B.C.A. – I YEAR

DJA1C : PROGRAMMING IN C

SYLLABUS

Unit I
C Declarations:- Introduction – Character Set – C tokens – Keywords and Identifiers – Identifiers – Constants – Variables – Data types – Declaration of Variables – Declaration of Storage Class – Assigning Values to Variables – Defining Symbolic Constants – Declaring Variable as Constant .
Operators and Expressions:- Introduction – Arithmetic Operators – Relational Operators – Logical Operators – Assignment Operators – Increment and Decrement Operators – Conditional Operator – Bitwise Operators – Special Operators – Arithmetic Expressions – Evaluation of Expressions – Precedence of Arithmetic Expressions.
Unit II.
Managing Input and Output Operations:- getchar( ) – putchar( ) – scanf( ) – printf( ).
Decision Making and Branching:- Introduction – Decision Making with IF Statement – Simple IF statement – The IF…Else Statement – Nesting of IF…Else Statements – The ELSE IF ladder – The Switch Statement – The ?: Operator – The GOTO statement.
Decision Making and Looping:- Introduction – The WHILE Statement – The DO Statement – The FOR statement – Jumps in Loops.
Unit III.
Arrays :- Introduction – One-dimensional arrays – Declaration of One-dimensional arrays – Initialization of One-dimensional arrays - Two-dimensional arrays – Initialization of Two-dimensional arrays – Multi-dimensional arrays.
Character Arrays and Strings:- Introduction – Declaring and Initializing String Variables – Reading Strings from Terminal – Writing Strings to Screen – String Handling Functions.
Unit IV.
User-Defined functions:- Introduction – Need for User-defined functions – Definition of functions – Return Values and their Types – Function Calls – Function Declaration – Category of functions – No Arguments and No return values – Arguments but No return Values – Arguments with return values – No arguments but a return a value – Recursion – Passing Arrays to functions – The Scope, Visibility and lifetime of a variables.
Structures and Unions:- Introduction – Defining a Structure – Declaring Structure Variables – Accessing Structure Members – Structure Initialization – Arrays of structures – Structures and functions – Unions.
Unit V
Pointers:- Introduction – Understanding pointers – Accessing the Address of a Variable – Declaring Pointer Variables – Accessing a variable through its pointer – Pointer Expressions – Pointers as function arguments.
File Management in C:- Introduction – Defining and Opening a file – Closing a File – Input/Output Operations on files – Error Handling During I/O Operating .
Text Book:
Programming in ANSI C – 6th Edition by E Balagurusamy – Tata McGraw Hill Publishing Company Limited.
Reference Book:
Computer Programming in C – V. Rajaraman-PHI

UNIT I : C DECLARATIONS

***C Declarations:-** Introduction – Character Set – C tokens – Keywords and Identifiers – Identifiers – Constants – Variables – Data types – Declaration of Variables – Declaration of Storage Class – Assigning Values to Variables – Defining Symbolic Constants – Declaring Variable as Constant .*

***Operators and Expressions:-** Introduction – Arithmetic Operators – Relational Operators – Logical Operators – Assignment Operators – Increment and Decrement Operators – Conditional Operator – Bitwise Operators – Special Operators – Arithmetic Expressions – Evaluation of Expressions – Precedence of Arithmetic Expressions.*

C-DECLARATIONS

INTRODUCTION

HISTORY OF C

C was evolved from ALGOL, BCPL and B by Dennis Ritchie at the Bell Laboratories in 1972. C is highly portable. ‘This means that C programs written for one computer can be run on another with little or no modification. Portability is important if we plane to use a new computer with a different operating system. C language is well structured in terms of function modules or blocks. This modular structure makes program debugging, testing and maintenance easier.

BASIC STRUCTURE OF C PROGRAMS

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. The documentation section consists of a set of comment lines giving the name of the program, the author and other details, which the programmer would like to use later. 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. This section also declares all the user-defined functions.

Every C program must have one Main () function. 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 the opening and the closing braces. The program execution begins at the opening brace and ends at the 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.

REVIEW QUESTIONS

1.1 State whether the following statements are true or false.

a) Every line in a C program should end with a semicolon.
b) In C language lowercase letters are significant.
c) **Main**() is where the program begins its execution .
d) The closing brace of the main() in a program is the logical end of the program.

1.2 Which of the following statements about comments are false?

a) Use of comments reduces the speed of execution of a program.
b) Comments serve as internal documentation for programmers.

1.3 Fill in the blanks with appropriate words in each of the following statements.

a) Every program statement in a C program must end with a ………………….
b) The ……………………… Function is used to display the output on the screen.
c) The ………………. Header file contains mathematical functions.

CHARACTER SET

The characters that can be used to form words, numbers and expressions form C set. The characters in C are grouped into the following categories:

1. Letters
2. Digits

3. Special characters
4. White spaces

C TOKENS

Individual words and punctuation marks are called tokens.

KEYWORDS AND INDENTIFIERS

All keywords have fixed meanings and these meanings cannot be changed. All keywords must be written in lowercase. Identifiers refer to the names of variables, functions and arrays. These are user-defined names and consist of a sequence of letters and digits, with a letter as first character.

Rules for Identifiers

1. First character must be an alphabet (or underscore).
2. Must consist of only letters, digits or underscore.
3. Only first 31 characters are significant.
4. Cannot use a keyword.
5. Must not contain white space.

CONSTANTS

Constants in C refer to fixed values that do not change during the execution of a program.

Integer Constants

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

Decimal integers consist of a set of digits, 0 through 9 preceded by an optional – or + sign. Valid examples of decimal integer constants are:

123 – 321 0 654321 + 78

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

037 0 0435 0551

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 letter A through F represents the numbers 10 through 15. Following are the examples of valid hex integers:

Real constants

Numbers containing fractional parts like 17.548 are called real (or floating point) constants are:

0.0083 – 0.75 435.36 + 247.0

A real number may also be expressed in exponential (or scientific) notation

This notation floating point form.

0.65e4 12e-2 1.5e+5 3.18E3 -1.2E-1

Single Character Constants

A single character constant (or simply character constant) contains a single character enclosed within a pair of single quote marks.

‘5’ ‘X’’ ‘;’ ‘ ’

String Constants

A string constant is sequence of characters enclosed in double quotes. The characters may be letters, numbers, special characters and blank space. Examples are:

Helo!” “1987” “WELL DONE” “?...!” “5+3” “x”

Backslash character Constants

C supports some special backslashes character constants that are used in output functions each one of them represents one character, although they consist of two characters. These characters combinations are known as escape sequences.

Table : Backslash character Constants

Constant	**Meaning**
‘\f’	Form feed
‘\n’	New line
‘\t’	Horizontal tab

VARIABLES

A variable is a data name that may be used to store a data value.

DATA TYPES

ANSI C supports three classes of data types:

1. Primary (or fundamental) data types
2. Derived data types
3. User-defined data types

Fig. Primary data types in C

Table Size and Range of Basic Data Types on 16- bit Machines

Data type	**Range of values**
Char	-128 to 127
Int	-32,768 to 32,767
Float	3.4e-38 to 3e+e38
double	1.7-308 to 1.7e+308

Integer Types

Integers are whole numbers with a range of values supported by a particular machine. Integers occupy one word of storage C has three classes of integer storage, namely short int, int, and long int, in both signed and unsigned forms. Short int represents fairly small integer values and requires half the amount of storage as a regular int number uses. Unlike signed integers, unsigned integers use all the bits for the magnitude of the number and are always positive. 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.

Floating Point Types

Floating point (or real) numbers are stored in 32 bits (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.

Void Types

The void type has no values. This is usually used to specify the type of functions. The type of a function is said to be void when it does not return any value to the calling function.

Character Types

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

DECLARATION OF VARIABLES

After designing suitable variable names, we must declare them to the compiler. Declaration does two things:

1. It tells the compiler what the variable name is.
2. It specifies what type of data the variable will hold.

Primary Type Declaration

A variable can be used to store a value of any data type.

The syntax

Data-type v1, v2,….vn ;

v1, v2,…..vn are the names of variables.

For example,

```
int count ;

int number, total ;

double ratio ;
```

User-Defined Type Declaration

C supports a feature known as “type definition” that allows users to define an identifier that would represent an existing data type. The user-defined data type identifier can later be used to declare variables. It takes the general form:

typedef type identifier:

Some examples

```
typedef int units ;

typedef float marks ;
```

Another “identifier” is a user-defined data type is enumerated data type provided by ANSI standard. It is defined as follows:

enum identifier {value1, value2,…valuen};

The “identifier” is a user-defined enumerated data type which can be used to declare variables that can have one of the values enclosed within the braces (known as enumeration constants). After this definition, we can declare variables to be of this ‘new’ type as below:

enum identifier v1, v2,…vn;

The enumerated variables v1,v2,…vn can only have one of the values value1, value2,…valuen.

An example:

```
enum day {Monday, Tuesday, Sunday} ;

enum day week_st, week_end ;

week_st = Monday ;

if (week_st == Tuesday)

week_end = Saturday ;
```

DECLARATION OF STORAGE CLASS

Variables in C can have not only data type but also storage class that provides information about their location and visibility. The storage class decides the portion of the program within which the variables are recognized.

C provides a variety of storage class specifies that can be used to declare explicitly the scope and lifetime of variables. There are four storage class specifies (auto, register, static, and extern) whose meanings are given in Table 2.10.

Static and external (extern) variables are automatically initialized to zero. Automatic (auto) variables contain undefined values (known as ‘garbage’) unless they are initialized explicitly.

Table : Storage classes and their Meaning

Storage class	**Meaning**
auto	Local variable known only to the function in which it is declared. Default is auto.
static	Local variable which exists and retains its value even after the control is transferred to the calling function.
extern	Global variable known to all functions in the file.
register	Local variable which is stored in the register.

ASSIGNING VALUES TO VARIABLES

Assignment Statement

Values can be assigned to variables using the assignment operator = as follows:

```
Variable_name = constant;

Balance        = 75.84 ;

Yes            = ‘x’ ;
```

Another way of giving values to variables is to input data through keyboard using the scanf function.

```
Scanf(“control string”, &variable1,&variable2,….);
```

The control sting contains the format of data being received. The ampersand symbol & before each variable name is an operator that specifies the variable name’s address.

DEFINING SYMBOLIC CONSTANTS

We often use certain unique constants in a program. These constants may appear repeatedly in a number of places in the program. One example of such a constant is 3.142, representing the value of the mathematical constant “pi”. Another example is the total number of students whose mark-sheets are analyzed by a ‘test analysis program’.

A constant is defined as follows:

```
#define symbolic-name value of constant
```

Valid examples

```
#define STRENGTH 100

# define PASS_MARK 50

#define p1 3.14159
```

Symbolic names are sometimes called constant identifiers. The following rules apply to a #define statement which define a symbolic constant:

1. Symbolic names have the same form as variable names. (Symbolic names are written in CAPITALS to visually distinguish them from the normal variable names, which are written in lowercase letters. This is only a convention, not a rule.
2. No blank space between the pound sign ‘#’ and the word define is permitted.
3. ‘#’ must be the first character in the line.
4. A blank space is required between # define and symbolic name and between the symbolic name and between the symbolic name and constant.
5. #define statements must not end with a semicolon.
6. After definition, the symbolic name should not be assigned any other value within the program by using an assignment statement. For example, STRENGTH = 200; is illegal.
7. Symbolic names are NOT declared for data types. Its data type depends on the type of constant.
8. #define statements may appear anywhere in the program but before it is referenced in the program (the usual practice is to place them in the beginning of the program).

2.10 DECLARING A VARIABLE AS CONSTANT

We may like value of certain variables to remain constant during the execution of a program.

Example:

Const int class_size = 40;

Const is a new data type qualifier defined by ANSI standard. This tells the compiler that the value of the int variable class_size must not be modified by the program.

Simple ‘c’ programs

1. Area of a square

```
void main ( )
{
    int a, s :
    print f (″ enter s value \n”);
    scanf (″ % d”,&s);
    print f (″ S = % d\ n″ , s);
    a = s * s;
    Print f (″ area of square = % d\n”, a);
}
```

2. Area of a triangle

   a = ½ bh

```
Void main ( )
{
    int b, h ;
    float a ;
    print f (″ enter b, h value \n”) ;
    scan f (″ %d% d” , &b, &h) ;
    print f (″ b = % d h = % d / n” , b, h) ;
    a = (0.5) * b * h ;
    printf (“area of a triangle = % f \ n” , a );
}
```

3. Volume of a Cone V = 1/3 $\pi r^2 h$

```
void main ( )
{
    int r, h ;
```

```
        float v ;
        printf ≠ (″ enter r, h value \n”);
        scanf ≠ (″ % d%d” , &r, &h) ;
    print f (″ r = % d \a = % d \n” , r , h) ;
        v = (0.33) * (3.14) * r * r * h ;
    print f (″ volume of a cone = % f\n”, v) ;
    }
```

4. String – sequence of characters

```
        eg : char name (20),regno (7)
    void main ( )
    {
        int s1, s2, s3, s4 ,total ;
        float average ;
        char name [20], reg. no [7] ;
        clrscrl);
        Scan f (″ %s%s% d%d%d% d” , name,
                      regno, & s1, &S2, &S3,&S4) ;
        total = S1 + S2 + S3 + S4
        average=total / 4 ;
        print f (″ name = % S \ n” , name ) ;
        print f (″ meg. no = % S \ n” , regno) ;
        print f (″ S1 = % d S2 = % d S3 = % d S4 = % d/n”, S1, S2, S3, S4)
        Print f (″ total = % d \ n” , total) ;
        Print f (″ average = % f \ n” , average) ;
    }
```

5.Display the address

```
    # include < stdio.h  >
≠ include < conio. h >
void main ( )
{
    char name [20], street [20], city [10]
                pin [7];
 clrscr ( );
```

```
    scanf (″ %s % s % s % s” , name, street, city, pin)
    printf  (″ name = % s \ n” , name) ;
    printf (″ street = % s \ n” , street) ;
    printf (″ city = % s \ n” , city) ;
   printf (″ Pincode= % s \ n”, pin) ;
    getch ( ) ;
}
```

Just Remember

- Do not use the underscore as the first character of identifiers (or variable names) because many of the identifiers in the system library start with underscore.
- Use only 31 or less characters for identifiers. This helps ensure portability of programs.
- Do not use keywords or any system library names for identifiers.
- Do not give any space between # and define.

Review Questions

1. State whether the following statements are true or false.
   a) Any valid printable ASCII character can be used in an identifier.
   b) All variables must be given a type when they are declared.
   c) Declarations can appear anywhere in a program.
   d) ANSI C treats the variables name and Name to be same.
   e) The underscore can be used anywhere in an identifier.
   f) The keyword void is a data type in C.
   g) All static variables are automatically initialized to zero.

Fill in the blanks with appropriate words.

a) The keyword ……………………….. Can be used to create a data type identifier.
b) A global variable is also known as …………………….. Variable.
c) A variable can be made constant by declaring it with the qualifier ………………. at the time of initialization.

1. Describe the four basic data types. How could we extend the range of values they represent?

2. Describe the purpose of the qualifiers const and volatile.

Programming Exercises

1. Write a program that prints the even numbers from 1 to 100.
2. Write a program the requests two float type numbers from the user and then divides the first number by the second and display the result along with the numbers.

OPTERATORS AND EXPRESSIONS

C SUPPORTS A RICH SET OF BUILT-IN OPEPRATORS

An operator is symbol used to manipulate

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

1. Arithmetic operators
2. Relational operators
3. Logical operators
4. Assignment operators
5. Increment and decrement operators
6. Conditional operators
7. Bitwise operators
8. Special operators

ARITHMETIC OPERATORS

C provided all the basic arithmetic operates.

Arithmetic Operators

Operator	**Meaning**
+	Addition or unary plus
-	Subtraction or unary minus

X	Multiplication
/	Division
%	Modulo division

Integer division truncates any fractional part. The modulo division operation produces the remainder of an integer division.

Integer Arithmetic

When both the operands in a single arithmetic expression such as a+b are integers, the expression is called an integer expression, and the operation is called integer arithmetic. Integer arithmetic always yields an integer value.

A and b are integers, a = 14 and b = 4

a – b = 10
a + b = 18
a X b = 56
a / b = 3 (decimal part truncated)
a % b = 2 (remainder of division)

Real Arithmetic

An arithmetic operation involving only real operands is called real arithmetic. A real operand any assume values either in decimal or exponential notation.

The operator % cannot be used with real operands.

Mixed-mode Arithmetic

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 the result is always a real number.

15/10.0 = 1.5

RELATIONAL OPERATORS

We often compare two quantities and depending on their relation, take certain decisions. An expression such as

A < b 1 < 20

An expression containing a relational operator is termed as a relational expression. The value of a relational expression is either True of false.

C supports six relational operators in all. These operators and their meanings are shown in Table

Table Relational Operators

Operator	**Meaning**
<	is less than
<=	is less than or equal to
>	is greater than
>=	is grater than or equal to
==	is equal to
!=	is not equal to

A simple relation expression contains only one relational operator and takes the following form:

ae-1 relational operator ae-2

LOGICAL OPERATORS

In addition to the relational operators, C has the following three logical operators.

&&	meaning logical	AND
||	meaning logical	OR
!	meaning logical	NOT

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

An expression of this kind, which combines two or more relational expressions, is termed as a logical expression or a compound relational expression.

Some examples of the usage of logical expressions are:

1. if (age > 55 && salary < 1000)

if (number < 0 || number > 100)

ASSIGMENT OPERATORS

Assignment operators are used to assign the result of an expression to a variable.

```
v op= exp;
```

Where V us a variable, exp is an expression and op is a C binary arithmetic operator. The operator op= is known as the shorthand assignment operator.

An example

```
x += y+1;
```

This is same as the statement

```
x = x + (y+1);
```

INCREMENT AND DECREMENT OPERATORS

C allows two very useful operators increment and decrement operators:

++ and ––

The operator ++ adds 1 to the operand, while – subtracts 1. Both are unary operators and takes the following form:

```
++m; or m++;
−−m; or m−−;
++m; is equivalent to m = m+1; (or m + = 1;)
−−m; is equivalent to m = m−1; (or m −= 1;)
```

CONDITIONAL OPERATOR

A ternary operator pair “? :” is available in C to construct conditional expressions of the form

```
ecp1 ? exp2 : exp3
```

Where exp1, exp2, and exo3 are expressions.

The operator? : works as follows: exp 1 is evaluated first. if it is nonzero (true), then the expression exp2 is evaluated and becomes the value of the expression. If exp 1 is false, exp3 is evaluated and its value becomes the value of the expression.

For example

```
a = 10;

b = 15;

x = (a > b) ? a : b;
```

BITWISE OPERATORS

C has special operators known as bitwise operators for manipulation of data at bit level. These operators are used for testing the its, r shifting them right or left. Bitwise operators may not be applied to float or double.

Table : Bitwise Operators

Operator	**Meaning**
&	bitwise AND
|	bitwise OR
^	bitwise exclusive OR

<<	shift left
>>	shift right

SPECIAL OPERATORS

C supports some special operators of interest such as comma operator, sixe of operator, pointer operators (& and *) and member selection operators (. and –>).

The comma Operator

The comma operator can be used to link the related expressions together.

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.

ARITHMETIC EXPRESSIONS

An arithmetic expression is a combination of variables, constants, and operators arranged as per the syntax of the language.

EVALUATION OF EXPRESSISSONS

Expressions are evaluated using an assignment statement of the form:

variable = expression;

Variable is any valid C variable name. When the statement is encountered, the expression is evaluated first and the result then replaces the previous value of the variable on the left-hand side.

Examples:

x = a * b – c;

z = a – b / c + d;

PRECEDENCE OF ARITHMETIC OPERATORS

An arithmetic expression without parentheses will be evaluated from left to right using the rules of precedence of operators. There are two distinct priority levels of arithmetic operators in C:

High priority * / %

Low priority + −

The basic evaluation procedure includes 'two' left-to-right passes through the expression. During the first pass, the high priority operators (if any) are applied as they are encountered. During the second pass, the low priority operators (if any) are applied as they are encountered.

Example:

x = a-b/3 + c*2−1

When a = 9, b = 12, and c = 3, the statement becomes

x = 9−12/3 + 3*2−1 Answer is 10

However, the order of evaluation can be changed by introducing parentheses into an expression. Consider the same expression with parentheses as shown below:

9−12/(3+3)*(2−1)

Whenever, parentheses are used, the expressions within parentheses assume highest priority, if two or more sets of parentheses appear one after another as shown above, the expression contained in the left-most set is evaluated first and the right-most in the last.

Answer is 7

Rules of Precedence and Associability

- Precedence rules decides the order in which different operators are applied
- Associability rule decides the order in which multiple occurrences of the same level operator are applied.

Just Remember

- Add parentheses wherever your feel they would help to make the evaluation order clear.
- Do not forget a semicolon at the end of an expression.
- Do not use increment or decrement operators with any expression other than variable identifier.
- Integer division always truncates the decimal part of the result. Use it carefully. Use casting where necessary.
- All mathematical functions implements double type parameters and return double type values.

Review Questions

(a) all arithmetic operators have the same level of precedence.

(b) The modulus operator % can be used only with integers.

(c) A unary expression consists of only one operand with no operators.

(d) Associability is used to decide which of several different expressions is evaluated first.

(e) An expression statement is terminated with a period.

(f) An explicit cast can be used to change the expression.

(g) Parentheses can be used to change the order of evaluation expressions.

Fill in the blanks with appropriate words.

(a) The expression containing all the integer operands is called ……………expression.

(b) The operator …………………. cannot be used with real operands.

(c) C supports as many as ………………. relational operators.

(d) The order of evaluation can be changed by using ……………………. in an expression.

Declared a as int and b as float, state whether the following statements are true of false.

(a) The statement a = 1/3 + 1/3 + 1/3; assigns the value 1 to a.

(b) The statement b = 1.0/3.0 + 1.0/3.0 + 1.0/3.0; assigns a value 1.0 to b.

(c) The statement b = 1.0/3.0 * 3.0 gives a value 1.0 to b.

Which of the following expressions are true?

(a) !(5 +5 >=10)

3.12 Find the output of the following program?

```
main ()
```

```
            {
                        int X = 100;
                        print f ("%d/n", 10 + X++) ;
                        print f ("%d/n", 10 + ++X);
            }
```

3.18 What is the error, if any, in the following segment?

```
int X = 10 ;
float y = 4.25 ;
x = y%x ;
```

3.5 Given an integer number, write a program that displays the number as follows:

First	:	all digits
Second line	:	all except first digit
Third line	:	all except first two digits

…….

Last line

For example, the number 5678 will be displayed as:

```
5 6 7 8
6 7 8
7 8
8
```

3.15 Write a program to read three values using scanf statement and print the following results:

(a) Sum of the values

(b) Average of the three values

(c) Largest of the three

(d) Smallest of the three

UNIT II : MANAGING INPUT AND OUTPUT OPERATIONS

Managing Input and Output Operations:- *getchar( ) – putchar( ) – scanf( ) – printf( ).*

Decision Making and Branching:- *Introduction – Decision Making with IF Statement – Simple IF statement – The IF…Else Statement – Nesting of IF…Else Statements – The ELSE IF ladder – The Switch Statement – The ?: Operator – The GOTO statement.*

Decision Making and Looping:- *Introduction – The WHILE Statement – The DO Statement – The FOR statement – Jumps in Loops.*

MANAGING INPUT AND OUTPUT OPERATIONS

WRITING A CHARACTER

Like getchar, there is an analogous function putchar for writing characters one at a time to the terminal

Putchar (variable_name);

Where variable_name is a type char variable containing a character. This statement displays the character contained in the variable_name at the terminal. For example, the statements

Answer = ‘y’;

Putchar (answer);

Will display the character Y on the screen. The statement

Putchar (‘/n’);

Would cause the cursor on the screen to move to the beginning of the next line.

FORMATTED INPUT

Formatted input refers to an input data that has been arranged in a particular format. The general form of scanf is

Scanf (“control string”, arg1, arg2,…..argn);

The control string specifies the field format in which the data is to be entered and the arguments arg1, arg2,…., argn specify the address of locations where the data is stored. Control string and arguments are separated by commas.

Control string (also known as format string) contains field specifications, which direct the interpretation of input data. It may include:

- Field (or format) specifications, consisting of the conversion character %, a data type character (or type specifier), and an optional number, specifying the field width.
- Blanks, tabs, or newlines.

Blanks, tabs and newlines are ignored. The data type character indicates the type of data that is to be assigned to the variable associated with the corresponding argument. The field width specifier is optional.

Some versions of scanf support the following conversion specifications for strings:

%[characters]

%[^characters]

The specification %[characters] means that only the characters specified within the brackets are permissible in the input string. If the input string contains any other character, the string will be terminated at the first encounter of such a character. The specification %[^characters] does exactly the reverse. That is, the characters specified after the circumflex(^) are not permitted in the input string. The reading of the string will be terminated at the encounter of one of these characters.

Table : commonly used scanf Format Codes

Code	Meaning
%c	Read a single character
%d	Read a decimal integer
%e	Read a floating point value

%f	Read a floating point value
%u	Read an unsigned decimal integer

The following letters may be used as prefix for contain conversion characters.

h	for short integers
% ld for long integers	
I	for long integers or double
L	for long double

FORMATTED OUTPUT

The printf statement provides certain features that can be effectively exploited to control the alignment and spacing of print-outs on the terminals. The general form of printf statement is:

printf("control string", arg1, aft2, ….., argn);

Control string consists of three types of items:

1. Characters that will be printed on the screen as they appear.
2. Format specifications that define the output format for display of each item.
3. Escape sequence characters such as \n, \t, and \b.

The control string indicates how many arguments follow and what their types are. The arguments arg1, arg2, ….., argn are the variables whose values are formatted and printed according to the specifications of the control string. The arguments should match in number, order and type with the format specifications.

Just Remember

- While using getchar function, care should be exercised to clear any unwanted characters in the input stream.

- Do not forget to include <stdio.h header files when using functions from standard input/output library.
- Do not forget to include <ctype.h Header file when using functions from character handling library.
- Do not forget to close the format string in the scanf or printf statement with double quotes.
- Do not forget the comma after the format string in scanf and printf statements.

Review Questions

State whether the following statements are true of false.

(a) The purpose of the header file <studio.h> is to store the programs created by the users.

(b) A the input list in a scannf statement can contain one or more variables.

(c) When an input stream contains more data items than the number of specifications in a scanf statement, the unused items will be used by the next scanf call in the program.

(d) The print list in a printf statement can contain function calls.

Fill in the blanks in the following statements.

(a) The …………………. Specification is used to read or write a short integer.
(b) By default, the real numbers are printed with a precision of …………….. decimal places.

What is the purpose of scanf () function?

Wrote a program to read three integers from the keyboard using one scanf statement and output them on one line using:

(a) three printf statements,

(b) only one printf with conversion specifiers, and

(c) only one printf without conversion specifiers.

Write a program to read the name ANIL KUMAR GUPTA in three parts using the scanf statement and to display the same in the following format using the printf statement..

(a) ANIL K. GUPTA

(b) A.K. GUPTA

(c) GUPTA A.K.

DECISION MAKING AND BRANCHING

C language possesses such decision-making capabilities by supporting the following statements:

1. If statement
2. Switch statement
3. Conditional operator statement
4. Goto statement

DECISION MAKING WITH IF STATEMENT

The if statement is a powerful decision-making statement and is used to control the flow of execution of statements. It is basically a two-way decision statement and is used in conjunction with an expression. It takes the following form

if (test expression)

The if statement may be implemented in different forms depending on the complexity of conditions to be tested. The different forms is

1.Simple if statement

2. if…..else statement

3. Nested if….else statement

4. else if ladder.

SIMPLE IF STATEMENT

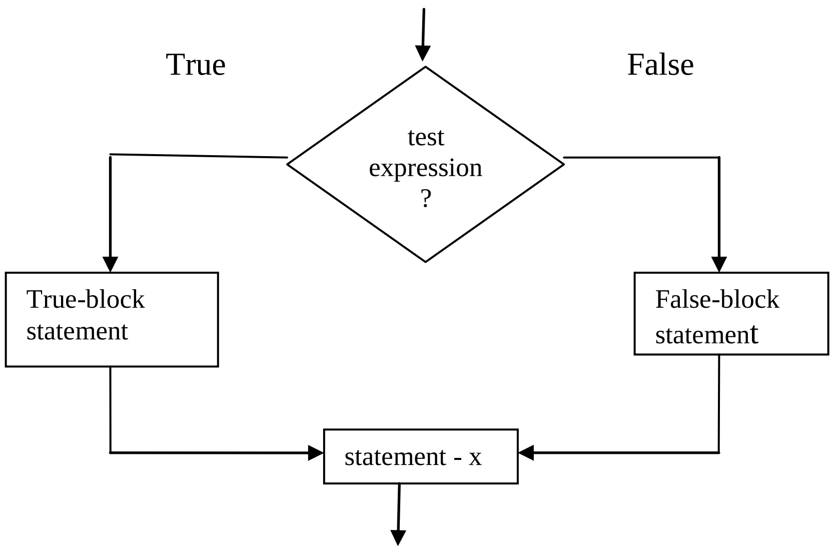
The general form of a simple if statement is

```
if (test expression)
{
        Statement-block;
}
Statement-x;
```

The ‘statement-block’ may be a single statement or a group of statements. If the test expression is true, the statement-block will be executed; otherwise the statement-block will be skipped and the execution will jump to the statement-x. Remember, when the condition is true both the statement-block and the statement-x are executed in sequence. This is illustrated in Fig.

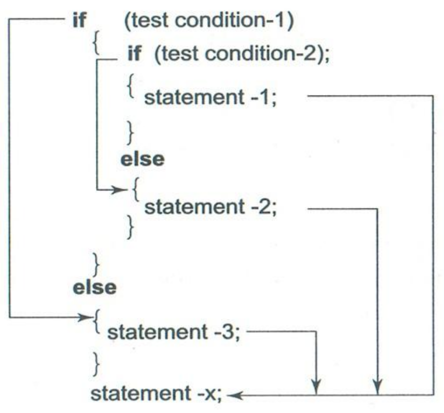
THE IF …. ELSE STATEMENT

The if….. else statement is an extension of the simple if statement. The general form is if the test expression is true, then the true-block statement(s), immediately following the if statements are executed; otherwise, the false-block statement(s), immediately following the if statements are executed; otherwise, the false-block statement(s) are executed. In either case, either true-block of false-block will be executed, not both.



NESTING OF IF….ELSE STATEMENTS

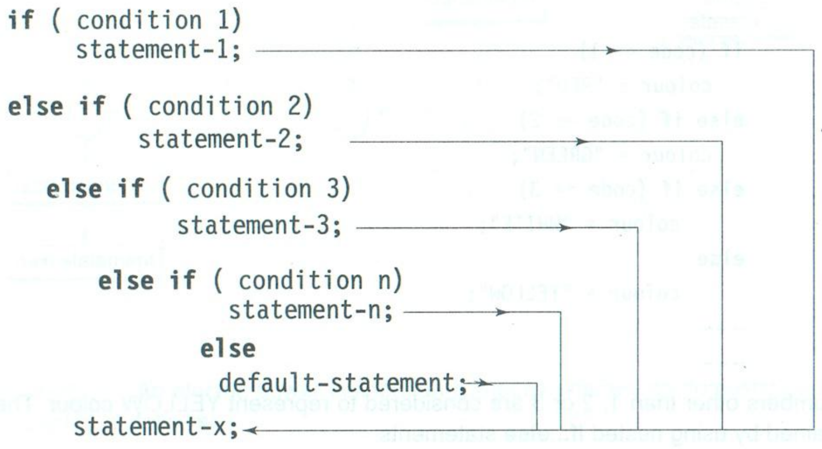
When a series of decisions are involved, we may have to use more than one if…else statement in nested form if the condition-1 is false, the statement-s will be executed; otherwise it continues to perform the second test. If the condition-2 is true,



```
if      (test condition-1)
{
   if  (test condition-2);
   {
     statement -1;
   }
   else
   {
     statement -2;
   }
}
else
{ statement -3;
}
statement -x;
```

THE ELSE IF LADDER

A multipath decision is a chain of ifs in which the statement associated with each else is an if.



```
if ( condition 1)
    statement-1;
else if ( condition 2)
         statement-2;
   else if ( condition 3)
            statement-3;
       else if ( condition n)
                statement-n;
              else
                default-statement;
    statement-x;
```

This construct is known as the else if ladder. The conditions are evaluated from the top (of the ladder), downwards. As soon as a true condition is found, the statement associated with it is executed and the control is transferred to the statement-x (skipping the rest of the ladder). When all the n conditions become false, then the final else containing the default-statement will be executed. Fig. 5.9 shows the logic of execution of else if ladder statements.

THE SWITCH STATEMENT

C has a built-in multiway decision statement known as a switch. The switch statement tests the value of a given variable (or expression) against a list of case values and when a match is found, a block of statements associated with that case is executed. The general form of the switch statement is as shown below:

```
Switch (expression)
{
        Case value-1 :
                Block-1
                Break;
        Case value-2 :
                block-2
                break;
…..
…..
Default:
                Default-block
                break;
}
Statement-x
```

The expression is an integer expression or characters. Value-1 value-2…. Are constants or constant expressions (evaluable to an integral constant) and are known as case labels. Each of these values should be unique within a switch statement. Block-1. Block-2…. Are statement lists and may contain zero or more statements. There is no need to put braces around these blocks. Note that case labels end with a colon (:)

When the switch is executed, the value of the expression is successfully compared against the values value-1, value-2,… if a case is found whose value matches with the value of the expression, then the block of statement, transferring the control to the statement-x following the switch.

The default is an optional case. When present, it will be executed if the value of the expression does not match with any of the case values. If not present, no action takes place if all matches fail and the control goes to the statement-x.

THE ? : OPERATOR

The C language has an operator, useful for making two-way decisions. This operator is a combination of? and :, and takes three operands. This operator is popularly known as the conditional operator. The general form of use of the conditional operator is as follows:

conditional expression? expression1: expression2

The conditional expression is evaluated first. If the result is non-zero, expression1 is evaluated and is returned as the value of the conditional expression. Otherwise, expression 2 is evaluated and its value is returned.

flag = (x < 0) ? 0 : 1;

THE GOTO STATEMENT

C supports the goto statement to branch unconditionally from one point to another in the program there may be occasions when the use of go to 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:

Just Remember

- Be aware of dangling else statements.
- Check the use of = operator in place of the equal operator = =.

- Do not give any spaces between the two symbols of relational operators = =, !=, >= and <=.
- Do not use the same constant in two case labels in a switch statement.

Programming Exercises

Write a program to determine whether a given number is 'odd' or 'even' and less than 200 that are divisible by 7

Write a program to find the number of and sum of all integers greater than 100 and less than 200 that are divisible by 7

DECISION MAKING AND LOOPING

INTRODUCTION

LOOP

Loop is a set of statements which is executed repeatedly.

Three types of Loop in 'C'

i) For loop (finite no. of times)
ii) While Loop (entry controlled Loop)
iii) do while (exit controlled loop)

Syntax

i) For

```
For ( initial      ; terminating ; increment /      )
      condition      condition       decrement

 {
     body of loop
 }
```

ii) While

```
while (condition)
{
```

```
                body of loop
         }
iii)  do while
         do
         {
                body of loop
         } while (condition) ;
```

Examples

Area of Square :

For

```
      # include < stdio.h >
      # include < conio.h >
Void main ( )
{
      int a, s, i, n ;
      clr scr ( ) ;
      Print f (″ enter no. of times \ n” ) ;
      scan f (″ % d” , & n) ;
      for ( i = o ; i < n ;i = i + 1)
{
      print f (″ enter s value \n” ) ;
      scanf (″ % d”, &s) ;
      a = s * s ;
      Print f (″ area = % d” , a ) ;
}
}
```

while

```
      void main ( )
{
      int a, s, n, i;
```

```
        Print f (″ enter no of times \ n″ ) ;
        scan f (″ % d”, & n) ;
        i = o; / * initial condtion * /
        while i < n / terminating condition * /
{
        scan  (″ % d”, &s)
        a = s * s;
        print f (″ area = % d” , s );
        i = i + 1; /* increment condition* /
        }
}
```

do … while

```
        void main ( )
{
        int a, s, i, n;
        printf (″ enter no of times \ n”)
        scanf (″ % d” , & n) ;
        i = o;
        do
        {
            scanf (″ % d” , & s ) ;
            a = s * s;
            printf (″ area = % d”,a) ;
            i = i + l;
            } while (i < n) ;
}
```

Difference between while and do… while

- While loop is executed only if the condition is true. If the condition is false, the loop will not be executed at all.
- Do while loop is executed then the condition is checked. If the condition is true, the loop will be executed again. If the condition is false, the loop will not be executed.

(i.e)

The do …. while loop will be executed minimum of 1 time.

Sum of digits

```
Void main ( )
{
        long int n;
        int n1, s = 0 ;
        clr scr ( ) ;
        print f (″ enter digit \n” ) ;
        scanf (″ % ld”,& n);
        print f (″ % ld” , n) ;
        while (n ! = 0)
        {
            n1 = n% 10 ;
            s = s + n1 ;
            n = n/10 ;
        }
        Print f (″ sum of digits = % d \ n”,s);
}
```

NOTE :

The entry-controlled and exit-controlleld loops are also known as pre-test and post-test loops respectively.

Sentionel Loops

Based on the nature of control variable and the kind of value assigned to it for testing the control expression, the loops may be classified into two general categories:

1. Counter-controlled loops
2. Sentinel-controlled loops

When we know in advance exactly how many times the loop will be executed, we use a counter controlled loop. We use a control variable known as counter. The counter must be initialized, tested and updated properly for the desired loop operations. The number of times we want to execute the loop may be a constant or a variable that is assigned a value. A counter-controlled loop is sometimes called definite repetition loop.

In a sentinel-controlled loop, a special value called a sentinel value is used to change the loop control expression from true to false. For example, when reading data we may indicate the "end of data" by a special value, like-1 and 999. The control variable is called sentinel variable. A sentinel controlled loop is often called indefinite repetition loop because the number of repetitions is not known before the loop begins executing.

Jumping out of a Loop

An early exit from a loop can be accomplished by using the break statement or the goto 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 will exit only a single loop.

During the loop operations, it may be necessary to skip a part of the body of the loop under certain conditions. Like the break statement, C supports another similar statement. However, unlike the break which causes the loop to be terminated, the continue, as the name implies, causes the loop to be continued with the next iteration after skipping any statements in between. The continue statement tells the compiler, "SKIP THE FOLLOWING STATEMENTS AND CONTINUE WITH THE NEXT ITERATION". The format of the continue statement is simply.

```
continue;
```

Jumping out of the program

We can jump out of a loop using either the break statement or goto statement. In a similar way, we can jump out of a program by using the library function exit().

The exit()

Function takes an integer value as its argument, Normally zero is used to indicate normal termination and a nonzero value to indicate termination due to some error or abnormal condition. The use of exit()

Function requires the inclusion of the header file <stdlib.h>

REVERSING THE DIGITS

```
Void main ( )
{
        long int n, s = o ;
        int n1 ;
        print f (″ enter the no\n”) ;
        scan f (″ % ld” , & n) ;
        print f (″ % ld” , n) ;
        while (n ! = 0)
        {
            n1 = n % 10;
            S = S * 10 + n1;
            n = n / 10 ;
        }
        print f (“reversed no. = % ld” , s) ;
        }
```

Nested Loop

A loop within another loop is known as nested loop.

Syntax

```
        For     (  initial;      terminating ; increment /  )
                   condition     condition     decrement
        {
            For ( i c ; t c ; in / de)
            {
                  body of loop
            }
```

```
}
```

eg. :

```
void main ( )
{
     int  i , j ;
For ( i = 0 ;  i < 3 ; i + +)
     {
            for ( j = 0 ; i < 2 ; i ++)
            {
            print f (″ Hello \n”) ;
            }
     }
}
```

output: Hello printed 6 times.

Just Remember

- Do not forget to place the semicolon at the end of do … while statement.
- Using commas rather than semicolon in the header of a for statement is an error.
- Do not forget to place the increment statement in the body of a while or do … while loop.
- Avoid using while and for statements for implementing exit-controlled (post-test) loops. Use do…while statement. Similarly, do not use do…while for pre-test loops.
- Use the function exit() only when breaking out of a program is necessary.

Review Questions

State whether the following statements are true or false.

(a) The do…while statement first executes the loop body and then evaluate the loop control expression.

(b) The number of times a control variable is updated always equals the number of loop iterations.

(c) While loops can be used to replace for loops without any change in the body of the loop.

(d) An exit-controlled loop is executed a minimum of one time.

(e) The use of continue statement is considered as unstructured programming.

Fill in the blanks in the following statements.

(a) In an exit-controlled loop, if the body is executed n times, the test condition is evaluated ……….. times.

(b) A for loop with the no test condition is known as ……..loop.

(c) The sentinel-controlled loop is also known as ………. Loop.

(d) In a counter-controlled loop, variable known as …… is used to count the loop operations.

Use of goto should be avoided. Explain a typical example where we find the application of goto becomes necessary.

Programming Exercises

Given a number, write a program using while loop to reverse the digits of the number. F or example, the number

12345

Should be written as

54321

(Hint: Use modulus operator to extract the last digit and the integer division by 10 to get the n-1 digit number from the n digit number.)

The factorial of an integer m is the product of consecutive integers from 1 to m. That is, factorial m = m! = m x (m-1) x ….. x 1.

Write a program to compute the sum of the digits of a given integer number.

UNIT III : ARRAYS

Arrays :- *Introduction – One-dimensional arrays – Declaration of One-dimensional arrays – Initialization of One-dimensional arrays - Two-dimensional arrays – Initialization of Two-dimensional arrays – Multi-dimensional arrays.*

Character Arrays and Strings:- *Introduction – Declaring and Initializing String Variables – Reading Strings from Terminal – Writing Strings to Screen – String Handling Functions.*

ARRAYS

Arrays

An array is a set of homogeneous (same data type) elements.

declaration

data type name of array (size) ;

eg.

float x [10] ; $\rightarrow$ 1 d float array

int a [5] ; $\rightarrow$ 1-d integer array

int s [5] [5] ; $\rightarrow$ 2-d integer array (matrix)

char name [24] [20] ;
char address [20] ; $\}$ character array

Types of Arrays

1. 1-d array :

one dimensional array (or) single subscripted variable.

2. 2-d array :

Two dimensional array (or) double subscripted variable.

eg : 1 – d array :

sum of ‘n’ nos

```
void main ( )
{
        int a [10] ,i , n, sum = o;
        print f (″ enter no. of nos to sum \ n”) ;
        scan f (″ % d”, & n) ;
        for ( i = 0 ; i < n ; i + +)
        scanf (″ % d” , &a[i] ) ;
        for (i = o ; i<n; i ++)
        sum = sum + a [i];
  for (i  = o ;i  < n ; i ++)
print f (″ % d \ n”, a[i]) ;
                print f (″ sum of % d no = % d \n” , n, sum) ;
}
```

2-d array (also known as matrix)

Addition of 2 matrices

```
Void main ( )
{
        int a [5] [5],  b [5] [5],  c [5] [5],  i, n, p, q ;
        print f (″ enter the order of matrices to be added \n”) ;
        scan f (″ % d % d” , & p, & q) ;
        for ( i = o ; i < p ; i + +)
        for (j = o ; j < q ; j ++)
        scan f (″ % d”, & a [i] [j]) ;
for ( i = o ; i < p ; i + +)
        for (j = o ; j < q ; j ++)
        scan f (″ % d”, & b [i] [j]) ;
        for (i = o i < p; i ++)
        for (j=o ; j < q; j ++)
        c [i] [j] = a [i] [j] + b [i] [j] ;
        print f (″ A matrix \n”) ;
        for (i = o ; i < p ; i ++)
        {
            for (j = o ; j < q ; j ++)
            print f (″ % d”,  a [i] [j] ) ;
            print f (″ \ n”) ;
        }
        Print f (″ B matrix \n”) ;
        for (i = o ; i < p ; i ++)
        {
            for (j = o; j < q ; j ++)
```

```
      print f (″ % d”, b [i] [j]) ;

      print f (″ \n”) ;

      }

      print f (″ Sum of matrices \n”) ;

  for (i = o ; i < p ; i ++)

  {

      for (j = o ; j < q ; j ++)

      print f (″ % d”,  c[i] [j] ) ;

      print f (″ \ n”) ;

  }

  }
```

INITIALIZATION OF ONE-DIMENSIONAL ARRAYS

Array can be initialized at either of the following stages:

- At compile time
- At run time

Compile Time Initialization

We can initialize the elements of arrays in the same way as the ordinary variables when they are declared. The general form of initialization of arrays is:

type array-name[size={list of values};

The values in the list are separated by commas. For example, the statement

int number [3] = { 0,0,0 };

Will declare the variable number as an array of size 3 and will assign zero to each element.

Float total[5] = {0,0,15,75,-10}:

An array can be explicitly initialized at run time. This approach is usually applied for initializing large arrays.

```
for (I = 0; I < 100; I = i+1)
{
        if      I < 50
                sum [i] = 0.0;          /* assignment statement */
        else
                sum[i] = 1.0;
}
```

MULTI-DIMENSIONAL ARRAYS

C allows arrays of three or more dimensions. The exact limit is determined by the compiler. The general form of a multi-dimensional array is

```
type array_name[s1] [s2] [s3]….[sm];
```

Where s is the size of the ith dimension. Some example are:

```
int survey [3] [5] [4] []12;

float table [5] [4] [3];
```

survey is a three-dimensional array declared to contain 180 integer type elements. Similarly table is a four-dimensional array containing 300 elements of floating-point type.

The array survey may represent a survey data of rainfall during the last three years from January to December in five cities.

Just Remember

- We need to specify three things, namely, name, type and size, when we declare an array.
- Always remember that subscripts begin at 0 (not 1) and end at size -1.
- Be aware of the difference between the “kth element” and the “element k”. The kth element has a subscript k-1, whereas the element k has a subscript of k itself.
- Referring a two-dimensional array element like x[I,j] instead of x[i][j] is a compile time error.
- During initialization of multi-dimensional arrays, it is an error to omit the array size for any dimension other than the first.

Review Questions

State whether the following statements are true or false.

(a) An array can store infinite data of similar type.

(b) When an array is declared, C automatically initializes its elements to zero.

(c) An expression that evaluates to an integral value may be used as a subscript.

(d) Accessing an array outside its range is a compile time error.

(e) A char type variable cannot be used as a subscript in an array.

7.2 Rill in the blanks in the following statements.

(a) The variable used as a subscript in an array is popularly known as …………. Variable.

(b) An array can be initialized either at compile time or at ……………

(c) An array created using malloc function at run time is referred to as ……….array.

What is the error in the following program?

```
main ( )

{
```

```
                int x ;

                float y [ ] ;

                ……

        }
```

What happens when an array with a specified size is assigned

(a) With values fewer than the specified size; and

(b) With values more than the specified size.

CHARACTER ARRAYS AND STRINGS

INTRODUCTION

A string is a sequence of characters that is treated as a single data item. Any group of characters (excepted double quote sign) defined between double quotation marks is a string constant. Example:

"Man is obviously made to think."

Character strings are often used to build meaningful and readable programs. The common operations performed on character strings include:

- Reading and writing strings.
- Combining strings together.
- Copying one string to another.
- Comparing strings for equality.
- Extracting a portion of a string.

DECLARING AND INITIALIZING STRING VARIABLES

C does not support strings as a data type. However, it allows us to represent strings as character arrays.

```
char string_name[size];
```

The size determines the number of characters in the string_name. Some examples are:

```
char city[10];

char name[30;]
```

Character arrays may be initialized when they are declared.

```
char city [9] = “ NEW YORK ”;

char city [9]={‘N’, ‘E’, ‘W’, ‘ ‘, ‘Y’, ‘0’, ‘R’, ‘K’, ‘/0’};
```

The familiar input function scanf can be used %s format specification to read in a string of characters. Example:

```
char address[10]

scanf(“%s”, address);
```

The problem with the scanf function is that it terminates its input on the first white space it finds. A white space includes blanks, tabs, carriage returns, form feeds, and new lines.

The scanf function automatically terminates the string that is read with a null character and therefore the character array should be large enough to hold the input string plus the null character.

gets()

Another and more convenient method of reading a string of text containing whitespaces is to use the library function gets available in the <studio.h header file. This is a simple function with one string parameter and called as under;

```
gets(str);
```

str is variable declared properly. It reads characters into str from the keyboard until a new-line character is encountered and then appends a null character to the string.

```
char line [80];

gets (line);
```

```
printf (“%s”, line);
```

WRITING STRINGS TO SCREEN

The printf function with % format is used to print strings to the screen.

For example:

```
Printf(“%s”, name);
```

Can be used to display the entire contents of the array name.

Using putcher and puts Function

We can use this putcher() function repeatedly to output a string of characters stored in an array using a loop. Example.

```
char name[6] = “PARIS”
for (i=0, i<5; i++)
putchar (name) [1];
putchar (‘\n’);
```

more convenient way of printing string values is to use the function puts declared in the header file (stdio.h). This is a one parameter function and invoked as under:

puts (str);

Where str is a variable containing a string value. This prints the value of the string variable str and then moves the cursor to the beginning of the next line on the screen. For example, the program segment.

```
char line [80];
gets (line);
puts (line);
```

Reads a line of text from the keyboard and displays it on the screen.

STRING-HANDLING FUNCTIONS

The C library supports a large number of string-handling functions that can be used to carry out many of the string manipulations the most commonly used string handling functions.

Function	**Action**
strcat()	concatenates two strings
strcmp()	compares two strings
strcpy()	copies one string over another
strlen()	finds the length of a string

The strcat function joins two strings together. It takes the following form

strcat(string1, string2);

String1 and string2 are character arrays. When the functions strcat is executed, string2 is appended to string1. It does so removing the null character at the end of string1 and placing string2 from there. The string at string2 remains unchanged.

The strcmp function compares two strings identified by the arguments and has a value 0 if they are equal. If they are not, it has the numeric difference between the first nonmatching characters in the strings. It takes the form:

Strcmp(string1, string2);

The strcpy function works almost like a string-assignment operator. It takes the form:

strcpy(string1, string2);

And assigns the contents of string2 to sting1. String2 may be a character array variable of a string constant.

This function counts and returns the number of characters in a string. It takes the form

n = strlen(string);

Where n is an integer variable, which receives the value of the length of the string. The argument may be a string constant. The counting ends at the first null character.

Just Remember

- Character constants are enclosed in single quotes and string constants are enclosed in double quotes.
- Using a string variable name on the left of the assignment operator is illegal.
- Do not forget to append the null character to the target string when the number of characters copied is less than or equal to the source string.
- Be aware the return values when using the functions strcmp and strncmp for comparing strings.
- The header file <string.h> is required when using string manipulation functions.

Review Questions

a) When initializing a string variable during its declaration, we must include the null character as part of the string constant, like “Good/0”.
b) The gets function automatically appends the null character at the end of the string read from the keyboard.
c) String variables cannot be used with the assignment operator.
d) The ASCII character set consists of 128 district characters.
e) The input function gets has one string parameter.

Fill in the blanks in the following statements.

a) We can use the conversion specification……………….. in scanf to read a line of text.
b) The function call strcat (s2,s1); appends …………. To …………

Write a program Program to do the following:

a) To output the question “Who is the inventor of C?”
b) To accept an answer.
c) To print out “Good” and then stop, if the answer is correct.
d) To output the message ‘try again’, if the answer is wrong.
e) To display the correct answer when the answer is wrong even at the third attempt and stop.

Write a program which will read a text and count all occurrences of a particular word.

UNIT IV : USER – DEFINED FUNCTIONS

User-Defined functions:- *Introduction – Need for User-defined functions – Definition of functions – Return Values and their Types – Function Calls – Function Declaration – Category of functions – No Arguments and No return values – Arguments but No return Values – Arguments with return values – No arguments but a return a value – Recursion – Passing Arrays to functions – The Scope, Visibility and lifetime of a variables.*

Structures and Unions:- *Introduction – Defining a Structure – Declaring Structure Variables – Accessing Structure Members – Structure Initialization – Arrays of structures – Structures and functions – Unions.*

USER – DEFINED FUNCTIONS

INDRODUCTION

C functions can be classified into two categories, namely, library functions and user-defined functions. Main is an example of user-defined functions. Printf and scanf belong to the category of library functions.

NEED FOR USER-DEFINED FUNCTIONS

main is a specially recognized function in C. Every program must have a main function to indicate where the program has to begin its execution. If a program is divided into functional parts, then each part may be independently coded and later combined into a single unit. I C, such subprograms are referred to as ‘functions’.

In order to make use of a user-defined function, we need to establish three elements that are related to functions.

1. Function definition.
2. Function call.
3. Function declaration.

The function definition is an independent program module that is specially written to implement the requirements of the function. In order to use this function we need to invoke it at a required place in the program. This is known as the function call. The program (or a function) that calls the function is referred to as the calling program or calling function. The calling program should declare any function (like declaration of a variable) that is to be used later in the program. This is known as the function declaration or function prototype.

DEFINITION OF FUNCTIONS

A function definition, also known as function implementation shall include the following elements:

1. Function name;
2. Function type;
3. List of parameters;
4. Local variable declarations;
5. Function statements; and
6. A return statement.

All the six elements are grouped into two parts, namely,

- function header (First three elements); and
- function body (second three elements).

A general format of a function definition to implement these two parts is given below:

```
function_type function_name(parameter list)
{
        local variable declaration;
        executable statement1;
        executable statement2;
        .  . . . .
        . . . . .
        return statement;
}
```

The first line function_type function_name(parameter list) is known as the function header and the statements within the opening and closing braces constitute the function body, which is a compound statement.

Function Header

The function header consists of three parts: the function type (also known as return type), the function name and the formal parameter list. Note that a semicolon is not used at the end of the function header.

Name and Type

The function type specifies the type of value (like float or double) that the function is expected to return to the program calling the function. If the return type is not explicitly specified, C will assume that it is an integer type. If the function is not returning anything, then we need to specify the return type as void. Remember, void is one of the fundamental data types in C. It is a good programming practice to code explicitly the return type, even when it is an integer. The value returned is the output produced by the functions.

The function name is any valid C identifier and therefore must follow the same rules of formation as other variable names in C. The name should be appropriate to the task performed by the function. However, care, must be exercised to avoid duplicating library routine names or operating system commands.

Formal Parameter List

The parameter list declares the variables that will receive the data sent by the calling program. They serve as input data to the function to carry out the specified task. Since they represent actual input values, they are often referred to as formal parameters. These parameters can also be used to send values to the calling programs.

Function Body

The function body contains the declarations and statements necessary for performing the required task.

The body enclosed in braces, contains three parts, in the order given below:

1. Local declarations that specify the variables needed by the function.
2. Function statements that perform the task of the function.
3. A return statement that returns the value evaluated by the function.

If a function does not return any value (like the printline function), we can omit the return statement. However, note that its return type should be specified as void. Again, it is nice to have a return statement even for void functions.

FUNCTION

Function is a self contained program performing a task. It has all qualities of a program. It is executed / called by main (or) another function, by calling the function name. It is also known as sub-program.

Syntax

```
return data type fn. name (optional arguments) → function heading

    {

        Function body

    }
```

function definition = function heading + fn. body)

CATEGORIES OF FUNCTIONS

1.No argument, No return values
2.No argument, with return values
3.With argument, No return values
4.With argument, with return values

1. No arguments, No return values

Eg : Area of square

```
    void main ( )
    {
        aos ( ) ;
    }
    void aos ( )
    {
        int a, s
        scan f (″ % d”, & s) ;
        a = s * s
        print f (″ a = % d” ; a) ;
    }
```

2. No arguments, with return values :

```
void main ( )
{
      int k;
      k = aos( ) ;
      print f (″ area = % d” , k) ;
}
            int aos ( )
      {
            int s, a ;
            scan f (″ % d” , & S) ;
            a = s * s;
            return (a) ;
      }
```

3. With argument No return type

```
Void main ( )
{
      int s ;
      print f (″ enter S value \ n”) ;
      scan f (″ % d” , &s) ;
            ┌──────►actual
      aos (s) ; argument
}
                         ┌──────► formal (or) dummy argument
void aos (int s1)
{
      int a ;
```

```
        a = s1*s1;
        print f (″ area = % d” , a) ;
    }
```

4. With argument with return type

```
    void main ( )
    {
        int s, k ;
        print f (″ enter s value \ n”);
        scanf (″ % d” , & s) ;
        k = aos (s) ;
        print f (″ area - % d” , k ) ;
    }
        int aos (int s1)
        {
            int a ;
            a = s1*s1;
            return (a) ;
        }
```

(eg) With argument ; with return value factorial

```
    void main ( )
    {
        int s ;long int k ;
        print f (″ enter & value \ n”) ;
        scan f (″ % d” , &s) ;
        k = fact (s) ;
        print f (″ factorial no = % ld” , k)
    }
     long int fact (int s1)
    {
        int i ; long int f = 1 ;
        for ( i =1 ; i < = n ; i + +)
        {
          f = f * i  ;
     }
          return (f) ;}
```

Recursion

A function calling itself is known as recursion. Recursion must have a terminating condition.

eg.

```
void main ( )
{
     int n ; long int k ;
     scan f (″ % d” , & n) ;
     k = fact (n) ;
     print f (″ factorial of % d is = % ld” ,n, k) ;
}
     long int fact (int nl)
     {
          long int f = 1;
          if (n 1< = 1) / * terminating condition * /
        return (1) ;
          else
          return (n1 * fact (n1 – 1)) ;
     }
```

note :

The default return data type of a function is int.

PASSING ARRAYS TO FUNCTIONS

One-Dimensional Arrays

It is sufficient to list the name of the array, without any subscripts, and the size of the array as arguments. For example, the call

```
                    largest(a,n)
```

will pass the whole array a to the called function.

```
main( )
{
        float largest(float a[ ], int n);
        float value[4] = {2.5, -4.75, 1.2,3.67};
        printf("%f\n", largest(value, 4)) ;
}
        float largest( (float a[]), int n)
{
        int ;
        float max;
        max = a[0] ;
        for(I = 1; 1 < n; i++)
                if(max < a[i])
        max = a[i] ;
  return(max) ;
}
```

Three Rules to Pass an Array to a Function

1. The function must be called by passing only the name of the array.
2. In the function definition, the formal parameter must be an array type; the size of the array does not need to be specified.
3. The function prototype must show that the argument is an array.

Two-Dimensional Arrays

Like simple arrays, we can also pass multi-dimensional arrays to functions. The rules are simple.

```
double average(iny x[] [N], int M, int N)
{
        int i, j;
        double sum = 0.0;
        for (i=0; i<M; i++)
                for(j=1; j<N; j++)
```

```
                        sum += x[i] [j] ;
                return(sum/(M*N)) ;
        }

        main( )
        {
                int M=3, N=2;
                double average(int [ ], [N], int, int);
                double mean;
                int mamtrix {M} [N] =
                        {
                                {1,2},
                                {3,4},
                                {5,6}
                        };
                mean = average(matrix, M, N) ;
                . . . . . .
                . . . . . .
                }
```

SCOPE, VISIBILITY AND LIFE TIME OF VARIABLES

1. Automatic variables.
2. External variables.
3. Static variables.
4. Register variables.

We shall briefly discuss the scope, visibility and longevity of each of the above class of variables. The scope of variable determines over what region of the program a variable is actually available for use (‘active’). Longevity refers to the period during which a variable retains a given value during execution of a program (‘alive’). So longevity has a direct effect on the utility of a given variable. The visibility refers to the accessibility of a variable from the memory.

The variables may also be broadly categorized, depending on the place of their declaration, as internal (local) or external

(global). Internal variables are those which are declared within a particular function, while external variables are declared outside or any function.

Automatic variables

Automatic variables are declared inside a function in which they are to be utilized. They are created when the function is called and destroyed automatically when the function is exited, hence the name automatic. Automatic variables are therefore private (or local) to the function in which they are declared. Because of this property, automatic variables are also referred to as local or internal variables.

```
    main( )

    {
            int number;
        -----
        -----
}
```

We may also use the keyword auto to declare automatic variables explicitly.

```
main( )
{
        auto int number;
        -----
        -----
}
```

Variables that are both alive and active throughout the entire program are known as external variables. They are also known as global variables. Unlike local variables, global variables can be accessed by any function in the program. External variables are declared outside a function. For example, the external declaration of integer number and float length might appear as.

```
int number;
float length = 7.5;
```

```
main( )
{
    ------
    ------
}
function1( )
}
     ------
     ------
}
function2( )
{
     ------
   ------
}
```

Static Variables

As the name suggests, the value of static variables persists until the end of the program. A variable can be declared static using the keyword static like

```
static int x;

static float y;
```

A static variable may be either an internal type or an external type depending on the place of declaration.

Internal static variables are those which are declared inside a function. The scope of internal static variables extend up to the function in which they are defined. Therefore, internal static variables are similar to auto variables, except that they remain in existence (alive) throughout the remainder of the program. Therefore, internal static variables can be used to retain values between function calls.

An external static variable is declared outside of all functions and is available to all the functions in that program. The difference between a static external variable and a simple external variable is that the static external variable is available only within the file where it is defined while the simple external variable can be accessed by other files.

Register variables

We can tell the compiler that a variable should be kept in one the machine's registers, instead of keeping in the memory (where normal variables are stored). Since a register access is much faster than a memory access, keeping the frequently accessed variables (eg., loop control variables) in the register will lead to faster execution of programs. This is done as follows:

register int count;

Storage class	**Where declared**	**Visibility (Active)**	**Lifetime (Alive)**
None	Before all functions in a file (may be initialized)	Entire file plus other files where variable is declared with extern	Entire program (Global)
extern	Before all functions in a file (cannot be initialized) extern and the file where originally declared as global.	Entire file plus other files where variable is declared	Global
static	Before all functions in a file	Only in that file	Global
None or auto	inside a function (or a block)	Only in that function or block	Until end of function of block
register	inside a function	Only in that function or block	Until end of function or block
static	Inside a function	Only in that function	Global

Just Remember

- It is a logic error if the parameters in the function call are placed in the wrong order.
- Using void as return type when the function is expected to return a value is an error.
- A return statement is required if the return type is anything other than void.
- Placing a semicolon at the end of header line is illegal.
- Defining a function within the body of another function is not allowed.

Review Questions

State whether the following statements are true or false.

a) C functions can return only one value under their function name.
b) A function in C should have at least one argument.
c) A function can be defined and placed before the main function
d) Only a void type function can have void as its argument.

Programming Exercises

Write a function exchange to interchange the values of two variables, say x and y. Illustrate the use of this function, in a calling function. Assume that x and y are defined as global variables.

The Fibonacci numbers are defined recursively as follows:

$$F1 – 1$$
$$F2 = 1$$
$$Fn = F_{n-1}+F_{n-2}, n > 2$$

PASSING STRUCTURE TO A FUNCTIONS

Program

```
/*      Passing a copy of the entire structure */

struct stores
{
        char    name[20] ;
        float   price ;
```

```
            int       quantity ;
            } ;
struct stores update (struct stores product, float p, int ) ;
float mul (struct store stock) ;
Main( )
{
floaot  p_increment, value;
int     q_increment ;
struct store item = {“XYZ”, 25.75,12} ;

printf (“\n Input increment values:”) ;
printf(“price increment and quantity increment\n”) ;
scan(“% %d”, &p_increment, &q_increment) ;

/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
     item = update(item, p_increment, q_increment) ;
/*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/
   printf(updated values of item\n\n”) ;
   printf(“Name          :  %s\n”,  item. name) ;
    printf(price          : %f\n, item. price);
    printf(Quantity       : %d\n”, item. quantity);
/*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -- - - - - - - - */
   value = mul (item)
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - --*/
printf(“nvalue of the item = %\n”’ value);
}
struct store update(struct stores product, float p, int q)
{
            product. Price += p;
            product.quantity += q;
            return(product);
}
float mul (struct stores stock)
```

```
}
```

Output

```
Input increment values: price increment and quantity increment
10 12
Updated values of item
Name        :XYZ
Price       :35.750000
Quantity    :24
Value of the item = 858.000000
```

STRUCTURES AND UNIONS

Structures

Structure is a collection of heterogenous data items. Each data item is known as structure member.

Syntax

```
                Struct tag name
                {
                        data type member 1 ;
                           ″             member 2 ;
                           ″      member n ;
                } ;
Template                (OR)
                Struct
                {
                        data type member 1 ;
                           ″       member 2 ;
                           ″         member n ;
                } 1, var 2, --- varn ;
```

Note :

tag name is optional struct is a keyword & it is must.

Var 1, Var 2 … varn are known as structure variables. Only through structure variables , structure members can be accessed.

Declaration :

When tagname is given & structure variables are not declared along with template, structure variables must be declared explicitly.

eg.

```
Struct  student
{
    char name [20] ;
    int regno ;
    int total ;
} ;
    struct student s1 ;
```

s1 is structure variable of struct type student. Each structure member is assigned individual memory location.

Accessing structure member.

Structure variable. Structure member.

```
eg.   scanf (″ %s %d%d” , s1. name, &s1. regno &s1. total);

print f (″ %s%d%d”, s1. name, s1. regno, s1. total) ;
```

Intializing structure members

Struct student S1 { “Joe”, 7572, 115}; structure members can be initialized according to the data types, while declaring the structure variable.

e.g.

```
Struct Student

{

    char name [20] ;

    int reg no;

    int sub1, sub2, sub3;

    int total ;

    float avg ;

} ;

    void main ( )

    {

        struct student s1 ;/*  explicit declaration  */

        scan f (″ %s%d%d%d%d”, s1.name, &s1. regno,

                                     &s1.sub1, &s1.sub2, &s1.sub3);

        s1.total =s1.sub1 +s1.sub 2 +s1.sub 3 ;
```

```
                s1.avg = s1.total /3 ;

                                                2)→S1

                print f (″ name =%s \n” name) ;

                print f (″ Register number = %d \n”,s1.regno) ;

                print f (″ subject 1 mark = %d\n”, s1.sub1) ;

print f (″ subject 2 mark = %d\n”, s1.sub2)

                print f (″ subject 3 mark = % d\n”,s1. sub3) ;

                print f (″ subject 3 mark = % d\n”, s1.total) ;

                print f (“ average = % f \n”, s1.avg) ;

          }
```

The above program accepts the name, register number, 3 subject marks of a student through keyboard calculates total & average, displays the same.

Array of structures

Array of structures can be declared and used.

eg.

To process the marks of students of a class.

```
    struct student

    {

        char name [20] ;

        int regno, m1, m2, m3 ;

        int total ; float avg ;
        } s1[24] ;
        void main ( )
        {
            int i, n ;
            print f (″ enter the no of students \n”);
            scanf (″ % d” &n);
            for (i = o; i < n;  i++)
                  scan f (″ %s%d%d%d%d”, s1[i]. name,
                          & s1[i]. regno, &s1[i]. m1, & s1[i]. m2, & s1[i]. m3) ;
            for (i = o ; i < n ; i ++)
            {
                  s1[i]. total = s1 [i] .m1 +s1[i]. m2 +s1 [i]. m3
                  s1[i].avg =s1[i]. total /3 ;
            }
            for ( i = o ; i < n; i ++)
            {
                  printf (″ name = %s \n”,  s1[i]. name) ;
                  printf (″ Register no = %d \n”,s1[i]. regno) ;
                  printf (″ mark 1 = % d\n”, s1[i]. m1) ;
                  print f (″ mark 2 = % d\n”, s1[i]. m2) ;
print f (″ mark 3= % d\n”, s1[i]. m3) ;
print f (″ Total marks = % d\n”, s1[i]. total) ;
            print f (″ Average = % f\n”, s1[i]. avg ) ;
        }
}
```

The above program accepts the input of name, register number and 3 subject marks, all the ‘n’ no. of students. Then total & average of ‘n’ no. of students are calculated & finally the marks of ‘n’ no. of students are displayed.

UNIONS

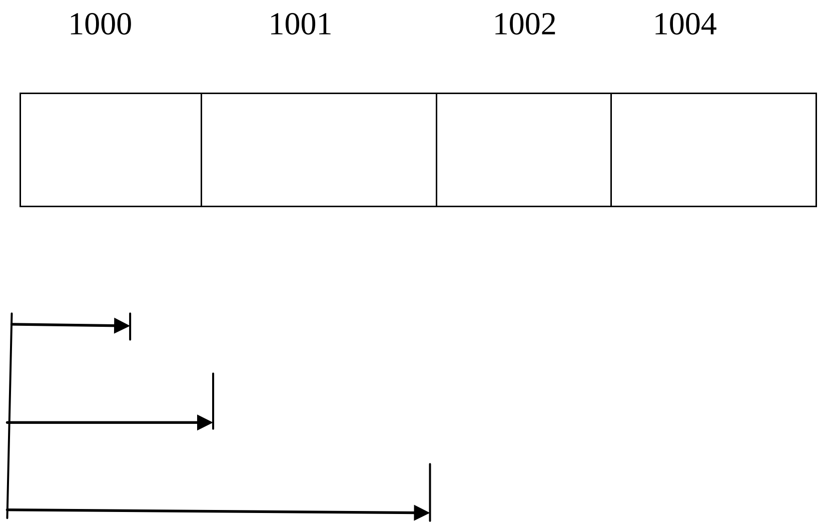
Unions follow the same syntax as structures. There is major distinction between them in terms of storage. In structures, each member has its own storage location, whereas all the members of a union is the same location. This implies that, although a union may contain many members of different types, it can handle only one member at t time.

A union can be declared using the keyword union as follows:

```
union item
{
      int m;
      float x;
      char c;
      code;
}     code;
```

This declares a variable code of type union item. The union contains three members, each with a different data type. However, we can use only one of them at a time.

Storage of 4 bytes



The compiler allocates a piece of storage that is large enough to hold the largest variable type in the union.

To access a union member union variable member eg.

```
code.m

code.x

code.c
```

Are all valid member variables. During accessing, we should make sure that we are accessing the member whose value is currently stored. For example, the statements such as

```
code.m = 379;

code.x = 7859.36;

printf("%d", code.m) ;
```

Would produce erroneous output(which is machine dependent.)

Union may be used in all places where a structure is allowed. Unions may be initialized when the variable is declared. But it can initialized only with a value of the type as the first union members.

union item abe = {100} ;

is valid.

Just Remember

- Remember to place a semicolon at the end of definition of structures and unions.
- Do not place the structure tag name after the closing brace in the definition. That will be treated as a structure variable. The tag name must be placed before the opening brace but after the keyword struct.
- When we use typed of definition, the type_name comes after the closing brace but before the semicolon.
- It is an error to use a structure variable as a member of its own struct type structure. It is an error to compare two structure variables. Use short and meaningful structure tag names.

Review Questions

State whether the following statements are true or false.

(a) A struct type in C is a built-in data type.
(b) The tag name of a structure is optional
(c) Structures may contain members of only one data type.
(d) The keyword typed of is used to define a new data type.
(e) An array cannot be used as a member of a structure.

Fill in the blanks in the following statements:

(a) The ……………. Can be used to create a synonym for a previously defined data type.
(b) A ………………… is a collection of data items under one name in which the items share the same storage.
(c) The name of a structure is referred to as ……………………..

Which of the following statements are legal?

(a) scanf (“%d, &a) ;

(b) printf (“%d”’ b) ;

(c) a = b;

(d) a = b + c;

UNIT V : POINTERS

***Pointers:-** Introduction – Understanding pointers – Accessing the Address of a Variable – Declaring Pointer Variables – Accessing a variable through its pointer – Pointer Expressions – Pointers as function arguments.*

***File Management in C:-** Introduction – Defining and Opening a file – Closing a File – Input/Output Operations on files – Error Handling During I/O Operating.*

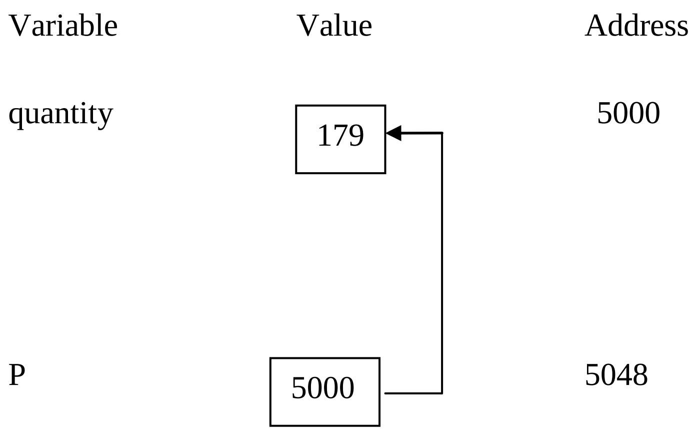
POINTERS

Introduction

Pointer is derived data type in C pointers contain memory addresses as their values. Pointers can be used to access and manipulate data stored in the memory. Whenever we declare a variable, the system allocates, somewhere in the memory, an appropriate location to hold the value of the variables. Consider the following statement

```
int quantity = 179;
```

This statement instructs the system of find a location for the integer variable quantity and puts the value 179 in that location. Let us assume that the system has chosen the address location 5000 for quantity. During execution of the program, the system always associates the name quantity with the address 5000. Since memory addresses are simply numbers, they can be assigned to some variables that can be stored in memory, like any other variable. Such variables that hold memory addresses are called pointer variables. A pointer variable is, therefore, nothing but a variable that contains an address, which is a location of another variable in memory.



The variable that contains a pointer value is called a pointer variable. The operator & immediately preceding a variable returns the address of the variable associated with it. For example, the statement

```
p = &quantity;
```

would assign the address 5000 (the location of quantity) to the variable p. The & operator can be remembered as 'address o'.

In C, every variable must be declared for its must be declared for its type. The declaration of a pointer variable takes the following form:

```
data_type *pt_name;
```

This tells the compiler three things about the variable pt_name.

1. The asterisk (*) tells that the variable pt_name is a pointer variable.
2. pt_name needs a memory location.
3. pt_name points to a variable of type data_type.

For example,

```
int *p; /* integer pointer */

float *x; / * float pointer */
```

Once a pointer has been assigned the address of a variable, the question remains as to how to access the value of the variable using the pointer? This is one by using another unary operator * (asterisk), usually known as the indirection operator. Another name for the indirection operator is the dereferencing operator. Consider the following statements:

```
int quantity, *p, m;

quantity = 179;

p = &quantity;

n = *p;
```

The fourth line contains the indirection operator*. When the operator* is placed before a pointer variable in an expression (on the right-hand side of the equal sign), the pointer returns the value of the variable of which the pointer value is the address.

Like other variables, pointer variables can be used in expressions, For example, if p1 and p2 are properly declared and initialized pointers, then the following statements are valid.

```
y = *p1 * *p2;            same as (*p1) * (*p2)

sum = sum + *p1;

z = 5* - *p2/ *p1;        same as (5 * (- (*p2)))/(*p1)

*p2 = *p2 + 10;
```

POINTERS AS FUNCTION ARGUMENTS

When we pass addresses to a function, the parameters receiving the addresses should be pointers. The process of calling a function using pointers to pass the addresses of variables is known as ‘call by reference’

The function exchange() receives the addresses of the variables x and y and exchanges their contents.

Program

```
void exchange (int *, int *);    /* prototype */

Main()

{

        int x,y;

        x = 100;

        y = 200;

        printf (Before exchange     : x = %d     y = %d\n\n”, x,x);
        exchange(&x,&y);    /* call */

        printf("After exchange       : x = %d       y = %d\n\n, x,y);

}

exchange (int *a, int *b)

{

        int t;

        t = *a;         /*Assign the value at address a to t */

        *a = *b;        /* put b into a */

        *b = t;         /* put t into b */

        }
```

Output

```
        Before exchange: x = 100     y = 200

        After exchange: x = 200      y = 100
```

Just Remember

- Only an address of a variable can be stored in a pointer variable.
- Do not store the address if a variable of one type into a pointer variable of another type.
- The value of a variable cannot be assigned to a pointer variable.
- When we pass a parameter by address, the corresponding formal parameter must be a pointer variable.
- When an array is passed as an argument to a function, a pointer is actually passed. In the header function, we must declare such arrays with proper size, except the first, which is optional.

Review Questions

State whether the following statements are true or false.

(a) Pointer constants are the addresses of memory locations.
(b) Pointer variables are declared using the address operator.
(c) When an array is passed as an argument to a function, a pointer is passed.
(d) Pointers cannot be used as formal parameters in headers to function definitions.

FILE MANAGEMENT IN C

INTRODUCTION

A file is a place on the disk where a group of related data is stored.Basic file operations.

- Naming a file,
- Opening a file,
- Reading data from a file,
- Writing data to a file, and
- Closing a file.

DEFINING AND OPENING A FILE

To store data in a file in the secondary memory, we must specify certain things about the file, to the operating system. They include:

High Level I/O Functions

Function name	Operation
fopen()	❖ Creates a new file for use.
fclose()	❖ Opens an existing file for use.
getc()	❖ Close a file which has been opened for use.
putc()	❖ Reads a character from a file.
fprintf()	❖ Writes a character to a file.

getw()	❖ Writes a set of data values to a file.
putw()	❖ Reads a set of data values from a file.

1. Filename.
2. Date structure.
3. Purpose.

Filename is a string of characters that make up a valid filename for the operating system. It contain two parts primary name an optional period with the extension.

Examples:

input.data

Student.c

Data structure of a file is defined as FILE in the library of standard I/O function definitions. FILE is a defined data type.

General format declaring and opening a file:

```
FILE *FP;

fp = fopen("filename', "mode") ;
```

The first statement declares the variable fp as a "pointer to the data type FILE", The second statement opens the file named filename and assigns an identifier to the FILE type pointer fp. This pointer, which contains all the information about the file is subsequently used as a communication link between the system and the program. Mode can be one of the following:

r	open the file for reading only.
w	open the file for writing only.
a	open the file for appending (or adding) data to it.

```
FILE *p1, *p2;
```

p1 = fopen(“data”, “r”);

p2 = fopen(“results”, “w”);

Additional modes of operation

r+	The existing file is opened to the beginning for both reading and writing
w+	Same as w except both for reading and writing.
a+	Same as a except both for reading and writing.

CLOSING A FILE

A file must be closed as soon as all operations on it have been completer.

fclose(file_pointer);

FILE *P_1, *P_2;

p1 = FOPEN(“INPUT”, “w”);

p2 = fopen(“OUTPUT”, “r”);

fclose(p1);

fclose(p2);

The getc and putc Functions

The simplest file I/O functions are getc and putc. These are analogous to getchar and putchar functions and handle one character at a time. Assume that a file is opened with mode w and file pointer fp1. Then, the statement

putc(c, fp1);

Writes the character contained in the character variable c to the file associated with FILE pointer fp1. Similarly, getc is used to read a character from a file that has been opened in read mode. For example, the statement

c = getc(fp2);

Would read character from the file whose file pointer is fp2.

The file pointer moves by one character position for every operation of getc or putc. The getc will return an end-of-file marker EOF, when end of the file has been reached. Therefore, the reading should be terminated when EOF is encountered.

The getw and putw Functions

The getw and putw are integner-oriented functions. They are similar to the getc and putc functions and are used to read and write integer values. These functions would be useful when we deal with only integer data.a The general forms of getw and putw are:

```
putw(integer,fp);

getw(fp) ;
```

WRITING & READING A FILE

The fprintf and fscanf Functions

The functions fprintf and fscanf perform I/O operations on files. The first argument of these functions is a file pointer which specifies the file to be used.

The general form of fprintf is

```
fscanf(fp, “control string”, list);
```

This statement would cause the reading of the items in the list from the file specified by fp, according to the specifications contained in the control string.

The general form of fprintf is

```
fprintf(fp, “control string”, list);
```

This statement would cause the writing of the items in the list from the file specified by fp, according to the specifications contained in the control string.

Write a program to open a file named inventory and store in it the following data:

Item name	Number	Price	Quantity
AAA-1	111	17.50	115
BBB-2	125	36.00	7
C-3	247	31.75	104

Extend the program to read this data from the file INVENTORY and display as the inventory table with the value of each item.

Program

```
#include <stdio.h>



main( )
{
        File *fp;
        int number, quantity, I;
        float   price,   value;
        char    item[10],       filename[10];

        printf(“Input file name\n);
        scanf(“5s”, filename);
        fp = fopen(filename, “w”);
        printf(:Inpaut inventory data\nn”);
        printf(“Item name Bumber price Quantity\n);
        for(i = 1; i <= 3; i++)
{
        fscanf(stdin,, “%e  %d %f %d,)
        item, & number, &price, &quantity;
        fprintf(fp, “%s %d %.2f %d”,
        item,   number,         price,   quanatity);
}
fclose(fp);
fprintf(stdout, “\n\n”);
fp = fopen(filename, “r”);
printf(“item name Number Price Quantity Value\n”);
for(I = 1; I <++)
{
        fscan(fp, “%s %d %f d”, item,&price,&quantity);
        value = price * quantity;
        fprintf(stdout, “%-8s %7d %8.2f %8d %11.2f\n”,
        item, number, price, quantity, values);
        }
        fclose (fp);
}
```

ERROR HANDLING DURING I/O OPERATIONS

It is possible that an error may occur during I/O operations on a file. Typical error situations include:

1. Trying to read beyond the end-of-file mark.
2. Device overflow.
3. Trying to use a file that has not been opened.
4. Trying to perform an operation on a file, when the file is opened for another type of operation.
5. Opening a file with an invalid filename.
6. Attempting to write to a write-protected file.

We have two status-inquiry library functions; feof and ferrror that can help us detect I/O errors in the files. The feof function can be used to test for an end of file condition. It takes a FILE pointer as its only argument and returns a nonzero integer value if all of the data from the specified file that has just been opened for reading, then the statement

```
if(feof(fp))

printf(“End of data.\n”);
```

The ferror function reports the status of the file indicated. It also takes a FILE pointer as its argument and returns a nonzero integer if an error has been detected up to that point, during processing. It returns zero otherwise. The statement

```
                                        if(ferror(fp) != 0

                                        printf(“An error has occurred.\n”))
```

Would print the error message, if the reading is not successful.

PROCESSING A DATA FILE

Program

```
          #include <stdio.h>

          struct invent_record

          {

               char name[10];

               int number;

               flooat price;

               int quantity;

          };

          main ( )

{

               struct invent_recorde item;

               char filename[10];

               int reponse;

               long n;
```

```
            void append (struct invent_record *x, file *y);
            printf("Type filename:");
            scanf("%s", filename);
            fp = fopen(filename, "a+");
            do
{
            append(&item, fp);
            printf("\nItem %s appended.\n" item.name);
            printf("\nDo you want to and another item\
            (1 for YES /0 for No)?");
            scanf("%d", &response);
}           while (response ++ 1);
n = ftell(fp);              /* position of last character     */
fclose(fp);
while(ftell (fp) < n);
{
            fscanf(fp, "%s %d %f %d,
            item.name, &item. Number, &item.price, &item.quantity);
            fprintf(stdout,"%-8s %7d %8.2f %8d\n",
            item.name, item.number, item.price, item.quantity);
}
```

```
        fclose(fp);
    }
    void append(struct invent_record *product, File *ptr)
    {
            printf("Item name:")
            scanf(""%s", product->name);
            printf("Item number:");
            scanf("%d", &product->number);
            printf("Item price:");
            scanf("%f", &product->quantityt);
            fprintf(ptr, "%s %d %.2f %d",
            product->name,
            product->number,
            product->price,
            product->quantity);
```

Output

```
    Type filename: INVENTORY
    Item name:XXX
    Item number:444
    Item price: 40.50
    Quantity: 34
```

```
Item XXX appended.

Do you want to add anotheritem()1 for YES /0 for No?1

Item name: YYY

Item number: 555

Item price: 50.50

Quanitity:45

Item YYY appended.

Do you want to add another item(item(1 for YES /0 for No?0)

AAA-1          111          17.50      115

BBB-2          125          36.00      7

C-3            247          31.75      104

XXX            444          40.50      34

YYY            555          50.50      45
```

What is a command line argument? it is a parameter supplied to a program when the program is invoked. This parameter may represent a filename the program should process. For example, if we want to execute a program at copy the contents of a file named X_FILE to another one named to another one named Y_FILE, then we may use a command line like

Just Remember

- Do not try to use a file before opening it.
- EOF is integer type with a value -1. Therefore, we must use an integer variable to test EOF.
- It is an error to open a file for reading when it does not exist.
- It is an error to attempt to place the file marker before the first byte of a file.

Review Questions

State whether the following statements are true or false.

(a) A file must be opened before it can be used.
(b) All files must be explicitly closed.
(c) Files are always referred to by name in C programs.
(d) Using fseek to position a file beyond the end of the file is an error.
(e) Function fseek may be used to seek from the beginning of the file only.

Fill in the blanks in the following statements.

(a) The mode ………………….. is used for opening a file for updating
(b) The function ……………… may be used to position a file at the beginning
(c) The function ……………….. gives the current position in the file.
(d) The function……………… is used to write data to randomly accessed file.

What is the significance of EOF?

Programming Exercises

Write a program to copy the contents of one file into another.

Write a program to create a sequential file that could store details about five products. Details include product code, cost and number of items available and are provided through keyboard.

Prepared by

Dr. V. JOSEPH PETER

Associate Professor of Computer Science

Kamaraj College, Thoothukudi.