#include<stdio.h>
main()
{
int a[10];
int i;
for(i=0;i<10;i++)
```

scanf(“%d”,&a[i]);
for(i=0;i<10;i++)
printf(“\ni=%d a[i]=%d *(a+i)=%d &a[i]=%u a+i=%u”, i, a[i], *(a+i), &a[i], a+i); }
Execute this program and observe the result.

8.3.2 Pointer Subtraction and Comparison

As we've seen, you can add an integer to a pointer to get a new pointer, pointing somewhere beyond the original (as long as it is in the same array). For example, you might write
ip2 = ip1 + 3;

Applying a little algebra, you might wonder whether
ip2 - ip1 = 3

and the answer is, yes. When you subtract two pointers, as long as they point into the same array, the result is the number of elements separating them. You can also ask (again, as long as they point into the same array) whether one pointer is greater or less than another: one pointer is “greater than” another if it points beyond where the other one points. You can also compare pointers for equality and inequality: two pointers are equal if they point to the same variable or to the same cell in an array, and are (obviously) unequal if they don't. (When testing for equality or inequality, the two pointers do not have to point into the same array.)
One common use of pointer comparisons is when copying arrays using pointers.

Example 8.3: Here is a code fragment which copies 10 elements from array1 to array2, using pointers. It uses an end pointer, endptr, to keep track of when it should stop copying.
int array1[10], array2[10];
int *ip1, *ip2 = &array2[0];

```
int *endptr = &array1[10];
for(ip1 = &array1[0]; ip1 < endptr; ip1++)
*ip2++ = *ip1;
```

As we mentioned, there is no element array1[10], but it is legal to compute a pointer to this (nonexistent) element, as long as we only use it in pointer comparisons like this (that is, as long as we never try to fetch or store the value that it points to.)

Prorgam 8.3: In the following program, two different pointer variables point to the first and the last element of an integer array.

```
#include<stdio.h>
main()
{
int *px, *py;
int a[5]={1, 2, 3, 4, 5};
px=&a[0];
py=&a[4];
printf("px=%u py=%u", px, py);
printf("\n\n py-px=%u", py-px);
}
```

Execute this program and observe the result.

8.3.3 Similarities between Pointers and One-dimensional Arrays There are a number of similarities between arrays and pointers in C. If you have an array

```
int a[10];
```

you can refer to a[0], a[1], a[2], etc., or to a[i] where i is an int. If you declare a pointer variable ip and set it to point to the

beginning of an array: int *ip = &a[0];
you can refer to *ip, *(ip+1), *(ip+2), etc., or to *(ip+i) where i is an int.

There are also differences, of course. You cannot assign two arrays; the code
int a[10], b[10];
a = b; /* WRONG */
is illegal. As we've seen, though, you *can* assign two pointer variables: int *ip1, *ip2;
ip1 = &a[0];
ip2 = ip1;
Pointer assignment is straightforward; the pointer on the left is simply made to point wherever the pointer on the right does. We haven't copied the data pointed to (there's still just one copy, in the same place); we've just made two pointers point to that one place.
The similarities between arrays and pointers end up being quite useful, and in fact C builds on the similarities, leading to what is called “the equivalence of arrays and pointers in C." When we speak of this “equivalence" we do not mean that arrays and pointers are the same thing (they are in fact quite different), but rather that they can be used in related ways, and that certain operations may be used between them.
The first such operation is that it is possible to (apparently) assign an array to a pointer:
int a[10];
int *ip;
ip = a;

What can this mean? In that last assignment ip = a, C defines the result of this assignment to be that ip receives a pointer to the first element of a. In other words, it is as if you had written

ip = &a[0];
The second facet of the equivalence is that you can use the “array subscripting" notation [i] on pointers, too. If you write
ip[3]
it is just as if you had written
*(ip + 3)

So when you have a pointer that points to a block of memory, such as an array or a part of an array, you can treat that pointer “as if" it *were* an array, using the convenient [i] notation. In other words, at the beginning of this section when we talked about *ip, *(ip+1), *(ip+2), and *(ip+i), we could have written ip[0], ip[1], ip[2], and ip[i]. As we'll see, this can be quite useful (or at least convenient).

The third facet of the equivalence (which is actually a more general version of the first one we mentioned) is that *whenever* you mention the name of an array in a context where the “value" of the array would be needed, C automatically generates a pointer to the first element of the array, as if you had written &array[0]. When you write something like
int a[10];
int *ip;
ip = a + 3;
it is as if you had written
ip = &a[0] + 3;
which (and you might like to convince yourself of this) gives the same result as if you had written
ip = &a[3];

For example, if the character array
char string[100];

contains some string, here is another way to find its length:

```
int len;
char *p;
for(p = string; *p != '\0'; p++)
;
len = p - string;
```

After the loop, p points to the '\0' terminating the string. The expression p string is equivalent to p - &string[0], and gives the length of the string. (Of course, we could also call strlen; in fact here we've essentially written another implementation of strlen.)

Self Assessment Questions

5.You can perform any type of arithmetic operation on pointers. (True or False)

6.For an int array a[10], If you declare a pointer variable ip then you can set it to point to the beginning of an array by assigning: int *ip = _____ .

7.One common use of pointer comparisons is when copying arrays using pointers. (True/False)

8.To increment what p points to, you can use the expression _____.

8.4 Summary

A pointer is a variable that points at, or refers to, another variable. The general format of a simple pointer declaration is: datatype *variablename. We discover the value pointed to by a pointer using the “contents-of” operator, *. Pointers are very useful when you want to refer to some other variable by pointing at it. Pointers not only point to single variables

but can also point at the cells of an array. Pointers are also closely associated with the arrays and therefore provide an alternate way to access individual array elements. Once we have a pointer pointing into an array, we can do some pointer arithmeticadd a constant to a variable or subtract a constant from a pointer, or you can subtract two pointers. There are similarities between
pointers and one dimension arrays in some aspects although they are entirely different concepts and there are differences also in other aspects.

8.5 Terminal Questions

1.How do you declare a pointer to a floating point quantity and a double precision quantity? Illustrate it.

2.What is the significance of the following declaration? Explain with an example.
int *p(int a)

3.What is the significance of the following declaration? Explain with an example.
int (*p) (int a)

4.Illustrate with an example how you use pointer arithmetic.

8.6 Answers to Self Assessment Questions

1 indirection
. operator. True
2 Incorrect

. True

3 False

. &a[0];

4 True

. (*p)++.)

5

.8.7 Answers to Terminal Questions

6 1.We declare a pointer to a floating point quantity and double precision ·quantity as follows:

7float *fptr;

·double *dptr;

8(Refer section 8. 2 and 8.3 for more details)

·2.int *p(int a) defines the function that accepts an integer argument and returns a pointer to an integer. (Refer section 8. 2 and 8.3 for more details)

3.int (*p) (int a) indicates a pointer to a function that accepts an integer argument and returns an integer. (Refer section 8. 2 and 8.3 for more details)

4.You can apply addition, subtraction or comparison with pointers. (Refer section 8. 2 and 8.3 for more details)

8.8 Exercise

1.Write a program that uses the pointer arithmetic.

2.Write a program that can accept any 5 data and sort it in ascending order using pointers.

3.With the help of pointers, write a program that uses functions to swap the data.

4.What are the various operators that are used when implementing a pointer in a program? Explain with an example.

Unit 9 Pointers – II

Structure:

9.1 Introduction

Objectives

9.2 Null pointers

9.3 Pointers as Function Arguments

9.4 Pointers and Strings

9.5 Pointers and two-dimensional arrays

Arrays of Pointers

9.6 Summary

9.7 Terminal Questions

9.8 Answers to Self Assessment Questions

9.9 Answers for Terminal Questions

9.10 Exercises

9.1 Introduction

In the previous unit, you read about the basics of pointers. A pointer is a variable that points at, or refers to, another

variable. Pointers are very useful when you want to refer to some other variable by pointing at it. Pointers not only point to single variables but can also point at the cells of an array. Pointers provide a convenient way to represent multidimensional arrays, allowing a single multidimensional array to be replaced by a lower- dimensional array of pointers. This feature permits a collection of strings to be represented within a single array, even though the individual strings may differ in length.

In this unit, you will read about null pointers and how pointers can be used to manipulate a string along with the usage of pointers in two dimensional arrays. You will also read about arrays of pointers briefly.

Objectives:
After studying this unit, you should be able to:
explain a null pointer
use pointers as function arguments ?
manipulate strings by the help of pointers
explain arrays of pointers

9.2 Null Pointers

We said that the value of a pointer variable is a pointer to some other variable. There is one other value a pointer may have: it may be set to a *null pointer*. A *null pointer* is a special pointer value that is known not to point anywhere. What this means that no other valid pointer, to any other variable or array cell or anything else, will ever compare equal to a null pointer.

The most straightforward way to “get" a null pointer in your program is by using the predefined constant NULL, which is defined for you by several standard header files, including <stdio.h>, <stdlib.h>, and <string.h>. To initialize a pointer to a null pointer, you might use code like

```
#include <stdio.h>
int *ip = NULL;
```

and to test it for a null pointer before inspecting the value pointed to, you might use code like

```
if(ip != NULL)

printf("%d\n", *ip);
```

It is also possible to refer to the null pointer by using a constant 0, and you will see some code that sets null pointers by simply doing

```
int *ip = 0;
```

(In fact, NULL is a preprocessor macro which typically has the value, or replacement text, 0.)

Furthermore, since the definition of “true" in C is a value that is not equal to 0, you will see code that tests for non-null pointers with abbreviated code like

```
if(ip)
printf("%d\n", *ip);
```

This has the same meaning as our previous example; if(ip) is equivalent to if(ip != 0) and to if(ip != NULL).

All of these uses are legal, and although the use of the constant NULL is recommended for clarity, you will come across the other forms, so you should be able to recognize them.
You can use a null pointer as a placeholder to remind yourself (or, more importantly, to help your program remember) that a pointer variable does not point anywhere at the moment and that you should not use the “contents of" operator on it (that is, you should not try to inspect what it points to, since it doesn't point to anything). A function that returns pointer values can return a null pointer when it is unable to perform its task. (A null pointer used in this way is analogous to the EOF value that functions like getchar return.)

As an example, let us write our own version of the standard library function strstr, which looks for one string within another, returning a pointer to the string if it can, or a null pointer if it cannot.
Example 9.1: Here is the function, using the obvious brute-force algorithm: at every character of the input string, the code checks for a match there of the pattern string:

```
#include <stddef.h>

char *mystrstr(char input[], char pat[])
{
char *start, *p1, *p2;
for(start = &input[0]; *start != '\0'; start++)
{ /* for each position in input string... */
p1 = pat; /* prepare to check for pattern string there */ p2 = start;
while(*p1 != '\0')
{
if(*p1 != *p2) /* characters differ */
break;
p1++;
```

```
p2++;
}
if(*p1 == '\0') /* found match */
return start;
}
}return NULL;
```

The start pointer steps over each character position in the input string. At each character, the inner loop checks for a match there, by using p1 to step over the pattern string (pat), and p2 to step over the input string (starting at start). We compare successive characters until either (a) we reach the end of the pattern string (*p1 == '\0'), or (b) we find two characters which differ. When we're done with the inner loop, if we reached the end of the pattern string (*p1 == '\0'), it means that all preceding characters matched, and we found a complete match for the pattern starting at start, so we return start. Otherwise, we go around the outer loop again, to try another starting position. If we run out of those (if *start == '\0'), without finding a match, we return a null pointer.

Notice that the function is declared as returning (and does in fact return) a pointer-to-char.

In general, C does not initialize pointers to null for you, and it never tests pointers to see if they are null before using them. If one of the pointers in your programs points somewhere some of the time but not all of the time, an excellent convention to use is to set it to a null pointer when it doesn't point anywhere valid, and to test to see if it's a null pointer before using it. But you must use explicit code to set it to NULL, and to test it against NULL. (In other words, just setting an unused pointer variable to NULL doesn't guarantee safety; you also have to check for the null value before using the pointer.) On the other hand, if you know that a particular pointer variable is always valid, you don't have to insert a paranoid test against NULL before using it.

Self Assessment Questions

1.A _____ is a special pointer value that is known not to point anywhere. 2.A function that returns _____ values

can return a null pointer when it is unable to perform its task.
3.In general, C does not initialize pointers to null for you, and it never tests pointers to see if they are null before using them. (True/False)

9.3 Pointers as Function Arguments

Earlier, we learned that functions in C receive copies of their arguments. (This means that C uses *call by value*; it means that a function can modify its arguments without modifying the value in the caller.)
Consider the following function to swap two integers

```
void swap(int x, int y)
{
int temp;
temp=x;
x=y;
y=temp;
return;
}
```

The problem with this function is that since C uses *call by value* technique of parameter passing, the caller can not get the changes done in the function. This can be achieved using pointers as illustrated in the following program.

Program 9.2: Program to swap two integers using pointers

```
#include<stdio.h>
main()
{
```

```
int a, b;
void swap(int *a, int *b);
printf(“ Read the integers:”);

scanf(“%d%d”, &a, &b);
swap(&a, &b); /* call by reference or call by address*/
printf(“ \nAfter swapping:a=%d b=%d”, a, b);
}
void swap(int *x, int *y)
{
int temp;
temp=*x;
*x=*y;
*y=temp;
return;
}
```

Execute this program and observe the result.
Because of the use of call by reference, the changes made in the function swap() are also available in the main().

9.4 Pointers and Strings

Because of the similarity of arrays and pointers, it is extremely common to refer to and manipulate strings as character pointers, or char *'s. It is so common, in fact, that it is easy to forget that strings are arrays, and to imagine that they're

represented by pointers. (Actually, in the case of strings, it may not even matter that much if the distinction gets a little blurred; there's certainly nothing wrong with referring to a character pointer, suitably initialized, as a "string.") Let's look at a few of the implications:

Any function that manipulates a string will actually accept it as a char * argument. The caller may pass an array containing a string, but the function will receive a pointer to the array's (string's) first element (character).

The %s format in printf expects a character pointer.

Although you have to use strcpy to copy a string from one array to another, you can use simple pointer assignment to assign a string to a pointer. The string being assigned might either be in an array or pointed to by another pointer. In other words, given

```
char string[] = "Hello, world!";
char *p1, *p2;
```

both

```
p1 = string
```

and

```
p2 = p1
```

are legal. (Remember, though, that when you assign a pointer, you're making a copy of the pointer but *not* of the data it points to. In the first example, p1 ends up pointing to the string in string. In the second example, p2 ends up pointing to the same string as p1. In any case, after a pointer assignment, if you ever change the string (or other data) pointed to, the change is "visible" to both pointers.

Many programs manipulate strings exclusively using character pointers, never explicitly declaring any actual arrays. As long as these programs are careful to allocate appropriate memory for the strings, they're perfectly valid and correct.

When you start working heavily with strings, however, you have to be aware of one subtle fact.
When you initialize a character array with a string constant:

```
char string[] = "Hello, world!";
```

you end up with an array containing the string, and you can modify the array's contents to your heart's content:
`string[0] = 'J';`

However, it's possible to use string constants (the formal term is *string literals*) at other places in your code. Since they're arrays, the compiler generates pointers to their first elements when they're used in expressions, as usual. That is, if you say

```
char *p1 = "Hello";
int len = strlen("world");
```

it's almost as if you'd said

```
char internal_string_1[] = "Hello";
char internal_string_2[] = "world";

char *p1 = &internal_string_1[0];
int len = strlen(&internal_string_2[0]);
```

Here, the arrays named internal_string_1 and internal_string_2 are supposed to suggest the fact that the compiler is actually generating little temporary arrays every time you use a string constant in your code. *However*, the subtle fact is that the arrays which are “behind" the string constants are *not* necessarily modifiable. In particular, the compiler may

store them in read-only-memory. Therefore, if you write

```
char *p3 = "Hello, world!";

p3[0] = 'J';
```

your program may crash, because it may try to store a value (in this case, the character 'J') into nonwritable memory. The moral is that whenever you're building or modifying strings, you have to make sure that the memory you're building or modifying them in is writable. That memory should either be an array you've allocated, or some memory which you've dynamically allocated. Make sure that no part of your program will ever try to modify a string which is actually one of the unnamed, unwritable arrays which the compiler generated for you in response to one of your string constants. (The only exception is array initialization, because if you write to such an array, you're writing to the array, not to the string literal which you used to initialize the array.)

Example 9.3: Breaking a Line into “Words"

First, break lines into a series of whitespace-separated words. To do this, we will use an *array of pointers to* char, which we can also think of as an “array of strings," since a string is an array of char, and a pointer-to-char can easily point at a string. Here is the declaration of such an array:

```
char *words[10];
```

This is the first complicated C declaration we've seen: it says that words is an array of 10 pointers to char. We're going to write a function, getwords, which we can call like this:

```
int nwords;
nwords = getwords(line, words, 10);
```

where line is the line we're breaking into words, words is the array to be filled in with the (pointers to the) words, and nwords (the return value from getwords) is the number of words which the function finds. (As with getline, we tell the function the size of the array so that if the line should happen to contain more words than that, it won't overflow the array).

Here is the definition of the getwords function. It finds the beginning of each word, places a pointer to it in the array, finds the end of that word (which is signified by at least one whitespace character) and terminates the word by placing a '\0' character after it. (The '\0' character will overwrite the first whitespace character following the word.) Note that the original input string is therefore modified by getwords: if you were to try to print the input line after calling getwords, it would appear to contain only its first word (because of the first inserted '\0').

```
#include <stddef.h>
#include <ctype.h>
getwords(char *line, char *words[], int maxwords)
{
char *p = line;
int nwords = 0;

while(1)
{
while(isspace(*p)) p++;

if(*p == '\0')
return nwords;
words[nwords++] = p;
```

```
while(!isspace(*p) && *p != '\0')
p++;
if(*p == '\0')
return nwords;
*p++ = '\0';

if(nwords >= maxwords)
return nwords;
}
}
```

Each time through the outer while loop, the function tries to find another word. First it skips over whitespace (which might be leading spaces on the line, or the space(s) separating this word from the previous one). The isspace function is new: it's in the standard library, declared in the header file <ctype.h>, and it returns nonzero ("true") if the character you hand it is a space character (a space or a tab, or any other whitespace character there might happen to be).

When the function finds a non-whitespace character, it has found the beginning of another word, so it places the pointer to that character in the next cell of the words array. Then it steps through the word, looking at nonwhitespace characters, until it finds another whitespace character, or the \0 at the end of the line. If it finds the \0, it's done with the entire line; otherwise, it changes the whitespace character to a \0, to terminate the word it's just found, and continues. (If it's found as many words as will fit in the words array, it returns prematurely.)

Each time it finds a word, the function increments the number of words (nwords) it has found. Since arrays in C start at [0], the number of words the function has found so far is also the index of the cell in the words array where the next word should be stored. The function actually assigns the next word and increments nwords in one expression:

words[nwords++] = p;

You should convince yourself that this arrangement works, and that (in this case) the preincrement form
words[++nwords] = p; /* WRONG */

would *not* behave as desired.

When the function is done (when it finds the \0 terminating the input line, or when it runs out of cells in the words array) it returns the number of words it has found.

Here is a complete example of calling getwords:

```
char line[] = "this is a test";
int i;
nwords = getwords(line, words, 10);
for(i = 0; i < nwords; i++)
printf("%s\n", words[i]);
```

Self Assessment Questions

4.An array of characters is called as _____.

5.You can represent an array of strings using pointers by using an array of _____ to character.

6.A string must always be terminated with a _____ character.

9.5 Pointers and two-dimensional arrays

Since a one-dimensional array can be represented in terms of pointer (the array name) and an offset (the subscript), it is reasonable to expect that a two-dimensional array can also be represented with an equivalent pointer notation. A two-dimensional array is actually a collection of one-dimensional arrays. Therefore we can define a two dimensional array as a pointer to a group of contiguous one-dimensional arrays. A two-dimensional array declaration can be written as:

*data-type (*ptr)[expression]*

where *data-type* is the type of array elements and *expression* is the positivevalued integer expression that indicates the number of columns in each row of the two-dimensional array.

Example 9.5: Suppose that x is a two-dimensional integer array having 10 rows and 20 columns. We can declare x as int (*x)[20];

In this case, x is defined to be a pointer to a group of contiguous, onedimensional, 20-element integer arrays. Thus x points to the first 20-element array, which is actually the first row(row 0) of the original two-dimensional array. Similarly, (x+1) points to the second 20-element array, which is the second row(row 1) of the original two-dimensional array and so on.

Arrays of pointers

A two-dimensional array can also be expressed in terms of an array of pointers rather than as a pointer to a group of contiguous arrays. Each element in the array is a pointer to a separate row in the two-dimensional array. In general

terms, a two-dimensional array can be defined as a onedimensional array of pointers by writing:
*data-type *ptr[expression]*

where *data-type* is the type of array elements and *expression* is the positivevalued integer expression that indicates the number of rows.

Example 9.6: Suppose that x is a two-dimensional array having 10 rows and 20 columns, we can define x as a one-dimensional array of pointers by writing
int *x[10];
Hence, x[0] points to the beginning of the first row, x[1] points to the beginning of the second row and so on.
An individual array element, such as x[2][4] can be accessed

by writing
*(x[2]+4)

In this expression, x[2] is a pointer to the first element in row 2, so that (x[2]+5) points to element 4(actually, the fifth element) within row 2. The element of this pointer, *(x[2]+4), therefore refers to x[2][4].

Self Assessment Questions
7.We can define a two dimensional array as a pointer to a group of
contiguous _____ dimensional arrays.
8.A two-dimensional array can also be expressed in terms of an _____ of pointers rather than as a pointer to a group of contiguous
arrays.

9.6 Summary

A pointer is a variable that represents the location of a data item, such as a variable or an array element. A pointer may be set to a null pointer. A null pointer is a special pointer value that is known not to point anywhere. Passing pointers as arguments to functions is called pass by reference. Because of the similarity of arrays and pointers, it is extremely common to refer to and manipulate strings as character pointers. We can define a two dimensional array as a pointer to a group of contiguous one-dimensional arrays. A two-dimensional array can also be expressed in terms of an array of pointers rather than as a pointer to a group of contiguous arrays.

9.7 Terminal Questions

1.Explain what a null pointer is. Illustrate it with an example.
2.How can you use pointers as function arguments? Illustrate it with an example.
3.Distinguish between pass by value and pass by reference with the help of an example.
4.How can you manipulate strings using character pointers? Illustrate it with the help of an example.

9.8 Answers to Self Assessment Questions

1 null pointer
. pointer
2 True

. string
3 pointers
. null
4 one
. array
5

9.9 Answers to Terminal Questions

6 1.A null pointer is a special pointer value that is known not to point . anywhere and is implemented by using the predefined constant NULL,
7 which is defined by several standard header files, including <stdio.h>, <stdlib.h>, and <string.h>. (Refer to section 9.2 for more details)
[8]2.(Refer to section 9.2 for more details)
· 3.In pass by value technique, the actual value is passed to the function whereas in pass by reference, the address is passed to the function.

4.You can use the pointer to the string considering string as an array and use (string's) first element i.e. character. The %s format in printf expects
a character pointer. (Refer to section 9.4 and 9.5 for more details)

9.10 Exercises

1.Write a program to find the number of characters in a string using pointers.
2.Write a program to multiply two matrices using pointers.
3.Suppose a formal argument within a function definition is a pointer to another function. How is the formal argument declared? Within the formal argument declaration, to what does the data type refer?
4.Write a program to concatenate two strings.
5.Write a function to sort an array of numbers using pointers.

Unit 10 Structures and Unions

Structure:

10.1 Introduction
Objectives
10.2 Basics of Structures
10.3 Structures and Functions
10.4 Arrays of Structures
10.5 Pointers to Structures
10.6 Self-referential Structures
10.7 Unions
10.8 Summary

10.9 Terminal Questions
10.10 Answers to Self Assessment Questions
10.11 Answers to Terminal Questions
10.12 Exercises

10.1 Introduction

In the previous units, you studied about the pointers as a most powerful tool in C programming language. It is due to the pointer only that makes C as the most beautiful language. In this unit, you will study another useful ways of making the program easy and efficient. This unit will enable you to learn about the structures and unions.

As we know an array is a data structure whose elements are all of the same data type. We now turn our attention to the structure, which is a data structure whose individual elements can differ in type. Thus, a single structure might contain integer elements, floating-point elements and character elements. Pointers, arrays and other structures can also be included as elements within a structure.
This unit is concerned with the use of structures within a C program. We will see how structures are defined, and how their individual members are accessed and processed within a program.

Closely associated with the structure is the union, which also contains multiple members. Unlike a structure, however, the members of a union share the same storage area, even though the individual members may differ in type.

Objectives:
After studying this unit, you should be able to:

handle a group of logically related data items known as structures. declare an array of structures, each element of the array representing a

structure variable.
pass structure as an argument to functions and return structure from
functions.
refer to (i.e., point to) an incomplete type, including itself.
handle a group of logically related data items in terms of unions.

10.2 Basics of Structures

C supports a constructed data type known as structure, which is a method for packing data of different types. A structure is a convenient tool for handling a group of logically related data items. Structures help to organize complex data in a more meaningful way. It is a powerful concept that we may often need to use in our program design.

Structure Definition: A Structure definition creates a format that may be used to declare structure variables. For e.g., Consider a book database consisting of book name, author, number of pages and price.

```
struct book_bank
{
char title[20];
char author[15];
int pages;
```

float price;
};

The keyword **struct** declares a structure to hold the details of four fields, namely title, author, pages and price. These fields are called structure elements or members. Each member may belong to a different type of data. book_bank is the name of the structure and is called the structure tag. The tag name may be used subsequently to declare variables that have the tag's structure. Note that the above declaration has not declared any variables. It simply describes a format called template to represent information as shown below:

struct book_bank
array of 20 characters **title**
author array of 15 characters
pages integer
price float

We can declare structure variables using the tag name anywhere in the program. e.g, the statement:
struct book_bank book1, book2, book3;
declares book1, book2 and book3 as variables of type **book_bank.** Each one of these variables has four members as specified by the template. The complete declaration might look like this:
struct book_bank
{
char title[20];
char author[15];
int pages;

float price;
};

struct book_bank book1, book2, book3;

It is also allowed to combine both the template declaration and variables declaration in one statement.

struct book_bank
{
char title[20];
char author[15];
int pages;
float price;
} book1, book2, book3;

General format of a Structure Definition:

The general format of a structure definition is as follows:

struct tag_name
{
data_type member1;
data_type member2;

};

In defining a structure you may note the following syntax:

1.The template is terminated with a semicolon.

2.While the entire declaration is considered as a statement, each member

is declared independently for its name and type in a separate statement inside the template.
3.The tag name such as tag_name can be used to declare structure variables of its type, later in the program.

Giving values to Members:
Structure members need to be linked to the structure variables in order to make them meaningful members. The link between a member and a variable is established using the member operator '.' which is also known as 'dot operator' or 'period operator'.

Here is how we would assign values to the members of book1. **strcpy**(book1.title,"BASIC");
strcpy(book1.author,"Balagurusamy");
book1.pages = 250;
book1.price = 28.50;
We can also give the values through the keyboard.
gets(book1.title);
gets(book1.author);
printf("%d",book1.pages);
printf("%f",book1.price);

Structure Initialization: void main( )

{
struct st_record
{
char name[20];
int weight;

float height;
};

static struct st_record student1 = {“Suresh”, 60, 180.75}; **static struct** st_record student2 = {“Umesh”, 53, 170.60}; }

Program 10.1: To print the date of joining of a person

```
#include<conio.h>
#include<stdio.h>
struct personal
{
char name[30];
int day;
int month;
int year;
float salary;
};
void main()
{
struct personal p;
printf(“Enter the name:\n)";
gets(p.name);
printf(“Enter the day of joining:\n)";
scanf(“%d”,&p.day);
printf(“Enter the month of joining:\n");
scanf(“%d”,&p.month);
```

```
printf(“Enter the year of joining:\n)";
scanf(“%d”,&p.year);
printf(“Enter the salary:\n)";
scanf(“%f”, & p.salary);
printf(“\nName:",p.name);
printf("\nDate of joining:%d %d %d",p.day,p.month,p.year);
printf(“Salary:",p.salary);
getch();
}
```

Comparison of structure variables

Two variables of same structure type can be compared in the same way as ordinary variables. If **person1** and **person2** belong to the same structure, then the following operations are valid:

Operation Meaning
person1 = person2 person1 == person2

person1 != person2 Assign **person2** to **person1**
Compare all members of **person1** and **person2** and return 1 if they are equal, 0 otherwise.
Return 1 if all the members are not equal, 0 otherwise

Program 10.2: To Compare structure variables

```
#include <stdio.h>
#include<conio.h>
struct stclass{
```

```
int number;
char name[20];
float marks;
};

void main() {
int x;
static struct stclass student1 = {111,"Rao",72.50};
static struct stclass student2 = {222,"Reddy",67.00};
struct stclass student3;
student3 = student2;

x=((student3.number == student2.number) && (student3.marks == student2.marks))? 1:0;
if(x==1)
{
printf("\nStudent2 and Student3 are same ");

printf(" %d\t %s\t %f\t",student3.number,student3.name,student3.marks); }
else
{
printf("\nStudent2 and student3 are different)";
}
getch();
}
```

Self Assessment Questions

1.A _____ is a convenient tool for handling a group of logically
related data items.

2.We can declare structure variables using the tag name anywhere in the
program. (True/False)

3._____ is a method for packing data of different types. 4.If person1 and person2 are variables of the same type structure then
the expression person1>person2 is valid. (True/False)

5._____ is a method for packing data of different types. 6.The link between a member and a variable is established using the
member operator _____.

10.3 Structures and Functions

We can write programs with structures by using modular programming. We can write a function that returns the structure. While writing the function, you should indicate the type of structure that is returned by the function. The **return** statement should return the structure using a variable. It is

possible

to pass a structure as an argument. We can modify a member of the structure by passing the structure as an argument. The changes in the member made by the function are retained in the called module. This is not against the principle of call by value because we are not modifying the structure variable, but are instead modifying the members of the structure.

Program 10.3: To illustrate the concept of structures and functions struct student

```
{
name char[30];
marks float;
};
main ( )
{
struct student student1;
student1 = read_student ( );
print_student( student1);
read_student_p(student1);
print_student (student1);
}
struct student read_student( )
{
struct student student2;
gets(student2.name);
scanf("%d",&student2.marks);
return (student2);
}
void print_student (struct student student2)
{
printf( "name is %s\n", student2.name);
printf( "marks are%d\n", student2.marks);
```

```
}
void read_student_p(struct student student2)
{
gets(student2.name);
scanf("%d",&student2.marks);
```

Explanation

1.The function read_student reads values in structures and returns the structure.

2.The function print_student takes the structure variable as input and prints the content in the structure.

3.The function read_student_p reads the data in the structure similarly to

read_student. It takes the structure **student** as an argument and puts the data in the structure. Since the data of a member of the structure is modified, you need not pass the structure as a pointer even though structure members are modified. Here you are not modifying the structure, but you are modifying the structure members through the structure.

Self Assessment Questions

7.We cannot write a function that returns the structure. (True/False) 8.We can modify a member of the structure by passing the structure as a _____.

10.4 Arrays of Structures

We can use structures to describe the format of a number of related variables. For example, in analyzing the marks obtained by a class of students, we may use a template to describe student name and marks obtained in various

subjects and then declare all the students as structure variables. In such cases, we may declare an array of structures, each element of the array representing a structure variable. e.g, **struct** stclass student[100]; defines an array called student, that consists of 100 elements. Each element is defined to be of the type struct stclass. Consider the following declaration:

```
struct marks
{
int subject1;
int subject2;
int subject3;
};

main( )
{
static struct marks student[3]={{45,68,81},{75,53,69},{57,36,71}};}
```

This declares the student as an array of three elements student[0], student[1] and student[2] and initializes their members as follows: student[0].subject1 = 45;
student[0].subject2 = 68;
……..
student[2].subject3 = 71;

Program 10.4: To process employee details using structures #include<conio.h>
#include<stdio.h>
struct employee
{

```
int empno;
char name[30];
int basic;
int hra;
};
void main()
{
int i,j,n,net[50];
float avg;
employee e[50];
printf("\nEnter the number of employees:");
scanf(“%d”, &n);
printf(“\nEnter Empno.\tName\tBasic\tHra of each employee:\n"); for(i=0;i<n;i++)
{
scanf(“%d”,&e[i].empno);
gets(e[i].name);
scanf(“%d”,&e[i].basic);
scanf(%d”,&e[i].hra);
net[i]= e[i].basic+e[i].hra;
avg=avg+net[i];
avg=avg/n;
printf("\nEmpno.\tName\tNetpay\n");
for(i=0;i<n;i++)
{
```

```
if(net[i]>avg)
{
printf(e[i].empno\t)";
printf(e[i].name\t)";
printf(net[i]\n");
}
}
getch();
}
```

Program 10.5: To process student details using structures

```
#include<conio.h>
#include<stdio.h>
struct student
{
int rollno;
char name[30];
int marks1;
int marks2;
int marks3;
};
void main()
{
int i,j,n,tot[50],t;
```

```
student s[50],temp;
printf("\nEnter the number of students:");
scanf(“%d”,&n);
printf("\nEnter Rollno.\tName\tMarks1\tMarks2\tMarks3 of each student:\n"); for(i=0;i<n;i++)
{
scanf(“%d”,&s[i].rollno);
gets(s[i].name);
scanf(“%d”,&s[i].marks1);
scanf(“%d”,&s[i].marks2);

scanf(“%d”,&s[i].marks3);
tot[i]= s[i].marks1+s[i].marks2+s[i].marks3;
}
for(i=0;i<n-1;i++)
{
for(j=i+1;j<n;j++)
{
if(tot[i]<tot[j])
{
temp=s[i];
s[i]=s[j];
s[j]=temp;
t=tot[i];
tot[i]=tot[j];
```

```
tot[j]=t;
}
}
}
printf("\nRollno.\tName\tTotal marks in decreasing order of total marks is:\n");
for(i=0;i<n;i++)
{
printf(“%d\t”,s[i].rollno);
printf(“%s\t”,s[i].name);
printf(“%d\t”,s[i].tot);
}
getch();
}
```

Self Assessment Questions

9.We can use structures to describe the format of a number of related variables. (True/False)

10.You can declare an array of structures where each element is defined to be of the type _____.

10.5 Pointers to Structures

Pass by value may be very inefficient if the structure is large (i.e., has many members). They have identical declaration syntax and member access, but they serve a very different purpose. Defining pointer types is the same as for variables

of primitive types.

Example:

```
struct Point {
int x;
int y;
};
struct Rectangle {
struct Point topleft;
struct Point bottomright;
};
struct Point pt = { 50, 50 };
struct Point *pp;
pp = &pt;
(*pp).x = 100; /* pt.x is now 100. */
```

Notice the parentheses around the de referenced pointer.

```
(*pp).x = 100;
```

This is necessary to enforce correct precedence.

An alternative notation permits simpler pointer access to structure members. (*pp).x = 100;

```
pp->x = 100; /* equivalent */
```

Another example,

```
struct Rectangle rect, *pr = &rect;
rect.topleft.x = 50; /* equivalent operations */
```

(*pr).topleft.x = 50;
pr->topleft.x = 50;

Self Assessment Questions

11.The parentheses around the de referenced pointer is necessary to enforce the correct _____.

12.An alternative notation other than dot, permits simpler pointer access to structure members is _____.

10.6 Self-referential Structures

The ability to refer to (ie, point to) an incomplete type, including itself, is an important property for constructing a variety of data-structures. For example: linked-lists, binary trees, graphs, hash tables, and more.

Linked lists come in two basic varieties: singly linked and doubly linked. We describe here a simple version of a singly linked list.

List consists of a set of nodes, where each node contains an item and a pointer to another list node.

```
struct List {
int item;
struct List *next;
};
```

More about linked lists, you will study in the units to come.

10.7 Unions

Unions look similar to structures. They have identical declaration syntax and member access, but they serve a very different purpose.

```
union Utype {
int ival;
float fval;
char *sval;
};

union Utype x, y, z;
```

Accessing members of a union is via "." member operator or, for pointers to unions, the -> operator.

A union holds the value of one-variable at a time. The compiler allocates storage for the *biggest* member of the union. The type retrieved from the union must be the type most recently stored. Otherwise, the result is implementation dependent.

```
union Utype x;
x.fval = 56.4; /* x holds type float. */
printf("%f\n", x.fval); /* OK. */
printf("%d\n", x.ival); /* Implementation dependent. */
```

Unions are used to store one of a set of different types. These are commonly used to implement a "variant" array. (This is a form of generic programming.) There are other uses also, but they are quite advanced (e.g., concern the alignment

properties of unions).

Self Assessment Questions

13.A _____ holds the value of one-variable at a time.

14.The compiler allocates storage for the *smallest* member of the union. (True/False)

10.8 Summary

A structure is a convenient tool for handling a group of logically related data items. Structure members need to be linked to the structure variables in order to make them meaningful members. We can write programs with structures by using modular programming. We can use structures to describe the format of a number of related variables. Passing a pointer to a structure is generally much more efficient than making a copy of the structure itself. The ability to refer to (i.e., point to) an incomplete type, including itself, is an important property for constructing a variety of data-structures.

Unions have identical declaration syntax and member access, but they serve a very different purpose. A union holds the value of one-variable at a time. The compiler allocates storage for the *biggest* member of the union.

10.9 Terminal Questions

1.Write the output that will be generated by the following C program: typedef struct

{

char *a;

char *b;

```
char *c;
} colors;
void main()
{
void fun( colors sample);
static colors sample = {“red”, “green”, “blue”};

printf( (“%s %s %s\n”, sample.a, sample.b, sample.c);
fun(sample);
printf( (“%s %s %s\n”, sample.a, sample.b,
sample.c);
}
```

```
void fun (colors sample)
{
strcpy (sample.a=”cyon”); strcpy (sample.b=”magenta”); strcpy (sample.c=”yellow”);

printf(“%s %s %s\n”, sample.a, sample.b, sample.c);
return;
}
```

2.Describe the output generated by the following program. Distinguish between meaningful and meaningless output.

```
#include <stdio.h>
main()
{
```

```
union {
int i;
float f;
double d;
} u;
printf("%d\n", sizeof(u));
u.i= 100;
printf("%d %f %f\n", u.i, u.f, u.d);
u.f=0.5;
printf("%d %f %f\n", u.i, u.f, u.d); u.d = 0.0166667;
printf("%d %f %f\n", u.i, u.f, u.d); }
```

10.10 Answers to Self Assessment Questions

1. structure
2. true
3. Array
4. false
5. Structure
6. dot(.)
7. true
8. argument.
9. True
10. struct

11. precedence
12. ->
13. union
14. false

10.11 Answers to Terminal Questions

1.red green blue
cyan magenta yellow
red blue green

2.8
100 0.000000 -0.000000
0 0.500000 -0.000000
-25098 391364288.000000 0.016667
The first line displays the size of the union (8 bytes, to accommodate double data). In the second line, only the first value (100) is meaningful. In the third line, only the second value (0.500000) is meaningful. In the last line, only the last value (0.016667) is meaningful.

10.12 Exercises

1.What is a structure? How does a structure differ from an array?

2.What is a member? What is the relationship between a member and a structure?

3.Describe what is wrong in the following structure declaration: struct

```
{
int number;
float price;
}
main()
{
.....
.....
}
```

4.Describe Array of structures with an example program.

5.Define a structure called **cricket** that will describe the following inf ormation:
(i)player name (ii) team name(iii)batting average

Using **cricket**, declare an array **player** with 50 elements and write a program to read the information about all the 50 players and print a team-wise list containing names of players and print a team-wise list containing names of players with their batting average.

6.How is a structure type pointer variable declared?

7.Write a program to find the number of characters in a string using pointers.

Unit 11 The Preprocessor

Structure:

11.1 Introduction

Objectives

11.2 File Inclusion

11.3 Macro Definition and Substitution

Macros with Arguments

Nesting of Macros

11.4 Conditional Compilation

11.5 Summary

11.6 Terminal Questions

11.7 Answers to Self Assessment Questions

11.8 Answers to Terminal Questions

11.9 Exercises

11.1 Introduction

In the previous unit, you studied about the structures and unions in C. You studied how those structures and unions are used to bind similar types of data structures. In this unit, you will study about a Preprocessor. Preprocessor is a unique feature of C language. The C preprocessor provides several tools that are unavailable in other high-level

languages. Conceptually, the “preprocessor" is a translation phase that is applied to your source code before the compiler proper gets its hands on it. (Once upon a time, the preprocessor was a separate program, much as the compiler and linker may still be separate programs today.) Generally, the preprocessor performs textual substitutions on your source code, in three sorts of ways:

File inclusion: inserting the contents of another file into your source file, as if you had typed it all in there.
Macro substitution: replacing instances of one piece of text with another.
Conditional compilation: Arranging that, depending on various circumstances, certain parts of your source code are seen or not seen by the compiler at all.

The next three sections will introduce these three preprocessing functions. The syntax of the preprocessor is different from the syntax of the rest of C in several respects. First of all, the preprocessor is “line based." Each of the preprocessor directives we're going to learn about (all of which begin with the # character) must begin at the beginning of a line, and each ends at the end of the line. (The rest of C treats line ends as just another whitespace character, and doesn't care how your program text is arranged into lines.) Secondly, the preprocessor does not know about the structure of C -- about functions, statements, or expressions. It is possible to play strange tricks with the preprocessor to turn something which does not look like C into C (or vice versa). It's also possible to run into problems when a preprocessor substitution does not do what you expected it to, because the preprocessor does not respect the structure of C statements and expressions (but you expected it to). For the simple uses of the preprocessor we'll be discussing, you shouldn't have any of these problems, but you'll want to be careful before doing anything tricky or outrageous with the preprocessor. (As it happens, playing tricky and outrageous games with the preprocessor is considered sporting in some circles, but it rapidly gets out of hand, and can lead to bewilderingly impenetrable programs.)

Objectives:
After studying this unit, you should be able to:

explain about a preprocessor
include Files in the source program
explain and implement Macro definition and substitution
use parentheses in macro definitions

11.2 File Inclusion

A line of the form
#include <filename.h>
or
#include "filename.h"

causes the contents of the file filename.h to be read, parsed, and compiled at that point. (After filename.h is processed, compilation continues on the line following the #include line.) For example, suppose you got tired of retyping external function prototypes such as
extern int getline(char [], int);
at the top of each source file. You could instead place the prototype in a header file, perhaps getline.h, and then simply place

#include "getline.h"

at the top of each source file where you called getline. (You might not find it worthwhile to create an entire header file for a single function, but if you had a package of several related function, it might be very useful to place all of their declarations in one header file.) As we may have mentioned, that's exactly what the Standard header files such as

stdio.h are collections of declarations (including external function prototype declarations) having to do with various sets of Standard library functions. When you use #include to read in a header file, you automatically get the prototypes and other declarations it contains, and you *should* use header files, precisely so that you will get the prototypes and other declarations they contain.

The difference between the <> and "" forms is where the preprocessor searches for filename.h. As a general rule, it searches for files enclosed in <> in central, standard directories, and it searches for files enclosed in "" in the “current directory," or the directory containing the source file that's doing the including. Therefore, "" is usually used for header files you've written, and <> is usually used for headers which are provided for you (which someone else has written).

The extension “.h", by the way, simply stands for “header," and reflects the fact that #include directives usually sit at the top (head) of your source files, and contain global declarations and definitions which you would otherwise put there. (That extension is not mandatory, you can theoretically name your own header files anything you wish, but .h is traditional, and recommended.)

As we've already begun to see, the reason for putting something in a header file, and then using #include to pull that header file into several different source files, is when the something (whatever it is) must be declared or defined consistently in all of the source files. If, instead of using a header file, you typed the something in to each of the source files directly, and the something ever changed, you'd have to edit all those source files, and if you missed one, your program could fail in subtle (or serious) ways due to the mismatched declarations (i.e. due to the incompatibility between the new declaration in one source file and the old one in a source file you forgot to change). Placing common declarations and definitions into header files means that if they ever change, they only have to be changed in one place, which is a much more workable system.

What should you put in header files?
External declarations of global variables and functions. We said that a global variable must have exactly one *defining instance*, but that it can have *external declarations* in many places. We said that it was a grave

error to issue an external declaration in one place saying that a variable or function has one type, when the defining instance in some other place actually defines it with another type. (If the two places are two source files, separately compiled, the compiler will probably not even catch the discrepancy.) If you put the external declarations in a header file, however, and include the header wherever it's needed, the declarations are virtually guaranteed to be consistent. It's a good idea to include the header in the source file where the defining instance appears, too, so that the compiler can check that the declaration and definition match. (That is, if you ever change the type, you do still have to change it in two places: in the source file where the defining instance occurs, and in the header file where the external declaration appears. But at least you don't have to change it in an arbitrary number of places, and, if you've set things up correctly, the compiler can catch any remaining mistakes.)

Preprocessor macro definitions (which we'll meet in the next section). Structure definitions
Typedef declarations

However, there are a few things *not* to put in header files:
Defining instances of global variables. If you put these in a header file, and include the header file in more than one source file, the variable will end up multi defined.
Function bodies (which are also defining instances). You don't want to put these in headers for the same reason -- it's likely that you'll end up with multiple copies of the function and hence “multiply defined" errors.

People sometimes put commonly-used functions in header files and then use #include to bring them (once) into each program where they use that function, or use #include to bring together the several source files making up a program,

but both of these are poor ideas. It's much better to learn how to use your compiler or linker to combine together separately-compiled object files.

Since header files typically contain only external declarations, and should *not* contain function bodies, you have to understand just what does and doesn't happen when you #include a header file. The header file may provide the declarations for some functions, so that the compiler can generate correct code when you call them (and so that it can make sure that you're calling them correctly), but the header file does *not* give the compiler the functions themselves. The actual functions will be combined into your program at the end of compilation, by the part of the compiler called the *linker*. The linker may have to get the functions out of libraries, or you may have to tell the compiler/linker where to find them. In particular, if you are trying to use a third-party library containing some useful functions, the library will often come with a header file describing those functions. Using the library is therefore a two-step process: you must #include the header in the files where you call the library functions, *and* you must tell the linker to read in the functions from the library itself.

Self Assessment Questions:

1.# include is called _____.

2.Nesting of included files is not allowed. (True/False)

3.Defining instances of global variables is not recommended in the header files. (True/False)

11.3 Macro Definition and Substitution

A preprocessor line of the form

#define *name text*

defines a *macro* with the given name, having as its *value* the given

replacement text. After that (for the rest of the current source file), wherever the preprocessor sees that name, it will replace it with the replacement text. The name follows the same rules as ordinary identifiers (it can contain only letters, digits, and underscores, and may not begin with a digit). Since macros behave quite differently from normal variables (or functions), it is customary to give them names which are all capital letters (or at least which begin with a capital letter). The replacement text can be absolutely anything
-it's not restricted to numbers, or simple strings, or anything.
The most common use for macros is to propagate various constants around and to make them more self-documenting. We've been saying things like char line[100];
...
getline(line, 100);

but this is neither readable nor reliable; it's not necessarily obvious what all those 100's scattered around the program are, and if we ever decide that 100 is too small for the size of the array to hold lines, we'll have to remember to change the number in two (or more) places. A much better solution is to use a macro:
#define MAXLINE 100
char line[MAXLINE];
...
getline(line, MAXLINE);

Now, if we ever want to change the size, we only have to do it in one place, and it's more obvious what the words MAXLINE sprinkled through the program mean than the magic numbers 100 did.

Since the replacement text of a preprocessor macro can be anything, it can also be an expression, although you have to realize that, as always, the text is substituted (and perhaps evaluated) later. No evaluation is performed when the macro is defined. For example, suppose that you write something like

#define A 2
#define B 3
#define C A + B
(this is a pretty meaningless example, but the situation does come up in practice). Then, later, suppose that you write
int x = C * 2;
If A, B, and C were ordinary variables, you'd expect x to end up with the value 10. But let's see what happens.

The preprocessor always substitutes text for macros exactly as you have written it. So it first substitutes the replacement text for the macro C, resulting in
int x = A + B * 2;
Then it substitutes the macros A and B, resulting in

int x = 2 + 3 * 2;

Only when the preprocessor is done doing all this substituting does the compiler get into the act. But when it evaluates that expression (using the normal precedence of multiplication over addition), it ends up initializing x with the value 8!
To guard against this sort of problem, it is always a good idea to include explicit parentheses in the definitions of macros which contain expressions. If we were to define the macro C as

#define C (A + B)
then the declaration of x would ultimately expand to
int x = (2 + 3) * 2;
and x would be initialized to 10, as we probably expected.

Notice that there does not have to be (and in fact there usually is *not*) a semicolon at the end of a #define line. (This is just one of the ways that the syntax of the preprocessor is different from the rest of C.) If you accidentally type

#define MAXLINE 100; /* WRONG */
then when you later declare
char line[MAXLINE];
the preprocessor will expand it to
char line[100;]; /* WRONG */

which is a syntax error. This is what we mean when we say that the preprocessor doesn't know much of anything about the syntax of C-- in this last example, the *value* or replacement text for the macro MAXLINE was the 4 characters 1 0 0 ; , and that's exactly what the preprocessor substituted (even though it didn't make any sense).

Simple macros like MAXLINE act sort of like little variables, whose values are constant (or constant expressions). It's also possible to have macros which look like little functions (that is, you invoke them with what looks like function call syntax, and they expand to replacement text which is a function of the actual arguments they are invoked with) but we won't be looking at these yet.

11.3.1 Macros with arguments

The preprocessor permits us to define more complex and more useful form of replacements. It takes the form:

#define *identifier*(f1, f2, ..., fn) *string*

Notice that there is no space between the macro identifier and the left parenthesis. The identifiers f1, f2, ..., fn are the formal macro arguments that are analogous to the formal arguments in a function definition.

There is a basic difference between the simple replacement discussed above and the replacement of macros with arguments. Subsequent occurrence of a macro with arguments is known as a *macro call*. When a macro is called, the preprocessor substitutes the *string*, replacing the formal parameters with actual parameters.

A simple example of a macro with arguments is
#define CUBE(x) (x*x*x)
If the following statement appears later in the program
volume=CUBE(side);
then the preprocessor would expand this statement to:
volume=(side*side*side);
Consider the following statement:
volume=CUBE(a+b);
This would expand to:
volume=(a+b*a+b*a+b);
which would obviously not produce the correct results. This is because the preprocessor performs a blind text substitution of the argument a+b in place of x. This shortcoming can be corrected by using parentheses for each occurrence of a formal argument in the *string*.
Example:
#define CUBE(x) ((x)*(x)*(x))
This would result in correct expansion of CUBE(a+b) as shown below: volume=((a+b)*(a+b)*(a+b));
Some commonly used definitions are:
#define MAX(a,b) (((a) > (b))?(a):(b))
#define MIN(a,b) (((a) <(b))?(a):(b))

#define ABS(x) (((x) > 0)?(x):(-(x)))

11.3.2 Nesting of Macros

We can also use one macro in the definition of another macro. That is, macro definitions may be nested. For instance, consider the following macro definitions:

#define M 5

#define N M+1

#define SQUARE(x) ((x)*(x))

#define CUBE(x) (SQUARE(x)*(x))

#define SIXTH(x) (CUBE(x)*CUBE(x))

Self Assessment Questions

4.The role of preprocessor in macro substitution is _____ substitution according to the macro definition.

5.Macros and functions are synonyms. (True/False)

6.Nesting of macros is not allowed. (True/False)

11.4 Conditional Compilation

The last preprocessor directive we're going to look at is #ifdef. If you have the sequence

#ifdef *name*

program text

#else

more program text

#endif

in your program, the code that gets compiled depends on whether a preprocessor macro by that *name* is defined or not. If it is (that is, if there has been a #define line for a macro called *name*), then "*program text* is compiled and "*more program text*" is ignored. If the macro is not defined, "*more program text*" is compiled and "*program text*" is ignored. This looks a lot like an if statement, but it behaves completely differently: an if statement controls which statements of your program are executed at run time, but #ifdef controls which parts of your program actually get compiled.

Just as for the if statement, the #else in an #ifdef is optional. There is a companion directive #ifndef, which compiles code if the macro is *not* defined (although the "#else clause" of an #ifndef directive will then be compiled if the macro *is* defined). There is also an #if directive which compiles code depending on whether a compile-time expression is true or false. (The expressions which are allowed in an #if directive are somewhat restricted, however, so we won't talk much about #if here.)

Conditional compilation is useful in two general classes of situations:

You are trying to write a portable program, but the way you do something is different depending on what compiler, operating system, or computer you're using. You place different versions of your code, one for each situation, between suitable #ifdef directives, and when you compile the program in a particular environment, you arrange to have the macro names defined which select the variants you need in that environment.

(For this reason, compilers usually have ways of letting you define macros from the invocation command line or in a configuration file, and
many also predefine certain macro names related to the operating
system, processor, or compiler in use. That way, you don't have to change the code to change the #define lines each time you compile it in a different environment.)

For example, in ANSI C, the function to delete a file is remove. On older Unix systems, however, the function was called unlink. So if filename is a variable containing the name of a file you want to delete, and if you want to be able to compile the program under these older Unix systems,

you might write
#ifdef unix
unlink(filename);
#else
remove(filename);
#endif
Then, you could place the line
#define unix
at the top of the file when compiling under an old Unix system. (Since all you're using the macro unix for is to control the #ifdef, you don't need to give it any replacement text at all. *Any* definition for a macro, even if the replacement text is empty, causes an #ifdef to succeed.) (In fact, in this example, you wouldn't even need to define the macro unix at all, because C compilers on old Unix systems tend to predefine it for you, precisely so you can make tests like these.)

You want to compile several different versions of your program, with different features present in the different versions. You bracket the code for each feature with #ifdef directives, and (as for the previous case) arrange to have the right macros defined or not to build the version you want to build at any given time. This way, you can build the several different versions from the same source code. (One common example is whether you turn debugging statements on or off. You can bracket each debugging printout with #ifdef DEBUG and #endif, and then turn on debugging only when you need it.)

For example, you might use lines like this:
#ifdef DEBUG
printf("x is %d\n", x);
#endif

to print out the value of the variable x at some point in your program to see if it's what you expect. To enable debugging printouts, you insert the line
#define DEBUG

at the top of the file, and to turn them off, you delete that line, but the debugging printouts quietly remain in your code, temporarily deactivated, but ready to reactivate if you find yourself needing them again later. (Also, instead of inserting and deleting the #define line, you might use a compiler flag such as -DDEBUG to define the macro DEBUG from the compiler invocating line.)
Conditional compilation can be very handy, but it can also get out of hand. When large chunks of the program are completely different depending on, say, what operating system the program is being compiled for, it's often better to place the different versions in separate source files, and then only use one of the files (corresponding to one of the versions) to build the program on any given system. Also, if you are using an ANSI Standard compiler and you are writing ANSI-compatible

code, you usually won't need so much conditional compilation, because the Standard specifies exactly how the compiler must do certain things, and exactly which library functions it much provide, so you don't have to

work so hard to accommodate the old variations among compilers and libraries.

Self Assessment Questions

7.In ANSI C, the function to delete a file is _____ .

8.On older Unix systems, the function was called _____.

9.You can use _____ directive which compiles code depending on whether a compile-time expression is true or false.

11.5 Summary

Preprocessor is a unique feature of C language. The C preprocessor provides several tools that are unavailable in other high-level languages. The preprocessor is a program that processes the source code before it passes through the compiler. Macro substitution is a process where an identifier in a program is replaced by a predefined text. The preprocessor permits us to define more complex and more useful form of replacements with macros with arguments. Nesting of macros is also allowed to simplify the complex macro definitions. An external file containing functions or macro definitions can be included as a part of a program.

11.6 Terminal Questions

1.What is the difference between a macro call and a function call?

2.List some of the Compiler Control preprocessor directives.

3.Distinguish between the following macro definitions:

#include “filename” and #include <filename>

4.What is the preprocessor directive used to undefine a macro? Show with an example.

5.Write a macro definition to find the square of a number. Explain it with the help of an example.

11.7 Answers to Self Assessment Questions

1 Preprocessor directive

. False

2 True

. Text

3 False

. False

4 remove

. unlink.

5 #if

.

[6]11.8 Answers to Terminal Questions

· 1.When a macro is called, the macro definition with string substitution will [7] be inserted in the place of call in the source program whereas in a ·function call the execution control will transfer to the function.

8

2.#ifdef, #ifndef, #if, #else

. 3.Preprocessor searches for files enclosed in <> in central, standard9 directories, and it searches for files enclosed in "" in the current directory.

4.#undef (Refer the entire unit for more details)

5.#define SQUARE(x) ((x)*(x)) (Refer the entire unit for more details)

11.9 Exercises

1.Write a macro definition to find the largest of three numbers.

2.Write a macro definition to compare two numbers.

3.What are the advantages of using macro definitions in a program?

4.Define a macro, PRINT_VALUE which can be used to print two values of arbitrary type.

5.Identify errors, if any, in the following macro definition.

#define ABS(x) (x>0)?(x):(-x)

Unit 12 Dynamic Memory Allocation and Linked List

Structure:

12.1 Introduction

Objectives

12.2 Dynamic Memory Allocation

Allocating Memory with **malloc**

Allocating Memory with **calloc**

Freeing Memory
Reallocating Memory Blocks
12.3 Pointer Safety
12.4 The Concept of linked list
Inserting a node by using Recursive Programs
Sorting and Reversing a Linked List

Deleting the Specified Node in a Singly Linked List
12.5 Summary
12.6 Terminal Questions
12.7 Answers to Self Assessment Questions
12.8 Answers for Terminal Questions
12.9 Exercises

12.1 Introduction

In the previous unit, you studied about the preprocessor in C. You studied that the preprocessor is a program that processes the source code before it passes through the compiler and hence it can be used to make C an easy and efficient language. In this unit, you will study about the dynamic memory allocation techniques. You will also study about the linked list that is another useful application using C.

You can use an array when you want to process data of the same data type and you know the size of the data. Sometimes you may want to process the data but you don't know what the size of the data is. An example of this is when you are reading from a file or keyboard and you want to process the values. In such a case, an array is not useful because

you don't know what the dimension of the array should be. C has the facility of dynamic memory allocation. Using this, you can allocate the memory for your storage. The allocation is done at runtime. When your work is over, you can deallocate the memory. The allocation of memory is done using three functions: malloc, **calloc,** and **realloc**. These functions return the pointers to void, so it can be typecast to any data type, thus making the functions generic. These functions take the input as the size of memory requirement.

Objectives:
After studying this unit, you should be able to:
explain about the dynamic memory allocation
implement dynamic memory allocation functions like -malloc, calloc and

realloc
explain the concept of linked lists
discuss the different operations on linked lists

12.2 Dynamic Memory Allocation

The process of allocating memory at run time is known as Dynamic Memory Allocation. Although C does not inherently have this facility, there are four library routines known as “Memory Management Functions” that can be used for allocating and freeing memory during program execution. They are listed in Table 12.1. These functions help us build complex application programs that use the available memory intelligently.

Table 12.1: Functions used for Dynamic Memory allocations and its tasks

Function Tasks

malloc Allocates requested size of bytes and returns a pointer to the
first byte of the allocated space
calloc Allocates space for an array of elements, initializes them to zero
and then returns a pointer to the memory
free Frees previously allocated space
realloc Modifies the size of previously allocated space

Figure 12.1 shows the conceptual view of storage of a C program in memory.

Local variables
Free memory
Global variables
C program instructions

Figure 12.1: storage of a C program- a conceptual view

The program instructions and global and static variables are stored in a region known as *permanent storage area* and the local variables are stored in another area called *stack*. The memory space that is located between these two regions is available for dynamic allocation during execution of the program. This free memory region is called *heap*. The size of the heap keeps changing when program is executed due to creation and death of variables that are local to functions and blocks. Therefore, it is possible to encounter memory "overflow" during dynamic allocation process.

12.2.1 Allocating Memory with malloc

A problem with many simple programs, including in particular little teaching programs such as we've been writing so far, is that they tend to use fixedsize arrays which may or may not be big enough. We have an array of 100 ints for the numbers which the user enters and wishes to find the average of them, what if the user enters 101 numbers? We have an array of 100 chars which we pass to getline to receive the user's input, what if the user types a line of 200 characters?

If we're lucky, the relevant parts of the program check how much of an array they've used, and print an error message or otherwise gracefully abort before overflowing the array. If we're not so lucky, a program may sail off the end of an array, overwriting other data and behaving quite badly. In either case, the user doesn't get his job done. How can we avoid the restrictions of fixed-size arrays?

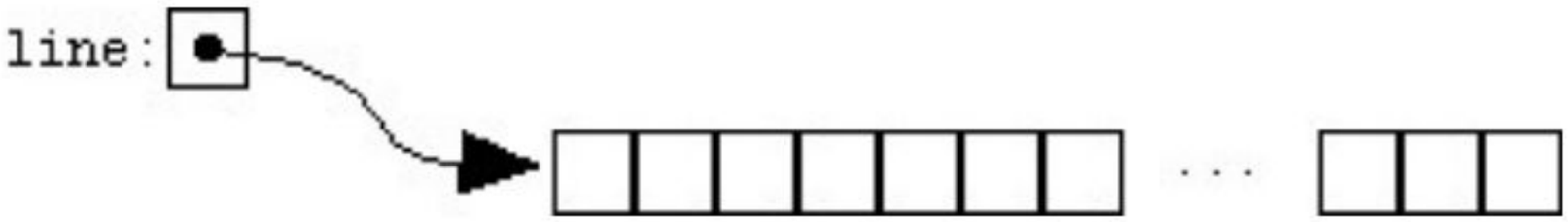
The answers all involve the standard library function malloc. Very simply, malloc returns a pointer to n bytes of memory which we can do anything we want to with. If we didn't want to read a line of input into a fixed-size array, we could use malloc, instead. It takes the following form:

ptr=(*cast-type* *)malloc(*byte-size*)
where ptr is a pointer of type *cast-type*. The malloc returns a pointer(of *casttype*) to an area of memory with size *byte-size*.

Here is an example

```
#include <stdlib.h>
char *line;
int linelen = 100;
line = (char *)malloc(linelen);
/* incomplete -- malloc's return value not checked */
getline(line, linelen);
```

malloc is declared in <stdlib.h>, so we #include that header in any program that calls malloc. A “byte" in C is, by definition, an amount of storage suitable for storing one character, so the above invocation of malloc gives us exactly as many chars as we ask for. We could illustrate the resulting pointer like this:



The 100 bytes of memory (not all of which are shown) pointed to by line are those allocated by malloc. (They are brand-new memory, conceptually a bit different from the memory which the compiler arranges to have allocated automatically for our conventional variables. The 100 boxes in the figure don't have a name next to them, because they're not storage for a variable we've declared.)

As a second example, we might have occasion to allocate a piece of memory, and to copy a string into it with strcpy:

```
char *p = (char *)malloc(15);

/* incomplete -- malloc's return value not checked */
strcpy(p, "Hello, world!");
```

When copying strings, remember that all strings have a terminating \0 character. If you use strlen to count the characters in a string for you, that count will *not* include the trailing \0, so you must add one before calling malloc:

```
char *somestring, *copy;
...
copy = (char *)malloc(strlen(somestring) + 1); /* +1 for \0 */
/* incomplete -- malloc's return value not checked */
strcpy(copy, somestring);
```

What if we're not allocating characters, but integers? If we want to allocate 100 ints, how many bytes is that? If we know how big ints are on our machine (i.e. depending on whether we're using a 16- or 32-bit machine) we could try to compute it ourselves, but it's much safer and more portable to let C compute it for us. C has a sizeof operator, which computes the size, in bytes, of a variable or type. It's just what we need when calling malloc. To allocate space for 100 ints, we could call

```
int *ip =(int *)malloc(100 * sizeof(int));
```

The use of the sizeof operator tends to look like a function call, but it's really an operator, and it does its work at compile time.

Since we can use array indexing syntax on pointers, we can treat a pointer variable after a call to malloc almost exactly as if it were an array. In particular, after the above call to malloc initializes ip to point at storage for 100 ints, we can access ip[0], ip[1], ... up to ip[99]. This way, we can get the effect of an array even if we don't know until run time how big the “array" should be. (In a later section we'll see how we might deal with the case where we're not even sure at the point we begin using it how big an “array" will eventually have to be.)

Our examples so far have all had a significant omission: they have not checked malloc's return value. Obviously, no real computer has an infinite amount of memory available, so there is no guarantee that malloc will be able to give us as much memory as we ask for. If we call

malloc(100000000), or if we call malloc(10) 10,000,000 times, we're probably going to run out of memory.

When malloc is unable to allocate the requested memory, it returns a *null pointer*. A null pointer, remember, points definitively nowhere. It's a “not a pointer" marker; it's not a pointer you can use. Therefore, whenever you call malloc, it's vital to check the returned pointer before using it! If you call malloc, and it returns a null pointer, and you go off and use that null pointer as if it pointed somewhere, your program probably won't last long. Instead, a program should immediately check for a null pointer, and if it receives one, it should at the very least print an error message and exit, or perhaps figure out some way of proceeding without the memory it asked for. But it cannot go on to use the null pointer

it got back from malloc in any way, because that null pointer by definition points nowhere. (“It cannot use a null pointer in any way" means that the program cannot use the * or [] operators on such a pointer value, or pass it to any function that expects a valid pointer.) A call to malloc, with an error check, typically looks something like this: int *ip = (int *)malloc(100 * sizeof(int));

```
if(ip == NULL)
{
printf("out of memory\n");
exit or return
}
```

After printing the error message, this code should return to its caller, or exit from the program entirely; it cannot proceed with the code that would have used ip.

Of course, in our examples so far, we've still limited ourselves to “fixed size" regions of memory, because we've been calling malloc with fixed arguments like 10 or 100. (Our call to getline is still limited to 100-character lines, or whatever number we set the linelen variable to; our ip variable still points at only 100 ints.) However, since the sizes are now values which can in principle be determined at run-time, we've at least moved beyond having to recompile the program (with a bigger array) to accommodate longer lines, and with a little more work, we could arrange that the “arrays" automatically grew to be as large as required. (For example, we could write something like getline which could read the longest input line actually seen).

12.2.2 Allocating memory with calloc

calloc is another memory allocation function that is normally used for requesting memory space at run time for storing derived data types such as arrays and structures. While malloc allocates a single block of storage space, calloc allocates multiple blocks of storage, each of the same size, and then sets all bytes to zero. The general form of calloc is:

ptr=(*cast-type* *)calloc(*n, elem-size*);

The above statement allocates contiguous space for n blocks, each of size *elem-size* bytes. All bytes are initialized to zero and a pointer to the first byte of the allocated region is returned. If there is not enough space, a NULL pointer is returned.

Program 12.1 Program to illustrate the use of malloc and calloc functions #include <stdio.h>

```
#include <malloc.h>
main()
{
int *base;
int i;
int cnt=0;
int sum=0;
printf("how many integers you have to store \n");
scanf("%d",&cnt);
base = (int *)malloc(cnt * sizeof(int));
printf("the base of allocation is %16lu \n",base);
if(!base)
printf("unable to allocate size \n");
else
{
for(int j=0;j<cnt;j++)
*(base+j)=5;
}
sum = 0;
```

```
for(int j=0;j<cnt;j++)
sum = sum + *(base+j);
printf("total sum is %d\n",sum);
free(base);
printf("the base of allocation is %16lu \n",base);
base = (int *)malloc(cnt * sizeof(int));
printf("the base of allocation is %16lu \n",base);
base = (int *)malloc(cnt * sizeof(int));
printf("the base of allocation is %16lu \n",base);
base = (int *)calloc(10,2);
printf("the base of allocation is %16lu \n",base);
}
```

12.2.3 Freeing Memory

Memory allocated with malloc lasts as long as you want it to. It does not automatically disappear when a function returns, as automatic variables do, but it does not have to remain for the entire duration of your program, either. Just as you can use malloc to control exactly when and how much memory you allocate, you can also control exactly when you deallocate it.

In fact, many programs use memory on a transient basis. They allocate some memory, use it for a while, but then reach a point where they don't need that particular piece any more. Because memory is not inexhaustible, it's a good idea to deallocate (that is, release or *free*) memory you're no longer using.

Dynamically allocated memory is deallocated with the free function. If p contains a pointer previously returned by malloc, you can call
free(p);

which will “give the memory back" to the stock of memory (sometimes called the “arena" or “pool") from which malloc requests are satisfied. Calling free is sort of the ultimate in recycling: it costs you almost nothing, and the memory you give back is immediately usable by other parts of your program. (Theoretically, it may even be usable by other programs.)

(Freeing unused memory is a good idea, but it's not mandatory. When your program exits, any memory which it has allocated but not freed should be automatically released. If your computer were to somehow “lose" memory just because your program forgot to free it, that would indicate a problem or deficiency in your operating system.)

Naturally, once you've freed some memory you must remember not to use it any more. After calling
free(p);

it is probably the case that p still points at the same memory. However, since we've given it back, it's now “available," and a later call to malloc might give that memory to some other part of your program. If the variable p is a global variable or will otherwise stick around for a while, one good way to record the fact that it's not to be used any more would be to set it to a null pointer: free(p);
p = NULL;
Now we don't even have the pointer to the freed memory any more, and (as long as we check to see that p is non-NULL before using it), we won't misuse any memory via the pointer p.

When thinking about malloc, free, and dynamically-allocated memory in general, remember again the distinction between a pointer and what it points to. If you call malloc to allocate some memory, and store the pointer which malloc gives you in a local pointer variable, what happens when the function containing the local pointer variable returns? If the local pointer variable has *automatic duration* (which is the default, unless the variable is declared static), it will disappear when the function returns. But for the pointer variable to disappear says nothing about the memory pointed to! That memory still exists and, as far as malloc and free are concerned, is still allocated. The only thing that has disappeared is the pointer variable you had which pointed at the allocated memory. (Furthermore, if it contained the only copy of the pointer you had, once it disappears, you'll have no way of freeing the memory, and no way of using it, either. Using memory and freeing memory both require that you have at least one pointer to the memory!)

12.2.4 Reallocating Memory Blocks

Sometimes you're not sure at first how much memory you'll need. For example, if you need to store a series of items you read from the user, and if the only way to know how many there are is to read them until the user types some “end" signal, you'll have no way of knowing, as you begin reading and storing the first few, how many you'll have seen by the time you do see that “end" marker. You might want to allocate room for, say, 100 items, and if the user enters a 101st item before entering the “end" marker, you might wish for a way to say ”uh, malloc, remember those 100 items I asked for? Could I change my mind and have 200 instead?"

In fact, you can do exactly this, with the realloc function. You hand realloc an old pointer (such as you received from an initial call to malloc) and a new size, and realloc does what it can to give you a chunk of memory big enough to hold the new size. For example, if we wanted the ip variable from an earlier example to point at 200 ints instead of 100, we could try calling

```
ip = realloc(ip, 200 * sizeof(int));
```

Since you always want each block of dynamically-allocated memory to be contiguous (so that you can treat it as if it were an array), you and realloc have to worry about the case where realloc can't make the old block of memory bigger

“in place," but rather has to relocate it elsewhere in order to find enough contiguous space for the new requested size. realloc does this by returning a new pointer. If realloc was able to make the old block of memory bigger, it returns the same pointer. If realloc has to go elsewhere to get enough contiguous memory, it returns a pointer to the new memory, after copying your old data there. (In this case, after it makes the copy, it frees the old block.) Finally, if realloc can't find enough memory to satisfy the new request at all, it returns a null pointer. Therefore, you usually don't want
to overwrite your old pointer with realloc's return value until you've tested it to make sure it's not a null pointer. You might use code like this:

```
int *newp;
newp = realloc(ip, 200 * sizeof(int));
if(newp != NULL)
ip = newp;
else {
printf("out of memory\n");
/* exit or return */
/* but ip still points at 100 ints */
}
```

If realloc returns something other than a null pointer, it succeeded, and we set ip to what it returned. (We've either set ip to what it used to be or to a new pointer, but in either case, it points to where our data is now.) If realloc returns a null pointer, however, we hang on to our old pointer in ip which still points at our original 100 values.

Putting this all together, here is a piece of code which reads lines of text from the user, treats each line as an integer by calling atoi, and stores each integer in a dynamically-allocated “array":

```
#define MAXLINE 100
char line[MAXLINE];
int *ip;
int nalloc, nitems;
nalloc = 100;
ip = (int *)malloc(nalloc * sizeof(int)); /* initial allocation */ if(ip == NULL)
{
printf("out of memory\n");
exit(1);
}

nitems = 0;
while(getline(line, MAXLINE) != EOF) {
if(nitems >= nalloc)
{ /* increase allocation */
int *newp;
nalloc += 100;
newp = realloc(ip, nalloc * sizeof(int)); if(newp == NULL)
{
printf("out of memory\n");
exit(1);
}
ip = newp;
}
```

```
ip[nitems++] = atoi(line);
}
```

We use two different variables to keep track of the “array" pointed to by ip. nalloc is how many elements we've allocated, and nitems is how many of them are in use. Whenever we're about to store another item in the “array," if nitems >= nalloc, the old “array" is full, and it's time to call realloc to make it bigger.

Finally, we might ask what the return type of malloc and realloc is, if they are able to return pointers to char or pointers to int or (though we haven't seen it yet) pointers to any other type. The answer is that both of these functions are declared (in <stdlib.h>) as returning a type we haven't seen, void * (that is, pointer to void). We haven't really seen type void, either, but what's going on here is that void * is specially defined as a “generic" pointer type, which may be used (strictly speaking, assigned to or from) any pointer type.

Self Assessment Questions

1.The process of allocating memory at run time is known as _____.
2.malloc() function returns a pointer to integer. (True/False)
3.For deallocating memory, you can use _____ function.

4.The function that is used to alter the size of a block previously allocated is _____.

12.3 Pointer Safety

The hard thing about pointers is not so much manipulating them as ensuring that the memory they point to is valid. When a pointer doesn't point where you think it does, if you inadvertently access or modify the memory it points to,

you can damage other parts of your program, or (in some cases) other programs or the operating system itself!

When we use pointers to simple variables, there's not much that can go wrong. When we use pointers into arrays, and begin moving the pointers around, we have to be more careful, to ensure that the moving pointers always stay within the bounds of the array(s). When we begin passing pointers to functions, and especially when we begin returning them from functions, we have to be more careful still, because the code using the pointer may be far removed from the code which owns or allocated the memory.
One particular problem concerns functions that return pointers. Where is the memory to which the returned pointer points? Is it still around by the time the function returns? The strstr function (as described in the earlier unit), returns either a null pointer (which points definitively nowhere, and which the caller presumably checks for) or it returns a pointer which points into the input string, which the caller supplied, which is pretty safe. One thing a function must *not* do, however, is return a pointer to one of its own, local, automatic arrays. Remember that automatic variables (which includes all non-static local variables), including automatic arrays, are deallocated and disappear when the function returns. If a function returns a pointer to a local array, that pointer will be invalid by the time the caller tries to use it.

Finally, when we're doing dynamic memory allocation with malloc, calloc, realloc, and free, we have to be most careful of all. Dynamic allocation gives us a lot more flexibility in how our programs use memory, although with that flexibility comes the responsibility that we manage dynamically allocated memory carefully. The possibilities for misdirected pointers and associated havoc are greatest in programs that make heavy use of dynamic memory allocation. You can reduce these possibilities by designing your program in such a way that it's easy to ensure that pointers are used correctly and that memory is always allocated and deallocated correctly. (If, on the other hand, your program is designed in such a way that meeting these guarantees is a tedious nuisance, sooner or later you'll forget or neglect to, and maintenance will be a nightmare.)

12.4 The Concept of Linked List

When dealing with many problems we need a dynamic list, dynamic in the sense that the size requirement need not be known at compile time. Thus, the list may grow or shrink during runtime. A *linked list* is a data structure that is used to model such a dynamic list of data items, so the study of the linked lists as one of the data structures is important.

An array is represented in memory using sequential mapping, which has the property that elements are fixed distance apart. But this has the following disadvantage: It makes insertion or deletion at any arbitrary position in an array a costly operation, because this involves the movement of some of the existing elements.

When we want to represent several lists by using arrays of varying size, either we have to represent each list using a separate array of maximum size or we have to represent each of the lists using one single array. The first one will lead to wastage of storage, and the second will involve a lot of data movement.

So we have to use an alternative representation to overcome these disadvantages. One alternative is a linked representation. In a linked representation, it is not necessary that the elements be at a fixed distance apart. Instead, we can place elements anywhere in memory, but to make it a part of the same list, an element is required to be linked with a previous element of the list. This can be done by storing the address of the next element in the previous element itself. This requires that every element be capable of holding the data as well as the address of the next element. Thus every element must be a structure with a minimum of two fields, one for holding the data value, which we call a data field, and the other for holding the address of the next element, which we call link field.

Therefore, a linked list is a list of elements in which the elements of the list can be placed anywhere in memory, and these elements are linked with each other using an explicit link field, that is, by storing the address of the next element in the link field of the previous element.

Program 12.2: Here is a program for building and printing the elements of the linked list

```
# include <stdio.h>
# include <stdlib.h>
struct node
{
int data;
struct node *link;
};
struct node *insert(struct node *p, int n)
{
struct node *temp;
/* if the existing list is empty then insert a new node as the
starting node */
if(p==NULL)
{
p=(struct node *)malloc(sizeof(struct node)); /* creates new node data value passes as parameter */
if(p==NULL)
{
printf("Error\n");
exit(0);
}
```

```
p-> data = n;
p-> link = p; /* makes the pointer pointing to itself because it

is a circular list*/
}
else
{
temp = p;
/* traverses the existing list to get the pointer to the last node of it */
while (temp-> link != p)
temp = temp-> link;
temp-> link = (struct node *)malloc(sizeof(struct node)); /* creates new node using data value passes as parameter and
puts its address in the link field of last node of the existing list*/ if(temp -> link == NULL)
{
printf("Error\n");
exit(0);
}
temp = temp-> link;
temp-> data = n;
temp-> link = p;
}
return (p);
}
void printlist ( struct node *p )
```

```
{
struct node *temp;
temp = p;
printf("The data values in the list are\n");
if(p!= NULL)
{
do
{
printf("%d\t",temp->data);
temp=temp->link;
} while (temp!= p);
}
else
printf("The list is empty\n"); }

void main()
{
int n;
int x;
struct node *start = NULL ;
printf("Enter the nodes to be created \n");
scanf("%d",&n);
while ( n -- > 0 )
{
```

```
printf( "Enter the data values to be placed in a node\n");
scanf("%d",&x);
start = insert ( start, x );
}
printf("The created list is\n");
printlist ( start );
}
```

12.4.1 Inserting a node by using Recursive Programs

A linked list is a recursive data structure. A *recursive data structure* is a data structure that has the same form regardless of the size of the data. You can easily write recursive programs for such data structures.

Program 12.3

```
# include <stdio.h>
# include <stdlib.h>
struct node
{
int data;
struct node *link;
};
struct node *insert(struct node *p, int n)
if(p==NULL)
{
p=(struct node *)malloc(sizeof(struct node)); if(p==NULL)
{
printf("Error\n");
```

```
exit(0);
}
p-> data = n;
p-> link = NULL;
}
else

p->link = insert(p->link,n);/* the while loop replaced by recursive call */
return (p);
}
void printlist ( struct node *p )
{
printf("The data values in the list are\n");
while (p!= NULL)
{
printf("%d\t",p-> data);
p = p-> link;
}
}
void main()
{
int n;
int x;
struct node *start = NULL ;
```

```
printf("Enter the nodes to be created \n");
scanf("%d",&n);
while ( n- - 0 )
{

printf( "Enter the data values to be placed in a node\n"); scanf("%d",&x);
start = insert ( start, x );
}
printf("The created list is\n");
printlist ( start );
}
```

12.4.2 Sorting and reversing a linked list

To sort a linked list, first we traverse the list searching for the node with a minimum data value. Then we remove that node and append it to another list which is initially empty. We repeat this process with the remaining list until the list becomes empty, and at the end, we return a pointer to the beginning of the list to which all the nodes are moved.
To reverse a list, we maintain a pointer each to the previous and the next node, then we make the link field of the current node point to the previous, make the previous equal to the current, and the current equal to the next.

Therefore, the code needed to reverse the list is

```
Prev = NULL;
While (curr != NULL)
{
```

```
Next = curr->link;
Curr -> link = prev;
Prev = curr;
Curr = next;
}
```

Program 12.4: Example in sorting lists.

```
# include <stdio.h>
# include <stdlib.h>
struct node
{
int data;
struct node *link;
};
struct node *insert(struct node *p, int n)
if(p==NULL)
{
p=(struct node *)malloc(sizeof(struct node));
if(p==NULL)
{
printf("Error\n");
exit(0);
}
p-> data = n;
```

```
p-> link = NULL;
}
else
{
temp = p;
while (temp-> link!= NULL)
temp = temp-> link;

temp-> link = (struct node *)malloc(sizeof(struct node)); if(temp -> link == NULL)
{
printf("Error\n");
exit(0);
}
temp = temp-> link;
temp-> data = n;
temp-> link = null;
}
return(p);
}

void printlist ( struct node *p )
{
printf("The data values in the list are\n"); while (p!= NULL)
{
```

```
printf("%d\t",p-> data);
p = p-> link;

}
}

/* a function to sort reverse list */ struct node *reverse(struct node *p) {
struct node *prev, *curr;
prev = NULL;
curr = p;
while (curr != NULL)
{
p = p-> link;
curr-> link = prev;
prev = curr;
curr = p;
}
return(prev);
}
/* a function to sort a list */
struct node *sortlist(struct node *p) {

struct node *temp1,*temp2,*min,*prev,*q; q = NULL;
while(p != NULL)
{
```

```
prev = NULL;
min = temp1 = p;
temp2 = p -> link;
while ( temp2 != NULL )
{
if(min -> data > temp2 -> data)
{
min = temp2;
prev = temp1;

} temp1 = temp2;
temp2 = temp2-> link;
}
if(prev == NULL)
p = min -> link;
else
prev -> link = min -> link;
min -> link = NULL;
if( q == NULL)
q = min; /* moves the node with lowest data value in the list
pointed to by p to the list pointed to by q as a first node*/
else
{
temp1 = q;
```

```
/* traverses the list pointed to by q to get pointer to its
last node */
while( temp1 -> link != NULL)
temp1 = temp1 -> link;

temp1 -> link = min; /* moves the node with lowest data value in the list pointed to by p to the list pointed to by q at the
end of list pointed by q*/ }
}
return (q);
}

void main()
{
int n;
int x;
struct node *start = NULL ;
printf("Enter the nodes to be created \n"); scanf("%d",&n);
while ( n- > 0 )
{
printf( "Enter the data values to be placed in a node\n");
scanf("%d",&x);

start = insert ( start,x);
}
printf("The created list is\n");
```

```
printlist ( start );
start = sortlist(start);
printf("The sorted list is\n");
printlist ( start );
start = reverse(start);
printf("The reversed list is\n");
printlist ( start );
}
```

12.4.3 Deleting the specified node in a singly linked list

To delete a node, first we determine the node number to be deleted (this is based on the assumption that the nodes of the list are numbered serially from 1 to n). The list is then traversed to get a pointer to the node whose number is given, as well as a pointer to a node that appears before the node to be deleted. Then the link field of the node that appears before the node to be deleted is made to point to the node that appears after the node to be deleted, and the node to be deleted is freed.

Program 12.5: Example of working with nodes.

```
# include <stdio.h>
# include <stdlib.h>
struct node *delet ( struct node *, int );
int length ( struct node * );
struct node
{
int data;
struct node *link;
```

```
};
struct node *insert(struct node *p, int n)
{
struct node *temp;
if(p==NULL)
{
p=(struct node *)malloc(sizeof(struct node));

if(p==NULL)
{
printf("Error\n");
exit(0);
}
p-> data = n;
p-> link = NULL;
}
else
{
temp = p;
while (temp-> link != NULL)
temp = temp-> link;

temp-> link = (struct node *)malloc(sizeof(struct node)); if(temp -> link == NULL)
{
printf("Error\n");
```

```
exit(0);
}
temp = temp-> link;
temp-> data = n;
temp-> link = NULL;
}
return (p);
}

void printlist ( struct node *p )
{
printf("The data values in the list are\n");
while (p!= NULL)
{
printf("%d\t",p-> data);
p = p-> link;
}
}
void main()
{
int n;
int x;
struct node *start = NULL;
printf("Enter the nodes to be created \n"); scanf("%d",&n);
```

```
while ( n-- > 0 )
{

printf( "Enter the data values to be placed in a node\n"); scanf("%d",&x);
start = insert ( start, x );
}
printf(" The list before deletion id\n");
printlist ( start );
printf("% \n Enter the node no \n");
scanf ( " %d",&n);
start = delet (start , n );
printf(" The list after deletion is\n");
printlist ( start );
}

/* a function to delete the specified node*/
struct node *delet ( struct node *p, int node_no ) {

struct node *prev, *curr ;
int i;

if (p == NULL )
{
printf("There is no node to be deleted \n"); }
else
```

```
{
if ( node_no > length (p))
{
printf("Error\n");
else

{
prev = NULL;
curr = p;
i = 1 ;
while ( i < node_no )
{
prev = curr;
curr = curr-> link;
i = i+1;
}
if ( prev == NULL )
{
p = curr -> link;
free ( curr );
}
else
{
prev -> link = curr -> link ;
```

```
free ( curr );
}
}
}
return(p);
}
/* a function to compute the length of a linked list */ int length ( struct node *p )
{
int count = 0 ;
while ( p != NULL )
{
count++;
p = p->link;
}
return ( count ) ;
```

Self Assessment Questions

5.A linked list is a data structure that is used to model a dynamic list of data items and is a collection of _____.

6.Linked list make use of _____ memory allocation technique.

12.5 Summary

The process of allocating memory at run time is known as Dynamic Memory Allocation. The allocation of memory is

done using three functions: malloc, calloc, and realloc. These functions return the pointers to void, so it can be typecast to any data type, thus making the functions generic. The memory space that is used for dynamic allocation during execution of the program is called *heap*. When a pointer doesn't point where you think it does, if you inadvertently access or modify the memory it points to, you can damage other parts of your program. So safety of pointers is essential in dynamic memory allocation. Linked list is one of the applications of dynamic memory allocation.

12.6 Terminal Questions

1.Assuming that there is a structure definition with structure tag **student,** write the **malloc** function to allocate space for the structure.

2.Write a structure definition to represent a node in the linked list with two data fields of type integer.
3.If **ptr** points to a node and **link** is the address field of the node, how can you move from the node to the next node?
4.Explain about the linked list.

12.7 Answers to Self Assessment Questions

1.Dynamic Memory Allocation.
2. False
3. free()
4. realloc()
5. nodes.
6. dynamic

12.8 Answers to Terminal Questions

1 ptr=(struct student *) malloc(sizeof(struct student));

. struct node

2{

.

int data1;

int data2;

struct node *link;

};

3 ptr=ptr->link;

. A linked list is a data structure that is used to model a dynamic list of

4 data items and is a collection of nodes. (Refer to section 12.4 for more . details)

12.9 Exercises

1.Write a menu driven program to create a linked list of a class of students and perform the following operations.

i.Write out the contents of a list

ii.Edit the details of a specified student

iii.Count the number of students above a specified age and weight

2.Write a program to insert a node after the specified node in a linked list.

3.Write a program to count the number of nodes in linked list.

4.Write a program to merge two sorted lists.

5.Explain briefly how to represent polynomials using linked lists. Write a program to add two polynomials.

Unit 13 File Management

Structure:

13.1 Introduction

Objectives

13.2 Defining and Opening a file

13.3 Closing Files

13.4 Input/Output Operations on Files

Predefined Streams

13.5 Error Handling during I/O Operations

13.6 Random Access to Files

13.7 Command Line Arguments

13.8 Summary

13.9 Terminal Questions

13.10 Answers to Self Assessment Questions

13.11 Answers to Terminal Questions

13.12 Exercises

13.1 Introduction

In nearly all the previous units, so far, we've been calling **printf** to print formatted output to the “standard output" (wherever that is). We've also been calling **getchar** to read single characters from the “standard input," and **putchar** to write single characters to the standard output. “Standard input" and “standard output" are two predefined I/O *streams* which are implicitly available to us. In this unit, you will study how to take control of input and output by opening our own streams, perhaps connected to data files, which we can read from and write to.

Objectives:
After studying this unit, you should be able to:
open your file and control input and output

connect to file and able to read from the file and write to the file using file pointers.
checkg for error during I/O operations on files.
implement the Random Access functions to allow control over the implied read/write position in the file.
supply a parameter to a program when the program is invoked using a command line arguments.

13.2 Defining and Opening a File

How will we specify that we want to access a particular data file? It would theoretically be possible to mention the name of a file each time it was desired to read from or write to it. But such an approach would have a number of drawbacks. Instead, the usual approach (and the one taken in C's stdio library) is that you mention the name of the file

once, at the time you *open* it. Thereafter, you use some little token in this case, the *file pointer* which keeps track (both for your sake and the library's) of which file you're talking about. Whenever you want to read from or write to one of the files you're working with, you identify that file by using its file pointer (that is, the file pointer you obtained when you opened the file). As we'll see, you store
file pointers in variables just as you store any other data you manipulate, so it is possible to have several files open, as long as you use distinct variables to store the file pointers.

You declare a variable to store a file pointer like this:
FILE *fp;

The type FILE is predefined for you by <stdio.h>. It is a data structure which holds the information the standard I/O library needs to keep track of the file for you. For historical reasons, you declare a variable which is a pointer to this FILE type. The name of the variable can (as for any variable) be anything you choose; it is traditional to use the letters fp in the variable name (since we're talking about a file pointer). If you were reading from two files at once you'd probably use two file pointers:

FILE *fp1, *fp2;

If you were reading from one file and writing to another you might declare an input file pointer and an output file pointer:
FILE *ifp, *ofp;

Like any pointer variable, a file pointer isn't good until it's initialized to point to something. (Actually, no variable of any type is much good until you've initialized it.) To actually open a file, and receive the “token" which you'll store in your file pointer variable, you call fopen. fopen accepts a file name (as a string) and a *mode* value indicating among other

things whether you intend to read or write this file. (The mode variable is also a string.) To open the file input.dat for reading you might call

```
ifp = fopen("input.dat", "r");
```

The mode string "r" indicates reading. Mode "w" indicates writing, so we could open output.dat for output like this:

```
ofp = fopen("output.dat", "w");
```

The other values for the mode string are less frequently used. The third major mode is "a" for append. (If you use "w" to write to a file which already exists, its old contents will be discarded.) You may also add “+” character to the mode string to indicate that you want to both read and write, or a “b” character to indicate that you want to do “binary" (as opposed to text) I/O.

One thing to beware of when opening files is that it's an operation which may fail. The requested file might not exist, or it might be protected against reading or writing. (These possibilities ought to be obvious, but it's easy to forget them.) fopen returns a null pointer if it can't open the requested file, and it's important to check for this case before going off and using fopen's return value as a file pointer. Every call to fopen will typically be followed with a test, like this:

```
ifp = fopen("input.dat", "r");
if(ifp == NULL)
{
printf("can't open file\n");
exit or return
}
```

If fopen returns a null pointer, and you store it in your file pointer variable and go off and try to do I/O with it, your program will typically crash. It's common to collapse the call to fopen and the assignment in with the test:

```
if((ifp = fopen("input.dat", "r")) == NULL)
{
printf("can't open file\n");
exit or return
```

You don't have to write these “collapsed" tests if you're not comfortable with them, but you'll see them in other people's code, so you should be able to read them.

Self Assessment Questions

1.The type FILE is predefined in the header file _____.

2.We may add a “+” character to the mode string in the fopen function to indicate that we want to both read and write. (True/False)

13.3 Closing Files

Although you can open multiple files, there's a limit to how many you can have open at once. If your program will open many files in succession, you'll want to close each one as you're done with it; otherwise the standard I/O library could run out of the resources it uses to keep track of open files. Closing a file simply involves calling fclose with the file pointer as its argument:

```
fclose(fp);
```

Calling fclose arranges that (if the file was open for output) any last, buffered output is finally written to the file, and that those resources used by the operating system (and the C library) for this file are released. If you forget to close a file, it will be closed automatically when the program exits.

Self Assessment Questions

3.Closing a file simply involves calling fclose with the _____ as its argument.

4.If you forget to close a file, it will be closed automatically when the program exits. (True/False)

13.4 Input/Output Operations on Files

For each of the I/O library functions we've been using so far, there's a companion function which accepts an additional file pointer argument telling it where to read from or write to. The companion function to printf is fprintf, and the file pointer argument comes first. To print a string to the output.dat file we opened in the previous section, we might call

```
fprintf(ofp, "Hello, world!\n");
```

The companion function to getchar is getc, and the file pointer is its only argument. To read a character from the input.dat file we opened in the previous section, we might call

```
int c;
c = getc(ifp);
```

The companion function to putchar is putc, and the file pointer argument comes last. To write a character to output.dat, we could call
putc(c, ofp);

Our own getline function calls getchar and so always reads the standard input. We could write a companion fgetline function which reads from an arbitrary file pointer:

```
#include <stdio.h>

/* Read one line from fp, */
/* copying it to line array (but no more than max chars). */
/* Does not place terminating \n in line array. */
/* Returns line length, or 0 for empty line, or EOF for end-of-file. */

int fgetline(FILE *fp, char line[], int max)
{
int nch = 0;
int c;
max = max - 1; /* leave room for '\0' */

while((c = getc(fp)) != EOF)
{
if(c == '\n')
break;
```

```
if(nch < max)
{
line[nch] = c;
nch = nch + 1;
}
if(c == EOF && nch == 0) return EOF;

line[nch] = '\0';
return nch;
}
```

Now we could read one line from ifp by calling char line[MAXLINE];

```
...
fgetline(ifp, line, MAXLINE);
```

Program 13.1: Writing a data to the file

```
#include <stdio.h>
main( )
{
FILE *fp;
char stuff[25];
int index;
fp = fopen("TENLINES.TXT","w"); /* open for writing */
strcpy(stuff,"This is an example line.");
for (index = 1; index <= 10; index++)
fprintf(fp,"%s Line number %d\n", stuff, index);
```

```
fclose(fp); /* close the file before ending program */
}
```

13.4.1 Predefined Streams

Besides the file pointers which we explicitly open by calling fopen, there are also three predefined streams. stdin is a constant file pointer corresponding to standard input, and stdout is a constant file pointer corresponding to standard output. Both of these can be used anywhere a file pointer is called for; for example, getchar() is the same as getc(stdin) and putchar(c) is the same as putc(c, stdout). The third predefined stream is stderr. Like stdout, stderr is typically connected to the screen by default. The difference is that stderr is not redirected when the standard output is redirected. For example, under Unix or MS-DOS, when you invoke

```
program > filename
```

anything printed to stdout is redirected to the file filename, but anything printed to stderr still goes to the screen. The intention behind stderr is that it is the “standard error output"; error messages printed to it will not disappear into an output file. For example, a more realistic way to print an error message when a file can't be opened would be

```
if((ifp = fopen(filename, "r")) == NULL)
{
fprintf(stderr, "can't open file %s\n", filename);
exit or return
}
```

where filename is a string variable indicating the file name to be opened. Not only is the error message printed to stderr, but it is also more informative in that it mentions the name of the file that couldn't be opened.

Program 13.2: To read a data file input.dat

Suppose you had a data file consisting of rows and columns of numbers:

1 2 34
5 6 78
9 10 112

Suppose you wanted to read these numbers into an array. (Actually, the array will be an array of arrays, or a “multidimensional" array;) We can write code to do this by putting together several pieces: the fgetline function we just showed, and the getwords function from which each line broken into words. Assuming that the data file is named input.dat, the code would look like this: #define MAXLINE 100
#define MAXROWS 10
#define MAXCOLS 10

```
int array[MAXROWS][MAXCOLS]; char *filename = "input.dat"; FILE *ifp;
char line[MAXLINE];
char *words[MAXCOLS];
int nrows = 0;
int n; int i;

ifp = fopen(filename, "r");
if(ifp == NULL)
{
fprintf(stderr, "can't open %s\n", filename); exit(EXIT_FAILURE);
}
```

```
while(fgetline(ifp, line, MAXLINE) != EOF) {
if(nrows >= MAXROWS)
{
fprintf(stderr, "too many rows\n");
exit(EXIT_FAILURE);
}

n = getwords(line, words, MAXCOLS);

for(i = 0; i < n; i++)
array[nrows][i] = atoi(words[i]);
nrows++;
}
```

Each trip through the loop reads one line from the file, using fgetline. Each line is broken up into “words" using getwords; each “word" is actually one number. The numbers are however still represented as strings, so each one is converted to an int by calling atoi before being stored in the array. The code checks for two different error conditions (failure to open the input file, and too many lines in the input file) and if one of these conditions occurs, it prints an error message, and exits. The exit function is a Standard library function which terminates your program. It is declared in <stdlib.h>, and accepts one argument, which will be the *exit status* of the program. EXIT_FAILURE is a code, also defined by <stdlib.h>, which indicates that the program failed. Success is indicated by a code of EXIT_SUCCESS, or simply 0. (These values can also be returned from main(); calling exit with a particular status value is essentially equivalent to returning that same status value from main.)

Self Assessment Questions

5.The companion function to putchar is putc, and the file pointer

argument comes first. (True/False)
6.Besides the file pointers which we explicitly open by calling fopen, there
are also _____ predefined streams.
7.getchar() is the same as getc(stdin).(True/False)

13.5 Error Handling during I/O operations

The standard I/O functions maintain two indicators with each open stream to show the end-of-file and error status of the stream. These can be interrogated and set by the following functions:

```
#include <stdio.h>
void clearerr(FILE *stream);
int feof(FILE *stream);
int ferror(FILE *stream);
void perror(const char *s);
```

clearerr clears the error and EOF indicators for the stream.
feof returns non-zero if the stream's EOF indicator is set, zero otherwise. **ferror** returns non-zero if the stream's error indicator is set, zero otherwise.

perror prints a single-line error message on the program's standard output, prefixed by the string pointed to by s, with a colon and a space appended. The error message is determined by the value of errno and is intended to give some explanation of the condition causing the error. For example, this program produces the error message shown:
#include <stdio.h>

```
#include <stdlib.h>
main(){

fclose(stdout);
if(fgetc(stdout) >= 0){
fprintf(stderr, "What - no error!\n");
exit(EXIT_FAILURE);
}
perror("fgetc");
exit(EXIT_SUCCESS); }

/* Result */
fgetc: Bad file number
```

Self Assessment Questions

8._____ clears the error and EOF indicators for the stream.

9._____ returns non-zero if the stream's error indicator is set, zero otherwise.

13.6 Random Access to files

The file I/O routines all work in the same way; unless the user takes explicit steps to change the file position indicator, files will be read and written sequentially. A read followed by a write followed by a read (if the file was opened in a mode to permit that) will cause the second read to start immediately following the end of the data just written. (Remember that stdio insists on the user inserting a buffer-flushing operation between each element of a read-write-read cycle.)

To control this, the Random Access functions allow control over the implied read/write position in the file. The file position indicator is moved without the need for a read or a write, and indicates the byte to be the subject of the next operation on the file.

Three types of function exist which allow the file position indicator to be examined or changed. Their declarations and descriptions follow. #include <stdio.h>

```
/* return file position indicator */
long ftell(FILE *stream);
int fgetpos(FILE *stream, fpos_t *pos);

/* set file position indicator to zero */
void rewind(FILE *stream);

/* set file position indicator */
int fseek(FILE *stream, long offset, int ptrname);
int fsetpos(FILE *stream, const fpos_t *pos);
```

ftell returns the current value (measured in characters) of the file position indicator if stream refers to a binary file. For a text file, a 'magic' number is returned, which may only be used on a subsequent call to **fseek** to reposition to the current file position indicator. On failure, -1L is returned and **errno** is set.

rewind sets the current file position indicator to the start of the file indicated by stream. The file's error indicator is reset by a call of rewind. No value is returned.

fseek allows the file position indicator for stream to be set to an arbitrary value (for binary files), or for text files, only to a position obtained from ftell, as follows:

In the general case, the file position indicator is set to offset bytes (characters) from a point in the file determined by the value of ptrname.

Offset may be negative. The values of ptrname may be SEEK_SET, which sets the file position indicator relative to the beginning of the file, SEEK_CUR, which sets the file position indicator relative to its current value, and SEEK_END, which sets the file position indicator relative to the end of the file. The latter is not necessarily guaranteed to work properly on binary streams.

For text files, offset must either be zero or a value returned from a previous call to ftell for the same stream, and the value of ptrname must be SEEK_SET.

fseek clears the end of file indicator for the given stream and erases the memory of any ungetc. It works for both input and output.
Zero is returned for success, non-zero for a forbidden request.

Note that for ftell and fseek it must be possible to encode the value of the file position indicator into a long. This may not work for very long files, so the Standard introduces fgetpos and fsetpos which have been specified in a way that removes the problem.

fgetpos stores the current file position indicator for stream in the object pointed to by pos. The value stored is ‘magic’ and only used to return to the specified position for the same stream using fsetpos.

fsetpos works as described above, also clearing the stream's end-of-file indicator and forgetting the effects of any ungetc operations.

For both functions, on success, zero is returned; on failure, non-zero is returned and errno is set.

Self Assessment Questions

10._____ function returns the current value (measured in characters) of the file position indicator if stream refers to a binary file. 11.**fseek** allows the file position indicator for stream to be set to an arbitrary value. (True/False)

12._____ stores the current file position indicator for stream in the object pointed to by pos.

13.7 Command Line Arguments

We've mentioned several times that a program is rarely useful if it does exactly the same thing every time you run it. Another way of giving a program some variable input to work on is by invoking it with *command line arguments*. It is a parameter supplied to a program when the program is invoked.

(We should probably admit that command line user interfaces are a bit oldfashioned, and currently somewhat out of favor. If you've used Unix or MSDOS, you know what a command line is, but if your experience is confined to the Macintosh or Microsoft Windows or some other Graphical User Interface, you may never have seen a command line. In fact, if you're learning C on a Mac or under Windows, it can be tricky to give your program a command line at all.

C's model of the command line is that it consists of a sequence of words, typically separated by whitespace. Your main program can receive these words as an array of strings, one word per string. In fact, the C run-time startup code is always willing to pass you this array, and all you have to do to receive it is to declare main as accepting two parameters, like this: int main(int argc, char *argv[])

{

...

}

When main is called, argc will be a count of the number of command-line arguments, and argv will be an array ("vector") of the arguments themselves. Since each word is a string which is represented as a pointerto-char, argv is an array-of-pointers-to-char. Since we are not *defining* the argv array, but merely declaring a parameter which references an array somewhere else (namely, in main's caller, the run-time startup code), we do not have to supply an array dimension for argv. (Actually, since functions never receive arrays as parameters in C, argv can also be thought of as a pointer-to- pointer-to-char, or char **. But multidimensional arrays and pointers to pointers can be confusing, and we haven't covered them, so we'll talk about argv as if it were an array.) (Also, there's nothing magic about the names argc and argv. You can give main's two parameters any names you like, as long as they have the appropriate types. The names argc and argv are traditional.)

The first program to write when playing with argc and argv is one which simply prints its arguments:

Program 13.3: To illustrate the use of command line arguments #include <stdio.h>

```
main(int argc, char *argv[])
{
int i;

for(i = 0; i < argc; i++)
printf("arg %d: %s\n", i, argv[i]);
return 0;
}
```

(This program is essentially the Unix or MS-DOS echo command.)

If you run this program, you'll discover that the set of “words" making up the command line includes the command you typed to invoke your program (that is, the name of your program). In other words, argv[0] typically points to the name of your program, and argv[1] is the first argument.

There are no hard-and-fast rules for how a program should interpret its command line. There is one set of conventions for Unix, another for MSDOS, another for VMS. Typically you'll loop over the arguments, perhaps treating some as option flags and others as actual arguments (input files, etc.), interpreting or acting on each one. Since each argument is a string, you'll have to use strcmp or the like to match arguments against any patterns you might be looking for. Remember that argc contains the number of words on the command line, and that argv[0] is the command name, so if argc is 1, there are no arguments to inspect. (You'll never want to look at argv[i], for i >= argc, because it will be a null or invalid pointer.)

A further example of the use of **argc** and **argv** now follows:

Program 13.4: To illustrate the use of command line arguments

```
void main(int argc, char *argv[])
{
if (argc !=2) {
printf("Specify a password");
exit(1);
}
if (!strcmp(argv[1], "password"))
printf("Access Permitted");
else
{
```

```
printf("Access denied");
exit(1);
}
program code here ......
}
```

This program only allows access to its code if the correct password is entered as a command-line argument. There are many uses for commandline arguments and they can be a powerful tool.

As another example, here is a program which copies one or more input files to its standard output. Since “standard output" is usually the screen by default, this is therefore a useful program for displaying files. (It's analogous to the obscurely-named Unix cat command, and to the MS-DOS type command.)

Program 13.5: To copy one or more input files to standard output #include

```
<stdio.h>
main(int argc, char *argv[])
{
int i;
FILE *fp;
int c;

for(i = 1; i < argc; i++)
{
fp = fopen(argv[i], "r");
```

```
if(fp == NULL)
{
fprintf(stderr, "cat: can't open %s\n", argv[i]); continue;
}

while((c = getc(fp)) != EOF) putchar(c);
fclose(fp);
}

return 0;
}
```

As a historical note, the Unix cat program is so named because it can be used to concatenate two files together, like this:

```
cat a b > c
```

This illustrates why it's a good idea to print error messages to stderr, so that they don't get redirected. The “can't open file" message in this example also includes the name of the program as well as the name of the file.

Yet another piece of information which it's usually appropriate to include in error messages is the reason why the operation failed, if known. For operating system problems, such as inability to open a file, a code indicating the error is often stored in the global variable errno. The standard library function strerror will convert an errno value to a human-readable error message string. Therefore, an even more informative error message printout would be

```
fp = fopen(argv[i], "r");
if(fp == NULL)
fprintf(stderr, "cat: can't open %s: %s\n",
```

argv[i], strerror(errno));
If you use code like this, you can #include <errno.h> to get the declaration for errno, and <string.h> to get the declaration for strerror().
Final example program takes two command-line arguments. The first is the name of a file, the second is a character. The program searches the specified file, looking for the character. If the file contains at least one of these characters, it reports this fact. This program uses **argv** to access the file name and the character for which to search.

Program 13.6: To search the specified file, looking for the character

```
/*Search specified file for specified character. */
#include <stdio.h>
#include <stdlib.h>
void main(int argc, char *argv[])
{
FILE *fp; /* file pointer */
char ch;

/* see if correct number of command line arguments */ if(argc !=3) {
printf("Usage: find <filename> <ch>\n");
exit(1);
}

/* open file for input */
if ((fp = fopen(argv[1], "r"))==NULL) { printf("Cannot open file \n");
exit(1);
```

```
}
/* look for character */
while ((ch = getc(fp)) !=EOF) /* where getc() is a */
if (ch== *argv[2]) { /*function to get one char*/
printf("%c found",ch); /* from the file */
break;
}
fclose(fp);
}
```

Self Assessment Questions

13.Command line argument is a parameter supplied to a program when the program is invoked. (True/False)

14.In the command line arguments of main(), argv is an array-of-pointersto-_____.

15.The _____ is a parameter supplied to a program when the program is invoked.

13.8 Summary

"Standard input" and "standard output" are two predefined I/O *streams* which are implicitly available to us. To actually open a file, and receive the "token" which you'll store in your file pointer variable, you call **fopen. fopen** accepts a file name (as a string) and a *mode* value indicating among other things whether you intend to read or write this file. The *file pointer* which keeps track (both for your sake and the library's) of which file you're talking about.

For each of the I/O library functions we've there's a companion function which accepts an additional file pointer argument telling it where to read from or write to. The standard I/O functions maintain two indicators with each

open stream to show the end-of-file and error status of the stream. The Random Access functions allow control over the implied read/write position in the file. The *command line argument* is a parameter supplied to a program when the program is invoked.

13.9 Terminal Questions

1.The skeletal outline of a C program is shown below: #include <stdio.h>
main()
{

FILE *pt1, *pt2; char name[20];

pt1 = fopen(“sample.old”, “r”); pt2= fopen(“sample.new”,”w”); ……..
fclose(pt1);
fclose(pt2);
}

a)Read the string represented by name from the data file sample.old. b)Display it on the screen
c)Enter the updated string.
d)Write the new string to the data file sample.new
2.What is a command line argument and what is its use? Explain

13.10 Answers to Self Assessment Questions

1. stdio.h
2. true
3.File pointer
4. True
5. false
6. three
7. True
8. clearerr
9. ferror
10. ftell
11. true
12. fgetpos
13. true
14. char
15.Command line argument

13.11 Answers to Terminal Questions

1.a) fscanf(pt1, “%s”, name);
b)printf(“Name: %s\n”, name);

c)puts(“New Name:”);
gets(name);
d)fputs(name, pt2);
2.Command Line argument is a parameter supplied to a program when the program is invoked. (Refer 13.7 for more details)

13.12 Exercises

1.Describe the use of functions **getc()** and **putc().**
2.What is meant by opening a data file? How is this accomplished?

3.Distinguish between the following functions:
a)getc and getchar
b)printf and fprintf
c)feof and ferror
4.Explain the general syntax of the **fseek** function.

5.Write a program to copy the contents of one file to another.

6.Two data files DATA1 and DATA2 contains sorted list of integers. Write a program to produce a third file DATA which holds the single sorted , merged list of these two lists. Use command line arguments to specify

the file names.
7.The skeletal outline of an C program is shown below.

```
#include <stdio.h>
main()
{
FILE *pt1, *pt2;
int a;
float b;
char c;
pt1 = fopen(“sample.old”, “r”);
pt2= fopen (“sample.new”,”w”);
……..
fclose(pt1);
fclose(pt2);
}
```

a)Read the values of a,b and c from the data file sample.old.

b)Display each value on the screen and enter an updated value. c)Write the new values to the data file sample.new. Format the floatingpoint value so that not more than two decimals are written to

sample.new.

Unit 14 Advanced Data Representation

Structure:

14.1 Introduction

Objectives
14.2 Exploring Data Representation
14.3 Abstract Data Types
14.4 Stack as an Abstract Data Type
Array Implementation of a Stack
Implementation of a Stack Using Linked Representation
Applications of Stacks
14.5 Queue as an Abstract Data Type
Array Implementation of a Queue
Implementation of a Queue Using Linked List Representation
Circular Queues
Applications of Queues
14.6 Summary
14.7 Terminal Questions
14.8 Answers to Self Assessment Questions
14.9 Answers to Terminal Questions
14.10 Exercises

14.1 Introduction

Learning a computer language is like learning music, carpentry, or engineering. At first, you work with the tools of the trade, playing scales, learning which end of the hammer to hold and which end to avoid, solving countless problems involving falling, sliding, and balanced objects. Acquiring and practicing skills is what you've been doing so far in this book, learning to create variables, structures, functions, and the like. Eventually, however, you move to a higher level in

which using the tools is second nature and the real challenge is designing and creating a project. You develop an ability to see the project as a coherent whole. This unit concentrates on that higher level. You may find the material covered here a little more challenging than the preceding units, but you may also find it more rewarding as it helps you move from the role of apprentice to the role of craftsperson.
We'll start by examining a vital aspect of program design: the way a program represents data. Often the most important aspect of program development is finding a good representation of the data manipulated by that program. Getting data representation right can make writing the rest of the program simple. By now you've seen C's built-in data types: simple variables, arrays, pointers, structures, and unions.

Finding the right data representation, however, often goes beyond simply selecting a type. You should also think about what operations will be necessary. That is, you should decide how to store the data, and you should define what operations are valid for the data type. For example, C implementations typically store both the C int type and the C pointer type as integers, but the two types have different sets of valid operations. You can multiply one integer by another, for example, but you can't multiply a pointer by a pointer. You can use the * operator to dereference a pointer, but that operation is meaningless for an integer. The C language defines the valid operations for its fundamental types. However, when you design a scheme to represent data, you might need to define the valid operations yourself. In C, you can do so by designing C functions to represent the desired operations. In short, then, designing a data type consists of deciding on how to store the data and of designing a set of functions to manage the data. You will also look at some algorithms, recipes for manipulating data. As a programmer, you will acquire a repertoire of such recipes that you apply over and over again to similar problems.

This unit looks into the process of designing data types, a process that matches algorithms to data representations. In it, you'll meet some common data forms, such as the stacks and queues.

You'll also be introduced to the concept of the abstract data type (ADT). An ADT packages methods and data representations in a way that is problemoriented rather than language-oriented. After you've designed an ADT, you can easily reuse it in different circumstances. Understanding ADTs prepares you conceptually for entering the world of object-oriented programming (OOP) and the C++ language.

Objectives:
After studying this unit, you should be able to:
use C to represent a variety of data types
implement the new algorithms and increasing your ability to develop

programs conceptually
implement abstract data types (ADTs)
implement two Abstract Data Types-stacks and queues discuss the applications of stacks and queues

14.2 Exploring Data Representation

Let's begin by thinking about data. Suppose you had to create an address book program. What data form would you use to store information? Because there's a variety of information associated with each entry, it makes sense to represent each entry with a structure. How do you represent several entries? With a standard array of structures? With a dynamic array? With some other form? Should the entries be alphabetized? Should you be able to search through the entries by zip code? by area code? The actions you want to perform might affect how you decide to store the information. In short, you have a lot of design decisions to make before plunging into coding.

How would you represent a bitmapped graphics image that you want to store in memory? A bitmapped image is one in which each pixel on the screen is set individually. In the days of black-and-white screens, you could use one computer bit (1 or 0) to represent one pixel (on or off), hence the name bitmapped. With color monitors, it takes more than one bit to describe a single pixel. For example, you can get 256 colors if you dedicate 8 bits to each pixel. Now the industry has moved to 65,536 colors (16 bits per pixel), 16,777,216 colors (24 bits per pixel), 2,147,483,648 colors (32 bits per pixel), and even beyond. If you have 16 million colors and if your monitor has a resolution of 1024x768, you'll need 18.9 million bits (2.25 MB) to represent a single screen of bitmapped graphics. Is this the way to go, or can you develop a way of compressing the information? Should this compression be lossless (no data lost) or lossy (relatively unimportant data lost)? Again, you have a lot of design decisions to make before diving into coding. Let's tackle a particular case of representing data. Suppose you want to write a program that enables you to enter a list of all the movies (including videotapes) you've seen in a year. For each movie, you'd like to record a variety of information, such as the title, the year it was released, the director, the lead actors, the length, the kind of film (comedy, science fiction, romance, drivel, and so forth), your evaluation, and so on. That suggests using a structure for each film and an array of structures for the list. To simplify matters, let's limit the structure to two members: the film title and your evaluation, a ranking on a 0 to 10 scale. Program 6.1 shows a bare- bones implementation using this approach.

Program 14.1: The *films1.c* Program

```
/* films1.c -- using an array of structures */
#include <stdio.h>
#define TSIZE 45 /* size of array to hold title */
#define FMAX 5 /* maximum number of film titles */
struct film {
char title[TSIZE];
```

```
int rating;
} ;

int main(void)
{
struct film movies[FMAX];
int i = 0;
int j;

puts(“Enter first movie title:”);
while (i < FMAX && gets(movies[i].title) != NULL && movies[i].title[0] != '\0')
{
puts("Enter your rating <0-10>:");
scanf("%d", &movies[i++].rating);
while(getchar() != '\n')
continue;
puts("Enter next movie title (empty line to stop):"); }
if (i == 0)
printf("No data entered. ");
else
printf ("Here is the movie list:\n");
for (j = 0; j < i; j++)
printf("Movie: %s Rating: %d\n", movies[j].title,
movies[j].rating);
printf("Bye!\n");
```

```
return 0;
}
```

The program creates an array of structures and then fills the array with data entered by the user. Entry continues until the array is full (the FMAX test), until end of file (the NULL test) is reached, or until the user presses the Enter key at the beginning of a line (the '\0' test).

This formulation has some problems. First, the program will most likely waste a lot of space because most movies don't have titles 40 characters long, but some movies do have long titles. Second, many people will find the limit of five movies a year too restrictive. Of course, you can increase that limit, but what would be a good value? Some people see 500 movies a year, so you could increase FMAX to 500, but that still might be too small for some, yet it might waste enormous amounts of memory for others. Also, some compilers set a default limit for the amount of memory available for automatic storage class variables such as movies, and such a large array could exceed that value. You can fix that by making the array a static or external array or by instructing the compiler to use a larger stack, but that's not fixing the real problem.

The real problem here is that the data representation is too inflexible. You have to make decisions at compile time that are better made at runtime. This suggests switching to a data representation that uses dynamic memory allocation. You could try something like this:

```
#define TSIZE 45 /* size of array to hold title */
struct film {
char title[TSIZE];
int rating;
} ;
...
int n, i;
```

```
struct film * movies; /* pointer to a structure */
...
printf("Enter the maximum number of movies you'll enter:\n");
scanf("%d", &n);
movies = (struct film *) malloc(n * sizeof(struct film));
```

Here, you can use the pointer movies just as though it were an array name: while (i < FMAX && gets(movies[i].title) != NULL &&
movies[i].title[0] != '\0')

By using malloc(), you can postpone determining the number of elements until the program runs, so the program need not allocate 500 elements if only 20 are needed. However, it puts the burden on the user to supply a correct value for the number of entries.

Self Assessment Questions

1.A _____ image is one in which each pixel on the screen is set individually.

2.The best solution to allocate memory locations for the objects when the

actual number of locations is not known in advance is by using _____ memory allocations.

14.3 Abstract Data Types

An Abstract Data Type (ADT) is a data type that is organized in such a way that the specification of the objects and the specification of the operations on the objects is separated from the representation of the objects and implementation of the operations.

Some programming languages provide explicit mechanisms to support the distinction between specification and implementation. Although C does not have an explicit mechanism for implementing ADTs, it is still possible and desirable to design your data types using the same notion.

How does the specification of the operations of an ADT differ from the implementation of the operations? The specification consists of the names of every function, the type of its arguments, and the type of its result. There should also be a description of what the function does, but without appealing to internal representation or implementation details. This requirement is quite important, and it implies that an abstract data type is implementation independent. Furthermore, it is possible to classify the functions of a data type into several categories:
1)**Creator/constructor:** These functions create a new instance of the designated type.
2)**Transformers:** These functions also create an instance of the designated type, generally by using one or more other instances. 3)**Observers/reporters:** These functions provide information about an instance of the type, but they do not change the instance.

Typically, an ADT definition will include at least one function from each of these three categories.

14.4 Stack as an Abstract Data Type

A stack is simply a list of elements with insertions and deletions permitted at one end – called the stack top. That means that it is possible to remove elements from a stack in reverse order from the insertion of elements into the stack. Thus, a stack data structure exhibits the LIFO (last in first out) property. Push and pop are the operations that are provided for insertion of an element into the stack and the removal of an element from the stack, respectively. Shown in Figure 14.1 are the effects of push and pop operations on the stack.

B C
A A A A A

Empty stack Push(A) Push(B) Pop() Push(C) Pop()

Indicates stack top position

Figure 14.1: Stack operations

Since a stack is basically a list, it can be implemented by using an array or by

using a linked representation.

14.4.1 Array Implementation of a Stack

When an array is used to implement a stack, the push and pop operations are realized by using the operations available on an array. The limitation of an array implementation is that the stack cannot grow and shrink dynamically as per the requirement.

Program 14.2: A complete C program to implement a stack using an array appears here:

```
#include <stdio.h>
#define MAX 10 /* The maximum size of the stack */
#include <stdlib.h>
void push(int stack[], int *top, int value)
{
if(*top < MAX )
{
*top = *top + 1;
stack[*top] = value;
```

```
}
else
{
printf("The stack is full can not push a value\n");
exit(0);
}
}
void pop(int stack[], int *top, int * value)
{
if(*top >= 0 )
{
*value = stack[*top];
*top = *top - 1;
}
else
{
printf("The stack is empty can not pop a value\n");
exit(0);
}
}

void main()
{
int stack[MAX];
```

```
int top = -1;
int n,value;
do
{
do
{
printf("Enter the element to be pushed\n"); scanf("%d",&value);
push(stack,&top,value);
printf("Enter 1 to continue\n");
scanf("%d",&n);
} while(n == 1);

printf("Enter 1 to pop an element\n");
scanf("%d",&n);
while( n == 1)
{
pop(stack,&top,&value);
printf("The value popped is %d\n",value);
printf("Enter 1 to pop an element\n");
scanf("%d",&n);
}
printf("Enter 1 to continue\n");
scanf("%d",&n);
```

```
} while(n == 1);
}
```

14.4.2 Implementation of a Stack Using Linked Representation Initially the list is empty, so the top pointer is NULL. The push function takes a pointer to an existing list as the first parameter and a data value to be pushed as the second parameter, creates a new node by using the data value, and adds it to the top of the existing list. A pop function takes a pointer to an existing list as the first parameter, and a pointer to a data object in which the popped value is to be returned as a second parameter. Thus it retrieves the value of the node pointed to by the top pointer, takes the top point to the next node, and destroys the node that was pointed to by the top.

If this strategy is used for creating a stack with the four data values: 10, 20, 30, and 40, then the stack is created as shown in Figure 14.2. Initially top=NULL.

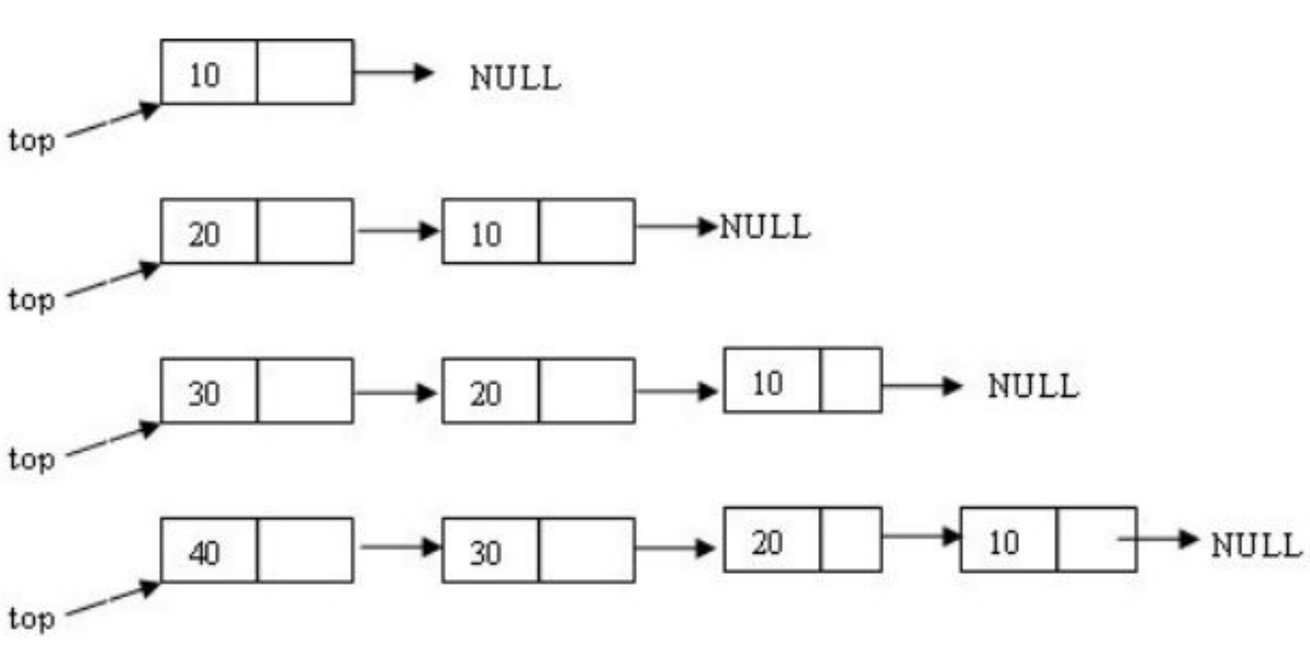


Figure 14.2: Linked stack

Program 14.3: A complete C program for implementation of a stack using the linked list is given here

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

```
# include <stdlib.h>
struct node
{
int data;
struct node *link;
};
struct node *push(struct node *p, int value)
{
struct node *temp;
temp=(struct node *)malloc(sizeof(struct node));
/* creates new node
using data value
passed as parameter */
if(temp==NULL)
{
printf("No Memory available Error\n"); exit(0);
}
temp->data = value;
temp->link = p;
p = temp;
return(p);
}
```

```
struct node *pop(struct node *p, int *value) {
struct node *temp;
if(p==NULL)
{

printf(" The stack is empty can not pop Error\n"); exit(0);
}
*value = p->data;
temp = p;
p = p->link;
free(temp);
return(p);
}

void main()
{
struct node *top = NULL;
int n,value;
do
{
do
{
printf("Enter the element to be pushed\n"); scanf("%d",&value);
top = push(top,value);
```

```
printf("Enter 1 to continue\n"); scanf("%d",&n);
} while(n == 1);

printf("Enter 1 to pop an element\n");
scanf("%d",&n);
while( n == 1)
{
top = pop(top,&value);
printf("The value popped is %d\n",value);
printf("Enter 1 to pop an element\n");
scanf("%d",&n);
}
printf("Enter 1 to continue\n");
scanf("%d",&n);
} while(n == 1);
}
```

14.4.3 Applications of Stacks

One of the applications of the stack is in expression evaluation. A complex assignment statement such as a = b + c*d/ e–f may be interpreted in many different ways. Therefore, to give a unique meaning, the precedence and associativity rules are used. But still it is difficult to evaluate an expression by computer in its present form, called the infix notation. In *infix notation*, the binary operator comes in between the operands. A unary operator comes before the operand. To get it evaluated, it is first converted to the postfix form, where the operator comes after the operands. For example, the postfix form for the expression a*(b–c)/d is abc – *d/. A good thing about postfix expressions is that they do not require

any precedence rules or parentheses for unique definition. So, evaluation of a postfix expression is possible using a stack-based algorithm.

Program 14.4: Program to convert an infix expression to prefix form #include <stdio.h>

```
#include <string.h>
#include <ctype.h>
#define N 80
typedef enum {FALSE, TRUE} bool;
#include "stack.h"
#include "queue.h"
#define NOPS 7

char operators [] = "()/^*+-";
int priorities[] = {4,4,3,2,2,1,1};
char associates[] = " RLLLL";

char t[N]; char *tptr = t; // this is where prefix will be saved.
int getIndex( char op ) {
/*
*returns index of op in operators.
*/
int i;
for( i=0; i<NOPS; ++i )
if( operators[i] == op )
```

```
return i;
return -1;
}
int getPriority( char op ) {
/*
*returns priority of op.
*/
return priorities[ getIndex(op) ];
}

char getAssociativity( char op ) { /*
*returns associativity of op. */
return associates[ getIndex(op) ]; }
void processOp( char op, queue *q, stack *s ) {
/*
*performs processing of op.
*/
switch(op) {
case ')':
printf( "\t S pushing )...\n" );
sPush( s, op );
break;
case '(':
while( !qEmpty(q) ) {
```

```
*tptr++ = qPop(q);
printf( "\tQ popping %c...\n", *(tptr-1) );
}
while( !sEmpty(s) ) {
char popop = sPop(s);
printf( "\tS popping %c...\n", popop );
if( popop == ')' )
break;
*tptr++ = popop;
}
break;
default: {
int priop; // priority of op.
char topop; // operator on stack top.
int pritop; // priority of topop.
char asstop; // associativity of topop.
while( !sEmpty(s) ) {
priop = getPriority(op);
topop = sTop(s);
pritop = getPriority(topop);
asstop = getAssociativity(topop);
if( (pritop < priop) || (pritop == priop && asstop == 'L') || (topop == ')' ))// IMP. break;
while( !qEmpty(q) ) {
*tptr++ = qPop(q);
```

```
printf( "\tQ popping %c...\n", *(tptr-1) );
}
*tptr++ = sPop(s);
printf( "\tS popping %c...\n", *(tptr-1) );
}
printf( "\tS pushing %c...\n", op );
sPush( s, op );
break;
}
}
}
bool isop( char op ) {
/* is op an operator? */
return (getIndex(op) != -1);
}
char *in2pre( char *str ) { /* returns valid infix expr in str to prefix. */ char *sptr;
queue q = {NULL};
stack s = NULL;
char *res = (char *)malloc( N*sizeof(char) );
char *resptr = res;
tptr = t;
for( sptr=str+strlen(str)-1; sptr!=str-1; -sptr ) {
printf( "processing %c tptr-t=%d...\n", *sptr, tptr-t );
if( isalpha(*sptr) ) // if operand.
```

```
qPush( &q, *sptr );
else if( isop(*sptr) ) // if valid operator.
processOp( *sptr, &q, &s );
else if( isspace(*sptr) ) // if whitespace.
;
else {

fprintf( stderr, "ERROR:invalid char %c.\n", *sptr ); return "";

} }
while( !qEmpty(&q) ) {
*tptr++ = qPop(&q);
printf( "\tQ popping %c...\n", *(tptr-1) );
}
while( !sEmpty(&s) ) {
*tptr++ = sPop(&s);
printf( "\tS popping %c...\n", *(tptr-1) );
}
*tptr = 0;
printf( "t=%s.\n", t );
for( -tptr; tptr!=t-1; -tptr ) {
*resptr++ = *tptr;
}
*resptr = 0;
```

```
return res;
}
int main() {
char s[N];

puts( "enter infix freespaces max 80." );
gets(s);
while(*s) {
puts( in2pre(s) );
gets(s);
}

return 0;
}
```

Explanation

1.In an infix expression, a binary operator separates its operands (a unary

operator precedes its operand). In a postfix expression, the operands of an operator precede the operator. In a prefix expression, the operator precedes its operands. Like postfix, a prefix expression is parenthesisfree, that is, any infix expression can be unambiguously written in its

prefix equivalent without the need for parentheses.

2.To convert an infix expression to reverse-prefix, it is scanned from right to left. A queue of operands is maintained noting that the order of operands in infix and prefix remains the same. Thus, while scanning the infix expression,

whenever an operand is encountered, it is pushed in a queue. If the scanned element is a right parenthesis (')'), it is pushed in a stack of operators. If the scanned element is a left parenthesis ('('), the queue of operands is emptied to the prefix output, followed by the popping of all the operators up to, but excluding, a right parenthesis in the operator stack.

3.If the scanned element is an arbitrary operator o, then the stack of operators is checked for operators with a greater priority than o. Such operators are popped and written to the prefix output after emptying the operand queue. The operator o is finally pushed to the stack.

4.When the scanning of the infix expression is complete, first the operand queue, and then the operator stack, are emptied to the prefix output. Any whitespace in the infix input is ignored. Thus the prefix output can be reversed to get the required prefix expression of the infix input.

Example

If the infix expression is a*b + c/d, then different snapshots of the algorithm, while scanning the expression from right to left, are shown in Table 14.1.

Table 14.1: Scanning the infix expression a*b+c/d from right to left Remaining Scanned Queue of Stack of Prefix Step expression element operands operators output 0 a*b+c/d nil empty empty nil

1 a*b+c/ D d empty nil
2 a*b+c / d / nil
3 a*b+ C d c / nil

4 a*b + empty + dc/
5 a* B b + dc/
6 A * b *+dc/

7 Nil A b a *+dc/

8 Nil nil empty empty dc/ba*+

The final prefix output that we get is dc/ba*+ whose reverse is +*ab/cd, which is the prefix equivalent of the input infix expression a*b+c*d. Note that all the operands are simply pushed to the queue in steps 1, 3, 5, and 7. In step 2, the operator / is pushed to the empty stack of operators.

In step 4, the operator + is checked against the elements in the stack. Since / (division) has higher priority than + (addition), the queue is emptied to the prefix output (thus we get 'dc' as the output) and then the operator / is written (thus we get 'dc/' as the output). The operator + is then pushed to the stack. In step 6, the operator * is checked against the stack elements. Since * (multiplication) has a higher priority than + (addition), * is pushed to the stack. Step 8 signifies that all of the infix expression is scanned. Thus, the queue of operands is emptied to the prefix output (to get 'dc/ba'), followed by the emptying of the stack of operators (to get 'dc/ba*+').

Points to remember

1.A prefix expression is parenthesis-free.

2.To convert an infix expression to the prefix equivalent, it is scanned from right to left. The prefix expression we get is the reverse of the required prefix equivalent.

3.Conversion of infix to prefix requires a queue of operands and a stack, as in the conversion of an infix to postfix.

4.The order of operands in a prefix expression is the same as that in its infix equivalent.

5.If the scanned operator o1 and the operator o2 at the stack top have the same priority, then the associativity of o2 is checked. If o2 is rightassociative, it is popped from the stack.

Self Assessment Questions:

3.Stack is also called _____ data structure.

4.The drawback of implementing a stack using an array is:
(a)Stack cannot grow
(b)Stack can grow
(c)Array cannot grow even if stack is empty
(d)Array can grow infinitely
5.Postfix expression is used for Evaluation of Expressions. (True/False)

14.5 Queue as an Abstract Data Type

A *queue* is also a list of elements with insertions permitted at one end – called the rear, and deletions permitted from the other end – called the front. This means that the removal of elements from a queue is possible in the same order in which the insertion of elements is made into the queue. Thus, a queue data structure exhibits the *FIFO* (*first in first out*) property. Insert and delete are the operations that are provided for insertion of elements into the queue and the removal of elements from the queue, respectively. Shown in Figure 14.3 are the effects of insert and delete operations on the queue. Initially both front and rear are initialized to -1.

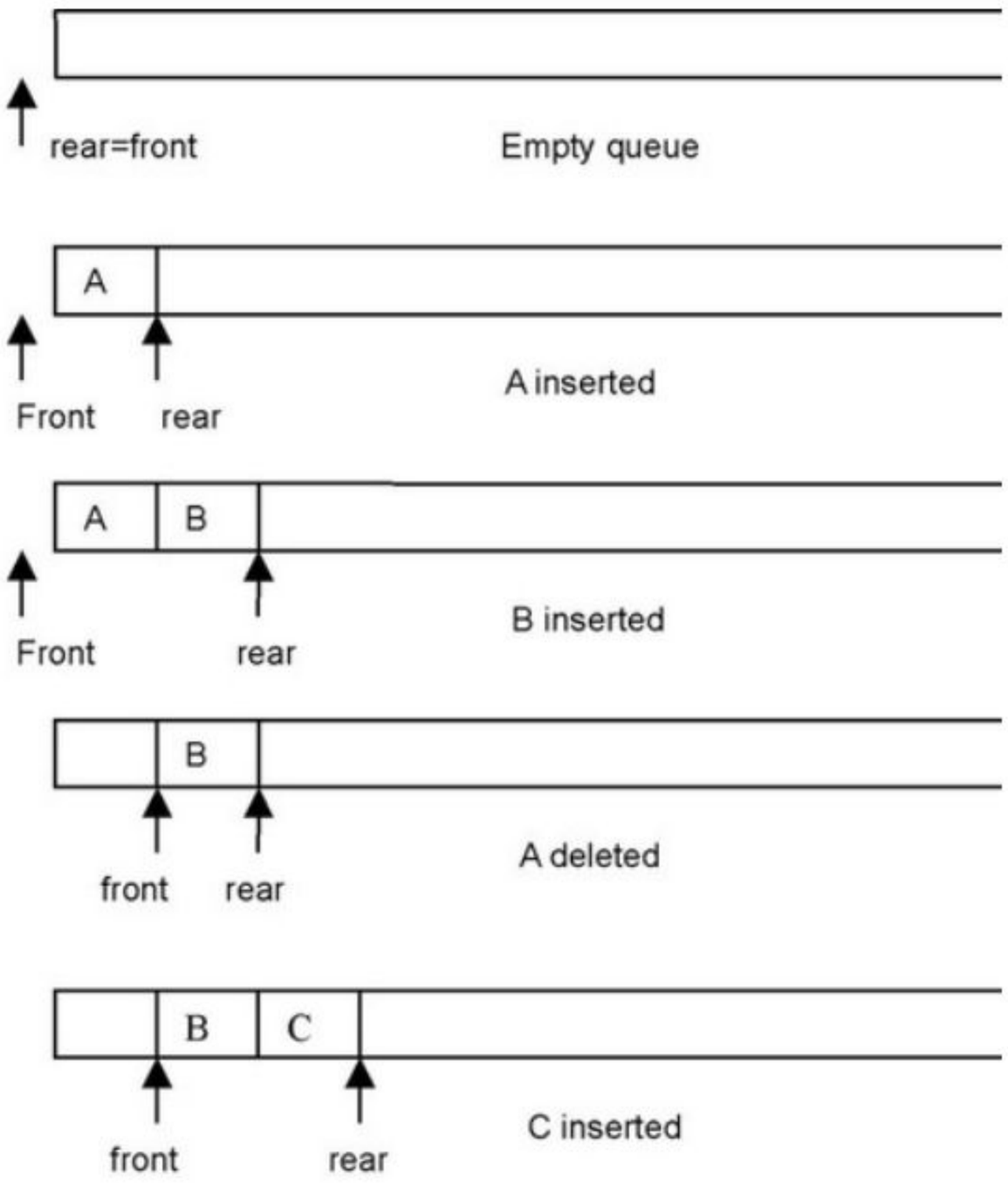


Figure 14.3: Queue operation

14.5.1 Array Implementation of a Queue

When an array is used to implement a queue, then the insert and delete operations are realized using the operations

available on an array. The limitation of an array implementation is that the queue cannot grow and shrink dynamically as per the requirement.

Program 14.5: A complete C program to implement a queue by using an array is shown here

```
#include <stdio.h>
#define MAX 10 /* The maximum size of the queue */
#include <stdlib.h>

void insert(int queue[], int *rear, int value)
{
if(*rear < MAX-1)
{
*rear= *rear +1;
queue[*rear] = value;
}
else
{
printf("The queue is full can not insert a value\n");
exit(0);
}
}

void delete(int queue[], int *front, int rear, int * value)
{
if(*front == rear)
```

```
{
printf("The queue is empty can not delete a value\n");
exit(0);
}
*front = *front + 1;
*value = queue[*front];
}

void main()
{
int queue[MAX];
int front,rear;
int n,value;
front=rear=(-1);
do
{
do
{
printf("Enter the element to be inserted\n"); scanf("%d",&value);
insert(queue,&rear,value);
printf("Enter 1 to continue\n");
scanf("%d",&n);
} while(n == 1);
```

```
printf("Enter 1 to delete an element\n");
scanf("%d",&n);
while( n == 1)
{
delete(queue,&front,rear,&value);
printf("The value deleted is %d\n",value);
printf("Enter 1 to delete an element\n");
scanf("%d",&n);
}
printf("Enter 1 to continue\n");
scanf("%d",&n);
} while(n == 1);
}
```

14.5.2 Implementation of a Queue using Linked List Representation

Initially, the list is empty, so both the front and rear pointers are NULL. The insert function creates a new node, puts the new data value in it, appends it to an existing list, and makes the rear pointer point to it. A delete function checks whether the queue is empty, and if not, retrieves the data value of the node pointed to by the front, advances the front, and frees the storage of the node whose data value has been retrieved.

If the above strategy is used for creating a queue with four data values – 10, 20, 30, and 40, the queue gets created as shown in Figure 14.4.

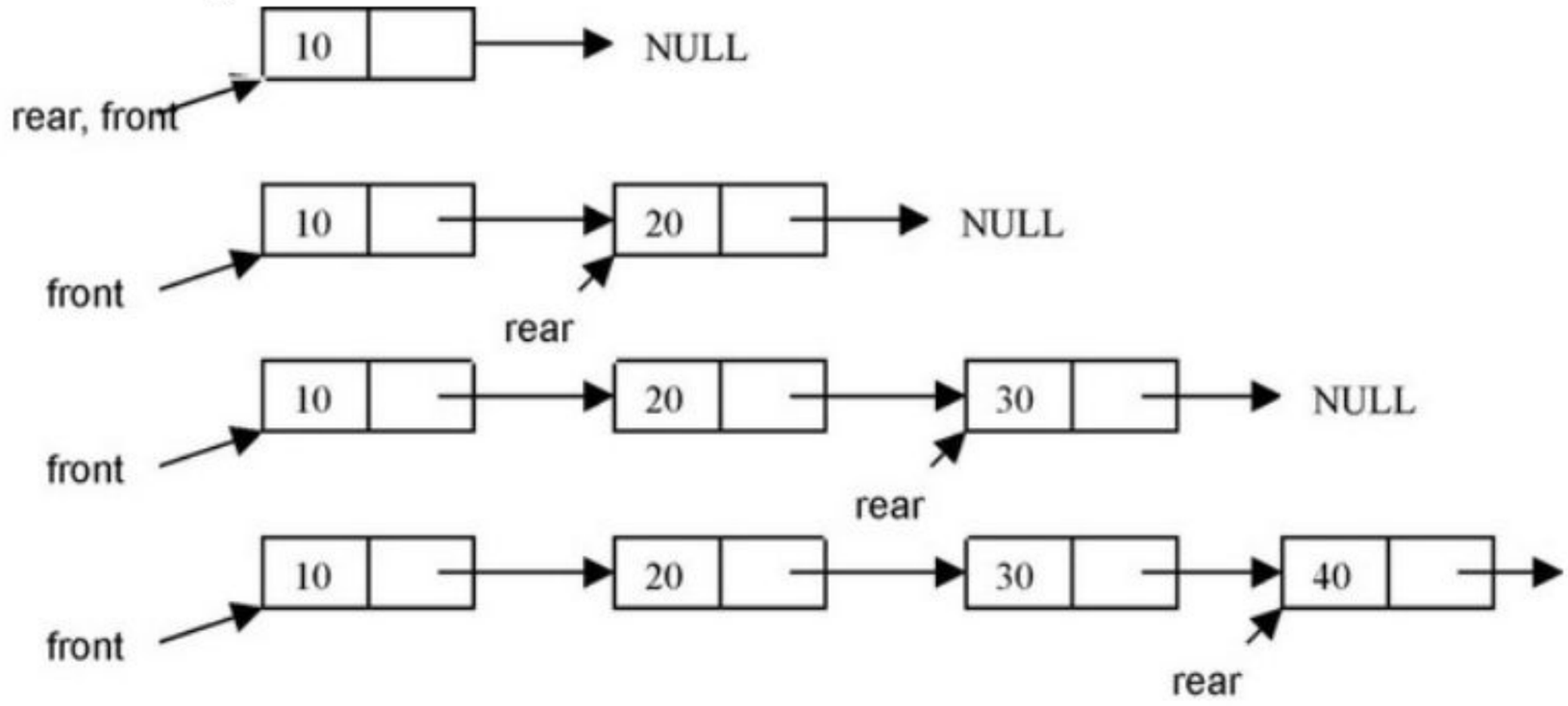


Figure 14.4: Linked queue

Program 14.6: A complete C program for implementation of a queue using the linked list is shown here

```
# include <stdio.h>
# include <stdlib.h>
struct node
{
int data;
struct node *link;
};
```

```
void insert(struct node **front, struct node **rear, int value)
{
struct node *temp;
temp=(struct node *)malloc(sizeof(struct node));
/* creates new node using data value passed as parameter */
if(temp==NULL)
{
printf("No Memory available Error\n");
exit(0);
}
temp->data = value;
temp->link=NULL;
if(*rear == NULL) {
*rear = temp;
*front = *rear;
}
else
{
(*rear)->link = temp; *rear = temp;
}
}

void delete(struct node **front, struct node **rear, int *value) {
struct node *temp;
```

```
if((*front == *rear) && (*rear == NULL))
{
printf(" The queue is empty cannot delete Error\n"); exit(0);
}
*value = (*front)->data;
temp = *front;
*front = (*front)->link;
if(*rear == temp)
*rear = (*rear)->link;
free(temp);
}

void main()
{
struct node *front=NULL,*rear = NULL; int n,value;
do
{
do
{
printf("Enter the element to be inserted\n"); scanf("%d",&value);
insert(&front,&rear,value);
printf("Enter 1 to continue\n");
scanf("%d",&n);
} while(n == 1);
```

```
printf("Enter 1 to delete an element\n");
scanf("%d",&n);
while( n == 1)
{
delete(&front,&rear,&value);
printf("The value deleted is %d\n",value);
printf("Enter 1 to delete an element\n");
scanf("%d",&n);
}
printf("Enter 1 to continue\n");
scanf("%d",&n);
} while(n == 1);
}
```

14.5.3 Circular Queues

The problem with the previous implementation is that the insert function gives a queue-full signal even if a considerable portion is free. This happens because the queue has a tendency to move to the right unless the 'front' catches up with the 'rear' and both are reset to 0 again (in the delete procedure). To overcome this problem, the elements of the array are required to shift one position left whenever a deletion is made. But this will make the deletion process inefficient. Therefore, an efficient way of overcoming this problem is to consider the array to be circular, as shown in Figure 14.5. Initially both front and rear are intitialized to 0.

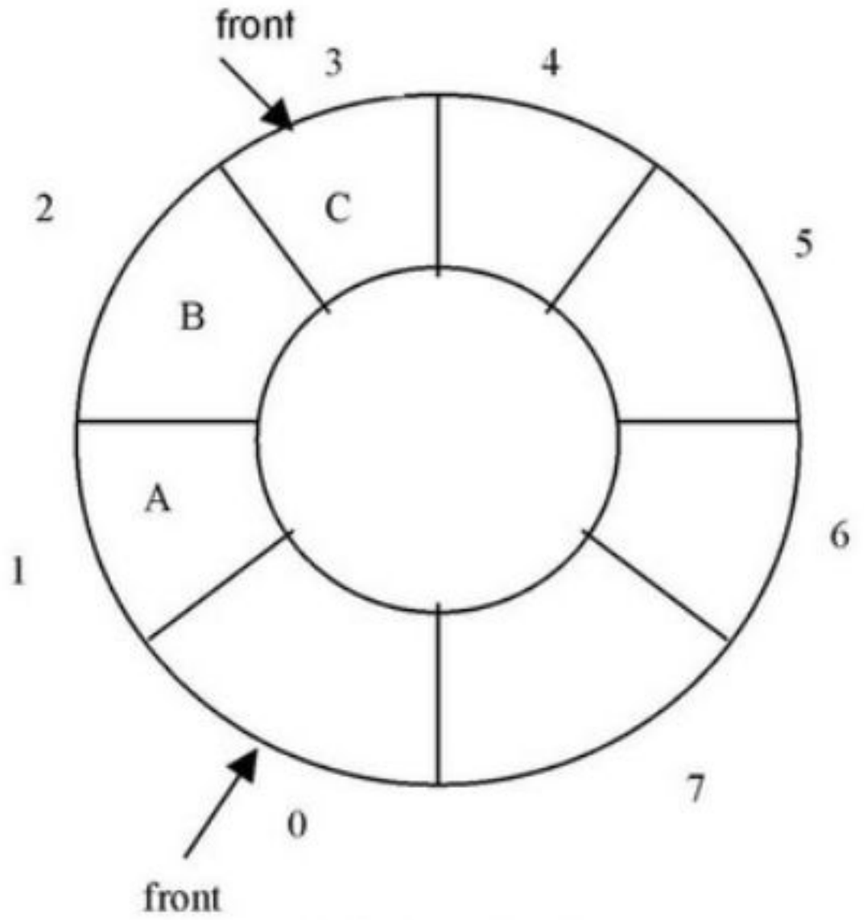


Figure 14.5: Circular queues

Program 14.7: A complete C program for implementation of a circular queue is shown here

```
#include <stdio.h>
#define MAX 10 /* The maximum size of the queue */
#include <stdlib.h>

void insert(int queue[], int *rear, int front, int value)
{
*rear= (*rear +1) % MAX;
if(*rear == front)
{
```

```
printf("The queue is full can not insert a value\n");
exit(0);
}
queue[*rear] = value;
}
void delete(int queue[], int *front, int rear, int * value)
{
if(*front == rear)
{
printf("The queue is empty can not delete a value\n");
exit(0);
}
*front = (*front + 1) % MAX; *value = queue[*front]; }
void main()
{
int queue[MAX];
int front,rear;
int n,value;
front=0; rear=0;
do
{
do
{
```

```
printf("Enter the element to be inserted\n"); scanf("%d",&value);
insert(queue,&rear,front,value);
printf("Enter 1 to continue\n");
scanf("%d",&n);
} while(n == 1);

printf("Enter 1 to delete an element\n");
scanf("%d",&n);
while( n == 1)
{
delete(queue,&front,rear,&value);
printf("The value deleted is %d\n",value); printf("Enter 1 to delete an element\n");
scanf("%d",&n);
}
printf("Enter 1 to continue\n");
scanf("%d",&n);
} while(n == 1);
}
```

14.5.4 Applications of Queues

One application of the queue data structure is in the implementation of priority queues required to be maintained by the scheduler of an operating system. It is a queue in which each element has a priority value and the elements are required to be inserted in the queue in decreasing order of priority. This requires a change in the function that is used for insertion of an element into the queue. No change is required in the delete function.

Program 14.8: A complete C program implementing a priority queue is shown here

```
# include <stdio.h>
# include <stdlib.h>
struct node
{
int data;
int priority;
struct node *link;
};
void insert(struct node **front, struct node **rear, int value, int priority) {
struct node *temp,*temp1;
temp=(struct node *)malloc(sizeof(struct node));
/* creates new node using data valuepassed as parameter */
if(temp==NULL)
{
printf("No Memory available Error\n");
exit(0);
}
temp->data = value;
temp->priority = priority;
temp->link=NULL;
if(*rear == NULL) /* This is the first node */
{
*rear = temp;
```

```
*front = *rear;
}
else
{
if((*front)->priority < priority)
/* the element to be inserted has highest priority hence should be the first element*/
{
temp->link = *front;
*front = temp;
}
else
if( (*rear)->priority > priority)
/* the element to be inserted has lowest priority hence should be the last element*/
{
(*rear)->link = temp;
*rear = temp;

}

else
{
temp1 = *front;
while((temp1->link)->priority >= priority)
/* find the position and insert the new element */
```

```
temp1=temp1->link;
temp->link = temp1->link;
temp1->link = temp;
}
}
void delete(struct node **front, struct node **rear, int *value, int *priority) {
struct node *temp;
if((*front == *rear) && (*rear == NULL))
{
printf(" The queue is empty cannot delete Error\n");
exit(0);
}
*value = (*front)->data;
*priority = (*front)->priority; temp = *front;
*front = (*front)->link;
if(*rear == temp)
*rear = (*rear)->link;
free(temp);
}

void main()
{
struct node *front=NULL,*rear = NULL;
int n,value, priority;
```

```
do
{
do
{
printf("Enter the element to be inserted and its priority\n");
scanf("%d %d",&value,&priority);
insert(&front,&rear,value,priority);
printf("Enter 1 to continue\n");
scanf("%d",&n);
} while(n == 1);
printf("Enter 1 to delete an element\n");
scanf("%d",&n);
while( n == 1)
{
delete(&front,&rear,&value,&priority);

printf("The value deleted is %d\ and its priority is %d \n",value,priority); printf("Enter 1 to delete an element\n");
scanf("%d",&n);
}
printf("Enter 1 to delete an element\n");
scanf("%d",&n);
} while( n == 1)
}
```

Self Assessment Questions

6.Queue is also called a _____ data structure.

7.In a linked implementation of a queue, the newly added element is the front element. (True/False)

8.An application of a _____ queue is in the implementation of priority queues required to be maintained by the scheduler of an operating system.

14.6 Summary

The most important aspect of program development is finding a good representation of the data manipulated by that program. The stack and the queue are examples of ADTs commonly used in computer programming. A stack is basically a list with insertions and deletions permitted from only one end, called the stack-top, so it is a data structure that exhibits the LIFO property. One of the important applications of a stack is in the implementation of recursion in the programming language. A queue is also a list with insertions permitted from one end, called rear, and deletions permitted from the other end, called front. So it is a data structure that exhibits the FIFO property. When the size of the stack/queue is known beforehand, the array implementation can be used and is more efficient.

When the size of the stack/queue is not known beforehand, then the linked representation is used. It provides more flexibility. A circular queue is a queue in which the element next to the last element is the first element.

14.7 Terminal Questions

1.Convert the following infix expression into postfix:

a/b-c+d*e-a*c

2.Convert the following infix expression into prefix

a*(b+c)/d-g

3.Write an expression to point the front pointer in a circular queue to the next position?

4.State true or false:

Stack is used in recursive programs.

5.The runtime memory allocation is also called _____.

14.8 Answers to Self Assessment Questions

1. Bitmapped
2. Dynamic
3. LIFO
4.(a) Stack cannot grow 5.

True

6. FIFO
7. False
8. priority

14.9 Answers to Terminal Questions

1. ab/c-de*+ac*
2. -/*a+bcdg
3.*front = (*front + 1) % MAX; 4.

True

5.Dynamic Memory Allocation.

14.10 Exercises

1.Write a C program to implement a stack of characters.

2.Write a C program to implement a double-ended queue, which is a queue in which insertions and deletions may be performed at either

end. Use a linked representation.

3.Write a program to reverse a linked list.

4.Write a program to delete a node from a linked list.

5.Write a program to insert a node into after the given node in a linked list.

6.What is meant by postfix expression? Write a program to convert an infix expression into a postfix expression.

7.Write a program to evaluate a postfix expression.

8.Write a function to check whether a stack is full.

9.Write a function to check whether a queue is empty.

10.Why stack and queues are called Abstract Data Types? Explain.

References:

1.E. Balagurusamy, “Programming with ANSI C”, Tata McGraw-Hill Publishers, New Delhi.

2.Byron S. Gottfried, Schaum’s Outline Series, “Theory and Problems of Programming wiith C”, ata McGraw-Hill Publishers, New Delhi. 3.Stephen C. Kochan, “Programming in C”, CBS Publishers, Revised Edition, New Delhi.

4.Brian W. Kernighan and Dennis M. Ritchie, “The C Programming Language”, Second Edition, Prentice-Hall of India, New Delhi.
